Study of an FPU

Yu-wen Pwu
Department of Computer Science, National Chiao Tung University
ywpu@cs.nctu.edu.tw

Abstract

This study examines how the floating point unit (FPU) is designed in modern computers. My study methods are including, but not limited to, source code analysis, behavioral simulation of the circuit and implement it onto a Xilinx Spartan-3E Starter Kit development board featuring Spartan-3 XC3S500E-4FG320C FPGA.

1. Introduction

The FPU core I am going to study is "*fpu*" by Rudolf Usselmann [1]. It complies the IEEE 754 standard, and is able to do addition, subtraction, multiplication and division operations for two single precision floating point numbers. I uses an IEEE paper format template for OpenOffice and LibreOffice to write this paper. The template is designed by fullmetalsoa [2].

Because I am so busy recently, I have little time to work on this project. I feel so sorry about that, but I have no choice… I only have 86,400 seconds a day! I have done my best.

2. FPU code analysis

The IEEE 754 floating point number representation specifies both single precision (4 bytes) and double precision (8 bytes) encodings. For single precision encodings, please refer to Table 1. The decimal values in the third row of the table is called **denormalized numbers**; value "*NaN"* in the fifth row means **not a number**.

The standard also defines five rounding types. The first one is **round half to even**, a.k.a. **unbiased rounding**, which rounds the fraction to the nearest even value; the second **round half away from zero**, which rounds the fraction to the nearest value above if it is positive, and to the nearest value below if it is negative; the third **round toward zero**, which truncates the fraction; the forth **round toward positive infinity**, which rounds the fraction up to its ceiling; the fifth **round toward negative infinity**, which rounds the fraction down to its floor [3]-[5].

Table 1. IEEE 754 single precision encodings

Sign (1 bits)	Exponent (8 bits)	Mantissa (23 bits)	Decimal Value
0/1	1 to 254	anything	$(-1)^S \times (1.M)_2 \times 2^{(E-127)}$
0/1	0	nonzero	$(-1)^S \times (0.M)_2 \times 2^{-126}$
0/1	0	0	$(-1)^S \times 0$
0/1	255	0	$(-1)^S \times \infty$
0/1	255	nonzero	NaN

2.1. Pre-normalization

For addition and subtraction operations, we need to align the exponents of the operands so that the two operands could have the same exponent. This can be done by right-shifting the mantissa of the operand with the smaller exponent. Thus we will need at least an integer comparator and a shift register. This procedure is demonstrated in module "*pre_norm*."

This module has seventy input signals and sixty-six output signals. The variable names of the I/O signals and their respective meanings are shown in Table 2.

After the pre-normalization procedure is done, we can simply add or subtract the integer mantissas and adjust the result. This relies on the arithmetic logic unit (ALU) and the post-normalization module, which will be discussed later.

For multiplication and division operations, our situation is quite simpler. Let $\mathbf{float_1} = \langle \mathbf{S_1, E_1, M_1} \rangle$ and $\mathbf{float_2} = \langle \mathbf{S_2, E_2, M_2} \rangle$, then $\mathbf{float_1} \times \mathbf{float_2}$ is roughly $\langle \mathbf{S_1 \oplus S_2, E_1{+}E_2, M_1{\times}M_2} \rangle$, and $\mathbf{float_1} / \mathbf{float_2}$ is $\langle \mathbf{S_1 \oplus S_2, E_1{-}E_2, M_1/M_2} \rangle$. Thus there is no need to align the exponents. However, we still need to calculate the exponent of the result.

The pre-normalization module for multiplication and division operations could be found in module "*pre_norm_fmul*." It is similar to module "*pre_norm*" mentioned earlier, but is a little bit less complex. After this procedure is done, the output data will also be transmitted to the ALU and the post-normalization module as described before.

