Introduction to the VHDL language

Goals

VHDL is a versatile and powerful hardware description language which is useful for modelling electronic systems at various levels of design abstraction.

- ✍ Introduce basic VHDL concepts and constructs
- ✍ Introduce the VHDL simulation cycle and timing model
- ✍ Illustrate VHDL's utility as a digital hardware description language: concurrent and sequential mode of execution

Topic: modelling digital systems

Digital System: any digital circuit that processes or stores information; (both the system as a whole and the various part from which is constructed)

Goals

It is not humanly possible to comprehend complex systems in their entirely

Use of model….

The model represent that information which is relevant and abstracts away from irrelevant details

Several models for the same system, since different information is relevant in different context

Module Outline(I)

- Introduction
- VHDL Design Examples
- VHDL Model Components
  - Entity Declarations
  - Architecture Descriptions
  - Timing Model

Module Outline (II)

- Basic VHDL Constructs
  - Data types
  - Objects
  - Sequential and concurrent statements
  - Packages and libraries
  - Attributes
  - Predefined operators
- Examples
- Summary

Module Outline

- **Introduction**
- VHDL Design Example
- VHDL Model Components
- Basic VHDL Constructs
- Examples
- Summary

VHDL Education Coverage

- In a survey of 71 US universities (representing about half of the EE graduating seniors in 1993), they reported
  - 44% have no training on or use of VHDL in any undergraduate EE course
  - 45% have no faculty members who can teach VHDL
  - 14% of the graduating seniors have a working knowledge of VHDL and only 8% know Verilog
- However, in the 1994 USE/DA Standards Survey, 85% of the engineers surveyed were designers and reported
  - 55% were familiar with EDIF
  - 55% were familiar with VHDL
  - 33% were familiar with Verilog

VHDL

- VHDL is an IEEE and U.S. Department of Defence standard for electronic system descriptions.
- It is also becoming increasingly popular in private industry as experience with the language grows and supporting tools become more widely available.
- Therefore, to facilitate the transfer of system description information, an understanding of VHDL will become increasingly important.

VHDL’s History

- The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is the product of a US Government request for a new means of describing digital hardware.
- The VHSIC Program was an initiative of the Defence Department to push the state of the art in VLSI technology, and VHDL was proposed as a versatile hardware description language.

VHDL’s History (cont.)

- Very High Speed Integrated Circuit (VHSIC) Program
  - Launched in 1980
  - Aggressive effort to advance state of the art
  - Object was to achieve significant gains in VLSI technology
  - Need for common descriptive language
  - $17 Million for direct VHDL development
  - $16 Million for VHDL design tools
- Woods Hole Workshop
  - Held in June 1981 in Massachusetts
  - Discussion of VHSIC goals
  - Comprised of members of industry, government, and academia

VHDL’s History (Cont.)

- In July 1983, a team of Intermetrics, IBM and Texas Instruments were awarded a contract to develop VHDL
- In August 1985, the final version of the language under government contract was released: VHDL Version 7.2
- In December 1987, VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI standard
- In September 1993, VHDL was restandardized to clarify and enhance the language (IEEE Standard 1076-1993)
- VHDL has been accepted as a Draft International Standard by the IEC (International Engineering Consortium)
- VHDL 1993, 1997, 2000, 2002 …

Reasons for Using VHDL

- VHDL is an international IEEE standard specification language (IEEE 1076-1993) for describing digital hardware used by industry worldwide
  - VHDL is an acronym for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- VHDL enables hardware modelling from the gate to system level
- VHDL provides a mechanism for <u>digital design</u> and <u>reusable design documentation</u>

Reasons for Using VHDL (Cont.)

- Formal Specification (not ambiguous) of system's requirements: formal model to communicate
- Modelling : documentation
- Testing & Validation using simulation
- Formal verification of correctness of a design: require mathematical statement of the required functions of the system
- Performance prediction
- Automatic synthesis

Gajski and Kuhn's Y Chart

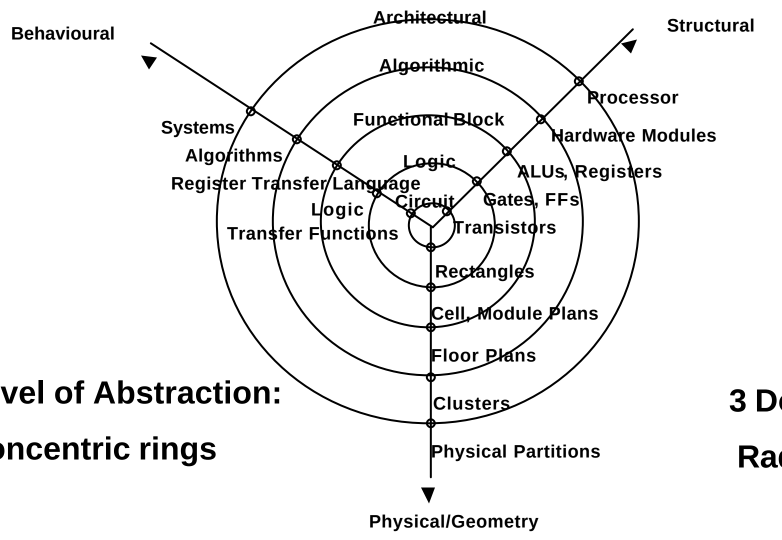
- VHDL allows one to model systems where more than one thing is going on at a time:
  - concurrency;
  - discrete event simulation.
- VHDL allows the designer to work at various level of abstraction:
  - behavioural;
  - RTL;
  - boolean equations;
  - gates.
- Many of the levels are shown pictorially in the Gajski and Kuhn's Chart.
- Although VHDL does not support system description at the physical/geometry level of abstraction, many design tools can take behavioural or structural VHDL and generate chip layout.

Gajski and Kuhn’s Y Chart (Cont.)

Domain and levels of abstraction

- ✍ **There are different models of a systems. We can classify these models into three domains:**
  - ✍ ***Functional (Behavioural) Domain***
    - ✍ **concerned with the operation performed by the system. It’s the most abstract domain of description, since it does not indicate how the function is implemented.**
  - ✍ **Structural Domain**
    - ✍ **deals with how the system is composed of interconnected subsystems.**
  - ✍ **Geometric Domain**
    - ✍ **deals with how the system is laid out in the physical space.**
- ✍ **Each of these domain can also be divided into level of abstraction.**
  - ✍ **At the top level (behavioural modelling) we consider an overview of function, structure or geometry, at the lower levels we introduce successively finer detail.**

Gajski and Kuhn’s Y Chart (Cont.)



Level of Abstraction:

Concentric rings

3 Domains:

Radial axis

Additional Benefits of VHDL

- Allows for various <u>design methodologies</u>:
  - top-down, bottom-up, delay of detail.
  - very flexible in its approach to describing hardware.
- Provides <u>technology independence</u>:
  - VHDL is independent of any technology or process (ASIC, FPGA…)
  - however VHDL code can be written and then targeted for many different technologies.
- Describes a wide variety of digital hardware
  - various levels of design abstraction(previous slides)
  - Mix descriptions: behavioural with gate level description.

Additional Benefits of VHDL

- Eases communication through standard language:
  - Communication among different designers and different tools
  - Easier documentation
  - Ability to run the same code in a variety of environment.
- Allows for better design management
  - Use of constructs (packages, libraries) allows common elements sharing.
- Provides a flexible design language.
- Has given rise to derivative standards:
  - WAVES, VITAL, Analog VHDL.

Features of VHDL Model

A single component model is composed of one entity and one of more architectures.

- The entity represents the interface specification (I/O) of the component.
  - It defines the component's external view, sometimes referred to as its "pins" (entity declaration).
- The architecture(s) describe(s) the internal implementation of an entity (architecture body).
  - There are three general types of architectures.

Design Entity

- Design Entity represents VHDL model's basic element. It could represent a whole system,a PCB (Printed Circuit Board), an IC (Integrated Circuit) or a gate.
- A VHDL Model can be created at different abstraction levels (behavioral, dataflow, structural), according to a refinement of starting specification.

Design Entity

- ✍ We call the module F a design *entity*, and the inputs and outputs are called *ports*.

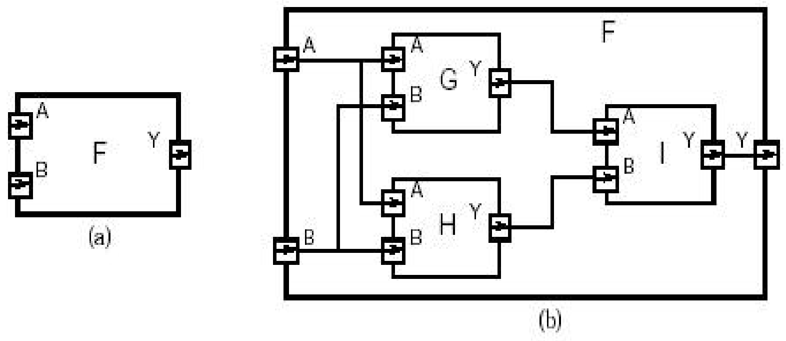
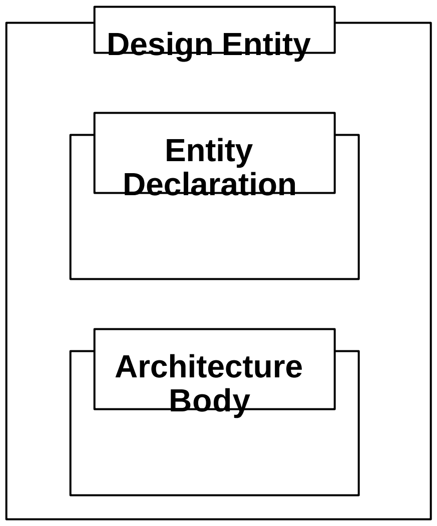


Figure 1-1. Example of a structural description.

VHDL Model: Structure



Features of VHDL Model:

Types of architectures

? One type of architecture described the structure of the design in terms of its sub-components and their interconnections.

? A second type of architecture, containing only concurrent statements, is commonly referred to as a dataflow description. Concurrent statements execute when data is available on their inputs. These statements occur in any order within the architecture.

? A third type of architecture is the behavioural description in which the functional and possibly timing characteristic are described using VHDL concurrent statements and processes. The process is the concurrent statement of an architecture All statements within a process execute sequential order until it gets suspended by a wait statement.

Features of VHDL Model (Cont.)

✍ Packages are used to provide a collection of common declaration, constants, and/or subprograms to entities and architectures.

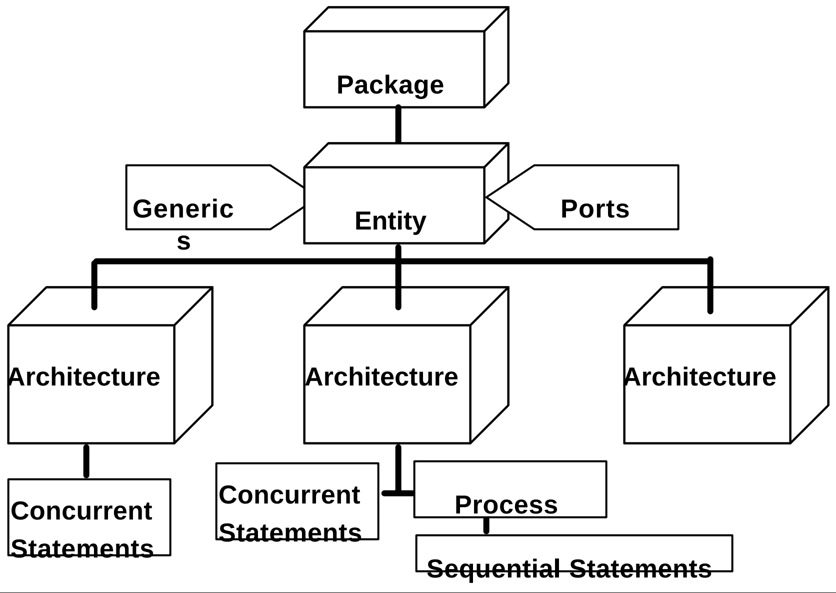
✍ Generics provide a method for communicating information to an architecture from the external environment. They are passed through the entity construct.

