VHDL:

Overview

Goals

- To obtain a general appreciation of what VHDL is as a hardware description language, and how it is used in the hardware design process.
- To get a comprehensive overview about the VHDL language.
- To become familiar with the main features of VHDL, particularly different design units.
- To get familiar with the three VHDL design representations.
- To introduce the use of package textio in reading and writing text files in VHDL.
- To understand the role of testbenches in gate level simulation.
- To develop a VHDL modeling template to follow.

Defining a Digital Design

There are several approaches for defining a digital design:

1. **Boolean equations**
   - This describes function of gates and 1-bit storage elements.
   - Each flip-flop or gate requires an equation.
   - It is not practical for a design that contains thousands of gates.
2. **Schematics**
   - This uses previously-defined designs in a hierarchical fashion.
   - This method is graphical and is easily understood.
   - This method is also limited to small designs. It becomes incomprehensible after 10,000 gates.
3. **Hardware description language (VHDL or Verilog)**

What is VHDL?

1. VHDL is the **VHSIC (Very High Speed Integrated Circuit) HDL (Hardware Description Language)**.
2. VHDL was initiated in 1981 by the USA DoD to address hardware life-cycle design problem.
3. VHDL has an ADA based syntax with several extensions, which make it useful for the modeling of digital circuits.
4. VHDL describes the behavior and structure of electronic systems.
5. IEEE published the first VHDL standard 1076-1987, followed by three updates in 1993, 2000, and 2008.
6. IEEE standard is defined by the language reference manual **(LRM)**.

The Different Uses of VHDL

1. Design representation and entry
2. Design documentation
3. Design simulation
4. Design synthesis
5. Design verification
6. Describe a design using different abstraction representations

Why VHDL?

- High-level programming languages are sequential in nature. Hardware is parallel in nature.
- Time is not embedded in other high-level languages. It is hard to simulate delays.
- VHDL is event driven by design. This speeds simulations.

Hardware Description

VHDL deals with a given component from two points of view:

- **External view** of component as seen by others.

- **Multiple internal views** describing component function.
  - One view might describe **structure** of the design.
  - Another view might describe **function** of the design.

VHDL Primary Design Units

ENTITY

CONFIGURATION

PACKAGE

VHDL Secondary Design Units

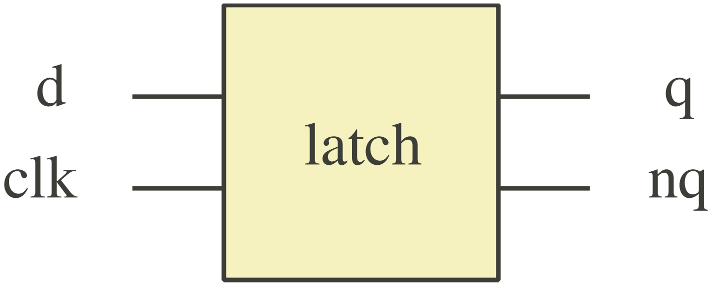
ARCHITECTURE

PACKAGE BODY

Entity Declaration

Example: 1 (Class) *Write entity declaration for a 1-bit latch (level-triggered).*

```
ENTITY latch IS
   PORT( d, clk:  IN bit;
         q, nq:  OUT bit);
END ENTITY latch;
```



Entity Declaration (Cont.)

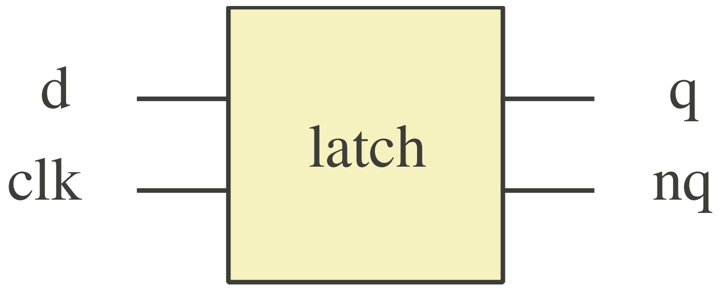
1. Reflects external (user's) view of design.
2. Associates a **name** to the design.
3. Defines the I/O **ports** (external interface).
4. Specifies **mode** of each port (i.e., direction of data flow).
5. Specifies **type** of data moving through the ports.
6. Specifies **generics (parameters**) of entity such as bus width, delay, clock speed, etc. (not shown in previous example, will be explained later).
7. Does not explain how the entity works.

Architecture Declaration

Example: 2 (Class) *Write entity and architecture declarations for a 1-bit latch.*

```
ENTITY latch IS
   PORT( d, clk:  IN bit;
         q, nq:  OUT bit);
END ENTITY latch;

ARCHITECTURE behav OF latch IS
BEGIN
   p1:  PROCESS (d, clk) IS
   BEGIN
      IF clk = ’1’ THEN
         q <= d; nq <= NOT(d);
      END IF;
   END PROCESS p1;
END ARCHITECTURE behav;
```



Architecture Declaration (Cont.)

1. Architecture reflects internal (programmer's) view of design.
2. Architecture explains how the entity works or how it is constructed.
3. Each entity could have one or more **ARCHITECTURES** associated with it.
4. All these architectures must be associated with the same entity.
5. Each architecture is associated with only an entity.

