Introduction to VHDL

Course Objectives

- Learn the basic constructs of VHDL
- Learn the modeling structure of VHDL
- Understand the design environments
  - Simulation
  - Synthesis

Course Outline

- VHDL Basics
  - Overview of language
- Design Units
  - Entity
  - Architecture
  - Configurations
  - Packages (Libraries)
- Architecture Modeling Fundamentals
  - Signals
  - Processes
    - Sequential Statements

Course Outline

- Understanding VHDL and Logic Synthesis
  - Process Statement
  - Inferring Logic
- Model Application
  - State Machine Coding
- Hierarchical Designing
  - Overview
  - Structural Modeling
  - Application of LPM's

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

VHDL History

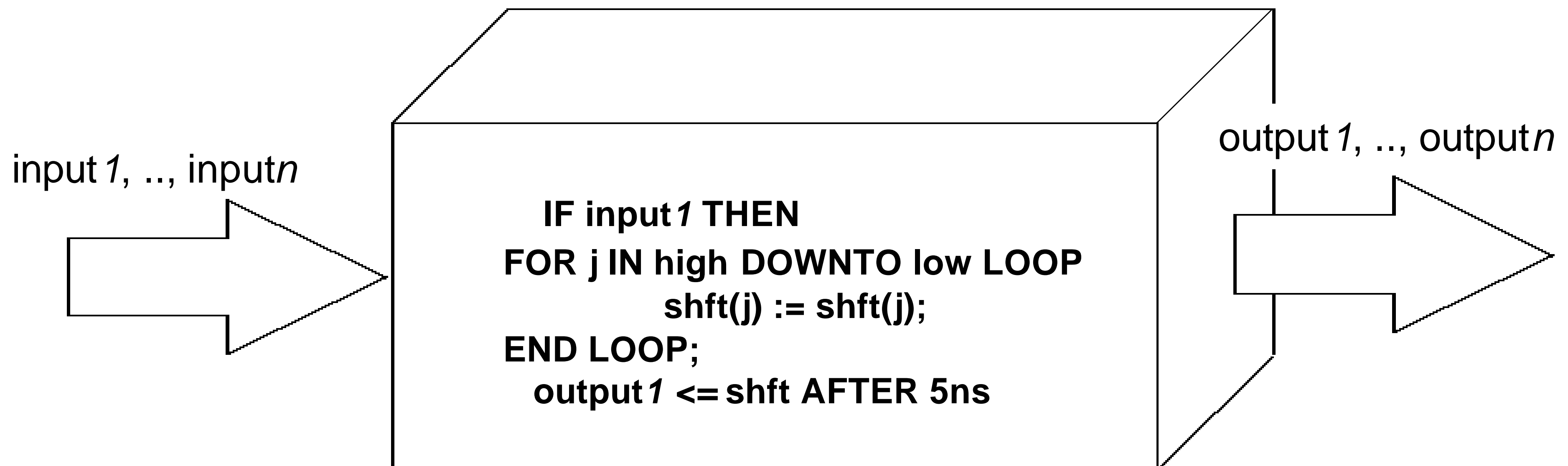
- 1980 - U.S. Department of Defense (DOD) funded a project to create a standard hardware description language under the Very High Speed Integrated Circuit (VHSIC) program.
- 1987 - the Institute of Electrical and Electronics Engineers (IEEE) ratified as IEEE Standard 1076.
- 1993 - the VHDL language was revised and updated to IEEE 1076 '93.

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

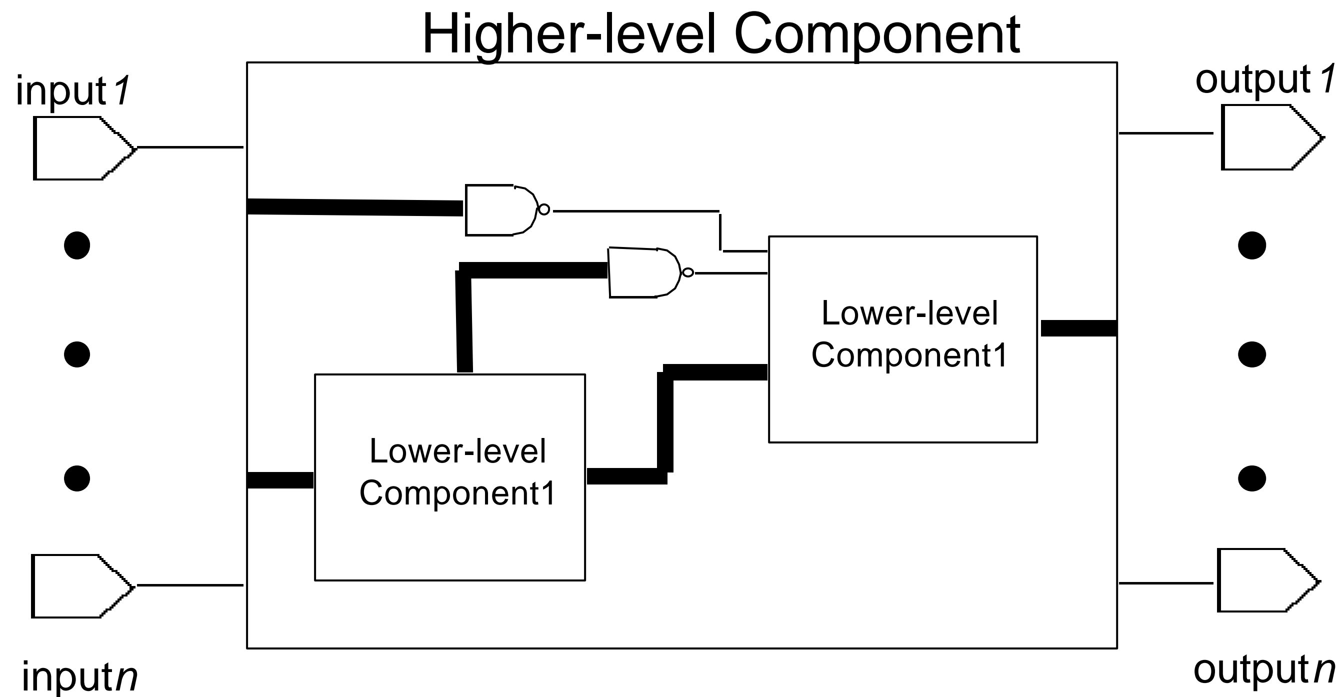
Behavior Modeling

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



Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware
- For the purpose of synthesis



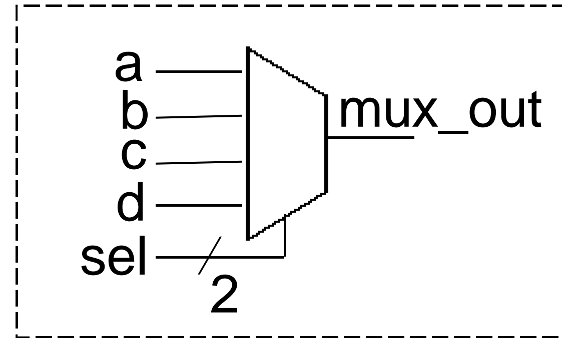
More Terminology

- Register Transfer Level (RTL) - A type of behavioral modeling, for the purpose of synthesis.
  - Hardware is implied or inferred
  - Synthesizable
- Synthesis - Translating HDL to a circuit and then optimizing the represented circuit
- RTL Synthesis - The process of translating a RTL model of hardware into an optimized technology specific gate level implementation

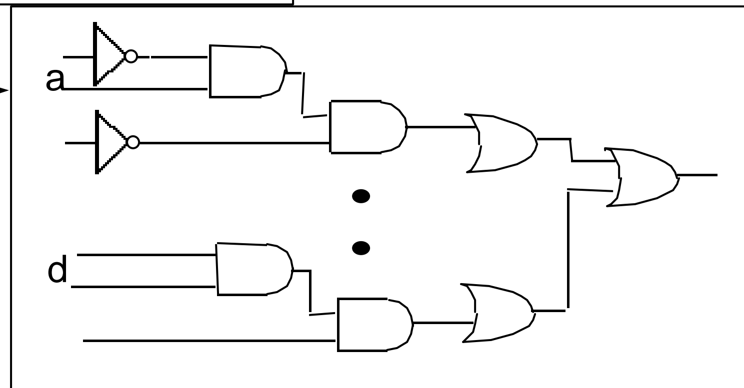
RTL Synthesis

```
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;
```

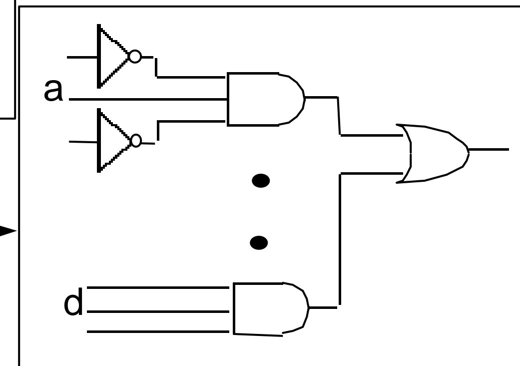
inferred



Translation



Optimization



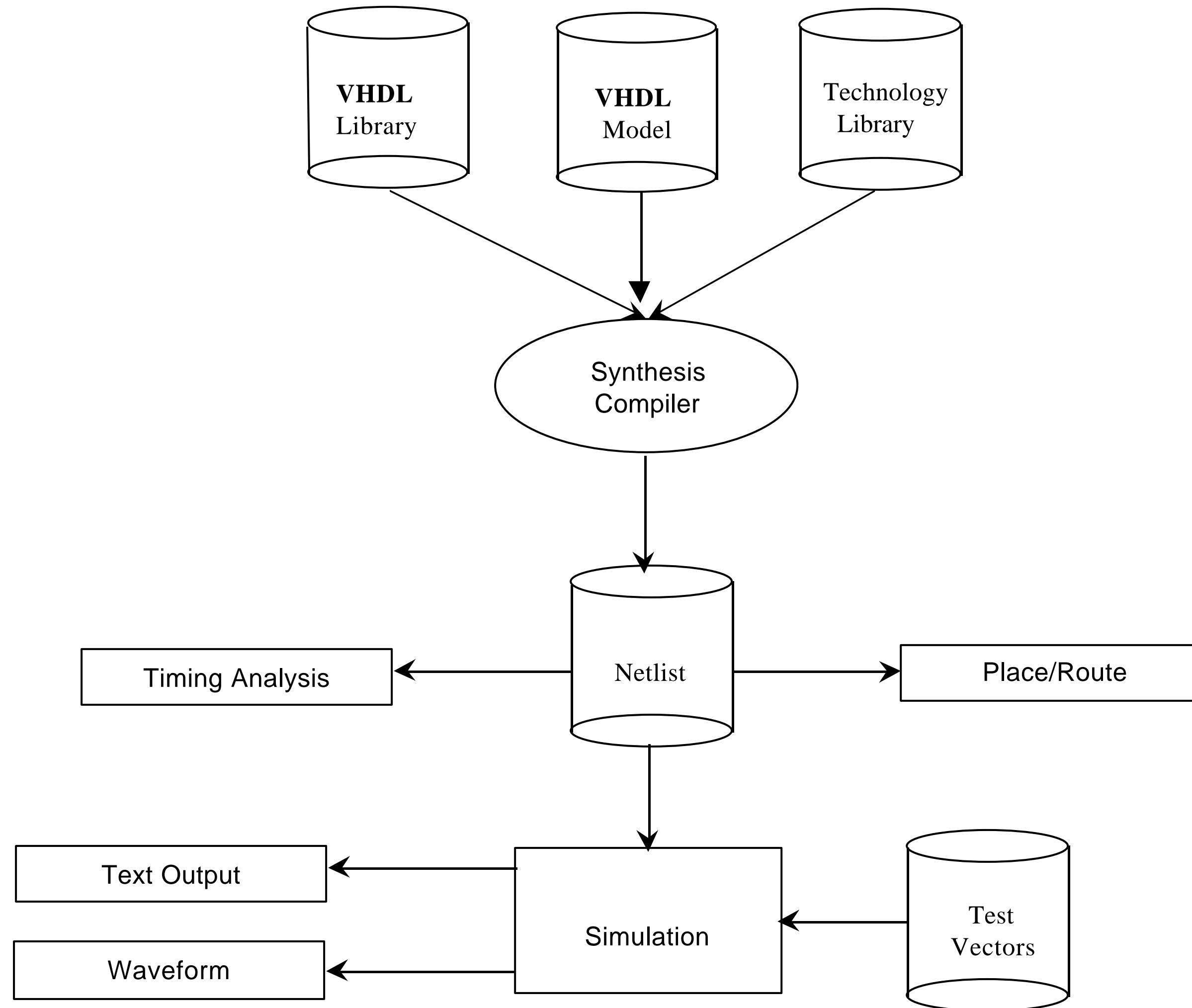
VHDL Synthesis vs. Other HDL Standards

- 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"

