Assignment 4: Phloating Point PinKY

William Harvey, Ben Smith, Reid Wahlbrink

December 11, 2018

1 Abstract

The assignment is to implement the PinKY floating point instructions in a pipelined processor.

2 General Approach

Because the focus of this assignment was on the floating point implementation we decided to use Dr. Dietz's sample solution for the PinKY pipelined processor. We thought this would help us avoid headaches with the pipelined implementation's from our previous groups. As such we also adopted Dr. Dietz's AIK specification and Instruction Encoding. Dr. Dietz's processor has 3 stages named Stage 0, Stage 1, and Stage 2. His processor also seems to properly handle register dependencies and setting the Z flag. We instantiated our floating point operation modules in the processor module and we latched their results into the res register in Stage 2.

With regard to conditional execution Dr. Diet'z processor had one small problem. Another classmate who was also using Dr. Dietz's processor pointed out to us that his 'define OP was written as [14:9] rather than [13:9]. This meant that OP codes were 6-bit values rather than 5-bit values throughout his processor. We didn't catch this error because we tested exclusively with AL instructions. The 14th bit position in Dr. Dietz's instruction encoding is the bottom bit of the condition code and for AL that bit is a 0. Let's take the example of an ADD AL whose OP code is 5'h08. When comparing a 6-bit OP code with a 5-bit 'define 0 1000 == 00 1000 is true and the instruction is decoded properly. AL and NE would both have worked fine because they both have 0 in the 14th bit position of the instruction word. But S and EQ instructions have a 1 in that 14th bit position and would not have been recognized by the Stage 2 case statement (0 1000 == 10 1000 is false). We changed the 'define OP to [13:9] which should resolve the issue.

We tested our processor using Icarus Verilog instead of Dr. Dietz's CGI website. This means our code is not written to support the CGI website (specifically the syntax for readmem()). Our processor uses readmem() to read in instructions, data, registers, and a lookup table for RECF. You can run different instructions through our processor by using the AIK spec in pinky.aik (source here) and pasting the generated instructions into our instrmem.txt file. The regfile.txt contains 0x0000 for all 16 registers so they are all initialized to 0x0000.

As discussed in class we found that it was possible to implement the floating point instructions in a single clock cycle. To achieve this we had to use always @ (*) rather than always @ (posedge clk). While this would never pass timing constraints in a real chip this is possible in the simulator. Getting the timing correct would have more to do with the pipeline than implementing the float operations and for that reason passing timing was not a requirement for this assignment.

In addition to using Dr. Dietz's pipelined processor, AIK spec, and instruction encoding another resource we used was a previous 480 student's code linked here. We talked to Dr. Dietz about whether it was acceptable to use this resource and he said that it was. But he gave us 3 pieces of advice. 1. Cite the reference. 2. Don't copy exactly. 3. Don't assume it's correct. We followed this advice closely by being careful to only use the resource as a logical reference and to thoroughly test our modules. Dr. Dietz said that our instruction set (and as a result our processor) was different enough that he didn't have any issues with us using the code as a logical reference for the floating point ops.

3 Issues

During development we had to resolve several issues with our processor. When we first started out we were using a pipelined processor from one of our member's previous groups. While working on ITOF we noticed that every other instruction was not happening because of interlock issues. We troubleshot this for a while by trying to make use of the interlock system which seemed to be implemented in the processor and alternately by trying to add in our own. We didn't have success with either method. Ultimately this was resolved by switching over to using Dr. Dietz's processor.

There were several occasions where we thought our float operation modules were not working properly but after debugging we realized we had incorrectly generated the instructions. Either the instruction was not the one we were trying to test or the operands were not what we expected. These issues had simple solutions but took a lot of time to discover.

We tried to use to the Barrel Shifter code provided in Dr. Dietz's slides but we could never modify it to handle our inputs correctly. The values we needed to shift were not 8-bits and we couldn't figure out how to change the Barrel Shifter to accommodate different size inputs. We ended up using the << and >> operators instead.

Our ITOF module wasn't handling negative integers properly at first. We discovered that the issue was our lead0s counter returning 0 leading 0s for all negative integers. This makes sense because the sign bit of a negative integer is 1. To fix the problem we took the two's complement of the negative integer before sending it to the lead0s module.

Our ADDF and SUBF modules ended up being nearly identical with the exception of toggling the sign bit of Op2 for SUBF. This was done because 5 - 3 is the same and 5 + -3. Even though these modules are so similar we did keep ADDF and SUBF as separate modules just in case we discovered the need for different logic for the two operations.

In ADDF and SUBF we needed to determine which operand was larger to appropriately set the sign of the result. At first our code only compared the two mantissas when trying to determine which operand is larger. This assumes that the two operands exponents are always the same, which is obviously not the case. So we had to change the code to first compare the exponents of the two operands. If the exponents are different we know which operand is larger and we can set the sign of the output at that point. If the exponents are the same then we needed to compare the mantissas of the operands to decide which one is bigger. It should be noted that when adding and subtracting floating point numbers you do shift the smaller operand to have the same exponent as the larger one for part of the operation. However it was still necessary for us to determine which incoming operand had the larger exponent to set the sign correctly.

One other issue we had with ADDF and SUBF is that we weren't getting the correct results for cases where the output's exponent is smaller than either of the operands' exponents. One example is -2 + 3 = -1. This situation involves the number of leading zeros in the intermediate register temp_man. To handle this special case we had to add another if/else block to our modules and do one of two operations to the larger of the two operands exponents'. For one case the output exponent is equal to the larger of the two operands' exponents - 1 - num_zeros. For the other case the output exponent is equal to the larger of the two operands' exponents + 1 - num_zeros.

One last issue of note was with our FTOI module. It seemed to be properly converting 0 as well as numbers >1 or <-1 but it was incorrectly returning a converted value of 0 for 1 and -1. After debugging we realized this was due to an error in our ternary operator which set the output in the case of 0. The expression we used had exp_less_bias > 0 when we needed to have exp_less_bias >= 0 to get 1 and -1 to convert properly.

4 Testing

To test each module we identified common test cases and then verified the correct output using GTKwave. Test cases included operands with similar exponents, operands with different exponents, operations with 0, operations with two positive numbers, operations with two negative numbers, and operations with one positive and one negative number. For ITOF and FTOI we tested known special cases such as the denormal 0. For one positive and one negative operand we even tested with each operand in the Op2 field (ex. -3 + 2 and 2 + -3). By no means was the processor exhaustively tested but we do feel that we thoroughly tested common cases as well as a few special cases for each instruction.