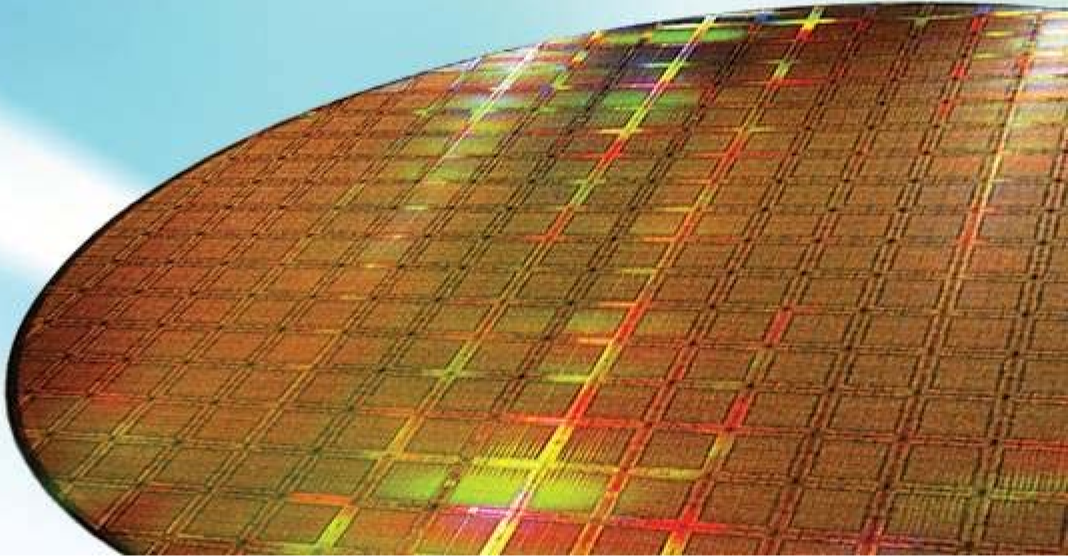
Introduction to VHDL



Course Objectives

- Implement basic constructs of VHDL
- Implement modeling structures of VHDL
- Create a new project in Quartus® II
- Compile a design into an FPGA
- Analyze the design environment
- Obtain an overview of Altera FPGA technology

Course Outline

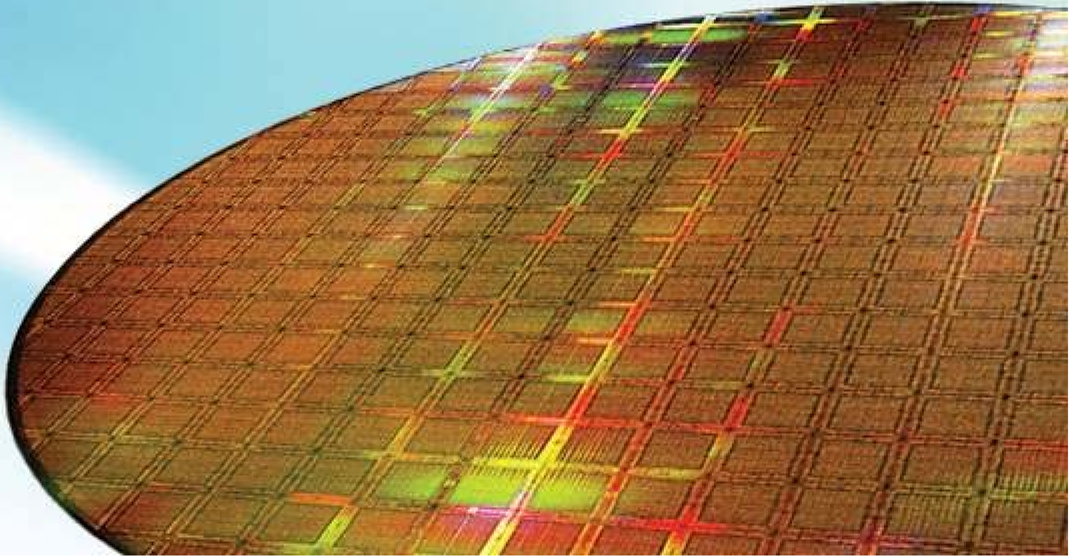
- Introduction to Altera devices and design software
- VHDL basics
  - Overview of language
- Design units
  - Entity
  - Architecture
  - Configurations
  - Packages (libraries)
- Architecture modeling fundamentals
  - Signals
  - Processes

Course Outline

- Understanding VHDL and logic synthesis
  - Process statement
  - Inferring logic
- Model application
  - State machine coding
- Hierarchical designing
  - Overview
  - Structural modeling
  - Application of library of parameterized modules (LPM)

Introduction to VHDL

Introduction to Altera Devices & Design Software



The Programmable Solutions Company ®

- **Devices**
  - Stratix® III
  - Stratix II
  - Cyclone® II
  - Stratix II GX
  - Stratix GX
  - Stratix
  - Cyclone

- **Devices (continued)**
  - MAX® II
  - Mercury™ Devices
  - ACEX® Devices
  - FLEX® Devices
  - MAX Devices
  - HardCopy® II & HardCopy

- **Intellectual Property (IP)**
  - Signal Processing
  - Communications
  - Embedded Processors
    - Nios ® II

- **Tools**
  - Quartus II Software
  - SOPC Builder
  - DSP Builder
  - Nios II IDE

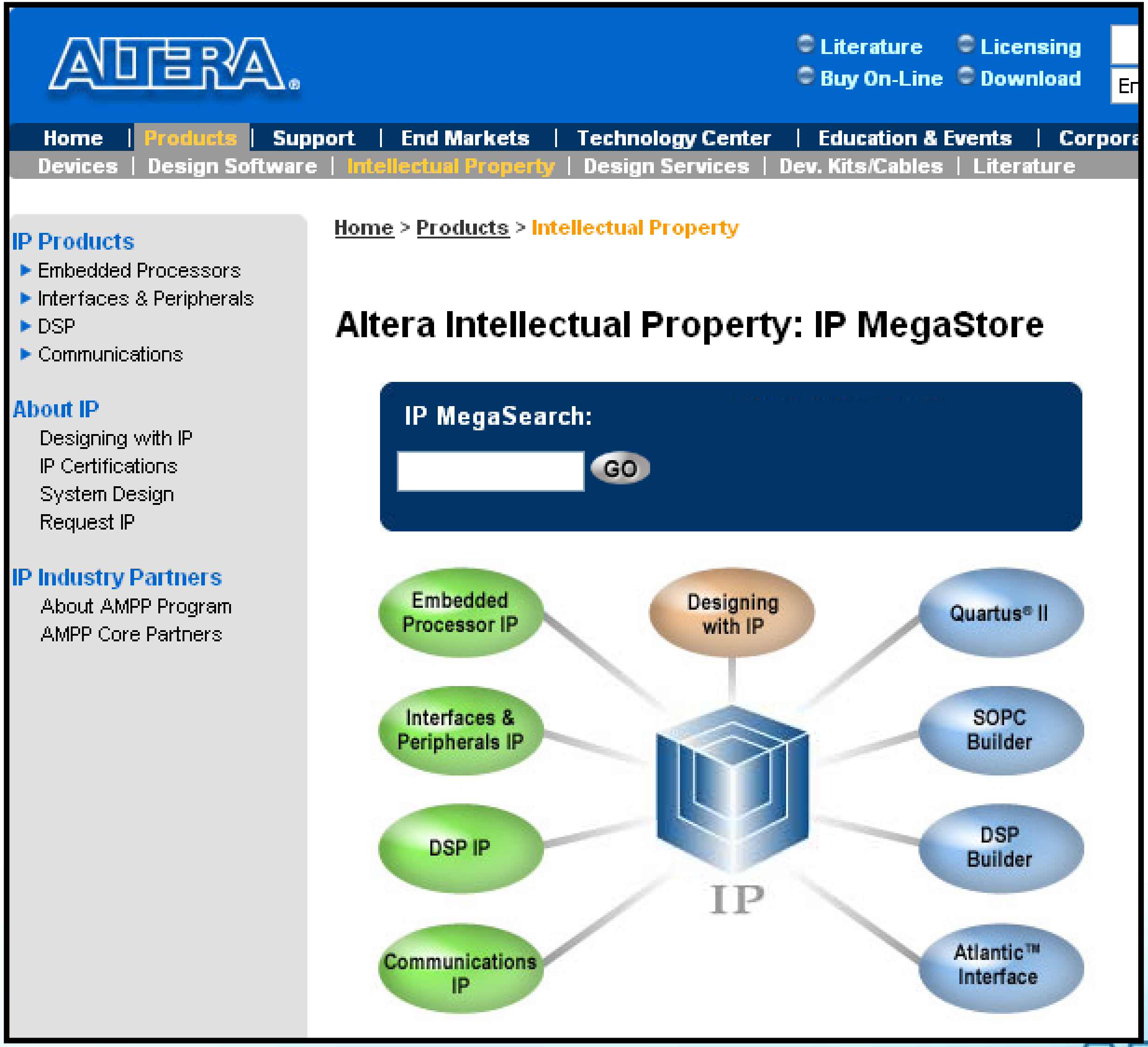
Programmable Logic Families

- Structured ASIC
  - HardCopy II & HardCopy Stratix devices
- High & medium density FPGAs
  - Stratix III, Stratix II & Stratix devices
- Low-cost FPGAs
  - Cyclone II & Cyclone devices
- FPGAs w/ clock data recovery
  - Stratix II GX & Stratix GX devices
- CPLDs
  - MAX II, MAX 7000 & MAX 3000 devices
- Configuration devices
  - Serial (EPCS) & enhanced (EPC)

Software & Development Tools

- Quartus II
  - Stratix III, Stratix II & Stratix devices
  - Stratix II GX & Stratix GX devices
  - Cyclone II & Cyclone devices
  - HardCopy II & HardCopy Stratix devices
  - MAX II, MAX 7000S/AE/B, MAX 3000A devices
  - Select older families
- Quartus II Web Edition
  - Free version
  - Not all features & devices included
    - See www.altera.com for feature comparison
- MAX+PLUS® II
  - All FLEX, ACEX, and MAX devices

Intellectual Property Megastore

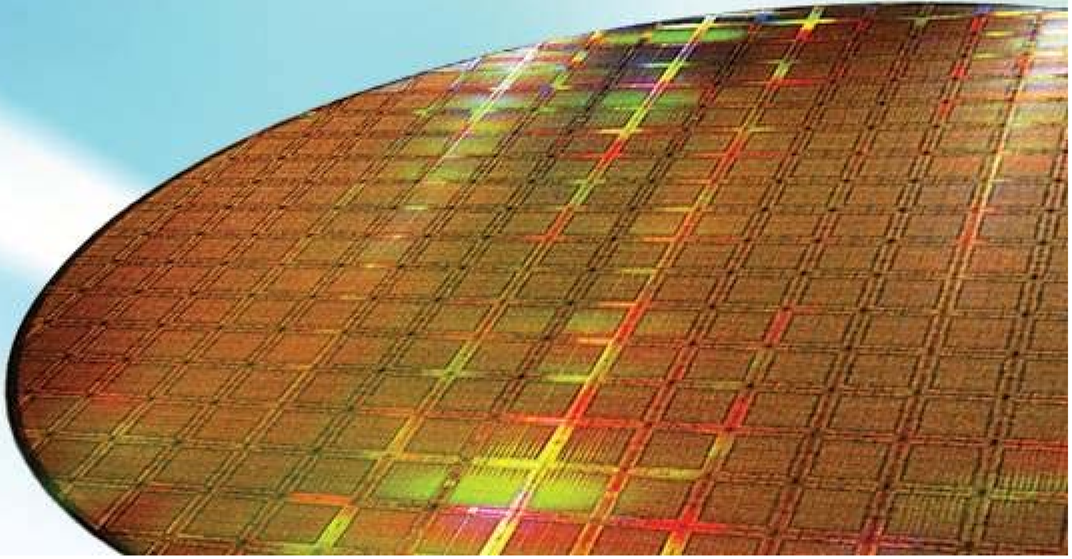


Home > Products > Intellectual Property

Altera Intellectual Property: IP MegaStore

Introduction to VHDL

VHDL Basics



VHDL

VHSIC (Very High Speed Integrated Circuit)

Hardware

Description

Language

What is VHDL?

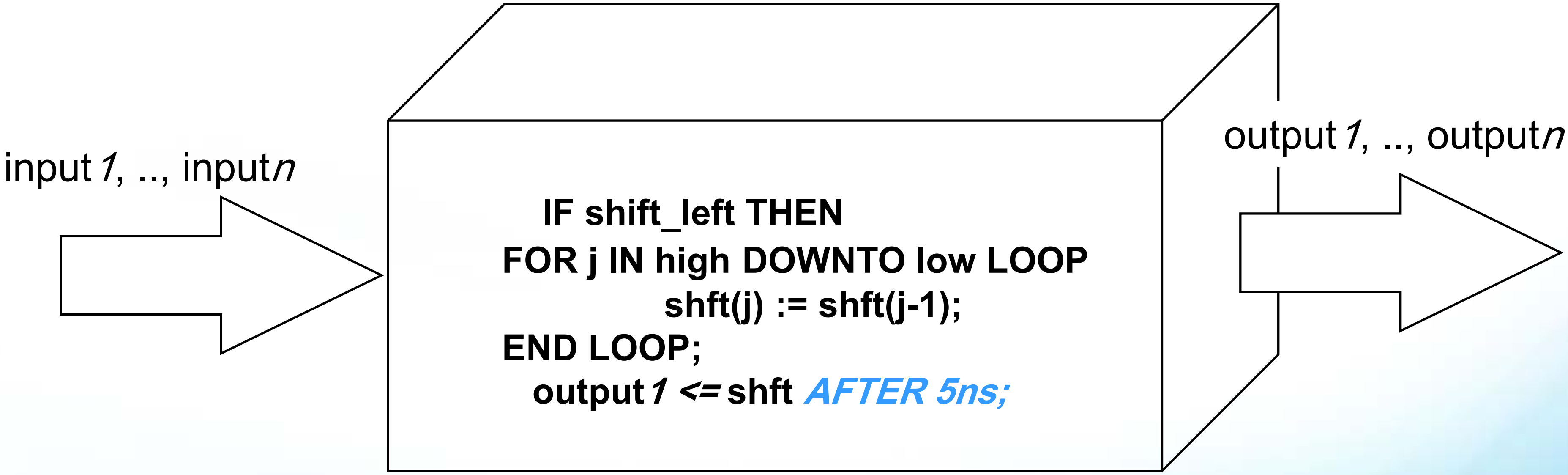
- IEEE industry standard hardware description language
- High-level description language for both simulation & synthesis

Terminology

- HDL - hardware description language is a software programming language that is used to model a piece of hardware
- Behavior modeling - A component is described by its input/output response
- Structural modeling - A component is described by interconnecting lower-level components/primitives

Behavior Modeling

- Only the functionality of the circuit, no structure
- No specific hardware intent
- For the purpose of synthesis, as well as *simulation*

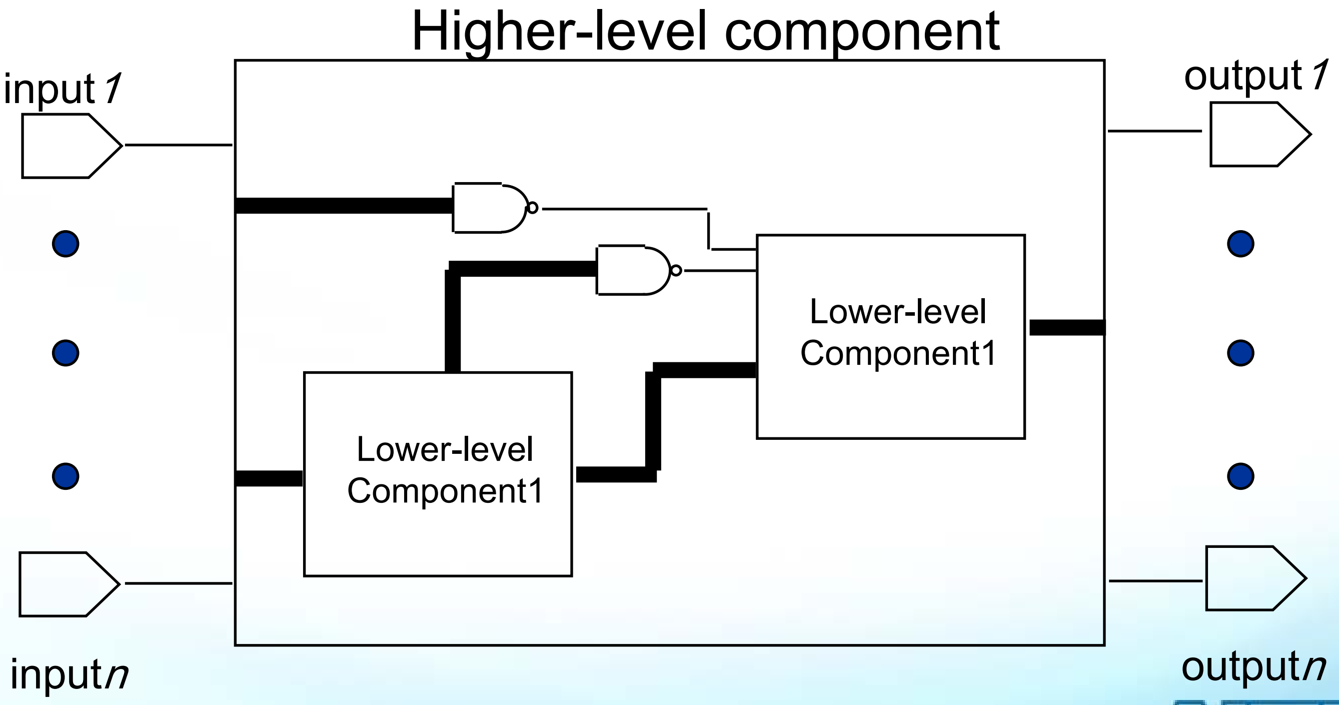


```
    IF shift_left THEN
FOR j IN high DOWNTO low LOOP
            shft(j) := shft(j-1);
END LOOP;
  output 1 <= shft AFTER 5ns;
```

Left bit shifter

Structural Modeling

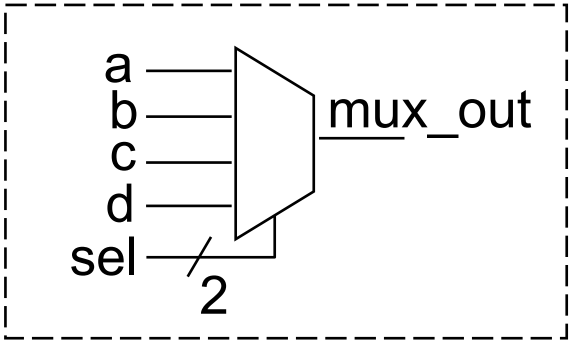
- Functionality and structure of the circuit
- Call out the specific hardware
- For the purpose of synthesis, as well as *simulation*



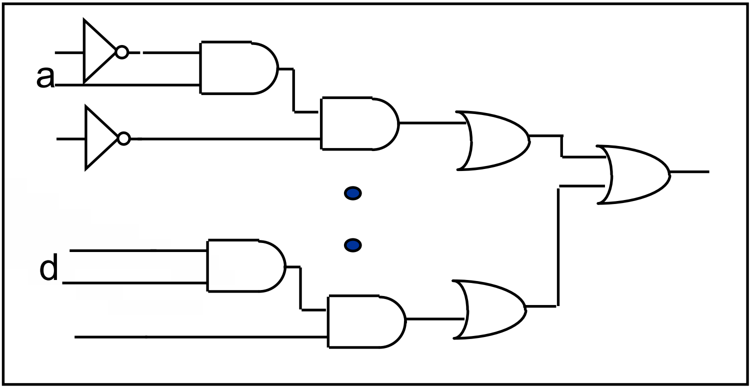
RTL Synthesis

```
Process (a, b, c, d, sel)
  begin
    case (sel) is
               when "00" => mux_out <= a;
               when "01" => mux_out <= b;
               when "10" => mux_out <= c;
               when "11" => mux_out <= d;
    end case;
```

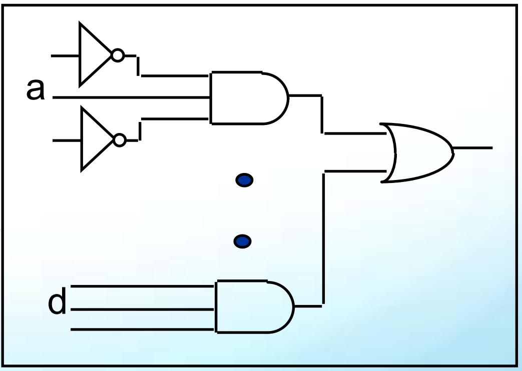
inferred



Translation



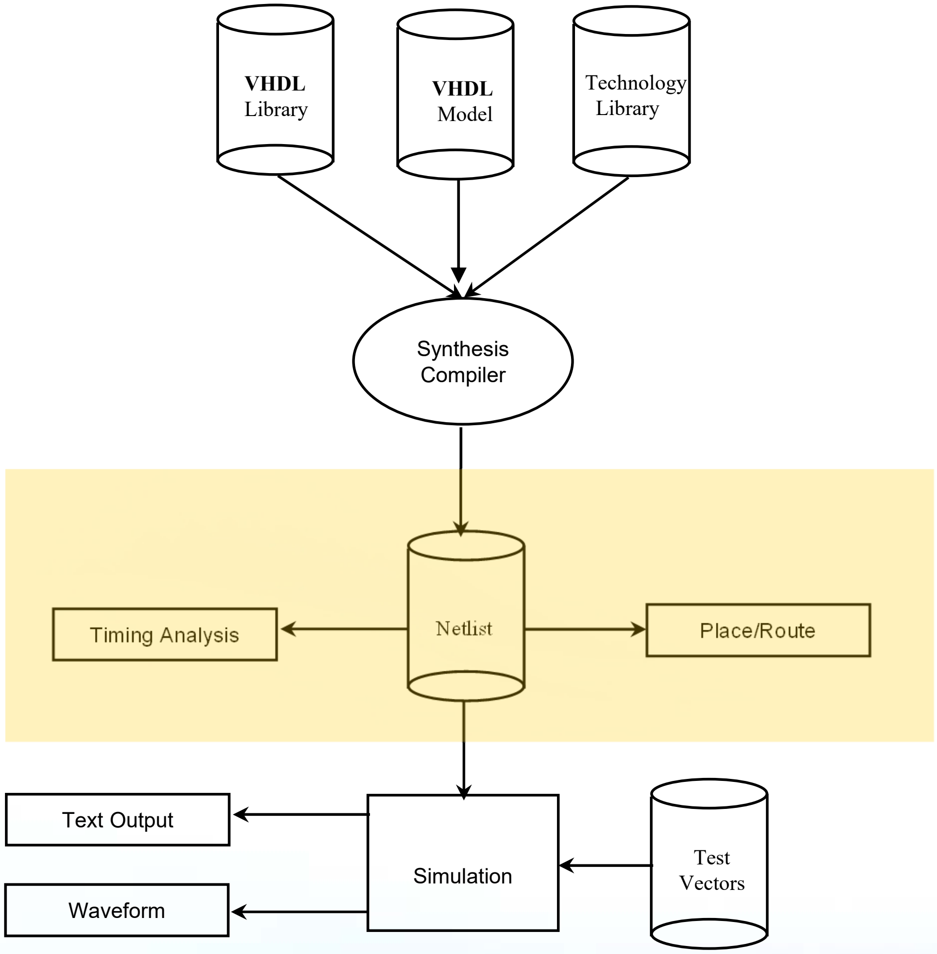
Optimization



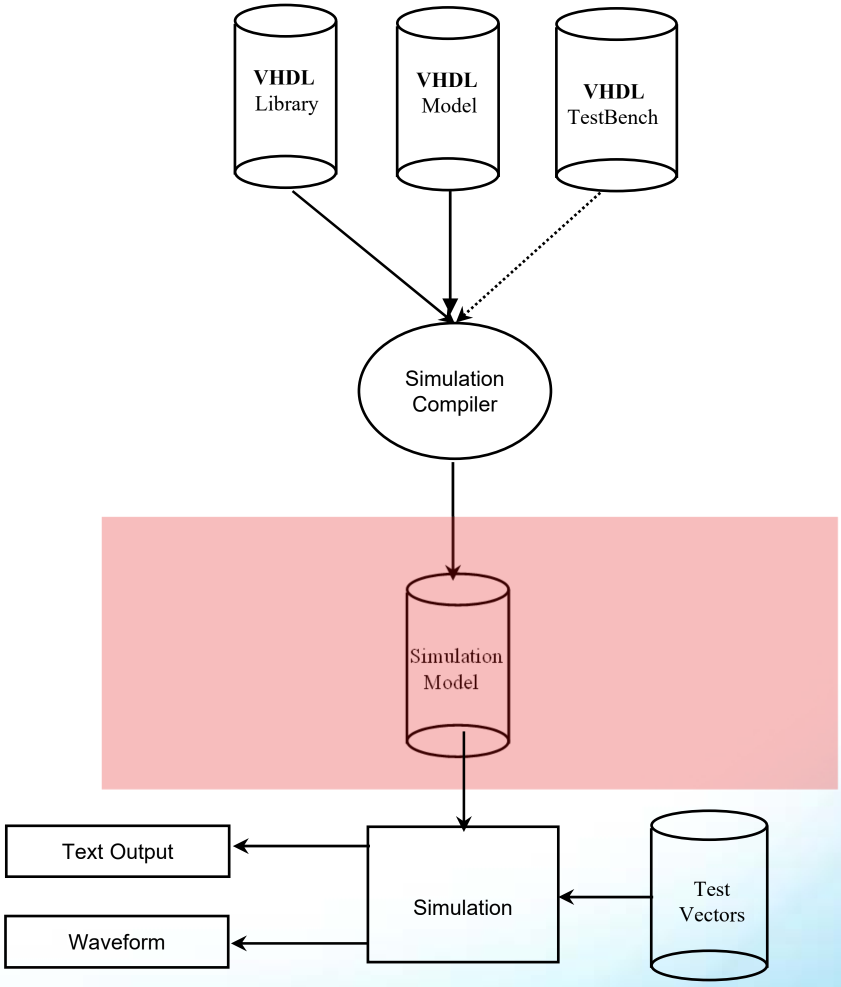
VHDL Synthesis Vs. Other HDL Standards

- Vhdl
  - “Tell me how your circuit should behave and I will give you hardware that does the job.”
- Verilog
  - Similar to VHDL
- ABEL, PALASM, AHDL
  - “Tell me what hardware you want and I will give it to you”

