8.1 Introduction to HDLs (VHDL)

A ***hardware description language*** is a textual language for describing digital systems, as Boolean functions, gate circuits, or other forms. Hardware description language is abbreviated as ***HDL***.

An HDL's key purpose is to support simulation. Given designer-provided test input values for particular times, ***simulation*** determines what output values a digital system would generate over time.

PARTICIPATION ACTIVITY 8.1.1: HDL simulator.

Animation captions:

1. An HDL can describe a digital system's inputs, outputs, and behavior.
2. An HDL can also describe test input values to the system.
3. The simulator executes HDL, generating the values that would be output at each indicated time.
4. At 0 ns, the simulator computes that y is 1.
5. At 10 ns, the simulator computes that y is 0.
6. The simulator proceeds, until all provided input values have been simulated.

PARTICIPATION ACTIVITY 8.1.2: HDL simulations.

Consider the above HDL simulator animation.

1) At time 25 ns, y is 0.
   - ○ True
   - ○ False

2) Input values are determined automatically by the HDL simulator.
   - ○ True
   - ○ False

3) Outputs values are determined automatically by the HDL simulator.
   - ○ True
   - ○ False

The two most common HDLs are Verilog and VHDL. ***VHDL*** is an HDL that was first published in 1987 as an IEEE standard, developed at the behest of the U.S. Dept. of Defense. The DoD wanted unambiguous simulatable models of chips being provided by suppliers. VHDL stands for VHSIC hardware description language, where VHSIC stands for Very High Speed Integrated Circuit. VHDL's syntax is borrowed largely from ***Ada***, an earlier DoD language for software programming.

Verilog is an HDL that originated in 1985 at a company called Gateway Design Automation (now Cadence). Verilog was also intended to simulate (or verify) logic circuits. Verilog also became an IEEE standard in 1995. Verilog syntax comes largely from the C programming language.

In the 1990s, HDLs began being used not just for simulating existing circuits, but to specify desired circuit behavior. A ***synthesis tool*** automatically converts specified behavior into a circuit.

PARTICIPATION ACTIVITY 8.1.3: HDLs.

1985 **1987** **VHDL** **Verilog** **C**

	Year that Verilog originated.
	Year that VHDL was first published, as an IEEE standard.
	An HDL developed by the U.S. Dept. of Defense.
	An HDL developed by Gateway Design Automation, originally proprietary, but later published as an IEEE standard.
	The popular programming language, known for conciseness, from which Verilog borrows much syntax.

Reset

Exploring further:

- Verilog (Wikipedia)
- VHDL (Wikipedia)

8.2 Combinational logic (VHDL)

In VHDL, an ***entity*** defines a new component. An entity includes an entity declaration and an architecture body. The ***entity declaration*** specifies the entity's external interface. The ***architecture body*** specifies the entity's behavior or structure.

The entity declaration specifies the entity's name and a list of the entity's ports. A ***port*** is a named input or output of an entity. Each entity port definition specifies the port's name, direction, and type. The ***std_logic*** type defines a bit that can have the values '0' or '1'. To use the std_logic type, the statements `library IEEE; use IEEE.STD_LOGIC_1164.ALL;` must come before the entity declaration.

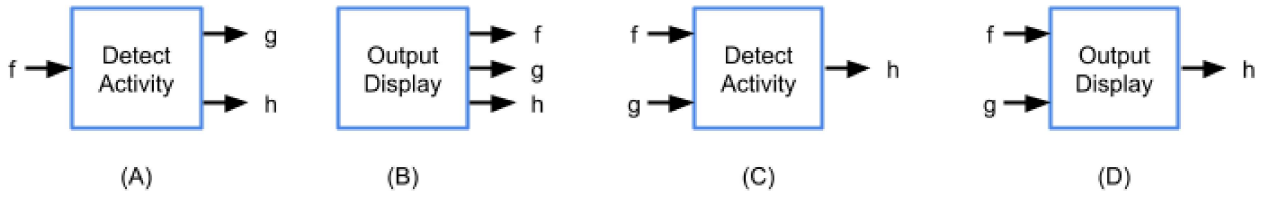
PARTICIPATION ACTIVITY 8.2.1: DoorOpenDetector entity definition.

Animation captions:

1. Entity declaration declares a component named DoorOpenDetector.
2. Entity declaration specifies the entity's name and interface.
3. Port definition specifies the port's name, direction, and type.
4. std_logic requires using the IEEE library and standard logic package.
5. The architecture body specifies the entity's behavior or structure.

PARTICIPATION ACTIVITY 8.2.2: Entity definitions.

Match the entity definition to the appropriate circuit.



(C) (A) (D) (B)

```
entity DetectActivity is
   port (f : in std_logic;
         g, h : out std_logic);
   ...
end DetectActivity;
```

```
entity DetectActivity is
   port (f, g: in std_logic;
         h : out std_logic);
   ...
end DetectActivity;
```

```
entity OutputDisplay is
   port (f, g, h : out std_logic);
   ...
end OutputDisplay;
```



```
entity OutputDisplay is
   port (f, g : in std_logic;
         h : out std_logic);
   ...
end OutputDisplay;
```

Reset

A designer can describe a combinational circuit's function using a process. A ***process*** defines a statement sequence that executes throughout simulation. A process' ***sensitivity list*** defines the inputs and signals whose value changes cause the process to execute. A combinational circuit is sensitive to all of the circuit's inputs.

`begin` and `end process;` surround a process' statements.

PARTICIPATION ACTIVITY 8.2.3: Process describing a combinational circuit.

Animation captions:

1. The process keyword defines a sequence of statements executed during simulation.
2. The sensitivity list defines inputs and signals whose value changes cause the process to execute. Combinational circuits are sensitive to all circuit inputs.
3. Assignment statements assign values to outputs. Bitwise operators in VHDL include and, or, and not.

PARTICIPATION ACTIVITY 8.2.4: Process for combinational circuits.

Complete the process for the provided combinational equations.

1) z = ab

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity GetSensorReading is
   port (a, b : in std_logic;
         z : out std_logic);
end GetSensorReading;

architecture Behavior of
GetSensorReading is
begin
   [          ](a, b) begin
      z <= a and b;
   end process;
end Behavior;
```


Check Show answer

2) m = j'
n = jk

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity UpdateDisplay is
   port (j, k : in std_logic;
         m, n : out
std_logic);
end UpdateDisplay;

architecture Behavior of
UpdateDisplay is
begin
   process([          ]) begin
      m <= not j;
      n <= j and k;
   end process;
end Behavior;
```

Check Show answer

3) t = v's'm'

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SetTimer is
   port (v, s, m : in
std_logic;
         t : out std_logic);
end SetTimer;

architecture Behavior of
SetTimer is
begin
   process(s,          )
begin
      t <= not s and not v
and not m;
   end process;
end Behavior;
```


Check Show answer

4) m = f
q = f'
p = f'

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity GetSensorReading is
   port (f : in std_logic;
         m, p, q : out
std_logic);
end GetSensorReading;

architecture Behavior of
GetSensorReading is
begin
                    begin
      m <= f;
      q <= not f;
      p <= not f;
   end process;
end Behavior;
```

Check Show answer

An ***assignment statement*** like y <= a and b assigns the left-side signal with the result of the right-side expression. A ***signal*** maintains a value over time, with the value being updated by assignment statements. All inputs and outputs of an entity are signals.

An ***expression*** is a combination of items, like inputs, signals, literals, and operators, that evaluates to a value. Ex: `a and b`, where `and` is the bitwise AND operator. If a is 0 and b is 1, then the expression evaluates to 0 and 1, which is 0.

An operator is a symbol for a built-in calculation. ***Bitwise operators*** perform the specified Boolean operation on each bit of the operands.

Table 8.2.1: Bitwise operators.

