PARTIAL PRODUCT ARRAY HEIGHT REDUCTION USING RADIX-16 FOR 64-BIT BOOTH MULTIPLIER

Jenitha.A[1], Ashwini S[2], Bharath Reddy S[3], Dinesh Kumar R[4], Sahana R[5]
[1]Assoc. Prof, Dept. of ECE, Dr.TTIT, jenitha@drttit.edu.in
[2,3,4,5]Student , Dept. of ECE , Dr.T.T.I.T, bharathreddy601@gmail.com

ABSTRACT
We describe an optimization for binary radix-16 (modified) Booth recoded multipliers to reduce the maximum height of the partial product array of columns to _*n*/4_ for *n* = 64-bit unsigned operands. This is contrast to the conventional maximum height of (*n* + 1)/4. Therefore, a reduction of one unit in the maximum height of partial product is achieved. The reduction may add flexibility during the design of the pipelined multiplier to meet the required design goals, it may allow further optimizations of the partial product array reduction stage in the area/delay/power and/or may allow additional addends to be included in the partial product array without increasing the delay. The method that can be extended to the Booth recoded multipliers, signed multipliers, combined signed/unsigned multipliers, and other values of *n*.

Keywords: Partial Product, Booth recoded multipliers

I. INTRODUCTION

Binary multipliers are a widely used building block element in the design of microprocessors and embedded systems, and therefore, they are an important target for implementation optimization. Current implementations of binary multiplication follow the steps of 1) recoding of the multiplier in digits in a certain number system 2) digit multiplication of each digit by the multiplicand, resulting in a certain number of partial products 3) reduction of the partial product array to two operands using multi operand addition techniques and 4) carry-propagate addition of the two operands to obtain the final result.

The recoding type is a key issue, since it determines the number of partial products. The usual recoding process recodes a binary operand into a signed-digit operand with digits in a minimally redundant digit set [7], [8]. Specifically, for radix-r (r = 2m), the binary operand is composed of no redundant radix-r digits (by just making groups of m bits), and these are recoded from the set {0, 1,…. r − 1} to these {−r/2, . . . ,−1, 0, 1, . . ., r/2} to reduce the complexity of digit multiplications. For n-bit operands, a total of n/m partial products are generated for two’s complement representation, and (n + 1)/m for unsigned representation. The maximum column height may determine the delay and complexity of the reduction tree, In this extra column of one bit could be assimilated (with just a simplified three bit addition) with the most significant part of the first partial product without increasing the critical path of the recoding and partial product generation stage.

The result is that the partial product array has a maximum height of n/2. This reduction of one bit in the maximum height might be of interest for high-performance short-bit width two’s complement multipliers (small n) with tight cycle time constraints that are very common in SIMD digital signal processing applications.Moreover, if n is a power of two, the optimization allows to use only 4-2 carry-save adders for the reduction tree, potentially leading to regular layouts. These kind of optimizations can become particularly important as they may add flexibility to the “optimal” design of the pipelined multiplier.

Optimal pipelining in fact, is a key issue in current and future multiplier (or multiplier-add) units: 1) the latency of the pipelined unit is very important, even for throughput oriented applications, as it impacts the energy consumption of the whole core, and 2) the placement of the pipelining flip-flops should at the same time minimize total power, due to the number of flip-flops required and the unbalanced signal propagation paths. The methods proposed in [1] and [2] were mostly focused on two's complement radix-4.

Booth multipliers, thus leaving open the research and extension to higher radices and unsigned multiplications (for unsigned integer arithmetic or mantissa times in a floating-point unit).For a radix higher than 4, it is necessary to generate the odd multiples (usually with address), a resulting in the reduction of the time slacks necessary to "hide" the simplified three bit assimilation. Unsigned multiplication may produce a positive carry out during recoding (this depends of the value of n and the radix used for recoding), leading to one additional row, increasing the maximum height of the partial product array by one row, not just in one but in several columns. For all these reasons, the extend techniques are presented in [1] and [2]. In this work, the present technique that allows partial product arrays of maximum height of n/m (with the goal of not increasing the delay of the partial product generation stage), for r >4 and unsigned multipliers. Since for the standard unsigned multiplier the maximum height is (n + 1)/m, the proposed method allows a reduction of one row when n is a multiple of m.

