CS391R1: Computer & Memory Architectures
Lab 1: ALU

Due: Friday Sept 26, 2025 at 2:30pm ET (before Discussion Section)

---

1. Part 1: Design Exercises (Individual Work)

Introduction

We'll begin with some written problems and design exercises. You can submit your answers as a PDF (see submission instructions at the end of the document). We strongly encourage you to show your work! Showing your work and writing comments makes it much more likely that graders can award partial credit on problems.

All exercises in this part are to be done as **Individual** work.

Problem 1.1: Review of Binary Addition

When building an ALU, one of the core operations it has to perform is binary addition, often on signed (positive and negative) numbers. Recall that subtraction can be performed by converting a number to negative and doing addition. Also recall that to convert between positive and negative numbers, you can "flip the bits and add one."

Calculate (by hand) the following additions using 6-bit 2's complement binary arithmetic (i.e., addition in base 2 keeping only 6 bits of your answer). Show your work, in binary notation.

(a) 12 + 11

```
  0 0 1 1 0 0
+ 0 0 1 0 1 1
-------------
  0 1 0 1 1 1
```

(b) 13 - 18

```
  0 0 1 1 0 1
+ 1 0 1 1 1 0
-------------
  - - - - - -
```

(c) 24 - 6

```
  0 1 1 0 0 0
+ 1 1 1 0 1 0
-------------
  0 1 0 0 1 0
```

(d) -10 - 13

```
  1 1 0 1 1 0
+ 1 1 0 0 1 1
-------------
  1 0 1 0 0 1
```

(e) 19 + (-19)

```
  0 1 0 0 1 1
+ 1 0 1 1 0 1
-------------
  0 0 0 0 0 0
```

(f) 31 + 10

```
  0 1 1 1 1 1
+ 0 0 1 0 1 0
-------------
  1 0 1 0 0 1
```

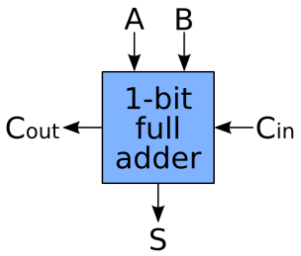
(g) Briefly explain what happened in the last addition. In what sense is your answer technically "correct"?

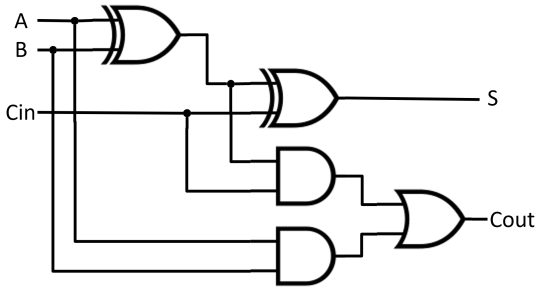
the 5th bit overflowed into the "sign" bit, making the result negative by 2's complement. By basic binary addition rule, it to treat the 6th

bit as a normal positive bit, it is "correct".

Problem 1.2: Designing a Combinational Circuit for Addition

One way to do binary addition is with a “ripple carry adder”. The ripple carry adder strings together a chain of single-bit “full adders”, sending the carry-out signal from the lower-order bits to the higher-order bits.





Gate	tPD	tCD
AND	25ns	8ns
OR	20ns	10ns
XOR	50ns	12ns

MUX 10ns 5ns

Fig. 1. Full adder and a combinational logic circuit implementing the full adder.

