Arithmetic Circuits

Dinesh Sharma
EE Department, IIT Bombay

October 30, 2019

Contents

List of Figures

Chapter 1

Adders

1.1 Half Adder

The truth table for addition of two bits is:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From this, we can see that

$$\text{sum} = A \cdot \overline{B} + B \cdot \overline{A} \quad (1.1)$$

$$\text{carry} = A \cdot B \quad (1.2)$$

What do we do with the carry? Obviously, it must be added to more significant bits. Therefore, for subsequent stages, we need an adder with *three* inputs: A, B and Carry in. This kind of adder is called a full adder.

1.2 Full Adder

Truth Table for the addition of three bits is:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

This leads to the Karnaugh map shown in fig. 1.1:

Cin \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

SUM

Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

CARRY

Figure 1.1: Karnaugh map for full adder

$$\text{So} \quad \text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}}$$

$$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A = A \cdot B + C_{in} \cdot (A + B)$$

1.3 Ripple Carry adder

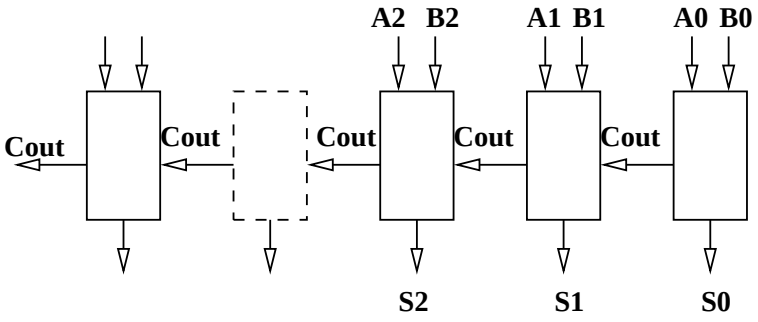


Figure 1.2: A ripple carry adder

- Carry out of one bit becomes Carry in of the next.
- This architecture is therefore called ripple carry adder.
- The critical delay path of the adder is the carry rippling from one bit to the next.

Because carry is on the critical path, Carry-out must be generated as quickly as possible. We need not optimize the delay of generating sum. We can in fact generate sum from

Carry out.

$$\begin{aligned}
\overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} && (1.3)\\
&= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) && (1.4)\\
&= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} && (1.5)\\
\overline{C_{out}} \cdot (A + B + C) &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} && (1.6)
\end{aligned}$$

$$\begin{aligned}
\text{sum} &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot C_{in} && (1.7)\\
&= \overline{C_{out}} \cdot (A + B + C_{in}) + A \cdot B \cdot C_{in} && (1.8)
\end{aligned}$$

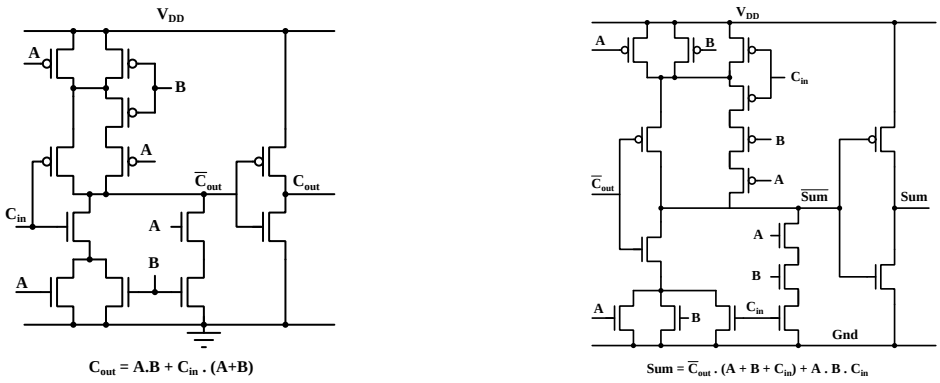


Figure 1.3: CMOS implementation for sum and carry

Both Sum and Carry show an interesting symmetry:

$$\begin{aligned}
\text{sum} &= \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} && (1.9)\\
\overline{\text{sum}} &= (A + B + \overline{C_{in}}) \cdot (A + \overline{B} + C_{in}) \cdot (\overline{A} + B + C_{in}) \cdot (\overline{A} + \overline{B} + \overline{C_{in}}) && (1.10)\\
&= (A + A \cdot \overline{B} + A \cdot C_{in} + A \cdot B + B \cdot C_{in} + \overline{C_{in}} \cdot A + \overline{C_{in}} \cdot \overline{B}) \cdot && (1.11)\\
&\quad (\overline{A} + \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C_{in}} + \overline{A} \cdot B + B \cdot \overline{C_{in}} + C_{in} \cdot \overline{A} + C_{in} \cdot \overline{B}) && (1.12)\\
&= (A + B \cdot C_{in} + \overline{B} \cdot \overline{C_{in}}) \cdot (\overline{A} + B \cdot \overline{C_{in}} + \overline{B} \cdot C_{in}) && (1.13)\\
&= A \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + \overline{A} \cdot \overline{B} \cdot \overline{C_{in}} && (1.14)
\end{aligned}$$

This shows that the **same hardware** that produces sum from A, B and C_{in}, will produce $\overline{\text{sum}}$ if the inputs are changed to $\overline{A}$, $\overline{B}$ and $\overline{C_{in}}$

$$\begin{aligned}
C_{out} &= A \cdot B + C_{in} \cdot (A + B) && (1.15)\\
\overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} && (1.16)\\
&= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) && (1.17)\\
&= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} && (1.18)
\end{aligned}$$

Thus the carry function also has the same property:
The same hardware which produces C_{out} from A, B and C_{in}, will produce $\overline{C_{out}}$ from $\overline{A}, \overline{B}$ and $\overline{C_{in}}$.

- In CMOS implementation, we interchange series and parallel configurations for the n and p channel transistors.
- This is to ensure that the pull up and pull down circuits are complementary.
- However, for sum and carry functions, we see that these functions are their own complements.
- Therefore, for implementing sum and carry, we can use the **same** configuration for n and p channel transistors.
- We use this to reduce the number of series connected transistors in pull up/pull down networks.

By making use of symmetry property of sum and carry, it is possible to simplify the implementations. These are called mirror gates because the n and p transistors have the **same**

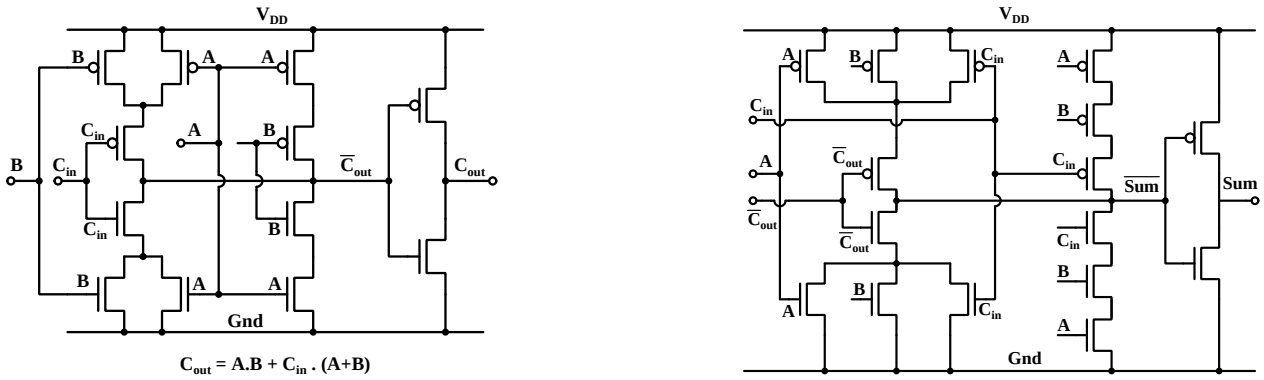


Figure 1.4: Mirror gate implementation of sum and carry

series parallel combination. This is highly unusual.

The worst case delay of the ripple carry adder is linear in number of bits to be added. To reduce the delay per stage, we can eliminate the inverter from the carry output. All even bit adders accept a, b and C_{in} as inputs. The mirror gate gives $\overline{C_{out}}$ as the output. All odd bit adders accept $\overline{A}, \overline{B}$ and $\overline{C_{in}}$ as inputs and thus produce C_{out} as output. Outputs of all bits

are now compatible with inputs of the next stage.

Extra inverters are required to produce $\overline{A}, \overline{B}$ and at the outputs to produce the proper result. However, these are not on the critical path, and do not add to the worst case delay. Extreme care needs to be taken in layout to ensure that the loading on the tree gate producing carry output is as small as possible.

1.4 Carry Look Ahead

Carry propagation is the critical path for a multi-bit adder. To speed up the adder, we would like an architecture where logic terms are classified as those dependent on carry and those which do not depend on carry. Then we can pre-compute all terms which do not depend on carry. When carry arrives, we can quickly compute the output carry and pass it on to the next stage.

Let us analyze what information can be pre-computed from A_i and B_i, which will help us in generating C_{out} quickly from C_{in}.

- When $A_i = 0$ and $B_i = 0$, C_{out} is 0, independent of C_{in}. We define this condition as 'Kill'. $K = \overline{A} \cdot \overline{B}$
- Similarly, when $A_i = 1$ and $B_i = 1$, C_{out} is 1, independent of C_{in}. We define this condition as 'Generate': $G = A.B$.
- Only when $A_i = 0$ and $B_i = 1$ or when $A_i = 1$ and $B_i = 0$,
  we need to wait for C_{in} to compute C_{out}.
  In both these cases, $C_{out} = C_{in}$.
- We call this condition as 'Propagate', and define $P = A.\overline{B} + \overline{A}.B$.

We define $K = \overline{A} \cdot \overline{B}, G = A.B$ and $P = A \oplus B$
Exactly one of K, G or P is true at any time.

When K = 1, C_{out} is 0, independent of C_{in}.
When G = 1, C_{out} is 1, independent of C_{in}.
When P = 1, $C_{out} = C_{in}$.

P is computed using an xor gate, which can be slow. However, the only difference between xor and or logic is when both inputs are 1, i.e. G = 1.

If we can ensure that G forces C_{out} to 1 irrespective of P, we can use the simpler 'or' logic to compute P.

C_{in} for bit i+1 is the C_{out} of bit i.
So we can write $C_{i+1} = G_i + P_i.C_i$
Notice that the Kill signal is not required.

If $G_i = 0$, $C_{i+1} = A \oplus B = A + B$ when $G = A.B = 0$
If $G_i = 1, C_{i+1} = 1$, and the value of P_i does not matter anyway.
So we can use P = $A + B$ instead of $P = A \oplus B$.

Now, we have the sequence:

$$C_{i+1} = G_i + P_i.C_i = G_i + P_i.G_{i-1} + P_i.P_{i-1}.C_{i-1} = \cdots$$

and so on, till we reach C_0.

Since all G_i, P_i and C_0 can be computed in parallel on arrival of the inputs, we can compute all sum and carry terms independently if we do not mind the added complexity.

$$C_{i+1} = G_i + P_i.C_i = G_i + P_i.G_{i-1} + P_i.P_{i-1}.C_{i-1} = \cdots$$

Unfortunately, static implementation of these gates has almost as much delay as the ripple carry implementation.

Therefore, the static implementation of computation of sum and carry terms as a logic expression depending on all A_i, B_i and C_0 is rarely used.

However, a dynamic implementation is still useful, and is known as the Manchester Carry Chain.

1.4.1 Manchester Carry Chain

When the clock is low, the output is unconditionally charged by the pMOS.
When the clock goes high, the output will be pulled low if G = 1 or if P = 1 and $\overline{C_{in}} = 0$.
In all other cases, the output will remain high. Thus this circuit implements the required logic.

This circuit can be concatenated for all bits and since P and G are ready before $\overline{C_{in}}$ arrives, the carry quickly ripples through from bit to bit.

Notice that the nMOS logic can be interpreted as:

$$\overline{P.Cin + G}$$

where C_{in} itself has been recursively generated by similar logic.

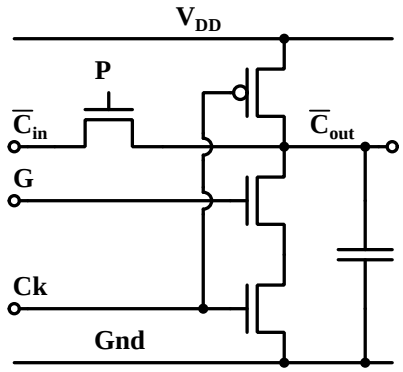


Figure 1.5: Manchester carry chain stage

As in the static case, there is a limit to the number of bits which can be so connected. If P = 1 for many successive bits, the discharge path is through series connected pass transistors of all these gates. The discharge time for this critical path has an n^2 dependence.