Operator	Description
and	Bitwise AND operation
or	Bitwise OR operation
not	Bitwise NOT operation
nand	Bitwise NAND operation
nor	Bitwise NOR operation
xor	Bitwise XOR operation
xnor	Bitwise XNOR operation

PARTICIPATION ACTIVITY 8.2.5: Boolean expression in VHDL.

Directly convert each given Boolean expression to VHDL.
Note: This activity requires answers that directly match the given expressions. Ex: For ab, type a and b. Variations like b and a are not accounted for.

1) z = a + b

```
z <= [          ];
```

Check Show answer

2) z = a'b

```
z <= [          ] and b;
```

Check Show answer

3) z = (ab) + c

```
z <= [          ];
```

Check Show answer

4) z = (ab') + (ac)

```
z <=
                    ;
```

Check Show answer

Forgetting to include all inputs of combinational logic in the process' sensitivity list is a common error. While the resulting process is legal VHDL, the process does not describe combinational logic.

PARTICIPATION ACTIVITY 8.2.6: Detect the error in defining an entity for combinational logic.

Click the erroneous code.

1)
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CloseDoor is
    port ( a, b, c : in std_logic;
           z : out std_logic);
end CloseDoor;

architecture Behavior of CloseDoor is
begin
    process (a, b) begin
        z <= not a or not b or not c;
    end process;
end Behavior;
```

2)
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SoundAlarm is
    port (a, b : in std_logic;
          y, z : out std_logic);
end SoundAlarm;

architecture Behavior of SoundAlarm is
begin
    process(a, b)
        y <= a and b;
        z <= a;
    end process;
end Behavior;
```


3)
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity LightController is
```

```
    port (a, b : in std_logic;
          z : std_logic);
end LightController;

architecture Behavior of LightController is
begin
    process(a, b) begin
        z <= a or b;
    end process;
end Behavior;
```


4)
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity UnlockDoor is
    port ( z : out std_logic;
           c, d : in std_logic);
end UnlockDoor;

architecture Behavior of UnlockDoor is
begin
    process(c, d) begin
        z = c xor d;
    end process;
end Behavior;
```

PARTICIPATION ACTIVITY 8.2.7: HDL simulator: Combinational process.

Animation content:

undefined

Animation captions:

1. Process executes the statements between the begin and end process.
2. Changes to inputs in the sensitivity list cause the process to execute. After assigning a value to output z, the process waits until another change occurs.
3. Input b changes (from 0 to 1). The process executes.
4. Inputs b and c remain the same. The process does not execute.
5. Inputs b and c change. The process executes and assigns a new value to output z.

PARTICIPATION ACTIVITY 8.2.8: Combinational logic simulator: Designer provides input values; running the simulator generates output values.

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
entity CombinationalLogicExample is
     port(a, b, c : in std_logic;
          z : out std_logic);
end CombinationalLogicExample;

architecture Behavior of CombinationalLogicExample is
begin
     process(a, b, c) begin
          z <= a and b and c;
     end process;
end Behavior;
```


Inputs: *click waveform to edit* **Load sample inputs**

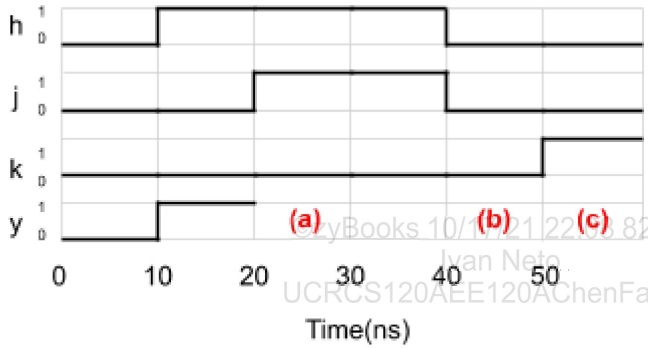
Run

Output:

PARTICIPATION ACTIVITY 8.2.9: Sensitivity list.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity AirControl is
    port(h, j, k : in std_logic;
         y : out std_logic);
end AirControl;

architecture Beh of AirControl is
begin
   process(h, j, k) begin
     y <= h or (not j and k);
   end process;
end Beh;
```

Timing diagram



1) Value of y at (a).

- O 0
- O 1

2) The process executes at 30 ns.

- O True
- O False

3) Value of y at (b).

- O 0
- O 1

4) The process executes at 50 ns.

- O True
- O False

5) Value of y at (c).

- O 0
- O 1

8.3 Identifiers (VHDL)

An ***identifer*** is a designer-defined name used for items such as entities, inputs, and outputs. An identifier must start with a letter (A-Z, a-z), followed by any number of letters (A-Z, a-z), digits (0-9), or underscores (_). Each underscore must be followed by a letter or digit. Identifiers are case insensitive, meaning upper and lower case letters are the same. So testEn and testEN are the same.

A ***keyword*** is a word that is part of the language, like the words entity and in. A designer cannot use a keyword as an identifier. A table of keywords appears at this section's end.

PARTICIPATION ACTIVITY 8.3.1: Identifier validator.

Enter an identifier: _____ **Validate**

PARTICIPATION ACTIVITY 8.3.2: Valid identifiers.

Which are valid identifiers?

1) SensorA
   - Valid
   - Invalid

2) red_led
   - Valid
   - Invalid

3) first!motor
   - Valid
   - Invalid

4) fsm input a
   - Valid
   - Invalid

©zyBooks 10/17/21 22:08 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

5) _REGISTER_LOAD_
   - Valid
   - Invalid

6) 1_MuxSel

- Valid
- Invalid

7) MuxSel_19

- Valid
- Invalid

8) TestVal__0

- Valid
- Invalid

9) process

- Valid
- Invalid

Table 8.3.1: VHDL keywords.

abs	constant	if	on	report	type
access	context	impure	open	restrict	unaffected
after	cover	in	or	restrict_guarantee	units
alias	default	inertial	others	return	until
all	disconnect	inout	out	rol	use
and	downto	is	package	ror	variable
architecture	else	label	parameter	select	vmode
array	elsif	library	port	sequence	vprop
assert	end	linkage	postponed	severity	vunit
assume	entity	literal	procedure	shared	wait
assume_guarantee	exit	loop	process	signal	when
attribute	fairness	map	property	sla	while

begin	file	mod	protected	sll	with
block	for	nand	pure	sra	xnor
body	force	new	range	srl	xor
buffer	function	next	record	strong	
bus	generate	nor	register	subtype	
case	generic	not	reject	then	
component	group	null	release	to	
configuration	guarded	of	rem	transport	

8.4 Testbench (VHDL)

A ***testbench*** provides a sequence of input values to test an entity. A testbench is itself an entity with no inputs or outputs, and consists of two main elements:

1. A component declaration and instantiation that creates an instance of the entity being tested.
2. A process for generating the sequence of input values to test the entity.

The testbench creates an instance of the entity being tested, known as a component. A ***component declaration*** specifies a component's name and ports, which should be the same as the entity declaration. A ***component instantiation*** creates an instance of a component, gives that instance a name, and specifies how the component's ports are connected. Signals connect to the component's inputs and outputs.

A ***port map*** connects signals to the inputs and outputs of the component instance. A ***positional port map*** specifies connections following the order of the entity's port list.

This material appends _tb to the names for component instances and signals within the testbench.

PARTICIPATION ACTIVITY 8.4.1: A testbench instantiates the component to test, and prepares to set the component's input values and check output values.

Animation captions:

1. The testbench tests an entity's functionality by providing a sequence of input values. The testbench itself has no inputs or outputs.

2. Component declaration specifies the entity to be tested. Component declaration has same name and ports as entity declaration.
3. Signals connect to the component's inputs and outputs.
4. Component instantiation creates an instance of the entity to test.
5. Ports are connected in the same order as in the component's declaration.
6. Component instantiation consists of instance name, entity name, and port connections.

A single VHDL statement may be split across multiple lines. Component instantiations involving many ports or long signal names are often too long to fit on a single line, so designers commonly use multiple lines to improve readability.

