

Digitale Elektronica en Processoren

Luc Van Eycken

Luc.VanEycken@esat.kuleuven.be

Inleiding

⇒ Doelstellingen

- Voorkennis
- Praktische informatie

Digitale elektronica?

= schakelingen die de kern vormen van uiteenlopende toepassingen:

- Krachtige computersystemen
- Multimedia, spelen
- Draagbare telecommunicatie
- Intelligente kledij
- Huishoudapparatuur
- ...



Wat is belangrijk?

- Hoge verwerkingskracht (voor ingewikkelde algoritmen)
- Zeer laag vermogenverbruik
- Zeer compacte implementatie
- Goedkoop (voor massaproducten)

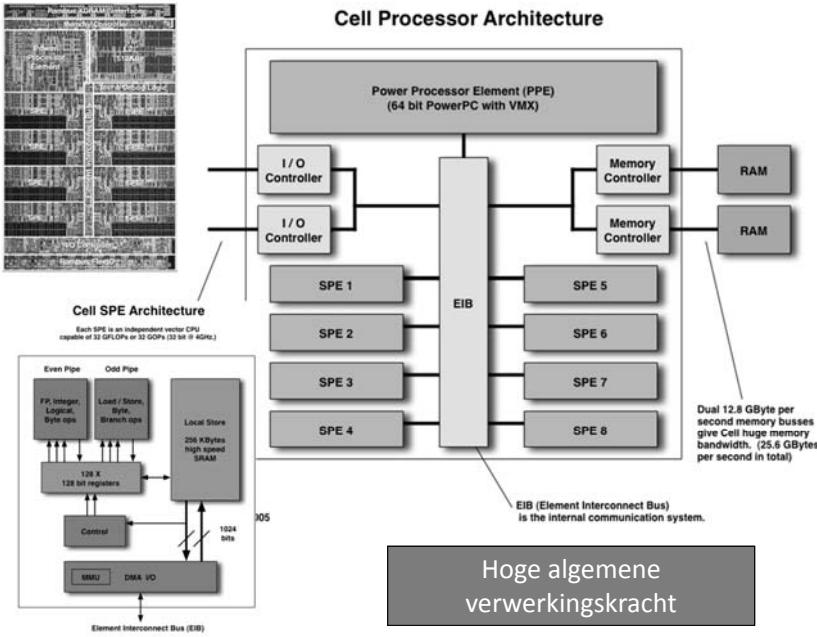


Inleiding

⇒ Doelstellingen

- Voorkennis
- Praktische informatie

Voorbeeld implementatie: de Cell-processor

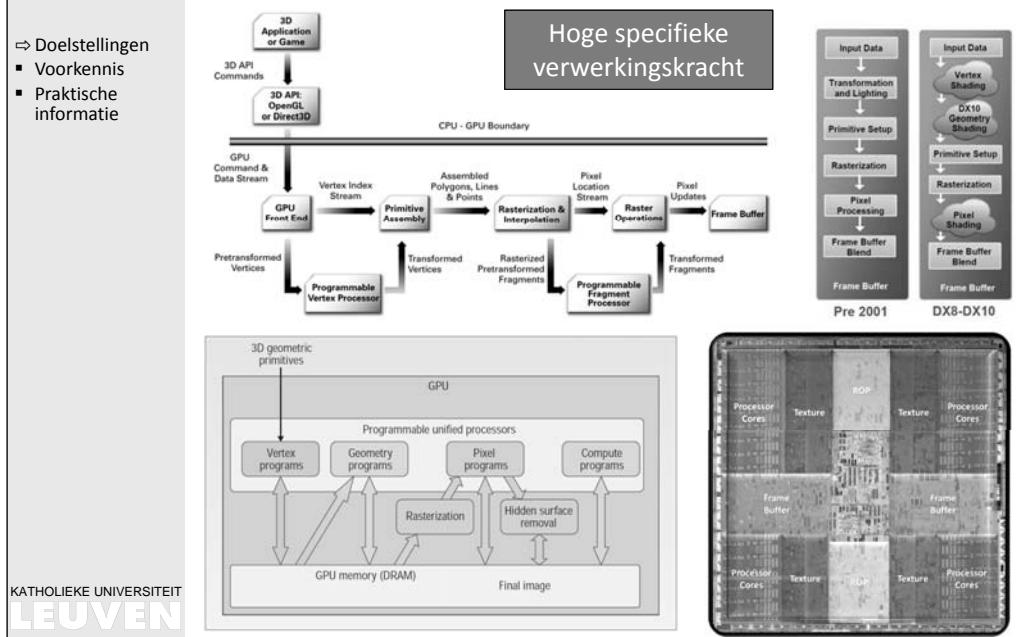


Inleiding

⇒ Doelstellingen

- Voorkennis
- Praktische informatie

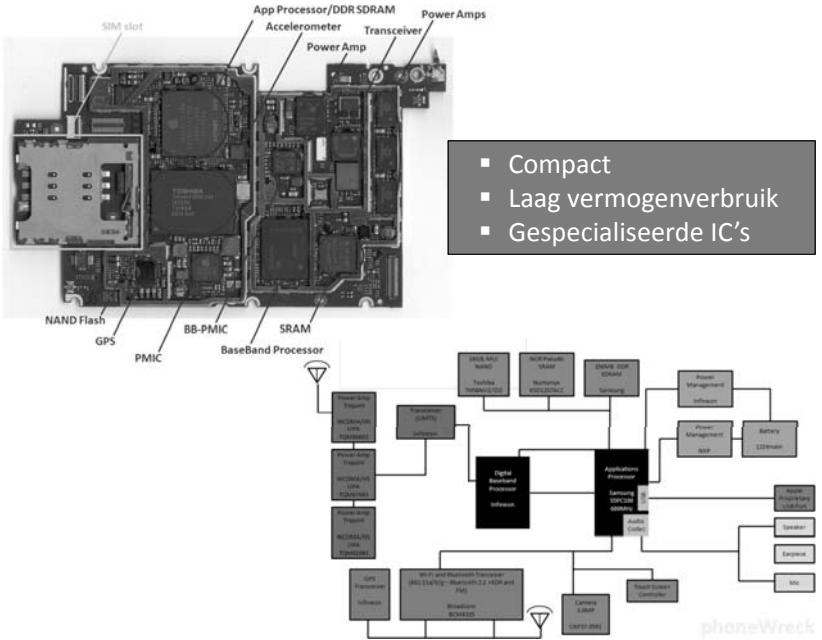
Voorbeeld implementatie: een GPU



Inleiding

- ⇒ Doelstellingen
- Voorkennis
 - Praktische informatie

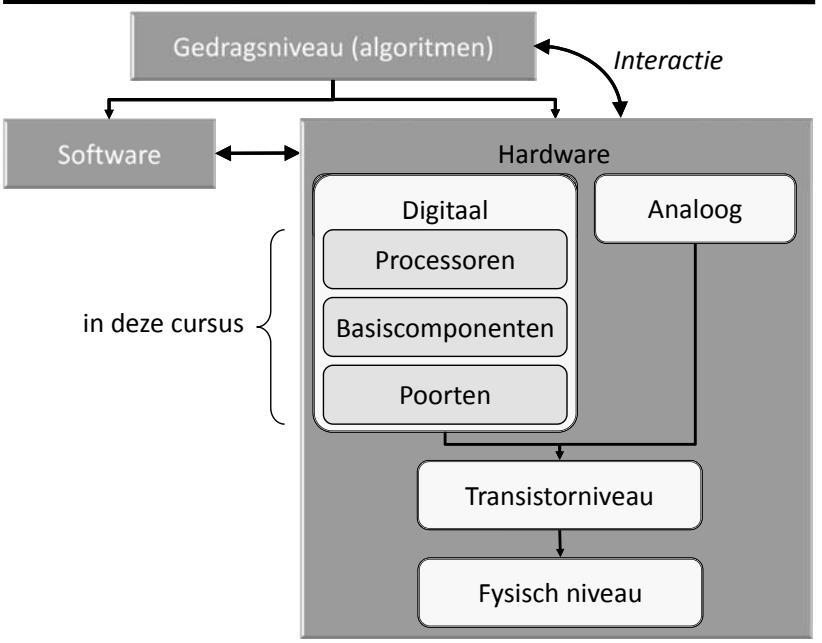
Voorbeeld implementatie: een ‘smartphone’



Inleiding

- ⇒ Doelstellingen
- Voorkennis
 - Praktische informatie

Hoe ontwerpen we dit?

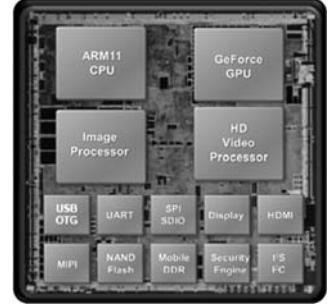


Inleiding

- ⇒ Doelstellingen
- Voorkennis
 - Praktische informatie

Voorbeeld implementatie: een ‘System on Chip’

- = combinatie (op een chip) van één of meerdere
 - programmeerbare processoren
 - ⇒ zeer flexibel
 - ⇒ goedkoop qua ontwerp
 - niet-programmeerbare processoren
 - ⇒ hoge verwerkingskracht
 - ⇒ compacte implementatie
 - ⇒ goedkoop qua componenten & vermogen
 - ⇒ kan (beperkt) reconfigurerbaar zijn in sommige implementaties
 - controle-eenheden om alles aan te sturen



Inleiding

- ⇒ Doelstellingen
- Voorkennis
 - Praktische informatie

Wat leren we in deze cursus?

- Inzicht verwerven in het ontwerp van digitale elektronische systemen op poort- en RTL-niveau
- Alle bouwblokken (inclusief processoren) leren kennen die nodig zijn om complexe digitale schakelingen te bouwen
- De basisconcepten van programmeertalen voor de beschrijving en ontwerp van digitale hardware (zoals VHDL) leren kennen
- Ervaring opdoen met moderne ontwerpomgevingen (voor FPGA)

Inleiding

- ⇒ Doelstellingen
- Voorkennis
- Praktische informatie

Inhoudstafel

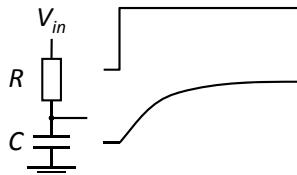
1. Inleiding
2. De basis van digitaal ontwerp
3. Technologische randvoorwaarden
4. Combinatorische schakelingen:
geheugenloze schakelingen
5. Sequentiële schakelingen:
schakelingen met geheugen
6. Niet-programmeerbare processoren
7. Programmeerbare processoren

Inleiding

- Doelstellingen
- ⇒ Voorkennis
- Praktische informatie

Vereiste voorkennis

- Elektrische grootheden
 - spanning, stroom, vermogen ($P = V \cdot I$)
- Eenvoudige elektrische netwerken
 - componenten:
 - weerstand ($V = R \cdot I$)
 - capaciteit ($I = C \cdot dV/dt$)
 - wet van Ohm
- Werking transistor
 - zie eventueel boek hoofdstuk 3.8
- Basis programmeertalen



Inleiding

- Doelstellingen
- ⇒ Voorkennis
- Praktische informatie

Inleiding

- Doelstellingen van de cursus
- ➔ Vereiste voorkennis
- Praktische informatie

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- Examen

Inleiding

- Doelstellingen van de cursus
- Vereiste voorkennis
- ➔ Praktische informatie
 - Cursusmateriaal
 - Oefeningen & labo's
 - Examen

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- Examen

Cursusmateriaal

- Verplicht materiaal
 - transparanten (ook beschikbaar via Toledo)
- Sterk aangeraden materiaal
 - *Fundamentals of Digital Logic with VHDL Design*,
Stephen Brown & Zvonko Vranesic,
McGraw-Hill, 2009, ISBN 0-07-352953-2
- Topics uit andere boeken:
 - *Principles of Digital Design*, Daniel D. Gajski,
Prentice Hall, 1997, ISBN 0-13-301144-5
- Andere referentiewerken:
 - *Digital Design: principles & practices*, John Wakerly
 - *The Designer's Guide to VHDL*, Peter J. Ashenden

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- ⇒ Examen

Examen

- Mondeling gesloten boek met schriftelijke voorbereiding
- Drie soorten vragen:
 - 1) Vertaling van een algoritme naar een hardware blokschema
 - 2) Ontwerp van een FSM
 - 3) Theorievragen
- Voorbeelden van vragen zijn te vinden verderop en op Toledo
- Aan het einde van het semester is een extra les als vragenuurtje voorzien

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- Examen

Oefeningen & labo's

- *Traditionele oefeningen* over
 - ontwerp combinatorische schakelingen
 - ontwerp synchrone sequentiële schakelingen
 - ontwerp niet-programmeerbare processoren
 - begrijpen VHDL-beschrijvingen van hardware
- *Volledig hardware-ontwerp* gebruik makend van Xilinx FPGA-hardware en ontwikkelomgeving
 - Vertaling van een probleem in schema's
 - Simulatie van het ontwerp
 - Uitvoering op de FPGA-hardware

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- ⇒ Examen

Vertaling van een algoritme naar een hardware blokschema

Ontwerp een FSMD die volgende functie uitvoert:

```
entity fsmd is
  port(i: in integer range 0 to 255; clk, start: in bit;
       o: out integer range 0 to 4095);
end entity fsmd;
architecture behav of fsmd is begin
  process is
    variable a,b,c: integer;
  begin
    wait until clk = '1'; if start = '0' then o <= 0; else
      a := i; wait until clk = '1';
      b := i; wait until clk = '1';
      c := i; wait until clk = '1';
      while abs(b-c) <= abs(a-b) loop
        o <= 5 * abs(a-b);
        c := i; wait until clk = '1';
      end loop;
      o <= 3 * c + 1;
    end if;
  end process;
end architecture behav;
```

Ontwerp het datapad tot op RTL componenten en de controller tot op FSM-niveau.

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- ⇒ Examen

Ontwerp van een FSM

Maak de goedkoopst mogelijke IC-realisatie van volgende FSM met JK-flipflops en NAND-poorten:

| | 00 | 01 | 11 | 10 |
|----|-------|-------|-------|-------|
| S0 | S0/10 | S0/00 | S7/00 | S2/01 |
| S1 | S0/11 | S4/11 | S4/11 | S5/11 |
| S2 | S1/01 | S2/00 | S2/10 | S7/10 |
| S3 | S0/10 | S3/00 | S1/00 | S2/01 |
| S4 | S5/00 | S5/11 | S2/01 | S7/10 |
| S5 | S5/00 | S4/11 | S6/01 | S1/10 |
| S6 | S7/01 | S2/00 | S2/10 | S7/10 |
| S7 | S0/11 | S4/11 | S5/11 | S4/11 |

Inleiding

- Doelstellingen
- Voorkennis
- ⇒ Praktische informatie
- ⇒ Examen

Enkele theorievragen

- Beschrijf het IEEE-formaat voor getallen met “enkelvoudige precisie vlootende komma voorstelling”. Geef ook aan welke getallen hiermee kunnen voorgesteld worden.
- Geef de realisatie en bespreek de werking van een “prioriteitsencoder”.
- Wat zijn de verschillende stappen in het ontwerp van een CISC-computer?
- Wat zijn de belangrijkste voordelen van het gebruik van VHDL?



Inhoudstafel

- Inleiding
- De basis van digitaal ontwerp
- Technologische randvoorwaarden
- Combinatorische schakelingen
- Sequentiële schakelingen
- Niet-programmeerbare processoren
- Programmeerbare processoren

Digitale schakelingen

- ⇒ Logische schakelingen
- wat?
 - realisatie
 - Boole-algebra
 - Synthese
 - Ontwerppad
 - VHDL

Digitale schakeling

- Discrete signalen
(beperkt # waarden)

Analoge schakeling

- Continue signalen

- Praktische digitale schakelingen werken (bijna uitsluitend) met binaire signalen (bits)
 - 2 waarden van spanning, stroom, druk, reflectie, ...
 - Deze waarden worden symbolisch voorgesteld als "0" en "1"
 - Gebruik combinaties van meerdere bits om niet-binaire waarden weer te geven

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Digitaal ontwerp

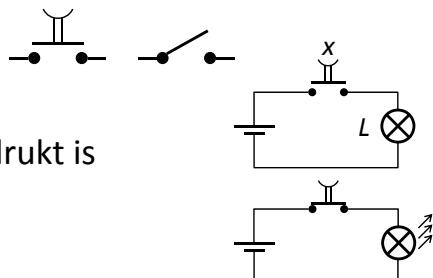
- Logische schakelingen
- Booleaanse algebra
- Synthese met logische poorten
- Digitaal ontwerp in grote lijnen
- Taalgebaseerd hardware ontwerp: VHDL

- ⇒ Logische schakelingen
- ⇒ wat?
- realisatie
 - Boole-algebra
 - Synthese
 - Ontwerppad
 - VHDL

Logische schakelingen in huis

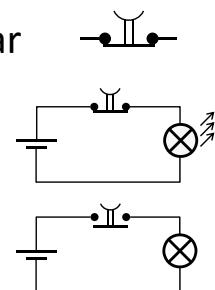
Lichtschakelaar

- Lamp L brandt als schakelaar x ingedrukt is
 - $x = 0 \Rightarrow L = 0$
 - $x = 1 \Rightarrow L = 1$



Complementaire lichtschakelaar

- L brandt als x niet ingedrukt is
 - $x = 0 \Rightarrow L = 1$
 - $x = 1 \Rightarrow L = 0$



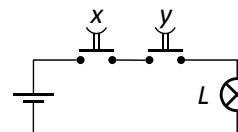
- ⇒ Logische schakelingen
- ⇒ wat?
- realisatie
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Logische schakelingen in huis

□ EN-schakeling

➢ $L = x \cdot y = x \text{ AND } y$

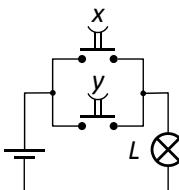
➢ $L = 1$ als $x = 1$ en $y = 1$



□ OF-schakeling

➢ $L = x + y = x \text{ OR } y$

➢ $L = 1$ als $x = 1$ of $y = 1$

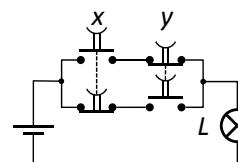


□ Exclusieve-OF-schakeling

➢ $L = x \oplus y = x \text{ XOR } y$

➢ $L = 1$ als ofwel $x = 1$ ofwel $y = 1$, maar niet beide

➢ $L = (x \cdot y') + (x' \cdot y)$



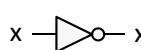
- ⇒ Logische schakelingen
- wat?
- ⇒ realisatie
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Logische poort

= (elektronische) schakeling die een logische basisfunctie realiseert

□ Basispoorten: realiseren basisoperaties

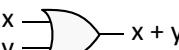
inverter



AND-poort



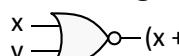
OR-poort



➢ Vereist voor realiseren logische schakelingen

□ Complexe poorten

➢ Kunnen gerealiseerd worden met basispoorten, maar in sommige technologieën eenvoudiger

➢ bijv. XOR  of NOR 

- ⇒ Logische schakelingen
- ⇒ wat?
- realisatie
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Waarheidstabellen

= tabel met de functiewaarden voor alle combinaties van ingangswaarden

➢ n (ingangs)variabelen
⇒ 2^n rijen

➢ eenduidige beschrijving van een functie:
equivalente functies ⇔ zelfde waarheidstabellen

| x | x' |
|---|----|
| 0 | 1 |
| 1 | 0 |

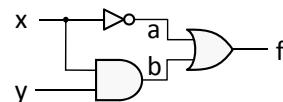
| x | y | x · y | x + y |
|---|---|-------|-------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- ⇒ Logische schakelingen
- wat?
- ⇒ realisatie
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Logische schakeling

= netwerk van poorten, beschreven door

➢ een schema (visueel)



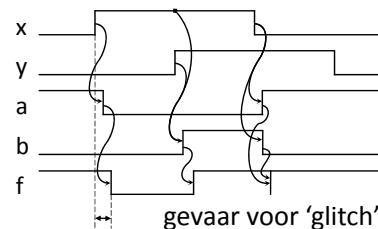
| x | y | a | b | f |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |

➢ een hardwaretaal (als programma): VHDL

□ Tijdsgedrag

➢ vertragingen

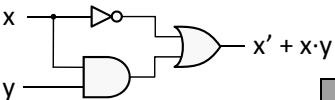
➢ problemen eigen aan implementatie



- ⇒ Logische schakelingen
 - wat?
 - ⇒ realisatie
- Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Logische schakeling

□ Wat is de beste realisatie?



| x | y | x' | $x \cdot y$ | $x' + x \cdot y$ | $x' + y$ |
|---|---|------|-------------|------------------|----------|
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

➤ Beide realisaties zijn equivalent:

➤ Baseer de keuze op

- minimale kostprijs
(# poortingangen + # poortuitgangen
– # inverters)
- maximale verwerkingskracht
⇒ minimale vertraging (\propto # ingangen)

- Logische schakelingen
- ⇒ Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Booleaanse algebra: definitie

= algebra van logische waarden {0; 1} met 3 logische operatoren: \wedge (\cdot), \vee (+), \neg ('')

➤ prioriteit: haakjes > \neg > \wedge > \vee

□ Definitie via axioma's

- | | |
|--------------------------------|----------------------------|
| 1) $0 \cdot 0 = 0$ | $1 + 1 = 1$ |
| 2) $1 \cdot 1 = 1$ | $0 + 0 = 0$ |
| 3) $0 \cdot 1 = 1 \cdot 0 = 0$ | $1 + 0 = 0 + 1 = 1$ |
| 4) $x = 0 \Rightarrow x' = 1$ | $x = 1 \Rightarrow x' = 0$ |

□ Duale uitdrukking ook steeds geldig

- vervang "0" door "1" en vice-versa
- vervang "+" door " \cdot " en vice-versa

- Logische schakelingen
- ⇒ Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Digitaal ontwerp

□ Logische schakelingen

→ Booleaanse algebra

□ Synthese met logische poorten

□ Digitaal ontwerp in grote lijnen

□ Taalgebaseerd hardware ontwerp: VHDL

- Logische schakelingen
- ⇒ Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Theorema's & eigenschappen

Regels voor 1 variabele x

- | | |
|---------------------|--------------|
| 5) $x \cdot 0 = 0$ | $x + 1 = 1$ |
| 6) $x \cdot 1 = x$ | $x + 0 = x$ |
| 7) $x \cdot x = x$ | $x + x = x$ |
| 8) $x \cdot x' = 0$ | $x + x' = 1$ |
| 9) $(x')' = x$ | |

□ Te bewijzen via waarheidstabellen met axioma's

□ Uit regel 6 volgen de eenheidselementen

- Logische schakelingen
- ⇒ Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Theorema's & eigenschappen

Regels voor meerdere variabelen

10) Commutativiteit

$$x \cdot y = y \cdot x \quad x + y = y + x$$

11) Associativiteit

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \quad x + (y + z) = (x + y) + z$$

12) Distributiviteit

$$\begin{aligned} \text{t.o.v. } + : \quad &x \cdot (y + z) = x \cdot y + x \cdot z \\ \text{t.o.v. } \cdot : \quad &x + (y \cdot z) = (x + y) \cdot (x + z) \end{aligned}$$

13) Absorptie

$$x + x \cdot y = x \quad x \cdot (x + y) = x$$

15) Wet van De Morgan

$$(x \cdot y)' = x' + y' \quad (x + y)' = x' \cdot y'$$

□ Ook bewijsbaar via algebraïsche manipulatie

$$\triangleright x + x \cdot y \stackrel{6}{=} x \cdot 1 + x \cdot y \stackrel{12}{=} x \cdot (1 + y) \stackrel{5}{=} x \cdot 1 \stackrel{6}{=} x$$

□ xy is een verkorte schrijfwijze van $x \cdot y$

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Digitaal ontwerp

□ Logische schakelingen

□ Booleaanse algebra

→ Synthese met logische poorten

➤ SOP & POS

➤ canonieke versus standaard realisatie

➤ realisaties met NAND & NOR

□ Digitaal ontwerp in grote lijnen

□ Taalgebaseerd hardware ontwerp: VHDL

- Logische schakelingen
- ⇒ Boole-algebra
- Synthese
- Ontwerppad
- VHDL

Verschil met gewone algebra

□ In Booleaanse algebra bestaat geen inverse bewerking voor de optelling (+) of de vermenigvuldiging (·)
⇒ aftrekking of deling bestaan niet

□ In gewone algebra kan geen y' gedefinieerd worden voor elke y zodat $y + y' = 1$ en $y \times y' = 0$

□ In gewone algebra is + niet distributief t.o.v. \times : $5 + (2 \times 4) \neq (5 + 2) \times (5 + 4)$

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Synthese

= realisatie van logische functionaliteit in een schakeling

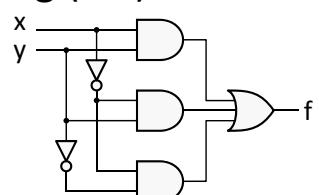
| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

□ functionaliteit
≡ waarheidstabel

$$\triangleright f = x' + xy$$

□ Rechtstreekse implementatie:
functie is de verzameling (OR) van alle rijen (AND) waarvan het resultaat 1 is

$$\triangleright f = x'y' + x'y + xy$$



- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Synthese

- Verdere minimalisering door algebraïsche manipulatie:

$$f = x'y' + x'y + xy$$

$$= x'y' + x'y + x'y + xy$$

regel 7

$$= x'(y' + y) + (x' + x)y$$

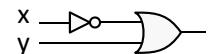
distributiviteit

$$= x' \cdot 1 + 1 \cdot y$$

regel 8

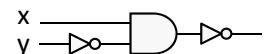
$$= x' + y$$

regel 6



$$= (xy)'$$

De Morgan



Niet evident welke combinatie van theorema's en eigenschappen hiervoor nodig zijn!

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

“Sum-of-Products” (SOP)

- = logische uitdrukking waarbij producttermen (AND) gesommeerd (OR) worden

$$\triangleright f = y'z + xz'$$

- Elke logische functie kan beschreven worden als de som van zijn 1-mintermen

(= canonieke SOP)

$$\triangleright f = x'y'z + xy'z'$$

$$+ xy'z + xyz'$$

$$= m_1 + m_4 + m_5 + m_6$$

$$= \sum m(1,4,5,6)$$

| x | y | z | f | 1-minterm |
|---|---|---|---|---------------|
| 0 | 0 | 0 | 0 | — |
| 0 | 0 | 1 | 1 | $m_1 = x'y'z$ |
| 0 | 1 | 0 | 0 | — |
| 0 | 1 | 1 | 0 | — |
| 1 | 0 | 0 | 1 | $m_4 = xy'z'$ |
| 1 | 0 | 1 | 1 | $m_5 = xy'z$ |
| 1 | 1 | 0 | 1 | $m_6 = xyz'$ |
| 1 | 1 | 1 | 0 | — |

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Mintermen

- Een minterm is een logische functie die waar is voor slechts één enkele rij van de waarheidstabel

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

- Een 1-minterm is een minterm waarvoor de functie 1 is (bijv. voor $f = y'z + xz'$)

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Maxtermen

- Een maxterm is een logische functie die waar is voor alle rijen van de waarheidstabel behalve één

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

- Een 0-maxterm is een maxterm waarvoor de functie 0 is (bijv. voor $f = y'z + xz'$)

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

“Product-of-Sums” (POS)

= logische uitdrukking waarbij somtermen (OR) vermenigvuldigd (AND) worden

$$\rightarrow f = (x+z) \cdot (y'+z') = (x+z)(y'+z')$$

□ Elke logische functie kan beschreven worden als het product van zijn 0-maxtermen (= canonieke POS)

$$\begin{aligned} & \rightarrow f = (x+y+z) \cdot (x+y'+z) \\ & \quad \cdot (x+y'+z') \cdot (x'+y'+z') \\ & = M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\ & = \prod M(0,2,3,7) \end{aligned}$$

| x | y | z | f | 0-maxterm |
|---|---|---|---|------------------|
| 0 | 0 | 0 | 0 | $M_0 = x+y+z$ |
| 0 | 0 | 1 | 1 | — |
| 0 | 1 | 0 | 0 | $M_2 = x+y'+z$ |
| 0 | 1 | 1 | 0 | $M_3 = x+y'+z'$ |
| 1 | 0 | 0 | 1 | — |
| 1 | 0 | 1 | 1 | — |
| 1 | 1 | 0 | 1 | — |
| 1 | 1 | 1 | 0 | $M_7 = x'+y'+z'$ |

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Canonieke en standaard vorm

□ Canonieke vormen

$$\begin{aligned} & \rightarrow f = x'y'z + xy'z' + xy'z + xyz' \\ & \rightarrow f = (x+y+z)(x+y'+z) \\ & \quad \cdot (x+y'+z')(x'+y'+z') \end{aligned}$$

□ Standaard vormen

$$\begin{aligned} & \rightarrow f = x'y'z + xy'z + xy'z' + xyz' \\ & = (x'+x)y'z + (y'+y)xz' \\ & = y'z + xz' \\ & \rightarrow f = (x+y+z)(x+y'+z)(x+y'+z')(x'+y'+z') \\ & = (y+y')(x+z)(x+x')(y'+z') \\ & = (x+z)(y'+z') \end{aligned}$$

| x | y | z | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Canonieke en standaard vorm

□ In de *canonieke vorm* is elke functie een som van 1-mintermen of een product van 0-maxtermen

➤ Elke minterm of maxterm bevat alle variabelen
⇒ dure implementatie

□ De *standaard vorm* is een som van producttermen of een product van somtermen met het kleinste aantal variabelen

➤ Een productterm of somterm moet niet altijd alle variabelen bevatten
⇒ goedkoper qua implementatie

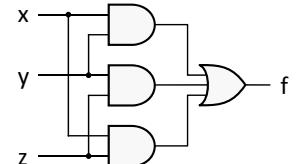
- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

- Logische schakelingen
- Boole-algebra
- ⇒ Synthese
 - canoniek
 - standaard
 - NAND/NOR
- Ontwerppad
- VHDL

Minimale implementatie

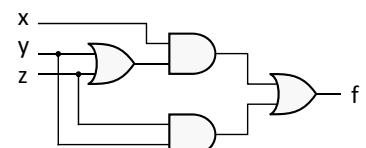
□ De standaard vorm is de goedkoopste implementatie in twee lagen

$$\rightarrow f = xy + xz + yz$$

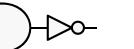


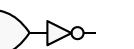
□ Een niet-standaard vorm met meer dan twee lagen kan goedkoper zijn, maar normaal niet sneller

$$\rightarrow f = x(y + z) + yz$$



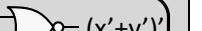
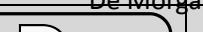
NAND/NOR-poort

NAND  \equiv 

NOR  \equiv 

Voordelen van deze poorten:

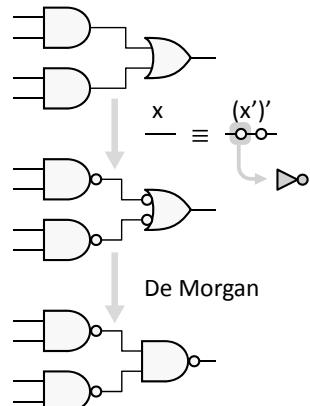
- efficiënte snelle implementatie
- kan elke basispoort emuleren

| Basispoort | met NAND | met NOR |
|---|--|--|
|  |  |  |
|  |  |  $(x+y)'$ |
|  |  De Morgan |  $(x'y)'$ |

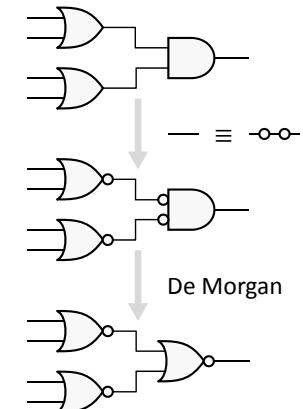
- Logische schakelingen**
- Booleaanse algebra**
- Synthese met logische poorten**
- ➔ **Digitaal ontwerp in grote lijnen**
- Taalgebaseerd hardware ontwerp: VHDL**

NAND/NOR-schakelingen

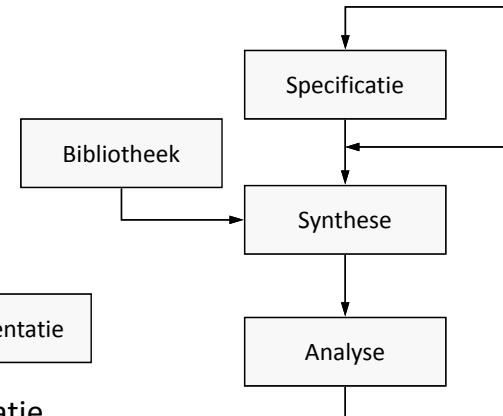
NAND-schakeling:
vertrek van SOP



NOR-schakeling:
vertrek van POS



Digitaal ontwerp in grote lijnen



Documentatie

- ❑ handleiding gebruikers
- ❑ handleiding hersteller
- ❑ documentatie voor verdere ontwikkelingen

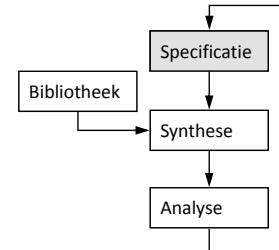
Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- ⇒ Ontwerppad
- VHDL

Specificatie

= beschrijf de functionaliteit, inclusief de 'interface' (interactie met de omgeving)

- Beschrijving
 - in de natuurlijke taal
⇒ dikwijls niet eenduidig
 - met een blokschema
- Dikwijls een onvolledige beschrijving
 - Niet altijd duidelijk wat mogelijk is en wat niet
 - Wordt vervolledigd/aangevuld later in het ontwerpproces
- Maakt dikwijls reeds implementatiebeslissingen die onnodige beperkingen opleggen aan het ontwerp



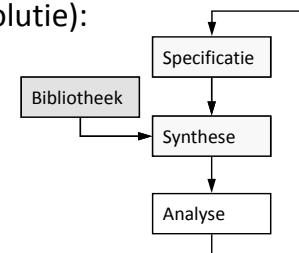
Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- ⇒ Ontwerppad
- VHDL

Bibliotheek van componenten

□ Ontwerpen evolueren (geen revolutie):

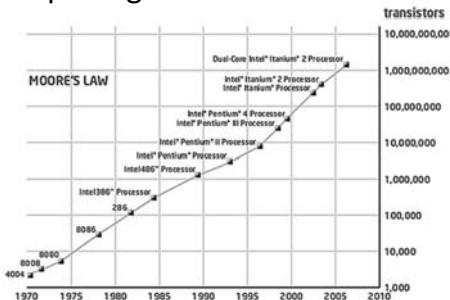
- Hergebruik componenten economisch zeer belangrijk
- Ontbrekende componenten kopen/ontwerpen



□ Apart voor elk syntheseniveau

□ Trend naar bibliotheken op hoog niveau wegens steeds hogere integratieniveaus

- Wet van Moore:
transistoren × 2 per 18–24 maanden ($\times 2^{20} \approx 10^6$ op 30–40 jaar)



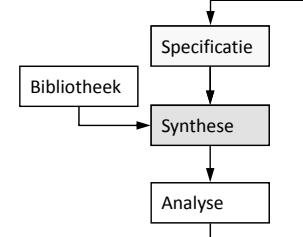
Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- ⇒ Ontwerppad
- VHDL

Synthese

= vertaling van een specificatie op een hoog abstractieniveau naar een lager niveau, waarbij implementatiebeslissingen genomen worden:

$$x+y \Rightarrow \text{16-bit ripple-carry adder} \\ \& 2 \text{ registers}$$



□ Achtereenvolgens op verschillende niveaus:

ASIC = Application Specific IC

RTL = Register Transfer Level

Systeemsynthese
(bouwblokken: processoren, geheugen, ASIC)

Architectuursynthese (of RTL-niveau)
(met RTL-componenten: optellers, tellers, schuifregisters)

Ontwerp op componentniveau
(met basiscomponenten: poorten, flipflops)

Ontwerp van componenten

Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- ⇒ Ontwerppad
- VHDL

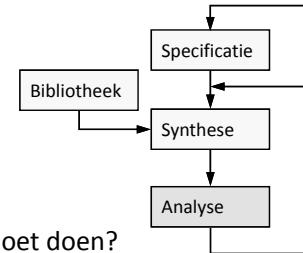
Analyse van het ontwerp

= test of het ontwerp voldoet aan zijn specificaties

□ Na elke synthesestap!

□ Wat wordt getest?

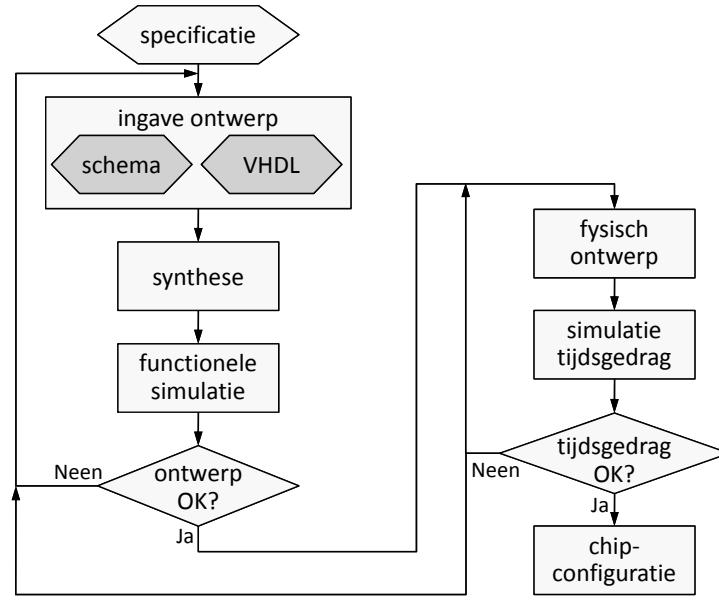
- *Functionaliteit*: doet het wat het moet doen?
- *Kostprijs*: oppervlakte, # pinnen
- *Vermogenverbruik*: $C \times f \times V^2$ (in 17 jaar × 1000)
 - $C \propto$ chipgrootte ↑ (0,25 cm² in 1983; 4 cm² in 2000)
 - $f \uparrow$ (1 MHz in 1983; 1 GHz in 2000)
 - $V \downarrow$ (5 V in 1983; 1,5 V in 2000)
- *Snelheid*: vertraging, 'throughput' (# resultaten/s), ontwikkelingstijd
- *Testbaarheid*: kunnen alle fouten ontdekt worden via testvectoren?



Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- ⇒ Ontwerppad
- VHDL

Ontwerpen met CAD



Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - ⇒ wat?
 - waarom?
 - voorbeeld

Wat is VHDL?

- Acroniem VHDL:
 - VHDL = VHSIC Hardware Description Language
 - VHSIC = Very High Speed Integrated Circuit
- Wat is VHDL?
 - Een programmeertaal om het gedrag van digitale systemen te beschrijven
 - Een taal om een ontwerp in te geven, bruikbaar voor
 - eenduidige specificatie op gedrags- & RTL-niveau
 - simulatie
 - synthese (goed bruikbaar voor RTL-niveau)
 - documentatie
- Standaardisatie
 - 1^e versie: VHDL-87: IEEE 1076
 - 2^e versie: VHDL-93: IEEE 1164
 - 3^e versie: VHDL-2002

Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
 - ⇒ VHDL
 - wat?
 - waarom?
 - voorbeeld

Digitaal ontwerp

- Logische schakelingen
- Booleaanse algebra
- Synthese met logische poorten
- Digitaal ontwerp in grote lijnen
- ➔ Taalgebaseerd hardware ontwerp: VHDL

Digitaal ontwerp

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - ⇒ wat?
 - waarom?
 - voorbeeld

VHDL Analog and Mixed Signal

- = uitbreiding van (zuiver digitale) VHDL met analoge signalen
- VHDL-AMS (IEEE standaard 1076.1-1999)
 - = superset van VHDL-93 (digitaal ontwerp)
 - + continue-tijd-model
 - = set differentiële & algebraïsche vergelijkingen
- Complex en veel minder gebruikt
- Meer info: www.vhdl.org/vhdl-ams

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
- ⇒ wat?
- waarom?
- voorbeeld

Andere 'Hardware Description Languages'

□ Verilog (IEEE 1364)

- Meer verspreid in USA dan in Europa
- Syntactisch verwant met C \Leftrightarrow VHDL meer 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.”
(uit ‘VHDL Made Easy!’, D. Pellerin & D. Taylor)

□ PLD-talen zoals ABEL, PALASM, ...

- Op poortniveau voor een specifieke technologie

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
- wat?
- ⇒ waarom?
- voorbeeld

Voordelen VHDL t.o.v. schema's

- Overdraagbaar over verschillende programma's voor simulatie, synthese, analyse, verificatie, ... van verschillende fabrikanten
- Gemakkelijker om complexe schakelingen te beschrijven: hoger abstractieniveau met automatische synthese
 - Je kan ‘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
 - Gemakkelijk te parametrizeren (woordlengte, geheugendiepte, ...)
 - Gemakkelijk om repetitieve structuren te beschrijven

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
- wat?
- ⇒ waarom?
- voorbeeld

Nadelen VHDL t.o.v. schema's

- Eenvoudig te leren maar moeilijk volledig te beheersen
 - Conceptueel verschillend van software talen
 - Schijnt moeilijke syntax te hebben
 - ⇒ gebruik taalgevoelige editor met sjablonen
- Nogal ‘langdradig’
(veel code nodig voor eenvoudige dingen)
- Een lijst instructies is minder overzichtelijk dan een (niet te groot) blokschema voor een mens
- VHDL bevat meer mogelijkheden dan strikt noodzakelijk voor hardware synthese
(bijv. specificatie tijdsgedrag voor simulatie)

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
- wat?
- ⇒ waarom?
- voorbeeld

Maar er zijn ook beperkingen

- Slechts een subset van VHDL kan automatisch gesynthetiseerd worden en elke fabrikant supporteert een verschillende subset
- De standaard beschrijft enkel de syntax en betekenis, niet hoe code te schrijven (codeerstijl)
 - Eenzelfde gedrag (bijv. selectie uit signalen) kan op heel wat verschillende manieren beschreven worden,
 - die ieder tot een totaal andere implementatie kunnen leiden (bijv. selector of 3-state bus),
 - wat ook nog afhangt van het syntheseprogramma.
- ⇒ *Je moet heel wat ervaring opdoen alvorens je aanvoelt hoe de code moet geschreven worden om tot de meest efficiënte hardware gesynthetiseerd te worden door een bepaald programma!*

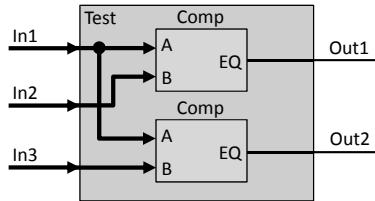
- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

Een voorbeeld

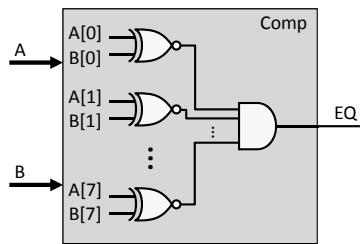
- Ontwerp schakeling ‘Test’ met
 - drie 8-bit ingangen (In_1, In_2, In_3)
 - twee 1-bit uitgangen:
 - $Out_1 = 1 \Leftrightarrow In_1 \equiv In_2$
 - $Out_2 = 1 \Leftrightarrow In_1 \equiv In_3$

- Hiërarchisch schema:

- Schema topniveau, gebruik makend van componenten “Comp”



- Schema component “Comp”



- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

VHDL: component & instantiatie

Specificatie van het topniveau “Test”:

```
-- 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
  Comp1: component Comparator port map (In1,In2,Out1);
  Comp2: component Comparator port map (In1,In3,Out2);
end architecture Struct1;
```

Opmerkingen:

- Deze architectuur beschrijft de structuur, nl. hoe deze entiteit opgebouwd is als verbonden componenten van lager niveau
- De twee componenten “Comparator” werken tegelijkertijd!

Virtuele component: laat onafhankelijke ontwikkeling van alle hiërarchische niveaus toe.
“Comparator” kan later aan “Comp” gekoppeld worden

Twee instantiaties van dezelfde component met zijn signaalbindingen

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

VHDL: entiteit & architectuur

- Declaratie van de entiteit “Comp”:

```
-- 8-bit comparator
-- 
entity Comp is
  port(A,B: in bit_vector(0 to 7);
       EQ: out bit);
end entity Comp;
```

‘entity’ specificeert de interface van de schakeling (zwarte doos in een schema)

```
architecture Behav1 of Comp is
begin
  EQ <= '1' when (A=B) else '0';
end architecture Behav1;
```

‘port’ specificeert een ingangs- of uitgangssignaal

‘architecture’ beschrijft het gedrag en/of de structuur van een entiteit (het binnenste van de doos)

Opmerkingen:

- Een entiteit kan meerdere architecturen hebben: dit zijn verschillende implementaties van hetzelfde gedrag
- Deze architectuur specificeert het gedrag op RTL-niveau; een synthese zal dit omzetten naar verbindingen op poortniveau
- Een ‘port’ heeft een expliciete richting en is een bit(vector)

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
- ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

VHDL: configuratie

- Hoe wordt aangegeven welke architectuur van een entiteit gebruikt moet worden?

- Hoe componenten aan entiteiten koppelen?

```
-- Configuratie: definieer koppeling component met een
--             bepaalde architectuur van een entiteit
```

```
configuration Build1 of Test is
  for Struct1
    for Comp1: Comparator use entity Comp(Behav1)
      port map (A => X, B => Y, EQ => Z);
    end for;
    for Comp2: Comparator use entity Comp(Behav1)
      port map (A => X, B => Y, EQ => Z);
    end for;
  end for;
end configuration Build1;
```

Opmerking:

- ‘configuration’ komt in software overeen met ‘linking’

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
 - ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

Vergelijking met een traditionele taal (zoals C++, Java, ...)

```
// 8-bit comparator
boolean Comp(int A, int B) {
    return (A == B);
}

// Topniveau Test
main() {
    int In1, In2, In3;
    boolean Out1, Out2;

    cin >> In1 >> In2 >> In3;
    Out1 = Comp(In1, In2);
    Out2 = Comp(In1, In3);
    cout << Out1 << Out2;
}
```

Opmerkingen:

- Functie-argumenten zijn altijd ingangen
- Slechts 1 gedragsbeschrijving per functie mogelijk
- De twee "Comp"-functies worden sequentieel uitgevoerd
- "main" wordt éénmaal uitgevoerd en stopt dan

Functie-interface:
argumenten = ingangen
resultaat = uitgang

Gedragsbeschrijving
van de functie

2 oproepen van de functie
"Comp" met de
gekoppelde argumenten

- Logische schakelingen
- Boole-algebra
- Synthese
- Ontwerppad
 - ⇒ VHDL
 - wat?
 - waarom?
 - ⇒ voorbeeld

Verschil met traditionele talen

□ Datatypes

- Nood aan typische hardware-types:
bitvectoren, getallen met een arbitraire
grootte, getallen met vaste komma

□ Gelijktijdigheid ('concurrency')

- Alle hardwarecomponenten werken in parallel

□ Tijdsconcept

- Alle componenten werken continu:
hardware stopt nooit!
- Voor simulatie is een koppeling met het reële
tijdsgedrag van componenten nodig



Inhoudstafel

- Inleiding
- De basis van digitaal ontwerp
 - ➔ Technologische randvoorwaarden
- Combinatorische schakelingen
- Sequentiële schakelingen
- Niet-programmeerbare processoren
- Programmeerbare processoren

Logische waarden voorstellen

Welke fysische waarden worden gebruikt om de twee logische waarden voor te stellen?

- Een “Low” & een “High” fysisch bereik,
 - Meestal 2 spanningen: V_L & V_H
 - Andere mogelijkheden: stromen, optische reflectie, druk, ...
 - Voordeel van een bereik:
 - minder gevoelig aan procesvariaties, die de karakteristieken van componenten veranderen
 - minder gevoelig aan omgevingsvariaties, zoals temperatuur, voedingsspanning, ...
- afgebeeld op de logische niveaus

| | L | H |
|------------------|---|---|
| Positieve logica | 0 | 1 |
| Negatieve logica | 1 | 0 |

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- Technologieën
- Praktische aspecten
- CAD-ontwerp in praktijk

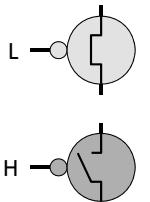
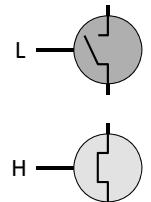
Schakelaars

- Logische poort meestal opgebouwd uit schakelaars met twee standen:



- Elektrische weerstand: $0 \leftrightarrow \infty$
- Drukventiel: open \leftrightarrow gesloten
- Licht doorlaten: wel \leftrightarrow niet

- Werking afhankelijk van stuursignaal



Technologie

Poorten

- Basispoorten
- Complex poorten

Negatieve logica

Technologie

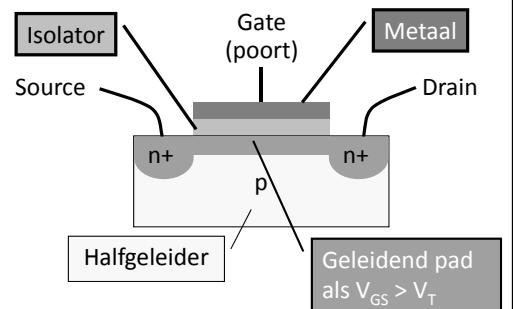
Praktische aspecten

FPGA-ontwerp

MOS-transistor als schakelaar

□ NMOS-transistor

➤ $V_{GS} < V_T$



➤ $V_{GS} > V_T$

□ PMOS-transistor (verwissel n↔p, S↔D)

➤ $V_{GS} < V_T$

➤ $V_{GS} > V_T$

noot:
 $V_{GS} \& V_T < 0$

Technologie

Poorten

- ⇒ Basispoorten
- ⇒ NMOS
- CMOS

- Complex poorten

Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

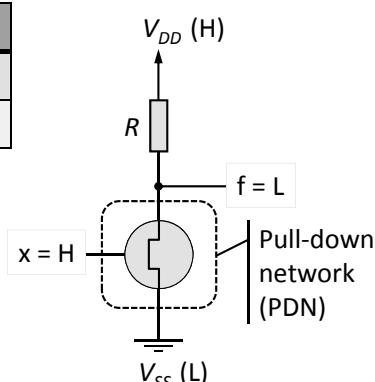
NMOS-inverter



| x | f |
|---|---|
| L | H |
| H | L |

Vanaf nu positieve logica:

| x | f |
|---|---|
| 0 | 1 |
| 1 | 0 |

**□ Nadeel:**

➤ $R_{on} \neq 0 \Rightarrow V_{out}(x=1) = \frac{R_{on}}{R + R_{on}} V_{DD}$

➤ statisch vermogenverbruik $P(x=1) = V_{DD}^2 / (R + R_{on})$

Technologie

Poorten

- ⇒ Basispoorten
- NMOS
- CMOS

- Complex poorten

Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

Technologie voor implementatie

→ Implementatie van poorten**→ Basispoorten: NMOS & CMOS****➤ Complex poorten****□ Negatieve logica****□ Technologieën****□ Praktische aspecten****□ CAD-ontwerp in praktijk**

Technologie

Poorten

- ⇒ Basispoorten
- ⇒ NMOS
- CMOS

- Complex poorten

Technologie

Poorten

- ⇒ Basispoorten
- ⇒ NMOS
- CMOS

- Complex poorten

Negatieve logica

Technologie

Praktische aspecten

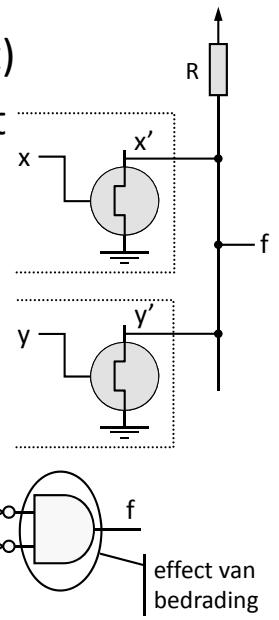
FPGA-ontwerp

'Open-drain' poort

= zet R extern (buiten poort)**□ Wired-AND functionaliteit**

➤ het verbinden van uitgangen resulteert in een bijkomend AND-gedrag

| x | y | x' | y' | f |
|---|---|----|----|---|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |



Technologie

Poorten

⇒ Basispoorten

• NMOS

▪ CMOS

Complex poorten

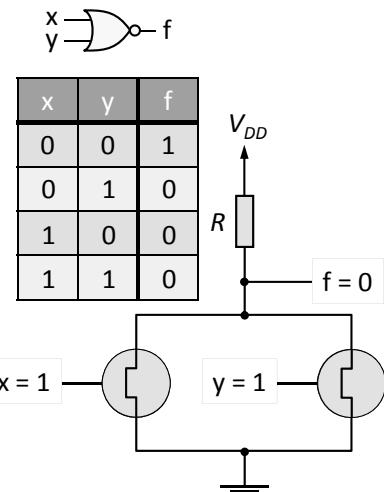
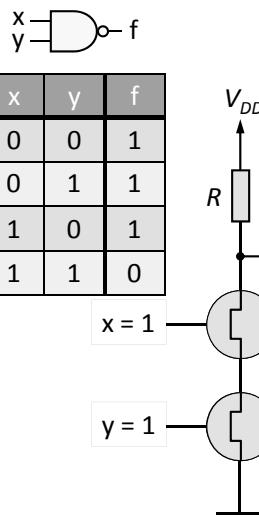
Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

NMOS-poorten: NAND & NOR



- Inverterende schakelingen ⇒ AND = NAND + inverter
⇒ OR = NOR + inverter

KATHOLIEKE UNIVERSITEIT

LEUVEN

Technologie

Poorten

⇒ Basispoorten

• NMOS

⇒ CMOS

▪ Complex poorten

Negatieve logica

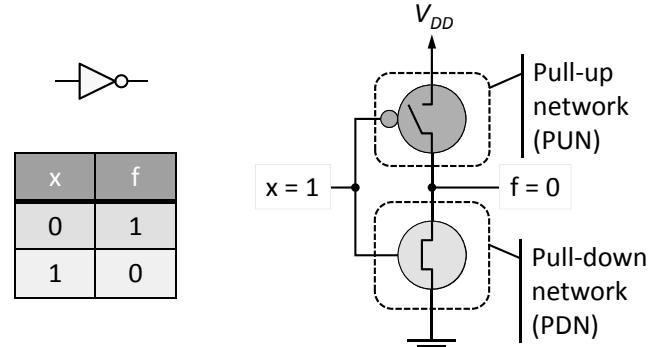
Technologie

Praktische aspecten

FPGA-ontwerp

CMOS-inverter

- PUN is een complementaire schakelaar



- Geen statisch vermogenverbruik!

KATHOLIEKE UNIVERSITEIT

LEUVEN

Technologie

Poorten

⇒ Basispoorten

• NMOS

⇒ CMOS

▪ Complex poorten

Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

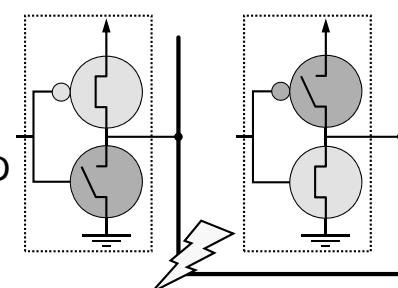
CMOS-poort

Geleidende transistoren hebben
een lage weerstand



Vermijd kortsleuteling (zeer hoge stromen)

- > PUN en PDN nooit samen actief
⇒ PUN = (PDN)'
- > Nooit uitgangen verbinden!
⇒ geen wired-AND



Technologie

Poorten

⇒ Basispoorten

• NMOS

⇒ CMOS

▪ Complex poorten

Negatieve logica

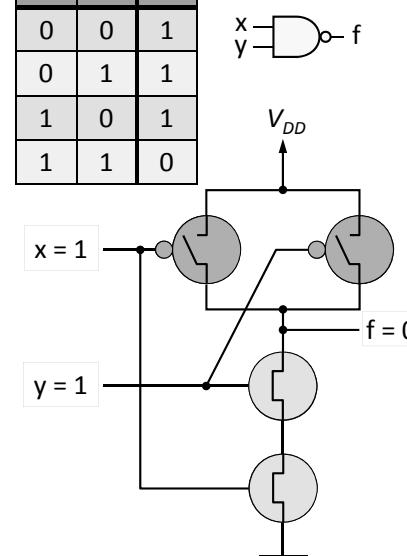
Technologie

Praktische aspecten

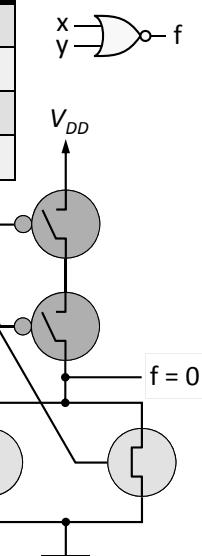
FPGA-ontwerp

CMOS-poorten: NAND & NOR

| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |



KATHOLIEKE UNIVERSITEIT

LEUVEN

Technologie

Poorten

- Basispoorten
- ⇒ Complexe poorten

Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

Technologie voor implementatie

- ➔ Implementatie van poorten
 - Basispoorten: NMOS & CMOS
 - ➔ Complexe poorten
- ❑ Negatieve logica
- ❑ Technologieën
- ❑ Praktische aspecten
- ❑ CAD-ontwerp in praktijk

Technologie

Poorten

- Basispoorten
- ⇒ Complexe poorten

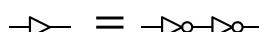
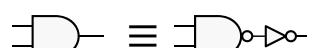
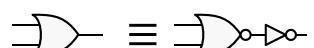
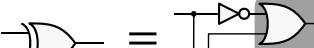
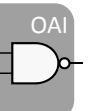
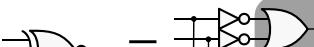
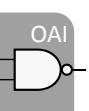
Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

Andere veel gebruikte poorten

- ❑ Buffer
 $f = x = (x')'$  ≡ 
- ❑ AND-poort
 $f = xy = ((xy)')'$  ≡ 
- ❑ OR-poort
 $f = x + y = ((x + y)')'$  ≡ 
- ❑ XOR-poort
 $f = x \oplus y = xy' + x'y = ((x' + y)(x + y'))'$  ≡ 
- ❑ XNOR-poort
 $f = (x \oplus y)' = xy + x'y' = ((x' + y')(x + y))'$  ≡ 

Technologie

Poorten

- Basispoorten
- ⇒ Complexe poorten

Negatieve logica

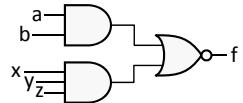
Technologie

Praktische aspecten

FPGA-ontwerp

Complexe CMOS-poorten

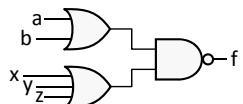
❑ AND-OR-Invert (AOI)



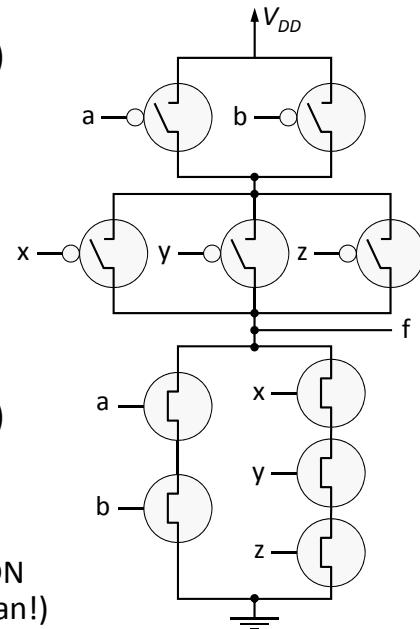
➢ PUN ($f = 1$) :
 $(ab + xyz)' = (a' + b')(x' + y' + z')$

➢ PDN ($f = 0$) :
 $ab + xyz$

❑ OR-AND-Invert (OAI)



➢ verwissel PUN en PDN
(PMOS blijft bovenaan!)



Technologie

Poorten

Negatieve logica

Technologie

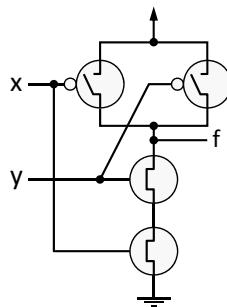
Praktische aspecten

FPGA-ontwerp

Technologie voor implementatie

- ❑ Implementatie van poorten
- ➔ Negatieve logica
 en actief lage signalen
- ❑ Technologieën
- ❑ Praktische aspecten
- ❑ CAD-ontwerp in praktijk

Negatieve logica



❑ Positieve logica

| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



❑ Negatieve logica

| x | y | f |
|---|---|---|
| L | L | H |
| L | H | H |
| H | L | H |
| H | H | L |



enkel als gemengde logica

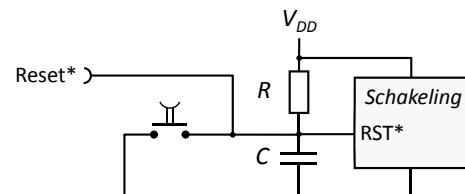
Reset: een typisch actief laag signaal

Reset = breng toestel in een gekende toestand

❑ Verschillende mogelijkheden

- Power-up
- Manueel (bv. met schakelaar)
- Interne of externe signalen

❑ OF-verband \Rightarrow Wired-OR



Actief laag signaal

= het signaal is actief als het "0" is

❑ Meestal aangeduid als \bar{x} , x^* of x'

❑ "Actief" zijn is een interpretatie van een signaal, niet een niveau!

➢ RST* = 0 reset toestel

➢ AND-poort = actief hoge EN
= actief lage OF

❑ Waarom actief lage signalen?

➢ afgesloten verbindingen zijn meestal "1" als ze niet aangestuurd (= niet actief) zijn

➢ actief lage Wired-OR \equiv actief hoge Wired-AND
(meestal OF functionaliteit nodig)

| x^* | y^* | f^* |
|-------|-------|-------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Technologie voor implementatie

❑ Implementatie van poorten

❑ Negatieve logica

➔ Technologieën

❑ Praktische aspecten

❑ CAD-ontwerp in praktijk

Implementatie logische functies

- Standaard chips
 - = poorten (SSI) & eenvoudige functies (MSI/LSI)
 - Beperkte complexiteit ⇒ “glue logic”
- Programmeerbare logica
 - = chip waarvan de logische functie kan geprogrammeerd worden
 - Tot 2M logische cellen (in 2010)
 - Voor prototypes & medium volumes (< 100K stukken/jaar)
- Specifieke chips (ASIC)
 - Duur ⇒ grote volumes

Maatwerk (‘custom design’)

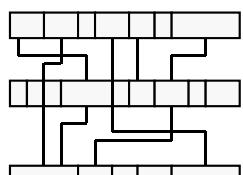
- = elke transistor en elke verbinding wordt afzonderlijk ontworpen als een set rechthoeken (chipoppervlakte)
 - optimaal qua chipoppervlakte, snelheid en vermogenverbruik
 - zeer duur (handwerk)
- Ideaal voor optimaal ontwerp van componenten uit een bibliotheek, die dikwijls kunnen herbruikt worden
- Moet volledig herontworpen worden bij elke technologiewijziging (alle 18 maanden!)

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- ➔ Technologieën
 - ➔ Specifieke chips:
maatwerk, standaard cellen, ‘gate array’
 - Programmeerbare chips
- Praktische aspecten
- CAD-ontwerp in praktijk

Standaard-cel ontwerp

- Bibliotheek van standaard-cellen
 - elke cel is een (complex) poort
 - standaard hoogte, variabele breedte, afgewisseld met ruimte voor bedrading (‘routing channels’)
 - alle ingangen bovenaan, alle uitgangen onderaan
- Sneller ontwerp van complexere bouwblokken
- Cellen ontworpen door chipfabrikanten, geoptimaliseerd voor hun procestechnologie
 - Allocatie componenten (‘placement’)
 - Allocatie bedrading (‘routing’)



Gate array

- = tweedimensionaal rooster van (identieke) poorten: "sea of gates"
 - elke cel is bijv. een 3-input NAND-poort
 - standaard hoogte, afgewisseld met ruimte voor bedrading
 - alle ingangen bovenaan, alle uitgangen onderaan
- Goedkoper dan standaard cellen
 - Alleen de laatste metallisatielaag (voor de verbindingen) is eigen aan een project

Programmeerbare logica

- Gebruik makend van zekeringen (PLA, PLD)
 - Doorbranden zekering verbreekt verbinding
⇒ irreversibel; slechts bijprogrammeren mogelijk
- EE- en flash-programmeerbaar (CPLD)
 - Verbindingen zijn transistoren waarvan het poortniveau opgeladen kan worden
⇒ herprogrammeren mogelijk,
maar traag en slechts een beperkt aantal keren
- Gebruik makend van geheugen (FPGA)
 - Verbindingen zijn transistoren waarvan het poortniveau in een geheugen opgeslagen wordt
⇒ herprogrammeren vlot mogelijk:
(statische of dynamische) herconfiguratie mogelijk
⇒ telkens herladen na aanleggen voedingsspanning

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- ➔ Technologieën
 - Specifieke chips
 - ➔ Programmeerbare chips:
PLA, PAL, PROM, PLD, CPLD, FPGA
- Praktische aspecten
- CAD-ontwerp in praktijk

PLA: Programmable Logic Array

- Programmeerbare AND-matrix & OR-matrix
- Ingangen

AND-matrix

OR-matrix

Uitgangen

$=$

Uitgangen
-
- Afnleiden (sneller & betrouwbaarder)
 - PAL: vaste OR-matrix
 - PROM: vaste AND-matrix (adresdecoder)

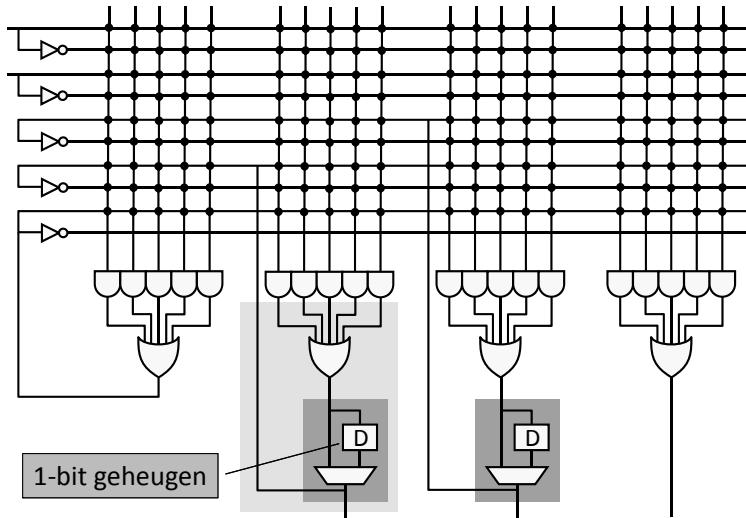
Technologie

Poorten
Negatieve logica
Technologie
▪ Specifiek
⇒ Programmeerbaar
• PLA/PAL/PLD
• FPGA
Praktische aspecten
FPGA-ontwerp

KATHOLIEKE UNIVERSITEIT
LEUVEN

PLD: Programmable Logic Device

- Uitgebreide macrocellen:
extra logica/flipflops aan de uitgangen

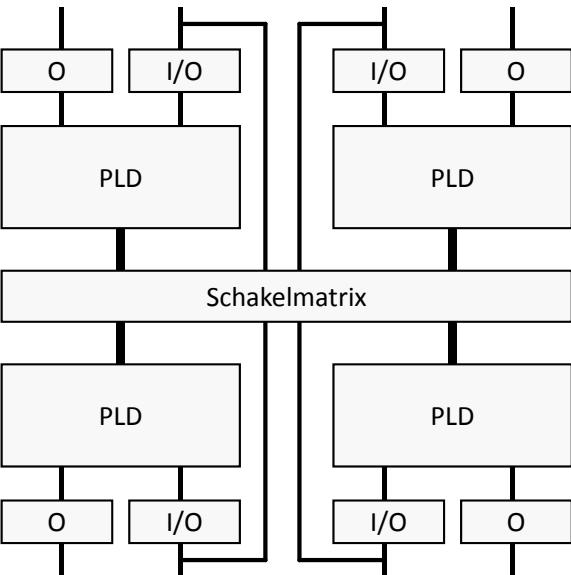


Technologie

Poorten
Negatieve logica
Technologie
▪ Specifiek
⇒ Programmeerbaar
• PLA/PAL/PLD
• FPGA
Praktische aspecten
FPGA-ontwerp

KATHOLIEKE UNIVERSITEIT
LEUVEN

CPLD: Complexe PLD



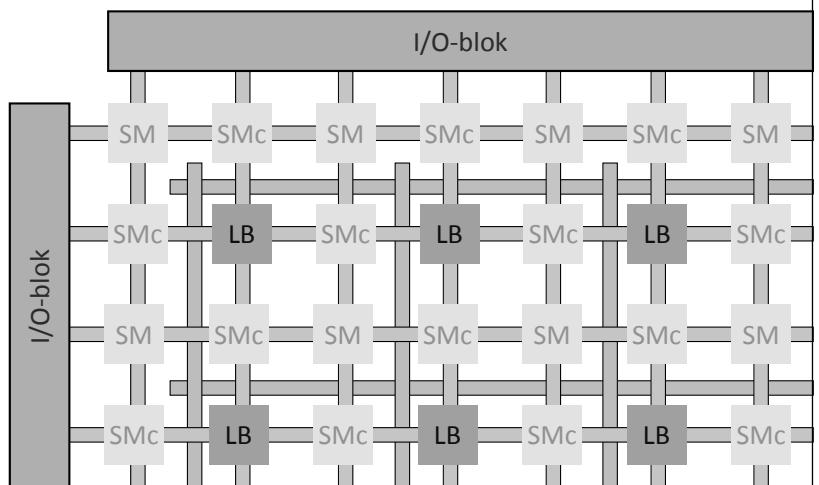
Technologie

Poorten
Negatieve logica
Technologie
▪ Specifiek
⇒ Programmeerbaar
• PLA/PAL/PLD
⇒ FPGA
Praktische aspecten
FPGA-ontwerp

KATHOLIEKE UNIVERSITEIT
LEUVEN

Field Programmable Gate Array

- Logische Blok (CLB)
= logische functionaliteit
- Lange lijnen
- Schakelmatrix en verbinding ernaar (SMC)
- Korte verbindingen



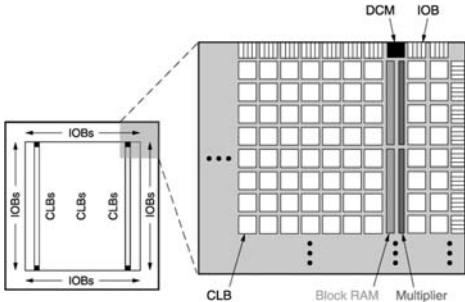
Technologie

Poorten
Negatieve logica
Technologie
▪ Specifiek
⇒ Programmeerbaar
• PLA/PAL/PLD
⇒ FPGA
Praktische aspecten
FPGA-ontwerp

KATHOLIEKE UNIVERSITEIT
LEUVEN

FPGA: extra specifieke hardware

- Spartan-3 (labo)
 - klokgeneratie
 - geheugen
 - vermenigvuldiger



- Andere moderne FPGA
 - meer geheugen (inclusief FIFO)
 - specifieke transceivers of voor hoge snelheid (bijv. PCIe of Ethernet, 10 Gbit/s + CRC)
 - meerdere DSP en microprocessoren (bijv. 32 bit PowerPC met specifieke interface voor hardwareversnellers)

Technologie

Poorten

Negatieve logica

Technologie

▪ Specifiek

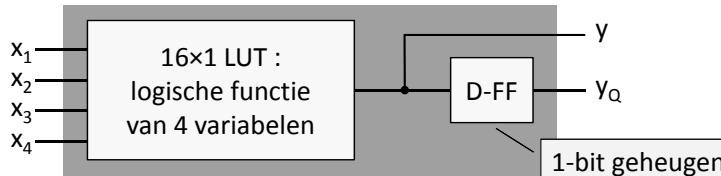
⇒ Programmeerbaar

- PLA/PAL/PLD
- ⇒ FPGA

Praktische aspecten

FPGA-ontwerp

FPGA: logische cel



□ Look-Up Table (opzoektafel) ≡ waarheidstabell

adres data adres data

| adres | data | adres | data |
|-------|------|-------|------|
| 0000 | 0 | 1000 | 1 |
| 0001 | 1 | 1001 | 0 |
| 0010 | 1 | 1010 | 1 |
| 0011 | 0 | 1011 | 0 |
| 0100 | 0 | 1100 | 1 |
| 0101 | 1 | 1101 | 0 |
| 0110 | 1 | 1110 | 1 |
| 0111 | 0 | 1111 | 0 |

x₁ .. x₄ adres

data → $y = (x_1 + x_3) \oplus x_4$

16x1-bit geheugen

Technologie

Poorten

Negatieve logica

Technologie

▪ Specifiek

⇒ Programmeerbaar

- PLA/PAL/PLD
- ⇒ FPGA

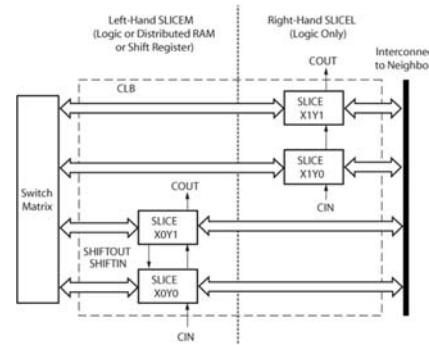
Praktische aspecten

FPGA-ontwerp

Spartan-3 CLB

□ 1 Spartan-3 CLB = 4 "slices"

- 1 slice bevat 2 logische cellen
- Enkel linkse slices kunnen als geheugen of schuifregister gebruikt worden



Technologie

Poorten

Negatieve logica

Technologie

▪ Specifiek

⇒ Programmeerbaar

- PLA/PAL/PLD
- ⇒ FPGA

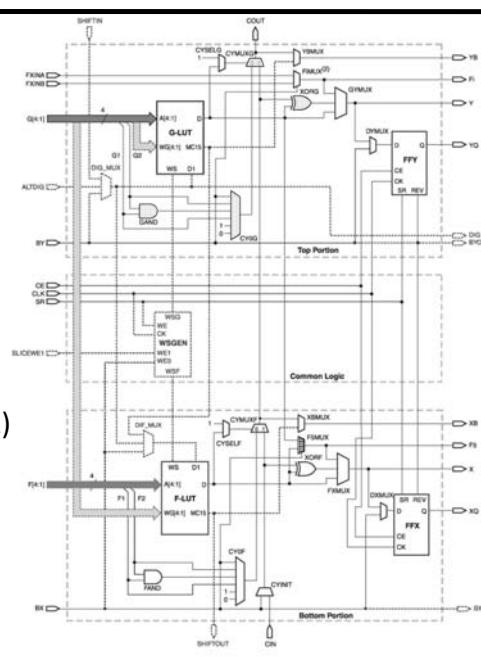
Praktische aspecten

FPGA-ontwerp

Spartan-3 CLB Slice

□ Standaard logica

- 2 × f(4 vars)
ev. met 2 × FF
- 1 × f(5 vars)
ev. met FF
via F5MUX



□ Extra logica

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

Technologie

Poorten

Negatieve logica

Technologie

▪ Specifiek

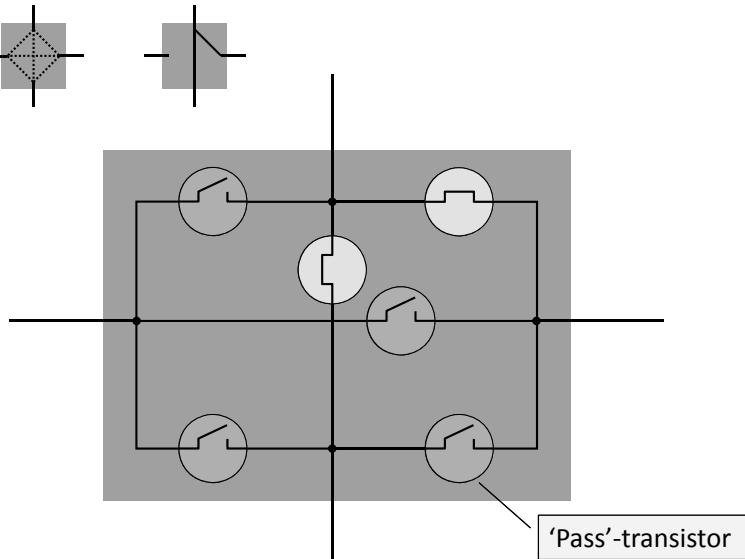
⇒ Programmeerbaar

- PLA/PAL/PLD
- ⇒ FPGA

Praktische aspecten

FPGA-ontwerp

FPGA schakelmatrix-element



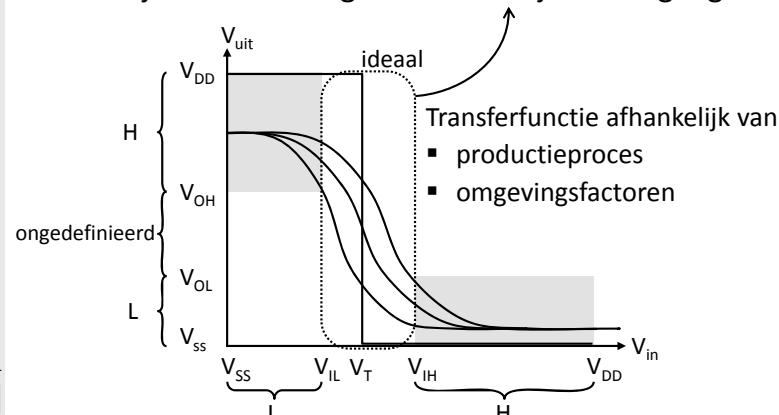
Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- Technologieën
- ➔ Praktische aspecten
 - Spanningsniveaus en ruismarge
 - Dynamisch gedrag
 - Vermogenverbruik
 - “1” en “0” doorgeven
 - Fan-in en fan-out
- CAD-ontwerp in praktijk

Onlogische spanningen

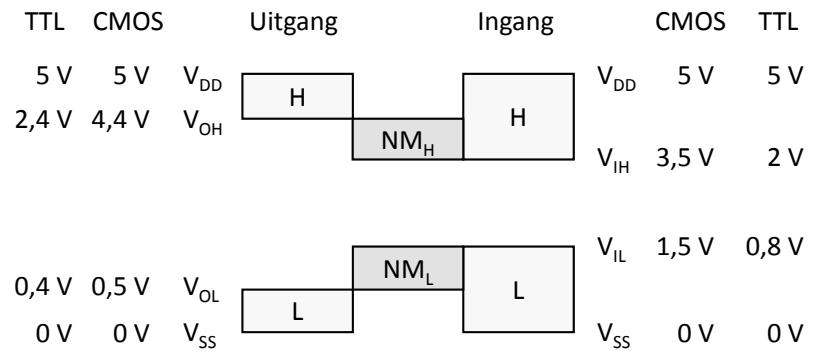
Tijdens de verandering van logisch niveau ontstaan spanningen die niet met een logisch niveau overeenkomen.

- Bij CMOS vermogenverbruik bij de overgang!



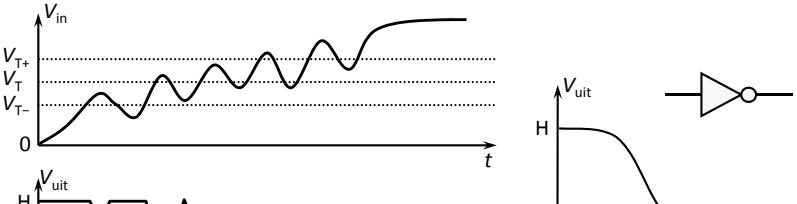
Ruismarge

- = spanningsmarge beschikbaar door een kleiner gebied van uitgangsspanningen te gebruiken dan toegelaten



Schmitt-trigger-ingangen

- Bijkomende problemen voor trage signalen



- Schmitt-trigger-ingangen hebben een hysteresis



Poorten
Negatieve logica
Technologie
Praktische aspecten

- Ruismarge
⇒ Dynamisch
- Vermogen
- Pass
- Fan-in & fan-out

FPGA-ontwerp

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- Technologieën
- ➔ Praktische aspecten
 - Spanningsniveaus en ruismarge
 - ➔ Dynamisch gedrag
 - Vermogenverbruik
 - "1" en "0" doorgeven
 - Fan-in en fan-out
- CAD-ontwerp in praktijk

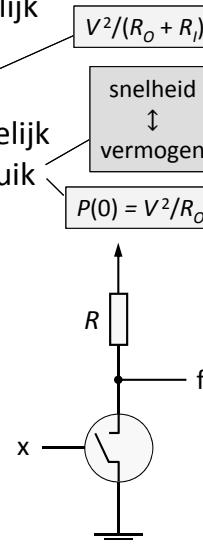
Poorten
Negatieve logica
Technologie
Praktische aspecten

- Ruismarge
⇒ Dynamisch
- Vermogen
- Pass
- Fan-in & fan-out

FPGA-ontwerp

Effect capacitive belasting

- Snelle overgangen als
 - ingangsimpedantie R_i zo groot mogelijk
 - + $V_\infty \approx V_{DD}$
 - + klein statisch vermogenverbruik
 - uitgangsimpedantie R_o zo klein mogelijk
 - groot dynamisch vermogenverbruik (grote schakelstromen)
 - C zo klein mogelijk
⇒ vermindert lange verbindingen
- Daarom NMOS-poort of een 'open-drain'-poort niet populair
 - beperkt vermogenverbruik
⇒ $R_o = R$ redelijk groot
 - grote stijgtijden



Poorten
Negatieve logica
Technologie
Praktische aspecten

- Ruismarge
⇒ Dynamisch
- Vermogen
- Pass
- Fan-in & fan-out

FPGA-ontwerp

Effect capacitieve belasting

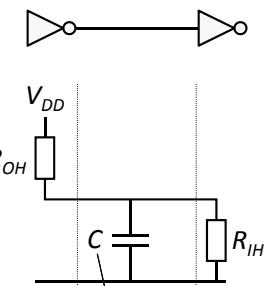
- L → H overgang

$$V(t) = V_\infty \cdot (1 - e^{-t/\tau})$$

met

$$V_\infty \approx V_{DD}$$

$$\Leftrightarrow R_{IH} \gg R_{OH}$$

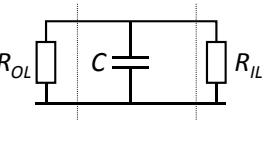


➢ parasitair (draad)
➢ poorten

- H → L overgang

$$V(t) = V_0 \cdot e^{-t/\tau}$$

$$\text{met } \tau = \frac{R_{IL} R_{OL}}{R_{IL} + R_{OL}} C$$



Poorten
Negatieve logica
Technologie
Praktische aspecten

- Ruismarge
⇒ Dynamisch
- Vermogen
- Pass
- Fan-in & fan-out

FPGA-ontwerp

Dynamisch gedrag poort

Stijgtijd (t_r)

Daaltijd (t_f)

90% of V_{IH}

50% V_T

10% V_{IL}

90%

50%

10%

t_{PLH}

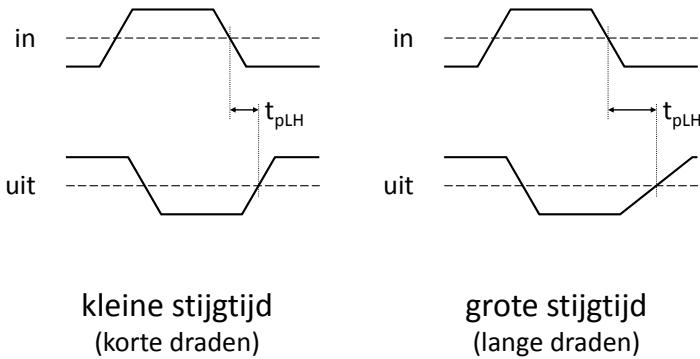
t_{PHL}

Vertragingstijd:
 $t_p = (t_{PLH} + t_{PHL})/2$

t_p

Vertragingstijd

Wordt beïnvloed door de stijg-/daaltijd en dus door de parasitaire capaciteit



Vermogenverbruik

- Verbruikt vermogen moet worden
 - geleverd: kostprijs energie, levensduur batterijen
 - gedissipeerd: warmte moet afgevoerd worden
- Statisch (continu) vermogenverbruik
 - NMOS-poort: $P_{\text{stat}} = 1 \text{ mW} \Rightarrow 1 \text{ miljoen poorten: } 1 \text{ KW!}$
 - CMOS-poort: $P_{\text{stat}} \approx 0 \text{ mW}$ (verwaarloosbare lekstroom)
- Dynamisch vermogenverbruik (tijdens schakelen)
 - = op-/ontladen $C \Rightarrow P_{\text{dyn}} = C \times f \times V^2$ (zie boek vb. 3.12)
 - in praktijk per poort: 35 nW/inverter in vb. 3.8
(20% van de poorten schakelt tegelijkertijd @ 100 MHz)

Bij grote schakelingen is vermogenverbruik een van de belangrijkste ontwerpbeperkingen!

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- Technologieën
- ➔ Praktische aspecten
 - Spanningsniveaus en ruismarge
 - Dynamisch gedrag
 - ➔ Vermogenverbruik
 - “1” en “0” doorgeven
 - Fan-in en fan-out
- CAD-ontwerp in praktijk

Technologie voor implementatie

- Implementatie van poorten
- Negatieve logica
- Technologieën
- ➔ Praktische aspecten
 - Spanningsniveaus en ruismarge
 - Dynamisch gedrag
 - Vermogenverbruik
 - ➔ “1” en “0” doorgeven
 - Fan-in en fan-out
- CAD-ontwerp in praktijk

Technologie

Poorten

Negatieve logica

Technologie

Praktische aspecten

- Ruismarge
 - Dynamisch
 - Vermogen
 - ⇒ Pass
 - Fan-in & fan-out
- FPGA-ontwerp

Technologie

Poorten

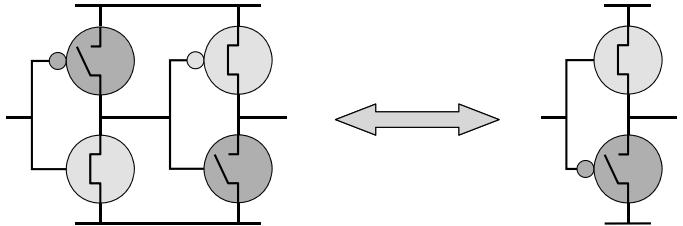
Negatieve logica

Technologie

Praktische aspecten

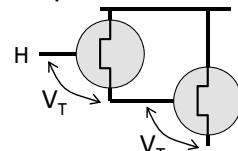
- Ruismarge
 - Dynamisch
 - Vermogen
 - Pass
 - ⇒ Fan-in & fan-out
- FPGA-ontwerp

Waarom enkel inverterende basispoorten?



