Multipliers

Introduction

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, Modified Booth algorithm is one of the most popular algorithms. To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages. Further by combining both Modified Booth algorithm and Wallace Tree technique we can see advantage of both algorithms in one multiplier. However 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.

Multiplication Algorithm

The multiplication algorithm for an N bit multiplicand by N bit multiplier is shown below:

$Y = Y_{n-1}\ Y_{n-2}\ \ldots\ldots\ldots\ldots\ldots\ldots Y_2\ Y_1\ Y_0$ Multiplicand
$X = X_{n-1}\ X_{n-2}\ \ldots\ldots\ldots\ldots\ldots\ X_2\ X_1\ X_0$ Multiplier

Generally

$Y = Y_{n-1}\ Y_{n-2}\ \ldots\ldots\ldots\ldots\ldots\ldots Y_2\ Y_1\ Y_0$
$X = X_{n-1}\ X_{n-2}\ \ldots\ldots\ldots\ldots\ldots\ X_2\ X_1\ X_0$

==========

Yn-1X0 Yn-2X0 Yn-3X0 …… Y1X0 Y0X0
Yn-1X1 Yn-2X1 Yn-3X1 …… Y1X1 Y0X1
Yn-1X2 Yn-2X2 Yn-3X2 …… Y1X2 Y0X2
… … … …
…. …. …. …. ….

Yn-1Xn-2 Yn-2X0 n-2 Yn-3X n-2 …… Y1Xn-2 Y0Xn-2

Yn-1Xn-1 Yn-2X0n-1 Yn-3Xn-1 …… Y1Xn-1 Y0Xn-1

P2n-1 P2n-2 P2n-3 P2 P1 P0

2

Example

1101 4-bits
1101 4-bits

1101
0000
1101
1101

10101001

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. The way that the partial products are generated or summed up is the difference between the different architectures of various multipliers.

Multiplication of binary numbers can be decomposed into additions. Consider the multiplication of two 8-bit numbers A and B to generate the 16 bit product P.

```
                                        A7    A6    A5    A4    A3    A2    A1    A0
                                    X   B7    B6    B5    B4    B3    B2    B1    B0
                                    ------------------------------------------------
                                      A7.B0 A6.B0 A5.B0 A4.B0 A3.B0 A2.B0 A1.B0 A0.B0
                                +   A7.B1 A6.B1 A5.B1 A4.B1 A3.B1 A2.B1 A1.B1 A0.B1
                            +   A7.B2 A6.B2 A5.B2 A4.B2 A3.B2 A2.B2 A1.B2 A0.B2
                        +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B3 A2.B3 A1.B3 A0.B3
                    +   A7.B4 A6.B4 A5.B4 A4.B4 A3.B4 A2.B4 A1.B4 A0.B4
                +   A7.B5 A6.B5 A5.B5 A4.B5 A3.B5 A2.B5 A1.B5 A0.B5
            +  A7.B6 A6.B6 A5.B6 A4.B6 A3.B6 A2.B6 A1.B6 A0.B6
        +A7.B7 A6.B7 A5.B7 A4.B7 A3.B7 A2.B7 A1.B7 A0.B7
        ------------------------------------------------------------------------------
        P15   P14   P13   P12   P11   P10   P9    P8    P7    P6    P5    P4    P3    P2    P1   P0
```

Patial Products to be added

The equation for the addition is: $P(m+n) = A(m)B(n) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j}$.

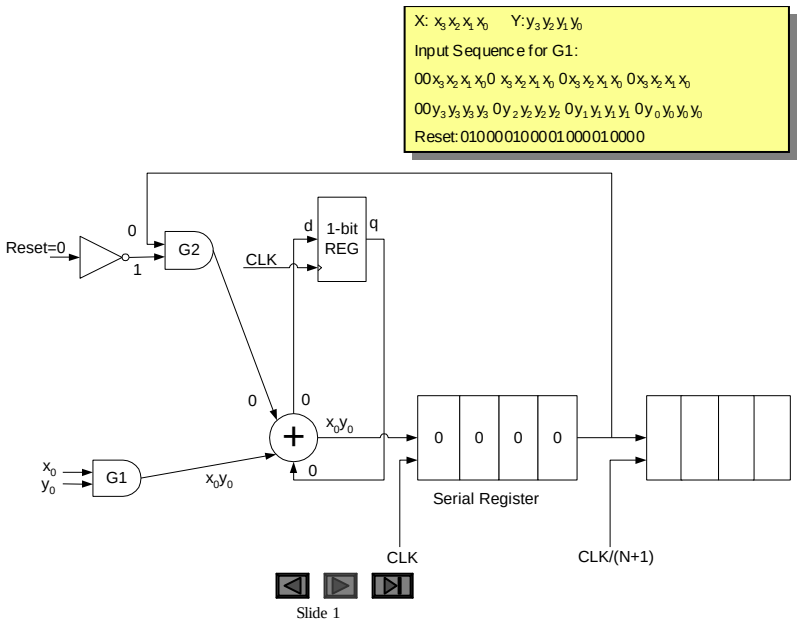
Multiplication Algorithm

- If the LSB of Multiplier is '1', then add the multiplicand into an accumulator.
- Shift the multiplier one bit to the right and multiplicand one bit to the left.
- Stop when all bits of the multiplier are zero.

From above it is clear that the multiplication has been changed to addition of numbers. If the Partial Products are added serially then a serial adder is used with least hardware. It is possible to add all the partial products with one combinational circuit using a parallel multiplier. However it is possible also, to use compression technique then the number of partial products can be reduced before addition .is performed.

Serial Multiplier

Where area and power is of utmost importance and delay can be tolerated the serial multiplier is used. This circuit uses one adder to add the m * n partial products. The circuit is shown in the fig. below for m=n=4. Multiplicand and Multiplier inputs have to be arranged in a special manner synchronized with circuit behavior as shown on the figure. The inputs could be presented at different rates depending on the length of the multiplicand and the multiplier. Two clocks are used, one to clock the data and one for the reset. A first order approximation of the delay is O (m,n). With this circuit arrangement the delay is given as D =[(m+1)n + 1] t_{fa}.

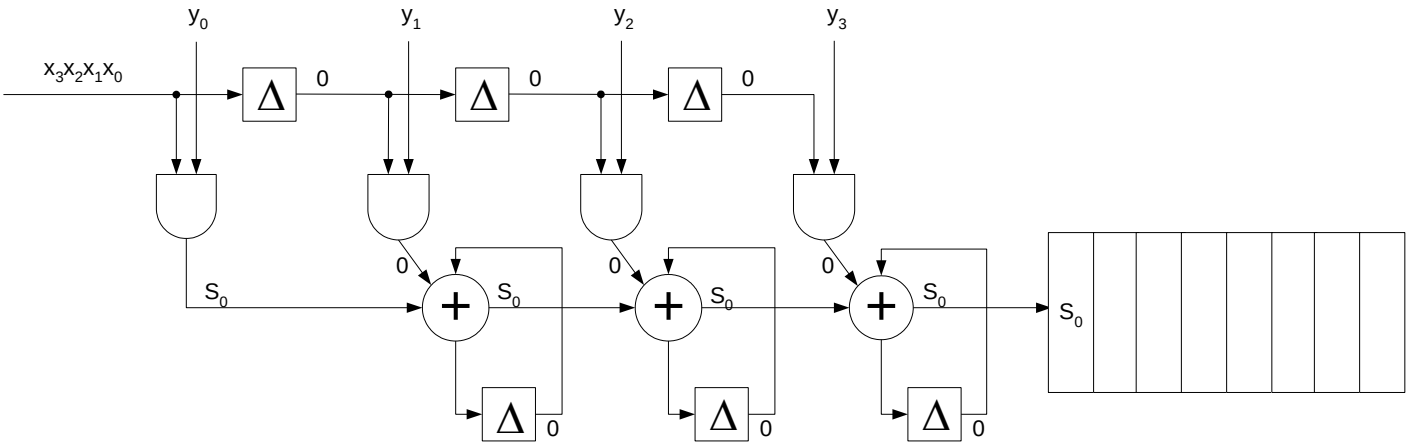


3

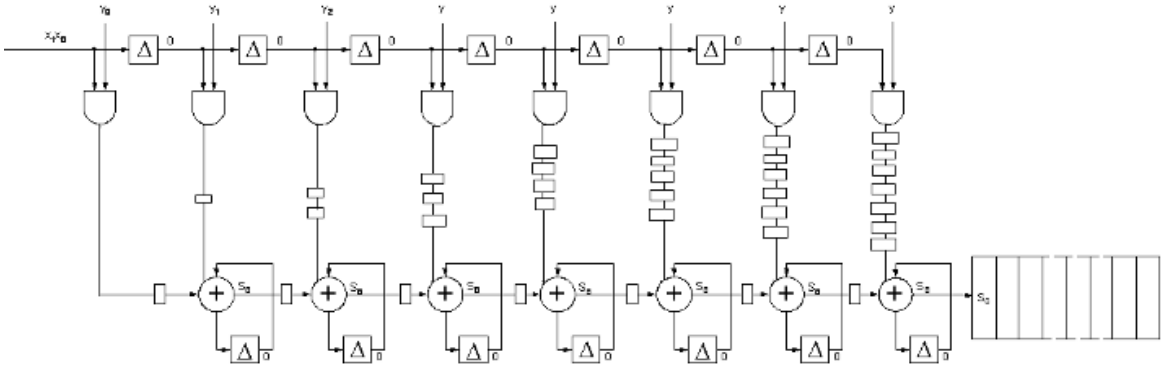
As shown the individual PP is formed individually. The addition of the PPs are performed as the intermediate values of PPs addition are stored in the DFF, circulated and added together with the newly formed PP. This approach is not suitable for large values of M or N.
For snapshots of data movements please see the course website/slides of lecture 3.

Serial/Parallel Multiplier

The general architecture of the serial/parallel multiplier is shown in the figure below. One operand is fed to the circuit in parallel while the other is serial. N partial products are formed each cycle. On successive cycles, each cycle does the addition of one column of the multiplication table of M*N PPs. The final results are stored in the output register after N+M cycles. While the area required is N-1 for M=N. For snapshots of data transfer through this multiplier please see the course website/slides of lecture

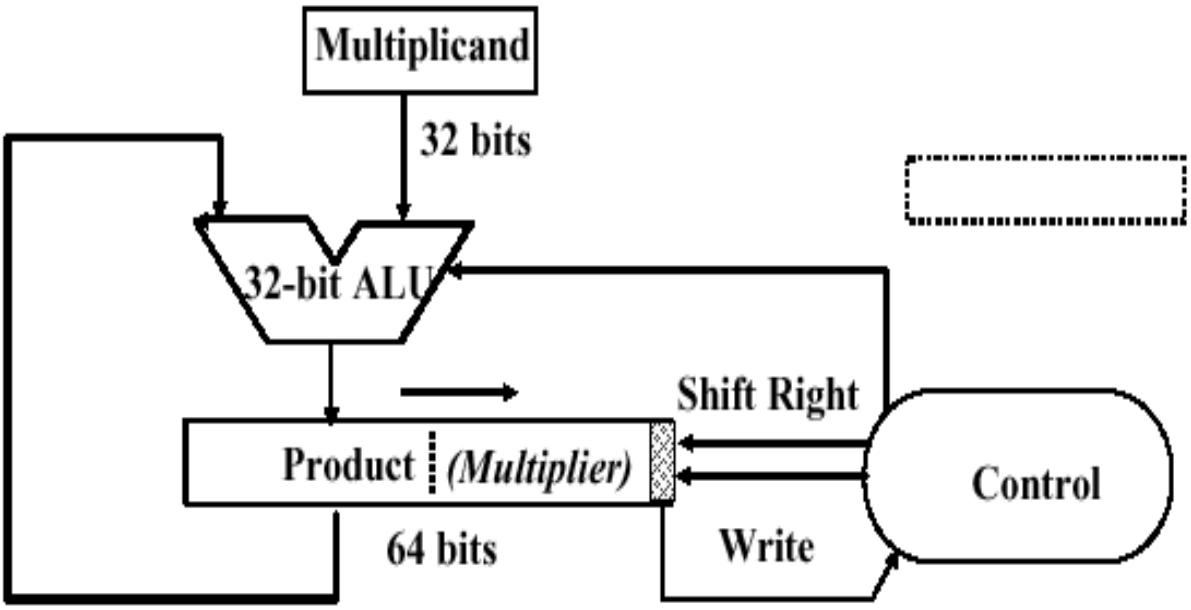


A pipelined version of an 8 bit multiplier is shown below.



