VHDL

An introduction to modeling, analyzing, and simulating digital circuits

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

- P. Ashenden: The Designer's Guide to VHDL, Morgan Kaufmann Publishers, 1996.
- K. Chang: Digital Design and Modeling with VHDL and Synthesis, IEEE Computer Society Press, 1997.
- S. Yalamanchili: VHDL Starter's Guide, Prentice Hall, 1998.
- C.H. Roth: Digital System Design using VHDL, PWS Publishing Company, 1998.

Outline

UNI FREIBURG

- VHDL
  - Very High Speed IC Hardware Description Language
- Main goal was developing a hardware description language to model digital circuits efficiently
  - Technology-independent
  - Various levels of abstraction
  - Behavioral, structural, or mixed descriptions
  - Structural hierarchy to model large systems
  - Modeling concurrent systems
  - Re-usability of specifications
  - Validation of designs based on the same description language for different levels of abstraction using a single simulation tool

- Started as an US Department of Defense project in 1983
- First IEEE standard was in 1987 (VHDL-87)
- Revised in 1993 (VHDL-93)
- Development still in progress (VHDL-2000/2002/2008)
- Nowadays, ...
  - VHDL is the standard for most of the US and EU industry
  - Mainly used for hardware generation (not all VHDL statements are synthesizable!)

Outline

Typically, VHDL is used for the following tasks

1. Modeling the behavior of a digital system
2. Validating the developed design
3. Translating the digital system into real hardware (using appropriate synthesis tools)
   - ASICs – Application-Specific Integrated Circuits
   - FPGAs – Field Programmable Gate Arrays

ASICs – Application-Specific Integrated Circuits

- Customized for a particular use
- Functionality cannot be changed after manufactoring

FPGAs – Field Programmable Gate Arrays

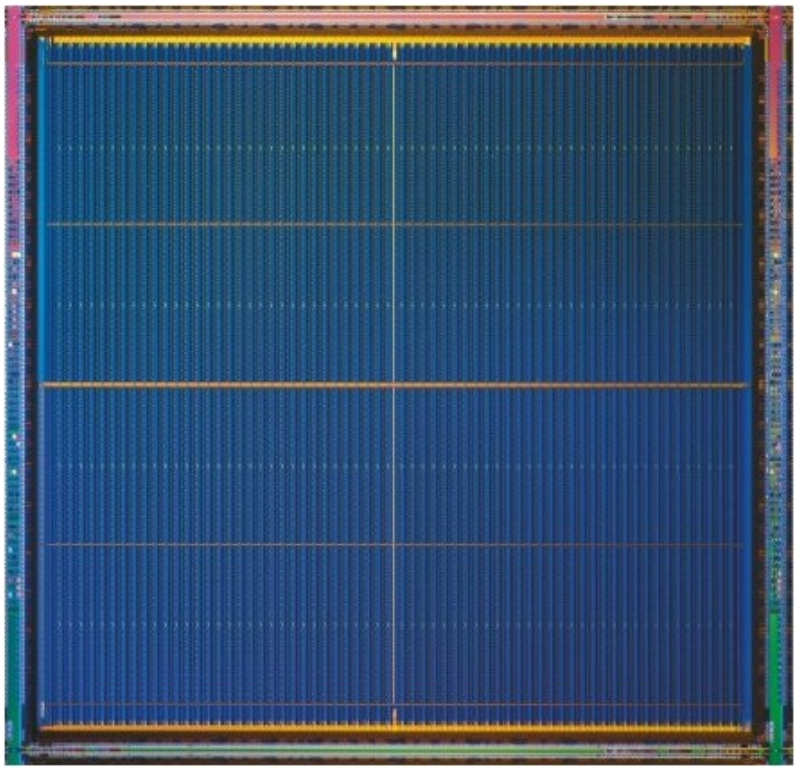
- Designed to be configured by the customers after manufactoring
- Programmable logic elements & interconnects (sometimes even embedded microprocessors)
- Can be reconfigured indefinitely often (i.e. to fix bugs)
- May be cheaper than ASICs (smaller designs / lower production volumes)
- Usually, slower and less power efficient than real hardware

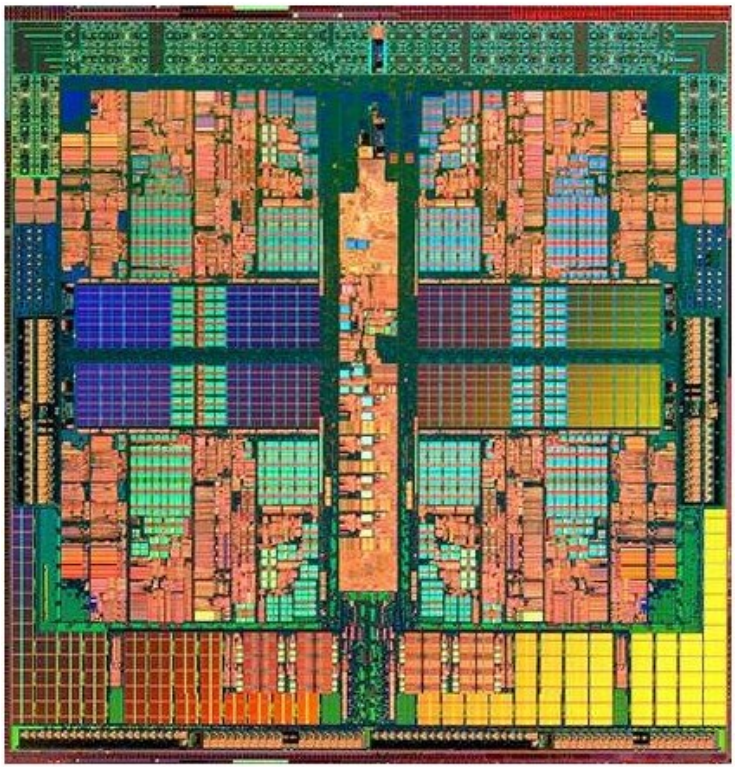
ASICs – Application-Specific Integrated Circuits

- Customized for a particular use
- Functionality cannot be changed after manufactoring

FPGAs – Field Programmable Gate Arrays

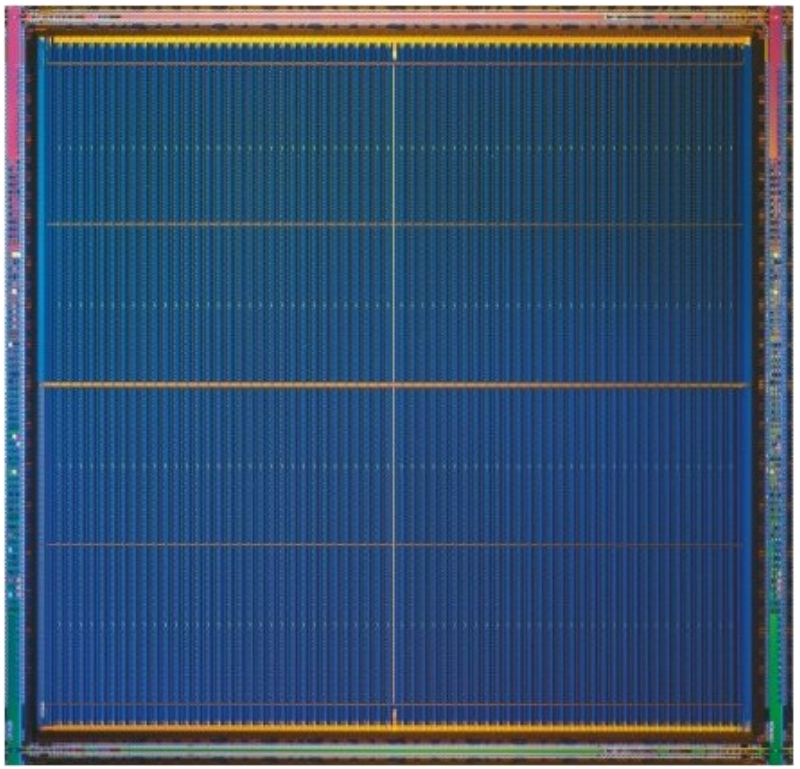
- Designed to be configured by the customers after manufactoring
- Programmable logic elements & interconnects (sometimes even embedded microprocessors)
- Can be reconfigured indefinitely often $\Rightarrow$ prototyping
- May be cheaper than ASICs (smaller designs / lower production volumes)
- Usually, slower and less power efficient than real hardware



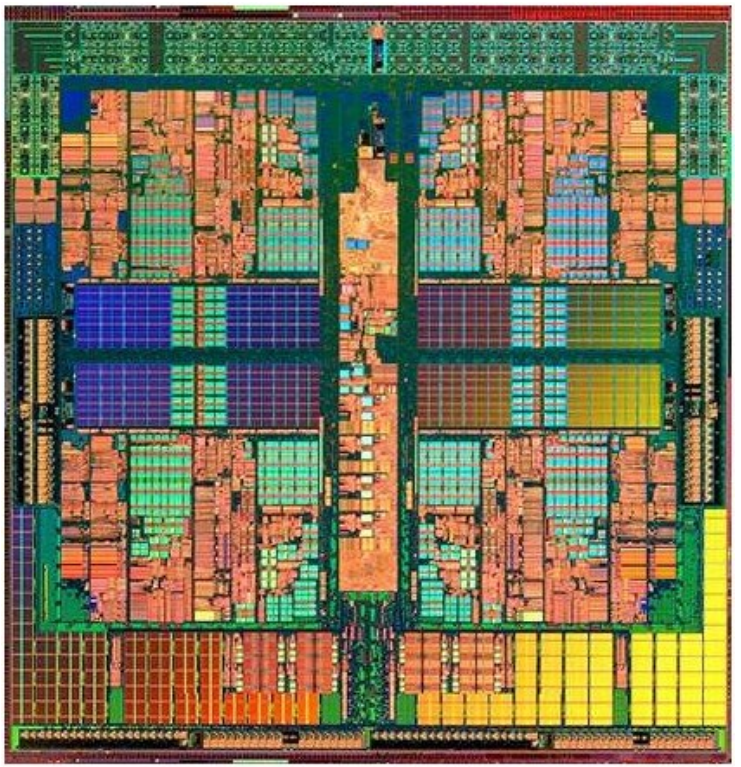


Which one is the FPGA?

