ISTD 50.002 Computation Structures

Lab #3 Arithmetic Logic Unit

Revamped by: Kenny Choo, Natalie Agus, Oka Kurniawan 2020.

Introduction

In this lab, we will build the ***arithmetic and logic unit*** **(ALU)** for the Beta processor. The ALU is a **combinational logic device** that has two 32-bit inputs (which we will call “A” and “B”) and produces one 32-bit output. We will start by designing each piece of the ALU as a separate circuit, each producing its own 32-bit output. We will then combine these outputs into a single ALU result.

Due Date

Refer to the course handout for due date.

Optimising Circuitry

When designing circuits there are three separate factors that can be optimised:

1. Design for **maximum** performance (minimum latency)
2. Design for **minimum** cost (minimum area)
3. Design for the **best** cost / performance **ratio** (minimise area * latency)

It is often possible to do all three at once but in some portions of the circuit some sort of *design tradeoff* will need to be made. When designing your circuitry you should choose which of these three factors is most important to you and optimize your design accordingly.

A functional ALU design will earn six points. Four additional points can be earned if you implement the multiplier unit – see the section labeled "Design Problem: Implementing Multiply" for details.

Standard Cell Library

The building blocks for our design will be a family of logic gates that are part of a standard cell library. The available combinational gates are listed in the table below along with information about their timing, loading and size. You can access the library by starting your netlist with the following include statements:

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
```

Everyone should use the provided cells in creating their design. The timings have been taken from a 0.18 micron CMOS process measured at room temperature.

Netlist	**Function**	**t_{CD} (ns)**	**t_{PD} (ns)**	**t_R (ns/pf)**	**t_F (ns/pf)**	**load (pf)**	**size (μm^2)**
Xid z constant0	$Z = 0$	—	—	—	—	—	0
Xid z constant1	$Z = 1$	—	—	—	—	—	0
Xid a z inverter	$Z = \overline{A}$	.005	.02	2.3	1.2	.007	10
Xid a z inverter_2		.009	.02	1.1	.6	.013	13
Xid a z inverter_4		.009	.02	.56	.3	.027	20
Xid a z inverter_8		.02	.11	.28	.15	.009	56
Xid a z buffer	$Z = A$	.02	.08	2.2	1.2	.003	13
Xid a z buffer_2		.02	.07	1.1	.6	.005	17
Xid a z buffer_4		.02	.07	.56	.3	.01	30

Xid a z buffer-8		.02	.07	.28	.15	.02	43
Xid e a z tristate	$Z = A$ when e=1 else Z not driven	.03	.15	2.3	1.3	.004	23
Xid e a z tristate_2		.03	.13	1.1	.6	.006	30
Xid e a z tristate_4		.02	.12	.6	.3	.011	40
Xid e a z tristate_8		.02	.11	.3	.17	.02	56
Xid a b z and2	$Z = A \cdot B$	.03	.12	4.5	2.3	.002	13
Xid a b c z and3	$Z = A \cdot B \cdot C$	.03	.15	4.5	2.6	.002	17
Xid a b c d z and4	$Z = A \cdot B \cdot C \cdot D$	.03	.16	4.5	2.5	.002	20
Xid a b z nand2	$Z = \overline{A \cdot B}$	.01	.03	4.5	2.8	.004	10
Xid a b c z nand3	$Z = \overline{A \cdot B \cdot C}$	.01	.05	4.2	3.0	.005	13
Xid a b c d z nand4	$Z = \overline{A \cdot B \cdot C \cdot D}$	.01	.07	4.4	3.5	.005	17
Xid a b z or2	$Z = A + B$	.03	.15	4.5	2.5	.002	13
Xid a b c z or3	$Z = A + B + C$	.04	.21	4.5	2.5	.003	17
Xid a b c d z or4	$Z = A + B + C + D$	.06	.29	4.5	2.6	.003	20
Xid a b z nor2	$Z = \overline{A + B}$	.01	.05	6.7	2.4	.004	10
Xid a b c z nor3	$Z = \overline{A + B + C}$	.02	.08	8.5	2.4	.005	13
Xid a b c d z nor4	$Z = \overline{A + B + C + D}$	.02	.12	9.5	2.4	.005	20
Xid a b z xor2	$Z = A \oplus B$	.03	.14	4.5	2.5	.006	27
Xid a b z xnor2	$Z = \overline{A \oplus B}$	.03	.14	4.5	2.5	.006	27
Xid a1 a2 b z aoi21	$Z = \overline{(A1 \cdot A2) + B}$	.02	.07	6.8	2.7	.005	13
Xid a1 a2 b z oai21	$Z = \overline{(A1 + A2) \cdot B}$	.02	.07	6.7	2.7	.005	17
Xid s d0 d1 z mux2	Z = D0 when S = 0 Z = D1 when S = 1	.02	.12	4.5	2.5	.005	27
Xid s0 s1 d0 d1 d2 d3 z mux4 (*Note order of s0 and s1*!)	Z=D0 when S_0=0, S_1=0 Z=D1 when S_0=1, S_1=0 Z=D2 when S_0=0, S_1=1 Z=D3 when S_0=1, S_1=1	.04	.19	4.5	2.5	.006	66
Xid d clk q dreg t_{setup} = .15, t_{hold} = 0	D→Q on CLK↑	.03	.19	4.3	2.5	.002	56

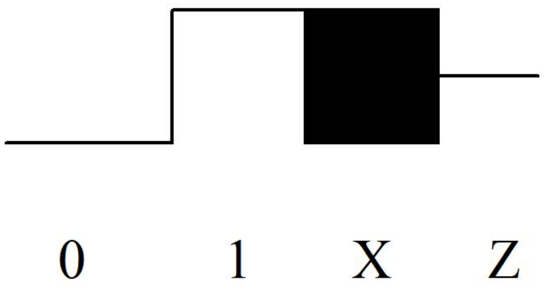
Gate-level Simulation

Since we are designing at the gate level we can use a faster simulator that only knows about gates and logic values (instead of transistors and voltages). You can run JSim's gate-level simulator by clicking in the toolbar. Note that your design cannot contain any mosfets, resistors, capacitors, etc.; this simulator only supports the gate primitives in the standard cell library.

Inputs are still specified in terms of voltages (to maintain netlist compatibility with the other simulators) but the gate-level simulator converts voltages into one of three possible logic values using the VIL and VIH thresholds specified in nominal.jsim:

0	logic low (voltages less than or equal to VIL threshold)
1	logic high (voltages greater than or equal to VIH threshold)
X	unknown or undefined (voltages between the thresholds, or unknown voltages)

A fourth value "Z" is used to represent the value of nodes that aren't being driven by any gate output (e.g., the outputs of tristate drivers that aren't enabled). The following diagram shows how these values appear on the waveform display:



Working with JSim

Connecting electrical nodes together using .connect

JSim has a control statement that lets you connect two or more nodes together so that they behave as a single electrical node:

```
.connect node1 node2 node3...
```

The .connect statement is useful for connecting two terminals of a subcircuit or for connecting nodes directly to ground. For example, the following statement ties nodes cmp1, cmp2, ..., cmp31 directly to the ground node (node "0"):

```
.connect 0 cmp[31:1]
```

Note that the .connect control statement in JSim works differently than many people expect. For example,

```
.connect A[5:0] B[5:0]
```

will connect all twelve nodes (A5, A4, ..., A0, B5, B4, ..., B0) together -- usually not what was intended. To connect two buses together, one could have entered

```
.connect A5 B5
.connect A4 B4
...
```

which is tedious to type. Or one can define a two-terminal device that uses `.connect` internally, and then use the usual iteration rules (see next section) to make many instances of the device with one "X" statement:

```
.subckt knex a b
.connect a b
.ends
X1 A[5:0] B[5:0] knex
```

Using iterators to create multiple gates with a single “X” statement

JSim makes it easy to specify multiple gates with a single "X" statement. You can create multiple instances of a device by supplying some multiple of the number of nodes it expects, e.g., if a device has 3 terminals, supplying 9 nodes will create 3 instances of the device. To understand how nodes are matched up with terminals specified in the .subckt definition, imagine a device with P terminals. The sequence of nodes supplied as part of the "X" statement that instantiates the device are divided into P equal-size contiguous subsequences. The first node of each subsequence is used to wire up the first device, the second node of each subsequence is used for the second device, and so on until all the nodes have been used. For example:

```
Xtest a[2:0] b[2:0] z[2:0] xor2
```

is equivalent to:

```
Xtest#0 a2 b2 z2 xor2
Xtest#1 a1 b1 z1 xor2
Xtest#2 a0 b0 z0 xor2
```

since xor2 has 3 terminals.

Duplicating a signal

There is also a handy way of duplicating a signal: specifying "foo#3" is equivalent to specifying "foo foo foo". For example, xor-ing a 4-bit bus with a control signal could be written as

```
Xbusctl in[3:0] ctl#4 out[3:0] xor2
```

which is equivalent to:

```
Xbusctl#0 in3 ctl out3 xor2
Xbusctl#1 in2 ctl out2 xor2
Xbusctl#2 in1 ctl out1 xor2
Xbusctl#3 in0 ctl out0 xor2
```

Connecting to ground

Using iterators and the “constant0” device from the standard cell library, here’s a better way of connecting cmp[31:1] to ground:

```
Xgnd cmp[31:1] constant0
```

Since the “constant0” has one terminal and we supply 31 nodes, 31 copies of the device will be made.

Task 1: Design the ALU

NOTE:

ALUFN $\neq$ OPCODE

The ALUFN signals used to control the operation of the ALU circuitry use an encoding chosen to make the design of the ALU circuitry as simple as possible. This encoding is ***not*** the same as the one used to encode the 6-bit opcode field of Beta instructions. In Lab 5, you will build some logic (actually a ROM) that will translate the opcode field of an instruction into the appropriate ALUFN control bits.

Part A: Adder / Subtractor

Design an **adder/subtractor** unit that operates on 32-bit two's complement inputs and generates a 32-bit output. It will be useful to generate three other output signals to be used by the comparison logic in Part B:

1. "Z" which is true when the S outputs are all zero
2. "V" which is true when the addition operation overflows (i.e., the result is too large to be represented in 32 bits), and
3. "N" which is true when the S is negative (i.e., S_{31} = 1).

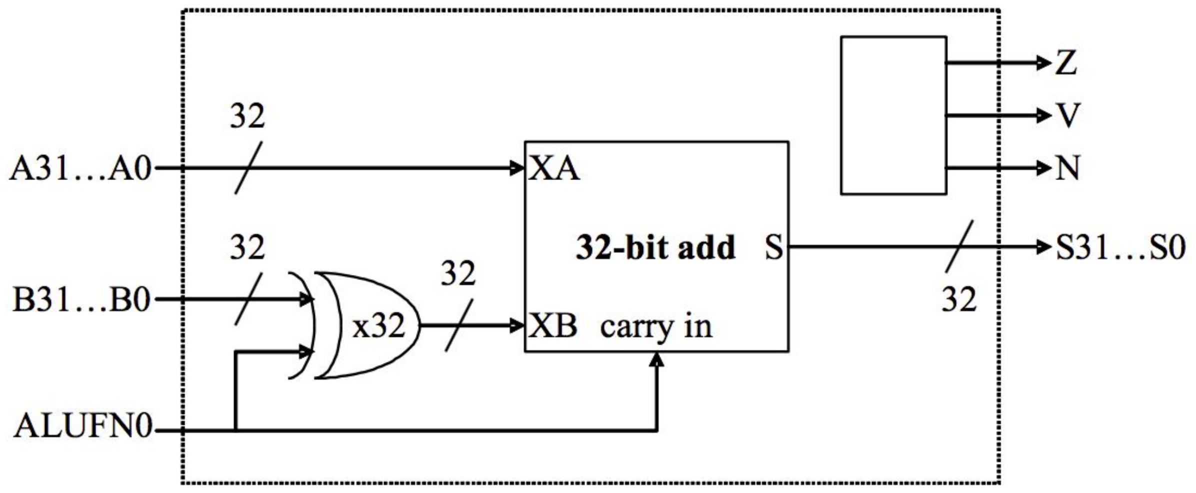
Overflow can never occur when the two operands to the addition have different signs; if the two operands have the same sign, then overflow can be detected if the sign of the result differs from the sign of the operands:

$$V = XA_{31} \cdot XB_{31} \cdot \overline{S_{31}} + \overline{XA_{31}} \cdot \overline{XB_{31}} \cdot S_{31}$$

Note that this equation uses XB_{31}, which is the high-order bit of the B operand to the adder itself (i.e., after the XOR gate – see the schematic below).

ALUFN0 will be set to 0 for an ADD (S = A + B) and 1 for a SUBTRACT (S = A – B); A[31:0] and B[31:0] are the 32-bit two's complement input operands; S[31:0] is the 32-bit result; Z/V/N are the three condition code bits described above. We will be using the "little-endian" bit numbering convention where bit 31 is the most-significant bit and bit 0 is the least-significant bit.

The following schematic is a suggestion for how to go about the design:



The ALUFN0 input signal selects whether the operation is an ADD or SUBTRACT. To do a SUBTRACT, the circuit first computes the two's complement negation of the "B" operand by inverting "B" and then adding one (which can be done by forcing the carry-in of the 32-bit add to be 1). Start by implementing the 32-bit add using a ripple-carry architecture (you'll get to improve on this later on the lab). You'll have to construct the 32-input NOR gate required to compute Z using a tree of smaller fan-in gates (the parts library only has gates with up to 4 inputs).

We’ve created a test jig to test your adder. Your netlist should incorporate the following three .include statements:

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3adder.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

