Academie voor Technology, Innovation & Society Delft
Academie voor ICT & Media

Digitale Systeem Engineering 1

Week 3 – Synthese, simuatie, testbenches, rekenen in VHDL

Jesse op den Brouw

DIGSE1/2018-2019

DE HAAGSE HOGESCHOOL

Synthese

- Synthese is het proces van het automatisch genereren van hardware uit een beschrijving.

- Aan het synthese-proces kunnen zogenaamde *constraints* worden opgegeven.

- Voorbeelden:
  - minimale werkfrequentie
  - maximaal logisch niveau
  - maximale oppervlakte (ASIC),
  - maximaal aantal cellen (FPGA)
  - maximale t_{pd} van klokflank naar output,
  - maximale dissipatie

DE HAAGSE HOGESCHOOL

Synthese

- Uiteraard moet de beschrijving echte hardware kunnen opleveren (denk weer aan simulatie...).
- De *synthesizer* probeert aan alle constraints te voldoen. Als dat lukt in het synthese-proces geslaagd.
- Lukt het niet, dan moeten de beschrijving en/of constraints worden aangepast.
- Beschrijving aanpassen: denk aan pipelining, retiming en *resource sharing*.

Synthese

- De synthesizer vertaalt de beschrijving eerst naar bekende structuren zoals poorten, multiplexer, optellers, vergelijkers, latches, flipflops.

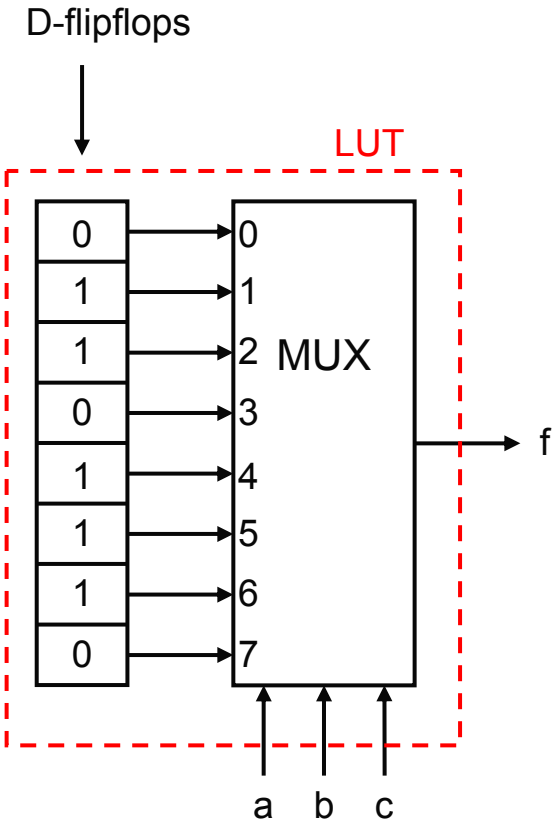
- Daarna zorgt de synthesizer dat deze structuren wordt gemapped op de beschikbare technologie.

- Structuren (primitives):
  - case, with : multiplexers
  - if, when else : priority encoders
  - ... ‘event and ... : flipflops
  - + - * / : adders, subtractors, ...

Technologie:
- Lookup Tables (LUT)
- D-flipflops
- on-chip multipliers
- on-chip memory

Synthese (lookup table)

- Met een lookup table (LUT) kan elke booleaanse functie worden gerealiseerd.
- Een LUT is niets anders dan een multiplexer waarvan de data-ingangen verbonden zijn met de constanten 0 en 1.
- De selectie-ingangen selecteren op enig moment één van de data-ingangen.
- Door voor de data-ingangen D-flipflops te plaatsen is de te realiseren functie te wijzigen (configureerbaar).

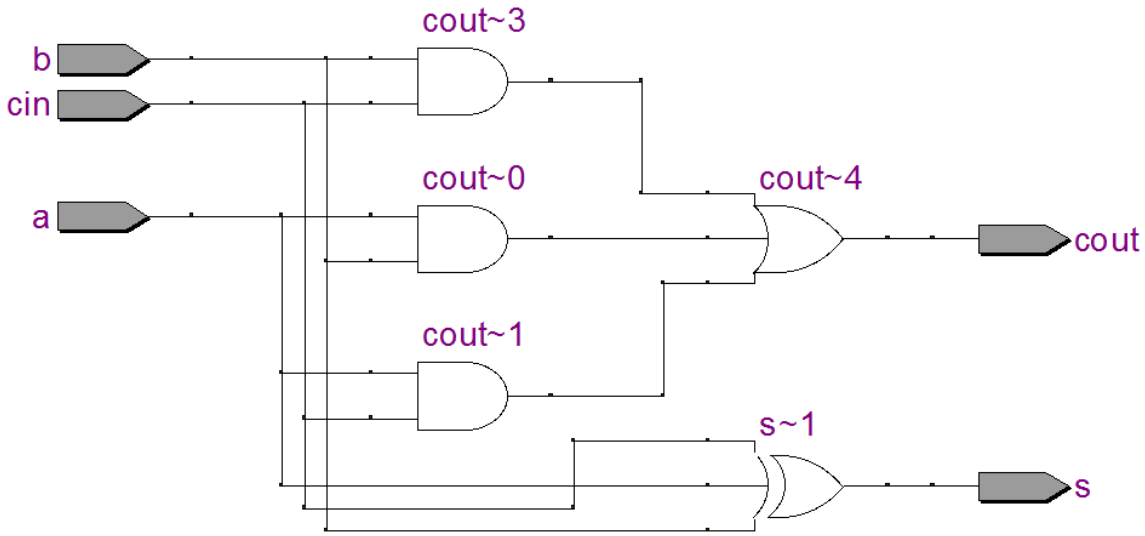


$$f_{a,b,c} = \sum (m_1, m_2, m_4, m_5, m_6)$$

Synthese logic

- Als voorbeeld een full adder:

```
s    <= a xor b xor cin;
cout <= (a and b) or (a and cin) or (b and cin);
```

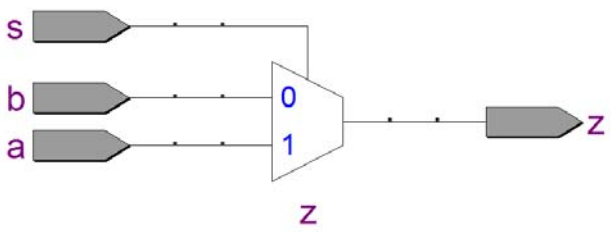


primitives

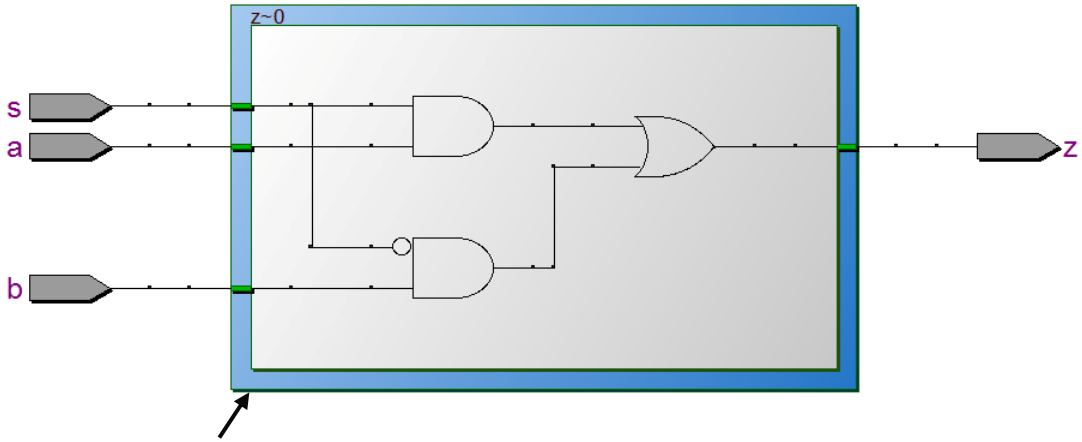
Synthese multiplexer

- Als voorbeeld een multiplexer:

```
process (a,b,s) is
begin
  if s = '1' then
    z <= a;
  else
    z <= b;
  end if;
end process;
```



