Turun yliopisto
University of Turku

HDL Based Design
ME620138

Course Information

- Lecturer:
  - University Research Fellow Tomi Westerlund
    tomi.westerlund@utu.fi
- Credits: 5 ects at UTU
- Course Requirement:
  - Exercises
- Pre-Requirements: none
- Helpful knowledge: digital system design, basic knowledge of HDL and object-oriented programming

Course Information

- Lecture and other material will be sent to you
- During the lectures, we will also look at the exercises
- No specific web pages for the course

Exercises

- All exercises are obligatory
- Always remember to **comment** your code
  - Insert comment blocks on files, classes and class members (variables and member functions)
  - Comment should describe the **intent** of the code
- Must be send to lecturers

Passing the Course

Do all the exercises

Pass the exam

What is the task of

a design engineer?

a verification engineer?

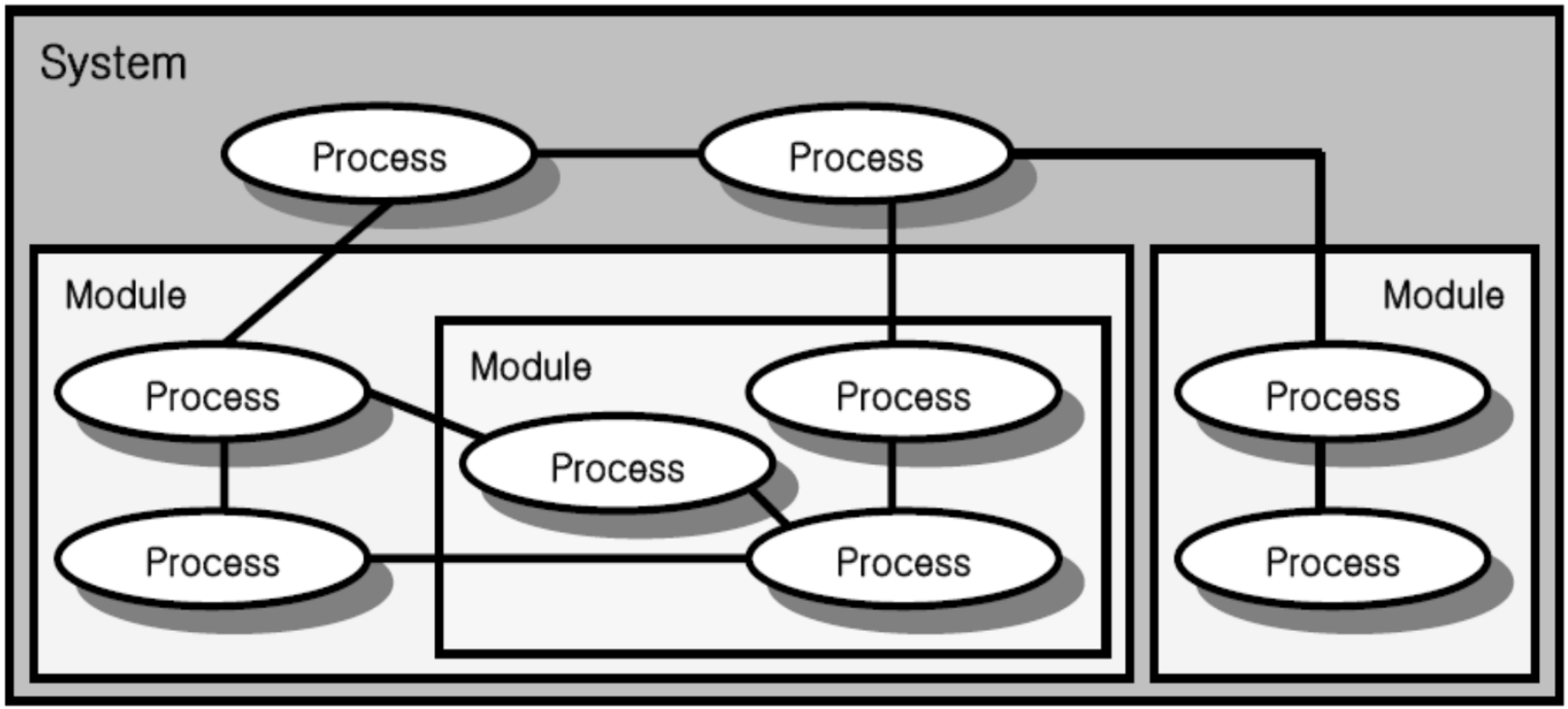
- Design Engineer
  - To create a device that performs a particular task based on a design specification

- Verification Engineer
  - To ensure that the model / design works correctly and succesfully

Can't see the wood for the tree

- In Finnish: Nähdä metsä puilta

What is the wood?



