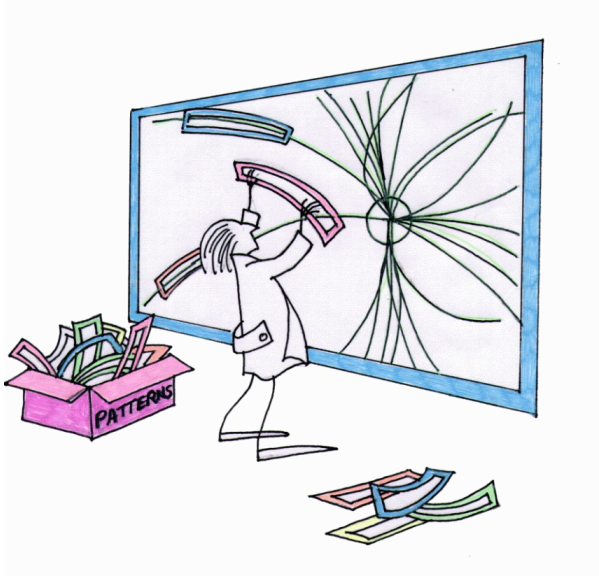
MARIE CURIE IAPP: FAST TRACKER FOR HADRON COLLIDER EXPERIMENTS

1ST SUMMER SCHOOL: VHDL BOOTCAMP

PISA, JULY 2013

Introduction to VHDL

Calliope-Louisa Sotiropoulou
PhD Candidate/Researcher
Aristotle University of Thessaloniki



AUTH e-LAB
Aristotle University of Thessaloniki-Electronics Laboratory

UNIVERSITÀ DI PISA

Introduction to VHDL

-

Introduction – VHDL

- What is VHDL?

A **V**ery **H**ard and **D**ifficult **L**anguage…

Introduction – VHDL

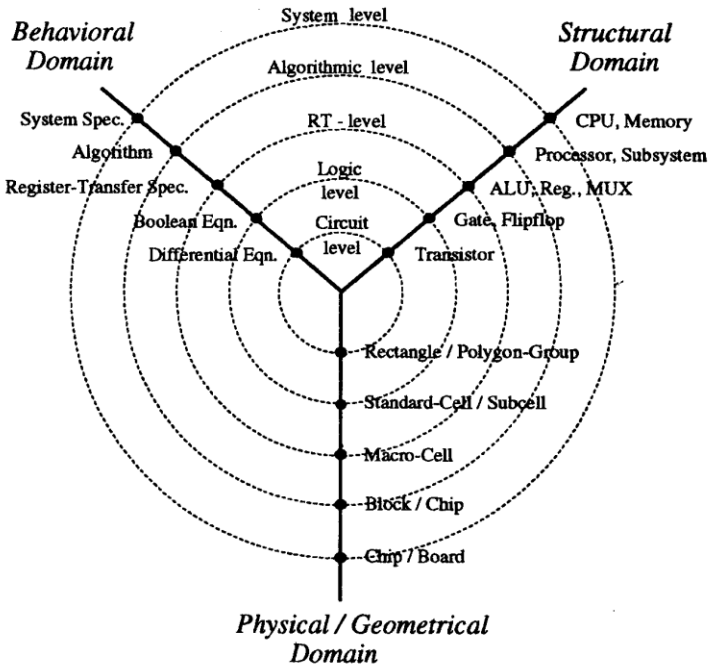
- **V**ery High Speed Integrated Circuit –
  Hardware **D**escription **L**anguage

- VHDL originated in the early 1980s
  - The American Department of Defense initiated the development of VHDL in the early 1980s
    - because the US military needed a standardized method of describing electronic systems
- VHDL was standardized in 1987 by the IEEE

Gajski's Y Chart

Each axis represents type of description

- Behavioral
  - Defines outputs as function of inputs
  - Algorithms but no implementation
- Structural
  - Implements behavior by connecting components with known behavior
- Physical
  - Gives size/locations of components and wires on chip/board
- Design process is illustrated by travel route



VHDL

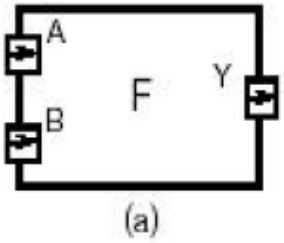
- VHDL is a programming language that allows one to model and develop complex digital systems in a dynamic environment.
- Allows you to designate in/out ports (bits) and specify behavior or response of the system.
- With VHDL we can design a system/circuit → Something that physically exists
- Or we can model a system's/circuit's behavior

Additional Benefits of VHDL

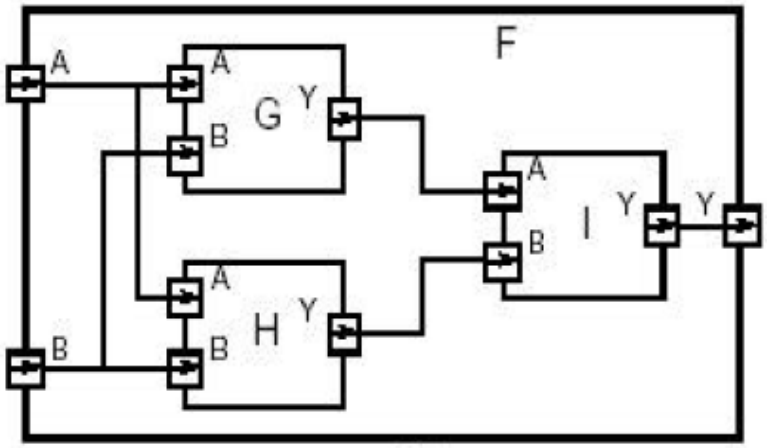
- Allows for various design methodologies :
  Top down, bottom up, delay of detail
- Very flexible in its approach to describing hardware
- Provides technology independence:
  VHDL is independent of any technology or process (ASIC, FPGA...)
  However VHDL code can be written and then targeted for many different technologies.
- Describes a wide variety of digital hardware
- Mix descriptions: behavioural with gate level description

VHDL - Hierarchy

- Allows Top-Down Design → Hierarchy
- A hierarchical design consists of modules which consist of submodules, VHDL code or combination of both
- Goal: Reduce the complexity of the design and allow for easier management
- Result: Faster and more effective development of complicated systems



(a)

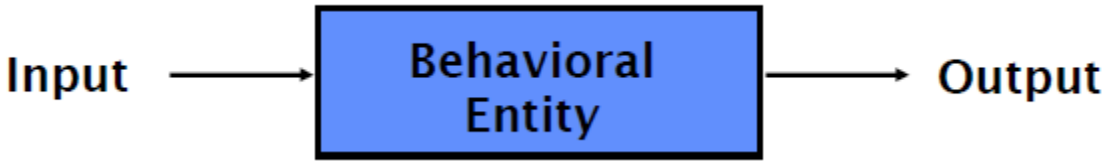


(b)

