

Digital Design and HDL

Mrs. Anjali Pise
Asst Prof
E&TC Dept.
SKNSCOE Korti

Course Outcomes

Course Name	:Digital Design and HDL (ET312)	Course Code:	: ET 312
Class	: TY	SEMESTER	: I
Academic Year	: 2021-22	Subject Teacher	: Prof. A. C. Pise
CO No.	Course Outcome Statements	Cognitive Level	
CO1	Explain different syntax of HDL language.	L2: Understand	
CO2	Model combinational logic circuits using VHDL and Verilog.	L3: Apply	
CO3	Model sequential logic circuits using VHDL.	L3: Apply	
CO4	Describe architecture and internal components of CPLD, FPGA, ASIC and SOC and compare them.	L2: Understand	
CO5	Explain different testing methods for combinational Logic, sequential logic, IC and write test bench for simple combinational circuits.	L2: Understand	

Outline

Introduction



Concepts and History of VHDL



VHDL Models of Hardware



VHDL Basics



Summary

Course Goals

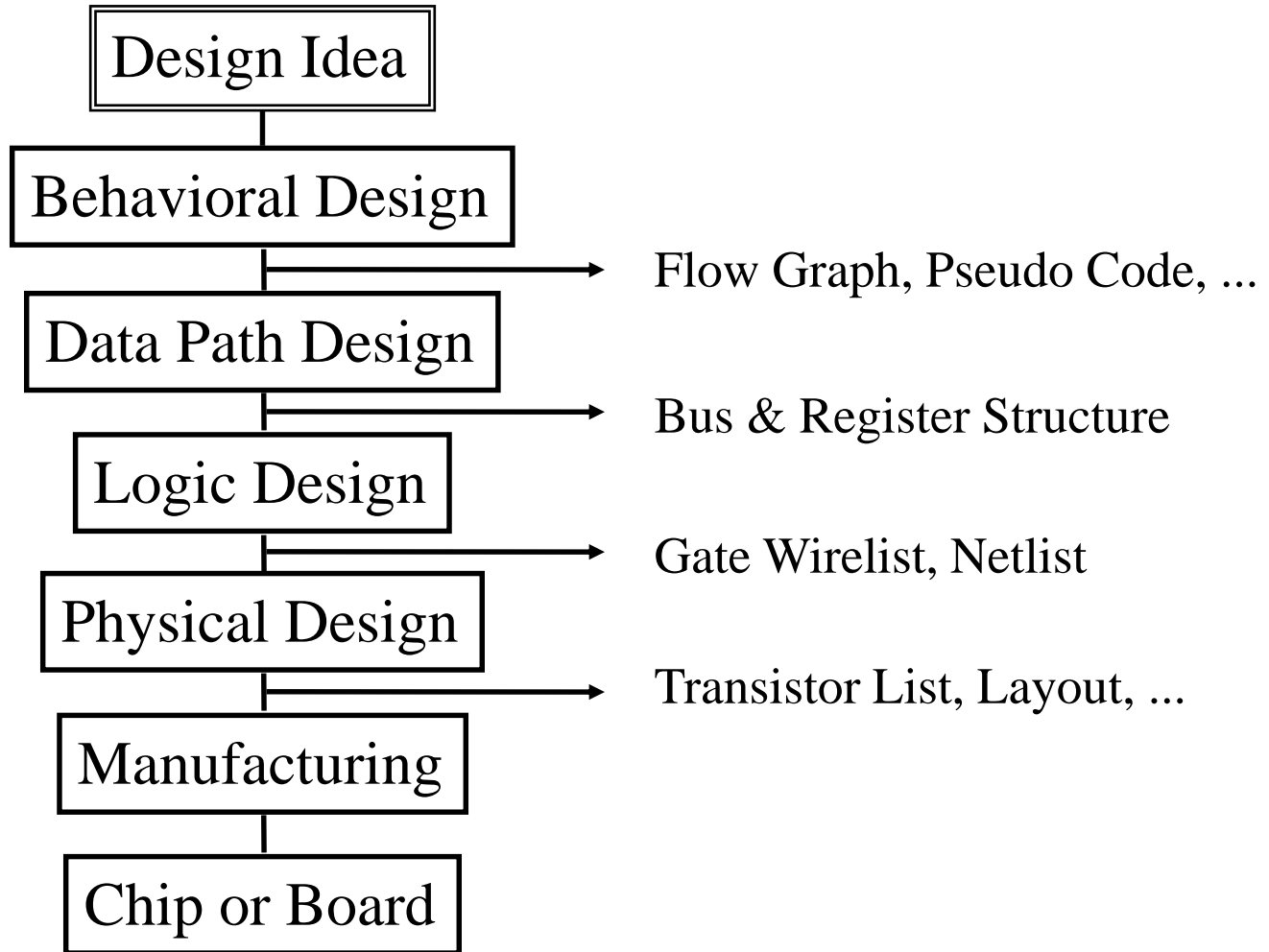
Comprehension of VHDL Basic Constructs

Familiarity with VHDL design descriptions

Understanding of the VHDL Timing Model

Introduction

Digital systems design process



Sample Design Process

* Problem

Design a single bit half adder with carry and enable

* Specifications

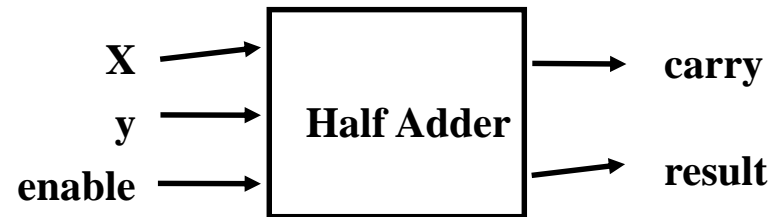
- Passes results only on enable high
- Passes zero on enable low
- Result gets x plus y
- Carry gets any carry of x plus y



Behavioral Design

- * Starting with an algorithm, a high level description of the adder is created.

```
IF enable = 1 THEN  
    result = x XOR y  
    carry = x AND y  
ELSE  
    carry = 0  
    result = 0
```

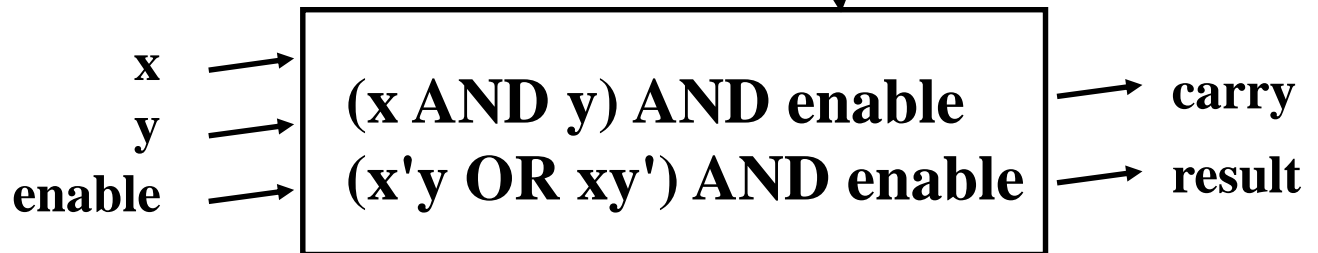


- * The model can now be simulated at this high level description to verify correct understanding of the problem.

Data Flow Design

- * With the high level description confirmed, logic equations describing the data flow are then created

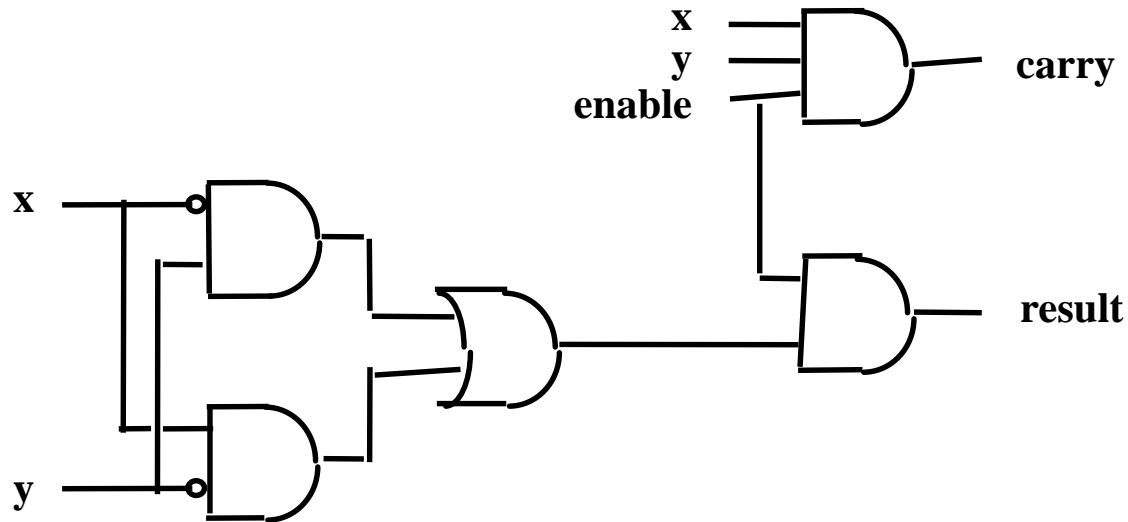
carry = (x AND y) AND enable
result = (x'y OR xy') AND enable



- * Again, the model can be simulated at this level to confirm the logic equations

Logic Design

*** Finally, a structural description is created at the gate level**



*** These gates can be pulled from a library of parts**

What is VHDL?

