SUBROUTINE NESTING AND THE PROCESSOR STACK:-

A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last on generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the processor stack. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

PARAMETER PASSING:-

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack used for saving the return address.

The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called passing by reference. The second parameter is passed by value, that is, the actual number of entries, n, is passed to the subroutine.

THE STACK FRAME:-

Now, observe how space is used in the stack in the example. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private workspace for

the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame.

In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R0 and R1 need to be saved because they will also be used within the subroutine.

The pointers SP and FP are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine. We begin by assuming that SP point to the old top-of-stack (TOS) element in fig b. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP.

Thus, the first two instructions executed in the subroutine are

```
Move    FP, -(SP)
Move    SP, FP
```

After these instructions are executed, both SP and FP point to the saved FP contents.

```
Subtract    #12, SP
```

Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the fig.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

```
Add     #12, SP
```

And pops the saved old value of FP back into FP. At this point, SP points to the return address, so the Return instruction can be executed, transferring control back to the calling program.

1.19 Logic instructions

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described. It is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

Not dst

SHIFT AND ROTATE INSTRUCTIONS:-

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a number, we use an arithmetic shift, which preserves the sign of the number.

Logical shifts:-

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a logical left shift instruction is

LShiftL count, dst

Rotate Operations:-

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Two versions of both the left and right rotate instructions

UNIT-2
ARTHEMATIC AND BASIC PROCESSING UNIT

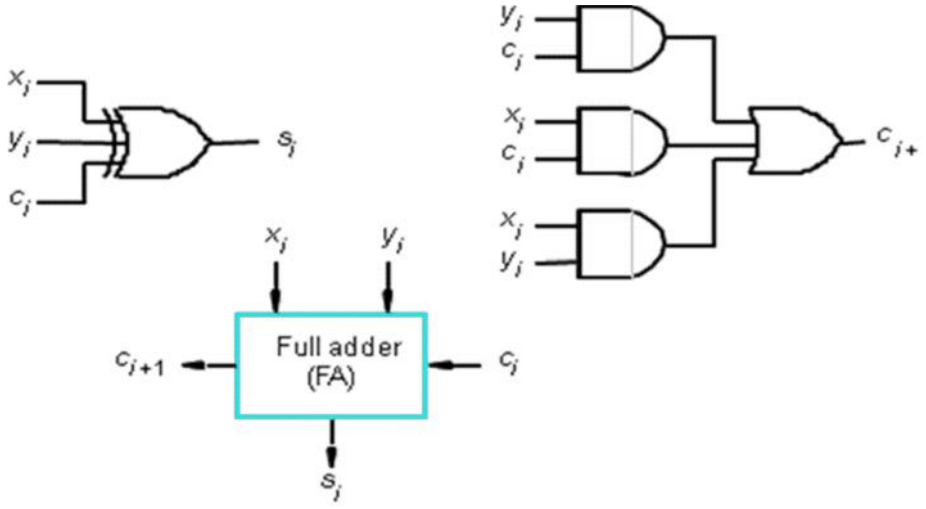
2.1ADDITION AND SUBTRACTION OF SIGNED NUMBERS:

In figure-1, the function table of a full-adder is shown; sum and carryout are the outputs for adding equally weighted bits x_i and y_i, in two numbers X and Y. The logic expressions for these functions are also shown, along with an example of addition of the4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use **ci,** to represent the carry-in to the i^{th} stage, which is the same as the carryout from the (i - 1) th stage.

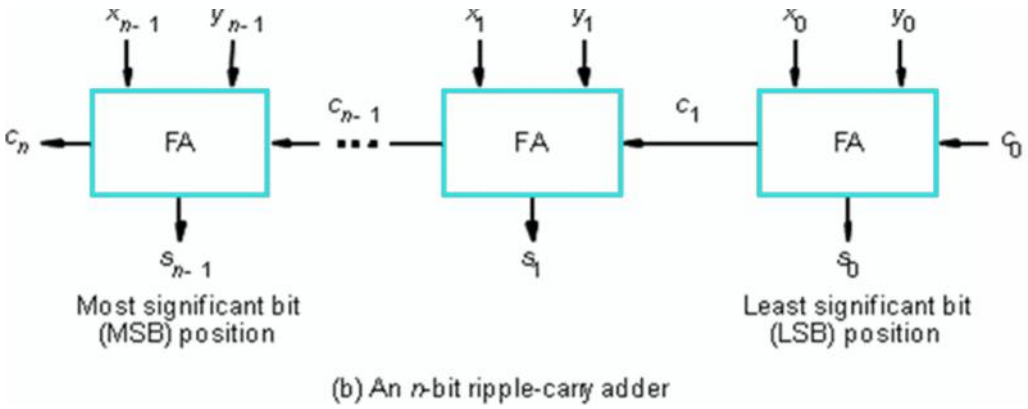
The logic expression for s_i in Figure-1 can be implemented with a 3-input XOR gate. The carryout function, c_{i+1} is implemented with a two-level AND-OR logic circuit. A convenient symbol for the complete circuit for a single stage of addition, called a full adder (FA), is as shown in the figure-1a.

A cascaded connection of such n full adder blocks, as shown in Figure 1b, forms a parallel adder & can be used to add two n-bit numbers. Since the carries must propagate, or ripple, through this cascade, the configuration is called an n-bit ripple-carry adder.

The carry-in, Co, into the least-significant-bit (LSB) position [Ist stage] provides a convenient means of adding 1 to a number. Take for instance; forming the 2's-complement of a number involves adding 1 to the 1's-complement of the number. The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are *kn* bits long, as shown in Figure-1c.



(a) Logic for a single stage



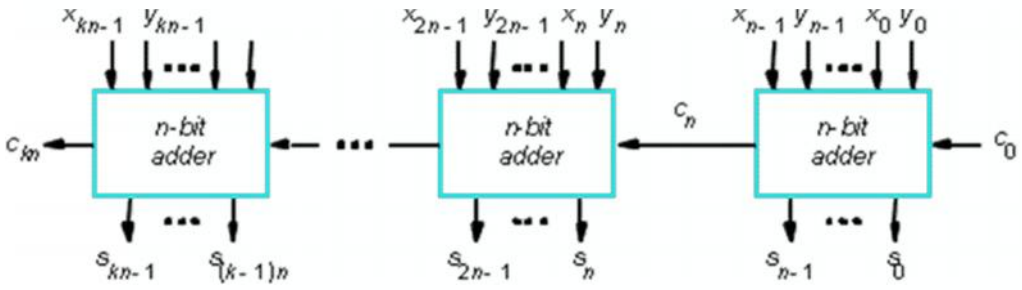


FIG: Addition of binary vectors.

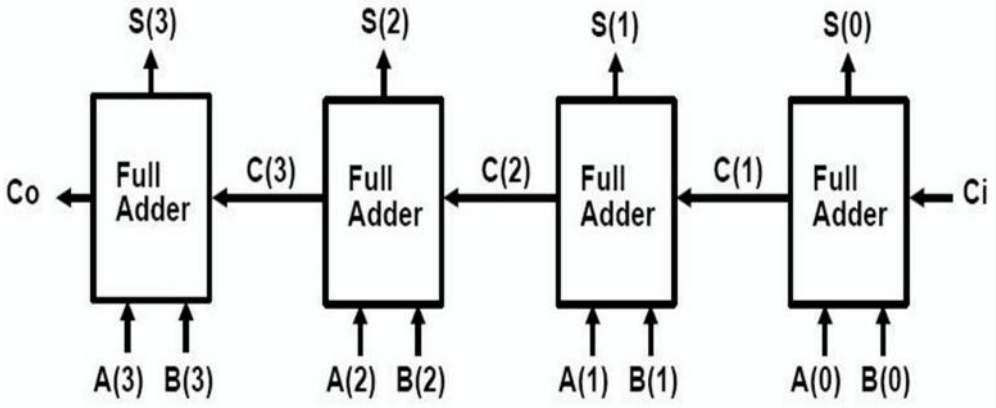


FIG: 4 - Bit parallel Adder.

2.2 DESIGN OF FAST ADDERS:

In an *n*-bit parallel adder (ripple-carry adder), there is too much delay in developing the outputs, s_o through s_{n-1} and c_n. On many occasions this delay is not acceptable; in comparison with the speed of other processor components and speed of the data transfer between registers and cache memories. The delay through a network depends on the integrated circuit technology used in fabricating the network and on the number of gates in the paths from inputs to outputs (propagation delay). The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the network. In the case of the n-bit ripple-carry adder, the longest path is from inputs x_0, y_0, and c_0 at the least-significant-bit (LSB) position to outputs c_n and s_{n-1} at the most-significant-bit (MSB) position.

Using the logic implementation indicated in Figure-1, c_{n-1} is available in $2(n—1)$ gate delays, and s_{n-1} is one XOR gate delay later. The final carry-out, c_n is available after 2n gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure-3, all sum bits are available in 2n gate delays, including the delay through the XOR gates on the Y input. Using the implementation c_{n-1} for overflow, this indicator is available after 2_n+2 gate delays. In summary, in a

parallel adder an **n**th stage adder can not complete the addition process before all its previous stages have completed the addition even with input bits ready. This is because, the carry bit from previous stage has to be made available for addition of the present stage.

In practice, a number of design techniques have been used to implement high- speed adders. In order to reduce this delay in adders, an augmented logic gate network structure may be used. One such method is to use circuit designs for fast propagation of carry signals (carry prediction).

Carry-Look ahead Addition:

As it is clear from the previous discussion that a parallel adder is considerably slow & a fast adder circuit must speed up the generation of the carry signals, it is necessary to make the carry input to each stage readily available along with the input bits. This can be achieved either by propagating the previous carry or by generating a carry depending on the input bits & previous carry. The logic expressions for s_i (sum) and c_{i+1} (carry-out) of stage **i**th are

$$s_i = x_i \oplus y_i \oplus c_i$$

and

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Factoring the second equation into

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

we can write

$$c_{i+1} = G_i + P_i c_i$$

where

$$G_i = x_i y_i \quad \text{and} \quad P_i = x_i + y_i$$

The above expressions G_i and P_i are called carry generate and propagate functions for stage **i**. If the generate function for stage **i** is equal to 1, then $c_{i+1} = 1$, independent of the input carry, c_i. This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i **or** y_i **or** both equal to 1. Now, using Gi & Pi functions we can decide carry for ith stage even before its previous stages have completed their addition operations. All G_i and P_i functions can be formed independently and in parallel in only one gate delay after the Xi and Yi inputs are applied to an n-bit adder. Each bit stage contains an AND gate to form G_i, an OR gate to form P_i and a three-input XOR gate to form s_i. However, a much simpler circuit can be derive

by considering the propagate function $P_i = x_i$ which differs from $P_i = x_i + y_i$ only

when $x_i = y_i = 1$ where $G_i = 1$ (so it does not matter whether P_i is 0 or 1). Then, the basic diagram in Figure-5 can be used in each bit stage to predict carry ahead of any stage completing its addition.

