Timing Errors In Floating Point Hardware

Ryan Jobson

Summer Research at Northwestern University

rjobson1@swarthmore.edu

Abstract

In the field of approximate computing, it is necessary to understand the kinds of errors that occur in the computation process. Understanding where errors are occur in hardware gives a better picture on how to generate accurate results. In addition, discounting known failure points can give give better guarantees on the accuracy of others. Finally, tradeoffs between power and reliability are now possible. To address these problems and advance the field of approximate computing, it is important to collect actual data on the failure rates of hardware. Focusing a single piece of hardware, a floating point unit or FPU, error rates were collected. In this study, error rates were different based on the operation (Addition, Subtraction, Multiplication...) and error occurred more frequently in different bit locations. Understanding these results will help to build more accurate as well as more consistent approximate hardware.

Introduction

Tracking error rates is an important step in building better approximate computers. Knowing where errors occur, and how likely they are to occur, will allow for more reliability and accuracy of results, while still utilizing lower power.

IEEE Floating Point

In 1985, in attempt to simplify the issues of floating point, the Institute of Electrical and Electronics Engineers developed a standard for representing floating point numbers. This standard has been adopted by all processors on the market. In addition to defining formats for arithmetic, the standard also includes rules on different forms of rounding. It is no surprise, then that the FPU chosen for this research conforms to the standard. However, it is first important to understand how floating point numbers are represented in computing hardware.

Generally, programming languages today include a floating point data type. This data type is used to represent both very large and very small numbers, with differing levels of precision. This is possible through the use of scientific notation. For example, the number 43.314 could be represented in scientific notation as $4.3314 * 10^1$. However, this methodology must be altered, due to the fact that computer hardware works only in binary. Therefore, the number must be represented in base two. Using a similar example, the number 4.5 in base ten, converted to binary would be 100.1 in base two, which could also be represented as $1.001 * 2^2$ in scientific notation.

In most floating point implementations, the bit width is 32, meaning that 32 bits are used to represent a single value. These bits can be broken into three distinct parts: Sign, Exponent, and Mantissa. Only one bit is reserved for the sign, where the sign bit of one represents a negative value while a zero represents a positive value. The next eight bits represent the exponent in base two. Finally, the remaining 23 bits are reserved for the mantissa, which represents all of the digits after the decimal point. In summary, the format can be represented as:. However, this presents a problem, how can numbers smaller than one be represented? The solution to this problem, is to subtract what is known as a bias from the exponent. For the 32 bit architecture, 127 is subtracted from the exponent. Therefore, there is a maximum value of 2127 and a minimum of 2-127 that can be represented in this standard. In conclusion, numbers in the IEEE-754 standard can be represented as such: -1^(Sign Bit)* 1.{Mantissa} * 2^(Exponent - Bias). These variables can and will also be changed to work with other implementations, such as the 64 bit and 16 bit FPU for this research.

Building the Infrastructure

As a starting point, an open source Floating Point Unit was used. Recommended by my advisor, the code provided included a well tested Verilog code for a 32 bit (Single Precision) FPU. However, the last update to the code was in 2001, over 15 years ago. As such, the provided testbench and test vector generation did not work. In addition, for our experiment, there needed to be a way to do 64 bit (Double Precision) and 16 bit (Half Precision) calculation as well. Consequently, the provided source needed to be modified.

The base FPU flow can be broken into a three stage process: A prenormalization phase, the calculation phase, and the post normalization phase. The prenormalization can be further broken into two distinct processes, one for addition and subtraction, and the other for multiplication and division. For the addition and subtraction pre normalization process, the two operands: a and b, are modified so that they share the same exponent. This can be accomplished by shifting the mantissa of the smaller operand by the difference of the two exponents. Additionally, overflow, underflow and other errors are checked as well. The pre normalization process for multiplication and division are much simpler, where only errors are checked. Next, the actual operation occurs. Finally, in the shared post normalization process, the final result is converted to its simplest form, checked for errors, and outputted. In parallel, the exceptions block will check for any errors that occur during runtime. This entire process completes from start to finish in four cycles. As a benefit of its modular design, every stage is pipelined, so the FPU can be given new inputs every clock cycle. To fit our research specifications, this hardware needed to be modified.