PARTICIPATION ACTIVITY 8.4.2: Testbench declaration.

1) The testbench must contain the same inputs and outputs as the component being tested.
   - True
   - False

2) The component instantiation may appear on a single line, as shown below.

```
UnlockDoor_tb : UnlockDoor port map
(b_tb, c_tb, d_tb, z_tb);
```

   - True
   - False

PARTICIPATION ACTIVITY 8.4.3: Component instantiation.

Complete the testbench for each entity. Use the naming convention of appending _tb for all signal and component instance names.

1)
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ControlValve is
   port(a : in std_logic;
        p, r : out std_logic);
end ControlValve;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of
Testbench is
    
is
      port(a : in std_logic;
           p, r : out
std_logic);
   end component;

   signal a_tb, p_tb, r_tb :
std_logic;

begin
   ControlValve_tb :
ControlValve
                      port map
(a_tb, p_tb, r_tb);

   -- Designer-provided input
values
end Behavior;
```

Check Show answer

2)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CheckSensor is
   port(a, b : in std_logic;
        m : out std_logic);
end CheckSensor;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of
Testbench is
   component CheckSensor is
      port(a, b : in
std_logic;
           m : out
std_logic);
   end component;

   [                    ],
m_tb : std_logic;

begin
   CheckSensor_tb:
CheckSensor
                    port map
(a_tb, b_tb, m_tb);

   -- Designer-provided input
values
end Behavior;
```


Check Show answer

3)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SafetyCheck is
   port(a : in std_logic;
        x, y : out std_logic);
end SafetyCheck;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of
Testbench is
   component SafetyCheck is
      port(a : in std_logic;
           x, y : out
std_logic);
   end component;

   signal a_tb, x_tb, y_tb :
std_logic;

begin

                port map (a_tb,
x_tb, y_tb);

   -- Designer-provided input
values
end Behavior;
```

Check Show answer

4)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity OpenVault is
   port(f : in std_logic;
        s, t : out std_logic);
end OpenVault;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of
Testbench is
   component OpenVault is
      port(f : in std_logic;
           s, t : out
std_logic);
   end component;

   signal f_tb, s_tb, t_tb :
std_logic;

begin
   OpenVault_tb : OpenVault
                  port map (
                           );
   -- Designer-provided input
values
end Behavior;
```

Check Show answer

The sequence of input values for testing a component appears within a process. Each unique sequence of values used to test a component is known as a ***test vector***.

PARTICIPATION ACTIVITY 8.4.4: A testbench generates input values using a process.

Animation captions:

1. Testbench assigns values to signals connected to component instance's inputs.
2. Process provides input values.
3. Testbench should assign values to all signals at the beginning of simulation. Each unique assignment of values is known as a test vector. Ex: b_tb <= 0, c_tb <= 0, d_tb <= 0 is one test vector.
4. The wait statement delays simulation for 10 nanoseconds.
5. Simulator will advance time before executing the next statement.
6. Wait and assignment statements generate the desired sequence of input values to test the component. A wait statement without a timeout clause will delay process' simulation forever.

A ***wait statement*** delays simulation of a process. A ***timeout clause*** used in a wait statement delays simulation for a specified time. Ex: `wait for 10 ns;` delays simulation for 10 nanoseconds of simulated time. A wait statement without a timeout clause, e.g., `wait;`, will delay the process' simulation forever.

The drawn value over time for an input or output is called a ***waveform***, , as in the above animation's timing diagram.

Table 8.4.1: Time unit specifications.

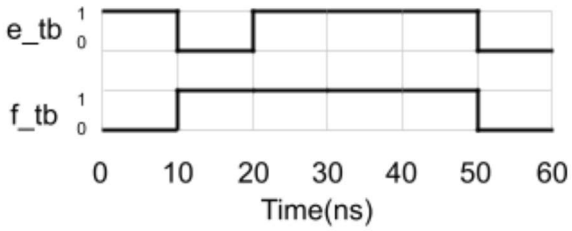
String	Time units
hr	hours
min	minutes
sec	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds

ps	picoseconds
fs	femtoseconds

PARTICIPATION ACTIVITY 8.4.5: Testbench: Generating waveforms.


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of Testbench is
   component SetTime is
      port(e, f : in std_logic;
           m : out std_logic);
   end component;

   signal e_tb, f_tb, m_tb : std_logic;

begin
   SetTime_tb : SetTime
                port map(e_tb, f_tb, m_tb);

   process begin
      (a)
      f_tb <= '0';
      wait for 10 ns;
      (b)
      f_tb <= '1';
      (c)
      e_tb <= '1';
      (d)
      e_tb <= '0';
      f_tb <= '0';
      wait;
   end process;
end Behavior;
```

Timing diagram



1) (a)

- e_tb <= 0;
- e_tb <= 1;

2) (b)

- e_tb <= 0;
- e_tb <= 1;
  f_tb <= 1;
- f_tb <= 0;

3) (c)

- wait for 10 ns;
- wait 10 ns;
- wait for 1 ns;

4) (d)

- wait for 10 ns;
- wait for 20 ns;
- wait for 30 ns;

A simulator will simulate both the testbench and component instance simultaneously, generating waveforms showing the input and output values. As the testbench assigns input values for the component being tested, those changes will cause the component's entity process to execute, which will then update the component's output values.

PARTICIPATION ACTIVITY 8.4.6: Testbench simulation process.

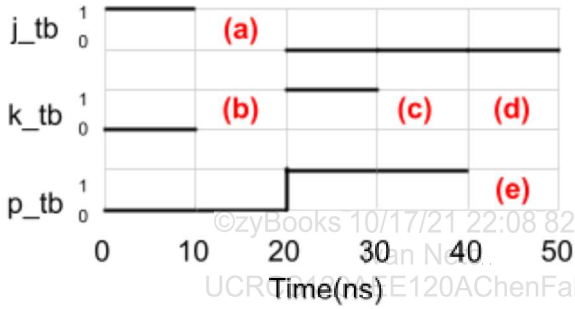
Animation captions:

1. Testbench assigns values to inputs.
2. Change in input values cause the process to execute and update the circuit's output.
3. Wait statement delays simulation. Input and output values are maintained.
4. Testbench assigns new values for inputs, process will execute.
5. Simulation continues.

PARTICIPATION ACTIVITY 8.4.7: Testbench simulation.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Testbench is
end Testbench;

architecture Behavior of Testbench is
   component AlarmOff is
      port(j, k : in std_logic;
           p : out std_logic);
   end component;

   signal j_tb, k_tb, p_tb : std_logic;

begin
   AlarmOff : AlarmOff
              port map (j_tb, k_tb, p_tb);

   process begin
      j_tb <= '1';
      k_tb <= '0';
      wait for 10 ns;
      k_tb <= '1';
      wait for 10 ns;
      j_tb <= '0';
      wait for 20 ns;
      k_tb <= '0';
      wait;
   end process;
end Behavior;
```