primitives

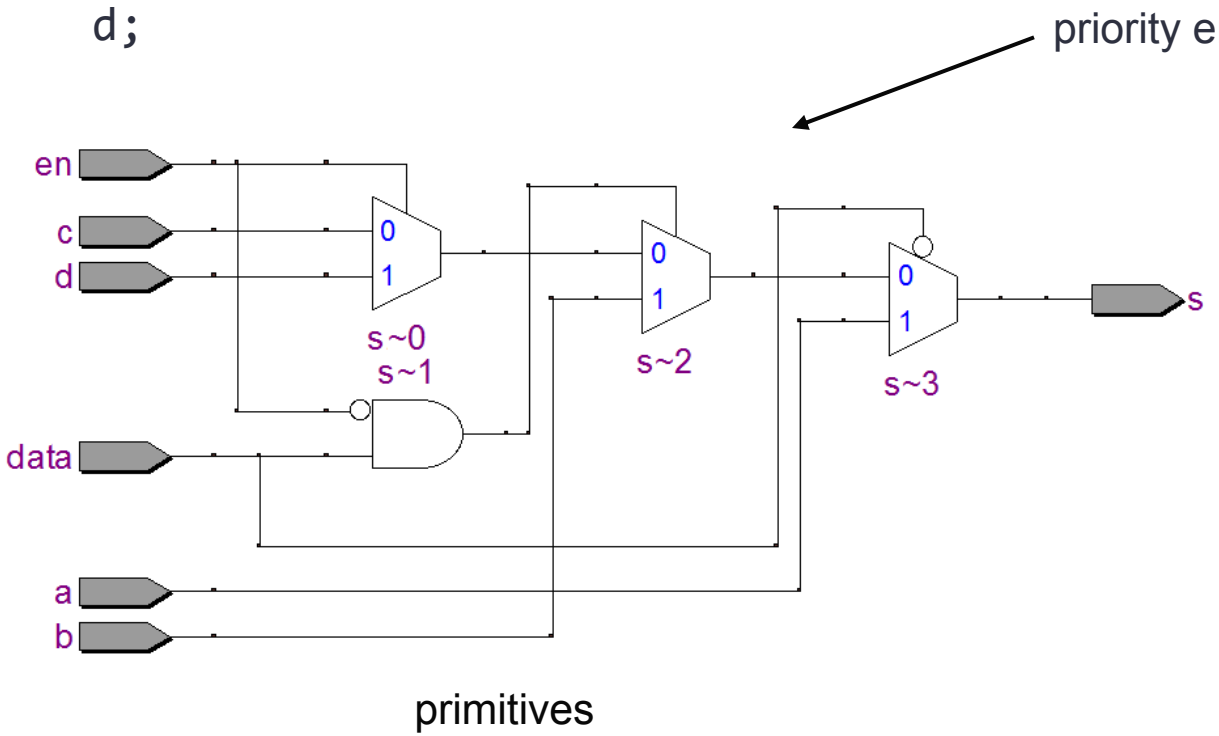


LUT technologie

Synthese CSA

- Als voorbeeld het gegeven statement:

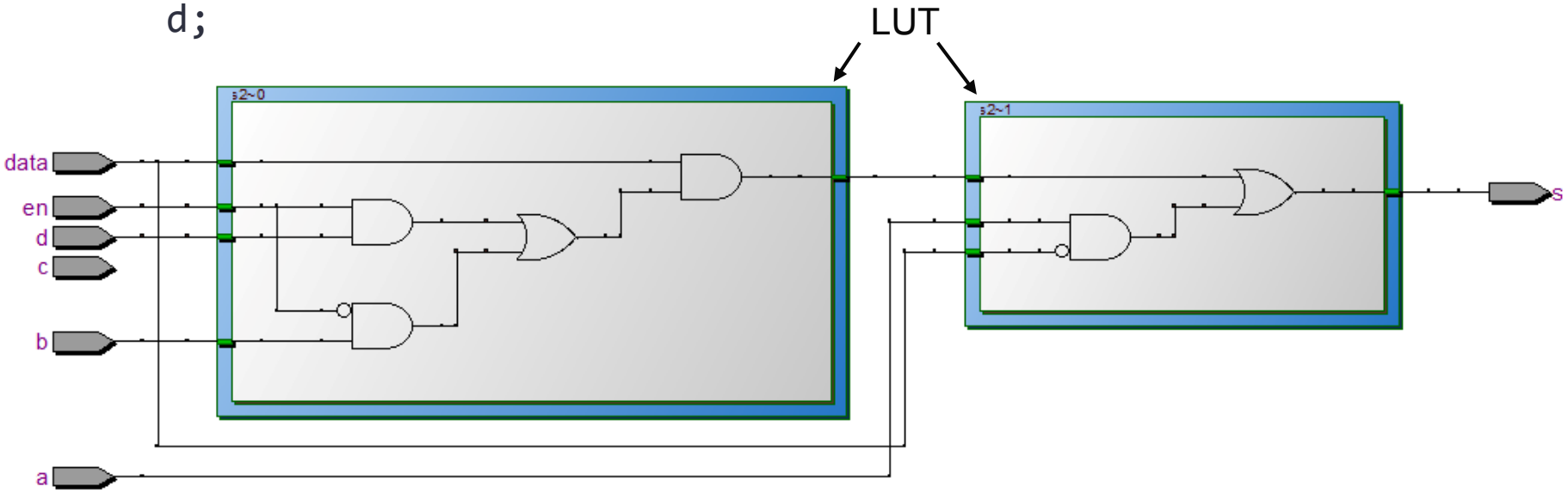
```
s <= a when data = '0' else
     b when data = '1' and en = '0' else
     c when en = '0' else
     d;
```



Synthese CSA

- Als voorbeeld de eerder gegeven statement:

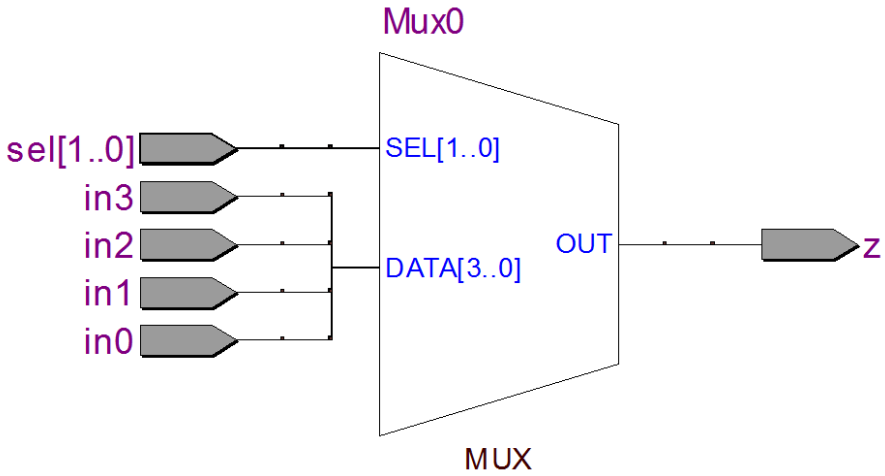
```
s <= a when data = '0' else
     b when data = '1' and en = '0' else
     c when en = '0' else
     d;
```



Synthese 4x1 mux met SSA

- Synthese van een 4x1 mux met Selected Signal Assignment:

```
architecture synthesis of mux4 is
begin
  with sel select
    z <= in0  when "00",
         in1  when "01",
         in2  when "10",
         in3  when "11",
         'X'  when others;
end synthesis;
```

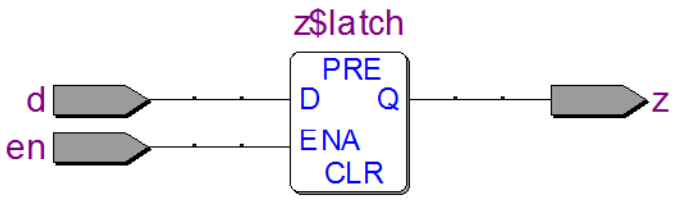


- When others wordt niet gesynthetiseerd, is bedoeld voor simulatie.

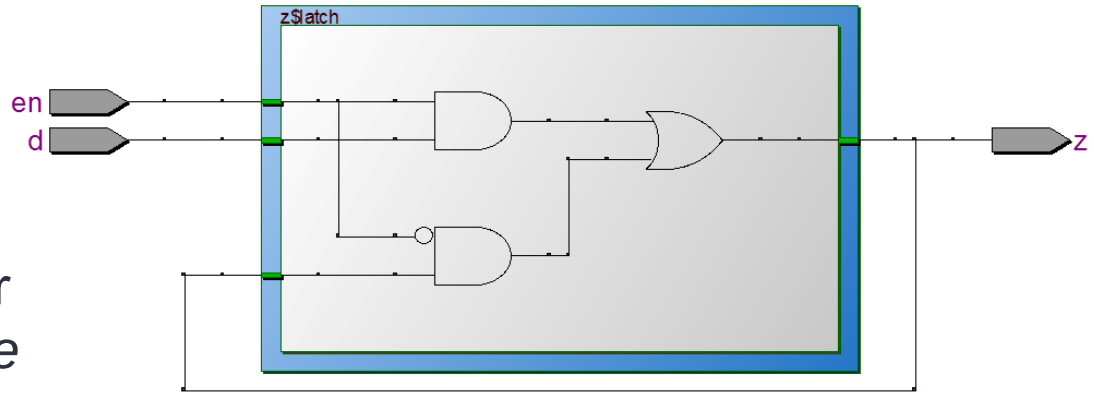
Synthese van een D-latch

- Beschrijving van een D-latch

```
process (en, d) is
begin
  if en = '1' then
    z <= d;
  end if;
end process;
```



primitives



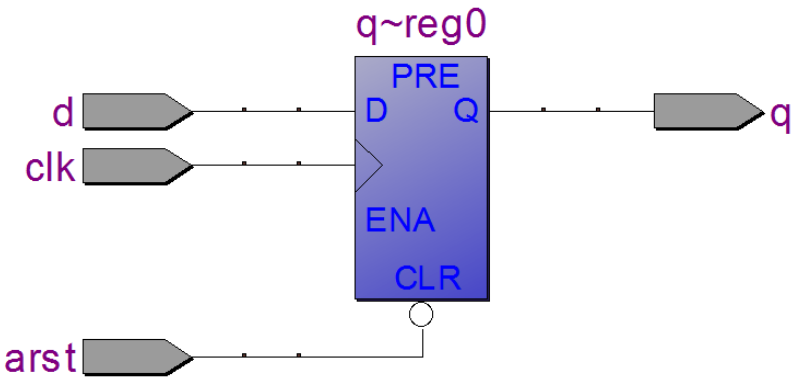
technologie

- Latch wordt gebouwd door logica met een *asynchrone feedback*.