Comments

- VHDL provides only line comments.
- Comments start with double dashes (−−) and terminate at the end of line

Example: 3 (Individual) *Here is an example of a VHDL comment.*

```
y <= data_in;    -- Assign data_in to y
```

VHDL Design Representations

- This is the **internal** view of a design.
- There are three different design representations:
  1. **Behavioral** description (algorithmic, functional): what it does.
  2. **Structural** description how is it constructed.
  3. **Dataflow** or **Register Transfer Level Description (RTL)**.
- A complete design will usually be described using a **mixed description** mixing any of the previous three representations.

Behavioral Description

Most abstract level of design description and is characterized by:

- Contains **PROCESS** statements.
- Statements in a process are executed sequentially.
- Processes include sequential statements like function, procedure, control flow, variables, etc.
- Sequential statements exist inside **PROCESS** body only.
- **PROCESS** itself is a **concurrent statement**; i.e., we could have several processes in an architecture, all running simultaneously.
- Used to:
  1. Describe complex designs
  2. Describe initial design of a system
  3. Documentation

Process Statements with Sensitivity List

Example: 4 (Class) *Describe a 2-input AND gate using a process with a sensitivity list.*

```
ENTITY and2 IS
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY and2;

ARCHITECTURE and2 OF and2 IS
BEGIN
   p1:  PROCESS (a, b) IS BEGIN
      c <= a AND b;
   END PROCESS p1;
END ARCHITECTURE and2;
```

Process Statements with Sensitivity List (Cont.)

1. Signals `a` and `b` are in the sensitivity list of the process.
2. Process p1 is executed once at start of simulation then it suspends until `a` or `b` change. It then starts again from the beginning.
3. Value for `c` is held between executions of the process.

- A process is either **active** or **suspended**.
- At start of simulation a process is activated.
- After executing all statements, process suspends.
- When a process is suspended, values of its variables are not lost.
- A process is activated on any change in the signals of **sensitivity list**.
- A process with sensitivity list cannot have **WAIT** statements.

Process Statements with Wait Statement

Example: 5 (Class) *Describe a 2-input AND gate using a process with a wait statement.*

```
ENTITY and2 IS
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY and2;

ARCHITECTURE and2w OF and2 IS
BEGIN
   p2:  PROCESS IS BEGIN
      c <= a AND b;
      WAIT ON a, b;
   END PROCESS p2;
END ARCHITECTURE and2w;
```

Example: 6 (Home) *Describe a 1-bit latch using a process with a WAIT statement.*

Process Statements with Wait Statement (Cont.)

1. Signals `a` and `b` are in the argument of the **WAIT** statement.
2. Process p2 is executed once at start of simulation then it suspends when it reaches the **WAIT** statement until `a` or `b` change. It then starts again from the beginning.
3. Value for `c` is held between executions of the process.

- At start of simulation a process is activated.
- Upon reaching the **WAIT** statement, process suspends.
- When a process is suspended, values of its variables are not lost.
- A process is activated on any change in the arguments of the **WAIT**.
- When a process starts again, it starts with statements after **WAIT**.
- A process with **WAIT** statement cannot have a sensitivity list.

WAIT Statements

- There are three forms of WAIT statements:

  1. `WAIT FOR time`: suspends a process for the specified amount of time. The time can be specified either explicitly or as an expression, which evaluates to a value of type time.

     Example: `WAIT FOR 20 ns;`

  2. `WAIT UNTIL boolean_condition`: suspends a process until the given condition becomes true due to a change in any of the signals involved in the condition.

     Example: `WAIT UNTIL clk = '1' and clk'event;`

  3. `WAIT ON signal_list`: suspends a process until an event is detected on any of the signals listed.

     Example: `WAIT ON a, b;`

WAIT Statements (Cont.)

- A complex wait statement contains a combination of two or three different forms of wait statements.

  Example: `WAIT ON a UNTIL clk = '1';`

  Example: `WAIT UNTIL clk = '1' FOR 20 ns;`

- `WAIT;` without an argument suspends the process forever.

WAIT Statements (Cont.)

Example: 7 (Home) *Check the equivalence of the following three processes.*

```
p1:  PROCESS (a, b) IS
BEGIN


         .
         .
         .
END PROCESS p1;
```

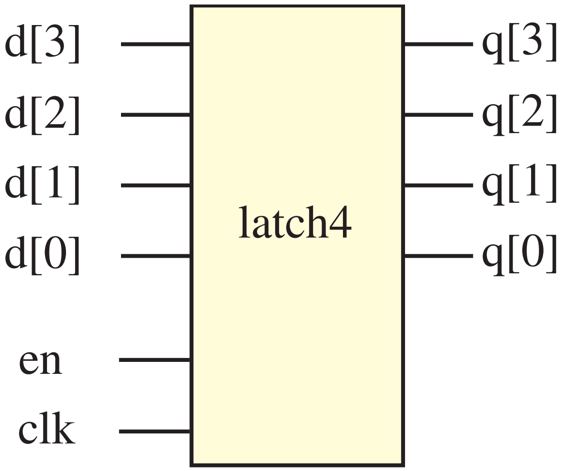
```
p2:  PROCESS IS
BEGIN
   WAIT on a, b;
         .
         .
         .
END PROCESS p2;
```

```
p3:  PROCESS IS
BEGIN
      .
      .
      .
   WAIT on a, b;
END PROCESS p3;
```

Process Statements with Wait Statement (Cont.)

