Lab 4: Sequential Logic Design

EE 316: Digital Logic Design

1 Overview

This lab is intended to familiarize you with how to implement finite state machines and other sequential logic in Verilog. By the end of this lab, you should be able to:

- describe structural, dataflow, and behavioral modeling.
- use dataflow and behavioral modeling to describe finite state machines (FSMs).
- describe and implement a clock divider with a given frequency in Verilog.
- implement a system described by an FSM in Verilog.
- explain the utility of time-multiplexing.

Before starting this lab, you will need to review FSMs (also called "automata") from lecture. You should be able to draw a state graph from a problem description, generate a state table, and draw an FSM controller circuit.

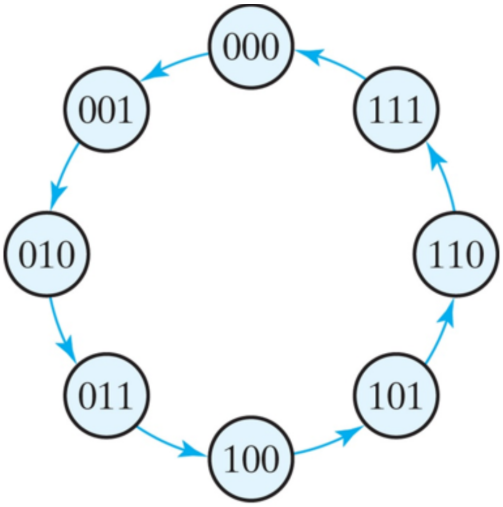
2 Background

2.1 Clock Dividers

A **clock** is a periodic signal that is used as a timing mechanism for sequential circuits such as FSMs. In an FSM, each state lasts one clock cycle. Because an FSM changes state in response to the clock, FSMs are called **synchronous**. The Basys3 board has a 100MHz internal clock that we can use; however, we usually want to run FSMs at much slower speeds. **Clock division** is the process by which we generate slower clocks from fast clocks, and the clocks we make are called **clock dividers**. Clock dividers are usually made with a **binary up-counter** that increments at the speed of the main clock. By masking for certain bits in the up-counter, we can get periodic signals that can serve as new, slower clocks.

An FSM and state transition table for a 3-bit binary up-counter are shown in Figure 1. In an up-counter, the states are usually named with the outputs, so state 0 outputs 000, state 1 outputs 001, etc. so that the present state number is what the system outputs. Unconditional transitions connect state 0 to state 1, 1 to 2, 2 to 3, etc. so the counter runs indefinitely.

If we draw a timing diagram showing when each bit in the counter is high in relation to the FSM clock (Figure 2), we can see that each bit in the counter actually transitions periodically - but at a slower rate than the FSM clock. Bit 0 transitions on every rising edge of the main clock; bit 1 transitions on every other rising edge; and bit 2 transitions on every 4 rising edges. Therefore, the period of bit 0 is 2 clock cycles, and the frequency of the signal on bit 0 is 1/2 the frequency



(a) State Machine

Present State			Next State		
$Q_2(t)$	$Q_1(t)$	$Q_0(t)$	$Q_2(t+1)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

(b) State Transition Table

Figure 1: 3-bit binary up-counter

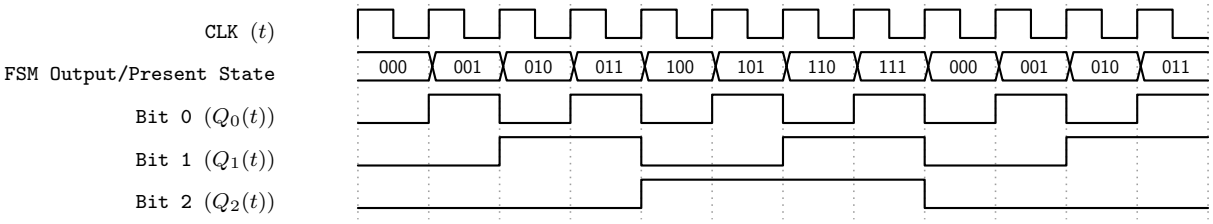


Figure 2: Timing diagram for 3-bit up-counter.

of the original clock. Similarly, the period of bit 1 is 4 clock cycles, which means its frequency is 1/4 the original clock. The period of bit 2 is 8 clock cycles, which means its frequency is 1/8 of the original clock. These clock dividers are also called "divide-by-2", "divide-by-4", and "divide-by-8" clocks, respectively. In this way, we can generate clocks that divide the original clock frequency by a power of 2.