Shift and Add Multiplier

The general architecture of the shift and add multiplier is shown in the figure below for a 32 bit multiplication. Depending on the value of multiplier LSB bit, a value of the multiplicand is added and accumulated. At each clock cycle the multiplier is shifted one bit to the right and its value is tested. If it is a 0, then only a shift operation is performed. If the value is a 1, then the multiplicand is added to the accumulator and is shifted by one bit to the right. After all the multiplier bits have been tested the product is in the accumulator. The accumulator is 2N (M+N) in size and initially the N, LSBs contains the Multiplier. The delay is N cycles maximum. This circuit has several advantages in asynchronous circuits. To view data movements please see course website/slides of lecture 3.

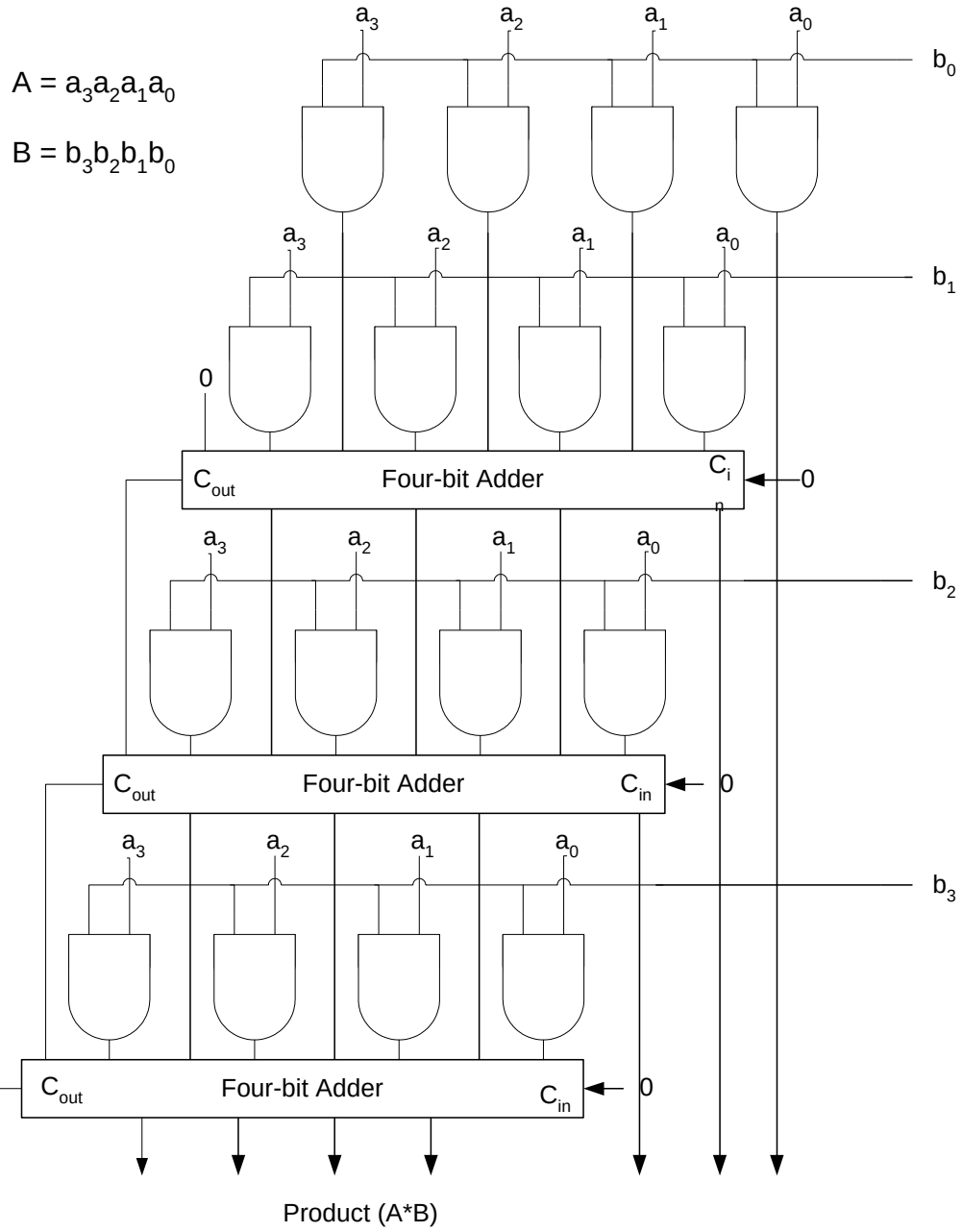


Array Multipliers

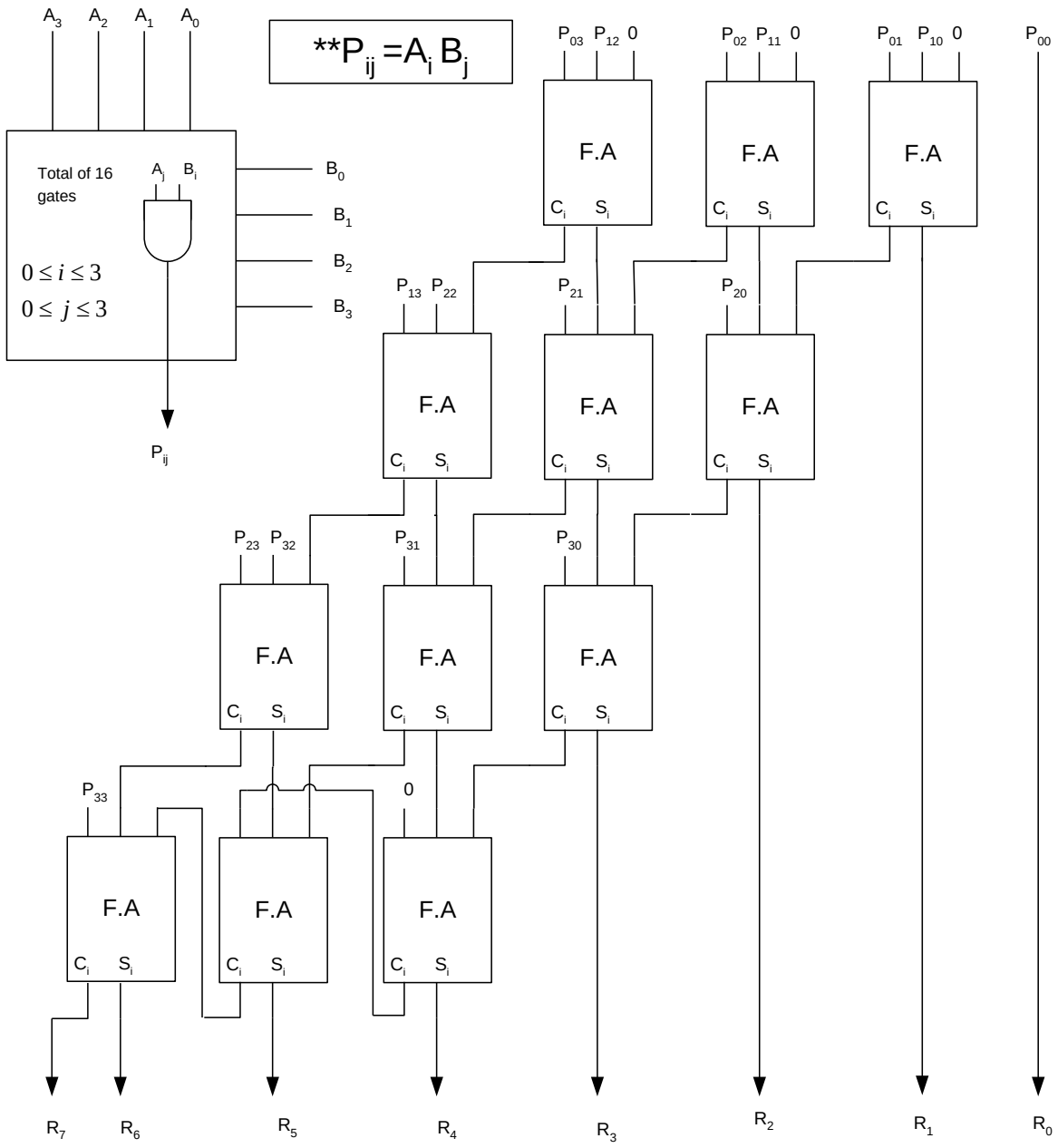
Array multiplier is well known due to its regular structure. Multiplier circuit is based on add and shift algorithm. Each partial product is generated by the multiplication of the multiplicand with one multiplier bit. The partial product are shifted according to their bit orders and then added.
The addition can be performed with normal carry propagate adder. N-1 adders are required where N is the multiplier length.

				A3	A2	A1	A0	Inputs
			x	B3	B2	B1	B0	
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0	
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0		
		C	sum	sum	sum	sum		
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0			Internal Signals
	C	sum	sum	sum	sum			
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0				
C	sum	sum	sum	sum				
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Outputs

An example of 4-bit multiplication method is shown below:



Although the method is simple as it can be seen from this example, the addition is done serially as well as in parallel. To improve on the delay and area the CRAs are replaced with Carry Save Adders, in which every carry and sum signal is passed to the adders of the next stage. Final product is obtained in a final adder by any fast adder (usually carry ripple adder). In array multiplication we need to add, as many partial products as there are multiplier bits. This arrangements is shown in the figure below



Total Area = (N-1) * M * Area $_{FA}$
Delay= 2(M-1) τ_{FA}

Now as both multiplicand and multiplier may be positive or negative, 2's complement number system is used to represent them. ***If the multiplier operand is positive*** then essentially the same technique can be used but care must be taken for sign bit extension.