Consider the c_{i+1}expression,

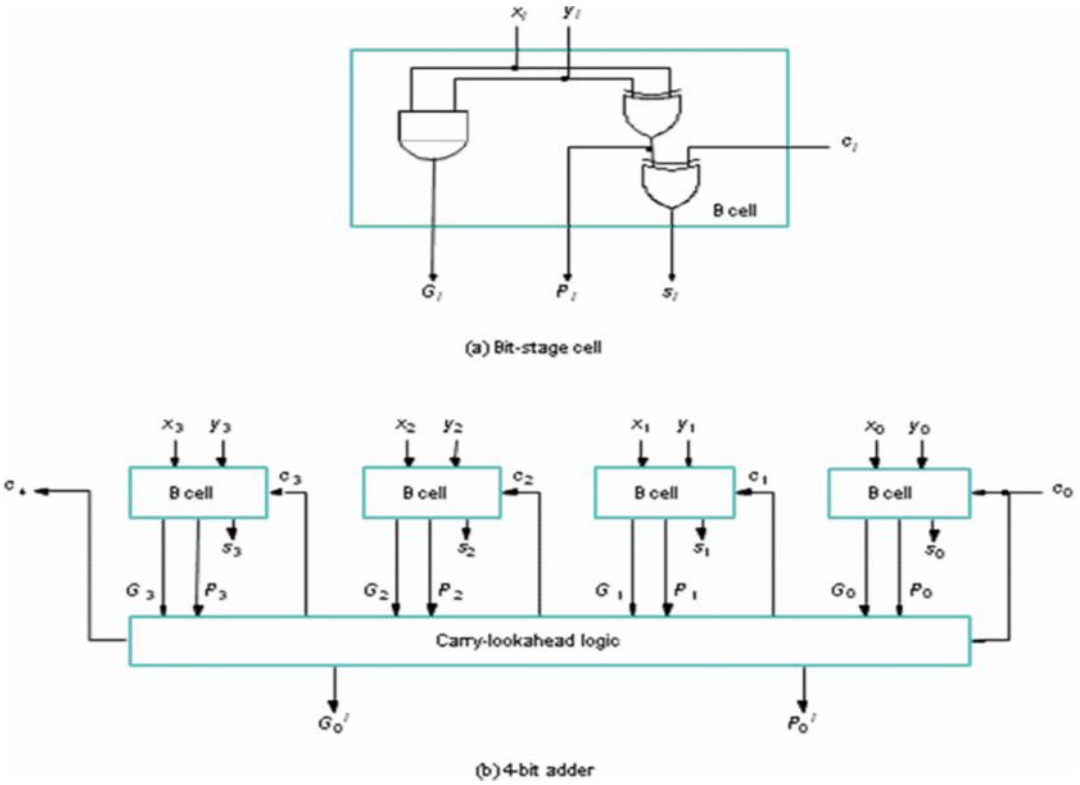
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

This is because, Ci = (Gi-1 + Pi-1Ci-1).

Further, Ci-1 = (Gi-2 + Pi-2Ci-2) and so on. Expanding in this fashion, the final carry expression can be written as below;

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 G_0$$

Thus, all carries can be obtained in three gate delays after the input signals Xi, Yi and Cin are applied at the inputs. This is because only one gate delay is needed to develop all P_i and G_i signals, followed by two gate delays in the AND-OR circuit (SOP expression) for c + After a further XOR gate delay, all sum bits are available.



Therefore, **independent of *n, the number of stages,* the n-bit *addition* process requires only four gate delays**.

Now, consider the design of a 4-bit parallel adder. The carries can be implemented as

$$c_1 = G_0 + P_0 c_0 \quad ;i = 0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0 \quad ;i = 1$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \quad ;i = 2$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \quad ;i = 3$$

The complete 4-bit adder is shown in Figure 5b where the B cell indicates Gi, Pi & Si generator. The carries are implemented in the block labeled carry look-ahead logic. An adder implemented in this form is called a *carry look ahead adder.* Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, note that a 4-bit ripple-carry adder requires 7 gate delays for *S3*(2n-1) and 8 gate delays(2n) for **c4**.

If we try to extend the carry look- ahead adder of Figure 5b for longer operands, we run into a problem of gate fan-in constraints. From the final expression for Ci+1 & the carry expressions for a 4 bit adder, we see that the last AND gate and the OR gate require a fan-in of i + 2 in generating c_{n-1}. For c_4 (i = 3)in the 4-bit adder, a fan-in of 5 is required. This puts the limit on the practical implementation. So the adder design shown in Figure
4b cannot be directly extended to longer operand sizes. However, if we cascade a number of 4-bit adders, it is possible to build longer adders without the practical problems of fan- in. An example of a 16 bit carry look ahead adder is as shown in figure 6. Eight 4-bit carry look-ahead adders can be connected as in Figure-2 to form a 32-bit adder.

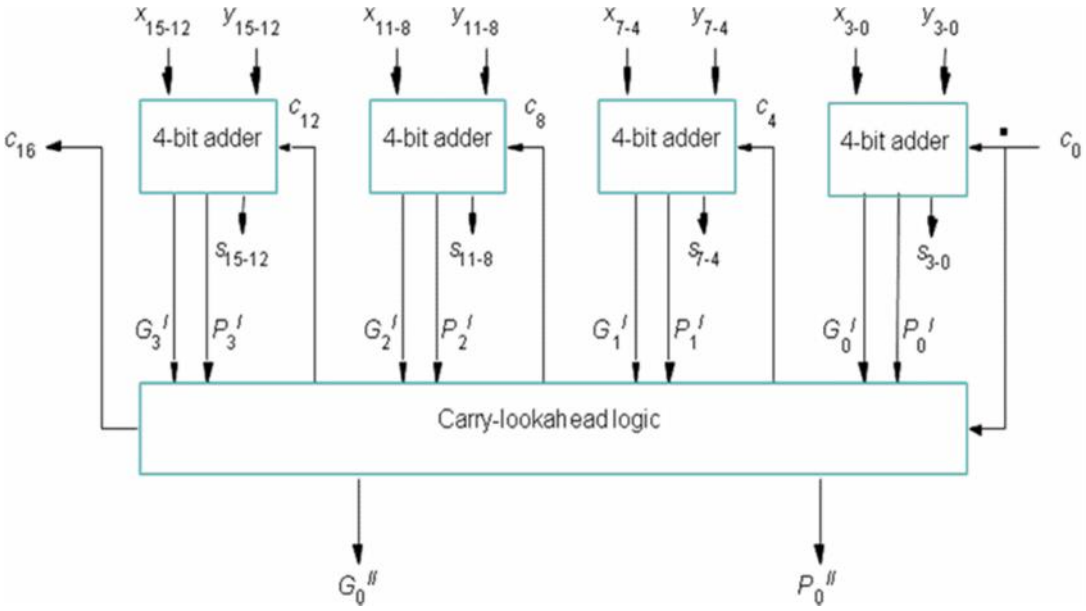


FIG: 16 bit carry-look ahead adder.

2.3 MULTIPLICATION OF POSITIVE NUMBERS:

Consider the multiplication of two integers as in Figure-6a in binary number system. This algorithm applies to unsigned numbers and to positive signed numbers. The product of two n-digit numbers can be accommodated in **2*n*** digits, so the product of the two 4-bit numbers in this example fits into 8 bits. In the binary system, multiplication by the multiplier bit '1' means the multiplicand is entered in the appropriate position to be added to the **partial product**. If the multiplier bit is '0', then 0s are entered, as indicated in the third row of the shown example.

Binary multiplication of positive operands can be implemented in a combinational (speed up) two-dimensional logic array, as shown in Figure 7. Here, **M**- indicates **multiplicand**, **Q**- indicates **multiplier** & **P**- indicates **partial product**. The basic component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit ***mj,*** is added to the incoming partial-product bit, based on the value of the multiplier bit, $\mathbf{q_i}$. For **i** in the range of 0 to 3, if q_i = 1, add the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i+ 1) & if q_i = 0, PPi is passed vertically downward unchanged. The initial partial product PPO is all 0s. PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. Since the multiplicand is shifted and added to the partial product depending on the multiplier bit, the method is referred as SHIFT & ADD

method. The multiplier array & the components of each bit cell are indicated in the diagram, while the flow diagram shown explains the multiplication procedure.

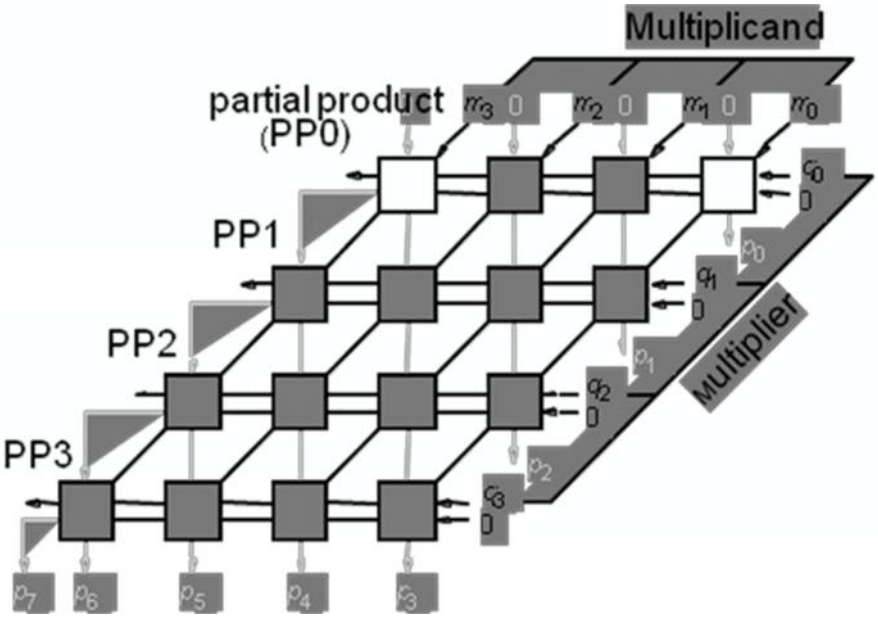


FIG-7a P7, P6, P5,…,P0 – product.

