Digitale Systemen (ET1 410)

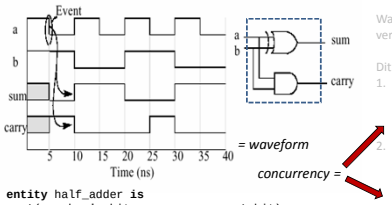
Arjan van Genderen
Stephan Wong

Faculteit EWI
Technische Universiteit Delft
Cursus 2010-2011

Samenvatting vorig college

- Wat is VHDL? Waarvoor gebruiken we het?
- Verschil sequentieel vs. concurrent
- Vereisten van een hardware modelleringstaal
- Basisconcepten VHDL:
  - Entity en architecture
  - Signalen (constanten, variabelen) en vectoren
  - *std_ulogic* en *std_ulogic_vectoren*
  - Simple, conditional, en selected signal assignment statements
  - Structurele beschrijvingen
  - Delay modellen (inertial, transport, en delta)

Een korte visuele herhaling



= waveform

concurrency =

Wat gebeurt er als signaal 'a' verandert?

Dit zal tot gevolg hebben:
1. de ingangen van de 'exor'-gate en de 'and'-gate veranderen **op dezelfde moment.**
2. als beide gates dezelfde vertraging hebben, dan zal ook de uitgangen **op dezelfde moment** veranderen.

```
entity half_adder is
port( a, b: in bit; sum, carry: out bit);
end half_Adder;

architecture my_half_adder_arch of Half_Adder is
begin
carry <= a and b after 5 ns;
sum   <= a exor b after 5 ns;
end my_half_adder_arch;
```

- Entity? Ports?
- Architecture?
- Signals?
- Values?
- Events?
- Propagation delays?
- Waveform?
- Behavioral or Structural?

Onderwerpen van vandaag

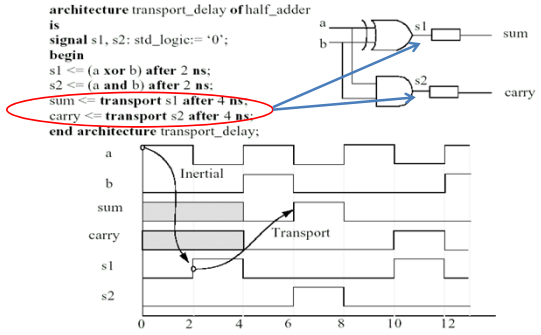
- Discrete Event Simulator
- Modelleren van complex gedrag → process
- Simpele "programmeer"-constructs
- Variabelen vs. signalen
- Wait statements
- State machines in VHDL (een voorbeeld)
- Hiërarchie van componenten
- Generics
- Configuraties

Delay Models

- Inertial delay
  - Standaard model
  - Geschikt voor modeleren van vertragingen van gates
- Transport delay
  - Modeleert (kleine) vertragingen door "draden" (wires)
  - Alle ingangen worden doorgegeven aan de uitgang
- Delta delay
  - Wat als er geen propagatie vertraging is?
  - Oneindig kleine vertraging die automatisch wordt toegevoegd door de simulator om correcte volgorde van events te waarborgen

Transport Delay Model

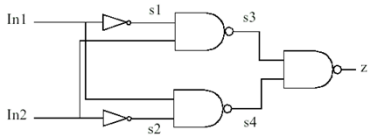
```
architecture transport_delay of half_adder
is
signal s1, s2: std_logic:= '0';
begin
s1 <= (a xor b) after 2 ns;
s2 <= (a and b) after 2 ns;
sum <= transport s1 after 4 ns;
carry <= transport s2 after 4 ns;
end architecture transport_delay;
```



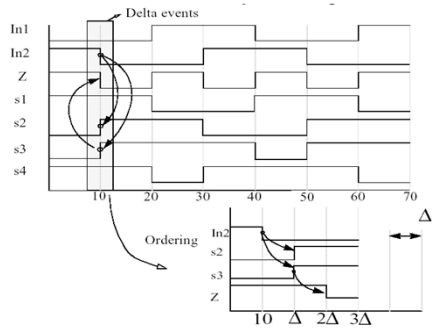
Delta Delay: een voorbeeld

```
library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
    z : out std_logic);
end          combinational;
```

