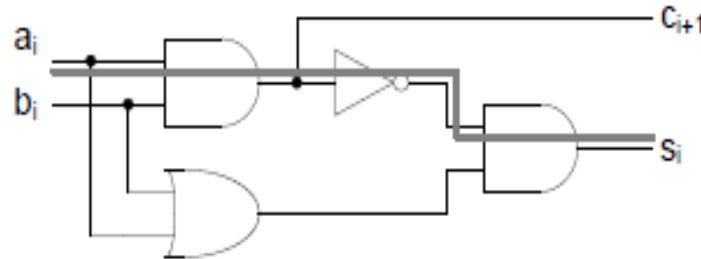


UNIT-IV

ADDERS:

Binary Adder Notations and Operations:

As mentioned previously, adders in VLSI digital systems use binary notation. In that case, add is done bit by bit using Boolean equations.



1-bit Half Adder.

Consider a simple binary add with two n-bit inputs A;B and a one-bit carry-in c_{in} along with n-bit output S.

$$S = A + B + C_{in}$$

Where $A = a_{n-1}, a_{n-2}, \dots, a_0$; $B = b_{n-1}, b_{n-2}, \dots, b_0$.

The + in the above equation is the regular and operation. However, in the binary world, only Boolean algebra works. For add related operations, AND, OR and Exclusive-OR (XOR) are required. In the following documentation, a dot between two variables (each with single bit), e.g. $a \cdot b$ denotes 'a AND b'. Similarly, $a + b$ denotes 'a OR b' and $a \oplus b$ denotes 'a XOR b'. Considering the situation of adding two bits, the sum s and carry c can be expressed using Boolean operations mentioned above.

$$S_i = a_i \oplus b_i$$

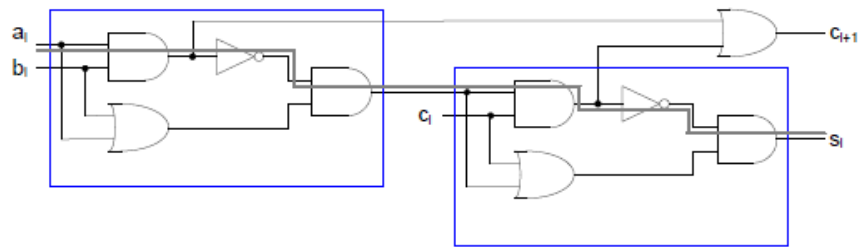
$$C_{i+1} = a_i \cdot b_i$$

The Equation of C_{i+1} can be implemented as shown in Fig.2.1. In the figure, there is a Half adder, which takes only 2 input bits. The solid line highlights the

critical path, which indicates the longest path from the input to the output. Equation of c_{i+1} can be extended to perform full add operation, where there is a carry input.

$$S_i = a_i \oplus b_i \oplus c_i$$

$$C_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$



1-bit Full Adder.

A Full adder can be built based on Equation above. The block diagram of a 1-bit full adder is shown in Fig.2.2. The full adder is composed of 2 half adders and an OR gate for computing carry-out. Using Boolean algebra, the equivalence can be easily proven. To help the computation of the carry for each bit, two binary literals are introduced. They are called carry generate and carry propagate, denoted by g_i and p_i . Another literal called temporary sum t_i is employed as well. There is relation between the inputs and these literals.

$$G_i = a_i \cdot b_i$$

$$P_i = a_i \oplus b_i$$

$$T_i = a_i \oplus b_i$$

Where i is an integer and $0 \leq i < n$.

With the help of the literals above, output carry and sum at each bit can be written as:

$$C_{i+1} = g_i + p_i \cdot c_i$$

$$S_i = t_i \oplus c_i$$

In some literatures, carry-propagate p_i can be replaced with temporary sum t_i in order to save the number of logic gates. Here these two terms are separated in order to clarify the concepts. For example, for Ling adders, only p_i is used as carry-propagate.

The single bit carry generate/propagate can be extended to group version G and P. The following equations show the inherent relations.

$$G_{i:k} = G_{i:j} + P_{i:j} \cdot G_{j-1:k}$$

$$P_{i:k} = P_{i:j} \cdot P_{j-1:k}$$

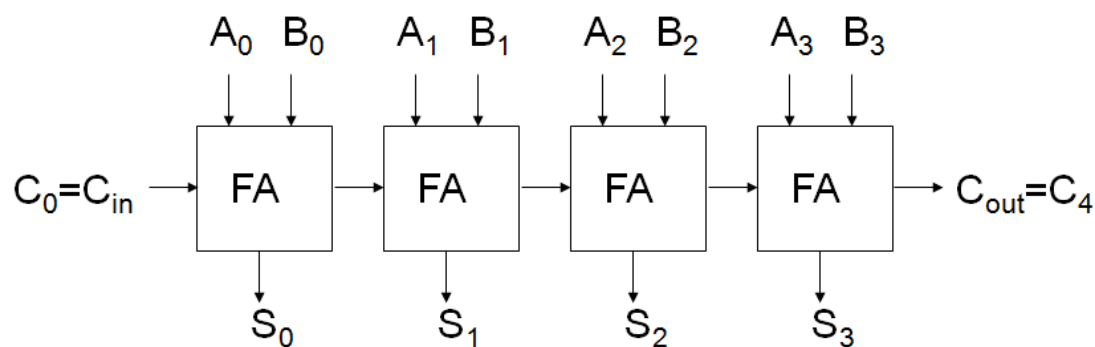
Where $i:k$ denotes the group term from i through k .

Using group carry generate/propagate, carry can be expressed as expressed in the following equation.

$$C_{i+1} = G_{i:j} + P_{i:j} \cdot C_j$$

Ripple carry adder

Ripple carry adder is an n -bit adder built from full adders. Fig 2.1 shows a 4-bit ripple carry adder. One full adder is responsible for the addition of two binary digits at any stage of the ripple carry. The carryout of one stage is fed directly to the carry-in of the next stage. Even though this is a simple adder and can be used to add unrestricted bit length numbers, it is however not very efficient when large bit numbers are used.



4-b Ripple Carry Adder

One of the most serious drawbacks of this adder is that the delay increases linearly with the bit length. The worst-case delay of the RCA is when a carry signal transition ripples through all stages of adder chain from the least significant bit to the most significant bit, which is approximated by:

$$T = (n-1) t_c + t_s$$

Delay :

The latency of a 4-bit ripple carry adder can be derived by considering the worst-case signal propagation path. We can thus write the following expressions:

$$TRCA_{4bit} = TFA(A_0, B_0 \rightarrow C_0) + TFA(C_{in} \rightarrow C_1) + TFA(C_{in} \rightarrow C_2) + TFA(C_{in} \rightarrow S_3)$$

And, it is easy to extend to k-bit RCA:

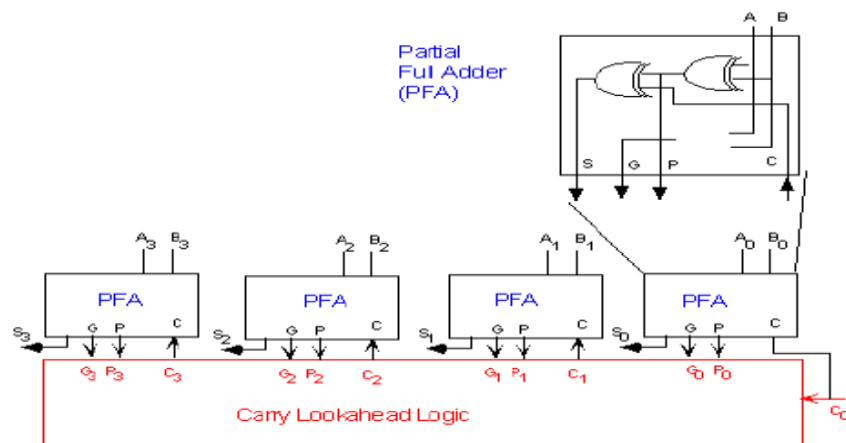
$$TRCA_{kbit} = TFA(A_0, B_0 \rightarrow C_0) + (K-2) * TFA(C_{in} \rightarrow C_i) + TFA(C_{in} \rightarrow S_{k-1}).$$

Drawbacks :

Delay increases linearly with the bit length and Not very efficient when large bit numbers are used.

Carry Look-Ahead Adder

Lookahead carry algorithm speed up the operation to perform addition, because in this algorithm carry for the next stages is calculated in advance based on input signals. In CLA, the carry propagation time is reduced to $O(\log_2(Wd))$ by using a tree like circuit to compute the carry rapidly. Fig. shows the 4-bit Carry Look-Ahead Adder.



