Due date: Thursday October 10th 11:59:59pm EST.

Points: This lab is worth 12 points (out of 200 points in 6.004).

Getting started: To create your initial Lab 4 repository, please visit the repository creation page at `https://6004.mit.edu/web/fall19/user/labs/lab4`. Once your repository is created, you can clone it into your workspace by running:

```
git clone git@github.mit.edu:6004-fall19/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. The checkoffs for Lab 4 will be held beginning on Friday October 4th. 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 combinational circuits that implement different operations, then combine them 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 `synth`. 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. However, the last exercise must be completed to receive full credit.

Implementation restrictions: In this lab you will build several circuits for which Minispec already has operators. Therefore, you cannot use these operators in your circuits. Specifically, you are **not allowed** to use the following operators in your logic: Arithmetic operators (`+ - * / % << >>`), relational operators (`<= >= < >`), and variable bit indexing (`a[b]`, where `a` and `b` are both `Bit#(n)` variables).

Bitwise-logical operators (`& | ^ ~`), equality/inequality operators (`== !=`), conditional expressions and statements (`?: if case`), and loops are allowed. You will get an illegal operator error if your circuit, when synthesized, makes use of any forbidden operator.

There is one exception to this rule: **you can use any operators on `Integer` types**. `Integer` expressions are always evaluated during compilation (e.g., in parametric functions, loops, etc.) so operations on `Integers` never produce circuits with these operations. 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 just performs bitwise XOR. (Note that you would not need to write a loop to achieve the same result; you could just write `c = a ^ b;`.)

Minispec resources: We recommend that you complete the Minispec combinational logic tutorial before jumping into the exercises. We especially recommend that you review **Sections 5 and 8 through 10**, which were not needed in lab 3 but are useful for this lab.

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 `Tb_<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`

Finally, you may want to test a particular function that the staff-provided tests do not cover. For example, you may have built your function out of smaller functions; if the whole function is not working properly, you'd want to test the smaller functions first. You can use the `ms eval` command for this purpose. `ms eval` takes two arguments: the file where the function is, and the call to the function in quotes. For example, to test function `barrelRShift` from the first exercise, you can run:

```
ms eval ALU.ms "barrelRShift(32'hBEBECAFE, 4, 1)"
```

Part 1: Designing a Functionally Correct ALU

An arithmetic-logic unit (ALU) is the part of a processor that carries out the arithmetic and logic operations specified by each instruction. 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 8.

Exercise 1 (13%): 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 (for two's complement numbers, this divides by 2^{sftSz}).

Hint: To replicate 1-bit `sft_in` to N bits, use the Minispec function `signExtend`, which extends the input bit by copying its MSB. Section 5 of the Minispec combinational logic tutorial explains `signExtend()` further.

Fill your code in the following skeleton functions in **`ALU.ms`**:

```
// 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 && ./Tb_sr32`

Exercise 2 (6%): Implement a *32-bit Logical Left Shifter*, **`sll32`**, 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` Minispec function does this.

Fill your code in the following skeleton function in **`ALU.ms`**.

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

Test your design by running: `make sll32 && ./Tb_sll32`

Exercise 3 (6%): Implement a *32-bit FULL Shifter*, **`sft32`**, 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.

Note: This function is the first to use an `enum` argument. Enums are covered in Section 9 of the Minispec combinational logic tutorial.

Fill your code in the following skeleton function in **`ALU.ms`**.

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

Test your design by running: `make sft32 && ./Tb_sft32`

2 32-bit Comparator

In these exercises, you will implement a 32-bit comparator for both signed and unsigned numbers. These exercises will leverage your knowledge of two's complement encoding from Lecture 1.

In general, you can build n-bit comparators by using a *chain* or a *tree* of 1-bit comparators. We discuss the chain approach below. Section 9 of the Minispec combinational logic tutorial includes an example of building comparators using a tree approach. The tree approach uses function composition to recursively decide whether `a` is less than, equal to, or greater than `b`. The tree approach is faster, but will require you to write a parametric function. You are free to choose either approach.

The chain approach compares 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.

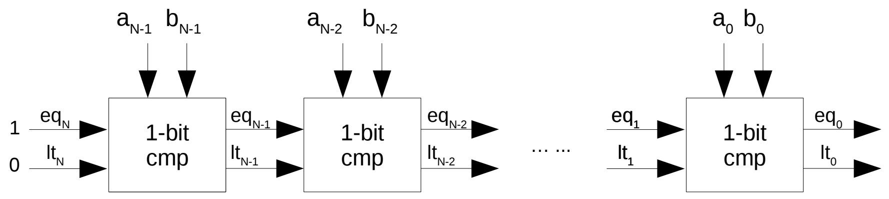

Figure 1: Circuit diagram of an unsigned N-bit less-than comparator.