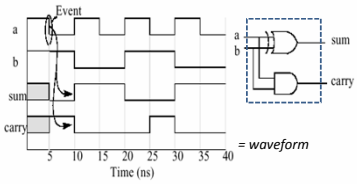
```
architecture behavior of combinational
signal s1, s2, s3, s4: std_logic:= '0';
begin
s1 <= not In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
end                  behavior;
```



Delta Delays: een voorbeeld



Discrete Event Simulator: Introductie




```
entity half_adder is
port( a, b: in bit; sum, carry: out bit);
end half_Adder;

architecture my_half_adder_arch of Half_Adder is
begin
carry <= a and b after 5 ns;
sum   <= a exor b after 5 ns;
end my_half_adder_arch;
```

* Wat zijn signalen?
Signalen zijn tijd-waarde paren!

* Wat zijn events?
Events zijn veranderingen in waarde van signalen op een bepaald tijdstip.

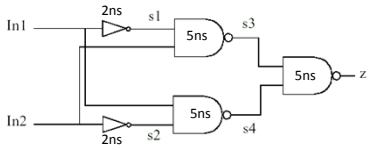
* Hoe worden nieuwe events gegenereerd?
Bij verandering (dus: event) in een van de signalen in de RHS van een signal assignment statement.

* Hoe ziet zo'n nieuwe event eruit?
"signaal X krijgt een nieuwe waarde Y op tijdstip Z"

* Wat doen we met alle events?
Opslaan in een event lijst.

* Hoe behandelt de simulator alle events?
Oftewel: hoe zit de concurrency verweven in de discrete event simulator?
Door de event lijst te ordenen en de eerstvolgende GELUKTIJDIGE events uit te voeren.

Discrete Event Simulator: Voorbeeld



Aanname:
Alle signalen zijn op '0' geïnitialiseerd.

Tijdstip 0ns:	Tijdstip 2ns:	Tijdstip 5ns:	Tijdstip 7ns:	Tijdstip 10ns:	Tijdstip 12ns:
- s1 (0, 2ns) - s2 (0, 2ns) - s4 (1, 5ns) - s3 (1, 5ns)	- s4 (1, 5ns) - s3 (1, 5ns) - s3 (1, 7ns) - s4 (1, 7ns)	- s3 (1, 7ns) - s4 (1, 7ns) - z (0, 10ns)	- z (0, 10ns) - z (0, 12ns)	- z (0, 12ns)	

Wat doen we nu?
Stoppen

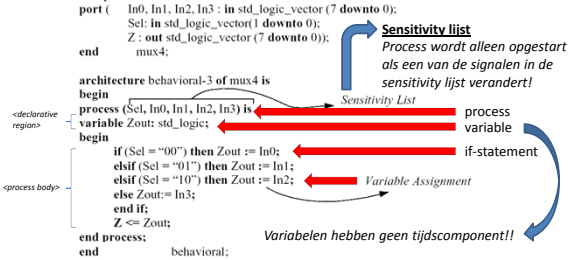
Waar zit de nauwkeurigheid van het model?
Bijv. in tijdstappen en detail

Complexe gedrag modeleren 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 complexere data structuren
- Dus: krachtigere contructen binnen VHDL nodig

De "process" statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4 is
port (    In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
          Sel: in std_logic_vector(1 downto 0);
          Z : out std_logic_vector (7 downto 0));
end       mux4;

architecture behavioral-3 of mux4 is
begin
process (Sel, In0, In1, In2, In3) is
variable Zout: std_logic;
begin
     if (Sel = "00") then Zout := In0;
     elsif (Sel = "01") then Zout := In1;
     elsif (Sel = "10") then Zout := In2;
     else Zout:= In3;
     end if;
     Z <= Zout;
end process;
end              behavioral;
```



De “process” construct

- Statements binnen een process worden sequentieel uitgevoerd
- De signal assignment statements worden dus genoemd: ***sequentiële signal assignment statements***
- De process body ziet uit als C of Pascal:
  - Declaratie en gebruik van variabelen
  - *If-then*, *if-then-else*, *case*, *for*, and *while* constructs
  - Body kan ook signal assignments statements bevatten
- Een process work concurrent uitgevoerd met andere concurrent signal assignment statements
- Een process kost 0 simulatietijd en kan events in de toekomst schedulen/genereren
- Een process kan worden gezien als een complexe signal assignment statement

Een paar programmeer-constructs

- If-then-else:

```
if (ingang = “0”) then
    Zout <= “1”;
end if;
```

```
if (ingang = “0”) then
    Zout <= “1”;
else
    Zout <= “0”;
end if;
```

```
if (ingang = “00”) then
    Zout <= “0”;
elsif (ingang = “01”) then
    Zout <= “0”;
elsif (ingang = “10”) then
    Zout <= “0”;
else
    Zout <= “1”;
end if;
```

- For-loops:

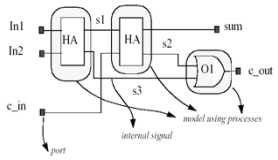
```
for INDEX in 1 to 32 loop
<loop body>
end loop;
```

- While-loops:

```
while j < 32 loop
<loop body> …
j := j + 1;
end loop;
```

- Case: (zie voorbeeld later)

Concurrent processen: Full Adder



- Elk van de componenten kan worden gemodelleerd door een process
- Processen “executeren” in parallel (concurrent)
- Processen “communiceren” met signalen

Concurrent processen: Full Adder

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

entity full_adder is
port (In1, c_in, In2: in std_logic;
      sum, c_out: out std_logic);
end          full_adder;

architecture behavioral of full_adder is
signal s1, s2, s3: std_logic;
constant delay:Time:= 5 ns;
begin
HA1: process (In1, In2) is
begin
s1 <= (In1 xor In2) after delay;
s3 <= (In1 and In2) after delay;
end process HA1;
```

```
HA2: process(s1, c_in) is
begin
sum <= (s1 xor c_in) after delay;
s2 <= (s1 and c_in) after delay;
end process HA2;

OR1: process (s2, s3) -- process
describing the two-input OR gate
begin
c_out <= (s2 or s3) after delay;
end process OR1;
end                  behavioral;
```


Concurrent processen: Half Adder

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

entity half_adder is
port (a, b : in std_logic;
sum, carry : out std_logic);
end          half_adder;

architecture behavior of half_adder is
begin
sum_proc: process(a,b) is
        begin
          if (a = b) then
              sum <= ‘0’ after 5 ns;
          else
              sum <= (a or b) after 5 ns;
          end if;
        end process;
```

```
carry_proc: process (a,b) is
begin
case a is
when ‘0’ =>
carry <= a after 5 ns;
when ‘1’ =>
carry <= b after 5 ns;
when others =>
carry <= ‘X’ after 5 ns;
end case;
end process carry_proc;
end                  behavior;
```


Processen: algemene opmerkingen

- Processen worden gebruikt om complexere gedrag van systemen te beschrijven.
- Elk concurrent signal assigment statement kan als process worden geschreven (single event).
- Processen kunnen meerdere events genereren
- Alle processen worden eenmalig gestart bij starten van simulatie, daarna bepalen de “data flow” welke worden geexecuteerd.
- Verschil in gebruik van signalen en variabelen.

Variabelen vs. Signals

```
proc1: process (x, y, z) is -- Process 1
variable var_s1, var_s2: std_logic;
begin
L1: var_s1 := x and y;
L2: var_s2 := var_s1 xor z;
L3: res1 <= var_s1 nand var_s2;
end process;
```

