Laboratory Exercise 3

The *case* statement, Adders and ALUs

Revision of September 28, 2023

In this lab, you will learn about designing multiplexers using 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` block in Verilog.

1 Always blocks and Case statements

In this part of the lab, you will learn 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. You will in future labs and lectures learn why the @(*) is necessary, but for the time being it should always be used with *always* blocks. A module can contain any number of *always* blocks just the same as any module can contain any number of other module instantiations. 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]`.

Some notes with regard to *always* blocks, *wires*, *regs*, and modules are:

1. While you can use *wires*, you can not assign a value to a *wire* inside an *always* block. Instead, you need to use *reg* variables.
2. In turn, *reg* variables can not be assigned outside *always* blocks, but they can be used as inputs.
3. Modules can only be instantiated outside *always* blocks.
4. The outputs of instantiated modules must target *wires* as they are instantiated outside *always* blocks. This is true even if the variable is declared as *reg* inside the module.
5. *regs* do not require the *assign* keyword.

With *case* statements inside *always* blocks it is important to have a *default* case to ensure that all cases are covered. In the case of the 7:1 mux in Listing 1, the MuxSelect can be one of 8 possible values. Without a default case, no assignment to `Out` is made for case `0'b111`. In hardware that means those pins on the mux are unconnected or connected somewhere randomly. Failing to specify the output for all possible cases can lead to strange implementation and simulation results and be very hard to debug! By using a *default* statement, 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.

```
module reshaping_mux(MuxSelect, MuxIn, Out);
  input wire [6:0] MuxIn;
  input wire [2:0] MuxSelect;
  output reg Out; // signal is set inside the always block
  wire [6:0] ReshapedMuxIn;
  reshape reshapeModuleName (.in(MuxIn),.out(ReshapedMuxIn));
    //We instantiate a reshape module and connect wire MuxIn to
        the input and wire ReshapedMuxIn to the output. All
       modules need a name so we call it reshapeModuleName
  always@(*)  // declare always block
  begin
    case (MuxSelect) // Construct a Mux
      3'b000: Out = ReshapedMuxIn[0]; // Case 0
      3'b001: Out = ReshapedMuxIn[1]; // Case 1
      3'b010: Out = ReshapedMuxIn[2]; // Case 2
      3'b011: Out = ReshapedMuxIn[3]; // Case 3
      3'b100: Out = ReshapedMuxIn[4]; // Case 4
      3'b101: Out = ReshapedMuxIn[5]; // Case 5
      3'b110: Out = ReshapedMuxIn[6]; // Case 6
      default: Out = 0; // Connect all other mux inputs to 0
    endcase
  end
endmodule
module reshape(in, out);
    input wire [6:0] in;
    output reg [6:0] out;
    always@(*)
    begin
        //Perform the reshaping. Using assign statements and a
           wire "out" would have been equivalent.
        out[6] = in[0];
        out[5] = in[1];
        out[4] = in[2];
        out[3] = in[3];
        out[2] = in[4];
        out[1] = in[5];
        out[0] = in[6];
    end
endmodule
```

Listing 1: Code for a 7:1 multiplexer which reshapes the inputs using the reshape module before they are used. While not the most efficient implementation, this highlights the various properties of always, reg, wire, and module instantiations that you need to know.

In Verilog, you are not writing code, you are giving a computer instructions to construct a circuit on a virtual breadboard inside the chip. Your design approach for the rest of this lab (and all remaining labs) should be to perform the following steps:

1. Draw a schematic of the circuit you are building with all wires, inputs and outputs labeled.
2. After drawing your schematic, write the code that corresponds to your schematic. Your code should use the same names for the wires, registers, and instances shown in the schematic.
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]

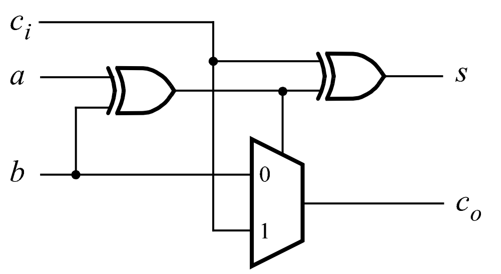
Table 1: Module port mapping to DE1-SoC/DE10-Lite pin names

2 Part I

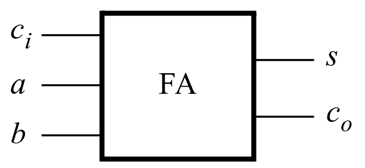
In this part, you must design a 4-bit Ripple Carry Adder (RCA). Figure 1(a) shows a circuit for a *full adder* (textbook Section 3.2), which has the inputs a, b, and c_i, and produces the outputs s and c_o. Note that Figure 1(a) shows one of many ways to implement a full adder circuit. You were shown 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 Verilog code that implements the circuit of Figure 1(d) following the steps below.



