The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 541 Digital Logic and Computer Design
Spring 2015

Lab #3: Designing an ALU

Issued Wed 1/21/15; Due Wed 1/28/15 (11:59pm)

This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission instructions are at the end.

You will learn the following:

- Designing a hierarchical system, with much deeper levels of hierarchy
- Designing arithmetic and logic circuits
- Using `define and parameter constants
- Verilog simulation, test fixtures and stimuli

Part 0: Modify the Adder-Subtractor of Lab 2 to make it parameterizable

In order to make the number of bits in the operands of the adder customizable, a parameterized version of the ripple-carry adder is shown below.

```
module adder #(parameter N=32) (
    input [N-1:0] A, B,
    input Cin,
    output [N-1:0] Sum,
    output FlagN, FlagC, FlagV
    );

    wire [N:0] carry;
    assign carry[0]=Cin;

    assign FlagN = Sum[N-1];
    assign FlagC = carry[N];
    assign FlagV = carry[N] ^ carry[N-1];

    fulladder a[N-1:0] (A, B, carry[N-1:0], Sum, carry[N:1]);

endmodule
```

By simply changing the parameter N, the width of the adder can be easily changed. The value "32" is specified as the default value of N, but this is overridden by the value of the parameter when this module is instantiated in the enclosing module. Also note that new outputs have been added to generate the *Negative, Carryout* and *Overflow* flags. The enclosing Adder-Subtractor unit is also accordingly modified, as shown:

```
module addsub #(parameter N=32) (
    input [N-1:0] A, B,
    input Subtract,
    output [N-1:0] Result,
    output FlagN, FlagC, FlagV
    );

    wire [N-1:0] ToBornottoB = {N{Subtract}} ^ B;
    adder #(N) add(A, ToBornottoB, Subtract, Result, FlagN, FlagC, FlagV);

endmodule
```

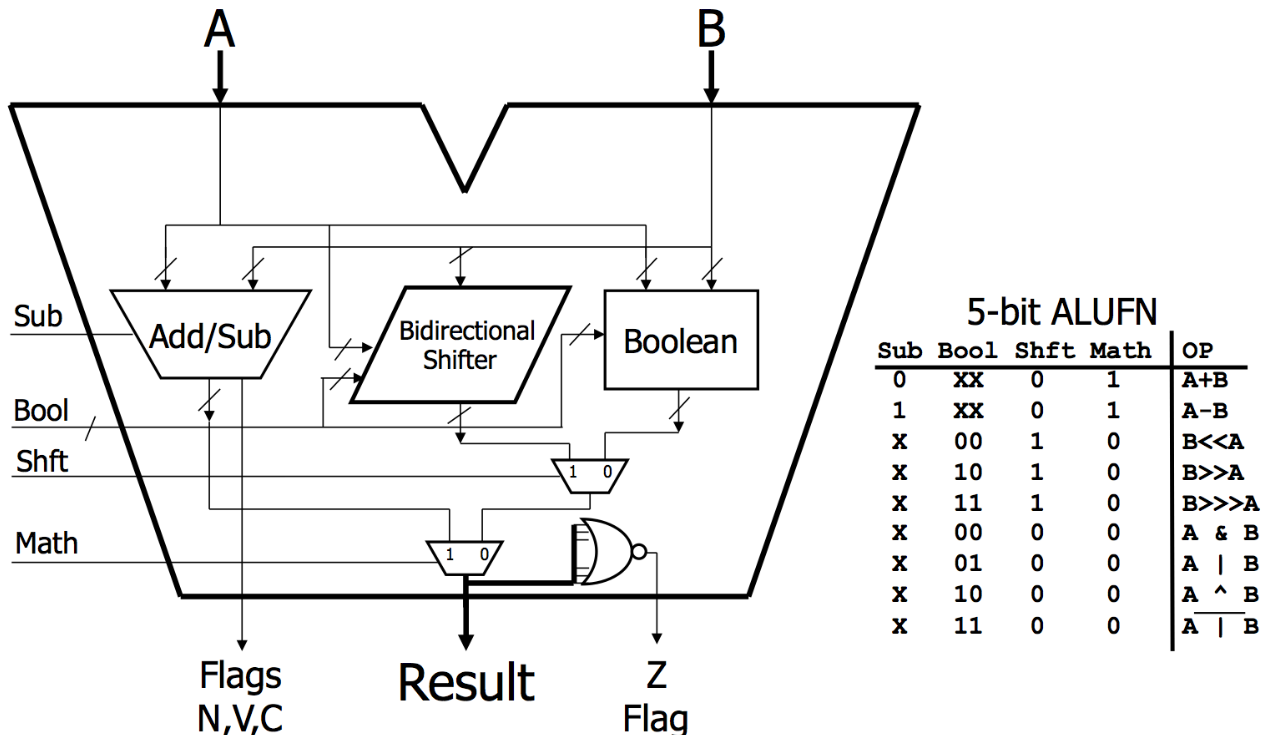
Once again, the Adder-Subtractor is parameterized, with a default width of 32 bits. This width parameter, N, is passed into the enclosed object, *adder*. Thus, changing the value of N in the top line of the module *addsub* to, say, 32 automatically passes the parameter value 32 to the adder module named *add*. The test fixture from Lab 2 has been modified to test these two modules thoroughly. *Tip:* In the text fixture, where you declare the unit under test, *uut,* you can set the parameter value as follows:

```
addsub #(width) uut (…)
```