1.5 Carry Bypass Adder

The worst case for addition occurs when P = 1 for all bits and carry has to ripple through all the bits. In carry bypass adder, we form groups of bits and if P = 1 for all members of a group, we pass on the carry input to this group directly to the input of the next group, without having to ripple through each bit.

This improves the worst case delay of the adder.

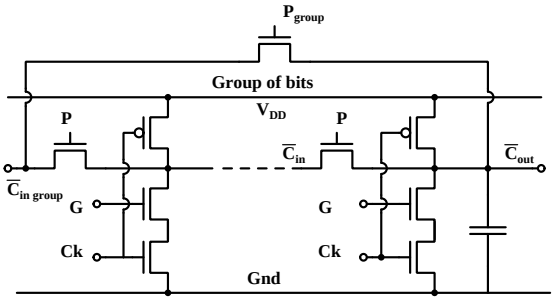


Figure 1.6: Carry By-pass adder

1.6 Carry Select Adder

An m bit carry select adder can be constructed as follows:

- We first compute the generate/propagate/kill signals for each bit (in parallel) from the input bits. Assuming unit gate delay model, this takes one unit of time.
- We use two m bit carry bypass adders. One of the adders assumes the carry input C_{in} to be 0, while the other assumes C_{in} to be 1. The two adders work in parallel and each takes m units of time.
- We now use a multiplexer controlled by the actual C_{in} to select the correct C_{out}. This takes one unit of time.
- The C_{out} of one such m bit adder will be used as the select input of the multiplexer of the next.
- The sum output of each bit is derived from P and C_{out} signals for the corresponding bit and appear one unit of time after C_{out} is available.

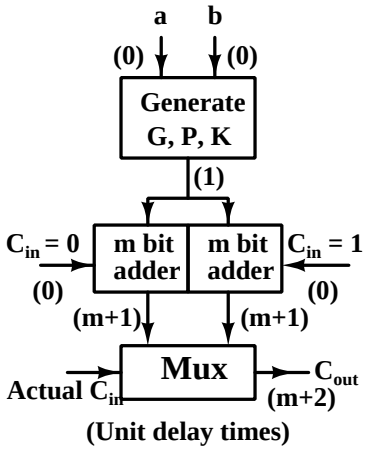


Figure 1.7: Carry select adder

Times of availability of various signals are noted in parentheses in the diagram.

- The two alternatives for the carry output are ready at (m+1) units of time.
- If the actual C_{in} is available at n units of time, the output will be available at (m+2) or (n+1), whichever is later.
- In case of 4 bit adders, this is at 6 units of time or at C_{in} arrival + 1, whichever is later.

1.6.1 Stacking Carry Select Adders

The sub-adders in carry select adder can use any architecture. They could be Manchester carry chains, carry bypass or ripple carry adders. Obviously, these sub adders should not be very long, otherwise, their outputs will be ready after a long time and we shall lose the advantage of carry select additions. Then, how do we make long adders using carry select?

This is done by stacking several smaller carry select adders.

We can stack several identical carry select adders. There is no need for carry select in the first stage, as C_{in} for this stage is available simultaneously with A_i and B_i.

Every subsequent stage will have two sub-adders, one assuming $C_{in} = 0$, the other assuming $C_{in} = 1$. The correct output will be selected by the actual C_{in} when it arrives.

Thus, after the first stage, each group of m bit adders will add only one unit of delay. This is much faster. However, the delay is still linear in number of bits.

A 32-bit adder made by cascading 8 4-bit carry select adders. Notice that the first group

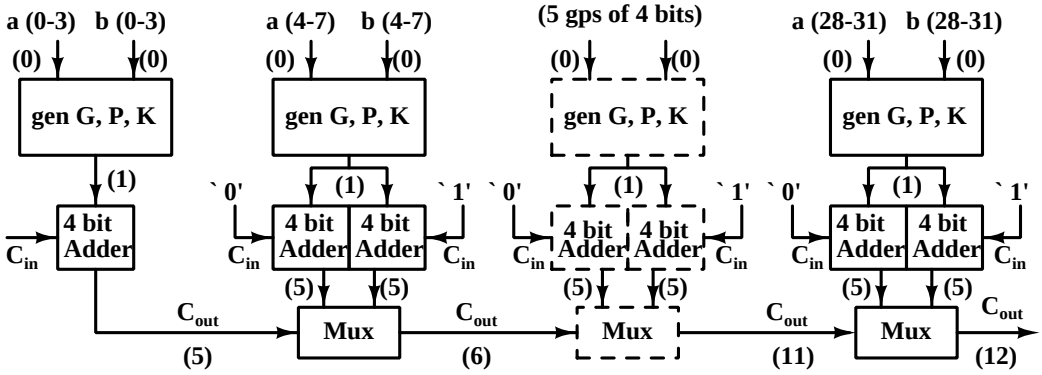


Figure 1.8: Linear stacking of carry select adders

does not need two adders and a mux, since the input carry is known to this group.

Bits	cy in	alt cy.s	cy out
0-3	0	-	5
4-7	6	5	6
8-11	7	5	7
12-15	8	5	8
16-19	9	5	9
20-23	10	5	10
24-27	11	5	11
28-31	12	5	12

The sum generation will take another unit of time, so the overall results will be available in 13 units of time.

Can we speed up the adder if we don't use the same no. of bits in every stage? In linear stacking, since all adders are identical, they are ready with their alternative outputs at the same time. But the carry arrives later and later at each successive group of carry select adders. We could have used this extra time to add up more bits in the later stages, and still be ready with the alternative results before carry arrives! Since the carry arrives one unit of time later at each successive group, each successive group could be longer by one bit.

We can do more bits of addition in the same time, if each successive stage is 1 bit longer than the previous one. Since the first stage does not add a mux delay, the first two stages will use the same number of bits. Thus, the number of bits which can be added is given by

$$m = n_0 + n_0 + (n_0 + 1) + (n_0 + 2) + \cdots = n_0 + \frac{s(n_0 + n_0 + s - 1)}{2}$$

where s is the number of stages following the first one without carry select.

The total delay will be $n_0 + 1$ for the first stage. Each subsequent stage takes just 1 unit of time since the candidates for selection are available just in time. Thus the time take is just $n_0 + s + 1$ units. When $s \gg n_0$, we have $m \approx s^2/2$, while the time taken is nearly s.

Thus the time taken to add m bits is $\approx \sqrt{2m}$

For a 32 bit adder, we could use a distribution like: 4,4,5,6,7,6.

Bits	carry in	carry alternatives	carry out
0-3	0	4	5
4-7	5	5	6
8-12	6	6	7
13-18	7	7	8
19-25	8	8	9
26-31	9	8	10

Our sum will be ready at 11 - which is much faster. This gain will be even higher for wider additions.

In square root tiling, the size of the sub-adders can become quite large for the later adders. Each of these can therefore be constructed as independent adders with any architecture. In particular, we might construct the sub-adders themselves as tiled carry select adders. Since these are faster, we can accommodate more bits in each of these stages. Thus the bits processed increase faster than square of the number of stages. In the asymptotic limit, the time for addition can be reduced to be logarithmic in the number of bits, rather than just square root.

However, it is possible to complete the addition in logarithmic time using tree adders, with much less complexity. These are the adders used in most modern systems using wide adders.

1.7 Tree Adders

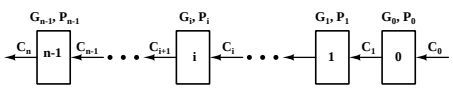
Tree adders use the idea of carry look ahead addition. However, these do not try to implement the complex logic expressions which result from looking ahead. Instead, these build up the logic in a tree like structure, where each node performs simple logic operations.

Recall that we had defined

$$K = \overline{A} \cdot \overline{B},\ G = A.B \text{ and } P = A \oplus B \tag{1.19}$$

When $K = 1,\ C_{out} = 0$, while if $G = 1,\ C_{out} = 1$, irrespective of C_{in}.
When $P = 1,\ C_{out} = C_{in}$ and this is the only case when we must wait for C_{in} to compute C_{out}

Consider an n bit adder with the least significant bit indexed as 0 and the most significant bit as $n-1$. The i'th bit accepts C_i as input carry and produces C_{i+1} as output carry.



We can express the output of the i'th stage as:

$$C_{i+1} = G_i + P_i \cdot C_i \tag{1.20}$$

Substituting for C_i, which is the output of cell no. (i-1),

$$\begin{aligned} C_{i+1} &= G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdot C_{i-1}) \\ &= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} \end{aligned}$$

If we try to compute the output carry for each bit by extending this logic to a function of $G(i)$, $P(i)$ and C_0, the expressions become very complex and their implementation will be slow. However, we can divide this work in a tree structure and that eventually leads to faster adders. Recall that

$$C_{i+1} = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1}$$

We can divide the expression for the output carry in parts which are independent of input carry and parts which are dependent on it. Let us call the original single bit G and P values for the i'th cell as G_i^1 and P_i^1. Then we can define

$$G_{i,i-1}^2 \equiv G_i^1 + P_i^1 \cdot G_{i-1}^1 \quad \text{and} \quad P_{i,i-1}^2 \equiv P_i^1 \cdot P_{i-1}^1 \tag{1.21}$$

This permits us to write

$$C_{i+1} = G_{i,i-1}^2 + P_{i,i-1}^2 \cdot C_{i-1} \tag{1.22}$$

Thus we can compute C_{i+1} from C_{i-1} directly using the same logic as before, except that we use $G_{i,i-1}^2$ and $P_{i,i-1}^2$ instead of G_i^1 and P_i^1.
Notice that the logic needed to compute $G_{i,i-1}^2$ from G_i^1 and P_i^1 is the same as that used to compute the output carry from input carry using G and P values of any order. Effectively, we have created a cell which is equivalent to two original adder cells and the carry has to be passed in groups of 2 bits across the adder. Of course, we need two stages to compute the second order G and P values, but these values are independent of carry and can be computed in parallel for all bit pairs.

Continuing the same process, we can combine two second order G and P values to compute third order G and P, which will permit computation of C_{i+1} directly from C_{i-3}.

$$G_{i,i-3}^2 \equiv G_{i,i-1}^2 + P_{i,i-1}^2 \cdot G_{i-2,i-3}^2 \quad \text{and} \quad P_{i,i-3}^3 \equiv P_{i,i-1}^2 \cdot P_{i-2,i-3}^2 \tag{1.23}$$

$$C_{i+1} = G_{i,i-3}^3 + P_{i,i-3}^3 \cdot C_{i-3} \tag{1.24}$$

Thus carry can now be passed over groups of 4 bits. Notice that the group size over which the carry can be computed directly *multiplies* by two each time we use a higher order for G and P values, while the time to compute the required higher order G and P values increments by one gate delay for logic $A + B \cdot C$ (for G) or $A \cdot B$ (for P). This is what results in ultimate

time to generate the final carry being logarithmic in the number of bits being added.

Since the generation of final carry has been speeded up substantially, we have to re-examine our assumption that carry propagation is the critical step in adder design. Addition is not complete unless all the sum bits and the terminal carry have been generated. In principle, we do not need the internal carries at each bit for the final result. The sum values at each bit are:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \tag{1.25}$$

For generating sums using this relation, we do need the internal carries at each bit. These can be computed using relations of the type described in equations 1.20,1.22 and 1.24 etc. Different architectures have been described in literature for the order of computation of G, P, C_{out} and Sum bits. All of these compute the final sum in times which are logarithmic functions of the number of bits. For wide adders,these can be much faster than other architectures.

1.7.1 Brent Kung adder

To illustrate the operation of tree adders, we use the architecture described by Brent and Kung for a tree adder. Many other tree adders have been described in the literature and we use this one as an example. In this adder, we compute the P and G values in a tree fashion, as shown in fig.1.9 for an 8 bit adder.