This technique is general, but its impact (reduction of one row without increasing the critical path of the partial product generation stage) depends on the specific timing of the different components. Therefore, it cannot claim a successful result for all practical values of r and n and different implementation technologies. Thus, it concentrates on a specific instance: a 64-bit radix-16 Booth recoded unsigned multiplier implemented with a synthesis tool and a standard-cell library. Therefore by using radix-16 since it is the most complex case, among the practical values of the radix, for the design of our scheme.

The unsigned multiplier is also more complex for the design of our scheme than the signed multiplier. By using 64 bits, since it is a representative large word length. The method proposed can be adapted easily to other instances (signed, combined unsigned/signed, radix-8 recoding, different values of n).

II. METHODOLOGY

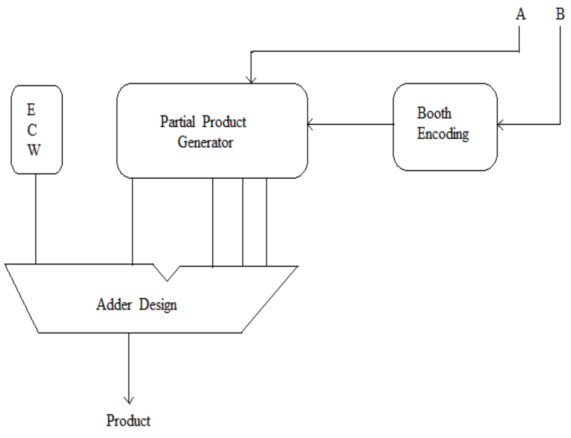


Fig 1: Block Diagram of Booth multiplier.

A & B are the Primary inputs of a Multiplier and it's given into Booth Encoding Block. Booth Encoding will generate the Encoded Data of B. The Encoded Values are given into Partial Product Generator. This Block generated Partial Product Values Based on the Radix Method.

Digital multipliers are widely used in arithmetic units of microprocessors, multimedia and digital signal processors. Many algorithms and architectures have been proposed to design high-speed and low power multipliers. This includes three steps by digital circuits in a Normal Binary (NB) multiplication. In the first step, it generates the partial product. In the second step, all partial products are added by a partial product reduction tree all the partial products are added until two partial product rows remains.

In the third step, the two partial product rows are added by the fast carry propagation adder. Two methods have been used to perform the second step for the partial product reduction. A first method uses 4-2 compressors, while a second method uses the redundant binary (RB) numbers. Both methods allows the partial product reduction tree to be reduced at a rate of 2:1.

The redundant binary number representation has been introduced to perform signed- digit arithmetic; the RB number has the capability to be represented in different ways.

For example, in radix-16 signed digit recoding the digit set is {−8, −7. . . 0, . . . , 7, 8}, so that some odd multiples of the multiplicand have to be generated. Specifically, it is required to generate ×3, ×5, and ×7 multiples (×6 is obtained by simple shift of×3). The generation of each of these odd multiplies requires a two term addition or subtraction, yielding a total of three carry-propagate additions.

The architecture of the basic radix-16 Booth multiplier is shown in Fig 1. For sake of simplicity, but without loss of generality, unsigned operands with n = 64 are considered. Let X denote the multiplicand operand with bit components xi (i = 0 to − 1, with the least-significant bit, LSB, at position 0 and with Y the multiplier operand and bit components yi.

The first step is the recoding of the multiplier operand: groups of four bits with relative values in the set {0, 1, ... 14, 15} are recoded to digits in the set{−8,−7, . . . , 0, . . . , 7, 8} (Minimally redundant radix- 16 digit set to reduce the number of multiples). This recoding is done with the help of a transfer digit ti and an interim digit wi.

After the generation of the partial product bit array, the reduction (multi operand addition) from a maximum height of 17 (for n = 64) to 2 is performed. The methods for multi operand addition are well known, with a common solution consisting of using 3 to 2 bit reduction with full adders (or 3:2 carry-save adders) or 4 to 2 bit reduction with 4:2 carry-save adders. The delay and design effort of this stage are highly dependent on the maximum height of the bit array. It is recognized that reduction arrays of 4:2 carry-save adders may lead to more regular layouts.

