Digitale Systemen (ET1 410)

Arjan van Genderen
Stephan Wong

Faculteit EWI
Technische Universiteit Delft
Cursus 2011

Verschil simulatie en synthese

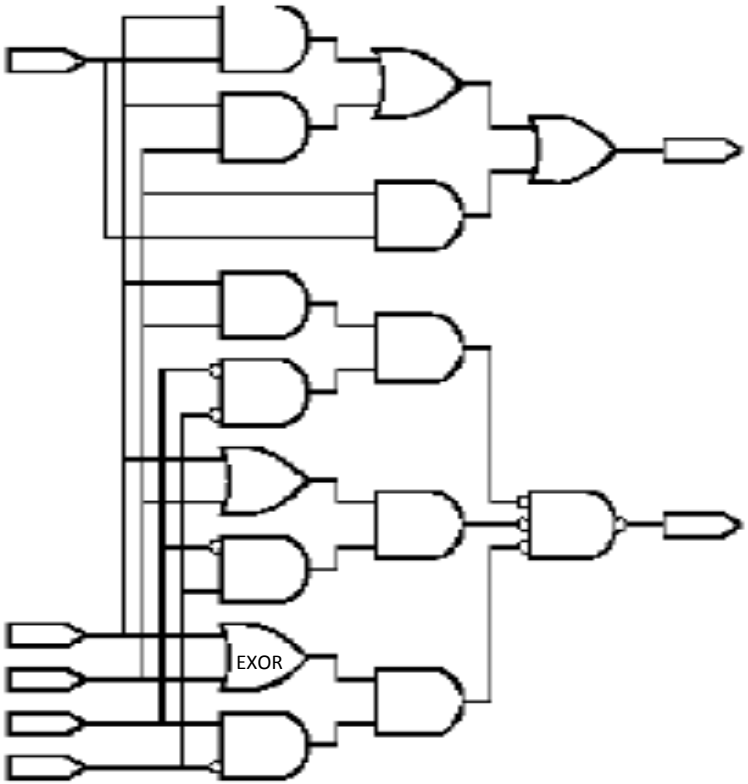
- **<u>Simulatie:</u>** functioneel testen van VHDL code
- **<u>Synthese:</u>** omzetten van VHDL code naar een in hardware implementeerbaar ontwerp
- *Opmerking: niet alles wat correct wordt gesimuleerd resulteert in correcte synthese*
- **<u>Inference:</u>** het vertalen van VHDL constructs naar hardware elementen

Aandachtspunten

- Denk aan hardware componenten (geen C)
- Hou componenten simpel
- Maak sensitivity-lijsten volledig (bv. alle RHS signalen)
- Synchrone vs. asynchrone schakelingen
- Gebruik van variabelen vs. signalen
- Combinatorische vs. sequentiële schakelingen:
  - Combinatorisch: voor alle ingangscombinaties is er een uitgang gedefinieerd
  - Sequentieel: integendeel, dus introductie van geheugen-elementen (latches of flip-flops)

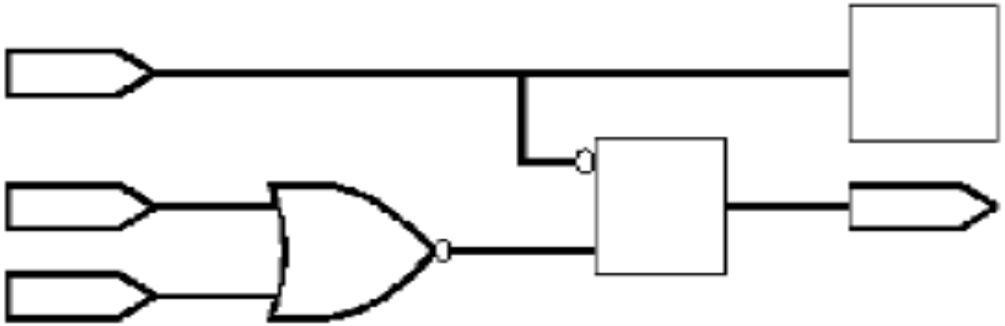
Inference van combinatorische schakelingen

```
--
-- pseudo code for a single bit crithmetic/
logic unit
--
s1 <= in1 and in2;
s2 <= in1 or in2;
s3 <= in1 xor in2; -- perform the sum opera-
tion
c_out <=   (in1 and c_in) or (in1 and in2)
           or (in2 and c_in);
out <= s1 when sel = "00" else
           s2 when sel = "01" else
           s3 when sel = "10" else
           '0';
```



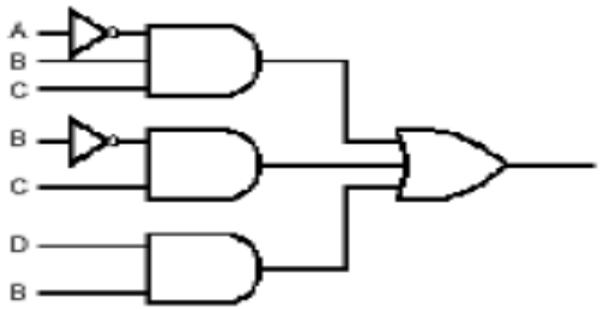
Inference van sequentiële schakelingen

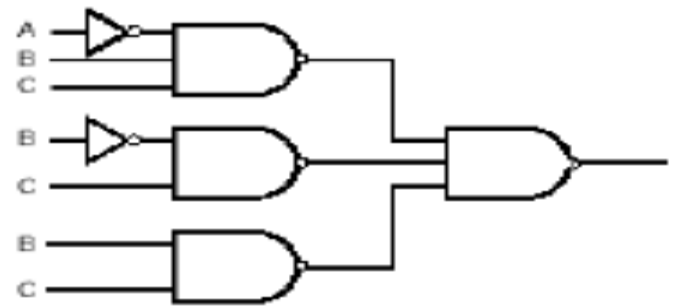
```
--
-- a simple conditional code block
--
if (sel = '0') then
z <= in1 nor in2;
end if;
```

