International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

# A Low-Power High-Performance 64-Bit Booth Multiplier for Optimized Circuit Design

**Govindu Kiranmayi [1], Dr. P. Ranjith Kumar [2]**

[1] PG Student, Dept.of ECE, Kallam Haranadhareddy Institute of Technology, Guntur, A.P, India
[2] Associate Professor, Dept.of ECE, Kallam Haranadhareddy Institute of Technology, Guntur, A.P, India

**Abstract**
Using a Recurrent Binary Partial Products Generator approach, we may lower the maximum rank of the complete item cluster produced by a radix-16 Enhanced Booth without increasing the time of the fractional item manufacturing Block. Multipliers encoded on a single line. For n = 64-bit unregistered operands, we show an optimisation for simultaneous radix-16 (adjusted) Booth recoded multiplier, which brings the maximum height of the intermediary item sections down to [n/4]. Here, we see a departure from the standard upper limit of [(n + 1)/4]. The result is a one-unit reduction in the height of the tallest individual. Arithmetic multipliers improve the visibility of an ALU and processor. To test the efficacy of the proposed approach, we juxtapose it with the Normal Booth Multiplier. Rational amalgamation was effective in terms of space, time, and power. To construct the job, we will use Verilog. The Xilinx ISE instrumentation is used for the simulation and synthesis.

**Keywords:** The normal Booth multiplier, Modified Booth encoding multiplier, Booth recoded multiplier and alternatives.

## 1. Introduction

Since binary multipliers are a frequently used constructing element in microprocessor & embedded system designs, optimising their implementation is vital [1]-[6]. In its current form, binary multiplication goes like this: 1) encode the multiplier in a particular numerical system; 2) multiply each digit by the multiplicand to produce a particular amount of partial products; 3) use multioperand addition techniques to reduce the array to two operands; and 4) perform carry-propagate in addition on the two operands. This process is outlined in [7]. An important consideration is the quantity of partial products, which is dictated by the outcome. By using a collection of integers with minimal duplication, conventional recoding techniques turn m operand s into a digit-signed operand, allowing for the recoding of -r digits [7], [8].Making each of these odd multiplies—a two-term addition or subtraction—results in a total of carry-propagate adds. However, the partial product is further reduced with a big radix, which is a gain. Take radix-16 with n-bit operands as an example; they provide about n/4 partial products. Although radix-4 is more often utilised in microprocessor implementations, radix-8[10]-[16] and radix-16 multiplication [17] are also used.

When selecting these radices to minimise the amount of pipelining flip-flops and maintain a balanced stage-to-stage delay, it is important to take into account the area, lag time, and power of

transmitted multiplying (or fusing multiplication address for an Apple Itanium CPU [17]). Partial product reductions trees still have a long way to go before they can overcome the obstacles of complex wiring & signal asymmetry, which may cause malfunctions, even if there have been recent attempts to improve energy use. Optimal pipelining is essential for both existing and future multiplication (or multiplier-add) units for two main reasons: First, even for throughput-oriented applications, the energy consumption of the pipelined unit affects the whole core [19]. Second, because of the entirety of flip-flops that have to be used and the signal propagation paths, the placement of the flip-flops used in the pipelining procedures should minimise total power consumption. floating-point units that deal with unsigned arithmetic or mantissa twice mantissa, and two's complements radix-4 Booth a multiplier, which opens the possibility of studying and expanding to higher radices. Producing the odd multiples, frequently utilising adders, especially for radix values larger than 4 causes the time delays necessary to "hide" the easier three bit assimilation. The highest point of the partial products array is raised by one row across many columns since unsigned multiplication causes an advantageous carry out during recoding. We need to add one more row for this to work. This is why expanding upon the approaches outlined in references [1] and [2] is crucial.To minimise the time it takes to generate partial products, we provide a technique that allows unsigned multipliers and partial products arrays with a height of _n/m_ for r > 4. If n is a multiple of m, the proposed method allows a row reduction because the typical unsigned multiplier has a maximum height of _(n + 1)/m_. Our approach is general, but it relies on a synthetic synthesis tool that has a standard-cell library to mitigate the impact of a single unsigned multiplier. Using radix-16 allows us to account for the most complex situation. Whether or whether the critical trajectories of the partial result generation stage is enhanced depends on the actual time of the various for all plausible values of r & n, as well as several implementation technologies. Therefore, for the benefit of our scheme's construction, we focus on a single case: 64-bit radix-16 Booth encoded among the radix's actual values. Compared to the signed multiplier, the unsigned one is much more difficult when it comes to formulating our approach. We make advantage of 64 bits as it is a nice illustration of high word length. The proposed method works well with a wide range of n values, including signed, radical eight recoding, and combinations of unsigned and signed.

## 2. Multiplier

The most basic form of the mathematical operation known as multiplication is the addition of an integer to itself a certain number of times. A product is the outcome of adding an integer (the multiplicand) to itself an integer (the multiplier) at the rate defined by the former. Placing it on top of the multiply is the way children learn to multiply in elementary school. After then, starting with the Least Significant Digit (LSD) on the right, the multiplicand multiplies itself by every single digit of the multiplier. A one-digit offset is used to align digits of a similar weight and stack intermediate outcomes (partial products) on top of each other. The total of all the partial products is what gives rise to the final product. Although binary is the most often thought of base for multiplication, this method works just as well with any basis. Just described is a simple multiplication approach. Figure 1.1 depicts the data flow for this technique. There is one digit for every black dot.