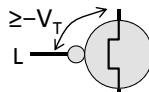
□ NMOS transistor is een slechte pull-up

- slechts geleidend als $V_{GS} > V_T$
- ⇒ uitgangsspanning daalt met V_T na elke tor
- ⇒ NMOS enkel in onderste tak (= kan enkel goed L doorgeven)



□ PMOS transistor is een slechte pull-down

- ⇒ PMOS enkel in bovenste tak (= kan enkel goed H doorgeven)



Technologie

Poorten

Negatieve logica

Technologie

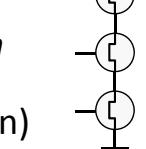
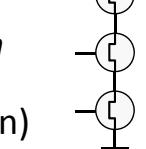
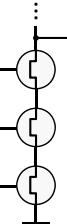
Praktische aspecten

- Ruismarge
 - Dynamisch
 - Vermogen
 - Pass
 - ⇒ Fan-in & fan-out
- FPGA-ontwerp

Fan-in & fan-out van een poort

Fan-in = # ingangen (van poort zelf)

- $t_p \propto$ fan-in
- fan-in $\nearrow \Rightarrow V_{OL} \nearrow$



Fan-out = # ingangen (van andere poorten) die de poort kan aansturen

- reden: er is een maximale I_{Omax}
- fan-out bepaalt maximale frequentie:

$$I_{Omax} = C \times dV/dt = C \times f_{max} \times \Delta V$$

$$\Rightarrow f_{max} = I_{Omax}/(C \times \Delta V)$$

ingangen \nearrow
 $\Rightarrow C \nearrow$

Buffer of "driver" : hogere I_{Omax} (meer vermogen)

- ofwel grotere C (lange draden of hoge fan-out)
- ofwel sneller



Technologie

Poorten

Negatieve logica

Technologie

Praktische aspecten

- Ruismarge
 - Dynamisch
 - Vermogen
 - Pass
 - ⇒ Fan-in & fan-out
- FPGA-ontwerp

Technologie voor implementatie

□ Implementatie van poorten

□ Negatieve logica

□ Technologieën

➔ Praktische aspecten

- Spanningsniveaus en ruismarge
- Dynamisch gedrag
- Vermogenverbruik
- "1" en "0" doorgeven
- ➔ Fan-in en fan-out

□ CAD-ontwerp in praktijk

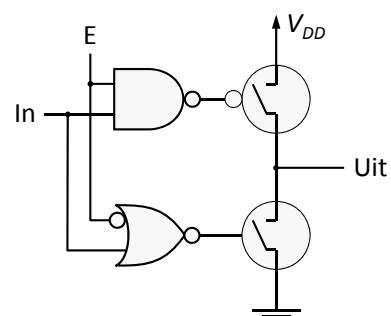
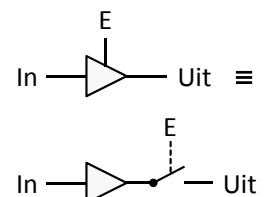
'Tri-state'

= derde mogelijkheid, naast 0 en 1

- Z ("hoog-impedant" of "zwevend")
- = aan niets verbonden

| E(enable) | Uit |
|-----------|-----|
| 0 | Z |
| 1 | In |

□ Een 3-state buffer laat toe om de uitgang los te koppelen van zijn aansturing



□ Een implementatie:

Poorten
Negatieve logica

Technologie

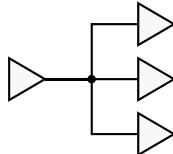
Praktische aspecten

- Ruismarge
- Dynamisch
- Vermogen
- Pass
- ⇒ Fan-in & fan-out

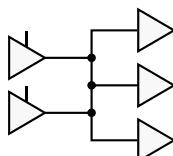
FPGA-ontwerp

Bus(verbinding)

- Traditioneel: 1 aansturing van verbinding



- Bus(verbinding): meerdere aansturingen mogelijk



Slechts 1 aansturing tegelijkertijd!

⇒ 3-state buffers gebruiken

Poorten

Negatieve logica

Technologie

Praktische aspecten

FPGA-ontwerp

- ingave
- synthese
- fysisch
- chip

Xilinx ISE-omgeving



Poorten
Negatieve logica

Technologie

Praktische aspecten

- ingave
- synthese
- fysisch
- chip

Technologie voor implementatie

- Implementatie van poorten

- Negatieve logica

- Technologieën

- Praktische aspecten

→ CAD-ontwerp in praktijk

➤ Xilinx ISE-omgeving
(gebruikt in labozittingen)

Poorten

Negatieve logica

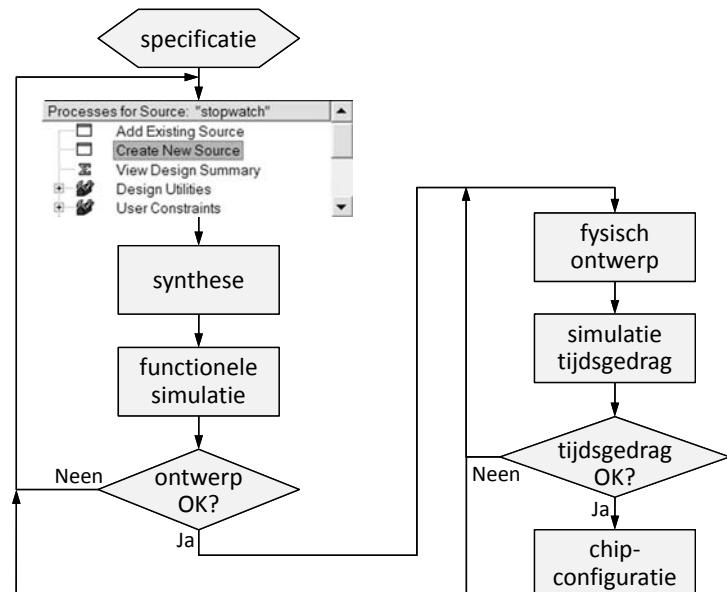
Technologie

Praktische aspecten

FPGA-ontwerp

- ⇒ ingave
- synthese
- fysisch
- chip

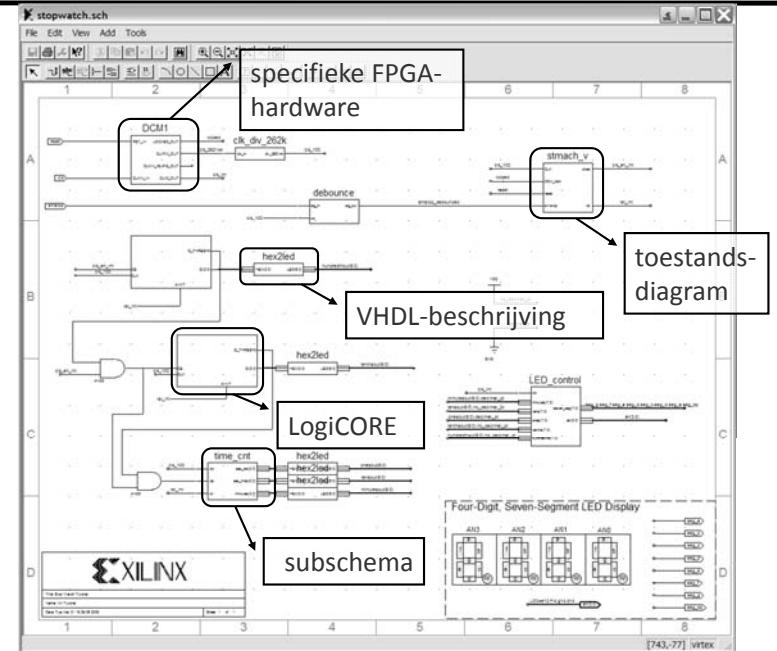
FPGA ontwerpstappen



Technologie

Poorten
Negatieve logica
Technologie
Praktische aspecten
FPGA-ontwerp
▪ ingave
▪ synthese
▪ fysisch
▪ chip

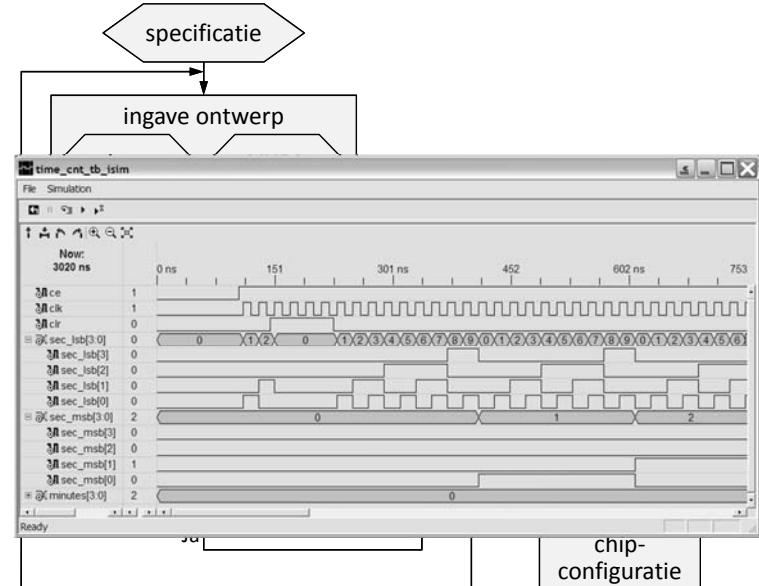
Ingave ontwerp op hiërarchische wijze



Technologie

Poorten
Negatieve logica
Technologie
Praktische aspecten
FPGA-ontwerp
▪ ingave
⇒ synthese
▪ fysisch
▪ chip

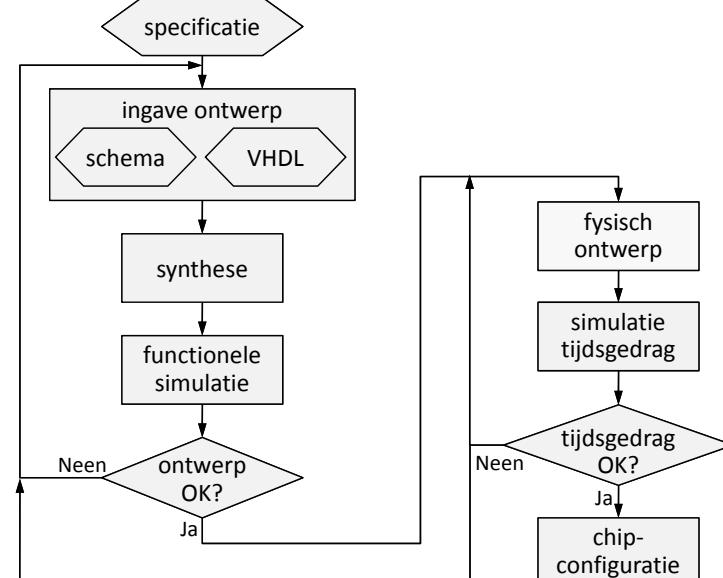
FPGA ontwerpstappen



Technologie

Poorten
Negatieve logica
Technologie
Praktische aspecten
FPGA-ontwerp
▪ ingave
▪ synthese
⇒ fysisch
▪ chip

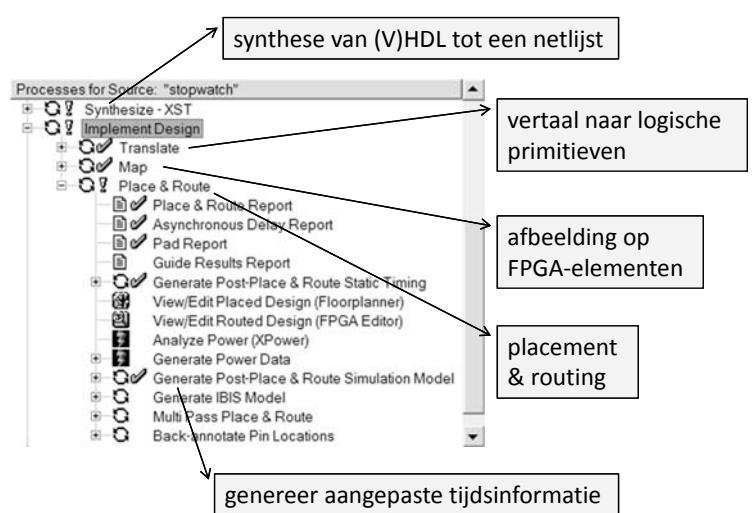
FPGA ontwerpstappen



Technologie

Poorten
Negatieve logica
Technologie
Praktische aspecten
FPGA-ontwerp
▪ ingave
▪ synthese
⇒ fysisch
▪ chip

Fysisch ontwerp



Technologie

Poorten

Negatieve logica

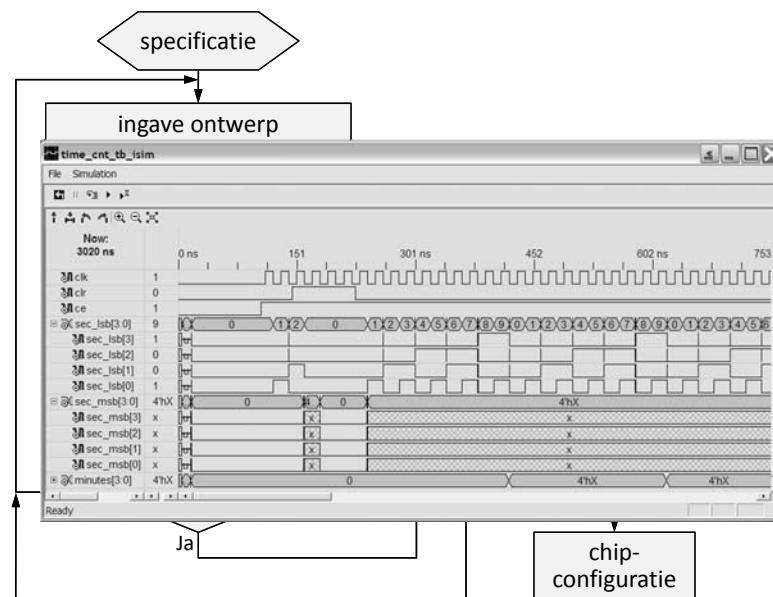
Technologie

Praktische aspecten

FPGA-ontwerp

- ingave
- synthese
- ⇒ fysisch
- chip

FPGA ontwerpstappen



Technologie

Poorten

Negatieve logica

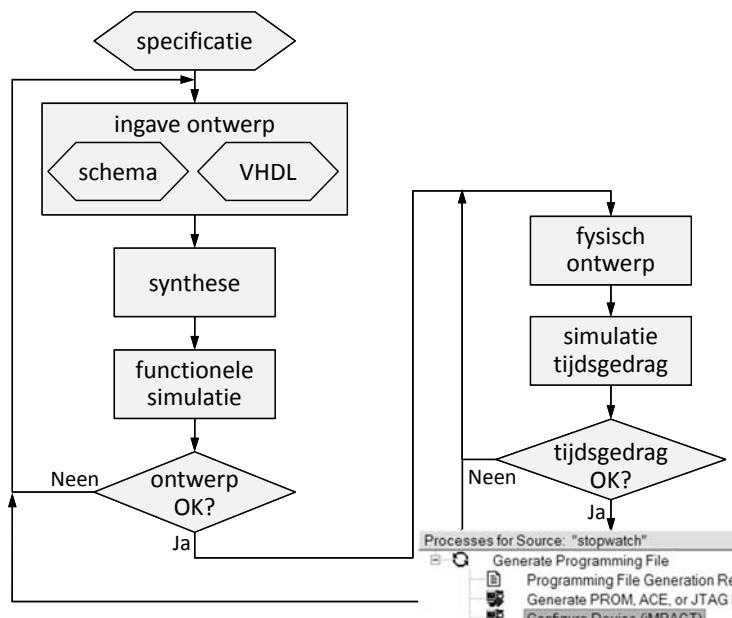
Technologie

Praktische aspecten

FPGA-ontwerp

- ingave
- synthese
- fysisch
- ⇒ chip

FPGA ontwerpstappen



Technologie

Poorten

Negatieve logica

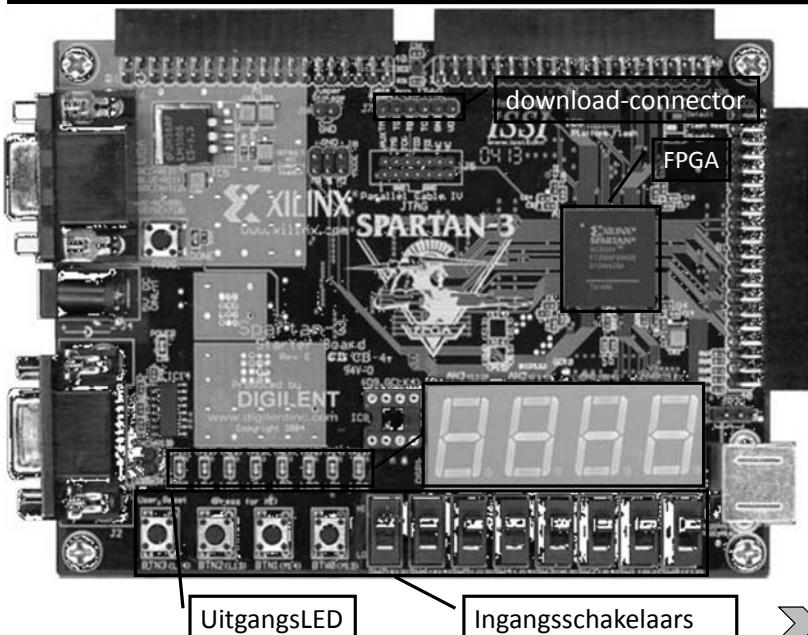
Technologie

Praktische aspecten

FPGA-ontwerp

- ingave
- synthese
- fysisch
- ⇒ chip

Testen op een FPGA-hardwarebord



Inhoudstafel

- Inleiding
- De basis van digitaal ontwerp
- Technologische randvoorwaarden
- ➔ Combinatorische schakelingen:
uitgang = combinatie ingangen
- Sequentiële schakelingen
- Niet-programmeerbare processoren
- Programmeerbare processoren

Waarom minimaliseren?

Schakeling zo goedkoop en snel mogelijk

Minimale kostprijs

$$\text{Kostprijs} = \sum_{\text{alle poorten}} \text{kost(poort)}$$

- Implementatie met CMOS-poorten (relatief t.o.v. inverter)
 - Inverterende poort (INV, NAND, NOR, AOI, OAI)
kostprijs = fan-in
 - Niet-inverterende poort (AND, OR)
kostprijs = fan-in + 1

➤ FPGA-implementatie

- kostprijs = # LB (of # LC)

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- Combinatorische schakelingen in VHDL

Waarom minimaliseren?

Maximale snelheid = minimale vertraging

➤ Vertraging afhankelijk van

- vertraging poort (\propto fan-in)
⇒ afhankelijk van technologie
- (capacitieve) belasting
⇒ afhankelijk van implementatie

➤ Wij gebruiken als rekenregel

- Inverterende poort : $0,6 + \text{fan-in} \times 0,4$
- Niet-inverterende poort : $1,6 + \text{fan-in} \times 0,4$

➤ Totale vertraging = $\sum_{\text{kritisch pad}} \text{vertraging(poort)}$

- Kritisch pad = pad met de grootste vertraging

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Combinatorische schakelingen

→ Minimalisering van logische functies

→ Karnaugh-kaart

- Minimalisering met Karnaugh-kaarten
- Quine-McCluskey
- Realisatie in meer dan 2 lagen
- Welke methode kiezen?

- Rekenkundige basisschakelingen
- Andere basisschakelingen
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

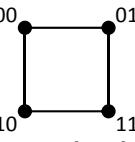
Andere schakelingen

VHDL

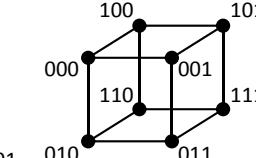
N-kubus

= n-dimensionale kubus waarbij elke dimensie met 1 variabele overeen komt

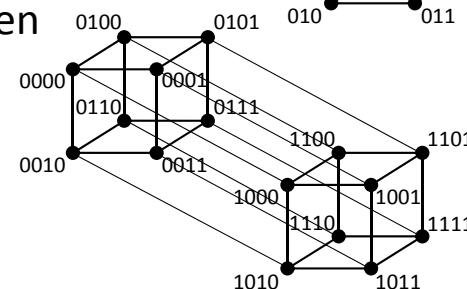
□ 2 variabelen



□ 3 variabelen



□ 4 variabelen



Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Hoe minimaliseren?

□ Via manipulatie van logische uitdrukkingen

- Zeer moeilijk: er bestaat geen methode om de opeenvolgende theorema's te kiezen die leiden tot de minimale oplossing

□ Een voorstelling gebruiken waarin opvalt welke ingang geen belang heeft

➢ Waarheidstabell?

- Duidelijk dat $f = 1$ als $x = 1$ en $y = 0$, onafhankelijk van z
- Maar onduidelijk dat $f = 1$ als $z = 0$, onafhankelijk van x en y

➢ N-kubus of Karnaugh-kaart

| x | y | z | f |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$f = xy' + z'$

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

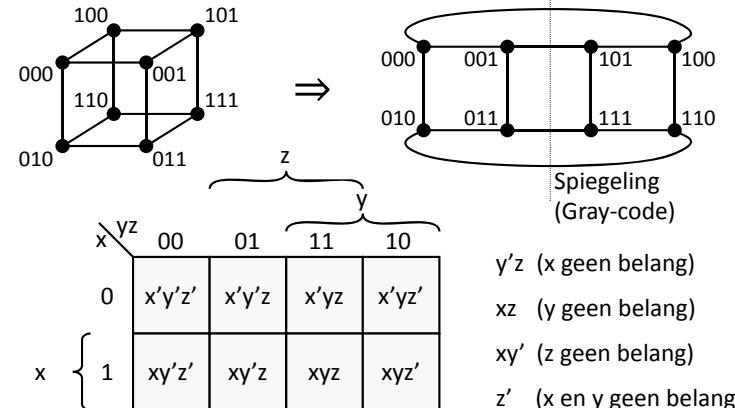
VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Karnaugh-kaart

= 2-dimensionale voorstelling van n-kubus



- elk vierkant heeft n buren

- naburige vierkanten hebben slechts voor 1 variabele een verschillende waarde

y'z (x geen belang)
xz (y geen belang)
xy' (z geen belang)
z' (x en y geen belang)

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Karnaugh: 4 variabelen en meer

□ Spiegeling van kaart met 1 variabele minder

| | | w | | z | | v | |
|----|----|-------|----|-------|----|----|----|
| | | 00 | 01 | 11 | 10 | 00 | w |
| xy | | 00 | 01 | 11 | 10 | 00 | z |
| y | 00 | 0 | 1 | 3 | 2 | 18 | 19 |
| | 01 | 4 | 5 | 7 | 6 | 22 | 23 |
| | 11 | 12 | 13 | 15 | 14 | 30 | 31 |
| | 10 | 8 | 9 | 11 | 10 | 26 | 27 |
| | | 00101 | | 01010 | | | |

➤ Niet alle buren liggen fysisch naast mekaar!

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Karnaugh: meer dan 4 variabelen

□ Herhaling van kaart met 1 variabele minder

| | | w | | z | | v | |
|----|----|-------|----|-------|----|----|----|
| | | 00 | 01 | 11 | 10 | 00 | w |
| xy | | 00 | 01 | 11 | 10 | 00 | z |
| y | 00 | 0 | 1 | 3 | 2 | 16 | 17 |
| | 01 | 4 | 5 | 7 | 6 | 20 | 21 |
| | 11 | 12 | 13 | 15 | 14 | 28 | 29 |
| | 10 | 8 | 9 | 11 | 10 | 24 | 25 |
| | | 00101 | | 01010 | | | |

□ Onoverzichtelijker naarmate meer variabelen!

Combinatorische schakelingen

Minimalisering

- ⇒ Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Gebruik van Karnaugh-kaart

□ Zoek zo groot mogelijke aaneengesloten gebieden (met groote 2^i) met dezelfde waarde

$$\begin{aligned} f = & x'y'z'w' + x'yz'w + x'yzw + xy'z'w' \\ & + xy'zw' + xyz'w + xyzw \end{aligned}$$

$$\begin{aligned} f = & yw \\ & + xy'w' \\ & + y'z'w' \end{aligned}$$

□ Dambordpatroon: XOR

$$x \oplus y \oplus z \oplus w$$

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

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

$$x \oplus y \oplus z$$

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

Combinatorische schakelingen

Minimalisering

- ⇒ Minimalisering
- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

➔ Minimalisering van logische functies

➤ Karnaugh-kaart

➔ Minimalisering met Karnaugh-kaarten

- Minimale AND-OR (SOP) realisatie
- Minimale OR-AND (POS) realisatie
- Onvolledig gespecificeerde functie
- Schakeling met meerdere uitgangen

➤ Quine-McCluskey

➤ Realisatie in meer dan 2 lagen

➤ Welke methode kiezen?

□ Rekenkundige basisschakelingen

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - ⇒ AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen

- Quine-McClusky

- Meerdere lagen

- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Terminologie

Implicit = productterm waarvoor de functie gelijk is aan 1

- $x'y'z'$; $x'y'z$; $x'yz'$; $x'yz$; xyz ;
 $x'y$; $x'z$; xz ; yz ; x'

| | | | |
|---|---|---|---|
| | | x | |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| | | y | |

Priemimplicit = implicit die geen onderdeel is van een andere implicit met minder variabelen

- $x'; yz$

Dekking ("cover") = verzameling implicanten die alle mogelijkheden voorziet waarvoor de functie 1 is

- $f = x'y'z' + x'y'z + x'yz' + x'yz + xyz$
- $f = x' + yz$
- minimale kostprijs als priemimplicanten gebruikt worden

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - ⇒ AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen

- Quine-McClusky

- Meerdere lagen

- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

1. Maak de Karnaugh-kaart

Plaats een op alle plaatsen in de Karnaugh-kaart waar een productterm 1 is.

$$f = x'y'z' + wz + xyz + w'y$$

| | | | |
|---|---|---|---|
| | | z | |
| | | y | |
| 1 | | 1 | 1 |
| | | 1 | 1 |
| | 1 | 1 | 1 |
| x | w | | |

$$\begin{aligned} f = & x'y'z' \\ & + wz \\ & + xyz \\ & + w'y \end{aligned}$$

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - ⇒ AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen

- Quine-McClusky

- Meerdere lagen

- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Minimalisering met Karnaugh-kaarten

Maak Karnaugh-kaart

Bepaal alle priemimplicanten

Bepaal alle essentiële priemimplicanten

Zoek de minimale dekking

Essentiële priemimplicit
= implicit met minstens één 1-minterm die niet in een andere priemimplicit zit

Vul de essentiële priemimplicanten aan met niet-essentiële

2. Bepaal alle priemimplicanten

Bepaal voor elke 1-minterm de grootste implicit(en) die de minterm bevat(ten) en breidt er de lijst van priemimplicanten mee uit.

| | | | |
|---|---|---|---|
| | | z | |
| | | y | |
| 1 | | 1 | 1 |
| | | 1 | 1 |
| | 1 | 1 | 1 |
| x | w | | |

$w'x'z'$
 $x'y'z'$
 $w'y$
 yz
 wz
 $wx'y'$

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - ⇒ AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

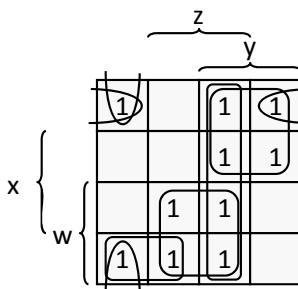
Rekenschakelingen

Andere schakelingen

VHDL

3. Bepaal alle essentiële priemimplicanten

Zoek 1-mintermen die slechts in 1 priemimplicant voorkomen.



$w'x'z'$
 $x'y'z'$
 $w'y$ essentieel
 yz
 wz essentieel
 $wx'y'$

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - ⇒ AND-OR
 - OR-AND
 - don't care
 - meerdere uitgangen
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

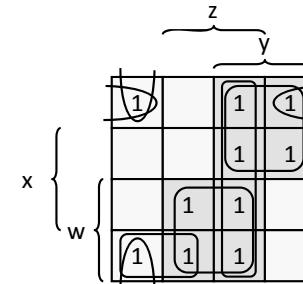
Andere schakelingen

VHDL

4. Zoek de minimale dekking

Zoek de kleinste set van (zo groot mogelijke) priemimplicanten die alle 1-mintermen omvat.

- Initieer de set met de essentiële priemimplicanten.
- Zoek alle dekkingen en kies de minimale.
In praktijk via een gulzige strategie (iteratief): voeg telkens een priemimplicant toe die zoveel mogelijk onbedekte 1-mintermen bevat.



| | |
|----------|------------|
| $w'x'z'$ | 1 |
| $x'y'z'$ | 2 |
| $w'y$ | essentieel |
| yz | 0 |
| wz | essentieel |
| $wx'y'$ | 1 |

$$f_{\min} = x'y'z' + w'y + wz$$

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - ⇒ OR-AND
 - don't care
 - meerdere uitgangen
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

→ Minimalisering van logische functies

- Karnaugh-kaart
- Minimalisering met Karnaugh-kaarten
 - Minimale AND-OR (SOP) realisatie
 - ⇒ Minimale OR-AND (POS) realisatie
 - Onvolledig gespecificeerde functie
 - Schakeling met meerdere uitgangen
- Quine-McCluskey
- Realisatie in meer dan 2 lagen
- Welke methode kiezen?

- Rekenkundige basisschakelingen
- Andere basisschakelingen
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - ⇒ OR-AND
 - don't care
 - meerdere uitgangen
- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Minimalisering voor POS

Duale methode: "0" ↔ "1", AND ↔ OR,
(1-)minterm → (0-)maxterm

2. Bepaal priemimplicanten

- $(w'y'z)', (w'xy)', (xy'z)', (wxz)', (wyz)'$
of $w+y+z', w+x'+y, x'+y+z, w'+x'+z, w'+y'+z$

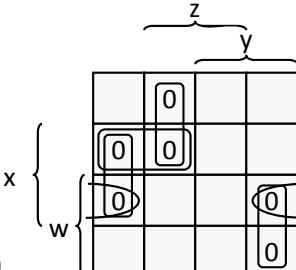
3. Bepaal alle essentiële priemimplicanten

- $w+y+z', w'+y'+z$

4. Zoek de minimale dekking

- voeg $x'+y+z$ toe

$$f = (w+y+z')(w'+y'+z)(x'+y+z)$$



Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - ⇒ don't care
 - meerdere uitgangen

- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

→ Minimalisering van logische functies

➤ Karnaugh-kaart

➔ Minimalisering met Karnaugh-kaarten

- Minimale AND-OR (SOP) realisatie
- Minimale OR-AND (POS) realisatie
- ➔ Onvolledig gespecificeerde functie
- Schakeling met meerdere uitgangen

➤ Quine-McCluskey

➤ Realisatie in meer dan 2 lagen

➤ Welke methode kiezen?

❑ Rekenkundige basisschakelingen

❑ Andere basisschakelingen

❑ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - ⇒ don't care
 - meerdere uitgangen

- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

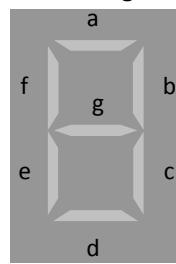
VHDL

Onvolledig gespecificeerde functies

Indien sommige ingangscombinaties niet kunnen optreden ⇒ uitgang is "don't care" (gelijk wat)

Ongespecificeerde uitgang wordt voorgesteld met "X", "d" of "-".

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

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - ⇒ don't care
 - meerdere uitgangen

- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Onvolledig gespecificeerde functies

Als priemimplicanten gezocht worden, behandel "—" als "1" (of "0") als dit de priemimplicant groter maakt.

➤ Na realisatie wordt "—" dan "1" of "0".

➤ Kan verschillend zijn voor SOP en POS!

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

SOP

| | | | |
|---|---|---|---|
| | | - | 0 |
| 0 | - | - | 0 |
| 0 | 0 | - | 0 |

POS

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - don't care
 - ⇒ meerdere uitgangen

- Quine-McClusky
- Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

→ Minimalisering van logische functies

➤ Karnaugh-kaart

➔ Minimalisering met Karnaugh-kaarten

- Minimale AND-OR (SOP) realisatie
- Minimale OR-AND (POS) realisatie
- Onvolledig gespecificeerde functie
- ➔ Schakeling met meerdere uitgangen

➤ Quine-McCluskey

➤ Realisatie in meer dan 2 lagen

➤ Welke methode kiezen?

❑ Rekenkundige basisschakelingen

❑ Andere basisschakelingen

❑ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - don't care
- ⇒ meerdere uitgangen

Quine-McClusky

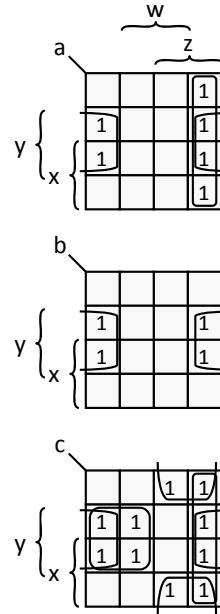
- Meerder lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Meerdere uitgangen



- Nadat alle essentiële priemimplicanten gerealiseerd zijn:
- Selecteer alle priemimplicanten die de mintermen realiseren en die al essentieel zijn voor een andere functie (en dus al gerealiseerd zijn)
 - yw' reeds essentieel voor a en b
 - zw' reeds essentieel voor a
 - Kies de implicant die in het kleinst aantal functies voorkomt (om de fan-out zo laag mogelijk te houden)
 - yw' reeds in a en b
 - zw' reeds in a
- $$\left\{ \begin{array}{l} a = yw' + zw' \\ b = yw' \\ c = yz' + y'z + zw' \end{array} \right.$$

Combinatorische schakelingen

Minimalisering

- Karnaugh
- ⇒ Minimalisering
 - AND-OR
 - OR-AND
 - don't care
- ⇒ meerdere uitgangen

Quine-McClusky

- Meerder lagen
- Methode kiezen

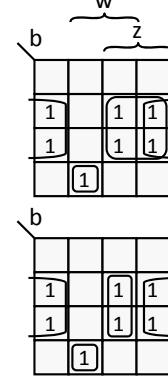
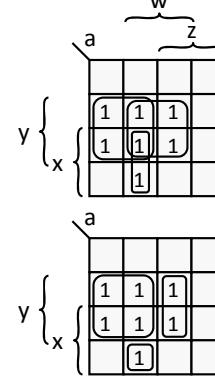
Rekenschakelingen

Andere schakelingen

VHDL

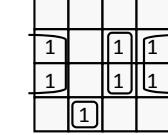
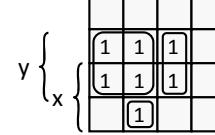
Meerdere uitgangen

- Soms voordelig om niet-priemimplicanten te gebruiken ('trial and error'!)



$$\left\{ \begin{array}{l} a = yz' + yw + xwz' \\ b = yw' + zy + xwy'z' \end{array} \right.$$

Kostprijs: 32



$$\left\{ \begin{array}{l} a = yz' + wyz + xwy'z' \\ b = yw' + wyz + xwy'z' \end{array} \right.$$

Kostprijs: 26

- Mogelijkheden stijgen als meerdere lagen toegelaten zijn

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
 - ⇒ Quine-McClusky
- Meerder lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

- ➔ Minimalisering van logische functies
- Karnaugh-kaart
 - Minimalisering met Karnaugh-kaarten
 - ➔ Quine-McCluskey
 - Realisatie in meer dan 2 lagen
 - Welke methode kiezen?
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
 - ⇒ Quine-McClusky
- Meerder lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Quine-McCluskey

- Karnaugh-kaart

- handmatige methode:
visuele patroonherkenning
- geen garantie op de optimale oplossing
(gulzige strategie)

- Quine-McCluskey

- dezelfde basisideeën als Karnaugh
- computermethode,
gebruik makend van tabellen
- vormt de basis van alle CAD-software van
het ontwerpen van schakelingen
- tijdrovend met de hand ⇒ zie boek

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
- Quine-McClusky
- ⇒ Meerdere lagen
- Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

→ Minimalisering van logische functies

- Karnaugh-kaart
- Minimalisering met Karnaugh-kaarten
- Quine-McCluskey
- Realisatie in meer dan 2 lagen
- Welke methode kiezen?

- Rekenkundige basisschakelingen
- Andere basisschakelingen
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
- Quine-McClusky
- ⇒ Meerdere lagen
- Methode kiezen

Rekenschakelingen

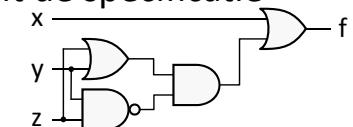
Andere schakelingen

VHDL

Realisatie in meer dan 2 lagen

□ Volgt soms rechtstreeks uit de specificatie

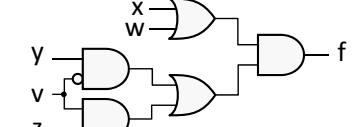
$$\rightarrow f = x \vee ((y \vee z) \wedge \neg(y \wedge z))$$



□ Factoranalyse

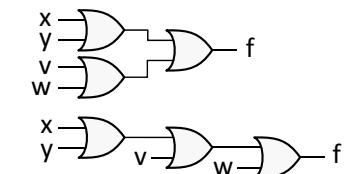
- algebraïsche manipulatie

$$\begin{aligned} f &= v'xy + v'wy + vxz + vwz \\ &= v'y(x + w) + vz(x + w) \\ &= (x + w)(v'y + vz) \end{aligned}$$



➤ beperkte fan-in

$$\begin{aligned} f &= x + y + v + w \\ &= (x + y) + (v + w) \\ &= ((x + y) + v) + w \end{aligned}$$



verkies een boomstructuur voor een minimale vertraging

Ook voor AND en XOR!

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
- Quine-McClusky
- ⇒ Meerdere lagen
- Methode kiezen

Rekenschakelingen

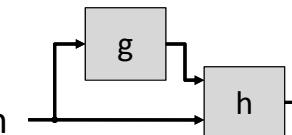
Andere schakelingen

VHDL

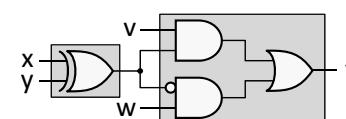
Realisatie in meer dan 2 lagen

□ Functionele ontbinding

= ontbind in subfuncties
waarvan de uitgangen kunnen herbruikt worden



$$\begin{aligned} f &= x'yv + xy'v + xyw + x'y'w \\ \text{stel } g &= x \oplus y = x'y + xy' \\ \text{dan } f &= gv + g'w \end{aligned}$$



□ Geen enkele methode evident: meestal "trial and error"!

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- ⇒ Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Combinatorische schakelingen

→ Minimalisering van logische functies

- Karnaugh-kaart
- Minimalisering met Karnaugh-kaarten
- Quine-McCluskey
- Realisatie in meer dan 2 lagen
- Welke methode kiezen?

□ Rekenkundige basisschakelingen

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

- Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- ⇒ Methode kiezen

Rekenschakelingen

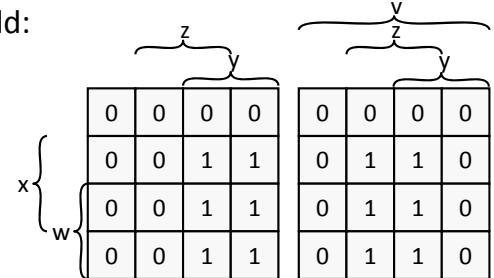
Andere schakelingen

VHDL

Welke methode kiezen?

Probeer alle methoden uit en kies de goedkoopste en snelste oplossing

Voorbeeld:

 Canonieke SOP

- Kostprijs:
 $5 \times 1 + 12 \times 6 + 1 \times 13 = 90$
- Vertraging:
 $1 + 3,6 + 6,4 = 11$

 Canonieke POS

- Kostprijs:
 $5 \times 1 + 20 \times 6 + 1 \times 21 = 146$
- Vertraging:
 $1 + 3,6 + 9,6 = 14,2$

Combinatorische schakelingen

Minimalisering

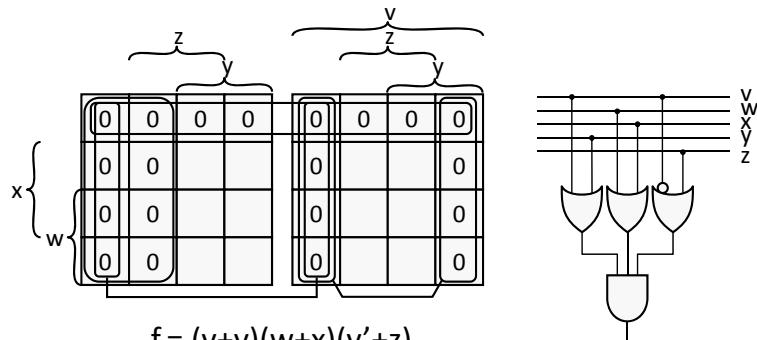
- Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- ⇒ Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Welke methode kiezen?

 Minimale OR-AND (POS)

- Kostprijs: $1 \times 1 + 3 \times 3 + 1 \times 4 = 14$
- Vertraging: $1 + 2,4 + 2,8 = 6,2$

Combinatorische schakelingen

Minimalisering

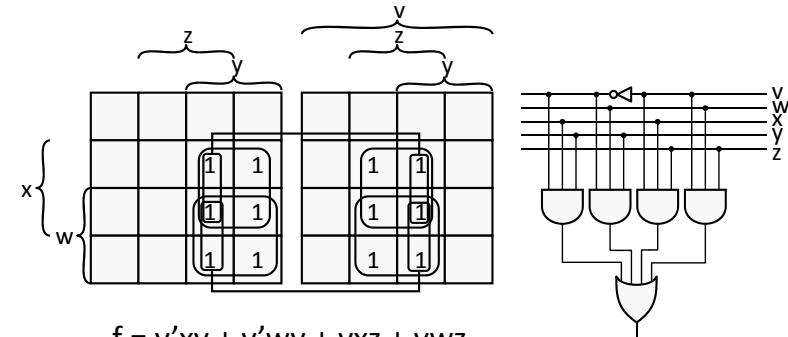
- Karnaugh
- Minimalisering
- Quine-McClusky
- Meerdere lagen
- ⇒ Methode kiezen

Rekenschakelingen

Andere schakelingen

VHDL

Welke methode kiezen?

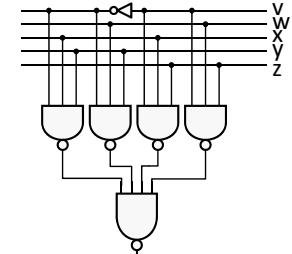
 Minimale AND-OR (SOP)

- Kostprijs: $1 \times 1 + 4 \times 4 + 1 \times 5 = 22$
- Vertraging: $1 + 2,8 + 3,2 = 7$

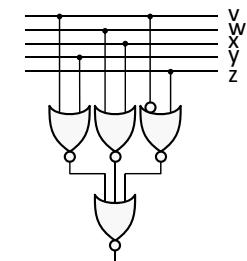
Welke methode kiezen?

 Minimale NAND-NAND

- kostprijs:
 $1 \times 1 + 4 \times 3 + 1 \times 4 = 17$
- vertraging:
 $1 + 1,8 + 2,2 = 5$

 Minimale NOR-NOR

- kostprijs:
 $1 \times 1 + 3 \times 2 + 1 \times 3 = 10$
- vertraging:
 $1 + 1,4 + 1,8 = 4,2$

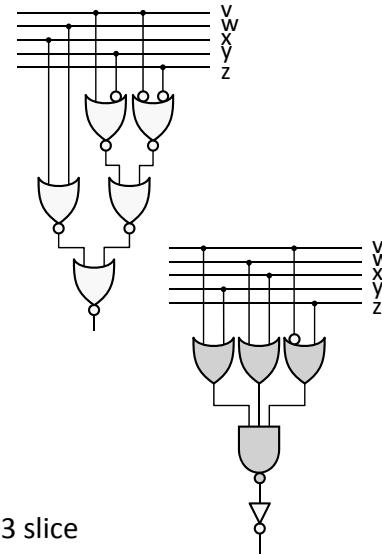


NAND-NAND of NOR-NOR altijd voordeleger!

Welke methode kiezen?

Meerlagenlogica

- kostprijs:
 $3 \times 1 + 5 \times 2 = 13$
- vertraging:
 $1 + 3 \times 1,4 = 5,2$



OAI-implementatie

- kostprijs:
 $2 \times 1 + 1 \times 6 = 8$
- vertraging:
 $1 + 3 + 1 = 5$

FPGA

- 5 variabelen ⇒ 1 Spartan-3 slice

...

Welke methode kiezen?

CAD voor PLD, FPGA en gate array

1. technologie-onafhankelijke synthese
2. "technology mapping": zet om naar beschikbare elementen (poorten of LB)

Samenvatting

| Realisatie | Kost | Rel. kost | Vertraging | Rel. vertr. |
|---------------|------|-----------|------------|-------------|
| Canonieke SOP | 90 | 100 % | 11 | 100 % |
| AND-OR | 22 | 24 % | 7 | 64 % |
| OR-AND | 14 | 16 % | 6,2 | 56 % |
| NAND-NAND | 17 | 19 % | 5 | 45 % |
| NOR-NOR | 10 | 11 % | 4,2 | 38 % |
| Meerlagen | 13 | 14 % | 5,2 | 47 % |
| OAI | 8 | 9 % | 5 | 45 % |

Combinatorische schakelingen

Minimalisering van logische functies

→ Rekenkundige basisschakelingen

- Getallen digitaal voorstellen
- Rekenen met gehele getallen
- Rekenen met vaste komma getallen
- Rekenen met vlottende komma getallen
- Andere voorstellingen van gegevens

Andere basisschakelingen

Combinatorische schakelingen in VHDL

Getallen digitaal voorstellen

$$\text{MSB} \quad \text{LSB} \\ D_r = d_{m-1} \dots d_0, d_{-1} \dots d_{-n} \equiv D = \sum_{i=-n}^{m-1} d_i \times r^i$$

➤ radix r (10 voor decimale notatie)

➤ cijfer $d_i = \lfloor D \times r^{-i} \rfloor \bmod r \quad (0 \leq d_i < r)$

iteratief: $I_0 = \lfloor D \rfloor; \quad F_0 = D - I_0$

$d_i = I_i \bmod r; \quad I_{i+1} = \lfloor I_i / r \rfloor$

$d_{-i} = \lfloor F_{i-1} \times r \rfloor; \quad F_i = F_{i-1} \times r - d_i$

Bijv. het getal $1234,56_{10}$ (in decimale notatie)

$$= 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} + 6 \times 10^{-2}$$

$$= 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1 + 5 \times 0,1 + 6 \times 0,01$$

Combinatorische schakelingen

Minimalisering Rekenschakelingen

⇒ Voorstelling getallen

- Gehele getallen
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen
VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Andere radixen

□ Binair : $r = 2, d_i = 0..1$

bijv. $1011,011_2 = 11,375_{10}$

$$\begin{aligned} &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 \times 8 + 1 \times 2 + 1 \times 1 + 1 \times 0,25 + 1 \times 0,125 \end{aligned}$$

□ Octaal : $r = 8, d_i = 0..7$

bijv. $7654,32_8 = 4012,40625_{10}$

$$\begin{aligned} &= 7 \times 8^3 + 6 \times 8^2 + 5 \times 8^1 + 4 \times 8^0 + 3 \times 8^{-1} + 2 \times 8^{-2} \\ &= 7 \times 512 + 6 \times 64 + 5 \times 8 + 4 \times 1 + 3 \times 1/8 + 2 \times 1/64 \end{aligned}$$

□ Hexadecimaal : $r = 16, d_i = 0..9 \text{ & A..F}$

bijv. $F9D,76_{16} = 3997,4609375_{10}$

$$\begin{aligned} &= 15 \times 16^2 + 9 \times 16^1 + 13 \times 16^0 + 7 \times 16^{-1} + 6 \times 16^{-2} \\ &= 15 \times 256 + 9 \times 16 + 13 + 7/16 + 6/256 \end{aligned}$$

Combinatorische schakelingen

Minimalisering Rekenschakelingen

⇒ Voorstelling getallen

- Gehele getallen
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen
VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Radix-conversie: $r_1 \rightarrow r_2$

Als $r_1 = r_0^p$ en $r_2 = r_0^q$

q cijfers (r_1) $\rightarrow pq$ cijfers (r_0)

$\rightarrow p$ cijfers (r_2)

bijv. hex ($r_1 = 2^4$) \rightarrow octaal ($r_2 = 2^3$)

$$\begin{aligned} 988B_{16} &\rightarrow 1001|1000|1000|1011 = \\ &1|001|100|010|001|011 \\ &\rightarrow 114213_8 \end{aligned}$$

Anders

➤ bepaal het getal D van de voorstelling D_{r_1} ,

➤ bepaal de cijfers van D voor r_2

Combinatorische schakelingen

Minimalisering Rekenschakelingen

⇒ Voorstelling getallen

▪ Gehele getallen

• +

• -

• ×

• ÷

• mod

▪ Vaste komma

▪ Vlottende komma

▪ Andere

Andere schakelingen
VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Combinatorische schakelingen

□ Minimalisering van logische functies

➔ Rekenkundige basisschakelingen

➤ Getallen digitaal voorstellen

➔ Rekenen met gehele getallen

- Optelling van natuurlijke getallen
- Negatieve getallen en aftrekking
- Vermenigvuldiging
- Andere courante bewerkingen

➤ Rekenen met vaste komma getallen

➤ Rekenen met vlottende komma getallen

➤ Andere voorstellingen van gegevens

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering Rekenschakelingen

⇒ Voorstelling getallen

▪ Gehele getallen

⇒ +

• -

• ×

• ÷

• mod

▪ Vaste komma

▪ Vlottende komma

▪ Andere

Andere schakelingen
VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Binaire optelling & aftrekking van natuurlijke getallen

Decimale optelling

overdracht 0 1 0

x 8 2 7 3

y 5 6 2

som 8 8 3 5

Engelse benaming:

- overdracht = 'carry'
- lenen = 'borrow'

Binaire optelling

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

Binaire aftrekking

lenen 1 1 1 0

x 1 1 1 0 1

y 1 1 1 1

resultaat 0 1 1 1 0

Steeds dezelfde bewerking per bit ⇒ repetitieve hardware

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

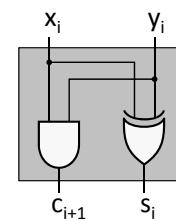
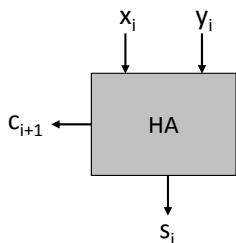
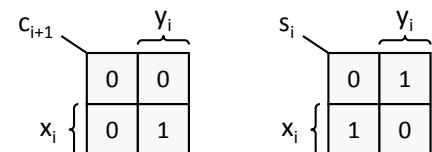
Andere

Andere schakelingen

VHDL

Halve opteller ('Half Adder')

| 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 |



Combinatorische schakelingen

Minimalisering

Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

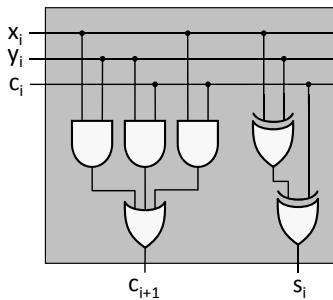
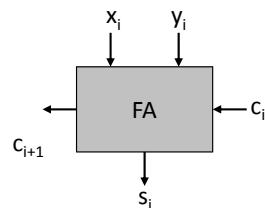
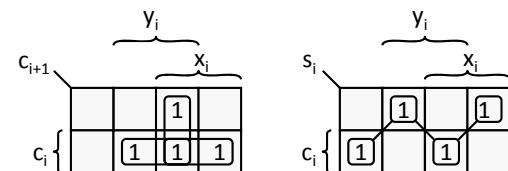
Andere

Andere schakelingen

VHDL

Opteller ('Full Adder')

| 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 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



Combinatorische schakelingen

Minimalisering

Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

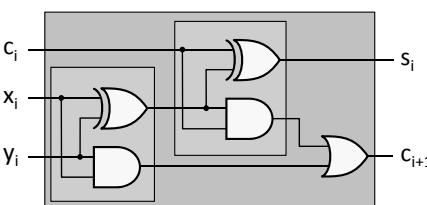
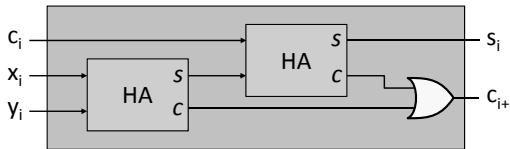
Andere

Andere schakelingen

VHDL

Alternatieve implementatie

Functionele ontbinding:



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

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

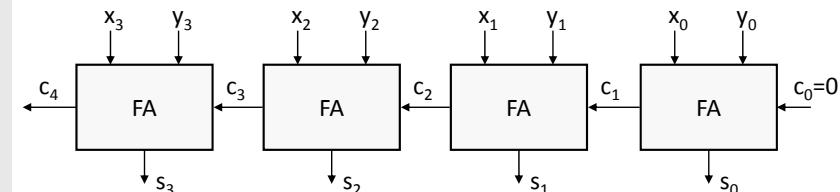
Andere

Andere schakelingen

VHDL

Ripple-carry opteller

4-bit ripple-carry opteller



Kritisch pad: x_0 of $y_0 \rightarrow c_n$: 1 XOR + n AND + n OR

Vertraging: $3,2 + n \times 2,4 + n \times 2,4 = 3,2 + 4,8n$

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

Carry-lookahead opteller

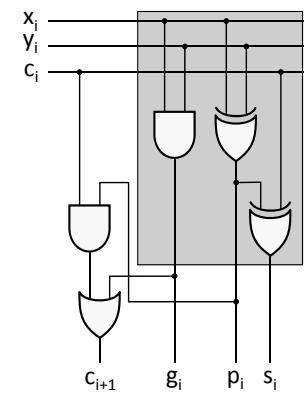
- Ripple-carry opteller is traag omdat het kritisch pad van x_0 naar c_{n+1} lang is
- Versnelling mogelijk door c_{n+1} rechtstreeks (bijv. in 2 lagen) te berekenen uit $c_0, x_0 \dots x_n$ en $y_0 \dots y_n$ = 'carry-lookahead'
- Cascadeerbare oplossing met 'carry-generate' g_i en 'carry-propagate' p_i :

$$c_{i+1} = g_i + p_i c_i$$

$$g_i = x_i y_i$$

$$p_i = x_i y'_i + x'_i y_i$$

Opmerking: c_{i+1} hangt niet rechtstreeks af van x_i of y_i



Combinatorische schakelingen

Minimalisering Rekenschakelingen

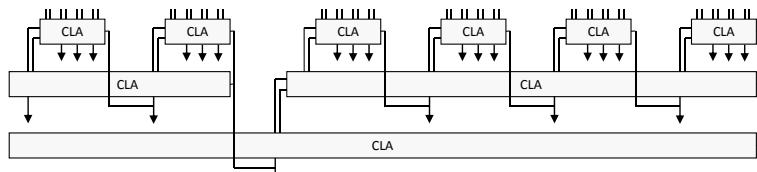
- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

Carry-lookahead opteller

- Realisatie van CLA-functies in 2 lagen
 - Kritisch pad: x_i of $y_i \rightarrow c_{i+n}$: fan-in probleem!
 - 1 XOR + 1 $(n+1)$ -input AND + 1 $(n+1)$ -input OR
 - Vertraging: $3,2 + 2 \times (2 + 0,4n) = 7,2 + 0,8n$
- Hiërarchie van k -bit CLA-generators in $\lceil \log_k n \rceil$ niveaus: bijv. $n = 24, k = 4$



$$\text{Vertraging} = 3,2 + (2 \lceil \log_k n \rceil - 1) \times (4 + 0,8k)$$

⇒ logaritmisch stijgend

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

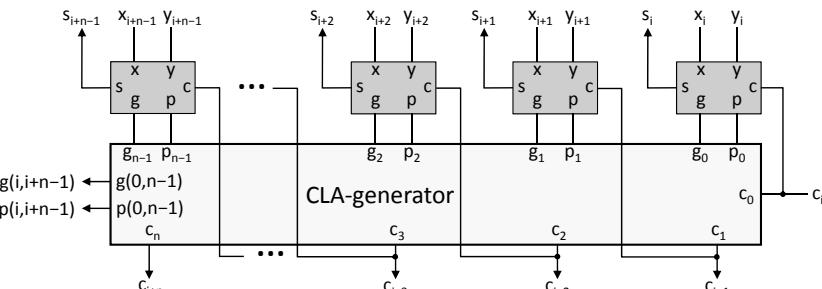
VHDL

Andere schakelingen VHDL

VHDL

Carry-lookahead opteller

- CLA-generatorfuncties:



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

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

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

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - ⇒ +
 - -
 - ALU
 - ×
 - ÷
 - mod

- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

VHDL

Andere schakelingen VHDL

VHDL

Combinatorische schakelingen

- Minimalisering van logische functies

- ➔ Rekenkundige basisschakelingen

- Getallen digitaal voorstellen

- ➔ Rekenen met gehele getallen

- Optelling van natuurlijke getallen

- ➔ Negatieve getallen en aftrekking

- Vermenigvuldiging

- Andere courante bewerkingen

- Rekenen met vaste komma getallen

- Rekenen met vlottende komma getallen

- Andere voorstellingen van gegevens

- Andere basisschakelingen

- Combinatorische schakelingen in VHDL

‘Sign-Magnitude’-voorstelling

= elk getal bestaat uit twee delen:
een teken & een grootte

- Decimaal voorbeeld: $+123_{10}$ en -123_{10}
- Generische voorstelling: $B = \langle s, m \rangle$

□ Binaire getallen

- MSB = teken ('0' = positief, '1' = negatief)
bijv. $01100_2 = +12_{10}$ en $11100_2 = -12_{10}$
- Een ‘sign-magnitude’-getal met n bits ligt tussen $-(2^{n-1} - 1)$ en $+(2^{n-1} - 1)$
met twee voorstellingen voor nul:
 $000\dots 0$ en $100\dots 0$

Twee-complement voorstelling

- Als slechts n bits gebruikt worden, dan
 $r^n \equiv 0 \quad \& \quad D^* = r^n - D \Rightarrow D^* \equiv -D$
 D^* kan dus voor een traditionele optelling gebruikt worden als binaire voorstelling van $-D$ met n bits
 Bijv. bij gebruik van 4 bits
 $D = 3_{10} = 0011_2 \Rightarrow D^* = 1100_2 + 0001_2 = 1101_2$
 $-D = -3_{10}$ kan voorgesteld worden als 1101_2
- Afspraak nodig om 1101_2 te kunnen interpreteren:
is het -3_{10} (2-complement) of 13_{10} (‘unsigned’)?
- Slechts één voorstelling voor nul:
 $D = 0000_2 \Rightarrow -D \equiv D^* = 1111_2 + 0001_2 = 10000_2$
- Een 2-complement getal van n bits kan een waarde van -2^{n-1} tot en met $2^{n-1} - 1$ hebben

Complement-voorstellingen

- Het (cijfer-)complement (‘diminished-radix complement’) D' van een getal D met radix r is het getal waarbij elk cijfer i vervangen is door zijn complement $r - 1 - i$
 - het 9-complement van 123_{10} is 876_{10}
 - het 1-complement van 1101_2 is 0010_2
- Het radix-complement D^* van een getal D met radix r en m cijfers is $D^* = r^m - D$
 - het 10-complement van 123_{10} is $10^3 - 123_{10} = 877_{10}$
 - het 2-complement van 1101_2 is $2^4 - 13_{10} = 3_{10} = 0011_2$
- $D^* = D' + 1$ omdat $D' = (r^m - 1) - D$
 - Dit gebruiken we om het 2-complement te berekenen!

Betekenis van binaire getallen

| | ‘unsigned’ | ‘sign-magnitude’ | 1-complement | 2-complement |
|-------------|------------|------------------|--------------|--------------|
| 0000 | 0 | +0 | +0 | 0 |
| 0001 | 1 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 | 7 |
| 1000 | 8 | -0 | -7 | -8 |
| 1001 | 9 | -1 | -6 | -7 |
| 1010 | 10 | -2 | -5 | -6 |
| 1011 | 11 | -3 | -4 | -5 |
| 1100 | 12 | -4 | -3 | -4 |
| 1101 | 13 | -5 | -2 | -3 |
| 1110 | 14 | -6 | -1 | -2 |
| 1111 | 15 | -7 | -0 | -1 |

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - ⇒ -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

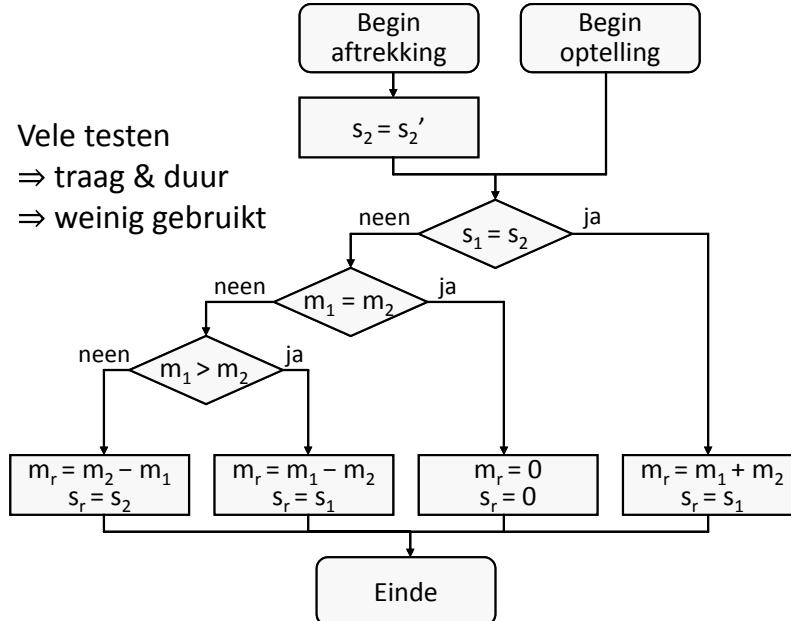
Andere

Andere schakelingen

VHDL

'Sign-magnitude' optelling & aftrekking

Vele testen
⇒ traag & duur
⇒ weinig gebruikt



Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - ⇒ -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

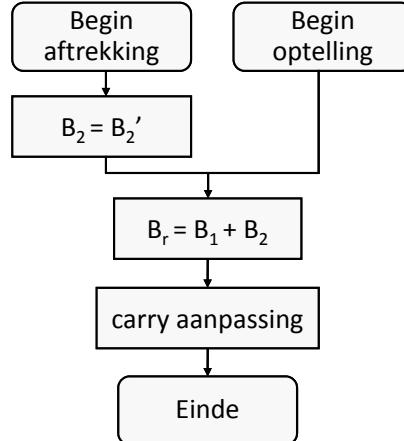
Vlottende komma

Andere

Andere schakelingen

VHDL

1-complement optelling & aftrekking



carry aanpassing:

$$\begin{array}{r}
 0101 \\
 + 1101 \\
 \hline
 10010
 \end{array}
 \quad
 \begin{array}{r}
 5 \\
 -2 \\
 \hline
 3
 \end{array}$$

Eenvoudige implementatie,
maar carry aanpassing verdubbelt vertraging.

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - ⇒ -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

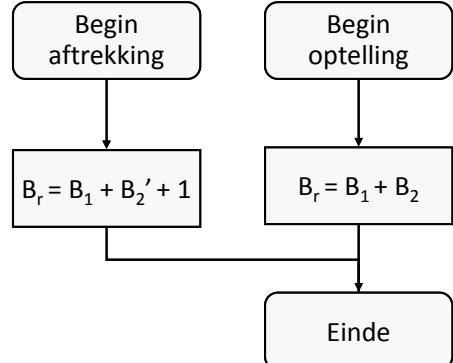
Vlottende komma

Andere

Andere schakelingen

VHDL

2-complement optelling & aftrekking



Eenvoudige en snelle implementatie, zelfs voor aftrekking:

- ❑ Weinig bewerkingen en geen testen
- ❑ Eenvoudige hardware beschikbaar voor 1-complement (B_2')
- ❑ Geen extra hardware nodig voor "+ 1"
(gebruik LSB carry-in)

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - ⇒ -
 - ALU
 - ×
 - ÷
 - mod

Vaste komma

Vlottende komma

Andere

Andere schakelingen

VHDL

2-complement optelling & aftrekking

Optelling

$$\begin{array}{r}
 00000 \\
 0010 +2 \\
 + 0100 +4 \\
 \hline
 0110 +6
 \end{array}
 \quad
 \begin{array}{r}
 00000 \\
 0010 +2 \\
 + 1100 -4 \\
 \hline
 1110 -2
 \end{array}
 \quad
 \begin{array}{r}
 11000 \\
 1110 -2 \\
 + 1100 -4 \\
 \hline
 1010 -6
 \end{array}$$

Aftrekking

$$\begin{array}{r}
 00111 \\
 0010 +2 \\
 + 1011 (-4)' \\
 \hline
 1110 -2
 \end{array}
 \quad
 \begin{array}{r}
 00111 \\
 0010 +2 \\
 + 0011 (-4)' \\
 \hline
 0110 +6
 \end{array}
 \quad
 \begin{array}{r}
 11111 \\
 1110 -2 \\
 + 0011 (-4)' \\
 \hline
 0010 +2
 \end{array}$$

'Overflow'

$$\begin{array}{r}
 01100 \\
 0111 +7 \\
 + 0110 +6 \\
 \hline
 1101 -3
 \end{array}
 \quad
 \begin{array}{r}
 10000 \\
 1001 -7 \\
 + 1010 -6 \\
 \hline
 0011 +3
 \end{array}$$

'Overflow' = resultaat is te groot om correct voorgesteld te kunnen worden

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - ⇒ -
 - ALU
 - ×
 - ÷
 - mod

▪ Vaste komma

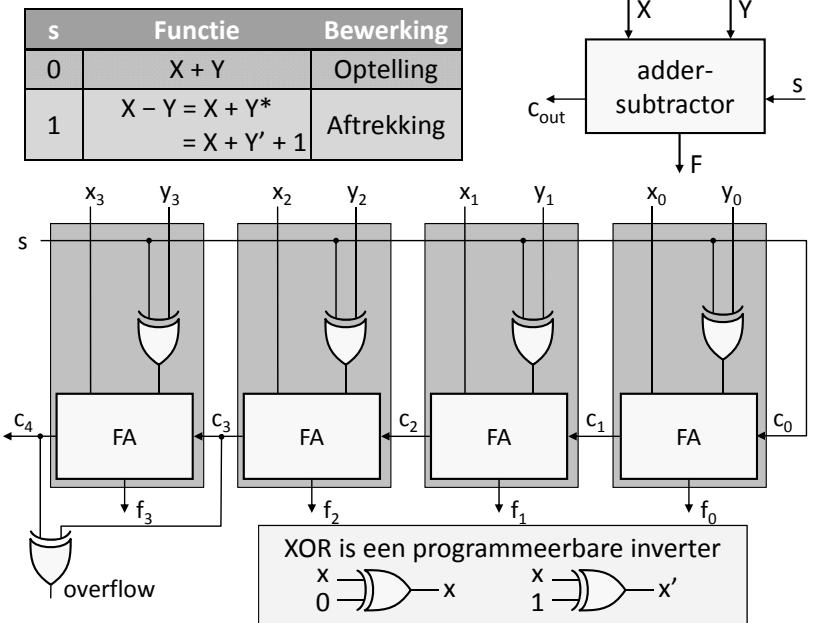
▪ Vlottende komma

▪ Andere

Andere schakelingen

VHDL

Opteller-aftrekker ('adder-subtractor') voor 2-complement getallen



Combinatorische schakelingen

Minimalisering
Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ⇒ ALU
 - ×
 - ÷
 - mod

▪ Vaste komma

▪ Vlottende komma

▪ Andere

Andere schakelingen

VHDL

'Arithmetic-Logic Unit' (ALU)

= component die een aantal rekenkundige en logische functies realiseert

Typisch:

- 4 rekenkundige bewerkingen: optelling, aftrekking, 'increment' (+1), 'decrement' (-1)
- 4 logische bewerkingen: AND, OR, INV, identiteit (doorgeven, nul-operatie)

□ Implementatie is een uitbreiding van opteller-aftrekker: vervang XOR door ALE ('Arithmetic-Logic Extender')

- conditionering ingangen opteller (aftrekking)
- logische operaties

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ⇒ ALU
 - ×
 - ÷
 - mod

▪ Vaste komma

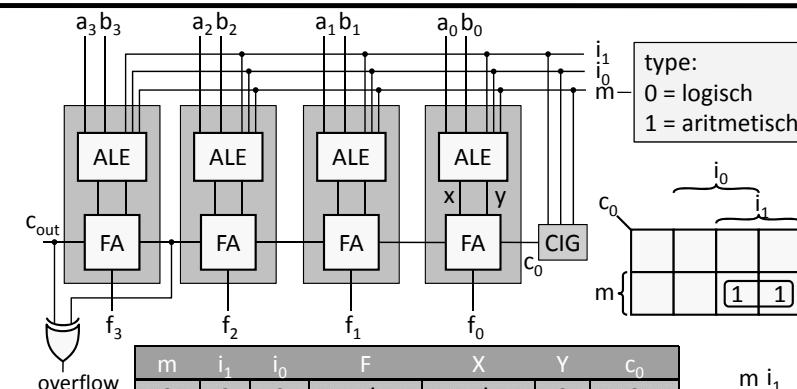
▪ Vlottende komma

▪ Andere

Andere schakelingen

VHDL

Implementatie ALU



Combinatorische schakelingen

Minimalisering
Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ⇒ ALU
 - ×
 - ÷
 - mod

▪ Vaste komma

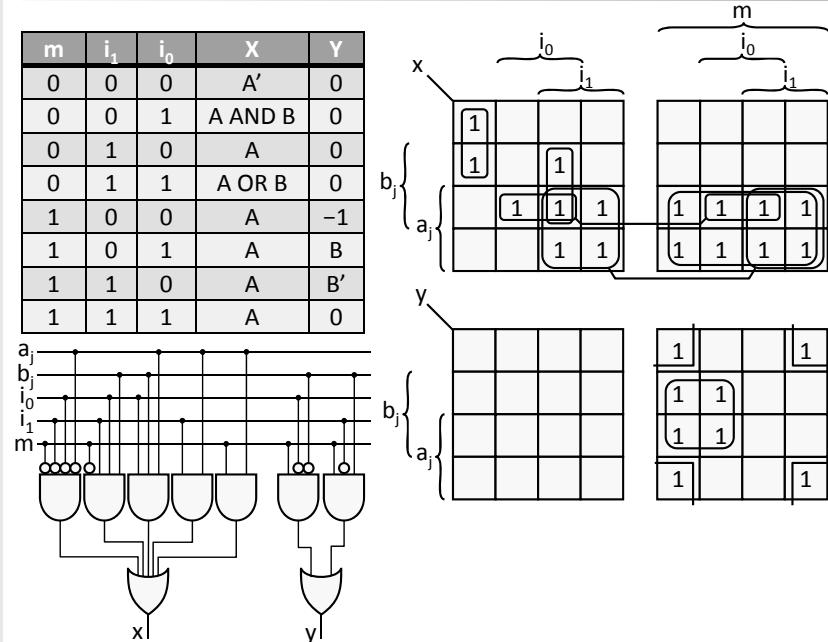
▪ Vlottende komma

▪ Andere

Andere schakelingen

VHDL

Implementatie ALE van ALU



Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ALU
 - ⇒ ×
 - ÷
 - mod
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

KATHOLIEKE UNIVERSITEIT LEUVEN

Combinatorische schakelingen

□ Minimalisering van logische functies

→ Rekenkundige basisschakelingen

➢ Getallen digitaal voorstellen

→ Rekenen met gehele getallen

- Optelling van natuurlijke getallen
- Negatieve getallen en aftrekking
- Vermenigvuldiging
- Andere courante bewerkingen

➢ Rekenen met vaste komma getallen

➢ Rekenen met vlottende komma getallen

➢ Andere voorstellingen van gegevens

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering Rekenschakelingen

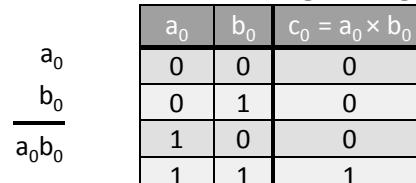
- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ALU
 - ⇒ ×
 - ÷
 - mod
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

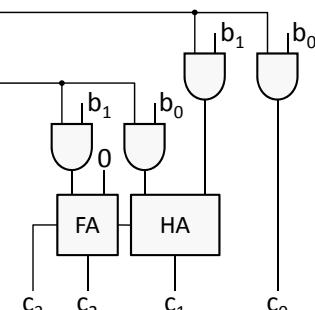
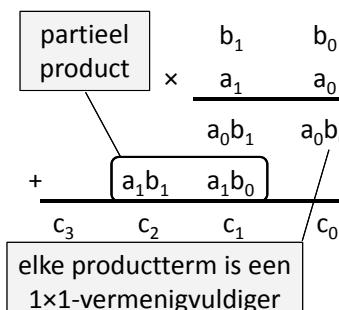
KATHOLIEKE UNIVERSITEIT LEUVEN

Parallelle vermenigvuldiger

□ 1-bit × 1-bit vermenigvuldiger = AND-poort



□ 2-bit × 2-bit vermenigvuldiger



Binaire vermenigvuldiging van natuurlijke getallen

1 0 1 1 0

× 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

Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ALU
 - ⇒ ×
 - ÷
 - mod
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

KATHOLIEKE UNIVERSITEIT LEUVEN

Combinatorische schakelingen

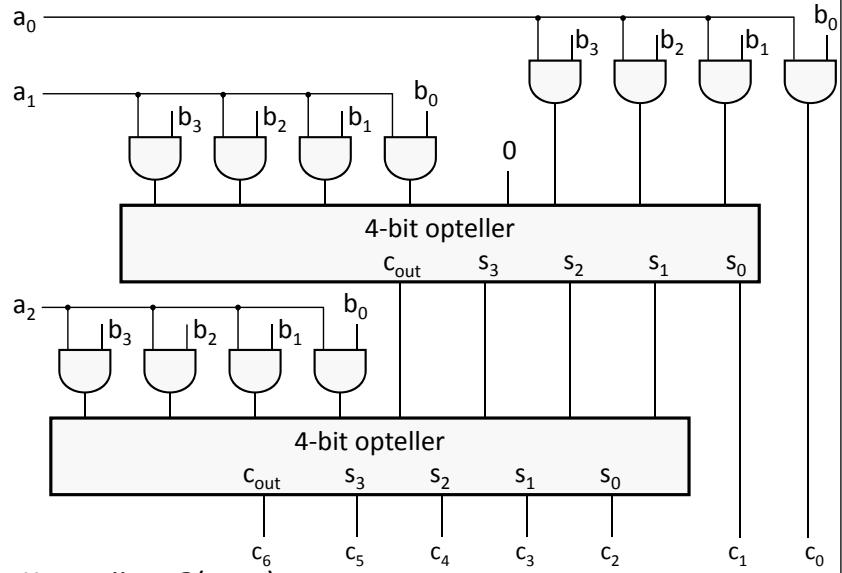
Minimalisering Rekenschakelingen

- Voorstelling getallen
- ⇒ Gehele getallen
 - +
 - -
 - ALU
 - ⇒ ×
 - ÷
 - mod
- Vaste komma
- Vlottende komma
- Andere

Andere schakelingen VHDL

KATHOLIEKE UNIVERSITEIT LEUVEN

4-bit × 3-bit vermenigvuldiger



- Voorstelling getallen
- Gehele getallen
- ⇒ Vaste komma
- Vlottende komma
- Andere

Combinatorische schakelingen

□ Minimalisering van logische functies

→ Rekenkundige basisschakelingen

- Getallen digitaal voorstellen
- Rekenen met gehele getallen
- Rekenen met vaste komma getallen
- Rekenen met vlottende komma getallen
- Andere voorstellingen van gegevens

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

- Voorstelling getallen
- Gehele getallen
- Vaste komma
- ⇒ Vlottende komma
- Andere

Combinatorische schakelingen

□ Minimalisering van logische functies

→ Rekenkundige basisschakelingen

- Getallen digitaal voorstellen
- Rekenen met gehele getallen
- Rekenen met vaste komma getallen
- Rekenen met vlottende komma getallen
- Andere voorstellingen van gegevens

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

- Voorstelling getallen
- Gehele getallen
- ⇒ Vaste komma
- Vlottende komma
- Andere

Getallen met vaste komma

□ fix*<i,f>*: 1101,010 ⇒ i = 4, f = 3

➢ puur geheel : f = 0

➢ puur fractioneel: i = 0

□ # bits na een bewerking (zonder afronding)

$$\text{fix} < i_1, f_1 > + \text{fix} < i_2, f_2 > = \text{fix} < i, f >$$

$$\Rightarrow i = \max(i_1, i_2) + 1 \quad \& \quad f = \max(f_1, f_2)$$

bijv. $\sum^n \text{fix} < i_1, f_1 > \Rightarrow i = i_1 + \lceil \log_2 n \rceil \quad \& \quad f = f_1$

$$\text{fix} < i_1, f_1 > \times \text{fix} < i_2, f_2 > = \text{fix} < i, f >$$

$$\Rightarrow i = i_1 + i_2 \quad \& \quad f = f_1 + f_2$$

bijv. $\prod^n \text{fix} < i_1, f_1 > \Rightarrow i = n \times i_1 \quad \& \quad f = n \times f_1$

- Voorstelling getallen
- Gehele getallen
- Vaste komma
- ⇒ Vlottende komma
- Andere

Getallen met vlottende komma

□ float <m,e>

| | | |
|---|------|----------|
| s | exp. | mantisse |
| 1 | e | m |

bits

$$N = (-1)^s \times \text{mantisse} \times R^{\text{exp}}$$

□ Genormaliseerde fractie: $1 \leq \text{mantisse} < R$

□ Bereik van 32 bit getallen

➢ Gehele getallen (2-complement)

$$-2^{31} \quad 0 \quad 2^{31}-1$$

➢ Genormaliseerde vlottende komma ($R=2, e=8$)

$$-2^{128} \quad -2^{-128} \quad 0 \quad 2^{-128} \quad 2^{128}$$

'underflow'

➢ Niet-genormaliseerde vlottende komma

$$-2^{128} \quad -2^{-151} \quad 0 \quad 2^{-151} \quad 2^{128}$$

IEEE-formaat voor vlottende komma

□ Formaat

| s | exp. | mantisse |
|---|------|----------|
| 1 | e | m |

➤ $R = 2$ bits

➤ E in 'excess'-formaat: bias $B = 2^{e-1} - 1$

| | $E = 0$ | $0 < E < 2^e - 1$ | $E = 2^e - 1$ |
|------------|-------------------------------------|------------------------------------|------------------------|
| $M = 0$ | 0 | $(-1)^s \times 1.M \times 2^{E-B}$ | $(-1)^s \times \infty$ |
| $M \neq 0$ | $(-1)^s \times 0.M \times 2^{-B+1}$ | $(-1)^s \times 1.M \times 2^{E-B}$ | NaN |

niet genormaliseerd

→ verborgen bit

□ Grootte

➤ Enkelvoudige precisie (32 bit)

$e = 8, m = 23, B = 127$

➤ Dubbele precisie (64 bit)

$e = 11, m = 52, B = 1023$

Rekenen met vlottende komma

□ Optelling

$$2,4 \times 10^1 + 9,9 \times 10^2$$

➤ Maak exponenten gelijk $= 0,2 \times 10^2 + 9,9 \times 10^2$

➤ Tel mantissen op $= 10,1 \times 10^2$

➤ Normaliseer $= 1,0 \times 10^3$

□ Vermenigvuldiging

$$5,7 \times 10^1 \times 4,3 \times 10^2$$

➤ Vermenigvuldig mantissen & tel exponenten op $= 24,51 \times 10^3$

➤ Normaliseer (hooguit 1 cijfer) $= 2,5 \times 10^4$

Combinatorische schakelingen

□ Minimalisering van logische functies

→ Rekenkundige basisschakelingen

➤ Getallen digitaal voorstellen

➤ Rekenen met gehele getallen

➤ Rekenen met vaste komma getallen

➤ Rekenen met vlottende komma getallen

→ Andere voorstellingen van gegevens:

BCD, ASCII, ECC, ...

□ Andere basisschakelingen

□ Combinatorische schakelingen in VHDL

BCD-voorstelling ('Binary Coded Decimal')

| Decimaal cijfer | BCD |
|-----------------|------|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

De tabel bevat de enige toegelaten codes

□ 1010_{BCD} bestaat niet!

□ $1010_2 = 10_{10}$
 $= 0001\ 0000_{BCD}$

Voordeel:
 gemakkelijke omzetting
 decimaal \leftrightarrow BCD

Nadeel:
 complexe berekeningen

Combinatorische schakelingen

Minimalisering Rekenschakelingen

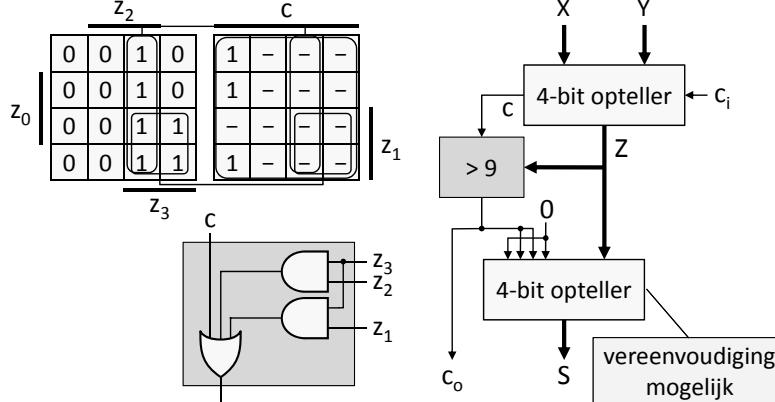
- Voorstelling getallen
- Gehele getallen
- Vaste komma
- Vlottende komma
- ⇒ Andere
 - ⇒ BCD
 - ASCII

Andere schakelingen VHDL

Bewerkingen met BCD

- ⇒ per cijfer zoals een binaire bewerking maar
- correctie als resultaat > 9
 - 10-complement i.p.v. 2-complement

bijv. optelling van BCD-cijfers



Combinatorische schakelingen

Minimalisering Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- Demux
- Encoder
- Vergelijken
- Schuiven

VHDL

Combinatorische schakelingen

Minimalisering Rekenschakelingen

- Voorstelling getallen
- Gehele getallen
- Vaste komma
- Vlottende komma
- ⇒ Andere
 - BCD
 - ⇒ ASCII

Andere schakelingen VHDL

'American Standard Code for Information Interchange' (ASCII)

| $b_3b_2b_1b_0$ | $b_6b_5b_4$ | | | | | | | |
|----------------|-------------|-----|-----|-----|-----|-----|-----|-----|
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NUL | DLE | SP | 0 | @ | P | ' | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | (| 8 | H | X | h | x |
| 1001 | HT | EM |) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [| k | { |
| 1100 | FF | FS | , | < | L | \ | l | |
| 1101 | CR | GS | - | = | M |] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | _ | o | DEL |

Combinatorische schakelingen

- ❑ Minimalisering van logische functies
- ❑ Rekenkundige basisschakelingen
- ➔ Andere basisschakelingen
 - Multiplexer
 - Decoder & demultiplexer
 - Encoder
 - Vergelijken
 - Schuifoperaties
- ❑ Combinatorische schakelingen in VHDL

Combinatorische schakelingen

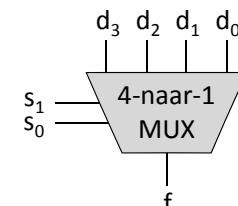
Minimalisering Rekenschakelingen

Andere schakelingen

- ⇒ MUX
- Decoder
- Demux
- Encoder
- Vergelijken
- Schuiven

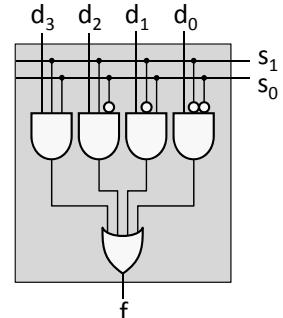
VHDL

Multiplexer (MUX) / Selector



- 2^n -naar-1 MUX:
- 2^n data-ingangen d_i
 - n selectie-ingangen s_i
 - 1 uitgang f

| s_1 | s_0 | f |
|-------|-------|-------|
| 0 | 0 | d_0 |
| 0 | 1 | d_1 |
| 1 | 0 | d_2 |
| 1 | 1 | d_3 |



Combinatorische schakelingen

Minimalisering

Rekenschakelingen

Andere schakelingen

⇒ MUX

▪ Decoder

▪ Demux

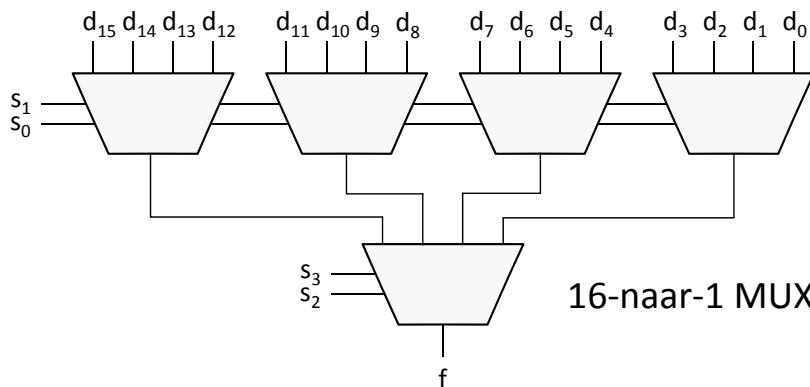
▪ Encoder

▪ Vergelijken

▪ Schuiven

VHDL

Multiplexers cascaderen



$$2^{nm}\text{-naar-1} = n \text{ niveaus van } 2^m\text{-naar-1}$$

$$(2^{nm} - 1)/(2^m - 1) \text{ elementen}$$

KATHOLIEKE UNIVERSITEIT

LEUVEN

H1L1 (10-11) 4-87

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

Andere schakelingen

▪ MUX

⇒ Decoder

▪ Demux

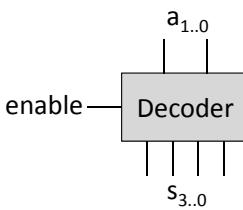
▪ Encoder

▪ Vergelijken

▪ Schuiven

VHDL

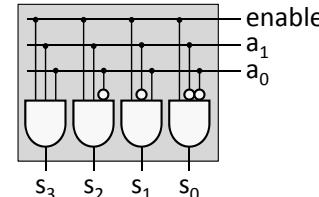
Decoder



n-naar-2ⁿ decoder:

- 2ⁿ uitgangen s_i
- n ingangen a_i
- 1 ingang enable (voor cascadering)

| enable | a ₁ | a ₀ | s ₃ | s ₂ | s ₁ | s ₀ |
|--------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 | - | - | 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 |



Belangrijkste toepassing:
decoderen van een geheugenadres

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

Andere schakelingen

▪ MUX

⇒ Decoder

▪ Demux

▪ Encoder

▪ Vergelijken

▪ Schuiven

VHDL

Combinatorische schakelingen

□ Minimalisering van logische functies

□ Rekenkundige basisschakelingen

→ Andere basisschakelingen

➢ Multiplexer

➔ Decoder & demultiplexer

➢ Encoder

➢ Vergelijken

➢ Schuifoperaties

□ Combinatorische schakelingen in VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

H1L1 (10-11) 4-88

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

Andere schakelingen

▪ MUX

⇒ Decoder

▪ Demux

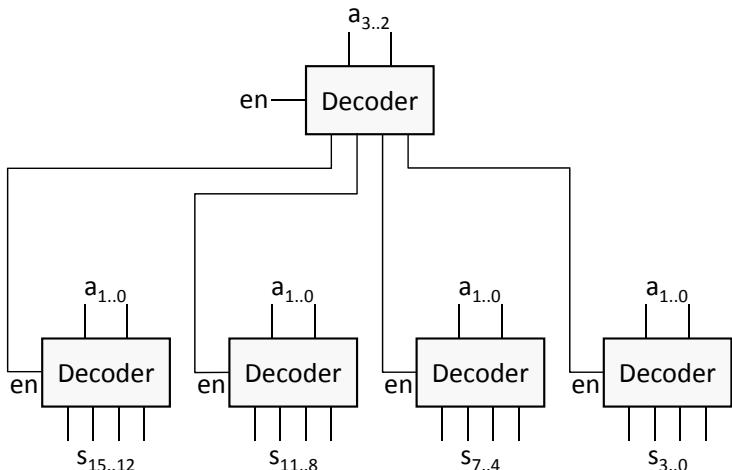
▪ Encoder

▪ Vergelijken

▪ Schuiven

VHDL

Decoders cascaderen



$$nm\text{-naar-}2^{nm} = n \text{ niveaus van } m\text{-naar-}2^m$$

$$(2^{nm} - 1)/(2^m - 1) \text{ elementen}$$

KATHOLIEKE UNIVERSITEIT

LEUVEN

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

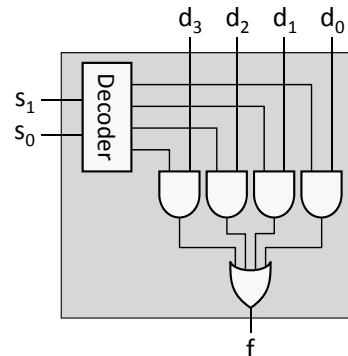
Andere schakelingen

- MUX
- ⇒ Decoder
- Demux
- Encoder
- Vergelijken
- Schuiven

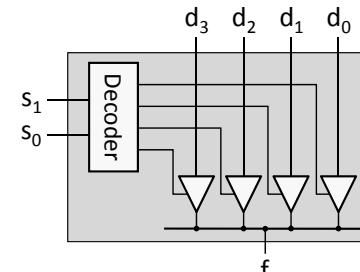
VHDL

Alternatieve implementatie MUX

1) Gebruik een decoder



2) Vervang AND door een 3-state en OR door een draad



Geen gevaar voor kortsleutiging!

