DOI: 30.1450/ijearst.44.59.6568 https://ijearst.co.in/

ENGINEERING IN ADVANCED RESEARCH SCIENCE AND TECHNOLOGY

ISSN 2386-7876
Vol.01, Issue.03
March-2026
Pages: 332-339

FAST MODULAR MULTIPLIERS FOR SUPERSINGULAR ISOGENY-BASED POST-QUANTUM CRYPTOGRAPHY

[1] **M Madhavi**, [2] **Kothapalli Nagendrachari**, [3] **Bonthu Ayyappa**, [4] **Jeeri Aditya**, [5] **B Sudhakaru**, [6] **Bayya Mahesh**

[1]Guide, Dept of ECE, ABR College of Engineering and Technology, Kanigiri, A.P.
[2,3,4,5,6]B. Tech, Dept of ECE, ABR College of Engineering and Technology, Kanigiri, A.P.

ABSTRACT: The VLSI implementation of the high-speed modular multiplier remains a big challenge. In this paper, we propose efficient which will be useful for both signed and unsigned modular multiplication algorithms based on an unconventional radix for which cost about 20% fewer computations than the prior art. Besides, a multi precision scheme is also introduced for the proposed algorithms to improve the scalability in hardware implementation, resulting in new algorithms.

Keywords: Post-Quantum Cryptography (PQC), Super singular Isogeny Cryptography (SIKE), Modular Multiplication, Fast Arithmetic Units, Finite Field Arithmetic, Montgomery Multiplication, Barrett Reduction, High-Speed Cryptographic Hardware, Low-Latency Arithmetic Circuits, VLSI Architecture, FPGA Implementation, ASIC Design, Parallel Processing.

INTRODUCTION: THE computational performance of a quantum computer is expected to be much higher than that of a traditional computer. In 1994, Shor [1] proposed a fast decomposition method for large numbers based on a quantum computer, whose complexity is O(logN), and the larger the number, the better the performance of Shor's algorithm. Later in 1996, Grover [2] proposed a quantum algorithm which can be used to search an unsorted database faster than a traditional computer (quadratic speedup O(N/2) time rather than O(N)). In recent years much effort has gone into the development of quantum computers by industry, and in 2019, Google developed a new 54-qubit processor, which is 9 orders of magnitude faster than the fastest supercomputer [3]. Thus, it is predictable that quantum computers will become practical in the near future. Traditional encryption methods, such as RSA [4] based on the large number factorization problem, and elliptic curve cryptography (ECC) [5], based on the discrete logarithm problem, will be easily broken by quantum computers, which will have an important impact on current communications and network security Post-Quantum Cryptography (PQC) [6] is a form of publickey cryptography with high complexity, and includes encryption, digital signatures and key encapsulation mechanisms. PQC contains a variety of algorithms based on different hard problems [7], such as lattice based-cryptography, multivariate quadratic cryptography, hash-based signatures and code-based cryptography. The supersingular isogeny Diffie-Hellman (SIDH) [8] key exchange protocol is the most recently proposed PQC scheme. SIDH is based on the theory of point addition and point doubling in ECC, combined with the theory of isogenies in elliptic curves. Two curves E/K and E0/K are isogenous over K if there exists a morphism $\varphi : E \rightarrow E0$ with coefficients in K mapping the neutral element of E to the neutral element of E0 . Therefore, SIDH is more complex than ECC, and is believed to be resistant to attacks by quantum computers [9]. In addition, SIDH is characterized by relatively small key sizes compared with other PQC schemes. SIDH is the basis of the Supersingular Isogeny Key Encapsulation (SIKE) protocol [10], which is a candidate in the NIST PQC standardization process. However, as SIDH was proposed more than a decade after other PQC cryptosystems, there is still much work needed to understand its practical capability. **R**ECENT improvements in quantum system control make it seem feasible to finally build a powerful quantum computer in the near future [1]. Many companies, such as IBM, Google, and Intel, have enthusiastically joined this field. A company named IonQ reported in December 2018 that its could built large as 160 qubits [2]. These achievements have brought to the flurry of research in publickey cryptography since most of the popular public-key ciphers, such as the RSA [3] and ECC [4] schemes based on the difficulty of factoring integers or the discrete logarithm problem, can be solved by Shor's algorithm [5] with quantum computers. Meanwhile, they have