✍ Ports provide the mechanism for a device to communicate with its environment. A port declaration defines the names, types directions and possible default values for the signals in a component's interface.

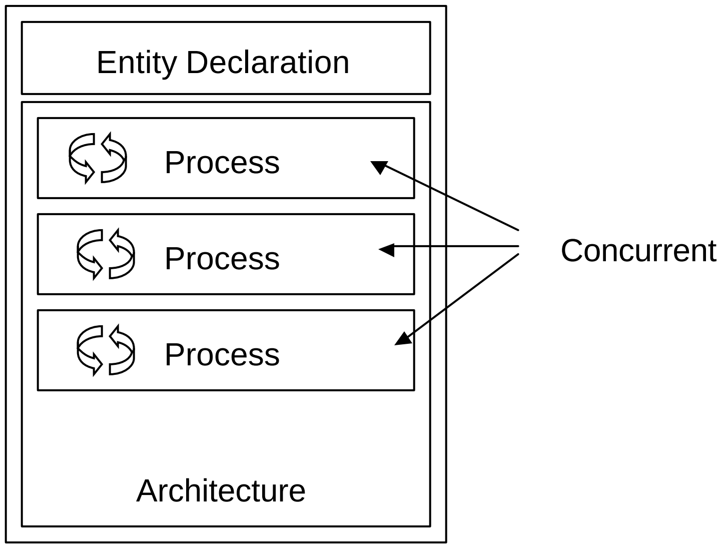
Other Features of VHDL Model (Cont.)

- ✍ Configuration is an instruction used to bind the component instances to design entities. In it, we specify which real entity interface and corresponding architecture body should be used for any component instances.
- ✍ Bus is a signals group or a particular method of communication.
- ✍ Driver is a source for a signal in that it provides values to be applied to the signal.
- ✍ Attribute is a VHDL object's additional information.

Putting It All Together



Putting It All Together

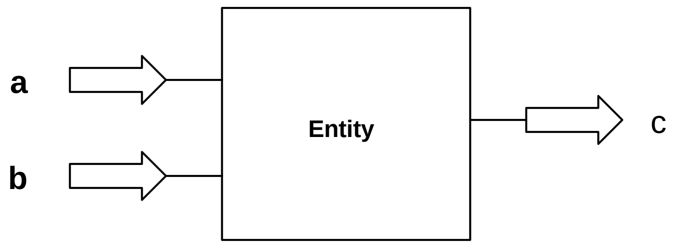


Module Outline

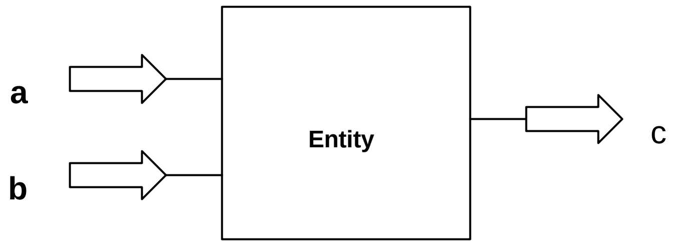
- ✍ **Introduction**
- ✍ **VHDL Design Example**
- ✍ **VHDL Model Components**
- ✍ **Basic VHDL Constructs**
- ✍ **Examples**
- ✍ **Summary**

Conceptual Specification

- ✍ High level specification represents a functionality description.



Specification Analysis



Truth Table Generation:

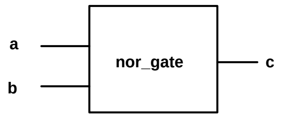
Input		Output
a	b	c
0	0	1
0	1	0
1	0	0
1	1	0

Entity Declaration

- ✍ It defines an Entity and the interface between Entity and the external environment.
- ✍ Format:

```
ENTITY entity_name  IS
[GENERIC (generic list);]
[PORT (port list);]
END [entity_name];
```

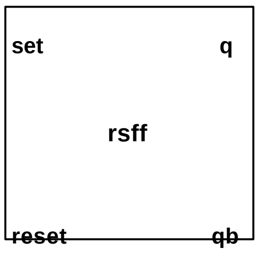
- ✍ Example:



```
ENTITY nor_gate  IS
[PORT (a, b : IN BIT;
              c : OUT BIT);
END nor_gate;
```

VHDL vs Traditional Design Methods

CAE WORKSTATION



A design is represented by fundamental symbols connected with signals.

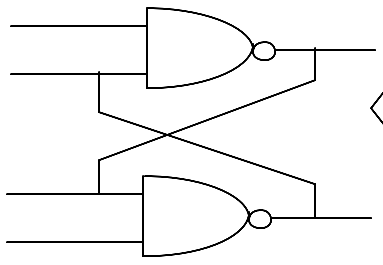
VHDL *Entity Declaration*

```
ENTITY rsff IS
PORT (set, reset: IN BIT;
          q,qb: BUFFER BIT);
END rsff;
```

Designs are made by entities.

VHDL vs Traditional Design Methods

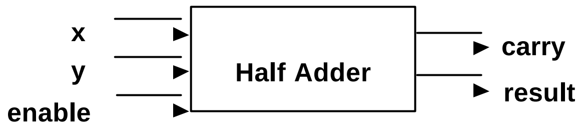
<u>Schematics for RSFF</u> <u>VHDL behavioral Architecture Body of the rfss entity</u>



```
ARCHITECTURE netlist OF rfss IS
  COMPONENT nand2
    PORT (a, b: IN BIT;
             C: OUT BIT);
    END COMPONENT
  BEGIN
     U1: nand2
       PORT MAP(set,qb,q);
     U2: nand2
       PORT MAP(reset,qb,q);
   END netlist;
```

VHDL Design Example

- Problem: Design a single bit half adder with carry and enable.
- Specifications
  - Inputs and outputs are each one bit.
  - When enable is high, result gets x plus y.
  - When enable is high, carry gets any carry of x plus y.
  - Outputs are zero when enable input is low.



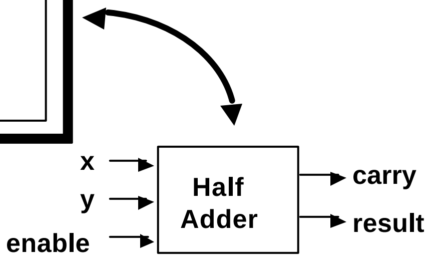
VHDL Design Example
Entity Declaration

- ✍ **As a first step, the entity declaration describes the interface of the component**
  - ✍ **input and output *ports* are declared**

```
ENTITY half_adder IS

    PORT( x, y, enable: IN BIT;
          carry, result: OUT BIT);

END half_adder;
```



VHDL Design Example
Entity Declaration

- ✍ **The term <u>*entity*</u> refers to the VHDL construct in which a component's interface (which is visible to other components) is described.**
- ✍ **The first line in an entity declaration provides the <u>name</u> of the entity.**
- ✍ **The <u>PORT</u> statement indicates the actual interface of the entity: the signals in the component's interface, the direction of data flow for each signal listed, and type of each signal.**
- ✍ **Notice that if signals are of the same mode and type, they may be listed on the same line.**
- ✍ **No semicolon is required before the closing parenthesis in the PORT declaration (or GENERIC declaration, for that matter, which is not shown here).**
- ✍ **The entity declaration statement is closed with the END keyword, and the name of the entity is <u>optionally</u> repeated.**

VHDL Design Example
Behavioral Specification

- ✍ A high level description can be used to describe the function of the adder
  - ✍ Abstract construct ✍ readable/understandable model
  - ✍ Simulation

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

VHDL Design Example
Behavioural Specification (Cont.)

- ✍ The model can then be simulated to verify correct functionality of the component.
- ✍ This level uses abstract construct (such as the IF-THEN_ELSE statement) to make the model more readable and understandable.
  - ✍ sequential statement only inside a process

VHDL Design Example
Data Flow Specification

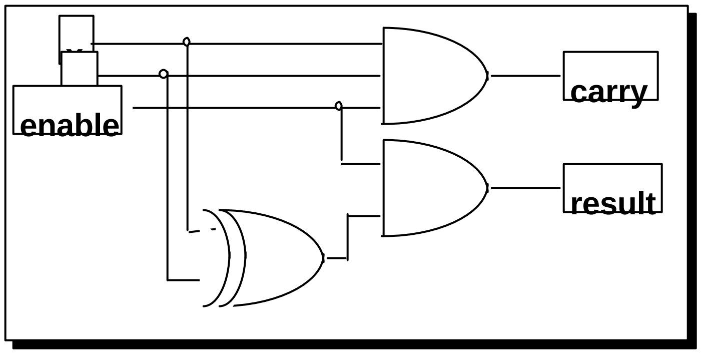
- A second method is to use logic equations to develop a data flow description
  - concurrent signal assignement statements.

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

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

VHDL Design Example
Structural Specification

- As a third method, a structural description can be created from pre described components:



- These gates can be pulled from a library of parts.

VHDL Design Example
Structural Specification (Cont.)

- The previous data flow description maps easily to logic gates.
- These gates can be pulled from many library of parts which may represent specific implementation technologies.
- VHDL structural description may similarly be used to describe the interconnections of high level components:
  - e.g. mult plexors, full adders, microprocessors.

VHDL Design Example
Structural Specification (Cont.)

```
ARCHITECTURE half_adder_c OF half_adder IS

    COMPONENT and2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT and3
      PORT (in0, in1, in2 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT xor2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    FOR ALL : and2 USE ENTITY gate_lib.and2_Nty(and2_a);
    FOR ALL : and3 USE ENTITY gate_lib.and3_Nty(and3_a);
    FOR ALL : xor2 USE ENTITY gate_lib.xor2_Nty(xor2_a);

-- description is continued on next slide
```

VHDL Design Example
Structural Specification (Cont.)

```
-- continuing half_adder_c description

    SIGNAL xor_res : BIT; -- internal signal
    -- Note that other signals are already declared in entity

    BEGIN

      A0 : and2 PORT MAP (enable, xor_res, result);
      A1 : and3 PORT MAP (x, y, enable, carry);
      X0 : xor2 PORT MAP (x, y, xor_res);

    END half_adder_c;
```

Module Outline

- Introduction
- VHDL Design Example
- **VHDL Model Components**
  - Entity Declarations
  - Architecture Descriptions
  - Timing Model
- Basic VHDL Constructs
- Examples
- Summary

VHDL Model Components

- **A complete VHDL component description requires a VHDL *entity* and at least one VHDL *architecture***
  - **The entity defines a component's interface (signals)**
  - **The architecture defines a component's functionality**
- **Several alternative architectures may be developed for use with the same entity**
- **Three areas of description for a VHDL component:**
  - **Behavioural descriptions:** VHDL provides some high level description language construct (e.g. variable, loops, conditionals) to model complex behaviour easily
  - **Data flow descriptions**
  - **Structural descriptions:** VHDL provides mechanisms for describing the structure of component which may be constructed from simpler sub-systems
- **Descriptions with timing and delay:** used for example in simulation; it supports both the concurrency and delay observed in digital electronic systems.

VHDL Model Components (cont.)

Goal:basic working definition for process and signal (not comprehensive)

- **Fundamental unit for component behaviour description is the *process.***
  - **Processes are packaged in architectures and may be explicitly (behavioural) or implicitly (data flow) defined**
- **Primary communication mechanism is the *signal***
  - **Process executions result in new values being assigned to signals**
  - **Signals are then accessible to other processes**
  - **Similarly, a signal may be accessed by a process in another architecture by connecting the signal to ports in the the entities associated with the two architectures**
  - **Example signal assignment statement :**

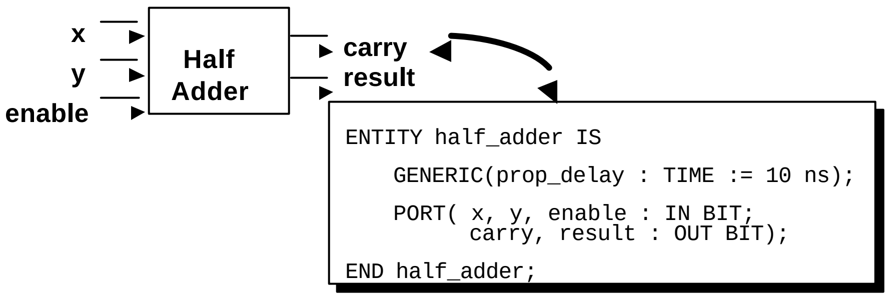
```
Output <= My_id + 10;
```

Entity Declarations

- ✍ A *VHDL Design Entity* could be seen as a twin ***declaration entity – architecture*** **.**
  Both have to be in the same library.