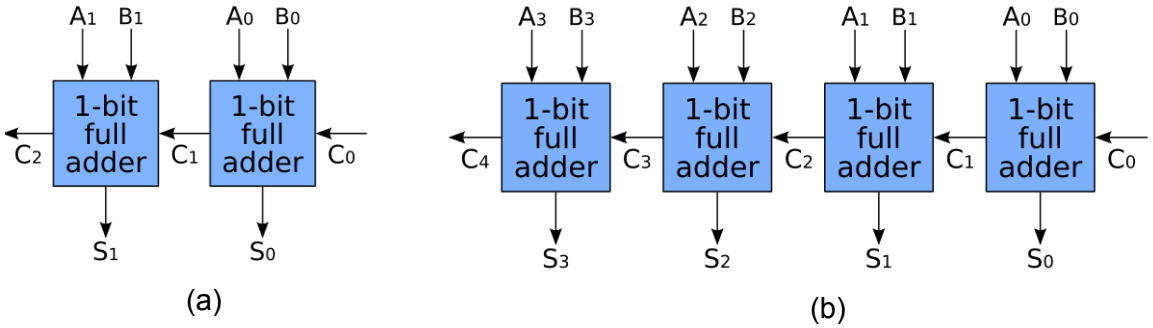


Fig. 2. Ripple carry adders: (a) two-bit adder, and (b) four-bit adder.

(a) Please fill in the missing truth table values below for the two-bit ripple carry adder (Fig. 2a). We will assume that C0 is permanently set to “0”. The table is divided into two parts.

A1	A0	B1	B0	C2	S1	S0	
0	0	0	0	0	0	0	
0	0	0	1	0	0	1	
0	0	1	0	0	1	0	←
0	0	1	1	0	1	1	←
0	1	0	0	0	0	1	
0	1	0	1	0	1	0	←
0	1	1	0	0	1	1	
0	1	1	1	1	0	0	←

A1	A0	B1	B0	C2	S1	S0	
1	0	0	0	0	1	0	
1	0	0	1	0	1	1	
1	0	1	0	1	0	0	
1	0	1	1	1	0	1	←
1	1	0	0	0	1	1	
1	1	0	1	1	0	0	←
1	1	1	0	1	0	1	←
1	1	1	1	1	1	0	←

Say that we assemble an 8-bit ripple carry adder out of two 4-bit ripple carry adders as pictured below. Assume that the 4-bit adders are assembled from the single-bit full adder circuit pictured above in Fig. 1.

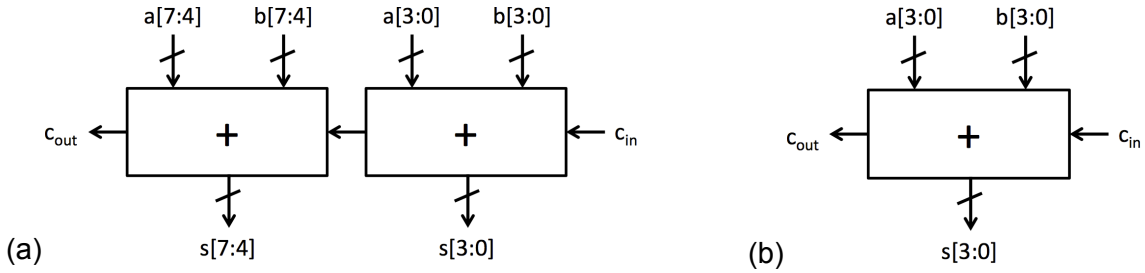


Fig. 3. An (a) 8-bit ripple carry adder, made from two cascaded (b) 4-bit adders (see Fig. 2b).

(b) How many total 1-input or 2-input logic gates (e.g., 1-input NOT gates, 2-input XOR gates, 2-input MUX gates) are required to build the 8-bit ripple carry adder design?

5x(4+2) = 40

(c) What is the tpd of the 8-bit ripple carry adder? Show your work. What was your approach to find this answer?

there, 8-bit ripple carry adder = 8 of 1-bit full adder, and each full adder need previous ones to finish a set of data to start working on it.

res = 8x(50+50) = 800 ns.

(d) What is the tcd of the 8-bit ripple carry adder? Show your work. What was your approach to find this answer?

res = 8x12ns = 96 ns

Here's a second potential design for the 8-bit adder, called a "carry-select adder" (see Fig. 4). It is assembled from 4-bit ripple carry adders (see Fig. 2b) and 2-input muxes.