"Divide-by-2^n" clock dividers are easy to generate with bit-accessing in Verilog, as shown in the code snippets below. Note that the sensitivity list of the always block (inside parentheses) now contains "posedge clk" instead of an asterisk. This indicates that the always block will now only run on the positive (rising) edge of the clock; therefore, the counter will only increment on the rising edges of the input clock.

```
wire div2, div4, div8;
reg [2:0] count = 0;          //Declare and initialize up-counter register
assign div2 = count[0];       //Divide-by-2 clock (50MHz on Basys3)
assign div4 = count[1];       //Divide-by-4 clock (25MHz on Basys3)
assign div8 = count[2];       //Divide-by-8 clock (12.5MHz on Basys3)
always @(posedge clk) count = count + 1;
```

We can also divide the clock by a non-power of 2 by adjusting the maximum value of the up-counter. Suppose we wanted to create a divide-by-3 clock. While there are several ways to go about this, one simple way to do it is to adjust the maximum value of the counter. In Verilog:

```
wire div3;
```

```
reg [1:0] count = 0;          //Declare and initialize up-counter register
assign div3 = count[1];       //Divide-by-3 clock (33.3MHz on Basys3)
always @(posedge clk) begin
    if(count == 3) count = 0; //Modify max value of counter (set to 2)
    else count = count + 1;
    end
```

Note that this solution only works if we do not care about the duty cycle of our divide-by-3 clock (how long the clock is high in relation to its total period). If we wanted a divide-by-3 clock with a 50% duty cycle (a very real and interesting problem in digital logic design), we would need to use some logic to generate the signal. (We would have to feed the output of the up-counter into a falling-edge flip-flop and OR the output of the up-counter and the falling-edge flip-flop to generate the 50% duty cycle.)

2.2 Seven-Segment Display: Time-Multiplexing

One key design problem raised in Lab 3 with the seven-segment display is that it cannot display four different numbers at a time because all the segments in the same position across the display are effectively shorted together when they are enabled. To solve this problem, instead of turning all four digits on at the same time, we can cycle through the digits, turning each one on individually. This design is called **time-multiplexing** - we use the clock to cycle through the digits, showing one at a time. If we speed up the clock to about 2-5kHz, the display will appear that all four digits are on simultaneously.

To implement this on the board, consider the timing diagram on page 17 of the Basys3 reference manual (on Canvas, reproduced in Figure 3). The anodes are enabled one at a time (remember: they are active-low devices), and the segments (cathodes) corresponding to the desired digit are displayed in the same window of time as the anode. This system behavior can be captured in an FSM, as you will do in this lab.

2.3 Blocking vs. Non-blocking: Assignment in Verilog

Since Verilog is a hardware description language (HDL), timing plays a key role in assignment. Consider the following always block:

```
always @(posedge clk) begin
    b = c; //a, b, c are regs
    a = b;
    end
```

If we didn't know any better, we would logically say that since $a = b$ implies that a is wired to b and similarly for $b = c$, if the always block runs, $a = c$. This is problematic because the value of b is always destroyed, rendering the variable meaningless, and a will always hold the same value as c unless it is updated elsewhere.

To fix this, Verilog has defined two kinds of assignment: blocking and non-blocking (see Table 1). **Blocking** (denoted by '=') executes the statements sequentially, *blocking* execution of the next statement until the current statement completes. In the above example, $a = 2, b = 3, c = 4$, then after execution, $a = 4, b = 4, c = 4$. **Non-blocking** (denoted by '<=') executes the statements simultaneously: if in the above example, $a = 2, b = 3, c = 4$, then after execution, $a = 3, b = 4, c = 4$. Note that a was assigned the old value of b instead of the new value because the assignment of c

