UNIT II

Addition and subtraction of two numbers are basic operations at the machine-instruction level in all computers. These operations, as well as other arithmetic and logic operations, are implemented in the arithmetic and logic unit (ALU) of the processor. In this chapter, we present the logic circuits used to implement arithmetic operations. The time needed to perform addition or subtraction affects the processor's performance. Multiply and divide operations, which require more complex circuitry than either addition or subtraction operations, also affect performance. We present some of the techniques used in modern computers to perform arithmetic operations at high speed. Operations on floating-point numbers are also described.

ADDITION AND SUBTRACTION OF SIGNED NUMBERS

Figure shows the truth table for the sum and carry-out functions for adding equally weighted bits xi and yi in two numbers X and Y . The figure also shows logic expressions for these functions, 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 ci to represent the carry-in to stage i, which is the same as the carry-out from stage (i − 1). The logic expression for si in Figure 9.1 can be implemented with a 3-input XOR gate, used in Figure a as part of the logic required for a single stage of binary addition. The carry-out function, ci+1, is implemented with an AND-OR circuit, as shown. A convenient symbol for the complete circuit for a single stage of addition, called a full adder (FA), is also shown in the figure.

A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, as shown in Figure b. Since the carries must propagate, or ripple, through this cascade, the configuration is called a ripple-carry adder. The carry-in, c0, into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number. 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 c.

Addition/Subtraction Logic Unit

The n-bit adder in Figure 9.2b can be used to add 2's-complement numbers X and Y , where the xn−1 and yn−1 bits are the sign bits. The carry-out bit cn is not part of the answer. It occurs when the signs of the two operands are the same, but 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

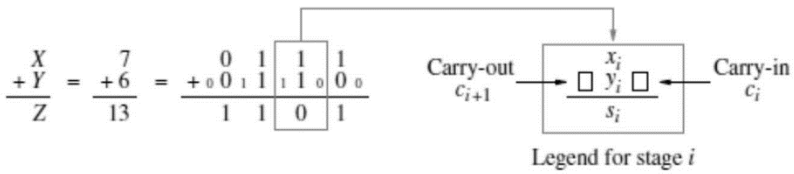
$$\text{Overflow} = x_{n-1}y_{n-1}\overline{s}_{n-1} + \overline{x}_{n-1}\overline{y}_{n-1}s_{n-1}$$

It can also be shown that overflow occurs when the carry bits cn and cn−1 are different. Therefore, a simpler circuit for detecting overflow can be obtained by implementing the expression cn $\oplus$ cn−1 with an XOR gate.

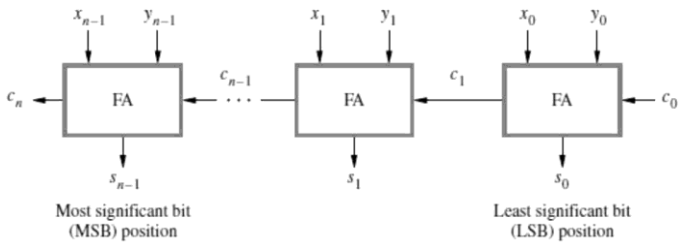
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

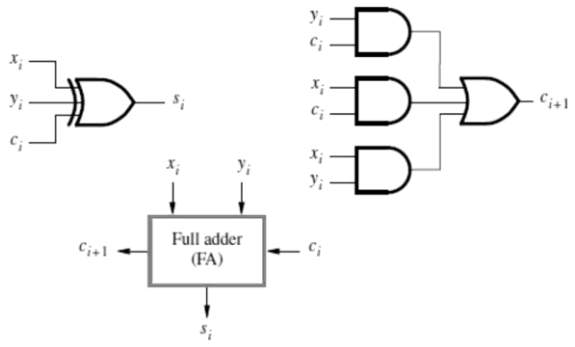
Example:



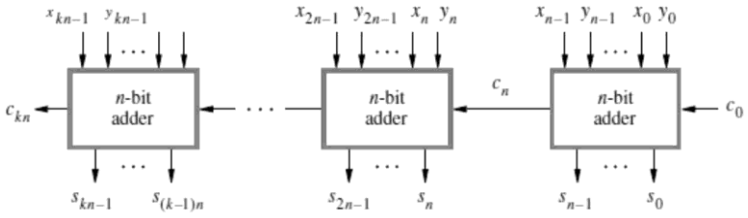
In order to perform the subtraction operation X − Y on 2's-complement numbers X and Y , we form the 2's-complement of Y and add it to X . The logic circuit shown in Figure can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, c0, of 0. When the Add/Sub control line is set to 1, the Y number is 1's-complemented (that is, bit-complemented) by the XOR gates and c0 is set to 1 to complete the 2's-complementation of Y . Recall 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 to detect the overflow condition cn $\oplus$ cn−1.



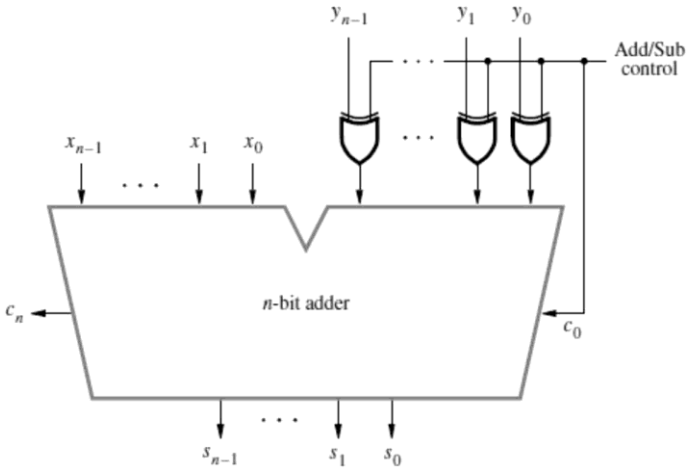
(b) An *n*-bit ripple-carry adder



(a) Logic for a single stage



(c) Cascade of k n-bit adders



Design of Fast Adders

If an n-bit ripple-carry adder is used in the addition/subtraction circuit, it may have too much delay in developing its outputs, s0 through sn−1 and cn. Whether or not the delay incurred is acceptable can be decided only in the context of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology used in fabricating the network and on the number of gates in the paths from inputs to outputs. The delay through any combinational circuit 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 circuit. In the case of the n-bit ripple-carry adder, the longest path is from inputs x0, y0, and c0 at the LSB position to outputs cn and sn−1 at the most-significant-bit (MSB) position.

Using the implementation indicated in Figure a, cn−1 is available in 2(n−1) gate delays, and sn−1 is correct oneXORgate delay later. The final carry-out, cn, is available after 2n gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure 9.3, all sum bits are available in 2n gate delays, including the delay through the XOR gates on the Y input. Using the implementation cn $\oplus$ cn−1 for overflow, this indicator is available after 2n + 2 gate delays.

Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology. The second approach is to use a logic gate network called a carry-lookahead network, which is described in the ***Previous VLSI design Notes***.

CARRY LO0K-AHEAD ADDER:

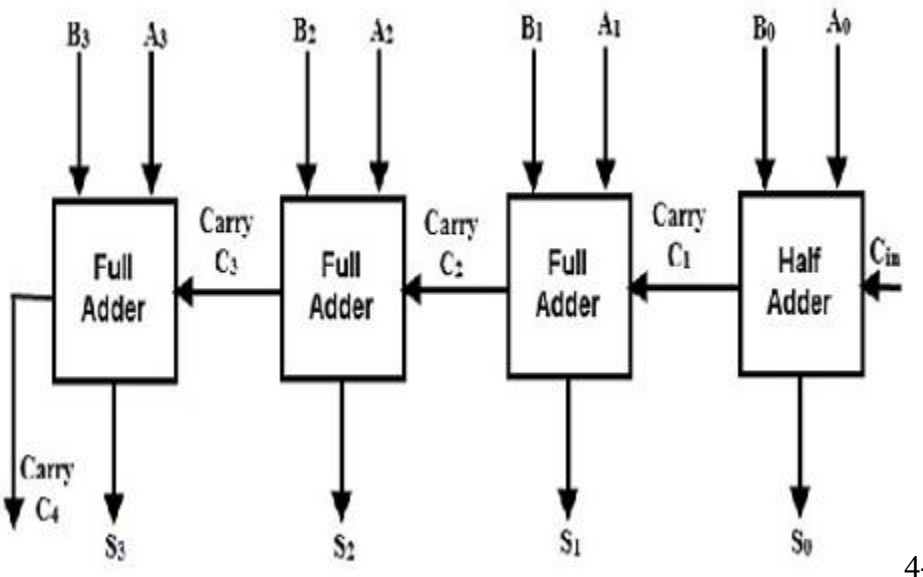
A digital computer must contain circuits which can perform arithmetic operations such as addition, subtraction, multiplication, and division. Among these, addition and subtraction are the basic operations whereas multiplication and division are the repeated addition and subtraction respectively.