From the values of a_i, b_i, we first calculate P_i^1, G_i^1, with $i = 0 \cdots 7$. Next, using these values,

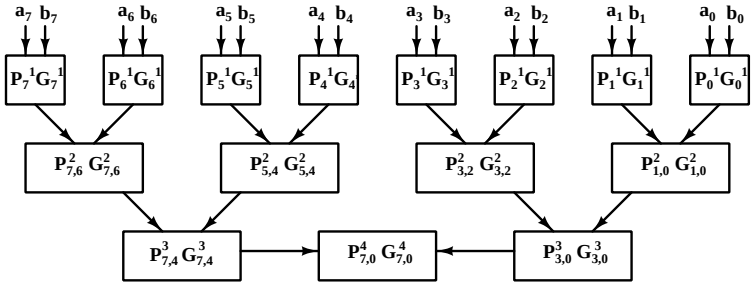


Figure 1.9: Generation of P and G values in Brent Kung adder

we can generate $P^2_{2i+1,2i}, G^2_{2i+1,2i}$ with $i = 0 \cdots 3$. In the next step, we use these values to generate $P^3_{4i+3,4i}, G^3_{4i+3,4i}$ with $i = 0, 1$. Finally, using these values, we can compute $P^4_{7,0}, G^4_{7,0}$.

As soon as values of P and G terms of various orders are known, we can compute the values of carry outputs which depend on these and the input carry.

$$C_1 = G_0^1 + P_0^1 \cdot C_0, \quad C_2 = G^2_{1,0} + P^2_{1,0} \cdot C_0$$

$$C_4 = G^3_{3,0} + P^3_{3,0} \cdot C_0, \quad C_8 = G^4_{7,0} + P^4_{7,0} \cdot C_0$$

When these carry values are valid, the other carry values which depend on these can be generated.

$$C_3 = G^1_2 + P^1_2 \cdot C_2, \quad C_5 = G^1_4 + P^1_4 \cdot C_4 \quad C_6 = G^2_{5,4} + P^2_{5,4} \cdot C_4,$$

Finally, C_7 can be generated from C_6.

$$C_7 = G^1_6 + P^1_6 \cdot C_6$$

With all carry values generated, the corresponding sum values can be calculated using the relation $\text{Sum}_i = P^1_i \oplus C_i$.

1.7.2 Other logarithmic adders

Brent Kung adder has low complexity and fanout from each stage. Other tree adders, (such as Kogge Stone adder) can be faster, following a similar scheme but with higher fanout and wiring congestion.

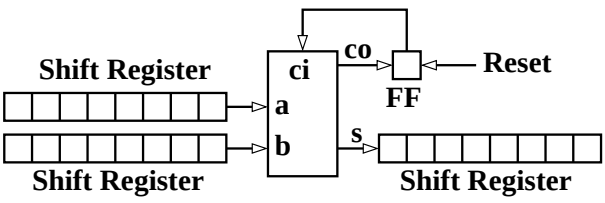
Rather than using radix 2 for generating P, G values, we can use radix 4. In this scheme, we use more complex logic for combining 4 P, G values every time to generate the next level P, G values. Thus, we can generate $P_{3,0}$ and $G_{3,0}$ directly from A_i, B_i. This reduces the depth of the tree, but at the same time, logic for each combining stage is more complex.

The other choice in logarithmic adders is for sparsity. The only use for generating bit-wise internal carries is for computing the sum bits. However, we may choose to generate only the even bit carries and compute sums for even as well odd bits using only these using more complex logic expressions.

1.8 Serial Adders

Up to now, we have been concerned with making fast adders, even at the cost of increased complexity and power. In many applications, speed is not as important as low power consumption and low cost.

Serial adders are an attractive option in such cases. A single full adder is used. If numbers to be added are available in parallel form, these can be serialized using shift registers.



A single full adder adds the incoming bits from operands A and B. Bits to be added are fed to it serially, LSB first.

- The sum bit goes to the output while carry is stored in a flip-flop.
- Carry then gets added to the next more significant bits which arrive in the next clock cycle.
- Output can be converted to parallel form if needed, using another shift register.

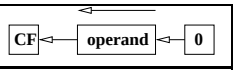
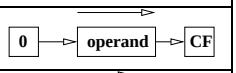
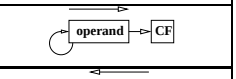
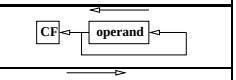
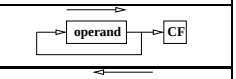
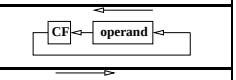
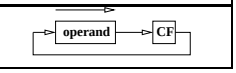
Chapter 2

Shift and Rotate Operations

We often need to shift and rotate operands in order to perform various operations such as serialization/de-serialization of data, multiplication etc.

2.1 Shift and Rotate Operations

The following operations are often required for processing of data:

SHL	op, count	
SAL	op, count	Same as SHL
SHR	op, count	
SAR	op, count	
ROL	op, count	
ROR	op, count	
RCL	op, count	
RCR	op, count	

These operations can be implemented by bi-directional shift registers with some control logic which provides circuits to choose the value and point of entry of new bits. However, this implementation can be very slow for a large number of shifts.

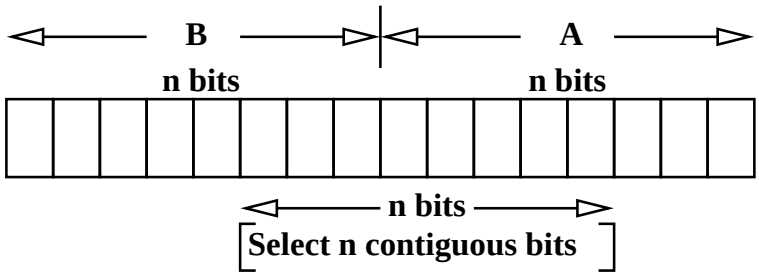
2.2 Barrel Shifters

Shift and rotate operations involve no computation. So why should we spend a large number of clock cycles for performing these operations? Ideally, we would like a shifter which produces the result in a single clock cycle.

2.2.1 Shift/Rotate as Select Operations

We can view Shift/Rotate operations as selection operations. In that case, the output can be produced directly by selecting the correct bits to appear in the desired position at the output.

Imagine two words A and B positioned as shown in the figure below.



We just have to choose B and A appropriately and select the correct range of n contiguous bits to implement various shift or rotate operations.

For all rotate operations, we choose B=A=data. For Shift Left, we choose B=data and A=0. For Logical Shift Right we choose B=0 and A=data. For Arithmetic Shift Right, we make B = replicated MSB of data, and set A = data.

Of course we do not actually copy data bits to A/B. Each output bit is produced by a mux which picks out the correct input data bit.

2.3 Barrel Shifters

Shifters which produce outputs as select operations are called barrel shifters. The name comes from viewing the inputs as well as outputs as a circular arrangement of bits. The shifter then connects the input circle to the output circle like the sections of a barrel.

A brute force implementation of such a shifter will require n multiplexers of n bits each, where the control inputs for each multiplexer are generated from the amount and type of shift/rotate that is desired. This requires quite a large number of complex n way multiplexers and puts a heavy load on data bits. Therefore rather than trying to complete the entire

operation in one go, we use logarithmic barrel shifters, which are not so complex and take of the order of $\log_2 n$ operations to produce the result

2.3.1 Logarithmic Barrel Shifters

The control logic of a one step barrel shifter is complex because the amount of shift is variable. The loading on data lines and control logic complexity can be reduced if we break up the shift/rotate process into parts. We can carry out shifts in different stages, each stage corresponding to a single bit of the binary representation of the **shift amount**. Now the i'th stage needs to produce either no shift (if the corresponding control bit is '0') or a shift by a constant amount which is 2^i. Thus each stage requires only a two way mux for each bit. For example, a shift by 6 (binary: 110) will be carried out by first doing a 4 bit shift and then a 2 bit shift.

Since we need n bits to represent a maximum shift amount of $2^n - 1$ places, the number of bits to express the shift amount (and hence the number of shift stages required) is logarithmic in the maximum shift desired.

That is why such shifters are called Logarithmic Barrel Shifters. We can optionally buffer the outputs after each stage.

Bit i of the **shift amount** represents no shift (if it is 0) and a constant shift by 2^i places (if it is 1). If the amount of shift is fixed, the required bit can just be wired from the input bits.

The 2 way mux is controlled by bit i of the **shift amount**, Using this, we can choose either the unshifted operand bit or the operand bit 2^i places away from it.

This can be done for all bits of the operand in parallel. This constitutes one stage of the logarithmic shifter. The output can then be shifted again in the next stage, controlled by the next significant bit.

2.3.2 Right Rotate for an 8 bit Operand

To perform a right rotate operation, we use the scheme shown in Figure 2.1. For an 8 bit operand, the amount of rotation will be between 0 to 7 places, which can be represented by 3 bits. So we shall need a 3 stage implementation.

- Each input bit drives just two muxes, each with just 2 inputs.
- At each stage, the muxes select either the unshifted bit or a bit 2^n places from it.
- 3 stages are required for 0 to 7 bits of shift.

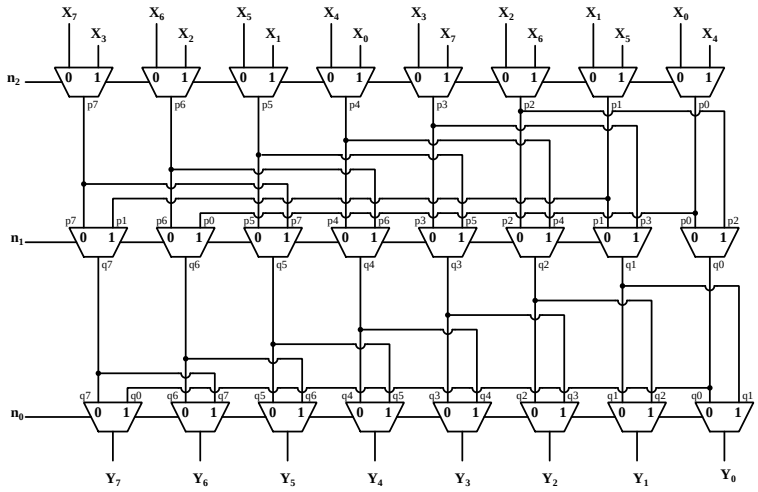


Figure 2.1: Right rotate for an 8 bit operand

2.3.3 8 bit Logical Shift Right

If we need a shift instead of a rotate, we feed a 0 instead of the corresponding data bit. We

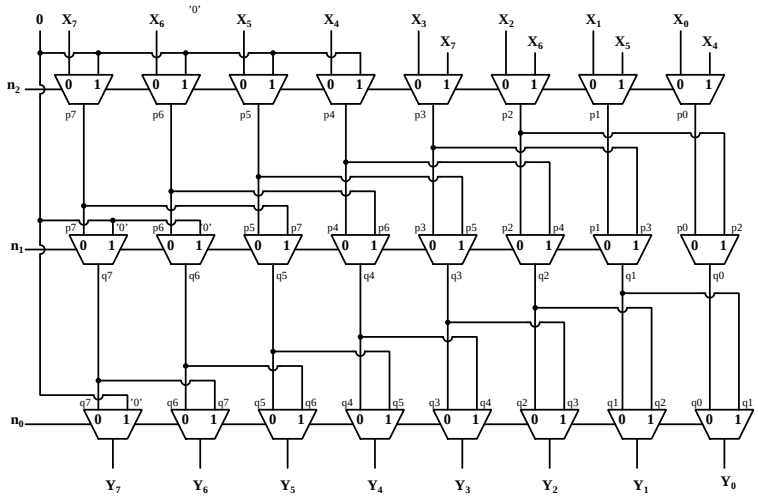


Figure 2.2: Logical Right Shift for an 8 bit operand

need to input 0's instead of data bits to the first 4 muxes in the first stage, to the first 2 muxes in the second stage and to 1 mux in the last stage.

2.3.4 Combining Rotate and Shift Operations

We can combine the circuits for rotate and shift functions by putting muxes where different inputs need to be presented for the two functions. Further, we can include the Arithmetic

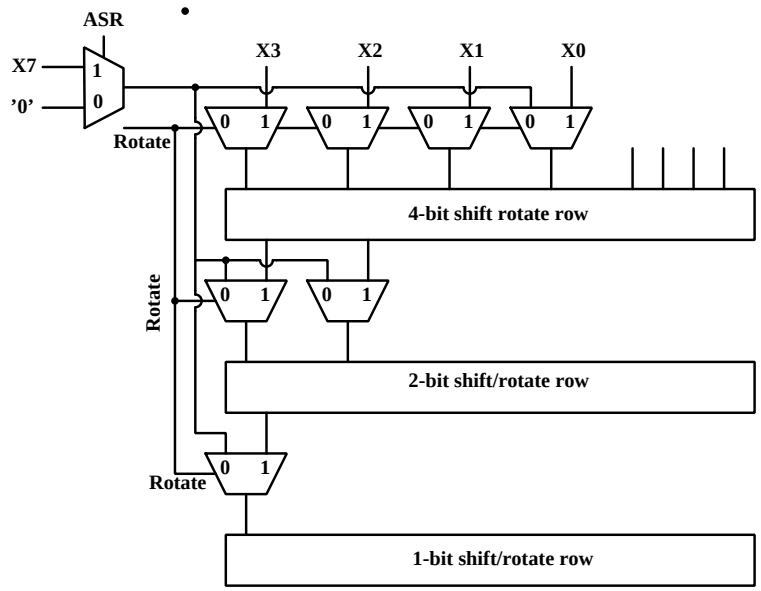


Figure 2.3: Combined Right Rotate/Shift for an 8 bit operand

Shift function by choosing between 0 or X7 as the bit to be inserted from the right.