Typical Synthesis and Simulation Flows



Synthesis



Simulation

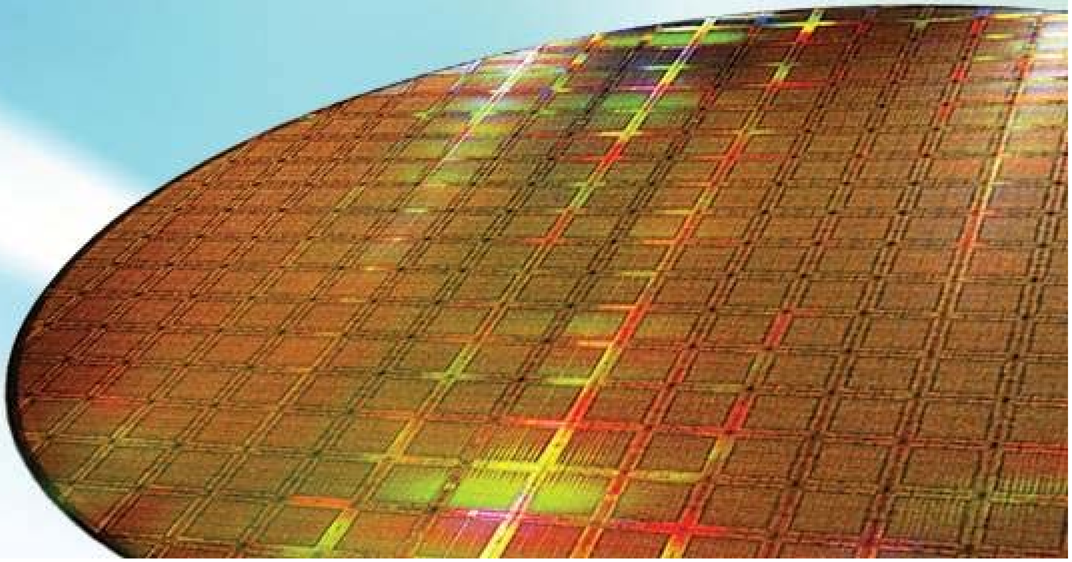
ALTERA®

VHDL Basics

- Two sets of constructs:
  - Simulation
  - Synthesis
- The VHDL language is made up of reserved keywords
- The language is, for the most part, **not** case sensitive
- VHDL statements are terminated with a **;**
- VHDL is white space insensitive
- Comments in VHDL begin with “**--**” to EOL
- VHDL models can be written:
  - Behavioral
  - Structural
  - Mixed

Introduction to VHDL

VHDL Design Units



VHDL Basics

- VHDL design units
  - Entity
    - Used to define external view of a model. i.e. symbol
  - Architecture
    - Used to define the function of the model. i.e. schematic
  - Configuration
    - Used to associate an architecture with an entity
  - Package
    - Collection of information that can be referenced by VHDL models. I.E. Library
    - Consist of two parts package declaration and package body

Entity Declaration

```
ENTITY  <entity_name>  IS
            Generic declarations
            Port Declarations
END <entity_name>; (1076-1987 version)
END ENTITY <entity_name> ; ( 1076-1993 version)
```

- Analogy : symbol
- <Entity_name> can be any alpha/numerical name
- Generic declarations
  - Used to pass information into a model
  - Quartus II & MAX+PLUS II place some restriction on the use of generics
- Port declarations
  - Used to describe the inputs and outputs i.e. pins

Entity : Generic Declaration

```
ENTITY  <entity_name> IS
        Generic ( constant tplh , tphl : time := 5 ns;
                -- Note constant is assumed and is not required
                  tphz, tplz : time := 3 ns;
                  default_value : integer := 1;
                  cnt_dir : string := “up”
                );
        Port Declarations
END <entity_name>; (1076-1987 version)
END ENTITY <entity_name> ; ( 1076-1993 version)
```

- New values can be passed during compilation
- During simulation/synthesis a generic is read only

Entity : Port Declarations

```
ENTITY  <entity_name>  IS
           Generic declarations
           Port ( signal clk : in  bit;
                   --Note: signal is assumed and is not required
                           q : out bit
                 );
END <entity_name>; (1076-1987 version)
END ENTITY <entity_name> ; ( 1076-1993 version)
```

- Structure : <class> object_name : <mode> <type> ;
  - <class> : what can be done to an object
  - Object_name : identifier
  - <mode> : directional
    - **in** (input) **out** (output)
    - **inout** (bidirectional) **buffer** (output W/ internal feedback)
  - <Type> : what can be contained in the object

Architecture

- Analogy : schematic
- Describes the functionality and timing of a model
- Must be associated with an **ENTITY**
- **ENTITY** can have multiple architectures
- Architecture statements execute concurrently (processes)
- Architecture styles
  - Behavioral : how designs operate
    - RTL : designs are described in terms of registers
    - Functional : no timing
  - Structural : netlist
    - Gate/component level
  - Hybrid : mixture of the above

Architecture

ARCHITECTURE <identifier> **OF** <entity_identifier> **IS**

--Architecture declaration section (list does not include all)

- **SIGNAL** temp **:** integer **:=** 1**;** -- signal declarations :=1 is default value optional
- **CONSTANT** load **:** boolean **:=** true**;** --constant declarations
- **TYPE** states **IS** (S1, S2, S3, S4**)** **;** --type declarations
- --Component declarations discussed later
- --Subtype declarations
- --Attribute declarations
- --Attribute specifications
- --Subprogram declarations
- --Subprogram body

Begin

- Process statements
- Concurrent procedural calls
- Concurrent signal assignment
- Component instantiation statements
- Generate statements

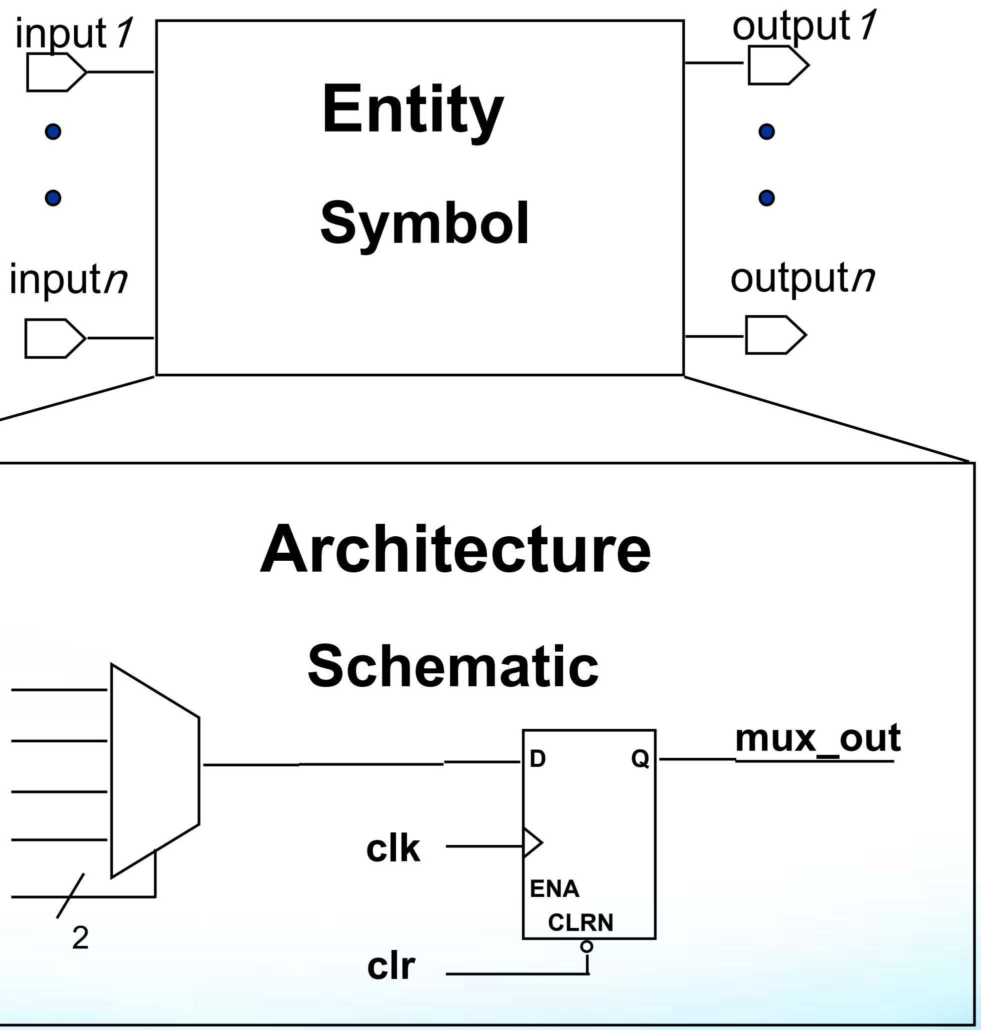
END <architecture identifier> **;** *(1076-1987 version)*

End ARCHITECTURE; *(1076-1993 version)*

VHDL - Basic Modeling Structure

```
ENTITY entity_name IS
        generics
        port declarations
END entity_name;

ARCHITECTURE arch_name OF entity_name IS
        enumerated data types
        internal signal declarations
        component declarations
BEGIN
        signal assignment statements
        process statements
        component instantiations
END arch_name;
```

VHDL : Entity - Architecture



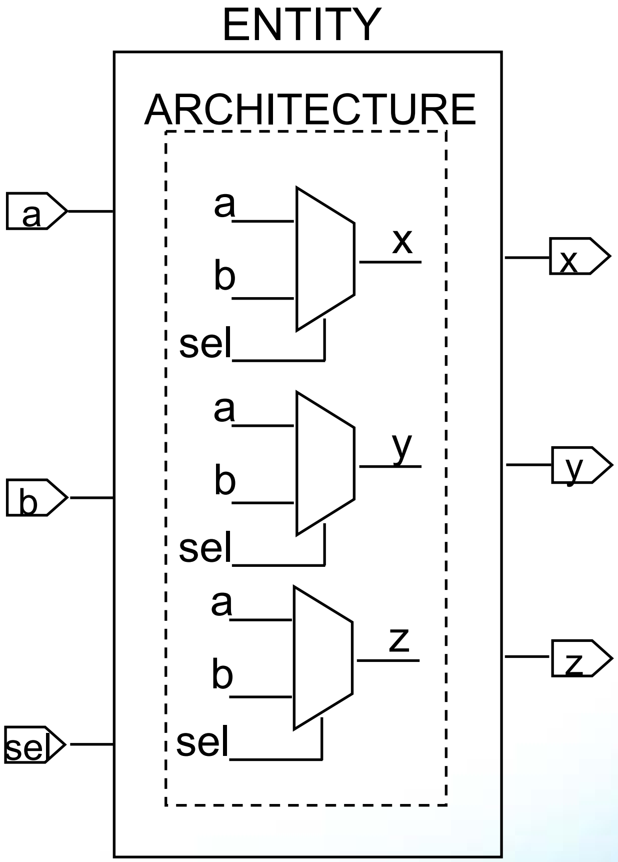
Configuration

- Used to make associations within models
  - Associate a entity and architecture
  - Associate a component to an entity-architecture
- Widely used in simulation environments
  - Provides a flexible and fast path to design alternatives
- Limited or no support in synthesis environments

```
CONFIGURATION <identifier> OF <entity_name> IS
        FOR <architecture_name>
        END FOR;
END; (1076-1987 version)
END CONFIGURATION; (1076-1993 version)
```

Putting It All Together

```
ENTITY cmpl_sig IS
PORT ( a, b, sel : IN bit;
           x, y, z : OUT bit);
END cmpl_sig;
ARCHITECTURE logic OF cmpl_sig IS
BEGIN
             -- simple signal assignment
           x <= (a AND NOT sel) OR (b AND sel);
             -- conditional signal assignment
           y <= a WHEN sel='0' ELSE
                b;
             -- selected signal assignment
           WITH sel SELECT
                     z <= a WHEN '0',
                          b WHEN '1',
                          '0' WHEN OTHERS;
END logic;
CONFIGURATION cmpl_sig_conf OF cmpl_sig IS
           FOR logic
           END FOR;
END cmpl_sig_conf;
```



Packages

- Packages are a convenient way of storing and using information throughout an entire model
- Packages consist of:
  - Package declaration (required)
    - Type declarations
    - Subprograms declarations
  - Package body (optional)
    - Subprogram definitions
- VHDL has two built-in packages
  - Standard
  - Textio

Packages

PACKAGE <package_name> **IS**

Constant declarations

Type declarations

Signal declarations

Subprogram declarations

Component declarations

--There are other declarations

END <package_name> **;** (1076-1987)

END PACKAGE <package_name> **;** (1076-1993)

PACKAGE BODY <package_name> **IS**

constant declarations

Type declarations

Subprogram body

END <package_name> **;** (1076-1987)

END PACKAGE BODY <package_name> **;** (1076-1993)

Package Example

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
PACKAGE filt_cmp IS
    TYPE state_type IS (idle, tap1, tap2, tap3, tap4);
    COMPONENT acc
                Port(xh : in Std_logic_vector(10 downto 0);
                      Clk, first: in Std_logic;
                       Yn : out Std_logic_vector(11 downto 4));
     End component;
FUNCTION compare (SIGNAL a , b : integer) RETURN boolean;
END filt_cmp;
PACKAGE BODY filt_cmp IS
FUNCTION compare (SIGNAL a , b : integer) RETURN boolean IS
   VARIABLE temp : boolean;
 Begin
                if a < b then
                    temp := true;
              Else
                    temp := false;
                End if;
              RETURN temp;
END compare;
END filt_cmp;
```

Package declaration

Package body

Libraries

- Contains a package or a collection of packages
- Resource libraries
  - Standard package
  - IEEE developed packages
  - Altera component packages
  - Any library of design units that are referenced in a design
- Working library
  - Library into which the unit is being compiled

Model Referencing of Library/Package

- All packages must be compiled
- Implicit libraries
  - Work
  - Std
  - ⇨ Note: items in these packages do not need to be referenced, they are implied
- **LIBRARY** clause
  - Defines the library name that can be referenced
  - Is a symbolic name to path/directory
  - Defined by the compiler tool
- **USE** clause
  - Specifies the package and object in the library that you have specified in the library clause

Example

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY cmpl_sig IS
PORT ( a, b, sel : IN Std_logic;
             X, y, z : OUT Std_logic);
END cmpl_sig;
ARCHITECTURE logic OF cmpl_sig IS
Begin
              -- Simple signal assignment
             X <= (a AND NOT sel) OR (b AND sel);
              -- Conditional signal assignment
             Y <= a WHEN sel='0' ELSE
                  B;
              -- Selected signal assignment
             WITH sel SELECT
                         Z <= a WHEN '0',
                              B WHEN '1',
                             '0' when others;
END logic;
CONFIGURATION cmpl_sig_conf OF cmpl_sig IS
             FOR logic
             End for;
END cmpl_sig_conf;
```

- **LIBRARY** <name>, <name> ;
  - Name is symbolic and defined by compiler tool
  - ⇨ Note: remember that WORK and STD do not need to be defined.
- **Use** lib_name.Pack_name.Object;
  - **All** is a reserved word
- Placing the library/use clause first will allow all following design units to access it

Libraries

- **Library** std;
  - Contains the following packages:
    - **Standard** (types: bit, boolean, integer, real, and time; all operator functions to support types)
    - **Textio** (file operations)
  - An implicit library (built-in)
    - Does not need to be referenced in VHDL design

Types Defined in Standard Package

- Type bit
  - 2 logic value system (‘0’, ‘1’)

    Signal a_temp : bit;
  - Bit_vector array of bits

    Signal temp : bit_vector(3 **downto** 0);

    Signal temp : bit_vector(0 **to** 3) ;
- Type boolean
  - (False, true)
- Integer
  - Positive and negative values in decimal

    Signal int_tmp : integer; -- 32 bit number

    Signal int_tmp1 : integer **range** 0 to 255; --8 bit number

⇨ Note: standard package has other types

Libraries

- **Library** IEEE;
  - Contains the following packages:
    - **Std_logic_1164** (Std_logic types & related functions)
    - **Std_logic_arith** (arithmetic functions)
    - **Std_logic_signed** (signed arithmetic functions)
    - **Std_logic_unsigned** (unsigned arithmetic functions)

Types Defined in Std_logic_1164 Package

- Type **Std_logic**
  - 9 logic value system (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’)
    - ‘W’, ‘L’, ‘H” weak values (not supported by synthesis)
    - ‘X’ - used for unknown
    - ‘Z’ - (not ‘z’) used for tri-state
    - ‘-’ Don’t care
  - Resolved type: supports signals with multiple drives
- Type **std_ulogic**
  - Same 9 value system as Std_logic
  - Unresolved type: does not support multiple signal drives; error will occur

- 'U': uninitialized. This signal hasn't been set yet.
- 'X': unknown. Impossible to determine this value/result.
- '0': logic 0
- '1': logic 1
- 'Z': High Impedance
- 'W': Weak signal, can't tell if it should be 0 or 1.
- 'L': Weak signal that should probably go to 0
- 'H': Weak signal that should probably go to 1
- '-': Don't care

User-defined Libraries/Packages

- User-defined packages can be in the same directory as the design

  Library work; *--optional*
  USE WORK.*<Package name>***.All**;

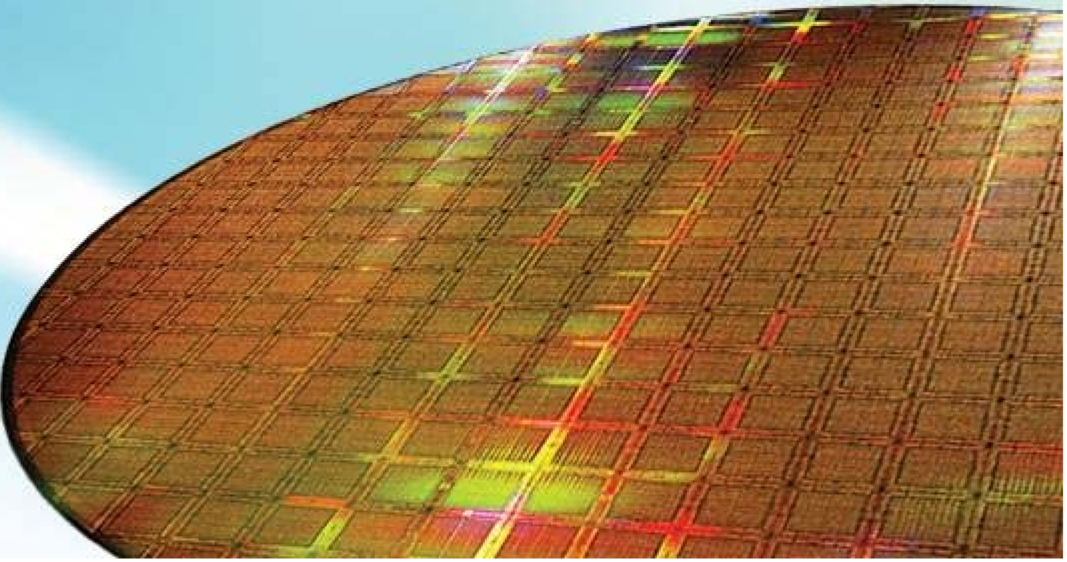
- Or can be in a different directory from the design

  LIBRARY <*any_name*>**;**

  Use <*any_name>.<Package_name>***.All;**

Introduction to VHDL

Architecture Modeling Fundamentals

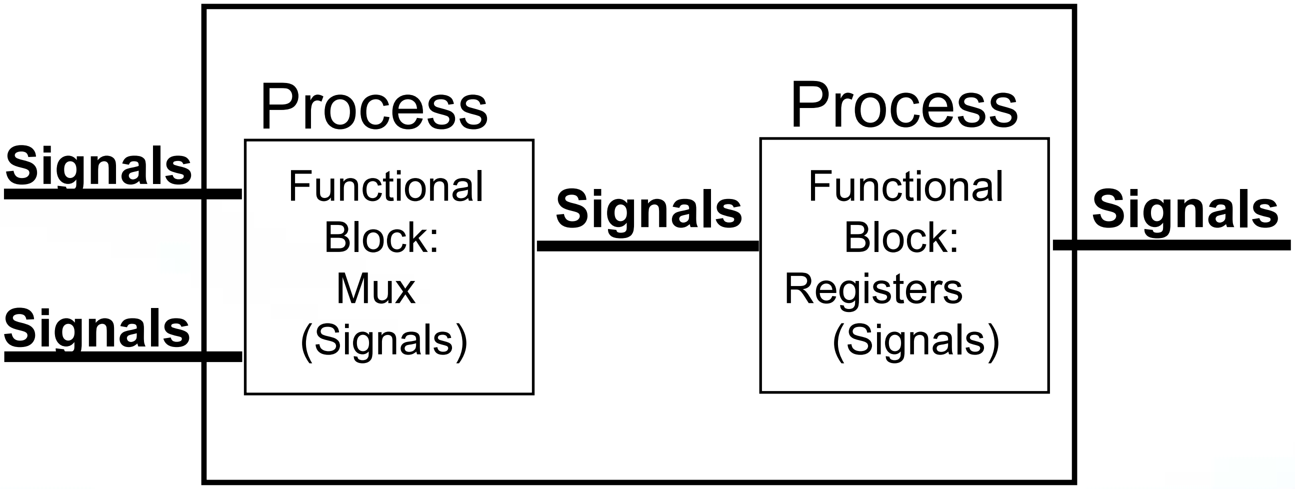


Section Overview

- Understanding the concept and usage of signals
  - Signal assignments
  - Concurrent signal assignment statements
  - Signal delays
- Processes
  - Implied
  - Explicit
- Understanding the concept and usage of variables
- Sequential statement
  - If-then
  - Case
  - Loops

Using Signals

- Signals represent physical interconnect (wire) that communicate between processes (functions)
- Signals can be declared in **packages**, **entity** and **architecture**


Process
Process
Signals
Signals
Functional Block: Mux (Signals)
Signals
Functional Block: Registers (Signals)
Signals

Assigning Values to Signals

```
SIGNAL   temp  :   Std_logic_VECTOR (7 downto 0);
```

- All bits:

```
        Temp  <=  “10101010”;
        temp <= x”aa” ; (1076-1993)
```