also accelerated the development of post-quantum cryptography (PQC) protocols. For example, the call for proposals for PQC standards hosted by the National Institute of Standards and Technology (NIST) [6] is driven by this demand. The supersingular isogeny key encapsulation (SIKE) protocol [7] has advanced to the third round of the PQC standardization process in July 2020 after submitted to the NIST in November 2017. The possible reason is that it is the only one that is similar to the classical ECC having very small public and secret keys and owning perfect forward secrecy. Recently, it has been proven in [8] and [9] that the security estimations in the SIKE proposal were extremely conservative both in quantum and classical situations. In other words, smaller key sizes can be used for the security levels presented in [7]. THE public key cryptography such as RSA and Elliptic Curve Cryptography (ECC) is widely used in current information systems for security. The modular multiplication of large integer is the crucial operation in these cryptographic schemes. For RSA, the recommended key sizes are larger than 2048 bits. Taking the long view, 2048-bit or even larger integer modular multiplications need to be performed in RSA cryptosystem. Montgomery Modular Multiplication (MMM) algorithm [1] is the most important algorithm to perform modular multiplication. So, the hardware implementation of MMM has received extensive attention. To date, the implementations of Montgomery multiplier fall into two categories: one includes designs focusing on high-performance [2], [3], in which high-performance and low latency multipliers are essential, and the other includes designs of small area [4], [5], in which various iterative structure is widely used. For high-performance designs, full-word MMM is a common option. Designing high-performance multipliers is critical to full-word MMM. Researchers have proposed various high-performance multipliers by improving the algorithms and circuit structures. Schoolbook multiplication, Karatsuba multiplier [2], and Toom–Cook multiplier [3] are most representative in the existing literature. Meanwhile, several potential techniques, such as FFT-based MMM[6], Residue Number System (RNS) [7], pipeline structure[2] and so on, are introduced into MMM for improving performance. However, it will consume tremendous hardware resources to implement MMM, if the sizes of the operands become very large. Another drawback of full-word MMM is that the pre-calculation of the multiplicative inverse is inevitable [1]. The small area designs usually adopt the Radix-2k MMM algorithm with the advantage of being flexible and scalable. Moreover, pre-calculation is unnecessary in the Radix-2k MMM algorithm when k is small. The Montgomery multiplier can achieve a good balance between performance and area by changing the value of k. So, the Radix-2k MMM algorithm is adopted In this brief. Researchers have proposed some modifications and implementations for the Radix-2k MMM algorithm. These improved schemes [8], [9] are majorly developed base on carry-save adder (CSA) for optimizing critical path delay. Additionally, Booth encoding is also introduced into MMM to reduce the complexity and latency of partial products generating circuit[10].

