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

Digitale Systeem Engineering 1

Week 2 – Delay, Sequential VHDL, hiërarchie, generics

Jesse op den Brouw

DIGSE1/2018-2019

DE HAAGSE HOGESCHOOL

VHDL delay models

- Het beschrijven van vertragingen en minimale pulsbreedte wordt gedaan door middel van de *transport delay model* en de *reject inertial delay model*.
- De *transport delay model* wordt gebruikt als een signaal alleen vertraagd moet worden zoals bij bedrading op een chip (*wire delay*).
- De *reject inertial delay model* wordt gebruikt om vertragingen én minimale pulsbreedte van logica (poorten etc) te modelleren.
- VHDL gebruikt ook nog delta delays. Wordt later uitgelegd.
- Informatie over (de werking van) het vertragingsmodel kan gevonden worden in de VHDL Language Manual 2008, par. 10.5.

VHDL *transport delay model*

- *Wire delays* worden gemodelleerd met de keywords *transport* en *after*:

```
-- exact delay of 10 ns for sig
dly <= transport sig after 10 ns;
```

- Voor *simulatie* kan ook een signaal gegenereerd worden:

```
x <= transport '1' after  2 ns, '0' after  7 ns,  -- pulse 5 ns
               '1' after 11 ns, '0' after 15 ns,  -- pulse 4 ns
               '1' after 18 ns, '0' after 21 ns;  -- pulse 3 ns
```

DE HAAGSE HOGESCHOOL

VHDL *reject inertial delay model*

- Vertragingen en minimale pulsbreedte voor logica worden gemodelleerd met de keywords *reject*, *inertial* en *after*.

- Inertial is optioneel:

```
f1  <=  inertial x  after  5 ns;   -- delay 5 ns
f1  <=           x  after  5 ns;   -- same
```

- Nadeel hiervan is dat (uitgangs-)pulsen korter dan 5 ns niet worden doorgegeven (minimale pulsbreedte is gelijk aan vertraging)!

VHDL *reject inertial delay model*

- Om minimale pulsbreedte te modelleren moeten de keywords reject en inertial gebruikt worden.

```
f3  <=  reject 1 ns inertial x  after  5 ns;
```

- De vertraging van signaal f3 is 5 ns en de minimale pulsbreedte van signaal x moet *langer* dan 1 ns zijn.

- Als de minimale pulsbreedte even lang is als de vertragingstijd, dan wordt de puls doorgegeven.

```
f2  <=  reject 5 ns inertial x  after  5 ns;
```

DE HAAGSE HOGESCHOOL

VHDL delay models

- De volgende statements zijn identiek:

```
f4 <= reject 0 ns inertial a after 10 ns;
f4 <= transport a after 10 ns;
```

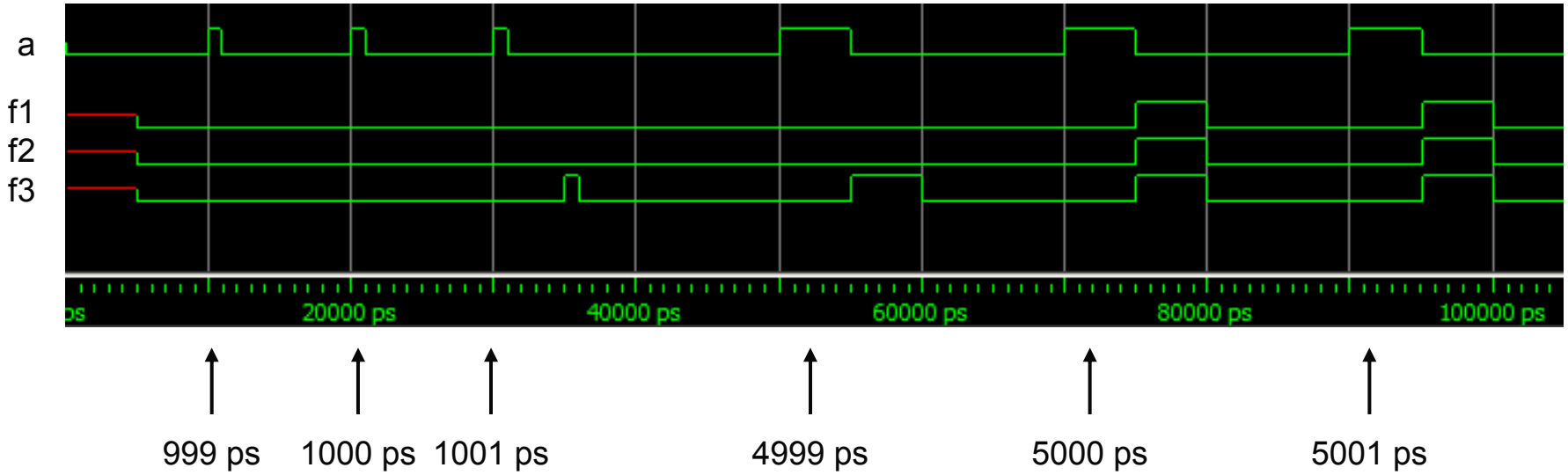
- Let op dat de minimale pulsbreedte over de *uitgang* gaat (het betreft een toekenning). Onderstaande toekenning bevat een OR-poort.

```
f5 <= reject 2 ns inertial a or b after 10 ns;
```

- Als de uitgang van de poort (f5) een puls produceert korter dan 2 ns, is deze niet te zien in de simulatie. Zie ook de opgaven.

VHDL *reject inertial delay model*

- Hieronder een simulatie van diverse pulsbreedtes.