For instance, with a maximum height of 16, a total of 3 levels of 4:2 carry-save adders would be necessary. A maximum height of 17 leads to different approaches that may increase the delay and/or require using arrays of 3:2 carry-save adders interconnected to minimize delay [20]. After the reduction to two operands, a carry-propagate addition is performed. This addition may take advantage of the specific signal arrival times from the partial product reduction step.

Multiplier Group Bits	Operation	Multiplier Group Bits	operation
00000	0*Multiplicand	10000	-15*Multiplicand
00001	1*Multiplicand	10001	-14*Multiplicand
00010	2*Multiplicand	10010	-13*Multiplicand
00011	3*Multiplicand	10011	-12*Multiplicand
00100	4*Multiplicand	10100	-11*Multiplicand
00101	5*Multiplicand	10101	-10*Multiplicand
00110	6*Multiplicand	10110	-9*Multiplicand
00111	7*Multiplicand	10111	-8*Multiplicand
01000	8*Multiplicand	11000	-7*Multiplicand
01001	9*Multiplicand	11001	-6*Multiplicand
01010	10*Multiplicand	11010	-5*Multiplicand
01011	11*Multiplicand	11011	-4*Multiplicand
01100	12*Multiplicand	11100	-3*Multiplicand
01101	13*Multiplicand	11101	-2*Multiplicand
01110	14*Multiplicand	11110	-1*Multiplicand
01111	15*Multiplicand	11111	0*Multiplicand

Table 2.1: Booth Encoding Table

III. IMPLEMENTATION

Implementation is the process that turns strategies and plans into an actions in order to accomplish strategic objectives and goals. Implementing is the realization of an application, or execution of a plan, idea, model, design, specification, standard, algorithm, or policy. Implementation is a realization of a technical specification or algorithm as a program, software component, or other computer system through computer programming and deployment. The following blocks booth encoder, partial product generator, adder, & ecw are implemented using Modelsim.

A. FLOW DIAGRAM

Booth's multiplication examines adjacent pairs of bits of 'N'-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit. Modified Booth Encoding will generate the Encoded Data as PP i.e., partial product. The encoding process is done using chain grouping method which is shown in the example. The Encoded Values are given into Partial Product Generator. This Block generated Partial Product Values from PP0 to PP15 each of 32 bit Based on the Radix Method.

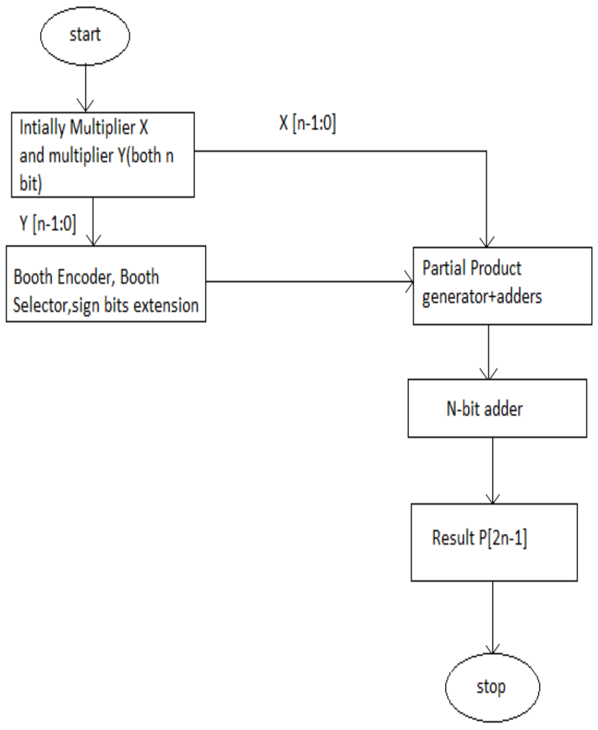

start
Intially Multiplier X and multiplier Y(both n bit)
X [n-1:0]
Y [n-1:0]
Booth Encoder, Booth Selector,sign bits extension
Partial Product generator+adders
N-bit adder
Result P[2n-1]
stop

Fig2: Flow diagram

Partial product for multiplying two or three digit numbers in columns that can be easier by making use of standard algorithm of multiplication. In a large number multiplication grouping the number to multiply into parts, multiply the parts separately, and then add. A product formed by multiplying the multiplicand by one digit of the multiplier when the multiplier has more than one digit. Partial products are used as intermediate steps in calculating larger products. The partial product to solve a multiplication equation can set it up like a traditional long multiplication equation. Just like traditional long multiplication, by multiplying the ones digit of the second factor first. When more digits are used then by multiplying the ones first, and then the tens.

