3

Arithmetic Units: Data Manipulation

If one considers that a computer is "one that computes," then perhaps the principle function of the machine is to operate on data. That is, we want to manipulate information in a predetermined fashion, according to some rules and methods that make sense. The earliest computers were built to do arithmetic at a higher rate than previously attainable, at an accuracy providing the detail needed. These machines were often used in some military capacity, such as building tables for ballistics operations. In the last chapter we examined some of the methods for information representation, and the limitations of those methods. In the next chapter we will discuss the instructions that the machines utilize, that is, instructions to manipulate the information and instructions to control the computer system itself. In this chapter, we are concerned with the design of the circuitry for doing the actual data manipulations, that is, hbw does one design circuitry for performing additions, multiplications, and divisions?

Many times in the discussion of a computer system we gather all of these functions together and consider them to be performed by a single block of logic called an arithmetic/logic unit (ALU). Such a block is shown in Figure 3.1. This diagram is directly applicable to LSI ALUs, such as the '181 or '381; however it is also applicable to dedicated units such as the THCT1010 Multiplier/Accumulator. Some ALUs may require additional lines to provide a carry input or to handle status bits on output. In the figure, the source of the operands is left unknown, as is the destination of the result. The interconnection of the components is a function of the type of computer and its intended application, as we will discuss later. But now our concern is with the ALU. Logical functions are achieved by gating the appropriate function to the output. For example, the function A AND B is achieved by having each A_i ANDed with the corresponding B_i to derive F_i. The logic operations can be achieved with minimal gate delays and is therefore a relatively fast operation. The more interesting operations are those required for the arithmetic manipulations.

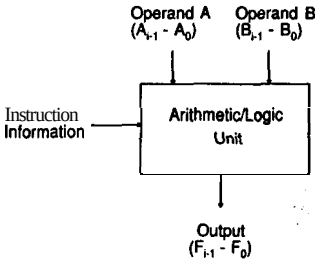


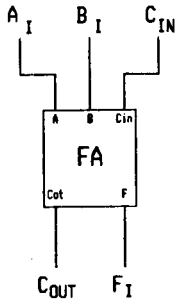
Figure 3.1. Connections for an Arithmetic/Logic Unit.

3.1. Addition: the Universal Data Operation

One of the favorite questions asked by instructors teaching basic logic design is. what is a universal logic gate? The basic premise demonstrated by this question is that a NAND gate is considered a universal logic gate because all of the basic functions — AND, OR, EX-OR, and so on — can be derived by different combinations of NAND gates. In a similar fashion. NOR gates are also universal logic gates. The same type of statement can be made concerning arithmetic operations and the add function. All of the various arithmetic operations — add, subtract, multiply, and divide — can be implemented by appropriate combinations of the add function. First we will look at the full adder, and some variations of it, then we will consider the look-ahead carry process that can be used to speed up the add function. Other applications of add functions, such as the carry save adder or the Wallace tree adder. will be treated with other functions such as multiply.

A basic cell that can be used to perform additions is the full adder (FA). shown diagrammatically in Figure 3.2(a). As shown, the function of the FA is to add two bits (A_I and B_I) and the carry fmm a stage of lower significance (C_{IN}) to produce a single bit of output (F_I) and a carry out to the next stage of higher significance (C_{OUT}). Thr truth table for this function is shown in Figure 3.2(b). Several observations can be made after examination of the truth table. For example, the function of a FA is to take three bits of equal significance — A_I, B_I, and C_{IN}— and create two bits, F_I, which has the same significance as the three input bits, and C_{OUT}, which is one bit more significant. Another observation is that the output forms a 2-bit number (C_{OUT}, F_I) which indicates how many "one" bits there are in the three input bits. The four possibilities (0, 1, 2, 3) are the permissible number of asserted bits on the inputs.

Figure 3.2 also contains Karnaugh maps for C_{OUT} and F_I, and the resulting logic equations in sum-of-products form. The sum bit (F_I) is also shown in an exclusive-OR representation. The equations are then implemented with the appropriate logic. The implementation of the sum bit is shown in the sum-of-products NAND implementation as well as the exclusive-OR implementation. In either case, the output bits are formed from two levels of logic. That is, between any input and an appropriate output there are two gates, and hence two gate delays. (One set of gates is for the AND function; the other set of gates is for the OR function.) This is true for *any* combinational circuit: if one is willing to utilize enough gates, each of which has the requisite number of inputs, it is possible



(a)

A_i	B_i	C_{in}	C_{out}	F_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b)

C_{in} \ A_iB_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

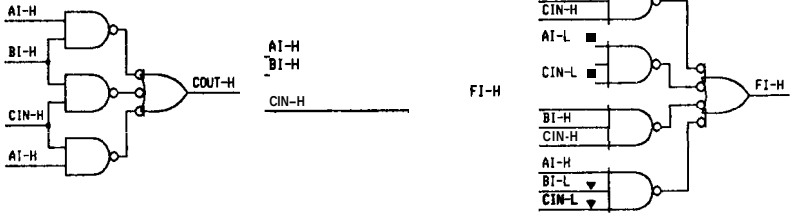
$C_{out} = A_iB_i + B_iC_{in} + A_iC_{in}$

C_{in} \ A_iB_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$F_i = \bar{A}_i\bar{B}_iC_{in} + \bar{A}_iB_i\bar{C}_{in} + A_iB_iC_{in} + A_i\bar{B}_i\bar{C}_{in}$

$F_i = A_i \oplus B_i \oplus C_{in}$

(c)



(d)

Figure 3.2. Design of a Full Adder (FA).

to accomplish any logical function in two gate delays. It may not be desirable or practicable, but it is possible. We will utilize this fact as we examine the times required to perform various functions. Thus, to perform the addition of the two bits A_I and B_I with carry, requires two gate delays from the time that the inputs are stable.

At this point it is useful to comment on the design methodology for combinational circuits, which is exemplified by the full adder. The first step in the design of any combinational system is to understand the problem at hand, which is a nontrivial requirement. Once the problem is understood, the problem and its solution can be stated succinctly in prose, identifying the input and output variables. From the problem statement, truth tables can be established, equations derived, and gating networks developed. When the solution is complete, simple tests can be performed to ascertain that the outputs do indeed perform the desired function, and that the requirements of the initial problem are satisfied.

For the full adder, the basic requirement is the addition of two numbers. As we discuss the various arithmetic operations in this chapter, we will first seek to understand the requirements of the underlying process, then proceed to determine a design which will perform the required work and meet the other needs of the system. The simplified block diagram shown in Figure 3.2(a) seems innocuous in appearance, but before progress can be made toward a reasonable design. the process which is being performed must be understood. One demonstration of the understanding required is a *correct* truth table, as shown in Figure 3.2(b). A designer's concept of what the device should do is identified by this table. If a design does not function properly, the usual debugging approach is to see if the wiring matches the logic as described by equations, and that the equations were correctly obtained from the truth table/Karnaugh maps. This approach will find errors that are implementation errors. However, it is often the case that the logic is an accurate implementation of the logic equations, and that the logic equations themselves are incorrect. This may be true not because the Boolean algebra was done incorrectly, but rather because the designer's understanding of the pmblem was flawed. And one place where that understanding will be displayed is in the truth table; thus, this step should also be examined in detail in both the design and checkout process.

In the design process, the logic equations are derived from the truth table representation of the problem. Each minterm can be written down individually from the truth table, and rules of logic utilized to find the minimal form. Or some other method can be used to find an acceptable logic equation. The Kamaugh map method is exemplified by Figure 3.2(c). From the equations. the proper arrangement of logic gates can be derived. The exact implementation techniques will be dictated by the design constraints established by the problem itself.

Portions of the process — from understanding to truth table to Kamaugh maps to logic equations to implementation — can be aided by CAE (computer aided engineering) systems or CAD systems (Computer Aided Design). However. it is imperative that a designer be able to understand the results of CAE/CAD systems, and be able to ascertain correctness of the final result. The computer aided systems will do a speedy and precise job. but the underlying algorithms used by the computer system may not coincide with the desires of the system designer. Therefore, care must be taken to assure that the final results provide a reasonable solution to the initial problem.

In general, we are not interested in computers operating on a single bit at a time. Rather, we are concerned with computers that operate on a collection of

bits. Full adders can be cascaded to the width of the system, as shown in Figure 3.3. In the figure, two 8-bit numbers are added to produce an 8-bit result. An additional input is the carry in (C_{IN}), which may come from a status register or other source; and the carry out (C_{OUT}) from the addition is available for the system.

This is not the fastest method to perform an addition, as we will see, but it will provide the correct answer. The time required to perform addition by this method, as measured from the time that all inputs are stable. is directly proportional to the number of bits in the word. This kind of addition process can be called a ripple carry adder (RCA), since the carry at each stage is propagated to the next stage. We will label the time required by this type of addition as $T_{ADD_{RCA}}$, and this time is given as:

$$T_{ADD_{RCA}} = N \times T_{FA}$$

$$= N \times (2 \times G)$$

That is. the time for an N-bit addition is just N times the time for a single bit addition (T_{FA}), and the time for a single bit of addition is two gate delays. Thus, the time for a full adder implementation of an addition module is linear in the number of bits to be added.

The details mentioned above are often hidden inside integrated circuits. However, in designing or understanding the circuitry embedded in ICs, this information may be very beneficial. Full adders can be purchased in IC form, such as the '80. Or one can consider that four such stages are cascaded in a single unit. such as the '83, a 4-bit adder. However. if one examines the circuitry internal to the '83, the carry out of the chip is generated in a different fashion than the FA method just described. This method is the look-ahead method. which we will examine later. But first let's apply the add technique described above to a subtractor.

Example 3.1: Full ***subtractor:*** Using the methods described above, design a full subtractor (FS).

The first step in this process is to understand the requirements of the design. Figure 3.4(a) is a diagram that indicates the function of the full

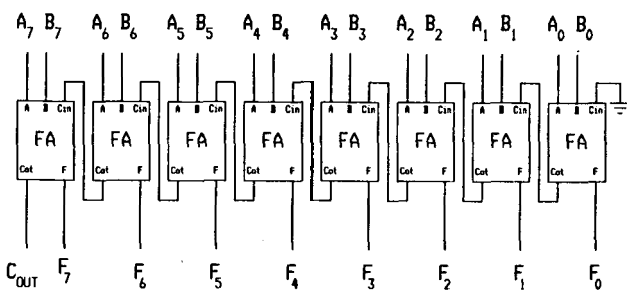
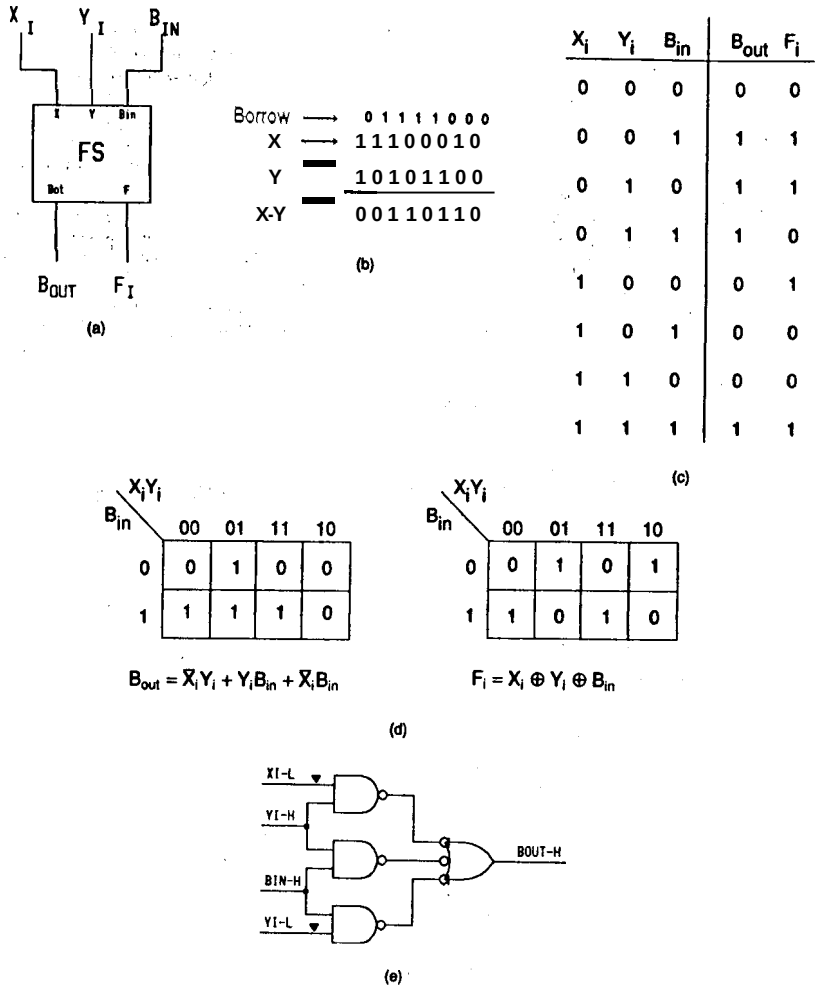


Figure 3.3. A Word Adder Composed of Full Adders.