- **VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language**
- **A Simulation Modeling Language**
- **A Design Entry Language**
- **A Standard Language**
- **A Netlist Language**

History of VHDL

- * **1981: Initiated in 1981 by US DoD to address the 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)**

How is VHDL used?

- * For design specification**
- * For design capture**
- * For design simulation**
- * For design documentation**
- * As an alternative to schematics**
- * As an alternative to proprietary languages**

WHY VHDL?

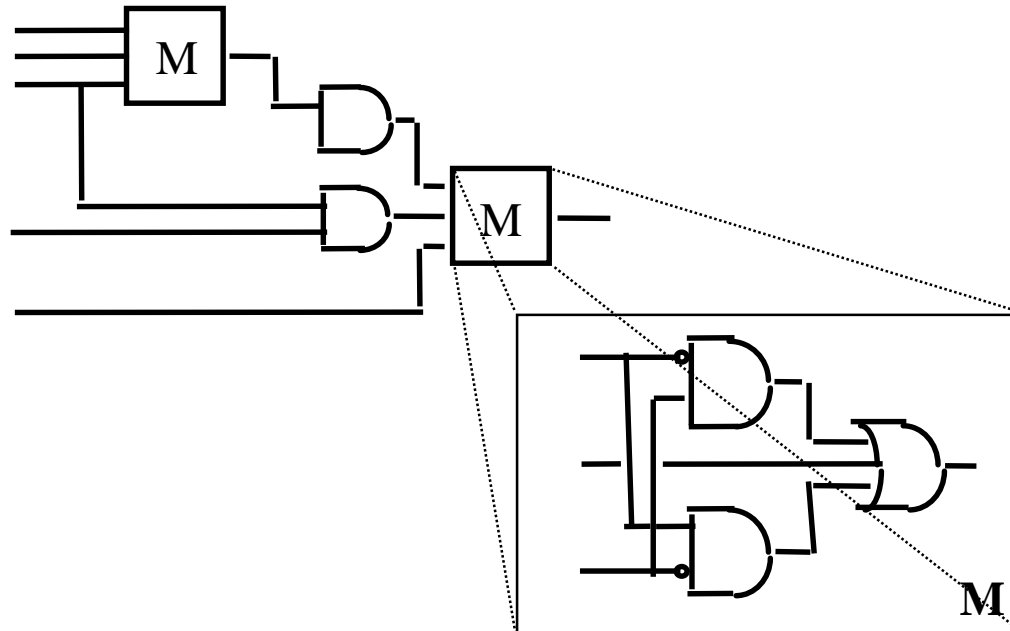
**It will dramatically improve your
productivity**

Features of VHDL

- * **Support for concurrent statements**
 - in actual digital systems all elements of the system are active simultaneously and perform their tasks simultaneously.
- * **Library support**
 - user defined and system predefined primitives reside in a library system
- * **Sequential statements**
 - gives software-like sequential control (e.g. case, if-then-else, loop)

Features of VHDL

* Support for design hierarchy



Features of VHDL

* Generic design

- generic descriptions are configurable for size, physical characteristics, timing, loading, environmental conditions.

(e.g. LS, F, ALS of 7400 family are all functionally equivalent. They differ only in timing.

* Use of subprograms

- the ability to define and use functions and procedures
- subprograms are used for explicit type conversions, operator re-definitions, ... etc

Features of VHDL

* Type declaration and usage

- a hardware description language at various levels of abstraction should not be limited to Bit or Boolean types.
- VHDL allows *integer, floating point, enumerate* types, as well as *user defined* types
- possibility of defining new operators for the new types.

Features of VHDL

* Timing control

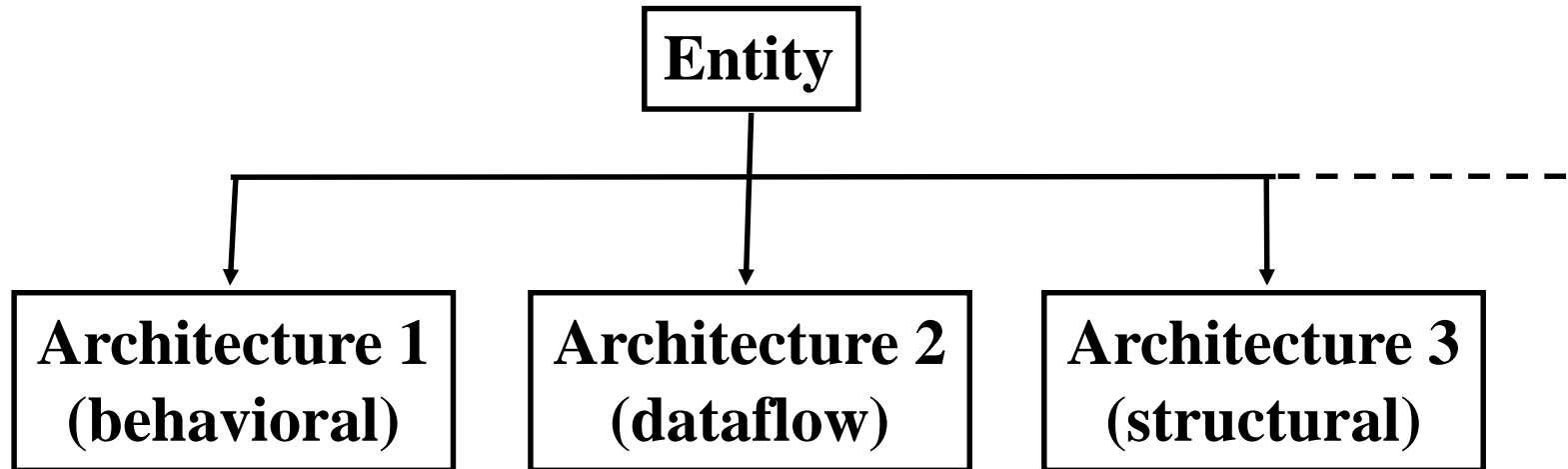
- ability to specify timing at all levels
- clocking scheme is completely up to the user, since the language does not have an implicit clocking scheme
- constructs for edge detection, delay specification, ... etc are available

* Technology independent

What about Verilog?

- * Verilog has the same advantage in availability of simulation models**
- * Verilog has a PLI that permits the ability to write parts of the code using other languages**
- * VHDL has higher-level design management features (configuration declaration, libraries)**
- * VHDL and Verilog are identical in function and different in syntax**
- * No one can decide which language is better.**

VHDL Design Process



Entity Declaration

- * An entity declaration describes the interface of the component
- * **PORT** clause indicates input and output ports
- * An entity can be thought of as a symbol for a component

```
ENTITY half_adder IS  
    PORT (x, y, enable: IN bit;  
          carry, result: OUT bit);  
END half_adder;
```



Port Declaration

- * **PORT** declaration establishes the interface of the object to the outside world
- * **Three parts of the PORT declaration**
 - o **Name**
 - o **Mode**
 - o **Data type**

```
ENTITY test IS  
    PORT (<name> : <mode> <data_type>);  
END test;
```

Name

Any legal VHDL identifier

- * **Only letters, digits, and underscores can be used**
- * **The first character must be a letter**
- * **The last character cannot be an underscore**
- * **Two underscore in succession are not allowed**

Legal names	Illegal names
rs_clk	_rs_clk
ab08B	signal#1
A_1023	A__1023
	rs_clk_

Port Mode

- * **The port mode of the interface describes the direction of the data flow with respect to the component**

- * **The five types of data flow are**
 - **In** : data flows in this port and can only be read
(this is the default mode)
 - **Out** : data flows out this port and can only be written to
 - **Buffer** : similar to **Out**, but it allows for internal feedback
 - **Inout** : data flow can be in either direction with any
number of sources allowed (implies a bus)
 - **Linkage**: data flow direction is unknown

Type of Data

- * **The type of data flowing through the port must be specified to complete the interface**
- * **Data may be of many different types, depending on the package and library used**
- * **Some data types defined in the standards of IEEE are:**
 - o **Bit, Bit_vector**
 - o **Boolean**
 - o **Integer**
 - o **std_ulogic, std_logic**

Architecture Body # 1

- * Architecture declarations describe the operation of the component
- * Many architectures may exist for one entity, but only one may be active at a time