2.3.5 Rotate and Shift by Masking

We can also combine the rotate and shift functions by masking. We use the rotate function which does not lose any information, as the primary circuit. Now we can mask n bits at the left to 0 if a right shift operation was desired instead. In case of an arithmetic shift, n bits on the left have to be set to the same value as X7.

Shift/Rotate Left case is similar, except that the Logical and Arithmetic shifts are not different operations.

2.3.6 Bidirectional Shift and Rotate Operations

It is possible to use the same hardware for left and right shift/rotate operations. This can be done by adding rows of muxes at the input and output which reverse the order of bits.

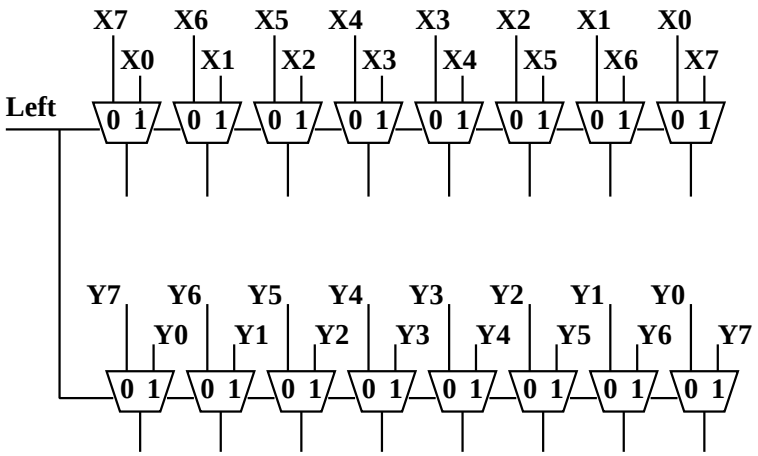


Figure 2.4: Bit reversal for bidirectional Shift and Rotate Operations

We can also make use of the fact that a left rotate by n places is the same as a right rotate by $2^n - n$ places. Now $2^n - n$ is just the 2's complement of n. By presenting the 2's complement of n at the mux controls, we can convert a right rotate to a left rotate. This can be followed by a mask operation, if a shift operation was required, rather than a rotate.

Chapter 3

Multipliers

3.1 Shift and Add Multipliers

An obvious way for implementing multipliers is to replicate the paper and pencil procedure in hardware.

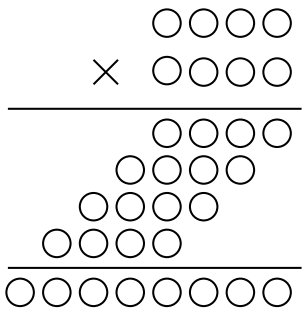


Figure 3.1: Shift and add multiplier

- Initialize the product to 0, extend multiplicand to left by n bits filled with 0s.
- If the least significant bit of the multiplier is 1, add the multiplicand to product, else do nothing.
- Shift the multiplier right by one bit.
- Shift the multiplicand left by one bit.
- Repeat for n bits

Each term being added to form the product is called a partial product. The name "partial product" is also used for individual bits of the terms being added - so beware!

The paper-pencil procedure requires n-1 additions to a 2n bit accumulator. This uses a single adder, but takes long to complete the multiplication. A 32 x 32 multiplication will require 31 addition steps to a 64 bit accumulator.

Multiplication can be made faster by using multiple adders and adding terms in a tree structure.

3.2 Array Multipliers

Suppose we want to multiply two n-bit numbers A and B, where

$$A = \sum_{i=0}^{n} 2^i a_i \qquad B = \sum_{j=0}^{n} 2^j b_j$$

We can regard all bits of the partial products as an array, whose (i,j)th element is $a_i \cdot b_j$. Notice that each element is just the AND of a_i and b_j. **All** elements of the array are available in parallel, within one gate delay of arrival of A and B. We can now use an array of full adders to produce the result. One input of each adder is the sum from the previous row, the other is the AND of appropriate a_i and b_j. This architecture is called an array multiplier.

A 4X4 array multiplier is shown in fig.3.2. Half adders can be used at the right end.

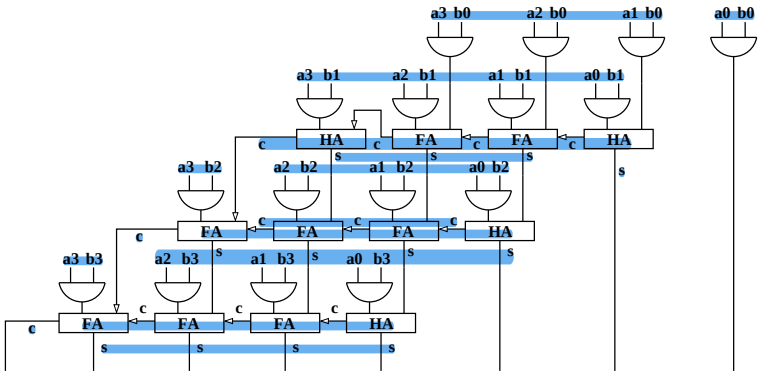


Figure 3.2: Array multiplier

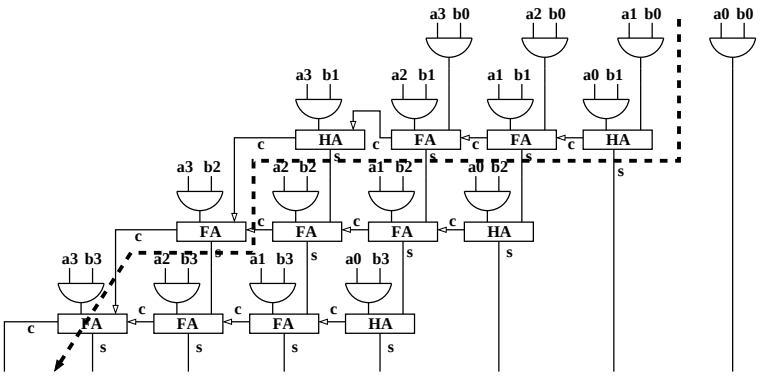


Figure 3.3: Critical path in an array multiplier

3.2.1 Critical Path through an Array Multiplier

The critical path through a 4X4 array multiplier is shown in fig.3.3 The critical path involves carry as well as sum outputs!

3.3 Speeding up Multipliers

The array multiplier has a regular layout with relatively short connections. However, it is still rather slow. How can we speed up a multiplier?

There are two possibilities:

- Somehow reduce the number of partial products to be added. For example, could we multiply 2 bits at a time rather than 1?
- Since we have to add more than two terms at a time, use an adder architecture which is optimized for this.

3.4 Booth Encoding

Booth Encoding reduces the number of partial products by multiplying 2 bits at a time.

Let the multiplicand be A and the multiplier B. Rather than multiplying A with successive bits of B, we can multiply it with two bits of B at a time. Depending on the two bits being 00, 01, 10 or 11, the partial product will be 0, A, 2A or 3A.

- 0 and A can be produced trivially.

- 2A can be produced easily by a left shift of A.
- Generating 3A presents a problem!

However, 3A can be expressed as 4A - A. The task of adding 4A is passed on to the next group of 2 bits of the multiplier. Since the place value of the next group of 2 bits is 4 times the current one, adding 4A to the product is equivalent to adding 1 to the next group of 2 bits of the multiplier. -A can be generated from A, using an adder/subtracter rather than an adder for accumulating the sum of partial products.

3.5 Modified Booth Encoding

To simplify the logic for deciding whether an additional 4A should be added on behalf of the less significant 2 bits in the multiplier, we express 2A also as 4A - 2A.

Since we anyway have an adder-subtracter, this requires no additional resources. The modified logic is: for 00, do nothing. For 01, add A.
for 10, subtract 2A, ask the next group to add 4A.
for 11, subtract A, ask the next group to add 4A.

Now the next group can just look at the more significant bit of the previous group and add 1 to the multiplier if it is '1'.

Current 2-bits	Multiplier for these	Previous MSBit	Pending Increment	Total Multiplier
00	0	0	0	0
01	+1	0	0	+1
10	-2	0	0	-2
11	-1	0	0	-1
00	0	1	+1	+1
01	+1	1	+1	+2
10	-2	1	+1	-1
11	-1	1	+1	0

Table 3.1: Effective multipliers for Modified Booth Algorithm

Table 3.1 summarizes the effective multiplier for generating the partial product. The partial product generator looks at the current 2 bits and the MSB of the previous group of 2 bits to decide its action. Thus, we scan the multiplier 3 bits at a time, with one bit overlapping. For the first group of 2 bits, we assume a 0 to the right of it. After handling the previous

group, the multiplicand is shifted left by 2 positions. Thus, it has already been multiplied by 4. Therefore, adding 4 A on behalf of the previous group is equivalent to adding 1 to the multiplier corresponding to the current group. Notice that a 111 in the 3 bit group being scanned requires no work at all.

What happens if there is a string of '1's in the multiplier? Consider multiplication by $111\cdots111$. As described earlier, we should add implied zeros to the right and left of this multiplier.

- Because the group begins with 110, there will be a -1 in the beginning.
- The group ends with 011. So there will be a +2 at the end,
- However, for the length of continuous '1's, nothing needs to be done (add zeros).

3-bits	Multiplier
000	0
001	+1
010	+1
011	+2
100	-2
101	-1
110	-1
111	0

Thus Booth encoding reduces the number of partial products to about half (multiplying 2 bits at a time). It also makes addition in columns of partial products fast because carry propagation during addition will be reduced.

3.6 Efficient Addition of Partial Products

Multipliers can be speeded up by using special adder architectures which are optimized for adding more than two numbers. One option is to use tree adders rather than an accumulator.

Figure 3.4: Tree additions in multipliers

Several additions proceed in parallel, since all partial products are generated together. Fig.3.4 shows the use of tree adders in a multiplier.

3.6.1 Carry Save Adders

Ordinary adders are large and complex. Also, these are slow due to rippling of carry. Let us consider an adder which presents its output not as one word - but two. The actual result is the sum of these.

Obviously, this kind of adder is of no use for adding just two words! But it can be useful in a multiplier where we are adding multiple terms. For each bit column, the sum goes into one output word, while carry outs go into the other (without being added to the next more significant column). Now there is no rippling of carry and the output is available in constant time.

We do need a conventional adder in the end to add these two words. This type of adder, which reduces the product to two words which must be added using a conventional adder is called a "Carry Save Adder" or CSA. For example, we can construct a useful CSA for adding

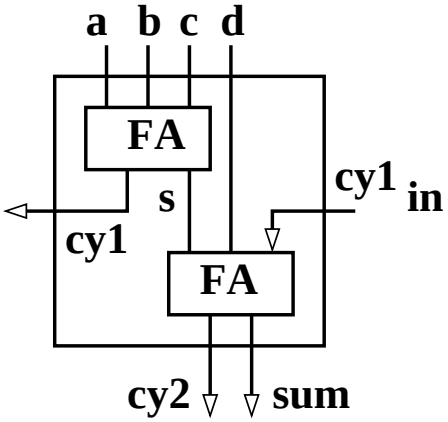


Figure 3.5: Carry save adder

4 bits in the same column. The 4 input 2 output CSA uses two full adders as shown in fig.3.5: We make use of the fact that all partial product bits are available in constant time after the application of inputs. Since there are 4 bits to be added, we feed three of them to a full adder. The sum and carry output of this adder is then available in constant time.

