Assignment 4: Floating Implementor's Notes

Dylan Wright
dylan.wright@uky.edu
Robert McGillivary

Evan Whitmer

***Abstract*—This document serves as the Implementor's Notes for Team2C's EE 480 Assignment 4. This document intends to describe the methodology used to implement the project as well as any difficulties and important details regarding this task.**

I. Definitions and Standards

A. IDIOT

IDIOT (Instruction Definition In Our Target) is the instruction set implemented by this project. IDIOT is the instruction set created for the Spring 2016 version of EE 480.[1]

B. AIK

AIK (Assembler Interpretor from Kentucky) is the tool used to create assembly language programs in the IDIOT ISA.[2]

C. BAD

BAD (Barely Acceptable Decimal) is the floating point format implemented by this project. BAD is a specification developed for the Spring 2016 version of EE 480.[3] This format is a variant of IEEE Std 754 `binary32`.[4] BAD drops all subnormal values except for positive zero and truncates the 16 least significant bits of the significand.

II. Implementation

A. Framework

This is a Make project. The entire project can be built using the root Makefile and any directory with a Makefile can be built using that. The project is split into a `doc/`, `IDIOT/` and `verilog/` directory.

1) `doc/`: This directory is dedicated to documentation. In particular it contains the LaTeXsource files for the implementor's notes (this document).

2) `IDIOT/`: This directory is dedicated to the IDIOT specification and the test program build framework. Of particular interest are the following files:

a) `aik.py`: This file is a python3 script that can be used to interface with the AIK cgi interface hosted on `aggregate.org`. This script greatly simplifies the process of building individual test programs.

b) `build.sh`: This file is a bash script that can be used to build all .idiot programs in the directory. This produces the .vmem files which can be used to exercise the processor.

3) `verilog/`: This directory is dedicated to the verilog implementation of the project and the framework used to test it. Of particular interest are the following files:

a) `pipe.v`: This file is the verilog source code which implements the processor.

b) `test.sh`: This file is used to test the processor. At the moment it runs the unit tests in `tests/testbenches/`.

B. Floating Point Instructions

The following instructions were added to the previously implemented integer instructions. They are described in the order they were implemented.

1) `i2f`: (integer to float) is an instruction which converts a 16-bit integer to a 16-bit float. This instruction converts the x signal of the ALU. This is done by normalizing the integer, determining the exponent, sign. For the subnormal value positive 0, the input is passed directly through the ALU.

2) `f2i`: (float to integer) is an instruction which converts a 16-bit float to a 16-bit integer. This instruction converts the x signal of the ALU. This is done by denormalizing the significand and adjusting the signal if the sign bit is set.

3) `mulf`: (multiply float) is an instruction which multiplies two 16-bit floats. This instruction multiplies the x and y signals of the ALU. This is done by adding the exponents, multiplying the significands, and setting the sign bit. For the subnormal value positive 0, the output is set to positive 0.

4) `addf`: (add float) is an instruction which adds two 16-bit floats. This instruction adds the x and y signals of the ALU. This is done by denormalizing the float with the smaller exponent, adding or subtracting the significands, and normalizing the result. For the subnormal value positive 0, the input which is not positive 0 is passed directly through the ALU.

5) `invf`: (invert float) is an instruction which computes the inverse of a 16-bit float. This instruction inverts the x signal of the ALU. This is done using a look up table provided for the assignment.[5] For the special case of the subnormal positive zero, zero is passed through the alu.

C. Pipeline Integration

The floating point alu was built into the alu module from the previous project. This module is instantiated by the pipelined processor provided as the solution to the previous project.[6]

III. Issues

In order to use the provided `idioc` compiler, the following changes were made:

- Fixed issue with function labels missing an underscore
- Fixed issue with data words being output as a list of values which was not being accepted by AIK

IV. Testing

A. Continuous Integration

In an effort to increase productivity and test usefulness, this group has been using a Jenkins continuous integration server. This server is running on a server Dylan Wright operates.[1] The server is configured to attempt to build whenever the project GitHub receives a commit. The build process makes all source files and reports if any step fails. Additionally it runs the unit test script (`verilog/test.sh`) which will cause the build to fail if any test case fails. The testscript also outputs a junit xml report to `reports/junit.xml`. The xml file is used to generated test result graphs by Jenkins.

B. Floating Point Unit Testing

The test script tests floating point operations with the unittests in the `verilog/tests/testbench/` directory. The files with the .v extensions are inserted into the testbench in the main verilog file.

C. Pipeline Program Testing

The test script tests the pipeline by using the unittests in the `verilog/tests/testprogs/` directory. The files with the .vmem extensions are concatenated together and the resulting file is used to initialize the memory. The testbench in the main verilog file by default (not floating point unittests) resets the processor and runs until the processor halts.

V. Philosophy

A. Sufficiency

The floating point instruction set is sufficient for the context of this project. Implementing features like comparison in hardware would be a nice addition but can be easily done by the compiler instead of the hardware. By leaving much of the complexity in the software domain the hardware is simpler to design and test.

B. Necessity

The floating point instructions implemented here represent a distinct advantage over implementation in software. These instructions occur often enough to warrant implementation in hardware. Taking more than one cycle to perform a floating point multiply or addition would significantly affect the performance of certain programs.

[1]intellproject.com:8090

C. Usefulness

The format implemented here is mostly academic but it is used in some real world situations. The larger dynamic range of this format is particularly advantageous. While binary 16 is also a useful format, the top half of binary 32 is equally useful in this context.

References

[1] H. Dietz. (2016) Idiot reference material. [Online]. Available: http://aggregate.org/EE480/idiot.html

[2] H. G. Dietz and W. R. Dieter, "Aik, the assembler interpreter from kentucky," Aggregate.Org online technical report, University of Kentucky, Tech. Rep., April 2007. [Online]. Available: http://aggregate.org/AIK/aik.pdf

[3] H. Dietz. (2016) Assignment 4: Floating. [Online]. Available: http://aggregate.org/EE480/a4.html

[4] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std. 754-2008*, pp. 1 –70, 2008.

[5] H. Dietz. (2016) `invf` look-up table.

[6] ——. (2016) pipe.v. [Online]. Available: http://aggregate.org/EE480/pipe.v