The tester uses a 32-bit width. You should try other widths (e.g., 8 bits) as well. You will need to set the width only *once,* where *uut* is declared.

Part I: Understand the ALU structure and operation

Below is a block diagram of the ALU (from Comp411), along with its 5-bit control. More details are available in the slides (from Comp411) available on the website. Your task is simply to review this information carefully and make sure you understand how the 5-bit control signal encodes the operation, and how the multiplexers select the result. Note: Comparison operations will be incorporated in Part IV.



Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

Part II: Logical and Shifter modules

Use the following code templates to complete the design of the Boolean logic unit and the bidirectional shifter unit. Put the logical module in a file named `logical.v`, and the put the shifter in a file named `shifter.v`.

```
module logical #(parameter N=32) (
    input [N-1:0] A, B,
    input [1:0] op,
    output [N-1:0] R
    );

   assign R =      (op == 2'b00) ? [redacted] :
                   (op == [redacted]) ? [redacted] :
                   (op == [redacted]) ? [redacted] :
                   (op == [redacted]) ? [redacted] : [redacted]
endmodule
```

You are to use the bitwise Verilog operators corresponding to the four logical operations listed in the table for the 5-bit control above.

Debugging Note: The final "else" clause in the code template above is not really necessary. You could modify the code above to eliminate the final if-else construct, or you could keep it and use it as a debugging aid. In particular, let's say that there was an error in the encoding of an instruction, and the 2-bit `op` received was 1x. This value will not match any of the four cases, and therefore default to final "else" clause. By assigning a particular "catch-all" value to this situation may help you later on. In more detail, say your catch-all value is all 1's (which could be written as {N{1'b1}}), then later on when you are simulating an entire MIPS processor, you find that the ALU is producing an unexpected result of all 1's, that could help you narrow the problem down to one of an invalid `op` value.

```
module shifter #(parameter N=32) (
    input signed [N-1:0] IN,
    input [$clog2(N)-1:0] shamt,     // ceiling log base 2
    input left, input logical,
    output [N-1:0] OUT
    );

   assign OUT = left ? ([redacted]) :
                (logical ? [redacted] : [redacted]);

endmodule
```

Observe IN is signed. Also, the maximum shift amount is *N,* so the number of bits in *shamt* is ceiling($\log_2(N)$).

You are to use Verilog operators for the three types of shift operations. Observe that the data type of IN is declared to be *signed.* Why? By default most types in Verilog are unsigned. However, for arithmetic-right-shift to operate correctly, the input must be declared to be of *signed* type, so that its leftmost bit is considered to indicate its sign. Also note that *shamt* only has ceiling($\log_2(N)$) bits (e.g., 5 bits for 32-bit operands).

