FPU

IEEE 754 based floating point unit

By:

- Abdulrahman Ragab Hashim
- Hussin Mostafa Saied
- Abdullah Khaled Kamal
- Mahmoud Hassan Muhammad
- Abdullah Mohamed Abo-Elmagd

Supervisor:

- Dr. Muhammad Mahmoud Mohammad Ibrahim

Table of content

Chapter 1: *IEE-754 Intro*

1.1 Implementation:

Floating point numbers are usually a multiple of word. The representation of a MIPS floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

These formats go beyond MIPS. They are part of the IEEE 754 floating-point standard, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

Before we go on board we have to bring these notes with us:

1- Number must be normalized in an understandable language (1.01000 * 2^E not 101.000* 2^E-3)

2- Exponents are biased by 127 which means 0 = 127 to get the the real exponent from IEEE754 one subtract 127 from it.

1.2 Special cases:

Exponent	Fraction	Number
0	0	0
0	Non Zero	± Denormalized number
1-254	Anything	± Floating point number
255	0	± INF
255	Non Zero	NaN

Chapter 2: *Adder/subtractor*

2.1 Implementation:

2.1.1 Claculating fraction:

Implicit bit is ignored in IEEE754 format , so we add it to fraction of inputs.

Add bit fraction [23] implicit = 1

Add another bit to fraction [24] = carryBit [24] (check for over flow and under flow also for aligning exponent)

2.1.2 Aligning exponents:

- Check exponents to get the biggest exponent.
- get the positive difference between them.
- shift the smallest fraction by the amount of the positive difference .
- As shifting results in lost data , it has be done to number with least exponent to lose data in the LSBs not the MSB to obtain higher accuracy.

Example:

$$2^5 \times (1.10101101) + 2^2 \times (1.10101101) =$$

$$2^5 \times (1.10101101) + 2^5 \times (0.000110101\boxed{101})$$

Lost data

2.1.3 Selecting operation:

1. Adding:

   Fill the selector variable with zero.

2. Subtracting:

   Fill the selector variable with one.

2.1.4 Operation on Input

1. Adding:

   Fraction of the sum: simply adding fractions of the operands.

2. Checking the carry bit

   If it equals 1 shift the sum right and adding 1 to the biggest exponent (number must be normalized).

3. Subtracting:
   - Subtract the smallest operand from the biggest number.
   - The sign bit of the sum is the sign of the biggest number.
     - If the implicit bit is 0.

       Shift till it be 1 and Subtract 1 from the exponent of the result in each shift but making sure that the exponent is not equal 0(Implemented with Multiplexers without FSM which save time but consume hardware).

2.2 Special Cases:

Operation	Result
INF+INF	INF
INF-INF	NAN
NAN ± any Number	NAN
INF ± any normal number	INF
ZERO + Number	That Number
Any other Number – the same Number	ZERO
Zero ± zero	Zero

Chapter 3: *Multiplier*

3.1 Implementation:

First we should concatenate the implicite one at the beginning so we now have the full number (1+fraction), using the synthesizable multiplication operator, we can calculate the result as descriped below.

3.1.1 Claculating the result:

1- Output fraction = Fraction1 * fraction2

2- Output exponent is the sum of input exponents

3- If the most significant bit in output fraction is 1 then leave the most significant bit and take the following 23 bits as fraction and increase the resuting exponent by one if it's not leave the first 2 bits and take the following 23 bits with no change to the output exponent

3.2 Special cases:

NaN*(Anything)	NaN
Inf * 0	NaN
Inf * (R - {0})	Inf
0 * (Anything)	0
Overflow	inf

Chapter 4: *divider*

4.1 Implementation:

4.1.1 Calculating fraction:

When dividing each input we can easily notice that the result has the form of “1.(mantissa)” or “0. (mantissa) ” and if the result is “0. (mantissa) ”the most bit in mantissa must be 1

- **note_1** : division process as FSM is not synthesizable so look at 1.3 to know what we do "
- **note_2** : in IEEE 754” representation delete the implicit but to calculate division we must restore it "

Formula 4.1:

If the result is “1.(mantissa)”

dividing each input gives result so

mantissa = Result

If the result is“0.(mantissa)”

dividing each input gives result so

mantissa = Result after lift shifting , to normalize it"

4.1.2 Calculating exponents:

We know that to calculate exponents we just subtract them as we learned at the primary school nothing amazing here.

