Indian Journal of Science and Technology, Vol 9(33), DOI: 10.17485/ijst/2016/v9i33/92999, September 2016
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Design of Out-of-Order Floating-Point Unit

Sumanth Sridhar*, Sourabh Zunke, Kumar Vaibhav and M. Mohana Sundaram

Department of ECE, SRM University, Chennai-603203, Tamil Nadu, India; sumanth_sridhar@srmuniv.edu.in; sourabh_mahesh@srmuniv.edu.in, vaibhav_ranjitkumar@srmuniv.edu.in, mohanasundharam.m@ktr.srmuniv.ac.in

Abstract

Objective: Field Programmable Gate Arrays (FPGAs) are often used to accelerate hardware systems by implementing algorithms on hardware. This paper presents the design and implementation of a fully pipelined single-precision Floating-Point Unit (FPU) on a Spartan-6 FPGA chip. **Methods:** This paper presents a high-speed, modular design for improving the performance of such applications. While the proposed design is capable of performing basic arithmetic operations and square-root extraction, its modularity enables designers to add more functionality easily; or remove modules that they deem unnecessary for a particular application. **Findings:** The investigation shows that the adder and multiplier modules can be clocked at over 300 MHz and the top-module at over 200 MHz High operating frequencies were achieved by pre-computing possible values in earlier pipelining stages, then correcting results in later pipelining stages. It was also found that splitting longer operations in the critical path is a better alternative than processing the whole operation at once. Limiting "Max_Fanout", an attribute provided by Xilinx XST tool, proved valuable in reducing delays on overloaded nets. **Applications:** This FPU would be a worthwhile addition as a floating-point extension in fixed-point processors for applications such as spectrum analyzers, 3D graphics, and audio processing units.

Keywords: DSP48A1Multiplier, FPU, FPGA, High-speed Pipeline, Out-of-order Processing, Non-restoring Algorithm, Spartan-6, Single-Precision,

1. Introduction

Floating points are away to represent real numbers into binary format. IEEE754 standard[1] specifies two types of representation 32-bit single precision FPUs and 64-bit double precision FPU. Figure 1 shows IEEE75432-bit binary format.

The value of a number, N, is determined as follows:

$$N = (-1)^{s} \times \left(\sum_{i=0}^{23} M_{23-i} \times 2^{-i}\right) \times 2^{E-127} \quad (1)$$

In the 32-bit single-precision floating-point format, the exponent has a bias of +127. So, −127 will be represented as 0, and +128 will be represented as 255. The mantissa includes another bit, an implicit 1, that should be pre-pended during calculation. The radix point lies to the right of this implicit bit. So, the mantissa lies in the range of [1,2).

In this design, the FPU has been divided into four stages of operation namely, In-order issue of inputs, exception handling, out-of-order processing of arithmetic operations and issue of outputs to respective output registers. Out-of-order processing helps in preventing wastage of cycles during pipelined execution of algorithms[2]. In designing the multiplication module, a single DSP multiplier DSP48A1 available with Xilinx Spartan 6 FPGA was used[3]. The division and square root modules were performed using non-restoring algorithms[4,5].

2. Design and Implementation

The following designs were implemented in the FPU:

2.1 Floating-point Multiplication

Floating point multiplication is by far the simplest of the coreoperations[6]. It is done by ExOR-ing the sign bits, multiplying the mantissas, and adding the exponents[7]. Abias value of 127 is subtracted from exponents to get the resultant exponent value. The critical path of this sub-module lies along the multiplier used for the mantissa.

*Author for correspondence

The mantissa multiplication is implemented as 5-cycle latency pipelined multiplier using a single DSP48A1 embedded multiplier unit on the Spartan-6FPGA fabric[8]. Each DSP slice can perform 18 × 18 signed multiplications and also includes pipeline registers and a 48-bit accumulator. Since the multiplication of two 24-bit mantissas generates a 48-bit result, the DSP slice can be effectively used. The 5-cycle pipe line is implemented as an FSM as shown in Figure 2.

The choice to use a single DSP unit presents a trade-off between multiplier through put and the number of DSP units used. The proposed multiplier can also be replaced by 4DSP units to achieve maximum through put as shown on pg. 28 of the DSP48A1 Slice Use Guide[3].