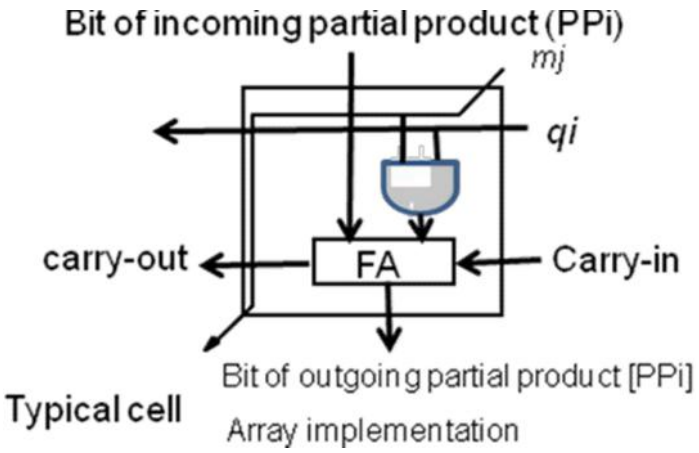
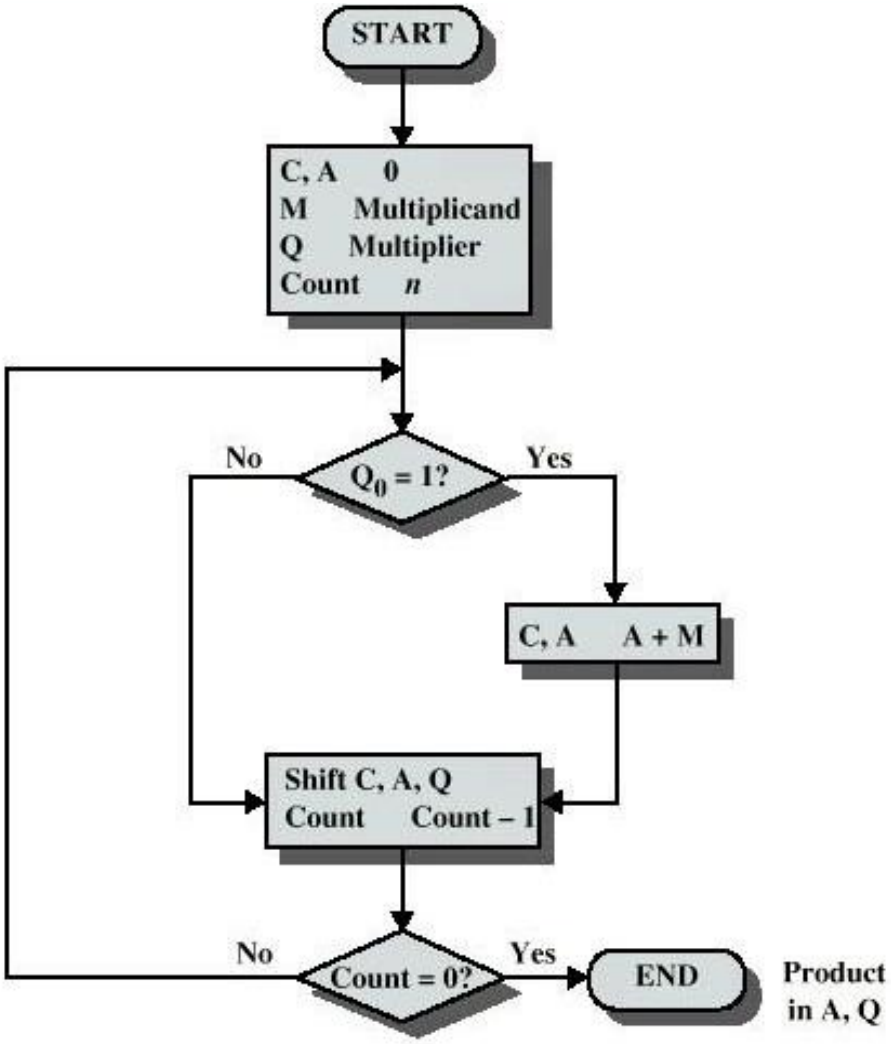


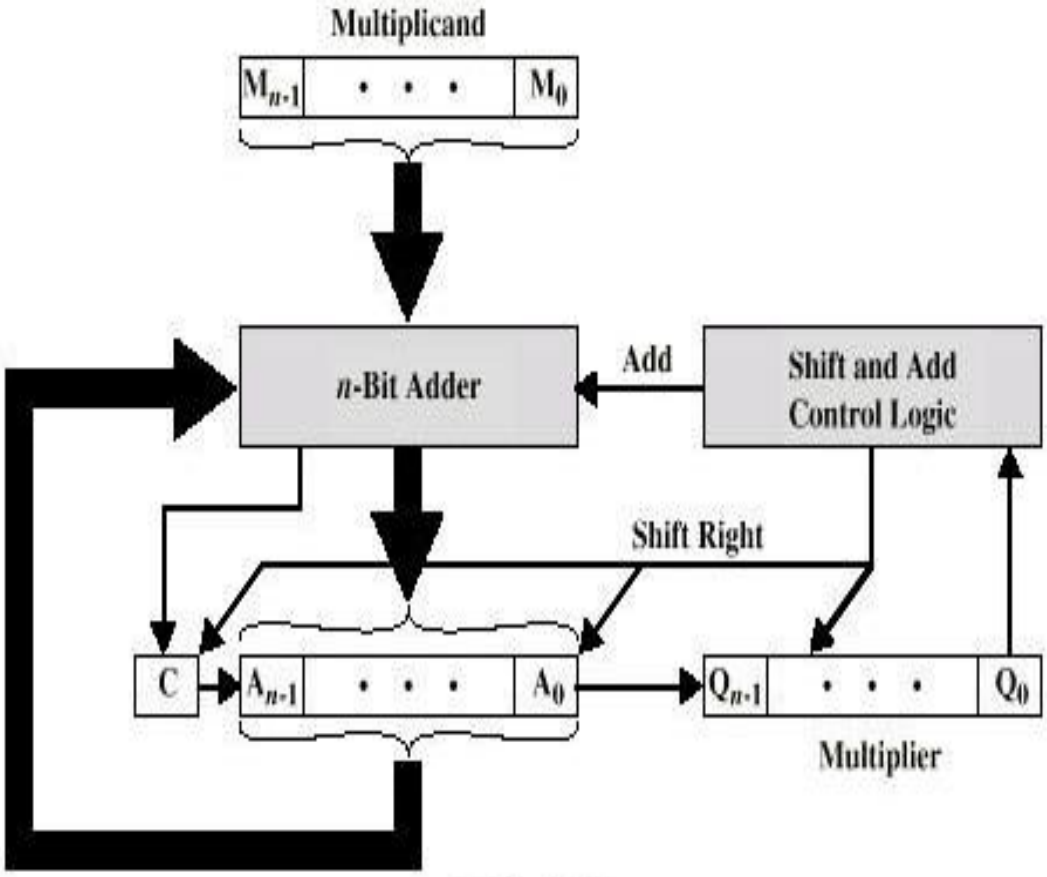
FIG-7b

The following SHIFT & ADD method flow chart depicts the multiplication logic for unsigned numbers.



Despite the use of a combinational network, there is a considerable amount of delay associated with the arrangement shown. Although the preceding combinational multiplier is easy to understand, it uses many gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. The worst case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. The path includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full adder block, the path has a total of 6(*n* - 1) - 1 gate delays, including the initial AND gate delay in all cells, for the *n* x *n* array. In the delay expression, (n-1) because, only the AND gates are actually needed in the first row of the array because the incoming (initial) partial product PPO is zero.

Multiplication can also be performed using a mixture of combinational array techniques (similar to those shown in Figure 7) and sequential techniques requiring less combinational logic. Multiplication is usually provided as an instruction in the machine instruction set of a processor. High-performance processor (DS processors) chips use an appreciable area of the chip to perform arithmetic functions on both integer and floating- point operands. Sacrificing an area on-chip for these arithmetic circuits increases the speed of processing. Generally, processors built for real time applications have an on- chip multiplier.



(a) Block Diagram

Another simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps. The block diagram in Figure *8a* shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using single n-bit adder n times to implement the spatial addition performed by the *n*

rows of ripple-carry adders. Registers A and Q combined to hold PPi while multiplier bit q_i generates the signal Add/No-add. This signal controls the addition of the multiplicand M to PPi to generate PP(i + 1). The product is computed in *n* cycles. The partial product grows in length by one bit per cycle from the initial vector, PPO, of ***n* 0**s in register A. The carry-out from the adder is stored in flip-flop C. To begin with, the multiplier is loaded into register Q, the multiplicand into register M and registers C and A are cleared to 0. At the end of each cycle C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit ***qi,*** appears at the LSB position of Q to generate the Add/No-add signal at the correct time, starting with *qo* during the first cycle, ***q*1** during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP(i + 1), and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After *n* cycles, the high-order half-of- the product is held in register A and the low-order half is in register Q. The multiplication example used above is shown in Figure *8b* as it would be performed by this hardware arrangement.

Using this sequential hardware structure, it is clear that a **multiply** instruction takes much more time to execute than an **Add** instruction. This is because of the sequential circuits associated in a multiplier arrangement. Several techniques have been used to speed up multiplication; bit pair recoding, carry save addition, repeated addition, etc.

2.4 SIGNED-OPERAND MULTIPLICATION:

Multiplication of 2's-complement signed operands, generating a double-length product is still achieved by accumulating partial products by adding versions of the multiplicand as decided by the multiplier bits. First, consider the case of a **positive multiplier and a negative multiplicand**. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. In Figure 9, for example, the 5-bit signed operand, - 13, is the multiplicand, and +11, is the 5 bit multiplier & the expected product -143 is 10-bit wide. The sign extension of the multiplicand is shown in red color. Thus, the hardware discussed earlier can be used for negative multiplicands if it provides for sign extension of the partial products.

For a **negative multiplier**, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. In order to take care of both negative and positive multipliers**, BOOTH algorithm** can be used.