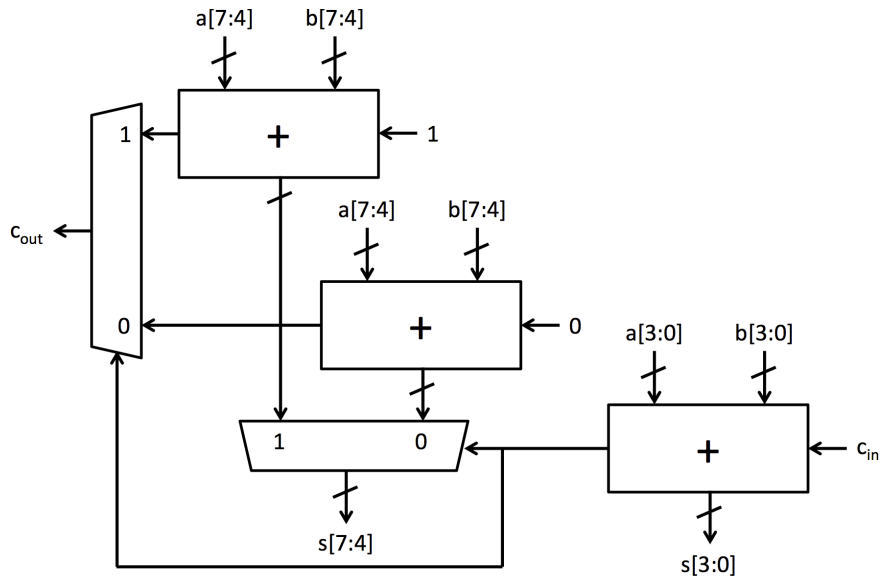


Fig. 4. An 8-bit carry-select adder, assembled from 4-bit ripple-carry adders (see Fig. 2b).

(e) In words, briefly describe what you think this circuit does, and how it might work.

This circuit allows less tpd as the circuit can calculate the 2nd 4-bit add together while the 1st 4-bit data is being calculated. Previously, we need Cout from 1st 4-bit to start calculating 2nd 4-bit, now we calculate both cases and keep one of the result depending on Cout.

(f) How many total 1-input or 2-input logic gates (e.g., 1-input NOT gates, 2-input XOR gates, 2-input MUX gates) are required to build the 8-bit carry-select adder design?

$5 \times 4 \times 3 + 2 = 62$

(g) What is the tpd of the 8-bit carry-select adder? Show your work.

$4 \times (50 + 50) + 10 = 410$ ns

(h) What is the tcd of the 8-bit carry-select adder? Show your work.

$4 \times (12\text{ns}) + 5 = 53$ ns

(i) What do you think might be potential advantages of the carry-select adder design versus the ripple-carry adder design?

It has less tpd, we do not have to allow as much time as with the ripple-carry adder design to ensure the output is valid.

(j) What do you think might be potential disadvantages of the carry-select adder design versus the ripple-carry adder design?

It has less tcd, it is easier and faster for a invalid input to prevent us from ensuring the output is valid compared to the ripple-carry adder design.

Problem 1.3: Designing Sequential Logic (FSMs) to Test an Adder

Let's say that we want to allow users to perform a sequence of additions on a combinational 2-bit ripple carry adder.

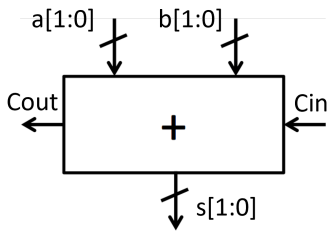


Fig. 5. A 2-bit ripple carry adder (see Fig. 2a).

We would like to use finite state machines to perform the following six additions in sequence:

0+0, 00
1+0, 01
1+0, 01
1+1, 10
2+1, 11
3+2 11+10 = 01 Cout=1

We design the following device with sequential logic. It has an input button for the user to press to move on to the next addition in the sequence (see above). This input button connects to the "input generator" finite state machine (FSM), FSM_inputgen. The outputs of FSM_inputgen connect to the input signals of the 2-bit ripple carry adder. The outputs of the 2-bit ripple carry adder then are given as inputs to a second "output checker" FSM, FSM_outputcheck. The output of FSM_outputcheck is a binary "success" signal that is "1" when the user successfully completes all six desired additions and their outputs are correct, and is "0" otherwise.

