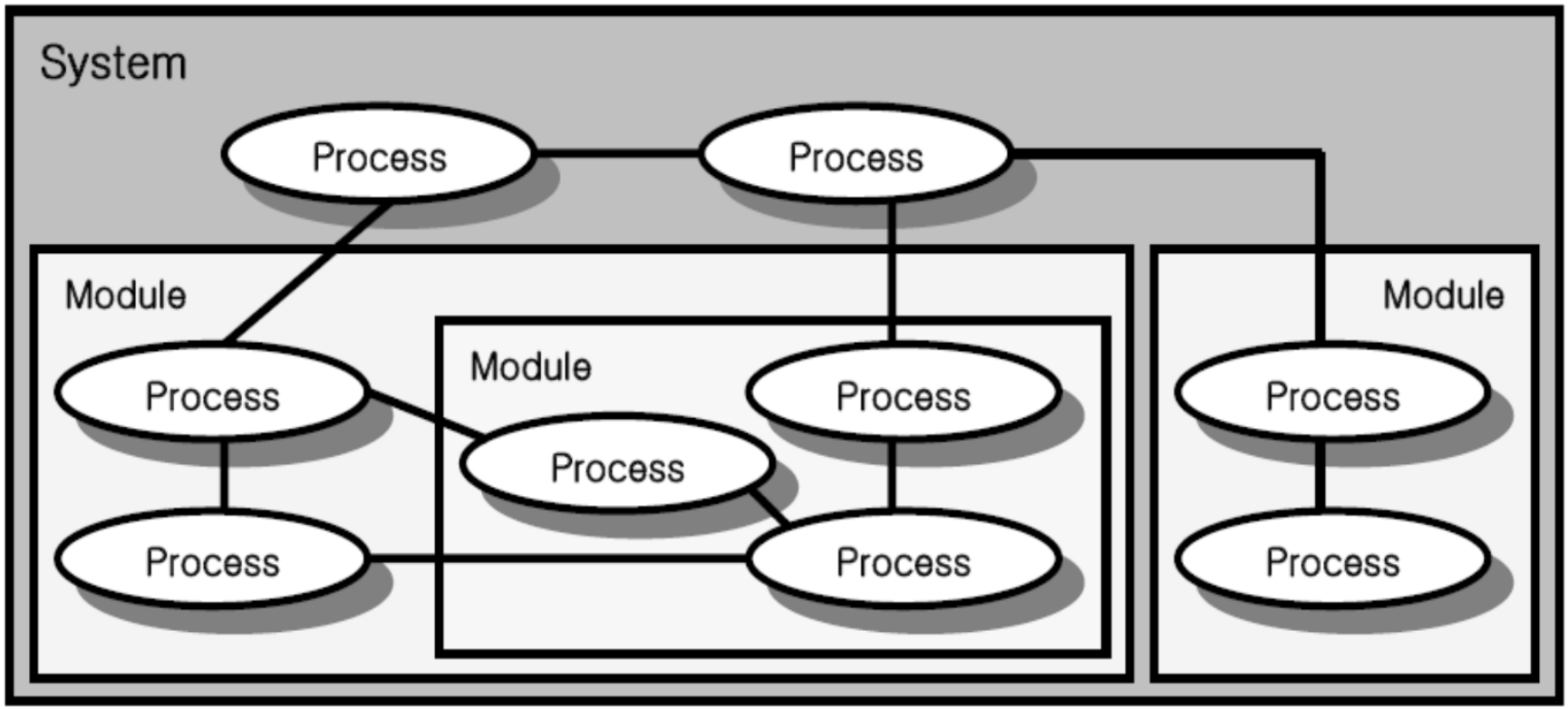
Turun yliopisto
University of Turku

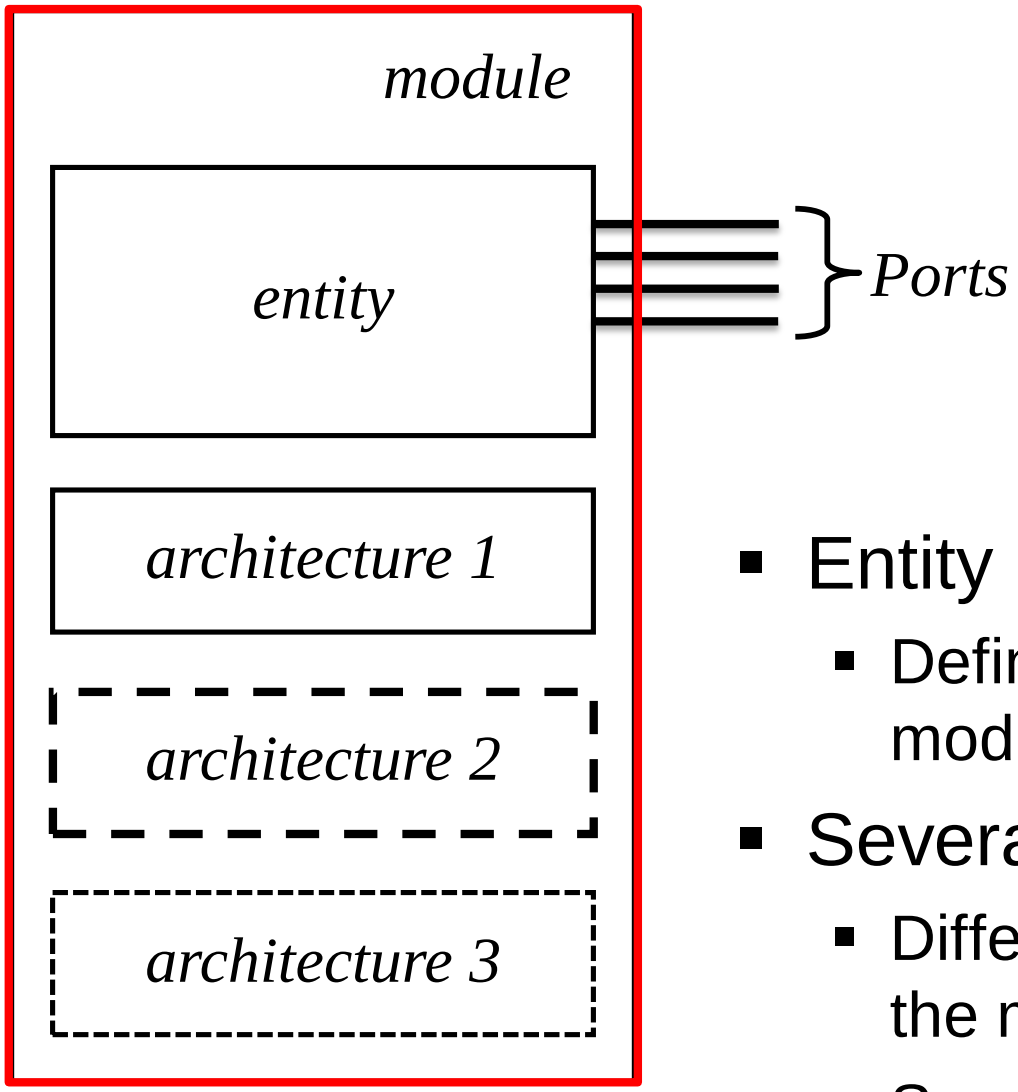
HDL Based Design
ME620138

2016

What is the wood?



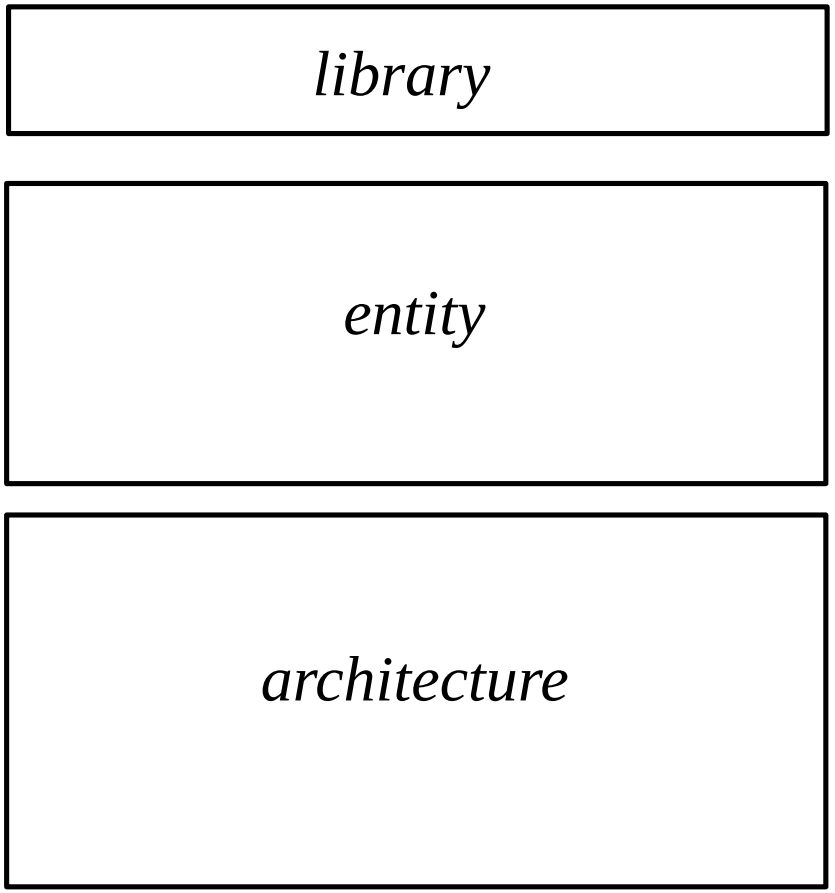
Design Entity



- Entity
  - Defines the interface of the module
- Several Architecture
  - Different implementations of the model
  - Same entity