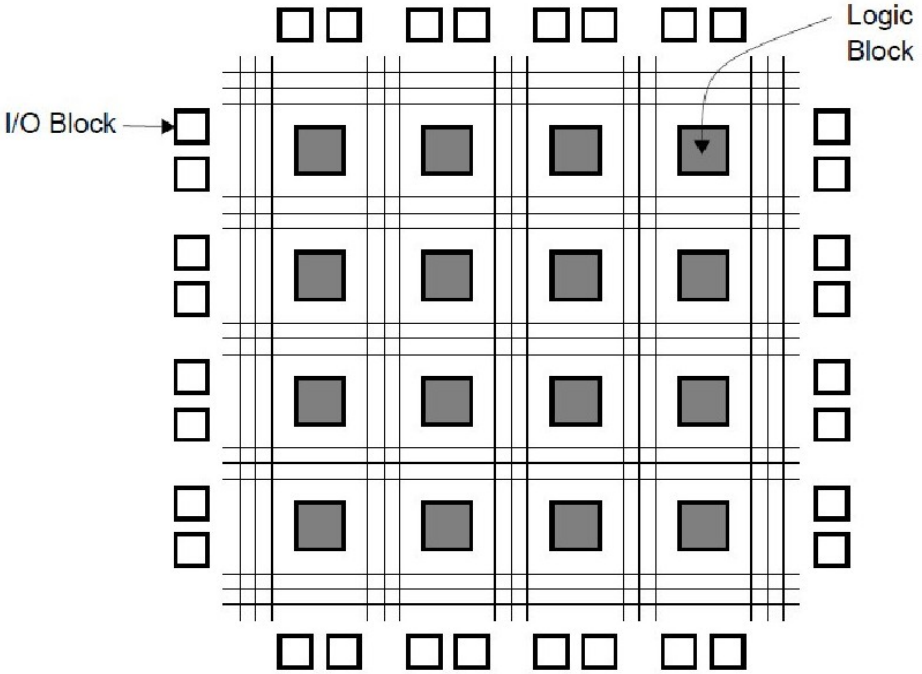
FPGAs – Field Programmable Gate Arrays

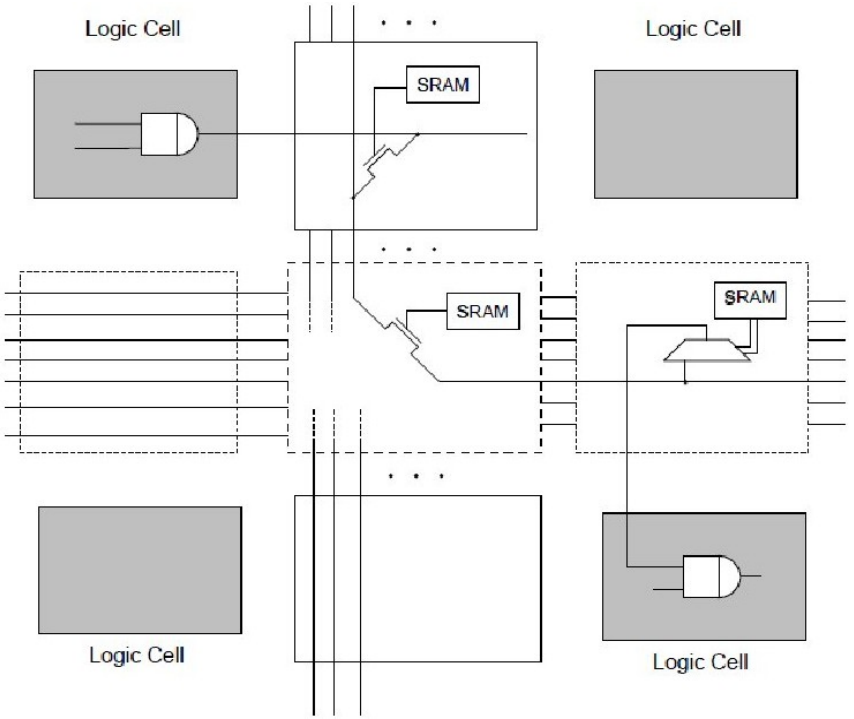


Xilinx Virtex FPGA

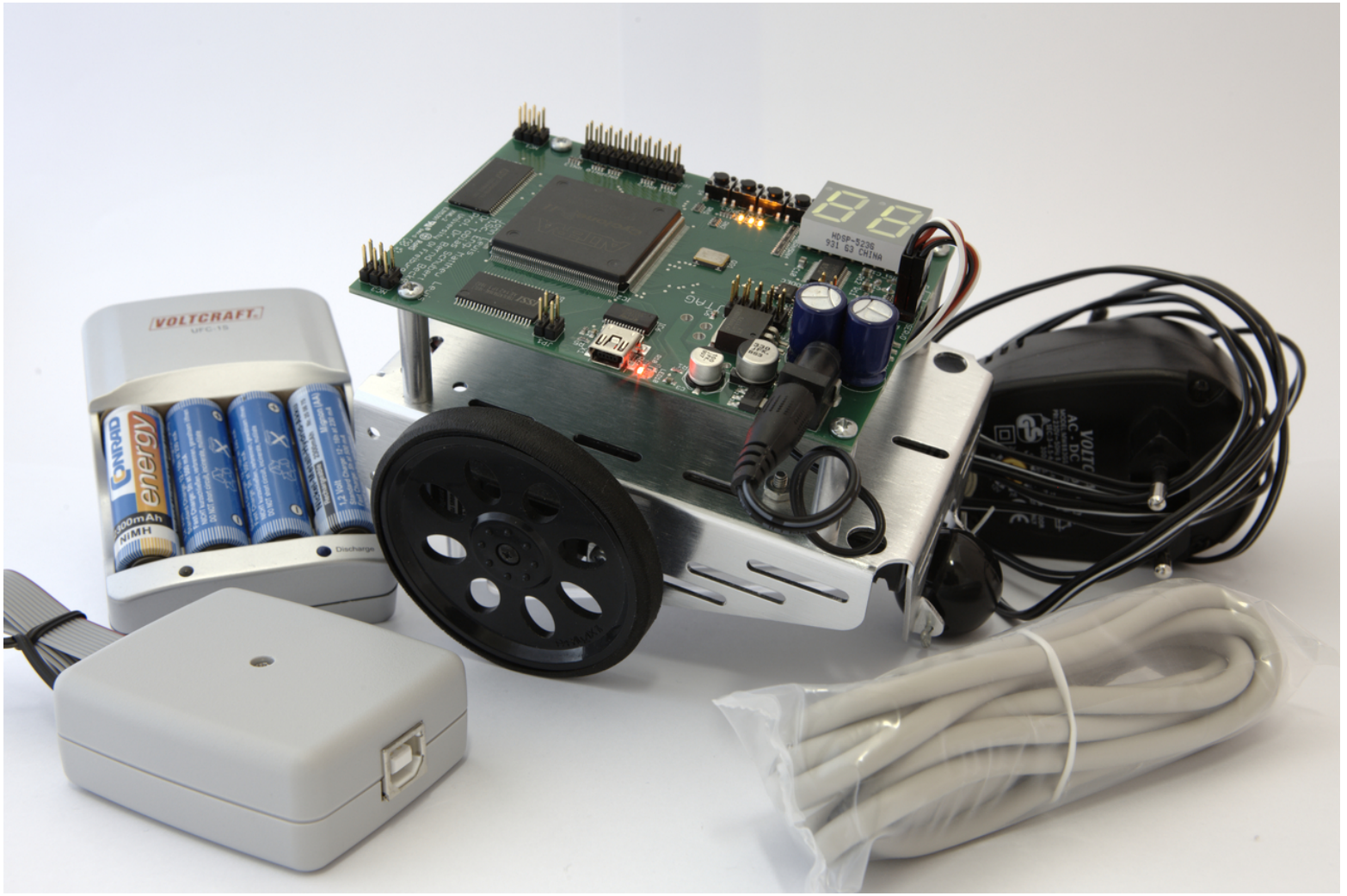


AMD Phenom CPU


Logic
Block
I/O Block

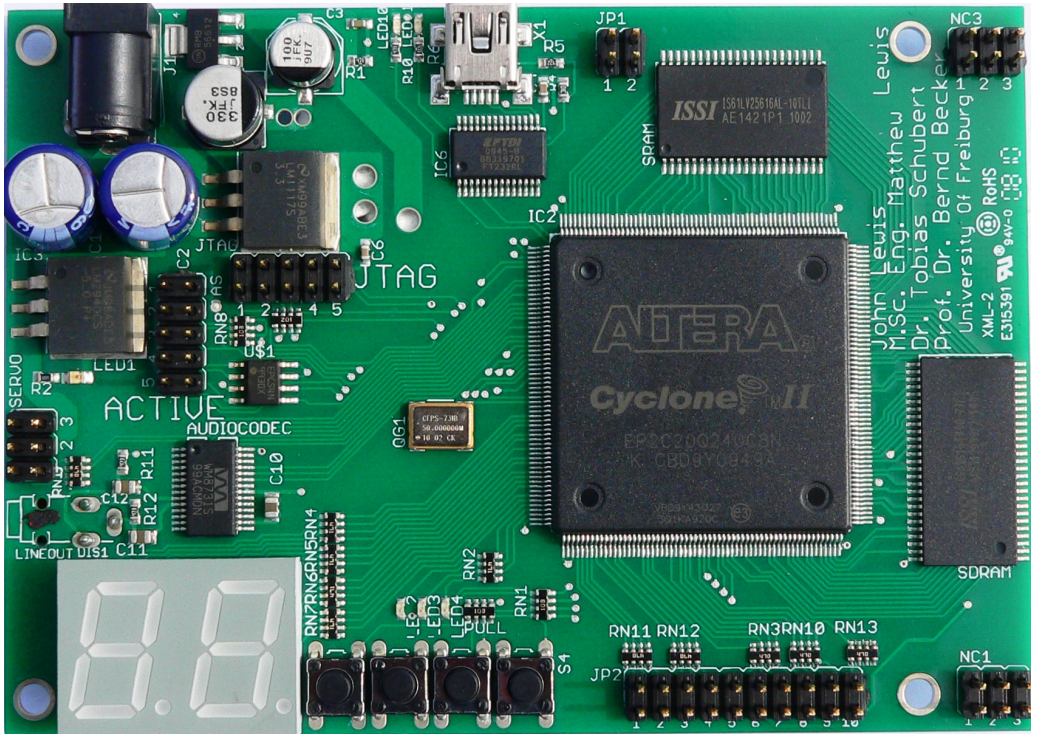

Logic Cell
Logic Cell
SRAM
SRAM
SRAM
Logic Cell
Logic Cell

FPGA Boards used in Freiburg

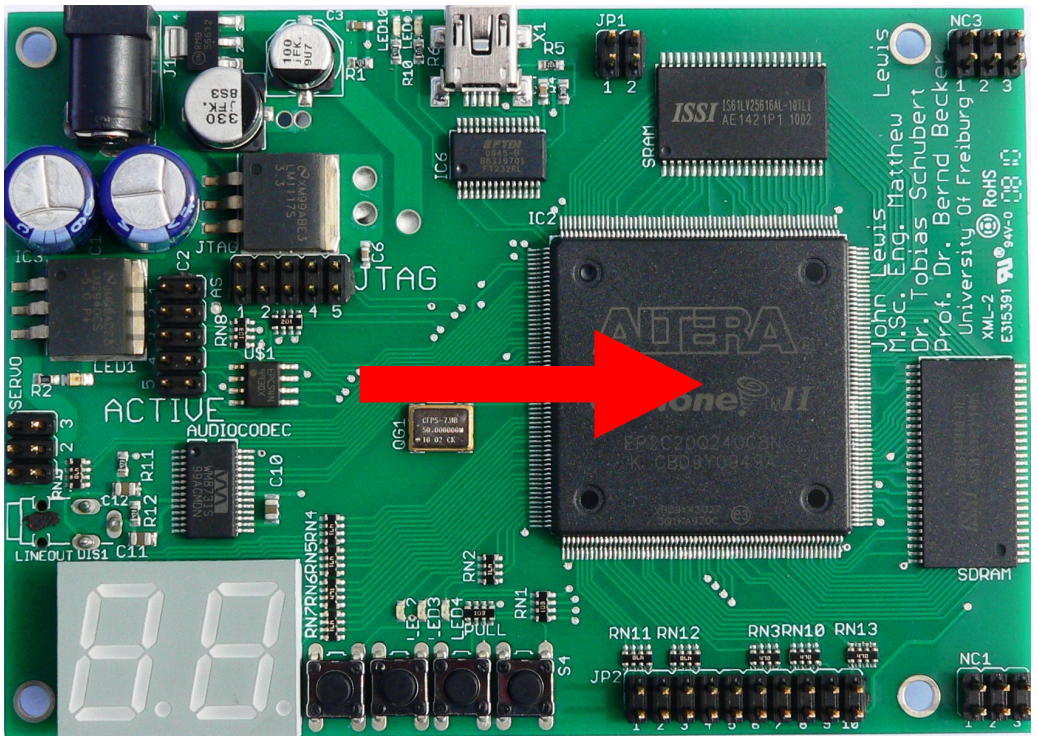


FPGA Boards used in Freiburg

Albert-Ludwigs-Universität Freiburg



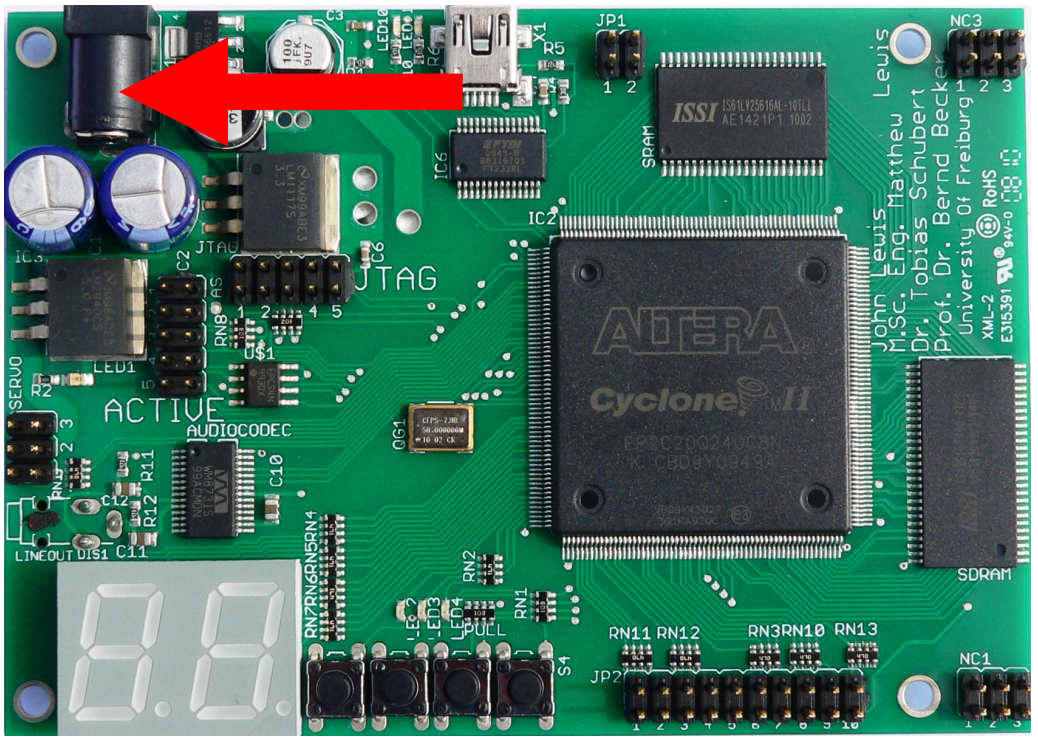
FPGA Boards used in Freiburg



Altera Cyclone II FPGA

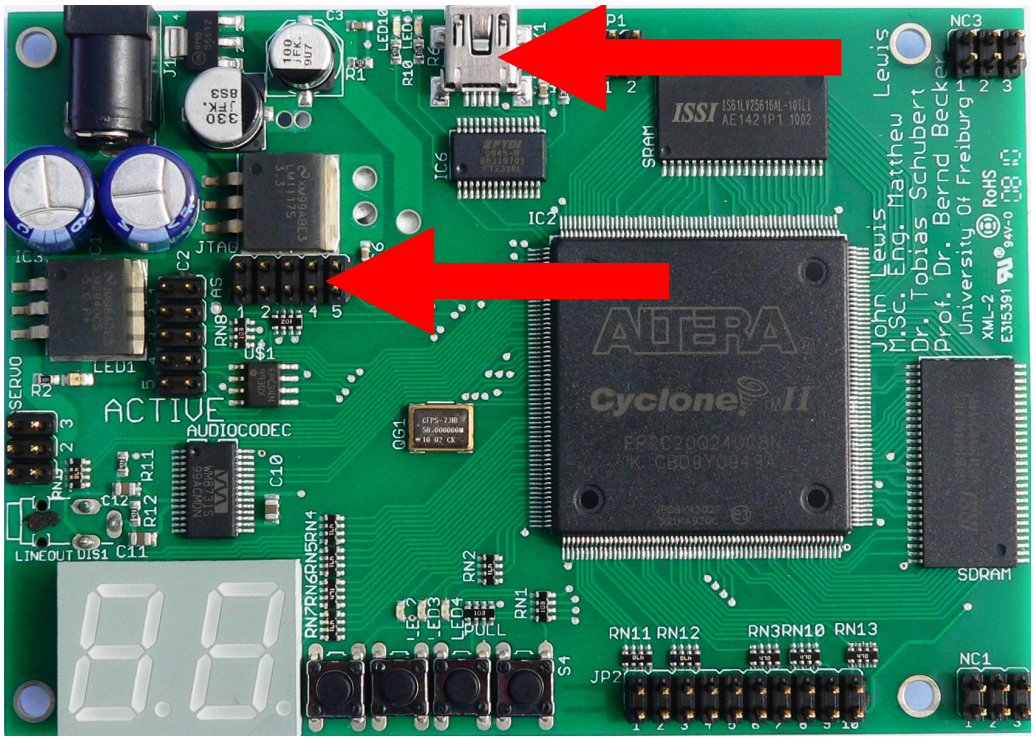
- About 240,000 bits of RAM
- About 18,000 logic blocks
- Supports softcore processors

FPGA Boards used in Freiburg



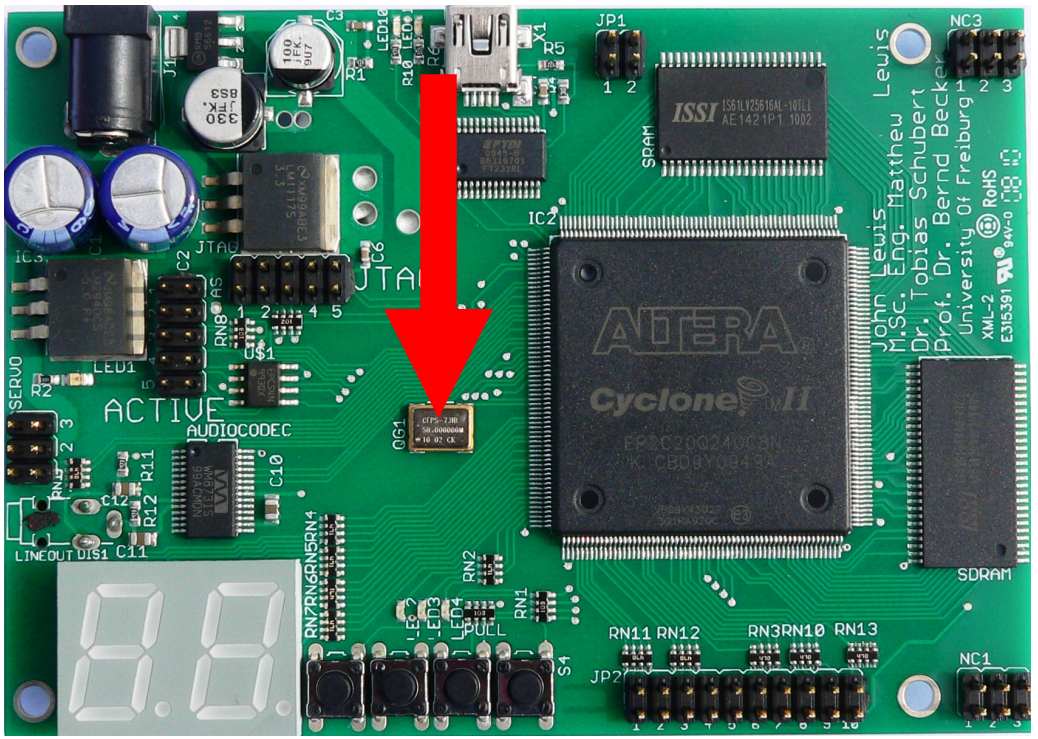
Power supply

FPGA Boards used in Freiburg



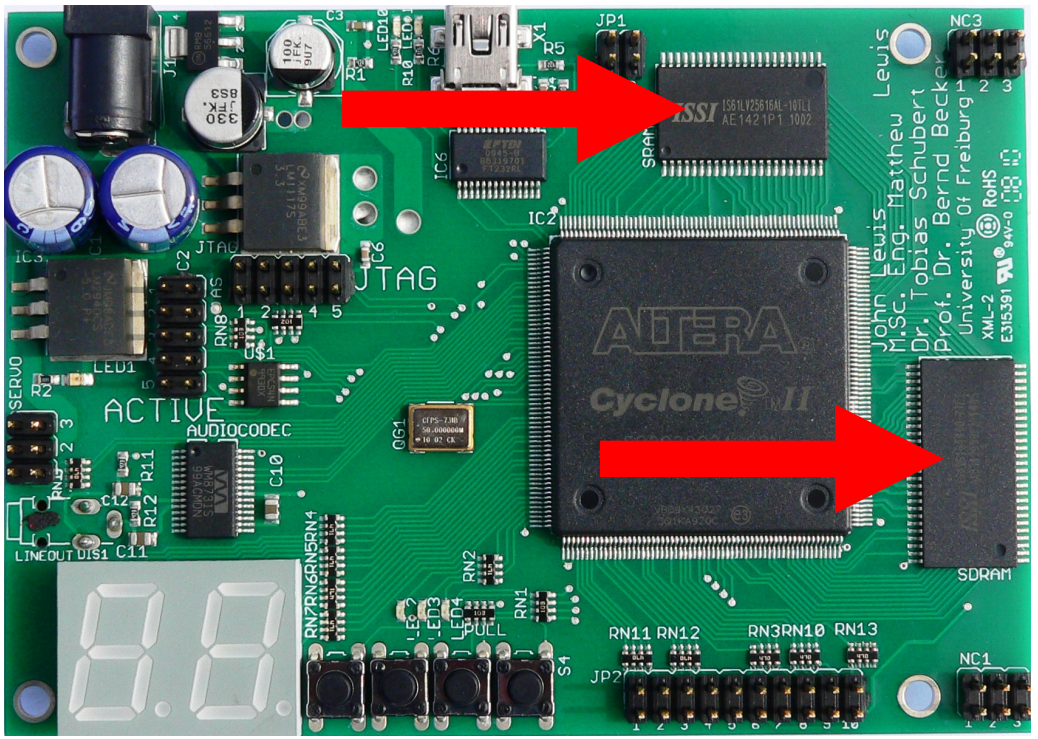
USB & JTAG interface

FPGA Boards used in Freiburg



50 MHz quartz

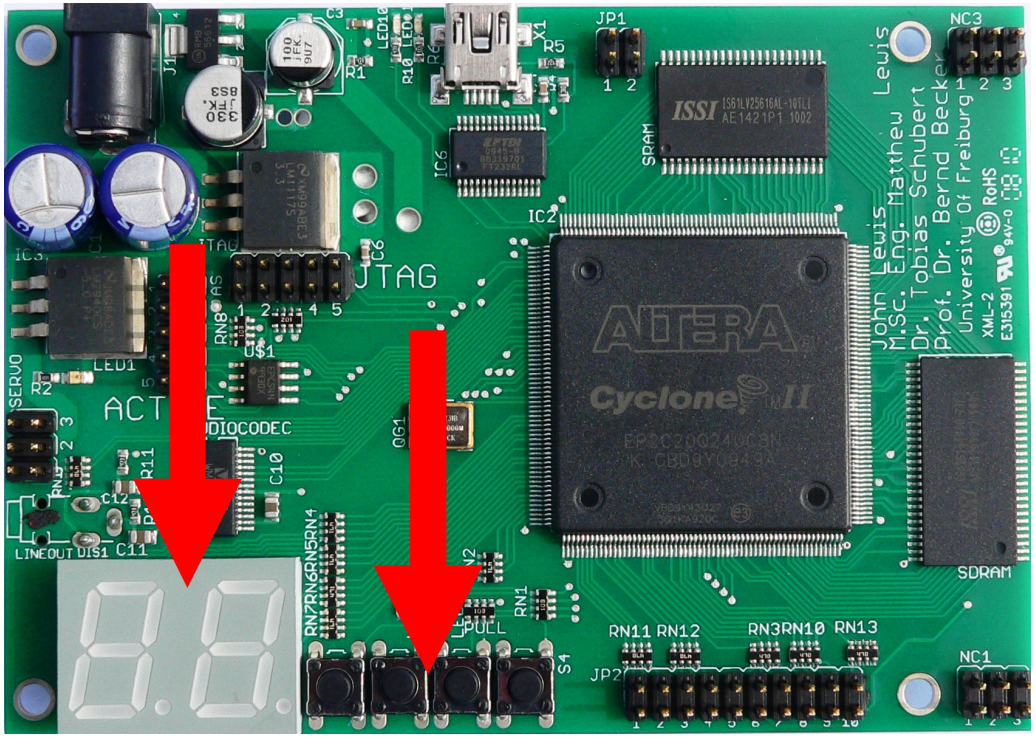
FPGA Boards used in Freiburg



512 kB SRAM, 8 MB SDRAM

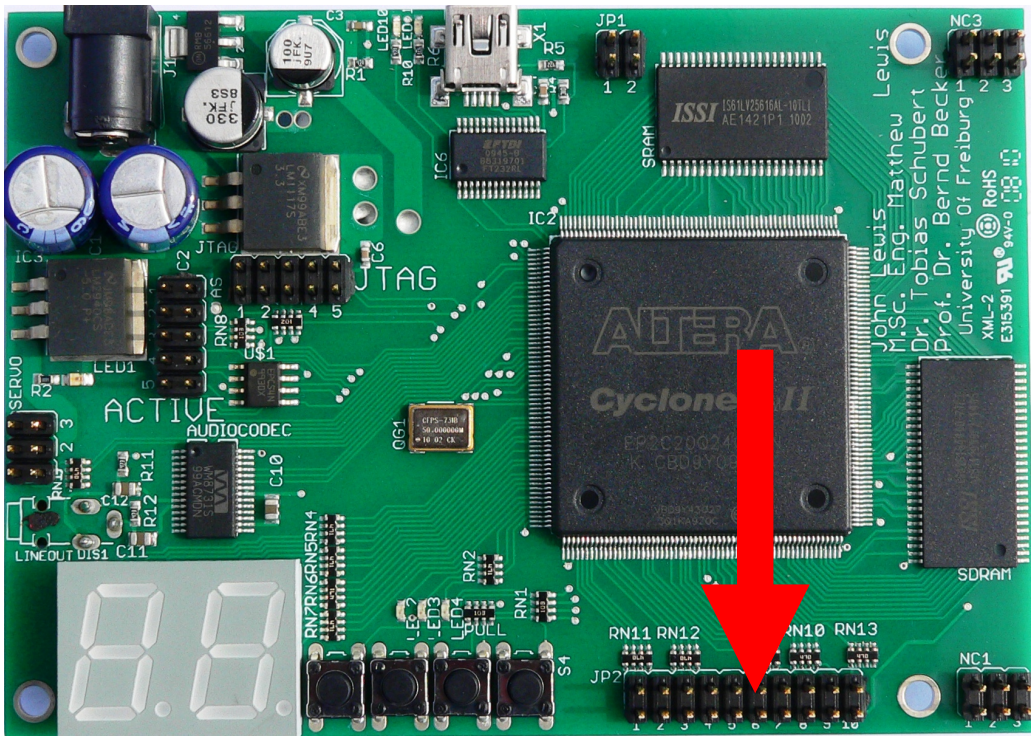
FPGA Boards used in Freiburg

Albert-Ludwigs-Universität Freiburg



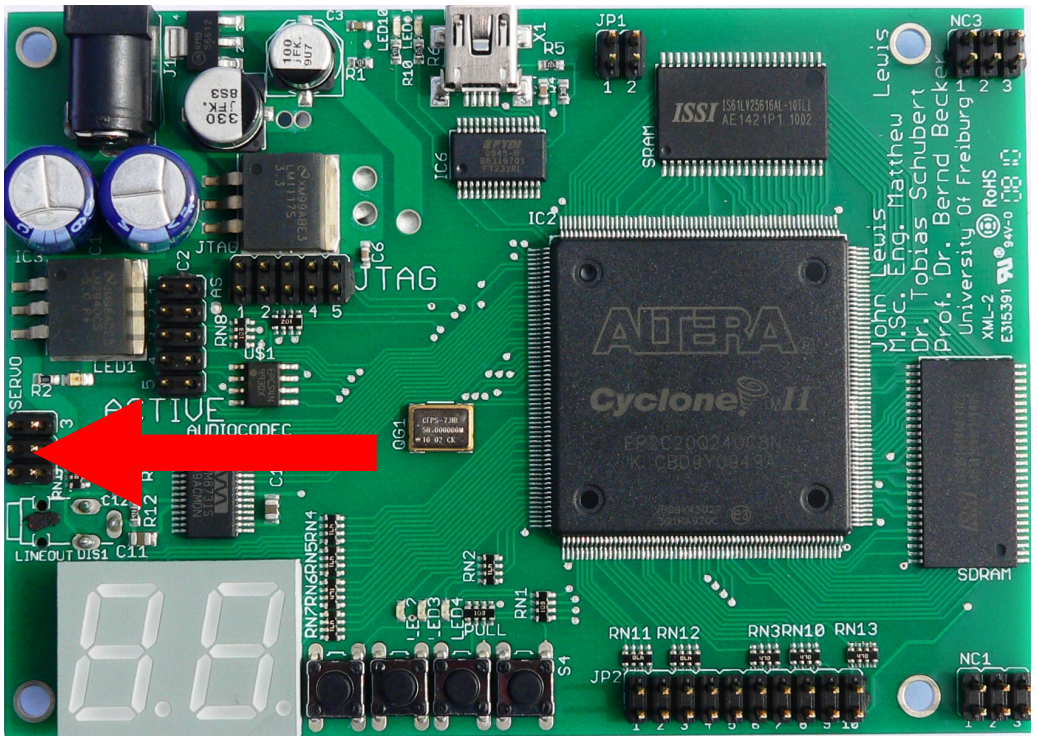
Push buttons, LEDs, display

FPGA Boards used in Freiburg



General purpose I/O pins

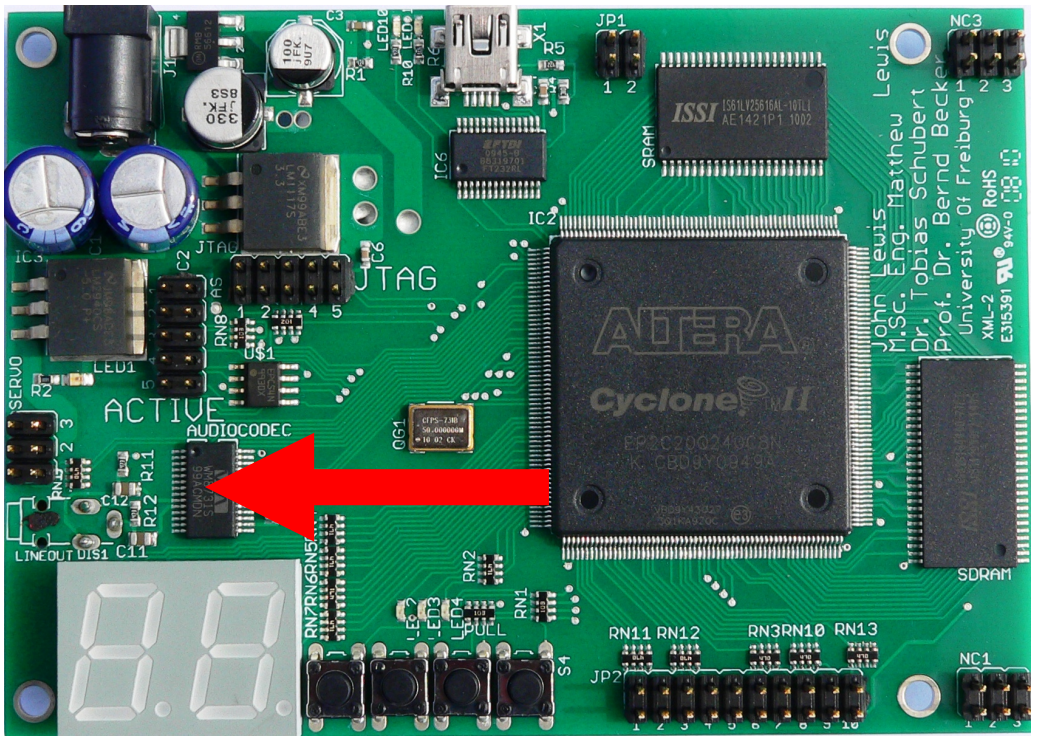
FPGA Boards used in Freiburg



Servo control pins

FPGA Boards used in Freiburg

Albert-Ludwigs-Universität Freiburg



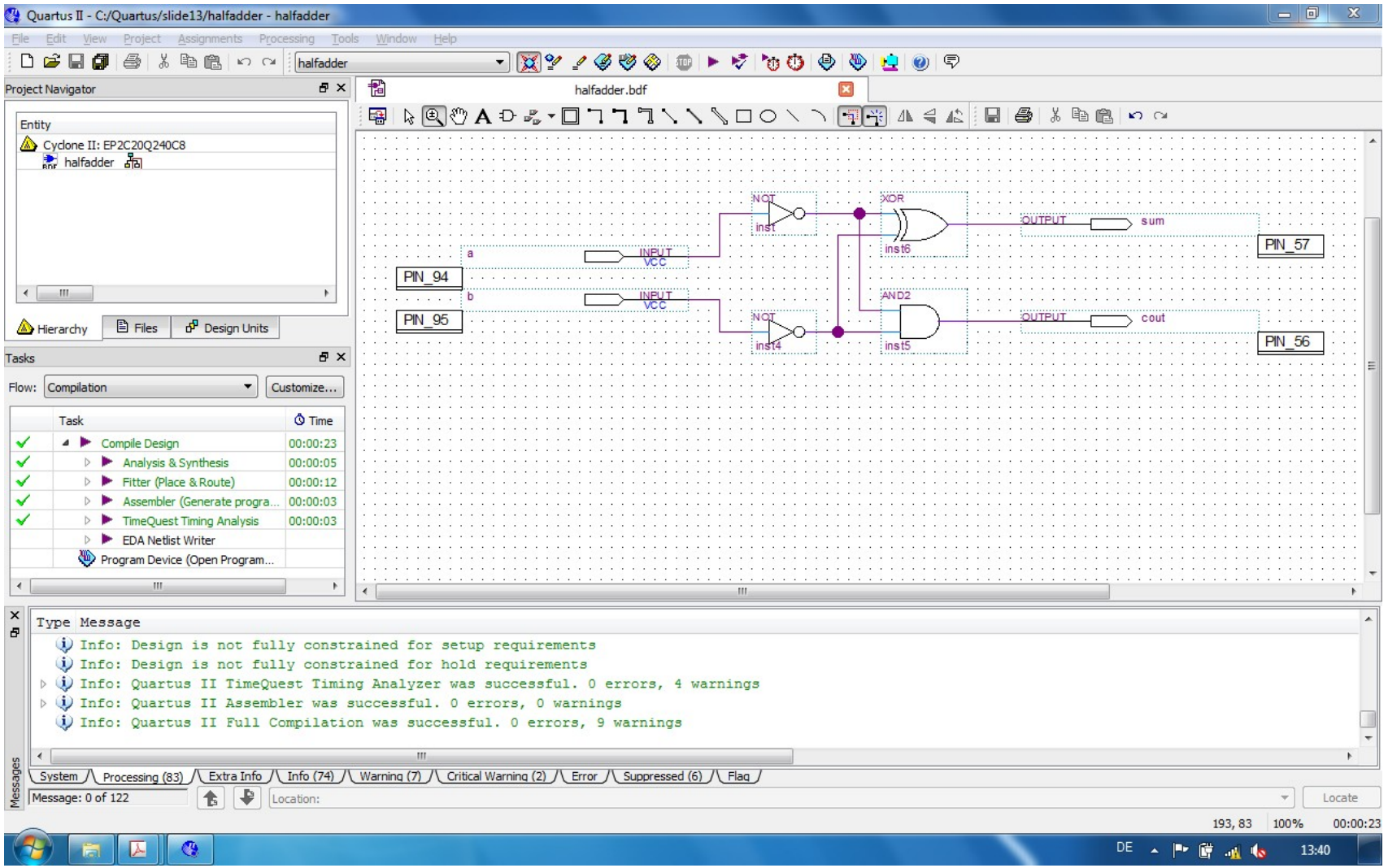
Audio chip

Altera Software used in Freiburg

- Quartus II Web Edition Version 10.1
- Modelsim Altera Starter Edition Version 6.6d
- Nios II Embedded Design Suite Version 10.1