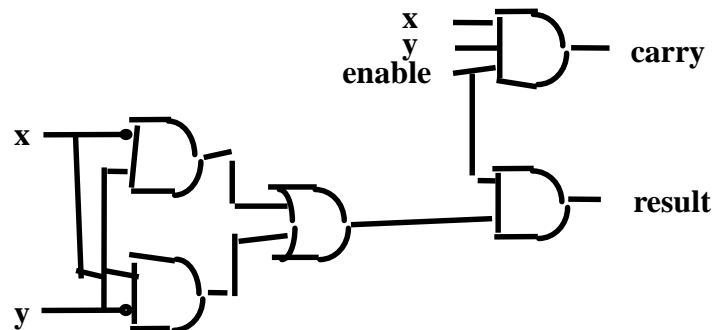
```
ARCHITECTURE behavior1 OF half_adder IS
BEGIN
    PROCESS (enable, x, y)
    BEGIN
        IF (enable = '1') THEN
            result <= x XOR y;
            carry <= x AND y;
        ELSE
            carry <= '0';
            result <= '0';
        END PROCESS;
    END behavior1;
```

Architecture Body # 2

```
ARCHITECTURE data_flow OF half_adder IS  
BEGIN  
    carry = (x AND y) AND enable;  
    result = (x XOR y) AND enable;  
END data_flow;
```

Architecture Body # 3

- * To make the structural architecture, we need first to define the gates to be used.
- * In the shown example, we need NOT, AND, and OR gates



Architecture Body # 3 (cntd.)

```
ENTITY not_1 IS
    PORT (a: IN bit; output: OUT bit);
END not_1;

ARCHITECTURE data_flow OF not_1 IS
BEGIN
    output <= NOT(a);
END data_flow;
```

```
ENTITY and_2 IS
    PORT (a,b: IN bit; output: OUT bit);
END not_1;

ARCHITECTURE data_flow OF and_2 IS
BEGIN
    output <= a AND b;
END data_flow;
```

Architecture Body # 3 (contd.)

```
ENTITY or_2 IS  
    PORT (a,b: IN bit; output: OUT bit);  
END or_2;  
  
ARCHITECTURE data_flow OF or_2 IS  
BEGIN  
    output <= a OR b;  
END data_flow;
```

```
ENTITY and_3 IS  
    PORT (a,b,c: IN bit; output: OUT bit);  
END and_3;  
  
ARCHITECTURE data_flow OF and_3 IS  
BEGIN  
    output <= a AND b AND c;  
END data_flow;
```

Architecture Body # 3 (contd.)

ARCHITECTURE structural OF half_adder IS

```
COMPONENT and2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT and3 PORT(a,b,c: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT or2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT not1 PORT(a: IN bit; output: OUT bit); END COMPONENT;
```

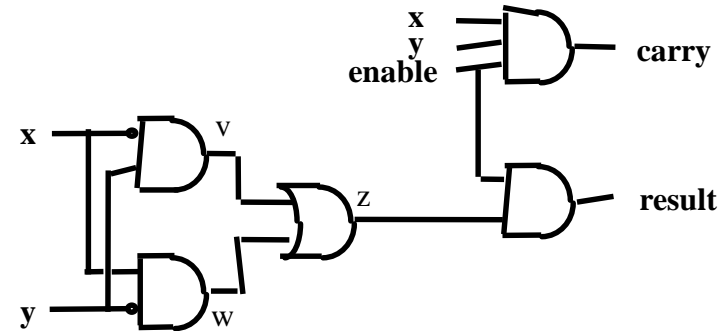
```
FOR ALL: and2 USE ENTITY work.and_2(dataflow);  
FOR ALL: and3 USE ENTITY work.and_3(dataflow);  
FOR ALL: or2 USE ENTITY work.or_2(dataflow);  
FOR ALL: not1 USE ENTITY work.not_2(dataflow);
```

```
SIGNAL v,w,z,nx,nz: BIT;
```

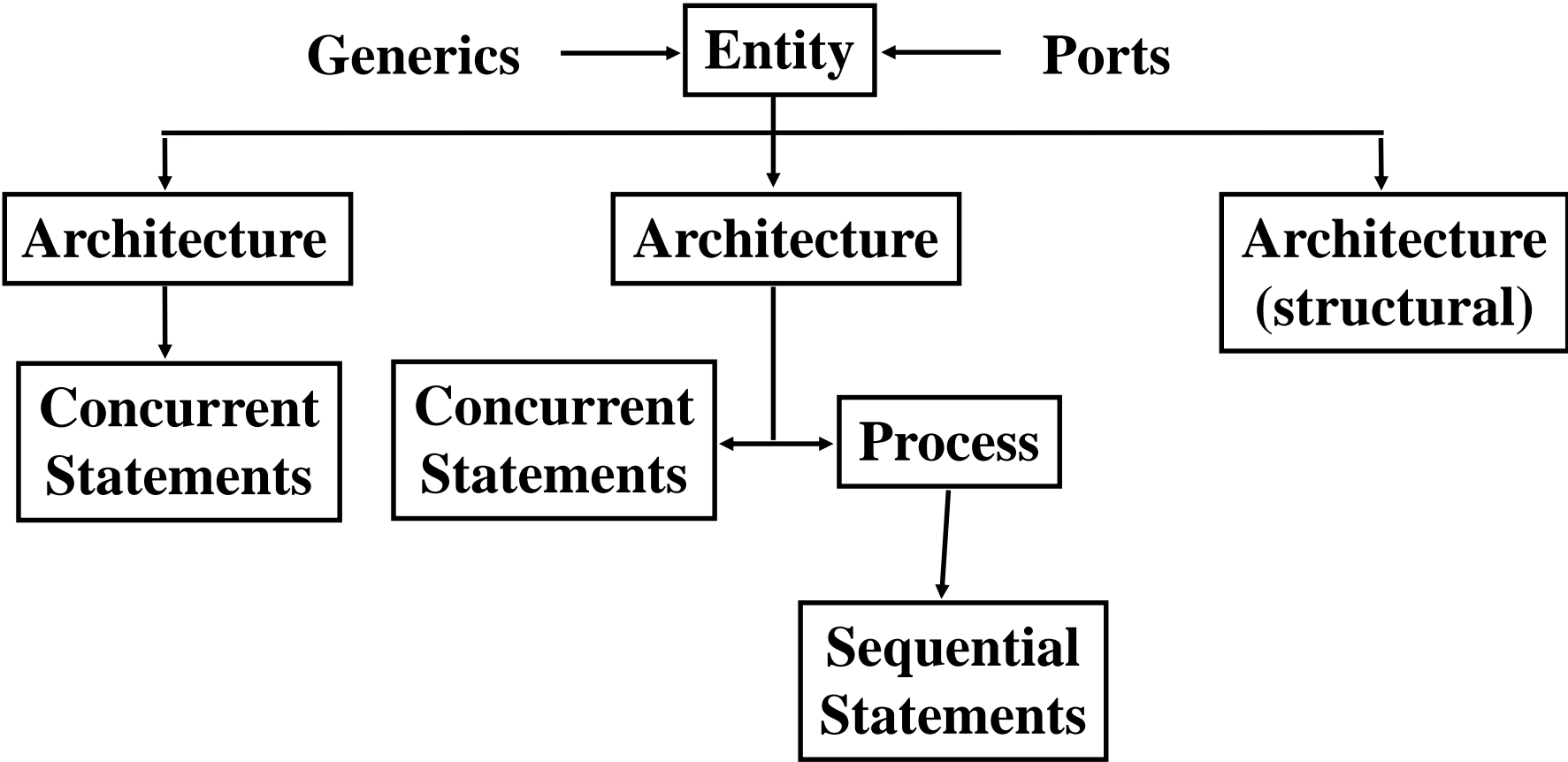
BEGIN

```
c1: not1 PORT MAP (x,nx);  
c2: not1 PORT MAP (y,ny);  
c3: and2 PORT MAP (nx,y,v);  
c4: and2 PORT MAP (x,ny,w);  
c5: or2 PORT MAP (v,w,z);  
c6: and2 PORT MAP (enable,z,result);  
c7: and3 PORT MAP (x,y,enable,carry);
```

END structural;



Summary



VHDL BASICS

- * **Data Objects**
- * **Data Types**
- * **Types and Subtypes**
- * **Attributes**
- * **Sequential and Concurrent Statements**
- * **Procedures and Functions**
- * **Packages and Libraries**
- * **Generics**
- * **Delay Types**

VHDL Objects

- * **There are four types of objects in VHDL**
 - **Constants**
 - **Signals**
 - **Variables**
 - **Files**
- * **File declarations make a file available for use to a design**
- * **Files can be opened for reading and writing**
- * **Files provide a way for a VHDL design to communicate with the host environment**

VHDL Objects

Constants

- * Improve the readability of the code
- * Allow for easy updating

```
CONSTANT <constant_name> : <type_name> := <value>;
```

```
CONSTANT PI : REAL := 3.14;
```

```
CONSTANT WIDTH : INTEGER := 8;
```

VHDL Objects

Signals

- * **Signals are used for communication between components**
- * **Signals can be seen as real, physical wires**

```
SIGNAL <signal_name> : <type_name> [:= <value>];
```

```
SIGNAL enable : BIT;
```

```
SIGNAL output : bit_vector(3 downto 0);
```

```
SIGNAL output : bit_vector(3 downto 0) := "0111";
```

VHDL Objects

Variables

- * Variables are used only in processes and subprograms (functions and procedures)
- * Variables are generally not available to multiple components and processes
- * All variable assignments take place immediately

```
VARIABLE <variable_name> : <type_name> [:= <value>];
```

```
VARIABLE opcode : BIT_VECTOR (3 DOWNT0 0) := "0000";  
VARIABLE freq : INTEGER;
```

