7.1 Introduction to HDLs (Verilog)

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 7.1.1: 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 7.1.2: 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) Output 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 7.1.3: HDLs.

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

	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)

7.2 Combinational logic (Verilog)

In Verilog, a ***module*** defines a new component. A module definition starts with the module name, and a list of the module's ports. A ***port*** is a named input or output of a module. Each module input and output must also appear in an input or output declaration.

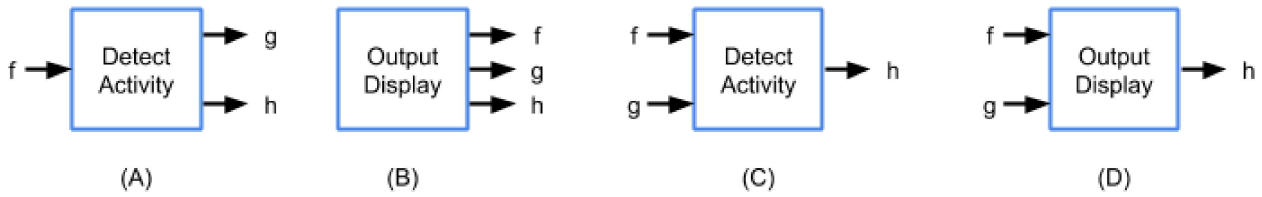
PARTICIPATION ACTIVITY 7.2.1: DoorOpenDetector module definition.

Animation captions:

1. Defines a module named DoorOpenDetector.
2. Inputs and outputs are listed in parentheses after the module name. A semicolon follows.
3. The input declaration specifies that ports b and c are inputs.
4. The output declaration specifies port z is an output. The reg keyword indicates a value will be assigned to the output.
5. The endmodule keyword ends the module definition.

PARTICIPATION ACTIVITY 7.2.2: Module definitions.

Match the module definition to the appropriate circuit.



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


```
module DetectActivity(f, g, h);
   input f;
   output reg g, h;
   ...
endmodule
```

```
module DetectActivity(f, g, h);
```

```
module OutputDisplay(f, g, h);
   output reg f, g, h;
   ...
endmodule
```

```
module OutputDisplay(f, g, h);
   input f, g;
   output reg h;
   ...
endmodule
```


Reset

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

begin and ***end*** surround an always procedure's statements.

PARTICIPATION ACTIVITY 7.2.3: Always procedure describing a combinational circuit.

Animation captions:

1. The always keyword defines the start of a procedure. The @ specifies the sensitivity list with inputs in parentheses.
2. Combinational circuits are sensitive to all circuit inputs.
3. Assignment statements assign values to outputs. & is the bitwise AND operator. ~ is the bitwise NOT operator.

PARTICIPATION ACTIVITY 7.2.4: Always procedures for combinational circuits.

Complete the always procedure for the provided combinational equations.

1) z = ab

```
module GetSensorReading(a, b,
z);
   input a, b;
   output reg z;

   [          ] @(a, b) begin
      z = a & b;
   end
endmodule
```

Check Show answer

2) m = j'
n = jk

```
module UpdateDisplay(j, k, m,
n);
   input j, k;
   output reg m, n;

   always @(          ) begin
      m = ~j;
      n = j & k;
   end
endmodule
```


Check Show answer

3) t = v's'm'

```
module SetTimer(s, t, m, v);
   input v, s, m;
   output reg t;

   always @(s,          )
begin
      t = ~s & ~v & ~m;
   end
endmodule
```

Check Show answer

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

```
module GetSensorReading(f, m,
p, q);
   input f;
   output reg m, p, q;

                      begin
      m = f;
      q = ~f;
      p = ~f;
   end
endmodule
```


Check Show answer

An ***assignment statement*** like y = a & b assigns the left-side variable with the result of the right-side expression. A ***variable*** stores a value, with the value being updated by assignment statements, and is declared using the reg keyword. An output can be declared as a variable as in: `output reg y;`

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

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

Table 7.2.1: Bitwise operators.

Operator	Description
&	Bitwise AND operation.
|	Bitwise OR operation.
~	Bitwise NOT operation.
^	Bitwise XOR operation.
^~	Bitwise XNOR operation.

PARTICIPATION ACTIVITY 7.2.5: Boolean expression in Verilog.

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

1) z = a + b

```
z = [          ];
```

Check Show answer

2) z = a'b

```
z = [          ] & 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 always procedure's sensitivity list is a common error. While the resulting always procedure is legal Verilog, the procedure does not describe combinational logic.

PARTICIPATION ACTIVITY 7.2.6: Detect the error in defining a module for combinational logic.

Click the erroneous code.

1)
```
module CloseDoor(a, b, c, z);
  input a, b, c;
  output reg z;
  always @ (a, b) begin
    z = ~a | ~b | ~c;
  end
endmodule
```

2)
```
module SoundAlarm(a, b, y, z);
  input a, b;
  output reg y, z;
  always @(a, b) y = a & b; z = a;
endmodule
```

3)
```
module LightController(a, b, z);
  input a, b;
  output z;
  always @ (a, b) begin
    z = a | b;
  end
endmodule
```


4)
```
module UnlockDoor(y, z, c, d);
  input c, d;
  output reg y, z;
  always @(c, d) begin
    y = ~c;
```

```
    z = c ^ d;
  end
end module
```

5)
```
module StartEngine(a, b, z)
  input a, b;
  output reg z;
  always @(a, b) begin
    z = ~(a & b);
  end
endmodule
```


PARTICIPATION ACTIVITY 7.2.7: HDL simulator: Always procedure.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 7.2.8: Combinational logic simulator: Designer provides input values; running the simulator generates output values.

