Due date: Thursday October 31st 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 5 repository, please visit the repository creation page at `https://6004.mit.edu/web/fall19/user/labs/lab5`. Once your repository is created, you can clone it into Athena by running:

```
git clone git@github.mit.edu:6004-fall19/labs-lab5-{YourMITUsername}.git lab5
```

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 (`https://6004.mit.edu`, Labs/Didit tab) to verify that your submission was correctly pushed and passes all the tests.** If you finish the lab in time but forget to push, you will incur the standard penalties for late submissions.

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 (by Wednesday Nov 6th). The checkoffs for Lab 5 will be held beginning on Friday October 25th. See the course website for lab hours.

Introduction

In this lab you are asked to build two *sequential circuits* in Minispec: a multiplier and a sorting network. Each of them is structured very differently, so that we can explore the different advantages of sequential logic: the multiplier is a *multi-cycle circuit* that computes a single result over multiple cycles to reduce the amount of hardware needed over a combinational implementation, whereas the sorting network is *pipelined* to achieve high throughput.

To pass the lab you must complete and PASS all of the exercises except Exercise 3 (Fast Folded Multiplier). However, all of the exercises must be completed to receive full credit.

Coding guidelines: You are only allowed to make changes to `Multipliers.ms` and `SortingNetworks.ms`. Modifications to other source files will be overwritten during Didit grading.

Discussion questions: The discussion questions in this lab are worth **10%** of your grade. Please write your answers in the `discussion_questions.txt` file. You can update your answers before your checkoff meeting, but you must submit an initial answer to each question when you submit the lab. You should be prepared to explain your answers during the checkoff.

Implementation restrictions: In this lab you will build some circuits for which Minispec already has operators. You cannot use these operators in your circuits. **Specifically, you are NOT ALLOWED to use the following operators in your logic: `*` `/` `%` (multiplication, division, modulus).** Unlike in lab 4, addition and subtraction (`+ -`), shifts (`<< >>`), and comparisons (`<= >= < >`) are allowed. Like in lab 4, bitwise-logical operators (`& | ^ ~`), equality/inequality operators (`== !=`), conditional expressions (`?:` `if`), and loops are also allowed.

Minispec resources: In Lecture 11, we learned how to describe sequential circuits in Minispec using modules, methods, and rules. We also recommend that you complete the Minispec sequential logic tutorial before jumping into the exercises. We especially recommend that you review **Sections 1, 2, and 8**. Sections 1 and 2 cover the basics, and Section 8 reviews how to design good interfaces and the use of Maybe types.

1 Multiplication by Repeated Addition

Multiplication can be calculated by a series of additions, similarly to how we perform multiplication by hand. Multiplication can be implemented with combinational and sequential circuits. Let's explore both styles.

1.1 Combinational Multiplier

In Lecture 12 we saw how to implement multiplication by repeated addition. Figure 1 illustrates this procedure for two 4-bit numbers, 13 and 11.

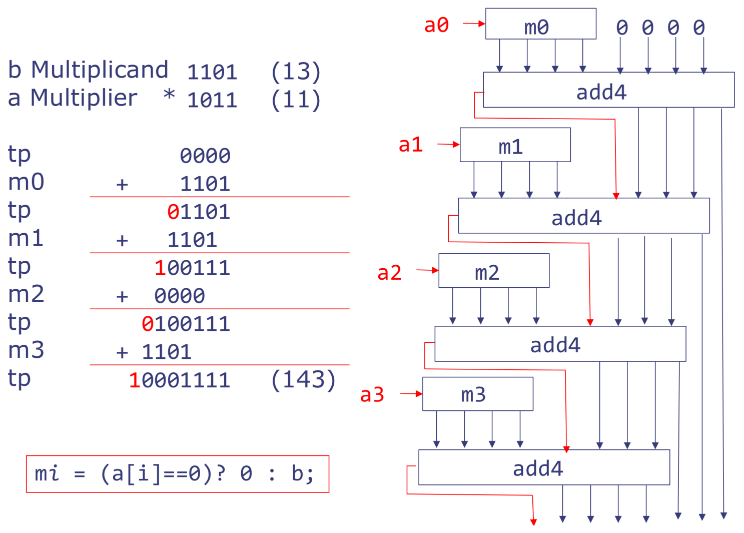


Figure 1: Implementation of multiplication by repeated addition.

