Digitale Systemen (EE1 410)

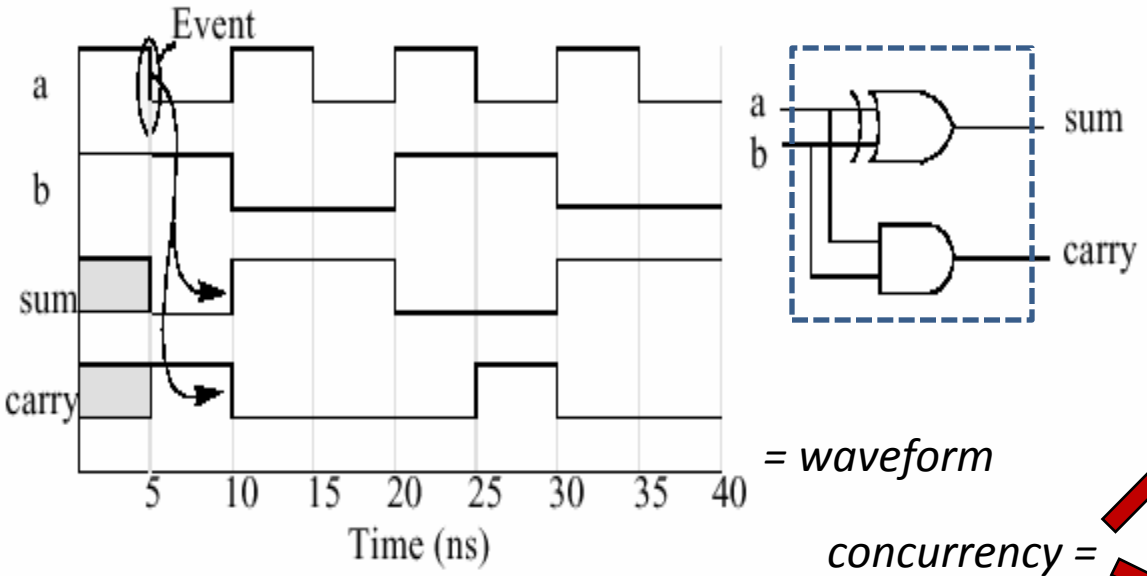
Arjan van Genderen
Stephan Wong

Faculteit EWI
Technische Universiteit Delft
Cursus 2011

Samenvatting 1ste 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 2de college

- 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
- Testbenches

Het “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;
```

Sensitivity lijst

Process wordt alleen opgestart als een van de signalen in de sensitivity lijst verandert!

Sensitivity List

<declarative region>

<process body>

process

variable

if-statement

Variable Assignment

Variabelen hebben geen tijdscomponent!!

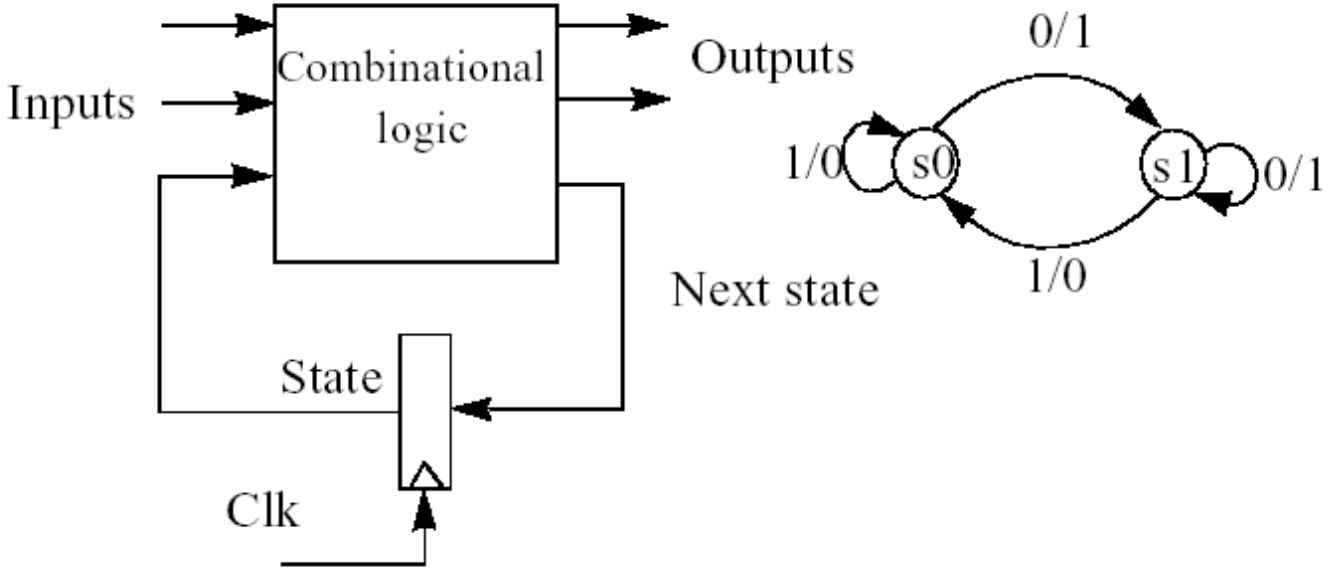
Hoe modeleer ik een D-flipflop?

```
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?

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

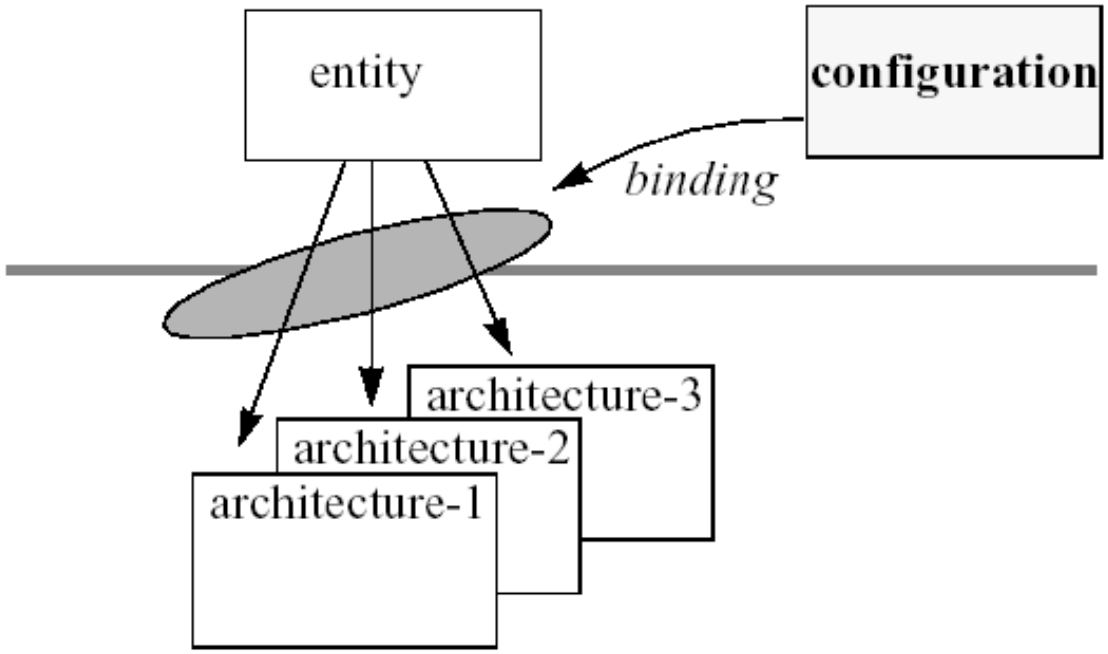
```
architecture structural of full_adder is
component half_adder is -- the declaration
port (a, b: in std_logic;
      sum, carry: out std_logic);
end component  ;

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

Configuraties en Binding Rules



Configuraties:

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

Binding Regels:

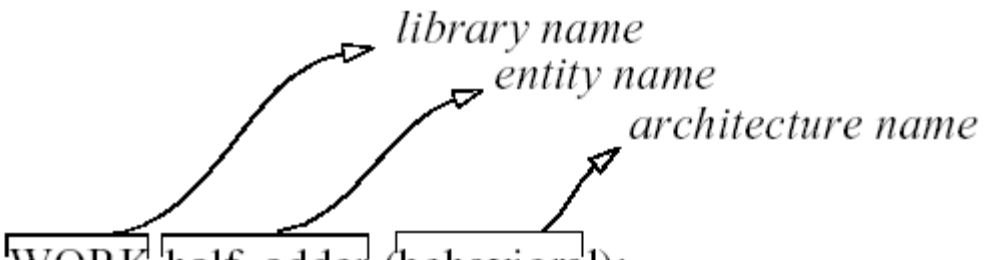
- Welke architectuur-beschrijving gebruiken??

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

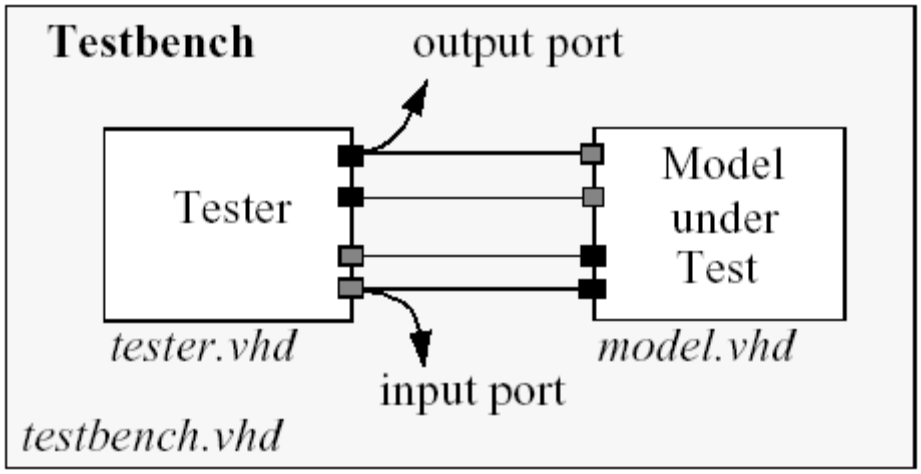
- Hoe kunnen we meer controle op “binding” uitoefenenen? → → →

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;
```