	1 0 0 1 1 (-13) X 0 1 0 1 1 (+11) **1 1 1 1 1 1 0 0 1 1** **1 1 1 1 1 0 0 1 1** **0 0 0 0 0 0 0 0 0** **1 1 1 0 0 1 1** **0 0 0 0 0 0** **1 1 0 1 1 1 0 0 0 1 (-143)**	

Booth Algorithm

The Booth algorithm generates a 2n-bit product and both positive and negative 2's-complement *n-bit* operands are uniformly treated. To understand this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110(+30). To derive the product, as in the normal standard procedure, we could add four appropriately shifted versions of the multiplicand,. However, using the Booth algorithm, we can reduce the number of required operations by
regarding this multiplier as the difference between numbers 32 & 2 as shown below;

0 1 0 0 0 0 0 (32)
0 0 0 0 0 1 0 (-2)
0 0 1 1 1 1 0 (30)

This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0

+1000 - 10. In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

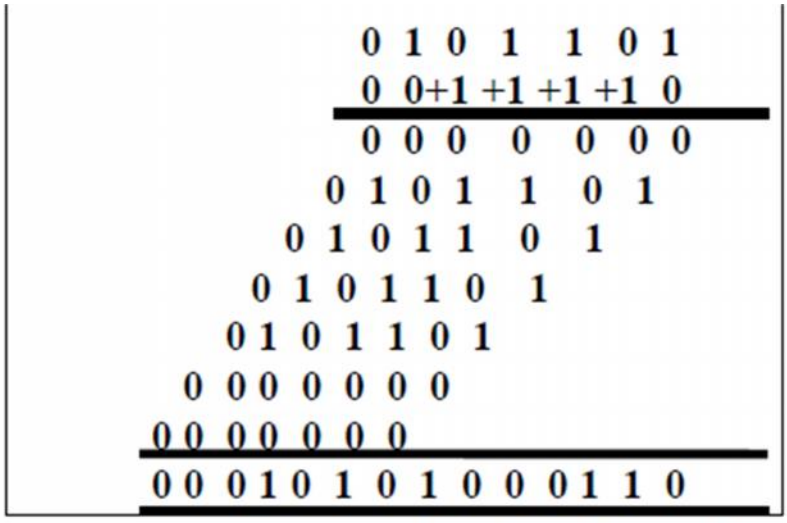


FIG-10a: Normal Multiplication

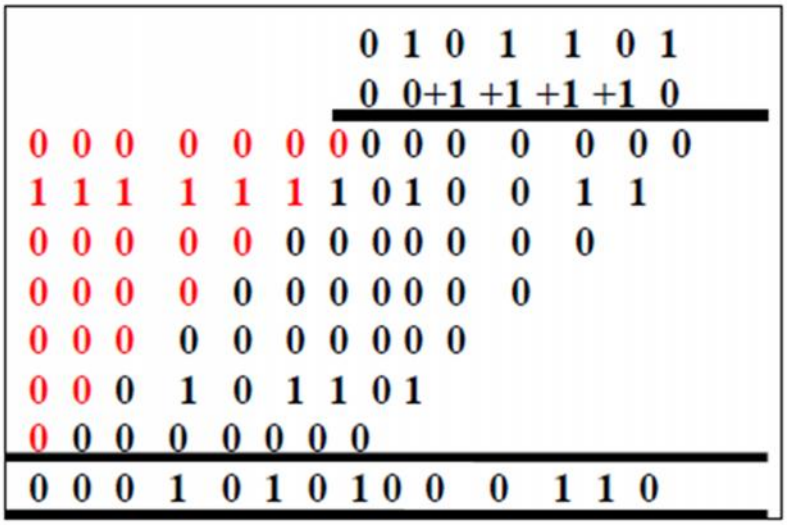
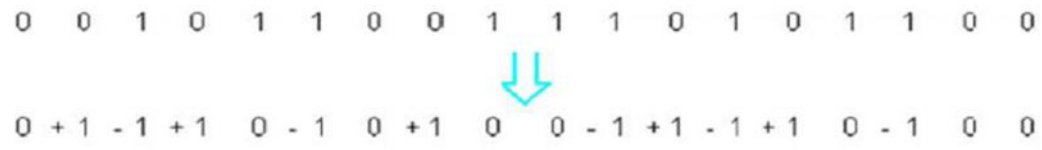


FIG-10b: Booth Multiplication

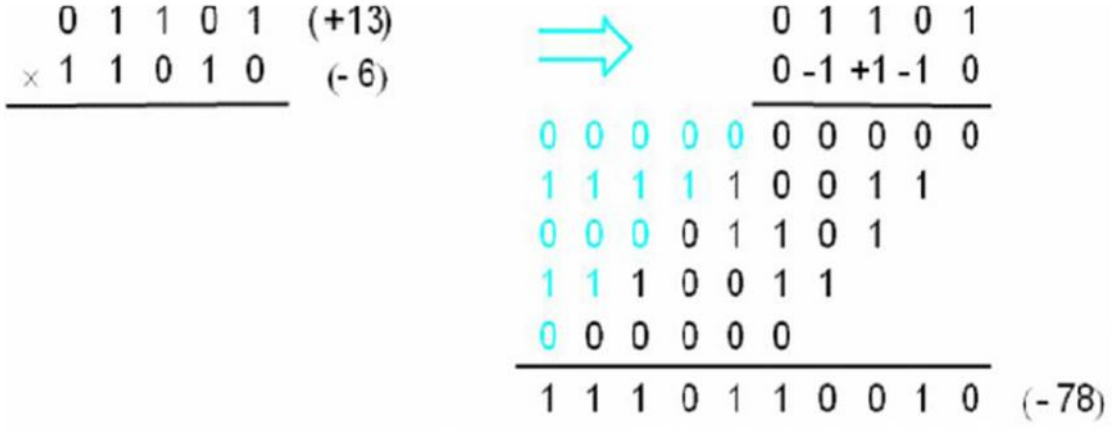
Figure 10 illustrates the normal and the Booth algorithms for the said example. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. See Figure 11a for another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 11a.

To verify the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement



Booth recoding of a multiplier.

FIG-11a



Booth multiplication with a negative multiplier.

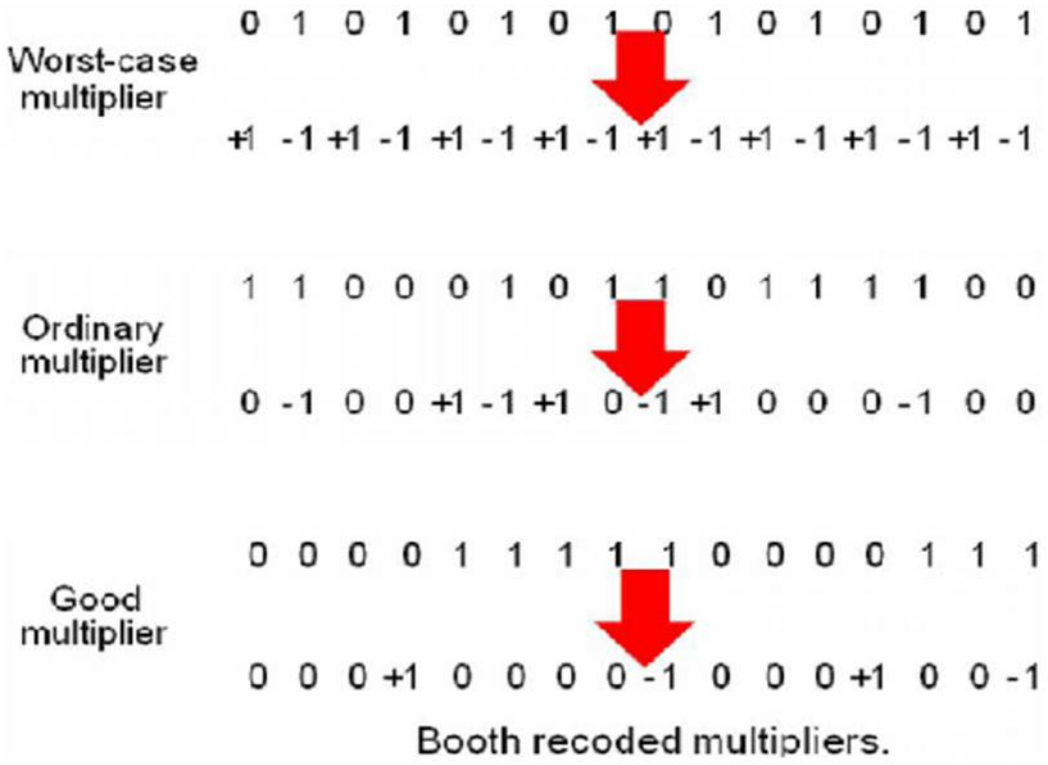
FIG-11b

The Booth technique for recoding multipliers is summarized in Figure 12a.
Thetransformation 011... 110 =>• +100... .0 -10 is called skipping over Is. This term is derivedfrom the case in which the multiplier has its 1 s grouped into a few contiguous blocks. Only a few versions of the shifted multiplicand (the summands) must be added to generate the product, thus speeding up the multiplication operation. However, in the worst case—that of alternating 1 s and 0s in the multiplier — each bit of the multiplier selects a summand.
In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit, worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Fig 12a.

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	0 × M
0	1	+1 × M
1	0	–1 × M
1	1	0 × M

Booth multiplier recoding table.

Fig:12.Booth multiplier recoding table.



Booth recoded multipliers.

Fig :12.aBooth recoded multipliers

T he Booth algorithm has two attractive features. First, it handles both positive andnegative multipliers uniformly. Second, it achieves some efficiency in the number ofadditions required when the multiplier has a few large blocks of 1 s. The speed gained byskipping

over 1s depends on the data. On average, the speed of doing multiplication withthe Booth algorithm is the same as with the normal algorithm.

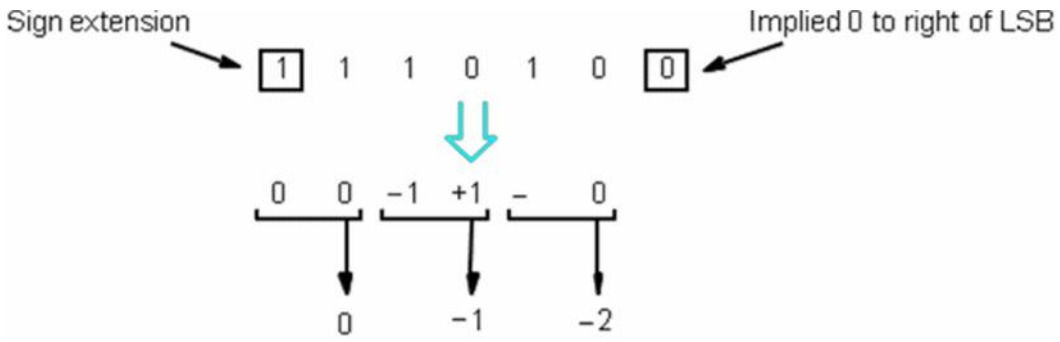
2.5 FAST MULIPLICATION:

There are two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is *n/2* for n-bit operands. The second technique reduces the time needed to add the summands (carry-save addition of summands method).