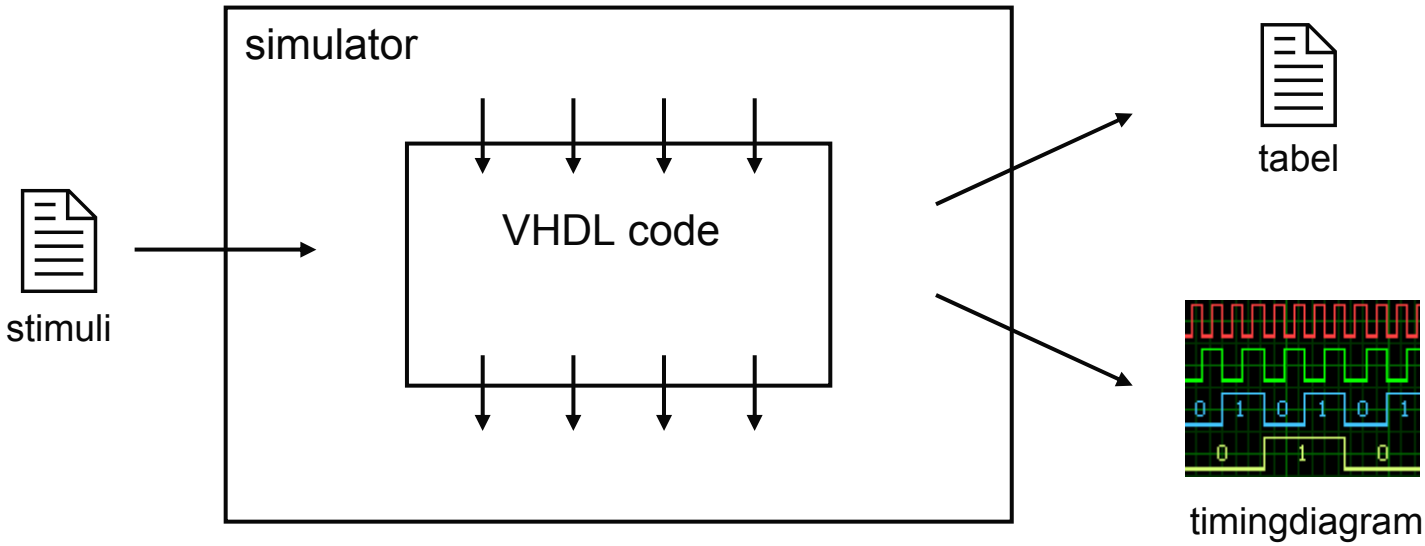

```
f1 <=                  inertial a after 5000 ps;
f2 <= reject 5000 ps inertial a after 5000 ps;
f3 <= reject 1000 ps inertial a after 5000 ps;
```

Simulatie

- VHDL is oorspronkelijk bedoeld voor simulatie van digitale systemen.
- Bij simulatie wordt de VHDL-code stap voor stap doorlopen (geinterpreteerd), alsof het een “gewone” programmeertaal is.
  - Hoe zit dat dan bij concurrent assignment → wordt verderop behandeld.
- Op de ingangssignalen (via entity-beschrijving) worden stimuli aangeboden.
- De stimuli worden aangeboden door middel van een *testbench*.
  - VHDL-code die signalen genereert.
  - Interactieve commando’s in de simulator, scripting.

Simulatie

- Het simulatie-systeem (simulator) berekent de uitgangssignalen bij gegeven ingangssignalen en geeft dit weer in een tabel of grafiek (tijddiagram).



Events

- Simulatoren zijn *event-driven*. Code wordt alleen maar uitgevoerd als daar noodzaak voor is. Deze noodzaak wordt veroorzaakt als de waarde van een signal verandert (*event*).

```
architecture logic of half_adder is
begin
    sum   <= a xor b after 5 ns;
    carry <= a and b after 5 ns;
end logic;
```

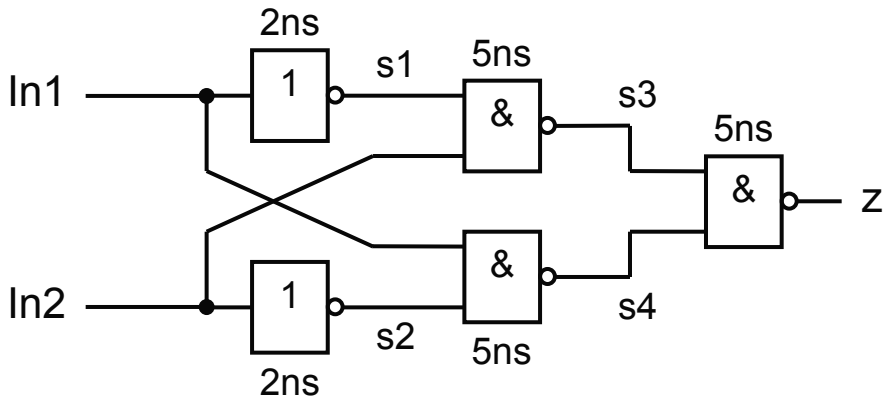
- Als a en/of b verandert (event), worden beide statements geëvalueerd.

- Te zien is dat de statements zelf ook weer events genereren en wel in de toekomst m.b.v. *after*.

Events

- De signal assignment statements genereren eventueel weer opnieuw events in de (simulatie-)toekomst.
- Een event is een nieuwe waarde van een signal samen met een tijd waarop de signal verandert.
- De simulator moet dus een *geordende lijst van events* bijhouden van wanneer nieuwe assignments moeten worden uitgevoerd.
- Concurrency wordt gesimuleerd door de eerstvolgende gelijktijdige events uit te voeren.

Voorbeeld simulatie



beginsituatie:

in1 = ‘0’, in2 = ‘1’

s1 = ‘1’, s2 = ‘0’,

s3 = ‘0’, s4 = ‘1’, z = ‘1’

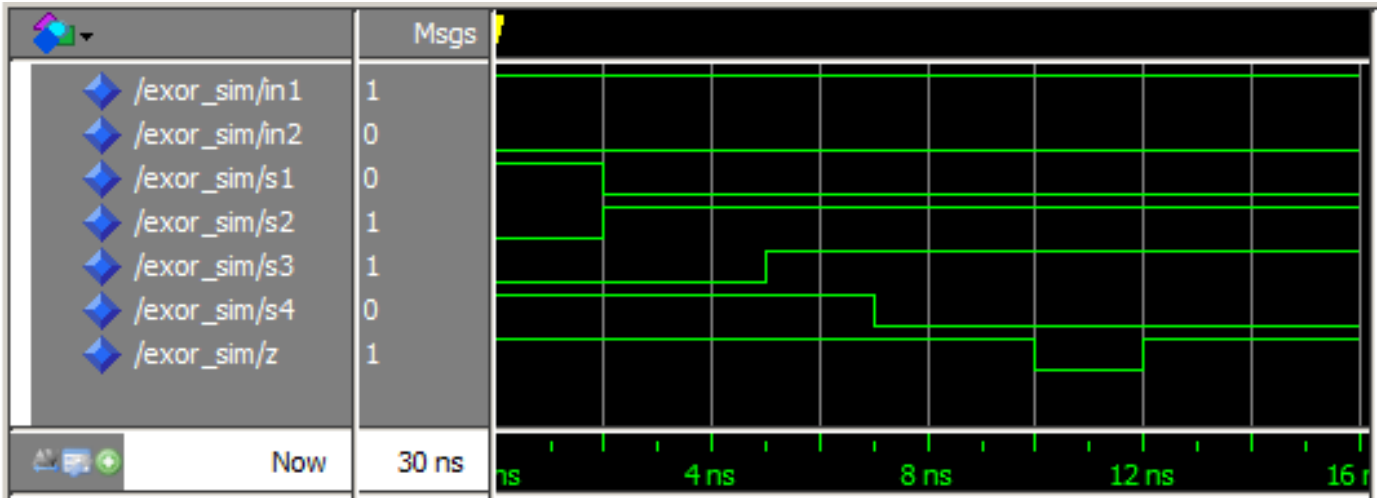
tijdstip 0 ns	tijdstip 2 ns	tijdstip 5 ns	tijdstip 7 ns	tijdstip 10 ns	tijdstip 12 ns
In1 → 1	s1 → 0	s3 → 1	s3 → 1	z → 0	z → 1
In2 → 0	s2 → 1	s4 → 1	s4 → 0	z (1, 12ns)	↓
↓	s3 (1, 5ns)	s3 (1, 7ns)	z (0, 10 ns)	↓	stop
s1 (0, 2ns)	s4 (1, 5ns)	s4 (0, 7ns)	↓	-	
s2 (1, 2ns)	↓	↓	z (1, 12ns)		
s3 (1, 5ns)	s3 (1, 7ns)	z (0, 10ns)			
s4 (1, 5ns)	s4 (0, 7ns)				

