CS220
Introduction to Computer Organisation
Lab 2

Amey Karkare

1 Introduction

In this lab, you will build 2 different implementations of multipliers for signed binary number multiplication. The numbers are represented in 2's complement encoding.

2 Sign-magnitude Multiplication Algorithm

In sign-magnitude algorithm, we multiply the magnitudes of the number in an unsigned manner, and then take care of the appropriate sign using the signs of the inputs. Let m and r be the signed numbers of length n, the algorithm can be described as follows:

```
a   = absolute value of m
sa  = sign of m
b   = absolute value of r
sb  = sign of r

p   = a * b /* use unsigned multiplication algorithm */

res = (sa == sb) ? p : -p
```

3 Booth's Multiplication Algorithm

The booth algorithm takes as input signed numbers m and r of length n, and returns a value res of length $2n$. This is done by repeatedly pre-determining two values A and S of size $2n+1$ to product P of size $2n+1$ and performing a right arithmetic shift on P.

The pseudo-code for multiplication of two numbers m and r using the booth algorithm can be described as follows[1]:

```
initialization:
  A = {m, 0}
  S = {(-m), 0}
  P = {0, r, 1'b0}

repeat n times:
  let pr = two least significant bits of P
  if ( pr == 01 }: P = P + A;
  if ( pr == 10 ): P = P + S;
      if ( pr == 00 or pr == 11}: do nothing;

  Arithmetically shift P one bit to the right;

res = 2n most significant bits of P;
```

Note: (-m) is in 2's complement encoding, and Bluespec would use that for that expression.

4 Different Implementations of the Algorithm

For this lab, you will be provides implementation of an unsigned multiplier, a test-bench and some helper files. You will implement two different implementations of the multiplier that multiplies two arguments of type `Data` which has width `DataSz`. Some `typedef`s have been provided in `MultiplierTypes.bsv` to guide you in your implementation:

```
      DataSz : Width of each argument
      Data : Bit type of width DataSz
      AccumSz: 1 + sum of the widths of the arguments
      Accum : Bit type of width AccumSz, to hold A, S and P
```

The implementation of a polymorphic unsigned multiplier is provided in the file `Multiplier.bsv`. The makefile generates `simDef` that tests this implementation. Compile and run using

```
$ make def
$ ./simDef
```

If the simulator finds any error case for the implementation, it will display the chosen inputs, the desired output and your implementation's output. Otherwise, it will display `PASSED`. Verify that the default implementation passes on your system.

[1]Compare this pseudo-code with the algorithm described in the class. You should convince yourself that the two algorithms compute the same value.

Exercise 1: Complete the implementation of the signed multiplier in the file `SignedMultiplier.bsv`. [10]

The makefile generates `simSign` that tests this implementation. Compile and run using

```
$ make sign
$ ./simSign
```

Exercise 2: Complete the implementation of the booth multiplier in file `BoothMultiplier.bsv`. [15]

The makefile generates `simBooth` that tests this implementation. Compile and run using

```
$ make booth
$ ./simBooth
```

5 Discussion Questions

Answer these questions in the file `discussions.txt`

Discussion 1: List a positive and a negative hardware aspect (latency, throughput or area) of using Booth's algorithm over simple repeated addition for multiplication [2]

Discussion 2: The file `TestBench.bsv` implements the testbench for the multipliers. It uses a data structure called `Fifo`—a standard structure for *First-In-First-Out* behaviour. In class, we discussed that only one write per clock cycle to an element is allowed by BSV rules. However, here you can see that at each clock cycle, we call two *write*-like routines on Fifo `f`: `enq` and `deq`. Still, the design seems to run fine. Explain this seemingly contradictory behviour? (I do not want a formal answer, just an intuition). [3]