Example: 8 (Class) *Design a 4-bit latch with enable using behavioral description.*

```
ENTITY latch4 IS
   PORT( d:   IN bit_vector (3 DOWNTO 0);
         en, clk:   IN bit;
         q:   OUT bit_vector (3 DOWNTO 0));
END ENTITY latch4;
ARCHITECTURE behav OF latch4 IS
BEGIN
   p2:  PROCESS IS
      VARIABLE x:  bit_vector (3 DOWNTO 0);
   BEGIN
      IF en = '1' AND clk = '1' THEN
         x := d;
      END IF;
      q <= x;
      WAIT ON d, en, clk;
   END PROCESS p2;
END ARCHITECTURE behav;
```



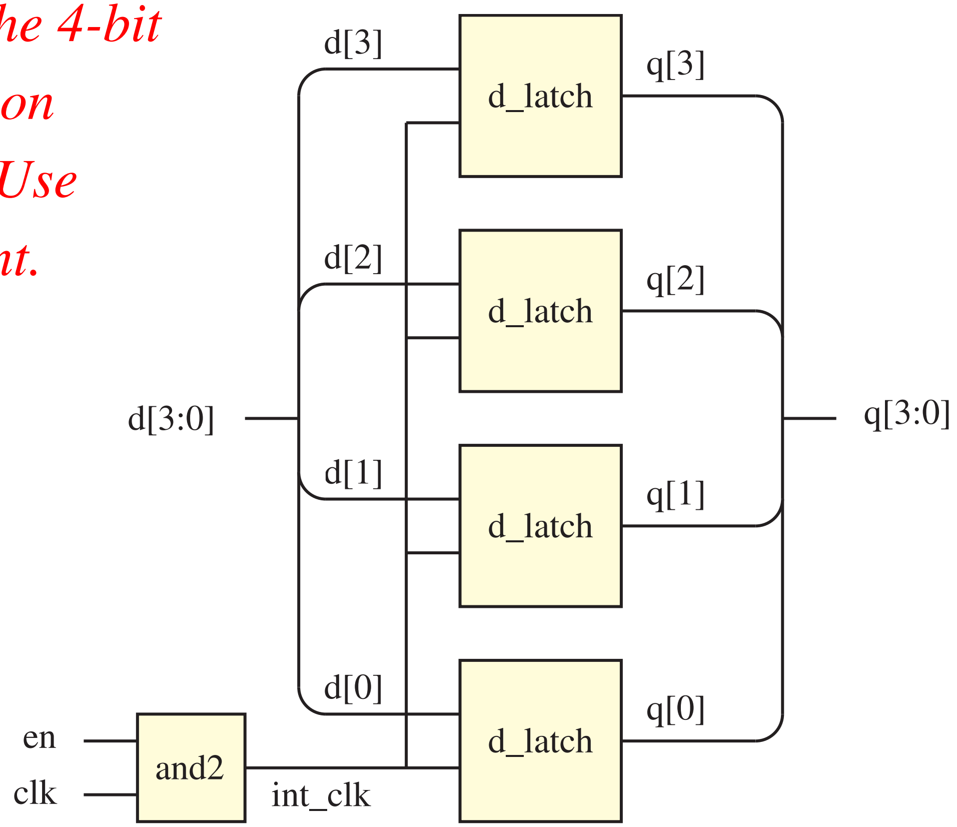
Structural Description

Explains how an entity is built but not how it works and is characterized by:

- A structural way of modeling mimicking schematic block diagrams.
- Describes a circuit in terms of components and their interconnections.
- Components can be described as structural, a behavioral, or dataflow model.
- At the lowest hierarchy each component is described as a behavioral model, using the basic logic operators defined in VHDL.
- Each component is supposed to be defined earlier (e.g., in package).
- The architecture declarative part declares: **components** being used and **internal signals** defining the nets that interconnect components.
- The architecture body: **instantiates** components using **PORT MAP** statements and uniquely **labels** multiple instances of the same component.
- **PORT MAP** itself is a concurrent statement; i.e., we could have several PORT MAP statements in an architecture, all running simultaneously.

Structural Description (Cont.)

Example: 9 (Class) *Rebuild the 4-bit latch using structural description instead of the behavioral one. Use* ***latch*** *as the building component.*



Structural Description using Architecture Configurations

Example 9: (Cont.)

```
ENTITY latch4 IS
   PORT( d:  IN bit_vector (3 DOWNTO 0);
         en, clk:  IN bit;
         q:  OUT bit_vector (3 DOWNTO 0));
END ENTITY latch4;
ARCHITECTURE struct OF latch4 IS
   COMPONENT latch IS
      PORT(d, clk:  IN bit; q, nq:  OUT bit);
   END COMPONENT latch;
   FOR ALL: latch USE ENTITY WORK.latch (behav);
   COMPONENT and2 IS
      PORT(a, b:  IN bit; c:  OUT bit);
   END COMPONENT and2;
   FOR gate:  and2 USE ENTITY WORK.and2 (and2);
   SIGNAL int_clk:  bit;
BEGIN
             ⋮
```

Structural Description using Positional Signal Association

Example 9: (Cont.)

```
            ⋮
   bit3:  latch
      PORT MAP (d(3), int_clk, q(3));
   bit2:  latch
      PORT MAP (d(2), int_clk, q(2));
   bit1:  latch
      PORT MAP (d(1), int_clk, q(1));
   bit0:  latch
      PORT MAP (d(0), int_clk, q(0));
   gate:  and2
      PORT MAP (en, clk, int_clk);
END ARCHITECTURE struct;
```