For each of these modules, you should visualize their structure by creating and viewing their schematics.

Part III: ALU module (without comparisons)

Use the following code template to design an ALU that can add, subtract, shift, and perform logical operations.

```
module ALU #(parameter N=32) (
    input [N-1:0] A, B,
    output [N-1:0] R,
    input [4:0] ALUfn,
    output FlagN, FlagC, FlagV, FlagZ
    );

   wire subtract, bool1, bool0, shft, math;
   assign {subtract, bool1, bool0, shft, math} = ALUfn[4:0];      // Separate ALUfn into named bits

   wire [N-1:0] addsubResult, shiftResult, logicalResult;         // Results from the three ALU components

   addsub #(N) AS(A, B, [redacted], [redacted], [redacted], [redacted], [redacted];
   shifter #(N) S(B, A[4:0], [redacted], [redacted], [redacted]);
   logical #(N) L(A, B, {[redacted], [redacted]}, [redacted]);

   assign R =  (~shft & math)? [redacted] :                       // 3-way multiplexer to select result
               (shft & ~math)? [redacted] :
               (~shft & ~math)? [redacted] : 0;

   assign FlagZ = [redacted];                                     // Use a reduction operator here

endmodule
```

TIP: Be careful in providing the correct `left` and `logical` inputs to the shifter from within the ALU module. Observe first which values of `bool1` and `bool0` represent *sll*, *srl* and *sra* operations. Then determine how `left` and `logical` should be generated from `bool1` and `bool0`. This is slightly tricky!

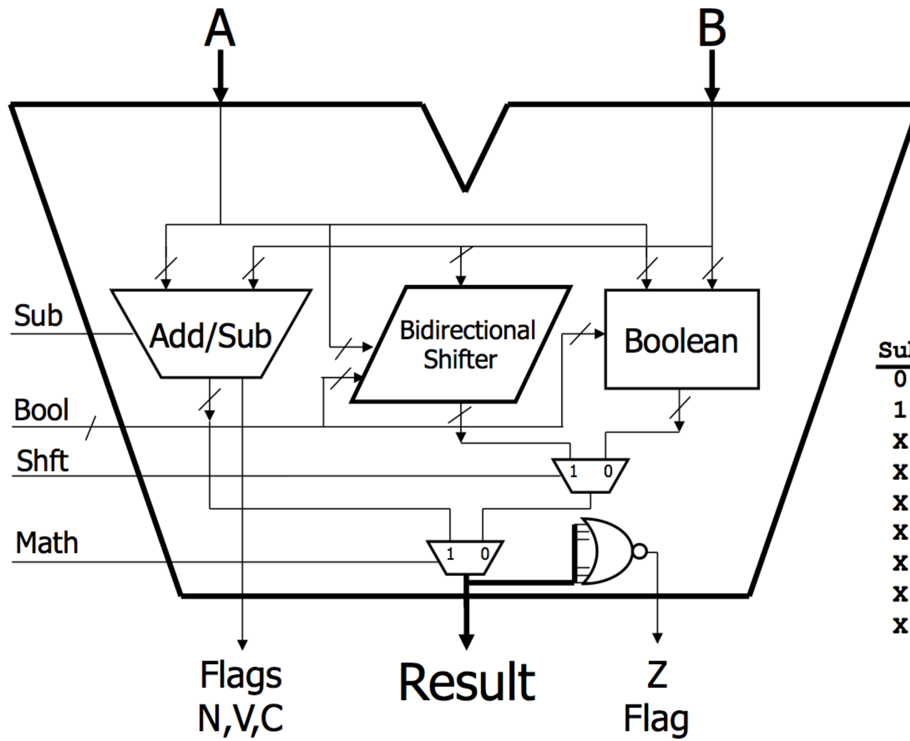
Use the test fixture provided on the website to test the full ALU. Please select "signed decimal" as the radix to display the inputs A and B, and the output R. Please select "binary" as the radix for the ALUfn.