As shown in Figure 3, a modified form of this model is used to implement the pipeline. AU and BU are the higher 7 bits of the mantissas. A Land BL are the lower 17 bits of the mantissas. EN is the enable signal for the accumulator. Sel 1 and Sel 2 are select signals for the multiplexers. All three are SET or RESET by the FSM.

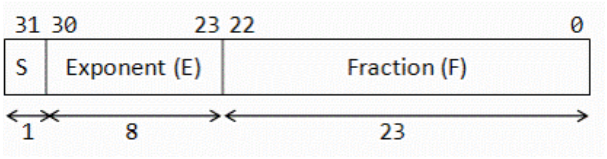

Figure 1. IEEE-754 single-precision binary format.

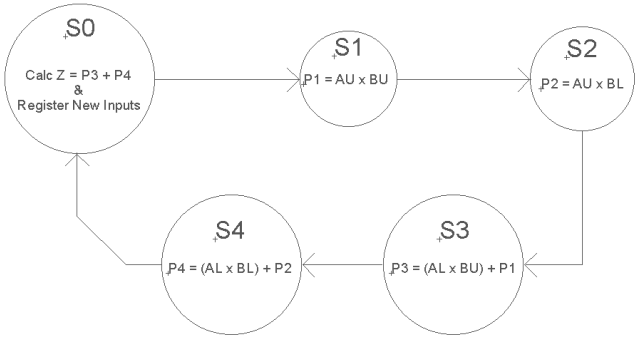

Figure 2. State Diagram of Mantissa Multiplication.

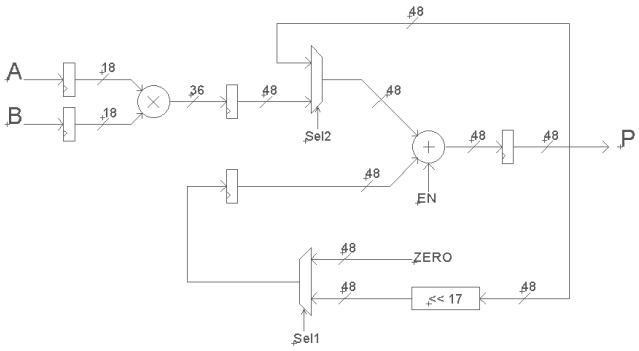

Figure 3. Mantissa Multiplication using DSP48A1

2.2 Floating-point Adder/Subtractor

Floating point addition is done by performing the following operations[9]:

$$S_O = (S_X \wedge gt) \vee ((op \oplus S_Y) \wedge (S_Y \vee (\overline{gt} \wedge \overline{eq}))) \quad (2)$$

$$M_O = \begin{cases} M_X + (M_Y * 2(E_Y - E_X)), & if\ E_X > E_Y \\ (M_X * 2(E_X - E_Y)) + M_Y), & if\ E_X < E_Y \end{cases} \quad (3)$$

$$E_O = \begin{cases} E_X\ , & if\ E_X > E_Y \\ E_Y, & if\ E_X < E_Y \end{cases} \quad (4)$$

After these operations are performed, three situations may arise:

- The result is already normalized and, thus, no action is required.
- When the effective operation (EOP) is an addition, an over flow of the result may occur. If the result overflows, the mantissa is right-shifted by 1 and the exponent is decremented by 1.
- When the effective operation is subtraction, there might be a variable amount of leading zeros (lz) in the result. In this case, the mantissa is left-shifted and the exponent is subtracted by the number of leading zeroes present.

Radix-point alignment of the mantissas before addition/subtraction lies along the critical path of the adder. The delay was reduced by splitting the required shifting operation in to two halves and distributing it between two pipeline stages as shown in Figure 4. The 'leading one' detection operation was also divided into two pipeline stages to reduce delay along the critical path. It is implemented using a 24-bit priority look-ahead encoder.

2.3 Floating-point Division

In this paper, we have implemented non-restoring division algorithm[5]. The steps involved areas follows-

- For given two input operands, compute the sign of the division by XOR operation of the sign bits
- Compute exponent by subtracting the exponents of the input operands and adding a bias of 127.
- The mantissa division is performed using a pipelined non-restoring division algorithm.
- The result is normalized and rounded.
- Perform post output exception handling.

