Laboratory Exercise 3

The *case* statement, Adders and ALUs

September 21, 2017

This is an exercise in designing multiplexers using the *case* statement, using hierarchy in Verilog, developing a simple adder and an ALU, where you can learn about the many Verilog operators.

Preparation Before the Lab

Review the instructions in Lab 2 about preparations.

Prepare for Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus. 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 2 for the simulation files. If you have not found it already, you should look at the *UsefulModelsimCommands* handout under *General Resources* on Piazza.

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.

Part I

For this part of the lab, you will be learning how to use *always* blocks (text Section 2.10.2, A.11.1) and *case* statements (text Section 4.6.3, A.11.4) 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 the same 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., a module can be reused.

Any output of an *always* block must have been declared as a **reg** type in the module containing the *always* block.

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. This is 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 the *default* statement is important, read on, else skip this paragraph. 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 clearly labeled. 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 $LEDR_0$.

   Be prepared to explain your schematic to the TA as part of your preparation.

2. After drawing your schematic, write the Verilog code that corresponds to your schematic. Your Verilog code should use the same names for the wires and instances shown in the schematic.

3. Create a new Quartus project for your circuit.

4. Include your Verilog file for the circuit in your project.

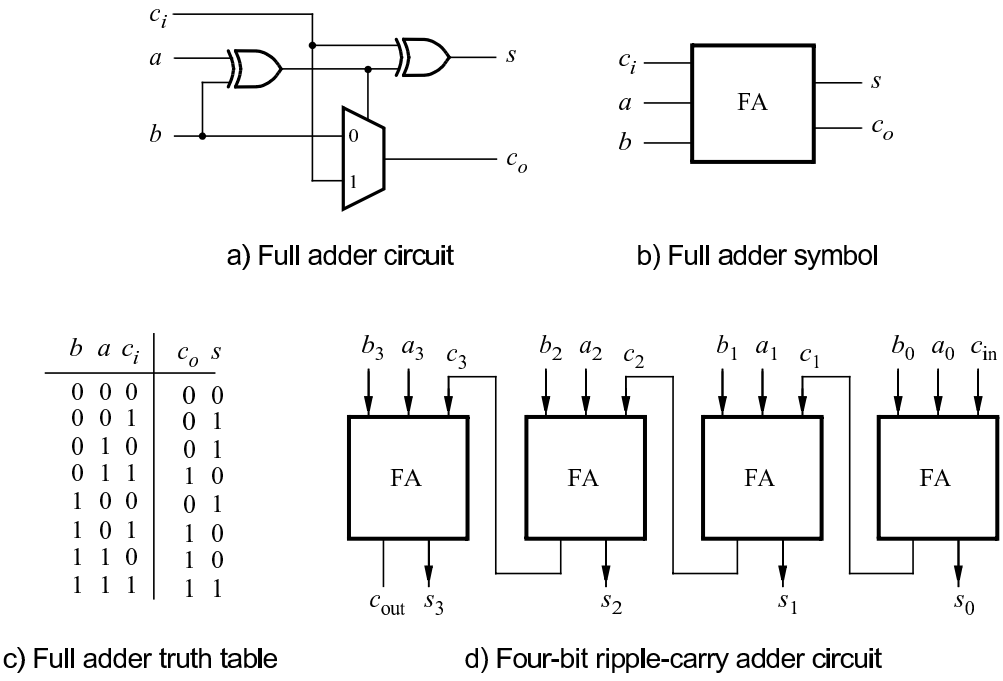
5. Simulate your circuit with ModelSim for different values of `MuxSelect` and `Input`. You must show these to the TA as part of your preparation.

6. Compile the project.

7. 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* (text Section 3.2), 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 $c_o s = a + b + c_i$. 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 2.



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

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. 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.

   Be prepared to explain your schematic to the TA as part of your preparation.

2. Create a new Quartus 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.

3. 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.

4. Compile the project.

5. 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 2 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 pseudo-code, i.e., not exact syntax, case statement below are the operations to be implemented in the ALU for each *function* value. 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 8'b00011000 if at least 1 of the 8 bits in the two inputs is 1 using a single OR operation
    4: Output 8'b11100111 if all of the 8 bits in the two inputs are 1 using a single AND operation
    5: Display the A switches in the most significant four bits and the complement of the B switches in the least significant four bits without complementing the bits individually
    default: . . . // default case
  endcase
end

Note that in this part of the lab, you will need to learn about number representations in Verilog (text Section A.5) and various other Verilog operators (text Section 4.6.5). You will need to learn about several operators including Verilog concatenation for the additions to stick in the extra bits you need and to create other 8-bit values, and 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 and set *HEX1* and *HEX3* to 0. *HEX4* and *HEX5* should display *ALUout[3:0]* and *ALUout[7:4]* respectively.

Perform the following steps:

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.

2. Write a Verilog module for the ALU including all inputs and outputs.

3. Create a new Quartus project for your circuit.

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.

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.