The reason for dealing with signed number incorrectly is the absence of sign bit expansion in this multiplier.

```
        a1  a0                      a1  a0
      X  b1  b0                   X   b1  b0
    a1b0   a0b0          a1b0 a1b0 a1b0 a0b0
a1b1 a0b1               a1b1 a1b1 a0b1
Wrong                   Correct
```

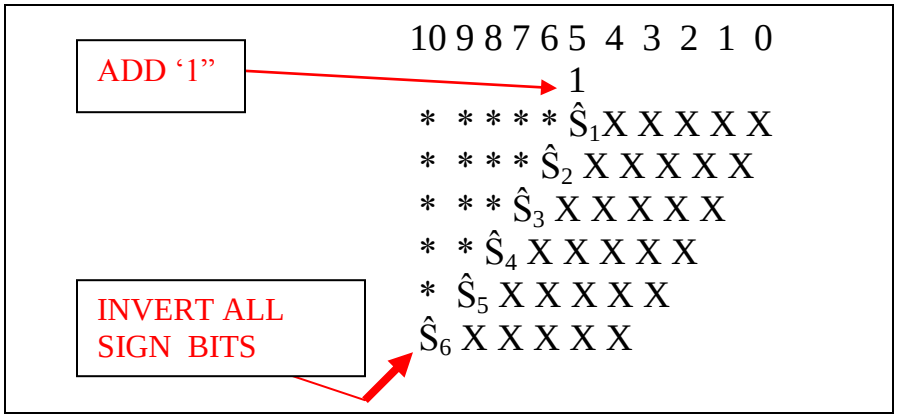
There is a way to correct this fault, which do not need to expand all of the bits in the partial product addition.
When 2's complement partial products are added in carry save arithmetic all numbers to be added in one adder stage have to be of equal bit length. Therefore, the sign bits of the partial product(s) in the first row and the sum and carry signals of each adder row are extended up to the most significant sign bit of the number with the largest absolute value to be added in this stage. The sign bit extension results in a higher capacitive load (fan out) of the sign bit signals compared to the load of other signals and accordingly slows down the speed of the circuit.
Algorithms exist when adding two partial products (**A+B**) which will eliminate the need of sign bit extension (Please see Appendix A when both numbers can be positive or negative):

1. Extend sign bit of A by one bit and invert this extended bit.
2. Invert the sign bit of B.
3. Add A and B. Add '1' to one position left of MSB of B

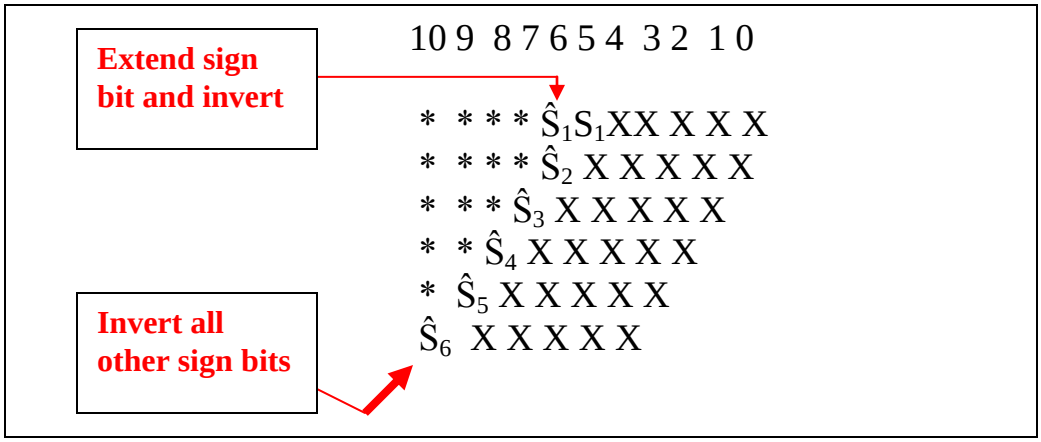
Here is an example of 6 bit sign addition:

$$
\begin{array}{lcl}
 & a_5\ a_5\ a_5\ a_4 \ldots & a_1\ a_0 \\
+ & b_5\ b_5\ b_4 & b1\ b_0 \\
\hline
 & a'_5\ a_5\ a_{4\ldots} & a_1\ a_0 \\
+ & 1\ \ b'_5\ b_4 & b_1\ b_0
\end{array}
$$

In General we can invert all the sign bits and add a "1" to column n as shown in the diagram below:



It is possible however to simplify this further and use the following template. Extend the sign of the first partial product row by 1 bit and invert this bit. Invert all other sign bits of all partial products as shown below



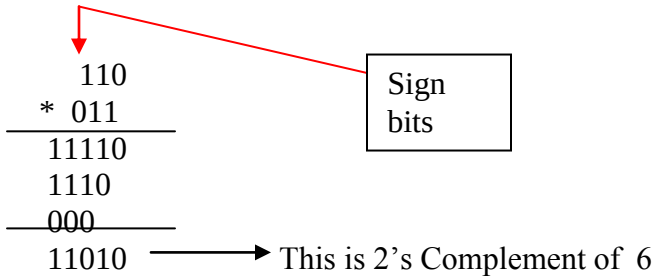
Below are some examples of this method

Example 1

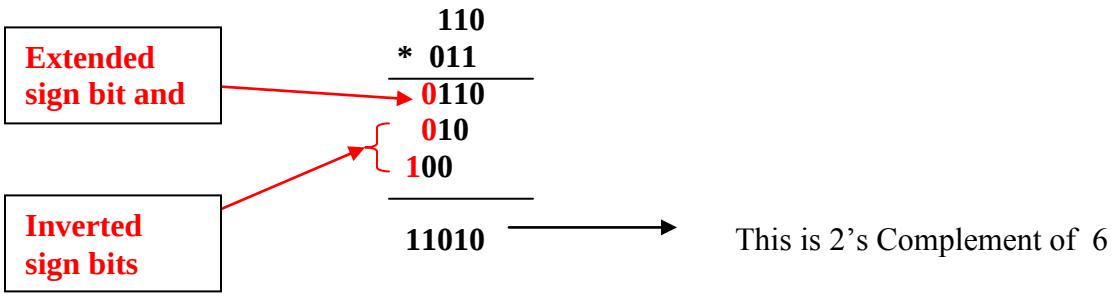
$-2_{10} = 110_2$
$* \quad 3_{10} = 011_2$
$-6 \quad = 11010$ This is 2's Complement of 6

By sign extension method

$-2_{10} = 110_2$
$* \quad 3_{10} = 011_2$
-6

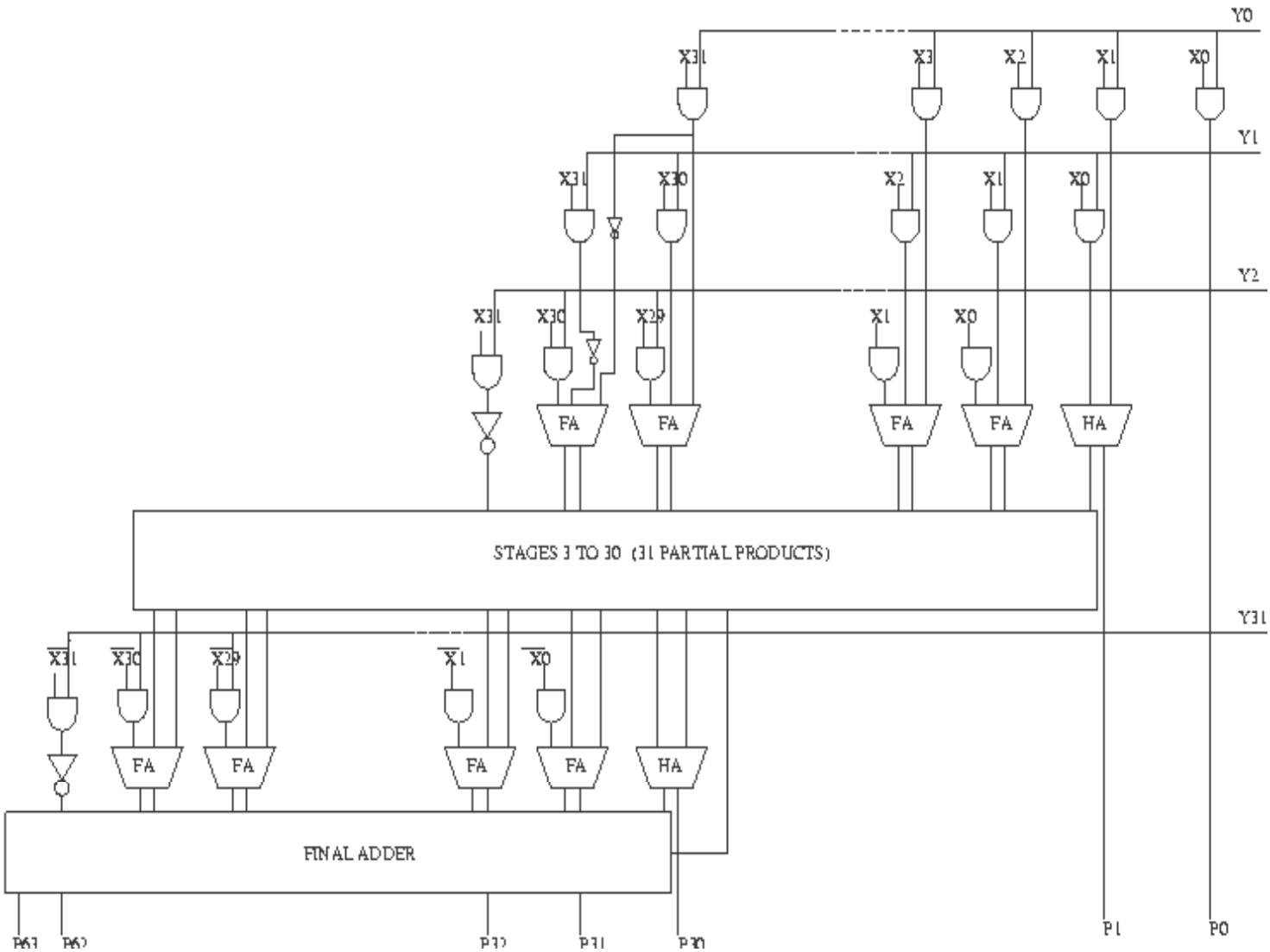


Now, according to the algorithm,



The Diagram below shows the architecture of a 32 bit array adder. (Please note that the design is modified to take care of 2"s complement numbers)

Array Multiplier for a 32 bit number (2"s complement numbers)



Booth Multipliers

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

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

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

Multiplicand	A =	● ● ● ●	
Multiplier	B =	(●●)(●●)	
Partial product bits		● ● ● ●	$(B_1B_0)_2\ A4^0$
		● ● ● ●	$(B_3B_2)_2\ A4^1$
Product	P =	● ● ● ● ● ● ● ●	

Radix-4 multiplication in dot notation.

As shown in the figure above, if multiplication is done in radix 4, in each step, the partial product term $(B_{i+1}B_i)_2\ A$ needs to be formed and added to the cumulative partial product. Whereas in radix-2 multiplication, each row of dots in the partial products matrix represents 0 or a shifted version of A must be included and added.

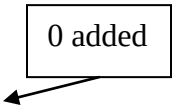
Table 1below is used to convert a binary number to radix-4 number .

Initially, a "0" is placed to the right most bit of the multiplier. Then 3 bits of the multiplicand is recoded according to table below or according to the following equation:

$$\mathbf{Z_i = -2x_{i+1} + x_i + x_{i-1}}$$

Example:

Multiplier is equal to 0 1 0 1 1 10



then a 0 is placed to the right most bit which gives 0 1 0 1 1 10 0

the 3 digits are selected at a time with overlapping left most bit as follows:

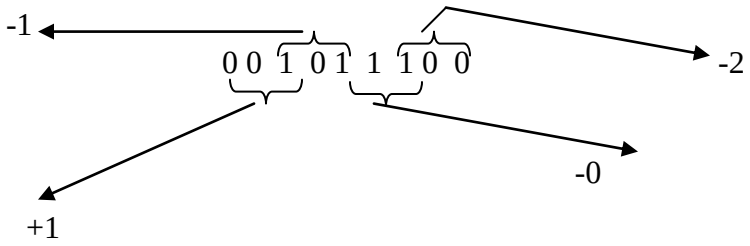


Table .1 Radix-4 Booth recoding

X_{i+1}	X	X_{i-1}	$Z_{i/2}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

For example, an unsigned number can be converted into a signed-digit number radix 4:

$(10\ 01\ 11\ 01\ 10\ 10\ 11\ 10)_2 = (-2\ \ 2\ \ -1\ \ 2\ \ -1\ \ -1\ \ 0\ \ -2)_4$

The Multiplier bit-pair recoding is shown in Table .2

Table Multiplier recoding

0	0	0	+0*multiplicand
0	0	1	+1*multiplicand

0	1	0	+1*multiplicand
0	1	1	+2*multiplicand
1	0	0	-2*multiplicand
1	0	1	-1*multiplicand
1	1	0	-1*multiplicand
1	1	1	-0*multiplicand

Here –2*multiplicand is actually the 2s complement of the multiplicand with an equivalent left shift of one bit position. Also, +2 *multiplicand is the multiplicand shifted left one bit position which is equivalent to multiplying by 2.