- Single bit:

```
        Temp(7)  <=  ‘1’;
```

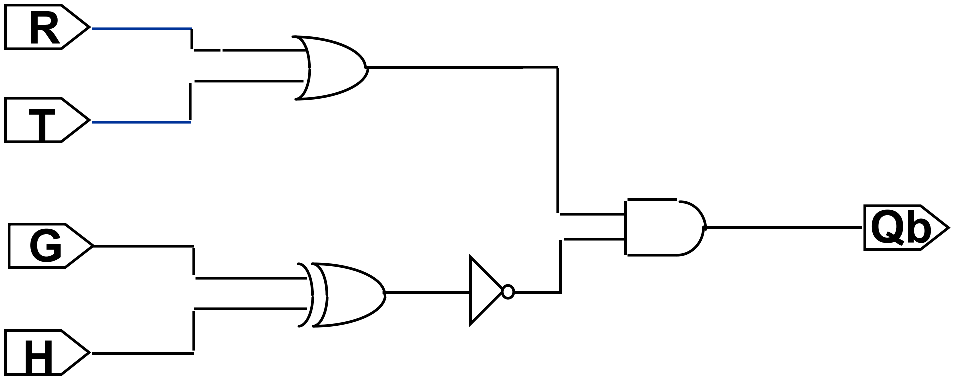
- Bit-slicing:

```
        Temp (7 downto 4)  <=  “1010”;
```

- Single-bit: single-quote (‘)
- Multi-bit: double-quote (“)

Signal Used as an Interconnect

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp IS
Port(r, t, g, h : IN Std_logic;
             Qb : OUT Std_logic);
END simp;
ARCHITECTURE logic OF simp IS
SIGNAL Qa : Std_logic;

Begin

Qa <= r or t;
Qb <= (qa and not(g xor h));

End logic;
```

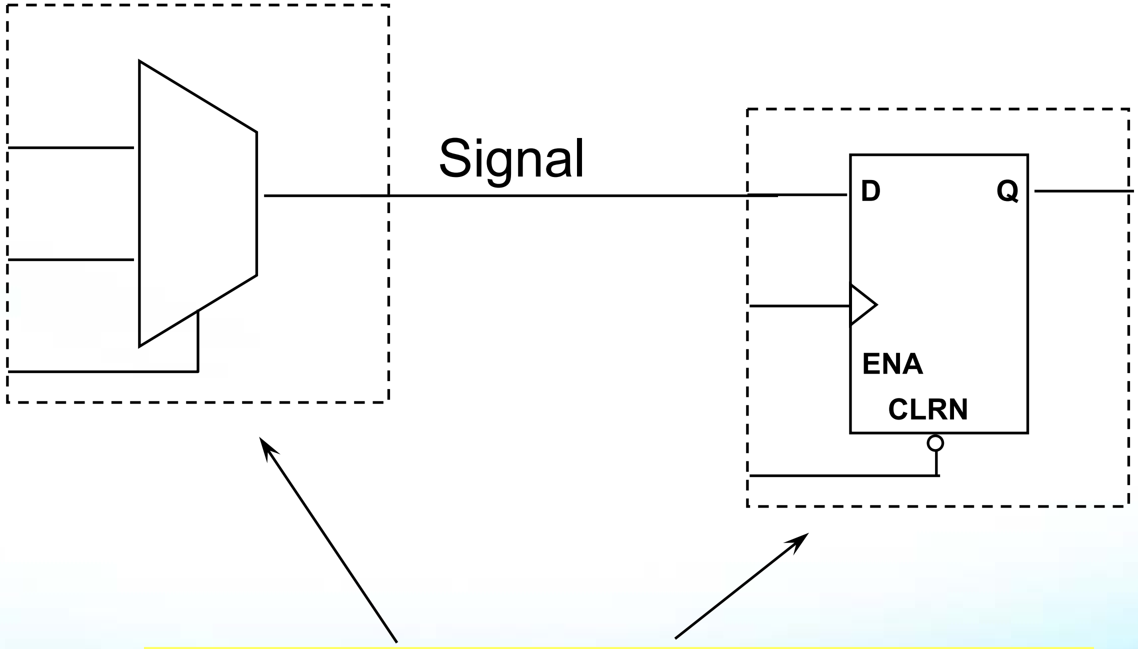


- **R, T, G, H**, and **Qb** are signals (by default)
- **Qa** is a buried signal and needs to be declared

Signal declaration inside architecture

Signal Assignments

- Signal assignments are represented by: **<=**
- Signal assignments have an ***implied*** process (function) that synthesizes to hardware



Signal assignment **<=** implied process

Concurrent Signal Assignments

- Three concurrent signal assignments:
  - Simple signal assignment
  - Conditional signal assignment
  - Selected signal assignment

Simple Signal Assignments

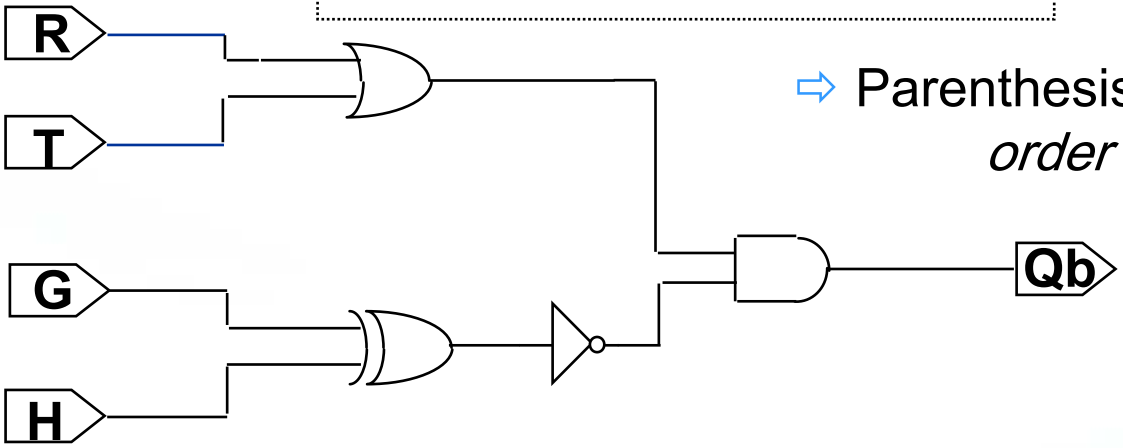
- Format:

```
<Signal_name> <= <expression>;
```

- Example:

```
Qa <=  r or t ;
Qb <=  (qa and not(g xor h));
```

Implied process



⇨ Parenthesis **( )** *give the order of operation*

- VHDL operators are used to describe the process

VHDL Operators

Operator type	operator name/symbol
Logical	and or nand nor xor xnor
Relational	= /= < <= > >=
Addition & concatenation	+ - &
Signing	+ -
Multiplying	* / mod rem
Miscellaneous	** abs not

VHDL Operators

- VHDL defines arithmetic & boolean functions only for built-in data types (defined in ***standard*** package)
  - Arithmetic operators such as **+**, **-**, **<**, **>**, **<=**, **>=** are defined ***only*** for **INTEGER** type
  - Boolean operators such as **AND**, **OR**, **NOT** are defined ***only*** for **BIT** type
- Recall: vhdl implicit library (built-in)
  - **Library STD**
    - Types defined in the **standard** package:
      - **Bit, boolean, integer**

⇨ Note: items in this package do not need to be referenced, they are implied

Arithmetic Function

```
ENTITY opr IS
        PORT ( a   : IN  INTEGER RANGE 0 TO 16;
            B   : IN  INTEGER RANGE 0 TO 16;
            Sum : OUT  INTEGER RANGE 0 TO 32);
END opr;

Architecture example of opr is
Begin
        Sum <= a + b;
END example;
```

The VHDL compiler can Understand this operation Because an arithmetic Operation is defined for The built-in data type ***Integer***

⇨ Note: remember the library **STD** and the package **standard** do not need to be referenced

Operator Overloading

- How do you use arithmetic & boolean functions with other data types?
  - ***Operator overloading*** - defining arithmetic & boolean functions with other data types
- Operators are overloaded by defining a function whose name is the same as the operator itself
  - Because the operator and function name are the same, the function name must be enclosed within double quotes to distinguish it from the actual VHDL operator
  - The function is normally declared in a package so that it is globally visible for any design

Operator Overloading Function/Package

- Packages that define these operator overloading functions can be found in the **LIBRARY IEEE**
- For example, the package ***Std_logic_unsigned*** defines some of the following functions

```
Package Std_logic_unsigned is

  Function "+"(l: Std_logic_vector; r: Std_logic_vector) return Std_logic_vector;
  Function "+"(L: Std_logic_VECTOR; R: INTEGER) return Std_logic_VECTOR;
  Function "+"(L: INTEGER; R: Std_logic_VECTOR) return Std_logic_VECTOR;
  Function "+"(L: Std_logic_VECTOR; R: Std_logic) return Std_logic_VECTOR;
  Function "+"(L: Std_logic; R: Std_logic_VECTOR) return Std_logic_VECTOR;

  Function "-"(l: Std_logic_vector; r: Std_logic_vector) return Std_logic_vector;
  Function "-"(L: Std_logic_VECTOR; R: INTEGER) return Std_logic_VECTOR;
  Function "-"(L: INTEGER; R: Std_logic_VECTOR) return Std_logic_VECTOR;
  Function "-"(L: Std_logic_VECTOR; R: Std_logic) return Std_logic_VECTOR;
  Function "-"(L: Std_logic; R: Std_logic_VECTOR) return Std_logic_VECTOR;
```

Use of Operator Overloading

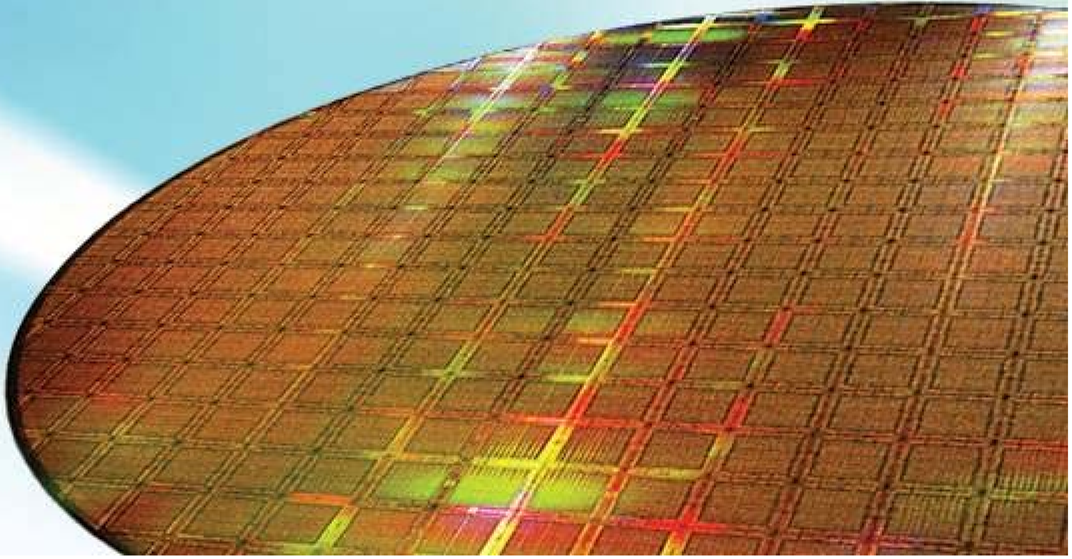
```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;

Entity overload is
        port ( a   : in Std_logic_vector (4 downto 0);
               B   : IN Std_logic_VECTOR (4 downto 0);
               Sum : OUT Std_logic_VECTOR (4 downto 0));
END overload;

Architecture example of overload is
Begin
        Sum <= a + b;
END example;
```

Include these statements at the beginning of a design file

This allows us to perform arithmetic on non-built-in data types

Exercise 1

Please Go to Exercise 1



Concurrent Signal Assignments

- Three concurrent signal assignments:
  - Simple signal assignment
  - Conditional signal assignment
  - Selected signal assignment

Conditional Signal Assignments

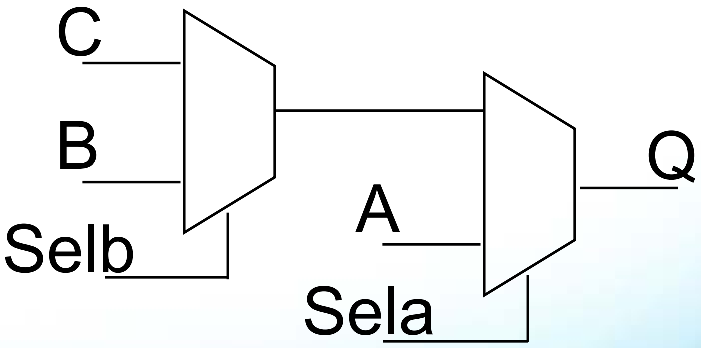
- Format:

```
<Signal_name>  <=  <signal/value> when <condition1> else
                   <signal/value> when <condition2> else
                                  .
                                  .
                   <Signal/value> when <condition3> else
                   <Signal/value>;
```

- Example:

```
Q <=    A WHEN sela = ‘1’ ELSE
        B WHEN selb = ‘1’ ELSE
        C;
```

Implied process



Selected Signal Assignments

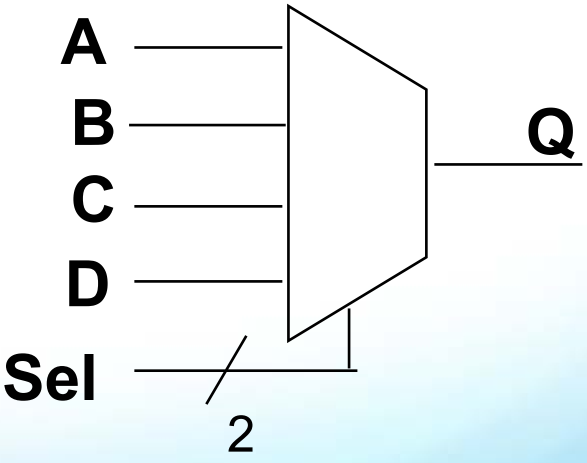
- Format:

```
With <expression> select
<Signal_name> <=      <signal/value> when <condition1>,
                      <Signal/value> when <condition2>,
                                                .
                                                .
                      <Signal/value> when others;
```

- Example:

```
WITH  sel SELECT
  Q <=  A WHEN "00",
        B WHEN "01",
        C WHEN "10",
        D WHEN OTHERS;
```

Implied process



Selected Signal Assignments

- **All** possible conditions must be considered
- **WHEN OTHERS** clause evaluates all other possible conditions that are not specifically stated

See next slide

Selected Signal Assignment

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity cmpl_sig is
PORT ( a, b, sel : IN Std_logic;
                Z : OUT Std_logic);
END cmpl_sig;

Architecture logic of cmpl_sig is
Begin
                -- Selected signal assignment
              WITH sel SELECT
                      Z <= a WHEN '0',
                          b WHEN '1',
                          '0' when others;
END logic;
```

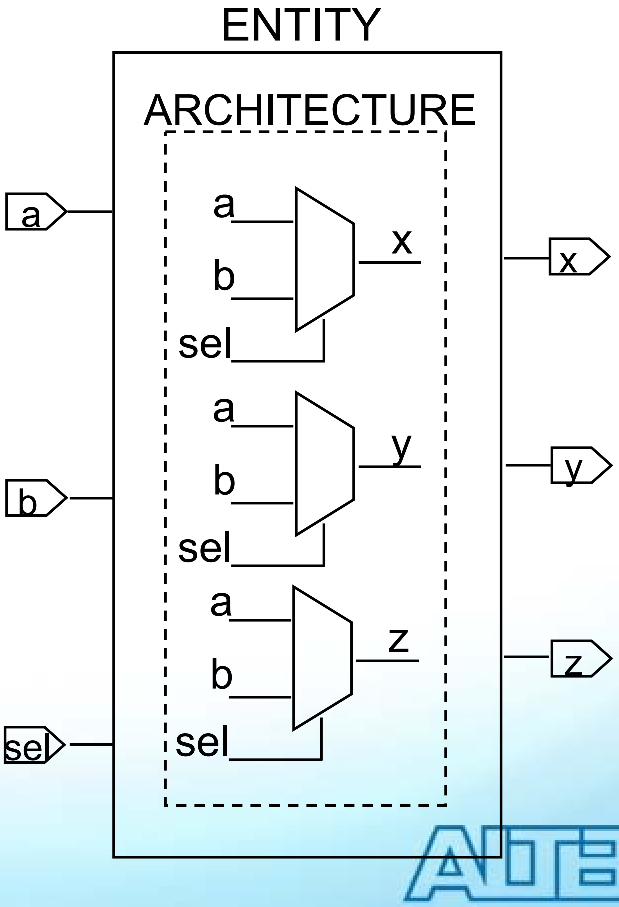
Sel has a ***Std_logic*** data type

- *What are the values for a* ***Std_logic*** *data type*
- *Answer:* **{'0','1','X','Z'}**
- *Therefore, is the* ***WHEN OTHERS*** *Clause necessary?*
- *Answer:* **YES**

VHDL Model - Concurrent Signal Assignments

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity cmpl_sig is
PORT ( a, b, sel : IN Std_logic;
           X, y, z : OUT Std_logic);
END cmpl_sig;

Architecture logic of cmpl_sig is
Begin
             -- Simple signal assignment
           X <= (a AND NOT sel) OR (b AND sel);
             -- Conditional signal assignment
           Y <= a WHEN sel='0' ELSE
                b;
             -- Selected signal assignment
           WITH sel SELECT
                     Z <= a WHEN '0',
                          b WHEN '1',
                         '0' when others;
END logic;
```

- The signal assignments execute in parallel, and therefore the order we list the statements should not affect the outcome



Explicit Process Statement

- Process can be thought of as
  - *Implied processes*
  - *Explicit processes*
- Implied process consist of
  - Concurrent signal assignment statements
  - Component statements
  - Processes' sensitivity is read side of expression
- Explicit process
  - Concurrent statement
  - Consist of sequential statements only

```
--    Explicit process statement
PROCESS (sensitivity_list)
      Constant declarations
       Type declarations
       Variable declarations
                 Begin
                 --  Sequential statement #1;
                 --  ……..
                 --  Sequential statement #n ;
                 end process;
```

Execution of Process Statement

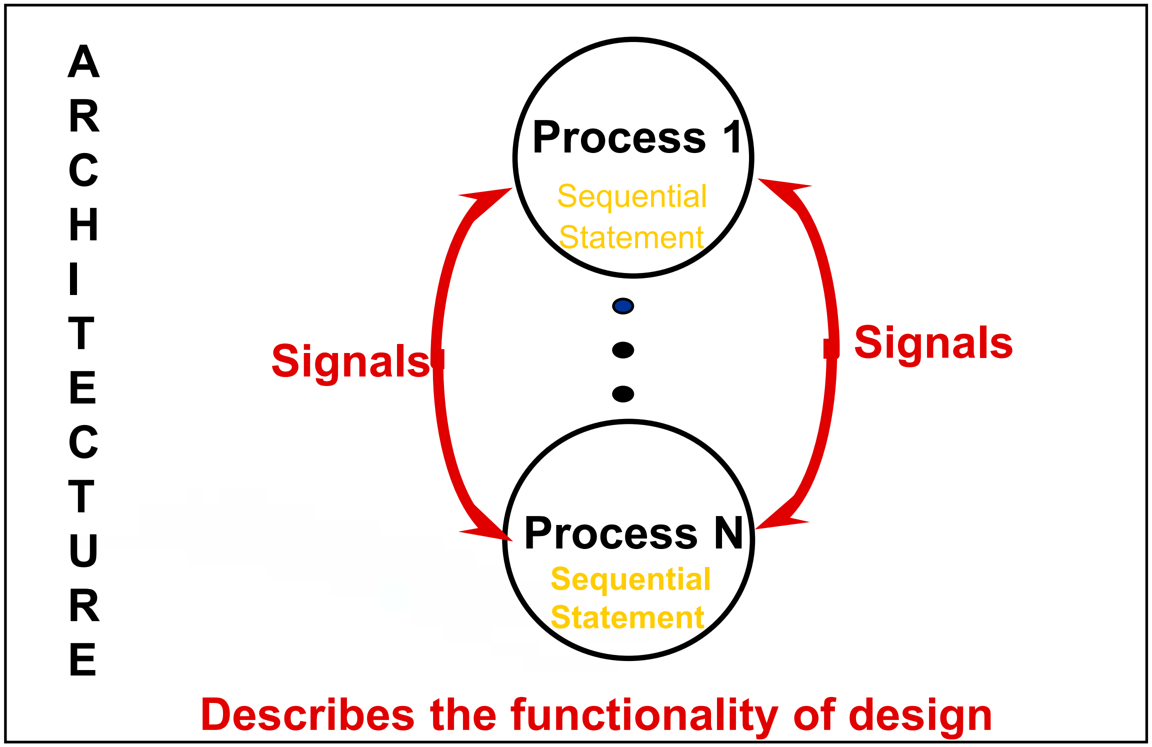
- Process statement is executed infinitely unless broken by a WAIT statement or sensitivity list
  - Sensitivity list implies a WAIT statement at the end of the process
  - Process can have multiple WAIT statements
  - Process can not have both a sensitivity list and WAIT statement

⇨ Note: logic synthesis places restrictions on wait and sensitivity list

```
PROCESS (a,b)
    Begin
       --Sequential statements
   End process;
```

```
Process
      Begin
           -- Sequential statements
      WAIT ON (a,b) ;
       END PROCESS;
```

Multi-Process Statements



Describes the functionality of design

- An architecture can have multiple process statements
- Each process executes in parallel with each other
- Within a process, the statements are executed sequentially

VHDL Model - Multi-Process Architecture

- The process statements execute in parallel and therefore, the order in which we list the statements should have no affect on the outcome

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity if_case is
PORT ( a, b, c, d : IN Std_logic;
          Sel : IN Std_logic_VECTOR(1 DOWNTO 0);
          Y, z : OUT Std_logic);
END if_case;

Architecture logic of if_case is
Begin
process(a, b, c, d, sel)
          Begin
                    IF sel="00" THEN
                              Y <= a;
                    ELSIF sel="01" THEN
                              Y <= b;
                    ELSIF sel="10" THEN
                              Y <= c;
                    Else
                              Y <= d;
                    End if;
END PROCESS;
```

```
process(a, b, c, d, sel)
          Begin
           CASE sel IS
                   When "00" =>
                             z <= a;
                   When "01" =>
                             z <= b;
                   When "10" =>
                             z <= c;
                   When "11" =>
                             z <= d;
                   When others =>
                             z <= '0';
           End case;
END PROCESS END logic;
```

- Within a process, the statements are executed sequentially

- Signal assignments can also be inside process statements

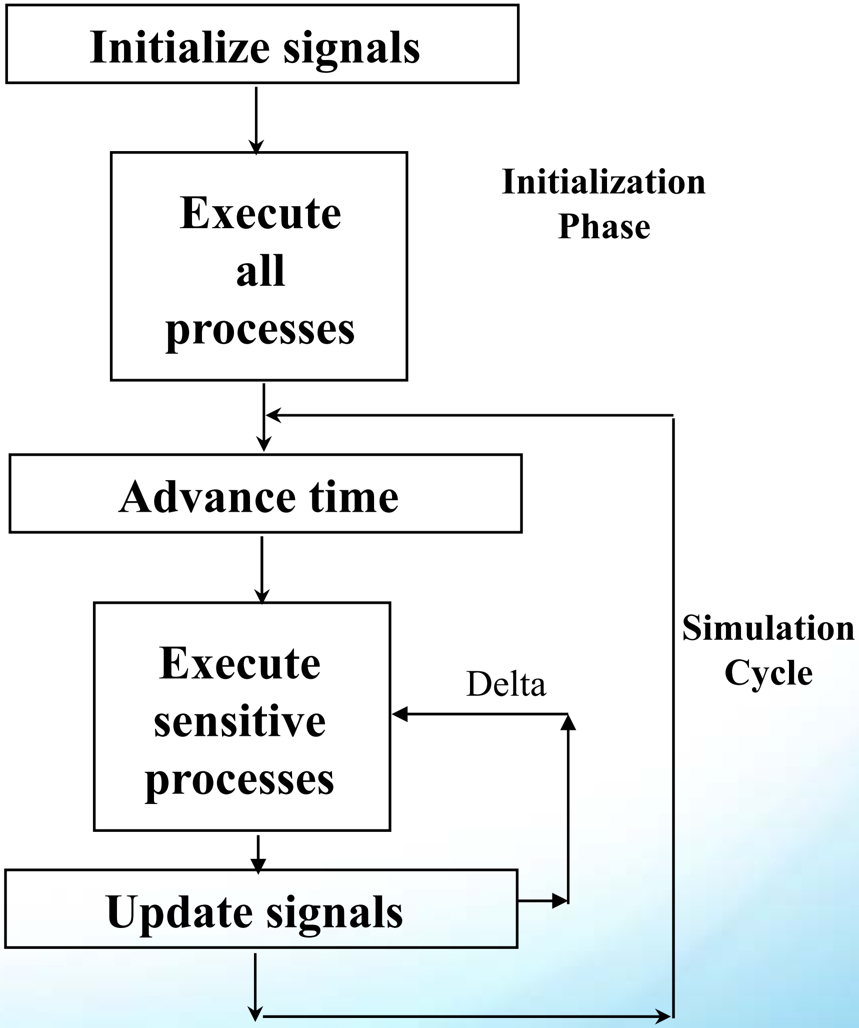
Signal Assignment - Delay

- Signal assignments can be inside process statements or outside (like the three concurrent signal assignments)
- Signal assignments incur delay
  - Two types of delays
    - Inertial delay (default)
      - A pulse that is short in duration of the propagation delay will not be transmitted
      - Ex. `A <= b AFTER 10 ns;`
    - Transport delay
      - Any pulse is transmitted no matter how short
      - Ex. `A <= TRANSPORT b AFTER 10 ns;`