In general, multiplication by repeated addition of two n-bit numbers produces a 2n-bit output. Multiplication begins by producing n partial products (the `mi` boxes above), which result from multiplying the multiplicand with each bit of the multiplier (in binary, multiplication by a single bit is just AND, hence the partial products are very simple to compute). Then, the multiplier uses n n-bit adders to compute the result from the n partial products.

Exercise 1 (10%): Implement the combinational function `multiply_by_adding` in `Multipliers.ms`, which multiplies two unsigned n-bit numbers.

```
// Multiplication by repeated addition
function Bit#(2*n) multiply_by_adding(Bit#(n) a, Bit#(n) b);
```

To compile and test your design, run `ms sim MultiplierTB.ms MultiplyByAddingTest`

Remember that to run your own test cases you can use `ms eval`. For example, to test the output of `3 * 5` on a 4-bit multiplier, you could run `ms eval Multipliers.ms "multiply_by_adding#(4)(3, 5)"`

1.2 Sequential, Multi-Cycle Multiplier

Next, let's build a sequential circuit for multiplication by repeated addition that takes multiple cycles (also known as a folded multiplier). The inputs and methods of the module are given below:

```
typedef struct { Bit#(n) a; Bit#(n) b; } MultiplierInput#(Integer n);

module FoldedMultiplier#(Integer n);
  input Maybe#(MultiplierInput#(n)) in default = Invalid;
  method Maybe#(Bit#(2*n)) getResult;
  ...
endmodule
```

To pass an input to a multiplier `m`, the user of `m` would set its input to a valid value, `m.in = Valid(value)`, where value is a `MultiplierInput` with the two operands to multiply. Similarly, when the multiplier is done calculating the product, calling `m.getResult` should return `Valid(product)`. If the Multiplier is still calculating the product or has never been given an input, then calling `m.getResult` should return `Invalid`.

You may find the functions `isValid` and `fromMaybe` helpful to check whether a Maybe is valid and to extract its value, respectively.

See Section 8 of the Minispec sequential logic tutorial for more on Maybe types and interfaces of multi-cycle sequential circuits.

Exercise 2 (30%): Implement the multiplier by completing the `FoldedMultiplier` module in **Multipliers.ms**. This includes defining any internal state, implementing the `getResult` method, and implementing a rule that updates the circuit's state.

To test your design, run `ms sim MultiplierTB.ms FoldedMultiplierTest`

This testbench runs many multiplications consecutively. If you would like to run a single multiplication operation, we have provided both 8-bit testbench that tests your `FoldedMultiplier#(8)` on a single custom input. To run this testbench on, say `3*10`, run `ms sim MultiplierTB.ms "FoldedMultiplierCustomTest#(3,10)"`. Remember that, to debug your own module, you can also add your own `$display` statements to the rule within the module. Section 7 of the Minispec sequential logic tutorial gives more information on `$display` and debugging of modules.
Note: If you get a failure message saying "FAILED due to cycle count", you are taking too many cycles to do your computation. It should take n cycles to compute the product of two n-bit numbers.

Discussion Question 1 (2%): Does the repeated addition algorithm work for multiplying two numbers in two's complement encoding? Why or why not?

You can test your intuition by running:

```
ms sim MultiplierTB.ms MultiplySignedTest
```

which tests your combinational multiply_by_adding to Minispec's built-in signed multiply. Do the outputs match? If not, suggest how you might make your implementation work with multiplying signed numbers. (*Note:* You do not have to actually implement this signed multiplier.)

1.3 Analyzing the Combinational and Sequential Multipliers

Now we want you to analyze the performance of your combinational and sequential multipliers in terms of area, critical-path delay, latency, and throughput. Please use the standard definitions for *latency* and *throughput* from Lecture 12. Like in Lab 4, we will use `synth` for this purpose.