Synthese D-flipflop met asynchrone reset

- Hieronder de beschrijving en hardware van een D-flipflop met actief lage asynchrone reset.

```
architecture synthesis of d_ff is
begin
  process (clk, arst) is
  begin
    if arst = '0' then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end synthesis;
```

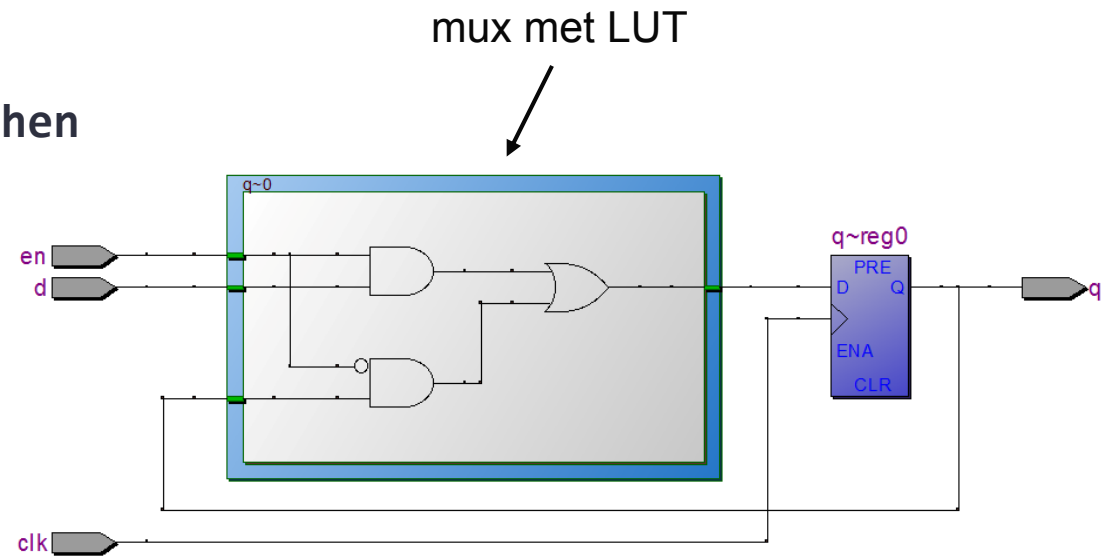


primitives

Synthese D-flipflop met enable

- Hieronder de beschrijving en synthese van een D-flipflop met enable.

```
process (clk) is
begin
  if rising_edge(clk) then
    if en = '1' then
      q <= d;
    end if;
  end if;
end process;
```

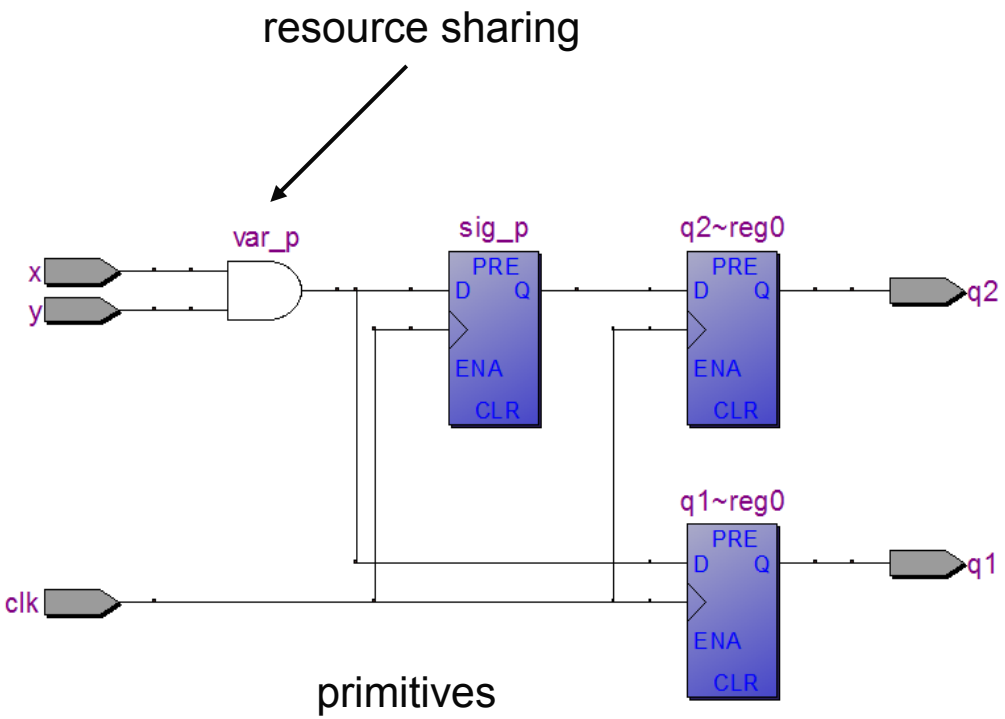


- Merk op dat de test op enable *binnen* de test voor de klokflank valt. Voor q wordt dan *kloksynchrone* logica gesynthetiseerd.

Synthese variable vs signal

- Het verschil tussen variable en signal wordt duidelijk bij synthese.

```
proc3: process (clk) is
variable var_p : std_logic;
begin
  if rising_edge(clk) then
    var_p := x and y;
    q1 <= var_p;
  end if;
end process;

proc4: process (clk) is
begin
  if rising_edge(clk) then
    sig_p <= x and y;
    q2 <= sig_p;
  end if;
end process;
```



Synthese variable vs signal

- Onderstaande processen functioneren identiek.

```
proc3: process (clk) is
variable var_p : std_logic;
begin
  if rising_edge(clk) then
    var_p := x and y;
    q1 <= var_p;
  end if;
end process;
```

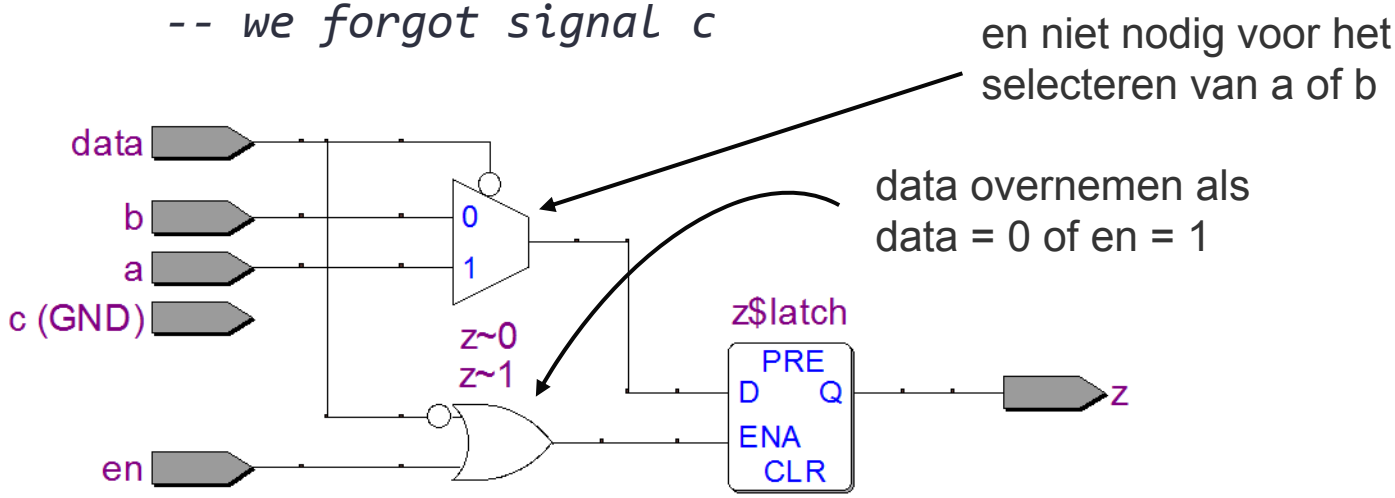
```
proc5: process (clk) is
variable var_p : std_logic;
begin
  if rising_edge(clk) then
    var_p := x and y;
  end if;
  q3 <= var_p;
end process;
```

- De toekenning `q3 <= var_p` mag ook buiten de klokflankbeschrijving, het resultaat wordt een flipflop voor q3.