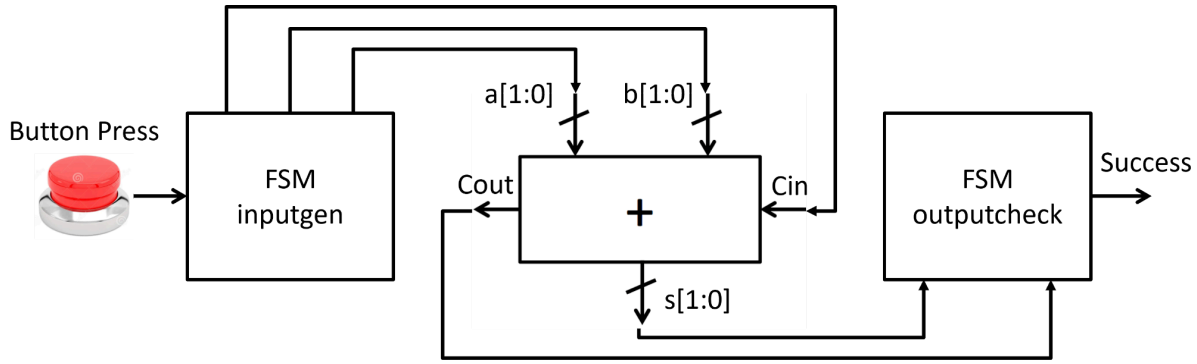
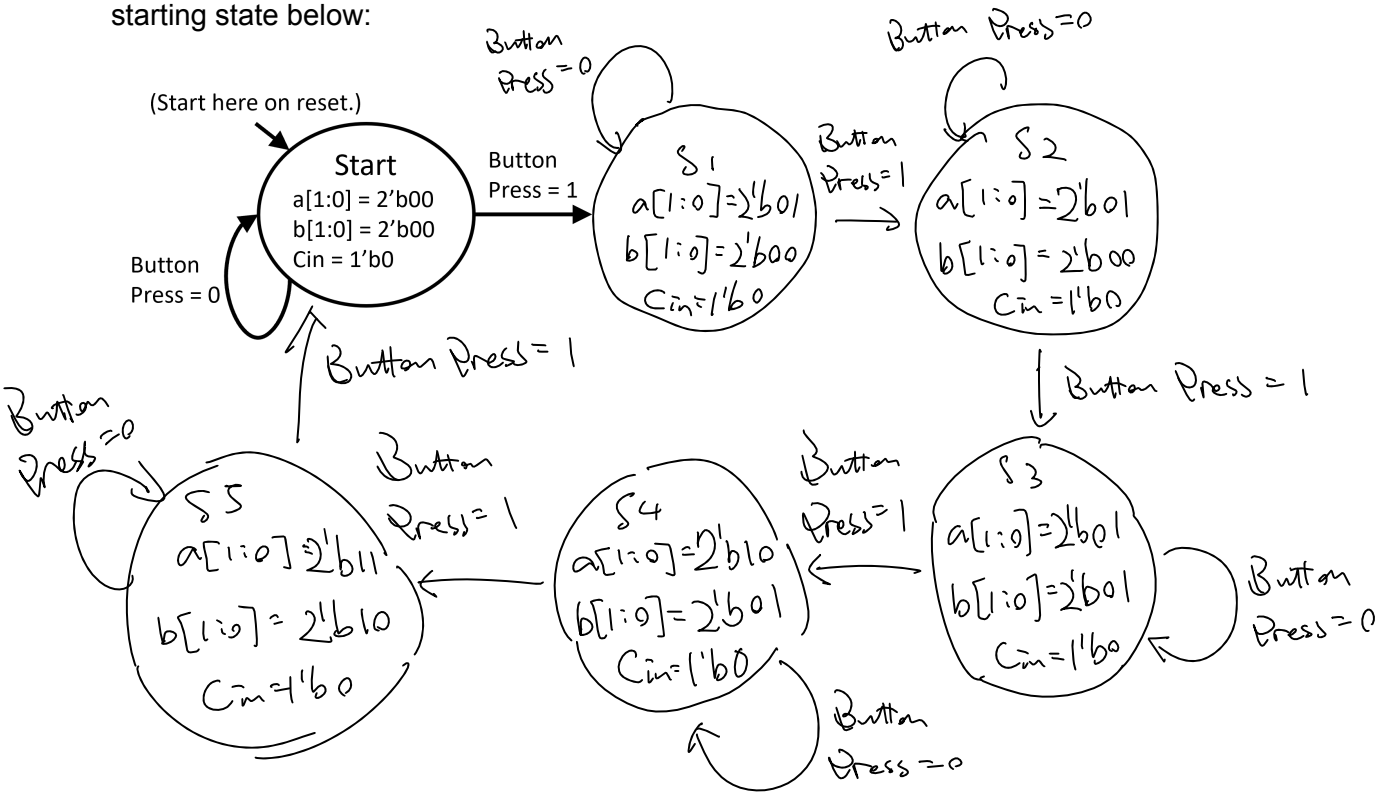