```
proc2: process (x, y, z) -- Process 2
begin
L1: sig_s1 <= x and y;
L2: sig_s2 <= sig_s1 xor z;
L3: res2 <= sig_s1 nand sig_s2;
end process;
```

Aanname → op tijdstip t=0 worden beide processen uitgevoerd EN hebben de volgende signalen de volgende waarden:

x=0 (t=0)
y=1 (t=0)
z=1 (t=0)
sig_s1=1 (t=0)
sig_s2=1 (t=0)

→ Zijn res1 en res2 hetzelfde in waarde na de "berekening"?

Hoe modeleer ik een D-flipflop?

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

entity dff is
port (   D, clk : in  std_logic;
         Q, Qbar: out std_logic);
end dff;

architecture gedrag1 of dff is
begin
output: process (clk) is
begin
  if (clk'event and clk="1") then
         Q <= D;
         Qbar <= not(D);
  end if;
end process;
end gedrag1;
```

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

entity dff is
port (   D, clk : in  std_logic;
         Q, Qbar: out std_logic);
end dff;

architecture gedrag1 of dff is
begin
output: process is
begin
  wait until(clk'event and clk="1");
         Q <= D;
         Qbar <= not(D);

end process;
end gedrag1;
```

Wat zijn de verschillen?

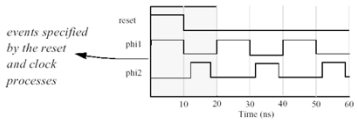
Wait statements

- Wachten voor het verstrijken van tijd:
  - wait for <*time expression*>
- Wachten op een signaal:
  - wait on <*signal*>
- Wachten totdat "iets waar" is:
  - Wait until <*boolean expression*>
- Er mogen meerdere wait statements binnen een process worden gebruikt.
- Expliciete controle van stoppen en herstarten binnen een process
- Maakt mogelijk om synchrone en asynchrone events binnen een digitale systeem te representeren.
- Wait statements en sensitivity lijsten mogen nooit tegelijkertijd binnen een dezelfde process worden gebruikt!! (Bedenk zelf waarom)

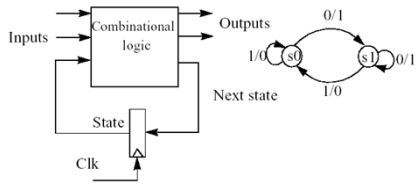
Clock generatie met wait statements

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;

architecture behavioral of two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
```

```
clock_process: process is
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns, '0' after
18 ns;
wait for 20 ns;
end process clock_process;
end architecture behavioral;
```



Modellering van state machines



2 basiscomponenten:

- Combinatorisch component: output en "next state" generatie
- Sequentieel component

Voorbeeld VHDL code

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

entity state_machine is
port (   reset, clk, x : in  std_logic;
         z : out std_logic);
end state_machine;

architecture gedrag1 of state_machine is
type statetype is (state0, state1);
Signal state, next_state: statetype := state0;
begin

comb_proc: process (state, x) is
begin
-- combinatorisch process beschrijving
end process;

seq_proc: process is
begin
-- sequentieel process beschrijving
end process;

end gedrag1;
```

```
case state is
when state0 =>
  if x='0' then
      next_state<=state1; z<='1';
  else
      next_state<=state0; z<='0';
  end if;
when state1 =>
  if x='1' then
      next_state<=state0; z<='0';
  else
      next_state<=state1; z<='1';
  end if;
end case;
```

```
wait until (clk'event and clk='1')
  if (reset='1') then
      state <= state0;
  else
      state <= next_state;
  end if;
```

Hiërarchie van componenten

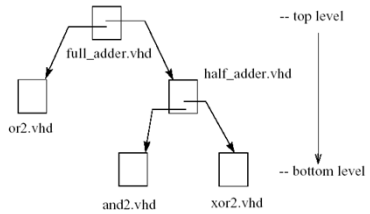
```
architecture structural of full_adder is
component half_adder is -- the declaration
port (a, b: in std_logic;
      sum, carry: out std_logic);
end component  ;

component or_2 is
 port(a, b : in std_logic;
      c : out std_logic);
end component  ;

signal s1, s2, s3 : std_logic;
begin
H1: half_adder port map (a => In1, b => In2, sum=>s1, carry=>s3);
H2:half_adder port map (a => s1, b => c_in, sum =>sum,
            carry => s2);
O1: or_2 port map (a => s2, b => s3, c => c_out);
end structural;
```

component instantiation statement

