FPGA implementation of floating point processor for Programmable calculator

Saurabh Gupta and Vikas Kumar Sharma
Department of Electrical Engineering, IIT Kanpur

Abstract— **Floating Point Unit (FPU) is an essential part of most of the processors. For handling computations of numbers of high order it is nessesary to operate on floating point numbers. Exception handling is essential and the FPU designed gives appropriate exception signals that can be used by any processor.**

Further, in this project a processor for a programmable calculator is designed that does arithmetic operations on floating point numbers.

I. Problem Statement

Design and implementation of a FPU (Floating Point Unit) which operates on IEEE 754 represented numbers on FPGA. Also, designing a processor that uses this FPU to make a programmable calculator.

II. Design Issues

A. FPU

Floating point numbers can handle much larger numbers than the fixed point and for scientific calculations the order of numbers is typically high. Hence, floating point arithmetic is chosen for the design. Though double precision floating point representation gives better accuracy, the hardware requirements for FPGA implementation are doubled if the unit operates with double precision. Hence, single precision best fits to the requirements.

B. Processor

Then the issue of deciding the ISA (Instruction Set Architecture) is faced. 16-bit instructions, in which first 5 bits are for operation-code, next 3 bits are destination address followed by 6 bits that contain addresses of two operand addresses (each of three bits), were accepted for design.

Implementation of move operation requires inputting a 32 bit sequence. This was resolved by reading three instructions at a time and if move operation is detected, then the second and the third instructions will be merged to form a 32-bit sequence. This methodology reduces the size of instruction by 32 bits which is otherwise must be used to input a 32 bit sequence.

The program that the user writes in the Instruction Memory of the system has a requirement that after all the instructions, one ending instruction should be written containing all zeros. Also the destination of the last result that comes out of ALU should be written back in last register (address = 3'b111).

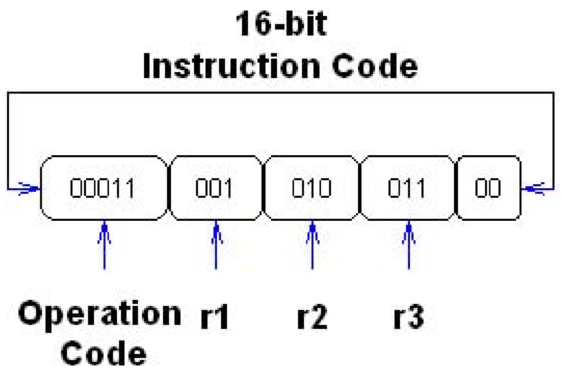

Fig. 1. Instruction Set Encoding Scheme

III. Design Methodology

For representing the floating point numbers, IEEE 754 standard was selected and the FPU is designed for the same.

Studying some famous algorithms both for signed and unsigned arithmetic operations, it was found that for addition and subtraction the signed algorithms are best. But, for multiplication and division unsigned algorithms are used. As the arithmetic is to be done on floating point numbers, so we handled mantissa and exponent separately. Also in division for better accuracy the dividend was extended to double of its length and the answer is rounded off to nearest integer.

The next step in the design flow was to write the hardware description for the Floating point unit. These were simulated and verified. Then for the processor units simulation and verification of hardware description was done.

Implementation of verified codes on FPGA for Floating point unit as well as for the whole processor was the last step.

IV. Floating Point Arithmetic

A. IEEE 754 standard

The IEEE 754 standard [1] for the representation of floating point numbers is followed. In this standard the first bit is sign bit (0 for positive number and 1 for negative number), next 8 bits are for exponent (with a bias of 127). It means that if exponent is 8'b01111111 then the actual exponent of 2 in the representation will be 127-127=0. Therefore this representation also handles negative exponents without using any sign bit. The remaining 23 bits are for storing mantissa. Actually the