- ✍ **The primary purpose of the *entity declaration* is to declare the signals in the component's interface which is visible to other components**
  - ✍ **The interface signals are listed in the PORT clause**
    - ✍ **In this respect, the *entity-declaration* is akin to the schematic *symbol* for the component**
  - ✍ **Additional entity clauses and statements (e.g. generic) will be introduced later**

Entity Declarations: example



```
ENTITY half_adder IS
    GENERIC(prop_delay : TIME := 10 ns);
    PORT( x, y, enable : IN BIT;
          carry, result : OUT BIT);
END half_adder;
```

Entity Declarations
Port Clause

- PORT clause declares the interface signals of the object to the outside world
- Three parts of the PORT clause
  - Name
  - Mode
  - Data type
  - Signal's initial value (optional)
- Example PORT clause:

```
PORT (signal_name : mode data_type);
```

  - Note port signals (i.e. 'ports') of the same mode and type or subtype may be declared on the same line

```
PORT ( input : IN BIT_VECTOR(3 DOWNTO 0);
       ready, output : OUT BIT );
```

Entity Declarations
Port Clause (cont.)

- The port mode of the interface describes the direction in which data travels with respect to the *component*
- The five available port modes are:
  - In - data comes in this port and can only be read
  - Out - data travels out this port
  - Buffer - data may travel in either direction, but only one signal driver may be on at any one time
  - Inout - data may travel in either direction with any number of active drivers allowed; requires a Bus Resolution Function
  - Linkage - direction of data flow is unknown
- The most used modes are "in" and "out"

Entity Declarations
Generic Clause

- **Generics may be used for readability, maintenance and configuration**
- **Generic clause syntax :**

```
GENERIC (generic_name : type [:= default_value]);
```

  - **If optional default_value is missing in generic clause declaration, it must be present when component is to be used (i.e. *instantiated*)**
- **Generic clause example :**

```
GENERIC (My_ID : INTEGER := 37);
```

  - **The generic *My_ID,* with a default value of 37, can be referenced by any architecture of the entity with this generic clause**
  - **The default can be overridden at component instantiation**

Architecture Bodies

- **Describe the operation of the component**
- **There can be many different architecture for each entity, however for each instantiation of the entity one must be selected**
- **Consist of two parts :**
  - **Declarative part -- includes necessary declarations**
    - **e.g. type declarations, signal declarations, component declarations, subprogram declarations**
  - **Statement part -- includes statements that describe organization and/or functional operation of component**
    - **e.g. concurrent signal assignment statements, process statements, component instantiation statements**

```
ARCHITECTURE half_adder_d OF half_adder IS
    SIGNAL xor_res : BIT;      -- architecture declarative part
    BEGIN                      -- begins architecture statement part
      carry <= enable AND (x AND y);
      result <= enable AND xor_res;
      xor_res <= x XOR y;
    END half_adder_d;
```

Syntax

- **Entity declaration Syntax**

entity identifier **is**
    entity header
end [entity] [identifier **];**

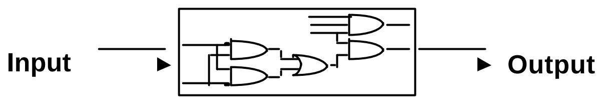
entity_header **: ::= [generic_clause] [port_clause]**
generic_clause : ::= generic (generic_list **);**
port_clause : ::= port (port_list **);**

- **Architecture Syntax**

architecture identifier **of** entity_name **is**
    architecture_declarative_part
    (no variable declarations!)
begin
    architecture_statement_part
end [architecture] [identifier **];**

Structural Descriptions

- **Pre-defined VHDL components are *instantiated* and connected together**
- **Structural descriptions may connect simple gates or complex, abstract components**
- **VHDL provides mechanisms for supporting hierarchical description**
- **VHDL provides mechanisms for describing highly repetitive structures easily**



Behavioral Descriptions

- VHDL provides two styles of describing component behaviour
  - Data Flow: concurrent signal assignment statements
  - Behavioural: *processes* used to describe complex behavior by means of high-level language constructs
    - e.g. variables, loops, if-then-else statements
- A behavioural model may bear little resemblance to system implementation ("black box")
  - Structure not necessarily implied

Input
Behavioural Description
Output

Module Outline

- Introduction
- VHDL Design Example
- VHDL Model Components
- **Basic VHDL Constructs**
  - Data types
  - Objects
  - Predefined operators
  - Sequential statements
  - Concurrent statements
  - Packages and libraries
  - Attributes
- Examples
- Summary

Learning a new language: Lexical Elements

- **VHDL identifier**
  - **letter { "_" | letter | digit }**
- **Comments**
  - **two dashes --**
- **Numbers**
  - **- Integer and real numbers:**
  - 0, 1, 123_456_789, 987E6 -- integer literals
  - 0.0, 0.5, 2.718_28, 12.4E-9 -- real literals
- **Characters & Strings**
  - **'A' "VHDL"**
- **Bit Strings:** Base specifier B stands for binary, O for octal and X for hexadecimal.
  - B"1010110", O"126", X"56"

Data Types

The concept of type is very important when describing data in a VHDL model

The type of a data object defines the set of values that an object can assume as well as the set of operation that can be performed .

VHDL is a strongly typed language: every object may only assume values of its nominated type

The aim of strong typing is to allow detection of errors at an early stage of the design process.

Data Types

The three defined data types in VHDL are

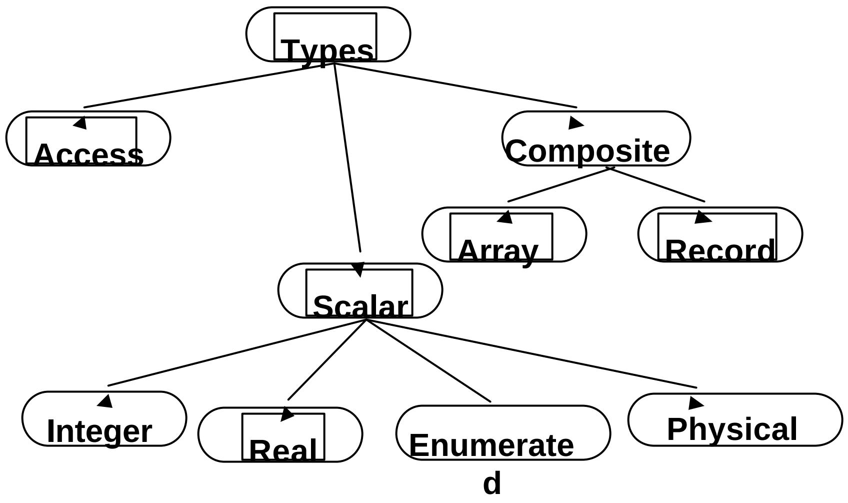
- ✍ **Scalar types:**
  are atomic units of information: a scalar objects can hold
  only a single, indivisible value at time.
- ✍ **Composite types:**
  are arrays and/or records; each object of this data type can hold more than one value
- ✍ **Access types:**
  are akin to pointers in other programming languages

In addition, subtypes and predefined types will also be introduced

Note that VHDL 1076-1987 defined a fourth data type, file, but files were reclassified as objects in VHDL 1076-1993.

Data Types

- ✍ **All declarations of VHDL ports, signals, and variables must specify their corresponding type or subtype**



VHDL Data Types
Scalar Types: Integer

- Integer
  - Assignments within a simulator-specific range
  - Minimum range for any implementation as defined by standard: - 2,147,483,647 to 2,147,483,647
  - Example assignments to a variable of type integer :

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

VHDL Data Types
Scalar Types: Real

- Real
  - Minimum range for any implementation is defined by standard: -1.0E38 to 1.0E38
  - Example: assignments to a variable of type real :

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

VHDL Data Types
Scalar Types : Enumerated

- Enumerated
  - This data type allows a user to specify the list of legal values that a variable or signal of the defined type may be assigned
  - User defined: the user specifies list of possible values ex: type colour is (red, green, blue);
  - As an example, this data type is useful for defining the various states of a FSM with descriptive names.
  - Predefined enumerated :
    - type bit is (‘0’ , ’1’);
    - type boolean is (false, true);
    - type character is ...
    - type severity level is (note, warning, error, failure)
    - The designer first declares the members of the enumerated type.

VHDL Data Types
Scalar Types (Cont.)

Example: declaration and usage of enumerated data type :

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

Note : VHDL is not case sensitive.

VHDL Data Types
Scalar Types (Cont.)

- ✍ **Physical Data Types**
  - ✍ **Used for values which have associated units**
  - ✍ **Range and name must be specified and then the unit**
  - ✍ **Example of physical type declaration :**

```
TYPE resistance IS RANGE 0 TO 10000000
-- no “;” between TYPE and UNIT statement
UNITS
ohm;  -- ohm, unit of the type
Kohm = 1000 ohm;  -- i.e. 1 K?
Mohm = 1000 kohm;  -- i.e. 1 M? ?????
END UNITS;
```

  - ✍ **Time is the only physical type predefined in VHDL standard**
  - ✍ **The line after the Units line states the base unit of the type**

VHDL Data Types
Scalar Types (Cont.)

- ✍ The predefined physical type **time** is important in VHDL, as it is used extensively to specify delays in simulations. Its definition is:

  type time **is range** implementation_defined

  units

  fs;

  ps = 1000 fs;

  ns = 1000 ps;

  us = 1000 ns;

  ms = 1000 us;

  sec = 1000 ms;

  min = 60 sec;

  hr = 60 min;

  end units;
- ✍ To write a value of some physical type, you write the number followed by the unit. For example: 10 mm, 1 rod, 1200 ohm, 23 ns

VHDL Data Types
Composite Types

- Array
  - Used to group elements of the same type (including arrays) into a single VHDL object
  - Range may be unconstrained in declaration
    - Range would then be constrained when array is used
  - The Array is declared in a TYPE statement
  - Multiple dimensional array, ex:

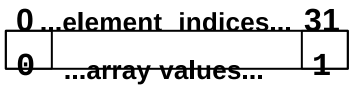
```
type twoDtype is array ( 0 to 4, 0 to 40 ) of real;
```

  - Predefined array types:
    - type string is array (positive range <>) of character
    - type bit-vector is array (natural range <>) of bit;

VHDL Data Types
Composite Types

Example declaration for one-dimensional array :
vector of 32 bits

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

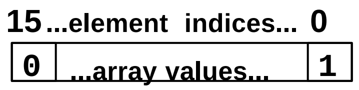


```
VARIABLE X : data_bus;
VARIABLE Y : BIT;
...
Y := X(12);  -- Y gets value of element at index 12
             -- the thirteen element from left
```

VHDL Data Types
Composite Types (Cont.)

