Design Of High Order Compression Multiplier For High-Speed DSP Applications

B Bhavani
MTech Student, Department Of ECE, VLSI System Design, Avanthi Institute of Engg. & Tech., India.
Email: Bhaskarabhavani20@gmail.com

Dr.S. Kishore Reddy
Associate professor, HOD, Department Of ECE, VLSI System Design, Avanthi Institute of Engg. & Tech., India.
Email: kishorereddy416@gmail.com

Mr. E Nagesh
Assistant professor, Department Of ECE, VLSI System Design, Avanthi Institute of Engg. & Tech., India.
Email: nagesherugu@gmail.com

Abstract- **Redundant Binary Partial Product Generator technique are used to reduce by one row the maximum height of the partial product array generated by a radix16 Modified Booth Encoded multiplier, without any raise in the delay of the partial product creation Block. In this paper, we 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. This is in contrast to the conventional maximum height of [(n + 1)/4]. Therefore, a reduction of one unit in the maximum height is achieved. These Arithmetic multipliers increase the performance of ALU and Processors. We evaluate the proposed approach by comparison with Normal Booth Multiplier. Logic synthesis showed its efficiency in terms of delay and power consumption when the word length of each operand in the multiplier is 64bits.**

Key words **- multiplier, binary radix-16, reduction, Booth Multiplier**

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.

Manuscript received Oct 10, 2022; Revised Oct 25, 2022; Accepted Nov 4, 2022

A. *radix-16 partial products generation*

However, the advantage of the high radix is that the number of partial products is further reduced. For instance, for radix-16 and n-bit operands, about n/4 partial products are generated. Although less popular than radix-4, there exist industrial instances of radix-8. and radix-16 multiplier in microprocessors implementations. The choice of these radices is related to area/delay/power optimization of pipelined multipliers (or fused multiplier adder as in the case of a Intel Itanium microprocessor), for balancing delay between stages and/or reduce the number of pipelining flip-flops.

A further consideration is that carry-propagate adders are today highly energy-delay optimized, while partial product reductions trees suffer the increasingly serious problems related to a complex wiring and glitching due to unbalanced signal paths. It is recognized in the literature that a radix-8 recoding leads to lower power multipliers compared to radix-4 recoding at the cost of higher latency (as a combinational block, without considering pipelining). Moreover, although the radix-16 multiplier requires the generation of more odd multiples and has a more complex wiring for the generation of partial products, a recent microprocessor design considered it to be the best choice for low power (under the specific constraints for this microprocessor).

In some optimizations for radix-4 two's complement multipliers were introduced. Although for n-bit operands, a total of n/2 partial products are generated, the resulting maximum height of the partial product array is n/2 + 1 elements to be added (in just one of the columns). This extra height by a single-bit row is due to the +1 introduced in the bit array to make the two's complement of the most significant partial product (when the recoded most significant digit of the multiplier is negative). The maximum column height may determine the delay and complexity of the reduction tree, authors showed that 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.

II. EXISTING METHODS-MULTIPLERS

A. *Multipliers*

Multipliers play an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets

- High speed,
- Low power consumption,
- Regularity of layout and hence less area or even combination of them in one multiplier thus making them suitable for various high speed,
- Low power and compact VLSI implementation.

The common multiplication method is "add and shift" algorithm. In parallel multipliers number of partial products to be added is the main parameter that determines the performance of the multiplier. To reduce the number of partial products to be added, with increasing parallelism, the amount of shifts between the partial products and intermediate sums to be added will increase which may result in reduced speed, increase in silicon area due to irregularity of structure and also increased power consumption due to increase in interconnect resulting from complex routing. On the other hand "serial-parallel" multipliers compromise speed to achieve better performance for area and power consumption. The selection of a parallel or serial multiplier actually depends on the nature of application. In this lecture we introduce the multiplication algorithms and architecture and compare them in terms of speed, area, power and combination of these metrics. AND gates are used to generate the Partial Products (PP). If the multiplicand is N-bits and the Multiplier is M-bits then there is N* M partial product.

B. *History Of Multipliers*

The early computer systems had what are known as Multiply and Accumulate units to perform multiplication between two binary unsigned numbers. The Multiply and Accumulate unit was the simplest implementation of a multiplier. The basic block diagram of such a system is given below.