Part IV: Modify the ALU of Part III to include comparisons

Below are two block diagrams of the ALU: the one you implemented in Part III, and a modified one that you are to implemented now. The latter includes additional functionality: to compare the operands A and B. Both *signed* and *unsigned* comparisons are implemented: *less-than signed* (LT) and *less-than-unsigned* (LTU). Note the differences between the two ALUs (new functionality is highlighted in red). You may refer to the slides (from Comp411) available on the Comp541 website. Review this information carefully before proceeding.

The comparison between the two operands, *A* and *B,* is performed by doing a subtraction (*A-B*) and then checking the flags generated (*N, V* and *C*). Observe that the *Sub* bit of the ALUFN is therefore on. The lower bit of *Bool* determines whether the comparison is signed or unsigned. When comparing *unsigned* numbers, the result of (*A-B*) is negative if and only if the leftmost carry out of the adder-subtractor (i.e., the *C* flag) is '0'. There cannot be an overflow when two positive numbers are subtracted. But when the numbers being compared are *signed* (i.e., in 2's-complement notation), then the result of (*A-B*) is negative if (i) either the result has its negative bit (i.e., *N* flag) set *and* there was no overflow; or (ii) the result is positive and there was an overflow (i.e., *V* flag set). For more details, you may refer to the slides (from Comp411) on the website.

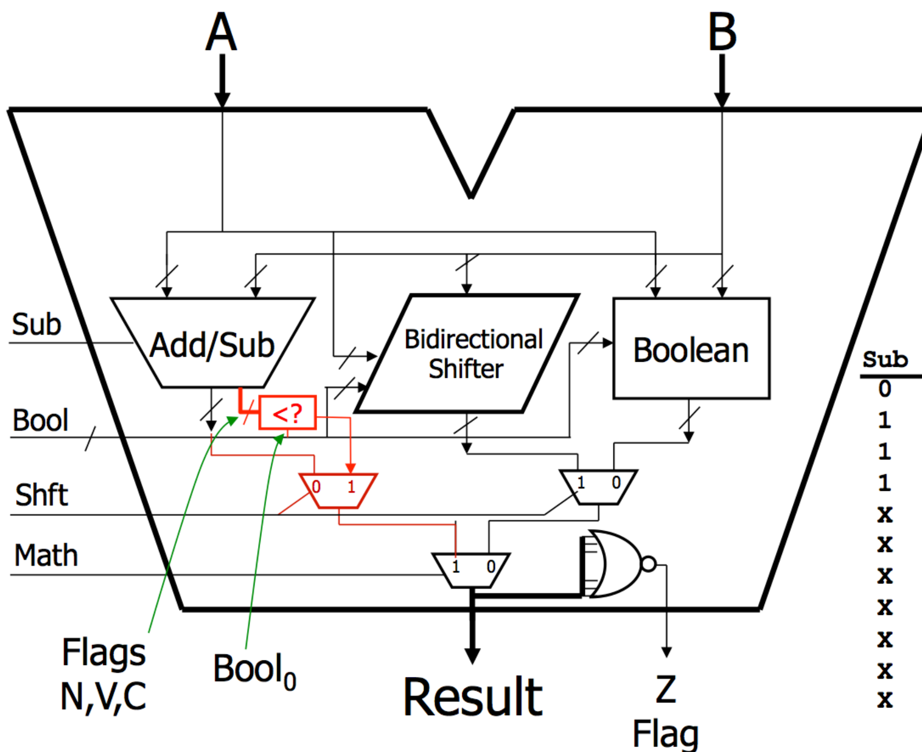
Without Comparisons:



5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{\text{A \| B}}$

With Comparisons:



5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
1	X0	1	1	A LT B
1	X1	1	1	A LTU B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	A | B

To implement the new ALU, first make a new module called *comparator* in a new Verilog file named `comparator.v`. Here is a code skeleton:

```
module comparator(input FlagN, FlagV, FlagC, bool0, output comparison)
      assign comparison = ………
endmodule
```

Next, modify the ALU module to include an instance of the comparator you just designed, and then modify the **`assign R`** line to make it a 4-way multiplexer instead of the original 3-way multiplexer. (*Tip:* The very last "else" case in the nested conditional assignment in Part III could be used now to handle the result of the comparator!)