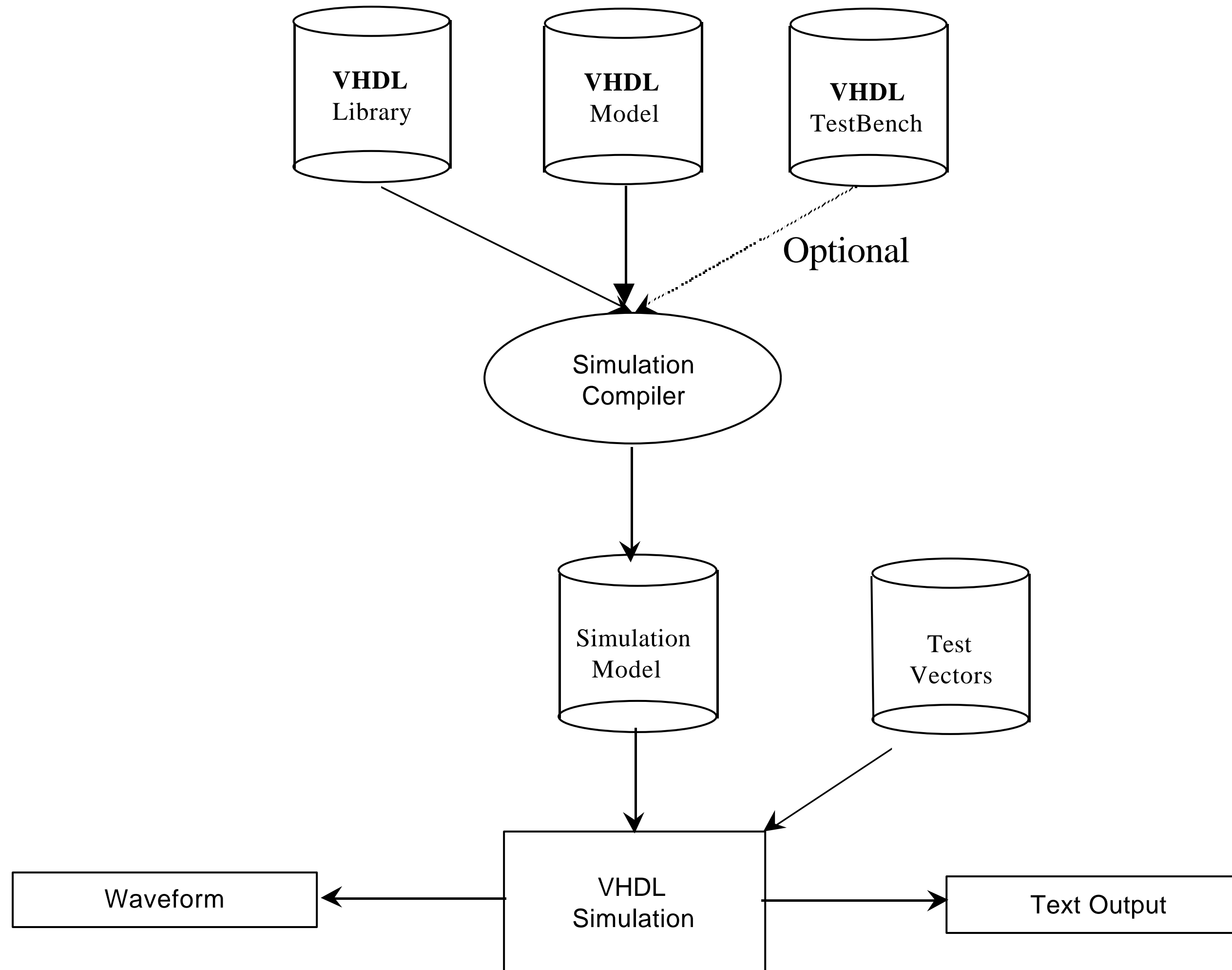
VHDL Synthesis vs. Other HDL Standards

- VHDL
  - "Give me a circuit whose output only changes when there is a low-to-high transition on a particular input. When the transition happens, make the output equal to the input until the next transition."
  - Result: VHDL Synthesis provides a positive edge-triggered flipflop
- ABEL, PALASM, AHDL
  - "Give me a D-type flipflop."
  - Result: ABEL, PALASM, AHDL synthesis provides a D-type flipflop. The sense of the clock depends on the synthesis tool.

Typical Synthesis Design Flow



Typical Simulation Design Flow



VHDL Basics

- Two sets of constructs:
  - Synthesis
  - Simulation
- 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. Used for readability.
- Comments in VHDL begin with "**--**" to eol
- VHDL models can be written:
  - Behavioral
  - Structural
  - Mixed

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
  - Note: MAX+PLUS II requires that the <entity_name> and <file_name> be the same.
- Generic Declarations
  - Used to pass information into a model.
  - 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

- Key aspects of the 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**

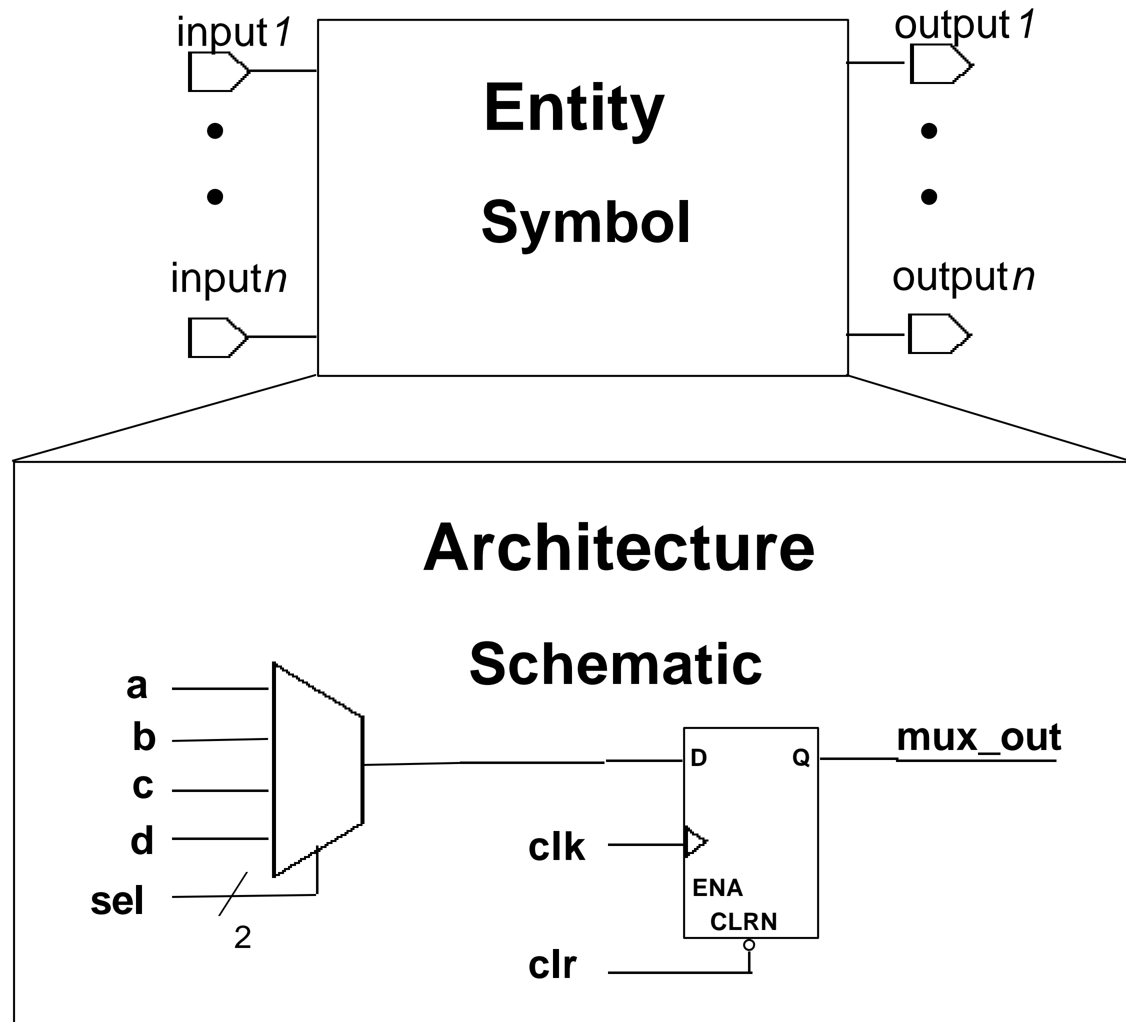
```
--architecture declaration section (list does not include all)
        signal temp : integer := 1;  -- Signal Declarations :=1 is default value optional
        constant load : boolean := true;  --Constant Declarations
        type states is ( S1, S2, S3, S4) ;   --Type Declarations
        --Component 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



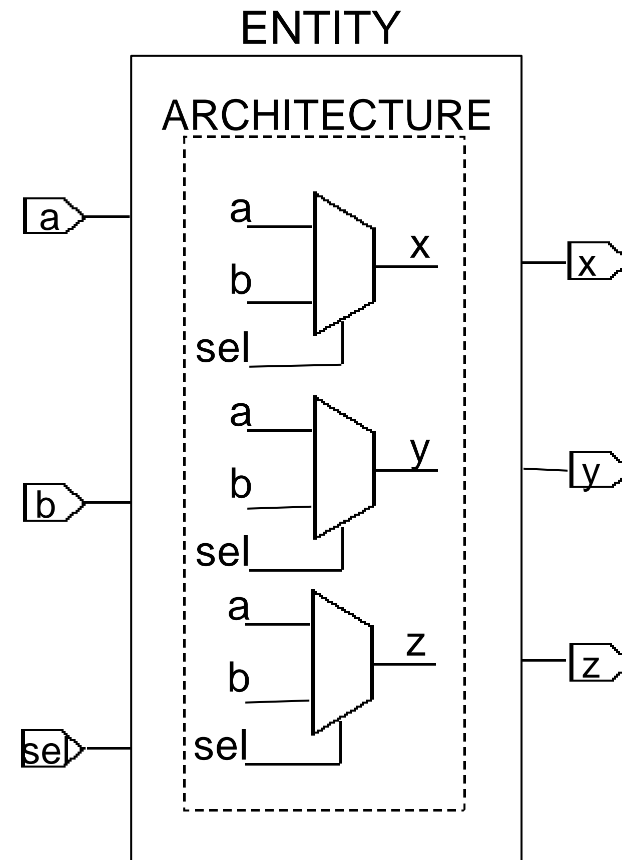
Configuration

- 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 (variable a , b : integer) RETURN boolean;
END filt_cmp;
PACKAGE BODY filt_cmp IS
FUNCTION compare (variable a , b : integer) IS
   VARIABLE temp : boolean;
 Begin
            If a < b then
                temp := true ;
           else
                temp := false ;
            end if;
           RETURN temp ;
END compare ;
END fily_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 define 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 1st 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.

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;**

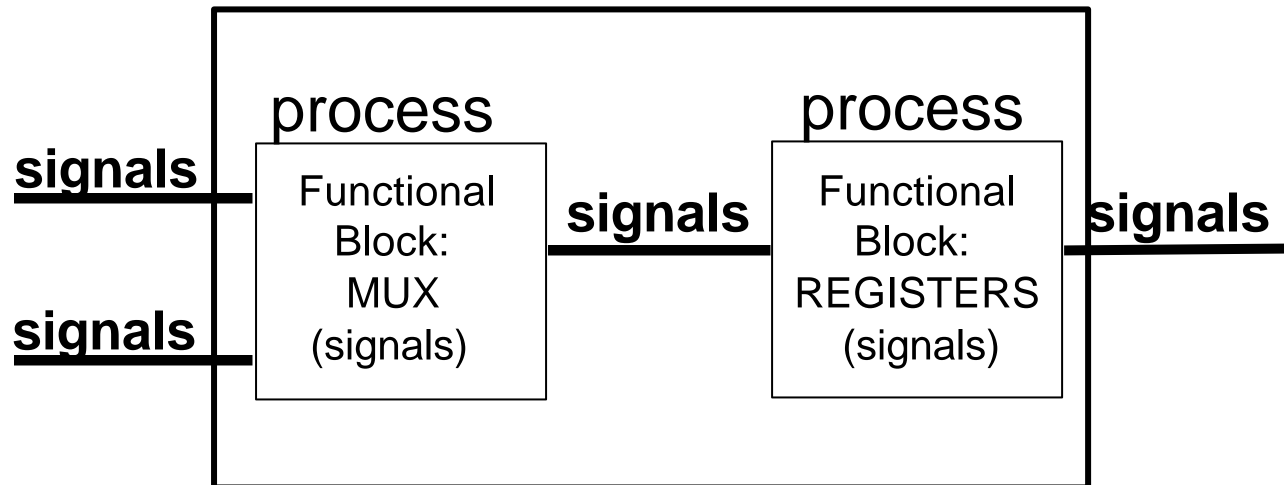
Architecture Modeling Fundamentals

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**



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';

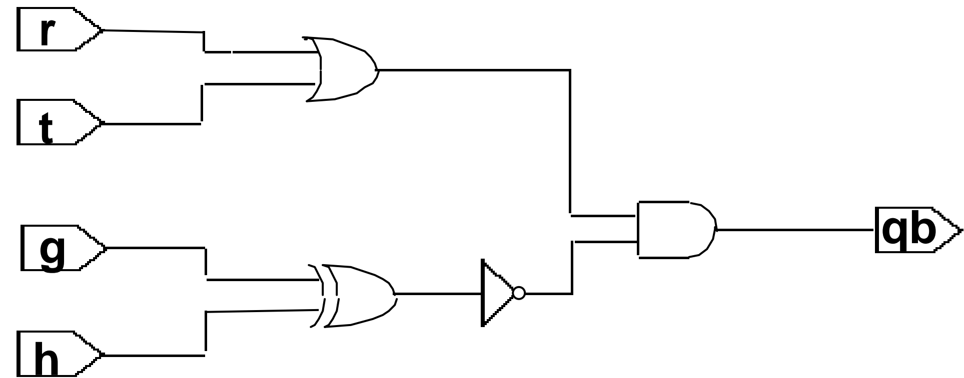
- Bit-slicing:

  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;
```