```
.subckt adder32 ALUFN[0] A[31:0] B[31:0] s[31:0] z v n
… your adder/subtractor circuit here …
.ends
```

To use the test jig:

- Ensure your design file contains a definition for an “adder32” subcircuit as shown above.
- Do a gate-level simulation; a waveform window showing the adder32 inputs and outputs should appear.
- Click the checkoff button ✓. JSim will check your circuit’s results against a list of expected values and report any discrepancies it finds. Using this test jig file, nothing will be sent to the on-line server – it’s provided to help test your design as you go.

The Beta instruction set includes three compare instructions (CMPEQ, CMPLT, CMPLE) that compare the “A” and “B” operands. We can use the adder unit designed above to compute “AB” and then look at the result (actually just the Z, V and N condition codes) to determine if A=B, A<B or AB. The compare instructions generate a 32-bit Boolean result, using “0” to represent false and “1” to represent true.

Part B: Compare Unit

Design a 32-bit compare unit that generates one of two constants (“0” or “1”) depending on the ALUFN control signals (used to select the comparison to be performed) and the Z, V, and N outputs of the adder/subtractor unit. **Clearly the high order 31 bits of the output are always zero.** The **least significant bit of the output** is determined by the answer to the **comparison** being performed.

Comparison	Equation for LSB	ALUFN2	ALUFN1
$A = B$	$LSB = Z$	0	1
$A < B$	$LSB = N \oplus V$	1	0
$A \le B$	$LSB = Z + (N \oplus V)$	1	1

ALUFN[2:1] are used to control the compare unit since we also need to control the adder/subtractor unit (i.e., ALUFN0 = 1 to force a subtract).

Performance note: the Z, V and N inputs to this circuit can only be calculated by the adder/subtractor unit after the 32-bit add is complete. This means they arrive quite late and then require further processing in this module, which in turn makes cmp0 show up very late in the game. You can speed things up considerably by thinking about the relative timing of Z, V and N and then designing your logic to minimize delay paths involving late-arriving signals.

We have created a test jig to test your compare unit. Your netlist should incorporate the following three .include statements:

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3compare.jsim"
```

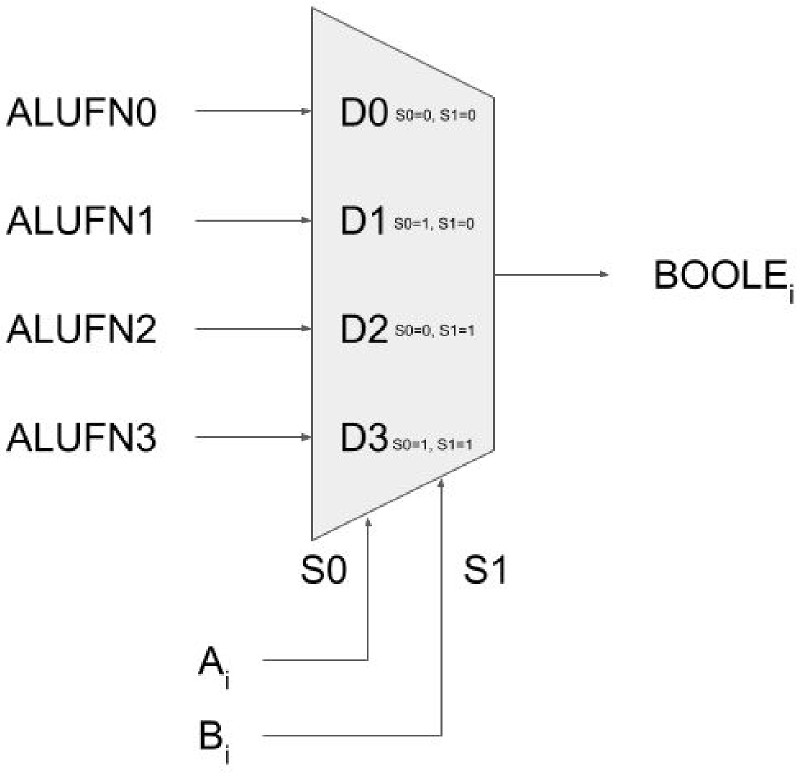
and the following subcircuit definition

```
.subckt compare32 ALUFN[2:1] z v n cmp[31:0]
… your compare circuit here …
.ends
```

Part C: Boolean Unit

Design a 32-bit Boolean unit for the Beta's logic operations. One implementation of a 32-bit boolean unit uses a 32 copies of a 4-to-1 multiplexer where ALUFN0, ALUFN1, ALUFN2, and ALUFN3 encode the operation to be performed, and Ai and Bi are hooked to the select inputs. This implementation can produce any of the 16 2-input Boolean functions; we'll only be using 4 of the possibilities.