DE HAAGSE HOGESCHOOL

Voorbeeld simulatie

```
architecture sim_delay of exor_sim is
signal in1 : std_logic := '1';
signal in2 : std_logic := '0';
signal s1  : std_logic := '1';
signal s2  : std_logic := '0';
signal s3  : std_logic := '0';
signal s4  : std_logic := '1';
signal z   : std_logic := '1';
begin
   s1 <= transport not in1 after 2 ns;
   s2 <= transport not in2 after 2 ns;
   s3 <= transport s1 nand in2 after 5 ns;
   s4 <= transport s2 nand in1 after 5 ns;
   z  <= transport s3 nand s4 after 5 ns;
end sim_delay;
```

ns / delta	/exor_sim/in1	/exor_sim/in2	/exor_sim/s1	/exor_sim/s2	/exor_sim/s3	/exor_sim/s4	/exor_sim/z
0 +0	1	0	1	0	0	1	1
2 +0	1	0	0	1	0	1	1
5 +0	1	0	0	1	1	1	1
7 +0	1	0	0	1	1	0	1
10 +0	1	0	0	1	1	0	0
12 +0	1	0	0	1	1	0	1



Delta delay

- Het is mogelijk om een toekenning zonder vertraging te beschrijven, dus een event te genereren op dezelfde (simulatie-)tijd als waar de simulator events aan het evalueren is.
- Het probleem is dat bij het simuleren alleen events is de toekomst kunnen worden gegenereerd.
- De simulator lost dat op door *delta delay* (Δ).
- Delta delay wordt gebruikt om herhaald doorrekenen van events te ordenen. Het is een oneindig kleine tijdstap.
- Er bestaat geen wereldlijke equivalent van delta delay, alleen bij simulatie!

Delta delay

- Als voorbeeld een buffer opgebouwd uit twee inverters.

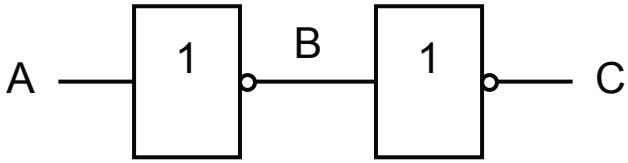
```
architecture double_inv of buf is
begin
    C <= not B;     -- let op: geen delay!
    B <= not A;     -- let op: geen delay!
end double_inv;
```

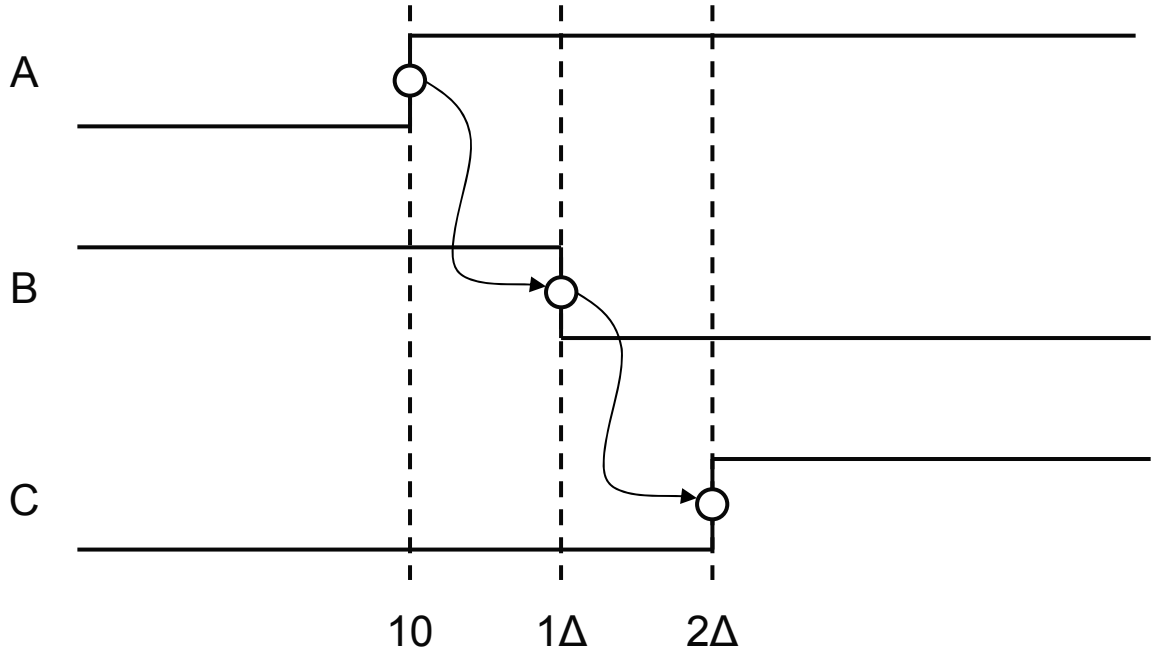
- Als A van waarde verandert wordt alleen de tweede assignment uitgevoerd.

- Het is direct te zien dat als A verandert, B ook verandert en dat de eerste assignment dus ook moet worden uitgevoerd.