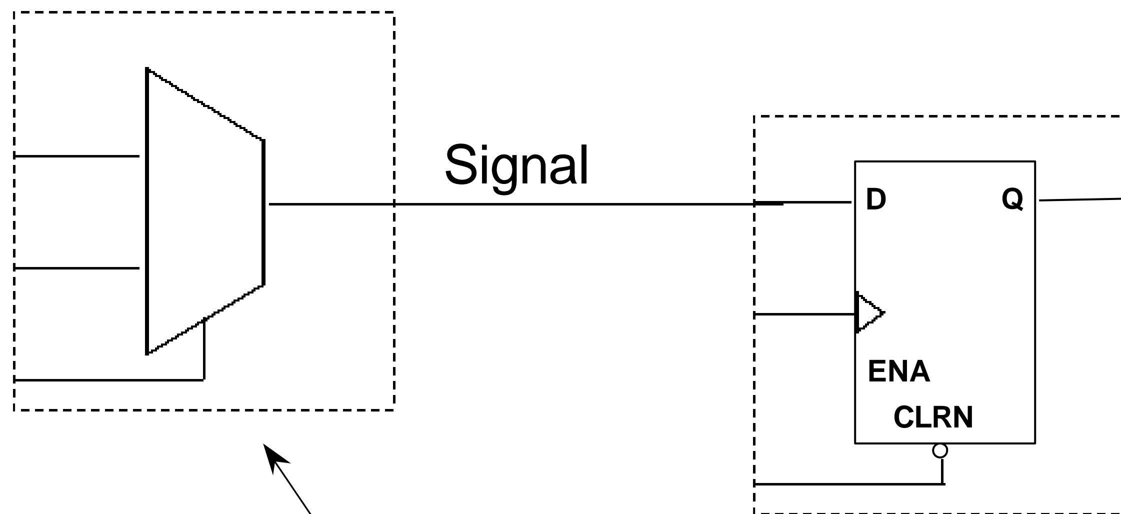


- **r, t, g, h**, and **qb** are Signals (by default)
- **qa** is a buried Signal and needs to be declared

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

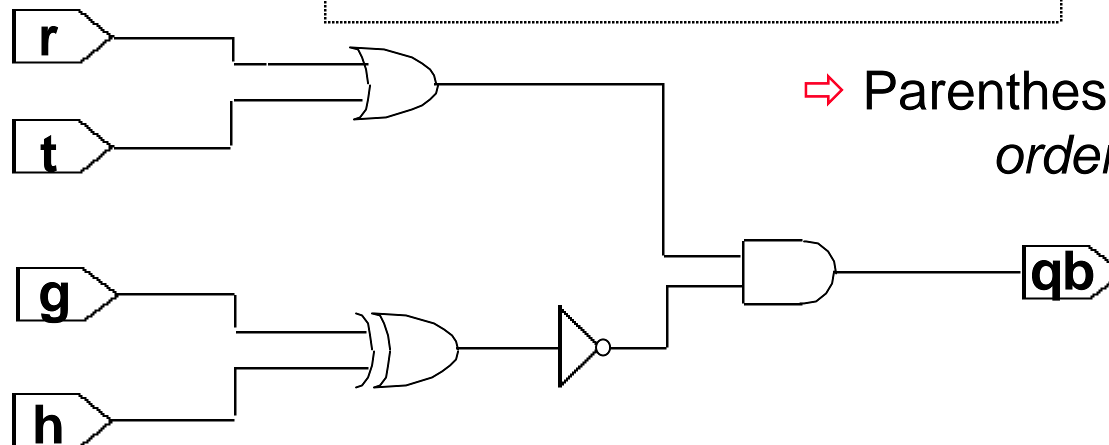
Simple Signal Assignments

- 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(1)
Relational	= /= < <= > >=
Adding	+ - &
Signing	+ -
Multiplying	* / mod rem
Miscellaneous	** abs not

(1) Supported in VHDL ‘93 only

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
adder_body:PROCESS (a, b)
BEGIN
        sum <= a + b;
END PROCESS adder_body;
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 (3 downto 0);
               b   : IN STD_LOGIC_VECTOR (3 downto 0);
               sum : OUT STD_LOGIC_VECTOR (4 downto 0));
END overload;

ARCHITECTURE example OF overload IS
BEGIN
adder_body:PROCESS (a, b)
BEGIN
        sum <= a + b;
END PROCESS adder_body;
END example;
```

Include these statements at the beginning of a design file

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

Concurrent Signal Assignments

- Three Concurrent Signal Assignments:
  - Simple Signal Assignment
  - Conditional Signal Assignment
  - Selected Signal Assignment

Conditional Signal Assignments

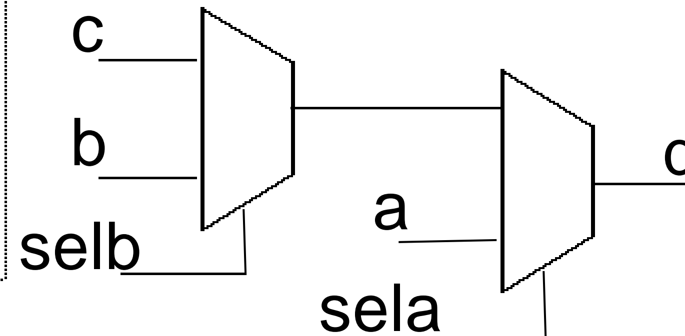
- Format:

```
<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

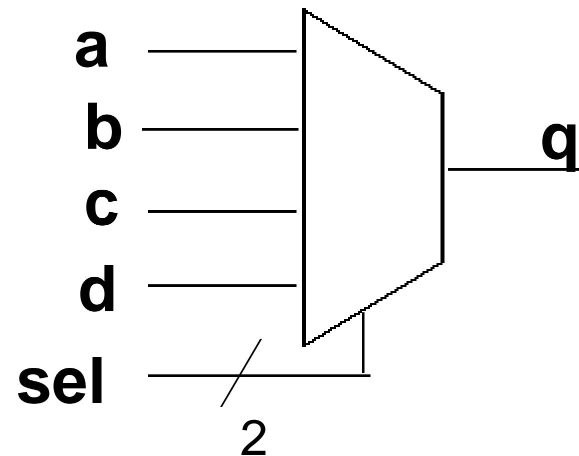
- Format:

```
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;
```

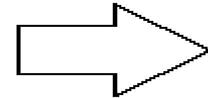
implied process



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;
```

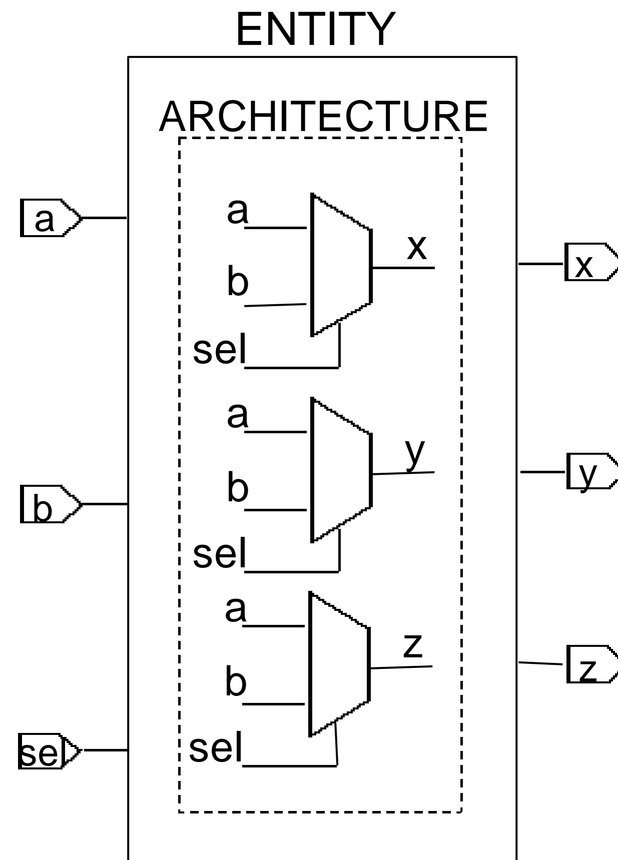
sel has a ***STD_LOGIC*** data type

- *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

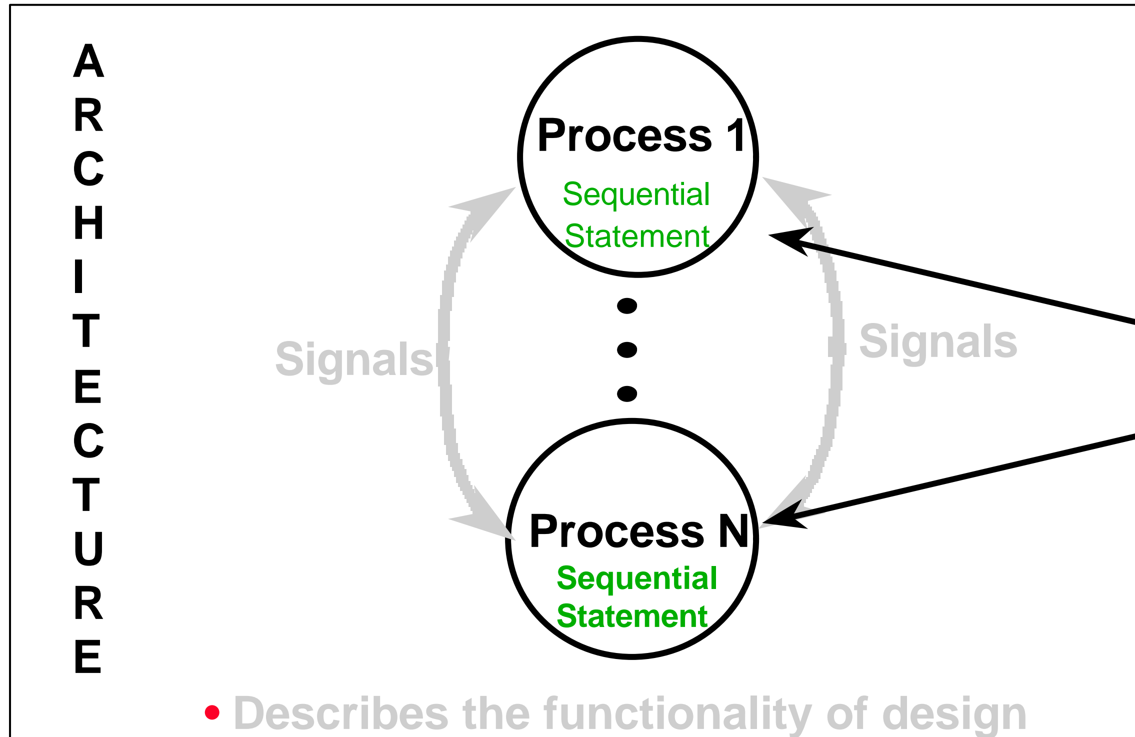
- Process Statement is executed infinitely unless broken by a WAIT statement or Sensitivity List.
  - Sensitivity list implies a WAIT statement at the end of the process.
  - Process can have multiple WAIT statements
  - Process can not have both a Sensitivity List and WAIT statement.

⇨ 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



- An Architecture can have multi-Process Statements.
- Each Process executes in parallel with each other.
- However, within a Process, the statements are executed sequentially.

VHDL Model - Multi-Process Architecture