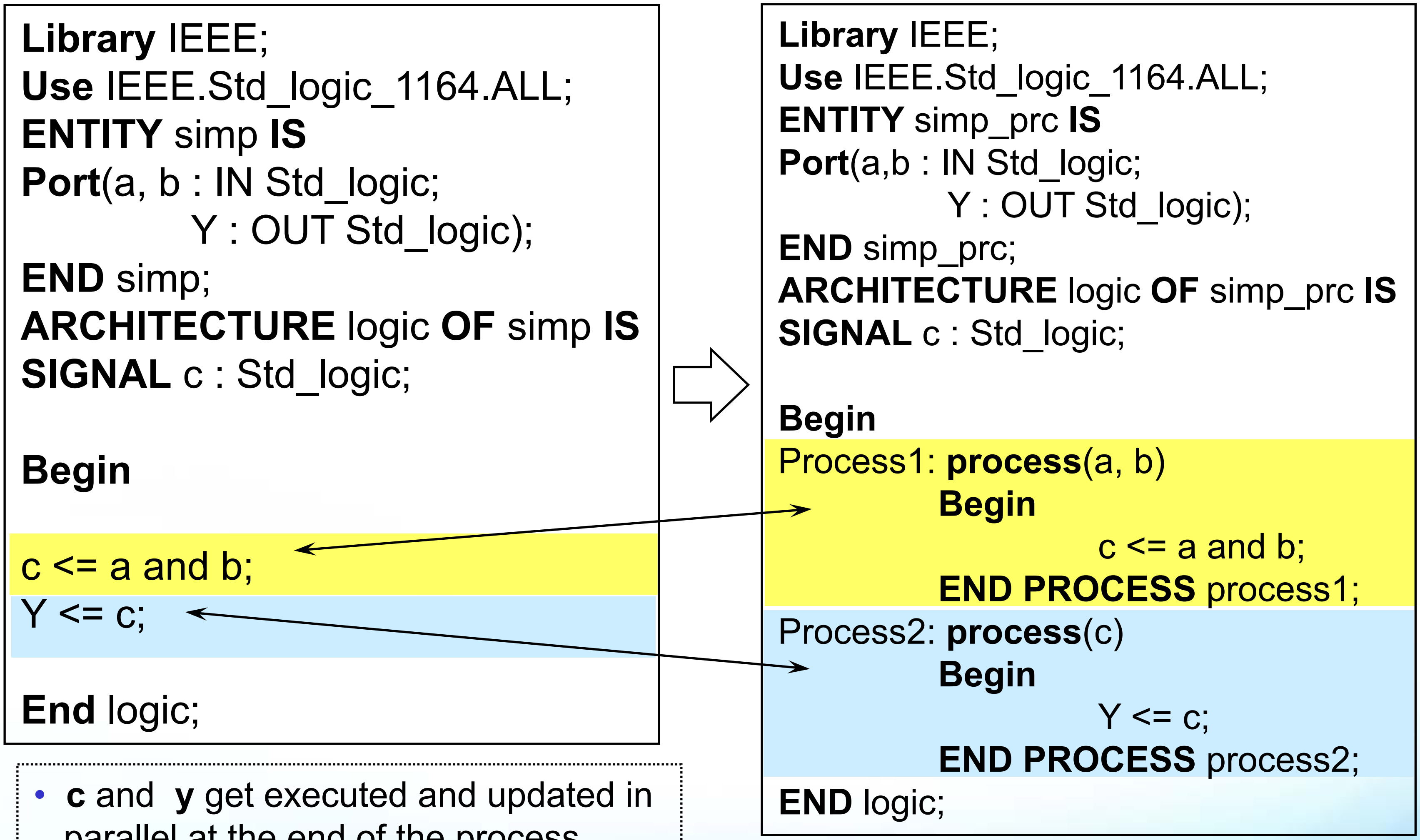
  ⇨ In VHDL, there are exceptions to this rule that will not be discussed

VHDL Simulation

- Event - A change in value: from 0 to 1; or from X to 1, etc.
- Simulation cycle
  - Wall clock time
  - Delta
    - Process execution phase
    - Signal update phase
- When does a simulation cycle end and a new one begin?
  - ⇨ **When:**
    - **All processes execute**
    - **Signals are updated**
- Signals get updated at the end of the delta cycle (delay)
  - **Typically end of process unless wait statement is used**



Equivalent Functions

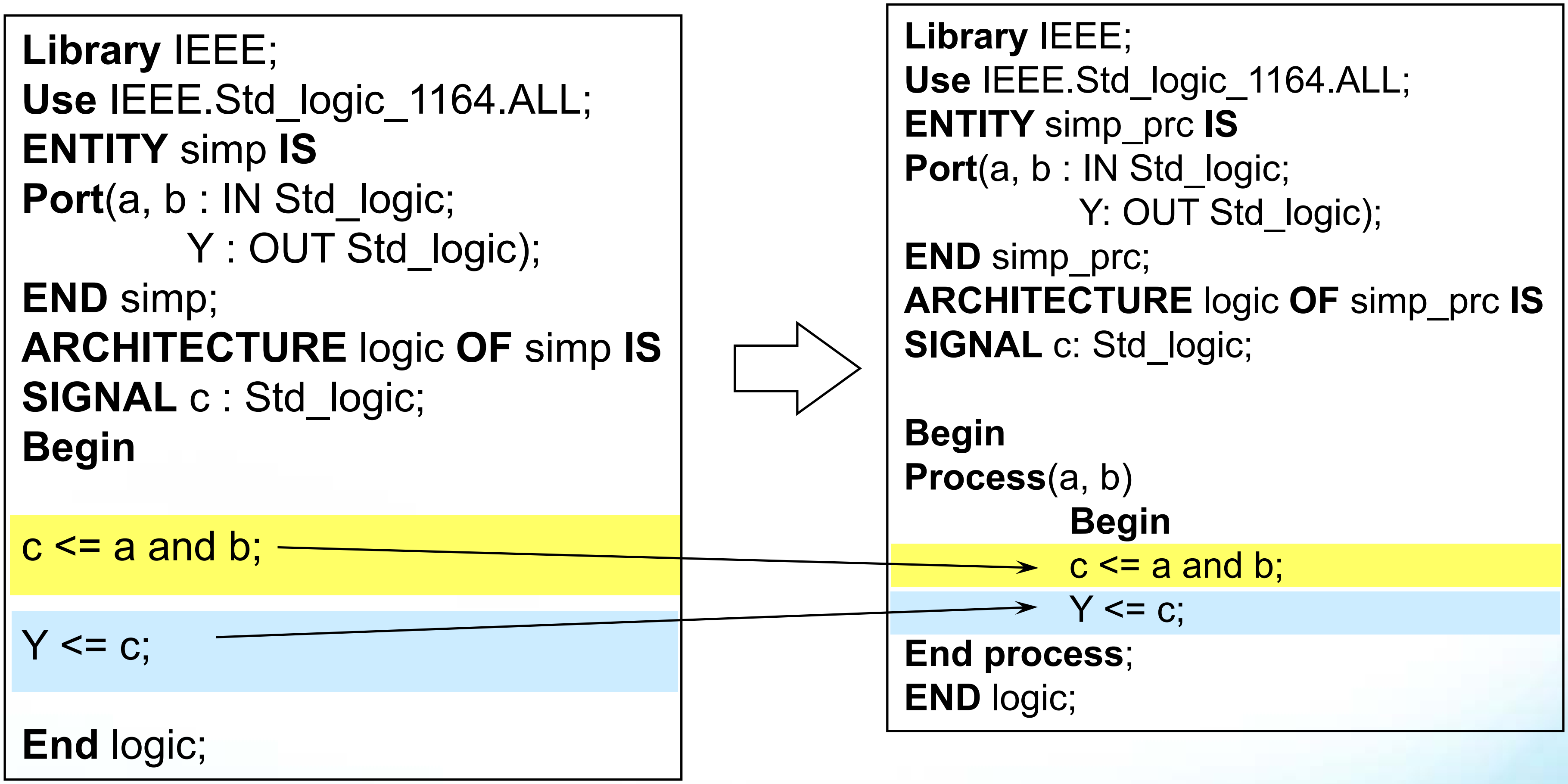


```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp IS
Port(a, b : IN Std_logic;
              Y : OUT Std_logic);
END simp;
ARCHITECTURE logic OF simp IS
SIGNAL c : Std_logic;

Begin

c <= a and b;
Y <= c;

End logic;
```

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp_prc IS
Port(a,b : IN Std_logic;
              Y : OUT Std_logic);
END simp_prc;
ARCHITECTURE logic OF simp_prc IS
SIGNAL c : Std_logic;

Begin
Process1: process(a, b)
              Begin
                            c <= a and b;
              END PROCESS process1;
Process2: process(c)
              Begin
                            Y <= c;
              END PROCESS process2;
END logic;
```

- **c** and **y** get executed and updated in parallel at the end of the process within one simulation cycle

Equivalent Functions?



```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp IS
Port(a, b : IN Std_logic;
          Y : OUT Std_logic);
END simp;
ARCHITECTURE logic OF simp IS
SIGNAL c : Std_logic;
Begin

c <= a and b;

Y <= c;

End logic;
```

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp_prc IS
Port(a, b : IN Std_logic;
           Y: OUT Std_logic);
END simp_prc;
ARCHITECTURE logic OF simp_prc IS
SIGNAL c: Std_logic;

Begin
Process(a, b)
          Begin
          c <= a and b;
          Y <= c;
End process;
END logic;
```

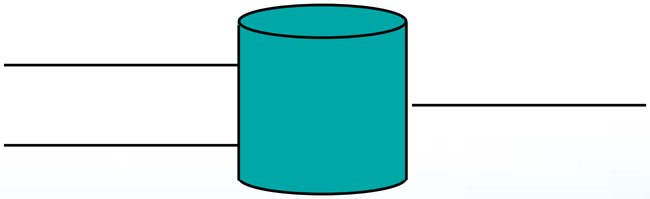
Variable Declarations

- Variables are declared inside a process
- Variables are represented by: **:=**
- Variable declaration

  VARIABLE <*name*> **:** <*DATA_TYPE*> **:=** <*value*>**;**

  Variable temp **: Std_logic_vector (**7 **downto** 0**);**
- Variable assignments are updated immediately
  - Do not incur a delay

Temporary storage



← No Delay →

Assigning Values to Variables

```
VARIABLE   temp  :  Std_logic_VECTOR (7 downto 0);
```

- All bits:

```
        Temp  :=  “10101010”;
        temp := x”aa” ; (1076-1993)
```

- Single bit:

```
        Temp(7)  :=  ‘1’;
```

- Bit-slicing:

```
        Temp (7 downto 4)  :=  “1010”;
```

- Single-bit: single-quote (‘)
- Multi-bit: double-quote (“)

Variable Assignment

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity var is
PORT      (a, b : IN   Std_logic;
           Y : OUT  Std_logic);
END var;

Architecture logic of var is
Begin

          PROCESS (a, b)
          VARIABLE   c  :  Std_logic;
          BEGIN
          c := a AND b;

          Y <= c;

          end process;
END logic;
```

Variable declaration

Variable assignment

***Variable** is assigned to a **Signal** to synthesize to a Piece of hardware*

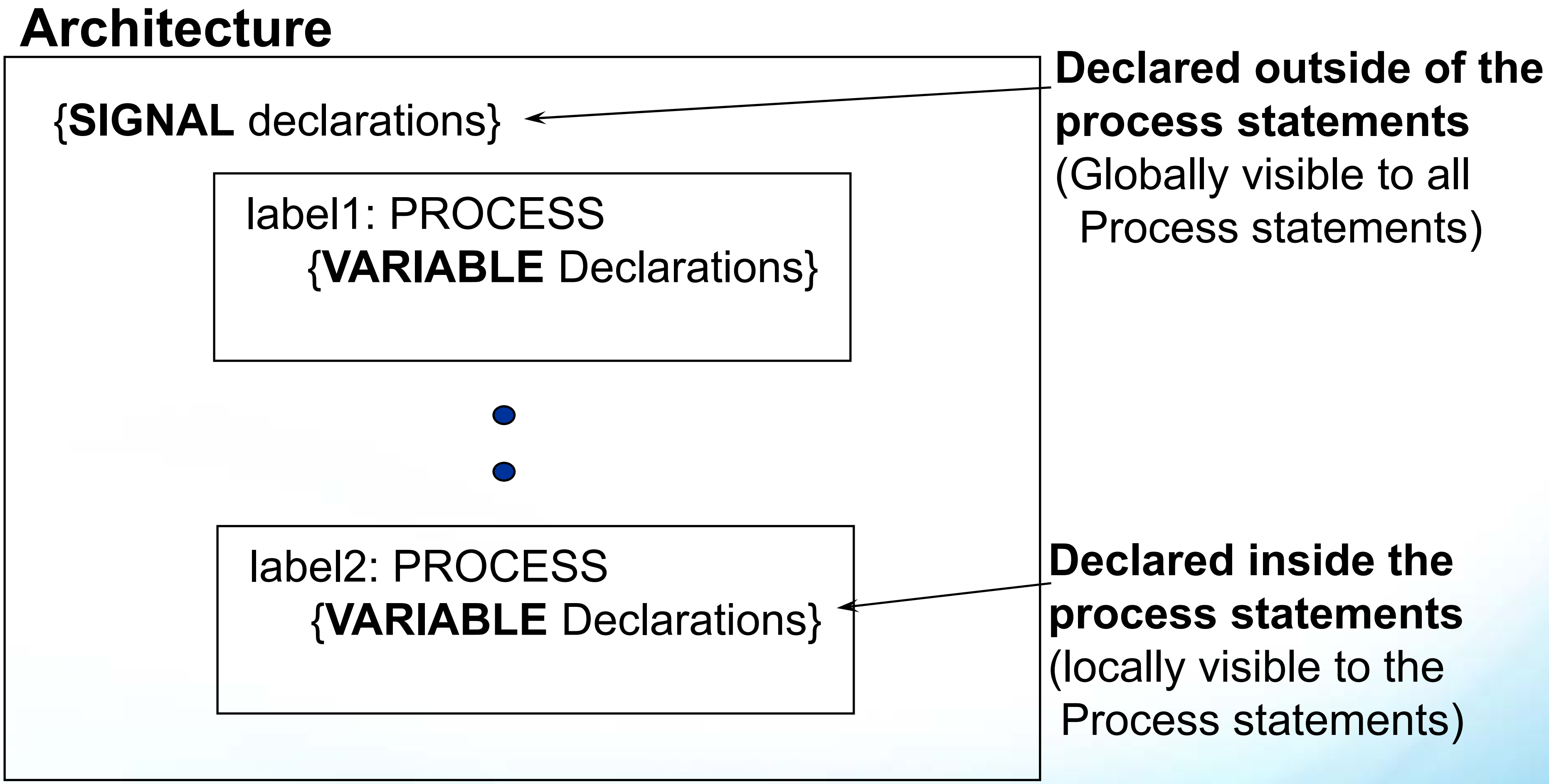
Use of a Variable

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY cmb_var IS
Port(i0, i1, a : IN BIT;
           Q : OUT BIT);
END cmb_var;
ARCHITECTURE logic OF cmb_var IS
Begin
        Process(i0, i1, a)
        VARIABLE val : INTEGER RANGE 0 TO 1;
        Begin
                Val := 0;
                IF (a = '0') THEN
                        val := val;
                Else
                        val := val + 1;
                End if;
                CASE val IS
                        WHEN 0 =>
                                q <= i0;

                        WHEN 1 =>
                                q <= i1;
                End case;
        End process;
END logic;
```

Val *is a variable that is updated at the instant an assignment is made to it*

Therefore, the updated value of **val** *is available for the CASE statement*

Signal and Variable Scope



Architecture

{**SIGNAL** declarations}

label1: PROCESS
{**VARIABLE** Declarations}

label2: PROCESS
{**VARIABLE** Declarations}

Declared outside of the process statements
(Globally visible to all Process statements)

Declared inside the process statements
(locally visible to the Process statements)

Review - Signals vs. Variables

	Signals (<=)	Variables (:=)
Assign	Assignee **<=** assignment	Assignee **:=** assignment
Utility	Represent circuit interconnect	Represent local storage
Scope	Global scope (communicate between PROCESSES)	Local scope (inside process)
Behavior	Signals updated at end of current delta cycle (new value not available)	Updated immediately (new value available)

Sequential Statements

- Sequential statements
  - IF-THEN statement
  - CASE statement
  - Looping statements

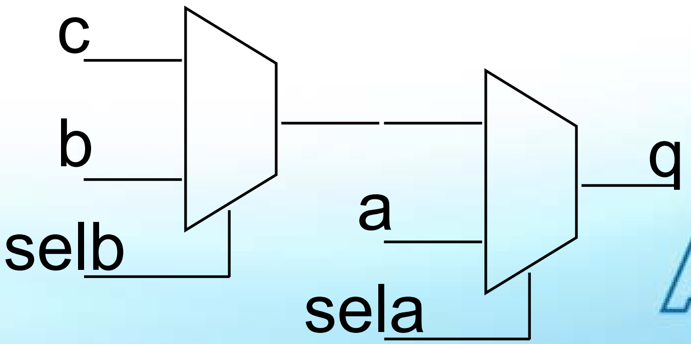
If-then Statements

- Format:

```
IF <condition1> THEN
                {Sequence of statement(s)}
ELSIF <condition2> THEN
                {Sequence of statement(s)}
                        .
                        .
Else
                {Sequence of statement(s)}
End if;
```

- Example:

```
Process (sela, selb, a, b, c)
Begin
      IF sela='1' THEN
            q <= a;
      ELSIF selb='1' THEN
            q <= b;
      Else
            q <= c;
       End if;
End process;
```



If-then Statements

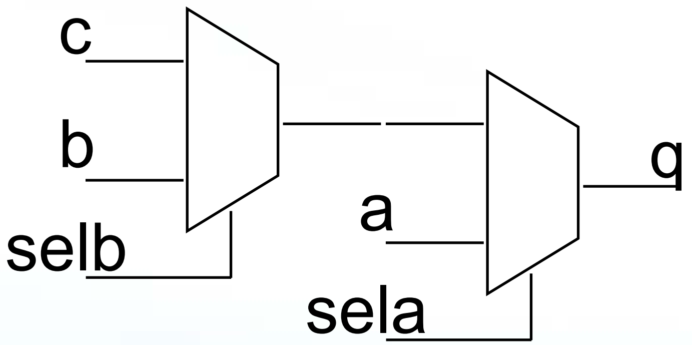
- Conditions are evaluated in order from top to bottom
  - Prioritization
- The first condition that is true causes the corresponding sequence of statements to be executed
- If all conditions are false, then the sequence of statements associated with the “ELSE” clause is evaluated

If-then Statements

- Similar to conditional signal assignment

Implied process

```
q <= a WHEN sela = ‘1’ ELSE
     b WHEN selb = ‘1’ ELSE
     c;
```

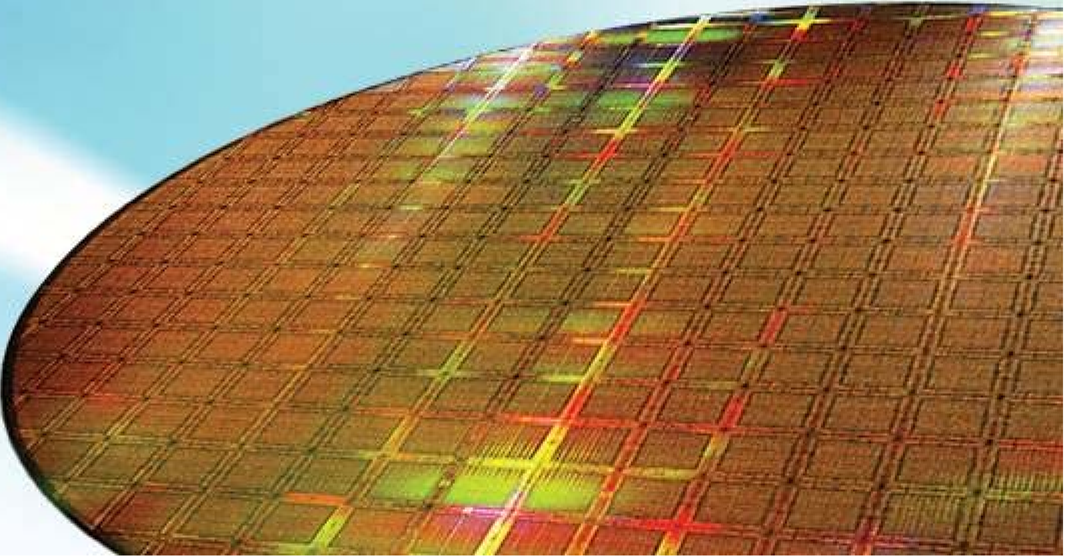


explicit process

```
PROCESS(sela, selb, a, b, c)
BEGIN
    IF sela=‘1’ THEN
          q <= a;
    ELSIF selb=‘1’ THEN
          q <= b;
    ELSE
          q <= c;
     END IF;
END PROCESS;
```

Exercise 2

Please Go to Exercise 2



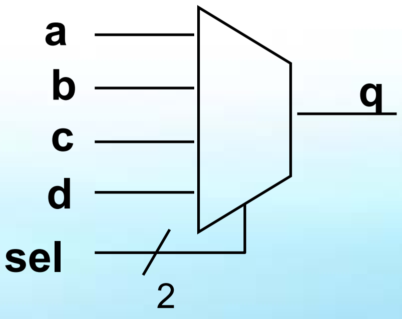
Case Statement

- Format:

```
CASE {expression} IS
          WHEN <condition1> =>
                    {sequence of statements}
          WHEN <condition2> =>
                    {sequence of statements}
                              .
                              .
          When others =>        -- (optional)
                    {Sequence of statements}
          End case;
```

- Example:

```
Process (sel, a, b, c, d)
Begin
    CASE sel IS
          When "00" =>
                    q <= a;
          When "01" =>
                    q <= b;
          When "10" =>
                    q <= c;
          When others =>
                    q <= d;
    End case;
End process;
```



Case Statement

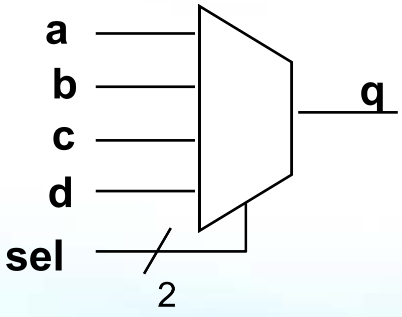
- Conditions are evaluated at once
  - No prioritization
- **All** possible conditions must be considered
- **WHEN OTHERS** clause evaluates all other possible conditions that are not specifically stated

Case Statement

- Similar to selected signal assignment

Implied process

```
WITH sel SELECT
        q <= a WHEN “00”,
            b WHEN “01”,
            c WHEN “10”,
            d WHEN OTHERS;
```