Libraries and Packages

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity half_adder is
port(
       x,y: in std_logic;
       sum, carry: out std_logic);
end half_adder;

architecture myadder of half_adder is
begin
     sum <= x xor y;
     carry <= x and y;
end myadder;
```



VHDL Coding Styles

Structural	Dataflow	Behavioural
• Component instantiations	• Concurrents signal assignments	• Process statements

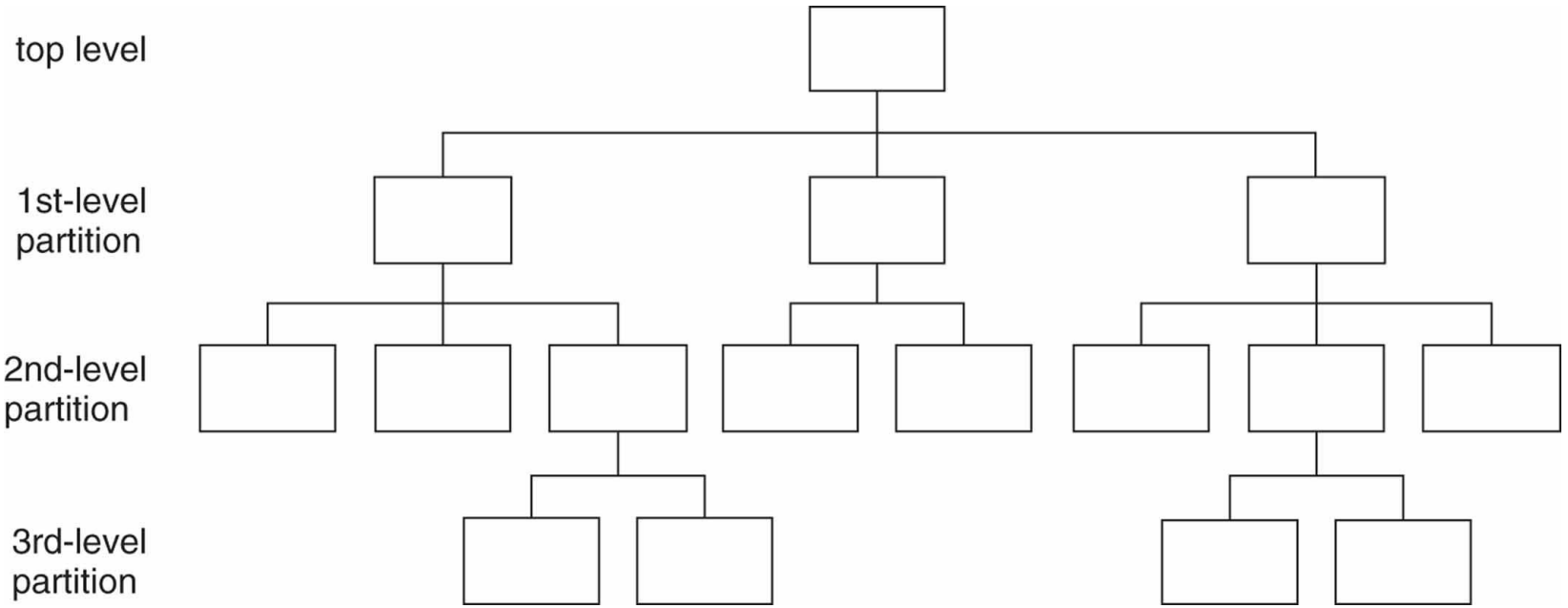
- Depending on the system being desinged, one style may be the most intuitive to use

All VHDL processes execute concurrently

Concurrent statements are executed only when input variables'/signals' values changes

Hierarchical Design

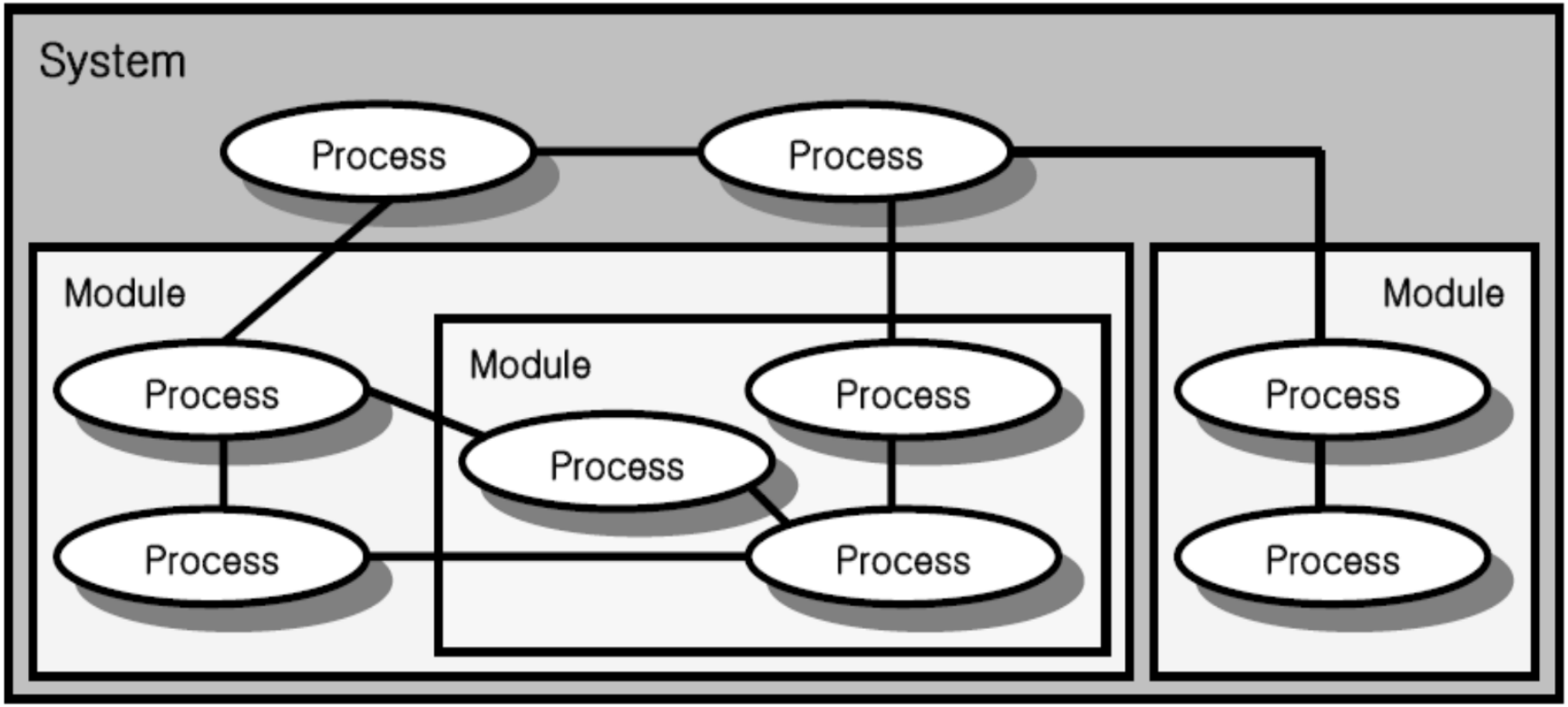
- Top-down design



- Each module may itself be partitioned to further reduce its complexity

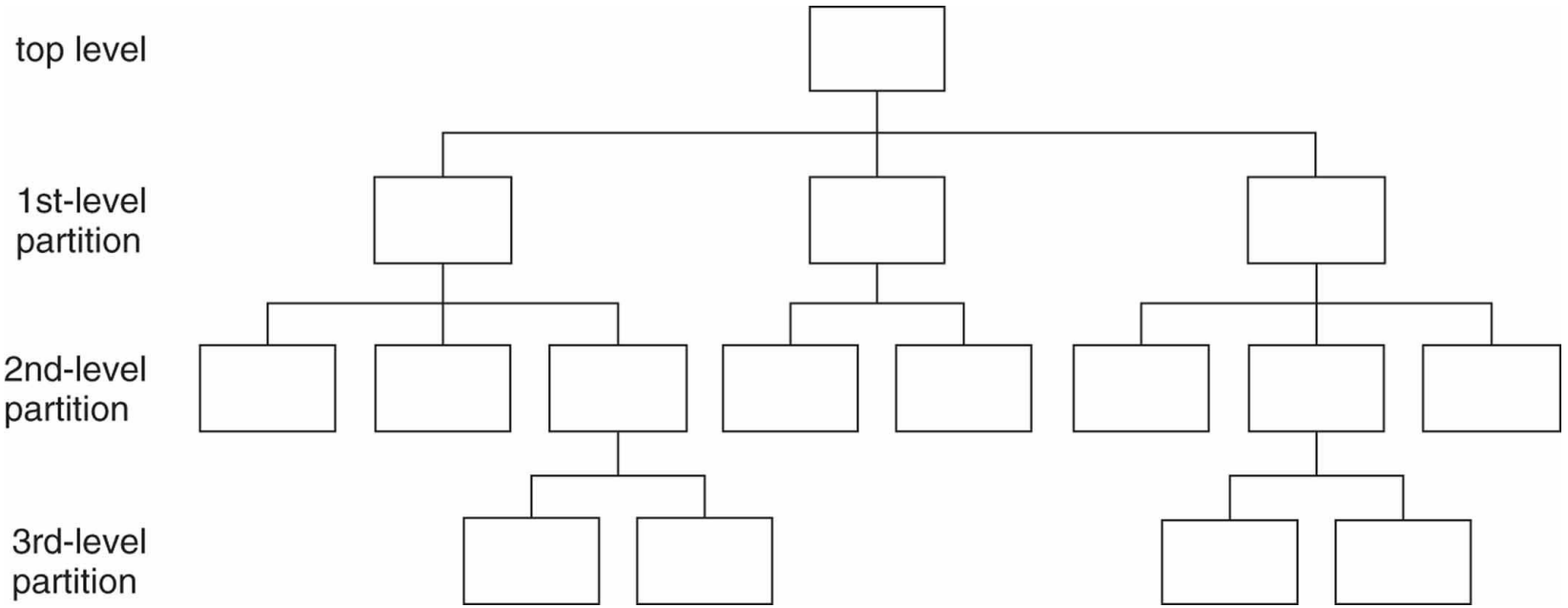
MODULAR DESIGN AND HIERARCHY

Modular Design



Hierarchical Design

- Top-down design



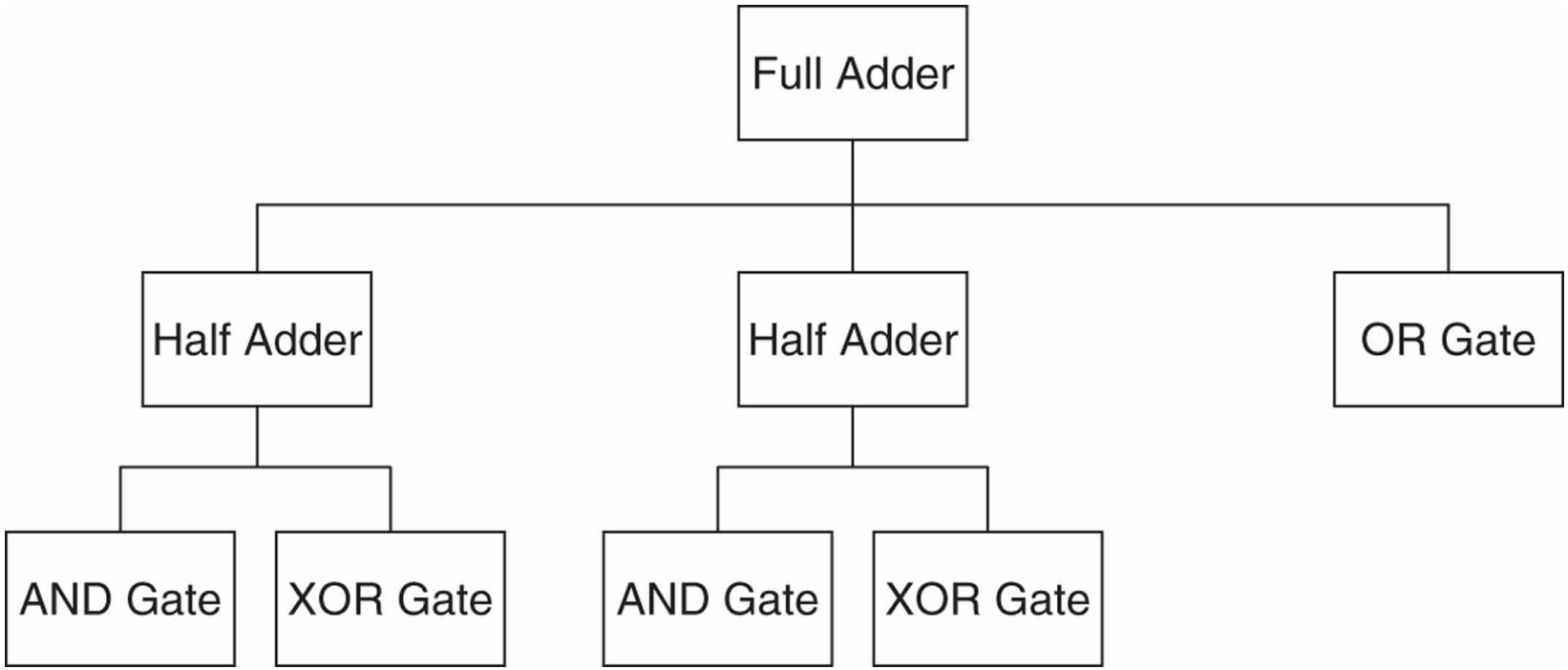
- Each module may itself be partitioned to further reduce its complexity

Advantages of Hierarchical Design

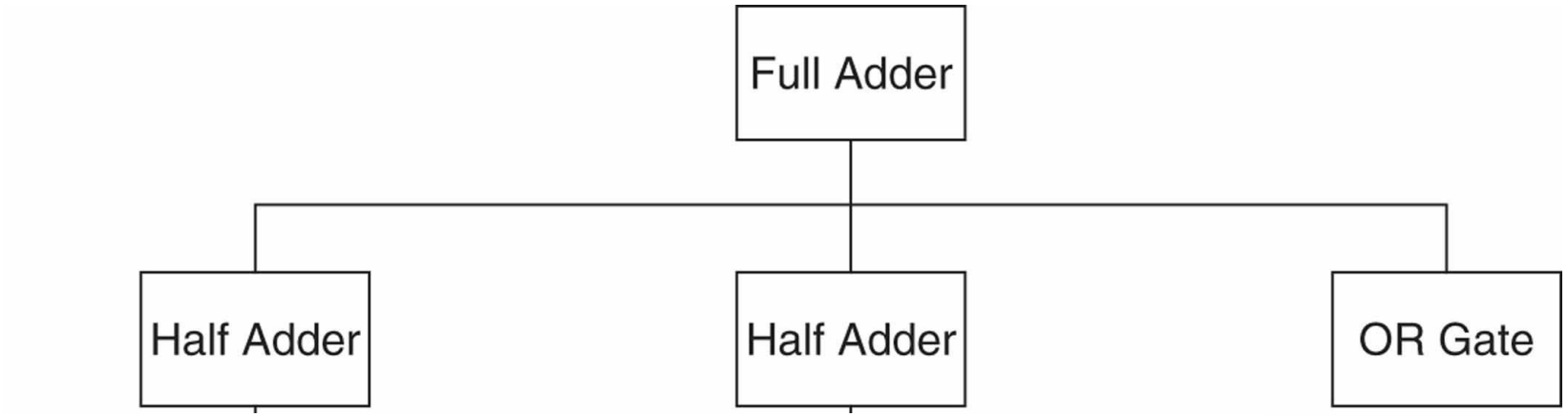
Most important benefits are:

- Easier design complexity management
  - Divided-and-conquer strategy
  - System can be implemented in stages (concurrently)
- Reuse of modules
  - Predefined modules or third-party designs
- Simple verification

Components


Full Adder
Half Adder
Half Adder
OR Gate
AND Gate
XOR Gate
AND Gate
XOR Gate

Components




