Laboratory Exercise 3

Ripple Carry Adder and Combinational and Sequential ALUs

Revision of October 3, 2024

In this lab, you will explore both combinatorial and sequential design by building a simple Arithmetic Logic Unit (ALU) and learning about hierarchy. First, you will design a 4-bit ripple carry adder, then use it to implement an combinatorial ALU. Then, you will learn how to incorporate sequential storage elements like D-flip-flops to design a sequential ALU.

1 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.

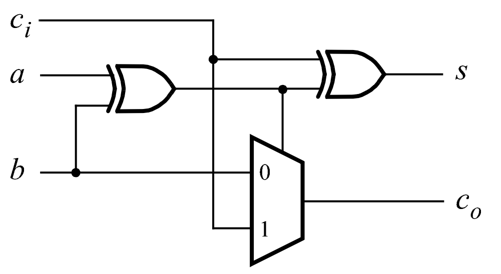
1.1 What to Do

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

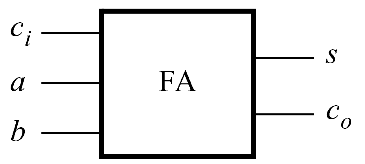
```
module part1(input logic [3:0] a, b, input logic c_in,
             output logic [3:0] s, c_out);
```

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:



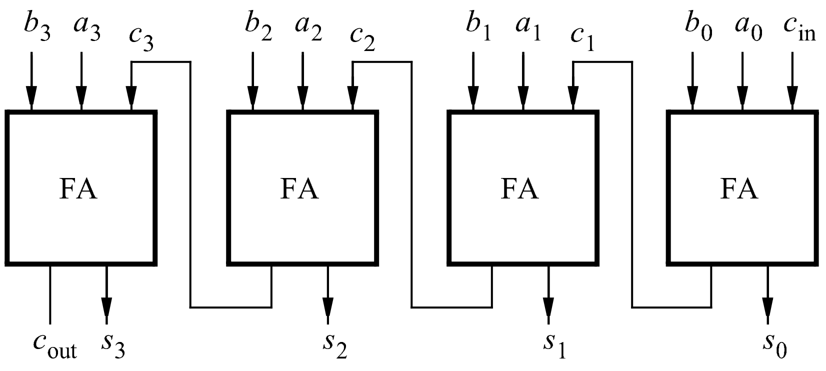
a) Full adder circuit



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.

Pre-lab preparation:

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 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.

In Lab:

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.

2 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 use a mux to select among 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 1: 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.

2.1 What to Do

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

The top-level module of your design should have the following signature declaration:

```
module part2(input logic [3:0] A, B, input logic [1:0] Function,
             output logic [7:0] ALUout);
```

3 Part III

In this part, you will extend the ALU to include sequential elements using registers to store values. A register consists of D flip-flops and is used to store intermediate results.

Note: The ALU you will build in this section will have different functionalities from the ALU in Part II.

3.1 D Flip Flops

Registers are composed of edge-triggered D flip-flops, which update their output on a clock edge. For a positive edge-triggered flip flop, the output changes when the clock edge *rises*. The code for a positive edge-triggered flip flop is shown in Listing 2 (textbook Section 4.4.2).

```
module D_flip_flop(
    input logic clk,
    input logic reset_b,
    input logic d,
    output logic q
);
    always_ff @(posedge clk)
    begin
        if (reset_b) q <= 1'b0;
        else q <= d;
    end
endmodule
```

Listing 2: Code for a 1-bit D-Flip flop with synchronous active-high reset.

There are several important things to note about the code shown in Listing 2:

1. The flip-flop is created using a new `always_ff` block. System Verilog uses different always blocks to make it easier to spot whether the code is creating combinational (using `always_comb`) or sequential (using `always_ff`) logic.
2. Statements inside `always_ff` blocks use <= (non-blocking) instead of regular = (blocking) assignments. Using = inside `always_ff` blocks can lead to your circuit behaving in odd ways. Always double check that you are using the right type of assignment.
3. This flip flop has an **active-high, synchronous reset**, meaning that the reset only happens when *reset_b* =1 on the rising clock edge.
4. If q is declared as **logic** q, then you get a single flip flop. If q is declared as **logic[7:0]** q, then you get eight parallel flip flops, which is called an *8-bit register*. Of course, d should have the same width as q.

NOTE: Think carefully about the order of if/elseif/else statements inside your always block. In order for the circuit to behave as we expect, you should check the reset condition first inside your `always_ff` block.

3.2 Designing a sequential ALU

You must build a new ALU with the four operations shown in Listing 3.

