Floating Point Unit Implementation on FPGA

Deepa Saini [1], Bijender M'dia[2]
[1, 2](Electronics & Communication, M.D.University, INDIA

ABSTRACT

As densities of FPGA are increasing day by day, the feasibility of doing floating point calculations on FPGAs has improved. Moreover, recent works in FPGA architecture have changed the design tradeoff space by providing new fixed circuit functions which may be employed in floating-point computations. By using high density multiplier blocks and shift registers efficient computational unit can be developed. This paper evaluates the use of such blocks for the design of floating-point units including adder, subtractor, multiplier and divider.

1. Introduction:

Floating-point unit is a part of a computer system specially designed to carry out operations on floating point numbers. Addition, subtraction, multiplication, divisio n are the typical operation of floating point unit.Floating-point operations are often pipelined. In superscalar architectures without general out-of-order execution, floating-point operations were sometimes pipelined separately from integer operations.

Over the past few years the use of FPGAs in compute-intensive applications has been growing. The vast majority of applications have employed fixed-point arithmetic due to its smaller size. The key advantage of floating-point over fixed-point is its ability to automatically scale to accommodate a wide range of values using its exponent. Floating-point is thus preferred by programmers for non-integer computations when it is available on CPUs due to its ease of use. However, this scaling behavior comes at the cost of reduced accuracy. A 64-bit fixed point Representation can have more accuracy (but less range) than a 64-bit floating-point representation.

2. Floating Point Unit Organization:

The FPU chip performs all floating-point functions for microprocessor chip set. The FPU has two fully- pipelined execution units, allowing two floating-point mathematical operations and two floating-point memory operations every cycle. The FPU register file contains 32, 64-bit entries and has eight read ports and four write ports. Load and store data queues provide a pipelined interface between the IU and the FPU, streamlining data flow and minimizing unused cycles. The FPU offers peak performance of 300 double-precision MFLOPS with a clock frequency of 75 MHz The IU places floating-point instructions in the floating-point instruction queue in the IU. Each entry in the floating-point instruction queue is arranged as a quad word. The dispatch mechanism is similar to that of the IU: from Zero to four floating-point instructions can be dispatched by the IU to the FPU each cycle. Instructions are dispatched only when the FPU has adequate resources available to execute the instructions. The floating-point instruction queue provides temporary storage for floating-point instructions while any dependencies that might prevent the floating-point instructions from being executed (such as waiting for dependent loads to complete, etc.) are cleaned up.

Floating-point instruction dispatches and floating-point loads and stores to the data streaming cache are controlled by the IU. The IU is responsible for generating all address and control signals for floating-point loads and stores to the data streaming cache. Accesses that miss in the data streaming cache and require interfacing to main memory are handled by the off-chip cache controller. During these miss cycles, the FPU continues to execute floating point instructions already in the queue.

Once floating-point data is retrieved from the data streaming cache, it is placed in the load data queue. For store operations, the result is placed in the store data queue. As soon as the corresponding address information from the tag RAM is available, the data is written out to the data streaming cache. Figure 1 diagrams the floating-point data path.

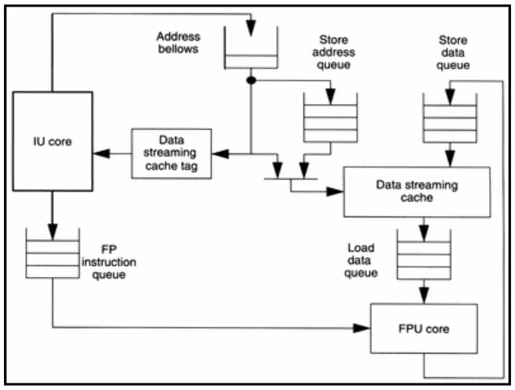


Figure1: Floating Point data path

3. Conceptual Overview:

The input operands are separated into their mantissa and exponent components. The comparison of the operands to determine which is larger only compares the exponents of the two operand, so in fact, if the exponents are equal then both the input numbers are treated equally to populate the registers.

This is not an issue because the reason the operands are compared is to find the operand with the larger exponent, so that the mantissa of the operand with the smaller exponent can be right shifted before performing the addition. If the exponents are equal, the mantissas are added without shifting.