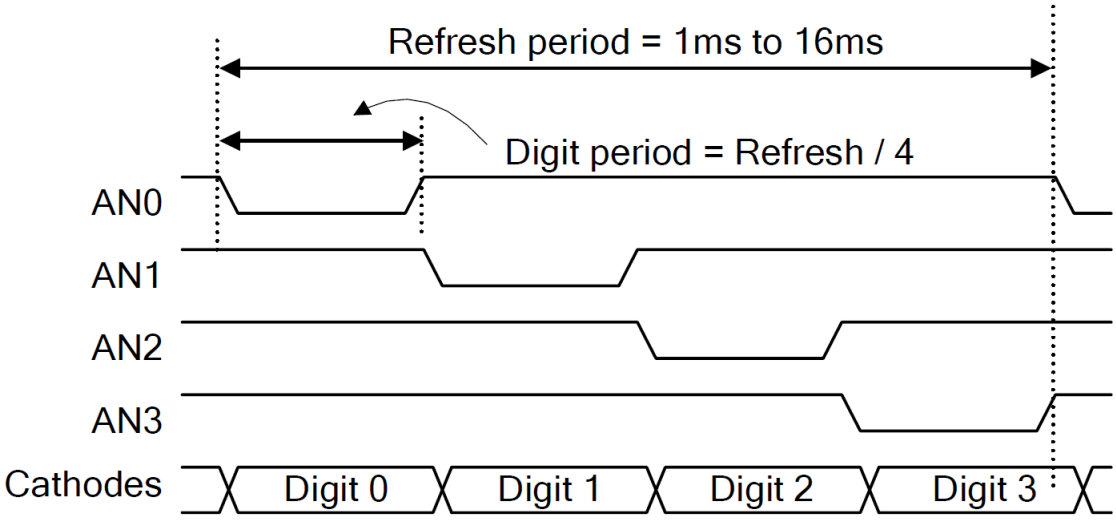


Figure 19. Four digit scanning display controller timing diagram.

Figure 3: Seven-segment display timing diagram

to b happens at the same time: when $a = b$ is executed, b holds the value 3 until the end of the assignment. (Another way to think about it is that these are non-ideal flip-flops in series.)

Blocking	Non-blocking

always @(**posedge** clk) **begin**
b = c; // *b = 4*
a = b; // *a = 4*
end
	always @(**posedge** clk) **begin**
	b <= c; // *b = 4*
	a <= b; // *a = 3*
	end

Table 1: Blocking vs non-blocking (assuming $a = 2, b = 3, c = 4$ initially)

3 Procedure

3.1 Part A: Flight Attendant Call System

In this part, we will implement a flight attendant call system. The call system has two inputs, `call` and `cancel`, and one output, `light`. The inputs correspond to `btnL` and `btnR` on the board respectively, and the output corresponds to `LED0` on the board.

The functionality of the system is defined as follows: when the `call` button is pressed, the LED mapped to `light` turns on and stays on. When the `cancel` button is pressed and the `call` button is released, the LED turns off. The buttons will never be pressed at the same time, and repeat press/releases of the same button should not change the state of the system.

1. Draw a state graph and a state table for this system. Make sure your state table matches Table 2.

2. Create a project, and add a design source. Edit the design source so it looks like Figure 4a.

Call	Cancel	Current State (Q)	Output (D)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 2: Part A Truth Table

```
`timescale 1ns / 1ps

module call_system(
    input clk,
    input call,
    input cancel,
    output reg light_state //Current state
    );

    reg next_state; //Holds next state
    initial light_state  = 0; //Initilizes output/current state to 0

    always @(*) begin
        case({call, cancel, light_state}) //Specify FSM/truth table here
            3'b000: next_state = 1'b0;
            3'b001: next_state = 1'b1;
            3'b010: next_state = 1'b0;
            3'b011: next_state = 1'b0;
            3'b100: next_state = 1'b1;
            3'b101: next_state = 1'b1;
            3'b110: next_state = 1'b1;
            3'b111: next_state = 1'b1;
            default: next_state = 1'b0;
        endcase
    end

    always @(posedge clk) begin //Sensitivity list triggers on rising edge of clk
        light_state <= next_state; //Next state gets clocked in on clk
    end

endmodule
```