Modelling

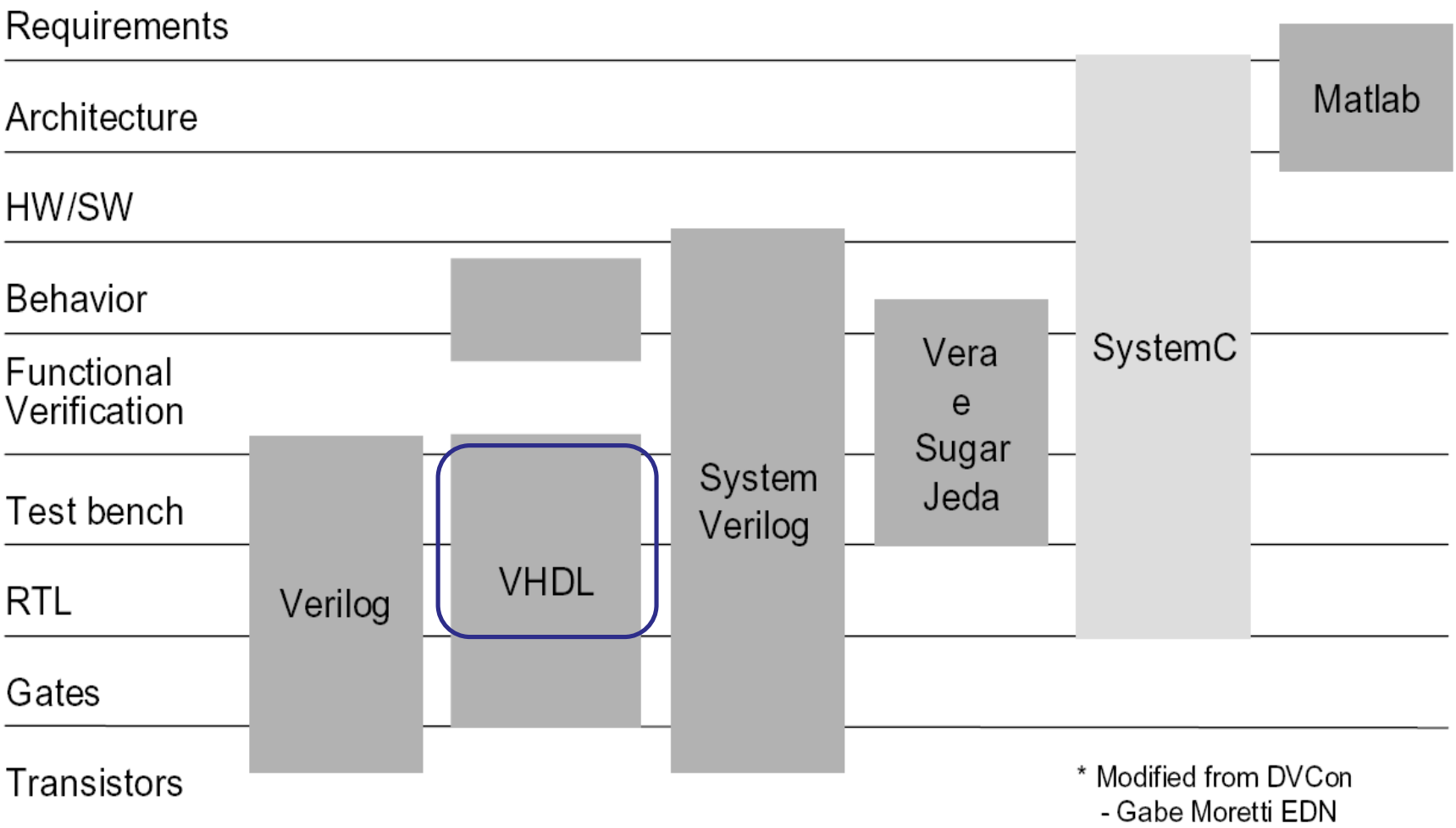
- A model helps us to understand the important aspects of a complex system *before* its implementation
- A model attempts to simulate an *abstract* model of a particular system
- The abstraction level of a model defines how much low-level details are included in the model

Levels of Abstraction

Level	Components	Representation
Behavioral	Algorithms, FSMs	Algorithmic representation of system functions
System (architectural)	Processors, memories, functional subsystems	Interconnection of major functional units
Register Transfer (dataflow)	Registers, adders, comparators, ALUs, counters, controllers, bus	Data movement and transformation and required sequential control
Gate (logic level)	Gates, flip-flops	Implementation of register level components using gates and flip-flops
Transistor	Transistors, resistors, capacitors	Implementation of gates and flip-flops using transistors, capacitors, and resistors

WHAT KIND OF MODELLING LANGUAGES THERE ARE?