Signals versus Variables

* A key difference between variables and signals is the assignment delay

```
ARCHITECTURE signals OF test IS
  SIGNAL a, b, c, out_1, out_2 : BIT;
BEGIN
  PROCESS (a, b, c)
  BEGIN
    out_1 <= a NAND b;
    out_2 <= out_1 XOR c;
  END PROCESS;
END signals;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0

Signals versus Variables (cont. 1)

```
ARCHITECTURE variables OF test IS  
  SIGNAL a, b, c: BIT;  
  VARIABLE out_3, out_4 : BIT;  
BEGIN  
  PROCESS (a, b, c)  
  BEGIN  
    out_3 := a NAND b;  
    out_4 := out_3 XOR c;  
  END PROCESS;  
END variables;
```

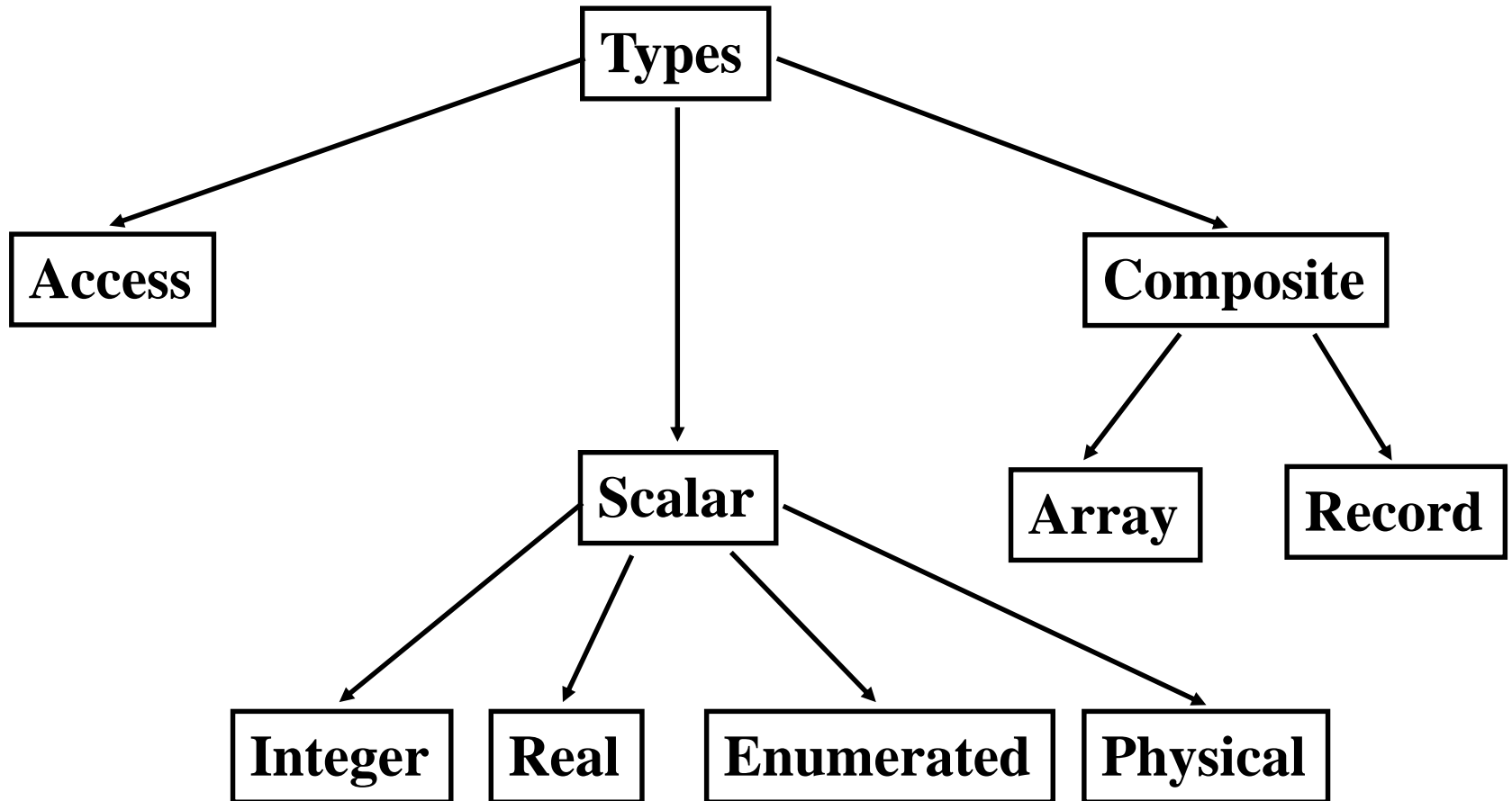
Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	1

VHDL Objects

Scoping Rules

- * VHDL limits the visibility of the objects, depending on where they are declared**
- * The scope of the object is as follows**
 - o Objects declared in a package are global to all entities that use that package**
 - o Objects declared in an entity are global to all architectures that use that entity**
 - o Objects declared in an architecture are available to all statements in that architecture**
 - o Objects declared in a process are available to only that process**
- * Scoping rules apply to constants, variables, signals and files**

Data Types



Scalar Types

* Integer Types

- Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647

```
ARCHITECTURE test_int OF test IS
BEGIN
    PROCESS (X)
        VARIABLE a: INTEGER;
    BEGIN
        a := 1; -- OK
        a := -1; -- OK
        a := 1.0; -- bad
    END PROCESS;
END TEST;
```

Scalar Types (cntd.)

* Real Types

- Minimum range for any implementation as defined by standard: **-1.0E38 to 1.0E38**

```
ARCHITECTURE test_real OF test IS
BEGIN
    PROCESS (X)
        VARIABLE a: REAL;
    BEGIN
        a := 1.3; -- OK
        a := -7.5; -- OK
        a := 1; -- bad
        a := 1.7E13; -- OK
        a := 5.3 ns; -- bad
    END PROCESS;
END TEST;
```

Scalar Types (cntd.)

* Enumerated Types

- User defined range

```
TYPE binary IS ( ON, OFF );
...some statements ...
ARCHITECTURE test_enum OF test IS
BEGIN
    PROCESS (X)
        VARIABLE a: binary;
    BEGIN
        a := ON; -- OK
        ... more statements ...
        a := OFF; -- OK
        ... more statements ...
    END PROCESS;
END TEST;
```

Scalar Types (cntd.)

* Physical Types:

- Can be user defined range

```
TYPE resistance IS RANGE 0 to 1000000
UNITS
    ohm; -- ohm
    Kohm = 1000 ohm; -- 1 K
    Mohm = 1000 kohm; -- 1 M
END UNITS;
```

- Time units are the only predefined physical type in VHDL

Scalar Types (cntd.)

* The predefined time units are as follows

```
TYPE TIME IS RANGE -2147483647 to 2147483647  
UNITS  
    fs; -- femtosecond  
    ps = 1000 fs; -- picosecond  
    ns = 1000 ps; -- nanosecond  
    us = 1000 ns; -- microsecond  
    ms = 1000 us; -- millisecond  
    sec = 1000 ms; -- second  
    min = 60 sec; -- minute  
    hr = 60 min; -- hour  
END UNITS;
```

Composite Types

* Array Types:

- Used to collect one or more elements of a similar type in a single construct
- Elements can be any VHDL data type

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```

```
0 ...element numbers...31
```

```
0 ...array values...1
```

```
SIGNAL X: data_bus;
```

```
SIGNAL Y: BIT;
```

```
Y <= X(12); -- Y gets value of 12th element
```

Composite Types (cntd.)

- * **Another sample one-dimensional array (using the DOWNTO order)**

```
TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;
```

```
15 ...element numbers... 0
```

```
0 ...array values... 1
```

```
Signal X: register;
```

```
SIGNAL Y: BIT;
```

```
Y <= X(4); -- Y gets value of 4th element
```

- * **DOWNTO keyword orders elements from left to right, with decreasing element indices**

Composite Types (cntd.)

* Two-dimensional arrays are useful for describing truth tables.

```
TYPE truth_table IS ARRAY(0 TO 7, 0 TO 4) OF BIT;  
CONSTANT full_adder: truth_table := (  
    "000_00",  
    "001_01",  
    "010_01",  
    "011_10",  
    "100_01",  
    "101_10",  
    "110_10",  
    "111_11");
```

Composite Types (cntd.)

* Record Types

- Used to collect one or more elements of a different types in single construct
- Elements can be any VHDL data type
- Elements are accessed through field name

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
RECORD
    status : binary;
    IDnumber : integer;
END RECORD;
VARIABLE switch : switch_info;
    switch.status := on; -- status of the switch
    switch.IDnumber := 30; -- number of the switch
```

Access Types

* Access

- Similar to pointers in other languages
- Allows for dynamic allocation of storage
- Useful for implementing queues, fifos, etc.