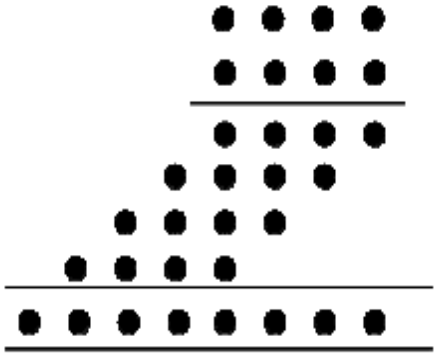

**Figure 1: Basic Multiplication**

In this case, we will pretend that MSB stands for the digit's sign. In digital electronics, multiplying is a very straightforward process. The traditional algorithm for multiplying two binary integers is its ancestor. This program multiplies two integers by performing addition & shift left operations. We may infer a method for every sort of multiplication from the preceding process. At the beginning of the process, we can also determine whether the item will be positive or negative. Once we have all of the results, we can see the product's sign on the MSB.

### 2.1 Multiplication Process

Finding the product of two integers is the quickest and easiest way to multiply them. There are three distinct phases to this process: initial product creation, subsequent product reduction, and the last addition. For a more detailed explanation of the operation, consider the following: while calculating the product by hand, as shown in figure 1.2, we get 11012 (−310) and 01012 (510), which are two two's complement integers. Sign extensions bits of the partial products are the bold italic digits. The multiplier and multiplicand are the names of the first and second operands, respectively. The product is the end outcome, whereas the partial products are the ones that get you there. With this approach directly translated to hardware, the multiplication process is shown in the image. The hardware multiplication action includes PP production, PP reduction, and final addition phases, as shown in the figures. The sum and carry bits are the two rows that come before the product. Methodically moving one bit of the multiplier to the left at a time, multiplying the multiplying factor by that bit, and then changing the intermediate item one position to the left of the prior intermediate products is how this approach works. The sum and carry bits are obtained by adding all the bits that make up the partial goods in each column. Lastly, it is necessary to add up the total and carry bits in every column. It is also possible to get m partial products and a product that is n + m bits long when an n-bit multiplicand with an m-bit multiplier are multiplied.

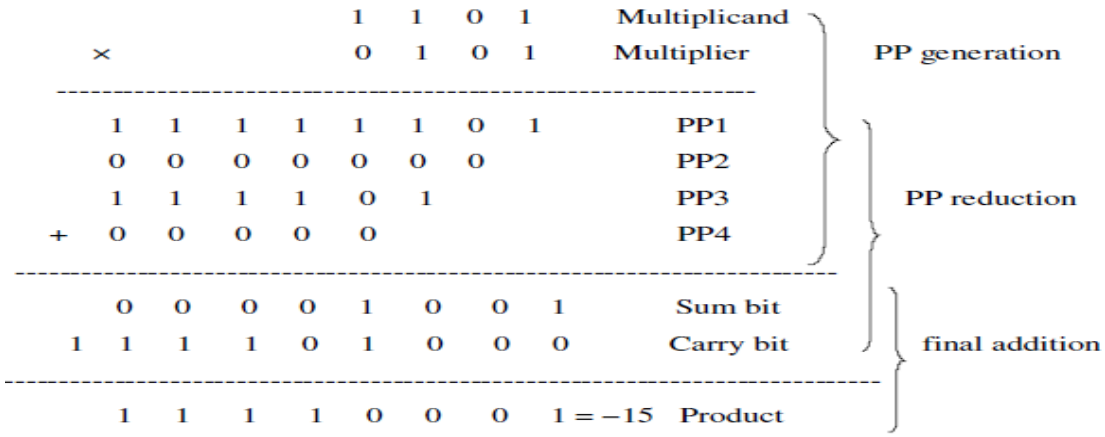


**Figure 2: Multiplication showing sum and carry**

### 2.2 Binary Multiplication

The binary number system restricts the digits, or bits, to the interval [0, 1]. Any binary number multiplied by a single bit will return either the original value or zero. This allows for the efficient and straightforward formation of the intermediate partial-products. The tedious element of using binary multipliers is adding all of these partial products. A reasonable strategy would be to create the partial products sequentially and add them up as they come up. This method, which is often implemented in software on processors without a hardware multiplier, is effective yet sluggish due to the fact that adding each new partial-product requires at least one machine cycle. Direct hardware implementation of multipliers is an option for instances where this method falls short in terms of performance. When it comes to binary multiplication, there are two primary types of numbers: signed and unsigned. In digital multiplication, the two numbers, the multiplicand and the multiplier, are blended into the result by a sequence of bit shifts and adds. When doing unsigned multiplication, it is necessary to add up to n shifted copies of the multiplicand, given that the multiplicand's bit representation is x = xn-1…..x1 x0 and the multiplier's bit representation is y = yn-1…..y1y0. There are three stages to the whole procedure: initial product creation, subsequent product reduction, and last addition.

## 3. Literature Survey

Modified booth multiplier with standard partial product array is an article by S. Kuang, J. Wang, and C. Guo. Because each partial product row has an extra partial output bit at a minimally significant bit position, the standard altered Booth encoding (MBE) results in an unbalanced partial product array. This brief suggests a straightforward approach to construct a regular part product array with fewer part product rows and little overhead, which will simplify partial product reductions and minimise the size, duration, and power of MBE multipliers. The partial product arrays of posttruncated MBE multipliers may also be regularised using the proposed approach. The proposed MBE multiplier with a regular full product array really beat the conventional ones when speaking of latency, power consumption, and area, according to the implementation findings. Complementary multipliers with a short bit width: Several applications rely on Two's complement multipliers, which F. Lamberti et al. minimise computational time. To reduce the maximum height of an incomplete product array created by a radix-4 Revised Booth Encoded multiplication by one row, we provide a solution that does not increase the latency of its partial product creation phase. Regular layouts or the partial outcome array could be compressed faster as a re-