Language Comparison



Language Comparison

- Verification
  - Object Oriented
  - Randomisation
  - Verification support
  - Verification methodology

SystemVerilog

- RTL
  - Multi-threaded
  - Hardware data types
  - Netlisting

VHDL | Verilog

Scope of VHDL

- VHDL is designed for the specification, design and description of digital hardware systems

- VHDL stands for VHSIC HDL

Very High Speed Integrated Circuits

Hardware Description Language

History of VHDL

1981	Initiated by US DoD to address hardware life-cycle crisis
1983-85	Development of baseline language by Intermetrics, IBM and TI
1986	All rights transferred to IEEE
1987	Publication of IEEE Standard
1987	Mil Std 454 requires comprehensive VHDL descriptions to be delivered with ASICs
1994	Revised standard (named VHDL 1076-1993)
2000	Revised standard (named VHDL 1076 2000, Edition)
2002	Revised standard (named VHDL 1076-2002)
2007	VHDL Procedural Language Application Interface standard (VHDL 1076c-2007)
2009	Revised Standard (named VHDL 1076-2008)

Free Format

- VHDL is a “free format” language
- No formatting conventions, such as spacing or indentation imposed by VHDL compilers
  - Space and carriage return treated the same way

Example:

```
if (x=y) then
```

or

```
if  (x=y)          then
```

or

```
if (x = y)
 then
```

are all equivalent

Naming and Labeling

- VHDL is case insensitive

  Example:

  Names or labels

  `databus`

  `Databus`

  `DataBus`

  `DATABUS`

  are all equivalent

Comments

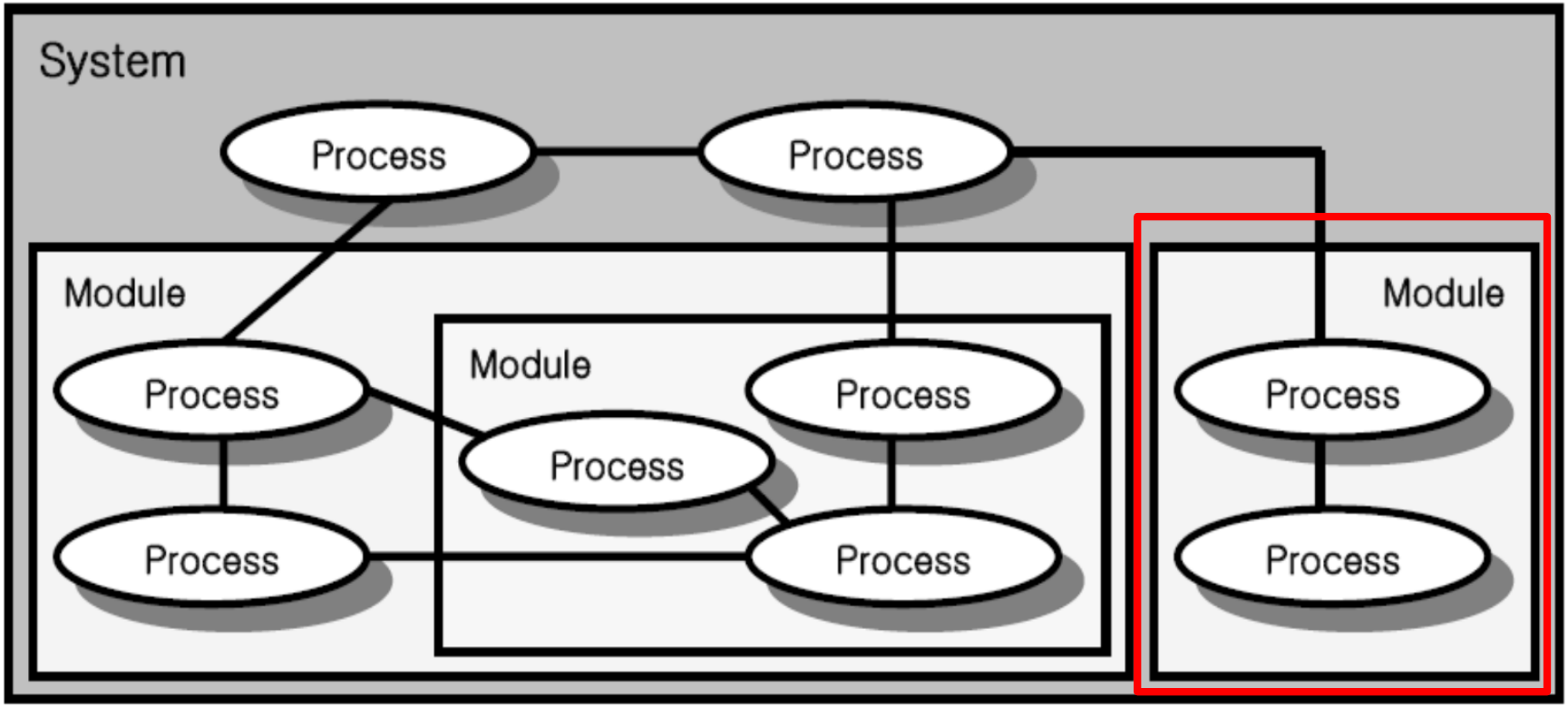
- Comments are indicated with a “double dash” “--”
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:

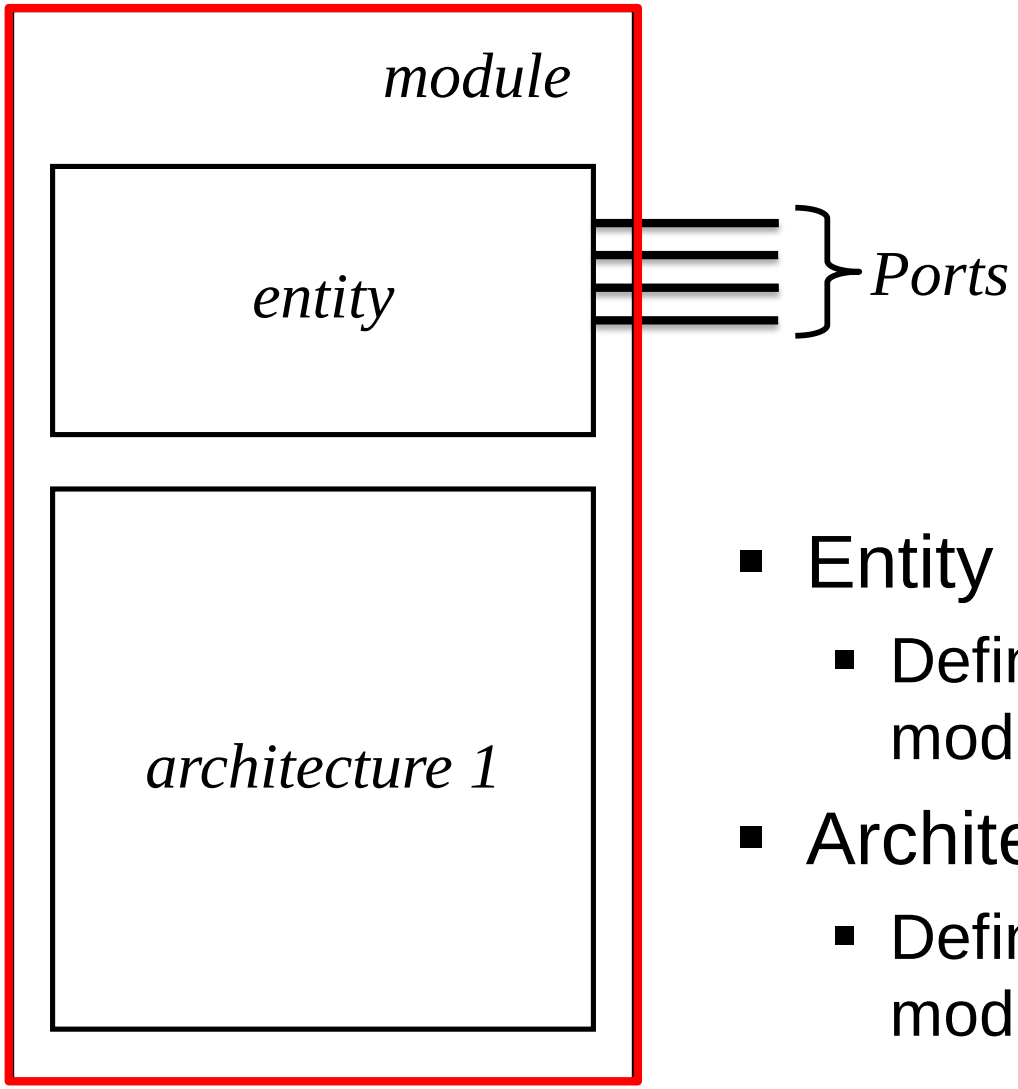
```
-- this is a comment
xy <= x * y;      -- comments also work in the end of a line
```