To perform these operations 'Adder circuits' are implemented using basic logic gates. Adder circuits are evolved as Half-adder, Full-adder, Ripple-carry Adder, and Carry Look-ahead Adder. Among these Carry Look-ahead Adder is the faster adder circuit. It reduces the propagation delay, which occurs during addition, by using more complex hardware circuitry. It is designed by transforming the ripple-carry Adder circuit such that the carry logic of the adder is changed into two-level logic.

4-Bit Carry Look-ahead Adder

In parallel adders, carry output of each full adder is given as a carry input to the next higher-order state. Hence, these adders it is not possible to produce carry and sum outputs of any state unless a carry input is available for that state.

So, for computation to occur, the circuit has to wait until the carry bit propagated to all states. This induces carry propagation delay in the circuit.



4-bit-Ripple-Carry-Adder

Consider the 4-bit ripple carry adder circuit above. Here the sum S3 can be produced as soon as the inputs A3 and B3 are given. But carry C3 cannot be computed until the carry bit C2 is applied whereas C2 depends on C1. Therefore to produce final steady-state results, carry must propagate through all the states. This increases the carry propagation delay of the circuit.

The propagation delay of the adder is calculated as "the propagation delay of each gate times the number of stages in the circuit". For the computation of a large number of bits, more stages have to be added, which makes the delay much worse. Hence, to solve this situation, Carry Look-ahead Adder was introduced.

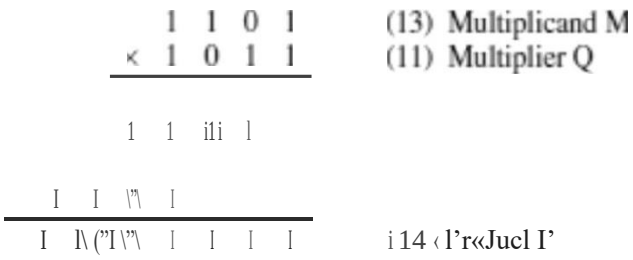
MULTIPLICATION ALGORITHM

The usual algorithm for multiplying integers by hand is illustrated for the binary system. The product of two, unsigned, n-digit numbers can be accommodated in 2n digits, so the product of the two 4-bit numbers in this example is accommodated in 8bits, as shown. In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position. If the multiplier bit is 0, then 0s are entered, as in the third row of the example. The product is computed one bit at a time by adding the bit columns from right to left and propagating carry values between columns.

Array Multiplier for Unsigned Numbers

Binary multiplication of unsigned operands can be implemented in a combinational, two dimensional, logic array, as shown in Figure b for the 4-bit operand case. The main 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, qi . Each row i, where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i + 1), if qi = 1. If qi = 0, PPi is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.



(a) Manual multiplication algorithm

Multiplication of Signed Numbers

We now discuss multiplication of 2's-complement operands, generating a double- length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected 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. Figure shows an example in which a 5-bit signed operand, −13, is the multiplicand. It is multiplied by +11 to get the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.

					1	0	0	1	1	(–13)
				×	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		
Sign extension is shown in blue	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)

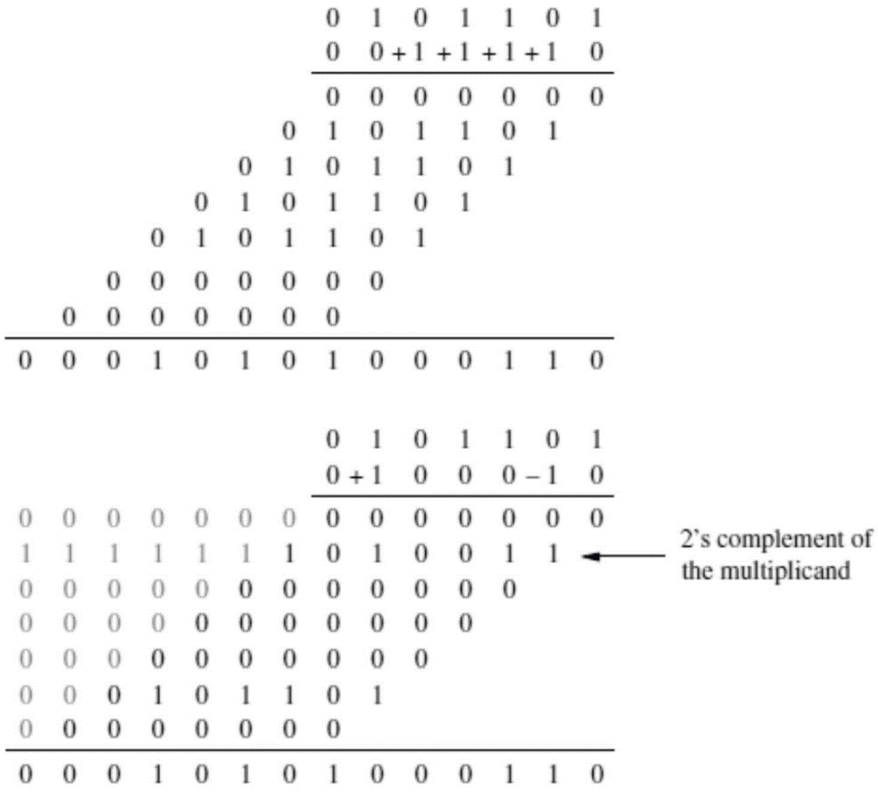
For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm.

The Booth Algorithm

The Booth algorithm [1] generates a 2n-bit product and treats both positive and negative 2'scomplement n-bit operands uniformly. To understand the basis of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers:

$$
\begin{array}{rll}
 & 0100000 & (32) \\
- & 0000010 & (2) \\
\hline
 & 0011110 & (30)
\end{array}
$$

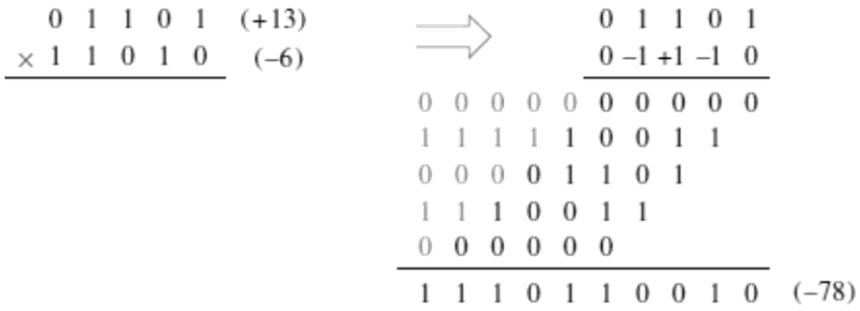
This suggests that the product can be generated by adding 25 times the multiplicand to the 2's-complement of 21 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0 +1 0 0 0 −1 0. In general, in the Booth algorithm, −1 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. Figure illustrates the normal and the Booth algorithms for the example just discussed. 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. Figure shows 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.



Normal and Booth multiplication schemes

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

Booth recoding of a multiplier



Booth multiplication with a negative multiplier

Suppose that the leftmost 0 of a negative number, X, is at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

Then the value of X is given by

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

The correctness of this expression for $V(X)$ is shown by observing that if X is formed as the sum of two numbers, as follows,

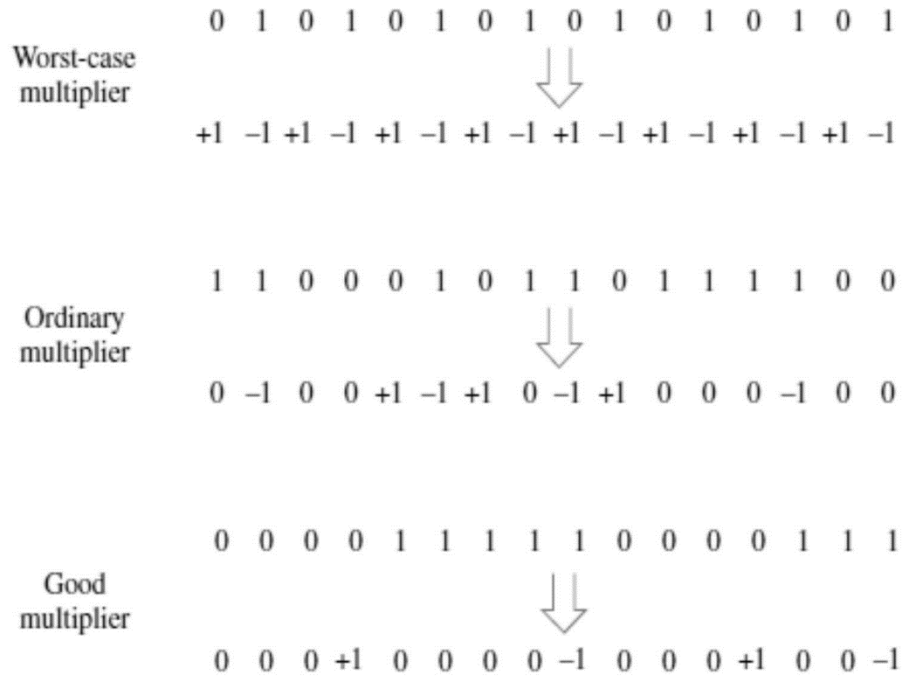
$$\begin{array}{rl} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline X = & 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

then the upper number is the 2’s-complement representation of −2k+1. The recoded multiplier now consists of the part corresponding to the lower number, with −1 added in position k + 1. For example, the multiplier 110110 is recoded as 0 −1 +1 0 −1 0. The Booth technique for recoding multipliers is summarized in Figure. The transformation 011 . . . 110⇒+1 0 0. . . 0 −1 0 is called skipping over 1s. 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) need to 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 Figure.

