International Journal Of Engineering Research & Management Technology ISSN: 2348-4039

IJERMT Email: editor@ijermt.org March 2014 Volume-1, Website: ijermt.org

Implementation Of High Speed IEEE 754Compliant Double & Quadruple Floating Point

Prashant Mani
Assistant Professor
Deptt. Electronics & Communication Engineering
SRM University, NCR Campus

Konakalla Aditya Ram
M.Tech Student
Deptt. Electronics & Communication Engineering
SRM University, NCR Campus

Abstract
According to scientific applications, level of precision is more demanding computational point which can be double precision floating point arithmetic's or quadruple precision floating-point arithmetic's. Here we analyze the evolution of double-precision floating-point& quadruple precision floating-point computing. Since last few years this application has more demand. Modern science and Engineering models mostly depend on supercomputer simulation to reduce experimentation requirements. The results show that peak-performance for precision addition, Subtraction, multiplication and division on FPGAs is already better than general-purpose processors (GPPs).The canonical signed digit (CSD) representation is one of the existing signed digit representations with unique features which make it useful in certain DSP applications focusing on low power, area efficient and high speed arithmetic. Canonical signed digit is a recoding technique, which recodes a number with minimum number of non-zero digits. As the number of partial products depends on the number of non-zero digits, by using Canonical recoding, the number of non-zero digits will be reduced, thereby reducing the number of partial products. In this paper, Double & quadruple precision floating point Addition, Subtraction, multiplication& Division using canonical signed digit is proposed and is compared with Conventional multiplication technique. The design is implemented in Verilog and simulated using Xilinx 9.2 ISE.

Keywords
Canonical signed digit, Double precision floating point number, quadruple floating IEEE, Verilog

Introduction

Soon after the introduction of the FPGA in the mid-1980's an interest developed in using the devices for DSPs and Digital communication applications, especially for digital filtering which can take advantage of specialized constants embedded in hardware. Since a large portion of most filtering approaches involves the use of multiplication, efficient adder and multiplier implementations in both fixed- and floating-point were of particular interest. Many early FPGA multiplier implementations used circuit structures adapted from the early days of LSI development and reflected the restricted circuit area available in initial FPGA devices. As FPGA capacities have increased, the diversity of arithmetic circuit's implementations has grown.

The design of embedded systems, that is, circuits designed for specific applications, is based on a series of decisions as well as on the use of several types of development techniques. For example:

1. Selection of the data representation
2. Generation or selection of algorithms
3. Selection of hardware platforms
4. Hardware–software partitioning

5. Program generation
6. New hardware synthesis
7. Co-simulation, co-emulation, and prototyping

Some of these activities have a close relationship with the study of arithmetic algorithms and circuits, especially in the case of systems including a great amount of data processing (e.g., ciphering and deciphering, image processing, digital signature, biometry).

	Sign	Exponent	Fraction	Bias
Double Precision	1[63]	11[62-52]	52[51-00]	1023
Quadruple Precision	1[128]	15[127-112]	112[111-00]	262143

Table I. Double & Quadruple Floating Point Sign, Exponent, Fraction & Bias bits

1. Architecture of Floating Point

A floating-point unit (FPU, colloquially a math coprocessor) is a part of a computer system specially designed to carry out operations on floating point numbers. Typical operations are addition, subtraction, multiplication & division