```
entity half_adder is
port( x,y: in std_logic;
        sum, carry: out std_logic);
end half_adder;
```

```
component half_adder
port( x,y: in  std_logic;
        sum, carry: out std_logic);
end component;
```

- Component instantiation defines a subcomponent of the desing entity in which it appears

Components in Use

```
entity full_adder is
port(  x,y, carry_in: in std_logic;
         sum, carry_out: out std_logic);
end half_adder_tb;

architecture structural of full_adder is

        signal s1, s2, s3: std_logic;

        -- Component declarations
        component half_adder
              port(x,y: in  std_logic;
                    sum, carry: out std_logic);
        end component;

begin
        -- Component instantiations
        hf1: entity half_adder(dataflow) port map (x, y, s1, s2);
        hf2: entity half_adder(dataflow) port map (s1, carry_in, sum, s3);

        carry_out <= s2 or s3;

end structural;
```

Component Instantiations

- Entity and architecture

```
ha1: entity
    half_adder(dataflow)
    port map (x, y, s1, s2);
```

- Library, entity and architecture

```
ha1: entity
    work.half_adder(behavioural)
    port map (x, y, s1, s2);
```

- Entity without architecture and library

```
ha1: half_adder
    port map (x, y, s1, s2);
```

Port Mapping

Declaration

```
component half_adder
port( x,y: in  std_logic;
      sum, carry: out std_logic);
end component;
```

Instantiation

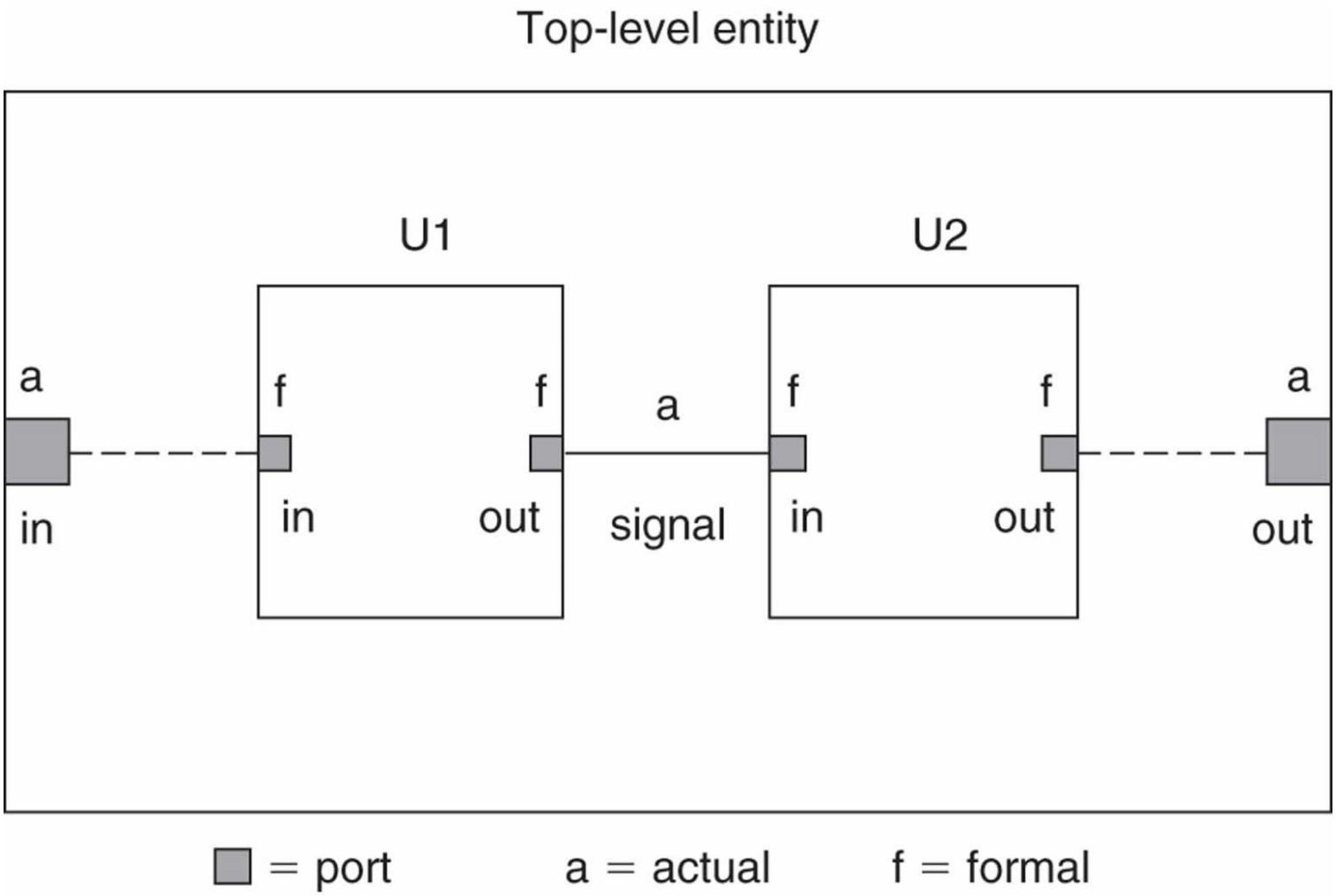
- Positional association

