Computer Architecture Lab 3: Single Cycle CPU with Floating Point and Multiplication Functionality

Daniel Connolly and Josh Deng

November 19, 2018

Abstract

In this lab, we chose to improve upon the single cycle, 32-bit CPU by adding fixed-point multiplication functionality and MIPS coprocessor 1, the standard IEEE-754 compliant floating-point unit. Using assembly code in order to utilize our CPU, we verified the functionality of these additional units.

1 Process

In order to create our upgraded CPU, we first implemented a multiplier. After coming up with the structural design for a multiplier using full adders, we implemented the module using behavioral verilog. As specified by MIPS, we stored the most significant thirty-two bits in a Hi register and the least significant thirty-two bits in a Lo register. We then constructed a simple test bench to verify the stand-alone functionality of the module. What a multiplier would allow us to do is to test recursive function calls to factorials of a given number.

Next, we proceeded onto the task of designing a floating point unit. This required us to investigate different methods of adjusting the mantissa and the exponent in order to achieve the correct result. Over time, we converged towards a model that utilized a subtractor to calculate the difference between the exponents of our two operands and then bit-shifted the mantissa accordingly, as is shown in figure 1. Additionally, we decided to perform signed operations with our floating point ALU, which required us to take the two's complement of the bit-shifted mantissa and then sign extend the result with the floating point number's signed bit. Finally, we needed to renormalize the output of the ALU in order to ensure the floating point unit's output contained the implicit leading one in the mantissa as specified by IEEE-754 regulations.

To test the functionality of our floating-point unit, which will from here on be referred to as our fpu, we relied heavily on a variety of manually generated test cases, which stressed our system with differences in the signs and magnitudes of the two operands. Due to some of earlier design decisions, we knew that the smaller of the two operands would always become operandA of the ALU that computed our mantissa, as the smaller of the two needed to be right-shifted according to the difference between the exponents. As a result, if the difference in the exponents when calculated by the subtractor was less than or equal to zero, we could determine the sign of our result by inverting the most significant bit of operandB of the mantissa-computing ALU. Additionally, we created a large lookup table in order to renormalize the output of the fpu according to the position of the implicit leading one in our result. Moreover, we determined that by looking at the most significant bit of our result that was truncated, we were able to determine whether or not to round the output of the fpu up or down. As we ran our fpu through a series of test cases that stressed its abilities, we built up its functionality and reliability.

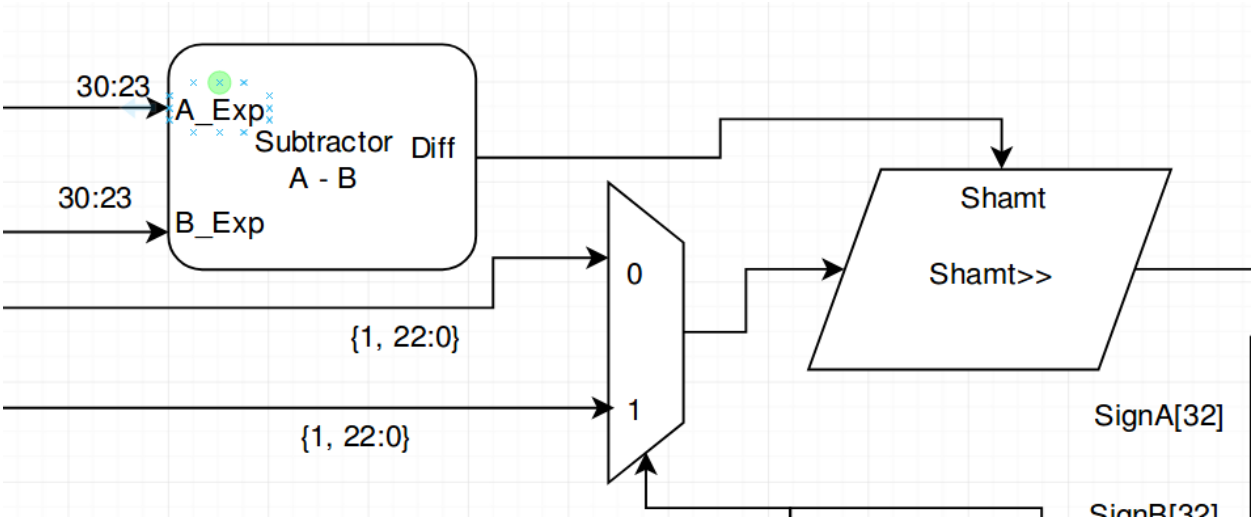

Figure 1: To determine which and the amount by which the mantissa of one of our operands must be shifted, we send both exponents through a subtractor, which finds the difference between the two exponents. In the diagram above, we show how the exponent difference is used to shift the mantissa.