- Example one-dimensional array using DOWNTO :

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



```
VARIABLE X : reg_type;
VARIABLE Y : BIT;

Y := X(4);  -- Y gets value of element at index 4
```

- DOWNTO keyword must be used if leftmost index is greater than rightmost index
  - e.g. 'Big-Endian' bit ordering

VHDL Data Types
Composite Types : Records

- Records
  - Used to group elements of possibly different types into a single VHDL object
  - Elements are indexed via field names
  - A record element may be of any data type, including another record.
  - A TYPE declaration is used to define a record.
    The types of a record's elements must be defined before the record is defined
  - There is no semi-colon after the word RECORD. The RECORD and END RECORD keywords bracket the field names. After the RECORD keyword, the record's field names are assigned and their data types is specified

VHDL Data Types
Composite Types: Records

- ✍ **Examples of record declaration and usage : a record type, switch_info, is declared. This example makes use of the binary enumerated type declared previously. Note that values are assigned to record elements by use of the field names.**

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

VHDL Data Types
Access Type

- ✍ **Access**
  - ✍ **Analogous to pointers in other languages**
  - ✍ **Allows for dynamic allocation and deallocation of storage space to the object**
  - ✍ **Useful for implementing queues, fifos buffer , etc. , abstract data structures where the size of of the structure may not be known at the compile time**

VHDL Data Types
Subtypes

- **Subtype**
  - **Allows for user defined constraints on a data type**
    - **e.g. a subtype based on an unconstrained VHDL type**
    - **constraints take the form of range constraints or index constraints**
  - **A subtype may include entire range of base type**
  - **Assignments that are out of the subtype range are illegal**
    - **Range violation detected at run time rather than compile time because only base type is checked at compile time**
  - **Subtype declaration syntax :**

```
SUBTYPE name IS base_type RANGE <user range>;
```

  - **Subtype example :**

```
SUBTYPE first_ten IS INTEGER RANGE 0 TO 9;
```

VHDL Data Types
Subtypes

- There are two predefined numeric subtypes, defined as:
  subtype natural **is** integer **range** 0 **to** highest_integer
  subtype positive **is** integer **range** 1 **to** highest_integer

VHDL Data Types
Summary

- All declarations of VHDL ports, signals, and variables must include their associated type or subtype
- Three forms of VHDL data types are :
  - Access -- pointers for dynamic storage allocation
  - Scalar -- includes Integer, Real, Enumerated, and Physical
  - Composite -- includes Array, and Record
- A set of built-in data types are defined in VHDL standard
  - User can also define own data types and subtypes

VHDL Objects

- Object in a VHDL model: named item with a value of a specified type
- There are four types of objects in VHDL (1076-1993)
  - Constants
  - Variables
  - Signals
  - Files (not discussed)

Declaration and use of constants, variables (and files) are like their use in other programming languages

VHDL Objects

- The scope of an object is as follows :
  - Objects declared in a package are available to all VHDL descriptions that *use* that package
  - Objects declared in an entity are available to all architectures associated with that entity
  - Objects declared in an architecture are available to all statements in that architecture
  - Objects declared in a process are available only within that process
- Simple scoping rules determine where object declarations can be used.
  - This allows the reuse of identifiers in separate entities within the same model without risk of inadvertent errors.

For example, a signal named data could be declared within the architecture body of one component and used to interconnect its underlying subcomponents. The identifier data may also be used again in a different architecture body contained within the same model

VHDL Objects
Constants

- Constant: Name assigned to a specific value of a type.

  VHDL constants are objects whose values can not be changed after they are created.
- Allow for easy update and readability
- Declaration of constant may omit value so that the value assignment may be deferred (deferred constants) (e.g.: in a package body)
  - Facilitates reconfiguration
- Declaration syntax :

  CONSTANT ***constant_name*****:*****type_name*****[:=*****value/expression*****];**
- Declaration examples :

```
CONSTANT PI : REAL := 3.14;
CONSTANT SPEED : INTEGER;
CONSTANT N : INTEGER := 8*SPEED;
```

VHDL Objects
Variables

- ✍ **Provide convenient mechanism for local storage**
  - ✍ **E.g. loop counters, intermediate values**
- ✍ **Scope is process in which they are declared**
  - ✍ **VHDL ‘93 provides for global variables, shared variables (discussion deferred)**
- ✍ **All variable assignments take place immediately**
  - ✍ **No delta or user specified delay is incurred (as in the case of signals).**
  - ✍ **This feature allows the sequential execution of statements within VHDL processes where variables are used as placeholders for temporary data, loop counters, etc.**
- ✍ **Declaration syntax:**

  `VARIABLE` ***`variable_name`*** **`:`** ***`type_name`*** **`[:=`** ***`value`*** **`];`**
- ✍ **Assignment syntax**

  `[label:] name := expression ;`

VHDL Objects
Variables

- ✍ **The initialisation expression in optional: the default initial value assumed by the variable when it is created depends on the type.**
  - ✍ **For scalar types: the default initial value is the leftmost value of the type (for integers is the smallest representable integer).**
- ✍ **Declaration examples :**

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

```
VARIABLE start : time := 0 ns;
VARIABLE finish : time := 0 ns;
-- equivalent to
VARIABLE start, finish : time := 0 ns;
```

VHDL Objects
Signals

- **Used for communication between VHDL components: signals are used to pass information directly between VHDL processes and entities.**
- **Real, physical signals in system often mapped to VHDL signals**
- **ALL VHDL signal assignments require either delta cycle or user-specified delay before new value is assumed: a signal may have a series of of future values with their respective timestamps pending in the signal's waveform.**
- **The need to maintain a waveform results in a VHDL signal requiring more simulator resources than a VHDL variable**
- **Declaration syntax :**

  SIGNAL *signal_name* : *type_name* [:= *value*]
- **Declaration and assignment examples :**

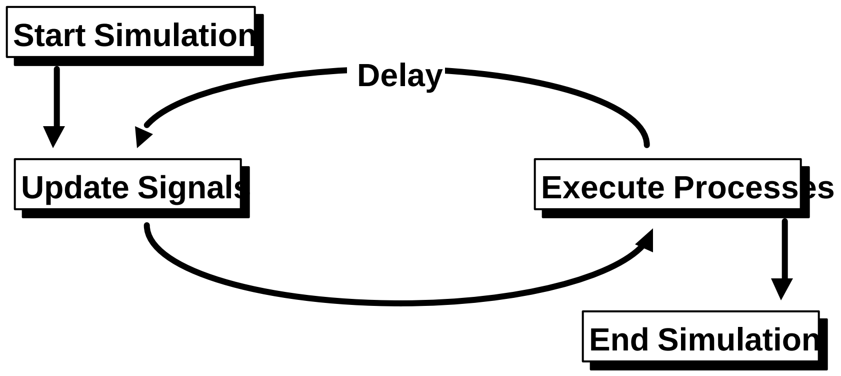
```
SIGNAL brdy : BIT;
brdy <= ‘0’ AFTER 5ns, ‘1’ AFTER 10ns;
```

Timing Model

- **Any signal assignment statement will incur a delay: understanding of how delay works in a process is key to writing and understanding VHDL**
- **Any signal assignment in VHDL is actually a scheduling for a future value to be placed on that signal.**
- **When a signal assignment statement is executed, the signal maintains its original value until the time for the scheduled update to the new value.**
- **Examples:**
- s <= '0' **after** 10 ns;
- s <= '1' **after** 4 ns, '0' **after** 20 ns;

Timing Model

✍ **VHDL uses the following simulation cycle to model the stimulus and response nature of digital hardware**

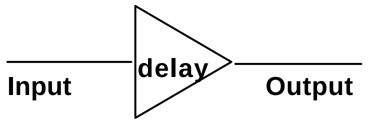


Timing Model

- At the start of a simulation, signals with default values are assigned those values.
- In the first execution of the simulation cycle, all processes are executed until they reach their first *wait* statement. These process executions will include signal assignment statements that assign new signal values after prescribed delays.
- After all the processes are suspended at their respective wait statements, the simulator will advance simulation time just enough so that the first pending signal assignments can be made (e.g. 1 ns, 3 ns, 1 delta cycle).
- After the relevant signals assume their new values, all processes examine their wait conditions to determine if they can proceed. Processes that can proceed will then execute concurrently again until they all reach their respective subsequent wait conditions.
- This cycle continues until the simulation termination conditions are met or until all processes are suspended indefinitely because no new signal assignments are scheduled to unsuspend any waiting processes.

Delay Types

- ✍ All VHDL signal assignment statements prescribe an amount of time that must transpire before the signal assumes its new value
- ✍ This prescribed delay can be in one of three forms:
  - ✍ Transport -- prescribes propagation delay only
  - ✍ Inertial -- prescribes propagation delay and minimum input pulse width
  - ✍ Delta -- the default if no delay time is explicitly specified



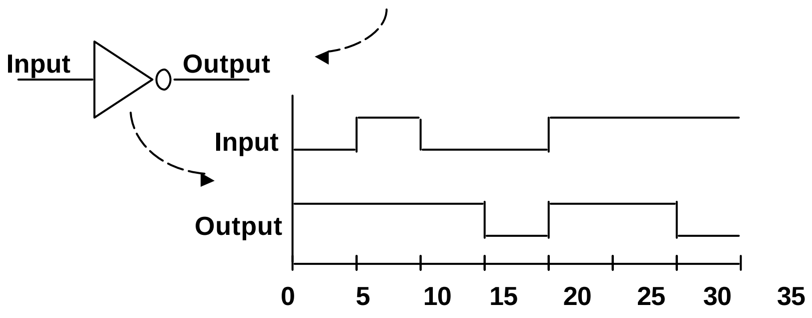
Transport Delay

- ✍ Transport delay is the simplest !
  - ✍ Any change in an input signal value may result in a new value being assigned to the output signal after the specified propagation delay.
  - ✍ No restrictions are specified on input pulse widths.
  - ✍ We are modelling an ideal device with infinite frequency response
- ✍ Transport delay must be explicitly specified
  - ✍ I.e. keyword “TRANSPORT” must be used

Transport Delay

✍ **Signal will assume its new value after specified delay**

```
Inv: process(input) is
begin
-- TRANSPORT delay example
Output <= TRANSPORT NOT Input AFTER 10 ns;
End process Inv;
```



Inertial Delay

Most real electronic circuit don't have infinite frequency response, this give the devices some inertia: it is not appropriate to model them using transport delay.

✍ **The keyword INERTIAL may be used in the signal assignment statement to specify an inertial delay, or it may be left out because inertial delay is used by default in VHDL signal assignment statements which contain "after" clauses.**

✍ **The REJECT construct is optional: if it is not used, the specified delay is then used as both the 'inertia' (i.e. minimum input pulse width requirement) and the propagation delay for the signal.**

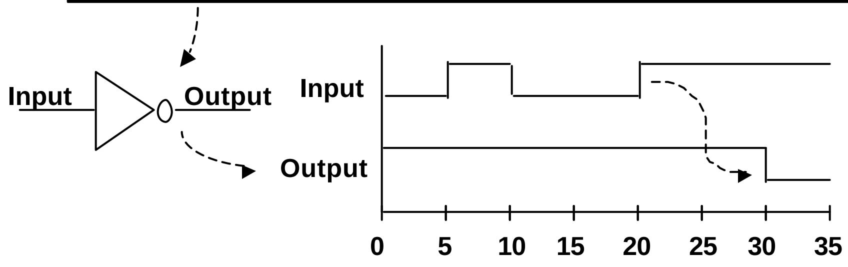
Inertial Delay

- ✍ Provides for specification propagation delay and input pulse width, i.e. ‘inertia’ of output:

```
target <= [REJECT time_expression] INERTIAL waveform;
```

- ✍ Inertial delay is default and REJECT is optional :

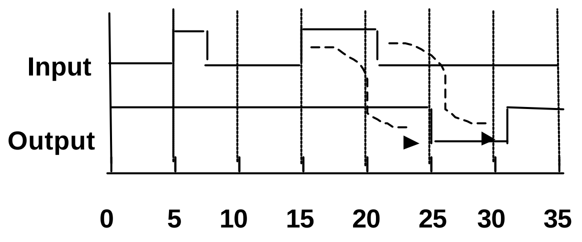
```
Output <= NOT Input AFTER 10 ns;
-- Propagation delay and minimum pulse width are 10ns
```



Inertial Delay (cont.)

- ✍ Example of gate with ‘inertia’ smaller than propagation delay
  - ✍ e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns
  - ✍ Note REJECT construct can only be used with the keyword INERTIAL

```
Output <=  REJECT 5ns INERTIAL NOT Input AFTER 10ns;
```



Inertial Delay

- **The REJECT construct is a new feature to VHDL introduced in the VHDL 1076-1993 standard.**
- **The REJECT construct can only be used with the keyword INERTIAL to include a time parameter that specifies the input pulse width constraint.**
- Prior to this, a description for such a gate would have needed the use of an intermediate signal with the appropriate inertial delay followed by an assignment of this intermediate signal's value to the actual output via a transport delay.