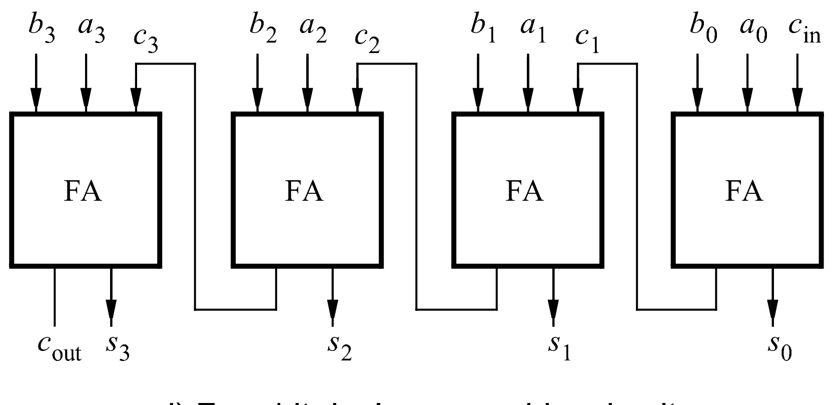
a) Full adder circuit



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.

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. Write the code for part 1 using the schematic in Figure 1d as a reference. First, write a module for the full adder sub-circuit and then write the `part1` module that will instantiate four instances of 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.
2. 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. 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@(*)
begin
  case (Function)
  0: A + B using the Ripple Carry Adder you designed in Part I
     instantiated as a separate module
  1: Output 8'b00000001 if at least 1 of the 8 bits in the two
     inputs is 1 using a single reduction OR operation.
  2: Output 8'b00000001 if all of the 8 bits in the two inputs
     are 1 using a single reduction 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. Draw a schematic with all wires, inputs and outputs labeled. It should look similar to Figure 1. You can use logic gates, multiplexers, and can draw blocks labeled *Part 1* to represent an instantiation of the part 1 module.
2. 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` 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.
3. 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?

4. For Functions 1 and 2, you must use `reduction`[1] operations and `concatenation`[2] to combine signals (textbook Sections A.7, 4.6.5, 3.5.6).
5. Most significant refers to the leftmost bits and least significant refers to the rightmost bits.

Implement your design in 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.

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

[1]The reduction operators use the same symbols as some of the bitwise operators, but have only one operand. The reduction B=&A produces the AND of all of the bits in A placing the single bit result in B, while B=∼&A produces the NAND. Similarly, the other reduction operators produce single-bit Boolean results.

[2]The {,} concatenate operator allows vectors to be combined to produce a larger resulting vector. Any operand that is a constant must have a specified size, as in 4'b0011. Operands can be repeated multiple times by using the replication operator. The operation {{3{A}}, {2{B}}} is equivalent to {A, A, A, B, B}. The replication operator can be used to form an n-bit vector of digits: the operation {n{1'b1}} represents n ones.

4 Part III

So far, we have built circuits using a fixed number of bits for inputs and outputs. However, Verilog allows you to specify the size of inputs and outputs when instantiating the module using `parameters` (textbook Section A.5, A.10, A.12). Shown below is an example of a parameterized 2:1 multiplexer.

```
module mux_2to1_Nbit (MuxIn0, MuxIn1, MuxSelect, MuxOut);
  parameter N = 4;
  input wire [N-1:0] MuxIn0, MuxIn1;
  input wire MuxSelect;
  output wire [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.[3] 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.
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 to the Automarker.

[3]It is possible to modify your RCA to work for arbitrary bit-widths but that is outside the scope of ECE241.

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.

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

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

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

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

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

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

5.1 Automarker Commands

The submission command is:

```
/cad2/ece241f/public/submit 3 part1.v part2.v part3.v
```

Assignments submitted with an incorrect command **will not be graded** (e.g., wrong course or lab number). You can check if it was submitted using the command:

```
/cad2/ece241f/public/check_submission 3
```

You may also run a sanity check using this command in the same directory where your files are:

```
/cad2/ece241f/public/3/tester
```

Note the tester only tests a couple trivial cases to make sure the file and module names are correct, it is not a full test of your code. You need to run simulations.