Bit-Pair Recoding of Multipliers:

This ***bit-pair recoding*** *technique* halves the maximum number of summands. It is derived from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1). That is, instead of adding —1 times the multiplicand M at shift position i to + 1 x M at position *i* + 1, the same result is obtained by adding +1 x M at position I Other examples are: (+1 0) is equivalent to (0 +2),(-l +1) is equivalent to (0 —1). and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits.

selection decisions for all possibilities. The multiplication operation in figure 11a is shown in Figure 15. It is clear from the example that the bit pair recoding method requires only n/2 summands as against n summands in Booth's algorithm



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 × M
0	0	1	+ 1 × M
0	1		+ 1 × M
0	1	1	+ 2 × M
1	0	0	− 2 × M
1	0	1	− 1 × M
1	1	0	− 1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

FIG – 15

```
          0  1  1  0  1   (+13)
      ×   1  1  0  1  0   (-6)
```

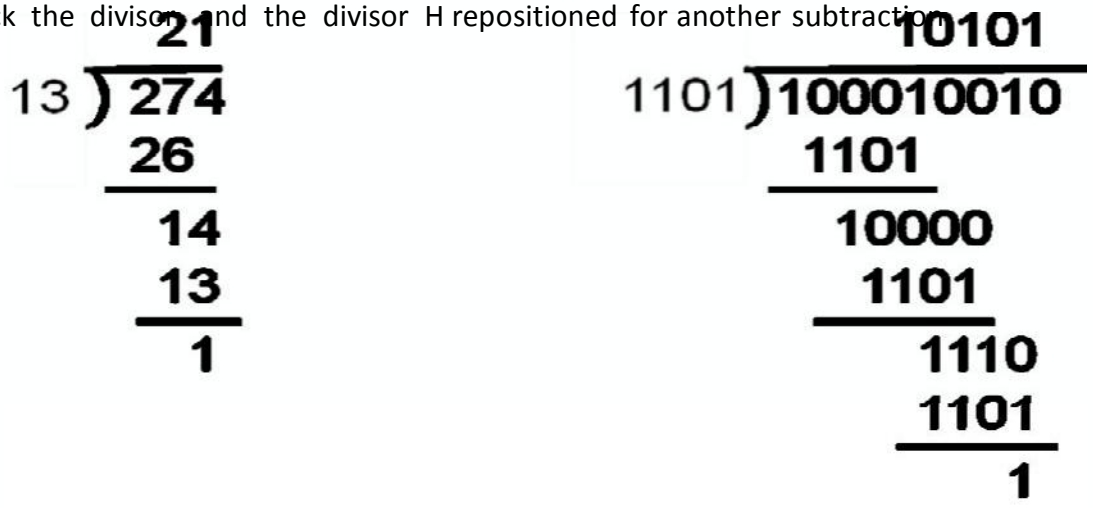
```
                   0  1  1  0  1
                   0 -1 +1 -1  0
                ------------------
   0  0  0  0  0   0  0  0  0  0
   1  1  1  1  1   0  0  1  1
   0  0  0  0  1   1  0  1
   1  1  1  0  0   1  1
   0  0  0  0  0   0
-------------------------------------
   1  1  1  0  1   1  0  0  1  0   (-78)
```

```
                   0  1  1  0  1
                   0    -1    -2
                ------------------
   1  1  1  1  1   0  0  1  1  0
   1  1  1  1  0   0  1  1
   0  0  0  0  0   0
-------------------------------------
   1  1  1  0  1   1  0  0  1  0
```

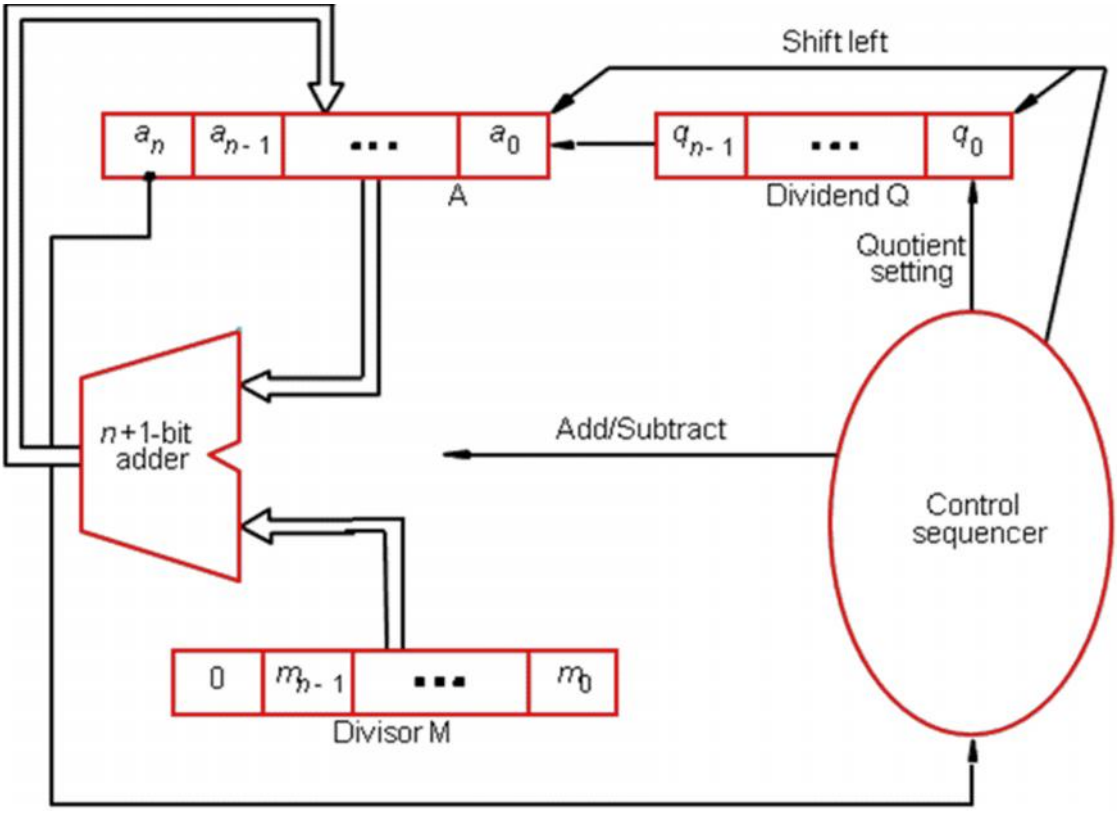
2.6 INTEGER DIVISION:

Positive-number multiplication operation is done manually in the way it is done in a logic circuit. A similar kind of approach can be used here in discussing integer division. First, consider positive-number division. Figure 16 shows examples of decimal division and its binary form of division. First, let us try to divide 2 by13, and it does not work. Next, let us try to divide 27 by 13. Going through the trials, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once and the remainder is 1. Binary division is similar to this, with the quotient bits only 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtrac- tion. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and sub- traction is performed. On the other hand, if the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor H repositioned for another subtraction.



Longhand division examples.



Circuit arrangement for binary division.

FIG – 17: Binary Division

Restoring Division:

Figure 17 shows a logic circuit arrangement that implements restoring division. Note its similarity to the structure for multiplication that was shown in Figure 8. An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following *n*times:

1. Shift A and Q left one binary position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A);otherwise, set $q_0$to 1.

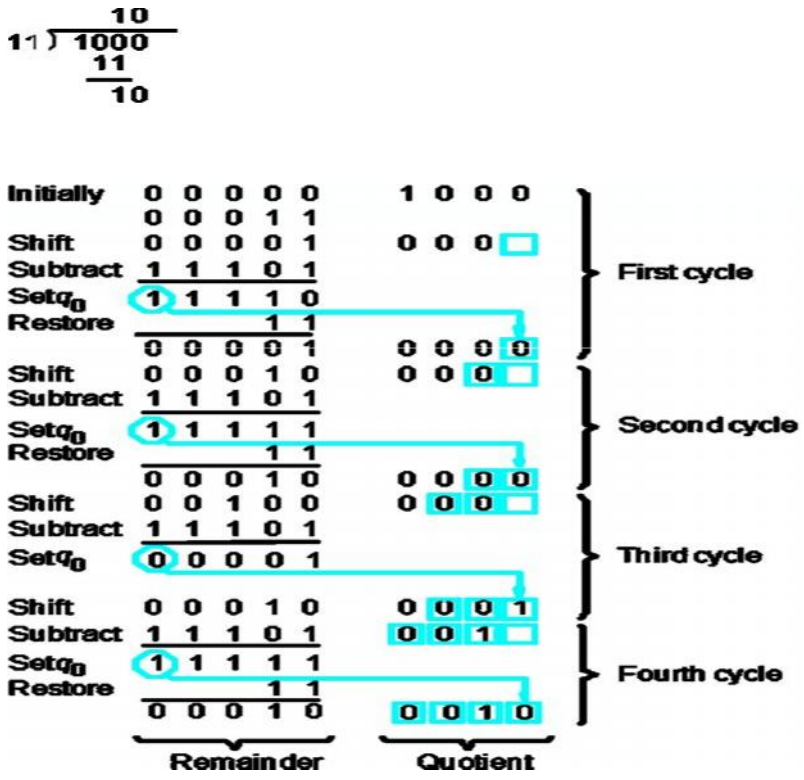


FIG – 18: Restoring Division

No restoring Division:

The restoring-division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A - M. If A is negative, we restore it by performing A + M, and then we shift it left and subtract M. This is equivalent to performing 2A + M. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed. We can summarize this in the following algorithm for no restoring division.

Step 1: Do the following times:

1.If the sign of A is 0, shift A and Q left one bit position and subtract M fromA; otherwise, shift A and Q left and add M to A.

2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Step 2: If the sign of A is 1, add M to A.

Step 2 is needed to leave the proper positive remainder in A at the end of the n cycles of Step 1. The logic circuitry in Figure 17 can also be used to perform this algorithm. Note that the Restore operations are no longer needed, and that exactly one Add or Subtract operation is performed per cycle. Figure 19 shows how the division example in Figure 18 is executed by the no restoring-division algorithm.

There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to transform them into positive values. After using one of the algorithms

just discussed, the results are transformed to the correct signed values, as necessary

.

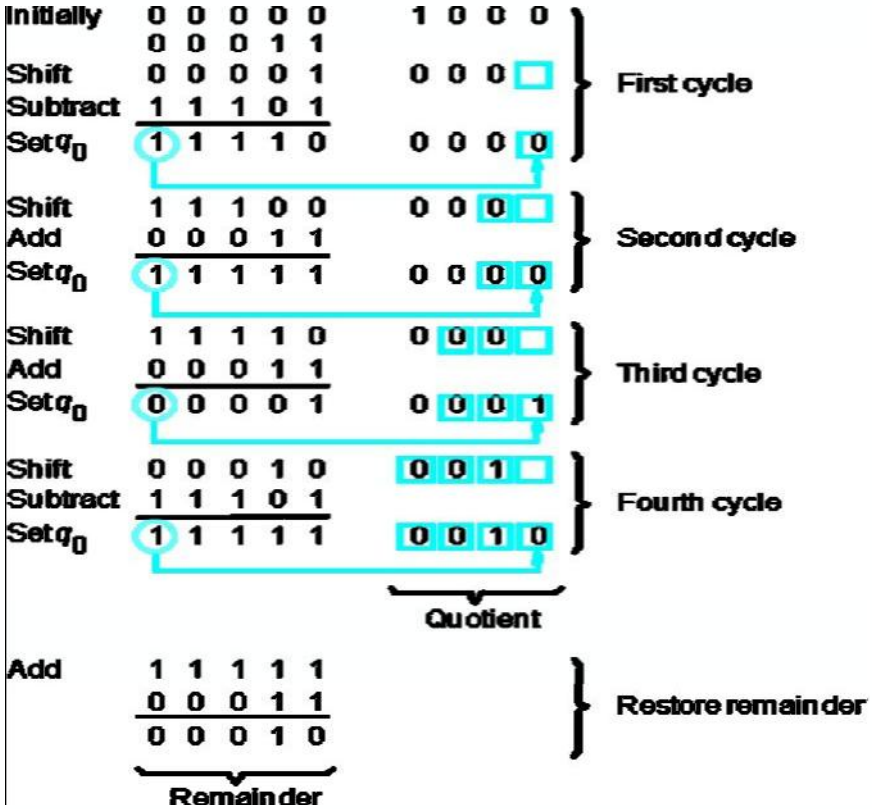


FIG – 19: Non-restoring Division

2.7 FLOATING-POINT NUMBERS AND OPERATIONS:

Floating – point arithmetic is an automatic way to keep track of the radix point. The discussion so far was exclusively with fixed-point numbers which are considered as integers, that is, as having an implied binary point at the right end of the number. It is also possible to assume that the binary point is just to the right of the sign bit, thus representing a fraction or any where else resulting in real numbers. In the 2's-complement system, the signed value F, represented by the n-bit binary fraction