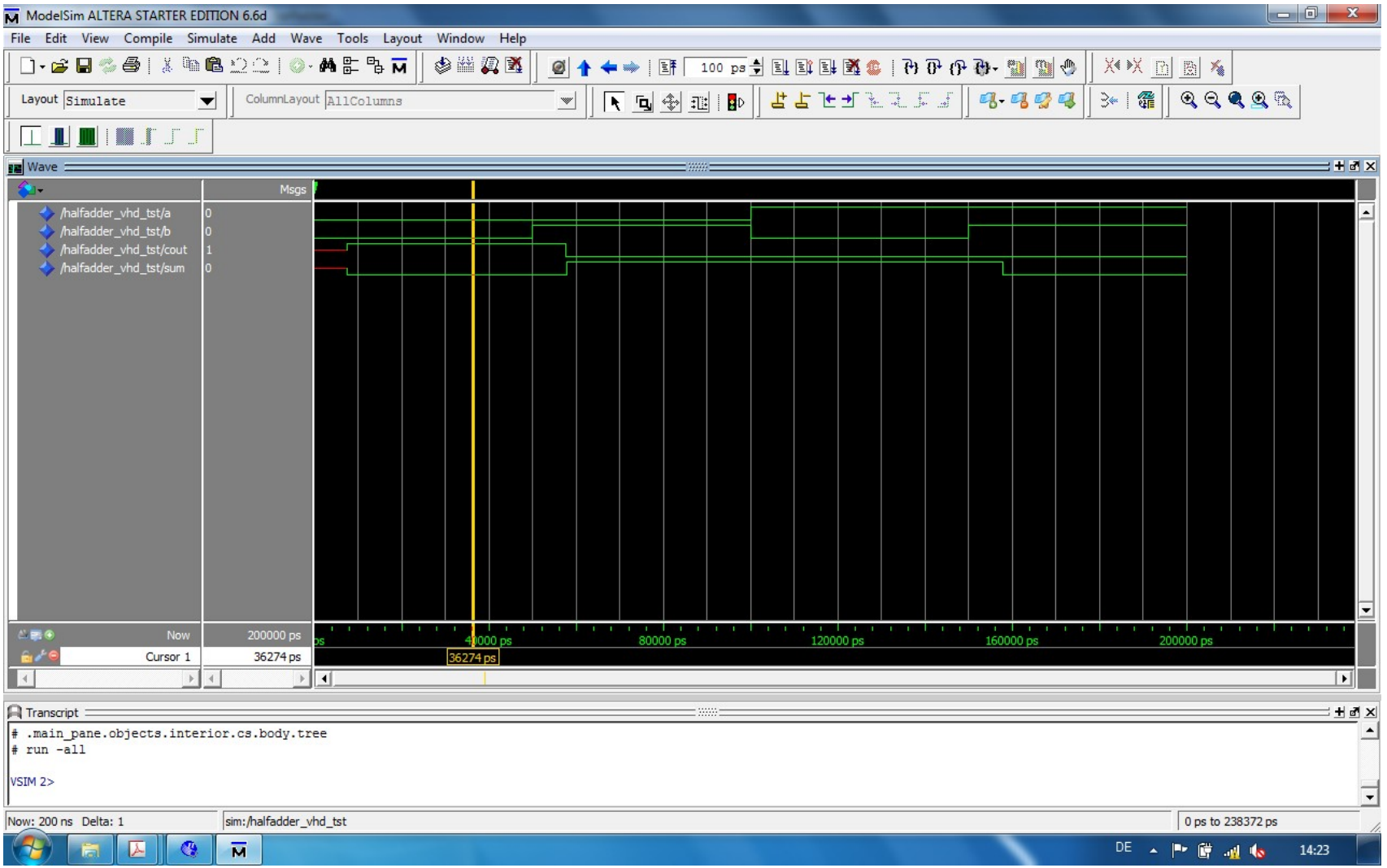
Quartus II Web Edition

Albert-Ludwigs-Universität Freiburg



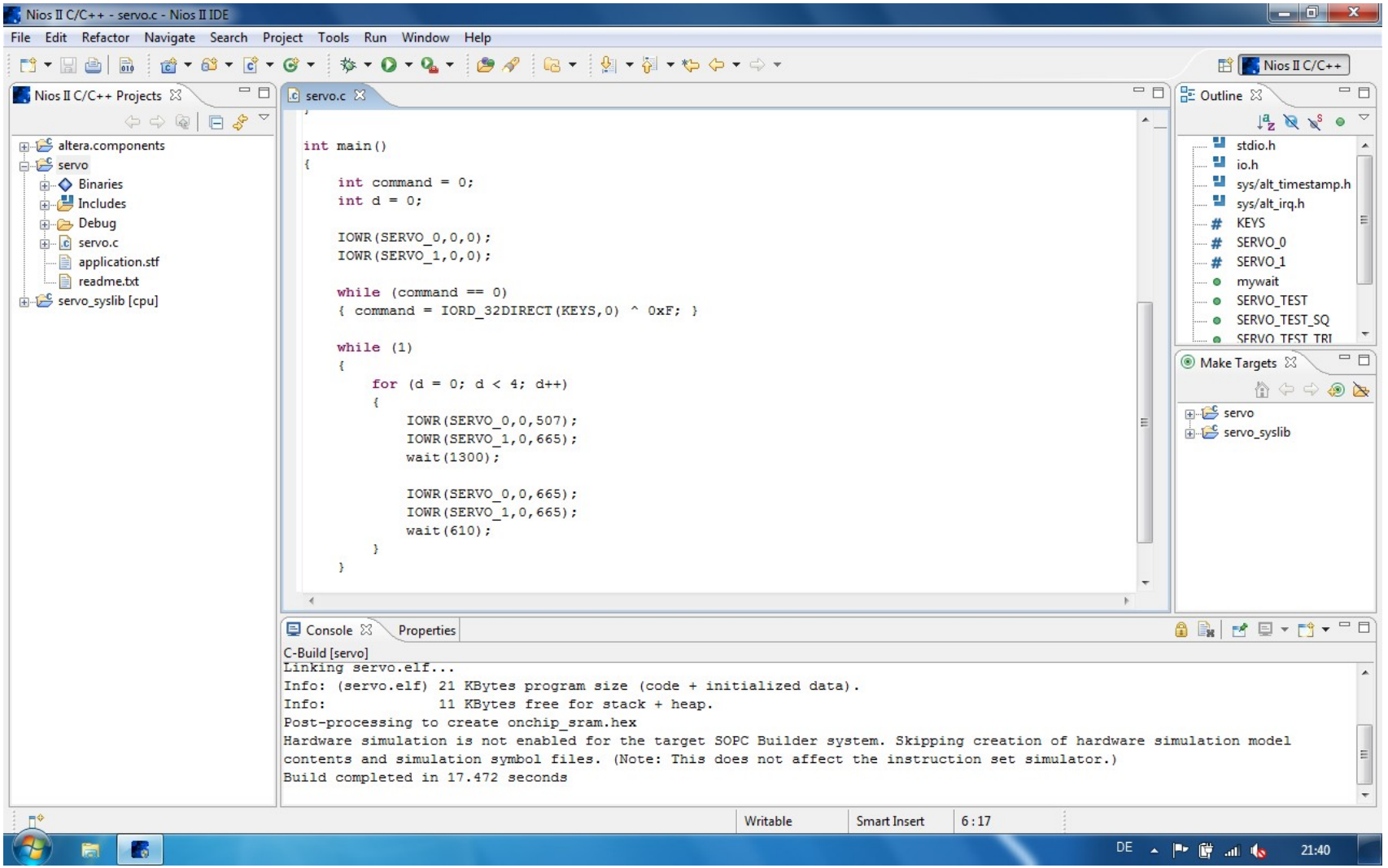
Modelsim Altera Starter Edition

Albert-Ludwigs-Universität Freiburg



Nios II Embedded Design Suite

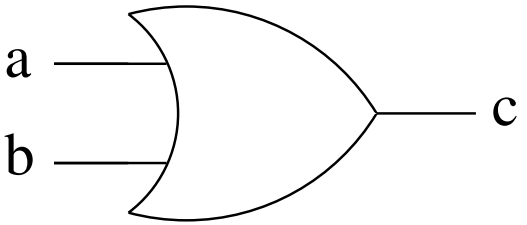
Albert-Ludwigs-Universität Freiburg



Outline

Entity

```
entity orGate is
   port ( a, b: in bit; c: out bit );
end orGate;
```



UNI FREIBURG

Entity

```
entity orGate is
   port ( a, b: in bit; c: out bit );
end orGate;
```

Entities

- Define the interface
- Entity name = file name
- Ports
  - Inputs ⇒ **in**
  - Outputs ⇒ **out**
  - Bi-directional ⇒ **inout**
  - ...
- Remarks
  - Outputs cannot be read
  - Inputs cannot be written to

Examples – OR Gate

Entity

```
entity orGate is
   port ( a, b: in bit; c: out bit );
end orGate;
```

Architecture(s)

```
architecture arch1 of orGate is
begin
   c <= a or b;
end arch1;
```

```
architecture arch2 of orGate is
begin
   c <= '1' when (a = '1' or b = '1') else '0';
end arch2;
```

Architectures define the implementation of an entity

Entity

```
entity fullAdder is
   port ( a, b, cin: in bit; sum, cout: out bit );
end fullAdder;
```

Examples – Full Adder (Behavioral Description)

Entity

```
entity fullAdder is
   port ( a, b, cin: in bit; sum, cout: out bit );
end fullAdder;
```

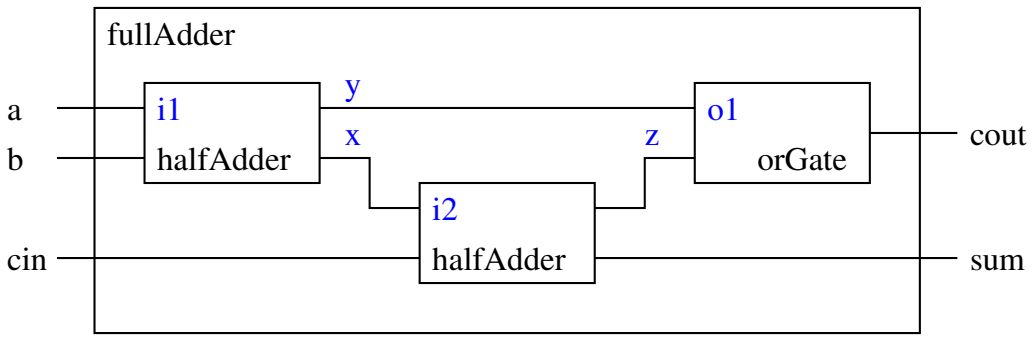
Architecture

```
architecture behavior of fullAdder is
begin
   sum <= (a xor b) xor cin;
   cout <= (a and b) or (a and cin) or (b and cin);
end behavior;
```

Examples – Full Adder (Structural Description)

Entity

```
entity fullAdder is
   port ( a, b, cin: in bit;
   sum, cout: out bit );
end fullAdder;
```



Architecture

```
architecture structure of fullAdder is
   component halfAdder port ( a, b: in bit; sum, cout: out bit ); end component;
   component orGate port ( a, b: in bit; c: out bit ); end component;
   signal x, y, z: bit;
begin
   i1: halfAdder port map ( a, b, x, y );
   i2: halfAdder port map ( x, cin, sum, z );
   o1: orGate port map ( y, z, cout );
end structure;
```

Examples – Full Adder (Structural Description)

UNI FREIBURG

halfAdder

```
entity halfAdder is
   port ( a, b: in bit; sum, cout: out bit );
end halfAdder;

architecture behavior of halfAdder is
begin
   sum <= a xor b;
   cout <= a and b;
end behavior;
```

orGate

```
entity orGate is
   port ( a, b: in bit; c: out bit );
end orGate;

architecture behavior of orGate is
begin
   c <= '1' when (a = '1' or b = '1') else '0';
end behavior;
```

Examples – Counter (using signals)

Entity & Architecture

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

entity counter is
   port (
      clk: in std_logic;
      q1, q2: out std_logic_vector (4 downto 0) );
end counter;

architecture behavior of counter is
   signal cnt: std_logic_vector (4 downto 0) := "00000";
begin
   process ( clk )
   begin
      if clk = '1' then cnt <= cnt + 1; end if;
      q1 <= cnt;
   end process;
   q2 <= cnt;
end behavior;
```

Processes

- Are declared within an architecture and are concurrent statements
- But, all statements inside a process are executed sequentially!
- Signal updates within a process are not done immediately!
- Processes will be executed whenever there is an event on one of the signals of the corresponding "sensitivity list"

$\Rightarrow$ In our example, the process counts the number of rising edges that occur on signal clk

Examples – Counter (using signals)

Entity & Architecture

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

entity counter is
   port (
      clk: in std_logic;
      q1, q2: out std_logic_vector (4 downto 0) );
end counter;

architecture behavior of counter is
   signal cnt: std_logic_vector (4 downto 0) := "00000";
begin
   process ( clk )
   begin
      if clk = '1' then cnt <= cnt + 1; end if;
      q1 <= cnt;
   end process;
   q2 <= cnt;
end behavior;
```

Values provided by std_logic

- '0' – logical zero
- '1' – logical one
- 'U' – uninitialized
- 'X' – unknown
- 'Z' – high impedance
- 'W' – weak unknown
- 'L' – weak low
- 'H' – weak high
- '-' – don't care

Examples – Counter (using variables)

Entity & Architecture

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

entity counter is
   port (
      clk: in std_logic;
      q: out std_logic_vector (4 downto 0) );
end counter;

architecture behavior of counter is
begin
   process ( clk )
      variable cnt: std_logic_vector (4 downto 0) := "00000";
   begin
      if clk = '1' then cnt := cnt + 1; end if;
      q <= cnt;
   end process;
end behavior;
```

Variables

- Appear in processes only
- Initial values can be set for simulation purpose
- Variables are updated without any time delay!

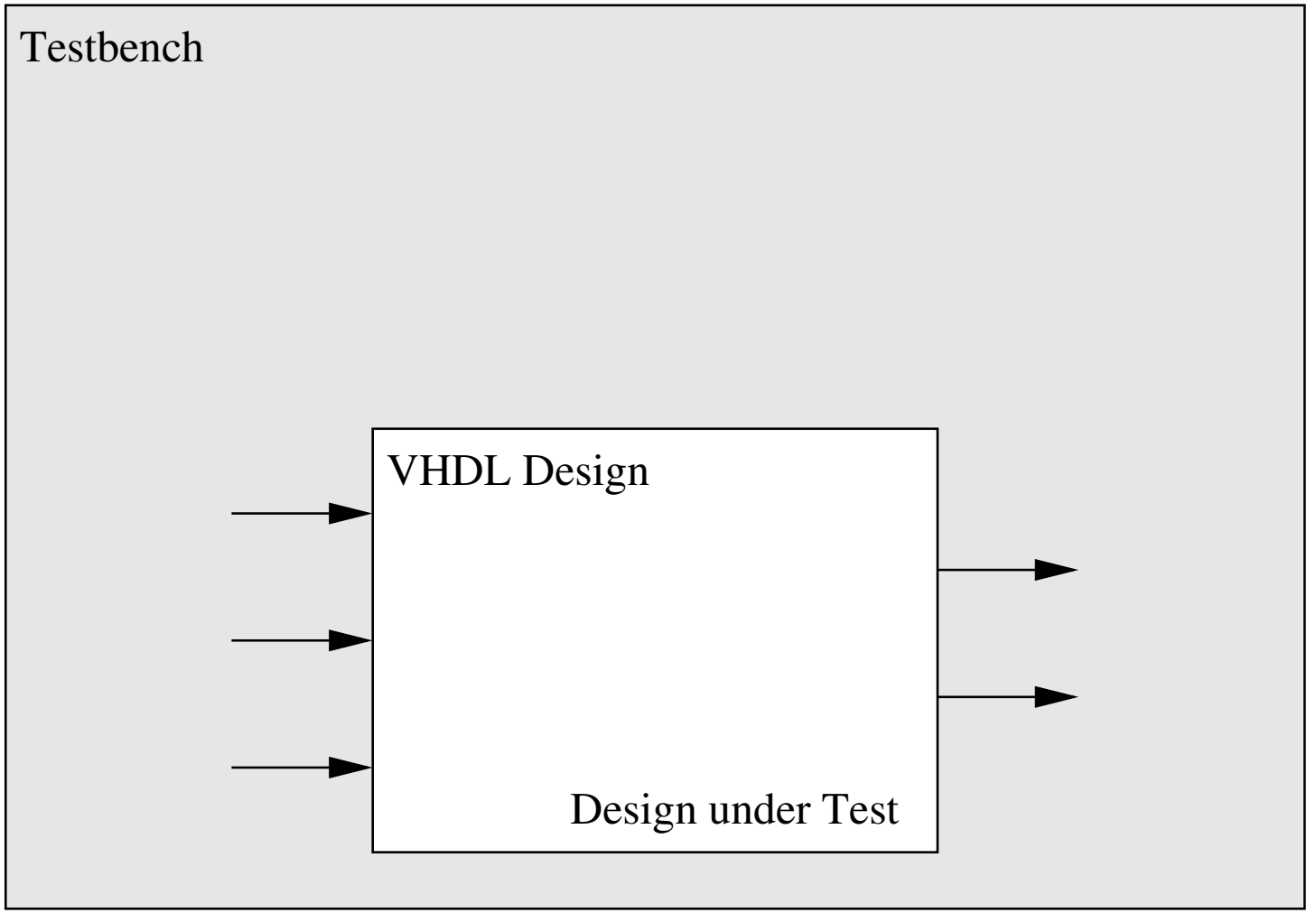
Outline

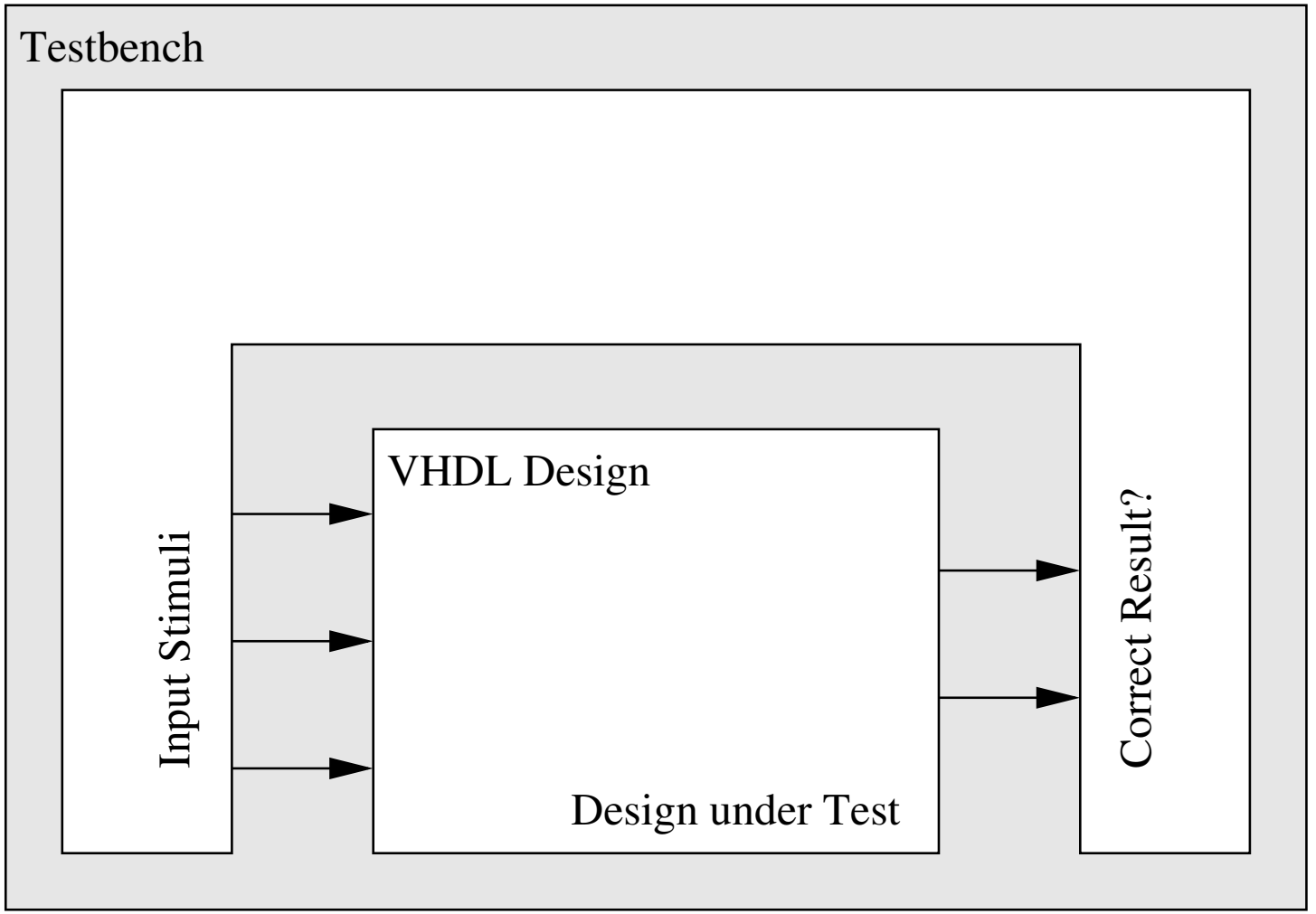
There are several methods to validate hardware components

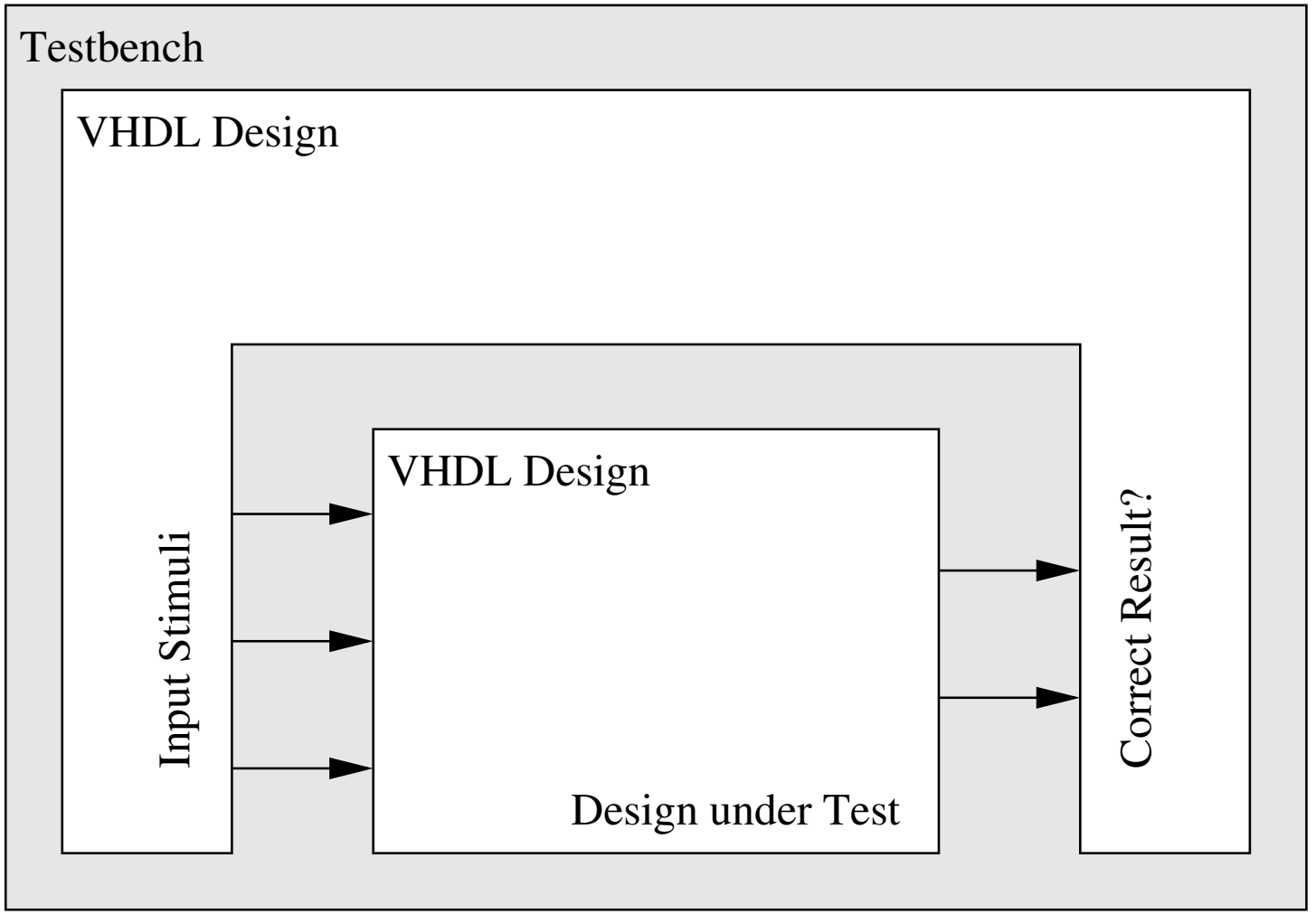
- Formal verification techniques
  - Equivalence checking
  - Symbolic model checking
  - Bounded model checking
- Simulation (complete vs. incomplete)

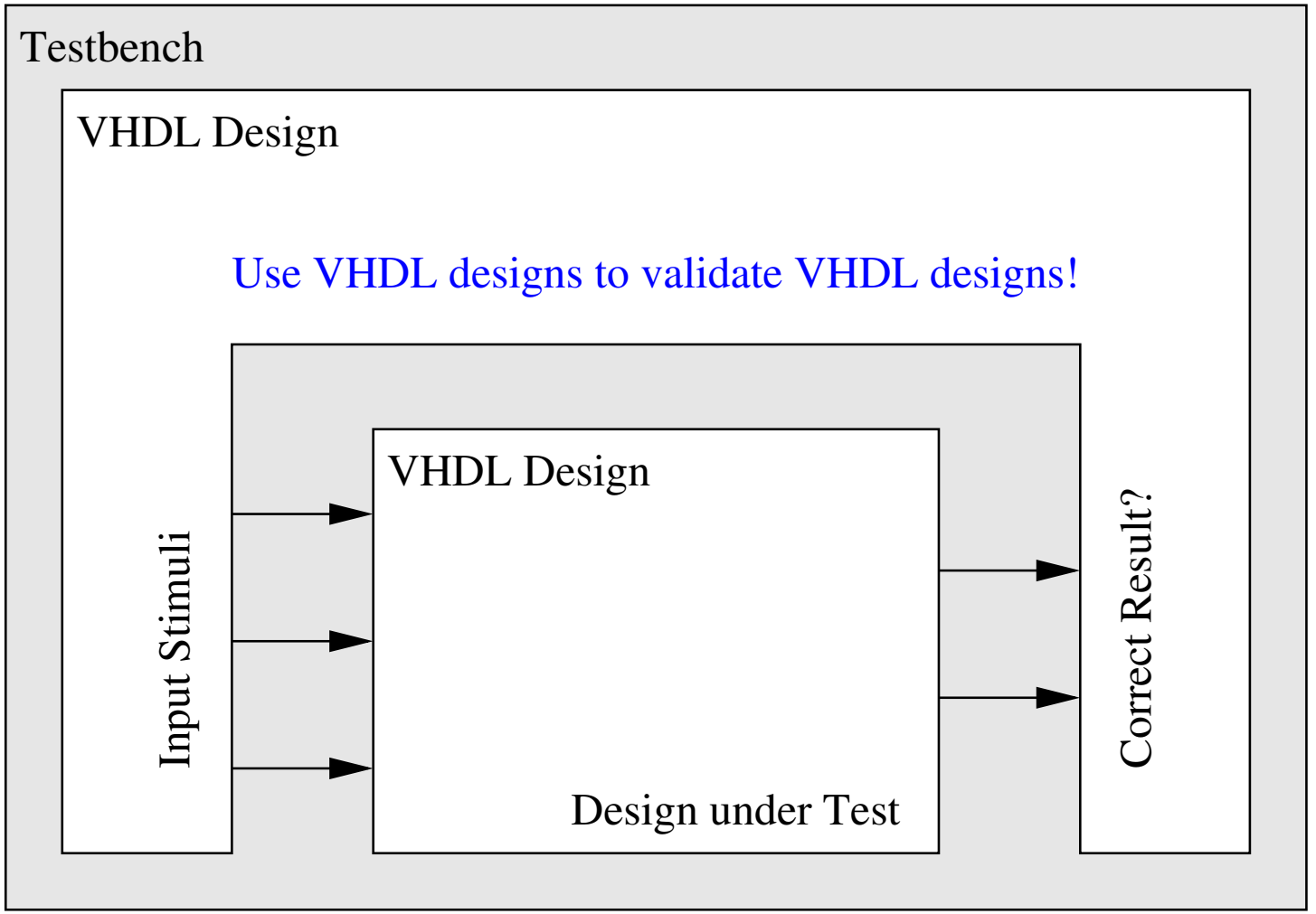
Here, the focus is on simulating VHDL designs using testbenches (by means of some examples)

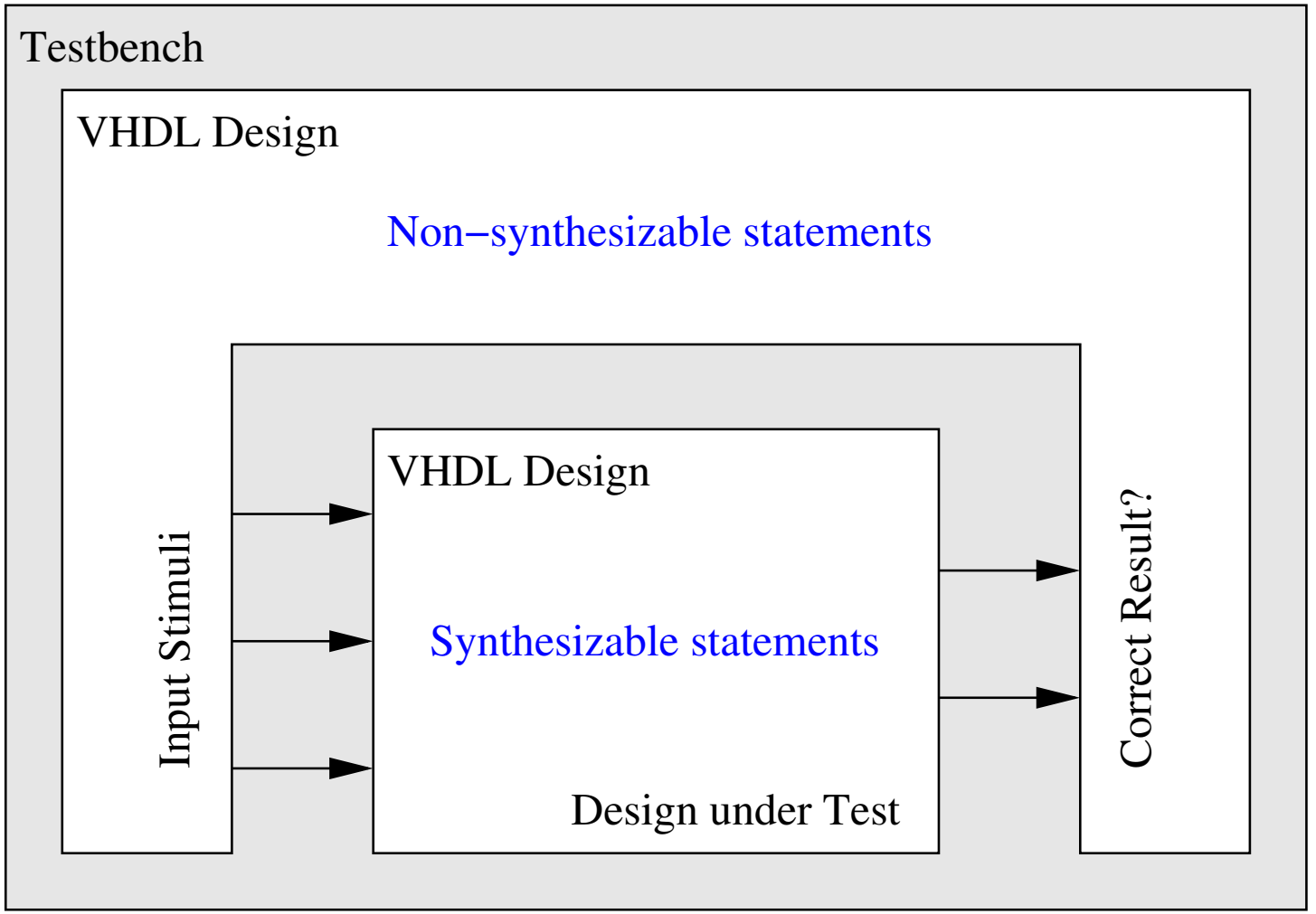
Testbench


Testbench
VHDL Design
Design under Test


Testbench
Input Stimuli
VHDL Design
Design under Test
Correct Result?


Testbench
VHDL Design
Input Stimuli
VHDL Design
Design under Test
Correct Result?


Testbench
VHDL Design
Use VHDL designs to validate VHDL designs!
VHDL Design
Input Stimuli
Correct Result?
Design under Test


Testbench
VHDL Design
Non−synthesizable statements
Input Stimuli
VHDL Design
Synthesizable statements
Design under Test
Correct Result?

Testbenches – OR Gate (Slide 19)

Testbench orGate

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

entity orGate_vhd_tst is
end orGate_vhd_tst;

architecture orGate_arch of orGate_vhd_tst is
   signal a, b, c : std_logic;
   component orGate
      port ( a, b : in std_logic; c : out std_logic );
   end component;
begin
   i1 : orGate port map ( a, b, c );
   process
   begin
      a <= '0'; b <= '0'; wait for 50 ns;
      a <= '0'; b <= '1'; wait for 50 ns;
      a <= '1'; b <= '0'; wait for 50 ns;
      a <= '1'; b <= '1'; wait for 50 ns;
      wait;
   end process;
end orGate_arch;
```