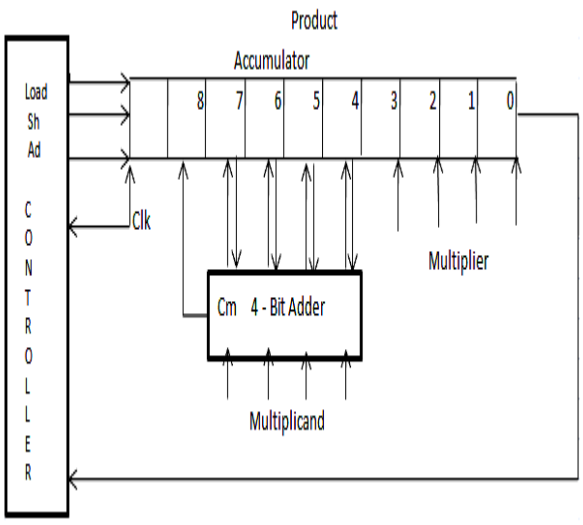


Figure 1: Multiplier Block Diagram

C. *Implementation*

The MAC unit requires a 4-bit multiplicand register, 4-bit multiplier register, a 4-

bit full adder and an 8-bit accumulator to hold the product. In the figure above the product register holds the 8-bit result. In a typical binary multiplication, based on the multiplier bit being processed, either zero or the multiplicand is shifted and then added.

III. PROPOSED MULTIPLIER

A. *Basic Radix-16 Booth Multiplier*

In this section, we describe briefly the architecture of the basic radix-16 Booth multiplier. For sake of simplicity, but without loss of generality, we consider unsigned operands with n = 64. Let us denote with X the multiplicand operand with bit components xi (i = 0 to n − 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 [8]: 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 [7]. The recoded digit zi is the sum of the interim and transfer digits

$$zi = wi + ti.$$

When the value of the four bits, vi, is less than 8, the transfer digit is zero and the interim digit wi = vi. For values of vi greater than or equal to 8, vi is transformed into vi = 16 − (16 − vi), so that a transfer digit is generated to the next radix-16 digit position (ti+1) and an interim digit of value wi = −(16 − v) is left. That is

$0 \le vi < 8 : ti+1 = 0 \; wi = vi \; wi \in [0, 7]$

$8 \le vi \le 15 : ti+1 = 1 \; wi = -(16 - vi) \; wi \in [-8, -1].$

The transfer digit corresponds to the most-significant bit (MSB) of the four-bit group, since this bit determines if the radix-16 digit is greater than or equal to 8. The final logical step is to add the interim digits and the transfer digits (0 or 1) from the radix-16 digit position to the right. Since the transfer digit is either 1 or 0, the addition of the interim digit and the transfer digit results in a final digit in the set {−8, −7,..., 0,..., 7, 8}.

Due to a possible transfer digit from the most significant radix-16 digit, the number of resultant radix-16 recoded digits is (n + 1)/4. Therefore, for n = 64 the number of recoded digits (and the number of partial products) is 17. Note that the most significant digit is 0 or 1 because it is in fact just a transfer digit. After recoding, the partial products are generated by digit multiplication of the recoded digits times the multiplicand X.

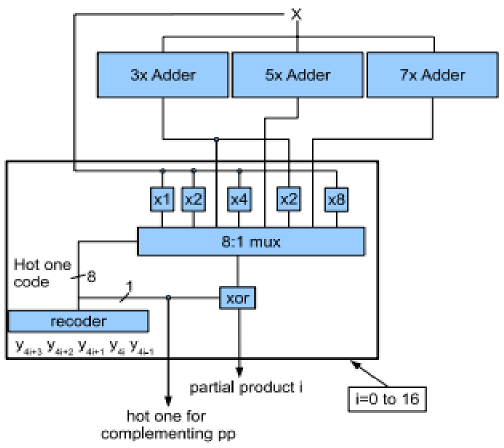


Figure 2: Partial product generation

For the set of digits {−8, −7,..., 0,..., 7, 8}, the multiples 1X, 2X, 4X, and 8X are easy to compute, since they are obtained by simple logic shifts. The negative versions of these multiples are obtained by bit inversion and addition of a 1 in the corresponding position in the bit array of the partial products. The generation of 3X, 5X, and 7X (odd multiples) requires carry-propagate adders (the negative versions of these multiples are obtained as before). Finally, 6X is obtained by a simple one bit left shift of 3X. Fig. 2.12 illustrates a possible implementation of the partial product generation. Five bits of the multiplier Y are used to obtain the recoded digit (four bits of one digit and one bit of the previous digit to determine the transfer digit to be added). The resultant digit is obtained as a one-hot code to directly drive a 8 to 1 multiplexer with an implicit zero output (output equal to zero when all the control signals of the multiplexer are zero).