To enter $\pm$2*multiplicand into the adder, an (n+1)-bit adder is required. In this case, the multiplicand is offset one bit to the left to enter into the adder while for the low-order multiplicand position a 0 is added. Each time the partial product is shifted two bit positions to the right and the sign is extended to the left.

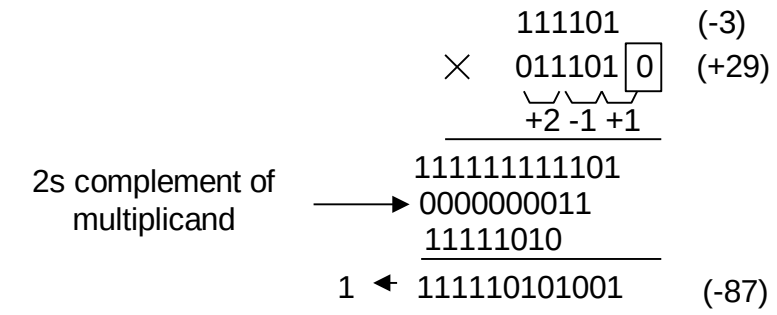
During each add-shift cycle, different versions of the multiplicand are added to the new partial product depends on the equation derived from the bit-pair recoding table above.

Let's see some examples:

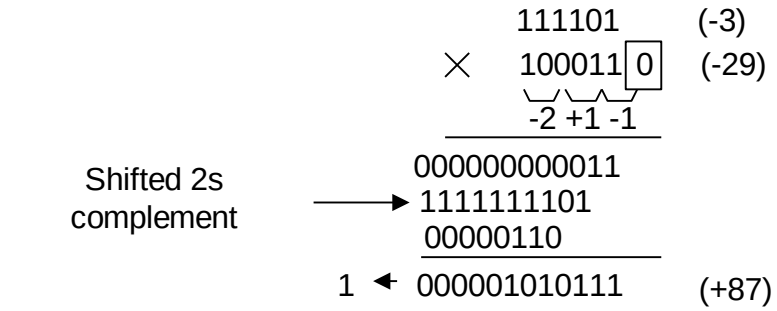
Example 1:

```
          000011        (+3)
      ×   011101 0      (+29)
            +2 -1 +1
  ------------------
  000000000011
  1111111101
  00000110
  ------------------
1 ◄ 000001010111        (+87)
```

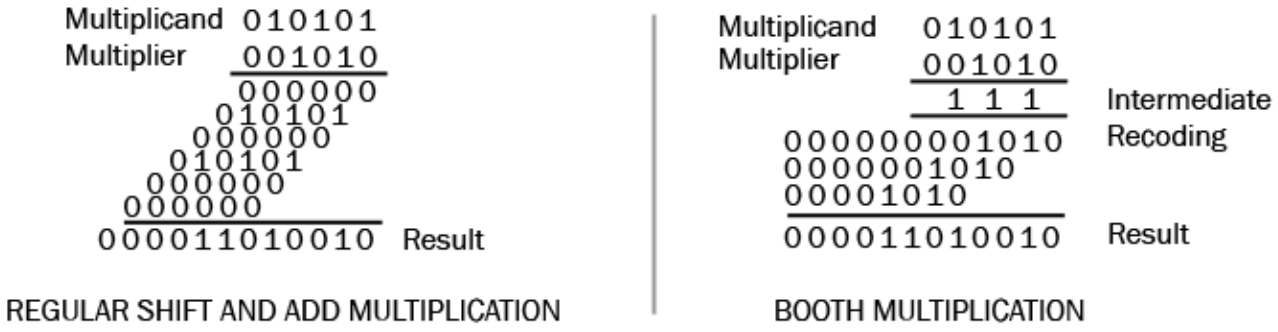
Example 2:



Example 3:



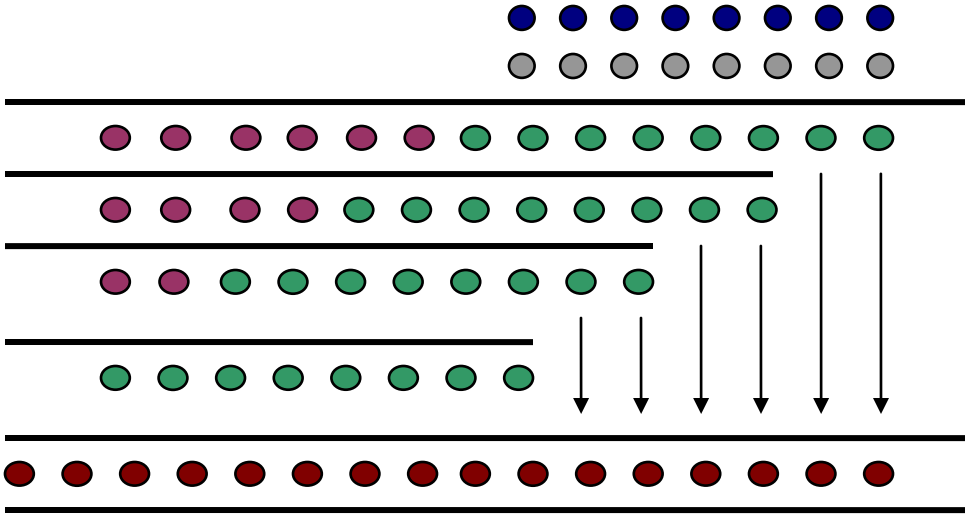
Comparison of Booth and shift and add methods



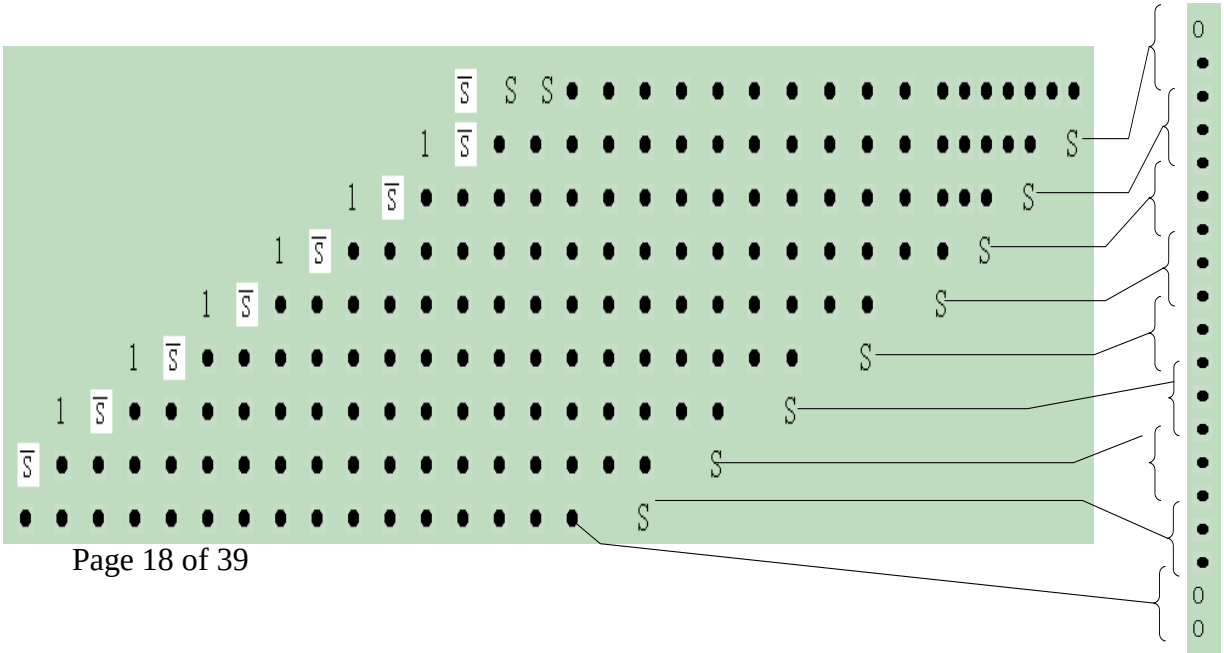
Hardware implementation of Booth

Once the partial products are generated then the addition process is very similar to the array multiplier. Usually carry save adders are used with the final sum added using a CRA.

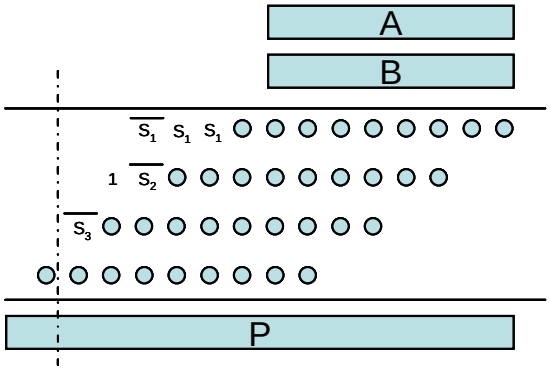
Since the Booth Method applies to 2's complement arithmetic, care must be taken to insure sign extensions are in place as shown in red dots in the following diagram.



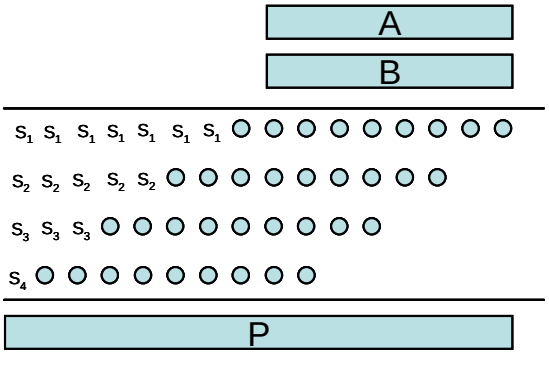
Several techniques exist that reduces this task with ready made templates.
Once the table of the partial products are drawn, all the rows of the partial products have to be arithmetically extended to 2*N, where N is the length of the multiplicand. This is necessary to obtain correct results but it increases the capacitive load, the area and the computational time. Instead the template above can be used (Copied from book: Advanced Computer Arithmetic Design, by M.J. Flynn, S F. Oberman, Wiley) to reduce the calculation. In the above template, there are 16 bit numbers. And the 17th bit is the sign bit. Also, the partial products on each row are entered as 1'complement numbers. If 2'complement numbers are used then the S entries



on the right side can be removed. Please note that the S bit is the sign bit of the booth encoding of that row)



Sign template



Sign extension

Example of using the template:

Let us multiply 25 * -35.