Testbenches – Full Adder (Slides 21–23)

UNI FREIBURG

Testbench fullAdder

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

entity fullAdder_vhd_tst is
end fullAdder_vhd_tst;

architecture fullAdder_arch of fullAdder_vhd_tst is
   signal a, b, cin, sum, cout : std_logic;
   component fullAdder
      port ( a, b, cin : in std_logic; sum, cout : out std_logic );
   end component;
begin
   i1 : fullAdder port map ( a, b, cin, sum, cout );
   process
   begin
      a <= '0'; b <= '0'; cin <= '0'; wait for 50 ns;
      a <= '0'; b <= '0'; cin <= '1'; wait for 50 ns;
      -- more test cases...
      a <= '1'; b <= '1'; cin <= '1'; wait for 50 ns;
      wait;
   end process;
end fullAdder_arch;
```

Testbenches – Counter (Slide 24)

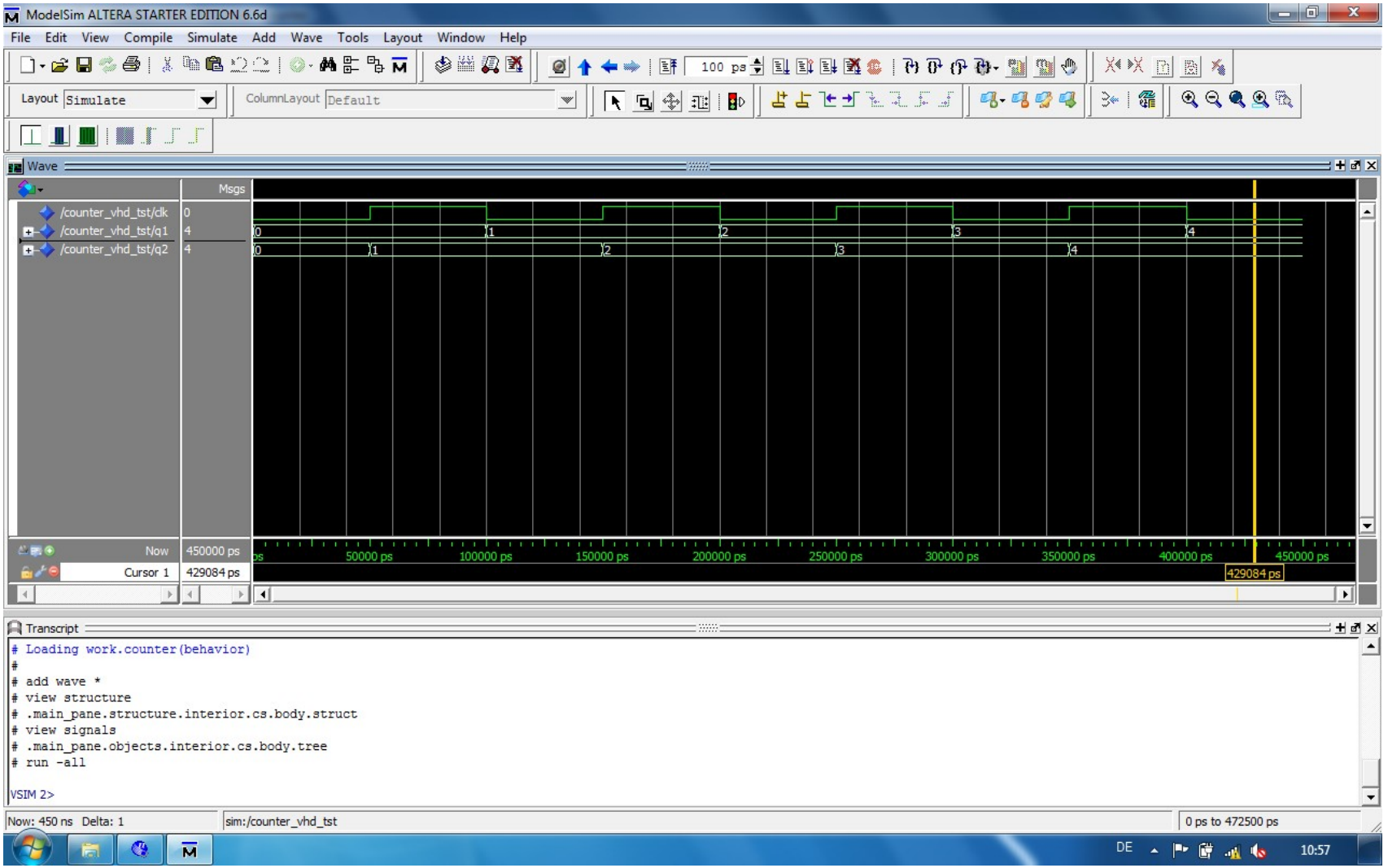
Testbench counter

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

entity counter_vhd_tst is
end counter_vhd_tst;

architecture counter_arch of counter_vhd_tst is
   signal clk : std_logic;
   signal q1, q2 : std_logic_vector (4 downto 0);
   component counter
      port ( clk : in std_logic; q1, q2 : out std_logic_vector (4 downto 0) );
   end component;
begin
   i1 : counter port map ( clk, q1, q2 );
   process
   begin
      clk <= '0'; wait for 50 ns;
      clk <= '1'; wait for 50 ns;
      clk <= '0'; wait for 50 ns;
      -- more test cases...
      wait;
   end process;
end counter_arch;
```

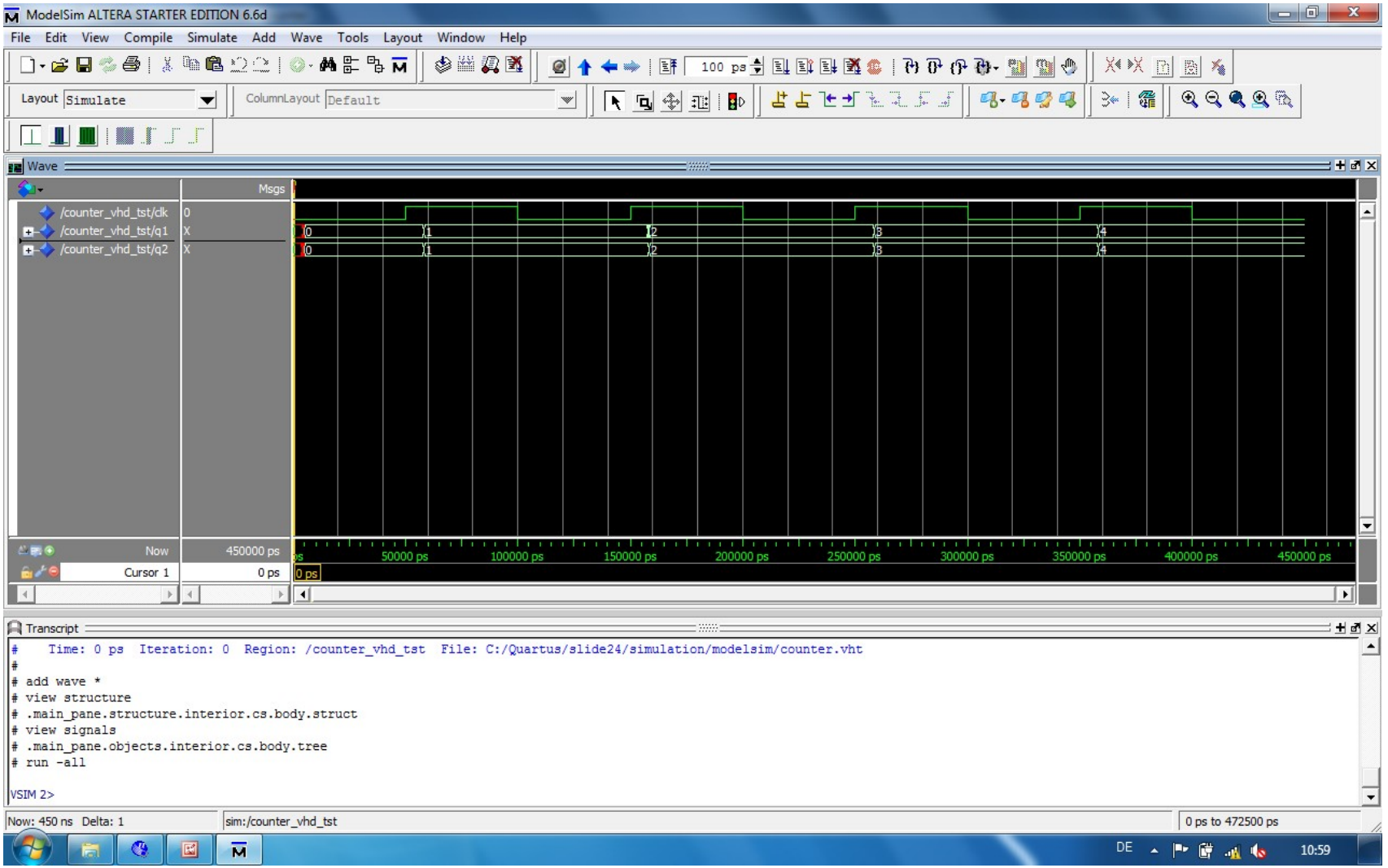
Altera provides two modes of simulation

- RTL: simulates the design without timing information
- Gate-Level: simulates the design with post-fit timing information (i.e. taking FPGA-specific gate delays into account, more realistic model)

Testbenches – Counter (Slide 24, RTL)



Testbenches – Counter (Slide 24, Gate-Level)



Testbenches – Modified Counter

Entity & Architecture

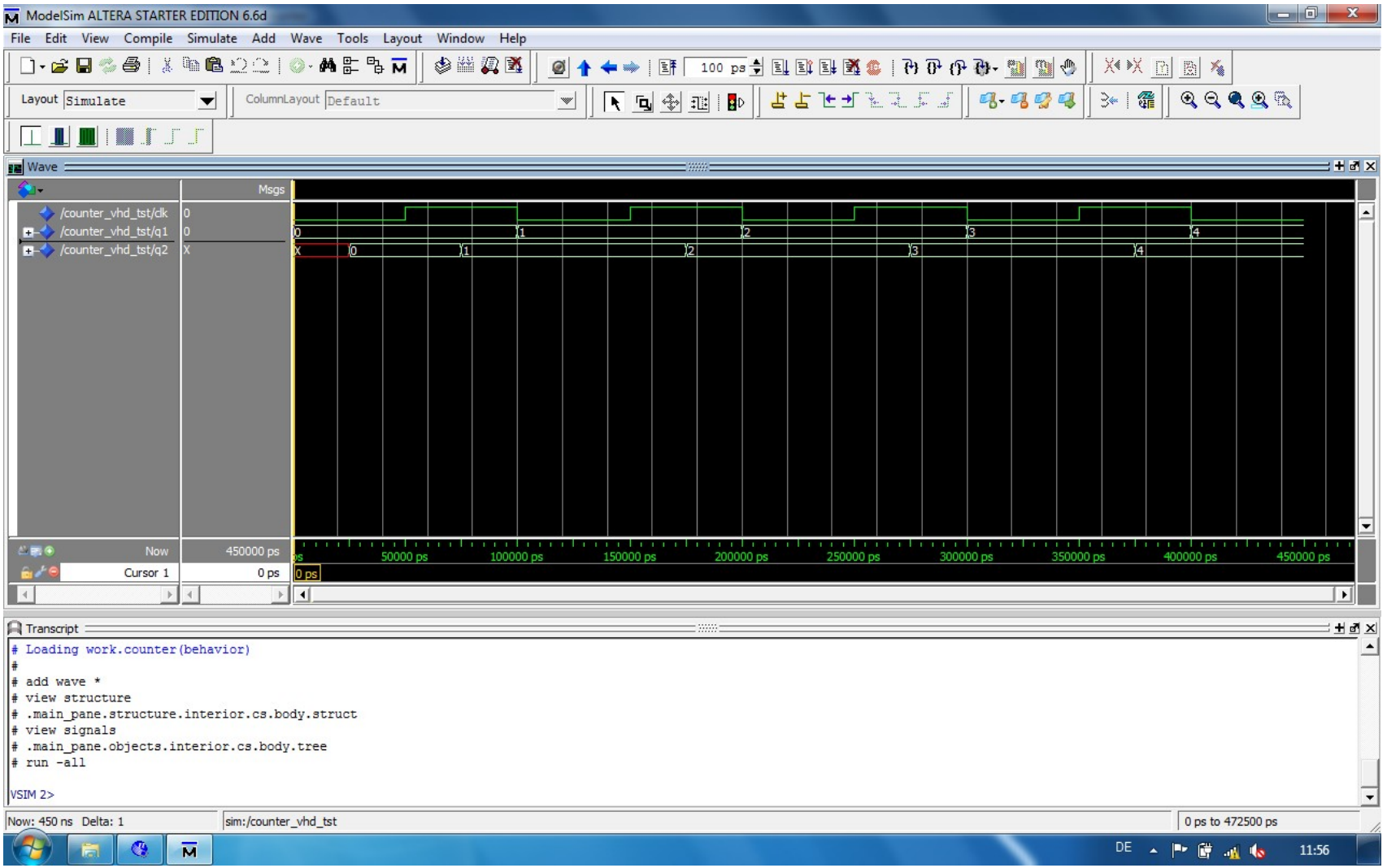
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
   port ( clk: in std_logic; q1, q2: out std_logic_vector (4 downto 0) );
end counter;

architecture behavior of counter is
   signal cnt: std_logic_vector (4 downto 0) := “00000”;
begin
   process ( clk )
   begin
      if clk = ’1’ then cnt <= cnt + 1; end if;
      q1 <= cnt;
   end process;
   q2 <= cnt after 25 ns;
end behavior;
```

Testbenches – Modified Counter

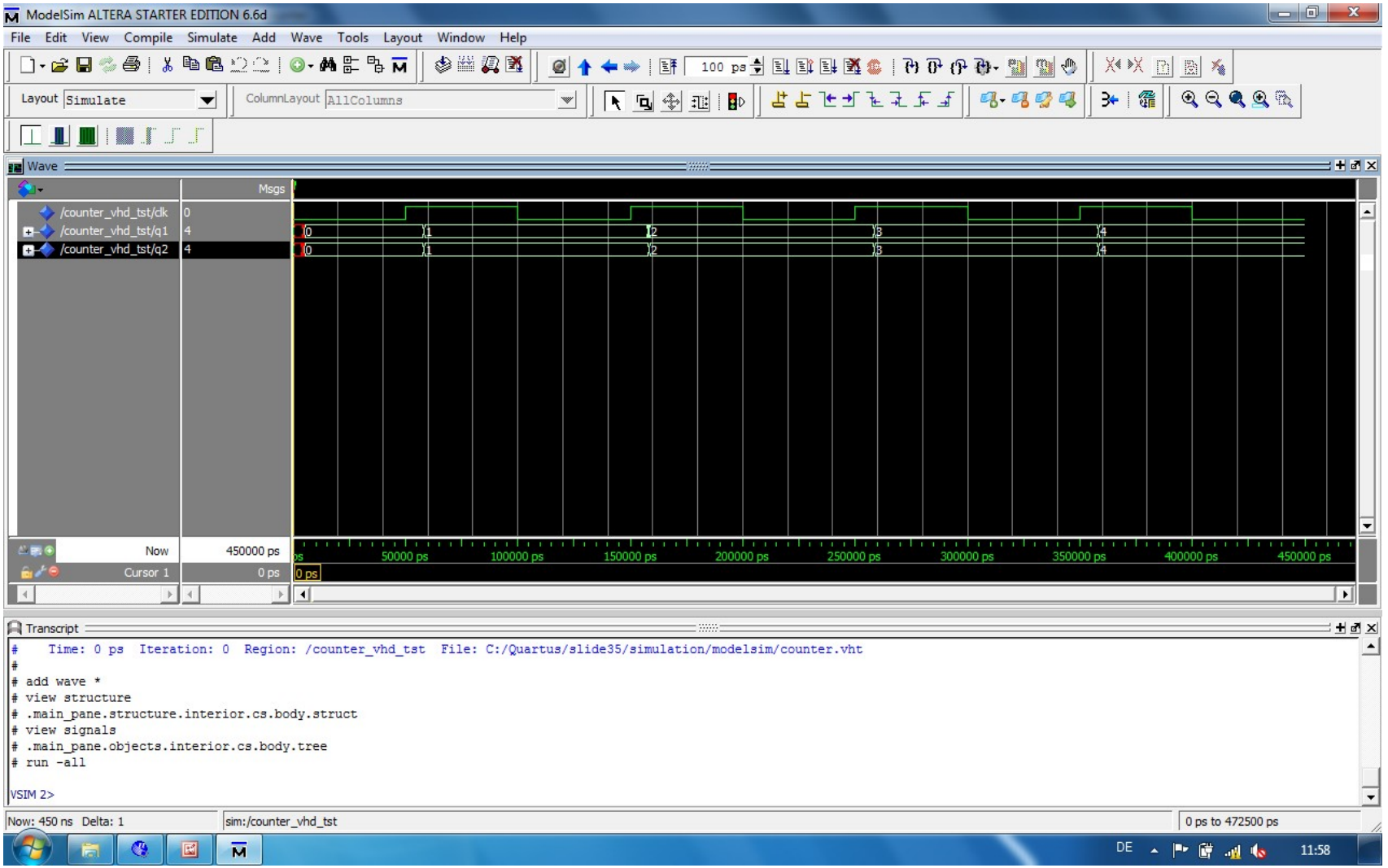
Testbench counter

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

entity counter_vhd_tst is
end counter_vhd_tst;

architecture counter_arch of counter_vhd_tst is
   signal clk : std_logic;
   signal q1, q2 : std_logic_vector (4 downto 0);
   component counter
      port ( clk : in std_logic; q1, q2 : out std_logic_vector (4 downto 0) );
   end component;
begin
   i1 : counter port map ( clk, q1, q2 );
   process
   begin
      clk <= '0'; wait for 50 ns;
      clk <= '1'; wait for 50 ns;
      clk <= '0'; wait for 50 ns;
      -- more test cases...
      wait;
   end process;
end counter_arch;
```