```
module CombinationalLogicExample(a, b, c, z);
    input a, b, c;
    output reg z;
    always @(a, b, c) begin
        z = a & b & c;
    end
endmodule
```


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

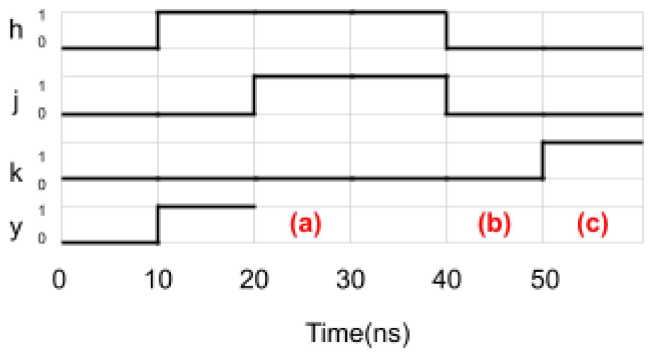
Run

Output:

PARTICIPATION ACTIVITY 7.2.9: Sensitivity list.

```
module AirController(h, j, k, y);
   input h, j, k;
   output reg y;

   always@(h, j, k) begin
      y = h | (~j & k);
   end
endmodule
```

Timing diagram



1) Value of y at (a).
   - 0
   - 1

2) The always procedure executes at 30 ns.
   - True
   - False

3) Value of y at (b).

- 0
- 1

4) The always procedure executes at 50 ns.

- True
- False

5) Value of y at (c).

- 0
- 1

7.3 Identifiers (Verilog)

An ***identifier*** is a designer-defined name used for items such as modules, inputs, and outputs. An identifier must start with a letter (A-Z, a-z) or underscore (_), followed by any number of letters (A-Z, a-z), digits (0-9), underscores (_), or dollar signs ($). Identifiers are case sensitive, meaning upper and lower case letters differ. So testEn and testEN are different.

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

PARTICIPATION ACTIVITY 7.3.1: Identifier validator.

Enter an identifier: _____ **Validate**

PARTICIPATION ACTIVITY 7.3.2: 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

5) _REGISTER_LOAD_
   - Valid
   - Invalid

6) 1_MuxSel
   - Valid
   - Invalid

7) MuxSel_19
   - Valid
   - Invalid

8) always
   - Valid
   - Invalid

Table 7.3.1: Verilog keywords.

always	end	ifnone	not	rnmos	tri
and	endcase	incdir	notif0	rpmos	tri0
assign	endconfig	include	notif1	rtran	tri1
automatic	endfunction	initial	or	rtranif0	triand

begin	endgenerate	inout	output	rtranif1	trior
buf	endmodule	input	parameter	scalared	trireg
bufif0	endprimitive	instance	pmos	showcancelled	unsigned
bufif1	endspecify	integer	posedge	signed	use
case	endtable	join	primitive	small	vectored
casex	endtask	large	pull0	specify	wait
casez	event	liblist	pull1	specparam	wand
cell	for	library	pulldown	strong0	weak0
cmos	force	localparam	pullup	strong1	weak1
config	forever	macromodule	pulsestyle_ondetect	supply0	while
deassign	fork	medium	pulsestyle_onevent	supply1	wire
default	function	module	rcmos	table	wor
defparam	generate	nand	real	task	xnor
design	genvar	negedge	realtime	time	xor
disable	highz0	nmos	reg	tran	
edge	highz1	nor	release	tranif0	
else	if	noshowcancelled	repeat	tranif1	

7.4 Testbench (Verilog)

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

1. A module instantiation that creates an instance of the module being tested.
2. A procedure for generating the sequence of input values to test the module.

The testbench creates an instance of the module being tested. A ***module instantiation*** creates an instance of a module, gives that instance a name, and specifies how the module's ports are

connected. Variables connect to the module's inputs, while wires connect to the module's output. A ***wire*** is a named connection within a module.

A ***port connection*** connects variables and wires to the inputs and outputs of the module instance. An ***ordered port connection*** specifies connections following the order of the module definition's port list.

This material appends _tb to the names for module instances, variables, and wires within the testbench.

PARTICIPATION ACTIVITY 7.4.1: A testbench instantiates the module to test, and prepares to set the module's input values and check output values.

Animation captions:

1. The testbench tests a module's functionality by providing a sequence of input values. The testbench itself has no inputs or outputs.
2. Variables are needed for inputs because the testbench assigns values to the module's inputs.
3. Wires are needed for outputs because the testbench observes the module's output values.
4. This statement Instantiates the module to test.
5. Ports are connected in the same order as in the module's definition.
6. Module instantiation consists of module name, instance name, and port connections.

PARTICIPATION ACTIVITY 7.4.2: Module instantiation.

Complete the testbench for each module. Use the naming convention of appending _tb for all variable, wire, and module instance names.

1)
```
module CheckSensor(a, b, m);
   input a, b;
   output reg m;

   // Module description
endmodule
```

```
module Testbench();
   
   wire m_tb;

   CheckSensor
CheckSensor_tb(a_tb, b_tb,
m_tb);

   // Designer-provided input
values
endmodule
```


Check Show answer

2)

```
module ControlValve(a, p, r);
   input a;
   output reg p, r;