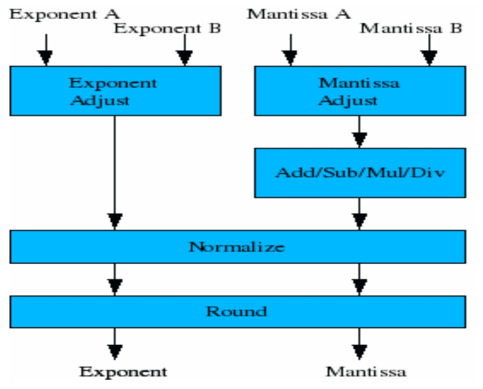


Figure 2: Conceptual overview of FPU

4. Floating Point Numbers

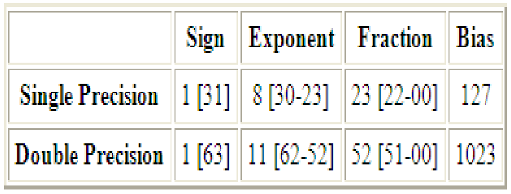
There are several ways to represent real numbers on computers. Floating-point representation - the most common solution - basically represents real in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23\text{abc} \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided. Floating point on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

4.1 STORAGE LAYOUT

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and doubles (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):



	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

Figure 3: Layout for single and double bit precision floating point values

4.1.1 THE SIGN BIT

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

4.1.2 THE EXPONENT

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. The exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

4.1.3 THE MANTISSA

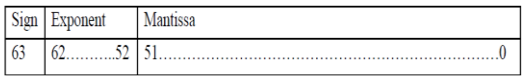
The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^0.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

4.2 Double Precision Floating Point Number

The IEEE 754 standard defines how double precision floating point number are represented. 64 bits are used to represent a double precision floating point number.



Sign	Exponent	Mantissa
63	62...........52	51...0

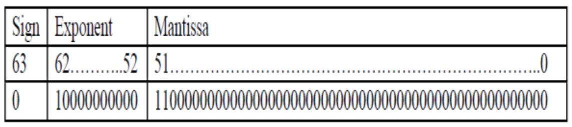
Figure 4: Double precision floating point number bit format

The sign bit occupies bit 63. '1' signifies a negative number, and '0' is a positive number. The exponent field is 11 bits long, occupying bits 62-52. The value in this 11-bit field is offset by 1023, so the actual exponent used to calculate the value of the number is 2^(e-1023). The mantissa is 52 bits long and occupies bits 51-0. There is a leading '1' that is not included in the mantissa, but it is part of the value of the number for all double precision floating point numbers with a value in the exponent field greater than 0. A 0 in the exponent field corresponds to a denormalized number, which is explained in the next section. The actual value of the double precision floating point number is the following:

Value = -1^(sign bit) * 2^(exponent – 1023) * 1.(mantissa)

(1.mantissa) being a base 2 representation of a number between 1 and 2, with 1 followed by a decimal point and the 52 bits of the mantissa.

For an example, how would the number 3.5 be represented in a double precision floating point format? The sign bit 63 is 0 to represent a positive number. The exponent will be 1024. This is calculated by breaking down 3.5 as (1.75) * 2^(1). The exponent offset is 1023, so you add 1023 + 1 to calculate the value for the exponent field. Therefore, bits 62-52 will be "1000000000". The mantissa corresponds to the 1.75, which is multiplied by the power of 2 (2^1) to get 3.5. The leading '1' is implied in the mantissa but not actually included in the 64-bit format. So .75 is represented by the mantissa. Bit 51, the highest bit of the mantissa, corresponds to 2^(-1). Bit 50 corresponds to 2^(-2), and this continues down to Bit 0 which corresponds to 2^(-52). To represent .75, bits 51 and 50 are 1's, and the rest of the bits are 0's. So 3.5 as a double-precision floating point number is:



Sign	Exponent	Mantissa
63	62...........52	51...0
0	10000000000	1100000000000000000000000000000000000000000000000000

Figure5: Double precision representation of 3.5

4.3 BASIC FLOATING POINT ARITHMETIC

The floating-point multiplier unit is the simplest of the arithmetic operations—the significand of the two operands are multiplied using a fixed-point multiplier and the exponents summed (the extra bias must be removed in the process). After multiplication the possibility of a one-bit overflow exists. Handling this and doing the desired rounding are then completed. In all our designs the rounding mode implemented is round to nearest even, which is the default for the IEEE standard. The sign bit of the result is the XOR of the operand sign bits. Note that in This and all other operations described below, the implied '1' of each significand is prepended at the outset of the computation then removed after its completion before the result is packed into the result word. Figure 6(a) shows a notional layout of a floating-point multiplier. It is drawn to reflect the relative sizes of its sub-parts that the majority of the area is consumed by the fixed-point significand multiplier when built from LUTs and flip flops.