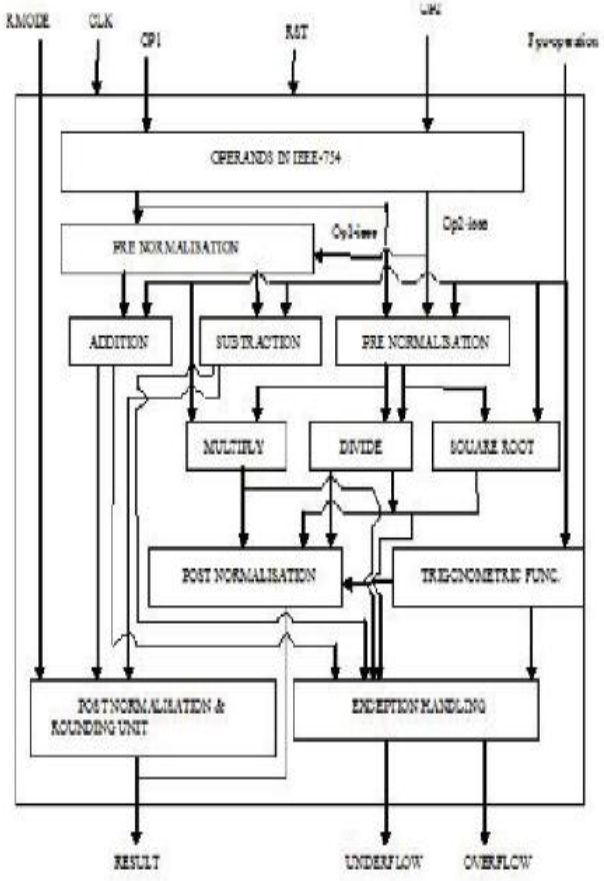


Fig 1.1 Floating Point Architecture

When a CPU is executing a program that calls for a floating-point operation, there are three ways to carry it out:

- A floating-point unit emulator (a floating-point library)
- Add-on FPU
- Integrated FPU

Some systems implemented floating point via a co-processor rather than as an integrated unit. This could be a single integrated circuit, an entire circuit board or a cabinet. Where floating-point calculation hardware has not been provided, floating point calculations are done in software, which takes more processor time but which avoids the cost of the extra hardware. For particular computer architecture, the floating point unit instructions may be emulated by a library of software functions; this may permit the same object code to run on systems with or without floating point hardware. Emulation can be implemented on any of several levels: in the CPU as microcode (not a common practice), as an operating system function, or in user space code. When only integer functionality is available the CORDIC floating point emulation methods are most commonly used.

A. A floating-point unit emulator

Some floating-point hardware only supports the simplest operations - addition, subtraction, and multiplication. But even the most complex floating-point hardware has a finite number of operations it can support - for example, none of them directly support arbitrary- precision arithmetic.

When a CPU is executing a program that calls for a floating-point operation not directly supported by the hardware, the CPU uses a series of simpler floating-point operations. In systems without any floating-point hardware, the CPU emulates it using a series of simpler fixed-point arithmetic operations that run on the integer unit. The software that lists the necessary series of operations to emulate floating-point operations is often packaged in a floating-point library.

B. Add-on FPU

In the 1980s, it was common in IBM PC/compatible microcomputers for the FPU to be entirely separate from the CPU, and typically sold as an optional add-on. It would only be purchased if needed to speed up or enable math-intensive programs.
The IBM PC, XT, and most compatibles based on the 8088 or 8086 had a socket for the optional 8087 coprocessor. The AT and 80286-based systems were generally socketed for the 80287, and 80386/80386SX based machines for the 80387 and 80387SX respectively, although early ones were socketed for the 80287, since the 80387 did not exist yet. Other companies manufactured co-processors for the Intel x86 series. These included Cyrix and Weitek.

C. Integrated FPU

In some cases, FPUs may be specialized, and divided between simpler floating-point operations (mainly addition and multiplication) and more complicated operations, like division. In some cases, only the simple operations may be implemented in hardware or microcode, while the more complex operations are implemented as software.

In some current architecture, the FPU functionality is combined with units to perform SIMD computation; an example of this is the replacement of the x87 instructions set with SSE instruction set in the x86-64 architecture used in newer Intel and AMD processors.

