Due date: Friday October 5th 11:59:59pm EST.

Getting started: To create your initial Lab 3 repository, please visit the repository creation page at https://6004.mit.edu/web/fall18/user/labs/lab3. Once your repository is created, you can clone it into your VM by running:

    git clone git@github.mit.edu:6004-fall18/labs-lab3-\{YourMITUsername\}.git lab3

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 October 15th Monday. See the course website for lab hours.

Introduction

In this lab you will build two sequential circuits in Bluespec. In Lecture 7 and its recitation, we learned how to describe sequential circuits in Bluespec using modules, methods, and rules. In Appendix 5, 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.

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

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 2, addition and subtraction (+ -), shifts (<< >>), and comparisons (<= >= < >) are allowed. Like in lab 2, bitwise-logical operators (& | ^ ∼), equality/inequality operators (== !=), conditional expressions (? if), and loops are also allowed.

1 Implementing Multiplication by Repeated Addition

1.1 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: Lecture 4 (Slide 11) and Lecture 5 (Slides 20-22) cover how to perform multiplication by repeated addition.

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

    To test your design, run make MultiplierByAdding & & ./simMultiplierByAdding
1.2 Sequential Multiplier Module

Exercise 2 (20 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`.

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

Discussion Question 1 (5 points): 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.

1.3 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 2, you will use `synth` for this purpose. Just give the name of the 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 2 (5 points): 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 3 (5 points): 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 4 (5 points): 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.

2 Completion Buffer

2.1 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 on 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) 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 gets 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 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:

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

The functionality of each of these methods is described below:

- **reserve** is an ActionValue method that returns a token indicating the number of an unoccupied 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 empty.

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 the number of valid elements in the CB is less than $n$. When a reservation executes, head is incremented by 1 modulo $n$. 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 (20 points): Fill 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`

### 2.2 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 -l multisize
```

**Discussion Question 5 (5 points):** 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 discussion questions 1 to 5 to PASS the lab. To get full credit, complete the exercises and discussion question below.

### 3 Building a Faster Sequential Multiplier

**Exercise 4 (10 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 \( \leq 280 \text{ps} \).

*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 2). Second, you can try a slightly different multiplier algorithm. According to the algorithm in subsection 1.2, the number to be added in the \( i^{th} \) step depends upon the \( i^{th} \) bit of operand \( a \). You may have implemented this using dynamic bit selection \( (a_i) \), which requires 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`

### 4 Allowed OOP Behaviors

**Discussion Question 6 (5 points):** 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.

### 5 Adding a Second OOP: The Dessert Kitchen

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.

**Exercise 5 (10 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: Fibonacci Example

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

```bluespec
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
```