VHDL - Hierarchy

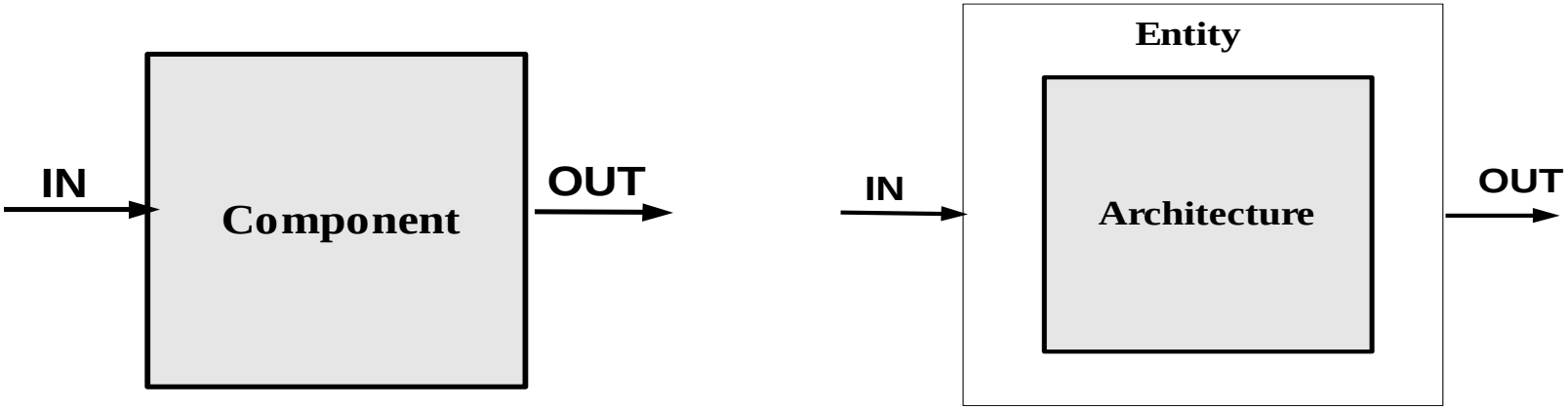
- Black Box Principle:
  At every hierarchy level only the absolutely necessary information is disclosed

- Input/Output Ports and their behavior



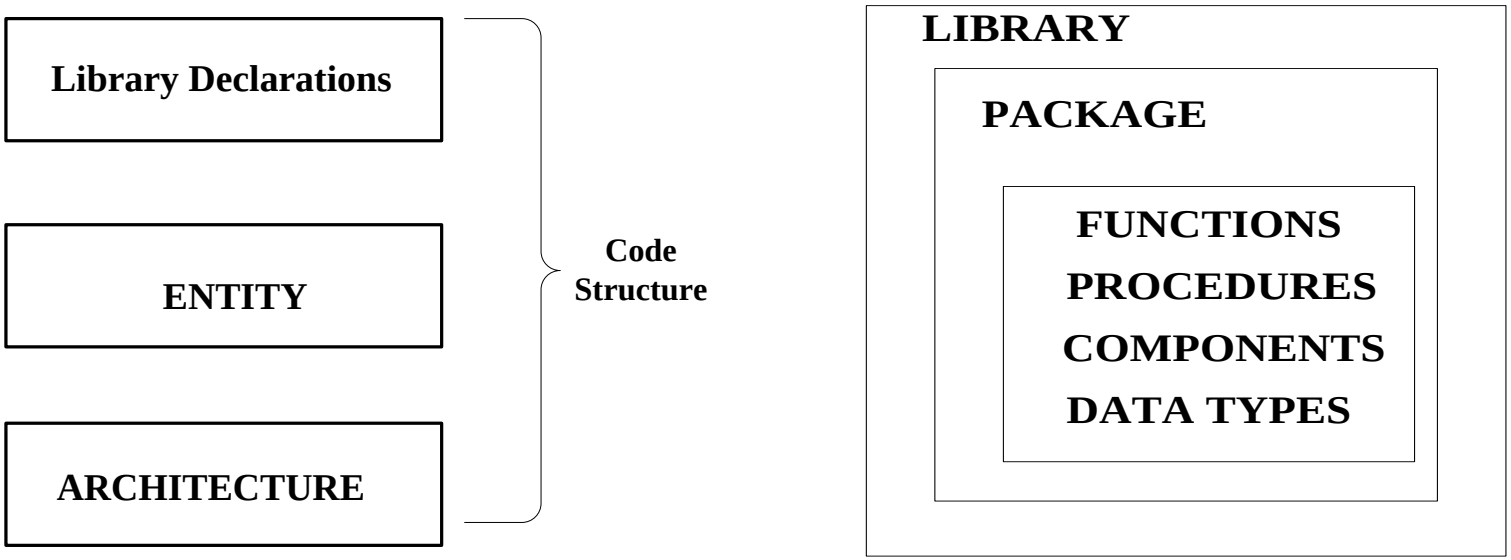
VHDL – Basic Structure

- Every component consists of two parts:
  - Entity: Input/Output ports and generics
  - Architecture: Description of the component's operation in either behavioral or structural level
  - We can even have an entity with multiple architectures