X_i	Y_i	B_{in}	B_{out}	F_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

B_{in} \ X_iY_i	00	01	11	10
0	0	1	0	0
1	1	1	1	0

$B_{out} = \bar{X}_i Y_i + Y_i B_{in} + \bar{X}_i B_{in}$

B_{in} \ X_iY_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$F_i = X_i \oplus Y_i \oplus B_{in}$

Figure 3.4. Design of a Full **Subtractor** (FS).

subtractor. Two bits from the data word (X_I, Y_I) are inputs to the FS, as is a line from the previous stage. This line is the **borrow** in. B_{IN}. The outputs are the subtract output for this stage. F_I, and the borrow output to the next stage, B_{OUT}. The algorithm for doing subtraction in base 2 is exactly the same as the algorithm used for the base 10 taught in grade school. One

"borrows" bits (digits) from places of higher significance as needed to be able to perform the subtraction of a bit. Figure 3.4(b) is ah example of a binary subtraction; this example is given because all of the information necessary to create the complete truth table is present. Figure 3.4(c) is the truth table for the full subtractor, which has been derived by examining the subtraction of Figure 3.4(b) and filling in the table as needed.

A good thing to do in unfamiliar circumstances (base 2 subtraction is not a daily occurrence for most people) is to convert to a familiar system and do the subtraction. The base 10 representation of this operation is 226 − 172 = 54; and since $001101\mathbf{10}_2 = \mathbf{54}_{10}$, we feel much better about the accuracy of the results. Alternative derivations of an answer provide methods for checking the results of an algorithm, and should be employed as necessary to build confidence and pmve correctness.

The Kamaugh maps for the subtractor are shown in Figure 3.4(d), as well as the resulting logic equations. Finally, the gating function for the borrow is given in Figure 3.4(e). The gating is not given for the subtract output since $F_I = X_I \oplus Y_I \oplus B_{IN}$ is exactly the same formula as the sum out for a full adder. The same circuitry can be used for both functions. Note also that the logic equation for the borrow has the same form as the logic equation for the carry out of the full adder. but the inputs are different. Thus, with a little ingenuity and some gating functions, the same circuitry could be used for the $A + B$, $A - B$, $B - A$, and $A \oplus B$. The latter function is achieved by disabling the carry function; forcing the carry to a logical zero allows $A \oplus B \oplus C$ to reduce to $A \oplus B$.

The timing for a multiple bit full subtractor is exactly the same as the timing for the carry propagate adder,

$$T_{SUB} = N \times T_{FS} = N \times 2 \times T_G .$$

Subtraction of two values can be accomplished by a system of subtractors created as described here. However. a subtraction system can also be created by using an adder system (composed, for example, of '283s) and the complement-and-increment method of negating a value. The value to be subtracted is complemented with a set of inverters, and the increment is supplied by asserting the carry-in of the adder system.

The similarity between the subtraction process and the addition process is not really surprising. but it points out a situation that often arises. In many circuits, both combinational circuits, such as those discussed here, and sequential circuits, such as direct multiplication methods discussed later, there are opportunities to utilize some of the same elements of the circuit for more than one function. Here, one set of gates can be utilized for both the addition and subtraction functions. The same concept applies in some sequential circuits, where counters (or other components) can be reused for different functions. The key to the effective use of system resources is to achieve a complete understanding of the functions to be performed by the system, and to combine that with a knowledge of the logic requited to perform those functions and the capabilities of that logic. This combination will allow a designer to trade off system resources against system requirements to achieve an effective design.

Word adders composed of full adders are an example of a minimal gate solution to a pmblem, but the time required for the result may provide an

unacceptable limit to system performance. Another approach is to add more complexity to the add process to do the function faster. In order to do this, we look again at the logic equations for the addition process:

$$F_I = A \oplus B \oplus C_{IN}$$

$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$

$$= AB + C_{IN} \cdot (A + B)$$

Looking at these equations we make the following observations, some of which have been made before. The creation of the F_I signal requires but two gate delays from stabilization of input to output stable. The same can be said for the first form of the carry equation, but the second form requires three gate delay?. However, the second form allows the addition process to proceed in a different fashion. Here the data inputs (as opposed to the carry input) are grouped into two terms: AB is called the carry generate (CG) function since if this term is asserted there will be a carry (hence, the carry is "generated") regardless of the value of the carry input. The $A + B$ term is called the carry propagate, since if this term is asserted any carry which is supplied to this stage is passed on to the next. (Note that the function $A \oplus G$ would also be. a valid carry propagate function. Why?) Arrangement of the add operation to include the carry generate (CG) and carry pmpagate (CP) functions results in a module which produces:

$$F_I = A \oplus B \oplus C_{IN}$$

$$CG = AB$$

$$CP = A + B$$

Figure 3.5(a) shows a diagram of such an adder. Note that the time required to create the carry generate and carry pmpagate is a single time delay. But more importantly, note that the carry generate and carry propagate lines are *not* functions of the carry input. This means that if we arrange several look-ahead carry adder (LACA) modules as shown in Figure 3.5(b), then *all* of the CG and CP lines will be stable one gate delay after the inputs are stable. In Figure 3.5(b) these lines are inputs to another module, called a look-ahead carry generator (LACG). The LACG has the responsibility of creating the cany for each stage; it does this by looking at the carry generate and cany propagate signals from all of the stages. If C_{IN} is asserted then C_0 will be asserted. C_1 will be asserted if the carry generate of the previous stage (CG_0) is asserted. OR if CP_0 is asserted AND C_{IN} is asserted. As the carries become more significant, the amount of logic needed to generate the carry becomes larger. But it is important to note that, if the designer of the LACG is willing to supply a sufficient number of gates, then *all* of the carries will be generated in two gate delays. Thus, the addition shown in Figure 3.5(b) requires 5 gate delays: one to generate the CG and CP for each LACA, two to generate all of the appropriate carries, and two more to propagate the effect of the carries to the outputs. This is faster than the $4 \times 2 \times G = 8$ gate delays required for the FA implementation.

It is apparent that much of the complexity has been moved to the LACG. which becomes more complex as the number of modules that it services increases. A LACG that provided the carries for all 64 bits of an adder would be

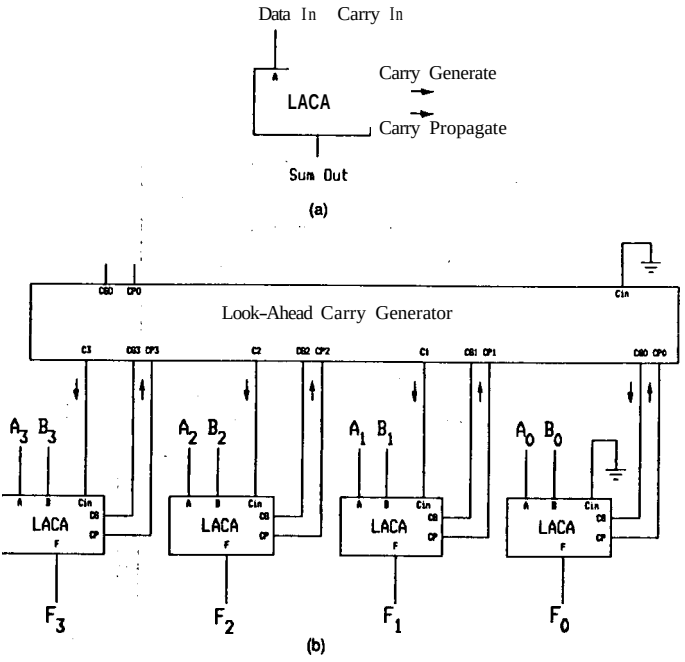


Figure 3.5. Look-Ahead Carry Adder (LACA) and its Connection in an Adder Circuit.

prohibitively expensive in terms of numbers of gates, or IC real estate. So, the LACGs are designed to cascade in exactly the same fashion as the LACAs. That is, in addition to the carries, the LACG generates a CG and CP that can be utilized by a second stage of LACG; the process continues as far as necessary to perform the work required. Such a system is shown in Figure 3.6. This figure shows the connection of '181s, which are 4-bit ALUs that generate the CG and CP signals required, and '182s, which are the LACGs. These units are both 4-bit units; that is, the ALU performs the addition of 4 bits, as well as generating the CP and CG signals for those 4 bits, and the LACG handles the CG and CP signals from 4 modules. Because of this added complexity in the ALU module, the CG and CP signals will require a minimum of two gate delays to create, as opposed to the single gate delay for a single bit unit. The time required for a carry lookahead addition is then given by:

$$T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$$

where there are N bits to be added, and the number of bits handled by the ALUs and LACGs is *b*. When no LACG is needed (up to b bits), then the time required

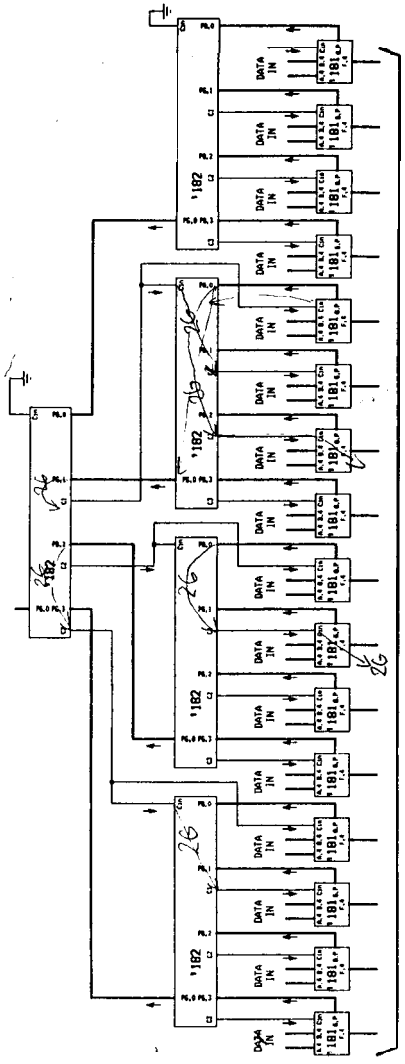


Figure 3.6. 64-Bit Addition with Carry Look-Ahead.

is simply two gate delays. Then. as the number of bits increases, the LACGs are added in a tree type of structure, where the fanout of each node of the tree is b. This gives rise to the second term in the above equation. As the number of bits (n) increases, each time the term $\log_b(N) - 1$ crosses a b boundary. that is, when the number of bits to be added crosses an exponential integer (b^i), then the depth of the tree increases by one, and the number of gate delays required increases by four. By utilizing the look-ahead process, the time required for addition has been changed from function linear in the number of bits to a process that is logarithmic in the number of bits. And the base of the logarithm is the number of bits handled by the LACAs and LACG; a larger b results in a faster adder for a given number of bits. For the adder shown in Figure 3.6, the time would be:

$$T_{LACA} = 2 + 4 \times (\lceil \log_4(64) \rceil - 1)$$

$$= 2 + 4 \times (3-1)$$

$$= 10 \text{ gate delays}$$

The 10 gate delays for the look-ahead process of Figure 3.6 are a limit, which will not, in general, be fully attained by commercial parts. This is because parts such as the '181 and '182 will reduce the number of gates required for the final function by allowing three or four levels of gates, instead of the theoretically possible two. Nevertheless, the look-ahead method for addition is much faster than addition with chains of full adders. At this time we will introduce another term for the add process: carry propagate adder, abbreviated CPA. By this term we indicate that the carry will propagate all the way through the addition, but the method of carry implementation, whether ripple carry or carry look-ahead, is not specified.

The addition function provides an example of the tradeoffs available in creating a system. A carry propagate adder will perform a function with a minimal number of gates, but the time will be correspondingly long. A carry look-ahead adder will perform an addition in a minimal amount of time, but the number of gates required for the function has correspondingly increased. Each system designer must examine the resources available (time, gates, silicon real estate, etc.) and allocate those rcaources in an appropriate manner.

3.2. Status: Results of Arithmetic Operations

Often when arithmetic operations are performed, some information about the answer is as important as the answer itself. That is, many operations are performed simply to find out how things compare: is *A* larger than *B?* Is *A* equal to *B?* Is *A* negative? Many of these questions are answerable if certain information is available concerning arithmetic operations. For example, is *A* equal to B? Well. subtract *A* from *B* (or *B* from *A);* if the result is zero, then *A* is indeed equal to *B*. In general, four pieces of information are produced by these arithmetic operations, and these pieces can be used to form bits in a status register. The four bits are zero. sign, overflow, and carry. We should hasten to add that other types of information are often available in a status register, and we will deal with this type of information in Chapter 4. At this time, we are interested in the arithmetic operations and status that can result from them.

The sign bit is perhaps the easiest to generate: it is the sign of the result of whatever operation was performed by the ALU. For two's complement numbers, this is the MSB of the result; for most floating point number systems. this is also the most significant bit. In either case. the sign of the number is fed directly to the status register. Instructions that manipulate arithmetic values (ADD, SUBTRACT, COMPARE, etc.) will modify this bit; instructions that do not do arithmetic (JUMP. CALL, etc.) will not modify the bit. For a precise list of the instructions that do modify the various bits of the status register of an existing machine, the instruction set definition for that machine must be consulted. The opposite is true for a system architect in the process of creating a set of instructions. That is, based on the application area of the machine, the arithmetic operations required. and the number systems utilized, the system designer can, at the time of the definition of the system, identify which operations will have an effect on the status register.