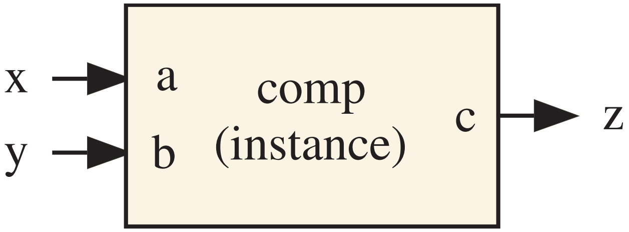
Example: 10 (Home) *Can you make an observation about above port associations?*

Structural Description using Named Signal Association

Example: 11 (Class)

```
            ⋮
   bit3:  latch
      PORT MAP (d => d(3), clk => int_clk, q => q(3), nq => OPEN);
   bit2:  latch
      PORT MAP (clk => int_clk, d => d(2), q => q(2));
   bit1:  latch
      PORT MAP (q => q(1), d => d(1), clk => int_clk);
   bit0:  latch
      PORT MAP (d => d(0), q => q(0), clk => int_clk);
   gate:  and2
      PORT MAP (a => en, b => clk, c => int_clk);
END ARCHITECTURE struct;
```

Ports and Port Maps



```
COMPONENT comp IS
   PORT( a, b:  IN bit;
         c:  OUT bit);
END COMPONENT comp;
            ⋮
   SIGNAL x, y, z:  bit;
            ⋮
instance:  comp PORT MAP (a => x, b => y, c => z);
-- OR
instance:  comp PORT MAP (x, y, z);
-- OR
instance:  comp PORT MAP (x, y, c => z);
```

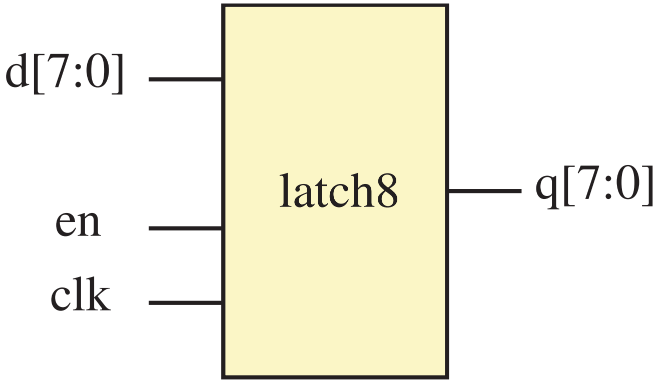
Note that we can mix positional and named signal association provided that the positional signal associations precede the named ones.

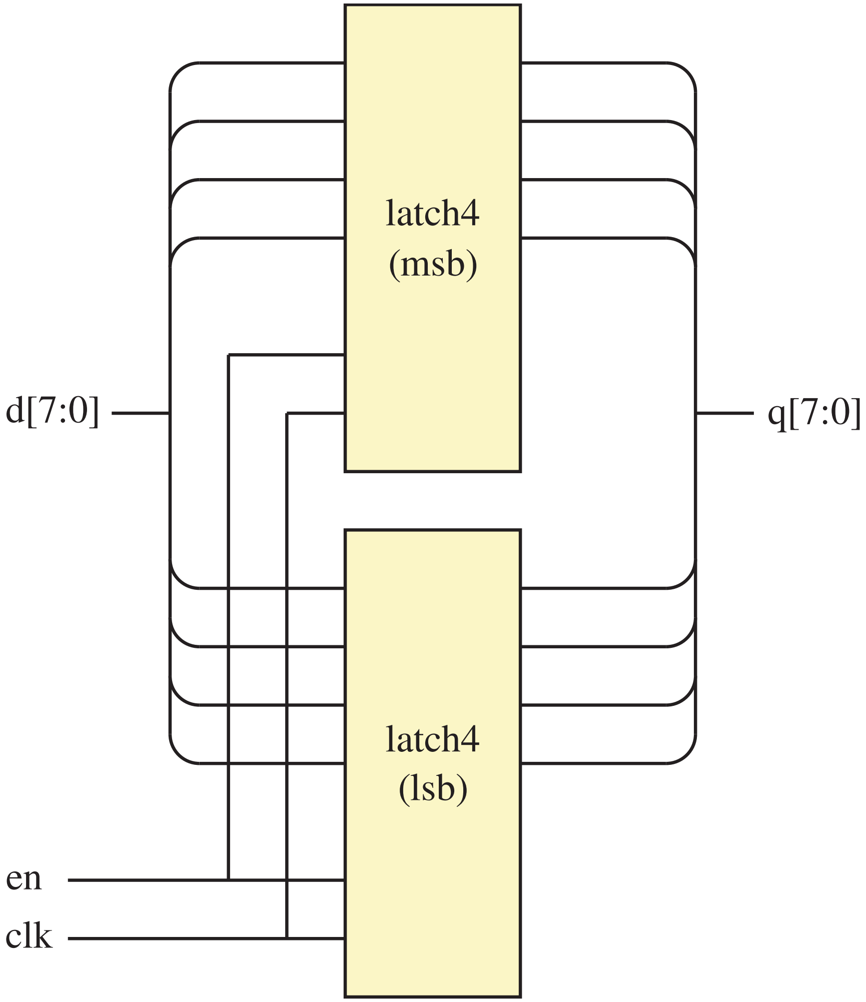
Configuration

- A design may consist of many entities with several architectures.
- The **CONFIGURATION** specifies the exact set of entities and architectures used in a particular simulation or synthesis run.
- A configuration specifies two things:
  1. the design entity used in place of each component instance.
  2. the architecture to be used for each design entity.
- Associating a design entity and architecture with a component instantiation is called **binding**.
- There are two ways for configuring a component:
  1. using **architecture configuration**
  2. using **configuration declaration**

Structural Description using a Configuration Declaration

Example: 12 (Class) *Build an 8-bit latch using latch4 using a configuration declaration.*





Structural Description using a Configuration Declaration (Cont.)