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Booth multiplier recoding table



Booth recoded multipliers

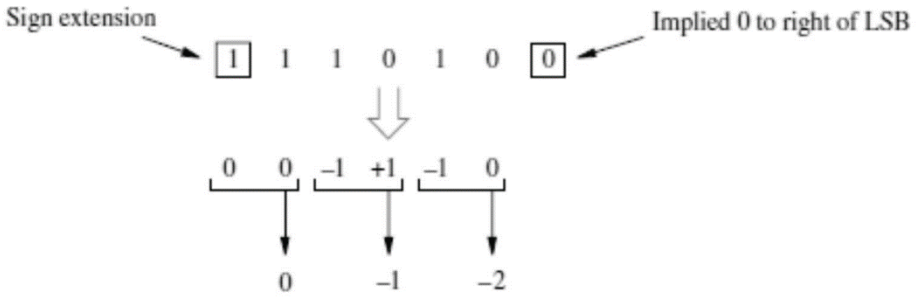
The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

Fast Multiplication

We now describe 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 leads to adding the summands in parallel.

Bit-Pair Recoding of Multipliers

A technique called bit-pair recoding of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following.



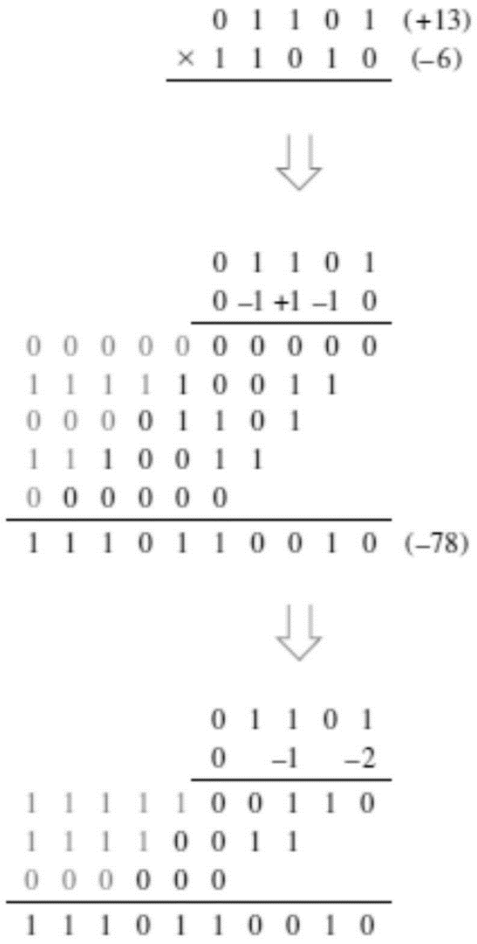
(a) Example of bit-pair recoding derived from Booth recoding

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

(b) Table of multiplicand selection decisions

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 ×M at position i + 1, the same result is obtained by adding +1 ×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.



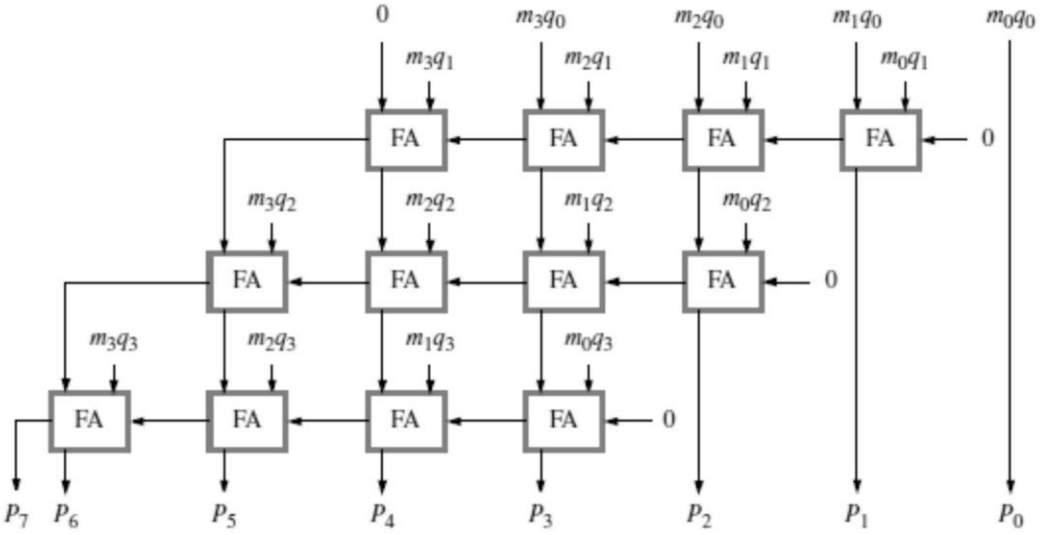
Multiplication requiring only n/2 summands

Figure a shows an example of bit-pair recoding of the multiplier in Figure and Figure b shows a table of the multiplicand selection decisions for all possibilities. The multiplication operation in Figure is shown in Figure as it would be computed using bit-pair recoding of the multiplier.

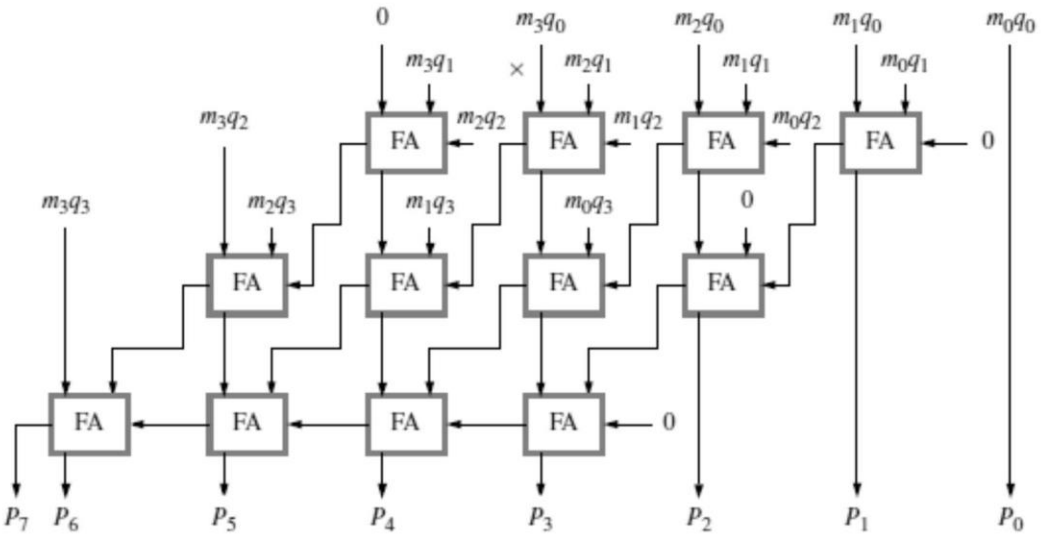
Carry-Save Addition of Summands

Multiplication requires the addition of several summands. A technique called carry- save addition (CSA) can be used to speed up the process. Consider the 4 × 4 multiplication array shown in Figure 9.16a. This structure is in the form of the array shown in Figure, in which the first row consists of just theAND gates that produce the four inputs m3q0, m2q0, m1q0, and m0q0. Instead of letting the carries ripple along the rows, they can be "saved" and

introduced into the next row, at the correct weighted positions, as shown in Figure 9.16b. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce the third summand bits m2q2, m1q2, and m0q2.