   // Module description
endmodule
```

```
module Testbench();
   reg a_tb;
   
   ControlValve
ControlValve_tb(a_tb, p_tb,
r_tb);
   // Designer-provided input
values
endmodule
```


Check Show answer

3)

```
module SafetyCheck(a, x, y, z);
   input a;
   output reg x, y, z;

   // Module description
endmodule
```

```
module Testbench();
   reg a_tb;
   wire x_tb, y_tb, z_tb;


(a_tb, x_tb, y_tb, z_tb);
   // Designer-provided input
values
endmodule
```

Check Show answer

4)

```
module OpenVault(f, s, t, u);
   input f;
   output reg s, t, u;

   // Module description
endmodule
```

```
module Testbench();
   reg f_tb;
   wire s_tb, t_tb, u_tb;

   OpenVault OpenVault_tb(
);
   // Designer-provided input
values
endmodule
```

Check Show answer

The sequence of input values for testing a module appears within an initial procedure. An ***initial procedure*** defines a statement sequence that executes once at the start of simulation. Each unique sequence of values used to test a module is known as a ***test vector***.

PARTICIPATION ACTIVITY 7.4.3: A testbench generates input values using an initial procedure.

Animation captions:

1. Timescale directive defines length of each time unit and simulator's time precision separated by /.
2. Testbench assigns values to variables connected to module instance's inputs.
3. Initial procedure provides input values. Initial procedure executes once at the start of simulation.
4. Testbench should assign values to all variables 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.
5. #10 specifies a delay control that delays simulation for 10 time units, or 10 nanoseconds (time unit was defined earlier by timescale directive).
6. Simulator will advance time before executing the next statement.
7. Delay controls and assignment statements generate the desired sequence of input values to test the module.

A ***delay control*** delays simulation of a procedure for a specified time, starting with the hash character (#) followed by the number of time units to delay simulation. Ex: `#10` delays simulation for 10 time units. Delay controls can be prepended to statements like `#10 b_tb = 1;` or appear as separate statements like `#10;`.

A ***timescale directive*** defines the length of each time unit. The timescale directive starts with `` `timescale `` followed by the time unit specification, a slash character (/), and the time precision specification. Note the timescale directive starts with an accent grave character (`) not an apostrophe ('). The time precision defines how the simulator should internally keep track of time. Ex: `` `timescale 1 ns / 1 ns `` specifies each time unit is 1 ns (or nanosecond).

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

Table 7.4.1: Time unit specifications.

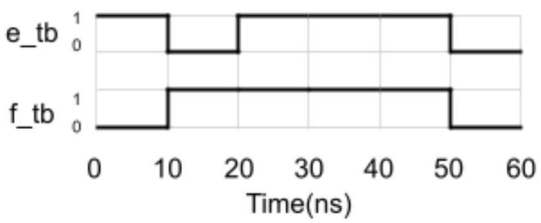
String	Time units

s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

PARTICIPATION ACTIVITY 7.4.4: Testbench: Generating waveforms.

```
`timescale 1 ns/ 1 ns

module SetTimer(e, f, m);
   input e, f;
   output reg m;

   always @(e, f) begin
      m = e | f;
   end
endmodule

module Testbench();
   reg e_tb, f_tb;
   wire m_tb;

   SetTimer SetTimer_tb(e_tb, f_tb, m_tb);

   initial begin
      (a)
      f_tb = 0;
      #10 (b)
      f_tb = 1;
      (c) e_tb = 1;
      (d) e_tb = 0;
      f_tb = 0;
   end
endmodule
```

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)

- #10
- 10
- #1

4) (d)

- #10
- #20
- #30

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

PARTICIPATION ACTIVITY 7.4.5: Testbench simulation process.

Animation content:

undefined

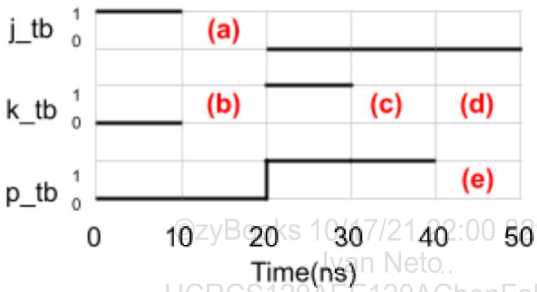
Animation captions:

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

PARTICIPATION ACTIVITY 7.4.6: Testbench simulation.

```
`timescale 1 ns/ 1 ns

module AlarmOff(j, k, p);
   input j, k;
   output reg p;

   always @(j, k) begin
      p = ~j & k;
   end
endmodule

module Testbench();
   reg j_tb, k_tb;
   wire p_tb;

