CSC258- Lab 2

The *case* statement, Adders and ALUs

Learning Objectives

In this lab you will design (a) multiplexers using the case statement, (b) a simple ripple-carry adder, and (c) an Arithmetic Logic Unit (ALU), a fundamental component of each processor. You will also gain more practice with hierarchical design in Verilog and with using binary and hexadecimal numbers.

Marking Scheme

This lab is worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows:

• Prelab + Simulations: 3 marks

• Part I (in-lab): 1 mark

• Part II (in-lab): 1 marks

• Part III (in-lab): 3 marks

Preparation Before the Lab

Carefully review the background section below, as well as the "Preparation Before the Lab" instructions in the Lab 2 handout.

You are required to complete Parts I to III of the lab by writing and testing your Verilog code. Include your schematics, Verilog code, and simulation outputs (printed screenshots) for Parts I to III in the prelab. You must simulate your circuit with ModelSim using reasonable test vectors written in the format described in Lab 2 and Lab 3 handouts. The term test vector simply refers to one combination of inputs that you will use to test your design. Each simulation should consist of multiple test vectors, sufficient to demonstrate that your design functions as intended.

In-Lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate all parts to the teaching assistants. As a reminder, the device name is *5CSEMA5F31C6* from the Cyclone V family.

Verilog Background

Representing Constants

Verilog has a specific notation for representing constants (i.e., literal values). The following code snippet creates a 1-bit wire named a, which is connected to ground (logic-0).

```
wire a;
assign a = 1'b0;
```

The number before the single quote is a decimal number representing the bit width (i.e., the number of binary digits, also known as bits). It is 1 for a single wire, and higher for a bus (i.e., a collection of wires). The letter after the single quote is called the radix. Its possible values are letters b, d, h, or o to specify whether the value that follows is in binary (b), decimal (d), hexadecimal (h), or octal (o) notation. (Octal representation is rarely used.) Lastly, the value after the radix is the number in that numerical representation.

For example, 2'b10 corresponds to decimal number two, while 8'h1a (or 8'h1A) corresponds to decimal number 26. The latter is written in binary as 8'b0001_1010 and in decimal as 8'd26. Verilog allows you to use underscores to separate groups of bits to improve readability. Here we separated groups of 4-bits as they each correspond to a single hexadecimal digit. Note that underscore characters are optional.

Verilog Operators

This section presents various Verilog operators that you might find useful for this lab.

- Arithmetic Operators: + (addition), - (subtraction), * (multiplication), / (division), % (remainder), ** (exponentiation).
- Reduction Operators: These are unary operators that reduce a vector to a single bit value. The operators are: & (AND all bits), | (OR all bits), and ^ (XOR all bits). You can prepend to any of these to get reduction NAND, NOR and XNOR operators respectively. An example of a reduction AND operation is & 3'b010 which is equivalent to an AND operation between all three bits of the value 3'b010, and will produce 1'b0. Similarly, & 3'b111 will produce 1'b1.
- Concatenation: Verilog uses curly braces for concatenation. For example, {2'b01, 1'b1} will produce 3'b011. Concatenation can be used in both the left hand side, and the right hand side of an assignment statement.
- Replication: {n{m}} will replicate n-times the value m. For example, {3{2'b01}} is equivalent to 6'b010101.

Advanced ModelSim Commands

The *wave.do* file in Lab 2 contained simple ModelSim commands that forced a signal to logic 0 or logic 1 (e.g., force {SW[0]} 0), followed by run commands that ran the simulation for a predetermined number of nanoseconds before applying a different test vector. However this approach does not scale well as your designs become more complex and have more inputs. ModelSim allows you to apply a periodic signal to your simulation's inputs which makes creation of test vectors much easier.

The format of a more advanced force command is the following:

```
force {<signal name>} <initial value><initial time>,<new value> <new time>
                       -repeat <repeat time> -cancel <cancel time>
```

This will set the `signal name` to `initial value` at initial time after the current time and will set it to `new value` after `new time` passed from the current time. This square wave will repeat repeat time after the current time and you can choose to cancel it at `cancel time`. In all cases, you can explicitly specify the time unit in your force command (*e.g.*, by writing *ns*) after any time value. You can use shorthands -r and -c instead of -repeat and -cancel.