$B = b_0.b_{-1}b_{-2} \ldots b_{-(n-1)}$ is given by

$F(B) = -b_o \times 2^{\circ} + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-(n-X)} \times 2^{-\{n\sim l\}}$ where the range of F is

$-1 \leq F \leq 1 - 2^{-(n-1)}$. Consider the range of values representable in a 32-bit, signed, fixed- point format. Interpreted as integers, the value range is approximately 0 to $\pm 2.15 \times 10^{9}$. If

we consider them to be fractions, the range is approximately $\pm 4.55 \times 10^{-10}$ to ±1. Neither of these ranges is sufficient for scientific calculations, which might involve parameters like Avogadro's number (6.0247×10^{23} $mole^{-1}$) or Planck's constant (6.6254×10^{-27} erg s). Hence, we need to easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In such a case, the binary point is said to float, and the numbers are called floating-point numbers. This distinguishes them from fixed-point numbers, whose binary point is always in the same position.

Because the position of the binary point in a floating-point number is variable, it must be given explicitly in the floating-point representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, $6.6254 \text{ -}10^{-27,}$ -1.0341×10^{2}, -7.3000×10^{-14}, and so on. These numbers are said to be given to fivesignificant digits. The scale factors (10^{23}, 10^{-27}, and so on) indicate the position of the decimal point with respect to the significant digits. By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be normalized. Note that the base, 10, in the scale factor is fixed and does not need to appear explicitly in the machine representation of a floating-point number. The sign, the significant digits, and the exponent in the scale factor constitute the representation. We are thus motivated

to define a floating-point number representation as one in which a number is represented by its sign, a string of significant digits, commonly called the mantissa, and an exponent to an implied base for the scale factor.

2.8 BASIC PROCESSING UNIT

The heart of any computer is the central processing unit (CPU). The CPU executes all the machine instructions and coordinates the activities of all other units during the execution of an instruction. This unit is also called as the Instruction Set Processor (ISP). By looking at its internal structure, we can understand how it performs the tasks of fetching, decoding, and executing instructions of a program. The processor is generally called as the central processing unit (CPU) or micro processing unit (MPU).An high-performance processor can be built by making various functional units operate in parallel. High-performance processors have a pipelined organization where the execution of one instruction is started before the execution of the preceding instruction is completed. In another approach, known as superscalar operation, several instructions are fetched and executed at the same time. Pipelining and superscalar architectures provide a very high performance for any processor.

A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program. A program is a set of instructions performing a meaningful task. An instruction is command to the processor & is executed by carrying out a sequence of sub-operations called as micro-operations. Figure 1 indicates various blocks of a typical processing unit. It consists of PC, IR, ID, MAR, MDR, a set of register arrays for temporary storage, Timing and Control unit as main units.

2.8.1 FUNDAMENTAL CONCEPTS:

Execution of a program by the processor starts with the fetching of instructions one at a time, decoding the instruction and performing the operations specified. From memory, instructions are fetched from successive locations until a branch or a jump instruction is encountered. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter (PC) or Instruction Pointer (IP). After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. But, when a branch instruction is to be executed, the PC will be loaded with a different (jump/branch address).

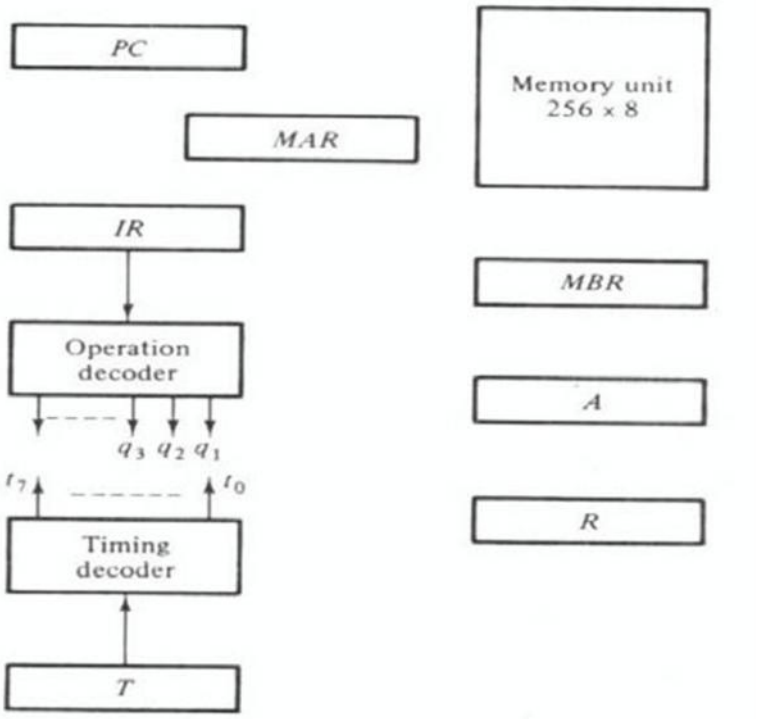


Fig-1

Instruction register, IR is another key register in the processor, which is used to hold the op-codes before decoding. IR contents are then transferred to an instruction decoder (ID) for decoding. The decoder then informs the control unit about the task to be executed. The control unit along with the timing unit generates all necessary control signals needed for the instruction execution. Suppose that each instruction comprises 2 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as an instruction code to be executed. Hence, they are loaded into the IR/ID. Symbolically, this operation can be written as

IR ← [(PC)]

2. Assuming that the memory is byte addressable, increment the contents of the PC by 2, that is,

PC ← [PC] + 2

3. Decode the instruction to understand the operation & generate the control signals necessary to carry out the operation.

4. Carry out the actions specified by the instruction in the IR.

In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction. These two steps together are usually referred to as the fetch phase; step 3 constitutes the decoding phase; and step 4 constitutes the execution phase.

To study these operations in detail, let us examine the internal organization of the processor. The main building blocks of a processor are interconnected in a variety of ways. A very simple organization is shown in **Figure 2**. A more complex structure that provides high performance will be presented at the end.

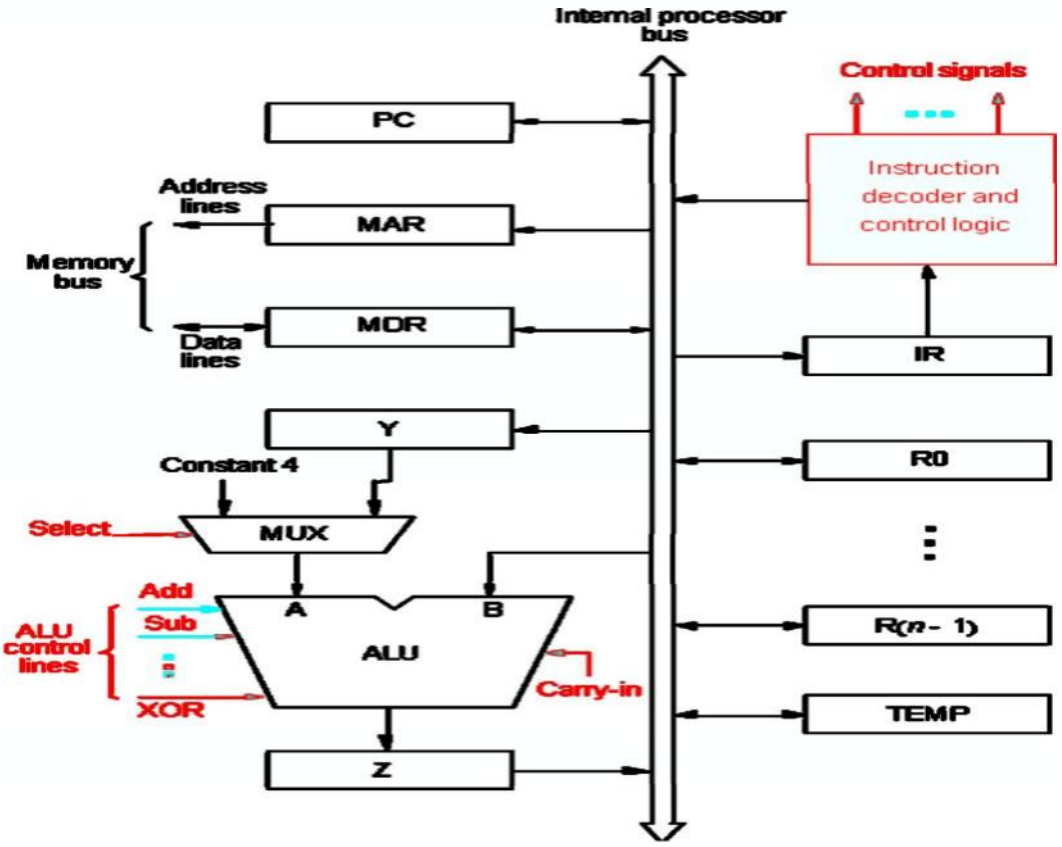


Figure shows an organization in which the arithmetic and logic unit (ALU) and all the registers are interconnected through a single common bus, which is internal to the processor. The data and address lines of the external memory bus are shown in Figure 7.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are

connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The number and use of the processor registers RO through R(n - 1) vary considerably from one processor to another. Registers may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, Z, and TEMP in Figure 2, have not been mentioned before. These registers are transparent to the programmer, that is, the programmer need not be concerned with them because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and Select Y for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The registers, the ALU, and the interconnecting bus are collectively referred to as the *data path.*

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

1. Transfer a word of data from one processor register to another or to the ALU
2. Perform an arithmetic or a logic operation and store the result in a processor register
3. Fetch the contents of a given memory location and load them into a processor register
4. Store a word of data from a processor register into a given memory location

We now consider in detail how each of these operations is implemented, using the simple processor model in Figure.

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register.

