

# 6.S195: Lab 2

## Multipliers

Andy Wright\*

September 16, 2013

**Due: Monday September 23, 2013**

### 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 Booth Multiplier using a folded architecture. Finally you will build a faster multiplier by implementing a radix-4 Booth Multiplier.

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

All of the materials for this lab are in the git repository `$GITROOT/lab2.git`. All questions asked throughout exercises in 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

Bluespec 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: <sup>1</sup>

---

\*The previous version of this lab was written by Muralidaran Vijayaraghavan and Sang-Woo Jun

<sup>1</sup>`pack` and `unpack` are built-in functions that convert to and from `Bit#(n)` respectively.

```

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

```

These functions will be the benchmark functions that your multipliers in this lab will be compared to for functionality.

### 3 Test Benches

This lab has two 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. `mkTbMulFunction` compares the output of two functions with the same random inputs, and `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.

```

(* synthesize *)
module mkTbDumb();
    function Bit#(16) test_function( Bit#(8) a, Bit#(8) b )
        = multiply_unsigned( a, b );
    Empty tb <- mkTbMulFunction(test_function, multiply_unsigned,
                               True);
    return tb;
endmodule

(* synthesize *)
module mkTbFoldedMultiplier();
    Multiplier#(8) dut <- mkFoldedMultiplier();
    Empty tb <- mkTbMulModule(dut, multiply_signed, True);
endmodule

```

```
    return tb;
endmodule
```

The two lines below instantiate a specific test bench using the test bench templates in `TestBenchTemplates.bsv`.

```
Empty tb <- mkTbMulFunction(test_function, multiply_unsigned, True);
Empty tb <- mkTbMulModule(dut, multiply_signed, True);
```

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 Bluespec's `*` 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 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):** Write a test bench `mkTbSignedVsUnsigned` that tests if `multiply_signed` produces the same output as `multiply_unsigned`. Compile this test bench and run it. Is unsigned multiplication the same as signed multiplication?

## 4 Implementing Multiplication by Repeated Addition

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

**Exercise 2 (2 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. 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 Points):** Create a test bench in `TestBench.bsv` to test the functionality of `multiply_by_adding`. Is your implementation a signed multiplier or an unsigned multiplier? (Note: if it does not match either `multiply_signed` or `multiply_unsigned`, it is wrong).

Multiplying two 32-bit numbers using repeated addition requires 31 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. In lecture, a folded version of the repeated addition multiplier was presented to reduce the amount of area needed for a multiplier. The folded version of the multiplier uses sequential circuitry to share a single 32-bit across all of the required computations by doing one of the required computations each clock cycle and storing the temporary result in a register.

**Exercise 4 (2 Points):** Fill in the code for the module `mkFoldedMultiplier` to implement a folded repeated addition multiplier.

**Exercise 5 (1 Points):** Create a test bench in `TestBench.bsv` to test the functionality of `mkFoldedMultiplier` against `multiply_by_adding`. Does it produce the same outputs?

## 5 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 sometimes known as 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  $i$ th bit represents  $(+1) \cdot 2^i$ , but a `-` in the  $i$ th bit correspond to  $(-1) \cdot 2^i$ .

The Booth encoding for a binary number can be obtained bitwise by looking at the current bit and the previous 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.

Current Bit	Previous Bit	Booth Encoding
0	0	0
0	1	+1
1	0	-1
1	1	0

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.

$$\begin{aligned}
 -5 \cdot m &= 1011\text{b} \cdot m \\
 &= -+0\text{-b} \cdot m \\
 &= (-m) \cdot 2^3 + m \cdot 2^2 + (-m) \cdot 2^0 \\
 &= -8m + 4m - m \\
 &= -5m
 \end{aligned}$$

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:
    // All 2n+1 bits wide
    m_pos = {m, 0}
    m_neg = {(-m), 0}
    p = {0, r, 1'b0}

repeat n times:
    let pr = two least significant bits of p
    if ( pr == 01 ): p = p + m_pos;
    if ( pr == 10 ): p = p + m_neg;
    if ( pr == 00 or pr == 11 ): do nothing;

    Arithmetically shift p one bit to the right;

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  $m = 10\dots 0\text{b}$ . Because of this restriction, the test bench has been modified to avoid the most negative number when testing <sup>2</sup>.

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

<sup>2</sup>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 zero extend the inputs instead of sign extending them, you can get the  $n$ -bit unsigned product of the two inputs. If 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 is functionality would be useful for a processor that has signed and unsigned multiplication instructions.

of the most significant bit back into the MSB place to keep the sign of the value of the same. This is done by Bluespec 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 (3 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  $n \geq 2$ .

**Exercise 7 (1 Point):** Write test benches for your Booth multiplier to test different bit widths of your choice.

## 6 Radix-4 Booth Multiplier

One more advantage of the booth multiplier is that it can be sped up efficiently by performing two steps of the original Booth algorithm at a time. This is equivalent to performing two bits worth of partial sum additions per cycle. This method of speeding up the Booth algorithm is known as the radix-4 Booth multiplier.

The radix-4 Booth multiplier looks at two current bits at a time when encoding the multiplier. The radix-4 multiplier is able to run faster than the original one because each two bit Booth encoding can be reduced down to a Booth encoding with no more than one non-zero bit. For example, the bits 01 following a previous 0 bit is converted to +- with the original Booth encoding. +- represents  $2^{i+1} - 2^i$  which is equal to  $2^i$  which is just 0+. The table below shows some cases for the radix-4 Booth encoding.

Current Bits	Previous Bit	Original Booth Encoding	Radix-4 Booth Encoding
00	0	00	00
00	1	0+	0+
01	0	+-	0+
01	1	+0	+0
10	0		
10	1		
11	0		
11	1		

**Exercise 8 (1 Point):** Fill in rest of the table above in `discussion.txt`.

Some pseudocode for a radix-4 Booth multiplier can be seen below:

```
initialization:
  // All 2n + 2 bits wide
  m_pos = {msb(m), m, 0}
  m_neg = {msb(-m), (-m), 0}
```

```

    p = {0, r, 1'b0}

repeat n/2 times:
    let pr = three least significant bits of p
    if ( pr == 000 ): do nothing;
    if ( pr == 001 ): p = p + m_pos;
    if ( pr == 010 ): p = p + m_pos;
    if ( pr == 011 ): p = p + (m_pos << 1);
    if ( pr == 100 ): ...
        ... fill in rest from table ...

    Arithmetically shift p two bits to the right;

res = p with MSB and LSB chopped off;

```

**Exercise 9 (3 Points):** Fill in the implementation for a radix-4 Booth multiplier in the module `mkBoothRadix4`. This module uses a parameterized input size `n`; your implementation will be expected to work for all *even*  $n \geq 2$ .

**Exercise 10 (1 Point):** Write test benches for your radix-4 Booth multiplier to test different bit widths of your choice.

When you have completed all the exercises and your code works, commit your changes to the repository, and push your changes back to the source.