**COMPUTER ORGANIZATION (CS 46)**

**Chapter 5:**

**ARITHMETIC:**

The basic operation needed for all arithmetic operations in a digital computer is the addition of two numbers. Subtraction can be achieved through addition & complement operations, multiplication through repeated addition & division can be achieved through repeated subtraction. All arithmetic operations & logical operations are implemented in the arithmetic and logic unit (ALU) of a processor. It can be shown that basic building block of all arithmetic & logic operation (ALU) is a parallel adder. That means, all arithmetic operations and basic logic functions such as AND, OR, NOT, and EXCLUSIVE-OR (XOR) are implemented using a parallel adder and additional combinational circuits. The time needed to perform an addition operation affects the processor's performance, similarly multiply and divide operations, which require more complex circuitry, also affect processor performance. It is therefore necessary to design some of the advanced techniques to perform arithmetic and logical operations at a very high speed.

Compared with arithmetic operations, logic operations are simple to implement using combinational circuitry. They require only independent Boolean operations on individual bit positions of the operands, whereas carry/borrow lateral signals are required in arithmetic operations.

It is already observed that 2's-complement form of representing a signed binary number is the best representation from the point of performing addition and subtraction operations. The examples already used show that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bits. In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement. If overflow does not occur, the sum is correct, and any output carry can be ignored. If overflow occurs, then the sum is to be corrected by taking 2's complement of output.

**ADDITION 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 the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use \(c_i\) 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 [1st 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.
FIG-1: Addition of binary vectors.

FIG-2: 4 - Bit parallel Adder.

Addition/Subtraction Logic Diagram:
The 4-bit adder shown in Figure 2 can be used to add 2's-complement numbers X and Y, where the $x_{n-1}$ and $y_{n-1}$ bits are the sign bits. In this case, the carry-out bit, $c_n$ is not part of the answer. In an addition, overflow can only occur when the signs of the two operands are the same. In this case, overflow obviously occurs if the sign of the result is different. Therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

$$\text{Overflow} = x_{n-1} y_{n-1} s_{n-1} + \overline{x_{n-1}} \overline{y_{n-1}} s_{n-1}$$

Also, an overflow can be detected by using the carry bits $c_n$ and $c_{n-1}$. An overflow occurs if $c_n$ and $c_{n-1}$ are different. Therefore, a much simpler alternative circuit for detecting an overflow can be obtained by implementing the expression $c_n \oplus c_{n-1}$ with an XOR gate.
Subtraction operation on two numbers X & Y can be performed using 2's-complement method. In order to perform X - Y, we form the 2's-complement of Y and add it to X. The logic circuit shown in Figure-3 can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying the Y vector unchanged to one of the adder inputs along with a carry-in signal, $c_n$. The control input along with the associated ex-or gates either inverts or applies Y vector as it is for subtraction or addition respectively, & hence it is called as control inverter circuit.

![FIG-3: Adder/Subtractor network.](image)

The design of an adder/subtractor circuit can be illustrated as follows; consider the parallel adder shown in figure 4a. Here, the B inputs are complemented and added to A inputs along with a carry for subtraction. For addition A & B inputs are added without changing & with no carry. Figure 4b indicates the necessary arrangement for an adder/subtractor. It is required to design the combinational circuit which receives A, B & S inputs and produces inputs for the full adder. Using K-map simplification for the table, the equations for $X_i$, $Y_i$ & $C_i$ are obtained as indicated. Implementing these equations results in figure3.

![FIG-4a](image)

$$S = A + B' + 1$$
FIG-4: Design of Adder/Subtractor.