Combinatorische schakelingen

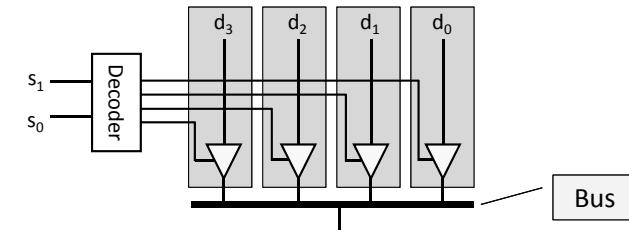
Minimalisering
Rekenschakelingen

Andere schakelingen

- MUX
- ⇒ Decoder
- Demux
- Encoder
- Vergelijken
- Schuiven

VHDL

Alternatieve implementatie MUX



Bus = gedistribueerde versie van MUX

- Gemakkelijk uit te breiden
- Nadeel MUX met hoge fan-in
 - fan-in OR-poort wordt te groot
 - alle ingangen moeten naar 1 centrale plaats
- 3-state buffer meestal gratis in een FPGA
 - elke LB heeft er minstens een die aan een horizontale lange lijn verbonden kan worden

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- ⇒ Demux
- Encoder
- Vergelijken
- Schuiven

VHDL

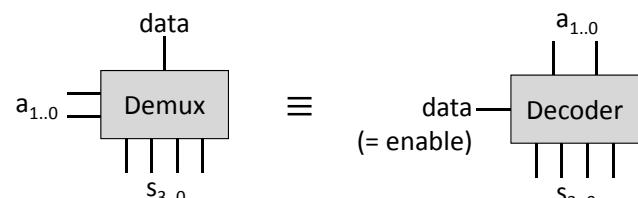
Demultiplexer

= inverse van MUX: 1-naar- 2^n

□ Te realiseren met een decoder:

- enable als data-ingang
- a_i als selectie-ingangen

| a_1 | a_0 | s_3 | s_2 | s_1 | e |
|-------|-------|-------|-------|-------|-----|
| 0 | 0 | 0 | 0 | 0 | e |
| 0 | 1 | 0 | 0 | e | 0 |
| 1 | 0 | 0 | e | 0 | 0 |
| 1 | 1 | e | 0 | 0 | 0 |



Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

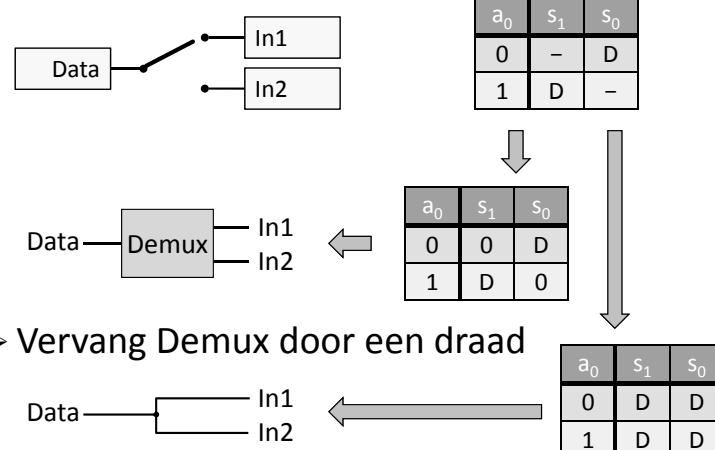
- MUX
- Decoder
- ⇒ Demux
- Encoder
- Vergelijken
- Schuiven

VHDL

Demultiplexer

□ Meestal niet nodig!

Bv. kies verbruiker van data



Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- Demux
- ⇒ Encoder
- Vergelijken
- Schuiven

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- ➔ Andere basisschakelingen
 - Multiplexer
 - Decoder & demultiplexer
 - ➔ Encoder
 - Vergelijken
 - Schuifoperaties
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

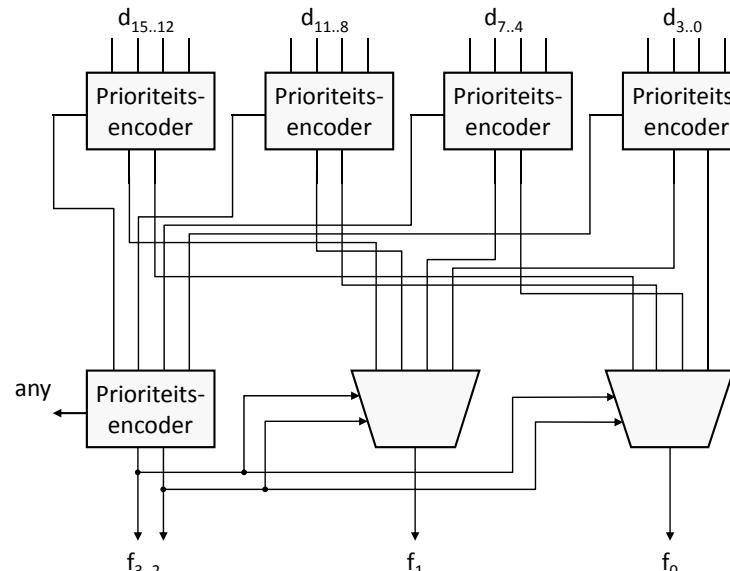
- MUX
- Decoder
- Demux
- ⇒ Encoder
- Vergelijken
- Schuiven

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Prioriteitsencoders cascaderen



Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- Demux
- ⇒ Encoder
- Vergelijken
- Schuiven

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Encoder

- = inverse van decoder: 2^n -naar- n
- Te realiseren met 1 OR-poort per uitgang

| d_3 | d_2 | d_1 | d_0 | any | f_0 | f_1 |
|-------|-------|-------|-------|-----|-------|-------|
| 0 | 0 | 0 | 0 | 0 | – | – |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |

$$\begin{aligned} \text{any} &= d_3 + d_2 + d_1 + d_0 \\ f_0 &= d_3 + d_2 \\ f_1 &= d_3 + d_1 \end{aligned}$$

Prioriteitsencoder = encoder waarvan meerdere ingangen 1 mogen zijn

| d_3 | d_2 | d_1 | d_0 | any | f_0 | f_1 |
|-------|-------|-------|-------|-----|-------|-------|
| 0 | 0 | 0 | 0 | 0 | – | – |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | – | 1 | 0 | 1 |
| 0 | 1 | – | – | 1 | 1 | 0 |
| 1 | – | – | – | 1 | 1 | 1 |

$$\begin{aligned} \text{any} &= d_3 + d_2 + d_1 + d_0 \\ f_0 &= d_3 + d_2 \\ f_1 &= d_3 + d_2'd_1 \end{aligned}$$

Combinatorische schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- Demux
- ⇒ Encoder
- Vergelijken
- Schuiven

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- ➔ Andere basisschakelingen
 - Multiplexer
 - Decoder & demultiplexer
 - Encoder
 - ➔ Vergelijken
 - Schuifoperaties
- Combinatorische schakelingen in VHDL

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

Andere schakelingen

- MUX
- Decoder
- Demux
- Encoder
- ⇒ Vergelijken
- Schuiven

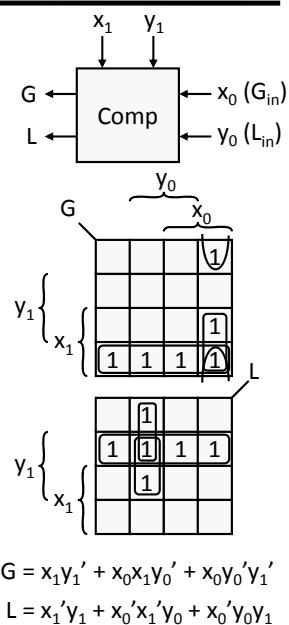
VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

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 |



Comparators cascaderen

Combinatorische schakelingen

Minimalisering

Rekenschakelingen

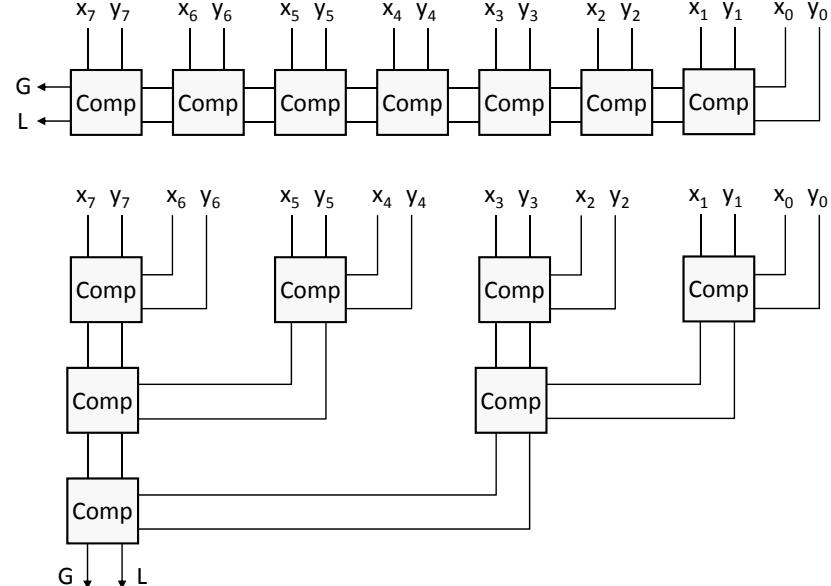
Andere schakelingen

- MUX
- Decoder
- Demux
- Encoder
- ⇒ Vergelijken
- Schuiven

VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

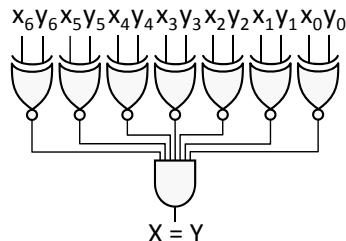


Speciale gevallen

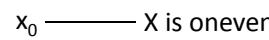
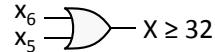
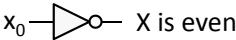
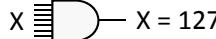
Testen op gelijkheid

➤ Gebruik XNOR per bit

Dit veronderstelt dat X en Y 7-bit getallen zijn.



Vergelijken met constanten



Combinatorische schakelingen

Minimalisering van logische functies

Rekenkundige basisschakelingen

Andere basisschakelingen

➤ Multiplexer

➤ Decoder & demultiplexer

➤ Encoder

➤ Vergelijken

➤ Schuifoperaties

Combinatorische schakelingen in VHDL

KATHOLIEKE UNIVERSITEIT

LEUVEN

Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

- MUX
 - Decoder
 - Demux
 - Encoder
 - Vergelijken
 - ⇒ Schuiven
- VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Schuifoperaties

- m posities schuiven: links ‘ \ll ’, rechts ‘ \gg ’

➤ Nieuwe bits

- Logisch schuiven
 - Een bit van een bijkomende ingang
- Aritmetisch schuiven
 - Links schuiven: m nullen rechts erbij
 - Rechts schuiven: m tekenbits links erbij
(MSB voor 2-complement, 0 voor ‘unsigned’)

$$\triangleright i \ll m \equiv i \times 2^m$$

$$i \gg m \equiv i \div 2^m$$

- m posities roteren, links of rechts

- De bits die aan de ene kant afvallen worden er aan de andere kant bijgezet

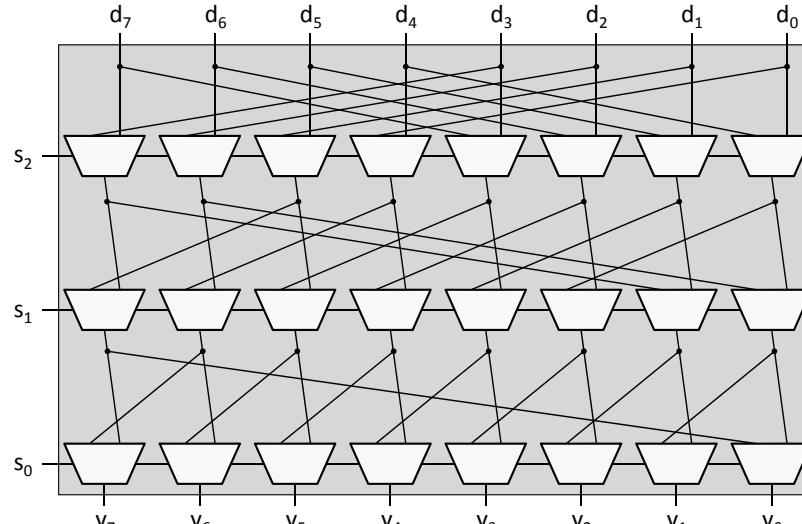
Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

- MUX
 - Decoder
 - Demux
 - Encoder
 - Vergelijken
 - ⇒ Schuiven
- VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

‘8-bit barrel left rotator’



Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

- MUX
 - Decoder
 - Demux
 - Encoder
 - Vergelijken
 - ⇒ Schuiven
- VHDL

KATHOLIEKE UNIVERSITEIT
LEUVEN

Schuifoperaties over 1 bit

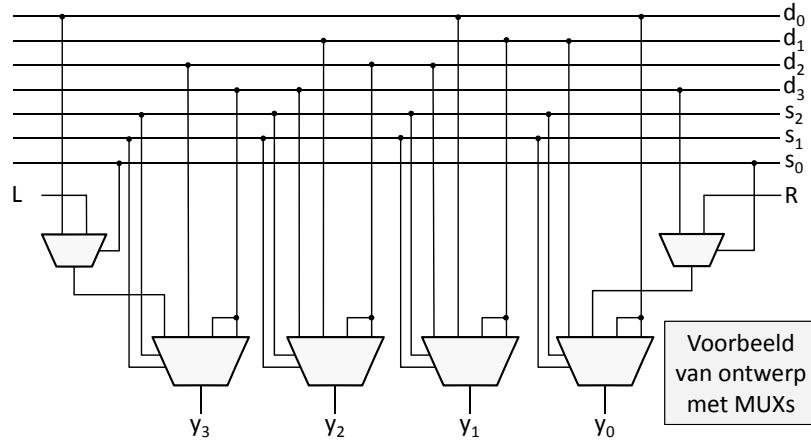
Minimalisering
Rekenschakelingen
Andere schakelingen

- MUX
 - Decoder
 - Demux
 - Encoder
 - Vergelijken
 - ⇒ Schuiven
- VHDL

$$R = \text{arit}' \cdot R_{in}$$

$$L = \text{arit}' \cdot L_{in} + \text{arit} \cdot 2\text{comp} \cdot d_3$$

| | 0 | 1 |
|-------|---------------------|----------------|
| s_2 | geen schuifoperatie | schuifoperatie |
| s_1 | links | rechts |
| s_0 | schuiven | roteren |



KATHOLIEKE UNIVERSITEIT
LEUVEN

Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

- VHDL
- Taal
- Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
 - Elementen van de VHDL-taal
 - Lexicale elementen (woordenschat)
 - Data-objecten & -types
 - Bibliotheken
 - Bewerkingen
 - Hardwarebeschrijving met VHDL

Combina-
torische
schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen
VHDL

⇒ Taal
⇒ woordenschat
• object
• type
• vector
• logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Lexicale elementen van VHDL

- Commentaar: van “--” tot einde lijn
- ‘Identifier’ (of naam)
 - = reeks van alfanumerische karakters of niet-opeenvolgende “_”, die start met een letter en niet eindigt met “_”: “Next_value_0”
 - Geen verschil hoofdletters / kleine letters!
- Getal
 - ‘integer literal’: geheel getal “1480”
 - ‘real literal’: fractioneel getal “1480.0”
 - Beide kunnen exponentieel “148E1”
 - “_” wordt genegeerd “1_480”
 - Niet-decimale basis: base#literal#exp
 $253.5 = 16\#FD.8\# \quad 2\#1\#E10 = 16\#4\#E2$

Combina-
torische
schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen
VHDL

⇒ Taal
⇒ woordenschat
• object
• type
• vector
• logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
 - ➔ Elementen van de VHDL-taal
 - Lexicale elementen
 - ➔ Data-objecten & -types
 - Bibliotheken
 - Bewerkingen
 - ➔ Hardwarebeschrijving met VHDL

Combina-
torische
schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen
VHDL

⇒ Taal
⇒ woordenschat
• object
• type
• vector
• logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Lexicale elementen van VHDL

- Karakter
 - ‘character literal’ = één karakter tussen enkele aanhalingstekens: ‘a’ ‘A’ ‘ ’
- Karakterreeks (‘string’)
 - ‘string literal’ = reeks karakters tussen dubbele aanhalingstekens: “A string”
 - “” om “” in een string te krijgen: “”“Quote it””, she said.”
- Bitreeks (‘bit string’)
 - ‘bit string literal’ = reeks bits, voorgesteld door een reeks cijfers voorafgegaan door een basisspecificatie (‘B’, ‘O’, of ‘X’): O"12" = b"001_010" ⇔ X"a" = B"1010"

Combina-
torische
schakelingen

Minimalisering
Rekenschakelingen

Andere schakelingen
VHDL

⇒ Taal
⇒ woordenschat
• object
• type
• vector
• logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

VHDL-objecten

- = benoemd item met een waarde van een specifiek type
- Constante
 - declaratie **constant** name(s): (sub)type := expression;
 - maakt programma beter verstaanbaar
 - constant num_bytes: integer := 4;**
 - constant num_bits: integer := 8 * num_bytes;**
- Variabele
 - declaratie **variable** name(s): (sub)type [:= expression]; initiële waarde
 - bevat tussenresultaten zonder fysische betekenis
- Signaal
 - declaratie **signal** name(s): (sub)type [:= expression];
 - draad, (interne) verbinding
 - toekenning **y <= a and b;** y — a
 - golfvorm, zoals zichtbaar tijdens een simulatie
 - toekenning **pulse <= '1', '0' after T_pw;**

VHDL-types en -subtypes

□ VHDL-types

➢ Scalaire types: set waarden

- gehele getallen: integer (minstens 32 bit)

`range integer_expression (down)to integer_expression`
- opsommingen ('enumeration')

`type bit is ('0','1');`
`type boolean is (false, true);`
- komma-getallen: real (minstens IEEE 32 bit)
- fysische waarden: time

namen of
karakters

➢ Samengestelde types: verzameling sets

- vectoren en matrices

□ Subtype: beperkte set waarden van basistype

```
subtype natural is integer
  range 0 to integer'high;
subtype positive is integer
  range 1 to integer'high;
```

integer'high : attribuut high
(hoogste waarde) van integer
(attribuut = informatie over object/type)

Matrices & vectoren in VHDL

□ Begrensd ('constrained'): grenzen indices vast

```
array (range [, range]...) of (sub)type
met range ofwel een discreet subtype ofwel
expression (down)to expression
```

MSB
LSB, bit 0

➢ type word is array (15 downto 0) of bit;

□ Onbegrensd: grenzen niet bepaald

```
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);
  variable sample_buf: sample(0 to 63);

➢ type string is
  array (positive range <>) of character;
  constant Error_message: string
    := "Unknown error: ask for help";
```

Fysische types

range expression (down)to expression units

```
identifier;
[identifier = physical_literal];...
end units
```

Primaire eenheid:
nauwkeurigheid

Secundaire eenheden

bijv. 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;
```

Primaire eenheid:
nauwkeurigheid

Secundaire eenheden

'Array literal'

□ '(Bit)string literal'

- variable w: word := "1010000101111111";
- variable w: word := x"A17F";

□ Matrixgeheel ('array aggregate')

- Associatie volgens positie
(expression [, expression]...)
- type point is array (1 to 3) of integer;

variable p: point := (4, 5, 5);
- Associatie volgens naam
(choice [| choice]... => expression
[, choice [| choice]... => expression]...)
- met choice ofwel een uitdrukking, een discreet bereik
of 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); init onafh. van grootte

Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

VHDL
⇒ Taal
• woordenschat
• object
• type
• vector
⇒ logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Scalaire logische waarden in VHDL

= waarde op één enkele draad

- **bit** (voorgedefinieerd)
 - > `type bit is ('0','1');`
- **std_logic** (IEEE 1164)
 - > `type std_ulogic is ('U', -- niet geïnitialiseerd (opstarten), 'X', -- sterk aangestuurd ongekend, '0', -- sterk aangestuurd logisch 0, '1', -- sterk aangestuurd logisch 1, 'Z', -- hoog-impedant (niet aangestuurd), 'W', -- zwak aangestuurd ongekend, 'L', -- zwak aangestuurd logisch 0, 'H', -- zwak aangestuurd logisch 1, '-'); -- don't care (kan 0 of 1 zijn)`
 - `type std_logic is resolved std_ulogic;`
 - > **Gebruik std_logic i.p.v. bit voor echte toepassingen!**

actieve aansturing
(bv. CMOS)

resistieve aansturing
(bv. NMOS)

Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

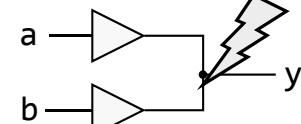
VHDL
⇒ Taal
• woordenschat
• object
• type
• vector
⇒ logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

std_ulogic ↔ std_logic

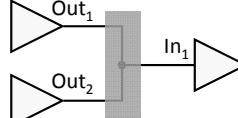
VHDL laat slechts enkelvoudige toekenningen toe

- > `y <= a; y <= b;`
kortsluiting als $a = b'$



⇒ Voeg een resolutiefunctie toe aan de signaldefinitie om de eigenlijke waarde te berekenen uit alle aangelegde waarden

- > `function resolved (s: std_ulogic_vector)`
`return std_ulogic;`
- `type std_logic is resolved std_ulogic;`
- > Voor inout poorten (= poorten die in en out zijn):
 - gebruik als uitgang ⇒ ingang resolutiefunctie
 - gebruik als ingang ⇒ uitgang resolutiefunctie



Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

VHDL
⇒ Taal
• woordenschat
• object
• type
• vector
⇒ logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Gebundelde logische waarden

Bundel draden

= vector van scalaire logische waarden

- **bit_vector** (voorgedefinieerd)
 - > `type bit_vector is array (natural range <>) of bit;`
 - `constant StateA: bit_vector(0 to 3) := "0100";` StateA[1] is '1'
- **std_logic_vector** (IEEE 1164)
 - > `type std_logic_vector is array (natural range <>) of std_logic;`

Opmerking: matrices kunnen slechts aan elkaar toegekend worden als ze dezelfde dimensies en grootte hebben.

> Correspondentie via positie, niet via index!

```
signal Down: std_logic_vector (3 downto 0);
signal Up: std_logic_vector (0 to 3);
Up <= Down; Up[i] is Down[3-i]
```

KATHOLIEKE UNIVERSITEIT
LEUVEN

Combinatorische schakelingen

Minimalisering
Rekenschakelingen
Andere schakelingen

VHDL
⇒ Taal
• woordenschat
• object
• type
• vector
⇒ logische waarde
• bibliotheek
• bewerkingen
▪ Hardware beschrijving

KATHOLIEKE UNIVERSITEIT
LEUVEN

Std_logic_1164 resolutiefunctie

```
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', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- 'X'
 ('U', 'X', '0', 'X', '0', '0', '0', 'X'), -- '0'
 ('U', 'X', 'X', '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')) -- '-'
```

```
function resolved(s : std_ulogic_vector)
  return std_ulogic;
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;
```

`s'range`: indexbereik van `s`
`s'length`: grootte van indexbereik
`s'low`: kleinste waarde

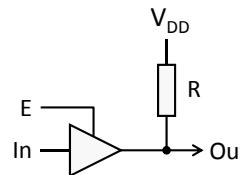
Voorbeeld: afgesloten bus

```
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 aansturing

actieve aansturing

Gebruik van bibliotheken

Evolutionair ontwerp: dikwijls kan tot 95% van een ontwerp hergebruikt worden

- Een 'Package' groepeert definities
 - Een 'Library' is de plaats waar de binaire code van analyse/compilatie gestockeerd wordt (folder, databank, ...)
- Default: work (huidige werkfolder)

Gebruik:

```
library library_name;
use library_name.package_name.all;

library ieee; use ieee.std_logic_1164.all;
```

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
 - ➔ Elementen van de VHDL-taal
 - Lexicale elementen (woordenschat)
 - Data-objecten & -types
 - ➔ Bibliotheken
 - Bewerkingen
 - Hardwarebeschrijving met VHDL

'Overloading'

- = laat verschillende subprogramma's (function of procedure) toe met dezelfde naam maar een verschillende datatypes voor de argumenten
 - ➔ context bepaalt welke gebruikt wordt
 - procedure incr(a : inout integer) is ...
 - procedure incr(a : inout bit_vector) is ...
- Om operatoren te 'overloaden', plaats ze tussen aanhalingstekens
 - function "+" (a,b: in bit_vector)

return bit_vector is ...
 - function "+" (a: in bit_vector, b: in integer)

return bit_vector is ...

(Meestal) aanwezige bibliotheken

Synopsis (de facto standaard)

- std_logic_1164
 - definitie & logische bewerkingen met std_logic[_vector]
- std_logic_arith
 - rekentijdige bewerkingen op [un]signed met als resultaat [un]signed of std_logic_vector
- std_logic_[un]signed
 - rekentijdige bewerkingen op std_logic_vector

IEEE

- 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

Maakt gebruik van 'overloading'!

Logische bewerkingen

- Lijst van logische operatoren: **not, and, or, xor, nand, nor, xnor**
- Prioriteit:
 - 'not' heeft de hoogste prioriteit
 - Alle andere hebben gelijke prioriteit lager dan 'not'
- Gedefinieerd voor de datatypes: **bit[_vector], boolean, std_logic[_vector]**
- Kan op matrices van dezelfde grootte
 - Bewerkingen gebeuren telkens op elementen met overeenkomende posities
- Resultaat: hetzelfde datatype als operanden

niet in VHDL-87

Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
 - ➔ Elementen van de VHDL-taal
 - Lexicale elementen (woordenschat)
 - Data-objecten & -types
 - Bibliotheken
 - ➔ Bewerkingen:
 - logisch, rekenkundig, schuiven & vergelijken
 - Hardwarebeschrijving met VHDL

Rekenkundige bewerkingen

- Lijst van rekenkundige operatoren: +, -, *, /, ** (exponent), abs (absolute waarde), mod (modulus), rem (rest)
- Gedefinieerd voor
 - datatypes integer en real (behalve mod en rem)
 - Gebruik 'overloading' voor bitvectoren (via IEEE-bibliotheek, niet standaard)
 - fysische datatypes (enkel +, -)
- Operanden moeten van hetzelfde type zijn, maar verschillende bereiken zijn 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 van het fysische type

Schuifoperaties

- Lijst van schuifoperatoren (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, schuif 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"
```

Specifieke bewerkingen op matrices

- Aaneenschakeling ('concatenate')
 - = samenvoeging door draden te bundelen
 - ```
signal Bus: bit_vector(7 downto 0);
signal NibbleA, NibbleB:
 bit_vector(3 downto 0);
Bus <= NibbleA & NibbleB; Bus(7) <= NibbleA(3)
 Bus(3) <= NibbleB(3)
```
- Matrixdeel ('slice')
  - = subset van opeenvolgende matrixelementen
  - ```
signal A: bit_vector (0 to 3); Bus(5) <= A(0)
                           Bus(5 downto 4) <= A(0 to 1); Bus(4) <= B(1)
```
 - Zorg ervoor dat de richting (**to** of **downto**) dezelfde is als in de declaratie!
 - ```
Bus(5 downto 4) <= A(1 downto 0); Fout !
```

# Vergelijkingen

- Lijst van relationele operatoren:
  - <, <=, =>, >, =, /=
- Beide operanden moeten van hetzelfde type zijn
- Resultaat is een boolean
- Kan op matrices, zelfs van verschillende grootte
  - Algoritme:
    - aligneer de matrices op linkerelement
    - vergelijk element per element, van links naar rechts
    - vergelijk maximaal zoveel elementen als er in de kleinste matrix aanwezig zijn
  - Daarom zijn de volgende vergelijkingen waar:
    - "1110" > "10111"
    - "1110" = "11101"
  - Dit werkt dus op bitvectoren van gelijke lengte *alsof* het positieve getallen waren

# Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
  - Elementen van de VHDL-taal
  - ➔ Hardwarebeschrijving met VHDL

Combina-  
torische  
schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
  - structuur
  - gedrag
  - repetitief

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# VHDL-entiteit

- Interface module = declaratie entiteit
 

```
entity name is
 [generic (generic_list);]
 [port (port_list);]
end [entity] [name];
```

generische constanten:  
verschillen per gebruik

interface naar entiteit
- Implementatie module
 

```
in, out, inout
```

= één of meerdere architecturen, die alternatieve implementaties beschrijven

```
architecture name of entity_name is
 [declaration]...
 [subprogram]...
begin
 {[label :] concurrent_statement}...
end [architecture] [name];
```

constante, variabele, signaal, type, attribuut, component

functie, procedure

benoem de uitdrukking/component; voor debugging/simulatie & configuratie

Combina-  
torische  
schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
  - ⇒ structuur
  - gedrag
  - repetitief

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Combinatorische schakelingen

- Minimalisering van logische functies
- Rekenkundige basisschakelingen
- Andere basisschakelingen
- ➔ Combinatorische schakelingen in VHDL
  - Elementen van de VHDL-taal
  - Hardwarebeschrijving met VHDL
    - ➔ Structurele beschrijving
    - Gedragsbeschrijving
    - Beschrijving van repetitieve structuren

Combina-  
torische  
schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
  - structuur
  - gedrag
  - repetitief

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Parallelle uitdrukkingen

- Structurele beschrijving: beschrijf de hiërarchie van componenten, als verbindingen tussen subsystemen
 

```
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;
```
- Gedragsbeschrijving (van een component): parallelle signaaltoekenningen & "process"
 

Opmerking: Alle *concurrent\_statements* worden gelijktijdig uitgevoerd, zoals we van hardware verwachten  
⇒ volgorde is dus onbelangrijk

Combina-  
torische  
schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
  - ⇒ structuur
  - gedrag
  - repetitief

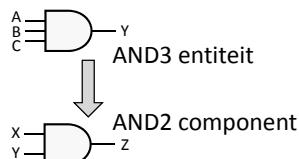
KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Structurele beschrijving: 2-naar-1 MUX

- ```
architecture Struct of MUX2 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 (U,B,V);
  Gate4: component OR2 port map (X=>W,Y=>V,Z=>Y);
end architecture Struct;
```
- component: virtueel element
 - ⇒ entiteit: reëel element
 - laat 'top-down' ontwerp toe
- in een bibliotheek bij meervoudig gebruik
- positioneel of op naam
-

Configuratie: 2-naar-1 MUX

- = koppeling tussen component en entiteit



- Stel slechts AND3, OR3 en INV in bibliotheek

```
configuration Use3InputGates of MUX2 is
  for Struct
    for Gate1:INV use entity INV(RTL)
      port map (A=>X, Y=>Z);
    end for;
    for all1: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;
```

implementatie RTL van entiteit AND3 (uit de bibliotheek)

Parallelle signaaltoekenningen

- Toekenning van conditionele signalen

```
name <= [waveform when boolean_expr else]...
          waveform [when boolean_expr];
```

➤ bijv. 2-naar-1 MUX

```
f <= d1 when s = '1' or s = 'H' else [ingebouwde prioriteit]
d0 when s = '0' or s = 'L' else 'X';
```

- Toekenning van geselecteerde signalen

```
with expression select [wordt slechts eenmaal geëvalueerd]
  name <= [waveform when choice(s) ,]...
            waveform when choice(s);
```

➤ bijv. 2-naar-1 MUX

```
with s select
  f <= d1 when '1' | 'H',
  d0 when '0' | 'L',
  'X' when others;
```

- Alle mogelijke waarden moeten exact eenmaal gespecificeerd worden.
- Waarden zijn constant en gekend op het moment van ontwerp.

Combinatorische schakelingen

- Minimalisering van logische functies

- Rekenkundige basisschakelingen

- Andere basisschakelingen

→ Combinatorische schakelingen in VHDL

➤ Elementen van de VHDL-taal

→ Hardwarebeschrijving met VHDL

- Structurele beschrijving
- Gedragsbeschrijving
- Beschrijving van repetitieve structuren

Rekenkundige hardware: een voorbeeld

Gebruik de gepaste IEEE-bibliotheek.

```
library ieee;
use ieee.std_logic_signed.all;

entity adder is
  generic(n : positive := 4);
  port(Cin : in std_logic;
        X,Y : in std_logic_vector(n-1 downto 0);
        S : out std_logic_vector(n-1 downto 0);
        Cout, Overflow : out std_logic);
end adder;

architecture behav of adder is
  signal Sum : std_logic_vector(n downto 0);
begin
  Sum <= (X(n-1) & X) + Y + Cin;
  S <= Sum(n-1 downto 0);
  Cout <= Sum(n);
  Overflow <= Sum(n) xor X(n-1) xor Y(n-1) xor Sum(n-1);
end behav;
```

Kracht van VHDL:
deze definitie is bruikbaar voor alle mogelijke bitbreedtes (default breedte = 4)

minstens 1 operand heeft n+1 bits

$c_n \oplus c_{n-1}$

Complexe gedragsbeschrijving: het “process”

- = programma van sequentiële uitdrukkingen dat één parallelle uitdrukking vormt
- process**(*signal_name(s)*) /is/ gevoelighedslijst
- [constant, variable, (sub)type declaration]...*
[subprogram]...
- begin**
 sequential_statement(s) volgorde is belangrijk
- end process** [*label*];
- ⇒ Voer alle *sequential_statement(s)* één na één uit telkens er één of meer van de (ingangs)signalen *signal_name(s)* van waarde verandert.
- Sequentiële uitdrukkingen:
- toekenningen van variabelen (“:=”) en signalen (“<=”)
 - conditionele uitdrukkingen: “if”, “case”
 - lussen: “loop”, “while”, “for”

Variabelen versus signalen

- Verschil toekenning waarde:
- **process(x)** cnt = aantal enen in x
- **process(x)**
begin
 cnt <= 0;
 for i in 1 to 3 loop
 if x(i) = '1' then
 tmp := tmp + 1;
 end if;
 end loop;
 cnt <= tmp;
end process;
- **process(x)**
begin
 cnt <= 0;
 for i in 1 to 3 loop
 if x(i) = '1' then
 cnt <= cnt + 1;
 end if;
 end loop;
end process;
- cnt = oude waarde + 1
als x minstens een 1 bevat;
anders cnt = 0
- Het volgende kan niet als combinatorische schakeling:
- **process(cnt)**
begin
 cnt <= cnt + 1;
end process;
-

Variabelen versus signalen

- Signalen hebben gewoonlijk een fysische betekenis, variabelen niet noodzakelijk; maar golfvormen kunnen enkel bij 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.
 - Een variabele wordt dadelijk aangepast; een signaal wordt aangepast aan het einde van een proces met gevoelighedslijst.
- ```
> v := '1';
 v := '0';
 if v = '0' then
 -- gebeurt altijd
 end if;
```

```
> s <= '1';
 s <= '0';
 if s = '0' then
 -- s was 0 bij begin proces
 end if;
```

# Conditionele uitdrukkingen

- “if”-uitdrukking
- ```
process(d0,d1,s) begin
  if s = '1' then
    f <= d1;
  elsif s = '0' then
    f <= d0;
  else f <= 'X';
  end if;
end process;
```

≡

```
f <= d1 when s = '1' else
d0 when s = '0' else
'X';
```
- “case”-uitdrukking
- ```
process(d0,d1,s) begin
 case s is
 when '1' =>
 f <= d1;
 when '0' =>
 f <= d0;
 when others =>
 f <= 'X';
 end case;
end process;
```

≡

```
with s select
 f <= d1 when '1',
 d0 when '0',
 'X' when others;
```

# Lussen

## □ Oneindige lus

```
loop
 statement(s)
end loop;
```

Extra uitdrukkingen:  
“exit” en “next”

➤ Oneindige lussen zijn typisch voor hardware!

## □ “while”-lus

```
while boolean_expression loop
 statement(s)
end loop;
```

## □ “for”-lus

```
for name in range loop
 statement(s)
end loop;
```

De lusvariabele *name*

- moet niet gedeclareerd worden
- kan alleen binnen de lus gebruikt worden

# 2-naar-4 decoder

```
library ieee; use ieee.std_logic_1164.all;
entity dec2to4 is
 port (w : in std_logic_vector(1 downto 0);
 en : in std_logic;
 y : out std_logic_vector(0 to 3));
end dec2to4;
architecture behav of dec2to4 is
begin
 process (w, en) begin
 y <= "XXXX";
 if (en = '1') or (en = 'H') then
 case To_X01(w) is
 when "00" => y <= "1000";
 when "01" => y <= "0100";
 when "10" => y <= "0010";
 when "11" => y <= "0001";
 when others => null;
 end case;
 elsif (en = '0') or (en = 'L') then
 y <= "0000";
 end if;
 end process;
end behav;
```

# Combinatorische schakelingen

## □ Minimalisering van logische functies

## □ Rekenkundige basisschakelingen

## □ Andere basisschakelingen

➔ Combinatorische schakelingen in VHDL

➤ Elementen van de VHDL-taal

➔ Hardwarebeschrijving met VHDL

- Structurele beschrijving

- Gedragsbeschrijving

➔ Beschrijving van repetitieve structuren

# Parallelle uitdrukking “generate”

## □ Genereren iteratieve structuren:

herhaal identieke cellen

```
for identifier in range generate
 [declaration(s)]
begin
 {[label :] concurrent_statement}...
end generate [this_label];
```

## □ Structuren conditioneel genereren:

behandel sommige cellen anders

```
if boolean_expression generate
 [declaration(s)]
begin
 {[label :] concurrent_statement}...
end generate [this_label];
```

Opmerking: concurrent\_statement kan ook een generate zijn!

Combinatorische schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
- structuur
- gedrag
- ⇒ repetitief

# Vermenigvuldiger (1)

---

```

library ieee;
use ieee.std_logic_unsigned.all;

entity umult is
 generic (n, m : positive);
 port (X : in std_logic_vector(n-1 downto 0);
 Y : in std_logic_vector(m-1 downto 0);
 P : out std_logic_vector(n+m-1 downto 0));
end umult;

architecture behav of umult is
begin
 P <= X * Y;
end behav;

```

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

Combinatorische schakelingen

Minimalisering  
Rekenschakelingen  
Andere schakelingen

VHDL

- Taal
- ⇒ Hardware beschrijving
- structuur
- gedrag
- ⇒ repetitief

# Vermenigvuldiger (2)

---

```

architecture struct of umult is
 component AND2
 port (i0, i1 : in std_logic; o : out std_logic);
 end component;
 signal PC : array(0 to m-1) of std_logic_vector(n-1 downto 0);
 signal PS : array(0 to m-2) of std_logic_vector(n downto 0);
begin
 for i in 0 to m-1 generate
 for j in 0 to n-1 generate
 component AND2 port map (X(j), Y(i), PC(i)(j));
 end generate;
 if i = 0 generate
 PS(0) <= '0' & PC(0);
 P(0) <= PC(0)(0);
 end generate;
 if (i > 0) and (i < m-1) generate
 PS(i) <= ('0' & PS(i-1)(n downto 1)) + PC(i);
 P(i) <= PS(i)(0);
 end generate;
 if i = m-1 generate
 P(n+m-1 downto m-1) <= ('0' & PS(i-1)(n downto 1)) + PC(i);
 end generate;
 end generate;
end struct;

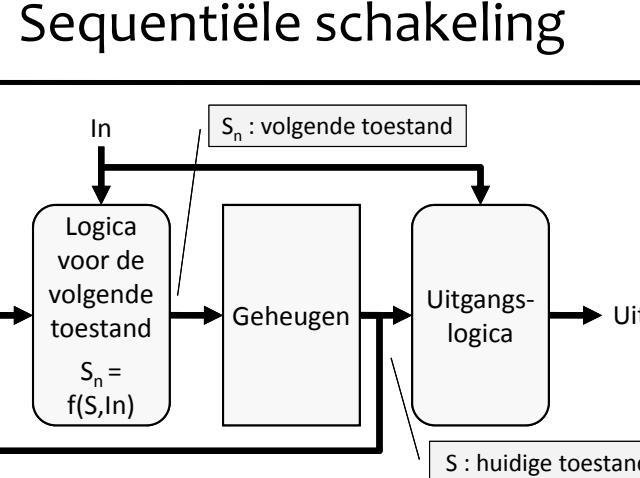
```



KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Inhoudstafel

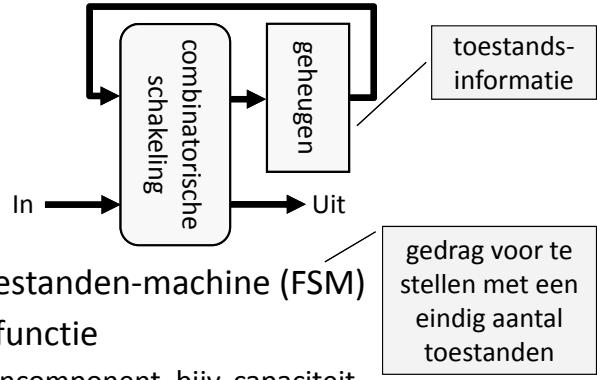
- Inleiding
- De basis van digitaal ontwerp
- Technologische randvoorwaarden
- Combinatorische schakelingen
- ➔ Sequentiële schakelingen:  
ook een functie van de voorgeschiedenis
- Niet-programmeerbare processoren
- Programmeerbare processoren



- Toestandgebonden (Moore):  
uitgang =  $f(\text{toestand})$
- Inputgebonden (Mealy):  
uitgang =  $f(\text{toestand}, \text{ingangen})$

# Sequentiële schakeling

= uitgangen zijn een functie van de ingangen en de toestand (m.a.w. ook functie van de vorige ingangen)



= eindige-toestanden-machine (FSM)

□ Geheugenfunctie

- Geheugengenerator, bijv. capaciteit
  - Ladingverlies ⇒ dynamische logica
- Positieve terugkoppeling: flipflop (of register)

# Sequentiële schakeling

# Sequentiële schakeling

□ Asynchrone schakeling: uitgangen & toestand veranderen wanneer een ingang verandert

□ Synchrone schakeling: uitgangen & toestand veranderen alleen wanneer de kloksignaal verandert

Beschrijving kloksignaal:

- Kloperiode = tijd tussen twee opeenvolgende  $1 \rightarrow 0$  klokovergangen
- Klofrequentie =  $1/\text{kloperiode}$
- 'Duty cycle' =  $(\text{tijd klok is } 1)/\text{kloperiode}$
- Stijgende flank ('rising edge') =  $0 \rightarrow 1$  klokovergang
- Dalende flank ('falling edge') =  $1 \rightarrow 0$  klokovergang

## Sequentiële schakelingen

### Bouwblokken

- Flipflop
  - Register
  - Teller
  - VHDL
- Synchroon Asynchroon*
- Latch
  - Flipflop
  - Types flipflops

### Registers

- Tellers
- In VHDL

### Ontwerp van synchrone sequentiële schakelingen

### Ontwerp van asynchrone sequentiële schakelingen

## SR-latch met NAND-poorten

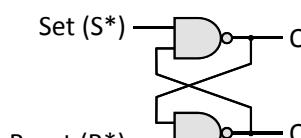
## Sequentiële schakelingen

### Bouwblokken

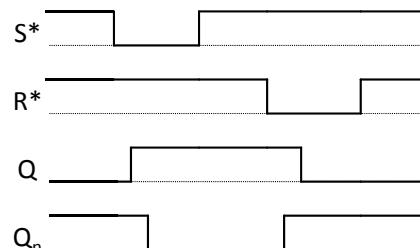
- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

*Synchroon Asynchroon*



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



Set en Reset zijn actief-lage signalen

## Ontwerp sequentiële schakelingen

### Bouwblokken

#### De flipflop

- Latch
- Flipflop
- Types flipflops

#### Registers

- Tellers
- In VHDL

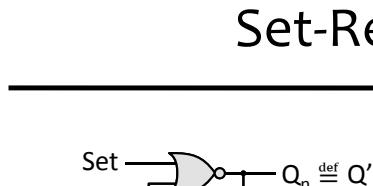
## Sequentiële schakelingen

### Bouwblokken

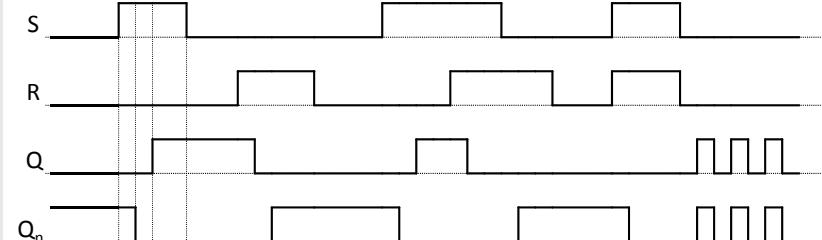
- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

*Synchroon Asynchroon*



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



'Race' : uiteindelijke waarde afhankelijk van de implementatie en poortvertragingen

- Oscillatie wanneer vertraging van poorten identiek
- Anders bepaalt de snelste poort de uiteindelijke waarde

Ongedefinieerd

## Geklokte SR-latch

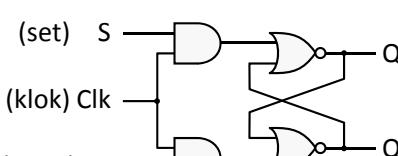
## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

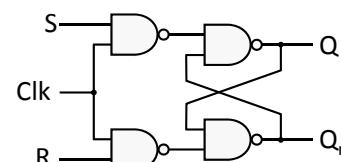
*Synchroon Asynchroon*



Clk = 1 : volg ingangen

Clk = 0 : behoud uitgang

| Clk | S | R | Q(next) |
|-----|---|---|---------|
| 0   | - | - | Q       |
| 1   | 0 | 0 | Q       |
| 1   | 0 | 1 | 0       |
| 1   | 1 | 0 | 1       |
| 1   | 1 | 1 | NA      |



## Sequentiële schakelingen

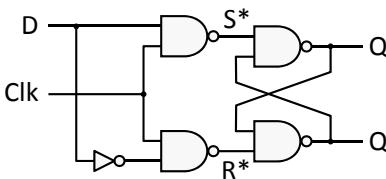
### Bouwblokken

- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

Synchroon  
Asynchroon

## Geklokte D-latch



| Clk | D | Q(next) |
|-----|---|---------|
| 0   | - | Q       |
| 1   | 0 | 0       |
| 1   | 1 | 1       |

### Vertragingen

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

## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

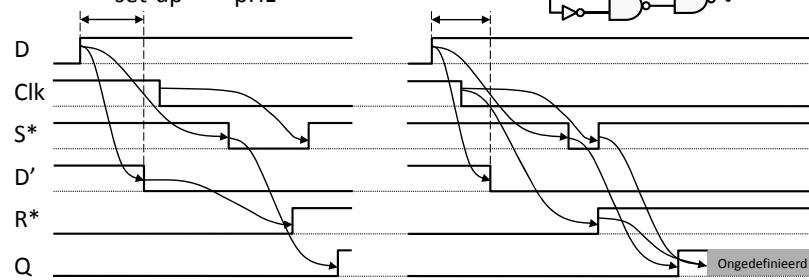
Synchroon  
Asynchroon

## Set-up/houdtijd

- Set-up-tijd = tijd voor actieve klokflank waarin ingangen niet mogen veranderen

➢ Bijv. geklokte D-latch

$$t_{set-up} = t_{pHL}(\text{inverter})$$



- Houdtijd = tijd na actieve klokflank waarin ingangen niet mogen veranderen

## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

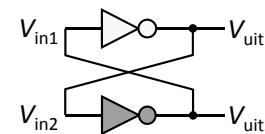
- Register
- Teller
- VHDL

Synchroon  
Asynchroon

## Metastabiliteit

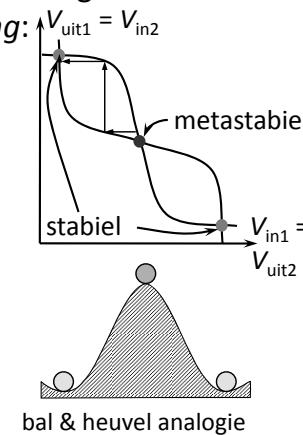
### Bi-stabiel element:

- 2 stabiele toestanden
- 1 metastabile toestand



### Element kan in metastabile toestand gebracht worden door *marginale triggering*:

- schending minimum pulsbreedte
- schending set-up/houdtijd



### $p_{\text{nog in meta}}(t) = \exp(-t/\tau)$ waarbij $\tau$ afhangt van

- de hoeveelheid ruis
- de steilheid van de curve

Geen probleem als signaal niet (meer) metastabiel bij gebruik!

## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
- ⇒ latch
- flipflop
- types

- Register
- Teller
- VHDL

Synchroon  
Asynchroon

## Ontwerp sequentiële schakelingen

### → Bouwblokken

#### → De flipflop

- Latch
- Flipflop
- Types flipflops

#### → Registers

#### → Tellers

#### → In VHDL

### □ Ontwerp van synchrone sequentiële schakelingen

### □ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

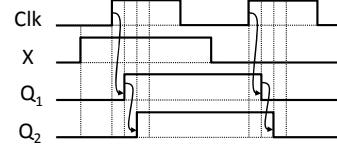
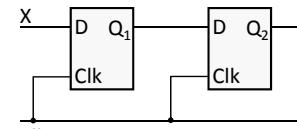
- ⇒ Flipflop
  - latch
  - ⇒ flipflop
  - types

- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# Gevoeligheid

- Alle voorgaande geklokte latches zijn gevoelig aan het niveau van de klok:
  - klok = 1 : transparant
  - klok = 0 : onthoud de laatste waarde
- Transparant zijn kan problemen geven als meerdere latches mekaar opvolgen (bijv. schuifregisters)
  - het ingangssignaal kan door meerdere latches doerrimpelen in 1 klokperiode
  - daardoor is het ook moeilijk om aan de set-up/houdtijd te voldoen
- Oplossing:  
flankgevoelig werken
  - Master-slave flipflop
  - Edge-triggered flipflop



## Sequentiële schakelingen

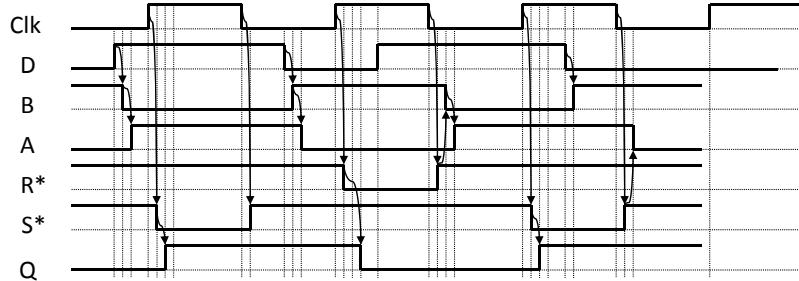
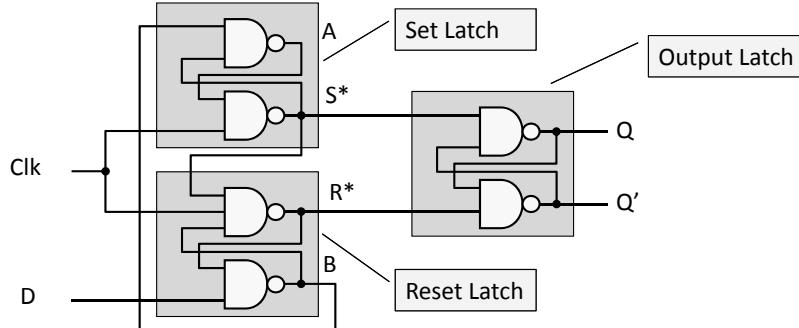
### Bouwblokken

- ⇒ Flipflop
  - latch
  - ⇒ flipflop
  - types

- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# Edge-triggered flipflop



## Sequentiële schakelingen

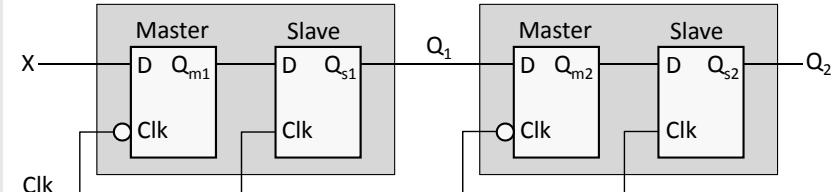
### Bouwblokken

- ⇒ Flipflop
  - latch
  - ⇒ flipflop
  - types

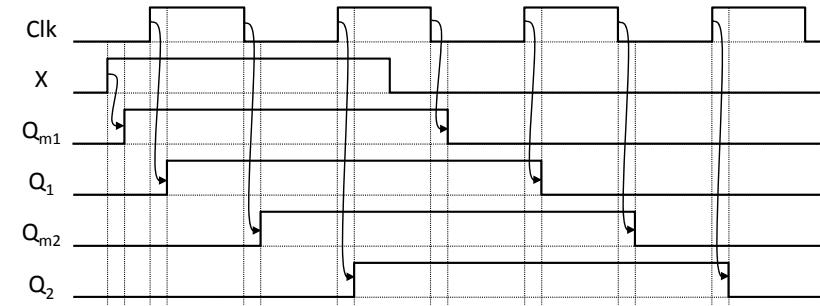
- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# Master-slave flipflop



Master : transparant als Clk = 0  
Slave : transparant als Clk = 1 } ⇒ werkt op een stijgende flank



## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
  - latch
  - ⇒ flipflop
  - types

- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# Asynchrone set & reset

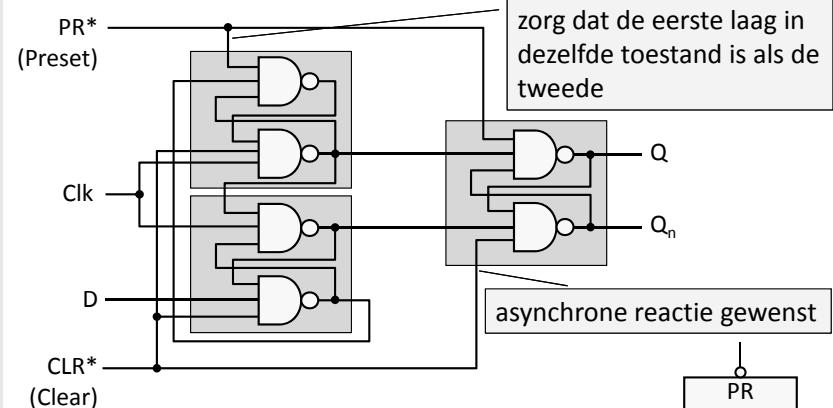
### Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
  - latch
  - ⇒ flipflop
  - types

- Register
- Teller
- VHDL

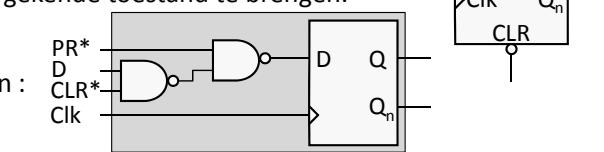
Synchroon  
Asynchroon



zorg dat de eerste laag in dezelfde toestand is als de tweede  
asynchrone reactie gewenst

Asynchrone set & reset zijn nuttig om de flipflop in 't begin in een gekende toestand te brengen.

Anders synchroon :



## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
  - latch
  - flipflop

- ⇒ types
- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# Ontwerp sequentiële schakelingen

## → Bouwblokken

### → De flipflop

- Latch
- Flipflop
- Types flipflops

### ➤ Registers

- Tellers
- In VHDL

- Ontwerp van synchrone sequentiële schakelingen
- Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

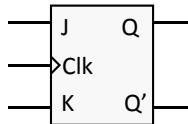
- ⇒ Flipflop
  - latch
  - flipflop

- ⇒ types
- Register
- Teller
- VHDL

Synchroon  
Asynchroon

# JK-flipflop

### Symbool



### Karakteristieke tabel

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

Realiseerbaar met SR-flipflop:

$$S = JQ' \text{ en } R = KQ$$

### Excitatietabel

| Q | Q(next) | J | K |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | - |
| 1 | 0       | - | 1 |
| 1 | 1       | - | 0 |

Schakelingen voor de aansturing van JK-flipflops zijn goedkoper dan die voor SR-flipflops:  
meer don't cares

## Sequentiële schakelingen

### Bouwblokken

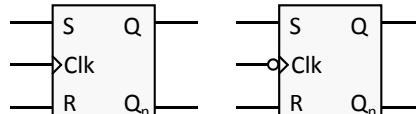
- ⇒ Flipflop
  - latch
  - flipflop

- ⇒ types
- Register
- Teller
- VHDL

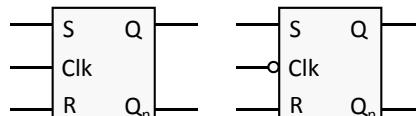
Synchroon  
Asynchroon

# SR-flipflop

### Symbool



Stijgende Dalende  
flanktriggering



Positieve Negatieve  
niveautriggering

### Karakteristieke tabel (voor ontwerp van flipflop)

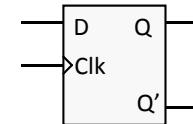
| S | R | Q(next) |
|---|---|---------|
| 0 | 0 | Q       |
| 0 | 1 | 0       |
| 1 | 0 | 1       |
| 1 | 1 | NA      |

### Excitatietabel (voor ontwerp met flipflop)

| Q | Q(next) | S | R |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | 0 |
| 1 | 0       | 0 | 1 |
| 1 | 1       | - | 0 |

# D-flipflop

### Symbool



### Karakteristieke tabel

| D | Q(next) |
|---|---------|
| 0 | 0       |
| 1 | 1       |

onthoud data gedurende 1 klokperiode

### Excitatietabel

| Q | Q(next) | D |
|---|---------|---|
| 0 | 0       | 0 |
| 0 | 1       | 1 |
| 1 | 0       | 0 |
| 1 | 1       | 1 |

Ontwerpen met D-flipflops is eenvoudig:  
 $D = Q(\text{next})$

## Sequentiële schakelingen

### Bouwblokken

- ⇒ Flipflop
- latch
- flipflop

- ⇒ types

- Register

- Teller

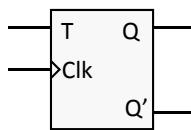
- VHDL

*Synchroon*

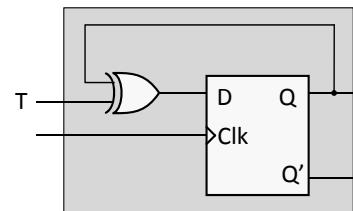
*Asynchroon*

## T-flipflop

### Symbol



'toggle' (verander van) toestand als  $T = 1$



### 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 |

## Sequentiële schakelingen

### Bouwblokken

- Flipflop

- ⇒ Register

- Teller

- VHDL

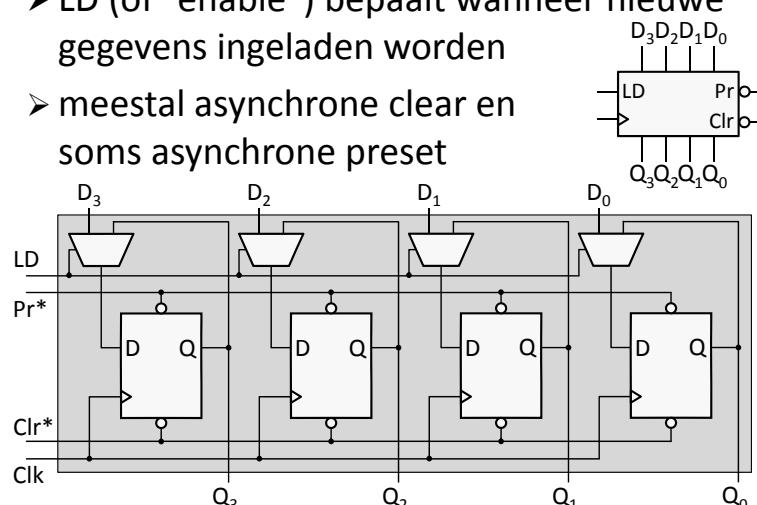
*Synchroon*

*Asynchroon*

## Register

= geheugen voor een datawoord ( $\equiv n$  FF)

- LD (of "enable") bepaalt wanneer nieuwe gegevens ingeladen worden
- meestal asynchrone clear en soms asynchrone preset



## Sequentiële schakelingen

### Bouwblokken

- Flipflop

- ⇒ Register

- Teller

- VHDL

*Synchroon*

*Asynchroon*

## Ontwerp sequentiële schakelingen

### → Bouwblokken

- De flipflop

### → Registers

- Tellers

- In VHDL

□ Ontwerp van synchrone sequentiële schakelingen

□ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

- Flipflop

- ⇒ Register

- Teller

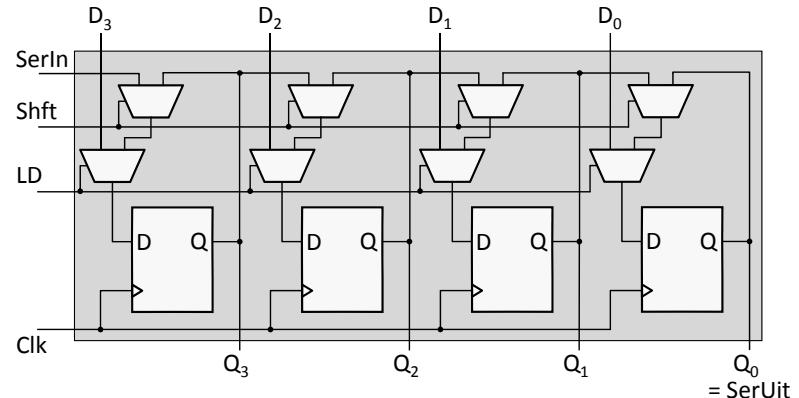
- VHDL

*Synchroon*

*Asynchroon*

## Schuifregister

= schuif inhoud van een register over één plaats



□ Kan uitgebreid worden om naar links te schuiven of in beide richtingen

□ Bruikbaar voor omzetting parallel ↔ serieel

## Sequentiële schakelingen

### Bouwblokken

- Flipflop
- Register
- ⇒ Teller
- VHDL

### Synchroon Asynchroon

# Ontwerp sequentiële schakelingen

## → Bouwblokken

- De flipflop
- Registers
- ➔ Tellers
- In VHDL

- ❑ Ontwerp van synchrone sequentiële schakelingen
- ❑ Ontwerp van asynchrone sequentiële schakelingen

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

## Sequentiële schakelingen

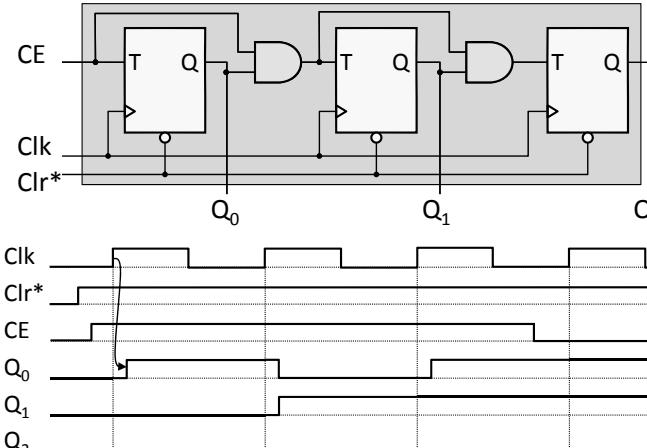
### Bouwblokken

- Flipflop
- Register
- ⇒ Teller
- VHDL

### Synchroon Asynchroon

# Synchrone teller

- = alle FF gebruiken hetzelfde kloksignaal
- ⇒ alle uitgangen veranderen tegelijkertijd



KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

## Sequentiële schakelingen

### Bouwblokken

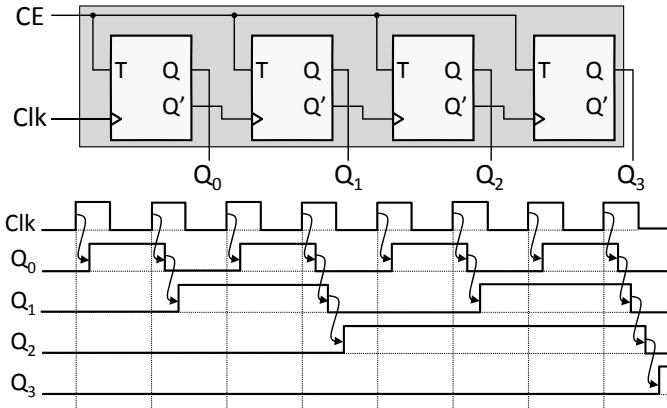
- Flipflop
- Register
- ⇒ Teller
- VHDL

### Synchroon Asynchroon

# Teller

= register + opteller (+1 of -1)

❑ Kan eenvoudiger: asynchrone teller



➢ telt naar boven ('up-counter');  
naar beneden ('down-counter') als Q i.p.v. Q' klokt

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

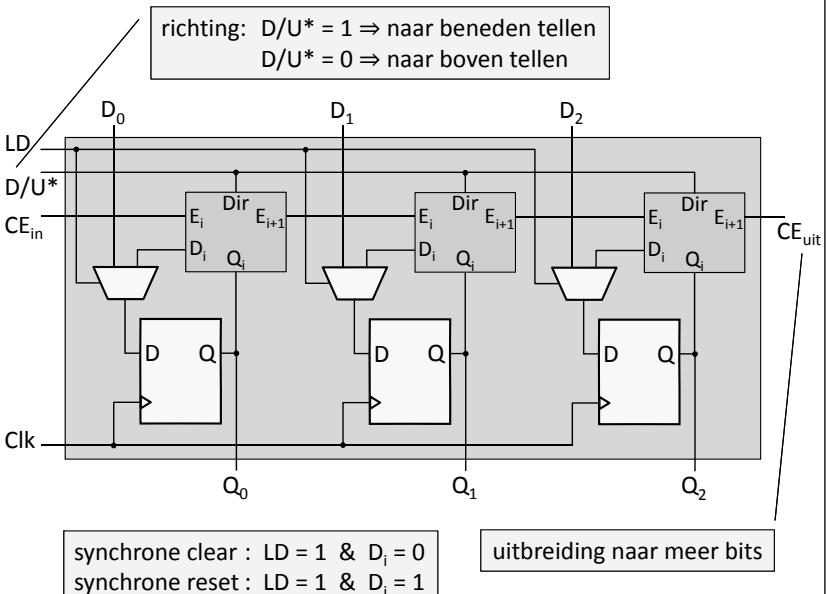
## Sequentiële schakelingen

### Bouwblokken

- Flipflop
- Register
- ⇒ Teller
- VHDL

### Synchroon Asynchroon

# Parallel-laadbare bidirectionele teller



KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

## Sequentiële schakelingen

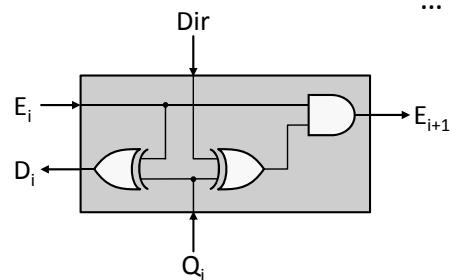
### Bouwbladen

- Flipflop
- Register
- ⇒ Teller
- VHDL

Synchroon  
Asynchroon

## Parallel-laadbare bidirectionele teller

- Bit  $i$  telt (verandert) telkens  $E_i = 1$   
 $\Rightarrow D_i = Q_i \oplus E_i$
  - Bit  $i+1$  telt als  $E_{i+1} = 1$ 
    - > Dir = 0 : als bit  $i$  1 → 0  
 $E_{i+1} = E_i \cdot Q_i$
    - > Dir = 1 : als bit  $i$  0 → 1  
 $E_{i+1} = E_i \cdot Q'_i$
- |         |         |
|---------|---------|
| Dir = 0 | Dir = 1 |
| 00      | 00      |
| 01      | 11      |
| 10      | 10      |
| 11      | 01      |
| 00      | 00      |
| 01      | 11      |
| ...     | ...     |



## Ontwerp sequentiële schakelingen

### Bouwbladen

- Flipflop
- Register
- Teller
- ⇒ VHDL

Synchroon  
Asynchroon

### → Bouwbladen

- De flipflop
- Registers
- Tellers
- In VHDL

- Ontwerp van synchrone sequentiële schakelingen
- Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

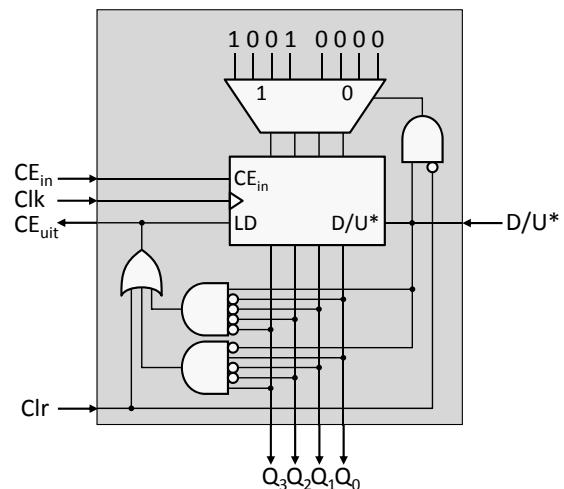
### Bouwbladen

- Flipflop
- Register
- ⇒ Teller
- VHDL

Synchroon  
Asynchroon

## Bidirectionele BCD-teller

BCD-teller  $\equiv$  modulo-10 teller



## Sequentiële schakelingen

### Bouwbladen

- Flipflop
- Register
- Teller
- ⇒ VHDL

Synchroon  
Asynchroon

## Flipflops in VHDL

VHDL heeft geen speciale uitdrukkingen voor flipflops!

- FF zijn impliciet aanwezig als een signaal of variabele zijn waarde behoudt gedurende een tijd.
- Dit gebeurt typisch bij een onvolledige if of case uitdrukking in een proces.

```
➤ process(D,Clk) is ➤ Mux: process(D,Clk) is
begin begin
 if (Clk='1') then if Clk = '1' then
 Q <= D; Q <= D;
 end if; else Q <= '0';
 end process; end if;
 end process Mux;
```

Dit is een latch: als Clk=1, Q volgt D:

- Clk-event & Clk=0 : er gebeurt niets (Q houdt zijn waarde)
- Clk-event & Clk=1 : D wordt gekopieerd naar Q
- D-event & Clk=1 : D wordt gekopieerd naar Q

## Sequentiële schakelingen

### Bouwblokken

- Flipflop
  - Register
  - Teller
- ⇒ VHDL

Synchroon  
Asynchroon

## Hoe een stijgende klokflank beschrijven?

### □ Met een “event”-attribuut:

```
DFF: process(D,Clk) is
begin
 if (Clk'event and Clk='1') then
 Q <= D;
 end if;
end process DFF;
```

- Gebruik “rising\_edge(Clk)” voor std\_logic signalen
  - Dit houdt ook rekening met 'H', ...

### □ Met een “wait until” uitdrukking (zie later)

= stijgende flank van Clk

## Sequentiële schakelingen

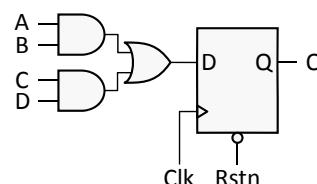
### Bouwblokken

- Flipflop
  - Register
  - Teller
- ⇒ VHDL

Synchroon  
Asynchroon

## Combinatorische schakelingen met registers aan de uitgangen

```
entity CombReg is
 port(A,B,C,D: in std_logic;
 Clk,Rstn: in std_logic;
 Q: out std_logic);
end entity CombReg;
```



```
architecture RTL of CombReg is
begin
 process(A,B,C,D,Clk,Rstn) is
 begin
 if Rstn = '0' then
 Q <= '0';
 elsif rising_edge(Clk) then
 -- combinatorische schakeling
 Q <= (A and B) or (C and D);
 end if;
 end process;
end architecture RTL;
```

1 proces efficiënter dan  
4 parallelle uitdrukkingen

## Sequentiële schakelingen

### Bouwblokken

- Flipflop
  - Register
  - Teller
- ⇒ VHDL

Synchroon  
Asynchroon

## Flipflop met reset

### Synchroone 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;
```

## Sequentiële schakelingen

### Bouwblokken

- Flipflop
  - Register
  - Teller
- ⇒ VHDL

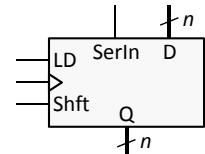
Synchroon  
Asynchroon

## Schuifregister in VHDL

```
library ieee; use ieee.std_logic_1164.all;
entity ShiftReg is
 generic (n: integer := 8);
 port(D: in std_logic_vector(n-1 downto 0);
 Clk,LD,Shft,SerIn: in std_logic;
 Q: out std_logic_vector(n-1 downto 0));
end entity ShiftReg;
```

```
architecture Behav of ShiftReg is
begin
```

```
 process(D,Clk,LD,Shft,SerIn) is
 begin
 if rising_edge(Clk) then
 if LD = '1' then
 Q <= D;
 elsif Shft = '1' then
 for i in 0 to n-2 loop
 Q(i) <= Q(i+1);
 end loop;
 Q(n-1) <= SerIn;
 end if;
 end if;
 end process;
end architecture Behav;
```



## Sequentiële schakelingen

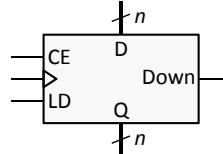
### Bouwblokken

- Flipflop
  - Register
  - Teller
  - ⇒ VHDL
- Synchroon**
- Asynchroon**

# Teller in VHDL

```
library ieee; use ieee.std_logic_unsigned.all;
entity BiDirCnt is
 generic (n: integer := 8);
 port(D: in std_logic_vector(n-1 downto 0);
 Clk,LD,CE,Down: in std_logic;
 Q: out std_logic_vector(n-1 downto 0));
end entity BiDirCnt;

architecture Behav of BiDirCnt is
begin
 process(D,Clk,LD,CE,Down) is
 begin
 if rising_edge(Clk) then
 if LD = '1' then
 Q <= D;
 elsif CE = '1' then
 if Down = '1' then Q <= Q - 1;
 else Q <= Q + 1; end if;
 end if;
 end if;
 end process;
end architecture Behav;
```

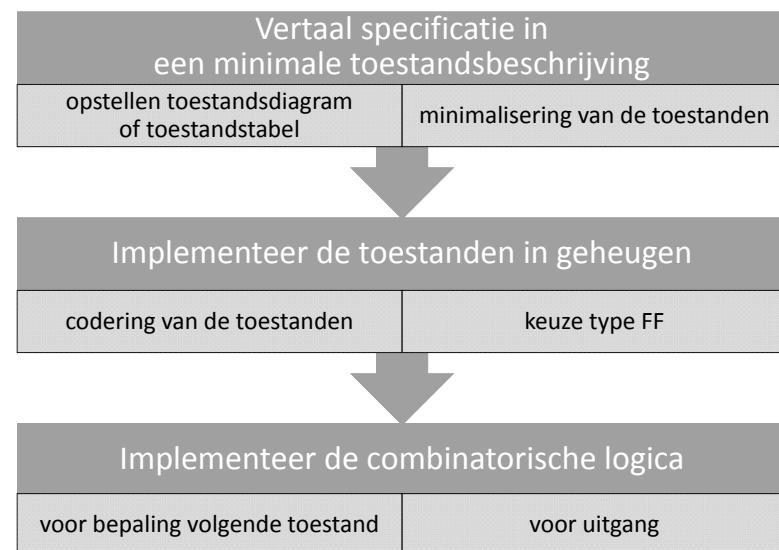


## Sequentiële schakelingen

### Bouwblokken

- Voorbeeld
  - Ontwerp
  - Tijdsgedrag
  - VHDL
- Synchroon**
- Asynchroon**

## Ontwerp van synchrone sequentiële schakelingen



## Sequentiële schakelingen

### Bouwblokken

- Synchroon**
- Voorbeeld
  - Ontwerp
  - Tijdsgedrag
  - VHDL
- Asynchroon**

# Ontwerp sequentiële schakelingen

## □ Bouwblokken

→ **Ontwerp van synchrone sequentiële schakelingen**

□ **Ontwerp van asynchrone sequentiële schakelingen**

## Sequentiële schakelingen

- Bouwblokken**
- Synchroon**
- Voorbeeld
  - Ontwerp
  - Tijdsgedrag
  - VHDL
- Asynchroon**

## Ontwerp sequentiële schakelingen

## □ Bouwblokken

→ **Ontwerp van synchrone sequentiële schakelingen**

➤ Een voorbeeld

- Moore-FSM (toestandgebaseerd)
- Mealy-FSM (inputgebaseerd)

➤ Het ontwerp in detail

➤ Tijdsgedrag

➤ FSM in VHDL

□ **Ontwerp van asynchrone sequentiële schakelingen**

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

- Moore

- Mealy

- Ontwerp

- Tijds gedrag

- VHDL

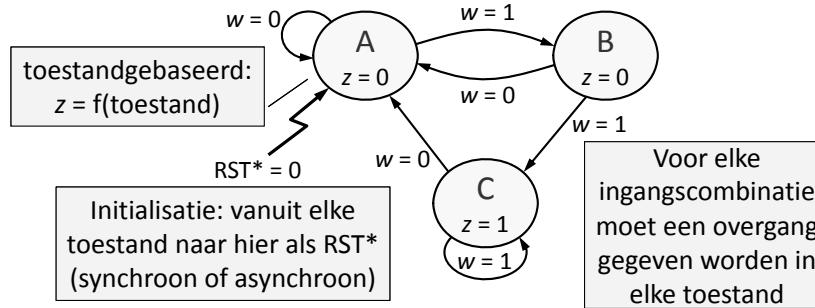
- Asynchroon

# Vertaal specificatie (1)

## Specificatie

- Schakeling met 1 ingang  $w$  en 1 uitgang  $z$ , geklokt op een stijgende flank
- $z(t_i) = 1 \Leftrightarrow w(t_{i-1}) = 1$  en  $w(t_{i-2}) = 1$

## Toestandsdiagram



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

- Moore

- Mealy

- Ontwerp

- Tijds gedrag

- VHDL

- Asynchroon

# Vertaal specificatie (2)

## Toestandstabel

= alternatieve voorstelling van het diagram

| Huidige toestand | Volgende toestand<br>$w = 0$ | Volgende toestand<br>$w = 1$ | Uitgang<br>$z$ |
|------------------|------------------------------|------------------------------|----------------|
| A                | A                            | B                            | 0              |
| B                | A                            | C                            | 0              |
| C                | A                            | C                            | 1              |

Gebruik een minimaal aantal toestanden

- Dit is reeds het minimaal aantal.

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

- Moore

- Mealy

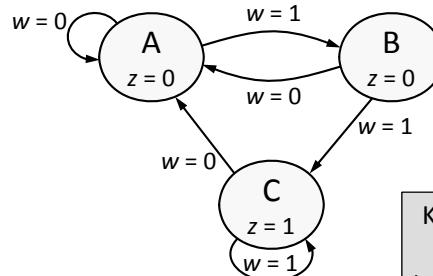
- Ontwerp

- Tijds gedrag

- VHDL

- Asynchroon

# Interpretatie toestandsdiagram



Kloksignaal **noot** in synchroon toestandsdiagram!

- We bevinden ons in toestand "A" en  $w$  is 0: we wachten aan de tip van de pijl
- $w$  wordt 1: spring naar tip andere pijl (zelfde toestand!)
- Stijgende klokflank (synchrone FSM!): ga naar toestand "B" (met  $w = 1$ )
- $w$  wordt 0: spring naar tip andere pijl
- Stijgende klokflank: ga naar "A" (met  $w = 0$ )

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

- Moore

- Mealy

- Ontwerp

- Tijds gedrag

- VHDL

- Asynchroon

# Implementeer toestanden

## Codeer de toestanden

= koppel toestand aan een combinatie van flipflop-uitgangen

| Huidige toestand | Volgende toestand |                 | Uitgang |
|------------------|-------------------|-----------------|---------|
|                  | $w = 0$           | $w = 1$         |         |
| $Q_1 Q_0$        | $Q_{1n} Q_{0n}$   | $Q_{1n} Q_{0n}$ | $z$     |
| A                | 00                | 00              | 01      |
| B                | 01                | 00              | 10      |
| C                | 10                | 00              | 10      |
|                  | 11                | --              | --      |

3 toestanden  
↓  
2 flipflops  
↓  
1 ongebruikte toestand

A ⇒ "00"  
(reset-toestand)

## Kies het type flipflop

- We kiezen de D-FF (eenvoud van ontwerp)

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

⇒ Moore

▪ Mealy

▪ Ontwerp

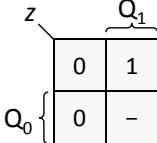
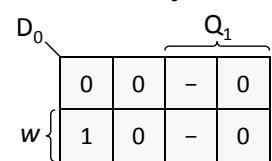
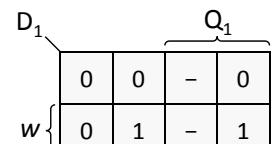
▪ Tijds gedrag

▪ VHDL

Asynchroon

# Implementeer combinatorische logica

| Huidige toestand | Volgende toestand<br>$w = 0$ | Volgende toestand<br>$w = 1$ | Uitgang |
|------------------|------------------------------|------------------------------|---------|
| $Q_1 Q_0$        | $Q_{1n} Q_{0n}$              | $Q_{1n} Q_{0n}$              | $z$     |
| 00               | 00                           | 01                           | 0       |
| 01               | 00                           | 10                           | 0       |
| 10               | 00                           | 10                           | 1       |
| 11               | --                           | --                           | -       |



- Bepaling volgende toestand: gebruik excitatietafel van FF

| $Q_i$ | $Q_{in}$ | $D_i$ |
|-------|----------|-------|
| 0     | 0        | 0     |
| 0     | 1        | 1     |
| 1     | 0        | 0     |
| 1     | 1        | 1     |

$$D_i = Q_{in}$$

- Bepaling uitgang

## Ontwerp sequentiële schakelingen

### Bouwblokken

- Ontwerp van synchrone sequentiële schakelingen

#### → Een voorbeeld

- Moore-FSM

→ Mealy-FSM (input gebaseerd)

➤ Het ontwerp in detail

➤ Tijds gedrag

➤ FSM in VHDL

- Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

⇒ Voorbeeld

⇒ Moore

▪ Mealy

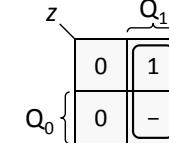
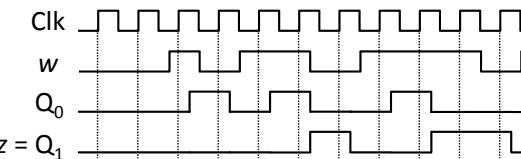
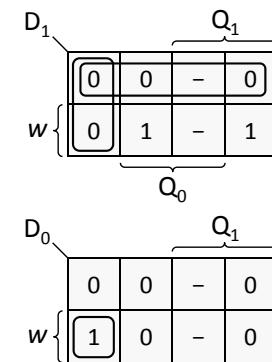
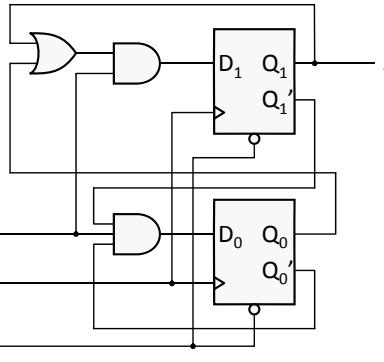
▪ Ontwerp

▪ Tijds gedrag

▪ VHDL

Asynchroon

# Implementeer combinatorische logica

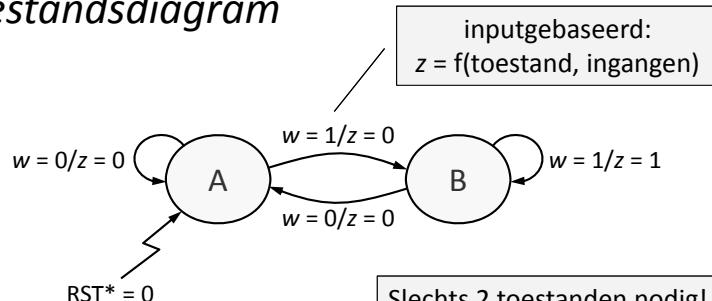


## Vertaalg specificatie (1)

### Specificatie

- Schakeling met 1 ingang  $w$  en 1 uitgang  $z$ , geklokt op een stijgende flank
- $z(t_i) = 1 \Leftrightarrow w(t_i) = 1$  en  $w(t_{i-1}) = 0$

### Toestandsdiagram



## Sequentiële schakelingen

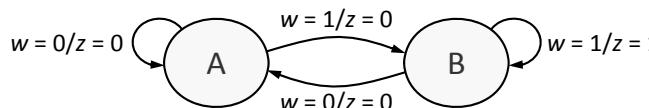
### Bouwblokken

#### Synchroon

- ⇒ Voorbeeld
  - Moore
  - Mealy
  - Ontwerp
  - Tijds gedrag
  - VHDL
- Asynchroon

# Vertaal specificatie (2)

## Toestandstabel



| Huidige toestand | Volgende toestand/Uitgang |       |
|------------------|---------------------------|-------|
|                  | w = 0                     | w = 1 |
| A                | A/0                       | B/0   |
| B                | A/0                       | B/1   |

Gebruik een minimaal aantal toestanden

- Dit is reeds het minimaal aantal.