sign bit

A= +25 00011001
B= -35 11011101

Now decode the multiplier

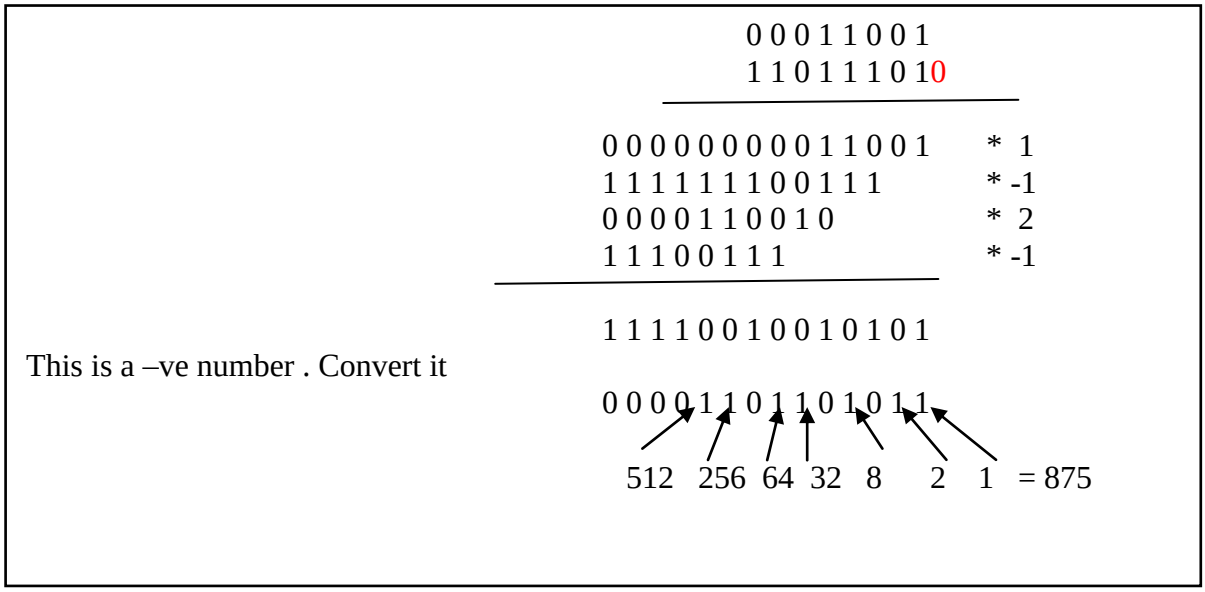
2 1

1 1 0 1 1 1 0 1 0

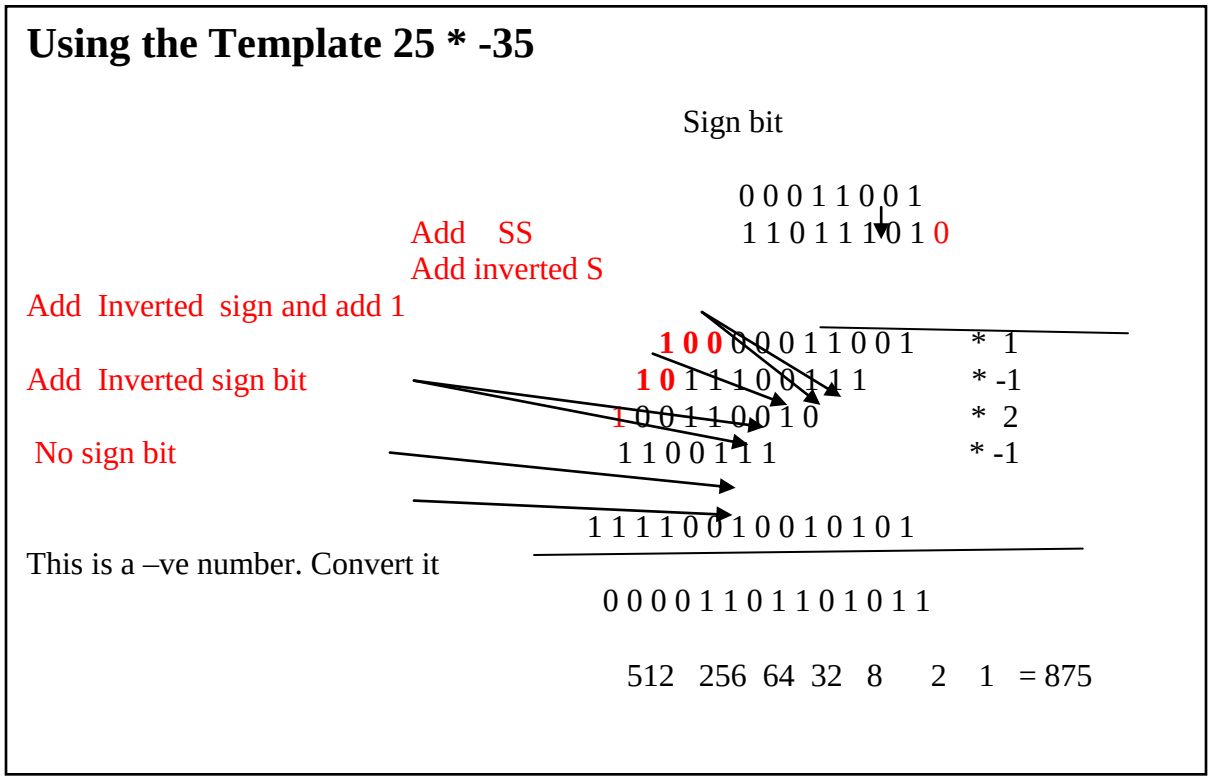
-1 -1

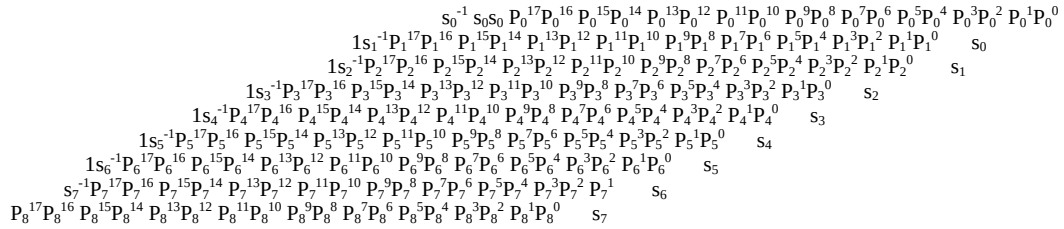
Check these values

B= $-1 * 4^3 + 2* 4^2 -1 * 4^1 + 1 * 4^0 = 35$



Now in order to reduce computation and extra computing units, all the capacitances use the provided template as below





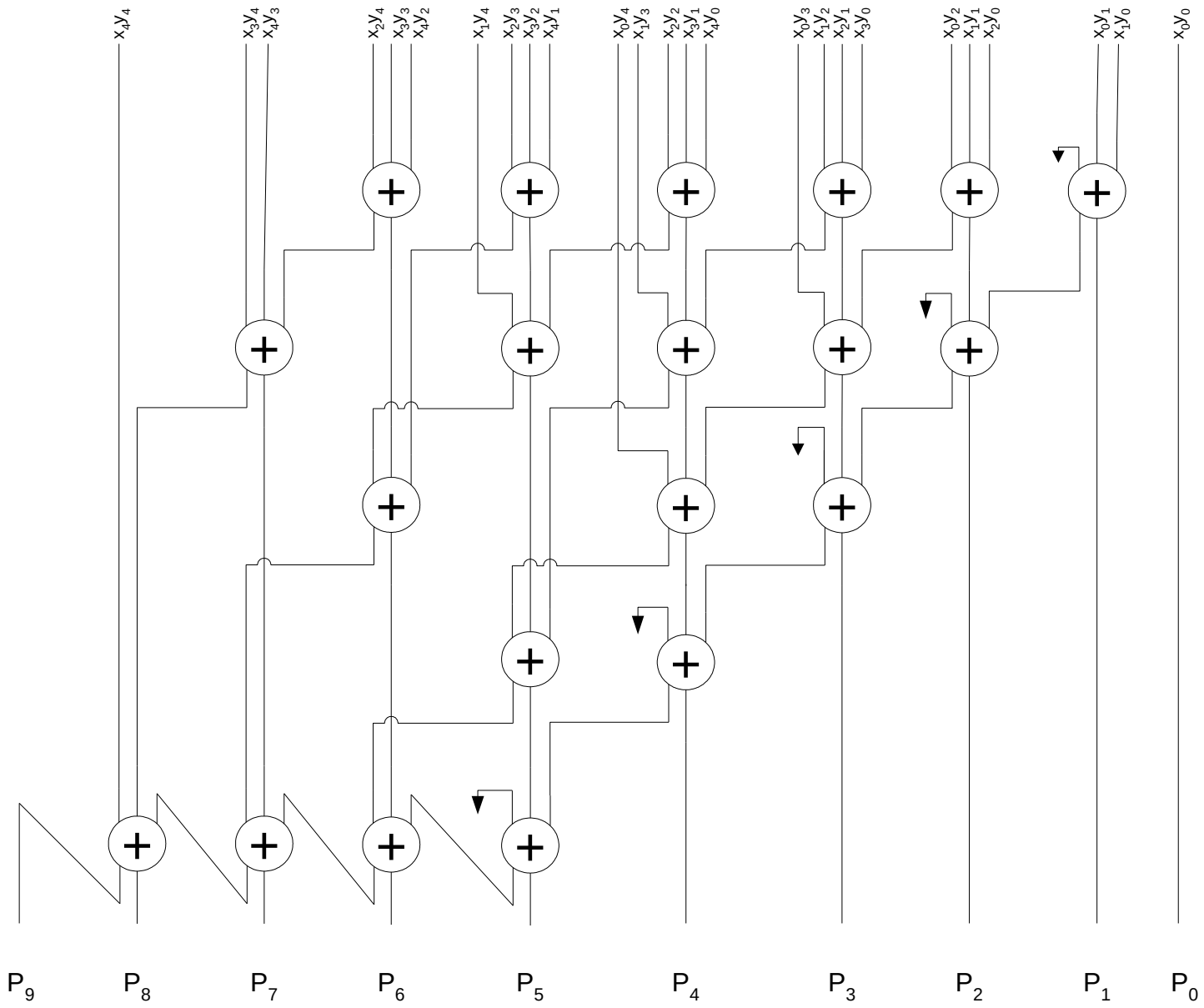
16 x 16 multiplier array with Booth encoding and sign-generation

A general example of 16x 16 bit multiplier using the given template is shown above.

Optimized Wallace Tree Multiplier

Several popular and well-known schemes, with the objective of improving the speed of the parallel multiplier, have been developed in past. Wallace introduced a very important iterative realization of parallel multiplier. This advantage becomes more pronounced for multipliers of bigger than 16 bits.

In Wallace tree architecture, all the bits of all of the partial products in each column are added together by a set of counters in parallel without propagating any carries. Another set of counters then reduces this new matrix and so on, until a two-row matrix is generated. The most common counter used is the 3:2 counter which is a Full Adder.. The final results are added using usually carry propagate adder. The advantage of Wallace tree is speed because the addition of partial products is now O (logN). A block diagram of 4 bit Wallace Tree multiplier is shown in below. As seen from the block diagram partial products are added in Wallace tree block. The result of these additions is the final product bits and sum and carry bits which are added in the final fast adder (CRA).