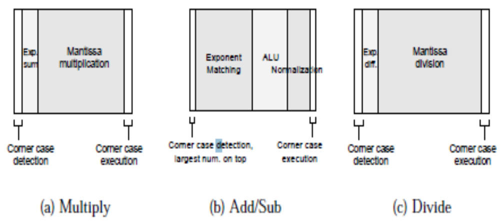


Figure 6: Floating Point Unit Floor Plan

Floating-point addition is much more complicated than multiplication. The first step is to compare the two operands' exponents to determine which is larger. The significand of the operand with the smaller exponent is then shifted right dictated by the difference in exponents. The two matched significands are then added or subtracted, depending
on the operands' sign bits. The result significand is then normalized to fall within the range by shifting and the exponent adjusted. Finally, rounding is done and the result packed into the output word. Figure 6(b), shows a notional layout for a floating point adder. Note that the exponent matching and normalization hardware dominate the area resources of the unit. Since the above adder requires an adder/subtractor as its core, subtraction of floating-point numbers is readily incorporated into the above design at the cost of a few gates' logic to determine when to add and when to subtract the significands.

A number of methods may be used for floating-point division in FPGAs. Division by reciprocal multiplication is discussed in both [1] and [2]. To accomplish this, the reciprocal of the denominator is computed via table lookup and then multiplied by the numerator. An bit significand requires a table with entries. This is problematic for anything other than small word sizes. A second approach that uses repeated multiplications to converge to the reciprocal of the denominator. In addition, for comparison purposes we present a restoring array divider design. The core of this array divider is the significand divider which consists of a series of stages, one per significand bit. Each stage consists of a subtractor and a multiplexor and two registers. As shown Figure 6(c) the array divider consumes the majority of the circuit area.

5 Floating Point Unit Ip Core

The floating point IP core is separated into 9 source files:
1. fpu_double.vhd (top level)
2. fpu_add.vhd
3. fpu_sub.vhd
4. fpu_mul.vhd
5. fpu_div.vhd
6. fpu_round.vhd
7. fpu_exceptions.vhd
8. fpupack.vhd
9. comppack.vhd

5.1 HIERARCHY:

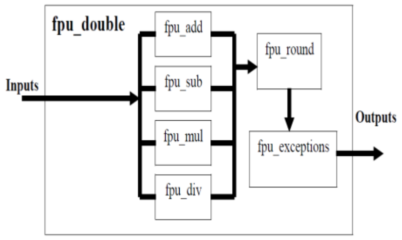


Figure 7: Hierarchy of various source file

5.2 TOP LEVEL

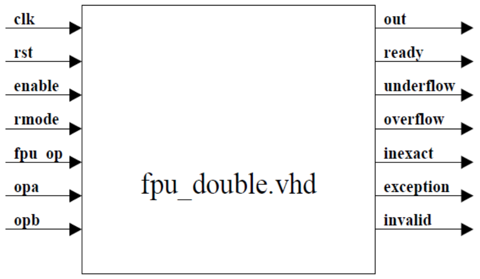


Figure 8: Top level module of Floating point unit

The input signals to the top level module are the following:
1. clk (global)
2. rst (global)
2. enable (set high to start operation)
3. rmode (rounding mode, 2 bits, 00 = nearest, 01 = zero,10 = pos inf, 11 = neg inf)
4. fpu_op (operation code, 3 bits, 000 = add, 001 = subtract, 010 = multiply, 011 = divide, others are not used)
5. opa, opb (input operands, 64 bits)

The output signals from the module are the following:
6. out_fp (output from operation, 64 bits)
7. ready (goes high when output is available)
8. underflow
9. overflow
10. inexact
11. exception
12. invalid

The top level, fpu_double, starts a counter (count_ready) one clock cycle after enable goes high. The counter (count_ready) counts up to the number of clock cycles required for the specific operation that is being performed. For addition, it counts to 20, for subtraction 21, for multiplication 24, and for division 71. Once count_ready reaches the specified final count, the ready signal goes high, and the output will be valid for the operation being performed. fpu_double contains the instantiations of the other 6 modules,
which are 6 separate source files of the 4 operations (add, subtract, multiply, divide) and the rounding module and exceptions module. If the fpu operation is addition, and one operand is positive and the other is negative, the fpu_double module will route the operation to the subtraction module. Likewise, if the operation called for is subtraction, and the A operand is positive and the B operand is negative, or if the A operand is negative and the B operand is positive, the fpu_double module will route the operation to the addition module. The sign will also be adjusted to the correct value depending on the specific case.

6 Simulation Result

The generic and Spartan3-E optimized designs were similar for the add/sub and multiplier units, with the optimized designs simply using shift registers and the 18x18 built-in multipliers. The divider units were fundamentally different from one another. The significand divider for the generic unit was a restoring array divider, while the optimized design used the 18x18 built-in multipliers. The word sizes tested which show the best performance for the optimizations presented include 16-bits (9-bit significand), 23-bits (16-bit significand), and 41-bits (32-bit significand).

This is due to there being a good match between the significand size and the width of the available multiplier blocks in Spartan-3E. In addition, a standard IEEE 32-bit format was run (23-bit significand) which shows less benefit due to not as good a match between significand size and multiplier block.