The sum output of first FA goes to the second FA. The carry output (cy1) of the first FS goes as intermediate input to the CSA used in the column to the left of this one.
The second FA accepts one left over bit from the partial product column, the sum output of the first Full Adder FA1 and cy1 output coming from the CSA to its right. All inputs to this Full Adder are also available in constant time. Notice that even though cy1 goes from one

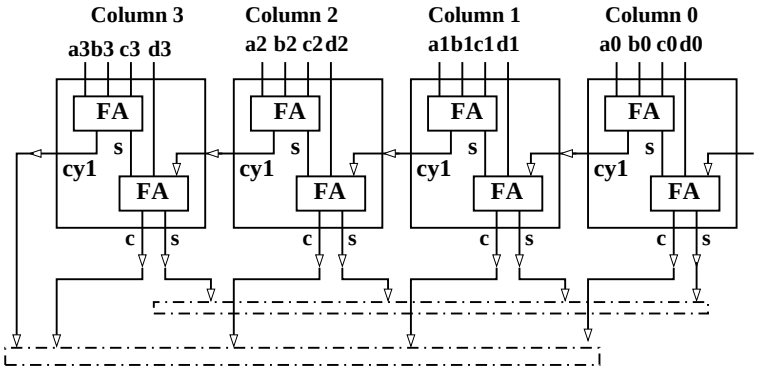


Figure 3.6: Tiling Carry Save Adders

column to the next significant column, **it does not ripple all the way** horizontally. It goes to FA2 of the more significant column whose output is not required by the next column.

3.6.2 Critical Path of Carry Save Adders

Figure3.6 shows how we can add 4 columns of 4 bits each.
Rows are labeled as a,b,c and d while columns are indexed as 0,1,2 and 3. Outputs are collected in two separate registers (shown in dotted lines). These must be added using a conventional adder. Critical path of a 4x4 Carry Save Adder is shown in fig.3.7. One can see

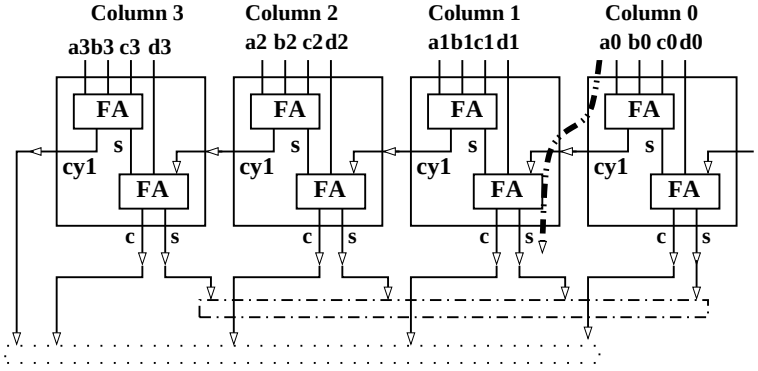


Figure 3.7: Critical path of a carry save adder

that the critical path has been broken up. Addition of 4 words of 32 bits each will have a critical path of the same length as that for 4 words of 4 bits each.

3.7 Wallace Multipliers

Multipliers do not have the same number of bits in every column. In 1964, Wallace proposed a method for a carry save adder like reduction scheme which is valid for columns of variable size. Wallace multiplier assumes adders which take multiple inputs of the same weight and produce sum outputs of the same weight and carry outputs with higher weights. These are combined in stages to reduce the number of terms at each weight to 2 or less. These two terms are then added by a conventional adder to produce the final result.

Wallace multipliers act in three stages:

1. Generate all bits of the partial products in parallel.
2. Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
3. For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

   Add these two numbers using a fast adder of appropriate size.

We assume that Full adders and Half adders will be used. A full adder takes 3 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is called a (3,2) adder. It reduces the number of wires at its own weight by 2 and adds one wire at the higher weight.

A half adder takes 2 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is a (2,2) adder. It reduces the number of wires at its own weight by 1 and adds one wire at the higher weight.

3.7.1 Reduction Stage of Wallace Multipliers

The reduction algorithm is general and can be used with any adders of type (n,m). For example, a carry save adder is of type (4,2).

- Each reduction stage looks at the number of wires for each weight and if any weight has more than 2 wires, it adds a layer of adders.
- When the numbers of wires for each weight have been reduced to 2 or less, we form one number with one of the wires at corresponding place values and another with the other wire (if present).

- These two numbers are added using a fast adder of appropriate size to generate the final product.

The original paper by Wallace did not give a reduction algorithm in detail. Subsequently, the Wallace reduction algorithm has been interpreted in many ways.We first describe the algorithm as interpreted by Dadda and other authors in many papers on multipliers. This, however, produces redundant most significant bits for multipliers of some sizes. For example, an 8x8 bit multiplier following this scheme will produce 17 bits. Of course, the 17th bit will always come out to be '0', because the product can be no wider than 16 bits. Since this scheme is widely described in literature, we shall first give the details of this scheme. Subsequently, we give a Wallace wire reduction scheme which does not lead to a redundant bit.

Wire Reduction Scheme for Wallace Multipliers

If any weight contains more than two wires, we use a reduction algorithm to reduce the number of wires. We divide the total number of rows to be added in groups of 3 rows. Any rows which are additional to these groups are passed on to the next stage as they are.

Next we take groups of 3 rows one at a time. If there are 3 wires in any column of this group, we place a full adder. The sum output of this adder goes to the same column in the next stage. The carry output of the adder goes to the column with next higher weight. If a column has two wires, it is reduced with a half adder. Again the sum output goes to the same column in the next stage and the carry wire joins the column with higher weight. If a column has a single wire, it is passed through to the next stage.

This is repeated for all groups of 3 rows.

When all groups of three rows have been reduced this way, we count wires at all weights and continue this procedure till no weight has more than two wires.

At the end, many contiguous bits at the least significant end will have a single wire. These are carried through to the result. For bits which have two wires, one is allocated to one word and the other to another word, and these two words are added using a fast adder.

3.7.2 Wallace Multiplier Example

Consider a multiplier for 4X4 bits. Partial products are generated in parallel and the number of wires at each weight are shown in the table on the right.

We shall use this example to introduce the "dot convention" used for representing wire reduction in many multipliers.

Bit	Terms	Wires
0	a0b0	1
1	a0b1, a1b0	2
2	a0b2, a1b1, a2b0	3
3	a0b3, a1b2, a2b1, a3b0	4
4	a1b3, a2b2, a3b1	3
5	a2b3, a3b2	2
6	a3b3	1

4X4 Wallace Multiplier: First Reduction

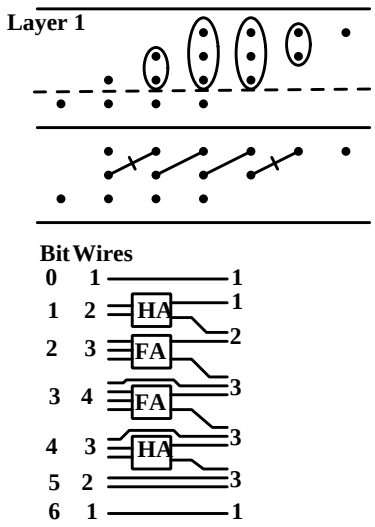


Figure 3.8: First reduction stage of a 4x4 Wallace multiplier

Figure 3.8 shows a 4x4 Wallace multiplier. This multiplier has 4 rows of partial products. The upper three rows are grouped together and can have 1, 2 or 3 wires. the bottom row is just passed through to the next stage.
Each dot in the upper diagram represents a bit at its respective weight. When we place a full adder to reduce a column, a dot representing the sum is placed at the same weight in the next stage. A dot representing carry is placed at the next higher weight. These two dots are joined by a line to show that these two are the results from a full adder. Similarly, when we use a half adder, we place a dot in the same column for the sum and a dot in the next higher weight column for the carry. These are joined with a crossed line to show that these are from a half adder. Bits which are passed through to the next stage are represented by dots not

connected to any line.
After grouping the original 4 rows of partial products in groups of 3 and 1:
Bit 0 has a single wire: which is passed through.
Bit 1 has 2 wires: which are fed to a half adder.
Bits 2 and 3 have 3 wires each, which are fed to full adders.
Bit 4 has 2 wires (in the group of 3), which are fed to a half adder.
Bit 5 has 1 wire, which is passed through.

4X4 Wallace Multiplier: Second Reduction

After first reduction, there are three rows of wires (at bits 3, 4 and 5). So we need another reduction stage. The three rows form a group and there are no passed through rows in this stage. Bits 0 and 1 have single input wires, which are passed through.

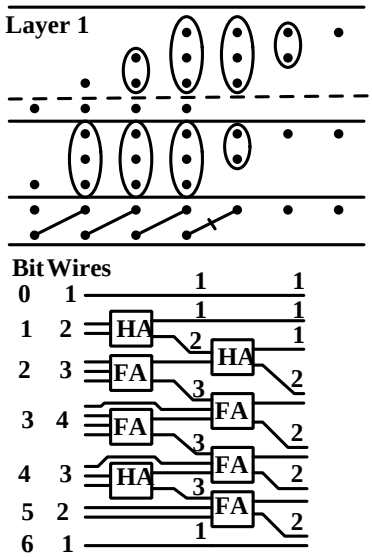


Figure 3.9: Second reduction stage of a 4x4 Wallace multiplier

Bit 2 has 2 wires. These are reduced using a half adder. The sum goes as the only output wire at this bit, while the carry goes to bit 3.
Bits 3, 4 and 5 have 3 input wires each.These are reduced using full adders. Thus these bits have two wires at their outputs – one from the sum of their full adder and the other from the carry produced by the adder at the lower weight.
Bit 6 has one input wire. This is joined by the carry of the adder at bit 5 to produce 2 output wires.

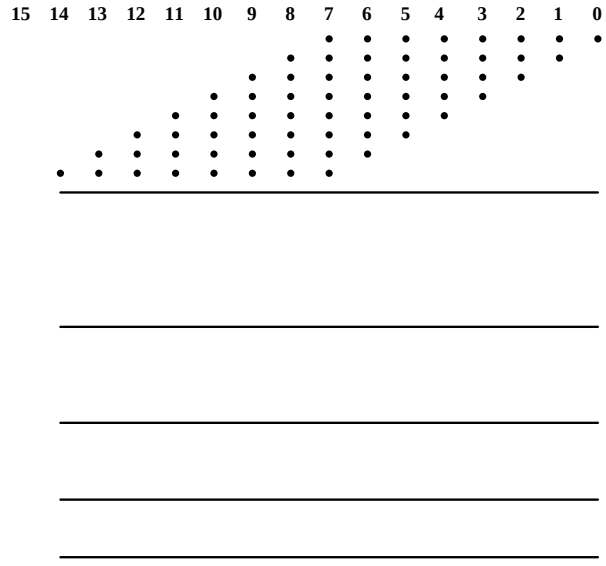
Final Addition

After the second layer, no weight has more than 2 wires. Single wires at bits 0, 1 and 2 are fed through to the output.

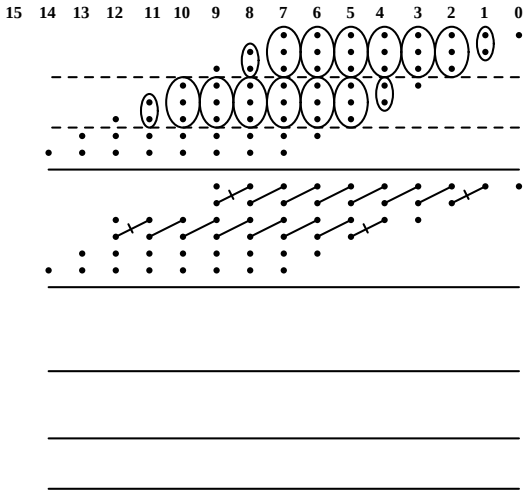
- A fast conventional adder is used to add the 2 bits each at bits 3, 4, 5 and 6.
- Notice that we do not need a full width fast adder. This is because the half adders at low weights have already rippled the carry while the rest of weights were being reduced.
- This makes the final adder smaller and faster.

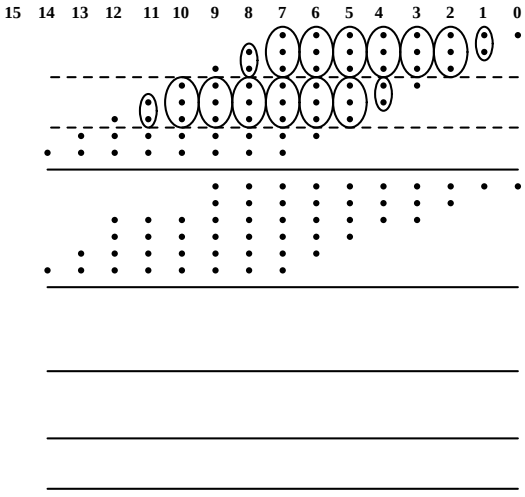
3.7.3 Redundant MSB in large Wallace Multipliers

The reduction scheme as described above sometimes produces a redundant most significant bit. The result is still correct and the redundant bit will always be zero. To see this effect, let us apply the above scheme to an 8x8 multiplier.