Truth Table:

<table>
<thead>
<tr>
<th>S</th>
<th>Ai</th>
<th>Bi</th>
<th>Xi</th>
<th>Yi</th>
<th>Cin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ai</td>
<td>Bi</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Ai</td>
<td>Bi’</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c|c}
  S & Ai & Bi & Xi & Yi \\
  \hline
  0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 1 \\
  0 & 1 & 0 & 1 & 0 \\
  0 & 1 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 & 1 \\
  1 & 0 & 1 & 0 & 0 \\
  1 & 1 & 0 & 1 & 1 \\
  1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{align*}
  Xi &= Ai \\
  Yi &= Bi \oplus S \\
  Cin &= S
\end{align*}
\]

For addition Cin = 0 & for subtraction Cin = 1 along with complement of B.
Hence, add/sub control line is connected to Cin. When the Add/Sub control line is set to 1, the Y vector is 1's-complemented (that is, bit complemented) by the XOR gates and c_o is set to 1 to complete the 2's-complementation of Y. Remember that 2's-complementing a negative number is done in exactly the same manner as for a positive number. An XOR gate can be added to Figure 3 to detect the overflow condition C_n \oplus C_{n-1}. As listed in the truth table Yi is equal to Bi when s = 0 & it is equal to complement of Bi when the control S = 1. Using K-maps the expressions for Xi & Yi can be obtained & implemented as in fig-3.

**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 \oplus c_{n-1} for overflow, this indicator is available after 2n+2 gate delays. In summary, in a parallel adder an nth 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 ith are
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 \( G_i \) & \( P_i \) functions we can decide carry for \( i \)th 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 \( X_i \) and \( Y_i \) 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 derived by considering the propagate function as \( P_i = x_i \oplus y_i \), 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, \( C_i = (G_i-1 + P_i-1 C_{i-1}) \).

Further, \( C_{i-1} = (G_{i-2} + P_{i-2} C_{i-2}) \) and so on. Expanding in this fashion, the final carry expression can be written as below:

\[
C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_1 P_0 G_0
\]

Thus, all carries can be obtained in three gate delays after the input signals \( X_i \), \( Y_i \) and \( C_{in} \) 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 \( c_{i+1} \). 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.
FIG-5: 4 bit carry look ahead adder.

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

\[ c_i = G_i + P_i C_{i-1} \]

where \( i = 0, 1, 2, 3 \)

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 \( S_3(2n-1) \) and 8 gate delays(2n) for \( C_4 \).

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 \( C_{i+1} \) & 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-6: 16 bit carry-look ahead adder.

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 $2n$ 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 \( m_j \), is added to the incoming partial-product bit, based on the value of the multiplier bit, \( q_i \). 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.
The following SHIFT & ADD method flow chart depicts the multiplication logic for unsigned numbers.

P7, P6, P5,…,P0 – product.

Bit of incoming partial product (Ppi) $m_j$

qi

carry-out

FA

Carry-in

Bit of outgoing partial product [Ppi]

Typical cell

Array implementation

FIG-7b
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 $n \times n$ 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 $PP_0$ 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.

![Block Diagram](FIG-8a)
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 qi generates the signal Add/No-add. This signal controls the addition of the multiplicand M to P Pi to generate P Pi(1 + 1). The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, P Po, of n 0s 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, q1 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 P Pi + 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.

**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.

**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

![FIG-9](image-url)
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:

\[
\begin{array}{c}
0 & 1 & 0 & 0 & 0 & 0 & 0 & (32) \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & (-2) \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & (30)
\end{array}
\]

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)
then the top number is the 2’s-complement representation of $-2^{k+l}$. The recoded multiplier now consists of the part corresponding to the second number, with -1 added in position $k+l$.

For example, the multiplier 110110 is recoded as 0-1+10-10.
The Booth technique for recoding multipliers is summarized in Figure 13a. The transformation $011\ldots 110 = \cdot +100\ldots 0 -10$ is called skipping over Is. This term is derived from the case in which the multiplier has its 1s 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 1s 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 13a. Fig 13b is the flow chart to explain the Booth algorithm.

The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s. The speed gained by skipping over 1s depends on the data. On average, the speed of doing multiplication with the Booth algorithm is the same as with the normal algorithm.

**FAST MULTIPLICATION:**

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),( -1 +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. Figure 14a shows an example of bit-pair recoding of the multiplier in Figure 11, and Figure 14b shows a table of the multiplicand.
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.
FIG – 15: Multiplication requiring n/2 summands.

**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 by 13, 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 subtraction. 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 subtraction 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 repositioned for another subtraction.

![Longhand division examples.](FIG - 16)
FIG – 17: Binary Division
FIG – 18: Restoring 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 q0 to 0 and add M back to A (that is, restore A); otherwise, set q0 to 1.