Additionally, you may also force a full *bus* of signals. For example, you can set a 4-bit wide bus to decimal 10, by setting its value to `2#1010`. The 2 means the constant value is in binary (i.e. base 2) while the 1010 is the binary constant itself. You could also specify the value in decimal (`10#10`, or base 10) or hexadecimal (`16#A`, or base 16); just ensure that your signal is wide enough to accommodate the size of your value.

Here is an example of test vectors for two 1-bit inputs `a` and `b`. Notice that the square wave applied to `b` has twice the period of the square wave applied to `a`, thus modeling all four possible input combinations, the same way they'd be present in a truth table.

```
force {a} 0 0, 1 20 -repeat 40
force {b} 0 0 ns, 1 40 ns -r 80
```

Also do not forget to use `-timescale 1ns/1ns` as an argument to the **vlog** command, as we did in the wave.do file provided in Lab 2. The *-timescale* switch tells ModelSim what the default time unit is when no unit is specified (first number), and what the smallest unit of time to simulate should be (second number).

Part I

For this part of the lab, you will learn how to use *always* blocks and *case* statements to design a 7-to-1 multiplexer.

An *always* block allows us to describe a circuit using *behavioral* style, using *case* and *if* statements. However, the circuit produced from this description still consists of basic logic gates.

Any identifier that appears on the left hand side of the equal (=) sign inside an always block must have been declared as a **reg** type in the module containing the *always* block. Please note that you cannot use a wire for this purpose. The reason is that in you may not specify what the value of the signal may be in some cases, which means that it should remain the same in those cases, so we should define it as a reg. We will discuss the **reg** type in more details later in the semester.

The model Verilog code for a 7-to-1 multiplexer built using a *case* statement is shown below. The seven inputs are from the signals named `Input[6:0]`. The output is called `Out`, and it is declared as a **reg** type, as explained above. The select lines are called `MuxSelect[2:0]`.

```
reg Out;     // declare the output signal for the always block
always @(*)  // declare always block
begin
    case (MuxSelect [2:0])        // start case statement
       3'b000: Out = Input[0];    // case 0
       3'b001: Out = Input[1];    // case 1
       3'b010: Out = Input[2];    // case 2
       3'b011: ...                // case 3
       3'b100: ...                // case 4
       3'b101: ...                // case 5
       3'b110: ...                // case 6
       default : . . .            // default case
    endcase
end
```

The always block uses an asterisk in the sensitivity list to indicate that the block describes *combinational* logic, i.e., any logic where the outputs rely strictly on the inputs. We will learn more about *combinational* and *sequential* logic later in the course. For now, use the asterisk in the *always* block of a *case* statement as shown above.

It is important to have a *default* case to ensure that all input cases are covered. Otherwise, the synthesized circuit may contain a loop, where some outputs of a logic function are connected back to its inputs. This produces undesirable effects in hardware that are difficult to analyze and control. You will later learn how to safely design circuits with memory. For now, remember that for all combinational circuits, such as multiplexers, encoders, decoders, . . . you should enumerate all possible input cases, and having a default statement helps you achieve this purpose.

In general, keep in mind that Verilog is synthesized into logic gates, and synthesis tools such as *Quartus* Prime determine what logic gates to generate by analyzing your overall specification. This is **NOT** a code that will be run one line at a time. The rules for conversion of Verilog code into logic gates assume certain coding style, and if this coding style is not followed, the logic gates that end up produced by the tools can be quite unexpected. To make matters worse, simulation may not match the actual behavior of logic gates, since the simulator does not always accurately model behavior of circuits produced by non-compliant Verilog descriptions. To avoid these issues, you will be shown the coding styles that avoid most problems. By following these coding styles you will ensure that tools produce results that are expected and easy to test and debug.

Using SW_{6-0} as the data inputs and SW_{9-7} as the select signals, display on $LEDR_0$ the output of a 7-to-1 multiplexer using the case statement style as shown above.