(a) Ripple-carry array



(b) Carry-save array

Ripple-carry and carry-save arrays for a 4 × 4 multiplier

Now, two inputs of each of three full adders in the second row are fed by the sum and carry outputs from the first row. The third input is used to introduce the bits m2q3, m1q3, and m0q3 of the fourth summand. The high-order bits m3q2 and m3q3 of the third and fourth summands are introduced into the remaining free full-adder inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in

the third row, which is a ripple-carry adder, to produce the final product bits. The delay through the carry-save array is somewhat less than the delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.

DIVISION ALGORITHM

We discussed the multiplication of unsigned numbers by relating the way the multiplication operation is done manually to the way it is done in a logic circuit. We use the same approach here in discussing integer division. We discuss unsigned-number division in detail, and then make some general comments on the signed-number case.

Integer Division

Figure shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that $27 - 26 = 1$ is less than 13, 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. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 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 another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the restoring division algorithm.

Restoring Division

Figure shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure. An n-bit positive divisor is loaded into registerMand 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.

```
      21                    10101
13 ) 274          1101 ) 100010010
     26                    1101
     14                   10000
     13                    1101
      1                    1110
                           1101
                              1
```

The following algorithm performs restoring division. Do the following three steps n times:

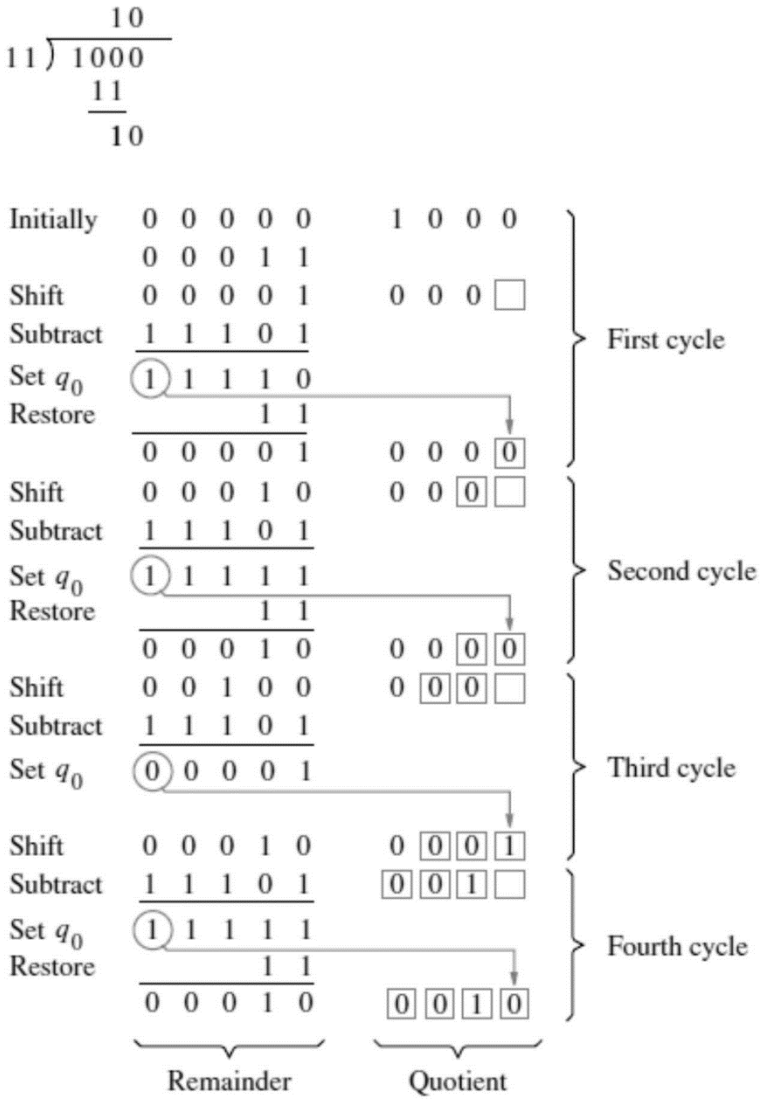
1. Shift A and Q left one bit position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q0 to 0 and add M back to A (that is, restore A); otherwise, set q0 to 1.

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

Non-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. IfAis 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 non-restoring division.

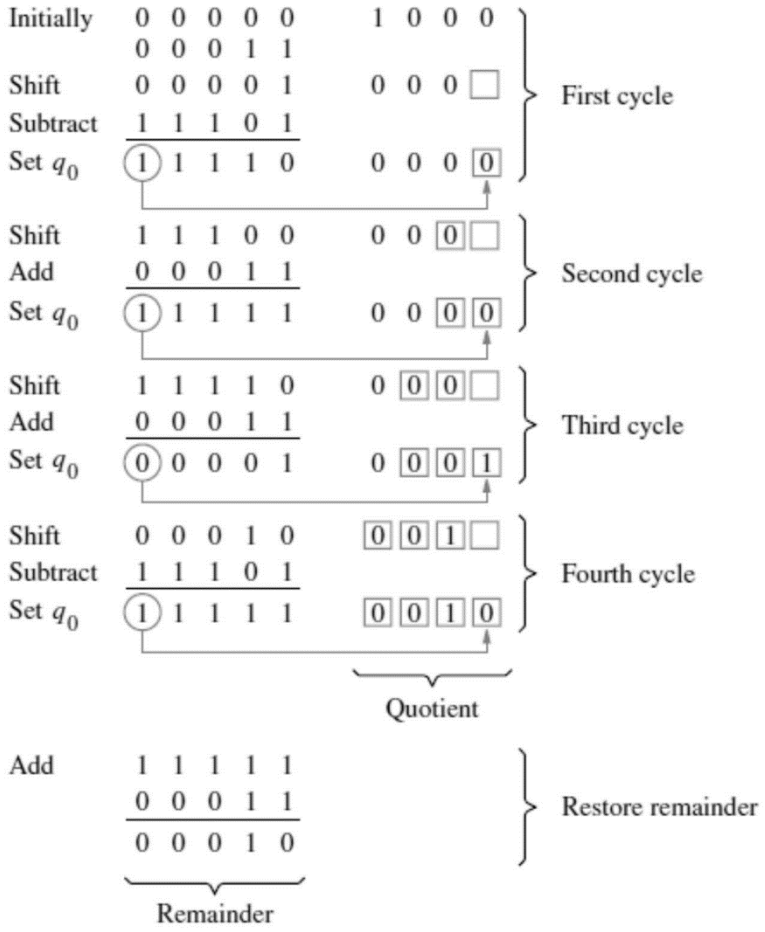
Stage 1: Do the following two steps n times:

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.

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

Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1. The logic circuitry in Figure can also be used to perform this algorithm, except that the Restore operations are no longer needed. One Add or Subtract operation is performed in each of the n cycles of stage 1, plus a possible final addition in Stage 2. Figure shows how the division example in Figure is executed by the non-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 change them into positive values. After using one of the algorithms just discussed, the signs of the quotient and the remainder are adjusted as necessary.

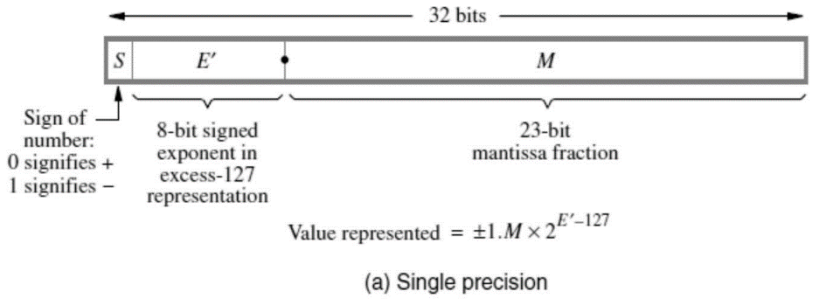
3. FLOATING POINT ARITHMETIC OPERATIONS

We know that the floating-point numbers and indicated how they can be represented in a 32-bit binary format. Now, we provide more detail on representation formats and arithmetic operations on floating-point numbers. The descriptions provided here are based on the 2008 version of IEEE (Institute of Electrical and Electronics Engineers) Standard.