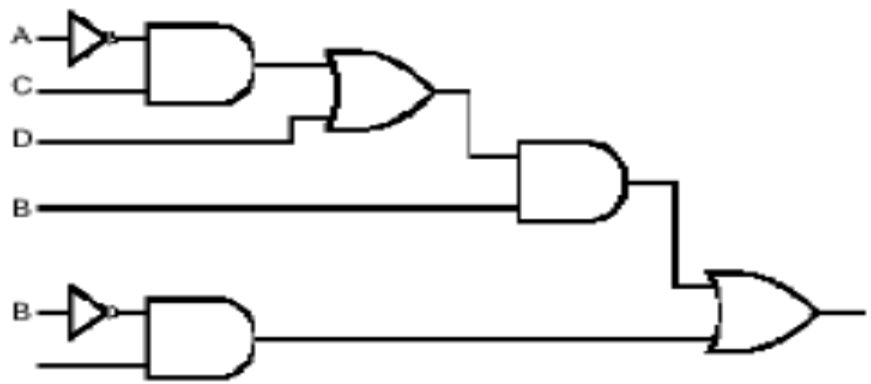


- Inference van sequentiële componenten is veel subtieler
- Resultaten zijn afhankelijk van de manier van schrijven van de VHDL code
- Inference van latches vs. flip-flops

Target primitives







$$\overline{A}BC + \overline{B}C + DB$$

Uiteindelijke resultaat is afhankelijk van bestaande componenten!

Inference from declarations

- Signalen worden vertaald naar:
  - Wires
  - latches/flip-flops
  - keuze afhankelijk van gebruik
- Gebruik van “types”:
  - voor synthese: bepaal de “bit widths” (expliciet of impliciet)

```
signal result: std_logic_vector (12 downto 0);
signal count: integer;
signal index: integer range 0 to 18;

type state_type is (state0, state1, state2, state3);
signal next_state: statetype;
```

- Geef hints om hardware te minimaliseren
- Denk aan hardware!!

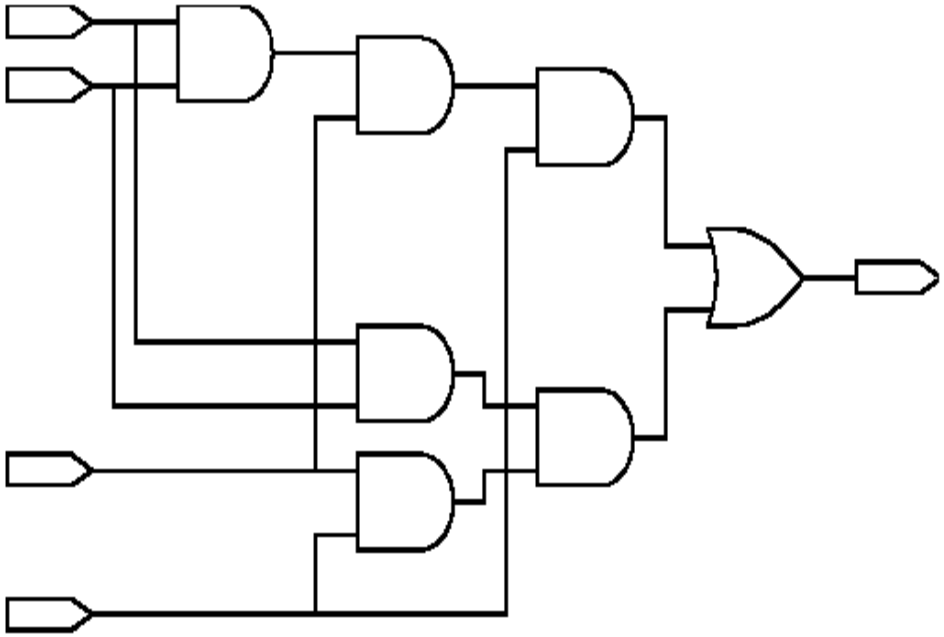
Inference vanuit Simple Concurrent Signal Assignment Statements

- Gedrag komt overeen met combinatorische circuits
  - Model produceert een signal assignment statement voor elk signaal in het circuit
  - Als de ingangen (RHS) veranderen wordt de uitgang opnieuw bepaald
- Operator inferencing:
  - Operaties, gate types
  - Interconnect (=verbindingen)
- Delay informatie wordt verwijderd

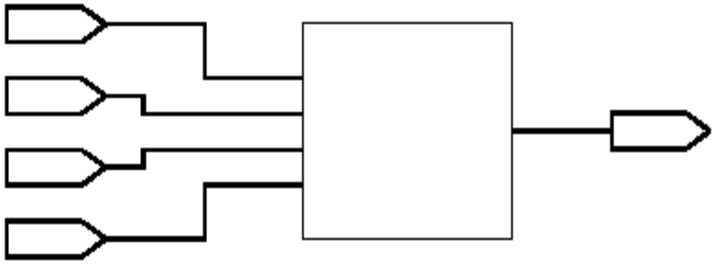
Inference vanuit Simple CSAs (voorbeeld 1)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity concurrent is
port (s, t, u, w: in std_logic;
      v: out std_logic);
end          concurrent;