sult of this reduction. This approach is useful for any multiplier design, but it excels in high-performance embedded cores using two's complement multipliers with a small bit width. The proposed method may be expanded to handle square and m × n rectangle multipliers of any dimension, and it can also be used to bigger radix encodings. We compared the proposed method to others since early theoretical analysis or logic synthesis showed that it was efficient in latency and area. N. Petra and colleagues' linear compensating function for fixed-width multipliers: Focusing on fixed-width multiplier with linear compensation functions, this study investigates the effects of coefficient quantisation in length. One may get alternative fixed-width multiplier topologies with different hardware complexity-accuracy trade-offs by modifying the quantisation technique. We isolate the two topologies that have performed the best. The second topology is based on a nonuniform quantisation approach, while the first one uses uniform coefficient quantisation. These novel fixed-width multiplier topologies are more accurate than previous solutions and go close to the theoretical minimum. The optimised altered booth recoder algorithm for maximising operator effectiveness in design by K. Tsoumanis and colleagues et al. Complex mathematical processes are often used in innovative (DSP) applications. The primary objective is to enhance the fusion Add-Multiply (FAM) user's design for enhanced efficiency. In order to directly encode the Revised Booth (MB) form of a combination of two numbers, we investigate strategies that integrate many different approaches. We provide a structured approach to efficient recoding and explore three different methods by implementing these techniques in FAM designs. When juxtaposed with FAM devices that use existing recoding techniques, the proposed solution drastically lowers crucial latency, hardware complexity, and battery consumption. A. Cilardo et al.'s rapid speculative multipliers built using speculation carry-save trees. This paper presents a novel approach to creating integer multiplication circuits using conjecture. This approach does the procedure more quickly, although it could be wrong sometimes. A multi-cycle error correction circuit is used in such atypical instances. The proposed speculative multiplier uses a three-stage speculation carry-save reduction tree, which begins with speculative compression and continues with partial products partitioning and recoding. Because m > 3, the speculative tree uses speculative (m:2) counters to beat a conventional tree that uses full-adders and half-adders. The article also details a way to automatically choose suitable speculative counters by taking latency and error likelihood into account. Error correction circuits and fast hypothetical carry-propagate adders complete the speculative tree. We have created speculative multipliers for different operand lengths using the UMC 65 nm library. Speculation, as opposed to conventional multipliers, is superior when processing speed is of the essence. If you need to go quickly, speculative multipliers are your best choice since they let you run faster than normal multipliers without wasting power.

## 4. Existing Method

Figure 1 depicts the basic radix-16 Booth multiplier's design. We assume n = 64 for unsigned operands for the sake of simplicity without sacrificing generalisability. The X-operand represents the multiplicand operand with bit components xi The Y-operand stands for the multiplier payload with bit components yi, and i is a real number between zero where n-1. The first stage in the process of recoding using a multiplier operand is to transform groups for four bits with values that are relative in the group {0, 1,..., 14, 15} to numbers in teh collection {−8, −7,.....…..,0,..., 7, 8}. The process of recoding is carried out using a transfer digit ti plus an intermediate digit wi. There are recoded and intermediate digits, and their sum, zi, is equal to wi plus ti. That is

The moving digit is related to the most important bit (MSB) within a four-bit group as it is needed to find out the extent to which the 16 radix digit is equal or greater to 8. The last logical stage is to add the intermediate and shift digits (0 or 1) from the radix-16 location to the right. By combining the dividing digit and the transferring digit, which can be either 1 or 0, a set of {−8,−7,..., 0,.., 7, 8} is finished. One way to get incomplete outcomes after recoding is to multiply the digits by the multiplied by X. For the set of integers {−8,−7,..., 0,..., 7, 8}, the multiples 1X, 2X, 4X, & 8X are simply calculable since they can be generated using simple logic shifts. By flipping the bits and adding a 1 to the appropriate position in the bit array of the partial products, we may get opposites of these multiples. Odd complexes that include 3X, 5X, and 7X are only possible to generate using carry-propagate based adders; the negatives of these multiples of creativity may be obtained in the same manner. Finally, to get 6X, 3X is just bit-shifted to the left.

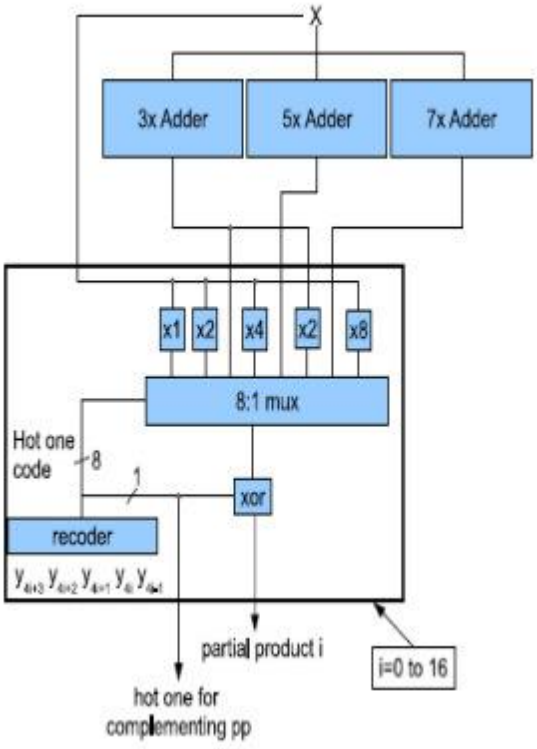


**Figure 3: Partial product generation**

Reducing the height (multi operand addition) from 17 (for n = 64) to 2 is carried out after the formation of the partial product bit array. Different techniques may result in an increase in latency or necessitate the usage of linked arrays of 3:2 carrying save adders when the maximum height is set to 17 [20]. A carry propagate addition is executed after reducing the value to two operands. The precise signal arrival timings obtained from the partial outcome reduction stage may be used in this addition.

## 5. Proposed System

### 5.1 Implementation Of Mutiplier

We use a short carry propagation addition to bring the partial outcome bit array's maximum down in parallel with the standard partial product construction. By lowering the maximum height by one row, this fast addition is an improvement over the conventional partial product generation method. In Fig. 4.1(b), we can see the bit array elements that will be inserted by the short adder. Figure 4.1(c) shows the

incomplete output bit array that results from the quick addition. In the case of n = 64, the comparison between the two numbers reveals that the maximum height decreases from 17 to 16.