Testbenches – Modified Counter (RTL)



Testbenches – Modified Counter (Gate-Level)



Testbenches – Counter (Slide 25)

UNI FREIBURG

Testbench counter

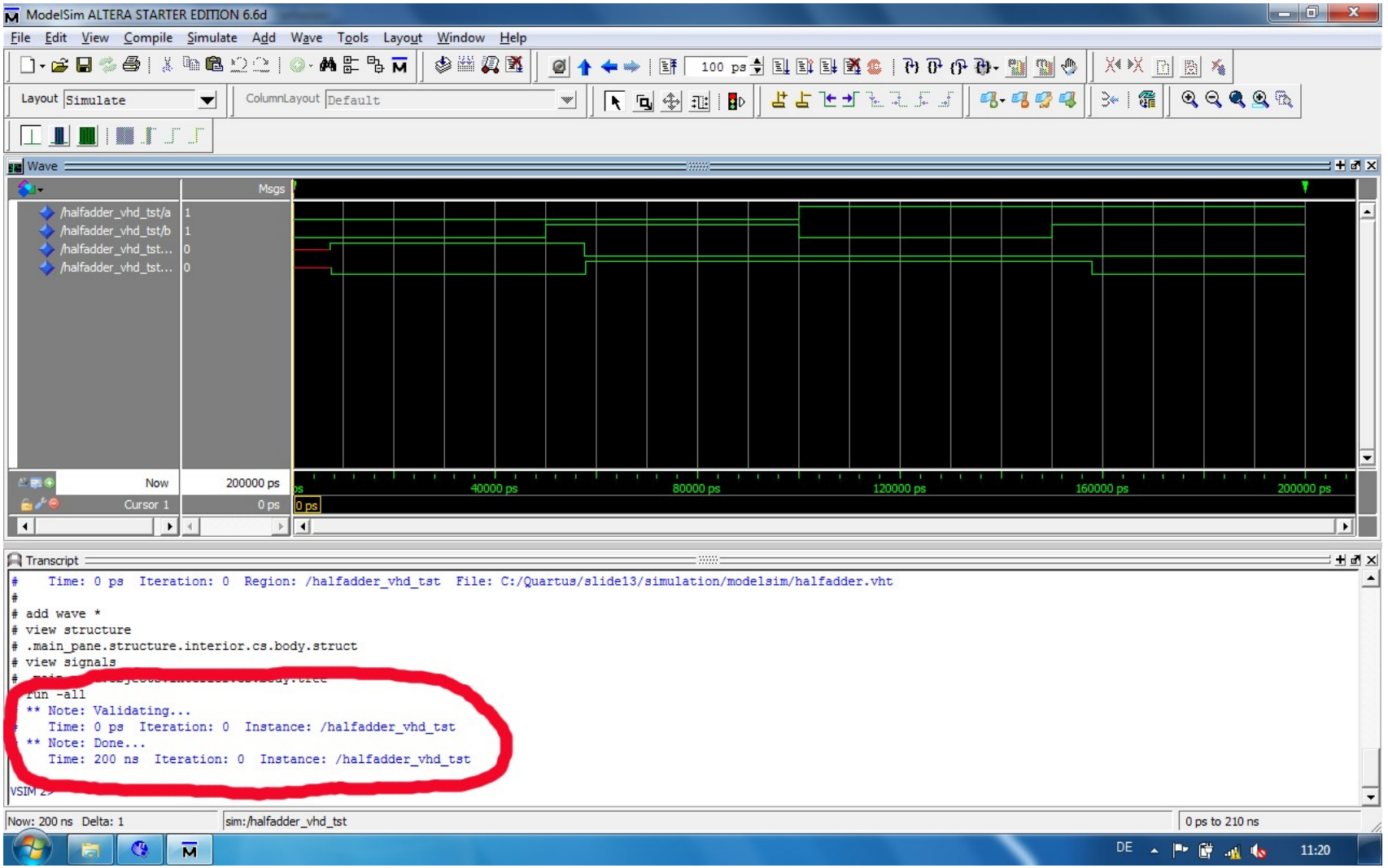
```
library ieee;
use ieee.std_logic_1164.all;

entity counter_vhd_tst is
end counter_vhd_tst;

architecture counter_arch of counter_vhd_tst is
   signal clk : std_logic;
   signal q : std_logic_vector (4 downto 0);
   component counter
      port ( clk : in std_logic; q : out std_logic_vector (4 downto 0) );
   end component;
begin
   i1 : counter port map ( clk, q );
   process
   begin
      clk <= '0'; wait for 50 ns;
      clk <= '1'; wait for 50 ns;
      clk <= '0'; wait for 50 ns;
      -- more test cases...
      wait;
   end process;
end counter_arch;
```

Testbenches – Half Adder (Slide 13)

Testbench halfAdder

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

entity halfadder_vhd_tst is
end halfadder_vhd_tst;

architecture halfadder_arch of halfadder_vhd_tst is
   signal a, b, cout, sum : std_logic;
   component halfadder
      port ( a, b : in std_logic; cout, sum : out std_logic );
   end component;
begin
   i1 : halfadder port map ( a, b, cout, sum );
   process
   begin
      report "Validating...";
      a <= '1'; b <= '0'; wait for 50 ns;
      assert (cout = '0' and sum = '1') report "Error";
      -- more test cases...
      report "Done...";
      wait;
   end process;
end halfadder_arch;
```

Testbenches – Half Adder (Slide 13)



Outline

Identifiers

- Allowed characters: ’a’ to ’z’, ’0’ to ’9’, and ’_’
- Must begin with a letter and the last one cannot be an underscore
- Only one underscore allowed at a time
- Case insensitive (’myBus’ and ’mybus’ refer to the same VHDL object)
- Reserved words cannot be used
- Examples
  - Allowed identifiers: myBus, my_bus_8_bit
  - Illegal identifiers: _bus, 8Bit_bus, myBus__8Bit

Reserved words in VHDL-93

abs, **access**, **after**, **alias**, **all**, **and**, **architecture**, **array**, **assert**, **attribute**, **begin**, **block**, **body**, **buffer**, **bus**, **case**, **component**, **configuration**, **constant**, **disconnect**, **downto**, **else**, **elsif**, **end**, **entity**, **exit**, **file**, **for**, **function**, **generate**, **generic**, **group**, **guarded**, **if**, **impure**, **in**, **inertial**, **inout**, **is**, **label**, **library**, **linkage**, **literal**, **loop**, **map**, **mod**, **nand**, **new**, **next**, **nor**, **not**, **null**, **of**, **on**, **open**, **or**, **others**, **out**, **package**, **port**, **postponed**, **procedure**, **process**, **pure**, **range**, **record**, **register**, **reject**, **rem**, **report**, **return**, **rol**, **ror**, **select**, **severity**, **signal**, **shared**, **sla**, **sll**, **sra**, **srl**, **subtype**, **then**, **to**, **transport**, **type**, **unaffected**, **units**, **until**, **use**, **variable**, **wait**, **when**, **while**, **with**, **xnor**, **xor**

Comments

- Comments start with $--$
- Example
  - a <= '1'; $--$ Signal 'a' is set to logical 1.

Outline

Standard data types

- bit: 0, 1
- boolean: true, false
- character: most ASCII characters
- integer: $-(2^{31}-1), \ldots, (2^{31}-1)$
- real: $-1.7e38, \ldots, 1.7e38$
- time: 1 fs, ..., 1 hr

User-defined integer and floating-point data types

- Syntax
  - **type** identifier **is** type_definition;
  - **subtype** sub_identifier **is** identifier [type_definition];
- Examples
  - **type** small **is range** 0 **to** 64;
  - **type** result32 **is range** 31 **downto** 0;
  - **type** ttl **is range** 0.0 **to** 5.0;
  - **subtype** result16 **is** result32 **range** 15 **downto** 0;

User-defined enumeration types

- Syntax
  - **type** identifier **is** (element [,element]);
- Examples
  - **type** state **is** (idle, start, stop);
  - **type** hexDigits **is** ('0', …, '9', 'A', 'B', 'C', 'D', 'E', 'F');
  - **type** std_logic **is** ('0', '1', 'U', 'X', 'Z', 'W', 'L', 'H', '-');
    - Defined in package ieee.std_logic_1164

Data Types – std_logic

orGateSTD

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

entity orGateSTD is
   port ( a, b: in std_logic; c: out std_logic );
end orGateSTD;

architecture arch1 of orGateSTD is
begin
   c <= a or b;
end arch1;

architecture arch2 of orGateSTD is
begin
   c <= '1' when (a = '1' or b = '1') else '0';
end arch2;
```

- Values provided by std_logic
  - '0' – logical zero
  - '1' – logical one
  - 'U' – uninitialized
  - 'X' – unknown
  - 'Z' – high impedance
  - 'W' – weak unknown
  - 'L' – weak low
  - 'H' – weak high
  - '-' – don't care
- Compared to data type "bit", std_logic allows more realistic VHDL designs
- Is there a difference between the two implementations?

Examples – OR Gate (using std_logic and arch1)

UNI FREIBURG

a / b	0	1	U	X	Z	W	L	H	-
0	0	1	U	X	X	0	0	1	0
1	1	1	1	1	1	1	1	1	1
U	U	1	U	X	X	X	X	1	X
X	X	1	X	X	X	X	X	1	X
Z	X	1	X	X	X	X	X	1	X
W	0	1	X	X	X	X	X	1	X
L	0	1	X	X	X	X	0	1	X
H	1	1	1	1	1	1	1	1	1
-	0	1	X	X	X	X	X	1	X

Examples – OR Gate (using std_logic and arch2)

UNI FREIBURG

a / b	0	1	U	X	Z	W	L	H	-
0	0	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
U	0	1	0	0	0	0	0	0	0
X	0	1	0	0	0	0	0	0	0
Z	0	1	0	0	0	0	0	0	0
W	0	1	0	0	0	0	0	0	0
L	0	1	0	0	0	0	0	0	0
H	0	1	0	0	0	0	0	0	0
-	0	1	0	0	0	0	0	0	0

User-defined arrays

- Syntax
  - **type** identifier **is array (**range [,range]**) of** element_type;
- Examples
  - **type** myVarArray **is array (**15 **downto** 0**) of** integer;
  - **type** myBitArray **is array (**0 **to** 7**) of** bit;
  - **type** my4x3Array **is array (**1 **to** 4, 1 **to** 3**) of** std_logic;

Some remarks

- VHDL is a strongly typed language
  - **type** t1 **is range** 1 **to** 64;
  - **type** t2 **is range** 1 **to** 64;

  $\Rightarrow$ t1 and t2 are different data types!
- Packages such as ieee.std_logic_1164 and ieee.std_logic_arith provide various conversion routines
- Some data types and so called attributes have been left out in this overview
  - natural, positive, record, access, ...
  - base, length, event, pos, succ, pred, leftof, ...

Outline

VHDL supports different classes of operators:

1. Logical operators
2. Relational operators
3. Shift operators
4. Addition operators
5. Unary operators
6. Multiplying operators
7. Miscellaneous operators

Order of precedence is highest for class 7 operators, followed by class 6 with the lowest precedence for class 1

Logical operators

- **and**, **or**, **nand**, **nor**, **xor**, **xnor**

Relational operators

- $=$: equal
- $/=$: not equal
- $<$: smaller than
- $<=$: smaller than or equal
- $>$: greater than
- $>=$: greater than or equal

Operators – Example

Entity & Architecture

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

entity smallerThan is
   port ( a, b: out std_logic );
end smallerThan;

architecture behavior of smallerThan is
   signal cnt: integer range 0 to 10 := 5;
begin
   a <= '1' when (cnt <= 5) else '0';
   b <= '1' when (cnt <= 3) else '0';
end behavior;
```

Shift operators (on any one-dimensional array type with elements of type bit or Boolean)

- **sll**: shift left logical (fill right vacated bits with 0)
- **srl**: shift right logical (fill left vacated bits with 0)
- **sla**: shift left arithmetic (fill right vacated bits with rightmost bit)
- **sra**: shift right arithmetic (fill left vacated bits with leftmost bit)
- **rol**: rotate left (circular)
- **ror**: rotate right (circular)

Operators – Example

Entity & Architecture

```
entity shiftLeftLogical is
   port ( a: in bit_vector (5 downto 0); b: out bit_vector (5 downto 0) );
end shiftLeftLogical;

architecture behavior of shiftLeftLogical is
begin
   b <= a sll 1;
end behavior;
```

Addition operators (require IEEE packages std_logic_1164 and std_logic_arith / std_logic_unsigned)

- $+$: addition (on operands of any numeric type)
- $-$: subtraction (on operands of any numeric type)
- &: concatenation of two vectors

Unary operators (used to specify the sign of a numeric type)

- $+$: identity
- $-$: negation

Multiplying operators

- $*$: multiplication (on operands of any numeric type)
- $/$: division (on operands of any numeric type)
- **mod**: modulus (any integer type)
- **rem**: remainder (any integer type)

Miscellaneous operators

- **abs**: absolute value (any numeric type)
- **not**: logical negation (any bit or Boolean type)
- $**$: Exponentiation (floating/integer or integer/integer)

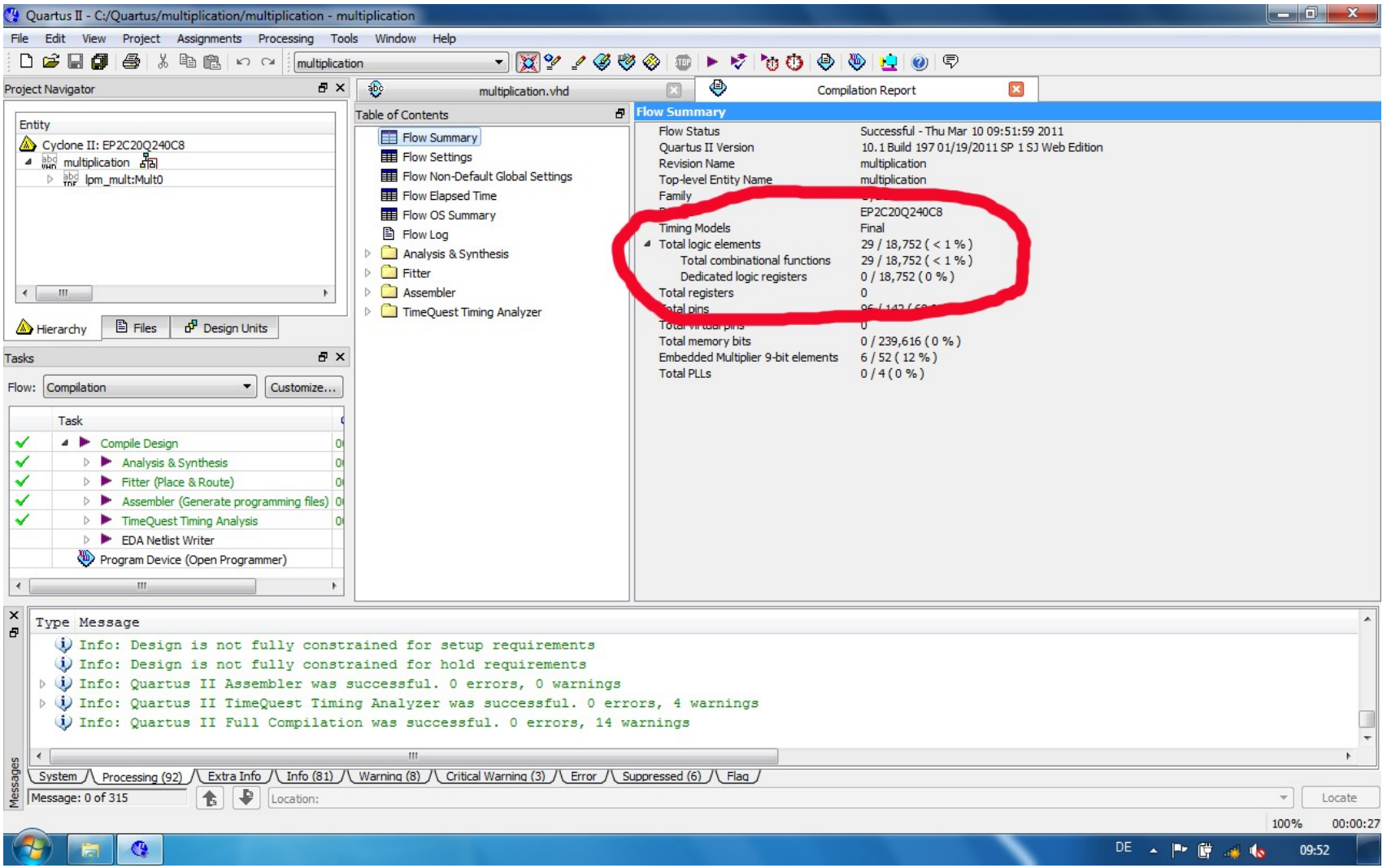
Entity & Architecture of multiplication

```
entity multiplication is
   port ( a, b: in integer; c: out integer );
end multiplication;

architecture behavior of multiplication is
begin
   c <= a * b;
end behavior;
```

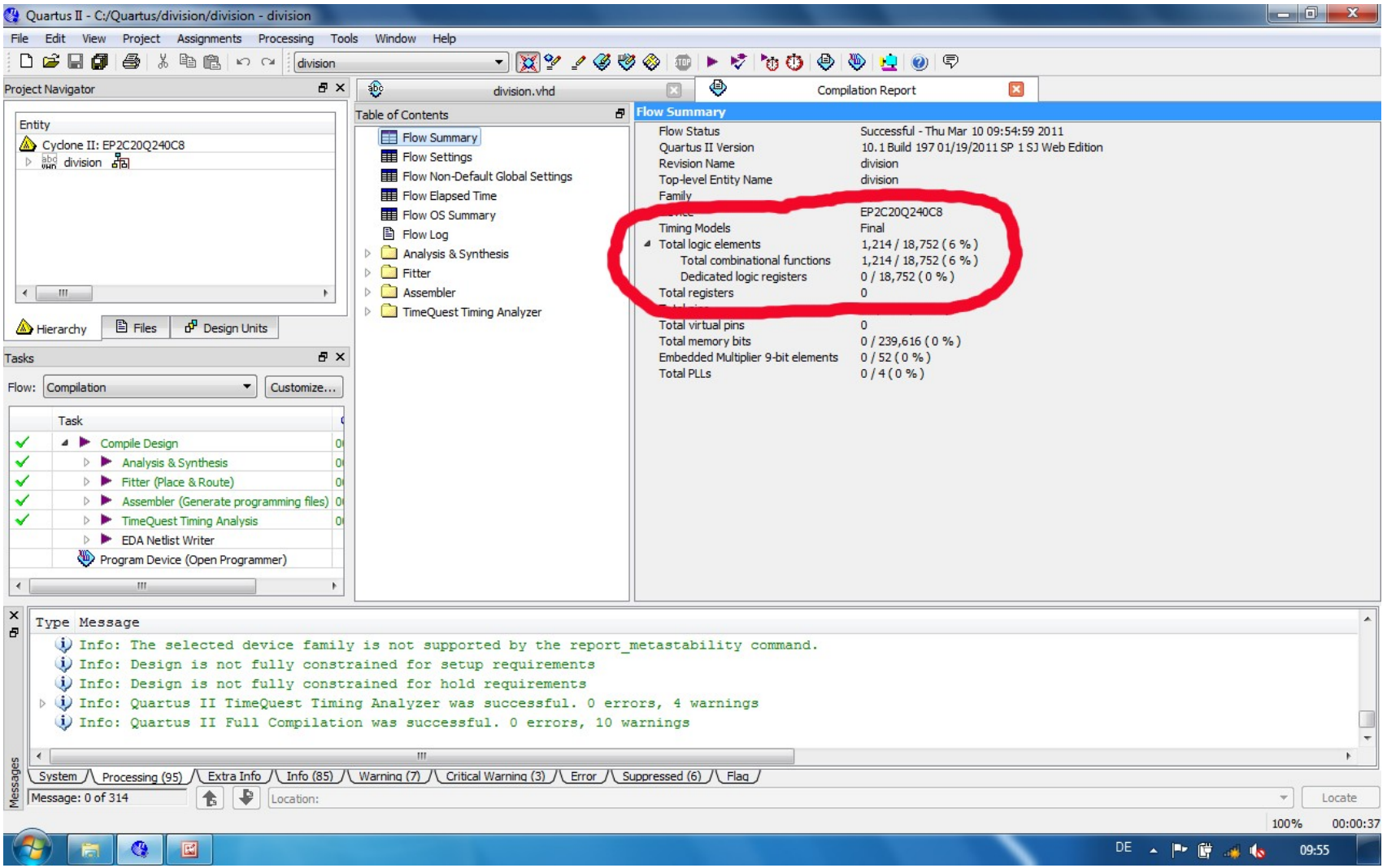
Entity & Architecture of division