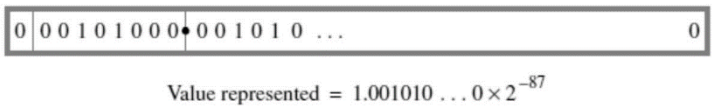
DECIMAL ARITHMETIC OPERATIONS

Recall from Chapter 1 that a binary floating-point number can be represented by

- A sign for the number
- Some significant bits
- A signed scale factor exponent for an implied base of 2

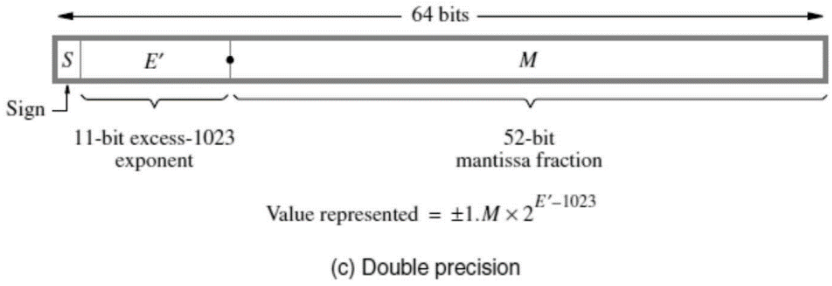


(a) Single precision



Value represented = $1.001010\ldots0 \times 2^{-87}$

(b) Example of a single-precision number



(c) Double precision

The basic IEEE format is a 32-bit representation, shown in Figure a. The leftmost bit represents the sign, S, for the number. The next 8 bits, E , represent the signed exponent of the scale factor (with an implied base of 2), and the remaining 23 bits, M, are the fractional part of

the significant bits. The full 24-bit string, B, of significant bits, called the mantissa, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa

$$B = 1.M = 1.b_{-1}b_{-2}\ldots b_{-23}$$

has the value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

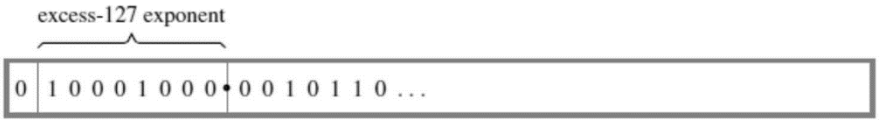
By convention, when the binary point is placed to the right of the first significant bit, the number is said to be normalized. Note that the base, 2, of the scale factor and the leading 1 of the mantissa are both fixed. They do not need to appear explicitly in the representation. Instead of the actual signed exponent, E, the value 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 later. 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 use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers. The 32-bit standard representation in Figure 9.26a is called a single-precision representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. An example of a single-precision floating-point number is shown in Figure b.

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a double-precision format, as shown in Figure c. 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, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a 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 provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations leading to a desired result.

For example, the dot product of two vectors of numbers involves accumulating a sum of products. The input vector components are given in a standard precision, either single or double, and the final answer (the dot product) is truncated to the same precision. All intermediate calculations should be done using extended precision to limit accumulation of errors. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. This is because they are usually evaluated by adding up a number of terms in a series representation. In addition to requiring the four basic arithmetic operations, the standard requires three additional operations to be provided: remainder, square root, and conversion between binary and decimal representations.



(There is no implicit 1 to the left of the binary point.)

Value represented = $+0.0010110\ldots \times 2^{9}$

(a) Unnormalized value

Value represented = $+1.0110\ldots \times 2^{6}$

(b) Normalized version

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent. Figure shows an unnormalized value, 0.0010110 . . . × 29, and its normalized version, 1.0110 . . . × 26. Since the scale factor is in the form 2i , 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.

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 0 is represented. When E = 255 and M = 0, the value ∞ is represented, where ∞ is the result of dividing a normal number

by zero. The sign bit is still used in these representations, so there are representations for ±0 and ±∞. When E = 0 and M = 0, denormal numbers are represented. Their value is ±0.M × 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. This is useful in dealing with very small numbers, which may be needed in certain situations. When E = 255 and M = 0, the value represented is called Not a Number (NaN). A NaN represents the result of performing an invalid operation such as 0/0 or √−1.

Exceptions

In conforming to the IEEE Standard, a processor must set exception flags if any of the following conditions arise when performing operations: underflow, overflow, 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 √−1 are attempted. When an exception occurs, the result is set to one of the 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

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules given below apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed.

Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections. When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ. Consider a decimal example in which we wish to add 2.9400 × 102 to 4.3100 × 104. We rewrite 2.9400 × 102 as 0.0294 ×

104 and then perform addition of the mantissas to get 4.3394 × 104. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127 to maintain the excess-127 representation.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

FUNDAMENTAL CONCEPTS

A typical computing task consists of a series of operations specified by a sequence of machine-language instructions that constitute a program. The processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor uses the program counter, PC, to keep track of the address of the next instruction to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence. A branch instruction may cause a different value to be loaded into the PC.

When an instruction is fetched, it is placed in the instruction register, IR, from where it is interpreted, or decoded, by the processor's control circuitry. The IR holds the instruction until its execution is completed. Consider a 32-bit computer in which each instruction is contained in one word in the memory, as in RISC-style instruction set architecture. To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer

notation, the required action is

$$IR \leftarrow [[PC]]$$

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is

$$PC \leftarrow [PC] + 4$$

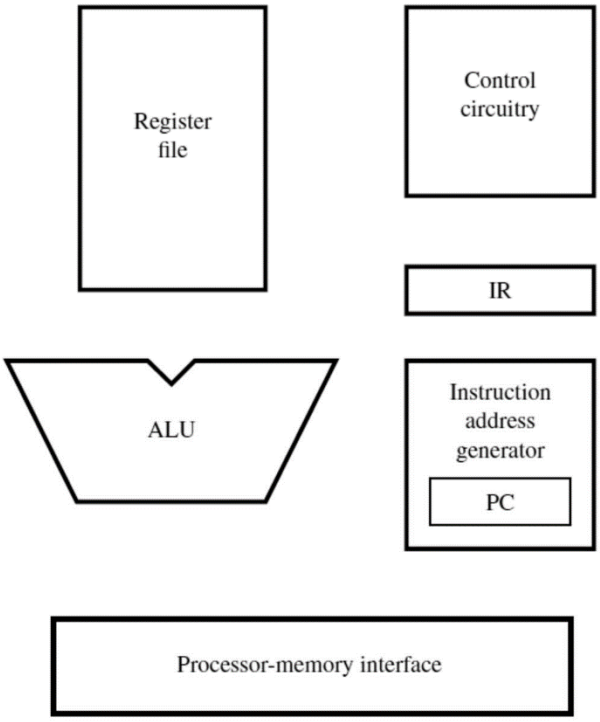
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the instruction fetch phase. Performing the operation specified in the instruction constitutes the instruction execution phase. With few exceptions, the operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

The hardware components needed to perform these actions are shown in Figure. The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations. The instruction

address generator updates the contents of the PC after every instruction is fetched. The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers. During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic unit (ALU), which performs the required computation. The results of the computation are stored in a register in the register file.



EXECUTION OF A COMPLETE INSTRUCTION

Atypical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. A clock signal is used to control the timing of data transfers. The registers comprise edge-triggered flip-flops into which new data are loaded at the active edge of the clock. In this chapter, we assume that the rising edge of the clock is the active edge. The clock period, which is the time between two successive rising edges, must be long enough to allow the combinational circuit to produce the correct result. Let us now examine the actions involved in fetching and executing instructions. We illustrate these actions using a few representative RISC-style instructions

SINGLE BUS ORGANIZATION

ALU and all the registers are interconnected via a Single Common Bus. Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR

and MAR respectively. (MDRà Memory Data Register, MAR à Memory Address Register). MDR has 2 inputs and 2 outputs. Data may be loaded into MDR either from memory-bus (external) or from processor-bus (internal).

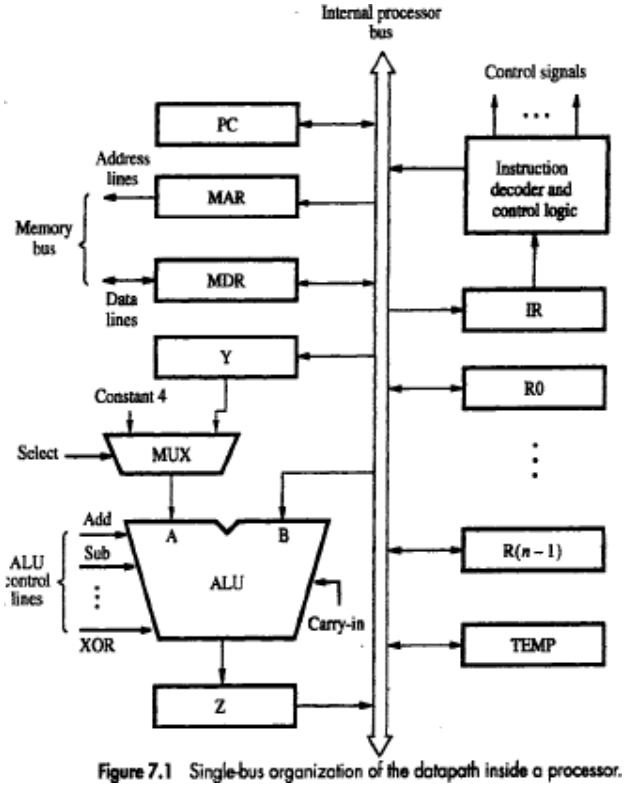


Figure 7.1 Single-bus organization of the datapath inside a processor.