architecture dataflow of concurrent is
signal s1, s2 : std_logic;
begin
L1: s1 <= s and t and u and w;
L2: s2 <= (s and t) and (u and w);
L3: v <= s1 or s2;
end                    dataflow;
```

Synthesized Gate level Implementation



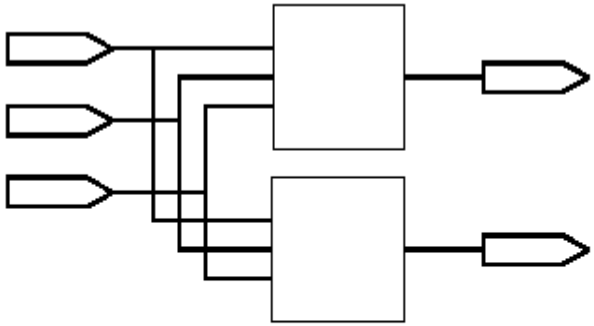
FPGA Implementation

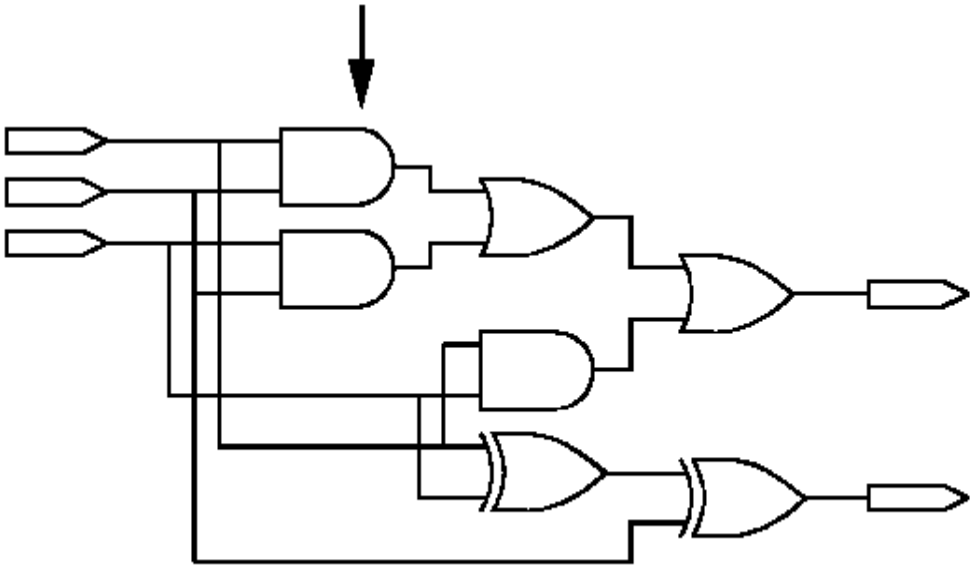


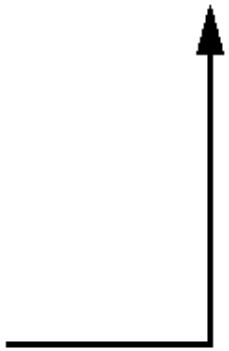
Let op het aantal gate niveaus!!

Inference vanuit Simple CSAs (voorbeeld 2)

```
architecture dataflow of full_adder is
signal s1, s2, s3: std_ulogic;
begin
sum <= in1 xor in2 xor c_in;
c_out <= (in1 and c_in) or (in2 and c_in) or (in1 and in2);
end dataflow;
```

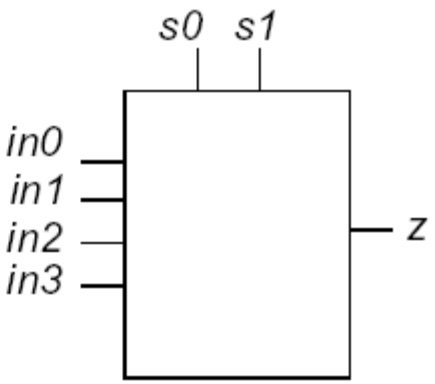






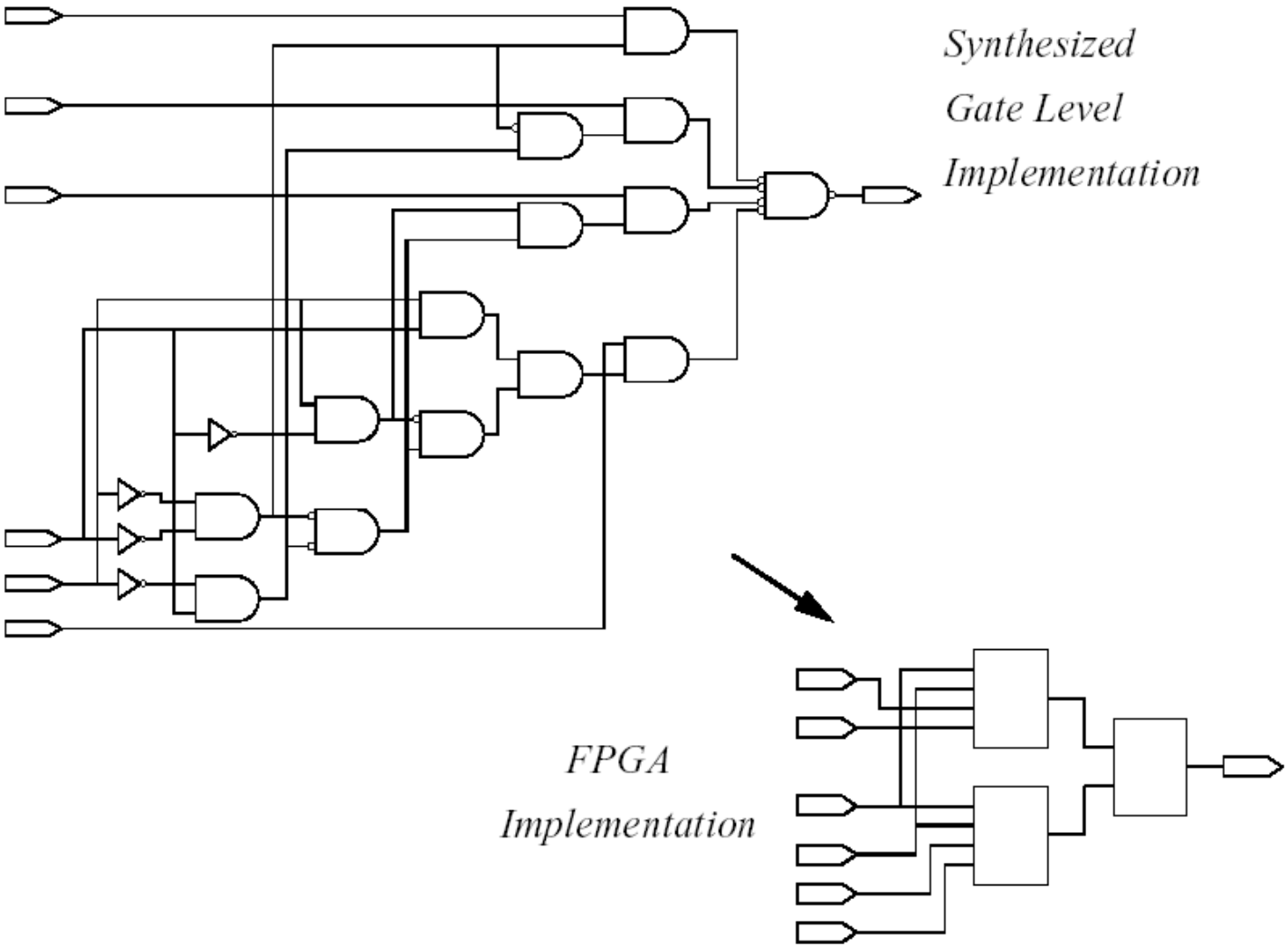
Inference vanuit Conditional CSAs