Note the control signals and its corresponding output. See stdcell documentation on the 4-to-1 mux if you're unsure how these are obtained.


ALUFN0
ALUFN1
ALUFN2
ALUFN3
D0 S0=0, S1=0
D1 S0=1, S1=0
D2 S0=0, S1=1
D3 S0=1, S1=1
BOOLE$_i$
S0
S1
A$_i$
B$_i$

The following table shows the encodings for the ALUFN[3:0] control signals used by the test jig. If you choose a different implementation you should also include logic to convert the supplied control signals into signals appropriate for your design.

Operation	**ALUFN[3:0]**
AND	1000
OR	1110
XOR	0110
"A"	1010

We've created a test jig to test your boolean unit. Your netlist should incorporate the following three .include statements

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3boolean.jsim"
```

and the following subcircuit definition

```
.subckt boole32 ALUFN[3:0] A[31:0] B[31:0] boole[31:0]
… your boolean unit circuit here …
.ends
```

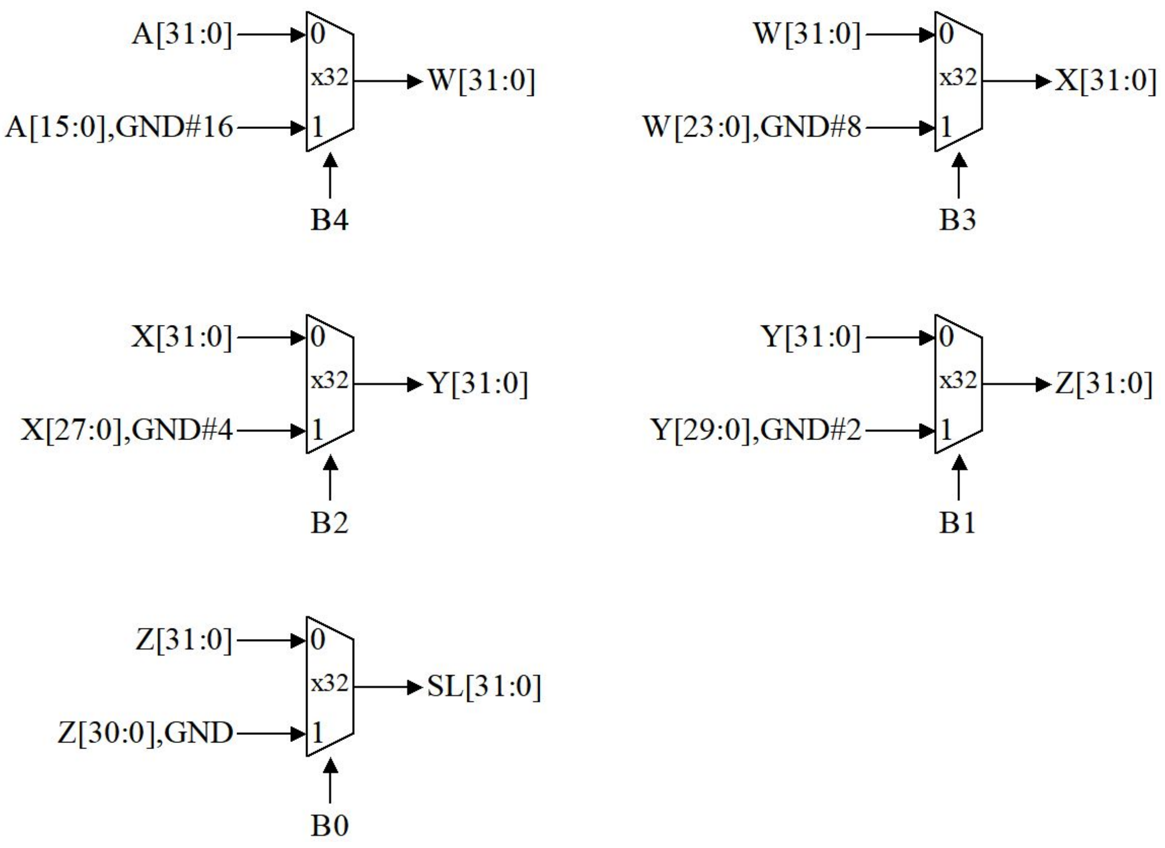
Part D: Shifter

Design a 32-bit shifter that implements SRA, SHR and SHL instructions. The "A" operand supplies the data to be shifted and the low-order 5 bits of the "B" operand are used as the shift count (i.e., from 0 to 31 bits of shift). The desired operation will be encoded on ALUFN[1:0] as follows:

Operation	**ALUFN[1:0]**
SHL (shift left)	00
SHR (shift right)	01
SRA (shift right with sign extension)	11