Once we had moved beyond simply testing our floating point unit as a standalone module, we connected it to the rest of our single-cycle CPU in order to test the whole coprocessor with assembly code. For our assembly test case, we stored several floating-point numbers in our data memory and loaded them to our register file before adding and subtracting several of them and storing them back in memory. Unfortunately, we had not realized that coprocessor 1 would require its own register file. As a result, we implemented a new register file specifically for holding floating point values along with all of the control logic, shown expanded beyond our single-cycle CPU's control signals in Figure 2, necessary for correctly handling its read and write operations. At the same time, we made it possible for our CPU to handle the LWC1 and SWC1 MIPS commands, which represent the CPU's ability to load and store words from and to data memory using coprocessor 1, otherwise known as our fpu.

Instruction Type	Instruction	Float_RegWrite	Float_RegWriteSelect	FloatWriteAddrSel	MultiplyEN	DMDataSelect	RegWrite	MuxA_EN	MuxB_EN	ALUOP	DMWrite_En	MuxPC	MuxWD3_EN	RegWriteAddSelec
R	jr	0	0		0		0	(x)0	x	(x)add	0	2	x	x
	sub	0	0		0		1	0	1	sub	0	0	1	2
	slt	0	0		0		1	0	1	slt	0	0	1	2
	add	0	0		0		1	0	1	add	0	0	1	2
I	lw	0	0		0		1	0	0	add	0	0	0	0
	sw	0	0		0		0	0	0	add	1	0	x	x
	beq	0	0		0		0	0	1	sub	0	3 if branch in	x	x
	bne	0	0		0		0	0	1	sub	0	3 if branch in	x	x
	addi	0	0		0		1	0	0	add	0	0	1	0
	xori	0	0		0		1	0	0	xor	0	0	1	0
J	j	0	0		0		0	x	x	(x)add	0	1	x	x
	jal	0	0		0		1	1	2	add	0	1	1	1
R	mult	0	0		1		0	0	1	(x)add	0	0	x	x
R	mfhi	0	0		0		1	x	x	(x)add	0	0	2	2
R	mflo	0	0		0		1	x	x	(x)add	0	0	3	2
FR	add.s	1	1	0	x		0	x	x	add	0	0	x(0)	x
FR	sub.s	1	1	0	0		0	x	x	sub	0	0	x(0)	x
I	lwc1	1	0	1	0	0	0	0	0	add	0	0	x(0)	x
I	swc1	0	0	x(0)	0	1	0	0	0	add	1	0	x(0)	x

Figure 2: Table of control signals based on various instructions.

2 Test Case Strategy

2.1 Memory Unfixed

We performed the majority of our testing of each of the new modules we added to our single-cycle cpu in standalone test benches. These checked the functionality of the component without necessitating that we wire the components to the rest of the CPU. For our floating point unit, for example, we test with both positive and negative numbers in each of the operands, as well as with numbers that differed greatly in magnitude. This allowed us to see the source of various errors, from sign errors to rounding errors.

Eventually, we moved onto more advanced testing of the fpu after wiring it to the rest of the CPU. However, as we tried to implement the tests in the Assembly program, we realized how the instructions of a FPU involves independent FPU register file and more commands, such as LWC1 and SWC1, that we did not apticipate the need to implement before our integration of FPU with our CPU. This led us to major redesign of our datapath and rewriting of our assembly test program to avoid using psuedo commands that would have complicated the implementation even further.

As we searched through the waveforms our CPU generated, we found errors that completely befuddled us. These errors showed our program counter correctly selecting the current instruction from instruction memory and the instruction correctly calculating the addresses within data memory in which we had stored our floating point values; however, though the address was correct, data memory output incorrect data, as shown in Figure 3. We had actually stored this incorrect data three words away from the word we wanted to retrieve.

This led us to thinking back of one of the test cases that terminated but failed due to Verilog's automated checking last time. Most of our test cases for the single cycle CPU functioned correctly because they at most used stack points to put data at specific addresses that the CPU simply needed to retrieve later. In contrast, the floating point assembly test and one of Lab3's test cases had failed because they required an array that was arranged beforehand, instead of being written to with the conscious decision of the CPU or the stack pointer, which might have led to the mismatch between the data at the addresses and the expected data. In order to further test this assumption, we padded our .data file with three words consisting of all zeros between the words that we actually wanted to look at, as is shown in Figure **??**. In turn, this led our CPU and fpu to function correctly, as seen in Figure 5, storing all of our expected values in their correct data memory addresses.

