Introduction to VHDL

Course Objectives

- Implement basic constructs of VHDL
- Implement modeling structures of VHDL

Course Outline

- Introduction to Altera devices and design software
- VHDL basics
  - Overview of language
- Design units
  - **ENTITY**
  - **ARCHITECTURE**
  - **CONFIGURATION**
  - **PACKAGE**
  - **LIBRARY**
- **ARCHITECTURE** modeling fundamentals
  - Signals
  - Processes

Course Outline (cont.)

- Understanding VHDL and logic synthesis
  - Process statement
  - Inferring logic
- Model application
  - State machine coding
- Hierarchical design
  - Overview
  - Structural modeling
  - Application of Library of Parameterized Modules (LPM)

Introduction to VHDL

Introduction to Altera Devices & Design Software

A Complete Solutions Portfolio

Nios II

QUARTUS II

Embedded soft processors

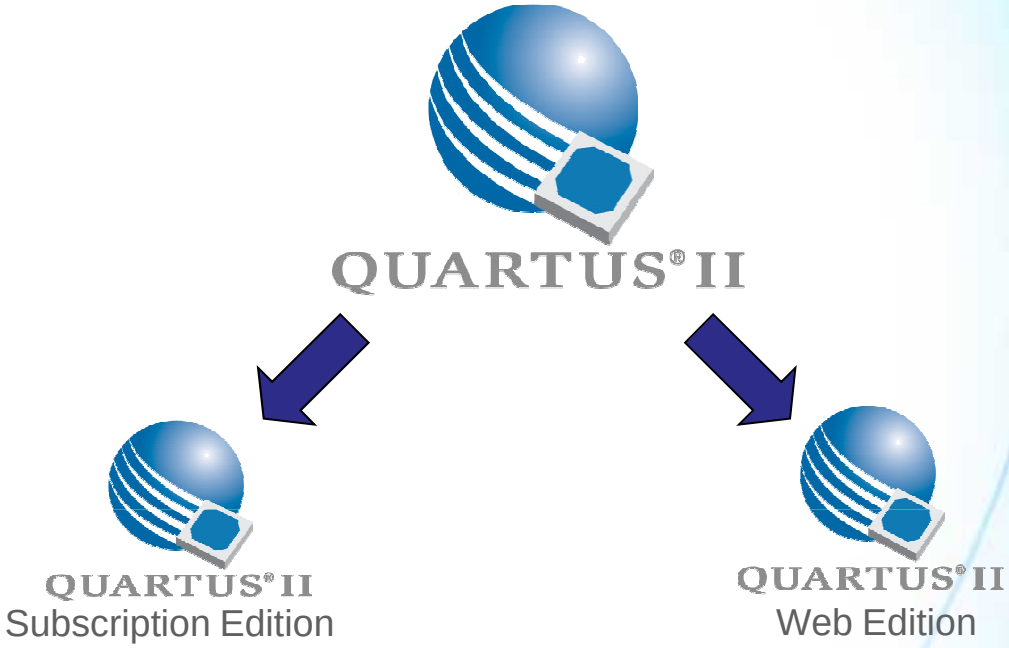
Intellectual Property (IP)

Design software

Development kits

ALTERA

Quartus II Software – Two Editions



	Subscription Edition	Web Edition
Devices Supported	**All**	**Selected Devices**
Features	**100%**	**95%**
Distribution	**Internet & DVD**	**Internet & DVD**
Price	**Paid**	**Free**

Feature Comparison available on Altera web site

Introduction to VHDL

VHDL Basics

VHDL

VHSIC (Very High Speed Integrated Circuit)

Hardware

Description

Language

What is VHDL?

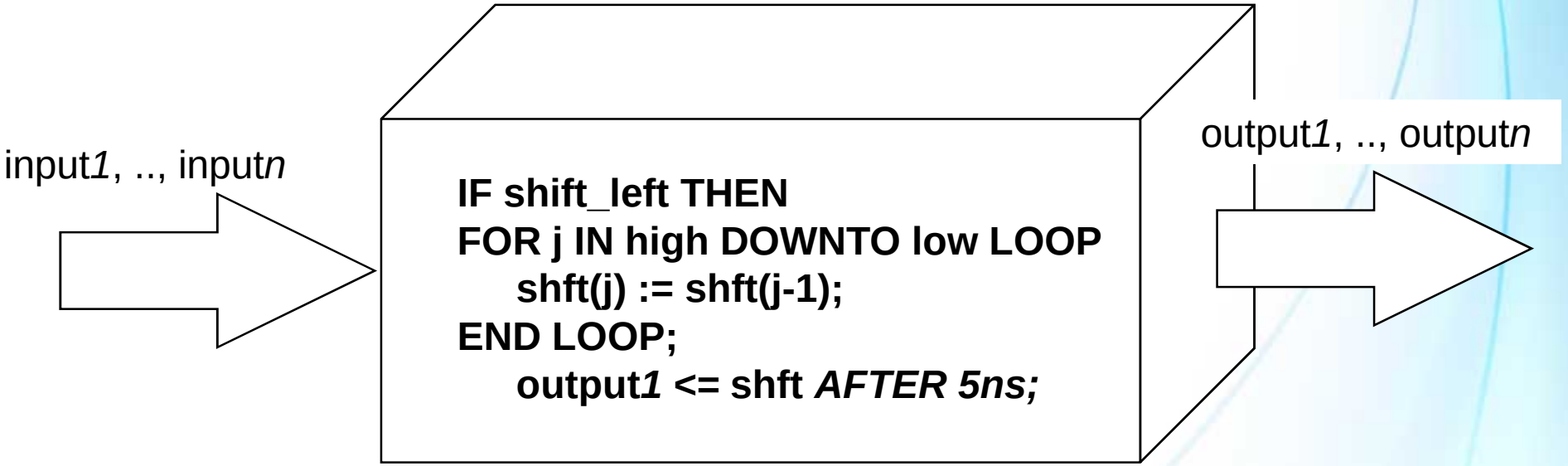
- IEEE industry standard hardware description language
- High-level description language for both simulation & synthesis

Terminology

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

Behavioral Modeling

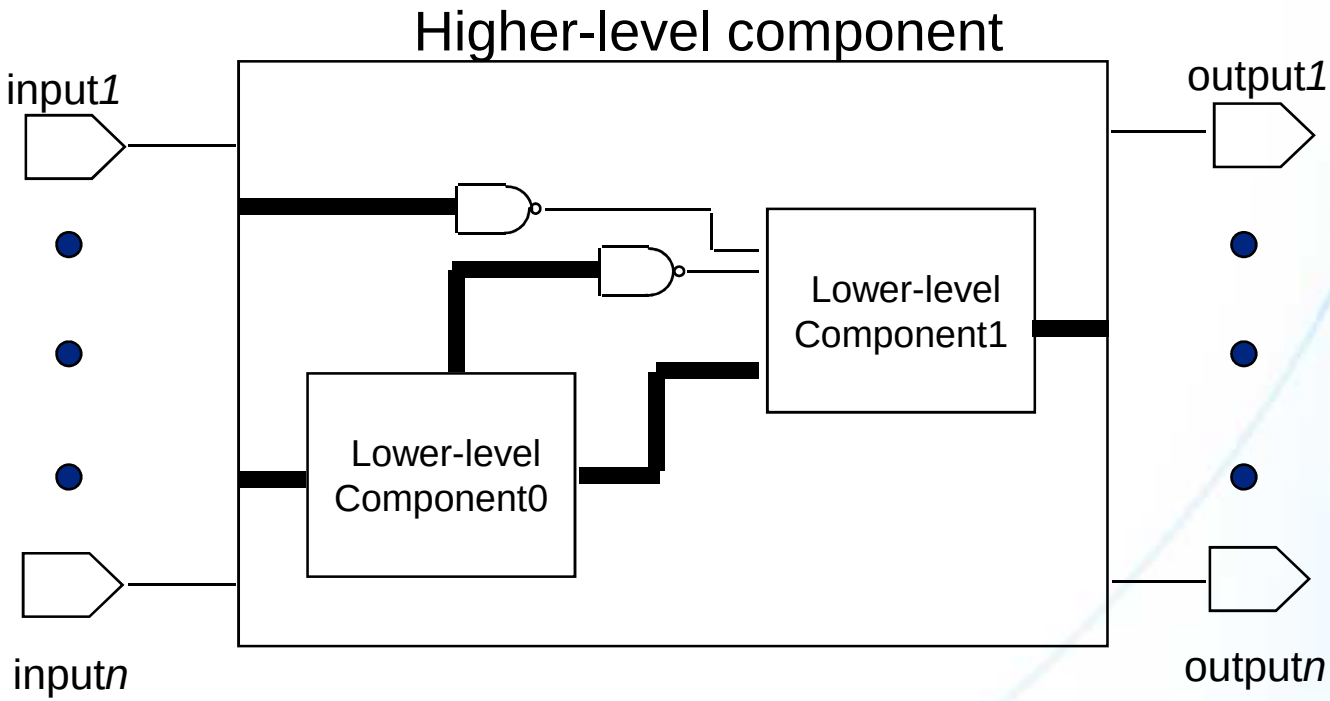
- Only the functionality of the circuit, no structure
- No specific hardware intent



Left bit shifter

Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware



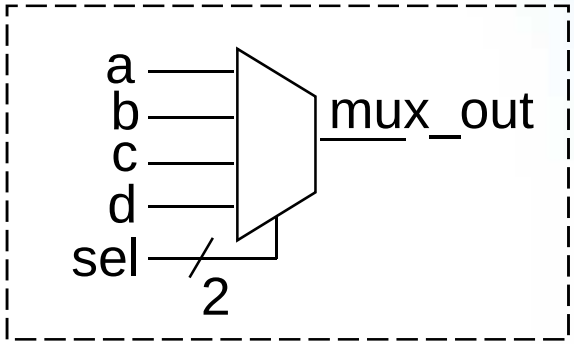
Terminology (cont.)

- 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
- Process – Basic unit of execution in VHDL
  - Process executions are converted to equivalent hardware

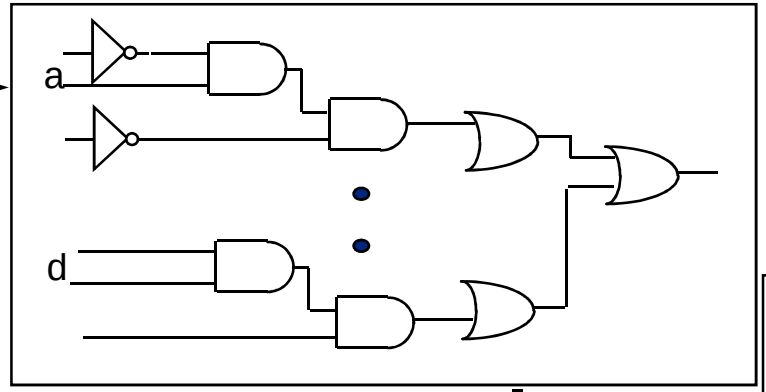
RTL Synthesis

```
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;
```

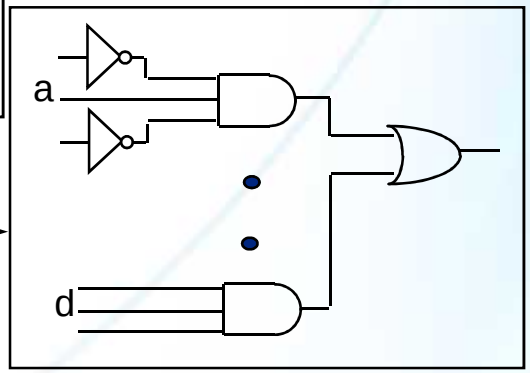
inferred



Translation



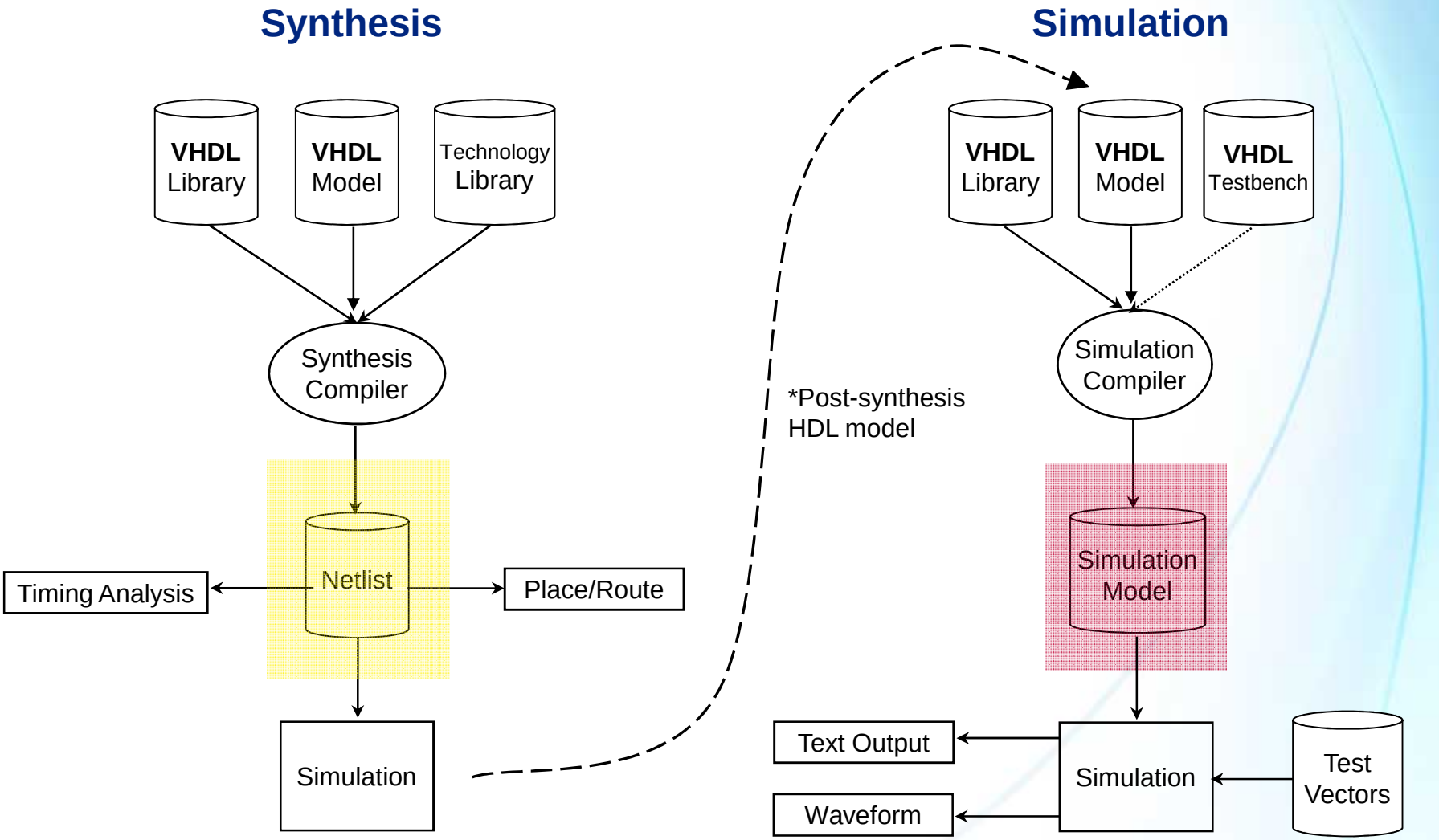
Optimization



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 RTL Synthesis & RTL Simulation Flows