Figure 2 shows a high-level schematic of the ALU you must build. The output of the ALU is to be stored in an 8-bit *register* and the four least-significant bits of the register output are connected to the B input of the ALU. The 8-bit register should have an **active-high synchronous** reset.

```
always_comb
begin
  case (Function)
      0: A + B the '+' operator.
      1: A * B using the '*' operator.
      2: Left shift B by A bits using the shift operator.
      3: Hold current value in the Register, i.e., the register
          value does not change.
      default: ...
  endcase
end
```

Listing 3: Pseudo-code for Part III's ALU

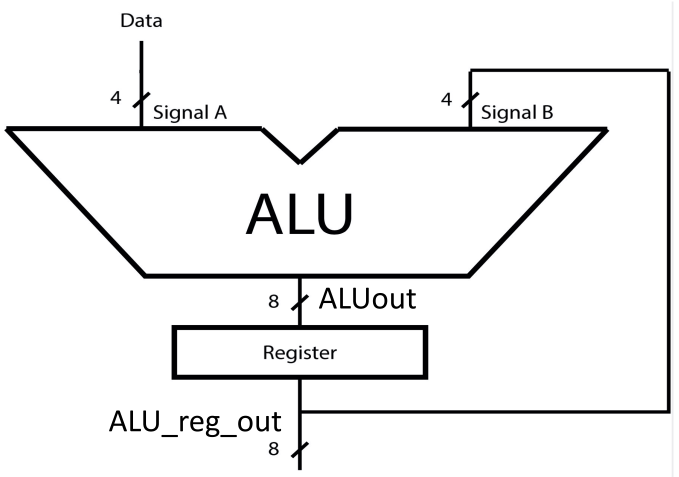


Figure 2: Simple ALU with register circuit for Part III.

3.3 What to Do

The top-level module of your design should have the following signature declaration:

```
module part3(input logic Clock, Reset_b, input logic [3:0] Data, input logic
             [2:0] Function, output logic [7:0] ALU_reg_out);
```

Pre-Lab work:

1. *Prior to coming to lab,* draw a schematic showing your code structure with all wires, inputs and outputs labeled.
2. After drawing your schematic, write code that corresponds to your schematic. Your System Verilog code should use the same names for the wires and instances as shown in your schematic. Use the code in Listing 2 as the model for your register code.

In Lab:

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 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.

4.1 Part I

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

```
module part1(input logic [3:0] a, b, input logic c_in, output logic [3:0] s, c_out);
```

4.2 Part II

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

```
module part2(input logic [3:0] A, B, input logic [1:0] Function, output logic
[7:0] ALUout);
```

4.3 Part III

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

```
module part3(input logic Clock, Reset_b, input logic [3:0] Data, input logic [2:0]
Function, output logic [7:0] ALU_reg_out));
```

5 Optional: Running your designs on the FPGA board

5.1 Part I

If you wish to try running your circuit on an FPGA, follow the steps introduced in Lab 2 and use the mapping below (Table 1).

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

5.2 Part II

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 $HEX2$ and $HEX0$, respectively so it is easier to visually check the functioning of your ALU. As part of the lab handout, you are provided with a file called `hex.sv`. This file has a `hex_decoder` module, that takes a 4 bit input and maps it on a 7-segment hex display, as well as a `hex_top` module that uses the `hex_decoder` module to connect inputs from SW[3:0] onto HEX0. Use the `hex.sv` file as an example to connect the values of A and B on $HEX2$ and $HEX0$ respectively. Display $ALUOut[7:4]$ on $HEX4$ and $ALUOut[3:0]$ on $HEX3$.

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 HEX2
B	Input	SW[3:0] and HEX0
$Function$	Input	KEY[1:0]
$ALUOut$	Output	LEDR[7:0] and HEX4, HEX3

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

5.3 Part III

If you wish to run your code on the FPGA, you can use the port mapping shown in Table 3.

NOTE 1: When using Quartus, avoid module names such as *dff*. This is a reserved Quartus keyword and modules with this name will not be synthesized. This is reported as a *warning* in Quartus but it's easy to miss.

Note 2: All mechanical switches, such as a push/toggle button, will often make contact several times due the electrical contacts bouncing. This happens quickly in human time, but not in electrical time. With a bouncing switch you can observe multiple high-frequency

module Port Name	Direction	DE1-SoC Pin Name
`Clock`	Input	KEY[0]
`Reset_b`	Input	KEY[1]
`Data`	Input	SW[3:0] and HEX0
`Function`	Input	SW[9:8]
`ALU_reg_out`	Output	LEDR[7:0] and HEX5, HEX4

Table 3: Module port mapping to DE1-SoC/DE10-Lite pin names

toggles making it difficult to create single clock edges. If you run into bounce problems with KEY_0 for your clock you can try to use KEY_1 instead to see if that is better.