With this encoding, ALUFN0 is 0 for a left shift and 1 for a right shift and ALUFN1 controls the sign extension logic on right shift. For SHL and SHR, 0's are shifted into the vacated bit positions. For SRA ("shift right arithmetic"), the vacated bit positions are all filled with A31, the sign bit of the original data so that the result will be the same as dividing the original data by the appropriate power of 2.

The simplest implementation is to build two shifters—one for shifting left and one for shifting right—and then use a 2-way 32-bit multiplexer to select the appropriate answer as the unit's output. It's easy to build a shifter after noticing that a multi-bit shift can be accomplished by cascading shifts by various powers of 2. For example, a 13-bit shift can be implemented by a shift of 8, followed by a shift of 4, followed by a shift of 1. So the shifter is just a cascade of multiplexers each controlled by one bit of the shift count. The schematic below shows a possible implementation of the left shift logic; the right shift logic is similar with the slight added complication of having to shift in either "0" or "A31." Another approach that adds latency but saves gates is to use the left shift logic for both left and right shifts, but for right shifts, reverse the bits of the "A" operand on the way in and reverse the bits of the output on the way out.



We have created a test jig to test your shift unit. Your netlist should incorporate the following three .include statements:

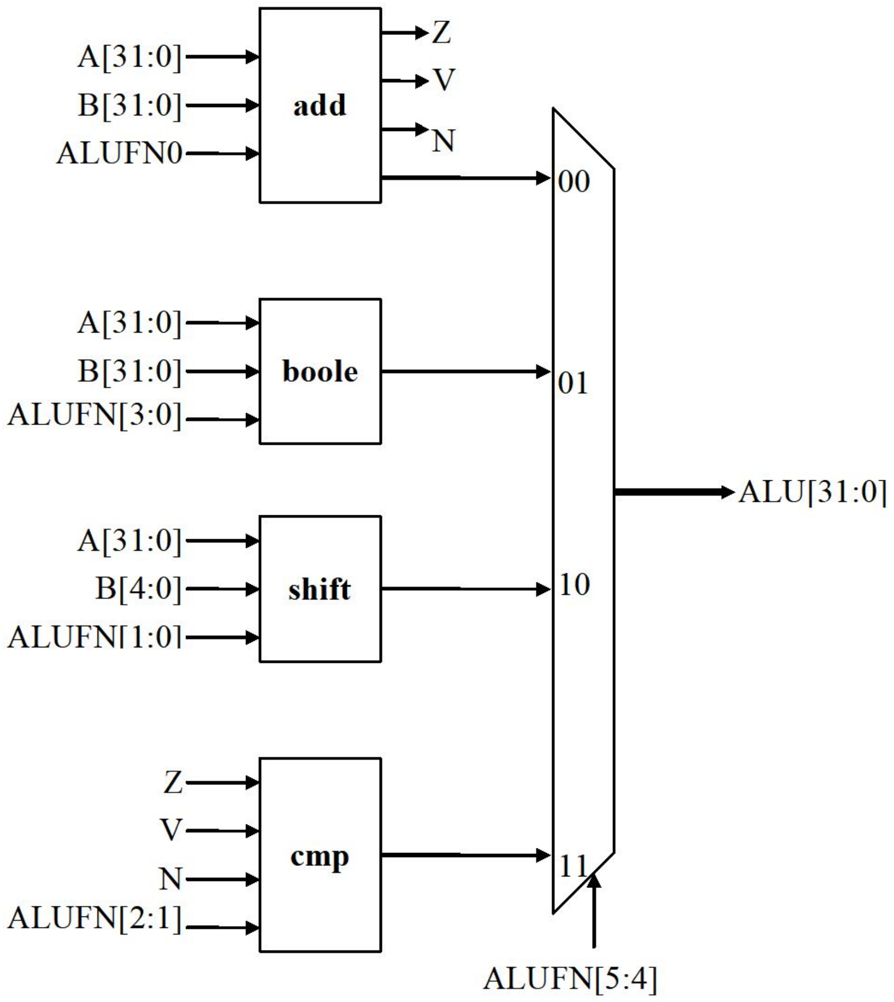
```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3shifter.jsim"
```

and the following subcircuit definition

