Comp 541 Digital Logic and Computer Design
Prof. Montek Singh
Fall 2021

Lab #2B: Designing an ALU

Issued Wed 8/25/21; Due Wed 9/1/21 (11:55pm, via Sakai)

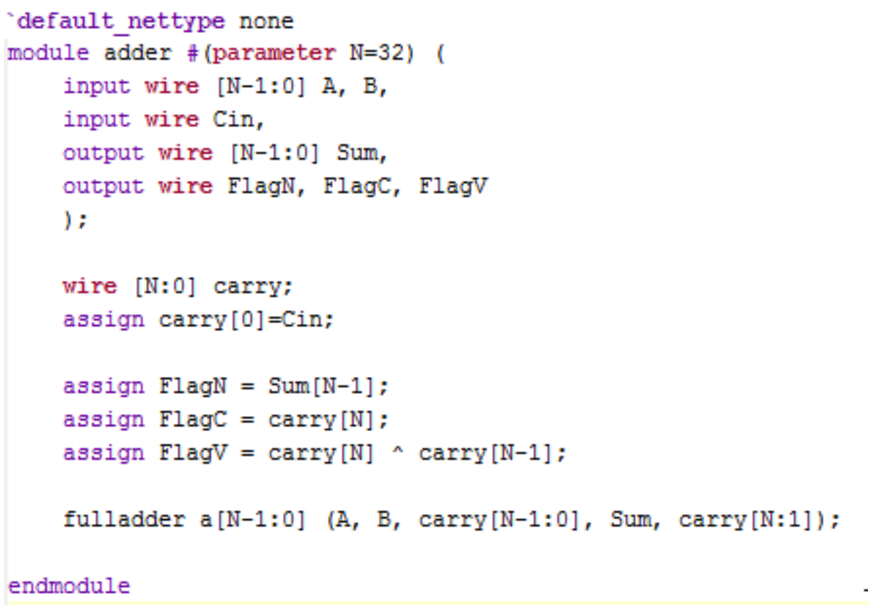
This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission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

Part 0: Modify the Adder-Subtractor of Lab 2A 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. They keyword `parameter` is used to specify a constant with a default value, but this value can be overridden while declaring an instance of the module within the *parent* module.



```
`default_nettype none
module adder #(parameter N=32) (
    input wire [N-1:0] A, B,
    input wire Cin,
    output wire [N-1:0] Sum,
    output wire 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 specified 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:

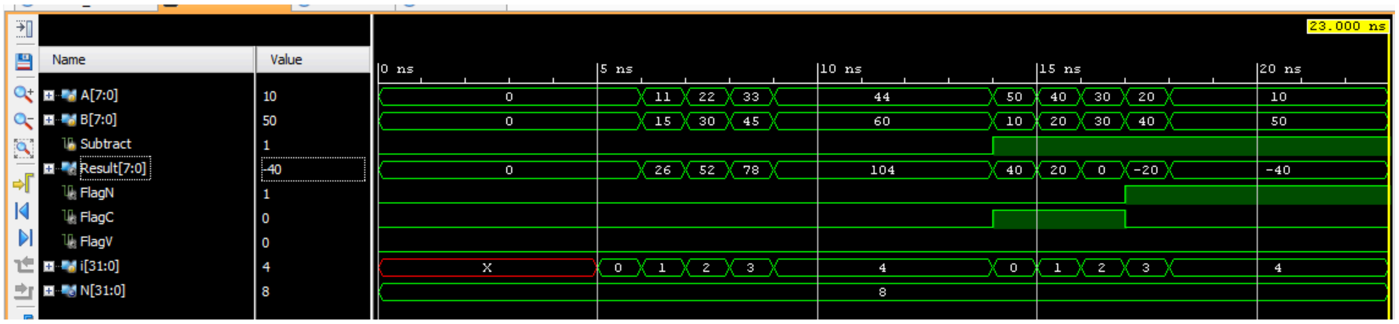
```
`default_nettype none
module addsub #(parameter N=32) (
    input wire [N-1:0] A, B,
    input wire Subtract,
    output wire [N-1:0] Result,
    output wire FlagN, FlagC, FlagV
    );

    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, 64 automatically passes the parameter value 64 to the adder module named *add.* The test fixture from Lab 2A has been modified to test these two modules. *Tip:* In the text fixture, where you declare the unit under test, *uut,* you can set the parameter value as follows:

```
            addsub #(width) uut (…)
```