```
architecture behavioral of mux4 is
begin
z <=    in0 when s0 = '0' and s1 = '0' else
        in1 when s0 = '0' and s1 = '1' else
        in2 when s0 = '1' and s1 = '0' else
        in3 when s0 = '1' and s1 = '1' else
        '0';
end                 behavioral;
```



- Multiplexer logic
- Elke verandering van ingangssignalen resulteert in herberekening uitgang
- Veronderstelling van een “priority order”
- Hoe ziet dit eruit met gebruik van 4-input LUTs (denk FPGAs) en 6 ingangssignalen?

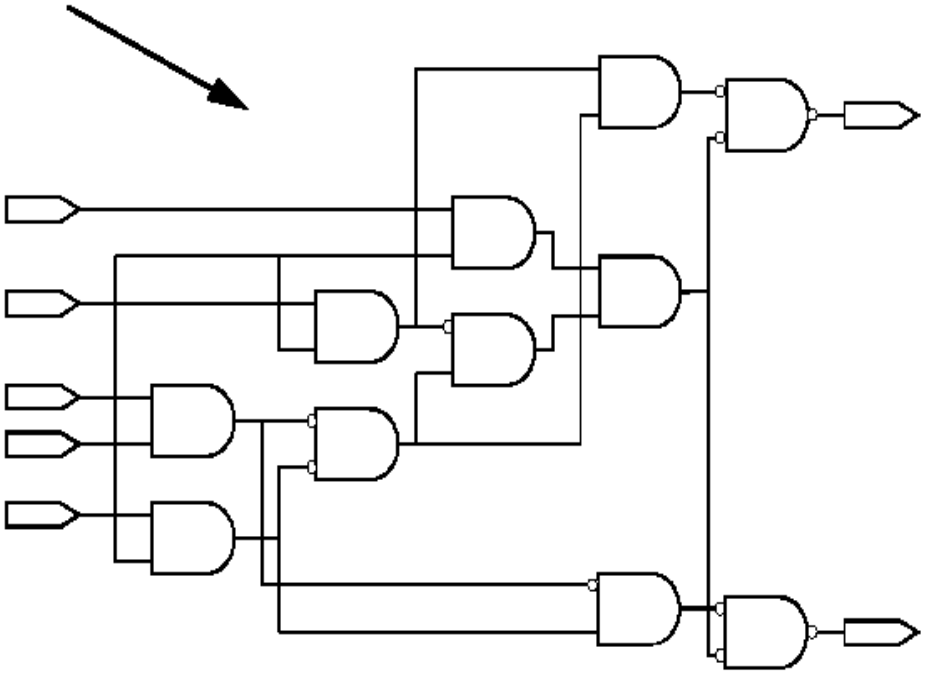
Inference vanuit Conditional CSAs



Voorbeeld: Priority Encoder