Subtypes

* Subtype

- Allows for user defined constraints on a data type
- May include entire range of base type
- Assignments that are out of the subtype range result in an error

```
SUBTYPE <name> IS <base type> RANGE <user range>;
```

```
SUBTYPE first_ten IS integer RANGE 1 to 10;
```

Subtypes

(Example)

```
TYPE byte IS bit_vector(7 downto 0);  
signal x_byte: byte;  
signal y_byte: bit_vector(7 downto 0);  
  
IF x_byte = y_byte THEN ...
```

← Compiler produces an error

Compiler produces no errors

```
SUBTYPE byte IS bit_vector(7 downto 0)  
signal x_byte: byte;  
signal y_byte: bit_vector(7 downto 0);  
  
IF x_byte = y_byte THEN ...
```

Summary

- * VHDL has several different data types available to the designer**
- * Enumerated types are user defined**
- * Physical types represent physical quantities**
- * Arrays contain a number of elements of the same type or subtypes**
- * Records may contain a number of elements of different types or subtypes**
- * Access types are basically pointers**
- * Subtypes are user defined restrictions on the base type**

Attributes

- * **Language defined attributes return information about certain items in VHDL**
 - **Types, subtypes**
 - **Procedures, functions**
 - **Signals, variables, constants**
 - **Entities, architectures, configurations, packages**
 - **Components**
- * **VHDL has several predefined attributes that are useful to the designer**
- * **Attributes can be user-defined to handle custom situations (user-defined records, etc.)**

Attributes

(Signal Attributes)

- * General form of attribute use is:

```
<name> ' <attribute_identifier>
```

- * Some examples of signal attributes

```
X'EVENT -- evaluates TRUE when an event on signal X has just  
-- occured.
```

```
X'LAST_VALUE -- returns the last value of signal X
```

```
X'STABLE(t) -- evaluates TRUE when no event has occured on  
-- signal X in the past t' time
```


Attributes

(Value Attributes)

'LEFT -- returns the leftmost value of a type

'RIGHT -- returns the rightmost value of a type

'HIGH -- returns the greatest value of a type

'LOW -- returns the lowest value of a type

'LENGTH -- returns the number of elements in a constrained array

'RANGE -- returns the range of an array

Attributes

(Example)

TYPE count is RANGE 0 TO 127;

TYPE states IS (idle, decision, read, write);

TYPE word IS ARRAY(15 DOWNT0 0) OF bit;

count'left = 0	states'left = idle	word'left = 15
count'right = 127	states'right = write	word'right = 0
count'high = 127	states'high = write	word'high = 15
count'low = 0	states'low = idle	word'low = 0
count'length = 128	states'length = 4	word'length = 16

count'range = 0 TO 127

word'range = 15 DOWNT0 0

Register Example

- * This example shows how attributes can be used in the description of an 8-bit register.
- * Specifications
 - Triggers on rising clock edge
 - Latches only on enable high
 - Has a data setup time of 5 ns.

```
ENTITY 8_bit_reg IS
    PORT (enable, clk : IN std_logic;
          a : IN std_logic_vector (7 DOWNTO 0);
          b : OUT std_logic_vector (7 DOWNTO 0);
    END 8_bit_reg;
```

Register Example (contd.)

- * A signal having the type `std_logic` may assume the values: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'
- * The use of 'STABLE detects for setup violations

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
    BEGIN
        PROCESS (clk)
            BEGIN
                IF (enable = '1') AND a'STABLE(5 ns) AND
                    (clk = '1') THEN
                    b <= a;
                END IF;
            END PROCESS;
        END first_attempt;
```

- * What happens if `clk` was 'X'?

Register Example (contd.)

- * The use of 'LAST_VALUE ensures the clock is rising from a 0 value

```
ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (enable = '1') AND a'STABLE(5 ns) AND
            (clk = '1') AND (clk'LASTVALUE = '0') THEN
            b <= a;
        END IF;
    END PROCESS;
END behavior;
```

Concurrent and Sequential Statements

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

Concurrent Statements

**Three types of concurrent statements
used in dataflow descriptions**

```
graph TD; A["Three types of concurrent statements used in dataflow descriptions"] --> B["Boolean Equations"]; A --> C["with-select-when"]; A --> D["when-else"];
```

Boolean Equations

For concurrent
signal assignments

with-select-when

For selective
signal assignments

when-else

For conditional
signal assignments

Concurrent Statements

Boolean equations

```
entity control is port(mem_op, io_op, read, write: in bit;  
                        memr, memw, io_rd, io_wr:out bit);  
end control;
```

```
architecture control_arch of control is  
begin  
    memw <= mem_op and write;  
    memr  <= mem_op and read;  
    io_wr <= io_op and write;  
    io_rd <= io_op and read;  
end control_arch;
```


Concurrent Statements

with-select-when

```
entity mux is port(a,b,c,d: in std_logic_vector(3 downto 0);  
                    s: in std_logic_vector(1 downto 0);  
                    x: out std_logic_vector(3 downto 0));  
end mux;
```

```
architecture mux_arch of mux is  
begin  
  with s select  
    x <= a when "00",  
        b when "01",  
        c when "10",  
        d when others;  
end mux_arch;
```

Concurrent Statements

with-select-when (cntd.)

```
architecture mux_arch of mux is
```

```
begin
```

```
with s select
```

```
    x <= a when "00",
```

```
    b when "01",
```

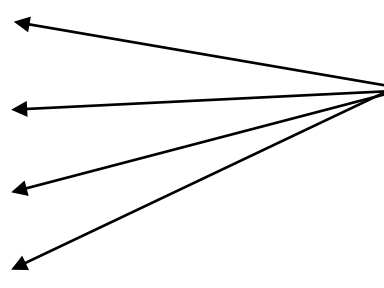
```
    c when "10",
```

```
    d when "11",
```

```
    "--" when others;
```

```
end mux_arch;
```

*possible values
of s*



Concurrent Statements


when-else

```
architecture mux_arch of mux is  
begin
```

```
    x <= a when (s = "00") else  
        b when (s = "01") else  
        c when (s = "10") else  
        d;
```

```
end mux_arch;
```

This may be
any simple
condition



Logical Operators

AND

OR

NAND

XOR

XNOR

NOT

* Predefined for the types:

- bit and Boolean.
- One dimensional arrays of bit and Boolean.

* Logical operators *don't have* an order of precedence

X <= A or B and C

will result in a compile-time error.

Relational Operators

=

<=

<

/=

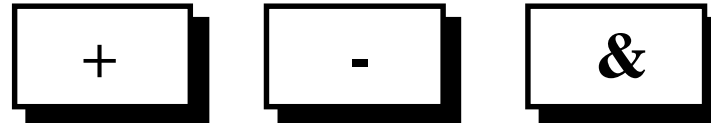
>=

>

- * Used for testing equality, inequality, and ordering.
- * (= and /=) are defined for all types.
- * (<, <=, >, and >=) are defined for scalar types
- * The types of operands in a relational operation must match.

Arithmetic Operators

Addition operators



Multiplication operators



Miscellaneous operators



Sequential Statements

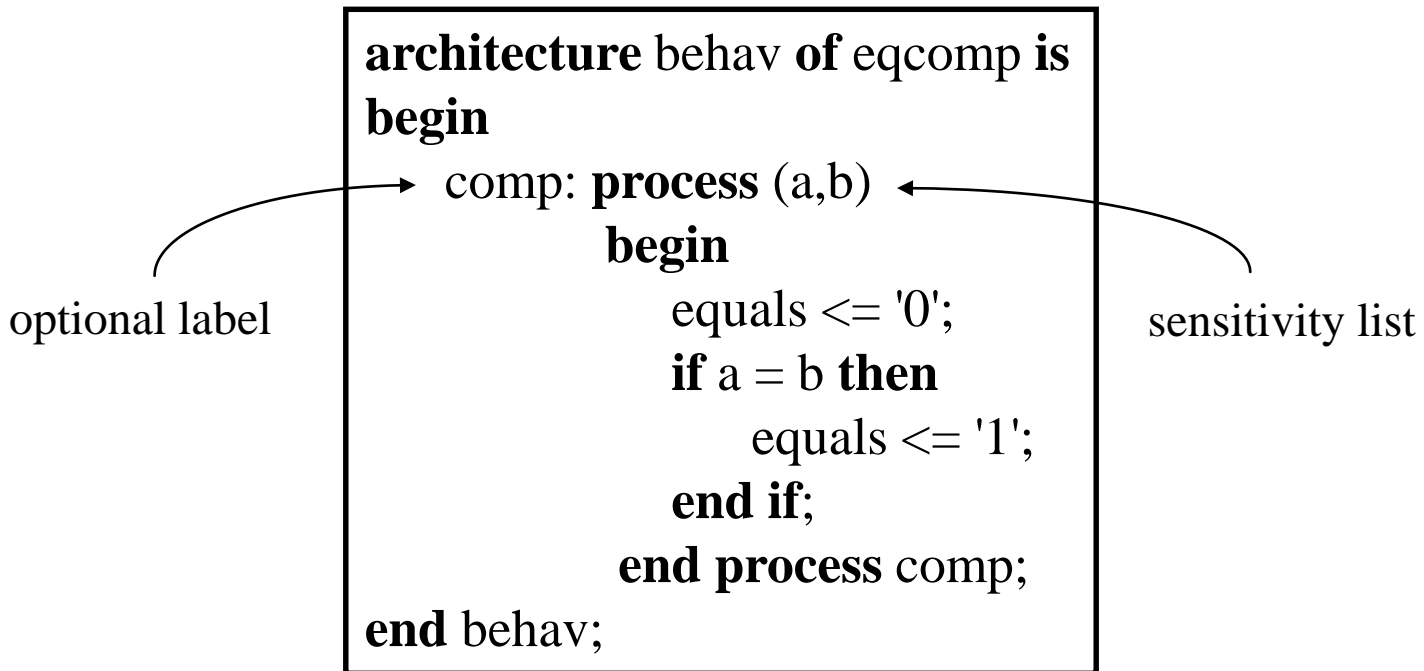
Sequential statements are contained in a process, function, or procedure.

Inside a process signal assignment is sequential from a simulation point of view.

The order in which signal assignments are listed does affect the result.

Process Statement

- * **Process statement is a VHDL construct that embodies algorithms**
- * **A process has a sensitivity list that identifies which signals will cause the process to execute.**

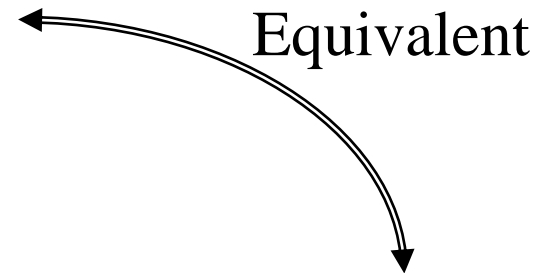


Process Statement

The use of wait statements