Delta Delay

- **Default signal assignment propagation delay if no delay is explicitly prescribed**
  - **VHDL signal assignments do not take place immediately**
  - **Delta is an infinitesimal VHDL time unit**

```
Output <= NOT Input;
-- Output assumes new value in one delta cycle
```

- **Supports a model of concurrent VHDL process execution**
  - **Order in which processes are executed by simulator does not affect simulation output**

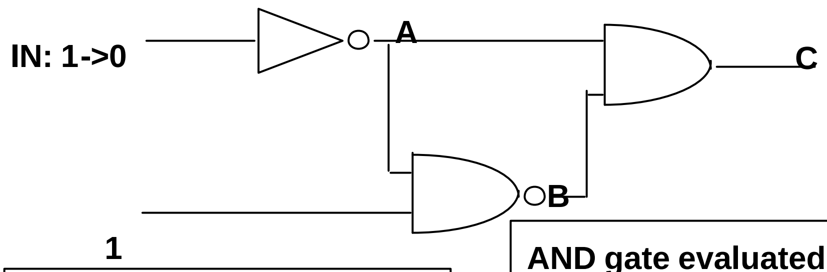
Delta Delay

At various levels of abstraction timing and delay information may not always be included in a VHDL description.

- A delta (or delta cycle) is essentially an infinitesimal, but quantized, unit of time. The delta delay mechanism is used to provide a minimum delay in a signal assignment statement so that the simulation cycle can operate correctly. That is:
  - all active processes can execute in the same simulation cycle
  - each active process will suspend at wait statement
  - when all processes are suspended simulation is advanced the minimum time necessary so that some signals can take on their new values
  - processes then determine if the new signal values satisfy the conditions to proceed from the wait statement at which they are suspended

Delta Delay
An Example without Delta Delay

- What is the behaviour of C? Unpredictable ?



NAND gate evaluated first:

IN: 1->0
A: 0->1
B: 1->0
C: 0->0

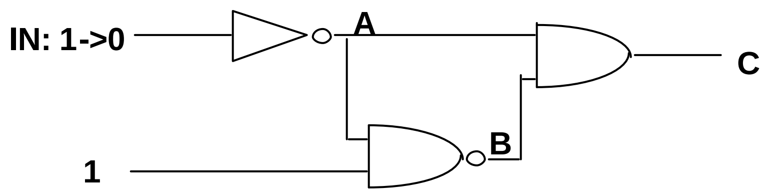
AND gate evaluated first:

IN: 1->0
A: 0->1
C: 0->1 glitch
B: 1->0
C: 1->0

Delta Delay

An Example with Delta Delay

- **What is the behaviour of C?**



Using delta delay scheduling

Time	Delta	Event
0 ns	1	IN: 1->0
		eval INVERTER
	2	A: 0->1
		eval NAND, AND
	3	B: 1->0
		C: 0->1
		eval AND
	4	C: 1->0
1 ns		

Signals and Variables

- **This example highlights the difference between signals and variables**

```
ARCHITECTURE test1 OF mux IS
   SIGNAL x : BIT := '1';
   SIGNAL y : BIT := '0';
BEGIN
   PROCESS (in_sig, x, y)
     BEGIN
       x <= in_sig XOR y;
       y <= in_sig XOR x;
     END PROCESS;
END test1;
```

```
ARCHITECTURE test2 OF mux IS
   SIGNAL y : BIT := '0';
BEGIN
   PROCESS (in_sig, y)
     VARIABLE x : BIT := '1';
   BEGIN
     x := in_sig XOR y;
     y <= in_sig XOR x;
   END PROCESS;
END test2;
```

- **Assuming a 1 to 0 transition on *in_sig*, what are the resulting values for *y* in the both cases?**

Signals and Variables(2)

- Signal assignments require that a delay be incurred before the signals assume their new values: at least a delta delay

```
ARCHITECTURE test1 OF mux IS
   SIGNAL x : BIT := '1';
   SIGNAL y : BIT := '0';
BEGIN
   PROCESS (in_sig, x, y)  -- in_sig  1->0
     BEGIN
       x <= in_sig XOR y; -- in_sig=0, y=0, x=1 actually
              -- x will be 0 after the next delta cycle
       y <= in_sig XOR x; -- in_sig=0, x=1, y=0
                           -- y will be 1 ...
     END PROCESS;
END test1;
```

- The example leads to x and y swapping values in delta time while in_sig has a value of ‘0’

Signals and Variables

- This example highlights the difference between signals and variables

```
ARCHITECTURE test2 OF mux IS
   SIGNAL y : BIT := '0';
BEGIN
   PROCESS (in_sig, y)            --  in_sig  1->0
     VARIABLE x : BIT := '1';
   BEGIN
     x := in_sig XOR y;     -- in_sig=0, y=0, x=0
     y <= in_sig XOR x;     -- in_sig=0, x=0, y=0 actually
                             -- y=0 the next delta cycle
   END PROCESS;
END test2;
```

- This example does not perform the swapping

VHDL Objects
Signals vs Variables

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

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

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0
1+2d	1	1	1	0	1

VHDL Objects
Signals vs Variables (Cont.)

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

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

VHDL Objects
Signals vs Variables (Cont.)

- The first example requires 2 delta cycles and two process executions to arrive at its quiescent state following a change to *a*
- I the second variables are used to achieve the same functionality as in the first one.
  - out_3 is visible only inside the process
  - the order in which the statements appear within the process is important because the two statements are executed sequentially

VHDL Objects
Files

- Files provide a way for a VHDL design to communicate with the host environment
- File declarations make a file available for use to a design
- Files can be opened for reading and writing
  - In VHDL87, files are opened and closed when their associated objects come into and out of scope
  - In VHDL93 explicit `FILE_OPEN()` and `FILE_CLOSE()` procedures were added

VHDL Objects
Files

- ✍ Files may be opened in read or write mode
- ✍ Once a file is opened, its contents may only be accessed sequentially. A detailed description of the use of file objects is beyond this module
- ✍ The package STANDARD defines basic file I/O routines for VHDL types
- ✍ The package TEXTIO defines more powerful routines handling I/O of text files

Operators(overview)

- ✍ Shift Operator
  - ✍ There are three basic forms of shift: Shift Logical Left (SLL), Shift Logical Right (SLR), and Shift Arithmetic Right (SAR).
  - ✍ In addition, some Shift instructions will store shifted bits in the Carry or Overflow bits, to be used in conjunction with other instructions such as conditional branches or rotates.
  - ✍ The "Logical" in "Shift Logical Left" and "Shift Logical Right" refers to the fact that the sign bit of the number is ignored -- the number is considered to be a sequence of bits that is either an unsigned quantity or a non-numeric quantity such as a graphic. When a number is shifted in this manner, the sign bit is not preserved. Logical shifts always bring in '0's in the bit positions being "shifted in".
  - ✍ The "Arithmetic" in "Shift Arithmetic Right" refers to the fact that the sign-bit is replicated as the number is shifted, thereby preserving the sign of the number. This is referred to as "sign extension", and is useful when dividing signed numbers by powers of 2.

Operators

- Operators can be chained to form complex expressions, e.g. :

```
res <= a AND NOT(B) OR NOT(a) AND b;
```

  - Can use parentheses for readability and to control the association of operators and operands
- The list of predefined operators in VHDL :
- Defined precedence levels in decreasing order :
  - Miscellaneous operators -- **, abs, not
  - Multiplication operators -- *, /, mod, rem
  - Sign operator -- +, -
  - Addition operators -- +, -, &
  - Shift operators -- sll, srl, sla, sra, rol, ror
  - Relational operators -- =, /=, <, <=, >, >=
  - Logical operators -- AND, OR, NAND, NOR, XOR, XNOR

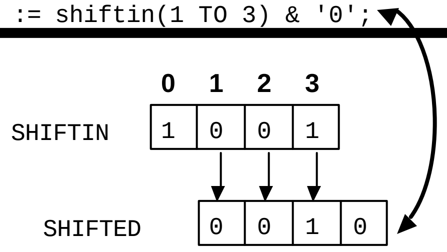
Operators

- The logical and relational operators are similar to those in other languages.
- The concatenation operator ('&') will be discussed in the next slide: it joins two vectors together. Both vectors must be of the same type. The example given in the next slide performs a logical shift left for a four bit array.
- The mod operator returns the modulus of the division and the rem operator returns the remainder.
- For the exponentiation operator ** from the package STD, the exponent must be an integer; no real exponents are allowed. Negative exponents are allowed only with real numbers.

Operators
Examples

✍ **The concatenation operator &**

```
VARIABLE shifted, shiftin : BIT_VECTOR(0 TO 3);
...
shifted := shiftin(1 TO 3) & '0';
```



✍ **The exponentiation operator **.**

```
x := 5**5  -- 5^5, OK
y := 0.5**3 -- 0.5^3, OK
x := 4**0.5 -- 4^0.5, Illegal
y := 0.5**(-2) -- 0.5^(-2), OK
```

VHDL Processes

The process is the key structure in behavioral VHDL modeling.

A process is the only means by which the executable functionality of a component is defined. In fact, for a model to be capable of being simulated, all components in the model must be defined using one or more processes.

- ✍ **Statements within a process are executed sequentially (although care needs to be used in signal assignment statements since they do not take effect immediately).**
- ✍ **Variables are used as internal place holders which take on their assigned values immediately.**
- ✍ **All processes in a VHDL description are executed concurrently. That is, although statements within a process are evaluated and executed sequentially, all processes within the model begin executing concurrently.**

VHDL Processes

- ✍ **A VHDL process statement is used for all behavioral descriptions**
- ✍ **Example simple VHDL process:**

```
ARCHITECTURE behavioral OF clock_component IS
BEGIN
  PROCESS
    VARIABLE periodic: BIT := ‘1’;
       BEGIN
         IF en = ‘1’ THEN
           periodic := not periodic;
         END IF;
         ck <= periodic;
         WAIT FOR 1 us;
  END PROCESS;
END behavioral;
```

VHDL Processes

- ✍ **In the example process given here, the variable *periodic* is declared and assigned the initial condition '1'.**
- ✍ **As long as *en* is '1', *periodic* changes value leading to a potentially new value (called a transaction) to be scheduled for *ck* by the simulator.The process then suspends for one microsecond of simulation time.**
- ✍ **The signal *ck* actually assumes its new value one delta cycle after the process suspends . After the one microsecond suspension, the process once again executes beginning with the IF statement.**
- ✍ **Note that only variables can be declared in a process, and signals (of a process) are used primarily for control (e.g., *en* in this case), inputs into a process, or outputs declared outside from a process (e.g., *ck* in this case).**

Process Syntax

```
[ process_label : ] PROCESS
[( sensitivity_list )]

  process_declarations

BEGIN

  process_statements

END PROCESS [ process_label ] ;
```

NO SIGNAL DECLARATIONS!