```
ha1: entity
    work.half_adder(dataflow)
    port map (x, y, s1, s2);
```

- Named association

```
ha1: half_adder
    port map (x => x, y => y,
              sum => s1,
              carry  => s2);
```

Component connections

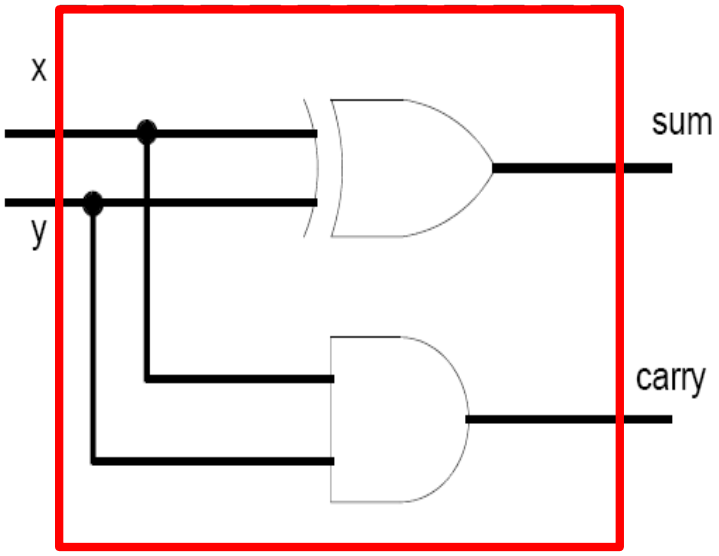


TEST BENCHES

Turun yliopisto
University of Turku

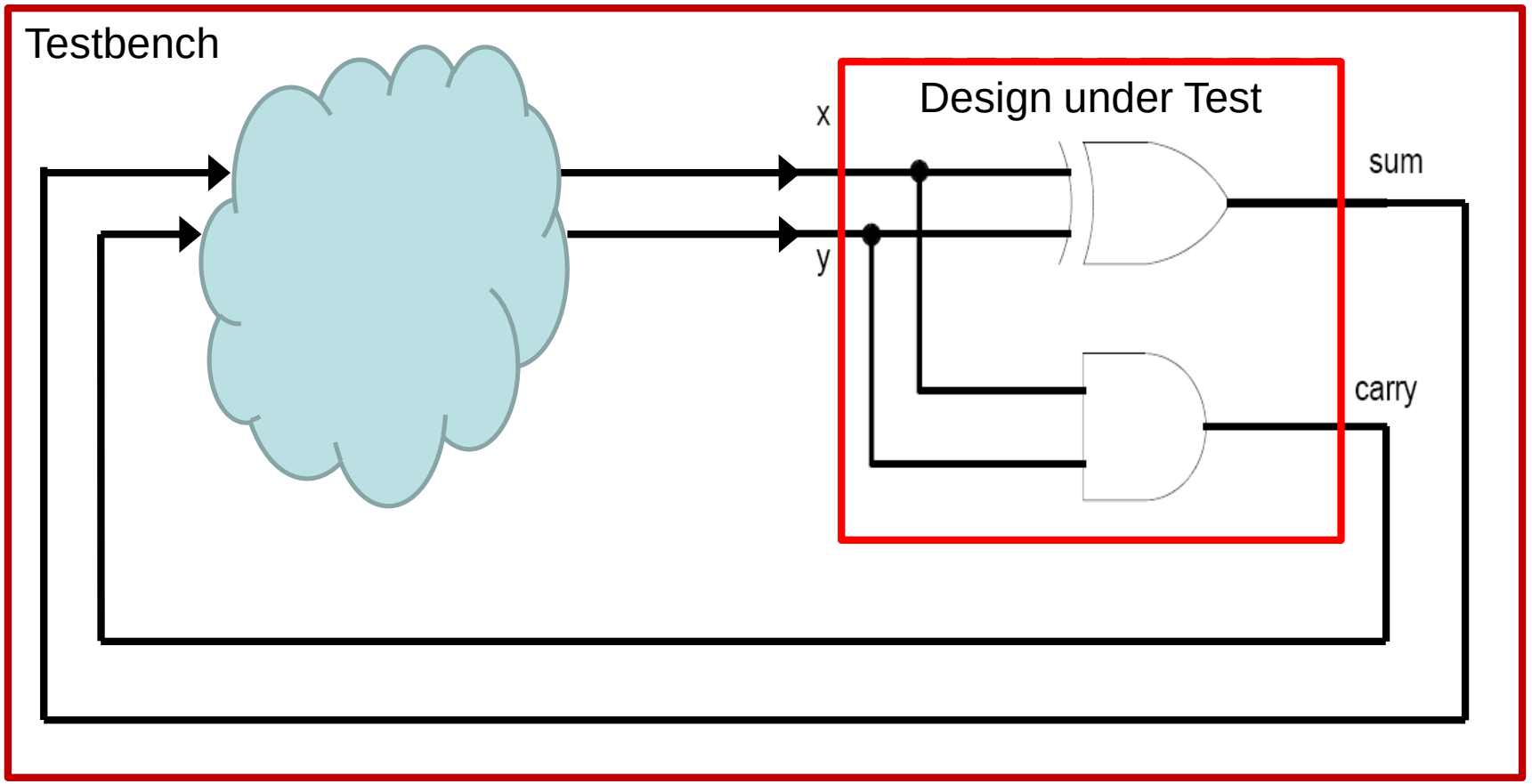
An Example Module (recap)

```
entity half_adder is
port(
      x,y: in std_logic;
      sum, carry: out std_logic);
end half_adder;

architecture myadder of half_adder is
begin
    sum <= x xor y;
    carry <= x and y;
end myadder;
```



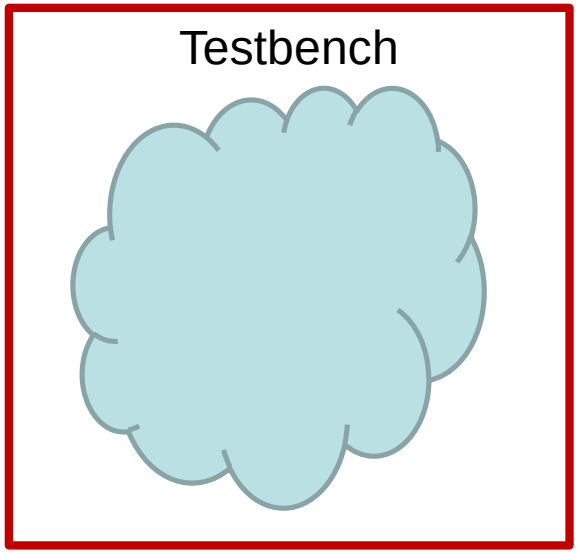
Testbench

- Testbench generates input signals for Device under Test (DUT) and observes DUT's response



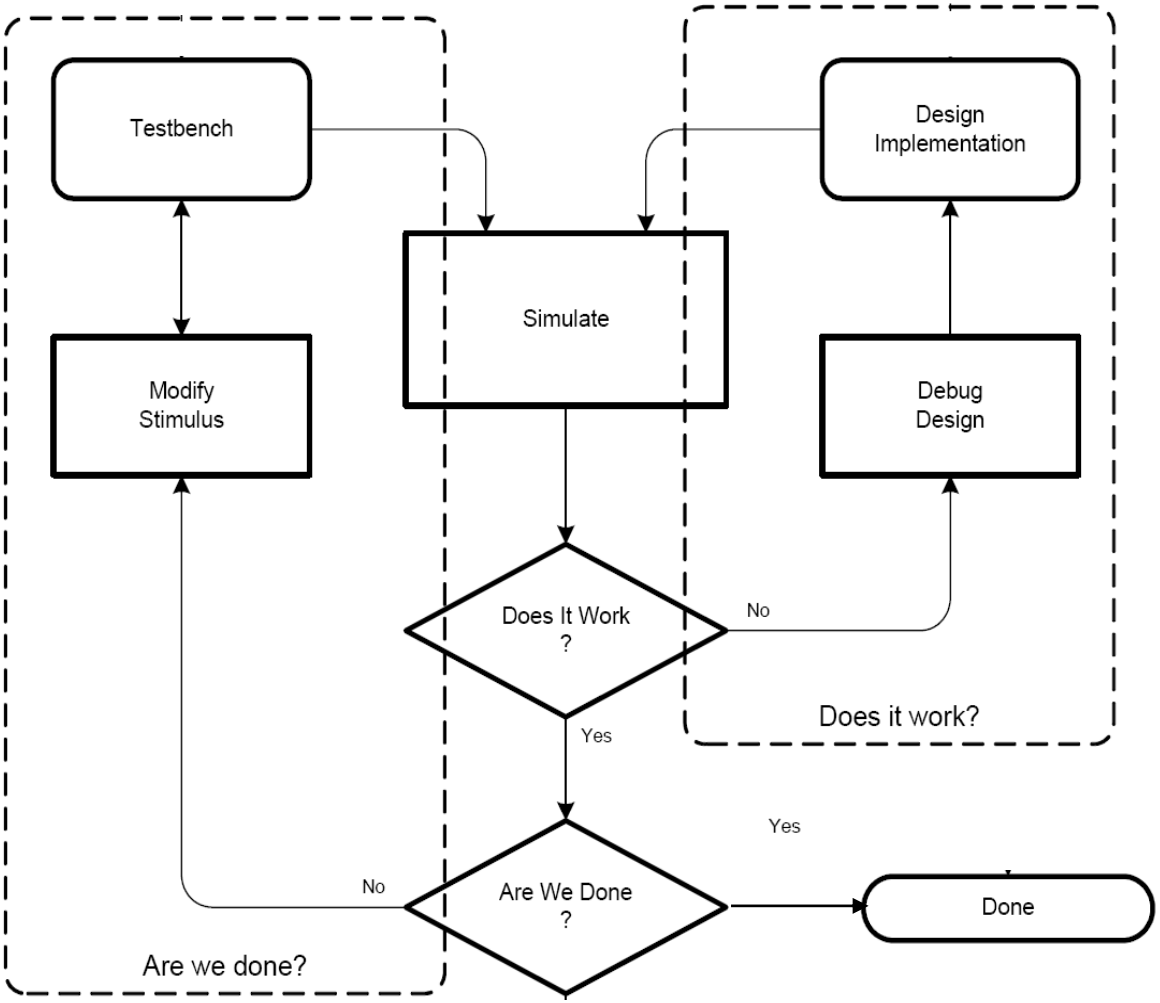
Testbench Structure

- Stimulus generator
  - Applies a sequence of predetermined stimulus values to the DUT inputs
- Response monitor
  - DUT's output values are checked to verify that they are equivalent with the expected ones
- Self-checking testbench
  - Golden model that provides "correct" results which are compared to DUT's response



Verification

- Verification of a design involves answering two fundamental questions:



Simple Testbench