Syntheseprobleem?

- Gegeven de volgende code. Wat wordt het syntheseresultaat?

```
z <= a when data = '0' else
     b when en = '1';
     -- we forgot signal c
```



Info (10041): Inferred latch for "z" at flipflops.vhd(76)

Warning: Latch z$latch has unsafe behavior

Warning: Ports D and ENA on the latch are fed by the same signal data

DET

- De volgende beschrijving is niet synthetiseerbaar:

```
proc6: process (clk) is
begin
    if clk'event then
        q3 <= x;
    end if;
end process;
```

- Wat voor hardware zou dit moeten opleveren?
  Double Edge Triggered Flipflop (DET).

DET

- De volgende beschrijving is niet synthetiseerbaar:

```
proc7: process (clk) is
begin
    if clk'event and clk = '1' then
        q3 <= x;
    end if;
    if clk'event and clk = '0' then
        q3 <= x;
    end if;
end process;
```

- Wat voor hardware zou dit moeten opleveren?

Tekenen schema

- Het tekenen van schema's vanuit VDHL-code maakt gebruik van bouwstenen op RTL-niveau:

- Registers (met enable, ook schuifregisters)
- Multiplexers
- Rekenkundige eenheden (optellers, aftrekkers, ...)
- Vergelijkers (gelijk, groter, kleiner, ...)
- Tellers
- Flipflops, losse poorten

- Probeer op een zo hoog mogelijk niveau te blijven, dat direct volgt uit de VHDL-beschrijving.

Opgaven

- Geef het schema van onderstaande VHDL-codes. Maak gebruik van multiplexers en poorten.

```
if sel = '1' then
   y <= a and b;
else
   y <= a or b;
end if;
```

```
if x = y then
   z <= '1';
else
   z <= '0';
end if;
```

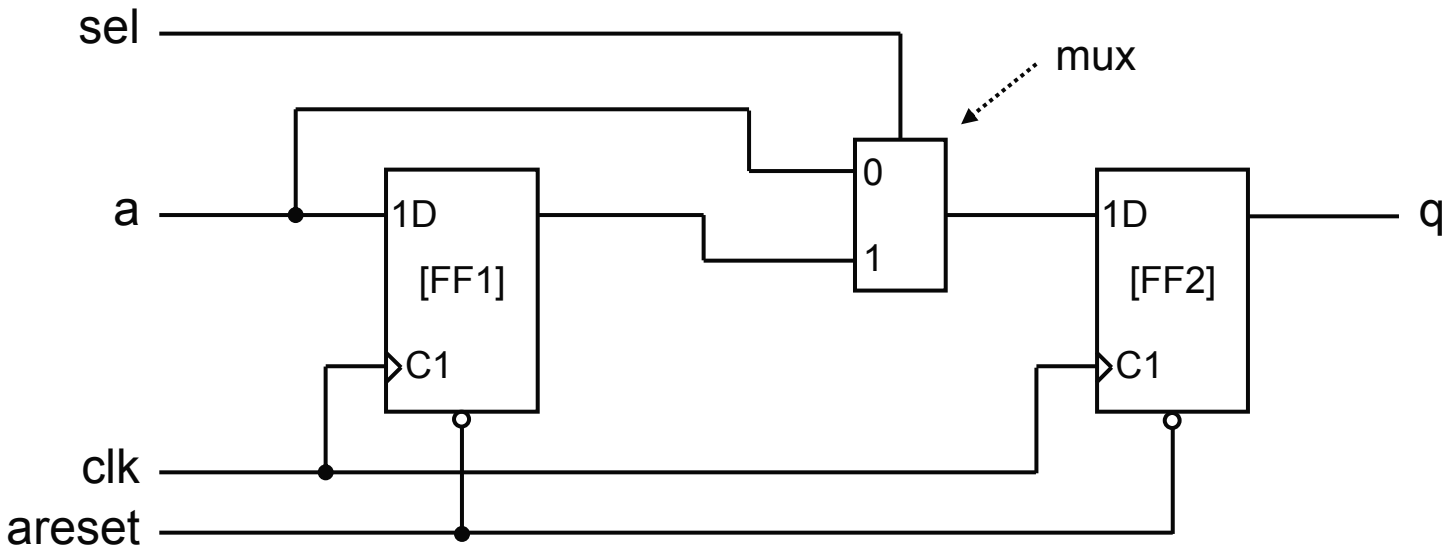
- Idem, maak gebruik van flipflops

```
if rising_edge(clk) then
   ff1 <= x;
   if ff1 = '1' and x = '0' then ff2 <= '1';
   else                          ff2 <= '0';
   end if;
end if;
```

DE HAAGSE HOGESCHOOL

Opgave

- Geef de complete VHDL-beschrijving van onderstaand schema.



Opgave

- Geef het schema van de onderstaande code. Probeer zo dicht mogelijk de originele code te volgen.

```
signal x : bit_vector (7 downto 0);
signal q : bit;
...
process (x) is                      -- sensitive for x
variable p : bit;                   -- only p, no need to declare i
begin
  p := '0';                         -- initialize to 0
  for i in 7 downto 0 loop          -- i = 7,6,5,4,3,2,1,0
    p := p or x(i);
  end loop;
  q <= not p;                       -- signal assignment
end process;
```

Testbenches

- VHDL is oorspronkelijk bedoeld als *simulatietaal*. Dat is te merken aan de vele taalconstructies die niet voor synthese bedoeld zijn (wait, after, transport, pointers, files).
- Het simuleren van een VHDL-beschrijving gebeurt met behulp van een *simulator*.
- Naast simuleren is het mogelijk een VHDL-beschrijving automatisch te *testen*.
- Bekende simulator: ModelSim.

Omgeving

- De simulatie-omgeving biedt alle faciliteiten om te simuleren en te testen:

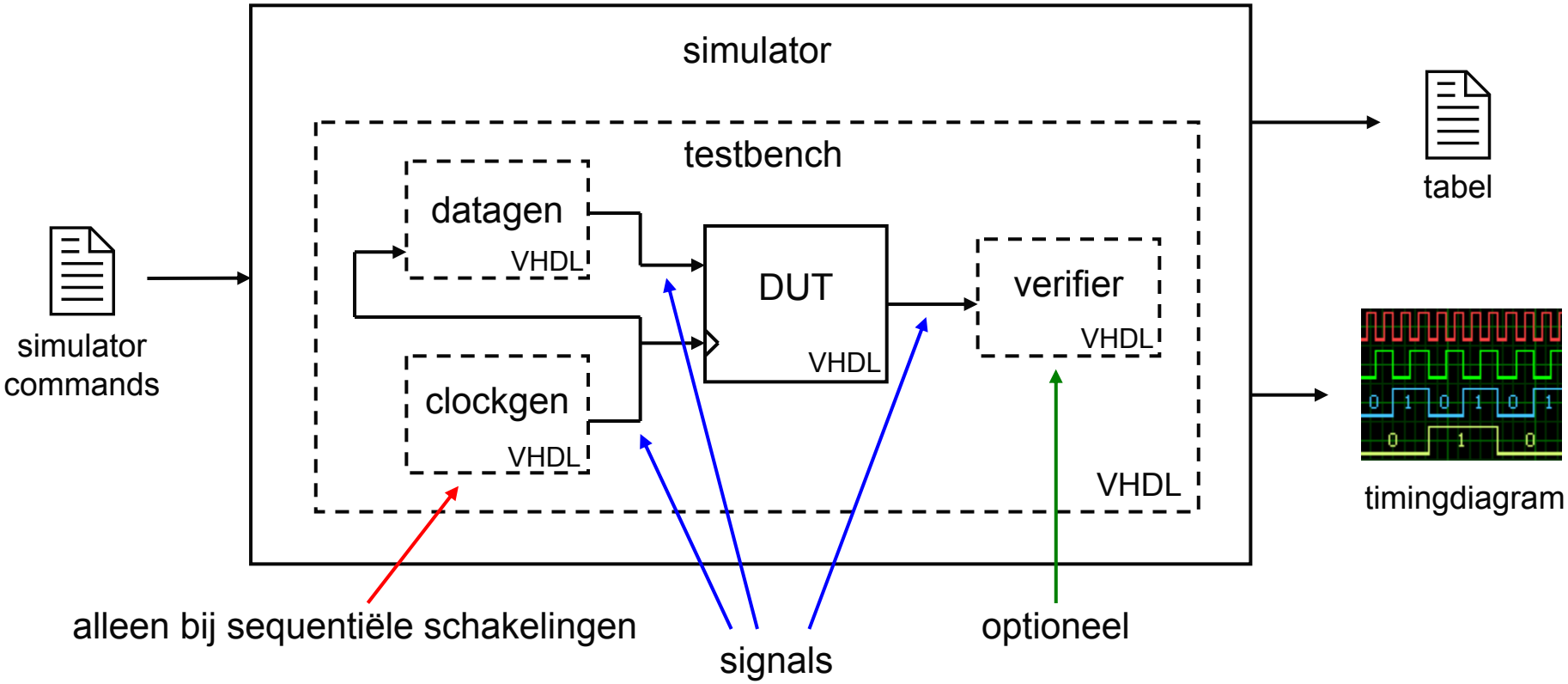
- aanbieden van testsignalen (*stimulus/stimuli*).
- afbeelden van de resultaten van de simulatie.
- het automatisch testen van de resultaten t.o.v. een referentiemodel.