VHDL Basics

- Two sets of constructs:
  - Simulation
  - 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
- Comments in VHDL begin with "**--**" to EOL

Introduction to VHDL

VHDL Design Units

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**
  - Consists of two parts: **PACKAGE** declaration and **PACKAGE** body

ENTITY Declaration

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

- Analogy : symbol
  - <entity_name> can be any alpha/numerical name
- Port declarations
  - Used to describe the inputs and outputs i.e. pins
- Generic declarations
  - Used to pass information into a model
- Close entity in one of 3 ways
  - **END ENTITY <entity_name>;** -- VHDL '93 and later
  - **END ENTITY;** -- VHDL '93 and later
  - **END;** -- All VHDL versions

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 <entity_name> ;
```

- Structure : <class> object_name **:** <mode> <type> **;**
  - <class> : what can be done to an object
  - object_name : identifier (name) used to refer to object
  - <mode> : directional
    - **IN** (input)
    - **OUT** (output)
    - **INOUT** (bidirectional)
    - **BUFFER** (output w/ internal feedback)
  - <Type> : what can be contained in the object (discussed later)

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 <entity_name> ;  ( 1076-1993 version)
```

- Generic values can be overwritten during compilation
  - i.e. Passing in parameter information
- Generic must resolve to a constant during compilation

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 two styles
- End architecture with
  - **END ARCHITECTURE <architecture_name>;** -- VHDL '93 & later
  - **END ARCHITECTURE;** -- VHDL '93 & later
  - **END;** -- All VHDL versions

ARCHITECTURE

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

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

SIGNAL temp **:** INTEGER **:=** 1**;** -- signal declarations with optional default values
CONSTANT load **:** boolean **:=** true**;** --constant declarations
--Type declarations (discussed later)
--Component declarations (discussed later)
--Subprogram declarations (discussed later)
--Subprogram body (discussed later)
--Subtype declarations
--Attribute declarations
--Attribute specifications

BEGIN

PROCESS statements
Concurrent procedural calls
Concurrent signal assignment
Component instantiation statements
Generate statements

END ARCHITECTURE <architecture_identifier>**;**

VHDL - Basic Modeling Structure

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

ARCHITECTURE *arch_name* **OF** *entity_name* **IS**
    internal signal declarations
    enumerated data type declarations
    component declarations
BEGIN
    signal assignment statements
    PROCESS statements
    component instantiations
END ARCHITECTURE *arch_name;*

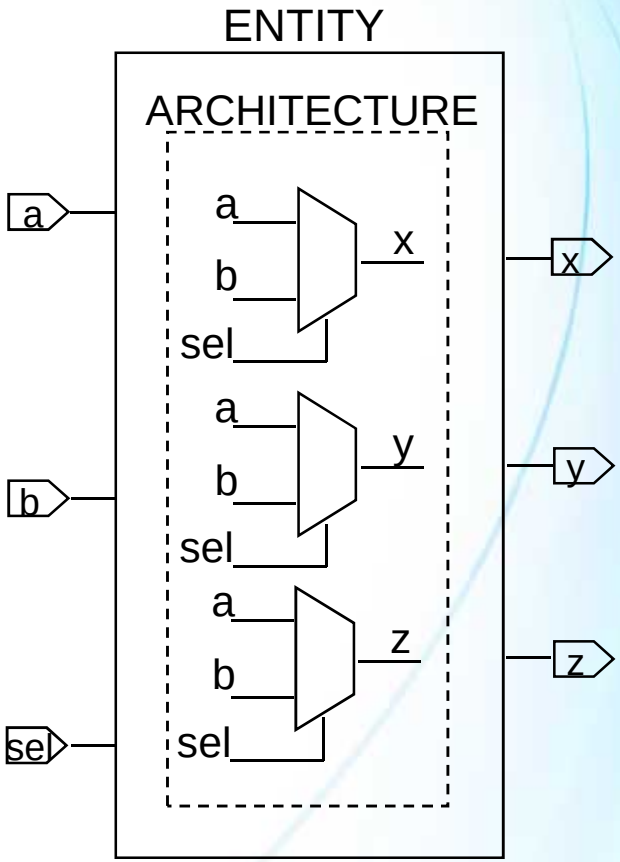
Configuration

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

- Makes associations within models
  - Associate an entity and architecture
  - Associate a component to an entity-architecture
- Uses
  - In simulation environments to execute different sets of vector stimulus
  - To provide flexible and fast path to design alternatives
    - e.g. Behavioral vs. synthesizable model; FPGA vs. ASIC model

Putting It All Together

```
ENTITY cmpl_sig IS
    PORT (
        a, b, sel       : IN    BIT;
        x, y, z         : OUT BIT
    );
END ENTITY 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 ARCHITECTURE logic;

CONFIGURATION cmpl_sig_conf OF cmpl_sig IS
    FOR logic
    END FOR;
END CONFIGURATION 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 <package_name> **;** (1076-1993)

PACKAGE BODY <package_name> **IS**
    Constant declarations
    Type declarations
    Subprogram body
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 PACKAGE 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 PACKAGE BODY filt_cmp;
```

Package declaration

Package body

Libraries

- A **LIBRARY** is a directory that contains a package or a collection of packages
- Two types of libraries
  - Working library
    - Current project directory
  - Resource libraries
    - **STANDARD** package
    - **IEEE** developed packages
    - Altera component packages
    - Any **LIBRARY** of design units that is referenced in a design

Model Referencing of LIBRARY and PACKAGE

- All packages must be compiled before being referenced
- Implicit libraries
  - **WORK**
  - **STD**
  - ⇨ Note: items in these packages do not need to be referenced, they are implied
- Referencing a package requires 2 clauses
  - **LIBRARY** clause
    - Defines the library name that can be referenced
    - Is a symbolic name to path/directory
    - Defined in 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;
USE WORK.filt_cmp.ALL;

ENTITY cmpl_sig IS
PORT ( a, b, sel   :   IN    STD_LOGIC;
     x, y, z :  OUT STD_LOGIC);
END ENTITY 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 ARCHITECTURE logic;
```

- **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 for object name
- Placing the library/use clause first will allow all following design units to access it

Libraries

- **LIBRARY STD;**
  - Contains the following packages
    - **STANDARD**
      - Pre-defined data types
      - Operator functions to support pre-defined data types
    - **TEXTIO**
      - File operations
      - Not discussed
  - An implicit library (built_in)
    - Does not need to be explicitly 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)
- Type **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

Other Types Defined in Standard Package

- Type **NATURAL**
  - Integer with range 0 to 2^{32}
- Type **POSITIVE**
  - Integer with range 1 to 2^{32}
- Type **CHARACTER**
  - ASCII characters
- Type **STRING**
  - Array of characters
- Type **TIME**
  - Value includes units of time (e.g. ps, us, ns, ms, sec, min, hr)
- Type **REAL**
  - Double-precision floating point numbers

Libraries (cont.)

- **LIBRARY IEEE**
  - Contains the following packages
    - **STD_LOGIC_1164** (**STD_LOGIC** types & related functions)
    - **NUMERIC_STD** (unsigned arithmetic functions using standard logic vectors defined as SIGNED and UNSIGNED data type)
    - **STD_LOGIC_ARITH*** (arithmetic functions using standard logic vectors as SIGNED or UNSIGNED)
    - **STD_LOGIC_SIGNED*** (signed arithmetic functions directly using standard logic vectors)
    - **STD_LOGIC_UNSIGNED*** (unsigned arithmetic functions directly using standard logic vectors)
    - **STD_LOGIC_TEXTIO*** (file operations using std_logic)

** Packages actually developed by Synopsys but accepted and supported as standard VHDL by most synthesis and simulation tools*

Types Defined in STD_LOGIC_1164 Package

- Type **STD_LOGIC**
  - 9 logic value system (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’)
    - ‘1’: Logic high
    - ‘0’: Logic low
    - ‘X: Unknown
    - ‘Z’: (not ‘z’) Tri-state
    - ‘-’: Don’t Care
    - ‘H’: Weak logic high
    - ‘L’: Weak logic low
    - ‘W’: Weak unknown
  - Resolved type: supports signals with multiple drivers
    - Driving multiple values onto same signal results in known value
- Type **STD_ULOGIC**
  - Same 9 value system as STD_LOGIC
  - Unresolved type: Does not support multiple signal drivers
    - Driving multiple values onto same signal results in error

User-Defined Libraries/Packages

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

  LIBRARY WORK; *--optional*
  USE WORK.<*Package name*>**.ALL**;

- User-defined packages in a different directory
  - Library name must be mapped to directory path via tool settings

  LIBRARY <*any_name*>**;**

  USE <*any_name*>.<*Package_name*>**.ALL;**

Introduction to VHDL

ARCHITECTURE Modeling Fundamentals

Architecture Modeling Fundamentals

- Constants
- Signals
- Signal Assignments
- Operators
- Processes
- Variables
- Sequential Statements
- Subprograms
- Types

VHDL Objects & Object Classes

- Objects
  - Used to describe model functionality
  - Assigned a value and a type
- Object classes
  - Define objects behavior and what operations can be performed with/to object
  - Four types
    - CONSTANT
    - SIGNAL
    - VARIABLE
    - FILE
      - Not discussed

Constants

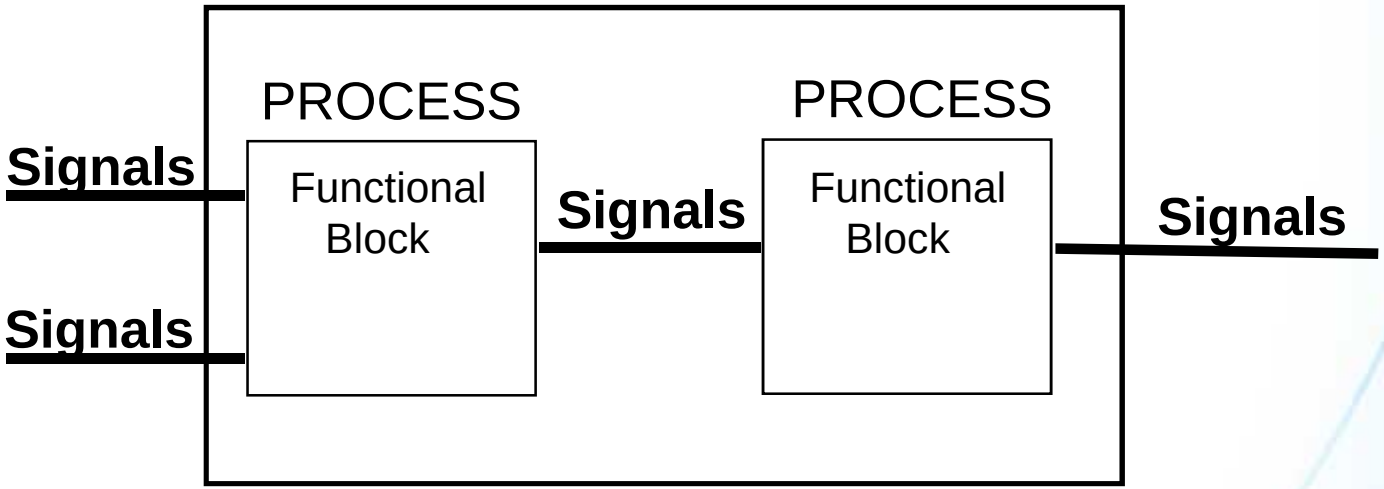
- Associates value to name
- Constant declaration
  - Can be declared in ENTITY, ARCHITECTURE or PACKAGE

```
CONSTANT <name> : <DATA_TYPE> := <value>;
CONSTANT bus_width : INTEGER := 16;
```

- Cannot be changed by executing code
  - Remember generics are constants (parameters) that can be overwritten by passing new values into the entity at compile time, not during code execution
- Improves code readability
- Increases code flexibility

Signals

- Signals represent physical interconnect (wire) that communicate between processes (functions)



- Signal declaration
  - Can be declared in PACKAGE, ENTITY and ARCHITECTURE