```
entity half_adder is
port(
        x,y: in std_logic;
        sum, carry: out std_logic);
end half_adder;
```

```
entity half_adder_tb is
end half_adder_tb;

architecture behav of half_adder_tb is
     signal t_x, t_y, t_sum, t_carry : std_logic;

component half_adder
         port(x,y: in  std_logic;
                 sum, carry: out std_logic);
end component;

begin

  . . .

end behav;
```

Simple Testbench

```
entity half_adder is
port(
        x,y: in std_logic;
        sum, carry: out std_logic);
end half_adder;
```

```
entity half_adder_tb is
end half_adder_tb;

architecture behav of half_adder_tb is
    . . .
begin
    dut: entity work.half_adder(dataflow) port map (t_x, t_y, t_sum, t_carry);

    stimulus: process is
    begin
            t_x <= '0'; t_y <= '0';
            wait for 10 ns;
            . . .
    end process stimulus;
end behav;
```

Simple Testbench using Automatic Check

```
entity half_adder_tb is
end half_adder_tb;

architecture behav of half_adder_tb is

    . . .

    stimulus: process is
    begin
        t_x <= '0'; t_y <= '0';
        wait for 10 ns;
        assert ((t_sum = '0') and (t_carry = '0'))
            report "test failed for input compination 00" severity error;
        . . .
    end process stimulus;
end behav;
```

```
entity half_adder is
port(
    x,y: in std_logic;
    sum, carry: out std_logic);
end half_adder;
```

Assert Statement

- Checks whether a specific condition is true
  - A message is displayed if the condition does not hold
- Several severity levels:
  - Note
  - Warning
  - Error (*default if not otherwise stated)*
  - Failure
- For debugging, it is good to indicate the input values in a report statement if an error occured
- Can be used as a sequential or a concurrent statement

OBJECTS AND DATA TYPES

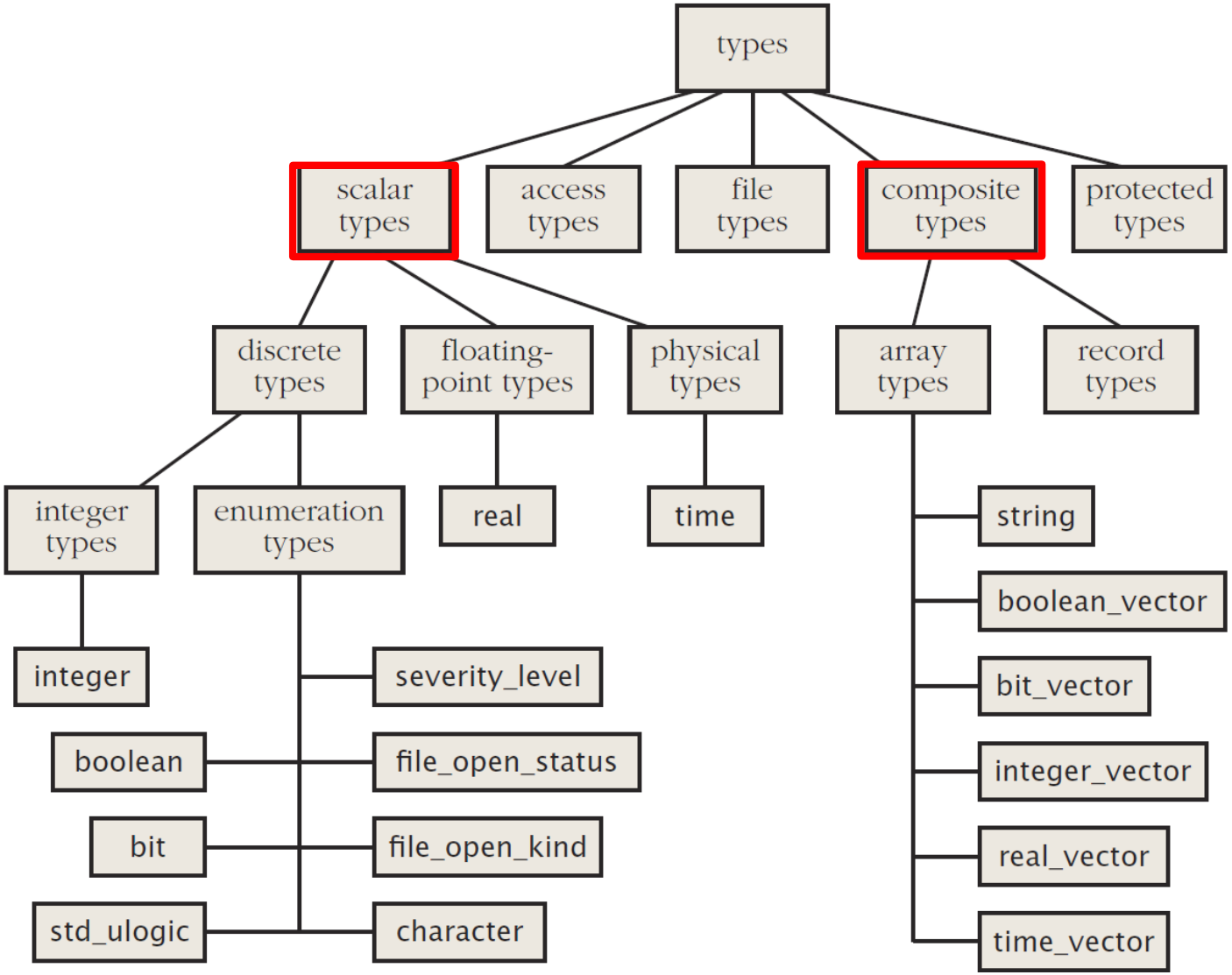
Four Classes of Objects

- **Signal**
  - An object with *current* and *future* values
  - Can be changed as many times as neccessary
- **Constant**
  - The value cannot be changed after initialisation
- **Variable**
  - An object with *current* value
  - Can be changed as many times as neccessary
- File

- An object is a named item in a VHDL model that has a value of a specified type

VHDL Type Classification

Synthesisable



VHDL is a **strongly typed** language

- Every object may only assume values of its nominated type
- The definition of each operation includes the types of values to which the operation may be applied

Type Declarations

- The declaration names a type and specifies which values may be stored into it

```
type spruce is range 0 to 100;
type birch is range 0 to 100;

variable v1: spruce;
variable v2: birch;

