**Due date:** Tuesday March 12 11:59:59pm EST.

**Getting started:** To create your initial Lab 4 repository, please visit the repository creation page at https://6004.mit.edu/web/spring19/user/labs/lab4. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-spring19/labs-lab4-{YourMITUsername}.git lab4
```

**Turning in the lab:** To turn in this lab, commit and push the changes you made to your git repository. After pushing, check the course website to verify that your submission passes all the tests. If you finish the lab in time but forget to push, you will incur the standard late submission penalties.

**Check-off meeting:** After turning in this lab, you are required to go to the lab for a check-off meeting within 6 days of the lab’s due date. See the course website for lab hours.

**Introduction**

In this lab you will build a 32-bit Arithmetic Logic Unit (ALU). An ALU performs arithmetic and bitwise-logical operations on integers. It is an important component of computer processors and many other digital systems. You will build an ALU for the RISC-V processor that you will design in later labs.

The first part of the lab focuses on building a functionally correct ALU. You will first design and implement a variety of arithmetic and logic operations. You will then use these to construct an ALU for the RISC-V processor.

The second part of the lab examines the performance and cost of several ALU components by using a simple synthesis tool. The last exercise concerns the design and evaluation of a fast adder.

To pass the lab you must complete and PASS all of the exercises except the LAST one.

**Implementation restrictions:** In this lab you will build several arithmetic circuits for which Bluespec already has operators. You cannot use these operators in your circuits. **Specifically, you are NOT ALLOWED to use the following operators in your logic: + - * / % (addition, subtraction, multiplication, division, modulus), << >> (left/right shifts), <= >= < > (comparisons) or variable bit indexing a[b], where a,b are both bit variables.** Bitwise-logical operators (& | ^ ∼), equality/inequality operators (== !=), conditional expressions (?: if), and loops are allowed. You will get an illegal operator error if your code is synthesized into non-logic gate primitives, such as adders, subtractors, multipliers, dividers, modulus, shifters and comparators.

There is one exception to this rule: you can use any operators on Integer types. Integer types are used as iteration variables in for loops and they do not produce circuits with high-level operators. For example, the following code is allowed:

```
for (Integer i = 0; i < 32; i = i + 1) begin
    c[i] = a[i] ^ b[i];
end
```

Although the code uses the < and + operators, it does so on Integer variable i. The loop is unrolled and the resulting circuit does not perform any of those operations (it performs bitwise XOR).

Please refer to Appendix A and B for additional debugging help.
Building and Testing Your Circuits

You can build your code with make. If you just run make it will build all of the exercises. You can instead pass in a target so that only one of the exercises will be built, like so:

    make <target>

This will then create a program sim_<target> that you can run which simulates the circuit. It will run through a set of test cases printing out each input and whether or not it fails to match the expected output. If it passes all the tests it will print out PASSED at the end.

- To build all the targets, run
  
  make all

- To build and test everything, run
  
  make test

Part 1: Designing a Functionally Correct ALU

An arithmetic-logic unit (ALU) is the part of a processor that carries out arithmetic and logic operations on the operands in computer instruction words. Many of these operations (e.g., addition and subtraction) can be performed by the same logic with minor changes. Therefore, rather than building a different circuit for each operation, each of these exercises asks you to first build a circuit that performs a single operation and then to reuse it or extend it to perform more operations. This results in a better design, as different operations share logic, and also results in simpler code.

1 32-bit Shifter

In these exercises, you will implement 32-bit shifters for logic and arithmetic operations. You will implement your shifters by building on the barrel shifter design from Lecture 6.

Exercise 1 (13 points): First, implement a general 32-bit barrel right shifter barrelRShift. barrelRShift can shift in ones or zeros as specified by input sft_in. Then, use barrelRShift to build a 32-bit arithmetic and logical right shifter. A logical right shift shifts in zeros, whereas an arithmetic right shift shifts in the most significant bit of the input (this performs a division by 2^sftSz for two’s complement numbers).

Hint 1: We do not recommend you use for loops for this exercise. Using Integer variables to select bits, such as a[31-i:0], will produce type errors that are beyond the scope of this class.

Hint 2: To replicate 1-bit sft_in to N bits, use Bluespec function signExtend, which extends the input bit by copying its MSB. Refer to Appendix A to see an example of how signExtend() is used.

Fill your code in the following skeleton functions in ALU.bsv