```
architecture behavior of priority is
begin
 z <= "00" when datain (0) = '1' and valid = '1' else
"10" when datain (2) = '1' and valid = '1'else
"01" when datain (1) = '1' and valid = '1'else
"11" when datain (3) = '1' and valid = '1'else
"00";
end                behavior;
```

- note generation of priority logic
- must be mapped to LUTs

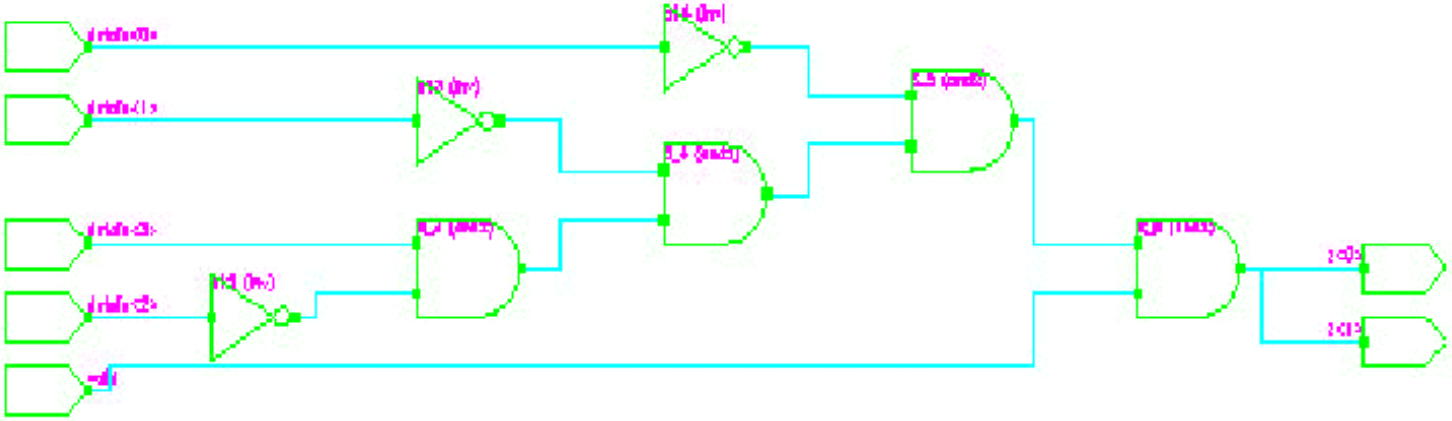


Synthese van “comparison logic”

- “Comparisons” die een betekenis hebben in simulatie hoeven niet per se betekenis te hebben in synthese
  - Bv. Don’t cares in “comparisons” retourneren altijd een false waarde!
- Denk altijd in werkelijke hardware wanneer je synthetiseerbare VHDL aan het schrijven bent
- “Equality tests” voor waarden anders dan 0/1 resulteren altijd in false!

Synthese en “Comparison logic”

```
architecture behavior of priority is
begin
 z <= “00” when datain = “---1” and valid = ‘1’ else
“10” when datain = “-100” and valid = ‘1’else
“01” when datain = “--10” and valid = ‘1’else
“11” when datain = “1000” and valid = ‘1’else
“00”;

end                    behavior;
```



Inference van Selected CSAs

- Alle “keuzes” moeten gelijktijdig worden geëvalueerd en alleen een kan waar zijn
- Geen “priority logic”

```
architecture behavioral-2 of mux4 is
begin
with Sel select
Z <= (In0          ) when “00”,
     (In1          ) when “01”,
     (In2          ) when “10”,
     (In3          ) when “11”
     (In3          ) when others;
end              behavioral;
```

All options must be covered and only one must be true!

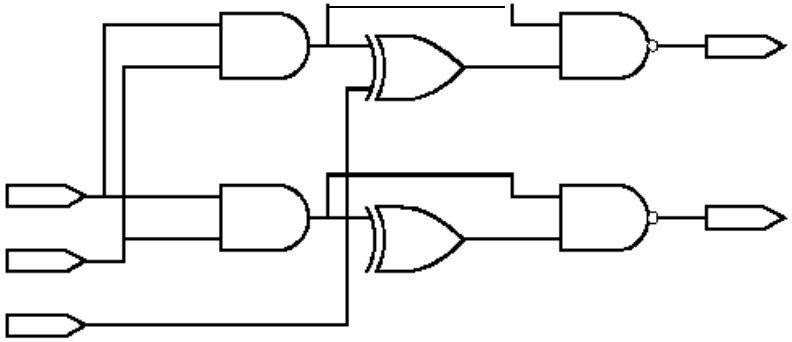
Synthese hints bij gebruik van CSAs

- Verschil simulatie en synthese door:
  - Delay statements
  - Comparison logic
- Geen initialisatie van declaraties – gebruik van expliciete resets nodig
- Gebruik ‘hints’ om synthese te sturen
- Selected vs. Conditional signal assignment statements:
  - “selected” resulteert in minder logic inference

Variabelen vs. signalen (voorbeeld)

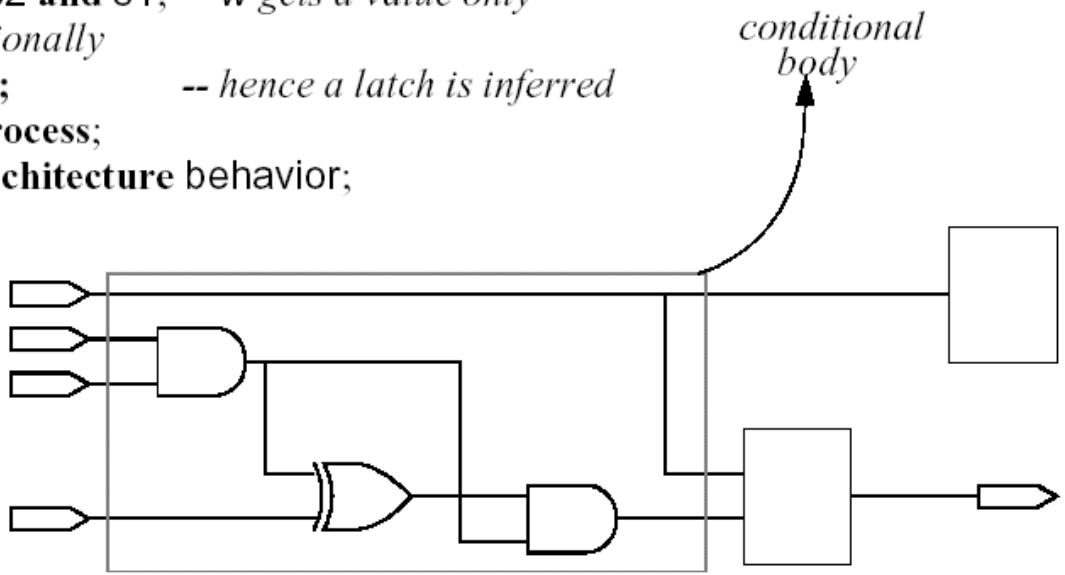
```
architecture behavior of stover is
signal sig_s1, sig_s2: std_logic;
begin
proc1: process (x, y, z) is -- Process 1 using
variables
variable var_s1, var_s2: std_logic;
begin
var_s1:= x and y;
var_s2:= var_s1 xor z;
res1 <= var_s1 and var_s2;
end process;

