Programming Assignment 2: ALU

CS141 Spring 2019

Part 1 due February 15, and Part 2 due February 22

Assignment

Digital circuits made out of gates (combinational logic) are ideal for arithmetic problems. Any truth table can be converted into a series of gates, but once the number of inputs/outputs grows large, it can quickly become intractable to do it this way. A 32 bit adder has, for instance, $> 2^{64}$ possible combinations of inputs. This can cause problems both with managing the design complexity, and in simulating and verifying that the design works. Even if you could simulate one possible input every nanosecond, it would take over 600 years for you to run all 2^{64} combinations.

Modular circuits solve both of these problems. They allow us to use simple building blocks to create complex behaviors, and if we can simulate each small module and make sure it works, then all we have to debug in larger designs is making sure that we wired the smaller modules correctly.

Remember to take a look at the Verilog cheat sheet on the course website for any shorthand that may be helpful in making your code smaller and more elegant.

Design an ALU

Your assignment for this week is to design and simulate a functioning 32 bit ALU according to the following specification. In all of your design files, you should be careful to only use structural Verilog. On the testbenches, you can use any Verilog construct.

> Structural Verilog is the subset of the language that defines things at an explicit gate/wiring level. As long as your modules have only wires and assign statements that only use the operators `&,|,^,~,?` you are using structural Verilog. You can also instantiate other modules that only use these simple types of assigns. DO NOT use operators like `+, ==, *, /, %, <<, >>` on this assignment outside the testbench! You may use the concatenation operator `{}` on arrays of fixed size only and may not use replication (`N{}`). Since we are making a 32 bit ALU you may find it helpful to use `generate` statements to shorten your code. Please see the end of the document for a short tutorial on how to use `generate`.

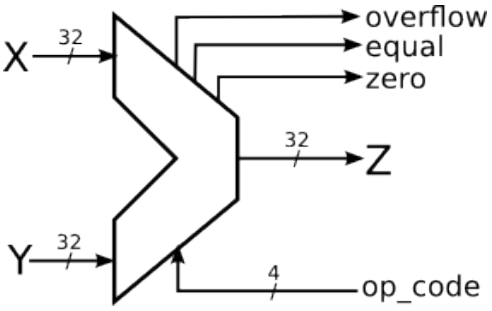
Remember that Verilog is just a tool we use to describe circuits - drawing out your circuits before writing a line of Verilog is the only reasonable way to approach this, and the deliverable requires that you submit your sketches of the circuits.

Get started early on this assignment - it's not easy!

Be sure to use your textbook as a reference! It describes how to do almost all of the ALU.

ALU Specification

The goal of this programming assignment is to build a 32 bit arithmetic logic unit (ALU). An ALU computes different things based on the inputs:



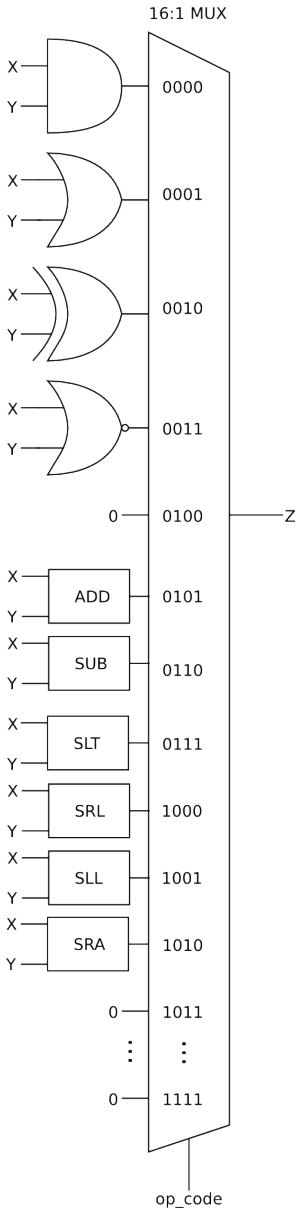
Make sure that your ports have the same naming convention as this diagram!

The output `Z` becomes different things based on the `op_code` input, according to the following table:

operation	`op_code`	description	points
`AND`	`4'b0000`	bitwise and	2
`OR`	`4'b0001`	bitwise or	2
`XOR`	`4'b0010`	bitwise xor	2
`NOR`	`4'b0011`	bitwise nor	2
reserved	`4'b0100`		
`ADD`	`4'b0101`	signed addition	6
`SUB`	`4'b0110`	signed subtraction	6
`SLT`	`4'b0111`	signed Set Less Than (`32'd1` if `X` is less than `Y`, `32'd0` otherwise)	4
`SRL`	`4'b1000`	shift right logical (shift `X` right by `Y` places)	6
`SLL`	`4'b1001`	shift left logical (shift `X` left by `Y` places)	6
`SRA`	`4'b1010`	shift right arithmetic (shift `X` right by `Y` places, but maintain the sign of `X`)	4
reserved	`4'b1011`		
reserved	`4'b1100`		
reserved	`4'b1101`		
reserved	`4'b1110`		
reserved	`4'b1111`		

If `X` and `Y` are ever equal, the equal bit should go high. If Z is ever all zeros, the zero bit should go high. For the `ADD` and `SUB` operations, the overflow bit should go high if you run out of space with 32 bit integers. All of the outputs should be zero if the `op_code` is one of the reserved categories.