## Sequentiële schakelingen

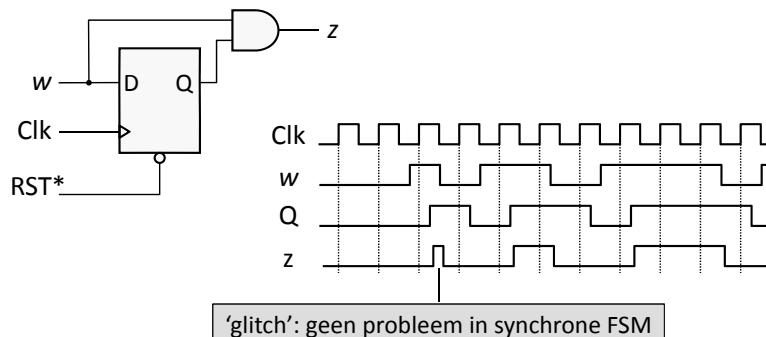
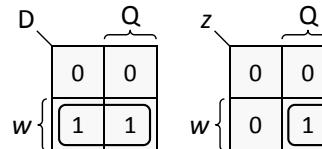
### Bouwblokken

#### Synchroon

- ⇒ Voorbeeld
  - Moore
  - Mealy
  - Ontwerp
  - Tijds gedrag
  - VHDL
- Asynchroon

## Implementeer combinatorische logica

| Huidige toestand | Volgende toestand/Uitgang |       |
|------------------|---------------------------|-------|
|                  | w = 0                     | w = 1 |
| Q                | D/z                       | D/z   |
| 0                | 0/0                       | 1/0   |
| 1                | 0/0                       | 1/1   |



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- ⇒ Voorbeeld
  - Moore
  - Mealy
  - Ontwerp
  - Tijds gedrag
  - VHDL
- Asynchroon

# Implementeer toestanden

## Codeer de toestanden

| Huidige toestand | Volgende toestand/Uitgang |                   |
|------------------|---------------------------|-------------------|
|                  | w = 0                     | w = 1             |
| Q                | Q <sub>n</sub> /z         | Q <sub>n</sub> /z |
| A                | 0/0                       | 1/0               |
| B                | 0/0                       | 1/1               |

2 toestanden  
↓  
1 flipflop

A ⇒ "0"  
(reset-toestand)

## Kies het type flipflop

- We kiezen de D-FF (eenvoud van ontwerp)  
⇒ D = Q<sub>n</sub>

## Sequentiële schakelingen

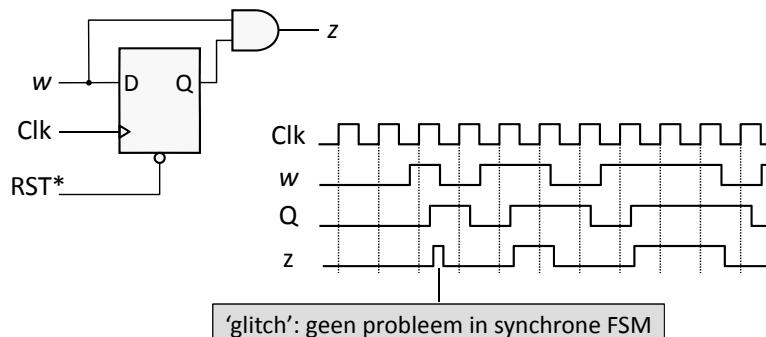
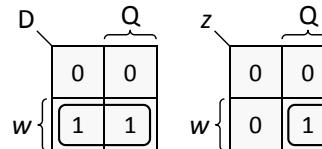
### Bouwblokken

#### Synchroon

- ⇒ Voorbeeld
  - Moore
  - Mealy
  - Ontwerp
  - Tijds gedrag
  - VHDL
- Asynchroon

## Implementeer combinatorische logica

| Huidige toestand | Volgende toestand/Uitgang |       |
|------------------|---------------------------|-------|
|                  | w = 0                     | w = 1 |
| Q                | D/z                       | D/z   |
| 0                | 0/0                       | 1/0   |
| 1                | 0/0                       | 1/1   |



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- ⇒ Voorbeeld
  - Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL
- Asynchroon

## Ontwerp sequentiële schakelingen

### Bouwblokken

- Ontwerp van synchrone sequentiële schakelingen
  - Een voorbeeld
  - Het ontwerp in detail

1. Toestandstabel: constructie FSM
2. Minimalisering aantal toestanden
3. Codering toestanden
4. Keuze van het type flipflop
5. Realisatie van de combinatorische logica

- Tijds gedrag
- FSM in VHDL

### Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
- ⇒ tabel
- minimaliseer
- codering
- keuze FF
- realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Constructie FSM

= construeer de FSM vertrekend van de beschrijving in een natuurlijke taal (onduidelijk & onvolledig)

Bijv. modulo-3 op/neer-teller

- C(ount enable) : tel als  $C = 1$
- D(irection) : tel op/neer als  $D = 0/1$
- uitgangen M(SB) en L(SB) : tellerwaarden
- uitgang Y : 'enable' voor uitbreiding  
 $Y = 1$  als
  - bij optellen ( $C = 1$  and  $D = 0$ )  
tellerwaarde = 2
  - bij neertellen ( $C = 1$  and  $D = 1$ )  
tellerwaarde = 0

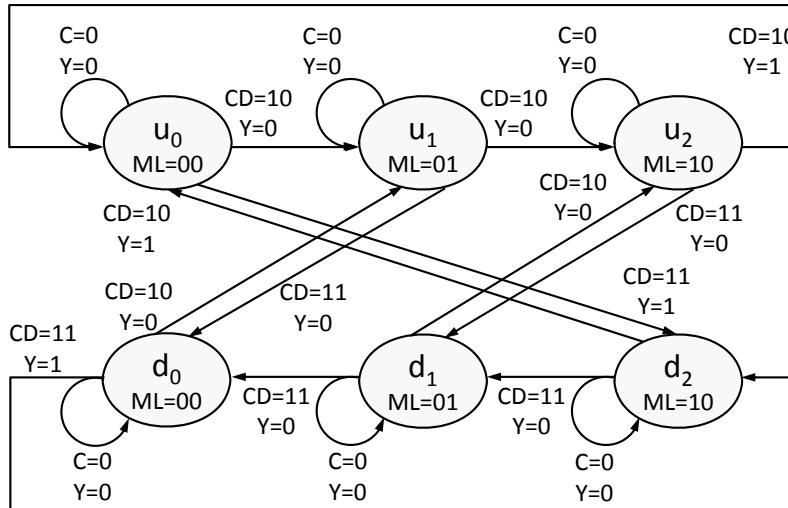
## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
- ⇒ tabel
- minimaliseer
- codering
- keuze FF
- realisatie
- Tijds gedrag
- VHDL

### Asynchroon

## Toestandsdiagram modulo-3 teller



## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
- ⇒ tabel
- minimaliseer
- codering
- keuze FF
- realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Constructie FSM

1) Is dit ontwerp toestand- of inputgebaseerd?

- Het voorbeeld is inputgebaseerd omdat de uitgang Y afhangt van de toestand (tellerwaarde) en de ingangen "C" & "D"

2) Maak het toestandsdiagram en/of de toestandstabel:

- vertrek van de initiële toestand
- voor elke nieuwe toestand, teken voor elke combinatie van ingangen de overgang naar de volgende toestand
- uitgangswaarden moeten toegekend worden
  - aan elke toestand als toestandgebaseerd
  - aan elke overgang als inputgebaseerd

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
- ⇒ tabel
- minimaliseer
- codering
- keuze FF
- realisatie
- Tijds gedrag
- VHDL

### Asynchroon

## Toestandstabel modulo-3 teller

| Huidige toestand | Volgende toestand / MLY |           |           |
|------------------|-------------------------|-----------|-----------|
|                  | C = 0                   | CD = 10   | CD = 11   |
| $u_0$            | $u_0/000$               | $u_1/000$ | $d_2/001$ |
| $u_1$            | $u_1/010$               | $u_2/010$ | $d_0/010$ |
| $u_2$            | $u_2/100$               | $u_0/101$ | $d_1/100$ |
| $d_0$            | $d_0/000$               | $u_1/000$ | $d_2/001$ |
| $d_1$            | $d_1/010$               | $u_2/010$ | $d_0/010$ |
| $d_2$            | $d_2/100$               | $u_0/101$ | $d_1/100$ |

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - ⇒ minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Ontwerp sequentiële schakelingen

## □ Bouwblokken

### → Ontwerp van synchrone sequentiële schakelingen

➤ Een voorbeeld

#### → Het ontwerp in detail

1. Toestandstabel
2. Minimalisering aantal toestanden: minimaal # FF
3. Codering toestanden
4. Keuze van het type flipflop
5. Realisatie van de combinatorische logica

➤ Tijds gedrag

➤ FSM in VHDL

## □ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - ⇒ minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Minimalisering door partitionering

Partitie = set van groepen toestanden waarbij

- elke *groep* bevat toestanden die *equivalent kunnen zijn*,
- toestanden uit *verschillende groepen* zijn *zeker niet equivalent*

Minimalisering door partitionering =

splits iteratief de groepen op tot elke groep enkel nog equivalente toestanden bevat

- 1) groepeer alle toestanden die voor alle ingangen dezelfde uitgangen hebben
- 2) iteratie: splits een groep op als toestanden voor bepaalde ingangen een volgende toestand hebben in een verschillende groep

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - ⇒ minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Equivalenten FSM

- = twee FSM zijn equivalent als ze dezelfde uitgangs sequentie produceren voor eenzelfde ingangs sequentie
- Twee toestanden kunnen door één toestand vervangen worden (m.a.w. zijn equivalent) als
  - beide toestanden dezelfde uitgangen produceren voor dezelfde ingangen
  - beide toestanden naar equivalente volgende toestanden gaan voor dezelfde ingangen
- Dus toestanden  $s_j$  and  $s_k$  zijn equivalent ( $s_j \equiv s_k$ ) als en slechts als
  - 1)  $\forall i: h(s_j, i) = h(s_k, i)$  : beide toestanden produceren dezelfde uitgang voor elke combinatie van ingangen
  - 2)  $\forall i: f(s_j, i) \equiv f(s_k, i)$  : de volgende toestanden zijn equivalent voor elke combinatie van ingangen

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - ⇒ minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Minimalisering door partitionering: voorbeeld 8.5



| Huidige toestand | Volgende toestand / Uitgang<br>w = 0 | w = 1 |
|------------------|--------------------------------------|-------|
| A                | B/1                                  | C/1   |
| B                | D/1                                  | F/1   |
| C                | F/0                                  | E/0   |
| D                | B/1                                  | G/1   |
| E                | F/0                                  | C/0   |
| F                | E/0                                  | D/0   |
| G                | F/0                                  | G/0   |

|   |     |     |
|---|-----|-----|
| A | B/1 | C/1 |
| B | A/1 | F/1 |
| C | F/0 | C/0 |
| F | C/0 | A/0 |

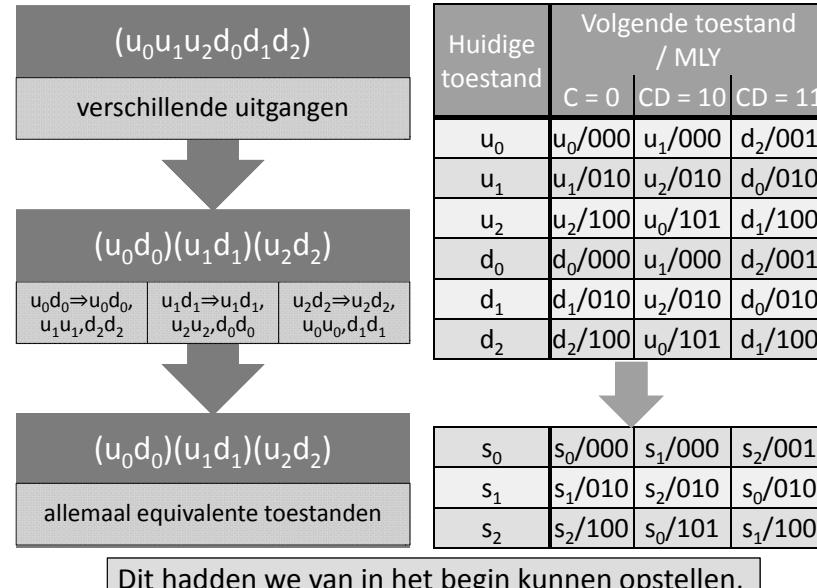
## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - ⇒ minimaliseer
  - codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Minimalisering door partitionering: modulo-3 teller



## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Ontwerp sequentiële schakelingen

## □ Bouwblokken

## → Ontwerp van synchrone sequentiële schakelingen

➢ Een voorbeeld

## → Het ontwerp in detail

1. Toestandsdiagram
2. Minimalisering aantal toestanden
3. Codering toestanden
4. Keuze van het type flipflop
5. Realisatie van de combinatorische logica

➢ FSM in VHDL

➢ Tijds gedrag

## □ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Toestandscodering

- $n$  toestanden vragen minstens  $\lceil \log_2 n \rceil$  flipflops
- Er zijn  $n!$  mogelijke coderingen ( $n$  keuzes voor de eerste toestand,  $n-1$  voor de tweede, ...)

| Nr. | s0 | s1 | s2 | s3 |
|-----|----|----|----|----|
| 1   | 00 | 01 | 10 | 11 |
| 2   | 00 | 01 | 11 | 10 |
| 3   | 00 | 10 | 01 | 11 |
| 4   | 00 | 10 | 11 | 01 |
| 5   | 00 | 11 | 01 | 10 |
| 6   | 00 | 11 | 10 | 01 |
| 7   | 01 | 00 | 10 | 11 |
| 8   | 01 | 00 | 11 | 10 |
| 9   | 01 | 10 | 00 | 11 |
| 10  | 01 | 10 | 11 | 00 |
| 11  | 01 | 11 | 00 | 10 |
| 12  | 01 | 11 | 10 | 00 |

| Nr. | s0 | s1 | s2 | s3 |
|-----|----|----|----|----|
| 13  | 10 | 00 | 01 | 11 |
| 14  | 10 | 00 | 11 | 01 |
| 15  | 10 | 01 | 00 | 11 |
| 16  | 10 | 01 | 11 | 00 |
| 17  | 10 | 11 | 00 | 01 |
| 18  | 10 | 11 | 01 | 00 |
| 19  | 11 | 00 | 01 | 10 |
| 20  | 11 | 00 | 10 | 01 |
| 21  | 11 | 01 | 00 | 10 |
| 22  | 11 | 01 | 10 | 00 |
| 23  | 11 | 10 | 00 | 01 |
| 24  | 11 | 10 | 01 | 00 |

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Toestandscodering

## □ Wat is de impact van de keuze?

- Elke keuze leidt tot een verschillende combinatorische schakeling, telkens met een andere kostprijs en vertraging (zie bijv. boek, hoofdstuk 8.2)

## □ Meest gekozen toestandscoderingen:

- Straightforward (voor de hand liggend)
- Minimum-bit-change
- One-hot (één-actief)

## □ Verdere verfijning mogelijk

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Straightforward-codering

- = gebruik de binaire voorstelling van het nummer van de toestand als code ( $s_0 \rightarrow 000, s_5 \rightarrow 101, \dots$ )
  - ❑ Meestal gebruikt als toestandsnummer een fysische betekenis heeft (bijv. een teller)
  - ❑ Meerdere FF veranderen bij elke overgang
    - meerdere bits veranderen zelden tegelijkertijd
    - elke bitverandering verbruikt vermogen
    - elke bitverandering heeft wat logica nodig
- Dit leidt tot
- gevaar voor 'glitches'
  - hoger vermogenverbruik
  - hogere kostprijs

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Straightforward-codering: modulo-3 teller

| Huidige toestand | Volgende toestand / MLY |           |           |
|------------------|-------------------------|-----------|-----------|
|                  | C = 0                   | CD = 10   | CD = 11   |
| $s_0$            | $s_0/000$               | $s_1/000$ | $s_2/001$ |
| $s_1$            | $s_1/010$               | $s_2/010$ | $s_0/010$ |
| $s_2$            | $s_2/100$               | $s_0/101$ | $s_1/100$ |

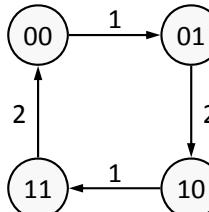


Voordeel:  
 $ML = Q_1 Q_0$

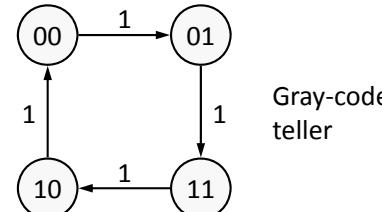
| Huidige toestand | Volgende toestand / Y |         |         |
|------------------|-----------------------|---------|---------|
|                  | C = 0                 | CD = 10 | CD = 11 |
| 00               | 00/0                  | 01/0    | 10/1    |
| 01               | 01/0                  | 10/0    | 00/0    |
| 10               | 10/0                  | 00/1    | 01/0    |

# Minimum-bit-change-codering

- = maak het totaal aantal bitveranderingen voor alle toestandsveranderingen samen minimaal



Straightforward



Minimum-bit-change

- ❑ Meestal gebruikt als kostprijs en vermogen minimaal moeten zijn (CMOS)

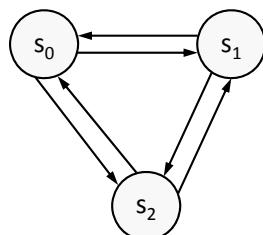
## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Minimum-bit-change-codering: modulo-3 teller



Mogelijke codering:  
 $s_0 = 00$   
 $s_1 = 10$   
 $s_2 = 11$

| Huidige toestand | Volgende toestand / MLY |           |           |
|------------------|-------------------------|-----------|-----------|
|                  | C = 0                   | CD = 10   | CD = 11   |
| $s_0$            | $s_0/000$               | $s_1/000$ | $s_2/001$ |
| $s_1$            | $s_1/010$               | $s_2/010$ | $s_0/010$ |
| $s_2$            | $s_2/100$               | $s_0/101$ | $s_1/100$ |

Alle overgangen zijn even waarschijnlijk.  
1 bit verschil bij een overgang is telkens slechts mogelijk voor 4 van de 6 overgangen.

| Huidige toestand | Volgende toestand / MLY |         |         |
|------------------|-------------------------|---------|---------|
|                  | C = 0                   | CD = 10 | CD = 11 |
| 00               | 00/000                  | 10/000  | 11/001  |
| 10               | 10/010                  | 11/010  | 00/010  |
| 11               | 11/100                  | 00/101  | 10/100  |

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# One-hot-codering

- = voorzie één flipflop per toestand (ongecodeerd)  
Q van 1 FF = 1, Q van de andere FF = 0
- Kostprijs FF =  $O(n)$  i.p.v.  $O(\log n)$   
⇒ enkel bruikbaar voor klein aantal toestanden
- Zeer eenvoudige ontwerp ⇒ snel ontwerp
- Eenvoudige combinatorische schakelingen aan de ingangen van de flipflops
  - > Goedkoop combinatorisch deel, duur FF-deel
  - > FPGA heeft in een logische cel een kleine combinatorische schakeling en 1 FF  
⇒ one-hot-codering is ideaal voor FPGA, tenzij teveel toestanden (bijv. tellers)

## Ontwerp sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - ⇒ keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - ⇒ codering
  - keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# One-hot-codering: modulo-3 teller

| Huidige toestand | Volgende toestand / MLY |           |           |
|------------------|-------------------------|-----------|-----------|
| C = 0            | CD = 10                 | CD = 11   |           |
| $s_0$            | $s_0/000$               | $s_1/000$ | $s_2/001$ |
| $s_1$            | $s_1/010$               | $s_2/010$ | $s_0/010$ |
| $s_2$            | $s_2/100$               | $s_0/101$ | $s_1/100$ |

$s_0 = 001$   
 $s_1 = 010$   
 $s_2 = 100$

| Huidige toestand | Volgende toestand / MLY |         |         |
|------------------|-------------------------|---------|---------|
| C = 0            | CD = 10                 | CD = 11 |         |
| 001              | 001/000                 | 010/000 | 100/001 |
| 010              | 010/010                 | 100/010 | 001/010 |
| 100              | 100/100                 | 001/101 | 010/100 |

# Vergelijking types flipflop

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - ⇒ keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

|                     | 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 & frequentiedelers |

- Bouwblokken
- ➔ Ontwerp van synchrone sequentiële schakelingen
  - > Een voorbeeld
  - ➔ Het ontwerp in detail
    1. Toestandstabel
    2. Minimalisering aantal toestanden
    3. Codering toestanden
    4. Keuze van het type flipflop
    5. Realisatie van de combinatorische logica
  - > Tijds gedrag
  - > FSM in VHDL
- Ontwerp van asynchrone sequentiële schakelingen

- # don't care bepalen kost en snelheid van de combinatorische logica (minder ⇒ duurder/trager)

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - ⇒ keuze FF
  - realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Keuze type flipflop

Geen algemene regel!

- Voor de goedkoopste schakeling moeten alle varianten uitgeprobeerd worden.
- Als een korte ontwerptijd belangrijk is, zijn D-flipflops de beste keuze.
- FPGA bevat enkel D-flipflops.

## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
  - ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchrone

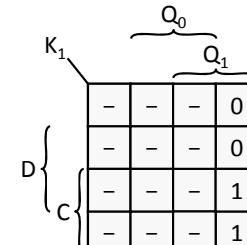
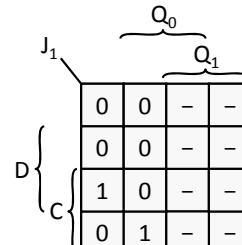
# Realisatie met JK-FF

Bepaal de excitatiefuncties voor FF 1

| Huidige toestand | Volgende toestand / Y |         |         |
|------------------|-----------------------|---------|---------|
|                  | C = 0                 | 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    |

Excitatietabel JK-FF

| Q | Q(next) | J | K |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | - |
| 1 | 0       | - | 1 |
| 1 | 1       | - | 0 |



## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
  - ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Ontwerp sequentiële schakelingen

## □ Bouwblokken

## → Ontwerp van synchrone sequentiële schakelingen

➤ Een voorbeeld

## → Het ontwerp in detail

1. Toestandstabel
2. Minimalisering aantal toestanden
3. Codering toestanden
4. Keuze van het type flip flop
5. Realisatie van de combinatorische logica

➤ Tijds gedrag

➤ FSM in VHDL

## □ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken Synchroon

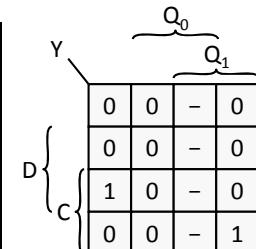
- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
  - ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchrone

# Realisatie met JK-FF

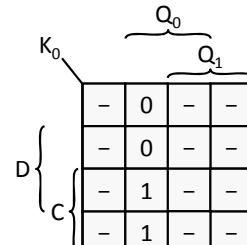
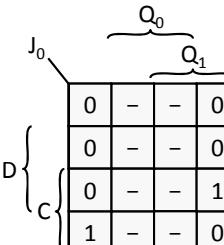
Bepaal de excitatiefuncties voor FF 0

| Huidige toestand | Volgende toestand / Y |         |         |
|------------------|-----------------------|---------|---------|
|                  | C = 0                 | 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    |



Excitatietabel JK-FF

| Q | Q(next) | J | K |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | - |
| 1 | 0       | - | 1 |
| 1 | 1       | - | 0 |



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

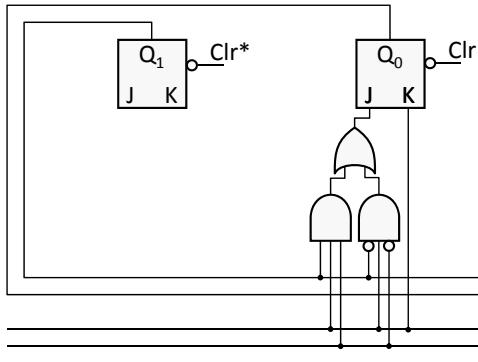
#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met JK-FF

|   |   | $Q_0$ |       | $Q_1$ |       |
|---|---|-------|-------|-------|-------|
|   |   | $J_0$ | $K_0$ | $J_0$ | $K_0$ |
| D | - | 0     | -     | -     | 0     |
| D | - | 0     | -     | 0     | -     |
| C | - | 0     | -     | -     | -     |
| C | - | 1     | -     | -     | -     |
| 1 | - | -     | -     | 0     | -     |
| 0 | - | -     | -     | 0     | -     |



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

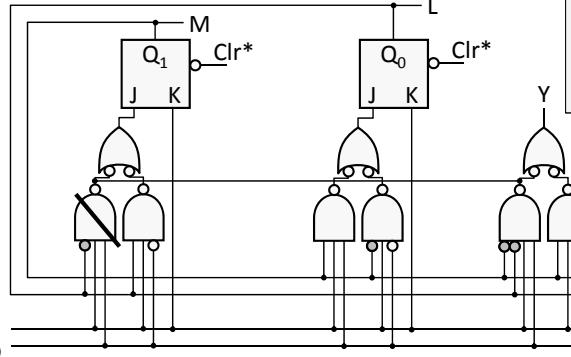
#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met JK-FF

|   |   | $Q_0$ |       | $Q_1$ |       |
|---|---|-------|-------|-------|-------|
|   |   | $J_1$ | $K_1$ | $J_1$ | $K_1$ |
| D | - | 0     | 0     | -     | -     |
| D | - | 0     | 0     | -     | -     |
| C | - | 1     | 0     | -     | -     |
| C | - | -     | -     | 1     | -     |
| 0 | 1 | -     | -     | -     | -     |
| - | - | -     | -     | -     | 1     |



Kostprijs: 23  
(bij gebruik van  $Q'$  en 2-lagen NAND)

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met SR-FF

### Bepaal de excitatiefuncties voor FF 1

|    |      | Volgende toestand / Y |           |           |
|----|------|-----------------------|-----------|-----------|
|    |      | $C = 0$               | $CD = 10$ | $CD = 11$ |
| 00 | 00/0 | 01/0                  | 10/1      |           |
| 01 | 01/0 | 10/0                  | 00/0      |           |
| 10 | 10/0 | 00/1                  | 01/0      |           |

### Excitatietabel SR-FF

| Q | Q(next) | S | R |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | 0 |
| 1 | 0       | 0 | 1 |
| 1 | 1       | - | 0 |

|   |   | $Q_0$ |       | $Q_1$ |       |
|---|---|-------|-------|-------|-------|
|   |   | $S_1$ | $R_1$ | $Q_0$ | $Q_1$ |
| D | - | 0     | 0     | -     | -     |
| D | - | 0     | 0     | -     | -     |
| C | - | 1     | 0     | -     | 0     |
| C | - | 0     | -     | -     | 1     |
| 1 | - | 0     | -     | 1     | -     |
| 0 | - | 1     | -     | 0     | -     |

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met SR-FF

### Bepaal de excitatiefuncties voor FF 0

|    |      | Volgende toestand / Y |           |           |
|----|------|-----------------------|-----------|-----------|
|    |      | $C = 0$               | $CD = 10$ | $CD = 11$ |
| 00 | 00/0 | 01/0                  | 10/1      |           |
| 01 | 01/0 | 10/0                  | 00/0      |           |
| 10 | 10/0 | 00/1                  | 01/0      |           |

|   |   | $Q_0$ |     | $Q_1$ |       |
|---|---|-------|-----|-------|-------|
|   |   | $Y$   | $D$ | $C$   | $Q_0$ |
| 0 | 0 | 0     | 0   | -     | 0     |
| 0 | 0 | 0     | 0   | -     | 0     |
| 1 | 0 | 1     | 0   | -     | 0     |
| 0 | 0 | -     | 0   | -     | 1     |

### Excitatietabel SR-FF

| Q | Q(next) | S | R |
|---|---------|---|---|
| 0 | 0       | 0 | - |
| 0 | 1       | 1 | 0 |
| 1 | 0       | 0 | 1 |
| 1 | 1       | - | 0 |

|   |   | $Q_0$ |       | $Q_1$ |       |
|---|---|-------|-------|-------|-------|
|   |   | $S_0$ | $R_0$ | $Q_0$ | $Q_1$ |
| D | - | 0     | 0     | -     | -     |
| D | - | 0     | 0     | -     | -     |
| C | - | 0     | -     | -     | 0     |
| C | - | 0     | -     | -     | 1     |
| 1 | - | -     | 1     | -     | 0     |
| 0 | - | 0     | 1     | -     | -     |

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

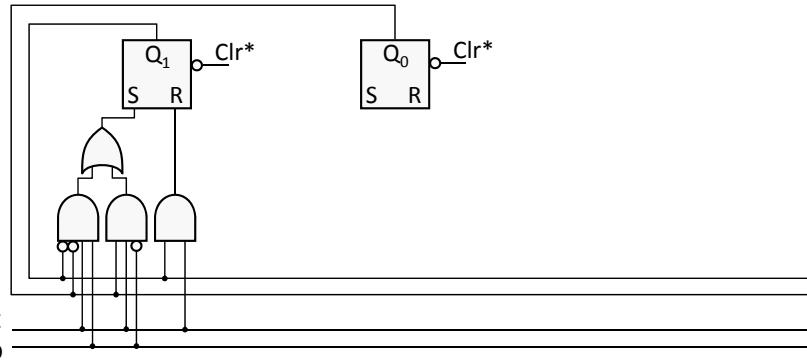
#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met SR-FF

|   | $S_1$   | $R_1$   | $Q_0$ | $Q_1$ |
|---|---------|---------|-------|-------|
| D | 0 0 - - | - - - 0 | $Q_0$ | $Q_1$ |
| C | 1 0 - 0 | - - - 0 | $Q_0$ | $Q_1$ |
|   | 0 1 - 0 | - 0 - 1 | $Q_0$ | $Q_1$ |
|   | 0 0 - - | - 0 - 1 | $Q_0$ | $Q_1$ |



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met D-FF

### Bepaal de excitatiefuncties

| Huidige toestand | Volgende toestand / Y |           |           |
|------------------|-----------------------|-----------|-----------|
|                  | $C = 0$               | $CD = 10$ | $CD = 11$ |
| 00               | 00/0                  | 01/0      | 10/1      |
| 01               | 01/0                  | 10/0      | 00/0      |
| 10               | 10/0                  | 00/1      | 01/0      |

|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 0 - 0 | 0 0 - 0 |
| C | 0 0 - 0 | 0 0 - 0 |
|   | 1 0 - 0 | 0 0 - 0 |
|   | 0 0 - 1 | 0 0 - 1 |

### Excitatietabel D-FF

| Q | $Q(\text{next})$ | D |
|---|------------------|---|
| 0 | 0                | 0 |
| 0 | 1                | 1 |
| 1 | 0                | 0 |
| 1 | 1                | 1 |

|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 0 - 1 | 0 0 - 1 |
| C | 0 0 - 1 | 0 0 - 1 |
|   | 1 0 - 0 | 0 0 - 1 |
|   | 0 1 - 0 | 0 0 - 0 |

|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 1 - 0 | 0 1 - 0 |
| C | 0 1 - 0 | 0 1 - 0 |
|   | 0 0 - 1 | 0 0 - 1 |
|   | 1 0 - 0 | 0 0 - 0 |

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

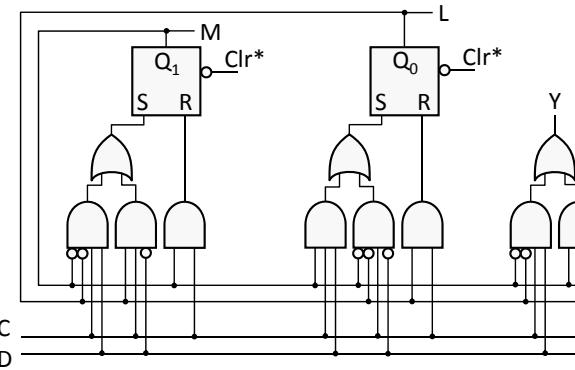
#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met SR-FF

|   | $S_0$   | $R_0$   | $Q_0$ | $Q_1$ |
|---|---------|---------|-------|-------|
| D | 0 - - 0 | - 0 - - | $Q_0$ | $Q_1$ |
| C | 0 - - 0 | - 0 - - | $Q_0$ | $Q_1$ |
|   | 0 0 - 1 | - 1 - 0 | $Q_0$ | $Q_1$ |
|   | 0 1 - 0 | - 0 - 1 | $Q_0$ | $Q_1$ |



Kostprijs: 28  
(dubbel gebruik  
NAND4)

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

## Realisatie met D-FF

### Bepaal de excitatiefuncties

| Huidige toestand | Volgende toestand / Y |           |           |
|------------------|-----------------------|-----------|-----------|
|                  | $C = 0$               | $CD = 10$ | $CD = 11$ |
| 00               | 00/0                  | 01/0      | 10/1      |
| 01               | 01/0                  | 10/0      | 00/0      |
| 10               | 10/0                  | 00/1      | 01/0      |

|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 0 - 1 | 0 0 - 1 |
| C | 0 0 - 1 | 0 0 - 1 |
|   | 1 0 - 0 | 0 0 - 1 |
|   | 0 1 - 0 | 0 0 - 0 |

| Q | $Q(\text{next})$ | D |
|---|------------------|---|
| 0 | 0                | 0 |
| 0 | 1                | 1 |
| 1 | 0                | 0 |
| 1 | 1                | 1 |

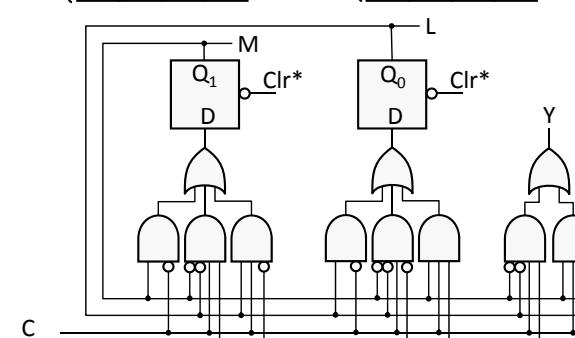
|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 1 - 0 | 0 1 - 0 |
| C | 0 1 - 0 | 0 1 - 0 |
|   | 0 0 - 1 | 0 0 - 1 |
|   | 1 0 - 0 | 0 0 - 0 |

|   | $Q_0$   | $Q_1$   |
|---|---------|---------|
| D | 0 0 - 1 | 0 0 - 1 |
| C | 0 0 - 1 | 0 0 - 1 |
|   | 1 0 - 0 | 0 0 - 1 |
|   | 0 1 - 0 | 0 0 - 0 |

|   | $D_1$   | $D_0$   | $Y$   |
|---|---------|---------|-------|
| D | 0 0 - 1 | 0 1 - 0 | $Q_0$ |
| C | 0 0 - 1 | 0 1 - 0 | $Q_1$ |
|   | 0 1 - 0 | 0 0 - 1 | $Q_0$ |
|   | 1 0 - 0 | 0 0 - 1 | $Q_1$ |

Kostprijs: 31  
(dubbel gebruik  
NAND4)

FPGA-kostprijs:  
3 logische cellen



## Sequentiële schakelingen

### Bouwblokken Synchroon

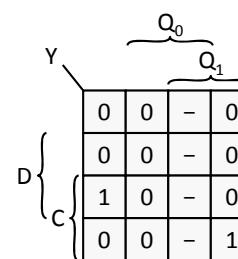
- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Realisatie met T-FF

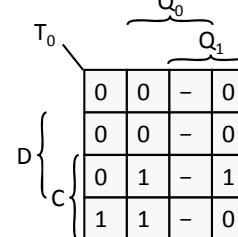
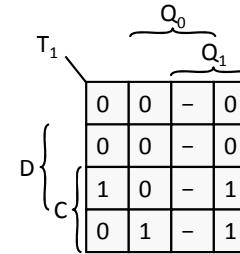
## Bepaal de excitatiefuncties

| Huidige toestand | Volgende toestand / Y |         |         |
|------------------|-----------------------|---------|---------|
|                  | C = 0                 | 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    |



### Excitatietabel T-FF

| Q | Q(next) | T |
|---|---------|---|
| 0 | 0       | 0 |
| 0 | 1       | 1 |
| 1 | 0       | 1 |
| 1 | 1       | 0 |



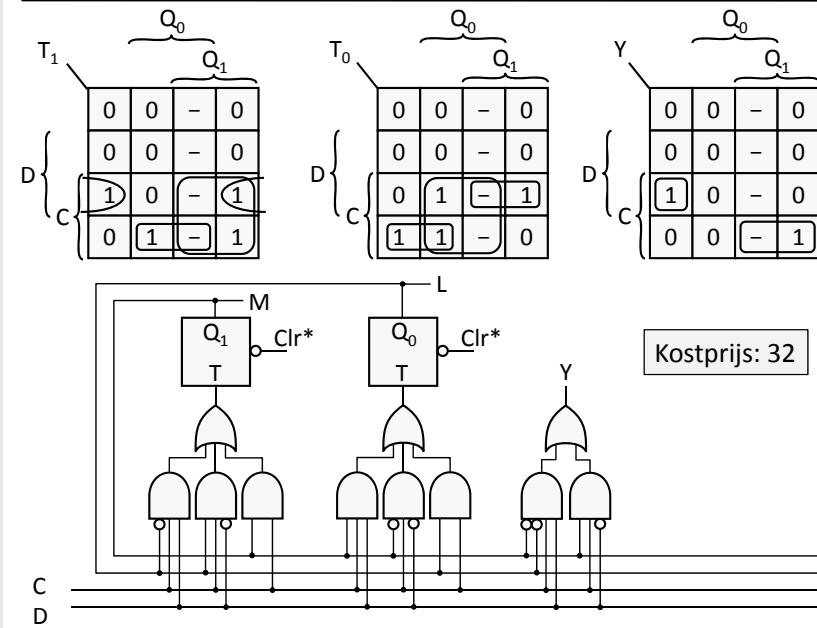
## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchroon

# Realisatie met T-FF



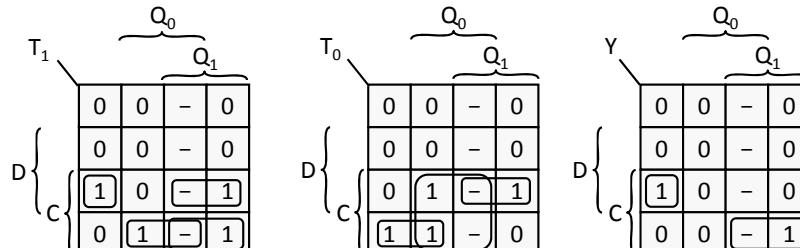
## Sequentiële schakelingen

### Bouwblokken Synchroon

- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie
- Tijds gedrag
- VHDL

### Asynchroon

## Alternatieve realisatie met T-FF



## One-hot-codering: realisatie zonder minimalisering

### One-hot met D-FF:

- Elke overgang die in een toestand toekomt vereist een AND-poort

### One-hot met 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

### Voor JK-FF en T-FF kan men vergelijkbare redeneringen opstellen

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

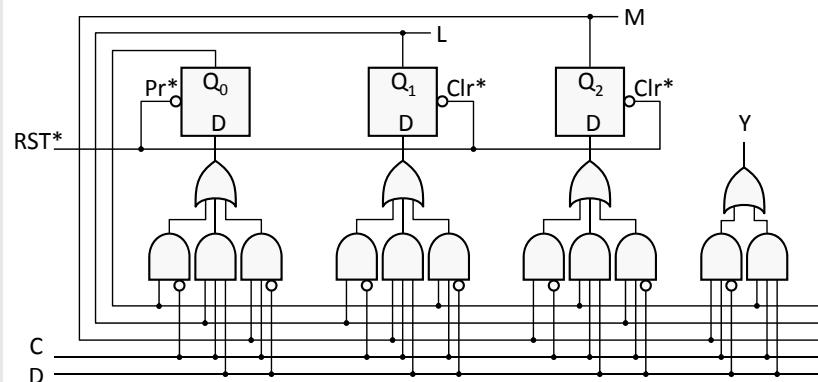
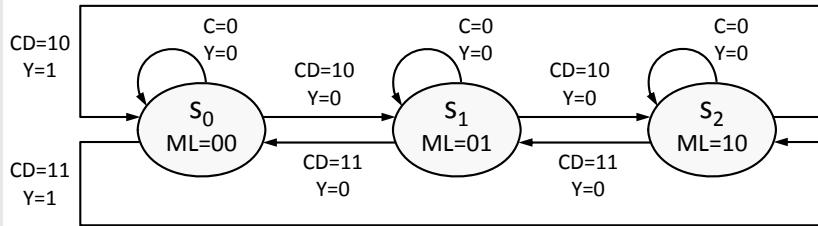
- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

# One-hot-codering met D-FF



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

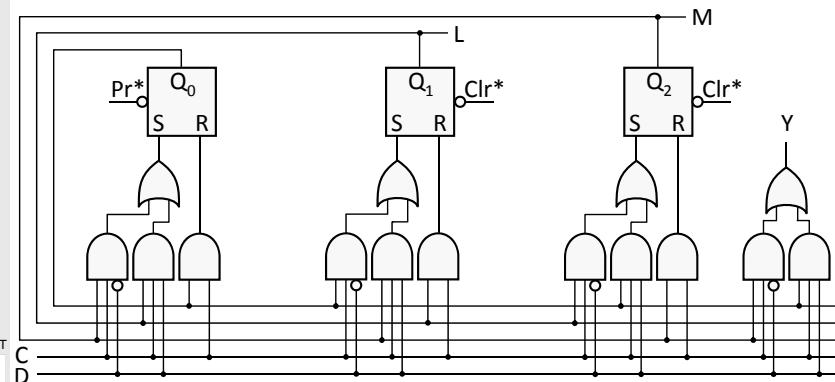
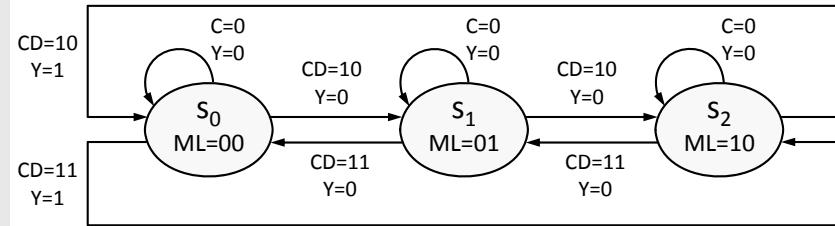
- Voorbeeld
- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
  - keuze FF
- ⇒ realisatie

#### Tijds gedrag

#### VHDL

#### Asynchroon

# One-hot-codering met SR-FF



## Ontwerp sequentiële schakelingen

### Bouwblokken

→ Ontwerp van synchrone sequentiële schakelingen

- Een voorbeeld
- Het ontwerp in detail
- Tijds gedrag: maximale klok frequentie
- FSM in VHDL

□ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

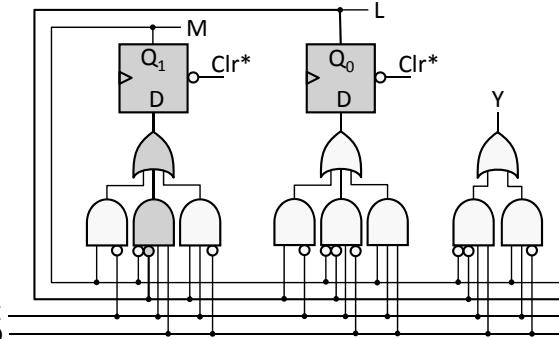
#### Synchroon

- Voorbeeld
- Ontwerp
  - ⇒ Tijds gedrag
  - VHDL
- Asynchroon

# Bepaling maximale klok frequentie

$$f_{\max} = 1/(\text{vertraging van kritisch pad})$$

Kritisch pad = het pad met de langste combinatorische vertraging tussen twee klok flanken



$$\begin{aligned} \text{Vertraging} &= \text{klok} \rightarrow Q_0' + \text{NAND4} + \text{NAND3} + \text{set-up D} \\ &= 4,2 + 2,2 + 1,8 + 1 = 9,2 \end{aligned}$$

stel referentievertraging = 1 ns ⇒  $f_{\max} = 108 \text{ MHz}$

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- Ontwerp
- Tijds gedrag

⇒ VHDL

#### Asynchroon

# Ontwerp sequentiële schakelingen

## □ Bouwblokken

### → Ontwerp van synchrone sequentiële schakelingen

- Een voorbeeld
- Het ontwerp in detail
- Tijds gedrag
- FSM in VHDL

## □ Ontwerp van asynchrone sequentiële schakelingen

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

- Voorbeeld
- Ontwerp
- Tijds gedrag

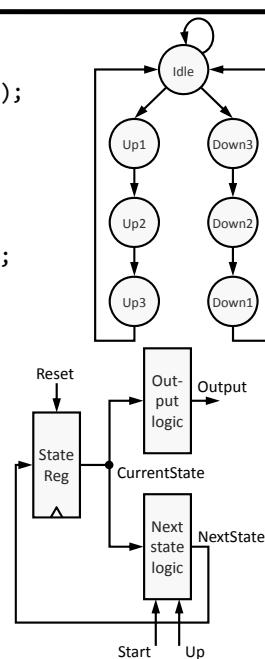
⇒ VHDL

#### Asynchroon

# FSM in VHDL: een voorbeeld

```
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;
```



## Sequentiële schakelingen

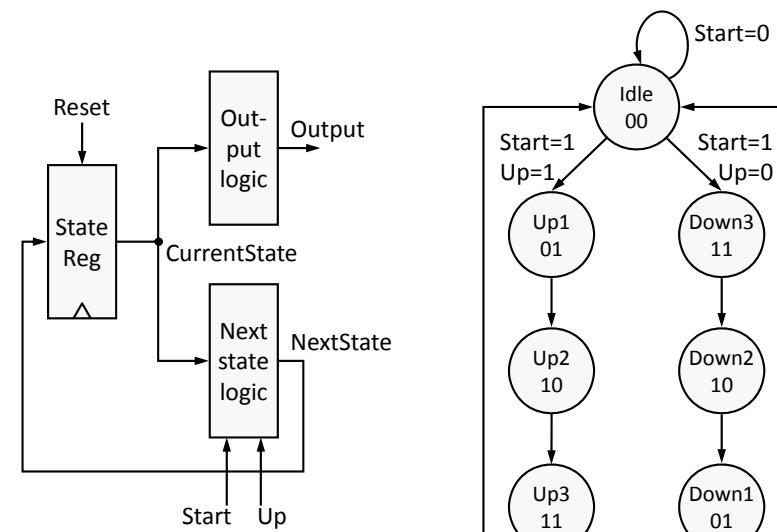
### Bouwblokken

#### Synchroon

- Voorbeeld
- Ontwerp
- Tijds gedrag

⇒ VHDL

# FSM in VHDL: een voorbeeld



## Sequentiële schakelingen

### Bouwblokken

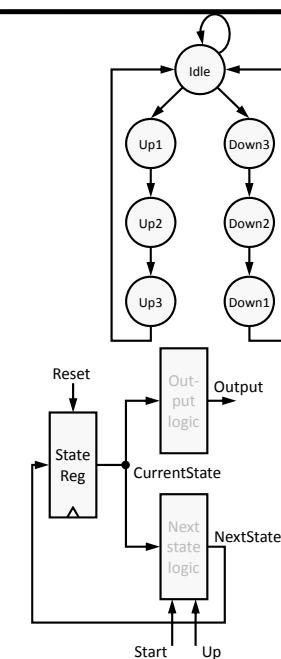
#### Synchroon

- Voorbeeld
- Ontwerp
- Tijds gedrag

⇒ VHDL

# FSM in VHDL: een voorbeeld

```
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;
```

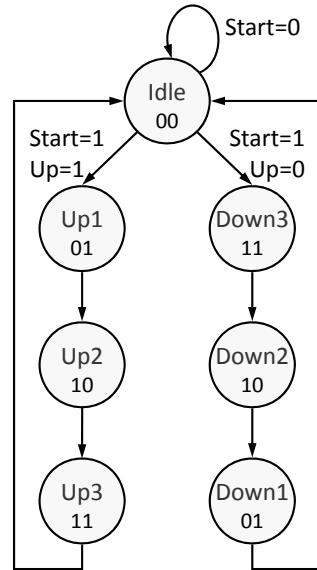


## Sequentiële schakelingen

Bouwblokken  
Synchroon  
▪ Voorbeeld  
▪ Ontwerp  
▪ Tijds gedrag  
⇒ VHDL  
Asynchroon

# FSM in VHDL: een voorbeeld

```
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 Down2 =>
 NextState <= Down1;
 when Down3 =>
 NextState <= Down2;
 end case;
end process NextStateLogic;
```



## Sequentiële schakelingen

Bouwblokken  
Synchroon  
▪ Voorbeeld  
▪ Ontwerp  
▪ Tijds gedrag  
⇒ VHDL  
Asynchroon

# Toestandscodering

Laat dit over aan de synthesis software of ...

- met enum\_encoding attribuut

```
architecture Behav of FSM is
 type FSM_State = (A,B,C);
 attribute enum_encoding : string;
 attribute enum_encoding of FSM_State:type
 is "00 01 11";
 signal CurState, NextState: FSM_State;
begin ...
```

Declaratie van een gebruikersattribuut

Specificatie van het attribuut

- met constanten

```
architecture Behav of FSM is
 constant A: std_logic_vector(0 to 1) := "00";
 constant B: std_logic_vector(0 to 1) := "01";
 constant C: std_logic_vector(0 to 1) := "10";
 signal CurState, NextState:
 std_logic_vector(0 to 1);
begin ...
```

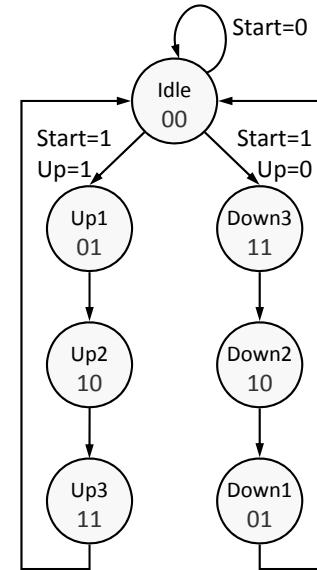
## Sequentiële schakelingen

Bouwblokken  
Synchroon  
▪ Voorbeeld  
▪ Ontwerp  
▪ Tijds gedrag  
⇒ VHDL  
Asynchroon

# FSM in VHDL: een voorbeeld

```
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;
```

Gemakkelijk uitbreidbaar naar Mealy-model



## Sequentiële schakelingen

Bouwblokken  
Synchroon  
▪ Voorbeeld  
▪ Ontwerp  
▪ Tijds gedrag  
⇒ VHDL  
Asynchroon

# Veilige toestanden

Stel dat we een toestandsmachine hebben met 3 toestanden, gecodeerd in 2 bits.

Wat gebeurt er als de FSM in de vierde toestand terecht komt, bijv. t.g.v. ruis, opstarten, ... ?  
Zal het hiervan herstellen?

→ neem voorzorgen in de VHDL-code:

```
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;
```

# Ontwerp sequentiële schakelingen

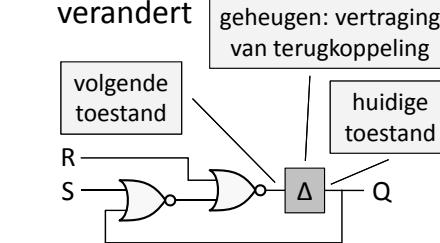
- Bouwblokken
- Ontwerp van synchrone sequentiële schakelingen
- ➔ Ontwerp van asynchrone sequentiële schakelingen

# Ontwerp sequentiële schakelingen

- Bouwblokken
- Ontwerp van synchrone sequentiële schakelingen
- ➔ Ontwerp van asynchrone sequentiële schakelingen
  1. Toestandstabel opstellen
  2. Minimalisering aantal toestanden
  3. Codering toestanden
  4. Realisatie  
(met aandacht voor tijdsgedragproblemen)

# Asynchrone sequentiële schakelingen

= teruggekoppelde schakeling waarvan uitgangen en toestand kunnen veranderen zodra een ingang verandert



| Q | volgende toestand |         |         |         |
|---|-------------------|---------|---------|---------|
|   | SR = 00           | SR = 01 | SR = 10 | SR = 11 |
| 0 | 0                 | 0       | 1       | 0       |
| 1 | 1                 | 0       | 1       | 0       |

stabiele  
schakeling

- Beperking tot fundamentele modus (“fundamental mode restriction”)

- geen twee (of meer) ingangen mogen tegelijkertijd veranderen
- een ingangsverandering mag slechts optreden als alle effecten van de vorige verandering uitgestorven zijn

# Toestanden bij asynchrone schakelingen

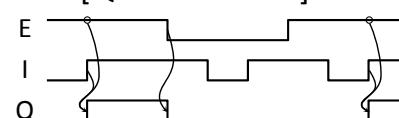
Elke mogelijke combinatie van ingangen *en* uitgangen is een mogelijke toestand!

Voorbeeld: ontwerp een schakeling met een uitgang Q en twee ingangen I en E zodat

1) E = 1 & stijgende flank van I  $\Rightarrow$  Q wordt 1  
[Q wisselt als Q=0 & IE: 01 $\rightarrow$ 11 (b $\rightarrow$ f)]

2) Q blijft 1 tot E 0 wordt  
[Q wisselt als Q=1 & E: 1 $\rightarrow$ 0 (e $\rightarrow$ a; f $\rightarrow$ c)]

3) E = 0  $\Rightarrow$  Q = 0  
[QE=10 kan niet]



| 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          |

## Sequentiële schakelingen

Bouwblokken

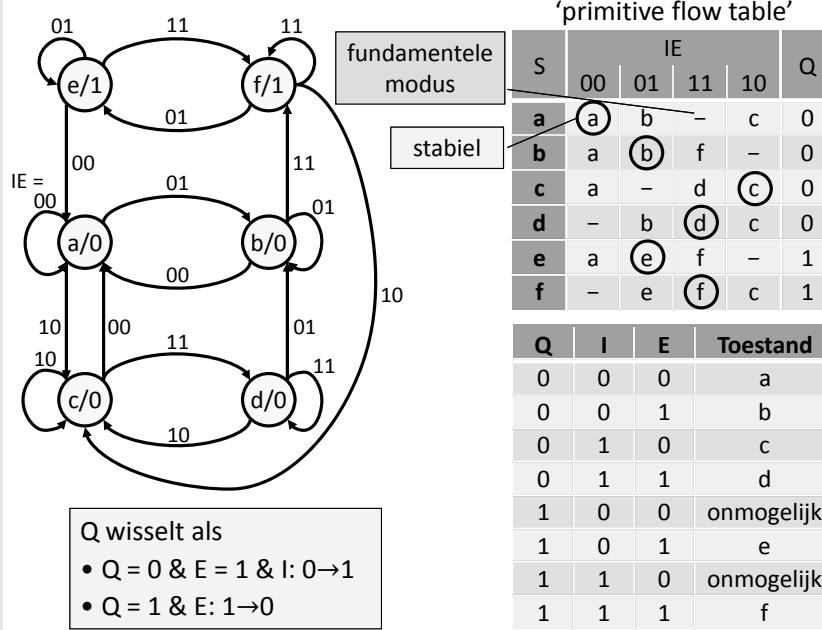
Synchroon

Asynchroon

⇒ Ontwerp

- minimaliseer
- codering
- realisatie

## Toestandstabel ('flow table')



## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Ontwerp sequentiële schakelingen

## □ Bouwblokken

## □ Ontwerp van synchrone sequentiële schakelingen

## ⇒ Ontwerp van asynchrone sequentiële schakelingen

1. Toestandstabel opstellen
2. Minimalisering aantal toestanden
3. Codering toestanden
4. Realisatie

## Sequentiële schakelingen

Bouwblokken

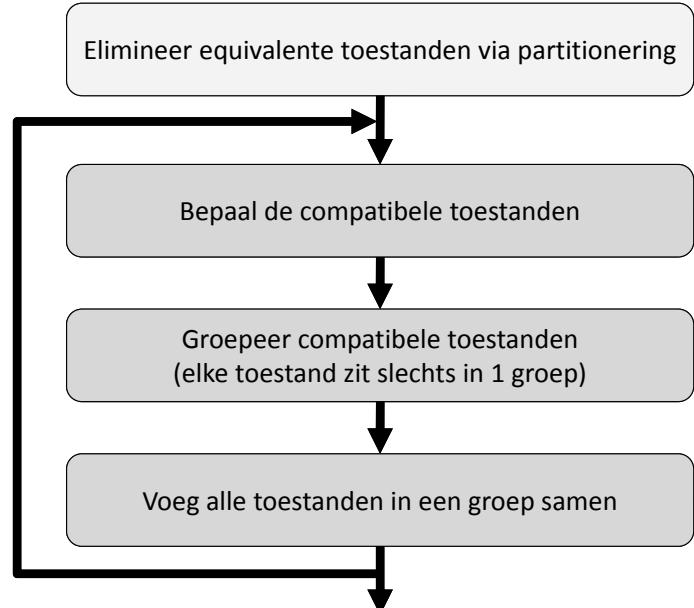
Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Minimalisering toestanden



## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Compatibele toestanden

- = geen toestandsconflicten
  - ⇒ toestanden  $S_i$  en  $S_j$  hebben dezelfde uitgangen
  - ⇒ voor elke ingangscombinatie ofwel
    - zowel  $S_i$  als  $S_j$  zijn stabiel;
    - zowel  $S_i$  als  $S_j$  hebben dezelfde volgende toestand;
    - de volgende toestand van  $S_i$  en/of  $S_j$  is niet bepaald (don't care).
- Verschil met equivalente toestanden
  - Dezelfde volgende toestanden i.p.v. equivalentie volgende toestanden
  - Als don't care moet dit niet bij  $S_i$  en  $S_j$ 
    - ⇒ meer compatibele toestanden
- Niet transitief (equivalente wel!)
  - compatibel(a,b) en compatibel(b,c)  $\Rightarrow$  compatibel(a,c)

## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Elimineer equivalente toestanden

| Toestand | Volgende toestand |     |     |     | Uitgang |
|----------|-------------------|-----|-----|-----|---------|
|          | 00                | 01  | 10  | 11  |         |
| A        | (A)               | F   | C   | -   | 0       |
| B        | -                 | (B) | -   | H   | 1       |
| C        | -                 | (C) | J   | G   | 0       |
| D        | -                 | F   | -   | (D) | 1       |
| E        | G                 | -   | (E) | D   | 1       |
| F        | -                 | (F) | -   | K   | 0       |
| G        | L                 | (G) | -   | J   | 0       |
| H        | -                 | L   | E   | (H) | 1       |
| I        | (I)               | E   | F   | -   | 1       |
| J        | B                 | -   | (J) | -   | 0       |
| K        | -                 | B   | E   | (K) | 1       |
| L        | -                 | (L) | -   | K   | 1       |
| M        | (M)               | L   | E   | -   | 1       |

Mogelijke initiële  
equivalente toestanden:

- zelfde uitgang  
(ACFGJ)(BDEHIKLM)
  - don't care inzelfde kolom  
(A)(C)(F)(G)(J)  
(BDL)(E)(HK)(IM)
- Iteratie:
- IM → {IM, EL, FE}  
(IM) ⇒ (I)(M)
  - BDL → {BFL, HDK}  
(BDL) ⇒ (BL)(D)
  - HK → {LB, EE, HK} : OK
  - BL → {BL, HK} : OK

## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

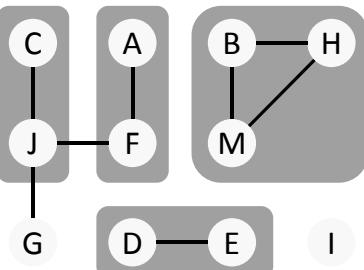
⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Voeg compatibele toestanden samen (1)

| Toestand | Volgende toestand |     |     |     | Uitgang |
|----------|-------------------|-----|-----|-----|---------|
|          | 00                | 01  | 10  | 11  |         |
| A        | (A)               | F   | C   | -   | 0       |
| B        | -                 | (B) | -   | H   | 1       |
| C        | -                 | (C) | J   | G   | 0       |
| D        | -                 | F   | -   | (D) | 1       |
| E        | G                 | -   | (E) | D   | 1       |
| F        | -                 | (F) | -   | H   | 0       |
| G        | B                 | (G) | -   | J   | 0       |
| H        | -                 | B   | E   | (H) | 1       |
| I        | (I)               | E   | F   | -   | 1       |
| J        | B                 | -   | (J) | -   | 0       |
| M        | (M)               | B   | E   | -   | 1       |

Diagram van  
compatibele toestanden



## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

## Voeg compatibele toestanden samen (2)

| Toestand | Volgende toestand |     |     |     | Uitgang |
|----------|-------------------|-----|-----|-----|---------|
|          | 00                | 01  | 10  | 11  |         |
| A        | (A)               | (A) | C   | B   | 0       |
| B        | (B)               | (B) | D   | (B) | 1       |
| C        | B                 | (C) | (C) | G   | 0       |
| D        | G                 | A   | (D) | (D) | 1       |
| G        | B                 | (G) | -   | C   | 0       |
| I        | (I)               | D   | A   | -   | 1       |



A B C

D I

minimaal # toestanden

| Toestand | Volgende toestand |     |     |     | Uitgang |
|----------|-------------------|-----|-----|-----|---------|
|          | 00                | 01  | 10  | 11  |         |
| A        | (A)               | (A) | C   | B   | 0       |
| B        | (B)               | (B) | D   | (B) | 1       |
| C        | B                 | (C) | (C) | (C) | 0       |
| D        | C                 | A   | (D) | (D) | 1       |
| I        | (I)               | D   | A   | -   | 1       |

| Toestand | Volgende toestand |     |     |     | Uitgang |
|----------|-------------------|-----|-----|-----|---------|
|          | 00                | 01  | 10  | 11  |         |
| A        | (A)               | (A) | C   | B   | 0       |
| B        | (B)               | (B) | D   | (B) | 1       |
| C        | B                 | (C) | (C) | (C) | 0       |
| D        | C                 | A   | (D) | (D) | 1       |
| I        | (I)               | D   | A   | -   | 1       |

## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- ⇒ minimaliseer
- codering
- realisatie

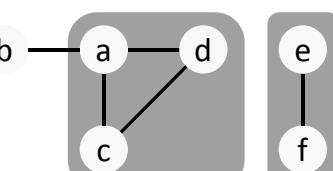
Voorbeeld:  
minimalisering van toestanden

| S | IE  |     |     |     | Q |
|---|-----|-----|-----|-----|---|
|   | 00  | 01  | 11  | 10  |   |
| a | (a) | b   | -   | c   | 0 |
| b | a   | (b) | f   | -   | 0 |
| c | a   | -   | d   | (c) | 0 |
| d | -   | b   | (d) | c   | 0 |
| e | a   | (e) | f   | -   | 1 |
| f | -   | e   | (f) | c   | 1 |

| S | IE  |     |     |     | Q |
|---|-----|-----|-----|-----|---|
|   | 00  | 01  | 11  | 10  |   |
| a | (a) | b   | (a) | (a) | 0 |
| b | a   | (b) | e   | -   | 0 |
| e | a   | (e) | (e) | a   | 1 |

- Elimineer equivalente toestanden
  - zelfde uitgang  
(abcd)(ef)
  - don't care inzelfde kolom  
(a)(b)(c)(d)(e)(f)

- Voeg compatibele toestanden samen



- Slechts 1 iteratie nodig

## Sequentiële schakelingen

Bouwblokken  
Synchroon  
Asynchroon  
⇒ Ontwerp  
• tabel  
• minimaliseer  
⇒ codering  
• realisatie

# Ontwerp sequentiële schakelingen

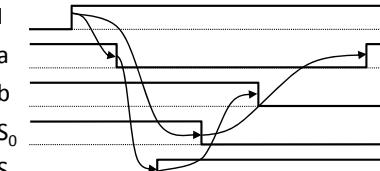
- Bouwblokken
- Ontwerp van synchrone sequentiële schakelingen
- ➔ Ontwerp van asynchrone sequentiële schakelingen
  1. Toestandstabel opstellen
  2. Minimalisering aantal toestanden
  3. Codering toestanden:  
vermijd problematische 'races'
  4. Realisatie

## Sequentiële schakelingen

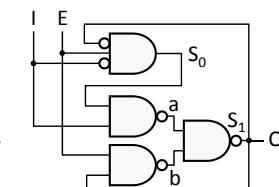
Bouwblokken  
Synchroon  
Asynchroon  
⇒ Ontwerp  
• tabel  
• minimaliseer  
⇒ codering  
• realisatie

# Voorbeelden race

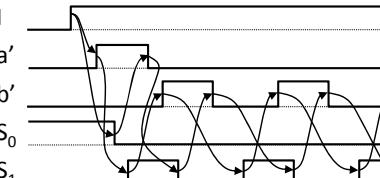
- **Verwacht gedrag:**  $01 \rightarrow 11 \rightarrow 10$   
als  $b = 0$  voor  $a = 1$  opnieuw



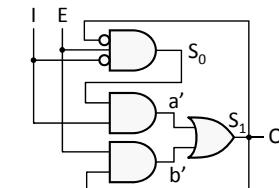
| S  | 00 | 01   | 11   | 10 | Q |
|----|----|------|------|----|---|
| 00 | 00 | 01   | 00   | 00 | 0 |
| 01 | 00 | (01) | 10   | -  | 0 |
| 10 | 00 | 10   | (10) | 00 | 1 |



- **Cycle:**  $01 \rightarrow 00 \rightarrow 10 \rightarrow 00 \rightarrow 10 \rightarrow \dots$   
als  $a' = 0$  terug voor  $b' = 1$



- **Critical race:**  $00 \rightarrow 01$   
als puls  $a' \approx 0$



## Sequentiële schakelingen

Bouwblokken  
Synchroon  
Asynchroon  
⇒ Ontwerp  
• tabel  
• minimaliseer  
⇒ codering  
• realisatie

# Codering van de toestanden

Op dezelfde manier als bij synchrone FSM maar vermijd 'critical races'.

- Race* : het feit dat 2 of meer toestandsvariabelen moeten veranderen als 1 ingangsbit verandert
- *Critical race* = race die tot een verkeerde toestand leidt
  - *Cycle* = race die tot een oscillatie tussen twee toestanden leidt
  - Het optreden van een race en het soort race hangt af van de vertragingskarakteristieken van de poorten

# Eliminatie van 'critical races'

Zorg ervoor dat nooit 2 of meer toestandsvariabelen moeten wijzigen t.g.v. 1 ingangsverandering.

- Zoek een zo goed mogelijke toestandscodering
- Eventueel andere tussentoestanden gebruiken
  - Je kan don't care ook omzetten naar een toestand
- Extra overgangstoestanden toevoegen indien nodig

| Toestand | Volgende toestand |    |    |    | Uitgang |
|----------|-------------------|----|----|----|---------|
|          | 00                | 01 | 10 | 11 |         |
| A        | A                 | B  | C  | A  | 00      |
| B        | B                 | B  | D  | C  | 01      |
| C        | A                 | C  | D  | C  | 10      |
| D        | B                 | -  | D  | A  | 11      |

| Toestand | Volgende toestand |    |    |    | Uitgang |
|----------|-------------------|----|----|----|---------|
|          | 00                | 01 | 10 | 11 |         |
| 00       | 00                | 10 | 10 | 00 | 00      |
| 11       | 11                | 11 | 10 | 01 | 01      |
| 01       | 00                | 01 | 11 | 01 | 10      |
| 10       | 11                | 11 | 10 | 00 | 11      |

Overgangsverschijnselen op uitgangen mogelijk!

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

#### Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- ⇒ codering
- realisatie

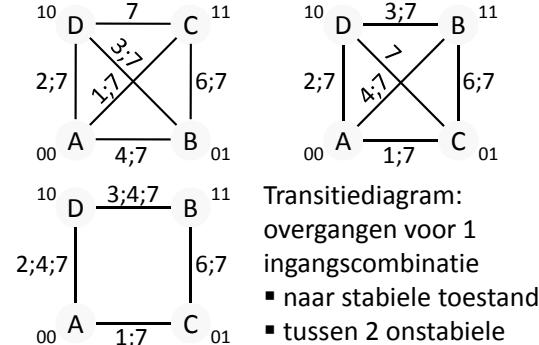
## Toestandscodering: voorbeeld 9.13

| Toestand | Volgende toestand |    |    |    | Uitgang |
|----------|-------------------|----|----|----|---------|
|          | 00                | 01 | 10 | 11 |         |
| A        | A                 | B  | C  | A  | 00      |
| B        | B                 | B  | D  | C  | 01      |
| C        | A                 | C  | D  | C  | 10      |
| D        | B                 | -  | D  | A  | 11      |

| Toestand | Volgende toestand |    |    |    | Uitgang |
|----------|-------------------|----|----|----|---------|
|          | 00                | 01 | 10 | 11 |         |
| 00       | 00                | 10 | 10 | 00 | 00      |
| 11       | 11                | 11 | 10 | 01 | 01      |
| 01       | 00                | 01 | 11 | 01 | 10      |
| 10       | 11                | 11 | 10 | 00 | 11      |

|   |   |   |   |   |
|---|---|---|---|---|
| A | 1 | 4 | 7 | 2 |
| B | 3 | 4 | 7 | 6 |
| C | 1 | 5 | 7 | 6 |
| D | 3 | - | 7 | 2 |

|   |   |   |   |   |
|---|---|---|---|---|
| A | 1 | 4 | 7 | 2 |
| B | 3 | 4 | 7 | 6 |
| C | 1 | 5 | 7 | 6 |
| D | 3 | 4 | 7 | 2 |



Transitiendiagram:  
overgangen voor 1  
ingangscombinatie  
■ naar stabiele toestand  
■ tussen 2 onstabiele

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

#### Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- ⇒ codering
- realisatie

## Wat is de initiële toestand?

Bij het opzetten van de spanning kan de schakeling in een willekeurige toestand opstarten, bijv.  $S = 11$  &  $IE = 00$ .

Wat er dan gebeurt hangt af van de toestanden die de don't cares vervangen na realisatie; eventueel is dit zelf een nieuwe stabiele toestand!

⇒ Vervang de don't cares door een evolutie naar een stabiele toestand met dezelfde ingangen

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

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

Race, maar niet kritisch

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

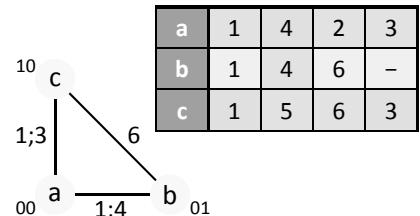
#### Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- ⇒ codering
- realisatie

## Voorbeeld: toestandscodering

| S | IE |    |    |    | Q |
|---|----|----|----|----|---|
|   | 00 | 01 | 11 | 10 |   |
| a | a  | b  | a  | a  | 0 |
| b | a  | b  | c  | -  | 0 |
| c | a  | c  | c  | a  | 1 |



Toevoegen van een extra overgangstoestand nodig

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



## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

#### Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- ⇒ codering
- realisatie

## Wat is de initiële toestand?

Bij het opzetten van de spanning kan de schakeling in een willekeurige toestand opstarten, bijv.  $S = 11$  &  $IE = 00$ .

Wat er dan gebeurt hangt af van de toestanden die de don't cares vervangen na realisatie; eventueel is dit zelf een nieuwe stabiele toestand!

⇒ Vervang de don't cares door een evolutie naar een stabiele toestand met dezelfde ingangen

## Sequentiële schakelingen

### Bouwblokken

#### Synchroon

#### Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- codering
- realisatie

## Ontwerp sequentiële schakelingen

### Bouwblokken

### Ontwerp van synchrone sequentiële schakelingen

- Ontwerp van asynchrone sequentiële schakelingen
  - Toestandstabel opstellen
  - Minimalisering aantal toestanden
  - Codering toestanden
  - Realisatie: hou rekening met mogelijke 'hazards'

## Sequentiële schakelingen

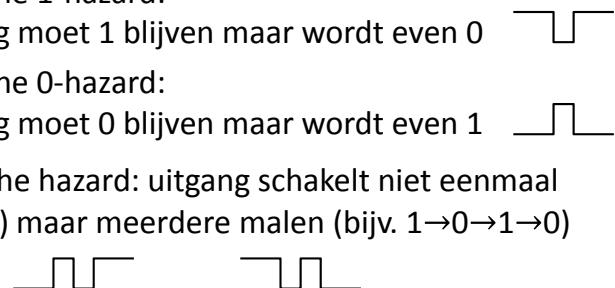
Bouwblokken  
Synchroon

Asynchroon

- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
- ⇒ realisatie

# Hazard (combinatorische schakeling)

- = bijkomende niveauverstoring ('glitch') wegens poortvertragingen
- Statische hazard: constant niveau vertoont verstoring
  - > statische 1-hazard:  
uitgang moet 1 blijven maar wordt even 0
  - > statische 0-hazard:  
uitgang moet 0 blijven maar wordt even 1
- Dynamische hazard: uitgang schakelt niet eenmaal (bijv. 1→0) maar meerdere malen (bijv. 1→0→1→0)



Hazards zijn moeilijk handmatig te detecteren  
⇒ simulatie tijdsgedrag belangrijk

## Sequentiële schakelingen

Bouwblokken  
Synchroon

Asynchroon

- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
- ⇒ realisatie

# Hazards vermijden

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

*Oorzaak van een hazard:* verschil in vertraging van eenzelfde ingang naar eenzelfde uitgang (of toestandsvariabele) langs verschillende paden

*Oplossing:* voeg redundante termen toe om disjuncte gebieden in de Karnaugh-kaart te bedekken

- > 1-hazards: mintermen voor disjuncte 1-gebieden
- > 0-hazards: maxtermen voor disjuncte 0-gebieden

## Sequentiële schakelingen

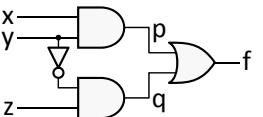
Bouwblokken  
Synchroon

Asynchroon

- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
- ⇒ realisatie

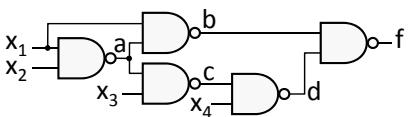
# Voorbeelden van hazards

### □ Statische 1-hazard

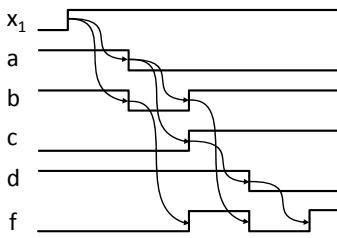
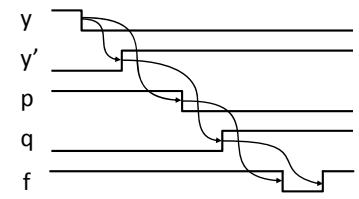


$$x = z = 1 \Rightarrow f = 1$$

### □ Dynamische hazard



$$x_2 = x_3 = x_4 = 1 \Rightarrow f = x_1$$



## Sequentiële schakelingen

Bouwblokken  
Synchroon

Asynchroon

- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
- ⇒ realisatie

# Hazards vermijden

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

*Oorzaak van een hazard:* verschil in vertraging van eenzelfde ingang naar eenzelfde uitgang (of toestandsvariabele) langs verschillende paden

*Oplossing:* voeg redundante termen toe om disjuncte gebieden in de Karnaugh-kaart te bedekken

- > 1-hazards: mintermen voor disjuncte 1-gebieden
- > 0-hazards: maxtermen voor disjuncte 0-gebieden

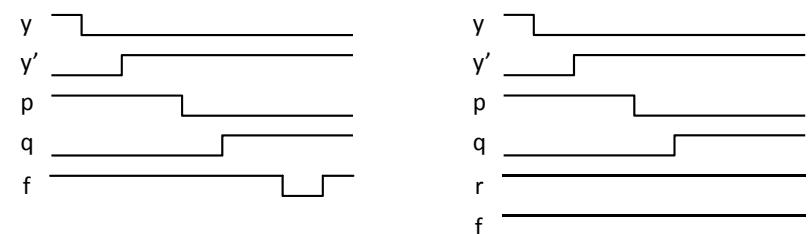
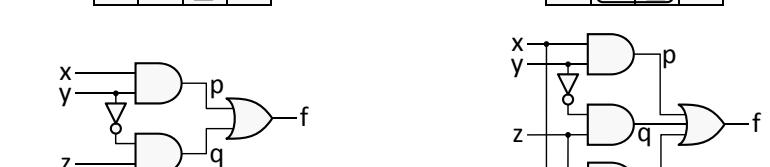
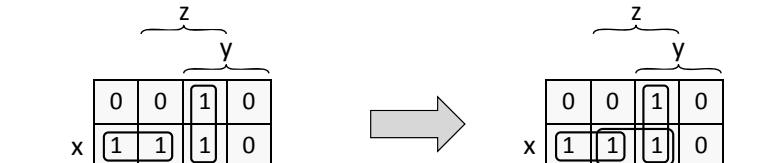
## Sequentiële schakelingen

Bouwblokken  
Synchroon

Asynchroon

- ⇒ Ontwerp
  - tabel
  - minimaliseer
  - codering
- ⇒ realisatie

# Hazards vermijden



## Sequentiële schakelingen

Bouwblokken

Synchroon

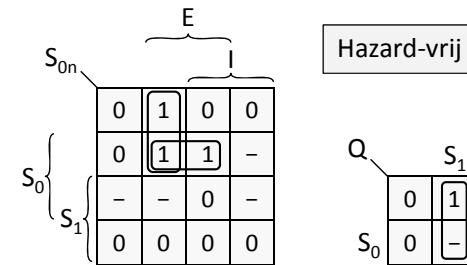
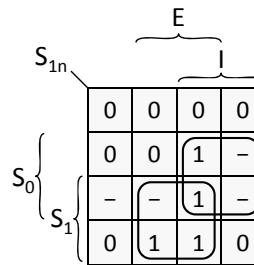
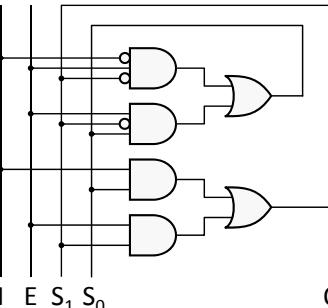
Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- codering
- ⇒ realisatie

## Voorbeeld: realisatie

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



## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- codering
- ⇒ realisatie

## Essentiële hazard

= één enkele ingangsverandering brengt de schakeling in een verkeerde toestand

## □ Hoe detecteren?

- Andere eindtoestand als een ingang 1 keer of 3 keer verandert

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

## □ Hoe vermijden?

- Toestandsvariabelen mogen enkel veranderen als alle ingangsveranderingen aan de poorten gekomen zijn
- ⇒ zorgvuldig ontwerp op elektrisch niveau, bijv. door extra vertragingen in te voeren

## Sequentiële schakelingen

Bouwblokken

Synchroon

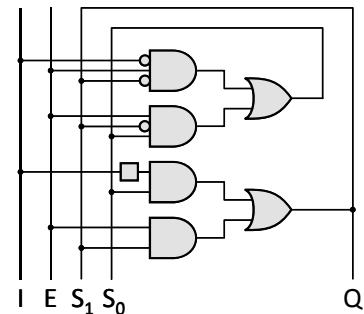
Asynchroon

⇒ Ontwerp

- tabel
- minimaliseer
- codering
- ⇒ realisatie

## Problemen t.g.v. 'skew' op ingangen

Veronderstel een extra vertraging van I op slechts 1 plaats in de schakeling (bijv. door gebruik van een lange lijn)



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

- start met S=00 & IE=11
- I wordt nul : S zou 01 moeten worden
- dan wordt de vertraagde versie van I nul
- toestand verandert niet meer en eindigt in S=10

## Sequentiële schakelingen

Bouwblokken

Synchroon

Asynchroon

▪ Ontwerp

## Besluit

*Vermijd asynchrone sequentiële schakelingen!*

- Veel ingewikkelder ontwerp
- Veel meer gevaren qua tijdsgedrag ('races' & 'hazards')
- Niet of slecht ondersteund door CAD-software

Slechts bruikbaar en zinvol

- voor kleine schakelingen tussen twee synchrone eilanden met verschillende klokken
- als snelheid belangrijk is



# Inhoudstafel

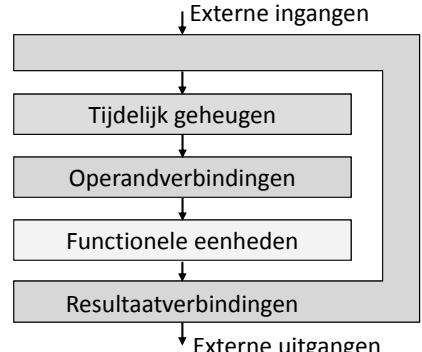
- Inleiding
- De basis van digitaal ontwerp
- Technologische randvoorwaarden
- Combinatorische schakelingen
- Sequentiële schakelingen
- ➔ Niet-programmeerbaar processoren
- Programmeerbare processoren

## Datapad

Drie delen:

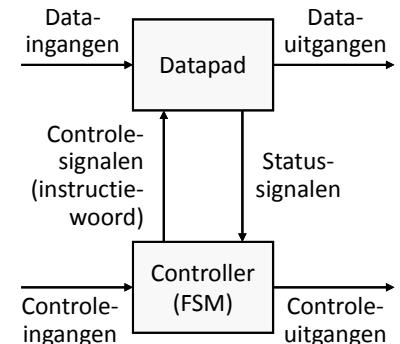
- *Functionele eenheden (FU)*, die datomanipulaties uitvoeren:  
ALU, vermenigvuldiger, rotator, comparator, ...
- *Tijdelijk geheugen*, voor tussenresultaten:  
register(bank), RAM, ...
- *Verbindingen*: bussen met MUX & 3-state buffers

Registertransfer:  
 $\text{register} \leftarrow \text{FU}(\text{registers})$



## Niet-programmeerbaar processor

- = ‘Finite State Machine with Data path’
- Bevat een datapad voor de berekeningen en een controller die het datapad zegt wanneer welke berekeningen op welke data moet uitgevoerd worden
- De controller voert altijd hetzelfde programma uit ➔ niet-programmeerbaar



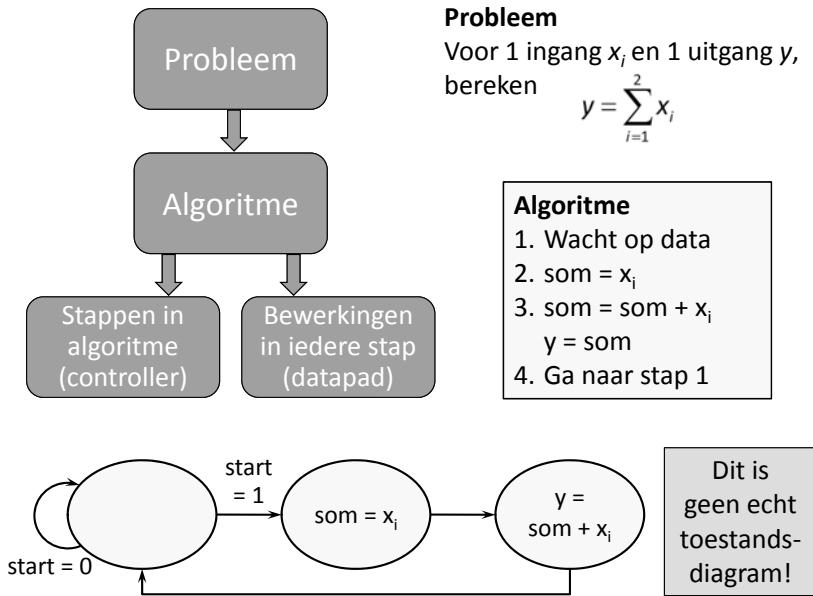
## Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

**Beschrijving**

- ASM
- Geheugen
- Synthese
- Tijds gedrag
- VHDL

# Van probleem naar toestands beschrijving



- Beschrijving**
- ⇒ ASM
  - Geheugen
  - Synthese
  - Tijds gedrag
  - VHDL

## Niet-programmeerbare processoren

- Beschrijving van een algoritme
  - ASM-schema
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

- Beschrijving**
- ASM
  - Geheugen
  - Synthese
  - Tijds gedrag
  - VHDL

# Toestand-actie tabel

= toestands beschrijving in tabelvorm

- conditionele tabel: per toestand enkel de relevante ingangs combinaties
- actie = geef verandering van variabele aan

| Huidige toestand | Volgende toestand Conditie | Uit-gang | Controle- & datapad-acties Conditie | Acties          |
|------------------|----------------------------|----------|-------------------------------------|-----------------|
| $S_0$            | start = 0                  | $S_0$    |                                     |                 |
|                  | start = 1                  | $S_1$    |                                     |                 |
| $S_1$            |                            | $S_2$    |                                     | $som = x_i$     |
| $S_2$            |                            | $S_0$    | klaar                               | $y = som + x_i$ |

- Kan vertaald worden naar een toestandstabel (maar deze voorstelling is veel compacter)

- Beschrijving**
- ⇒ ASM
  - Geheugen
  - Synthese
  - Tijds gedrag
  - VHDL

## 'Algorithmic-State-Machine chart'

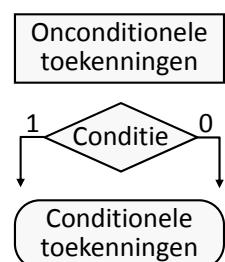
= visuele voorstelling toestand-actie tabel

- Eenvoudiger te verstaan voor een mens
- Elke rij in een toestand-actie tabel komt overeen met een ASM-blok
- Een ASM-blok bestaat uit een of meerdere ASM-elementen

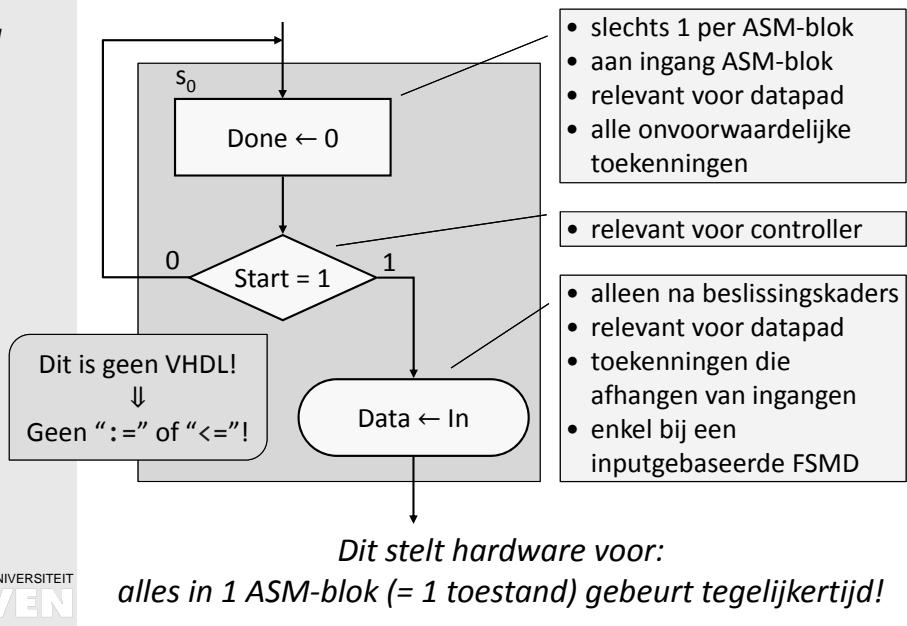
ASM-blok = klokcyclus  
⇒ ASM-schema bevat tijds gedrag informatie!

□ Drie soorten ASM-elementen:

- *toestandskader* ('state box')
- *beslissingskader* ('decision box')
- *conditioneel kader* ('condition box')



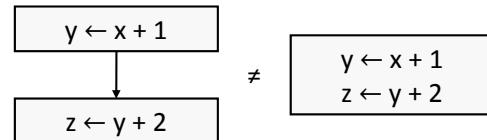
# Voorbeeld ASM-blok



# Meerdere uitdrukkingen in 1 kader

- Als er meerdere uitdrukkingen in één kader staan, worden ze in parallel uitgevoerd.

Programma:

$$\begin{aligned} y &= x + 1 \\ z &= y + 2 \end{aligned}$$


≡

≠

- Vermits ze in parallel uitgevoerd worden is hun volgorde onbelangrijk.



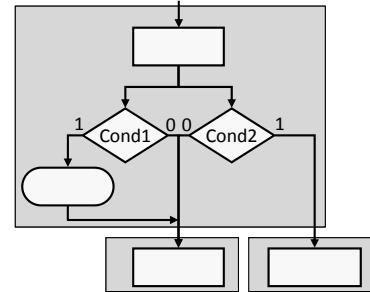
≡

Dit geldt niet alleen voor ASM-toestandskaders maar voor een ASM-blok als geheel!