Since Wallace Tree is a summation method, it can be used in conjunction with array multiplier of any kind including Booth array. The diagram below shows the implementation of 8 bit squarer using the Wallace tree for compressing the addition process.

									n7	n6	n5	n4	n3	n2	n1	n0
									n7	n6	n5	n4	n3	n2	n1	n0
								n7n0	n6n0	n5n0	n4n0	n3n0	n2n0	n1n0	n0n0
							n7n1	n6n1	n5n1	n4n1	n3n1	n2n1	n1n1	n0n1	
						n7n2	n6n2	n5n2	n4n2	n3n2	n2n2	n1n2	n0n2		
					n7n3	n6n3	n5n3	n4n3	n3n3	n2n3	n1n3	n0n3			
				n7n4	n6n4	n5n4	n4n4	n3n4	n2n4	n1n	n0n4				
			n7n5	n6n5	n5n5	n4n5	n3n5	n2n5	n1n5	n0n5					
		n7n6	n6n6	n5n6	n4n6	n3n6	n2n6	n1n6	n0n6						
	n7n7	n6n7	n5n7	n4n7	n3n7	n2n7	n1n7	n0n7							
C_{out}	n7n6	n7n5	n7n4	n7n3	n7n2	n7n1	n7n0	n6n0	n5n0	n4n0	n3n0	n2n0	n1n0	0	n0
	n7		n6n5	n6n4	n6n3	n6n2	n6n1	n5n1	n4n1	n3n1	n2n1		n1		
			n6		n5n4	n5n3	n5n2	n4n2	n3n2		n2				
					n5		n4n3		n3						
							n4								

Figure 4. Operation of 8 bit square

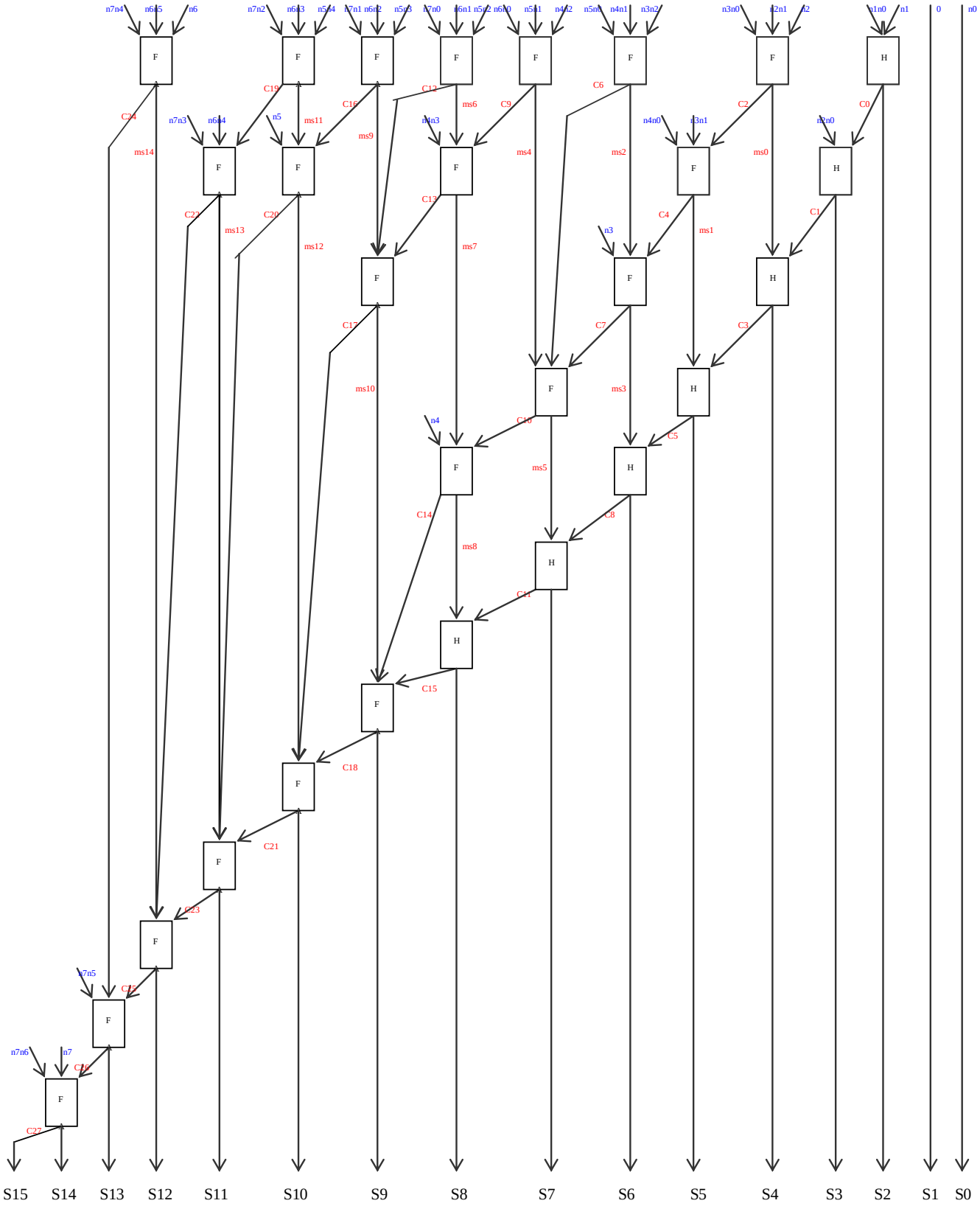
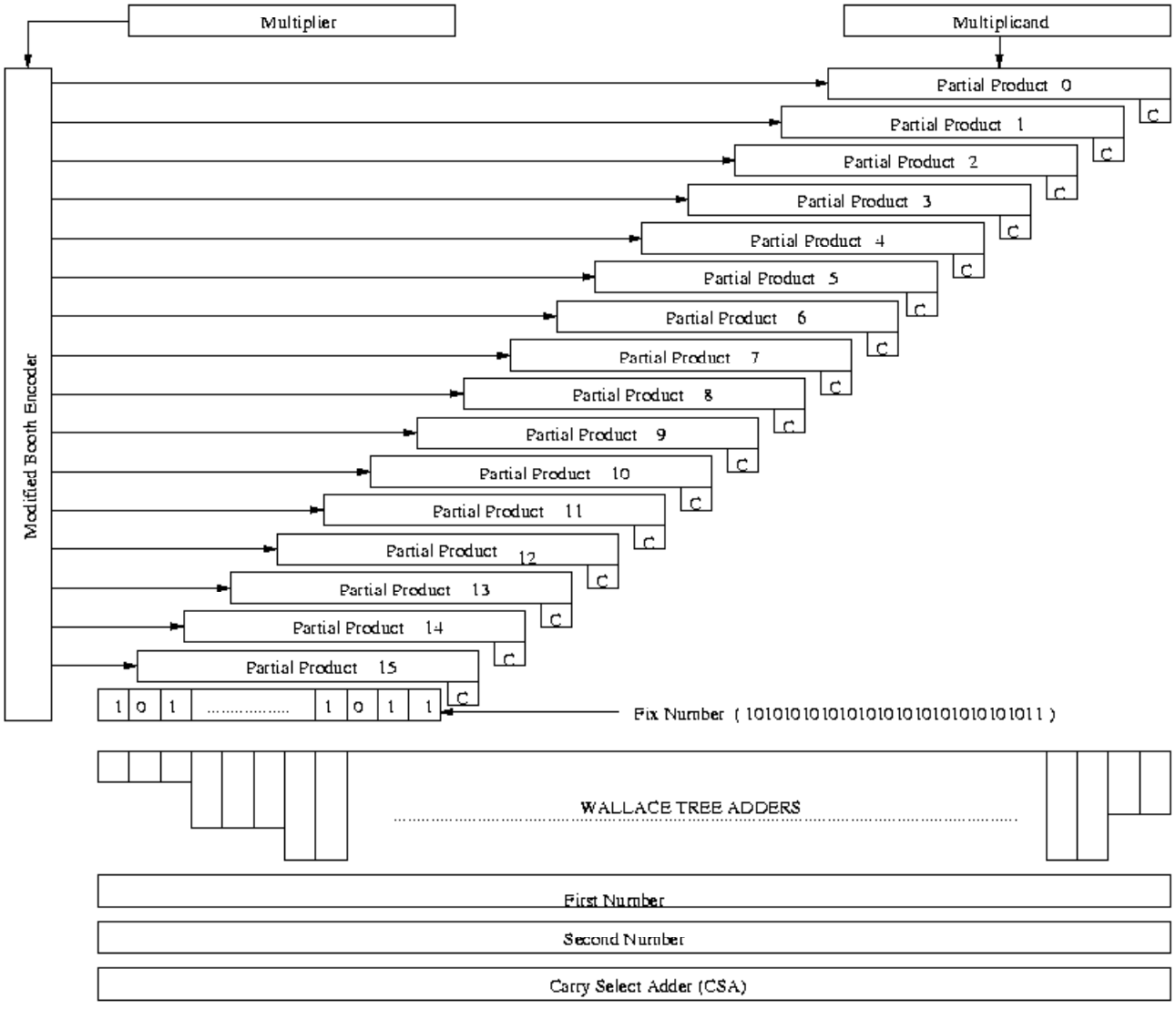


Figure 5. Wallace Tree structure of 8 bit square

32 bit multiplication using Booth and Wallace tree.



Summary

In this section performance measures of multipliers discussed so far are summarized and compared. These results were obtained after synthesizing individual architectures targeting Xilinx FPGA 4052XL-1HQ240C. All comparisons are based on the synthesis reports keeping one common base for comparison. We summarize Area (Total number of CLBs required), Delay and Power Consumption and also calculate Delay·Power (DP), Area·Power (AP), Area·Speed (AT) and Area·Speed^2 (AT^2) product.

From the Table we can see that delay of Wallace tree multiplier and Combined Booth-Wallace tree multiplier is almost the same and is the least. Hence they are fastest among five multipliers. DP product is also the least for the above two multiplier and are a good choice for this performance measure. Serial Parallel multiplier is a best choice when speed is not important but reduced area and power consumption is of more interest and also for AP and AT product Serial Parallel multiplier is a good choice. However, one of the most important performance parameter is AT^2. From the table we see that Modified Booth-Wallace Tree multiplier is the best choice as far as AT^2 is concerned. The Serial Parallel multiplier which is a good choice for AP and AT product has worst performance for AT^2.

	Array Multiplier	Modified Booth Multiplier	Wallace Tree Multiplier	Modified Booth -Wallace Tree Multiplier	Twin Pipe Serial-Parallel Multiplier
Area – Total CLB's (#)	1165	1292	1659	1239	133
Maximum Delay D (ns)	187.87	139.41	101.14	101.43	22.58 (722.56)*
Power(mW) (at highest speed)	16.6506 (at188 ns)	23.136 (at 140ns)	30.95 (101.14ns)	30.862 (at 101.43ns)	2.089 (at 722.56ns)
Power P (mW) when delay = 722.56ns	4.329	4.638	4.332	4.332	2.089
Delay ·Power Product (DP) (ns mW)	813.28	622.30	438.138	439.39	1509.42
Area·Power Product (AP) (# mW)	5043.28	5767.23	7186.788	5367.35	277.837
Area·Delay Product (AD) (# ns)	218.868 x 10^3	180.118 x 10^3	167.791 x 10^3	125.671 x 10^3	96.101 x 10^3 *
Area·Delay^2 Product(AD^2) (# ns^2)	41.119 x 10^6	25.110 x 10^6	16.970 x 10^6	12.747 x 10^6	69.438 x 10^6 *

Appendix A

Signed Number Multiplication

1. Introduction

Direct two's complement array multiplication can perform "direct" multiplication of two's complement numbers without requiring the complementing stages, significantly speeds up the multiplication process. This appendix will discuss two direct two's complement multiplication algorithms and their implementation.
These two direct two's complement multiplication algorithms are:
1) Tri-section modified Pezaris two's complement multiplication
2) Baugh-Wooley two's complement multiplication
These two algorithms are generally used in systems where the operands are less than 16-bit. They are relatively simpler than Booth multiplier whose structure is based on recoding the 2's complement operand in order to reduce the number of partial products to be added.

2. Tri-section modified Pezaris two's complement multiplier:

In 2's complement number representation, the most significant bit (MSB) is weighted negatively. In realizing such a system, Pezaris generalizes the full adders into four types. In type 0, which represents a normal adder, all three inputs x, y, z are weighted positively and the result lies in the range {0,3}. This result is represented by a 2-bit binary number C S where C and S are also weighted positively. In the other three types there are some signals, indicated by the dots, that are weighted negatively.

Listed below are four arithmetic equations that describe the input/output relationships of the four types of generalized full adders.

Type 0: $C2^1 + S2^0 = X2^0 + Y2^0 + Z2^0$
Type 1: $C2^1 + (-S)2^0 = X2^0 + Y2^0 + (-Z)2^0$
Type 2: $(-C)2^1 + S2^0 = (-X)2^0 + (-Y)2^0 + Z2^0$
Type 3: $(-C)2^1 + (-S)2^0 = (-X)2^0 + (-Y)2^0 + (-Z)2^0$

These four arithmetic equations lead to the truth-table descriptions of the four generalized full adders given in the following table.