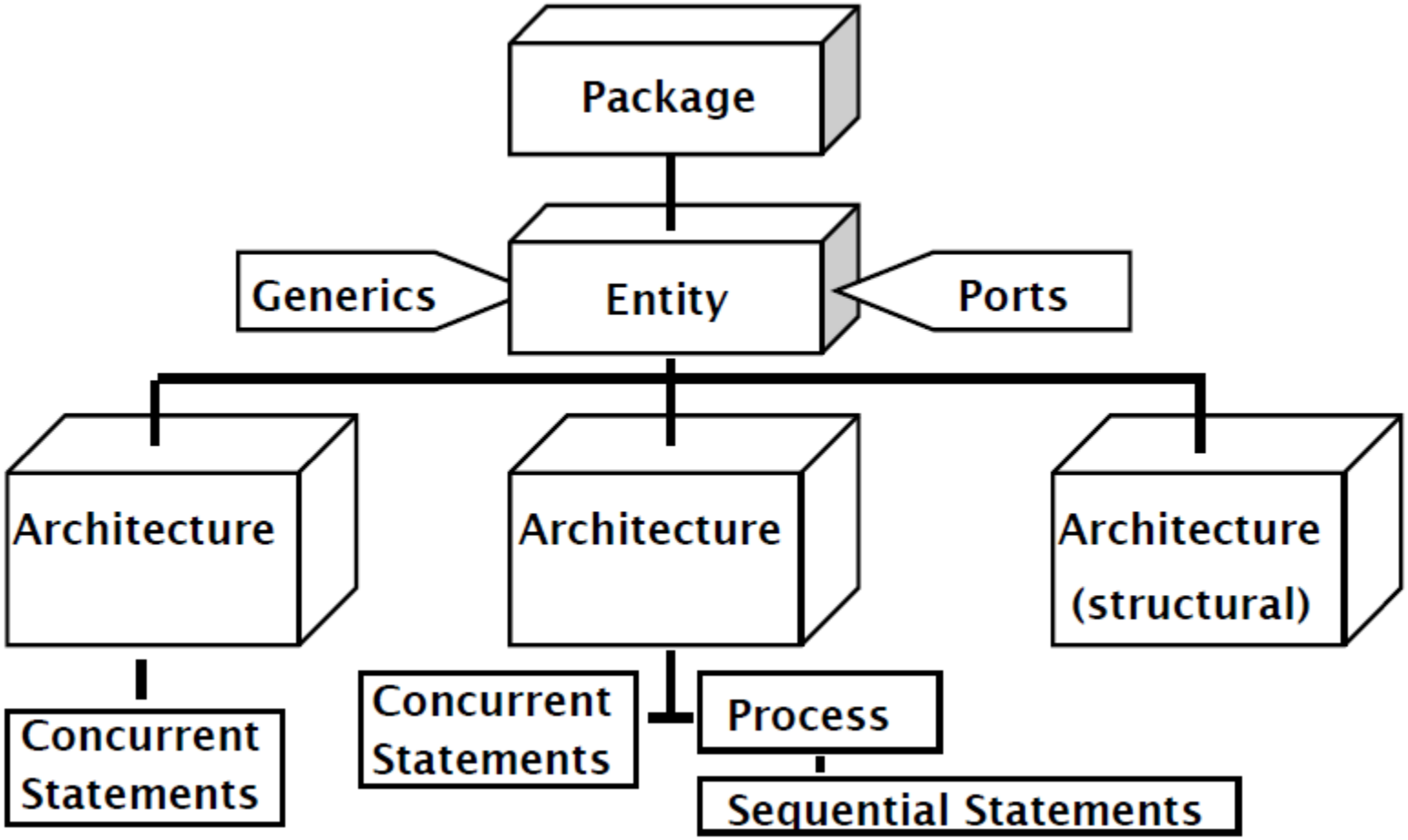


VHDL – Code Structure

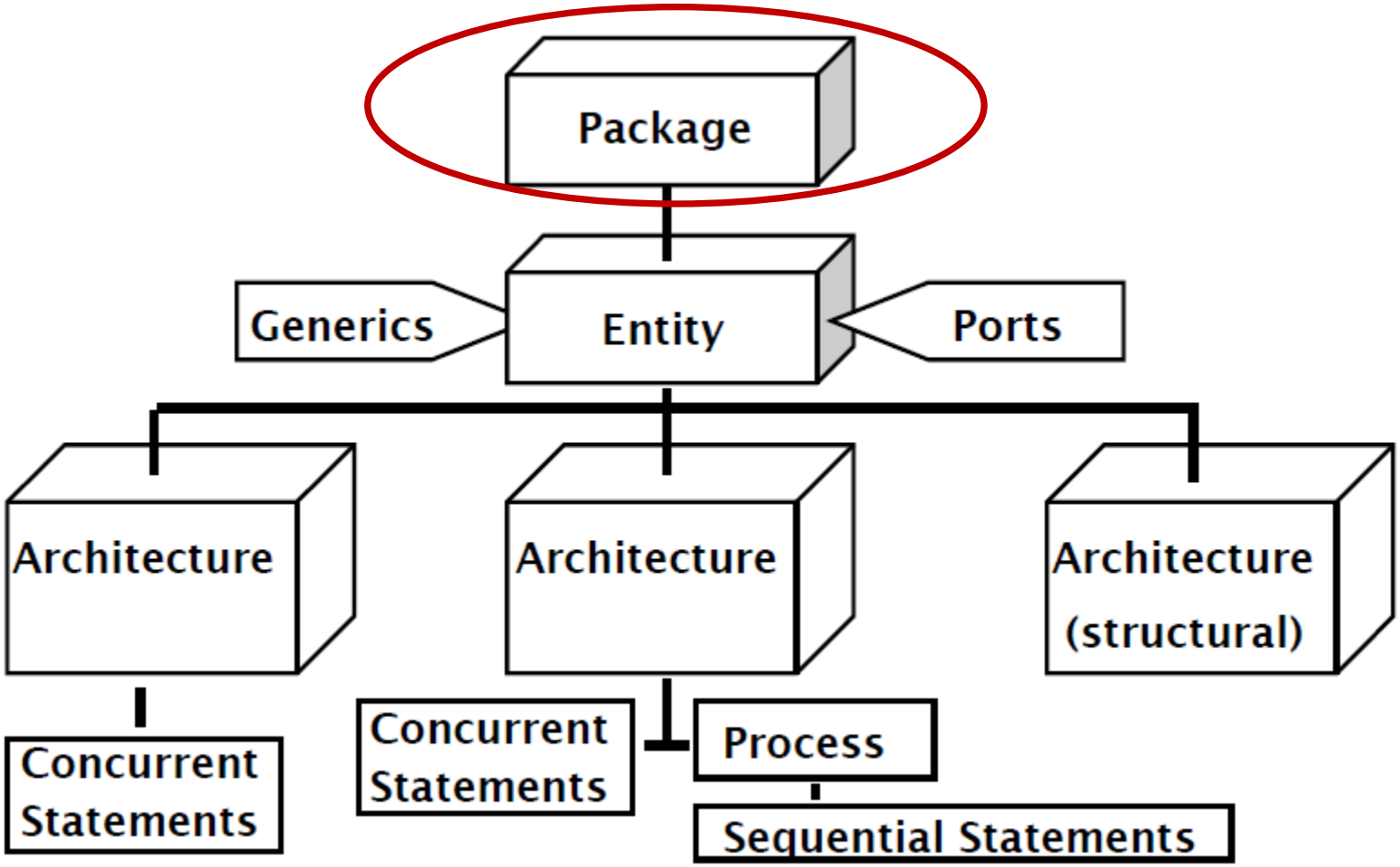
- Basic elements of code structure:
  - **Library declarations** : The list of libraries/packages which will be used for the code
  - **Entity**: Defines the input/output ports
  - **Architecture**: Defines the component operation



VHDL – Basic Structure



VHDL – Basic Structure



Library Declarations

- Declaring and using a VHDL library
- **LIBRARY** <library name >;
- **USE** <library name >. <package name>. <package parts>;

- Usually at least three packages from three libraries are needed: *ieee.std_logic_1164* (*lib. ieee*), *standard* (*lib. std*), *work*
- **LIBRARY** ieee;
- **USE** ieee.std_logic_1164.all;

- **LIBRARY** std;
- **USE** std.standard.all;

- **LIBRARY** work;
- **USE** work.all;

Library Declarations

Libraries ***std*** and ***work*** are always visible and do not need to be declared

- Include *library ieee;* before entity declaration
- ieee.std_logic_1164 defines a standard for designers to use in describing interconnection data types used in VHDL modeling
- The ***ieee*** library needs to be declared when the **std_logic (std_ulogic)** data type is used
- ieee.numeric_std provides a set of arithmetic, conversion, comparison functions for signed, unsigned, std_ulogic, std_logic, std_logic_vector

Library Declarations (numeric_std)

- There are two basic arithmetic libraries:
  - The std_logic_arith:
    It was created by Synopsis and it was included in their IEEE VHDL packages → In the future it will not be supported (someday)
    Usually combined with the unsigned and signed libraries. The problem is only one type (signed or unsigned logic) can be used in every entity
  - The numeric_std:
    The official IEEE library
    Numeric_std does not attempt to imply a numerical interpretation on SLV(std_logic_vector) , but rather defines related types UNSIGNED and SIGNED which have both numerical and bitwise interpretations

Signed/Unsigned Logic (Just in case…)

- A numeric variable is *signed* if it can represent both positive and negative numbers, and *unsigned* if it can only represent non-negative numbers (zero or positive numbers).
- For example:
  an 8bit integer variable if it is unsigned can represent values from 0→255
  an 8bit integer variable if it is signed can represent values from -128→127

The std_logic and std_logic_vector types

- STD_LOGIC is defined in the library **std_logic_1164**.This is a nine valued logic system
- It has 9 values: 'U', 'X', '0', '1', 'Z', 'W', 'L' ,'H' and '-'.
  The meaning of each of these characters are:
  U = uninitialized (unresolved)
  X = unknown - a multisource line is driven '0' and '1' simultaneously (*)
  0 = logic 0
  1 = logic 1
  Z = high impedance (tri state)
  W = weak unknown
  L = weak "0"
  H = weak "1"
  - = dont care

  Type **std_logic is unresolved** type because of 'U','Z' etc.