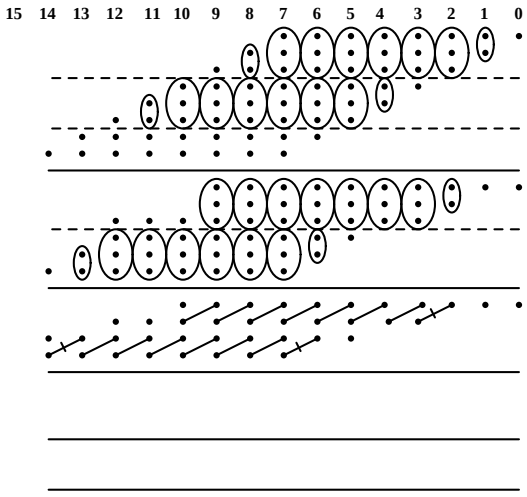


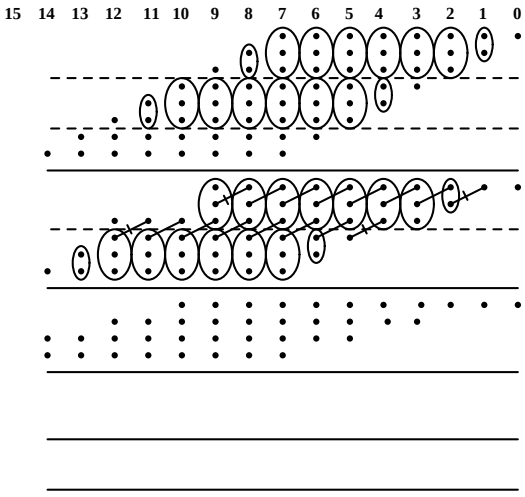
The eight rows of partial products are divided into groups 3, 3 and 2. The last two rows are just passed through to the next stage.



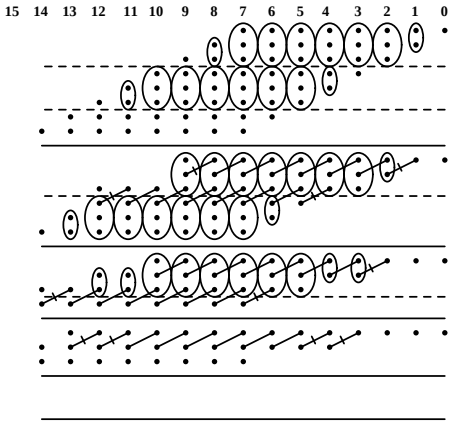


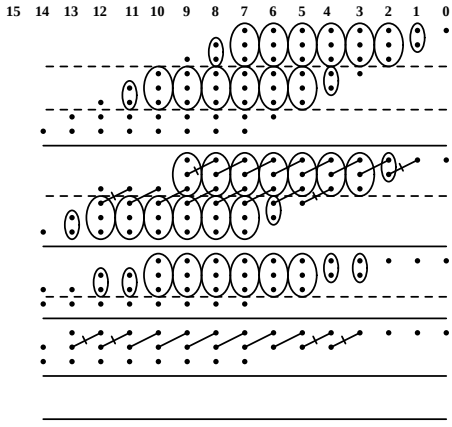
Taking each group of 3 rows, we place a full adder wherever we find three dots and a half adder where there are two dots. Single dots are passed through. This reduces the output rows to 6.



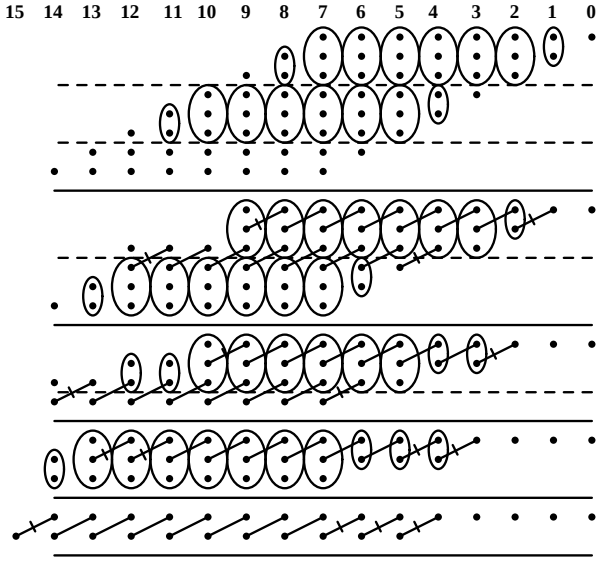


The six rows are divided into two groups of three each and we place full and half adders as shown in the figure on the left above. This reduces the number of rows to four.





The four rows output from the previous stage are grouped as 3 and 1 as shown in the figure on the left. The fourth row is just passed through, while the group of 3 rows is reduced by full and half adders. we are now left with 3 rows of output wires which need to be reduced to two by the next stage.



The 3 rows of input wires to the last stage form a single group and are reduced by full and half adders to 2 rows as shown in the figure above.

As one can see, there are two bits at bit-14, which can produce a carry during the final addition. When this carry is added to the bit at bit-15, it could produce another carry which will go to bit-16. This would be an extra bit. (Bits 0 to 16 will be 17 bits). In practice, 17 bits will not be produced, as multiplying 8 bit operands should generate at the most a 16 bit result.

3.7.4 Avoiding the Redundant MSB in Wallace Multipliers

One can avoid the redundant bit by modifying the reduction scheme:
We treat all wires in a column as equivalent. (No groups of 3 rows).
As long as there are 3 or more wires, make bunches of 3 wires and send each to a full adder. Now we can be left with 0, 1 or 2 wires. There is nothing to do for 0 wires left. If one wire is left, it is passed through to next layer.

When two wires are left, we have a more complex decision to take. For this, we first need to define the capacity of a reduction layer.

Wire capacity of a reduction layer

We define the capacity of a layer as the maximum number of wires it can accommodate. How can we determine it?

We know that the final reduction layer should have no more than 2 wires. Now we can work **backwards** from the final layer to the first. Let d_j represent the maximum number of wires for any weight in layer j, where $j = 1$ for the final adder. (Thus $d_1 = 2$). The maximum number of wires which can be handled in layer j+1 (from the end) is the integral part of $(3/2)d_j$ with $j = 1$ for the final adder. Thus $d_1 = 2$. We go up in j, till we reach a number which is just greater than or equal to the largest bunch of wires in any weight. The number of reduction layers required is this $j_{final} - 1$. Capacities of layers starting from last layer and moving towards the top are 2, 3, 4, 6, 9, 13, 19 ....

Reduction of two wires

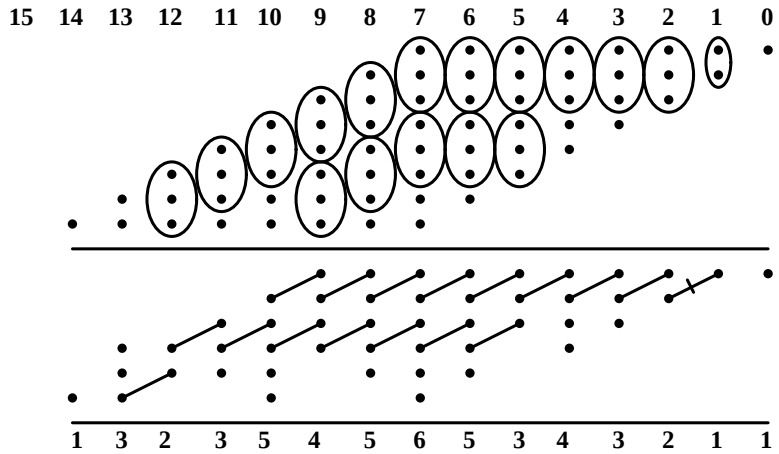
Now we can define the policy for reduction of 2 left over wires after deploying the maximum number of full adders.

- If all columns at the right have a single wire, we reduce the two wires using a half adder. (This helps in reducing the width of final adder).
- If there is a column to the right with more than one wire, we pass through the two wires to the next layer if it can accommodate these. (That is, the total number of wires do not exceed the capacity of that layer).
- If passing through the two wires would exceed the capacity of next layer, we reduce these with a half adder.

3.7.5 Wallace 8x8 Reduction without redundant MSB

We start with a maximum of 8 wires in any column. Capacity of the next layer is 6.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1
FA	0	0	0	1	1	1	2	2	2	2	2	1	1	1	0	0
Remaining	0	1	2	0	1	2	0	1	2	1	0	2	1	0	2	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
PT	0	1	2	0	1	2	0	1	2	1	0	2	1	0	0	1
Sums	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Carries to Higher bits	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Output Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1

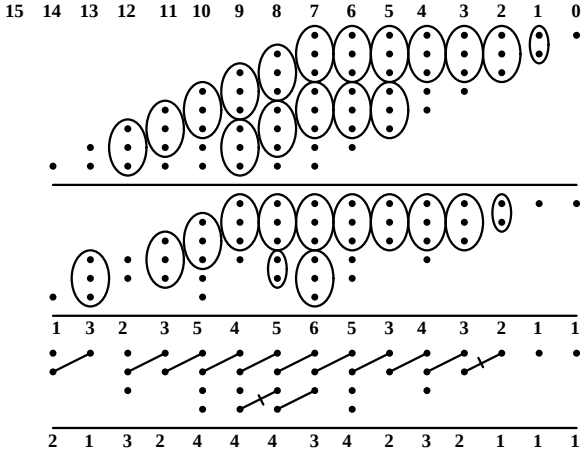


Two wires need to be handled at bits 1, 4, 7, 10 and 13. Since there is a single wire at bit 0, the two wires at bit 1 are reduced using a half adder. In all other cases, passing through the two wires will not make the number of output wires greater than 6. So we pass those through.

Second reduction layer

At this layer, maximum wires in any column is 6. Capacity of the next layer is 4.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1
FA	0	0	1	0	1	1	1	1	2	1	1	1	1	0	0	0
Remaining	0	1	0	2	0	2	1	2	0	2	0	1	0	2	1	1
HA	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
PT	0	1	0	2	0	2	1	0	0	2	0	1	0	0	1	1
Sums	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Carries to Higher bits	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Output Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1

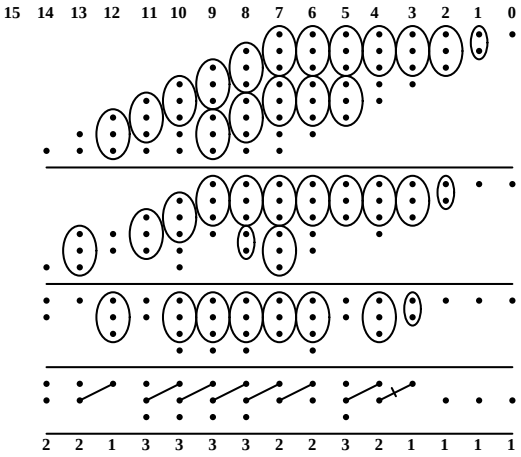


This time, we have to handle 2 wires at bits 2, 6, 8, 10 and 12. Since there is a single wire at bits 0 and 1, we reduce the two wires at bit 2 using a half adder. Two wires at bit 6 can be passed through because the number of output wires will not exceed 4. However, at bit 8, we anticipate two carry wires from bit 7 and a sum wire from the bunch of 3 wires at bit 8 itself. Passing through the 2 wires will make the number of output wires 5, which will exceed the capacity of the next layer (4). Therefore the 2 wires left at bit 8 must be reduced using a half adder. Two wires at bits 10 and 12 can be passed through because the number of output wires do not exceed 4.