```
Proc1: process (a,b,c)
  begin
    x <= a and b and c;
  end process;
```

Equivalent



```
Proc2: process
  begin
    x <= a and b and c;
    wait on a, b, c;
  end process;
```

Sequential Statements

**Four types of sequential statements
used in behavioral descriptions**

```
graph TD; A[Four types of sequential statements used in behavioral descriptions] --> B[if-the-else]; A --> C[case-when]; A --> D[for-loop]; A --> E[while-loop];
```

if-the-else

case-when

for-loop

while-loop

Sequential Statements

if-then-else

```
signal step: bit;  
signal addr: bit_vector(0 to 7);  
      ⋮  
p1: process (addr)  
  begin  
    if addr > x"0F" then  
      step <= '1';  
    else  
      step <= '0';  
    end if;  
  end process;
```

```
signal step: bit;  
signal addr: bit_vector(0 to 7);  
      ⋮  
p2: process (addr)  
  begin  
    if addr > x"0F" then  
      step <= '1';  
    end if;  
  end process;
```

P2 has an implicit memory

Sequential Statements

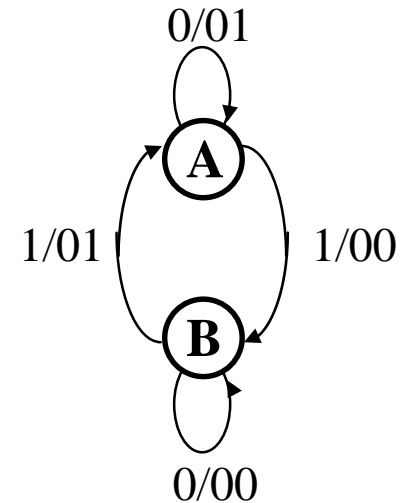
if-then-else (cntd.)

```
architecture mux_arch of mux is
begin
mux4_1: process (a,b,c,d,s)
begin
    if s = "00" then
        x <= a;
    elsif s = "01" then
        x <= b;
    elsif s = "10" then
        x <= c;
    else
        x <= d;
    end if;
end process;
end mux_arch;
```

Sequential Statements

case-when

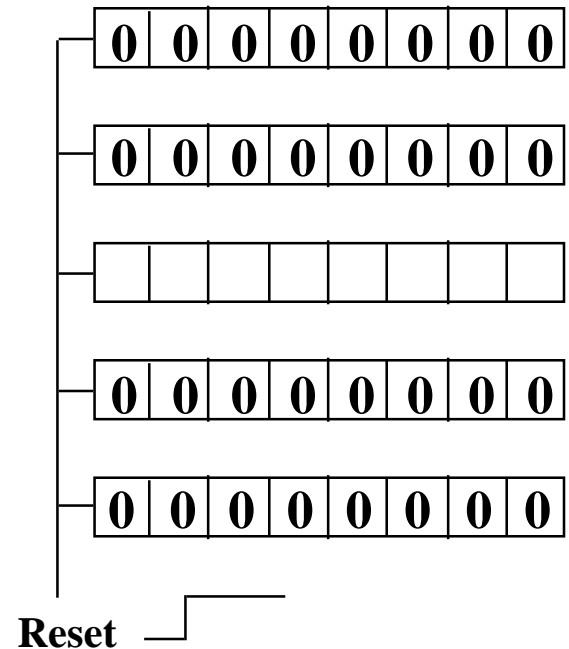
```
case present_state is
  when A => y <= '0'; z <= '1';
    if x = '1' then
      next_state <= B;
    else
      next_state <= A;
    end if;
  when B => y <= '0'; z <= '0';
    if x = '1' then
      next_state <= A;
    else
      next_state <= B;
    end if;
end case;
```



inputs: x
outputs: y,z

Sequential Statements for-loop

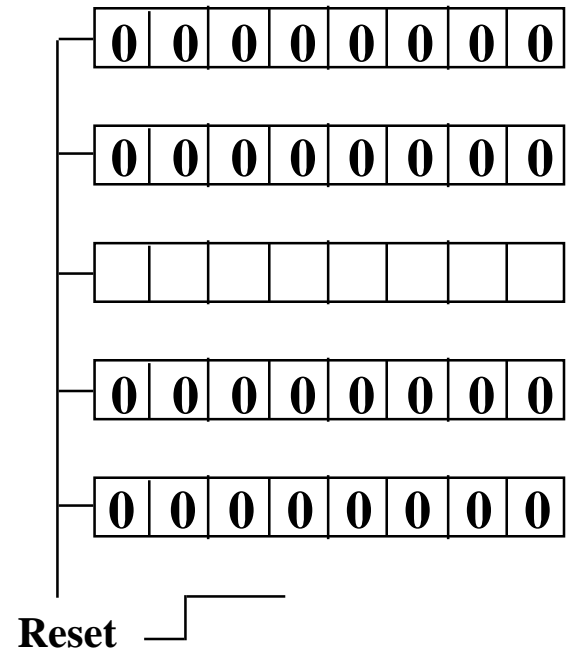
```
type register is bit_vector(7 downto 0);  
type reg_array is array(4 downto 0) of register;  
signal fifo: reg_array;  
  
process (reset)  
begin  
    if reset = '1' then  
        for i in 4 downto 0 loop  
            if i = 2 then  
                next;  
            else  
                fifo(i) <= (others => '0');  
            end if;  
        end loop;  
    end if;  
end process;
```



Sequential Statements

while-loop

```
type register is bit_vector(7 downto 0);  
type reg_array is array(4 downto 0) of register;  
signal fifo: reg_array;  
  
process (reset)  
    variable i: integer := 0;  
begin  
    if reset = '1' then  
        while i <= 4 loop  
            if i /= 2 then  
                fifo(i) <= (others => '0');  
            end if;  
            i := i + 1;  
        end loop;  
    end if;  
end process;
```



Functions and Procedures

- * **High level design constructs that are most commonly used for:**
 - **Type conversions**
 - **Operator overloading**
 - **Alternative to component instantiation**
 - **Any other user defined purpose**

- * **The subprograms of most use are predefined in:**
 - **IEEE 1076, 1164, 1076.3 standards**

Functions

Type conversion

```
function bv2I (bv: bit_vector) return integer is  
  variable result, onebit: integer := 0;  
begin  
  myloop: for i in bv'low to bv'high loop  
    onebit := 0;  
    if bv(i) = '1' then  
      onbit := 2**(I-bv'low);  
    end if;  
    result := result + onebit;  
  end loop myloop;  
  return (result);  
end bv2I;
```

* **Statements within a function must be sequential.**

* **Function parameters can only be inputs and they cannot be modified.**

* **No new signals can be declared in a function (variables may be declared).**

Functions

Shorthand for simple components

```
function inc (a: bit_vector) return bit_vector is  
    variable s: bit_vector (a'range);  
    variable carry: bit;  
begin  
    carry := '1';  
    for i in a'low to a'high loop  
        s(i) := a(i) xor carry;  
        carry := a(i) and carry;  
    end loop  
    return (s);  
end inc;
```

*** Functions are restricted to substitute components with only one output.**

Functions

Overloading functions

```
function "+" (a,b: bit_vector) return  
bit_vector is  
    variable s: bit_vector (a'range);  
    variable c: bit;  
    variable bi: integer;  
begin  
    carry := '0';  
    for i in a'low to a'high loop  
        bi := b'low + (i - a'low);  
        s(i) := (a(i) xor b(bi)) xor c;  
        c := ((a(i) or b(bi)) and c) or  
            (a(i) and b(bi));  
    end loop;  
    return (s);  
end "+";
```