Process Syntax

- **The use of *process_label* at the beginning and end of a process is <u>optional</u> but recommended to enhance code readability.**
- **The *sensitivity_list* is optional in that a process may have either a *sensitivity_list*, or it must include WAIT statements. However, a process cannot include both a *sensitivity_list* and WAIT statements. WAIT statements will be covered in a subsequent section.**
- **The *process_declaration* includes declarations for variables, constants, aliases, files, and a number of other VHDL constructs.**
- **The *process_statements* include variable assignment statements, signal assignment statements, procedure calls, wait statements, if statements, while loops, assertion statements, etc.**
- **The use of functions and procedures enables code compaction, enhances readability**

VHDL Sequential Statements

- ✍ Assignments executed sequentially in processes
- ✍ Sequential statements
  - ✍ {Signal, variable} assignments
  - ✍ Flow control
    - ✍ IF <condition> THEN <statements> [ELSIF <statements] [ELSE <statements>] END IF;
    - ✍ FOR <range> LOOP <statements> END LOOP;
    - ✍ WHILE <condition> LOOP <statements> END LOOP;
    - ✍ CASE <condition> IS WHEN <value> => <statements>
      {WHEN <value> => <statements>}
      [WHEN others => <statements>]
      END CASE;
  - ✍ WAIT [ON <signal>] [UNTIL <expression>] [FOR <time>] ;
  - ✍ ASSERT <condition> [REPORT <string>] [SEVERITY <level>] ;

IF Statement: Syntax Rule

```
if_statement <=
[if_label : ]
if boolean_expression then
    { sequential_statement }
{ elsif boolean_expression then
    {sequential_statement }}
[else
   { sequential_statement }]
end if [if_label];
```

IF Statement: Example

```
**if** mode=immediate **then**
        operand:= immed_operand;
**elsif**  opcode=load **or** opcode=add **or** opcode=subtract  **then**
        operand:=memory_operand;
**else**
        operand:=address_operand;
**end if**;
```

CASE Statement: Syntax Rule

```
case_statement <=
[case_label : ]
case expression is
       ( when choices =>
                  {sequential_statement })
               { … }
end case  [case_label];


choices <=  ( simple_expression| discrete_range| others) { |… }
```

CASE Statement: Example

```
Type alu_func is (pass1, pass2, add, subtract);
--
case func is
    when pass1 =>
        result:=operand1;
    when pass2 =>
        result:=operand2;
    when add=>
        result:= operand1+operand2;
    when subtract=>
        result:= operand1-operand2;
end case;
```

NULL Statement

When we need to state that when some condition arises no action is to be performed: the null statement has no effect

Syntax:

null_statement <= [label :] **null** ;

```
CASE opcode IS
    WHEN add =>
                Acc := Acc + operand;
    WHEN subtract =>
                Acc:= Acc - operand;
    WHEN nop =>
                null ;
END CASE;
```

Loop Statements

When we need to write a sequence of statements that is to be repeatedly executed;

- ✍ VHDL has a basic loop statement, which can augmented to form the usual loop statement of other programming languages;
- ✍ The simplest of loop loop repeats a sequence of statement indefinitely

loop_statement <=

[*loop*_label :]

loop

{sequential_statement }

end loop [*loop*_label];

Loop Statement:example

```
entity counter is
    port( clk:  in bit; count : out natural);
end entity counter;
--------------------------------------------------------------------------
architecture behaviour of counter is
begin
    incr: process is
        variable  cont_value: natural := 0;
    begin
        count<= count_value;
        loop
                wait until clk='1';
                count_value:=(count_value +1) mod 16;
                count<=count_value;
        end loop;
    end process incr;
end architecture behaviour;
```

Loop Statement
exit statement

Usually we need to exit the loop when some condition arises.

- ✍ We can use an exit statement to exit to a loop.
- ✍ Syntax

  exit_statement <=

  [label :] exit [loop_label] [when boolean_expression];
- ✍ When this statement is executed, any remaining statements in the loop are skipped: the control is transferred to statement after the “end loop” keyword

Loop Statement:example(2)

Previous counter model including a reset input

```
entity counter is
    port( clk, reset :  in bit; count : out natural);
end entity counter;
------------------------------------------------------------------------
architecture behaviour of counter is
begin
    incr: process is
        variable  cont_value: natural := 0;
    begin
        count<= count_value;
        loop
                loop
                        wait until clk=‘1’ or reset =‘1’;
                        exit when reset =‘1’;
                        count_value:=(count_value +1) mod 16;
```

Loop Statement:example(2)

```
---------------                <--  previous page
                                                         count<=count_value;
                  end loop;
                  count_value := '0';                    --  reset = '1'
                  count<=count_value;
                  wait until reset='0';
          end loop ;
     end process incr;
end architecture behaviour;
```

- When the process is waiting for reset to return to '0', any change s on the clock input are ignored
- when reset changes to '0' process resume: the outer loop repeat
- the exit statement causes control to be transferred out of the inner loop only

Loop Statement
next statement

When next statement is executed, the current iteration of the loop is completed without executing any further statement, and the next iteration is begun.

- Syntax

  next_statement <=

  [label :] next [loop_label] [when boolean_expression];
- Complicated loop/next structures ca be confusing, making the model hard to read, understand and reusing

Loop Statements
while loop

While loop test a condition before each iteration, including the first: if the condition is true, iteration proceeds, if not the loop is terminated

- Syntax

  loop_statement <=

  [*loop*_label :]

  while condition **loop**

  {sequential_statement }

  end loop [*loop*_label];

- In absence of exit statement, the while loop terminates only when condition becomes false.
- We must make sure that the condition eventually become false, or that an exit statement will exit the loop: otherwise for an infinite loop is better use a simple loop statement
- **i**

Loop Statements
for loop

For loop include specification of how many times the body of the loop is to be executed

- Syntax

  loop_statement <=

  [*loop*_label :]

  for identifier **in** discrete_range **loop**

  {sequential_statement }

  end loop [*loop*_label];

  discrete_range <=

  simple_expression (**to** | **downto**) simple_expression ;

- A discrete range can be specified using a discrete type or subtype
- **i**

Loop Statements
for loop

```
for count_value  in 0 to 127 loop
      count_out <= count_value;
wait for  5 ns;
end loop ;
```

Summary of Loop Statements

- Syntax

  loop_statement <=

  [*loop*_label :]

  [while condition | **for** identifier **in** discrete_range] **loop**

  {sequential_statement }

  end loop [*loop*_label];

- **i**

The Wait Statement

- ✍ The *wait* statement causes the suspension of a process statement or a procedure
- ✍ wait [sensitivity_clause] [condition_clause] [timeout_clause] ;
  - ✍ sensitivity_clause ::= ON signal_name { , signal_name }

    ```
    WAIT ON clock;
    ```
  - ✍ condition_clause ::= UNTIL boolean_expression

    ```
    WAIT UNTIL clock = ‘1’;
    ```
  - ✍ timeout_clause ::= FOR time_expression

    ```
    WAIT FOR 150 ns;
    ```

The Wait Statement

- ✍ Wait statements are used to suspend the execution of a process until some condition is satisfied.
- ✍ Processes in VHDL are actually code loops.
  The execution of the last statement in the process is followed by the execution of the first statement in the process and process execution continues until a wait statement is reached. For this reason, every process must have at least one wait statement (a sensitivity is actually an implied wait statement which will be described in the next page of this module).
- ✍ The structure of a wait statement contains optional clauses which can be used in any combination:
  - ✍ The sensitivity_clause: the wait statement will only evaluate its condition clause when there is an event (i.e. a change in value) on at least one of the signals listed in the sensitivity_clause. If no sensitivity_clause is given, the signals listed in the condition_clause constitute an implied sensitivity_clause.

The Wait Statement

- The condition_clause: an expression which must evaluate to TRUE in order for the process to proceed past the wait statement. If no condition_clause is given, a TRUE value is implied when there is an event on a signal in the sensitivity_clause.
- The timeout_clause: specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout_clause is given, STD.STANDARD.TIME'HIGH STD.STANDARD.NOW (effectively until the end of simulation time) is assumed.

- Wait statements assist the modeling process by synchronizing concurrently executing processes, implementing delay conditions into a behavioral model, or establishing event communications between processes. Sometimes, wait statements are used to sequence process execution relative to the simulation cycle.

Equivalent Processes

- “Sensitivity List” vs “wait on”

```
Summation:PROCESS( A, B, Cin)
  BEGIN
    Sum <= A XOR B XOR Cin;
END PROCESS Summation;
```

=

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
        WAIT ON A, B, Cin;
END PROCESS Summation;
```

if you put a sensitivity list in a process, you can’t have a wait statement!

if you put a wait statement in a process, you can’t have a sensitivity list!

VHDL Standard proibition

“wait until” and “wait for”

✍ **What do these do?**

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
    WAIT UNTIL A = ‘1’;
END PROCESS Summation;
```

The first process above executes once at the beginning of the VHDL simulation and then suspends until the input *A* is assigned a value of ‘1’ before it executes again.

This cycle continues in that the process executes every time *A* is assigned a value of ‘1’.

“wait until” and “wait for”

✍ **What do these do?**

The second process also executes once at the beginning of the VHDL simulation

It then waits for 100ns of simulation time and executes again.

This cycle continues with the process executing every 100ns of simulation time.

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
  WAIT FOR 100 ns;
END PROCESS Summation;
```

Mix and Match

- Within an architecture we have two signals and the following process

```
DoSomething:  PROCESS
   BEGIN

      WAIT ON TheirSignal;

      OurSignal <= ‘1’;
      WAIT FOR 10 ns;

      OurSignal <= ‘0’;
      WAIT UNTIL (TheirSignal = ‘1’);

      OurSignal <= ‘1’;

END PROCESS DoSomething;
```

Mix and Match

- We show here how wait statements can be used to synchronize the execution of the process, and also how to sensitize a process to signal changes in another.
- In this example, the process does not execute until *TheirSignal* changes value. Then we schedule a transaction to ‘1’ on *OurSignal* and wait for 10 ns. Note, however, that since there is no AFTER clause in the assignment for *OurSignal*, it will assume its new value in one delta cycle.
- After waiting 10 ns, DoSomething assigns a value of '0' to *OurSignal*; Again, *OurSignal* will actually take on the new value after one delta cycle. Execution of DoSomething is then suspended until *TheirSignal* becomes '1'. When execution resumes, *OurSignal* is set to '1' and the process immediately repeats from the top.

Assertion Statement

- ✍ One of the reasons for writing model of computer systems is to check that a design functions correctly.
- ✍ We can partially test a model applying sample inputs and checking that the outputs meets our expectations.
- ✍ This task can be made easier using assertion statement that check that expected condition are met within the model

Assertion Statement: Syntax Rule

An assertion statement is used to verify a specified condition and to report if the condition is violated

Syntax:

assertion_statement ::=
assert condition
   [**report** expression]
   [**severity** expression]

- ✍ note: **type** severity_level **is** (note, warning, error, failure)
- ✍ An assertion statement can be included anywhere in a process body

Assertion Statement

```
Assert initial_value <= max_value
      report "initial value  too large"
     severity  error ;
```

- during simulation, the simulator reports if an assertion violation arises
- during synthesis the condition is assumed to be true and the circuit is optimized on this information
- during formal verification the condition may be interpreted as a condition to be proven by the verifier

Assertion Statement :example

```
entity SR_flipflop is
      port( S,R:  in bit; Q: out bit);
end entity SR_flipflop ;
--------------------------------------------------------------------------
architecture checking of SR_flipflop is
begin
      set_reset: process (S,R) is
      begin
            assert S='1' nand R='1';
            if S = '1' then
                        Q<= '1';
            end if;
            if R='1' then
                        Q<='0';
            end if;
      end process set_reset;
end architecture checking;
```

Subprograms

- ✍ Similar to subprograms found in other languages
  - ✍ Allow repeatedly used code to be referenced multiple times without rewriting
  - ✍ Break down large blocks of code into small, more manageable parts
- ✍ The use of functions and procedures enables code compaction, enhances readability, and supports hierarchy by allowing code sequences that are used frequently to be defined and subsequently reused easily.

Subprograms

VHDL provides functions and procedures

- ✍ Functions return a value
- ✍ Functions can be used in signal and variable assignment statements:
  - ✍ e.g. A <= abs(-1); -- where abs() returns absolute value
- ✍ Procedures do not have return values
- ✍ Procedures can manipulate the signals or variables passed to them as parameters:
  - ✍ e.g. absolute(A);

  "absolute()" here is a procedure that directly assigns *A* its absolute value

Functions

- Produce a single return value:
  - Functions must have a return value and a return statement
- Called by expressions
  - They are called in statements where a value is needed (e.g. assignment statements, conditional tests).
  - Note that only one value can be returned by a function call.
- Cannot modify the parameters passed to them
- Require a RETURN statement
- A function can have multiple Return statements;
  - the simulator will exit the function when it encounters the first return statement in the execution of the function

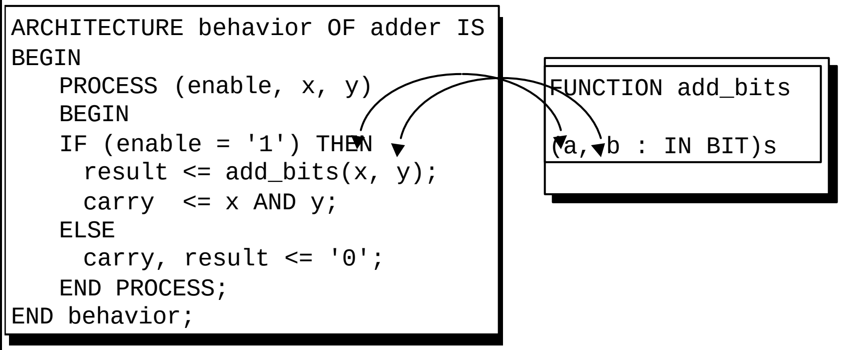
Functions

- Produce a single return value
- Called by expressions
- Cannot modify the parameters passed to them
- Require a RETURN statement