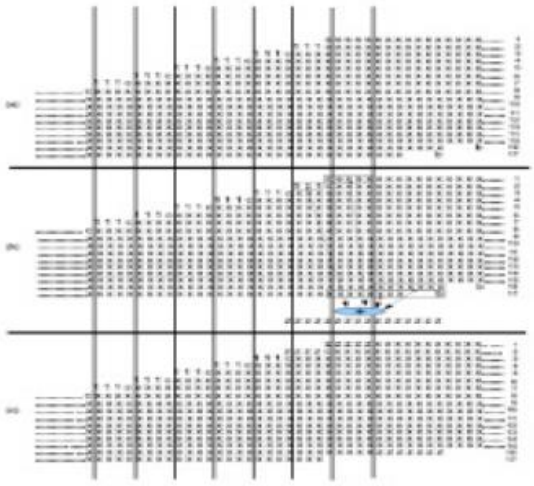

**Figure 4: Radix-16 Partial Product Reduction Array**

Because they are left-shifted by four bits, the partial products would need a costly sign extension. But a few bits are added to each incomplete product to make the sign extension easier; for instance, the first incomplete product is CSSS and all the following partial products are 111C (except for the bottom one, which isn't negative because its associated multiplier digit is either 0 or 1). The bits indicated by b in Fig. 4.1, which match logic 1, are supplemented with the two's complements for negative partial products. Two parallel processes, A and B, do the computation, as shown in Illustration 4.2. Components A and B need less time to manufacture than component B. Bit y3 is used to acquire the constituents of part A directly considering that there is no transferring digit from a preceding radix-16 digit. To put part A into practice as a speculative addition, we computed two sets of results: one using carry-in = 0 and another with carry-in = 1. An efficient way to compute this is via a compound adder. The execution of component A is shown in Figure 4.2. As a matter of speculation, the compound adder finds out the two potential outcomes. A multiplexer chooses the right outcome once the carry-in is acquired (in part B). The compound adder only uses five bits since it's easy for the carry to propagate across the three most important ones. A more involved calculation is required for part B. The need for the seven smallest bits of partial product 15 is the primary concern. Our goal is to keep the brief addition delay out of the critical path, so naturally, we can't afford to wait for half product 15 to be generated.

The proposed approach is shown in figure within the context of the whole recoding or partial product development process. The most crucial component of incomplete product 15 may still be incorrectly calculated using the same procedure as part B. When the partial result is 15 and the final product is an odd multiple, the most important element already contains a possible carry resulting from the 7 least-important bits. This principle is especially true for partial products with 15 elements. When we calculate part B, we must be careful not to produce this carry again. Let me show you how to resolve this issue. Positive odd multiples will be examined first. The computation of component B may provide two carry-outs, as seen in Figure 4.3. The 3:2 carry-save adder (Coout1) and the carry-propagate adder (Coout2) both provide two outputs. For simplicity's sake, we'll subtract the two carries generated in part B from the load that travelled to the most crucial area of partial product 15, here referred to as CM.

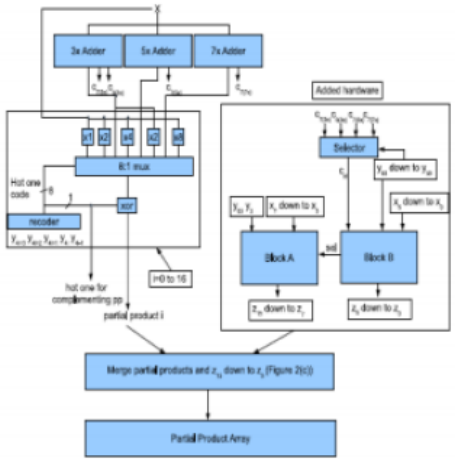


**Figure 5: Partial Product Creation Stages.**

Part A: The Ripple Carry Adder (RCA) One simple kind of adder circuit is the ripple carry adder, which uses individual complete adder cells to transmit the carry that is created during addition [4]. Only after passing the carry from the prior stage into the current stage can the result be computed [4]. The delay is inevitable because of these propagation delays, which is a big drawback. You can see the 32-bit RCA's design in figure.

The increment is computed using the CAI adder. Several RCA blocks are used by the CAI adder to get the numbers. The first RCA block takes carry-in, carry(c1), and the additions for sum and carry as input. The carry-in for the others RCA blocks will begin logic '0' in order to get a temporary total (sum1) plus carry, which are then supplied to the increment circuit. Adding the fictitious sum with carries utilising half adders, which make up the increment circuit, yields the genuine sum (sum) with charge (cy). Our goal is to get the increment circuit's carry-out by performing an operation known as OR within the carry (cy) with the carry-out from the previous step. Because the carry-in of the RCA stages is logically '0,' the amount of carry time for propagation is dramatically decreased. The CAI 32-bit architecture is shown in the figure.