(a) Design File

```
module tb_call_system;

//Initialize my inputs to 0
reg clk = 0;
reg call = 0;
reg cancel = 0;
wire light_state;

call_system uut(
    .clk(clk),
    .call(call),
    .cancel(cancel),
    .light_state(light_state)
    );

//Simulate clock signal for FSM
always #5 clk = ~clk;

initial begin
    #20 call = 0; cancel = 0;
    #20 call = 1; cancel = 0;
    #20 call = 0; cancel = 1;
    #20 call = 1; cancel = 1;
    #20 call = 0; cancel = 0;
    #20 call = 1; cancel = 0;
    #20 cancel = 1;
    #20 cancel = 0;
    #20 call = 0; cancel = 1;
    #20 call = 0; cancel = 0;
end

endmodule
```

(b) Testbench File

Figure 4: Part A Code

```
// Clock signal
set_property PACKAGE_PIN W5 [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}]

//Buttons
set_property PACKAGE_PIN W19 [get_ports {call}]
set_property IOSTANDARD LVCMOS33 [get_ports {call}]
set_property PACKAGE_PIN T17 [get_ports {cancel}]
set_property IOSTANDARD LVCMOS33 [get_ports {cancel}]

//LED
set_property PACKAGE_PIN U16 [get_ports {light_state}]
set_property IOSTANDARD LVCMOS33 [get_ports {light_state}]
```

Figure 5: Constraints file for part A

```
`timescale 1ns / 1ps

module call_system(
    input clk,
    input call,
    input cancel,
    output reg light_state //Current state
    );

    wire next_state; //Holds next state
    initial light_state  = 0; //Initilizes output/current state to 0

    assign next_state = 0; // Replace `0' with your logic expression.

    always @(posedge clk) begin //Sensitivity list triggers on rising edge of clk
        light_state <= next_state; //Next state gets clocked in on clk
    end

endmodule
```

Figure 6: Dataflow Model of Part A

3. Create a testbench and simulate the design. Your testbench should look like Figure 4b.

4. Add a constraints file, and edit it to meet the specifications of the design file and the problem description. The call system has two inputs, `call` and `cancel`, and one output, `light`. The inputs correspond to `btnL` and `btnR` on the board respectively, and the output corresponds to `LED0` on the board. The uncommented portion of your constraints file should look like Figure 5.

5. Program the board and verify that the system works as specified.

6. Now generate the minimized equation for `next_state` from the state table.

7. Create a new design file and set it as top. Copy your code from Figure 4a, but make two changes: 1. make the `next_state` variable a `wire`, and 2. replace the first always block with the statement `assign next_state = 0;`, as shown in Figure 6.

8. Replace in the right-hand side of the `assign` statement with your minimized equation. Be sure to use bitwise operators and not logical operators (do a quick google search about these; they are similar to C). This is called **dataflow modeling** and is used to express combinational logic as functions instead of gates, thereby greatly simplifying combinational logic design.

```
module clkdiv(
    input clk,
    input reset,
    // Remove reg if you use the assign statement
    output reg clk_out
    );

    // Initialize count register
    // Be sure to change the width of the counter
    // between simulation and on-board implementation
    reg [15:0] count = 0;

    // Use this line if you want to use bitmasking to
    // create your divider. (See background section.)
    // assign clk_out = count[15];

    always @(posedge clk) begin
        if(reset) count = 0;
        else count = count+1;

        // Use this line if you want to use an if statement
        // to define your frequency.
        // Change '10' to match desired frequency
        // Remember that the count value only gives half the clock cycle.
        if(count == 10) clk_out = ~clk_out;
    end
endmodule
```

Figure 7: Clock divider module

3.2 Part B: Rising-Edge Detector

A rising-edge detector is a sequential circuit that outputs a '1' for one clock cycle when an input changes from a '0' to a '1'. The rising-edge detector can be implemented using two separate blocks: a combinational block and a sequential block. The combinational block should calculate the next state and the outputs, and the sequential block should store the calculated state and outputs.

Your rising-edge detector should have the following functionality: an output `outedge`, connected to `LED0`, should flash once for 100ms (i.e. turn on for 100ms, then turn off) when a rising edge is detected on input `signal`, connected to `SW0`. The rising-edge detector should also include a `reset` button, tied to `btnC`. When the `reset` button is pushed, the state machine should reset to its initial state on the clock edge. As with all state machines, the state machine requires the board clock `clk` as an input.