Exercise 4 (13%): First, implement a *one-bit comparator* **`cmp`** (each of the blocks in Figure 1). Then, use it to construct an *unsigned 32-bit less-than comparator* **`ltu32`**.

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.

Finally, note that it doesn't make sense for eq_i and lt_i to both be 1. Therefore, this is an illegal input and the output in this case is undefined. In other words, you can return any value you like if eq_i and lt_i are both 1.

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.ms**.

```
// one-bit less-than comparator
// Arguments: a, b (1-bit values), eq, lt (eq and lt from previous comparator)
// Return: {eq_i, lt_i}
function Bit#(2) cmp(Bit#(1) a, Bit#(1) b, Bit#(1) eq, Bit#(1) lt);
function Bit#(1) ltu32(Bit#(32) a, Bit#(32) b); // unsigned 32-bit less-than comparator
```

- To test the 1-bit comparator, run `make cmp && ./Tb_cmp`
- To test the unsigned 32-bit less-than comparator, run `make ltu32 && ./Tb_ltu32`

Exercise 5 (6%): Implement a *signed/unsigned 32-bit less-than comparator* **lt32** using at most a **SINGLE** call to the unsigned comparator function you just implemented.

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

$$v_{\text{signed}} = -2^{N-1}a_{N-1} + \sum_{i=0}^{N-2} 2^i a_i$$

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

$$v_{\text{unsigned}} = \sum_{i=0}^{N-1} 2^i a_i = +2^{N-1}a_{N-1} + \sum_{i=0}^{N-2} 2^i a_i$$

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 leftmost comparator. To derive how you should change the inputs to the leftmost 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.ms**.

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

Test your design by running: `make lt32 && ./Tb_lt32`

3 n-bit Ripple-Carry Adder/Subtractor

In these exercises you will implement an n-bit ripple-carry adder, then use it to build a single circuit that performs addition and subtraction.

These exercises are the first that require you to implement *parametric functions*, i.e., functions where the arguments and/our output have generic bit-widths. In this case, bit-widths are given by the *Integer parameter n*. Section 8 of the Minispec combinational logic tutorial covers parametric functions.

Exercise 6 (6%): First, implement a *full adder* circuit **fullAdder**. Then, construct an *n-bit ripple-carry adder* **rca**#(n).

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 in lecture, but in RISC-V, every operand and output has the same width (32 bits for RV32). Because the inputs and output have the same width, adding both operands may cause an overflow.

Fill your code in the following skeleton functions in **ALU.ms**.

```
// full adder
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) carryIn);

// n-bit ripple-carry adder
function Bit#(n) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) carryIn);
```

- To test your full adder, run `make fullAdder && ./Tb_fullAdder`
- To test your ripple-carry adder, run `make rca32 && ./Tb_rca32`

Exercise 7 (6%): Design an *n-bit adder/subtractor* **addSub**#(n), 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 `rca#(n)` function you just implemented.

Hint: Recall that $a - b$ is equivalent to $a + (-b)$. In two's complement representation, $-b$ is equal to flipping all the bits of b and adding 1.

Fill your code in the following skeleton function in **ALU.ms**.

```
// n-bit ripple-carry adder/subtractor
// returns a - b is isSub==1 and a + b otherwise
function Bit#(n) addSub#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) isSub);
```

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

Exercise 8 (15%): Implement a *32-bit ALU*, **alu**, for the RISC-V processor. Your implementation should only perform a **SINGLE** call to each of the circuits you implemented in the previous exercises: the adder/subtractor (`addSub`), 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 following skeleton function in **ALU.ms**.

```
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra} AluFunc;
function Bit#(32) alu(Bit#(32) a, Bit#(32) b, AluFunc func);
```

Test your design by running `make alu && ./Tb_alu`

ALU Function	Operation	Output
`Add`	32-bit ADD	`a + b`
`Sub`	32-bit SUBTRACT	`a - b`
`And`	32-bit AND	`a & b`
`Or`	32-bit OR	`a | b`
`Xor`	32-bit XOR	`a ^ b`
`Slt`	Set if less than signed	a $<_s$ b? 1 : 0
`Sltu`	Set if less than unsigned	a $<_u$ b? 1 : 0
`Sll`	SHIFT Left Logical	`a << b[4:0]`
`Srl`	SHIFT Right Logical	a $>>_u$ b[4:0]
`Sra`	SHIFT Right Arithmetic	a $>>_s$ b[4:0]

Table 1: RISC-V ALU functions

Part 2: Analyzing and Improving Circuit Performance

So far we have focused on describing circuits in Minispec 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. Like in lab 3, we will use the `synth` synthesis tool to translate your Minispec code into optimized gate-level implementations.