Synthesize `multiply_by_adding#(8)` and `FoldedMultiplier#(8)` by running:

```
synth Multipliers.ms "multiply_by_adding#(8)" -l multisize
synth Multipliers.ms "FoldedMultiplier#(8)" -l multisize
```

Discussion Question 2 (1%): How do the delay and area of the sequential multiplier compare with those of the combinational one? How many cycles does the sequential multiplier take to calculate a result? Given these, which of the two multipliers has a lower latency? (i.e., from input to result, which one can calculate a product faster?)

Now synthesize `multiply_by_adding#(X)` for $X \in \{4, 8, 16, 32, 64\}$.

Discussion Question 3 (1%): For the combinational multipliers, how does critical-path delay grow with the number of bits of the operands? How does area grow with the number of bits of the operands? Use order-of notation.

Now synthesize `FoldedMultiplier#(X)` for $X \in \{4, 8, 16, 32, 64\}$).

Discussion Question 4 (1%): For the folded sequential multipliers, how does critical-path delay grow with the number of bits of the operands? How does area grow with the number of bits of the operands? Use order-of notation.

1.4 Building a Faster Sequential Multiplier

The following exercise is not necessary to pass the lab, but it is necessary to receive full credit.

Exercise 3 (15%): Fill in the skeleton code for the module `FastFoldedMultiplier` in **Multipliers.ms**, such that your 32-bit sequential multiplier achieves a critical-path delay $\leq 280ps$.

Hint: There are at least two ways to speed up your multiplier. First, you can use a faster adder (e.g., the one from Lab 4). Second, you can try a multiplier algorithm that avoids dynamic bit selection and shifting by a variable number of bits. For example, you may have used an algorithm in the folded multiplier exercise that required the number to be added in the i^{th} step to depend upon the i^{th} bit of operand a. If you implemented this using dynamic bit selection ($a[i]$), it would require a significant number of gates (it requires an n-input multiplexer). It is possible to replace this dynamic selection by a simpler one-bit shift at each step.

To test your design, run `ms sim MultiplierTB.ms FastFoldedMultiplierTest`

To synthesize your design, run `synth Multipliers.ms "FastFoldedMultiplier#(32)" -l multisize`

2 Sorting Network

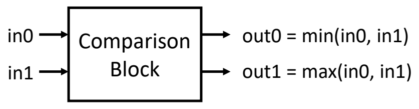
In labs 1 and 2, we explored different implementations of two different sorting algorithms in RISC-V assembly, bubblesort and quicksort. Even in efficient algorithms like quicksort, each comparison and potential swap in the sort took multiple instructions. Each instruction takes at least one cycle in the processors you'll build later, so sorting in software will take multiple cycles per element.

We'll now see how to speed up sorting in hardware, building circuits that sort *multiple elements per cycle*. The design project will be about optimizing a processor for sorting, so you'll have a chance to reuse the hardware you build here as a special functional unit in your processor—a sorting accelerator.

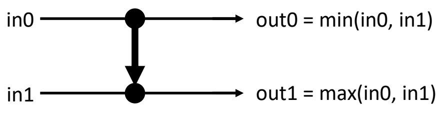
2.1 Combinational Sorting Network

A *sorting network* is a way to sort a group of values in parallel. Sorting networks can be implemented easily in hardware using wires and comparators.

A *comparison block*, shown in Figure 2a, is the basic building block of a sorting network. A comparison block is a combinational circuit that takes two elements as inputs and outputs them sorted, with the smaller one on the top output and the larger one on the bottom output. You can build this circuit with a comparator and two muxes.



(a) Combinational comparison block



(b) Equivalent circuit diagram

Figure 2: A comparison block sorts two input elements.

A comparison block itself is a 2-element sorting network. We can build larger sorting networks by combining comparison blocks. To make the circuit diagrams easier to understand, sorting networks represent a comparison block as an arrow between two wires, as shown in Figure 2b.

There are many ways to build a sorting network. In this exercise we'll build a **bitonic sorting network**, an efficient and easy-to-implement network that is based on the mergesort algorithm.