4-bit Carry Look Ahead Adder

The CLA exploits the fact that the carry generated by a bit-position depends on the three inputs to that position. If 'X' and 'Y' are two inputs then if X=Y=1, a carry is generated independently of the carry from the previous bit position and if

$X=Y=0$, no carry is generated. Similarly if $X \neq Y$, a carry is generated if and only if the previous bit-position generates a carry. 'C' is initial carry, "S" and "Cout" are output sum and carry respectively, then Boolean expression for calculating next carry and addition is:

$$P_i = X_i \text{ xor } Y_i \text{ -- Carry Propagation}$$

$$G_i = X_i \text{ and } Y_i \text{ -- Carry Generation}$$

$$C_{i+1} = G_i \text{ or } (P_i \text{ and } C_i) \text{ -- Next Carry}$$

$$S_i = X_i \text{ xor } Y_i \text{ xor } C_i \text{ -- Sum Generation}$$

Thus, for 4-bit adder, we can extend the carry, as shown below:

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

As with many design problems in digital logic, we can make tradeoffs between area and performance (delay). In the case of adders, we can create faster (but larger) designs than the RCA. The Carry Look ahead Adder (CLA) is one of these designs (there are others too, but we will only look at the CLA).

Drawbacks :

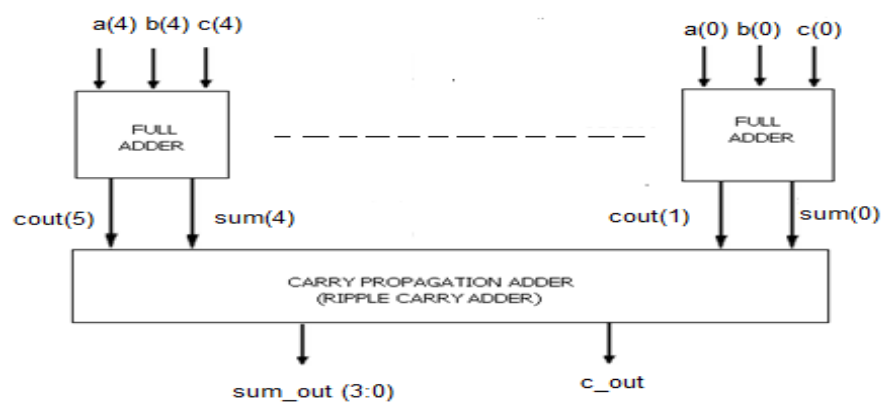
For long bit length, a carry look-ahead adder is not practical, but a hierarchical structure one can improve much. The disadvantage of CLA is that the carry logic block gets very complicated for more than 4-bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4-bits.

Carry Save Adder

The carry-save adder reduces the addition of 3 numbers to the addition of 2 numbers. The propagation delay is 3 gates regardless of the number of bits. The carry-save unit consists of n full adders, each of which computes a single sum and carries bit based solely on the corresponding bits of the three input numbers.

The entire sum can then be computed by shifting the carry sequence left by one place and appending a 0 to the front (most significant bit) of the partial sum sequence and adding this sequence with RCA produces the resulting n+1-bit value.

This process can be continued indefinitely, adding an input for each stage of full adders, without any intermediate carry propagation. These stages can be arranged in a binary tree structure, with cumulative delay logarithmic in the number of inputs to be added, and invariant of the number of bits per input. The main application of carry save algorithm is, well known for multiplier architecture is used for efficient CMOS implementation of much wider variety of algorithms for high speed digital signal processing .CSA applied in the partial product line of array multipliers will speed up the carry propagation in the array.



4-bit Carry Save Adder

Basically, carry save adder is used to compute sum of three or more n-bit binary numbers. Carry save adder is same as a full adder. As shown in the Fig.2.4, here we are computing sum of two 4-bit binary numbers, so we take 4 full adders at first stage. Carry save unit consists of 4 full adders, each of which computes single sum and carry bit based only on the corresponding bits of the two input numbers. Let X and Y are two 4-bit numbers and produces partial sum and carry as S and C as shown in the below :

$$S_i = X_i \text{ xor } Y_i ; C_i = X_i \text{ and } Y_i$$

The final addition is then computed as:

1. Shifting the carry sequence C left by one place.
2. Placing a 0 to the front (MSB) of the partial sum sequence S.
3. Finally, a ripple carry adder is used to add these two together and computing the resulting sum.

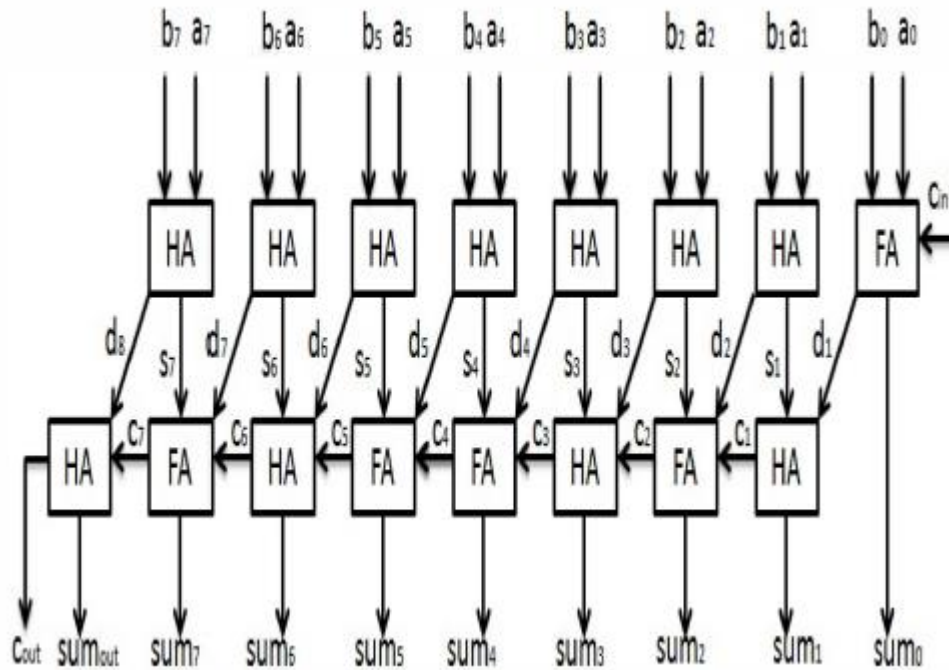
Carry Save Adder Computation :

```

X:    1 0 0 1 1
Y:    1 1 0 0 1
Z:  + 0 1 0 1 1
S:    0 0 0 0 1
C:  + 1 1 0 1 1
SUM:  1 1 0 1 1 1

```

In this design 128 bit carry save adder is used since the output of the multiplier is 128 bits (2N). The carry save adder minimize the addition from 3 numbers to 2 numbers. The propagation delay is 3 gates despite of the number of bits. The carry save adder contains n full adders, computing a single sum and carries bit based mainly on the respective bits of the three input numbers. The entire sum can be calculated by shifting the carry sequence left by one place and then appending a 0 to most significant bit of the partial sum sequence. Now the partial sum sequence is added with ripple carry unit resulting in n + 1 bit value. The ripple carry unit refers to the process where the carryout of one stage is fed directly to the carry in of the next stage. This process is continued without adding any intermediate carry propagation. Since the representation of 128 bit carry save adder is infeasible, hence a typical 8 bit carry save adder is shown in the figure 3. Here we are computing the sum of two 128 bit binary numbers, then 128 half adders at the first stage instead of 128 full adder. Therefore, carry save unit comprises of 128 half adders, each of which computes single sum and carry bit based only on the corresponding bits of the two input numbers.



bit carry save adder

If x and y are supposed to be two 128 bit numbers then it produces the partial products and carry as S and C respectively.

$$S_i = x_i \oplus y_i \quad (4)$$

$$C_i = x_i \& y_i \quad (5)$$

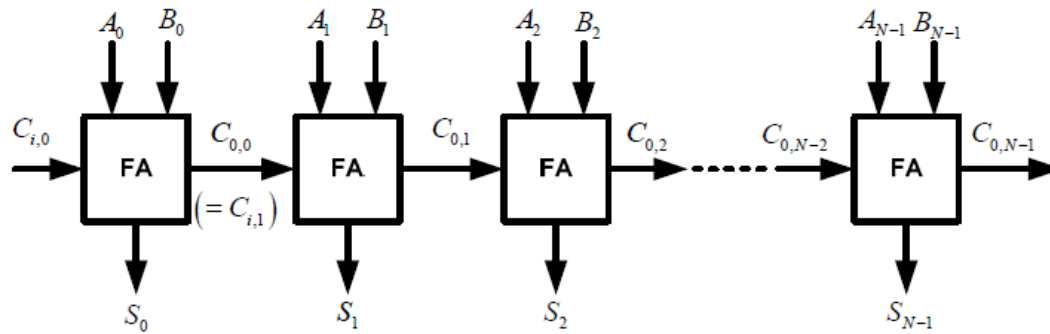
During the addition of two numbers using a half adder, two ripple carry adder is used. This is due the fact that ripple carry adder cannot compute a sum bit without waiting for the previous carry bit to be produced, and hence the delay will be equal to that of n full adders. However a carry-save adder produces all the output values in parallel, resulting in the total computation time less than ripple carry adders. So, Parallel In Parallel Out (PIPO) is used as an accumulator in the final stage.