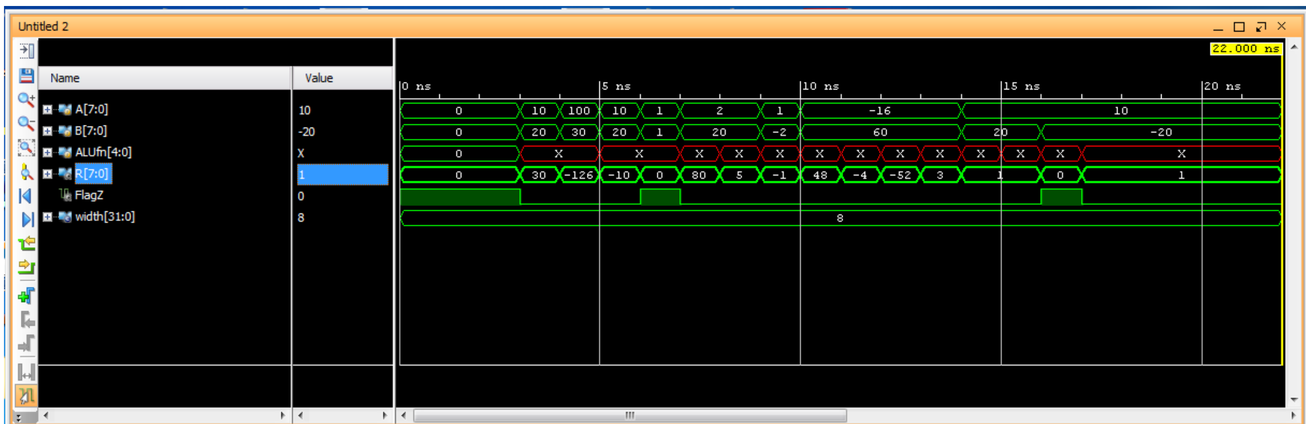
Keep in mind that the result of the comparator is a single bit '0' or '1', but the ALU's result is multibit. To avoid a compiler warning here, you will have to pad the comparator result with (*N*-1) zeros to its left. The Verilog expression for doing so is:

```
{{(N-1){1'b0}}, compResult}
```

The `{(N-1){1'b0}}` part replicates a 1-bit zero *N*-1 times, and then the 1-bit result of the comparator is concatenated to it.

Finally, eliminate the flags N, V and C from the output of the ALU. These flags are only used for comparison instructions in our version of the MIPS, and since the comparator has now been included inside the ALU, these three flags are not needed outside the ALU. The flag Z, however, still needs to be an output (since it will be used by the control unit later on for `beq`/`bne` instructions).

<u>*Testing:*</u> Use the text fixture provided on the website to test your ALU. Please select "signed decimal" as the radix to display the inputs A and B, and the output R. Please select "binary" as the radix for the ALUfn. You should first test your ALU with the width set to 8 bits. You should see exactly these waveforms:



Next, you should try to change the width to other values (e.g., 32 bits), and modify the test fixture accordingly, and verify that the ALU works correctly. Some results will change, e.g., some operations (100 + 30) which caused an overflow for 8 bits may not cause an overflow for 32 bits. You only need to submit your work for 8 bits.

Schematic: Create and view the schematic of your ALU design. Zoom in one or two levels down the hierarchy and try to relate the schematic generated with the structure of these components as discussed in class.

What to submit:

- **A screenshot of the simulation waveforms clearly showing the final simulation result for PART IV.**
- **Your Verilog source for the following modules from Part IV: the ALU, comparator, logical and shifter modules.**

How to submit: **Please submit your work by email as follows:**

- **Send email to:** `comp541submit-cs@cs.unc.edu`
- **Attach the following Verilog files:** `alu.v`, `comparator.v`, `logical.v`, `shifter.v`.
- **Attach the simulator screenshot using the filename** `waveforms.png` **(or other appropriate extension)**