```
SIGNAL temp : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Assigning Values to Signals

```
SIGNAL temp : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

- Signal assignments are represented by **<=**
- Examples
  - All bits

    temp **<=** "10101010"**;**

    temp **<=** x"aa" **;** (1076-1993)
      - VHDL also supports 'o' for octal and 'b' for binary
  - Bit-slicing

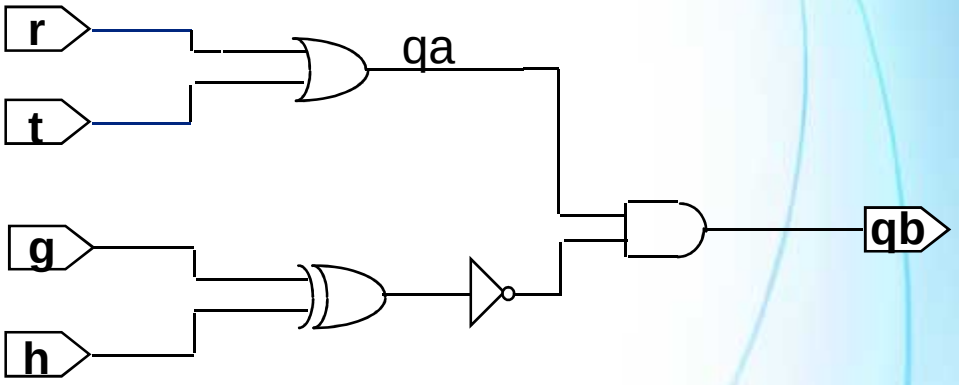
    temp (7 **DOWNTO** 4) **<=** "1010";
  - Single bit

    temp(7) **<=** '1';

- Use double-quotes (" ") to assign multi-bit values and single-quotes (' ') to assign single-bit values

Signal Used as 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 ENTITY simp;

ARCHITECTURE logic OF simp IS
    SIGNAL qa : STD_LOGIC;
BEGIN

    qa <= r OR t;
    qb <= (qa AND NOT (g XOR h));

END ARCHITECTURE logic;
```



Signal declaration inside architecture

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

Concurrent Signal Assignments

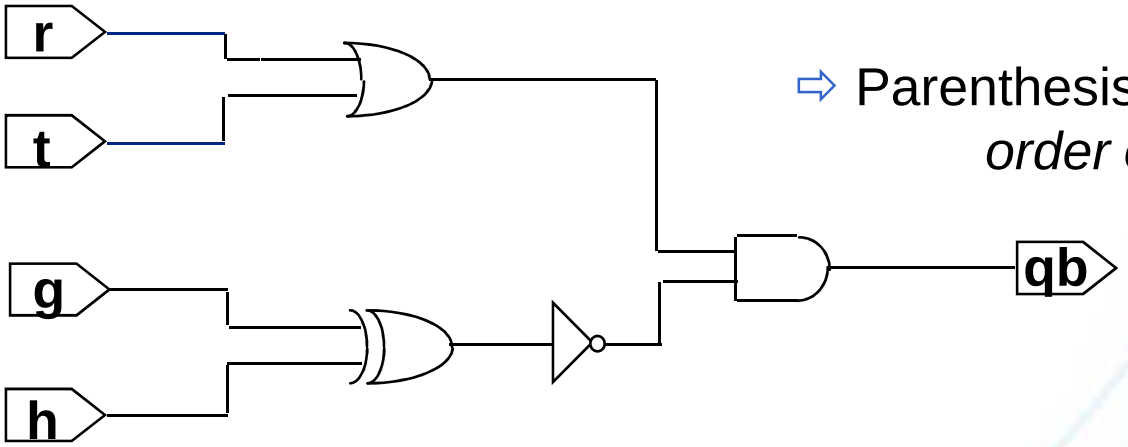
- Used to assign values to signals using expressions
- Represent ***implied*** processes that execute in parallel
  - Process sensitive to anything on read (right) side of assignment
- Three types
  - Simple signal assignment
  - Conditional signal assignment
  - Selected signal assignment

Simple Signal Assignments

- Format: `<signal_name> <= <expression>;`
- Example:

```
qa <=  r OR t ;
qb <=  (qa AND NOT (g XOR h));
```

→ *2 implied processes*



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

- Expressions use VHDL operators to describe behavior

VHDL Operators

Operator Type		Operator Name/Symbol	Priority
Logical		**NOT AND OR NAND NOR XOR XNOR(1)**	Low
Relational		**= /= < <= > >=**	
Shifting(1)(2)		**SLL SRL SLA SRA ROL ROR**	
Arithmetic	Addition & Sign	**+ -**	
	Concatenation	**&**	
	Multiplication	*** / MOD REM**	
Miscellaneous		**** ABS**	High

***** = exponentiation***
abs = absolute value

(1) Not supported in VHDL '87
(2) Supported in NUMERIC_STD package for SIGNED/UNSIGNED data types

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 ENTITY opr;

ARCHITECTURE example OF opr IS
BEGIN
    sum <= a + b;
END ARCHITECTURE 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

- VHDL defines arithmetic & boolean functions only for data types defined in STANDARD package
- 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 operator overloading functions can be found in the **LIBRARY IEEE**
  - **STD_LOGIC_ARITH** (arithmetic functions)
  - **STD_LOGIC_SIGNED** (signed arithmetic functions)
  - **STD_LOGIC_UNSIGNED** (unsigned arithmetic functions)
  - **NUMERIC_STD** (signed & unsigned arithmetic)
- For example, the package **STD_LOGIC_UNSIGNED** defines some of the following functions

```
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 ENTITY overload;

ARCHITECTURE example Of overload IS
BEGIN
    sum <= a + b;
END ARCHITECTURE 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 (again)

- Simple signal assignment
  - Already discussed
- Conditional signal assignment
- Selected signal assignment

Conditional Signal Assignments

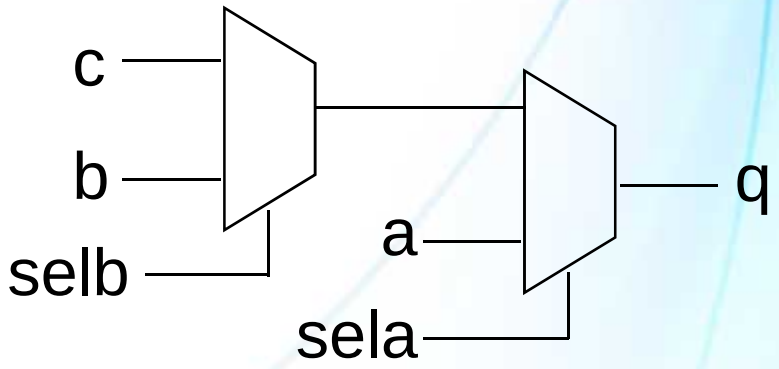
- Format:

```
<signal_name> <= <signal/value> WHEN <condition_1> ELSE
                 <signal/value> WHEN <condition_2> ELSE
                                ...
                 <signal/value> WHEN <condition_n> ELSE
                 <signal/value>;
```

- Example:

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

Implied process



Selected Signal Assignments

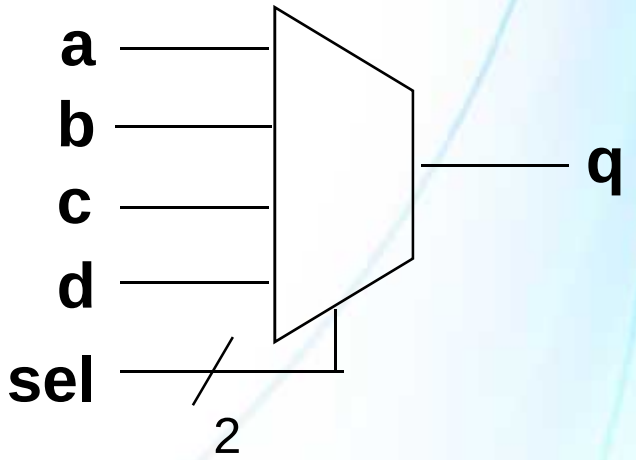
- Format:

```
WITH <expression> SELECT
    <signal_name> <= <signal/value> WHEN <condition_1>,
                    <signal/value> WHEN <condition_2>,
                    ...
                    <signal/value> WHEN OTHERS;
```

- Example:

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

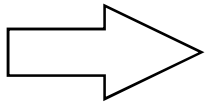
Implied process



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 ENTITY 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 ARCHITECTURE logic;
```

sel is of **STD_LOGIC** data type

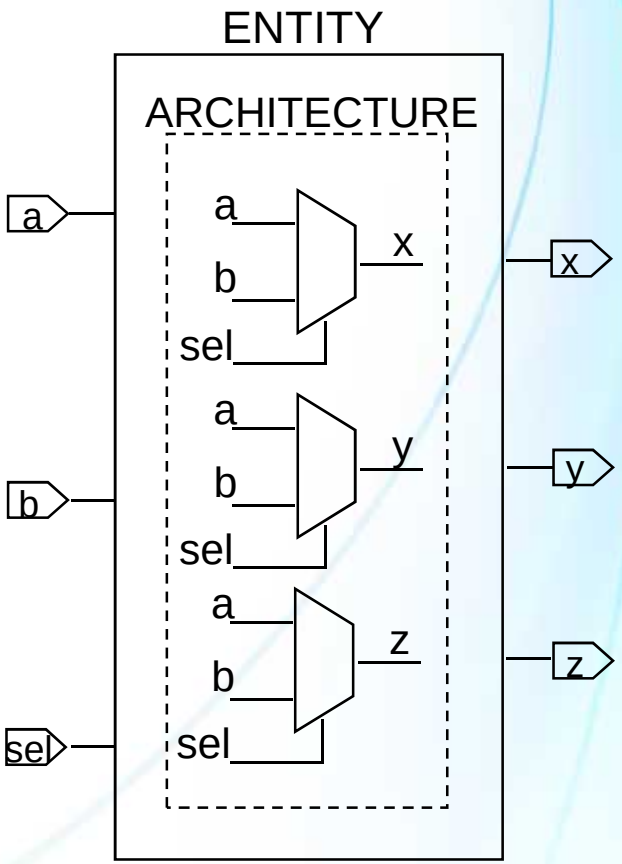
- *What are the values for a* **STD_LOGIC** *data type*
- *Answer:* **{'0','1','X','Z'…}**

- *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 ENTITY 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',
              'X' WHEN OTHERS;
END ARCHITECTURE logic;
```

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



Architecture Modeling Fundamentals

- Constants
- Signals
- Signal Assignments
- Operators
- Processes
- Variables
- Sequential Statements
- Subprograms
- Types

Process Statements

- Implicit process
  - Types
    - Concurrent signal assignments
    - Component instantiations
  - Process sensitive to all inputs
    - e.g. Read side of signal assignment
- Explicit process
  - PROCESS keyword defines process boundary

Explicit PROCESS Statement

- Process sensitive to explicit (optional) sensitivity list
  - Events (transitions) on signals in sensitivity list trigger process
    - 0→1, X→1, 1→Z, etc
- Declaration section allows declaration of local objects and names
- Process contents consist of sequential statements (discussed later) and simple signal assignments

```
--    Explicit PROCESS statement
label : PROCESS (sensitivity_list)
    Constant declarations
    Type declarations
    Variable declarations
BEGIN
    Sequential statement #1;
        …
    Sequential statement #n ;
END PROCESS;
```

Execution of PROCESS Statement

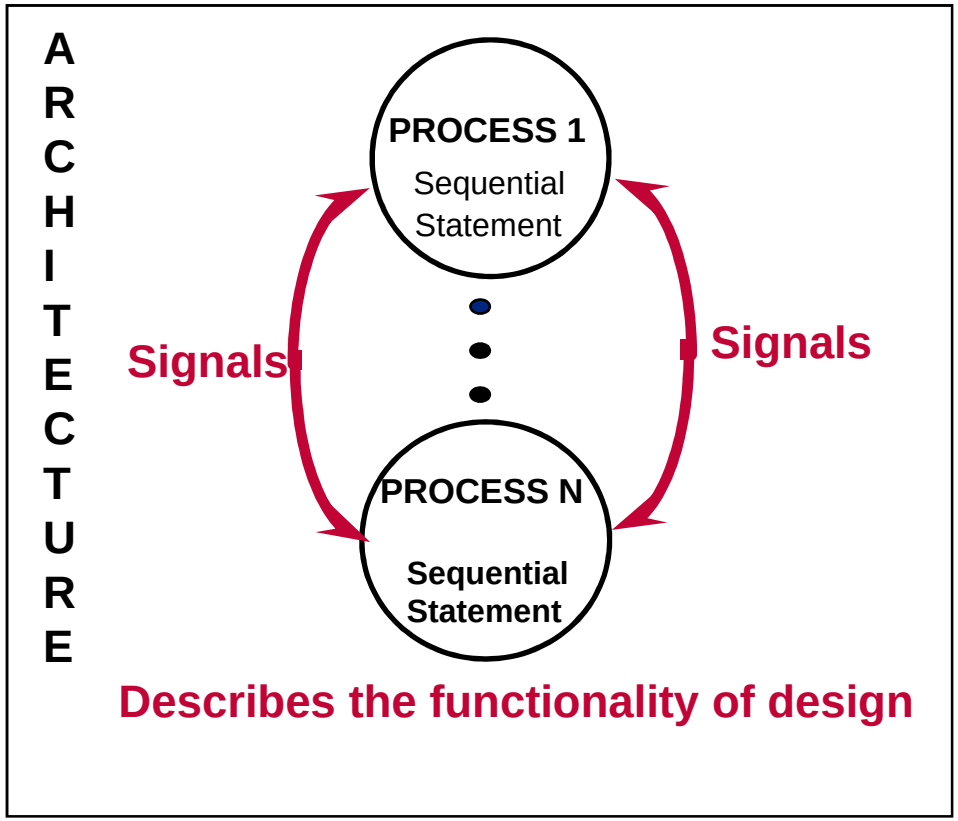
- Process statement is executed infinitely unless broken by a WAIT statement or sensitivity list
- Sensitivity list implies a WAIT statement at the end of the process
  - Due to all processes being executed once at beginning of model execution
- Process can have multiple WAIT statements
- Process can not have both a sensitivity list and WAIT statement

  ⇨ Note: Logic synthesis places restrictions on the usage of WAIT statements as well as on the usage of sensitivity lists