```
entity division is
   port ( a, b: in integer; c: out integer );
end division;

architecture behavior of division is
begin
   c <= a / b;
end behavior;
```

Operators – Example



Operators – Example



Outline

UNI FREIBURG

Constants

- Can be of any data type
- Cannot be changed or written to
- Examples
  - **constant** PI: real := 3.1415;
  - **constant** PERIOD: time := 100 ns;
  - **type** vecType **is array** (0 **to** 3) **of** integer;
    constant VEC: vecType := (2,4,-1,7);

Variables

- Like variables in (other) programming languages
- Used to store temporary values
- Have to be declared within processes and are only visible in this scope
- Are updated immediately without any delay
- Examples
  - Declaration
    - **variable** var1: boolean := false;
    - **variable** var2: integer **range** 0 **to** 16 := 4;
    - **type** var3Type **is array** (3 **downto** 0) **of** std_logic;
      variable var3: var3Type;
  - Assignment
    - var1 := true;
    - var2 := var2 + 6;
    - var3(2) := 'X';

Signals

- Represent a wire or register value
- Can be of any data type
- Can be declared in architectures only
- Have a time delay associated with them
  - user-specified, e.g. ”**after** 10 ns”
  - small delta delay
- Examples
  - **signal** sum: std_logic;
  - **signal** clk: bit;
  - **signal** data: std_logic_vector (0 **to** 7) := “00X0X011”;
  - **signal** value: integer **range** 16 **to** 31 := 17;

Simple signal assignments

- sum <= (a **xor** b) **after** 2 ns;
- data(1) <= 'X'; −− implicit delta delay.
- clk <= '0', '1' **after** 5 ns, '0' **after** 10 ns, '1' **after** 15 ns;

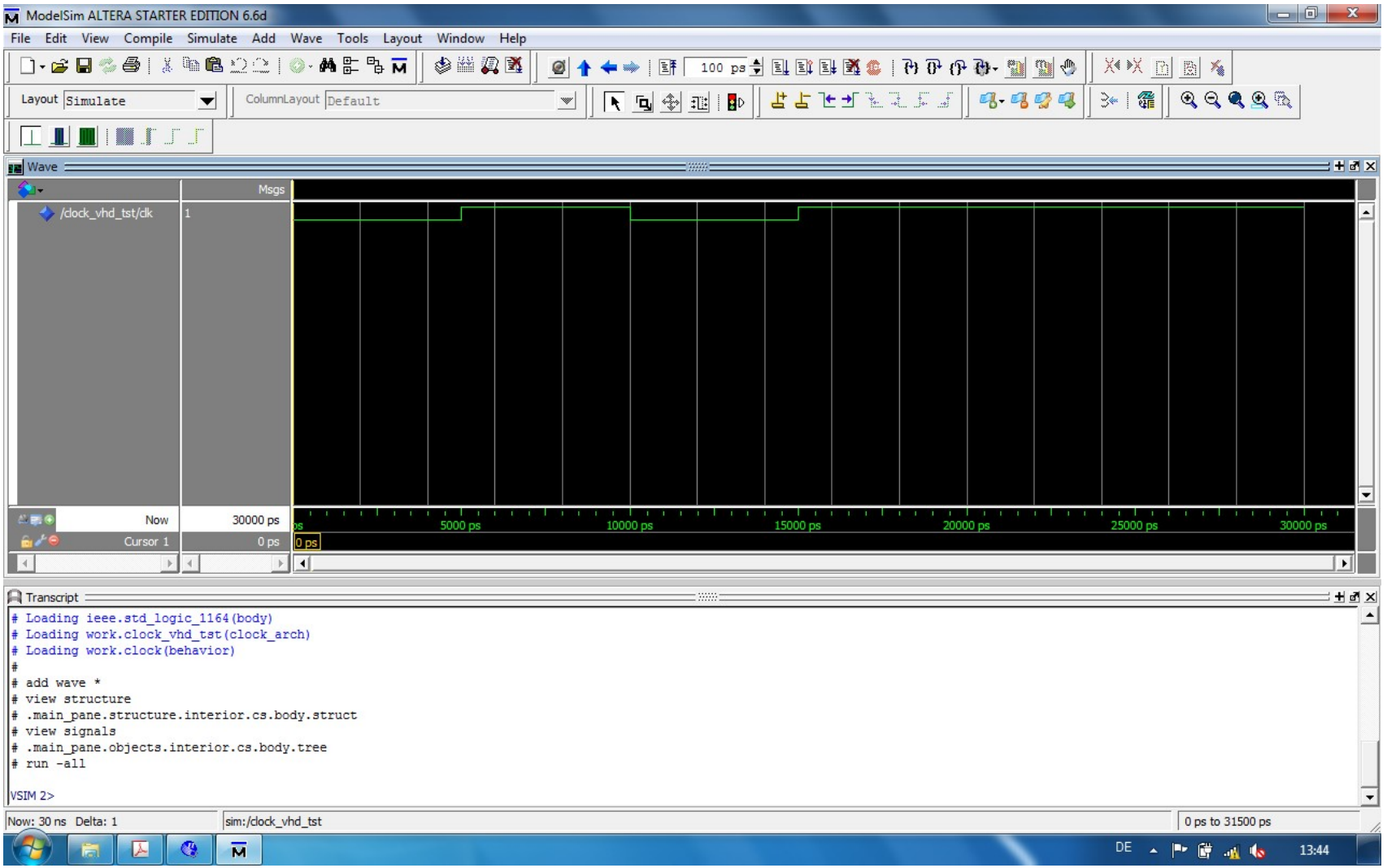
Conditional signal assignments

- **when-else**
- **with-select**

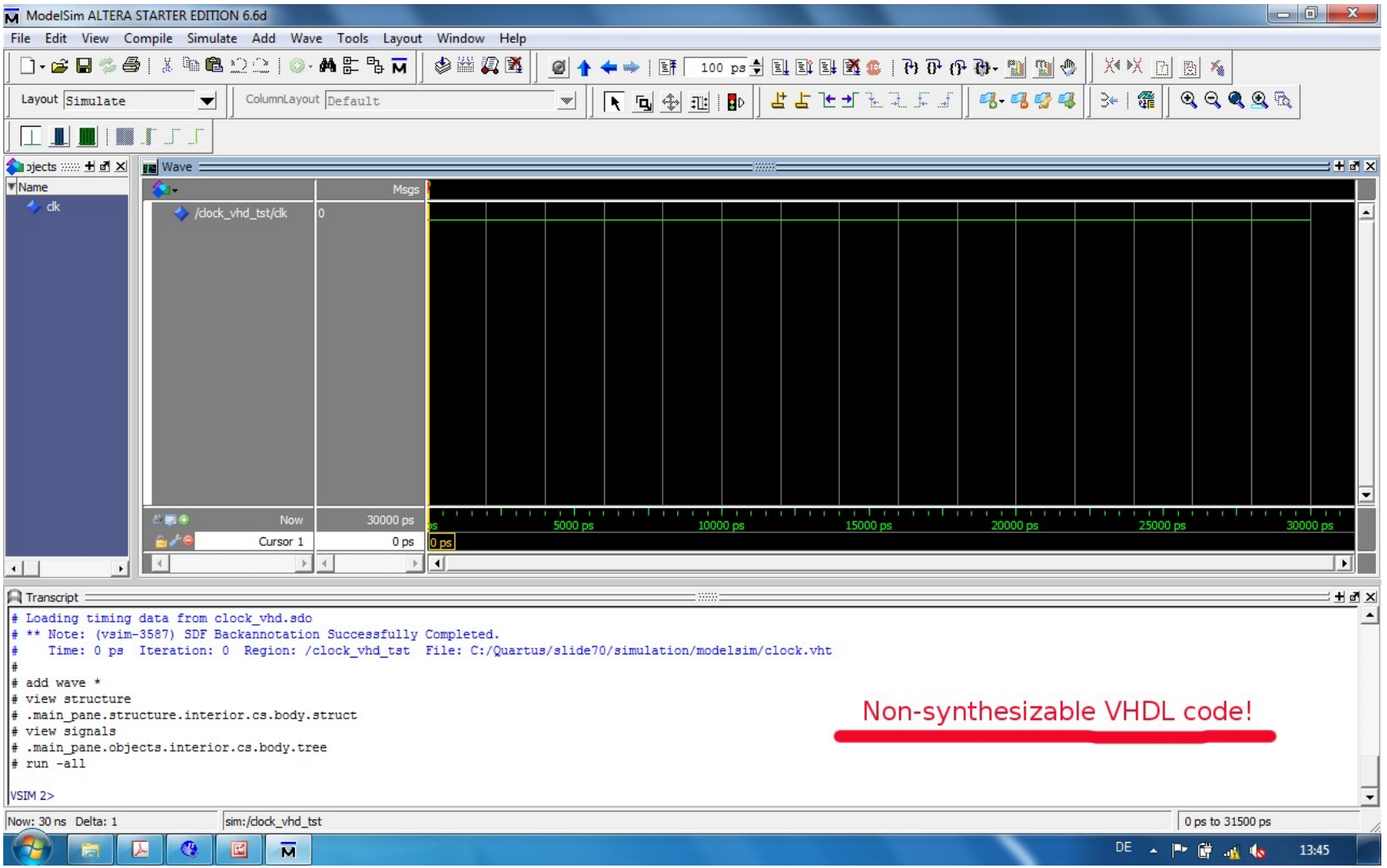
Constants, Variables & Signals – Example

Entity & Architecture of clock

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

entity clock is
   port ( clk: out std_logic );
end clock;

architecture behavior of clock is
begin
   clk <= '0', '1' after 5 ns, '0' after 10 ns, '1' after 15 ns;
end behavior;
```


ModelSim ALTERA STARTER EDITION 6.6d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Wave
Msgs
/clock_vhd_tst/clk
Now
30000 ps
Cursor 1
0 ps
5000 ps
10000 ps
15000 ps
20000 ps
25000 ps
30000 ps
Transcript
Loading ieee.std_logic_1164(body)
Loading work.clock_vhd_tst(clock_arch)
Loading work.clock(behavior)
add wave *
view structure
.main_pane.structure.interior.cs.body.struct
view signals
.main_pane.objects.interior.cs.body.tree
run -all
VSIM 2>
Now: 30 ns Delta: 1
sim:/clock_vhd_tst
0 ps to 31500 ps
13:44

Constants, Variables & Signals – Gate-Level Sim.



Constants, Variables & Signals – Example

UNI FREIBURG

Entity

```
entity mux4to1 is
   port ( i3, i2, i1, i0: in bit;
          sel: in bit_vector (1 downto 0);
          otp: out bit );
end mux4to1;
```

when-else

```
architecture wElse of mux4to1 is
begin
   otp <= i0 when sel = “00” else
          i1 when sel = “01” else
          i2 when sel = “10” else
          i3;
end wElse;
```

with-select

```
architecture wSelect of mux4to1 is
begin
   with sel select
      otp <= i0 when “00”,
             i1 when “01”,
             i2 when “10”,
             i3 when others ;
end wSelect;
```

when-else

- Specifies a logic function in the form of a truth table

with-select

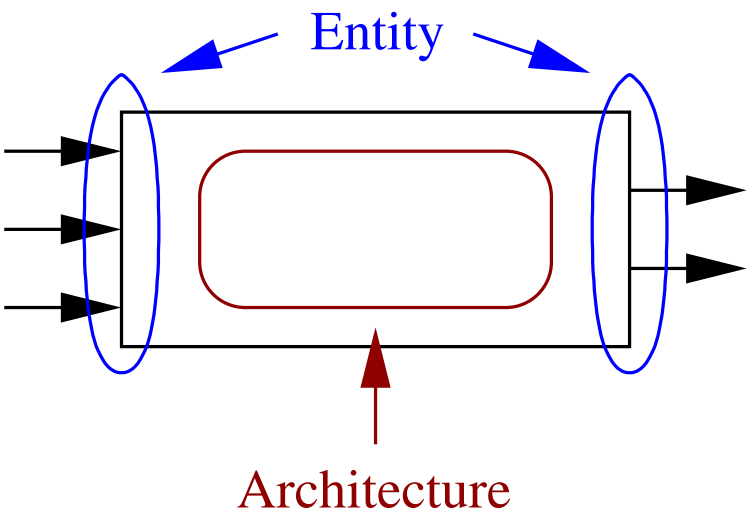
- Choices can express
  - single values, e.g. . . . **when** “01”
  - a range, e.g. . . . **when** 51 **to** 60
  - combined choices, e.g. . . . **when** “00” **or** “01” **or** “11”
- No two choices are allowed to overlap
- All possible values of the “choice expression” have to be covered by the set of choices (unless an **others** choice is present)

Both **when-else** and **with-select** cannot be used inside a process!

Outline

Each VHDL design is based on two parts

- An entity defining the system's interface
- An architecture describing an entity's implementation



The entity describes the interface of a digital system:

```
entity name_of_entity is
   [ generic ( parameter list ); ]
   [ port ( I/O pins ); ]
   [ begin assertions to be checked; ]
end [ entity ] [ name_of_entity ];
```

- Generic declarations are optional and determine local constants used for e.g. sizing the entity (bus widths, ...)
- Assertions are optional and can be used to perform consistency checks

Entities & Architectures – Examples

“Empty” entity

```
entity test is
end test;
```

D-Flipflop

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

entity flipflop is
   port ( data, clk, set, reset: in std_logic; q: out std_logic );
end flipflop;
```

8-bit 4-to-1 multiplexer

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

entity mux_4_to_1_8bit is
   port
   (
      i3, i2, i1, i0: in std_logic_vector (7 downto 0);
      sel: in std_logic_vector (1 downto 0);
      otp: out std_logic_vector (7 downto 0)
   );
end mux_4_to_1_8bit;
```

n-bit 4-to-1 multiplexer

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

entity mux_4_to_1_nbit is
   generic ( n: positive );
   port
   (
      i3, i2, i1, i0: in std_logic_vector (n-1 downto 0);
      sel: in std_logic_vector (1 downto 0);
      otp: out std_logic_vector (n-1 downto 0)
   );
end mux_4_to_1_nbit;
```

The architecture defines the implementation of an entity:

```
architecture name_of_architecture of name_of_entity is
  -- Declarations like ...
    -- component declarations
    -- type declarations
    -- signal declarations
    -- constant declarations
    -- function declarations
    -- procedure declarations
begin
  -- Statements, processes, ...
end name_of_architecture;
```

Statements within an architecture are executed in parallel!

Entity

```
entity halfAdder is
   port ( a, b: in bit; sum, cout: out bit );
end halfAdder;
```

Architecture arch1

```
architecture arch1 of halfAdder is
begin
   sum <= a xor b;
   cout <= a and b;
end arch1;
```

Architecture arch2

```
architecture arch2 of halfAdder is
begin
   cout <= a and b;
   sum <= a xor b;
end arch2;
```

$\Rightarrow$ arch1 and arch2 describe the same behavior!

Entities & Architectures – Examples

D-Flipflop

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

entity flipflop is
   port ( data, clk, set, reset: in std_logic; q: out std_logic );
end flipflop;

architecture behavior of flipflop is
begin
   process ( clk, set, reset )
   begin
      if reset = '1' then
         q <= '0';
      elsif set = '1' then
         q <= '1';
      elsif (clk'event and clk = '1') then
         q <= data;
      end if;
   end process;
end behavior;
```

Entities & Architectures – Examples

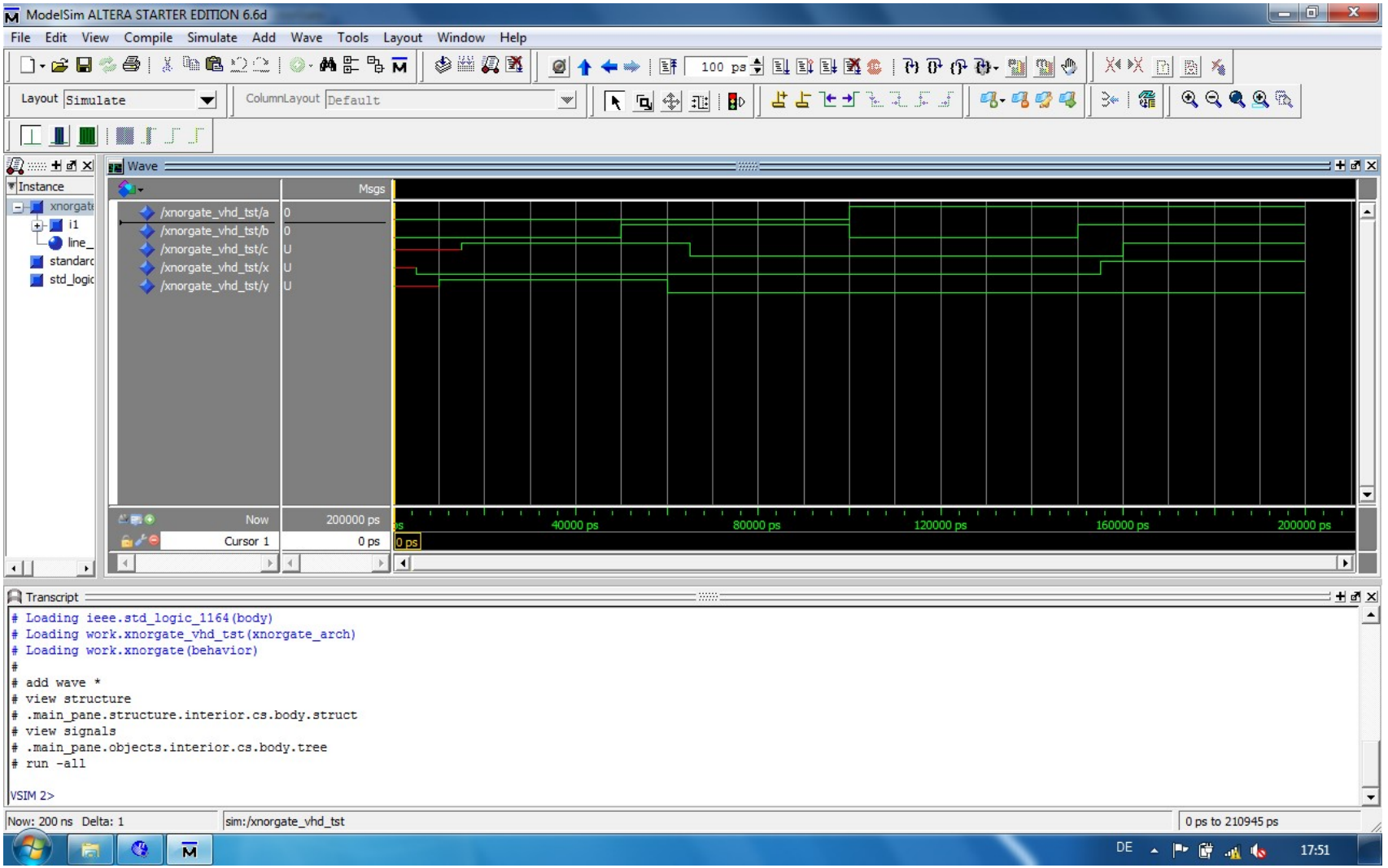
8-bit 4-to-1 multiplexer

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

entity mux_4_to_1_8bit is
   port
   ( i3, i2, i1, i0: in std_logic_vector (7 downto 0);
      sel: in std_logic_vector (1 downto 0);
      otp: out std_logic_vector (7 downto 0) );
end mux_4_to_1_8bit;

architecture behavior of mux_4_to_1_8bit is
begin
   process ( i3, i2, i1, i0, sel )
   begin
      if sel = “11” then otp <= i3;
      elsif sel = “10” then otp <= i2;
      elsif sel = “01” then otp <= i1;
      elsif sel = “00” then otp <= i0;
      end if;
   end process;
end behavior;
```

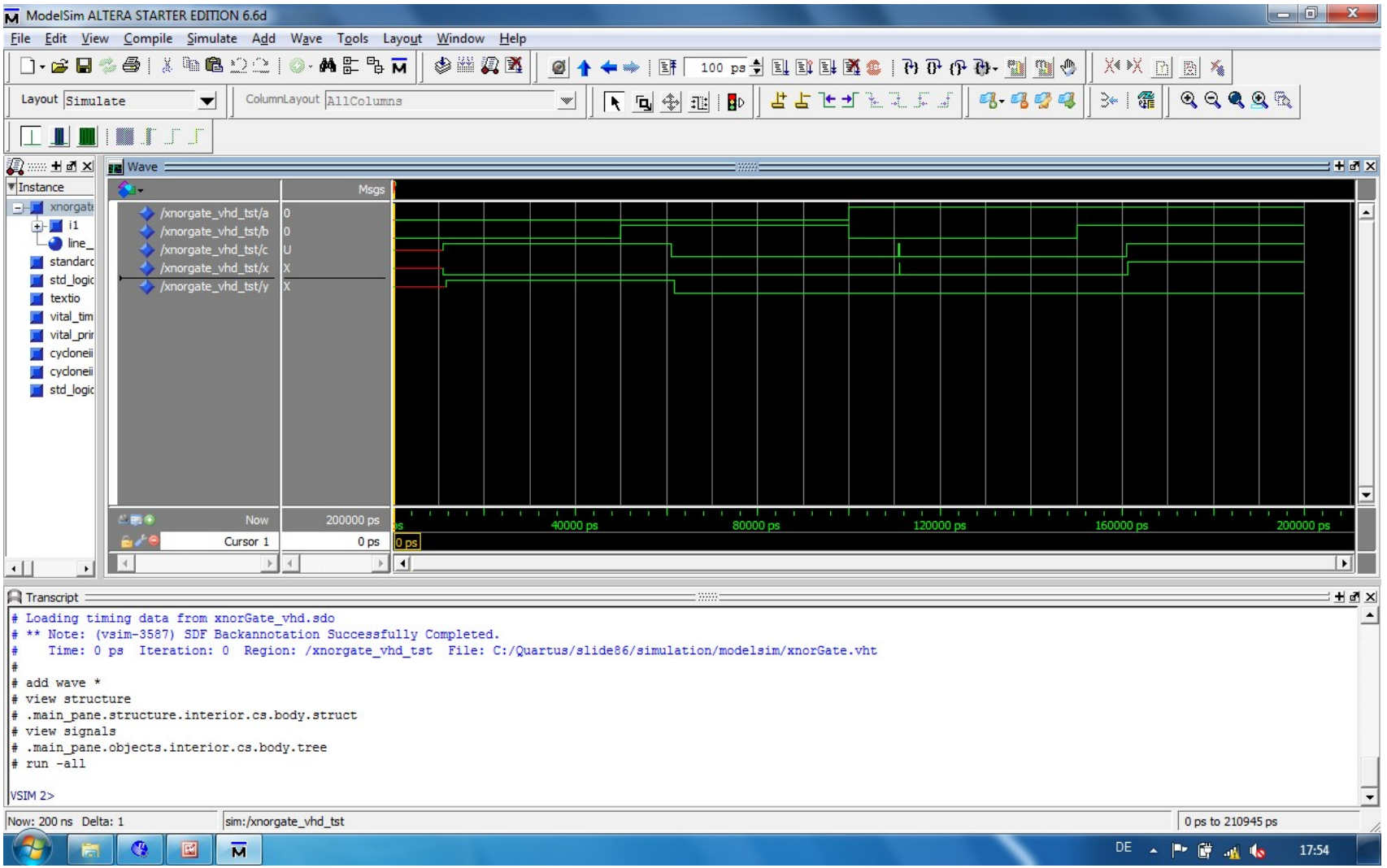
Entities & Architectures – Examples

XNOR Gate

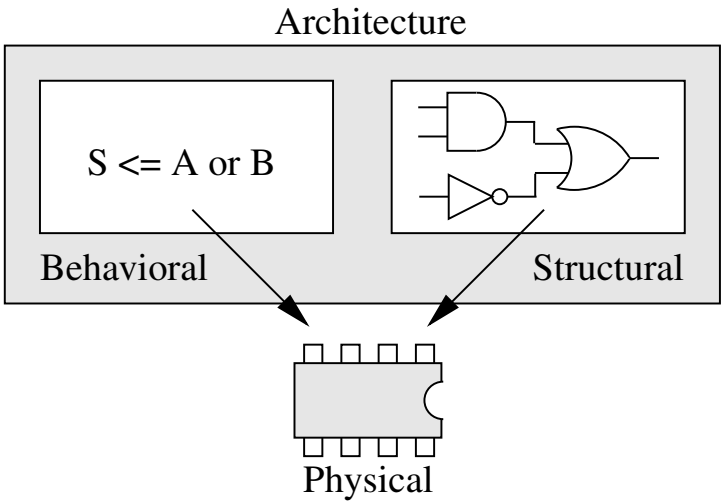
```
library ieee;
use ieee.std_logic_1164.all;

entity xnorGate is
   port ( a, b : in std_logic; c: out std_logic );
end xnorGate;

architecture behavior of xnorGate is
   signal x, y: std_logic;
begin
      x <= (a and b) after 5 ns;
      y <= ((not a) and (not b)) after 10 ns;
      c <= (x or y) after 5 ns;
end behavior;