MAR‟s input is connected to internal-bus; MAR‟s output is connected toexternal- bus. Instruction Decoder & Control Unit is responsible for issuing the control-signals to all the units inside the processor. We implement the actions specified by the instruction (loaded in the IR). Register R0 through R(n-1) are the Processor Registers. The programmer can access these registers for general-purpose use. Only processor can access 3 registers Y, Z & Temp for temporary storage during program-execution. The programmer cannot access these 3 registers. In ALU,1) "A‟ input gets the operand from the output of the multiplexer (MUX). 2) "B‟ input gets the operand directly from the processor-bus. There are 2 options provided for "A‟ input of the ALU. MUX is used to select one of the 2 inputs. MUX selects either output of Y or constant-value 4(which is used to increment PC content). An instruction is executed by performing one or more of the following operations:

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

Disadvantage: Only one data-word can be transferred over the bus in a clock cycle.
Solution: Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel.

REGISTER TRANSFERS

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: Riin & Riout. These are called Gating Signals. Riin=1 = data on bus is loaded into Ri. Riout=1 as content of Ri is placed on bus. Riout=0, makes bus can be used for transferring data from other registers.For example, Move R1, R2; This transfers the contents of register R1 to register R2. This can be accomplished as follows:

1) Enable the output of registers R1 by setting R1out to 1 (Figure 7.2). This places the contents of R1 on processor-bus.

2) Enable the input of register R2 by setting R2out to 1. This loads data from processor-bus into register R4.

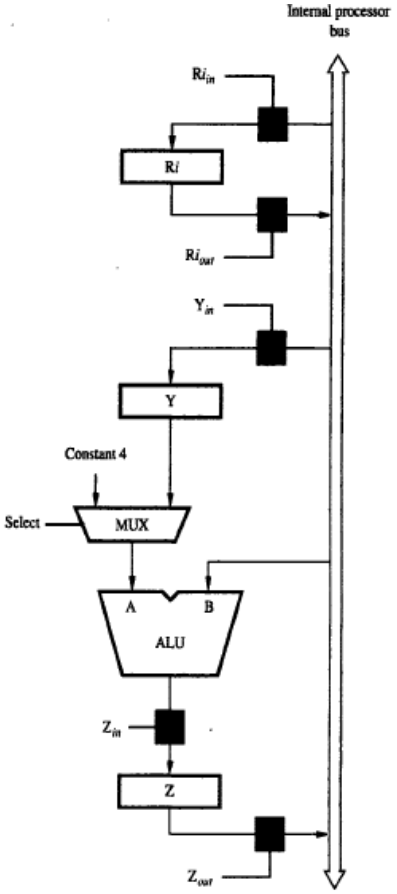


Figure 7.2 Input and output gating for the registers in Figure 7.1.

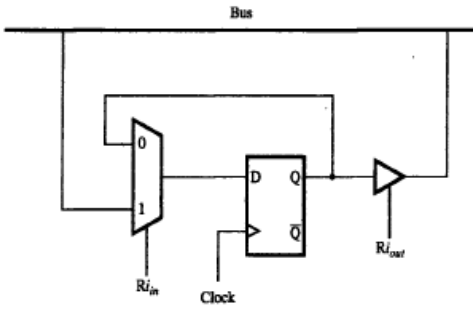


Figure 7.3 Input and output gating for one register bit.

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.

Input & Output Gating for one Register Bit

A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. Riin=1 makes mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. Riin=0 makes mux feeds back the value currently stored in flip-flop (Figure). Q output of flip-flop is connected to bus via a tri-state gate. Riout=0 makes gate's output is in the high-impedance state. Riout=1 makesthe gate drives the bus to 0 or 1, depending on the value of Q.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs. One of the operands is output of MUX and, the other operand is obtained directly from processor-bus. The result (produced by the ALU) is stored temporarily in register Z. The sequence of operations for [R3] [R1]+[R2] is as follows:

1) R1out, Yin
2) R2out, SelectY, Add, Zin
3) Zout, R3in

Instruction execution proceeds as follows:

Step 1 --> Contents from register R1 are loaded into register Y.

Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU; Addition is performed & Result is stored in the Z register.

Step 3 --> The contents of Z register is stored in the R3 register.

The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

CONTROL-SIGNALS OF MDR

The MDR register has 4 control-signals (Figure). MDRin & MDRout control the connection to the internal processor data bus & MDRinE & MDRoutE control the connection to the memory Data bus. MAR register has 2 control-signals. MARin controls the connection to the internal processor address bus & MARout controls the connection to the memory address bus.

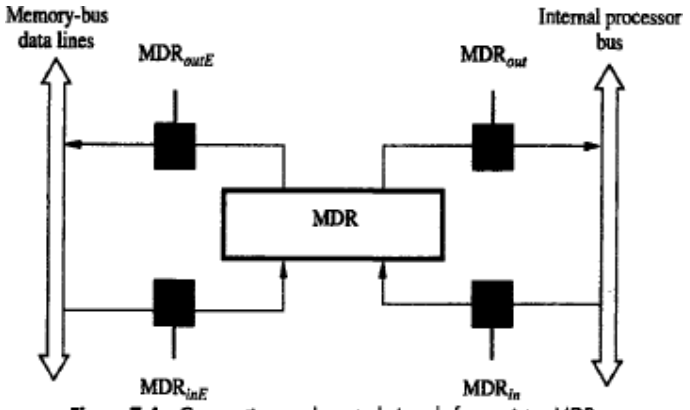
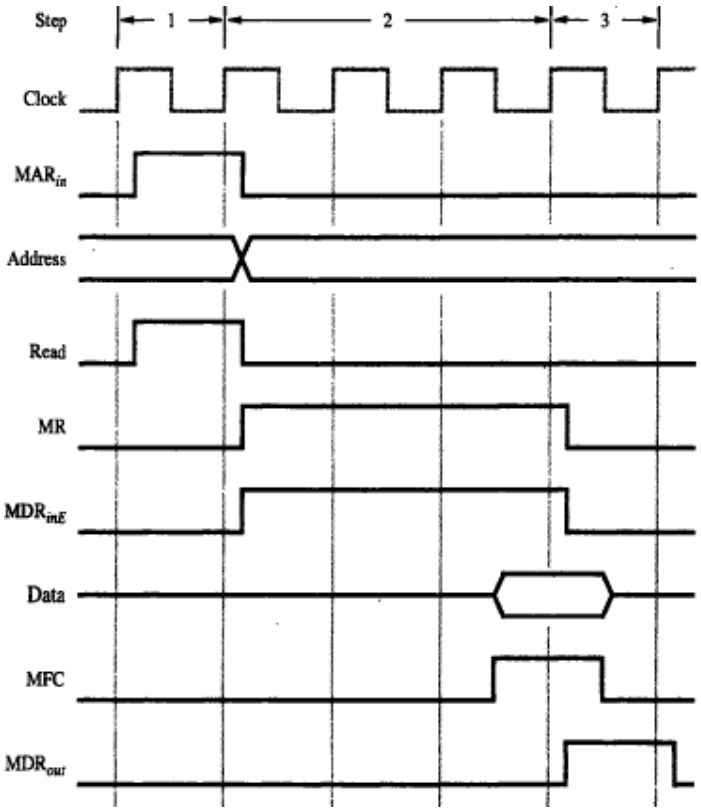


Figure 7.4 Connection and control signals for register MDR.

FETCHING A WORD FROM MEMORY

To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus. When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers. The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC makes Memory Function Completed). MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.



Consider the instruction Move (R1),R2. The sequence of steps is (Figure): R1out,

MARin, Read ;desired address is loaded into MAR & Read command is issued. MDRinE, WMFC; load MDR from memory-bus & Wait for MFC response from memory. MDRout, R2in; load R2 from MDR where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

Storing a Word in Memory

Consider the instruction Move R2,(R1). This requires the following sequence: R1out, MARin; desired address is loaded into MAR. R2out, MDRin, Write; data to be written are loaded into MDR & Write command is issued. MDRoutE, WMFC ;load data into memory location pointed by R1 from MDR.

EXECUTION OF A COMPLETE INSTRUCTION

Consider the instruction Add (R3),R1 which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instruction.
2) Fetch the first operand.
3) Perform the addition &
4) Load the result into R1.

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

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

Instruction execution proceeds as follows:

Step1: The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC‟s content), and the result is stored in Z.

Step2: Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3: Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the Fetch Phase. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7. The step 4 through 7 constitutes the Execution Phase.

Step4: Contents of R3 are loaded into MAR & a memory read signal is issued.

Step5: Contents of R1 are transferred to Y to prepare for addition.

Step6: When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7: Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

Control sequence for an unconditional branch instruction is as follows: Instruction execution proceeds as follows:

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

Figure 7.7 Control sequence for an unconditional Branch instruction.

Step 1-3: The processing starts & the fetch phase ends in step3.