```
function "+" (a: bit_vector; b: integer)  
return bit_vector is  
begin  
    return (a + i2bv(b,a'length));  
end "+";
```

Using Functions

Functions may be defined in:

- * **declarative region of an architecture**
(visible only to that architecture)
- * **package**
(is made visible with a use clause)

```
use work.my_package.all
architecture myarch of full_add is
begin
    sum <= a xor b xor c_in;
    c_out <= majority(a,b,c_in)
end;
```

```
use work.my_package.all
architecture myarch of full_add is
    . } ← Here we put the function
    . } ← definition
begin
    sum <= a xor b xor c_in;
    c_out <= majority(a,b,c_in)
end;
```

Procedures

```
entity flop is port(clk: in bit;  
    data_in: in bit_vector(7 downto 0);  
    data_out, data_out_bar: out bit_vector(7 downto 0));  
end flop;
```

architecture design of flop **is**

```
procedure dff(signal d: bit_vector; signal clk: bit;  
    signal q, q_bar: out bit_vector) is  
begin  
    if clk'event and clk = '1' then  
        q <= d; q_bar <= not(d);  
    end if;  
end procedure;
```

```
begin  
    dff(data_in, clk, data_out,data_out_bar);  
end design;
```

Libraries and Packages

*** Used to declare and store:**

- Components**
- Type declarations**
- Functions**
- Procedures**

*** Packages and libraries provide the ability to reuse constructs in multiple entities and architectures**

Libraries

- * **Library is a place to which design units may be compiled.**
- * **Two predefined libraries are the IEEE and WORK libraries.**
- * **IEEE standard library contains the IEEE standard design units. (e.g. the packages: std_logic_1164, numeric_std).**
- * **WORK is the default library.**
- * **VHDL knows library only by logical name.**

Libraries

How to use ?

- * A library is made visible using the library clause.

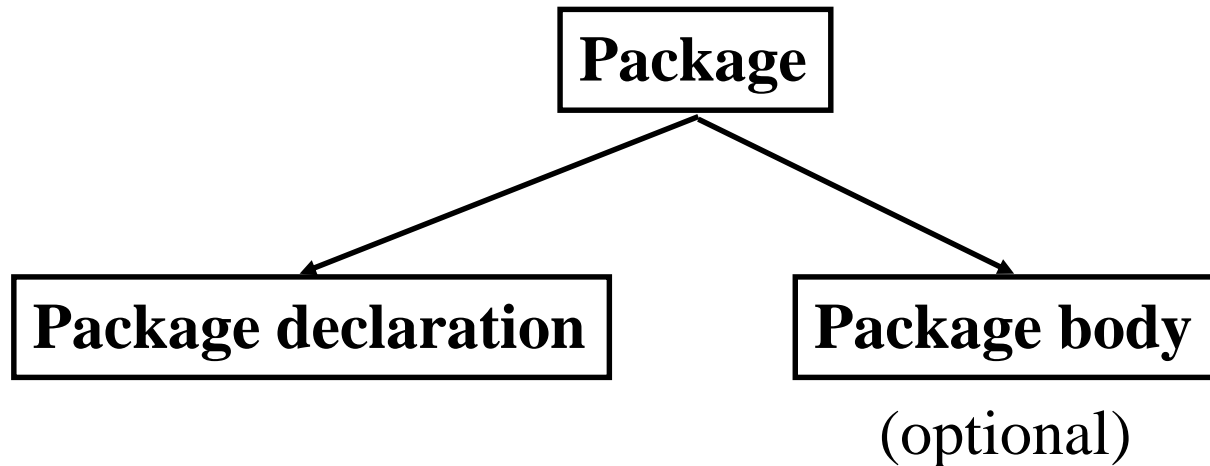
```
library ieee;
```

- * Design units within the library must also be made visible via the use clause.

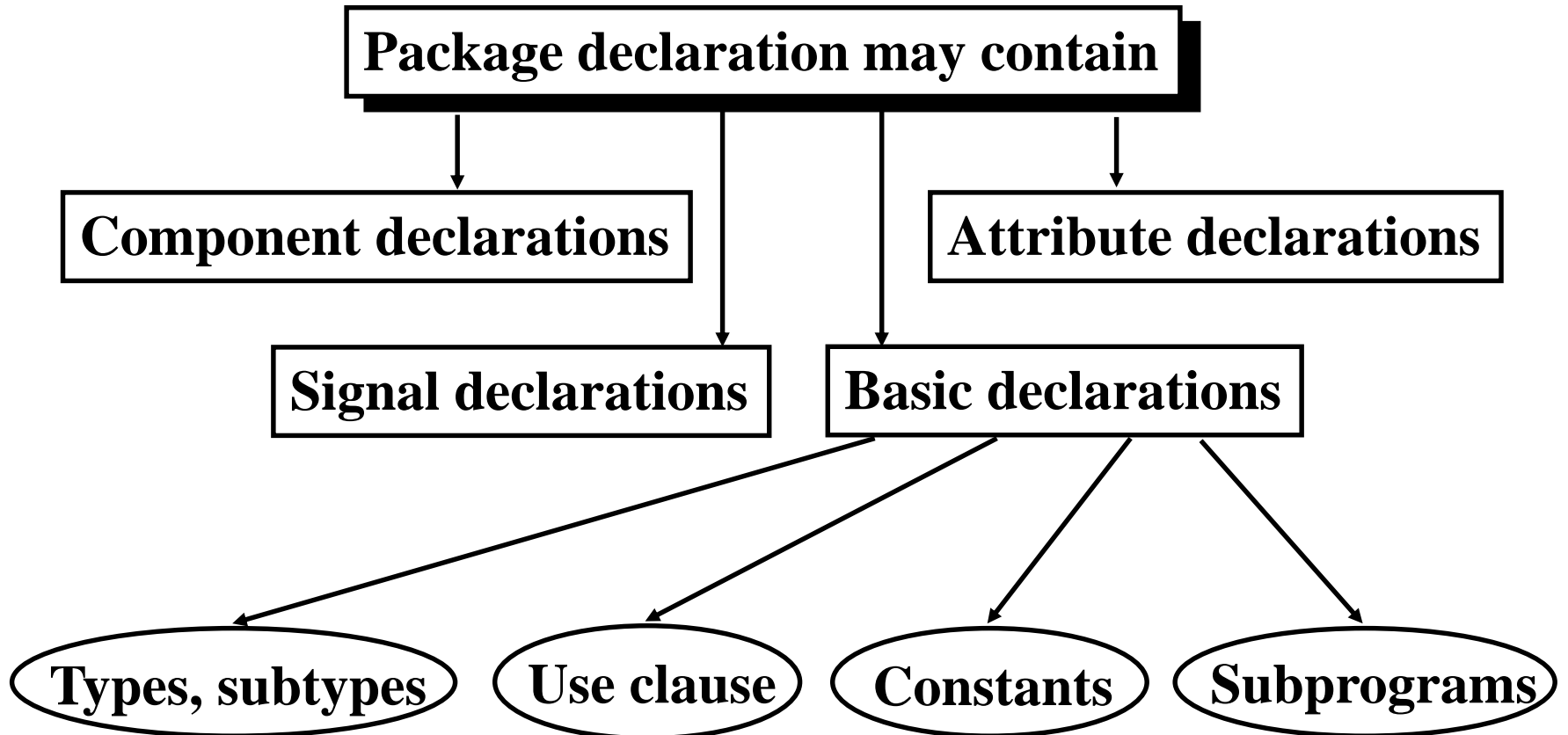
```
for all: and2 use entity work.and_2(dataflow);  
for all: and3 use entity work.and_3(dataflow);  
for all : or2 use entity work.or_2(dataflow);  
for all : not1 use entity work.not_2(dataflow);
```


Packages

* Packages are used to make their constructs visible to other design units.



Packages



Package Declaration

Example of a package declaration

```
package my_package is  
  type binary is (on, off);  
  constant pi : real := 3.14;  
  procedure add_bits3 (signal a, b, en : in bit;  
    signal temp_result, temp_carry : out bit);  
end my_package;
```

The procedure body is defined in the "package body"

Package Body

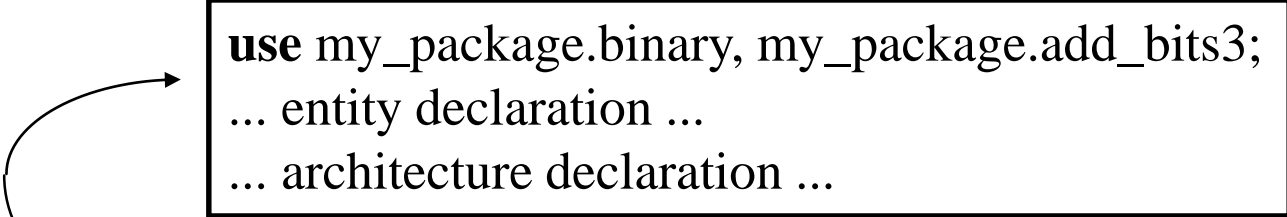
- * The package declaration contains only the declarations of the various items
- * The package body contains subprogram bodies and other declarations not intended for use by other VHDL entities

```
package body my_package is
  procedure add_bits3 (signal a, b, en : in bit;
    signal temp_result, temp_carry : out bit) is
  begin
    temp_result <= (a xor b) and en;
    temp_carry <= a and b and en;
  end add_bits3;
end my_package;
```

Package

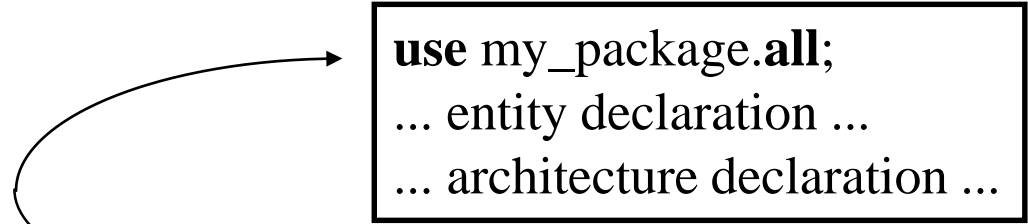
How to use ?

* A package is made visible using the use clause.