Timing diagram




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity AlarmOff is
   port(j, k : in std_logic;
        p : out std_logic);
end AlarmOff;

architecture Behavior of AlarmOff is
begin
   process (j, k) begin
      p <= not j and k;
   end process;
end Behavior;
```

1) (a)

- 0
- 1

2) (b)

- 0
- 1

3) (c)

- 0
- 1

4) (d)

- 0
- 1

5) (e)

-

0

1

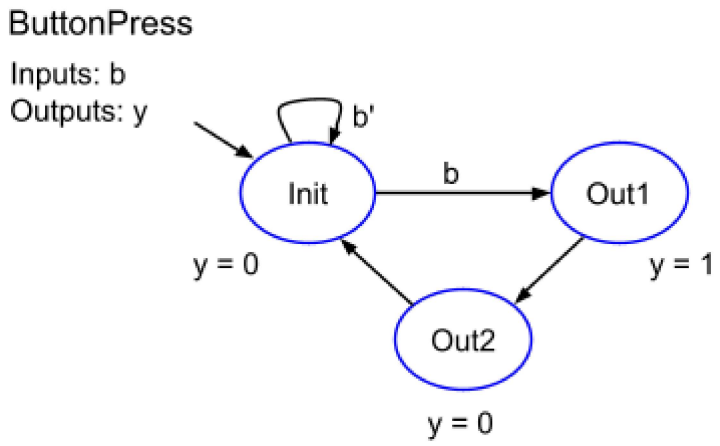
8.5 Sequential logic (VHDL)

A VHDL description of an FSM consists of:

1. Inputs and outputs including clock and reset inputs.
2. A type declaration creating a new state type and state names, and a variable for storing the state.
3. Statements that reset the FSM to an initial state.
4. Case statements for describing the state transitions and state actions.

The following is an example of an FSM described in VHDL, with each part discussed further below.

Figure 8.5.1: Example ButtonPress FSM and VHDL description.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ButtonPress_FSM is
   port (clk, rst : in std_logic;
         b : in std_logic;
         y : out std_logic);
end ButtonPress_FSM;

architecture Behavior of ButtonPress_FSM is
begin

   process(clk)
      type BP_statetype is (BP_Init, BP_Out1, BP_Out2);
      variable BP_State : BP_statetype;
   begin
      if (rising_edge(clk)) then
         if (rst = '1') then
            -- Initial state
            BP_State := BP_Init;
         else
            -- State transitions
            case BP_State is
               when BP_Init =>
                  if ((not b) = '1') then
                     BP_State := BP_Init;
                  elsif (b = '1') then
                     BP_State := BP_Out1;
                  end if;

               when BP_Out1 =>
                  BP_State := BP_Out2;

               when BP_Out2 =>
                  BP_State := BP_Init;

               when others =>
                  BP_State := BP_Init;
            end case;
         end if;

         -- State actions
         case BP_State is
            when BP_Init =>
               y <= '0';

            when BP_Out1 =>
               y <= '1';

            when BP_Out2 =>
               y <= '0';

            when others =>
               y <= '0';
         end case;
      end if;
   end process;

end Behavior;
```


©zyBooks 10/17/21 22:08 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

PARTICIPATION ACTIVITY 8.5.1: First two parts of an FSM description in VHDL.

Animation captions:

1. Sequential logic includes clk and rst inputs, in addition to inputs and outputs.
2. A type declaration creates a new type name to represent an FSM's states and specifies named values for each state.
3. A local variable stores the state.

The FSM needs to store a set of named values for each state. A ***type declaration*** creates a new type name and specifies the type's possible values. An ***enumeration type*** defines a set of named values. A type declaration for an enumeration type starts with the keyword ***type*** followed by the type name, the keyword ***is***, and a comma-separated list of the enumerated names within parentheses, and ending with a semicolon. Ex: `type BP_statetype is (BP_Init, BP_Out1, BP_Out2);` defines a new type named BP_statetype that has three possible values, which correspond to the FSM's state.

A variable of the new type stores the FSM's state: `variable BP_State : BP_statetype;`. A ***variable*** stores a value local to a process, with the value being immediately updated by variable assignment statements. A ***variable assignment statement*** like `BP_State := BP_Init;` immediately assigns the left-side variable with the result of the right-side expression. A variable declaration appears between the process' declaration and begin parts. The variable is only accessible within that process.

This material prepends a short prefix to each FSM-related item. Ex: For an FSM named MotorController, the VHDL state variable might be MC_State, and the type's enumerated names might be MC_Init, MC_Wait, etc.

PARTICIPATION ACTIVITY 8.5.2: FSM state type and variable declarations.

Click the erroneous code.

1)
```
type SP_statetype is (SP_Init, SP_Next,
SP_Out );
variable SP_State;
```

2)
```
variable LC_statetype is ( LC_Init,
LC_CheckSensor,
                           LC_WriteData,
LC_ReadData
                           LC_ReadData );
variable LC_State : LC_statetype;
```

```
type DH_statetype is ( DH_Init,
```

3) DH_AlarmOn; DH_AlarmOff);
variable DH_State : DH_statetype;

4) type ME_statetype is (ME_Init,
ME_Activate,
ME_Deactivate);
variable ME_statetype : ME_State;

The FSM's VHDL description requires clock and reset inputs, named clk and rst.

The process is sensitive to any change on clk, but an if statement (described below) ensures the FSM only executes on rising clock edges (not falling edges). The built-in rising_edge() function detects whether a 1-bit signal just changed from 0 to 1.

If the clock rose, another if statement checks if rst is 1. If so, the state variable is assigned with an initial state, else the normal FSM execution occurs.

A process' ***if statement*** executes or skips a statement group, depending on an expression's value. An if statement may optionally have one or more ***elsif*** parts (short for else if) to support more expressions and statement groups, evaluated sequentially. An if statement may optionally have one final ***else*** part that executes if none of the preceding expressions were true.

PARTICIPATION ACTIVITY 8.5.3: An if-else statement is used to assign the FSM's initial state.

Animation captions:

1. Initial state is unknown. The if statement only executes on a rising clock edge.
2. On the rising clock edge, if rst is 1, BP_State is assigned with the initial state.
3. The BP_State variable stores the current state until the next rising clock edge.

After initialization, the FSM's process consists of two steps:

1. State transitions: The process first evaluates the state transitions, and updates the state variable with the new state.
2. State actions: The process then executes the new state's actions.

Each step uses a case statement. A ***case statement*** selects one case choice's statements to execute based on the case expression. Execution jumps to the ***case choice*** whose constant expression matches the value of the case expression. A case choice starts with the keyword ***when*** followed by the case choice's constant expression and the two-character sequence =>. The case choice's statements are executed, and execution then jumps to `end case;`. A default case choice `when others =>` executes if no other case choice matches the case expression.

PARTICIPATION ACTIVITY 8.5.4: Case statement executes the case choice for the current state, thus updating the state variable.

Animation captions:

1. Case statement evaluates the case expression and jumps to the matching case choice.
2. The case choice's statements are executed.
3. Case statement jumps to the end case.

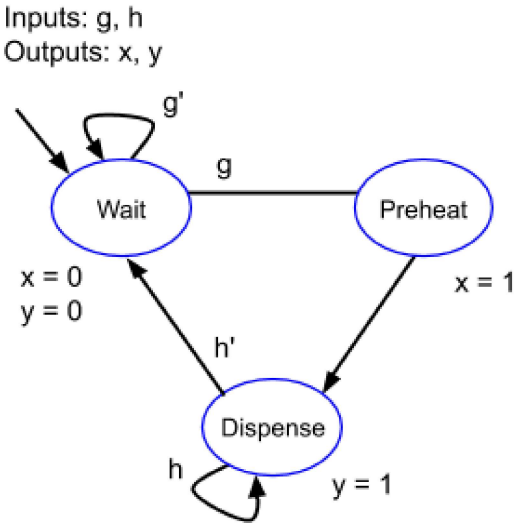
A good practice is to follow a state-by-state conversion of the FSM to state transitions and then state actions.

PARTICIPATION ACTIVITY 8.5.5: A process used in a testbench to generate the clock input for an FSM.

Animation captions:

1. Testbench uses a process to generate a clock signal.
2. Process generates clock throughout simulation.
3. The process for generating other input values can delay simulation for a clock cycle using CLK_PERIOD constant.

PARTICIPATION ACTIVITY 8.5.6: VHDL: FSM state and output assignments.




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity AD_FSM is
   port (clk, rst : in std_logic;
         g, h : in std_logic;
         x, y : out std_logic);
end AD_FSM;

architecture Behavior of AD_FSM is
begin
   process(clk)
      type AD_statetype is (AD_Wait, AD_Preheat,
                            AD_Dispense);
      variable AD_State : AD_statetype;
   begin
...
```


1) Complete the initial state assignment.