The outputs of partial product generator PP0 to PP15 are given as the input to Ripple carry adder, Ripple carry adder is a digital circuit that produces the arithmetic sum of binary number it can be constructed using full-adders connected in cascaded, with the carry output. From each full-adder connected to the carry input of the next full-adder in the chain.

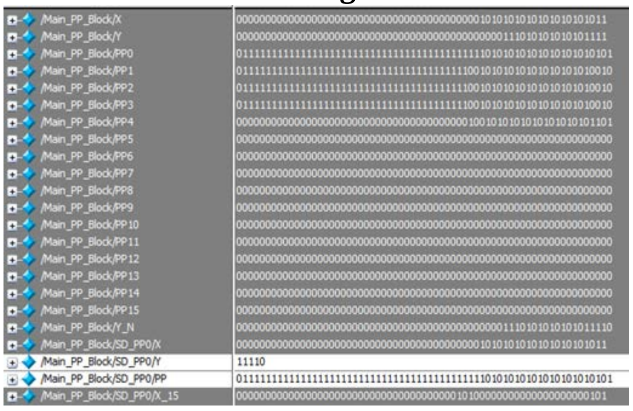
A Booth Encoding and Partial Product Generation Block is implemented. Initially, Booth Encoding is made by Padding a Zero in LSB Side. Next, Grouping is Made by Radix - 16 Method. So the Multiplier has to be grouped as 5 bits. Then Based on the Radix 16 Table by Multiply with Multiplicant to Generate Partial product Generation. Thus design make only 16 partial products in this Proposed method.

IV. RESULTS

In this presented method to reduce by one the maximum height of the partial product array for 64-bit radix-16 Booth recoded multipliers. This reduction may allow more flexibility in the design of the reduction tree of the FIR Filter Application. We have implemented the Proposed Multiplier design into FIR Filter. Radix-16 Booth recoded multipliers are attractive for low power designs, mainly to the lower complexity and depth of the reduction tree, and therefore they might be very popular in this era of power-constrained designs with increasing overheads due to wiring in Communication Design Field.

Simulation:

A. Main Partial Product generation



```
/Main_PP_Block/X          0000000000000000000000000000000000000000000010101010101010101011
/Main_PP_Block/Y          0000000000000000000000000000000000000000000011101010101010101111
/Main_PP_Block/PP0        0111111111111111111111111111111111111111110101010101010101010101
/Main_PP_Block/PP1        0111111111111111111111111111111111111110010101010101010101010010
/Main_PP_Block/PP2        0111111111111111111111111111111111111110010101010101010101010010
/Main_PP_Block/PP3        0111111111111111111111111111111111111110010101010101010101010010
/Main_PP_Block/PP4        0000000000000000000000000000000000000001001010101010101010101101
/Main_PP_Block/PP5        0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP6        0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP7        0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP8        0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP9        0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP10       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP11       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP12       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP13       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP14       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/PP15       0000000000000000000000000000000000000000000000000000000000000000
/Main_PP_Block/Y_N        0000000000000000000000000000000000000000000111010101010101011110
/Main_PP_Block/SD_PP0/X   0000000000000000000000000000000000000000000010101010101010101011
/Main_PP_Block/SD_PP0/Y   11110
/Main_PP_Block/SD_PP0/PP  0111111111111111111111111111111111111111110101010101010101010101
/Main_PP_Block/SD_PP0/X_15 0000000000000000000000000000000000000101000000000000000000000101
```