Example 12: (Cont.)

```
ENTITY latch8 IS
   PORT( d:  IN bit_vector (7 DOWNTO 0);
         en, clk:  IN bit;
         q:  OUT bit_vector (7 DOWNTO 0));
END ENTITY latch8;

ARCHITECTURE struct OF latch8 IS
   COMPONENT latch4 IS
      PORT( d:  IN bit_vector (3 DOWNTO 0);
            en, clk:  IN bit;
            q:  OUT bit_vector (3 DOWNTO 0));
   END COMPONENT latch4;
BEGIN
   msb:  latch4 PORT MAP (d(7 DOWNTO 4),
            en, clk, q(7 DOWNTO 4));
   lsb:  latch4 PORT MAP (d(3 DOWNTO 0),
            en, clk, q(3 DOWNTO 0));
END ARCHITECTURE struct;
             ⋮
```

Structural Description using a Configuration Declaration (Cont.)

Example 12: (Cont.)

```
            ⋮
CONFIGURATION struct_conf OF latch8 IS
   FOR struct
      FOR msb:  latch4
         USE ENTITY WORK.latch4 (struct);
      END FOR;

      FOR lsb:  latch4
         USE ENTITY WORK.latch4 (struct);
      END FOR;
   END FOR;
END CONFIGURATION struct_conf;
```

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		

Register Transfer Level (RTL)/Dataflow Description

Describes the flow of data in a concurrent style of modeling and is characterized by:

- Nearest to RTL description of a circuit, giving an idea about its architecture.
- Operations at each clock cycle are explicitly defined.
- Uses **concurrent signal assignment** statements.
- Concurrent statements are not included in a **PROCESS**.
- Unlike behavioral modeling, the order of statements is not important.
- Unlike structural modeling, does not use **PORT MAP** statements to describe the flow of data from input to output through components.
- **PROCESS** and **PORT MAP** statements themselves are **concurrent statements** within a dataflow description.

RTL (Dataflow) Description (Cont.)

Example: 13 (Class) *Model a half-adder using dataflow description.*

```
ENTITY half_adder IS
   PORT( a, b:  IN bit;
         s, c:  OUT bit);
END ENTITY half_adder;

ARCHITECTURE dataflow OF half_adder IS
BEGIN
   s <= a XOR b;
   c <= a AND b;
END ARCHITECTURE dataflow;
```

Note signals `s` and `c` will be evaluated **concurrently**.

RTL (Dataflow) Description (Cont.)

Example: 14 (Class) *Using a dataflow description, model a 4-bit equality comparator.*

```
ENTITY eqcomp4 IS
   PORT( a, b:  IN bit_vector (3 DOWNTO 0);
         equals:  OUT bit);
END ENTITY eqcomp4;

ARCHITECTURE dataflow OF eqcomp4 IS
BEGIN
   equals <= ’1’ WHEN (a = b) ELSE
             ’0’;
END ARCHITECTURE dataflow;
```

Packages

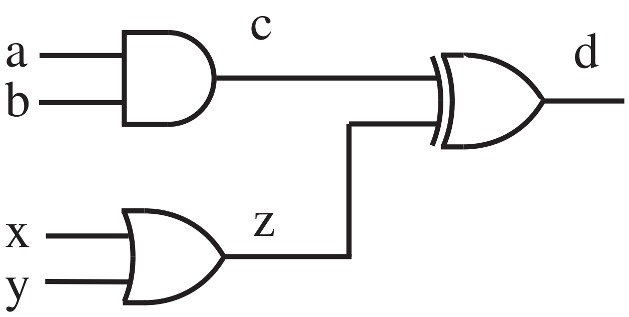
- A **PACKAGE** is a design unit to make **TYPE**, **COMPONENT**, **FUNCTION**, and other declarations visible to other design units.
- A **PACKAGE** consists of:
  1. a **PACKAGE Declaration** that is used to declare **TYPES**, **COMPONENTS**, **CONSTANTS**, **FUNCTIONS (header)**, **PROCEDURES (header)**, etc.
  2. an optional **PACKAGE BODY**, where **PROCEDURES (body)** and **FUNCTIONS (body)** are defined.
- A package is made visible to an entity with a **USE** clause.

Mixed Description

Example: 15 (Class) *Model the shown circuit using a mixed description, defining the AND gate as a component in a package. Identify the different description methods.*

```
PACKAGE andPackage IS
   COMPONENT and2 IS
      PORT( a, b:  IN bit;
            c:  OUT bit);
   END COMPONENT and2;
END PACKAGE andPackage;

USE WORK.andPackage.ALL;
ENTITY circuit IS
   PORT( a, b, x, y:  IN bit;
         d:  OUT bit);
END ENTITY circuit;
            ⋮
```



Mixed Description (Cont.)

Example 15: (Cont.)

```
              ⋮
ARCHITECTURE mixed OF circuit IS
   FOR gate:  and2
      USE ENTITY WORK.and2 (and2);
   SIGNAL c, z:  bit;
BEGIN
   gate:  and2
      PORT MAP (a, b, c);     -- positional association

   d <= c XOR z;

   op:  PROCESS (x, y) IS
   BEGIN
      z <= x OR y;
   END PROCESS op;
END ARCHITECTURE mixed;
```

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		

VHDL Libraries

- The **analyzer** processes our design units (entities and packages).
- Analyzer places successfully analyzed components in a **design library**.
- VHDL Language specification does not specify how a library is stored since this is specific to host operating system.
- A library may be defined as a database or as a directory.
- The library becomes visible to our design through the **USE** clause.