The recoding requires the implementation of simple logic equations that are not in the critical path due to the generation in parallel of the odd multiples (carry-propagate addition). The XOR at the output of the multiplexer is for bit complementation (part of the computation of the two's complement when the multiplier digit is negative). Fig. illustrates part of the resultant bit array for n = 64 after the simplification of the sign extension [7].

In general, each partial product has n + 4 bits including the sign in two's complement representation. The extra four bits are required to host a digit multiplication by up to 8 and a sign bit due to the possible multiplication by negative multiplier digits. Since the partial products are left-shifted four bit positions with respect to each other, a costly sign extension would be necessary. However, the sign extension is simplified by concatenation of some bits to each partial product (S is the sign bit of the partial product and C is S complemented): CSSS for the first partial product and 111C for the rest of partial products (except the partial product at the bottom that is non negative since the corresponding multiplier digit is 0 or 1). The bits denoted by b in Fig. corresponds to the logic 1 that is added for the two's complement for negative partial products.

After the generation of the partial product bit array, the reduction (multioperand addition) from a maximum height of 17 (for n = 64) to 2 is performed. The methods for multioperand 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 [16]. 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 to use 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.

B. *Partial product generation stage including our proposed scheme*

To reduce the maximum height of the partial product bit array we perform a short carry-propagate addition in parallel to the regular partial product generation. This short addition reduces the maximum height by one row and it is faster than the regular partial product generation. Fig. shows the elements of the bit array to be added by the short adder.

Fig. shows the resulting partial product bit array after the short addition. Comparing both figures, we observe that the maximum height is reduced from 17 to 16 for n = 64. Fig. shows the specific elements of the bit array (boxes) to be added by the short carry-propagate addition. In this figure, pi,j corresponds to the bit j of partial product i, s0 is the sign bit of partial product 0, c0 = NOT(s0), bi is the bit for the two's complement of partial producti, and zi is the ith bit of the result of the short addition.

The selection of these specific bits to be added is justified by the fact that, in this way, the short addition delay is hidden from the critical path that corresponds to a regular partial product generation. We perform the computation in two concurrent parts A and B as indicated in Fig. The elements of the part A are generated faster than the elements of part B. Specifically the elements of part A are obtained from:

• the sign of the first partial product: this is directly obtained from bit y3 since there is no transfer digit from a previous radix-16 digit;

• bits 3 to 7 of partial product 16: the recoded digit for partial product 16 can only be 0 or 1, since it is just a transfer digit. Therefore the bits of this partial product are generated by a simple AND operation of the bits of the multiplicand X and bit y63 (that generates the transfer from the previous digit).

Therefore, we decided to implement part A as a speculative addition, by computing two results, a result with carry-in = 0 and a result with carry-in = 1. This can be computed efficiently with a compound add. Fig. shows the implementation of part A. The compound adder determines speculatively the two possible results. Once the carry-in is obtained (from part B), the correct result is selected by a multiplexer. Note that the compound adder is of only five bits, since the propagation of the carry through the most significant three ones is straightforward.

The computation of part B is more complicated. The main issue is that we need the 7 least-significant bits of partial product 15. Of course waiting for the generation of partial product 15 is not an option since we want to hide the short addition delay out of the critical path. We decided to implement a specific circuit to embed the computation of the least-significanbits of partial product 15 in the computation of part B (and also the addition of the bit b15). Note that for the method to be correct the computation of the partial product embedded in part B should be consistent with the regular computation performed for the most significant bits of partial product 15.

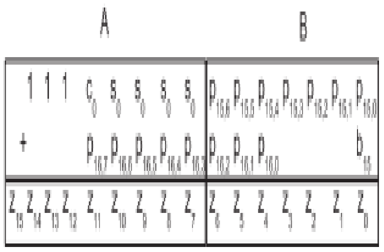


Figure 3: Detail of the elements to be added by the short addition.