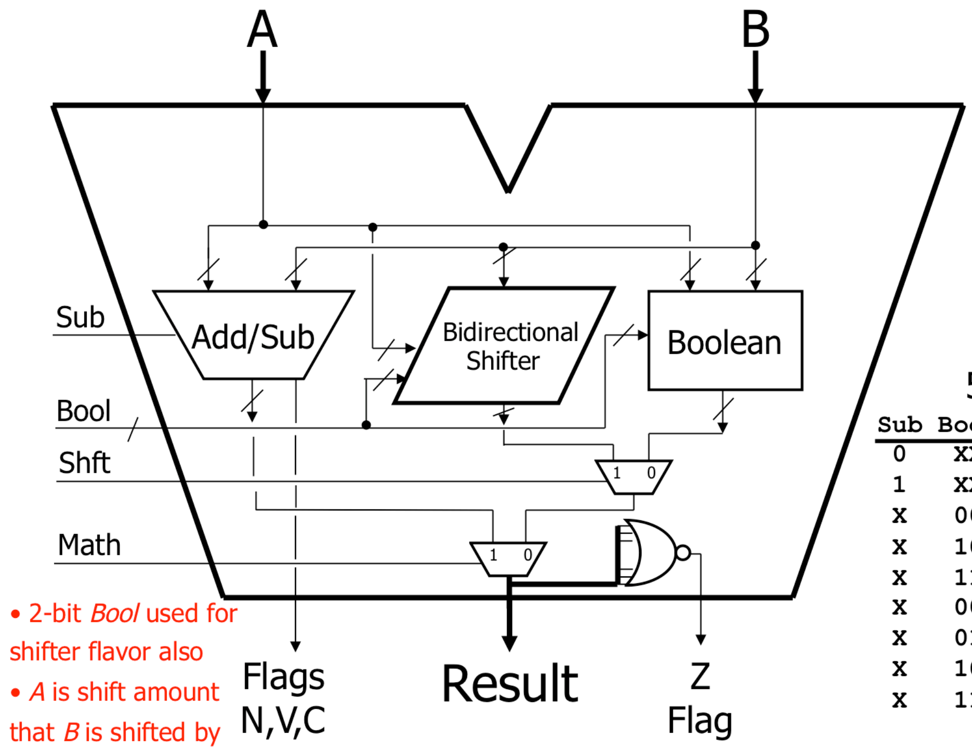
The tester provided uses an 8-bit width. You should try other widths (e.g., 32 bits) as well, by simply changing the `localparam` *N* within the tester. A `localparam` is another form of constant declaration, but one which is *local* to the module, i.e., not visible or changeable by its parent. Look at the tester to see how a `localparam` is declared and use.

Observe that you will need to set the width only *once,* where *uut* is declared, which overrides the default values of the width as it is passed down the hierarchy. You simulation output should look like this (once you have set up the display radix to signed decimal):



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.


A
B
Sub
Bool
Shft
Math
Add/Sub
Bidirectional Shifter
Boolean
1 0
1 0
• 2-bit *Bool* used for shifter flavor also
• *A* is shift amount that *B* is shifted by
Flags N,V,C
Result
Z Flag

X means "don't-care". That bit does not matter.

5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	X	1	A+B
1	XX	X	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 and B
X	01	0	0	A or B
X	10	0	0	A xor B
X	11	0	0	A nor 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. Put the logical module in a file named `logical.sv`, and the put the shifter in a file named `shifter.sv`.

```
`default_nettype none
module logical #(parameter N=32) (
    input wire [N-1:0] A, B,
    input wire [1:0] op,
    output wire [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 "? :" construct in the code template above is not necessary. You could modify the code above to eliminate the final "? :", 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. Assigning a particular "catch-all" value to this situation may help you later on. For example, say your catch-all value is all 1's (which could be written as {N{1'b1}}). Later on when you are simulating an entire MIPS processor, if you find that the ALU output is an unexpected pattern of all 1's, that could help you narrow the problem down to an invalid `op` value.