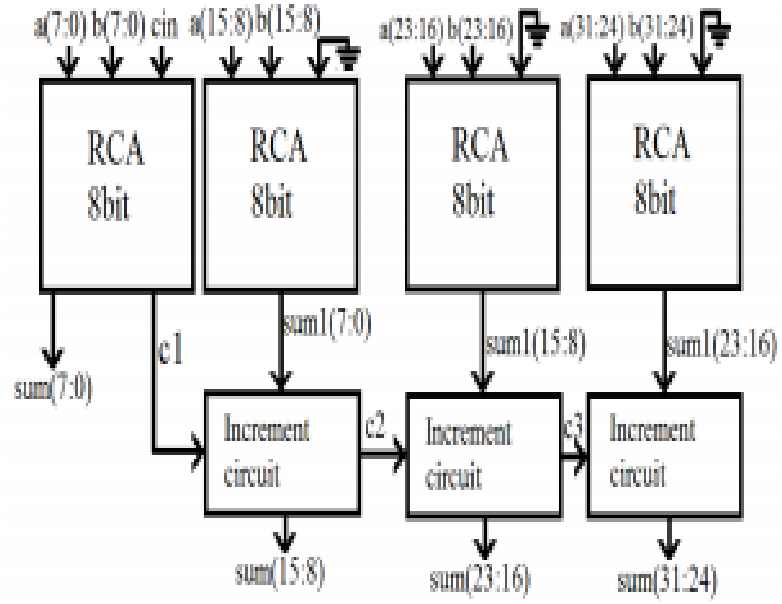


**Figure 6: 8-bit RCA blocks are used by CAI Adder**

### 5.2 MODELSIM

To see outputs and internal signals as well as to stimulate the inputs of your modules, ModelSim is a great tool to employ. While both behavioural and timing simulation are possible with it, the emphasis of this work will be on the former. Always remember that the outcomes of these simulations are directly proportional to the accuracy of the underlying models. Produced by Model TechnologyTM Incorporated are ModelSim /VHDL, ModelSim /VLOG, ModelSim /LNL, and ModelSim /PLUS. Without Model Technology's explicit written permission, you are not authorised to copy, duplicate, or reproduce in any way. Model Technology makes no promises or guarantees about the accuracy of the information included in this manual, and the contents may change at any time. No use or copying of the software contained in this manual is permitted unless specifically authorised under the licensing agreement.

### 5.3 Configurable Logic Blocks

Configurable Logic Blocks (CLBs) contain the logic for the FPGA. In large-grain architecture, these CLBs contain enough logic to create a small state machine. In fine-grain architecture, more like a true gate-array ASIC, the CLB contains only very basic logic [26]. The diagram in Figure 5.2 would be considered a large-grain block. It contains RAM for creating arbitrary combinational logic functions. It also contains flip-flops for clocked storage elements and multiplexers to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection.

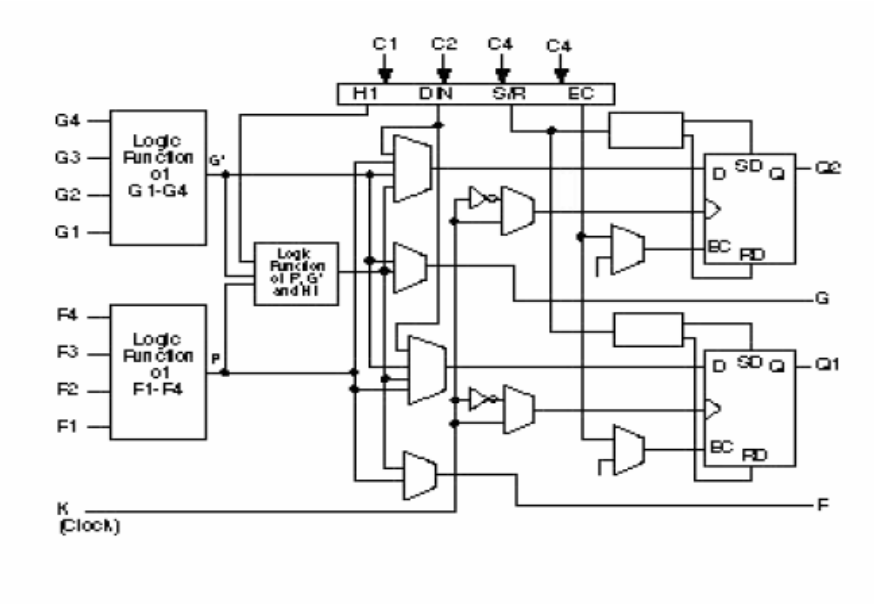


**Figure 7: FPGA Configurable Logic Block**

### 5.4 Configurable I/O Blocks

A Configurable I/O Block shown in Figure, is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three-state and open-collector output controls. Typically, there are pull-up resistors on the outputs and sometimes pull-down resistors. The polarity of the output can usually be programmed for active-high or active-low output, and often the slew rate of the output can be programmed for fast or slow rise and fall times. In addition, there is often a flip-flop on the outputs so that clocked signals can be output directly to the pins without encountering significant delay. The same is done for inputs so that there is not much delay on a signal before reaching a flip-flop, which would otherwise increase the device hold-time requirement.

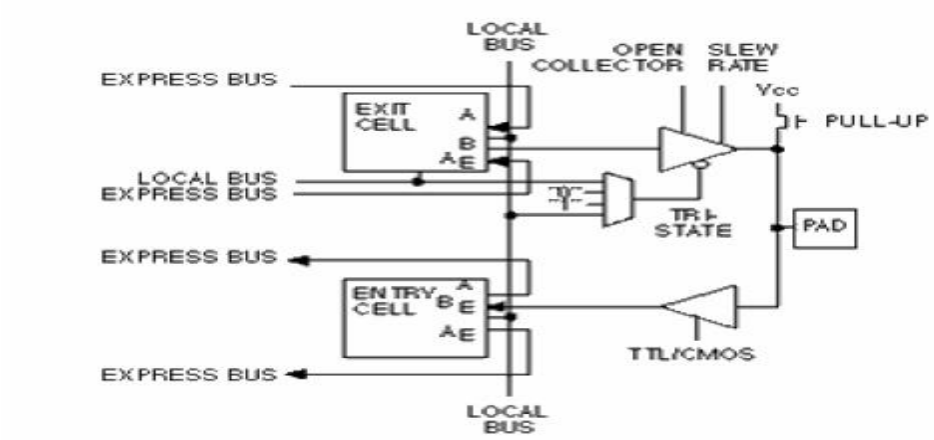


**Figure 8: FPGA Configurable I/O Block**

### 5.5 Programmable Interconnect

The interconnect of an FPGA is very different from that of a CPLD, but is rather similar to that of a gate-array ASIC. In Figure 5.4, a hierarchy of interconnect resources can be seen. There are long lines that can be used to connect critical CLBs that are physically far from each other on the chip without

inducing much delay. They can also be used as buses within the chip. There are also short lines that are used to connect individual CLBs that are located physically close to each other. There is often one or several switch matrices, similar to those in a CPLD, to connect these long and short lines together in specific ways. Programmable switches inside the chip allow the connection of CLBs to interconnect lines, interconnect lines to each other, and to the switch matrix.