```haskell
// 32-bit right barrel shifter
// Arguments: in (value to be shifted); sftSz (shift size); sft_in (bit value shifted in)
function Bit#(32) barrelRShift(Bit#(32) in, Bit#(5) sftSz, Bit#(1) sft_in);

// 32-bit arithmetic/logical right shifter
// arith = 1, arithmetic shift; logical shift otherwise
function Bit#(32) sr32(Bit#(32) in, Bit#(5) sftSz, Bit#(1) arith);
```

Test your design by running make sr32 && ./sim_sr32
Exercise 2 (6 points): Implement a 32-bit Logical Left Shifter by reusing the `barrelRShift` circuit you implemented above. Your design must call `barrelRShift` only once.

*Hint:* To construct a right shifter from a left shifter, you just need to reverse the bits of the input and output. The `reverseBits` Bluespec function does this.

Fill your code in the following skeleton function in `ALU.bsv`.

```plaintext
function Bit#(32) sll32(Bit#(32) in, Bit#(5) sftSz); // 32-bit logical left shifter
```

Test your design by running: `make sll32 && ./sim`

Exercise 3 (6 points): Implement a 32-bit FULL Shifter, which has arithmetic/logical and right/left shift operations. Use a SINGLE call to the `barrelRShift` function you implemented above, so that your circuit uses a single barrel shifter. Fill your code in the following skeleton function in `ALU.bsv`.