   AlarmOff AlarmOff_tb(j_tb, k_tb, p_tb);

   initial begin
      j_tb = 1;
      k_tb = 0;
      #10 k_tb = 1;
      #10 j_tb = 0;
      #20 k_tb = 0;
   end
endmodule
```

Timing diagram



1) (a)

- 0
- 1

2) (b)

- 0
- 1

3) (c)

- 0
- 1

4) (d)

- 0
- 1

5) (e)

- 0

1

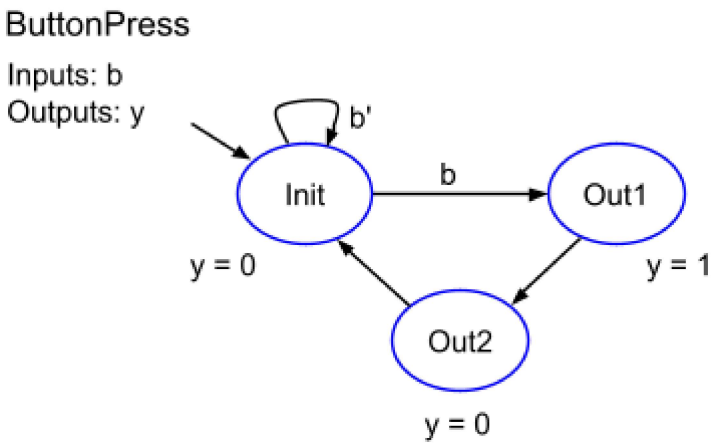
7.5 Sequential logic (Verilog)

A Verilog description of an FSM consists of:

1. Inputs and outputs including clock and reset inputs.
2. Parameters for defining 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 Verilog, with each part discussed further below.

Figure 7.5.1: Example ButtonPress FSM and Verilog description.



```
// ButtonPress_FSM
module BP_FSM(clk, rst, b, y);
   input clk, rst;
   input b;
   output reg y;

   localparam [1:0] BP_Init = 0,
                    BP_Out1 = 1,
                    BP_Out2 = 2;

   reg [1:0] BP_State;

   always @(posedge clk) begin
      if (rst) begin
         // Initial state
         BP_State = BP_Init;
      end
      else begin
         // State transitions
         case (BP_State)
            BP_Init: begin
               if (!b) begin
                  BP_State = BP_Init;
               end
               else if (b) begin
                  BP_State = BP_Out1;
               end
            end

            BP_Out1: begin
               BP_State = BP_Out2;
            end

            BP_Out2: begin
               BP_State = BP_Init;
            end

            default: begin
               BP_State = BP_Init;
            end
         endcase
      end

      // State actions
      case (BP_State)
         BP_Init: begin
            y = 0;
         end

         BP_Out1: begin
            y = 1;
         end

         BP_Out2: begin
            y = 0;
         end

         default: begin
            y = 0;
         end
      endcase
   end
endmodule
```


©zyBooks 10/17/21 22:00 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

PARTICIPATION ACTIVITY 7.5.1: First two parts of an FSM description in Verilog.

Animation captions:

1. Sequential circuits include clk and rst inputs.
2. A local parameter is a named constant value. For this FSM, each parameter is a 2-bit vector to match the size of the state encoding.
3. A comma-separated list specifies parameter names and values for each state.
4. A vector variable stores the state. The state variable is 2-bits to match the size of the state encoding.

A vector variable stores the state: `reg [1:0] BP_State;`. A ***vector*** is a type specification that indicates a variable (or input, output, wire, etc.) consists of multiple bits. A vector's ***range specification***, defined as [msb:lsb], indicates the position and ordering of bits in the vector, where msb is the most significant bit and lsb is the least significant bit. Ex: [3:0] defines a 4-bit vector with bits numbered 3, 2, 1, and 0.

Local parameters define state names and encodings. A ***local parameter*** is a named constant value local to a module. Ex: `localparam [1:0] BP_Init = 0;` defines a 2-bit parameter named BP_Init with the value 0. A local parameter definition starts with the keyword ***localparam*** followed by a comma-separated list of parameter names and values assigned to each name.

This material prepends a short prefix to each state variable's name and parameters. Ex: For an FSM named MotorController, the Verilog state variable might be MC_State, and all Verilog state parameter name's would begin with MC_.

PARTICIPATION ACTIVITY 7.5.2: FSM local parameters and state variable.

Click the erroneous statement.

1) localparam [1:0] SP_Init = 0,
SP_Next = 1,
SP_Out = 2;

reg SP_State;

2) localparam [2:0] LC_Init = 0,
LC_CheckSensor = 1,
LC_WriteData = 2,

```
                LC_ReadData = 3,
                LC_ClearMem = 4;

    reg [1:0] LC_State;
```

3)
```
localparam [1:0] DH_Init = 0,
                 DH_AlarmOn = 1;
                 DH_AlarmOff = 2;

reg [1:0] DH_State;
```

4)
```
localparam [1:0] BP_Init = 0,
                 BP_RisingEdge = 1,
                 BP_FallingEdge = 1;

reg [1:0] DH_State;
```

5)
```
localparam [2:0] ME_Init = 0,
                 ME_Activate = 1,
                 ME_Deactivate = 2;