## 6. Software

Computerised circuits implemented using Xilinx programmable gate arrays (FPGA) or Compound Programming Logic Device (CPLD) may be designed with the help of Xilinx Tools, a collection of programming tools. Here are the steps that make up the outline technique: (a) passing the plan, (b) combining and executing the plan, (c) testing and checking, and (d) utilitarian recreation. The aforementioned CAD tools provide a number of options for entering digital schematics, including a schematic segment instrument, an equipment depiction language (HDL) such as Verilog or VHDL, or a combination of the two. Only the plan streams that incorporates Verilog HDL will be used in this postulation.On Windows, you may launch Xilinx Tools by right-clicking the Project Navigator icon. Your screen should now display the Project Navigator window.

## 7. Simulation Results

### 7.1 Simulation Result of Multiplier

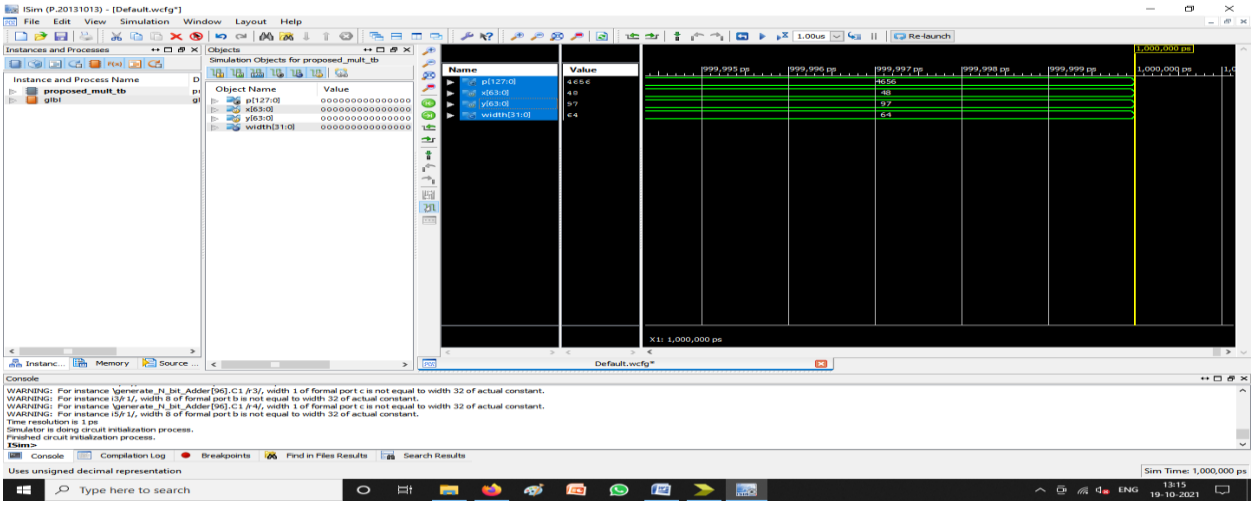

**Figure 9: Simulation Result of Multiplier**

## 7.2 Block Diagram of Multiplier

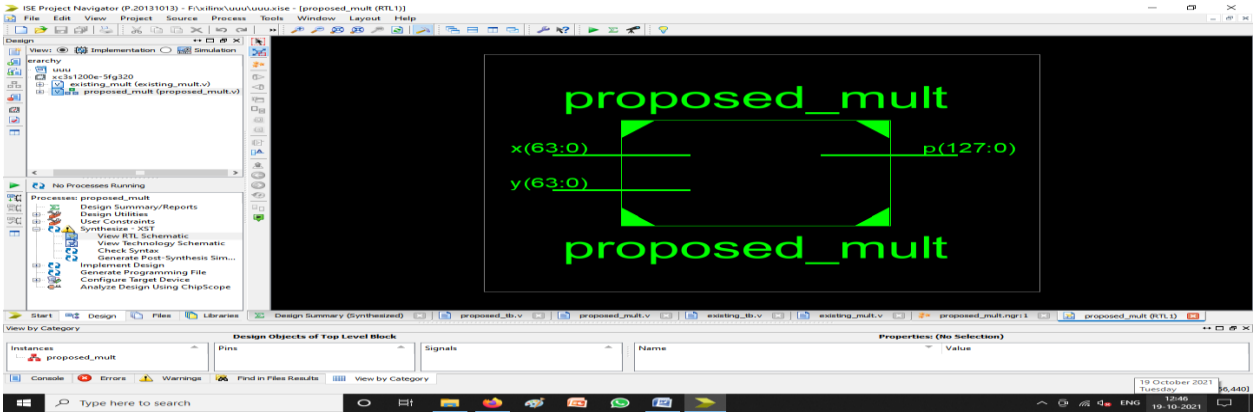