LITERATURE SURVEY: In 2011, Jao and Feo presented software implementation results showing that their proposed SIDH key-exchange protocols are over two orders of magnitude faster than classical isogeny-based cryptosystems over ordinary curves. Later Azarderakhs et al. implemented the same SIDH protocol on PC (x86-64) and ARM (ARMv7) platforms [19]. Their implementation is between 18-26% faster depending on the security level. In 2016, Costello et al. presented a high-speed implementation of SIDH, which is more than 2.5 times faster than the previous SIDH software results [20]. The first hardware implementation of SIDH was recently proposed, targeting a Virtex-7 Field Programmable Gate Array (FPGA) [21], and is 1.5 times faster than the best software implementation for the 512-bit SIDH scheme. Arithmetic unit of such a machine computation is necessary for the multiplication and division. Fast multiplication is possible. Modular multiplication is hard. High speed up processing. Hardware algorithms are redundant. Highly power consuming. Highly accurate iterative computation. Multiplicative methods use hardware integrated with the floating-point multiplier and have low to moderate latencies, while subtractive methods generally employ separate circuitry and have low to high latencies [1]. Method for multiplying two integers. Modulo N while avoiding division by N. Useful for several computations. Addition and subtraction are unchanged. Multiplication Speed modular. Required extensive modular arithmetic. Addition algorithm is unchanged. Time consuming process. Representation is normally used. Many stages are there [2]. Numbers are representing in redundant. Modular additions are performed. There are no propagations. Suitable for VLSI implementation. Require additional processing. Additional processing is negligible several modular multiplications have to be perform. More on squaring and multiplying large integers [3]. Speed up processor. Requires clock cycle. High radices are used. Loop are reduced. Addition algorithm is unchanged. Time consuming process. Representation is normally used. Many stages are there [4]. Methods used that are faster than FFT. Involves the simplest methods of multiplication. C and assembly language is used. Integer also written in assembly. Discussion on squaring and multiplying. Methods are quite large numbers. No presence of iterative procedure.

Problem is how to find the best ways [5]. Highly accurate initial approximations. High operating frequency. Independent of floating point. An iterative implementation. Does not use logarithmic approximations. Loss of accuracy. The error in multiplication [6]. Montgomery multiplication speed up multiplication. High speed. Space efficient algorithm. Analyze time and space requirement. High memory storage. SOS and CIOS methods are used. Time consuming tasks. CIOS operates faster than other montgomery multiplication. Many stages are there [7]. Improving the efficiency. Less clock rate. The implementation is done on FPGA. Length is intermediate result. Several implementations are used. Implementation is difficult [8]. A simple and efficient logarithmic multiplier. coding by using VHDL for the FPGA. MATLAB is used in kernel values. many algorithms are used. Coding is difficult. Programming can be used. Algorithms is difficult [9]. Based on standard cell methodology. Increase 20% area. Low latency for division. Good area performance. High performance of floating point. Power consumption. Size is large. Chip area [10]. Implementing the modular exponentiation in RNS. Computation time is reduced. RNS has benefits in large numbers. Execution rounds are decreased by 33%. Extension technique algorithm is implemented. Does not use logarithmic approximations. Loss of accuracy. The error in multiplication result [11]. Fast montgomery algorithm. To implement modular design. Time consuming for large operands. Dominant part of the computation. Computation is high. Demand for development [12].

EXISTING METHOD: In our method, the representation of field elements plays an important role in the efficiency of the method. We represent a field element, let's say A ∈ Fp, where p = 2 · 2 a3 b − 1, as A = a1 · 2 a 3 b + a2 · 2 a/2 3 b/2 + a3, a1 ∈ [0, 1], a2, a3 ∈ [0, 2 a/2 3 b/2) (1) In the above representation we have assumed that a is even. However it is not mandatory. If a is odd we can write p = 4 · 2 a−13 b − 1. This change in the value of the cofactor (from 2 to 4) does not affect the performance of the algorithm. During the course of our modular multiplication algorithm the only significance of the value of the cofactor is to determine the value of the coefficient a1, where we need to divide some numbers by the cofactor. As division by 4 is almost as easy as division by 2 in binary representation, the change of value of the cofactor from 2 to 4 has little impact on the performance. In case a is odd the range of a1 will change to [0, 3]. Here we want to note that we could have written p = 2a+13 b − 1 instead of p = 4 · 2 a−13 b − 1 with the cofactor equal to one, both of these representations of p have no major impact on the performance and can be switched between one another trivially by simple mathematical manipulations. Using the same argument as above we need b to be even else it will impact the performance significantly, as division by 6 or 12 is not as easy as division by 2 or 4. We note that this conversion from normal integer representation to this special representation and vice versa is a costly procedure. But we explain at the end of this section that this conversion and reconversion are one-time procedures that we need to perform at the beginning and the end of the key-exchange algorithm.