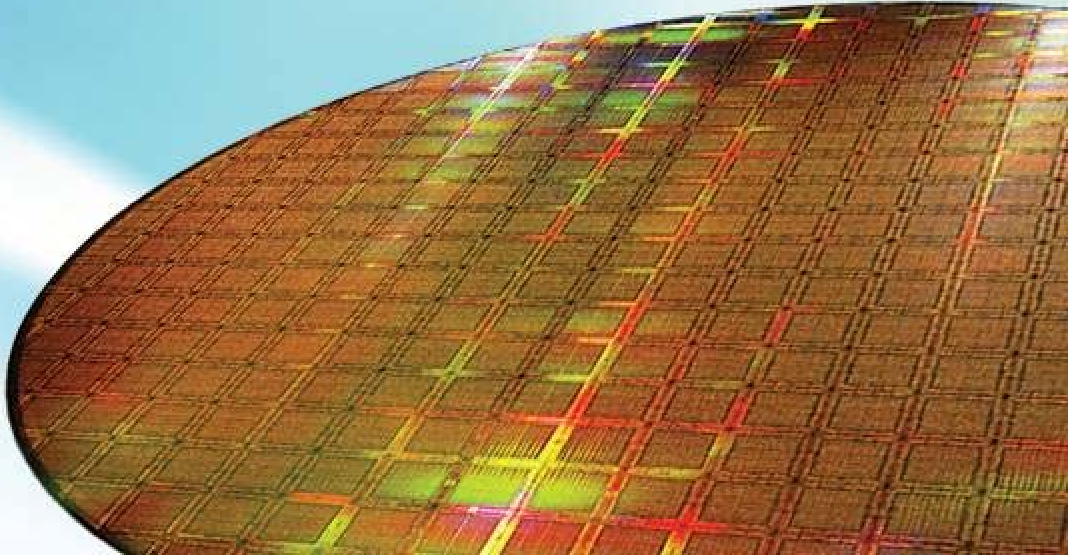


explicit process

```
PROCESS(sel, a, b, c, d)
BEGIN
   CASE sel IS
         WHEN “00” =>
               q <= a;
         WHEN “01” =>
               q <= b;
         WHEN “10” =>
               q <= c;
         WHEN OTHERS =>
               q <= d;
    END CASE;
END PROCESS;
```

Exercise 3

Please Go to Exercise 3



Sequential LOOPS

- Infinite loop
  - Loops infinitely unless EXIT statement exists

```
[Loop_label]loop
  --Sequential statement
 EXIT loop_label ;
END LOOP;
```

- While loop
  - Conditional test to end loop

```
WHILE <condition> LOOP
   --Sequential statements
End loop;
```

- For loop
  - Iteration loop

```
For <identifier> in <range> loop
  --Sequential statements
End loop;
```

FOR LOOP Using a Variable: 4-bit Left Shifter

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
ENTITY shift4 IS
PORT (  shft_lft : in Std_logic;
                    D_in : in Std_logic_vector(3 downto 0);
                    Q_out : out Std_logic_vector(7 downto 0));
END shift4;
ARCHITECTURE logic OF shift4 IS
Begin

Process(d_in, shft_lft)
          VARIABLE shft_var : Std_logic_vector(7 DOWNTO 0);
Begin
          Shft_var(7 downto 4) := "0000";
          Shft_var(3 downto 0) := d_in;
```

Variable declaration

Variable is initialized

FOR LOOP Using a Variable: 4-bit Left Shifter

```
IF shft_lft = '1' THEN
            FOR i IN 7 DOWNTO 4 LOOP
                        Shft_var(i) := shft_var(i-4);
            End loop;
                        Shft_var(3 downto 0) := "0000";
Else
            shft_var := shft_var;
End if;

Q_out <= shft_var;

End process;
END logic;
```

Enables shift-left

I is the index for the FOR LOOP
And does not need to be declared

Shifts left by 4

Fills the LSBs with zeros

No shifting

Variable is assigned to a **signal** before the end of the process to synthesize to a piece of hardware

FOR LOOP: ‘1’s Counter

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
Use IEEE.Std_logic_arith.All;

Entity bc is port (invec: in Std_logic_vector(31 downto 0);
                Outvec: out Std_logic_vector(7 downto 0));
END bc;
ARCHITECTURE rtl OF bc IS
Begin
Process(invec)
                VARIABLE count: Std_logic_vector(7 downto 0);
```

Variable declaration

FOR LOOP: ‘1’s Counter

```
Begin
    Count:=(others=>’0’);
    FOR i IN invec’right TO invec’left LOOP
        IF (invec(i)/=’0’) THEN
            Count:=count+1;
        End if;
    End loop;
Outvec<=count;
End process;
END rtl;
```

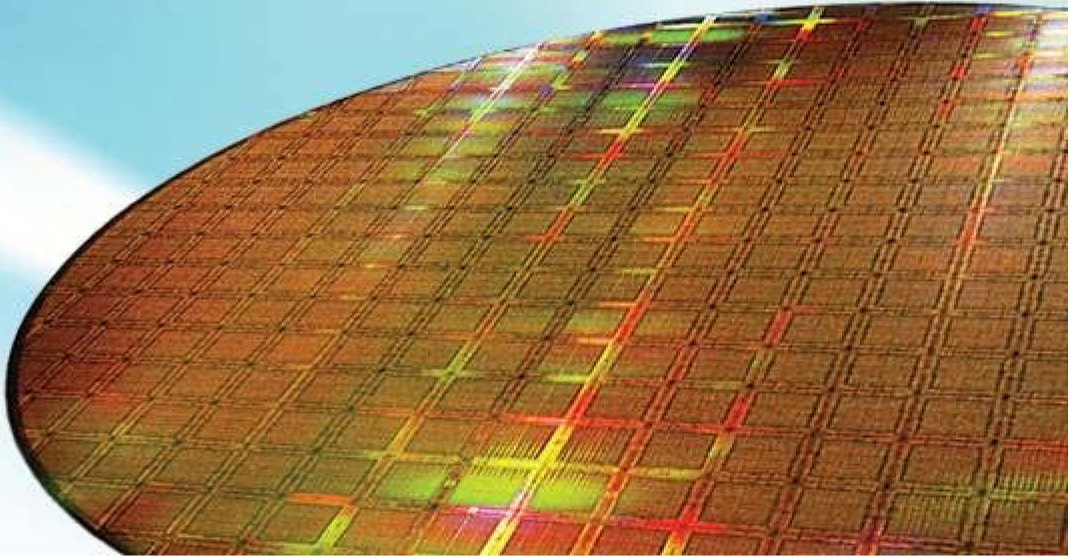
Variable is initialized

I is the loop index
This loop examines all 32 bits of invec. If the current bit does not equal zero, count is incremented.

Variable is assigned to a **signal** before the end of the process to synthesize to a piece of hardware

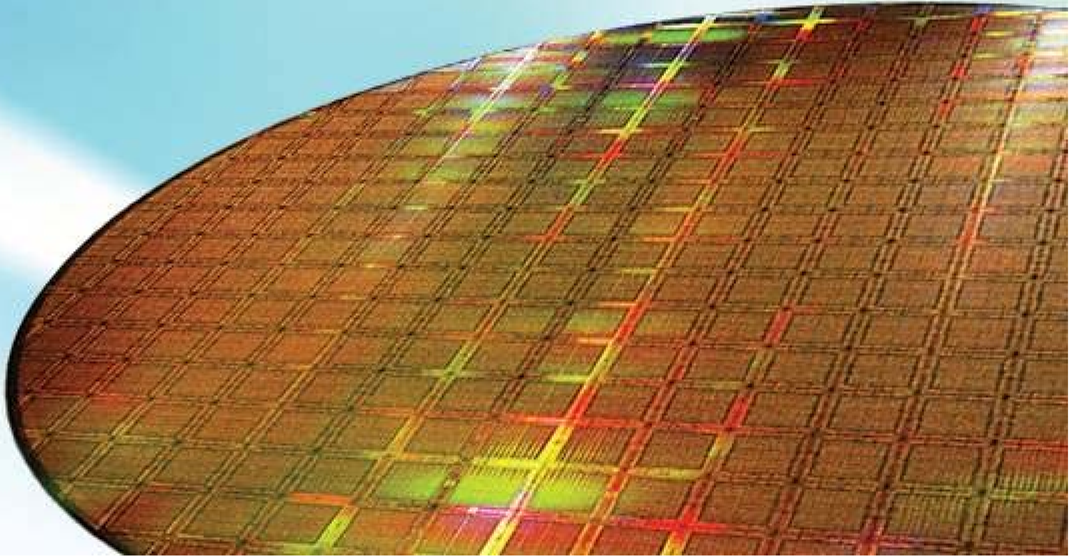
Exercise 4

Please Go to Exercise 4



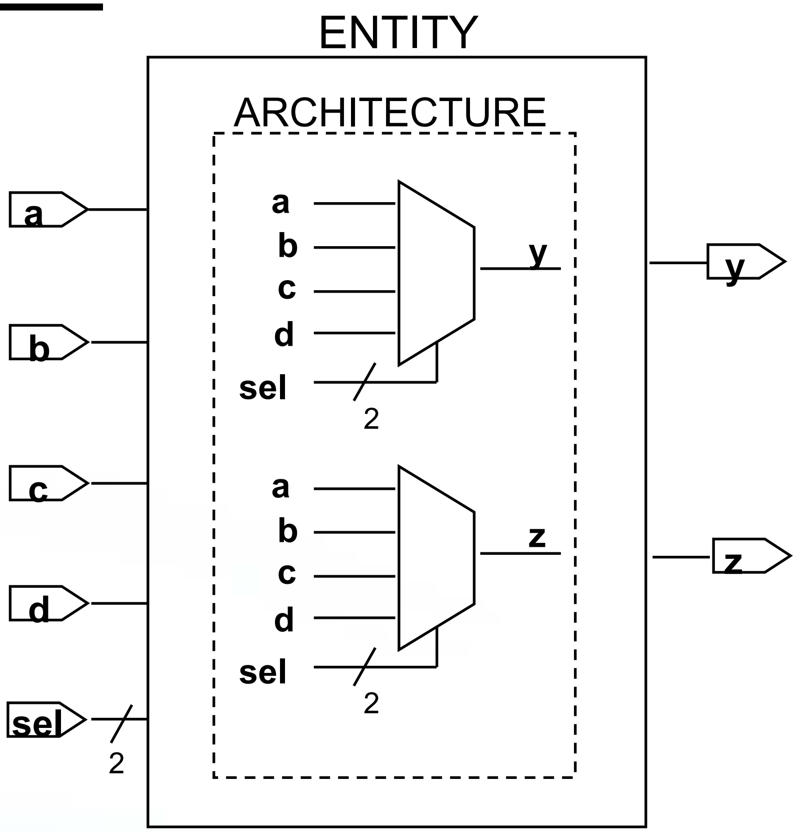
Introduction to VHDL

VHDL and Logic Synthesis



VHDL Model - RTL Modeling

Result:

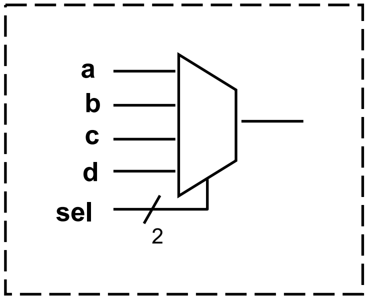


- **RTL** - type of behavioral modeling that implies or infers hardware
- Functionality and somewhat structure of the circuit
- For the purpose of synthesis, as well as simulation

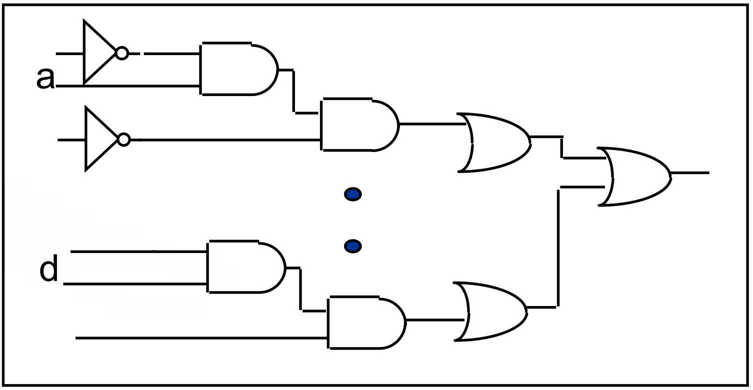
Recall - RTL Synthesis

```
IF sel="00" THEN
            Mux_out <= a;
ELSIF sel="01" THEN
            Mux_out <= b;
 ............
 Else sel="11" then
            Mux_out <= d;
```

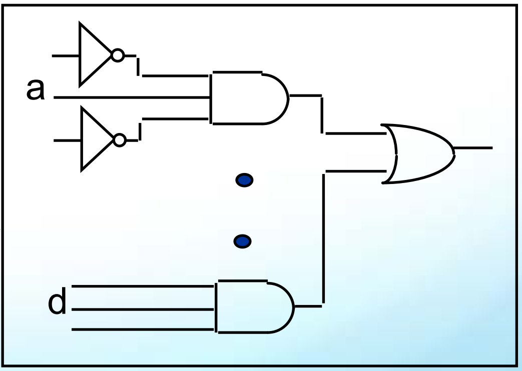
Inferred



Translation



Optimization

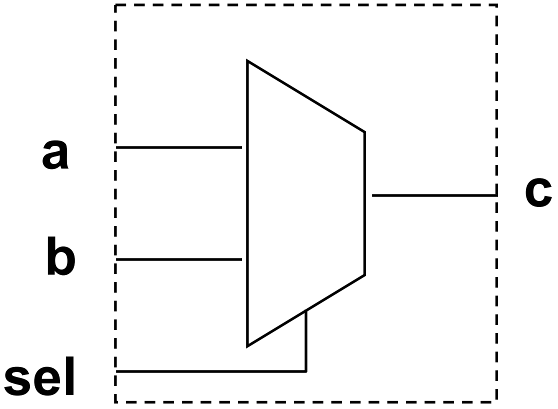


Two Types of Process Statements

- **Combinatorial process**
  - Sensitive to all inputs used in The combinatorial logic
- **Example**

Process(a, b, sel)

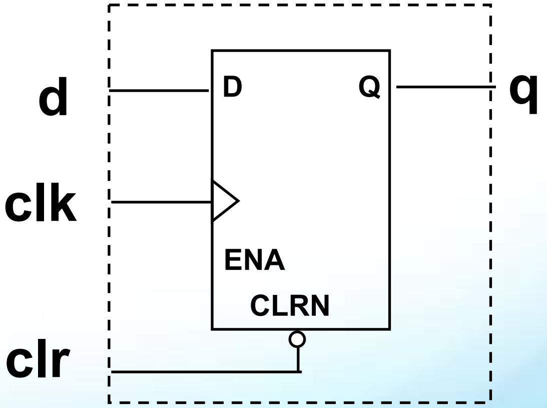
Sensitivity list includes all inputs used In the combinatorial logic



- **Sequential process**
  - Sensitive to a clock or/and control signals
- **Example**

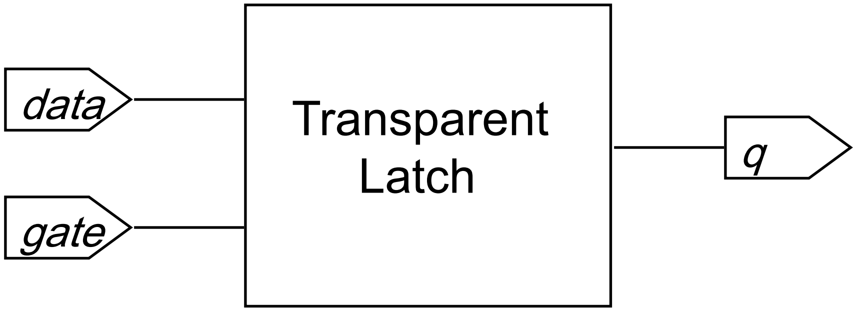
Process(clr, clk)

Sensitivity list does not include the **d** *input, only the clock or/and control signals*



Latch

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity latch1 is
PORT ( data : IN Std_logic;
                        Gate : IN
Std_logic;
                        Q : OUT Std_logic
            );
End latch1;

Architecture behavior of latch1 is
Begin

Label_1: process (data, gate)
            Begin
            IF gate = '1' THEN
                        Q <= data;
            End if;
End process;

End behavior;
```



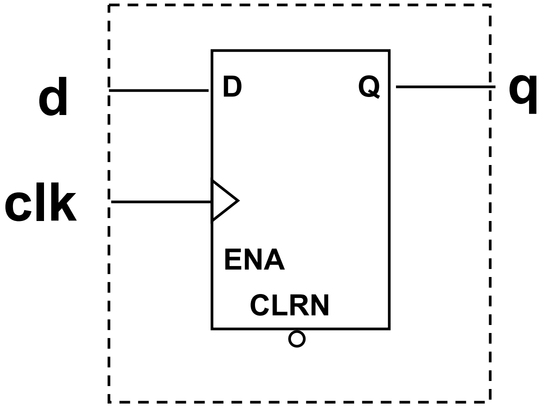
Sensitivity list includes both inputs

What happens if gate = ‘0’?

⇨ ***Implicit memory***

DFF with WAIT Statement

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity wait_dff is
PORT ( d, clk : in Std_logic;
                    Q : out Std_logic
          );
End wait_dff;

Architecture behavior of wait_dff is
Begin
Process
          begin
          Wait until clk = '1';
                    Q <= d;
End process;
END behavior;
```



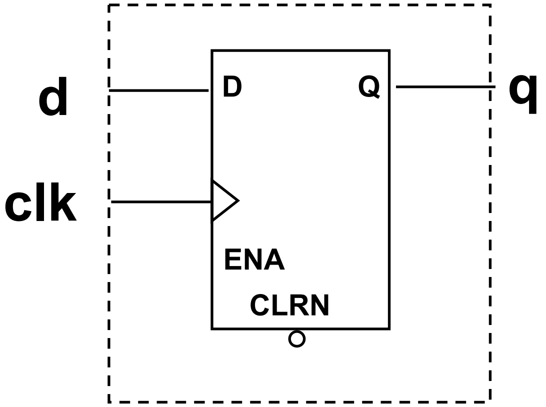
Note: there is no sensitivity list

Wait until

- *Acts like the sensitivity list*

DFF - clk’event and clk=‘1’

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity dff_a is
PORT ( d : in Std_logic;
                    Clk : in Std_logic;
                    Q : out Std_logic
          );
End dff_a;

Architecture behavior of dff_a is
Begin
PROCESS (clk)
          BEGIN
          IF clk'event and clk = '1' THEN
                    Q <= d;
          End if;
End process;
END behavior;
```

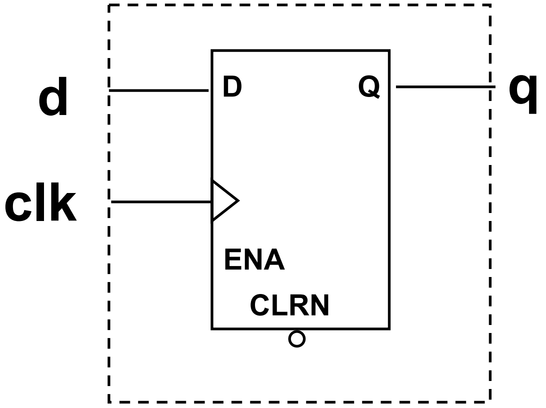


Clk’event **and** ***clk=‘1’***

– ***Clk*** *is the signal name (any name)*
– ***‘Event*** *is a VHDL attribute, specifying that there needs To be a change in signal value*
– ***Clk=‘1’*** *means positive-edge triggered*

DFF - rising_edge

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;

Entity dff_b is
PORT ( d : in Std_logic;
                    Clk : in Std_logic;
                    Q : out Std_logic
          );
End dff_b;

Architecture behavior of dff_b is
Begin
Process(clk)
          Begin
          IF rising_edge(clk) THEN
                    Q <= d;
          End if;
End process;
END behavior;
```

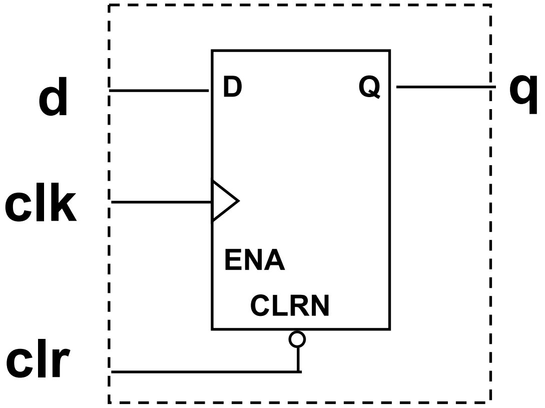


Rising_edge

- *IEEE function that is defined in the Std_logic_1164 package*
- *Specifies that the signal value* ***must*** *be 0 to 1*
- *X, Z to 1 transition is not allowed*

DFF with Asynchronous Clear

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;

Entity dff_clr is
PORT (  clr : in bit;
                D, clk : in Std_logic;
                Q : out Std_logic
                );
End dff_clr;

Architecture behavior of dff_clr is
Begin
Process(clk, clr)
                Begin

                If clr = '0' then
                                Q <= '0';

                Elsif rising_edge(clk) then
                                Q <= d;
                End if;
End process;
END behavior;
```



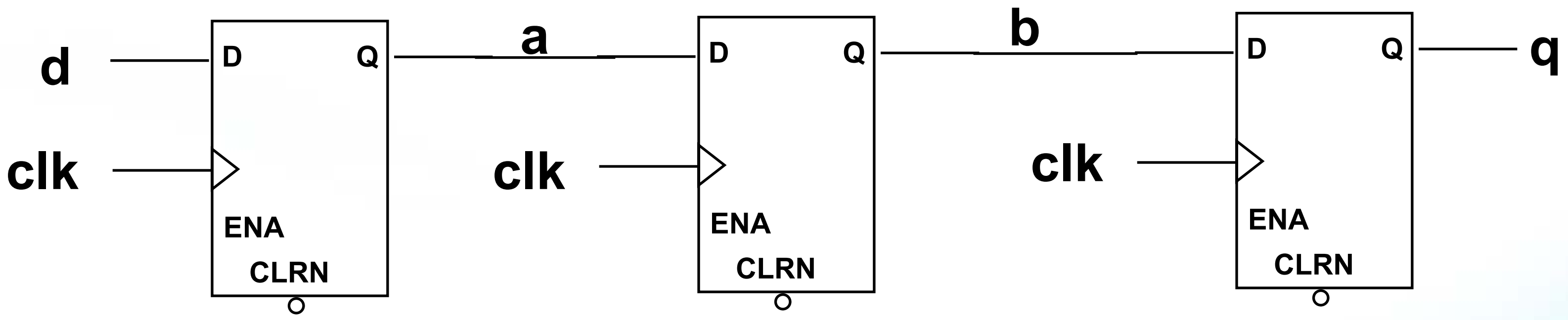
- *This is how to implement asynchronous control signals for the register*
- *Note: this IF-THEN statement is outside the IF-THEN statement that checks the condition **rising_edge***
- *Therefore, **clr='1'** does not depend on the clock*

How Many Registers?

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
ENTITY reg1 IS
        PORT (   d       : in Std_logic;
                 Clk     : in Std_logic;
                 Q       : out Std_logic);
END reg1;

Architecture reg1 of reg1 is
SIGNAL a, b : Std_logic;
Begin
        PROCESS (clk)
        Begin
                IF rising_edge(clk) THEN
                        a <= d;
                        b <= a;
                        Q <= b;
                End if;
        End process;
END reg1;
```

How Many Registers?

- Signal assignments inside the IF-THEN statement that checks the clock condition infer registers



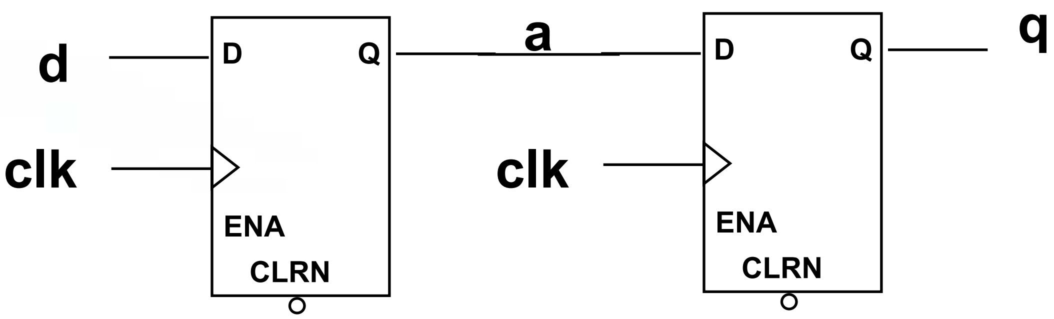
How Many Registers?