2.5. Carry Select Adder

A carry-select adder is divided into sectors, each of which – except for the least-significant – performs two additions in parallel, one assuming a carry-in of zero, the other a carry-in of one. A four bit carry select adder generally consists of two ripple carry adders and a multiplexer. The carry-select adder is simple but rather fast, having a gate level depth of $O(\sqrt{n})$. Adding two n -bit numbers with a carry select

adder is done with two adders (two ripple carry adders) in order to perform the calculation twice, one time with the assumption of the carry being zero and the other assuming one.

After the two results are calculated, the correct sum, as well as the correct carry, is then selected with the multiplexer once the correct carry is known. The design schematic of Carry Select Adder is shown in Fig.



The N-bit Ripple Carry Adder constructed by N set single bit Full-adder

In the N-bit carry ripple adder, the delay time can be expressed as:

$$TCRA = (N-1) T_{carry} + T_{sum}$$

In the N-bit carry select adder, the delay time is:

$$TCSA = T_{setup} + (N/M) T_{carry} + MT_{mux} + T_{sum}$$

In our proposed N-bit area-efficient carry select adder, the delay time is:

$$T_{new} = T_{setup} + (N-1) T_{mux} + T_{sum}$$

The carry select adder comes in the category of conditional sum adder. Conditional sum adder works on some condition. Sum and carry are calculated by assuming input carry as 1 and 0 prior the input carry comes. When actual carry input arrives, the actual calculated values of sum and carry are selected using a multiplexer.

The conventional carry select adder consists of $k/2$ bit adder for the lower half of the bits i.e. least significant bits and for the upper half i.e. most significant bits (MSB's) two $k/2$ bit adders. In MSB adders one adder assumes carry input as one for

performing addition and another assumes carry input as zero. The carry out calculated from the last stage i.e. least significant bit stage is used to select the actual calculated values of output carry and sum. The selection is done by using a multiplexer. This technique of dividing adder in two stages increases the area utilization but addition operation fastens.

2.6 Ripple Carry Adder

The basic addition operation at the bit level can be accomplished with a Full Adder (FA) circuit. FA adds two input bits X_i and Y_i along with an input carry C_{in} , resulting in a sum S_i and a carry-out bit C_{out} as shown in Figure 3(b). The operation performed by the FA is defined by the following boolean equations for the sum and the carry-out bits:

$$S_i = X_i \oplus Y_i \oplus C_{in}$$

$$C_{out} = (X_i \wedge Y_i) \vee (C_{in} \wedge (X_i \oplus Y_i))$$

$$= \text{Majority}(X_i, Y_i, C_{in})$$

The following notation for various Boolean operators will be used in this work to avoid ambiguity

$$x \vee y \leftrightarrow x \text{ OR } y$$

$$x \wedge y \leftrightarrow x \text{ AND } y$$

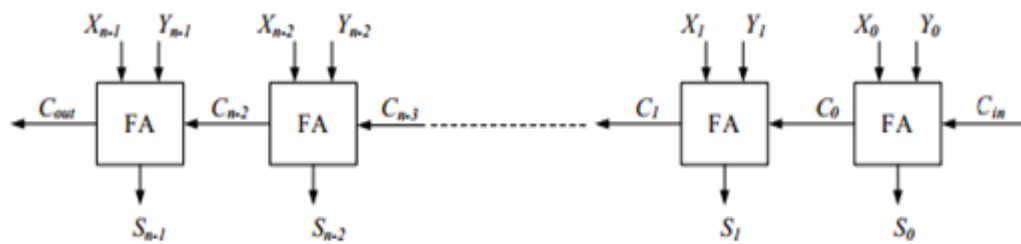
$$x \oplus y \leftrightarrow x \text{ XOR } y$$

$$x \leftrightarrow \text{NOT } x$$

It is apparent from equations 2.4 and 2.5 that the realization of the sum function requires two XOR logic gates, while two AND and one OR logic gates are needed for the carry-out function. Despite that, FA sum and carry-out functions can be represented in many different logic expressions and, thereby, determine the structure of the circuit. Based upon those different logic expressions, many full-adder cells and modules can be conceived. This provides the digital designer with various alternatives for the FA adder implementations to choose from and to investigate. Recently Shams et al. carried out detailed performance analysis of twenty three 1-bit FA. Their study

showed that each adder cell exhibits its own figures of power consumption, delay and area. The area and power-delay product performance of six existing 1-bit FA adders and proposed a new design based on XOR/XNOR. Proposed five different FA expressions based on XOR/XNOR implementation to explore different performance tradeoffs. Then, they used their proposed FA cells to improve the area and power of an array tree multiplier.

The 1-bit FA is cascaded as illustrated in Figure 4 to create n-bit wide operand adder known as Ripple Carry Adder (RCA). The sum at each bit position i is determined by the corresponding bit values of the operands at that position and the incoming carry bit value from $(i - 1)$ th position. The addition is completed once the carry value propagates along the entire structure to the most significant bit (MSB) position.



Ripple carry adder block diagram.

The area and delay of this adder can be roughly estimated using the unit-gate delay and area model. This model is technology independent and assumes that each gate, excluding exclusive-OR, counts as one elementary gate for both area and delay. An exclusive-OR gate counts for two elementary gates for both area and delay.

Complex gates as well as multi-input gates are built from 2-input basic gates and their gate count equals the sum of gate counts of the composing cells. Thus, RCA delay is estimated as $2n$ unit-gate delay while its area is $7n$ unit-gate area, where n is the operand size. The main advantage of RCA implementation is that it is area efficient and easy to construct. However, its linear delay characteristics makes it less suitable for high-speed implementations. An improved addition approach is given next.

Multipliers

Introduction

Multiplication is important fundamental function in arithmetic logic operation. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. The objective of good multiplier to provide a physically compact high speed and low power consumption unit. To reduce significant power consumption of multiplier design it is a good direction to reduce number of operations thereby reducing a dynamic power which is a major part of total power dissipation.

An efficient multiplier should have following characteristics:-

Accuracy:- A good multiplier should give correct result.

Speed:- Multiplier should perform operation at high speed.

Area:- A multiplier should occupies less number of slices and LUTs.

Power:- Multiplier should consume less power.

Multiplication process or A multiplier can be divided into three steps-

1. Partial product generation-The first is radix 4 booth encoding in which a partial product is produced from the multiplier and multiplicand.
2. Partial product reduction-The second is adder array or partial product compression to add all partial products and convert them into the form of sum and carry.
3. Final addition-The last is the final addition in which the final multiplication result is generated by adding the sum and carry . $Z=A*B+Z$.

For the multiplication of an n-bit multiplicand with an m bit multiplier, m partial products are generated and products formed is n + m bits long.

$$\begin{array}{rcccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 & \\
 & & & & B_3 & B_2 & B_1 & B_0 & \times \\
 \hline
 & & & & A_3 \cdot B_0 & A_2 \cdot B_0 & A_1 \cdot B_0 & A_0 \cdot B_0 & \\
 & & & A_3 \cdot B_1 & A_2 \cdot B_1 & A_1 \cdot B_1 & A_0 \cdot B_1 & & \\
 & & A_3 \cdot B_2 & A_2 \cdot B_2 & A_1 \cdot B_2 & A_0 \cdot B_2 & & & \\
 & A_3 \cdot B_3 & A_2 \cdot B_3 & A_1 \cdot B_3 & A_0 \cdot B_3 & & & & + \\
 \hline
 P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & &
 \end{array}$$

$$A=(a_n \ a_{n-1} \ a_{n-2} \dots a_0) \quad B=(b_n \ b_{n-1} \ b_{n-2} \dots b_0)$$

$$AB= (A2^n b_n + A2^{n-1} b_{n-1} + A2^{n-2} b_{n-2} + \dots + A2^0 b_0)$$

Types of Multipliers

The common multiplication method is “add and shift” algorithm.