```
FUNCTION add_bits (a, b : IN BIT) RETURN BIT IS
BEGIN  -- functions cannot return multiple values
       RETURN (a XOR b);
END add_bits;
```

```
FUNCTION add_bits2 (a, b : IN BIT) RETURN BIT IS
 VARIABLE result : BIT;  -- variable is local to function
   BEGIN
     result := (a XOR b);
     RETURN result;  -- the two functions are equivalent
END add_bits2;
```

Functions



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

```
FUNCTION add_bits

(a, b : IN BIT)s
```

✍ **Functions must be called by other statements**

✍ **Parameters use positional association**

Functions

✍ **This example illustrates the use of a function call in a signal assignment statement where the value returned by the function add_bits() is assigned to the signal result.**

✍ **Also note that the parameters passed to the function during the call are associated either by position (as in the example above) or by name.**

✍ **In this example, *x* is associated with parameter *a*, and *y* is associated with parameter *b*.**

Procedures

- May produce multiple output values
- Are invoked by statements
- May modify the parameters

```
PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
           SIGNAL temp_result, temp_carry : OUT BIT) IS

BEGIN -- procedures can return multiple values
     temp_result <= (a XOR b) AND en;
     temp_carry <= a AND b AND en;

END add_bits3;
```

- Do not require a RETURN statement

Procedures

- Unlike functions, procedures may modify multiple signals and variables in a single call.
- Procedures can operate on their parameters and are able to make assignments to signals and variables in their parameter lists that are of mode OUT or INOUT.
- A procedure call, therefore, is itself a complete VHDL statement.. May produce multiple output values
- Parameter types and modes must be compatible with the signals in the parameter list during a procedure call.
- Actually, procedure overloading is achieved by defining multiple procedures (or functions, for that matter) with different parameter types to distinguish among them in procedure (or function) calls.

Procedures (Cont.)

```
ARCHITECTURE behavior OF adder
IS
BEGIN
   PROCESS (enable, x, y)
   BEGIN
   add_bits3(x, y, enable,
             result, carry);
   END PROCESS;
END behavior;
```

- ✍ **With parameter passing, it is possible to further simplify the architecture**

```
PROCEDURE add_bits3

(SIGNAL a, b, en : IN BIT;
 SIGNAL temp_result,
temp_carry : OUT BIT)
```

- ✍ **The parameters must be compatible in terms of data flow and data type**

Packages and Libraries

- ✍ **User defined constructs declared inside architectures and entities are not visible to other VHDL components**
  - ✍ **Scope of subprograms, user defined data types, constants, and signals is limited to the VHDL components in which they are declared**
- ✍ **Packages and libraries provide the ability to reuse constructs in multiple entities and architectures**
  - ✍ **Items declared in packages can be *used* (i.e. included) in other VHDL components**
  - ✍ **VHDL provides the package mechanism so that user-defined types, subprograms, constants, aliases, etc. can be defined once and reused in the description of multiple VHDL components.**
  - ✍ **VHDL libraries are collections of packages, entities, and architectures. The use of libraries allows the organization of the design task into any logical partition the user chooses (e.g. component libraries, package libraries to house reusable functions and type declarations).**

Packages

- A package contains a collection of user-defined declarations and descriptions that a designer makes available to other VHDL entities with a *use* clause.
- Packages consist of two parts
  - Package declaration -- contains declarations of objects defined in the package
  - Package body -- contains necessary definitions for certain objects in package declaration
    - e.g. subprogram descriptions
- Examples of VHDL items included in packages :
  - Basic declarations
    - Types, subtypes
    - Constants
    - Subprograms
    - Use clause
  - Signal declarations
  - Attribute declarations
  - Component declarations

Packages
Declaration

- An example of a package declaration :

```
PACKAGE my_stuff IS
   TYPE binary IS ( ON, OFF );
   CONSTANT PI : REAL := 3.14;
   CONSTANT My_ID : INTEGER;
   PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
               SIGNAL temp_result, temp_carry : OUT BIT);
END my_stuff;
```

- Note some items only require declaration while others need further detail provided in subsequent package body
  - for type and subtype definitions, declaration is sufficient
  - subprograms require declarations and descriptions

Packages
Package Body

- **The package body includes the necessary functional descriptions needed for objects declared in the package declaration**
  - **e.g. subprogram descriptions, assignments to constants**

```
PACKAGE BODY my_stuff IS
   CONSTANT My_ID : INTEGER := 2;

   PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
          SIGNAL temp_result, temp_carry : OUT BIT) IS
   BEGIN    -- this function can return a carry
      temp_result <= (a XOR b) AND en;
      temp_carry <= a AND b AND en;
   END add_bits3;
END my_stuff;
```

Packages
Use Clause

- Packages are made visible to a VHDL description through the *use* of the USE clause.
- This statement comes at the beginning of the entity or architecture file and makes the contents of a package available within that file.
- USE clause is analogous to the *include statement* of some programming languages.
- The USE clause can select all or part of a particular package. In the first example above, only the *binary* data type and *add_bits3* procedure are made visible. In the second example, the full contents of the package are made visible by use of the keyword ALL in the use clause

Packages
Use Clause

- **Packages must be made visible before their contents can be used**
  - **The USE clause makes packages visible to entities, architectures, and other packages**

```
-- use only the binary and add_bits3 declarations
USE my_stuff.binary, my_stuff.add_bits3;

... ENTITY declaration...
... ARCHITECTURE declaration ...
```

```
-- use all of the declarations in package my_stuff
USE my_stuff.ALL;

... ENTITY declaration...
... ARCHITECTURE declaration ...
```

Libraries

- **Increasingly <u>complex</u> VLSI technology requires configuration and revision control management.**
- **<u>Efficient</u> design calls for reuse of components when applicable and revision of library components when necessary.**
- **VHDL uses a library system to maintain designs for modification and shared use.**
  - **VHDL refers to a library by an assigned logical name; the host operating system must translate this logical name into a real directory name and locate it.**
  - **The current design unit is compiled into the Work library by default; Work is implicitly available to the user with no need to declare it.**
  - **Similarly, the predefined library STD does not need to be declared before its packages can be accessed via *use* clauses. The STD library contains the VHDL predefined language environment, including the package STANDARD which contains a set of basic data types and functions and the package TEXTIO which contains some text handling procedures.**

Libraries

- Analogous to directories of files
  - VHDL libraries contain analized (i.e. *compiled*) VHDL entities, architectures, and packages
- Facilitate administration of configuration and revision control
  - E.g. libraries of previous designs
- Libraries accessed via an assigned logical name
  - Current design unit is compiled into the *Work* library
  - Both *Work* and *STD* libraries are always available
  - Many other libraries usually supplied by VHDL simulator vendor
    - E.g. proprietary libraries and IEEE standard libraries

Attributes

- Attributes provide information about certain items in VHDL
  - E.g. types, subtypes, procedures, functions, signals, variables, constants, entities, architectures, configurations, packages, components
- General form of attribute use :

```
name'attribute_identifier  -- read as “tick”
```

- VHDL has several predefined, e.g :
  - X'EVENT -- TRUE when there is an event on signal X
  - X'LAST_VALUE -- returns the previous value of signal X
  - Y'HIGH -- returns the highest value in the range of Y
  - X'STABLE(t) -- TRUE when no event has occurred on signal X in the past ‘t’ time

Attributes

- ✍ Attributes can return various types of information.
  - ✍ For example, an attribute can be used to determine the depth of an array, its range, its leftmost index, etc.
  - ✍ Additionally, the user may define new attributes to cover specific situations. This capability allows user-defined constructs and data types to use attributes.
    - ✍ An example of the use of attributes is in assigning information to a VHDL construct, such as board location, revision number, etc.
  - ✍ ,

Attributes
Register Example

- ✍ The following example shows how attributes can be used to make an 8-bit register
- ✍ Specifications :
  - ✍ Triggers on rising clock edge
  - ✍ Latches only on enable high
  - ✍ Has a data setup time of x_setup
  - ✍ Has propagation delay of prop_delay

```
ENTITY 8_bit_reg IS
   GENERIC (x_setup, prop_delay : TIME);
   PORT(enable, clk : IN qsim_state;
        a : IN qsim_state_vector(7 DOWNTO 0);
        b : OUT qsim_state_vector(7 DOWNTO 0));
END 8_bit_reg;
```

- ✍ qsim_state type is being used : includes logic values 0, 1, X, and Z

Attributes
Register Example (Cont.)

- **The following architecture is a first attempt at the register**
- **The use of 'STABLE is to used to detect setup violations in the data input**

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
BEGIN
  PROCESS (clk)
  BEGIN
  IF (enable = '1') AND a'STABLE(x_setup) AND
         (clk = '1') THEN -- clk may<- X,Z
              b <= a AFTER proper_delay;
  END IF;
  END PROCESS;
END first_attempt;
```

- **What happens if *a* does not satisfy its setup time requirement of *x_setup*?**

Attributes
Register Example (Cont.)

- **What happens if *a* does not satisfy its setup time requirement of *x_setup*?**
  - **If the setup requirement is not met, the body of the IF statement will not execute, and the value on *a* will not be assigned to *b*.**
- **T**he process checks that *clk* and *enable* are ‘1’ to store the data, it does not does not consider the possibility that clk may have transitioned to ‘1’ from either ‘X’ or ‘Z’.

Attributes

Register Example (Cont.)