VHDL Libraries (Cont.)

There are 3 important libraries:

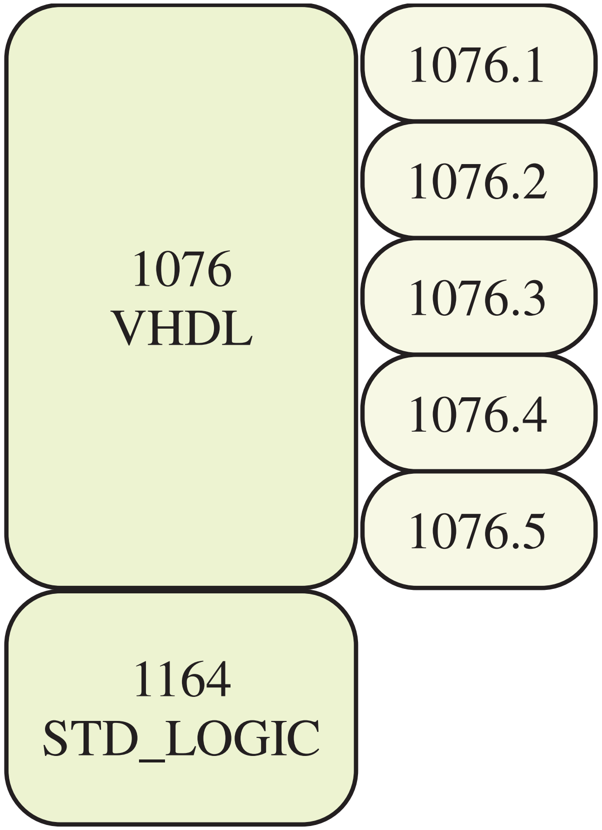
1. **WORK**: the working library where the analyzed design is stored in. Every model we write uses the contents of this library.
2. **STD**: contains a number of packages, including:
   - **STANDARD**: contains predefined types and operators. Every model we write uses the contents of this package.
   - **TEXTIO**: contains declarations of types and objects relating to reading and writing texts.
3. **IEEE**: has packages to define common data types and operators.

VHDL includes an implicit context clause of the form:

```
LIBRARY WORK;
LIBRARY std;
USE std.standard.ALL;
```

IEEE VHDL Standards

- **IEEE 1076 standards:**
  1. **1076.1** for analog and mixed analog-digital systems.
  2. **1076.2** for mathematical operations.
  3. **1076.3** for synthesis.
  4. **1076.4** VITAL (VLSI Initiative Towards ASIC Libraries) for standardizing ASIC models.
  5. **1076.5** for extensions and unification of IEEE library components.



1076
VHDL

1076.1

1076.2

1076.3

1076.4

1076.5

1164
STD_LOGIC

- **IEEE 1164:** defines the package std_logic_1164 that extends the VHDL language with multivalue logic, which is needed for describing real-time systems.

The Package Textio

- **FILE** is a VHDL data type.
- Files are not synthesizable. They are used for simulation only.
- Reading and writing to the file are done from/to variables of type **line**. Lines are read or written one by one from files.
- A file is opened with an access mode. Access modes are: **READ_MODE**, **WRITE_MODE**, or **APPEND_MODE**.
- File-access procedures return a status flag. Predefined flags are: **OPEN_OK**, **STATUS_ERROR**, **NAME_ERROR**, or **MODE_ERROR**.
- The package textio in the library std provides operations for reading and writing text files. Include:

```
USE std.textio.ALL;
```

The Package Textio (Cont.)

Example: 16 (Class) *Write a process that reads a text line from a file and parses it into internal variables.*

```
pr:  PROCESS IS
   FILE f:  text OPEN READ_MODE IS "test.txt";
   VARIABLE l:  line;
   VARIABLE t:  time;
   VARIABLE i:  integer;
   VARIABLE c:  character;
   VARIABLE s:  string (1 TO 6);
BEGIN
   WHILE NOT endfile (f) LOOP
      READLINE (f, l);  -- l = 100 ns  99  ABCDEZ  27
      READ (l, t);      -- t = 100 ns
      READ (l, i);      -- i = 99
      READ (l, c);      -- c = ’ ’
      READ (l, s);      -- s = " ABCDE"
      READ (l, i);      -- ERROR!
   END LOOP;
   WAIT;
END PROCESS pr;
```

The Package Textio (Cont.)

Example: 17 (Class) *Write a process that forms a text line from internal variables and writes it to a file.*

```
pr:  PROCESS IS
   FILE f:  text OPEN WRITE_MODE IS "log.txt";
   VARIABLE l:  line;
   VARIABLE t:  time := 100 ns;
   VARIABLE i:  integer := 99;
   VARIABLE c:  character := ’ ’;
   VARIABLE s:  string (1 TO 6) := " ABCDE";
BEGIN
   WRITE (l, t);
   WRITE (l, i);
   WRITE (l, c);
   WRITE (l, s);
   WRITELINE (f, l); -- l = 100 ns 99  ABCDE
   WAIT;
END PROCESS pr;
```

Testbenches

- Testbenches simulate **devices under test (DUT)**.
- A testbench consists of:
  1. Entity declaration without any ports.
  2. Instance of the DUT.
  3. Process to generate test waveforms.