A. Significant

The significant or coefficient or mantissa is the part of floating that contains its significant digits. Depending on the interpretation of the exponent, the significant may be considered to be an integer or a fraction. For example, the number 123.45 can be represented as a decimal floating-point number with integer significant 12345 and exponent −2. Its value is given by the arithmetic:

12345×10^{-2}

This same value could also be represented in normalized form with the fractional (non-integer) coefficient 1.2345 and exponent +2:

$1.2345 \quad 0^{+2}$

B. Exponentiation

Exponentiations a mathematical operation, written as an, involving two numbers, the based and the exponent (or power) n. When n is a positive integer, exponentiation corresponds to repeated multiplication; in other words, a product of n factors of a(the product itself can also be called power)

$$a^n = \underbrace{a \times \cdots \times a}_{n},$$

Just as multiplication by a positive integer corresponds to repeated addition

$$a \times n = \underbrace{a + \cdots + a}_{n}.$$

The exponent is usually shown as a super script to the right of the base. The exponentiation can be read as raised to tenth power, raised to the power [of]n, or possibly raised to the exponent [of]n, or more briefly as a top then. Some exponents have their own pronunciation: for example,$a^2$is usually read asasquaredanda^{3}asacubed. When superscripts cannot be used, as in plain ASCII text, common alternative formats include a^n and a**n.

C. Special Cases

A summary of special cases is shown in Table 1.1. These are useful in representing exceptional cases like adding ∞ to ∞, multiplying ∞ to ∞ etc.,

i. De-normalized Numbers

A de-normalized number is any nonzero number with an exponent field of 0. The exponent offset is 1023, but for the case when the value in the exponent field is 0, then the offset is changed to 1022. The value multiplied by the mantissa is 2^ (-1022). The exponent offset is changed to 1022 because for de-normalized numbers, the implied leading '1' is no longer included. So to calculate the actual value of a de-normalized number, you multiply 2^ (-1022) by the mantissa without the leading '1'. The range of values that can be represented by a de-normalized number is approximately 2.225e-308 to 4.94e-324.
For example, the number 2e-309 is represented in the double precision floating point

ii. Infinity

The behavior of infinity in floating-point arithmetic is derived from the limiting cases of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is: $-\infty <$ {every finite number} $< +\infty$. Operations

on infinite operands are usually exact and therefore signal no exceptions, including, among others,

- Addition (∞, x), addition(x, ∞), subtraction (∞, x), or subtraction (x, ∞), for finite x
- Multiplication (∞, x) or multiplication(x, ∞) for finite or infinite $x \neq 0$
- Division (∞, x) or division(x, ∞) for finite x

iii. NaN

A NaN is (not a number) — a symbolic floating-point datum. There are two kinds of NaN representations: quiet and signaling. Most operations propagate quiet NaNs without signaling exceptions, and signal the invalid operation exception when given a signaling NaN operand. Quiet NaNs are used to propagate errors resulting from invalid operations or values, whereas signaling NaNs can support advanced features such as mixing numerical and symbolic computation or other extensions to basic floating-point arithmetic. For example, 0/0 is undefined as a real number, and so represented by NaN; the square root of a negative number is imaginary, and thus not represent able as a real floating-point number, and so is represented by NaN; and NaNs may be used to represent missing values in computations.

I. Double Precision Floating Point

Double precision is a computer numbering format that occupies two adjacent storage locations in computer memory. A double precision number, sometimes simply called a double, may be defined to be an integer, fixed point, or floating point. Modern computers with 32-bit storage locations use two memory locations to store a 64-
bit double precision number (a single storage location can hold a single precision number).

1 sign bit	11 bits-exponent	53-bits significant

Fig.1.3 Double precision floating point format

Double precision floating point is an IEEE-754 standard for encoding binary or decimal floating point numbers in 64 bits. Double precision binary floating-point is a commonly used format on PCs, due to its wider range over single precision floating point, even if it's at a performance and bandwidth cost. As with single precision floating point format, it lacks precision on integer numbers when compared with an integer format of the same size. It is commonly known simply as double. The IEEE 754 standard defines a double as

- Sign bit: 1 bit
- Exponent width: 11 bits
- Significant precision: 53 bits (52 explicitly stored)