```
if (rising_edge(clk)) then
   if (rst = '1') then
      -- Initial state
      AD_State :=
               ;
   else
```

Check Show answer

2) Complete the case statement.

```
// State transitions
case              is
...
end case;
```

Check Show answer

3) Complete Wait's state transitions.

```
when AD_Wait =>
   if ((not g) = '1') then
      AD_State := AD_Wait;
   elsif (g = '1') then

   end if;
```

Check Show answer

4) Complete Preheat's state transitions.

```
when AD_Preheat =>

```

Check Show answer

5) Complete Wait's state actions.


```
when AD_Wait =>

```

Check Show answer

6) Complete Dispense's state actions.

```
when AD_Dispense =>
```

Check Show answer

An if statement's expression must evaluate to a Boolean value true or false. To implement a state transition's condition involving std_logic signals, the equality operator must be used. The ***equality operator*** = compares two expressions and evaluates to true if equal, and false otherwise. Ex: `g = '1'` evaluates to true if the signal g is equal to '1'.

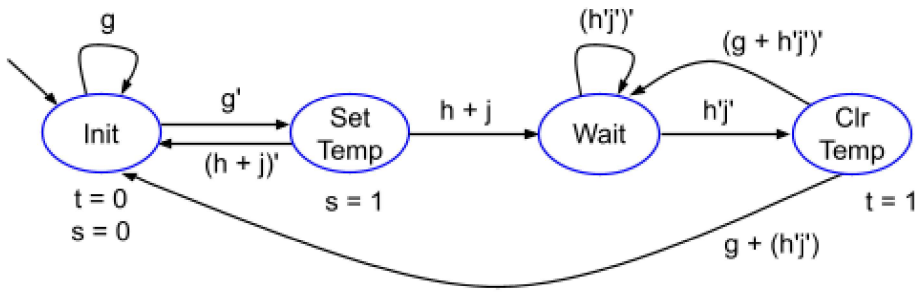
For each state transition, the corresponding if statement's expression compares the state transition's condition with the value '1'. A state transition condition dc' is translated to the Boolean expression `(d and not c) = '1'`.

PARTICIPATION ACTIVITY 8.5.7: VHDL: State transitions.

Convert each FSM state transition to VHDL (do not reorder the variables).

QuickSelector

Inputs: g, h, j
Outputs: s, t



1) g'

```
when QS_Init =>
   if ((
) = '1') then
      QS_State := QS_SetTemp;
   ...
```

Check Show answer

2) h'j'

```
when QS_Wait =>
   if ((
) = '1') begin
      QS_State := QS_ClrTemp;
   elsif ...
```

Check Show answer

3) g + (h'j')

```
when QS_ClrTemp =>
   if ((
                                )
= '1' ) then
      QS_State := QS_Init;
   elsif ...
```


Check Show answer

4) h + j

```
when QS_SetTemp =>
   ...
   elsif (
                            ) then
      QS_State := QS_Wait;
   end if;
```

Check Show answer

A testbench for sequential logic must generate a clock input with the desired clock period, reset the sequential logic at the beginning of simulation, and provide test sequences of the other inputs. The clock period can be defined as a constant. A ***constant*** defines a name value that cannot change. A constant definition starts with the keyword `constant` followed by the constant name, type, and value. Ex: `constant CLK_PERIOD : time := 20 ns;` defines a constant of type time named CLK_PERIOD with the value 20 nanoseconds. A process uses this constant to repeatedly generate the clock input. Another process uses this parameter to delay simulation for a clock cycle to generate test vectors.

PARTICIPATION ACTIVITY 8.5.8: A process used in a testbench to generate the clock input for an FSM.

Animation captions:

1. Testbench uses a process to generate a clock signal.
2. Process generates clock throughout simulation.
3. The process for generating other input values can delay simulation for a clock cycle using CLK_PERIOD constant.

PARTICIPATION ACTIVITY 8.5.9: Clock process.

1) The HDL simulator will automatically generate a clock for the FSM.

   - True
   - False

2) The following process repeatedly executes statements that set the clock signal to 0 and 1.

```
process begin
   clk_tb <= '0';
   wait for CLK_PERIOD/2;
   clk_tb <= '1';
   wait for CLK_PERIOD/2;
end process;
```

   - True
   - False

8.6 Datapath components: Structural (VHDL)

Structural description of datapath components

Datapath components can often be designed using multiple instances of smaller components. Such a design can be described in VHDL structurally, by first declaring components, and then instantiating the components with appropriate connections.

PARTICIPATION ACTIVITY 8.6.1: Structural VHDL description of a 4-bit carry ripple adder.

Animation content:

undefined

Animation captions:

1. A full adder can be described in VHDL behaviorally using the equations for each output s and co. The entity defines the inputs and outputs. The architecture describes the combinational behavior of a full adder.
2. A 4-bit carry ripple adder has four full adders. Each pair of bits for the operands is added by one full-adder, and the carry is passed on to the full-adder for the next higher bit.
3. A carry-ripple adder can be implemented as a structural VHDL description. CarryRippleAdder4 has two 4-bit inputs a and b, a carry in input ci, a 4-bit sum output s, and a final carry out co.

4. The architecture begins by declaring the component FullAdder and three internal signals, co1, co2 and co3, used for internal connection between the full adders.
5. The four FullAdder components are instantiated with the names FA0, FA1, FA2, and FA3. a, b, and s are vectors, so the individual bits are accessed using a(0), b(0), s(0), etc.

PARTICIPATION ACTIVITY 8.6.2: Structural VHDL of a carry ripple adder.

1) How many FullAdder components need to be instantiated for a 32-bit carry ripple adder?
   - 31
   - 32

2) Structural VHDL description is advantageous when smaller components of the same type used several times in the datapath.
   - True
   - False

Structural description an up-counter

A 4-bit up-counter can be implemented structurally by instantiating a 4-bit register, 4-bit incrementer, and a 4 input AND gate. Two signals are used for internal connection.

PARTICIPATION ACTIVITY 8.6.3: Structural VHDL description of a 4-bit up-counter.

Animation content:

undefined

Animation captions:

1. The UpCounter entity defines the counter's inputs and outputs, consisting of a clock input (clk), a reset (rst), a count enable control input (cnt), the 4-bit count output (C), and a terminal count output (tc).
2. The architecture declares the components used by the up-counter. Reg4 is a 4-bit parallel-load register with a load control input ld. Inc4 is a 4-bit incrementer. AND4 is a 4-input AND gate.
3. Signals tempC and incC are used as internal wires. The register output cannot be directly connected to the output port C, because the register output connects internally to the

incrementer and AND gate.

4. The output Q of Reg4_1 is connected to tempC, which connects the register's output to both the incrementer, Inc4_1, and the AND gate, AND4_1.
5. Each bit of tempC is connected to one input of the AND gate. tempC(3) is connected to AND_1's w input, tempC(2) is connected to AND_1's x input, and so on. The output of the AND gate is tc.
6. Because tempC is needed for internal connections, a process is used to assign the output Q with tempC.

PARTICIPATION ACTIVITY 8.6.4: Structural VHDL description of a 4-bit up-counter.

1) To design a 32-bit up-counter using a structural VHDL description, 32 incrementer components must be instantiated.
   - True
   - False

2) The statement `C <= tempC;` in the above example connects the register's output to the output port C.
   - True
   - False

3) The process that assigns C with tempC is executed every time there is a change in the clk.
   - True
   - False

4) Inc4 is the only sequential component within the up-counter.
   - True
   - False

Behavioral code for the datapath components in the 4-bit up-counter.

The register, incrementer, and AND gate used in the structural description of the up-counter are each implemented using a behavioral description.

Figure 8.6.1: Behavioral VHDL for a 4-bit register.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg4 is
   port(clk, rst: in std_logic;
        ld: in std_logic;
        I: in std_logic_vector(3 downto 0);
        Q: out std_logic_vector(3 downto 0));
end Reg4;

architecture Behavior of Reg4 is
begin
   process(clk) begin
      if (rising_edge(clk)) then
         if ld = '1' then
            Q <= I;
         end if;
      end if;
   end process;
end Behavior;
```


Figure 8.6.2: Behavioral VHDL for a 4-bit incrementer.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Inc4 is
   port (a: in std_logic_vector(3 downto 0);
         s: out std_logic_vector(3 downto 0));
end Inc4;

architecture Behavior of Inc4 is
begin
   process(a) begin
      s <= a + 1;
   end process;
end Behavior;
```


Figure 8.6.3: Behavioral VHDL for a 4-input AND gate.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND4 is
   port(w, x, y, z: in std_logic;
        f: out std_logic);
end AND4;

architecture Behavior of AND4 is
begin
   process(w, x, y, z) begin
      f <= w and x and y and z;
   end process;
end Behavior;
```


PARTICIPATION ACTIVITY 8.6.5: Behavioral VHDL descriptions for down-counter components.