```
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
if_label: 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 if_label;

case_label: 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 case_label;
END logic;
```

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

- Within a Process, the statements are executed sequentially

- Signal Assignments can also be inside Process statements.

Signal Assignment - delay

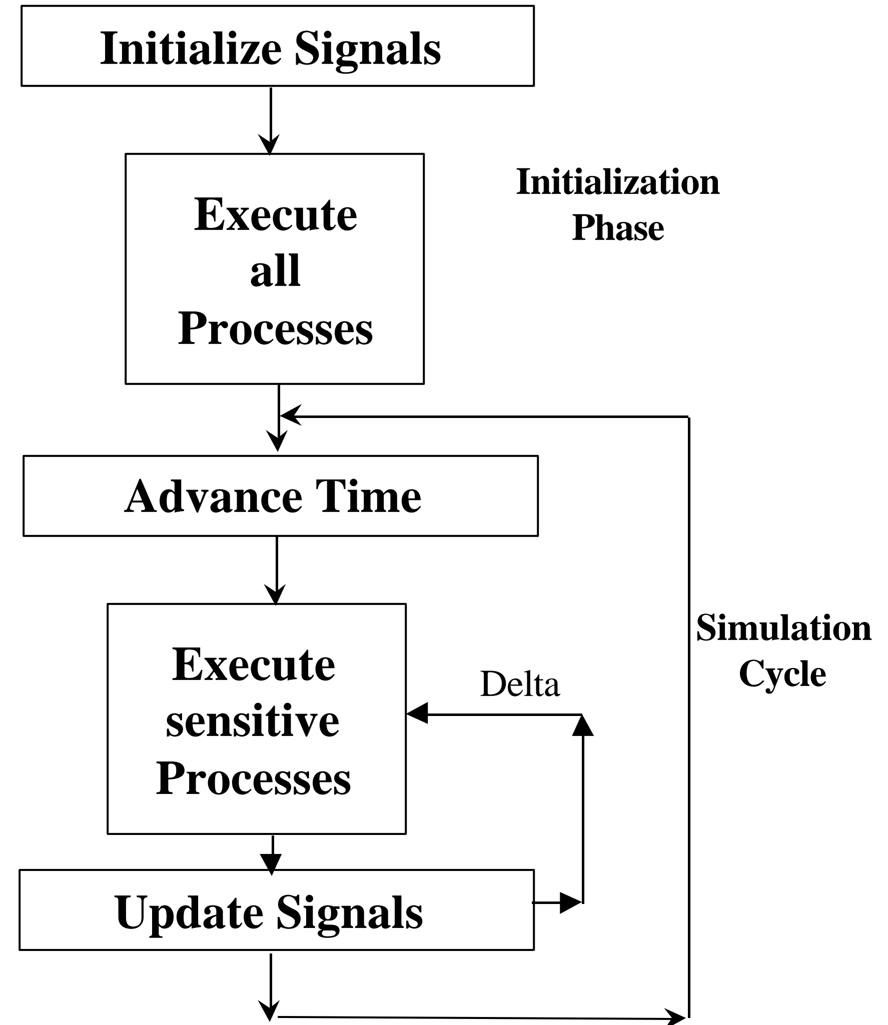
- Signal Assignments can be inside Process statements or outside (like the three concurrent signal assignments).

- 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
    - Transport Delay
      - Any pulse is transmitted no matter how short.

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

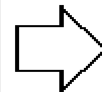
VHDL Simulation

- 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 begins?
  ⇨ **When:**
  - **All processes execute**
  - **Signals are updated**
- Signals get updated at end of process.



Equivalent Function

```
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;
```

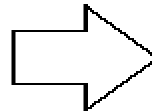


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY simp_prc IS
PORT(a,b : IN STD_LOGIC;
          y : OUT STD_LOGIC);
END simp_prc;
ARCHITECTURE logic OF simp_prc IS
SIGNAL c : STD_LOGIC;

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;
```



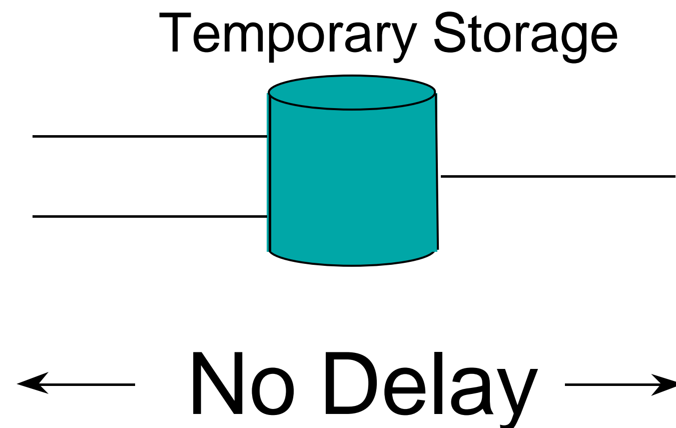
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY simp_prc IS
PORT(a, b : IN STD_LOGIC;
            y: OUT STD_LOGIC);
END simp_prc;
ARCHITECTURE logic OF simp_prc IS
SIGNAL c: STD_LOGIC;

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



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*

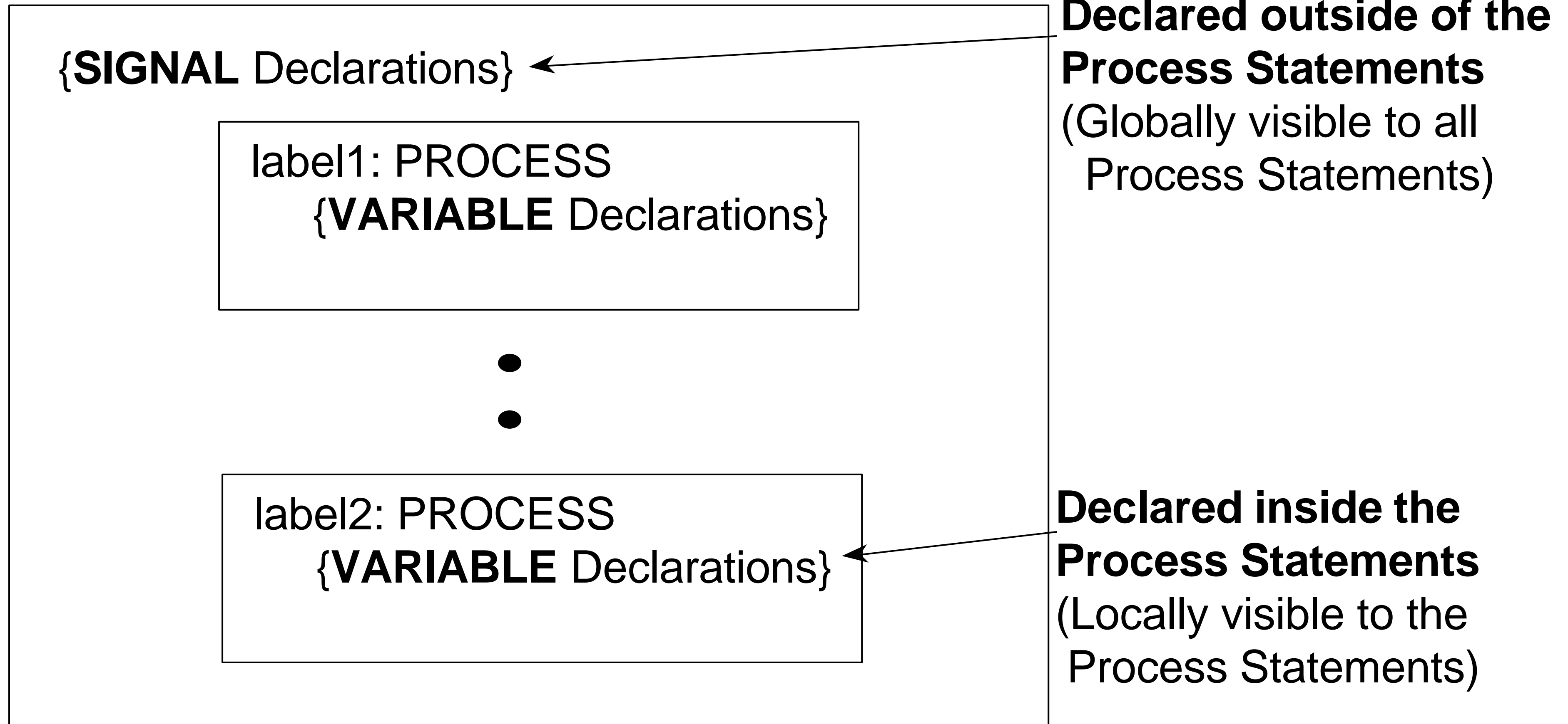
Use of a Variable

```
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
                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



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	Updated at end of Process Statement (new value not available)	Updated Immediately (new value available)

Sequential Statements

- Sequential Statements
  - IF-THEN statement
  - CASE statement
  - Looping Statements

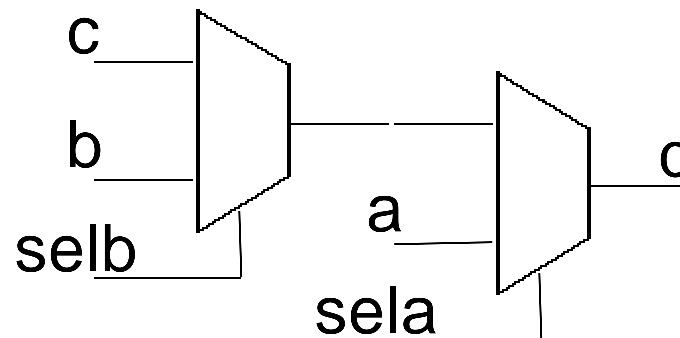
If-Then Statements

- 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

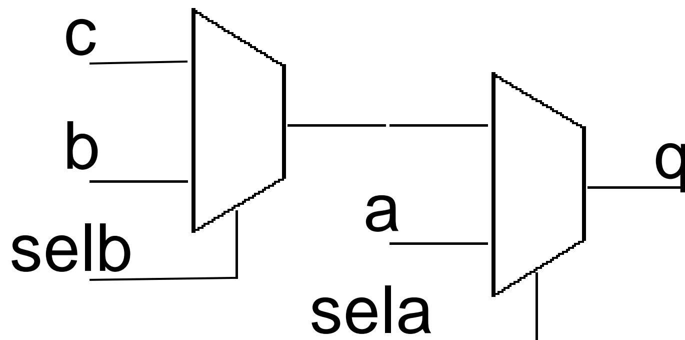
- Conditions are evaluated in order from top to bottom
  - Prioritization
- The first condition, that is true, causes the corresponding sequence of statements to be executed.
- If all conditions are false, then the sequence of statements associated with the “ELSE” clause is evaluated.

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;
```

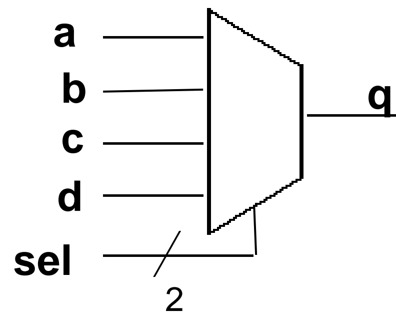
Case Statement

- 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

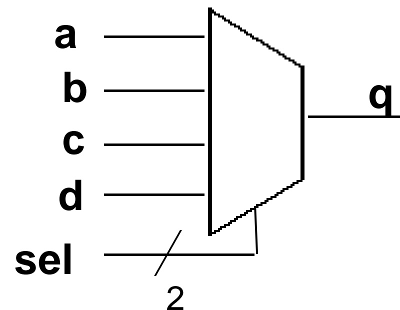
- Conditions are evaluated at once
  - No Prioritization
- **All** possible conditions must be considered
- **WHEN OTHERS** clause evaluates all other possible conditions that are not specifically stated.