Testbenches




```
library IEEE; use IEEE.std_logic_1164.all;
entity testbench is -- testbench
end entity testbench;
architecture behav of testbench is
-- put component declarations
-- configuration specification
begin
-- connect tester with the models under test
end behav;
```

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

entity tester is -- generator of test signals
port (A, clk : out std_logic; B : in std_logic);
end entity tester;

architecture behav of tester is
begin
-- generate clocks
clk_process: process
begin
clk <= '1', '0' after 10 ns, ..; wait for 40 ns;
end clk_process;
-- other processes generating signals (out)
-- other processes checking inputs (in)
end behav;
```

Onderwerpen van vandaag

- Attributes
- Function
- Procedure
- Package
- Library
- Van simulatie naar synthese
- FPGA als “prototyping platform”

- Samenvoegen van vaakgebruikte code

Attributes

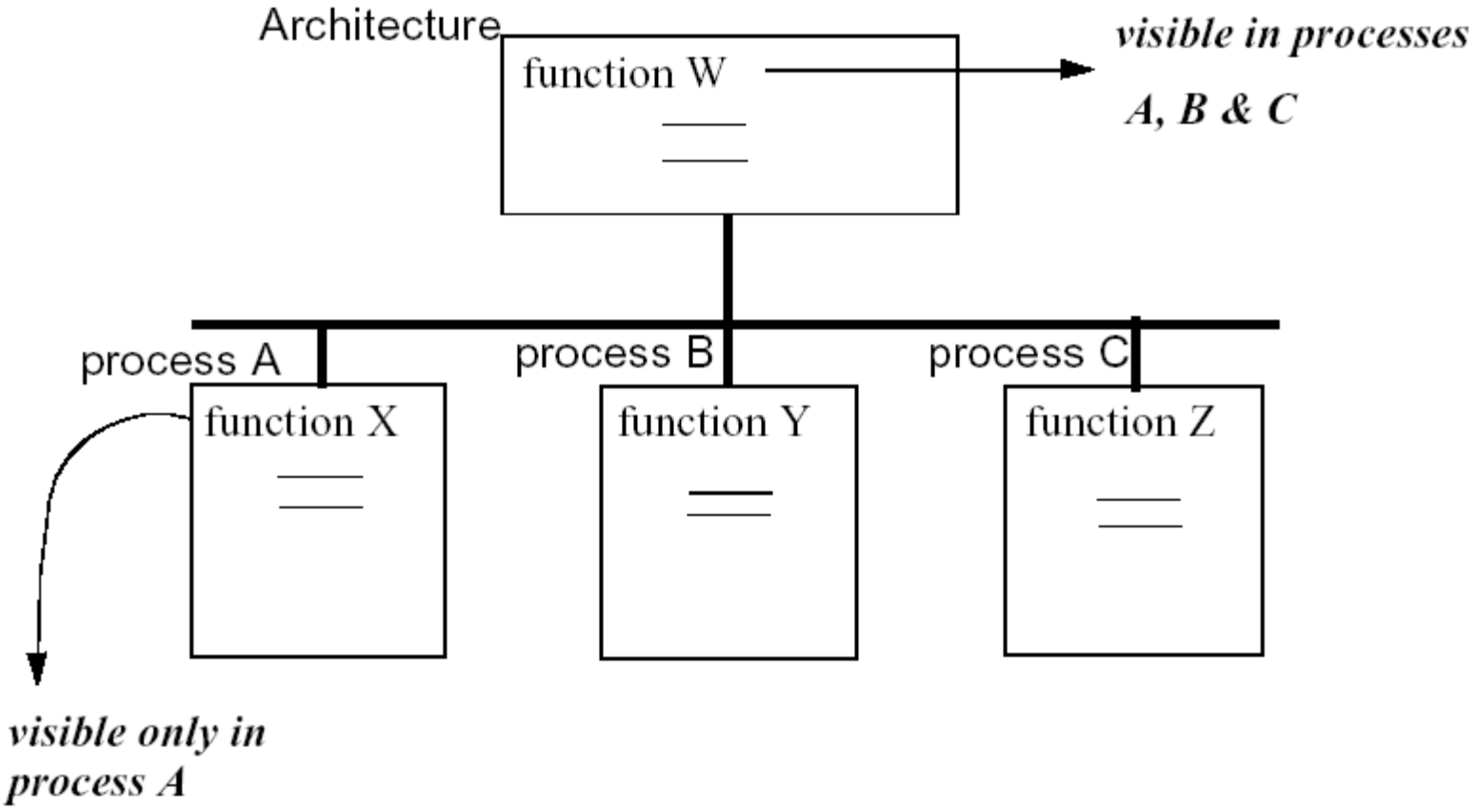
- Informatie verkrijgen van VHDL types zoals ‘types’, ‘arrays’, en ‘signals’ → *object’**attribute***
- Type attributes:
  - **Value attributes** *(returns a constant value)*
    type statetype **is** (state0, state1, state2, state3);
    e.g., statetype‘**left**, statetype’**right**, array_name’**length**, etc.
  - **Function attribute** *(invokes function that returns a value)*
    e.g., clk’**event**, signal_name’**last_event**, etc.
    e.g., array_name’**left**, array_name’**right**, array_name’**high**, etc.
  - **Signal attributes** *(creates a new signal)*
    e.g., signal_name’**delayed(T)**, signal_name’**last_value**
  - **Type attributes** *(returns a type)*
  - **Range attributes** *(returns a range)*
    e.g., value_range’**range**

VHDL "function"

```
function rising_edge (signal clock: std_logic) return boolean is
-- declarative region : declare variables local to the function
--
begin
-- body
--
return (expression);
end rising_edge;
```

- "**formal parameters**" zijn: constants, signals, en files & *mode* **in**
- Default parameter is: constant
- Functies kunnen de parameters niet veranderen
- Bij aanroep van functie moeten de type van parameters kloppen
- Functies kunnen geen wait statements bevatten

Plaatsing van functies



- Als meerdere entities functies gebruiken – plaats ze in packages

Voorbeelden van functies (I)

```
architecture behavioral of dff is
function rising_edge (signal clock : std_logic) return boolean is
variable edge : boolean:= FALSE;
begin
edge := (clock = '1' and clock'event);
return (edge);
end rising_edge;

 begin
 output: process
 begin
 wait until (rising_edge(Clk));

     Q <= D after 5 ns;
     Qbar <= not D after 5 ns;

 end process output;
 end architecture  behavioral;
```

Architecture Declarative Region

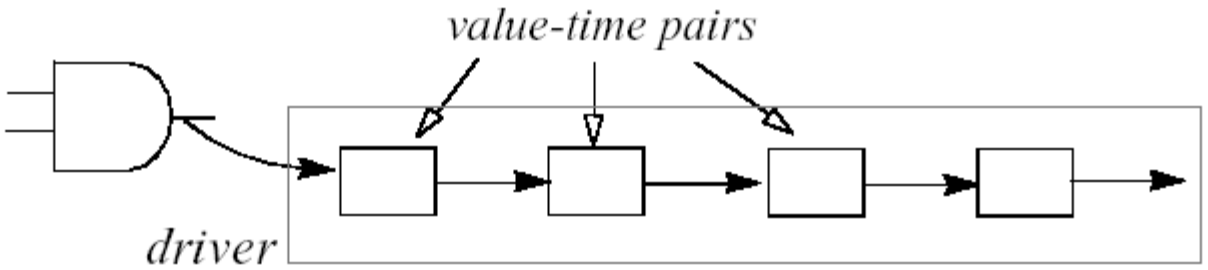
Voorbeelden van functies (II)