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
ENTITY reg1 IS
    PORT ( d        : in Std_logic;
           Clk      : in Std_logic;
           Q        : out Std_logic);
END reg1;
ARCHITECTURE reg1 OF reg1 IS
SIGNAL a, b : Std_logic;
Begin
    PROCESS (clk)
    Begin
        IF rising_edge(clk)  THEN
            a <= d;
            b <= a;
        End if;
    End process;
    Q <= b;
END reg1;
```

Signal Assignment Moved

How Many Registers?

- b to Q assignment is no longer edge-sensitive because it is not inside the if-then statement that checks the clock condition



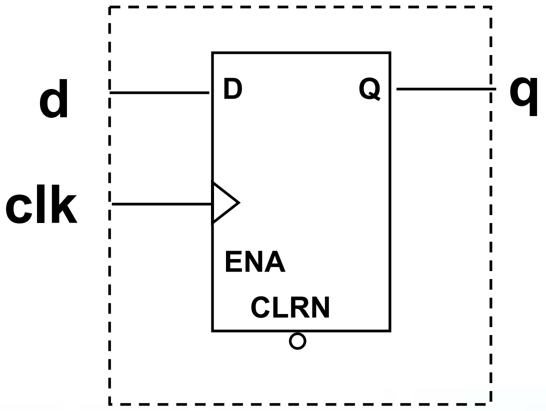
How Many Registers?

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
ENTITY reg1 IS
        PORT (    d       : in Std_logic;
                  Clk     : in Std_logic;
                  Q       : out Std_logic);
END reg1;

Architecture reg1 of reg1 is
Begin
        PROCESS (clk)
        VARIABLE a, b : Std_logic;
        Begin
                IF rising_edge(clk) THEN
                        a := d;
                        b := a;
                        Q <= b;
                End if;
        End process;
END reg1;
```

Signals changed to variables

How Many Registers?

- Variable assignments are updated immediately
- Signal assignments are updated on clock edge



Variable Assignments in Sequential Logic

- Variable assignments inside the IF-THEN statement, that checks the clock condition, usually don't infer registers
  - Exception: if the variable is on the right side of the equation in a clocked process prior to being assigned a value, the variable will infer a register(s)
- Variable assignments are temporary storage and have no hardware intent
- Variable assignments can be used in expressions to immediately update a value
  - Then the variable can be assigned to a signal

Example - Counter Using a Variable

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
ENTITY count_a IS
PORT (clk, rst, updn : in Std_logic;
        Q : out Std_logic_vector(15 downto 0));
END count_a;
ARCHITECTURE logic OF count_a IS
BEGIN
Process(rst, clk)
VARIABLE tmp_q : Std_logic_vector(15 downto 0);
Begin
        IF rst = '0' THEN
                Tmp_q := (others => '0');
        ELSIF rising_edge(clk) THEN
                IF updn = '1' THEN
                        Tmp_q := tmp_q + 1;
                Else
                        Tmp_q := tmp_q - 1;
                End if;
        End if;
        Q <= tmp_q;
End process;
END logic;
```

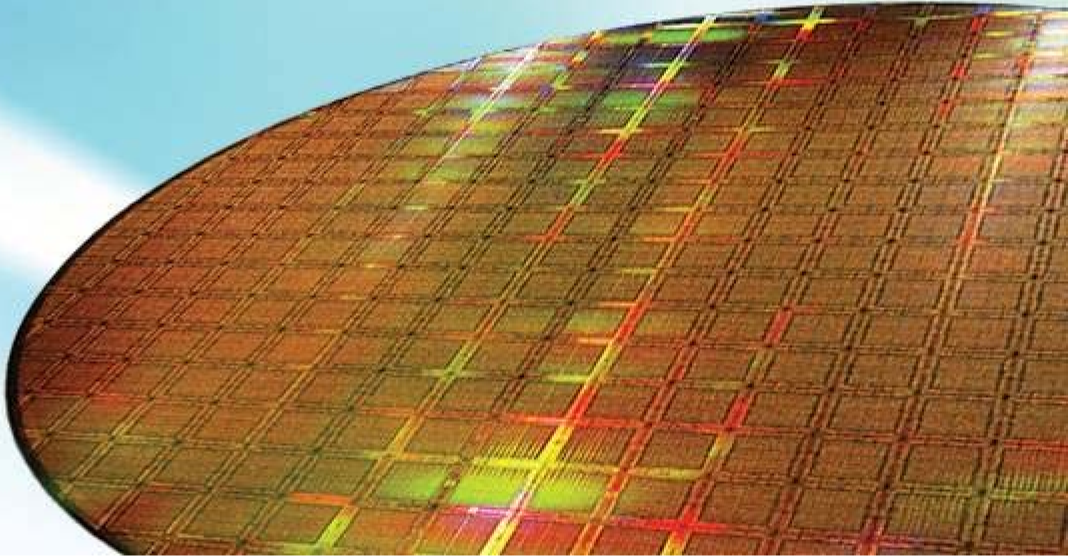
- Counters are accumulators that always add a ‘1’ or subtract a ‘1’
- This example takes 17 LE

Arithmetic expression assigned to a Variable

Variable assigned to a signal

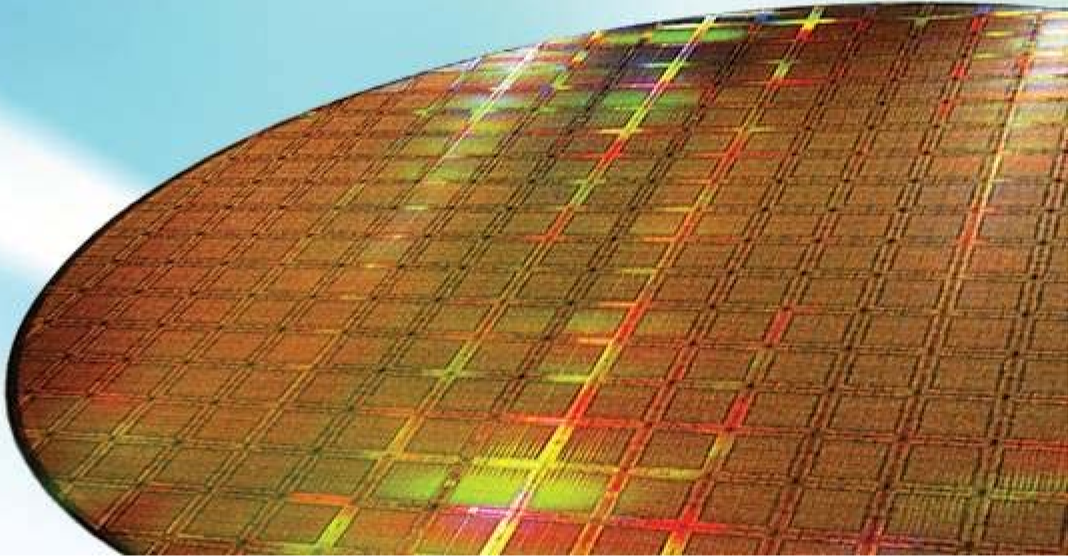
Exercise 5

Please Go to Exercise 5



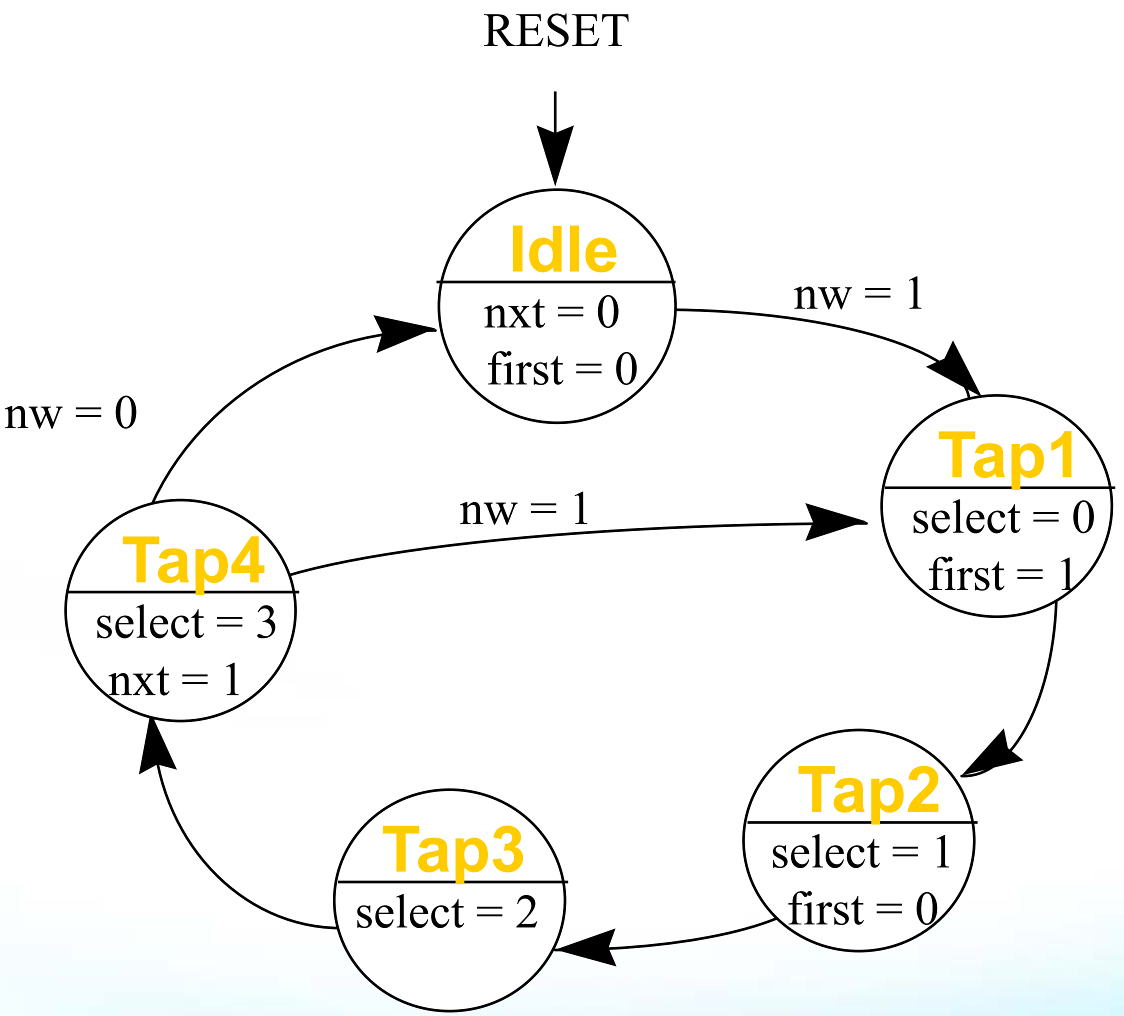
Introduction to VHDL

Model Application



Finite State Machine (Fsm) - State Diagram

Inputs:
reset
nw



Outputs:
select
first
nxt

Enumerated Data Type

- Recall the built-in data types:
  - bit
  - Std_logic
  - integer
- What about user-defined data types?:
  - Enumerated data type:

```
TYPE <your_data_type> IS
      (items or values for your data type separated by commas)
```

Writing VHDL Code for FSM

- State machine states must be an enumerated data type:

```
TYPE state_type IS (idle, tap1, tap2, tap3, tap4 );
```

- Object which stores the value of the current state must be a ***signal*** of the user-defined type:

```
SIGNAL filter :  state_type;
```

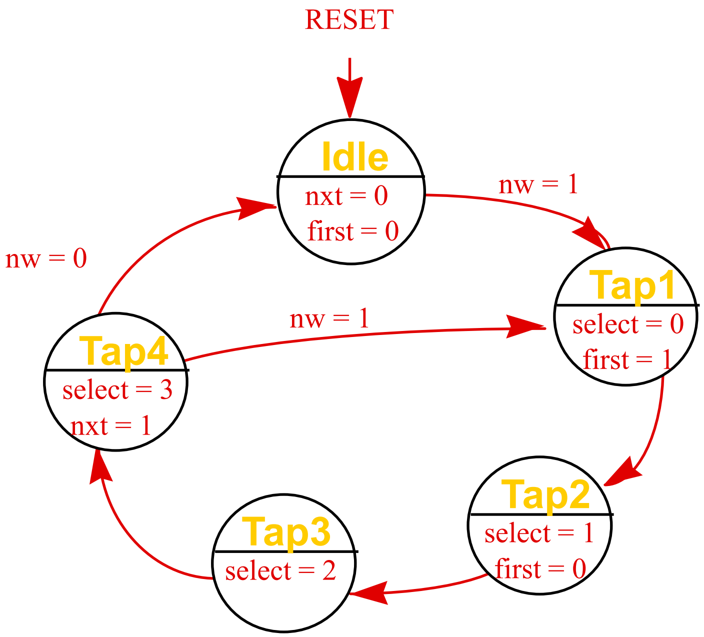
Writing VHDL Code for FSM

- To determine next state transition/logic:
  - Use a **CASE** statement inside IF-THEN statement that checks for the clock condition
    - Remember: state machines are implemented using registers
- To determine state machine outputs:
  - Use **conditional** and/or **selected** signal assignments
  - Or use a second **case** statement to determine the state machine outputs

FSM VHDL Code - Enumerated Data Type

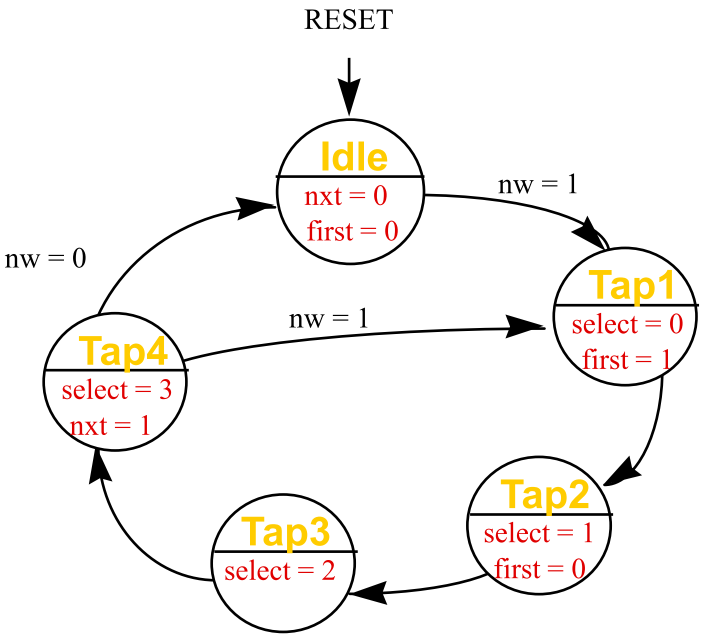
```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;
Use IEEE.Std_logic_arith.All;

Entity state_m2 is
Port(clk, reset, nw : in Std_logic;
        Sel: out Std_logic_vector(1 downto 0);
        Nxt, first: out Std_logic);
END state_m2;

Architecture logic of state_m2 is
        TYPE state_type IS
                (idle, tap1, tap2, tap3, tap4);
        SIGNAL filter : state_type;
```

Enumerated data type



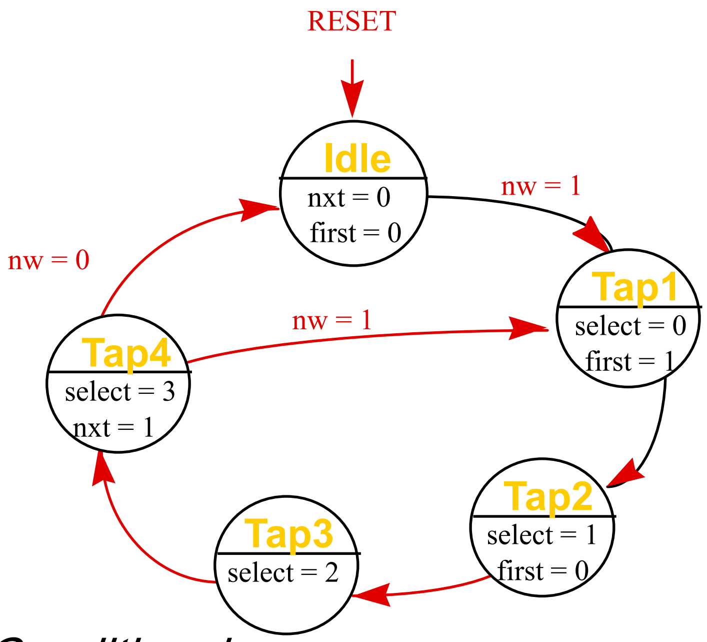
FSM VHDL Code - Next State Logic

```
Begin
PROCESS (reset, clk)
        Begin
        IF reset = '1' THEN
                Filter <= idle;
        ELSIF clk'event and clk = '1' THEN
                CASE filter IS
                WHEN idle =>
                        IF nw = '1' THEN
                                Filter <= tap1;
                        End if;
                WHEN tap1 =>
                        filter <= tap2;
                WHEN tap2 =>
                        filter <= tap3;
                WHEN tap3 =>
                        filter <= tap4;
                WHEN tap4 =>
                        IF nw = '1' THEN
                                Filter <= tap1;
                        Else
                                Filter <= idle;
                        End if;

                End case;
        End if;
End process;
```



FSM VHDL Code - Outputs

```
Nxt <= '1' WHEN filter=tap4 ELSE
               '0';

First <= '1'  when filter=tap1 else
                              '0';

With filter select
            Sel <= "00" WHEN tap1,
                          "01" WHEN tap2,
                          "10" WHEN tap3,
                          "11" WHEN tap4,
                          "00" WHEN others;

End logic;
```

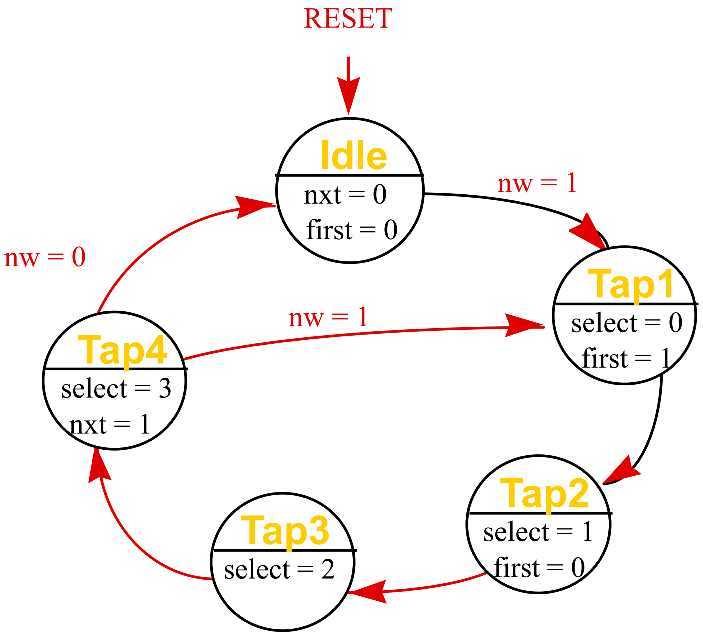


Conditional signal assignments

Selected signal assignments

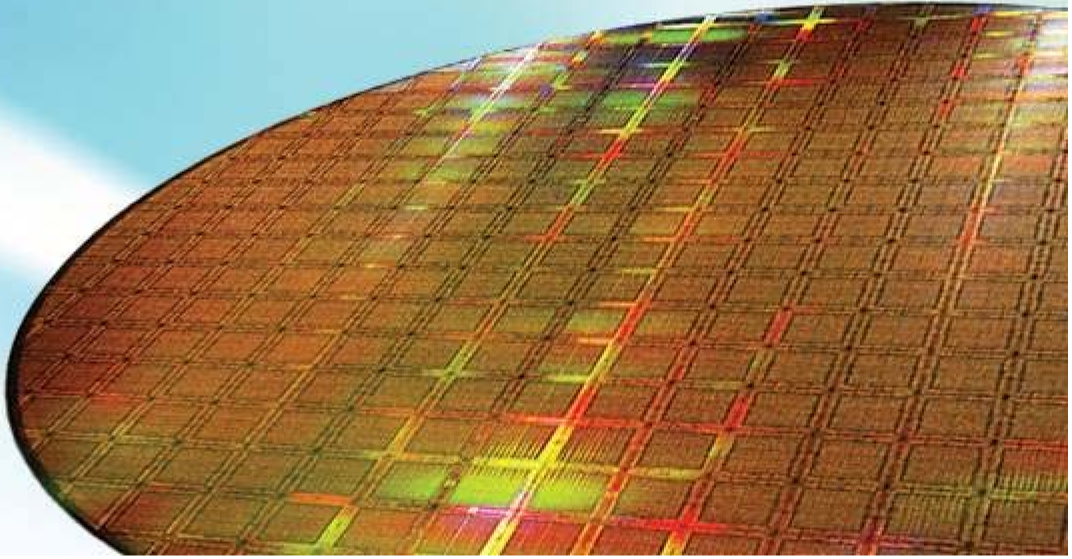
FSM VHDL Code - Outputs Using a Case

```
Output: process(filter)
          Begin
          case  filter is
                    WHEN idle =>
                              nxt <= '0';
                              First <= '0';
                    WHEN tap1 =>
                              sel <= "00";
                              First <= '1';
                    WHEN tap2 =>
                              sel <= "01";
                              First <= '0';
                    WHEN tap3 =>
                              sel <= "10";
                    WHEN tap4 =>
                              sel <= "11";
                              Nxt <= '1';
                    End case;
End process output;