**Figure 10: Block diagram**

## 7.3 RTL Schematic

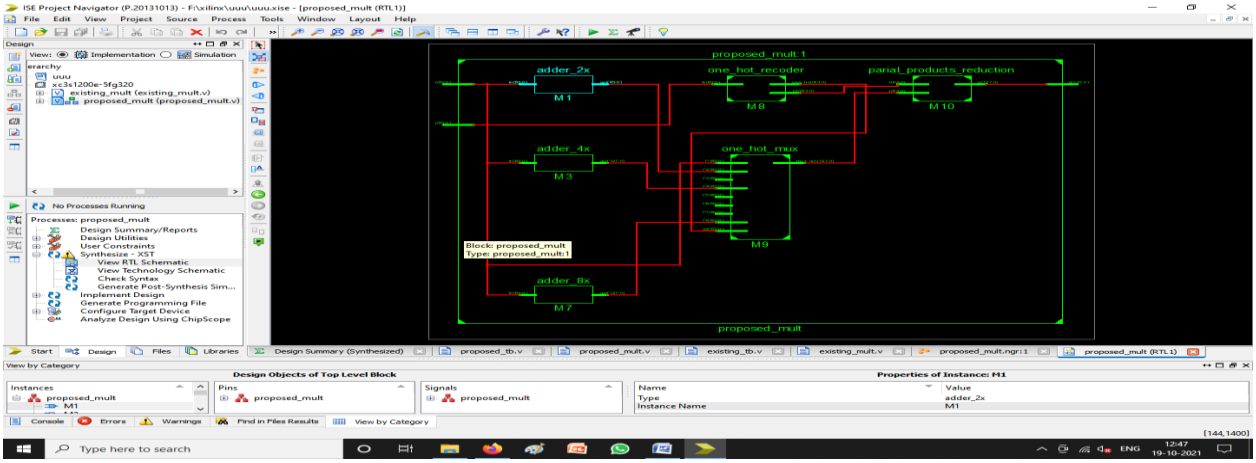


**Figure 11: Internal diagram of RTL schematic**

## 7.4 Estimation of power

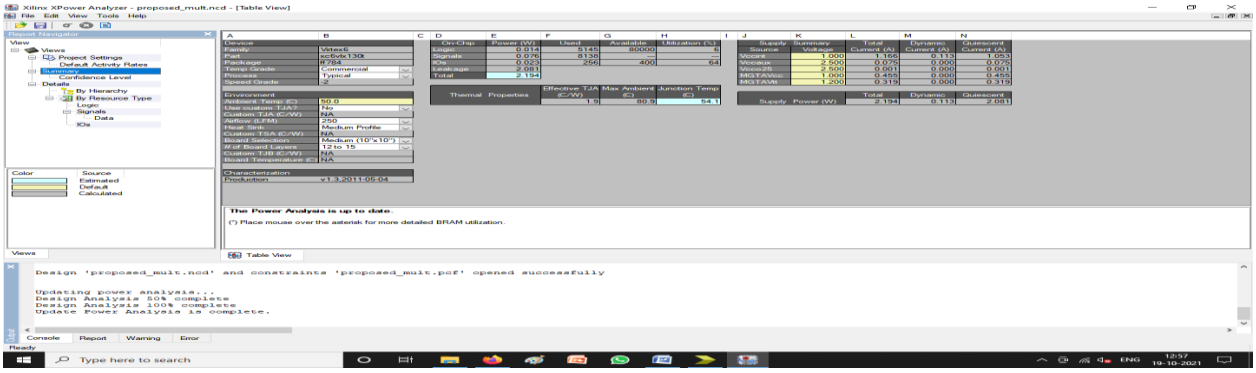

**Figure 12: Estimation of power**

### 7.5 Estimation of Delay

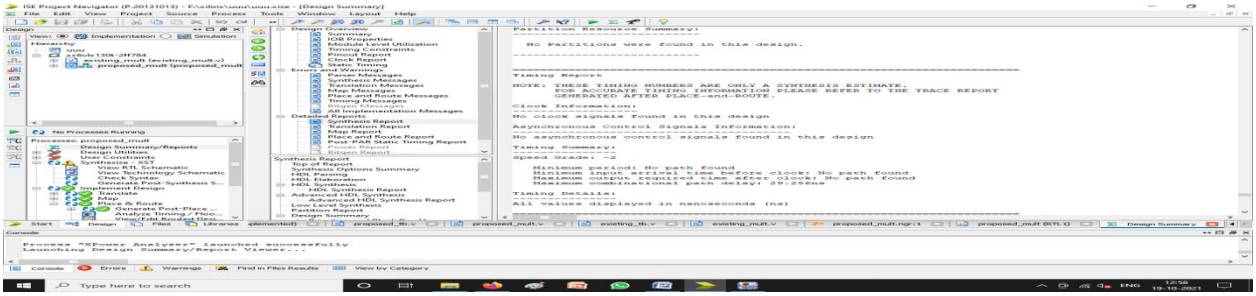

**Figure 13: Estimation of Delay**

### 7.6 Estimation of Area

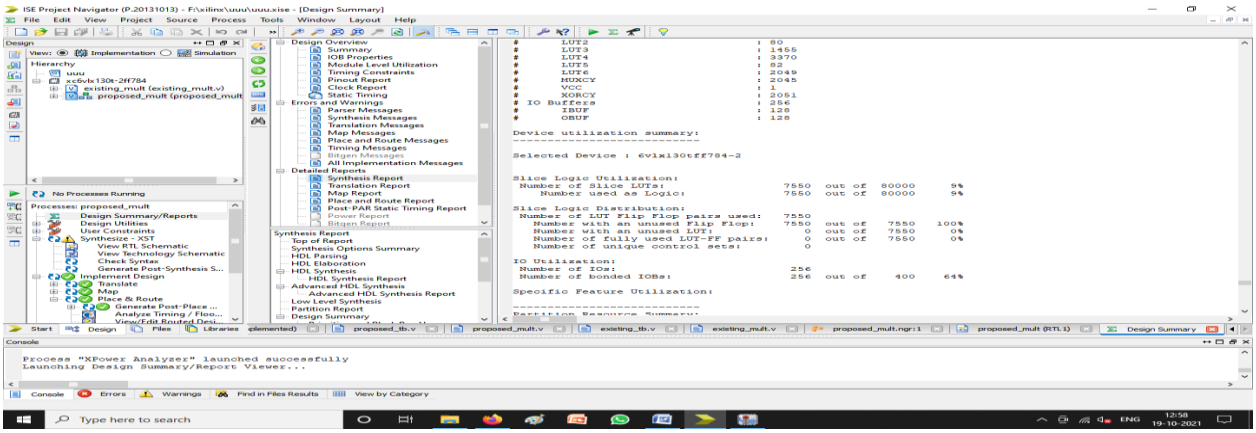

**Figure 14: Estimation of Area**

### 8. C



### onclusion

Compared to conventional Booth multipliers, the power-delay analysis generated by the multiplier using the proposed method is superior. Our method allows for a one-bit reduction in the maximum height associated with the product's component array for 64-bit and 128-bit radix-16 Booth revised magnitude multipliers. For n ≥ 32 when using a cell-based based design, this reduction eliminates any extra delay and might lead to more flexibility in reducing the tree construction of the coupled multiplier. Our recommendation is for mathematical units that use Booths encoding, including as general-purpose, multimedia, and specialist mobile application processors.