- Het simuleren en testen gebeurt met een *testbench:*
  - testbench: een VHDL-beschrijving die signalen genereert om een (andere) VHDL-beschrijving te simuleren.

- De simulator wordt aangestuurd door commando's.

Testbenches

- Hieronder is de simulatie- en testomgeving schematisch weergegeven.



DUT = Design Under Test

Testbenches

- Clockgen genereert het kloksignaal dat zowel voor de DUT als datagen gebruikt wordt.
- Datagen genereert data-stimuli voor de DUT (inclusief resetsignaal).
- Verifier verifieert de uitkomst van de DUT met een bekend en werkend model. Deze beschrijving is optioneel.
- Alle onderdelen binnen de testbench zijn geschreven in VHDL.
- De onderdelen wisselen gegevens uit via signals.

Kloksignaal

- Bij een sequentiële VHDL-beschrijving is een kloksignaal nodig.
- Meestal is dit een symmetrisch signaal, maar dat is niet verplicht.
- Een kloksignaal kan beschreven worden door toekenningen en waits in een eeuwig durende lus:

```
clockgen: process is  -- no sensitivity list
begin
  clk <= '0';         -- clock 50 ns low
  wait for 50 ns;
  clk <= '1';         -- clock 50 ns high
  wait for 50 ns;
end process clockgen;
```

DE HAAGSE HOGESCHOOL

Datasignalen

- Met de datasignalen worden waarden aan de ingangen voor de DUT toegekend. Dat kan door middel van toekenningen en waits.

- Toekenningen aan de ingangssignalen kunnen gedaan worden op de bekende wijze:

```
x <= '0';
bus <= "1101";
```

- De statement `wait for` kan gebruikt worden om een process te laten wachten (suspend). De simulatietijd loopt wel door.

```
constant Tperiod : time := 50 ns;
wait for 10 ns;
wait for Tperiod;
```

DE HAAGSE HOGESCHOOL

Synchonisatie, loops

- De statement `wait until` kan gebruikt worden om te *synchroniseren* op het signaal, bijvoorbeeld een kloksignaal:

```
wait until clk = '1';  -- wait for positive edge
wait until clk = '0';  -- wait for negative edge
```

- Door het gebruik van loops (for, while) kunnen herhalingen beschreven worden.

```
while index > 0 loop ... end loop;
for all_values in 0 to 255 loop... end loop;
```

Gesynchroniseerde stimuli

- Bij het simuleren van sequentiële schakelingen is het handig om stimuli te synchroniseren met de klok. Dat kan eenvoudig met `wait until`:

```
x <= '0';                -- set data
wait until clk = '1';    -- synchronize on positive edge
wait for 10 ns;          -- slightly after edge

x <= '1';                -- set data
wait until clk = '1';    -- synchronize on positive edge
wait for 10 ns;          -- slightly after edge
```

- De kleine vertraging zorgt ervoor dat de nieuwe stimuli iets later aangeboden worden zodat ze goed zichtbaar zijn in het timingdiagram.

Reset

- Het reset-signaal wordt als eerste beschreven zodat de schakeling naar een gedefinieerde toestand gaat:

```
datagen: process is          -- no sensitivity list
begin
  x <= '0';                  -- data 0
  areset <= '1';             -- activate reset
  wait for 100 ns;           -- reset 100 ns active

  areset <= '0';             -- deactivate reset
  wait for 50 ns;            -- wait a bit ...
  ...
```

Testbench T-flipflop

- Op de volgende slides wordt een T-flipflop besproken.

- De testbench voor de T-flipflop is als volgt opgebouwd:
  - lege entity-beschrijving (geen port)
  - declaratie signalen die gestimuleerd en gemonitord moeten worden door de simulator
  - component-definitie (de DUT)
  - component-instantiëring
  - de feitelijke testbench (code met stimulus-opdrachten)

- Om het simulatie-proces goed te laten verlopen worden commando's aangeboden aan de simulator d.m.v. een script:
  - compile-opdrachten voor de DUT en testbench
  - simulatie-opdracht voor testbench
  - afbeeldingsopdrachten voor grafische interface

Simulatie T-flipflop (1)

- Een T-flipflop is een flipflop die van stand verandert als de stuuringang (T) logisch ‘1’ is. Als de stuuringang logisch ‘0’ is, blijft de T-flipflop de huidige stand vasthouden.

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

entity tflipflop is
port (clk    : in std_logic;
      areset : in std_logic;
      t      : in std_logic;
      q      : out std_logic
     );
end entity tflipflop;

architecture behavioral of tflipflop is
signal q_int : std_logic;  -- internal signal
begin
```

Simulatie T-flipflop (2)

- Het gedrag van een T-flipflop kan als volgt beschreven worden.

```
   process (clk, areset) is
   begin
      if areset = '1' then              -- reset is active high
         q_int <= '0';
      elsif rising_edge(clk) then       -- on posedge of clock
         if t = '1' then                -- if T = 1 ...
            q_int <= not q_int;         -- ... then toggle ...
         end if;
      end if;
   end process;

   q <= q_int;                          -- copy internal to external

end architecture behavioral;
```

Testbench T-flipflop (1)

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

entity tb_tflipflop is                              -- empty entity
end entity tb_tflipflop;

architecture sim of tb_tflipflop is
constant Tperiod : time := 20 ns;                   -- 50 MHz
constant small_delay : time := Tperiod / 10;        -- 2 ns
signal clk    : std_logic;                          -- monitoring signals
signal areset : std_logic;
signal t      : std_logic;
signal q      : std_logic;

component tflipflop is
port (clk    : in std_logic;
      areset : in std_logic;
      t      : in std_logic;
      q      : out std_logic);
end component tflipflop;
...
```

Testbench T-flipflop (2)

```
...
begin

   -- Instantiate a T flipflop and map the port signals
   -- to the monitoring signals
   dut : tflipflop
   port map (clk => clk, areset => areset, t => t, q => q);

   clockgen: process is   -- no sensitivity list
   begin
      clk <= '0';         -- clock has 50% Duty Cyle
      wait for Tperiod/2;
      clk <= '1';
      wait for Tperiod/2;
   end process clockgen;
...
```

Testbench T-flipflop (3)

```
...
datagen: process is                -- no sensitivity list
   begin
      t <= '0';
      areset <= '1';
      wait for 2*Tperiod;          -- wait for two clock cycles

      areset <= '0';
      wait for 2*Tperiod;
      wait until clk = '1';        -- synchronize on positive edge
      wait for small_delay;        -- and wait a bit

      t <= '1';                    -- set value
      wait until clk = '1';        -- wait for three clocks
      wait until clk = '1';
      wait until clk = '1';
      wait for small_delay;

      t <= '0';
      wait;                        -- wait forever

   end process;

end architecture sim;
```

DE HAAGSE HOGESCHOOL

Commando’s

- De simulator kent een groot aantal opdrachten.
- Beheersopdrachten
  - Manipulatie van mappen, bestanden, afsluiten simulator, cd, do
- Compile- en simulatie-opdrachten
  - vcom, vlog, vsim, run
- Manipulatie signalen
  - add {wave,list}, force, view {signals,process,structure}

Commando's

- De commando's kunnen interactief worden gegeven of worden uitgevoerd via een script.
- Een script is een bestand waarin opdrachten staan die sequentiëel worden uitgevoerd. Als een opdracht tot een foutmelding leidt, stopt de verwerking.
- Een script wordt in Modelsim een do-file genoemd.
- De commando-interpreter heeft als basis *tcl* ('tickle'), een scripttaal met veel mogelijkheden (if, for, while, functies etc).