In this part of the lab, we will first synthesize and analyze 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; for a quick reference, run `synth -h` .

5 Analyzing the Ripple-Carry Adder

First, synthesize your n-bit ripple-carry adder function with $n = 4$ bits, i.e., function `rca#(4)`, by running:

```
synth ALU.ms "rca#(4)"
```

`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 ALU.ms "rca#(4)" -v
```

This will produce a diagram in `rca_4.svg`. Open by running `inkview rca_4.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. (*Note* that that's the lowercase letter L, not the digit 1.) 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 ALU.ms "rca#(4)" -l extended
```

Warm-up Question 1: How does `rca#(4)` 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 ALU.ms "rca#(4)" -l extended -v && inkview rca_4.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 ALU.ms "rca#(4)" -l extended -d 1000
```

Warm-up Question 2: Synthesize `rca#(4)` using the command above. How do its area and delay change vs. the previous (delay-optimized) circuit?

`synth` also performs some optimizations, such as Boolean simplification. Take a look at the `add4_rca` and `add4_addSub` functions in Adders.ms. Both implement the same function, adding two 4-bit numbers. However, `add4_rca` uses your `rca#(4)` circuit with the carry-in argument set to 0, whereas `add4_addSub` uses your `addSub#(4)` circuit with `isSub` set to 0.

Warm-up Question 3: Synthesize `add4_rca` and `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.

Warm-up Question 4: Synthesize your `rca#(n)` function with $n \in \{4, 8, 16, 32, 64\}$. How do you intuitively expect the area and delay of the circuit to grow as you increase the number of bits in the input? Do the results match your expectation?

Discussion Questions

The following Discussion Questions are worth **5%** of your grade. Please write your answers in the provided file `discussion_questions.txt`. You can update your answers before your checkoff meeting, but you are required to submit an initial answer to each question when you submit the lab.

6 Multiplexers and Fanout

Muxes.ms 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.

Warm-up Question 5: Synthesize the above functions. Why don't their implementations meaningfully 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). 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]
         Gate/port  Fanout  Gate delay (ps)  Cumulative delay (ps)
         ---------  ------  ---------------  ---------------------
  binary_number[3]       3              7.6                    7.6
               INV       7             18.9                   26.5
             NAND2       2             12.2                   38.7
              NOR3       5             43.4                   82.1
              NOR2       3             13.4                   95.5
             NAND3       5             23.0                  118.5
             NAND2       1              8.7                  127.2
            out[0]       0              0.0                  127.2
```

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 (and we will see in more detail in Lecture 9), in current technology (CMOS), 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 affects 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.ms has a parametric n-bit multiplexer `mux#(n)`. Synthesize `mux#(n)` for $n \in \{4, 16, 64, 256\}$ *using a really high delay target*, e.g., 10000 ps, like so:

```
synth Muxes.ms "mux#(64)" -l extended -d 10000
```

Discussion Question 1 (1%): How do you intuitively expect the delay of the circuit to grow as you increase the number of bits in the input? Does the result match your expectations? 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 `mux#(n)` for $n \in \{4, 16, 64, 256\}$ again, but this time do not specify a target delay so that the muxes are delay-optimized instead of area-optimized, like so:

```
synth Muxes.ms "mux#(64)" -l extended
```

Discussion Question 2 (1%): How do you expect the delay and area to grow as you increase the width for these delay-optimized multiplexers? Do the results match your expectations?

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.

`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 `mux#(64)` with the `multisize` library, like so:

```
synth Muxes.ms "mux#(64)" -l multisize
```

Discussion Question 3 (1%): How do `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:

```
synth ALU.ms alu -l multisize
```

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

Discussion Question 4 (1%): 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, `sft32_alt` in ALU.ms. `sft32_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 `sft32_alt` circuit.

Discussion Question 5 (1%): 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%): 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 $d > 400$ ps, score $= 0$
- For $d \in (350\text{ ps}, 400\text{ ps}]$, score $= 5$
- For $d \in (300\text{ ps}, 350\text{ ps}]$, score $= 10$
- For $d \in (250\text{ ps}, 300\text{ ps}]$, score $= 15$
- For $d \in (200\text{ ps}, 250\text{ ps}]$, score $= 20$
- For $d \leq 200$ 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 recursive carry-select adder will be fast enough to earn full credit. Carry-lookahead adders are more sophisticated but are faster—any good carry-lookahead adder should be fast enough to earn full credit.

Implement your fast adder in the following skeleton function in **`ALU.ms`**.

```
// N-bit fast adder
function Bit#(n) fastAdd#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) carryIn);
```

Test your design by running `make fastAdd32 && ./Tb_fastAdd32`
Synthesize your design by running `synth ALU.ms "fastAdd#(32)" -l multisize`