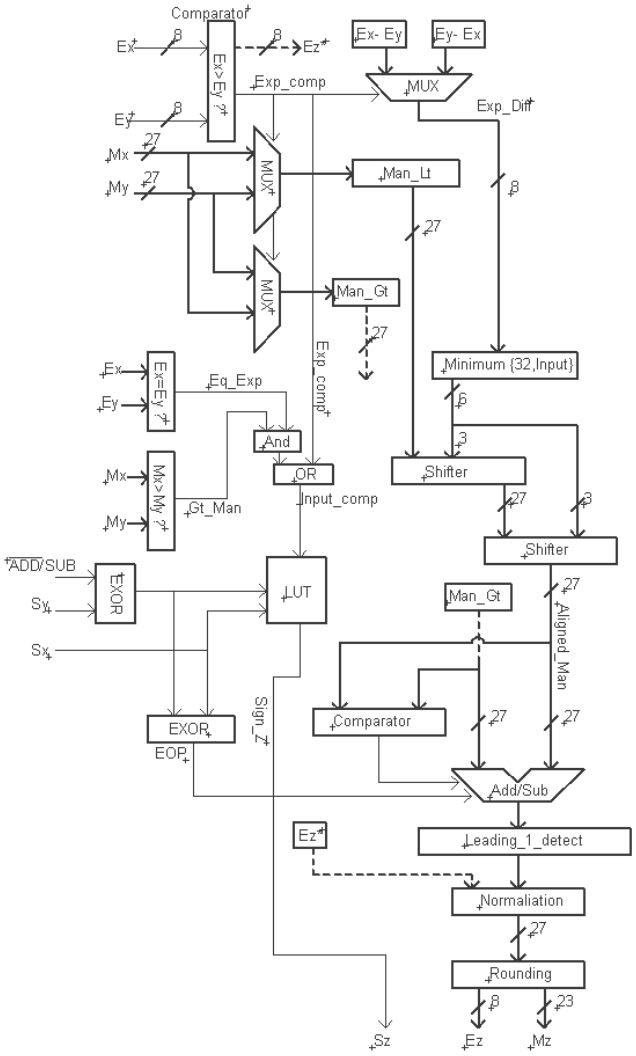

Figure 4. Adder/Subtractor Pipeline

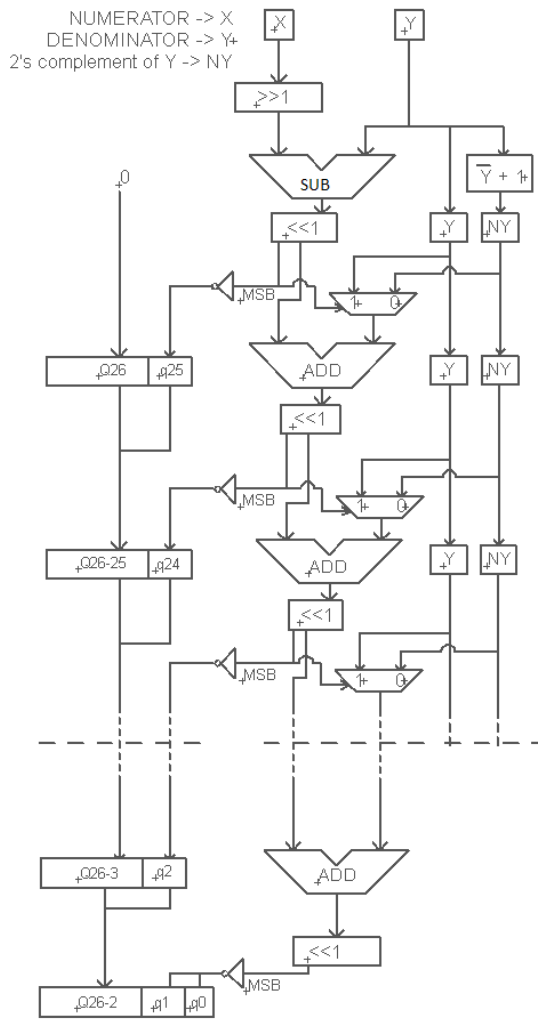

Figure 5. Mantissa Division Pipeline.