```
function to_bitvector (svalue : std_logic_vector) return bit_vector is
variable outvalue : bit_vector (svalue'length-1 downto 0);
begin
for i in svalue'range loop -- scan all elements of the array
case svalue (i) is
when '0' => outvalue (i) := '0';
when '1' => outvalue (i) := '1';
when others => outvalue (i) := '0';
end case;
end loop;
return outvalue;
end to_bitvector
```

- Type conversie functies → interoperability
- Gebruik van attributes for flexibele functie definities
- Bekijk packages voor voorbeelden

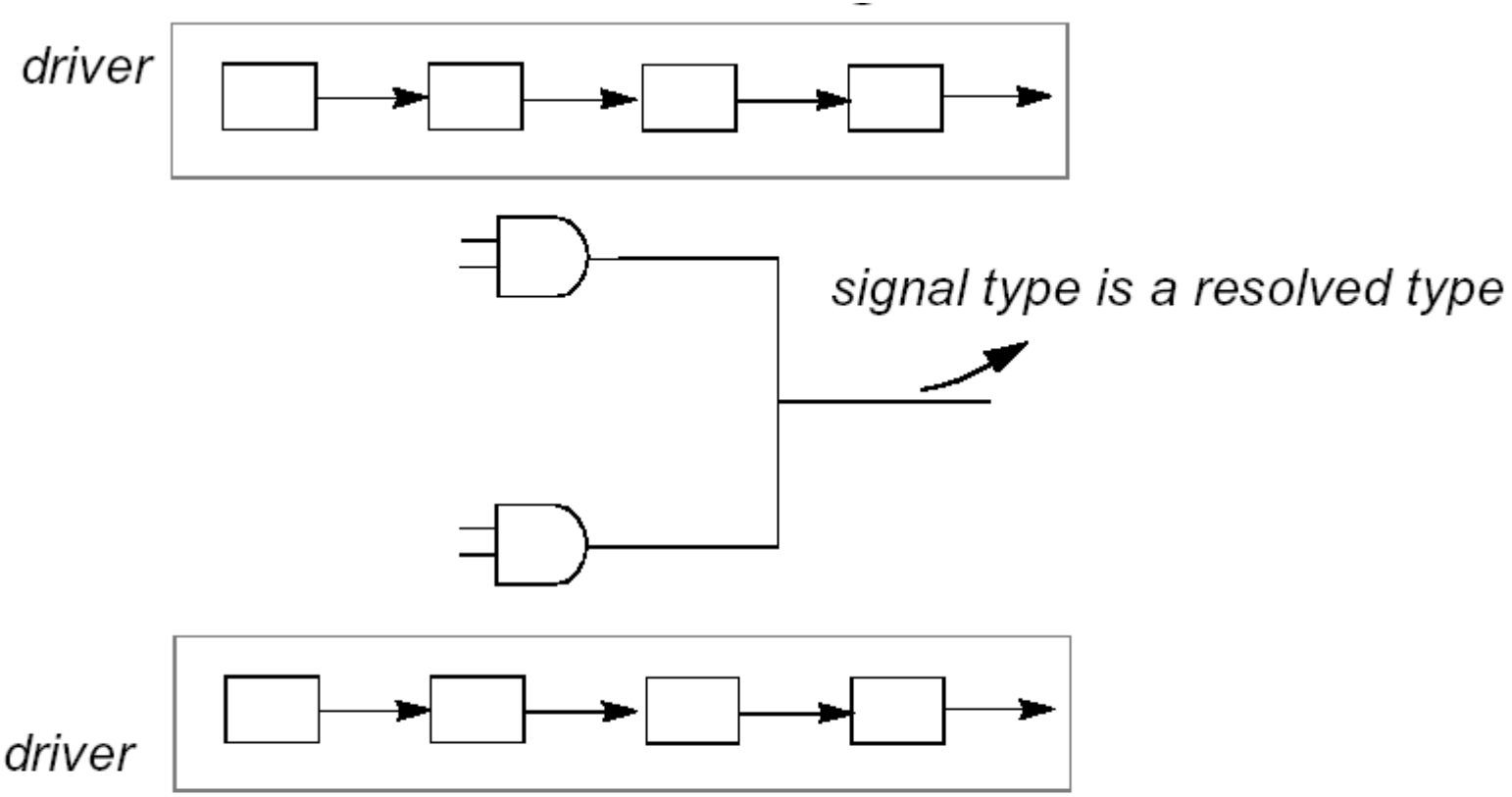
Implementatie van signalen

- The structure of a signal assignment statement
  - *signal* <= (*value expression* **after** *time expression*)
  - RHS is referred to as a *waveform element*
- Every signal has associated with it a *driver*



- holds the current and future values of the signal - a *projected waveform*
- signal assignment statements modify the driver of a signal
- value of a signal is the value at the head of the driver
- note the hardware analogy

Shared signalen



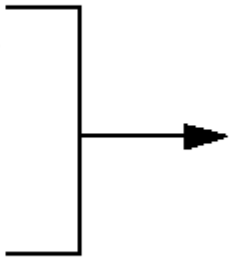
- Resolution function is invoked whenever an event occurs on this signal
- Resolution must be an associative operation (x *op* y) *op* z = x *op* (y *op* z)

Resolved types: std_logic

```
type std_ulogic is (
'U', -- Uninitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- Weak Unknown
'L', -- Weak 0
'H', -- Weak 1
'-' -- Don't care
 );
```

supports only single drivers

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

subtype std_logic is resolved std_ulogic;
```



supports multiple drivers

Resolution function: std_logic & resolved

- resolving values for std_logic types

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

- Pairwise resolution of signal values from multiple drivers
- Resolution operation must be associative

Overzicht gebruik van functies

- Plaatsing van functies bepaalt de ‘visibility’
- Formal parameters (mode **in**)
- Gebruik van wait statements verboden
- Bekijk vele voorbeelden in source listings van bijgeleverde packages van software (zullen we doen tijdens het practicum)

VHDL ‘procedure’

```
procedure proc (variable f: in std_logic; v : out std_logic_vector)
-- declarative region : declare variables local to the function
--
begin
-- body
--
end proc;
```

- “**formal parameters**” zijn: variables, constants, signals, en files
- Modes kunnen zijn: **in, out, inout**
- Default type van parameters is: **constant** als mode **in**, anders **variable**
- Bij aanroepen van procedures worden de variabelen altijd opnieuw geinitialiseerd

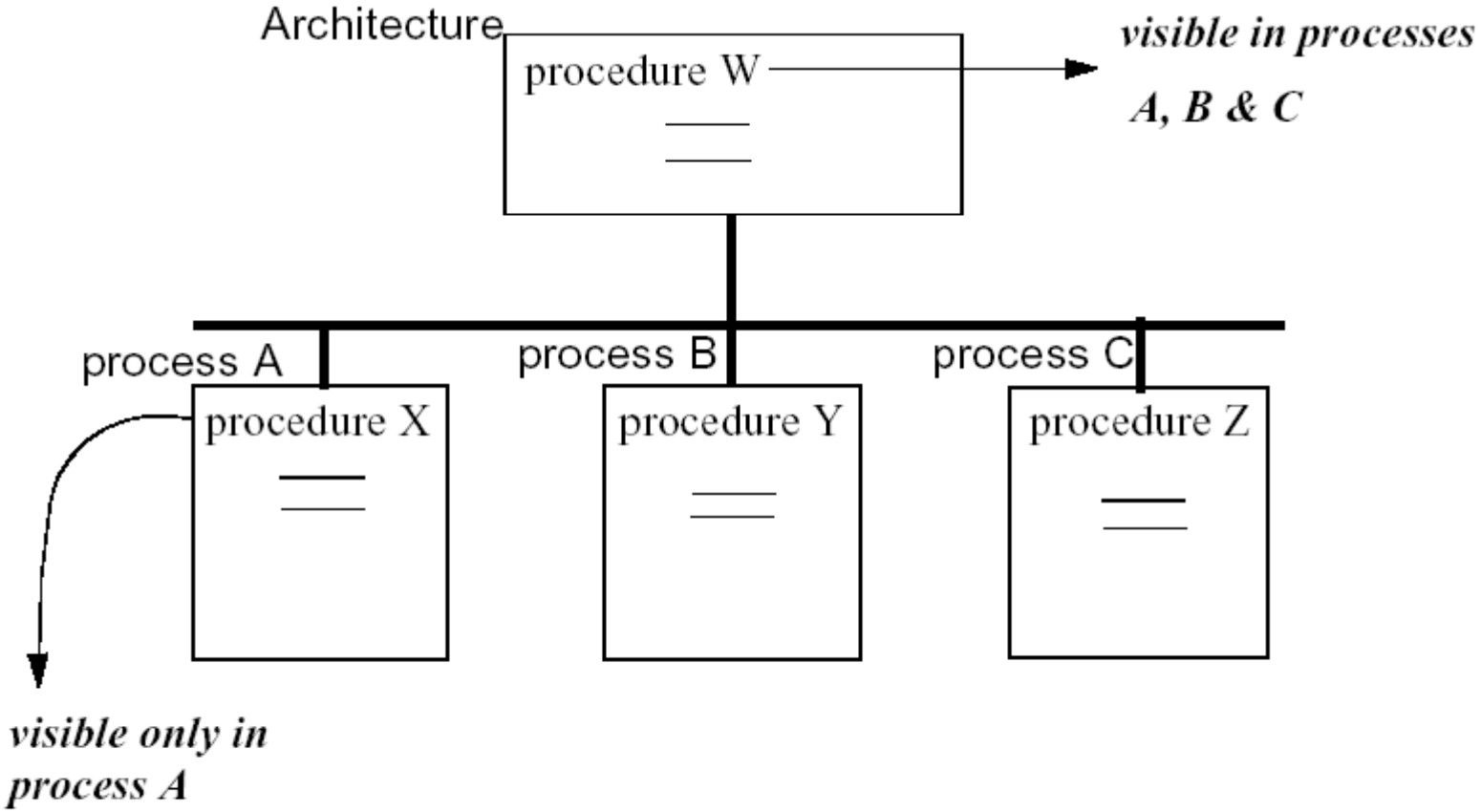
Procedures: locatie (I)

```
architecture behavioral of cpu is
--
-- decalarative region
-- procedures can be placed in their entirety here
--
begin
    process_a: process
    -- declarative region of a process
    -- procedures can be placed here
    begin
    --
    -- process body
    --
    end process_a;
    process_b: process
    --declarative regions
    begin
    -- process body
    end process_b;
end behavioral;
```

visible to all processes

visible only within process_a

Procedures: locatie (II)

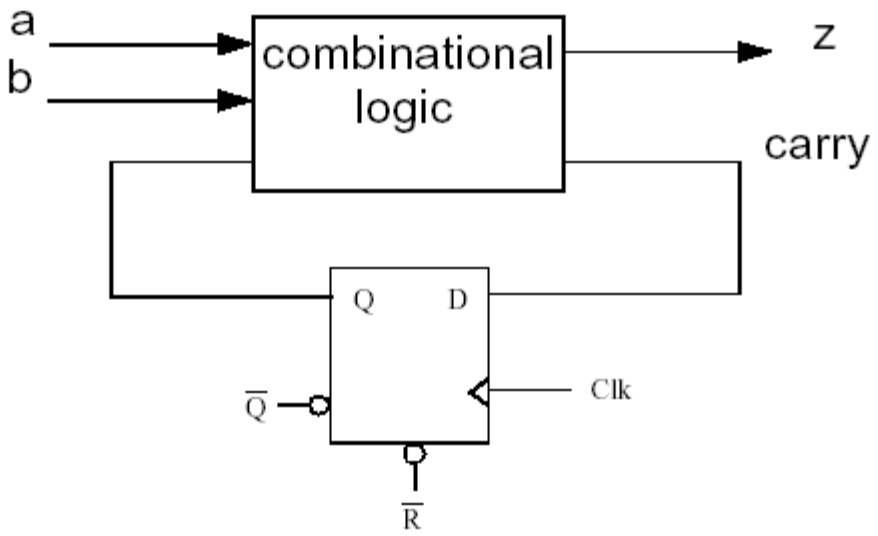


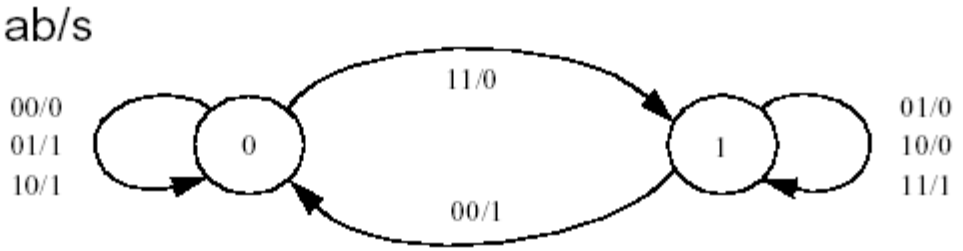
- placement of procedures determines visibility in its usage

