Due date: Thursday March 1st 11:59:59pm EST.

Turning in the lab: To turn in this lab, commit and push your changes you made to your git repository.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting within a week of the lab’s due date. See the course website for lab hours.

## 1 Introduction

In this lab you will be building different multiplier implementations and testing them using custom instantiations of provided test bench templates. First, you will implement multipliers using repeated addition. Next, you will implement a Sequential Multiplier using a folded architecture. Finally, you will implementing a Booth Multiplier.

The output of all of these modules will be tested with test benches that compare the output of the modules to BSV’s * operator for functionality.

All of the materials for this lab are in the git repository lab3.git. All discussion questions asked throughout this lab should be answered in discussion.txt. When you have completed the lab, commit your changes to the repository and push the changes.

## 2 Built-in Multiplication

BSV has a built-in operation for multiplication: *. It is either a signed or unsigned multiply depending on the types of the operands. For Bit#(n) and UInt#(n), the * operator performs unsigned multiplication. For Int#(n), it performs signed multiplication. Just like the + operator, the * operator assumes the inputs and the output are all the same type. If you want a 2n-bit result from n-bit operands, you have to first extend the operands to be 2n-bit values.

Multipliers.bsv contains functions for signed and unsigned multiplication on Bit#(n) inputs. Both functions return Bit#(TAdd#(n,n)) outputs. The code for these functions are shown below. (Note: pack and unpack are built-in functions that convert to and from Bit#(n) respectively.)

```haskell
function Bit#(TAdd#(n,n)) multiply_unsigned( Bit#(n) a, Bit#(n) b );
    UInt#(n) a_uint = unpack(a);
    UInt#(n) b_uint = unpack(b);
    UInt#(TAdd#(n,n)) product_uint = zeroExtend(a_uint) * zeroExtend(b_uint);
    return pack( product_uint );
endfunction

function Bit#(TAdd#(n,n)) multiply_signed( Bit#(n) a, Bit#(n) b );
    Int#(n) a_int = unpack(a);
    Int#(n) b_int = unpack(b);
    Int#(TAdd#(n,n)) product_int = signExtend(a_int) * signExtend(b_int);
    return pack( product_int );
endfunction
```

In this lab you will use these functions to test the correctness of your multiplier implementations.

## 3 Background on Modules

To complete this lab you will need to write Modules to describe sequential circuits in BSV. This section gives background on the different features of sequential circuits in BSV including Interfaces, Rules, Guards, and Methods. Section 7 has a full example of a Module that illustrates these features.
3.1 Rules

Sometimes we want modules to always perform some operation. For example, we may want to implement a 32-bit counter that continuously counts up, i.e., it increments itself every cycle. This differs from the counter module we saw in Lecture 5, which was incremented externally through an inc method. We could call the inc method continuously, or we could make inc a rule.

You can think of a rule as a method that is executed automatically every clock cycle.

3.2 Guards

Let’s say that you want to have some code that does not run every cycle, but instead only when some condition is true. There are two ways to do this. The first way is to make a rule with an if statement within it like this:

```hs
rule step;
  if( i < fromInteger(valueOf(n)) ) begin
    // Do stuff
  end
endrule
```

This rule runs every cycle, but it only executes the desired code when \(i < n\). The second way is to make a rule with a guard like this:

```hs
rule step( i < fromInteger(valueOf(n)) );
// Do stuff
endrule
```

This rule will not run every cycle. Instead, it will run only when its guard, \(i < \text{fromInteger(valueOf(i))}\), is true. While this does not make a difference functionally, it makes a difference in the semantics of the BSV language and to the compiler. This difference will be covered in later lectures, but until then, you should use guards in your designs for this lab. Otherwise, you may encounter test benches failing because they run out of cycles. (Note: If you are curious why this happens, the reason is that the BSV compiler may prevent multiple rules from firing in the same cycle if they may write to the same register. If your module has a rule that may update its state every cycle, it will prevent the test benches from initializing your module and starting a multi-cycle operation.)

3.3 Methods

The outside world can only interact with a module through its Interface. Interfaces are made up of methods, and there are three types of Methods: Value Methods, Action Methods, and ActionValue Methods.

3.3.1 Value Methods

These are methods that return a value to the caller. When these methods are called, the module’s internal state is not changed. For example, in Lecture 5’s counter module, read is a value method: the counter does not change, and its value is returned.