ENTITIES AND ARCHITECTURES

Basic VHDL Concepts

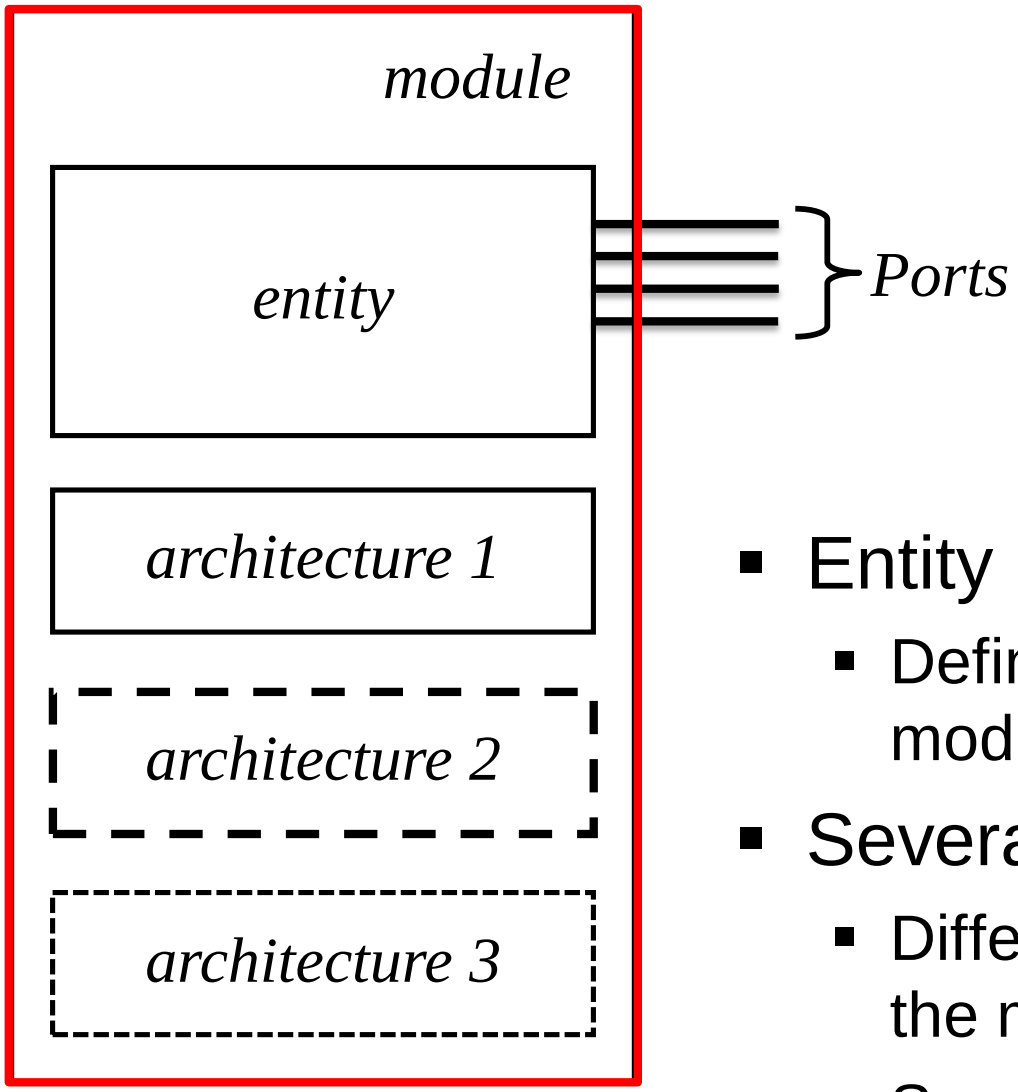


Design Entity



- Entity
  - Defines the interface of the module
- Architecture
  - Defines the behaviour of the module