End logic;
```



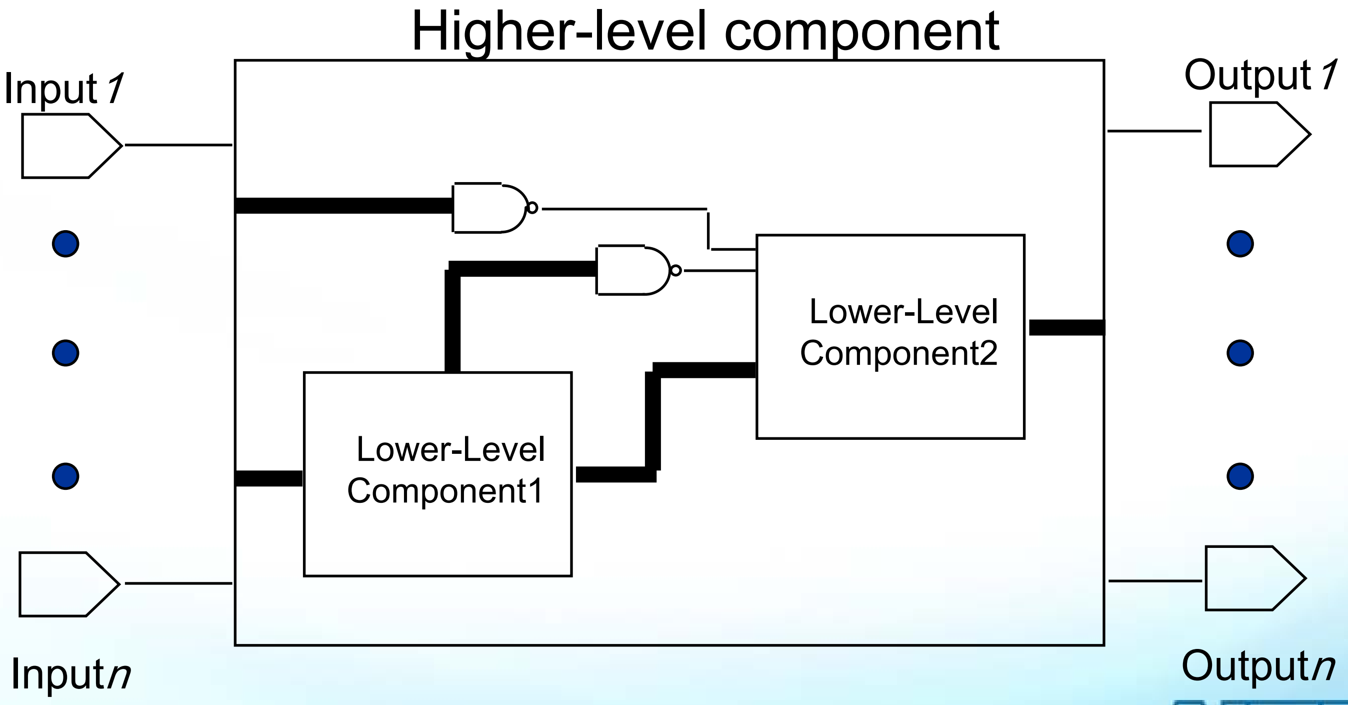
Introduction to VHDL

Designing hierarchy



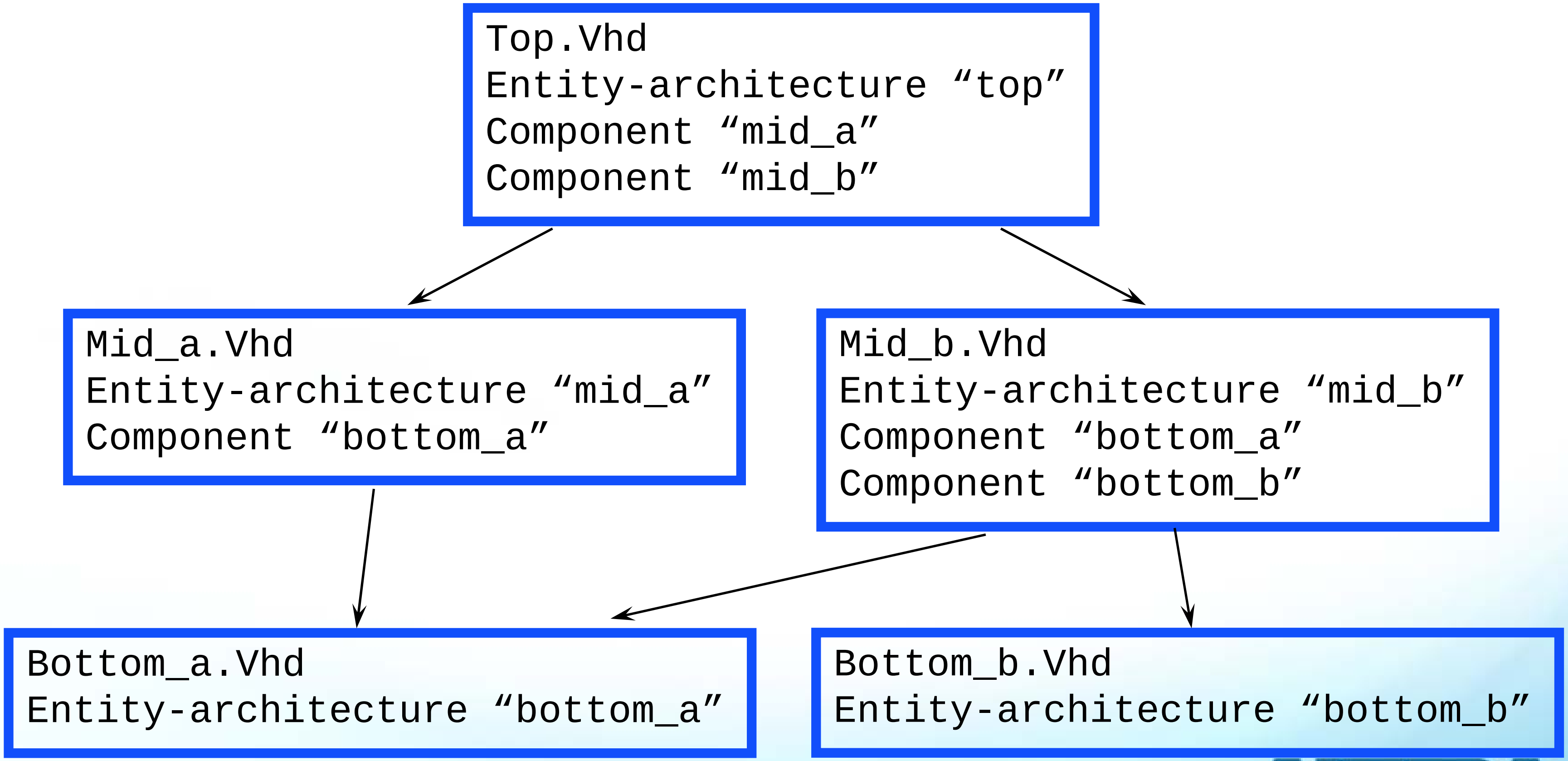
Recall - Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware, lower-level components
- For the purpose of synthesis



Design Hierarchically - Multiple Design Files

- VHDL hierarchical design requires component declarations and component instantiations


Top.Vhd
Entity-architecture “top”
Component “mid_a”
Component “mid_b”

Mid_a.Vhd
Entity-architecture “mid_a”
Component “bottom_a”

Mid_b.Vhd
Entity-architecture “mid_b”
Component “bottom_a”
Component “bottom_b”

Bottom_a.Vhd
Entity-architecture “bottom_a”

Bottom_b.Vhd
Entity-architecture “bottom_b”

Component Declaration and Instantiation

- Component declaration - used to declare the *port types* and the *data types* of the ports for a lower-level design

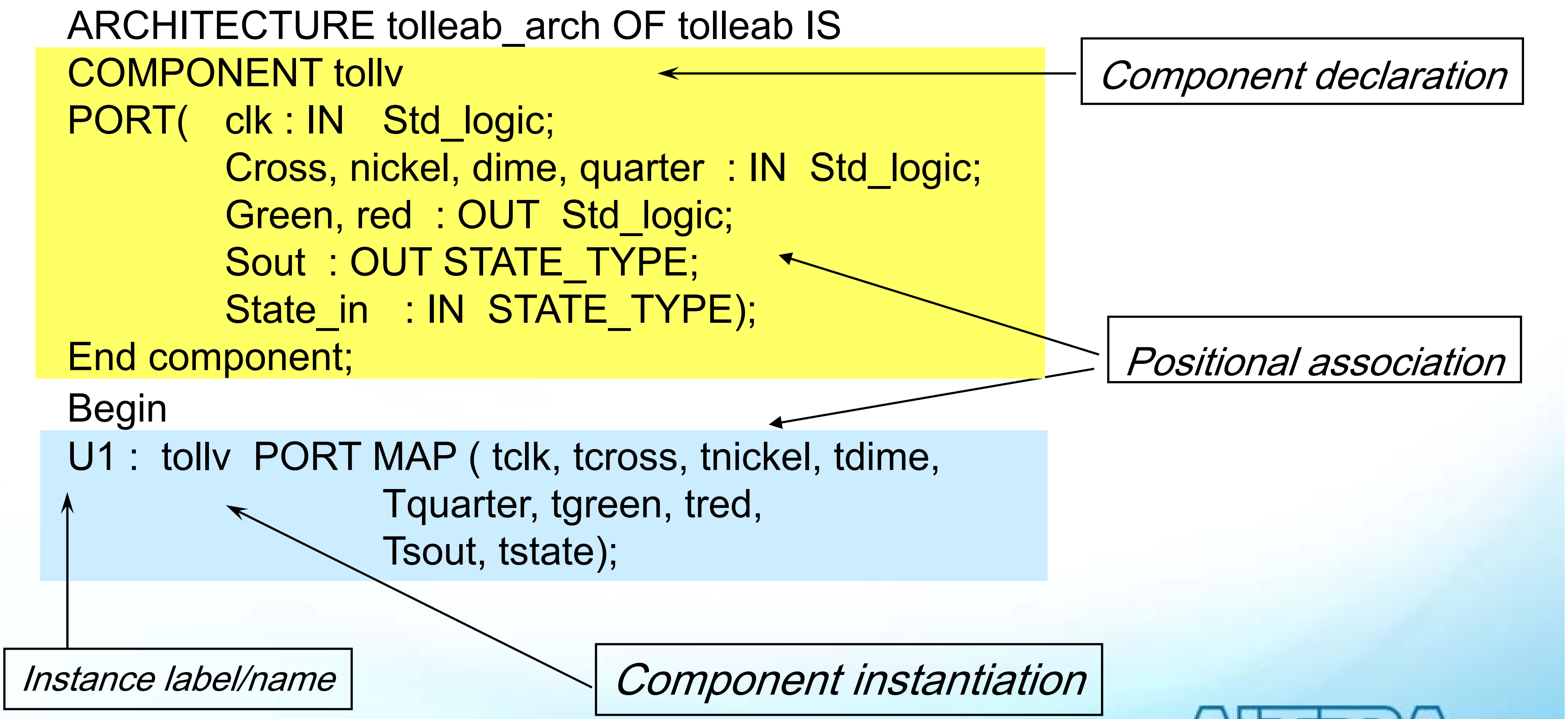
```
COMPONENT <lower-level_design_name> IS
PORT  ( <port_name> :  <port_type> <data_type>;
                              .
                              .
          <Port_name> :  <port_type> <data_type>);
End component;
```

- Component instantiation - used to map the ports of a lower-level design to that of the current-level design

```
<Instance_name> : <lower-level_design_name>
PORT map(<lower-level_port_name>  =>  <current_level_port_name>,
     …,<lower-level_port_name> => <current_level_port_name>);
```

Component Declaration and Instantiation

- Upper-level of hierarchy design must have a component declaration for a lower-level design before it can be instantiated



```
ARCHITECTURE tolleab_arch OF tolleab IS
COMPONENT tollv
PORT(   clk : IN   Std_logic;
        Cross, nickel, dime, quarter  : IN  Std_logic;
        Green, red  : OUT  Std_logic;
        Sout  : OUT STATE_TYPE;
        State_in   : IN  STATE_TYPE);
End component;
Begin
U1 :  tollv  PORT MAP ( tclk, tcross, tnickel, tdime,
                 Tquarter, tgreen, tred,
                 Tsout, tstate);
```

Component declaration

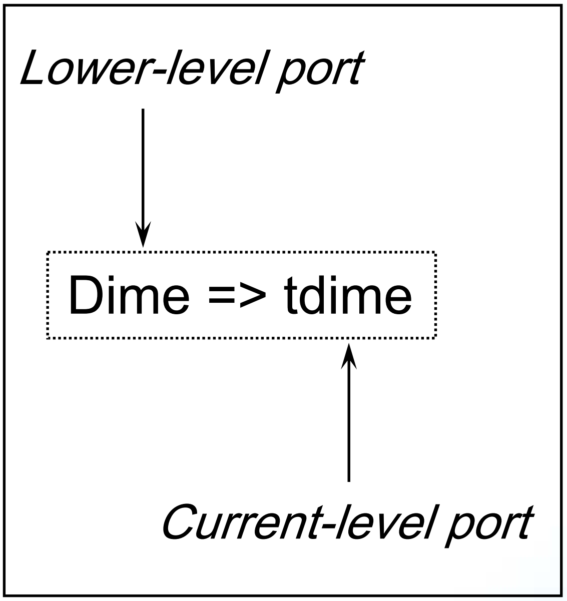
Positional association

Instance label/name

Component instantiation

Component Declaration and Instantiation

```
Library IEEE;
Use  IEEE.Std_logic_1164.ALL;
ENTITY tolleab IS
PORT(          tclk : IN Std_logic;
                        Tcross, tnickel, tdime, tquarter : IN Std_logic;
                        Tgreen, tred : OUT Std_logic);
END tolleab;
ARCHITECTURE tolleab_arch OF tolleab IS
TYPE STATE_TYPE IS (cent0, cent5, cent10, cent15, cent20, cent25, cent30,
                        Cent35, cent40, cent45, cent50, arrest);
SIGNAL connect : STATE_TYPE;

Component tollv
PORT(          clk: IN Std_logic;
               Cross, nickel, dime, quarter : IN Std_logic;
               Green, red : OUT Std_logic;
               Sout : OUT STATE_TYPE;
               State_in : IN STATE_TYPE);
End component;

Begin

U1 :  tollv port map (clk => tclk, cross => tcross, nickel => tnickel, dime => tdime,
                        Quarter => tquarter, green => tgreen, red => tred,
                        Sout => connect, state_in => connect);

End tolleab_arch;
```

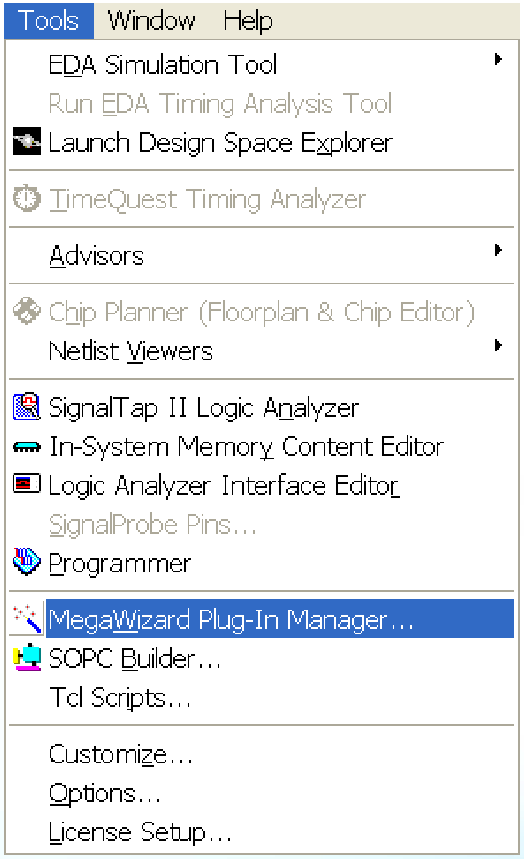


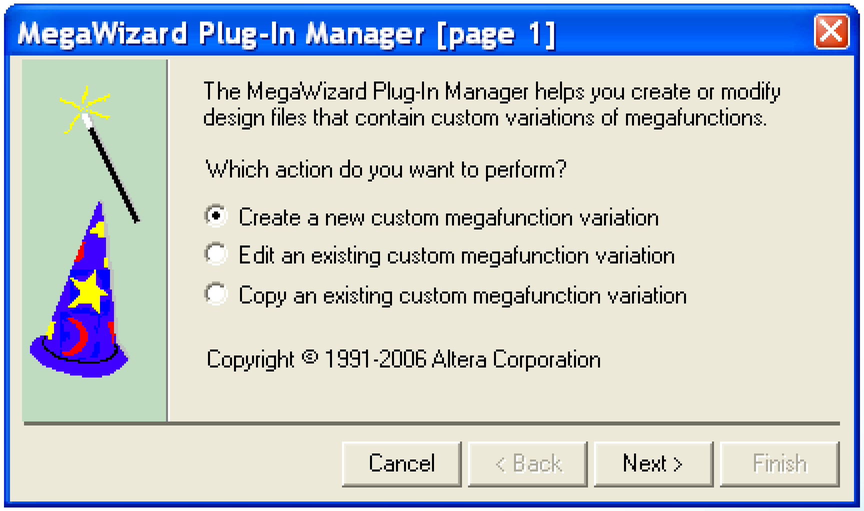
Vendor Libraries

- Silicon vendors often provide libraries of macrofunctions & primitives
  - Altera library
    - Maxplus2
    - Megacore
- Can be used to control physical implementation of design within the PLD
- Vendor-specific libraries improve performance & efficiency of designs
- Altera provides a collection of library of parameterized modules (LPM) plus other megafunctions and primitives

Accessing the MegaWizard Tool

- Altera's IP, megafunction, & LPMs accessed and edited through the MegaWizard plug-in manager


Tools
Window
Help
EDA Simulation Tool
Run EDA Timing Analysis Tool
Launch Design Space Explorer
TimeQuest Timing Analyzer
Advisors
Chip Planner (Floorplan & Chip Editor)
Netlist Viewers
SignalTap II Logic Analyzer
In-System Memory Content Editor
Logic Analyzer Interface Editor
SignalProbe Pins...
Programmer
MegaWizard Plug-In Manager...
SOPC Builder...
Tcl Scripts...
Customize...
Options...
License Setup...



MegaWizard Plug-In Manager [page 1]

The MegaWizard Plug-In Manager helps you create or modify design files that contain custom variations of megafunctions.

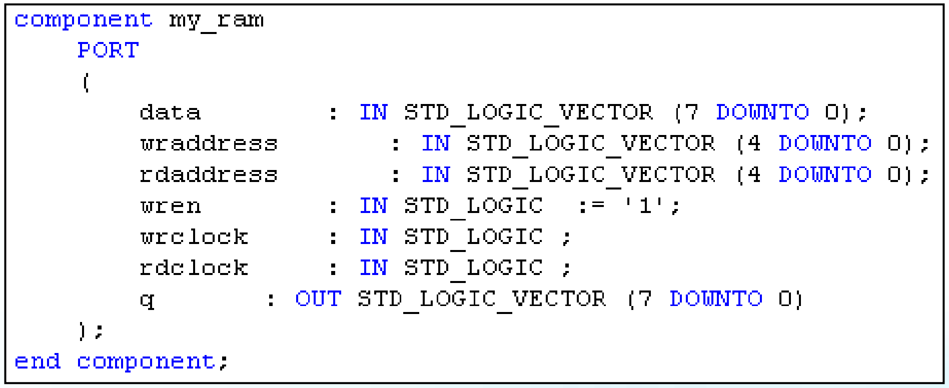
Which action do you want to perform?

● Create a new custom megafunction variation
○ Edit an existing custom megafunction variation
○ Copy an existing custom megafunction variation

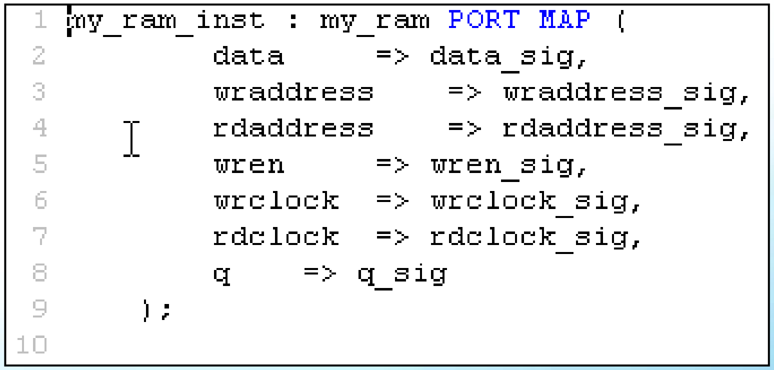
Copyright © 1991-2006 Altera Corporation

MegaWizard Tool Files

- The MegaWizard plug-in manager produces three files relevant to VHDL
  - my_ram.vhd
    - Instantiation and parameterization of megafunction
  - my_ram.cmp
  - Component declaration for use in higher level file
  - my_ram_inst.vhd
    - Instantiation of my_ram for use in higher level file



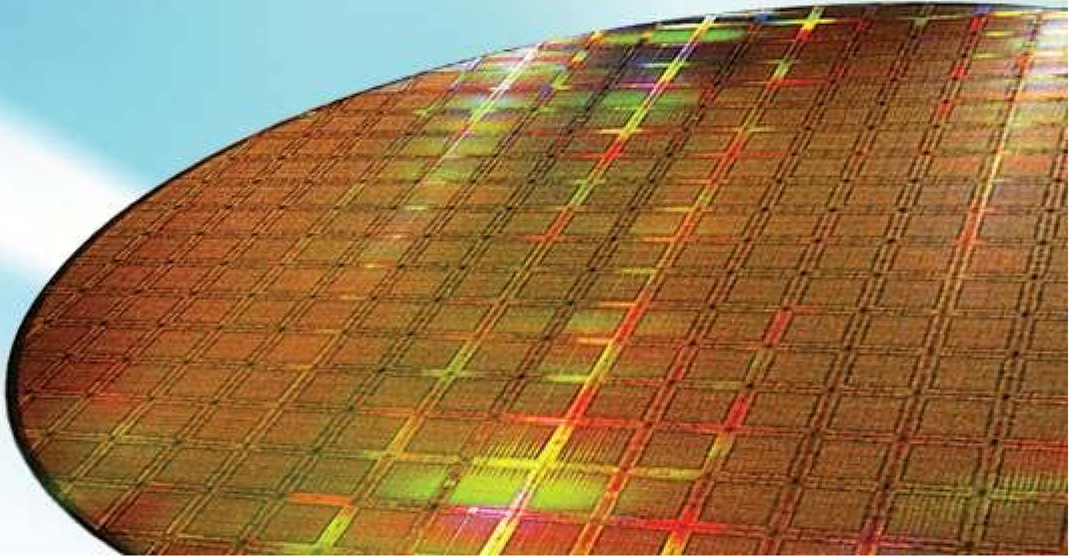
```
component my_ram
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        wraddress       : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        rdaddress       : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        wren        : IN STD_LOGIC  := '1';
        wrclock     : IN STD_LOGIC ;
        rdclock     : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
end component;
```



```
my_ram_inst : my_ram PORT MAP (
        data     => data_sig,
        wraddress    => wraddress_sig,
        rdaddress    => rdaddress_sig,
        wren     => wren_sig,
        wrclock  => wrclock_sig,
        rdclock  => rdclock_sig,
        q    => q_sig
    );

```

Exercise 6

Please Go to Exercise 6



Altera Technical Support

- Reference Quartus II software on-line help
- Consult altera applications (factory applications engineers)
  - Hotline: (800) 800-EPLD (7:00 a.m. - 5:00 p.m. PST)
  - Mysupport: http://www.altera.com/mysupport
- Field applications engineers: contact your local Altera sales office
- Receive literature by mail: (888) 3-ALTERA
- FTP: ftp.altera.com
- World-wide web: http://www.altera.com
  - Use solutions to search for answers to technical problems
  - View design examples

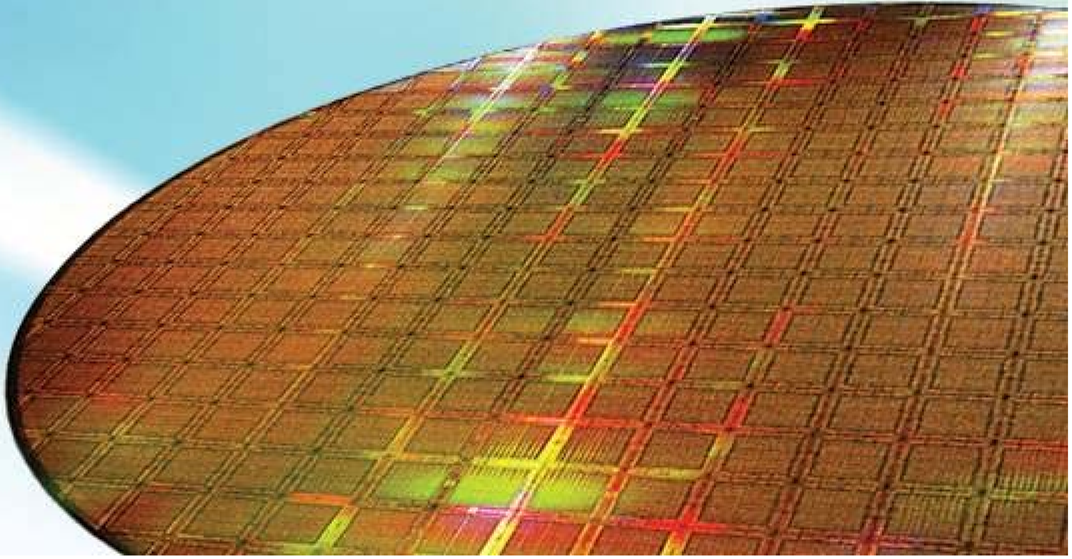
Learn More Through Technical Training

Instructor-Led Training	Online Training
 with Altera's instructor-led training courses, you can: ➢Listen to a lecture from an Altera technical training engineer (instructor) ➢Complete hands-on exercises with guidance from an Altera instructor ➢Ask questions & receive real-time answers from an Altera instructor ➢Each instructor-led class is one or two days in length (8 working hours per day).	 with Altera's online training courses, you can: ➢Take a course at any time that is convenient for you ➢Take a course from the comfort of your home or office (no need to travel as with instructor-led courses) Each online course will take about one hour to complete.

www.altera.com/training

View Training Class Schedule & Register for a Class

Appendix



LPM

- **L**ibrary of **Parameterized M**odules
  - Large building blocks that are easily configurable by using the megawizard plug-in manager

- Altera's LPMs have been optimized to access the architectural features of Altera devices

LPM Instantiation

- All of the Altera LPM macrofunctions are declared in the package **LPM_components.all** in the **LIBRARY LPM**
- The M**egaWizard plug-in manager** in Quartus II software creates the VHDL code instantiating the LPM or megafunction
- After the code is created you will see at the top of the VHDL code:

```
Library LPM;
Use LPM.LPM_components.all;
```

Manual LPM Instantiation – LPM_MUX

- Quartus II or MAX+plus II Online HELP: VHDL Component Declaration:

```
COMPONENT LPM_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_PIPELINE: INTEGER:= 0;
    LPM_SIZE: POSITIVE;
    LPM_HINT: STRING := UNUSED);
  PORT (data: IN Std_logic_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    aclr: IN Std_logic := '0';
    clock: IN Std_logic := '0';
    sel: IN Std_logic_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT Std_logic_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_arith.All;
Use IEEE.Std_logic_signed.All;