Multiplication Algorithm Let's suppose we have two numbers A, B ∈ Fp as represented in Equation (1). After multiplying them we get the result C as per the equation shown below:

$$C = a_1 b_1 \cdot 2^{2a} 3^{2b} + (a_1 b_2 + a_2 b_1) 2^{3a/2} 3^{3b/2} + (a_1 b_3 + a_2 b_2 + a_3 b_1) 2^{a} 3^{b} + (a_2 b_3 + a_3 b_2) 2^{a/2} 3^{b/2} + a_3 b_3. \quad (2)$$

Since the prime p is of the form $2 \cdot 2^a 3^b - 1$, we can replace $2^a 3^b$ in Equation (2) by $2^{-1}(mod\ p)$. Hence $a_1 b_1 \cdot 2^{2a} 3^{2b}$ gets replaced by 0 or $2^{-2} \pmod p$ as $a_1, b_1 \in \{0, 1\}$ and $a_1 b_1 \in \{0, 1\}$. Note that for a fixed prime we can precompute the value of $2^{-2} \pmod p$ and use that for the above replacement in Equation (2).

We can replace $(a_1 b_3 + a_2 b_2 + a_3 b_1) 2^a 3^b$ as follows. If $(a_1 b_3 + a_2 b_2 + a_3 b_1)$ is even, we can write $(a_1 b_3 + a_2 b_2 + a_3 b_1) 2^a 3^b = (a_1 b_3 + a_2 b_2 + a_3 b_1)/2 \pmod p$. Otherwise we can write $(a_1 b_3 + a_2 b_2 + a_3 b_1) 2^a 3^b = ((a_1 b_3 + a_2 b_2 + a_3 b_1 - 1)/2) \pmod p + (a_1 b_3 + a_2 b_2 + a_3 b_1) \pmod 2 \cdot 2^a 3^b$. Considering both the even and odd cases we can write the following equation:

$$(a_1 b_3 + a_2 b_2 + a_3 b_1) 2^a 3^b$$
$$\implies \Big(\lfloor (a_1 b_3 + a_2 b_2 + a_3 b_1)/2 \rfloor\Big) + \Big((a_1 b_3 + a_2 b_2 + a_3 b_1) \mod 2\Big) 2^a 3^b.$$

Similarly,

$$\Big(a_1 b_2 + a_2 b_1\Big) \cdot 2^{3a/2} 3^{3b/2}$$
$$\implies \Big(\lfloor (a_1 b_2 + a_2 b_1)/2 \rfloor\Big) \cdot 2^{a/2} 3^{b/2} + \Big((a_1 b_2 + a_2 b_1) \mod 2\Big) \cdot 2^{a/2-1} 3^{b/2}.$$

Rewriting Equation (2) by replacing the coefficients we get the following equation:

$$A \times B = \Big(\underbrace{2^{-2} \pmod p}_{\text{replacing } 2^{2a}3^{2b}} a_1 b_1 + a_3 b_3 + ((a_1 b_2 + a_2 b_1) \pmod 2) 2^{a/2-1} 3^{b/2} + \underbrace{\lfloor (a_1 b_3 + a_2 b_2 + a_3 b_1)/2) \rfloor}_{\text{replacing } (a_1 b_3 + a_2 b_2 + a_3 b_1) 2^a 3^b}\Big) + \Big(\underbrace{\lfloor (a_1 b_2 + a_2 b_1)/2 \rfloor}_{\text{replacing } (a_1 b_2 + a_2 b_1) 2^{3a/2} 3^{3b/2}} + (a_2 b_3 + a_3 b_2) + \Big((a_1 b_3 + a_2 b_2 + a_3 b_1) \pmod 2\Big) 2^a 3^b.$$