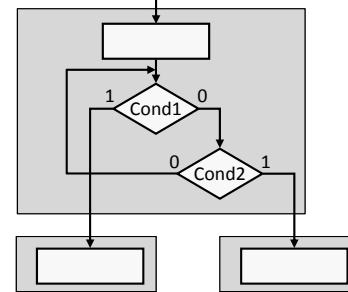
# Geldige ASM-blokken

In een geldig ASM-blok moet elke combinatie van ingangen tot exact 1 volgende toestand leiden.

Voorbeelden van ongeldige ASM-blokken:



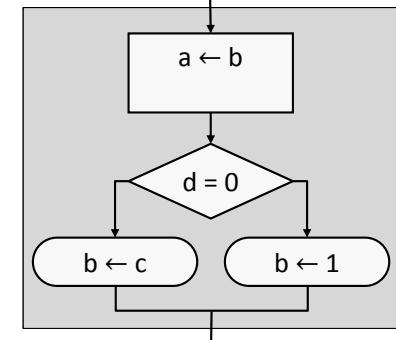
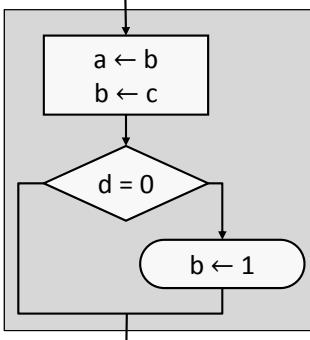
Als Cond2 = 1 zijn er twee volgende toestanden!



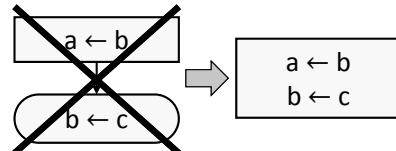
Als Cond1 = 0 en Cond2 = 0 is er geen volgende toestand!

# In een ASM-blok ...

- krijgt een variabele maar 1 nieuwe waarde

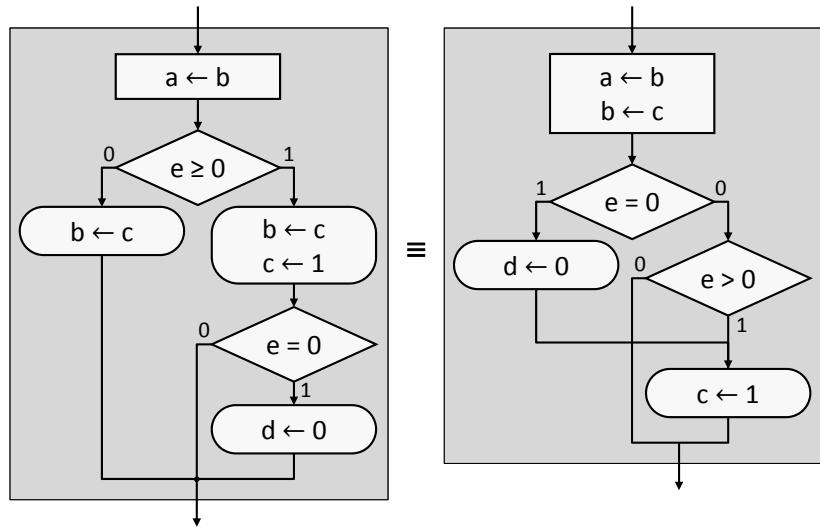


Enkel conditionele kaders na een beslissingskader!



## In een ASM-blok ...

- gebeurt alles tegelijkertijd



## En combinatorische uitgangen?

- Bijv. een OR-poort:  $z = x \text{ or } y$

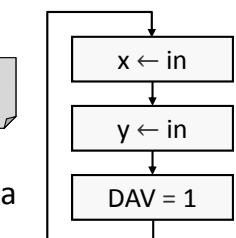
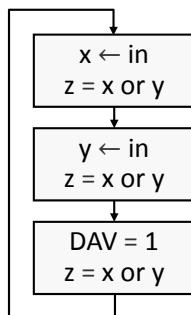
### In het ASM-schema

- toestandsonafhankelijke controller-uitgang
- herhaal in elke ASM-blok
- overladen ASM-schema!

### Buiten het ASM-schema

- niet in ASM-schema maar apart bewaren
- makkelijker te vergeten maar overzichtelijker ASM-schema

$z = x \text{ or } y$



## Hoe controller-uitgangen aanduiden?

In toestandskader of conditioneel kader:

### Datapad-acties

- geven een wijziging aan van de inhoud van een (ASM-)variabele (= register in het datapad)

$y \leftarrow x + 1$

- wanneer afwezig: inhoud blijft bewaard

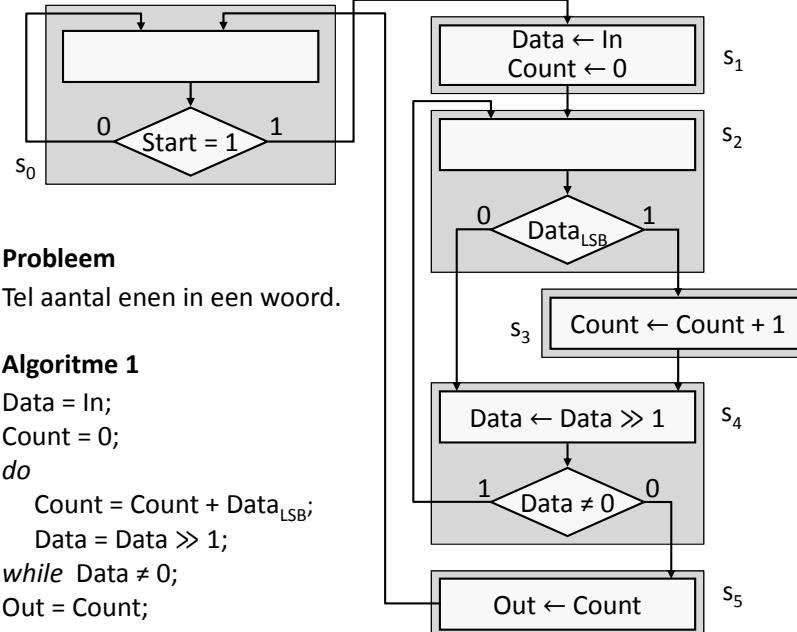
### Controller-uitgangen

- geeft de waarde in die toestand van de controller-uitgang, traditioneel wanneer die 1 is  
⇒ waarde moet niet vermeld worden
- wanneer afwezig: uitgang is 0

Done = 1

Done

## Toestand gebaseerd ASM-schema



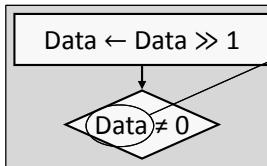
### Probleem

Tel aantal enen in een woord.

### Algoritme 1

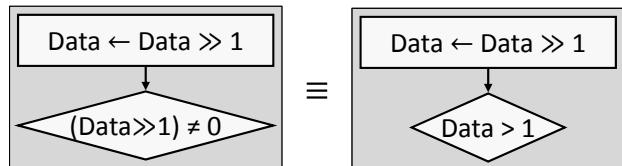
```
Data = In;
Count = 0;
do
 Count = Count + DataLSB;
 Data = Data ≫ 1;
while Data ≠ 0;
Out = Count;
```

# Opgelet ...

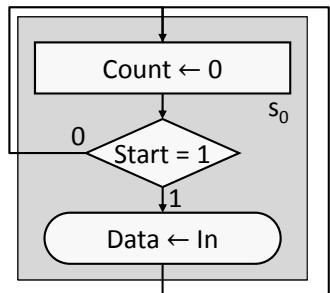


Welke Data wordt getest?  
De oude (niet geschoven) of  
de nieuwe (geschoven)?

- ❑ De nieuwe is wat we verwachten bij traditionele stroomschema's, maar we gebruiken de oude volgens de regels ("alles in één ASM-blok, d.w.z. één toestand, gebeurt tegelijkertijd")
- ❑ Gebruik anders:



# Inputgebaseerd ASM-schema 2



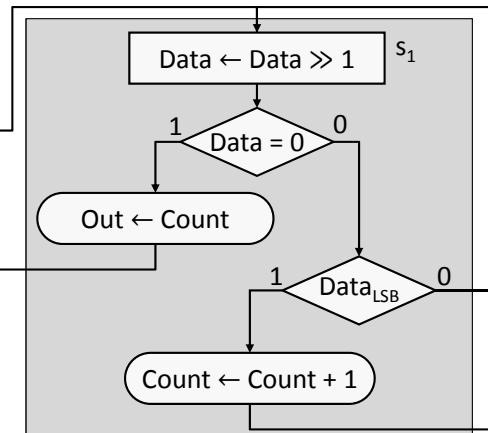
Zowel "Count ← 0" als "Data ← In"  
kan al dan niet conditioneel zijn.

## Algoritme 2

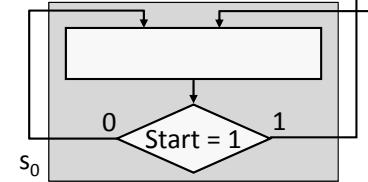
```

Data = In;
Count = 0;
while Data ≠ 0 do
 Count = Count + DataLSB;
 Data = Data >> 1;
end while;
Out = Count;

```



# Inputgebaseerd ASM-schema



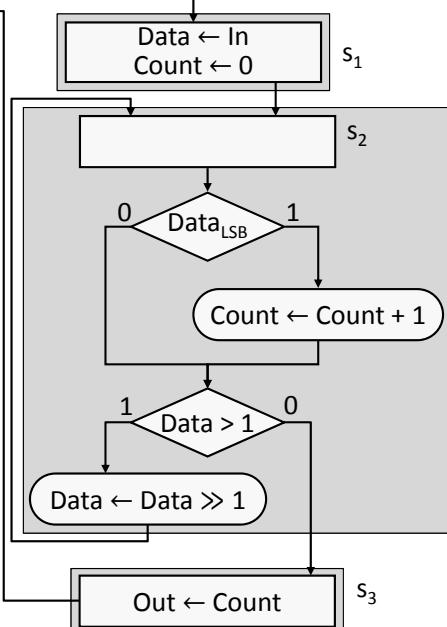
Slechts 4 toestanden ↔  
6 als toestand gebaseerd

## Algoritme 1

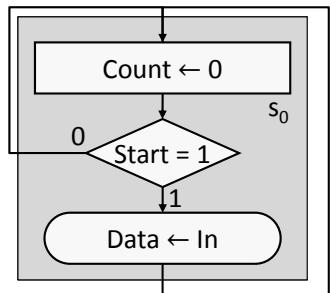
```

Data = In;
Count = 0;
do
 Count = Count + DataLSB;
 Data = Data >> 1;
 while Data ≠ 0;
 Out = Count;

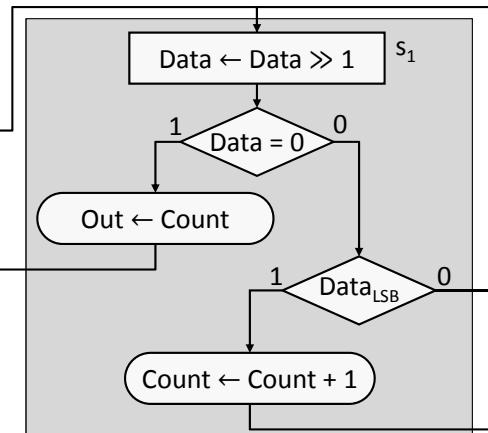
```



# Inputgebaseerd ASM-schema 2



Zowel "Count ← 0" als "Data ← In"  
kan al dan niet conditioneel zijn.



# Niet-programmeerbare processoren

- ❑ Beschrijving van een algoritme
- ➔ Extra geheugen bouw blokken
  - Registerbank
  - RAM
  - LIFO & FIFO
- ❑ Synthese: naar een minimale hardware
- ❑ Aandachtspunten qua tijds gedrag
- ❑ VHDL voor synthese en simulatie

## Beschrijving

## Geheugen

⇒ Registerbank

- RAM
- LIFO
- FIFO

## Synthese

## Tijds gedrag

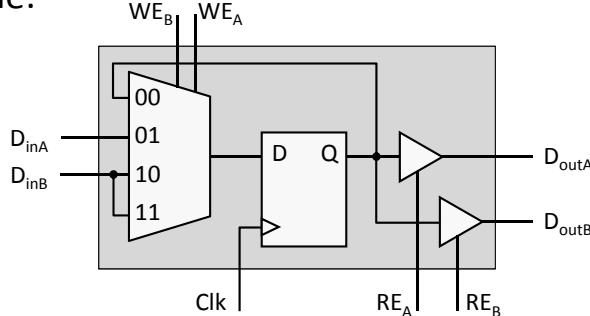
VHDL

# 'Register File Cell'

= geklokte geheugencel voor 1 bit

- WE=1 : schrijf een bit in de cel (op klok!)
  - Hooguit 1 WE actief tegelijkertijd
- RE=1 : lees een bit uit de cel (niet geklokt!)
  - # RE = # leespoorten van registerbank

Realisatie:



## Beschrijving

## Geheugen

⇒ Registerbank

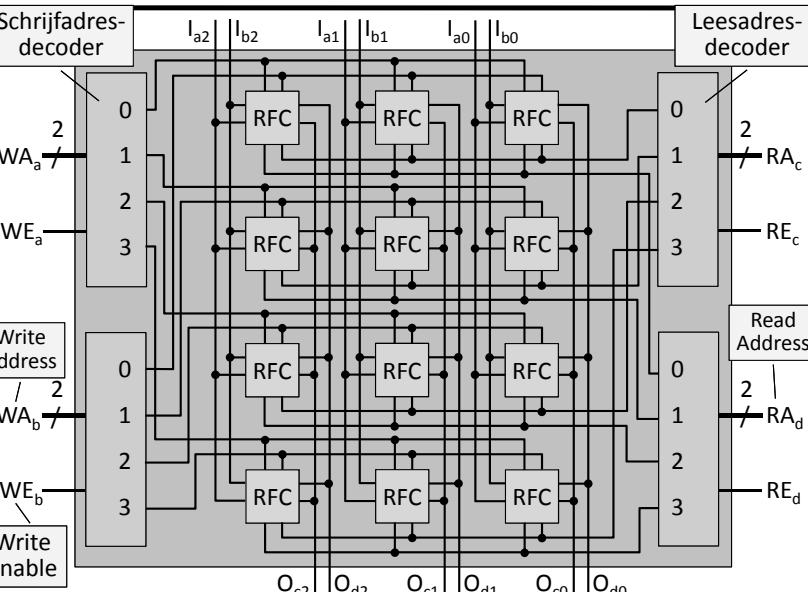
- RAM
- LIFO
- FIFO

## Synthese

## Tijds gedrag

VHDL

# $4 \times 3$ registerbank met 2 schrijf- en 2 leespoorten



□ 'dual port' registerbank = 1 schrijf- en 1 leespoort

## Beschrijving

## Geheugen

- Registerbank

⇒ RAM

▪ LIFO

▪ FIFO

## Synthese

## Tijds gedrag

VHDL

# Niet-programmeerbare processoren

□ Beschrijving van een algoritme

➔ Extra geheugen bouwblokken

- Registerbank
- ➔ RAM
- LIFO & FIFO

□ Synthese: naar een minimale hardware

□ Aandachtspunten qua tijds gedrag

□ VHDL voor synthese en simulatie

## Beschrijving

## Geheugen

- Registerbank

⇒ RAM

▪ LIFO

▪ FIFO

## Synthese

## Tijds gedrag

VHDL

# 'Random Access Memory'

= niet-sequentiële geheugentoegang mogelijk

= wijzigbaar geheugen,  
meestal vluchtig (↔ ROM)

□ Vergelijkbaar met registerbank maar

- groter geheugen

- compactere RFC
  - Statische RAM: FF-geheugen (4 à 6 transistors)
  - Dynamische RAM: condensator (1 transistor)

- trager

⇒ niet voor kleine tijdelijke tussenresultaten

- 1 poort (gecombineerd lees/schrijf)

- lezen: RE = CS · R/W\*

- schrijven: klok = CS · (R/W\*)' ⇒ niet geklokt!

CS = Chip Select

**Beschrijving****Geheugen**

- Registerbank

⇒ RAM

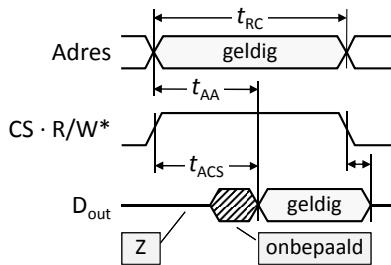
▪ LIFO

▪ FIFO

**Synthese****Tijds gedrag**

VHDL

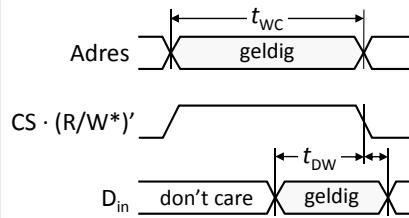
# Tijds gedrag RAM

**□ Lezen ( $R/W^* = 1$ )**

2147H SRAM (4k × 1)

|           |                     | Min  | Max  |
|-----------|---------------------|------|------|
| $t_{RC}$  | cyclustijd lezen    | 35ns |      |
| $t_{AA}$  | toegangstijd        |      | 35ns |
| $t_{ACS}$ | CS toegangstijd     |      | 35ns |
| $t_{HZ}$  | $CS' \rightarrow Z$ | 0ns  | 30ns |

toegangstijd = vertraging RAM

**□ Schrijven ( $R/W^* = 0$ )**

|          |                      | Min  |
|----------|----------------------|------|
| $t_{WC}$ | cyclustijd schrijven | 35ns |
| $t_{DW}$ | data set-up-tijd     | 20ns |
| $t_{DH}$ | data houdtijd        | 0ns  |

**Beschrijving****Geheugen**

- Registerbank

⇒ RAM

▪ LIFO

▪ FIFO

**Synthese****Tijds gedrag**

VHDL

# Geheugen met impliciete adressen

**'Last In First Out'**

(stapelgeheugen, 'stack')

|       |      |
|-------|------|
| Top   | 10   |
| Top-1 | 23   |
| Top-2 | 45   |
| Top-3 | leeg |
| Top-4 | leeg |
| Top-5 | leeg |
| Top-6 | leeg |
| Top-7 | leeg |

1. Reset

2. Schrijf 45

3. Schrijf 23

4. Schrijf 12

5. Lees

LIFO → 12

FIFO → 45

6. Schrijf 10

Terminologie LIFO:

- 'push' = schrijf
- 'pop' = lees

**'First In First Out'**

(als buffergeheugen)

|       |      |
|-------|------|
| Top   | 10   |
| Top-1 | 12   |
| Top-2 | 23   |
| Top-3 | leeg |
| Top-4 | leeg |
| Top-5 | leeg |
| Top-6 | leeg |
| Top-7 | leeg |

**Beschrijving****Geheugen**

- Registerbank

⇒ RAM

▪ LIFO

▪ FIFO

**Synthese****Tijds gedrag**

VHDL

# Niet-programmeerbare processoren

**□ Beschrijving van een algoritme**

→ Extra geheugen bouwblokken

➤ Registerbank

➤ RAM

→ LIFO &amp; FIFO

**□ Synthese: naar een minimale hardware****□ Aandachtspunten qua tijds gedrag****□ VHDL voor synthese en simulatie****Beschrijving****Geheugen**

- Registerbank

⇒ RAM

▪ LIFO

▪ FIFO

**Synthese****Tijds gedrag**

VHDL

# Geheugen met impliciete adressen

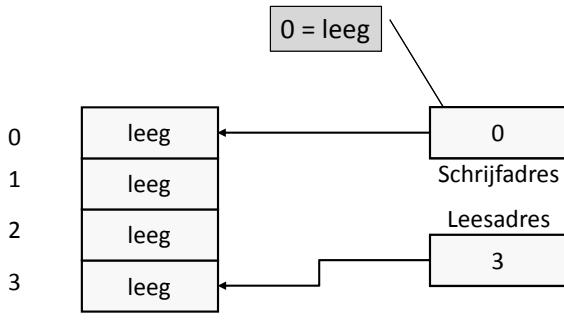
**'Last In First Out'**

(stapelgeheugen, 'stack')

**'First In First Out'**

(als buffergeheugen)

# Implementatie LIFO



Implementatie kan ook  
met schuifregisters  
voor kleine dieptes

**Beschrijving****Geheugen**

- Registerbank

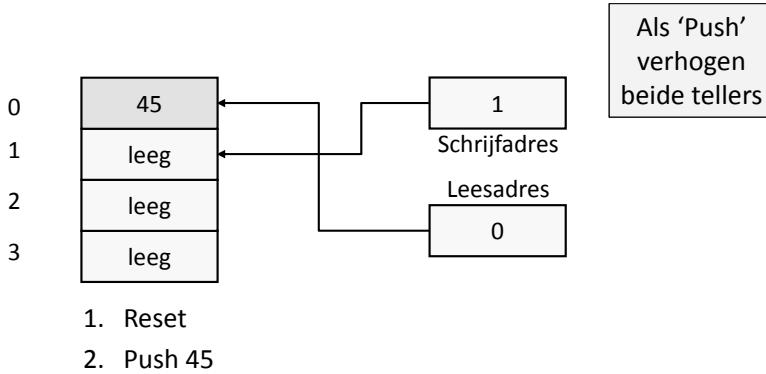
## ▪ RAM

⇒ LIFO

▪ FIFO

**Synthese****Tijds gedrag****VHDL**

# Implementatie LIFO

**Beschrijving****Geheugen**

- Registerbank

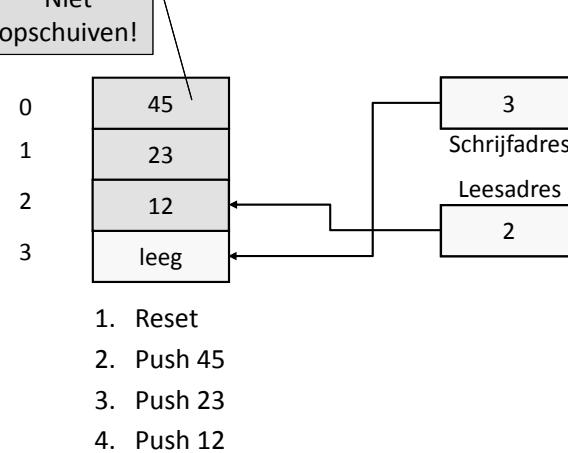
## ▪ RAM

⇒ LIFO

▪ FIFO

**Synthese****Tijds gedrag****VHDL**

# Implementatie LIFO

**Beschrijving****Geheugen**

- Registerbank

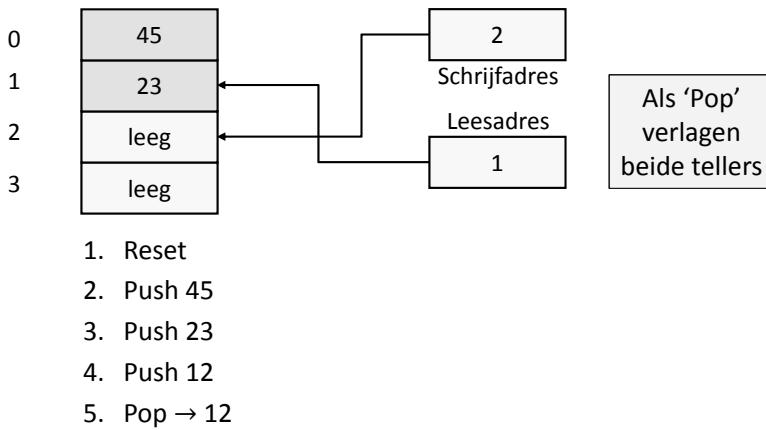
## ▪ RAM

⇒ LIFO

▪ FIFO

**Synthese****Tijds gedrag****VHDL**

# Implementatie LIFO

**Beschrijving****Geheugen**

- Registerbank

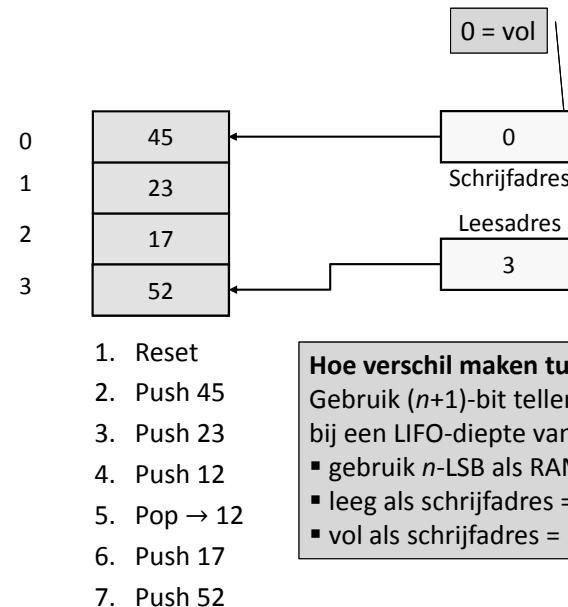
## ▪ RAM

⇒ LIFO

▪ FIFO

**Synthese****Tijds gedrag****VHDL**

# Implementatie LIFO



## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ⇒ LIFO

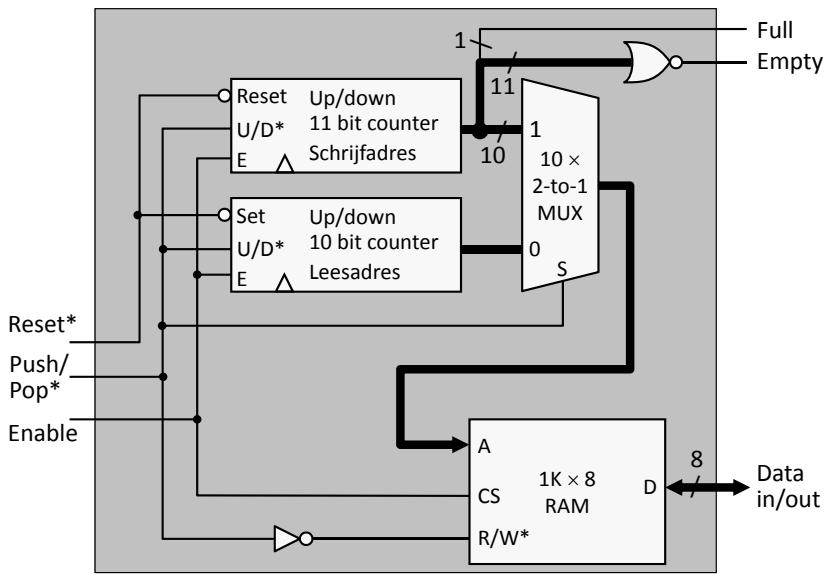
## ▪ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie LIFO



## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ⇒ FIFO

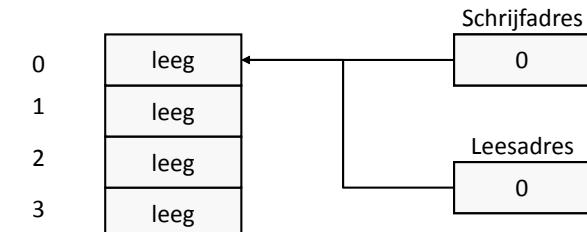
## ▪ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie FIFO



1. Reset

## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ⇒ FIFO

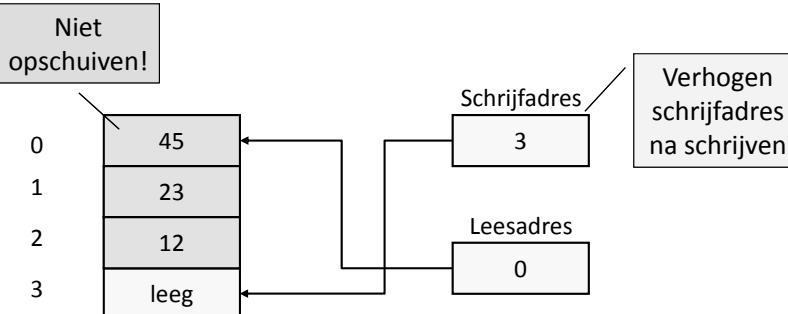
## ▪ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie FIFO



1. Reset
2. Schrijf 45
3. Schrijf 23
4. Schrijf 12

## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ⇒ FIFO

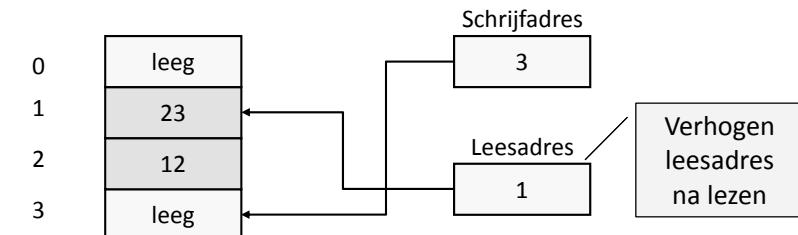
## ▪ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie FIFO



1. Reset
2. Schrijf 45
3. Schrijf 23
4. Schrijf 12
5. Lees → 45

## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ▪ LIFO

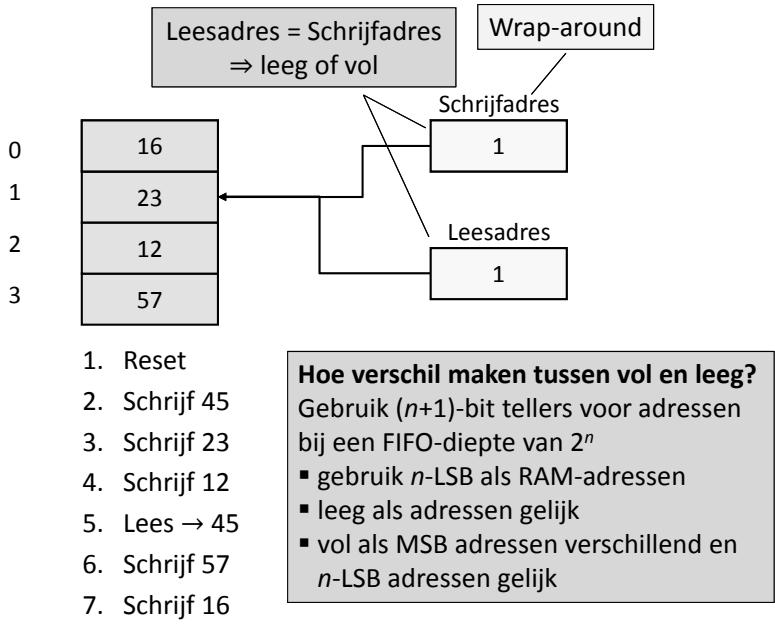
## ⇒ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie FIFO



## FSMD

## Beschrijving

## Geheugen

## Synthese

- Basis

## ▪ Controller

## ▪ Datapad

## ▪ Extra

## Tijds gedrag

## VHDL

## Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basisprincipes
  - Sneller ontwerp van controller
  - Minimalisering datapad
  - Andere optimaliseringen
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

## FSMD

## Beschrijving

## Geheugen

- Registerbank

## ▪ RAM

## ▪ LIFO

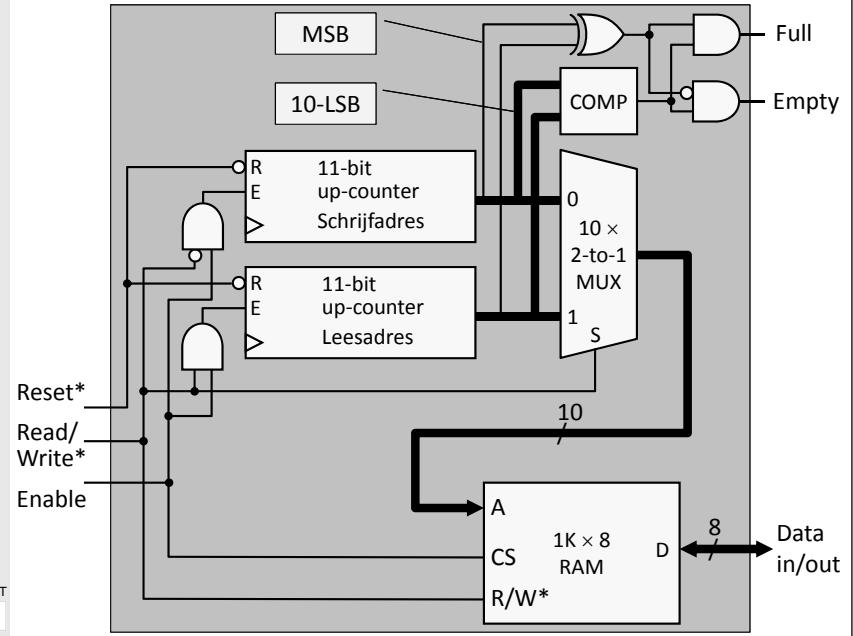
## ⇒ FIFO

## Synthese

## Tijds gedrag

## VHDL

## Implementatie FIFO



## FSMD

## Beschrijving

## Geheugen

## Synthese

- ⇒ Basis

## ▪ Controller

## ▪ Datapad

## ▪ Extra

## Tijds gedrag

## VHDL

## Basisprincipe synthese

- Vertrekkend van een toestand-actie tabel of een ASM-schema, kan een FSMD gerealiseerd worden als volgt:
  - elke variabele komt overeen met een register
  - elke operatie komt overeen met een FU
  - het gebruiken (lezen) van een variabele komt met een verbinding van een register naar een FU overeen
  - het wijzigen (schrijven) van een variabele komt met een verbinding van een FU naar een register overeen
  - elke ASM-blok (of een toestand van een toestand-actie tabel) komt met een toestand van de controller overeen
- Dit leidt niet tot een minimale realisatie!
  - Bijv. elke '+' resulteert in een aparte opteller
- Mogelijke minimaliseringen
  - controller = minimalisering FSM (zie vroeger)
  - datapad : hergebruik registers, FU en verbindingen
  - alternatief ASM-schema

# Voorbeeld synthese

**Probleem:** tel het aantal bits dat 1 is in een woord

**Algoritme 3:**

```
Data = In; Cnt = 0; Mask = 1;
while Data ≠ 0 do
 Temp = Data AND Mask;
 Cnt = Cnt + Temp; Data = Data ≫ 1;
end while;
Out = Cnt;
```

**Opmerkingen:**

- Alle uitdrukkingen die tegelijkertijd (d.w.z. in parallel) kunnen uitgevoerd worden zijn hier op eenzelfde lijn gezet.
- Belangrijk verschil met software programma:
  - programma-instructies worden sequentieel uitgevoerd, slechts één op een bepaald ogenblik
  - hardware componenten werken altijd en in parallel
- Compromis mogelijk: snelheid/performantie ↔ kostprijs

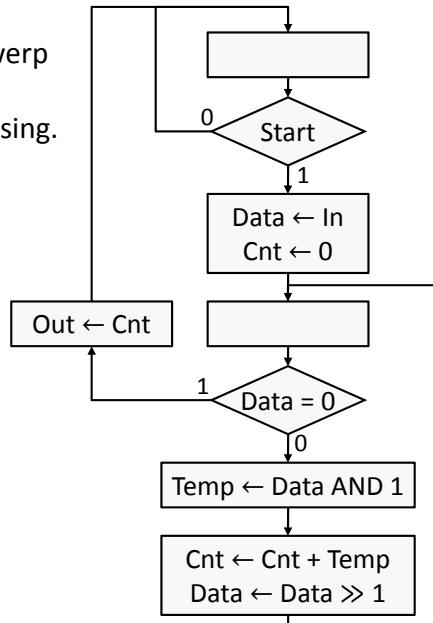
# Synthese: ASM-schema

Voor de eenvoud van ontwerp kiezen we hier voor een toestand gebaseerde oplossing.

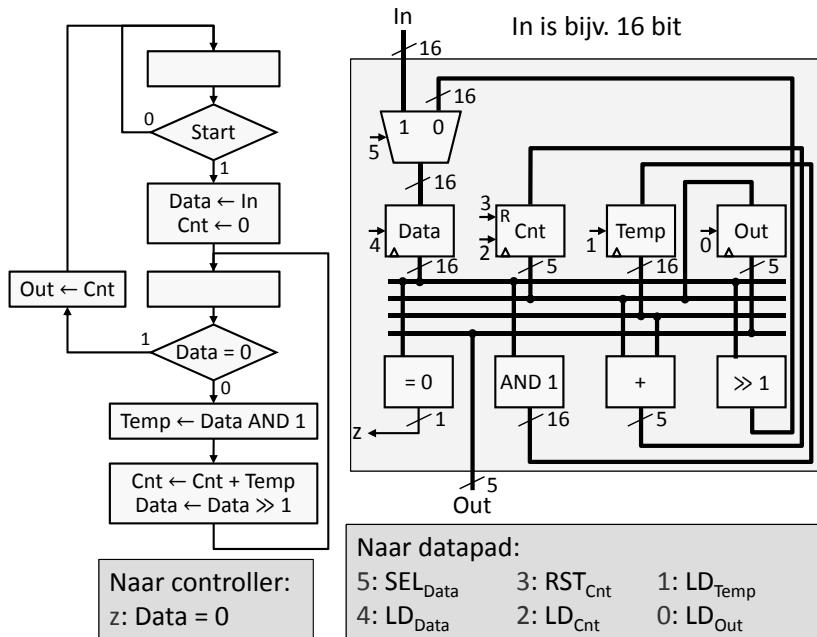
**Algoritme 3**

```
Data = In; Cnt = 0;
while Data ≠ 0 do
 Temp = Data AND 1;
 Cnt = Cnt + Temp;
 Data = Data ≫ 1;
end while;
Out = Cnt;
```

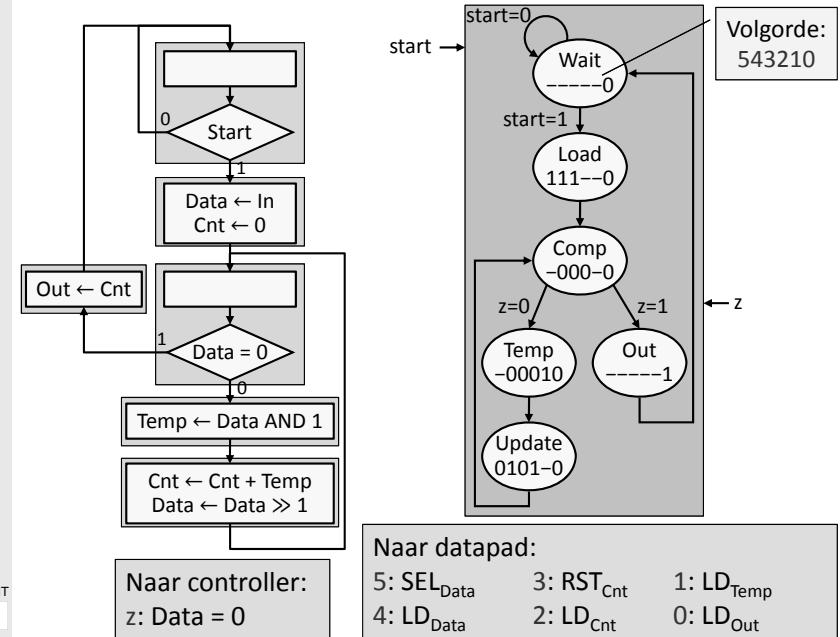
Constanten worden direct ingevuld!



# Synthese: datapad

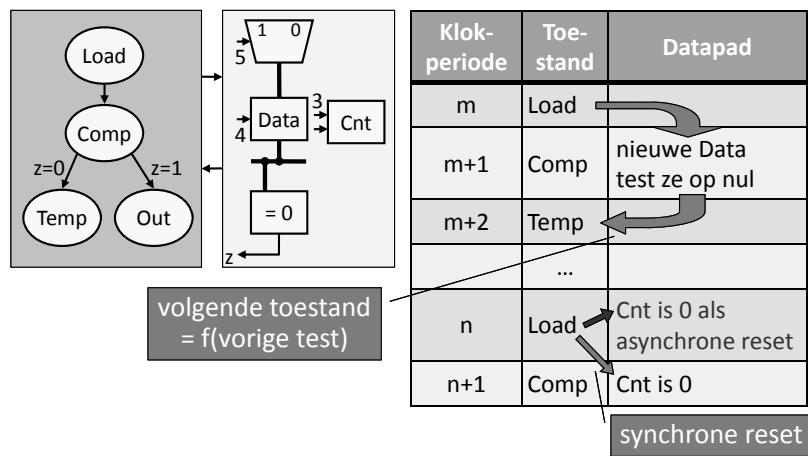


# Synthese: controller



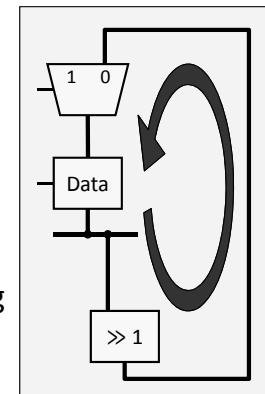
# Interactie datapad-controller

Altijd 1 klokperiode vertraging tussen de twee delen voor toestandsgestuurde synchrone systemen!



# Vertraging vermijden (als nodig)

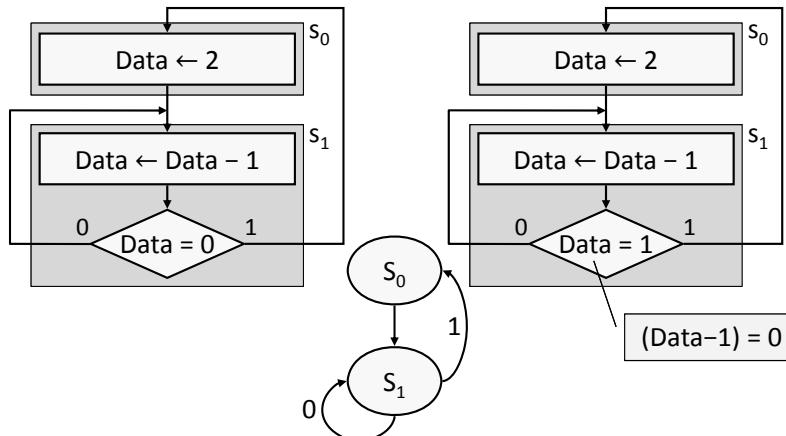
- Laat controller reageren in dezelfde klokperiode als zijn ingangen ⇒ inputgebaseerde controller
- Laat datapad reageren in dezelfde klokperiode als zijn controlesignalen  
⇒ asynchrone controlesignalen (LD, reset, ... )
- Gevaren:
  - mengen synchrone en asynchrone acties
  - doorrimpelen bij terugkoppeling



# Waarom altijd de oude data gebruiken in een ASM-schema?

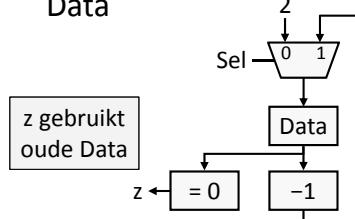
Bijv. genereer de sequentie 2 1 0 2 1 0 ...

- Twee mogelijke ASM-schema's en de bijbehorende controller:

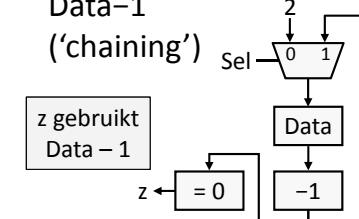


# Waarom altijd de oude data gebruiken in een ASM-schema?

- Test registeruitgang Data
- Test uitgang operator Data-1 ('chaining')

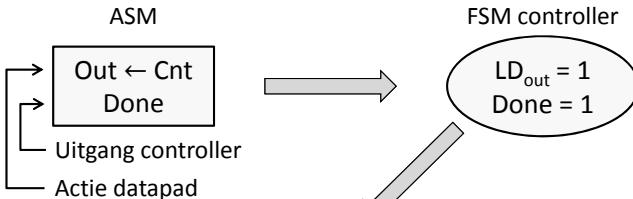


| Toestand          | Sel | Data | z |
|-------------------|-----|------|---|
| S <sub>1</sub>    | 1   | 1    | 0 |
| S <sub>1</sub>    | 1   | 0    | 1 |
| (S <sub>0</sub> ) | 0   | -1   | 0 |
| S <sub>1</sub>    | 1   | 2    | 0 |
| S <sub>1</sub>    | 1   | 1    | 1 |
| S <sub>0</sub>    | 0   | 0    | 1 |

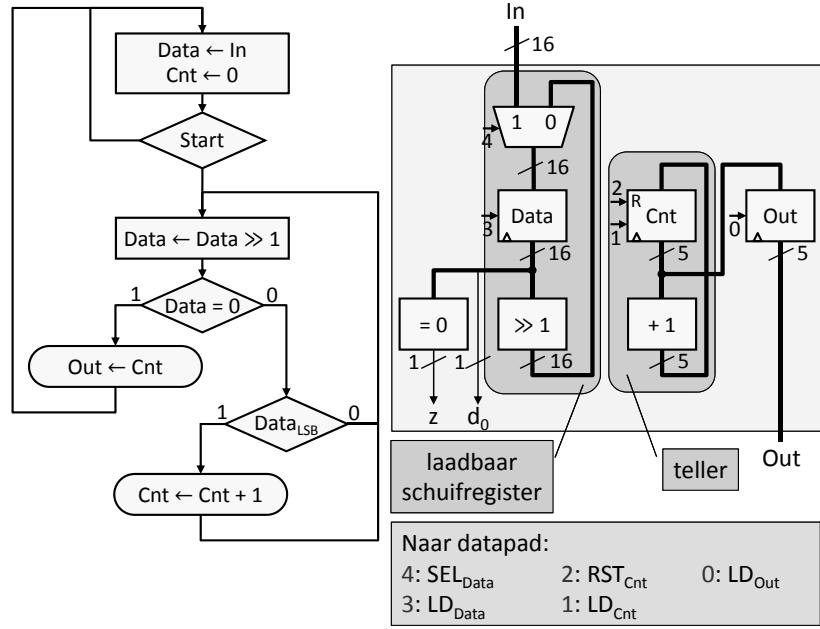


| Toestand       | Sel | Data | z |
|----------------|-----|------|---|
| S <sub>1</sub> | 1   | 1    | 1 |
| S <sub>0</sub> | 0   | 0    | 0 |
| S <sub>1</sub> | 1   | 2    | 0 |
| S <sub>1</sub> | 1   | 1    | 1 |
| S <sub>0</sub> | 0   | 0    | 0 |

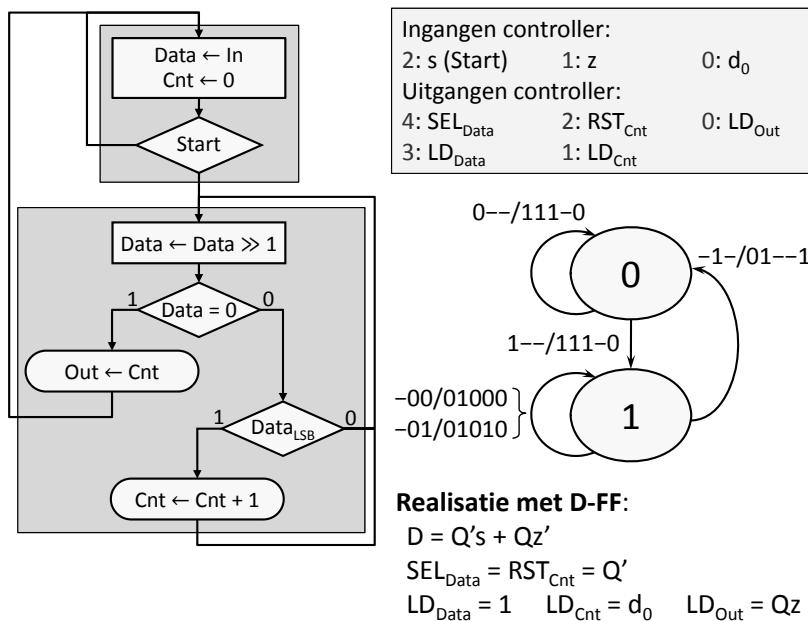
## Tijdsverband met controller-uitgangen



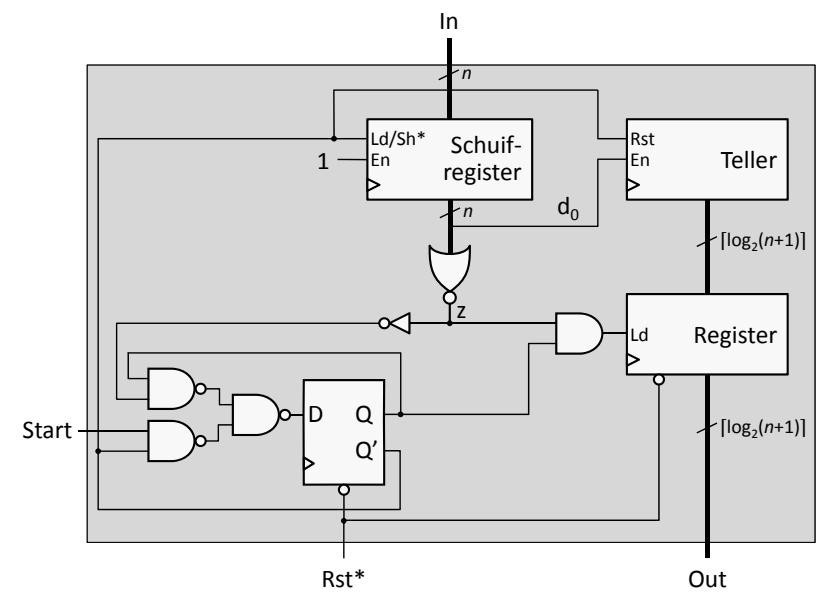
## Ander ASM-schema (algoritme 2): datapad



## Ander ASM-schema (algoritme 2): controller



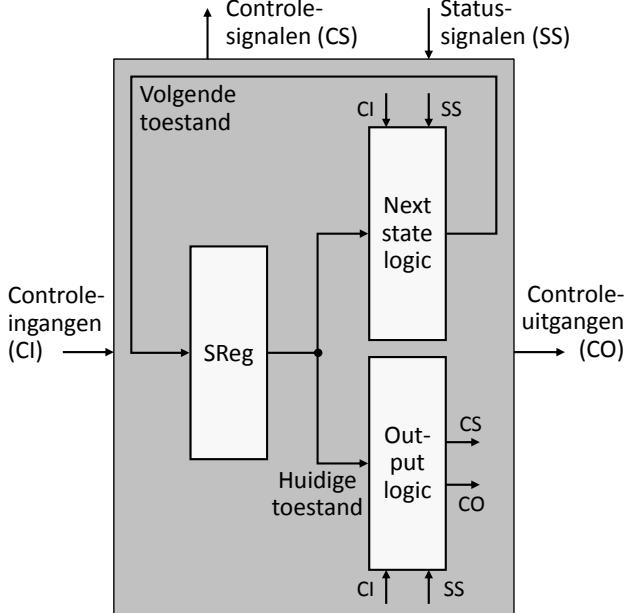
## Ander ASM-schema (algoritme 2): totale implementatie



# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basis principes
  - ➔ Sneller ontwerp van controller
  - Minimalisering datapad
  - Andere optimaliseringen
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

# Algemene vorm van een controller



# Ontwerp van de controller

- ≡ ontwerp van een FSM

➢ Dikwijs complex wegens het groot aantal toestanden

Hoe maken we snel een complexe controller?

Door patronen te herkennen in het ontwerp:

- natuurlijke volgorde van toestanden

- terugkerende subset van toestanden (subroutines)

- eenvoudige logica bij gebruik van een one-hot toestandsregister

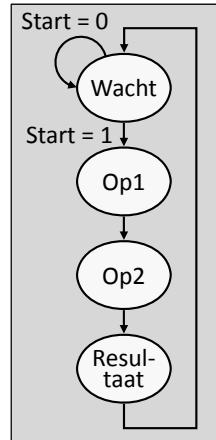
⇒ microprogrammeerbare controller

# Natuurlijke volgorde van toestanden

Dikwijs volgen de toestanden elkaar onvoorwaardelijk op, op een paar uitzonderingen na  
 ⇒ volgende toestand = huidige + 1

- Gebruik laadbare teller als toestandsregister

- ‘Next state logic’ eenvoudig: moet enkel conditionele volgende toestand genereren



Beschrijving

Geheugen

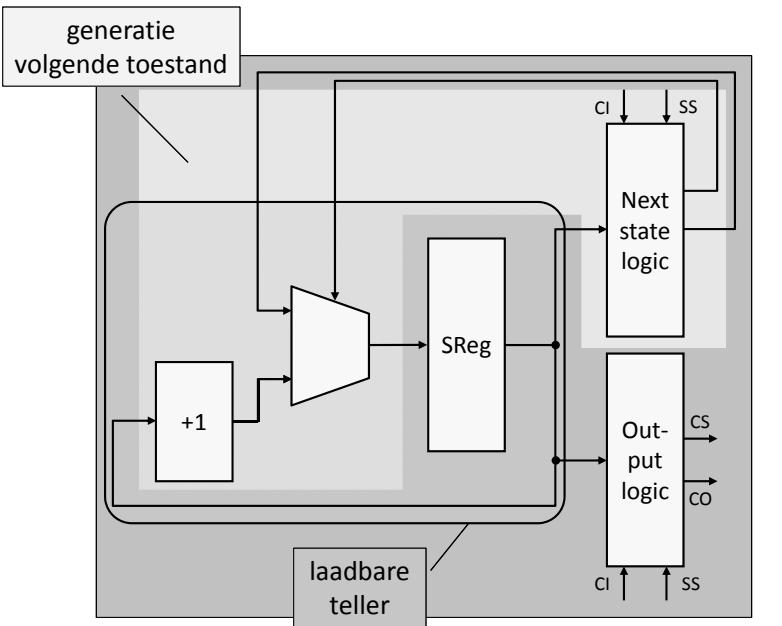
Synthese

- Basis
- ⇒ Controller
- Datapad
- Extra

Tijds gedrag

VHDL

## Natuurlijke volgorde van toestanden



Beschrijving

Geheugen

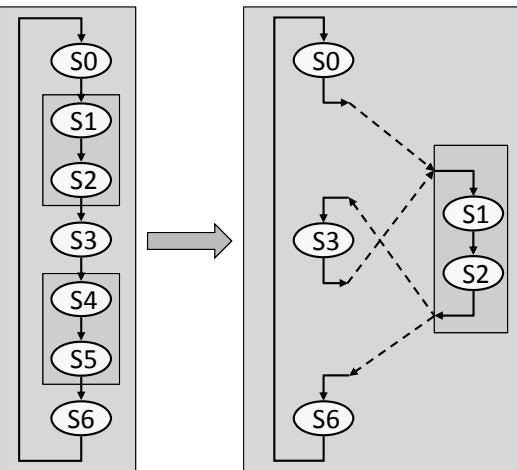
Synthese

- Basis
- ⇒ Controller
- Datapad
- Extra

Tijds gedrag  
VHDL

## Subroutine

Soms bevat een toestandsdiagram een deel dat verschillende keren herhaald wordt ⇒ subroutine



5 toestanden

De terugkeertoestand (= toestand na het subroutine-einde) is slechts gekend op het moment van uitvoering  
⇒ LIFO

Beschrijving

Geheugen

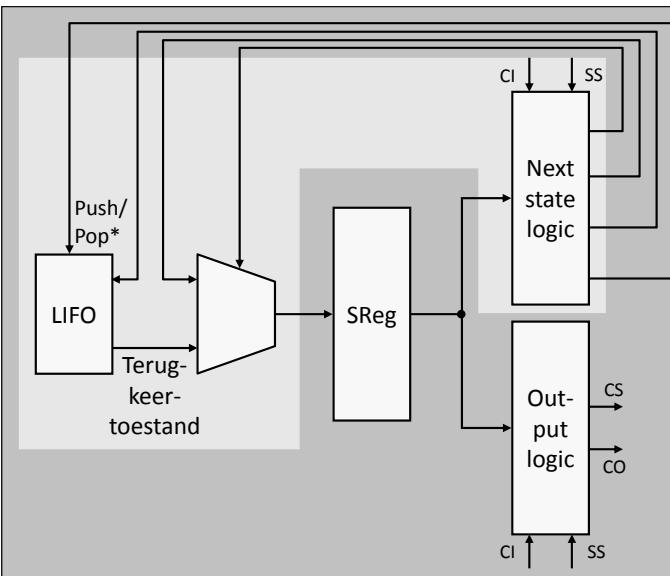
Synthese

- Basis
- ⇒ Controller
- Datapad
- Extra

Tijds gedrag

VHDL

## Subroutine



Beschrijving

Geheugen

Synthese

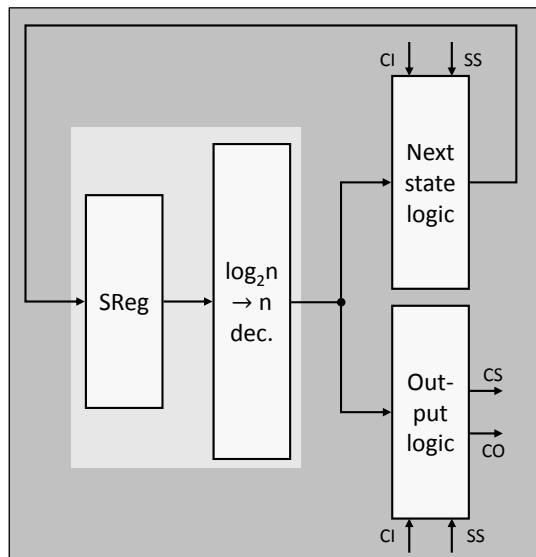
- Basis
- ⇒ Controller
- Datapad
- Extra

Tijds gedrag  
VHDL

## One-hot toestandsregister

Voordelen van

- one-hot: eenvoudig ontwerp en beperkte logica
- straightforward: klein # FF



## FSMD

## Beschrijving

## Geheugen

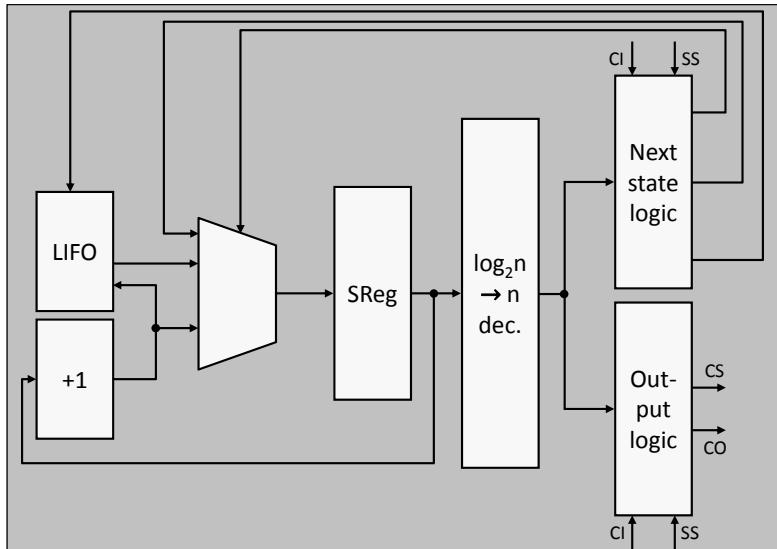
## Synthese

- Basis
- ⇒ Controller
- Datapad
- Extra

## Tijds gedrag

## VHDL

## Alle wijzigingen samen



Veronderstelling: terugkeertoestand = springtoestand + 1

## FSMD

## Beschrijving

## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
- variabelen
- bewerkingen
- verbindingen
- registers

## Extra

## Tijds gedrag

## VHDL

## Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
  - Basis principes
  - Sneller ontwerp van controller
  - Minimalisering datapad
  - Andere optimaliseringen
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

## FSMD

## Beschrijving

## Geheugen

## Synthese

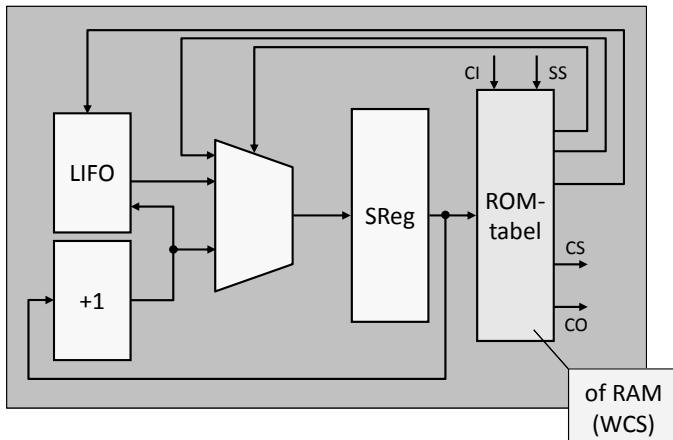
- Basis
- ⇒ Controller
- Datapad
- Extra

## Tijds gedrag

## VHDL

## Microprogramma-controller

= realisatie 'next state' en 'output' logica met een (ROM-)geheugen



## FSMD

## Beschrijving

## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
- variabelen
- bewerkingen
- verbindingen
- registers

## Extra

## Tijds gedrag

## VHDL

## Voorbeeld minimalisering

Benaderende berekening van de wortel van een kwadratische som (SRA: Square Root Approximation):

$$\sqrt{a^2 + b^2} \approx \max((0,875x + 0,5y), x)$$

$$\text{met } x = \max(|a|, |b|) \text{ en } y = \min(|a|, |b|)$$

- Gebruikt om het vermogen van QAM-transmissie te berekenen (o.a. voor CATV-transmissie):
  - $a$  is het reële deel van het signaal
  - $b$  is het imaginaire deel van het signaal

## FSMD

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller

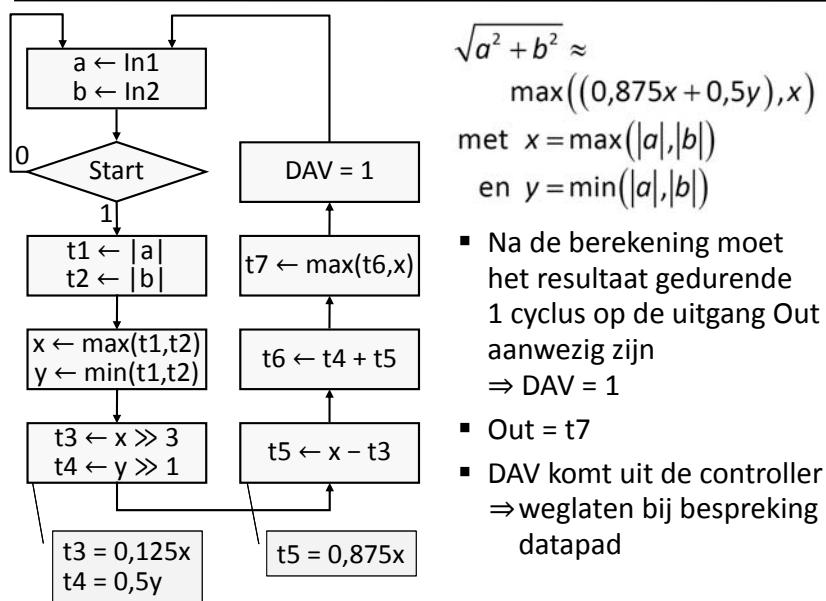
⇒ Datapad

- variabelen
- bewerkingen
- verbindingen
- registers

▪ Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

**SRA: ASM-schema**

## FSMD

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller

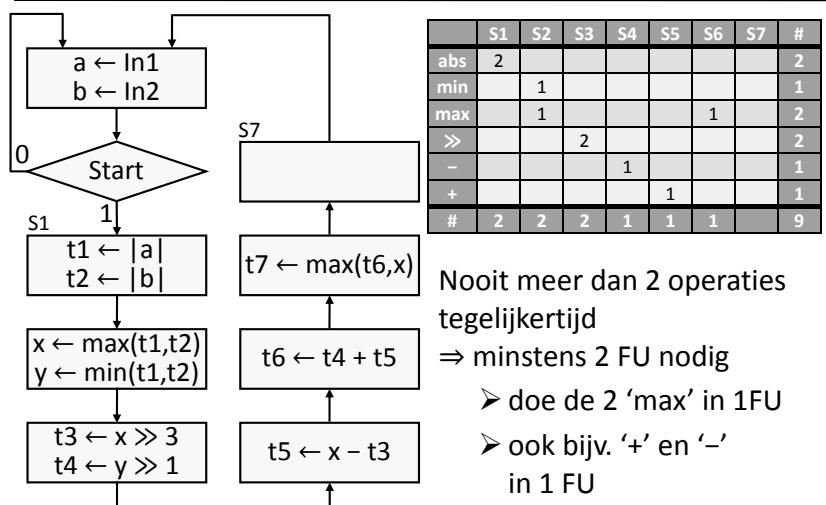
⇒ Datapad

- variabelen
- bewerkingen
- verbindingen
- registers

▪ Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

**SRA: gebruik FU**

Hoe meer verschillende bewerkingen in 1 FU,  
hoe complexer de FU!

## FSMD

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller

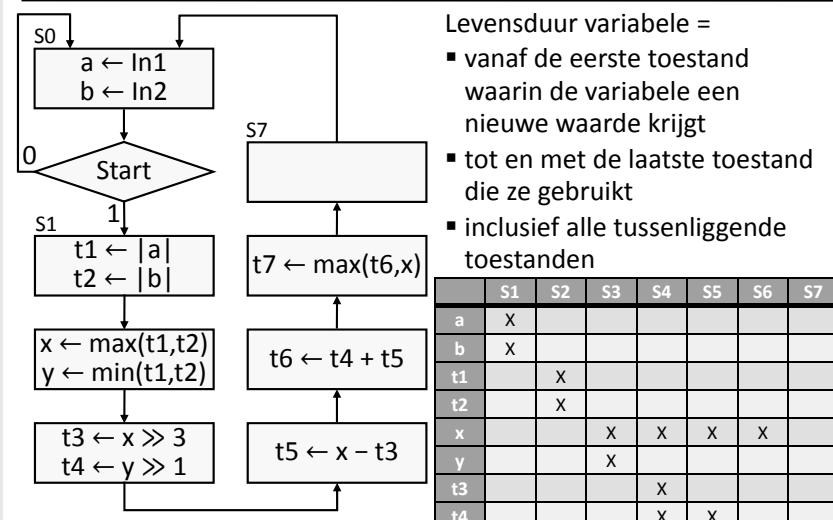
⇒ Datapad

- variabelen
- bewerkingen
- verbindingen
- registers

▪ Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

**SRA: gebruik variabelen**

Nooit meer dan 3 variabelen levend tegelijkertijd  
⇒ slechts 3 registers nodig

## FSMD

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller

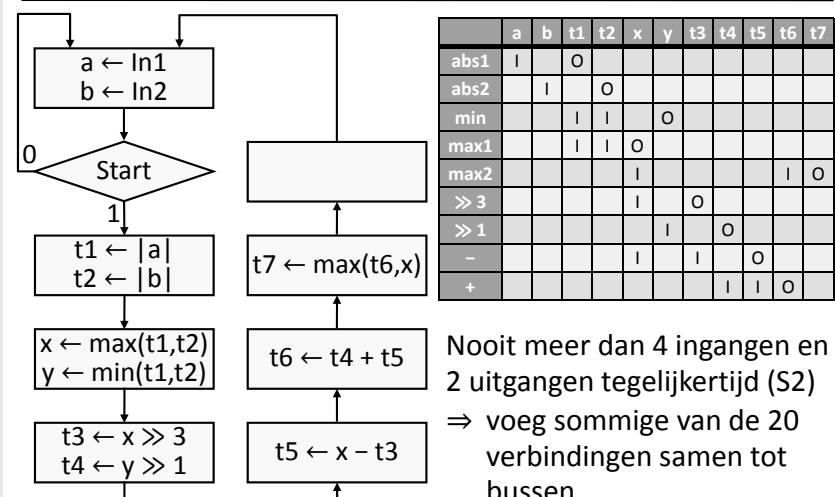
⇒ Datapad

- variabelen
- bewerkingen
- verbindingen
- registers

▪ Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

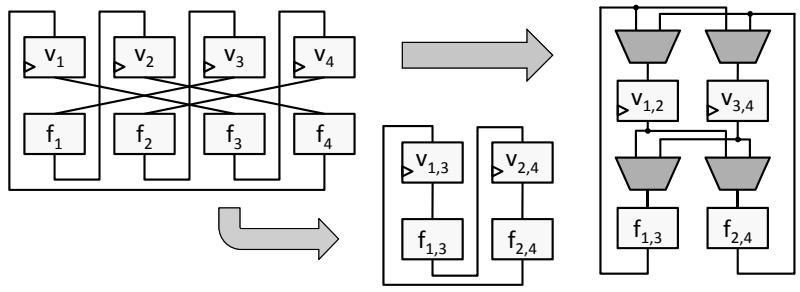
**SRA: verbindingen**

# Globaal optimum

*Optimaliseringen zijn niet onafhankelijk van elkaar!*

Bijv. samenvoegen registers en FU:

- Dit introduceert MUX aan de ingangen van de registers en de FU en het aantal hangt af van welke registers en FU samengevoegd zijn.



# Volgorde optimaliseringen

Wat brengt het meest op?

- Meestal het samenvoegen van registers
  - Er zijn meer variabelen dan FU
  - Het samenvoegen van FU resulteert meestal in een duurdere FU dan ieder van de FU 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
  - 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
- Zelden het samenvoegen van verbindingen
  - Verbinding heeft minder impact op kostprijs

# Globaal optimum

Patstelling ('deadlock'): om een deelprobleem optimaal te lossen moeten de andere deelproblemen al optimaal opgelost zijn

Praktische benadering:

- Optimaliseer eerst wat het meest oplevert en gebruik hiervoor een schatting van de andere optimaliseringen
- Optimaliseer dan de andere aspecten
- Itereer tot een bevredigend resultaat bereikt is

# Kostprijsberekening

Algemene opmerking: alles wat voor meer dan 1 bit gebruikt wordt, wordt hier niet meegeteld

- Kostprijs van 3-state buffer
  - 10 transistoren
  - gratis 3-state buffer aan een lange lijn in FPGA
- Kostprijs van 1-bit MUX

|                 | 2-naar-1 | 3-naar-1 | 4-naar-1 | 5-naar-1 |
|-----------------|----------|----------|----------|----------|
| transistoren    | 12       | 18       | 24       | 30       |
| Logische Cellen | 1        | 2        | 2        | 3        |

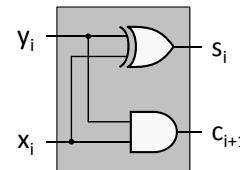
- Kostprijs van een 1-bit register met 'enable' en asynchrone preset of clear
  - 44 transistoren
  - 1 LC, maar gratis na een MUX

# Kostprijsberekening

## □ Kostprijs van een optelling

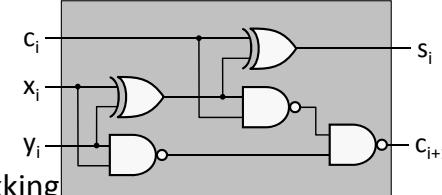
- Halve opteller (HA)

- 12 + 6 = 18 transistoren
- 2 LC (1 als carry logica bruikbaar)



- Opteller (FA)

- 36 transistoren
- 2 LC (1 als carry logica bruikbaar)

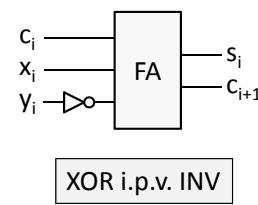


## □ Kostprijs van een aftrekking

- 38 transistoren
- 2 LC

## □ Kostprijs optelling/aftrekking

- 48 transistoren
- 2 LC

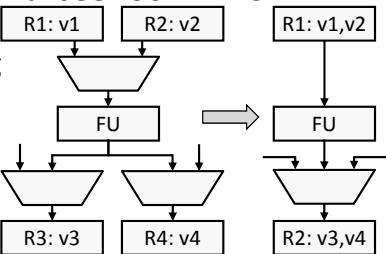


# Variabelen samenvoegen

## □ Groepeer variabelen met niet-overlappende levensduur in 1 register

## □ Bijkomend criterium: minimaliseer ook # MUX

- Voeg twee variabelen samen die dezelfde ingang van een FU gebruiken
- Voeg twee variabelen samen die dezelfde uitgang van eenzelfde FU zijn



## □ Hiervoor schatting samengevoegde FU nodig;

voor SRA:

- Combineer de twee max-bewerkingen in één
- Combineer de optelling en de aftrekking in 1 FU

|     | S1 | S2 | S3 | S4 | S5 | S6 | S7 | # |
|-----|----|----|----|----|----|----|----|---|
| abs | 2  |    |    |    |    |    |    | 2 |
| min |    | 1  |    |    |    |    |    | 1 |
| max |    | 1  |    |    |    | 1  |    | 2 |
| >>  |    |    | 2  |    |    |    |    | 2 |
| -   |    |    |    | 1  |    |    |    | 1 |
| +   |    |    |    |    | 1  |    |    | 1 |

# Niet-programmeerbare processoren

## □ Beschrijving van een algoritme

## □ Extra geheugenbouwblokken

## ➔ Synthese: naar een minimale hardware

- Basisprincipes

- Sneller ontwerp van controller

## ➔ Minimalisering datapad

- Variabelen samenvoegen ('register sharing')
- Bewerkingen samenvoegen ('functional-unit sharing')
- Verbindingen samenvoegen ('bus sharing')
- Registers samenvoegen tot een registerbank ('register port sharing')

- Andere optimaliseringen

## □ Aandachtspunten qua tijds gedrag

## □ VHDL voor synthese en simulatie

# Minimalisering met compatibiliteitsgraaf

Compatibiliteitsgraaf bestaat uit drie delen:

## □ Knopen ('nodes')

- Komen hier overeen met variabelen

## □ Incompatibiliteitsranden tussen knopen die onverenigbaar zijn

- Tussen twee variabelen met overlappende levensduur: deze zijn niet samen te voegen

## □ Prioriteitsranden tussen knopen die goed verenigbaar zijn, waarbij het gewicht de wenselijkheid van samenvoegen aangeeft

- Tussen twee variabelen die aan dezelfde ingang van dezelfde FU gebruikt worden of die dezelfde uitgang van dezelfde FU gebruiken

- Gewicht = # keren dat de twee variabelen aan de vorige voorwaarde voldoen

## Beschrijving

## Geheugen

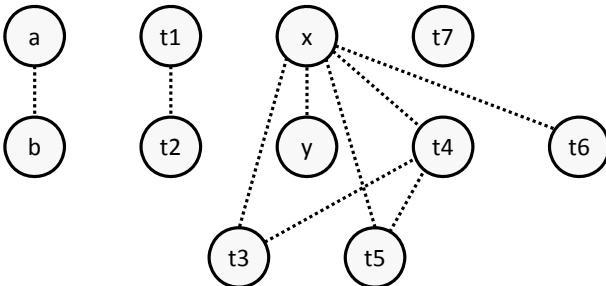
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

# SRA: compatibiliteitsgraaf (1)



|    | 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  |    |

Knopen zijn variabelen

Incompatibiliteitsranden:  
variabelen met  
overlappende levensduur

## Beschrijving

## Geheugen

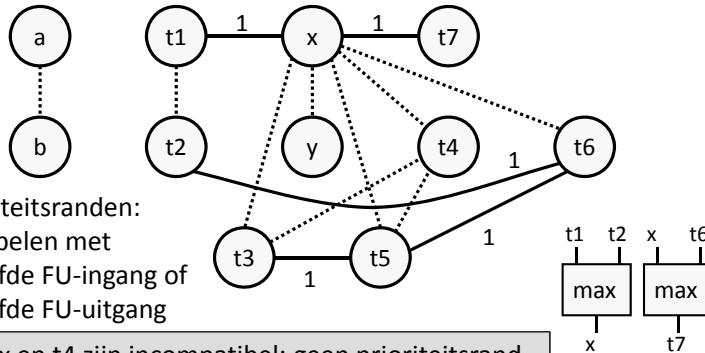
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

# SRA: compatibiliteitsgraaf (2)



| In 1    | a | b | t1 | x | y | t4 | In 2    | t2 | t3 | t5 | t6 | Uit     | t1 | t2 | x | y | t3 | t4 | t5 | t6 | t7 |
|---------|---|---|----|---|---|----|---------|----|----|----|----|---------|----|----|---|---|----|----|----|----|----|
| abs1    | x |   |    |   |   |    | abs1    |    |    |    |    | abs1    | x  |    |   |   |    |    |    |    |    |
| abs2    |   | x |    |   |   |    | abs2    |    |    |    |    | abs2    |    | x  |   |   |    |    |    |    |    |
| min     |   |   | x  |   |   |    | min     | x  |    |    |    | min     |    |    | x |   |    |    |    |    |    |
| max     |   |   | x  | x |   |    | max     | x  |    | x  |    | max     |    | x  |   |   |    |    | x  |    |    |
| $\gg 3$ |   |   |    | x |   |    | $\gg 3$ |    |    |    |    | $\gg 3$ |    |    |   |   |    |    | x  |    |    |
| $\gg 1$ |   |   |    |   | x |    | $\gg 1$ |    |    |    |    | $\gg 1$ |    |    |   |   |    |    | x  |    |    |
| $-/+$   |   |   |    | x | x |    | $-/+$   |    | x  | x  |    | $-/+$   |    | x  | x |   |    |    | x  | x  |    |

## Beschrijving

## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

# Minimalisering met compatibiliteitsgraaf

## 'Max-cut graph partitioning'

= verdeel de graaf in het kleinst aantal groepen van verenigbare knopen zodat het totale gewicht van alle groepen samen maximaal is

- Totale gewicht van een groep
  - = de som van de gewichten van alle prioriteitsranden in de groep

We doen dit visueel als voorbeeld

- Voor implementaties,  
zie cursussen over optimalisering

## Beschrijving

## Geheugen

## Synthese

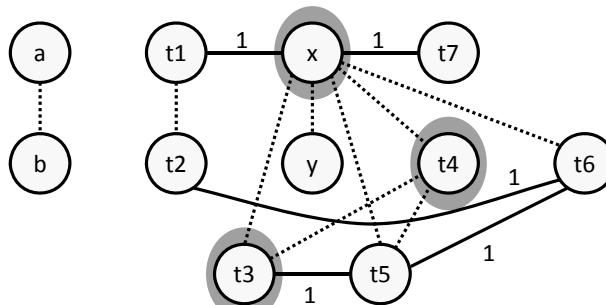
- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

# SRA: max-cut graph partitioning

- x, t3 en t4 zijn onderling onverenigbaar  
⇒ elk in een apart register



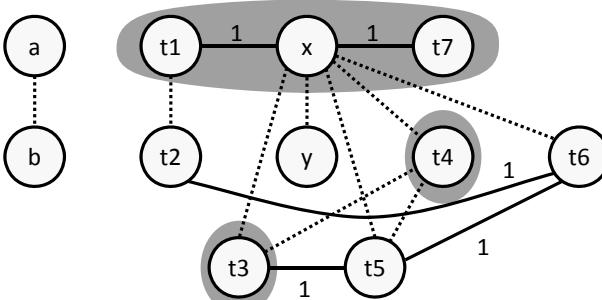
**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

**Tijds gedrag****VHDL**

# SRA: max-cut graph partitioning

- t1 & x en t7 & x zijn verenigbaar en verbonden door een prioriteitsrand met het hoogste gewicht (nl. 1) ⇒ alle drie in hetzelfde register

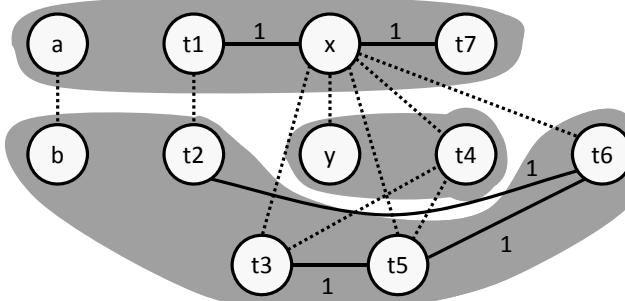
**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

**Tijds gedrag****VHDL**

# SRA: max-cut graph partitioning

- Geen prioriteitsranden meer ⇒ overblijvende variabelen kunnen bij gelijk welk register gevoegd worden waarmee ze verenigbaar zijn



R1: a, t1, x, t7  
 R2: b, t2, t3, t5, t6  
 R3: y, t4

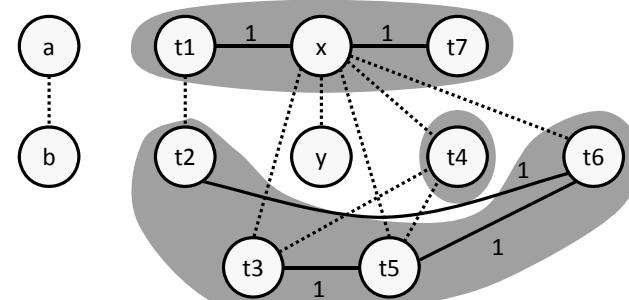
**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

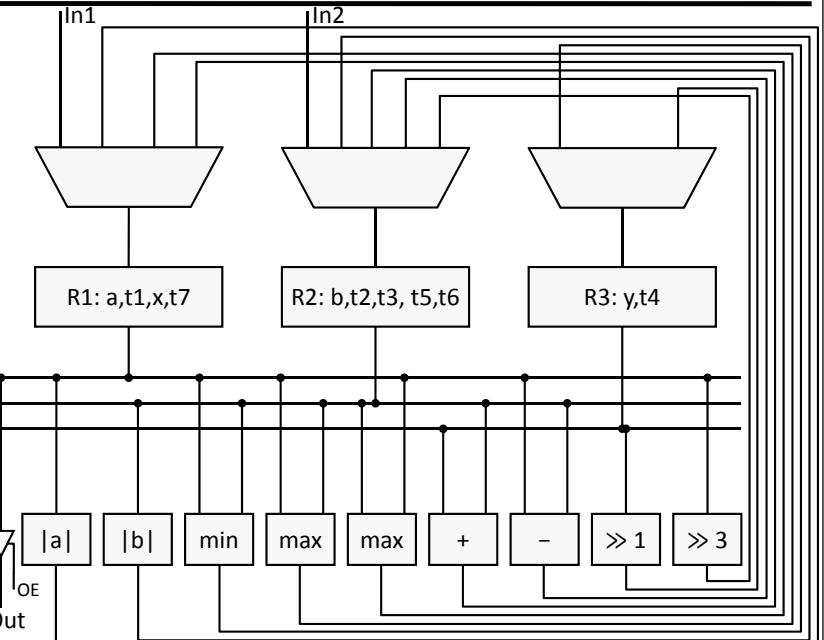
**Tijds gedrag****VHDL**

# SRA: max-cut graph partitioning

- t3, t5, t6 en t2 zijn verenigbaar en verbonden door prioriteitsranden ⇒ in hetzelfde register



## Implementatie SRA na samenvoegen registers



Beschrijving  
Geheugen  
Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Kostprijsberekening per bit

- Originele FSMD (geen optimalisering):
  - 11 registers
    - $11 \text{ reg} \times 44 \text{ tor/bit} = 484 \text{ tors/bit}$
    - $11 \text{ reg} \times 1 \text{ LC/bit} = 11 \text{ LC/bit}$
- Na samenvoegen registers:
  - 1 register met een 4-naar-1 MUX
    - $24 \text{ tor/MUXbit} + 44 \text{ tor/REGbit} = 68 \text{ tors}$
    - $2 \text{ LC/MUXREGbit}$
  - 1 register met een 5-naar-1 MUX
    - $30 \text{ tor/MUXbit} + 44 \text{ tor/REGbit} = 74 \text{ tors}$
    - $3 \text{ LC/MUXREGbit}$
  - 1 register met een 2-naar-1 MUX
    - $12 \text{ tor/MUXbit} + 44 \text{ tor/REGbit} = 56 \text{ tors}$
    - $1 \text{ LC/MUXREGbit}$

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basisprincipes
  - Sneller ontwerp van controller
  - ➔ Minimalisering datapad
    - Variabelen samenvoegen
      - ➔ Bewerkingen samenvoegen ('functional-unit sharing')
      - Verbindingen samenvoegen
      - Registers samenvoegen tot een registerbank
    - Andere optimaliseringen
  - Aandachtspunten qua tijds gedrag
  - VHDL voor synthese en simulatie

Beschrijving  
Geheugen  
Synthese

- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Vergelijking kostprijs per bit

| Samenvoegen | transistoren |    |     | LC  |    |     | Verbindingen |
|-------------|--------------|----|-----|-----|----|-----|--------------|
|             | Reg          | FU | Tot | Reg | FU | Tot |              |
| Origineel   | 484          |    |     | 11  |    |     | 20           |
| Register    | 198          |    |     | 6   |    |     | 12           |
| FU          |              |    |     |     |    |     |              |
| Bus         |              |    |     |     |    |     |              |
| Reg.bank    |              |    |     |     |    |     |              |

Optimaliseringen beïnvloeden elkaar:

samenvoegen registers verandert het aantal verbindingen

- We hadden een schatting van de reductie in verbindingen kunnen gebruiken bij het samenvoegen van registers

Beschrijving  
Geheugen  
Synthese

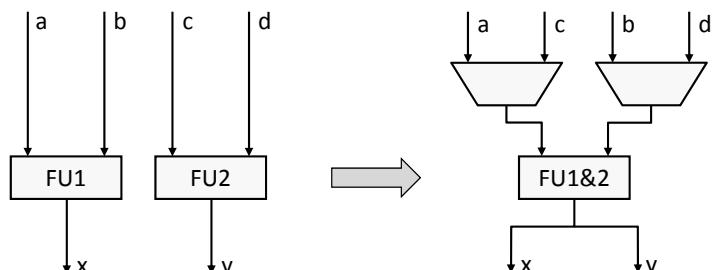
- Basis
- Controller
- ⇒ Datapad
  - ⇒ variabelen
  - bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Samenvoegen bewerkingen

- = vervang FU die niet tegelijkertijd gebruikt worden door 1 FU met de gecombineerde functionaliteit en MUX aan zijn ingangen
  - Kunnen identieke FU (bijv.  $2 \times \text{MAX}$ ) of gelijkaardige (bijv. ADD & SUBTRACT) zijn
  - Enkel wanneer nieuwe FU&MUX goedkoper is!
  - Samenvoegen registers beïnvloedt MUX-kost



Beschrijving

Geheugen

Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
    - verbindingen
    - registers
- Extra

Tijds gedrag

VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# Samenvoegen bewerkingen

## Compatibiliteitsgraaf voor FU

- Knopen
  - Komen overeen met bewerkingen
- Incompatibiliteitsranden
  - Tussen twee bewerkingen die in eenzelfde toestand gebruikt worden
    - ⇒ niet samen te voegen
- Prioriteitsranden
  - Tussen twee of meer bewerkingen die door eenzelfde (nieuwe) FU kunnen uitgevoerd worden
  - Gewicht = winst in kostprijs door het samenvoegen

Beschrijving

Geheugen

Synthese

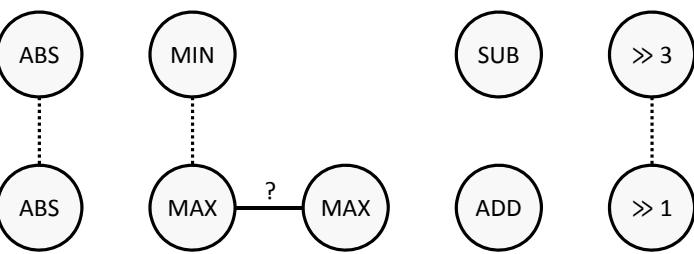
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
    - verbindingen
    - registers
- Extra

Tijds gedrag

VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# SRA: compatibiliteitsgraaf FU



Knopen zijn bewerkingen

Incompatibiliteitsrand: twee bewerkingen in dezelfde toestand

Prioriteitsrand: gewicht geeft winst door samenvoegen

|     | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|-----|----|----|----|----|----|----|----|
| abs | 2  |    |    |    |    |    |    |
| min |    | 1  |    |    |    |    |    |
| max |    | 1  |    |    |    | 1  |    |
| >>  |    |    | 2  |    |    |    |    |
| -   |    |    |    | 1  |    |    |    |
| +   |    |    |    |    | 1  |    |    |
| #   | 2  | 2  | 2  | 1  | 1  | 1  |    |

Beschrijving

Geheugen

Synthese

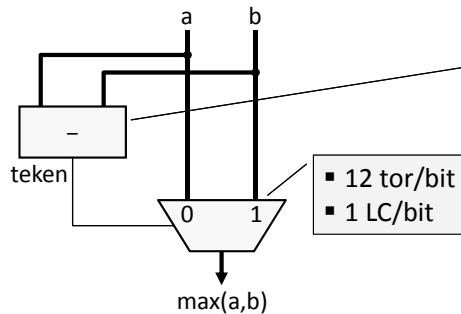
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
    - verbindingen
    - registers
- Extra

Tijds gedrag

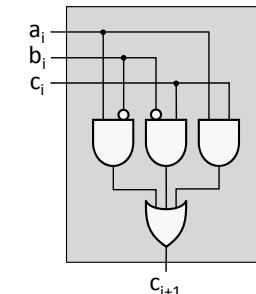
VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

## Kostprijs MAX/MIN-bewerking



Alleen carry-logica, behalve voor MSB waar som nodig is:  
 ■ 20 tors/bit  
 ■ 1 LC/bit



Kostprijs per bit:  
 ■ 32 tors  
 ■ 2 LC

Dezelfde kostprijs voor  $\min(a, b)$ : verwissel ingangen MUX

Beschrijving

Geheugen

Synthese

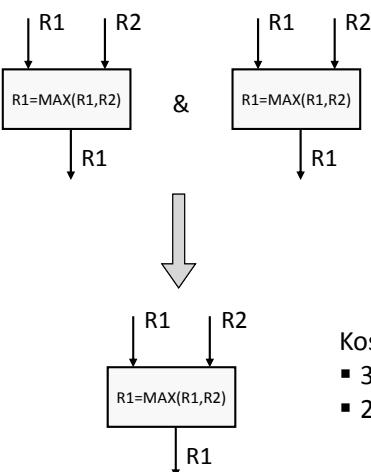
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
    - verbindingen
    - registers
- Extra

Tijds gedrag

VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

## Winst van één MAX & MAX FU



Kostprijs:  
 ■ 64 tors  
 ■ 4 LC

Kostprijs:  
 ■ 32 tors  
 ■ 2 LC

Winst:  
 ■ 32 tors  
 ■ 2 LC

Bemerkt dat dit hier geen MUX nodig is omdat de respectievelijke operanden en het resultaat uit eenzelfde register komen!