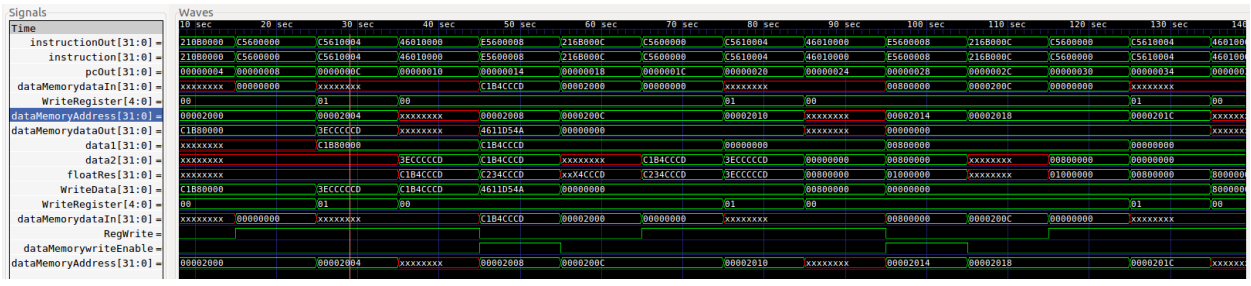

Figure 3: The waveforms above show that our memory unit was reading from the seemingly correct address but retrieving incorrect data.

2.2 Memory Fixed

After consulting with the Computer Architecture Professor Hill, we realize our mistakes with how we configured our memory. At each address of a standard MIPS memory, there is a byte of

```
c1b80000
00000000
00000000
00000000
c5af3800
00000000
00000000
00000000
```

Figure 4: The figure above shows exactly how we zero-padded our data to achieve the accurate functionality of our CPU and fpu.

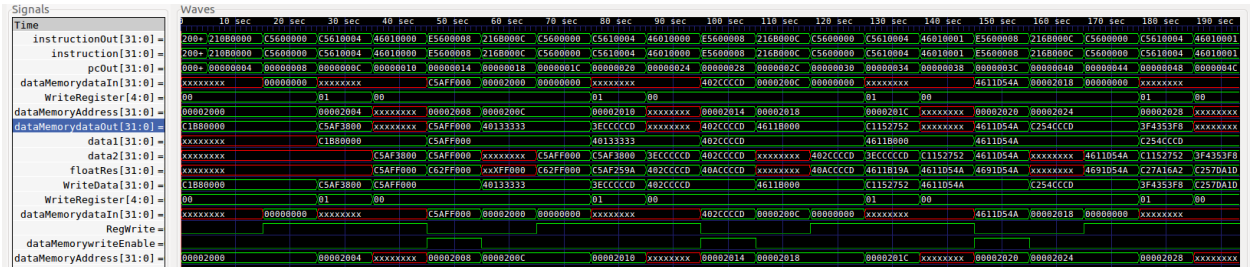

Figure 5: The waveforms above show that our CPU and fpu worked correctly when we padded the .data file with zero entries.

information. However, as we were building a 32-bit MIPS CPU, we configured our memory to have 32 bits at each address in order to be word align throughout instruction- and data-read or write operations. Thus, such a design require us to increment or decrement the 3rd LSB in the address since each word is a word, or four bytes, away from its neighbors. A right shift of two would help us get rid of the two LSBs and increment correctly within our specific memory configuration. While we caught this detail and right shifted by 2 for reading our instruction memory, we inadvertently made discrepant treatment to our data memory. The correct implementation requires us to right shift by 2 on all of instruction read address, data memory write address, and data memory read address.

First, when we loaded .text and .data dumped from MARS, we forgot to apply the right shifted addresses for where we would start loading. It was completely fine for the .text which contains Assembly instructions since its address offset is zero. Any operations still give us back zero. It became problematic for .data, however, because in MARS's compact memory configuration it would start at 0x2000, with its two right shift being 0x800. Failing to implement the right shifted address led us to our mistaken belief that right shift by 2 on data memory read address was not the way to go.

In retrospect, what further complicated our confusion about our memory is that when we experimented with right shifting the data memory address, we only applied the right shift on the read address but not the write address. Since we started our testing recursive functions in Assembly, such a treatment caused the same stack pointers to write and read from totally different addresses. By random chance, we tried out removing the right shift on the read address instead of adding a right shift to the write address first. That made a majority of Assembly tests in Lab 3 to pass, which led us to believe that it was simply a test bench error in our array Assembly test rather than mistakes in one of our modules.

3 Performance Analysis

Both of our multiplier and FPU execute their individual commands within one clock cycle. With the single cycle design, however, came the enormous amount of area that accompanied both the FPU and the multiplier. For example, our single cycle multiplier would require 62 adders with over a thousand AND gates. Our FPU is even much more enormous. The second register file for the FPU is dwarfed by the giant lookup tables needed for our variable shifter and normalization unit, in addition to three ALU/adder/subtractors and a number of needed to parse IEEE 754 floating point format correctly.