1. Draw a state graph for the rising-edge detector. You should have 3 states.

2. Create a module labeled `clkdiv.v`, as shown in Figure 7. This module will be your clock divider. Remember that a clock divider is necessary because we want to slow down the 100MHz board clock to a reasonable speed to where we can see the response of `LED0` when we tie a clock to our FSM.

3. Use behavioral modeling to model the rising-edge detector in Verilog. Your design file should look something like Figure 8. Note the instantiation of the clock divider module.

4. Create a testbench and simulate your design. Be sure to screenshot your waveform.
   Note: When simulating FSMs (or anything using a clock), it is important that you change the width of your clock divider to much smaller than the actual board. This is because the simulation simulates a clock signal using software, which is significantly slower than hardware. My recommendation is to have a 2-bit wide clock divider for simulation.

5. Program the board with your state machine, and ensure it works. Debug if it doesn't. Be sure that your clock divider meets the 100ms flash specification stated in the problem statement

```
module edge_detector(
    input clk,
    input signal,
    input reset,
    output reg outedge
    );

    wire slow_clk; // Divided clk
    reg [1:0] state; // Stores current state
    reg [1:0] next; // Stores next state

    // Instantiate clk divider module
    clkdiv c1(.clk(clk), .reset(reset), .clk_out(slow_clk));

    // Combinational logic
    always @(*) begin
        case(state)
            2'b00: begin
                outedge = 1'b0;
                if(~signal) next = 2'b00;
                else next = 2'b01;
            end
            2'b01: // Fill in your FSM code here
            2'b10: // Fill in your FSM code here
            default: // Set your default to your initial state
        endcase
    end

    // Sequential logic
    always @(posedge slow_clk) begin
        if(reset) state <= 2'b00; // Note that this is a 'synchronous' reset
        else state <= next; // Change state
    end
endmodule
```

Figure 8: Rising edge detector design file

above.

3.3 Part C: Seven-Segment Display Controller

For this part of the lab, you will design a seven-segment display controller, as described in the Background section. Be sure to review time-multiplexing before beginning this part.

Your seven-segment display controller should take in four four-input numbers, represented by `SW[15:12]`, `SW[11:8]`, `SW[7:4]`, `SW[3:0]`. The hex number represented by `SW[15:12]` should be displayed on `AN[3]`, `SW[11:8]` on `AN[2]`, and so forth. You should also have a `reset` button to re-initialize the controller on `btnC`. Make sure you enable the board clock in your constraints file.

1. Write a module named `hex2seg` that will convert a four-bit input into a seven-bit output seven-segment output using behavioral modeling. It should serve the same functionality as Part C of Lab 3. You may use the truth table you wrote in Lab 3 to assist you.

2. Now create a design file for your state machine/controller circuit. Be sure to add `initial` statements to initialize any registers you create. One example of an implementation based on a state machine is given in Figure 9. You are encouraged to come up with your own design. There are several ways of simplifying the provided design that may include using a dataflow model or a modified behavioral model.

3. Now create a design file for your main module. It should contain four instances of the `hex2seg` module (one for each digit), one instance of your clock divider, and one instance of your state machine. Be sure to connect each module and create intermediate `wires` to connect between modules when appropriate. An example main module is given in Figure 10.

```
`timescale 1ns / 1ps

module time_mux_state_machine(
    input clk,
    input reset,
    input [6:0] in0,
    input [6:0] in1,
    input [6:0] in2,
    input [6:0] in3,
    output reg [3:0] an,
    output reg [6:0] sseg
    );

    reg [1:0] state ;
    reg [1:0] next_state;

    always @ (*) begin
        case(state)          // State transition
            2'b00: next_state = 2'b01;
            2'b01: /*Your code here*/;
            2'b10: /*Your code here*/;
            2'b11: /*Your code here*/;
        endcase
     end

    always @(*) begin
        case (state)         // Multiplexer
            2'b00 : sseg = in0;
            2'b01 : /*Your code here*/;
            2'b10 : /*Your code here*/;
            2'b11 : /*Your code here*/;
        endcase

        case (state)         // Decoder
            2'b00 : an = 4'b1110;
            2'b01 : /*Your code here*/;
            2'b10 : /*Your code here*/;
            2'b11 : /*Your code here*/;
        endcase
    end

    always @(posedge clk or posedge reset) begin
        if(reset)
            state <= 2'b00;
        else
            state <= next_state;
    end
endmodule
```