Delta delay

- Voorbeeld: na 10 ns wordt A = ‘1’



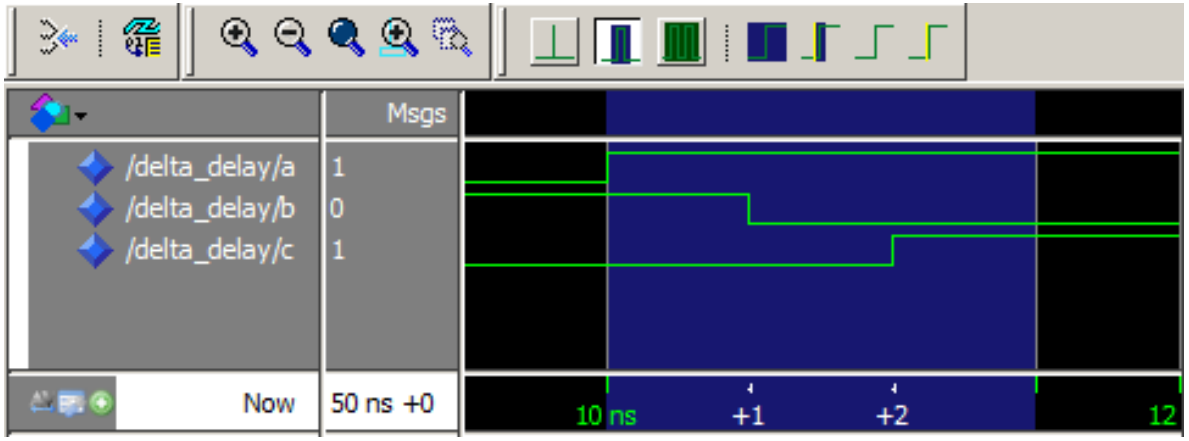


Delta delay

- In Modelsim kunnen de events zichtbaar gemaakt worden.

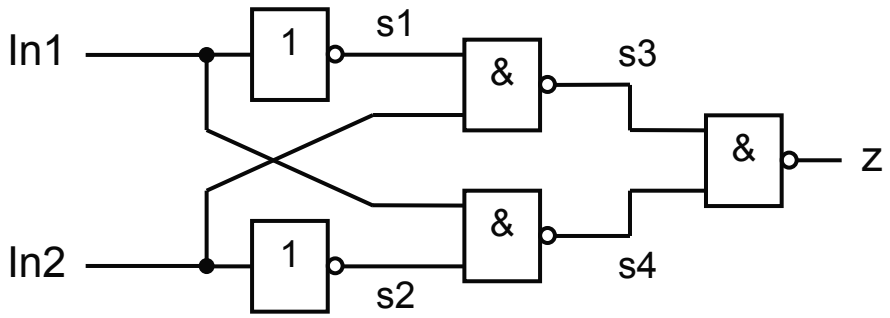
```
entity delta_delay is
end entity;

architecture sim of delta_delay is
signal a, b, c : bit := '0';
begin
   c <= not b;
   b <= not a;
   a <= '1' after 10 ns;
end sim;
```

ns delta	/delta_delay/a	/delta_delay/b	/delta_delay/c
0 +0	0	0	0
0 +1	0	1	1
0 +2	0	1	0
10 +0	1	1	0
10 +1	1	0	0
10 +2	1	0	1



Deze code is alleen simuleerbaar.

Voorbeeld simulatie



beginsituatie:

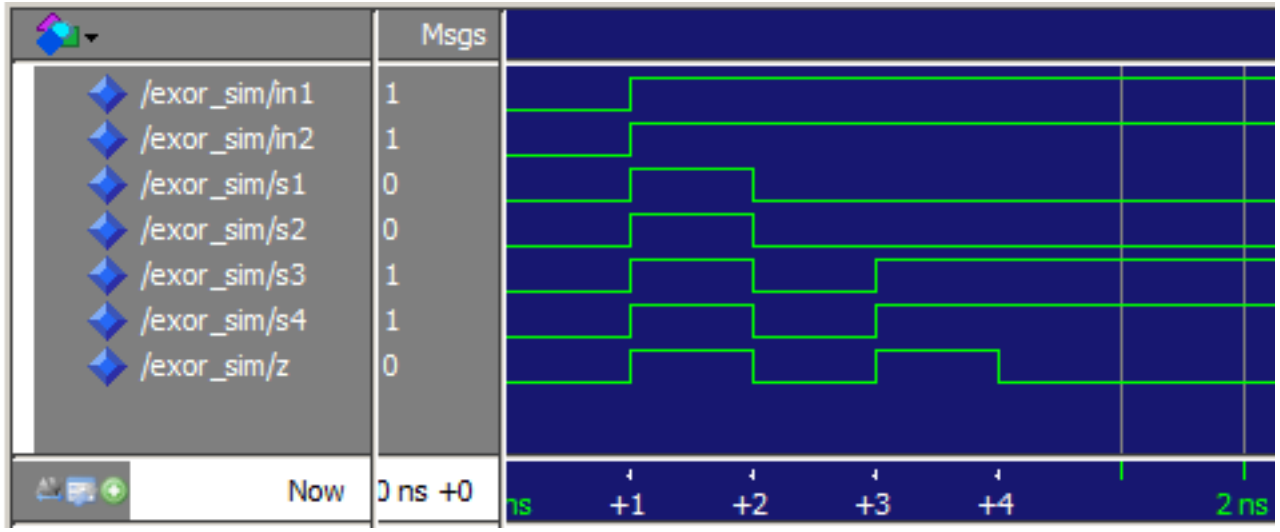
Alle signalen op ’0’

tijdstip 0 ns	tijdstip 0 ns + Δ	tijdstip 0 ns + 2Δ	tijdstip 0 ns + 3Δ	tijdstip 0 ns + 4Δ
In1,In2 <= 1	In1,In2,s1 → 1	s1,s2,s3 → 0	s3,s4,z → 1	z → 0
↓	s2,s3,s4,z → 1	s4,z → 0	↓	↓
In1 (1,Δ)	↓	↓	z (0,4Δ)	stop
In2 (1,Δ)	s1 (0,2Δ)	s3 (1,3Δ)		
s1 (1,Δ)	s2 (0,2Δ)	s4 (1,3Δ)		
s2 (1,Δ)	s3 (0,2Δ)	z (1,3Δ)		
s3 (1,Δ)	s4 (0,2Δ)			
s4 (1,Δ)	z (0,2Δ)			
z (1,Δ)				

Voorbeeld simulatie

```
architecture sim_delta of exor_sim is
signal in1 : std_logic := '0';
signal in2 : std_logic := '0';
signal s1  : std_logic := '0';
signal s2  : std_logic := '0';
signal s3  : std_logic := '0';
signal s4  : std_logic := '0';
signal z   : std_logic := '0';
begin
   s1 <= not in1;
   s2 <= not in2;
   s3 <= s1 nand in2;
   s4 <= s2 nand in1;
   z  <= s3 nand s4;

   in1 <= '1';
   in2 <= '1';
end sim_delta;
```

ns delta	/exor_sim/in1 /exor_sim/in2 /exor_sim/s1 /exor_sim/s2 /exor_sim/s3	/exor_sim/s4 /exor_sim/z
0 +0	0 0 0 0 0	0 0
0 +1	1 1 1 1 1	1 1
0 +2	1 1 0 0 0	0 0
0 +3	1 1 0 0 1	1 1
0 +4	1 1 0 0 1	1 0



Complex gedrag modelleren in VHDL

- Concurrent signal assignments statements worden gebruikt om digitale systemen te specificeren op gate-niveau.