```
.subckt shift32 ALUFN[1:0] A[31:0] B[4:0] shift[31:0]
… your shifter circuit here …
.ends
```

Part E: Combine and make the ALU

Combine the outputs of the adder, compare, boolean and shift units to produce a single ALU output. The simplest approach is to use a 4-way 32-bit multiplexer as shown in the schematic below:



Two additional control signals (ALUFN[5:4]) have been introduced to select which unit will supply the value for the ALU output. The encodings for ALUFN[5:0] used by the test jig are shown in the following table:

Operation	**ALUFN[5:0]**	**hex**
ADD	000000	0x00
SUB	000001	0x01
AND	011000	0x18
OR	011110	0x1E
XOR	010110	0x16

“A” (LDR)	011010	0x1A
SHL	100000	0x20
SHR	100001	0x21
SRA	100011	0x23
CMPEQ	110011	0x33
CMPLT	110101	0x35
CMPLE	110111	0x37

Part F: Test Your Circuit

When you have completed your design, use lab3checkoff_6.jsim to test your circuit. Your netlist should incorporate the following three .include statements

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3checkoff_6.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

```
.subckt alu ALUFN[5:0] A[31:0] B[31:0] alu[31:0] z v n
… your ALU circuit here …
.ends
```

Note that the z, v, and n signals from the adder/subtractor unit are included in the terminal list for the alu subcircuit. While these signals are not needed when using the ALU as part of the Beta, they are included here to make it easier for the test jig to pinpoint problems with your circuit.

Before using lab3checkoff_6.jsim remember to comment out any test circuitry and .tran statements you may have added to your netlist while debugging your circuit. Also remember to use JSim’s gate-level simulator to simulate your circuit.

If this test jig runs okay, it will offer to check-in your lab with the on-line server.

Task 2: Design a Multiplier

The goal of this design project is to build a combinational multiplier that accepts 32-bit operands and produces a 32-bit result. Multiplying two 32-bit numbers produces a 64-bit product; the result we're looking for is the low-order 32-bits of the 64-bit product.

Your multiplier circuitry should be integrated into the ALU design you completed in the first part of this lab. We will use the following encoding for ALUFN[5:0] when requesting a multiply operation by the ALU.

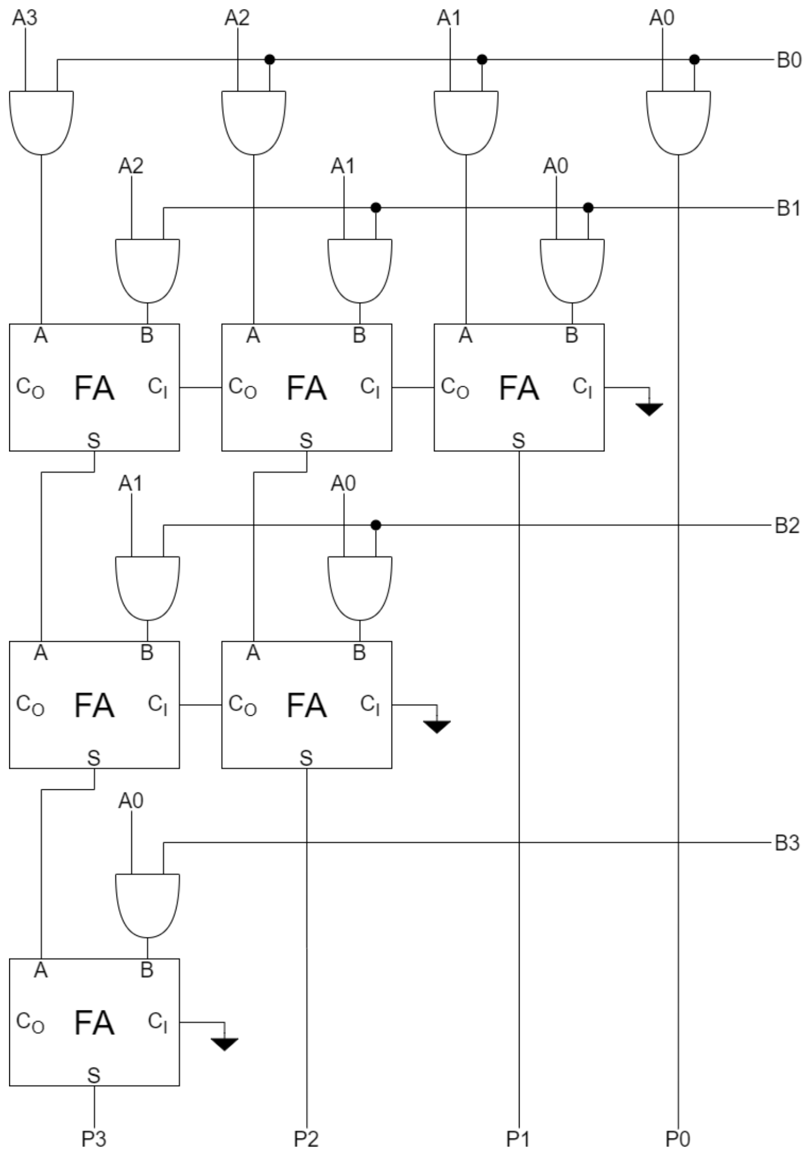
Operation	**ALUFN[5:0]**	**hex**
MUL	000010	0x02

Here is a detailed bit-level description of how a 4-bit by 4-bit unsigned multiplication works. This diagram assumes we only want the low-order 4 bits of the 8-bit product.