Resolving the std_logic type

- Resolving signal values when they are driven by more than one source

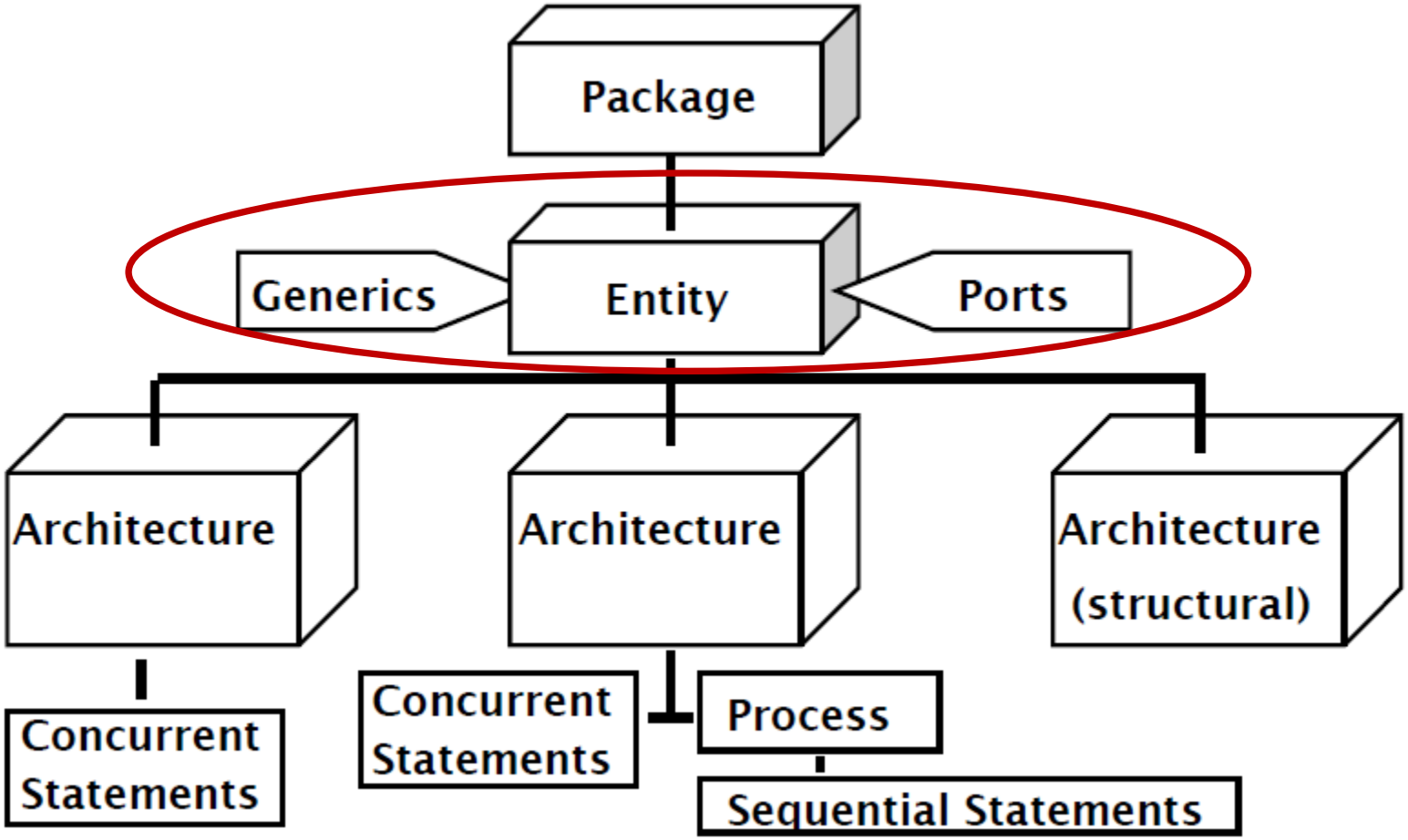
	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

This is probably a good time to…

Ask you to refresh your memory on…

- Binary numeral system
- Hexadecimal numeral system
- Boolean logic

VHDL – Basic Structure



Entity

Defines the communication (input/output ports) of the design with its environment.

Declares the:

- ENTITY name which is the same as the component name
- The INPUT/OUTPUT ports/signals
- Also the GENERIC → values that need to be passed to the entity by the environment

Syntax:

```
Entity  <identifier_name> is
 port  ( [signal] <identifier> : [mode] <type_indication> ;
          [signal] <identifier> : [mode] <type_ indication >) ;
 end  [<identifier_name>]   ;
```

Port Modes

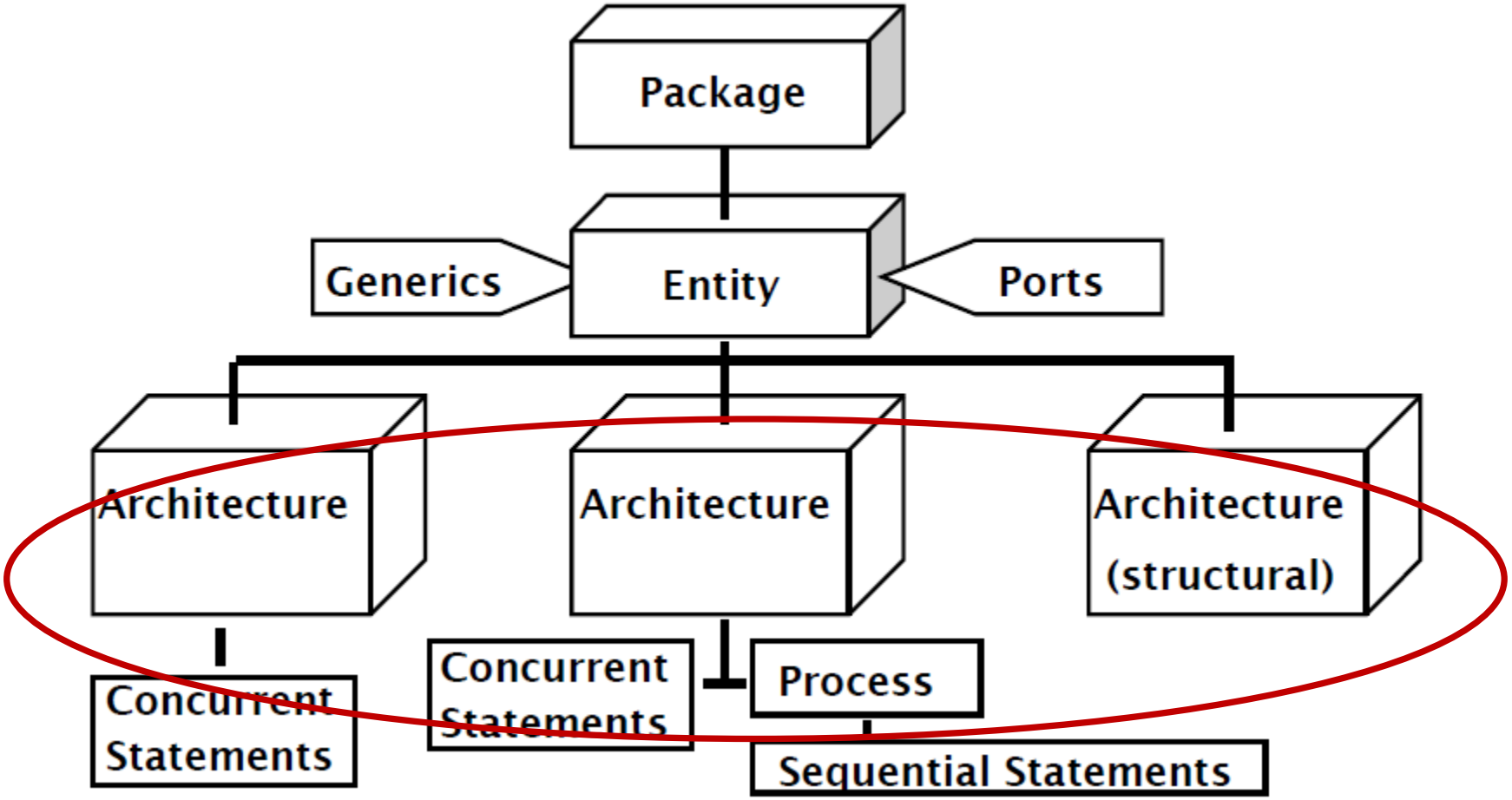
- **In:** input port
  A variable or a signal can read a value from a port of mode **in,** but is not allowed to assign a value to it.
- **Out:** output port
  It is allowed to make signal assignments to a port of the mode **out,** but it is not legal to read from it.
- **Inout:** bi-directional port
  Both assignments to such a port and reading from it are allowed.
- **Buffer:** output port with read capability
  It differs from **inout** in that it can be updated by at most one source, whereas **inout** can be updated by zero or more sources.
- **Linkage:**
  The value of the port may be read or updated, but only by appearing as an actual corresponding to an interface object of mode linkage.