v1:= 10;
v2:= 12;
```

Data Types

Type	Range of values	Example
bit	‘0’, ‘1’	signal x: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
std_logic, std_ulogic	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	signal x: std_logic;
std_logic_vector, std_ulogic_vector	an array with each element of type bit	signal x: std_logic_vector(0 to 7);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE’
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. ‘#’)	variable VAL: character :=’$’;
integer	range is implementation dependent but includes at least –$(2^{31} – 1)$ to +$(2^{31} – 1)$	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;
positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;

BIT

Value	Meaning
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1

STD_ULOGIC

Value	Meaning
‘U’	Uninitialized
‘X’	Forcing (Strong driven) Unknown
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1
‘Z’	High Impedance
‘W’	Weak (Weakly driven) Unknown
‘L’	Weak (Weakly driven) 0. Models a pull down.
‘H’	Weak (Weakly driven) 1. Models a pull up.
‘-’	Don't Care

synthesisable

Resolved Types

- Resolved types are declared with a resolution function
- A resolution function defines the resulting value of a signal if there are multiple driving sources

subtype std_logic is *resolved* std_ulogic;

- *As a subtype, all operations and functions defined for std_ulogic apply to std_logic*

Resolution Table for std_logic

	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	W	W	X
H	U	X	0	1	H	W	H	H	X
-	U	X	X	X	X	X	X	X	X

Enumerated Types

- An enumeration type is defined by listing, that is, enumerating all possible values

type type_name **is (** enumeration_literal
{, enumeration_literal} **);**

type colour is (green, yellow, red);
variable status :colour;
status :=red;

SIGNALS

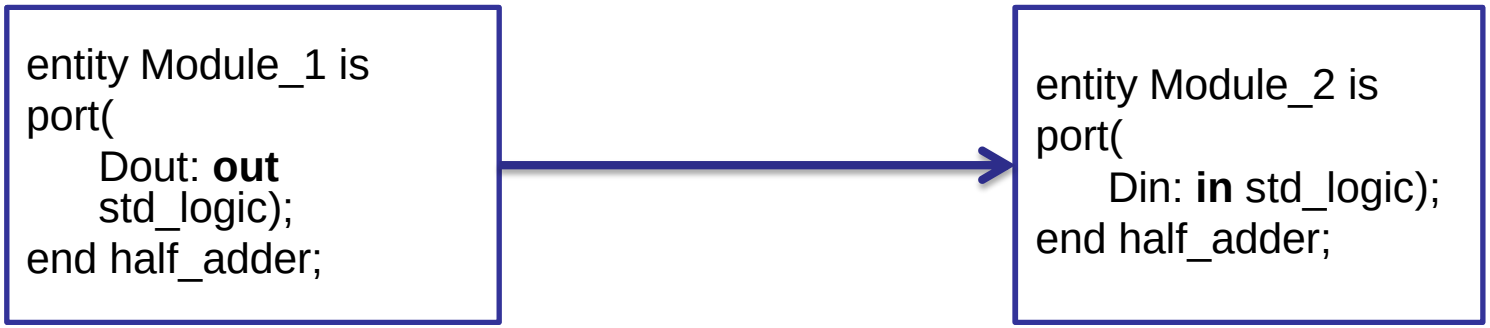
Difference between Variables and Signals

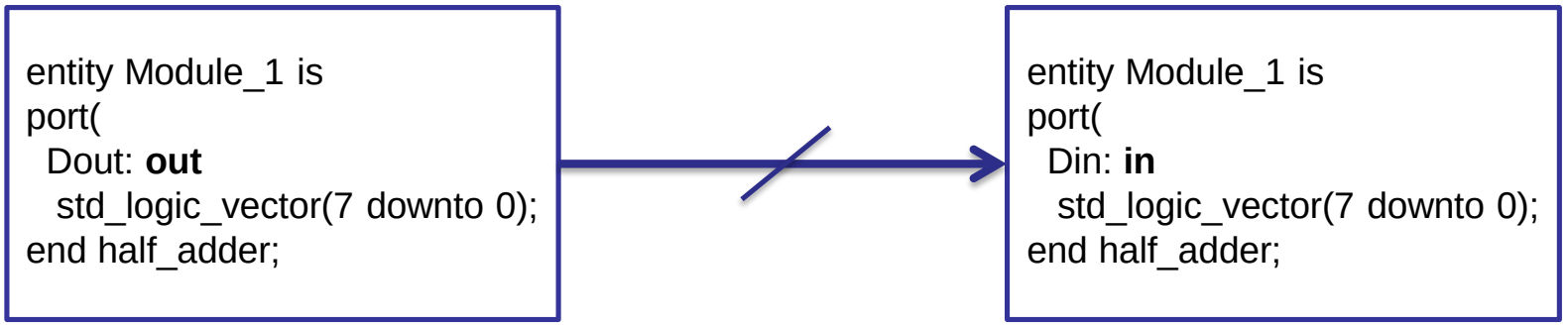
- Variables are sequential statements
  - That is, they are used in processes
- Signals are concurrent statements

- **The value of a variable is updated immediately whereas the value of a signal is updated after a delay**
  - Very important to remember

Signals

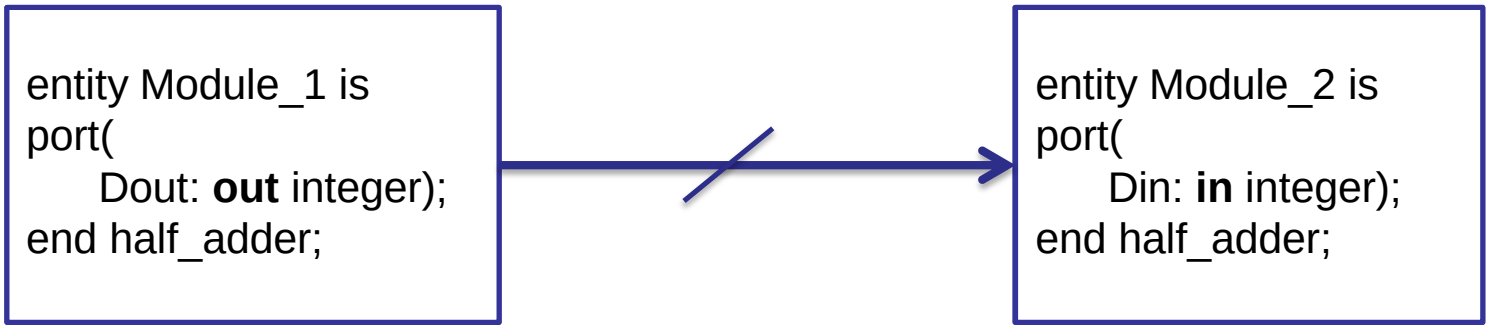
- signals as the means of joining component instances

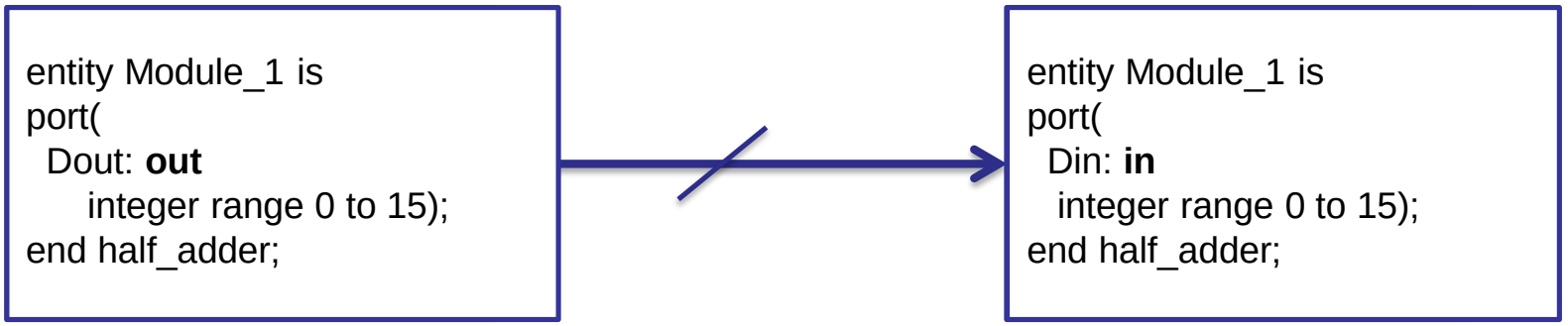




Signals

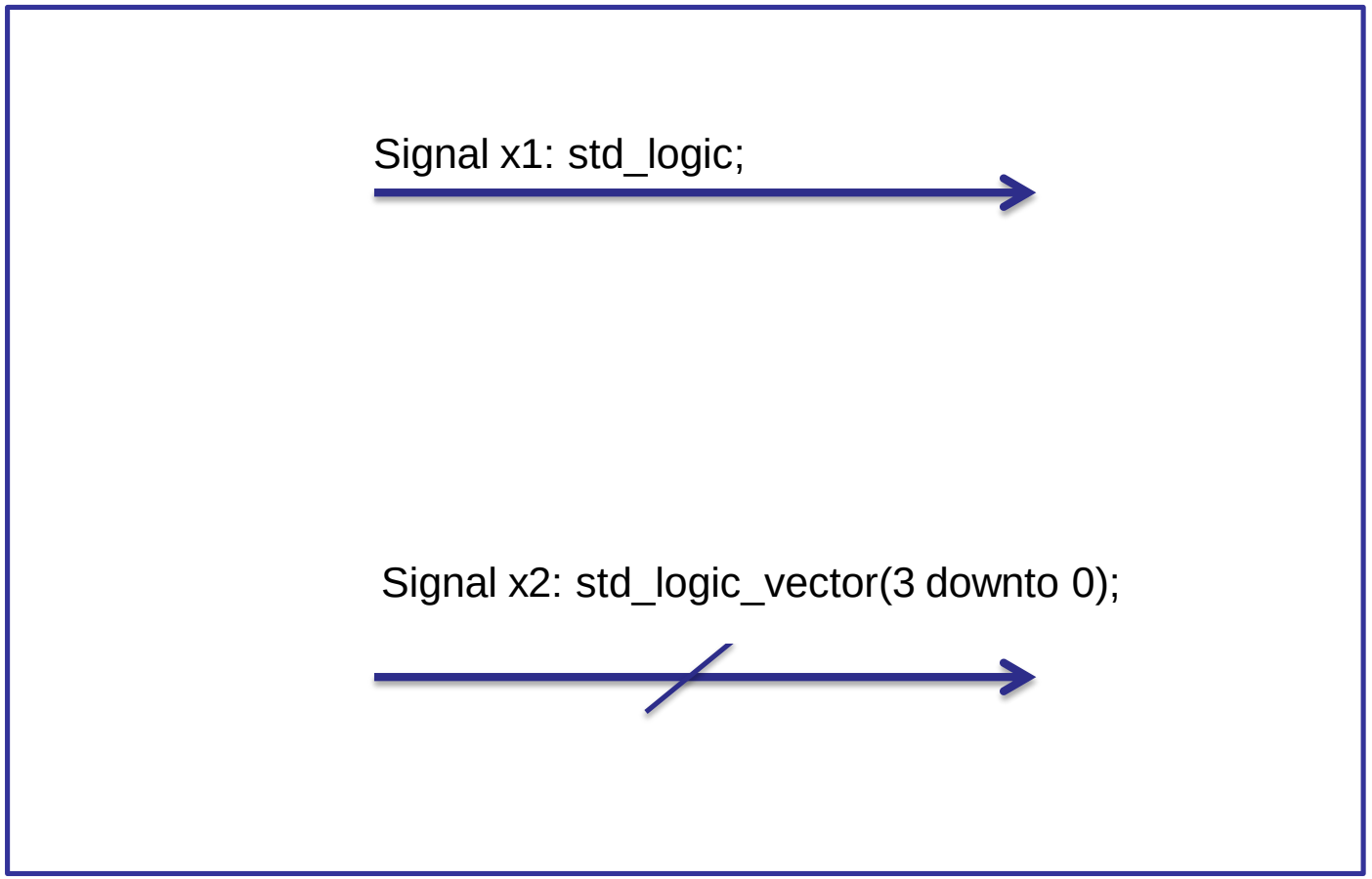
- signals can also be of type integer





Signals within a Module

- signals as the means of joining processes within a module



Signals within a Module

- The *signal declaration within* an architecture defines internal signals for the module

