Laboratory Exercise 3

The *case* statement, Adders and ALUs

Revision of October 3, 2022

In this lab, you will learn about designing multiplexers using System Verilog's *case* statement. You will also learn about using hierarchy by developing a simple adder and an arithmetic and logic unit (ALU). First, you will learn how to use the `always_comb` block in System Verilog.

1 Always blocks and Case statements

In this part of the lab, you will learn how to use *always_comb* blocks and *case* statements (textbook Section 4.5) to design a 7-to-1 multiplexer. A module can contain any number of *always* blocks just the same as any module can contain any number of other module instantiations. System Verilog code for a 7-to-1 multiplexer built using a case statement is shown in Listing 1. The seven inputs are from the signals named `MuxIn[6:0]`. The output is called `Out`. The select lines are called `MuxSelect[2:0]`.

```
module part1(MuxSelect, MuxIn, Out);
  input logic [6:0] MuxIn;
  input logic [2:0] MuxSelect;

  output logic Out; // signal set by always_comb block

  always_comb // declare always_comb block
  begin
    case (MuxSelect) // start case statement
      3'b000: Out = MuxIn[0] // Case 0
      3'b001: Out = MuxIn[1] // Case 1
      3'b010: Out = MuxIn[2] // Case 2
      3'b011: ...
      3'b100: ...
      3'b101: ...
      3'b110: ...
      default: ...
    endcase
  end
endmodule
```

Listing 1: Code skeleton for 7:1 multiplexer

An *always_comb* block functions in a similar way to the *assign* statement you used in Lab 2. It is important to have a *default* case to ensure that all cases are covered. In the case of the 7:1 mux, the MuxSelect can be one of 8 possible values. Without a default case, no assignment to Out is made for case `0b111`. Failing to specify the output for all possible cases can lead to incorrect simulation results and be very hard to debug! By using a *default* statement, System Verilog will connect any mux inputs that are not explicitly enumerated in the case statement (such as `3'b111` in the mux above) to the signal in the *default* case.

To practice the approach you should take for the rest of this lab (and all remaining System Verilog labs), perform the following steps:

1. Draw a schematic of the 7:1 multiplexor with all wires, inputs and outputs clearly labeled.
2. After drawing your schematic, write the code that corresponds to your schematic. Your code should use the same names for the wires and instances shown in the schematic. You can use the code fragment from Listing 1 as a starting point. Complete Steps 1 and 2 as part of the pre-lab preparation.
3. Simulate your circuit with ModelSim for different values of `MuxSelect` and `MuxIn`.

Question: If you were to test every possible combination of inputs, how many tests cases would you need? Do you think you need to test every one of these cases? If not, how many different patterns of inputs do you need to simulate to have confidence that your circuit is working?

1.1 Running on the FPGA

If you have completed your design and want to see it running on an FPGA, follow the steps below. Running on the FPGA is strongly encouraged and can help you connect the *idea* of your circuit to working hardware with switches and lights!

1. Download the appropriate FPGA Kit from Quercus and unzip the contents into a new folder. Then, instantiate your module inside the `de1soc_top` module and connect the FPGA pins and the ports of your module. The mapping between the module ports and the FPGA pins is given in Table 1.
2. Compile the project to see if your code can be synthesized. It is possible, and not uncommon, to write code that simulates properly, but cannot be synthesized to working hardware. Just because it can be synthesized, doesn't mean it works! Hence, the reason for simulation.
3. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing $LEDR_0$.

module Port Name	Direction	DE1-SoC/DE10-Lite Pin Name
`MuxIn`	Input	SW[6:0]
`MuxSelect`	Input	SW[9:7]
`Out`	Output	LEDR[0]

Table 1: Module port mapping to DE1-SoC/DE10-Lite pin names

2 Part I

In this part, you must design a 4-bit Ripple Carry Adder (RCA). Figure 1(a) shows a circuit for a *full adder* (textbook Section 5.2.1), which has the inputs a, b, and c_i, and produces the outputs s and c_o. Note that Figure 1(a) shows one of many ways to implement a full adder circuit. You were shown a different, yet equally valid implementation in lecture. 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.

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 $c_o s = a + b + c_i$. Here, + refers to the arithmetic *addition operator* not the Boolean *or operator*. 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.