- Hogere niveau digitale componenten hebben een complexer gedrag:
  - Input/output gedrag dat niet makkelijk kan worden vertaald naar concurrent signal assignments
  - Modellen die gebruik maken van toestanden
  - Gebruik van complexe datastructuren

- Dus: krachtigere constructies binnen VHDL nodig.

- Noot: we beschouwen eerst simulatie, later synthese.

Bron: Stephan Wong

Process

- Hardware is van nature concurrent, maar de schrijfwijze van broncode is sequentieel.
- VHDL kent de mogelijkheid om (het gedrag van) hardware sequentieel te beschrijven.
- Dit kan door de code te plaatsen binnen een *process*.
- Een process kan worden gezien als één complexe signal assignment statement.
- Processen communiceren met elkaar via *signals*.

Process

- Statements binnen een process worden sequentieel uitgevoerd.
- De signal assignment statements worden dus ***sequentiële signal assignment statements*** genoemd.
- De process body ziet uit als C of Pascal:
  - Declaratie en gebruik van variabelen (zie verderop)
  - *If - then*, *if - then - else*, *case*, *for*, and *while* constructs
  - Body kan ook signal assignments statements bevatten
- Een process wordt concurrent uitgevoerd met andere concurrent signal assignment statements.

Bron: Stephan Wong

Process

- Een voorbeeld:

```
begin                -- architecture
  process is         -- process statement
  begin              -- process body begins
    A <= '1';
    B <= '0';
    A <= '0';
    ...
  end process;       -- process body ends
end;                 -- architecture ends
```

- Wat is de waarde van A na het uitvoeren van het process?
- Noot: de uiteindelijke toekenningen gebeuren één delta later!

Process

- Nog een voorbeeld:

```
begin
  NAAM: process (A,B) is
  begin
    ...
  end process;
end;
```

- NAAM is een optionele label. De opsomming (A, B) is een *sensitivity list*. Het process is alleen gevoelig voor de signalen A en B, dus als er een event op A en/of B komt. Dit kan zijn bij simulatie en synthese.

Process

- Processen werken onderling concurrent.

```
begin
  NAAM: process (A,B) is
  begin
    ...
  end process;
end;
```

```
begin
  NAAM2: process (B,C) is
  begin
    ...
  end process;
end;
```

- Als signaal B verandert, worden beide processen éénmaal doorlopen.

- Het doorlopen van een process gebeurt in 0 simulatietijd en kan events in de toekomst genereren.

If

- Met `if` kan een beslissing beschreven worden. Gebruik van `else` en `elsif` is mogelijk.

```
IF_EX1: process (A, B) is
begin
  if A = '1' and B = '0' then
    P <= '0';
  elsif A = '0' and B = '1' then
    P <= '0';
  else  -- don't forget
    P <= '1';
  end if;
end process;
```

wat is de logische functie van P?

- De laatste else mag niet vergeten worden, anders worden geheugenelementen (latches) gesynthetiseerd.

If

- De volgende code is volgens het boekje:

```
IF_EX2: process (A,B) is
begin
  R <= '0';     -- default value
  S <= '1';     -- default value
  if A = '1' and B = '1' then
    R <= '1';
    S <= '0';
  end if;       -- no else!
  if A = '1' and B = '0' then
    R <= '1';
  end if;       -- no else!
end process;
```

A	B	R	S
0	0	0	1
0	1	0	1
1	0	1	1
1	1	1	0

- Vraag: wat zijn de functies voor R en S?

DE HAAGSE HOGESCHOOL

Case

- Case kan gebruikt worden voor het testen van één signal.

```
CASE_EX: process (SEL) is
begin
  case SEL is
    when "00" =>   F <= "000";
    when "01" =>   F <= "110";
    when "10" =>   F <= "110";
    when "11" =>   F <= "011";
    when others => F <= "000";  -- catch-all
  end case;
end process;
```

- Deze wijze sluit goed aan bij het beschrijven van waarheidstabellen.
- `when others` is een *catch-all* indien eerdere condities niet waar zijn.

Variable

- Binnen een process is het mogelijk om lusconstructies te gebruiken.
- Zo is het mogelijk om for-loops en while-loops te gebruiken.
- Bij deze programmeerconstructie wordt gebruik gemaakt van een loop-variabele.
- Dat kan geen signal zijn, want de toekenning aan signals gebeurt één delta later dan de uitvoering ervan (indien geen delay is gegeven).
- Er is een nieuw data-object nodig: *variable*.

Variable

- Variables hebben alleen waarden en bestaan alleen binnen een process. Toekenning van een waarde gebeurt gelijk op het moment van het uitvoeren van de toekenningsopdracht.

- Variables kunnen dezelfde datatypes aannemen als signals. Vectoren zijn ook mogelijk.

```
variable test : boolean;  -- declaratie
...
test := true;             -- assignment met :=
...
if test = true then       -- gebruik in expressie
...
```

For

- Met een for kan een exact aantal keer een lus worden doorlopen.
- For wordt meestal gebruikt om langs een array te lopen.
- Er kan oplopend (`to`) of aflopend (`downto`) geïtereerd worden
- De lusvariabele is tijdens het uitvoeren van de *body* van de lus een constante en kan dus niet aangepast worden! Daarnaast hoeft de lusvariabele niet gedeclareerd te worden!
- Op de volgende slide staat de beschrijving van een 8-input NOR. Deze is eenvoudig te wijzigen in een ander aantal.

For

```
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;
```

- De variabelen i en p krijgen direct nieuwe waarden bij de toekenning.

DE HAAGSE HOGESCHOOL

While

- De while-lus wordt gebruikt voor *iteratie*.
- Op de volgende slide staat een algoritme beschreven voor het uitrekenen van de cosinus van een hoek.
- Merk op dat de while-lus een variabel aantal keer wordt doorlopen en dus geen vast aantal keer (denk hierbij aan synthese).
- Het berekenen van de cosinus gebeurt in 0 simulatietijd.