Case Statements

- 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;
```

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;
```

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

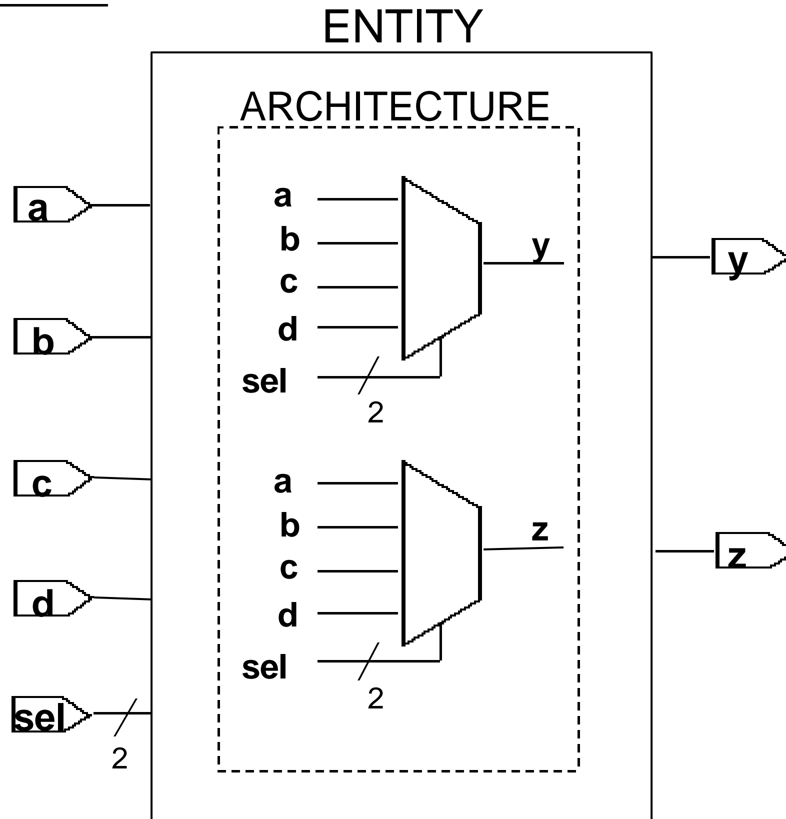
No shifting

Variable is assigned to a **Signal** before the end of the Process to synthesize to a piece of hardware

Understanding VHDL and Logic Synthesis

VHDL Model - RTL Modeling

Result:

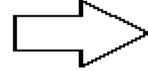


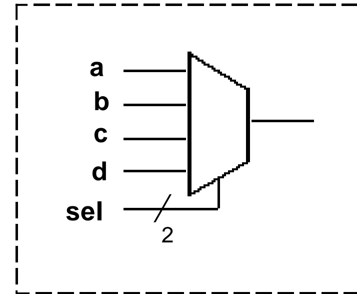
- **RTL** - Type of behavioral modeling that implies or infers hardware
- Functionality and somewhat structure of the circuit
- For the purpose of synthesis, as well as simulation

Recall - RTL Synthesis

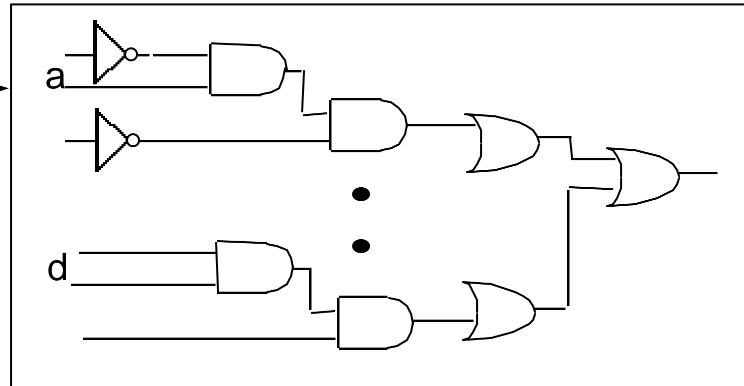
```
IF sel="00" THEN
        mux_out <= a;
ELSIF sel="01" THEN
        mux_out <= b;
 …………
 ELSE sel="11" THEN
        mux_out <= d;
```

inferred

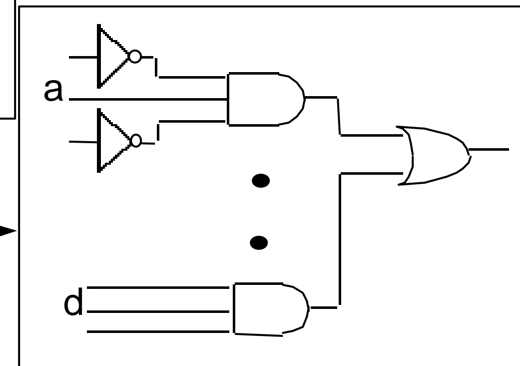




Translation



Optimization

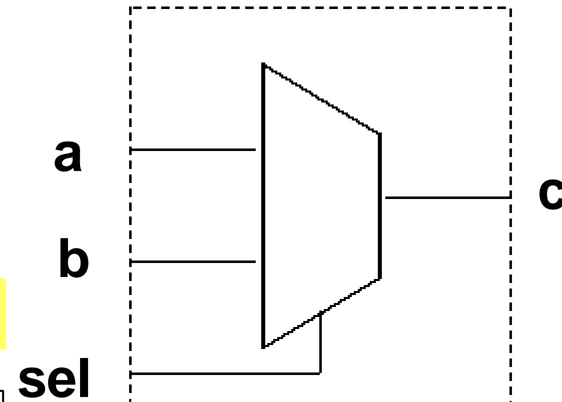


Two Types of Process Statements

- **Combinatorial Process**
  - Sensitive to all inputs used in the combinatorial logic
- **Example**

PROCESS(a, b, sel)

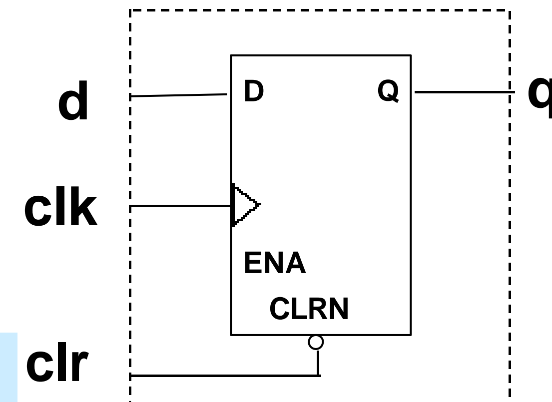
sensitivity list includes all inputs used in the combinatorial logic



- **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 latch IS
PORT ( data : IN std_logic;
                        gate : IN std_logic;
                        q : OUT std_logic
            );
END latch;

ARCHITECTURE behavior OF latch IS
BEGIN

label_1: PROCESS (data, gate)
            BEGIN
            IF gate = '1' THEN
                        q <= data;
            END IF;
END PROCESS;

END behavior;
```

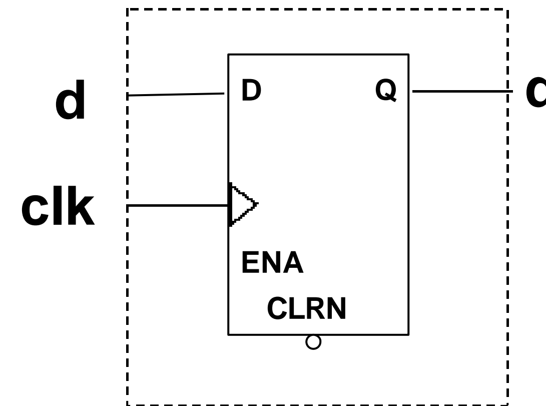
sensitivity list includes both inputs

What happens if gate = '0'?

⇨ ***Implicit Memory***

DFF - clk='1'

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

ENTITY dff IS
PORT ( d : in std_logic;
                        clk : in std_logic;
                        q : out std_logic
            );
END dff;

ARCHITECTURE behavior OF dff IS
BEGIN
PROCESS (clk)
            BEGIN
            IF clk = '1' THEN
                        q <= d;
            END IF;
END PROCESS;
END behavior;
```

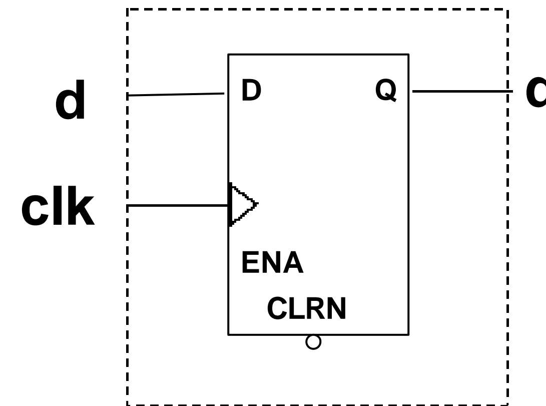


*sensitivity list only includes the triggering signal, in this case, **clk***

***clk = '1'** means that it is positive-edge triggered*

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;
```



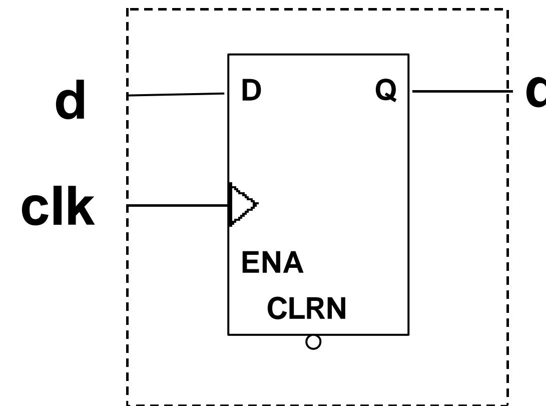
Note: There is no sensitivity list

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 OFdff_a IS
BEGIN
PROCESS (clk)
            BEGIN
            IF clk'event and clk = '1' THEN
                        q <= d;
            END IF;
END PROCESS;
END behavior;
```

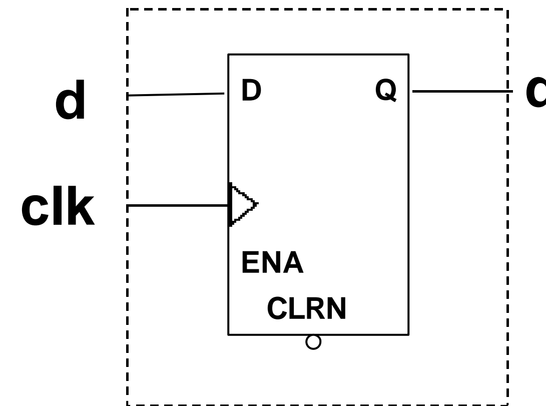


clk’event* and *clk=‘1’

- ***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 OFdff_b IS
BEGIN
PROCESS(clk)
          BEGIN
          IF rising_edge(clk) THEN
                    q <= d;
          END IF;
END PROCESS;
END behavior;
```

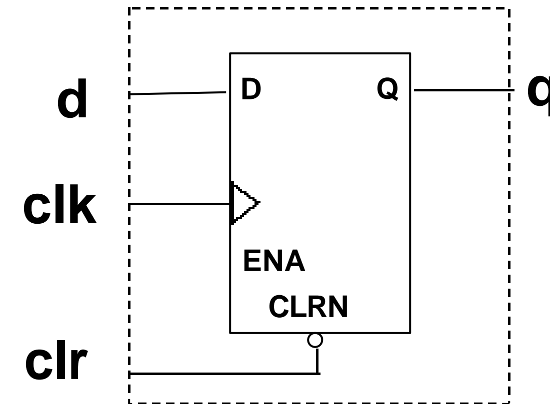


rising_edge

- *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;
```



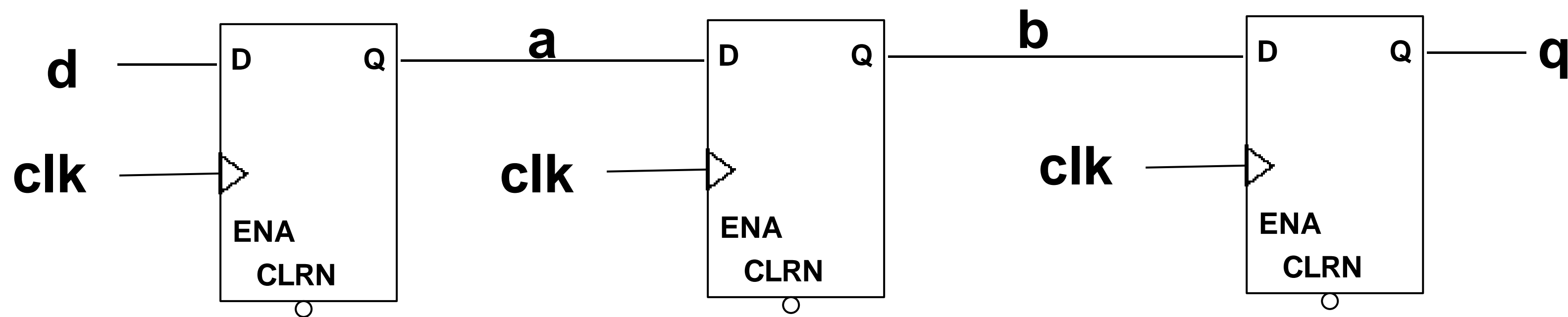
- *This is how to implement asynchronous control signals for the register*
- *Note: This IF-THEN statement is outside the IF-THEN statement that checks the condition **rising_edge***
- *Therefore, **clr='1'** does not depend on the clock*

How Many Registers?

```
ENTITY reg1 IS
    PORT ( d     : in BIT;
           clk   : in BIT;
           q     : out BIT);
END reg1;

ARCHITECTURE reg1 OF reg1 IS
SIGNAL a, b : BIT;
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.



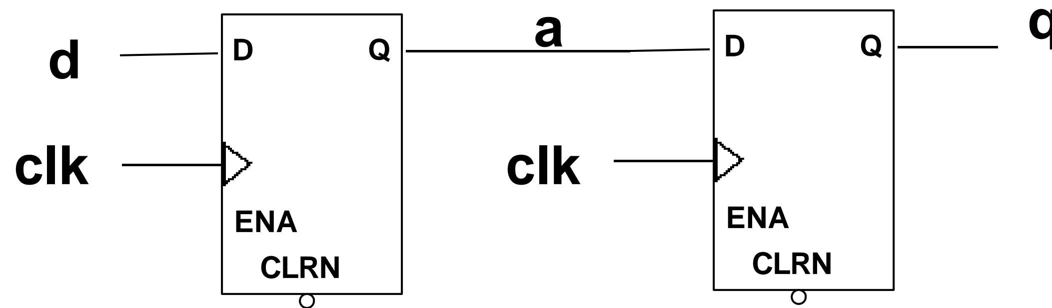
How Many Registers?