- 1.) In parallel multipliers number of partial products to be added is the main parameter that determines the performance of the multiplier.
- 2.) To reduce the number of partial products to be added, Modified Booth algorithm is one of the most popular algorithms.
- 3.) To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages.
- 4.) 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.

Applications

- 1.) Multiplication is a heavily used arithmetic operation that figures prominently in signal processing and scientific applications
- 2.) Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc.
- 3.) Multipliers play an important role in today’s digital signal processing and various other applications.

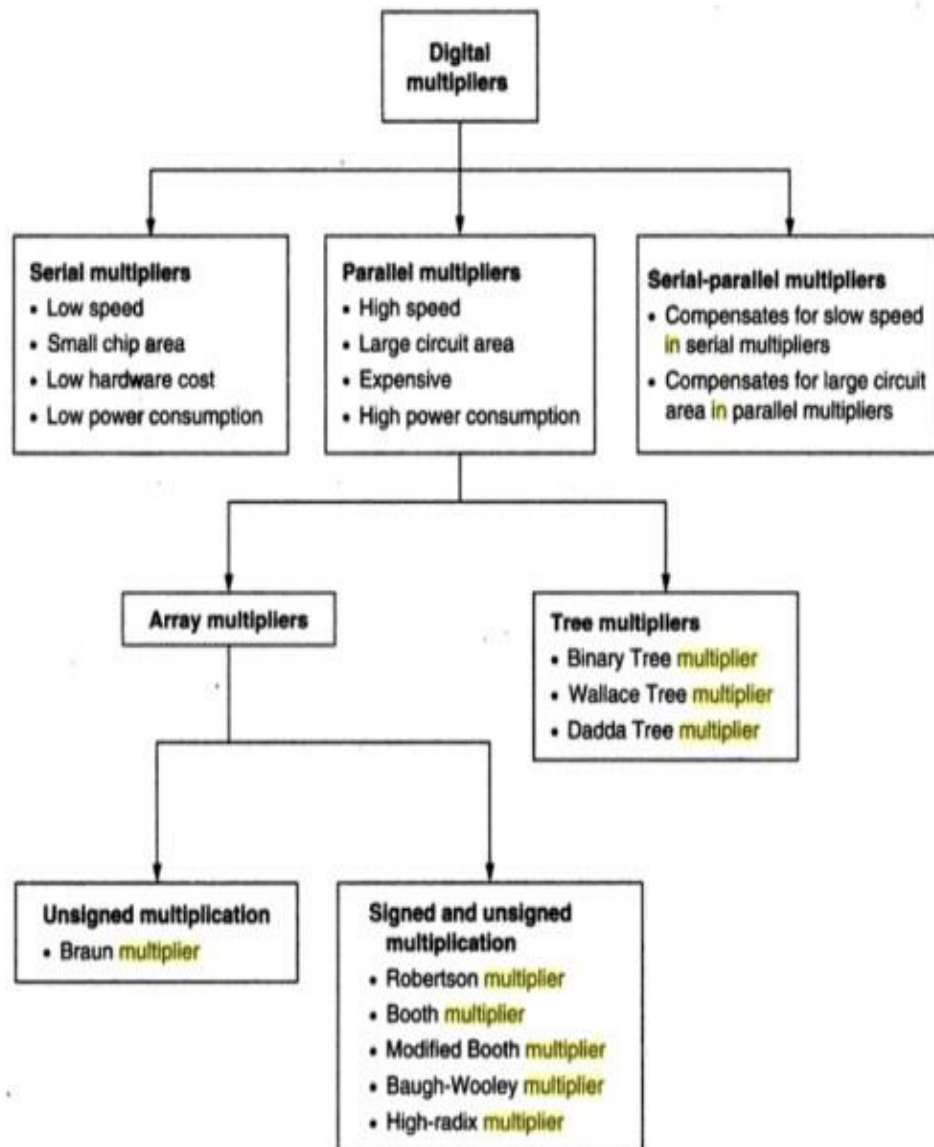


Figure 4.1 Types of digital multipliers.

Types of Multipliers

Serial-Parallel Multiplier

The serial multiplier uses successive addition algorithm, where both operands are entered in serial manner, which leads to poor speed performance. However in the parallel multiplier both operands are entered in parallel manner, which gives high

speed but occupies much larger area when compared to serial multiplier. Hence, we go for serial-parallel multiplier.

The serial-parallel multiplier serves as a good trade-off between the time consuming serial multiplier and area consuming parallel multipliers. These multipliers are used when there is demand for both high speed and small area. In a device using the serial-parallel multiplier, one operand is entered serially and the other operand is stored in parallel with a fixed number of bits. The resultant enhancement in the processing speed and the chip area will become more significant when a large number of independent operations are performed.

1.) This multiplier is the simplest one, the multiplication being considered as a succession of additions.

If $A = (a_n \ a_{n-1} \ a_{n-2} \dots a_0)$

$B = (b_n \ b_{n-1} \ b_{n-2} \dots b_0)$

then the product $A.B$ may be expressed as

$$AB = (A2^n b_n + A2^{n-1} b_{n-1} + A2^{n-2} b_{n-2} + \dots + A2^0 b_0)$$

2.) To implement this we use D flip-flop and full adder.

3.) In this D flip-flop acts as a memory to store the data values and full adder circuit is used for adding the partial products.

4.) A possible form of this multiplier for multiplying 4-bit quantities based on this expression is shown in figure (1). The operation of the multiplier is as follows-

i.) 'Number A' is entered in the 4 right most bits of the top row of D flip-flop, which are further connected to three D flip-flops to form a 7-bit shift register. The first left most column of D flip-flops holds B values.

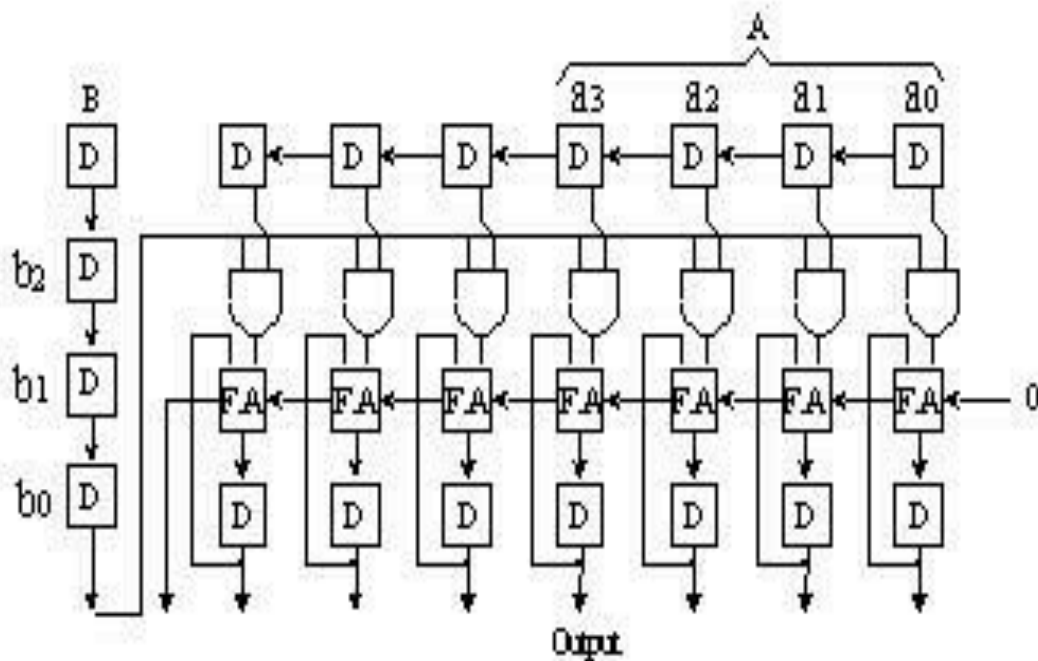
ii.) The number $A(a_3 \ a_2 \ a_1 \ a_0)$ is then multiplied with the least significant bit of $B(b_0)$. Later the number A is shifted and then multiplied with the other bits of B one after the other simultaneously. The partial products are then added using full adders.

iii.) This is done to allow the multiplication of 'number A' by $2^1 \ 2^2 \dots 2^n$, thus forming the partial product at each stage of the process.

5.) This approach can be used to eliminate the least significant bits of the product.

6.) A further reduction in hardware can be done by using 3 additional D flip-flops (which were earlier used as shifting of A proceeds) for holding b values.

7.) This structure is suited only for positive or unsigned operands. If the operands are negative and 2's complement encoded then -



- i.) The most significant bit of B will have a negative weight and so a subtraction must be performed as the last step.
- ii.) The most significant bit of A must be replicated since operand A must be expanded to $2N$ bits.