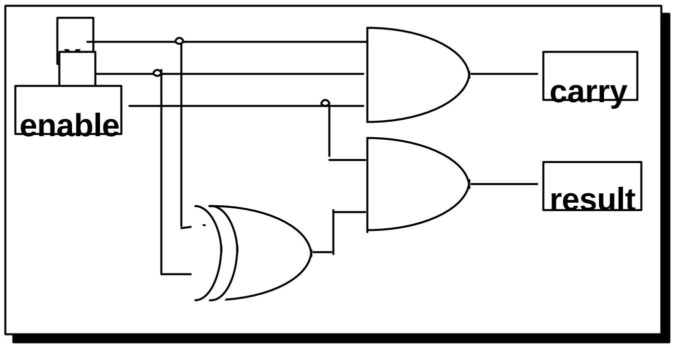
- **The following architecture is a second and more robust attempt**
- **The use of 'LAST_VALUE ensures the clock is rising from a value of ‘0’**

```
ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
  PROCESS (clk)
  BEGIN
  IF (enable = '1') AND a'STABLE(x_setup) AND
         (clk = '1') AND (clk'LAST_VALUE = '0') THEN
              b <= a AFTER delay;
  END IF;
  END PROCESS;
END behavior;
```

- **An ELSE clause could be added to define the behaviour when the requirements are not satisfied**

Structural VHDL

- **Models can be constructed by interconnecting subcomponents**
  - **A *structural* model lists the required subcomponents and prescribes their interconnections**
  - **Akin to a design schematic : the components and interconnection are visible, but internal functions are hidden.**



General Steps to Incorporate VHDL Design Objects

- A VHDL design object to be incorporated into an architecture must *generally* be :
  - declared -- where a local interface is defined
  - instantiated -- where local signals are connected to the local interface
    - Regular structures can be created easily using *GENERATE* statements in component instantiations
  - bound -- where an entity/architecture object which implements it is selected for the instantiated object

General Steps to Incorporate VHDL Design Objects

```
USE work.resources.all;

ENTITY reg4 IS
   PORT (d0, d1, d2, d3 : IN bit;
         clk : IN bit;
         q0, q1, q2, q3 : OUT bit);
END ENTITY;

ARCHITECTURE struct_2  OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
         q : OUT level);
  END COMPONENT reg1;
  CONSTANT enabled : level := '1';
BEGIN
    r0 : reg1 PORT MAP (d=>d0,clk=>clk,q=>q0);
    r1 : reg1 PORT MAP (d=>d1,clk=>clk,q=>q1);
    r2 : reg1 PORT MAP (d=>d2,clk=>clk,q=>q2);
    r3 : reg1 PORT MAP (d=>d3,clk=>clk,q=>q3);
END struct_2;
```

General Steps to Incorporate VHDL Design Objects

```
USE work.resources.all;

ENTITY reg4 IS
   PORT (d0, d1, d2, d3 : IN bit;
         clk : IN bit;
         q0, q1, q2, q3 : OUT bit);
END ENTITY;

ARCHITECTURE struct_2  OF reg4 IS
  COMPONENT reg1 IS
    PORT (d, clk : IN level;
         q : OUT level);
  END COMPONENT reg1;
  CONSTANT enabled : level := '1';
  FOR ALL : reg1 USE work.dff(behav)
     PORT MAP(d=>d, clk=>clk, enable=>enabled, q=>q, qn=>OPEN);
BEGIN
    r0 : reg1 PORT MAP (d=>d0,clk=>clk,q=>q0);
    r1 : reg1 PORT MAP (d=>d1,clk=>clk,q=>q1);
    r2 : reg1 PORT MAP (d=>d2,clk=>clk,q=>q2);
    r3 : reg1 PORT MAP (d=>d3,clk=>clk,q=>q3);
END struct_2;
```

Using Component Declarations and Local Bindings

- **A *component declaration* defines the interface to an idealized local component.**
  - **component declaration may be placed in a package declaration and made visible to the architecture via a USE clause.**
- **Instantiation statements create copies of the component to be plugged into the architecture description by connecting local signals to signals in the component interface.**
- **A binding indication assigns a VHDL entity/architecture object to component instances:**
  - **all *reg1* components will use the *behav* architecture description for the *dff* entity in the *work* library.**

Power of Configuration Declarations

- Configurations can be used to customize the use of VHDL design objects interfaces as needed:
  - Entity name can be different than the component name
  - Entity of incorporated design object may have more ports than the component declaration
  - Ports on the entity declaration of the incorporated design object may have different names than the component declaration

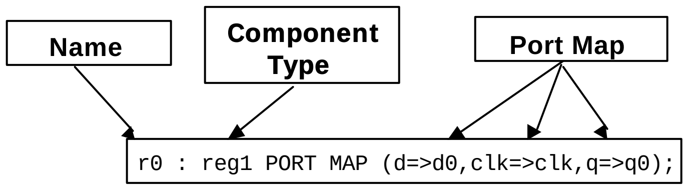
Instantiation Statement

- The instantiation statement connects a declared component to signals in the architecture
- The instantiation has 3 key parts
  - Name -- to identify unique *instance* of component
  - Component type -- to select one of the declared components
  - Port map -- to connect to signals in architecture:how the signals in the interface of the component are assigned to local signals.
  - Optional Generic Map

Instantiation Statement (Cont. 1)

✍ **The example below, shows an instance of reg1 component which has been given the name r0.**

✍ **The PORT MAP section of the instantiation indicates how the signals in the interface of the component are assigned to local signals.**



```
r0 : reg1 PORT MAP (d=>d0,clk=>clk,q=>q0);
```

Instantiation Statement (Cont. 2)

✍ **Note that in this example, we associated each signal to a port on the component by naming the PORT signals explicitly: association by name.**

✍ **VHDL also allows for positional association, but it may only be used if all the signals in a PORT are to be assigned in the order in which they appear in the component declaration (association by position).**

Generic Map

- Generics allow the component to be customized upon instantiation
  - Entity declaration of design object being incorporated provides default values
- The GENERIC MAP is similar to the PORT MAP in that it maps specific values to the generics of the component
- As in PORT MAP signal associations, associations may be made by position or by name.

Generic Map (Cont.)

- If no default values are assigned in the design object’s ENTITY declaration, a GENERIC MAP must be provided in the component’s declaration, instantiation, or binding.

```
USE Work.my_stuff.ALL
ARCHITECTURE test OF test_entity
      SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : my_stuff.and_gate -- component found in package
      GENERIC MAP (tplh=>2 ns, tphl=>3 ns) –- no semicolon
      PORT MAP (S1, S2, S3);
END test;
```

Component Binding Specifications

- A component binding specification provides binding information for instantiated components

  - Single component

    ```
    FOR A1 : and_gate USE binding_indication;
    ```

  - Multiple components

    ```
    FOR A1, A2 : and_gate USE binding_indication;
    ```

  - All components

    ```
    FOR ALL : and_gate USE binding_indication;
    ```

    -- All components of this type are effected
  - Other components

    ```
    FOR OTHERS : and_gate USE binding_indication;
    ```

    -- i.e. for components that are not otherwise specified: the keyword OTHERS selects all components not yet configured.

Binding Indication

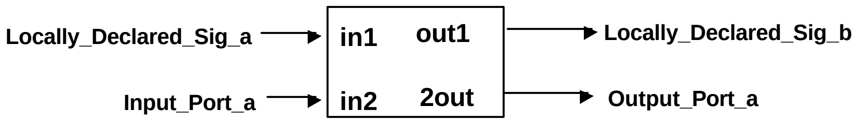
- The *binding indication* identifies the design object to be used for the component
  - The binding indication identifies the design entity (i.e. entity/architecture object or configuration declaration) to bind with the component and maps the two interfaces together. That is, binding indication associates component instances with a particular design entity

```
FOR ALL : reg1 USE work.dff(behav);
```

- Binding indication may also include a PORT MAP and/or GENERIC MAP to customize the component(s)

Rules for Actuals and Locals

- **An *actual* is either <u>signal</u> declared within the architecture or a <u>port</u> in the entity declaration**
- **Locals are defined as the ports of the *component*.**
  - **A port on a *component* is known as a *local* and must be matched with a compatible *actual***
- **VHDL has two main <u>restrictions</u> on the association of *locals* with *actuals***
  - **Local and actual must be of same data type**
  - **Local and actual must be of compatible modes**
    - **Locally declared signals do not have an associated mode and can connect to a local port of any mode**



Rules for Actuals and Locals (Cont.)

- **The local and actual must be of compatible modes.**
- **An actual of mode IN (i.e. a PORT of mode IN since locally declared signals do not have a mode) can only be associated with a local of mode IN, and an actual of mode OUT ((i.e. a PORT of mode OUT) can only be associated with a local of mode OUT.**
- **A local INOUT port is generally associated with an INOUT or OUT local.**
- **Locally declared signals can be connected to locals of any mode, but care must be exercised to avoid illegal connections (e. g. a single actual connected to two mode OUT locals).**

Generate Statement

Structural descriptions of large, but highly regular structures can be tedious.

- VHDL provides the GENERATE statement to create well-patterned structures easily
  - Some structures in digital hardware are repetitive in nature (e.g. RAMs, adders): some common examples include the instantiation and connection of multiple identical components such as half adders to make up a full adder, or exclusive or gates to create a parity tree.

Generate Statement

A VHDL GENERATE statement can be used to include as many concurrent VHDL statements (e.g. component instantiation statements) as needed to describe a regular structure easily

- Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement
  - Specifically, component instantiations may be made within GENERATE bodies
- VHDL provides two different schemes of the GENERATE statement, the FOR-scheme and the IF-scheme.

Generate Statement
FOR-Scheme

- The FOR-scheme is reminiscent of a FOR loop used for sequence control in many programming languages.
- The FOR-scheme generates the included concurrent statements the assigned number of times.
- In the FOR-scheme, all of generated concurrent statements must be the same: all objects created are similar .
- The loop variable is created in the GENERATE statement and is undefined outside that statement (i.e. it is not a variable or signal visible elsewhere in the architecture).The GENERATE parameter must be discrete.
- Loop cannot be terminated early

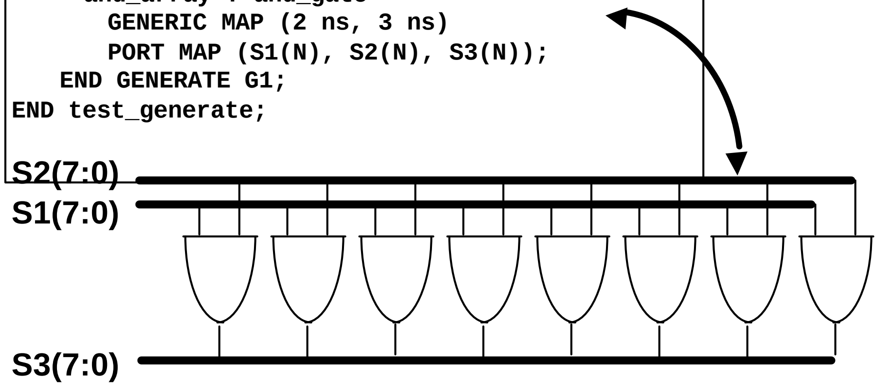
Generate Statement
FOR-Scheme

- The syntax for the FOR-scheme GENERATE statement.
  - The loop variable in this case is N.
  - The range can be any valid discrete range.
  - After the GENERATE keyword, the concurrent statements to be generated are stated.
  - The GENERATE statement is closed with END GENERATE

```
name : FOR N IN 1 TO 8 GENERATE
   concurrent-statements
END GENERATE name;
```

FOR-Scheme Example

```
-- this uses the and_gate component from before
ARCHITECTURE test_generate OF test_entity IS
     SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNTO 0);
BEGIN
   G1 : FOR N IN 7 DOWNTO 0 GENERATE
     and_array : and_gate
       GENERIC MAP (2 ns, 3 ns)
       PORT MAP (S1(N), S2(N), S3(N));
   END GENERATE G1;
END test_generate;
```



Generate Statement
IF-Scheme

- **The second form of the GENERATE statement: IF-scheme.**
- **This scheme allows for conditional generation of concurrent statements.**
- **One difference between this scheme and the FOR-scheme:all the concurrent statements generated do not have to be the same.**
- **While this IF statement may seem reminiscent to the IF-THEN-ELSE constructs in programming languages, note that the GENERATE IF-scheme does not provide ELSE or ELSIF clauses.**

Generate Statement IF-Scheme

- **Syntax of the IF-scheme GENERATE statement.**
- **The boolean expression of the IF statement can be any valid boolean expression. Allows for conditional creation of components**
- **Cannot use ELSE or ELSIF clauses with the IF-scheme**

```
name : IF (boolean expression) GENERATE
    concurrent-statements
END GENERATE name;
```

IF-Scheme Examples

```
ARCHITECTURE test_generate OF test_entity
     SIGNAL S1, S2, S3: BIT_VECTOR(7 DOWNTO 0);
BEGIN
   G1 : FOR N IN 7 DOWNTO 0 GENERATE

     G2 : IF (N = 7) GENERATE
       or1 : or_gate
         GENERIC MAP (3 ns, 3 ns)
         PORT MAP (S1(N), S2(N), S3(N));
     END GENERATE G2;

     G3 : IF (N < 7) GENERATE
       and_array : and_gate
         GENERIC MAP (2 ns, 3 ns)
         PORT MAP (S1(N), S2(N), S3(N));
     END GENERATE G3;

   END GENERATE G1;
END test_generate;
```

IF-Scheme Examples

- **The example here uses the IF-scheme GENERATE statement to make a modification to the and_gate array such that the seventh gate of the array will be an or_gate.**