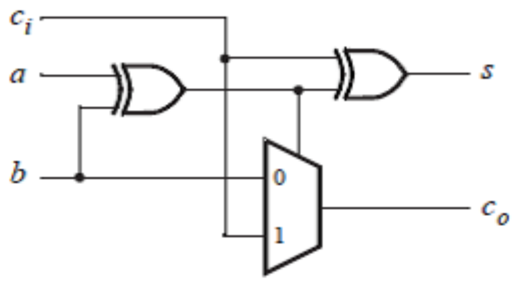
1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Be prepared to explain it to the TA as part of your preparation. **(PRELAB)**
2. Write Verilog code for a 7-to-1 multiplexer, based on the template provided above. Use switches SW_{9-7} on the DE1-SoC board as the **MuxSelect** inputs and switches SW_{6-0} as

the Input data inputs. Connect the output to $LEDR_0$. **(PRELAB)**

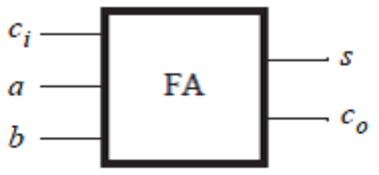
3. Simulate your circuit with ModelSim for different values of **MuxSelect** and **Input**. You must include a screenshot of the simulation output as part of your prelab. **(PRELAB)**
4. Create a new Quartus Prime project for your circuit. Make sure it is stored in your W:\ drive.
5. Compile the project.
6. In Quartus Prime, select Tools > Netlist Viewers > RTL Viewer and observe the circuit that got produced from your Verilog code. Show this to the TA when you demonstrate the functionality of your circuit to the TA. Note that Quartus uses slightly different symbol for the multiplexer than what you have seen in class: select inputs are denoted by name instead of being drawn from the top of the symbol. **(IN-LAB)**
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs. When you are sure that it works correctly, demonstrate the circuit behaviour to your TA. **(IN-LAB)**

Part II

Figure 1(a) shows a circuit for a full adder, which has the inputs a, b, and c_i, and produces the outputs s and co. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum cos = a + b + c_i. Please note that the + operator here means addition and not logic OR. Figure 1(d) shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is called a ripple-carry adder, because of the way that the carry signals are passed from one full adder to the next. Write Verilog code that implements this circuit, as described below. Be sure to use what you learned about hierarchy in Lab 2.



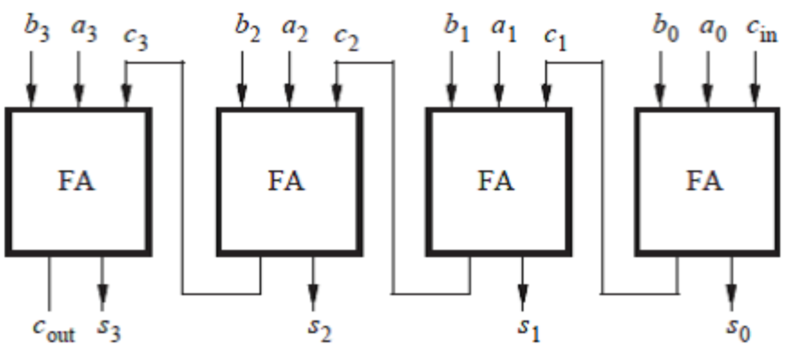
a) Full adder circuit



b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Figure 1: A ripple-carry adder circuit

Perform the following steps:

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Your schematic should resemble Figure 1(d), though it should also contain module and signal labels, and shows external connections to the switches and LEDs. Be prepared to explain it to the TA as part of your preparation. **(PRELAB)**
2. Write a Verilog module for the full adder subcircuit and write another Verilog module that instantiates four instances of this full adder. Name the input ports *A*, *B* and *cin*, and the output ports *S* and *cout*. Note: You should **NOT** use the arithmetic addition operator + in your Verilog implementation of the full-adder. Doing so will earn you 0 marks for this part. **(PRELAB)**
3. Simulate your 4-bit ripple-carry adder with ModelSim for intelligently chosen values of *A* and *B* and *cin*. You should include a screenshot of it in the prelab. Note that as circuits get more complicated, you will not be able to simulate or test all possible cases. This means that you can test only a subset. Here *intelligently chosen* means to find particular corner cases that exercise key aspects of the circuit. An example would be a pattern that shows that the carry signals are working. Be prepared to explain why your test cases are good enough. **(PRELAB)**
4. Create a new Quartus Prime project for the adder circuit. Make sure it is stored in your W:\ drive. In the new top-level module, use switches SW_{7-4} and SW_{3-0} to connect to inputs *A* and *B* respectively. Use SW_8 for the carry-in, c_{in} of the adder. Connect the outputs of the adder, c_{out} and *S*, to the LEDs $LEDR_4$ and $LEDR_{3:0}$ respectively.
5. Compile the project.
6. In Quartus Prime, select Tools > Netlist Viewers > RTL Viewer and observe the circuit that got produced from your Verilog code. Try expanding modules in the diagram, and selecting different hierarchy levels in the Netlist Navigator on the left.
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs. Demonstrate the circuit behaviour to