## Beschrijving

## Geheugen

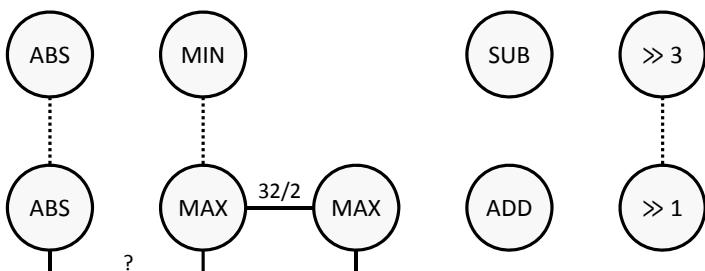
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



## Beschrijving

## Geheugen

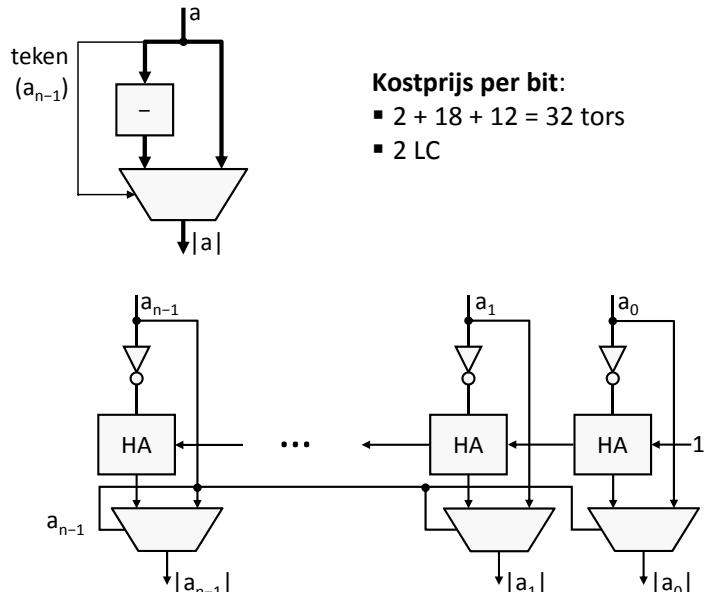
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## Kostprijs ABS-bewerking



## Beschrijving

## Geheugen

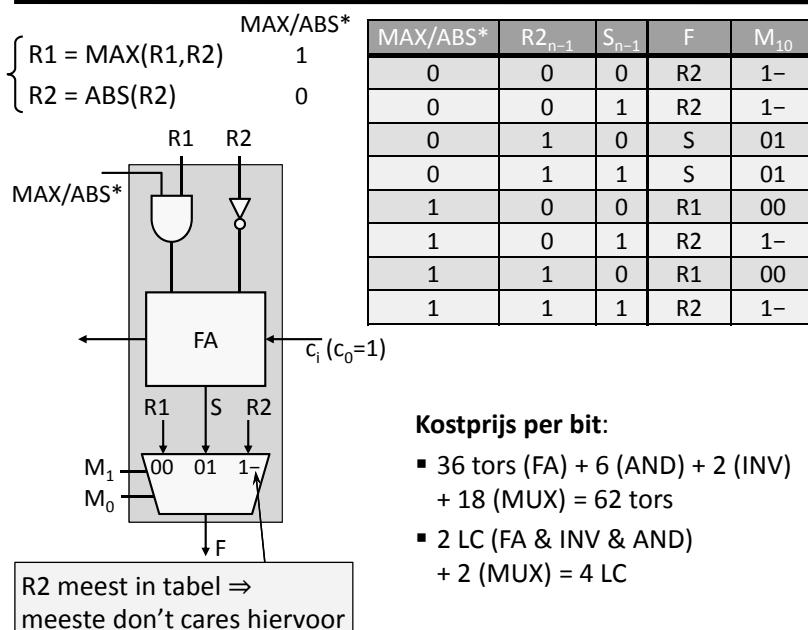
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## Realisatie ABS &amp; MAX FU



## Beschrijving

## Geheugen

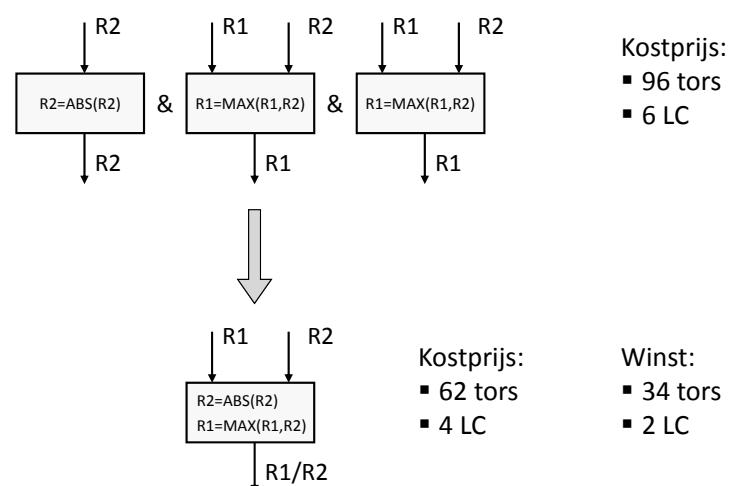
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## Winst van één ABS &amp; MAX &amp; MAX FU



Beschrijving

Geheugen

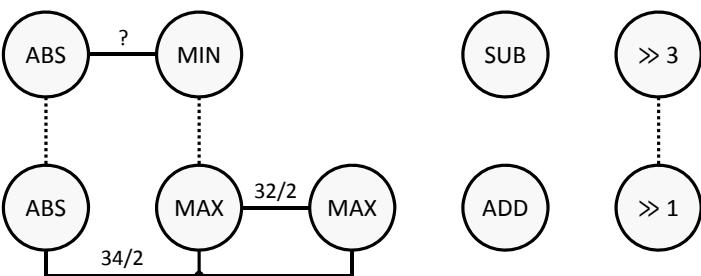
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

# SRA: compatibiliteitsgraaf FU



Beschrijving

Geheugen

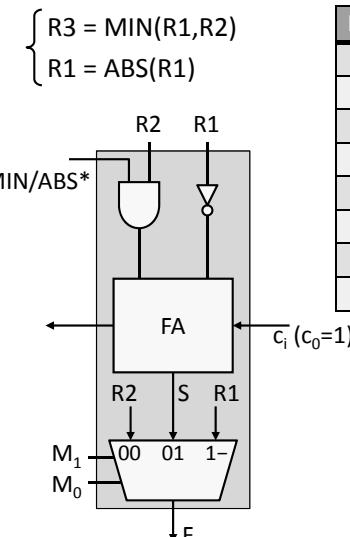
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

# Realisatie ABS & MIN FU



| MIN/ABS* | R1 <sub>n-1</sub> | S <sub>n-1</sub> | F  | M <sub>10</sub> |
|----------|-------------------|------------------|----|-----------------|
| 0        | 0                 | 0                | R1 | 1-              |
| 0        | 0                 | 1                | R1 | 1-              |
| 0        | 1                 | 0                | S  | 01              |
| 0        | 1                 | 1                | S  | 01              |
| 1        | 0                 | 0                | R1 | 1-              |
| 1        | 0                 | 1                | R2 | 00              |
| 1        | 1                 | 0                | R1 | 1-              |
| 1        | 1                 | 1                | R2 | 00              |

## Kostprijs per bit:

- 36 tors (FA) + 6 (AND) + 2 (INV) + 18 (MUX) = 62 tors
- 2 LC (FA & INV & AND) + 2 (MUX) = 4 LC

Beschrijving

Geheugen

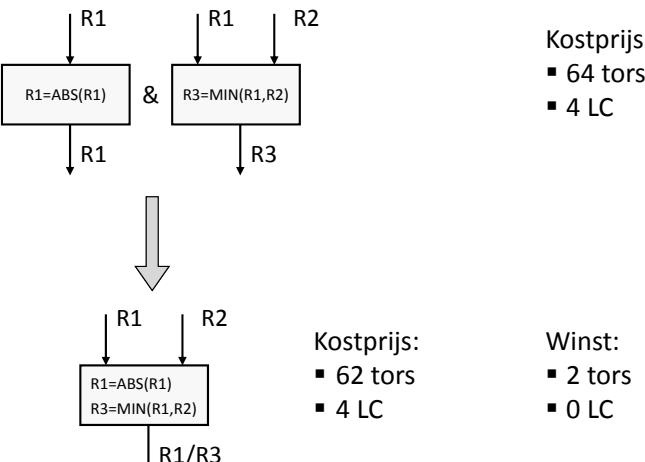
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

# Winst van één ABS & MIN FU



Klaarblijkelijk niet altijd winstgevend om ABS en MIN samen te voegen!

Beschrijving

Geheugen

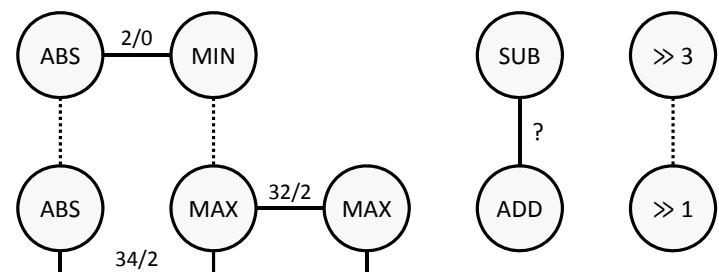
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

# SRA: compatibiliteitsgraaf FU



## FSMD

Beschrijving

Geheugen

Synthese

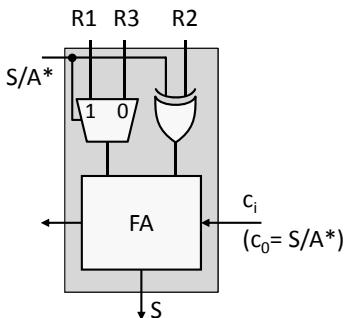
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

## Realisatie ADD &amp; SUB FU

$$\begin{cases} R2 = ADD(R3, R2) \\ R2 = SUB(R1, R2) \end{cases}$$



## Kostprijs per bit:

- 48 tors (FAS) + 12 (MUX) = 60 tors
- 2 LC (FAS) + 1 (MUX) = 3 LC

## FSMD

Beschrijving

Geheugen

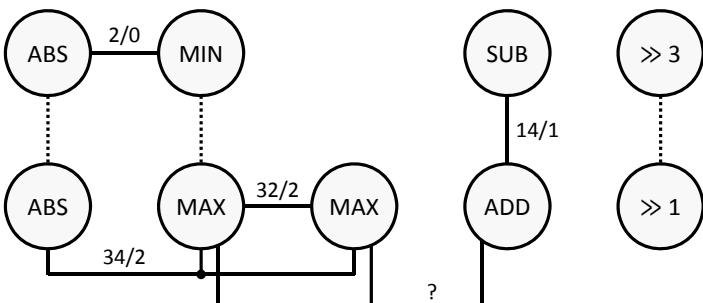
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

Beschrijving

Geheugen

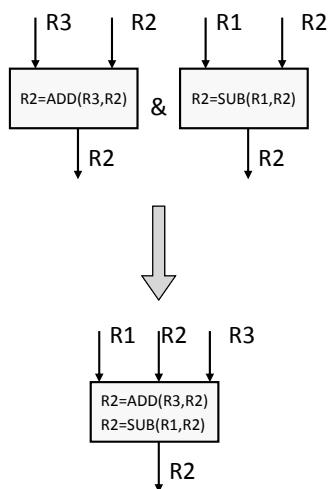
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

## Winst van één ADD &amp; SUB FU



Kostprijs:  

- 74 tors
- 4 LC

Kostprijs:  

- 60 tors
- 3 LC

Winst:  

- 14 tors
- 1 LC

## FSMD

Beschrijving

Geheugen

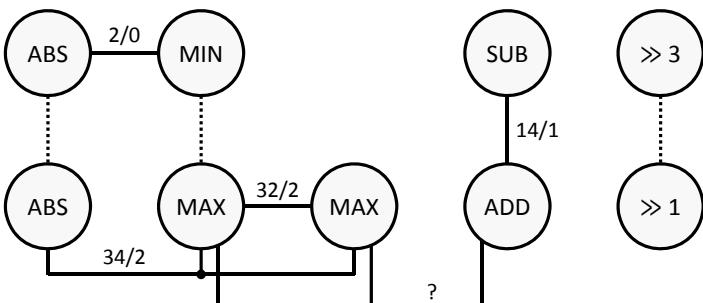
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

Beschrijving

Geheugen

Synthese

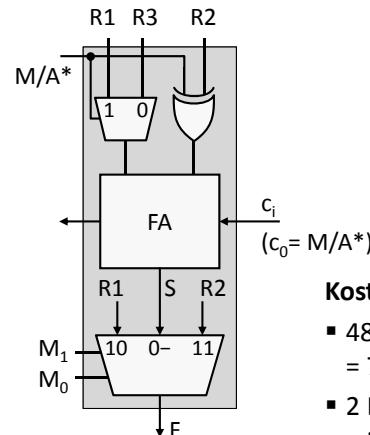
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

Tijds gedrag

VHDL

## Realisatie ADD &amp; MAX FU

$$\begin{cases} R2 = ADD(R3, R2) \\ R1 = MAX(R1, R2) \end{cases}$$



| M/A* | S_{n-1} | F  | M_{10} |
|------|---------|----|--------|
| 0    | 0       | S  | 0-     |
| 0    | 1       | S  | 0-     |
| 1    | 0       | R1 | 10     |
| 1    | 1       | R2 | 11     |

$$\begin{aligned} M_1 &= MAX/ADD^* \\ M_0 &= S_{n-1} \end{aligned}$$

## Kostprijs per bit:

- 48 tors (FAS) + 12 (MUX) + 18 (MUX) = 78 tors
- 2 LC (FAS) + 1 (MUX) + 2 (MUX) = 5 LC

## FSMD

## Beschrijving

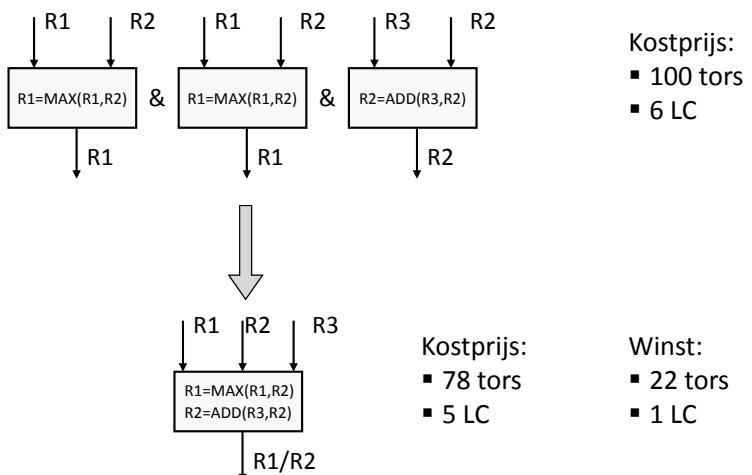
## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

Winst van één  
MAX & MAX & ADD FU

## FSMD

## Beschrijving

## Geheugen

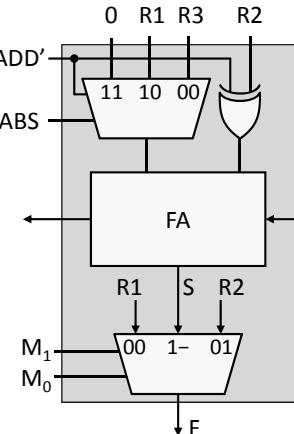
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

## Realisatie ABS &amp; MAX &amp; ADD FU



$$\begin{cases} R2 = \text{ABS}(R2) & \text{ABS} \\ R2 = \text{ADD}(R3, R2) & \text{ADD} \\ R1 = \text{MAX}(R1, R2) & \end{cases}$$

$$M_1 = \text{ADD} + \text{ABS} \cdot S_{n-1}'$$

$$M_0 = S_{n-1}$$

## Kostprijs per bit:

- 48 tors (FAS) + 12 (MUX)  
 + 18 (MUX) = 78 tors
- 2 LC (FAS) + 1 (MUX)  
 + 2 (MUX) = 5 LC

## FSMD

## Beschrijving

## Geheugen

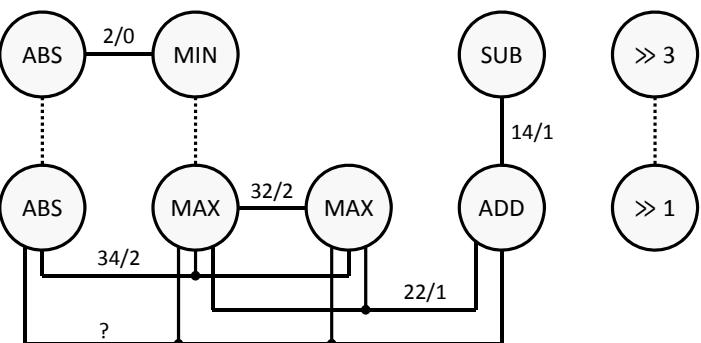
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

## Beschrijving

## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

## Realisatie ABS &amp; MAX &amp; ADD FU

$$\begin{cases} R2 = \text{ABS}(R2) & \text{ABS} \\ R2 = \text{ADD}(R3, R2) & \text{ADD} \\ R1 = \text{MAX}(R1, R2) & \end{cases}$$

$$M_1 = \text{ADD} + \text{ABS} \cdot S_{n-1}'$$

$$M_0 = S_{n-1}$$

## Kostprijs per bit:

- 48 tors (FAS) + 12 (MUX)  
 + 18 (MUX) = 78 tors
- 2 LC (FAS) + 1 (MUX)  
 + 2 (MUX) = 5 LC

## FSMD

## Beschrijving

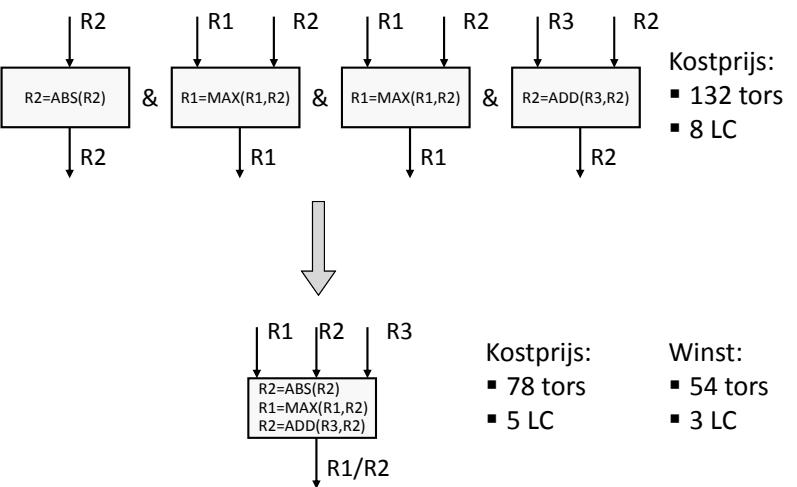
## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijdsgedrag

## VHDL

Winst van één  
ABS & MAX & MAX & ADD FU

## FSMD

## Beschrijving

## Geheugen

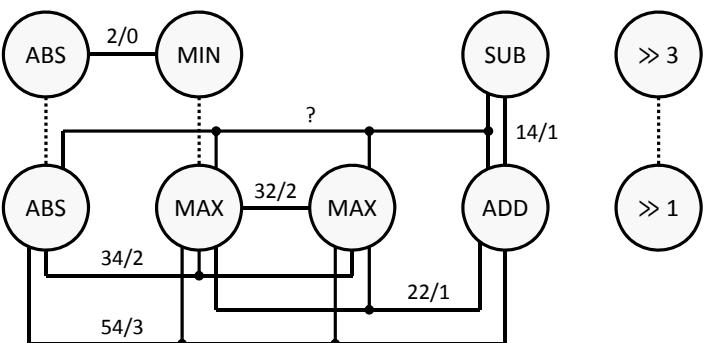
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

## Beschrijving

## Geheugen

## Synthese

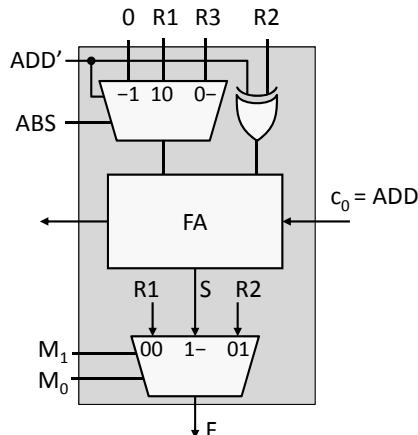
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

Realisatie  
ABS & MAX & ADD & SUB FU

$$\left\{ \begin{array}{l} R2 = ABS(R2) \\ R2 = ADD(R3, R2) \\ R2 = SUB(R1, R2) \\ R1 = MAX(R1, R2) \end{array} \right.$$



$$\begin{aligned} M_1 &= ADD + SUB + ABS \cdot S_{n-1}' \\ M_0 &= S_{n-1} \end{aligned}$$

## Kostprijs per bit:

- 48 tors (FAS) + 12 (MUX)  
+ 18 (MUX) = 78 tors
- 2 LC (FAS) + 1 (MUX)  
+ 2 (MUX) = 5 LC

## FSMD

## Beschrijving

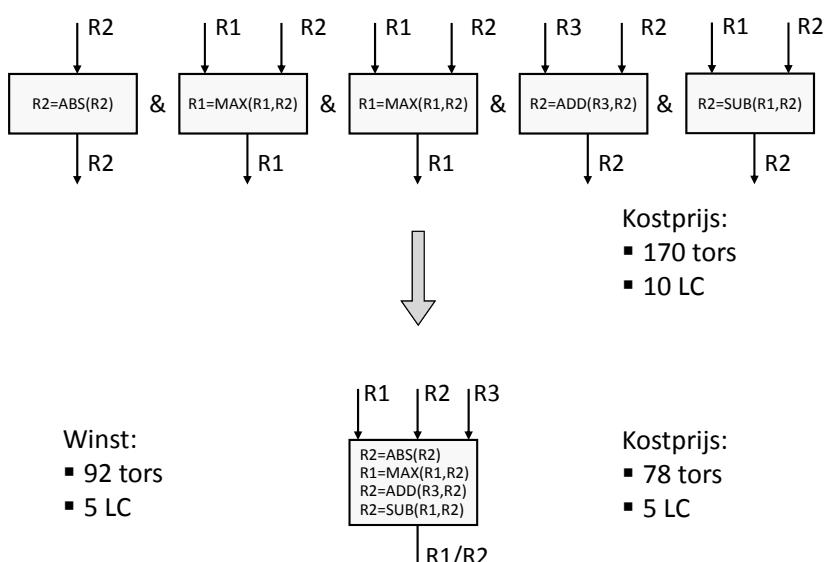
## Geheugen

## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

Winst van één  
ABS & MAX & MAX & ADD & SUB FU

## FSMD

## Beschrijving

## Geheugen

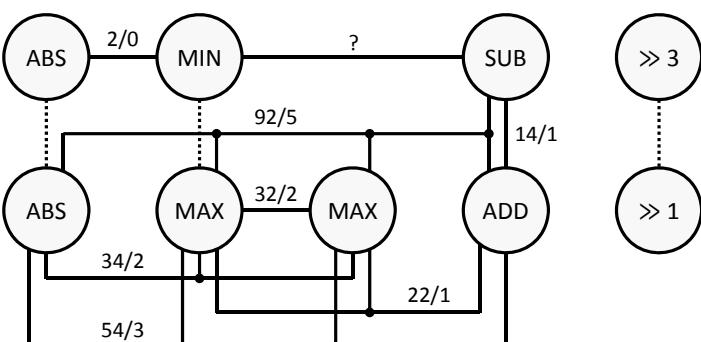
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

Beschrijving

Geheugen

Synthese

▪ Basis

▪ Controller

⇒ Datapad  
• variabelen  
⇒ bewerkingen  
• verbindingen  
• registers

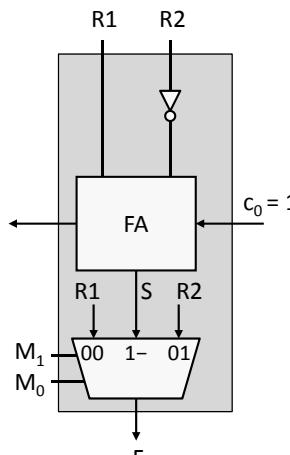
▪ Extra

Tijds gedrag

VHDL

## Realisatie MIN &amp; SUB FU

$$\begin{cases} R3 = \text{MIN}(R1, R2) \\ R2 = \text{SUB}(R1, R2) \end{cases}$$



$$M_1 = \text{SUB/MIN}^* \\ M_0 = S_{n-1}'$$

## Kostprijs per bit:

- 38 tors (SUB) + 18 (MUX) = 56 tors
- 2 LC (SUB) + 2 (MUX) = 4 LC

## FSMD

Beschrijving

Geheugen

Synthese

▪ Basis

▪ Controller

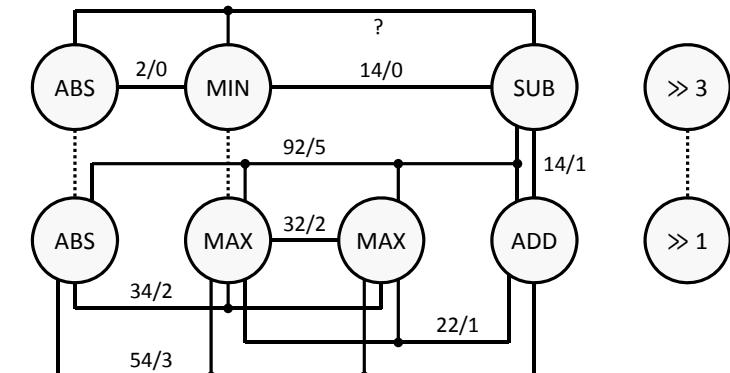
⇒ Datapad  
• variabelen  
⇒ bewerkingen  
• verbindingen  
• registers

▪ Extra

Tijds gedrag

VHDL

## SRA: compatibiliteitsgraaf FU



## FSMD

Beschrijving

Geheugen

Synthese

▪ Basis

▪ Controller

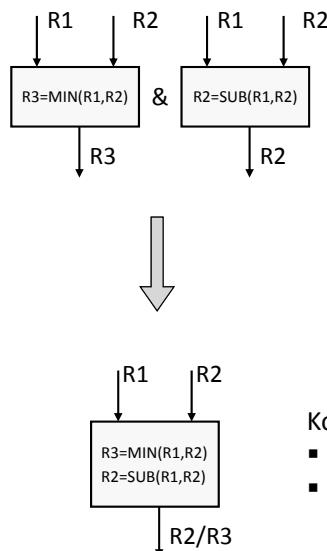
⇒ Datapad  
• variabelen  
⇒ bewerkingen  
• verbindingen  
• registers

▪ Extra

Tijds gedrag

VHDL

## Winst van één MIN &amp; SUB FU



- Kostprijs:
- 70 tors
  - 4 LC

- Kostprijs:
- 56 tors
  - 4 LC

- Winst:
- 14 tors
  - 0 LC

## FSMD

Beschrijving

Geheugen

Synthese

▪ Basis

▪ Controller

⇒ Datapad  
• variabelen  
⇒ bewerkingen  
• verbindingen  
• registers

▪ Extra

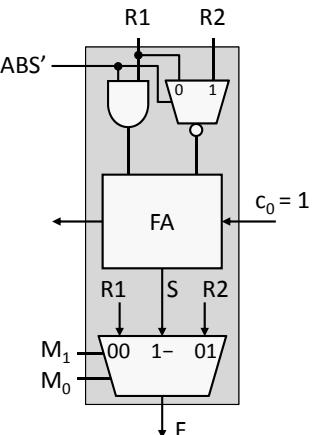
Tijds gedrag

VHDL

## Realisatie ABS &amp; MIN &amp; SUB FU

$$\begin{cases} R1 = \text{ABS}(R1) \\ R3 = \text{MIN}(R1, R2) \\ R2 = \text{SUB}(R1, R2) \end{cases}$$

$$\begin{aligned} \text{SUB als ABS} &= \text{MIN} = 0 \\ M_1 &= \text{ABS}' \cdot \text{MIN}' + \text{MIN}' \cdot S_{n-1}' \\ M_0 &= \text{ABS}' \cdot S_{n-1}' \end{aligned}$$



Een ABS & SUB FU heeft dezelfde hardware nodig!

## Kostprijs per bit:

- 36 tors (FA) + 6 (AND) + 14 (MUX & INV) + 18 (MUX) = 74 tors
- 2 LC (FA & AND & MUX & INV) + 2 (MUX) = 4 LC

## FSMD

## Beschrijving

## Geheugen

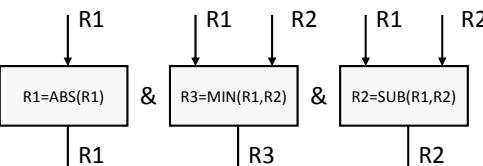
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

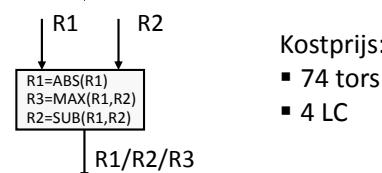
## Tijds gedrag

## VHDL

## Winst van één ABS &amp; MIN &amp; SUB FU



Kostprijs:  
 ▪ 102 tors  
 ▪ 6 LC



Kostprijs:  
 ▪ 74 tors  
 ▪ 4 LC

Winst:  
 ▪ 28 tors  
 ▪ 2 LC

## FSMD

## Beschrijving

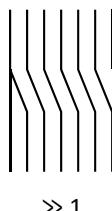
## Geheugen

## Synthese

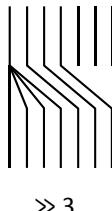
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL



Kostprijs per bit:  
 ▪ 0 tors  
 ▪ 0 LC



Vermits de SHIFT niets kost kunnen we de globale kostprijs niet verminderen door het met iets te combineren!

## FSMD

## Beschrijving

## Geheugen

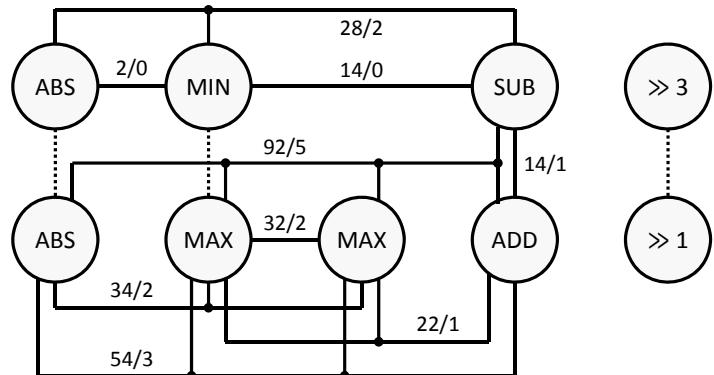
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



Is het zinvol om SHIFT met enige andere FU te combineren?

## Realisatie SHIFT FU

## FSMD

## Beschrijving

## Geheugen

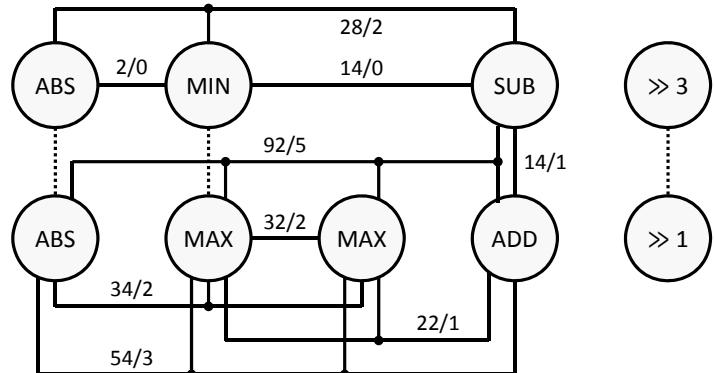
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: compatibiliteitsgraaf FU



- Niet alle combinaties zijn onderzocht, maar we nemen aan dat de andere 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!

## FSMD

## Beschrijving

## Geheugen

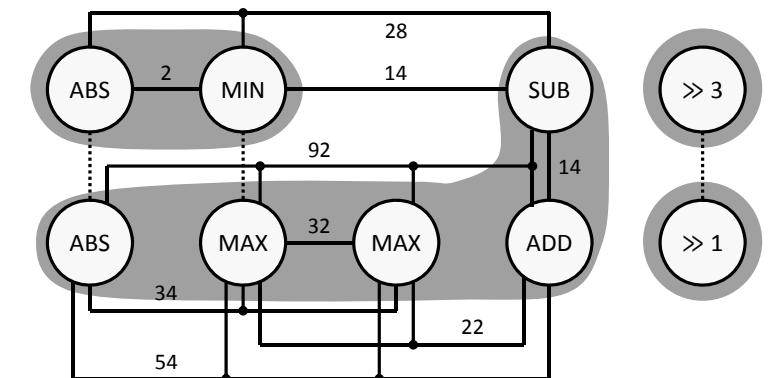
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: minimalisering voor CMOS ASIC



Mogelijkheid: (ABS & MIN), (ABS & MAX & MAX & ADD & SUB),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 94 tors

## FSMD

## Beschrijving

## Geheugen

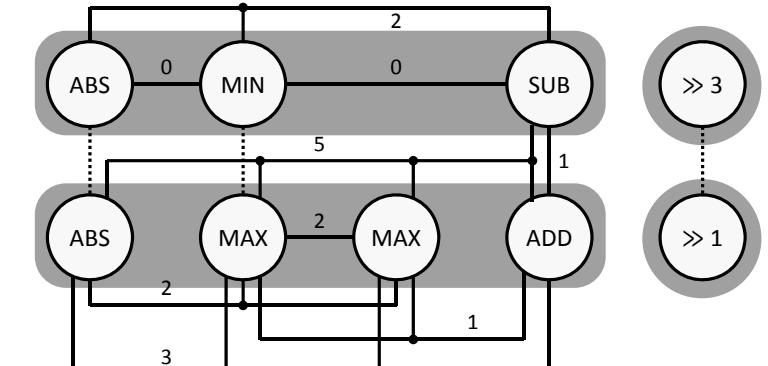
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: minimalisering voor FPGA



Mogelijkheid 1: (ABS), (MIN), (ABS & MAX & MAX & ADD & SUB),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 5 LC

Mogelijkheid 2: (ABS & MIN & SUB), (ABS & MAX & MAX & ADD),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 5 LC

Mogelijkheid 2 ⇒ 1 FU minder nodig (minder verbindingen)  
 $\Rightarrow$  grotere MUX aan ingang R2

## FSMD

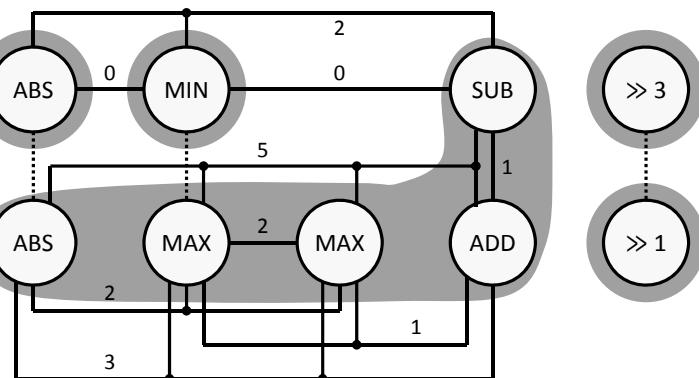
## Beschrijving

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

## SRA: minimalisering voor FPGA



Mogelijkheid 1: (ABS), (MIN), (ABS & MAX & MAX & ADD & SUB),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 5 LC

ABS & MIN bij voorkeur niet samenvoegen  
 omdat dit complexere aanstuurlogica geeft.

## FSMD

## Beschrijving

## Geheugen

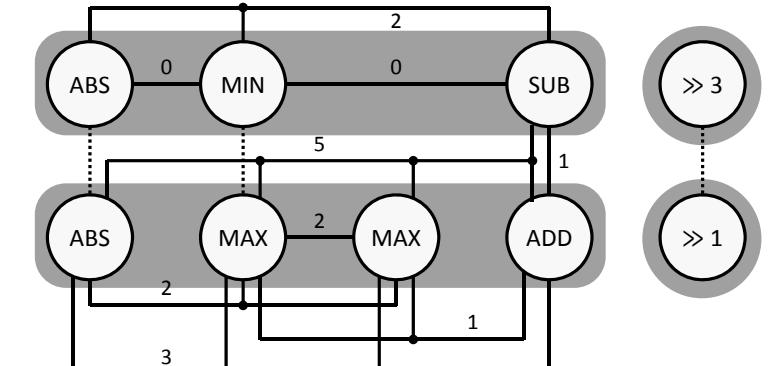
## Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

## Tijds gedrag

## VHDL

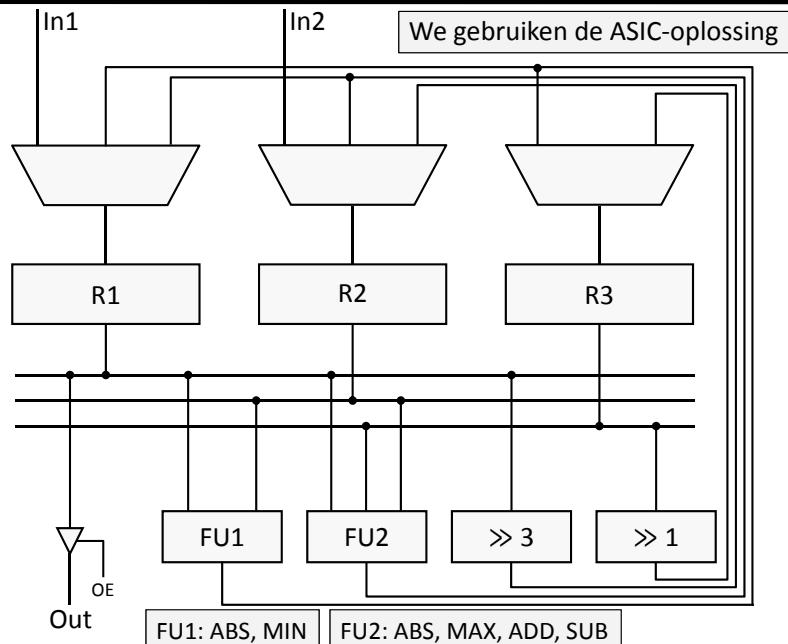
## SRA: minimalisering voor FPGA



Mogelijkheid 1: (ABS), (MIN), (ABS & MAX & MAX & ADD & SUB),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 5 LC

Mogelijkheid 2: (ABS & MIN & SUB), (ABS & MAX & MAX & ADD),  
 $(\gg 3)$ ,  $(\gg 1)$ : winst 5 LC

Mogelijkheid 2 ⇒ 1 FU minder nodig (minder verbindingen)  
 $\Rightarrow$  grotere MUX aan ingang R2

Implementatie SRA  
na samenvoegen bewerkingen

We gebruiken de ASIC-oplossing

FU1: ABS, MIN      FU2: ABS, MAX, ADD, SUB

*Beschrijving  
Geheugen  
Synthese*

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

*Tijds gedrag  
VHDL*

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Kostprijsberekening per bit

## □ Kostprijs FU originele FSMD:

- ABS (32 tors; 2 LC) × 2
- MIN (32 tors; 2 LC)
- MAX (32 tors; 2 LC) × 2
- SUB (38 tors; 2 LC)
- ADD (36 tors; 2 LC)

## □ Na samenvoegen bewerkingen:

- FU1: 62 tors; 4 LC
- FU2: 78 tors; 5 LC

## □ Nieuwe kostprijs registers & MUX

- Registers met 3-naar-1 MUX: (44 + 18 tors; 2 LC) × 2
- Register met 2-naar-1 MUX: (44 + 12 tors; 1 LC)

*Beschrijving  
Geheugen  
Synthese*

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

*Tijds gedrag  
VHDL*

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Niet-programmeerbare processoren

## □ Beschrijving van een algoritme

## □ Extra geheugen bouwblokken

## ➔ Synthese: naar een minimale hardware

- Basisprincipes
- Sneller ontwerp van controller
- ➔ Minimalisering datapad
  - Variabelen samenvoegen
  - Bewerkingen samenvoegen
  - ➔ Verbindingen samenvoegen ('bus sharing')
  - Registers samenvoegen tot een registerbank
- Andere optimaliseringen

## □ Aandachtspunten qua tijdsgedrag

## □ VHDL voor synthese en simulatie

*Beschrijving  
Geheugen  
Synthese*

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - ⇒ bewerkingen
  - verbindingen
  - registers
- Extra

*Tijds gedrag  
VHDL*

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Vergelijking kostprijs per bit

| Samenvoegen | transistoren |     |     | LC  |    |     | Verbindingen |
|-------------|--------------|-----|-----|-----|----|-----|--------------|
|             | Reg          | FU  | Tot | Reg | FU | Tot |              |
| Origineel   | 484          | 234 | 718 | 11  | 14 | 25  | 20           |
| Register    | 198          | 234 | 432 | 6   | 14 | 20  | 12           |
| FU          | 180          | 140 | 320 | 5   | 9  | 14  | 7            |
| Bus         |              |     |     |     |    |     |              |
| Reg.bank    |              |     |     |     |    |     |              |

Optimaliseringen beïnvloeden elkaar: samenvoegen FU verandert

- het aantal registers
- het aantal verbindingen

*Beschrijving  
Geheugen  
Synthese*

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

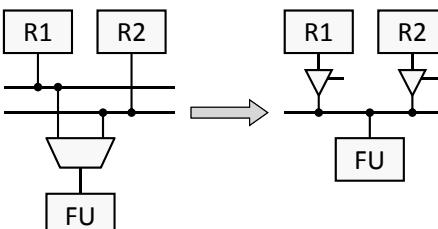
*Tijds gedrag  
VHDL*

KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

# Verbindingen samenvoegen

= vervang twee of meer verbindingen, die niet tegelijkertijd gebruikt worden door 1 verbinding

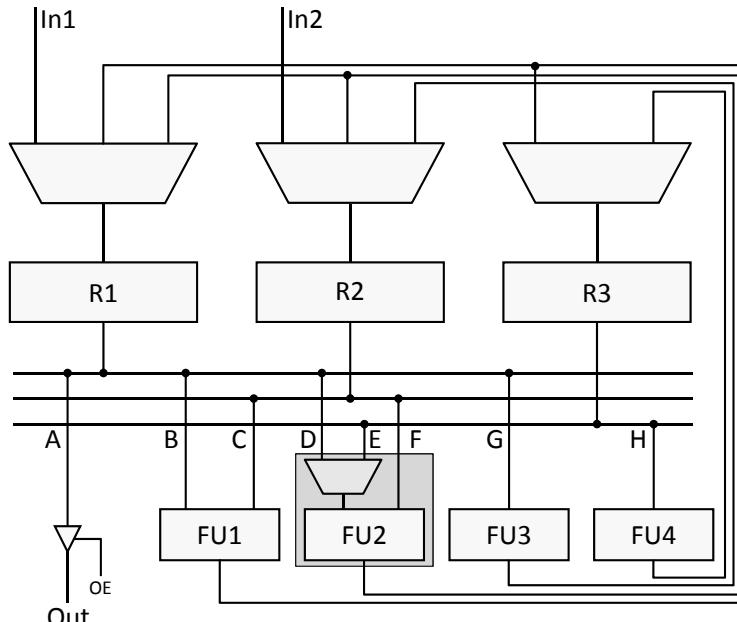
- Dit vermindert de bedrading, wat tegenwoordig een belangrijke kost vertegenwoordigt,
- ten koste van extra 3-state buffers wanneer meerdere verschillende bronnen dezelfde bus aansturen,
- maar het vermindert ook het aantal MUX wanneer één bus verschillende verbindingen verenigt die dezelfde FU-ingang aansturen.



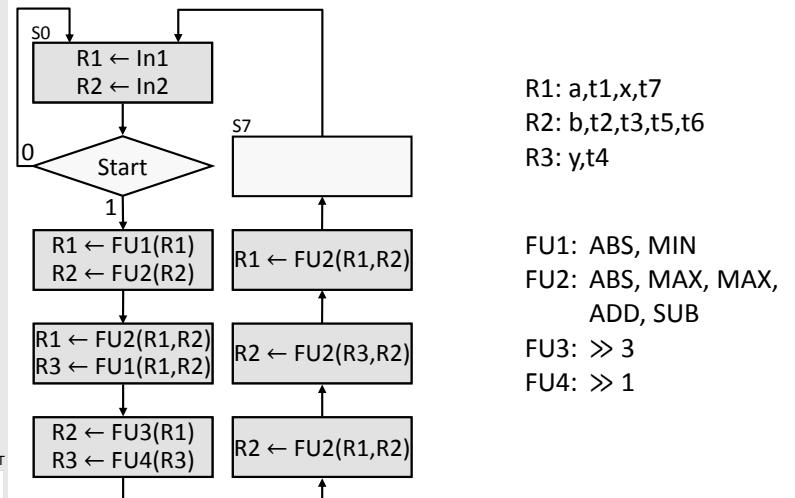
# Verbindingen samenvoegen

- Compatibiliteitsgraaf voor verbindingen:
  - Knopen komen overeen met verbindingen
  - Incompatibiliteitsranden tussen twee verbindingen die in eenzelfde toestand gebruikt worden en verschillende aansturingen hebben
  - Prioriteitsranden tussen twee verbindingen die
    - eenzelfde aansturing hebben (besparing # 3-state buffers) of
    - eenzelfde gebruiker (besparing # MUX)
- Tweemaal voor de twee groepen bussen:
  - 1) operanden (verbindingen registers → FU)
  - 2) resultaten (verbindingen FU → registers)

# SRA: operandverbindingen

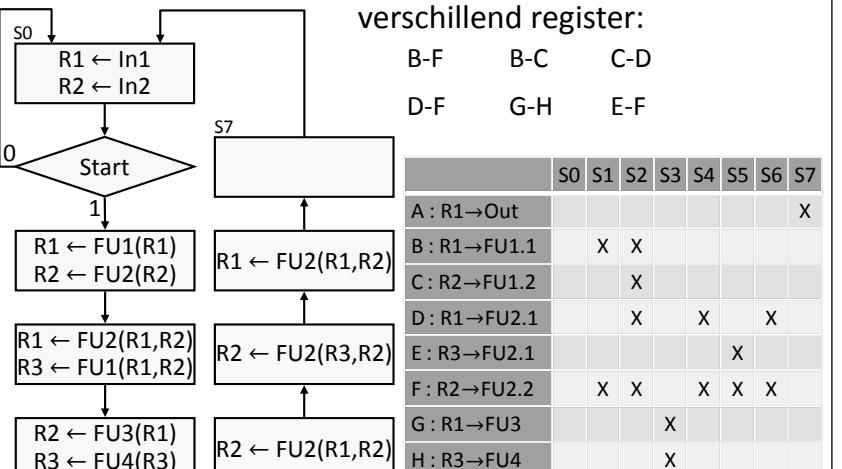


# SRA herschreven met registers & FU



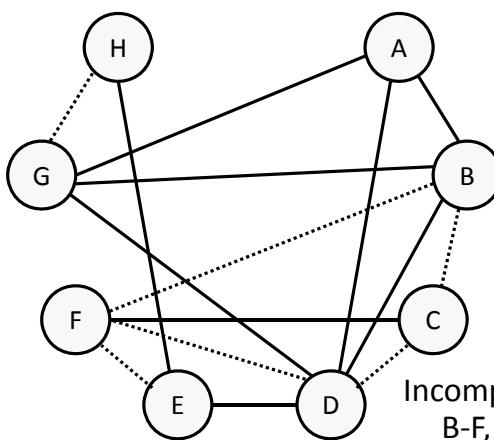
# SRA: compatibiliteitsgraaf operandverbindingen

Nu ook verbindingen naar de uitgangen in rekening brengen!



# SRA: compatibiliteitsgraaf operandverbindingen

Prioriteitsranden:  
zelfde aansturing of gebruik

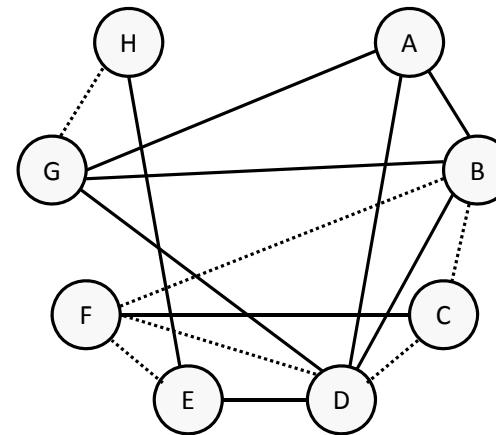


|   |          |
|---|----------|
| A | R1→Out   |
| B | R1→FU1.1 |
| C | R2→FU1.2 |
| D | R1→FU2.1 |
| E | R3→FU2.1 |
| F | R2→FU2.2 |
| G | R1→FU3   |
| H | R3→FU4   |

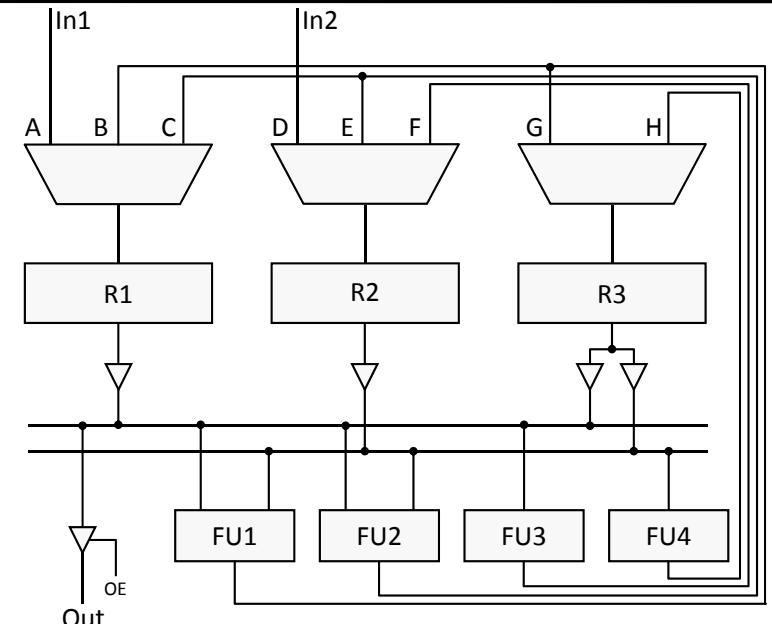
Incompatibiliteitsranden:  
B-F, B-C, C-D, D-F, G-H, E-F

# SRA: compatibiliteitsgraaf operandverbindingen

- Operandbus 1: A, B, D, E, G
- Operandbus 2: C, F, H



# SRA: resultaatverbindingen



# SRA: compatibiliteitsgraaf resultaatverbindingen

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 : FU2→R1 |    |    | X  |    |    |    |    |    |
| D : In2→R2 |    | X  |    |    |    |    |    |    |
| E : FU2→R2 |    |    | X  | X  | X  |    |    |    |
| F : FU3→R2 |    |    |    | X  |    |    |    |    |
| G : FU1→R3 |    |    | X  |    |    |    |    |    |
| H : FU4→R3 |    |    |    | X  |    |    |    |    |

Beschrijving

Geheugen

Synthese

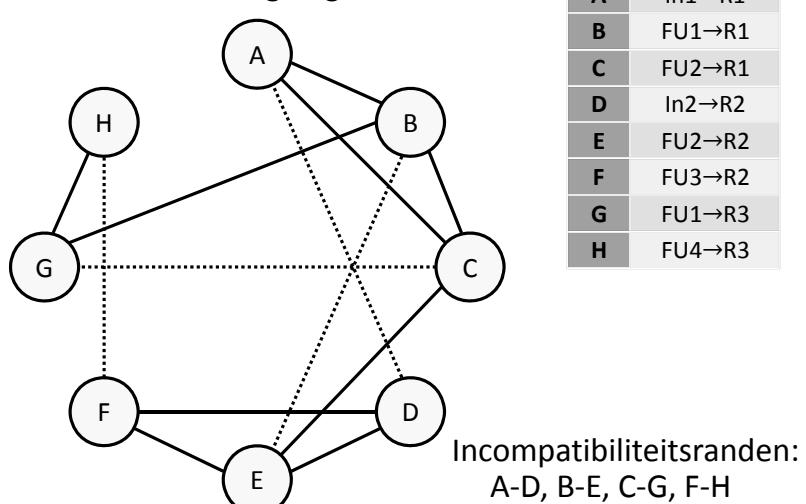
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

Tijdsgedrag

VHDL

# SRA: compatibiliteitsgraaf resultaatverbindingen

Prioriteitsranden:  
zelfde aansturing of gebruik



Beschrijving

Geheugen

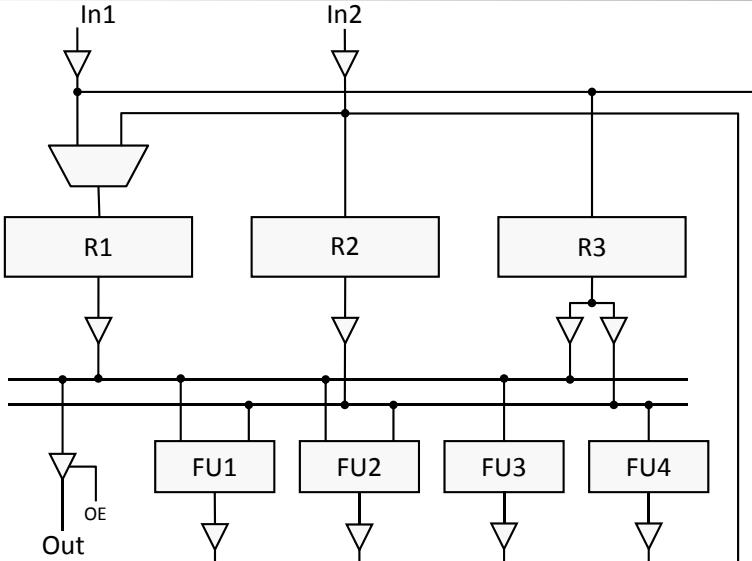
Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

Tijdsgedrag

VHDL

# Implementatie SRA na samenvoegen verbindingen



Beschrijving

Geheugen

Synthese

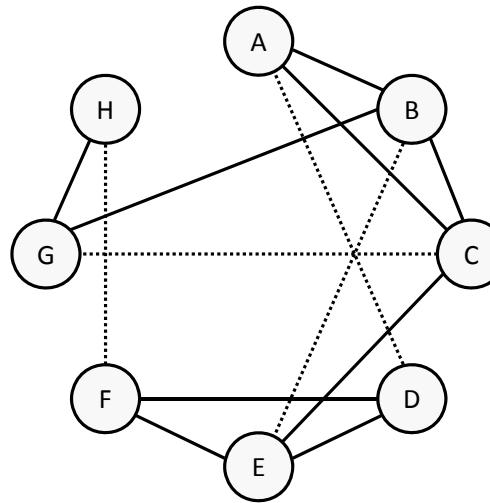
- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

Tijdsgedrag

VHDL

# SRA: compatibiliteitsgraaf resultaatverbindingen

- Resultaatbus 1: A, B, G, H
- Resultaatbus 2: C, D, E, F



Beschrijving

Geheugen

Synthese

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

Tijdsgedrag

VHDL

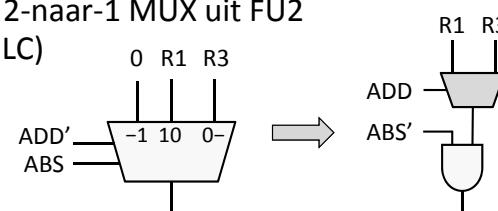
# Berekening extra kostprijs per bit

- Qua registers en ingangen

- +  $2 \times$  verwijdering 3-naar-1 MUX voor register  
=  $-2 \times (18 \text{ tors; } 1 \text{ LC})$
- +  $6 \times$  3-state buffers  
 $= 6 \times (10 \text{ tors; } 0 \text{ LC})$

- Qua FU

- +  $4 \times$  3-state buffers  
 $= 4 \times (10 \text{ tors; } 0 \text{ LC})$
- + verwijdering 2-naar-1 MUX uit FU2  
 $= -(6 \text{ tors; } 0 \text{ LC})$



**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

**Tijds gedrag****VHDL**

# Vergelijking kostprijs per bit

| Samenvoegen | transistoren |     |     | LC  |    |     | Verbindingen |
|-------------|--------------|-----|-----|-----|----|-----|--------------|
|             | Reg          | FU  | Tot | Reg | FU | Tot |              |
| Origineel   | 484          | 234 | 718 | 11  | 14 | 25  | 20           |
| Register    | 198          | 234 | 432 | 6   | 14 | 20  | 12           |
| FU          | 180          | 140 | 320 | 5   | 9  | 14  | 7            |
| Bus         | 204          | 174 | 378 | 3   | 9  | 12  | 4            |
| Reg.bank    |              |     |     |     |    |     |              |

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - ⇒ verbindingen
  - registers
- Extra

**Tijds gedrag****VHDL**

# Niet-programmeerbare processoren

□ Beschrijving van een algoritme

□ Extra geheugen bouwblokken

→ Synthese: naar een minimale hardware

➢ Basis principes

➢ Sneller ontwerp van controller

→ Minimalisering datapad

- Variabelen samenvoegen

- Bewerkingen samenvoegen

- Verbindingen samenvoegen

→ Registers samenvoegen tot een registerbank ('register port sharing')

➢ Andere optimaliseringen

□ Aandachtspunten qua tijds gedrag

□ VHDL voor synthese en simulatie

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers
- Extra

**Tijds gedrag****VHDL**

# Samenvoegen registers

- = voeg registers samen tot een registerbank om
  - # leespoorten te verminderen  
⇒ minder ingangs-MUX
  - # schrijfpoorten te verminderen  
⇒ minder 3-state buffers

## Methodologie

- Registertoegangstabel ('Register Access Table')
  - = de lees- en schrijfoperaties van registers voor elke toestand
- Optimaliseer het samenvoegen van registers
  - traditionele methoden (bijv. max-cut)
  - exhaustief voor eenvoudige gevallen (zoals voor SRA voorbeeld)

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers
- Extra

**Tijds gedrag****VHDL**

# Registertoegangstabel (lezen)

Vertrek van tabel met het gebruik van operandverbindingen

|                | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|----------------|----|----|----|----|----|----|----|----|
| A : R1 → Out   |    |    |    |    |    |    |    | X  |
| B : R1 → FU1.1 |    | X  | X  |    |    |    |    |    |
| C : R2 → FU1.2 |    |    | X  |    |    |    |    |    |
| D : R1 → FU2.1 |    |    | X  | X  |    |    | X  |    |
| E : R3 → FU2.1 |    |    |    |    |    | X  |    |    |
| F : R2 → FU2.2 |    | X  | X  |    | X  | X  | X  |    |
| G : R1 → FU3   |    |    |    | X  |    |    |    |    |
| H : R3 → FU4   |    |    |    | X  |    |    |    |    |



|    | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|----|----|----|----|----|----|----|----|----|
| R1 |    |    | R  | R  | R  | R  |    | R  |
| R2 |    |    | R  | R  |    | R  | R  | R  |
| R3 |    |    |    |    | R  |    | R  |    |

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers

**Extra****Tijdsgedrag****VHDL**

# Registertoegangstabel (schrijven)

Vertrek van tabel met het gebruik van resultaatverbindingen

|            | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|------------|----|----|----|----|----|----|----|----|
| A : In1→R1 | X  |    |    |    |    |    |    |    |
| B : FU1→R1 |    | X  |    |    |    |    |    |    |
| C : FU2→R1 |    |    | X  |    |    |    | X  |    |
| D : In2→R2 | X  |    |    |    |    |    |    |    |
| E : FU2→R2 |    | X  |    |    | X  | X  |    |    |
| F : FU3→R2 |    |    |    | X  |    |    |    |    |
| G : FU1→R3 |    |    | X  |    |    |    |    |    |
| H : FU4→R3 |    |    |    | 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   |     |    |

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers

**Extra****Tijdsgedrag****VHDL**

# SRA: exhaustieve minimalisering

|    | 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   |     |    |

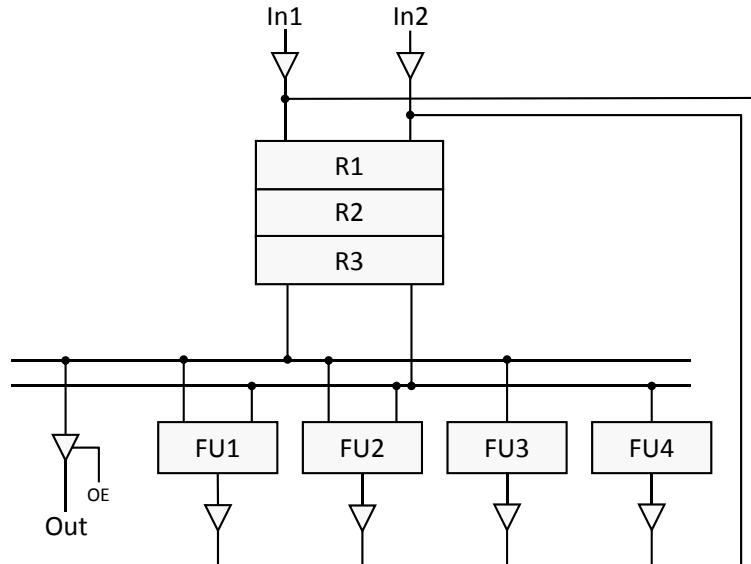
| registerbanken   | # lees/schrijfpoorten | # poorten |
|------------------|-----------------------|-----------|
| (R1), (R2), (R3) | (1;1) + (1;1) + (1;1) | 6         |
| (R1,R2), (R3)    | (2;2) + (1;1)         | 6         |
| (R1,R3), (R2)    | (2;2) + (1;1)         | 6         |
| (R1), (R2,R3)    | (1;1) + (2;2)         | 6         |
| (R1,R2,R3)       | (2;2)                 | 4         |

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers
- Extra

**Tijdsgedrag****VHDL**

# Implementatie SRA na alle samenvoegingen

**Beschrijving****Geheugen****Synthese**

- Basis
- Controller
- ⇒ Datapad
  - variabelen
  - bewerkingen
  - verbindingen
  - ⇒ registers
- Extra

**Tijdsgedrag****VHDL**

# Vergelijking kostprijs per bit

| Samenvoegen | transistoren |     |     | LC  |    |     | Verbindingen |
|-------------|--------------|-----|-----|-----|----|-----|--------------|
|             | Reg          | FU  | Tot | Reg | FU | Tot |              |
| Origineel   | 484          | 234 | 718 | 11  | 14 | 25  | 20           |
| Register    | 198          | 234 | 432 | 6   | 14 | 20  | 12           |
| FU          | 180          | 140 | 320 | 5   | 9  | 14  | 7            |
| Bus         | 204          | 174 | 378 | 3   | 9  | 12  | 4            |
| Reg.bank    | 152          | 174 | 326 | 3   | 9  | 12  | 4            |

□ Winst t.g.v. gebruik registerbank

> 1 × 2-naar-1 MUX (12 tors; 0 LC)

> 4 × 3-state buffers (40 tors; 0 LC)

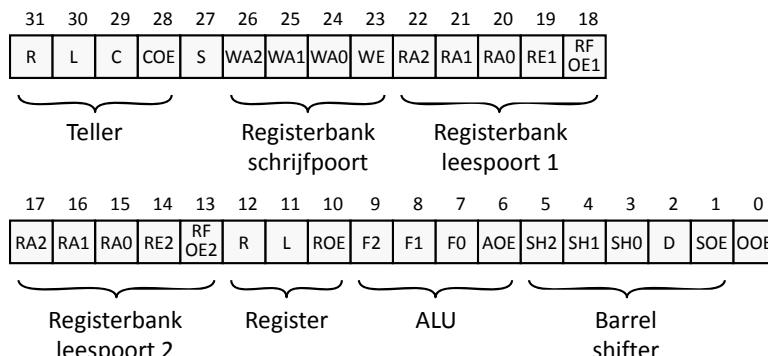
Opmerking: dit houdt geen rekening met de hogere complexiteit van een registerbank (bijv. de adresseerlogica)

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugenbouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basisprincipes
  - Sneller ontwerp van controller
  - Minimalisering datapad
  - Andere optimaliseringen
    - Minimaal instructiewoord (uitgangen controller)
    - 'Chaining': meerdere bewerkingen in 1 klokcyclus
    - 'Pipelining' & 'multicycling':  
meerdere klokcycli voor 1 bewerking
- Aandachtspunten qua tijdsgezag
- VHDL voor synthese en simulatie

## Instructiewoord

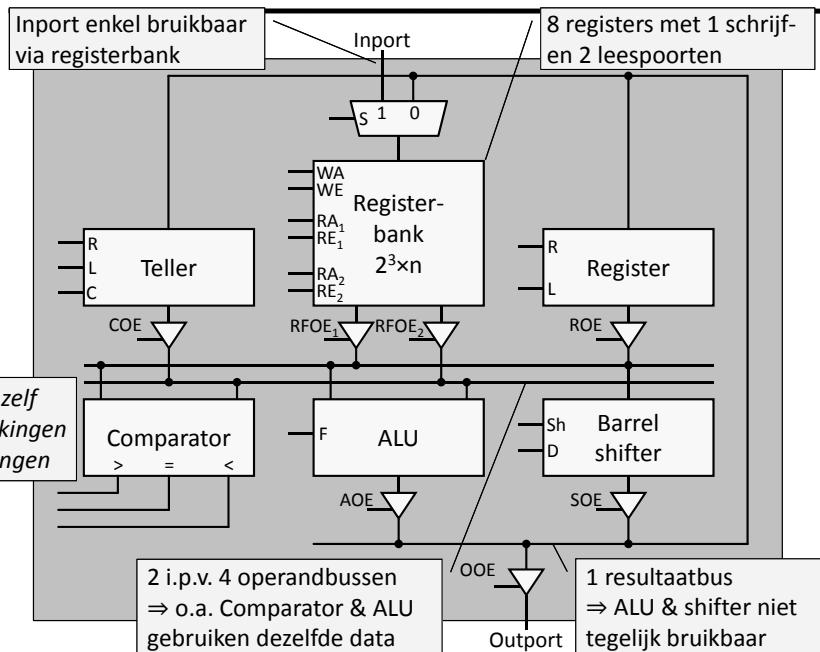
- = verzameling bits voor aansturing datapad
- 32-bit instructiewoord voor vorig voorbeeld



- Geheugensteun ('mnemonic') voor bitpatronen van (deel van) instructiewoord, bijv. ADD  
⇒ eenvoudiger en duidelijker

Resultaat van zelf opgelegde beperkingen en minimaliseringen

## Een typisch datapad



## Instructieword

- |    |    |    |     |    |     |     |     |    |     |     |     |     |        |     |     |     |     |        |    |    |     |    |    |    |     |     |     |     |   |     |     |
|----|----|----|-----|----|-----|-----|-----|----|-----|-----|-----|-----|--------|-----|-----|-----|-----|--------|----|----|-----|----|----|----|-----|-----|-----|-----|---|-----|-----|
| 31 | 30 | 29 | 28  | 27 | 26  | 25  | 24  | 23 | 22  | 21  | 20  | 19  | 18     | 17  | 16  | 15  | 14  | 13     | 12 | 11 | 10  | 9  | 8  | 7  | 6   | 5   | 4   | 3   | 2 | 1   | 0   |
| R  | L  | C  | COE | S  | WA2 | WA1 | WA0 | WE | RA2 | RA1 | RA0 | RE1 | RF OE1 | RA2 | RA1 | RA0 | RE2 | RF OE2 | R  | L  | ROE | F2 | F1 | FO | AOE | SH2 | SH1 | SH0 | D | SOE | OOE |
- De grootte kan verminderd worden omdat sommige operaties niet tegelijkertijd kunnen:
    - 1<sup>e</sup> operandbus bevat ofwel Registerbank leespoort 1 ofwel Register leespoort (-1)
    - 2<sup>e</sup> operandbus bevat ofwel Registerbank leespoort 2 ofwel Teller uitgang (-1)
    - Registerbank RE<sub>i</sub> = RFOE<sub>i</sub> (-2)
    - resultaatbus bevat resultaat van ofwel ALU ofwel Barrel shifter (-1)
    - ALU & Shift niet tegelijkertijd: zelfde instructiebits (-3)
    - Teller 'Count' & 'Load' zijn exclusief (-1)
  - Dit resulteert in een eenvoudiger ontwerp, minder verbindingen en eventueel minder componenten
  - Extra beperkingen qua parallelisme zijn mogelijk, maar dit resulteert in een grotere uitvoeringstijd

Beschrijving  
Geheugen  
Synthese  
▪ Basis  
▪ Controller  
▪ Datapad  
⇒ Extra  
• instructie  
⇒ chaining  
• multicycling  
• pipelining

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basis principes
  - Sneller ontwerp van controller
  - Minimalisering datapad
  - ➔ Andere optimaliseringen (hogere verwerkingskracht)
    - Minimaal instructiewoord
    - ‘Chaining’: meerdere bewerkingen in 1 klokcyclus
    - ‘Pipelining’ & ‘multicycling’
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

Beschrijving  
Geheugen  
Synthese  
▪ Basis  
▪ Controller  
▪ Datapad  
⇒ Extra  
• instructie  
• chaining  
⇒ multicycling  
• pipelining

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- ➔ Synthese: naar een minimale hardware
  - Basis principes
  - Sneller ontwerp van controller
  - Minimalisering datapad
  - ➔ Andere optimaliseringen (hogere verwerkingskracht)
    - Minimaal instructiewoord
    - ‘Chaining’
    - ‘Pipelining’ & ‘multicycling’:  
meerdere klokcycli voor 1 bewerking
- Aandachtspunten qua tijds gedrag
- VHDL voor synthese en simulatie

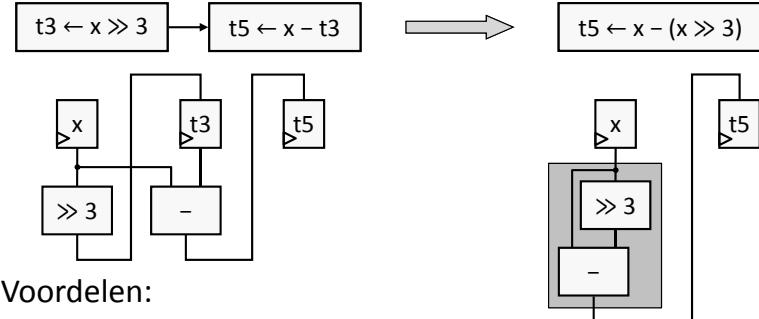
Beschrijving  
Geheugen  
Synthese  
▪ Basis  
▪ Controller  
▪ Datapad  
⇒ Extra  
• instructie  
⇒ chaining  
• multicycling  
• pipelining

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# ‘Chaining’

- = plaats een tussenresultaat niet in een register als
  - levensduur van tussenresultaat = 1 klokperiode
  - vertraging<sub>FU1</sub> + vertraging<sub>FU2</sub> < klokperiode
  - OK qua specificaties tijds gedrag



## Voordelen:

- minder toestanden
- minder registers eventueel
- minder klok cycli voor het hele algoritme

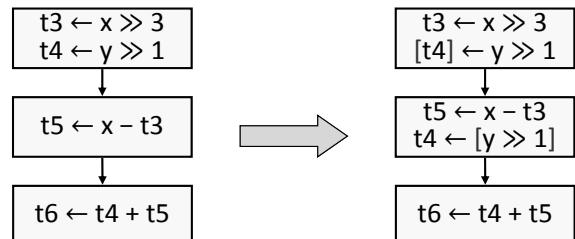
Beschrijving  
Geheugen  
Synthese  
▪ Basis  
▪ Controller  
▪ Datapad  
⇒ Extra  
• instructie  
• chaining  
⇒ multicycling  
• pipelining

Tijds gedrag  
VHDL

KATHOLIEKE UNIVERSITEIT  
LEUVEN

# ‘Multicycling’

- = geef een FU (of RAM) meerdere klok cycli om een bewerking af te ronden
  - als meer dan 1 klok cyclus tussen generatie en gebruik van het resultaat van een bewerking



## Voordelen:

- Goedkopere FU (want trager) of hogere klof frequentie

## Opmerking:

- Levensduur resultaat start als bewerking eindigt
- Eventueel extra toestand(en) nodig

Beschrijving  
Geheugen

Synthese

- Basis
- Controller
- Datapad
- ⇒ Extra
  - instructie
  - chaining
  - multicycling
  - ⇒ pipelining

Tijds gedrag  
VHDL

# Versnellen trage operaties

## □ Sequentieel



## □ Parallel



## □ Pipelining (lopende band)



Beschrijving  
Geheugen

Synthese

- Basis
- Controller
- Datapad
- ⇒ Extra
  - instructie
  - chaining
  - multicycling
  - ⇒ pipelining

Tijds gedrag  
VHDL

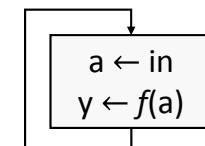
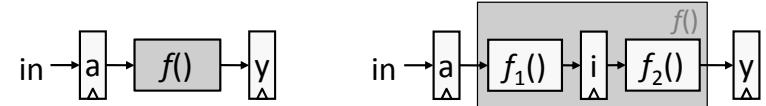
# ‘Pipelining’

= opsplitsing bewerking in  $n$  onderdelen,  
die ieder slechts  $1/n$  van de tijd gebruiken

➤ klokfrequentie  $\pm \times n$

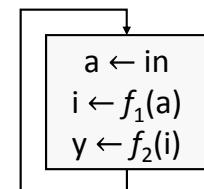
⇒ meestal  $\approx$  datafrequentie (performantie)

➤ berekening 1 resultaat ('latency time')  $\pm$  zelfde



$$a_t = in_{t-1}$$

$$y_t = f(a_{t-1})$$



$$a_t = in_{t-1}$$

$$i_t = f_1(a_{t-1})$$

$$y_t = f_2(i_{t-1})$$

$$= f_2(f_1(a_{t-2}))$$

klokfrequentie  $\times 2$

Beschrijving  
Geheugen

Synthese

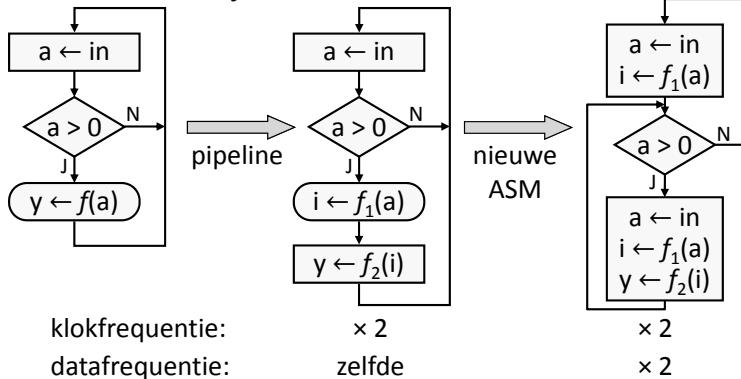
- Basis
- Controller
- Datapad
- ⇒ Extra
  - instructie
  - chaining
  - multicycling
  - ⇒ pipelining

Tijds gedrag  
VHDL

# Pipelining: impact op ASM

## □ Normaal extra toestanden nodig

- enkel waar pipelining
- geen performantieverlies want klokfreq.  $\times n$
- performantie dikwijls te verbeteren door de ASM te herschrijven



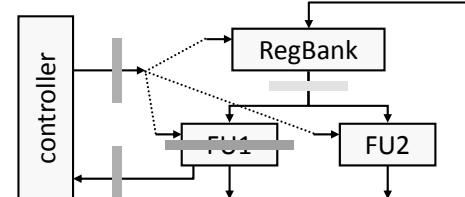
Beschrijving  
Geheugen

Synthese

- Basis
- Controller
- Datapad
- ⇒ Extra
  - instructie
  - chaining
  - multicycling
  - ⇒ pipelining

Tijds gedrag  
VHDL

# Soorten pipelining



## □ van FU (die in het kritisch pad)

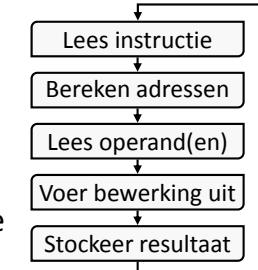
## □ van datapad (als trage registerbank)

## □ van controle (kritisch pad van processoren meestal door controller)

## □ van ASM-schema (splits ASM)

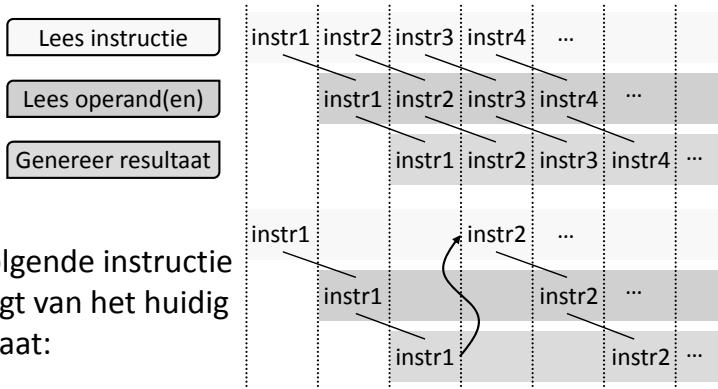
- ieder deel aparte hardware  
⇒ parallelisme

- eventueel ont dubbeling hardware



# Pipelining: nadelen

- Extra hardware (registers)
- Verlies performantiewinst bij terugkoppeling (bewerking gebruikt resultaat van de vorige)  
⇒ “stop de pipeline!” (geen nieuwe data)



# Besluit synthese

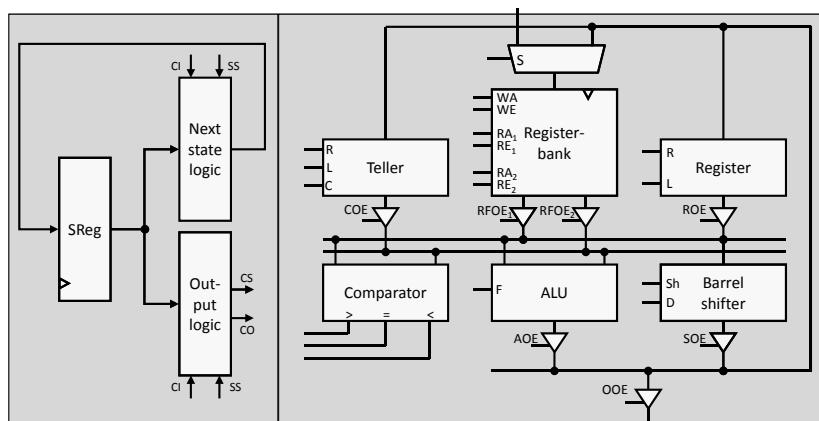
- Zéér moeilijk om optimale implementatie te vinden!
- Eenvoudige optimaliseringen (inclusief max-cut) niet geschikt voor sommige minimaliseringen (bijv. samenvoegen FU)
  - Beste keuze sterk afhankelijk van technologie
  - Algoritme herschrijven geeft veel mogelijkheden
    - ligt niet voor de hand
    - herminimalisering nodig
  - Optimaliseringen beïnvloeden elkaar veel  
⇒ globale optimalisering moeilijk

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
  - ➔ Aandachtspunten qua tijds gedrag
    - Kritisch pad
    - Verschoven kloksignalen ('clock skew')
    - Asynchrone ingangen
- VHDL voor synthese en simulatie

# Kritisch pad

- = langste combinatorisch pad (klok→klok) in beide delen samen

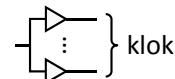
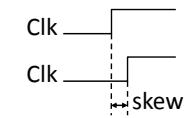


# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
  - ➔ Aandachtspunten qua tijds gedrag
    - Kritisch pad
    - Verschoven kloksignalen ('clock skew')
    - Asynchrone ingangen
- VHDL voor synthese en simulatie

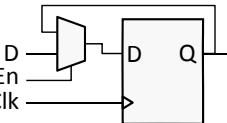
# 'Clock skew' (verschoven kloksignaal)

- = verschil in de tijds ogenblikken waarop een klokflank op verschillende plaatsen in de schakeling wordt waargenomen
- Verschillende looptijd kloksignaal over een verbinding
  - draad = transmissielijn bij hoge frequenties
  - doorlooptijd schakelmatrices FPGA
- Verschillende stijg/daaltijden t.g.v. verschillende capacitive belasting
- Verschillende vertragingstijd van combinatorische logica op klokpad
  - klok buffers (wegen grote fan-out klok)
  - 'clock enable'



# Laadbaar register

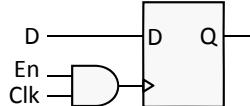
## Met MUX



Nadelen:

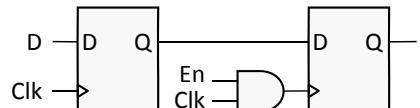
- n-bit MUX duurder dan 1 poort
- FF schakelt ook als geen nieuwe waarde geladen wordt ⇒ nodeloos vermogenverbruik

## Met 'clock enable'



Nadelen:

- "En" mag enkel veranderen als Clk = 0
- introduceert 'clock skew'



# Niet-programmeerbare processoren

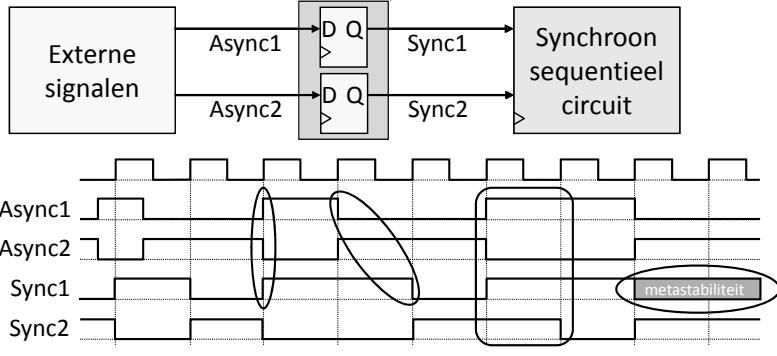
- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
  - ➔ Aandachtspunten qua tijds gedrag
    - Kritisch pad
    - Verschoven kloksignalen
    - Asynchrone ingangen:
      - ➔ mogelijke metastabiliteitsproblemen
- VHDL voor synthese en simulatie

## Wanneer metastabiliteitsproblemen?

- Slecht ontworpen synchrone schakeling
  - Lagere klokfrequentie of snellere logica

kan ook  
met een FIFO

- Asynchrone ingangen

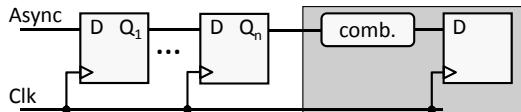


- Synchronisatie

- Enkel onafhankelijke signalen!
- Apart synchronisatiesignaal (bijv. bus-enable)

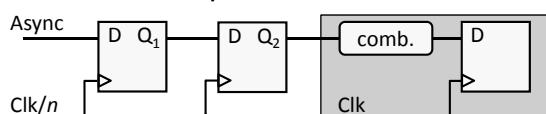
## Hoe kans op metastabiliteit bij synchronisatie verminderen?

- Synchronisatie door  $n$  FF na elkaar



- $t_r = (n - 1) \times (t_{klok} - t_{set-up}) + (t_{klok} - t_{comb} - t_{set-up})$
- Om  $p_{meta}(t_r)$  klein genoeg te maken volstaat  $n = 2$  of  $3$

- Synchronisatie door 2 FF na elkaar op een frequentie = klokfrequentie/ $n$

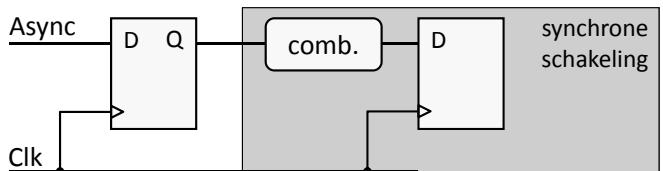


- $t_r = n \times t_{klok} - t_{set-up} + (t_{klok} - t_{comb} - t_{set-up})$
- Als externe signalen niet veel veranderen t.o.v. klok
- Als geen onmiddellijke reactie op verandering vereist

## Hoe kans op metastabiliteit bij synchronisatie verminderen?

Wacht lang genoeg tot FF uit metastabiele toestand:

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



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

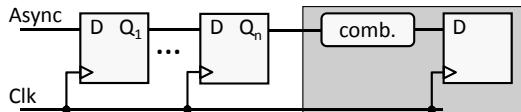
$p_{meta}(t_r)$  kan verkleind worden maar wordt nooit 0!