Design Entity

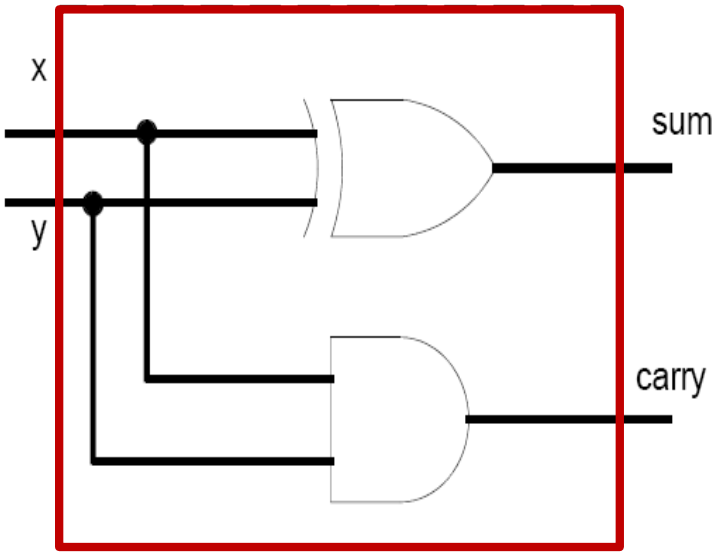


- Entity
  - Defines the interface of the module
- Several Architecture
  - Different implementations of the model
  - Same entity

An Example Module

```
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;
```



```
ENTITY entity_name IS
   PORT (
      port_name :  port_mode  signal_type;
      port_name :  port_mode  signal_type;
      …
      port_name :  port_mode  signal_type
       );
END entity_name;
```

```
ARCHITECTURE architecture_name OF entity_name IS
   [ declarations ]
BEGIN
    code (concurrent statements)
END architecture_name;
```

Port Modes

The *Port Mode* of the interface describes the direction in which data travels with respect to the c*omponent*

- **In**: Signal can only be read, that is, it can appear **only on the right side** of a signal or variable assignment
- **Inout**: The value of a bi-directional port can be read and written. It can appear on **both sides** of a signal assignment
- **Out**: Depending on the VHDL version **out** can only be written or read as well
- **Buffer:** Output signal, but it can also be read, and therefore a signal can appear on **both sides** of a signal assignment.
  - Not recommended to be used in the synthesisable code

Port Modes Out and Buffer

- **Out** port is intended to be used when there is no internal use of the port's value
- **Buffer** port is intended to be used when the port's value is used within the module

- In VHDL-2002, the rules for **buffer** ports were relaxed
  - Allowed to be connected to external **out** and **buffer** ports
- In VHDL-2008, the restriction on reading **out** ports is removed

Signal Types

- **bit** and **bit_vector**
- **std_logic** and **std_logic_vector**
- **std_ulogic** and **std_ulogic_vector**
- **boolean**
- **signed** and **unsigned**

- All the above signal types are defined in libraries

LIBRARIES

Libraries and Packages

- A *design library* is a logical storage area for compiled design units

Table 13.2.1
Standard packages commonly used in synthesis.

Packages	Library	Standard Number	Standard Title
standard, textio	std	IEEE Std 1076	IEEE Standard VHDL Language Reference Manual
std_logic_1164	ieee	IEEE Std 1164	IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)
numeric_bit, numeric_std	ieee	IEEE Std 1076.3	IEEE Standard VHDL Synthesis Packages
std_logic_unsigned, std_logic_signed	ieee	None	None
std_logic_arith	ieee	None	None

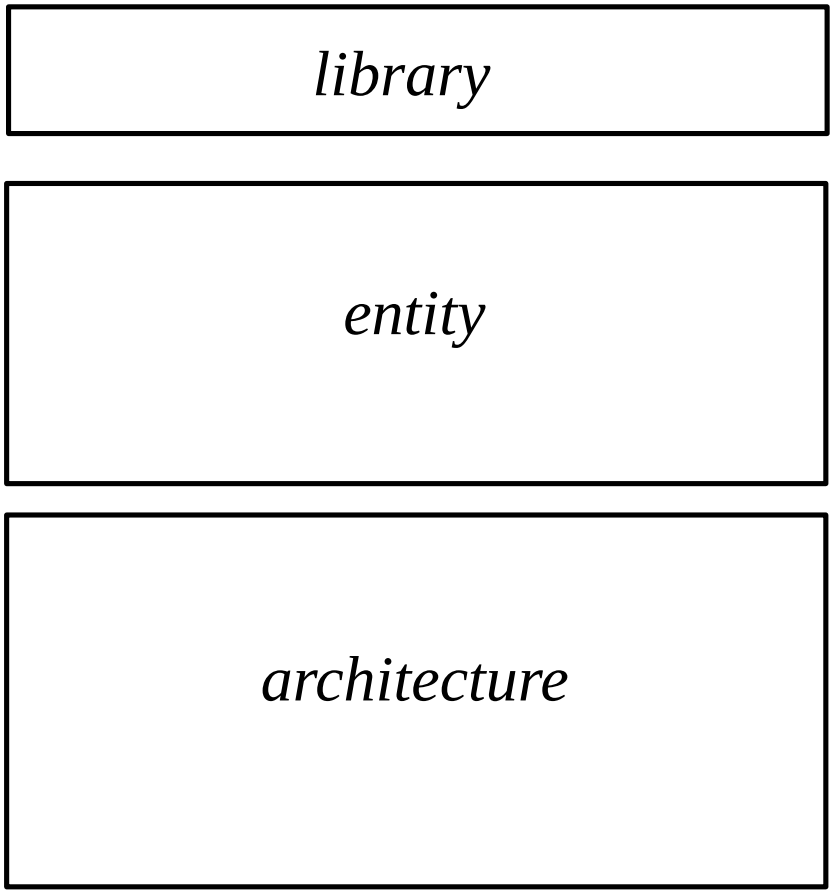
- http://standards.ieee.org/downloads/1076/1076-2008/