Library LPM;
USE LPM.LPM_components.All;

Entity tst_mux is
PORT (a : in Std_logic_2d (3 downto 0, 15 downto 0);
                    Sel : in Std_logic_vector(1 downto 0);
                    Y : out Std_logic_vector (15 downto 0));
END tst_mux;

Architecture behavior of tst_mux is
Begin

U1: LPM_mux generic map(LPM_width => 16, LPM_size => 4, LPM_widths => 2)
            PORT MAP (data => a, sel => sel,  result => y);

End behavior;
```

Manual LPM Instantiation – LPM_MULT

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
Use IEEE.Std_logic_unsigned.All;

Library LPM;
USE LPM.LPM_components.All;

Entity tst_mult is
PORT ( a, b : in Std_logic_vector(7 downto 0);
             Q_out  : out Std_logic_vector(15 downto 0));
END tst_mult;

Architecture behavior of tst_mult is

Begin

             U1 : LPM_mult generic map (LPM_widtha => 8, LPM_widthb => 8,
                                         LPM_widths => 16, LPM_widthp => 16)
                                 PORT map(dataa => a, datab => b, result => q_out);

End behavior;
```

Benefits of LPMs

- Industry standard
- Larger building blocks, so you don't have to start from scratch
  - Reduces design time
  - Therefore, faster time-to-market
- Easy to change the functionality by using the MegaWizard
- Consistent synthesis

Attributes

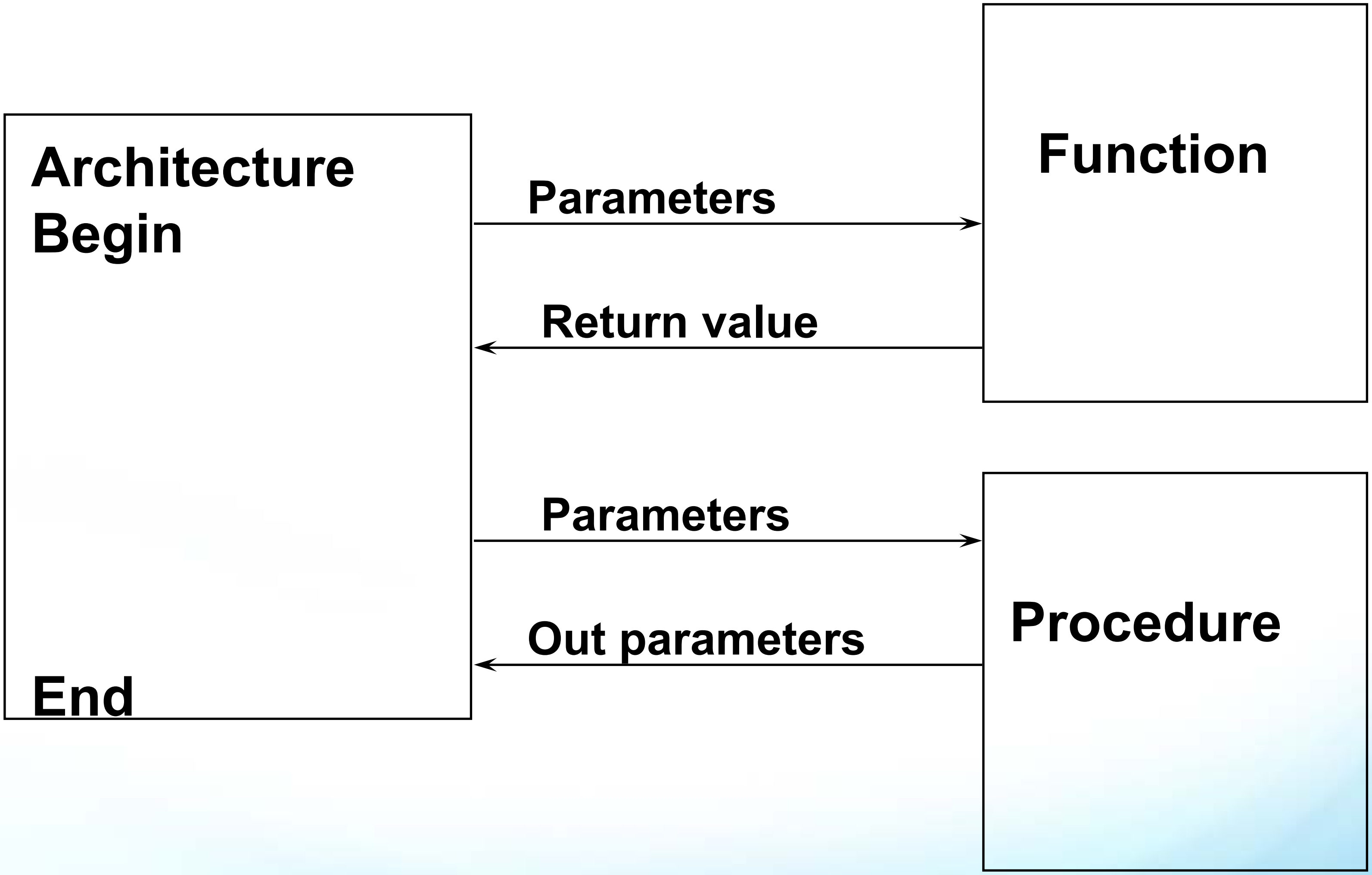
<Signal_name> : IN Std_logic_VECTOR(7 DOWNTO 0)

- **‘High** - 7
- **‘low** - 0
- **‘Right** - 0
- **‘Left** - 7
- **‘Range** - 7 downto 0
- **‘Reverse range** - 0 to 7
- **‘Length** - 8

Subprograms

- Functions
- Procedures

Subprograms



Functions

- Format:

```
Function <function_name> (<input_parameters>)
Return <DATA_TYPE> is
                {Any declarations}
Begin
                {Functionality}
      Return <name_of_a_declaration>
End <function_name>;
```

Functions

- For functions:
  - Only allowable mode for parameters is **in**
  - Only allowed object classes are **constant** or **signal**
  - If the object class is not specified, **constant** is assumed

Procedures

- Format:

```
Procedure <procedure_name> (<mode_parameters>)
    Begin
            {Functionality}
    End <procedure_name>;
```

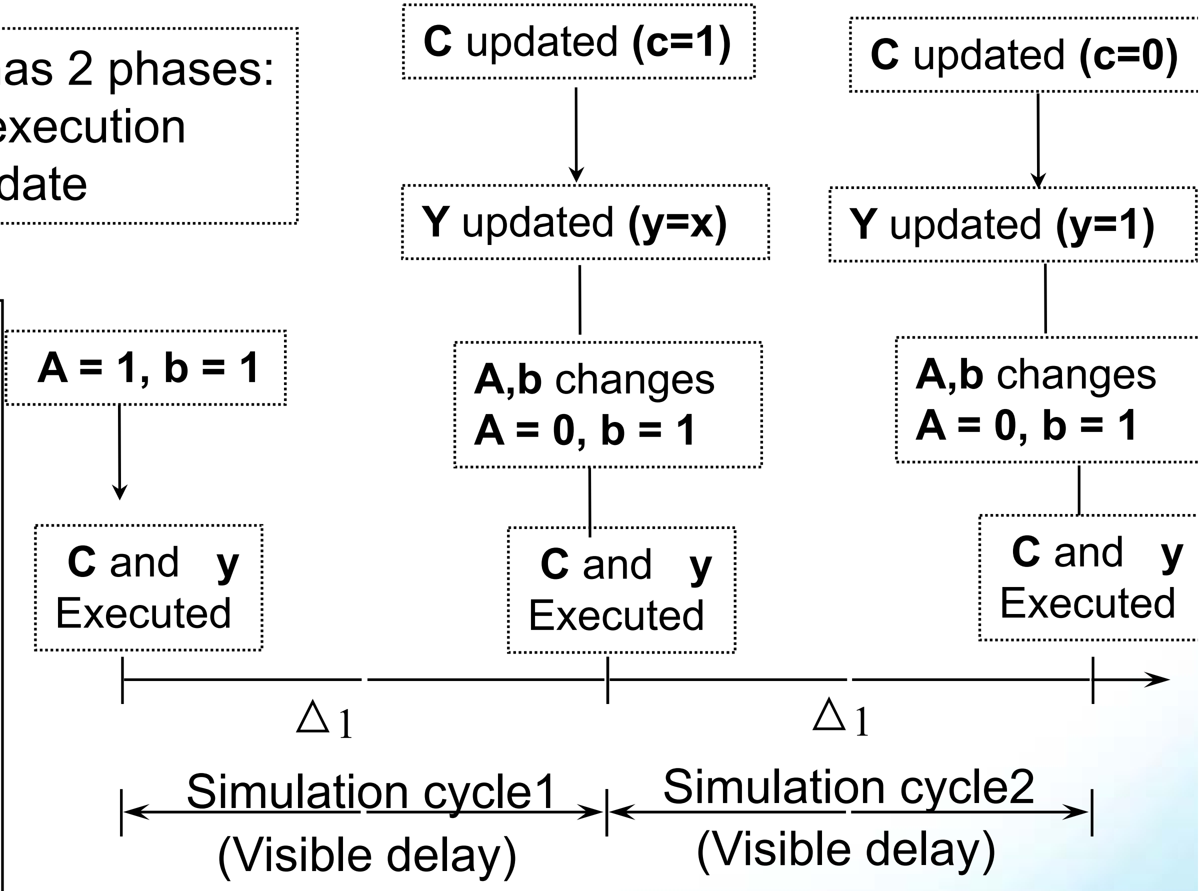
Procedures

- For procedures:
  - Allowable modes for parameters are **in, out,** and **inout**
  - Allowable object classes for parameters are **constant, variable** and **signal**
  - If the mode is **in** and no object class is specified, then **constant** is assumed
  - If the mode is **inout** or **out** and if no object class is specified, then **variable** is assumed

Signal Assignment Inside a Process - Delay

- △Delta cycle has 2 phases:
  – Process execution
  – Signal update

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY simp_prc IS
Port(a, b : IN Std_logic;
                  Y: out Std_logic);
END simp_prc;
ARCHITECTURE logic OF simp_prc IS
SIGNAL c: Std_logic;

Begin
Process(a, b)
                  Begin
                  C <= a and b;
                  Y <= c;
End process;
END logic;
```



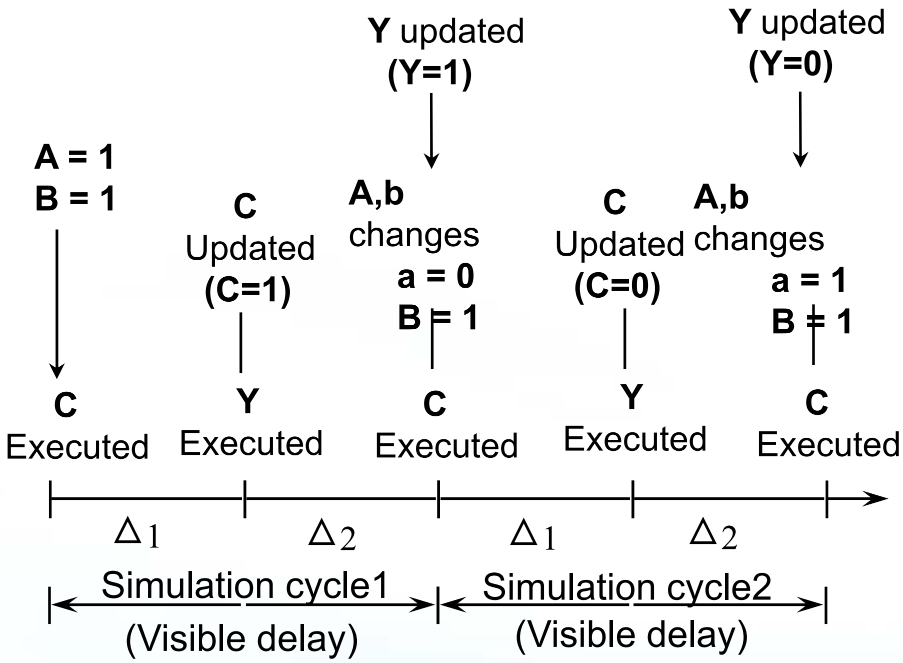
- **Y** does not get the newest value of **c** until a simulation cycle later

- △Delta cycle is non-visible delay (Very small, close to zero)

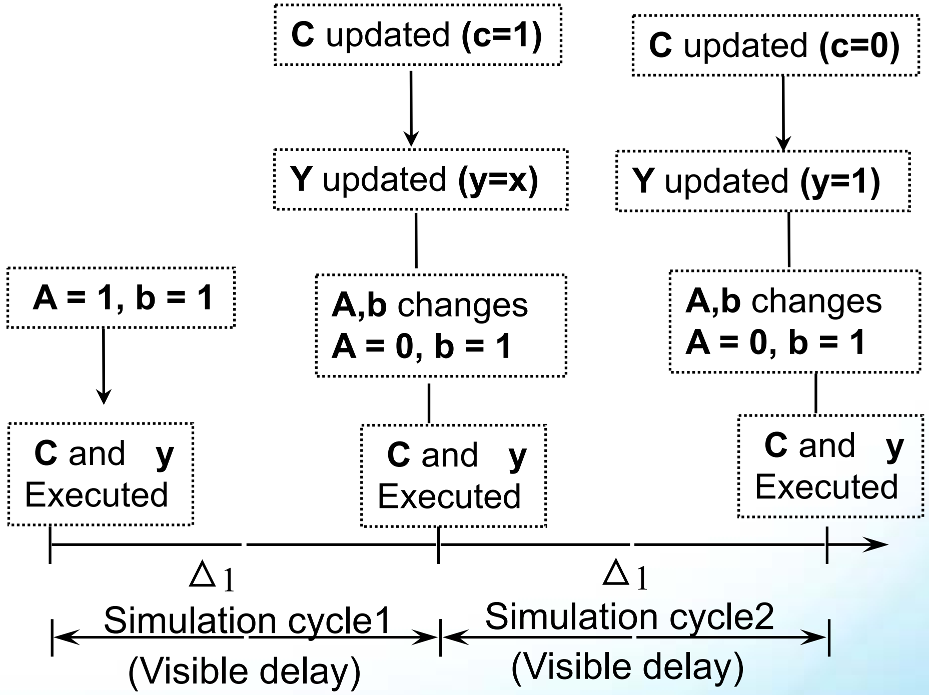
2 Processes vs. 1 Process

```
Process1: process(a, b)
          Begin
                    C <= a and b;
          END PROCESS process1;
Process2: process(c)
          Begin
                    Y <= c;
          END PROCESS process2;
```

```
Process(a, b)
          Begin
          C <= a and b;
          Y <= c;
End process;
```



- **C** and **y** gets executed and updated within the same simulation cycle

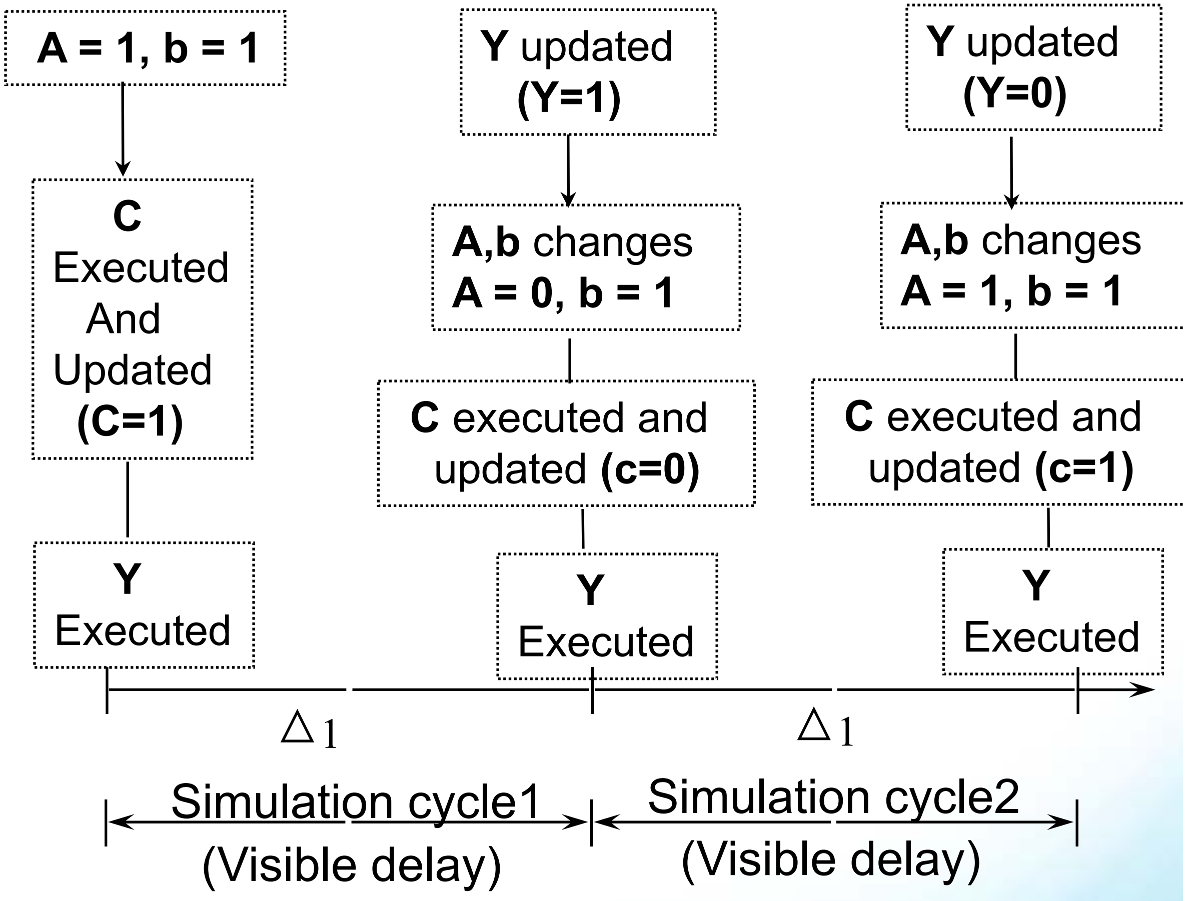


- **Y** does not get the newest value of **c** until a simulation cycle later

Variable Assignment - No Delay

- △Delta Cycle has 2 Phases:
  - Process Execution
  - Signal Update

```
Library IEEE;
Use IEEE.Std_logic_1164.ALL;
ENTITY var IS
PORT       (a, b : IN    Std_logic;
                Y : out  Std_logic);
END var;
ARCHITECTURE logic OF var IS
Begin
PROCESS (a, b)
VARIABLE   c  :  Std_logic;
BEGIN
C := a AND b;
Y <= c;

END PROCESS;
END logic;
```



A = 1, b = 1

C Executed And Updated (C=1)

Y Executed

Y updated (Y=1)

A,b changes A = 0, b = 1

C executed and updated (c=0)

Y Executed

Y updated (Y=0)

A,b changes A = 1, b = 1

C executed and updated (c=1)

Y Executed

△1

△1

Simulation cycle1 (Visible delay)

Simulation cycle2 (Visible delay)

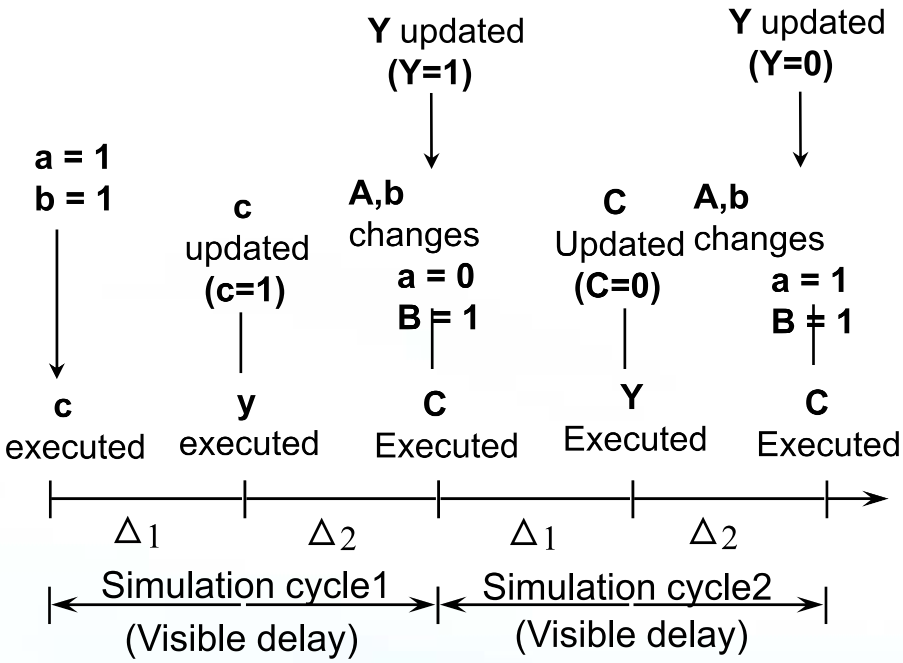
- **C** and **y** gets executed and updated within the same simulation cycle (at the end of the process)

- △Delta cycle is non-visible delay (Very small, close to zero)

2 Processes vs. 1 Process

```
Process1: process(a, b)
          Begin
                    C <= a and b;
          END PROCESS process1;
Process2: process(c)
          Begin
                    Y <= c;
          END PROCESS process2;
```

```
Process(a, b)
          Begin
          C <= a and b;
          Y <= c;
End process;
```



Y updated **(Y=1)**

Y updated **(Y=0)**

a = 1
b = 1

c executed

c updated **(c=1)**

y executed

A,b changes **a = 0** **B = 1**

C Executed

C Updated **(C=0)**

Y Executed

A,b changes **a = 1** **B = 1**

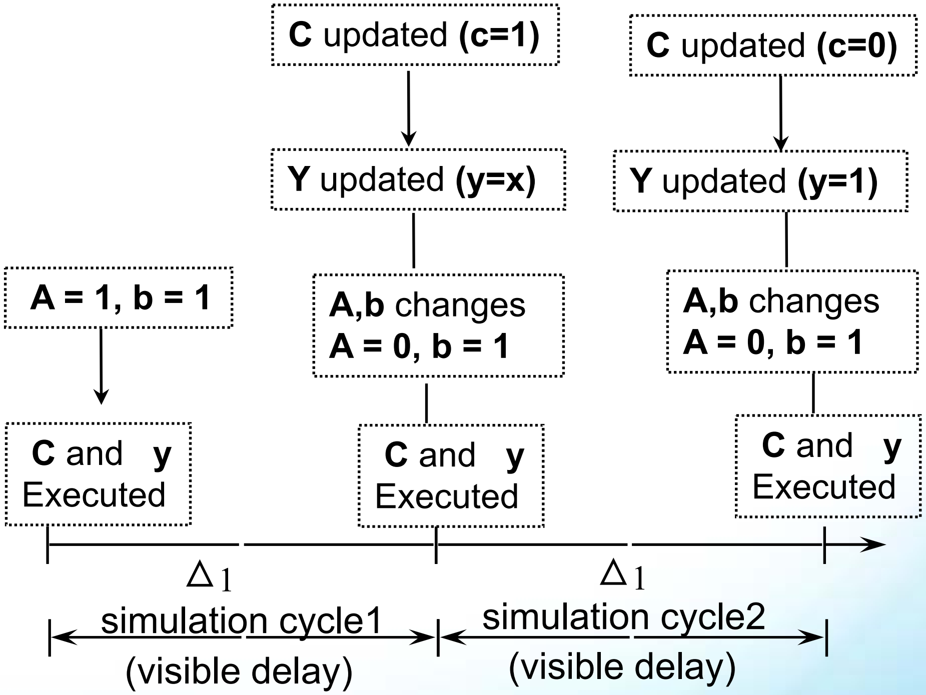
C Executed

Δ1 Δ2 Δ1 Δ2

Simulation cycle1 (Visible delay)

Simulation cycle2 (Visible delay)

- **C** and **y** gets executed and updated within the same simulation cycle



C updated **(c=1)**

C updated **(c=0)**

Y updated **(y=x)**

Y updated **(y=1)**

A = 1, b = 1

A,b changes **A = 0, b = 1**

A,b changes **A = 0, b = 1**

C and **y** Executed

C and **y** Executed

C and **y** Executed

Δ1 Δ1

simulation cycle1 (visible delay)

simulation cycle2 (visible delay)

- **Y** does not get the newest value of **c** until a simulation cycle later