You must write System Verilog code that implements the circuit of Figure 1(d) following the steps below.

2.1 What to Do

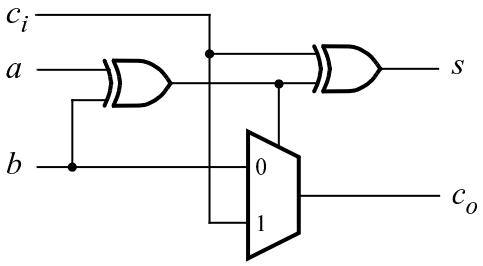
The module for your four-bit ripple-carry adder design should have the following signature declaration:

```
module part1(a, b, c_in, s, c_out);
```

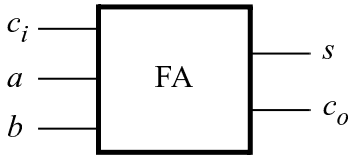
Note that `a` and `b` are the 4-bit inputs, `s` is the 4-bit sum output and `c_out` in this signature is the 4-bit vector of the four carry outputs from the four full adders in the form (c_{out},c_3,c_2,c_1) using the labels in Figure 1.

To build your `part1` module, perform the following steps:

1. Draw a schematic with all wires, inputs and outputs labeled. It should look similar to Figure 1.
2. Write the code that corresponds to your schematic. First, write a module for the full adder sub-circuit and then write the `part1` module that will instantiate four instances of



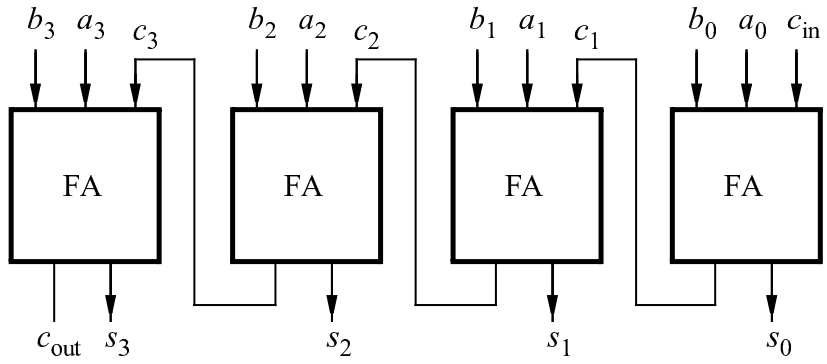
a) Full adder circuit



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.

your full adder module. Your code should use the same names for the wires and instances as shown in your schematic. Complete Steps 1 and 2 as part of your pre-lab preparation.

3. Simulate your adder with ModelSim for intelligently chosen values of the inputs a, b and c_{in}. When you are satisfied with your simulations, you can submit to the Automarker.

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.

If you wish to try running your circuit on an FPGA, you can use the mapping shown in Table 2.

module Port Name	Direction	DE1-SoC Pin Name
a	Input	SW[7:4]
b	Input	SW[3:0]
c_in	Input	SW[8]
s	Output	LEDR[3:0]
c_out	Output	LEDR[9:6]

Table 2: Module port mapping to DE1-SoC/DE10-Lite pin names

3 Part II

You must now implement a simple Arithmetic Logic Unit (ALU), which is used in processors to perform operations such as addition, subtraction and logical operations. You will learn more about the operations that a processor supports in the second half of ECE253. An ALU has two inputs and a single output, which is selected by the *Function* bits. To build an ALU, you should implement a mux to implement the different functions required. 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]*.

```
always_comb
begin
  case (Function)
  0: A + B using the Ripple Carry Adder you designed in Part I.
  1: Output 8'b00000001 if at least 1 of the 8 bits in the two
     inputs is 1 using a single OR operation.
  2: Output 8'b00000001 if all of the 8 bits in the two inputs
     are 1 using a single AND operation.
  3: Display A in the most significant four bits and B in the
     lower four bits.
  default: ...
endcase
end
```

Listing 2: Pseudo-code for ALU

You are asked to implement several different operations, so let's take a look at each of them in turn.