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;
```



Libraries and Packages

Selected arithmetic operators in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
+	signed, unsigned integer, natural	signed, unsigned	numeric_std
–	signed, unsigned integer, natural	signed, unsigned	numeric_std
*	signed, unsigned integer, natural	signed, unsigned	numeric_std
/	signed, unsigned integer, natural	signed, unsigned	numeric_std
rem	signed, unsigned integer, natural	signed, unsigned	numeric_std
mod	signed, unsigned integer, natural	signed, unsigned	numeric_std
+	signed, unsigned integer, std_logic	signed, unsigned std_logic_vector	std_logic_arith
–	signed, unsigned integer, std_logic	signed, unsigned std_logic_vector	std_logic_arith
*	signed, unsigned integer, std_logic	signed, unsigned, std_logic_vector	std_logic_arith
+	std_logic, integer, std_logic_vector	std_logic_vector	std_logic_unsigned
–	std_logic, integer, std_logic_vector	std_logic_vector	std_logic_unsigned
*	std_logic, integer, std_logic_vector	std_logic_vector	std_logic_unsigned

Selected relational operators in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
>	signed, unsigned integer, natural	boolean	numeric_std
<	signed, unsigned integer, natural	boolean	numeric_std
=	signed, unsigned integer, natural	boolean	numeric_std
>=	signed, unsigned integer, natural	boolean	numeric_std

Libraries and Packages

Function	Operand Types	Result Types	Package
<=	signed, unsigned integer, natural	boolean	numeric_std
/=	signed, unsigned integer, natural	boolean	numeric_std
>	signed, unsigned integer	boolean	std_logic_arith
<	signed, unsigned integer	boolean	std_logic_arith
=	signed, unsigned integer	boolean	std_logic_arith
>=	signed, unsigned integer	boolean	std_logic_arith
<=	signed, unsigned integer	boolean	std_logic_arith
/=	signed, unsigned integer	boolean	std_logic_arith
>	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
<	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
=	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
>=	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
<=	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
/=	std_logic_vector, integer	std_logic_vector	std_logic_unsigned

Libraries and Packages

Selected shift operators in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
shift_left	signed, unsigned, natural	signed, unsigned	numeric_std
shift_right	signed, unsigned, natural	signed, unsigned	numeric_std
rotate_left	signed, unsigned, natural	signed, unsigned	numeric_std
rotate_right	signed, unsigned, natural	signed, unsigned	numeric_std
sll	signed, unsigned, integer	signed, unsigned	numeric_std
srl	signed, unsigned, integer	signed, unsigned	numeric_std
rol	signed, unsigned, integer	signed, unsigned	numeric_std
ror	signed, unsigned, integer	signed, unsigned	numeric_std
shl	signed, unsigned integer	signed, unsigned	std_logic_arith
shr	signed, unsigned integer	signed, unsigned	std_logic_arith
shl	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
shr	std_logic_vector, integer	std_logic_vector	std_logic_unsigned

Libraries and Packages

Selected type conversion functions in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
to_integer	signed, unsigned	integer, natural	numeric_std
to_unsigned	std_logic_vector, natural	unsigned	numeric_std
to_signed	std_logic_vector, natural	signed	numeric_std
conv_integer	signed, unsigned std_ulogic, integer	integer	std_logic_arith
conv_signed	signed, unsigned std_ulogic, integer	signed	std_logic_arith
conv_unsigned	signed, unsigned std_ulogic, integer	unsigned	std_logic_arith
conv_std_logic_vector	signed, unsigned std_ulogic, integer	std_logic_vector	std_logic_arith
conv_integer	std_logic_vector	integer	std_logic_unsigned

Libraries and Packages

Selected type conversion functions in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
to_integer	signed, unsigned	integer, natural	numeric_std

- function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
  - Value cannot be negative since parameter is an UNSIGNED vector.
  - Result: Converts the UNSIGNED vector to an INTEGER
- function TO_INTEGER (ARG: SIGNED) return INTEGER;
  - Result: Converts a SIGNED vector to an INTEGER

Libraries and Packages

Selected type conversion functions in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
to_integer	signed, unsigned	integer, natural	numeric_std

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
 ...
variable int : integer;
variable unsigned_vec : UNSIGNED(0 to 7);
variable signed_vec : SIGNED(0 to 7);
 ...
int := TO_INTEGER(unsigned_vec);
int := TO_INTEGER(signed_vec);
```