```
use my_package.binary, my_package.add_bits3;  
... entity declaration ...  
... architecture declaration ...
```

use the *binary* and *add_bits3* declarations



```
use my_package.all;  
... entity declaration ...  
... architecture declaration ...
```

use *all* of the declarations in package my_package

Generics

- * **Generics may be added for readability, maintenance and configuration.**

```
entity half_adder is  
  generic (prop_delay : time := 10 ns);  
  port (x, y, enable: in bit;  
        carry, result: out bit);  
end half_adder;
```

Default value
when half_adder
is used, if no other
value is specified

In this case, a generic called prop_delay was added to the entity and defined to be 10 ns

Generics (cntd.)

```
architecture data_flow of half_adder is  
begin  
    carry = (x and y) and enable after prop_delay;  
    result = (x xor y) and enable after prop_delay;  
end data_flow;
```

Generics (cntd.)

architecture structural of two_bit_adder is

```
component adder generic( prop_delay: time);  
  port(x,y,enable: in bit; carry, result: out bit);  
end component;
```

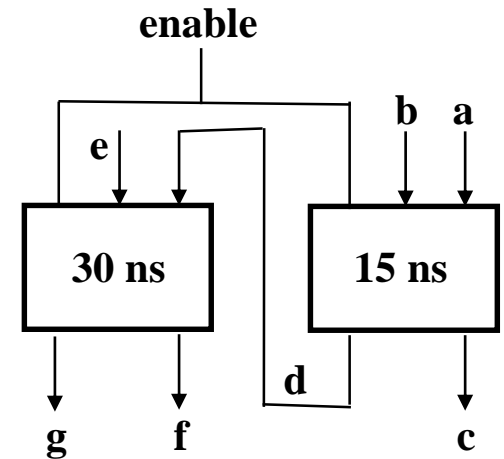
```
for c1: adder use entity work.half_adder(data_flow);  
for c2: adder use entity work.half_adder(data_flow);
```

```
signal d: bit;
```

```
begin
```

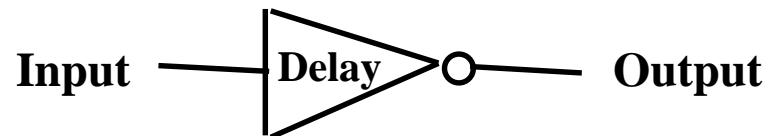
```
  c1: adder generic map(15 ns) port map (a,b,enable,d,c);  
  c2: adder generic map(30 ns) port map (e,d,enable,g,f);
```

```
end structural;
```



Delay Types

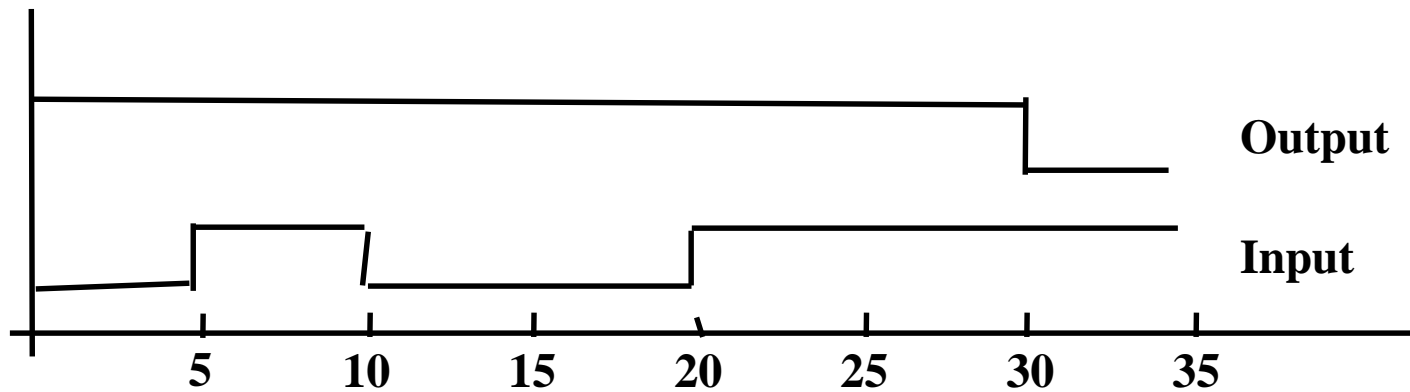
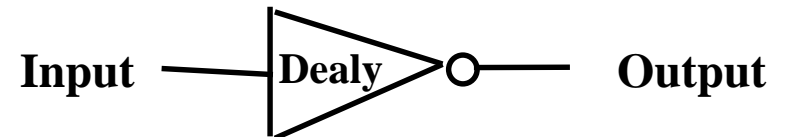
- * **Delay is created by scheduling a signal assignment for a future time**
- * **There are two main types of delay supported VHDL**
 - **Inertial**
 - **Transport**



Inertial Delay

- * Inertial delay is the default delay type
- * It absorbs pulses of shorter duration than the specified delay

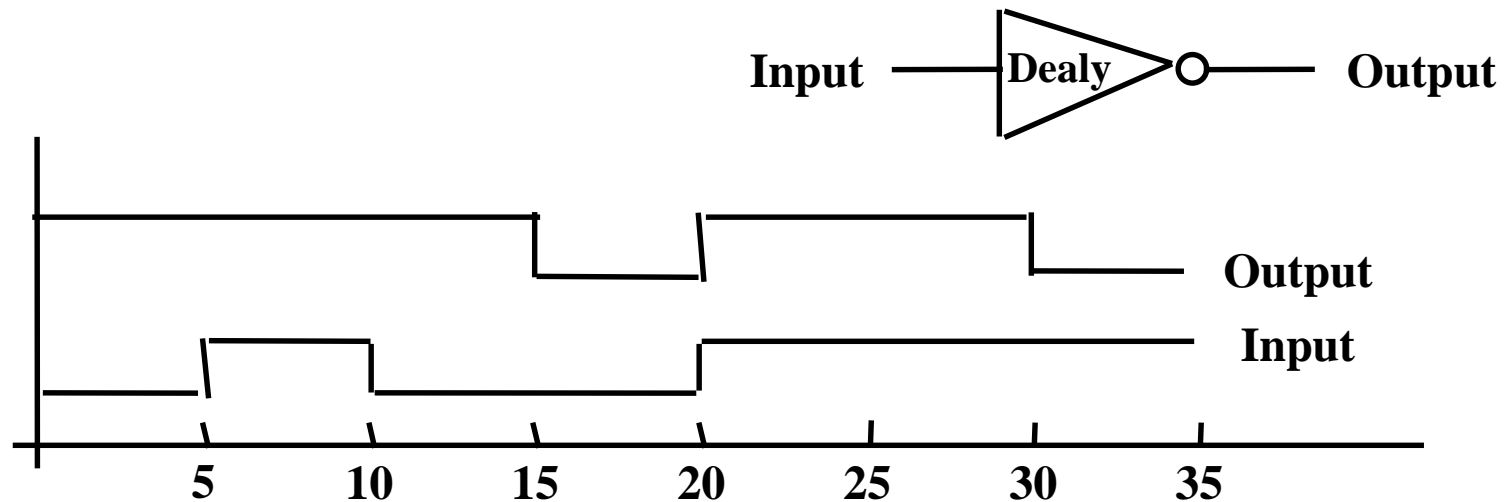
-- Inertial is the default
Output <= not Input after 10 ns;



Transport Delay

- * **Must be explicitly specified by user**
- * **Passes all input transitions with delay**

```
-- TRANSPORT must be specified  
Output <= transport not Input after 10 ns;
```



Summary

- * VHDL is a worldwide standard for the description and modeling of digital hardware**
- * VHDL gives the designer many different ways to describe hardware**
- * Familiar programming tools are available for complex and simple problems**
- * Sequential and concurrent modes of execution meet a large variety of design needs**
- * Packages and libraries support design management and component reuse**

References

D. R. Coehlo, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

***IEEE Standard VHDL Language Reference Manual*,
IEEE Std 1076-1993.**

References

J. Bhasker, *A VHDL Primer*, Prentice Hall, 1995.

Perry, D.L., *VHDL*, McGraw-Hill, 1994.

K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley, 1996