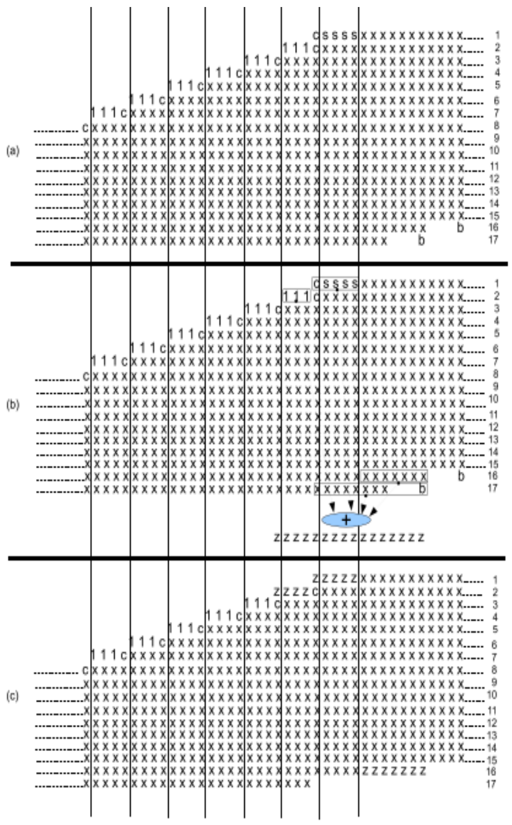


Figure 4: Radix-16 partial product reduction array

Fig. shows the computation of part B. We decided to compute part B as a three operand addition with a 3:2 carrysave adder and a carry-propagate adder. Two of the operands correspond to the least-significant bits of the partial product 15 and the other operand corresponds to the three least-significant bits of partial product 16 (that are easily obtained by an AND operation). We perform the computation of the bits of the radix-16 partial product 15 as the addition of two radix-4 partial products. Therefore, we perform two concurrent radix-4 recodings and multiple selection. The multiples of the least significant radix-4 digit are {−2, −1, 0, 1, 2}, while the multiples for the most significant radix-4 digit are {−8, −4, 0, 4, 8} (radix-4 digit set {−2, −1, 0, 1, 2}, but with relative weight of 4 with respect to the least-significant recoding). These two radix-4 recodings produce exactly the same digit as a direct radix-16 recoding for most of the bit combinations. However, among the 32 5-bit combinations for a full radix-16 digit recoding, there are six not consistent with the two concurrent radix-4 recodings. Specifically:

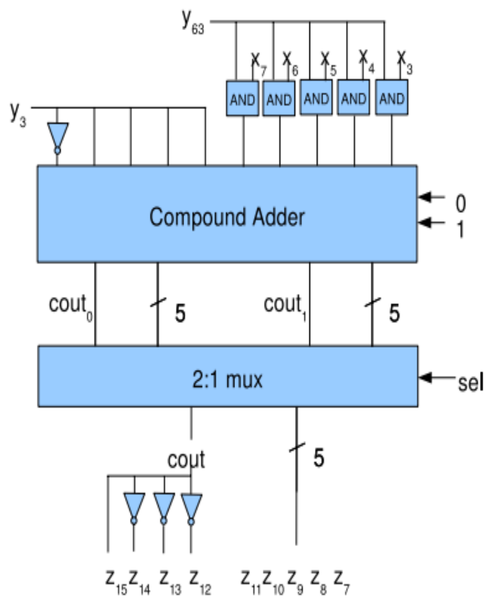


Figure 5: Speculative addition of part A

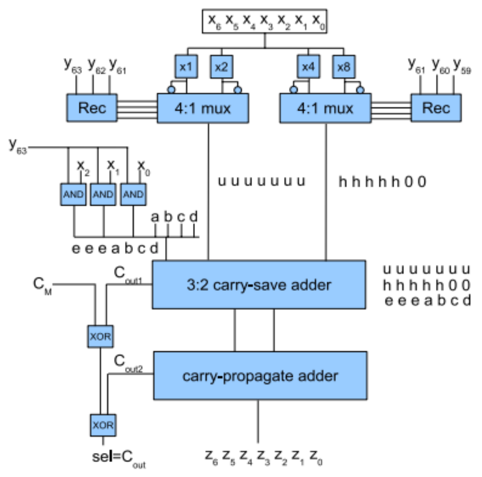


Figure 6: Computation of part B

• The bit strings 00100 and 11011 are recoded in radix-16 to 2 and −2 respectively. However, when performing two parallel radix-4 recodings the resulting digits are (4, −2) and (−4, 2) respectively. That is, the radix-4 recoding performs the computation of 2X (-2X) as 4X-2X (−4X + 2X). To have a consistent computation we modified the radix-4 recoders so that these strings produce radix-4 digits of the form (0, 2) and (0, −2).