```
PROCESS (a,b)
BEGIN
   Sequential statements
END PROCESS;
```

```
PROCESS
BEGIN
   Sequential statements
   WAIT ON (a,b) ;
END PROCESS;
```

Multi-Process Architectures



- An architecture can have multiple process statements
- Each process executes in parallel with other processes
  - Order of process blocks does not matter
- Within a process, the statements are executed sequentially
  - Order of statements within a process does matter

Signal Assignment - Delay

- Signal assignments can delay updating their target by using a delay construct
- Signal assignments can incur delay
  - Two types of delays
    - Inertial delay (default)
      - Schedules output to be changed after delay passes unless input changes again
        - Input must remain stable while delay expires (i.e. switching circuit)

```
a <= b AFTER 10 ns;
a <= INERTIAL b AFTER 10 ns;
```

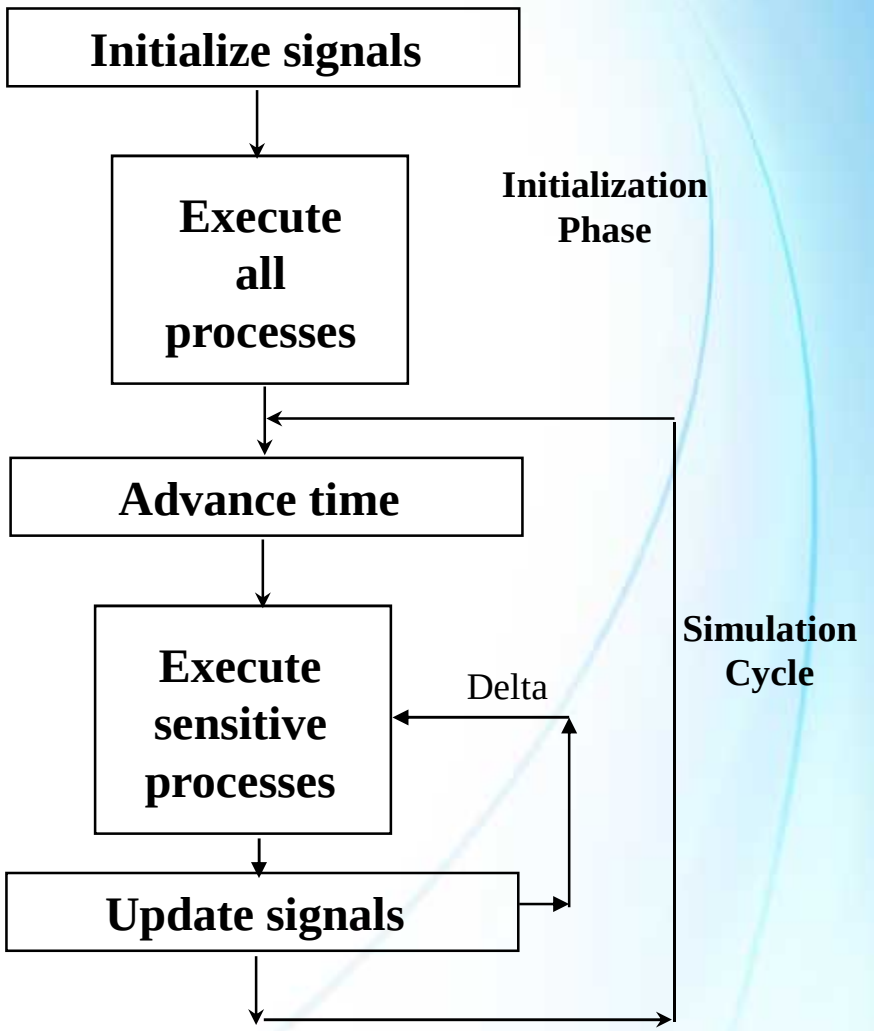
identical statements

    - Transport delay
      - Always schedules output to be changed after delay passes
        - Any transition on input transmitted to output (i.e. transmission line)

```
c <= TRANSPORT d AFTER 10 ns
```

Evaluating Model Behavior*

- Simulation cycle
  - Wall clock time
  - Delta
    - PROCESS execution phase
    - Signal update phase
- When does a delta cycle end?
  - After all processes end execution
    - End of the process (with sensitivity list)
    - Process encounters WAIT statement
  - After which, any signals written to during delta cycle are updated
- When does a simulation cycle end?
  - When updating signals to new values at the end of delta cycle does not cause a brand new delta cycle (i.e. new processes aren't triggered by changing signals)
- **Signals get updated at the end of the delta cycle (delay)**



**Note: Both VHDL simulation and synthesis tools evaluate models the same way.*

Equivalent Functions?? YES

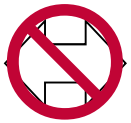
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp IS
    PORT (
        a, b : IN STD_LOGIC;
        y : OUT STD_LOGIC
    );
END ENTITY simp;

ARCHITECTURE logic OF simp IS
    SIGNAL c : STD_LOGIC;
BEGIN
    c <= a AND b;
    y <= c;
END ARCHITECTURE logic;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp_prc IS
    PORT (
        a,b : IN STD_LOGIC;
        y : OUT STD_LOGIC
    );
END ENTITY 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 ARCHITECTURE logic;
```

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

Equivalent Functions?? NO

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp IS
    PORT (
        a, b : IN STD_LOGIC;
        y : OUT STD_LOGIC
    );
END ENTITY simp;

ARCHITECTURE logic OF simp IS
    SIGNAL c : STD_LOGIC;
BEGIN
    c <= a AND b;
    y <= c;
END ARCHITECTURE logic;
```



```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp_prc IS
    PORT (
        a, b : IN STD_LOGIC;
        y: OUT STD_LOGIC
    );
END ENTITY 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 ARCHITECTURE logic;
```

New value of **c** not available for **y** until next process execution (requires another simulation cycle or transition on **a/b**)

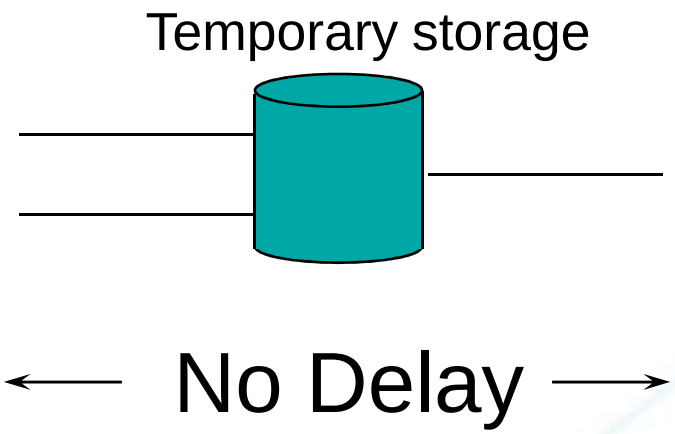
Variable Declarations

- 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);
```

- Variable assignments are represented by :=
- Examples
  - All bits

    temp := "10101010";

    temp := x"aa" ; (1076-1993)
      - VHDL also supports 'o' for octal and 'b' for binary
  - Bit-slicing

    temp (7 **DOWNTO** 4) := "1010";
  - Single bit

    temp(7) := '1';

- Use double-quotes (" ") to assign multi-bit values and single-quotes (' ') to assign single-bit values

Variable Assignment

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY var IS
    PORT (
        a, b : IN STD_LOGIC;
        y : OUT  STD_LOGIC
    );
END ENTITY var;

ARCHITECTURE logic OF var IS
BEGIN
    PROCESS (a, b)
        VARIABLE  c : STD_LOGIC;
    BEGIN
        c := a AND b;
        y <= c;
    END PROCESS;
END ARCHITECTURE logic;
```

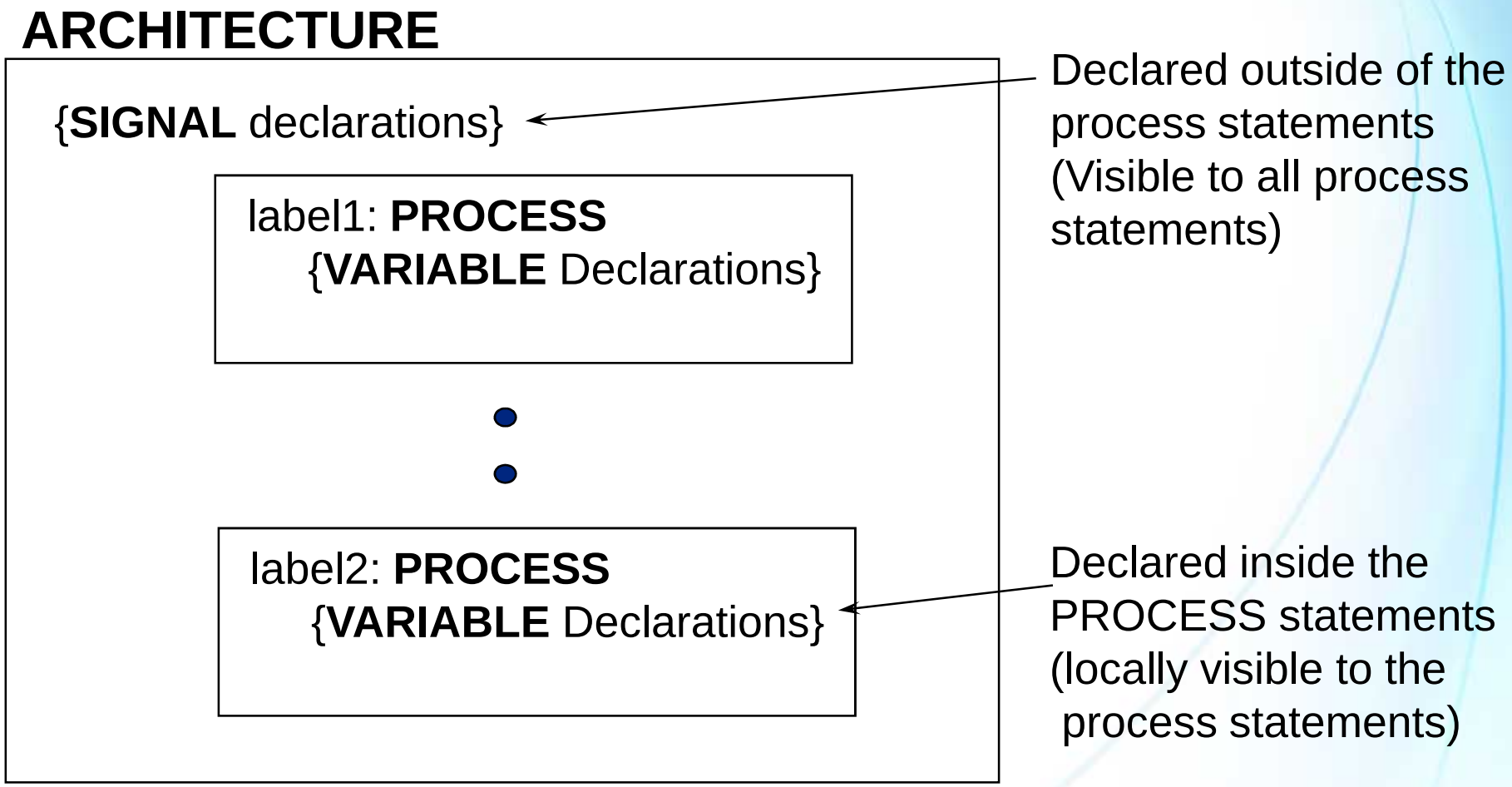
Variable **c** updated immediately and new value is available for assigning to **y**

Variable declaration

Variable assignment

Variable is assigned to a signal to synthesize to a piece of hardware

Signal and Variable Scope



Signals vs. Variables

	Signals (<=)	Variables (:=)
Assign	assignee <= assignment	assignee := assignment
Utility	Represent circuit interconnect	Represent local storage
Scope	Architecture scope (communicate between processes within architecture)	Local Scope (inside processes)
Behavior	Updated at end of current delta cycle (new value not immediately available)	Updated immediately

Sequential Statements

- Indicate behavior and express order
- Must be used inside explicit processes

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

Note: Simple signal assignment is considered both a sequential statement and a concurrent statement

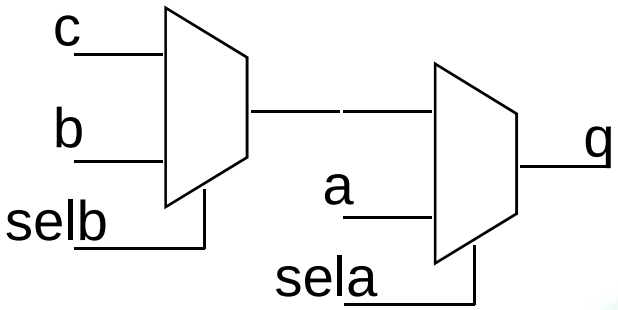
IF-THEN Statements

- 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

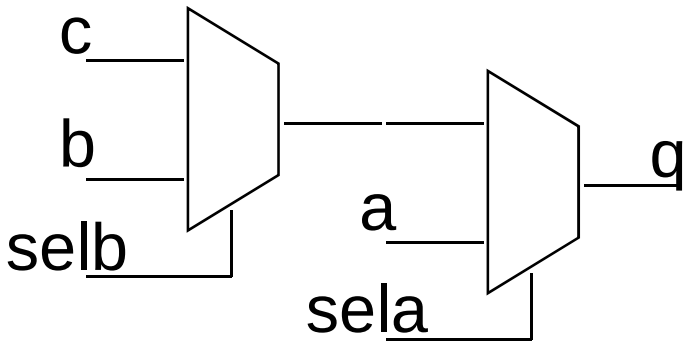
- Conditions are evaluated in order from top to bottom
  - Prioritization
- The first condition that is true causes the corresponding sequence of statements to be executed
- If all conditions are false, then the sequence of statements associated with the “ELSE” clause is evaluated

IF-THEN Statements

- Similar to conditional signal assignment

Implicit 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

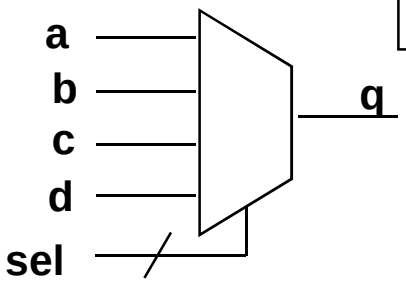
CASE Statement

- 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

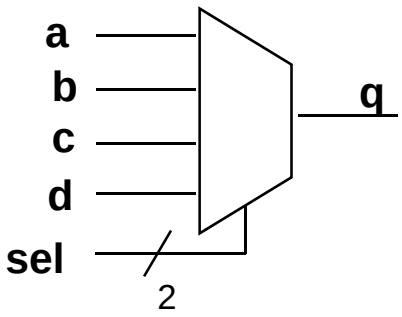
- Conditions are evaluated at once
  - No prioritization
- **All** possible conditions must be considered
- **WHEN OTHERS** clause evaluates all other possible conditions that are not specifically stated

CASE Statement

- Similar to selected signal assignment

Implicit 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 forever
- **While loop**
  - Loops until conditional test is false
- **For loop**
  - Loops for certain number of Iterations
  - Note: Iteration identifier not required to be previously declared
- Additional loop commands (each requires loop label)
  - **NEXT**
    - Skips to next loop iteration
  - **EXIT**
    - Cancels loop execution

Infinite Loop

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

While Loop