In addition to the sign bit, t h h is also readily available from the ALU. If an arithmetic operation resulted in a carry. then this bit is asserted in the status register. Again, the instructions modifing the bit are obtained from the instruction set definition.. The hardware of the system, then, must prevent instructions that cannot modify the bit (as defined by the instruction set) from actual modification capability. This is accomplished by disabling the load function of the status register bit (carry bit, in this case) within the status register.

The zero bit is also easy to visualize, conceptually. If the result of the operation is zero, then the bit should be set. Often this operation will be utilized by more instructions than strictly the arithmetic ones. For example, in some systems MOVE instructions will test the value being moved to see if it is zero. As before, the exact list of instructions that modify the zero bit will be obtained from the instruction set definition. The logic required is a test on each line to check its assertion level. For ALUs not providing this information on a separate status line, then all of the output lines must be checked. However, some ALUs provide a single line that will be asserted if any of the ALU lines are *not* zero. The advantage of this method is that these lines are constructed with open collector technology, and can be tied together without external gating. Thus, when all ALU outputs are zero, none of the lines is asserted, and the recognizable output is high, which is exactly what is needed by the status register.

The overflow bit is the condition that requires more than rudimentary logic. When should the overflow bit be set? The overflow bit indicates that the operation performed has exceeded the ability of the number system to represent information. Thus, one of the basic pieces of information needed (or assumed) is the number system being utilized. Our examples will concentrate on the two's complement number system. Other number systems may call for other conditions to identify an overflow. For example. consider an 8-bit, two's complement number system. From our previous considerations we know that this number system can represent values from –128 to +127. If we add 61_{10} to 45_{10}:

```
00111101   This is 61 in base 2.
00101101   This is 45 in base 2.
--------   Now add them together.
01101010   The result is equivalent to 106, the correct answer.
```

This operation does not exceed the ability of the number syslem to represent information. However. if we add 75_{10} to 58_{10}:

```
01001011   This is 75 in base 2.
00111010   This is 58 in base 2.
10000101   In 2's complemenl. this is -123.
```

If the pattern is considered an unsigned integer, then the answer is correct (133_{10}). But as a two's complement number, the ability of the number system to represent information has been exceeded. Two positive numbers have been added together, and the result was a negative number. The same thing will happen if two large negative numbers are added together: a positive number will be the apparent result. Again, the ability of the system to represent information has been exceeded: an overflow has occurred. When this happens in an arithmetic operation, then the overflow bit of the status register will be set. If a number system other than the two's complement number system is to be used, then a similar set of operations must be checked, identified by the number system itself.

Example 3.2: Overflow *circuit:* Design a circuit that will detect the occurrence of an overflow condition for a two's complement system.

As stated above, the overflow will occur when two positive numbers are added together and a negative number results, or when two negative numbers are added together to form a positive result. So the observation points are the sign bits: if the two input sign bits are positive (zero), and the output sign bit is negative (one), then an overflow has occurred. Likewise, if the two input sign bits are negative (one). and the output sign bit is positive (zero), then an overflow has occurred. A circuit to detect this condition is shown in Figure 3.7.

If the internal carries of the addition process are available, this circuit can be replaced by a single exclusive-OR gate. The exclusive-OR gate would detect a difference between the carry-in and the carry-out of the most significant stage; these two lines will differ when the overflow condition exists.

The arithmetic bits included in the status register are set and cleared as directed by the control logic for the system. That is, not all of the instructions will be allowed to modify the status bits, and some status bits will be modified by more instructions than other bits. This will require a system which is capable of selectively controlling each of the bits. If we limit ourselves to fairly standard TTL parts, then such a circuit is shown in Figure 3.8. Note that each of the bits is individually setable and clearable, as well as being reset jointly by a system reset. If the instruction set does not require the ability to individually set and clear each of the bits, then the amount of logic required for this function will be reduced.

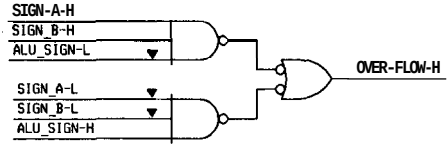


Figure 3.7. Circuit for Overflow Detection (Two's Complement System).

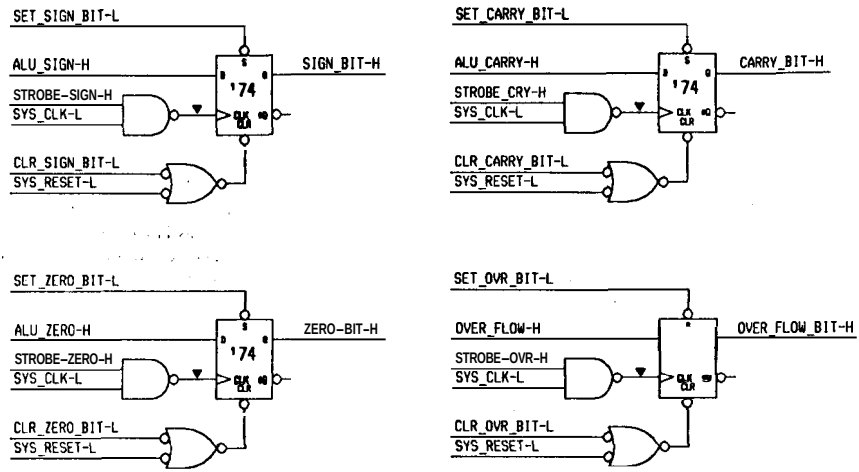


Figure 3.8. Arithmetic Bit Formation for a Status Register.

Some manufacturers provide many of these functions in a single IC, such as the AM2904. This reduces the number of chips required but not the control lines: The AM2904 has 17 control lines associated with this function.

These status bits form both a source and a destination of information in the performance of computer functions. Arithmetic operations often require a carry input, which is provided from the status register. Addition operations may change all four of the bits discussed above. Logical operations can also affect the zero bit. And program control operations can test status bits to control the flow of control in the system. Thus. these four bits can form a portion of a status register. which performs a central function in the overall system operation. We will include other kinds of status information in the discussion of instruction sets in Chapter 4.

3.3. Iterative Multiplication Methods

Fmm the very early days of computers one of the things needed was a multiplication capability. Many of the early machines were funded by defense needs, such as calculation of ballistics tables and other strictly computational tasks. For these tasks a multiply was required, and many early machines had a hardware multiply instruction. Later, when memory speeds improved dramatically, subroutines could be used to do the multiply and still accomplish the function faster than the previous hardware systems. This allowed a sizeable reduction in hardware for the computer. Still. hardware multiplication capabilities have been utilized more and more as the relative cost of hardware has decreased. Let us examine some of the methods for doing multiplication.

First, let us define the problem in exact terms, then select a sample problem to follow through the various methods of multiplication. What we want to find is the product, P, of two values, A and B.

$$P = A \times B$$

A and B are called the multiplicand and multiplier; let us assume that they are both 5-bit numbers: $A_4A_3A_2A_1A_0$ and $B_4B_3B_2B_1B_0$. We know from Chapter 2 that these can assume values from 0 to $2^5 - 1 = 31$. So, the largest product would be $31 \times 31 = 961$. To represent the number 961 requires $\lceil \log_2 (961) \rceil = 10$ bits; hence, we say that the product of two N-bit numbers requires $2 \times N$ bits to represent. With our assumption of a positional notation system, the product can be represented as:

$$\begin{aligned} P &= A \times B \\ &= A \times B_4 B_3 B_2 B_1 B_0 \\ &= A \times B_4 \times 2^4 + A \times B_3 \times 2^3 + \\ &\quad A \times B_2 \times 2^2 + A \times B_1 \times 2^1 + \\ &\quad A \times B_0 \times 2^0 \end{aligned}$$

In practice. we write this as follows:

				×	A_4	A_3	A_2	A_1	A_0
					B_4	B_3	B_2	B_1	B_0
$PP_0 \rightarrow$					$A_4 \cdot B_0$	$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
$PP_1 \rightarrow$				$A_4 \cdot B_1$	$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
$PP_2 \rightarrow$			$A_4 \cdot B_2$	$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		
$PP_3 \rightarrow$		$A_4 \cdot B_3$	$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$			
$PP_4 \rightarrow$	$A_4 \cdot B_4$	$A_3 \cdot B_4$	$A_2 \cdot B_4$	$A_1 \cdot B_4$	$A_0 \cdot B_4$				
$PR \rightarrow$	Sum of all rows								

The five rows labeled PP_0 to PP_4 are known as the partial product array. For this multiplication, the rows of the partial product array are composed of 5 bits, and each bit is an AND function of a bit from the A input and a bit from the B input. The product itself (PR) is the sum of the rows of the partial product array. when the rows have been aligned appropriately for bit significance. The effect of the multiplication by powers of two in the above equation is accounted for by the shifting of the rows in the partial product array. This is the same situation as that taught in grade school for base 10:

```
  1324
  2435
  6620
 3972
5296
2648
3223940
```

In the base 10 example, each mw in the partial product array is the result of the multiplication of the first number by one digit in the second number. As

explained above, in the base 2 system this product is very easy to obtain, since multiplication in base 2 is accomplished on a bit-by-bit basis. Therefore, the creation of the partial product array for a base 2 example is very simple: merely AND each bit in the multiplicand with the appropriate bit in the multiplier. Then the rows of the partial product array are summed in some fashion. Let us examine some methods for accomplishing this.

The most straightforward method for doing the multiply is the traditional "shift and add" method. One implementation of this is shown in Figure 3.9. The multiplier shown in the figure is set up to do an 8x8-bit multiply. Several

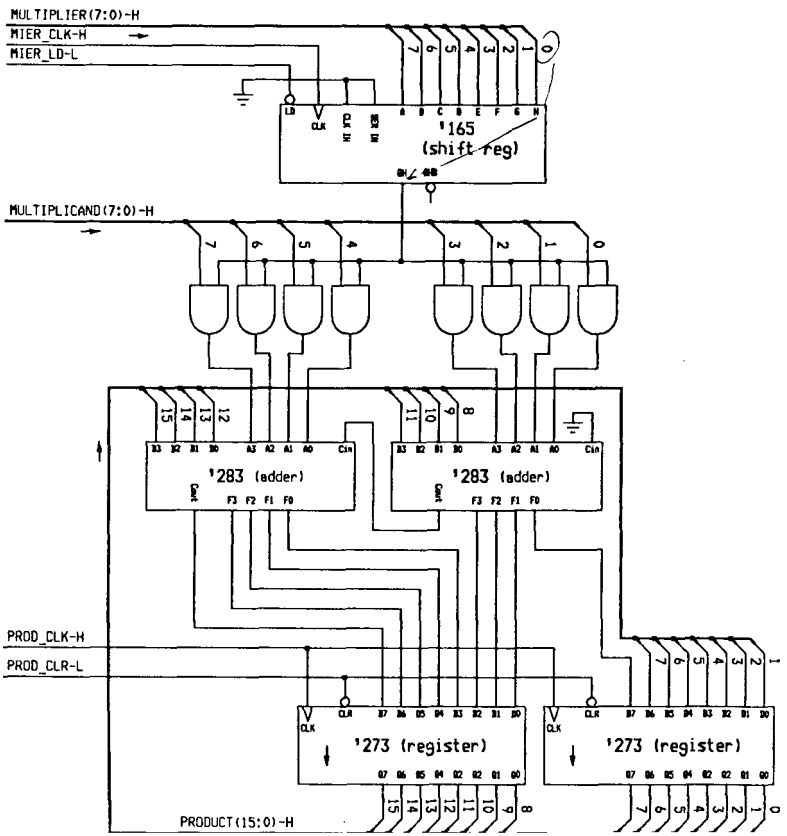


Figure 3.9. Data Path Logic Diagram for Simple Multiplier.

observations can be made concerning this system. Fist of all, the adder used is an 8-bit adder; this will function properly since the partial product addition is done from the least significant partial product to the most significant partial product. There is nothing magic about the order of partial product addition, so long as the bits are added in their appropriate significance. That is. for an N-bit multiply, the partial products PP_{N-1} to PP_0 could be added in the order shown (PP_0 first to PP_{N-1} last). in the reverse order (PP_{N-1} first to PP_0 last), or in any order deemed convenient because of design considerations.

In Figure 3.9, the shifting of the result is accomplished by hard wiring the accumulating sum to line up with the appropriate bit positions in the partial product. And the partial product is created exactly as shown in the above expansion of a binary multiplication: AND gates are used to generate the partial product from the multiplicand. The multiplier bit to be used is obtained from a shift register. A timing diagram that will assert the control signals in an appropriate fashion to do the work is shown in Figure 3.10. The timing diagram shows a set of control signals that will work in all cases; however, the resul: can be obtained faster in some circumstances if the control section is modified to look for specific conditions. One such condition is that either the multiplier or the multiplicand is zero; in such a case, the result is zero, and the answer can be given immediately. A flow diagram showing such a set of decisions is shown in Figure 3.11. The design of a control section that will create the appropriate signals is the topic of Chapter 5 and will not be covered here.

