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). You will also gain more practice with hierarchical design in Verilog and with using binary and hexadecimal numbers.

Marking Scheme

Each lab is worth 4% of your final grade, but you will be graded out of 6 marks for this lab, as follows.

- Prelab: 2 marks
- Part I (in-lab): 1 mark
- Part II (in-lab): 1 mark
- Part III (in-lab): 2 marks

Preparation Before the Lab

Review the instructions in Lab 1 about preparations.

You are required to complete Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus II. Show your schematics, Verilog, and simulations for Parts I to III to the teaching assistants. You must simulate your circuit with ModelSim using reasonable test vectors written in the format used in Lab 1 for the simulation files.

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.

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., number of binary digits, bits). This is one for a single wire, but is higher for a bus (i.e., a collection of wires). The letter after the single quote is called the *radix*. Its possible values are `b`, `d` or `h` to specify whether the value that follows is in *binary*, *decimal* or *hexadecimal* notation. (Octal is another valid representation albeit rarely used.) Lastly, the value after the radix is the number in that numerical representation.

For example, `2'b10` corresponds to number two, while `8'h1a` (or `8'h1A`) to 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.

More Verilog Operators

This section presents various Verilog operators that you might find useful for this lab. You are also encouraged to refer to the *Verilog Supplements* section on Portal under Course Materials.

- 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), | (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 XOR operation is `^ 3'b010` which will produce `1'b1`.
- Concatenation: Verilog uses curly braces for concatenation. For example, `{2'b01, 1'b1}` will produce `3'b011`. Concatenation can also be used in the left 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`.

More Advanced ModelSim commands

The wave.do file in Lab 1 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 this much simpler.

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` time `new time` after 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 quantum 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 while the `1010` is the binary constant itself. You could also specify the value in decimal (`10#10`) or hexadecimal (16#A), just ensure that your signal is wide enough to accomodate 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
```

Additionally, in the mux.v file provided in Lab 1, the first line reads:

```
`timescale 1ns / 1ns // `timescale time_unit/time_precision
```

This line tell modelsim what the units of time should be for the simulation (`time_unit`) and what the smallest unit of time to simulate should be (`time_precision`). Be sure to include this line for every verilog file you simulate from now on.

Part I

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

Like a module, an *always* block can have inputs and outputs. A module can contain any number of *always* blocks just as any module can contain any number of other module instantiations. The difference is that an *always* block can only instantiate logic within the module where it is defined. A module can be instantiated in any other module, i.e., it can be reused.

Any output of 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 that purpose.

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`. 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
```

An *always* block is triggered to execute in simulation whenever there is a change in the sensitivity list. This list is denoted by the asterisk character in the above example. This means that whenever *any* input to the *always* block is changed, the code in the *always* block will be simulated. We can change the asterisk to certain inputs to limit when this code is triggered, but this can lead to simulations that do not match the real hardware. One of the (bad) features of the language. The accepted practice today is to always use the asterisk in your *always* block for *combinational* logic, i.e., any logic where the outputs rely strictly on the inputs. You will learn more about *combinational* and *sequential* logic later. For now, use the asterisk in the *always* block for a *case* statement as shown above.

It is important to have a *default* case to ensure that all cases are covered. Otherwise, you can again have simulations that do not match the hardware. Yet another Verilog feature! Your goal is to write Verilog that will generate hardware that exactly matches the simulation, so please put in the *default* statement.

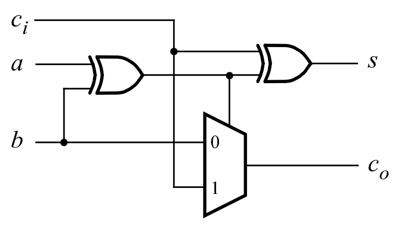
If you want to know why, read on. When you execute an *always* block, the use of *if* and *case* statements can take you through different code paths. If you reach the end of the *always* block and there is an unassigned (reg) variable, then a memory element, a latch, will be created because the meaning is that the variable keeps its previous value, so a memory element is inferred. The problem becomes more subtle because if *MuxSelect* in the above example is three bits, there are actually more than eight cases! Each bit can be (1, 0, x (unknown value), z (high-impedance)), so there are really $4^3 = 64$ possible paths. Synthesis tools will likely assume only (1,0) and create the correct circuit, but the simulator may not do the same. Always, always put in the *default* statement. If you did not understand the above, at least you, hopefully, now see how Verilog can have subtle side effects that can cause problems. To avoid these issues, you will be shown the coding styles that will avoid most problems.