```
            A3       A2       A1       A0   (multiplicand)
*           B3       B2       B1       B0   (multiplier)
    ----------------------------------------------------
         A3*B0    A2*B0    A1*B0    A0*B0   (partial product)
+        A2*B1    A1*B1    A0*B1        0
+        A1*B2    A0*B2        0        0
+        A0*B3        0        0        0
    ----------------------------------------------------
            P3       P2       P1       P0
```

This diagram can be extended in a straightforward way to 32-bit by 32-bit multiplication. Note that since we only want the low-order 32-bits of the result, you don't need to include the circuitry that generates the rest of the 64-bit product.

As you can see from the diagram above, forming the partial products is easy! Multiplication of two bits can be implemented using an AND gate. The hard part is adding up all the partial products (there will be 32 partial products in your circuit). One can use full adders (FAs) hooked up in a ripple-carry configuration to add each partial product to the accumulated sum of the previous partial products (see the diagram below). The circuit closely follows the diagram above but omits an FA module if two of its inputs are "0".



The circuit above works with both unsigned operands and signed two's complement operands. This may seem strange – don't we have to worry about the most significant bit (MSB) of the operands? With unsigned operands the MSB has a weight of 2^{MSB} (assuming the bits are numbered 0 to MSB) but with signed operands the MSB has a weight of -2^{MSB}. Doesn't our circuitry need to take that into account?

It does, but when we are only saving the lower half of the product, the differences don't appear. The multiplicand (A in the figure above) can be either unsigned or two's complement, the FA circuits will perform correctly in either case. When the multiplier (B in the figure above) is signed, we should subtract the final partial product instead of adding it. But subtraction is the same as adding the negative, and the negative of a two's complement number can be computed by taking its complement and adding 1. When we work this through we see that the low-order bit of the partial product is the same whether positive or negated. And the low-order bit is all that we need when saving only the lower half of the product! If we were building a multiplier that computed the full product, we'd see many differences between a multiplier that handles unsigned operands and one that handles two's complement operands, but these differences only affect how the high half of the product is computed.

We've created a test jig to help debug your multiplier. Your netlist should incorporate the following three .include statements:

```
.include "/50002/nominal.jsim"
.include "/50002/stdcell.jsim"
.include "/50002/lab3multiply.jsim"
```

and the standard alu subcircuit definition:

```
.subckt alu ALUFN[5:0] A[31:0] B[31:0] alu[31:0] z v n
… your ALU with multiplier circuit here …
.ends
```

This test jig includes test cases for

all combinations of (0, 1, -1)*(0,1,-1),
2i*1 for i = 0, 1, …, 31
-1*2i for i = 0, 1, …, 31
(3 << i) * 3 for i = 0, 1, …, 31

Submission

When you've completed your design, you can use lab3checkoff_10.jsim to test your improved ALU implementation. This checkoff file contains all the tests from lab3checkoff_6.jsim plus additional tests to verify that your multiplier circuitry is working correctly. There is also a handy set of debugging tests in lab3multiply.jsim which can help track down problems in your design.

Afterwards, do not forget to do the Lab Quiz. Refer to the class calendar for dates.

Design Note: Combinational multipliers implemented as described above are pretty slow! There are many design tricks we can use to speed things up – see the appendix on "Computer Arithmetic" in any of the editions of ***Computer Architecture*** *A Quantitative Approach* by John Hennessy and David Patterson (Morgan Kauffmann publishers).