Figure (1): 4- bit Serial and Parallel Multiplier

Braun Multiplier

Braun Edward Louis proposed the braun multiplier in 1963. It is the simplest parallel multiplier, that is commonly known as the Carry Save Array Multiplier.

This multiplier consists of an array of AND gates and adders arranged in an iterative structure that does not require logic registers. This is also known as the non-additive multiplier since it does not add an operand to result of the multiplication. The completion time is limited by the depth of the Carry Save Array, and by the Carry propagation in the adder. This multiplier is suited only for positive operands. This multiplier is restricted to performing multiplication of two unsigned numbers.

Architecture

- 1.) An $n \times n$ bit Braun multiplier is constructed with $n(n-1)$ adders, n^2 AND gates and $(n-1)$ rows of Carry Save Adder.
- 2.) In the first rows there is no Carry propagation (using Carry Save adder). At the bottom of the array, the output of the array is noted in Carry Save, so an adder converts it (by means of a Carry propagation) into the classical notation.
- 3.) Each product can be generated in parallel with the AND gates. Each partial product can be added to the previous sum of partial products (which has been produced by using the row of adders).
- 4.) The carry out will be shifted one bit to the left or right and then are added to the sum of first adder and the newly generated partial product.
- 5.) The shifting would carry out with the help of Carry Save Adder (CSA) and the Ripple carry adder should be used for the final stage of the output.
- 6.) The schematic diagram is as shown figure(4.2).

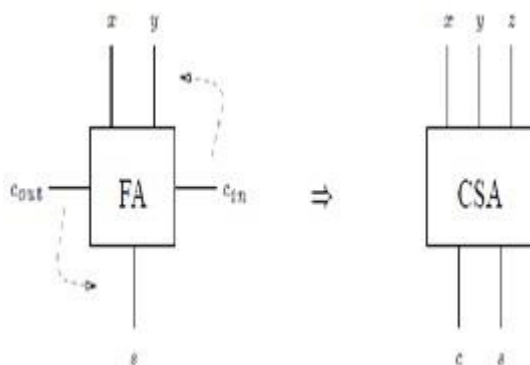


Figure - Carry save adder

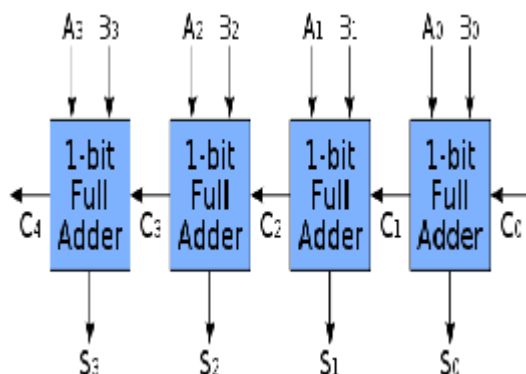
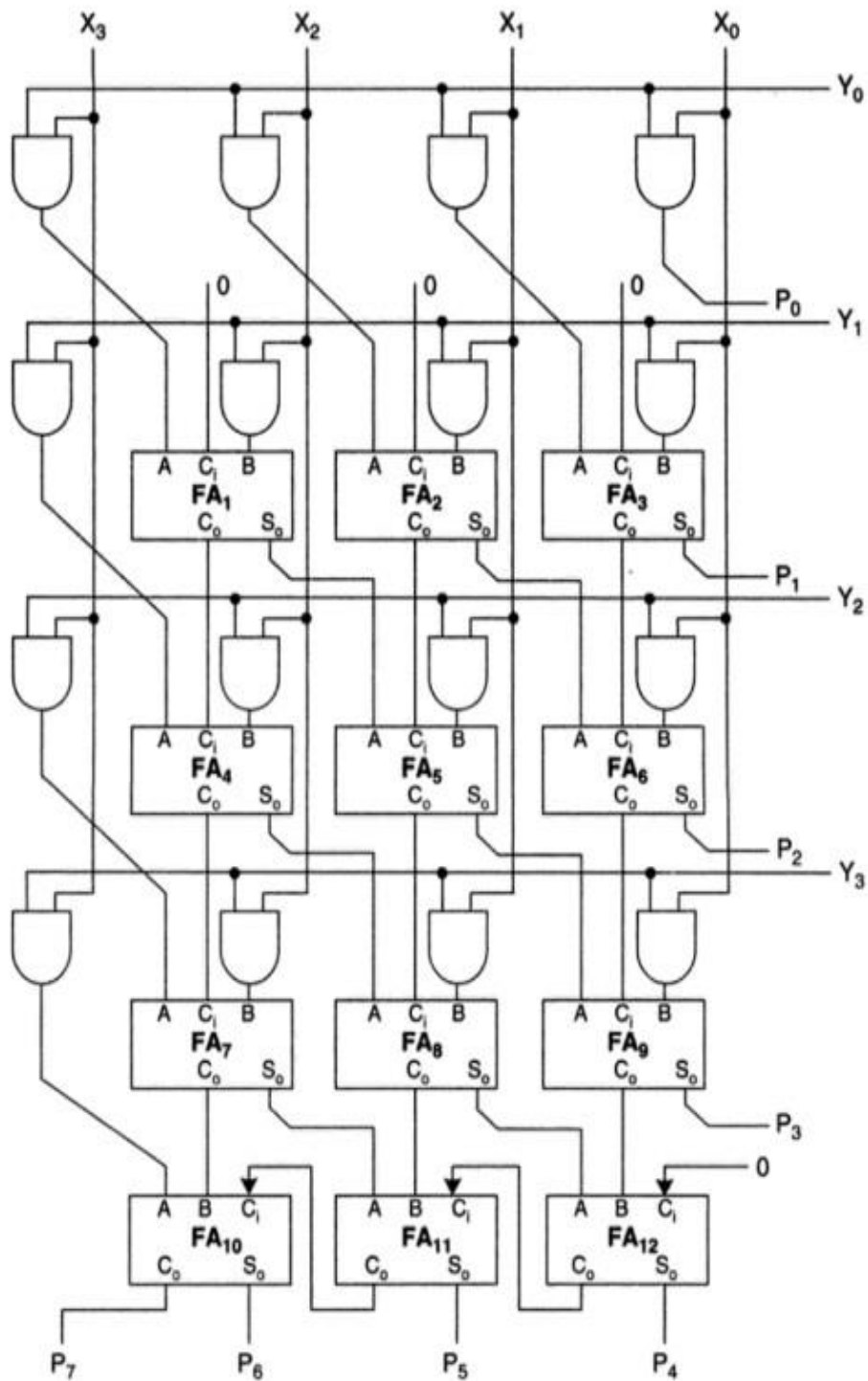


Figure - Ripple Carry adder



X: 4-bit multiplicand
Y: 4-bit multiplier
P: 8-bit product of X and Y
 $P_n = X_i Y_j$ is a product bit

Figure 4.2 Schematic diagram of a 4 x 4-bit Braun multiplier.

Performance of Braun Multiplier

- 1.) Braun multiplier performs well for the unsigned operands that are less than 16 bits in terms of speed, power and area. But it is simple structure when compared to the other multipliers.
- 2.) The number of components required in building the Braun multiplier increases quadratically with the number of bits, which makes it inefficient.
- 3.) The main drawback of this multiplier is that the potential susceptibility of glitching problem due to the Ripple Carry Adder in the last stage. The delay depends on the delay of the Full Adder and also a final adder in the last stage.

To overcome drawback

- 1.) The internal structure of the full adder can be realized using FPGA. The power and area can also be reduced by using two bypassing techniques called Row bypassing technique and Column bypassing technique.
- 2.) Delay due to the final ripple adder can be minimized by using very fast one of a Parallel Prefix Adder “KOGGE STONE ADDER” which is a type of Carry Look Head Adder.

Speed consideration:

- 1.) The delay of the Braun multiplier is dependent on the delay of the full Adder cell and also on the final Adder in the last row.
- 2.) In the multiplier array, a full Adder with balanced Carry and sum delays is desirable because the sum and carry signals are both in the critical path.
- 3.) The speed and power of the full Adder is very important for large arrays.

□Enhanced Braun Multipliers

- 1.) The performance of Braun Multiplier can be enhanced by replacing full adders with half adder, which will result in saving three logic gates, but regularity of structure gets disturbed.
- 2.) The another way to do this is by optimising the interconnection between the adders, so that delay through out each adders path is approximately same.

Baugh-Wooley multiplier

In signed multiplication the length of the partial products and the number of partial products will be very high. So an algorithm was introduced for signed multiplication called as Baugh- Wooley algorithm. The Baugh-Wooley multiplication is one amongst the cost-effective ways to handle the sign bits. This method has been developed so as to style regular multipliers, suited to 2's compliment numbers.