Libraries and Packages

Selected type conversion functions in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
to_integer	signed, unsigned	integer, natural	numeric_std
conv_signed	signed, unsigned std_ulogic, integer	signed	std_logic_arith
conv_unsigned	signed, unsigned std_ulogic, integer	unsigned	std_logic_arith

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
 ...
variable int : integer;
variable std_vec : STD_LOGIC_VECTOR(0 to 7);
...
int := TO_INTEGER(conv_unsigned(std_vec));
int := TO_INTEGER(conv_signed(std_vec));
```

Library and Package declarations - syntax

```
LIBRARY  library_name;
USE library_name.package_name.package_parts;
```

CODING STYLES

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

Dataflow Modelling

- Functionality is implemented with concurrent statements
- Statements define the actual flow of data
- In statements, one can use logical operators as well as selected and conditional signal assingments

```
architecture dataflow of SR_flipflop is
begin

  gate_1 : q <= s_n nand q_n;
  gate_2 : q_n <= r_n nand q;

end architecture dataflow;
```

Behavioural Modelling

- Algorithmic presentation
  - No implementation information
  - Defines how the system is to behave

```
architecture checking of SR_flipflop is
begin

    set_reset : process (S, R) is
    begin
      assert S nand R;
      if S then
        Q <= '1';
      end if;
      if R then
        Q <= '0';
      end if;
    end process set_reset;

end architecture checking;
```

Structural Modelling

- System is implemented as a composition of subsystems
- Interconnection of components
  - Signals for internal connections

```
architecture struct of reg4 is
begin

  bit0 : entity work.edge_triggered_Dff(behavioral)
    port map (d0, clk, clr, q0);
  bit1 : entity work.edge_triggered_Dff(behavioral)
    port map (d1, clk, clr, q1);
  bit2 : entity work.edge_triggered_Dff(behavioral)
    port map (d2, clk, clr, q2);
  bit3 : entity work.edge_triggered_Dff(behavioral)
    port map (d3, clk, clr, q3);

end architecture struct;
```

```
architecture struct of reg4 is
begin

  bit0 : entity work.edge_triggered_Dff(behavioral)
    port map (d0, clk, clr, q0);
  bit1 : entity work.edge_triggered_Dff(behavioral)
    port map (d1, clk, clr, q1);
  bit2 : entity work.edge_triggered_Dff(behavioral)
    port map (d2, clk, clr, q2);
  bit3 : entity work.edge_triggered_Dff(behavioral)
    port map (d3, clk, clr, q3);

end architecture struct;
```

```
architecture dataflow of SR_flipflop is
begin

  gate_1 : q <= s_n nand q_n;
  gate_2 : q_n <= r_n nand q;

end architecture dataflow;
```

```
architecture checking of SR_flipflop is
begin

    set_reset : process (S, R) is
    begin
      assert S nand R;
      if S then
        Q <= '1';
      end if;
      if R then
        Q <= '0';
      end if;
    end process set_reset;

end architecture checking;
```

Concurrency

- Concurrent statements are evaluated at the same time; thus, the order of these statements doesn't matter
- Sequential/behavioural statements are executed in sequence

```
architecture dataflow of SR_flipflop is
begin

  gate_1 : q <= s_n nand q_n;
  gate_2 : q_n <= r_n nand q;

end architecture dataflow;
```

```
architecture checking of SR_flipflop is
begin

    set_reset : process (S, R) is
    begin
      assert S nand R;
      if S then
        Q <= '1';
      end if;
      if R then
        Q <= '0';
      end if;
    end process set_reset;

end architecture checking;
```

All VHDL processes execute concurrently

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