Third reduction layer

Capacity of next layer is 3.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1
FA	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	0
Remaining	0	2	1	0	2	1	1	1	0	1	2	0	2	1	1	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
PT	0	2	1	0	2	1	1	1	0	1	2	0	0	1	1	1
Sums	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Carries to Higher bits	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Output Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1

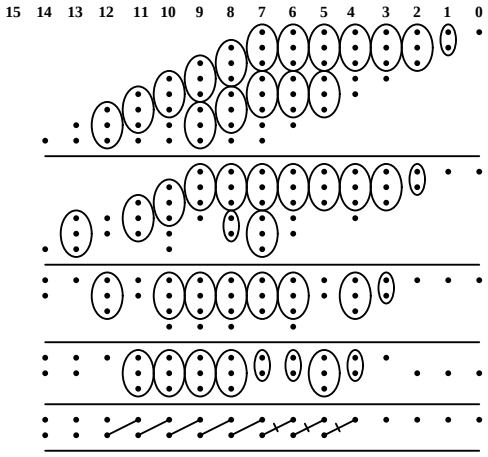


This time we have to handle two wires at bits 3, 5, 11 and 14. Since there is a single wire at bits 0, 1 and 2, we should reduce the 2 wires at bit 3 using a half adder. Passing through the 2 wires at bits 5, 11 and 14 does not exceed 3 output wires, so these are passed through.

Final reduction layer

Capacity of next layer is 2.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1
FA	0	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0
Remaining	0	2	2	1	0	0	0	0	2	2	0	2	1	1	1	1
HA	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0
PT	0	2	2	1	0	0	0	0	0	0	0	0	1	1	1	1
Sums	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Carries to Higher bits	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Output Wires	0	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1



This time there are two wires at bit positions b4, b6, b7, b13 and b14. As before, we should reduce b4 wires using a half adder because there are single wires at all less significant positions. Passing through the 2 wires at bits 6 and 7 would produce 3 output wires, exceeding the target of 2 wires. Therefore here too we place half adders. However, there is no incoming carry at bits b13 and b14 and so we can pass the 2 wires through.

Thus we have reached two wires without generating a wire at b15. There are two wires at b14 and if these produce a carry, it will go to b15. However, there is no redundant b16 now.

We have reduced the number of wires at all bit positions to ≤ 2 without generating a bit at b15.
There are two wires at b14 and if these produce a carry, it will go to b15 and there is no redundant b16.

3.7.6 Dadda Multipliers

Dadda multipliers are very similar to Wallace multipliers and use the same 3 stages:

1. Generate all bits of the partial products in parallel.
2. Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
3. For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

   Add these two numbers using a fast adder of appropriate size.

The difference is in the reduction stage.

Wallace multipliers reduce as soon as possible, while Dadda multipliers reduce as late as possible. Dadda multipliers plan on reducing the final number of wires for any weight to 2 with as few and as small adders as possible. We determine the number of layers required first, beginning from the **last** layer, where no more than 2 wires should be left. The number of layers in Dadda multipliers is the same as in Wallace multipliers.

We work back from the final adder to earlier layers till we find that we can manage all wires generated by the partial product generator.

We know that the final adder can take no more than 2 wires for each weight.

Let d_j represent the maximum number of wires for any weight in layer j, where $j = 1$ for the final adder. (Thus $d_1 = 2$).
The maximum number of wires which can be handled in layer j+1 (from the end) is the integral part of $3/2d_j$.

We go up in j, till we reach a number which is just greater than or equal to the largest bunch of wires in any weight. The number of reduction layers required is this final j -1.

Wire Reduction in Dadda Multipliers

At each layer we know the maximum number of wires which should be left for the next layer.

For each weight, we place the least number of smallest adders, such that the wires going out to the next layer do not exceed the maximum number of wires it can handle.

At each weight, we must consider all the incoming wires, as well as the carry wires which will be transferred from the less significant weights in the next layer.

That is why we must begin with the lowest weight and go towards higher weights in each layer.

Dadda Multiplier: Example

Take the example of 4-bit by 4-bit multiplication multiplying a3a2a1a0 by b3b2b1b0. As before, partial products are generated in parallel and we have the following wires:

Weight	Terms	Wires
1	a0b0	1
2	a0b1, a1b0	2
4	a0b2, a1b1, a2b0	3
8	a0b3, a1b2 a2b1 a3b0	4
16	a1b3, a2b2, a3b1	3
32	a2b3, a3b2	2
64	a3b3	1

Number of Reduction Layers

Maximum no. of wires for any weight in this example is 4. $d_1 = 2$, $d_2 = 3$, $d_3 = 4$. So we need 2 layers of reduction.

The first reduction layer should reduce the number of wires at any weight to a maximum of 3. The second layer will then reduce these to a maximum of 2 wires.

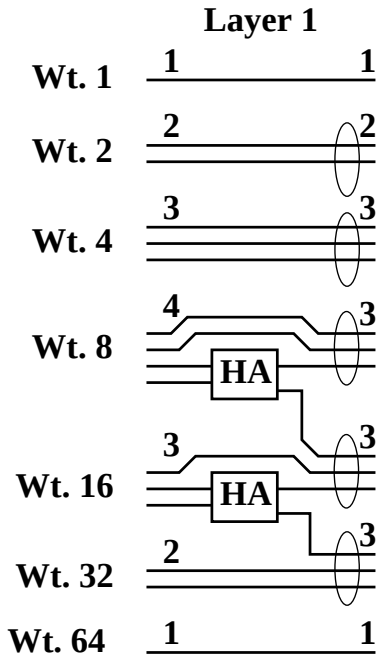
At each reduction layer, we scan from less significant weights to more significant ones, keeping track of additional carry wires which will be transferred *at the output* from lower weights to higher ones.

First Reduction Layer

Weight	Wires
1	1
2	2
4	3
8	4
16	3
32	2
64	1

- Weights 1, 2 and 4 have 3 or less wires. These are passed through.
- Weight 8 has 4 wires. No carry is anticipated from lower weights. A half adder is used to reduce the output wires to 3. (Half Adder Sum + 2 wires passed through).
- Weight 16 has 3 wires, but we anticipate a carry from the adder at weight 8. So we should reduce by 1 to keep the total **output** wires to 3. So this column is also reduced using a half adder.

After First Reduction



- Wt.1 has the single wire which was fed through.
- Wt.2 has 2 fed through wires.

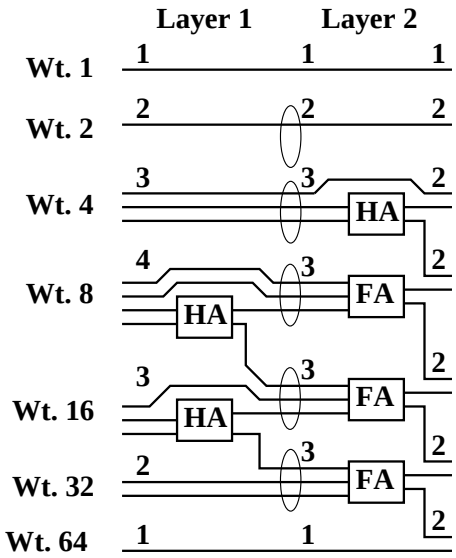
- Wt.4 has 3 wires: all passed through.
- Wt.8 has 3 wires: sum of the half adder at wt.4, and 2 passed through.
- Wt.16 has 3 wires: carry of wt. 8, sum of half adder at 16 and 1 passed through.
- Wt.32 has 3 wires: carry of wt. 16 and 2 passed through.
- Wt.64 has 1 fed through wire.

Second Reduction

In the second layer, we should leave no more than 2 wires at any weight, as this is the last stage.

- As before, we anticipate the number of carry wires transferred from the lower weight when planning reduction using half or full adders.
- In Dadda multipliers, we use minimum hardware during reduction. So the smallest adder which will reduce the output wires to 2 will be used.
- At the lowest weights, if the number of wires is less than or equal to 2, we just pass these through.
- So the single wire at Wt. 1, and the 2 wires at Wt. 2 are just fed through.

4X4 Dadda Multiplier: Second Reduction

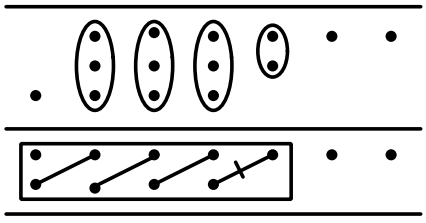


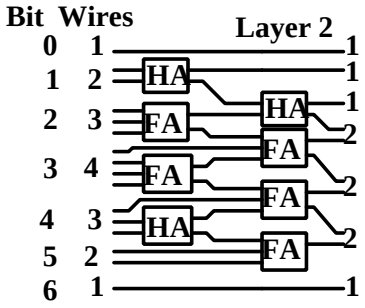
- 3 wires at Wt. 4 are reduced to 2 by a half adder: No carry in expected.

- Wt. 8 has 3 input wires. Carry will arrive from Wt. 4. Reduced using a Full adder.
- Wt. 16 has 3 input wires. Carry will arrive from Wt. 8. Reduced using a full adder.
- Wt. 32 has 3 input wires. Carry will arrive from Wt. 16. Reduced using a full adder.
- Wt. 64 has 1 input wire. Carry will arrive from Wt. 32, making it 2 output wires, which will be fed through.

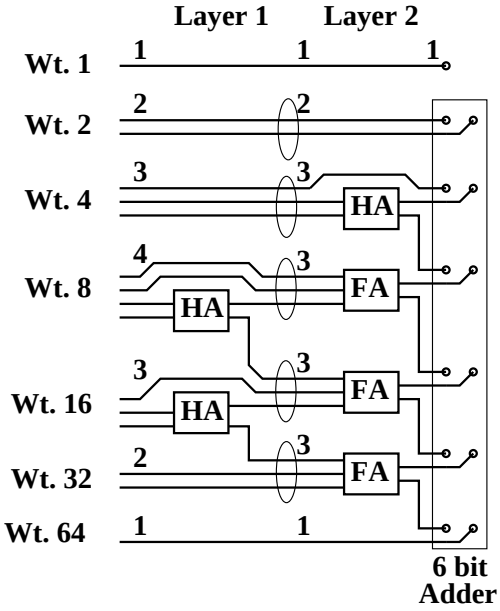
4X4 Multipliers: Final Addition

Wallace Multiplier





Dadda Multiplier



Notice that we have used only 3 Full Adders and 3 Half Adders during reduction, whereas Wallace multiplier requires 5 Full Adders and 3 Half Adders.
However, we require a 6-bit final adder for Dadda multiplier, whereas Wallace multiplier needs only a 4 bit final adder.

3.8 Multiply and Accumulate circuits

A common operation required in data processing is evaluation of quantities of the type $\sum c_i X_i$. This requires that the value of the product at each term should be added to the current value of the sum to get its new value. For implementing such calculations, it would be useful if we can have a circuit in hardware which will efficiently multiply two operands as well as add a third. Notice that this third operand will be of the size of the product. Such circuits are called Multiply and Accumulate circuits.

Multiply and Accumulate circuits are easy to implement because during multiplication, we are anyway adding multiple bits in a column. The accumulator just provides an additional bit in every column and this can be easily added along with the partial product bits in a carry-save fashion using any of the wire reduction techniques described above.