```
WHILE <condition> LOOP
  --Sequential statements
END LOOP;
```

For Loop

```
FOR <identifier> IN <range> LOOP
  --Sequential statements
END LOOP;
```

1's Counter Using FOR LOOP

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;
USE IEEE. STD_LOGIC_UNSIGNED.ALL ;
USE IEEE. STD_LOGIC_ARITH.ALL ;

ENTITY ones_count IS
      PORT (
            invec : IN STD_LOGIC_VECTOR  (31 DOWNTO 0);
            outvec : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
      );
END ENTITY ones_count;

ARCHITECTURE rtl OF ones_count IS
BEGIN
      PROCESS (invec)
            VARIABLE count: STD_LOGIC_VECTOR (7 DOWNTO 0);
      BEGIN
            count:= (OTHERS=>'0');
            FOR i IN invec'RANGE LOOP
                  IF (invec(i) /= '0') THEN
                        count:=count+1;
                  END IF;
            END LOOP;
            outvec <=count;
      END PROCESS;
END ARCHITECTURE rtl;
```

Variable **count** 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 **count** is assigned to a signal (**outvec**) before the end of the process to be visible outside process & to synthesize to a piece of hardware

WAIT Statements

- Pauses execution of process until WAIT statement is satisfied
- Types
  - WAIT ON <signal>
    - Pauses until signal event occurs

      WAIT ON a, b;
  - WAIT UNTIL <boolean_expression>
    - Pauses until boolean expression is true

      WAIT UNTIL (int < 100);
  - WAIT FOR <time_expression>
    - Pauses until time specified by expression has elapsed

      WAIT FOR 20 ns;
  - Combined WAIT

    WAIT UNTIL (a = '1') **FOR** 5 us;

* *Wait statement usage limited in synthesis*

Using WAIT Statements

```
stim: PROCESS
      VARIABLE error : BOOLEAN;
BEGIN
      WAIT UNTIL clk = '0';
      a <= (OTHERS => '0');
      b <= (OTHERS => '0');
      WAIT FOR  40 NS;
      IF (sum /= 0) THEN
            error := TRUE;
      END IF;

      WAIT UNTIL clk = '0';
      a <= "0010";
      b <= "0011";
      WAIT FOR 40 NS;
      IF (sum /= 5) THEN
            error := TRUE;
      END IF;

      …

      WAIT;
END PROCESS stim;
```

Pause execution of the process until clk transitions to a logic 0

Pause execution of the process until the equivalent of 40 ns passes in simulation time

Pause execution of the process indefinitely

Architecture Modeling Fundamentals

- Constants
- Signals
- Signal Assignments
- Operators
- Processes
- Variables
- Sequential Statements
- Subprograms
- Types

Subprograms

- VHDL has 2 subprograms
  - FUNCTION
    - Performs calculation and returns value
  - PROCEDURE
    - Performs sequence of defined sequential statements
- Uses
  - Replacing repetitive code
  - Enhancing readability
- Defined by means of subprogram declaration (optional) and subprogram body
  - Subprogram declarations required if subprogram is called before subprogram body is read
- Consist of sequential statements (like a process)
- May be declared in process, architecture or package
  - Determines visibility
  - When placed in package, subprogram declaration goes in package declaration and subprogram body goes in package body (see earlier package example)
- Synthesis places restrictions on use of subprograms

Function Definition & Call

Function Declaration

```
FUNCTION ones_count (SIGNAL a : STD_LOGIC_VECTOR) RETURN VARIABLE;
```

- Must return a single n value based on zero or more inputs
- Must be called in an expression

Function Body

```
Function ones_count (signal a: std_logic_vector)
                                    return  std_logic_vector is
   variable r: integer;
BEGIN

    r := 0;
    FOR i IN  a'RANGE LOOP
        IF a(i) /= '0' THEN
            r := r + 1 ;
        END IF;
    END LOOP;
    RETURN std_logic_vector(to_unsigned(r, 8));
END FUNCTION ones_count;
```

Invoking a Function

```
total_ones <= ones_count (input) WHEN test_ones = '1';
```

Note: 'RANGE is a VHDL attribute which returns the range of the object it is applied to (e.g. 7 DOWNTO 0)

Procedure Definition & Call

Procedure Declaration

```
PROCEDURE incr_comp (
      SIGNAL cnt_sig : INOUT STD_LOGIC_VECTOR;
      CONSTANT max : IN INTEGER;
      SIGNAL maxed_out : OUT BOOLEAN
);
```

- May have inputs, inouts and outputs
- May return zero or multiple outputs
- Must be called as a separate sequential statement

Invoking a Procedure

```
incr_comp (err_cnt, 12, err_cnt_maxed);
incr_comp (code_cnt, 144, code_cnt_maxed);
```

Procedure Declaration

```
PROCEDURE incr_comp (
      SIGNAL cnt_sig : INOUT STD_LOGIC_VECTOR;
      CONSTANT max : IN INTEGER;
      SIGNAL maxed_out : OUT BOOLEAN
      ) IS
      -- declare any local objects (i.e. constants,
      --                                  variables,...)
BEGIN
      IF cnt_sig >= max THEN
            maxed_out <= TRUE;
      ELSE
            maxed_out <= FALSE;
            cnt_sig <= cnt_sig + 1;
      END IF;
END PROCEDURE incr_comp;
```

Functions vs. Procedures

Functions

- Always execute in zero time
  - Cannot pause their execution
  - Can not contain any delay, event, or timing control statements
- Must have at least one input argument
  - Inputs may not be affected by function
- Arguments may not be outputs and inouts
- Always return a single value

Procedures

- May execute in non-zero simulation time
  - May contain delay, event, or timing control statements
- May have zero or more input, output, or inout arguments
- Modify zero or more values

Types

- VHDL has built-in data types to model hardware (e.g. BIT, BOOLEAN, STD_LOGIC)
- VHDL also allows creation of brand new types for declaring objects (i.e. constants, signals, variables)

- Subtype
- Enumerated Data Type
- Array

Subtype

- A constrained type
- Synthesizable if base type is synthesizable
- Use to make code more readable and flexible
  - Place in package to use throughout design

```
ARCHITECTURE logic OF subtype_test IS

    SUBTYPE word IS std_logic_vector (31 DOWNTO 0);
    SIGNAL mem_read, mem_write : word;

    SUBTYPE dec_count IS INTEGER RANGE 0 TO 9;
    SIGNAL ones, tens : dec_count;

BEGIN
```

Enumerated Data Type

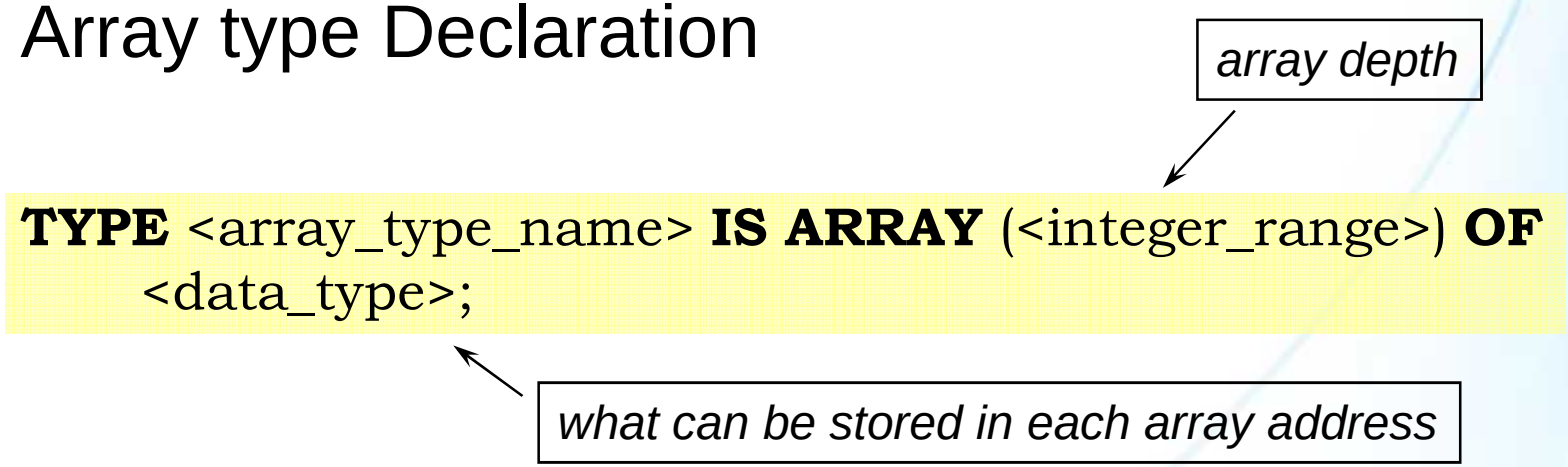
- Allows user to create data type ***name*** and ***values***
  - Must create constant, signal or variable of that type to use
- Used in
  - Making code more readable
  - Finite state machines
- Enumerated type declaration

TYPE <*your_data_type*> **IS**
(*data type items or values separated by commas*);

```
TYPE enum IS (idle, fill, heat_w, wash, drain);
SIGNAL dshwshr_st : enum;
        …
drain_led <= '1' WHEN dshwsher_st = drain ELSE '0';
```

Array

- Creates multi-dimensional data type for storing values
  - Must create constant, signal or variable of that type
- Used to create memories and store simulation vectors
- Array type Declaration



Array Example

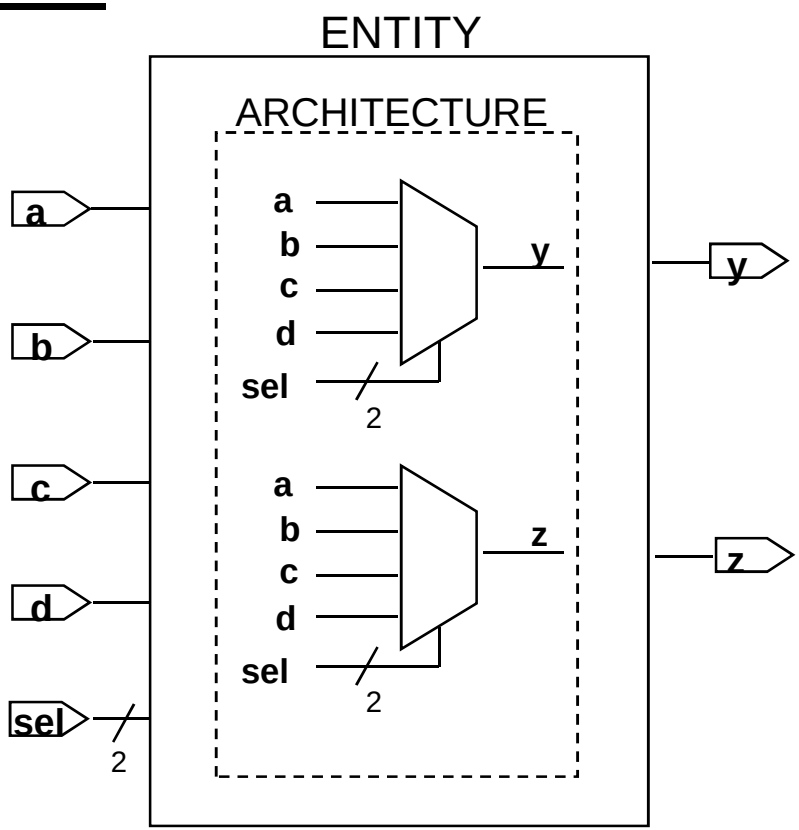
```
ARCHITECTURE logic OF my_memory IS
    -- Creates new array data type named mem which has 64
    --   address locations each 8 bits wide
    TYPE mem IS ARRAY (0 to 63) OF std_logic_vector (7 DOWNTO 0);

    -- Creates 2 - 64x8-bit array to use in design
    SIGNAL mem_64x8_a, mem_64x8_b : mem;

BEGIN
    …
    mem_64x8_a(12) <= x“AF”;
    mem_64x8_b(50) <= “11110000”;
    …
END ARCHITECTURE logic;
```

Introduction to VHDL

VHDL and Logic Synthesis

VHDL Model - RTL Modeling

Result:

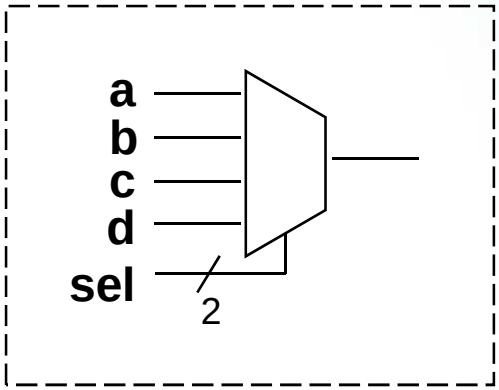


- Type of behavioral modeling that implies or infers hardware
- Describes functionality and implies structure of the circuit

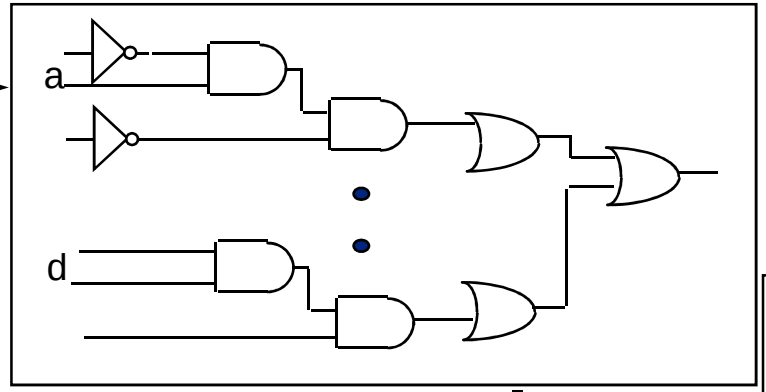
Recall - RTL Synthesis

```
PROCSS (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;
```

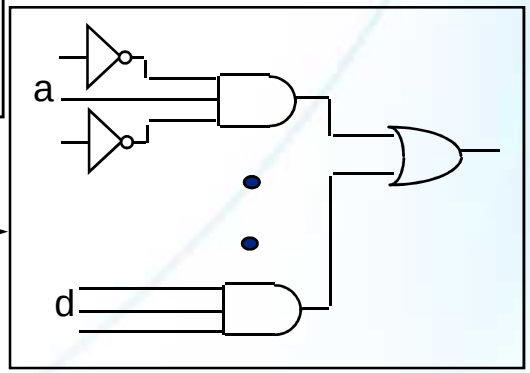
Inferred



Translation



Optimization

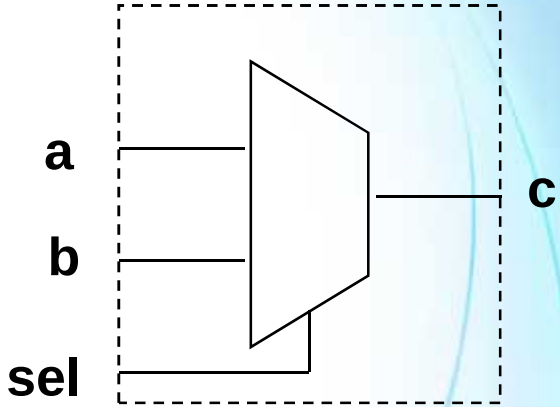


Two Types of Process Statements

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

```
PROCESS (a, b, sel)
```

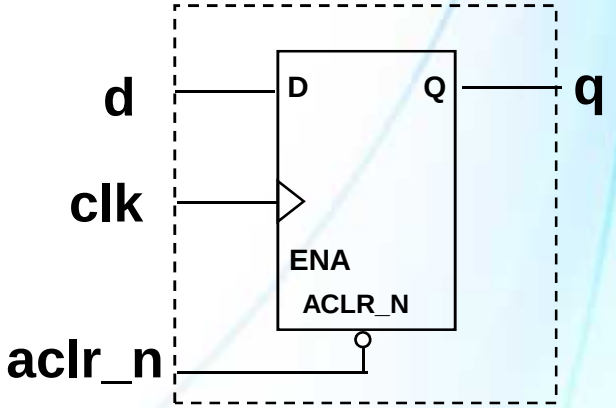
Sensitivity list includes all inputs used In the combinatorial logic



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

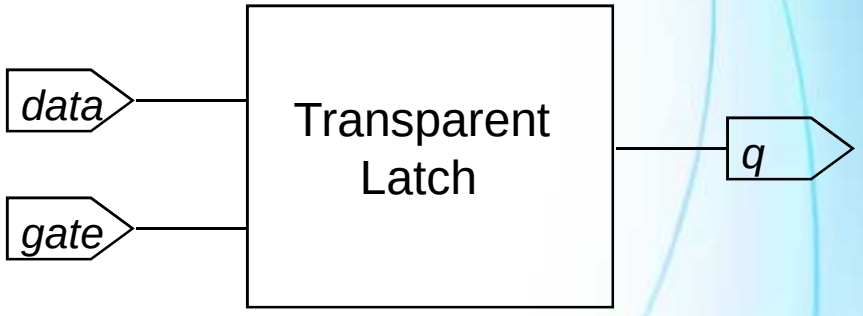
```
PROCESS (clr, clk)
```

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 ENTITY latch1;

ARCHITECTURE logic OF latch1 IS
BEGIN
    label_1: PROCESS (data, gate)
    BEGIN
        IF gate = '1' THEN
            q <= data;
        END IF;
    END PROCESS label_1;
END ARCHITECTURE behavior;
```



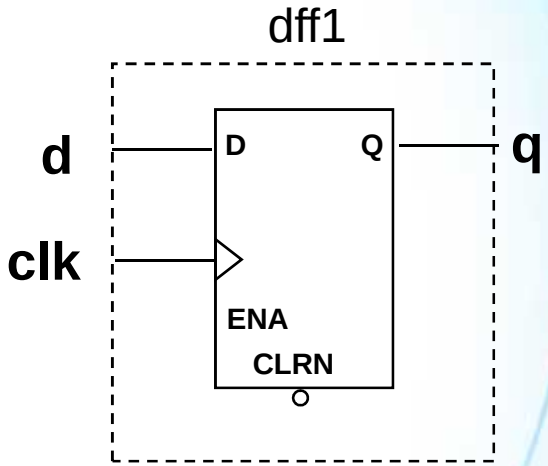
Sensitivity list includes both inputs

What happens if gate = '0'?

⇨ ***Implicit memory***

DFF – clk'EVENT AND clk='1'

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff1 IS
    PORT (
        d  : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY dff1;

ARCHITECTURE logic OF dff1 IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF clk'EVENT AND clk = '1' THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE behavior;
```

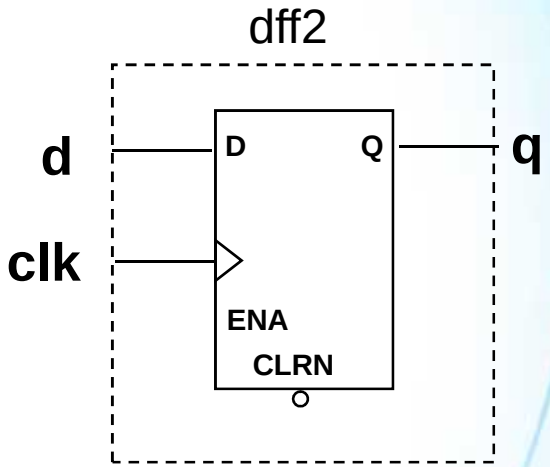


clk'EVENT* AND *clk='1'

- *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 dff2 IS
    PORT (
        d : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY dff2;

ARCHITECTURE logic OF dff2 IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge (clk) THEN
            q <= d;
    END IF;
    END PROCESS;
END ARCHITECTURE behavior;
```

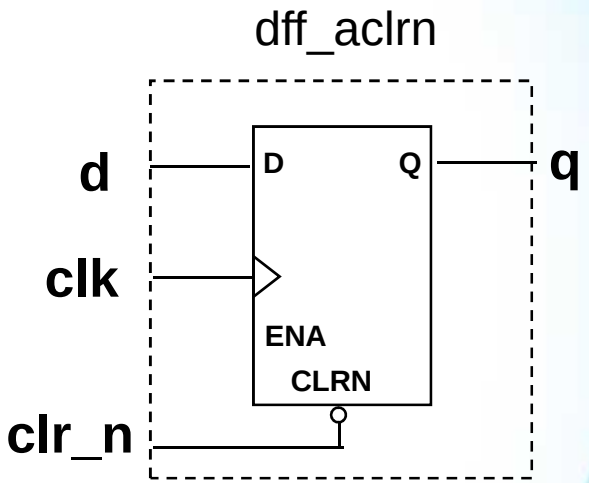


rising_edge

- *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

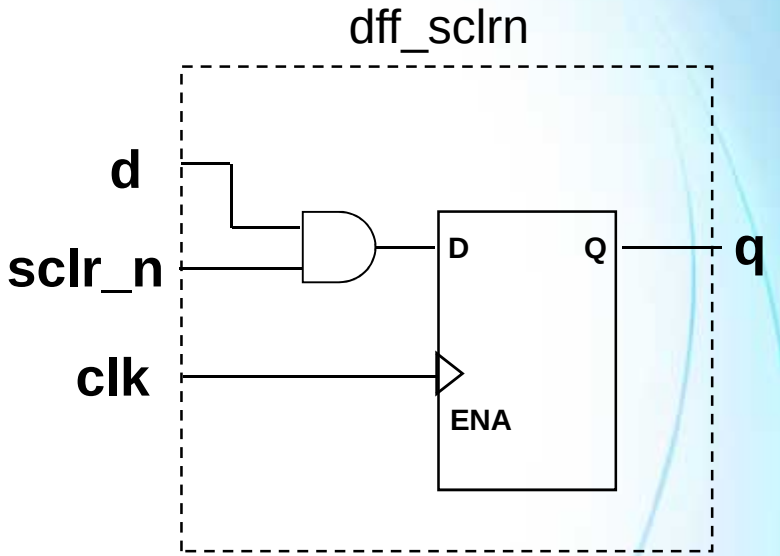
```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;
USE IEEE. STD_LOGIC_UNSIGNED.All;

ENTITY dff_aclrn IS
    PORT (
        aclr_n : IN STD_LOGIC;
        d, clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY dff_aclrn;

ARCHITECTURE logic OF dff_aclrn IS
BEGIN
    PROCESS (clk, aclr_n)
    BEGIN
        IF aclr_n = '0' THEN
            q <= '0';
        ELSIF rising_edge (clk) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE behavior;
```



- *This is how to implement asynchronous control signals for the register*
- *Asynchronous control signal is included in the sensitivity list and is checked before the clock condition (**rising_edge**)*
- *Therefore, **aclr_n='0'** does not depend on the clock*

DFF with Synchronous Clear

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;
USE IEEE. STD_LOGIC_UNSIGNED.All;

ENTITY dff_sclrn IS
      PORT (
            sclr_n : IN STD_LOGIC;
            d, clk : IN STD_LOGIC;
            q :      OUT STD_LOGIC
      );
END ENTITY dff_sclrn;

ARCHITECTURE logic OF dff_sclrn IS
BEGIN
      PROCESS (clk)
      BEGIN
            IF rising_edge (clk) THEN
                  IF sclr_n = '0' THEN
                        q <= '0';
                  ELSE
                        q <= d;
                  END IF;
            END IF;
      END PROCESS;
END ARCHITECTURE behavior;
```



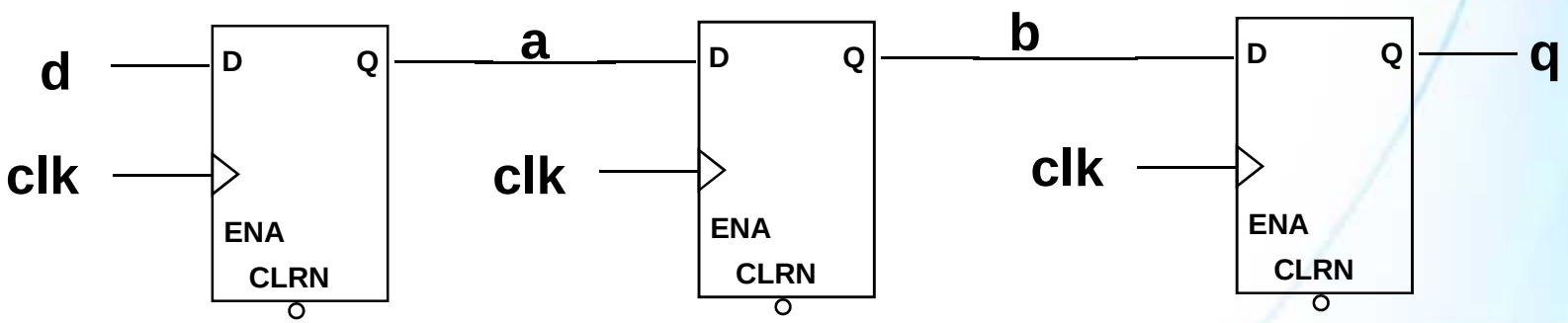
- *This is how to implement synchronous control signals for the register*
- *Synchronous control signals are not included in the sensitivity list and are checked inside in the rising_edge IF-THEN statement that checks the condition **rising_edge***
- *Therefore, **sclr_n='0'** does depend on the clock*

How Many Registers?