Baugh-Wooley Two's complement Signed multipliers is the best known algorithm for signed multiplication because it maximizes the regularity of the multiplier and allow all the partial products to have positive sign bits.

$$\begin{array}{r}
 \begin{array}{cccc}
 A_3 & A_2 & A_1 & A_0 \\
 B_3 & B_2 & B_1 & B_0
 \end{array} \times \\
 \hline
 \begin{array}{cccc}
 A_3 \cdot B_0 & A_2 \cdot B_0 & A_1 \cdot B_0 & A_0 \cdot B_0 \\
 A_3 \cdot B_1 & A_2 \cdot B_1 & A_1 \cdot B_1 & A_0 \cdot B_1 \\
 A_3 \cdot B_2 & A_2 \cdot B_2 & A_1 \cdot B_2 & A_0 \cdot B_2 \\
 A_3 \cdot B_3 & A_2 \cdot B_3 & A_1 \cdot B_3 & A_0 \cdot B_3
 \end{array} \\
 \hline
 \begin{array}{cccccc}
 P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
 \end{array}
 \end{array}$$

Figure- unsigned 4-bit multiplication

						a3	a2	a1	a0
*						x3	x2	x1	x0
						-a3x0	a2x0	a1x0	a0x0
					-a3x1	a2x1	a1x1	a0x1	
				-a3x2	a2x2	a1x2	a0x2		
			a3x3	-a2x3	-a1x3	-a0x3			
=		s7	s6	s5	s4	s3	s2	s1	s0

Figure-signed 4-bit multiplication

	a_3	a_2	a_1	a_0	
x	b_3	b_2	b_1	b_0	
<hr/>					
	$\overline{a_3 b_0}$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$	pp0
	$\overline{a_3 b_1}$	$a_2 b_1$	$a_1 b_1$	$a_0 b_1$	pp1
	$\overline{a_3 b_2}$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$	pp2
	$a_3 b_3$	$\overline{a_2 b_3}$	$\overline{a_1 b_3}$	$\overline{a_0 b_3}$	pp3
1		1			
<hr/>					

Figure-Baugh-Wooley 4-bit algorithm

When multiplying two's complement numbers directly, each of the partial products to be added is a signed numbers. Thus each partial product has to be sign extended to the width of the final product in order to form a correct sum by the Carry Save Adder (CSA) tree. According to Baugh-Wooley approach, an efficient method of adding extra entries to the bit matrix suggested to avoid having deal with the negatively weighted bits in the partial product matrix.

Baugh-Wooley algorithm

Here are using fewer steps and also lesser adders. Here a_0, a_1, a_2, a_3 & b_0, b_1, b_2, b_3 are the inputs. I am getting the outputs that are $p_0, p_1 \dots p_7$. As I am using pipelining resister in this architecture ,so it will take less time to multiply large number of 2's compliment.

$$A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \quad (1)$$

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \quad (2)$$

Let us consider two numbers A and B (2's compliment number)

The product of two numbers is

$$\begin{aligned}
 A \cdot B &= a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} \\
 &\quad + b_{n-1} \left[-2^{2n-2} + 2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^{i+n-1} \right] \\
 &\quad + a_{n-1} \left[-2^{2n-2} + 2^{n-1} + \sum_{i=0}^{n-2} \overline{b_i} \cdot 2^{i+n-1} \right]
 \end{aligned} \quad \text{--->(3)}$$

The first two terms of above equation are positive and last two terms are negative. The last two terms are n-1 bits that extend in binary weight from 2^{n-1} upto 2^{2n-3} . On the other hand, the final product is 2n bits that extends in binary weight from 2^0 to 2^{2n-1} . In order to calculate the product, instead of subtracting the last two terms, it is possible to add the opposite values.

We see that subtractor cells must be used. In order to use only adder cells, the negative terms may be rewritten as :

$$A \cdot B = a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} - a_{n-1} \sum_{i=0}^{n-2} b_i \cdot 2^{n+i-1} - b_{n-1} \sum_{i=0}^{n-2} a_i \cdot 2^{n+i-1} \quad \text{----->(4)}$$

Then A.B becomes

$$- a_{n-1} \sum_{i=0}^{n-2} b_i \cdot 2^{n+i-1} = a_{n-1} \cdot \left(-2^{2n-2} + 2^{n-1} + \sum_{i=0}^{n-2} \overline{b_i} \cdot 2^{i+n-1} \right) \quad \text{---->(5)}$$

The final equation is

$$A \cdot B = -2^{2n-1} + (\overline{a_{n-1}} + \overline{b_{n-1}} + a_{n-1} \cdot b_{n-1}) \cdot 2^{2n-2} \text{ -----}(6)$$

$$\begin{aligned} & + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} + (a_{n-1} + b_{n-1}) \cdot 2^{n-1} \\ & + \sum_{i=0}^{n-2} b_{n-1} \cdot \overline{a_i} \cdot 2^{i+n-1} + \sum_{i=0}^{n-2} a_{n-1} \cdot \overline{b_i} \cdot 2^{i+n-1} \\ & - (b_{n-1} + a_{n-1}) \cdot 2^{2n-2} = -2^{2n-1} + (\overline{a_{n-1}} + \overline{b_{n-1}}) \cdot 2^{2n-2} \end{aligned}$$

The above equation signifies the Baugh-Wooley algorithm for multiplication process in two's complement form.

Baugh-Wooley Multiplier provides a high speed, signed multiplication algorithm . It uses parallel products to complement multiplication and adjusts the partial products to maximize the regularity of multiplication array . When number is represented in two's complement form, sign of the number is embedded in Baugh-Wooley multiplier. This algorithm has the advantage that the sign of the partial product bits are always kept positive so that array addition techniques can be directly employed. In the two's complement multiplication, each partial product bit is the AND of a multiplier bit and a multiplicand bit, and the sign of the partial product bits are positive . In this scheme , a total of $n(n - 1) + 3$ full adders are required. Hence, for the case of $n = 4$, the array requires 15 adders.

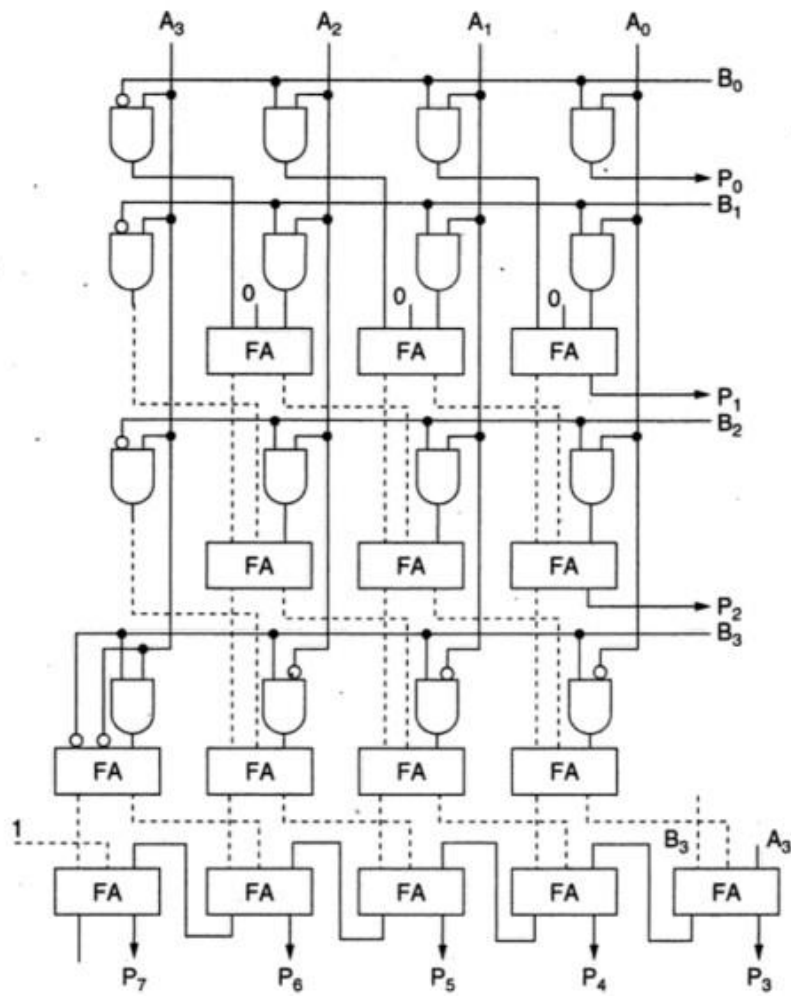
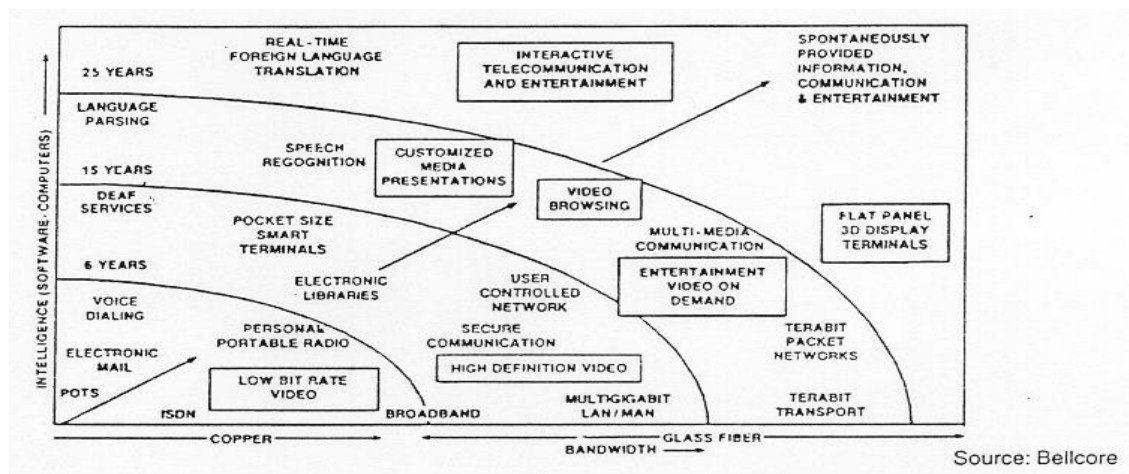


