Due date:  **Tuesday** April 21st 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 [https://6004.mit.edu/web/spring20/user/labs/lab5](https://6004.mit.edu/web/spring20/user/labs/lab5). Once your repository has been created, you can clone it into Athena by running:

```
git clone git@github.mit.edu:6004-spring20/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](https://6004.mit.edu)) to verify that your submission was correctly pushed and passes all expected 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 do an on-line check-off meeting by Monday April 27th. The checkoffs for this lab will begin on Monday, April 6th. See the course website for check-off hours/instructions.

To be able to checkoff this lab you must complete all of the exercises except Exercise 3 (Fast Folded Multiplier). You must pass all of the “Mandatory” tests in Didit. Exercise 3 is optional but must be completed to receive full credit.

### Introduction

In this lab you are asked to build three *sequential circuits* in Minispec: a multiplier, an arithmetic pipeline and a sorting network. Each of them is structured 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 versus a combinational implementation. The next two circuits are *pipelined*, a technique used to increase throughput and allow higher clock frequencies. The arithmetic pipeline is a simple circuit that will introduce you to the concepts. The sorting network is a more complex example that explores different ways to implement pipelined circuits in Minispec.

Coding guidelines:  You are only allowed to make changes to *Multiplier.ms*, *PipelineMath.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.

Building and testing:  You can easily build all code for this lab using `make all` and run all tests using `./test_mandatory.sh` and `./test_optional.sh`. Use the commands in each section for more detailed output.
1 Multiplication by Repeated Addition

Multiplication can be calculated by a series of additions, similar to how we perform multiplication by hand. Multiplication can be implemented with combinational or 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.](image)

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.

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

To compile and test your design, run `make multiply_by_adding_test`
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:

```verilog
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, \( m.\text{in} = \text{Valid}(\text{value}) \), where "value" is a \text{MultiplierInput} with the two operands to multiply. Similarly, when the multiplier is done calculating the product, calling \( m.\text{getResult} \) should return \text{Valid}(\text{product}). If the Multiplier is still calculating the product or has never been given an input, then calling \( m.\text{getResult} \) should return \text{Invalid}. You may find the functions \text{isValid} and \text{fromMaybe} helpful to check whether a \text{Maybe} is valid and to extract its value, respectively. See Appendix A.1 for a quick reference to \text{Maybe} types. See Section 8 of the Minispec sequential logic tutorial for more on \text{Maybe} types and interfaces of multi-cycle sequential circuits.

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

To test your design, run: `make folded_multiplier_test`

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 \text{FoldedMultiplier\#(8)} on a single custom input. To run this testbench on, say 3*10, run `ms sim TBMultiplier.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.

**Discussion Question 1 (1%)**: 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 TBMultiplier.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. As in Lab 4, we will use synth for this purpose.

Synthesize \text{multiply\_by\_adding\#(8)} and \text{FoldedMultiplier\#(8)} by running:

```
synth Multipliers.ms \text{"multiply\_by\_adding\#(8)"} -l multisize
synth Multipliers.ms \text{"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 \text{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 ∈ {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 ≤ 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 the functionality of your design, run:

```
make fast_folded_multiplier_test
```

To check the critical-path delay, synthesize your design:

```
synth Multipliers.ms "FastFoldedMultiplier#(32)" -l multisize
```

### 2 Pipelining Math

In software, we may not be especially concerned about how long certain arithmetic operations take. For example, unless we are writing extremely time-sensitive code, we may not consider how long it takes to perform an addition versus a multiplication or division. However, the same is not true in hardware.

As you have seen in Lab 4 and Exercise 1 of this lab, the time it takes to perform different operations can vary significantly. For example, a 32-bit combinational adder will be faster than a 32-bit combinational multiplier, which will be faster than a 32-bit combinational divider. The time each operation takes generally correlates to its complexity. Now suppose that we want to calculate an expression like $((|a| ÷ 3) + 10) \times 7$.

In hardware, we might chain together combinational blocks like this:

![Figure 2: A hardware diagram of the expression. Here, abs() represents the absolute value function.](image)

Each operation within the expression is calculated by a separate hardware block that takes a different amount of time. The time to compute the entire expression will be the sum of the delays on the longest path as the data must flow through all of them before the calculation is complete.

### 2.1 Analyzing Delay of Combinational Components

We have provided you with a number of functions in MathFunctions.ms to help you perform this computation. Now, we want to analyze the performance of each component in terms of area and critical-path delay.
As in Exercise 1.3, we will use synth for this purpose.

Synthesize abs, add10, divide3, and multiply7 by running:

```
synth MathFunctions.ms abs -l multisize
synth MathFunctions.ms add10 -l multisize
synth MathFunctions.ms divide3 -l multisize
synth MathFunctions.ms multiply7 -l multisize
```

**Discussion Question 5 (1%):** Compare the area and delay of each of the combinational circuits. Which components have the longest critical-path delay? What would be the overall propagation delay if we implemented this as a single combinational function?

2.2 Pipelining the Computation

When building special-purpose hardware, it is common to calculate an expression like the one above on every element of an array or on a continuous stream of incoming values. In this situation, we can increase the overall throughput by pipelining the circuit and allowing the computations for multiple elements to overlap in time. Pipelining can also help divide up a lengthy computation into smaller pieces if we have already chosen a clock period for our device and need to make things fit.

Suppose that we plan to use this circuit in a processor with a 500 ps clock period. As we saw above, we cannot perform this entire computation within 500 ps. However, now that you know the timing of individual components, you can decide how to divide them up between pipeline stages so that each stage is less than 500 ps. You may choose a 1-stage, 2-stage or 3-stage pipeline as needed to make sure your circuit works with a 500 ps clock period.

**Exercise 4 (10%):** Implement a pipeline to calculate the expression \(((|a| \div 3) + 10) \times 7\) by completing the rule pipeline in `PipelineMath.ms` using the components provided in `MathFunctions.ms`. Your implementation should use at most three registers and achieve a critical-path delay \(\leq 500\) ps.

**Notes:**
- Your first stage should take the input, partially compute a result and store that value in a pipeline register. Subsequent stages should take the value from the preceeding pipeline register, compute more of the expression, and store the new result in the next pipeline register.
- Your pipeline should continue to operate and allow in-progress computations to proceed even when it receives an invalid input. When you receive a valid input, you should produce the answer exactly \(k\) cycles later (where \(k\) is the number of stages in your pipeline). When you receive an invalid input, you should produce an invalid output exactly \(k\) cycles later. To achieve this, every pipeline register you use should be assigned something every time the rule in your code fires (once per cycle).
- Although not required, you may want to implement a 1-stage pipeline first and verify that your functionality is correct, then divide your code into multiple stages until you meet the timing requirement.
- Remember that, by our 6.004 convention, all pipelines should have a register at the output. In the code skeleton, you can see this because the output method gets its value directly from the result register. Do not change this.
- You must check both the functionality and the critical path of your code using separate steps below. Getting “PASSED” from the functionality test does not mean that your critical path is OK. There are separate tests for these two things in Didit.

To test the functionality of your design, run:

```
make pipeline_math_test
```

To check the critical-path delay, synthesize your design:

```
synth PipelineMath.ms PipelineMath -l multisize
```
3 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.

3.1 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 3a, 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.

![Comparison Block](image)

(b) Equivalent circuit diagram

Figure 3: 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 3b.

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 Network](image)

Figure 4: n-element bitonic sorting network.

Bitonic sorting networks follow a simple recursive construction. An n-element network has three main blocks, shown in Figure 4. First, it sorts the top and bottom halves of the input (in parallel). Then, it 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).

Merging a bitonic sequence can be done recursively, as shown in Figure 5. 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...
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. In fact, both the 2-element sort and the 2-element merge are just the single comparator block—they are the same. Then, the 4-element network is two 2-element sorts, a reverse, and a 4-element merge, as shown in Figure 7. 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 8.

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 4 and Figure 5. 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 clumsier. 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.

3.2 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 9 shows this pipeline for an 8-element sorting network—a 6-stage pipeline.
Exercise 5 (25%): 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. Verify that you have only one comparator in each stage by synthesizing and checking that your critical path is < 250ps.

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. The main advantage of this approach is that it is the most straightforward conceptually. However, the disadvantages of this method are that it contains quite a lot of repetition and is also hard to build larger pipelines. Repetitive code is often 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.

2. The bottom-up way is to build up the 8-element sorter by composing smaller modules. While the per-stage way contains a lot of repetitive code, 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. This approach is somewhat of a middle ground between the general way and the per-stage way - there’s less repetition than the per-stage way, but the lack of parameters makes it harder than the general way in building wider networks.

Figure 10 shows this approach by breaking the pipeline into modules. Note how each module in Figure 10 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.

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.
3.3 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 \( \text{bitonicSort#}(X) \) for \( X \in \{2, 4, 8, 16\} \), like so:

\[
\text{synth SortingNetworks.ms "bitonicSort#(4)" -l multisize}
\]

**Discussion Question 6 (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?

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

\[
\text{synth SortingNetworks.ms "bitonicSort#(8)" -l multisize}
\]
\[
\text{synth SortingNetworks.ms BitonicSorter8 -l multisize}
\]

**Discussion Question 8 (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 9 (1%)**: We are considering using an 8-element sorting network for a low-power processor that runs at 500 MHz. We want to maximize throughput first, then minimize area and latency if possible. Which of the two sorting networks would you choose, and why?
A Appendix A : Related Minispec Reference Excerpts

A.1 Maybe#(T)

Maybe#(T) represents an optional value of type T. A Maybe#(T) can be either Valid if it holds a value of type T, or Invalid if it does not hold a value. Maybe#(T) is especially useful for modules, which often do not have valid inputs or outputs every cycle.

Creating Maybe#(T) values: Given a value v of type T, Valid(v) is a valid Maybe#(T) that holds v. The literal Invalid can be assigned to any Maybe#(T) variable to make it invalid.

Checking for validity: The built-in function isValid returns True if its argument is Valid, and False if it is Invalid.

Unpacking Maybe#(T)’s optional value: The built-in function fromMaybe allows extracting the value of a valid Maybe value. Its signature is T fromMaybe(T defaultValue, Maybe#(T) x). If x is Valid, fromMaybe returns x’s value; if x is Invalid, fromMaybe returns defaultValue.

```verbatim
// Creating Maybe#(T)'s
Maybe#(Bit#(1)) a = Valid(1);
Maybe#(Bit#(4)) b = Invalid;

// Validity checking
Bool aValid = isValid(a);

// Unpacking Maybe#(T)'s
let x = fromMaybe(0, a); // 1
let y = fromMaybe(4, b); // 4

// Common unpacking idiom:
// the if condition checks
// validity, so fromMaybe's
// arg is always valid and
// defaultValue is irrelevant
if (isValid(a)) begin
    let aVal = fromMaybe(? , a);
    [...] end
```