The non-restoring division algorithm gives a linear convergence. The final result is obtained after 29 clock cycles. The minimum clock period of the pipeline was improved by preserving both the value of the divisors mantissa as well as its 2's complement in dedicated registers for the ADD/SUB block. A similar technique was also applied in the square-root sub-module. The complete division algorithm that has been implemented is shown in Figure 5.

Mantissa division was initially attempted using multiplicative algorithms based on Taylor-series expansion[10,11], but they turned out to be too in accurateue to an accumulation of truncation errors during partial product calculation. Truncation was necessary to limit the use of DSP multipliers in the implementation but the resulting mantissa has an error in the order of 10^{-5}. This magnitude of error is unacceptable, especially for large exponent values.

2.4 Floating-point Square Root

Calculating square root in FPU is far more complex than addition and multiplication operation. We have implemented a model of non-restoring square root calculation algorithm[4]. The calculation of exponent in square root is done by following steps:

- Subtract a bias of +127. For an even exponent, right-shift the exponent by 1-bit. For an odd exponent, subtract 1 from the exponent and then right-shift. Also, left-shift the mantissa by 1-bit for an odd exponent.
- In the resultant exponent, add a bias of 127 for positive input exponent, whereas for negative input exponents subtract the resultant exponent from 126.

The non-restoring square root algorithm uses two input bits for generating one output bit, so the mantissa is made 48-bit wide by appending 24 zeros. The input mantissa is further extended to 54-bits to produce a 27-bit result. The last 3 bits of the result are used for rounding the result to 24-bits. This has been depicted in Figure 6.

2.5 Rounding

The IEEE-754 standard[1] defines four types of rounding:

- Round towards nearest even.
- Round towards positive infinity.
- Rounding towards negative infinity.
- Rounding towards zero.

All four round-modes are supported by this FPU. The FPU architecture includes a 2-bit input port used to specify the rounding-mode for input set. If the round-mode is fixed for any application, this port can be tied to a constant value prior to synthesis to save on resources and enhance logic optimisation.

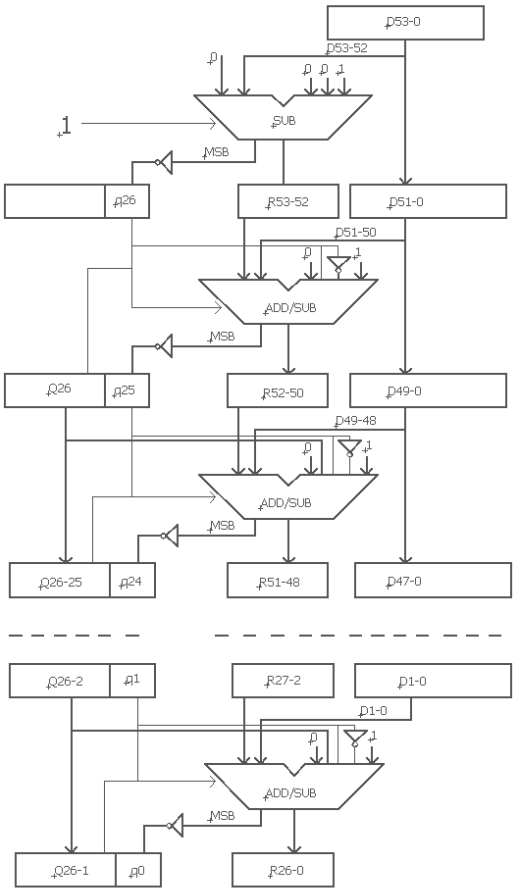

Figure 6. Mantissa Square-Root Pipeline.

Of the four supported modes, round towards nearest even is recommended. It has the advantage of being an unbiased-rounding modes in ceith and less rounding of positive and negative numbers symmetrically.