Command script T-flipflop (1)

```
# Turn on transcript
transcript on

# Set up RTL work library
if {[file exists rtl_work]} {
  vdel -lib rtl_work -all
}

# Create RTL work library and use it as work
vlib rtl_work
vmap work rtl_work

# Compile VHDL description of T flip-flop
vcom -93 -work work tfliflop.vhd

# Compile VHDL testbench of T flip-flop
vcom -93 -work work tb_tflipflop.vhd
```

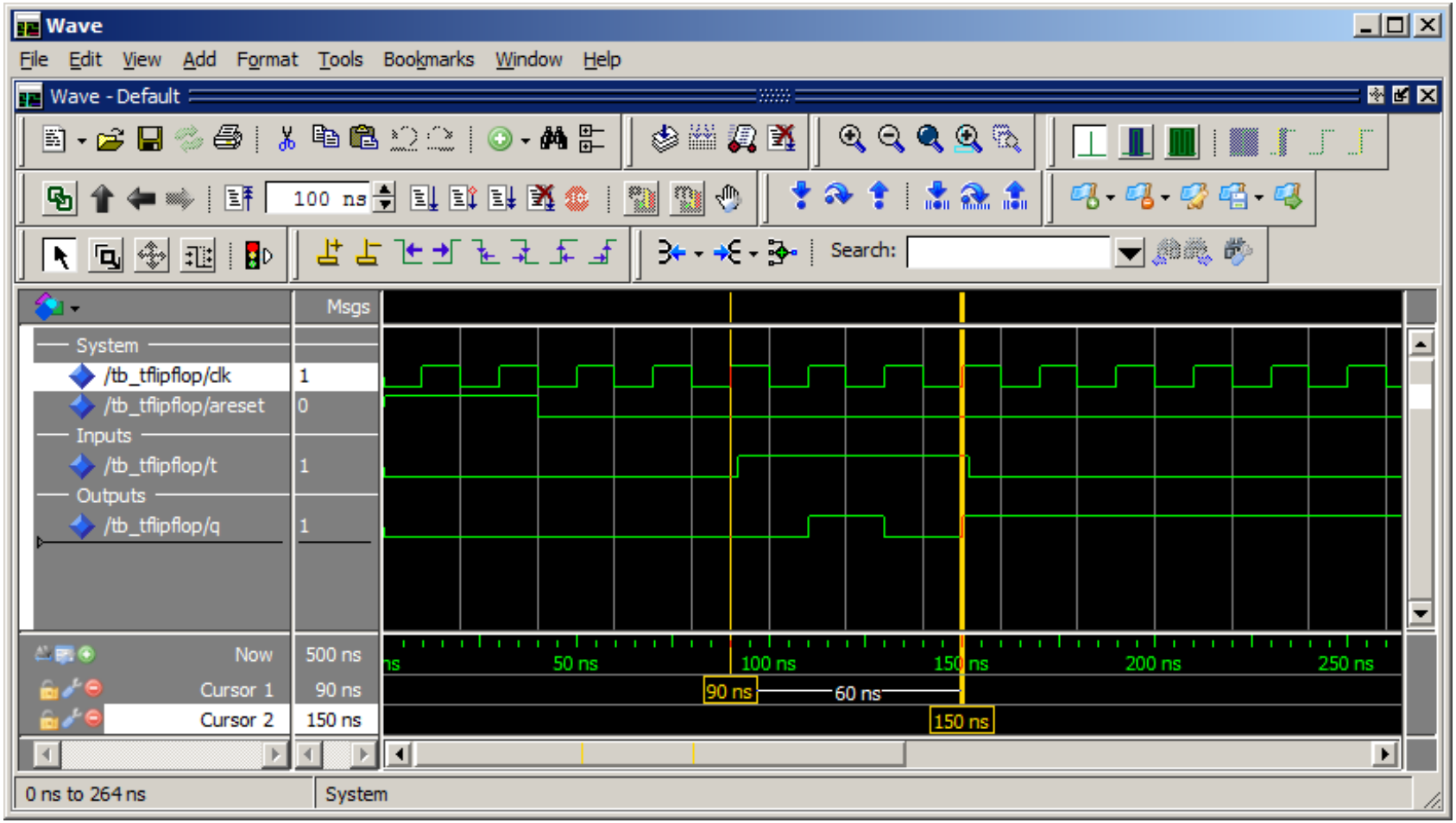
Command script T-flipflop (2)

```
# Set up simulation with the testbench as top level
#   -t 1ns  =  1 ns time step
#   -L ...  =  use library ...
vsim -t 1ns -L rtl_work -L work tb_tflipflop

# Log all signals in the design, good if the number
# of signals is small.
add log -r *

# Add a number of signals of the simulated design
# to the tabular list (text oriented)
add list clk
add list areset
add list t
add list q
```

Command script T-flipflop (3)

```
# Add a number of signals of the simulated design
# to the waveform list (GUI oriented)
add wave -divider “System“
add wave clk
add wave areset
add wave -divider "Inputs"
add wave t
add wave -divider "Outputs"
add wave q

# Open List and Waveform window
view list
view wave

# Run simulation for 500 ns
run 500 ns
```

Simulatieresultaat T-flipflop



DE HAAGSE HOGESCHOOL

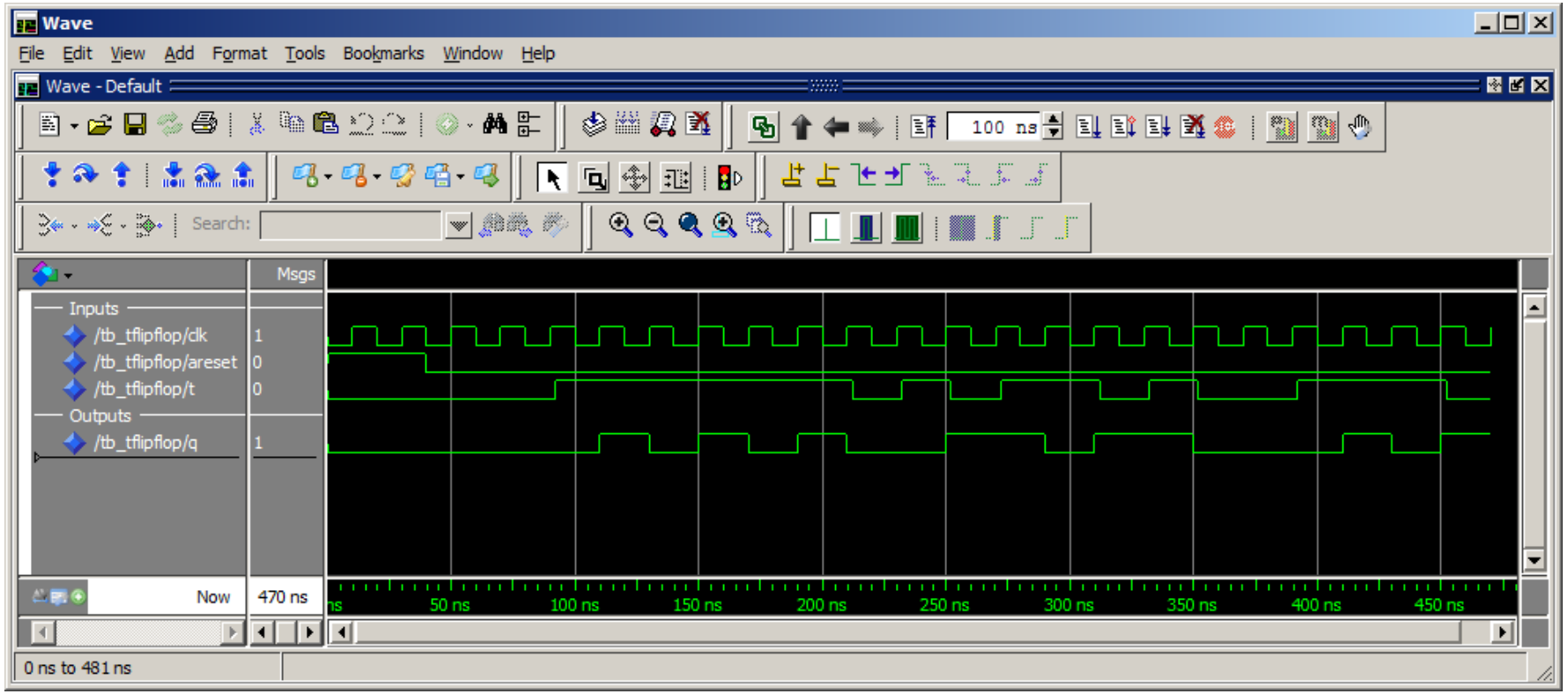
Alternatieve datagen

- VHDL is een taal met veel constructies (een rijke taal).
- Het is mogelijk om array's te gebruiken en lussen te construeren.
- Deze mogelijkheden kunnen goed ingezet worden bij testbenches.
- Een string (array van characters) wordt geladen met een bitpatroon.
- Eén voor één worden de bits toegekend aan de ingang van de T-flipflop.

Alternatieve datagen

```
datagen: process is

-- the valuations for the T-flipflop as a string
constant t_i_s : std_logic_vector := "11111101011010011110";
-- range is from 0 to 18 (19 elements)
variable t_i : std_logic_vector(t_i_s'range) := t_i_s;
begin

   -- reset omitted for clarity

   for index in t_i'range loop   -- loop through all input bits
      t <= t_i(index);           -- assign to flipflop input
      wait until clk = '1';
      wait for small_delay;
   end loop;

   wait;
end process;
```

Simulatieresultaat alternatieve datagen



Testbench full adder

- Op de volgende slides wordt een full adder besproken.

