Assignment 4: Floating Implementor's Notes

Abu-Taha Abdulfattah
tat_489@hotmail.com
Casey O'kane
casey.okane@uky.edu
LiangLiang Zheng
lzh229@g.uky.edu

***Abstract*—The goal of this assignment involved implementing the floating point based instructions of the IDIOT instruction set. The newly adapted ALU was then interfaced with the provided pipeline solution and a series of tests were created to ensure module and interface correctness.**

I. General Approach

Similar to the approach suggested by Dr. Dietz in the assignment prompt, the ALU received most of the group's early attention. First, the floating point instructions were implemented using the ALU module of the provided **pipe.v** file. Then after integrating this ALU with floating point this ALU was then placed inside of the **floatpipe.v** file and a changes were made to account for the floating point instructions, as detailed below.

II. Implementation

This section describes how each instruction was implemented.

A. Pipeline Solution

The general pipeline solution is taken verbatim from the assignment 3 solution that Dr. Dietz posted on the course webpage, with the addition of the floating point implementations (that are detailed below). The only difference between the code presented and the code provided is the ALU floating point instructions. This decision was made to decrease the possibility of error.

B. Floating Point Instructions

1) Integer to Float: For integer to float, the sign bit is first determined and stored in the signBit register. After finding the sign, the mantissa is constructed by creating a buffer of 24 bits, which contains the absolute value of the input with 8 bits of zero.The mantissa is constructed in the slice of the buffer contained from $15-numZero+7$ to $15-numZero$ to account for offset. After the mantissa, the exponent is found by adding the hexadecimal value *0x7f* to 15, which is then subtracted by the number of zeros. After all of these individual portions are collected, they are concatenated together to form the resulting floating point value.

2) Float to Integer: In the float to integer instruction, first the case where the input is zero is accounted for. After checking for zero, a 24 bit buffer is instantiated (**valueNew**) with the 16 leading 1's concatenated with the input mantissa. After creating the buffer, the exponent sign is checked (by comparing it with 127). If the exponent is positive, then the exponent value is taken from the input (with an offset of 127) and is used to shift the created buffer (like a barrel shifter) and the result is taken from the 16 most significant bits. If the sign bit is set, then the 2's complement of the result is found to display the proper value.

3) Float Addition: To start the addition instruction several cases are checked first (determining in the any of the inputs are zero or they are cancelled out) and then the main conditional branch is entered. Inside the main conditional branch the larger value is determined by comparing exponents and the difference between the two is found and used to shift the mantissa of the smaller input. Then logic is used to account for input values of different signs and to account for under/overflow conditions. If the sign of the input values are the same then the mantissa values are added and subtracted otherwise. The exponent value is collected from the larger input (plus 1) and the resulting mantissa is is offset by one position and placed in the mantissa variable. Ultimately, these values are concatenated together to provide the result

As it stands right now, the under/overflow logic is incomplete and presents a serious issue with the add instruction. This is detailed further in the **Issues** section of this report.

4) Float Multiply: To multiply two floating point numbers, the sign bit of the result is found first by xoring the sign bits of each of the input values. After finding the sign, the mantissas are updated to include a leading 1 bit and these new values are multiplied together. The resulting mantissa is found by taking bits 13:6 to account for multiplication offset. After the mantissa is formed, the exponents for each of the input values are recorded (into variables **expVal1** and **expVal2** respectively and the exponent value is found by adding these individual exponents and subtracting the specified bias of 127. After each of these components are calculated, they are concatenated together to form the result.

5) Float Inverse: After a numerous algorithms were provided by Dr. Dietz, the implementation for the inverse float

operation was finally settled after all proposed solutions were presented and it was decided that the ideal tactic would involve the use of a lookup table **reqFiles/recip.vmem**. After finding the sign bit, the reciprocal value of the mantissa is found by indexing the lookup table at the mantissa address. After the mantissa is collected the exponent is found by finding the difference between 254 and the value of input exponent (if the mantissa is equal to zero), if the mantissa bits do not result in the value zero however, the result must be shifted and the difference between 253 is used to record the exponent value. Finally, after all of the components have been constructed, the sign, exponent and mantissa are concatenated together to form the inverse result.

III. Testing

A. ALU Module

To test the new ALU module, there is a separate series of tests which use a combination of .vmem files that consist of hex values that represent different X,Y,Z and Opcode values (stored in the /tests directory). There are four tests associated with each instruction, usually testing simple cases or boundary cases. The user can see the results simply by casting "make", which will compile the code and generate the output which is structured to show the current opcode, x/y/z values and the expected value.

1) Results: For all of the 20 test cases, only two seem to fail. These cases involve the multiply and add float instructions. These issues are detailed later on in this report.

B. Integrated Pipeline ALU

The test bench module simply instantiates and starts the processor running. The processor reads **pipeinst.vmem** when reset, then the test bench starts the clock. There are many debugging text displays that tell when memory writes happen, when instructions are seen, and so on. The accompanying set of instructions can be used to test the processor.

For the most part, the test instructions provided by Dr. Dietz in assignment 3 were used, with the addition of a small sz instruction that would be used to stop an li instruction. In order to test the floating point instructions however, the end of *nottaken* branch was utilized to test all five instructions for correctness. First register 13 was assigned the floating point value of 0.5, and then the inverse was found to produce a result of 2.0. This 2.0 value was then converted to an integer and then back to a float using the fi2 and i2f commands respectively. After this, the result was multiplied with register 14 (1.5) to form the result 3.0. This result is then added to register 15 (-3.0) and then values are assigned to set up a jump to the *good* branch. If the instructions executed correctly then this jump should be ignored and the sys instruction should be called and initiate a halt. The source code for this test is found as **/reqFiles/assemblyInput** and the aik assembled code that is read by the processor is found as **/regFiles/pipeinst.vmem**.

1) Results: Prior to the implementation of the addf instruction, the processor seemed to execute ideally and would halt (with a slightly different assembly source) at the right time. However, before submission it seemed like GTKwave was displaying incorrect values. This is detailed further in the **Issues** section of this report.

2) Inefficiency: Using the procedural code is most likely not the most efficient. The logic could have been implemented using many more modules and specifying wires and gates, but it would greatly expand the programming time. Also, the way memory is accessed leads to a very complex memory with more ports than it should. This is admittedly inefficient hardware but had efficient coding at the current time.

IV. Issues

A. Notable Workarounds

1) Negative Number Leading Zeros Block: One issue that the group experienced involved finding the leading zeros for numbers that were negative. To combat this, an always block was utilized before the leading zeros module was instantiated so that if the number was negative, that the its positive sign binary representation would have been used to find the number of leading zeros. This was placed in an always block as checking for the sign bit later (inside the instruction implementation) since the **lead0s** module could not be instantiated inside the always block where the instructions are handled.

B. Features Not implemented

1) addf (about 25% complete): Currently, while the **addf** instruction does show some level of implementation, it isn't working as desired. While it works for simple cases (like ones involving zero) it doesn't account for more complex issues like over/underflow. This is something that would hopefully be ironed out with additional time.

C. Known Errors

1) addf implementation: As stated previously, the addf implementation has many issues in it's logic that were not corrected before the submission of this report. As a result, it is important that we stress that the full implementation of this instruction is a realized error.

2) mulf for larger numbers: While it is expected that some multiplication solutions might be off (especially as results get more and more precise), after testing many values (one of which is shown in the **ALUTest** directory) it was realized that for more precise values, that the solution was differ from the value expected. This is something that the group hoped to look into more if more time was allowed.

This might be the result of not adding exponent enable bit to the exponent values of the inputs and the bias but the team wasn't sure.

3) Execution not displaying properly in GTKwave: Following the integration of the floating instructions with that of the provided **pipe.v** file, the test file (**reqFiles/pipeinst.vmem**) was read into the processor as detailed in the **Testing** section above. Also mentioned is that the results seen in GTKwave

were not shown as was desired, specifically, the pc[15:0] didn't seem to be pointing in the right direction (even for the provided test file).

The team was unable to discern why this testing was not working, so it was decided to mention the issue here.

V. Theory

The operation are not sufficient to cover most floating-point needs, we still are missing other operations most floating-point units have (ex. Division, square root, etc.). Since we are missing the compare instructions the floating point unit can still work but it?s not up to modern day standards.

The other operations are necessary if we are designing a floating point unit for the industry but it is not important for our project. It would be a lot harder to implement the other operations using just the integer instructions. Since the provided processor already has some inefficiency the payoff in performance would be present.

It would be harder to implement the IEEE standard binary 16 format. So it is less useful for this assignment but it would be a more versatile tool. Our processor will be suited for some floating point computations but not all of them.