Table 2. I/O ports of module "pre_norm"

Type	Name	Explanation
input	clk	clock
input	[1:0] rmode	rounding modes
input	add	+ rather than −
input	[31:0] opa	operand_1
input	[31:0] opb	operand_2
input	opa_nan	operand_1 is NaN
input	opb_nan	operand_2 is NaN
output	[26:0] fracta_out	mantissa of operand_1
output	[26:0] fractb_out	mantissa of operand_2
output	[7:0] exp_dn_out	exponent of both
output	sign	result is negative
output	nan_sign	result is NaN
output	result_zero_sign	result is 0
output	fasu_op	+ or − indeed

2.2. Top module and integer arithmetic

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec quis tincidunt lorem, at ultrices turpis. Etiam molestie et quam ut laoreet. Mauris nec est libero. In quis orci fringilla, tincidunt odio sit amet, imperdiet tortor. Donec elit ligula, mattis pulvinar bibendum id, porttitor at nibh. Integer laoreet nisl tempor felis pharetra, ac lacinia risus ultrices. Suspendisse rhoncus sem diam, eget molestie eros egestas sit amet. Pellentesque ut urna odio.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec quis tincidunt lorem, at ultrices turpis. Etiam molestie et quam ut laoreet. Mauris nec est libero. In quis orci fringilla, tincidunt odio sit amet, imperdiet tortor. Donec elit ligula, mattis pulvinar bibendum id, porttitor at nibh. Integer laoreet nisl tempor felis pharetra, ac lacinia risus ultrices. Suspendisse rhoncus sem diam, eget molestie eros egestas sit amet. Pellentesque ut urna odio.

2.3. Post-normalization and rounding

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed fringilla ac velit ut vehicula. Nullam sit amet blandit nisl. Duis cursus turpis nisl, at lacinia ante scelerisque eu. Suspendisse sit amet consectetur nunc. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla pellentesque, nibh eget sodales aliquam, orci justo fermentum erat, quis efficitur diam turpis eget sem. Vivamus vel turpis ut massa sollicitudin volutpat. Sed bibendum sagittis mauris in vulputate. Phasellus vel risus lobortis, maximus lectus porttitor, fringilla ipsum. Cras nec tortor congue, malesuada metus facilisis, vehicula elit. Donec lectus lectus, viverra ac orci non, ornare commodo tortor. Praesent rhoncus at justo a cursus. Fusce aliquet condimentum justo a finibus.

3. Porting to FPGA

I have not ported the FPU core to an FPGA yet due to insufficient time.

4. Experiments

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi ultrices cursus augue. Donec a rhoncus erat. Nulla finibus iaculis orci. Integer viverra lacus et accumsan blandit. Phasellus ut nunc feugiat, consequat felis id, ultricies urna. Donec at leo egestas, ultrices odio vitae, rutrum justo. In auctor felis sed velit tincidunt, quis elementum ipsum sollicitudin. Vivamus laoreet sollicitudin est non dictum. Nulla suscipit id diam at commodo. Vivamus scelerisque, libero at pulvinar mattis, turpis erat rhoncus dolor, a tristique velit nunc vel nisl. Vivamus finibus felis id dignissim placerat. Quisque scelerisque, risus at consectetur pulvinar, nunc metus luctus nunc, ac tempus libero elit a lorem.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis lacinia mi eget tristique. Suspendisse potenti. Nam condimentum odio eu risus imperdiet molestie. Sed eget urna fermentum, volutpat dolor sed, luctus nibh. Ut quis erat erat. Duis faucibus elit metus. Duis tempus est sed eros cursus, eu pretium libero pellentesque. Aliquam erat volutpat. Donec posuere, est non aliquet efficitur, justo nisi dictum diam, eu finibus leo odio non mi. Sed id urna nec nisl vehicula maximus a ut nibh. Mauris tempus magna dictum, facilisis velit nec, ultrices ligula. Sed vestibulum, neque id posuere euismod, est dolor posuere est, eu tristique lectus ipsum ut tortor. Proin eget elit sit amet nisl fringilla tempor et nec felis. Integer imperdiet suscipit sodales. Etiam pharetra porta purus sed vulputate.