Figure 18 shows a 4-bit example as it would be processed by the circuit in Figure 17.

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 q0 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 n times:
1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 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

**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 anywhere 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

\[
F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-(n-1)} \times 2^{-(n-1)}
\]

The range of values representable in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to ±2.15 x 10^9. If we consider them to be fractions, the range is approximately ±4.55 x 10^10 to ±1. Neither of these ranges is sufficient for scientific calculations, which might involve parameters like Avogadro's number (6.0247 x 10^{23} mole^{-1}) or Planck's constant (6.6254 x 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 $6.0247 \times 10^{23}$, $6.6254 \times 10^{-27}$, $-1.0341 \times 10^{-23}$, and so on. These numbers are said to be given to five significant 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.

**FLOATING-POINT DATA**

Floating-point representation of numbers needs two registers. The first represents a signed fixed-point number and the second, the position of the radix point. For example, the representation of the decimal number $+6132.789$ is as follows:

The first register has a 0 in the most significant flip-flop position to denote a plus. The magnitude of the number is stored in a binary code in 28 flip-flops, with each decimal digit occupying 4 flip-flops. The number in the first register is considered a fraction, so the decimal point in the first register is fixed at the left of the most significant digit. The second register contains the decimal number $+4$ (in binary code) to indicate that the actual position of the decimal point is four decimal positions to the right. This representation is equivalent to the number expressed by a fraction times 10 to an exponent, i.e., $+6132.789$ is represented as $+.6132789 \times 10^{+4}$. Because of this analogy, the contents of the first register are called the coefficient (and sometimes mantissa or fractional part) and the contents of the second register are called the exponent (or characteristic).
The position of the actual decimal point may be outside the range of digits of the coefficient register. For example, assuming sign-magnitude representation, the following contents:

```
Coeficient  exponent
02601000    104
```

represent the number $+0.2601 \times 10^{-4} = +0.000026010000$, which produces four more 0's on the left. On the other hand, the following contents:

```
Coefficient  exponent
12601000    012
```

represent the number $-0.2601 \times 10^{-12} = -260100000000$, which produces five more 0's on the right.

In these examples, we have assumed that the coefficient is a fixed-point fraction. Some computers assume it to be an integer, so the initial decimal point in the coefficient register is to the right of the least significant digit.

Another arrangement used for the exponent is to remove its sign bit altogether and consider the exponent as being "biased." For example, numbers between $10^{+49}$ and $10^{+50}$ can be represented with an exponent of two digits (without sign bit) and a bias of 50. The exponent register always contains the number $E + 50$, where $E$ is the actual exponent. The subtraction of 50 from the contents of the register gives the desired exponent. This way, positive exponents are represented in the register in the range of numbers from 50 to 99. The subtraction of 50 gives the positive values from 00 to 49. Negative exponents are represented in the register in the range of 00 to 49. The subtraction of 50 gives the negative values in the range of -50 to -1.

A floating-point binary number is similarly represented with two registers, one to store the coefficient and the other, the exponent. For example, the number $+1001.110$ can be represented as follows:

```
Sign     Initial binary point
0 1 0 0 1 1 1 0 0 0
```

```
Sign
0 0 1 0 0
```
The coefficient register has ten flip-flops: one for sign and nine for magnitude. Assuming that the coefficient is a fixed-point fraction, the actual binary point is four positions to the right, so the exponent has the binary value +4. The number is represented in binary as .100111000 \( \times 10^{+4} \) (remember that 10\(^{+4}\) in binary is equivalent to decimal \(2^4\)).