Consider for example a Multiply and Accumulate circuit which multiplies two 8 bit operands and adds a 16 bit operand to it. Let us apply the Wallace wire reduction technique to compute $A \times B + C$, where A and B are 8 bit operands, while C is a 16 bit operand. The product computation will generate wires at bit positions 0 through 14 in the usual trapezoidal shape. Thus the number of wires from partial sums of the multiplier will be:

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Partial Sum Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1

The operand to be added just provides one additional wire at each bit position from 0 to 15. Thus the number of wires at each bit position becomes

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wire count	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2

We can now proceed to reduce these wires as a Wallace tree till we are left with no more than 2 wires at each position. Finally, we shall add these two groups using a fast traditional adder. The tables below show the wire reduction at each stage:

Stage 1: Max. wires:9, capacity of next stage = 6

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2
Full Adders	0	0	1	1	1	2	2	2	3	2	2	2	1	1	1	0
Half Adders	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
Pass Through	1	2	0	1	2	0	1	0	0	2	1	0	2	1	0	0
Output Wires	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1

Stage 2: capacity of next stage = 4

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1
Full Adders	0	1	0	1	1	1	2	2	1	2	1	1	1	1	0	0
Half Adders	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
Pass Through	1	0	2	0	2	1	0	0	0	0	2	0	1	0	0	1
Output Wires	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1

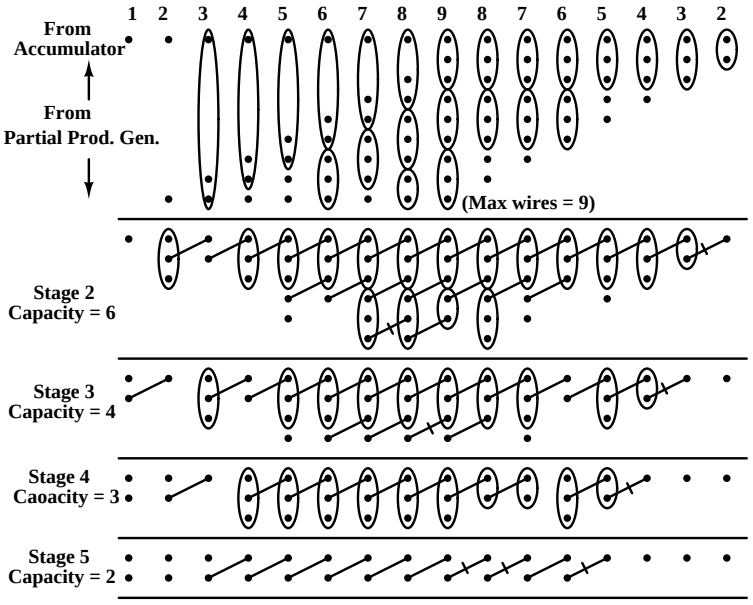
Stage 3: capacity of next stage = 3

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1
Full Adders	0	0	1	0	1	1	1	1	1	1	1	0	1	0	0	0
Half Adders	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Pass Through	2	1	0	2	1	1	1	1	1	0	1	2	0	0	1	1
Output Wires	2	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1

Stage 4: capacity of next stage = 2

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	2	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1
Full Adders	0	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0
Half Adders	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
Pass Through	2	2	1	0	0	0	0	0	0	0	0	0	0	1	1	1
Output Wires	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1

The dot diagram for this scheme is shown below:



Finally, we need a 12 bit fast adder to get the final result.

The complexity of this circuit is not much more than a plain 8x8 multiplier and the time taken to produce the results is much less than the time taken to multiply first and then to add. This is because the latter requires two additions in which the carry ripples while the Multiply and Accumulate circuit requires only one such addition.

3.9 Serial Multipliers

Often, we need multipliers which have very low complexity or very low power consumption and speed is not very important. Serial multipliers are a good option in such cases.

Low complexity multipliers can be bit serial or row serial. Bit serial multipliers require $m \times n$ clocks for completing an $m \times n$ multiplication. Row serial multipliers require only n steps, but we require m full adders rather than just one.

3.9.1 Bit Serial Multipliers

For serial multiplication, partial product bits need not be generated in parallel. These can be generated as and when these are required. Each bit of the multiplier needs to be ANDed

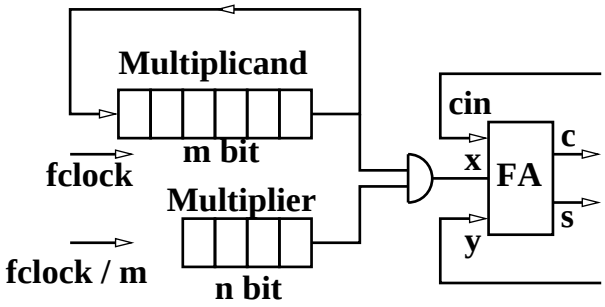
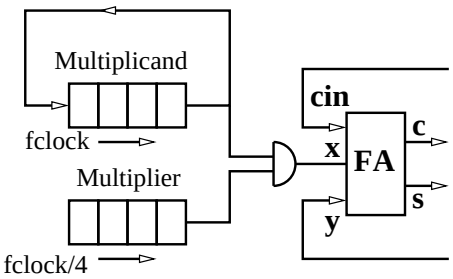


Figure 3.10: Partial product generation for bit-serial multiplication

with each bit of the multiplicand. This requires that all multiplicand bits be presented one after the other every time a new bit from the multiplier is taken up. This can be managed by using a re-circulating shift register for the multiplicand, which is clocked at a rate which is m times faster than the clock supplied to the multiplier shift register. The inputs y and Cin to the full adder have to be appropriately selected and timed to generate the correct product.

Consider a 4×4 bit serial multiplier.

The x input to the Full Adder appears in the following order:



ck	x	ck	x	ck	x	ck	x
0	a0b0	4	a0b1	8	a0b2	12	a0b3
1	a1b0	5	a1b1	9	a1b2	13	a1b3
2	a2b0	6	a2b1	10	a2b2	14	a2b3
3	a3b0	7	a3b1	11	a3b2	15	a3b3

We need additions as follows:

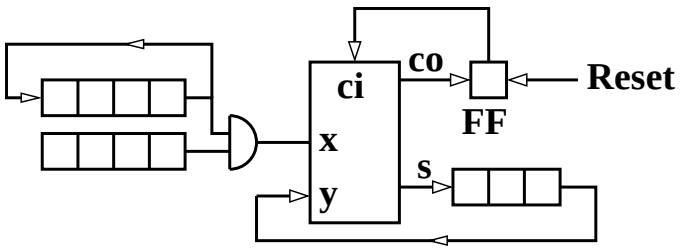
			a3	a2	a1	a0
		×	b3	b2	b1	b0
			a3b0	a2b0	a1b0	a0b0
		a3b1	a2b1	a1b1	a0b1	
	a3b2	a2b2	a1b2	a0b2		
a3b3	a2b3	a1b3	a0b3			

Let us put the arrival time of terms in parentheses next to each term.

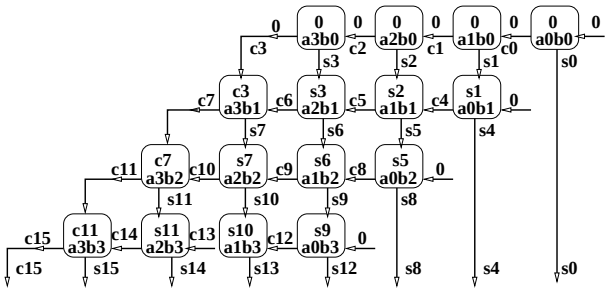
			a3	a2	a1	a0
		×	b3	b2	b1	b0
			a3b0(3)	a2b0(2)	a1b0(1)	a0b0(0)
		a3b1(7)	a2b1(6)	a1b1(5)	a0b1(4)	
	a3b2(11)	a2b2(10)	a1b2(9)	a0b2(8)		
a3b3(15)	a2b3(14)	a1b3(13)	a0b3(12)			

It is clear that for all additions, the earlier terms have to wait for 3 clock cycles before the later terms arrive.

We can manage this by putting a 3 bit shift register at the sum output and presenting the delayed output at the 'y' input of the full adder. The carry output can be added immediately in the next clock, since it should go to the next column to its left. A 3 clock delay for sum and a 1 clock delay for carry leads to the following circuit.



Unfortunately, it does not work as shown!
We need to take care of a few exceptions. Let us look at all the exceptions in detail. In the figure below, each box contains the y and x inputs presented to the adder. The sum and carry terms are indexed by the clock cycle in which these were generated. For example, $c7$ is the carry generated in the 7th clock cycle. Notice that in the first four cycles, 0 is being added to partial products and therefore the carry is always 0.



- At clocks 0, 4, 8 and 12, carry input should be forced to 0.
- At clocks 7, 11 and 15, the adder y input should receive carry terms (c3, c7 and c11) instead of sum terms (s4, s8 and s12).
- The sum terms s4, s8 and s12 should be taken out as result bits and not inserted in the 3 bit delay shift register.

We can take care of these exceptions by inserting the carry FF output (which is a 1 clock delayed version of cout) in the 3-bit shift register instead of the sum terms. Thus C3 (which is always 0), C7 and C11 will emerge from the shift register at clocks 7, 11 and 15 respectively and will be added to the correct partial product bits. The corresponding sum terms should be taken out as result bits.

3.9.2 Bit Serial Multiplier: Implementation

With exception handling at the end of rows, the serial multiplier will work. Notice the changes

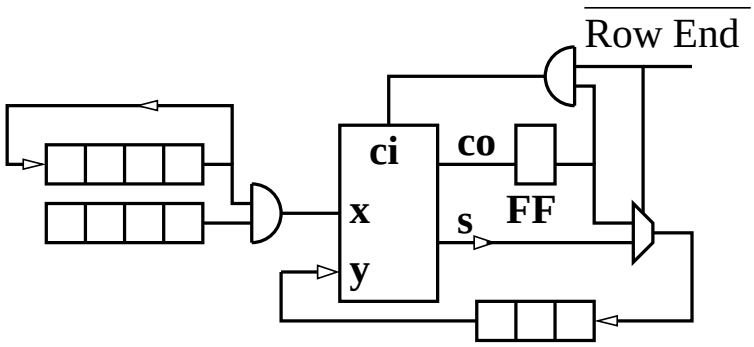


Figure 3.11: 4x4 bit serial multiplier

from the earlier suggested circuit:

- Carry input is forced to 0 at row ends.
- The mux normally inserts the sum into the shift register. However, at row ends, it inserts the delayed carry output.
- The sum terms at row ends are taken out as the low bits of the product.

- One can add another shift register at the output to collect these.
- The 2 more significant bits of the shift register and the last sum and carry provide the high bits of the product at the end.

3.9.3 Row Serial multipliers

We need not reduce the complexity all the way down to a single adder for serial multipliers. We could have a row of n adders in parallel performing n serial additions. We can use n full adders arranged in n columns. The carry output of previous addition is retained at the same column. The sum from the previous addition *in the left column* is brought to this column by a shift operation. This sum has the same weight as the carry generated during the previous clock in this column. These two are added to the partial product bit for this column.

Taking the example of 4×4 multiplication, we are trying to perform the following operations:

				a3	**a2**	**a1**	**a0**
				×	**b3**	**b2**	**b1**	**b0**
				a3b0	**a2b0**	**a1b0**	**a0b0**
			a3b1	**a2b1**	**a1b1**	**a0b1**	
		a3b2	**a2b2**	**a1b2**	**a0b2**		
	a3b3	**a2b3**	**a1b3**	**a0b3**			

The addition process using 4 adders is represented in Figure 3.12. Notice that the same 'a' term is used for generating partial products for a given column. The 'b' term has to be shifted right every time to generate the right partial product bit.

Sums have to be shifted right to be added to the carry of the previous addition in the same column. 4 additional clock cycles will be required to ripple the carry in the last addition. During these, the partial product bits will be 0.

This scheme can be implemented as follows:

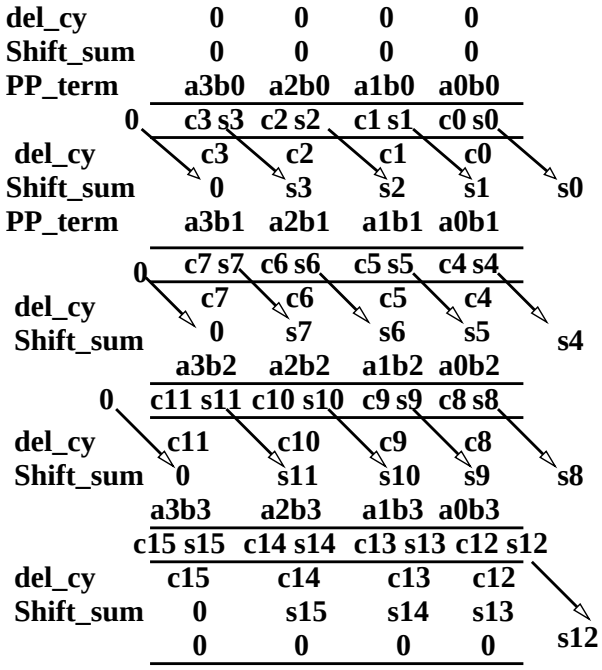


Figure 3.12: 4x4 row serial multiplication

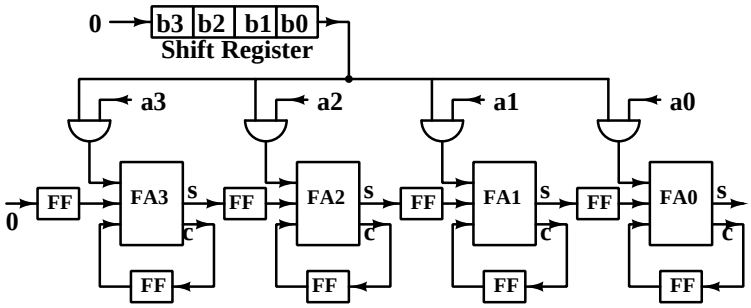


Figure 3.13: 4x4 row serial multiplier