2.6 Exception Handling

Exceptions can be categorized into two types:

- Initialisation-time exceptions
- Run-time exceptions

There are five types of exceptions that need to be handled:

- Invalid operations
- Overflow
- Underflow
- Inexact

Invalid operation detection such as the divide-by-zero exception can be handled at the initialisation-time, and hence, the exceptions are flagged immediately and an appropriate output is generated. However, overflow, underflow and in exact are run time exceptions and are handled in the respective modules. Operations on infinity, zero and NaN can cause exceptions depending on the operation. Correct outputs are generated when one or both of the input operands are Infinity, zero or NaN. Exception flags are raised, and aqNaN is generated at the output only for the following operations:

- $+\infty - \infty$ (OR) $-\infty + \infty$
- $\infty \times 0$
- Division by zero
- $(\pm\infty) \div (\pm\infty)$
- Square root of negative numbers

2.7 FPU Architecture

The architecture implemented in the design is as shown in Figure 7. The computing process is distributed in four phases:

- Inputs are checked for initialisation-time exceptions.
- Incase an invalid operation is detected, qNaN is produced at E_ Z along with respective exception flags and op_ID.
- If no exception is detected, inputs are processed by the individual sub-module depending on the op-code.
- 32-bit result is produced by the sub-module along with the op_ID and any run-time exception flags that were raised.

3. Results

Functional (behavioral) simulation was performed using Model Sim software as shown in Figures 8–11. Each sub-module was thoroughly test educing comprehensive test-benches. Test vectors used for testing and verification were randomly generated using MATLAB software. Each test-bench was fed with the correct (expected) hex-value of the output. Each output was then checked against the expected result and, should they match, the respective wire of the correct signal in the test-bench is SET.

Place-And-Route(PAR) was done, and the generated PAR reports highlight the behaviour of the design and consumption of resources. The PAR reports were studied and optimizations were performed by applying relevant Xilinx XST attributes[11] to nets on the critical path. Results from the PAR report and the Post-PAR static timing report are given in Table 1.