The format is written with the significant having an implicit integer bit of value 1, unless the written exponent is all zeros. With the 52 bits of the fraction significant appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, $\log_{10} 2^{53} \approx = 15.95$. The bits are laid out as shown in fig 1.3

The real value assumed by a given 64 bit double precision data with a given biased exponent and a52 bit fraction can be more precisely,

$$value = (-1)^{sign}\left(1 + \sum_{i=1}^{52}\lceil b_{-i}\rceil 2^{-i}\right) \times 2^{(e-1023)}$$

II. Quadruple Precision Floating Point

In computing, quadruple precision (also commonly shortened to quad precision) is a binary floating-point-based computer number format that occupies 16 bytes (128 bits) in computer memory and whose precision is about twice the 53-bit double precision.

This 128 bit quadruple precision is designed not only for applications requiring results in higher than double precision,[1] but also, as a primary function, to allow the computation of double precision results more reliably and accurately by minimizing overflow and round-off errors in intermediate calculations and scratch variables: as William Kahan, primary architect of the original IEEE-754 floating point standard noted, "For now the 10-byte Extended format is a tolerable compromise between the value of extra-precise arithmetic and the price of implementing it to run fast; very soon two more bytes of precision will become tolerable, and ultimately a 16-byte format... That kind of gradual evolution towards wider precision was already in view when IEEE standard 754 for Floating-Point Arithmetic was framed."This gives from 33 - 36 significant decimal digits precision (if a decimal string with at most 33 significant decimal is converted to IEEE 754 quadruple precision and then converted back to the same number of significant decimal, then the final string should match the original; and if an IEEE 754 quadruple precision is converted to a decimal string with at least 36 significant decimal and then converted back to quadruple, then the final number must match the original).

The format is written with an implicit lead bit with value 1 unless the exponent is stored with all zeros. Thus only 112 bits of the significant appear in the memory format, but the total precision is 113 bits (approximately 34 decimal digits, $\log_{10}(2^{113}) \approx 34.016$). The bits are laid out as follows:

1 sign bit	15 bits-exponent	112 bits significant

- Sign bit: 1 bit
- Exponent width: 15 bits
- Significant precision: 112 bits

III. Implementation & Simulation of Floating Point

A. Addition

In this Float64 Add is a block name that is provided for free in the form of Verilogcode. The code is difficult to read because of removed text formatting and identifiers replaced with automatically generated strings. The fully functional code that should work correctly in any simulation and synthesis tool can be designed