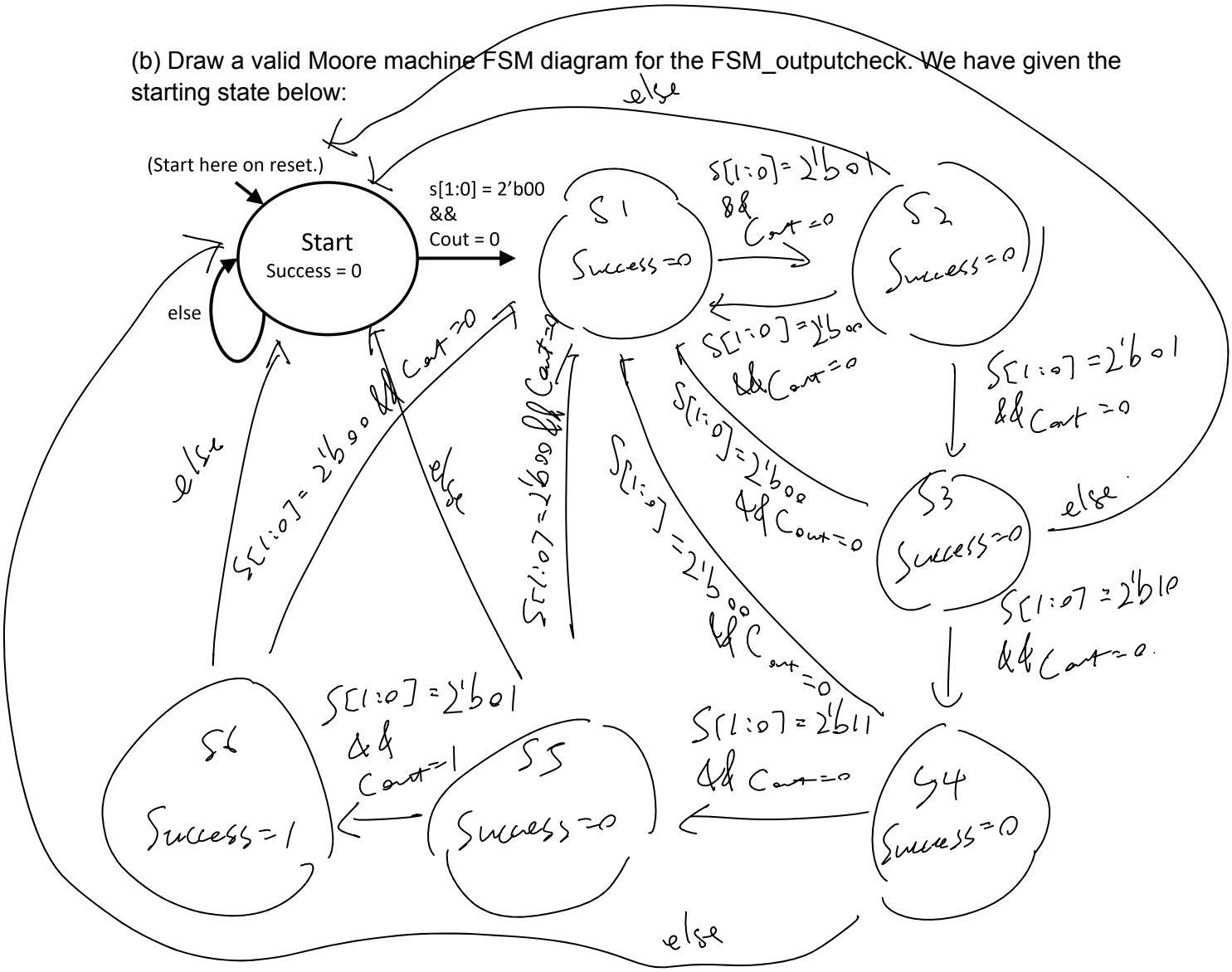