```
architecture structural of my_module is
        signal x1 : std_logic;
        signal x2: std_logic_vector(3 downto 0);
begin
  ...
end architecture;
```

Assigning Values to Signals

```
signal x1: std_logic;
signal x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(15 downto 0);
signal x4: std_logic_vector(9 downto 0);
```

```
x1 <= '1';
x2 <= "0101";     -- Binary base assumed by default (B"0000")
x3 <= X"FF67";  -- Hexadecimal base
x4 <= O"713";    -- Octal base
```

Assigning Values to Signals (Vectors)

```
signal x1: std_logic;
signal x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(15 downto 0);
signal x4: std_logic_vector(9 downto 0);
```

```
x2(0) <= '1'; -- assigns value to the rightmost element of x2
```

```
x2 <= x3(3 downto 0); -- assigns the 4 rightmost values to x2
x2 <= ( 3 => '1', 0 => '1', 2 => '1', 1 => '0');
x2 <= ( 3|0 |2 => '1', 1 => '0');
x2 <= ( others => '1', 1 => '0');
x2 <= (others => '0');
```

Assigning Values to Signals - VHDL-2008

- An array aggregate forms an array from a collection of elements

  ('0', "1001", '1')

- Can be used for the target of an assignment statement as well

```
signal a, b, sum : unsigned(7 downto 0);
signal carry     : std_ulogic;
...
(carry, sum) <= ('0' & a) + ('0' & b);

( 13 downto 8 => opcode,
  6 downto 0 =>register_address,
  7 => destination)   := instruction;
```

Assigning Values to Signals – VHDL 2008

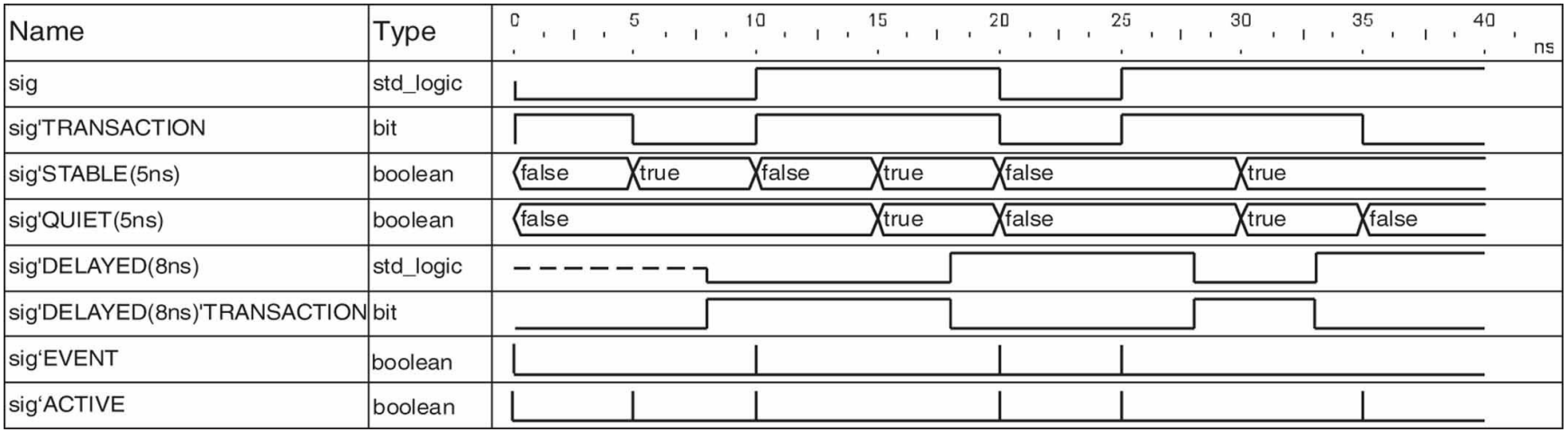
```
signal x1: std_logic_vector(5 downto 0);

x1 <= 6x"0f";
x1 <= 6SX"F";   (sign extension)
x1 <= 6Ux"f";   (zero extension)
x1 <= 6sb"11";  (binary format)
x1 <= 6uO"7";   (octal format)
```

Signal Attributes

Attribute	Result
S'Transaction	Implicit bit signal whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active).
S'Stable(t)	Implicit boolean signal. True when no event has occurred on S for t time units up to the current time, False otherwise.
S'Quiet(t)	Implicit boolean signal. True when no transaction has occurred on S for t time units up to the current time, False otherwise.
S'Delayed(t)	Implicit signal equivalent to S, but delayed t units of time.
S'Event	A boolean value. True if an event has occurred on S in the current simulation cycle, False otherwise.
S'Active	A boolean value. True if a transaction occurred on S in the current simulation cycle, False otherwise.
S'Last_event	Amount of elapsed time since last event on S, if no event has yet occurred it returns TIME'HIGH.
S'Last_active	Amount of time elapsed since last transaction on S, if no transaction has yet occurred it returns TIME'HIGH.
S'Last_value	Previous value of S immediately before last event on S.
S'Driving	True if the process is driving S or every element of a composite S, or False if the current value of the driver for S or any element of S in the process is determined by the null transaction.
S'Driving_value	Current value of the driver for S in the process containing the assignment statement to S.

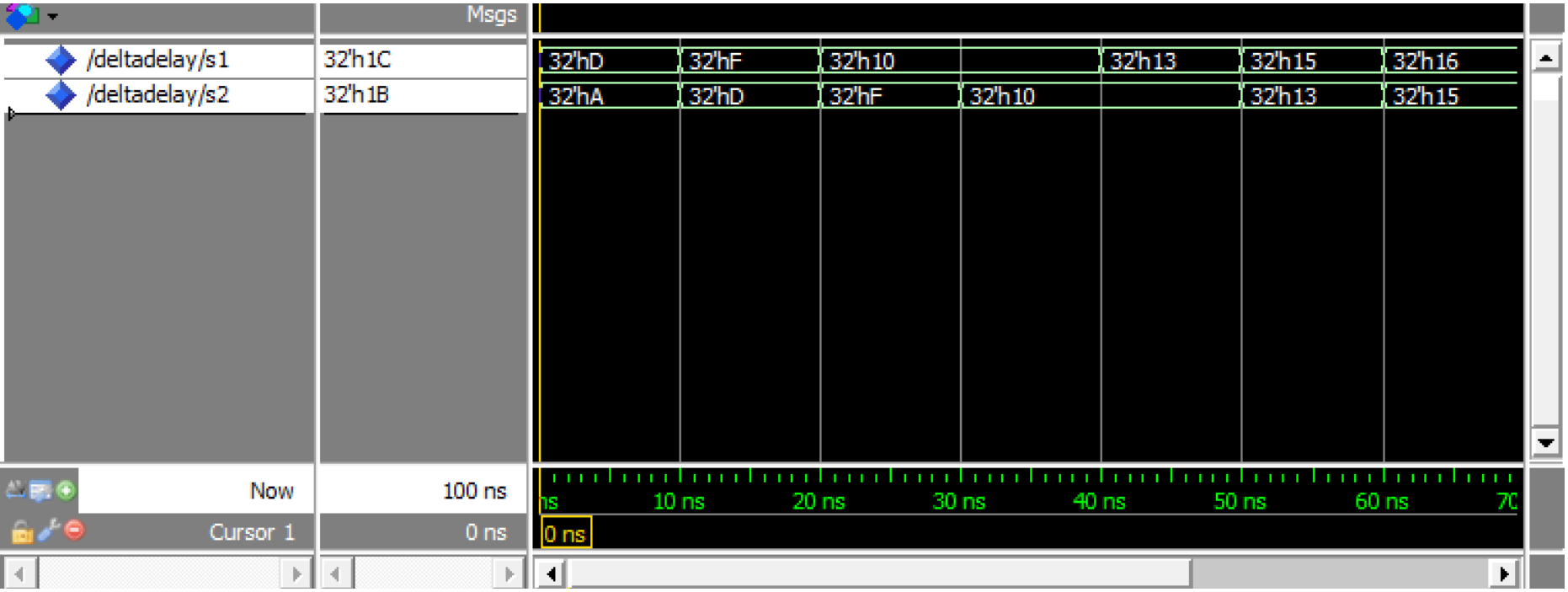
Signal Attributes

Attribute	Result
S'Transaction	Implicit bit signal whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active).
S'Stable(t)	Implicit boolean signal. True when no event has occurred on S for t time units up to the current time, False otherwise.
S'Quiet(t)	Implicit boolean signal. True when no transaction has occurred on S for t time units up to the current time, False otherwise.
S'Delayed(t)	Implicit signal equivalent to S, but delayed t units of time.
S'Event	A boolean value. True if an event has occurred on S in the current simulation cycle, False otherwise.
S'Active	A boolean value. True if a transaction occurred on S in the current simulation cycle, False otherwise.

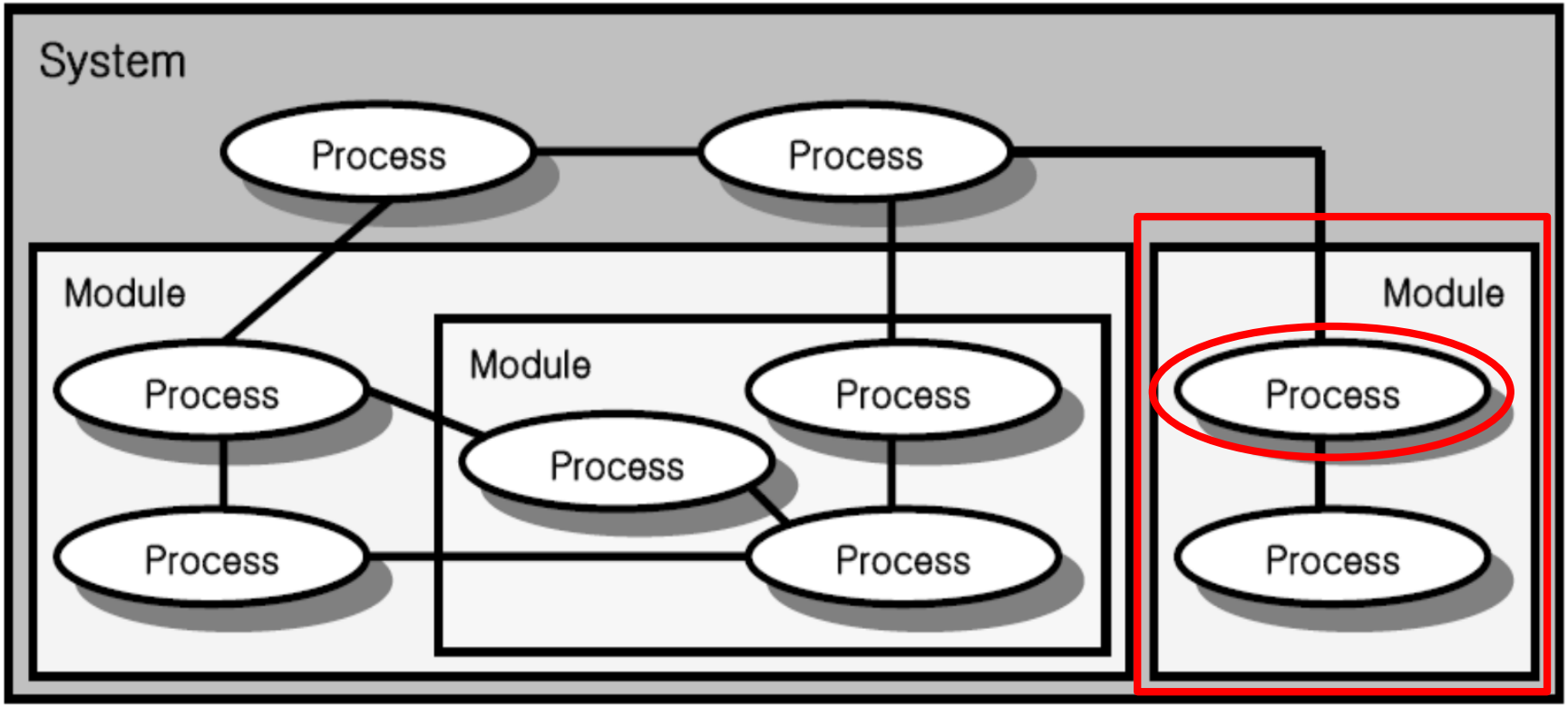


Delta Delay

