Ahmed Abdulkareem
ECE 351 HW 3
Design Report

Introduction/Overview

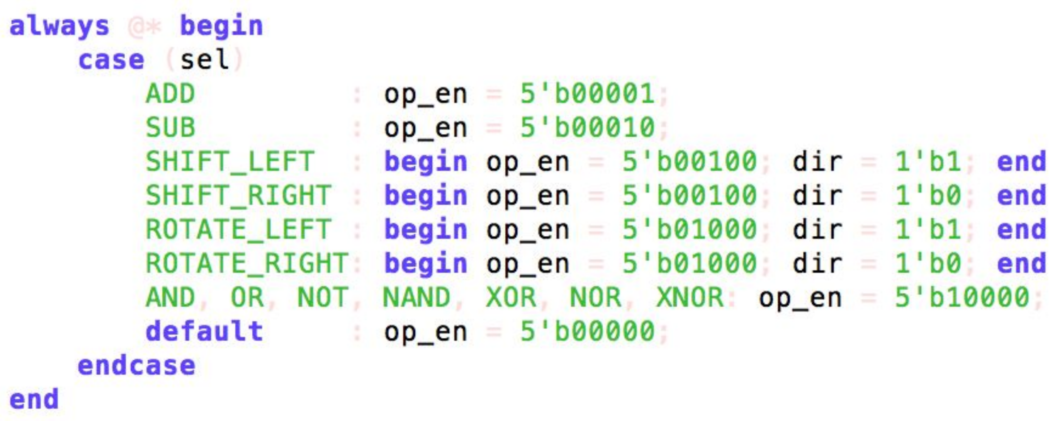
ALU (Arithmetic logic unit) exists in every CPU. This project involves designing a simple 8 bit ALU with basic operations. The operations are shown in the table 1. The ALU has 4 total inputs. 2 8-bit operands A, and B, 1-bit carry_in, and a 4-bit sel input which acts like a select input to select one of the operations in table 1. There are 2 outputs. A 1-bit carry_out output and an 8-bit Y which holds the results.

Operation	sel (select)
Add	0000
Subtract	0001
Left Shift	0010
Right Shift	0011
Rotate Left	0100
Rotate Right	0101
AND, OR, NOT, NAND, NOR, XOR, XNOR	0110, 0111, 1000, 1001, 1010, 1011, 1100

Table 1: ALU Operations

There are 4 units implemented in this ALU and they are:
adder_sub.v unit which does the addition and subtraction, **shift_unit.v** which does the shifting, **rotate_unit.v** which does the rotation, and **logical_unit.v** which does the 7 logical operations. Each of these units has an operation enable input to be asserted when the unit is to be used by the ALU. Also, each of these units has an output enable to be able to distinguish where the output came from. This is because we only have Y output, and we need to be able to assign it the right output. An Output enable of 0001 specifies that the output is from the adder_sub.v unit, 0010 specifies an output from the shift_unit.v, 0100, specifies an output from the rotate_unit.v, and 1000 specifies an output from the logical_unit.v. For shifting operations, the directions of the

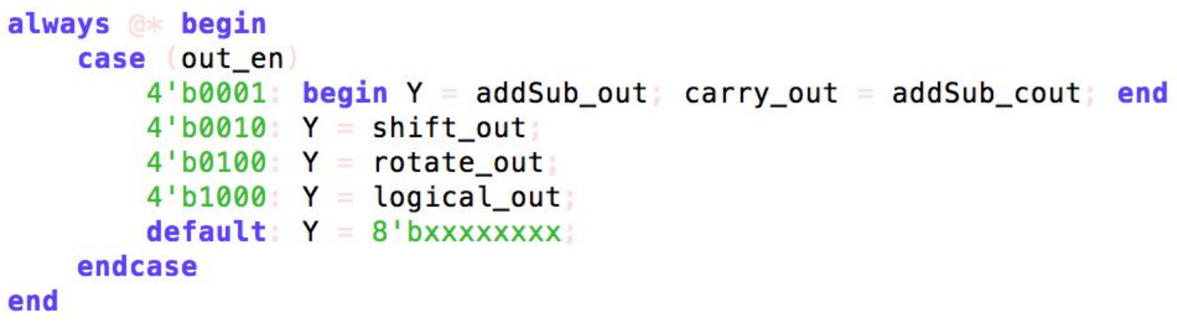
shifts are specified with a dir input where 1 specifies a left shift, and a 0 represents a right shift. Figure 1 and 2 make this clearer.



```
always @* begin
    case (sel)
        ADD          : op_en = 5'b00001;
        SUB          : op_en = 5'b00010;
        SHIFT_LEFT   : begin op_en = 5'b00100; dir = 1'b1; end
        SHIFT_RIGHT  : begin op_en = 5'b00100; dir = 1'b0; end
        ROTATE_LEFT  : begin op_en = 5'b01000; dir = 1'b1; end
        ROTATE_RIGHT: begin op_en = 5'b01000; dir = 1'b0; end
        AND, OR, NOT, NAND, XOR, NOR, XNOR: op_en = 5'b10000;
        default      : op_en = 5'b00000;
    endcase
end
```

Figure 1: Operation Enable Logic and Directions of Shifts

The op_en specifies what to do, For example, an op_en of 00001 turns on the adder, a 00100 turns on the shifter, etc.

