Due date: Thursday March 21st 11:59:59pm EST.

Getting started: To create your initial Lab 5 repository, please visit the repository creation page at https://6004.mit.edu/web/spring19/user/labs/lab5. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-spring19/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 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 by Wednesday April 3rd. See the course website for lab hours.

Introduction

In this lab you will build two sequential circuits in Bluespec. In Lecture 9 and its recitation, we learned how to describe sequential circuits in Bluespec using modules, methods, and rules. In Appendix C, we have included a complete example of a module that computes the $n^{th}$ Fibonacci number in Bluespec. This may help you structure your code.

To pass the lab you must complete and PASS all of the exercises except the ones in the last three sections (marked as challenge exercises). However, in order to get full credit, complete ALL of the exercises (including the challenge ones) and be ready to answer all of the discussion questions below.

Coding guidelines: You are only allowed to make changes to Multipliers.bsv, CompletionBuffer.bsv, and TwoKitchenRestaurant.bsv. Modifications to other files will be overwritten during didit grading. You should be prepared to give answers to the discussion questions during the checkoff.

Implementation restrictions: In this lab you will build some 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: $*$ / $\%$ (multiplication, division, modulus). Unlike in lab 4, addition and subtraction ($+$ $-$), shifts ($<<$ $>>$), and comparisons ($<=$ $>$ $<=$ $>$) are allowed. Like in lab 4, bitwise-logical operators ($\&$ $|$ $\wedge$ $\sim$), equality/inequality operators ($==$ $!=$), conditional expressions ($?$ if), and loops are also allowed.

Please refer to Appendix A for additional debugging help.

For general Bluespec help, especially with module syntax, see this fantastic guide: https://github.com/kcamenzind/BluespecIntroGuide/blob/master/BluespecIntroGuide.md

1 Designing and Analyzing Sequential Circuits

In this lab you will be asked to create and analyze multiple sequential circuits. This section is to help you get started in understanding how to make Bluespec modules and how to do order-of analysis for space and time.
1.1 Making Modules

Modules in Bluespec are sort of like objects in programming languages. They contain state (that is, some number of registers inside) and they implement an interface using methods. They also have something called rules, which are procedures that are executed each cycle (provided the guard, which is a condition for a rule (more on that in a bit), is satisfied). That was a lot of new terms, but some examples may help contextualize that.

1.1.1 Interface Declaration

An interface defines how we interact with a module. Sort of like a class in Java, C++, or Python, the interface declares methods, which define the ways the module can be interacted with. The interface doesn’t hold the methods themselves, just their names and function signatures. It is up to the module to actually define them.

Here’s how an interface definition looks:

```plaintext
// Basic interface declaration
interface InterfaceName;
  method MethodType method1name(ArgType1 arg1, ArgType2 arg2 ... );
  method MethodType method2name(ArgType3 arg1, ArgType4 arg2 ... );
  ...
  method MethodType methodNname; // methods don’t have to have inputs
endinterface
```

1.1.2 Module Declaration

Modules are implementations of interfaces. This is handy because we could have multiple different modules that implement the same interface. Let’s step through a section at a time.

A module declaration follows the structure below. By convention, a module’s name usually begins with mk, which is short for make.

```plaintext
// Basic module declaration
module mkModuleName(InterfaceName);
  // Internal state here
  // Rules here
  // Methods here
endmodule
```

1.1.3 Module State

One of the key features of modules is that they have internal state that can change each cycle. Instantiating internal state is a lot like initializing any other variable. However, most of this is implemented using registers, so we need a new syntax for dealing with them.

Here’s how some instantiating looks:

```plaintext
Reg#(Bit#(1)) myReg <- mkRegU();  // Undefined initial value
Reg#(Bool) myRegFlag <- mkReg(False);  // Initialized to False
Reg#(Bit#(4)) myRegValue <- mkReg(4'b1001);  // Initialized to 4'b1001
ModuleName myModule <- mkModuleName();  // Creates an instance of the module ModuleName
```
This shows the $\leftarrow$ operator, which is how we initialize state variables in a module. The values inside of the mkReg function calls are initial values. mkRegU leaves the value uninitialized, so be careful about reading from an uninitialized register. Note that registers need a type which defines what type of variable they can hold. For instance, in the first register above, the type is Bit#(1).