Bitonic sorting networks follow a simple recursive construction. An n-element network has three main blocks, shown in Figure 3. First, it sorts the top and bottom halves of the input (in parallel). Then, it

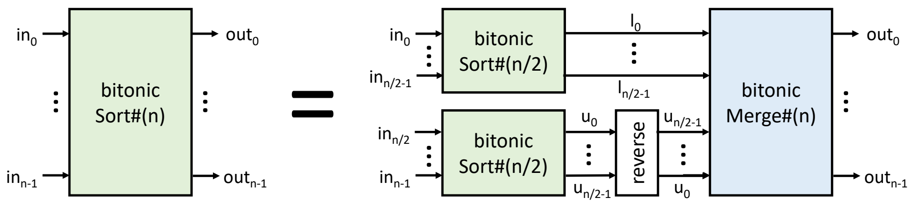


Figure 3: n-element bitonic sorting network.

reverses the bottom half (this is just wires), so that the top half is sorted from small-to-large and the bottom half is sorted from large-to-small. Finally, it merges both sorted halves.

The reversing of the bottom half makes the merge step very regular. The input to the merge block is called a *bitonic* sequence. This name comes from the fact that both the top and bottom parts of the sequence are monotonic, but both have different directions (e.g., top half is ascending, bottom half is descending).

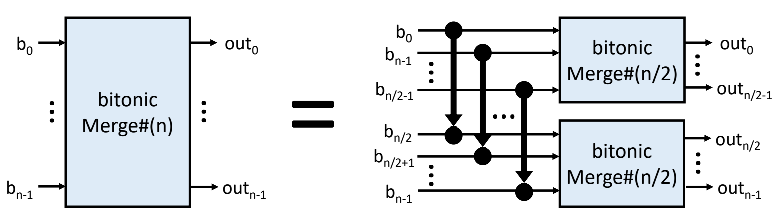


Figure 4: n-element bitonic merge.

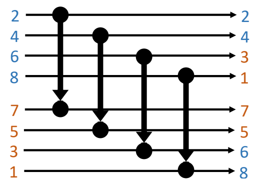


Figure 5: Example showing the first step of an 8-element merge.

Merging a bitonic sequence can be done recursively, as shown in Figure 4. For an n-element merge, the merge circuits first features a group of $n/2$ comparison blocks, where each block compares elements i and $i + n/2$ (i.e., the corresponding elements in the top and bottom halves) for $i \in [0, ..., n/2 - 1]$. Then, the circuit merges the resulting top and bottom halves.

This merging circuit works because the first step with $n/2$ comparator blocks has two properties when the input sequence is bitonic. First, the outputs in the top half are smaller than the outputs in the bottom half. This is because the inputs in the bottom half are increasing and the outputs in the bottom half are decreasing, so when they cross over, the comparators start sending the smaller numbers of the bottom half of the input to the top half of the output, and vice versa. Second, both the top and bottom halves are bitonic, so the merge can be recursively applied. Figure 5 shows an example of an 8-element merge with specific input values (the bitonic sequence 2, 4, 6, 8, 7, 5, 3, 1) that shows both properties.

Let's apply this procedure to build an 8-element combinational bitonic sorting network. First, the 2-element network is just a single comparator block. Then, the 4-element network is two 2-element sorts, a reverse, and a 4-element merge, as shown in Figure 6. Note how the 4-element merge is just the merge step we showed above, for n=4, followed by merges of the upper and lower halves. Finally, the 8-element sorting network is two 4-element sorts, a reverse, and an 8-element merge, as shown in Figure 7.

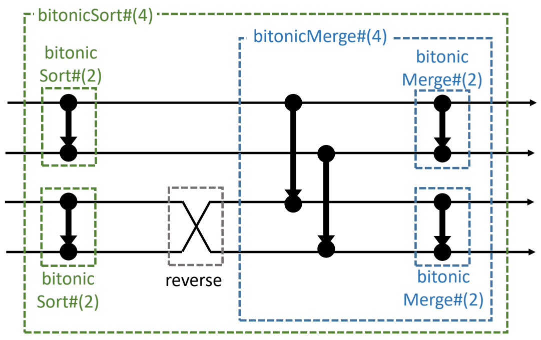


Figure 6: 4-element bitonic sorting network.