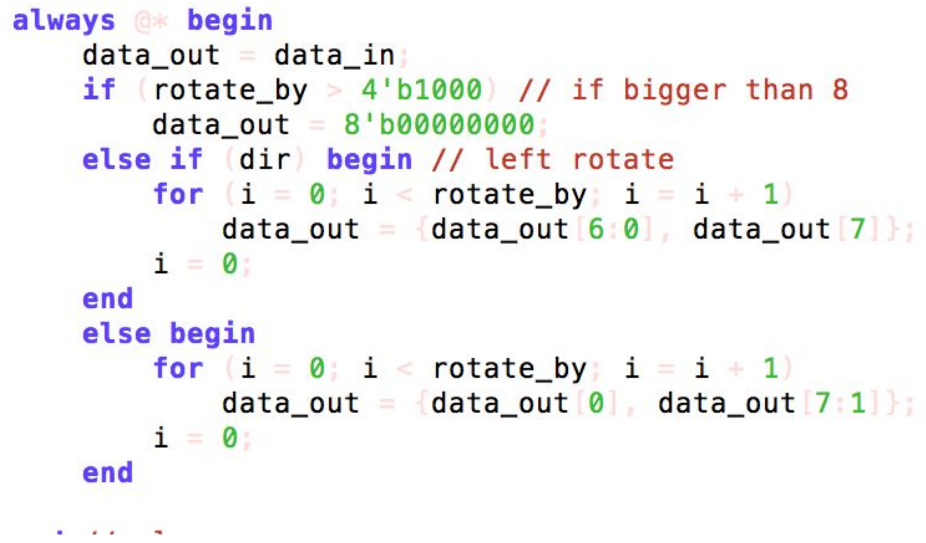


```
always @* begin
    case (out_en)
        4'b0001: begin Y = addSub_out; carry_out = addSub_cout; end
        4'b0010: Y = shift_out;
        4'b0100: Y = rotate_out;
        4'b1000: Y = logical_out;
        default: Y = 8'bxxxxxxxx;
    endcase
end
```

Figure 2: Output Logic

Above is where the output enable comes into play. It tells us what Y should be.

The ALU is designed using an algorithmic model. The adder_sub.v is designed by using the + and - operator. The shift_unit.v is designed with << >> operators. However, the rotate_unit.v is designed using a for loop and verilog bit concatenation. The for loop starts at 0 and goes up to 7 and rotates one bit a time. An example is shown in figure 3. Last but not least, the logical_unit.v is designed using the logical operators.



```
always @* begin
    data_out = data_in;
    if (rotate_by > 4'b1000) // if bigger than 8
        data_out = 8'b00000000;
    else if (dir) begin // left rotate
        for (i = 0; i < rotate_by; i = i + 1)
            data_out = {data_out[6:0], data_out[7]};
        i = 0;
    end
    else begin
        for (i = 0; i < rotate_by; i = i + 1)
            data_out = {data_out[0], data_out[7:1]};
        i = 0;
    end
```

Figure 3: rotate_unit.v implementation

Tests/Verification

Approach

This is an 8-bit ALU with a lot of operations, doing an exhaustive test bench would take a long time. Hence, a different approach is taken. The approach taken is to test the boundaries, and some test cases in between the boundaries. Because we have a lot of test cases, a self-checking test bench would be the best solution. A self-checking test bench is implemented with useful messages indicating if there's an error. The test bench checks all outputs and compares it with the correct output. If they don't match, a fail flag is set to indicate at least one test case failed, as well as display what went wrong. Let's start talking about each of the test cases generated for each unit.

Adder_sub.v: For the adder, the test cases generated for this one is pretty simple and basic. It starts with A = 0, and iterates B from 0 to 255 with a step increment of 15, then iterates A by 15 each time. This will check and make sure that the boundaries are working, as well as inputs in the middle. Of course, the same is done with the carry_in input. **For the subtraction,** it's a bit more complicated. It starts with A = 0 (and iterates it by 15 each time), then iterates B from 0 to A = B. This will generate data where A > B. Then, some addition data is generated where A < B to make sure we get a negative number. This technique makes sure that the boundaries are working, as well as any random input without doing an exhaustive test.

shift_unit.v: For both left and right shifts, the same technique used for the adder_sub.v unit is used. A starts at 0, then B iterates from 0 to 255 at a step of 15, then A increments by 15. This makes sure it works without doing an exhaustive test for all the input combinations.

rotate_unit.v: For both left and right rotates, the same technique used for the shift_unit.v is used here. Again, this makes sure that the module works properly without an exhaustive test.