Generics

- **Generics** are a means of passing specific information into an entity. They do not have a mode (direction):

```
entity PARITY is
generic (N : integer);
port (
  A : in std_logic_vector (N-1 downto 0);
  ODD : out std_logic
);
end PARITY;
```

VHDL – Basic Structure



Architecture

Defines the operation of the component/design

Consists of:

- the declarations of the internal signal/variables etc.
- the description of the **component operations**

Syntax:

Architecture <architecture_name> **of** <entity_identifier> **is**

[declarative_part]

begin

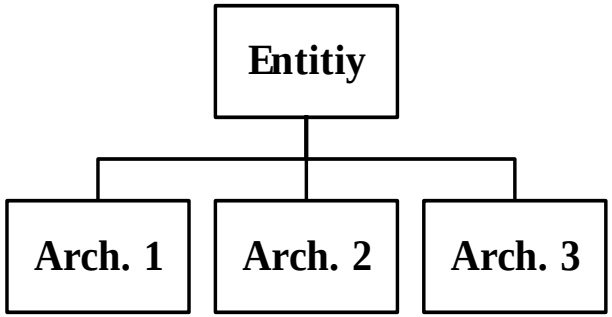
[architecture_statement_part]

end <architecture_name>;

Architecture

Every component can have more than one architectures but only one can be active at a time:

```
Entity nand is
 port ( a, b : in   std_logic;
        c    : out std_logic);
end nand ;
```




```
Architecture nand_behav of nand is
 begin
 c <= a nand b ;
end nand ;
```

```
Architecture nand_struct of nand is
Signal int : std_logic;
 begin
int <= a and b;
c <= not int ;
 end nand ;
```

One bit adder

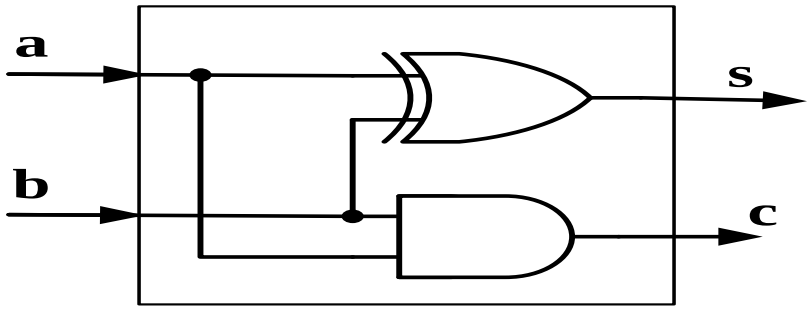
```
library ieee;
use ieee std_logic_1164.all;

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

architecture rtl of HA is
begin
s <= a xor b;
c <= a and b;
end rtl;
```

Library Declarations

ENTITY



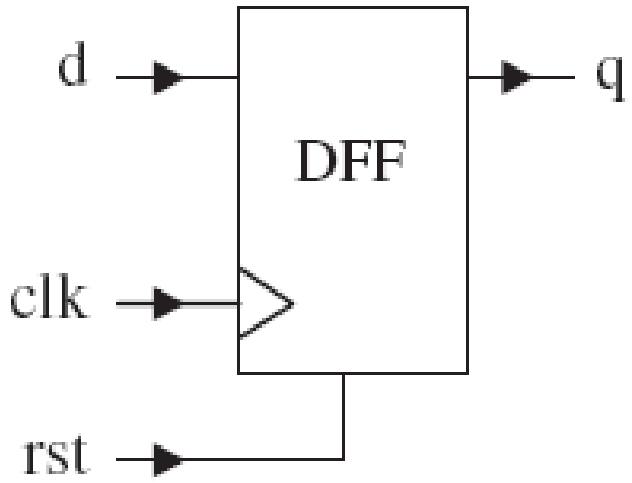
ARCHITECTURE

D-flip flop with an asynchronous reset

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

entity dff  is
port ( d, clk, rst : in   std_logic;
         q                 : out std_logic);
end dff ;

architecture behavior of dff is
begin
    process (rst, clk)
         begin
            if (rst='1') then;
               q<='0';
            elsif (clk'event and clk='1');
               q<=d;
            end if;
    end process;
end behavior;
```



Classes – Object - Data Types

- Every object belongs to a class and has a specific data type

- Classes: signals, variables, constants

Class	Object	Data Type
Signal	a:	std_logic

Data Types

Predefined data types

- Bit and bit_vector (package: *standard* – lib: *std*)
- Std_logic and Std_logic_vector (pack. *std_logic_1164* – lib.*ieee*)
- Boolean (package: *standard* – lib: *std*)
- Integer (package: *standard* – lib: *std*)
- signed/unsigned (package: *numeric_std* – lib: *ieee*)
- Natural (synthesizable)

- Real (non synthesizable) (package: *standard* – lib: *std*)
- Physical (non synthesizable)
- ASCII (non synthesizable)

User Defined Data Types

Arrays

VHDL is very strict with data compatibility

Vectors

```
architecture rtl of ex is
signal a,b, c : std_logic_vector (2 downto 0);
signal d      : std_logic_vector (0 to 2);
Begin
        a <= d;
        b <= c;
End;
```

```
a(2) <= d(0);    b(2) <= c(2);
a(1) <= d(1);    b(1) <= d(1);
a(0) <= d(2);    b(0) <= d(0);
```

When values are assigned to a vector the size and the direction of the vector must be taken into account

VHDL Operators

- Value assignment operators:
  Used to assign values to SIGNALS, VARIABLES and CONSTANTS
  - '<=' for SIGNALS
  - ':=' for VARIABLES, CONSTANTS, GENERIC and initial values
  - '=>' for component signals and for the OTHERS command