```
ENTITY reg1 IS
    PORT ( d      : in BIT;
           clk    : in BIT;
           q      : out BIT);
END reg1;

ARCHITECTURE reg1 OF reg1 IS
SIGNAL a, b : BIT;
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

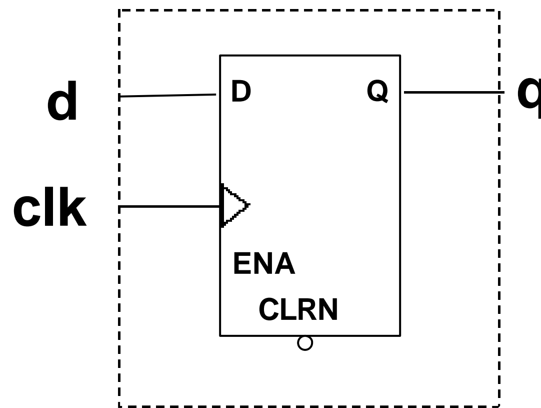


How Many Registers?

```
ENTITY reg1 IS
      PORT ( d      : in BIT;
                 clk    : in BIT;
                 q      : out BIT);
END reg1;

ARCHITECTURE reg1 OF reg1 IS
BEGIN
      PROCESS (clk)
      VARIABLE a, b : BIT;
      BEGIN
            IF rising_edge(clk) THEN
                 a := d;
                 b := a;
                 q <= b;
            END IF;
      END PROCESS;
END reg1;
```

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, will not infer registers.
- 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
                        q <= 0;
            ELSIF rising_edge(clk) THEN
                        IF updn = '1' THEN
                                    tmp_q := tmp_q + 1;
                        ELSE
                                    tmp_q := tmp_q - 1;
                        END IF;
             q <= tmp_q;
            END IF;
END PROCESS;
END logic;
```

- Counters are accumulators that always add a '1' or subtract a '1'

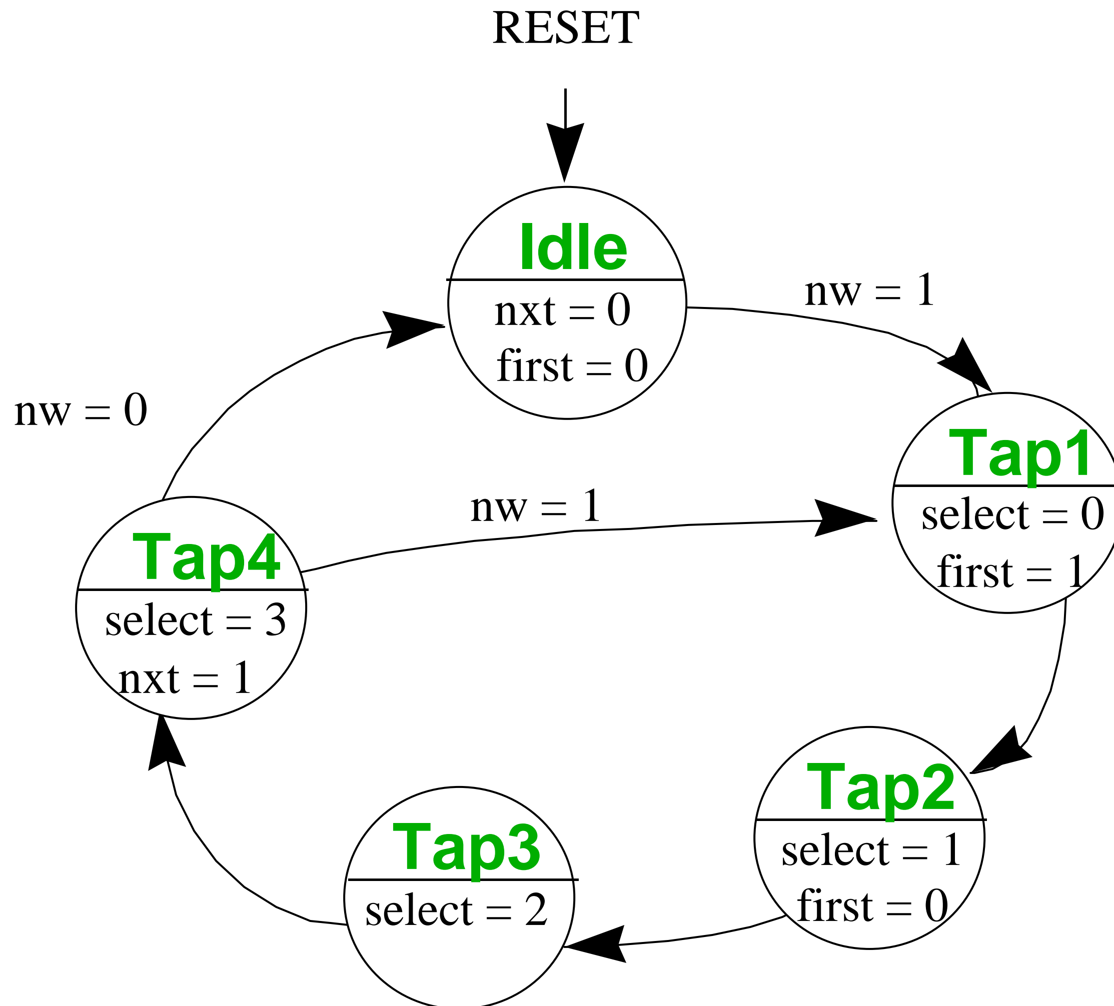
Arithmetic expression assigned to a variable

Variable assigned to a Signal inside the IF-THEN statement, that checks the clock condition, will infer registers

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

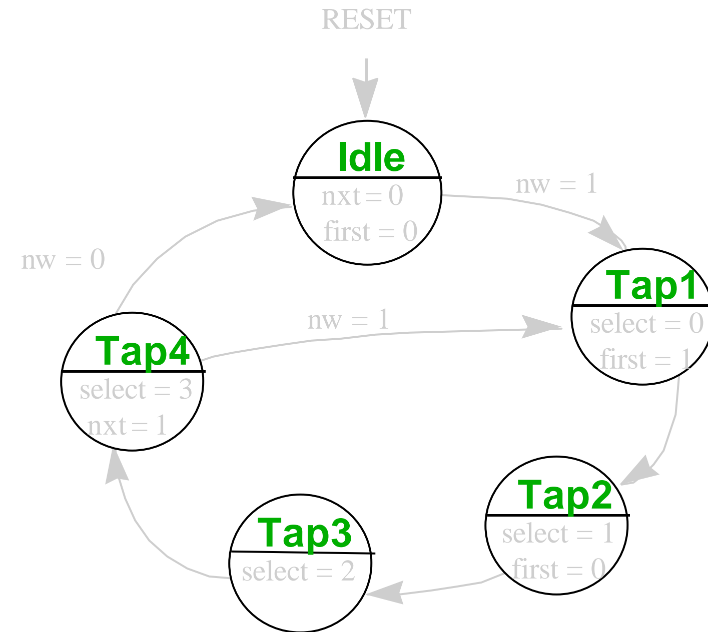
- To determine next state transition/logic:
  - Use a **CASE** statement inside IF-THEN statement that checks for the clock condition
    - Remember: State machines are implemented using registers

- 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

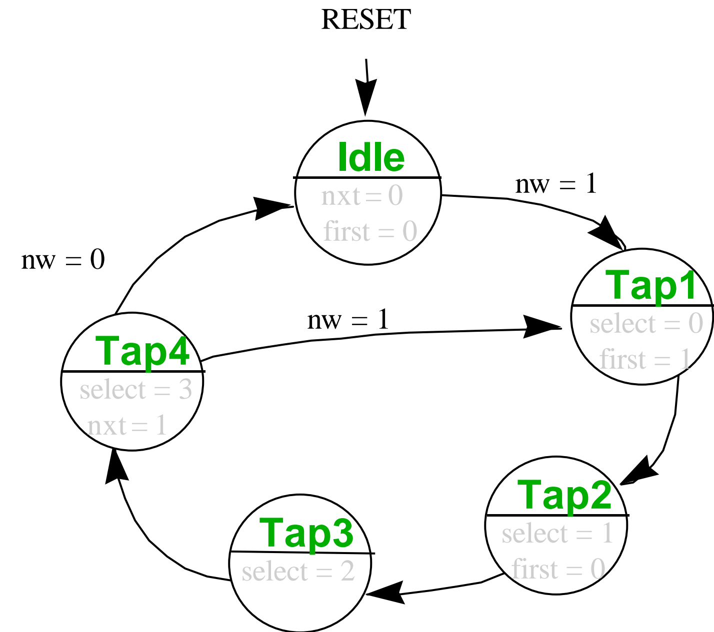
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;

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



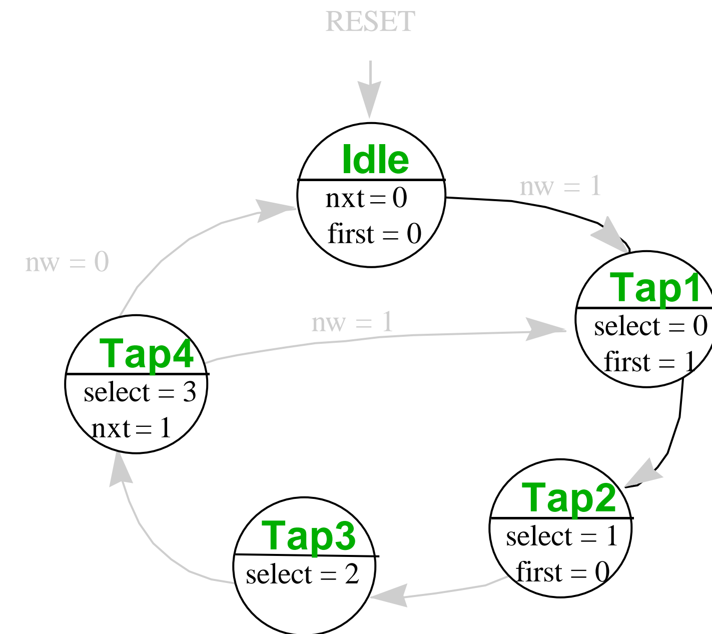
FSM VHDL Code - Next State Logic