proc2: process (x, y, z) -- Process 2 using signals
begin
sig_s1 <= x and y;
sig_s2 <= sig_s1 xor z;
res2 <= sig_s1 and sig_s2;
end process;

end architecture behavior;
```

- simulation mismatch
- dependence analysis



Combinatorisch vs. sequentieel

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
if (sel = '1')then
s1:= x and z;
s2:= s1 xor y;
w <= s2 and s1; -- w gets a value only
conditionally
 end if;            -- hence a latch is inferred
end process;
end architecture behavior;
```



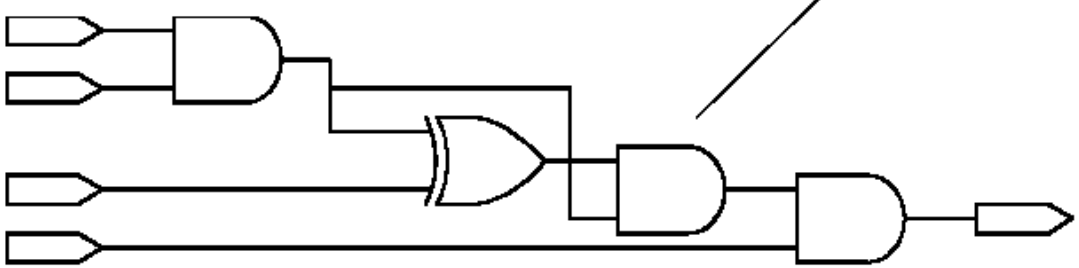
Vermijden van latches bij combinatorische schakelingen

- Zorg ervoor dat voor alle ingangscombinaties een uitgang wordt gedefinieerd:
  - Zorg altijd voor alle “else”-statements
  - Let op geneste structuren (if-then-elsif)
- Gebruik van initiële waardes (volgende slide)

Nogmaals hetzelfde voorbeeld

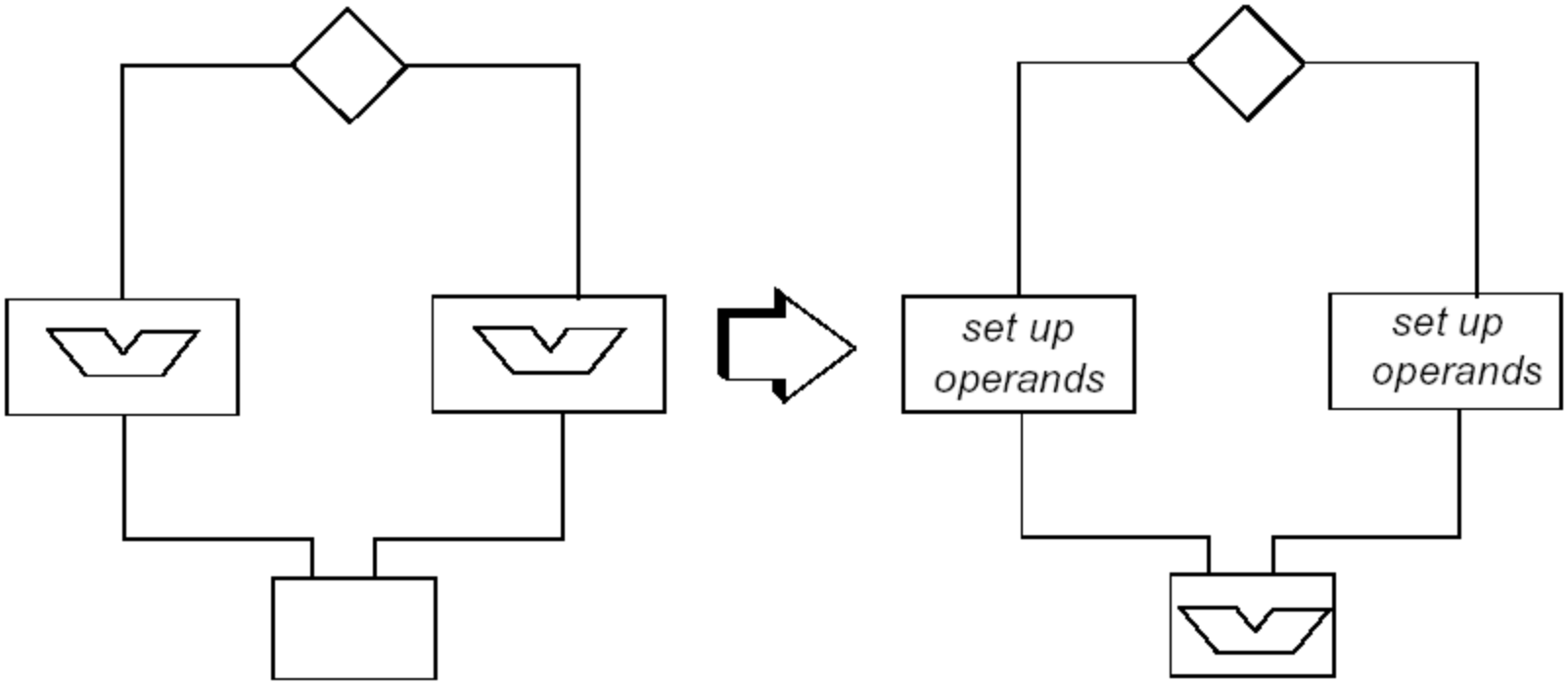
```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
w <= '0';          -- output signal set to a default value to avoid latch inference
if (sel = '1')then
s1:= x and z;      -- body generates combinational logic
s2:= s1 xor y;
w <= s2 and s1;
 end if;

end process;
end architecture behavior;
```