```
entity full_adder is
port (In1,In2: in std_logic;
      c_in: in std_logic
      sum,c_out: out std_logic);
end full_adder;
```

Componenten kunnen zelf weer zijn opgebouwd door 'kleinere' componenten → hierarchie

Hiërarchie en abstractie



- Geneste structurele beschrijvingen produceren hiërarchische modellen
- Hierarchie wordt "platgemaakt" voordat simulatie wordt begonnen
- De componenten op de laagste niveau moeten een functionele beschrijving hebben (dus niet structureel)

Generics

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

entity xor2 is
generic (gate_delay : Time:= 2 ns);
port(In1, In2 : in std_logic;
     z : out std_logic);
end entity xor2;

architecture behavioral of xor2 is
begin
z <= (In1 xor In2) after gate_delay;
end architecture behavioral;
```

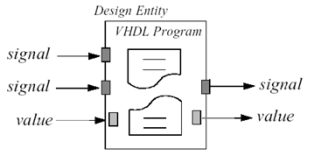
Met gebruik van generics kunnen geparameterizeerde modellen worden gemaakt.

Generics in Hierarchische modellen

```
architecture generic_delay of half_adder is
component xor2
generic (gate_delay: Time);
port (a, b : in std_logic;
      c : out std_logic);
end component;

component and2
generic (gate_delay: Time);
port (a, b : in std_logic;
      c : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => 6 ns)
          port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 3 ns)
          port map(a=> a, b=> b, c=> carry);
end generic_delay;
```

Precedence rules van generics:
- De 'generic map' waarde
- De default gedefineerde waarde
- De default waarde van gebruikte type

Eigenschappen van generics

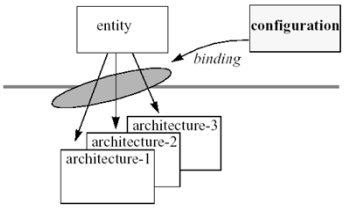


- Generics zijn constant objecten en kunnen alleen worden gelezen
- De waarden van generics moeten bekend zijn tijdens compile-tijd
- Generics maken deel uit van de interface van een entity, maar hebben geen fysische equivalent
- Het gebruik van generics is niet alleen gelimiteerd tot "delay" parameters

Voorbeel van een N-input gate

```
 entity generic_or is
generic (n: positive:=2);
port (in1 : in std_logic_vector ((n-1) downto 0);
      z : out std_logic);
end          generic_or;

architecture behavioral of generic_or is
begin
process (in1)
variable sum : std_logic:= '0';
begin
sum := '0'; -- on an input signal transition sum must be reset
for i in 0 to (n-1) loop
sum := sum or in1(i);
end loop;
z <= sum;
end process;
end                  behavioral;
```

Configuraties en Binding Rules



Configuraties:

- Een entity kan meerdere verschillende architectuur-beschrijvingen hebben
- De configuratie beschrijft welke architectuur-beschrijving te gebruiken

Binding Regels:

- Welke architectuur-beschrijving gebruiken??

1. Vind de entity met dezelfde naam
2. "bind" de laatste gecompileerde architecture-beschrijving met die entity

- Hoe kunnen meer controle op "binding" uitoefenenen? → → →

Configuratie specificatie

```
architecture structural of full_adder is
-
--declare components here
signal s1, s2, s3: std_logic;
--
-- configuration specification
for H1: half_adder use entity WORK.half_adder (behavioral);
for H2: half_adder use entity WORK.half_adder (structural);
for O1: or_2 use entity POWER.lpo2 (behavioral)
generic map(gate_delay => gate_delay)
port map (I1 => a, I2 => b, Z=>c);

begin    -- component instantiation statements
H1: half_adder port map (a =>In1, b => In2, sum => s1, carry=> s2);

H2: half_adder port map (a => s1, b => c_in, sum => sum, carry => s2);

O1: or_2 port map(a => s2, b => s3, c => c_out);
end structural;
```

library name
entity name
architecture name