3.3.2 Action Methods

Actions are methods that cause a change in state. Note that the result of the action does not become visible until the next cycle. For example, when writing to a register, the new value of the register will not show up until the next cycle. This means that the following code results in \(a\) and \(b\) swapping values.

```hs
method Action swap();
  a <= b;
```
b <= a;
endmethod

3.3.3 ActionValue Methods

This is a method that both modifies registers and also returns a value. An $\text{ActionValue#(t)}$ method returns a value of type $t$.

4 Test Benches

This lab has two parameterized test bench templates that can be easily instantiated with specific parameters to test two multiplication functions against each other, or to test a multiplier module against a multiplier function. These parameters include functions and module interfaces. $\text{mkTbMulFunction}$ compares the output of two functions with the same random inputs, and $\text{mkTbMulModule}$ compares the outputs of a test module (the device under test or DUT) and a reference function with the same random inputs.

The following code shows how to implement test benches for specific functions and/or modules.

\[
\begin{align*}
(* \text{ synthesize } *) \\
\text{module mkTbDumb();} \\
\quad \text{function Bit#(16) test (Bit#(8) a, Bit#(8) b) = multiply\_unsigned(a, b);} \\
\quad \text{Empty tb <- mkTbMulFunction(test, multiply\_unsigned, True);} \\
\quad \text{return tb;} \\
\end{align*}
\]

\[
\begin{align*}
(* \text{ synthesize } *) \\
\text{module mkTbFoldedMultiplier();} \\
\quad \text{Multiplier#(8) dut <- mkFoldedMultiplier();} \\
\quad \text{Empty tb <- mkTbMulModule(dut, multiply\_signed, True);} \\
\quad \text{return tb;} \\
\end{align*}
\]

The two lines below instantiate a specific test bench using the test bench templates in $\text{TestBenchTemplates.bsv}$.

\[
\begin{align*}
\text{Empty tb <- mkTbMulFunction(test\_function, multiply\_unsigned, True);} \\
\text{Empty tb <- mkTbMulModule(dut, multiply\_signed, True);} \\
\end{align*}
\]

The first parameter in each (test\_function and dut) is the function or the module to test. The second parameter (multiply\_unsigned and multiply\_signed) is the correctly implemented reference function. In this case, the reference functions were created using BSV’s $\ast$ operator. The last parameter is a Boolean that designates if you want a verbose output. If you just want PASSED or FAILED to be printed by the test bench, set the last parameter to False.

These test benches (mkTbDumb and mkTbFoldedMultiplier) can be easily built using the provided Makefile. To compile these examples, you would write make Dumb.tb for the first and make FoldedMultiplier.tb for the second. The Makefile will produce the executables simDumb and simFoldedMultiplier. To compile your own test bench mkTb<name>, run make <name>.tb; ./sim<name> There are no .tb files produced by the compilation process, the extension is just used to signal what build target should be used.

Exercise 1 (2 Points): In $\text{TestBench.bsv}$, write a test bench $\text{mkTbSignedVsUnsigned}$ that tests if multiply\_signed produces the same output as multiply\_unsigned. If the test bench “passes”, then the two functions produce the same results. If the test bench “fails”, the two functions do not always produce the same output. Compile this test bench as described above and run it. (That is, run make SignedVsUnsigned.tb and then ./simSignedVsUnsigned)
Discussion Question 1 (1 Point): Hardware-wise, unsigned addition is the same as signed addition when using two’s complement encoding. Using evidence from the test bench, is unsigned multiplication the same as signed multiplication?

Discussion Question 2 (2 Points): In mkTBdumb, excluding the line

```haskell
function Bit#(16) test_function( Bit#(8) a, Bit#(8) b ) = multiply_unsigned( a, b );
```

and modifying the rest of the module to be

```haskell
(* synthesize *)
module mkTBdumb();
    Empty tb <- mkTBmulFunction(multiply_unsigned, multiply_unsigned, True);
    return tb;
endmodule
```

will result in a compilation error. What is that error? How does the original code fix the compilation error? You could also fix the error by having two function definitions as shown below.

```haskell
(* synthesize *)
module mkTBdumb();
    function Bit#(16) test_function( Bit#(8) a, Bit#(8) b ) = multiply_unsigned( a, b );
    function Bit#(16) ref_function( Bit#(8) a, Bit#(8) b ) = multiply_unsigned( a, b );
    Empty tb <- mkTBmulFunction(test_function, ref_function, True);
    return tb;
endmodule
```

Why are two function definitions not necessary? (i.e., why can the second operand to mkTBmulFunction have variables in its type?).