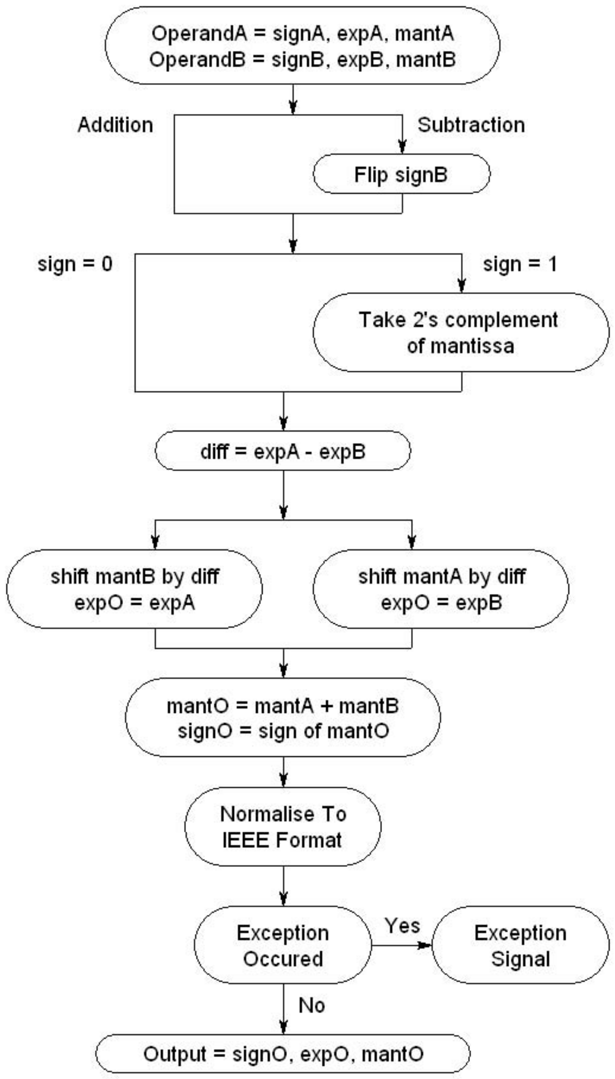
OperandA = signA, expA, mantA
OperandB = signB, expB, mantB
Addition
Subtraction
Flip signB
sign = 0
sign = 1
Take 2's complement of mantissa
diff = expA - expB
shift mantB by diff expO = expA
shift mantA by diff expO = expB
mantO = mantA + mantB signO = sign of mantO
Normalise To IEEE Format
Exception Occured
Yes
Exception Signal
No
Output = signO, expO, mantO

Fig. 2. Block Diagram of Floating Point Addition-Subtraction

mantissa is of 24 bits and the first bit is always '1' in the representation. Hence, there is no need to represent this implicit 1 in the mantissa and the standard represents 24 bit mantissa in 23 bits. For example $1\_10000001\_10000101011001011101001 = (-1) * (1.10000101011001011101001) * (2^{10000001})$

B. Addition and Subtraction

Floating point addition involves much more hardware than an integer addition as for adding two numbers we need to equate the exponents of the two numbers and this involves shifting of mantissa. If we shift the number by simple shifter many clock cycles are wasted. Therefore we have used barrel shifter that shifts any sequence in combinatorial way in just one cycle. The next step is signed addition of mantissa. If the number is negative, the 2's complement of the mantissa is fed into the signed-adder. Now for the further operations the result has to be again in IEEE 754 format, it means the 24 bit mantissa must have 1 as its MSB. For this we again used a block called Count Leading Zeros that gives the shift needed to convert the number in IEEE 754 format. At every shift in mantissa we take care of increment and decrement in exponent and wherever any overflow in exponent it is reported as exception. Using the same hardware, addition and subtraction both are implemented, as for subtraction we just need to alter the sign bit of the second operand. (Ref. fig.1)

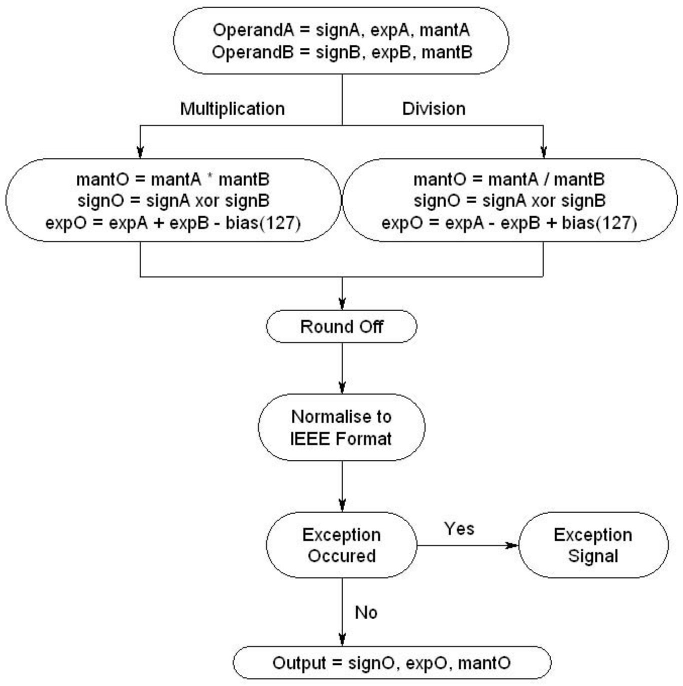
OperandA = signA, expA, mantA
OperandB = signB, expB, mantB
Multiplication
Division
mantO = mantA * mantB signO = signA xor signB expO = expA + expB - bias(127)
mantO = mantA / mantB signO = signA xor signB expO = expA - expB + bias(127)
Round Off
Normalise to IEEE Format
Exception Occured
Yes
Exception Signal
No
Output = signO, expO, mantO