```
`default_nettype none
module shifter #(parameter N=32) (
    input wire signed [N-1:0] IN,
    input wire [$clog2(N)-1:0] shamt,     // ceiling log base 2
    input wire left, logical,
    output wire [N-1:0] OUT
    );

   assign OUT = left ? (IN << shamt) :
                  (logical ? IN >> shamt : IN >>> shamt);

endmodule
```

Observe IN is **signed.** Also, the shift amount can range from 0 to *N*-1, so the number of bits in *shamt* is ceiling(log$_2$(*N*)), written as **`$clog2(N)`**.

<< is shift left logical
>> is shift **right logical**
>>> is shift **right arithmetic**

Note the meaning of the control signals *left* and *logical*. If *left* is true, the type of shift performed is left shift (which is always logical). Otherwise, right shift is performed. For right shifts, if *logical* is true, then shift right logical is performed; else shift right arithmetic is performed.

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.

NOTE: Compared with Verilog, some other languages (e.g., java, python) swap the ">>" and ">>>" symbols.

Part III: ALU module (without comparisons)

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

```
`default_nettype none
module ALU #(parameter N=32) (
    input wire [N-1:0] A, B,
    output wire [N-1:0] R,
    input wire [4:0] ALUfn,
    output wire 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[$clog2(N)-1:0], [redacted], [redacted], [redacted]);
   logical #(N) L(A, B, {[redacted] [redacted]}, [redacted]);

   assign R =  (~shft & math)?[redacted] :                        // 4-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 by carefully reviewing the table of control values for the ALU shown in Part I. Then determine how `left` and `logical` should be generated from `bool1` and `bool0`. This is slightly tricky! (Do not modify the shifter itself to change the meaning of its `left` and `logical` inputs!)

A systematic method to determine the relationship between `bool1/bool0` and `left/logical` is to complete the following truth table first:

Inputs		Outputs	
bool1	*bool0*	*left*	*logical*
0	0	T	T
0	1	T	F
1	0	F	T
1	1	F	F

Write Boolean equations:

$left =$ $\boxed{}$

$logical =$ $\boxed{}$

Then, write out one Boolean equation for `left`, and one for `logical`, in terms of `bool1` and `bool0`. Use the expressions for `left` and `logical` you just developed to complete the shifter instantiation.

Finally, concatenate the two relevant control signals using "{ }" to produce the 2-bit control signal for the logical unit. 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

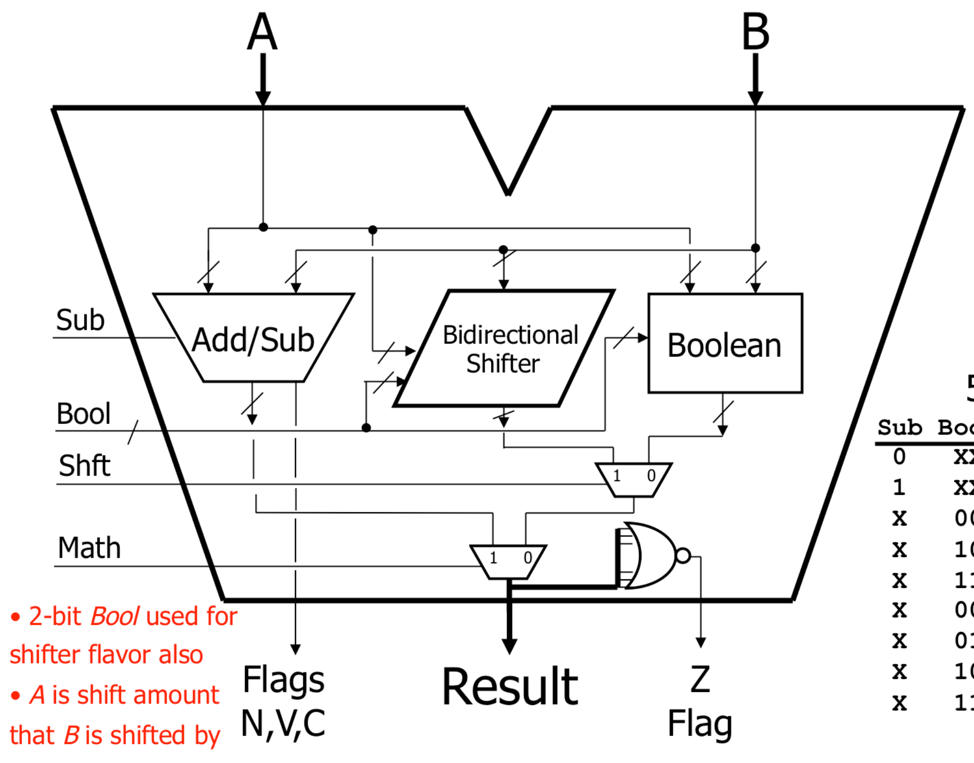
Below are two block diagrams of the ALU: the one you implemented in Part III, and a modified one that you are to implemented now. The latter includes additional functionality: to compare the operands A and B. Both *signed* and *unsigned* comparisons are implemented: *less-than signed* (LT) and *less-than-unsigned* (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.