Given the structural VHDL for a 4-bit down-counter, complete the behavioral VHDL description for each component.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DownCounter is
   port (clk, rst: in std_logic;
         cnt: in std_logic;
         C: out std_logic_vector(3 downto 0);
         tc: out std_logic);
end DownCounter;

architecture Structure of DownCounter is
   component Reg4
      port (clk, rst: in std_logic;
            ld: in std_logic;
            I: in std_logic_vector(3 downto 0);
            Q: out std_logic_vector(3 downto 0));
   end component;

   component Dec4
      port (a: in std_logic_vector(3 downto 0);
            s: out std_logic_vector(3 downto 0));
   end component;

   component NOR4
      port (w, x, y, z: in std_logic;
            f: out std_logic);
   end component;

   signal tempC: std_logic_vector(3 downto 0);
   signal decC: std_logic_vector(3 downto 0);

begin
   Reg4_1: Reg4 port map(clk, rst, cnt, decC, tempC);
   Dec4_1: Dec4 port map(tempC, decC);
   NOR4_1: NOR4 port map(tempC(3), tempC(2), tempC(1), tempC(0), tc);

   process(tempC) begin
      C <= tempC;
   end process;
end Structure;
```


1) Behavioral description for 4-bit decrementer:

```
entity Dec4 is
   port (a: in
std_logic_vector(3 downto 0);
         s: out
std_logic_vector(3 downto
0));
end entity Dec4;

architecture Behavior of Dec4
is
begin
   process(a) begin

   end process;
end Behavior;
```


Check Show answer

2) Behavioral description for 4 input

NOR gate:

```
entity NOR4 is
   port(w, x, y, z: in
std_logic;
        f: out std_logic);
end NOR4;

architecture Behavior of NOR4
is
begin
   process(w, x, y, z) begin

   end process;
end Behavior;
```

Check Show answer

8.7 RTL design (VHDL)

A VHDL entity describing a high-level state machine (HLSM) uses the same template as an FSM with the addition of:

1. Multi-bit data: An HLSM may have multi-bit inputs, outputs, and variables.
2. Arithmetic expressions: Assignments to variables and outputs may use arithmetic expressions like T := T + 1.
3. Equality and relational operations: Transitions may compare the values of inputs and variables like A < 40.

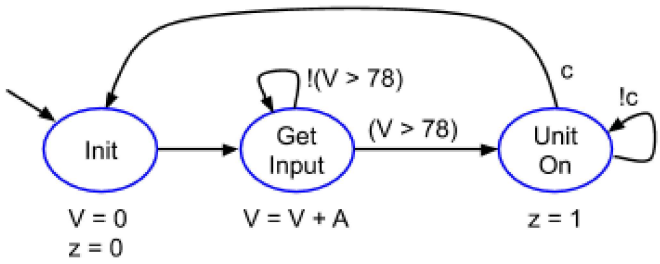
The following is an example of an HLSM described in VHDL, with each part discussed below.

An HLSM described using the following template avoids the earlier-described one-clock-cycle delay for datapath outputs by relying on the synthesis tool to create a more complex controller and datapath implementation. The design of the more complex controller and datapath is an advanced topic and not discussed here.

Figure 8.7.1: Example HLSM and VHDL description.

UnitControl

Inputs: c, A(8)
Outputs: z
Variables: V(8)



©zyBooks 10/17/21 22:08 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity UnitControl_HLSM is
   port (clk, rst : in std_logic;
         c : in std_logic;
         A : in unsigned(7 downto 0);
         z : out std_logic);
end UnitControl_HLSM;

architecture Behavior of UnitControl_HLSM is
begin

   process(clk)
      type UC_statetype is (UC_Init, UC_GetInput, UC_UnitOn);
      variable UC_State : UC_statetype;
      variable V : unsigned(7 downto 0);
   begin
      if (rising_edge(clk)) then
         if (rst = '1') then
            -- Initial state
            UC_State := UC_Init;
         else
            -- State transitions
            case UC_State is
               when UC_Init =>
                 UC_State := UC_GetInput;

               when UC_GetInput =>
                  if (V > 78) then
                     UC_State := UC_UnitOn;
                  elsif (not(V > 78)) then
                     UC_State := UC_GetInput;
                  end if;

               when UC_UnitOn =>
                  if (c = '1') then
                     UC_State := UC_Init;
                  elsif ((not c) = '1') then
                     UC_State := UC_UnitOn;
                  end if;

               when others =>
                  UC_State := UC_Init;
            end case;
         end if;

         -- State actions
         case UC_State is
            when UC_Init =>
               V := to_unsigned(0, 8);
               z <= '0';

            when UC_GetInput =>
               V := V + A;

            when UC_UnitOn =>
               z <= '1';

            when others =>
               V := to_unsigned(0, 8);
               z <= '0';
         end case;
      end if;
   end process;

end Behavior;
```


©zyBooks 10/17/21 22:08 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

Multi-bit inputs and outputs are declared as unsigned signals, like `A : in unsigned(7 downto 0);`. An ***unsigned*** signal is a vector that represents values that are non-negative (0 or greater). A ***vector*** is a type specification that indicates a signal or variable consists of multiple bits. A vector must specify the number and ordering of bits in the vector. Ex: unsigned(3 downto 0) defines a 4-bit unsigned vector with bits numbered 3, 2, 1, and 0.

A signed input, output, signal, or variable can be declared using the ***signed*** type. To use the unsigned or signed types, the statement `use IEEE.NUMERIC_STD.ALL;` must come before the entity declaration.

Each variable in the HLSM is also declared as an unsigned variable in the VHDL description.

VHDL provides several types for defining multi-bit inputs, outputs, signals, and variables. A std_logic_vector is another common method for defining vectors. This material uses unsigned and signed types because these types support arithmetic, relational, and equality operations and allow a VHDL description to use both unsigned and signed types at the same time.

PARTICIPATION ACTIVITY 8.7.1: VHDL: HLSM entity definitions.

Complete the VHDL description's input, output, and variable declarations for the given HLSM items. Keep names in the same order.

1) Inputs: a, b
Outputs: m

```
entity CheckMail_HLSM is

    port (clk, rst : in
std_logic;

            m : out
std_logic);

end CheckMail_HLSM;

...
```


Check Show answer

2) Inputs: d, E(16)
Outputs: r

```
entity LightOff_HLSM is

    port (clk, rst : in
std_logic;

            d : in std_logic;


            r : out
std_logic);

end LightOff_HLSM;

...
```


Check Show answer

3) Inputs: F(32), G(32)
Outputs: P(64)

```
entity SendData_HLSM is

     port (clk, rst : in
std_logic;

           F, G : in
unsigned(31 downto 0);


  );

end SendData_HLSM;

...
```

Check Show answer

4) Inputs: h
Outputs: Q(12)
Variables: M(12)

```
entity LoopCount_HLSM is

    port (clk, rst : in
std_logic;

          h : in std_logic;

          Q : out unsigned(11
downto 0));

end LoopCount_HLSM;


architecture Behavior of
UnitControl_HLSM is

begin

    process(clk)
```



```
...
```

Check Show answer

5) Inputs: a, C(16)
Outputs: t

Note: C is a temperature sensor capable of readings from -55 to 125 degrees Celsius.

```
entity ConvertReading_HLSM is

    port (clk, rst : in
std_logic;

          a : in std_logic;
```



```
           t : out
std_logic);

...
```

Check Show answer

VHDL is a strongly typed language, which requires explicit conversions to assign a value of one type to a value of another type. The ***to_unsigned()*** function converts an integer value to an unsigned value of the specified size. The first argument is the integer value and the second argument is the size of the unsigned result. Ex: to_unsigned(5, 16) converts the integer value 5 to 16-bit unsigned value.