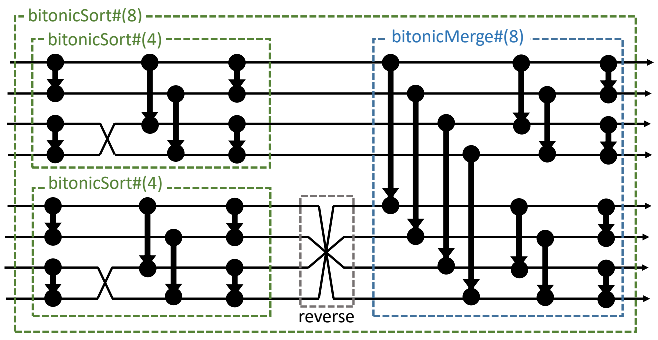


Figure 7: 8-element bitonic sorting network.

The **SortingNetworks.ms** file already implements a combinational bitonic sorting network: the Minispec function `bitonicSort#(n)` takes an n-element vector of 32-bit values, and returns a sorted n-element vector of these values. `bitonicSort#(n)` itself uses `bitonicMerge#(n)`, which implements the recursive merge.

Take a look at these functions and compare them with the diagrams in Figure 3 and Figure 4. For simplicity, these functions use `Vector` inputs and outputs. Vector is a built-in type that denotes a group of values of the same type. Although Vectors, like all other types, are just a bunch of wires in the end, they are clearer to use than a `Bit#(n)` variable. For example, `Vector#(4, Bit#(32)) v` is a 4-element vector of 32-bit variables, so it takes 128 bits. We could avoid using vectors at all and use a `Bit#(128) v` instead, but this would be cumbersome. For example, in the first case (i.e., with a vector), we can obtain element 1 with `v[1]`. But in the second case (i.e., with a `Bit#(128)` variable), we'd need to type `v[63:32]` to get element 1.

The `bitonicSort#(n)` and `bitonicMerge#(n)` functions use a few built-in functions to manipulate vectors. The comments in **SortingNetworks.ms** detail what each of the built-in functions does. Section 6.4 of the Minispec reference has more information on vectors.

2.2 Pipelining the Sorting Network

The combinational implementation above has a problem: it's too slow! It has many comparators between the inputs and outputs, so its propagation delay is quite high. Your job is to solve this by *pipelining* the network, so that each pipeline stage has only *one* comparator between its input and output. Figure 8 shows this pipeline for an 8-element sorting network—a 6-stage pipeline.

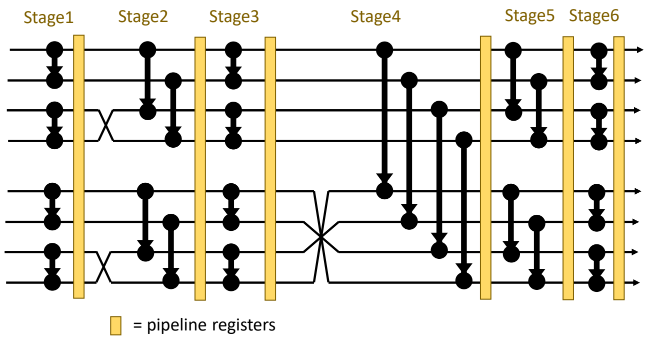


Figure 8: 6-stage pipelined 8-element bitonic sorting network.

Note: Pipelining doesn't help if we only need to sort one array of 8 elements. However, it is helpful for large inputs because we can break a sort of a large array into many smaller, independent sorts, which can be fed into our circuit on successive clock cycles.

Exercise 4 (35%): Implement the pipelined 8-element sorting network above by completing the skeleton code for the `BitonicSorter8` module in **SortingNetworks.ms**. Your implementation should receive a vector of *unsigned* 32-bit numbers at its input, and return them sorted at the output 6 cycles later. The input and output are both Maybe values, so if the module user feeds an Invalid input, an Invalid output should come out 6 cycles later.

Hint: There are at least three ways to implement this pipeline. We'll call them the per-stage way, the bottom-up way, and the general way:

1. The per-stage way is to code a separate function for each stage. Then we can implement the module by having a bunch of pipeline registers with the same type as the input, and a rule that does: `pipeReg1 <= stage1(in); pipeReg2 <= stage2(pipeReg1);`, etc. This approach can result in a quite a lot of repetitive code that is easy to mess up and hard to debug. You can save some code by recognizing similarities across stages (e.g., stages 1, 3, and 6 are identical), and since this is a small pipeline, building it stage by stage can be done with a reasonable amount of code. However, there's still a fair amount of repetition and it's hard to build larger pipelines this way. Thus, we don't recommend this approach.