No latches!



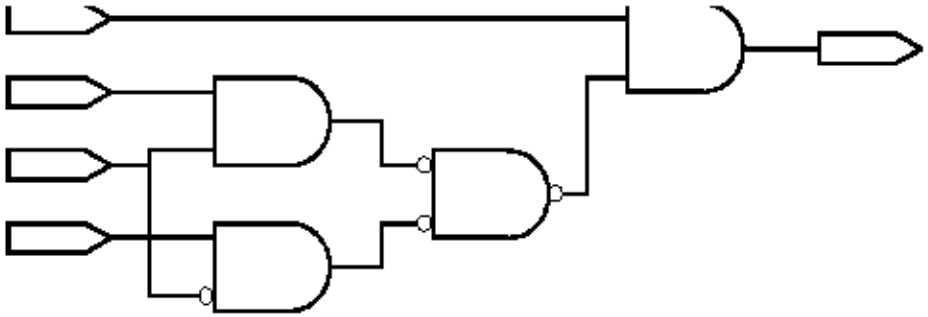
Beperken van hardware

- Move common operations (hardware) out of the branches
  - good programming practice in general
  - trade multiplexors for more expensive hardware



Beperken van hardware (2)

```
architecture behavior of inference is
begin
process (x, y, z, sel)
variable right: std_logic;
begin
if (sel = '1')then
right:= y;
 else
right:= z;
 end if;
w <= x and right;

end process;
end behavior;
```

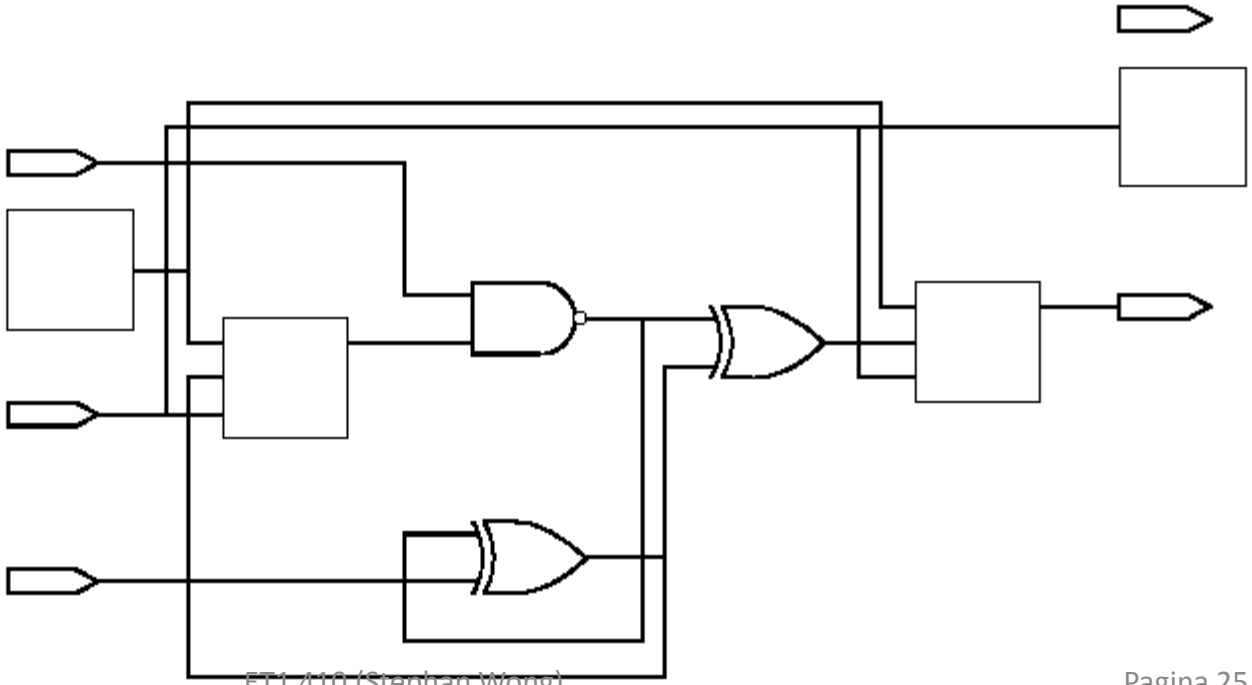


- Computation (logical AND) preceded by operand selection (mux)

Inferring Storage for Variables

```
architecture behavior of sig_var is
begin
process
variable var_s1, var_s2 :std_logic;
begin
wait until (rising_edge(clk));
var_s1 := x nand var_s2;
var_s2 := var_s1 xor y;
res <= var_s1 xor
var_s2;
end process;
end behavior;
```

- variables used before it is defined
- there exists an execution sequence(first) where use precedes definition



Latch vs. Flip Flop Inference

- Predicates in conditional expressions lead to latch inference

  if (sel = ‘1’) then...