### References

1. S. Kuang, J. Wang, and C. Guo, “Modified Booth multipliers with a regular partial product array,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 56, no. 5, pp. 404–408, May 2009.
2. F. Lamberti et al., “Reducing the computation time in (short bit-width) two's complement multipliers,” IEEE Transactions on Computers, vol. 60, no. 2, pp. 148–156, Feb. 2011.
3. N. Petra et al., “Design of fixed-width multipliers with linear compensation function,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 58, no. 5, pp. 947–960, May 2011.

4. S. Galal et al., “FPU generator for design space exploration,” in Proceedings of the 21st IEEE Symposium on Computer Arithmetic (ARITH), Apr. 2013, pp. 25–34.
5. K. Tsoumanis et al., “An optimized modified Booth recoder for efficient design of the add-multiply operator,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 61, no. 4, pp. 1133–1143, Apr. 2014.
6. A. Cilardo et al., “High-speed speculative multipliers based on speculative carry-save tree,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 61, no. 12, pp. 3426–3435, Dec. 2014.
7. M. Ercegovac and T. Lang, Digital Arithmetic. Burlington, MA, USA: Morgan Kaufmann, 2004.
8. S. Vassiliadis, E. Schwarz, and D. Hanrahan, “A general proof for overlapped multiple-bit scanning multiplications,” IEEE Transactions on Computers, vol. 38, no. 2, pp. 172–183, Feb. 1989.
9. Sunil M., Ankith R. D., Manjunatha G. D., and Premananda B. S., “Design and Implementation of Faster Parallel Prefix Kogge-Stone Adder,” International Journal of Electrical and Electronic Engineering & Telecommunications, vol. 4, no. 1, pp. 116–121, 2015.
10. Raghumanohar Adusumilli and Vinod Kumar K., “Design and Implementation of a High-Speed 64-Bit Kogge-Stone Adder Using Verilog HDL,” International Journal of Electrical and Electronic Engineering & Telecommunications, vol. 3, no. 1, pp. 13–18, 2014.
11. Nurdiani Zamhari, Peter Voon, Kuryati Kipli, Kho Lee Chin, and Maimun Huja Husin, “Comparison of Parallel Prefix Adders (PPA),” Proceedings of the World Congress on Engineering (WCE 2012), vol. II, London, U.K., July 2012.
12. David H. K. Hoe, Chris Martinez, and Sri Jyothsna Vundavalli, “Design and Characterization of Parallel Prefix Adders Using FPGAs,” Proceedings of the IEEE 43rd Southeastern Symposium on System Theory (SSST), USA, pp. 168–172, Mar. 2011.
13. Jasmine Saini, Somya Agarwal, and Aditi Kansal, “Performance Analysis and Comparison of Digital Adders,” Proceedings of the IEEE International Conference on Advances in Computer Engineering and Applications (ICACEA), Ghaziabad, India, pp. 80–84, 2015.
14. Sudheer Kumar Yezerla and B. Rajendra Naik, “Design and Estimation of Delay, Power and Area for Parallel Prefix Adders,” Proceedings of IEEE RAECS, UIET Panjab University, Chandigarh, India, Mar. 2014.
15. R. Brent and H. Kung, “A Regular Layout for Parallel Adders,” IEEE Transactions on Computers, vol. C-31, no. 3, pp. 260–264, Mar. 1982.
16. K. Babulu and Y. Gowthami, “Implementation and Performance Evaluation of Prefix Adders Using FPGAs,” IOSR Journal of VLSI and Signal Processing, vol. 1, no. 1, pp. 51–57, 2012.
17. S. Sri Katyayani, Dr. M. Chandramohan Reddy, and Murali K., “Design of Efficient Han-Carlson Adder,” International Journal of Innovations in Engineering and Technology, Special Issue on ETiCE, pp. 69–75, 2016.
18. N. Weste and K. Eshragian, Principles of CMOS VLSI Design: A System Perspective, 2nd ed. Reading, MA, USA: Addison-Wesley, 1993.
19. Milos D. Ercegovac and Thomas Lang, Digital Arithmetic. San Francisco, CA, USA: Morgan Kaufmann, Elsevier, 2004.
20. J. M. Rabaey, Digital Integrated Circuits: A Design Perspective. Upper Saddle River, NJ, USA: Prentice Hall, 2001.
21. Himanshu Bansal, K. G. Sharma, and Tripti Sharma, “Digital Multiplier Designs: A Performance Comparison Review,” Innovative Systems Design and Engineering, vol. 5, no. 5, 2014.

22. K. Nehru, A. Shanmugam, and S. Vadivel, “Design of 64-Bit Low-Power Parallel Prefix VLSI Adder for High-Speed Arithmetic Circuits,” in Proceedings of the International Conference on Computing, Communication and Applications, pp. 1–4, IEEE, 2012.
23. Rakesh S. and K. S. Vijula Grace, “A Comprehensive Review on the VLSI Design Performance of Different Parallel Prefix Adders,” Materials Today: Proceedings, vol. 11, pp. 1001–1009, 2019.
24. R. Bala Sai Kesava, B. Lingeswara Rao, K. Bala Sindhuri, and N. Udaya Kumar, “Low-Power and Area-Efficient Wallace Tree Multiplier Using Carry Select Adder with Binary-to-Excess-1 Converter,” in Proceedings of the IEEE Conference on Advances in Signal Processing (CASP), pp. 248–253, 2016.