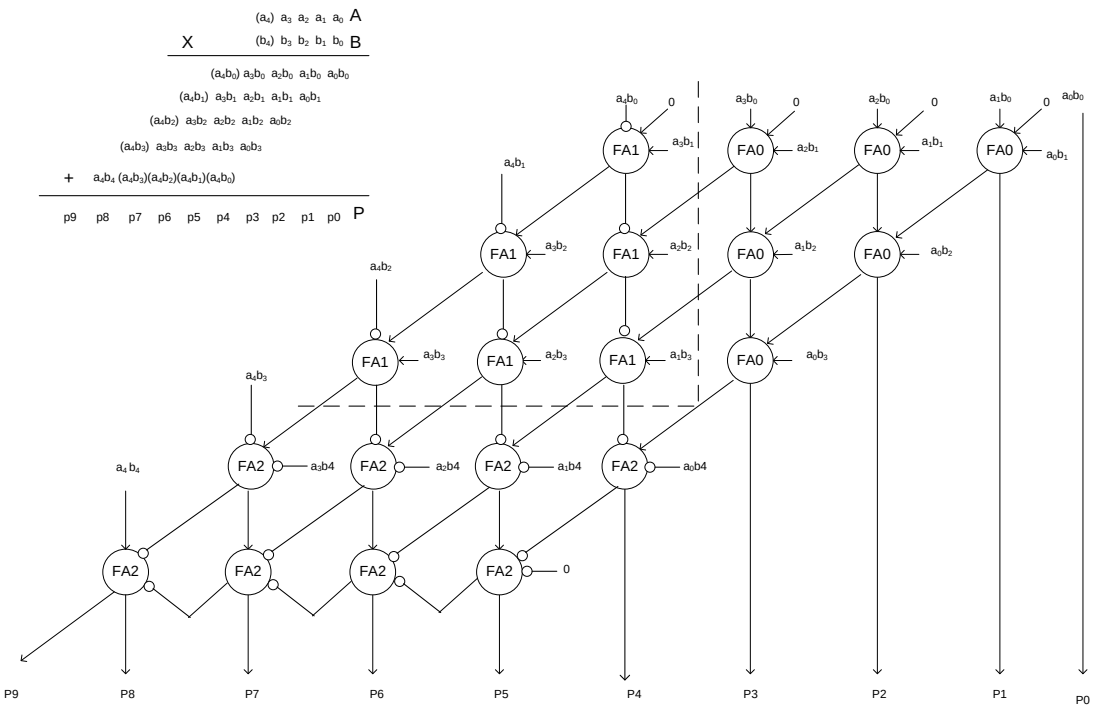
Table: Truth Table Describing the Four Types of Generalized Full Adders

Full Adder	**Weighted Inputs**			**Weighted Outputs**	
Type 0	**$X2^0$**	**$Y2^0$**	**$Z2^0$**	**$C2^1$**	**$S2^0$**
Type 3	**- $X2^0$**	**- $Y2^0$**	**- $Z2^0$**	**- $C2^1$**	**- $S2^0$**
	0	0	0	0	0
	0	0	1	0	1

Truth Table	0	1	0	0	1
	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
	1	1	1	1	1
Type 1	**$X2^0$**	**$Y2^0$**	**$-Z2^0$**	**$C2^1$**	**$-S2^0$**
Type 2	**$-X2^0$**	**$-Y2^0$**	**$Z2^0$**	**$-C2^1$**	**$S2^0$**
Truth Table	0	0	0	0	0
	0	0	1	0	1
	0	1	0	1	1
	0	1	1	0	0
	1	0	0	1	1
	1	0	1	0	0
	1	1	0	1	0
	1	1	1	1	1

One can easily derive the Boolean equations governing the four types of full adders from the table entries.
Type 0 or Type 3:
S = X'Y'Z + X'YZ' + XY'Z' + XYZ
C = XY + YZ +ZX
Type 1 or Type 2:
S = X'Y'Z + X'YZ' + XY'Z' + XYZ
C = XY + YZ' +Z'X

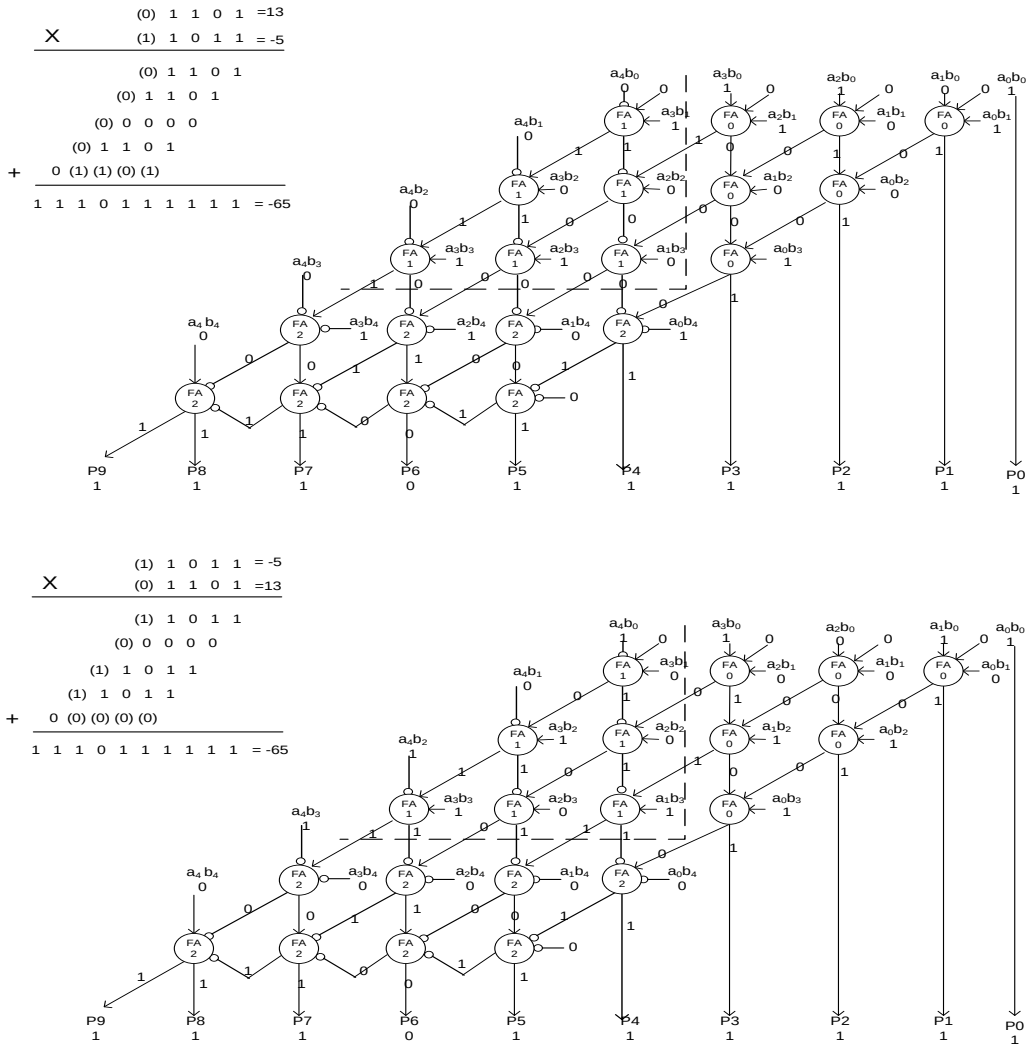
Pezaris two's complement multiplier use mixture types of full adders.
The schematic circuit diagram of a 5-by-5 Pezaris array multiplier is shown below:



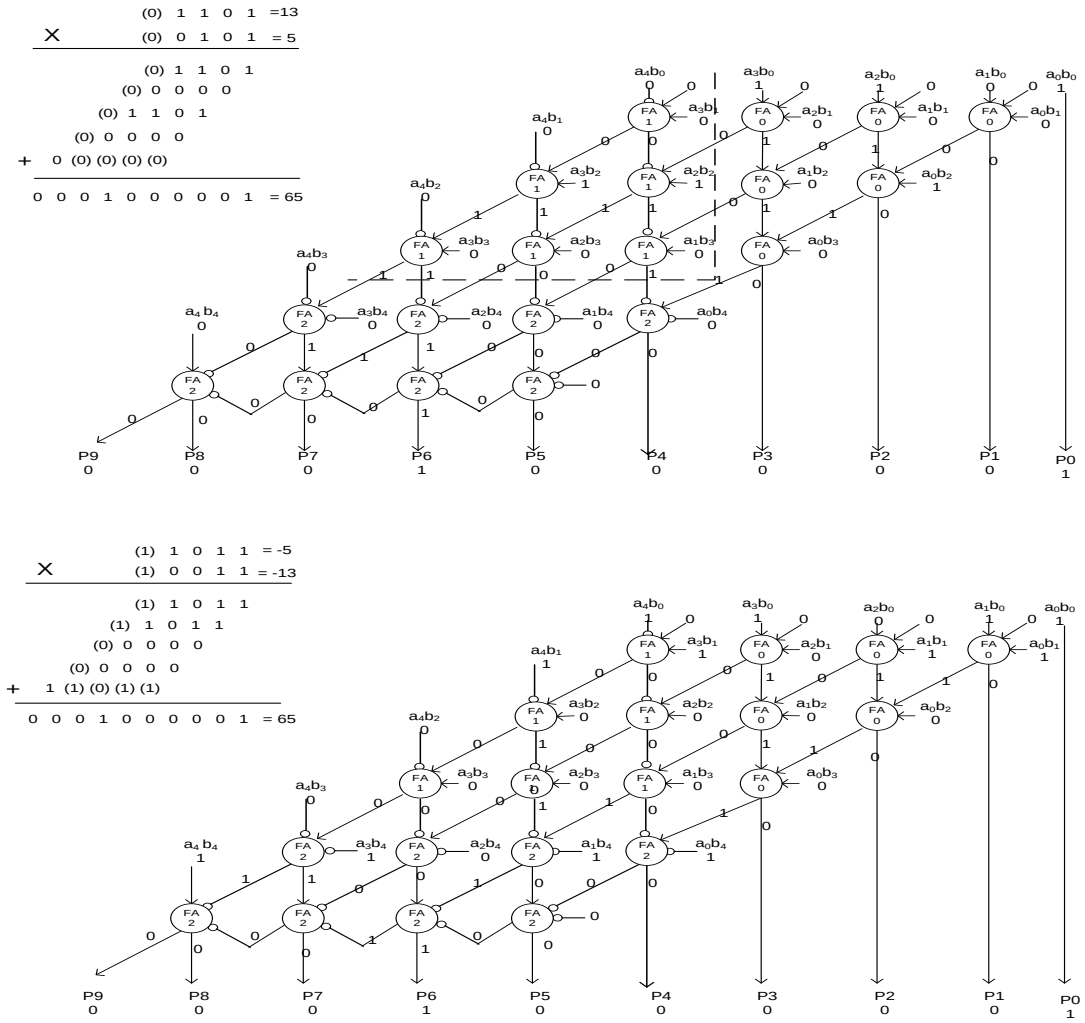
The schematic logic circuit diagram of a 5-by-5 Tri-section modified Pezaris two's complement array multiplier

The examples of 5-by-5 Pezaris are shown below:

multiplicand	**multiplier**
positive	negative
negative	positive
positive	positive
negative	negative



Example of Tri-section modified Pezaris Two's Complement Multiplication

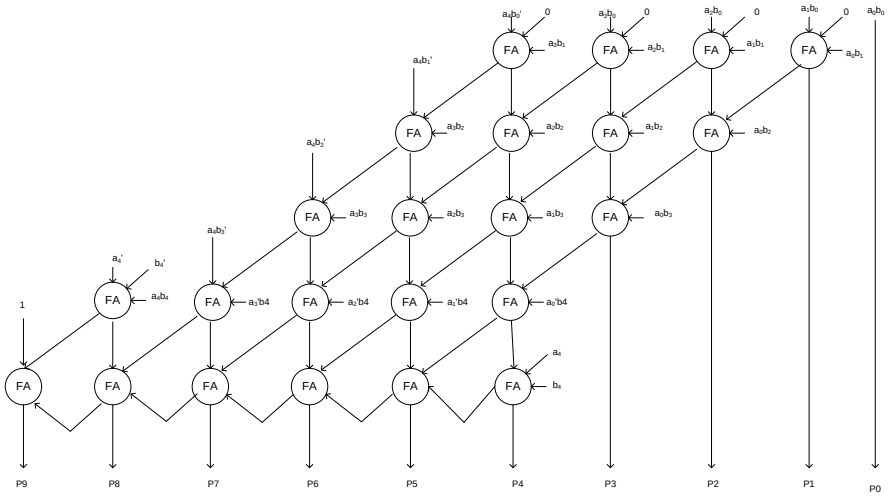


Example of Tri-section modified Pezaris Two's Complement Multiplication

3. Baugh-Wooley two's complement multiplier:

Baugh and Wooley have proposed an algorithm for direct two's complement array multiplication. The principal advantage of their algorithm is that the signs of all summands are positive, thus allowing the array to be constructed entirely with the conventional Type 0 full adders. This uniform structure is very attractive for VLSI.