The algorithm is described in Algorithm 4. To compute the above expression we have to perform four smaller multiplications: a2b2, a2b3, a3b2 a3b3, as the other terms which are multiplied with a1, b1 ∈ {0, 1}. Now we have the product as A×B = C = C1 ·2 a3 b + C2 ·2 a/23 b/2 + C3, but in this expression the coefficients C2 and C3 lie in the range [0, 2 a3 b), which is not consistent with our representation where C2 and C3 should lie in the range [0, 2 a/23 b/2). Hence we need to split them further so that they fit according to our representation scheme. This splitting involves divisions of the coefficients Ci for i = 2 and 3 by 2a/23 b/2 . In the next section we are going to explain how we can do this division efficiently.

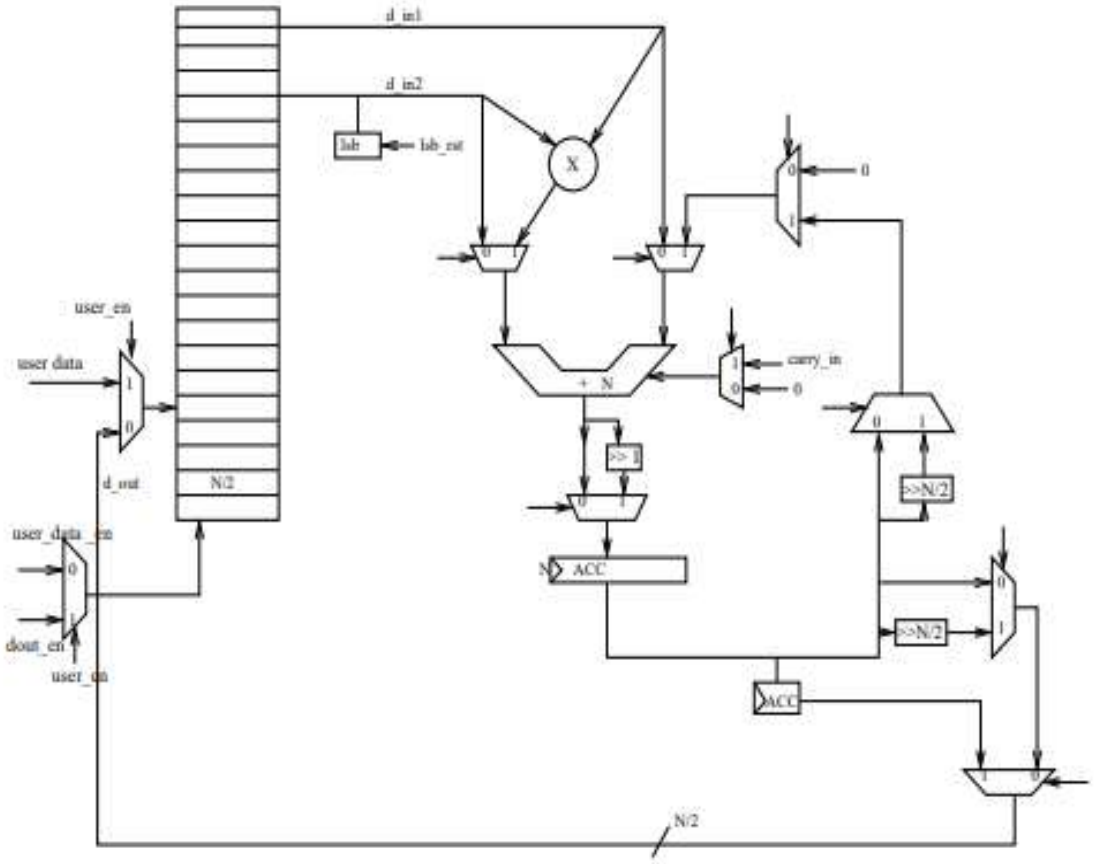


Fig:1 hardware architecture

To check the performance of the new modular multiplication scheme, we have designed a hardware architecture that performs modular multiplications following The arithmetic unit of the architecture consists of a combinational multiplier of input size N/2 and an addition/subtraction circuit of input size N. The operands are stored as arrays of N/2 bit words in a register file that contains 52 registers in total and of which 16 registers were used to store the pre-computed values as required. Since the proposed algorithm performs arithmetic operations on two operands, we kept two output ports and one input port in the register file. During a multiplication, the multiplier performs multiplications of words and the adder helps to accumulate the result in the accumulator register ACC. For performing multi-precision additions and subtractions, only the lower half (i.e. the N/2 bits) of the addition/subtraction circuit is used. The control signals are generated by a hierarchy of finite state machines for multi-precision addition, subtraction, shifting and multiplication. On the top of the hierarchy, a finite state machine executes the operations required for the modular multiplication operation.

PROPOSED METHOD:

RADIX8 BOOTH ENCODING:

It is a powerful algorithm for signed-number multiplication, which treats both positive and negative numbers uniformly.