Floating-point is always interpreted to represent a number in the following form:

\[ c - r^e \]

where \(c\) represents the contents of the coefficient register and \(e\), the contents of the exponent register. The radix (base) \(r\) and the radix-point position in the coefficient are always assumed. Consider, for example, a computer that assumes integer representation for the coefficient and base 8 for the exponent. The octal number \( +17.32 = +1732 \times 8^{-2} \) will look like this

\[
\begin{array}{c}
\text{sign} \\
01732 \\
\text{initial octal point} \\
\end{array}
\quad
\begin{array}{c}
\text{sign} \\
102 \\
\text{exponent} \\
\end{array}
\]

When the octal representation is converted to binary, the binary value of the registers becomes:

\[
\begin{array}{c}
000111101101 \\
\text{Coefficient} \\
\end{array}
\quad
\begin{array}{c}
1000010 \\
\text{exponent} \\
\end{array}
\]

A floating point number is said to normalized if the most significant position of the coefficient contains a nonzero digit. In this way, the coefficient has no leading zeros and contains the maximum possible number of significant digits. Consider, for example, a coefficient register that can accommodate five decimal digits and a sign. The number \( +.00357 \times 10^3 = 3.57 \) is not normalized because it has two leading zeros and the unnormalized coefficient is accurate to three significant digits. The number can be normalized by shifting the coefficient two positions to the left and decreasing the exponent by 2 to obtain: \( +.35700 \times 10^1 = 3.5700 \), which is accurate to five
significant digits.

Arithmetic operations with floating-point number representation are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is more convenient because of the scaling problems involved with fixed-point operations. Many computers have a built-in capability to perform floating-point arithmetic operations. Those that do not have this hardware are usually programmed to operate in this mode.

Adding or subtracting two numbers in floating-point representation requires first an alignment of the radix point, since the exponent part must be made equal before the coefficients are added or subtracted. This alignment is done by shifting one coefficient while its exponent is adjusted until it is equal to the other exponent. Floating-point multiplication or division requires no alignment of the radix point. The product can be formed by multiplying the two coefficients and adding the two exponents. Division is accomplished from the division with the coefficients and the subtraction of the divisor exponent from the exponent of the dividend.

**IEEE Standard for Floating-Point Numbers:**

We start with a general form and size for floating-point numbers in the decimal system and then relate this form to a comparable binary representation. A useful form is

\[ \pm X_1, X_2, X_3, X_4, X_5, X_6, X_7 \times 10^{\pm y_1 y_2} \]

where \( X_i \) and \( F_i \) are decimal digits. Both the number of significant digits (7) and the exponent range (±99) are sufficient for a wide range of scientific calculations. It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits, which is a standard computer word length. A 24-bit mantissa can approximately represent a 7-digit decimal number, and an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. One bit is needed for the sign of the number. Since the leading nonzero bit of a normalized binary mantissa must be a 1, it does not have to be included explicitly in the representation. Therefore, a total of 32 bits is needed.

This standard for representing floating-point numbers in 32 bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE) [1]. The standard describes both the representation and the way in which the four basic arithmetic operations are to be performed. The 32-bit representation is given in Figure 20a. The sign of the number is given in the first bit,
followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent, $E$, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$.

This is called the excess-127 format. Thus, $E'$ is in the range $0 \leq E' \leq 255$. The end values of this range, 0 and 255, are used to represent special values, as described below. Therefore, the range of $E'$ for normal values is $1 \leq E' \leq 254$. This means that the actual exponent, $E$, is in the range $-126 \leq E \leq 127$. The excess-$x$ representation for exponents enables efficient comparison of the relative sizes of two floating-point numbers.

The last 23 bits represent the mantissa. Since binary normalization is used, the most significant bit of the mantissa is always equal to 1. This bit is not explicitly represented: it is assumed to be to the immediate left of the binary point. Hence, the 23 bits stored in the $M$ field actually represent the fractional part of the mantissa, that is, the bits stored the right of the binary point. An example of a single-precision floating-point number is shown in Figure 20.
The 32-bit standard representation in Figure 20a is called a single-precision representation because it occupies a single 32-bit word. The scale factor has a range of $2^{-26}$ to $2^{+127}$, which is approximately equal to $10^{\pm38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. To provide more precision and range for floating-point numbers, the IEEE standard also specifies a double precision format, as shown in Figure 20. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent $E'$ has the range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values. Thus, the actual exponent $E$ is in the range $-1022 \leq E \leq 1023$, providing scale factors of $2^{-1022}$ to $2^{1023}$ (approximately $10^{\pm308}$). The 53-bit mantissa provides precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions are intended to provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. For example, the dot product of two vectors of numbers can be computed by accumulating the sum of products: extended precision. The inputs are given in a standard precision, either single or double, and the answer is truncated to the same precision. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. In addition to requiring the four basic arithmetic operations, the standard requires that the operations of remainder, square root, and conversion between binary and decimal representations be provided.