The circuit shown in Figure 3.9 is only one of a variety of implementations that will accomplish the work of multiplication. Other solutions to the problem would try to create the "best" design based on some criteria of the designer. For example, in the design shown in Figure 3.9 two chips are required for the AND function; these can be removed by using a slightly more complicated product register capable of shifting internally as well as loading from an external source. This reduces the number of chips (and hence board area required) for the function. but will necessitate a slightly more complex control. Another type of design may test for the condition that the remainder of the multiplier is all zero, hence the multiplication is essentially complete. The challenge in that type of design is to be sure that the final product bits are in the correct bit positions.

No matter what type of data path is selected, and its appropriate algorithm devised, the designer is faced with the problem of proving correctness. Several methods are available to do this, from simulation of the hardware if such a

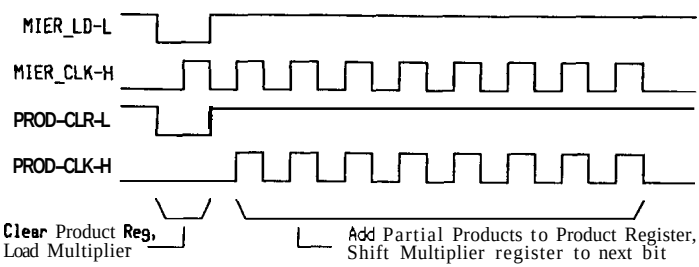


Figure 3.10. Timing of Control Signals for Simple Multiplier.

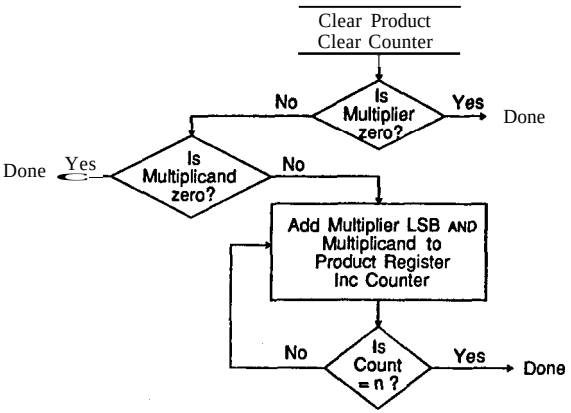


Figure 3.11. Flow Diagram for Data Dependent Multiply Algorithm.

simulation system is available, to examples worked through by hand. Before the design is fabricated, the designer should select several appropriate examples and show that the system will provide proper results.

Example 3.3: Multiplier Design: Design a data path for a multiplier that will add the partial products in "reverse" order, from the most significant to the least significant. What are some of the benefits and penalties of doing this?

This could be accomplished in a number of ways, one of which is shown in Figure 3.12. The figure shows the parts and principal interconnections needed; a more detailed schematic representation is found in Appendix B. This method requires an adder as wide as the final product. For simplicity this is shown as four '283s; faster add times could be attained by using an adder with lookahead capabilities. The product register is constructed out of '273s, which are 8-bit edge triggered registers. The bits from this register are fed back to one set of inputs on the adders. The inputs to the product registers come from the same bit positions in the adder.

The multiplier register is composed of two '195s which have been configured to be a shift register. The control section will be responsible for asserting the clock line (PLIER_CK-H) when data is available to be loaded. and also when the multiplication is proceeding. The output of the multiplier register is constantly checked to see if it is zero (PLIER_ZERO-L).

The multiplicand register is composed of two types of shift registers: '195s and a '164. The '195s provide for the load of the multiplicand value, at the same time clearing the '164 (PCAND_LD-L). Again note that the control section will be responsible for asserting the load and clock lines in the proper sequence to cause the data to be loaded at the appropriate time, and then shifted during the execution of the multiplication itself. As the multiplicand is shifted out of the '195s, it will be shifted toward lower significance in the '164. This is the method whereby the stated design

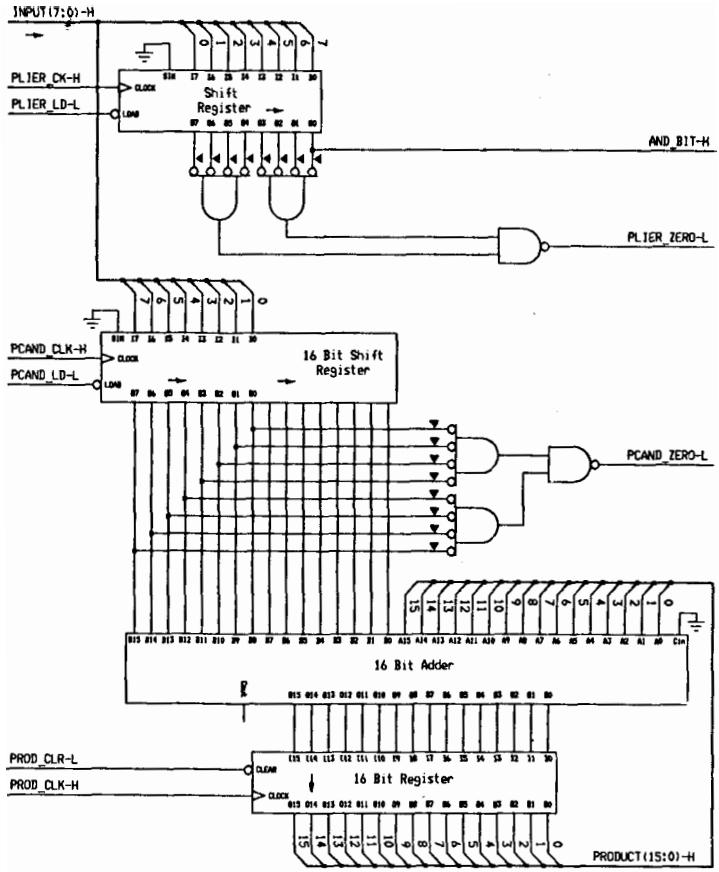


Figure 3.12. Data Path for Multiplier of Example 3.3

objective of "reverse" order of partial products will be accomplished. The multiplicand can also be checked for a zero value (PCAND_ZERO-L) when it is loaded, but this will only be effective at the beginning of the algorithm.

A flow diagram for implementation of the multiplication algorithm is shown in Figure 3.13. This diagram indicates how the algorithm proceeds. and identifies some of the benefits of this organization. The first step is to clear the product register; this is the correct answer if the multiplicand is zero, which is checked next. It is also the correct answer if the initial value

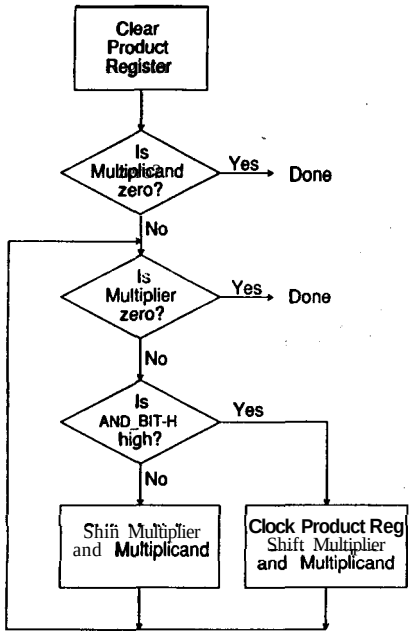


Figure 3.13. Flow Diagram for Multiplier of Example 3.3.

in the multiplier is zero, which is the next condition checked. Then the iterations begin in earnest. The value in the multiplicand register is added to the value in the product register; this result may or may not be placed into the product register. That decision is based on the most significant bit in the multiplier register (AND-BIT-H); if the bit is asserted. then the product register is loaded. In either case. the multiplier and multiplicand registers are shifted by one bit position. If the number of iterations is N (8 in this case), then we are done. If not, check the multiplier register to see if we have added in all of the appropriate values. If we have, then the algorithm is finished. As this description points out, the time required for this algorithm is data dependent. It is not necessary to check for zeros, since the algorithm would function correctly with an iteration counter and no data checks. However, by testing the values during execution of the multiply. the number of iterations will depend on the arrangement of one's and zero's in the data. By doing the additions in the "reverse" order, the product bits are in the correct position whenever all the required additions have been performed. Thus. the time to complete the instruction will vary according lo the data. which will speed up the processing.

Another benefit fmm this method is the absence of AND gates to do the individual partial product multiplications. The partial product is always

added to the accumulating product, but this value is conditionally loaded into the product register, based on the appropriate bit in the multiplier. Thus the **AND** function is supplied by control of the product load line, rather than an **AND** line on every bit.

One of the obvious tradeoffs with this method is that the speed benefits and the reduction in gating (no **AND** gates) have been obtained at the expense of a larger adder and multiplicand register. So, before a designer declares this method better/worse than another method, he needs to ascertain the various costs of the method and decide if the tradeoffs match his system resources.

The multiplication methods discussed to this point are iterative methods: the same adder system is used a number of times until the correct result is obtained. One of the questions to be addressed is the time required for the multiplication. The time that we worry about here does not include the time required to load the multiplier and multiplicand registers, and, in an actual implementation, those times need to be included in any timing estimates. The multiplication time. T_{MULT}, can be grouped into two contributing factors: the setup time. T_{SETUP}, and the iteration time. T_{ITER}.

$$T_{MULT} = T_{SETUP} + N \times T_{ITER}$$

The setup time includes the time required to clear the product register and perform any initial checks identified by the algorithm. This is reflected in the "is multiplicand zero" condition in the algorithm of Example 3.3. The iteration time is the time required to create the partial product (perform the **AND** function), to add the partial product to the running sum, and to load the resulting value into the product register. These times are reflected in the following equation:

$$T_{ITER} = T_{AND} + T_{SUM} + T_{REG}$$

The first term (T_{AND}) is the time needed by the algorithm to form the partial product. Note that in some algorithms, such as that described in Example 3.3, this time will be zero, since the same effect is obtained by conditionally loading the product register. The second term (T_{SUM}) accounts for the time required to form the sum of the partial product with the product register. This time will be determined by the adders being used and the interconnection method (carry propagate adders or carry look-ahead adders). The term should reflect the time required fmm all data inputs stable to all outputs stable. The last term T_{REG} is a combination of the times required for the register being used, which can be obtained from the data sheet for the device. These include the setup time (the time that the data must be stable prior to the assertion of the clock), the hold time (the time the data must be stable after the assertion of the clock), and propagation delay (the time for stable outputs, from the assertion of the clock). All of these times must be accounted for in deciding on the time required for the clock cycle of the unit. However, if a designer is willing to provide for nonequal clock times, then the time required by the system of Example 3.3 can be reduced. That is, if the AND_BIT·H is not asserted, then the add will not be needed, and the system can move on to the next bit (shift multiplier and multiplicand) without waiting for T_{SUM}.

These multiplication methods can be used to build multipliers out of commercially available parts, such as the system shown in Figure 3.12. Or they can be used to implement multiplications by using resources (adders, registers, and data paths) internal to a chip. such as a microprocessor. Since these methods are iterative in nature, they can be readily implemented with microcode methods. We will lwk more closely at microcode in Chapter 5, but an understanding of the iterative nature of the system helps to explain why some manufacturers identify the times required by multiplication instructions in numbers of cycles. And why some multiplication instructions indicate that the time for instruction completion is dependent on the data being used.

Before we move on to direct methods of multiplication, we will note that in the considerations thus far we have carefully avoided any mention of negative numbers. Without any modification, the techniques mentioned will not function for negative numbers. A number of techniques have been used to allow use of negative as well as positive numbers. The technique we will describe here is called Booth's algorithm, after a pair of British th but similar techniques are used elsewhere. These techniques are classified as recoding techniques. since the multiplication is modified by a recoding scheme. Let us see how this is applicable to the problem of multiplicationof signed numbers.

First of all, we need to remember from Chapter 2 that the bits in the number have a different meaning for signed numbers. That is, the most significant bit has a different meaning. The five bit numbers which were used earlier for an example had the form and meaning:

$$B = B_4\,B_3\,B_2\,B_1\,B_0$$

$$= B_4 \times 2^4 + B_3 \times 2^3 + B_2 \times 2^2 +$$

$$B_1 \times 2^1 + B_0 \times 2^0$$

$$= B_4 \times 16 + B_3 \times 8 + B_2 \times 4 +$$

$$B_1 \times 2 + B_0 \times 1$$

The difference for a two's complement number is shown in the following fashion:

$$B_{2\text{'s COMP}} = B_4 \times (-16) + B_3 \times 8 + B_2 \times 4 +$$

$$B_1 \times 2 + B_0 \times 1$$

As can be seen from the equation, the most significant bit is different in its weighting formulation and must be treated accordingly. The Booth's algorithm approach can be understood by first doing some algebra on the number. In a step-by-step fashion. we can express the two's complement number in a new form:

$$B_{2\text{'s COMP}} = (-16) \times B_4 + 8 \times B_3 + 4 \times B_2 +$$

$$2 \times B_1 + 1 \times B_0$$

$$= (-16) \times B_4 + (16-8) \times B_3 + (8-4) \times B_2 +$$

$$(4-2)\times B_1 + (2-1)\times B_0$$

$$= (-16)\times B_4 + 16\times B_3 - 8\times B_3 + 8\times B_2 - 4\times B_2 +$$

$$4\times B_1 - 2\times B_1 + 2\times B_0 - 1\times B_0$$

$$= -16\times(B_4 - B_3) - 8\times(B_3 - B_2) - 4\times(B_2 - B_1) - 2\times(B_1 - B_0) - 1\times(B_0 - 0)$$

The values in parentheses in the above equation are composed of the subtraction of two bits. and can have the values **+1, 0,** or –1. Note that the weights are what we would expect in that all are powers of two. Therefore, multiplication by the weighting factors can be. achieved by the shifting used in the first algorithm. The complexity comes in that now, instead of strictly adding, we need the ability to add, subtract, or do nothing. However. once a subtraction (addition) has been performed, the next operation will be an addition (subtraction). (This can be easily seen by examining possible bit patterns and the resulting order of operations.) This alternate nature of the operations guarantees that the size of the **adder/subtractor** will be limited to N bits. The easiest way to visualize this process is to work through an example:

Example 3.4: Signed multiplication with recoding: Utilize the Booth's algorithm **recoding** scheme to perform the multiplication: $25_{10} \times -19_{10}$.