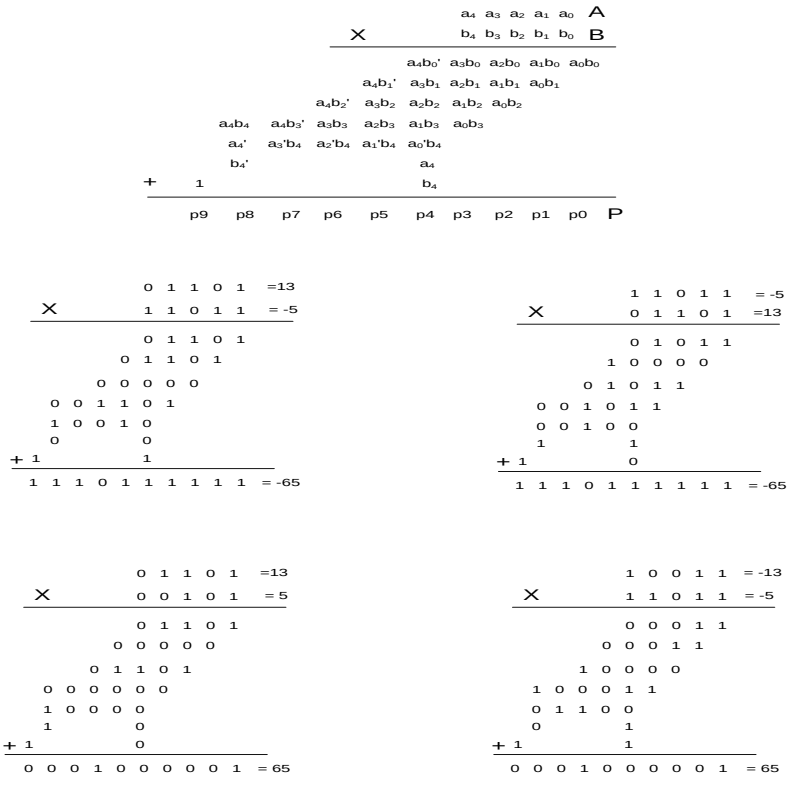
The schematic circuit diagram of a 5-by-5 Baugh-Wooley array multiplier is shown below:



The schematic logic circuit diagram of a 5-by-5 Baugh-Wooley two's complement array multiplier

The examples of 5-by-5 Baugh-Wooley are shown below:

multiplicand	**multiplier**
positive	negative
negative	positive
positive	positive
negative	negative



Example of Baugh-Wooley Two's Complement Multiplication

4. Comparison

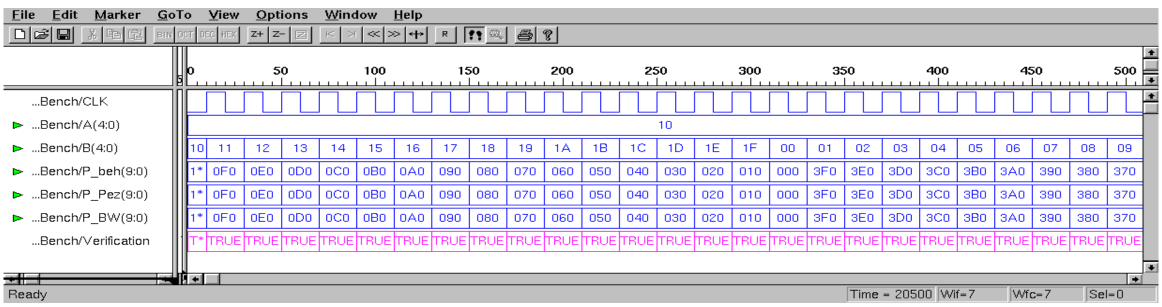
Table: Direct two's complement multiplication

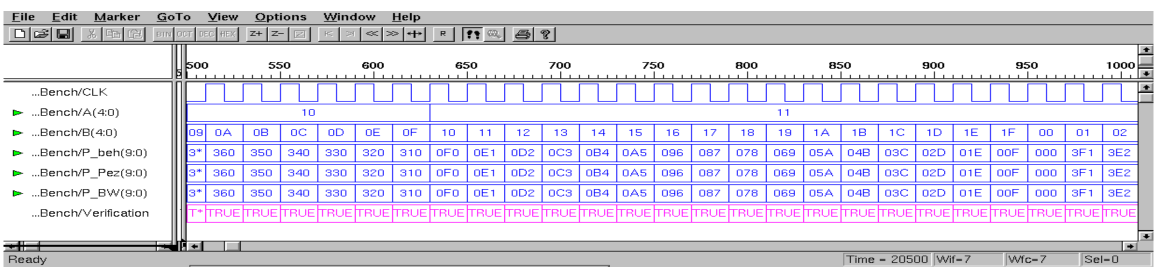
n * n two's Complement Array Multiplier			
		Tri-section Pezaris	Baugh-Wooley
	Advantage	Regular format array	Irregular format array, two more rows
	Disadvantage	Three type full adder uesd	Only one type full adder uesd
Full Adder Used	Type 0	$(n^2 - 3n + 2) / 2$	$n^2 - n + 3$
	Type 1	$(n^2 - 3n + 2) / 2$	0
	Type 2	$2n - 1$	0
	Type 3	0	0
	Total	$n^2 - n$	$n^2 - n + 3$
Total time delay (Multiply time)		$4n\Delta - 2\Delta$	$4n\Delta$

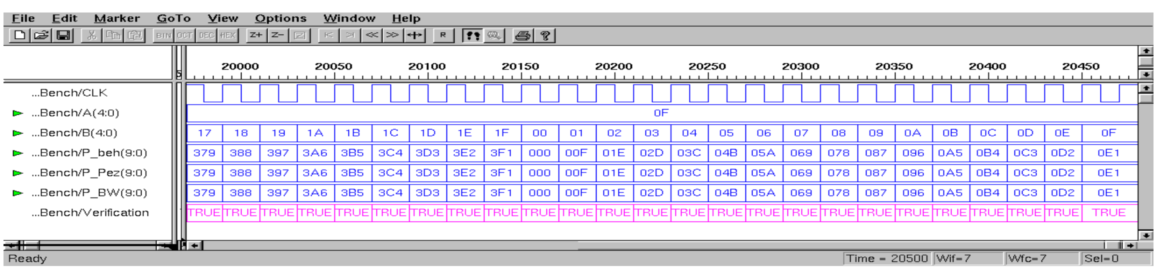
* Δ is the unit gate delay.

5. VHDL coding:

As an example a 5-bit two's complement multiplication of Tri-section modified Pezaris and Baugh-Wooley are implemented by VHDL code and part of the simulation result are shown below:







6. FPGA Implementation:

Implement Multipliers in Xilinx Virtex II FPGAs.
Then indicate the critical path, compare the performance, area and power consumption.

References:

[1] Kai Hwang "Computer Arithmetic: Principles, Architecture, and Design" John Wiley & Sons 1979

[2] S. D. Pezaris "A 40-ns 17-Bit by 17-Bit Array Multiplier", IEEE Trans. on Computers, pp. 442-447,.Abr. 1971

[3] C. Baugh y A. Wooley "A Two's Complement Parallel Array Multiplication Algorithm". IEEE Trans.on Computer, Vol.C-22, Nº12. Dic.1973.

Appendix B

Examples of signed multiplication (When multiplier operand is positive)

Example. 1

$$\begin{array}{rl} -100 & -100_{10}=10011100_2 \\ \times\ 4 & 4_{10} = 0100_2 \\ \hline -400 & \end{array}$$

By Sign Extension method,

$$\begin{array}{l} 10011100 \\ \times\ 0100 \\ \hline 00000000000 \\ 0000000000 \\ 110011100 \\ 00000000 \\ \hline 11001110000 \end{array} \longrightarrow -400$$

According to the extend and invert algorithm,

```
   10011100
 X     0100
 ----------
  100000000
   10000000
  00011100
 10000000
 ----------
 11001110000   ------------>  Ans is -400
```

Ex 2

```
     -5          -5(10) = 1011(2)
X     4           4(10) = 0100(2)
 ---------
    -20
```

By Sign Extension method,

```
     1011
X    0100
 ----------
  0000000
  000000
  11011
  0000
 ----------
 1101100   ------------>  2's complement of -20
```

According to the algorithm of extend and invert method,

```
    1011
X   0100
   10000
    1000
   0011
  1000
 1101100   ——————→  –20 in 2's complement
```

Ex 3

```
   -4          -4₁₀ = 1100₂
X   3          3₁₀  = 0011₂
  -12
```

By Sign Extension method,

```
     1100
X    0011
   ------
    111100
     11100
      0000
   0000
  1110100   ——————→  -12 in 2's complement
```

According to the sign extend and invert algorithm,

```
     1100
X    0011
   ------
    01100
     0100
    1000
  1000
 1110100   ——————→  -12 in 2's complement
```

Ex 4

-12 $\quad -12_{10} = 10100_2$

X 12 $\quad 12_{10} = 01100_2$

-144

By Sign Extension method,

```
    10100
  X 01100
 ---------
  000000000
  00000000
  1110100
  110100
  00000
 ---------
  101110000  ------------------>  -144
```

According to the sign extend and invert algorithm,

```
    10100
  X 01100
 ---------

   100000
   10000
  00100
 00100
 10000
 ---------
 101110000  ------------------>  - 144
```

Examples of B00th multiplication

Example

Using Booth algorithm multiply A and B.
A= 20
B=30

A= 0010100 }
B= 0011110 } Please note that both numbers are extended to cover 2A or 2B and the sign bit (whichever is larger).

A * B = A= 0 0 1 0 1 0 0

-0

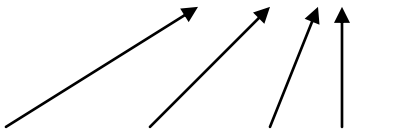
B= 0 0 1 1 1 1 0 0

+2 -2

2A = 40 = 00101000
-2A = 11011000

Now performing the addition we have

1 1 1 1 1 1 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0
---
0 0 0 1 0 0 1 0 1 1 0 0 0



512 + 64 + 16 + 8 = $(600)_{10}$

Now let us try

B * A = B= 0 0 1 1 1 1 0

+ 1

A= 0 1 0 1 0 0 0

+1 +0

P

Now performing the addition we have