Laboratory Exercise 3

The *case* statement, Adders and ALUs

Revision of October 9, 2022

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

1 Work Flow

For each part of the lab you should begin by writing and testing Verilog code and compiling it with Quartus. You should be prepared to show schematics, Verilog, and simulations to your TA, if requested. 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 *Useful ModelSim Commands* handout.

2 Part I

For this part of the lab, you will be learning how to use *always* blocks (textbook Section 2.10.2, A.11.1) and *case* statements (textbook 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. In contrast, a module can be instantiated in any other module, i.e., a module can be reused.

A key requirement of using an *always* block is that 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 in Figure 1. The seven inputs are from the signals named `Input[6:0]`. The output is called `Out`. The select lines are called `MuxSelect[2:0]`.

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 Figure 1. This means that

```
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
```

Figure 1: Model Verilog code for a 7-to-1 multiplexer using a case statement.

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, always 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, or else you may 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. You will learn more about latches in module 12. But for now, just know that they are bad! The problem becomes even 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 patterns that will avoid most problems.

2.1 What to Do

The module you are writing for Part I should have the following signature declaration:

```
module part1(MuxSelect, Input, Out);
```

To build your module, perform the following steps:

1. Draw a schematic showing your code structure with all wires, inputs and outputs clearly labeled.

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. You can use the code fragment from Figure 1 as a starting point.

3. Simulate your circuit with ModelSim for different values of `MuxSelect` and `Input`. How many different patterns of inputs do you need to simulate to have confidence that your circuit is working? When you are satisfied with your simulations, you can submit to the Automarker. Keep in mind that this part does not need Quartus; there is no need to compile your code in Quartus each time you make a change. You can just re-run the wave.do file to redo simulation.

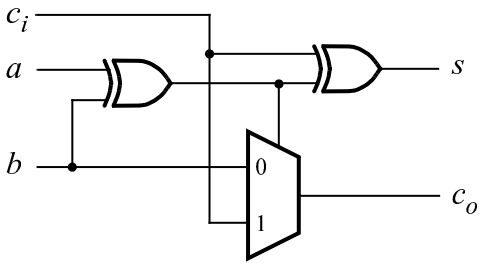
4. Open Quartus and create a new project for your circuit. You will need a top-level module to make connections from the instantiation of your `part1` module to the switches and LEDs of the DE1-SoC board. The connections used are shown in Table 1.

`part1` Port Name	Direction	DE1-SoC Pin Name
`MuxSelect`	Input	SW[9:7]
`Input[6:0]`	Input	SW[6:0]
`Out`	Output	LEDR[0]

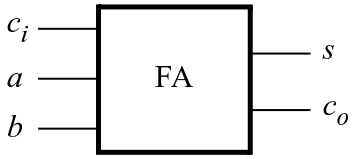
Table 1: Module `part1` mapping to DE1-SoC pin names

   You may want to review the structure of the `mux.v` module from Lab 2 to see how to do this.

5. Compile the project to generate a bitstream to make sure your code can be synthesized. It is possible, and not uncommon, to write Verilog that simulates properly, but it cannot be synthesized to working hardware. If you can successfully create a bitstream, your code can create some kind of hardware. Whether it works is another question!



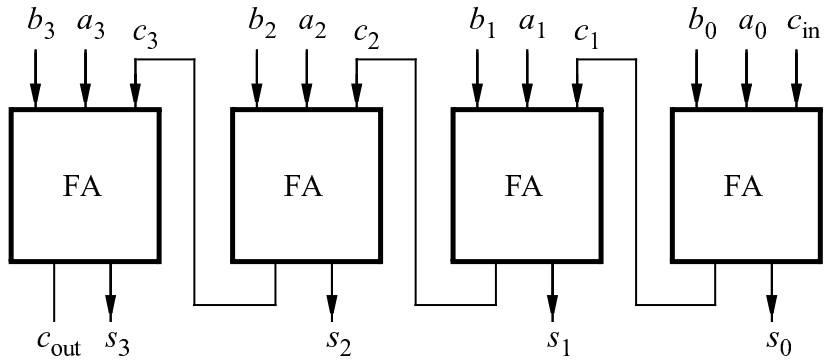
a) Full adder circuit



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.

6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing $LEDR_0$.

3 Part II

Figure 2(a) shows a circuit for a *full adder* (textbook Section 3.2), which has the inputs a, b, and c_i, and produces the outputs s and c_o. Note that Figure 2(a) shows one of many ways to implement a full adder circuit. The most important consideration is that the behaviour is correct. How you implement the behaviour is left to the designer. 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 2(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 the circuit of Figure 2(d) following the steps below. Be sure to use what you learned about hierarchy in Lab 2.