- De testbench voor de full adder is als volgt opgebouwd:
  - lege entity-beschrijving (geen port)
  - declaratie signalen die gestimuleerd en gemonitord moeten worden door de simulator
  - component-definitie (de DUT)
  - component-instantiëring
  - de feitelijke testbench (code met stimulus-opdrachten)

- Om het simulatie-proces goed te laten verlopen worden commando's aangeboden aan de simulator d.m.v. een script:
  - compile-opdrachten voor de DUT en testbench
  - simulatie-opdracht voor testbench
  - afbeeldingsopdrachten voor grafische interface

Beschrijving full adder

- De beschrijving van de te simuleren full adder:

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

entity full_adder is
  port (a, b, cin : in std_logic;
        s, cout   : out std_logic
  );
end entity full_adder;

architecture logic of full_adder is
begin
  s    <= a xor b xor cin;
  cout <= (a and b) or (a and cin) or (b and cin);
end logic;
```

Testbench full adder (1)

- De testbench in delen:

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

entity tb_full_adder is
-- empty entity
end tb_full_adder;

architecture testbench of tb_full_adder is

-- signals to be monitored
signal a : std_logic;
signal b : std_logic;
signal cin : std_logic;
signal s : std_logic;
signal cout : std_logic;
```

Testbench full adder (2)

```
component full_adder    -- the full adder as component
   port (
     a : in std_logic;
     b : in std_logic;
     cin : in std_logic;
     s : out std_logic;
     cout : out std_logic
   );
end component;

begin

   dut : full_adder   -- instantiate full adder
   port map (a => a, b => b, cin => cin, s => s, cout => cout);
```

Testbench full adder (3)

```
datagen : process      -- no sensitivity list
begin
  a <= '0'; b <= '0'; cin <= '0';    -- set stimuli (delta delay!)
  wait for 10 ns;                    -- wait a while
  a <= '0'; b <= '0'; cin <= '1';
  wait for 10 ns;
  a <= '0'; b <= '1'; cin <= '0';
  wait for 10 ns;
  a <= '0'; b <= '1'; cin <= '1';
  wait for 10 ns;
  a <= '1'; b <= '0'; cin <= '0';
  wait for 10 ns;
  a <= '1'; b <= '0'; cin <= '1';
  wait for 10 ns;
  a <= '1'; b <= '1'; cin <= '0';
  wait for 10 ns;
  a <= '1'; b <= '1'; cin <= '1';
  wait;                              -- wait forever
end process datagen;
```

DE HAAGSE HOGESCHOOL

Command script full adder (1)

```
# Turn on transcript
transcript on

# Set up RTL work library
if {[file exists rtl_work]} {
  vdel -lib rtl_work -all
}

# Create RTL work library and use it as work
vlib rtl_work
vmap work rtl_work

# Compile full adder entity with VHDL 1993 standard
vcom -93 -work work full_adder.vhd

# Compile testbench full adder entity with VHDL 1993 standard
vcom -93 -work work tb_full_adder.vhd
```

Command script full adder (2)

```
# Set up simulation with the testbench as top level
#   -t 1ns  =  1 ns time step
#   -L ...  =  use library ...
vsim -t 1ns -L rtl_work -L work tb_full_adder

# Log all signals in the design, good if the number
# of signals is small.
add log -r *

# Add a number of signals of the simulated design
# to the tabular list (text oriented)
add list a
add list b
add list cin
add list s
add list cout
```

Command script full adder (3)

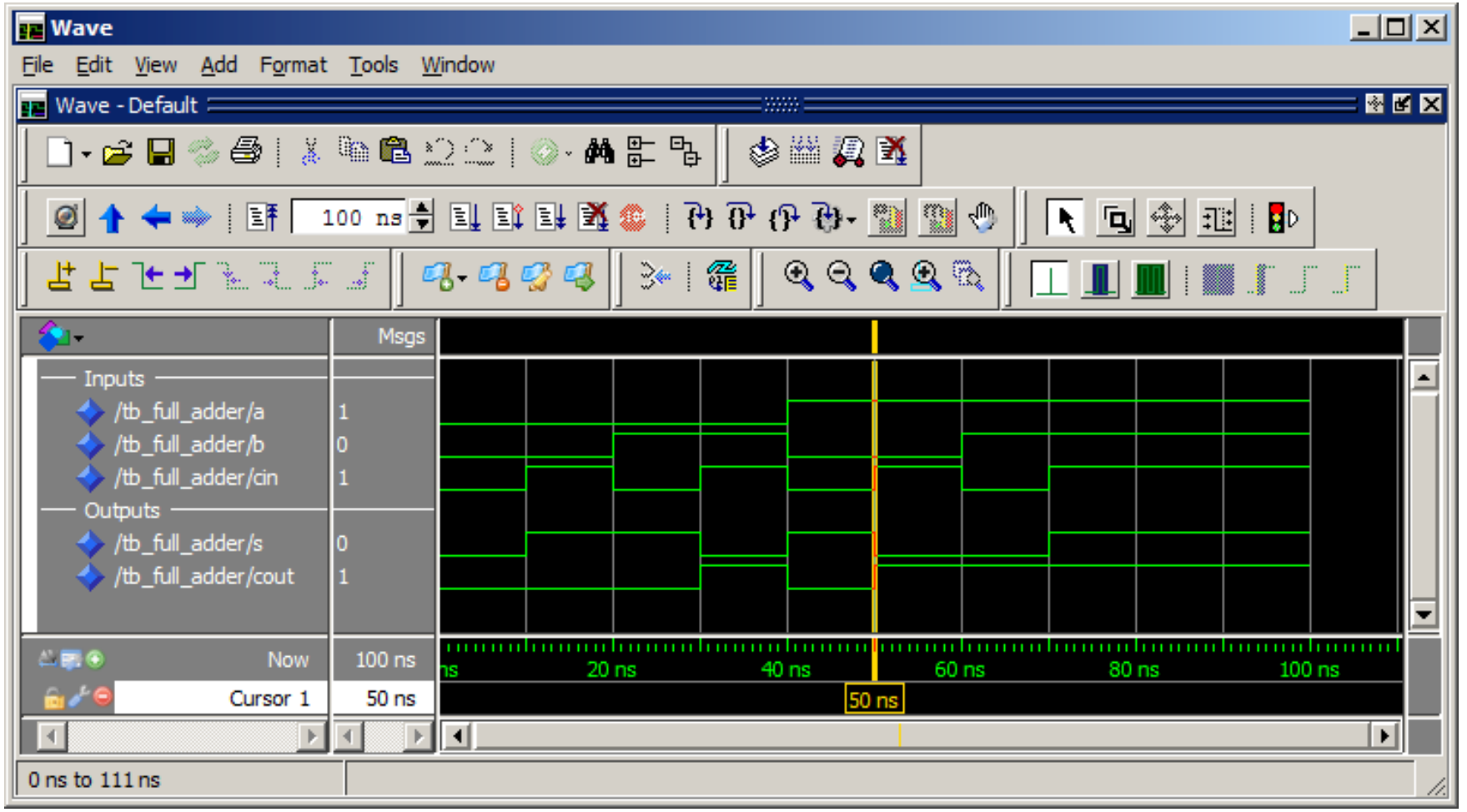
```
# Add a number of signals of the simulated design
# to the waveform list (GUI oriented)
add wave -divider "Inputs"
add wave a
add wave b
add wave cin
add wave -divider "Outputs"
add wave s
add wave cout

# Open List and Waveform window
view list
view wave

# Run simulation for 100 ns
run 100 ns
```

Simulatieresultaat full adder (1)

- Hiernaast is het resultaat van het uitvoeren van de testbench zichtbaar.
- Merk op dat in de lijst ook de delta delays te zien zijn.
- Op de volgende slide is het timingdiagram te zien.
- Hier is het mogelijk om *cursors* te gebruiken.

ns	delta	/tb_full_adder/a	/tb_full_adder/b	/tb_full_adder/cin	/tb_full_adder/s	/tb_full_adder/cout
0	+0	U	U	U	U	U
0	+1	0	0	0	U	U
0	+2	0	0	0	0	0
10	+1	0	0	1	0	0
10	+2	0	0	1	1	0
20	+1	0	1	0	1	0
30	+1	0	1	1	1	0
30	+2	0	1	1	0	1
40	+1	1	0	0	0	1
40	+2	1	0	0	1	0
50	+1	1	0	1	1	0
50	+2	1	0	1	0	1
60	+1	1	1	0	0	1
70	+1	1	1	1	0	1
70	+2	1	1	1	1	1

Simulatieresultaat full adder (2)



DE HAAGSE HOGESCHOOL

Optellen/aftrekken in VHDL

- VHDL kent de library `numeric_std` om optellers/aftrekkers te beschrijven.
- Het definieert twee nieuwe typen: `signed` (two's complement) en `unsigned`.
- In feite zijn dit array's van `std_logic` (net als `std_logic_vector`).
- Op deze nieuwe typen zijn diverse operatoren gedefinieerd zoals optellen, aftrekken, vermenigvuldigen en delen.
- Er zijn ook conversieroutines, bijvoorbeeld tussen `signed`/`unsigned` en `integer`.
- Door de strikte hantering van typen zijn `signed`/`unsigned` niet zonder meer te bewerken als `std_logic_vector`.