The bit **patterns** for the two numbers are:

011001 **Let** A = 25_{10} be the multiplicand.
101101. And $B = -19_{10}$ be the multiplier.

The **recoding** algorithm works on pairs of bits a . shown below. Note that the product is sequentially formed; the steps shown below to **form** P_0 to P_4 correspond to the cumulation of the **partial** products to that point.

$-1\times(b_0 - 0)$	= –1	**Subtract** A **from 0 to** form P_0.
$-2\times(b_1 - b_0)$	= +2	**Add** $2\times A$ **to** P_0 to form P_1.
$-4\times(b_2 - b_1)$	= –4	Subtract 4 x A from P, to form P_2.
$-8\times(b_3 - b_2)$	= 0	$P_3 = P_2$.
$-16\times(b_4 - b_3)$	= +16	Add $16\times A$ **to** P_3 **to form** P_4.
$-32\times(b_5 - b_4)$	= –32	Subtract $32\times A$ **from** P_4 to form P_5.

These steps can **be** followed as identified to ascertain that the answer is 4 7 5 as expected. The multiplication by powers of two called for here is achieved by the appropriate shift of the operand A. The hardware that would perform this kind of a multiplication can be visualized as shown in the following example.

Example 3.5: Hardware for recoding multiplication: Design the ***data*** path for a multiplier that will perform multiplication according tc **Booth's** algorithm. Assume that the input values are 8 bits each.

One solution to the problem is shown in Figure 3.14. The multiplicand register and the product register are formed using **'273s,** which are 8-bit registers. The multiplier is loaded into a **'165,** which is a parallel-inherial-out shift register. Note that, when the **multiplier** is loaded, the

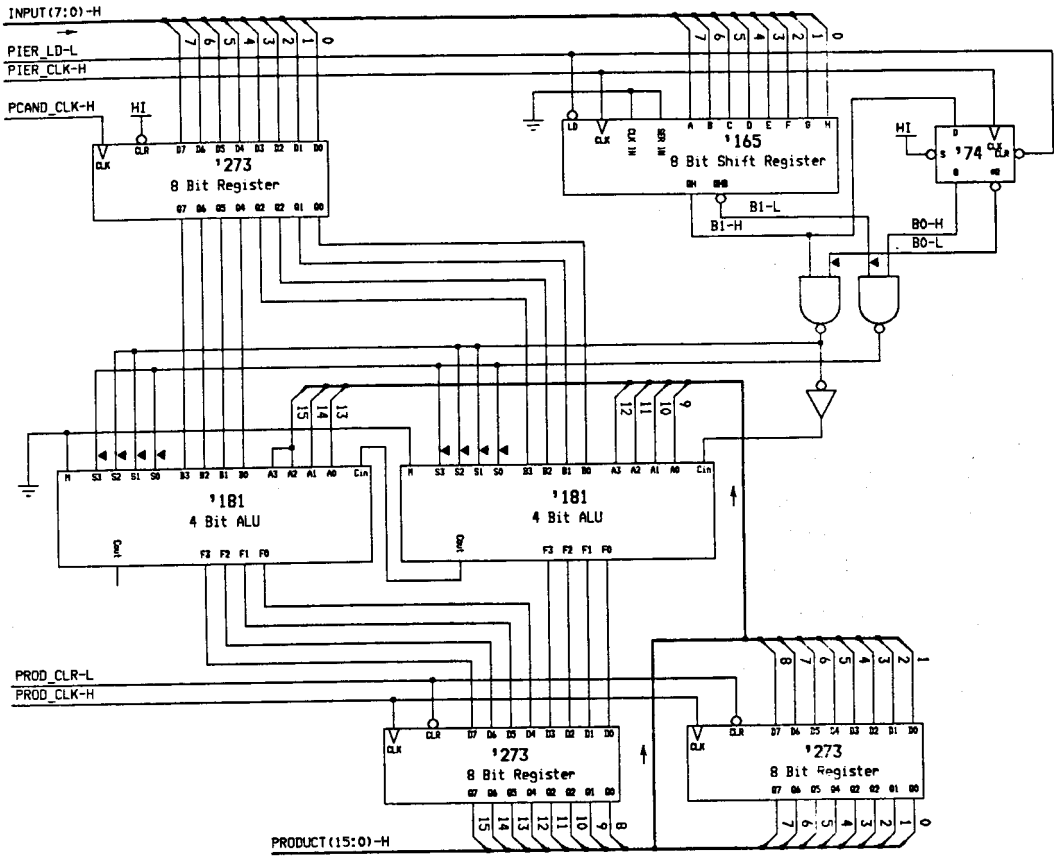


Figure 3.14. Multiplication by Booth's Algorithm.

flip-flop for storing the previous bit in sequence is cleared. The add/subtract/do–nothing requirement of the algorithm is handled by a pair of '181s, which are capable of performing all of the functions. The determination of the function of the '181s is handled by the arithmetic select lines (S3–SO), and the desired function is identified by the bits of the multiplier, as seen in the previous example. The appropriate bits are called simply B0-H and BI-H in the diagram. As the multiplier shifts through the register during the execution of the algorithm, the appropriate bits will appear on these lines. The function of the '181s should then be determined by the following table:

'Plier Bits		*ALU Sel*				*Function*
B1-H	BO-H	S3-H	SZ-H	**S1-H**	SO-H	
0	**0**	0	**0**	**0**	0	Pass product value.
0	I	I	**0**	**0**	I	Product plus multiplicand.
I	0	**0**	**1**	**1**	**0**	Product minus multiplicand.
I	1	0	**0**	0	**0**	Pass product value.

This logic is implemented in the few gates in Figure 3.14. Like the first multiplication method, this one will require a fixed number of clock pulses on the control lines to complete. Of course, it would be possible to check for a zero input condition, but it will not function properly if it is stopped in the middle of a multiply.

As can be seen from the example, the logic required for multiplication of negative as well as positive numbers does not greatly increase, but more care must be taken in the design and verification of the system. Nevertheless, the iterative approach will produce the proper result if enough caution is used in its implementation. Some of the many references for design techniques and examples of iterative methods of multiplication are listed at the end of this chapter. This is by no means intended to be an exhaustive explanation of multiplication methods, but rather it should identify some practical systems that can be used to perform the needed operations. For systems requiring more speed, then are faster methods for accomplishing the multiply, as we see in the next section.

3.4. Direct Multiplication Methods

All of the above methods require that the product be formed by combining the partial product with a value that will eventually form the final result. One of the reasons that an iterative approach is desirable from a resources standpoint is that it requires a single adder to perform the entire multiplication. The tradeoff has been made to sacrifice speed in favor of minimal logic resources. But in what way could more resources be applied to the problem? That is, given the situation where a designer is willing for purposes of speed to include a great number of gates, how should those gates be configured? We have already seen that, by examining the addition problem and using a different technique, the addition time could be changed from a linear function to a logarithmic function. Now we will analyze the multiplication function and identify methods that can be used to decrease the multiplication time.

Consider the following multiplication:

Multiplier →	01101001
Multiplicand →	01011010
PP_0 →	00000000
PP_1 →	01101001
PP_2 →	00000000
PP_3 →	01101001
PP_4 →	01101001
PP_5 →	00000000
PP_6 →	01101001
PP_7 →	00000000
Product →	0010010011101010

The multiplication process requires two separate functions: forming of the partial products and adding all of the partial products together. The formation of all the partial products $(PP_7 - PP_0)$ can be done in a single gate delay from the time that the data is stable. The hardware cost in the above example is 64 two-input AND gates, but, with that gate investment, the partial product array can be generated in parallel. Once the partial products are available, they can be summed as before. However, our objective here is speed, so rather than have a single adder and iterate to a register, let's use multiple adders and feed the result of one adder directly into another. The system resulting from this is shown in Figure 3.15(a), and it would require $N-1$ adders for N rows of partial products.

In the previous section, T_{MULT} was a function of a setup time and a multiple number of iteration times, T_{ITER}. The system shown in Figure 3.15(a) reduces the time by changing T_{ITER} to be simply the add time, T_{SUM}. The adders shown in the figure are carry look-ahead adders, but any kind could be used. The point here is that the time for a direct method with a linear connection of adders. $T_{MULT_{DIRECT-LIN}}$, is given by:

$$T_{MULT_{DIRECT-LIN}} = (N-1) \times T_{SUM}$$

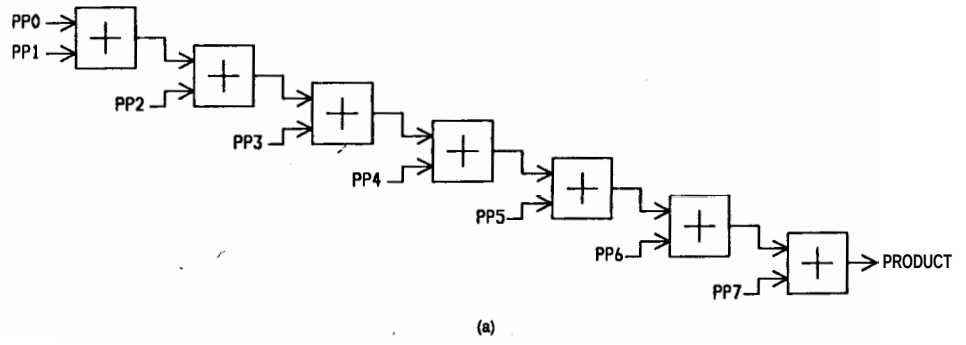
The time is linear in the number of rows (bits), which is a situation that will only get worse for more bits. The obvious solution is to get a time reduction to a logarithmic function by arranging the adders in a tree fashion, such as that shown in Figure 3.15(b). This would change the time from a linear function to a logarithmic function: $T_{MULT_{DIRECT-TREE}} = \lceil \log_2(N) \rceil \times T_{SUM}$ where there are N bits in the multiplier. This system will indeed obtain the product in a smaller time than the linear system, but other methods can achieve even higher speed.

The next method to consider has received several names, but we will call it row reduction. To understand what is going on, let us return to a simple example, a 4x4-bit multiplication for positive values only. The problem setup is exactly as we have seen' it before, with the elements of the partial product array being formed as the AND of the appropriate bits. Here we want to emphasize the rows formed in the partial product array, so we will consider the multiplication by labeling elements in the partial product array as $R_{X,Y}$, where X gives the row number and Y is the element in the row. Thus, a 4-bit multiplication becomes:

				A_3	A_2	A_1	A_0
			×	B_3	B_2	B_1	B_0
				$R_{0,3}$	$R_{0,2}$	$R_{0,1}$	$R_{0,0}$
			$R_{1,3}$	$R_{1,2}$	$R_{1,1}$	$R_{1,0}$	
		$R_{2,3}$	$R_{2,2}$	$R_{2,1}$	$R_{2,0}$		
	$R_{3,3}$	$R_{3,2}$	$R_{3,1}$	$R_{3,0}$			

Sum of **partial** products

Each row of the partial product array forms a more significant portion of the final product. as seen by the shifting nature of the information. Now let's put together a set of full adders to do this multiplication according to the above setup. That is, we will do a multiplication in the method of $T_{DIRECT\text{-}LIN}$ above, but use full adders for this simple case. This is shown in Figure 3.16. As expected, the partial product bits ($R_{X,Y}$) are added into the product by shifting them appropriately and



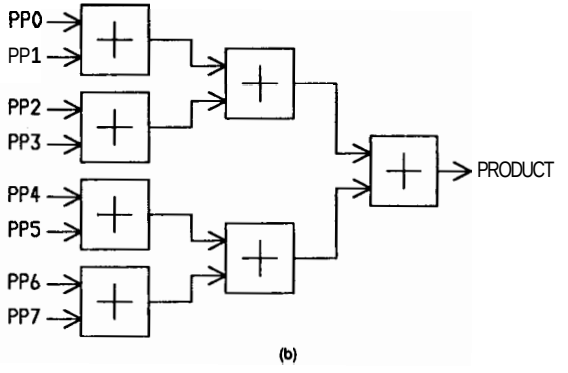


Figure 3.15. Multiplication by Direct Methods: Linear and Tree.