Procedures en signalen

- Procedures kunnen signal assignments uitvoeren op uitgangssignalen van de parameterlijst
- Procedures kunnen geen wait statements bevatten als de process (waarin ze worden gebruikt) al een sensitivity lijst gebruikt
- Procedures kunnen ook signal assignments uitvoeren op signalen die niet in de parameter lijst staan -- *dus goed oppassen als je code "snijdt" uit grote applicaties en in een procedure stopt*

Concurrent vs. sequential procedure calls





- Example: bit serial adder

Concurrent procedure call

```
architecture structural of serial_adder is
component comb
 port (a, b, c_in : in std_logic;
        z, carry : out std_logic);
end component;

procedure dff(signal d, clk, reset : in std_logic;
                signal q, qbar : out std_logic) is
begin
 if (reset = '0') then
      q <= '0' after 5 ns;
      qbar <= '1' after 5 ns;
    elsif (clk'event and clk = '1') then
    q <= d after 5 ns;
    qbar <= (not D) after 5 ns;
 end if;
end dff;

signal s1, s2 : std_logic;
```

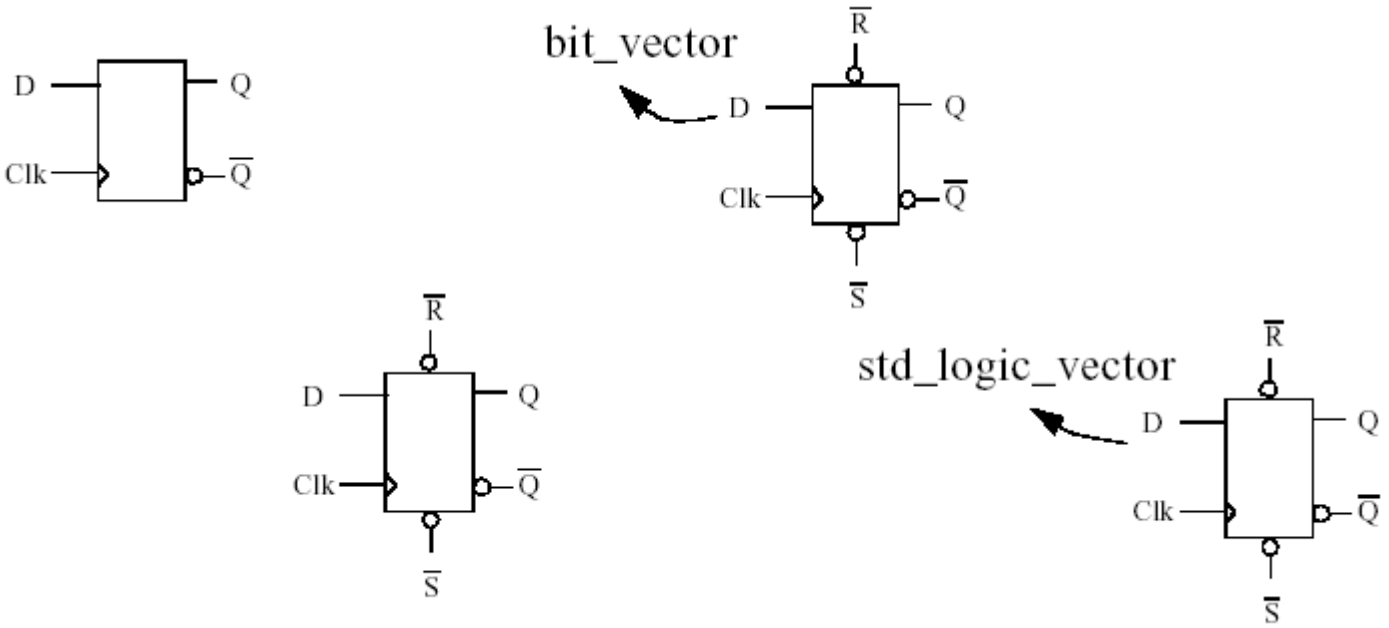
```
begin
C1: comb port map (a => a, b => b,
c_in => s1, z =>z, carry => s2);
--
-- concurrent procedure call
--
dff(clk => clk, reset =>reset, d=> s2,
q=>s1, qbar =>open);
end structural;
```

Equivalente sequential procedure call

```
architecture structural of serial_adder is
component comb
 port (a, b, c_in : in std_logic;
        z, carry : out std_logic);
end component;

procedure dff(signal d, clk, reset : in std_logic;
                signal q, qbar : out std_logic) is
begin
 if (reset = '0') then
      q <= '0' after 5 ns;
      qbar <= '1' after 5 ns;
    elsif (clk'event and clk = '1') then
    q <= d after 5 ns;
    qbar <= (not D) after 5 ns;
 end if;
end dff;

signal s1, s2 : std_logic;
```

```
begin
C1: comb port map (a => a, b => b,
c_in => s1, z =>z, carry => s2);
--
-- sequential  procedure call
--
process
begin
dff(clk => clk, reset =>reset, d=> s2,
q=>s1, qbar =>open);
wait on clk, reset,s2;
end structural;
```

Subprogram overloading (I)



- Hardware componenten kunnen vele verschillende ingangen hebben en ook de gebruikte typen kunnen ook verschillen
- → gebruik verschillende procedures om de functionaliteit van verschillende componenten te beschrijven
- → moeten we nu steeds een andere naam gebruiken voor elke procedure?

Subprogram overloading (II)

- Beschouw de volgende procedures voor de getoonde componenten:
  - **dff_bit** (clk, d, q, qbar);
  - **asynch_dff_bit** (clk, d, q, qbar, reset, clear);
  - **dff_std** (clk, d, q, qbar);
  - **asynch_dff_std** (clk, d, q, qbar, reset, clear);
- Alle getoonde componenten kunnen dezelfde procedure naam gebruiken → subprogram overloading
- De geschikte procedure wordt gekozen op basis van procedure call argumenten

Kenmerken van ‘packages’

- Package declaration:
  - Declaration van functies, procedures, en typen die worden gebruikt in een package
  - Wordt gebruikt als een package interface
  - Alleen de gedeclareerde inhoud is ‘visible’ voor gebruik
- Package body:
  - Implementatie van functies en procedures gedeclareerd in the package header
  - ‘Instantiation’ van constanten in de package header
- Gebruik van **use**-clause

Voorbeeld: package header std_logic_1164

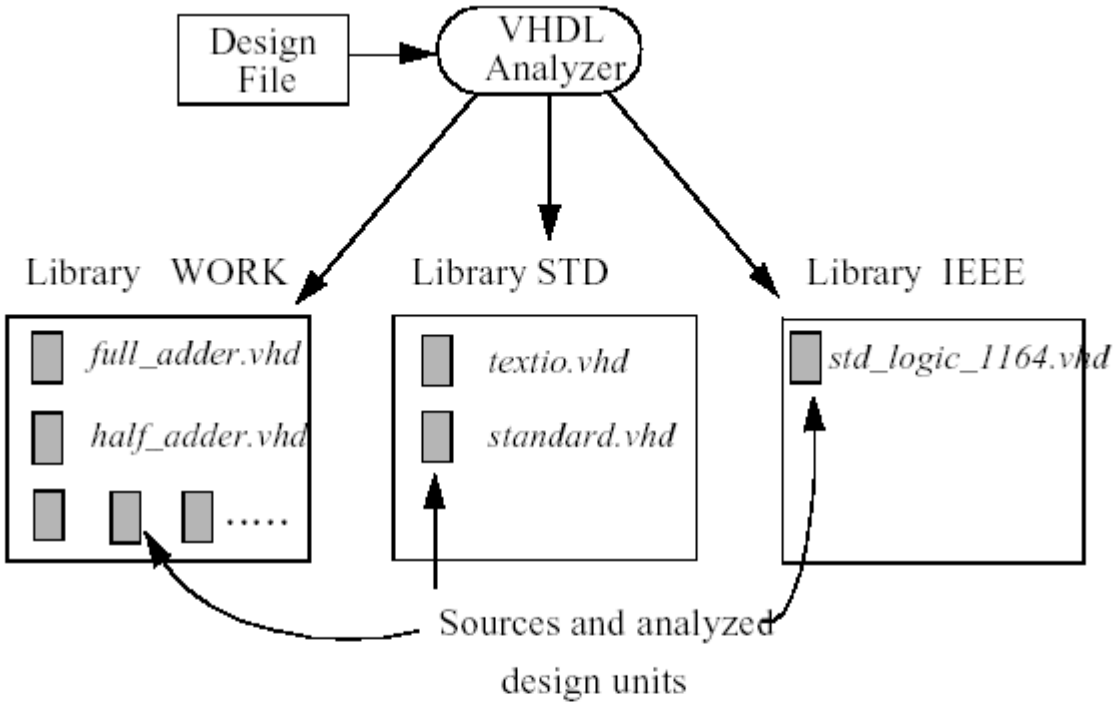
```
package std_logic_1164 is
 type std_ulogic is ('U', --Unitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L',  -- Weak 0
                     'H', -- Weak 1
                     '-' -- Don't care
                    );
type std_ulogic_vector is array (natural range <>) of std_ulogic;

function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;

type std_logic_vector is array (natural range <>) of std_logic;

function "and" (l, r : std_logic_vector) return std_logic_vector;
--..<rest of the package definition>
end package std_logic_1164;
```