reg [1:0] ME_State;
```


The FSM's description includes clk and rst inputs, for the clock and reset inputs of the sequential logic. The always procedure is sensitive to the rising clock edge, defined as `posedge clk`.

If the reset input is 1, the FSM's description assigns the state variable with the initial state, using an if-else statement. An ***if-else*** statement directs a procedure to execute one group of statements or another, depending on an expression's value. If the expression's value is true, the if's statements execute; otherwise the else's statements execute.

PARTICIPATION ACTIVITY 7.5.3: An if-else statement is used to assign the FSM's initial state.

Animation captions:

1. Initial state is unknown. The always procedure waits for a 0 to 1 transition on clk.
2. On the rising clock edge, if rst is 1, BP_State is assigned with the initial state.
3. The state variable stores the current state until the next rising clock edge.

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

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

Each step uses a case statement. A ***case statement*** selects one case item's statements to execute based on the case expression. Execution jumps to the ***case item*** whose constant expression matches the value of the case expression. The case item's statements are executed, and execution then jumps to endcase. The ***endcase*** keyword ends a case statement. The ***default*** case item executes if no other case item matches the case expression.

PARTICIPATION ACTIVITY 7.5.4: Case statement executes the case item for the current state, thus updating the state variable.

Animation captions:

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

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

PARTICIPATION ACTIVITY 7.5.5: State-by-state conversion of an FSM to case statements for state transitions and actions.

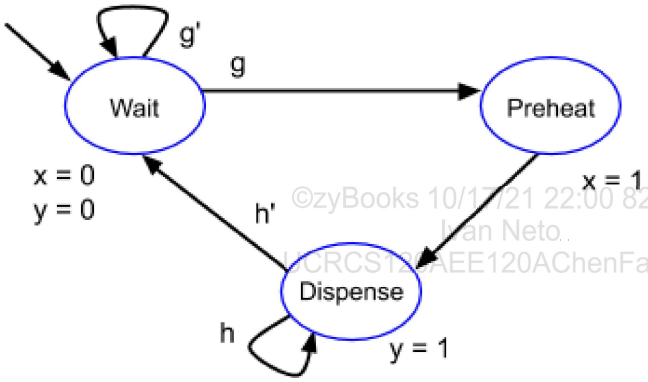
Animation captions:

1. Case statements specify state transitions and state actions.
2. States' transitions are specified state-by-state.
3. Good practice is to include a default case item, which executes if no other case item matches the case expression.
4. States' actions are also specified state-by-state.

PARTICIPATION ACTIVITY 7.5.6: Verilog: FSM state and output assignments.

```
module AD_FSM(clk, rst, g, h, x, y);
input clk, rst;
input g, h;
output reg x, y;

localparam [1:0] AD_Wait = 0,
                 AD_Preheat = 1,
                 AD_Dispense = 2;

reg [1:0] AD_State;

...
```

Inputs: g, h
Outputs: x, y



1) Complete the initial state assignment.

```
always @(posedge clk) begin
   if (rst) begin
      // Initial state
      AD_State =
;
```

Check Show answer

2) Complete the case statement.

```
// State transitions
case (
...
endcase
```

Check Show answer

3) Complete Wait's state transitions.

```
AD_Wait: begin
   if (!g) begin
      AD_State = AD_Wait;
   end
   else if (g) begin

   end
end
```


Check Show answer

4) Complete Preheat's state transitions.

```
AD_Preheat: begin

end
```

Check Show answer

5) Complete Wait's state actions.

```
AD_Wait: begin

end
```

Check Show answer

6) Complete Dispense's state actions.

```
AD_Dispense: begin

end
```

Check Show answer

A Verilog if-else expression commonly involves logical operators. A ***logical operator*** treats operands as being true or false, and evaluates to true or false.

Table 7.5.1: Logical operators.

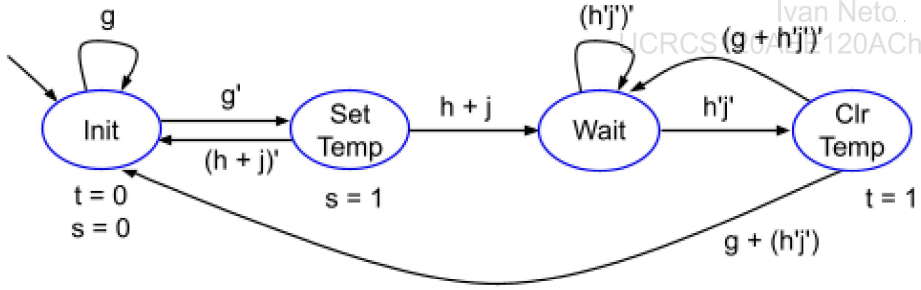
Equality and logical operator	Description
a **&&** b	Logical AND: true when both operands are true
a || b	Logical OR: true when at least one of the operands is true
!a	Logical NOT: true when operand is false (and false when operand is true)

PARTICIPATION ACTIVITY 7.5.7: Verilog: State transitions.

Convert each FSM state transition to Verilog (do not reorder the variables). Use logical operators (not bitwise operators).

QuickSelector

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



1) g'

```
QS_Init: begin
   if (                    )
begin
      QS_State = QS_SetTemp;
   end
   ...
end
```

Check Show answer

2) h + j

```
QS_SetTemp: begin
   ...
   else if (
                           ) begin
      QS_State = QS_Wait;
   end
end
```

Check Show answer

3) h'j'

```
QS_Wait: begin
   if (                    )
begin
      QS_State = QS_ClrTemp;
   end
   ...
end
```

Check Show answer

4) g + (h'j')

```
QS_ClrTemp: begin
   if (                    )
begin
      QS_State = QS_Init;
   end
   ...
end
```

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 local parameter. An always procedure uses this parameter to repeatedly generate the clock input. The initial procedure also uses this parameter to delay simulation for a clock cycle.

PARTICIPATION ACTIVITY 7.5.8: An always procedure used in a testbench to generate the clock input for an FSM.

Animation captions:

1. Testbench uses a procedure to generate a clock signal.
2. Always procedure generates clock throughout simulation.
3. Initial procedure can delay simulation for a clock cycle using CLK_PERIOD parameter.

PARTICIPATION ACTIVITY 7.5.9: Clock procedure.

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

   ○ True

False

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