Figure 9: Example state machine

```
`timescale 1ns / 1ps

module time_multiplexing_main(
    input clk,
    input reset,
    input [15:0] sw,
    output [3:0] an,
    output [6:0] sseg);

    wire [6:0]  in0, in1, in2, in3;
    wire slow_clk;

    // Module instantiation  of hexto7segment decoder
    hexto7segment c1 (.x(sw[3:0]), .r(in0));
    hexto7segment c2 (.x(sw[7:4]), .r(in1));
    hexto7segment c3 (.x(sw[11:8]), .r(in2));
    hexto7segment c4 (.x(sw[15:12]), .r(in3));

    // Module instantiation of the clock divider
    clk_div_disp c5 (.clk(clk), .reset(reset), .slow_clk(slow_clk));

    // Module instantiation of the multiplexer
    time_mux_state_machine c6(
      .clk (slow_clk),
      .reset (reset),
      .in0 (in0),
      .in1 (in1),
      .in2 (in2),
      .in3 (in3),
      .an (an),
      .sseg (sseg));
endmodule
```

Figure 10: Seven-seg controller main module

4. Create a testbench to simulate your controller circuit. Verify that it works in simulation. Be sure to edit your clock divider to accommodate for the simulation speed.

5. Edit your `clkdiv` module so that your seven-segment display controller FSM runs at exactly 5Hz on the board. When you calculate your counter values for your clock divider, be sure to show your work in the lab report.

6. Test your design on the board. The seven-segment display should appear to cycle through each set of switches, displaying each number individually.

7. Edit your `clkdiv` module so that your seven-segment display controller runs from 3-5kHz on the board. When you calculate your counter/bitmask values for your clock divider, be sure to show your work in the lab report.

8. Test your design on the board. The seven-segment display should appear as if all four numbers are on at the same time. If it doesn't, adjust the frequency.

9. Once you complete this section, draw a high-level block diagram of the system. Be sure to label all input and output wires/connections. Remember that a block diagram does not show any circuitry or internal functions, only inputs and outputs of black-boxes and how they connect together in the system.

4 Submission

There are two items to submit:

1. A .zip file containing your design/testbench files (.v), constraints file (.xdc), and bitstream (.bit) files for each component of the lab. If you need assistance locating these files in your directory, please visit a TA in office hours.

2. A PDF with the items listed on the Cover Sheet in Canvas.

Submit these two files as one single submission on Canvas. Be sure to schedule a checkout slot via the Canvas appointment scheduler. Be sure to save the bitstream for each part in a convenient location in your machine so you can easily program theboard during checkout without having to regenerate the bitstream.

5 Sample Checkout Questions

Sample checkout questions may include, but are not limited to (in no particular order):

1. What is one advantage of dataflow modeling over behavioral modeling?

2. What is clock division, and why do we need it?

3. How would you design a divide-by-5 clock?

4. What is a sensitivity list? Why is it important?

5. Define blocking and non-blocking. When is each used?

6. Why is it impossible to get a clock of exactly 10Hz by using bit-masking in Verilog?
7. Define synchronous and asynchronous. Is an FSM synchronous or asynchronous?
8. Why do we need to change the clock divider between the board and simulation?
9. How do we get around the "problem" of not being able to display four different numbers on the seven-seg at the same time?
10. Show me your calculations to achieve precisely 5Hz frequency.
11. How fast does the board clock run? What is its maximum speed (you may have to look this up)?
12. How would you change your code to create a falling edge detector?
13. How would you change your state machine to create an edge detector that senses both rising and falling edges?
14. What is dataflow modelling, and how is it different from structural and behavioral modelling?
15. What do `posedge` and `negedge` refer to?
16. How did you generate the 100ms delay on `LED0` for the rising-edge detector?

Note that all of the answers to these questions are given somewhere in the lab document or in previous labs/lectures on Verilog; you do not need to do any external reading/searching for knowledge in the internet unless the lab document says so. Any material/content from previous labs is also fair game to be asked during checkout.