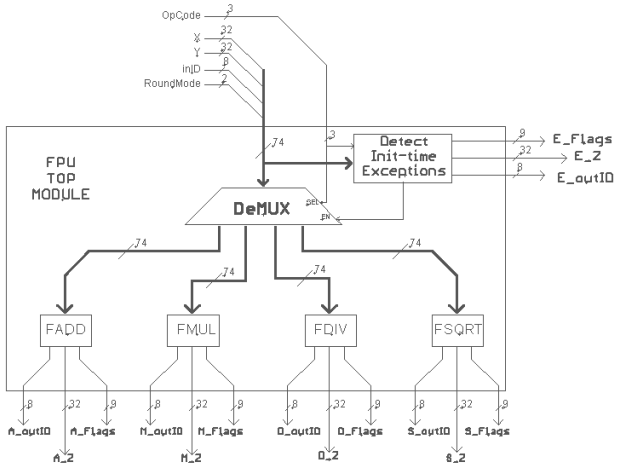

Figure 7. FPU Architecture Overview

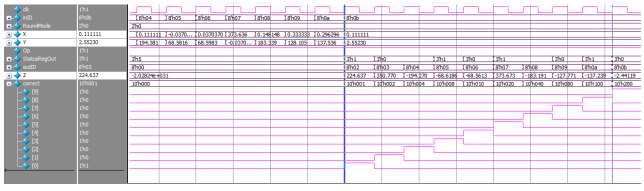

Figure 8. FADD Functional Simulation

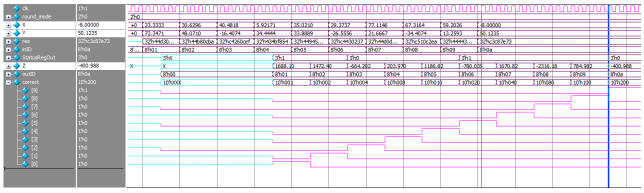

Figure 9. FMUL Functional Simulation

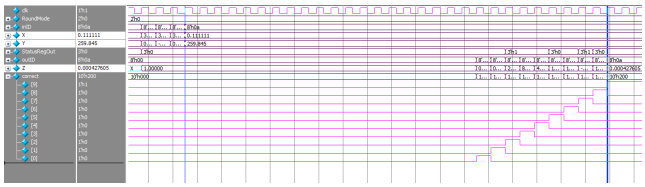

Figure 10. FDIV Functional Simulation

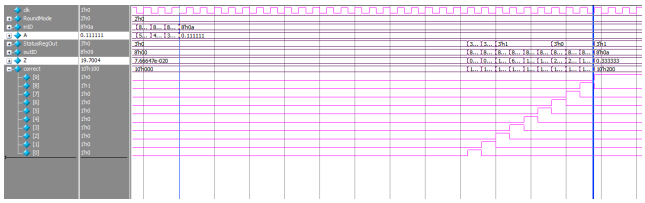

Figure 11. FSQRT Functional Simulation

Table 1. Timing and resource utilization summary

Module	Min Period (ns)	Max clock Frequency (Mhz)	Slice Register Used	Slice LUT's Used	Latency Cycles	Throughput (outputs per cycle)
FADD	3.115	321	6%	9%	9	1
FMUL	3.203	312	4%	6%	17	0.2
FDIV	3.404	293	18%	23%	29	1
FSQRT	3.613	276	13%	26%	29	1
TOP MODULE	4.738	211	52%	74%	(varies)	(varies)

4. Conclusions and Future Work

The primary objective was to develop an FPU that could be used as a sub-module for other FPGA based projects, while trying to achieve maximum speed. The design has been completed; however, there is still scope for development in the future. Further improvements in the implementation of mantissa calculation for FDIV and FSQRT sub-modules can be made[13,14]. Also, a different strategy for place and route of the top module can be attempted to get better timin results. More developments that can be added are listed below:

- Adding a floating point to integer and integer to floating point converter.
- Inclusion of floating-point comparison operations.
- Support for signaling NaN(sNaN).
- Combining the FDIV and FSQRT in to one sub-module due to their functional similarities.
- Adding support for subnormal numbers.
- Including an output decoder.

5. References

1. IEEE Standard for Floating-Point arithmetic. IEEE Xplore Digital Library; 2008 Aug 29. p.1–70.
2. Patil V, Raveendran A, Sobha PM, Selvakumar AD, Vivian D. Out of order floating point coprocessor for RISCVISA. 19th International Symposium on VLSI Design and Test(VDAT), Ahmedabad: India; 2015. p .1–7.
3. Xilinx Spartan-6DSP48A1SliceUserGuide [Internet]. [Cited 2016 Feb]. Available from: www.xilinx.com.
4. Li Y, Chu W. Anewnon-restoring square root algorithmandits VLSI implementations. International Conferenc eonComputerDesign(ICCD96), Austin, Texas: USA; 1996 Oct.p.538–544.
5. Koren I. Computer arithmetic algorithms, 2nd edn. A K Peters Ltd.; 2001.
6. Nesam JJJ, Sathasivam S. An efficient single precision floating point multiplier architecture based on classical recoding algorithm. Indian Journal of Science and Technolog. 2016 Feb; 9(5):1–7.DOI: 10.17485/ijst/2016/v9i5/87159.
7. Shengalet PA, Dahake V, Mahendra M. Single precision floating point ALU. International Research Journal of Engineering and Technology. 2015 May; 2(2):1–4.
8. XilinxSpartan-6FamilyOverview [Internet]. [Cited 2016 Feb]. Available from: www.xilinx.com.
9. Gollamudi PS, Kamaraju M. Design of high performance IEEE- 754 single precision (32 bit) floating point adder using VHDL. International Journal of Engineering Research and Technology. 2013 Jul; 2(7).
10. Liddicoat AA, Flynn MJ. High-performance floating-point divide. Proceedings of Euromicro Symposiumon Digital System Design, Warsaw; 2001 Sep. p.354–61.
11. Kwon TJ, Sondeen J, Draper J. Floating-point division and square root implementation using a Taylor-series expansion algorithm. 15th IEEE International Conference on Electronics, Circuits and Systems, ICECS; 2008. p.702–5.
12. Xilinx XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices [Internet]. [Cited 2016 Feb]. Available from: www.xilinx.com.s
13. Hassan SK, Monica PR. Floating point high performance low area SFU. Indian Journal of Science and Technology. 2015 Aug; 8(20):1–7.DOI: 10.17485/ijst/2015/v8i20/78367.
14. Ragunath G, Sakthivel R. Low - power and area - efficient square - root carry select adders using Modified XOR Gate. Indian Journal of Science and Technology. 2016 Feb; 9(5):1–8.DOI: 10.17485/ijst/2016/v9i5/87181.