• The bit strings 00101 and 00110 are recoded in radix-16 to 3 in both cases. However, the resulting radix-4 digits are (4, −1). This means that the radix-4 recoding performs the computation of 3X as 4X-X. To address this inconsistency problem, in this case, we decided to implement the radix-16 multiple 3X as 4X-X. This avoids the combination of radix-4 digits (2, 1) and simplifies the multiplexers in Fig.

• The bit strings 11001 and 11010 are recoded in radix-16 to −3 in both cases. However, the resulting radix-4 digits are (−4, 1). Therefore, for consistency, we proceed as in the previous case by generating the radix-16 multiple −3X as −4X + X.

In the multiplexers and place 1 in a slot of the input of the 3:2 carry-save adder with relative binary weight equal to the absolute value of the corresponding radix-4 digit. These hot ones for two's complement are indicated in Fig. 5 as the string "abcd." For instance, if the least-significant radix-4 digit is −2 and the most significant radix-4 digit is −4, then c = 1 and b = 1. Therefore, "abcd" signals are obtained directly from the selection bits of the 4:1 multiplexer. Fig. shows the recoding and partial product generation stage including the high-level view of the hardware scheme proposed. The way we compute part B may still lead to an inconsistency with the computation of the most significant part of partial product 15. Specifically, when partial product 15 is the result of an odd multiple, a possible carry from the 7 least-significant bits is already incorporated in the most significant part of the partial product. During the computation of part B we should not produce again this carry.

This issue is solved as follows. Let us consider first the case of positive odd multiples. Fig. shows that the computation of part B may generate two carry outs: the first from the 3:2 carry-save adder (Cout1), and the second from the carry-propagate adder (Cout2). To avoid inconsistencies, we detect the carry propagated to the most significant part of the partial product 15 (we call this CM) and subtract it from the two carries generated in part B. Specifically, Table I shows the truth table to generate the carry out of part B. This truth table corresponds to the XOR of the three inputs. The CM carry is obtained from a multiplexer that selects among the carry to bit position 7 from the odd multiple generators (×3, ×5, and ×7), the carry to bit position 6 from the multiple generator ×3 (to get the carry to position 7 of multiple ×6), or carry zero for the other multiples. The resultant carry out is the selection signal used in the multiplexer of part A.

For negative odd multiples we use a similar scheme. In this case the output of adder is complemented, but the only information available about the carry to position 7 is obtained directly from the adders that generate the positive odd multiple. Next, we show how to obtain the carry to the most significant part of the resultant complemented odd multiple from the carry to position 7 obtained from the adders. Let us call M the result of the positive odd multiple (output of the adder), and express

M as M = N + P (1)

with P being the seven least-significant bits of the result from the adder, and N the remaining most significant bits of the result of the adder. Let us express N in terms of C7 (carry to position 7)

N = Q + C727 (2)

that is, Q are the remaining most significant bits of the positive odd multiple minus the carry to position 7. Assuming a m bit partial product, the complement of M is expressed as

M = 2n − 1 − M = 2n − 1 − N − C727 – Q
(3)

By adding and subtracting 27 and rearranging terms results in

M = 2n − 27 − N − C727 + 27 − 1 – Q
(4)

We identify the terms N = 2n − 27 − N and Q = 27 − 1 − Q. Taking into account these terms and adding and subtracting 27 and 2n−1 results in

M = −2n−1 + N + (2n−1 − 27) + (1 − C7)27 + Q
(5)

The term (1 − C7)27 + Q = C7 + Q is computed in part B of the proposed scheme (see Fig. 2.17), but (1 − C7)27 = C7 is also part of the most significant part of partial product 15. Therefore, for a negative partial product we need to subtract C7. In summary, we take CM as the carry to position 7 of the adders that generates the multiple when the partial product is positive, and complement this carry, when the partial product is negative.