your TA when you finished testing. **(IN-LAB)**

Part III

Using Part II from this lab and the HEX decoder from Lab 2 Part III, you will implement a simple Arithmetic Logic Unit (ALU). This ALU has two data inputs and can perform multiple operations on the data inputs such as addition, subtraction, logical operations, etc. The output of the ALU is selected by function inputs that specify the function to be performed by the ALU. The easiest way to build an ALU is to implement all required functions and connect the outputs of the functions to a multiplexer. Choose the output value for the ALU using the ALU *function* inputs to drive the multiplexer select lines. The output of the ALU will be displayed on the LEDs and HEX displays.

The following case statement pseudo code shows the operations that you should implement in the ALU, given the specified function value. The ALU has two 4-bit inputs, *A* and *B* and an 8-bit output, called $ALU_{out}[7:0]$. Note that in some cases, the output will not require the full 8 bits so do something reasonable with the extra bits, such as making them 0 so that the value is still correct. Adding zeros in front of any positive number is called *sign-extension*, and does not change the value of the number.

Note that it is **not** permissible to enclose a module instantiation inside an always block. An *always* block is only used to describe one part of the circuit using behavioral style (e.g., using a *case* statement). If you want to connect inputs or outputs of an instance of a module to the circuit described using an always block, you should create a module instance outside the *always* block and use *wires* to connect the two circuits.

```
always @(*)             // declare always block
begin
    case (function)
        0: ...    // Make the output equal to A+1, using the adder
                  // circuit from Part II of this Lab.
        1: ...    // A + B using the adder from Part II of this lab
        2: ...    // A + B using the Verilog '+' operator
        3: ...    // A XOR B in the lower four bits and A OR B in
                  // the upper four bits
        4: ...    // Output 1 (8'b00000001) if any of the 8 bits in
                  // either A or B are high, and 0 (8'b00000000)
                  // if all the bits are low (use a reduction OR
                  // operator)
        5: ...    // Make the input appear at the output , with A in
                  // the most significant (left-most) four bits and
                  // B in the least significant (right-most) bits.
        default: ...  // default case, output 0
    endcase
end
```

Once you've created this ALU module, the inputs and outputs of this ALU will need to be connected to the hardware on the DE1 board. Use the following for the ALU inputs and outputs:

- The A and B inputs connect to switches SW_{7-4} and SW_{3-0} respectively.
- Use KEY_{2-0} for the function inputs.
- Display $ALU_{out}[7:0]$ in binary on $LEDR_{7-0}$
- Have *HEX0* and *HEX2* display the values of B and A respectively in hexadecimal and set *HEX1* and *HEX3* to 0.
- *HEX4* and *HEX5* should display $ALU_{out}[3:0]$ and $ALU_{out}[7:4]$ respectively in hexadecimal.

Perform the following steps to complete the lab:

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Your schematic should contain a block diagram of your design showing any design hierarchy. You should show the multiplexer that is implied by the case statement for your ALU as well as all inputs to this multiplexer. Also show all connections to switches and LEDs. Be prepared to explain it to the TA. **(PRELAB)**
2. Write a Verilog module for the ALU including all inputs and outputs. **(PRELAB)**
3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must include screenshots of output waveforms as part of your prelab. **(PRELAB)**
4. Create a new Quartus Prime project for your circuit, and compile the project.
5. In Quartus Prime, select Tools > Netlist Viewers > RTL Viewer and observe the circuit that got produced from your Verilog code. Try expanding modules in the diagram, and selecting different hierarchy levels in the Netlist Navigator on the left.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit. Demonstrate the circuit behaviour to your TA after testing it. **(IN-LAB)**

Note: In your simulation, KEY_{3-0} are inverted. Remember that the DE1-SoC board recognizes an unpressed pushbutton as a value of 1 and a pressed pushbutton as a 0.