Voorbeeld: package body

```
package body my_package is
--
-- type definitions, functions, and procedures
--
end my_package;
```

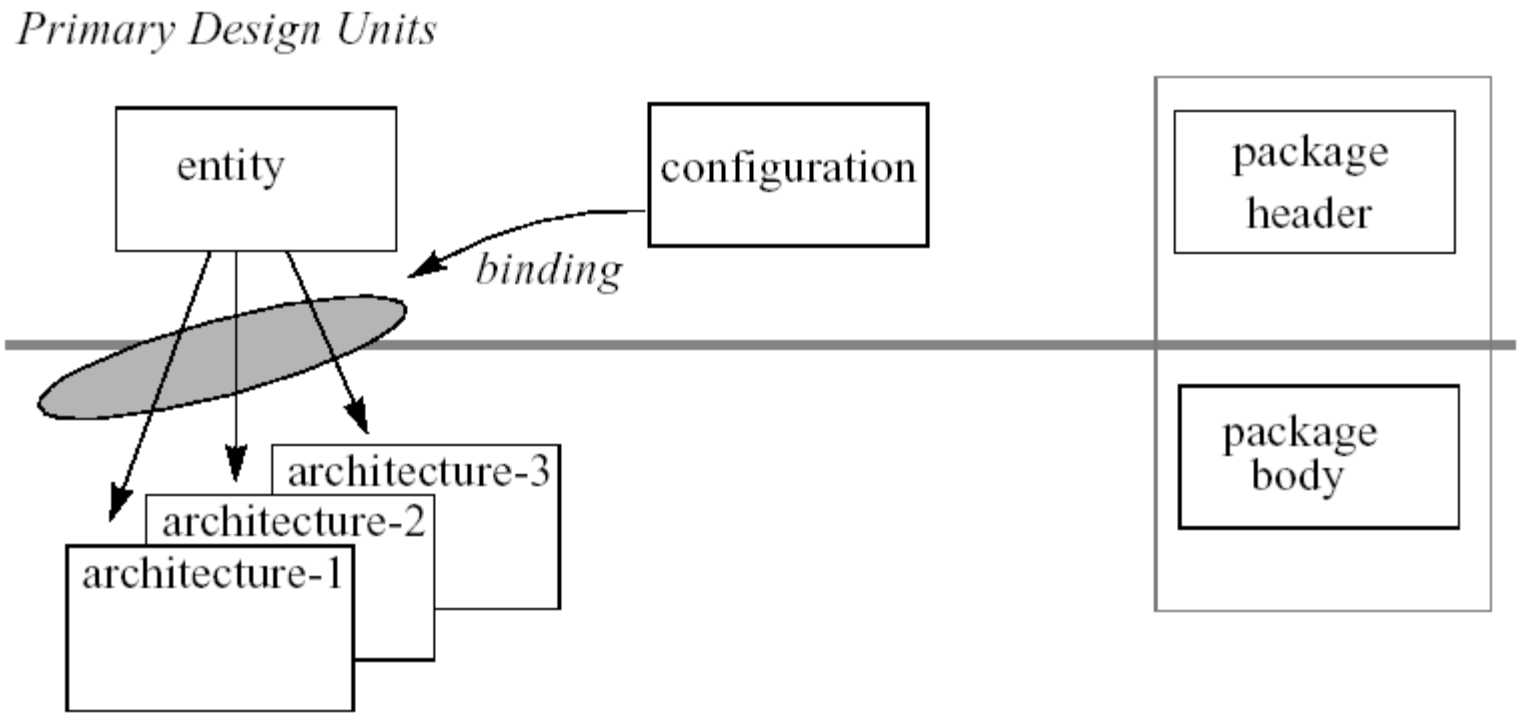
- Packages worden verzameld in libraries
- Nieuwe types moeten geassocieerde definities van operaties zoals logische operaties (bv., and, or) en wiskundige operaties (bv., +, *)
- Bekijk de std_logic_1164 library voor voorbeelden

Libraries



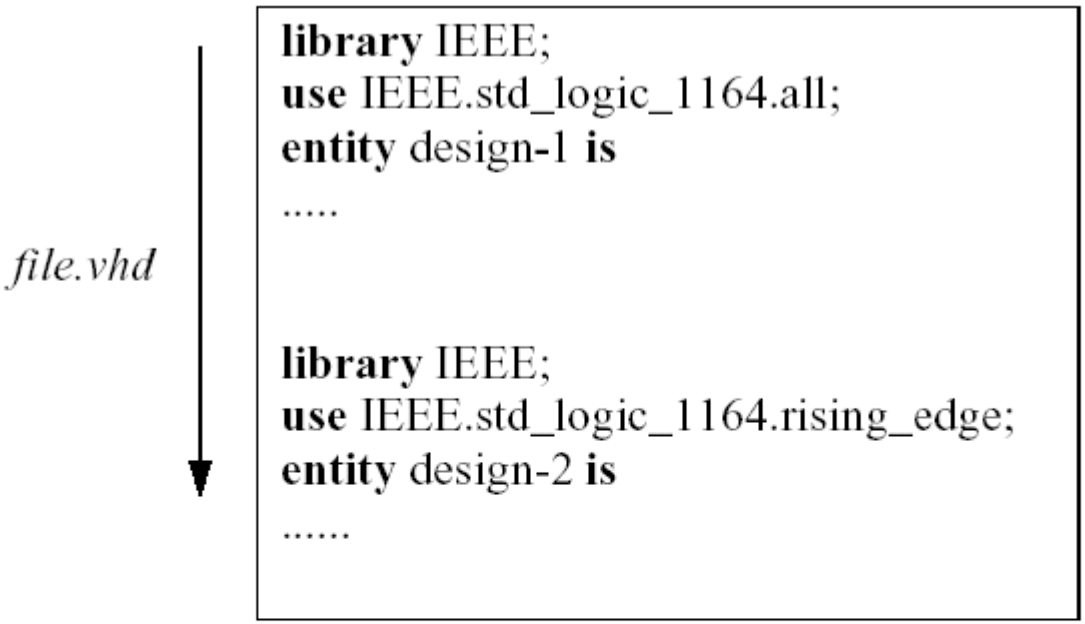
- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared

Design Units



- Distinguish the primary design units

Visibility rules



- when multiple design units are in the same file visibility of libraries and packages must be established for each ***primary*** design unit (entity, package header, configuration) separately!
- The **use** clause may selectively establish visibility, e.g., only the function rising_edge() is visible within entity design-2

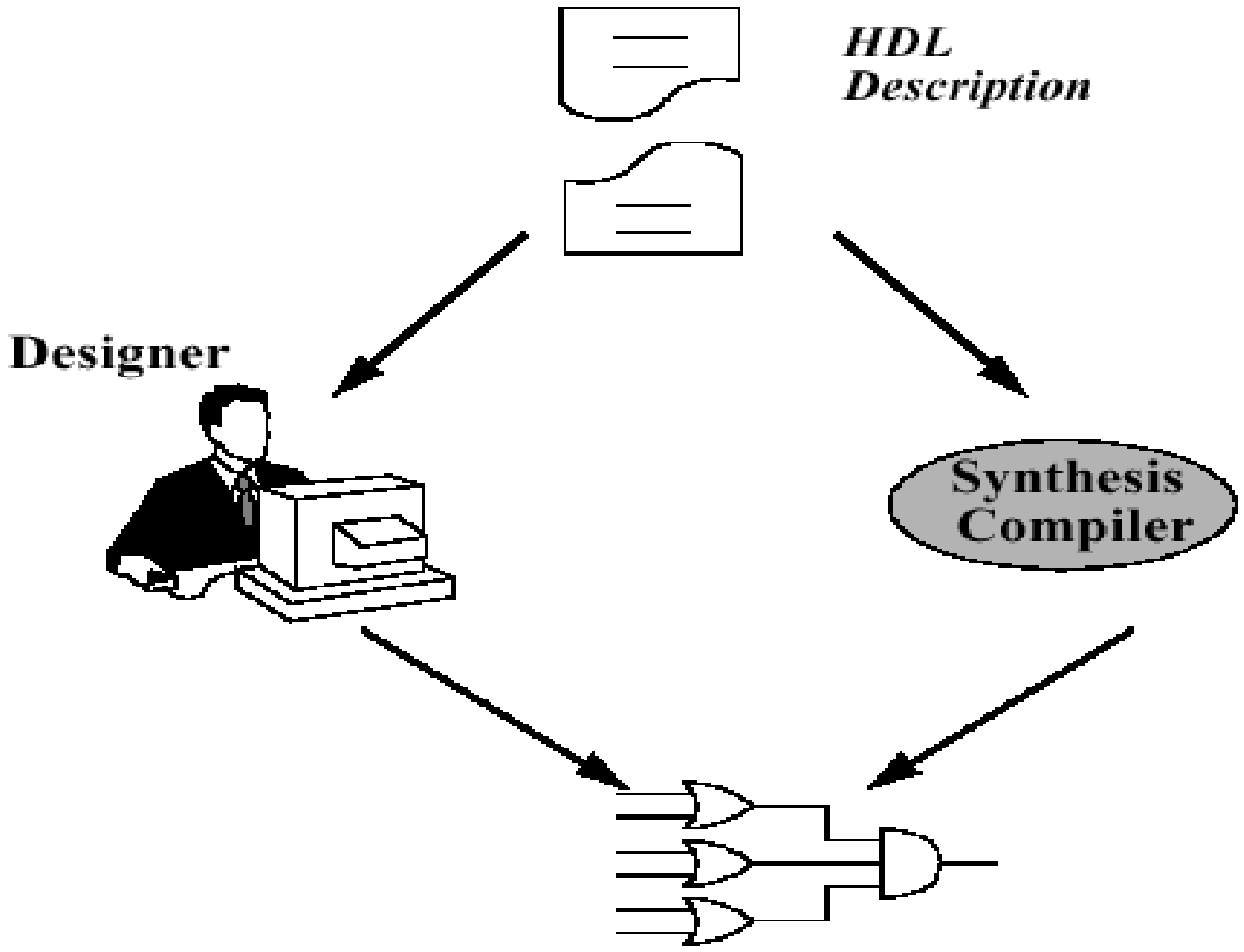
Simulatiestappen

- Analyse en volgorde van analyse (compilatie)
  - Primary vs. secondary design units
  - Organisatie van design units en files
- Elaboration (“uitwerken”) van:
  - Hierarchy
  - Generics
  - Storage allocation
  - Initialization
- Initialisatie:
  - Alle processen worden uitgevoerd totdat ze zijn “suspended”
  - Initialisatie van simulatietijd
- Simulatie:
  - Discrete event simulator
  - Two step model of time:
    - Bepaal nieuwe waarden
    - Voer alle “sensitive” processen uit
  - Simulatie tijdstappen