While

```
entity cos is
  port (theta : in real; result : out real );
end entity cos;

architecture series of cos is
begin
  summation: process (theta) is
  variable sum, term : real;
  variable n : natural;
  begin
    sum := 1.0; term := 1.0; n := 0;
    while abs (term) > abs (sum / 1.0E6) loop
      n := n + 2;
      term := (-term) * theta**2 / real(((n-1) * n));
      sum := sum + term;
    end loop;
    result <= sum;
  end process summation;
end architecture series;
```

$$\cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \ldots$$

DE HAAGSE HOGESCHOOL

VHDL beschrijving geheugenelementen

- Met een process is het mogelijk om latches en flipflops te beschrijven.
- Voor het beschrijven van een gated latch is het genoeg om alleen de voorwaarde voor wanneer de latch transparant is te beschrijven.
- Voor een flipflop is het lastiger, immers een flipflop klokt de data in op een flank.
- Voor een flipflop is dus een *flankbeschrijving* nodig.

Latch

- Met een if kan een latch beschreven worden.

```
LATCH: process (EN, D) is
begin
  if EN = '1' then
    Z <= D;
  end if;
end process;
```

- Merk op dat er geen else is. Merk op dat zowel EN als D in de sensitivity list staan.

Opmerking: als alleen EN in de sensitivity list zou staan, levert deze beschrijving een D-flipflop op. Het proces wordt dan immers alleen gestart als er een event is op EN (EN'event).

Flipflop

- Met een if kan een flipflop beschreven worden. Hieronder de beschrijving een positive edge triggered D-flipflop met behulp van een *signal attribute*.

```
DFF: process (CLK) is  -- alleen gevoelig voor CLK!
begin
  if CLK'event and CLK = '1' then
    Q <= D;
  end if;
end process;
```

- De toekenningsopdracht `Q <= D` wordt uitgevoerd als er een event is op signal CLK én CLK = 1 (dat is dus nadat het event heeft plaatsgevonden).

Edge

- De programmeerconstructies `clk'event and clk = '1'` en `clk'event and clk = '0'` komen zo vaak voor dat er twee functies voor zijn gemaakt.

  `rising_edge(clk)`

  levert true als er een opgaande flank is geconstateerd.

  `falling_edge(clk)`

  levert true als er een neergaande flank is geconstateerd.

- Deze werken ook beter in een simulatieomgeving waar een signal andere waarden dan '0' en '1' kan hebben.

Flipflop

- Voorbeeld:

```
SHFT: process (CLK) is
begin
  if rising_edge(CLK) then
    A        <= DATA_IN;
    B        <= A;
    C        <= B;
    DATA_OUT <= C;
  end if;
end process;
```

- Dit is een schuifregister.

Hoe gaat de toekenning?

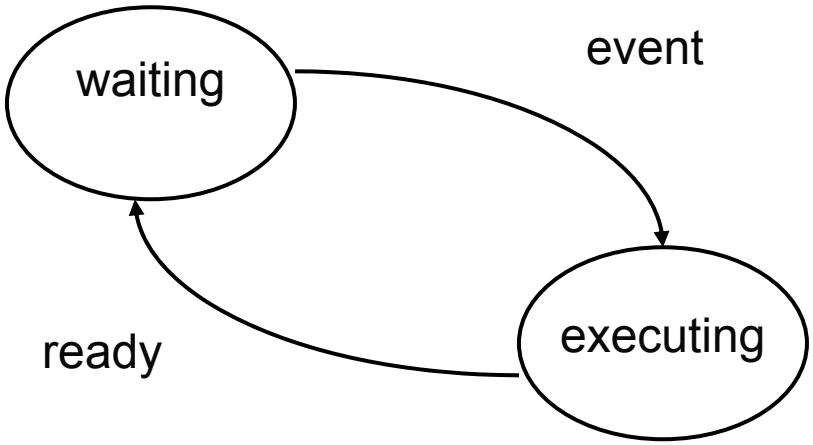
Op de opgaande flank wordt:

- de huidige waarde van DATA_IN toegekend aan A,
- de huidige waarde van A toegekend aan B,
- de huidige waarde van B toegekend aan C,
- de huidige waarde van C toegekend aan DATA_OUT.

Merk op: toekenning gebeurt één delta later!

Process

- Een process kent twee toestanden tijdens simulatie.

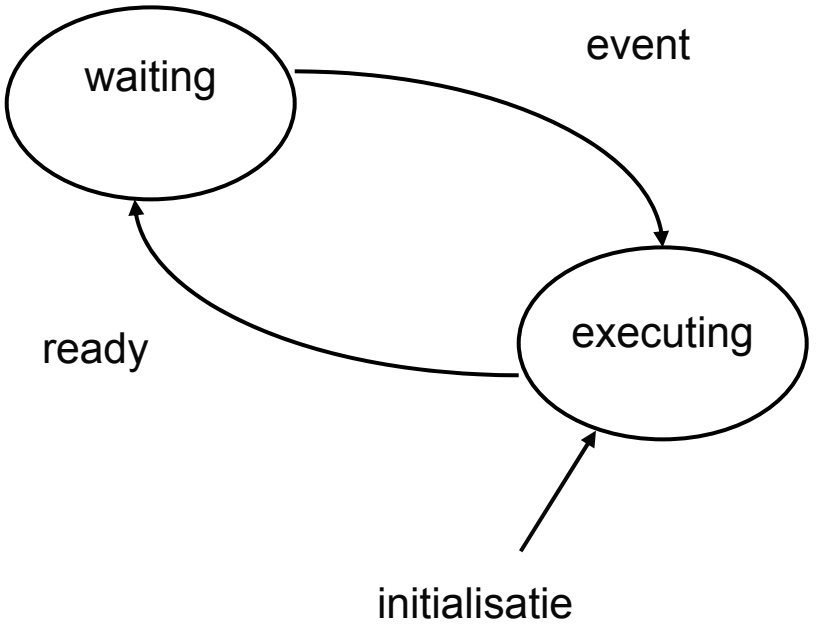


- Een process staat eerst in de waiting toestand.

- Als er een event binnenkomt (verandering op een ingangssignaal), wordt het process gestart. Het komt in de executing toestand.

Process

- Nadat alle statements binnen het process zijn uitgevoerd, en de waarden van de uitgangen bekend zijn, gaat het weer in de waiting toestand.



- Deze veranderingen op de uitgangen kunnen tot gevolg hebben dat weer nieuwe events optreden.

- Noot: bij initialisatie van de simulatie-omgeving wordt elk process automatisch één keer gestart.

Wait

- Een process kan in de waiting-toestand gebracht worden door een wait-statement.