Optellen/aftrekken in VHDL

- Optellen/aftrekken van vectoren is eenvoudig.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
signal sum, a, b : unsigned(3 downto 0);
signal dif, c, d : signed(3 downto 0);
...
sum  <= a + b;
diff <= c - d;
```

- Bij deze oplossing is er geen overflowdetectie.

Unsigned overflow in VHDL

- Voor het detecteren van een overflow bij unsigned getallen is eigenlijk een extra sombit nodig. Dit hoogste bit geeft aan of er unsigned overflow is.

```
signal sum, a, b : unsigned(3 downto 0);
signal tmp : unsigned(4 downto 0);  -- one bit more
signal cout : std_logic;
...
tmp  <= ('0' & a) + ('0' & b);
sum  <= tmp(3 downto 0);
cout <= tmp(4);
```

- De concatenatie levert 5-bits vectoren.

Signed overflow in VHDL

- Het detecteren van een overflow bij signed getallen kan direct uit de twee op te tellen getallen en het antwoord.

```
signal sum, a, b : signed(3 downto 0);
signal over : std_logic;
...
sum  <= a + b;
over <= (not a(3) and not b(3) and sum(3)) or
        (a(3) and b(3) and not sum(3));
-- het kan ook zo
over <= '1' when (a(3) = b(3)) and (a(3) /= sum(3)) else '0';
```

- Voor aftrekken moet de functie worden aangepast.

Carry-in

- Een carry-in kan direct met een extra + aan een optelling worden toegevoegd. De carry-in is van het type `std_logic` en er is geen optelfunctie van een (un)signed en een `std_logic`.

```
sum <= a + b + ("0" + cin);
sum <= a + b + ("" + cin);      -- werkt ook
```

- Of er kan als volgt een extra bit gebruikt worden:

```
tmp <= (a & '1') + (b & cin);  -- tmp is 5 bits vector
sum <= tmp(4 downto 1);
```

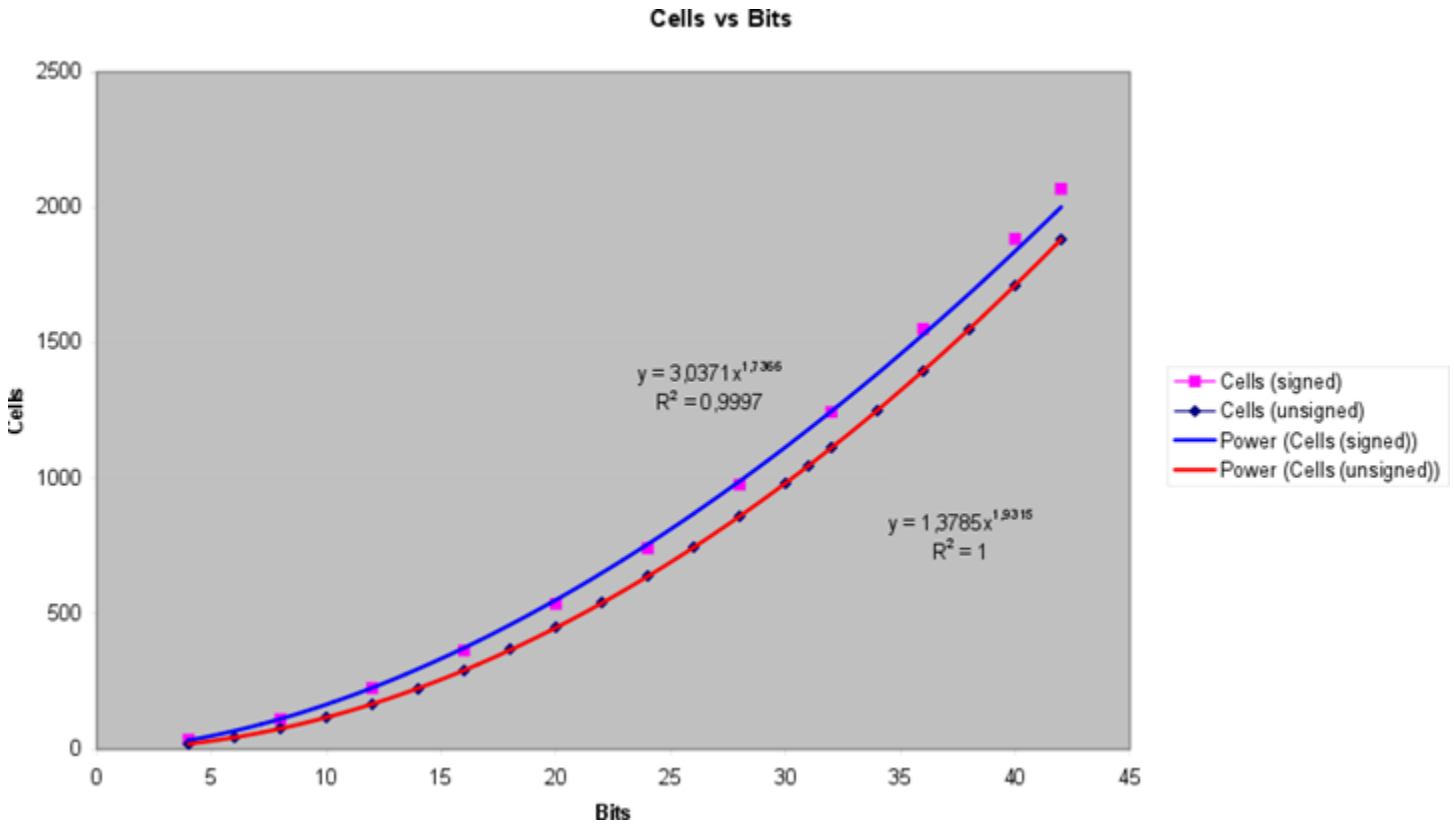
Vermenigvuldigen, delen

- Vermenigvuldigen en delen kan ook. Let erop dat combinatorische schakelingen hiervoor veel poorten kost. Onderstaande beschrijving van een signed deler kost 1245 cellen.

```
entity divider_comb is
   generic (n: positive := 32);
   port (a : in signed(n-1 downto 0);
         b : in signed(n-1 downto 0);
         q : out signed(n-1 downto 0));
end entity divider_comb;

architecture comb of divider_comb is
begin
   q <= a/b;
end architecture comb;
```

Delen in VHDL

- Grootte van de schakelingen loopt ongeveer kwadratisch op



numeric_std

- De package `numeric_std` bevat nog veel meer routines.

- Conversie, resizing van signed en unsigned, relationele bewerkingen, schuifoperaties.

```
sv1 <= std_logic_vector(u1);  -- unsigned to std_logic_vector
sv2 <= std_logic_vector(s1);  -- signed to std_logic_vector
u2  <= unsigned(sv3);         -- std_logic_vector to unsigned
s2  <= signed(sv4);           -- std_logic_vector to signed
n1  <= to_integer(u3);        -- unsigned to natural
i1  <= to_integer(s3);        -- signed to integer
u4  <= to_unsigned(n2, u4'length); -- natural to unsigned
s4  <= to_signed(i2, s4'length);   -- integer to signed
```

Referenties

- http://www.mrc.uidaho.edu/mrc/people/jff/vhdl_info/Synthesis_Art_2P.pdf
- http://mini.li.ttu.ee/~lrv/IAY0050/vhdl-synt.pdf
- http://vhdlguru.blogspot.nl/2010/03/how-to-write-testbench.html
- http://www.seas.upenn.edu/~ese171/vhdl/VHDLTestbench.pdf

- Fundamentals of Digital Logic with VHDL Design – Brown, 3rd Ed, 2008, ISBN 9780071268806
- Digital Design: Principles and Practices – John F. Wakerly, 4th Ed, 2006, ISBN 0-13-173349-4
- The Designer's Guide To VHDL – Peter J. Ashenden, 2nd Ed, 2002, ISBN 1-55860-674-2
- Digital System Design With VHDL – Mark Zwoliński, 2nd Ed, 2004, ISBN 0-13-039985-X

DE HAAGSE HOGESCHOOL

Academie voor Technology, Innovation & Society Delft

Academie voor ICT & Media

De Haagse Hogeschool, Delft
+31-15-2606311
J.E.J.opdenBrouw@hhs.nl
www.dehaagsehogeschool.nl

DE HAAGSE HOGESCHOOL