Hint: Look at the types of the operands of mkTBmulFunction in TestBenchTemplates.bsv.

5 Implementing Multiplication by Repeated Addition

5.1 As a Combinational Function

Multipliers.bsv has skeleton code for a function to calculate multiplication using repeated addition. Since this is a function, it must represent a combinational circuit.

Exercise 2 (3 Points): Fill in the code for multiply_by_adding so it calculates the product of a and b using repeated addition in a single clock cycle. (You will verify the correctness of your multiplier in Exercise 3.) If you need an adder to produce an \((n + 1)\)-bit output from two \(n\)-bit operands, follow the model of multiply_unsigned and multiply_signed and extend the operands to \((n + 1)\)-bit before adding.

Exercise 3 (1 Point): Fill in the test bench mkTBEx3 in TestBench.bsv to test the functionality of multiply_by_adding. Compile it with make Ex3.tb and run it with ./simEx3 .

Discussion Question 3 (1 Point): Is your implementation of multiply_by_adding a signed multiplier or an unsigned multiplier? (Note: if it does not match either multiply_signed or multiply_unsigned, it is wrong.)
5.2 As a Sequential Module

Multiplying two 32-bit numbers using repeated addition requires thirty-one 32-bit adders. Those adders can take a significant amount of area depending on the restrictions of your target and the rest of your design. To reduce the amount of area used, you will instead implement a folded multiplier. The folded version of the multiplier uses sequential circuitry to share a single 32-bit adder across all of the required computations by doing one of the required computations each clock cycle and storing the temporary result in a register. This requires multiple cycles per operation, but far less area.

In this lab we will create an n-bit folded multiplier. The register \( i \) will track how far the module is in the computation of the result. If \( 0 \leq i < n \), then there is a computation going on and the rule \texttt{mul\_step} should be doing work and incrementing \( i \).

When \( i \) reaches \( n \), there is a result ready for reading, so \texttt{result\_ready} should return true. When the action value method result is called, the state of \( i \) should increase by 1 to \( n + 1 \). \( i == n + 1 \) denotes that the module is ready to start again, so \texttt{start\_ready} should return true. When the action method \texttt{start} is called, the states of all the registers in the module (including \( i \)) should be set to the correct value so the computation can start again.

Remember to include guards on any rules that need them.

Exercise 4 (4 Points): Fill in the code for the module \texttt{mkFoldedMultiplier} to implement a folded repeated addition multiplier. Can you implement it without using a variable-shift bit shifter? Without using dynamic bit selection? (In other words, can you avoid shifting or bit selection by a value stored in a register?)

Exercise 5 (1 Points): Fill in the test bench \texttt{mkTbEx5} to test the functionality of \texttt{mkFoldedMultiplier} against \texttt{multiply\_by\_adding}. They should produce the same outputs if you implemented \texttt{mkFoldedMultiplier} correctly. To run these, run \texttt{make Ex5.tb; ./simEx5}.

6 Booth’s Multiplication Algorithm

The repeated addition algorithm works well multiplying unsigned inputs, but it is not able to multiply (negative) numbers in two’s complement encoding. To multiply signed numbers, you need a different multiplication algorithm.

Booth’s Multiplication Algorithm is an algorithm that works with signed two’s complement numbers. This algorithm encodes one of the operands with a special encoding that enables its use with signed numbers. This encoding is known as a Booth encoding. A Booth encoding of a number is sometimes written with the symbols +, −, and 0 in a series like this: 0 + −0b. This encoded number is similar to a binary number because each place in the number represents the same power of two. A + in the ith bit represents \((+1)2^i\), but a − in the ith bit correspond to \((-1)2^i\).