- Er zijn vier vormen:

`wait until`	wacht tot een signal een bepaalde waarde heeft
`wait on`	wacht op een event van één of meerdere signals
`wait for`	wacht een bepaalde (simulatie-)tijd
`wait`	wacht voor altijd

- Indien minstens één wait-statement wordt gebruikt, mag er geen sensitivity list worden opgegeven!

Wait until

- Met `wait until` wordt een process in de waiting-toestand gebracht tot een signal een bepaalde waarde heeft gekregen.

- Dan moet er dus ook een event zijn geweest!

```
dff_alt: process is  -- geen sensitivity list
begin
  wait until clk = '1';
  q <= d;
end process;
```

- Heel erg fout: `wait until clk'event and clk = '1';`

Wait on

- `wait` on wordt gebruikt om op events van signals te wachten.

```
begin
  proc1: process (A,B) is
  begin
  ...

  end process;
end;
```

```
begin
  proc2: process is
  begin
  ...
    wait on A,B;
  end process;
end;
```

- Beide processen worden identiek verwerkt, dus een process met een sensitivity list is identiek aan een process zonder sensitivity list maar met een `wait` on statement als laatste statement.

Wait for

- Met `wait for` kan een process voor een bepaalde (simulatie-)tijd in de waiting-toestand gezet worden. Dit wordt vooral gebruikt bij high-level beschrijvingen en testbenches.

```
testbench: process is  -- geen sensitivity list
begin
  data <= '0';         -- data is 0
  reset <= '1';        -- activate reset
  wait for 100 ns;     -- wait a little bit
  data <= '1';         -- data is 1
  reset <= '0';        -- deactivate reset
  wait for 200 ns;     -- wait a bit more
  ...
  wait;
end process;
```

Wait

- Met alleen het keyword `wait` wordt een process voor eeuwig in de waiting-toestand geplaatst. Dit wordt vaak gebruikt aan het eind van een testbench.
- De verschillende wait-constucties kunnen door elkaar heen gebruikt worden.
- De constructies `wait until` en `wait on` *kunnen* leiden tot synthese.
- `wait for` en `wait` kunnen niet gesynthetiseerd worden.
- Uitvoeren van een wait leidt tot updaten van signals (denk aan deltas)!

Opgaven

- Geef een VHDL-beschrijving voor een EXOR-poort d.m.v. sequentiële VHDL-statements. Geef zoveel mogelijke oplossingen.

- Geef een VHDL-beschrijving voor een negative edge triggered D-flipflop.

- Geef een VHDL-beschrijving voor een 8-bit schuifregister dat schuift op de opgaande flank.

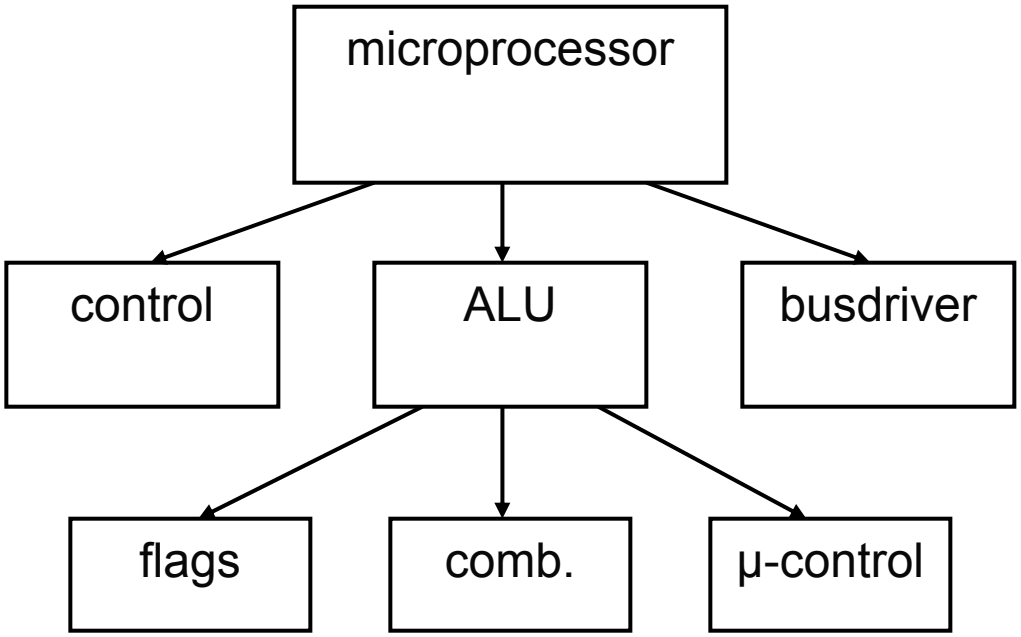
- Eerder is code van een 8-input NOR gegeven. Geef nog twee andere mogelijkheden voor het beschrijven van de NOR. Gebruik de eerder gegeven code als leidraad.

Hiërarchie

- Grote projecten worden opgedeeld in meerdere *design units*.
- Deze design units vormen samen het hele systeem.
- Indien zo'n design unit nog te groot is om in één keer ontwikkeld te worden, moet het weer onderverdeeld worden in een aantal design units.
- Er worden dus meerdere lagen in het ontwerp gemaakt: er wordt hiërarchisch beschreven.
- Deze manier van beschrijven wordt *structural VHDL* genoemd.

Hiërarchie

- Voorbeeld van een microprocessor:



-- top level

-- mediate level

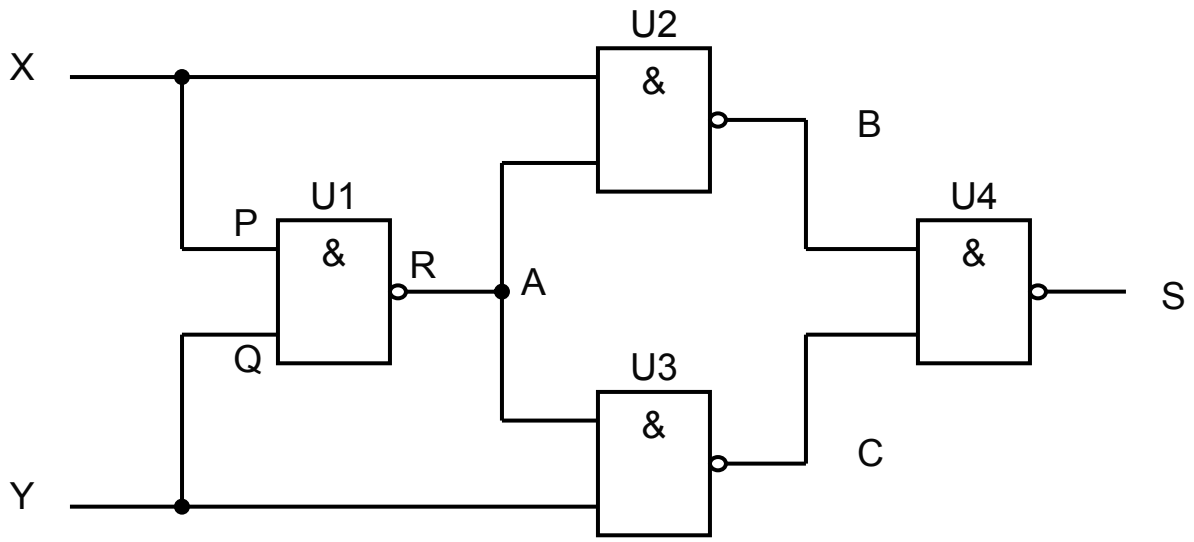
-- bottom level

Hiërarchie

- De werkwijze in VHDL is als volgt.
- Eerst wordt een design unit beschreven, zowel de entity als de architecture.
- Deze design unit wordt gesimuleerd en geverifieerd op functionaliteit en timing.
- Daarna wordt de design unit als *component* gebruikt in een hogere laag in de hiërarchie.
- In deze hogere laag wordt de component gedefinieerd en *geïnstantieerd*.

Hiërarchie

- Als voorbeeld nemen we een EXOR die opgebouwd is uit vier NANDs.



- Het geheel heeft de ingangen X en Y en uitgang S.
- De interne signalen zijn A, B en C.
- De NANDs hebben de ingangen P en Q en uitgang R.

Hiërarchie

- Eerst de NANDs:

```
entity Nand_2 is
  port (
    P: in std_logic;
    Q: in std_logic;
    R: out std_logic
  );
end Nand_2;
architecture logical of Nand_2 is
begin
  R <= not (P and Q); -- the nand
end logical;
```

Hiërarchie

```
entity xor4nand is          -- entity of our XOR
  port (
    X: in std_logic;
    Y: in std_logic;
    S: out std_logic
  );
end xor4nand;
architecture structural of xor4nand is
  component Nand_2 is       -- component declaration
    port (
      P: in std_logic;
      Q: in std_logic;
      R: out std_logic
    );
  end component;
  -- verder op volgende slide
```

DE HAAGSE HOGESCHOOL

Hiërarchie

- Hieronder volgt de structurele beschrijven d.m.v. component-instantiëring

```
-- vervolg van vorige slide
signal A,B,C : std_logic;
begin
  U1: Nand_2     -- component instantiation
    port map (P => X, Q => Y, R => A);
  U2: Nand_2     -- component instantiation
    port map (P => X, Q => A, R => B);
  U3: Nand_2     -- component instantiation
    port map (P => A, Q => Y, R => C);
  U4: Nand_2     -- component instantiation
    port map (P => B, Q => C, R => S);
end structural;
```

Uitleg port map:

Bekijk U2. Ingang X van xor4nand wordt gekoppeld aan ingang P van Nand_2. Intern signal A wordt gekoppeld aan ingang Q van Nand_2 en uitgang R wordt gekoppeld aan intern signal B.

Generics

- Het is mogelijk om een VHDL-component *parametriseerbaar* maken.
- Bijvoorbeeld: de bitbreedte van een opteller, vertragingstijd van een poort.
- Dit kan elegant worden opgelost d.m.v. een *generic constant*.
- Bij gebruik van de component in een hogere hierachie (structural VHDL) kan dan de werkelijke waarde worden opgegeven.
- Bij gebrek aan een parameter wordt de *default* waarde gebruikt

Generics

- Voorbeeld entity-beschrijving multi-NOR:

```
entity multinor is
  generic (tpd : time := 2 ns;
           n : natural := 8);
  port (
    x : in std_logic_vector (n-1 downto 0);
    q : out std_logic
  );
end multinor;
```

- De constante tpd is van het type `time`. De defaultwaarde is 2 ns. De constante n is van het type `natural` (integer van 0 tot ...). De default waarde is 8.

Generics

```
entity big_multinor is  -- use in higher level
  port (x : in std_logic_vector(11 downto 0);
        q : out std_logic);
end big_multinor;

architecture structural of big_multinor is

component multinor       -- declaration of component multinor
    generic (tpd : time := 2 ns; n : natural := 8);
    port (x : in std_logic_vector(n-1 downto 0);
          q : out std_logic);
end component;

begin
  instnor : component multinor   -- instantiation of multinor
    generic map (tpd => 8 ns, n => 12)
    port map (q => q, x => x);
end structural;
```

DE HAAGSE HOGESCHOOL

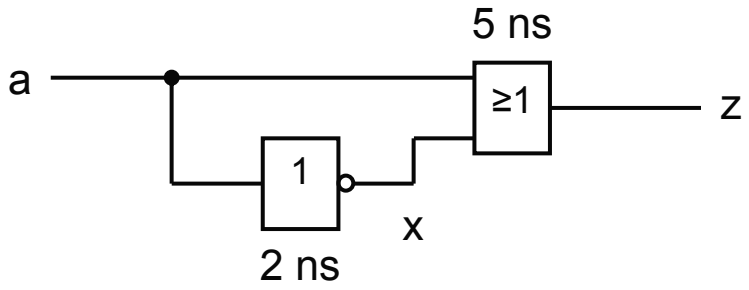
Generics

- Voorbeeld architecture-beschrijving multi-NOR:

```
architecture cascode of multinor is
begin
  process (x) is
  variable p : std_logic;
  begin
    p := '0';                       -- Initialize to '0'
    for i in n-1 downto 0 loop      -- Iterate the loop
      p := p or x(i);               -- OR-ing the lot
    end loop;
    q <= not p after tpd;           -- Make it a NOR
  end process;
end architecture cascode;
```

Opgaven

- Gegeven onderstaande schakeling. Alle signalen zijn van het type `bit` op ‘0’ geinitialiseerd. Op tijdstip t = 3 ns wordt a logisch ‘1’. Reken het schema door en stel de eventlijsten op voor elk event-tijdstip.



- Als boven, maar nu wordt signaal a na 10 ns weer ‘0’.

- Als boven, maar nu als geen vertragingen zijn opgegeven.

Literatuur

- IEEE-1076: IEEE Standard VHDL Language Reference Manual.
- Digitale Systemen, ontwerpen met VHDL, Wim Dolman, 1e druk, ISBN: 978-90-484-2020-9.
- Slides VHDL bij het vak EE1410 Digitale Systemen, S. Wong, TU Delft.

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