Here X and Y are the inputs

X=000000000000000000000000000000000000 0000001010101010101010101011

Y=000000000000000000000000000000000000 0000000000111010101010101111

Step 1: Booth Encoding

Pad the Zero in the LSB Multiplier

Y_N=00000000000000000000000000000000 00000000000001110101010101011110

Step 2: Grouping 1

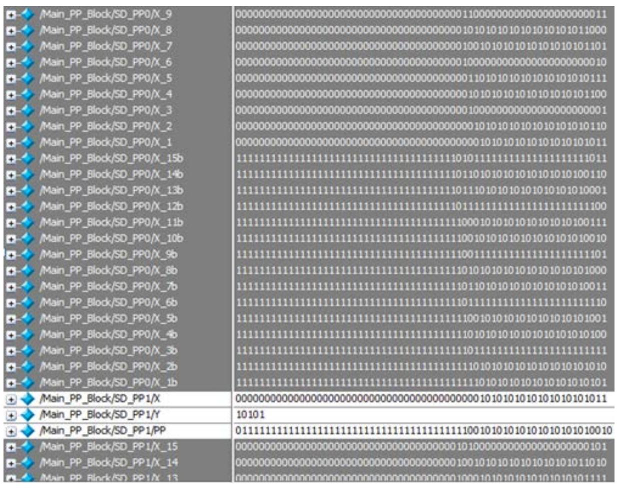
11110

Step 3: According to the Table 3.5.1

-1 multiply with Multiplicand

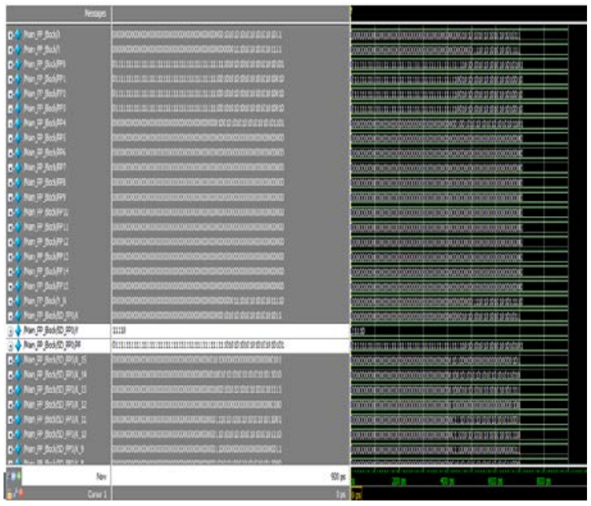
PP0=0111111111111111111111111111111111 111111111010101010101010101010101