```
initial begin
   clk_tb = 0;
   #(CLK_PERIOD / 2);
   clk_tb = 1;
   #(CLK_PERIOD / 2);
end
```


True

False

7.6 Datapath components: Structural (Verilog)

Structural description of datapath components

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

PARTICIPATION ACTIVITY 7.6.1: Structural Verilog description of a 4-bit carry ripple adder.

Animation content:

undefined

Animation captions:

1. A full adder can be described behaviorally using the equations for each output s and co. The module defines the inputs and outputs. The always procedure defines 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 Verilog 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 design has three internal wires, co1, co2, and co3, that are used for internal connection between the full-adders.

5. The four full adder components are uniquely identified by 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 7.6.2: Structural Verilog 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 Verilog description is advantageous when smaller components of the same type are used several times in the datapath.
   - ○ True
   - ○ False

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 7.6.3: Structural Verilog description of a 4-bit up-counter.

Animation content:

undefined

Animation captions:

1. The UpCounter module defines the counter's inputs and outputs, consisting of a clock input (clk), a reset input (rst), a count enable control input (cnt), the 4-bit count output (C), and a terminal count output (tc).
2. The UpCounter's module uses three components Reg4, Inc4, and AND4. 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, an always procedure is used to assign the output Q with tempC.

PARTICIPATION ACTIVITY 7.6.4: Structural Verilog description of a 4-bit up-counter.

1) To design a 32-bit up-counter using structural Verilog, 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 always procedure 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 7.6.1: Behavioral Verilog for a 4-bit register

register.

```
`timescale 1ns / 1ps
module Reg_4(clk, rst, ld, I, Q);
   input clk, rst;
   input ld;
   input [3:0] I;
   output reg [3:0] Q;

   always @(posedge clk) begin
      if (rst) begin
         Q = 0;
      end
      else if (ld) begin
         Q = D;
      end
   end
endmodule
```


Figure 7.6.2: Behavioral Verilog for a 4-bit incrementer.

```
`timescale 1ns / 1ps
module Inc4(a, s);
   input [3:0] a;
   output reg [3:0] s;

   always @(a) begin
      s = a + 1;
   end
endmodule
```

Figure 7.6.3: Behavioral Verilog for a 4-input AND gate.

```
`timescale 1ns / 1ps
module AND4(w, x, y, z, f);
   input w, x, y, z;
   output reg f;

   always @(w, x, y, z) begin
      f = w & x & y & z;
   end
endmodule
```

PARTICIPATION ACTIVITY 7.6.5: Behavioral Verilog descriptions for down-counter components.

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

```
module DownCounter(clk, rst, cnt, C, tc);
   input clk, rst;
   input cnt;
   output [3:0] C;
   reg [3:0] C;
   output tc;

   Reg4 Reg4_1(clk, rst, cnt, incC, tempC);
   Dec4 Dec4_1(tempC, incC);
   NOR4 NOR4_1(tempC(3), tempC(2), tempC(1), tempC(0), tc);

   always @(tempC)
   begin
      C = tempC;
   end
endmodule
```


1) Behavioral description for 4-bit decrementer:

```
`timescale 1ns / 1ps
module Dec4(a, s);
   input [3:0] a;
   output reg [3:0] s;

   always @(a) begin

   end
endmodule
```

Check Show answer

2) Behavioral description for 4 input NOR gate:

```
`timescale 1ns / 1ps
module NOR4(w, x, y, z, f);
   input w, x, y, z;
   output reg f;

   always @(w, x, y, z) begin

   end
endmodule
```


Check Show answer

7.7 RTL design (Verilog)

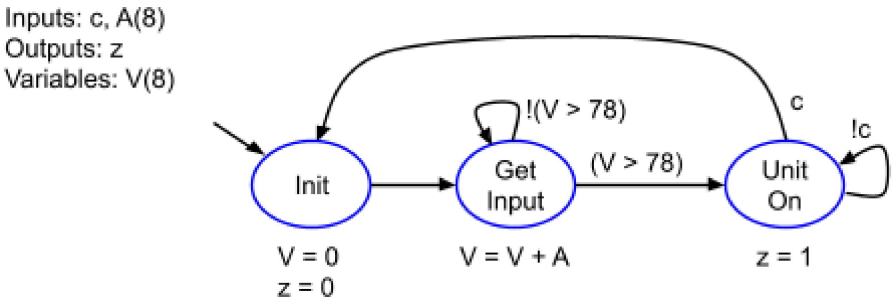
A Verilog module describing a high-level state machine (HLSM) uses the same template as an FSM with the addition of:

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 HSLM described in Verilog, with each part discussed below.

An HLSM described using this 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 7.7.1: Example HLSM named UnitControl, and Verilog description.




```
module UnitControl_HLSM(clk, rst, c, A, z);
   input clk, rst;
   input c;
   input [7:0] A;
   output reg z;

   localparam [1:0] UC_Init = 0,
                    UC_GetInput = 1,
                    UC_UnitOn = 2;

   reg [1:0] UC_State; // HLSM State

   reg [7:0] V;        // HLSM variables

   always @(posedge clk) begin
      if (rst) begin
         // Initial state
         UC_State = UC_Init;
      end
      else begin
```

```
      else begin
         // State transitions
         case (UC_State)
            UC_Init: begin
               UC_State = UC_GetInput;
            end

            UC_GetInput: begin
               if (V > 78) begin
                  UC_State = UC_UnitOn;
               end
               else if (!(V > 78)) begin
                  UC_State = UC_GetInput;
               end
            end