Step 4: The offset-value is extracted from IR by instruction-decoding circuit. Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

Step 5: The result, which is the branch-address, is loaded into the PC.

The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address. The branch target address is usually obtained by adding the offset in the contents of PC. The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.

In case of conditional branch, we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to

load a new value into PC.

MULTIPLE BUS ORGANIZATION

The disadvantage of Single-bus organization is only one data-word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction. The solution to reduce the number of steps, most processors provide multiple internal-paths. Multiple paths enable several transfers to take place in parallel.

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

Figure 7.9 Control sequence for the instruction Add R4,R5,R6

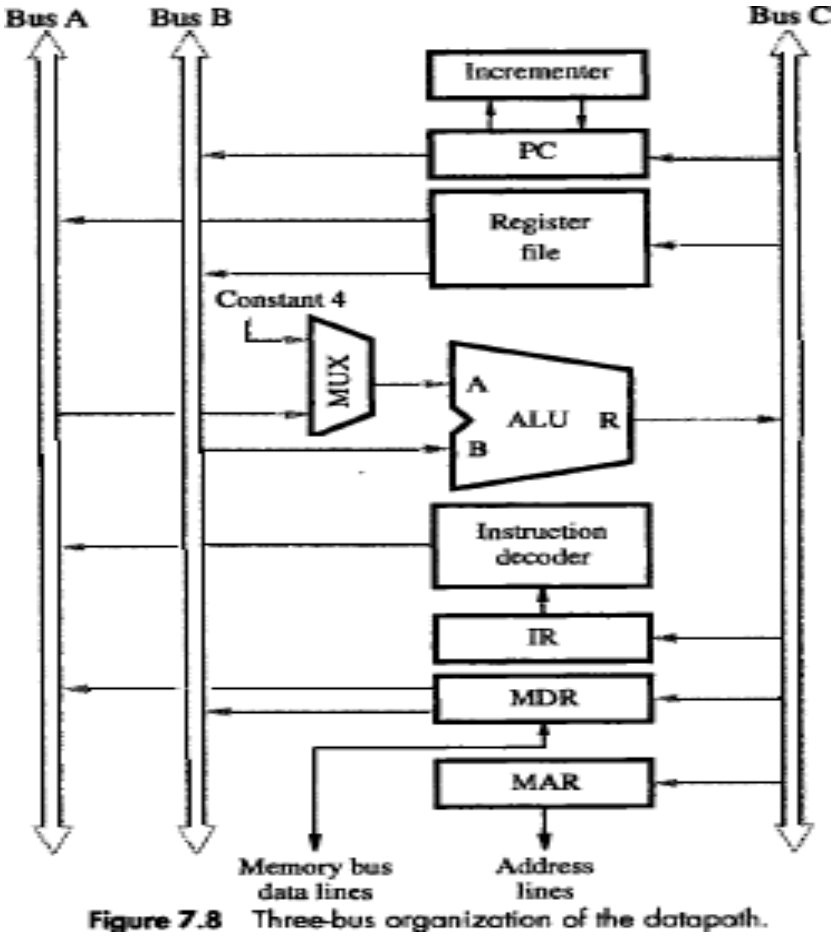


Figure 7.8 Three-bus organization of the datapath.

As shown in figure, three buses can be used to connect registers and the ALU of the processor. All general-purpose registers are grouped into a single block called the Register

File. Register-file has 3 ports:

1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.

2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.

Buses A and B are used to transfer source-operands to A & B inputs of ALU. The result is transferred to destination over bus C. Incrementer Unit is used to increment PC by 4. Instruction execution proceeds as follows:

Step 1: Contents of PC are passed through ALU using R=B control-signal & loaded into MAR to start memory Read operation. At the same time, PC is incremented by 4.

Step2: Processor waits for MFC signal from memory.

Step3: Processor loads requested-data into MDR, and then transfers them to IR.

Step4: The instruction is decoded and add operation takes place in a single step.

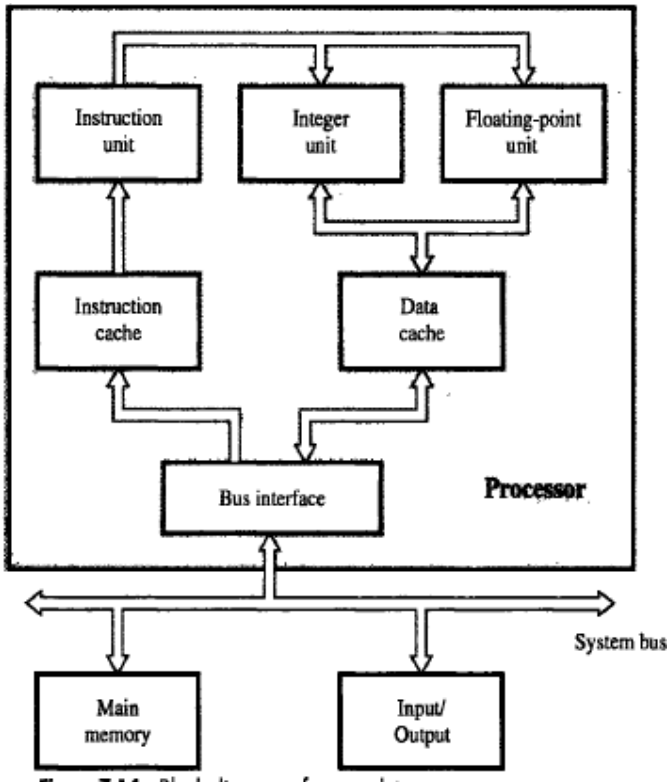


Figure 7.14 Block diagram of a complete processor.

COMPLETE PROCESSOR

This has separate processing-units to deal with integer data and floating-point data. Integer unit has to process integer data. (Figure). Floating unit has to process floating point

data. Data-Cache is inserted between these processing-units & main-memory. The integer and floating unit gets data from data cache. Instruction-Unit fetches instructions from an instruction-cache or from main-memory when desired instructions are not already in cache.

Processor is connected to system-bus & hence to the rest of the computer by means of a Bus Interface. Using separate caches for instructions & data is common practice in many processors today. A processor may include several units of each type to increase the potential for concurrent operations. The 80486 processor has 8-kbytes single cache for both instruction and data. Whereas the Pentium processor has two separate 8 kbytes caches for instruction and data.

Note: To execute instructions, the processor must have some means of generating the control-signals. There are two approaches for this purpose:

1) Hardwired control and

2) Microprogrammed control.

HARDWIRED CONTROL

Hardwired control is a method of control unit design (Figure). The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc. Decoder / Encoder Block is a combinational-circuit that generates required control-outputs depending on state of all its inputs. Instruction decoder decodes the instruction loaded in the IR. If IR is an 8 bit register, then instruction decoder generates 28(256 lines); one for each instruction. It consists of a separate output-lines INS1 through INSm for each machine instruction. According to code in the IR, one of the output-lines INS1 through INSm is set to 1, and all other lines are set to 0. Step-Decoder provides a separate signal line for each step in the control sequence. Encoder gets the input from instruction decoder, step decoder, external inputs and condition codes. It uses all these inputs to generate individual control-signals: Yin, PCout, Add, End and so on. For example (Figure 7.12), Zin=T1+T6.ADD+T4.BR; This signal is asserted during time-slot T1 for all instructions during T6 for an Add instruction. During T4 for unconditional branch instruction, when RUN=1, counter is incremented by 1 at the end of every clock cycle. When RUN=0, counter stops counting. After execution of each instruction, end signal is generated. End signal resets step counter. Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence
the name “hardwired”.

Advantage: Can operate at high speed.

Disadvantages: 1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.

2) It is costly and difficult to design.

3) The control unit is inflexible because it is difficult to change the design.