B. Booth Encoding

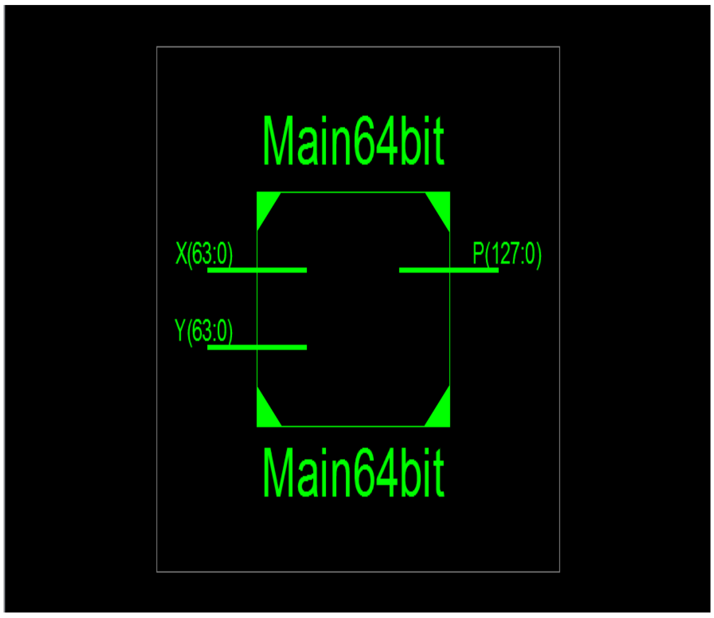


```
/Main_PP_Block/SD_PP0/X_9     000000000000000000000000000000000000001100000000000000000000011
/Main_PP_Block/SD_PP0/X_8     000000000000000000000000000000000000001010101010101010101011000
/Main_PP_Block/SD_PP0/X_7     000000000000000000000000000000000000001001010101010101010101101
/Main_PP_Block/SD_PP0/X_6     000000000000000000000000000000000000001000000000000000000000010
/Main_PP_Block/SD_PP0/X_5     000000000000000000000000000000000000000110101010101010101010111
/Main_PP_Block/SD_PP0/X_4     000000000000000000000000000000000000000101010101010101010101100
/Main_PP_Block/SD_PP0/X_3     000000000000000000000000000000000000000100000000000000000000001
/Main_PP_Block/SD_PP0/X_2     000000000000000000000000000000000000000010101010101010101010110
/Main_PP_Block/SD_PP0/X_1     000000000000000000000000000000000000000001010101010101010101011
/Main_PP_Block/SD_PP0/X_15b   111111111111111111111111111111111111111010111111111111111111011
/Main_PP_Block/SD_PP0/X_14b   111111111111111111111111111111111111111011010101010101010100110
/Main_PP_Block/SD_PP0/X_13b   111111111111111111111111111111111111111011101010101010101010001
/Main_PP_Block/SD_PP0/X_12b   111111111111111111111111111111111111111011111111111111111111100
/Main_PP_Block/SD_PP0/X_11b   111111111111111111111111111111111111111000101010101010101100111
/Main_PP_Block/SD_PP0/X_10b   111111111111111111111111111111111111111100101010101010101010010
/Main_PP_Block/SD_PP0/X_9b    111111111111111111111111111111111111111100111111111111111111101
/Main_PP_Block/SD_PP0/X_8b    111111111111111111111111111111111111111101010101010101010101000
/Main_PP_Block/SD_PP0/X_7b    111111111111111111111111111111111111111101101010101010101010011
/Main_PP_Block/SD_PP0/X_6b    111111111111111111111111111111111111111101111111111111111111110
/Main_PP_Block/SD_PP0/X_5b    111111111111111111111111111111111111111110010101010101010101001
/Main_PP_Block/SD_PP0/X_4b    111111111111111111111111111111111111111110101010101010101010100
/Main_PP_Block/SD_PP0/X_3b    111111111111111111111111111111111111111110111111111111111111111
/Main_PP_Block/SD_PP0/X_2b    111111111111111111111111111111111111111111010101010101010101010
/Main_PP_Block/SD_PP0/X_1b    111111111111111111111111111111111111111111101010101010101010101
/Main_PP_Block/SD_PP1/X       000000000000000000000000000000000000000000101010101010101010011
/Main_PP_Block/SD_PP1/Y       10101
/Main_PP_Block/SD_PP1/PP      01111111111111111111111111111111111111111100101010101010101010010
/Main_PP_Block/SD_PP1/X_15    000000000000000000000000000000000000001010000000000000000000101
/Main_PP_Block/SD_PP1/X_14    000000000000000000000000000000000000001001010101010101010101010
/Main_PP_Block/SD_PP1/X_13    0000000000000000000000000000000000000010001010101010101010101111
```