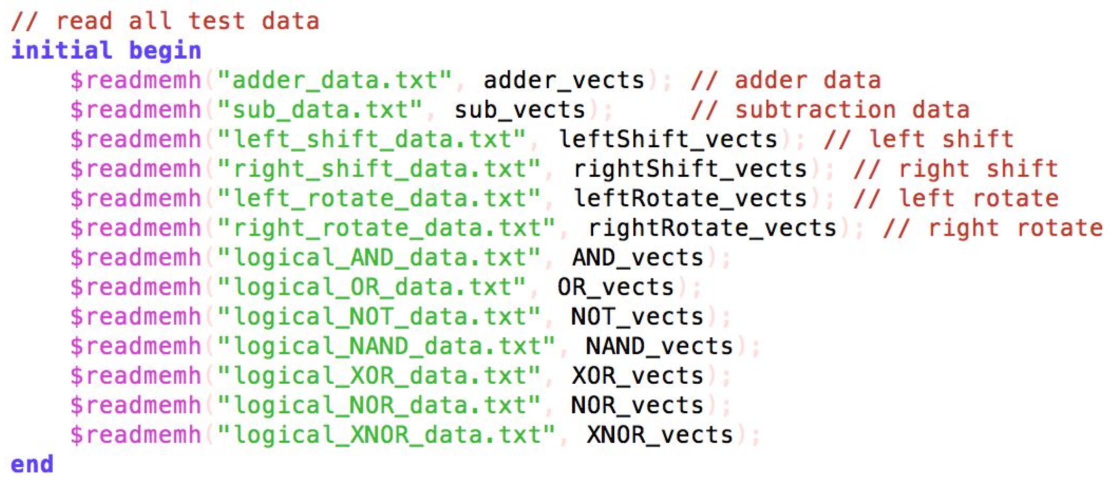
logical_unit.v: For this one, the boundaries are checked, as well as a few inputs in between. A starts at 0, then B goes from 0 to 255 in steps of 85, then A increments by 85 each time. This really takes very little time and also makes sure the module works properly.

How it's Done

For the test bench, there's a stimulus bench, which generates test vectors and checks the results from the alu block. Also, there is the tb.v which instantiates the stimulus block, and the ALU design.

stimulus.v

This file only feeds the ALU and checks the results. The test cases are generated externally by a file called generate_data.py. It reads all the data in an initial statements, then loops through them all and feed them to the ALU. If there's an error, it sets a flag reg to indicate that there's at least one test case that failed, and of course a message will be written to the console to show where the error occurred. Figure 2 shows a snapshot of the code where the files are loaded into memory.



```
// read all test data
initial begin
    $readmemh("adder_data.txt", adder_vects); // adder data
    $readmemh("sub_data.txt", sub_vects);     // subtraction data
    $readmemh("left_shift_data.txt", leftShift_vects); // left shift
    $readmemh("right_shift_data.txt", rightShift_vects); // right shift
    $readmemh("left_rotate_data.txt", leftRotate_vects); // left rotate
    $readmemh("right_rotate_data.txt", rightRotate_vects); // right rotate
    $readmemh("logical_AND_data.txt", AND_vects);
    $readmemh("logical_OR_data.txt", OR_vects);
    $readmemh("logical_NOT_data.txt", NOT_vects);
    $readmemh("logical_NAND_data.txt", NAND_vects);
    $readmemh("logical_XOR_data.txt", XOR_vects);
    $readmemh("logical_NOR_data.txt", NOR_vects);
    $readmemh("logical_XNOR_data.txt", XNOR_vects);
end
```

Figure 4: Reading all Test Vectors

generate_data.py

This is the python script that generates all the test vectors. The test vectors are generated and dumped into a .txt file with descriptive names. The format of these files are:

Operand_A operand_B Results

The first two are always operands and the results is always the third element. For example, for an addition operation with A = 5, B = 5, results = 10. In the file, it'd look like this:

5 5 A

Note that the numbers are in hex format. The operands are 8-bit operands and the result is 9-bit results. Also note that this file doesn't generate anything else but the operands and the results. For a NOT operation, the B operand would just be ignored.

Technical Problems

Not many technical problems were encountered, but there were some. One of the major ones was my self-checking test bench would fail even though the wave form was correct. For example, in an addition operation, where A = 5, and previous A = 4, B = 5, my result would be 9, because it'd be using the previous A value. I realized I needed a delay after setting A and B.
Another issue that's not very technical but it wasn't obvious why the tb.v file was needed and how to code it. However, I realized that in verilog, you can use the dot operator to access elements of a module, which solved the issue.