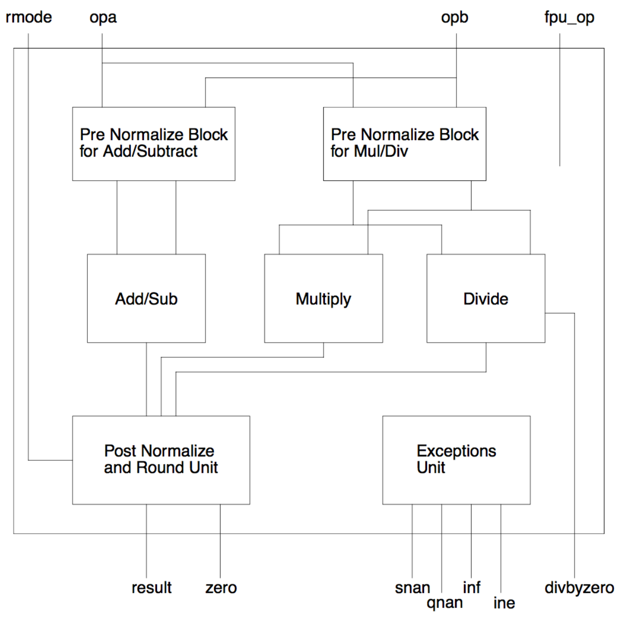
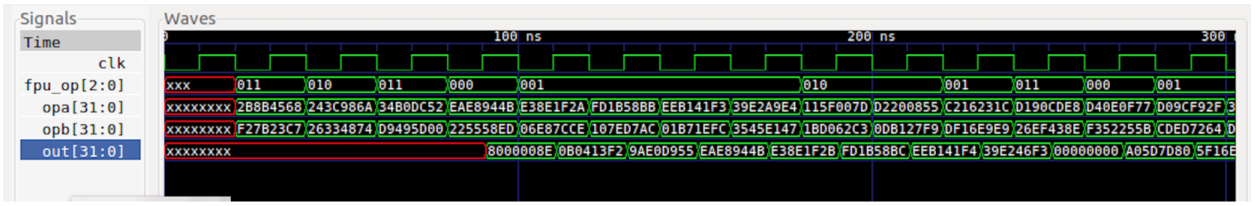

rmode
opa
opb
fpu_op
Pre Normalize Block for Add/Subtract
Pre Normalize Block for Mul/Div
Add/Sub
Multiply
Divide
Post Normalize and Round Unit
Exceptions Unit
result
zero
snan
qnan
inf
ine
divbyzero

Figure 1: FPU Pipeline

Over the course of a few weeks, the original design was modified, line by line, to fit the research specification. In the Verilog netlist, all of the hardcoded values were parameterized to fit for different bit widths. In addition, some of the logic in the pre and post normalization processes were modified. Once the Verilog was changed, the different FPUs needed to be tested. A testing framework was developed in Python to generate test vectors, as well as verify the outputs. In addition, Working in the IEEE-754 standard meant that all values were represented in binary. For readability and ease of use, all values were converted to hexadecimal values. Additionally, another script was developed for floating point conversion. Once the functionality was thoroughly tested, gate level synthesis could occur.

To understand how the floating point unit would behave in the real world, a gate level synthesis of the Verilog code needed to occur. This process simplifies the entire design down to basic gates. However, in this process, division could not be synthesized without adding sizable area to the chip, as well as adding more complex logic to the design. Unfortunately, the division operation had to be removed from the design. Once the gate level design synthesized, analysis on timing could be run. The gate level design, consisting of only basic gates, had information on the delays of every component in the design. Timing errors, where data arrives to a gate after a full clock cycles, could be located. Once the timing reports were generated, the infrastructure necessary to run the experiment was constructed.



```
fpu_op[2:0]  xxx      011      010      011      000      001                                                 010               001      011      000      001
opa[31:0]    xxxxxxxx 2B8B4568 243C986A 34B0DC52 EAE8944B E38E1F2A FD1B58BB EEB141F3 39E2A9E4 115F007D D2200855 C216231C D190CDE8 D40E0F77 D09CF92F 3
opb[31:0]    xxxxxxxx F27B23C7 26334874 D9495D00 225558ED 06E87CCE 107ED7AC 01B71EFC 3545E147 1BD062C3 0DB127F9 DF16E9E9 26EF438E F352255B CDED7264 D
out[31:0]    xxxxxxxx 8000008E 0B0413F2 9AE0D955 EAE8944B E38E1F2B FD1B58BC EEB141F4 39E246F3 00000000 A05D7D80 5F16E
```

Figure 2: Test Bench Waveform

Collecting Data and Final Results

Conducting the experiment after building the infrastructure was simple. A script would run generating an FPU and running a test vector at a given clock frequency, then the error rate would be recorded. Building upon that, error rate vs. clock frequency could be plotted. Running this, however, took a very long time to compile. After many days of processing, the error rates of both 32 bit and 64 bit FPU were calculated. In addition, a more detailed experiment was run, tracking exactly which category (sign, exponent or mantissa) was failing the most.