This is represented symbolically in Figure 3. The input and output of register Ri are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out} respectively. When Ri_{in} is set to 1, the data on the bus are loaded into Ri. Similarly, when Ri_{out}, is set to 1, the contents of register Ri_{out} are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register RI to register R4. This can be accomplished as follows:

1. Enable the output of register $R1_{out}$ by setting Rl_{out}, tc 1. This places the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time periods defined by the processor clock. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, Rl_{out} and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals Rl_{out} and $R4_{in}$ will return to 0. We will use this simple model of the timing of data transfers for the rest of this chapter. However, we should point out that other schemes are possible. For example, data transfers may use both the rising and falling edges of the clock. Also, when edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as multiphase clocking.

An implementation for one bit of register Ri is shown in Figure 7.3 as an example. A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When the control input Ri_{in} is equal to 1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock. When Ri_{in} is equal to 0, the multiplexer feeds back the value currently stored in the flip-flop.

The Q output of the flip-flop is connected to the bus via a tri-state gate. When Ri_{out}, isequal to 0, the gate's output is in the high-impedance (electrically disconnected) state. This corresponds to the open-circuit state of a switch. When Ri_{out}, = 1, the gate drives the bus to 0 or 1, depending on the value of Q.

2.9 EXECUTION OF A COMPLETE INSTRUCTION:

Let us now put together the sequence of elementary operations required to execute one instruction. Consider the instruction

Add (R3), R1

which adds the contents of a memory location pointed to by R3 to register R1.
Executing this instruction requires the following actions:

1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition.
4 .Load the result into Rl.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

The listing shown in figure above indicates the sequence of control steps required to perform these operations for the single-bus architecture of Figure 2. Instruction execution proceeds as follows. In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z. The updated value is moved from register Z back into the PC during step 2, while waiting for the memory to respond. In step 3, the word fetched from the memory is loaded into the IR. Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions. The instruction decoding circuit interprets the contents of the IR at the beginning of step 4. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated.

Then the contents of Rl are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed in step 6. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing Select Y. The sum is stored in register Z, then transferred to Rl in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 7.6 except Y in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction. But, in Branch instructions the updated value of the PC is needed to compute the Branch target address. To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instructions:

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Listing in figure 8 below gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Fig 8

For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The reason for this can be readily appreciated from the control sequence in Figure 7. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on- negative (Branch<0) instruction, step 4 is replaced with

Offset-field-of-IR_{out} Add, Z_{in}, If N = 0 then End

Thus, if N = 0 the processor returns to step 1 immediately after step 4. If N = 1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

2.10 MULTIPLE-BUS ORGANIZATION:

The resulting control sequences shown are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Figure 7 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the register file. In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the imple- mentation of random-access memories (RAMs) described in Chapter 5. The register file in Figure 9 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z in Figure 2.

A second feature in Figure 9 is the introduction of the Incremental unit, which is used to increment the PC by 4.. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the
memory addresses in Load Multiple and Store Multiple instructions.

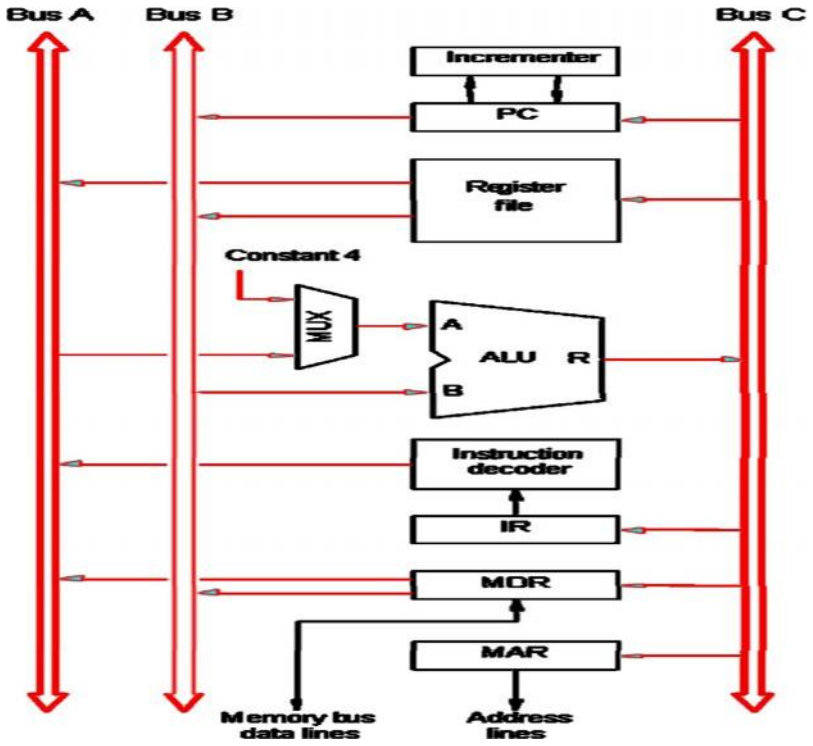


Fig: multibus organization

Consider Add R4,R5,R6

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

The control sequence for executing this instruction is given in Figure. In step 1, the contents of the PC are passed through the ALU, using the R=B control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock

cycle and will not affect the contents of MAR. In step 2, the processor waits for MFC and loads the data received into MDR, then transfers them to IR in step 3. Finally, the execution phase of the instruction requires only one control step to complete, step 4.

By providing more paths for data transfer a significant reduction in the number of clock cycles needed to execute an instruction is achieved.

2.11 HARDWIRED CONTROL:

To execute instructions, the processor must have some means of generating the con- trol signals needed in the proper sequence. Computer designers use a wide variety of techniques to solve this problem. The approaches used fall into one of two categories: hardwired control and micro programmed control. We discuss each of these techniques in detail, starting with hardwired control in this section.

Consider the sequence of control signals given in Figure 7. Each step in this sequence is completed in one clock period. A counter may be used to keep track of the control steps, as shown in Figure 11. Each state, or count, of this counter corresponds to one control step. The required control signals are determined by the following information:

1. Contents of the control step counter
2. Contents of the instruction register
3. Contents of the condition code flags
4. External input signals, such as MFC and interrupt requests

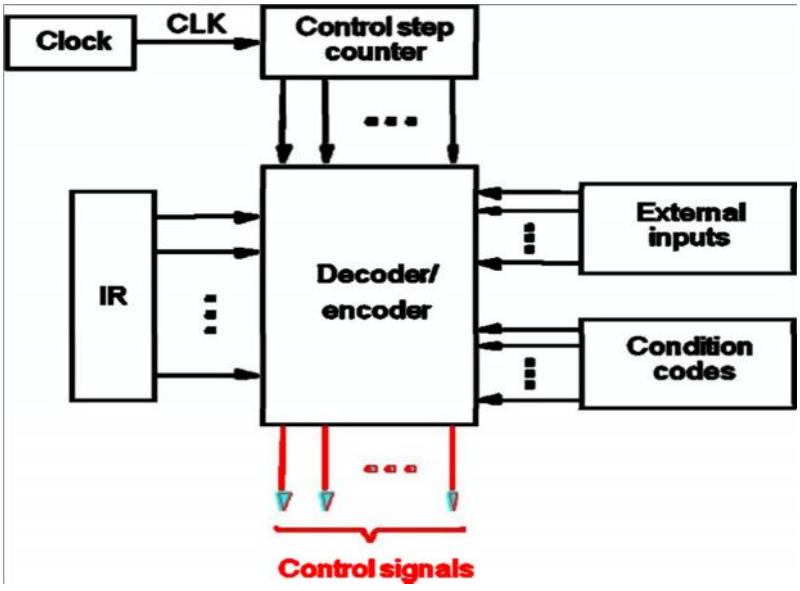


Fig :Hardwired control

To gain insight into the structure of the control unit, we start with a simplified view of the hardware involved. The decoder/encoder block in Figure is a combinational circuit that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 12. The step decoder provides a separate signal line for each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines are set to 0. (For design details of decoders, refer to Appendix A.) The input signals to the encoder block in Figure 12 are combined to generate the individual control signals Y_{in}, PC_{out}, Add, End, and so on. An example of how the encoder generates the Z_{in} control signal for the processor organization in Figure 2 is given in Figure 13. This circuit implements the logic function

$$Z_{in}=T_1+T_6 - ADD + T_4-BR+---$$

This signal is asserted during time slot Ti for all instructions, during T_6 for an Add instruction, during T4 for an unconditional branch instruction, and so on. The logic function for Z_{in} is derived from the control sequences in Figures 7 and 8. As another example, Figure 14 gives a circuit that generates the End control signal from the logic function

$$End = T7 \bullet ADD + T5 \bullet BR + (T5 \bullet N + T4 \bullet N) \bullet BRN + \bullet \bullet \bullet$$

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 12 contains another control signal called RUN. When

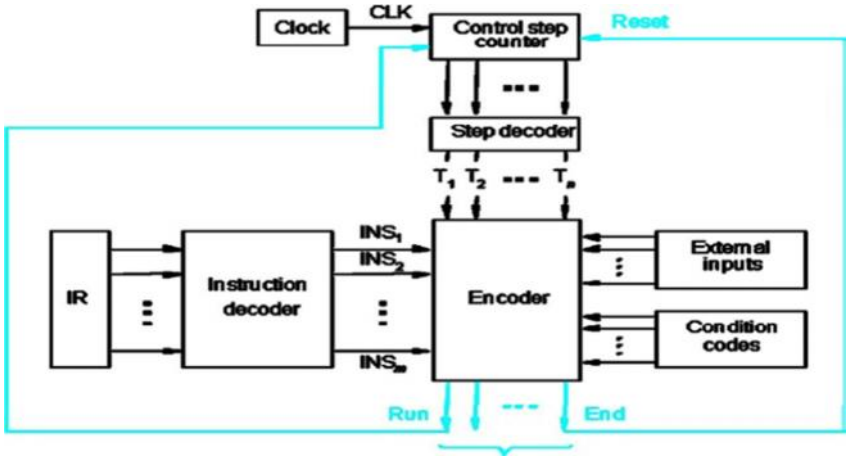


Fig 12

set to 1, RUN causes the counter to be incremented by one at the end of every clock cycle. When RUN is equal to 0, the counter stops counting. This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

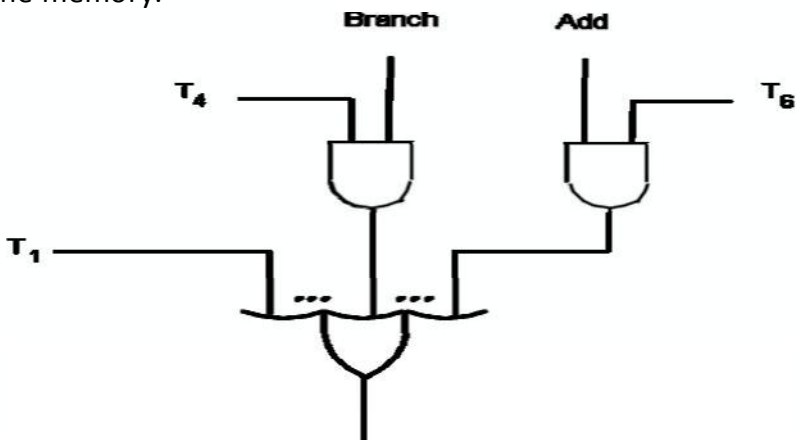


Fig 13a

The control hardware shown can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of operations carried out by this machine is determined by the wiring of the logic elements, hence the name "hardwired." A controller that uses this approach can operate at high speed. However, it has little flexibility, and the complexity of the instruction set it can implement is limited.

