CSE125, Spring'20, Lab 5: Dedicated Pipelined Multiplier Circuit

Due Date: 05/22/20

1 Introduction

In this lab, you will be building off your Lab 4 code by adding a pipelined multiplication function. This will require you to make modifications to extend the instruction set architecture, build your own dedicated multiplication circuit, and implement the necessary control and data flow to support your circuit.

2 ALU Based Implementation

To start off, don't worry about pipelining and focus on extending the instruction set to support multiplication. To do this, implement a single cycle multiply function into your existing ALU. You should not need to make many modifications to the surrounding circuitry, as you will be using the R-type data paths which already exist. In order to ensure your implementation is compatible with our test cases, ensure you use the correct opcode and functions from the RISC-V 32 bit M extension provided below:

```
mul     rd rs1 rs2 31..25=1 14..12=0 6..2=0x0C 1..0=3
```

You will only be responsible for implementing a 16 x 16 bit multiply which takes the lowest 16 bits of each input and stores the result into a single 32 bit register. For the pipelined implementation you will be required to support signed inputs, but for this single cycle ALU implementation it is not necessary.
Hint: "*" is synthesizable in Chisel!

3 Testing ALU Based Implementation

Implement your own unit test to ensure the correctness of your multiply operation. Take a look at /src/tests/scala/components/ALUControlUnitTest.scala for an example on how to use the PeekPokeTester. This allows you to apply some input stimulus to your circuit (poke) and observe the outputs (peek). To be explicit you should poke instructions into your cpu, and make assertions about the contents of the data memory with the "expect" function. Also take a look at how the lab testbenches are implemented in the /src/test/scala/labs/

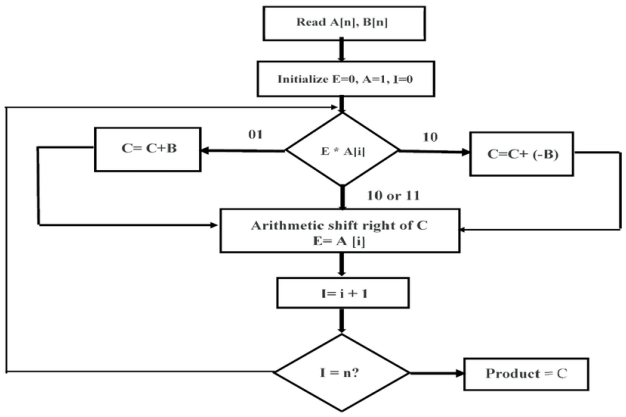

Figure 1: Booth multiplier flow chart

directory. It is recommended to create a new testbench to easily run all multiplication unit tests you create in batch.

4 Multiplication Circuitry

Next you will implement a radix 4 Booth multiplier circuit. Booth multiplication is based on the idea of partial products meaning that portions of a multiplication problem can be solved independently. Think back to multiplying large numbers together by hand and how the least significant digit of the result depended only on the least significant digits of the multiplier and multiplicand. In normal (radix 2) Booth multiplication, we add or subtract our multiplicand to an aggregate accumulator depending on what is known as the lost bit, which is the least significant bit of the product. At the end of every cycle, the lost bit is dropped and the entire product is shifted one bit to the right. See the figure above for details, and try to use this website to view examples and convince yourself that this works.
The circuit you will be implement, the radix 4 Booth multiplier, is a slightly more complicated version of the radix 2 Booth multiplier which evaluates 2 input digits per cycle instead of 1. This means we can produce a 32 bit output in 8 clock cycles. For details on this algorithm, check out this paper which also reviews the standard Booth multiplier.

5 Testing the Booth Multiplier

Before working your new multipler into you circuit, build a testbench to verify it independently from the rest of the CPU. Just like you did for your ALU implementation, use the PeekPokeTester to apply stimulus to your multiplier and assert with the "expect" function at the output. This should work for both

singed and unsigned inputs. The output should be ready after eight cycles. Feel free to add any extra control logic you think will make this task easier, such as a multipler ready or multipler processing signal.

6 Adding to the CPU

Since the multiplication operation is an R-type instruction, you will have to be careful about how you handle the datapathing as to not conflict with other R-type reasources. Also consider how an eight cycle operation will effect your hazard detection and forwarding. There exist different solutions leading to different performance. We will grade with maximum points every solution that works, but you may get extra points for a particularly efficient solution. As DINO CPU is an in-order processor, you do not have to worry about executing operations out of order while you multipler is processing, and always assume branch not taken for branch prediction. Again, you should add some test cases for the pipelined multiply instruction just as you did for the ALU based version, but also ensure that you do not break the functionality of other instructions. To this end, either run the tests you used for lab 4, or add them to your new testbench to also test the non-multiply instructions.