In a configurable computing environment there may be no special significance to using the standard IEEE word sizes other than they match what is used on CPUs, simplifying validation. In many cases, however, non-standard word sizes may be profitably employed Three different versions of each module are represented — the generic module, an optimized module which uses both built-in multipliers and shift registers. Of those area savings, the multiplier and divider the majority of the area savings was due to the use of the multiplier blocks Results of of synthesize and simulation are shown in figure 9 ,10 and 11.g

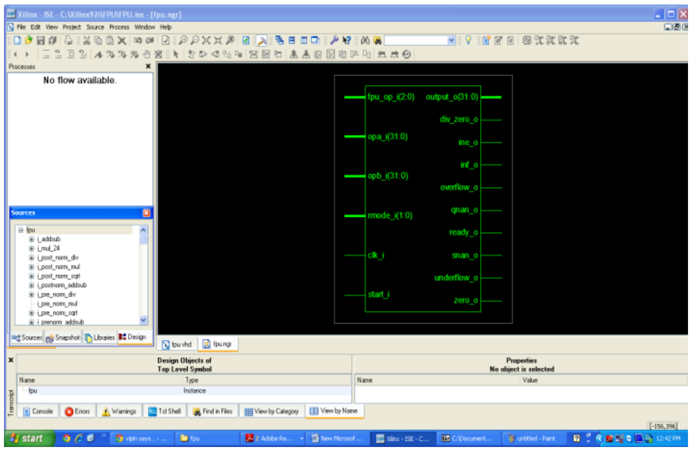


Figure 9: Synthesize Result of Floating Point unit

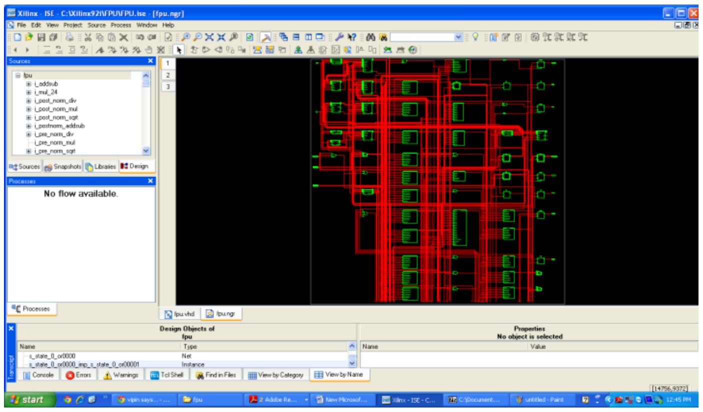
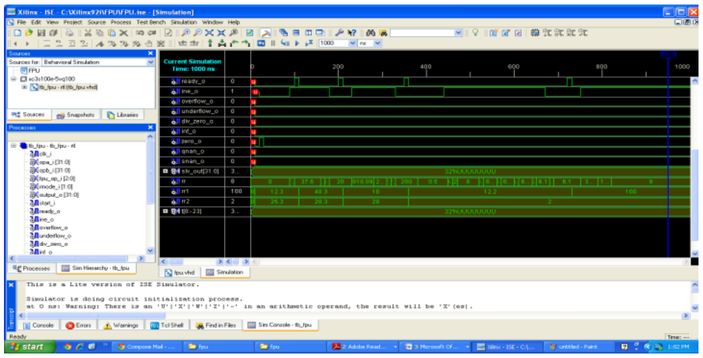


Figure 10: Detailed view of Floating point unit after synthesize



```
This is a Lite version of ISE Simulator.

Simulator is doing circuit initialization process.
at 0 ns: Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
```

Figure 11: Simulation Result of Floating Point unit

7, Conclusions

This paper see the effects of new FPGA features like multiplier blocks and shift registers on the designing of floating point unit that performs addition, subtraction ,multiplication and division. And research shows that area required by Floating point unit for doing multiplication and division by using newly added block is much less in comparison to floating point unit using LUT's FF's only.

References

[1] N. Shirazi, A.Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA-based custom computing machines," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, Eds., Napa, CA, Apr. 1995,pp. 155–163.

[2] Behrooz Parhami, *Computer Arithmetic*, Oxford Press, 2000.

[3] Joseph J. F. Cavanagh, *Digital Computer Arithmetic*, McGraw-Hill, 1984.

[4] *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg,
published in the March, 1991 issue of Computing Surveys. Copyright 1991, Association for Computing Machinery, Inc., reprinted by permission.

[5] LOW COST FLOATING-POINT UNIT DESIGN FOR AUDIO APPLICATIONS by *Sung-Won Lee and In-Cheol Park* Division of Electrical Engineering, Department of EECS, KAIST 373-1 Gusong-dong Yusong-gu, Taejon, 305-701, KOREA