Van simulatie naar synthese

- Simulatie = bepalen van gedrag VHDL model
- Synthese = genereren van hardware aan de hand van VHDL model
- Goede simulatieresultaten bieden geen garantie voor goede synthese
  - Denk goed na (vergeet niet de reden van de delta delay)
- Denk altijd aan echte hardware:
  - Verschil tussen variabelen en signalen
  - Betekenis sensitivity-lijsten
- Schrijven van synthetiseerbare VHDL code heeft andere regels (onderwerp volgende college – een paar voorbeelden in volgende slides)

Synthesis and Hardware Inference

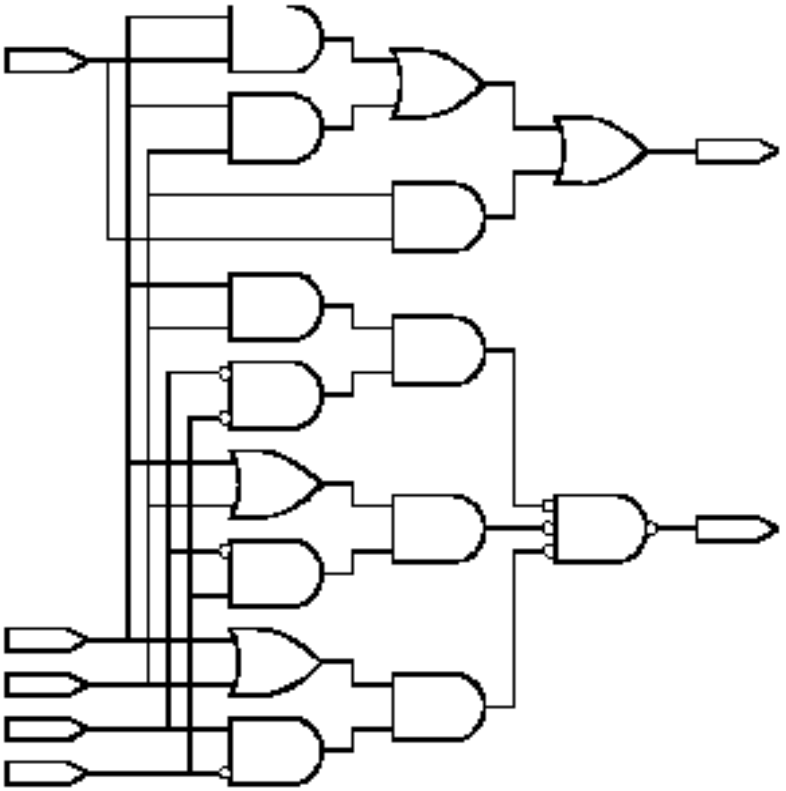


- Both processes can produce very different results!

Inferring Combinational Logic

```
--
-- pseudo code for a single bit arithmetic/ logic unit
--
s1 <= in1 and in2;
s2 <= in1 or in2;
s3 <= in1 xor in2; -- perform the sum operation
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';
```

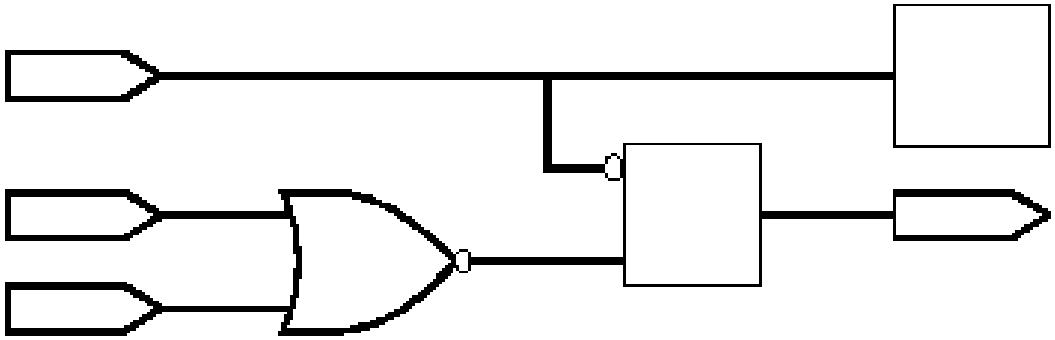
Reserved word



- Each statement implies a logic component

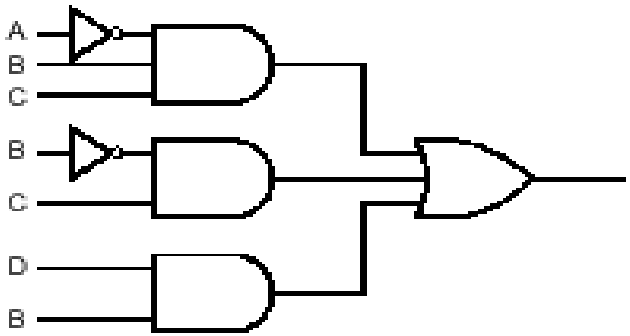
Inferring Sequential Logic

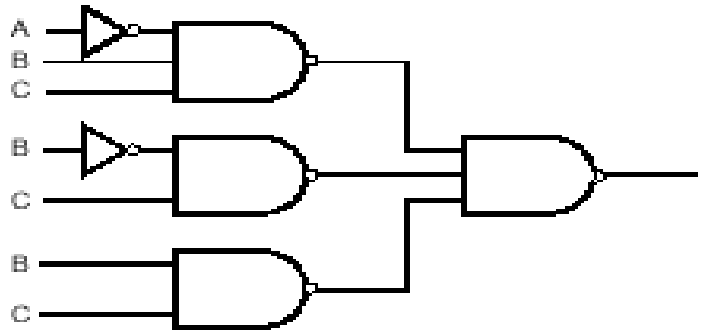
```
--
-- a simple conditional code block
--
if (sel = '0') then
z <= in1 nor in2;
end if;
```

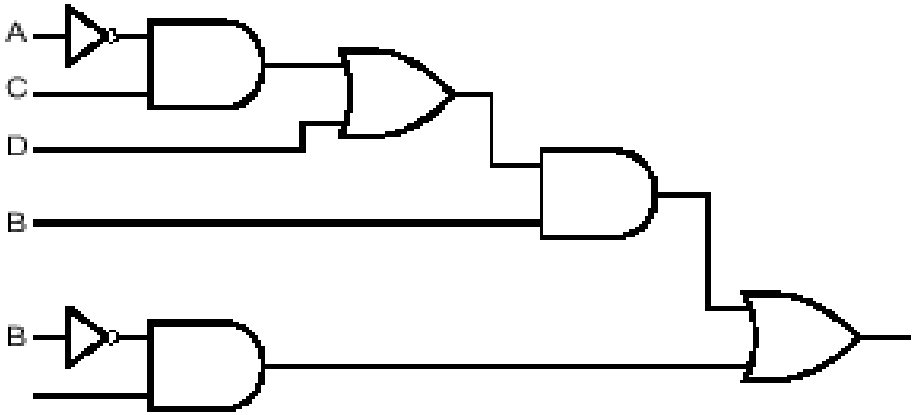


- Inference of sequential components is more subtle
- Results are very sensitive to the manner in which code is written
- Inferences of latches vs. flip flops