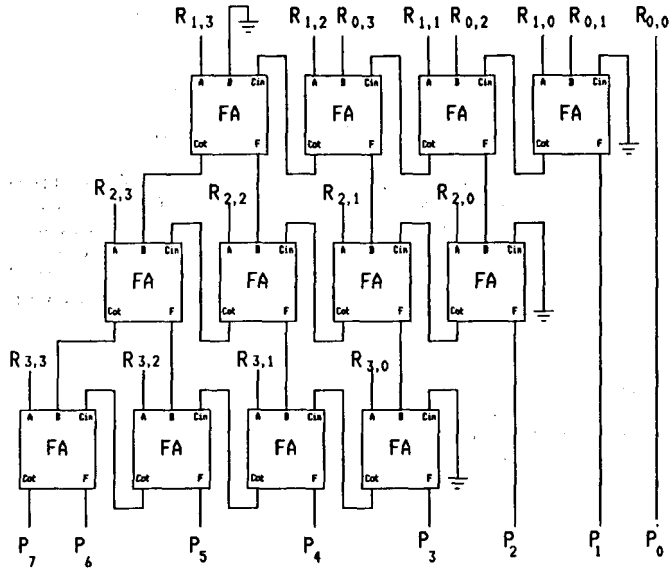
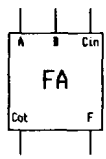


Figure 3.16. Partial Product Addition with Full Adders.

using full adders to add to the running sum. Now we ask the question, what is the function of the full adder? We often see a symbol for a full adder as shown:



We mentioned earlier that the outputs form a 2-bit number that gives the number of one's on the input lines. The three inputs (A, B, C_{IN}) all have the same significance; the sum output has the same significance, and the carry out has a significance of one higher bit position. There is no reason that the carry needs to be added into the sum in *the same row* that it is generated; that is, the carry can be saved for the next level of adders. The benefit of passing the carry to the next set of adders is that the work accomplished by the first stage no longer requires a time based on the number of bits in the word; the time is always two gate delays. The policy of saving the carry to the next stage gives rise to the name "carry save adder." or CSA. The multiplier of Figure 3.16 is redone to utilize this feature, and the result is shown in Figure 3.17.

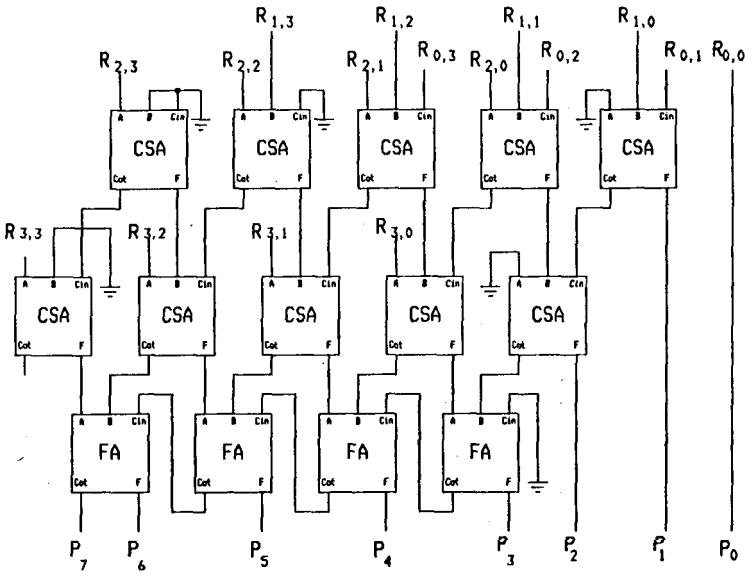


Figure 3.17. Partial Product Addition with *Carry* Save Adders.

The use of carry save adders to speed up the addition process reduces the time required for the intermediate steps to two gate delays. but the carry process cannot be put off forever. The final stage of such a system must be an adder that implements the carry process the width of the final result. The effect is that the intermediate stages can be designed with a relaxed resource criteria, and more design effort can be directed toward speeding up the final stage. One way of looking at what is happening is to recognize that using adders as shown above (saving the carry to the next level of addition) reduces the number of rows that need to be added. The carry save adder, then, is a 3-row-to-2-row reduction unit: 3 rows of bits are reduced to an equivalent operation that requires only 2 rows. For example, in the first level of CSAs in Figure 3.17. 3 mws of bits fmm the partial product array ($R_{0,x}$, $R_{1,x}$, and $R_{2,x}$) are reduced to 2 mws of bits. Then, the 2 rows of bits obtained by that process, plus the remaining row of bits from the partial product array ($R_{3,x}$) are reduced to 2 rows of bits. Finally, these 2 mws form the input to a set of full adders, which does the final addition. Thus, Figure 3.17 shows an implementation of two 3-2 (3-row-to-2-row) reduction units and a final CPA stage.

The output for any bit position of a row reduction unit contains a value that identifies the number of "ones" found in that bit position of a number of mws. Thus. a row reduction unit with k outputs will be able to represent numbers from zero to $2^k - 1$. Hence. a row reduction unit with k outputs will be able to reduce $2^k - 1$ rows; therefore, 7-3, 15-4, 31-5, and so on, are all possible configurations

however. that the complexity of the 15-4 reduction units will be much larger than the carry save adders, which form the 3-2 reduction stage. The final stage is a carry look-ahead adder that will produce the 112-bit result. ,We will assume that each row reduction unit requires only two gate delays. Thus, the time required for signals to propagate from the data inputs to the final addition stage is 9 gate delays (one for formation of partial products, two each for the four stages of row reduction units). From the equation for time required for carry look-ahead addition, the final addition process will require

$$T_{LACA} = 2 + 4 \times (\lceil \log_8(112) \rceil - 1)$$

$$= 2 + 4 \times (3-1)$$

$$= 10 \text{ gate delays}$$

So, the final result will require 19 gate delays. The cost of doing this is an enormous amount of hardware. This is not really practical in systems made of individual gates; however, this could be done in a reasonable fashion internal to an integrated circuit.

To better understand the multiplication mechanism, let us consider what is happening at each stage of the above process. The action being performed is to group portions of the partial product array together, and to then provide a number that is a count of the number of "1"s in the appropriate columns. This sectioning of the partial product array can be done in any manner that will produce the same results as the lengthy "normal" process. Thus, portions of the partial product array can be formed and summed, and then these intermediate sums combined to produce the final result. Any consistent mechanism can be used to identify portions of the multiply process for sectioning. The simplest example of this is the 3-2 reduction unit (CSA), which provides a count on the two output lines of the number of "1"s on the input lines. Other types of sectioning can be performed by using special purpose ICs, or by using similar techniques in multipliers that are internal to processor chips.

An example of the concept of subdividing the partial product array into sections can be found in the stepwise creation of the final result by considering only portions of the original problem. That is, using special purpose integrated circuits, portions of the partial product array are formed (the ANDing is done inside the chips) and the resulting elements combined in the fashion described above. The output of these chips is a number that is a sum of parts of the partial product array. Conceptually, this is shown in Figure 3.19. The figure indicates that some of the bits of the partial product array are formed, and then summed in an initial step in the multiplication process. These partial sums are then combined together to produce the final result. Using these techniques, large multipliers can be built using multipliers that work only on portions of the input values, as shown by the following example.

Example 3.7: Multiplication with sectioning: Design an 8x8 multiplier. using 74284 and 74285 4x4-bit multipliers.

These devices jointly form the 8-bit product of two 4-bit numbers: the '285 produces the four least significant bits; the '284 produces the four most

for row reduction units. One additional benefit of row reduction is the ability to do portions of the partial product addition in parallel. That is, since all of the partial products can be generated simultanwusly, the row reduction process can begin immediately to reduce the N rows of bits to 2 rows, which will then be added to form the final result. And independent row reduction units can operate on different rows of the partial product array simultaneously. This is demonstrated in the following example:

Example 3.6: Multiplication with row reduction: The DEC floating point number system has a double precision configuration with a mantissa length of 56 bits (including the hidden bit). Design a high speed multiplier to do a 56x56-bit multiply. Assume that the largest row reduction unit you have to work with is a 15-4 row reduction unit. Also assume that there is an adder at the last stage organized in 8-bit units for carry generate and carry propagate. How long will the multiplication take?

The formation of the partial product results in 56 rows of bits that need to be added together. These are then fed into row reduction units to reduce the total number of rows from 56 to 2. The overall design approach for this system, using 15-4, 7-3, and 3-2 reduction units, is shown in Figure 3.18. As can be seen from the figure. this requires two stages of 15-4 mw reduction units, one stage of 7-3 row reduction, and a stage of 3-2 row reduction. These steps can all be accomplished in 8 gate delays. Note,

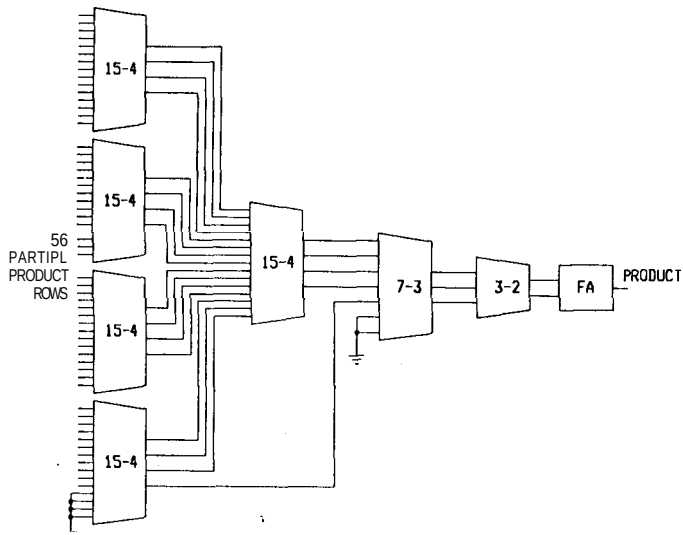


Figure 3.18. 56-Bit Multiplication Using **Row** Reduction.

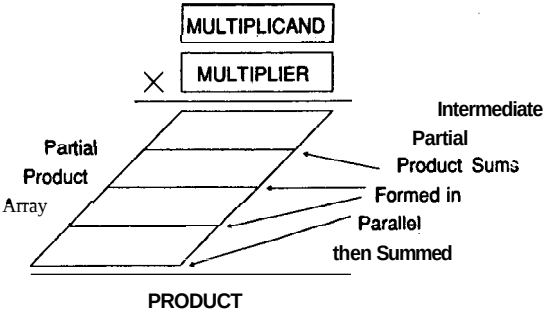


Figure 3.19. Partial Product Reduction by Sectioning.

significant bits. Therefore, an 8x8 multiply will be able to effectively use. four pairs of devices. First, let's look at what one pair will produce:

$$
\begin{array}{cccccccc}
 & & & X_3 & X_2 & X_1 & X_0 \\
 & & \times & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
 & & & R_{0,3} & R_{0,2} & R_{0,1} & R_{0,0} \\
 & & R_{1,3} & R_{1,2} & R_{1,1} & R_{1,0} & \\
 & R_{2,3} & R_{2,2} & R_{2,1} & R_{2,0} & & \\
R_{3,3} & R_{3,2} & R_{3,1} & R_{3,0} & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

So, the organization of an 8x8 multiply will be exemplified by the following configuration:

$$
\begin{array}{cccccccc}
 & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\hline
 & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & \\
\hline
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
$$

Note that the pattern established above is utilized four times, and that the various portions of the partial product overlap. So some type of an adder tree would be needed to do the summation of the partial products. This is accomplished with a carry save adder stage (3-2 reduction) and an adder. The schematic for this is shown in Figure 3.20. This works rather well, but doesn't handle large multiplications without a corresponding large number of chips. Texas Instruments is no longer making this part; however, this same method of building portions of the partial product array can be utilized using larger multipliers. For example, several manufacturers make a 16x16

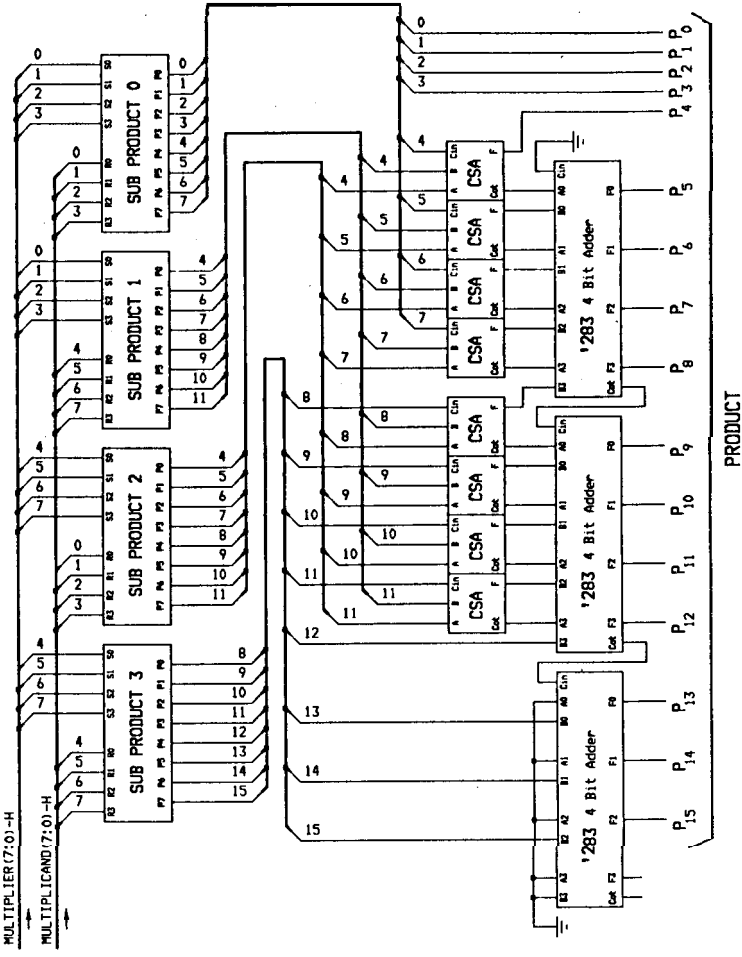


Figure 3.20. Logic diagram for a Fast Multiplier Using 74284 and 74285.

multiplier (like TDC1010), which could be cascaded in exactly this fashion to create a 32x32-bit multiply.