```plaintext
// 32-bit FULL shifter
typedef enum {LogicalRightShift, ArithmeticRightShift, LeftShift} ShiftType deriving (Bits, Eq, FShow);
function Bit#(32) sft32(Bit#(32) in, Bit#(5) sftSz, ShiftType shiftType);
```

Test your design by running: `make sft32 && ./sim`

2 32-bit Comparator

In these exercises, you will implement a 32-bit comparator for both signed and unsigned numbers. You will use your knowledge of two’s complement encoding from Lecture 2.

You can compare two *unsigned* 32-bit values `a` and `b` by comparing their bits one by one left-to-right, i.e., from the most-significant bit (MSB) to the least-significant bit (LSB), as shown in Figure 1. In the figure, `a_i` and `b_i` denote the `i`th bit of `a` and `b`, respectively. `eq_i` is 1 if `a` and `b` are equal from the MSB until their `i`th bit, i.e., if `a[N-1:i] == b[N-1:i]`. Likewise, `lt_i` is 1 if `a` is less than `b` from the MSB until the `i`th bit, i.e., if `a[N-1:i] < b[N-1:i]`. We are interested in `lt_0`, which will be 1 if `a` is less than `b` and 0 otherwise.

![Figure 1: Circuit diagram of an unsigned N-bit less-than comparator.](image)

Exercise 4 (13 points): First, implement a *one-bit comparator* (each of the blocks in Figure 1). Then, use it to construct an *unsigned 32-bit less-than comparator*.

*Hint:* To build the one-bit comparator, we recommend you derive the Boolean equations for `eq_i` and `lt_i` as functions of inputs `eq_{i+1}`, `lt_{i+1}`, `a_i`, and `b_i`. Note that `eq_i` depends only on `eq_{i+1}`, `a_i`, and `b_i`; for `a` and `b` to be equal down to the `i`th bit, they must be equal down to the `(i+1)`th bit and bits `a_i` and `b_i` must be equal.

However, `lt_i` does depend on all four inputs: for `a` to be less than `b` down to the `i`th bit, either `a` is less than `b` down to the `(i+1)`th bit (in which case all lower-order bits don’t matter), or `a` is equal to `b` down to the `(i+1)`th bit and `a_i` is less than `b_i`.

To build the 32-bit comparator, feed inputs `eq_N = 1` and `lt_N = 0` to the left-most one-bit comparator.

Fill your code in the following skeleton functions in `ALU.bsv`. 
// one-bit less-than comparator
// Arguments: a, b (1-bit values), eq, lt (eq and lt from previous comparator)
// Return: {eq, lt}
function Bit#(2) cmp(Bit#(1) a, Bit#(1) b, Bit#(1) eq, Bit#(1) lt);

function Bit#(1) ltu32(Bit#(w) a, Bit#(w) b); // unsigned 32-bit less-than comparator

• To test the 1-bit comparator, run make cmp && ./sim_cmp
• To test the unsigned 32-bit less-than Comparator, run make ltu32 && ./sim_ltu32

Exercise 5 (6 points): Implement a signed/unsigned 32-bit less-than comparator using at most a SINGLE call to the unsigned comparator function you just implemented.

Hint 1: Consider bit string \( a = a_{N-1}a_{N-2}...a_1a_0 \). If we interpret \( a \) as a two’s complement signed binary number, it represents the value:

\[
v_{\text{signed}} = -2^{N-1}a_{N-1} + \sum_{i=0}^{N-2} 2^ia_i
\]

However, if we interpret \( a \) as an unsigned binary number, it represents the value:

\[
v_{\text{unsigned}} = \sum_{i=0}^{N-1} 2^ia_i = +2^{N-1}a_{N-1} + \sum_{i=0}^{N-2} 2^ia_i
\]

The only difference between the signed and unsigned representations is that the weight of the most significant bit is negative in the signed case and positive in the unsigned case. Therefore, to perform a signed comparison using an unsigned comparator, you only need to tweak the most significant bit of inputs \( a \) and \( b \), i.e., the inputs to the left-most comparator. To derive how you should change the inputs to the left-most comparator, think about what a less-than comparison becomes when you change the signs of the numbers you’re comparing.

Hint 2: To choose between signed and unsigned comparison, you only need the isSigned input to control the most significant bit of inputs \( a \) and \( b \) of the unsigned comparator.

Fill your code in the following skeleton function in ALU.bsv.

// Signed/Unsigned 32-bit less-than comparator
// isSigned, signed comparison; unsigned otherwise
function Bit#(1) lt32(Bit#(w) a, Bit#(w) b, Bit#(1) isSigned);

Test your design by running: make lt32 && ./sim_lt32

3 N-bit Ripple-Carry Adder/Subtractor

In these exercises you will implement an N-bit Ripple-Carry adder, that is able to perform both addition and subtraction. However, these exercises differ from the previous ones in that the Ripple Carry Adder’s bitwidth is variable. There are some additional considerations to have when designing functions that take in inputs and produce outputs of variable bitwidths. Review the contents from lecture 7 and observe how we’ve modified the 32-bit Ripple-Carry Adder to instead take in N-bit inputs. Please refer to Appendix A for more information on how to deal with parametric functions.

Exercise 6 (6 points): First, implement a full adder circuit. Then, construct an N-bit ripple-carry adder using full adders. Note that the output of the adder is only N-bits wide instead of N+1. This is slightly different than what we’ve seen before, but if you remember from the Assembly labs, every operand and
output had to be able to fit in a 32-bit register. Hence, the operands in our adders are 32-bits, and their output, which has to fit in a register, is also 32-bits wide. This means even though adding the two operands together would cause an overflow, our adder’s design will not handle the overflow and instead allow it to occur. Fill your code in the following skeleton functions in ALU.bsv.

```haskell
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in); // full adder
function Bit#(w) rcaN(Bit#(w) a, Bit#(w) b, Bit#(1) c_in); // N-bit ripple-carry adder
```

- To test your full adder, run `make fa && ./sim fa`
- To test your ripple-carry adder, run `make rca32 && ./sim_rca32`

**Exercise 7 (6 points):** Design an N-bit adder/subtractor, a single circuit that can add and subtract integers in two’s complement encoding. Your adder/subtractor must use a single ripple-carry adder. To this end, your function may only use a SINGLE function call to the `rcaN` function you just implemented.

*Hint:* Recall that `A - B` is equivalent to `A + (-B)`. In 2’s complement representation `-B` is equal to flipping all the bits of `B` and adding 1.

Fill in the following skeleton function in `ALU.bsv`.