Figure 4.7 Modified Baugh-Wooley multiplier with optimized interconnection (dotted lines).

multiplier architectures. This algorithm is valid for both signed and unsigned operands.

FPGA AND CPLDS

Integrated circuits (IC) technology is the enabling technology for a whole host of innovative devices and systems that have changed the way of living. VLSI systems are much smaller and consume less power than discrete components used to build electronic systems before 1960's. The electronics industry has achieved a phenomenal growth over the last two decades, mainly due to the rapid advances in integration technologies, large-scale systems design in short, due to the advent of VLSI. The number of applications of integrated circuits in high-performance computing, telecommunications, and consumer electronics has been rising steadily, and at a very fast pace. Typically, the required computational power (or, in other words, the intelligence) of these applications is the driving force for the fast development of this field. Below figure gives an overview of the prominent trends in information technologies over the next few decades. The current leading-edge technologies (such as low bit-rate video and cellular communications) already provide the end-users a certain amount of processing power and portability.



Trends of VLSI

This trend is expected to continue, with very important implications on VLSI and systems design. One of the most important characteristics of information services is their increasing need for very high processing power and bandwidth (in order to handle real-time video, for example). The other important characteristic is that the information services tend to become more and more personalized (as opposed to collective services such as broadcasting), which means that the devices must be more

intelligent to answer individual demands, and at the same time they must be portable to allow more flexibility/mobility. As more and more complex functions are required in various data processing and telecommunications devices, the need to integrate these functions in a small system/package is also increasing. The level of integration as measured by the number of logic gates in a monolithic chip has been steadily rising for almost three decades, mainly due to the rapid progress in processing technology and inter connect technology.

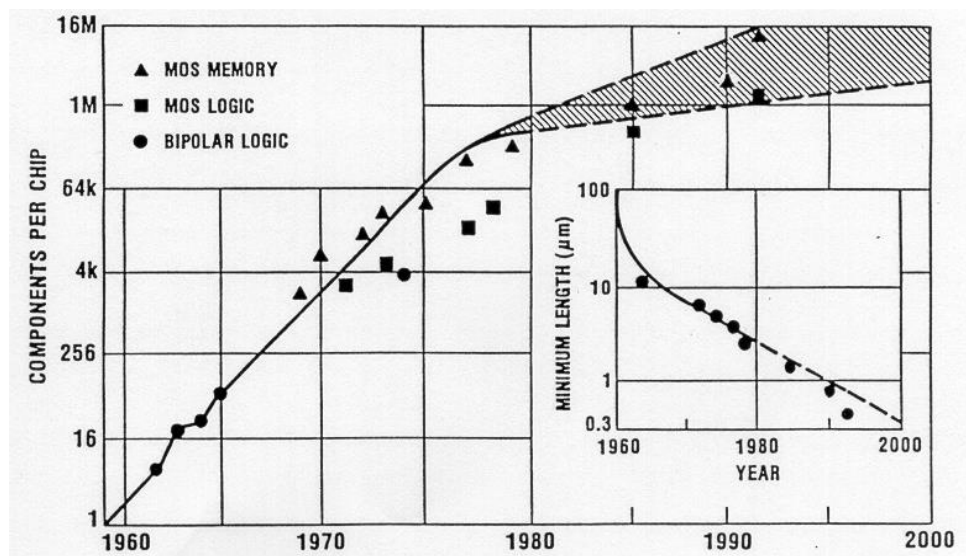


Figure 6.2: Evolution of integration density and minimum feature size, as seen in the early 1980s.

2. VLSI Design Flow

The design process at various levels is usually evolutionary in nature. It starts with a given set of requirements. Initial design is developed and tested against the requirements. When requirements are not met, the design has to be improved. If such improvement is either not possible or too costly, then the revision of requirements and its impact analysis must be considered. The three important domains in VLSI are Behavioral domain, Structural domain, Geometrical layout domain. The design flow starts from the algorithm that describes the behavior of the target chip. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning. The next design evolution in the behavioral domain defines

finite state machines (FSMs) which are structurally implemented with functional modules such as registers and arithmetic logic units (ALUs).

These modules are then geometrically placed onto the chip surface using CAD tools for automatic module placement followed by routing, with a goal of minimizing the interconnects area and signal delays. The third evolution starts with a behavioral module description. Individual modules are then implemented with leaf cells. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program. The last evolution involves a detailed Boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use.

3. Design Hierarchy

The use of hierarchy, or divide and conquer technique involves dividing a module into sub- modules and then repeating this operation on the sub-modules until the complexity of the smaller parts becomes manageable. This approach is very similar to the software case where large programs are split into smaller and smaller sections until simple subroutines, with well-defined functions and interfaces, can be written. The design of a VLSI chip can be represented in three domains. Correspondingly, a hierarchy structure can be described in each domain separately. However, it is important for the simplicity of design that the hierarchies in different domains can be mapped into each other easily.

In the physical domain, partitioning a complex system into its various functional blocks will provide a valuable guidance for the actual realization of these blocks on chip. Obviously, the approximate shape and size (area) of each sub-module should be estimated in order to provide a useful floor plan. This physical view describes the external geometry of the adder, the locations of input and output pins, and how pin locations allow some signals (in this case the carry signals) to be transferred from one sub-block to the other without external routing.

4.VLSI Design Styles

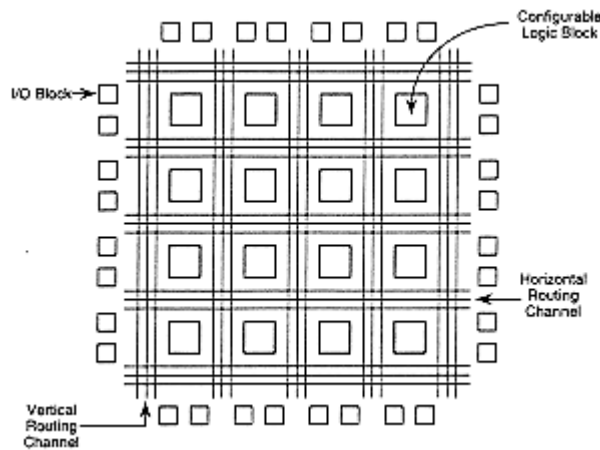
Several design styles can be considered for chip implementation of specified algorithms or logic functions. Each design style has its own merits and shortcomings, and thus a proper choice has to be made by designers in order to provide the functionality at low cost.

Field Programmable Gate Array (FPGA)

Fully fabricated FPGA chips containing thousands of logic gates or even more, with programmable interconnects, are available to users for their custom hardware programming to realize desired functionality. This design style provides a means for fast prototyping and also for cost-effective chip design, especially for low-volume applications. A typical field programmable gate array (FPGA) chip consists of I/O buffers, an array of configurable logic blocks (CLBs), and programmable interconnect structures. The programming of the inter connects is implemented by programming of RAM cells whose output terminals are connected to the gates of MOS pass transistors.

Performance of the design can be simulated and verified before downloading the design for programming of the FPGA chip. The programming of the chip remains valid as long as the chip is powered-on or until new programming is done. In most cases, full utilization of the FPGA chip area is not possible - many cell sites may remain unused.