```
signal x : std_logic;
variable y : std_logic_vector (3 downto 0);
signal w : std_logic_vector (0 to 7);
…
x <='1';
y := "0010";
w <= (0=>'1', OTHERS => '0')
```

VHDL Operators

operator	description	data type of operand a	data type of operand b	data type of result
`a ** b`	exponentiation	`integer`	`integer`	`integer`
`abs` `a`	absolute value	`integer`		`integer`
`not` `a`	negation	`boolean`, `bit`, `bit_vector`		`boolean`, `bit`, `bit_vector`
`a * b`	multiplication	`integer`	`integer`	`integer`
`a / b`	division			
`a` **`mod`** `b`	modulo			
`a` **`rem`** `b`	remainder			
`+ a`	identity	`integer`		`integer`
`- a`	negation			
`a + b`	addition	`integer`	`integer`	`integer`
`a - b`	subtraction			
`a & b`	concatenation	1-D array, element	1-D array, element	1-D array

VHDL Operators

a **sll** b	shift left logical	bit_vector	integer	bit_vector
a **srl** b	shift right logical			
a **sla** b	shift left arithmetic			
a **srl** b	shift right arithmetic			
a **rol** b	rotate left			
a **ror** b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a **and** b	and	boolean, bit, bit_vector	same as a	boolean, bit, bit_vector
a **or** b	or			
a **xor** b	xor			
a **nand** b	nand			
a **nor** b	nor			
a **xnor** b	xnor			

VHDL Operators (Overloaded for Numeric_std)

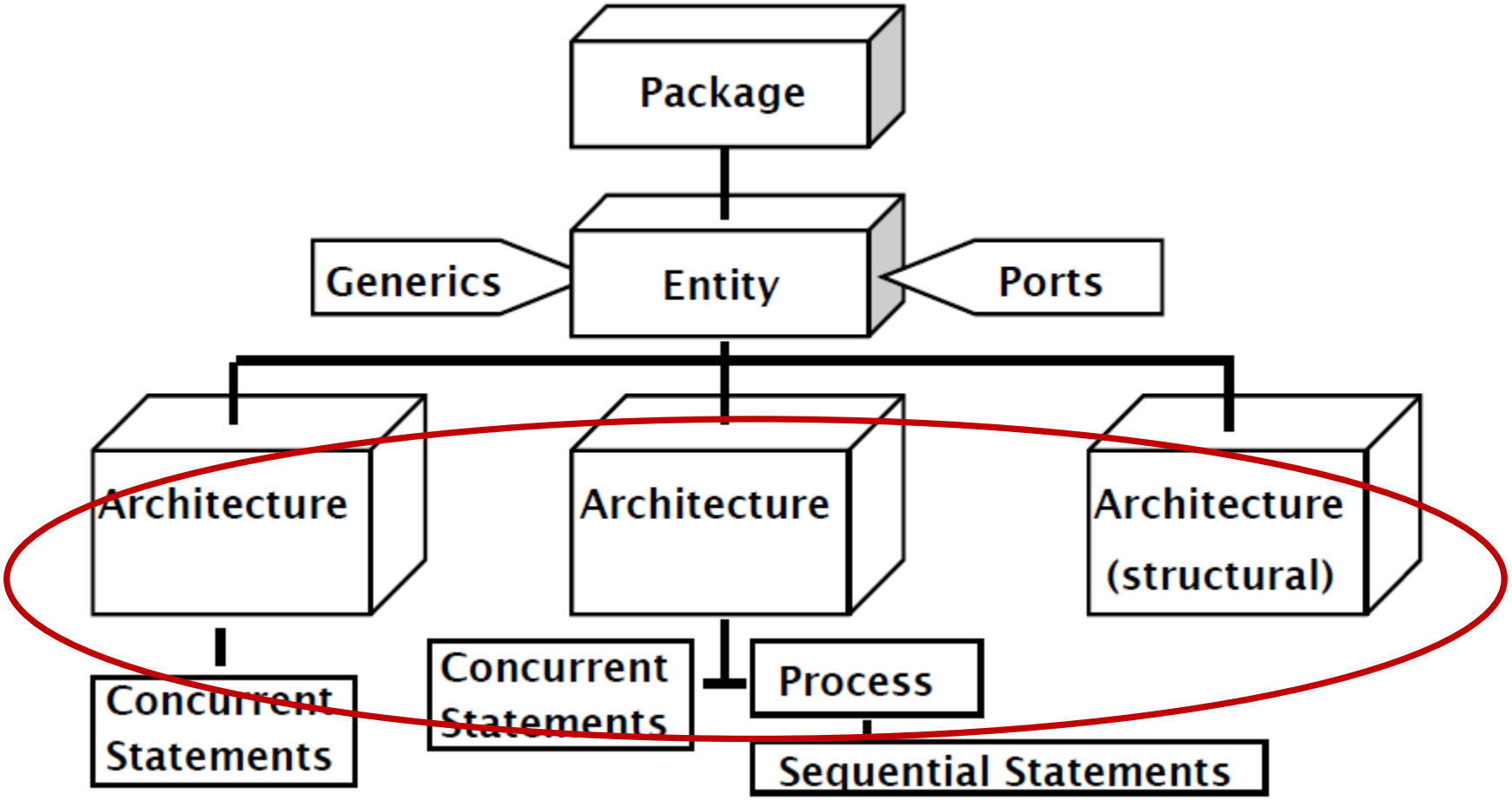
overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a – a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a – b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

VHDL Operators (Overloaded for Numeric_std)

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b) shift_right(a,b) rotate_left(a,b) rotate_right(a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize(a,b) std_match(a,b)	resize array compare '-'	unsigned, signed unsigned, signed std_logic_vector, std_logic	natural same as a	same as a boolean
to_integer(a) to_unsigned(a,b) to_signed(a,b)	data type conversion	unsigned, signed natural integer	 natural natural	integer unsigned signed

data type of a	to data type	conversion function / type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
unsigned, std_logic_vector	unsigned	unsigned(a)
unsigned, signed	std_logic_vector	std_logic_vector(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

VHDL – Basic Structure



Concurrent vs Sequential VHDL

- VHDL provides two different types of execution: sequential and concurrent.
- Different types of execution are useful for modeling of real hardware.
  - Supports various levels of abstraction.
- Sequential statements view hardware from a "programmer" approach.
- Concurrent statements are order-independent and asynchronous.

Concurrent VHDL

- Hardware operates in parallel
- Therefore structures are needed to model this behaviour
  - Concurrent statements
  - Concurrent objects - Signals

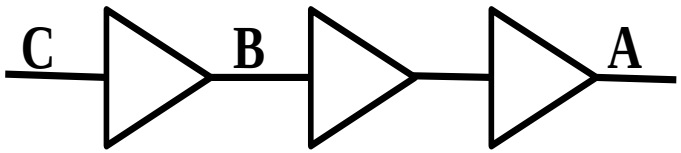
VHDL signals model actual connections (wiring and buses)

Concurrent VHDL

- The sequence of the signal assignments in concurrent VHDL is irrelevant