```haskell
// N-bit RC adder/subtractor
// returns a - b is isSub==1 and a + b otherwise
function Bit#(w) addSubN(Bit#(w) a, Bit#(w) b, Bit#(1) isSub);
```

Test your design by running `make addSub32 && ./sim_addSub32`

## 4 32-bit Arithmetic Logic Unit

Now that you have built the ALU’s main components, it is time to build the full ALU. Table 1 shows the functions that the RISC-V processor needs the ALU to perform.

<table>
<thead>
<tr>
<th>ALU Function</th>
<th>Operation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>32-bit ADD</td>
<td><code>a + b</code></td>
</tr>
<tr>
<td>Sub</td>
<td>32-bit SUBTRACT</td>
<td><code>a - b</code></td>
</tr>
<tr>
<td>And</td>
<td>32-bit AND</td>
<td><code>a &amp; b</code></td>
</tr>
<tr>
<td>Or</td>
<td>32-bit OR</td>
<td>`a</td>
</tr>
<tr>
<td>Xor</td>
<td>32-bit XOR</td>
<td><code>a ^ b</code></td>
</tr>
<tr>
<td>Slt</td>
<td>Set if less than signed</td>
<td><code>a&lt;b,?1:0</code></td>
</tr>
<tr>
<td>Sltu</td>
<td>Set if less than unsigned</td>
<td><code>a&lt;b_u,?1:0</code></td>
</tr>
<tr>
<td>Sll</td>
<td>SHIFT Left Logical</td>
<td><code>a&lt;&lt;b[4:0]</code></td>
</tr>
<tr>
<td>Srl</td>
<td>SHIFT Right Logical</td>
<td><code>a&gt;&gt;a,b[4:0]</code></td>
</tr>
<tr>
<td>Sra</td>
<td>SHIFT Right Arithmetic</td>
<td><code>a&gt;&gt;a,b[4:0]</code></td>
</tr>
</tbody>
</table>

Table 1: RISC-V ALU functions

**Exercise 8 (20 points):** Implement a 32-bit ALU for the RISC-V processor. Your implementation should perform a SINGLE call to each of the circuits you implemented in the previous exercises: the adder/subtractor (`addSubN`), the signed/unsigned comparator (`lt32`), and the full shifter (`sft32`).

*Hint:* Use `zeroExtend` to extend the 1-bit output of `lt32` to 32 bits. This extends the input with zeros.

Fill in the following skeleton function in `ALU.bsv`.

```haskell
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra} AluFunc deriving (Bits, Eq, FShow);
function Bit#(32) alu(Bit#(32) a, Bit#(32) b, AluFunc func); // Arithmetic Logic Unit (ALU)
```
Test your design by running `make alu32 && ./sim_alu32`

**Part 2: Analyzing and Improving Circuit Performance**

So far we have focused on describing circuits in Bluespec and simulating them to test their functionality, without paying much attention to their implementation. Now that you have designed your first large digital circuit, it is time we start looking at implementation tradeoffs. To do this, we need a synthesis tool that translates Bluespec designs into optimized gate-level implementations.

We will use `synth`, a simple synthesis tool. From Lecture 6, recall that synthesis tools require three types of input: the circuit to be synthesized, the standard cell library of gates to use to implement the circuit, and the optimization objective (area, delay, power, or some combination of them). synth comes with several standard cell libraries, all of them derived from the open-source FreePDK45 library (https://research.ece.ncsu.edu/eda/freepdk/freepdk45/). This is representative of 45 nm silicon fabrication technology introduced in 2008. By default, synth optimizes for delay, but this is controllable by specifying a target delay. synth will try to reduce the circuit’s area as long as its delay is under the target. Under the hood, synth uses the Bluespec compiler and the Yosys open synthesis suite (http://www.clifford.at/yosys/).

In this part of the lab, we will first learn how to use synth by synthesizing and analyzing several circuits. These exercises will build your intuition of delay and area implementation tradeoffs and of the optimizations synth can perform. We will then study delay-area tradeoffs of our barrel shifter implementation. Finally, you will (optionally) build a better adder to improve your ALU’s performance. We will discuss synth’s usage options along the way; to see a quick reference, run `synth -h`.

### 5 Analyzing the Ripple-Carry Adder

The `rca4` function in Adders.bsv uses your N-bit ripple-carry adder circuit (rcaN) to implement a 4-bit ripple-carry adder. Synthesize `rca4` by running:

```
synth Adders.bsv rca4
```

synth reports three pieces of information. First, it reports three summary statistics: the number of gates, the total area these gates take (in square micrometers), and the circuit’s critical-path delay (i.e., the longest propagation between any input-output pair) in picoseconds. Second, it reports the delay across the different gates of the critical path. Third, it shows a breakdown of the types of gates used and their area.

synth can also produce circuit diagrams. Run:

```
synth Adders.bsv rca4 -v
```

This will produce a diagram in `rca4.svg`. Open by running `firefox rca4.svg` (or use another SVG viewer).

By default, synth uses the basic standard cell library, which only has a buffer, an inverter, and two-input NAND and NOR gates. You can control the library used with the `-l` flag. Let’s try using the extended library, which also has AND, OR, XOR, and XNOR gates, as well as 2, 3, and 4-input gates. Run:

```
synth Adders.bsv rca4 -l extended
```

**Discussion Question:** How does `rca4` change when synthesized with the extended library? What gates are used now vs. with the basic library? How does this affect area and delay? (You can visualize the new circuit by running `synth Adders.bsv rca4 -l extended -v && firefox rca4.svg`.)

By default, synth tries to minimize delay by setting a target delay of 1 ps. You can control the target delay with the `-d` flag. Lax delays will cause the synthesis tool to optimize for area instead. This way, you can trade off delay for area (in current technology power is also a crucial consideration, even more so than area; but power analysis is a complex topic, so in this course we will focus on area and delay). For example, the following command uses a target delay of 1000 ps:

```
synth Adders.bsv rca4 -l extended -d 1000
```
Discussion Question: Synthesize \texttt{rca4} using the command above. How do its area and delay change vs. the previous (delay-optimized) circuit?

\texttt{synth} also performs some optimizations, such as Boolean simplification. Take a look at the \texttt{add4\_rca} and \texttt{add4\_addSub} functions in Adders.bsv. Both implement the same function, adding two 4-bit numbers. However, \texttt{add4\_rca} uses your \texttt{rcaN} circuit with the carry-in argument set to 0, whereas \texttt{add4\_addSub} uses your addSubN circuit with \texttt{isSub} set to 0.

**Discussion Question:** Synthesize \texttt{add4\_rca} and \texttt{add4\_addSub}. Why don’t the circuits differ? Give one example of how Boolean simplification is helping produce this result.

Finally, let’s analyze how area and delay grow with the number of inputs. Adders.bsv has \texttt{rca} functions of different widths, \texttt{rcaX} for $X \in \{2, 4, 8, 16, 32, 64, 128\}$.

**Discussion Question:** Synthesize the above functions. How does delay grow with the number of bits? How does area grow with the number of bits?
6 Multiplexers and Fanout

Muxes.bsv has several implementations of a 2-bit multiplexer:

- `mux2_sop` uses a minimal sum-of-products representation.
- `mux2_select` uses the conditional (ternary) operator.
- `mux2_if` uses if-else blocks.
- `mux2_case` uses a case statement.

Each of these may result in different implementations.

**Discussion Question:** Synthesize the above functions. Why don’t their implementations differ?

Now let’s focus on critical-path analysis. Figure 2 shows a sample critical-path analysis from synth (this is for the seven-segment decoder circuit from lab 3—try it out!). This analysis shows how delay grows over the logic gates in the critical path. Each row corresponds to a single gate. The gate delay column reports the gate’s propagation delay (from its input becoming a valid and stable digital value to its output becoming a valid and stable digital value). The fanout column reports the number of inputs that the output of the gate is driving. For example, the NOR3 gate in Figure 2 contributes the longest delay, 43.4 ps, and its output is connected to the inputs of five gates (among them, the NOR2 gate that’s next in the critical path).

The first row in this table always corresponds to an input, and the last row corresponds to an output. Note that the first row has some delay because there is always a gate (a buffer in our case) driving every input pin. By contrast, the output has no gate delay, as the output of the last gate is simply the output.

**Critical-path delay:** 127.19 ps

**Critical path:** `binary_number[3]` -> `out[0]`

<table>
<thead>
<tr>
<th>Gate/port</th>
<th>Fanout</th>
<th>Gate delay (ps)</th>
<th>Cumulative delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary_number[3]</code></td>
<td>3</td>
<td>7.6</td>
<td>7.6</td>
</tr>
<tr>
<td><code>INV</code></td>
<td>7</td>
<td>18.9</td>
<td>26.5</td>
</tr>
<tr>
<td><code>NAND2</code></td>
<td>2</td>
<td>12.2</td>
<td>38.7</td>
</tr>
<tr>
<td><code>NOR3</code></td>
<td>5</td>
<td>43.4</td>
<td>82.1</td>
</tr>
<tr>
<td><code>NOR2</code></td>
<td>3</td>
<td>13.4</td>
<td>95.5</td>
</tr>
<tr>
<td><code>NAND3</code></td>
<td>5</td>
<td>23.0</td>
<td>118.5</td>
</tr>
<tr>
<td><code>NAND2</code></td>
<td>1</td>
<td>8.7</td>
<td>127.2</td>
</tr>
<tr>
<td><code>out[0]</code></td>
<td>0</td>
<td>0.0</td>
<td>127.2</td>
</tr>
</tbody>
</table>

Figure 2: Example critical-path analysis report.

In general, the gate’s propagation delay depends on two main factors: the complexity of the gate and how loaded the gate’s output is.

The complexity of the gate is technology-specific; as we saw in Lecture 6, in current technology inverting logic is faster than non-inverting logic (so you’ll see the synthesis tool use inverting gates much more often), and gates with more inputs are slower (e.g., NAND3 is slower than NAND2).

How loaded a gate is also determines its delay. The output of each gate has a certain drive strength, which refers to how much current the gate can draw from the power supply to control the output voltage. In turn, each gate input is essentially a capacitor, so the capacitance at the output of a gate grows the more gate inputs we connect to it. In short: the higher the gate’s drive strength, the faster it can switch the output between 0 and 1; and the more inputs the gate’s output is connected to, the more capacitance that needs to be charged or discharged and the slower the output will switch between 0 and 1.

While the exact relationship between output load and delay is complex, a good first-order approximation is to consider that delay grows linearly with fanout, the number of inputs connected to each output. Indeed, Figure 2 shows that gates with higher fanout have higher delay (e.g., compare both NAND2 gates).

You should consider the effect of fanout when designing circuits. For example, it is common to believe that the delay of a multiplexer does not depend on its width—after all, a 1-bit and a 32-bit mux both have the same levels of logic. But as the number of bits grows, the single select signal drives a growing number
of gates, i.e., its fanout grows.

Muxes.bsv has multiplexers of different widths, \( \text{mux}_X \) for \( X \in \{2, 4, 8, 16, 32, 64, 128, 256\} \). Synthesize these muxes with a really high delay target, e.g., 10000 ps, like so:

```bash
synth Muxes.bsv mux64 -l extended -d 10000
```

**Discussion Question:** How does delay grow with multiplexer width? How is this delay distributed across the nodes of the critical path?

To ameliorate the effect of high fanout, tools automatically insert chains or trees of buffers and inverters to reduce the load at each gate. For example, consider a single gate that is driving 64 inputs, which causes excessive delay. The tool could instead connect the gate to four inverters, each of the four inverters to another four inverters, and each of these 16 inverters to four of the original 64 gates. Each inverter adds some internal delay, but each gate in the tree now only drives four other gates.

Synthesize muxes \( \text{mux}_X \) for \( X \in \{2, 4, 8, 16, 32, 64, 128, 256\} \) again, but this time do not specify a target delay so that the muxes are delay-optimized instead of area-optimized, like so:

```bash
synth Muxes.bsv mux64 -l extended
```

**Discussion Question:** How does delay grow with multiplexer width for these delay-optimized multiplexers? How does area grow with multiplexer width?

Standard cell libraries often include gates of different sizes. Larger gates have higher drive strength, but they also take more area and have higher capacitance at its inputs, so they take more effort to switch. Mixing gates of different sizes gives the synthesis tool more freedom to trade area for delay. For example, the tool can use larger gates along the critical path to reduce delay, and smaller gates elsewhere to keep area low.

```bash
synth has a multisize library that includes the same gates as extended, but each gate has variants of multiple sizes (denoted X1, X2, X4, and X8). Synthesize \( \text{mux}_{64} \) with the multisize library, like so:

```bash
synth Muxes.bsv mux64 -l multisize
```

**Discussion Question:** How do \( \text{mux}_{64} \)’s delay and area change with multisize vs. with extended? Where does the tool use larger gates?

### 7 Analyzing your ALU

Synthesize your ALU with the multisize library and optimizing for delay:

```bash
synth ALU.bsv alu -l multisize
```

Also synthesize the ALU’s main components: `addSub32`, `lt32`, and `sft32`. Use the same settings.

**Discussion Question:** Report the area, delay, and gate count of the ALU and each component. Based on these results, which of the main components determines the ALU’s critical-path delay? Which component consumes more of the ALU’s area?

In building the full shifter, we emphasized using a single barrel shifter at its core to perform all functions (left/right shifts and arithmetic/logical shifts). Now consider an alternative shifter implementation, \( \text{sft32}_{\text{alt}} \) in ALU.bsv. \( \text{sft32}_{\text{alt}} \) calls your full barrel shifter three times. This will instantiate several copies of the barrel shifter; because the tool performs Boolean simplification, each copy will be optimized for each operation.

Synthesize the \( \text{sft32}_{\text{alt}} \) circuit.

**Discussion Question:** Which shifter implementation takes less area, and which has a shorter delay? Which variant is more appropriate for your ALU? (you can modify your ALU to check, but you don’t need to).

Completing the previous exercises correctly is sufficient to PASS the lab. To get full credit on the lab, finish the exercise below.
8 One Last Design Problem: Building a Better Adder

Exercise 9 (24 points): Processors use several adders, so it pays off to optimize them. Your task is to implement a faster 32-bit adder. Your score depends on how fast you can make the adder. If your adder achieves a critical-path delay $d$,

- For $d > 425\text{ps}$, score = 0
- For $d \in (375\text{ps}, 425\text{ps}]$, score = 5
- For $d \in (325\text{ps}, 375\text{ps}]$, score = 10
- For $d \in (275\text{ps}, 325\text{ps}]$, score = 15
- For $d \in (225\text{ps}, 275\text{ps}]$, score = 20
- For $d \leq 225\text{ps}$, score = 24

You can implement any type of adder you want. Your design can take as much area as needed.

We recommend you try implementing either of the two designs from Lecture 8, a carry-select adder or a carry-lookahead adder. Carry-select adders are easy to code, and a properly designed carry-select adder will be fast enough to earn at least 20 points. Carry-lookahead adders are more complicated to code but are significantly faster—a good carry-lookahead adder should be fast enough to earn full credit.

Implement your fast 32-bit adder in the following skeleton function in ALU.bsv.

```haskell
function Bit#(32) fastAdd32(Bit#(32) a, Bit#(32) b, Bit#(1) c in); // 32-bit fast adder
```

Test your design by running `make fastAdd32 && ./sim_fastAdd32`

Synthesize your design by running `synth ALU.bsv fastAdd32 -l multisize`

9 Appendix A : Bluespec FAQ

- Parametric functions:

  When writing bluespec, we've only been dealing with bit type variables or constants. However, when designing functions with variable bitwidths, the variable that represents the bitwidth is said to be of Numeric Type. Observe the function definition below. The variable $w$ refers to the facts that the length of $a$ and $b$ are parametric and given by $w$. The variable $w$ is inferred from the provided inputs $a$ and $b$ provided to the function, i.e., if variables $a$ and $b$ are defined to be 32-bits wide, using them as inputs in the parametric function call would cause $w$ to be 32.