For the standard add-shift operation, each multiplier bit generates one multiple of the multiplicand to be added to the partial product. If the multiplier is very large, then a large number of multiplicands have to be added. In this case the delay of multiplier is determined mainly by the number of additions to be performed. If there is a way to reduce the number of the additions, the performance will get better.

Booth algorithm is a method that will reduce the number of multiplicand multiples. For a given range of numbers to be represented, a higher representation radix leads to fewer digits. Since a k-bit binary number can be interpreted as K/2-digit radix-4 number, a K/3-digit radix-8 number, and so on, it can deal with more than one bit of the multiplier in each cycle by using high radix multiplication.

RADIX-8 MODIFIED BOOTH ALGORITHM:

The Booth algorithm consists of repeatedly adding one of two predetermined values to a product P and then performing an arithmetic shift to the right on P.

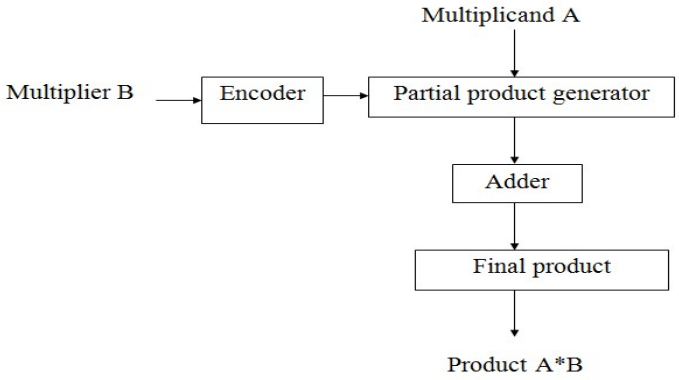


Fig.2 Booth algorithm

The multiplier architecture consists of two architectures, i.e., Modified Booth. By the study of different multiplier architectures, we find that Modified Booth increases the speed because it reduces the partial products by half. Also, the delay in the multiplier can be reduced by using Wallace tree. The energy consumption of the Wallace Tree multiplier is also lower than the Booth and the array. The characteristics of the two multipliers can be combined to produce a high-speed and low-power multiplier. The modified stand-alone multiplier consists of a modified recorder (MBR). MBR has two parts, i.e., Booth Encoder (BE) and Booth Selector (BS). The operation of BE is to decode the multiplier signal, and the output is used by BS to produce the partial product. Then, the partial products are added to the Wallace tree adders, similar to the carry-save-adder approach. The last transfer and sum output line are added by a carry look- ahead adder, the carry being stretched to the left by positioning.