```
Architecture example of ex is
begin
    a <= b;
    b <= c;
end example;
```

```
Architecture example of ex is
begin
    b <= c;
    a <= b;
end example;
```



Concurrent Statements

- In concurrent code can be used:

  - Operators

  - When (When/Else ή With/Select/When)

  - Generate

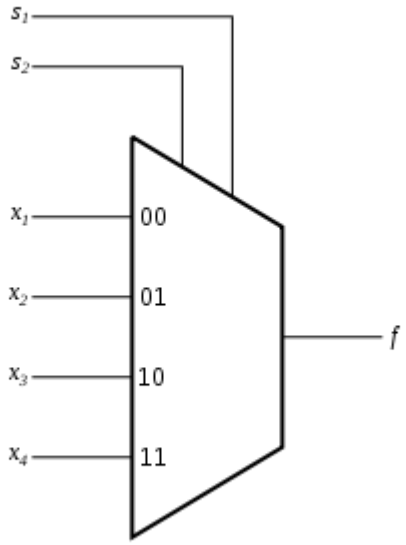
  - Block

- Usually for more complicated designs which must be described behaviourally sequencial code is preferred

4-to-1 Multiplexer

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (a, b, c, d, s0, s1: in std_logic;
            y: out std_logic);
end mux;

architecture operators of mux is
begin
   y <= (a and not s1 and not s0) or
        (b and not s1 and s0) or
        (c and s1 and not s0) or
        (d and s1 and s0);
end operators ;
```



Concurrent statements - When

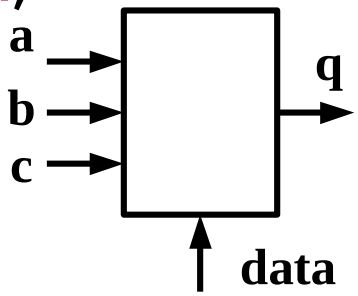
Syntax:

<target> <= <expres.> [**after** <expres.>] **when** <expres.> **else**
<expres.> [**after** <expres.>] … ;

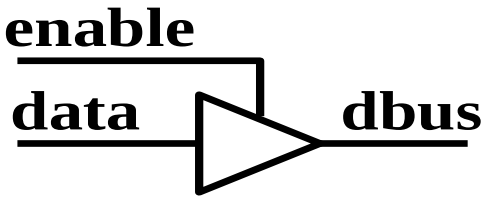
- The sequence of the arguments is very important because the code inside the **when** statement is sequential

Concurrent statements - When

```
Architecture rtl of ex is
begin
q <= a when data ="00"
else
        b when data ="11"
else
        c;
end;
```




```
Architecture rtl of three_state is
begin
dbus <= data when enable = '1' else
          'Z';
end;
```




```
dbus <= data when enable =
'1'
 else  (others =>'Z');
```

Concurrent statements - With

Syntax:

<with> <expression> **select**
<target> <= <expression> **when** <chose>;

```
entity example is
port ( a,b,c : in  std_logic;
   data  : in  std_logic_vector (1 downto 0);
   q     : out std_logic);
end example;
```

```
architecture rtl of example is
begin
with data select
q <= a when "00" ,
     b when "11" ,
     c when others;
end;
```

All signal combinations must be enumerated

- Use of when others for the rest of the cases

Less flexible than when

- Allows only one expression

4-to-1 Multiplexer with when and with

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port ( a, b, c, d: in std_logic;
          sel: in std_logic_vector (1 downto 0);
          y: out std_logic);
end mux;
```

```
architecture mux1 of mux is
begin
  y <=a when sel="00" else
      b when sel="01" else
      c when sel="10" else
      d;
end mux1;
```

```
architecture mux2 of mux is
begin
   with sel select
      y <= a when "00",
           b when "01",
           c when "10",
           d when others;
end mux2;
```

Sequential VHDL

- Sequential structures are executed in the sequence they appear in the VHDL code
- They are used for sequential data processing

- Sequential Commands:
  - IF
  - WAIT
  - CASE
  - LOOP

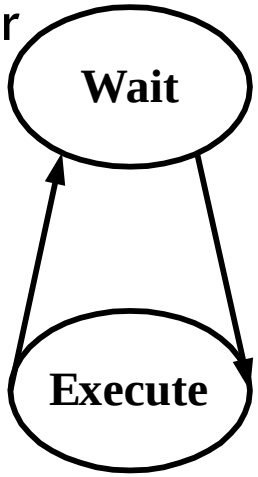
Sequential Structures

- If-then-else statement
- Case statement
- Variable declaration
- Loop statement
- Return statement
- Null statement
- Wait statement

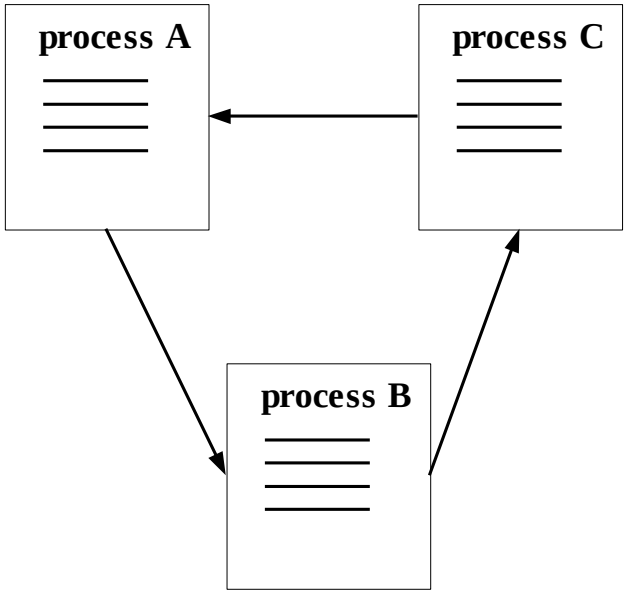
```
architecture rtl of ex is
concurrent declaration part
begin
concurrent VHDL
process (…)
sequential declaration part
   begin
            sequential VHDL
   end process;
concurrent VHDL
end;
```

VHDL Process

- The **Process** is a concept which derived from computer programming
- It operates in two different states:
  - Wait
  - Execute



- More than one process can be executed in parallel



VHDL Process - Syntax

```
[<process_name> :] process [ (sensitivity_list)]
        [<process_declarative_part>]
begin
        <process_statement_part>
end process [<process_name>];
```

- The **Process Execution** starts when a signal from the sensitivity list changes value

- There are two kinds of Process statements
  - Combinational
  - Clocked

Combinational Processes

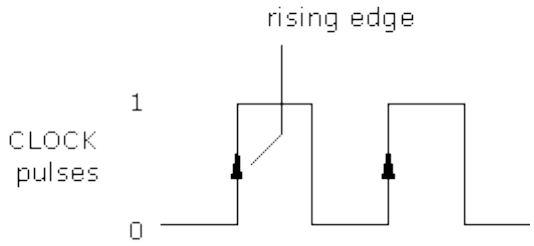
```
process (a, b, s)
begin
        if (s='1') then
                y <= a;
        else y <= b;
end process;
```

- The above code leads to a 2-to-1 mux

- In the combinational processes all the input signals must be in the sensitivity list
- All the input/output combinations must be covered by the code or unwanted latches may appear

Clocked Processes

- Used for modeling synchronous sequential circuits
- A change in the value of the **clock signal** is required to initiate a clocked process
- A synchronous sequential design changes state only when the clock signal changes value (**clock edges**)



Clocked Processes

- DFF → The D-type flip-flop samples an incoming signal at the rising (or falling) edge of a clock. This example has an asynchronous, active-high reset, and samples at the rising clock edge.