            UC_UnitOn: begin
               if (c) begin
                  UC_State = UC_Init;
               end
               else if (!c) begin
                  UC_State = UC_UnitOn;
               end
            end

            default: begin
               UC_State = UC_Init;
            end
         endcase
      end

      // State actions
      case (UC_State)
         UC_Init: begin
            V = 0;
            z = 0;
         end

         UC_GetInput: begin
            V = V + A;
         end

         UC_UnitOn: begin
            z = 1;
         end

         default: begin
            V = 0;
            z = 0;
         end
      endcase
   end

endmodule
```


Multi-bit inputs and outputs are declared as vector inputs and outputs, like `input [7:0] A;`. Each variable in the HLSM is also declared as a variable in the Verilog description.

By default, all vector inputs, outputs, and variables are unsigned integers. An ***unsigned*** variable represents values that are non-negative (0 or greater). A signed input, output, or variable is declared using the ***signed*** keyword like `reg signed [7:0] B;`.

PARTICIPATION

ACTIVITY 7.7.1: Verilog: HLSM module definitions.

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

1) Inputs: a, b
Outputs: m


```
module CheckMail_HLSM(clk,
rst, a, b, m);

    input clk, rst;

    

    output reg m;

...
```

Check Show answer

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

```
module LightOff_HLSM(clk,
rst, d, E, r);

    input clk, rst;

    input d;

    

    output reg r;

...
```

Check Show answer

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

```
module SendData_HLSM(clk,
rst, F, G, P);

    input clk, rst;

    input [31:0] F, G;

    

...
```


Check Show answer

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

```
module LoopCount_HLSM(clk,
rst, h, Q);

    input clk, rst;

    input h;

    output reg [11:0] Q;

    

...
```

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.

```
module ConvertReading_HLSM(clk,
rst, a, C, t);

    input clk, rst;

    input a;


    output reg t;

...
```


Check Show answer

Assignment statements often use arithmetic expressions. An ***expression*** is a combination of items, like 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 7.7.1: Arithmetic operators.

Arithmetic operator	Description
+	addition
-	subtraction
*****	multiplication
/	division
%	modulo (remainder)

PARTICIPATION ACTIVITY 7.7.2: Arithmetic operators.

Determine the final value for the variable X. A, B, C, and X are 8-bit vectors. 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 % 3;

- 0
- 1
- 3.33

An if-else expression commonly involves a relational or equality operator. 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 7.7.2: 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 7.7.3: State transitions and state actions using relational and arithmetic operators.

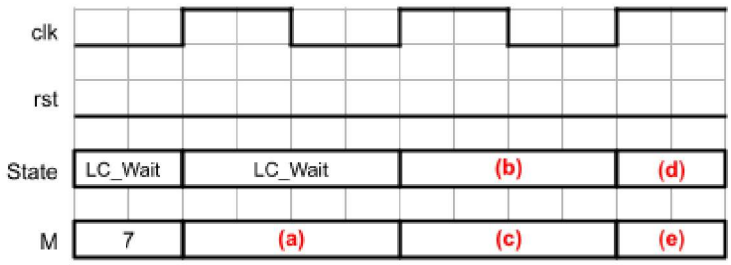
Animation captions:

1. Case statement evaluates the case expression and jumps to the matching case item.
2. If-else 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. Procedure 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 7.7.4: HLSM timing diagram.

Complete the timing diagram.

```
always @(posedge clk) begin
   if (rst) begin
      // ...
   end
   else begin
      // State transitions
      case (LC_State)
         LC_Wait: begin
            if (M < 10) begin
               LC_State = LC_Wait;
            end
            else if (!(M < 10)) begin
               LC_State = LC_Off;
            end
         end
         // ...
      endcase
   end

   // State actions
   case (LC_State)
      // ...
      LC_Wait: begin
         M = M + 2;
      end

      LC_Off: begin
         M = 0;
      end
      // ...
   endcase
end
```



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

7.8 Datapath components: Behavioral (Verilog)

Multi-function register

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

PARTICIPATION ACTIVITY 7.8.1: Multi-function register Verilog 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 module. The if statement maintains the priorities among the control inputs through the ordering of the elsif parts.
2. The register R is used internally to store the value of the shift 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 the always procedure, called a continuous assignment statement, executing whenever R changes.

PARTICIPATION ACTIVITY 7.8.2: Multi-function register Verilog implementation.

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

```
`timescale 1 ns/1 ns

module MfReg4(clk, rst, I, Q, ld, shl, shl_in, set);

   input clk, rst;
   input ld, shl, shl_in, set;
   input [3:0] I;
   output [3:0] Q;

   reg [3:0] R;

   always @(posedge clk) begin
      if (rst == 1)
         R = 0;
      else if (ld == 1)
         R = I;
      else if (shl == 1) begin
         R[3] = R[2];
         R[2] = R[1];
         R[1] = R[0];
         R[0] = shl_in;
      end
      else if(set == 1)
         R[3] = 1;
         R[2] = 1;
         R[1] = 1;
         R[0] = 1;
      end
   end
   assign Q = R;
endmodule
```

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 7.8.3: Behavioral description of a 4-bit magnitude comparator.

Animation content:

```
undefined
```

Animation captions:

1. The inputs A and B are 4-bit wide and can be defined as unsigned, signed, or a combination of both. Outputs gt, lt, and eq are single bits.
2. An if statement is defined within an always procedure that checks if A is less than, greater than, or equal to B and sets the corresponding output to 1.

PARTICIPATION ACTIVITY 7.8.4: Behavioral description of a comparator.

Given the following behavioral Verilog, answer the following questions.