Fig. 6. Diagram of FSM-based testing system for 2-bit adder.

(a) Draw a valid Moore machine FSM diagram for the FSM_inputgen. We have given the starting state below:



(b) Draw a valid Moore machine FSM diagram for the FSM_outputcheck. We have given the starting state below:



(c) Now, say that someone gives you a broken 2-bit adder, where the S1 output signal is stuck always outputting a "0". What happens when a user tries to use the input button and step through the six addition operations above? Describe your answer in terms of the state transitions and signals in your FSM_inputgen and FSM_outputcheck diagrams above.

FSM_inputgen will work normally, repeating it self in cycle; FSM_outputcheck will not be able to go further than S3, it will move to Start, and stay there for 3 button pushes, then move on to S1 → S2 → S3 and repeat the same process again.

2. Part 2: Lab Activity (Individual Work)

Introduction

In this lab activity, we will build an ALU (Arithmetic and Logic Unit). The ALU is a crucial part of any processor architecture. It is a circuit that is responsible for performing arithmetic and logic operations. We will start with a very simple design and gradually build more functionality into it. By the end of this lab, we should have an ALU as a standalone module which supports numerous logical (NOT, AND, OR, XOR, XNOR, UNSIGNED SHIFT) and arithmetic (ADD, SUBTRACT, LESS, EQUALS, GREATER, SIGNED SHIFT) operations on multi-bit inputs. The ALU that you build in this lab will be an important building block for future assignments, so make sure you fully understand how it works and that you are able to debug and modify it.

All exercises in this part are to be done as **Individual** work.

Required: Make Intermediate Github Git Commits of Your Code

In modern digital design, we are often working with other engineers in large teams, and making careful modifications to a complex larger system. This makes it critical to document the step-by-step changes that we make to a Verilog code base through a series of git commits with comments as we work. Frequent and well-commented git commits can also save progress and prevent having to start over in the event that local data becomes corrupted.

To encourage best coding practices, in this class we will submit code for our lab activities by submitting private Github repositories with the required files in them. We will be looking for a record of intermediate git commits of work-in-progress (with comments), and this will be part of the grade for the assignment. So, remember: Commit early, commit often!

Problem 2.1: Basic ALU and Testbench

First, we start with something very simple. Let’s create a SystemVerilog module (you can call it alu.sv) that takes in one single-bit input wire (call it op1 which stands for operand 1) and produces a NOT of it (call the output wire res, which stands for result).

Create a testbench file (call it alu_tb.sv) in the Simulation Sources directory. Write tests to make sure your module works as desired.

Problem 2.2: NOT and XOR

Now, let's expand the functionality of our module. Introduce two new single-bit input wires for a second operand and control (call them op2 and control).

Update your ALU code to behave as follows:

- If control wire is LOW, res should be a NOT of op1. Same behavior as before.
- If control wire is HIGH, res should be an XOR of op1 and op2.