This example works on the basis of a "normal" partial product array; the respective portions of the array are generated internal to the multiplier chips, and partial sums formed. Then these sums can be combined to form the appropriate result. This same type of organization, forming portions of the partial product array from multiplicand and multiplier bits, can also use the recoding ideas introduced with Booth's algorithm. Indeed. the 74261 is a 2x4-bit multiplier that requires three bits of the multiplier in order to do the recoding necessary. But the system handles both positive and negative numbers, and the results are correct two's complement numbers. Like the system of Example 3.7, several sections of the parts can be combined to handle larger numbers.

All of these algorithms for high speed multiplies attempt to form the appropriate portions of the partial product array in parallel, then do as much of the partial product addition in parallel as possible. This includes delaying the final stage of the addition, where the carry will need to propagate all of the way across the output, as long as possible. Therefore, much of the design emphasis can be placed on this stage, which will be the speed bottleneck.

The multiplication process, then, adds into the final result the appropriate number of copies of the multiplicand. This can be accomplished by using a single adder and a register, and iterating through the necessary calculations. This typed system consumes considerable time resources (takes a relatively long time), but few hardware resources. One advantage to this approach is that it can be easily incorporated into a microprogrammed machine. Another multiplication method is to organize the calculation to use parallel application of partial product generation hardware. and then sum the final result with mw reduction elements and high speed adders. This design consumes little time, but requires many hardware devices. The type of design selected will be dictated by the intended application. and the relative cost of system resources.

3.5. Direct Division: Basic Division Algorithm

Whereas multiplication finds the sum of multiple copies of an operand, division is concerned with finding out how many times one value can be found in another value. The numbers involved are the divisor, D_S, the dividend. D_D, the quotient, Q, and the remainder R. Mathematically, these elements are easily related to one another:

$$D_D = Q \times D_S + R$$

The division operation determines the quotient and the remainder. One of the assumed requirements on R is that it has a smaller magnitude than D_S. In the process of designing a system to do division, care must be taken to provide hardware that will do the work required by the system. That is, magnitudes should be considered, the number of bits to be provided in the operands, the bits required in the answers, and the placement of the radix point. All of this information must be considered in the design process.

One of the most straightforward methods to use in the approach to the design of the system is to mimic the operations of paper-and-pencil long division

for positive numbers. Consider, for example, the steps required for dividing 58 by 5:

	1011	Quotient, Q
Divisor. $D_S \rightarrow$	101) 111010	$\leftarrow$ dividend, D_D
	101	$Q_3 \times D_S$
	10010	R > D_S, continue
	000	$Q_2 \times D_S$, shifted
	10010	R > D_S, continue
	101	$Q_1 \times D_S$, shifted
	1000	R > D_S, continue
	101	$Q_0 \times D_S$, shifted
	11	$R < D_S$, done

This operation proceeds in the same fashion as paper-and-pencil, long division. The base 10 algorithm with which we are familiar will produce a new base 10 digit at each iteration; the base 2 equivalent exemplified here will produce a new base 2 digit (bit) at each iteration. This is accomplished by subtracting the appropriate shifted divisor from the remainder value. The result of the subtraction determines the value of the bit in the quotient. A block diagram of such a divider is shown in Figure 3.21. The division process involves repetitive shifts and arithmetic operations, so the hardware is organized to accomplish that. The operation begins by placing the divisor in the register marked D_S, the dividend in the Q register, clearing the R register. In each iteration another bit of the answer is created, and this bit is shifted into the Q register as the dividend is shifted into the R register to be used in the calculations. As seen in the above example. the most significant bits are used in the first comparisons, so the shifting is configured to do least-to-most significant shifts. And this is exactly what is needed for the final result, since the quotient is generated most significant bit first and shifted into Q one bit at a time. At the completion of the process, the remainder will be found in R, and the quotient will be in Q.

The basic algorithm for the direct divide is very simple. After the operands are in place, the division process begins by subtracting the divisor from the value in R, which is the accumulating remainder. If the subtraction would result in a positive number, that number is loaded back into the R register and conditions are set up to introduce a "1" into the Q register. Otherwise, the R register is not changed and a "0" is readied for the Q register. Then the Q and R registers are shifted left, and the process is continued. We make the observation that since we

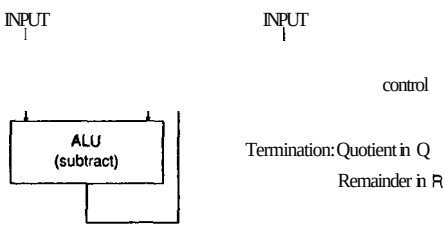


Figure 3.21. Block Diagram for Basic Division.

are working with positive numbers in this system, the subtraction will not change the bits in the lesser significant places. This observation indicates why the results of the subtraction are loaded only into the R register, and do not affect the Q register. Thus, the only information loaded into the Q register, once the process has begun, are the individual bits as they are generated and shifted in.

A flow chart for the divide operation is given in Figure 3.22. As can be seen from the flow chart, two decisions need to be made in the execution of the operation. The first concerns the action at the R register: should the value available from the subtraction be loaded into the R register or not? This decision is made based on the results of the subtraction: if the result is a positive number, then it is loaded and a "1" is setup for loading into the Q register. Otherwise, the result is not loaded, and a "0" is readied for loading. Then a count is checked to see if we are done with the operation.

The algorithm shown in Figure 3.22 conditionally loads the results of the subtraction ($R - D_D \rightarrow$ R) based on the value to be loaded. This is easily accomplished if the hardware is set up specifically to accomplish the divide. However. note that the hardware to do the direct multiply is very similar to that required for the divide. Hence, some systems are so configured that the ALUs and registers can be used for either function, and the control is slightly more complicated. In

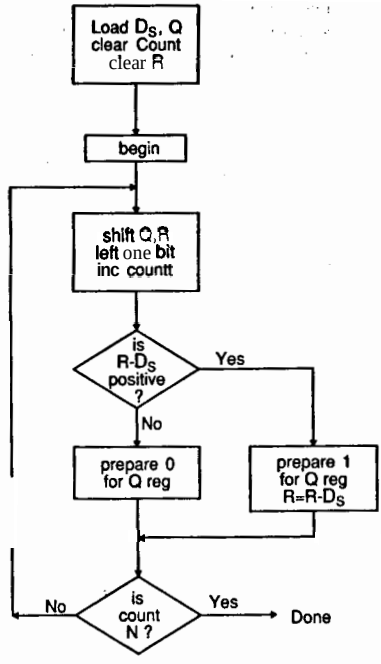


Figure 3.22. Flow Diagram for Division Operation.

such a system. it may be that the subtraction result must be stored (*ALU* out $\rightarrow$ *R*) in order to set bits to be tested by microcode, or some other control mechanism. The algorithm shown in Figure 3.22 must then be changed accordingly. The net effect is that as well as setting up to put a "0" in Q, the value which was subtracted out must be restored, requiring another addition operation. This kind of an algorithm is called a restoring algorithm; another algorithm, called the nonrestoring divide, is so configured that the value is not restored, but set up to contribute the appropriate value for the next iteration of the process. The net result is fewer overall *ALU* operations.

Example 3.8: Hardware system for direct division: Design a set of hardware that will accept data from a bus and perform a 16-bit division using the operations identified by the flow chart of Figure 3.22.

The block diagram for one solution to this problem is shown in Figure 3.23. The actual logic diagram is found in Appendix B. Here the bus provides input for three registers: D_S, Q, and R. The divisor register is made of two '564s. Since the operation needed is to subtract the divisor, this is an inverting register. Two's complement subtraction can be accomplished by inverting the divisor (hence the inverting register), incrementing the result, and then adding the other operand, which in this ease is the remainder. Here the remainder (*R*) register is made of two '198s, which can load or shift. The remainder register can be initialized to the dividend (D_D) value from the bus by using the '157s multiplexors. Finally. a pair of '198s are used for the Q register. The $R - D_S$ subtraction is accomplished by using adders; the R value comes directly from the *R* register, and the inverse of D_S provides the other input. And the increment part of the "complement and increment" two's complement negation is done by asserting the carry in of the adders. The result of this subtraction is returned to the R register through the MUXs, which allows the control to load the bus value of the subtraction value as required. That is, the bus provides the information for initialization, and from the adder comes any parallel load information required in the execution of the process. If the parallel load is required by the algorithm, then the control section causes the load. Then the '198s can be shifted simultaneously, with the control section providing the correct bit as input to the *Q* register. Missing from this diagram is the counter needed to identify the termination conditions. The control design methods required will be covered in Chapter 5.

The direct division mechanisms here can be implemented with individual adders as demonstrated by Example 3.8. Also, networks of divider cells can be constructed to produce results faster than the divide algorithms described above, since time is not required for storing and shifting operations. But the basic concepts of those division mechanisms are the same. Most high speed computers. however, do division by repeated multiplication, as shown in the next section.

3.6. High Speed Division: the Iterative Approach

We know from the definition of division that a reciprocal relationship holds for the values involved. One of the design approaches to the problem is to recognize the reciprocal relationship, and to utilize that to build a faster system. A great

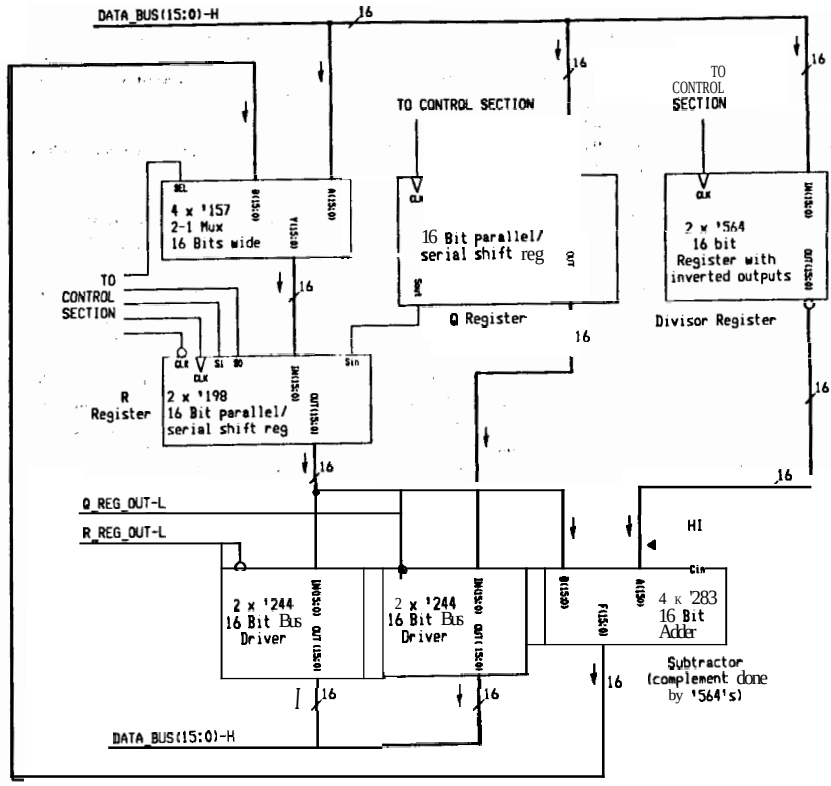


Figure 3.23. Logic for Divide Operation.

deal of effort has gone into making the multiply operation as fast as possible; is there some way that the multiplier can be utilized to do the division, so that the process benefits from the speed mechanisms available in the multiply? One way for the hardware for the multiplier to be used to do the division is to utilize the Newton-Raphson iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

We know that for a well behaved function f, and an appropriate initial value x_0, this iteration system can deliver a desired result, which is the root of $f(x) = 0$. Thus, to find the reciprocal value, we first select a well behaved function which has a root at the reciprocal. We will choose to let

$$f(x) = \frac{1}{x} - w$$

The mot of this equation will be x = 1lw. Iff (x) = **(1/x) − w,** then

$$f'(x) = -\frac{1}{x^2}$$

and the iteration system will **be**

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - w}{-\frac{1}{x_i^2}}$$

$$= x_i + (x_i - w \times x_i^2)$$

$$= 2 \times x_i - w \times x_i^2$$

$$= x_i \times (2 - w \times x_i)$$

Therefore. the operation $A \div B$ can become $A \times (1/B)$, and the system hardware can produce $1/B$ according to the above equation using only the multiplier, and a **sub**tractor for other operation required in the iteration. The Taylor series expansion of the function shows quadratic convergence, which indicates that the number of correct bits doubles every iteration. Therefore, the desired precision can **be** approached by using the proper number of iterations.