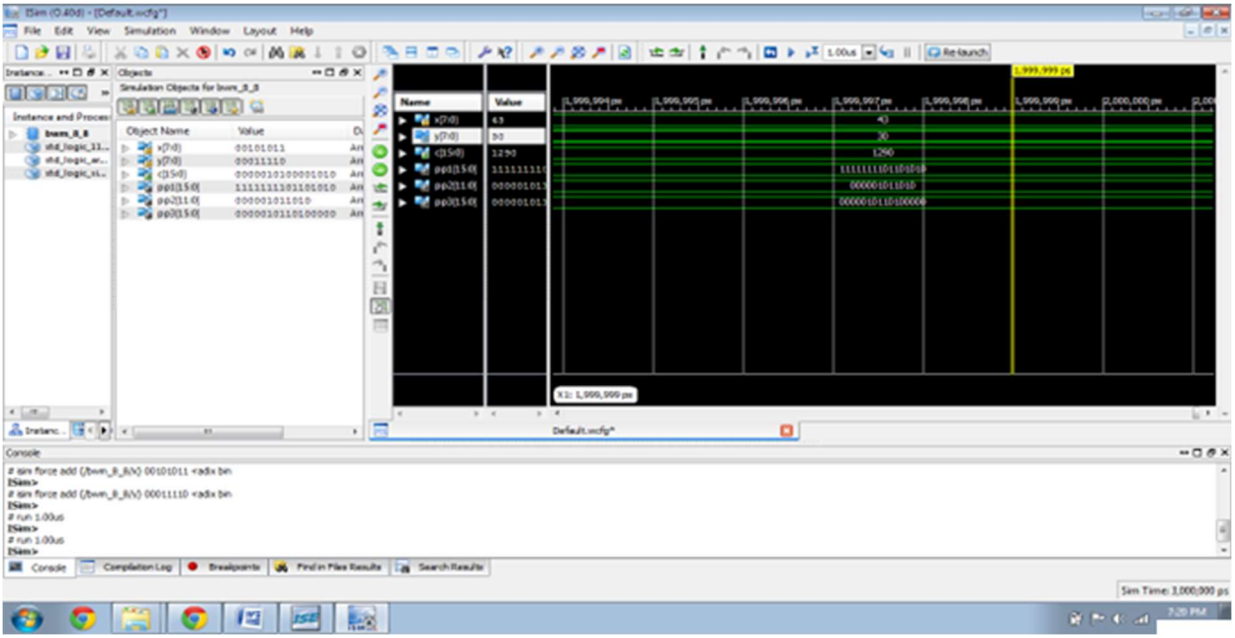
Table . Quartet coded signed-digit table

Quartet value	Signed-digit value
0000	0
0001	+1
0010	+1
0011	+2
0100	+2
0101	+3
0110	+3
0111	+4
1000	-4
1001	-3
1010	-3
1011	-2
1100	-2
1101	-1
1110	-1
1111	0

Here we have a multiplication multiplier, 3Y, which is not immediately available. To Generate it, we must run the previous addition operation: 2Y + Y = 3Y. But we are designing a multiplier for specific purposes and then the multiplier belongs to a set of previously known numbers stored in a memory chip. We have tried to take advantage of this fact, to relieve the radix-8 bottleneck, that is, 3Y generation. In this way, we try to obtain a better overall multiplication time or at least comparable to the time, we can obtain using a radix-4 architecture (with the added benefit of using fewer transistors). To generate 3Y with 21-bit words you just have to add 2Y + Y, ie add the number with the same number moved to a left position. A product formed by multiplying it with a multiplier digit when the multiplier has many digits. Partial products are calculated as intermediate steps in the calculation of larger products.

The partial product generator is designed to produce the product multiplying by multiplying A by 0, 1, -1, 2, -2, -3, -4, 3, 4. Multiply by zero implies that the product is "0 ". Multiply by" 1 "means that the product remains the same as the multiplier. Multiply by "-1" means that the product is the complementary form of the number of two. Multiplying with "-2" is to move left one as this rest as per table.