-- The VHDL statements given above are executed in parallel!
-- Signal assignments are executed concurrently!
-- Changing the order does not matter!
```


ModelSim ALTERA STARTER EDITION 6.6d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Wave
Instance
Msgs
/xnorgate_vhd_tst/a
/xnorgate_vhd_tst/b
/xnorgate_vhd_tst/c
/xnorgate_vhd_tst/x
/xnorgate_vhd_tst/y
Now
200000 ps
Cursor 1
0 ps
40000 ps
80000 ps
120000 ps
160000 ps
200000 ps
Transcript
Loading ieee.std_logic_1164(body)
Loading work.xnorgate_vhd_tst(xnorgate_arch)
Loading work.xnorgate(behavior)
add wave *
view structure
.main_pane.structure.interior.cs.body.struct
view signals
.main_pane.objects.interior.cs.body.tree
run -all
VSIM 2>
Now: 200 ns Delta: 1
sim:/xnorgate_vhd_tst
0 ps to 210945 ps
17:51

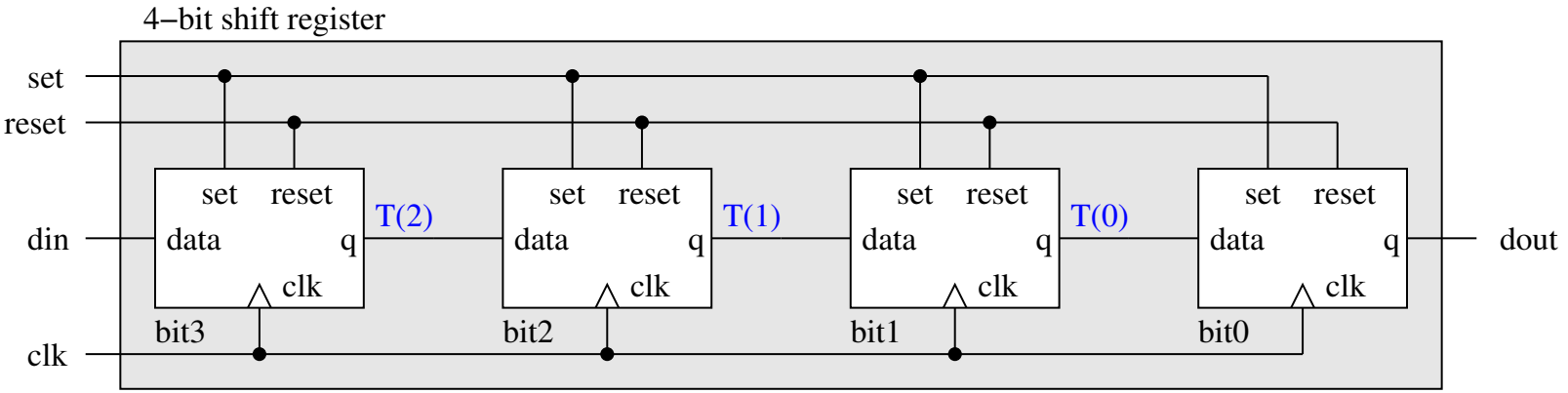
Entities & Architectures – Gate-Level Simulation



Entities & Architectures



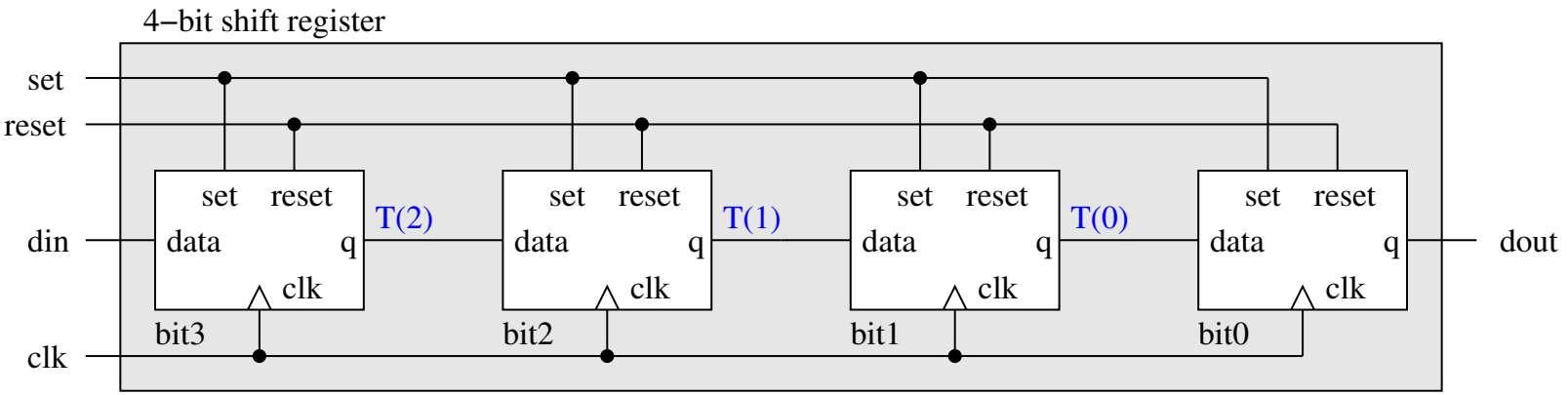
- The architecture can be structural, behavioral, or mixed
  - Structural descriptions use component instances
  - Behavioral descriptions describe the behavior of a digital system without defining its structure
- Structural hierarchy is essential for a compact and clear modeling of large systems
- In hierarchical designs, mixing structural and behavioral descriptions is mandatory (at least for the “leafs” in the hierarchy)



4-Bit Shift Register – Entity

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

entity shift_register_4bit is
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_4bit;
```

Entities & Architectures – Examples



4-Bit Shift Register – Architecture

```
architecture structural of shift_register_4bit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (2 downto 0);
begin
   bit3: flipflop port map ( din, clk, set, reset, T(2) );
   bit2: flipflop port map ( T(2), clk, set, reset, T(1) );
   bit1: flipflop port map ( T(1), clk, set, reset, T(0) );
   bit0: flipflop port map ( T(0), clk, set, reset, dout );
end structural;
```

Outline

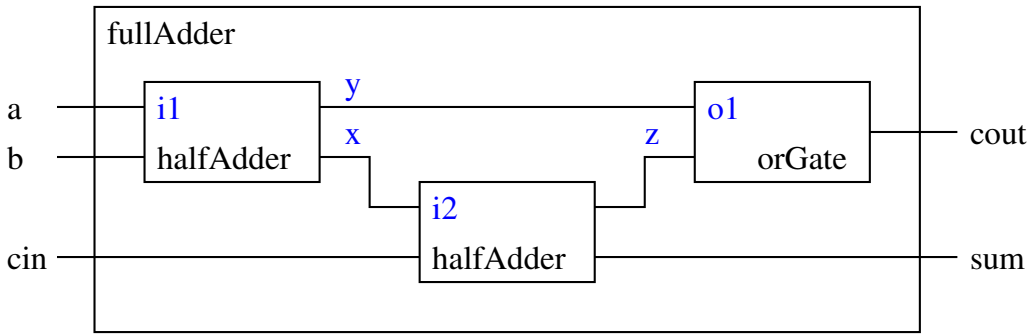
As demonstrated in the previous example, hierarchical VHDL designs combine “basic building blocks” (entities and their associated architectures) to form new designs

- **component** describes the interface of an entity that will be used as an instance in the current design
- **port map** describes the interconnection between components, which can be realized in two ways
  - Positional association
  - Explicit association

$\Rightarrow$ Architectures and their components can define a hierarchy of arbitrary depth!

Entity

```
entity fullAdder is
   port ( a, b, cin: in bit;
   sum, cout: out bit );
end fullAdder;
```

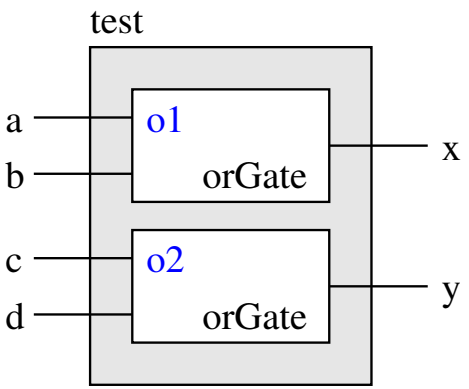


Architecture

```
architecture structure of fullAdder is
   component halfAdder port ( a, b: in bit; sum, cout: out bit ); end component;
   component orGate port ( a, b: in bit; c: out bit ); end component;
   signal x, y, z: bit;
begin
   i1: halfAdder port map ( a, b, x, y ); -- positional.
   i2: halfAdder port map ( sum => sum, a => x, cout => z, b => cin ); -- explicit.
   o1: orGate port map ( y, z, cout );
end structure;
```

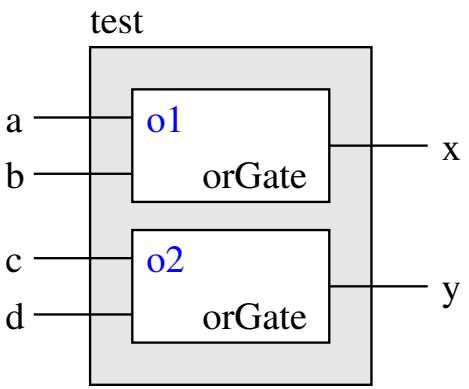
VHDL specification = entity + architecture(s)

- Each entity requires at least one architecture
- Multiple architectures can be defined for a single entity
  - Allows designers to use different implementations without changing the interface
  - Mapping an architecture to a particular component can be done via **configuration** statements
- By default, the most recently analyzed architecture is used



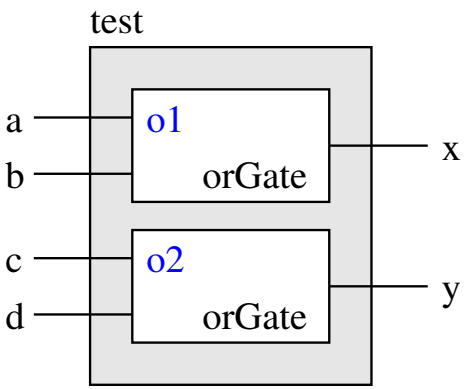
test

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

entity test is
   port ( a, b, c, d: in std_logic; x, y: out std_logic );
end test;

architecture structure of test is
   component orGate
      port ( a, b: in std_logic; c: out std_logic );
   end component;
begin
   o1: orGate port map ( a, b, x );
   o2: orGate port map ( c, d, y );
end structure;
```



orGate

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

entity orGate is
   port ( a, b: in std_logic; c: out std_logic );
end orGate;

architecture ok of orGate is
begin
   c <= a or b;
end ok;

architecture faulty of orGate is
begin
   c <= a and b;
end faulty;
```



orGate

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

entity orGate is
   port ( a, b: in std_logic; c: out std_logic );
end orGate;

architecture ok of orGate is
begin
   c <= a or b;
end ok;

architecture faulty of orGate is
begin
   c <= a and b;
end faulty; -- default configuration!
```

Components & Configurations – Example

test

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

entity test is
   port ( a, b, c, d: in std_logic; x, y: out std_logic );
end test;

architecture structure of test is
   component orGate
      port ( a, b: in std_logic; c: out std_logic );
   end component;
begin
   o1: orGate port map ( a, b, x );
   o2: orGate port map ( c, d, y );
end structure;
```

configuration 1

```
configuration cfg1 of test is
   for structure
      for all: orGate use entity work.orGate(ok); end for;
   end for;
end cfg1;
```

Components & Configurations – Example

test

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

entity test is
   port ( a, b, c, d: in std_logic; x, y: out std_logic );
end test;

architecture structure of test is
   component orGate
      port ( a, b: in std_logic; c: out std_logic );
   end component;
begin
   o1: orGate port map ( a, b, x );
   o2: orGate port map ( c, d, y );
end structure;
```

configuration 2 (not the intended functionality, but possible!)

```
configuration cfg2 of test is
   for structure
      for o1: orGate use entity work.orGate(ok); end for;
      for o2: orGate use entity work.orGate(faulty); end for;
   end for;
end cfg2;
```

Outline

Behavioral VHDL descriptions usually consist of a set of concurrently executed processes:

```
[ process_label: ] process [ ( sensitivity_list ) ] [ is ]
   -- constants & variables used inside the process have to be defined here.
begin
   -- list of sequential statements such as:
      -- signal & variable assignments
      -- if statement
      -- case statement
      -- loop statement
      -- null statement
      -- wait statement
      -- ...
end process [ process_label ];
```

- While all processes of a design are executed concurrently, the statements within a single process are executed sequentially in the order they are written
- No hierarchy of processes allowed in VHDL
- Only simple signal assignments allowed inside a process
- When a simulation starts, each process will be executed once
- Afterwards, processes will be reactivated according to events regarding signals of the associated sensitivity list
- Processes are mainly used to describe sequential circuits (e.g. the DFF shown on Slide 84), but not necessarily

Signal assignments outside processes can be viewed as implicit processes!

Entity

```
entity halfAdder is
   port ( a, b: in bit; sum, cout: out bit );
end halfAdder;
```

Architecture arch1

```
architecture arch1 of halfAdder is
begin
   cout <= a and b;
   sum <= a xor b;
end arch1;
```

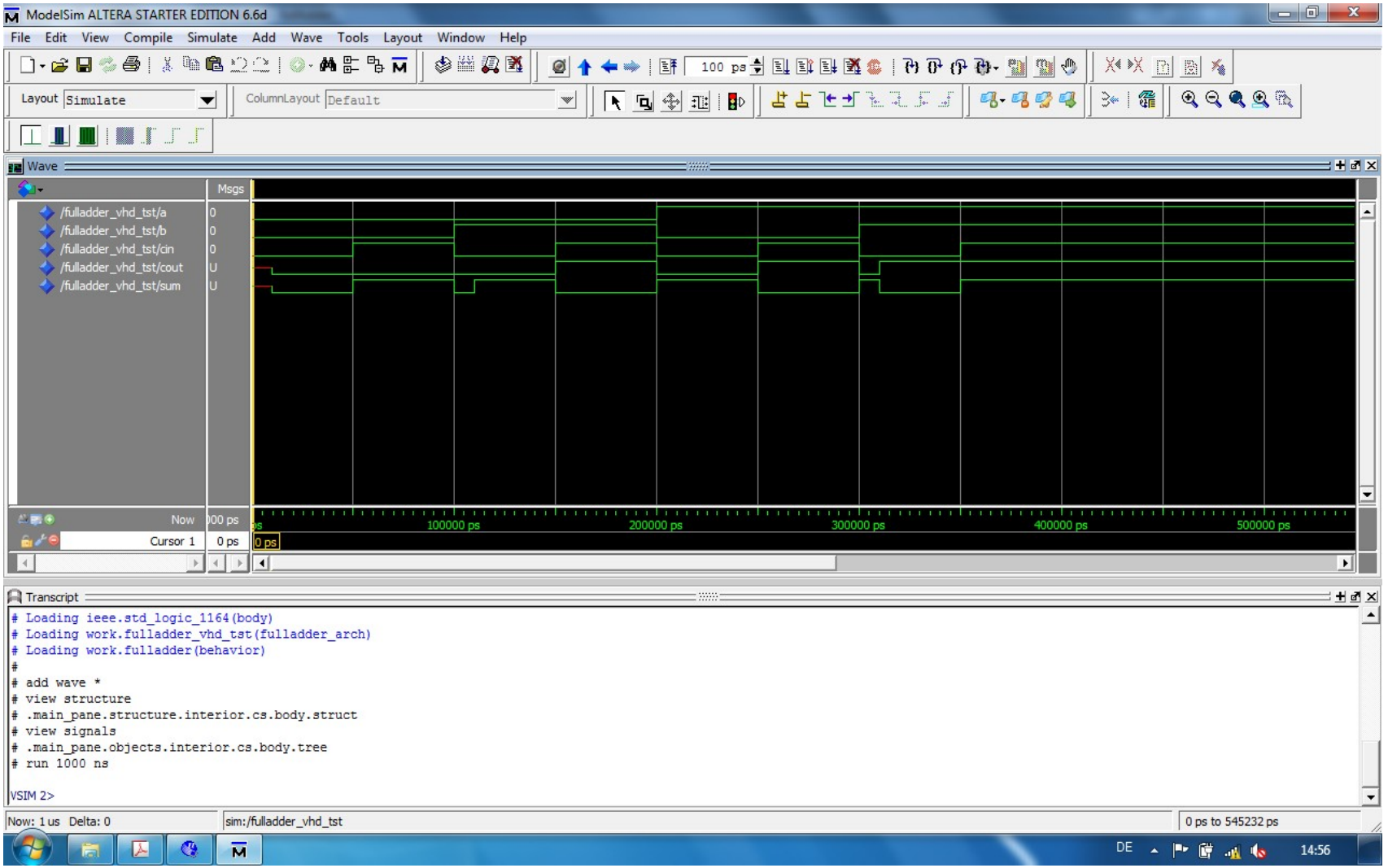
Architecture arch2

```
architecture arch2 of halfAdder is
begin
   process ( a, b )
   begin
      cout <= a and b;
   end process;
   process ( a, b )
   begin
      sum <= a xor b;
   end process;
end arch2;
```

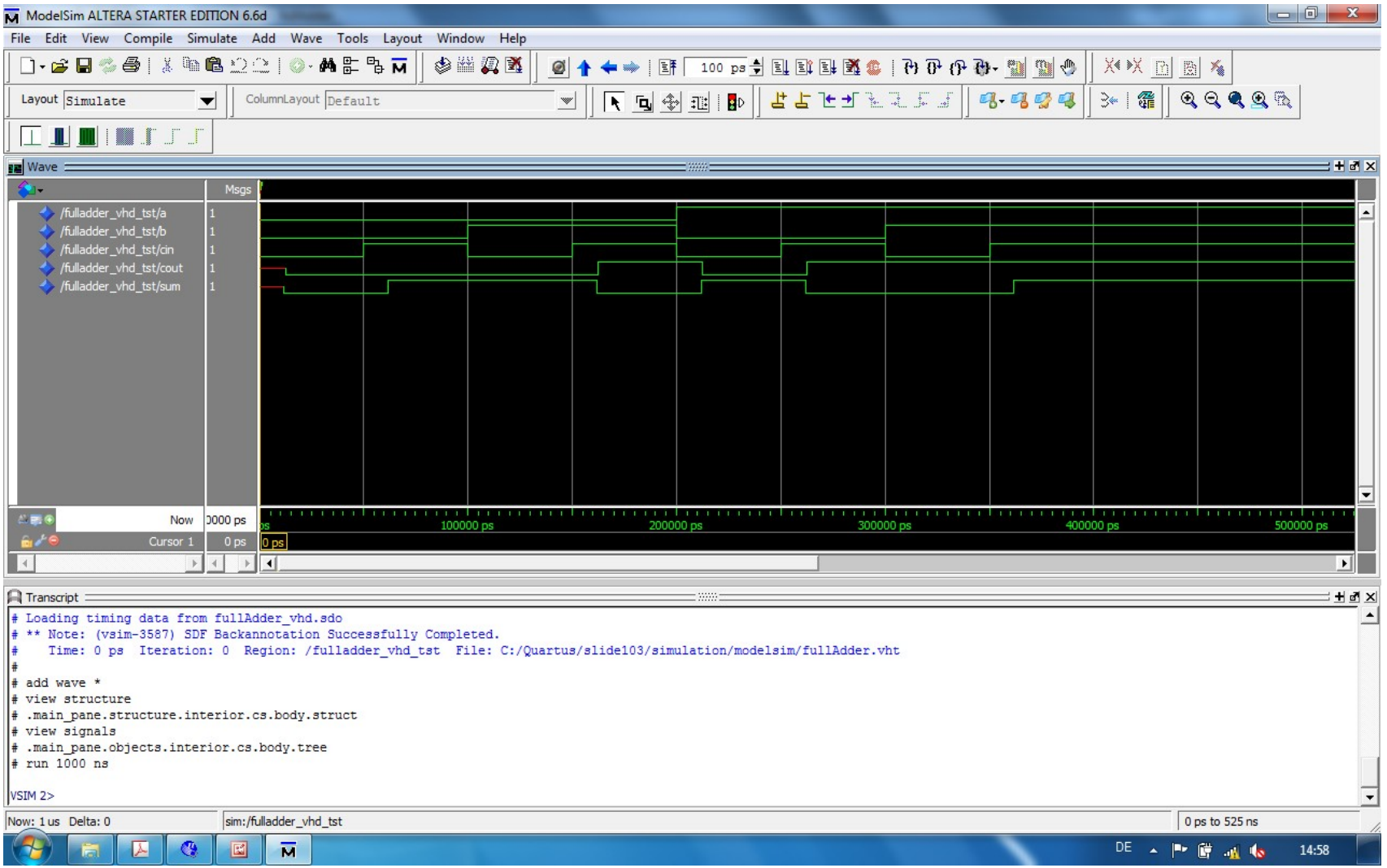
$\Rightarrow$ Again, arch1 and arch2 describe the same behavior!

Full Adder

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

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

architecture behavior of fullAdder is
   signal i1, i2: std_logic;
begin

   P1: process ( a, b ) -- P1 defines the 1st half adder.
   begin
      i1 <= a xor b after 10 ns;
      i2 <= a and b after 10 ns;
   end process;

   P2: process ( i1, i2, cin ) -- P1 defines the 2nd half adder plus the OR gate.
   begin
      sum <= i1 xor cin;
      cout <= i2 or (i1 and cin);
   end process;

end behavior;
```


ModelSim ALTERA STARTER EDITION 6.6d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Wave
Msgs
/fulladder_vhd_tst/a
/fulladder_vhd_tst/b
/fulladder_vhd_tst/cin
/fulladder_vhd_tst/cout
/fulladder_vhd_tst/sum
100000 ps
200000 ps
300000 ps
400000 ps
500000 ps
Cursor 1
Transcript
Loading ieee.std_logic_1164(body)
Loading work.fulladder_vhd_tst(fulladder_arch)
Loading work.fulladder(behavior)
add wave *
view structure
.main_pane.structure.interior.cs.body.struct
view signals
.main_pane.objects.interior.cs.body.tree
run 1000 ns
VSIM 2>
Now: 1 us Delta: 0
sim:/fulladder_vhd_tst
0 ps to 545232 ps
14:56

Processes – Gate-Level Simulation



Clock generator (not synthesizable, but often used in testbenches!)

```
entity clock is
   port ( clk: out bit );
end clock;

architecture behavior of clock is
begin
   process
   begin
      clk <= '0';
      wait for 10 ns;
      clk <= '1';
      wait for 10 ns;
   end process;
end behavior;
```

Testbench for the full adder using clocks.

```
...

process
begin
   a <= '0'; wait for 160 ns;
   a <= '1'; wait for 160 ns;
end process;

process
begin
   b <= '0'; wait for 80 ns;
   b <= '1'; wait for 80 ns;
end process;

process
begin
   cin <= '0'; wait for 40 ns;
   cin <= '1'; wait for 40 ns;
end process;

...
```

Sensitivity lists are a shorthand for a single **wait on**-statement at the end of the process body

Sensitivity list

```
process ( a, b )
begin
   c <= a and b;
end process;
```

is equivalent to

wait on-Statement

```
process
begin
   c <= a and b;
   wait on a, b;
end process;
```

- VHDL supports different types of **wait**-statements
  - **wait on** signal_list
    - Wait until at least one of the signals in signal_list changes
    - Example: **wait on** a, b;
  - **wait until** condition
    - Wait until condition is met
    - Example: **wait until** (clk'event **and** clk = '1');
  - **wait for** duration
    - Wait for specified amount of time
    - Example: **wait for** 25 ns;
  - **wait**
    - Suspend indefinitely
- If no sensitivity list is specified, a **wait**-statement has to be included to ensure that the process will stop
- It is not allowed to have both sensitivity lists and **wait**-statements

Other “sequential” statements that can be used inside processes

- **if**
- **case**
- **loop**, **while-loop**, **for-loop**
- **null**

Except for **if** and **case**...