1. You must instantiate the ripple carry adder (RCA) module you wrote in Part I to perform this operation. **NOTE:** You cannot instantiate a module inside a case statement. You must instantiate your module outside the `always_comb` block and use a signal to connect

it to the mux. This is where having a schematic is essential. You should go back to your schematic to see exactly how to connect your RCA module with the mux.

2. You should also pay attention to the bit-widths of all your signals. How many bits will your RCA output when adding two 4-bit numbers?
3. For Functions 1 and 2, you must use reduction operations (textbook Section 4.2.3).
4. You will also need to use `concatenation` (textbook Section 4.2.9) to combine signals.
5. Most significant refers to the leftmost bits and least significant refers to the rightmost bits.

Implement your design in System Verilog, test it using ModelSim to make sure your design works before moving forward.

If you wish to implement your ALU on an FPGA, you can use the mapping shown in Table 3. We recommend displaying the values of A and B on $HEX[2]$ and $HEX[0]$, respectively so it is easier to visually check the functioning of your ALU. Display $ALUOut[7:4]$ on $HEX[4]$ and $ALUOut[3:0]$ on $HEX[3]$. You can re-use the Hex decoder module you wrote for Lab 2 or write a new one using a `case` statement, as shown in class.

Note: The KEY inputs are inverted. This means that when a KEY is not pressed, it has a value of 1 and when pressed has a value of 0.

module Port Name	Direction	DE1-SoC Pin Name
A	Input	SW[7:4] and HEX[2]
B	Input	SW[3:0] and HEX[0]
$Function$	Input	KEY[1:0]
$ALUOut$	Output	LEDR[7:0] and HEX[4], HEX[3]

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

4 Part III

So far, we have built circuits using a fixed number of bits for inputs and outputs. However, System Verilog allows you to specify the size of inputs and outputs when instantiating the module using `parameters` (textbook Section 4.8). Shown below is an example of a parameterized 2:1 multiplexer.

```
module mux_2to1_Nbit(MuxIn0, MuxIn1, MuxSelect, MuxOut)
  parameter N = 4;
  input logic [N-1:0] MuxIn0, MuxIn1;
  input logic MuxSelect;
```

```
  output logic [N-1:0] MuxOut;

  assign MuxOut = MuxSelect?MuxIn0, MuxIn1;
endmodule
```

Listing 3: Parameterized 2:1 Multiplexer

There is now an additional line which indicates that a `parameter` called N as been created. The width of the signals `MuxIn0`, `MuxIn1` and `MuxOut` is then set based on N. You can also specify a default value for the parameter (4 in this case).

Instantiating a parameterized module is similar to instantiating any other module, except that you can pass in a value for the parameter. In the example below, the module `u0` will be instantiated with a value of $N = 8$.

```
mux_2to1_Nbit #(8) u0(MuxIn0, MuxIn1, MuxSelect, MuxOut)
```

If you do not specify a value when instantiating a module, the default value specified in the module will be used. Try to vary the size of the parameter and check the difference in the instantiated modules in ModelSim before proceeding.

4.1 What to Do

Modify the ALU you wrote in Part II to support passing a parameter to indicate the bit-width of inputs A and B. How can you calculate the correct bit-width for $ALUout$? The module you are writing for Part III should have the following signature declaration:

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

You should use a default parameter size of $N = 4$. A few things to keep in mind when modifying your design:

1. You will not be able to use your RCA from Part I–it only works for 4-bit inputs.[1] Instead, you can directly perform addition using the '+' operator. The '+' operator will automatically instantiate the correctly sized adder based on the widths of your inputs.
2. For functions 1 and 2, you must once again use reduction operations. While you may have been able to implement the required operation in Part II without using a reduction, in Part III, only a reduction operator will work for Part III.

[1] It is possible to modify your RCA to work for arbitrary bit-widths but that is outside the scope of ECE253.

3. For function 3, you should output A in the most significant N bits of $ALUout$ and B in the least significant N bits of $ALUout$.

Test your ALU for a few different bit-widths to make sure your parameterized ALU works as expected. Once you are satisfied, you can submit your code for automarking.

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.sv` with the following module in it:

```
module part1(a, b, c_in, s, c_out);
```

5.2 Part II

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

```
module part2(A, B, Function, ALUout);
```

5.3 Part III

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

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