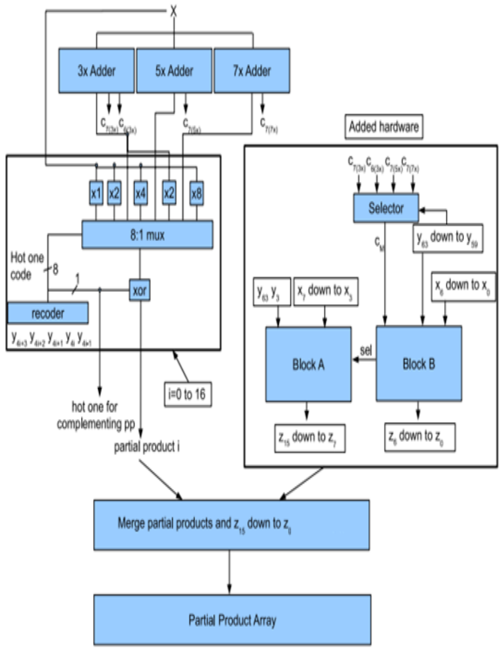


Figure 7: High level view of the recoding and partial product generation stage including our proposed scheme

IV. BOOTH MULTIPLICATION

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London.[1] Booth's algorithm is of interest in the study of computer architecture.

A. *The algorithm*

Booth's algorithm examines adjacent pairs of bits of the 'N'-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i, for i running from 0 to $N - 1$, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator P is left unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to P; and where $y_i = 1$ and $y_{i-1} = 0$, the multiplicand times 2^i is subtracted from P. The final value of P is the signed product. The representations of the multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at $i = 0$; the multiplication by 2^i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P.[2] There are many variations and optimizations on these details. The algorithm is often described as converting strings of 1s in the multiplier to a high-order +1 and a low-order −1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

B. *Booth's Algorithm Flowchart*

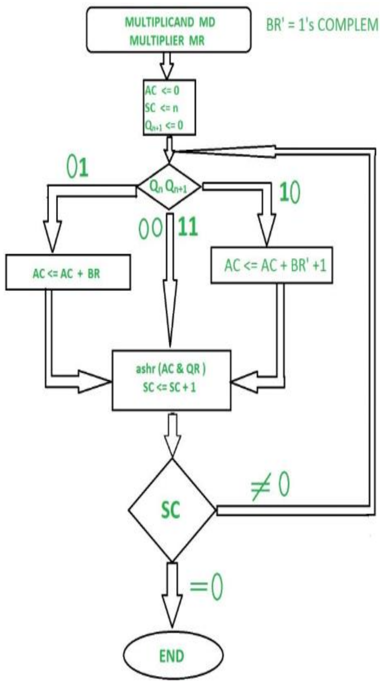
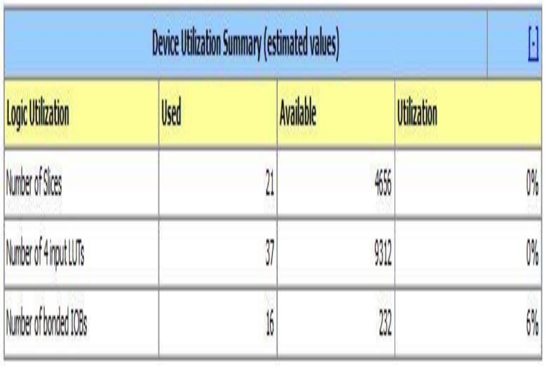


Figure 8: Booth's Algorithm Flowchart

AC and the appended bit Qn+1 are initially cleared to 0 and the sequence SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1are inspected. If the two bits are equal to 10, it means that the first 1 in a string has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the 2 bits are equal to 01, it means that the first 0 in a string of 0's has n = been encountered. This requires the addition of the multiplicand to the partial product in AC.

When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the 2 numbers that are added always have a opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including Qn+1). This is an arithmetic shift right (ashr) operation which AC and QR ti the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

V. SIMULATION RESULTS



Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization
Number of Slices	21	4656	0%
Number of 4 input LUTs	37	9312	0%
Number of bonded IOBs	16	232	6%