5. Conclusions

This study remind me of the inaccuracy of floating point numbers. Consider the conditions in Figure 1 and Figure 2. The former is a simple program written in C++, the latter in Python. They both show that IEEE 754 floating point numbers are just approximate representations of decimals. If high precision is required, we would better use fixed point numbers. The main drawback of fixed point numbers is that they are not flexible and may result in poor efficiency.

Another thing I want to point out is that Xilinx Core Generator already have had an "*Floating-Point Operator*" IP core! Moreover, the core generator offers a handy configuration interface so that we can easily customize our cores. I think reusing ready-made products to enhance our productivity is also an indispensable skill for engineers.

Figure 1. The accuracy issue with C++ on Linux

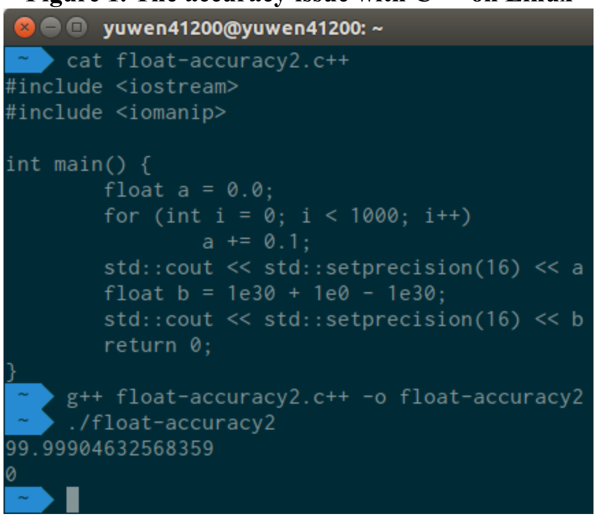
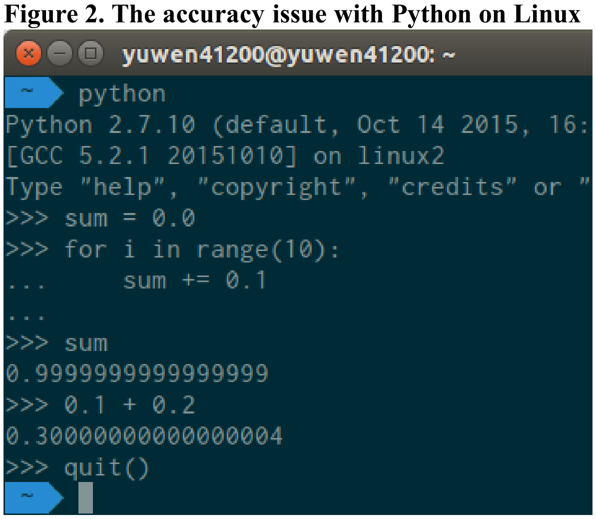


Figure 2. The accuracy issue with Python on Linux



6. References

[1] Rudolf Usselmann (2000, September 16). *Open Floating Point Unit*, The Free IP Cores Projects [Online]. Available: http://opencores.org/project,fpu

[2] fullmetalsoa. *Author's Instruction for the Preparation of Regular Papers* [Online]. Available: http://templates.openoffice.org/en/template/ieee-paper-format

[3] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, August 29, 2008.

[4] various contributors (2016, January 9). *Floating Point*, Wikipedia [Online]. Available: https://en.wikipedia.org/wiki/Floating_point

[5] Kevin J. Brewer and Quanfei Wen (1999, May 28). *IEEE-754 Floating-Point Conversion from Floating-Point to Hexadecimal*, City University of New York [Online]. Available: http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html

[6] Milo M. K. Martin, *CIS 371 Computer Organization and Design, Unit 7: Floating Point*, University of Pennsylvania Computer and Information Sciences Department, Philadelphia, PA, 2008.

[7] Chun-Jen Tsai, *FPU Design Example*, National Chiao Tung University Department of Computer Science, Hsinchu, Taiwan, 2015.