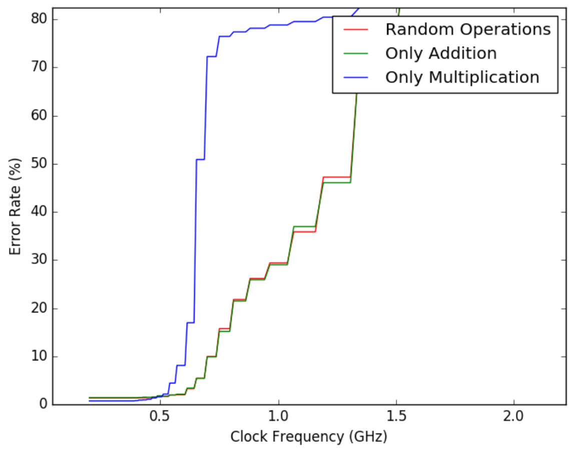


Figure 3: 32 Bit FPU Error Rates

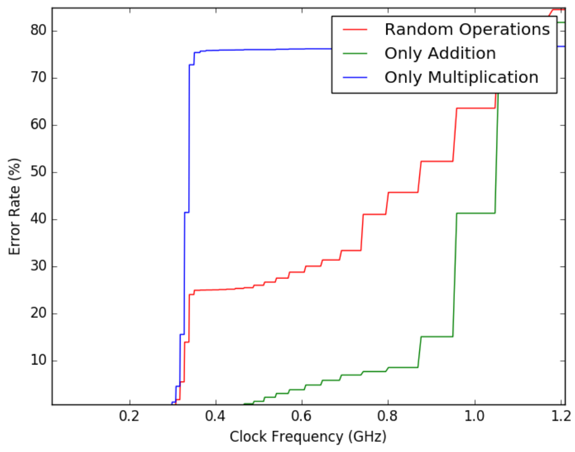


Figure 4: 64 Bit FPU Error Rates

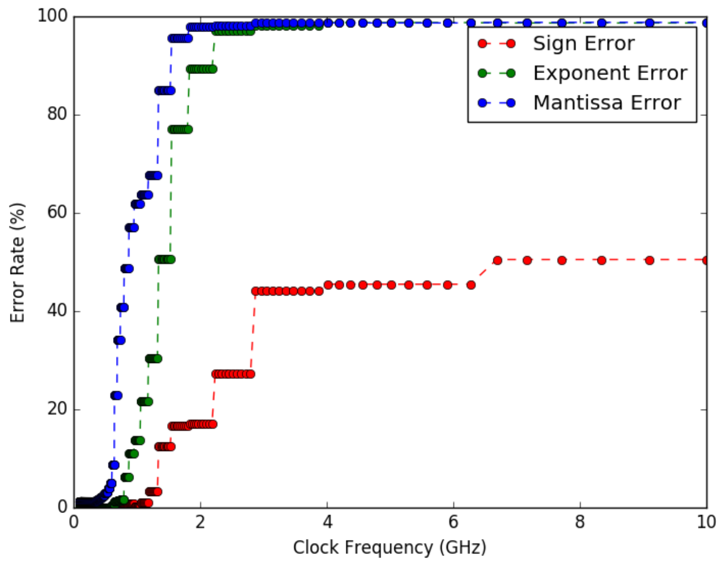


Figure 5: 32 Bit FPU Error Locations

Conclusion

Tracking error rates is an important step in building better approximate computers. Knowing where errors occur, and how likely they are to occur, will allow for more reliability and accuracy of results, while still utilizing lower power. Over the past 10 weeks, An infrastructure was built for synthesizing floating point units for a variety of bit widths, as well as a testing framework to verify the results. The results achieved can be used in the process of building better hardware.

Future Work

In addition to fixing the 16 bit architecture, pipeline depth can also be modified and tested, which might also have an effect on the error rate.

Acknowledgements

I would like to thank the DREU program for giving me this wonderful opportunity to study at Northwestern. I would like to thank Dr. Russ Joseph for guiding me through the project, being a mentor, and showing me around Chicago. I would also like to thank Professor Jie Gu as well as graduate students: Tianyu Jia and Yuanbo Fan, for help in the debugging process.

Citations

IEEE Standards Committee. "754-2008 IEEE standard for floating-point arithmetic." *IEEE Computer Society Std* 2008 (2008).

Overton, Michael L. *Numerical computing with IEEE floating point arithmetic*. Society for Industrial and Applied Mathematics, 2001.

Han, Jie, and Michael Orshansky. "Approximate computing: An emerging paradigm for energy-efficient design." *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 2013.