Figure 9: Design summary

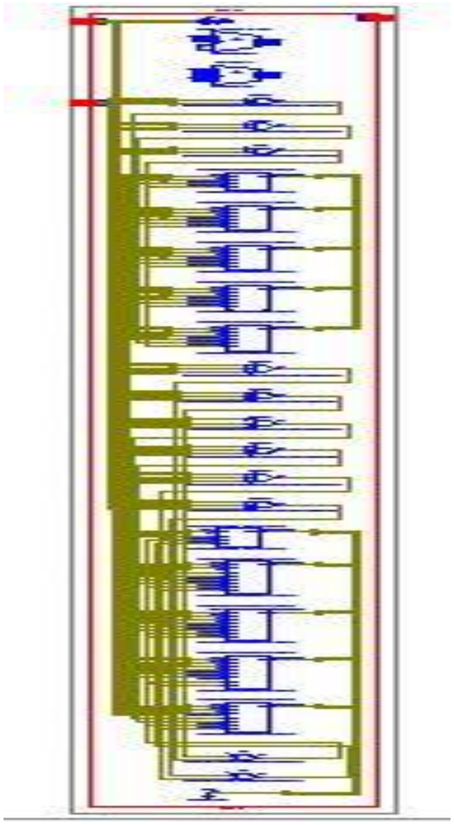
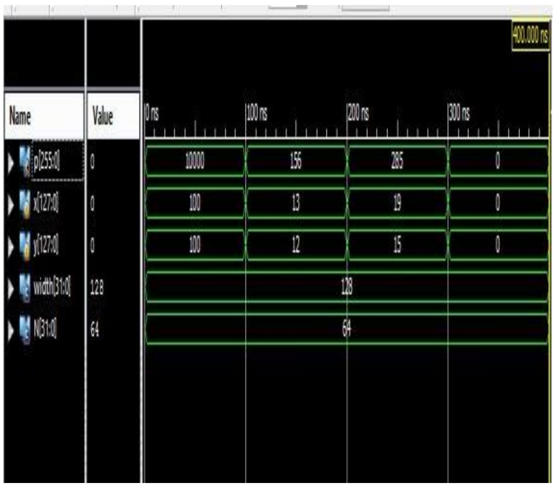
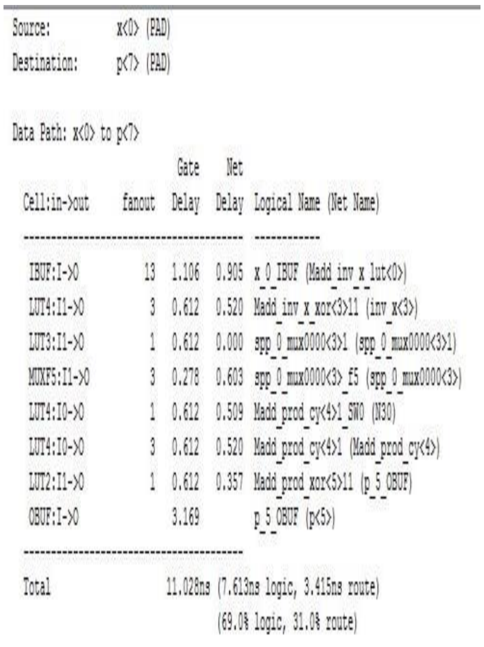


Figure 10: RTL schematic



```
                      400.000 ns
Name         Value   0 ns     100 ns    200 ns    300 ns
p[255:0]     0       10000    156       285       0
x[127:0]     0       100      13        19        0
y[127:0]     0       100      12        15        0
width[31:0]  128              128
N[31:0]      64               64
```

Figure 11: Simulation results



```
Source:          x<0> (PAD)
Destination:     p<7> (PAD)

Data Path: x<0> to p<7>
                          Gate    Net
  Cell:in->out    fanout  Delay   Delay  Logical Name (Net Name)
  ------------------------------------- -----------
   IBUF:I->O          13   1.106   0.905  x_0_IBUF (Madd_inv_x_lut<0>)
   LUT4:I1->O          3   0.612   0.520  Madd_inv_x_xor<3>11 (inv_x<3>)
   LUT3:I1->O          1   0.612   0.000  spp_0_mux0000<3>1 (spp_0_mux0000<3>1)
   MUXF5:I1->O         3   0.278   0.603  spp_0_mux0000<3>_f5 (spp_0_mux0000<3>)
   LUT4:I0->O          1   0.612   0.509  Madd_prod_cy<4>1_SW0 (N30)
   LUT4:I0->O          3   0.612   0.520  Madd_prod_cy<4>1 (Madd_prod_cy<4>)
   LUT2:I1->O          1   0.612   0.357  Madd_prod_xor<5>11 (p_5_OBUF)
   OBUF:I->O               3.169          p_5_OBUF (p<5>)
  -------------------------------------
  Total                   11.028ns (7.613ns logic, 3.415ns route)
                                   (69.0% logic, 31.0% route)
```

Figure 12: Time Summary

VI. CONCLUSION

Pipelined large wordlength digital multipliers are difficult to design under the constraints of core cycle time (for nominal voltage), pipeline depth, power and energy consumption and area. Low level optimizations might be required to meet these constraints. In this work, we have presented a method to reduce by one the maximum height of the partial product array for 64-bit radix-16 Booth recoded magnitude multipliers. This reduction may allow more flexibility in the design of the reduction tree of the pipelined multiplier. We have shown that this reduction is achieved with no extra delay for n ≥ 32 for a cell-based design. The method can be extended to Booth recoded radix-8 multipliers, signed multipliers and combined signed/unsigned multipliers. 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.