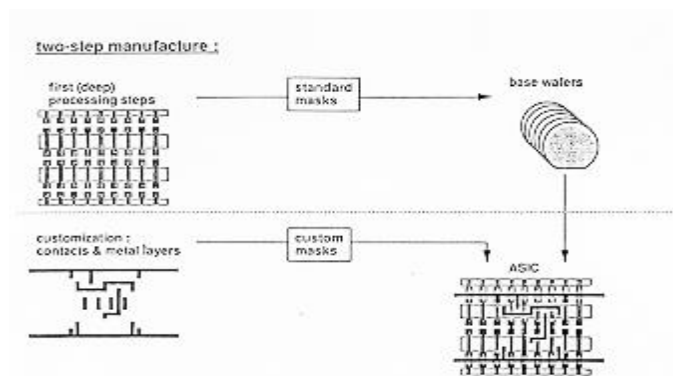
The largest advantage of FPGA-based design is the very short turn-around time, i.e., the time required from the start of the design process until a functional chip is available. Since no physical manufacturing step is necessary for customizing the FPGA chip, a functional sample can be obtained almost as soon as the design is mapped into a specific technology. The typical price of FPGA chips are usually higher than other realization alternatives (such as gate array or standard cells) of the same design, but for small-volume production of ASIC chips and for fast prototyping, FPGA offers a very valuable option.



General architecture of Xilinx FPGA.

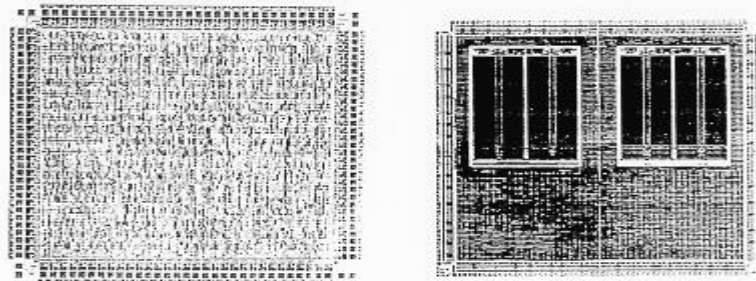
Gate Array Design

In view of the fast prototyping capability, the gate array (GA) comes after the FPGA. While the design implementation of the FPGA chip is done with user programming, that of the gate array is done with metal mask design and processing. Gate array implementation requires a two-step manufacturing process: The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip. These uncommitted chips can be stored for later customization, which is completed by defining the metal interconnects between the transistors of the array. Since the patterning of metallic interconnects is done at the end of the chip fabrication, the turn-around time can be still short, a few days to a few weeks.

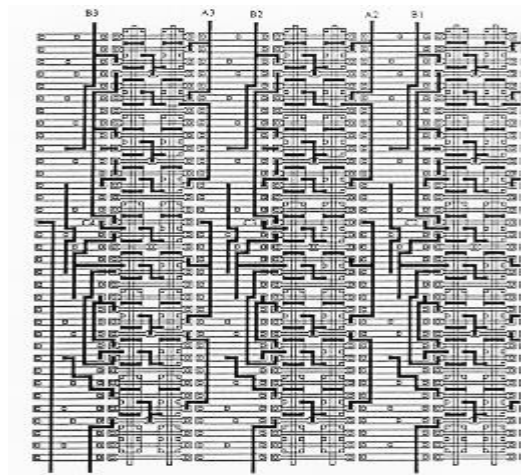


Basic processing steps required for gate array implementation.

Figure above shows a magnified portion of the internal array with metal mask design (metal lines highlighted in dark) to realize a complex logic function. Typical gate array platforms allow dedicated areas, called channels. The availability of these routing channels simplifies the interconnections, even using one metal layer only. The interconnection patterns to realize basic logic gates can be stored in a library, which can then be used to customize rows of uncommitted transistors according to the net list. While most gate array platforms only contain rows of uncommitted transistors separated by routing channels, some other platforms also offer dedicated memory (RAM) arrays to allow a higher density where memory functions are required.



Layout views of a conventional GA chip and a gate array with two memory banks.



Metal mask design to realize complex logic function on channeled GA

In general, the GA chip utilization factor, as measured by the used chip area divided by the total chip area, is higher than that of the FPGA and so is the chip speed,

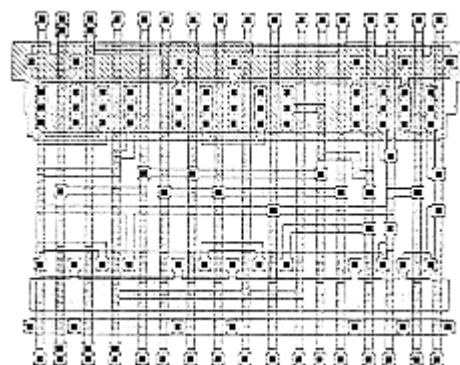
since more customized design can be achieved with metal mask designs. The current gate array chips can implement as many as hundreds of thousands of logic gates.

Standard-Cells Based Design

The standard-cells based design is one of the most prevalent full custom design styles which require development of a full custom mask set. For instance, the inverter gate can have standard size transistors, double size transistors, and quadruple size transistors so that the chip designer can choose the proper size to achieve high circuit speed and layout density.

The standard cell is also called the poly cell. In this design style, all of the commonly used logic cells are developed, characterized, and stored in a standard cell library. A typical library may contain a few hundred cells including inverters, NAND gates, NOR gates, complex AOI, OAI gates, D-latches, and flip-flops. Each gate type can have multiple implementations to provide adequate driving capability for different fan outs.

VHDL contains constructs that are more specific to simulation and verification than for synthesis. Synthesis software may ignore such constructs or rules. However, the goal is to match the simulation specification with the codes for synthesis. Depending on tools, the goal may (or usually) not be achievable. For example, the following two VHDL codes are different but describe the same design - one for simulation (cannot be synthesized efficiently) and the other for synthesis.

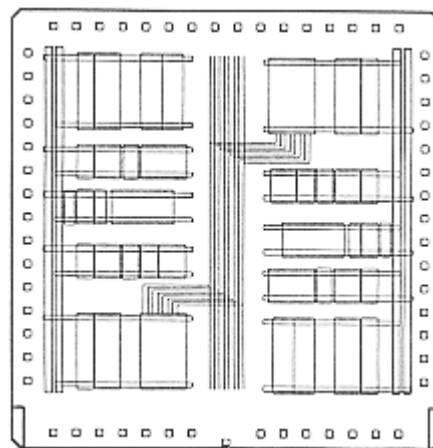


A standard cell layout example.

After chip logic design is done using standard cells in the library, the most challenging task is to place individual cells into rows and interconnect them in a way that meets stringent design goals in circuit speed, chip area, and power consumption. Many advanced CAD tools for place-and-route have been developed and used to achieve such goals.

Full Custom Design

Full custom design, in a strict sense, it is somewhat less than fully custom since the cells are pre-designed for general use and the same cells are utilized in many different chip designs. In a fuller custom design, the entire mask design is done anew without use of any library. However, the development cost of such a design style is becoming prohibitively high. Thus, the concept of design reuse is becoming popular in order to reduce design cycle time and development cost.



Simplified floor plan consisting of two separate blocks and a common signal bus.

The most rigorous full custom design can be the design of a memory cell, be it static or dynamic. Since the same layout design is replicated, there would not be any alternative to high density memory chip design. For logic chip design, a good compromise can be achieved by using a combination of different design styles on the same chip, such as standard cells, data-path cells and PLAs. In real full-custom layout in which the geometry, orientation and placement of every transistor is done individually by the designer, design productivity is usually very low - typically 10 to 20 transistors per day, per designer. In digital CMOS VLSI, full-custom design is rarely used due to the high labor cost. Exceptions to this include the design of high-

volume products such as memory chips, high- performance microprocessors and FPGA masters.

Cost of Manufacturing

IC manufacturing plants are extremely expensive. A single plant costs as much as \$4 billion. Given that a new state –of-the-art manufacturing process is developed every three years that is a sizeable investment. The investment makes sense because a single plant can manufacture so many chips and can easily be switched to manufacture different types of chips. In early years of the IC business companies focused on building large quantities of a few standard parts. These parts are commodities one 80 ns, 256 MB dynamic RAM is one more or less the same as any other regardless of the manufacturer. Companies concentrated on commodity parts because manufacturing variations are easier to keep track of when the same part is being fabricated day after day. Standard parts also made sense because designing integrated circuits was hard not only the circuit but the layout had to be designed, and there were few computer programs to help automate the design process.