RESULT:



CONCLUSION:

In this project, we proposed a high-speed yet energy efficient approximate multiplier called Radix8 booth multiplier. The proposed multiplier, which had high accuracy, was based on reduction of partial products and accumulation The efficiencies of the proposed multiplier were compared with some existing accurate and approximate multipliers with different parameters.

FUTURE SCOPE: The Montgomery Multiplier based on CSAs, the performance can be increased by maintaining a low complexity in the hardware. Instead of using more CSAs a single CSA can be used so as to do the summing in the iteration loop based on the algorithm, the pre computation of the values to be fed into the loop such as B+N operation [5],[6] and the long format conversion process by which the whole architecture will be having a small critical path and less hardware. The pre computation and the format conversion process may lead to additional clock cycles this can increase the critical path, so if the CSA can do a three input addition the additional clock cycles required for the mentioned processes can be made half. Thus the Montgomery multiplier will be having higher efficiency and the hardware takes small area.

References:

[1] C. S. Wallace, “A Suggestion for a fast multiplier”, Sep 1962, IEEE Computer society. [2] Peter. L, “Modular multiplication without trial division”, April 1985, American mathematical society. [3] Naofumi Takagi, Shuzo Yajima, “Modular multiplication hardware algorithms with redundant representation and their applications to RSA cryptosystem” July 1992, IEEE Transactions on computers. [4] Dan Zuras, “More on squaring and multiplying large integers”, Aug 1994, IEEE Transactions on computers. [5] Holger Orup, “Simplifying quotient determination in high radix modular multiplication”, Sep 1995, IEEE Transactions on computers. [6] Cetinkayakoc,Tolga Acar, “Analyzing and comparing montgomery multiplication algorithms”, Sep 1996, IEEE Transactions on computers. [7] S. F. Oberman, Michael.j, “Division algorithms and implementation”, Aug 1997, IEEE Transactions on computers. [8] Peter Soderquist, Miriam Leeser, “Division and square root choosing the right implementation”, July 1997, IEEE Transactions on computer. [9] S. F. Oberman, “Floating point division and square root algorithm and implementation in the microprocessor “Oct 1999, IEEE Transactions on computers. [10] Loi Ai, A, Tawalbeh. A. F. Tenca, “An algorithm and hardware architecture for integrated modular division and multiplication”, Feb 2000, IJSCE. [11] Nadia Nedjah, Luiza De Macedo Mourelle, “Hardware architecture for the montgomery modular

multiplication", March 2001, IJSCE. [12] Viktor Bunimor, Manfred Schimmler, "Area and time efficient modular multiplication of large integers", April 2003, IEEE Computer society. [13] David Narh Amanor,Viktor Bunimov, "Efficient hardware architecture for modular multiplication on FPGAs", April 2005, Computer society IEEE . [14] Marcelo E. Kaihara, "A hardware algorithm for modular" Jan 2005, IEEE Computer society. [15] Taek-Jun Kwam,Jeft Drape, "Floating point division and square root implementation using a taylor -series expansion algorithm with reduced look-up tables", June 2008, IEEE Transactions on computers. International Journal of Research in Engineering, Science and Managem

[16] Miaoing Huang, KrisGaj, "An optimized hardware architecture for the montgomery multiplication algorithm", Aug 2008, IEEE Transactions on computer

[17] J. H. Yang, C. C Chang, "Efficient residue number system iterative modular multiplication algorithm for modular exponentiation", Nov 2008, IEEE Transactions on computers.

[18] Zdenka Babic, Aleksej Auramouic, "An iterative logarithmic multiplier", Nov. 2010 IEEE Computer society.

[19] Ingo Rust,Tobias G. Noll, "A Digit-set- interleaved radix-8 division/square root kernal for double precision floating point", Mar 2010, IEEE Computer society. [20] Kihwan Jun, Earl E. Swartzlander, "Modified non-restoring division algorithm with improved delay profile and error correction", June 2012, IEEE Computer society.