VII. FUTURE SCOPE

we will extend an optimization for binary radix-32 (modified) Booth recoded multipliers to reduce the maximum height of the partial product columns to [n/4] for n = N-bit unsigned operands. This is in contrast to the conventional maximum height of [(n + 1)/4]. Therefore, a reduction of one unit in the maximum height is achieved. This reduction may add flexibility during the design of the pipelined multiplier to meet the design goals, it may allow further optimizations of the partial product array reduction stage in terms of area/delay/power and/or may allow additional addends to be included in the partial product array without increasing the delay. The method can be extended to Booth recoded radix-8 multipliers, signed multipliers, combined signed/unsigned multipliers, and other values of n.

REFERENCES

[1] I. Blake, G. Seroussi, andN.P.Smart, Elliptic Curves in Cryptography,ser. London Mathematical Society Lecture Note Series.. Cambridge,U.K.: Cambridge Univ. Press, 1999.

[2] N. R. Murthy and M. N. S. Swamy, "Cryptographic applications of brahmaqupta-bha skara equation," IEEE Trans. Circuits Syst. I, Reg.Papers, vol. 53, no. 7, pp. 1565–1571, 2006.

[3] L. Song and K. K. Parhi, "Low-energy digit-serial/parallel finite field multipliers," J. VLSI Digit. Process., vol. 19, pp. 149–C166, 1998.

[4] P. K. Meher, "On efficient implementation of accumulation in finite field over GF(2^m) and its applications," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 17, no. 4, pp. 541–550, 2009.

[5] L. Song, K. K. Parhi, I. Kuroda, and T.Nishitani, "Hardware/software codesign of finite field datapath for low-energy Reed-Solomn codecs,"IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 8, no. 2, pp.160–172, Apr. 2000.

[6] G. Drolet, "A new representation of elements of finite fields GF(2^m) yielding small complexity arithmetic circuits," IEEE Trans. Comput.,vol. 47, no. 9, pp. 938–946, 1998.

[7] C.-Y. Lee, J.-S. Horng, I.-C. Jou, and E.-H. Lu, "Low-complexity bit-parallel systolic montgomery multipliers for special classes of GF(2^m)," IEEE Trans. Comput., vol. 54, no. 9, pp. 1061–1070, Sep. 2005.

[8] P. K. Meher, "Systolic and super-systolic multipliers for finite field GF(2^m) based on irreducible

trinomials," IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 55, no. 4, pp. 1031–1040, May 2008.

[9] J. Xie, J. He, and P. K. Meher, "Low latency systolic montgomery multiplier for finite field GF(2^m) based on pentanomials," IEEE Trans.Very Large Scale Integr. (VLSI) Syst., vol. 21, no. 2, pp. 385–389, Feb.2013.

[10] H.Wu, M. A. Hasan, I. F. Blake, and S. Gao, "Finite field multiplier using redundant representation," IEEE Trans. Comput., vol. 51, no. 11, pp. 1306–1316, Nov. 2002.

[11] A. H. Namin, H. Wu, and M. Ahmadi, "Comb architectures for finite field multiplication in F_{2m}," IEEE Trans. Comput., vol. 56, no. 7, pp. 909–916, Jul. 2007.

[12] A. H. Namin, H. Wu, and M. Ahmadi, "A new finite field multiplier using redundat representation," IEEE Trans. Comput., vol. 57, no. 5, pp. 716–720, May 2008.

[13] A. H. Namin, H.Wu, and M. Ahmadi, "A high-speed word level finite field multiplier in F_{2m} using redundant representation," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 17, no. 10, pp. 1546–1550,Oct. 2009.

[14] A. H. Namin, H. Wu, and M. Ahmadi, "An efficient finite field multiplier using redundant representation," ACMTrans. Embedded Comput. Sys., vol. 11, no. 2, Jul. 2012, Art. 31.

[15] North Carolina State University, 45 nm FreePDK wiki [Online]. Available:http://www.eda.ncsu.edu/wiki/FreePDK 45:Manual