```
DFF : process(RST, CLK)
  begin
  if RST = '1' then
    Q <= '0';
  elsif rising_edge(CLK) then
     Q <= D;
  end if;
 end process DFF;
```

```
DFF : process(RST, CLK)
Begin
if RST = '1' then
   Q <= '0';
 elsif CLK'event and CLK = '1' then
   Q <= D;
end if;
end process DFF;
```

And now…

- Time for a break???

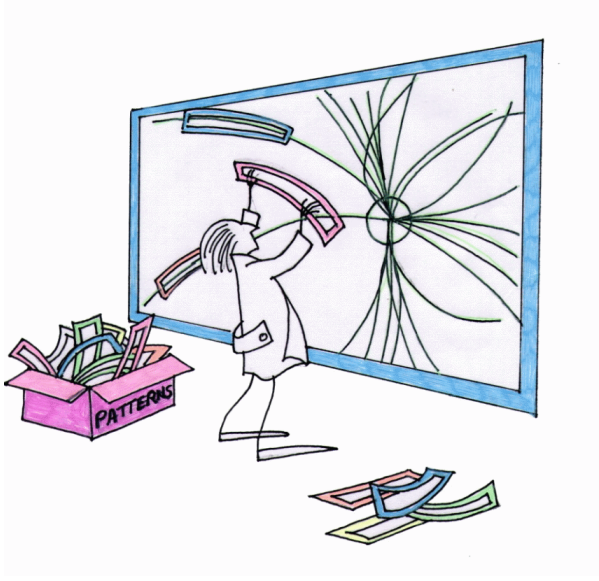
MARIE CURIE IAPP: FAST TRACKER FOR HADRON COLLIDER EXPERIMENTS

1ST SUMMER SCHOOL: VHDL BOOTCAMP

PISA, JULY 2013

Lab 1: Writing your first ever VHDL code

Calliope-Louisa Sotiropoulou
PhD Candidate/Researcher
Aristotle University of Thessaloniki



AUTH e-LAB
Aristotle University of Thessaloniki-Electronics Laboratory

IN SUPREMÆ DIGNITATIS
1343
UNIVERSITÀ DI PISA

Lab 1: Writing your first ever VHDL code

Writing VHDL code

- VHDL is of course plain text, so you just need a plain text editor to write it
- So what you need is a good editor of your choice (notepad++, ultraedit, emacs etc.) with the VHDL language template loaded
- The real issue is “compiling” the code which totally depends on the target platform of the implementation
- Is it an ASIC? Is it an FPGA? Whose vendor FPGA is it?

Xilinx Tools – Where and how…

- Xilinx offers a free version of their tool, the XILINX ISE Webpack
- You can also have a trial version for 30 days of the full version
- You need to make a Xilinx account to download the software (www.xilinx.com)

Xilinx ISE Project Navigator

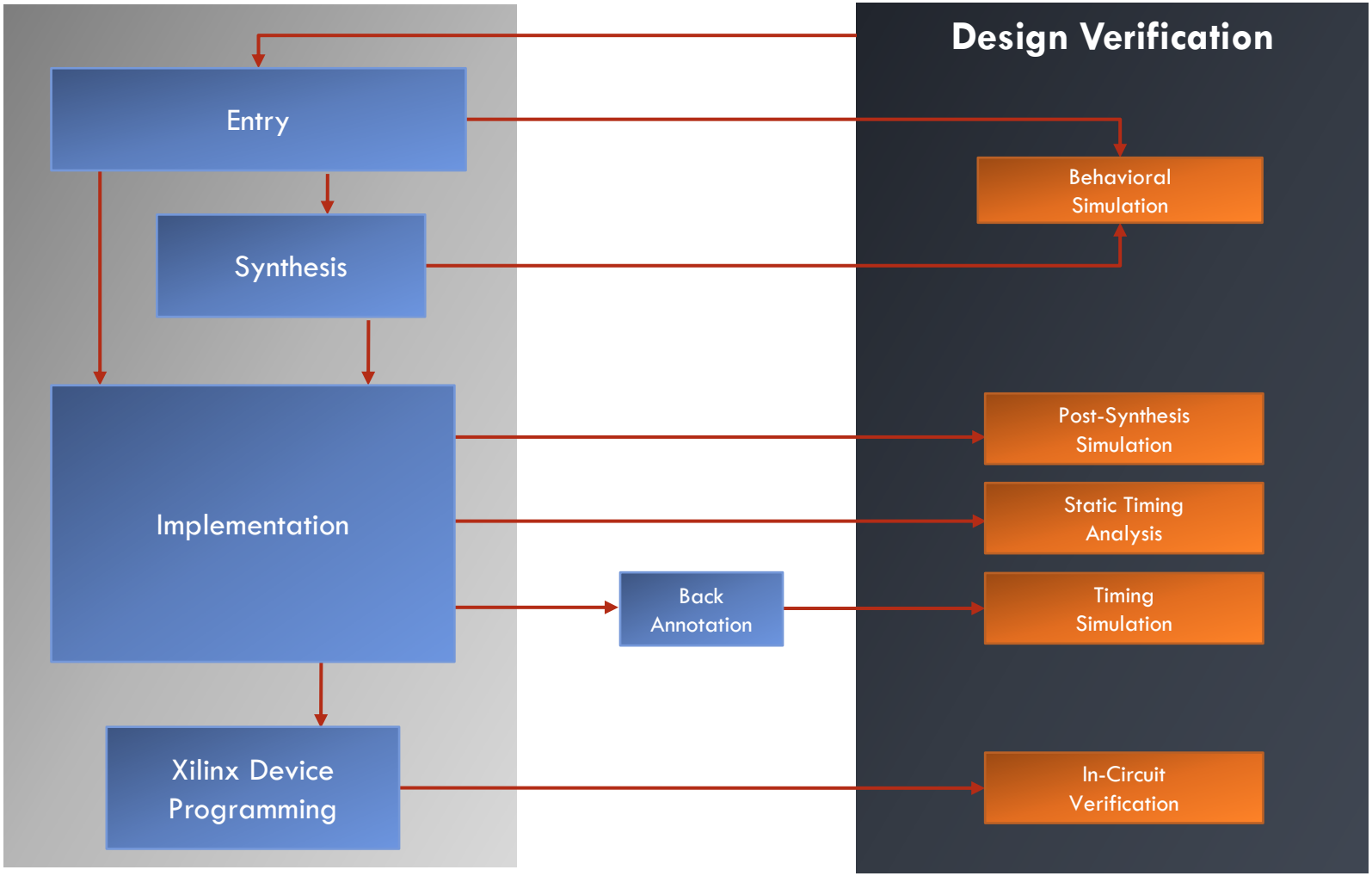
- So now we move to the computers…

Finally…..

XILINX ISE Design Suit

- The *ISE Design Suite* is an Electronic Design Automation (EDA) tool for the FPGA families produces by Xilinx Inc.
- It supports Verilog and VHDL design entry, place and route (PAR), verification and debug and bitstream generation for device programming.
- The ISE software controls all aspects of the design flow.
- Through the Project Navigator interface, you can access all of the design entry and design implementation tools.
- You can also access the files and documents associated with your project.

Xilinx ISE Design Flow



XILINX Synthesis

- During HDL synthesis, XST analyzes the HDL code and attempts to infer specific design building blocks or macros (such as MUXes, RAMs, adders, and subtractors) for which it can create efficient technology implementations.
- To reduce the amount of inferred macros, XST performs a resource sharing check.→ This usually leads to a reduction of the area as well as an increase in the clock frequency.
- Finite State Machine (FSM) recognition is also part of the HDL synthesis step. XST recognizes FSMs independent of the modeling style used.
- To create the most efficient implementation, XST uses the target optimization goal, whether area or speed, to determine which of several FSM encoding algorithms to use.