4 Work Plan Reflection

We did not developed a work plan for this specific lab. Although we did have a micro-poposal, we hugely overestimated what we could possibly have hoped to accomplish over the course of two weeks. As we discovered the need for an independent FPU register file the night before this lab was due, we have most certainly realized and experienced the consequences of our overestimation.

5 Module Reuse

For Lab 3, we reused the following modules, with alterations as deemed necessary, from the following sources:

- Everything from Daniel, Josh, and Will's Single Cycle CPU Lab

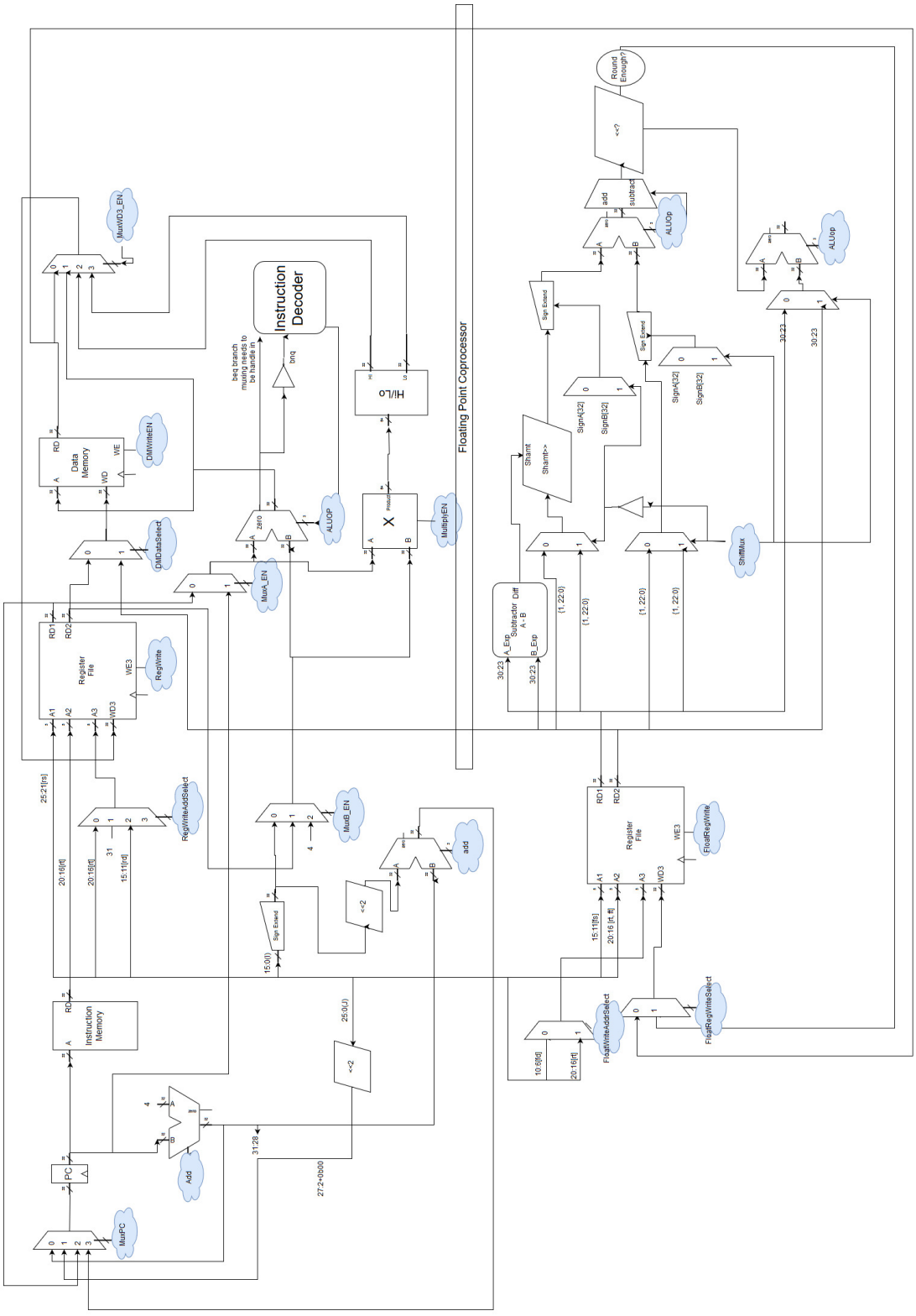

Figure 6: The block diagram of our Single-Cycle CPU is shown above. In this diagram, the multiplier is shown as a black box that populates the Hi and Lo registers, while the Floating Point Coprocessor is shown as its own datapath from the register file.