- Edge detection expressions lead to flip flop inference

  if (rising_edge(clk)) then...

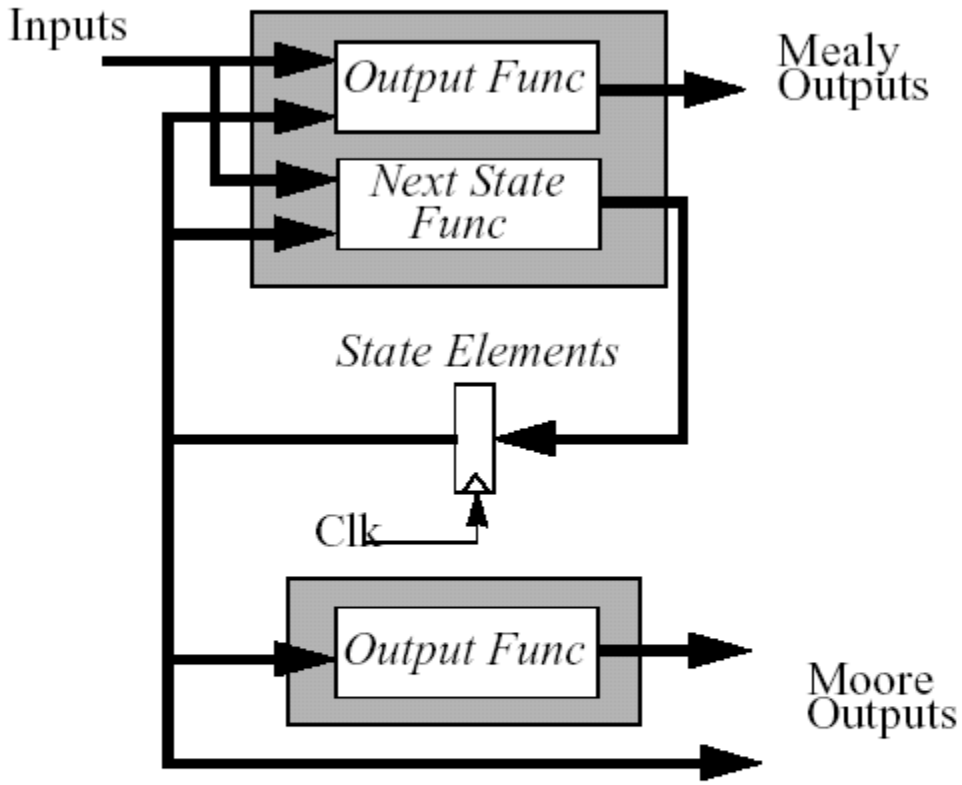
  if (clk’event and clk = ‘0’) then..

  if (clk’lastvalue = ‘0’ and clk = ‘1’ and clk’event) ..

  simulation semantics only

- Semantics must be consistent with available parts
  - latches vs. flip flops
  - trigger condition

Synthesis of State Machines



- Combinatorial component and sequential component
- Asynchronous and synchronous components
- Control inference through use of wait and edge detection expressions

State Encodings

- State encodings

State	Sequential	Gray Code	One Hot
0	000	000	00000001
1	001	001	00000010
2	010	011	00000100
3	011	010	00001000
4	100	110	00010000
5	101	111	00100000
6	110	101	01000000
7	111	100	10000000

- Source-level constructs for user supplied encodings
- Goal: optimize area or speed

Controlling Inference: Example

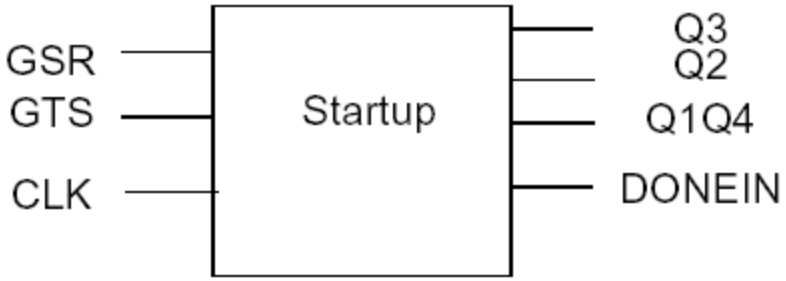
```
architecture behavioral of edge is
begin
process
begin
wait until (rising_edge(clk));
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
Bout <= 0;
end process;
end architecture behavioral;
```

```
architecture behavioral of edge is
begin
process (reset, clk) is
begin
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
if (rising_edge(clk)) then
Bout <= 0;
end if;
end process;
end architecture behavioral;
```

- control inference by use of the wait statement

Implementing Global Set/Reset in VHDL

- Use special on-chip circuitry
- GSR inferencing optimization
- Instantiate the following element to explicitly use this facility



- Connect your reset signal to this global net
- Note
  - both set/reset for a flip flop not supported
  - improves “routability” of designs
  - active low reset?

“Synthese”stappen (FPGA)

- Analyse (zelfde als bij simulatie)
- Synthese van ontwerp
- Mapping van ontwerp
- Place and Route
- Generatie van bitstream
- Programmeren van (FGPA)-chip