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. Figure 21 shows an unnormalized value, $0.0010110... \times 2^9$ and its normalized version, $1.0110... \times 2^6$. Since the scale factor is in the form $2^i$, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than +127. In the first case, we say that underflow has occurred, and in the second case, we say that overflow has occurred. Both underflow and overflow are arithmetic exceptions that are considered below.
Special Values

The end values 0 and 255 of the excess-127 exponent $E'$ are used to represent special values. When $E' = 0$ and the mantissa fraction $M$ is zero, the value exact 0 is represented. When $E' = 255$ and $M = 0$, the value $\infty$ is represented, where $\infty$ is the result of dividing a normal number by zero. The sign bit is still part of these representations, so there are $\pm 0$ and $\pm \infty$ representations. When $E' = 0$ and $M \neq 0$, denormal numbers are represented. Their value is $\pm O.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and $M$ is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for gradual underflow, providing an extension of the range of normal representable numbers that is useful in dealing with very small numbers in certain situations. When $E' = 255$ and $M \neq 0$, the value represented is called Not a Number (NaN). A NaN is the result of performing an invalid operation such as $0/0$ or $\sqrt{-1}$.

Exceptions

In conforming to the IEEE Standard, a processor must set exception flags if any of the following occur in performing operations: underflow, overflow, and divide by zero, inexact, invalid. We have already mentioned the first three. Inexact is the name for a result that requires rounding in order to be represented in one of the normal
formats. An invalid exception occurs if operations such as 0/0 or $\sqrt{-1}$ are attempted. When exceptions occur, the results are set to special values. If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

**Arithmetic Operations on Floating-Point Numbers:**

The rules apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations. Intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively & overflow or an underflow may occur. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add $2.9400 \times 10^2$ to $4.3100 \times 10^4$. We rewrite $2.9400 \times 10^2$ as $0.0294 \times 10^4$ and then perform addition of the mantissas to get $4.3394 \times 10^4$. The rule for addition and subtraction can be stated as follows:

**Add/Subtract Rule**

The steps in addition (FA) or subtraction (FS) of floating-point numbers $(s_1, e_1, f_1)$ and $(s_2, e_2, f_2)$ are as follows.

1. Unpack sign, exponent, and fraction fields. Handle special operands such as zero, infinity, or NaN (not a number).
2. Shift the significand of the number with the smaller exponent right by $|e_1 - e_2|$ bits.
3. Set the result exponent $e_r$ to $\max(e_1, e_2)$.
4. If the instruction is FA and $s_1 = s_2$ or if the instruction is FS and $s_1 \neq s_2$ then add the significands; otherwise subtract them.
5. Count the number $z$ of leading zeros. A carry can make $z = -1$. Shift the result significand left $z$ bits or right 1 bit if $z = -1$.
6. Round the result significand, and shift right and adjust $z$ if there is rounding overflow, which is a carry-out of the leftmost digit upon rounding.
7. Adjust the result exponent by $e_r = e_r - z$, check for overflow or underflow, and pack the result sign, biased exponent, and fraction bits into the result word.
Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

**Multiply Rule**
1. Unpack signs, exponents, and significands. Handle exceptional operands.
2. Compute result sign, \( S_r = S_1 \oplus S_2 \), add exponents, \( e_r = e_1 + e_2 \), and multiply significands, \( f_r = f_1 \times f_2 \).
3. If necessary, normalize by one left shift and decrement result exponent. Round and shift right if rounding overflow occurs.
4. If the exponent is too positive, handle overflow, and if it is too negative, handle underflow.
5. Pack result, encoding or reporting exceptions

**Divide Rule**
2. Compute result sign, \( S_r = S_1 \oplus S_2 \), subtract exponent of divisor from that of
dividend $e_r = e_1 - e_2$, and divide the significands, $f_r = f_1 / f_2$