- None of the above statements are synthesizable
- Most of them are used for testing

if-statement

- Comparable to if-statements in other programming languages

8-bit 4-to-1 multiplexer

```
...
process ( i3, i2, i1, i0, sel )
begin
   if sel = "11" then otp <= i3;
   elsif sel = "10" then otp <= i2;
   elsif sel = "01" then otp <= i1;
   elsif sel = "00" then otp <= i0;
   end if;
end process;
...
```

case-statement

- Executes one of several sequences of statements, based on a single expression to be evaluated
- Choices can be
  - single values, e.g. . . . **when** “01”
  - a range, e.g. . . . **when** 51 **to** 60
  - combined choices, e.g. . . . **when** “00” **or** “01” **or** “11”
- No two choices are allowed to overlap
- If there is no **when others**-statement, all possible values of the expression have to be covered by the set of choices

Temperature

```
entity temperature is
   port ( val: in integer range 0 to 100; a, b, c, d: out bit );
end temperature;

architecture behavior of temperature is
begin
   process ( val )
   begin
      a <= '0'; b <= '0'; c <= '0'; d <= '0';
      case val is
         when 51 to 60 => a <= '1';
         when 61 to 70 => b <= '1';
         when 71 to 80 => c <= '1';
         when others => d <= '1';
      end case;
   end process;
end behavior;
```

loop-statements

- **loop**
  - Requires a **wait**-statement
  - Executed continuously until an **exit** is encountered
- **while-loop**
  - Requires a **wait**-statement
  - Executed as long as the “iteration condition” is true
- **for-loop**
  - Uses an integer iteration scheme, determining the number of iterations
- All three variants can be combined with…
  - **next**: terminates the rest of the current loop iteration; execution will proceed to the next loop iteration
  - **exit**: terminates the loop entirely; continues with the next statement after the exited loop

Testbench for the full adder using **loop**

```
...

process
   variable cnt: std_logic := '1';
   variable num: integer := 0;
begin
   loop
      if cnt = '1' then cnt := '0'; else cnt := '1'; end if;
      num := num + 1;
      a <= cnt;
      wait for 160 ns;
      exit when num > 9;
   end loop;
end process;

...
```

Testbench for the full adder using **while-loop**

```
...

process
   variable cnt: std_logic := '1';
   variable num: integer := 0;
begin
   while num <= 9 loop
      if cnt = '1' then cnt := '0'; else cnt := '1'; end if;
      num := num + 1;
      b <= cnt;
      wait for 80 ns;
   end loop;
end process;

...
```

Testbench for the full adder using **for-loop**

```
...

process
   variable cnt: std_logic := '1';
begin
   for num in 1 to 9 loop
      if cnt = '1' then cnt := '0'; else cnt := '1'; end if;
      cin <= cnt;
      wait for 40 ns;
   end loop;
end process;

...
```

null-statement

- States that no action will occur
- Useful as part of **case**-statements where all choices must be covered, even if some of them can be ignored

Example

```
-- Assumptions:
-- 1) 'val' is an integer, ranging from 0 to 7.
-- 2) If 'val' is equal to 0|1|3|4|6|7, nothing has to be done.
-- 3) 'c' is an output, while 'a' and 'b' are inputs.
...
process ( val )
begin
   case val is
      when 2 or 5 => c <= a xor b;
      when others => null;
   end case;
end process;
...
```

Variable vs. signal assignments inside processes

- A variable changes instantaneously when the corresponding variable assignment is executed (without any time delay)
- A signal changes a delay after the assignment expression is executed (user-specified or delta delay)
  - $\Rightarrow$ All signal assignments inside a process are sequentially evaluated until the process is stopped and are performed in parallel afterwards

Example of a process using variables

```
architecture var of example is
   signal trigger: integer := 0;
   signal rst: integer := 0;
begin
   process
      variable var1: integer := 1;
      variable var2: integer := 2;
      variable var3: integer := 3;
   begin
      wait on trigger;
      var1 := var2;
      var2 := var1 + var3;
      var3 := var2;
      rst <= var1 + var2 + var3;
   end process;
end var;

-- All variables are computed sequentially and their values are updated
-- instantaneously after the 'trigger' signal arrives. Next, 'rst' is computed using
-- the new values of 'var1..3' and is updated a delay delta after 'trigger' arrives.
-- var1 = ?, var2 = ?, var3 = ?, rst = ?
```

Example of a process using variables

```
architecture var of example is
   signal trigger: integer := 0;
   signal rst: integer := 0;
begin
   process
      variable var1: integer := 1;
      variable var2: integer := 2;
      variable var3: integer := 3;
   begin
      wait on trigger;
      var1 := var2;
      var2 := var1 + var3;
      var3 := var2;
      rst <= var1 + var2 + var3;
   end process;
end var;

-- All variables are computed sequentially and their values are updated
-- instantaneously after the 'trigger' signal arrives. Next, 'rst' is computed using
-- the new values of 'var1..3' and is updated a delay delta after 'trigger' arrives.
-- var1 = 2, var2 = 5, var3 = 5, rst = 12
```

Example of a process using signals

```
architecture sig of example is
   signal trigger: integer := 0;
   signal rst: integer := 0;
   signal sig1: integer := 1;
   signal sig2: integer := 2;
   signal sig3: integer := 3;
begin
   process
   begin
      wait on trigger;
      sig1 <= sig2;
      sig2 <= sig1 + sig3;
      sig3 <= sig2;
      rst <= sig1 + sig2 + sig3;
   end process;
end sig;

-- All signal assignments will be executed at the same time (event on 'trigger'),
-- using the old values of 'sig1', 'sig2', 'sig3', but updated a delta time after
-- 'trigger' has arrived.
-- sig1 = ?, sig2 = ?, sig3 = ?, rst = ?
```

Example of a process using signals

```
architecture sig of example is
   signal trigger: integer := 0;
   signal rst: integer := 0;
   signal sig1: integer := 1;
   signal sig2: integer := 2;
   signal sig3: integer := 3;
begin
   process
   begin
      wait on trigger;
      sig1 <= sig2;
      sig2 <= sig1 + sig3;
      sig3 <= sig2;
      rst <= sig1 + sig2 + sig3;
   end process;
end sig;

-- All signal assignments will be executed at the same time (event on 'trigger'),
-- using the old values of 'sig1', 'sig2', 'sig3', but updated a delta time after
-- 'trigger' has arrived.
-- sig1 = 2, sig2 = 4, sig3 = 2, rst = 6
```

Outline

Parameterized Hardware Descriptions

UNI FREIBURG

VHDL provides a mechanism to easily describe parameterized hardware components

- Generic entities by including lists to define the **generic** constants that parameterize the entity
- Iterative component instantiation with **for-generate**
- Conditional component instantiation with **if-generate**

4-Bit Shift Register

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

entity shift_register_4bit is
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_4bit;

architecture structural of shift_register_4bit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (2 downto 0);
begin
   bit3: flipflop port map ( din, clk, set, reset, T(2) );
   bit2: flipflop port map ( T(2), clk, set, reset, T(1) );
   bit1: flipflop port map ( T(1), clk, set, reset, T(0) );
   bit0: flipflop port map ( T(0), clk, set, reset, dout );
end structural;
```

n-Bit Shift Register

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

entity shift_register_nbit is
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_nbit;

architecture structural of shift_register_nbit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (2 downto 0);
begin
   bit3: flipflop port map ( din, clk, set, reset, T(2) );
   bit2: flipflop port map ( T(2), clk, set, reset, T(1) );
   bit1: flipflop port map ( T(1), clk, set, reset, T(0) );
   bit0: flipflop port map ( T(0), clk, set, reset, dout );
end structural;
```

n-Bit Shift Register

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

entity shift_register_nbit is
   generic ( n: positive );
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_nbit;

architecture structural of shift_register_nbit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (2 downto 0);
begin
   bit3: flipflop port map ( din, clk, set, reset, T(2) );
   bit2: flipflop port map ( T(2), clk, set, reset, T(1) );
   bit1: flipflop port map ( T(1), clk, set, reset, T(0) );
   bit0: flipflop port map ( T(0), clk, set, reset, dout );
end structural;
```

n-Bit Shift Register

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

entity shift_register_nbit is
   generic ( n: positive );
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_nbit;

architecture structural of shift_register_nbit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (n-2 downto 0);
begin
   bit3: flipflop port map ( din, clk, set, reset, T(2) );
   bit2: flipflop port map ( T(2), clk, set, reset, T(1) );
   bit1: flipflop port map ( T(1), clk, set, reset, T(0) );
   bit0: flipflop port map ( T(0), clk, set, reset, dout );
end structural;
```

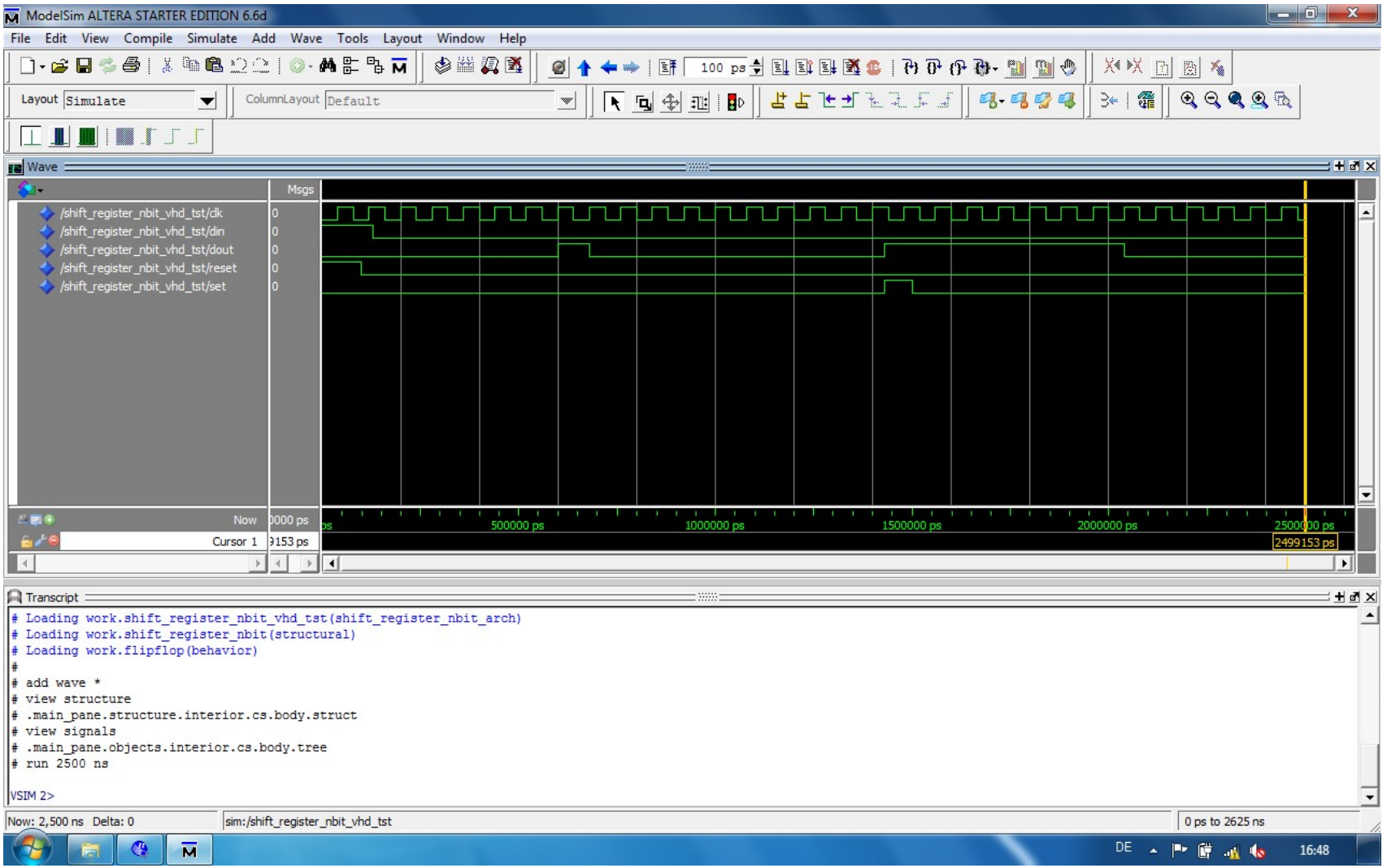
n-Bit Shift Register

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

entity shift_register_nbit is
   generic ( n: positive );
   port ( din, clk, set, reset: in std_logic; dout: out std_logic );
end shift_register_nbit;

architecture structural of shift_register_nbit is
   component flipflop
      port ( data, clk, set, reset: in std_logic; q: out std_logic );
   end component;
   signal T: std_logic_vector (n-2 downto 0);
begin
   label1: for i in n-1 downto 0 generate
      label2: if i = n-1 generate
         bitL: flipflop port map ( din, clk, set, reset, T(n-2) ); end generate;
      label3: if i > 0 and i < n-1 generate
         bitM: flipflop port map ( T(i), clk, set, reset, T(i-1) ); end generate;
      label4: if i = 0 generate
         bitR: flipflop port map ( T(0), clk, set, reset, dout ); end generate;
   end generate;
end structural;
```

Component instantiation (in this example within a testbench)

```
...

architecture arch of shift_register_nbit_vhd_tst is
   ...
   component shift_register_nbit
      generic ( n : positive );
      port ( din, clk, set, reset: in std_logic; dout: out std_logic );
   end component;
begin
   i1: shift_register_nbit
      generic map ( 7 ) -- Note, here is no semicolon!
      port map ( din, clk, set, reset, dout );
   ...
end arch;
```

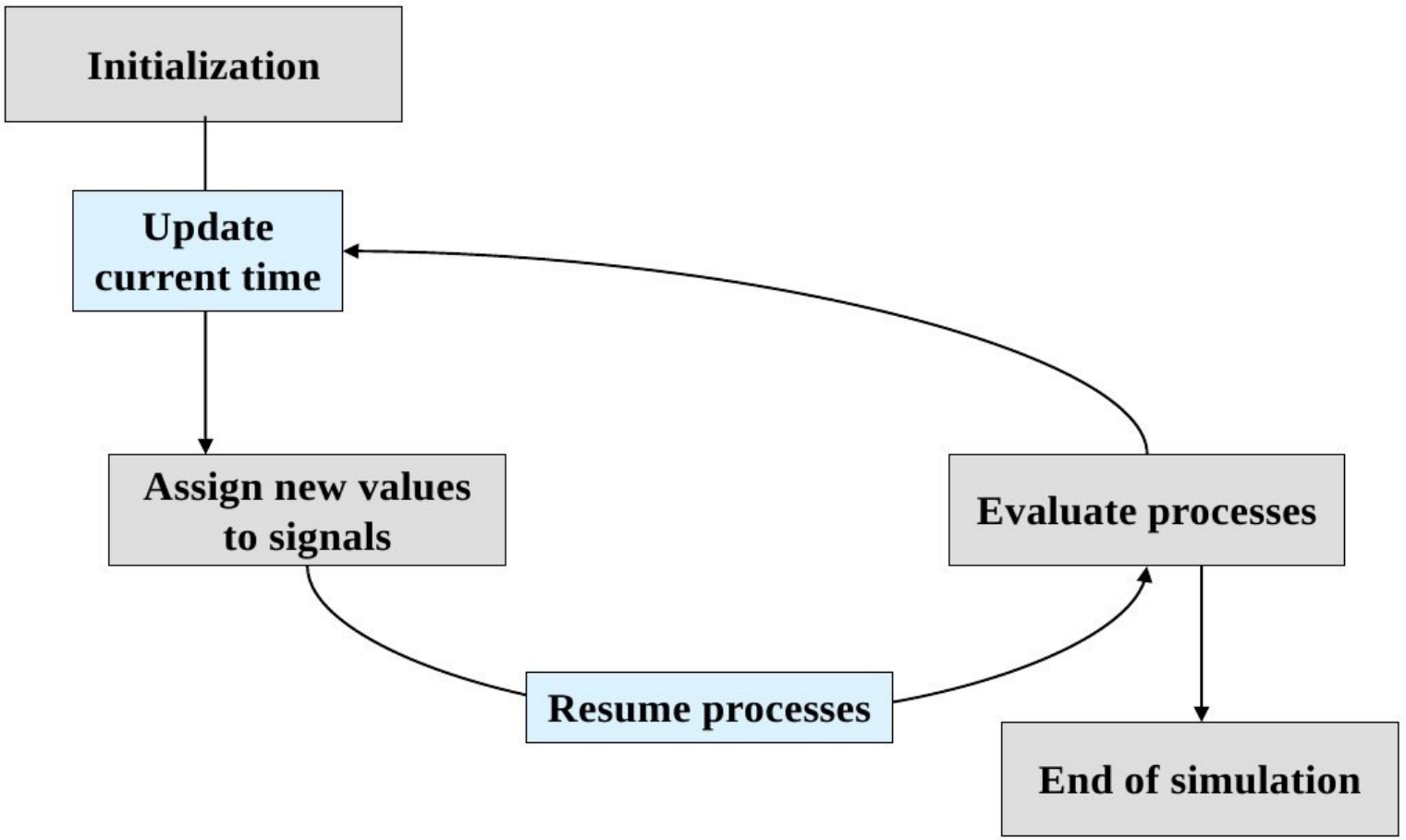
Parameterized Hardware Descriptions – RTL Sim.



Outline

Overview

- To define the semantics of VHDL, one can assume that the structural hierarchy of a design “is flattened” into a behavioral description
- Execution of VHDL models consists of an initialization phase followed by the repetitive execution of process statements
- A (simulation) step consists of two phases
  - Signal assignments
  - Execution of processes


Initialization
Update current time
Assign new values to signals
Evaluate processes
Resume processes
End of simulation

Transaction list

- For signal assignments
- Entries of the form (s,v,t), meaning “signal s is set to value v at time t”

Process activation list

- For reactivating processes
- Entries of the form (p, t), meaning “process p resumes at time t”

Initialization

- At the beginning of the simulation, the current time t_{curr} is assumed to be 0 ns
- An initial value is assigned to each signal
  - Default value (if specified)
  - First value in enumeration based types
  - The initial value is assumed to have been the value of the particular signal for an infinite length of time prior to the start of the simulation

Initialization

- During the initialization phase each process is executed once (until it suspends)
- Signal assignments stated inside processes are collected in the transaction list (the statements get evaluated, but the corresponding signals are not updated immediately!)
- After the initialization, the time of the next simulation cycle is calculated: $t_{next} = \min(t_1, t_2, t_3)$ with
  1. t_1: time'high (end of simulation time)
  2. t_2: earliest time in transaction list (if non-empty)
  3. t_3: earliest time in process activation list (if non-empty)

Example

```
architecture behavior of semantics is
   signal a : std_logic := '0';
   signal b : std_logic := '1';
   signal c : std_logic := '1';
   signal d : std_logic := '0';
begin
   swap1: process ( a, b )
   begin
      a <= b after 10 ns;
      b <= a after 10 ns;
   end process swap1;
   swap2: process
   begin
      c <= d;
      d <= c;
      wait for 15 ns;
   end process swap2;
end behavior;
```

"Update current time" / "Assign new values to signals"

- Each simulation cycle starts with setting the current time to the next time at which changes must be considered: $t_{curr} = t_{next}$
- For all (s,v,t_{curr}) in the transaction list
  - Remove (s,v,t_{curr}) from the list
  - Set s to value v
- For all processes p waiting on signal s (**wait on**)
  - Insert (p,t_{curr}) into the process activation list
  - Similarly, if condition of a **wait until**-statement changes its value

Example

```
architecture behavior of semantics is
   signal a : std_logic := '0';
   signal b : std_logic := '1';
   signal c : std_logic := '1';
   signal d : std_logic := '0';
begin
   swap1: process ( a, b )
   begin
      a <= b after 10 ns;
      b <= a after 10 ns;
   end process swap1;
   swap2: process
   begin
      c <= d;
      d <= c;
      wait for 15 ns;
   end process swap2;
end behavior;
```

"Resume processes" / "Evaluate processes"

- Resume all processes p with entries (p,t_{curr}) in the process activation list
- Execute all activated processes "in parallel", i.e. in arbitrary order
- Signal assignments are collected in the transaction list (again, the corresponding signals are not updated immediately!)
  - s <= a **and** b; $\Rightarrow$ insert (s,$a \wedge b$,t_{curr})
  - clk <= '0' **after** 10ns; $\Rightarrow$ insert (clk,0,t_{curr}+10ns)

"Resume processes" / "Evaluate processes"

- Processes are executed until a **wait**-statement is encountered
- If process p stops at a **wait for**-statement, then update the process activation list
  - "p stops at **wait for** 20 ns" $\Rightarrow$ insert (p,t_{curr}+20ns)
- When all processes have stopped, the time of the next simulation cycle is calculated: $t_{next} = \min(t_1, t_2, t_3)$ with
  1. t_1: time'high (end of simulation time)
  2. t_2: earliest time in transaction list (if non-empty)
  3. t_3: earliest time in process activation list (if non-empty)
- Simulation terminates, if t_{next} is equal to time'high and both the transaction list and the process activation list are empty

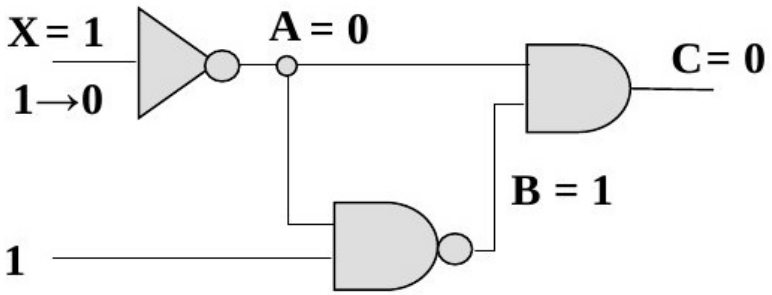
Example

```
architecture behavior of semantics is
   signal a : std_logic := '0';
   signal b : std_logic := '1';
   signal c : std_logic := '1';
   signal d : std_logic := '0';
begin
   swap1: process ( a, b )
   begin
      a <= b after 10 ns;
      b <= a after 10 ns;
   end process swap1;
   swap2: process
   begin
      c <= d;
      d <= c;
      wait for 15 ns;
   end process swap2;
end behavior;
```

Delta delay

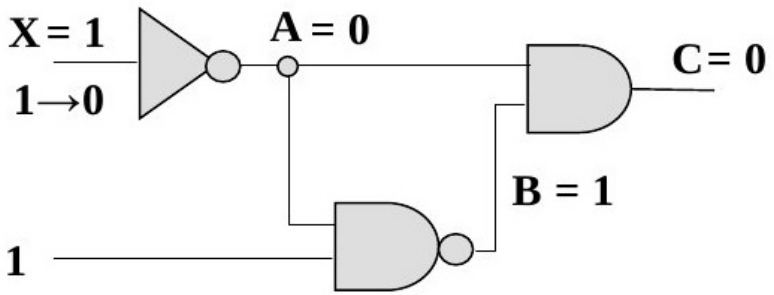
- Time does not necessarily proceed between two (simulation) steps
- Several (potentially an infinite number of) steps can take place at the same time t_{curr}

$\Rightarrow$ Signal assignments which take place at the same time in two consecutive steps are separated by one delta delay



Simulation time does not proceed due to delta delays!

Current time	Delta delay	Event
0 ns	1	-- evaluation of inverter -- (A, 1, 0 ns)



Simulation time does not proceed due to delta delays!

Current time	Delta delay	Event
0 ns	1	-- evaluation of inverter -- (A, 1, 0 ns)
	2	-- evaluation of AND and NAND -- (B, 0, 0ns), (C, 1, 0ns)
	3	-- evaluation of AND -- (C, 0, 0ns)

Issues not covered in this section

- Resolution functions
  - Necessary to deal with “write-write-conflicts”
- Inertial delay model
  - Usually, physical gates absorb short pulses
  - Default model
- Transport delay model
  - Transmits all pulses at the inputs ideally

Outline

Evaluation of VHDL

- Powerful hardware description language
- Hierarchical specifications of concurrent systems
- No nested processes
- No specification of non-functional properties
- No object-orientation
- Static number of processes
- Complicated simulation semantics
- May be too low level for an initial specification of very large digital systems
- Mainly used for hardware generation / synthesis / rapid prototyping (not all VHDL statements are synthesizable!)