The next page shows two diagrams. The first is the ALU without support for comparisons that you just implemented in Part III (repeated for convenience). The second is the complete ALU that includes the functionality for comparisons. Carefully observe the difference between the two.

(see next page)

ALU from Part III (without comparisons):

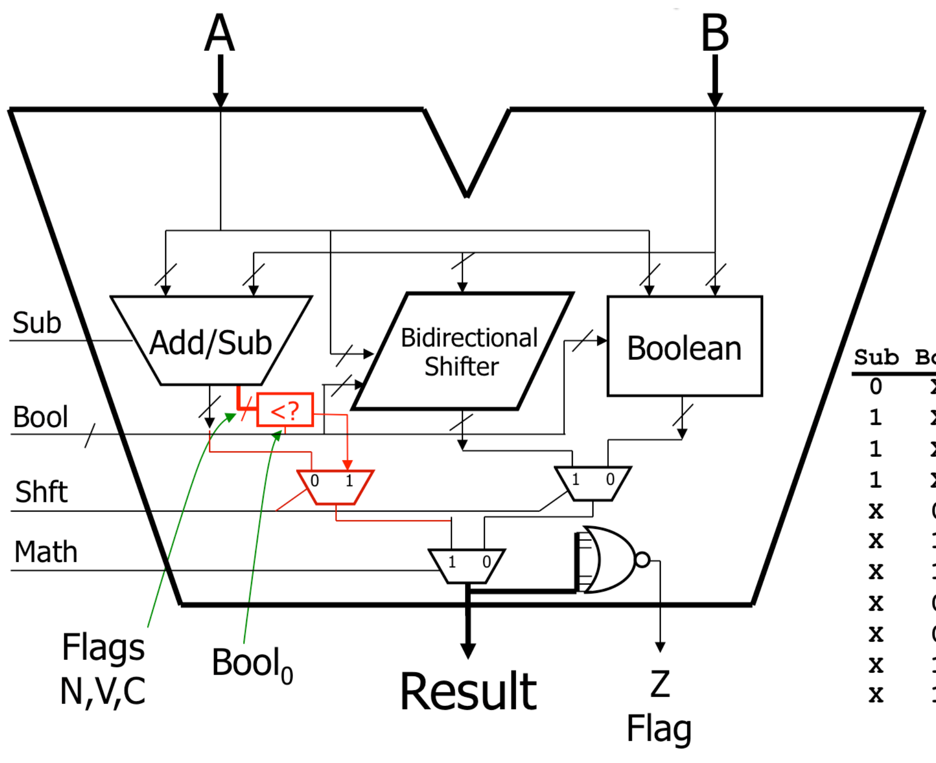

A
B
Sub
Bool
Shft
Math
Add/Sub
Bidirectional Shifter
Boolean
1 0
1 0
• 2-bit *Bool* used for shifter flavor also
• *A* is shift amount that *B* is shifted by
Flags N,V,C
Result
Z Flag

X means "don't-care". That bit does not matter.

5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	X	1	A+B
1	XX	X	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 and B
X	01	0	0	A or B
X	10	0	0	A xor B
X	11	0	0	A nor B

Complete ALU (with comparisons):



X means "don't-care". That bit does not matter.

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 and B
X	01	0	0	A or B
X	10	0	0	A xor B
X	11	0	0	A nor B

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

```
`default_nettype none
module comparator(
   input wire FlagN, FlagV, FlagC, bool0,
   output wire comparison
   );

assign comparison = [redacted]

endmodule
```

Observe that the value of `bool0` determines whether a signed comparison is performed or an unsigned comparison (see ALUFN table next to the diagram). Accordingly, write a conditional expression (using "? … :") in the above code snippet to check `bool0`, and then choose the correct Boolean formula for the appropriate flavor of comparison (refer to the writeup at the beginning of this section, as well as the Powerpoint slides, for how to perform comparisons).

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 last "else" case in the nested conditional assignment in Part III can be used now to handle the result of the comparator!)