2. The bottom-up way is to build up the 8-element sorter by composing smaller modules. Writing a couple of intermediate modules will save you most repetitive work: first `BitonicMerger4` to implement a pipelined 4-way merge, then `BitonicSorter4` to implement a pipelined 4-way sort. You can then implement `BitonicSorter8` using both modules as submodules to avoid most repetition.

Figure 9 shows this approach by breaking the pipeline into modules. Note how each module in Figure 9 follows the pipeline rules from Lecture 12 (a K-pipe has registers at the outputs and always has K registers between inputs and outputs). This makes the pipelines easy to compose.

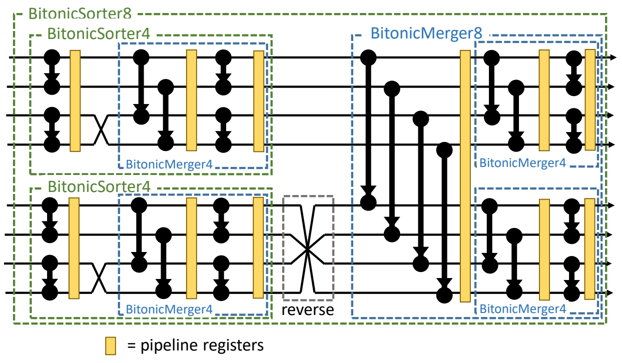


Figure 9: 6-stage pipelined 8-element bitonic sorting network built from smaller modules.

3. The general way is to use parametric modules and recurse on the parameters, e.g., `BitonicSorter#(n)` and `BitonicMerger#(n)`, that mirror the structure of the `bitonicSort#(n)` and `bitonicMerge#(n)` functions, using smaller bitonic sorters and mergers as submodules. You'll also need to define trivial base-case modules `BitonicSorter#(2)` and `BitonicMerger#(2)` to stop the recursion. Then, you can implement `BitonicSorter8` as a trivial wrapper of `BitonicSorter#(8)`.

This approach has two advantages. First, it has no repetitive code and there are fewer opportunities to make mistakes. Second, you get a parametric implementation, so you can build wider sorting networks, which may be helpful in the design project. However, this approach requires that you are comfortable with recursion and understand how parametric modules work.

To test your design, run

```
ms sim SortingNetworkTB.ms BitonicSorter8Test
```

To run all tests against your design with additional output for debugging, run

```
ms sim SortingNetworkTB.ms BitonicSorter8DebugTest
```

To synthesize your design, run

```
synth SortingNetworks.ms BitonicSorter8 -l multisize
```

2.3 Analyzing the Combinational and Pipelined Sorting Networks

Finally, let's analyze and compare the performance and cost of our sorting networks. Please use the standard definitions for *latency* and *throughput* from Lecture 12.

First, synthesize the combinational sorting network `bitonicSort#(X)` for $X \in \{2, 4, 8, 16\}$, like so:

```
synth SortingNetworks.ms "bitonicSort#(4)" -l multisize
```

Discussion Question 5 (1%): How do you intuitively expect the *latency* of the circuit to grow as you increase the number of elements in the input? Does the result match your expectations?

Discussion Question 6 (1%): How do you intuitively expect the *area* of the circuit to grow as you increase the number of elements in the input? Does the result match your expectations?

Now, synthesize the combinational and pipelined 8-element sorting networks:

```
synth SortingNetworks.ms "bitonicSort#(8)" -l multisize
synth SortingNetworks.ms BitonicSorter8 -l multisize
```

Discussion Question 7 (2%): What are the *latency* and *throughput* of both implementations? If you wanted to minimize latency, which one would you choose? How about if you wanted to maximize throughput?

Discussion Question 8 (1%): We are considering using an 8-element sorting network for a low-power processor that runs at 500 MHz. We primarily want to maximize throughput, but also to reduce area and latency. Which of the two sorting networks would you choose, and why?