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY reg1 IS
    PORT (
        d : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY reg1;

ARCHITECTURE logic 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 ARCHITECTURE reg1;
```

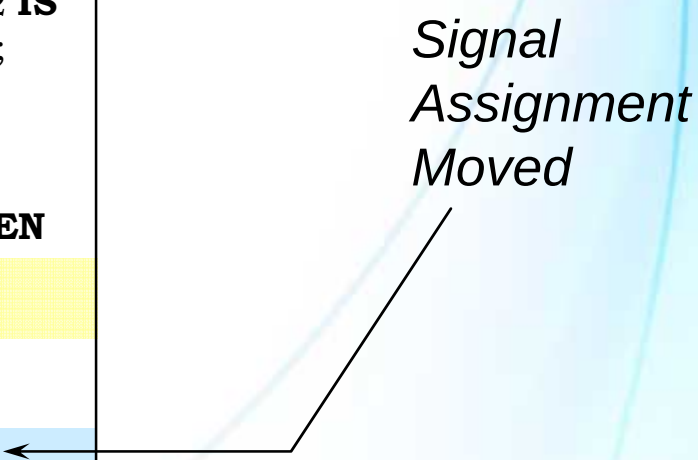
How Many Registers?

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



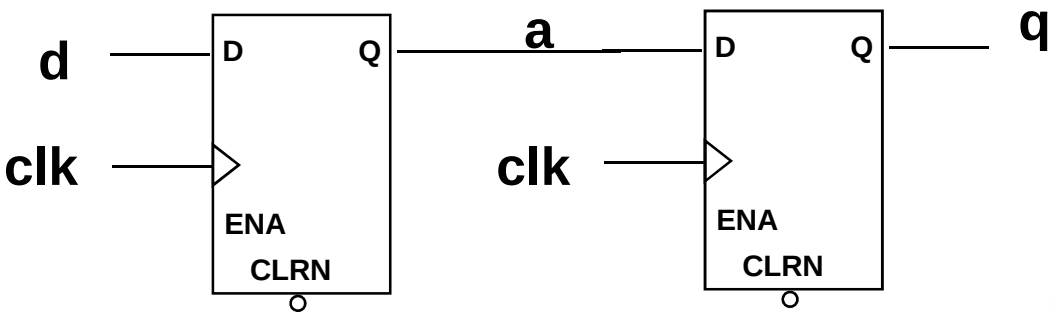
How Many Registers?

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY reg2 IS
    PORT (
        d : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY reg2;

ARCHITECTURE logic OF reg2 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 ARCHITECTURE reg1;
```



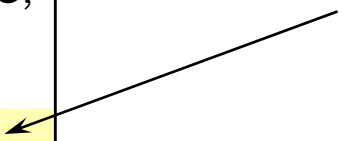
How Many Registers?

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



How Many Registers?

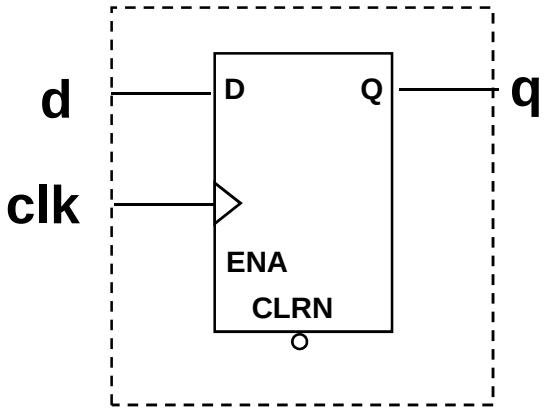
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY reg3 IS
    PORT (
        d : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : OUT STD_LOGIC
    );
END ENTITY reg3;

ARCHITECTURE logic OF reg3 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 ARCHITECTURE 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 are temporary storage and have no hardware intent
- Variable assignments can be used in expressions to immediately update a value
  - Variable can be assigned to a signal to infer hardware
- Exception: If a variable is read before a value is written to it, that implies feedback and will infer hardware
  - Register in a clocked process
  - Latch in a combinatorial process

Example - Counter Using a Variable

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

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY counter IS
    PORT (
        clk, aclr, clk_ena : IN STD_LOGIC;
        q : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END ENTITY counter;

ARCHITECTURE logic OF counter IS
BEGIN
    PROCESS (clk)
        VARIABLE q_var : STD_LOGIC_VECTOR (15 DOWNTO 0);
    BEGIN
        IF aclr = '1' THEN
            q_var :=  (OTHERS => '0');
        ELSIF rising_edge (clk) THEN
            IF clk_ena = '1' THEN
                q_var := q_var + 1;
            END IF;
        END IF;
        q <= q_var;
    END PROCESS;
END ARCHITECTURE logic;
```

Arithmetic expression assigned to variable before writing known value

Variable assigned to a signal to create hardware

Exercise 4

Please Go to Exercise 4

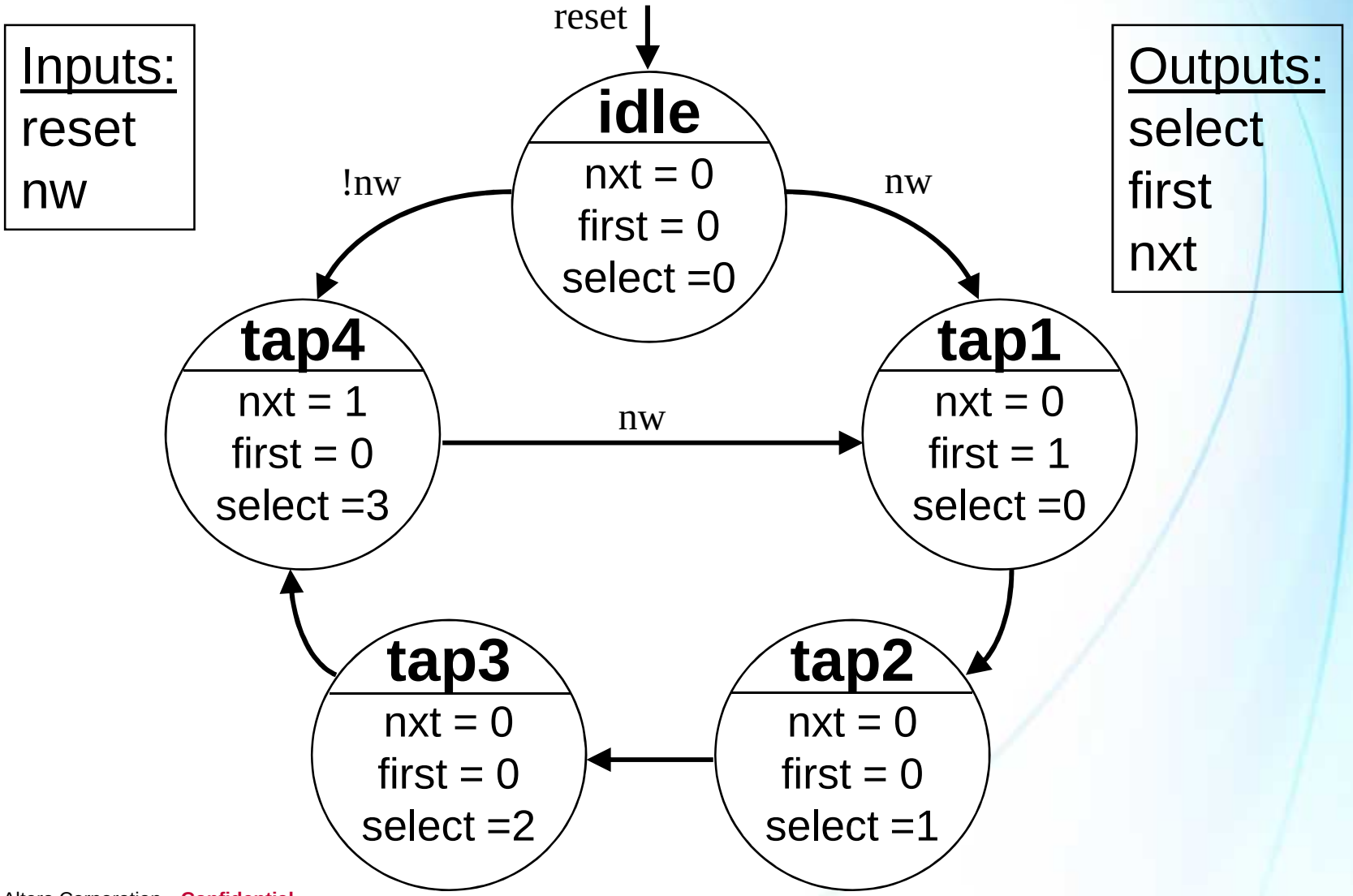
Introduction to VHDL

Model Application

Model Application

- Let's look at an example that combines some of the constructs learned so far
  - Type declaration
  - Combinatorial process
  - Sequential process
  - Combined sequential statements
- Also illustrates how more complex logic can be easily described in VHDL using a behavioral method
  - Building a model directly from a behavioral description

Finite State Machine (FSM) - State Diagram



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*;

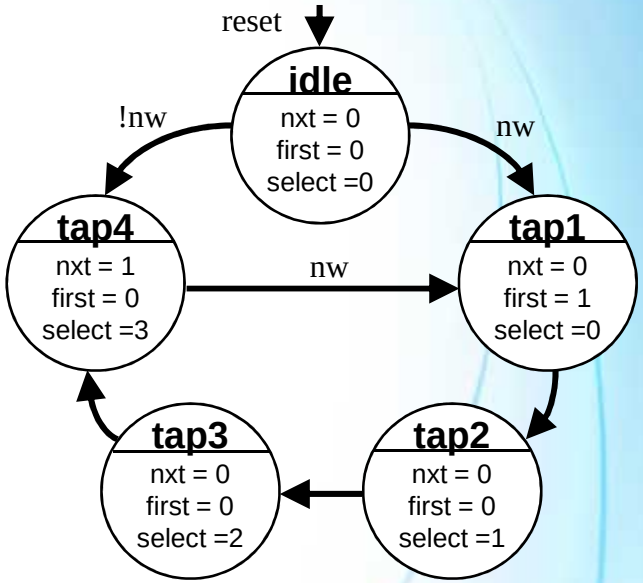
Note: This is one of several similar methods in which a state machine can be described. Here the emphasis isn't on optimizing a state machine, but just on building the VHDL model.

Writing VHDL Code for FSM (cont.)

- To determine next state transition/logic
  - Use a **CASE** statement inside a sequential process
- Use 1 of 2 methods to determine state machine outputs
  1. Use a combinatorial process with a **CASE** statement
  2. Use **conditional** and/or **selected** signal assignments for each output

FSM VHDL Code - Enumerated Data Type

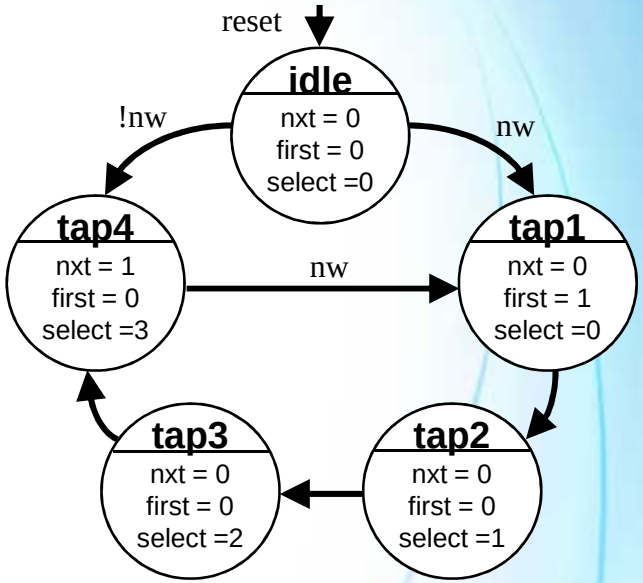
```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;

ENTITY filter_sm IS
    PORT (
        clk, reset, nw : IN STD_LOGIC;
        select : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
        nxt, first : OUT STD_LOGIC
    );
END ENTITY filter_sm;

ARCHITECTURE logic OF filter_sm IS

    TYPE state_type IS (idle, tap1, tap2, tap3, tap4);
    SIGNAL filter : state_type;

BEGIN
```



Enumerated data type

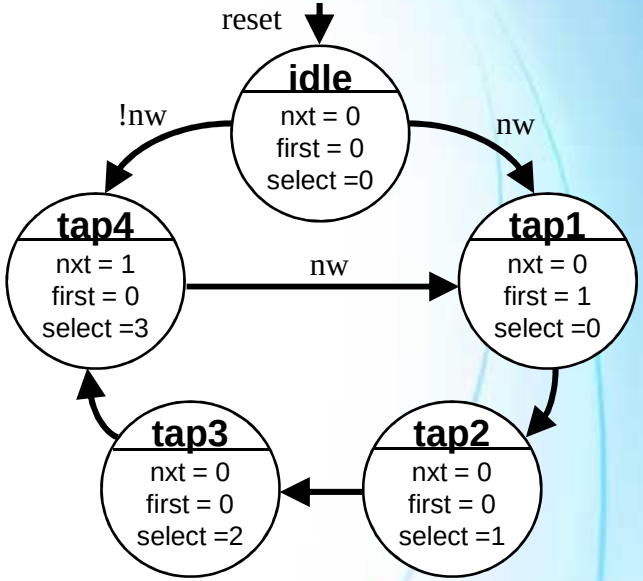
FSM VHDL Code - Next State Logic

```
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;
```



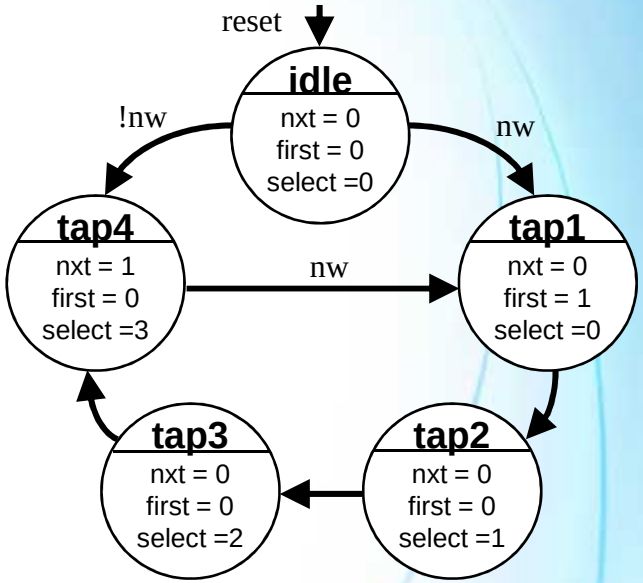
FSM VHDL Code - Outputs Using CASE

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



FSM VHDL Code - Outputs Using Signal Assigments

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

first <= '1'  WHEN filter=tap1 ELSE
    '0';

WITH filter SELECT
    select <=  "00" WHEN tap1,
               "01" WHEN tap2,
               "10" WHEN tap3,
               "11" WHEN tap4,
               "00" WHEN OTHERS;

END ARCHITECTURE logic;
```