Assignment statements often use arithmetic expressions. An ***expression*** is a combination of items, like signals, variables, inputs, literals, and operators, that evaluates to a value. Expressions mostly follow standard arithmetic rules, such as order of evaluation (items in parentheses first, * and / have precedence over + and -, etc.).

Table 8.7.1: Arithmetic operators.

Arithmetic operator	Description
+	Addition
-	Subtraction
*****	Multiplication
****	Exponentiation
/	Division
rem	Remainder
mod	Modulo

PARTICIPATION ACTIVITY 8.7.2: Arithmetic operators.

Determine the final value for the variable X. A, B, C, and X are 8-bit unsigned variables. Assume A is 4, B is 7, and C is 10.

1) X := B + A * C;

- 47
- 110

2) X := A + 1 * B + 2;

- 37
- 13
- 45

3) X := C / 4;

- 2.5
- 2

4) X := B * 3 / 2;

- 7
- 10

5) X := C rem 3;

- 0
- 1
- 3.33

An if-else expression commonly involves a relational or equality operator, which can be used with the unsigned type. A ***relational operator*** determines if a first operand is greater or less relative to a second operand. An ***equality operator*** determines if two operands are equal or not.

Table 8.7.2: Relational (first four) and equality (last two) operators.

Arithmetic operator	Description
a **<** b	a is less than b
a **>** b	a is greater than b
a **<=** b	a is less than or equal to b
a **>=** b	a is greater than or equal to b
a **=** b	a is equal to b
a **/=** b	a is not equal to b

PARTICIPATION ACTIVITY 8.7.3: State transitions and state actions using relational and arithmetic operators.

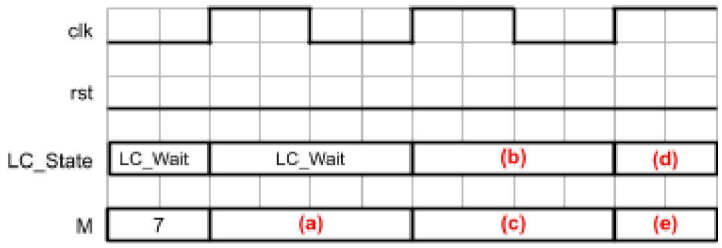
Animation captions:

1. Case statement evaluates the case expression and jumps to the matching case choice.
2. If-elsif statement compares variable V and updates state. V is not greater than 78, and UC_State is assigned with UC_GetInput.
3. State actions' case statement evaluates current UC_State and performs state actions. Variable V is updated.
4. Process waits for next rising clock.
5. V is greater than 78, and UC_State is assigned with UC_UnitOn.
6. State actions for UC_UnitOn state execute.

PARTICIPATION ACTIVITY 8.7.4: HLSM timing diagram.

Complete the timing diagram.

```
process(clk)
  -- ...
begin
  if (rising_edge(clk)) then
    if (rst = '1') then
      -- ...
    else
      -- State transitions
      case LC_State is
        when LC_Wait =>
          if (M < 10) then
            LC_State := LC_Wait;
          elsif (not(M < 10)) then
            LC_State := LC_Off;
          end if;

        -- ...
      end case;
    end if;

    -- State actions
    case (LC_State) is
      -- ...
      when LC_Wait =>
        M := M + 2;

      when LC_Off =>
        M := to_unsigned(0, 8);

      -- ...
    end case;
  end if;
end process;
```



1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

4) (d)

Check Show answer

5) (e)

Check Show answer

8.8 Datapath components: Behavioral (VHDL)

Multi-function register

A multi-function register can be described behaviorally in VHDL. The following example is a behavioral description of a multi-function register supporting parallel load, shift right, and shift left.

PARTICIPATION ACTIVITY 8.8.1: Multi-function register VHDL implementation.

Animation content:

undefined

Animation captions:

1. A 4-bit multi-function register with load, shift left, shift right can be described behaviorally within a process. The if statement maintains the priorities among the control inputs through the ordering of the elsif parts.
2. The signal R is used internally to store the value of the register. rst has highest priority. If rst = 1, then R is reset to 0000.

3. ld has the next highest priority. ld = 1 assigns R with the input D.
4. shr = 1 shifts the input D to the right by 1 bit using shr_in as the most significant bit. The order of the assignment statements do not matter as each statement is updated on the next rising edge of the clock.
5. shl = 1 shifts the input D to the left by 1 bit using shl_in as the least significant bit. Again, the order of the statements do not matter as each statement is updated on the rising edge of the clock.
6. Lastly, an assignment statement assigns Q with R. The statement is placed outside a process in an architecture, called a concurrent signal assignment, executing concurrently with any other processes.

PARTICIPATION ACTIVITY 8.8.2: Multi-function register VHDL implementation.

Given the following behavioral VHDL for a multi-function register.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MfReg4 is
   port (clk, rst: IN std_logic
         D: in std_logic_vector(3 downto 0);
         Q: out std_logic_vector(3 downto 0);
         ld, shl, shl_in, set: in std_logic);
end MfReg4;

architecture Behavior of MfReg4 is
   signal R: std_logic_vector(3 downto 0);
begin

   process (Clk)
   begin
      if (rising_edge(clk)) then
         if (rst = '1') then
            R <= "0000";
         elsif (ld = '1') then
            R <= D;
         elsif (shl = '1') then
            R(3) <= R(2);
            R(2) <= R(1);
            R(1) <= R(0);
            R(0) <= shl_in;
         elsif (set = '1') then
            R <= "1111";
         end if;
      end if;
   end process;
   Q <= R;
end behavior;
```


1) Which of the following functions is supported by the multi-function register implementation?

- shift right
- increment
-

set

2) Assume the register's initial value is 0001, D = 0111, ld = 0, shl = 0 (shl_in = 0), and set = 1. What is Q after the next rising clock?

- 0111
- 0010
- 1111

3) Assume the register's initial value is 0001. D = 1101, ld = 0, shl = 1 (shl_in = 0), and set = 1. What is Q after the next rising clock?

- 1101
- 0010
- 1111

4) Assume the register's initial value is 0001, D = 1010, ld = 1, shl = 1 (shl_in = 0), and set = 1. What is Q after the next rising clock?

- 1010
- 0100
- 1111

Comparator

A 4-bit magnitude comparator can be described behaviorally using an if statement that compares the inputs A and B.

PARTICIPATION ACTIVITY 8.8.3: Behavioral description of a 4-bit magnitude comparator.

Animation content:

undefined

Animation captions:

1. The behavioral description of a 4-bit magnitude comparator uses the arithmetic package that includes signed and unsigned vectors and the <, > operators.

2. With the use of the arithmetic package, inputs A and B can be defined as unsigned, signed, or a combination of both. Outputs gt, lt, and eq are defined as std_logic.
3. An if statement is defined within a process that checks if A is less than, greater than, or equal to B and sets the corresponding output to 1.

PARTICIPATION ACTIVITY 8.8.4: Behavioral description of a comparator.

Given the following behavioral VHDL:

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_arith.ALL;
entity Comparator4 is
   port (A: in unsigned(3 downto 0);
         B: in unsigned(3 downto 0);
         min: out unsigned(3 downto 0));
end Comparator4;

architecture Behavior of Comparator4 is
begin
   process (A, B)
   begin
      if (A < B) then
         min <= A;
      else
         min <= B;
      end if;
   end process;
end Behavior;
```

1) What does the Comparator4 entity output?