```
module Comparator4(A, B, min);

   input unsigned [3:0] A;
   input unsigned [3:0] B;
   output reg [3:0] min;

always @(A, B) begin
   if (A < B) begin
      min = A;
   end
   else begin
      min = B;
   end
end
endmodule
```

1) What does the Comparator4 module output?

   - An 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 module 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.

Shifter using shift operators

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 the built-in shift operators << and >>. The logical right and left shift operators, ">>" and "<<", shift the first operand by the number of bit locations specified by the second operand, filling in the most significant or least significant bit positions with 0.

PARTICIPATION ACTIVITY 7.8.5: Behavioral description of a 32-bit shifter.

Animation content:

undefined

Animation captions:

1. A 32-bit right shifter has a 32-input D, and 32-bit output Q, and a 5-bit input N that indicates the shift amount.
2. The shift right operator >> shifts the input D by N positions to the right.

PARTICIPATION ACTIVITY 7.8.6: 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 = _____;

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

Arithmetic shift

For computations like multiply and divide using signed numbers, an arithmetic shift can be used to preserve the sign. The arithmetic left shift operator, "<<<", shifts the first operand left by the number of bit locations specified by the second operand, filling in the least significant bit positions with 0. The arithmetic right shift operator, ">>>", shifts the first operand right by the number of bit locations specified by the second operand, filling in the most significant bit positions with the original value of the most significant bit, in order to retain the sign.

7.9 Example: Laser-based distance measurer (Verilog)

Top-level Verilog

The HLSM of a laser-based distance measurer can be described in Verilog using an always procedure sensitive to a rising clock edge. The Verilog description uses a module to declare the inputs and outputs of the laser-based distance measurer.

PARTICIPATION ACTIVITY 7.9.1: Laser-based distance measurer: Top-level Verilog.

Animation content:

undefined

Animation captions:

1. The LaserDistMeasurer module declares the input and output ports, clk, rst, b, and s. The output port D, which is the distance measured, is declared as a 15-bit vector.
2. The localparam declares the states S0, S1, S2, S3, and S4. The vector variable State holds the HLSM's current state. Dctr is a 15-bit vector variable to support the HLSM's local variable.
3. 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 7.9.2: Laser-based distance measurer: top-level Verilog.

1) D is declared as a(n) _____ output.

- signed
- unsigned

2) The always procedure is executed anytime ____ changes.

- clk
- clk or rst

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 7.9.3: Laser-based distance measurer: state actions and state transitions.

Animation content:

undefined

Animation captions:

1. The always procedure 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-else statement is used to assign State with one of two possible values. If b is 1 (b), State is assigned with S2. If b is not 1 (!b), 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-else statement is used.
5. The state actions case statement assigns the internal variables and outputs with appropriate values. S0 resets all variables and outputs. S2 sets I to 1. S3 increments Dctr. S4 uses the shift operator >> to divide Dctr by 2.
6. Both case statements use a default condition that resets the HLSM to S0 and prevents the HLSM from entering an unknown state for an unpredictable input.

PARTICIPATION ACTIVITY 7.9.4: Laser-based distance measurer: State transitions and state actions.

1) Which is the correct statement for state S3's state action?
   - D = Dctr / 2;
   - Dctr = Dctr + 1;
   - Dctr := Dctr + 1;

2) Which is the state transition statement in state S2?
   - State = S3;
   - I = 1;
   - State = S1;

3) In state S1, which if-else statement condition is used for the transition to state S2?
   - (!b)
   - (!b == '1')
   - (b)

4) Which statement performs the following state action?
   D = Dctr / 2;
   - D = Dctr << 1;
   - D = Dctr >> 1;
   - D = 0;

Figure 7.9.1: Verilog for Laser-based Distance

Measurer HLSM.

```
module LaserDistMeasurer_HLSM(clk, rst, b, s, l, D);
   input clk, rst;
   input b, s;
   output reg l;
   output reg [15:0] D;

   localparam [3:0] S0 = 0,
                    S1 = 1,
                    S2 = 2,
                    S3 = 3,
                    S4 = 4;

   reg [3:0] State;        // HLSM State
   reg [15:0] Dctr;        // HLSM Variable

   always @(posedge clk) begin
      if (rst) begin
         // Initial state
         State = S0;
      end
      else begin
         // State transitions
         case (State)
            S0: begin
               State = S1;
            end
            S1: begin
               if (b) begin
                  State = S2;
               end
               else if (!b) begin
                  State = S1;
               end
            end
            S2: begin
               State = S3;
            end
            S3: begin
               if (s) begin
                  State = S4;
               end
               else if (!s) begin
                  State = S3;
               end
            end
            S4: begin
               State = S1;
            end
            default: begin
               State = S0;
            end
         endcase
      end

      // State actions
      case (State)
         S0: begin
            l = 0;
            D = 0;
            Dctr = 0;
         end
         S1: begin
            l = 0;
         end
         S2: begin
            l = 1;
```


©zyBooks 10/17/21 22:00 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

```
            end
            S3: begin
               l = 0;
               Dctr = Dctr + 1;
            end
            S4: begin
               D = Dctr >> 1;
            end
            default: begin
               l = 0;
               D = 0;
               Dctr = 0;
            end
         endcase
      end
endmodule
```