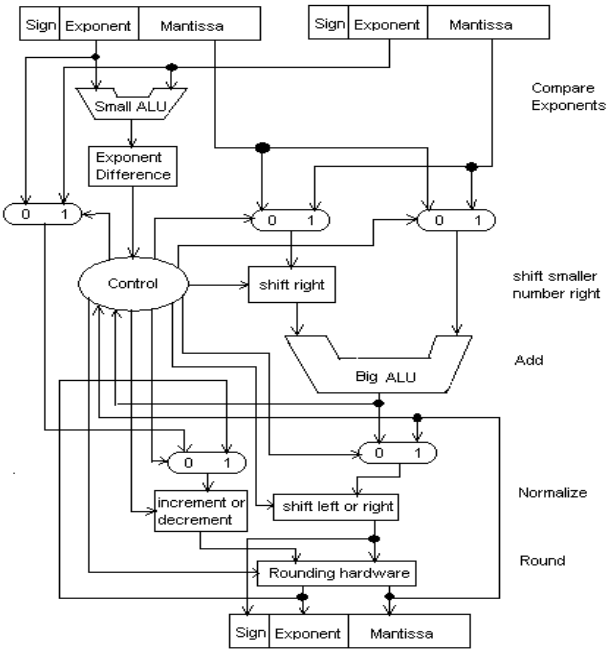


Fig 1.2 Floating Point Addition

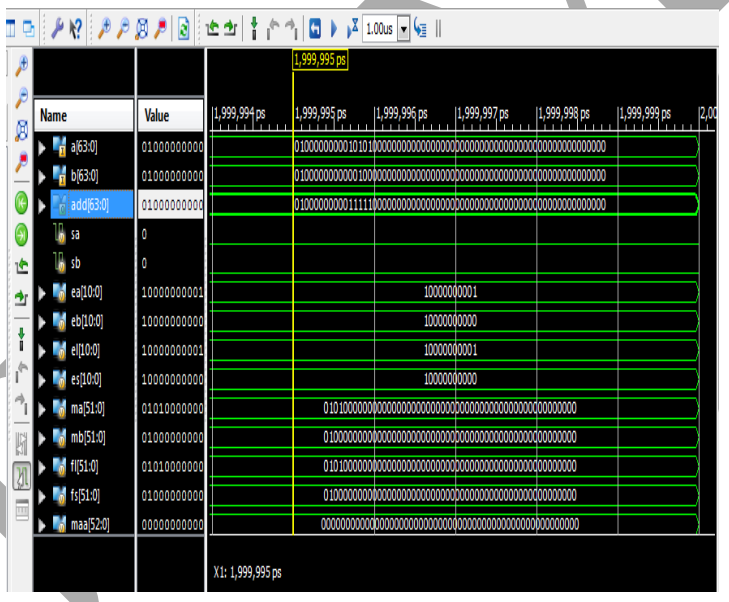


Fig 1.3 Floating Point Addition Simulation

B. Subtraction

In this Float64 Sub is a block name that is provided for free in the form of Verilogcode. The code is difficult to read because of removed text formatting and identifiers replaced with automatically generated strings.

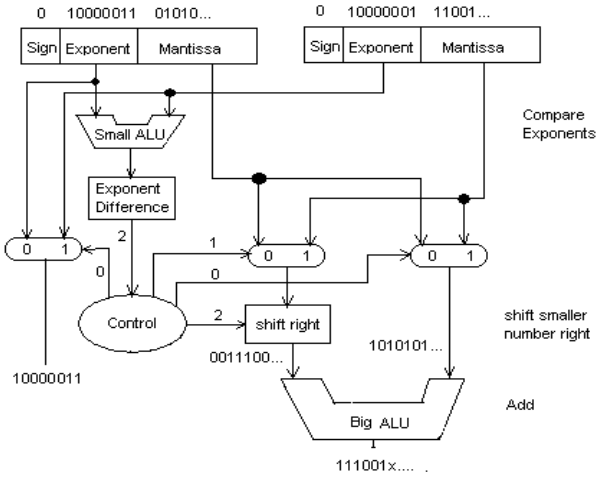


Fig 1.4 Floating Point Subtraction

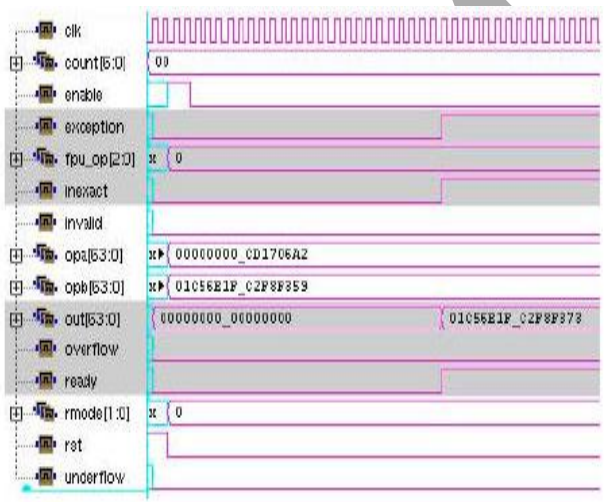
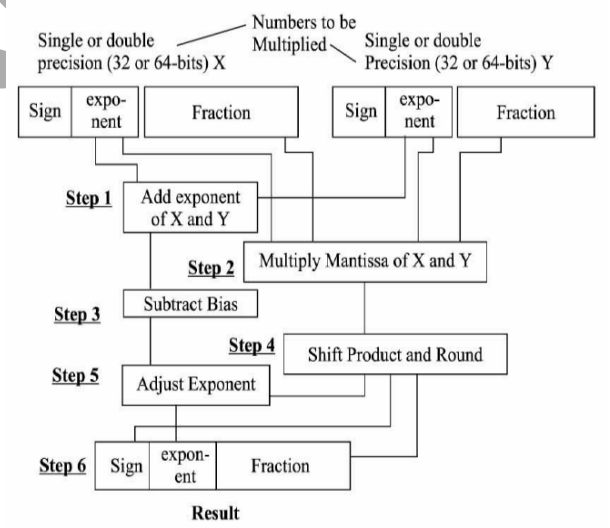