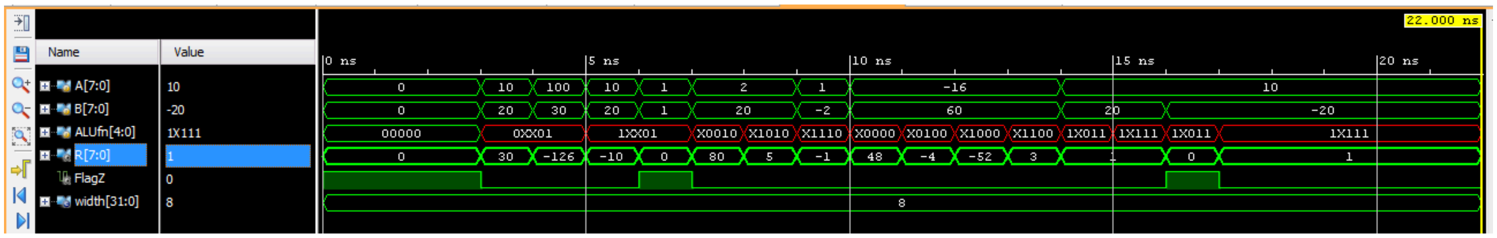
Keep in mind that the result of the comparator is a single bit '0' or '1', but the ALU's result is multibit. To avoid leaving all the higher-order result bits *undefined*, you must pad the 1-bit comparator result with (*N*-1) zeros to its left. This "zero extension" must be done *inside the ALU module,* not inside the comparator. The Verilog expression for doing so is:

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

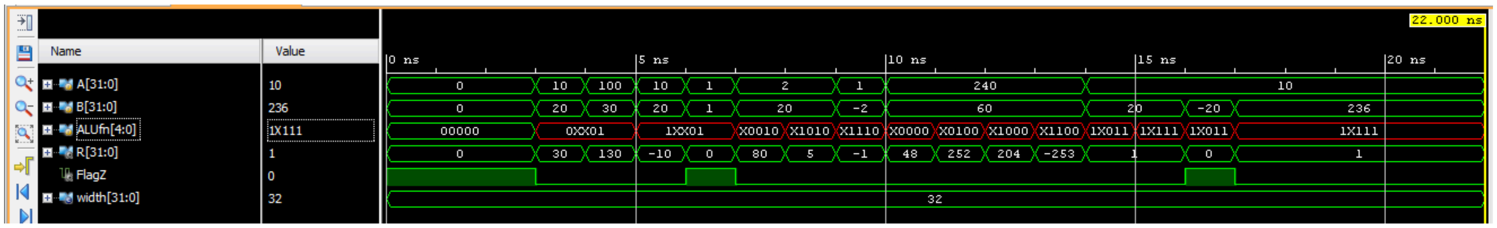
The `{(N-1){1'b0}}` part replicates a 1-bit zero *N*-1 times, and then the 1-bit result of the comparator is concatenated to it.

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 fixtures provided on the website to test your ALU (both 8-bit and 32-bit versions). 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 (use *Lab2a_Part4_test_8bit.sv*). You should see exactly these waveforms:



Next, test your design using the 32-bit version of the tester (*Lab2a_Part4_test_32bit.sv*). Some results will change, e.g., the operation (100 + 30), which caused an overflow for 8 bits will not cause an overflow for 32 bits. The simulation output for a 32-bit design is shown below. Observe carefully all the differences between the 8-bit output and the 32-bit output, and see if you can explain them.



Schematic: Close the simulation, and create and view the schematic of your ALU design. Take a screenshot of the schematic to attach to the submission. Zoom in one level down the hierarchy and try to relate the schematic generated with the structure of these components as discussed in class. You will learn a lot by relating the circuit schematic to the Verilog description, and trying to find correspondences between circuit components and lines of Verilog.

What to submit:

- **A screenshot of the simulation waveforms clearly showing the <u>final simulation results for PART IV only (both 8-bit and 32-bit versions).</u>**
- **Your Verilog source for the following modules from Part IV: the ALU, comparator, logical and shifter modules.**
- **A picture of your circuit schematic for the final 32-bit design (top level only) from Part IV.**

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

- **Submit via Sakai ("Lab 2B" under Assignments).**
- **Attach the following Verilog files: `alu.sv`, `comparator.sv`, `logical.sv`, `shifter.sv`.**
- **Attach the simulator screenshots using the filenames `waveforms8bit.png` and `waveforms32bit.png`, and the schematic using the filename `schematic.png` (or other appropriate names and /or extensions).**
- **Submit your work by 11:55pm on Wednesday, Sep 1.**