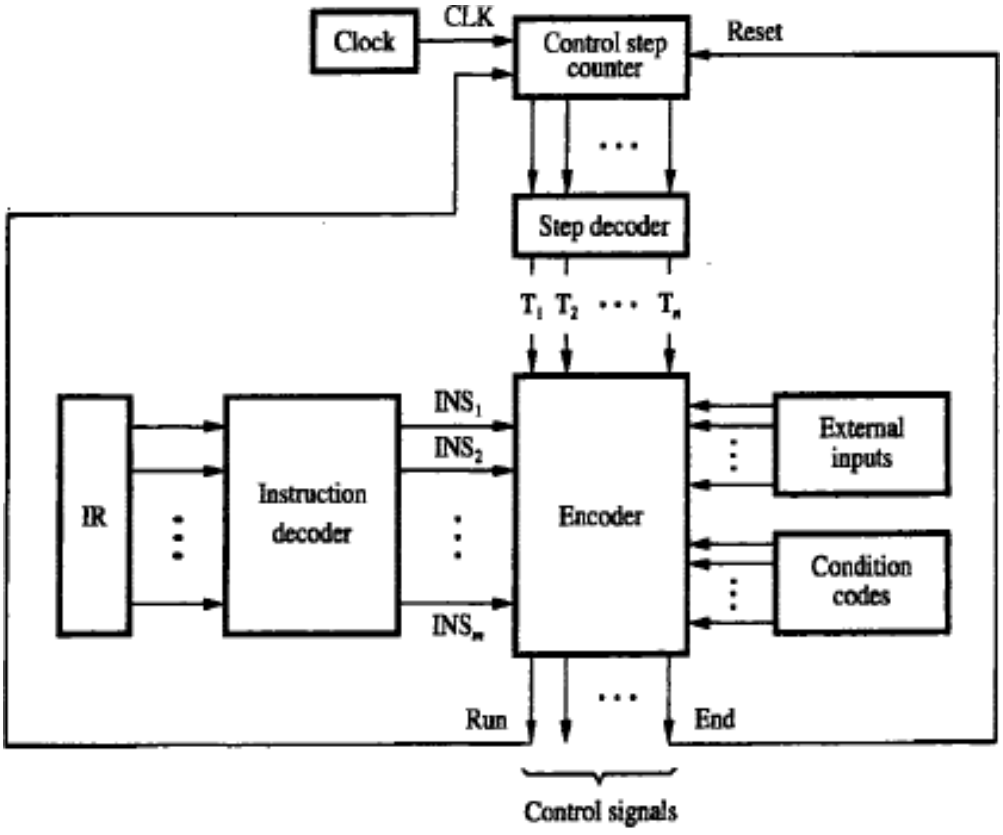


Figure 7.11 Separation of the decoding and encoding functions.

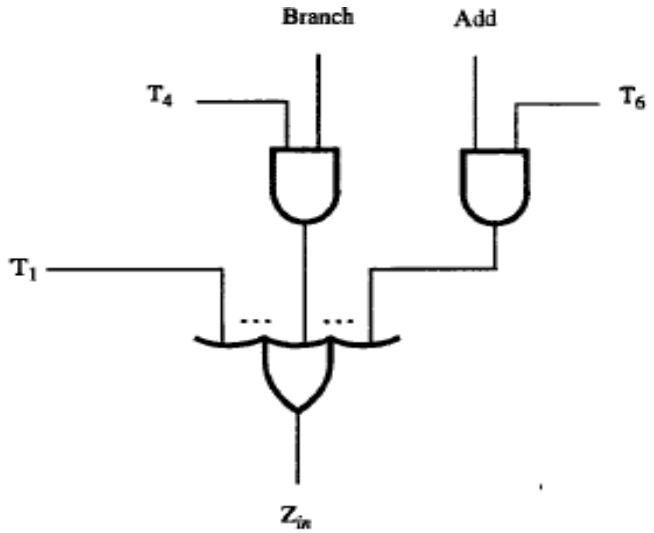
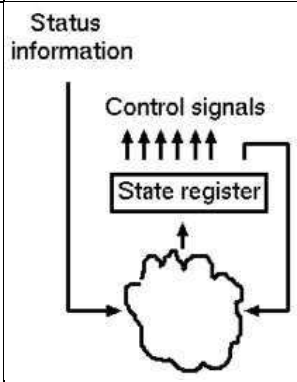
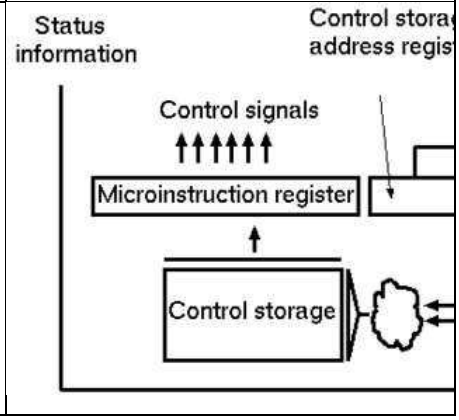


Figure 7.12 Generation of the Z_{in} control signal

HARDWIRED CONTROL VS MICROPROGRAMMED CONTROL

Attribute	**Hardwired Control**	**Microprogrammed Control**
Definition	Hardwired control is a control mechanism to generate control-signals by using gates, flip- flops, decoders, and other digital circuits.	Micro programmed control is a control mechanism to generate control-signals by using a memory called control store (CS), which contains the control-signals.
Speed	Fast	Slow
Control functions	Implemented in hardware.	Implemented in software.
Flexibility	Not flexible to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to handle large or complex instruction sets	Difficult.	Easier.
Ability to support Operating systems& diagnostic features	Very difficult.	Easy.
Design process	Complicated.	Orderly and systematic.
Applications	Mostly RISC microprocessors.	Mainframes, some microprocessors.
Instruction set size	Usually under 100 instructions.	Usually over 100 instructions.
ROM size	-	2K to 10K by 20-400 bit microinstructions.
Chip area efficiency	Uses least area.	Uses more area.
Diagram		

MICROPROGRAMMED CONTROL

Microprogramming is a method of control unit design (Figure). Control-signals are generated by a program similar to machine language programs. Control Word(CW) is a word whose individual bits represent various control-signals (like Add, PCin). Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in CW. Individual control-words in microroutine are referred to as microinstructions (Figure).

A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the microroutine. The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the Control Store (CS). Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS. µPC is used to read CWs sequentially from CS. (µPC Microprogram Counter). Every time new instruction is loaded into IR, o/p of Starting Address Generator is loaded into µPC. Then, µPC is automatically incremented by clock; causing successive microinstructions to be read from CS. Hence, control-signals are delivered to various parts of processor in correct sequence.

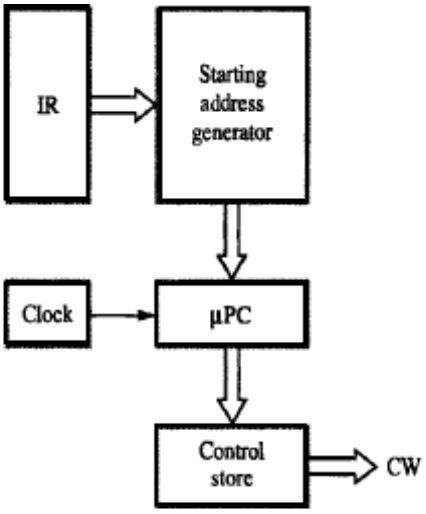


Figure 7.16 Basic organization of a microprogrammed control unit.

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	

Figure 7.15 An example of microinstructions for Figure 7.6.

Advantages

• It simplifies the design of control unit. Thus it is both, cheaper and less error prone implement.

• Control functions are implemented in software rather than hardware.

• The design process is orderly and systematic.

• More flexible, can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.

• Complex function such as floating point arithmetic can be realized efficiently.

Disadvantages

• A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.

• The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.

Organization Of Microprogrammed Control Unit To Support Conditional Branching

Drawback of previous Microprogram control

 It cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.

Solution:

 Use conditional branch microinstruction.

In case of conditional branching, microinstructions specify which of the external inputs, condition- codes should be checked as a condition for branching to take place. Starting and Branch Address Generator Block loads a new address into µPC when a microinstruction instructs it to do so (Figure). To allow implementation of a conditional branch, inputs to this block consist of external inputs and condition-codes & contents of IR.

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}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

µPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations:

1) When a new instruction is loaded into IR, µPC is loaded with starting-address of microroutine for that instruction.

2) When a Branch microinstruction is encountered and branch condition is satisfied, µPC is loaded with branch-address.

3) When an End microinstruction is encountered, µPC is loaded with address of first CW in microroutine for instruction fetch cycle.

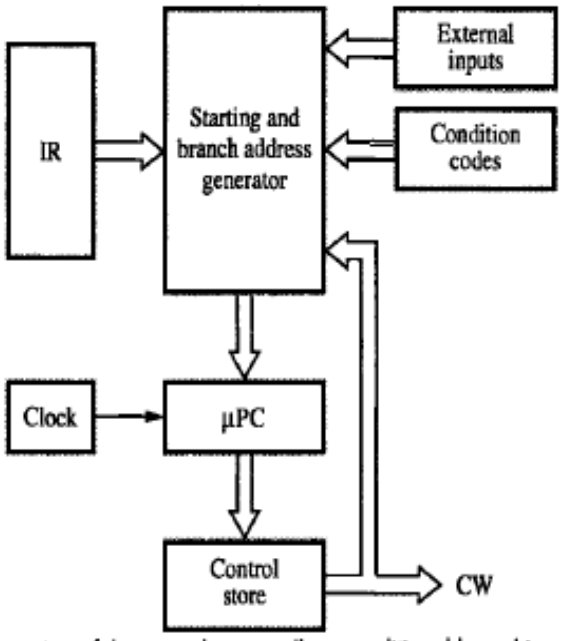


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.