Verbeteringen:

- Gebruik traagst mogelijke klok
- Gebruik snelle FF (kleine  $\tau$  & kleine  $t_{set-up}$ )
  - Hoger vermogenverbruik!

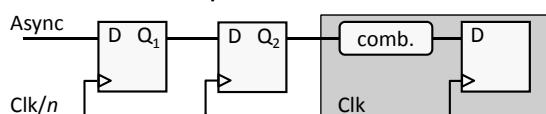
## Hoe kans op metastabiliteit bij synchronisatie verminderen?

- Synchronisatie door  $n$  FF na elkaar



- $t_r = (n - 1) \times (t_{klok} - t_{set-up}) + (t_{klok} - t_{comb} - t_{set-up})$
- Om  $p_{meta}(t_r)$  klein genoeg te maken volstaat  $n = 2$  of  $3$

- Synchronisatie door 2 FF na elkaar op een frequentie = klokfrequentie/ $n$



- $t_r = n \times t_{klok} - t_{set-up} + (t_{klok} - t_{comb} - t_{set-up})$
- Als externe signalen niet veel veranderen t.o.v. klok
- Als geen onmiddellijke reactie op verandering vereist

## Niet-programmeerbare processoren

- Beschrijving van een algoritme

- Extra geheugen bouwblokken

- Synthese: naar een minimale hardware

- Aandachtspunten qua tijds gedrag

### → VHDL voor synthese en simulatie

- Het "process" met meer mogelijkheden

- Simulatie met VHDL

- Synthese: VHDL omzetten in hardware

## Het “process” zonder gevoelighedslijst

**process [is]**  
[declarations and subprograms]...

**begin**  
sequential\_statement(s)  
**end process [label];**

Herhaalt **eindeloos** de sequential\_statement(s) tot aan een “wait”!

- gevoelighedslijst als vereenvoudigde uitdrukking:

**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];**

## Enkele voorbeelden

- Een flankgetriggerde flipflop

DFF: **process is**  
**begin**  
**wait until Clk='1';**  
Q <= D;  
**end process DFF;**

Volwaardig alternatief voor het gebruik van het event-attribuut.

- Een klokgenerator

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

globale constante

## Sequentiële uitdrukking “wait”

- = geef signalen hun nieuwe waarde en onderbreek het proces tot aan een bepaalde voorwaarde voldaan is

**wait [on signal\_name(s)]**  
[until boolean\_expression]  
[for time\_expression];

signal\_name(s)  
= gevoelighedslijst

- **on**: proces hervat als één van de *signal\_name(s)* verandert van waarde
- **until**: proces hervat als *boolean\_expression* waar is of waar wordt als er geen “on” is
- **for**: wacht (niet langer dan) een (simulatie)tijd *time\_expression*
- **wait;**  
= wacht eindeloos (bijv. na een eenmalige puls)
- **wait until clk = '1';**  
= wacht tot clk 1 wordt
- **wait on clk until reset = '0' for 1 ms;**  
= wacht tot reset 0 is op een verandering van clk, maar niet langer dan een simulatietyl van 1 ms

## Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag

### → VHDL voor synthese en simulatie

- Het “process” met meer mogelijkheden
- Simulatie met VHDL
  - Gebeurtenis gedreven simulatie
  - Beschrijving tijds gedrag
  - Testbank
- Synthese: VHDL omzetten in hardware

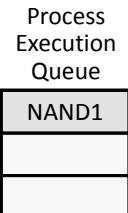
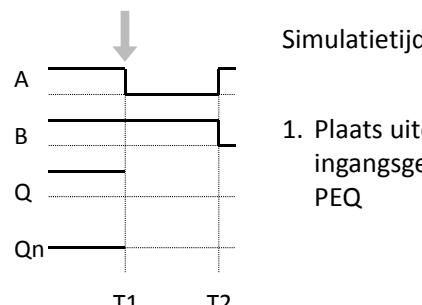
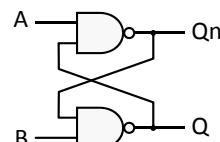
# Gebeurtenis gedreven simulatie

- Het continu berekenen van de uitgangen (bijv. per fs), ook als er niets verandert, zorgt voor nodeloos veel rekenwerk
- Oplossing: 'event-driven' simulatie
  - Een signaaltoekenning creëert een transactie (nieuwe waarde @ nieuwe simulatietijd)
  - Wanneer de simulatietijd voortgaat naar de nieuwe tijd wordt het signaal aangepast (signaal is actief tijdens deze deltacyclus)
  - Een gebeurtenis ('event') treedt enkel op als de nieuwe waarde verschilt van de oude
  - Enkel parallelle uitdrukkingen met gebeurtenissen op hun gevoeligheidslijst worden opnieuw geëvalueerd
- Dit mechanisme zorgt er *enkel* voor dat de simulatie versnelt zonder het gesimuleerde gedrag te wijzigen

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



# Implementatie simulator

1. Plaats alle uitdrukkingen met minstens één gewijzigde ingang in de 'process execution queue' PEQ.
2. Voer alle uitdrukkingen in de PEQ één voor één uit (of tegelijkertijd op een parallelle computer) *zonder* de uitgangssignalen aan te passen ('transaction scheduling').
  - Uitdrukkingen in een proces worden sequentieel uitgevoerd en nieuwe signaalwaarden worden onthouden tot de volgende "wait"-uitdrukking; pas dan zijn ze ter beschikking voor simulatie.
3. Pas de (actieve) uitgangssignalen aan nadat alle uitdrukkingen in de PEQ uitgevoerd zijn.
4. Voeg alle uitdrukkingen, waarvoor een gebeurtenis optreedt t.g.v. een veranderd uitgangssignaal, toe aan de PEQ.
5. Herhaal dit tot de PEQ leeg is.
6. Verhoog de simulatietijd tot het volgende ogenblik waarop een nieuwe gebeurtenis gepland is.

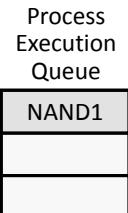
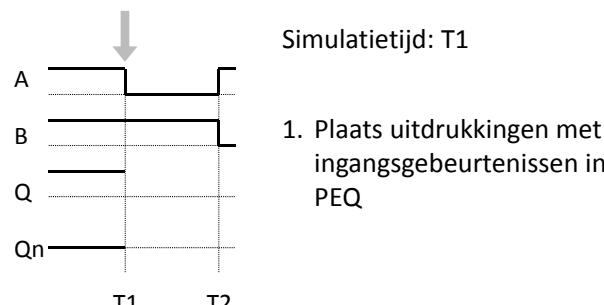
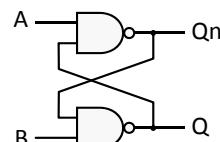
Deltacyclus-convergentie

Deltacyclus

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

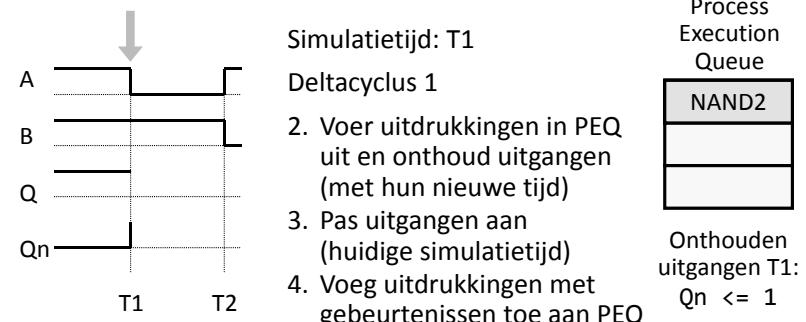
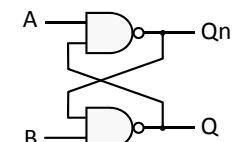
architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;
```

```
architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```

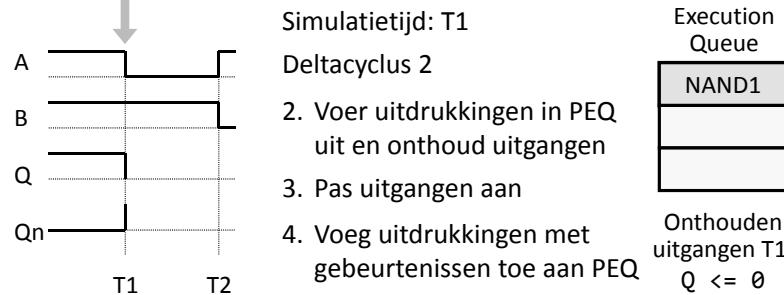
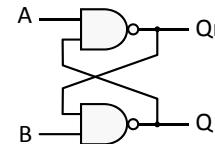


- Proces
- ⇒ Simulatie
  - ⇒ gebeurtenis gedreven
  - beschrijving tijds gedrag
  - testbank
- Synthese

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```

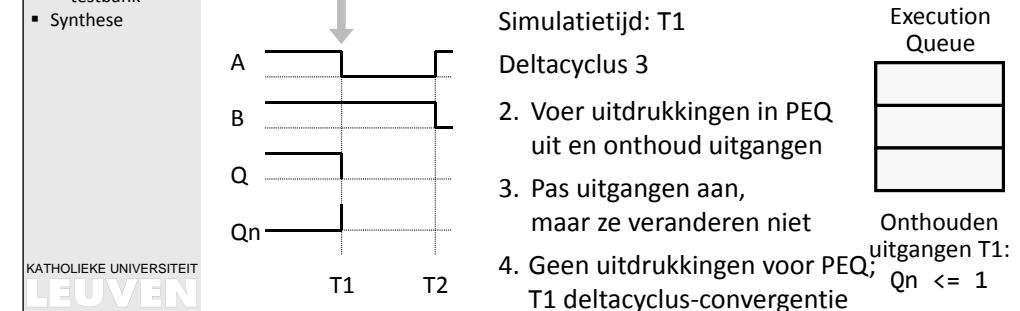
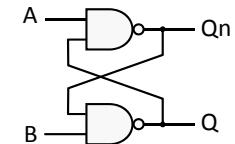


- Proces
- ⇒ Simulatie
  - ⇒ gebeurtenis gedreven
  - beschrijving tijds gedrag
  - testbank
- Synthese

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```

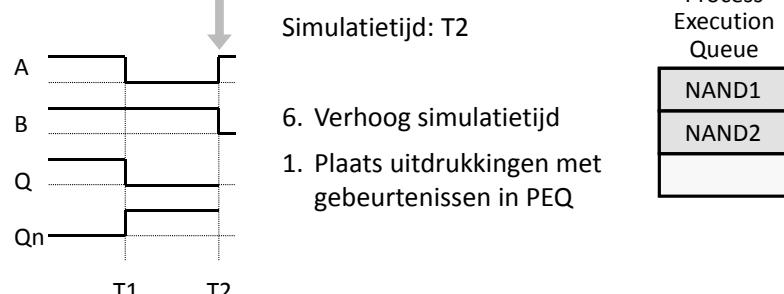
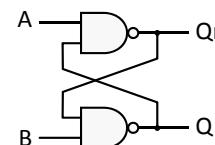


- Proces
- ⇒ Simulatie
  - ⇒ gebeurtenis gedreven
  - beschrijving tijds gedrag
  - testbank
- Synthese

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```

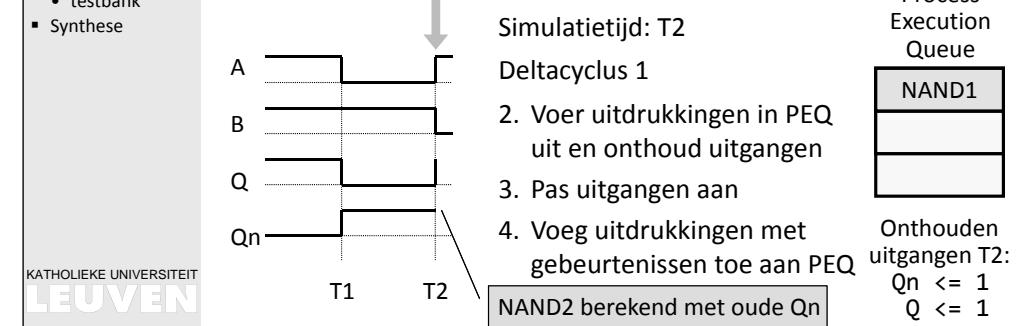
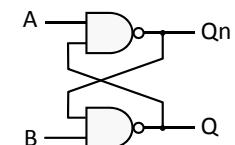


- Proces
- ⇒ Simulatie
  - ⇒ gebeurtenis gedreven
  - beschrijving tijds gedrag
  - testbank
- Synthese

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

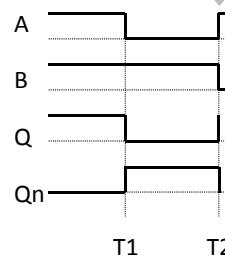
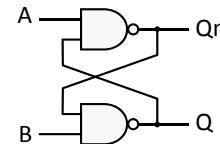
architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```

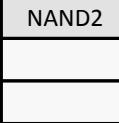


Simulatietijd: T2

Deltacyclus 2

2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan
4. Voeg uitdrukkingen met gebeurtenissen toe aan PEQ

Process Execution Queue

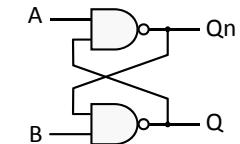


Onthouden uitgangen T2:  
 $Qn \leq 0$

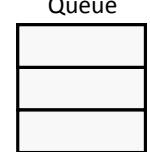
# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;
```

```
architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



Process Execution Queue

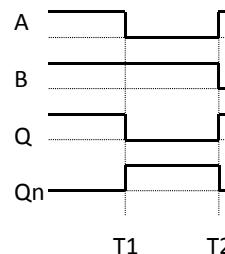
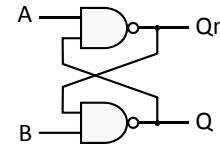


Onthouden uitgangen T2:  
 $Q \leq 1$

# Simulatie van een SR-latch

```
entity Latch is
 port (A,B: in std_logic;
 Q,Qn: buffer std_logic);
end entity Latch;

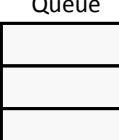
architecture Struct of Latch is
begin
 NAND2: entity NAND2 port map (Qn,B,Q);
 NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



Simulatietijd: T3

6. Verhoog simulatietijd

Process Execution Queue



# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- ➔ VHDL voor synthese en simulatie
  - Het “process” met meer mogelijkheden
  - ➔ Simulatie met VHDL
    - Gebeurtenis gedreven simulatie
    - ➔ Beschrijving tijds gedrag
    - Testbank
  - Synthese: VHDL omzetten in hardware

# Golfvorm ('waveform')

- = wat toegekend wordt aan een signaal  
 $[delay\_mechanism] \ expression \ [after \ a\_time]$   
 $[, \ expression \ [after \ a\_time]]...$

- Transacties worden gepland
  - met waarde = *expression*
  - op tijdstip = *a\_time* + huidige simulatietijd  
(default *a\_time* = 0 fs)
  - NAND-poort met 10 ns vertraging  
 $y \leq a \text{ nand } b \text{ after } 10 \text{ ns};$
  - 20 ns brede resetpuls na 5 ns  
 $\text{rst} \leq '1' \text{ after } 5 \text{ ns}, '0' \text{ after } 25 \text{ ns};$
- *delay\_mechanism* geldt enkel voor het eerste element; de andere hebben altijd een transportvertraging

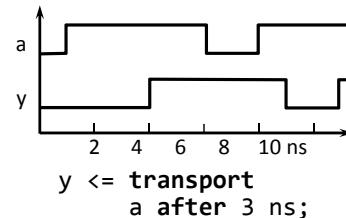
# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouwblokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- ➔ VHDL voor synthese en simulatie
  - Het "process" met meer mogelijkheden
  - ➔ Simulatie met VHDL
    - Gebeurtenis gedreven simulatie
    - Beschrijving tijds gedrag
    - ➔ Testbank
  - Synthese: VHDL omzetten in hardware

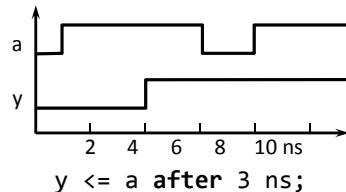
# Vertragings mechanismen

- Transportvertraging  
transport
  - uitgang is vertraagde ingang

- Inertievertraging  
 $[[\text{reject } \text{ reject time}] \text{ inertial}]$ 
  - inertie t.g.v. capaciteit/inductantie
  - ⇒ ( $\text{pulses} < \text{reject time}$ ) verdwijnen
  - (default *reject\_time* = *a\_time*)

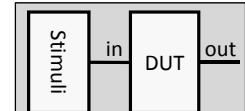


default =  
inertievertraging



# Testbank

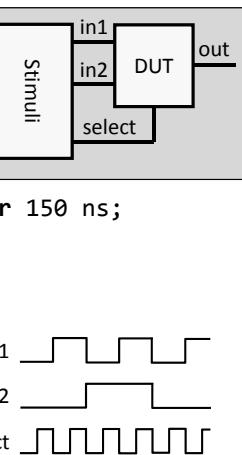
- Hoe wordt een schakeling getest?
  - We leggen aan de ingangen representatieve stimuli aan
  - en controleren of de uitgangen de correcte waarden op het juiste ogenblik vertonen.
- Een VHDL "testbank" is het hoogste hiërarchisch niveau van een ontwerp
  - Het creëert een instantie van het 'Design Under Test',
  - voert stimuli toe aan de ingangen van DUT,
  - controleert de uitgangen ervan door ze
    - te analyseren,  
met bijv. "assertion"- of "report"-uitdrukkingen
    - als golfvorm te visualiseren
  - Vermits dit het hoogste niveau is, heeft het zelf geen ingangen of uitgangen!



# Voorbeeld: MUX-testbank

```
entity Testbench is
end entity Testbench;
```

```
architecture BehavTest of Testbench is
 signal in1, in2, select, out: bit;
begin
 DUT: entity MUX2(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;
```



# Synthese van VHDL

Een syntheseprogramma ('hardware compiler') zet de VHDL-beschrijving om in een structurele beschrijving op lager niveau (poorten/cellen)

- RTL-synthese goed ondersteund
- Synthese van hoger niveau (nog altijd?) te complex voor de meeste programma's
- Programma's verschillen in de subsets van VHDL die ze aankunnen
  - IEEE 1076.6 standaard voor VHDL RTL-synthese = grootste gemene deler, bijv. in de 1999-versie is alleen VHDL-87 toegelaten
- Informatie i.v.m. tijds gedrag wordt genegeerd
  - Een *waveform* kan enkel een *expression* zijn: *delay\_mechanism* of **after** is niet toegelaten
  - Een **wait for** wordt genegeerd

# Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouw blokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- ➔ VHDL voor synthese en simulatie
  - Het "process" met meer mogelijkheden
  - Simulatie met VHDL
- ➔ Synthese: VHDL omzetten in hardware
  - Synthetiseerbare VHDL
  - VHDL-synthese verbeteren
  - Vertaling naar een ASM-schema
  - Synthese-aspecten voor Xilinx

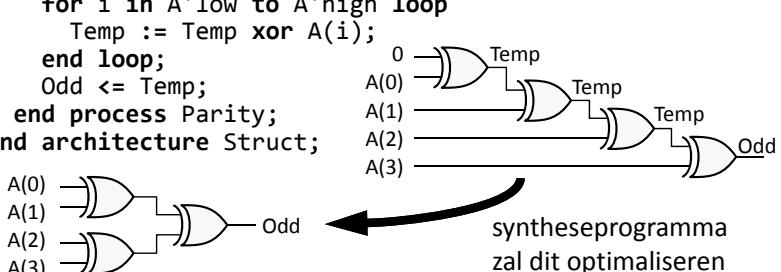
# Een combinatorisch voorbeeld

```
entity Parity is
 generic (n: integer);
 port (A: in std_logic_vector (0 to n-1);
 Odd: out std_logic);
end entity Parity;
```

n moet gekend zijn voor de synthese kan beginnen!

```
architecture Struct of Parity is
begin
```

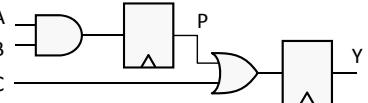
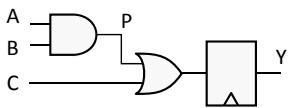
```
 Parity: process(A) is
 variable Temp: std_logic;
 begin
 Temp := '0';
 for i in A'low to A'high loop
 Temp := Temp xor A(i);
 end loop;
 Odd <= Temp;
 end process Parity;
end architecture Struct;
```



syntheseprogramma zal dit optimaliseren

## Een sequentieel voorbeeld: opgelet voor verschil signaal/variabele

- **process (clk) is**  
**variable P: std\_logic;**  
**begin**  
**if rising\_edge(clk) then**  
**P := A and B;** } ≡  $Y \leq (A \text{ and } B) \text{ or } C;$   
**Y <= P or C;**  
**end if;**  
**end process;**
  
- **process (clk) is**  
**signal P: std\_logic;**  
**begin**  
**if rising\_edge(clk) then**  
**P <= A and B;**  
**Y <= P or C;**  
**end if;**  
**end process;**



## Enkele andere beperkingen

- Geen initiële waarde voor signalen
- Enkel for-lussen toegelaten
  - om een lus te kunnen ontvouwen moet # iteraties gekend zijn
- Sequentiële schakelingen
  - Flankgevoelige synchrone: enkel een proces met één van
    - **if (Clk'event and Clk='1') then ...**
    - **wait until Clk='1';**
 Ander gebruik van "wait" niet toegelaten!
  - Niveaugevoelige synchrone: minder ondersteund
  - Asynchrone: niet ondersteund

## Toegelaten datatypes

- Hardware bits: bit, boolean, std\_logic
  - toegelaten waarden std\_logic: '1'/'H', '0'/'L' en 'Z'
    - 'Z' genereert een 3-state buffer
    - Y <= A when Enable else 'Z';
- Gegroepeerde hardware bits:
  - integer & subtypes
  - type addr is range -64 to 63;
  - ⇒ 2-complement, 7 bits
  - opsommingen (ook als gedefinieerd door gebruiker)
- Vectoren van bovenstaande

## Niet-programmeerbare processoren

- Beschrijving van een algoritme
- Extra geheugen bouw blokken
- Synthese: naar een minimale hardware
- Aandachtspunten qua tijds gedrag
- ➔ VHDL voor synthese en simulatie
  - Het "process" met meer mogelijkheden
  - Simulatie met VHDL
- ➔ Synthese: VHDL omzetten in hardware
  - Synthetiseerbare VHDL
  - ➔ VHDL-synthese verbeteren
  - Vertaling naar een ASM-schema
  - Synthese aspecten voor Xilinx

# VHDL-synthese verbeteren

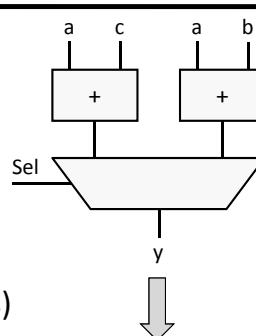
Code herschrijven kan het resultaat na synthese sterk beïnvloeden:

- Een programma kan maar proberen te begrijpen wat met de code bedoeld werd: wat is essentieel en wat is een gevolg van de schrijfstijl?
  - Mag hier een FF i.p.v. een latch gebruikt worden?
- Een programma kan zich niet bewust zijn van alle mogelijke implementaties
  - Wat is de meest optimale toestandscodering?
- Een programma kan niet alle reële beperkingen in rekening brengen
  - Vermogen, grootte, fan-out, tijds gedrag (slechts te schatten), ...
- De auteur kan verkeerde veronderstellingen maken i.v.m. de beschikbare hardware
  - Gebruik van een asynchrone set die niet aanwezig is
- ⇒ De mogelijkheden van het syntheseprogramma **en** de schrijfstijl bepalen het uiteindelijke resultaat!

# 'Resource sharing'

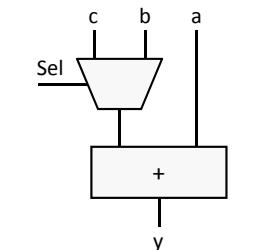
- Originele code:
 

```
if Sel = '1' then
 y <= a + b;
else
 y <= a + c;
end if;
```



- Sommige programma's kunnen dit (meestal slechts binnen een proces) omvormen tot

```
if Sel = '1' then
 x := b;
else
 x := c;
end if;
y <= a + x;
```

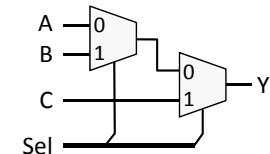


- Anders moet je zelf de code herschrijven!

# Conditionele toekenningen

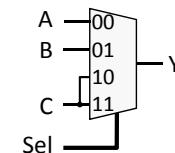
- **if**-uitdrukking of toekenning van conditionele signalen heeft een ingebouwde prioriteit

```
> Y <= C when Sel[1]='1' else
 B when Sel[0]='1' else
 A;
```



- **case**-uitdrukking of toekenning van geselecteerde signalen resulteert meestal in eenvoudigere hardware

```
> with Sel select
 Y <= A when "00",
 B when "01",
 C when others;
```



# Niet-programmeerbare processoren

- Beschrijving van een algoritme

- Extra geheugen bouwblokken

- Synthese: naar een minimale hardware

- Aandachtspunten qua tijds gedrag

- ➔ VHDL voor synthese en simulatie

- Het "process" met meer mogelijkheden

- Simulatie met VHDL

- ➔ Synthese: VHDL omzetten in hardware

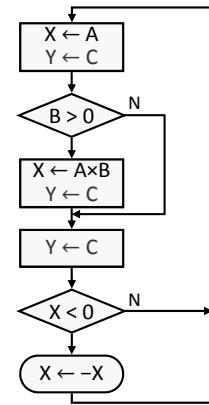
- Synthetiseerbare VHDL
- VHDL-synthese verbeteren
- ➔ Vertaling naar een ASM-schema
- Synthese-aspecten voor Xilinx

Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

## Vertaling naar een ASM-schema

- 1 ASM-blok bevat alles van 1 toestand
  - 1 toestand = alles in een proces tussen opeenvolgende "wait until clk='1';"
  - 1 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;
```



Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

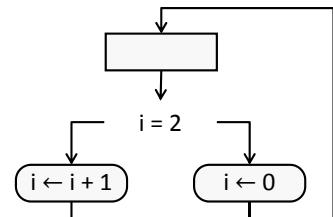
## Vertaling naar een ASM-schema

- ASM-variabele (= een register)
  - = VHDL-variabele of een signaal, waarvan de waarde langer dan 1 klokcyclus blijft bestaan
  - Verschil VHDL-variabele ↔ signaal:

```
process begin
 wait until Clk='1';
 i := i + 1;
 if i = 3 then
 i := 0; end if;
end process;
```

≡

```
process begin
 wait until Clk='1';
 i <= i + 1;
 if i = 2 then
 i <= 0; end if;
end process;
```

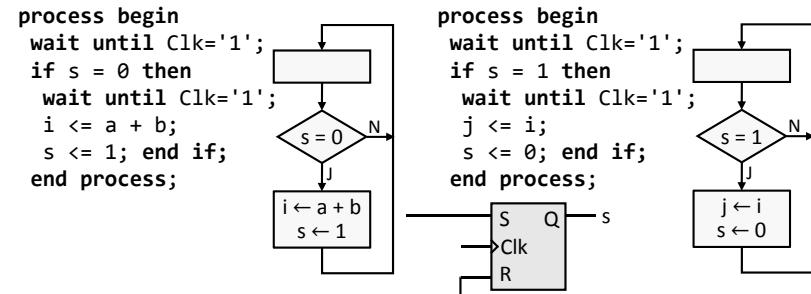


Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

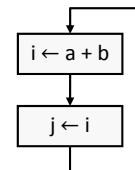
## Vertaling naar een ASM-schema

- Dikwijls 1 ASM/proces eenvoudiger
  - = aparte (gekoppelde) hardware (controller & datapad)

```
process begin
 wait until Clk='1';
 if s = 0 then
 wait until Clk='1';
 i <= a + b;
 s <= 1; end if;
 end process;
```



- Soms 1 globale ASM efficiënter
  - bijv. door integratie  
verdwijnt koppelingsvariabele



Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

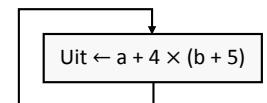
Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

## Vertaling naar een ASM-schema

- VHDL-variabele/signaal is tussenresultaat
  - als de waarde niet langer dan 1 klokcyclus bestaat
  - kan geëlimineerd worden

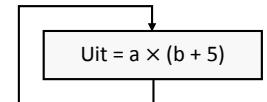
```
process begin
 wait until Clk='1';
 { i := b + 5; i := 4*i;
 Uit <= a + i;
 end process;
```

$$\rightarrow \equiv \text{Uit} <= a + 4 * (b + 5);$$



```
process(a,b,j) begin
 j <= b + 5;
 Uit <= a*j;
 end process;
```

$$\rightarrow \equiv \text{Uit} <= a * (b + 5);$$



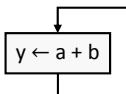
Beschrijving  
Geheugen  
Synthese  
Tijds gedrag  
VHDL  
▪ Proces  
▪ Simulatie  
⇒ Synthese  
• synthetiseerbare VHDL  
• verbeteren  
⇒ naar ASM  
• Xilinx

# Vertaling naar een ASM-schema

## □ Asynchrone signalen

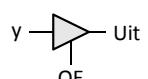
- Soms te combineren met een register

- ```
x <= a + b;
process begin
    wait until Clk='1';
    y <= x;
end process;
```

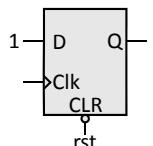
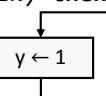


- Anders bijkomende info buiten het ASM-schema

- ```
Uit <= y when OE='1'
else 'Z';
```



- ```
process(rst,clk) begin
    if rst='0' then
        y <= '0';
    elsif rising_edge(clk) then
        y <= '1';
    end if;
end process;
```

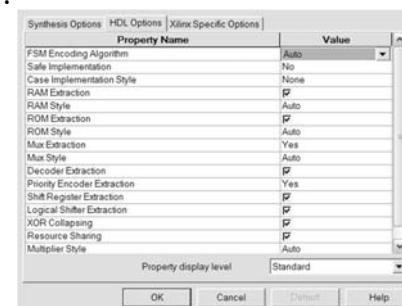


Beschrijving
Geheugen
Synthese
Tijds gedrag
VHDL
▪ Proces
▪ Simulatie
⇒ Synthese
• synthetiseerbare VHDL
• verbeteren
• naar ASM
⇒ Xilinx

Xilinx-specifieke aspecten

□ Sommige beperkingen zijn gekend, andere zijn vast te leggen:

- Automatische synthese van o.a. klok buffers (\Leftarrow fan-out)



- Dikwijls is de default codering one-hot omdat dit overeen komt met de LB-structuur; de codering kan in VHDL ook aangegeven worden met het attribuut enum_encoding

□ Extra componenten:

- Extra hardware (vermenigvuldiger, ...)
- LogiCORE-modules, inclusief RAM
- I/O-buffer, eventueel met pull-up/down weerstand



Beschrijving
Geheugen
Synthese
Tijds gedrag
VHDL
▪ Proces
▪ Simulatie
⇒ Synthese
• synthetiseerbare VHDL
• verbeteren
⇒ naar ASM
• Xilinx

Niet-programmeerbare processoren

□ Beschrijving van een algoritme

□ Extra geheugen bouwblokken

□ Synthese: naar een minimale hardware

□ Aandachtspunten qua tijds gedrag

➔ VHDL voor synthese en simulatie

- Het "process" met meer mogelijkheden

- Simulatie met VHDL

➔ Synthese: VHDL omzetten in hardware

- Synthetiseerbare VHDL
- VHDL-synthese verbeteren
- Vertaling naar een ASM-schema

➔ Synthese-aspecten voor Xilinx: overdraagbaarheid ↔ performantie

Inhoudstafel

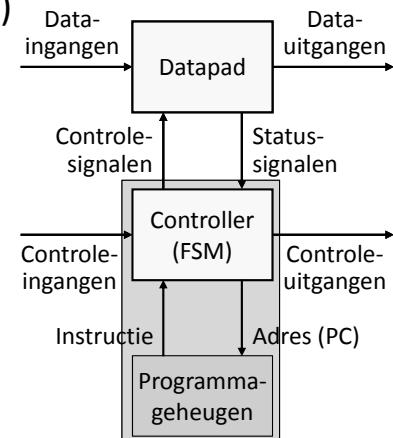
- Inleiding
- De basis van digitaal ontwerp
- Technologische randvoorwaarden
- Combinatorische schakelingen
- Sequentiële schakelingen
- Niet-programmeerbare processoren
- ➔ Programmeerbare processoren

Ontwerp programmeerbare processoren

- Het programma
 - Instructies
 - Adresseermodi
- Processorontwerp
 - CISC-processoren
 - RISC-processoren

Programmeerbare processor

- = FSMD met een programmeerbare controller
- vaste controller (FSMD)
 - andere FSM voor elk algoritme
- generische controller:
 - leest acties uit programmageheugen
 - ander programma voor elk algoritme
 - vaste FSM



Instructies

- Programma
 - = opeenvolging van instructies
 - bevat dezelfde informatie als een FSM
- Instructie
 - komt overeen met een FSM-toestand,
maar kan eventueel meerder klokcycli duren
 - = aanduiding
 - wat volgende instructie is
 - aansturing datapad (registers, FU, MUX, ...)
 - data-uitwisseling (laden/stockeren) met (extern) geheugen
 - typische notaties:
 - mnemonisch (zoals in assembleertaal):
Add A, B, C ($\equiv a=b+c$)
 - acties (zoals bij talen voor hardwarespecificatie):
Mem[A] \leftarrow Mem[B] + Mem[C]

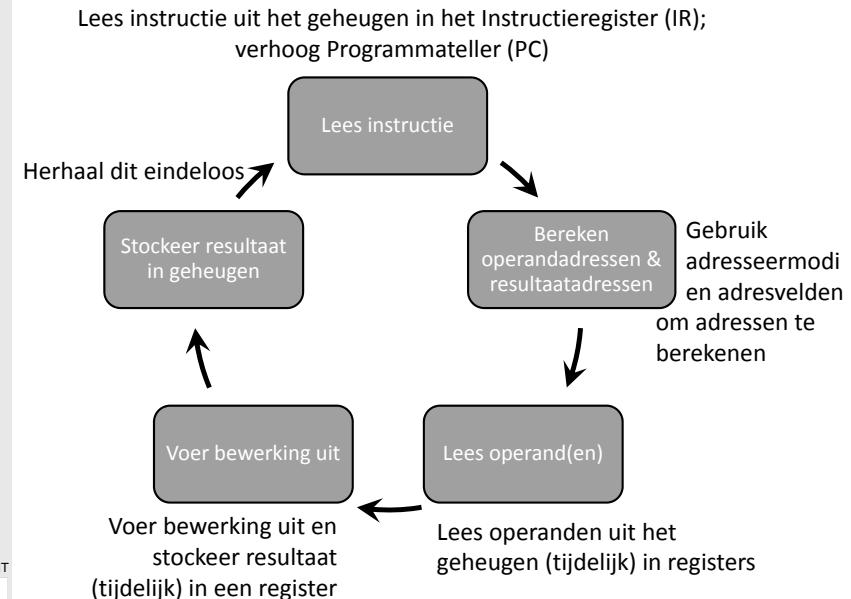
Instructieformaat

- Instructies onderverdeeld in velden
 - = groep bits waaraan een betekenis/subtaak kan toegekend worden
- Veel voorkomende instructievelden:
 - > instructietype: klasse van instructie (registerbewerkingen, sprong, geheugenoperatie, ...)
 - > opcode ('operation code'): welke bewerking (optelling, ...)
 - > adres: locatie operanden & resultaten
 - kan zowel in register(bank) als geheugen zijn bijv. "Load R2, A" ($RF[2] \leftarrow Mem[A]$)
 - > adresseermode: hoe effectief adres berekenen uitgaand van adresveld(en)
 - > constante: bijv. "Add R2, R3, #1"
- Zowel het aantal als de grootte van de velden kunnen variabel zijn
- Sommige soorten velden kunnen meerdere keren voorkomen (bijv. adres)

Uitvoeringssnelheid

- Uitvoeringssnelheid instructie hangt af van
 - > snelheid datapad ⇒ te verhogen door extra hardware
 - > aantal toegangen tot extern geheugen ⇒ korte instructies
- Lengte instructie hoofdzakelijk bepaald door het aantal adresvelden/instructie
 - > Weinig: korte (snelle) instructies ⇔ veel instructies/taak
 - > Veel: lange (trage) instructies ⇔ minder instructies/taak
- Voorbeeld: bereken $c = (a + b) \times (a - b)$
 - > Stel
 - we gebruiken 32-bit data en eveneens 32-bit adressen
 - een instructie bestaat uit
 - 1 woord met alle velden behalve geheugenadressen
 - 1 woord per geheugenadresveld
 - > Het aantal geheugentoegangen is dan
 - $n+1$ (halen instructie met n geheugenadresvelden)
 - +1 per lees/schrijfoperatie data ↔ geheugen

Generische instructiecyclus



$c = (a + b) \times (a - b)$: impact # adresvelden

- Instructies met 3 adressen
 - > Add X,A,B $Mem[X] \leftarrow Mem[A] + Mem[B]$
 - > Sub C,A,B $Mem[C] \leftarrow Mem[A] - Mem[B]$
 - > Mul C,X,C $Mem[C] \leftarrow Mem[X] \times Mem[C]$
 - > (4 + 3) geheugentoegangen/instructie
⇒ programma heeft 21 toegangen nodig
- Instructies met 2 adressen
 - = resultaat overschrijft eerste operand bij operaties met 2 bronnen
⇒ verplaatsinstructies ('move') nodig
 - > Move X,A $Mem[X] \leftarrow Mem[A]$
 - > Add X,B $Mem[X] \leftarrow Mem[X] + Mem[B]$
 - > Move C,A $Mem[C] \leftarrow Mem[A]$
 - > Sub C,B $Mem[C] \leftarrow Mem[C] - Mem[B]$
 - > Mul C,X $Mem[C] \leftarrow Mem[C] \times Mem[X]$
 - > (3 + (2 of 3)) geheugentoegangen/instructie
⇒ programma heeft 28 toegangen nodig
 - 1,67 meer instructies maar instructies zijn sneller (5,6 geheugentoegangen/instructie)

$c=(a+b) \times (a-b)$: impact # adresvelden

□ Instructies met 1 adres

= gebruik speciaal register ACC (Accumulator), waarin de eerste operand en het resultaat geplaatst worden

| | |
|----------|------------------------------------|
| > Load A | $ACC \leftarrow Mem[A]$ |
| Add B | $ACC \leftarrow ACC + Mem[B]$ |
| Store X | $Mem[X] \leftarrow ACC$ |
| Load A | $ACC \leftarrow Mem[A]$ |
| Sub B | $ACC \leftarrow ACC - Mem[B]$ |
| Mul X | $ACC \leftarrow ACC \times Mem[X]$ |
| Store C | $Mem[C] \leftarrow ACC$ |

- > (2 + 1) geheugentoegangen/instructie
- ⇒ programma heeft 21 toegangen nodig

$c=(a+b) \times (a-b)$: impact # adresvelden

□ Instructies zonder adres

= gebruik een LIFO-stapel voor data

- Beide operanden staan bovenaan stapel
- Bewerking vervangt ze door het resultaat
- Verplaatsinstructies gebruiken relatieve adressering

| | |
|----------------|-------------------------------------|
| > Load OffsetA | $Push \leftarrow Mem[base+OffsetA]$ |
| Load OffsetB | $Push \leftarrow Mem[base+OffsetB]$ |
| Add | $Push \leftarrow Pop + Pop$ |
| Load OffsetA | $Push \leftarrow Mem[base+OffsetA]$ |
| Load OffsetB | $Push \leftarrow Mem[base+OffsetB]$ |
| Sub | $Push \leftarrow Pop - Pop$ |
| Mul | $Push \leftarrow Pop \times Pop$ |
| Store OffsetC | $Mem[base+OffsetC] \leftarrow Pop$ |

- > (1 + (0 of 1)) geheugentoegangen/instructie
- ⇒ programma heeft 13 toegangen nodig

$c=(a+b) \times (a-b)$: impact # adresvelden

□ Instructies met 3 adressen, waarvan 2 van een registerbank

> de paar adresbits van een registerbank passen meestal nog in het opcode-woord zodat geen extra adreswoord nodig is

| | |
|-------------|--|
| > Load R1,A | $RF[1] \leftarrow Mem[A]$ |
| Add R2,R1,B | $RF[2] \leftarrow RF[1] + Mem[B]$ |
| Sub R1,R1,B | $RF[1] \leftarrow RF[1] - Mem[B]$ |
| Mul C,R1,R2 | $Mem[C] \leftarrow RF[1] \times RF[2]$ |

- > (2 + 1) geheugentoegangen/instructie
- ⇒ programma heeft 12 toegangen nodig

□ Er zijn verschillende combinaties van register- en geheugenadressering mogelijk, o.a. ...

$c=(a+b) \times (a-b)$: impact # adresvelden

□ Instructies met 3 registeradressen of met 2 adressen, waarvan 1 van een registerbank

= bewerkingen: 3 registeradressen laden/stoken: 1 registeradres & 1 geheugenadres

| | |
|--------------|---------------------------------------|
| > Load R1,A | $RF[1] \leftarrow Mem[A]$ |
| Load R2,B | $RF[2] \leftarrow Mem[B]$ |
| Add R3,R1,R2 | $RF[3] \leftarrow RF[1] + RF[2]$ |
| Sub R4,R1,R2 | $RF[4] \leftarrow RF[1] - RF[2]$ |
| Mul R5,R3,R4 | $RF[5] \leftarrow RF[3] \times RF[4]$ |
| Store C,R5 | $Mem[C] \leftarrow RF[5]$ |

- > (2 + 1) of (1 + 0) geheugentoegangen/instructie
- ⇒ programma heeft 12 toegangen nodig
- > Tijdwinst als de meeste data of tussenresultaten verschillende malen gebruikt wordt, omdat registertoegangen veel sneller zijn dan geheugentoegangen
- > Gebruikt in alle moderne processoren

Ontwerp programmeerbare processoren

→ Het programma

- Instructies

- Adresseermodi

- = hoe effectief adres berekenen uitgaande van adresveld(en)
- ⇒ reduceert dikwijls grootte adresveld
- ⇒ analoog aan hogere programmeertalen (cfr. matrices)

- Processorontwerp

Onmiddellijke adressering

- = het adresveld bevat niet het adres maar de waarde zelf van de operand

- Wordt gebruikt voor constanten



Impliciete adressering

- = het adres zit impliciet in de opcode

Bijv.

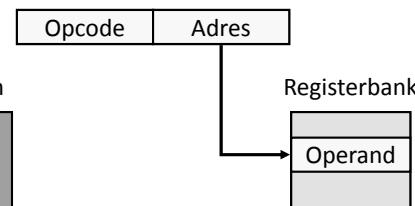
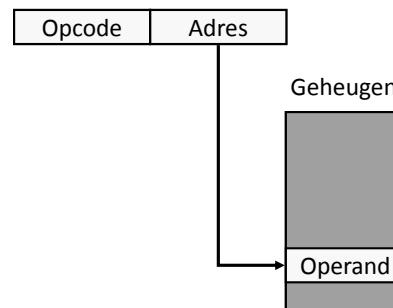
- "ADD" betekent bij een 0-adres machine
"Vervang de twee waarden bovenaan de stapel door hun som"
- "CLRA" betekent bij een 1-adres machine
"Reset de accumulator"

Opcode

Directe adressering

- = het adresveld bevat het adres van de operand
- adres van geheugen (MEM) is dikwijls 32 bit
- adres van een registerbank (RF) is typisch 3 tot 8 bit

⇒ operand = MEM[adres] of RF[adres]



Voordelen registers:
▪ kleiner instructiewoord
▪ snellere toegang

Het programma

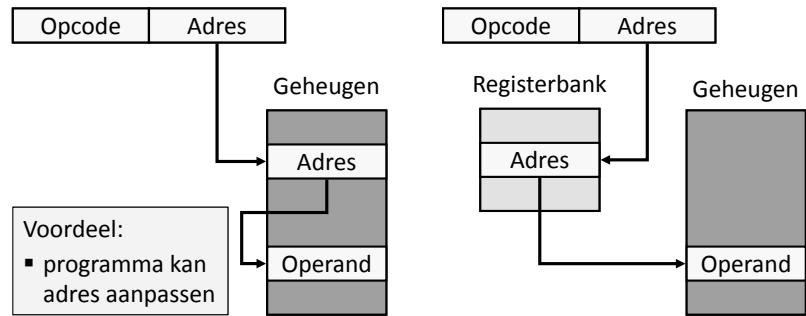
- Instructies
- ⇒ Adresseermodi

Processorontwerp

- CISC
- RISC

Indirecte adressering

- het adresveld verwijst naar de plaats waar het eigenlijke adres van de operand zich bevindt
 - indirect: eigenlijke adres in geheugen
⇒ operand = $\text{MEM}[\text{MEM}[\text{adres}]]$
 - register-indirect: in registerbank
⇒ operand = $\text{MEM}[\text{RF}[\text{adres}]]$



Het programma

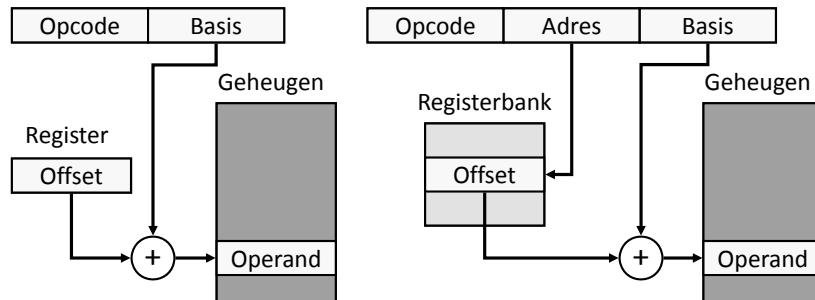
- Instructies
- ⇒ Adresseermodi

Processorontwerp

- CISC
- RISC

Geïndexeerde adressering

- effectieve adres = basisadres + (kleine) offset
 - basis = beginadres matrix, stapel, wachtrij, ...
 - geïndexeerd: offset in impliciet register
 - adresveld = basis
 - register-geïndexeerd:
 - offset in expliciet register ($\text{offset} = \text{RF}[\text{adres}]$)
 - adresveld = (registeradres; basis)



Het programma

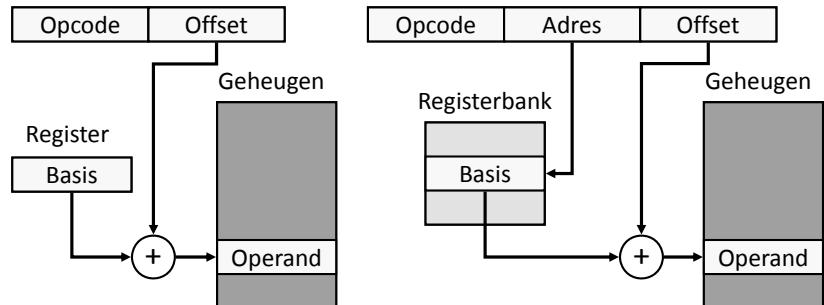
- Instructies
- ⇒ Adresseermodi

Processorontwerp

- CISC
- RISC

Relatieve adressering

- effectieve adres = basisadres + (kleine) offset
⇒ operand = $\text{MEM}[\text{basis} + \text{offset}]$
 - relatief: basisadres in impliciet register
 - adresveld = offset
 - bijv. voor sprongadres (basis = PC)
 - register-relatief: in expliciet register (basis = $\text{RF}[\text{adres}]$)
 - adresveld = (registeradres; offset)
 - voor vectoradres, bijv. opzoektabel ('Look-Up Table')



Het programma

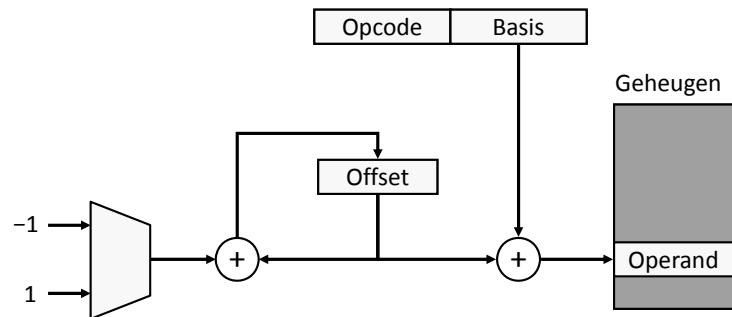
- Instructies
- ⇒ Adresseermodi

Processorontwerp

- CISC
- RISC

Geïndexeerde adressering met autoincrement/autodecrement

- geïndexeerde adressering waarbij de offset automatisch aangepast wordt
 - meestal ± 1 maar soms ook $\pm 2, \pm 4, \dots$



Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- RISC

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

- Complex Instruction Set Computer
- Reduced Instruction Set Computer

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- RISC

Basiskeuze ontwerp instructieset

□ CISC: Complex Instruction Set Computer

- = vele complexe (trage) instructies
 - meerdere operaties, instructietypes, adresseermodi en adresvelden
 - complex datapad met veel FU, veel registers en complexe verbindingen ⇒ lage klokfrequentie
 - kort programma, maar een instructie kan veel klokcycli verbruiken

□ RISC: Reduced Instruction Set Computer

- = enkele snelle (eenvoudige) instructies
 - weinig operaties en instructietypes; 0 of 1 adresvelden; weinig adresmodi
 - eenvoudig datapad ⇒ hoge klokfrequentie
 - lang programma, maar elke instructie verbruikt maar een paar klokcycli

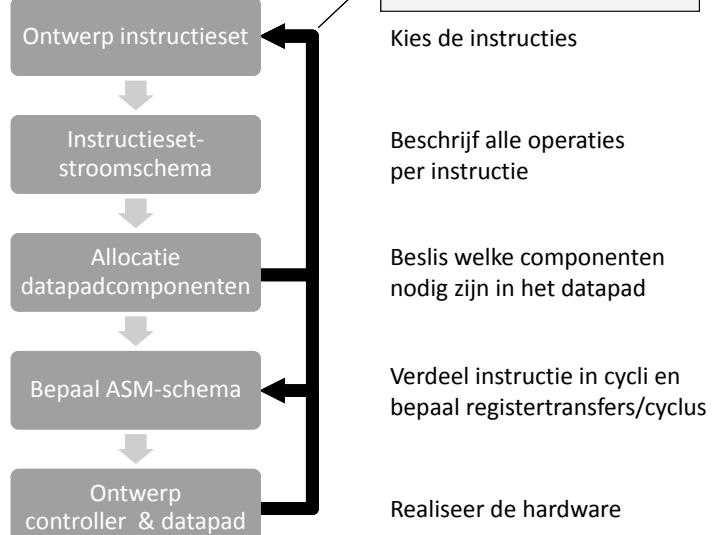
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- RISC

Ontwerp van een programmeerbare processor



Datapad hangt af van instructieset en vice versa

Kies de instructies

Beschrijf alle operaties per instructie

Beslis welke componenten nodig zijn in het datapad

Verdeel instructie in cycli en bepaal registertransfers/cyclus

Realiseer de hardware

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
- RISC

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
- RISC

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

→ Complex Instruction Set Computer

- Ontwerp instructieset
- Instructieset-stroomschema
- Allocatie datapadcomponenten
- ASM-schema
- Ontwerp controller
- Ontwerp datapad

→ Reduced Instruction Set Computer

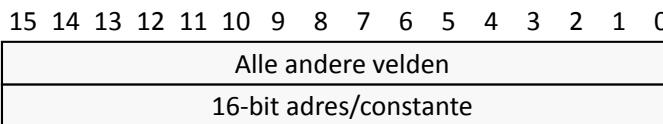
Ontwerp CISC-instructieset

□ Specificaties

- 16-bit woordlengte
- 16-bit adresbereik
- Maximale grootte instructie: twee 16-bit woorden

□ Bijkomende keuzes

- Voor de hand liggend uit specificaties
 - 1 adresveld van 16 bit veld
 - alle andere velden samen ook 16 bit
- We kiezen
 - laden/stockeren registers: 1 16-bit adresveld
 - de rest enkel registeroperaties

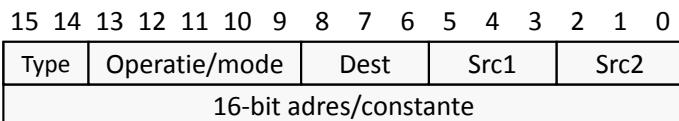


Ontwerp CISC-instructieset

Ontwerp CISC-instructieset

□ Bijkomende keuzes

- We hebben 4 types instructies (2 bits):
 - registerinstructies (bewerkingen met 2 of 3 registers)
 - verplaatsinstructies (1 registeradres & 1 geheugenadres)
 - spronginstructies (bepaal programmavoorloop)
 - andere instructies, zoals NOP
- Hoewel niet alle instructies evenveel registeradressen nodig hebben, voorzien we er 3 voor allemaal
 - We kiezen 3 registeradresvelden van 3 bits
 - Nog 5 bits beschikbaar voor bewerking en adresseermodes ("Operatie/mode")



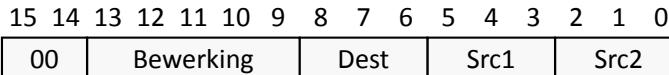
Registerinstructies (type 00)

$$\left\{ \begin{array}{ll} OP\ R_{Dest}, R_{Src1}, R_{Src2} & RF[Dest] \leftarrow RF[Src1] \text{ OP } RF[Src2] \\ OP\ R_{Dest}, R_{Src1} & RF[Dest] \leftarrow RF[Src1] \text{ OP } \end{array} \right.$$

□ Bewerking OP

- Aritmetische, logische en schuifoperaties
 - Keuze bewerkingen hangt af van frequentie gebruik in typische toepassingen
- Op 1 of 2 operanden

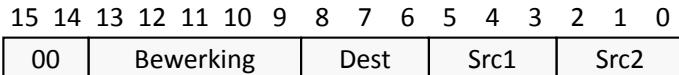
□ Slechts 1 instructiewoord



Registerinstructies (type 00)

□ We kiezen voor ons toepassingsdomein

| 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 |



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

• IS-schema

• Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Registerinstructies (type 00)

| bit ₁₃ | bit ₁₂ | bit ₁₁ | bit ₁₀ | bit ₉ | Bewerking |
|-------------------|-------------------|-------------------|-------------------|------------------|-------------------------------------|
| 1 | 0 | | | | Aritmetische bewerking |
| | 0 | | | | Optelling/aftrekking |
| | | 0 | | | 2 bronregisters |
| | | | 0 | | Add: RF[Dest] ← RF[Src1] + RF[Src2] |
| | | | 1 | | Sub: RF[Dest] ← RF[Src1] - RF[Src2] |
| | | 1 | | | 1 bronregister |
| | | | 0 | | Inc: RF[Dest] ← RF[Src1] + 1 |
| | | | 1 | | Dec: RF[Dest] ← RF[Src1] - 1 |
| | 1 | | | | Vermenigvuldiging/deling |
| | | 0 | | | 2 bronregisters |
| | | | 0 | | Mul: RF[Dest] ← RF[Src1] × RF[Src2] |
| | | | 1 | | Div: RF[Dest] ← RF[Src1] / RF[Src2] |
| | | 1 | | | 1 bronregister |
| | | | 0 | | Root: RF[Dest] ← SQRT(RF[Src1]) |
| | | | 1 | | Negate: RF[Dest] ← - RF[Src1] |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

• IS-schema

• Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Verplaatsinstructies (type 01)

- = uitwisseling tussen geheugen en registerbank
- Slechts 2 operaties mogelijk ⇒ Op is 1 bit
 - Load (Op = 0) : geheugen → register
 - Store (Op = 1) : register → geheugen
- Overblijvende 4 bits worden gebruikt voor de adresseermode van het geheugen
 - Keuze adresseermode hangt af van frequentie gebruik in typische toepassingen
 - Src2 kan eventueel gebruikt worden als offset
 - Adresseermode zal ook bepalen of tweede woord (adres/constante) al dan niet gebruikt wordt

| | | | | | | | | | | | | | | | |
|------------------------|----|------|----|------|----|------|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 01 | Op | Mode | | Dest | | Src1 | | Src2 | | | | | | | |
| 16-bit adres/constante | | | | | | | | | | | | | | | |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

• IS-schema

• Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Registerinstructies (type 00)

| 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 0001 ₁₆ |
| | | 1 | 1 | 1 | Inv: RF[Dest] ← INV(RF[Src1]) |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

• IS-schema

• Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Load (type 01, op = 0)

- Onmiddellijk adres (2 woorden):
RF[Dest] ← Constante
- Direct adres (2 woorden):
RF[Dest] ← Mem[Adres]
- Indirect adres (2 woorden):
RF[Dest] ← Mem[Mem[Adres]]
- Register-indirect adres (1 woord):
RF[Dest] ← Mem[RF[Src1]]
- Register-relatief adres (1 woord):
RF[Dest] ← Mem[RF[Src1] + Src2] =
RF[Dest] ← Mem[Basis + Offset]
- Register-geïndexeerd adres (2 woorden):
RF[Dest] ← Mem[Adres + RF[Src1]] =
RF[Dest] ← Mem[Basis + Offset]
- Kopieer register (1 woord):
RF[Dest] ← RF[Src1]

Store (type 01, op = 1)

- Onmiddellijk adres: zinloos!
- Direct adres (2 woorden):
 $\text{Mem}[\text{Adres}] \leftarrow \text{RF}[\text{Src1}]$
- Indirect adres (2 woorden):
 $\text{Mem}[\text{Mem}[\text{Adres}]] \leftarrow \text{RF}[\text{Src1}]$
- Register-indirect adres (1 woord):
 $\text{Mem}[\text{RF}[\text{Dest}]] \leftarrow \text{RF}[\text{Src1}]$
- Register-relatief adres (1 woord):
 $\text{Mem}[\text{RF}[\text{Dest}] + \text{Src2}] \leftarrow \text{RF}[\text{Src1}] =$
 $\text{Mem}[\text{Basis} + \text{Offset}] \leftarrow \text{RF}[\text{Src1}]$
- Register-geïndexeerd adres (2 woorden):
 $\text{Mem}[\text{Adres} + \text{RF}[\text{Dest}]] \leftarrow \text{RF}[\text{Src1}] =$
 $\text{Mem}[\text{Basis} + \text{Offset}] \leftarrow \text{RF}[\text{Src1}]$

Spronginstructies (type 10)

4 instructies (Op gebruikt enkel bits 13 en 12):

- JMP (Op = 00): onconditionele sprong (2 woorden)
 $\text{PC} \leftarrow \text{Adres}$
- CJMP (Op = 01): conditionele sprong (2 woorden)
 Als Status = 0 dan ($\text{PC} \leftarrow \text{PC} + 1$) anders ($\text{PC} \leftarrow \text{Adres}$)
- JSR (Op = 10): spring naar subroutine (2 woorden)
 $\text{Mem}[\text{RF}[\text{Src1}]] \leftarrow \text{PC} + 1$ terugkeeradres op LIFO met
 $\text{RF}[\text{Src1}]$ het LIFO-schrijfadres
 en ($\text{RF}[\text{Src1}] - 1$) het leesadres
 $\text{RF}[\text{Src1}] \leftarrow \text{RF}[\text{Src1}] + 1$ verhoog LIFO-schrijfadres
 $\text{PC} \leftarrow \text{Adres}$ spring naar subroutine
- RTS (Op = 11): keer terug uit subroutine (1 woord)
 $\text{RF}[\text{Src1}] \leftarrow \text{RF}[\text{Src1}] - 1$ verminder LIFO-schrijfadres;
 $\text{RF}[\text{Src1}]$ bevat nu ook het
 LIFO-adres van het terugkeeradres
 $\text{PC} \leftarrow \text{Mem}[\text{RF}[\text{Src1}]]$ lees terugkeeradres uit LIFO

Verplaatsinstructies (type 01)

- Toewijzing bits:

- Bit 13:
 0 = Load
 1 = Store
- Bit 12:
 0 = twee woorden (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)

Niet alle combinaties zijn geldig!
 Bijv. "01000" of "10000".

Andere (1-woord) instructies (type 11)

| bit ₁₃ | bit ₁₂ | bit ₁₁ | bit ₁₀ | bit ₉ | Bewerking |
|-------------------|-------------------|-------------------|-------------------|------------------|--|
| 0 | 0 | - | - | - | NOP (doe niets) |
| 0 | 1 | | | | Zet op 0 |
| | | 0 | - | - | CLR: $\text{RF}[\text{Dest}] \leftarrow 0$ |
| | | 1 | - | - | ClrS: Status $\leftarrow 0$ |
| 1 | 0 | | | | Vergelijk $\text{RF}[\text{Src1}]$ en $\text{RF}[\text{Src2}]$ Als ... dan Status $\leftarrow 1$ anders Status $\leftarrow 0$ |
| | | 0 | 0 | 0 | GT: $\text{RF}[\text{Src1}] > \text{RF}[\text{Src2}]$ |
| | | 0 | 0 | 1 | GE: $\text{RF}[\text{Src1}] \geq \text{RF}[\text{Src2}]$ |
| | | 0 | 1 | 0 | LT: $\text{RF}[\text{Src1}] < \text{RF}[\text{Src2}]$ |
| | | 0 | 1 | 1 | LE: $\text{RF}[\text{Src1}] \leq \text{RF}[\text{Src2}]$ |
| | | 1 | 0 | 0 | EQ: $\text{RF}[\text{Src1}] = \text{RF}[\text{Src2}]$ |
| | | 1 | 0 | 1 | NE: $\text{RF}[\text{Src1}] \neq \text{RF}[\text{Src2}]$ |
| | | 1 | 1 | - | Niet gebruikt ⇒ ongeldige instructie |
| 1 | 1 | - | - | - | SetS: Status $\leftarrow 1$ |

Programmeerbare processoren

Het programma
▪ Instructies
▪ Adresseermodi

Processorontwerp

- ⇒ CISC
 - ⇒ IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
 - RISC

Voorstelling instructie

Machinetaal = het binair patroon van een instructie

- Meestal hexadecimaal voorgesteld om schrijffouten te vermijden
bijv. $0010000111110000 = 21F0_{16}$
- Hardware “interpreteert” de binaire instructie
 - 0010000111110000 : registerbewerking
 - 0010000111110000 : aritmatisch
 - 0010000111110000 : optelling
 - 0010000111110000 : resultaat in RF[7]
 - 0010000111110000 : operand 1 = RF[6]
 - 0010000111110000 : operand 2 = RF[0]
- Interpretatie = broncode wordt rechtstreeks uitgevoerd
- De controller is dus een ‘interpreter’ in hardware

Programmeerbare processoren

Het programma
▪ Instructies
▪ Adresseermodi

Processorontwerp

- ⇒ CISC
 - ⇒ IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
 - RISC

Programmeerbare processoren

Het programma
▪ Instructies
▪ Adresseermodi

- Processorontwerp
- ⇒ CISC
 - ⇒ IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
 - RISC

Assembleertaal voor onze CISC

□ 18 registerinstructies

| Assembleertaal | Opcode | D | S1 | S2 | A | Actie |
|---|--|---|----|----|---|---------------------------------------|
| ASR R _D ,R _{S1} ,#n | 0000n ₂ n ₁ n ₀ | ✓ | ✓ | – | / | RF[D] ← RF[S1] >> n |
| ASL R _D ,R _{S1} ,#n | 0001n ₂ n ₁ n ₀ | ✓ | ✓ | – | / | RF[D] ← RF[S1] << n |
| ADD R _D ,R _{S1} ,R _{S2} | 0010000 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] + RF[S2] |
| SUB R _D ,R _{S1} ,R _{S2} | 0010001 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] - RF[S2] |
| INC R _D ,R _{S1} | 0010010 | ✓ | ✓ | – | / | RF[D] ← RF[S1] + 1 |
| DEC R _D ,R _{S1} | 0010011 | ✓ | ✓ | – | / | RF[D] ← RF[S1] - 1 |
| MUL R _D ,R _{S1} ,R _{S2} | 0010100 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] × RF[S2] |
| DIV R _D ,R _{S1} ,R _{S2} | 0010101 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] ÷ RF[S2] |
| ROOT R _D ,R _{S1} | 0010110 | ✓ | ✓ | – | / | RF[D] ← Root(RF[S1]) |
| NEG R _D ,R _{S1} | 0010111 | ✓ | ✓ | – | / | RF[D] ← -RF[S1] |
| AND R _D ,R _{S1} ,R _{S2} | 0011000 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] AND RF[S2] |
| NAND R _D ,R _{S1} ,R _{S2} | 0011001 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] NAND RF[S2] |
| OR R _D ,R _{S1} ,R _{S2} | 0011010 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] OR RF[S2] |
| NOR R _D ,R _{S1} ,R _{S2} | 0011011 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] NOR RF[S2] |
| XOR R _D ,R _{S1} ,R _{S2} | 0011100 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] XOR RF[S2] |
| XNOR R _D ,R _{S1} ,R _{S2} | 0011101 | ✓ | ✓ | ✓ | / | RF[D] ← RF[S1] XNOR RF[S2] |
| MASK R _D ,R _{S1} | 0011110 | ✓ | ✓ | – | / | RF[D] ← RF[S1] AND 0001 ₁₆ |
| INV R _D ,R _{S1} | 0011111 | ✓ | ✓ | – | / | RF[D] ← INV RF[S1] |

Programmeerbare processoren

Het programma
▪ Instructies
▪ Adresseermodi

- Processorontwerp
- ⇒ CISC
 - ⇒ IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
 - RISC

Voorstelling instructie

Assembleertaal = mnemonicische voorstelling

- 1-naar-1 relatie met machinetaal
bijv. $21F0_{16} \equiv \text{ADD } R7, R6, R0$
- Assembleren = compilatie naar machinecode
 - Vertaling van instructies via een eenvoudige opzoektabel
 - Bijkomend gebruiksgemak:
geef variabelen een betekenisvolle naam
 - Assembler kent er zelf een geheugenadres aan toe (relatief t.o.v. begin datageheugen)
 - Bijkomend gebruiksgemak:
geef een betekenisvolle symbolische naam (label) aan adressen waar naartoe gesprongen wordt
 - Assembler berekent zelf het sprongadres (relatief t.o.v. begin programmageheugen)

Assembleertaal voor onze CISC

□ 12 verplaatsinstructies

| Assembleertaal | Opcode | D | S1 | S2 | A | Actie |
|--|---------|---|----|----|---|---------------------------|
| LOAD R _D ,#waarde | 0100000 | ✓ | – | – | ✓ | RF[D] ← waarde |
| LOAD R _D ,adres | 0100001 | ✓ | – | – | ✓ | RF[D] ← Mem[adres] |
| LOAD R _D ,(adres) | 0100010 | ✓ | – | – | ✓ | RF[D] ← Mem[Mem[adres]] |
| LOAD R _D ,(R _{S1}) | 0101010 | ✓ | ✓ | – | / | RF[D] ← Mem[RF[S1]] |
| LOAD R _D ,(R _{S1})+R _{S2} | 0101011 | ✓ | ✓ | ✓ | / | RF[D] ← Mem[RF[S1]+S2] |
| LOAD R _D ,adres+(R _{S1}) | 0101100 | ✓ | ✓ | – | ✓ | RF[D] ← Mem[adres+RF[S1]] |
| STOR adres, R _{S1} | 0110001 | – | ✓ | – | ✓ | Mem[adres] ← RF[S1] |
| STOR (adres), R _{S1} | 0110010 | – | ✓ | – | ✓ | Mem[Mem[adres]] ← RF[S1] |
| STOR (R _D), R _{S1} | 0111010 | ✓ | ✓ | – | / | Mem[RF[D]] ← RF[S1] |
| STOR (R _D)+R _{S2} , R _{S1} | 0111011 | ✓ | ✓ | ✓ | / | Mem[RF[D]+S2] ← RF[S1] |
| STOR adres+(R _D), R _{S1} | 0111100 | ✓ | ✓ | – | ✓ | Mem[adres+RF[D]] ← RF[S1] |
| COPY R _D , R _{S1} | 01--101 | ✓ | ✓ | – | / | RF[D] ← RF[S1] |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- ⇒ IS-ontwerp
- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Assembleertaal voor onze CISC

□ 4 spronginstructies

| Assembleertaal | Opcode | D | S1 | S2 | A | Actie |
|------------------------------|---------|---|----|----|---|--|
| JMP adres | 1000--- | - | - | - | ✓ | PC ← adres |
| CJMP adres | 1001--- | - | - | - | ✓ | als (Status=0) dan (PC ← PC+1) anders (PC ← adres) |
| JSR adres,(R _{S1}) | 1010--- | - | ✓ | - | ✓ | Mem[RF[S1]] ← PC+1; RF[S1] ← RF[S1]+1; PC ← adres |
| RTS (R _{S1}) | 1011--- | - | ✓ | - | / | RF[S1] ← RF[S1]-1; PC ← Mem[RF[S1]] |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- ⇒ IS-ontwerp
- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld

Bepaal de som, het minimum en het maximum van 1024 getallen.

□ Programma in een hoog-niveau taal:

Min = MAXINT

Max = 0

Sum = 0

FOR i: 0..1023

 Sum = Sum + A[i]

 IF A[i] > Max THEN Max = A[i]

 IF A[i] < Min THEN Min = A[i]

END FOR

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- ⇒ IS-ontwerp
- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Assembleertaal voor onze CISC

□ 10 andere instructies

| Assembleertaal | Opcode | D | S1 | S2 | A | Actie |
|--------------------------------------|----------|---|----|----|---|---|
| NOP | 1100--- | - | - | - | / | Niets |
| CLR R _D | 11010--- | ✓ | - | - | / | RF[D] ← 0 |
| ClrS | 11011--- | - | - | - | / | Status ← 0 |
| SetS | 1111--- | - | - | - | / | Status ← 1 |
| GT R _{S1} , R _{S2} | 1110000 | - | ✓ | ✓ | / | als (RF[S1] > RF[S2]) dan (Status ← 1) anders (Status ← 0) |
| GE R _{S1} , R _{S2} | 1110001 | - | ✓ | ✓ | / | als (RF[S1] ≥ RF[S2]) dan (Status ← 1) anders (Status ← 0) |
| LT R _{S1} , R _{S2} | 1110010 | - | ✓ | ✓ | / | als (RF[S1] < RF[S2]) dan (Status ← 1) anders (Status ← 0) |
| LE R _{S1} , R _{S2} | 1110011 | - | ✓ | ✓ | / | als (RF[S1] ≤ RF[S2]) dan (Status ← 1) anders (Status ← 0) |
| EQ R _{S1} , R _{S2} | 1110100 | - | ✓ | ✓ | / | als (RF[S1] = RF[S2]) dan (Status ← 1) anders (Status ← 0) |
| NE R _{S1} , R _{S2} | 1110101 | - | ✓ | ✓ | / | als (RF[S1] ≠ RF[S2]) dan (Status ← 1) anders (Status ← 0) |

⇒ 44 instructies in totaal

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- ⇒ IS-ontwerp
- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

ORG DATA

Hierna volgen declaraties van variabelen waarvoor geheugenruimte moet voorbehouden worden in het datageheugen (vanaf 0)

Min Word
Max Word
Sum Word
A Array[1024]

-- op adres 0000₁₆
-- op adres 0001₁₆
-- op adres 0002₁₆
-- op adressen 0003₁₆ tot 0402₁₆

ORG PROGRAM

Dit geeft het begin van het programma aan.
Dit startadres (0403₁₆) verandert elke keer wanneer een bijkomende variabele gedeclareerd wordt, waardoor alle sprongadressen gewijzigd moeten worden!

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

```
ORG PROGRAM
LOAD R0,#0xffff -- RF[0] ← FFFF16 Min = MAXINT
CLR R1 -- RF[1] ← 0 Max = 0
CLR R2 -- RF[2] ← 0 Sum = 0
CLR R3 -- RF[3] ← 0 i = 0
LOAD R4,#0x03ff -- RF[4] ← 102310 MaxCount
LOAD R5,#A -- RF[5] ← 310 adres A[i]
```



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

```
ORG PROGRAM
LOAD R0,#0xffff -- RF[0] ← FFFF16 Min = MAXINT
CLR R1 -- RF[1] ← 0 Max = 0
CLR R2 -- RF[2] ← 0 Sum = 0
CLR R3 -- RF[3] ← 0 i = 0
LOAD R4,#0x03ff -- RF[4] ← 102310 MaxCount
LOAD R5,#A -- RF[5] ← 310 adres A[i]
BODY: LOAD R6,(R5)
      -- RF[6] ← Mem[RF[5]] A[i]
      ADD R2,R2,R6
      -- RF[2] ← RF[2] + RF[6]
      LE R6,R1
      -- Status ← 1 als RF[6] ≤ RF[1]
      CJMP NEXT
      -- PC ← NEXT als Status = 1
      COPY R1,R6
      -- RF[1] ← RF[6]
NEXT: GE R6,R0
      -- Status ← 1 als RF[6] ≥ RF[0]
      CJMP CTU
      -- PC ← CTU als Status = 1
      COPY R0,R6
      -- RF[0] ← RF[6]
CTU: INC R3,R3
      -- RF[3] ← RF[3] + 1
      INC R5,R5
      -- RF[5] ← RF[5] + 1
      LE R3,R4
      -- Status ← 1 als RF[3] ≤ RF[4]
      CJMP BODY
      -- PC ← BODY als Status = 1
      Test of lus opnieuw moet uitgevoerd worden
      Verhoog lusteller en adres A[i]:
      i = i+1; (adres A[i]) = (adres A[i])+1
```

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema
- Datapad-componenten

• ASM-schema

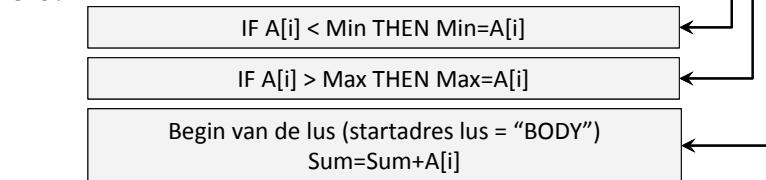
• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

```
ORG PROGRAM
LOAD R0,#0xffff -- RF[0] ← FFFF16 Min = MAXINT
CLR R1 -- RF[1] ← 0 Max = 0
CLR R2 -- RF[2] ← 0 Sum = 0
CLR R3 -- RF[3] ← 0 i = 0
LOAD R4,#0x03ff -- RF[4] ← 102310 MaxCount
LOAD R5,#A -- RF[5] ← 310 adres A[i]
BODY: LOAD R6,(R5)
      -- RF[6] ← Mem[RF[5]] A[i]
      ADD R2,R2,R6
      -- RF[2] ← RF[2] + RF[6]
      LE R6,R1
      -- Status ← 1 als RF[6] ≤ RF[1]
      CJMP NEXT
      -- PC ← NEXT als Status = 1
      COPY R1,R6
      -- RF[1] ← RF[6]
NEXT: GE R6,R0
      -- Status ← 1 als RF[6] ≥ RF[0]
      CJMP CTU
      -- PC ← CTU als Status = 1
      COPY R0,R6
      -- RF[0] ← RF[6]
CTU:
```



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema
- Datapad-componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

```
ORG PROGRAM
LOAD R0,#0xffff -- RF[0] ← FFFF16 Min = MAXINT
CLR R1 -- RF[1] ← 0 Max = 0
CLR R2 -- RF[2] ← 0 Sum = 0
CLR R3 -- RF[3] ← 0 i = 0
LOAD R4,#0x03ff -- RF[4] ← 102310 MaxCount
LOAD R5,#A -- RF[5] ← 310 adres A[i]
BODY: LOAD R6,(R5)
      -- RF[6] ← Mem[RF[5]] A[i]
      ADD R2,R2,R6
      -- RF[2] ← RF[2] + RF[6]
      LE R6,R1
      -- Status ← 1 als RF[6] ≤ RF[1]
      CJMP NEXT
      -- PC ← NEXT als Status = 1
      COPY R1,R6
      -- RF[1] ← RF[6]
NEXT: GE R6,R0
      -- Status ← 1 als RF[6] ≥ RF[0]
      CJMP CTU
      -- PC ← CTU als Status = 1
      COPY R0,R6
      -- RF[0] ← RF[6]
CTU: INC R3,R3
      -- RF[3] ← RF[3] + 1
      INC R5,R5
      -- RF[5] ← RF[5] + 1
      LE R3,R4
      -- Status ← 1 als RF[3] ≤ RF[4]
      CJMP BODY
      -- PC ← BODY als Status = 1
      Stockeer de resultaten
      Mem[Min] ← RF[0]
      Mem[Max] ← RF[1]
      Mem[Sum] ← RF[2]
```

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema

• Datapad-

componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: assembleertaal

```

ORG PROGRAM
LOAD R0,#0xffff -- RF[0] ← FFFF16 Min = MAXINT
CLR R1 -- RF[1] ← 0 Max = 0
CLR R2 -- RF[2] ← 0 Sum = 0
CLR R3 -- RF[3] ← 0 i = 0
LOAD R4,#0x03ff -- RF[4] ← 102310 MaxCount
LOAD R5,#A -- RF[5] ← 310 adres A[i]
BODY: LOAD R6,(R5)
      -- RF[6] ← Mem[RF[5]] A[i]
      ADD R2,R2,R6 -- RF[2] ← RF[2] + RF[6]
      LE R6,R1 -- Status ← 1 als RF[6] ≤ RF[1]
      CJMP NEXT -- PC ← NEXT als Status = 1
      COPY R1,R6 -- RF[1] ← RF[6]
      COPY R1,R6 -- Status ← 1 als RF[6] ≥ RF[0]
      CJMP CTU -- PC ← CTU als Status = 1
      COPY R0,R6 -- RF[0] ← RF[6]
      ...