Initializing registers is only part of the story though. Registers have two methods that let us read from and write to them. _read() and _write() For example if we have a 2-bit register foo, and wish to read from it, we would do:

```
Bit#(2) var = foo.read();
```

But there is a shorthand as well, where bluespec assumes you wanted to call read

```
Bit#(2) var = foo;
```

Similarly for writing, we avoid using the _write() method by using the $\leftarrow$ operator:

```
foo $\leftarrow$ 2'b11
```

Note that this $\leftarrow$ operator is distinct from the $\leftarrow$ operator. $\leftarrow$ (double arrow) is used when changing the value in a register, but $\leftarrow$ is used when initializing the contents of a module variable.

$\leftarrow$ has one additional use, which is to read values from an ActionValue method. If, for example, we had a method foo which returned an ActionValue#(Bit#(32)) and we wanted to store the result of that call, we would do as follows:

```
Bit#(32) bar $\leftarrow$ foo();
```

### 1.1.4 Methods and Return Values

Methods are like functions, except they can use the syntax above to read from and write to our state. However, we have to be more careful with return types, because there are some new restrictions. Whenever we declare a method that changes the internal state, its return value must be either Action or ActionValue#(type). You use Action when you intend to alter internal state, but don’t intend to return any value. You use ActionValue#(type) when you intend to alter internal state and return a value. Be sure to replace type with the actual type of your return value. Methods that do not alter internal state can have return values just like normal functions.

### 1.1.5 Rules

Rules are similar to methods or functions. They use temporary variables, can make method or function calls, and can do calculation. However, they don’t take input, don’t return anything, and can’t be directly interacted with from outside of the module. They are there to define how the internal state gets updated from cycle to cycle. Rules execute whenever they can, once per cycle. However, they can have conditions on when they execute called guards. For example:

```
rule ruleName if (guard);
  statement1;
  statement2;
  ...
  statementN;
endrule
```
The word guard above should be replaced by a condition (if you intend to have a guard at all). This condition is free to read from internal state (for example, I might want my rule to execute only when some register contains the value 0). However, so long as all guards are satisfied, the rule will execute each cycle.

It is possible, and often useful, to have multiple rules in one module. If their guards aren’t mutually exclusive, multiple rules can fire in one cycle.

For an example of a full module, see Appendix C.

### 1.2 Order-of Analysis

You can find some info on order-of analysis in lecture 8 slide 4. We will go a bit more in depth.

Order-of analysis (or asymptotic analysis) is a way of describing how a function grows as its input grows. For programmers and circuit designers this is usually used to analyse time and space complexity. For a program, space complexity might refer to how much memory a program takes up. For us, space is the area our circuit takes up.

The basics of how it works is you formulate a function that describes (for example) the time that it takes to execute a program. For example, I might write bubble sort, which takes roughly $n(n + 1)/2$ operations for an input of length $n$. It is important that the function you write relates the size of some input parameter to the amount of operations or size.

Now we’ll simplify my runtime formula. We know that $n(n + 1)$ is equal to $n^2 + n$, so our whole function looks like $n^2 + n/2$. We are concerned with giving a rough idea of how this function grows as $n$ tends to infinity, so constant factors don’t affect much in the long term. In addition, the $n^2$ term will grow much faster than the $n$ term as $n$ gets very large. This means that at very large $n$, our function just looks like $n^2$, so we say that our bubble sort has $O(n^2)$ time complexity.

This is not the most mathematically rigorous approach, but this should give enough background to begin your own analyses of circuits. Here are some other common complexities:

- $O(1)$ aka. constant time/space. Basically, increasing the input size has almost no effect on time/area
- $O(\log n)$ aka logarithmic time/space. Basically, doubling the input increases time/area by a constant factor
- $O(n)$ aka. linear time/space. Basically, if you double the input size, you double the time/area.
- $O(n \log n)$ No fancy name for this one. This one is a bit more complicated, but lives in the space between $O(n)$ and $O(n^2)$. It is much closer to $O(n)$ in terms of performance though.
- $O(2^n)$ aka exponential time/space. This one is to be avoided if at all possible. It blows up incredibly fast, way faster than any of the other ones listed. For example, for $n = 32$, $n^2$ gives us 1024, but $2^n$ gives over 4 billion. It only gets worse as $n$ increases.

With these tools in our belts, we should be much more prepared to tackle sequential circuits.

### 2 Implementing Multiplication by Repeated Addition

#### 2.1 Combinational Multiplier Function

**Exercise 1 (10 points):** Implement the combinational function `multiply_by_adding` in `Multipliers.bsv`, which multiplies two unsigned 32-bit numbers.