Conditional signal assignments

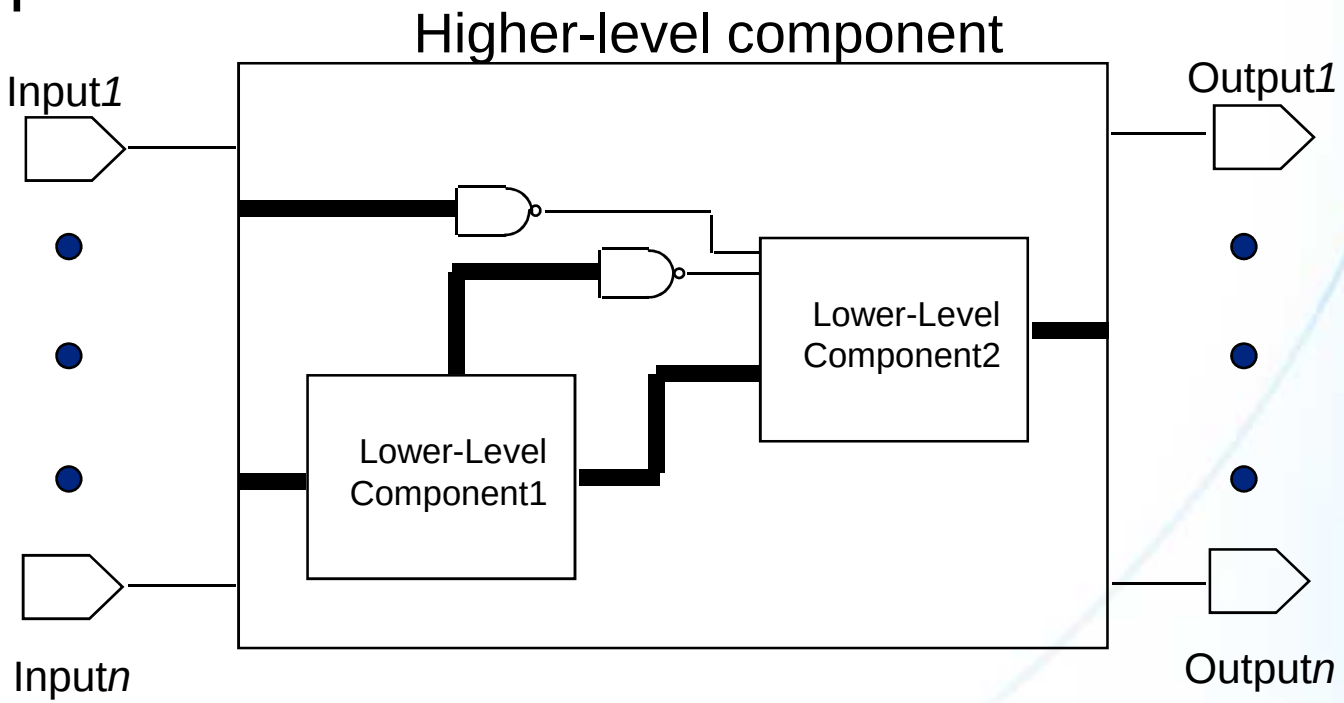
Selected signal assignments

Introduction to VHDL

Designing hierarchy

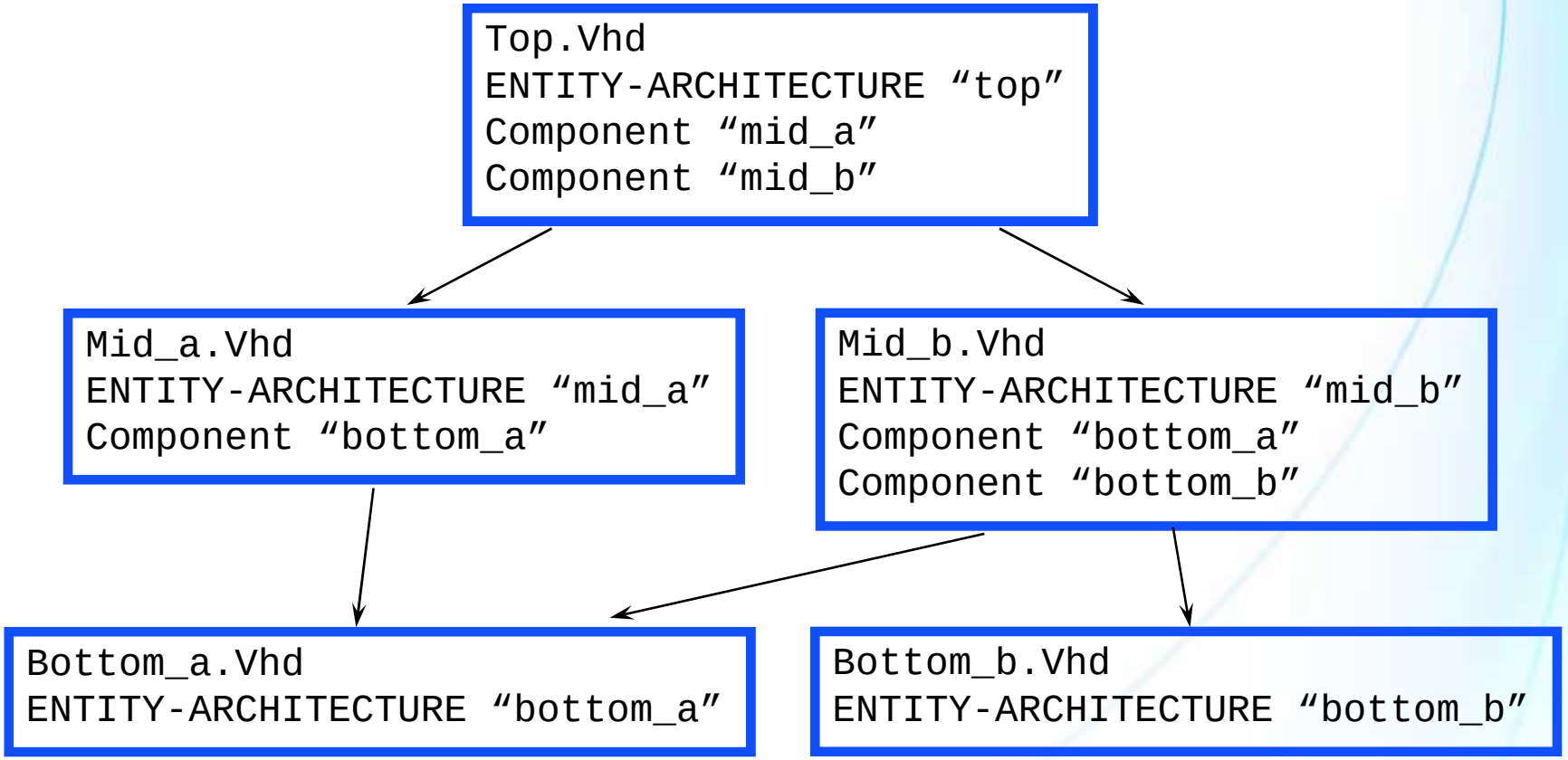
Recall - Structural Modeling

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



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>
        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

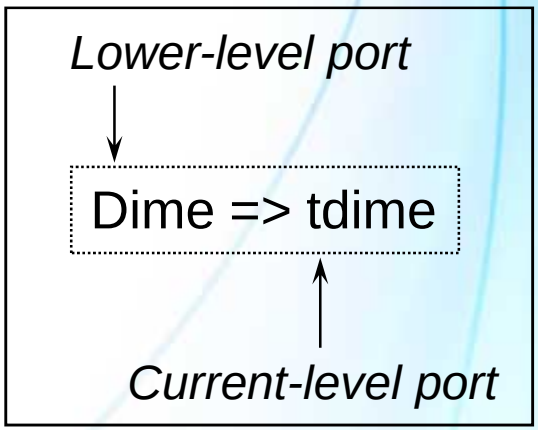
```
<Instance_name> : <lower-level_design_name>
        PORT MAP(<lower-level_port_name>  => <current_level_port_name>,
                …, <lower-level_port_name> => <current_level_port_name>);
```

Component Declaration and Instantiation (1)

```
LIBRARY IEEE;
USE  IEEE. STD_LOGIC_1164.ALL;

ENTITY tolleab IS
    PORT (
        clk, tcross, tnickel, tdime, tquarter : IN STD_LOGIC;
        tgreen, tred : OUT STD_LOGIC
    );
END ENTITY tolleab;

ARCHITECTURE tolleab_arch OF tolleab IS
    COMPONENT tollv
        PORT(
            clk, cross, nickel, dime, quarter : IN STD_LOGIC;
            green, red : OUT STD_LOGIC;
        );
    END COMPONENT;
BEGIN
    U1 :  tollv PORT MAP (clk => tclk, cross => tcross,
        nickel => tnickel, dime => tdime, quarter => tquarter,
        green => tgreen, red => tred);
END ARCHITECTURE tolleab_arch;
```

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

Component declaration



Named Association

Instance label/name

Component instantiation

Component Declaration and Instantiation (2)

- Component instantiation using **positional association**
  - Order of ports in declaration maps to order of ports in instantiation
  - Not recommended

```
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;
        );
    END COMPONENT;
BEGIN
    U1 :  tollv  PORT MAP (tclk, tcross, tnickel, tdime,
        tquarter, tgreen, tred);
```

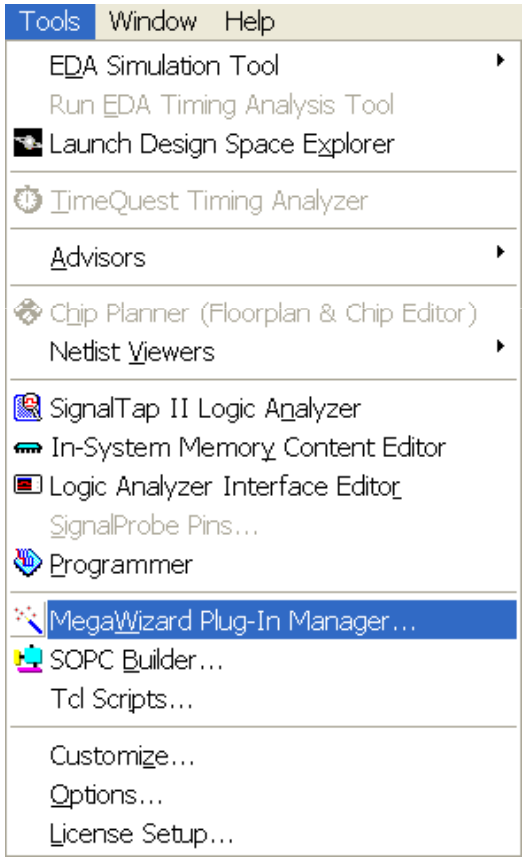
Positional association

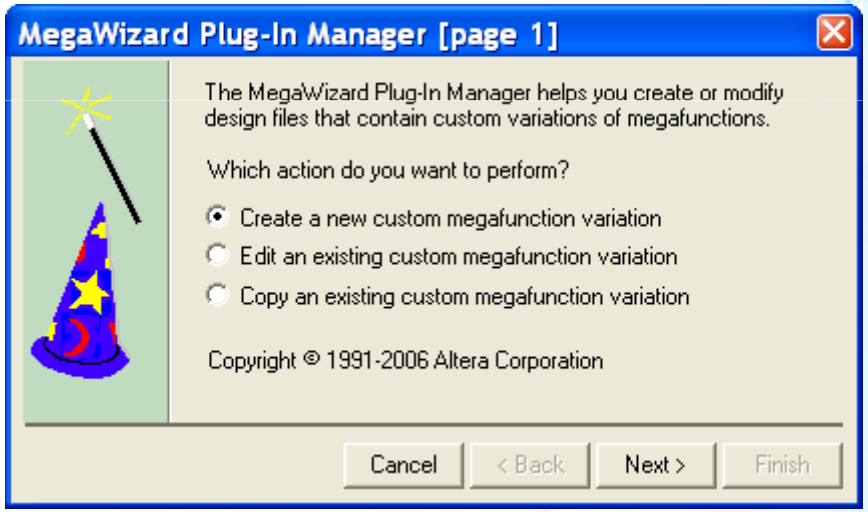
Vendor Libraries

- Silicon vendors often provide libraries of macro functions & primitives
  - Altera library
    - Maxplus2
    - MegaCore® functions
- 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 mega-functions and primitives

Accessing the MegaWizard® Tool

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





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;
```

```
1  my_ram_inst : my_ram PORT MAP (
2           data     => data_sig,
3           wraddress    => wraddress_sig,
4           rdaddress    => rdaddress_sig,
5           wren     => wren_sig,
6           wrclock  => wrclock_sig,
7           rdclock  => rdclock_sig,
8           q    => q_sig
9       );
10
```

Exercise 5

Please Go to Exercise 5

Introduction to VHDL

Summary

Summary

- Implement basic constructs of VHDL
  - Entity
  - Architecture
  - Package/Library
- Implement modeling structures of VHDL
  - Constant, signal & variable
  - Process
  - Sequential statement
  - Subprogram
  - Type
  - Component declaration & instantiation

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

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).

Online Training

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 approximate one to three hours to complete.

http://www.altera.com/training

View training class schedule & register for a class