```
12      dd: process
13      begin
14        for i in 3 downto 0 loop
15          S1 <= s1+i;
16          S2 <= s1;
17          wait for 10 ns;
18        end loop;
19      end process dd;
```



Processes


System
Process
Process
Module
Module
Module
Process
Process
Process
Process
Process
Process
Process

Processes

- Processes are executed

  in parallel

- Statements within a process are executed

  in sequece

- After all process's statements are executed, the process starts again from the beginning
  - Signals in the sensitity list awakes the process for executions

Table 5.3.1
VHDL's sequential statements.

Sequential Statements
signal assignment
variable assignment
wait
assertion
report
procedure call
return
case
null
if
loop
next
exit

Processes

- Each process starts with a keyword *process*
- Processes are executed *in parallel*
- Statements within a process are executed *in sequece*
- After all process's statements are executed, the process starts again from the beginning

```
[ process_label : ]
    process [ ( sensitivity_list ) ] [ is ]
            { process_declarative_item }  -- process declarative part
    begin
            { sequential_statement }  -- process statement part
    end process [ process_label ] ;
```

Sensitivity List

- A list of *signals* to which a process is sensitive
- Signals in the sensitity list awakes the process for executions
- **NOTE**, only those signals that are read, that is, are on the right side of a statement are allowed to be in the sensitivity list
  - Improper use of sensitivity list causes mismatch between the design and its implementation
- The keyword *all* can be used to denote all the signals that are read within a process
  - VHDL-2008 only

Sequential Statements

VHDL's sequential statements.

Sequential Statements
signal assignment
variable assignment
wait
assertion
report
procedure call
return
case
null
if
loop
next
exit

- Can be used *only* within a process
- Executed one after another

Case Statement

- Chooses one of the given statements
- *One and only one* choice can and must match
- Choices must be *mutually exclusive*
- *Others* can be used to denote unspecified choices

```
case expression is
when choices => sequence_of_statements
{ when choices => sequence_of_statements }
end case [ case_label ] ;
```

---

choices ::= choice { | choice }

choice ::= simple_expression | discrete_range | *element*_simple_name | **others**

Case Statement

```
entity simple is
port (x1: in std_logic_vector(1 downto 0);
      x2: out std_logic_vector(7 downto 0) );
end;

architecture behaviour of simple is
begin
casex :Process  ( x1) begin
      case x1 is
          when “00” => x2 <= X”01”;
          when ”01” => x2 <= X”10”;
          when ”10” => x2 <= X”11”;
          when ”11” => x2 <= X”11”;
      end case;
end process;
end architecture;
```

Case Statement (equivalent)

```
entity simple is
port (x1: in std_logic_vector(1 downto 0);
      x2: out std_logic_vector(7 downto 0) );
end;

architecture behaviour of simple is
begin
casex :Process  ( x1) begin
      case x1 is
           when ”00” => x2 <= X”01”;
           when ”10” => x2 <= X”10”;
           when others => x2 <= X”11”;
      end case;
end process;
end architecture;
```

Null Statement

- Null statement explicitly indicates that there is nothing to be done
- The null statement can be used anywhere where a sequential statement is required
  - Used mainly in case statements
  - For example, in processor models, one may use *null* when *nop* (*no operation)* is performed

```
case x1 is
        when ”00” => x2 <= X”01”;
        when ”10” => x2 <= X”10”;
        when others => null;
end case;
```

If Statement

- Chooses one of the given statements as case
- Chooses the first matching condition for execution
- If none of the conditions are met, the construct is terminated and not statements are executed

```
if condition then
        sequence_of_statements
{ elsif condition then
        sequence_of_statements }
[ else
        sequence_of_statements ]
end if [ if_label ] ;
```

If Statement

```
1  entity simple is
2  port (
3      signal x1, x2: std_logic_vector(1 downto 0);
4      signal x3: std_logic_vector(3 downto 0) );
5  end;
6
7  architecture behaviour of simple is
8  begin
9      ifx :Process  ( x1)
10     begin
11         If  x1 /= x2 then
12             x3 <= X"1";
13         else
14             x3 <= X"0";
15         end if;
16     end process;
17 end architecture;
```

Loop Statement

- Repeatedly executes a sequence of sequential statements
- Continues until one of the following happens:
  - Completion or the iteration scheme or the execution of *exit, next* or *return* statement

```
[ loop_label:]
[ iteration_scheme ] loop
sequence_of_statements
end loop [ loop_label ] ;
```

```
iteration_scheme ::= for identifier in discrete_range | while condition
discrete_range ::= discrete_subtype_indication | range
```

While Loop

- Boolean iteration scheme
- The condition is evaluated before the execution of sequential statements
- One must ensure that the loop condition will eventually become false
- the execution continues with the next statement after the loop if the condition does not hold

```
while loop_index > 0 loop
        sequential statements;
        update loop_index;
end loop;
sequential statements;
```

For Loop

- *identifier* takes successive values of the discrete range in each iteration of the loop
  - the number of repetitions is determined by an integer range
  - The counting starts from the left element
- After the last value in the iteration range is reached, the loop is skipped, and execution continues with the next statement after the loop

```
for loop_counter in 0 to 15 loop
        counter_out <= loop_counter;
        wait for 10 ns;
end loop;
```

Infinite Loop

- NO *while* or *for* iteration scheme
- Enclosed statements are executed repeatedly forever until an *exit* or *next* statement is encountered
- Infinet loops are not synthesisable

Exit Statements

- The next statement skips the remainder of the current loop and continues with the next loop iteration

```
[ label : ] next [ loop_label ] [ when condition ] ;
```

- The exit statement skips the remainder of the current loop and continues with the next statement after the exited loop

```
[ label : ] exit [ loop_label ] [ when condition ] ;
```

Loop Statements

```
signal X, Y: std_logic_vector( 7 downto 0);
. . .
a_loop: for i in X'range loop
. . .
        b_loop: for j in Y'range loop
            . . .
            next a_loop when i <  j;
            . . .
        end loop b_loop;
. . .
end loop a_loop;
```

- A next statement with label terminates the current iteration of the named loop

Conditional Signal Assignments

- Shorthand notation for the IF statement

```
conditional_signal_assignment ::=
target <= { value_expression when condition else }
          value_expression [ when condition ];
```

- Conditions are evaluated in the order of apperance
- Last values must NOT have an associated condition
- Only available in VHDL 2008
  - VHDL 2008 provides a conditional variable assignement as well (Older versions must use the if statement)

Selected Signal Assignment

- Shorthand notation for the CASE statement

```
selected_signal_assignment ::=
with expression select
  target <= { value_expression when choices , }
            value_expression  when choices ;
```

- The value of *expression* is first evaluated after which the value is simultaneously compared againts all the *choices*
- Only available in VHDL 2008
  - VHDL 2008 provides a selected variable assignement as well (Older versions must use the if statement)

Don’t Care Inputs and Outpus

- Don’t care indicates that we are not interested is the value ’1’ or ’0’
- Don’t care value is represented with the character ’-’
- NOTE, don’t care value is treated differently by simulators and synthesisers
  - Synthesisers take advantage of the don’t care value to mimise the logic it synthesises

```
x1  <=  ”00” when s = ”0---” else
        ”10” when s = ”1---” else
        ”11”;
```

```
x1  <=  ”0” when s = ”1000” else
        ”1” when s = ”1100” else
        ”-”  when others;
```

Taks

- Create a 4-1 mux
  - One entity-architecture pair
  - Use std_logic/std_logic_vector for the signals
  - In the 4-1 mux, one process to select one constant value to output
    - A case statement
  - In testbench, create the select signal
    - Using a for loop
    - Use *wait for 10 ns* to create a delay for the selection signal
    - No clock signal