```
BEGIN
PROCES (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;
```

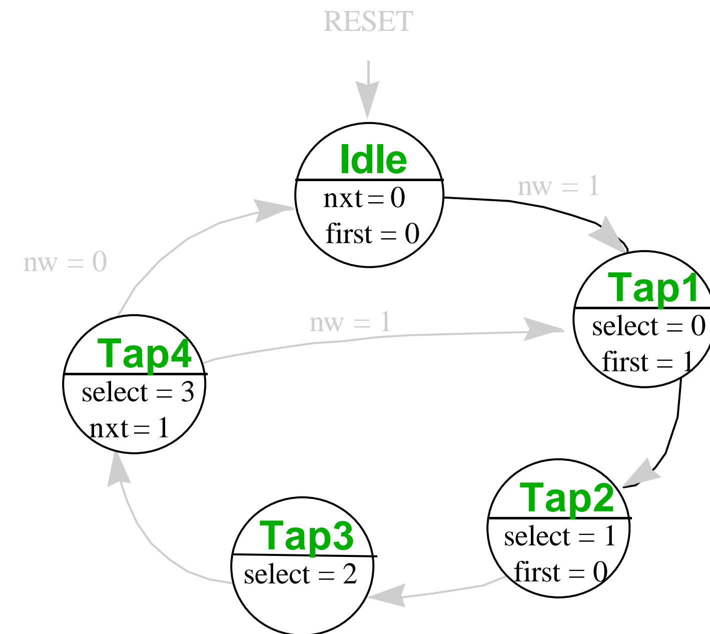


conditional signal assignments

selected signal assignments

FSM VHDL Code - Outputs using a Case

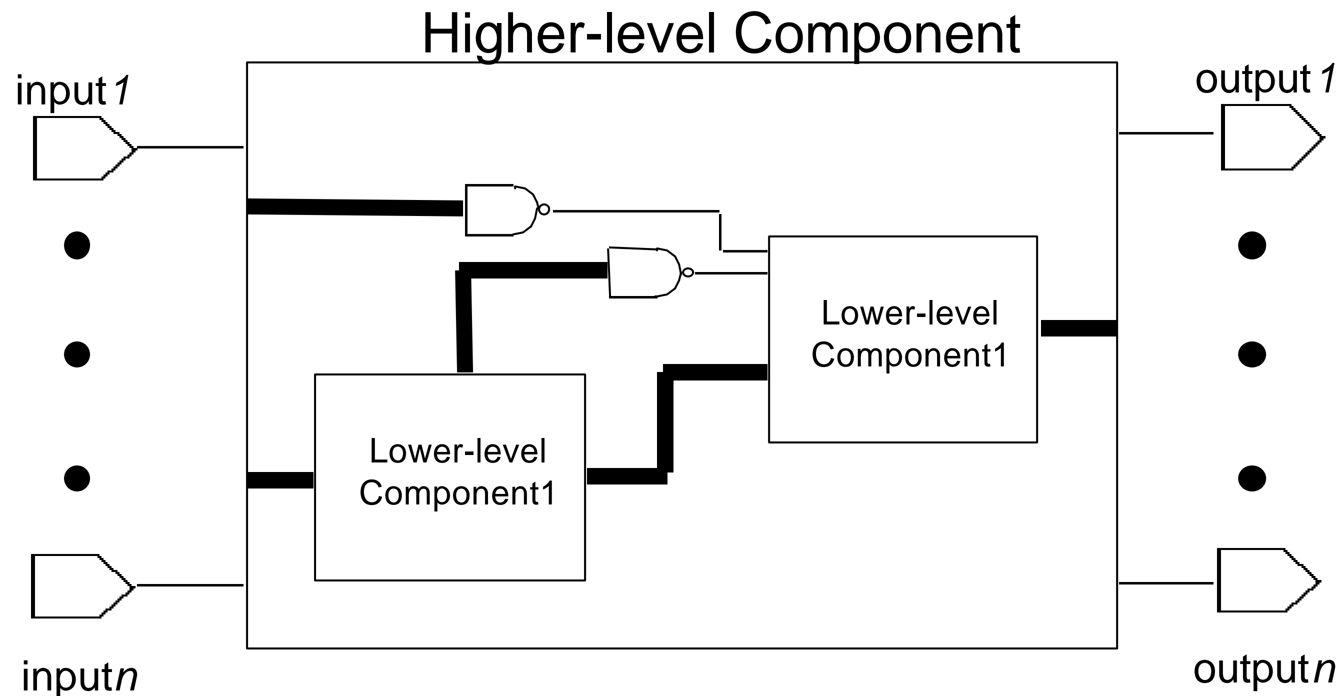
```
output: PROCESS(filter)
        BEGIN
        CASE  filter IS
                WHEN idle =>
                        nxt <= '0';
                        first <= '0';
                WHEN tap1 =>
                        sel <= "00";
                        first <= '1';
                WHEN tap2 =>
                        sel <= "01";
                        first <= '0';
                WHEN tap3 =>
                        sel <= "10";
                WHEN tap4 =>
                        sel <= "11";
                        nxt <= '1';
                END CASE;
END PROCESS output;

END logic;
```



Designing Hierarchically

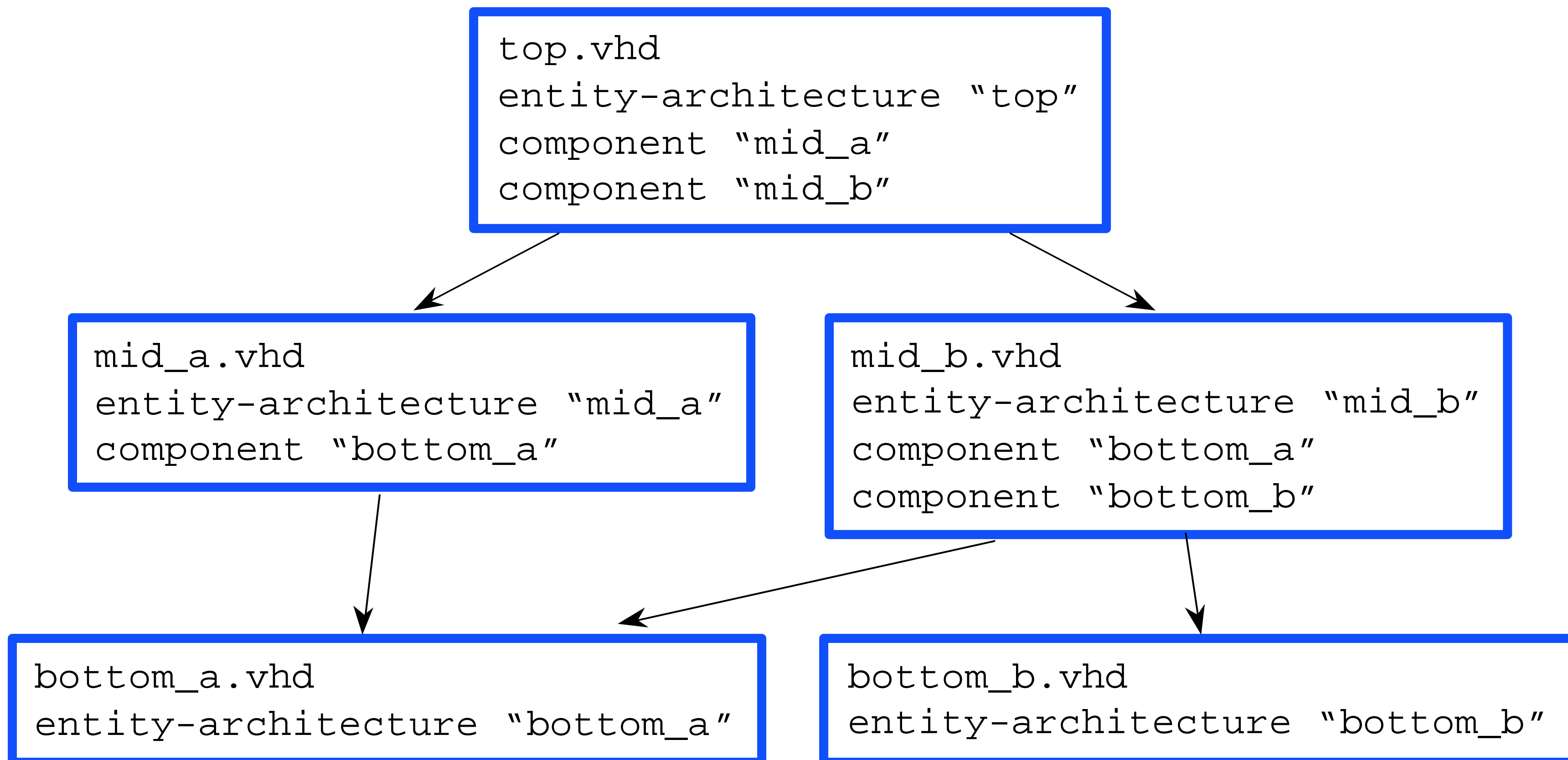
Recall - Structural Modeling

- 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



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

- Next-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

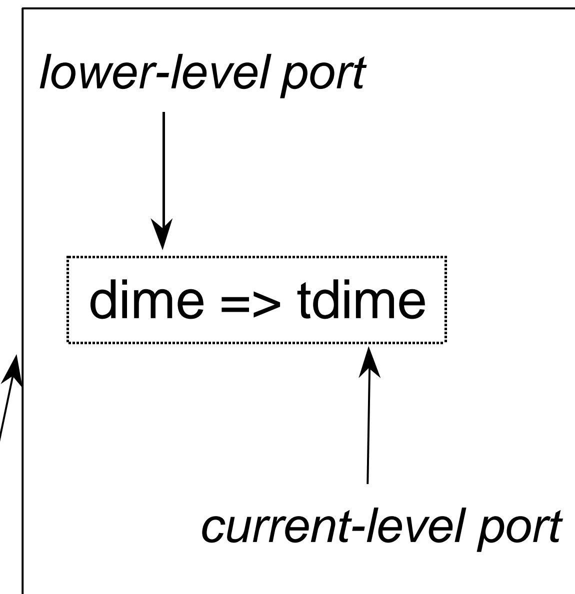
Positional Association

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;
```



Benefits of Hierarchical Designing

Designing Hierarchically

- In a design group, each designer can create seperate functions (components) in separate design files.
- These components can be shared by other designers or can be used for future projects.
- Therefore, designing hierarchically can make designs more modular and portable
- Designing Hierarchically can also allow easier and faster alternative implementations
  - Example: Try different counter implementations by replacing component declaration and component instantiation

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 complete library of LPM-compliant macrofunctions, plus other primitives

Library Altera/LPM

- **LIBRARY** ALTERA **;**
  - Contains the following packages:
    - **maxplus2** (Component declarations for all primitives and megafunction Altera libraries)
    - **megacore** (Component declarations for all Altera Megacores)

- **LIBRARY** LPM**;**
  - Contains the following packages:
    - **lpm_components** (Component Declarations for all Altera LPM functions)

⇨ Note: See MAX+PLUS II online help for more information

LPMs

- **L**ibrary of **P**arametrized **M**odules
  - Large building blocks that are easily configurable by:
    - Using different ***Ports***
    - Setting different ***Parameters***
- Industry standard:
  - Port names
  - Parameters
- However, the source code is different for each vendor.
- 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;**

- In the VHDL Code:

```
LIBRARY lpm;
USE lpm.lpm_components.all;
```

LPM Instantiation - lpm_mux

• MAX+plus II On-line 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;
```

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 different ***Ports*** and/or ***Parameters***
- Consistent synthesis

Appendix

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>;
```