Fig. 3. Block Diagram of Floating Point Multiplication and Division

C. Multiplication

The multiplication of floating point number is mainly multiplication of their mantissas. The two mantissas are multiplied by Booth's Algorithm[3]. The 48 bit multiplication result is again converted into the IEEE format using the leading zeros counter and the barrel shifter. Then it is rounded off to nearest integer. The exponents of operand1 and operand2 are added and 'bias' is subtracted from the result. The shift due to barrel shifter is suitably adjusted in the exponent. If any overflow is observed then the exception is reported. (Ref. fig.2)

D. Division

The division mainly involves the division of mantissas[4] and it is done by extending the dividend to twice of its length. This is done for better accuracy of the result. This result is converted into the IEEE format using the leading zeros counter and the barrel shifter followed by rounding off to the nearest integer. Now the exponents are subtracted and 'bias' is added taking care of the shift due to barrel shifter. If the exponent overflows or becomes negative then exception is reported. Divide by zero is reported as invalid operation. (Ref. fig.2)

If any of the operand is infinity then the invalid operation is reported as exception.

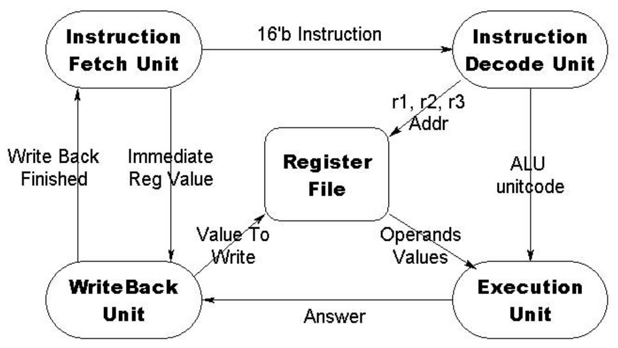

Fig. 4. Block Diagram of the Processor

V. Processor Design

1) Instruction fetch unit at reset assigns the program counter to zero, reads the instruction from the program memory and fetches that to instruction decoder unit. Processor gets next increased counter value also.
2) Instruction decoder gets 16 bit instruction and gives operand addresses, destination address and ALU unit code. Those addresses are passed to register file and ALU unit code is given to the execution unit.
3) Execution unit takes operand values from register file and gives the result to Write Back unit.
4) Write Back unit takes the result form execution unit and at move command it takes data vale from the instruction memory. This value is passed to register file and stored to the given destination address.

For other processor units (IF, ID, EX and WB), we used trigger and finish pulses to activate the subsequent units. Finish of one unit triggers the start of appropriate subsequent unit. This process continues till all the instructions are executed and the result is written back to the final register.

VI. FPGA Implementation

For implementing the FPU on FPGA verified Hardware description were synthesized. The syntesization results are tabulated in TABLE I and TABLE II. TABLE I contains results for FPU and TABLE II contains results for processor units. Clock cycles used by complete processor are not reported as it depends on the program (instruction set) given by the user.

TABLE I
Results of FPGA implementation of FPU

	Logic elements used	Memory Bits used	Clock cycles used
Addition-Subtraction	6	0	4
Multiplication	1202	8896	33
Division	685	0	54
Complete Processor	5330	21568	–

TABLE II
Results of FPGA implementation of Processor Units

	Clock cycles used (simulation result)
Instruction Fetch Unit	1
Instruction Decode Unit	2
Register File Read	1
Write Back Unit	2
Register File Write	1

VII. Further Extentions

1) Further by the same algorithms implementing arithmetic of double precision floating point numbers.
2) Jump and branch operations can be included in our processor, by which a lot of clock cycles can be saved.
3) Square Root calculation is an generally used operation so, it can also be included in FPU.
4) In multiplication the barrel shifter used has same structure as the shifter in other units. But, algorithm can work with an optimized shifter. This optimization can be easily included.

Acknowledgment

The authors would like to thank Veeramani V. and Abhinav Agarwal for their valuable support and fruitful discussions.

References

[1] Institute of Electrical and Electronics Engineers, ”IEEE Standard for Binary Floating-Point Arithmetic”, ANSI/IEEE Standard 754-1985, August 1985
[2] http://docs.sun.com/source/806-3568/ncg_goldberg.html
[3] http://en.wikipedia.org/wiki/Booth’s multiplication algorithm
[4] http://www.cs.wisc.edu/ smoler/x86text/lect.notes/arith.flpt.html