   - A 4-bit calculator that performs addition and subtraction.
   - The maximum of two numbers.
   - The minimum of two numbers.

2) What changes are required to make the entity output the maximum of two numbers?

   - Rename the output port as max.
   - Rename the output port as max, and change assignments in the if-else statement.
   - No such change is possible.

3) Which of the following revised entity declarations would create a

component that outputs the minimum of two signed values?

```
entity Comparator4 is
   port (A: in unsigned(3 downto 0);
         B: in unsigned(3 downto 0);
         min: out signed(3 downto 0));
end Comparator4;
```

```
entity Comparator4 is
   port (A: in signed(3 downto 0);
         B: in signed(3 downto 0);
         min: out unsigned(3 downto 0));
end Comparator4;
```

```
entity Comparator4 is
   port (A: in signed(3 downto 0);
         B: in signed(3 downto 0);
         min: out signed(3 downto 0));
end Comparator4;
```


Shifter using functions

While a 4-bit shifter can be implemented by assigning the individual bits as shown in the multi-function register example, for larger registers, assigning the individual bits can be time consuming. A simpler and more compact description utilizes predefined functions, shift_left() and shift_right(). The predefined functions are defined in the `ieee.numeric_std` package, and can be applied to signed and unsigned types.

PARTICIPATION ACTIVITY 8.8.5: Behavioral description of a 32-bit shifter.

Animation content:

undefined

Animation captions:

1. The shift_left() and shift_right() functions defined in the ieee.numeric_std package can be used to shift unsigned or signed values.
2. A 32-bit barrel shifter shifting to the right can shift the 32-bit input D by 0 to 31 positions. The ShiftByN entity has input D and output Q declared as 32-bit unsigned values. N is a 5-bit unsigned input that specifies the shift amount.

3. The statement Q <= shift_right(D, N); shifts the argument D by N positions to the right, and assigns Q with the value returned.

PARTICIPATION ACTIVITY 8.8.6: Behavioral shifter description.

1) Complete the statement to assign a 12-bit output Q with a 12-bit input D shifted right by 8 positions.

Q <= shift_right(____);

Check Show answer

2) Write a statement that assigns a 16-bit output Result with a 16-bit input Sum shifted by 2 positions to the right.

Check Show answer

3) Write a statement that assigns a 32-bit output Q with a 32-bit input D shifted to the left by a number of positions specified by a 5-bit input N.

Check Show answer

8.9 Example: Laser-based distance measurer (VHDL)

Entity and basic architecture

The HLSM of a laser-based distance measurer can be described in VHDL using an entity with a single process sensitive to the clock input. The VHDL description uses an entity to declare the inputs and outputs of the laser-based distance measurer.

PARTICIPATION ACTIVITY 8.9.1: Laser-based distance measurer: Top-level VHDL.

Animation content:

undefined

Animation captions:

1. The VHDL description uses the numeric_std package within the IEEE library for the unsigned data type and to enable using arithmetic operations and comparisons.
2. The LaserDistMeasurer entity declares the input and output signals. clk, rst, b, and s are declared as std_logic. The output D, which is the distance measured, is declared as a 15-bit unsigned signal.
3. The statetype type declaration declares the states S0, S1, S2, S3, and S4. The variable State is defined to hold the HLSM's current state. Dctr is a 15-bit unsigned signal to support the HLSM's local variable.
4. On a rising clock edge, if rst is 1, the State variable is assigned with the initial state S0. If rst is 0, the HLSM first executes the state transitions, and then executes the state actions based on the updated State.

PARTICIPATION ACTIVITY 8.9.2: Laser-based distance measurer entity and architecture.

1) D is declared as a(n) _____ output.
   - ○ std_logic
   - ○ unsigned

2) The process is executed anytime ____ changes.
   - ○ clk
   - ○ clk or rst

3) The IEEE _____ package supports arithmetic operations and the signed and unsigned types.
   - ○ numeric_std
   - ○ std_logic_1164

State transitions and state actions

The laser-based distance measurer HLSM is implemented using two case statements. The first case statement describes the state transitions and updates the State variable. The following case statement executes the state actions for the current state.

PARTICIPATION ACTIVITY 8.9.3: Laser-based distance measurer: state actions and state transitions.

Animation content:

undefined

Animation captions:

1. The process uses two case statements to describe the HLSM. The first case statement defines the HLSM's state transitions, assigning the State variable with the current state. The second case statement defines the state actions.
2. The State variable is immediately updated to reflect the current state. If the state was S0, then the State variable is assigned with S1, as state S0 has an implicit transition to S1.
3. For S1, an if-elsif statement is used to assign State with one of two possible values. If b is 1 (b = '1'), State is assigned with S2. If b is not 1 ((not b) = '1'), then State is assigned with S1.
4. The transitions for states S2, S3, and S4, are similarly converted. S2 and S4 have implicit transitions, so only an assignment statement is needed. S3 has multiple transitions, so an if-elsif statement is used.
5. The state actions case statement assigns the internal signals and outputs with appropriate values. S0 resets all the signals. S2 sets I to 1. S3 increments Dctr. S4 uses the shift_right function to divide Dctr by 2.
6. Both case statements use a when others => condition as a default statement to reset to S0 and prevent the HLSM from entering an unknown state for an unpredictable input.

PARTICIPATION ACTIVITY 8.9.4: Laser-based distance measurer: State transitions and state actions.

1) Which is the correct VHDL statement for state S3's state action?

   - D <= Dctr / 2;
   - Dctr <= Dctr + 1;
   - Dctr++;

2) Which is the state transition statement in state S2?

- State := S3;
- l <= '1';
- State := S1;

3) In state S1, which if-elsif statement condition is used for the transition to state S2?

- (b = '0')
- ((not b) = '1')
- (b = '1')

4) Which VHDL statement performs the following state action?
D = Dctr / 2;

- D <= Dctr << 1;
- D <= shift_right(Dctr, 1);
- D <= to_unsigned(0, 16);

Figure 8.9.1: VHDL for Laser-based distance measurer HLSM.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity LaserDistMeasurer is
port (clk, rst : in std_logic;
      b, s : in std_logic;
      l: out std_logic;
      D: out unsigned(15 downto 0));
end LaserDistMeasurer;


architecture behavior of LaserDistMeasurer is
   type statetype is (S0, S1, S2, S3, S4);
   variable State : statetype;
   signal Dctr: unsigned(15 downto 0);
begin
   process(clk)
   begin
      if (rising_edge(clk)) then
         if(rst = '1') then
            State := S0;
         else
         -- State transitions
            case State is
               when S0 =>
                  State := S1;
               when S1 =>
                  if(b = '1') then
                     State := S2;
```

```
                elsif ((not b) = '1') then
                   State := S1;
                end if;
             when S2 =>
                State := S3;
             when S3 =>
                if(s = '1') then
                   State := S4;
                elsif ((not s) = '1') then
                   State := S3;
                end if;
             when S4 =>
                State := S1;
             when others =>
                State := S0;
          end case;
       end if;
       -- State actions
       case State is
          when S0 =>
             l <= '0';
             D <= to_unsigned(0, 16);
             Dctr <= to_unsigned(0, 16);
          when S1 =>
             l <= '0';
          when S2 =>
             l <= '1';
          when S3 =>
             l <= '0';
              Dctr <= Dctr + 1;
          when S4 =>
             D <= shift_right(Dctr, 1);
          when others =>
             l <= '0';
             D <= to_unsigned(0, 16);
             Dctr <= to_unsigned(0, 16);
        end case;
     end if;
  end process;
end behavior;
```