Testbenches (Cont.)

Example: 18 (Class) *Write a testbench for a free-running counter.*

```
PACKAGE counterPackage IS
   COMPONENT counter IS
      PORT( clk:  IN bit;
            count:  OUT natural RANGE 0 TO 15);
   END COMPONENT counter;
END PACKAGE counterPackage;

USE WORK.counterPackage.ALL;
ENTITY counter_test IS
END ENTITY counter_test;

ARCHITECTURE testbench OF counter_test IS
   FOR dut:  counter USE ENTITY WORK.counter (behav);
   SIGNAL clock:  bit;
   SIGNAL count_value:  integer;
            ⋮
```

Testbenches (Cont.)

Example 18: (Cont.)

```
            ⋮
BEGIN
   dut:  counter PORT MAP (clk => clock, count => count_value);
   clk_gen:  PROCESS IS
   BEGIN
      WAIT FOR 5 ns; -- clock period is 10 ns

      IF clock = ’0’ THEN
         clock <= ’1’;
      ELSE
         clock <= ’0’;
      END IF;
   END PROCESS clk_gen;
END ARCHITECTURE testbench;
```

The ASSERT Statement

- We test a model by applying inputs and writing statements that check the output and report any errors. This way, we do not have to observe the actual waveforms and come to that conclusion ourselves.
- The **ASSERT** statement serves as an exception handling and is most often used to verify that a design functions correctly.
- **ASSERT** statements are used for simulation purposes only. They are ignored during design synthesis.
- An **ASSERT** statement could be used as a sequential statement inside a process or a concurrent one.

Example: 19 (Individual) *Assert that initial value is not more than the maximum value.*

```
ASSERT initial_value <= max_value
   REPORT "initial value exceeds max_value"
   SEVERITY note;
```

The ASSERT Statement (Cont.)

```
ASSERT condition REPORT message SEVERITY level;
```

- A condition is defined, which should be fulfilled in normal operation mode. If the **ASSERT** statement is violated, the **REPORT** clause will inform us of this situation, along with the **SEVERITY** level.
- If the condition is false, the simulator writes out:
  1. That an assertion violation has occurred with the severity level
  2. Simulation time
  3. Design unit name
  4. Message
- **Example:**

```
Assertion NOTE at 30 NS in design unit EQUAL_TEST(A) from process
/EQUAL_TEST/_P1:  "Signal n has an error!"
```

The ASSERT Statement (Cont.)

- **SEVERITY** level can be **note**, **warning**, **error**, or **failure**.

Severity level	Effect on simulation	Simulator action (depend on the simulator)
note	Continue	Ignore
warning	Continue	Write to file
error	Continue	Write to window
failure	Break	Write to window

- If the **ASSERT** part is ignored, the message is always written out:

```
REPORT "Finished OK" SEVERITY note;
```

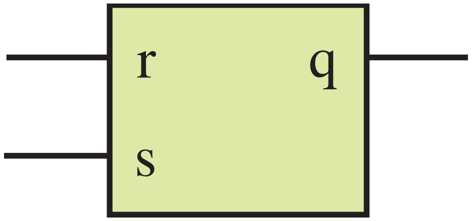
- Another method to write out a message:

```
ASSERT FALSE REPORT "Finished OK" SEVERITY note;
```

- If the **REPORT** part is ignored, only the simulator action is taken.
- If the **SEVERITY** part is ignored, a default level of error is assumed.
- To stop the simulator: `ASSERT FALSE SEVERITY failure;`

The ASSERT Statement (Cont.)

Example: 20 (Class) *Model the shown RS-flip-flop using concurrent signal assignments. Use a concurrent assert statement to generate a warning when R and S are concurrently set to* `'1'`.



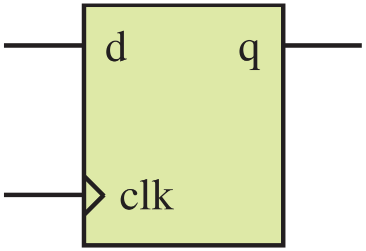
```
ENTITY rs_ff IS
   PORT( r, s:  IN bit;
         q:  OUT bit);
END ENTITY rs_ff;

ARCHITECTURE dataflow OF rs_ff IS
BEGIN
   q <=  '1' WHEN s = '1' ELSE
         '0' WHEN r = '1'; -- a conditional signal assignment statement
   ASSERT NOT (s = '1' and r = '1')
      REPORT "Setting S = '1' and R = '1' is illegal!"
      SEVERITY error;
END ARCHITECTURE dataflow;
```

Example: 21 (Home) *When is a concurrent ASSERT statement active?*

Testbenches (Cont.)

Example: 22 (Class) *Propose a test strategy for the shown D-flip-flop.*



Tested feature	Inputs		Delay after	Expected output
	D	CLK		Q
	1	0	15 ns	–
Sensitivity to rising edges Data acquisition (1)	1	1	15 ns	1
Input change, no output change	0	1	15 ns	1
Falling edge, no output change	0	0	15 ns	1
Sensitivity to rising edges Data acquisition (0) Stuck at faults on the output	0	1	15 ns	0

Testbenches using ASSERT Statements

Example: 23 (Class) *Write a testbench for a D-flip-flop based on the test strategy in Example 22 using sequential assert statements.*

```
ENTITY testbench IS
END ENTITY testbench;
ARCHITECTURE basic OF testbench IS
   COMPONENT d_ff IS
      PORT( d, clk:  IN bit;
            q:  OUT bit);
   END COMPONENT d_ff;
   FOR dut:  d_ff USE ENTITY WORK.d_ff (d_ff);

   SIGNAL d_in, clock, q_out:  bit;
BEGIN
   dut:  d_ff PORT MAP (d_in, clock, q_out);
   pd:  PROCESS IS
   BEGIN
      d_in <= ’1’;
      clock <= ’0’;        WAIT FOR 15 ns;
            ⋮
```

Testbenches using ASSERT Statements (Cont.)