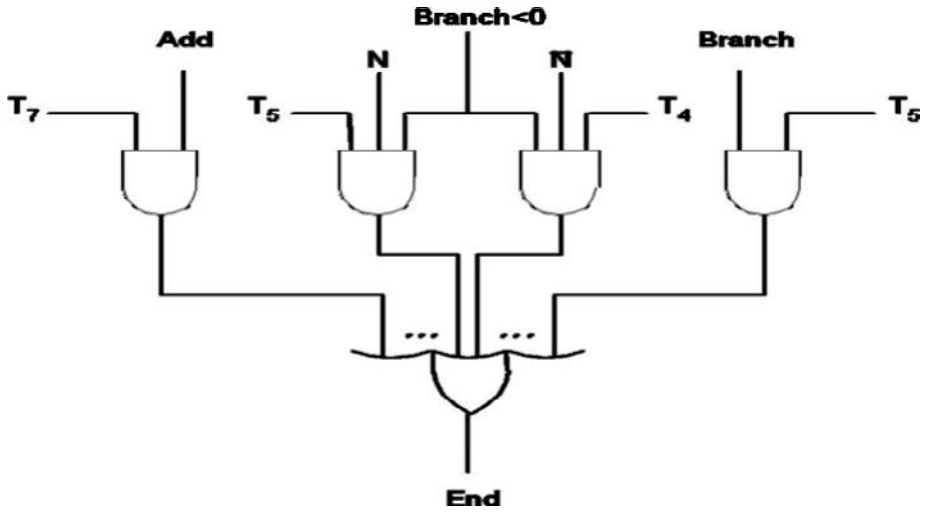
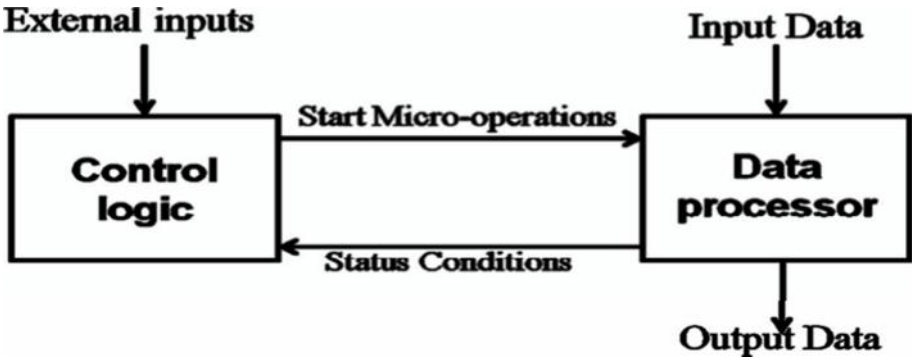


Fig 13b

2.12 MICROPROGRAMMED CONTROL

ALU is the heart of any computing system, while Control unit is its brain. The design of a control unit is not unique; it varies from designer to designer. Some of the commonly used control logic design methods are;

- Sequence Reg & Decoder method
- Hard-wired control method
- PLA control method
- Micro-program control method



Micro-instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Fig 15

The control signals required inside the processor can be generated using a control step counter and a decoder/ encoder circuit. Now we discuss an alternative scheme, called micro programmed control, in which control signals are generated by a program similar to machine language programs.

First, we introduce some common terms. A control word (CW) is a word whose individual bits represent the various control signals in Figure 12. Each of the control steps in the control sequence of an instruction defines a unique combination of Is and Os in the CW. The CWs corresponding to the 7 steps of Figure 6 are shown in Figure 15. We have assumed that Select Y is represented by Select = 0 and Select4 by Select = 1. A sequence of CWs corresponding to the control sequence of a machine instruction constitutes the micro routine for that instruction, and the individual control words in this micro routine are referred to as microinstructions.

The micro routines for all instructions in the instruction set of a computer are stored in a special memory called the control store. The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding micro routine from the control store. This suggests organizing the control unit as shown in Figure 16. To read the control words sequentially from the control store, a micro program counter (µPC) is used. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the µPC. The µPC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence, the control signals are delivered to various parts of the processor in the correct sequence.

One important function of the control unit cannot be implemented by the simple organization in Figure 16. This is the situation that arises when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In the case of hardwired control, this situation is handled by including an appropriate logic function, in the encoder circuitry. In micro programmed control, an alternative approach is to use conditional branch microinstructions. In addition to the branch address, these microinstructions specify which of the external inputs, condition codes, or, possibly, bits of the instruction register should be checked as a condition for branching to take place.

The instruction Branch <0 may now be implemented by a micro routine such as that shown in Figure 17. After loading this instruction into IR, a branch

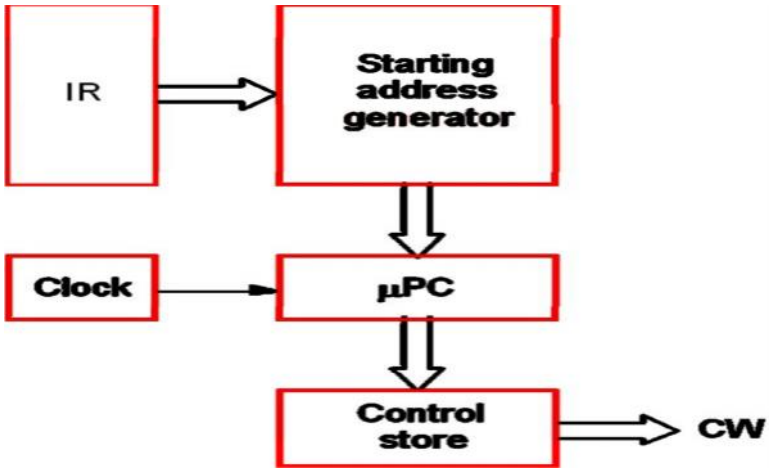


Fig 16

Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
1	Z_{out} , PC_{in} , Y_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
...	...
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out} , Select Y, Add, Z_{in}
27	Z_{out} , PC_{in} , End

Fig 17

microinstruction transfers control to the corresponding micro routine, which is assumed to start at location 25 in the control store. This address is the output of staring address generator block codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 0 to fetch a new machine instruction. Otherwise the microinstruction at location 27 loads this address into the PC

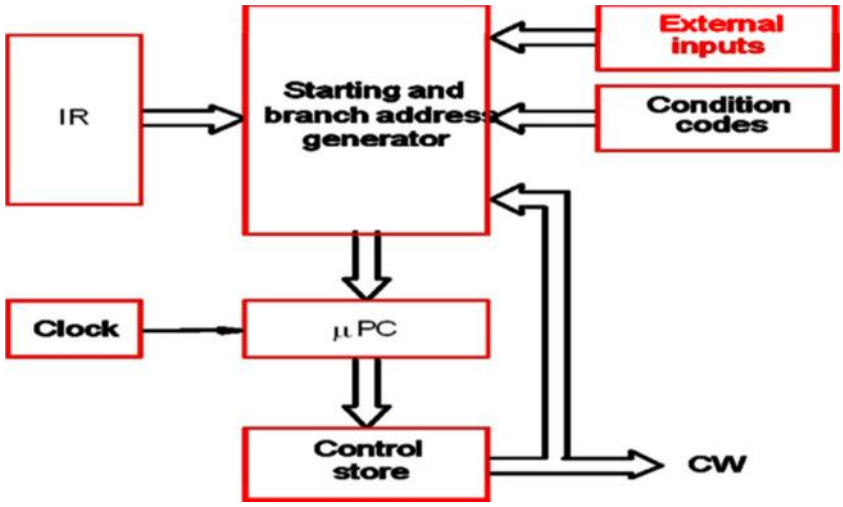


Fig 18

To support micro program branching, the organization of the control unit should be modified as shown in Figure 18. The starting address generator block of Figure 16 becomes the starting and branch address generator. This block loads a new address into the µPC when a microinstruction instructs it to do so. To allow implementation of a conditional branch, inputs to this block consist of the external inputs and condition codes as well as the contents of the instruction register. In this control unit, the µPC is incremented every time a new microinstruction is fetched from the micro program memory, except in the following situations:

1. When a new instruction is loaded into the IR, the µPC is loaded with the starting address of the micro routine for that instruction.
2. When a Branch microinstruction is encountered and the branch condition is satis- fied, the µPC is loaded with the branch address.
3. When an End microinstruction is encountered, the µPC is loaded with the address of the first CW in the micro routine for the instruction fetch cycle

Microinstructions

Having described a scheme for sequencing microinstructions, we now take a closer look at the format of individual microinstructions. A straightforward way to structure Microinstruction is to assign one bit position to each control signal, as in Figure 15.

However, this scheme has one serious drawback — assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to 1 (to be used for active gating) in any given microinstruction, which means the available bit space is poorly used. Consider again the simple processor of Figure 2, and assume that it contains only four general- purpose registers, R0, Rl, R2, and R3. Some of the connections

in this processor are permanently enabled, such as the output of the IR to the decoding circuits and both inputs to the ALU. The remaining connections to various registers require a total of 20 gating signals. Additional control signals not shown in the figure are also needed, including the Read, Write, Select, WMFC, and End signals. Finally, we must specify the function to be performed by the ALU. Let us assume that 16 functions are provided, including Add, Subtract, AND, and XOR. These functions depend on the particular ALU used and do not necessarily have a one-to-one correspondence with the machine instruction OP codes. In total, 42 control signals are needed.

If we use the simple encoding scheme described earlier, 42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive. For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously. This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one *micro operation* per group is specified in any microinstruction. Then it is possible to use a binary coding scheme to represent the signals within a group. For example, four bits suffice to represent the 16 available functions in the ALU. Register output control signals can be placed in a group consisting of PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$ Rl_{out}, $R2_{out}$, $R3_{out}$, and $TEMP_{out}$. Any one of these can be selected by a unique 4-bit code.

Further natural groupings can be made for the remaining signals. Figure 19 shows an example of a partial format for the microinstructions, in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for the case in which no action is required. For example, the all-zero pattern in Fl indicates that none of the registers that may be specified in this field should have its contents placed on the bus. An inactive code is not needed in all fields.

For example, F4 contains 4 bits that specify one of the 16 operations performed in the ALU. Since no spare code is included, the ALU is active during the execution of every microinstruction. However, its activity is monitored by the rest of the machine through register Z, which is loaded only when the Z_{in} signal is activated.

Grouping control signals into fields requires a little more hardware because decoding circuits must be used to decode the bit patterns of each field into individual control signals. The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller