

Study of an FPU

Yu-wen Pwu

Department of Computer Science, National Chiao Tung University
ywpw@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 sixth 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.

Once 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 $\text{float}_1 = \langle S_1, E_1, M_1 \rangle$ and $\text{float}_2 = \langle S_2, E_2, M_2 \rangle$, then $\text{float}_1 \times \text{float}_2$ is roughly $\langle S_1 \oplus S_2, E_1 + E_2, M_1 \times M_2 \rangle$, and $\text{float}_1 / \text{float}_2$ is $\langle S_1 \oplus S_2, E_1 - E_2, M_1 / M_2 \rangle$. 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_fm*.” 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	system clock
input	[1:0] rmode	rounding mode
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

The top module of this FPU is module “*fpu*.” It handles the control flow of the whole core. After finishing pre-normalization, it sends the integer mantissas to their corresponding integer arithmetic units. These are defined in module “*add_sub27*,” “*mul_r2*” and “*div_r2*,” respectively. Nevertheless, these three modules may not be synthesizable due to the use of complex arithmetic operators. Whether they could be synthesized or not depend on the synthesis tool we use. If our synthesis tool cannot do them automatically, we may need to implement them manually.

The Xilinx ISE Design Suite uses Xilinx Synthesis Technology (XST) by default, but the design suit also comes with Xilinx Core Generator. The core generator includes several complex integer arithmetic cores, so we could simply instantiate the core we need from the core generator.

2.3. Post-normalization and rounding

After the previous operations are done, the final phase is to post-normalize and round the output. These operations can be found in module “*post_norm*.” We must adjust the format of the result so that it fits the IEEE 754 standard. This also counts on several shift registers.

The module “*post_norm*” is more complicated than pre-normalization modules because it also need to perform the rounding, whereas the former ones need not. The rounding modes also follow the specification in IEEE 754. At the end of the source code, there are a few lines enclosed by // **synopsys translate_off** and // **synopsys translate_on**. These comments are named “*foundation express directives*,” and they tell some synthesis tools that the codes within the comments are for simulations only. In this example, these lines are about delaying signals and displaying errors.

Finally, we need a module to deal with exceptions. That is the module “*except*.” This module takes the system clock and two 32-bit operands as input, then it determines if NaN, infinity or zero condition occurs. These output information is used by module “*fpu*,” “*pre_norm*” and “*post_norm*.” Module “*post_norm*” also contains output ports indicating overflow and underflow conditions.

In software design, we use **throw** statement and **try...catch** block to handle exceptions, while in hardware design, for each special or unexpected case, we use a dedicated signal to indicate whether it happens.

3. Porting to FPGA

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

4. Experiments

I am going to examine the correctness of the design by taking module “*pre_norm*” as an example. The waveform from the ISE Simulator (Isim) is shown in Figure 1 and Figure 2. This is a behavioral simulation for module “*pre_norm*,” and the test bench written for it is demonstrated in module “*pre_norm_tb*.”

Because the exponent of **opa** is smaller than that of **opb**, **exp_dn_out** equals to the exponent of **opb**. The exponent of **opa** needs to be increased by one, so its mantissa should be shifted right one bit. This becomes the value of **fractb_out**; please note that the output of the mantissas are sorted.

For the second testing dataset (at 20ns), it requests to add a negative number to a positive one. That is why signal **fasu_op** is low; the actual operation it should perform is subtraction. Furthermore, since the subtrahend, i.e. **opa**, is smaller than the minuend, i.e. **opb**, the signal **sign** is high, which means the result is negative. Signal **result_zero_sign** is also high due to the value “11” of **rmode**. This sets the rounding mode to “*floor*,” and makes the result become zero.

For the third and the forth testing datasets (at 30ns and 40ns, respectively), their output signals can be explained in the similar manner.

Figure 1. Waveform for module “pre_norm”

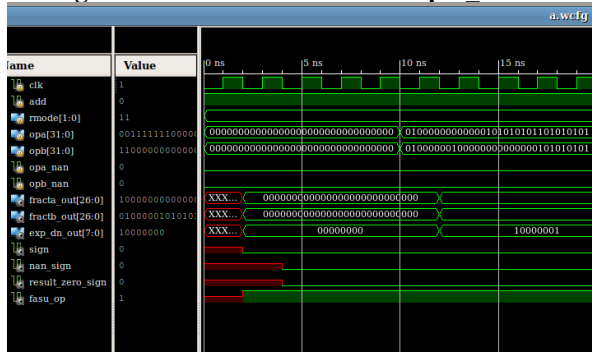
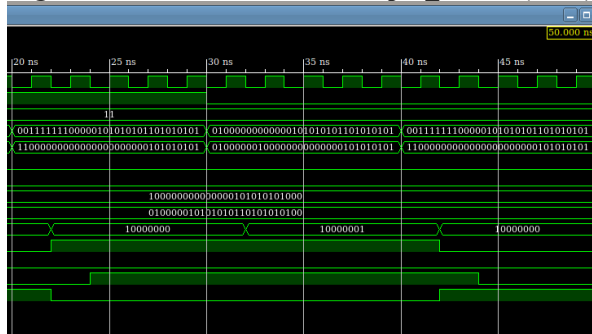


Figure 2. Waveform for module “pre_norm” (cont.)



5. Conclusions

This study remind me of the inaccuracy of floating point numbers. Consider the conditions in Figure 3 and Figure 4. 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.

6. References

- [1] Rudolf Usselman (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.

Figure 3. The accuracy issue with C++ on Linux

```
yuwen41200@yuwen41200: ~
cat float-accuracy2.c++
#include <iostream>
#include <iomanip>

int main() {
    float a = 0.0;
    for (int i = 0; i < 1000; i++)
        a += 0.1;
    std::cout << std::setprecision(16) << a
    float b = 1e30 + 1e0 - 1e30;
    std::cout << std::setprecision(16) << b
    return 0;
}

g++ float-accuracy2.c++ -o float-accuracy2
./float-accuracy2
99.99904632568359
0
```

Figure 4. The accuracy issue with Python on Linux

```
yuwen41200@yuwen41200: ~
python
Python 2.7.10 (default, Oct 14 2015, 16:
[GCC 5.2.1 20151010] on linux2
Type "help", "copyright", "credits" or "
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
>>> 0.1 + 0.2
0.30000000000000004
>>> quit()
```