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.

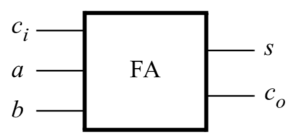
1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Be prepared to explain it to the TA as part of your preparation. **(PRELAB)**
2. Create a new Quartus II project for your circuit. **(PRELAB)**
3. Include your Verilog file for the circuit in your project. Use 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)**
4. Simulate your circuit with ModelSim for different values of `MuxSelect` and `Input`. You must show these to the TA as part of your preparation. **(PRELAB)**
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part II

Figure $2a$ shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and c_o. 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 $c_o s = a + b + c_i$. Please note that the + operator here means addition and not logic-OR. Figure $2d$ 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 1.



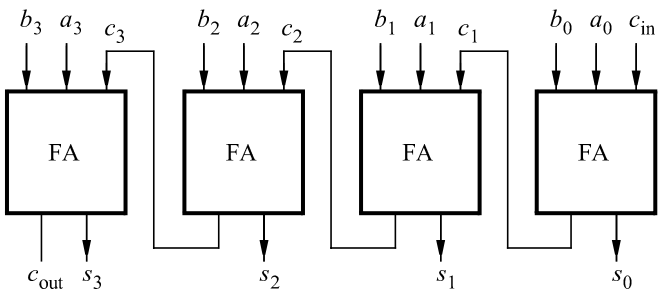
a) Full adder circuit



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 2. 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 Fig. 2d, though it should also contain module and signal labels, also showing external connections to the switches and LEDs. Be prepared to explain it to the TA as part of your preparation. **(PRELAB)**

2. Create a new Quartus II project for the adder circuit. Write a Verilog module for the full adder subcircuit and write a top-level Verilog module that instantiates four instances of this full adder. Note: You should **NOT** use the arithmetic adddition operator + in your Verilog implementation of the full-adder. Doing so will earn you 0 marks for this part. **(PRELAB)**

3. Use switches SW_{7-4} and SW_{3-0} to represent the 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*$_9$ and *LEDR*$_{3:0}$ respectively. **(PRELAB)**

4. Simulate your adder with ModelSim for intelligently chosen values of A and B and c_{in}. You must show these to the TA as part of your preparation. 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)**

5. Compile the project.

6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part III

Using Parts I and II from this lab and the HEX decoder from Lab 1 Part III, you will implement a simple Arithmetic Logic Unit (ALU). An ALU has two inputs and can perform multiple operations on the inputs such as addition, subtraction, logical operations, etc. The output of the ALU is selected by *function* bits 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.

Shown in the case statement below are the operations to be implemented in the ALU for each *function* value in pseudo code format. The ALU has two 4-bit inputs, A and B and an 8-bit output, called *ALUout[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.

```
always @(*)          // declare always block
begin
    case (function)  // start case statement
        0: A + B using the adder from Part II of this Lab
        1: A + B using the Verilog '+' operator
        2: A XOR B in the lower four bits and A OR B in the upper four bits
        3: Output 1 (8'b00000001) if at least 1 of the 8 bits in the two inputs is 1 using a single OR operation
        4: Output 1 (8'b00000001) if all of the 8 bits in the two inputs are 1 using a single AND operation
        5: Make the inputs appear at the output, with A in the most significant four bits and B in the least significant four bits.
        default: ...     // default case, output 0
    endcase
end
```

Note that in this part of the lab, you will need to learn about Verilog concatenation for the additions, sign extension, and the Verilog reduction operations for ORing and ANDing multiple bits without typing out the operation for each bit individually.

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 *ALUout[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 *ALUout[3:0]* and *ALUout[7:4]* respectively in hexadecimal.

Perform the following steps:

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 as part of your preparation. **(PRELAB)**
2. Write a Verilog module for the ALU including all inputs and outputs. **(PRELAB)**
3. Create a new Quartus II project for your circuit. **(PRELAB)**
4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your preparation. **(PRELAB)**
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

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.