The Booth encoding for a binary number can be obtained bitwise by looking at the current bit and the previous (less significant) bit of the original number. When encoding the least significant bit, a zero is assumed as the previous bit. The table below shows the conversion to Booth encoding:

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Previous Bit</th>
<th>Booth Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The Booth multiplication algorithm can best be described as the repeated addition algorithm using the Booth encoding of the multiplier. Instead of switching between adding 0 or adding the multiplicand as in repeated addition, the Booth algorithm switches between adding 0, adding the multiplicand, or subtracting the multiplicand, depending on the Booth encoding of the multiplier. The example below shows a multiplicand \( m \) is being multiplied by a negative number by converting the multiplier to its Booth encoding.
The Booth multiplication algorithm can be implemented efficiently in hardware using the following algorithm. This algorithm assumes an $n$-bit multiplicand $m$ is being multiplied by an $n$-bit multiplier $r$.

initialization:

\[
\begin{align*}
// & All 2n+1 bits wide \\
m_{\text{pos}} &= \{m, 0\} \\
m_{\text{neg}} &= \{(-m), 0\} \\
p &= \{0, r, 1'b0\}
\end{align*}
\]

repeat $n$ times:

\[
\begin{align*}
&\text{let } pr = \text{two least significant bits of } p \\
&\text{if ( } pr == 2'b01): p = p + m_{\text{pos}}; \\
&\text{if ( } pr == 2'b10): p = p + m_{\text{neg}}; \\
&\text{if ( } pr == 2'b00 \text{ or } pr == 2'b11): \text{do nothing;}
\end{align*}
\]

Arithmetically shift $p$ one bit to the right;

\[\text{res = 2n most significant bits of } p;\]

The notation $(−m)$ is the two’s complement inverse of $m$. Since the most negative number in two’s complement has no positive counterpart, this algorithm does not work when $m = 10...0b$. Because of this restriction, the test bench has been modified to avoid the most negative number when testing.

**Note:** This is not a good way to design hardware. Never remove tests from your test bench just because your hardware fails them. One way around this problem is to implement an $(n + 1)$-bit Booth multiplier to perform $n$-bit signed multiplication by sign extending the inputs. If you are zero-extending the inputs instead of sign-extending them, you can get the $n$-bit unsigned product of the two inputs. If you add an extra input to the multiplier that allows you to switch between sign-extending and zero-extending the inputs, then you have a 32-bit multiplier that you can switch between signed and unsigned multiplication. This functionality would be useful for processors that have signed and unsigned multiplication instructions.

This algorithm also uses an arithmetic shift. This is a shift designed for signed numbers. When shifting the number to the right, it shifts in the old value of the most-significant bit back into the new most-significant bit to keep the sign of the value the same. This is done in BSV when shifting values of type Int#(n). To do an arithmetic shift for Bit#(n), you may want to write your own function similar to multiply_signed. This function would convert Bit#(n) to Int#(n), do the shift, and then convert back.

**Exercise 6 (4 Points):** Fill in the implementation for a folded version of the Booth multiplication algorithm in the module mkBooth. This module uses a parameterized input size $n$; your implementation will be expected to work for all $n >= 2$.

**Exercise 7 (1 Point):** Fill in the test benches mkTbEx7a and mkTbEx7b for your Booth multiplier to test different bit widths of your choice. You can test them with make Ex7a.tb; ./simEx7a and make Ex7b.tb; ./simEx7b.
7 Appendix: Fibonacci Example

This Appendix illustrates Rules, Guards and Methods through a module that computes the \( n^{th} \) Fibonacci number.

```vhdl
// We first define the interface
interface Fibonacci;
  // start is an Action since it will set the registers to start the computation
  method Action start(Bit#(5) n);
  // getResult is an ActionValue since it will both return the result and change a register
  method ActionValue#(Bit#(32)) getResult;
  // busy and ready are just methods that we can call to check the status
  method Bool busy;
  method Bool ready;
endinterface

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);

  // The rule with a guard.
  // This will run on each step where the value of i is greater than 0.
  // Note that this rule is not part of the interface. The interface
  // does not define the internal working of a module
  rule computeFibonacciStep(i > 0);
    x <= y;
    // y will get the sum of the old values for y and x
    // since the change has not taken effect yet
    y <= x + y;
    i <= i - 1;
  endrule

  // This is an Action so we only change the values of internal state registers
  method Action start(Bit#(5) n);
    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

  // This is an ActionValue, so we both change the internal registers and also return a value
  method ActionValue#(Bit#(32)) getResult;
    busy_flag <= False;
endmodule
```
return y;
endmethod

method Bool busy;
    return busy_flag;
endmethod

method Bool ready;
    return busy_flag && (i == 0);
endmethod
endmodule