We could design this ALU by considering an extremely large truth table with `X`, `Y`, and `op_code` as the inputs that drive the outputs. This quickly gets intractable, and a far more reasonable approach is to divide the problem into submodules, and then route the appropriate submodule's output to the input based on the `op_code`, like so:



We have 9 op codes to implement, so we have to use the next power of two (16) for the number of ports in our mux. If this seems wasteful, it's true - you could use an 8:1 mux and an extra 2:1 mux and save a lot of area. What you lose in area, you do gain in extensibility though - so it's not all a loss. Xilinx will also optimize away any unused wires/gates, so when you are first creating something, it's better design practice to design it a bit bigger than you need, and then to trim down when the design is proven to work. The op codes for this assignment have been deliberately chosen to make using a 16:1 mux unnecessary if you're clever, but it's far more important to have a working design than an optimized one.

Deliverable

Part 1 (due February 15th):

For the first part of this assignment, you should design and simulate the 16:1 multiplexer (mux) as well as the `AND`, `OR`, `XOR`, `NOR`, and `ADD` operations (`ADD` is more difficult!). Your mux should be parameterized so that the inputs/output buses can be of arbitrary size. Also make sure that you create testbenches to test functionality! Refer back to the Verilog cheatsheet as you complete this assignment to learn more about creating parameterized modules and writing testbenches.

Part 2 (due by February 22nd):

For this part of the assignment, you should complete the remaining ALU functions - `SUB`, `SLT`, `SRL`, `SLL`, and `SRA` - and make sure that your ALU has properly functioning zero, equal, and overflow bits. Also make sure that you create testbenches to test functionality!

Extra Credit: If you implement a carry-lookahead adder for part 2, you will receive up to 10 extra credit points. However, we strongly suggest that you stick with a simpler adder type for part 1!

Submit the following on Canvas for each part:

1. Your Xilinx project
2. Sketches of each of the main component circuits (SLT, SLL, SRL, ADD, SUB, SRA). Make sure any scans are legible and/or any pictures are clear.
3. A description (one or two paragraphs) of your simulation/testing methodology

Unzipping the compressed file should yield the following general directory structure (the individual file/folder names don't have to match what is written below, but be descriptive when appropriate):

`lab2_xilinx_project/` `sketches/` `description.doc` / `description.txt` / `description.pdf`

As you work on this assignment, remember that you should only have one module for each source file. This helps keep your code much more organized and modular. Instructions for adding new modules and testbenches can be found here. Also, please be sure to thoroughly comment your Verilog!

Here is the rubric we will be using for grading:

Component	Part 1 Points	Part 2 Points
Circuit Sketches	5	15
Simulation Description	2	6
16:1 mux	4	
Student testbench rigor (see below)	5	15
ALU's functional correctness under instructor testbench		40
Synthesis (works or not)	4	4
Extra Credit		10

Student testbench rigor

A portion of the grade will depend on the testbench you write to ensure that your Verilog code is correct. Please test at least one standard case and one special case for each op code (you should also consider testing many more cases otherwise your implementation may fail on our testbench). A special case is one where one of the flags is set (overflow, equal, or zero). You may want to create new tests

for each new gate that you create (for example a `test_adder.v`, `test_sll.v`, etc..)

Submission

You should submit the assignment to the page on Canvas. Please see the submission guidelines for how to prepare your project for submission.

Generate tutorial

If you are copying many lines that have a similar form, you may find it useful to use the `generate` statement, which specifies the generation of hardware objects. A simplified syntax for the `generate` statement is;

```
generate
    genvar [index_variables];
    for ([initial_assignment]; [expression]; [step_assignment])
    begin [: optional_label]
        [concurrent_constructs];
        ...;
    end
endgenerate
```

Here is an example to create a cascading xor circuit (which you saw in the last programming assignment can determine the odd parity of an input signal). Here is the schematic:

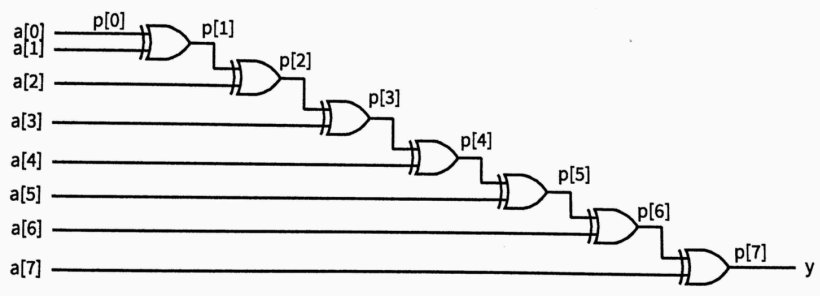


Figure from *FPGA Prototyping by SystemVerilog Examples* By Pong P. Chu.

Verilog has shorthand to do this with one operator but we will build the circuit from scratch for demonstration purposes.

```
module cascade_xor(a, y);
    input wire [7:0] a;
    output wire y;

    wire [7:0] p;

    assign p[0] = a[0];

    generate
        genvar i;
        for (i = 1; i < 8; i = i + 1) begin
```

```
            xor xor_gen (p[i], a[i], p[i-1]);
        end
    endgenerate

    assign y = p[7];
endmodule
```