But if we calculate ".11/.1"the expected result is ".11" but the actual result is "1.1" so we subtract 1 from the Result exponent.

And also If the result of division has the form of“0.(mantissa)” we need to normalize it so we subtract another 1

Formula 4.2:

If the result has the form“1.(mantissa)”

Result exponent = exponent_dividend - exponent_divisor - 1

If the result has the form“0.(mantissa)”

Resulting exponent = exponent_dividend - exponent_divisor - 2

And if the "exponent_dividend > exponent_divisor"

The most bit must be 1,to be bigger than 127

And if the "exponent_dividend < exponent_divisor"

The most bit must be 0 , to be smaller than 127

4.2 Special cases:

Inf/Inf	NaN
Zero / Zero	NaN
Number/Inf	Zero
NaN/ Number	NaN
Inf/ Number	Inf
Number/Zero	Inf
Number/NAN	NaN

Chapter 5: *FPU and Test Bench*

5.1 FPU Truth Table:

Funct	Output
00	A + B
01	A – B
10	A / B
11	A X B
XX	ZZ

5.2Test Bench:

Output is compared to excepted value and if the difference is less than the desired value the case is thought to be passed with no errors.

5.2.2 Test cases generation:

- A list of all permutations of special cases is generated
- A list of all random numbers is generated
- A list of the two lists is generated which is the final list

5.2.2 how random floating point numbers is generated with C

- Two random integer numbers are divided .
- The random floating number can be multiplied by fixed amount to control the size of it
- The numbers could be transformed to binary form using the concept of UNINON

```
//contains the same binary data
typedef union container
{
    float ieee_754;
    int binary_decimal;
}   f_i;
void Float_binary(char  *c ,float f)
{
f_i contains;
contains.ieee_754=f;
int sign=(contains.binary_decimal&0x80000000)==0x80000000;
contains.binary_decimal=contains.binary_decimal&0x7fffffff;
for (int i=31; i>=0;i--)
{//add 48 to convert to ASCII
*(c+i)=(contains.binary_decimal&0x7fffffff)%2+48;
contains.binary_decimal=contains.binary_decimal/2;
}
if (sign==1)
*(c+0)='1';
*(c+32)='_';
*(c+33)='\0';
//terimnate ASCII
}
```

Chapter 5: *fpuGUI*

5.1 Usage:

An interactive software made for testing our implementation of fpu using a simple grphical user interface

5.2 Toolkit:

1- Tcl (programming language)

2- Tk (gui library)

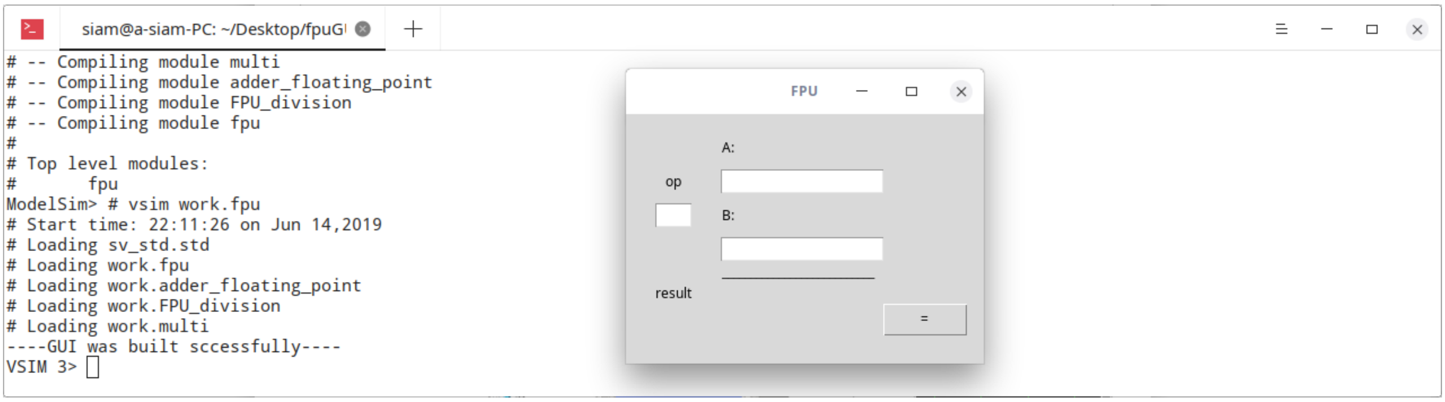
3- python3 (programming language)