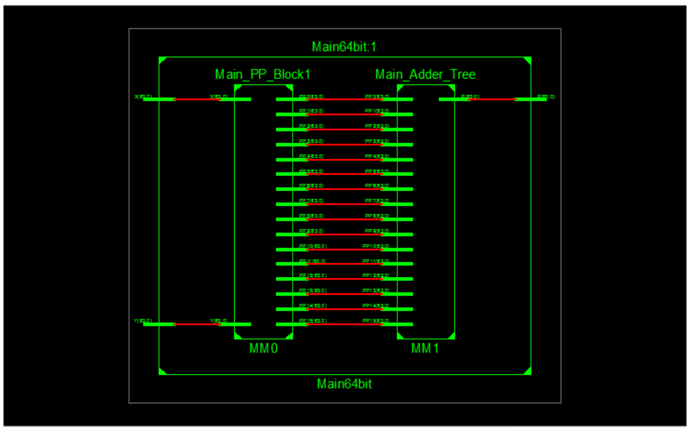
C. The Overall View



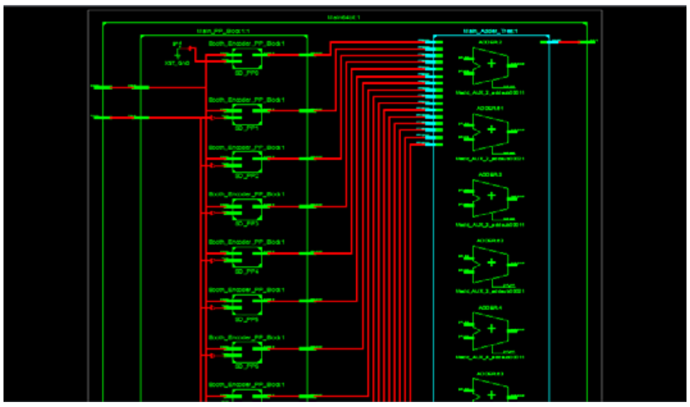
D. RTL schematic of Rdix-16 multiplier



E. RTL schematic of PP block1 and Adder tree



F.Overall Schematic view of 64-bit Booth algorithm



V. CONCLUSION

The Multiplier using the proposed algorithm improves the efficiency of area in terms of LUT Slices and gates,Decreasing the Delay and Power consumption & achieves reduction of one unit in the maximum height. Here we have presented a method to reduce by one the maximum height of the partial product array for 64-bit by describe an optimization for binary radix-16 (modified) Booth recoded multipliers to reduce the maximum height of the partial product columns to [n/4] for n = 64 - bit unsigned operands.

Radix-8 and radix-16 Booth recoded multipliers are attractive for low power designs, mainly to the lower complexity and depth of the reduction tree, and therefore they might be very popular in this era of power-constrained designs with increasing overheads due to wiring.

The booth multiplier is an efficient multiplier that can be used in the designing of digital signal processing systems. It not only performs multiplication of signed numbers without errors but also increases the speed. Memory consumption is minimal.

APPLICATIONS

i) It has the most basic advantage in digital signal processing.

ii) It is used along with multiplier-accumulator (MAC) that reduces the partial derivatives of multiplication product with ease in circuitry.

iii) It increases the efficiency of the system by enhancing its speed.

iv) Better performance in low cost at low power consumption.

REFERENCE

[1] Weiqiang Liu, Liangyu Qian, Chenghua Wang, and Jie Han "*Design of Approximate Radix 4 Booth Multipliers for Error-Tolerant Computing* ," IEEE Trans. INSPEC Accession Number: 17149541 Aug. 1 2017

[2] F. Lamberti et al., "*Reducing the computation time in (short bit-width) twos complement multipliers,*" IEEE Trans. Comput., vol. 60, no. 2, pp. 148–156, Feb. 2011.

[3] N. Petra et al., "*Design of fixed-width multipliers with linear compensation function,*" IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 58, no. 5, pp. 947–960, May 2011.

[4] S. Galal et al., "*FPU generator for design space exploration,*" in Proc. 21st IEEE Symp. Comput. 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,*" IEEETrans.Circuits Syst.I,Reg. 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 Trans. Circuits Syst. I, Reg. Papers, vol. 61, no. 12, pp. 3426–3435, Dec. 2014.

[7] S. Vassiliadis, E. Schwarz, and D. Hanrahan, "*A general proof for overlapped multiple-bit scanning multiplications,*" IEEE Trans. Comput., vol. 38, no. 2, pp. 172–183, Feb. 1989.

[8] E. M. Schwarz, R. M. A. III, and L. J. Sigal, "*A radix-8 CMOS S/390 multiplier,*" in Proc. 13th IEEE Symp. Comput. Arithmetic (ARITH), Jul. 1997, pp. 2–9.

[9] K. Muhammad et al., "*Speed, power, area, latency tradeoffs in adaptive FIR filtering for PRML read channels,*" IEEE Trans. Very Large Scale Intgr. Syst., vol. 9, no. 1, pp. 42–51, Feb. 2001.

[10] G. Colon-Bonet and P. Winterrowd, "*Multiplier evolution: A family of multiplier VLSI implementations,*" Comput. J., vol. 51, no. 5, pp. 585–594, 2008.