3.1 What to Do

The module for your four-bit ripple-carry adder design should have the following signature declaration:

```
module part2(a, b, c_in, s, c_out);
```

Note that `a` and `b` are the 4-bit inputs, `s` is the 4-bit sum output and `c_out` in this signature is the 4-bit vector of the four carry outputs from the four full adders in the form (c_{out},c_3,c_2,c_1) using the labels in Figure 2.

To build your `part2` module, perform the following steps:

1. Draw a schematic that will reflect your code structure with all wires, inputs and outputs labeled. It should look much like Figure 2.

2. After drawing your schematic, write the Verilog code that corresponds to your schematic. You should make use of there being four instances of the same full adder block. First, write a Verilog module for the full adder sub-circuit and then write the `part2` module that will instantiate four instances of your full adder module. Your Verilog code should use the same names for the wires and instances as shown in your schematic.

3. Simulate your adder with ModelSim for intelligently chosen values of the inputs a, b and c_{in}. Note that as circuits get more complicated, you will not be able to simulate or test all possible cases. How many input combinations would you need to simulate all possible input combinations for this four-bit adder? What about a 32-bit adder? When it is not possible to simulate all input combinations then 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. When you are satisfied with your simulations, you can submit to the Automarker.

4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part2` module to the switches and LEDs of the DE1-SoC board. The connections used are shown in Table 2. Note that c_{in} is the carry-in of your adder and c_{out} is the carry-out of your adder.

5. Compile the project to generate a bitstream to make sure your code can be synthesized.

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

part2 Port Name	Direction	DE1-SoC Pin Name
a	Input	SW[7:4]
b	Input	SW[3:0]
c_{in}	Input	SW[8]
c_{out}	Output	LEDR[9]
s	Output	LEDR[3:0]

Table 2: Module part2 mapping to DE1-SoC pin names

4 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: Sign extension of B to 8 bits // textbook page 167
    3: Output 8'b00000001 if at least 1 of the 8 bits in the two inputs is 1 using a single OR operation
    4: Output 8'b00000001 if all of the 8 bits in the two inputs are 1 using a single AND operation
    5: Display A in the most significant four bits and B in the lower four bits
    default: ... // default case
  endcase
end

4.1 Things to Watch For

Note that in this part of the lab, you will need to learn about number representations in Verilog (textbook Section A.5) and various other Verilog operators (textbook 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 *case* statement must always be inside an *always* block. If you look at the operations for the ALU, four of the cases can be implemented with logic expressions, which can be described within an *always* block, so they can be expressed within the relevant cases in the *case* statement. However, for *Case 0* you have to use the adder module you built for Part II. **You cannot instantiate a module within an *always* block, so how do you make this work?** If you come up with the correct schematic for your circuit, that will tell you what to do. Remember to label all the internal wires of your circuit to help you figure this part out.

4.2 What to Do

The module you are writing for Part III should have the following signature declaration:

```
module part3(A, B, Function, ALUout);
```

To build your module, perform the following steps:

1. Draw a schematic that will reflect your code structure with all wires, inputs and outputs labeled.
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 as shown in your schematic.
3. Simulate your ALU with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working. When you are satisfied with your simulations, you can submit to the Automarker.
4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part3` module to the switches, KEYs, LEDs and HEX displays of the DE1-SoC board. The connections used are shown in Table 3.

part3 Port Name	Direction	DE1-SoC Pin Name
A	Input	SW[7:4]
B	Input	SW[3:0]
Function	Input	KEY[2:0]
ALUout[7:0]	Output	LEDR[7:0]
B	Output	HEX0
A	Output	HEX2
ALUout[3:0]	Output	HEX4
ALUout[7:4]	Output	HEX5

Table 3: Module part3 mapping to DE1-SoC pin names

In addition display the digit 0 on *HEX1* and *HEX3*.

5. Compile the project to generate a bitstream to make sure your code can be synthesized.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and keys and observing the outputs.

Note: Be aware that 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. Your circuit will need to account for this.

5 Submission

When submitting to the Automarker make sure you have modules declared as shown below as the Automarker will be looking for modules with these exact signatures.

5.1 Part I

For Part I, you need to submit a file named `part1.v` with the following module in it:

1. `module part1(MuxSelect, Input, Out);`

5.2 Part II

For Part II, you need to submit a file named `part2.v` with the following module in it:

1. `module part2(a, b, c_in, s, c_out);`

5.3 Part III

For Part III, you need to submit a file named `part3.v` with the following module in it:

1. `module part3(A, B, Function, ALUout);`