Example 23: (Cont.)

```
            ⋮
      clock <= '1';         WAIT FOR 15 ns;
      ASSERT q_out = '1'
         REPORT "Problem with capturing a 1 on a rising edge."
         SEVERITY error;
      d_in <= '0';          WAIT FOR 15 ns;
      ASSERT q_out = '1'
         REPORT "Problem with No Change on a high level."
         SEVERITY error;
      clock <= '0';         WAIT FOR 15 ns;
      ASSERT q_out = '1'
         REPORT "Problem with No Change on a falling edge."
         SEVERITY error;
      clock <= '1';         WAIT FOR 15 ns;
      ASSERT q_out = '0'
         REPORT "Problem with capturing a 0 on a rising edge."
         SEVERITY error;
      WAIT;                 -- stop simulation run
   END PROCESS pd;
END ARCHITECTURE basic;
```

Testbenches using Test Files

- Assert statements are limited to strings only and need conversion functions for displaying signals.
- Use TEXTIO instead of assert statements to print messages.
- To write a text string to a file, a qualified expression must be provided using **`string'`** to explicitly state the type of a string literal to avoid ambiguity.
- To include current simulation time on the output result file, use system variable **NOW** that returns a time type value.

Testbenches using Test Files (Cont.)

Example: 24 (Class) *Rewrite Example 23 reading the set of stimuli from a file and writing error messages to another file.*

Contents of input file **`d_ff_test_vectors.txt`**, with a test vector per line.

```
1 1 15 ns 1 Problem with capturing a 1 on a rising edge.
0 1 15 ns 1 Problem with No Change on a high level.
0 0 15 ns 1 Problem with No Change on a falling edge.
0 1 15 ns 0 Problem with capturing a 0 on a rising edge.
```

Sample of output file **`d_ff_test_results.txt`**, with a test result per line.

```
Time is now 30 ns. D = 1, Clock = 1, Expected Q = 1, Actual Q = 1. Test passed.
Time is now 45 ns. D = 0, Clock = 1, Expected Q = 1, Actual Q = 1. Test passed.
Time is now 60 ns. D = 0, Clock = 0, Expected Q = 1, Actual Q = 1. Test passed.
Time is now 75 ns. D = 0, Clock = 1, Expected Q = 0, Actual Q = 1. Test failed! Error message: Problem with capturing a 0 on a rising edge.
```

Testbenches using Test Files (Cont.)

Example 24: (Cont.)

```
USE std.textio.ALL;
ENTITY testbench IS
END ENTITY testbench;
ARCHITECTURE test_file OF testbench IS
   COMPONENT d_ff IS
      PORT( d, clk:  IN bit;
            q:  OUT bit);
   END COMPONENT d_ff;
   FOR dut:  d_ff USE ENTITY WORK.d_ff (d_ff);
   SIGNAL d_in, clock, q_out:  bit;
BEGIN
   dut:  d_ff PORT MAP(d_in, clock, q_out);
   pd:  PROCESS IS
      FILE vectors_f:  text OPEN READ_MODE IS "d_ff_test_vectors.txt";
      FILE results_f:  text OPEN WRITE_MODE IS "d_ff_test_results.txt";
      VARIABLE stimuli_l, res_l:  line;
      VARIABLE d_in_file, clock_in_file, q_out_file:  bit;
      VARIABLE pause:  time;
      VARIABLE message:  string (1 TO 44);
         ⋮
```

Testbenches using Test Files (Cont.)

Example 24: (Cont.)

```
      ⋮
BEGIN
   d_in <= '1';      clock <= '0';  WAIT FOR 15 ns;
   WHILE NOT endfile (vectors_file) LOOP
      READLINE (vectors_f, stimuli_l);
      READ (stimuli_l, d_in_file);
      READ (stimuli_l, clock_in_file);
      READ (stimuli_l, pause);
      READ (stimuli_l, q_out_file);
      READ (stimuli_l, message);
      d_in <= d_in_file;   clock <= clock_in_file;  WAIT FOR pause;

      WRITE (res_l, string'("Time is now "));
      WRITE (res_l, NOW);          -- Current simulation time
      WRITE (res_l, string'(" D = "));
      WRITE (res_l, d_in_file);
      WRITE (res_l, string'(", Clock = "));
      WRITE (res_l, clock_in_file);
      ⋮
```

Testbenches using Test Files (Cont.)

Example 24: (Cont.)

```
         ⋮
         WRITE (res_l, string'(", Expected Q = "));
         WRITE (res_l, q_out_file);
         WRITE (res_l, string'(", Actual Q = "));
         WRITE (res_l, q_out);

         IF q_out /= q_out_file THEN
            WRITE (res_l, string'(".  Test failed!  Error message:"));
            WRITE (res_l, message);
         ELSE
            WRITE (res_l, string'(".  Test passed."));
         END IF;

         WRITELINE (results_f, res_l);
      END LOOP;
      WAIT;                          -- stop simulation run
   END PROCESS pd;
END ARCHITECTURE test_file;
```