*Note:* You can find more information on writing parameterized Bluespec in Appendix B

*Note:* Lecture 7 (Slides 9-11) covers how to perform multiplication by repeated addition.

```plaintext
// Multiplication by repeated addition
function Bit#(TAdd#(n,n)) multiply_by_adding( Bit#(n) a, Bit#(n) b );
```
Remember that the result of multiplying two n-bit numbers is potentially 2n-bits long.

To test your design, run `make MultiplierByAdding && ./simMultiplierByAdding`.

### 2.2 Sequential Multiplier Module

**Exercise 2 (30 points):** In this exercise you will build a sequential circuit for multiplication by repeated addition (also known as a folded multiplier). The interface of the module is given below:

```haskell
interface Multiplier#( numeric type n );
    method Action start( Bit#(n) a, Bit#(n) b );
    method ActionValue#(Bit#(TAdd#(n,n))) result();
endinterface
```

Fill in the skeleton code of the module `mkFoldedMultiplier` in `Multipliers.bsv`.

- `start(a,b)` begins the multiplication process for numbers a and b.
- `result()` returns the product of a and b (which is a 2n-bit number). Make sure to specify a guard which makes sure result can only be called once the computation is completed and find a way to ensure that it can only be called once per calculation. Notice that it returns an `ActionValue`. Consider what might need to change about the internal state to ensure the conditions above.

To test your design, run `make FoldedMultiplier && ./simFoldedMultiplier`.

**Note:** If you get a failure message saying “Failed, cycle limit exceeded” consider whether you’re putting your module into deadlock (i.e. a situation where all the rules are stuck or can’t fire).

**Discussion Question:** The repeated addition algorithm does not work for multiplying two negative numbers in two’s complement encoding. Explain why this is the case and suggest a way to make your implementation work with multiplying two signed numbers.

### 2.3 Analyzing Combinational and Sequential Multipliers

Now we want you to analyze the performance of your combinational and sequential multipliers in terms of area and critical-path delay. Like in **Lab 4**, you will use `synth` for this purpose. Just give the name of module as the target circuit to `synth` as shown below.

```
synth <BSVFILE> <Function Name | Module Name> [options]
```

Synthesize `multiply_by_adding_8` and `mkFoldedMultiplier_8` by running:

```
synth Multipliers.bsv multiply_by_adding_8 -l multisize
synth Multipliers.bsv mkFoldedMultiplier_8 -l multisize
```

**Discussion Question:** How do the delay and area of the sequential multiplier compare with those of the combinational one? Any surprises?

Now synthesize `multiply_by_adding_X` for \( X \in \{4, 8, 16, 32\} \).

**Discussion Question:** For the combinational multipliers, how does 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 `mkFoldedMultiplier_X` for \( X \in \{4, 8, 16, 32, 64, 128\} \).

**Discussion Question:** For the folded sequential multipliers, how does 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.
3 Completion Buffer

3.1 Implementing a Completion Buffer

Sometimes we use modules that process multiple requests concurrently and do not produce responses in the same order in which requests are submitted. We call these modules out-of-order processors. For example, imagine a restaurant’s kitchen. At any given time the kitchen is preparing multiple dishes, and the time to prepare a dish depends the complexity of the recipe and the chef’s skill. To keep customers happy, the restaurant may want to serve the dishes in the same order that the customers requested them. This will require a buffer area to keep dishes that have been prepared but cannot yet be delivered to customers (because a dish that was requested earlier has not yet been finished).

In this exercise, we will explore how to wrap an out-of-order processor module (OOP) using Bluespec code, to produce another module (IOP or In-order-processor) that behaves in a FIFO (First-In-First-Out) manner. Figure 1 illustrates how this may be done. Inputs and outputs to a module with out-of-order behavior have to be tagged with an identifier (a small integer) so that an output can be associated with the corresponding input. We call this identifier a token.

The Completion Buffer (CB) is a module where responses (e.g., dishes) wait until they can be served in order. Since the CB has a finite capacity, a request (e.g., a request for a dish) has to reserve a slot and get a token from the CB before it can be dispatched to the OOP (e.g., the kitchen). When the OOP completes a request, it simply puts the response in the CB in the slot indicated by the token. The CB ensures a response is taken out only if all earlier requests have been served.

In this exercise you will implement a CB of size \( n \) (\( n \) is a power of 2), which will work with any OOP we supply you (e.g., a kitchen). In fact, the OOP is a black box to you. The rest of the code where the CB is called is given in the `mkItalianRestaurant` module in `ItalianRestaurant.bsv`. The interface of the CB is as follows:

```verilog
interface CompletionBuffer#(numeric type logn, type t);
  method ActionValue#(Bit#(logn)) reserve;
  method Action complete(Bit#(logn) tok, t d);
  method ActionValue#(t) drain;
endinterface
```

Note: The use of `numeric type` above allows us to make completion buffers for any numeric type (e.g., the \( n \)’s as used in `Bit#(n)` or the number 17 as used in `Bit#(17)`). Similarly, `type` lets `t` be of any type at all. This gives your completion buffer a lot of flexibility, allowing it to store any type of data.

The functionality of each of these methods is described below:

- **`reserve`** is an `ActionValue` method that reserves a CB slot that is not currently reserved and returns a token indicating the number of the CB slot.
- **`complete(tok, d)`** is an `Action` method that puts the completed response value `d` into the slot `tok` of
the CB.

- **drain** is an **ActionValue** method which returns a response from the CB corresponding to the oldest request. After draining, the slot is marked as invalid (and unreserved).

**Note:** Your CB must be able to allow a full \( n \) requests to be “in-flight” (i.e., it must be able to give out \( n \) reservations before any are completed and drained). To enforce this, the OOP in our testbench code will not start completing orders until at least \( n \) requests have been received.

Figure 2 shows the internal structure of the CB. Internally, the CB maintains a vector of registers to hold responses, and is managed as a **circular buffer**. Each slot in the circular buffer stores a **valid bit** and the **response value**. **head** points to the next slot to be reserved and **tail** points to the next response to be drained. Initially both pointers are zero, and every slot in CB is marked invalid. A reservation can execute only if there are unreserved slots. When a reservation executes, **head** is incremented by 1 modulo \( n \). Once an entry is completed, the valid bit of its reserved slot is set to 1. When a completed response is taken out of the CB (drained), **tail** is incremented by 1 modulo \( n \), and the valid bit of the drained slot is set to 0.

**Exercise 3 (25 points):** Fill in the skeleton module **mkCompletionBuffer** for a CB of size \( n \) (\( n = 2^{\log n} \)) in CompletionBuffer.bsv.

To instantiate a vector of registers, we will use the **replicateM()** Bluespec module as shown below. This has also been provided in the skeleton module **mkCompletionBuffer**.

```haskell
// Vector of n registers initialized to 0
Vector#(n, Reg#(Bit#(1))) valid <- replicateM(mkReg(0));

// Vector of n Uninitialized registers
Vector#(n, Reg#(dataT)) data <- replicateM(mkRegU);
```

You can access an element in a vector by its index, e.g., valid[i] <= 1.

To test your design, run **make ItalianRestaurant && ./simItalianRestaurant**

### 3.2 Analyzing Completion Buffer Performance

Before you run the synthesis tool, make sure your parameterized completion buffer is fully functional by running it for sizes 8, 16 and 32 (i.e., \( X \in \{ 8, 16, 32 \} \)).

```bash
synth CompletionBuffer.bsv mkCompletionBuffer_X -l multisize
```

**Discussion Question:** Report the synthesis results of your completion buffer design with sizes 8 and 16. How do the area and critical-path delay of your design change as the size grows?

You must complete exercises 1 to 3 and be prepared to answer the associated discussion questions to PASS the lab. However, in order to get full credit, complete ALL of the exercises (including the challenge ones) and discussion questions below.

### 4 Building a Faster Sequential Multiplier

**Challenge Exercise 4 (20 points):** Fill in the skeleton code for the module **mkFastFoldedMultiplier** in Multipliers.bsv, such that your 32-bit sequential multiplier achieves a critical-path delay \( \leq 280 \)ps.

**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 slightly different multiplier algorithm. 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 shift at each step.

To test your design, run **make FastFoldedMultiplier && ./simFastFoldedMultiplier**

To synthesize your design, run **synth Multipliers.bsv mkFastFoldedMultiplier_32 -l multisize**
5 Allowed OOP Behaviors

Discussion Question: For a CB of size 8, can the OOP output each of the following two sequences for the first 8 tokens?

(7, 3, 2, 4, 5, 0, 3, 1)
(7, 3, 2, 4, 5, 0, 1, 3)

Hint: Think about when a token can be reused.

6 Adding a Second OOP: The Dessert Kitchen

![In-order processing system with two OOPs.](image)

We want to add a second out-of-order module, a dessert kitchen, to our in-order processing system. Figure 3 shows this setup. Both OOPs use the same CB. Like in the single-OOP system, a dish order reserves a slot and gets a token from the CB before being dispatched to one of the kitchens. After getting the token, the dish request is sent to one of the kitchens depending on whether it is a main course or a dessert. When a kitchen finishes making a dish, it will put the completed dish into the CB along with its associated token. If the two kitchens finish dishes at the same time, the dishes can be put into the CB in any order. You do not have to concern yourself with this arbitration: when two rules invoke the same method at the same time (complete in this case), the Bluespec compiler picks a fixed arbitration schedule to execute them one by one (the arbiter box in Figure 3). The CB will take out a finished dish order when all of the earlier dish orders have been drained.

Challenge Exercise 5 (15 points): Fill in the skeleton code for the mkTwoKitchenRestaurant module in TwoKitchenRestaurant.bsv by modifying module mkItalianRestaurant in ItalianRestaurant.bsv, which implements an in-order processing system with only one kitchen. All the kitchen-related types and functions are in KitchenCommon.bsv.

To test your design, run make TwoKitchenRestaurant && ./simTwoKitchenRestaurant.
A  Appendix A : 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:

\[
\text{debug\_helper(function\_to\_test, expected\_return\_value, args\_for\_func\_to\_test, \ldots)}
\]

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:

```plaintext
#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
```

B  Appendix B : Doing Math with Types in Bluespec

In Bluespec there are a few ways to do arithmetic to figure out widths in parameterized functions. Specifically, we have access to TAdd, TSub, TMul, and TLog.

TAdd takes two parameters (i.e. some widths m and n) and adds them together. You can use this to set the width of another entry. For example, if we had m and n, but wanted to have a bitvector of length \( m + n \), you would declare:

\[
\text{Bit#(TAdd#(n,m)) foo; // bitvector of length n + m}
\]

The same goes for TSub and TMul. These work just like above, but give us lengths of \( n - m \) and \( n \times m \) respectively:

\[
\text{Bit#(TSub#(n,m)) bar; //bitvector of length n - m}
\]

\[
\text{Bit#(TMul#(n,m)) baz; //bitvector of length n * m}
\]

TLog works a little differently, however. It takes in some parameter n, and returns \( \lceil \log_2 n \rceil \), which is the ceiling of the base 2 logarithm of n. What this means is that if we have some n-bit number, Bit#(TLog(n))
will give us enough bits to represent any number up to \( n \). Here’s an example with concrete numbers:

```plaintext
// n = 32
Bit#(TLog#(n)) qux; // bitvector of length 5
```

### C Appendix C: Fibonacci Example

This code implements a complete Bluespec module that computes the \( n^{th} \) Fibonacci number.

```plaintext
interface Fibonacci;
    method Action start(Bit#(5) n);
    method ActionValue#(Bit#(32)) getResult;
endinterface

// start is an Action method which executes only when the module is not busy with
// computing the previous call.

// getResult is an ActionValue method which returns the result when it is
// ready. It also marks the module as not busy.

module mkFibonacci(Fibonacci);
    // At the beginning of module we instantiate the registers.
    // These registers make up the internal state of the module.
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(1);
    Reg#(Bit#(5)) i <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

    rule computeFibonacciStep if (i > 0);
        // The rule has a guard and executes only when i > 0. This rule defines the
        // internal working of the module and consequentially, it is not part of the
        // interface.
        x <= y;
        y <= x + y; // y will get the sum of the old values of x and y
        i <= i - 1;
    endrule

    method Action start(Bit#(5) n) if (!busy_flag);
        // This is a guarded Action method, which only changes the values of the internal
        // state registers
        if (n == 0) begin
            // special case for 0
            i <= 0;
            x <= 0;
            y <= 0;
        end else begin
            i <= n - 1;
            x <= 0;
            y <= 1;
        end
        busy_flag <= True;
    endmethod

    method ActionValue#(Bit#(32)) getResult if (busy_flag && i == 0);
```
// This is a guarded ActionValue method, which changes the internal registers and
// also returns a value
busy_flag <= False;
return y;
endmethod
endmodule