Target Primitives





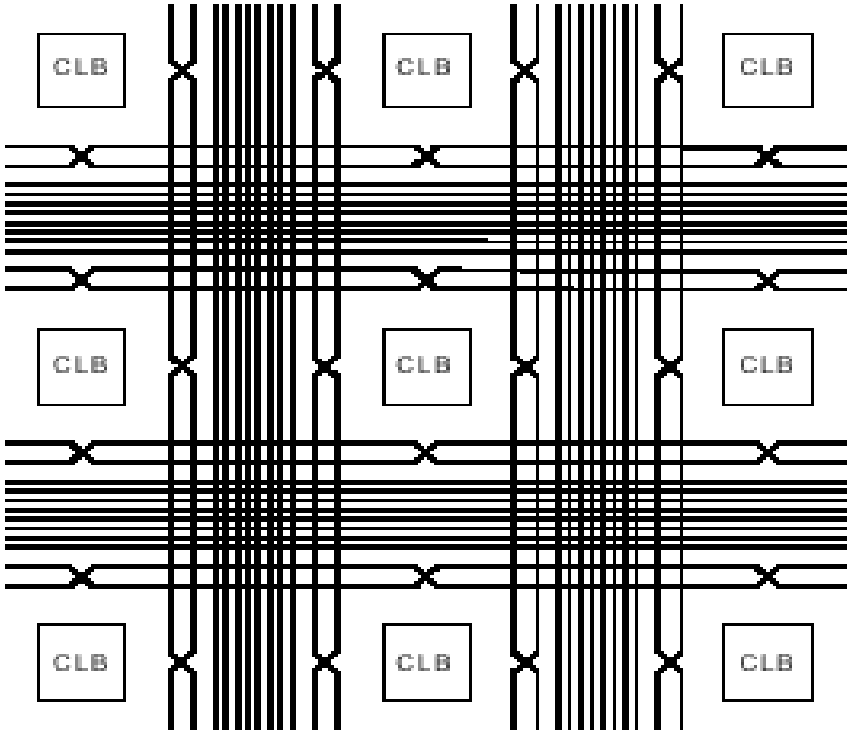


$$\overline{A}BC + \overline{B}C + DB$$

- The resulting circuit depends on the available building blocks

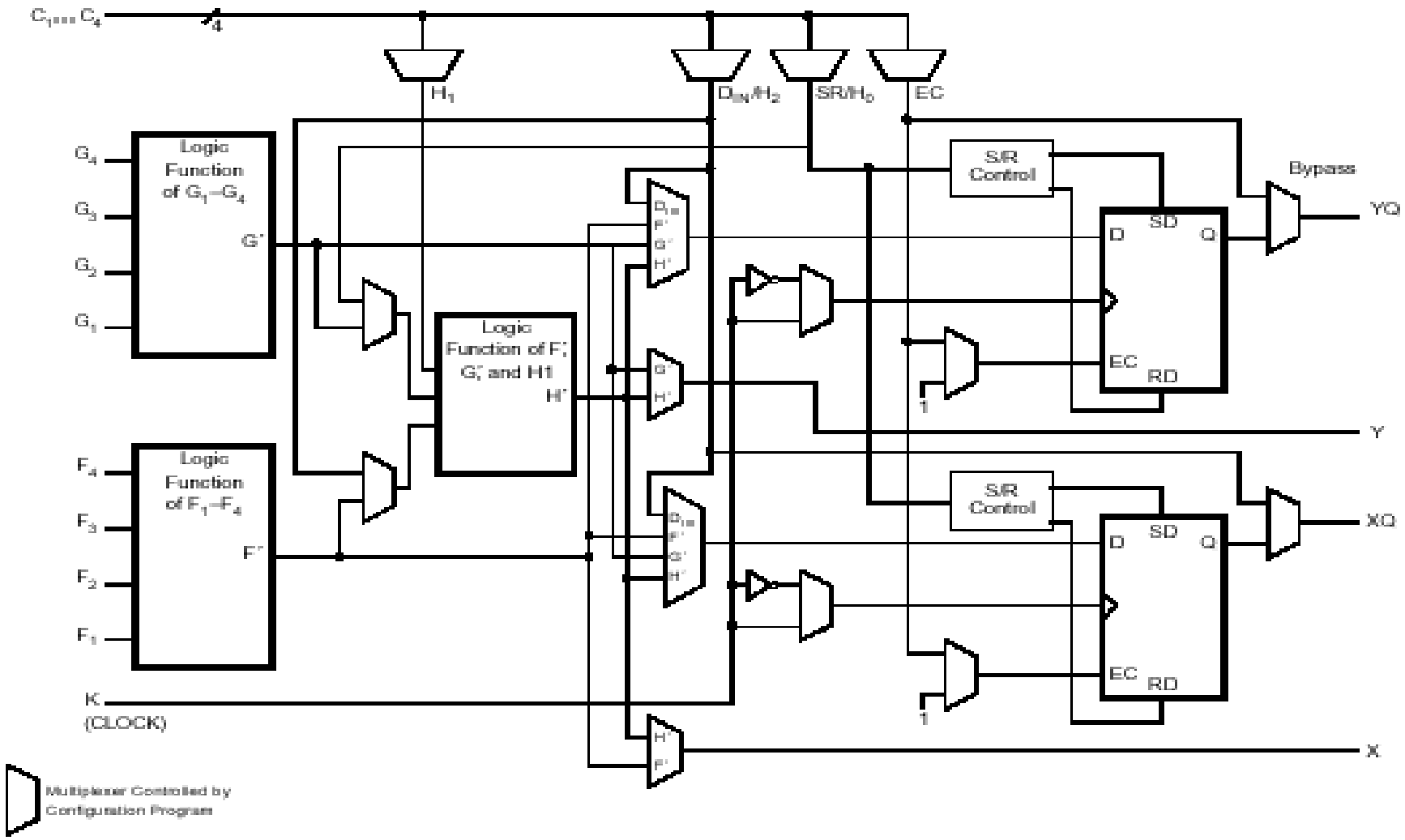
FPGA structuren

Field Programmable Gate Arrays: Principles



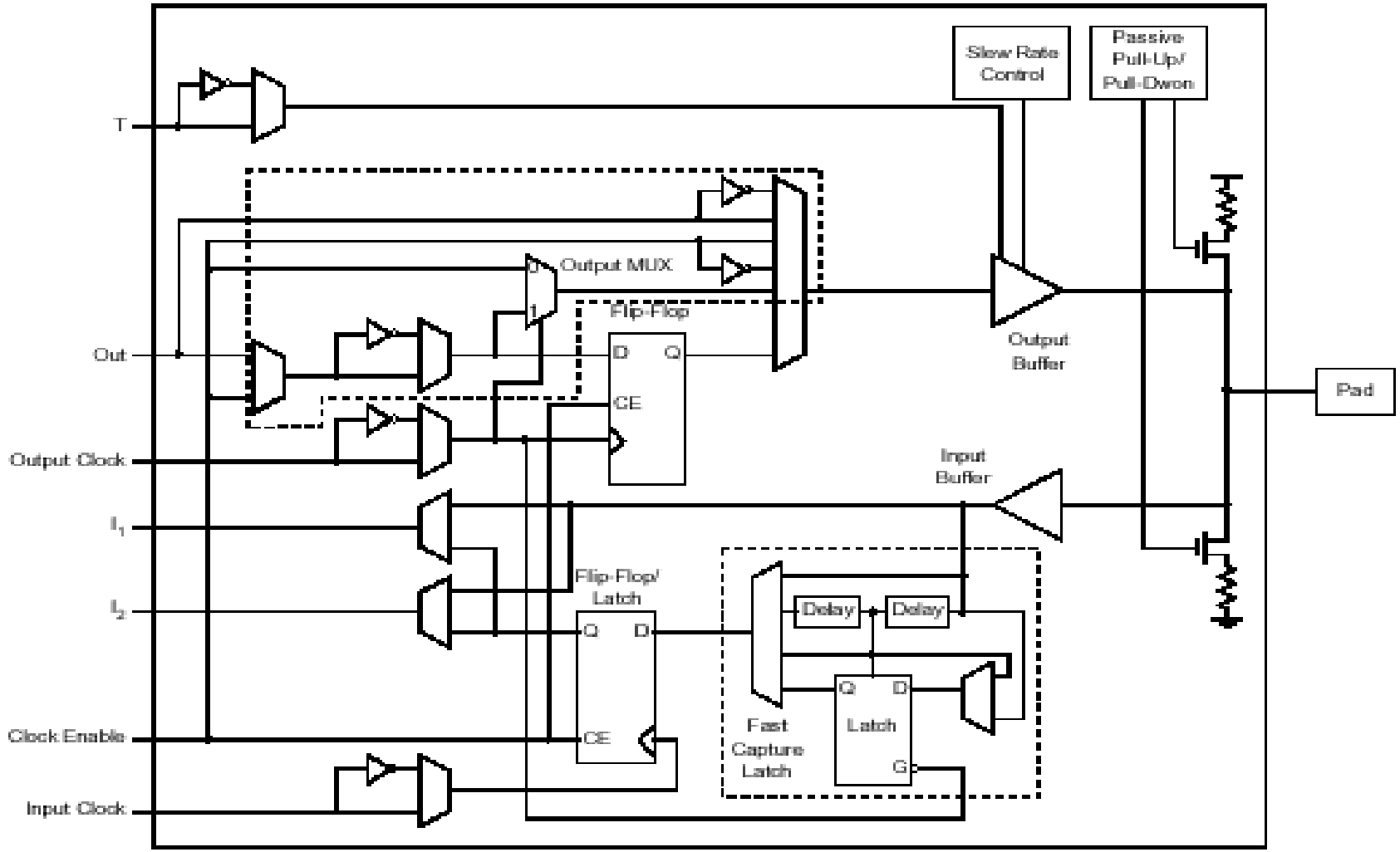
- The chip is tiled with Configurable Logic Blocks (CLBs)
  - each block can “implement” a few gates and flip flops
- A switching matrix is interleaved with this array of CLBs
- Usage: partition, place, and route a design

Inside a Configurable Logic Block (CLB)



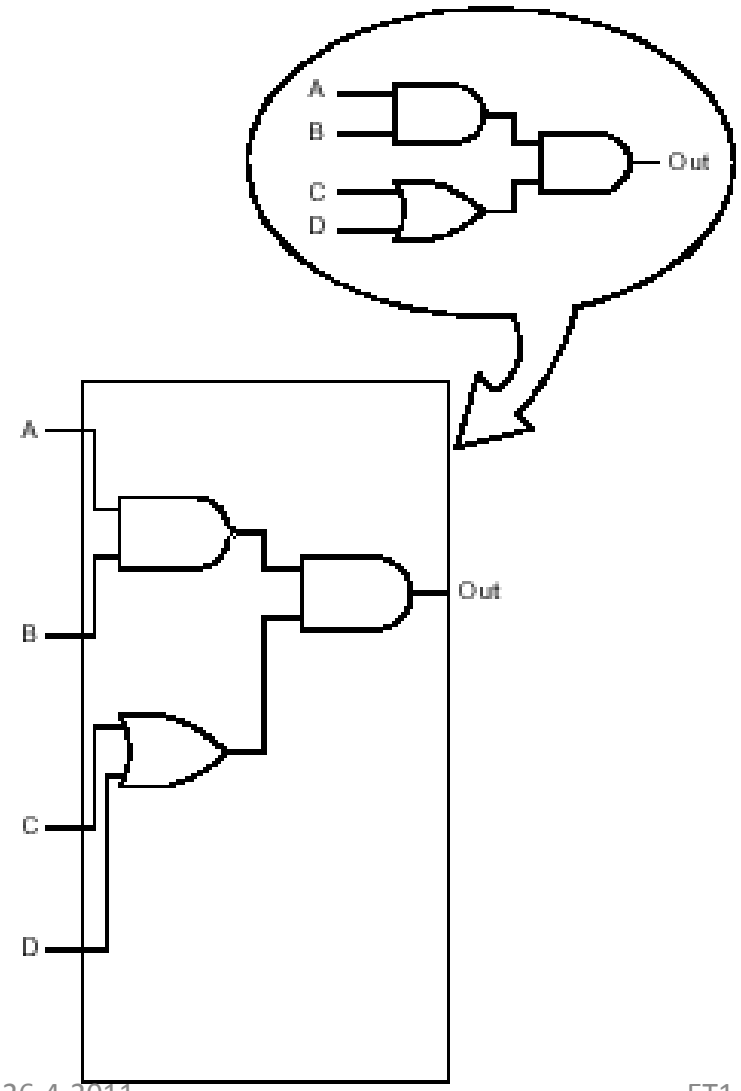
- Combinational and sequential components