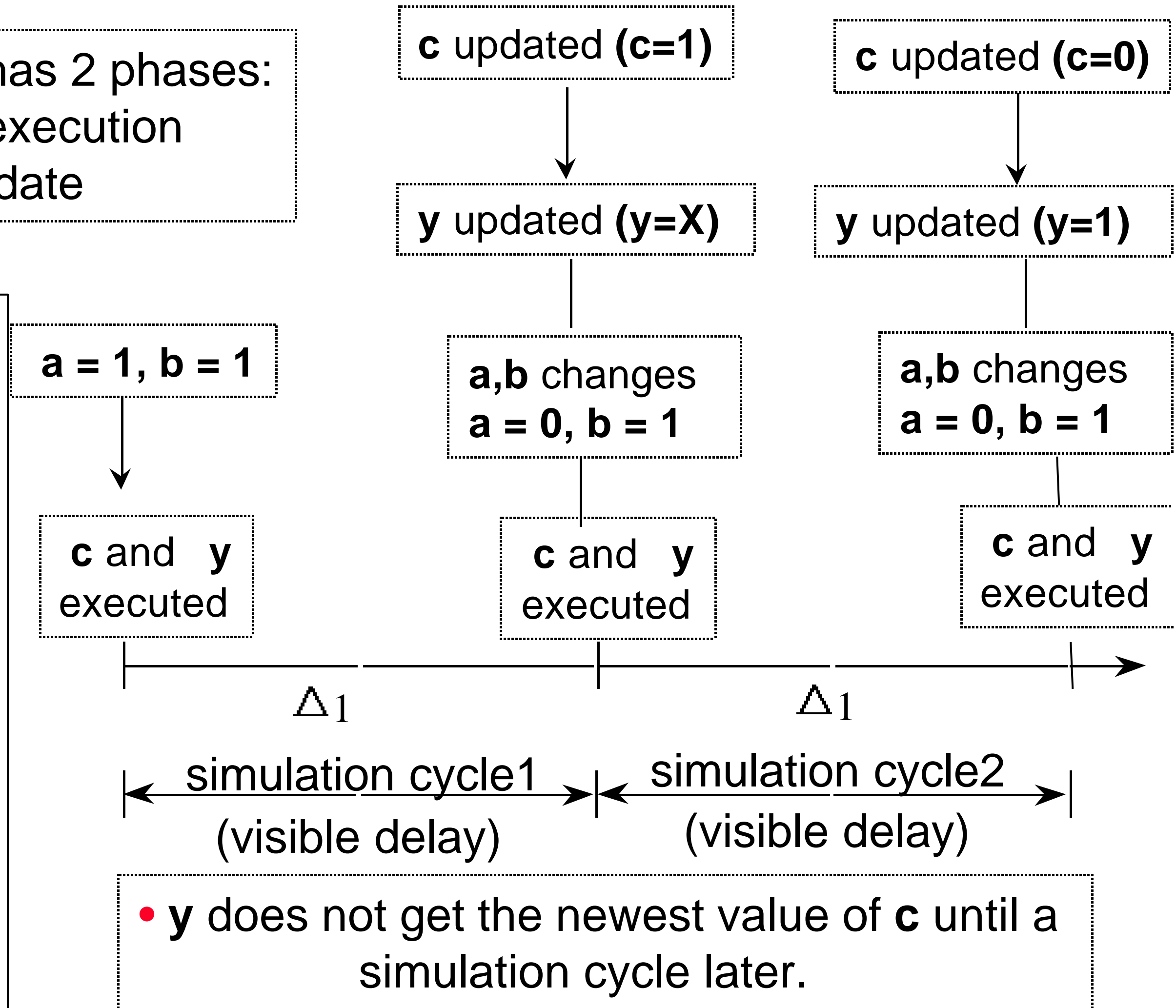
PROCEDURES

- 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

- ΔDelta 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;
```

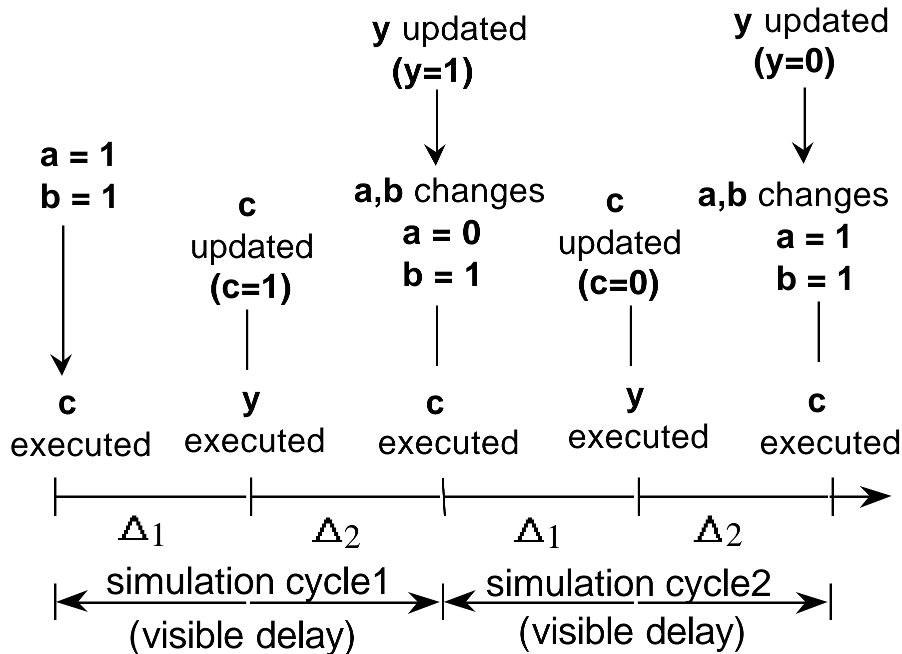


- **y** does not get the newest value of **c** until a simulation cycle later.

- ΔDelta cycle is non-visible delay (very small, close to zero)

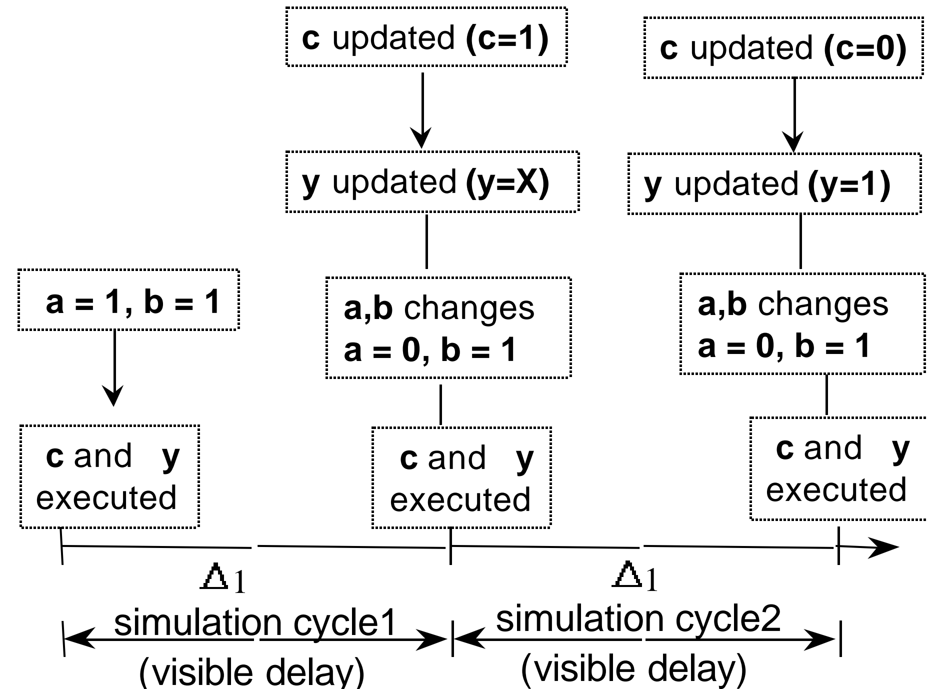
2 Process vs. 1 Process

```
process1: PROCESS(a, b)
          BEGIN
                    c <= a and b;
          END PROCESS process1;
process2: PROCESS(c)
          BEGIN
                    y <= c;
          END PROCESS process2;
```



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

```
PROCESS(a, b)
          BEGIN
          c <= a and b;
          y <= c;
END PROCESS;
```

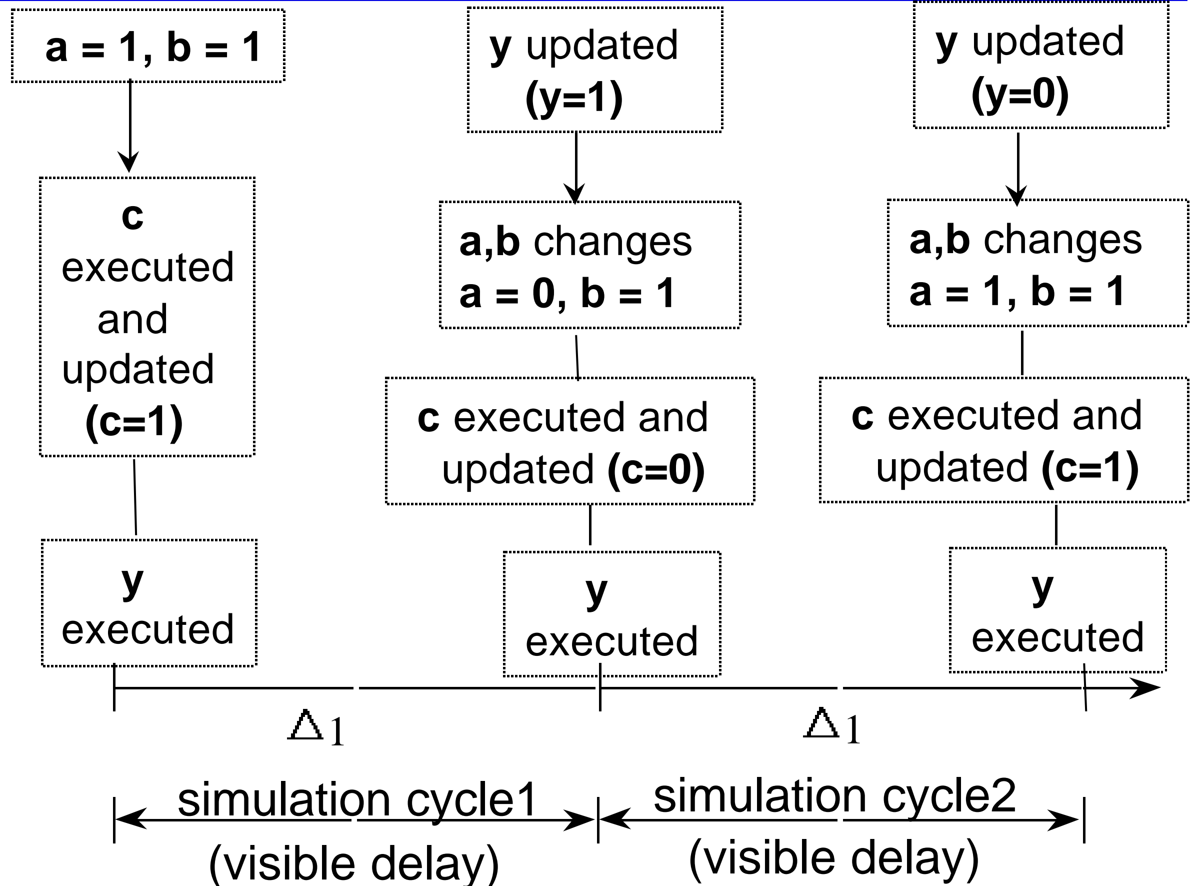


- **y** does not get the newest value of **c** until a simulation cycle later.

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;
```

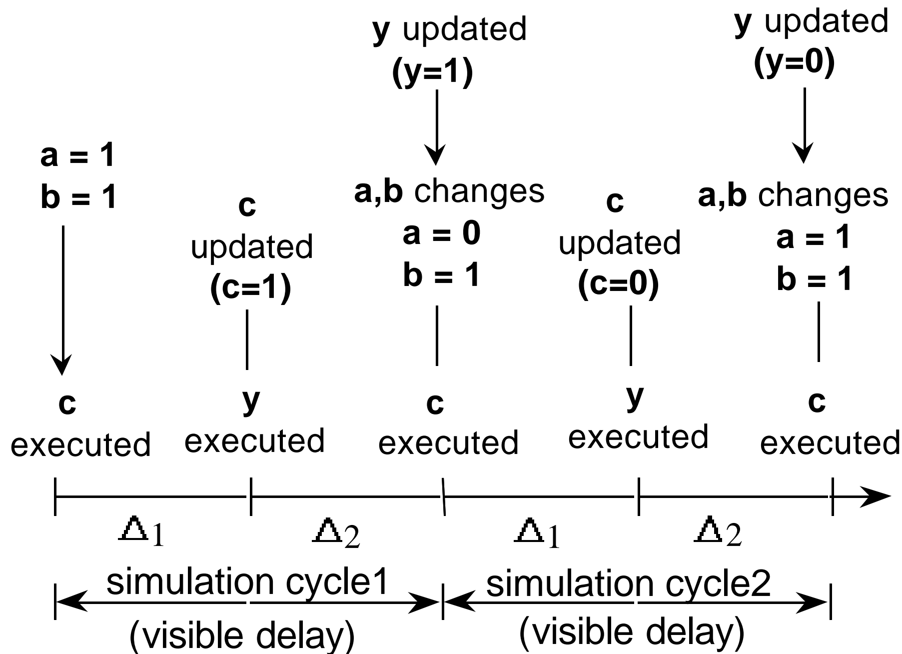


- **c** and **y** gets executed and updated within the same simulation cycle (at the end of the process)

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

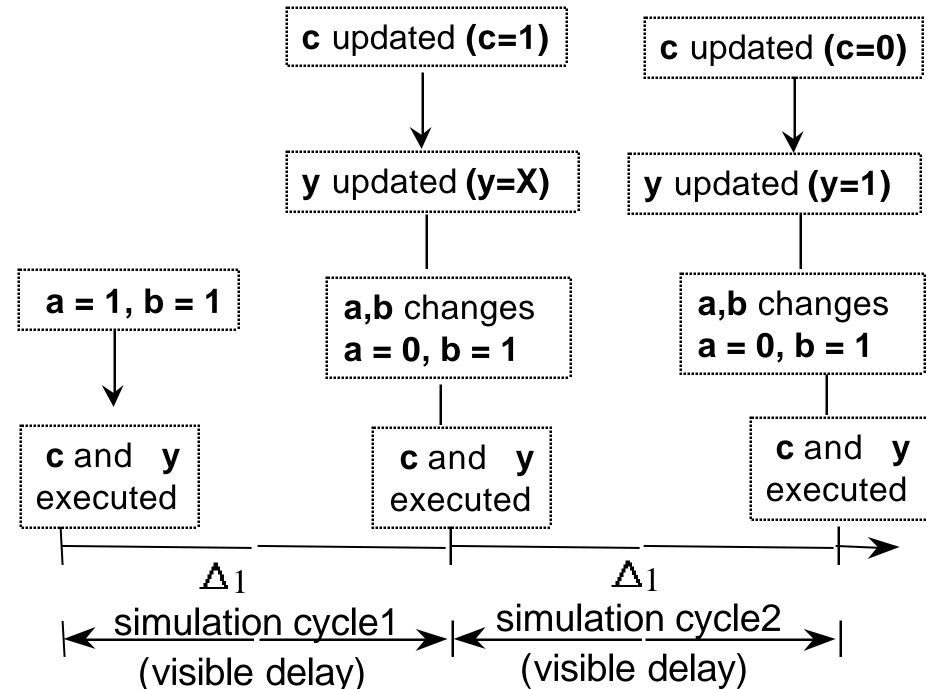
2 Process vs. 1 Process

```
process1: PROCESS(a, b)
    BEGIN
        c <= a and b;
    END PROCESS process1;
process2: PROCESS(c)
    BEGIN
        y <= c;
    END PROCESS process2;
```



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

```
PROCESS(a, b)
    BEGIN
    c <= a and b;
    y <= c;
END PROCESS;
```



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