3. If necessary, normalize by one right shift and increment result exponent. Round and correct for rounding overflow.

4. Handle overflow and underflow on exponent range as in multiply.

5. Pack result and treat exceptions.

**Implementing Floating-Point Operations:**

If all numbers have the same scale factor, addition and subtraction are easy, since $f \times 2^e + g \times 2^e = (f+g) \times 2^e$ provided that $(f+g)$ does not overflow. The scale changes in multiplication and division because, even if both operands are scaled the same,

$$ (f \times 2^e) \cdot (g \times 2^e) = (f \cdot g) \times 2^{2e} \text{ and } (f \times 2^e) + (g \times 2^e) = f/g $$

Multiplication and division compute a new scale factor for the result from those of the operands as shown below;

$$ (f_1 \times 2^{e_1}) + (f_2 \times 2^{e_2}) = (f_1/f_2) \times 2^{e_1-e_2} $$

$$ (f_1 \times 2^{e_1}) \cdot (f_2 \times 2^{e_2}) = f_1 \cdot f_2 \times 2^{e_1+e_2} $$

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In most general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

An example of the implementation of floating-point operations is shown in Figure 22. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 20. Following the Add/Subtract rule, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, $n$, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_{A} - E'_{B}$, or $n$, is sent to the SHIFTER unit. If $n$ is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result.
FIG – 22: Floating point Arithmetic

The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network, if the sign is 0, then $E'_A > E'_B$ and the mantissas $M_A$ and $M_B$ are sent straight through the SWAP network. This results in $M_B$ being sent to the SHIFTER, to be shifted $n$ positions to the right. The other mantissa, $M_A$, is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, $E'$, is tentatively determined as $E'_A$ if $E'_A > E'_B$, or $E'_B$ if $E'_A < E'_B$, based on the sign of the difference resulting from comparing exponents in step 1.

Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands ($S_A$ and $S_B$) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, $S_R$. For example, if A is negative ($S_A = 1$), B is positive ($S_B = 0$), and the operation is $A - B$, then the mantissas are added and the sign of the result is negative ($S_R = 1$). On the other hand, if A and B are both positive and the operation is $A - B$, then the mantissas are subtracted. The sign of the result, $S_R$, now depends on the mantissa subtraction operation. For instance, if $E'_A > E'_B$, then $M_A$ — (shifted $M_B$) is positive and the result is positive. But if $E'_B = E'_A$, then $M_B$ — (shifted $M_A$) is positive and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When $E'_A = E'_B$ and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be
able to construct the complete truth table for the CONTROL network.

Step 4 of the Add/Subtract rule consists of normalizing the result of step 3, mantissa M. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M. The normalized value is truncated to generate the 24-bit mantissa, \( M_R \), of the result. The value X is also subtracted from the tentative result exponent \( E' \) to generate the true result exponent, \( E'_R \). Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form 1 \( .xx... \) were added. The vector M would then have the form 1 \( .xxx... \). This would correspond to an X value of -1 in the figure.

Let us consider the actual hardware that is needed to implement the blocks in Figure 22. The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign-and-magnitude form, we must modify some of our earlier discussions. A combination of 1's-complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation. If a design with a modest logic gate count is required, the operations can be implemented with shift registers. However, they can also be built as combinational logic units for high-performance, but in that case, a significant number of logic gates is needed. In high-performance processors, a significant portion of the chip area is assigned to floating-point operations.

**KEY CONCEPTS: FLOATING – POINT REPRESENTATION**

- Floating-point numbers are generally represented with the significand having a sign-magnitude representation and the exponent having a biased representation. The exponent base is implicit.
- Floating-point standards must specify the base, the representation, and the number of bits devoted to exponent and significand.
- Normalization eliminates multiple representations for the same value, and simplifies comparisons and arithmetic computations.
- Floating-point arithmetic operations are composed of multiple fixed-point operations on the exponents and significands.
- Floating-point addition and subtraction are more complicated than multiplication and division because they require comparison of exponents and shifting of the significands to "line up the binary points" prior to the actual addition or subtraction operation.

Floating-point multiplication and division, on the other hand, require only a maximum 1-bit shift of the significand to normalize the numbers.