Inside a I/O Block (IOB)



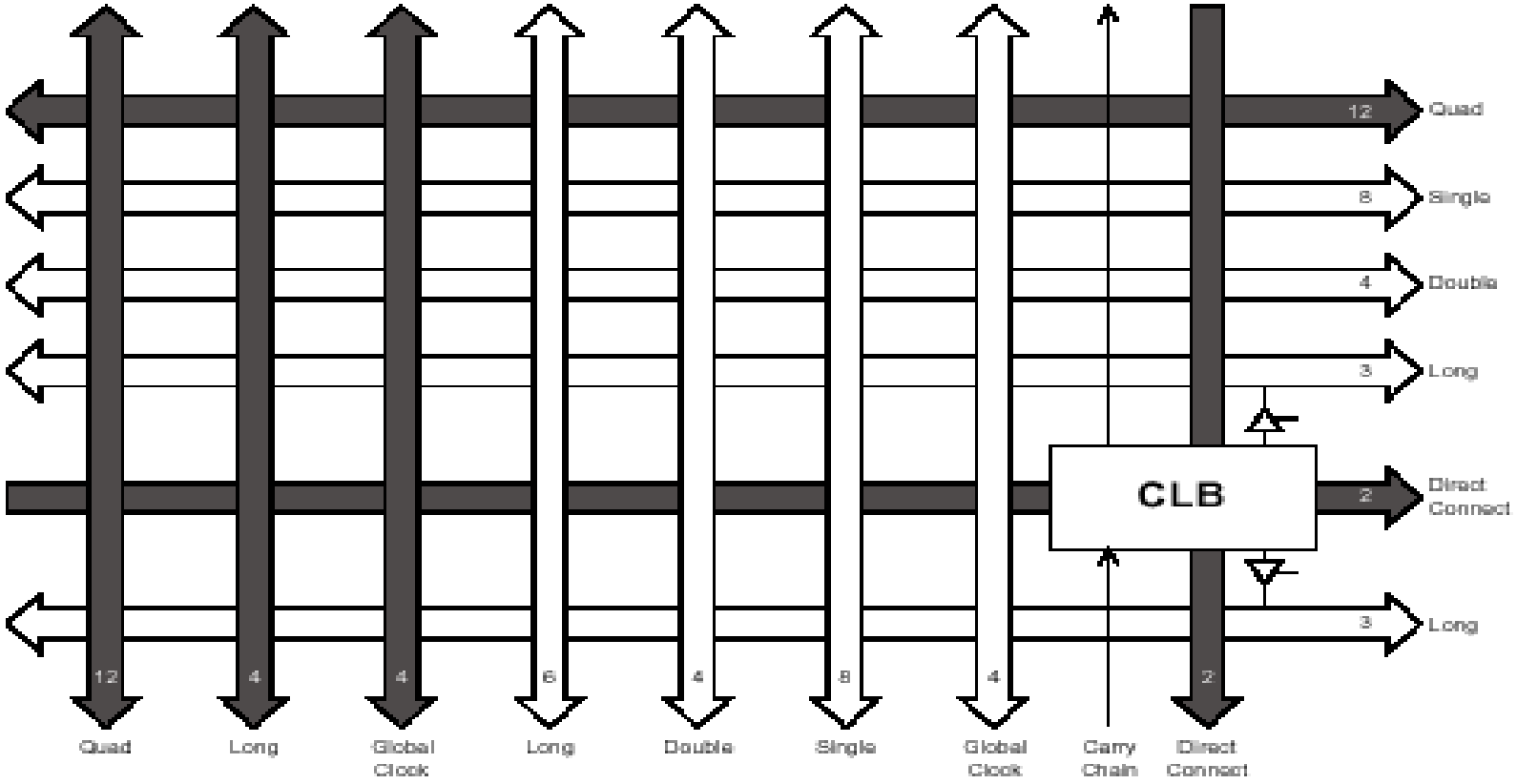
- Pin multiplexing and distinct clocks

Implementing Combinational Logic in a CLB



- Boolean functions implemented as look-up tables
- Combinations of lookup tables for multivariable functions
- Implementation of behavior
- Sequential circuits use CLB latches/flip flops

Wiring Resources



- Optimize signal delay
- “Chip length” signals can be configured as buses
- Global nets for low skew clocks and reset signals

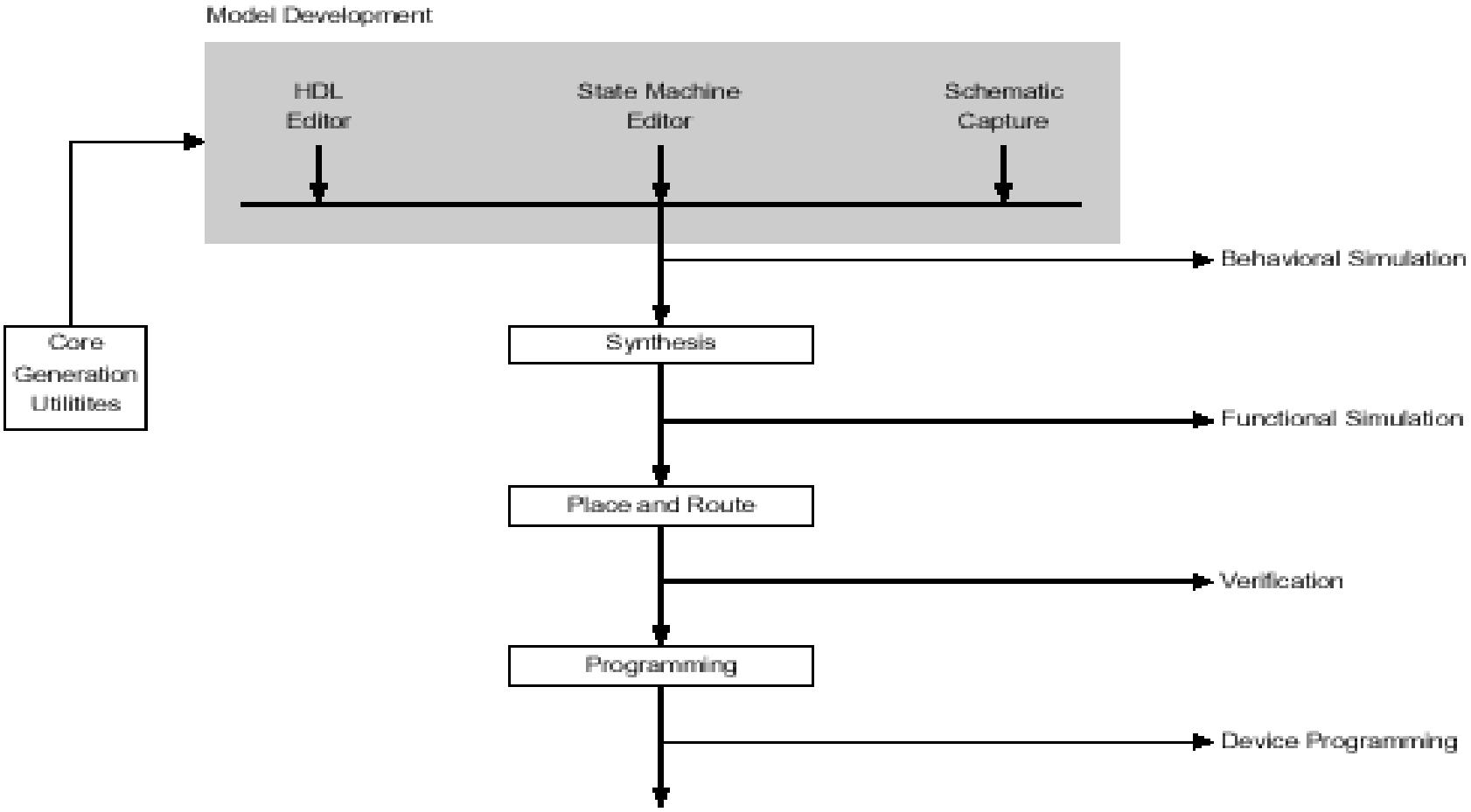
Specialized Resources

- Carry chains between columns of CLBs
- Configuration of CLB RAM as memories rather than lookup tables
- Core generators
    - optimized libraries of components
    - vendor supplied
- What about editing the bit stream directly!

Chip Configuration (Xilinx)

- Configuration bits for CLBs
  - bits for loading the LUTs
  - bits for configuring the flip flops
  - bits for setting multiplexors
- Configuration bits for the switch matrix
  - connecting horizontal and vertical lines
  - tristate devices within the CLBs
- Configuration bits for the IOBs
  - IO clocks
  - storage vs. direct “access” to the pin
- Equivalent concepts for all vendors

Design Flow



- Design flow is a function of the target implementation, for example, FPGAs