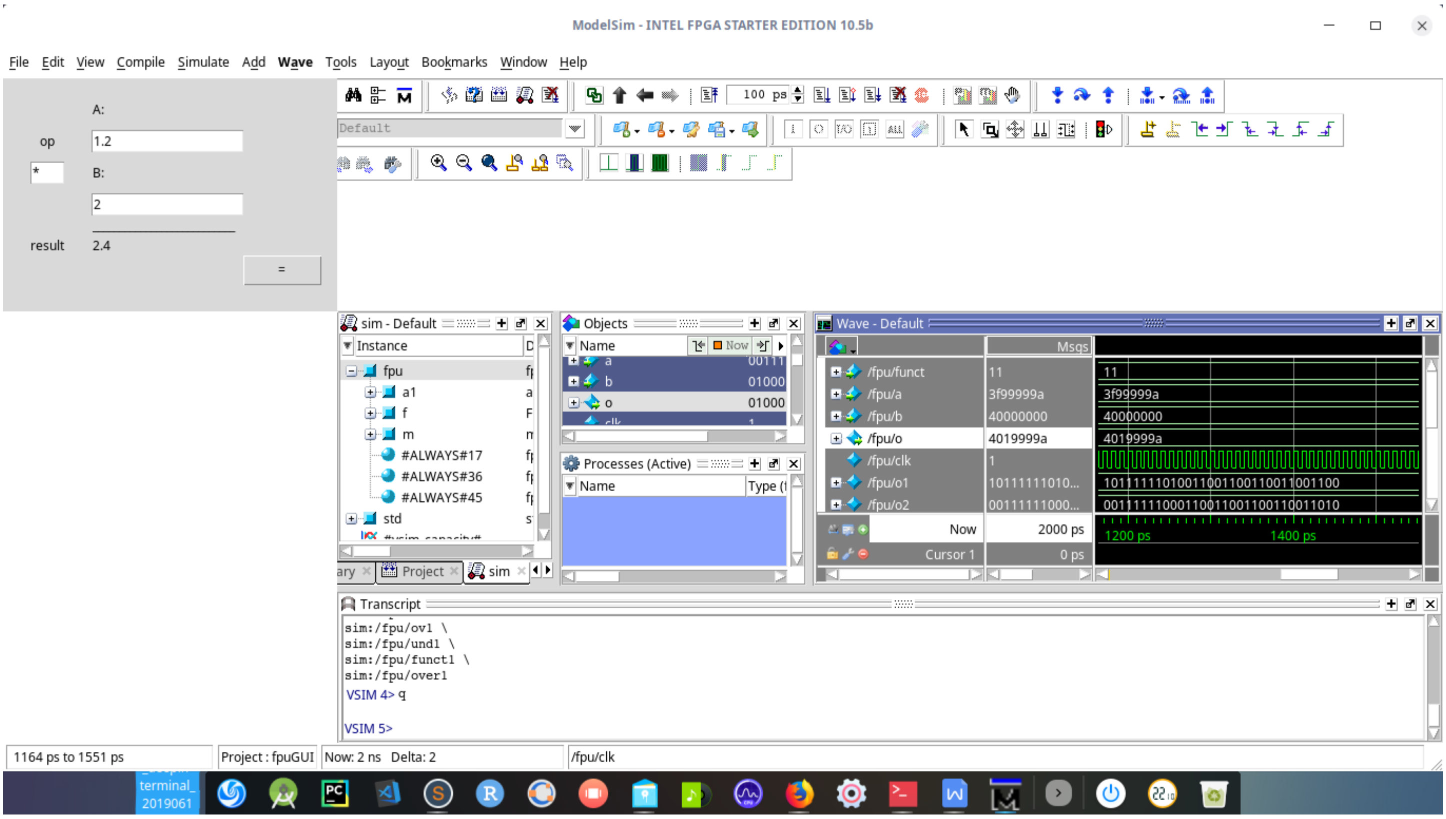
4- Modelsim interpreter for macro (.do) files

5.2 Requirements:

1-unix based system (Linux /Mac) with python and modelsim interpreters installed

5.3 Screenshots:





References

- Computer Organization and Design MIPS Edition
- Think OS A Brief Introduction to Operating Systems by by Allen B. Downey