  An important property to note about numeric type variables is that regular operations don’t apply to them. If we desire to obtain $w+1$, as is the case for the return value’s bitwidth of the function below, we need to use special functions that define these kinds of operators for numeric types. In the case below, we’ve use the $\text{TAdd#}(a, b)$ function to obtain $a + b$.

  ```haskell
  function Bit#(TAdd#(w, 1)) some_parametric_function(Bit#(w) a, Bit#(w) b);
  ```

  Additionally, you may want to use the numeric type variable $w$ in a for loop as the upper bound of the loop. However, numeric type variables cannot be compared against Integer type variables, such as your loop iterator $i$. In order to be able to compare them, we need to convert the numeric type variable into an Integer type variable. To do so, we use the $\text{valueOf}(a)$ function. Hence, the for loop statement can be rewritten as:

  ```haskell
  for( Integer i = 0; i < valueOf(w); i = i + 1) begin
    // Loop body
  end
  ```
• Using \texttt{signExtend(a)} and \texttt{zeroExtend(a)}:

\begin{verbatim}
// Sign extending
Bit#(2) a = 2'b01;
Bit#(4) b = signExtend(a);  // b = 4'b0001;
Bit#(4) c = zeroExtend(a);  // c = 4'b0001;

Bit#(2) d = 2'b10;
Bit#(4) e = signExtend(d);  // e = 4'b1110;
Bit#(4) f = zeroExtend(d);  // f = 4'b0010;
\end{verbatim}

• Using Conditional statements in Bluespec. These statements are all equivalent:

\begin{verbatim}
Bit#(1) ret = 0;

// Using ternary operator
ret = (cond == 1) ? a : b;

// Using if statements
if (cond == 1) ret = a;
else ret = b;

// Add begin/end around the if statement for clarity. It especially helps
// if the conditional’s body is longer than 1 line
if (cond == 1) begin
  ret = a;
end
else begin
  ret = b;
end

// case statement
case(cond)
  1'b0: ret = b;
  1'b1: ret = 1;
endcase
\end{verbatim}

• Bitwise vs Logical Operators:

Logical operators are used when dealing with Boolean variables (True/False conditions). For example:

\begin{verbatim}
if (cond_a && cond_b || cond_c) begin  // (Meaning cond_a and cond_b or cond_c)
  // Do something
end
\end{verbatim}
On the other hand, bitwise operators are used when we apply a given boolean function between every bits of two given variables. For example:

```
Bit#(2) a = 2'b10;
Bit#(2) b = 2'b01;

// Some bitwise operators in bluespec
Bit#(2) c = a & b; // c = 2'b00;
Bit#(2) d = a | b; // d = 2'b11;
Bit#(2) e = a ^ b; // e = 2'b11;
```

## 10 Appendix B: Using the Debug Helper module

We’ve provided you with a series of test cases for all the different functions we’ve asked you to write. However, you might want to write your own test case in order to make sure that your code is correct. In that case, we’ve provided you with a Bluespec module called DebugHelper.bsv. The module makes calls to a function called debug_helper, which takes the following arguments:

```
dependent debug_helper(function to test, expected return value, args for func to test, ...);
```

Therefore, you can write a series of test cases for a given function and ensure that the return value of the function matches the values you expect. As an example, we’ve filled out DebugHelper.bsv with the full adder function:

```
#include "debug_helper.h"
import ALU::*;
module mkDebugHelper(Empty);
rule doTest;
    debug_helper(fa, 2, 1, 1, 0);
    // 2 is the expected value, and 1,1,0 are the arguments to fa()
    $finish();
endrule
endmodule
```

In order to run the test case, type the following:

```
make debug_helper && ./debug_helper
```