Fig 1.5 Floating Point Subtraction Simulation

C. Multiplication

Fig 1.6 Floating Point Multiplication



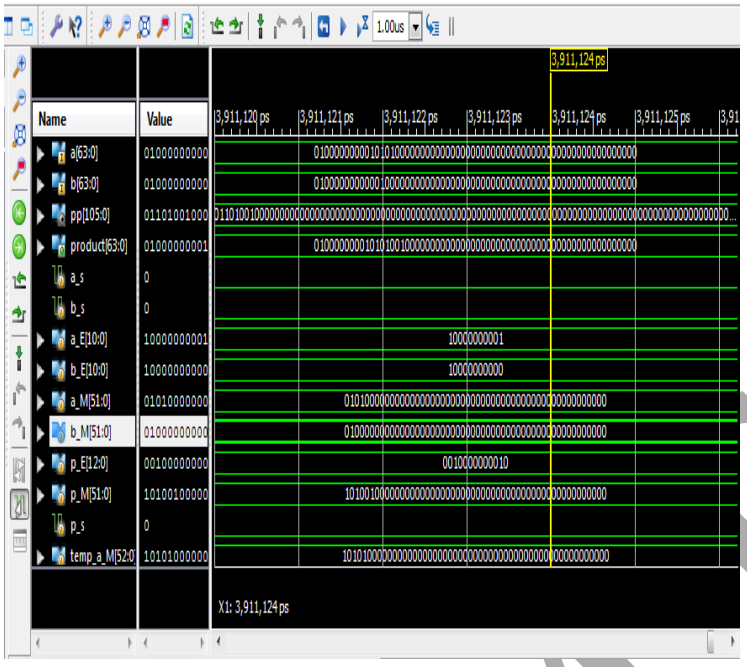


Fig 1.7 Floating Point Multiplication Simulation

D. Division:

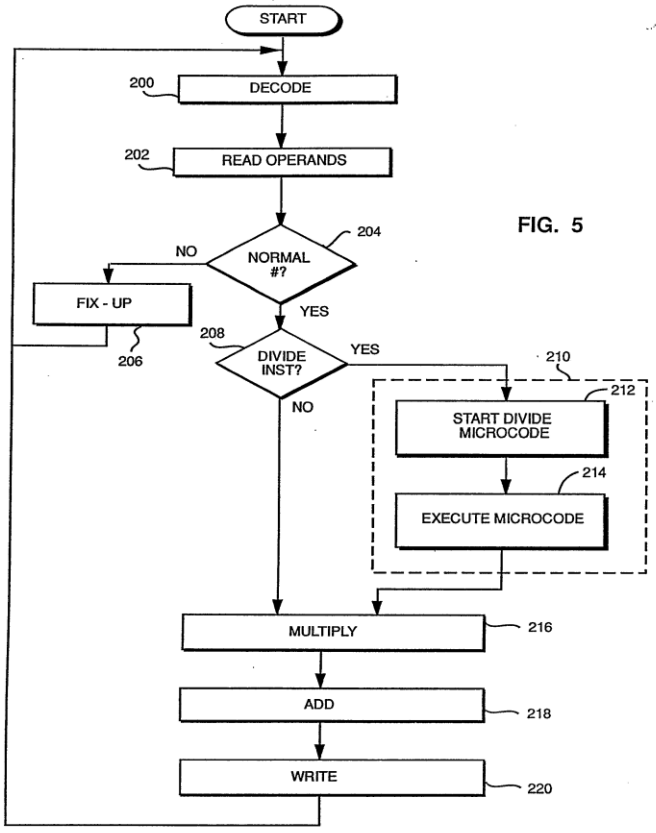


FIG. 5

Fig 1.8 Floating Point Division

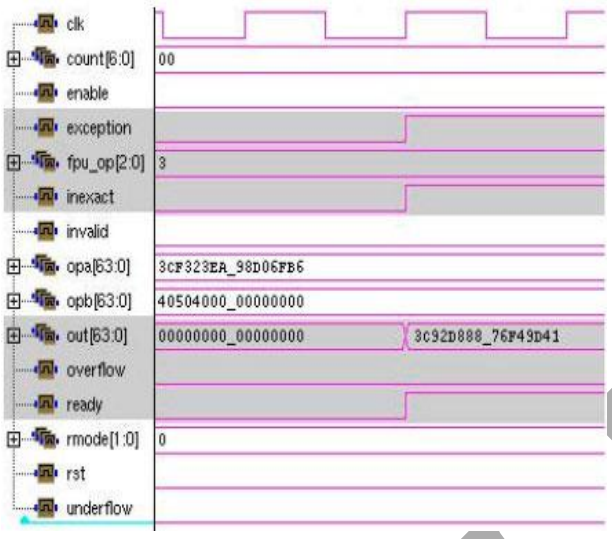


Fig 1.9 Floating Point Division Simulation

References

1. Geetanjali Wasson, "IEEE-754 compliant algorithms for fast multiplication of double precision floating point numbers" International Journal of Research in Computer Science, eISSN 2249-8265 Volume1 Issue 1(2011) pp.1-7
2. B. C. Jinaga, Saroja. V Siddamal, R. M. Bankar, "Design of high-speed floating point multiplier", 4th IEEE International symposium on Electronic Design, Test & Applications, IEEE computer society,2008, pp.285-289.
3. Yunhua Wang, Linda S. DeBrunner, Dayong Zhou, Victor E. DeBrunner, "A novel multiplier less hardware implementation method for adaptive filter coefficients, ICASSP 2007.
4. S. Jagadeesh, S. Venkata chary " Design of parallel multiplier- accumulator based on Radix-4 modified Booth algorithm with SPST" , International Journal of Engineering research and applications, volume 2, issue 5,September-october 2012, pp.425-431.
5. Yunhua Wang, "Iterative radix-8 multiplier structure based on a novel real-time CSD recoding" ACSSC 2007, conference record of 41st Asilomar conference on signals, systems& computers, 2007, pp. 977-981.
6. Sang-min Kim, "Low error fixed-width CSD multiplier with efficient sign extension", IEEE transactions on circuits and systems-II analog and digital signal processing, volume 50, December 2003.
7. M. Saad, M. Taher, "High speed area efficient FPGA based floating point arithmetic modules", National conference on radio science, March 2007, pp. 1-8.
8. Mrs. Pushpawathi changlekar, Mrs. Sujatha, "Implementation of Binary Canonic Signed Digit multiplier using Application specific IC", International journal of engineering research and applications, volume 3, Issue 1, Jan- Feb 2013, pp.1912-1915.
9. Cetin K. KOC, Chin -Yu Hung, "Adaptive m-ary segmentation and Canonical recoding algorithms for multiplication of large binary numbers", Computers Math. Applications. Vol 24, No.3, pp. 3-12, 1992.
10. Kavita, Jasbir Kaur, "Design and implementation of an efficient modified booth multiplier using VHDL", proceedings of 2nd international conference on emerging trends in engineering and management, July 2013.