Update your testbench to verify that your module works correctly.

Problem 2.3: Parameterizing Bit-Width

So far we have worked with single-bit inputs to our ALU. Let's introduce a parameter (call it OP WIDTH) which can be set when the module is instantiated. This parameter will determine the bit length of the operands and the result. Make sure that both of the operations still work correctly (NOT and XOR should be bit-wise operations).

Update your testbench.

Problem 2.4: AND, OR, XNOR

Let's update our module to include the remaining logical operations (AND, OR, XNOR).

Hints:

- When you do this, the first issue that you'll encounter is that our control wire is a single bit, so it can only represent two operations. Think about how you can overcome this.
- Once you have figured out the control wire issue, you might notice that in some cases our ALU has an undefined behavior. Introduce a new single-bit output wire called error. This wire should be raised HIGH for inputs that either result in a computational error (e.g. division by zero) or do not have a defined behavior. Otherwise, it should be LOW. From now on, when you update your ALU, always make sure that you properly set this error output.

Update your testbench.

Problem 2.5: Shifts

Update the ALU to include logical (UNSIGNED) and arithmetic (SIGNED) bit-wise shifts. op1 will be the input on which we want to perform the shift. op2 will be the input that determines by how many bits we shift.

Update your testbench.

Problem 2.6: Addition and Subtraction

Update the ALU to include simple arithmetic operations: addition and subtraction. To have some common convention, in the case of subtraction set res = op1 - op2.

Update your testbench.

Problem 2.7: Comparisons

Update the ALU to include comparison operations: less than, equals to, greater than. The first bit of the result output determines whether the comparison is true (1'b1) or false (1'b0). Other bits of the result should be set to 0.

To have some common convention, make sure the op1 is always on the left side of the operation. For example, if we do less than, res = op1 < op2.

Update your test-bench.

Extra Challenges: More ALU Ops

Congratulations, you have completed the required part of the lab! However, we don't want to let you off so easily, so we prepared some additional challenges for you. All of these challenges are completely optional, however, we strongly encourage you to try to do some of them as they can be very useful for future assignments.

Here is the list of challenges in the order of increasing difficulty:

- Introduce the remaining logical operations: NAND, NOR, XNOR.
- Introduce not bit-wise logical operations: AND, OR, NOT. They should behave the same as &&, ||, ! do in the C programming language.
- Complete the comparison functionality of your ALU. Introduce greater than or equal to, less than or equal to, not equal to operations.

- Complete the arithmetic functionality of your ALU. Introduce multiplication, integer division, modulus, and exponent operations.
- Create a new module that takes in a single 4-byte input and produces a boolean output which determines if at least one of the bytes in the input is zero. Hint: use your ALU module inside the new module.

Don’t forget to update your testbench.

3. Submission Instructions

We will use Gradescope for all submissions. See detailed instructions below.

Design Exercises: Submit your written solutions to the Lab 1 Design Exercises as a PDF file. We strongly encourage you to show your work! Showing your work and writing comments makes it much more likely that graders can award partial credit on problems.

Source Code: You should submit your complete source code which includes any of the module sources and testbenches.

Follow the same instructions as Lab 0:

1. Create a **private** GitHub repository.
2. Push your code to that repository. Remember to make the required intermediate git commits of work-in-progress as you go!
3. These are all **Individual** exercises in Lab 1, so please submit your own work. You can submit directly from your GitHub repository. If you are having difficulties submitting your code to Gradescope from GitHub, please see the TF at Discussion or office hours, and let us know as soon as possible before the deadline, to have time to debug.

Lab Report: In addition to that, you should include a concise report (as a PDF file) of how you approached the problems, what challenges you encountered, and if you have any questions and/or recommendations. There is no need to write an in-depth report, we just want to see that you have a decent understanding of what you implemented.

Summary of Deliverables:

- Design Exercises PDF file
- Source Code
    - alu.sv
    - alu_tb.sv
    - … and any other supporting source code or files, as needed
    - Remember to make the required intermediate git commits as you go!
- Lab Report PDF file