```

A[i] wordt 1 maal geladen in RF[6] en wordt 5 maal gebruikt:
dit bespaart vermogen en verhoogt de snelheid

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema

• Datapad-

componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Ontwerp programmeerbare processoren

❑ Het programma

➔ Processorontwerp

➔ Complex Instruction Set Computer

- Ontwerp instructieset
- ➔ Instructieset-stroomschema
(‘Instruction Set Flowchart’)
- Allocatie datapadcomponenten
- ASM-schema
- Ontwerp controller
- Ontwerp datapad

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema

• Datapad-

componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Programmavoorbeeld: machinetaal

| | | |
|--------------------|---------------------|--------------------|
| 0403 ₁₆ | 0100 0000 00-- ---- | -- LOAD R0,#0xFFFF |
| 0404 ₁₆ | 1111 1111 1111 1111 | -- CLR R1 |
| 0405 ₁₆ | 1101 0--0 01-- ---- | -- CLR R2 |
| 0406 ₁₆ | 1101 0--0 10-- ---- | -- CLR R3 |
| 0407 ₁₆ | 1101 0--0 11-- ---- | -- LOAD R4,#0x03FF |
| 0408 ₁₆ | 0100 0001 00-- ---- | -- LOAD R5,#A |
| 0409 ₁₆ | 0000 0011 1111 1111 | -- ADD R2,R2,R6 |
| 040A ₁₆ | 0100 0001 01-- ---- | -- LE R6,R1 |
| 040B ₁₆ | 0000 0000 0000 0011 | -- CJMP NEXT |
| 040C ₁₆ | 0101 0101 1010 1--- | -- COPY R1,R6 |
| 040D ₁₆ | 0010 0000 1001 0110 | -- GE R6,R0 |
| 040E ₁₆ | 1110 011- --1 0001 | -- CJMP CTU |
| 040F ₁₆ | 1001 ----- ---- | -- COPY R0,R6 |
| 0410 ₁₆ | 0000 0100 0001 0010 | -- INC R3,R3 |
| 0411 ₁₆ | 01-- 1010 0111 0--- | BODY |
| 0412 ₁₆ | 1110 0011 1000 0--- | NEXT |
| 0413 ₁₆ | 1001 ----- ---- | CTU |
| 0414 ₁₆ | 0000 0100 0001 0110 | |
| 0415 ₁₆ | 01-- 1010 0011 0--- | |
| 0416 ₁₆ | 0010 0100 1101 1--- | |

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

⇒ IS-ontwerp

- IS-schema

• Datapad-

componenten

• ASM-schema

• Controller

• Datapad

▪ RISC

Instructieset-stroomschema

- = visuele voorstelling van alle instructies en hun registertransfers

- Duidelijker als op een groot stuk papier
- Handig om de instructiedecoder te ontwerpen

De enige architecturale beslissing tot hietoe is de veronderstelling dat we beschikken over

- ❑ een geheugen (Mem)
- ❑ een registerbank (RF)
- ❑ een programmateller (PC)
- ❑ een instructieregister (IR)
- ❑ een statusvlag (Status)

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

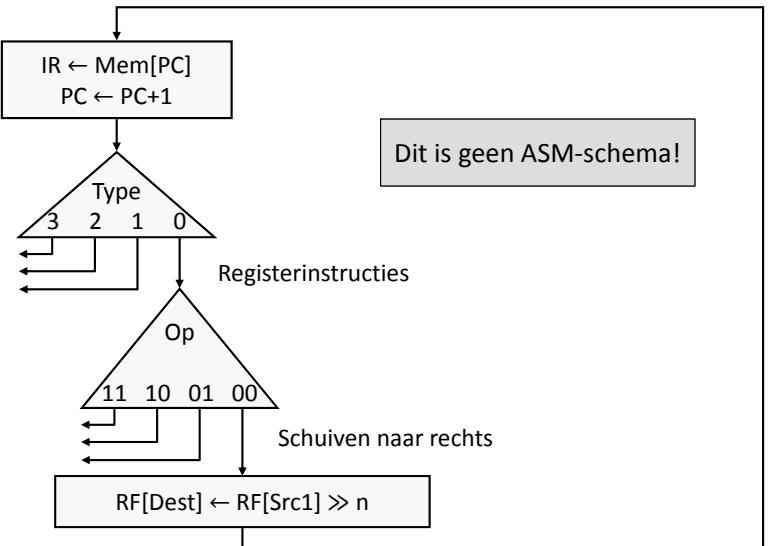
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Aritmetisch schuiven naar rechts



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

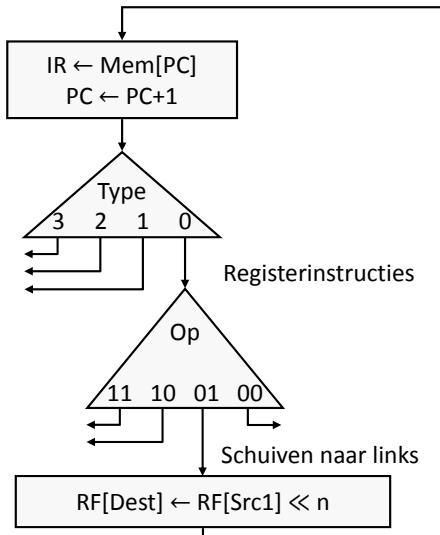
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Aritmetisch schuiven naar links



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

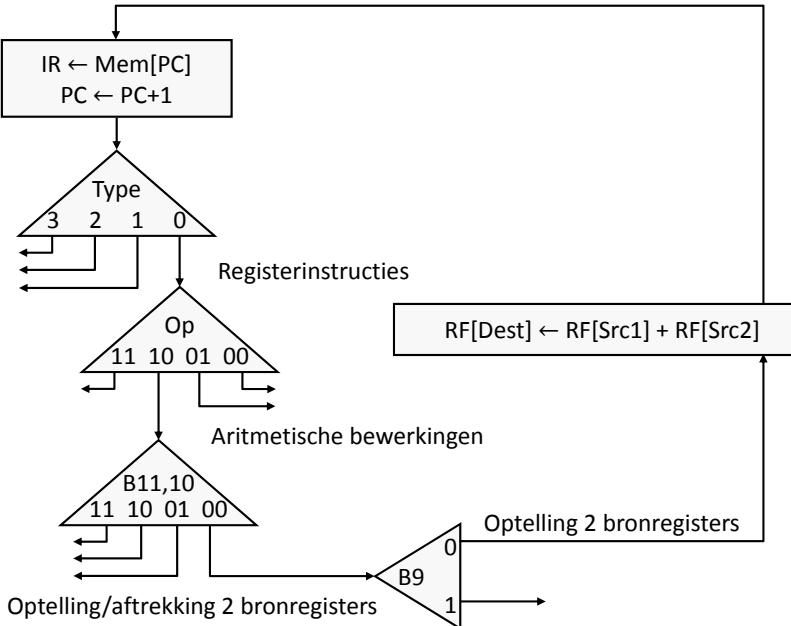
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Optelling van 2 bronregisters



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

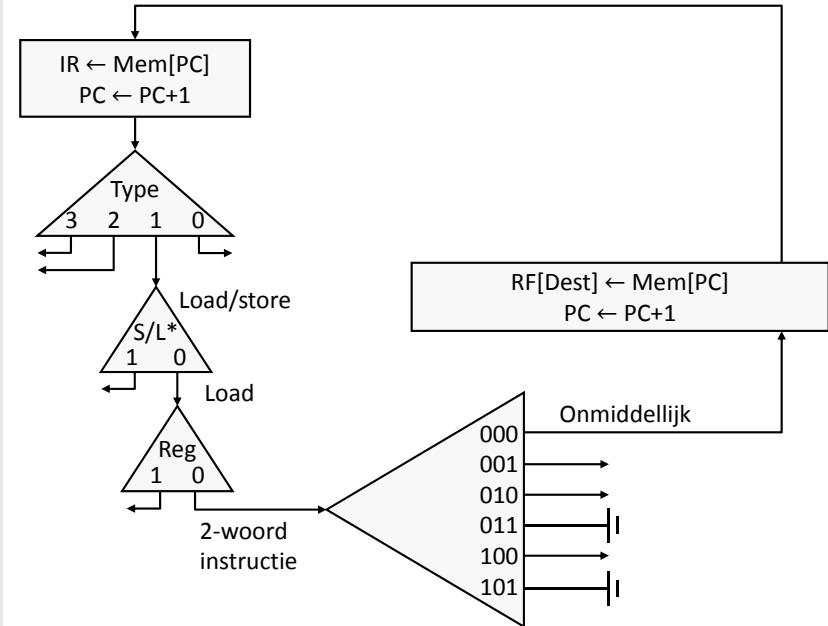
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Laad constante



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

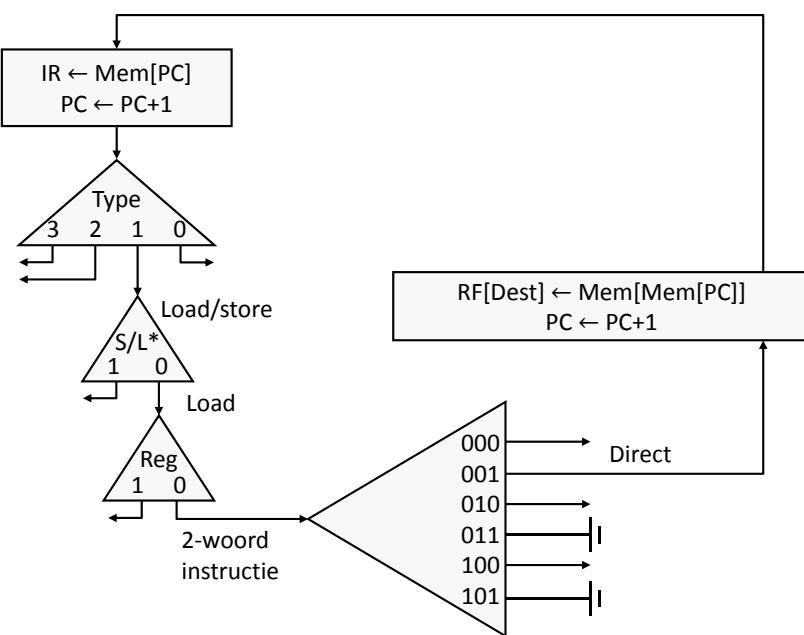
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Laad direct uit geheugen



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

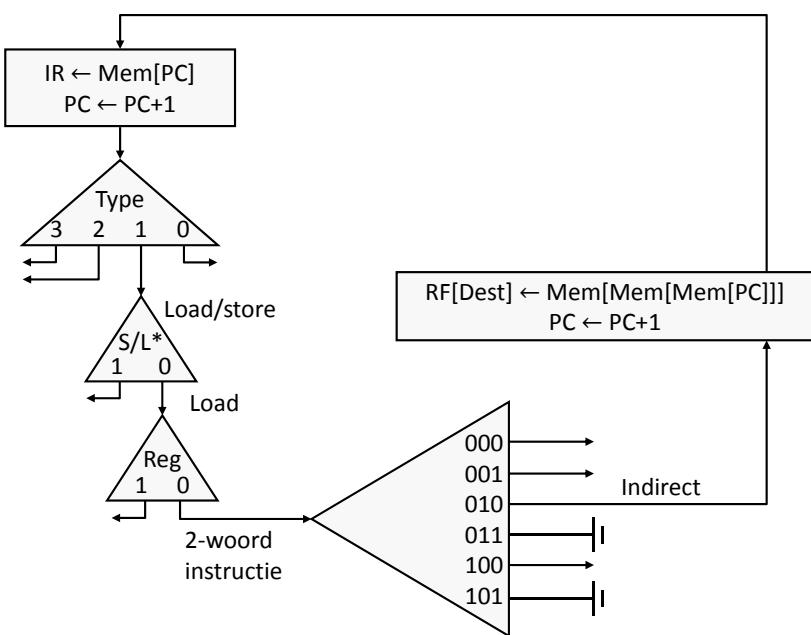
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Laad indirect



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

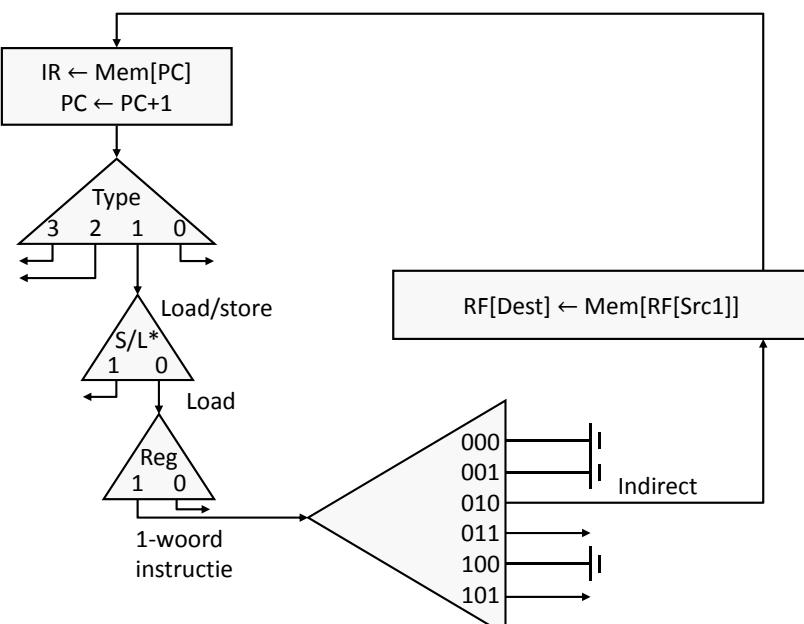
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Laad register-indirect



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

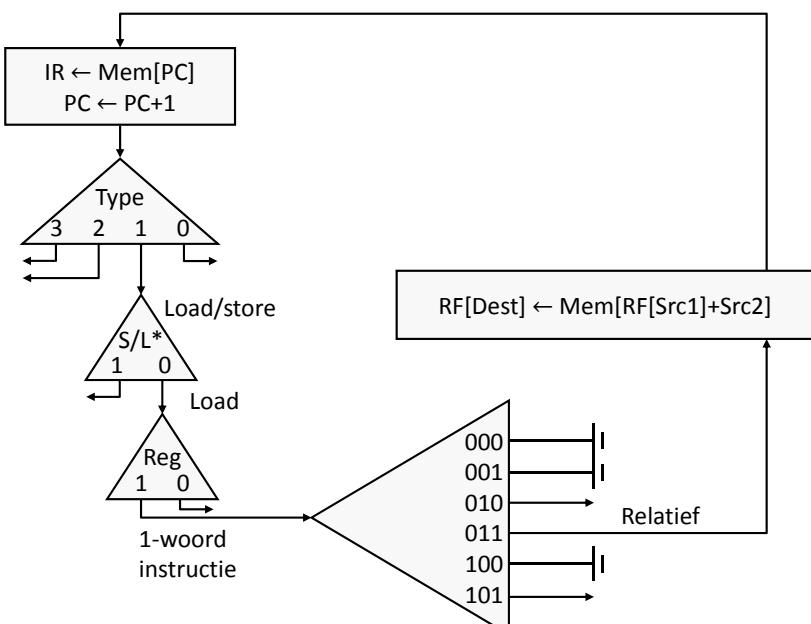
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Laad register-relatief



Programmeerbare processoren

Het programma

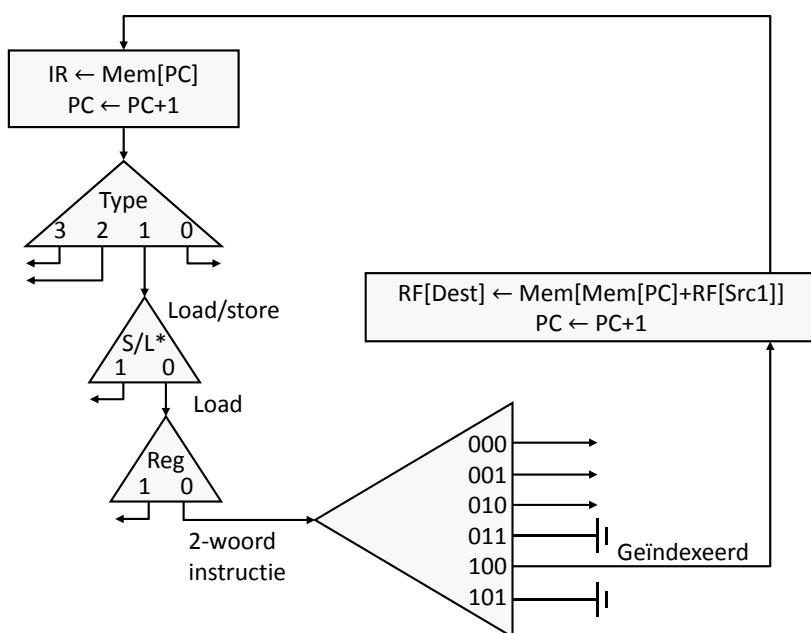
- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad
- RISC

Laad geïndexeerd



Programmeerbare processoren

Het programma

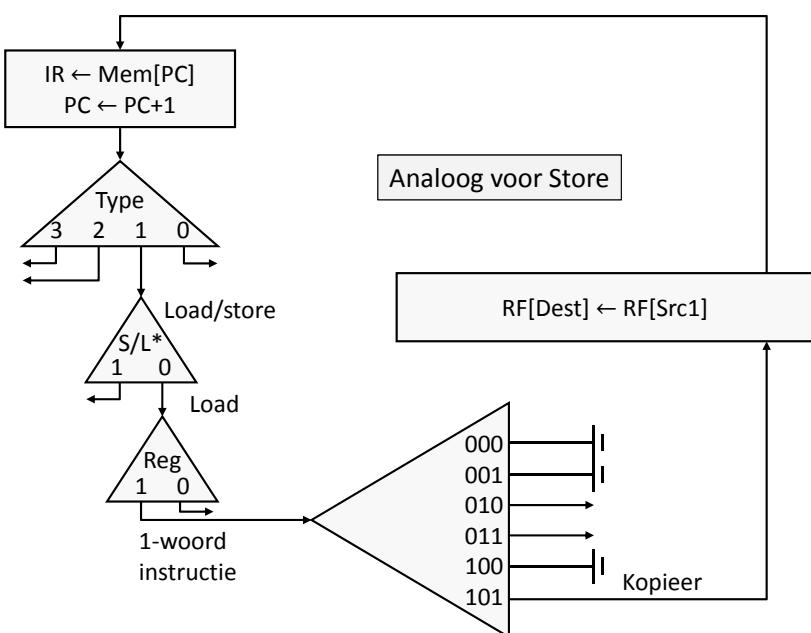
- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad
- RISC

Kopieer register



Programmeerbare processoren

Het programma

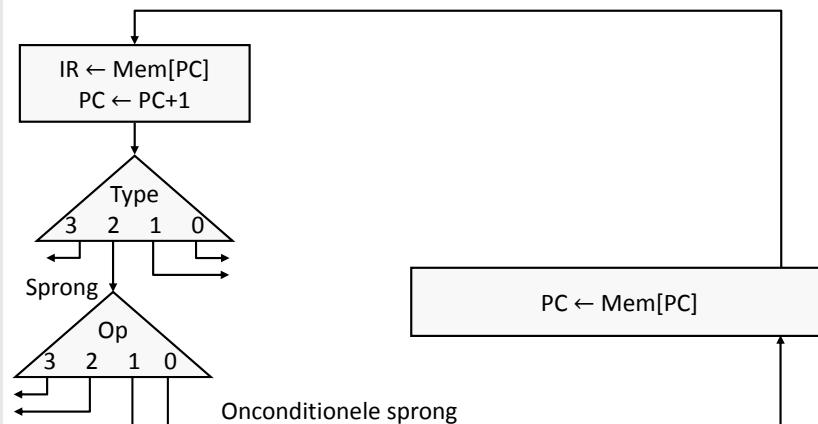
- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad
- RISC

Onconditionele sprong



Programmeerbare processoren

Het programma

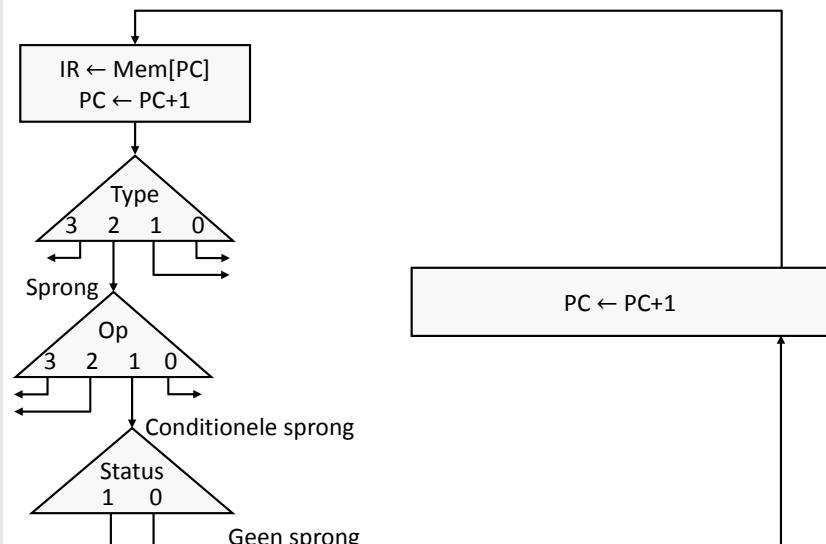
- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad
- RISC

Conditionele sprong



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

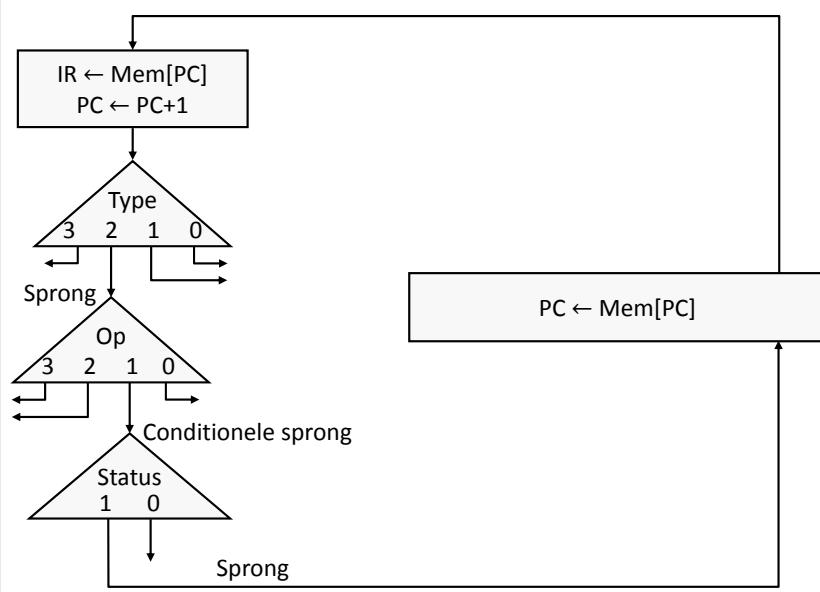
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Conditionele sprong



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

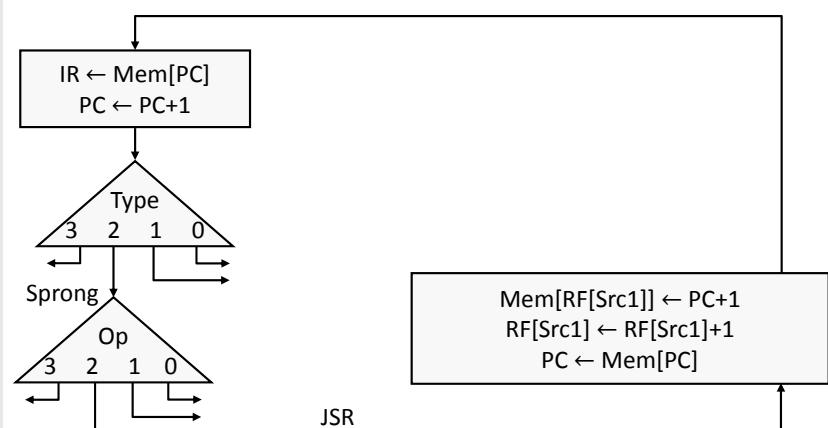
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Sprong naar subroutine



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

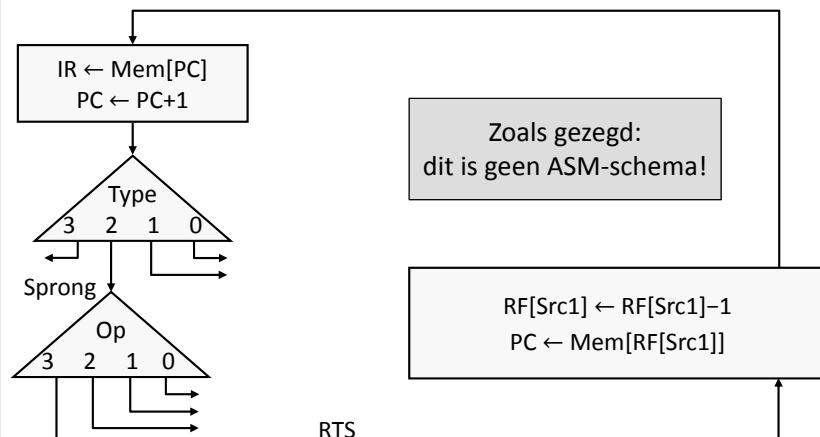
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Terugkeer uit subroutine



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

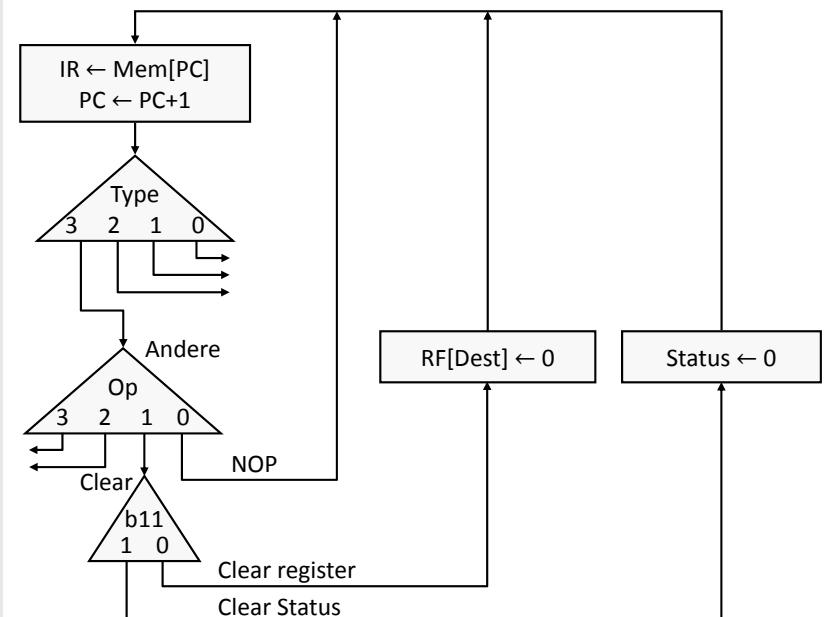
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

NOP & Clear



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

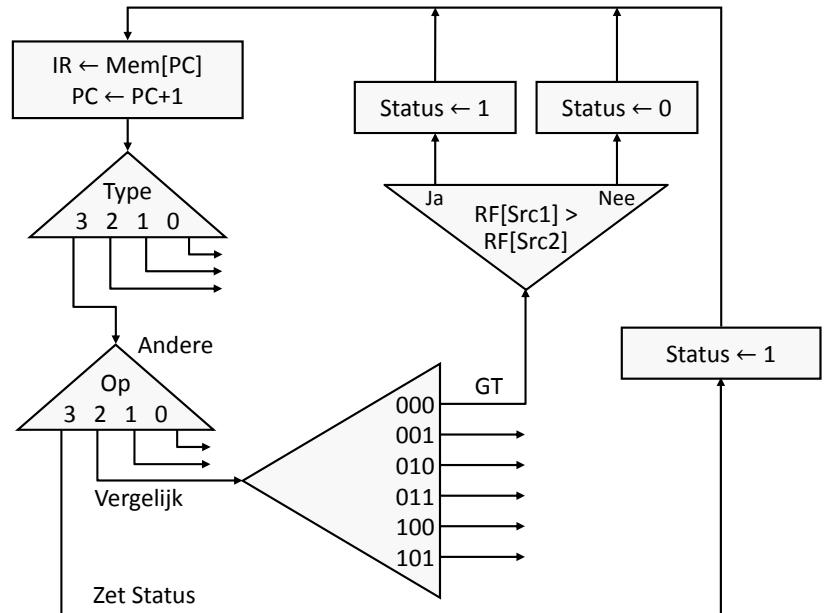
Processorontwerp

⇒ CISC

- IS-ontwerp
- ⇒ IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Zet status, incl. vergelijkingen



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- ⇒ Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

→ Complex Instruction Set Computer

- Ontwerp instructieset
- Instructieset-stroomschema
- Allocatie datapadcomponenten
- ASM-schema
- Ontwerp controller
- Ontwerp datapad

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- ⇒ Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Allocatie datapadcomponenten

□ Datapad tot nu toe nodig voor instructies:

- Registers:
 - Extern geheugen (1 lees/schrijfpoort; 64Kx16)
 - Registerbank (2 lees- & 1 schrijfpoort; 8x16)
 - Statusvlag (1 bit)
- FU:
 - ALU
 - Comparator
 - 16-bit links/rechts schuifoperatie

□ Om voor sommige toepassingen een hogere snelheid te halen kunnen bijkomende registers en/of FU toegevoegd worden.

Maar de instructieset moet herzien worden bij elke wijziging van het datapad!

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- ⇒ Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Allocatie datapadcomponenten

□ We beslissen hier om 1 toevoeging te voorzien omwille van de snelheid:

- Traagste instructie: Laad indirect

$$RF[Dest] \leftarrow Mem[Mem[adres]] \quad \& \quad adres \leftarrow Mem[PC]$$
- Deze instructie leest 3 × uit extern geheugen en duurt bijv. 150 ns als 50 ns/toegang
 - ⇒ maximum klokfrequentie = 6,7 MHz
 - ⇒ registerinstructies zullen dan ook 150 ns duren, zelfs als hun combinatorische vertraging maar 10 ns is
- Als we elke externe geheugentoegang in een aparte klokcyclus uitvoeren, kunnen we klokcycli van 50 ns gebruiken (pipelining)
 - ⇒ maximum klokfrequentie = 20 MHz
 - ⇒ traagste instructie duurt 3 klokcycli van 50 ns
 - ⇒ registerinstructie duurt slechts 50 ns

Programmeerbare processoren

Het programma

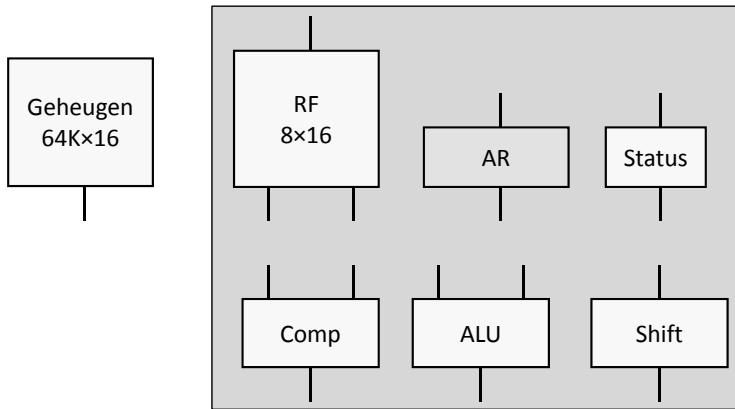
- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
- ⇒ Datapad-componenten
 - ASM-schema
 - Controller
 - Datapad
- RISC

Allocatie datapadcomponenten

- Daarom een bijkomend *Adresregister AR*: bewaar een adres, dat uit geheugen gelezen wordt, voor gebruik in de volgende klokcyclus



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
 - Controller
 - Datapad
- RISC

Afleiden van het ASM-schema

- {> Het IS-stroomschema geeft aan welke registertransfers er in een instructie gebeuren
- {> Het ASM-stroomschema geeft aan welke registertransfers er in een klokcyclus gebeuren



- {> Bepaal uit het IS-stroomschema welke registertransfers tegelijkertijd gebeuren
- {> Splits de instructie op in (zo weinig mogelijk) verschillende klokcycli zodat er geen hardware conflicten tussen operaties/datatransfers optreden in eenzelfde klokcyclus

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
 - Controller
 - Datapad
- RISC

Ontwerp programmeerbare processoren

Het programma

→ Processorontwerp

→ Complex Instruction Set Computer

- Ontwerp instructieset
- Instructieset-stroomschema
- Allocatie datapadcomponenten

→ ASM-schema

- Ontwerp controller
- Ontwerp datapad

➤ Reduced Instruction Set Computer

Programmeerbare processoren

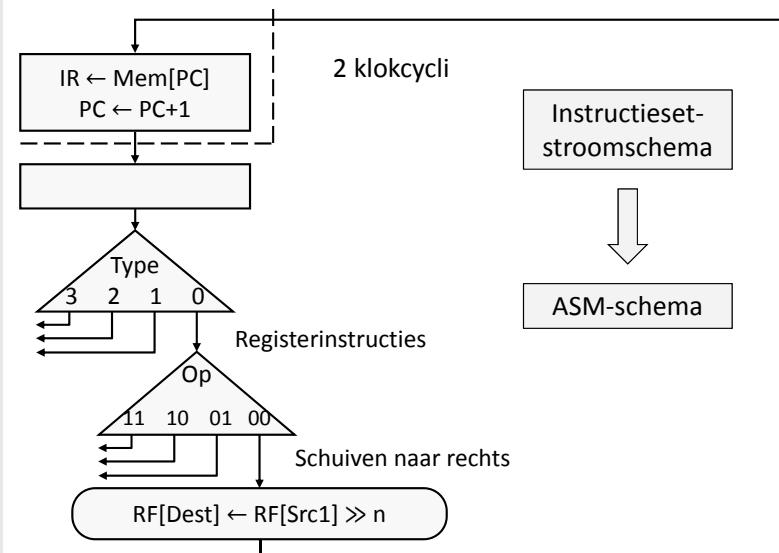
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
 - Controller
 - Datapad
- RISC

Aritmetisch schuiven naar rechts



Programmeerbare processoren

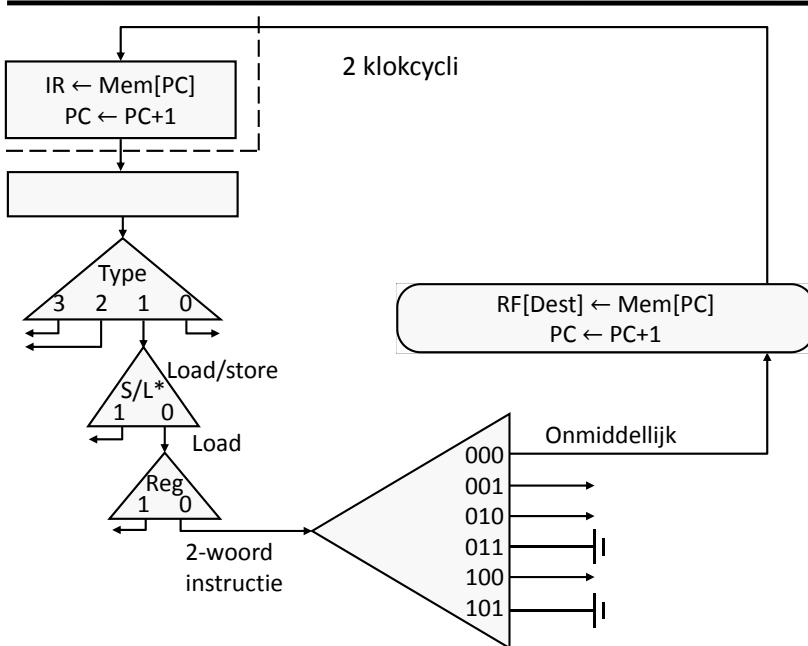
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
- Controller
- Datapad
- RISC

Laad constante



Programmeerbare processoren

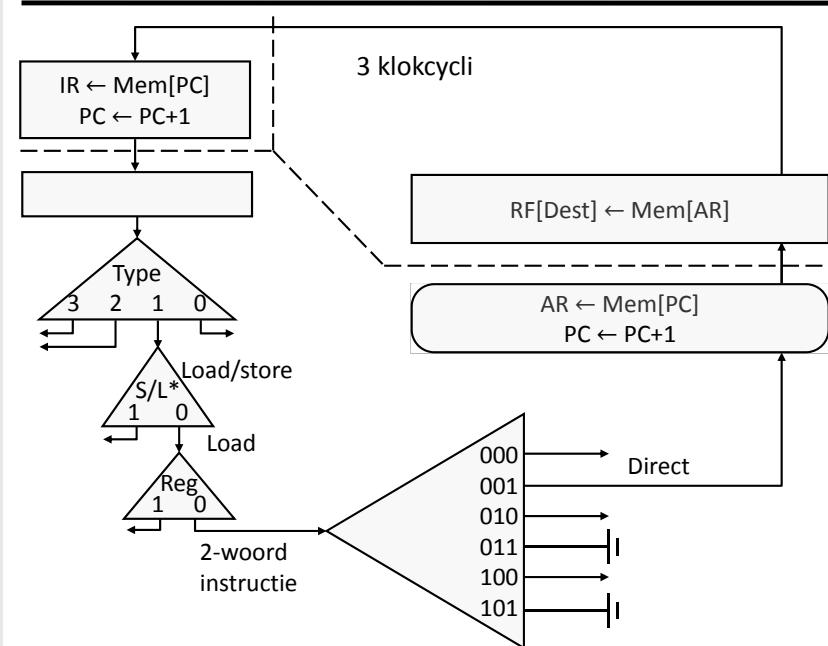
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
- Controller
- Datapad
- RISC

Laad direct uit geheugen



Programmeerbare processoren

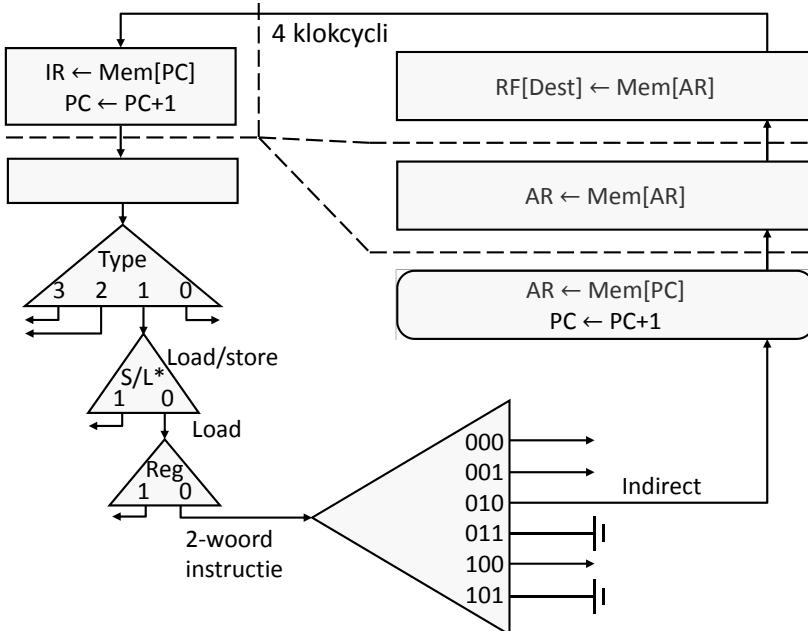
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
- Controller
- Datapad
- RISC

Laad indirect



Programmeerbare processoren

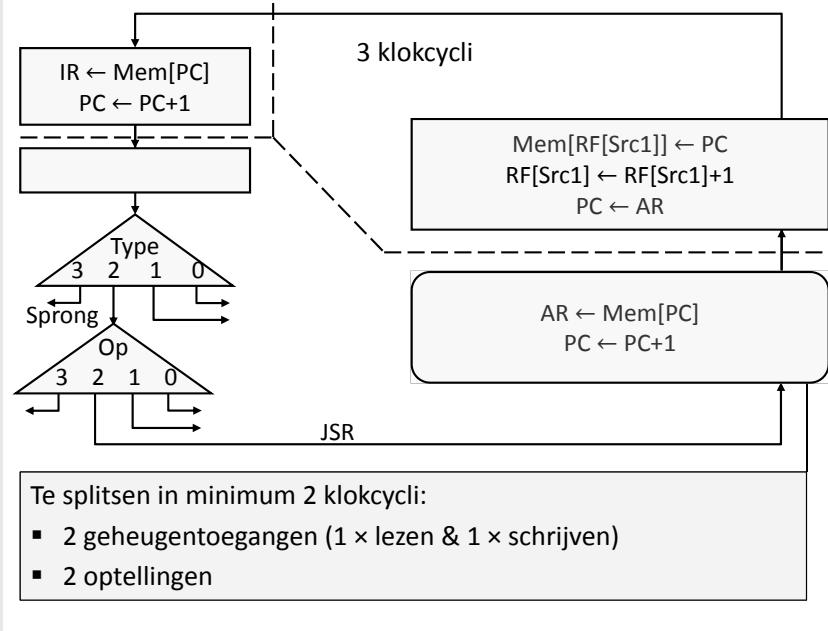
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
- ⇒ ASM-schema
- Controller
- Datapad
- RISC

Sprong naar subroutine



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

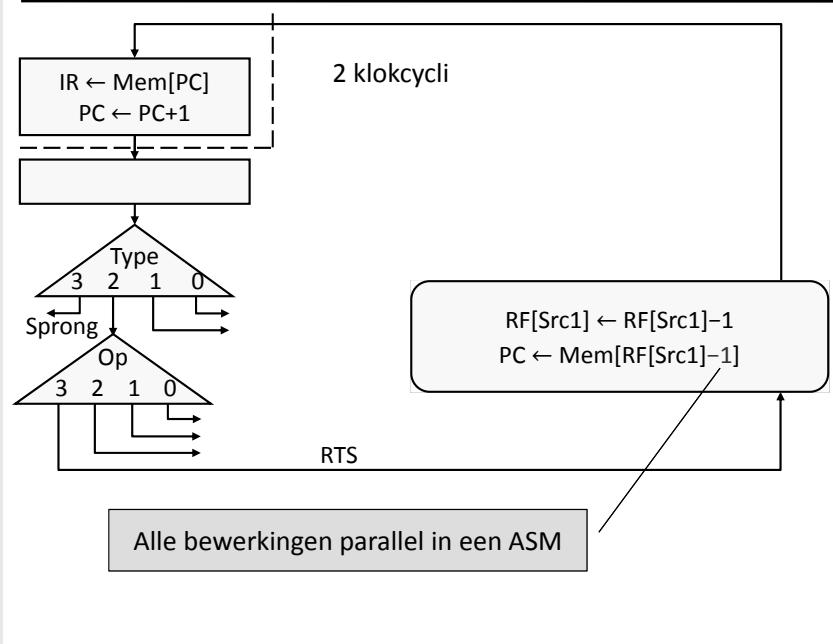
- IS-ontwerp
- IS-schema
- Datapad-componenten

⇒ ASM-schema

- Controller
- Datapad

RISC

Terugkeer uit subroutine



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema

⇒ Controller

• Datapad

RISC

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

→ Complex Instruction Set Computer

- Ontwerp instructieset
- Instructieset-stroomschema
- Allocatie datapadcomponenten
- ASM-schema

→ Ontwerp controller

▪ Ontwerp datapad

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema

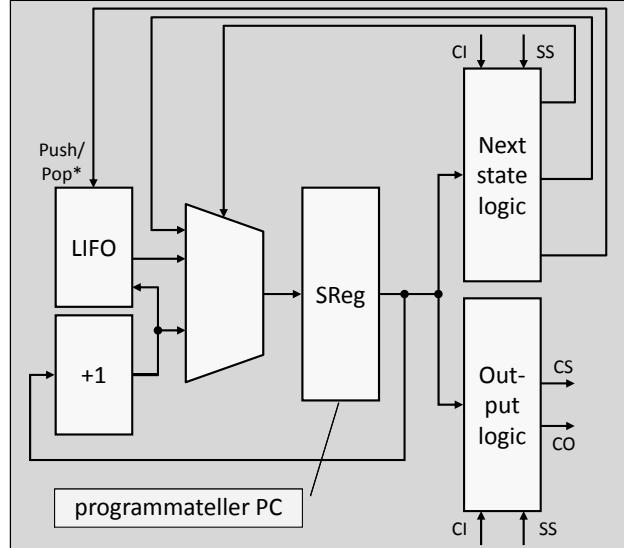
⇒ Controller

• Datapad

RISC

Ontwerp CISC-controller

Vertrek van de algemene FSMD-controller



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema

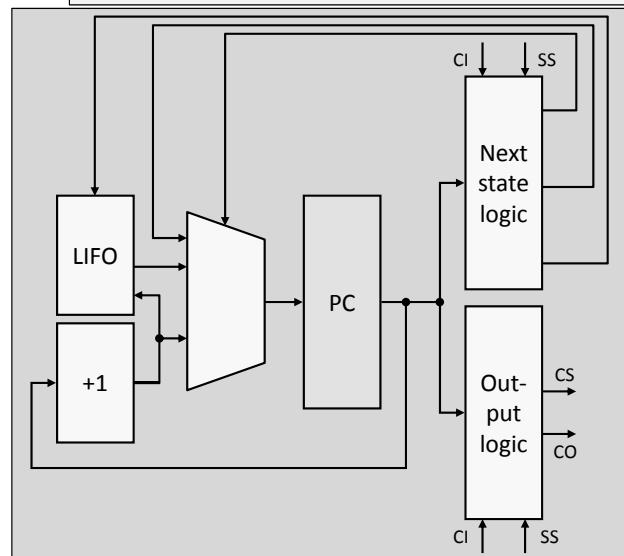
⇒ Controller

• Datapad

RISC

Ontwerp CISC-controller

Algoritme niet vast \Rightarrow relatie (PC \rightarrow next state & output) niet vast
 \Rightarrow bijkomende vertaalstap via Programmageheugen en IR



Programmeerbare processoren

Het programma

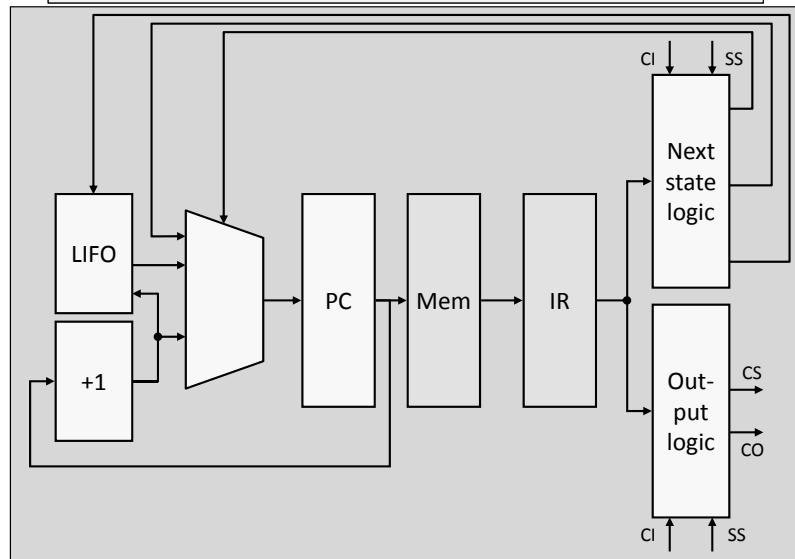
- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
- ⇒ Controller
- Datapad
- RISC

Ontwerp CISC-controller

1 instructie = sequentie van een paar kleine (micro-)instructies,
1 per klokcyclus: Next state & output zijn FSM i.p.v. combinatorisch



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
- ⇒ Controller
- Datapad
- RISC

Ontwerp CISC-controller

gewoonlijk zelfde (extern) geheugen voor programma & data

Programmeerbare processoren

Het programma

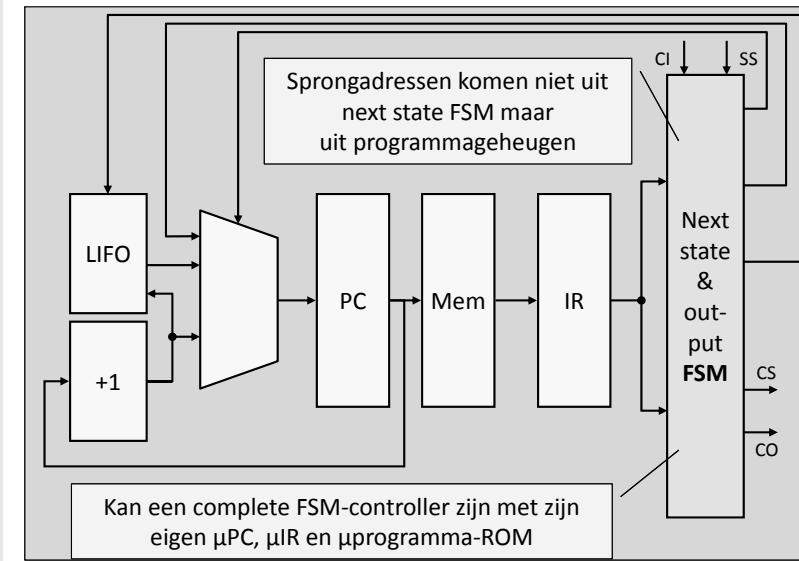
- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
- ⇒ Controller
- Datapad
- RISC

Ontwerp CISC-controller

Sprongadressen komen niet uit next state FSM maar uit programmageheugen



Kan een complete FSM-controller zijn met zijn eigen μPC, μIR en μprogramma-ROM

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
- ⇒ Controller
- Datapad
- RISC

Ontwerp CISC-controller

gewoonlijk zelfde (extern) geheugen voor programma & data

Programmeerbare processoren

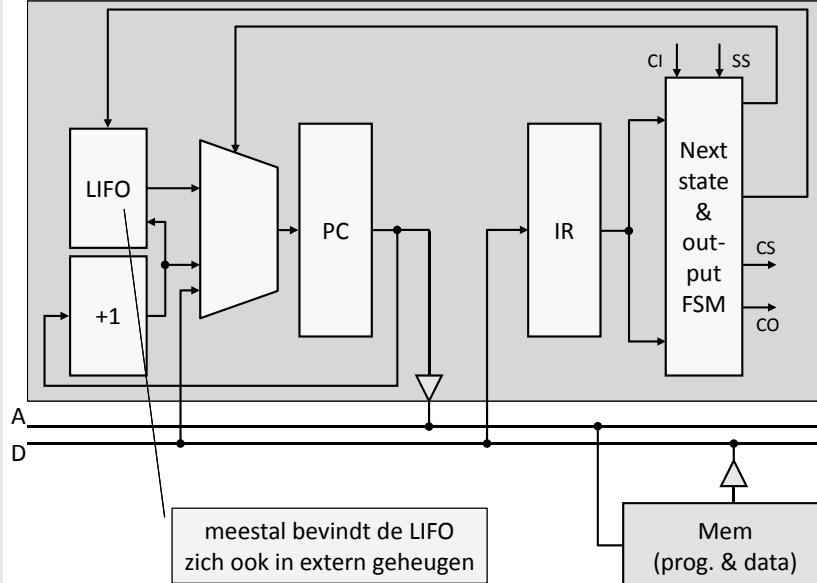
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- ⇒ CISC
 - IS-ontwerp
 - IS-schema
 - Datapad-componenten
 - ASM-schema
- ⇒ Controller
- Datapad
- RISC

Ontwerp CISC-controller



meestal bevindt de LIFO zich ook in extern geheugen

Mem (prog. & data)

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

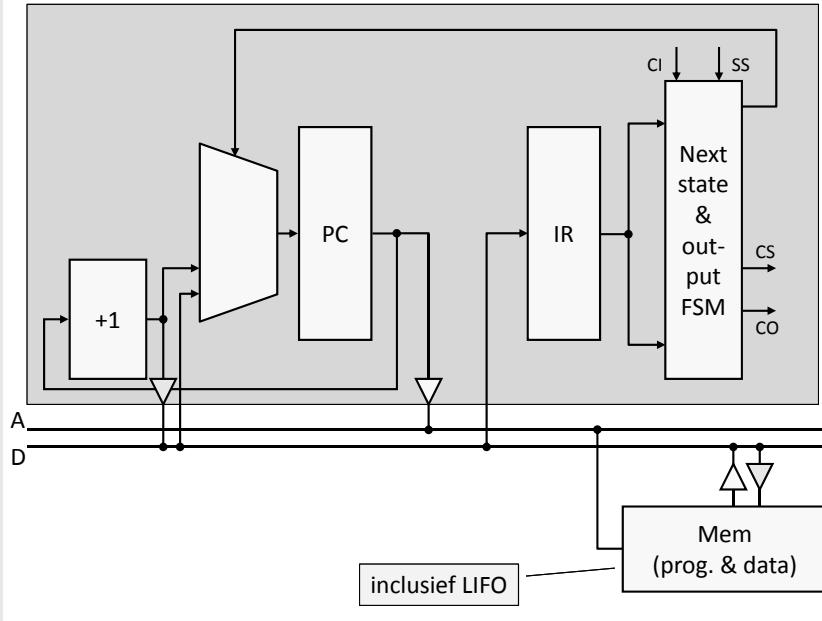
- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema

⇒ Controller

• Datapad

▪ RISC

Ontwerp CISC-controller



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-d datapad

- De componenten (= registers & FU) zijn reeds vroeger vastgelegd bij allocatie datapadcomponenten
- Nu enkel nog de verbindingen tussen de componenten (inclusief 3-state buffers)
 - Af te leiden uit ASM-schema
- Als er conflicten tussen het ASM-schema en de componenten/verbindingen zijn
 - Extra componenten ⇒ herbegin vanaf de allocatie van de componenten
 - Extra klokcycli ⇒ herbegin vanaf de bepaling van het ASM-schema

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

→ Complex Instruction Set Computer

- Ontwerp instructieset
- Instructieset-stroomschema
- Allocatie datapadcomponenten
- ASM-schema
- Ontwerp controller

→ Ontwerp datapad

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

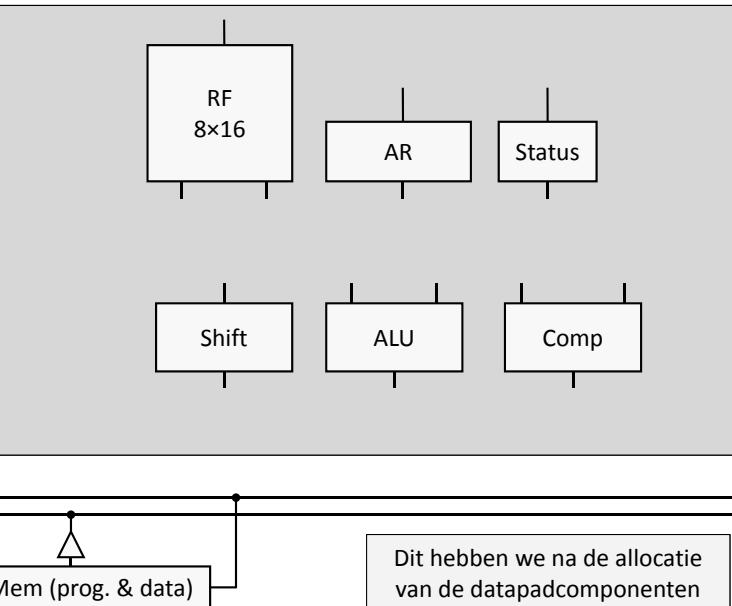
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-d datapad



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

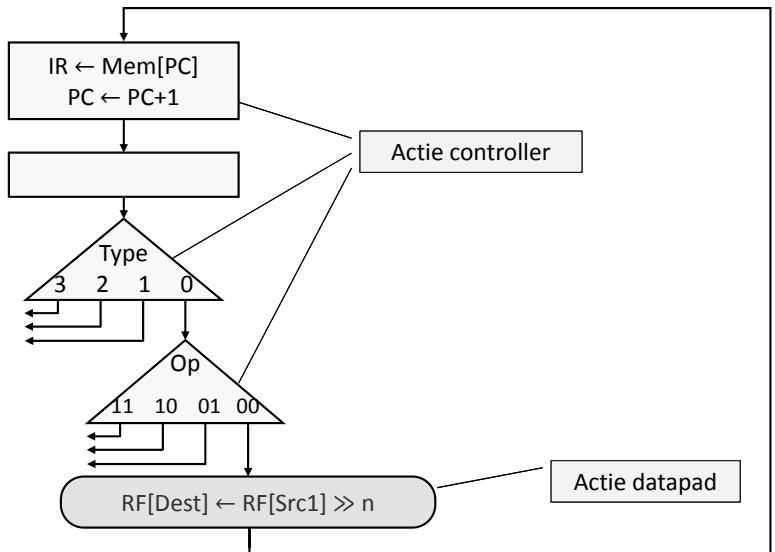
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

ASM-schema: schuiven naar rechts



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

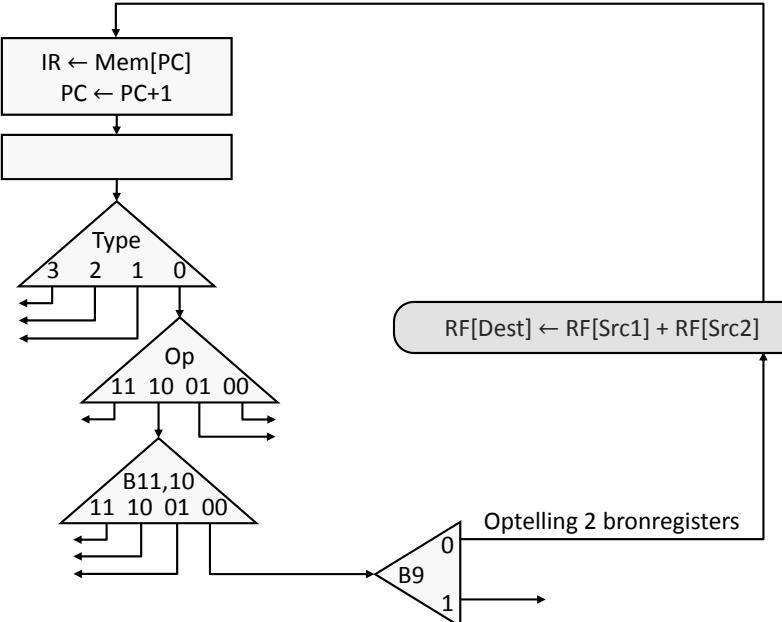
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

ASM-schema: optelling van 2 bronregisters



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

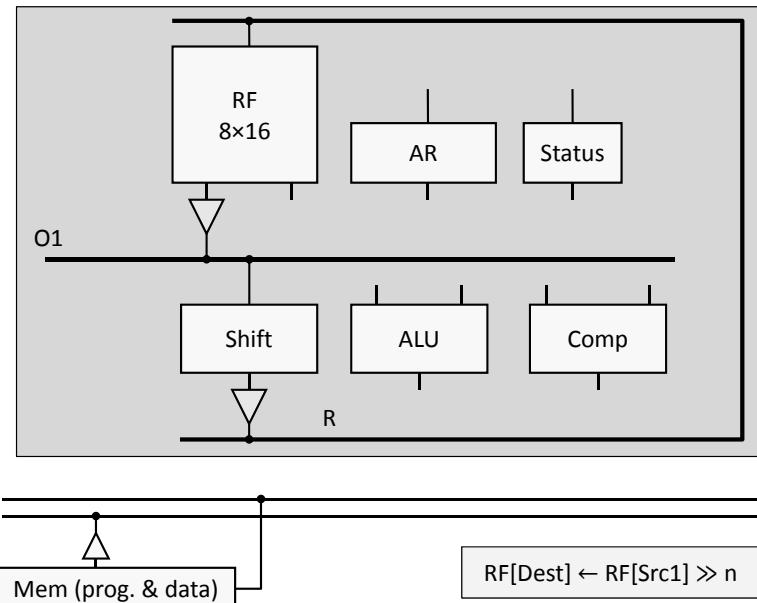
⇒ Datapad

▪ RISC

Ontwerp CISC-datapad

KATHOLIEKE UNIVERSITEIT
LEUVEN

Ontwerp CISC-datapad



$$RF[Dest] \leftarrow RF[Src1] \gg n$$

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

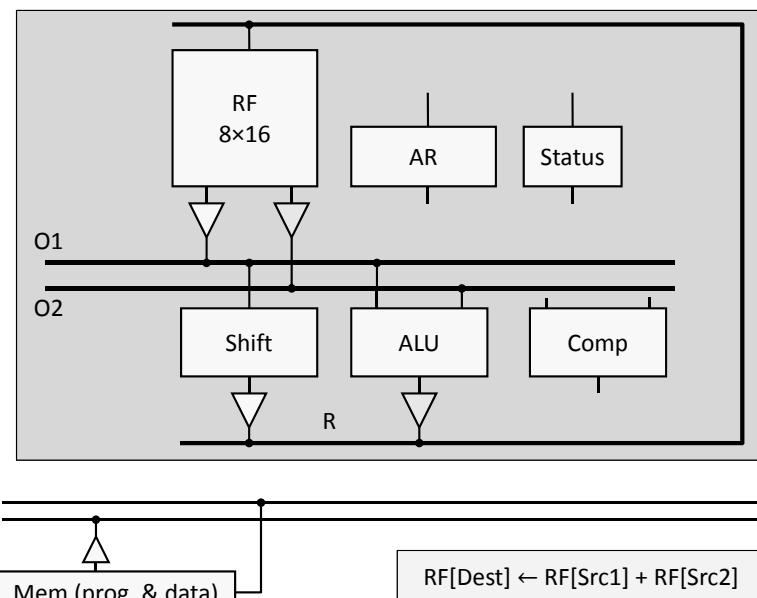
- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-datapad

KATHOLIEKE UNIVERSITEIT
LEUVEN



$$RF[Dest] \leftarrow RF[Src1] + RF[Src2]$$

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

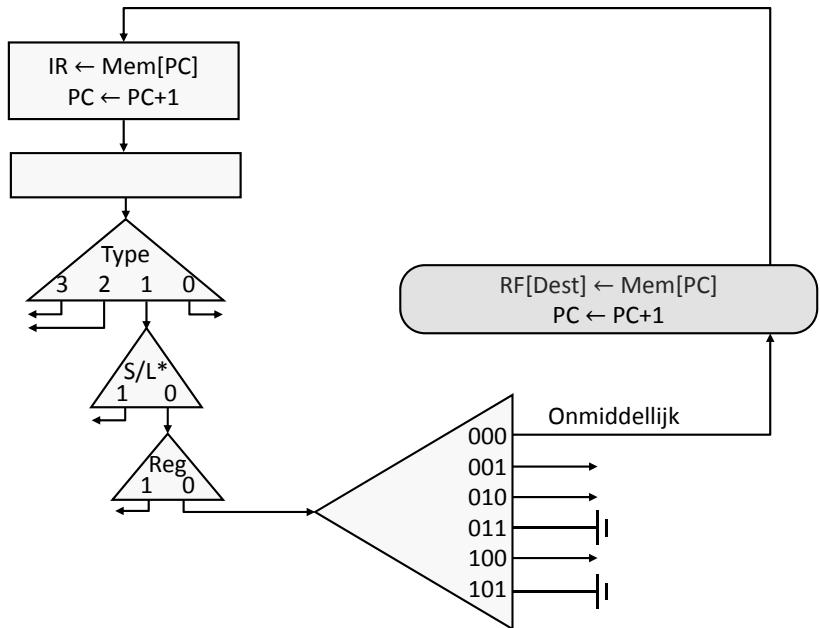
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: laad constante



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

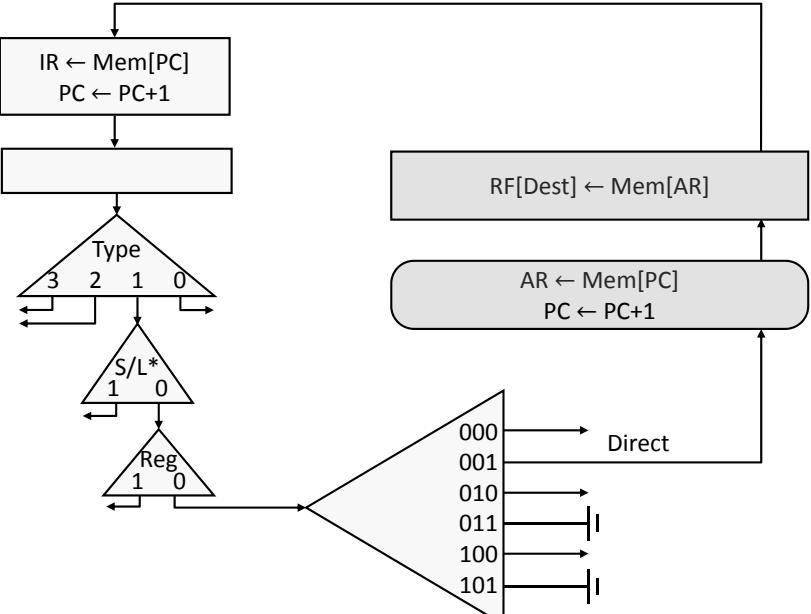
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: laad direct uit geheugen



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

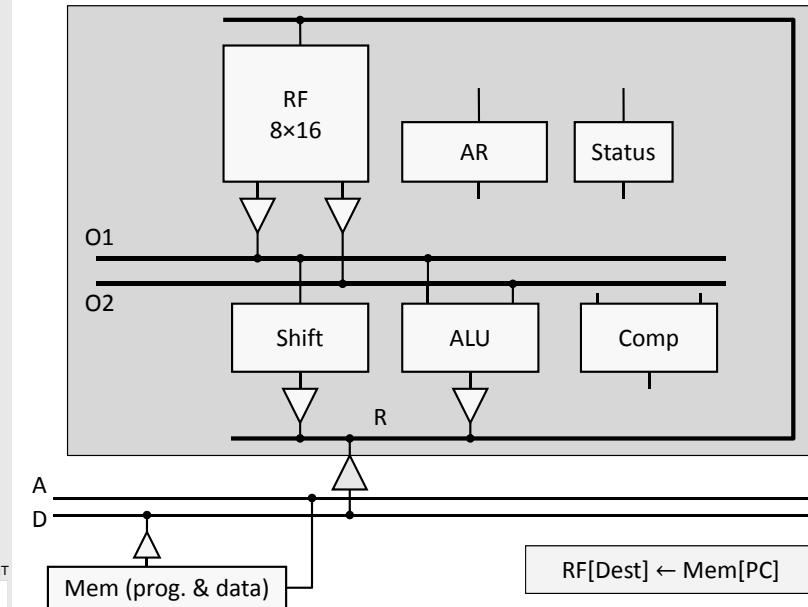
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Ontwerp CISC-datapad



KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

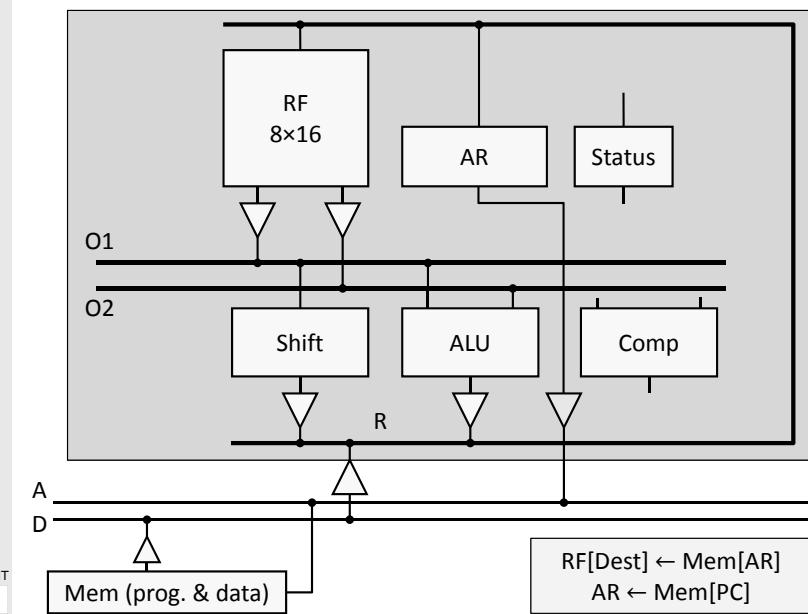
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Ontwerp CISC-datapad



KATHOLIEKE UNIVERSITEIT
LEUVEN

Program-
meerbare
processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema

• Datapad-
componenten

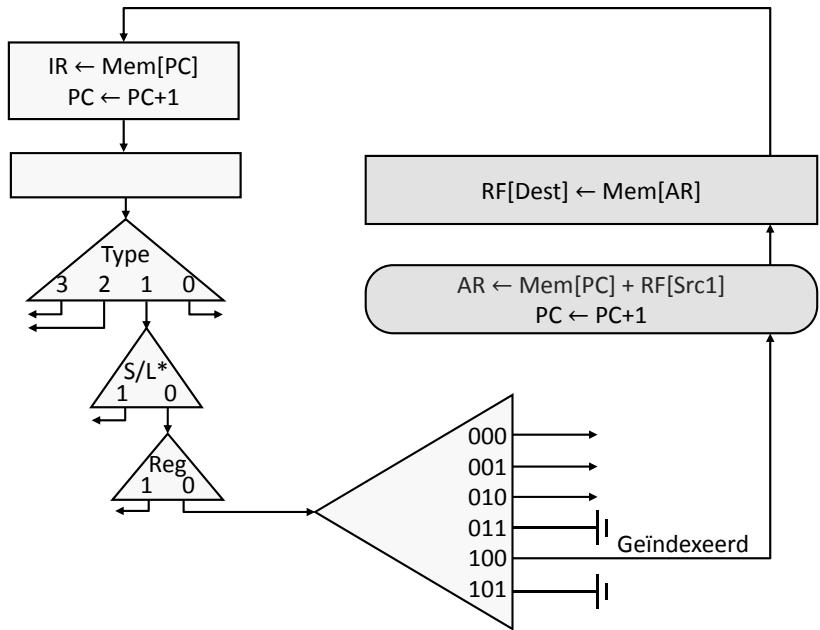
• ASM-schema

• Controller

⇒ Datapad

▪ RISC

ASM-schema: laad geïndexeerd



Program-
meerbare
processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema

• Datapad-
componenten

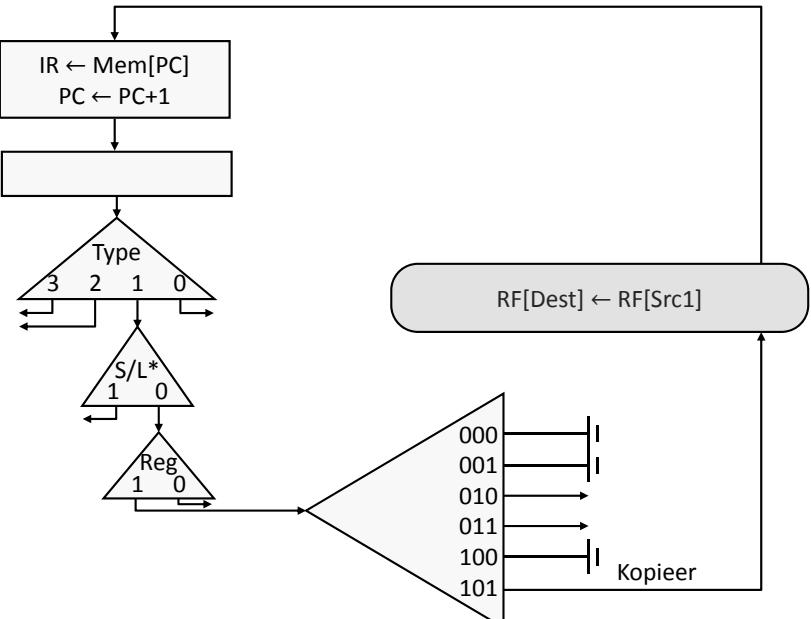
• ASM-schema

• Controller

⇒ Datapad

▪ RISC

ASM-schema: kopieer register



Program-
meerbare
processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema

• Datapad-
componenten

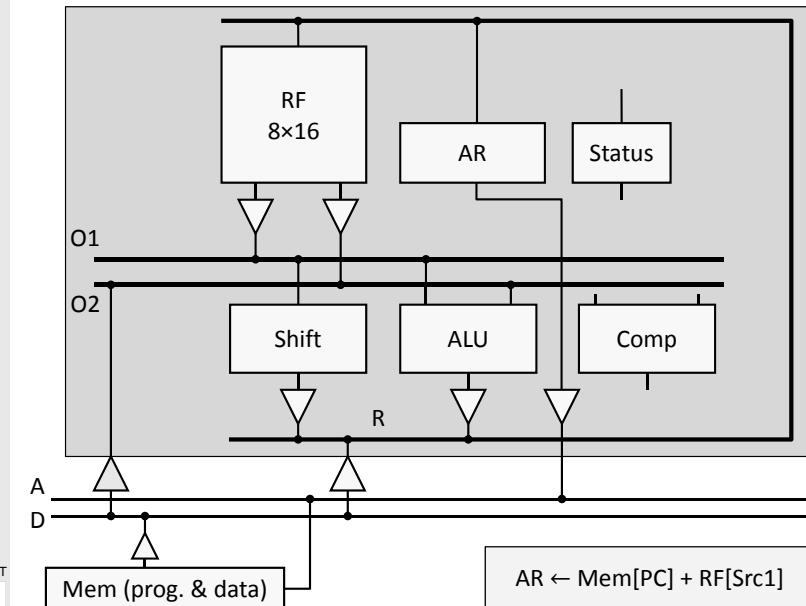
• ASM-schema

• Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-datapad



Program-
meerbare
processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema

• Datapad-
componenten

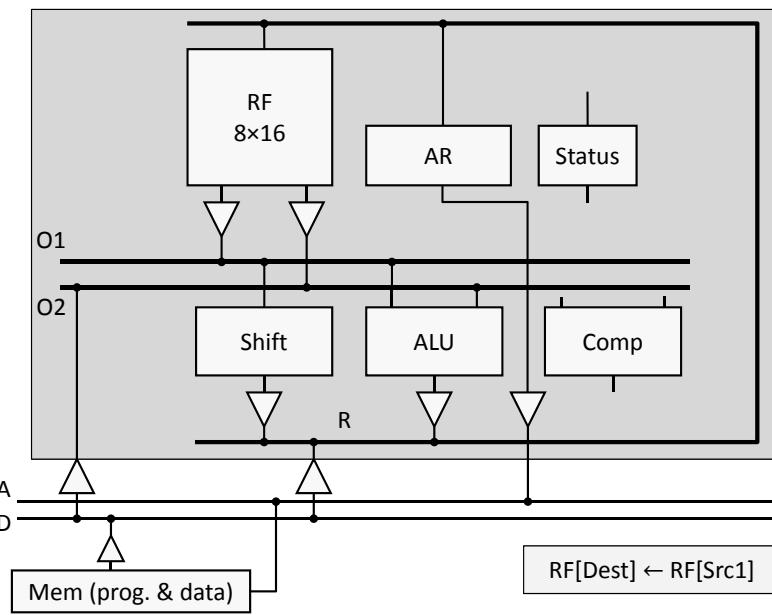
• ASM-schema

• Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-datapad



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

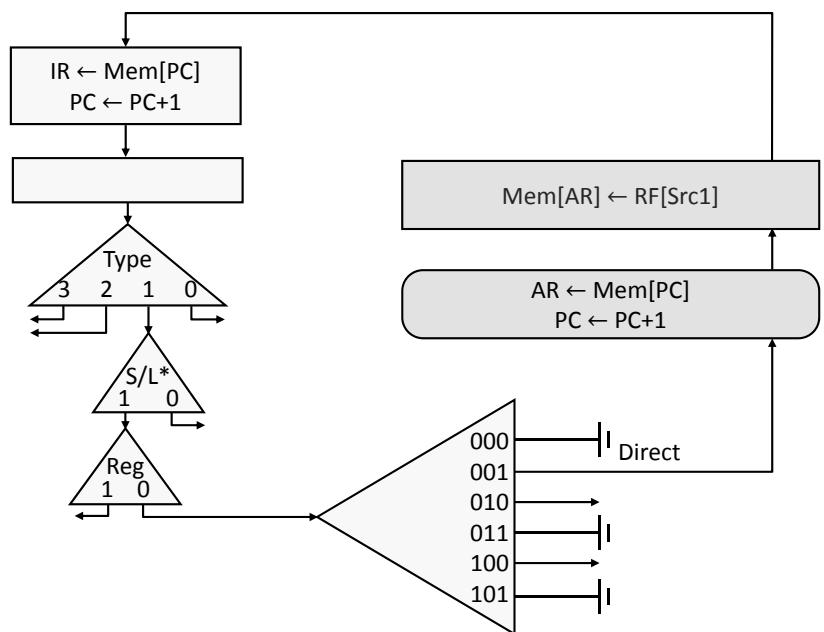
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: stockeer direct

KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

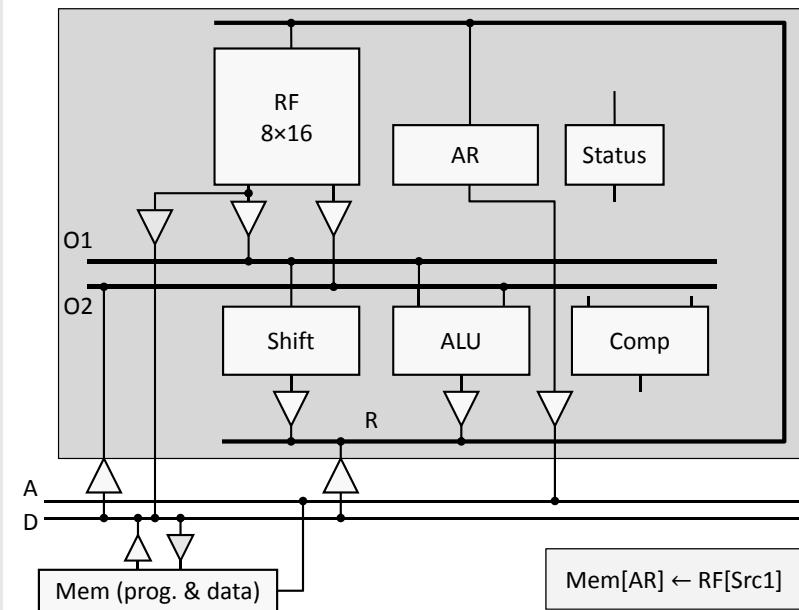
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Ontwerp CISC-datapad

KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

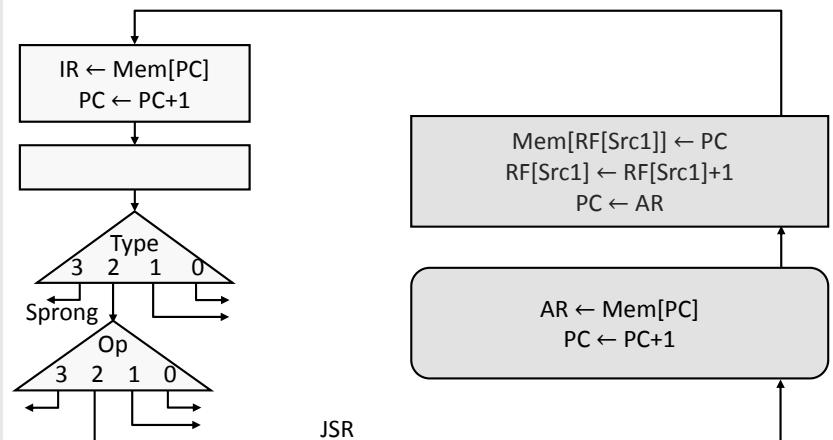
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: sprong naar subroutine

KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

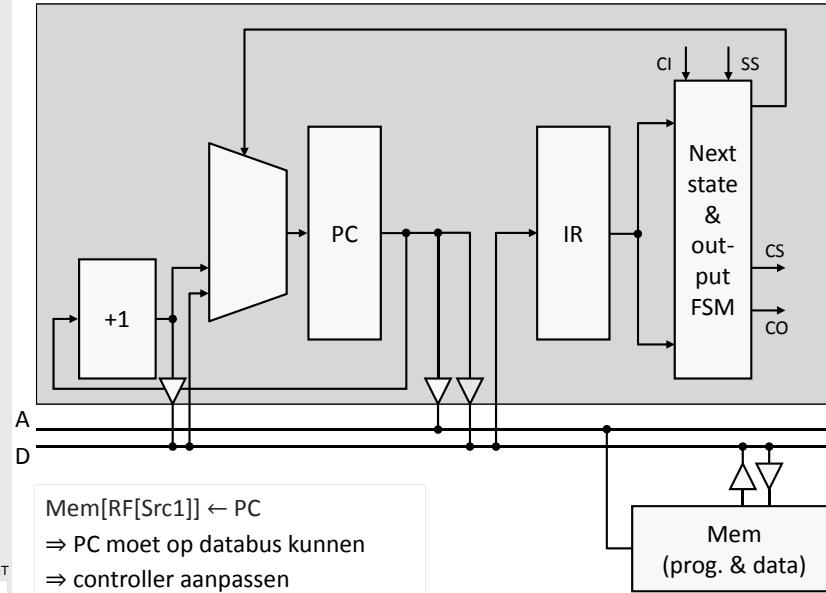
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Aanpassing CISC-controller

KATHOLIEKE UNIVERSITEIT
LEUVEN

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

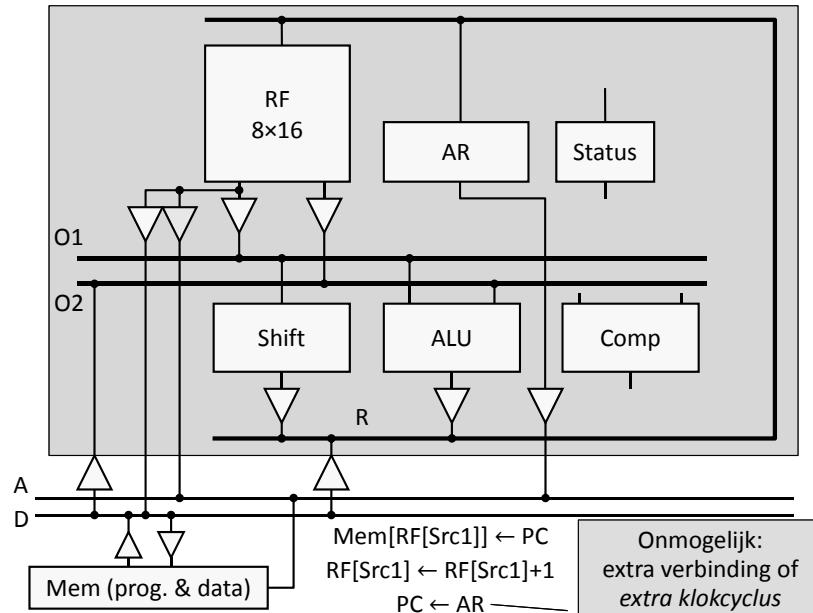
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Ontwerp CISC-d datapad



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

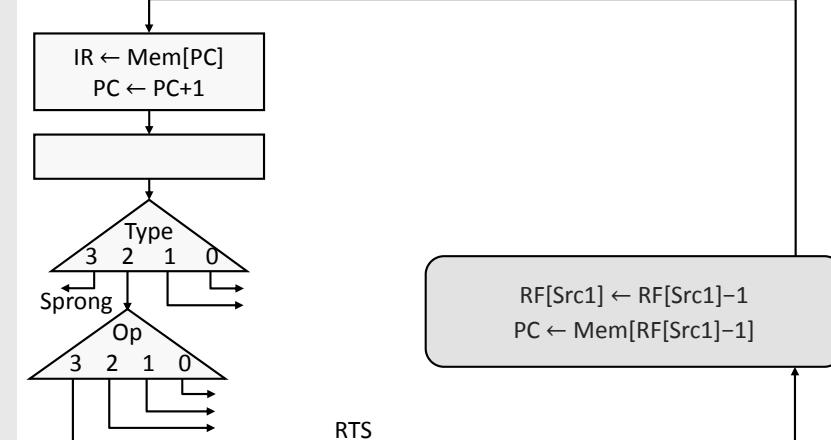
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: terugkeer uit subroutine



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

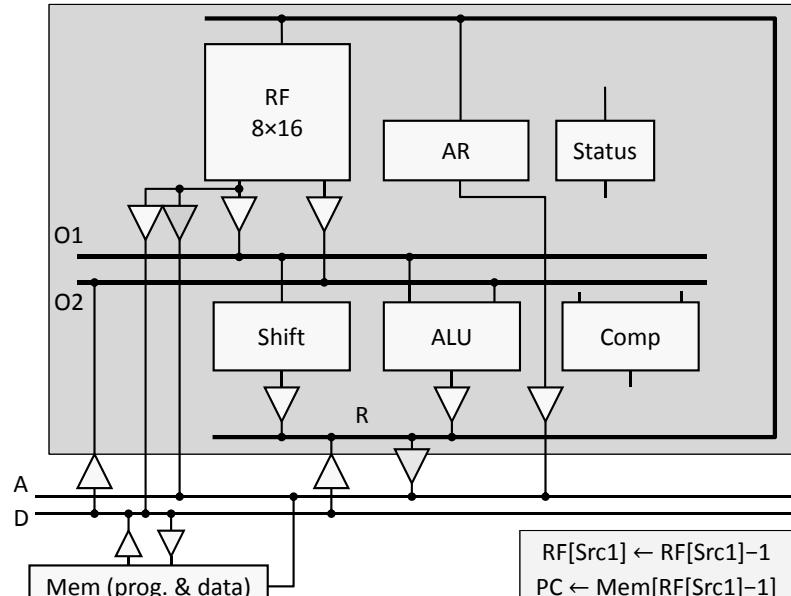
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

Ontwerp CISC-d datapad



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

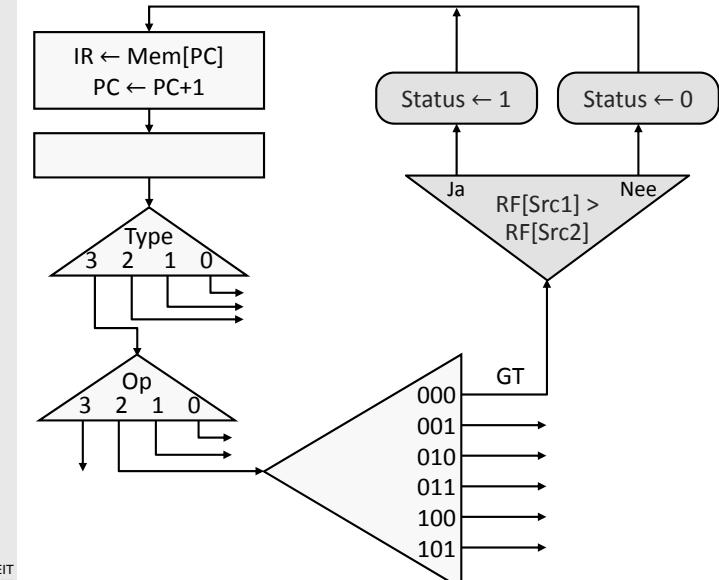
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

- RISC

ASM-schema: vergelijk



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

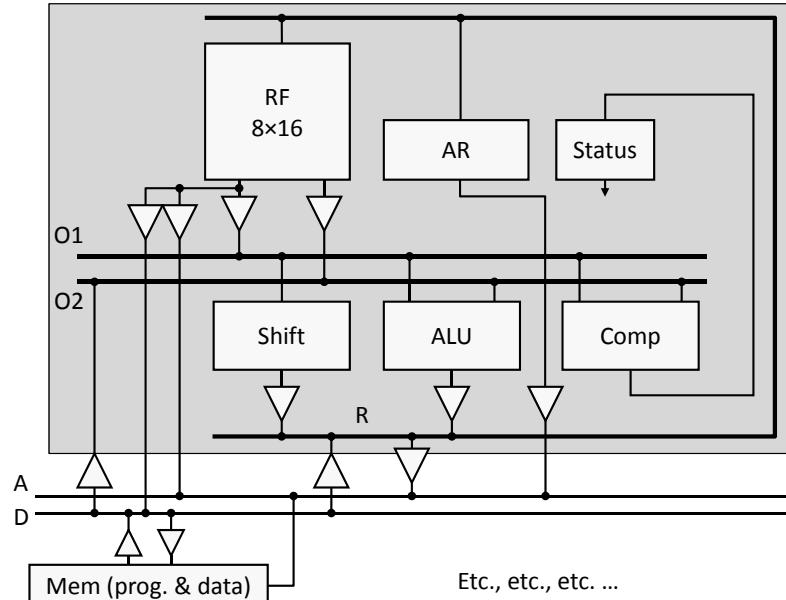
⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller

⇒ Datapad

▪ RISC

Ontwerp CISC-d datapad



Programmeerbare processoren

Het programma

Processorontwerp

➤ Complex Instruction Set Computer

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

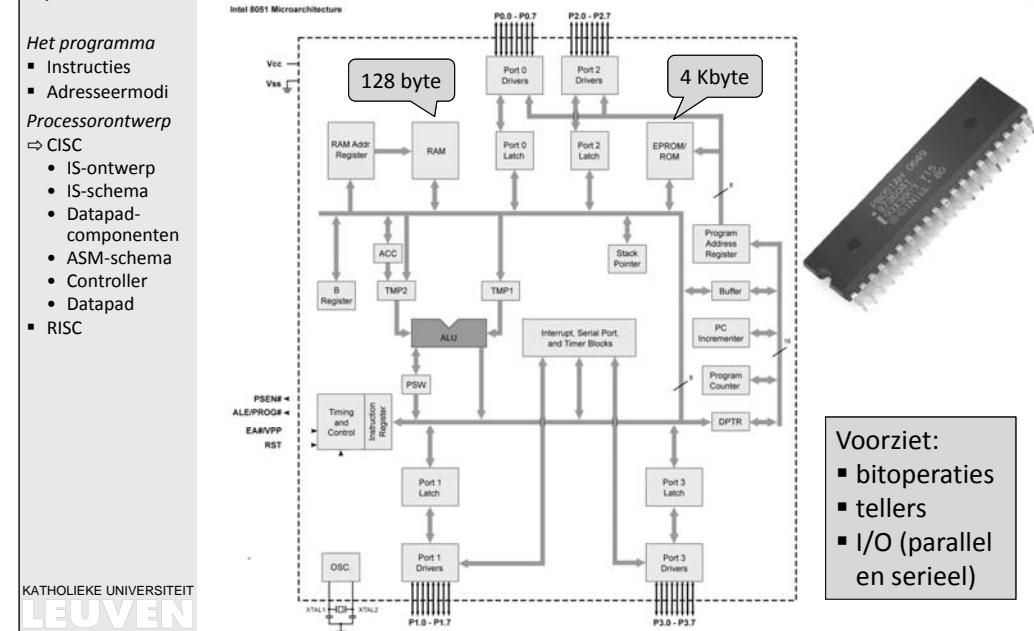
Processorontwerp

⇒ CISC

- IS-ontwerp
- IS-schema
- Datapad-componenten
- ASM-schema
- Controller
- Datapad

▪ RISC

Voorbeeld: 8051-microcontroller



- Voorziet:
- bitoperaties
 - tellers
 - I/O (parallel en serieel)

Ontwerp programmeerbare processoren

□ Het programma

→ Processorontwerp

➤ Complex Instruction Set Computer

➤ Reduced Instruction Set Computer

Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

▪ CISC

⇒ RISC

Ontwerp RISC

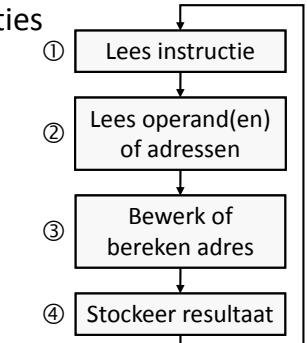
Ongeveer zoals CISC maar hoge klokfrequentie zeer belangrijk ⇒ zo veel mogelijk pipelining

➤ Pipelining van uitvoering instructies

| 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 |

⇒ beperkingen op instructieset

- eenvoudige instructies
- instructies ± even lang



➤ Pipelining van controller en datapad

- laat het stoppen van de pipeline toe bijv. sprong: nieuwe PC moet eerst berekend worden

Programmeerbare processoren

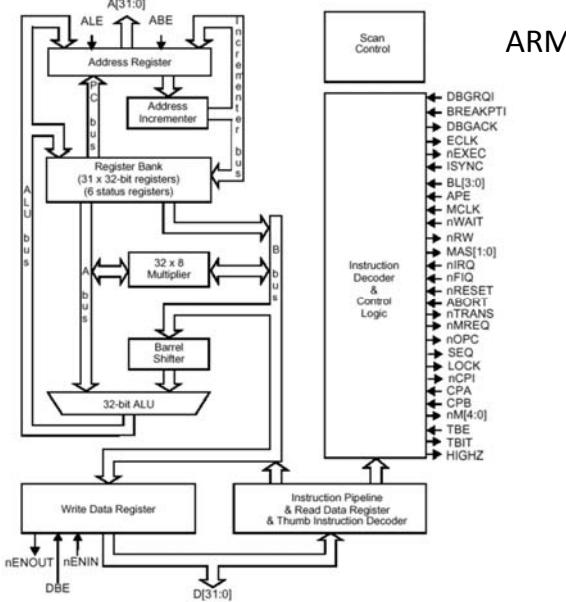
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- ⇒ RISC

Voorbeeld: ARM-processor



Programmeerbare processoren

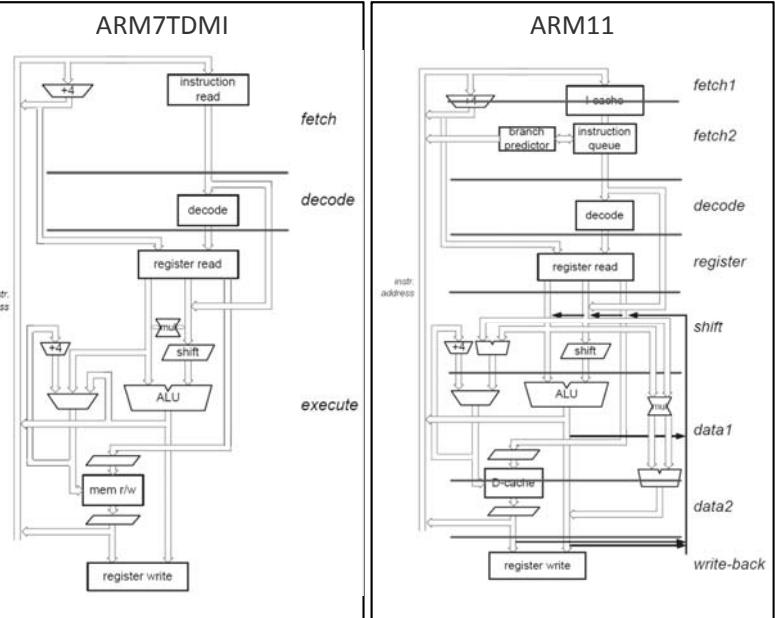
Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- ⇒ RISC

Voorbeeld: ARM-processor pipelining



Programmeerbare processoren

Het programma

- Instructies
- Adresseermodi

Processorontwerp

- CISC
- ⇒ RISC

Voorbeeld: ARM-processor instructieset

| Cond | 0 | 0 | I | Opcode | S | Rn | Rd | Operand 2 | | | Data Processing / PSR Transfer | | | | |
|------|---|---|---|------------------------------|---|----|------------------------|-----------|----------------------|-------------------------------|--|---------------------------|---|-------------------------------|----|
| Cond | 0 | 0 | | ■ NOT, AND, OR, XOR, AND NOT | | | ■ +/- met/zonder carry | | | ■ eventueel enkel voor testen | | | Multiply | | |
| Cond | 0 | 0 | | | | | | | | | | | Multiply Long | | |
| Cond | 0 | 0 | | | | | | | | | | | Single Data Swap | | |
| Cond | 0 | 0 | | | | | | | | | | | Branch and Exchange | | |
| Cond | 0 | 0 | P | U | 0 | W | L | Rn | Rd | 0 | 0 | 0 | 1 | S H 1 | Rm |
| Cond | 0 | 0 | P | U | 1 | W | L | Rn | Rd | Offset | 1 | S H 1 | Halfword Data Transfer: register offset | | |
| Cond | 0 | 1 | I | P | U | B | W | L | Rn | Rd | Halfword Data Transfer: immediate offset | | | Single Data Transfer | |
| Cond | 0 | 1 | I | | | | | | | 1 | | | | Undefined | |
| Cond | 1 | 0 | 0 | P | U | S | W | L | Rn | Register List | | | Block Data Transfer | | |
| Cond | 1 | 0 | 1 | L | | | | | | Offset | | | Branch | | |
| Cond | 1 | 1 | 0 | P | U | N | W | L | Rn | CRd | CP# | Coprocessor Data Transfer | | | |
| Cond | 1 | 1 | 0 | | | | | | CRd | CP# | CP | 0 | CRm | Coprocessor Data Operation | |
| Cond | 1 | 1 | 1 | | | | | | Rd | CP# | CP | 1 | CRm | Coprocessor Register Transfer | |
| Cond | 1 | 1 | 1 | 1 | | | | | Ignored by processor | | | Software Interrupt | | | |