Division by the above process first finds the reciprocal **(1/B),** and then using that value to multiply by the other operand to get the final result. Some commercially available devices include all of the capabilities needed to do the iteration described above, and hence can be used to perform the iterative divide algorithm. See, for example, the AM29C325 by Advanced Micm Devices.

Another similar approach to iterative division is to form the result directly. rather than specifically calculate a reciprocal. In this approach, we assume that the numbers in question are normalized floating point numbers. This means that the dividend and divisor will **be** expressed as a fraction (at least, the mantissa is a normalized fraction). Now we want to find the quotient Q, where

$$Q = \frac{D_D}{D_S}$$

To achieve this we will multiply both the dividend and the diviscr by the same factor, f_k:

$$Q = \frac{D_D \times f_0 \times f_1 \times f_2 \times f_3 \times \cdots}{D_S \times f_0 \times f_1 \times f_2 \times f_3 \times \cdots}$$

We want the result of the various multiplications to **approach** the correct answer, Q, so we will choose **the** f_k in such a way that the denominator approaches unity. This will result in the numerator approaching the correct answer Q. Since we

know that the part of D_S that we are working with is a normalized fraction, then let us represent this fraction as:

$$D_S = 1 - x$$

where. the value of x is determined by the particular D_S. But since D_S is less than 1, x is also less than I. Now, choose

$$\begin{aligned} f_0 &= 1 + x \\ &= 1 + (1 - D_S) \\ &= 2 - D_S \end{aligned}$$

But notice that the product of D_S and f_0 is:

$$\begin{aligned} D_S \times f_0 &= (1-x)(1+x) \\ &= 1 - x^2 \end{aligned}$$

which is closer to 1 than D_S is. Each iteration both numerator and denominator are multiplied by f_k, and each iteration the result gets closer to Q. With $D_S \times f_0 = I - x^2$, let us choose f_1 so that

$$f_1 = 1 + x^2$$

With this condition, then

$$D_S \times f_0 \times f_1 = 1 - x^4$$

which is even closer to the correct answer. And so the iterations continue, each time getting the answer closer to the correct value. One of the questions to be addressed is how to find the succeeding values of f_k. We know f_1 in terms of x. but we only know x in terms of D_S and f_0:

$$\begin{aligned} f_1 &= 1 + x^2 \\ &= 1 + (1 - D_S \times f_0) \\ &= 2 - D_S \times f_0 \end{aligned}$$

Thus, each new f_k is formed by taking the two's complement of the multiplication of the f_{k-1} and the denominator result to that point. Within a computer. then, the values are presented to the divide hardware, and the iterations carried out until the answer is at the desired precision. The number of iterations required is determined by the value of f_k; when f_k is close enough to "1," the result will be close enough to the correct answer. How close is "close enough" will be determined by the application and the number of bits in the representation. However, rather than test each f_k to determine when to stop. generally a fixed number of iterations is used. Therefore, to assure that the process converges sufficiently close to the correct answer under all conditions, rather than use $2 - D_S$ to calculate f_0, a ROM is used to find an appropriate value for f_0. Providing the initial "seed" value in

this fashion guarantees that the results will be acceptable after a fixed number of iterations.

A block diagram of the hardware required to do this operation is shown in Figure 3.24. The divisor and dividend are presented to the divide hardware, and the quotient is iteratively generated. each stage getting closer to the desired value. The ROM is used to be sure that the initial precision of f_0 is close enough ω complete the process in a reasonable number of iterations.

Example 3.9: Iterative divide operations: For the divider shown in Figure 3.24, show the values of the numerator, the denominator, and the f_k at each step along the way for the following calculations: 0.4 / 0.7, 0.7 / 0.4. 0.1 / 0.15. Give the values for six iterations, rather than the three shown in the figure. Assume that the f_0 is calculated as $2-D_S$ rather than to use a ROM.

The division operation begins by calculating f_0, then multiplying this value times the D_D and D_S, as shown in Figure 3.24. For the calculation 0.4 10.7. the calculation proceeds in the following fashion:

D_{D_0}	0.4000000	D_{S_0}	0.7000000	f_0	1.3000000
D_{D_1}	0.5200000	D_{S_1}	0.9099999	f_1	1.0900000
D_{D_2}	0.5668000	D_{S_2}	0.9918999	f_2	1.0081000
D_{D_3}	0.5713911	D_{S_3}	0.9999344	f_3	1.0000656
D_{D_4}	0.5714286	D_{S_4}	0.9999999	f_4	1.0000000
D_{D_5}	0.5714286	D_{S_5}	1.0000000	f_5	1.0000000
D_{D_6}	0.5714286	D_{S_6}	1.0000000		

With an x value of 0.3, this calculation approaches the correct value within four iterations. The next requested calculation is 0.7 10.4, which is the inverse of the calculation just done:

D_{D_0}	0.7000000	D_{S_0}	0.4000000	f_0	1.5999999
D_{D_1}	1.1199999	D_{S_1}	0.6400000	f_1	1.3599999
D_{D_2}	1.5231999	D_{S_2}	0.8704000	f_2	1.1295999
D_{D_3}	1.7206066	D_{S_3}	0.9832038	f_3	1.0167962
D_{D_4}	1.7495062	D_{S_4}	0.9997178	f_4	1.0002821
D_{D_5}	1.7499998	D_{S_5}	0.9999999	f_5	1.0000001
D_{D_6}	1.7499999	D_{S_6}	1.0000000		

This calculation takes longer to approach the correct value, since the initial x was 0.6. Note that the result in this case ended up greater than one, which

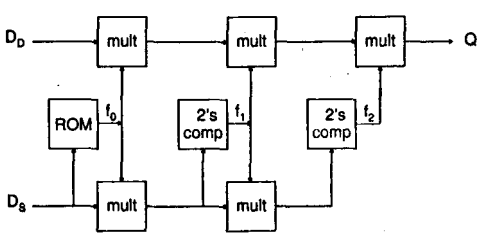


Figure 3.24. Block Diagram for Iterative Divide Operation.

is to be expected and must be handled by the hardware. That is, with normalized fractions for initial values, there is a limit that the results will not exceed, but the hardware must be able to generate numbers to that limit. The final calculation for this example is 0.1 / 0.15.

D_{D_0}	0.1000000	D_{S_0}	0.1500000	f_0	1.8499999
D_{D_1}	0.1850000	D_{S_1}	0.2775000	f_1	1.7224999
D_{D_2}	0.3186625	D_{S_2}	0.4779938	f_2	1.5220062
D_{D_3}	0.4850063	D_{S_3}	0.7275094	f_3	1.2724905
D_{D_4}	0.6171659	D_{S_4}	0.9257489	f_4	1.0742511
D_{D_5}	0.6629912	D_{S_5}	0.9944868	f_5	1.0055132
D_{D_6}	0.6666464	D_{S_6}	0.9999696		

This calculation doesn't quite get to the desired value, even with six iterations. This highlights the fact that in order to approach a desired precision within a specific number of iterations, a ROM is needed in the initial stage.

The iterative approach to the divide operation allows the hardware associated with the multiply to be used for more than one type of operation. For processors with single instruction stream capability the same hardware may be utilized for both operations. But since several steps are required for the divide operation, these instructions will generally take three to five times longer to execute than a multiply instruction.

3.7, Floating Point Arithmetic

In the previous sections we have looked at the problem of designing hardware to do the basic arithmetic operations: add/subtract, multiply, and divide. Storing information in a floating point format compounds the complexity of the problem and requires additional hardware to complete the operations. Let's first examine addition and some issues raised by addition, then look at multiplication and division. The floating point addition also includes subtraction, since the sign/magnitude method of storing information necessitates that the hardware be capable of both.

3.7.1 Floating point addition

The difficulty when adding two floating point numbers stems from the fact that the mantissas, in general, have different significance. That is, unless the exponents of the two numbers are the same, the most significant digit of one mantissa has a different magnitude associated with it than does the most significant digit of the other mantissa. Therefore, before the two numbers can be properly added together, the mantissas must be aligned. This involves determining which operand value is smaller, and then aligning the mantissa of that operand appropriately with the mantissa of the larger operand. The alignment is accomplished by shifting the mantissa of the smaller operand a number of positions to the right, hence making the digits of the smaller operand line up with the digits of the same significance in the larger operand. The amount of the alignment, the number of positions to shift, is determined by the difference in the exponents. The addition element then receives the mantissa directly from the larger operand. and the aligned mantissa from the smaller operand.

To demonstrate this process, assume that A, B, and C are floating point numbers, and find $A = B + C$. Furthermore, assume that $B <$ C. (Also, for simplicity, assume B and C are positive numbers.)

$$\begin{aligned} A &= B + C \\ &= M_B \times r_s^{E_B} + M_C \times r_s^{E_C} \\ &= (M_B \times r_s^{E_B - E_C} + M_C) \times r_s^{E_C} \end{aligned}$$

With the assumption that $B <$ C, the value of $E_B - E_C$ in the above equation is nzgative. and multiplying M_B by $r_s^{E_B - E_C}$ is nothing more than shifting the mantissa M_B to the right $E_B - E_C$ places. Note that we have said nothing about the radix of the system; this applies to base 10, base 2, or any other base. The shift for alignment is accomplished by moving the value the appropriate number of digit positions.

A block diagram for floating point addition is given in Figure 3.25. This diagram shows the arithmetic portion as an ADD/SUBTRACT unit, instead of strictly an add operation. The reason for this is that floating point numbers are almost always stored in sign-magnitude form; hence there is no sign associated with the mantissa itself. Therefore. if two numbers are to be added together, and one of the numbers has a negative sign, then what should actually be performed is a subtraction. Thus, the arithmetic unit associated with the floating point adder must be capable of doing both addition and subtraction.

The selection of the appropriate mantissa to be aligned (fmm the smaller number) is made based on a comparison of the magnitude of the two exponents. Thus, the result of this comparison directs the SELECT multiplexers to select the unaligned mantissa, and the same signal directs the ALIGN network to select the other mantissa and align it by shifting the appropriate number of positions. These two results, one unaligned mantissa and one aligned mantissa, are then fed to the

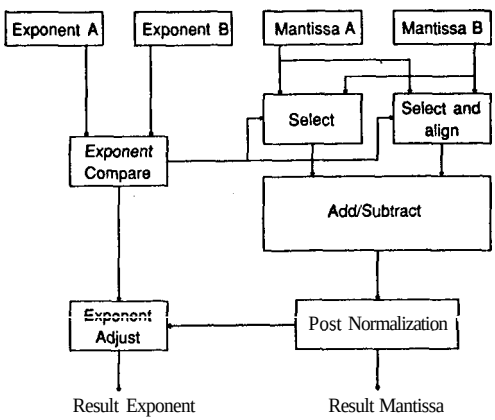


Figure 3.25. Block Diagram for Floating Point Addition.

ADD/SUBTRACT unit for the actual calculation. The resulting number is then provided to the POST NORMALIZATION unit.

The function to be provided in a post normalization step is to be sure that the final result is itself a normalized number. This unit must be capable of shifting to the right to take care of examples like the following base 10 examples (the same principles hold in any base):

```
  0.8045   Input A is normalized.
+ 0.7132   Input B is normalized.
  ------
  1.5177   Result is not normalized.
```

Thus the post normalization unit must be capable of a shift of at least one position to lesser significance. The unit must also be capable of shifts of many positions to higher significance:

```
  0.8045   Input A is normalized.
- 0.8033   Input B is normalized.
  ------
  0.0012   Result is not normalized.
```

The result of this example must be shifted left two positions to be properly normalized. Note that two *N*-digit floating point numbers, when subtracted. may result in a required post normalization alignment of *N*–1 positions. This post normalization network must then be capable of adjusting the size of the exponent to reflect any normalization. At the end of this process, the result will have been properly formed and ready for any additional operation required of it.

Floating point addition, then, requires many more operations, and hence more hardware, than its integer counterpart. The addition techniques examined earlier will apply in the arithmetic unit inside a floating point adder, but other functions are also required.

Example 3.10: Mantissa alignment for floating point add: Design the network used to align the smaller mantissa to be added to the larger mantissa in Figure 3.25. Use readily available ICs, and assume that the mantissa is 24 bits, base 2.

A mantissa of 24 bits is a fairly common size for 32-bit floating point number system. Since the number system is base 2, the alignment network must be capable of shifting any number of bits. from 0 to 24. Figure 3.26 shows that one way of accomplishing this is to use a number of 2-1 multiplexers. The figure shows the logic in a block diagram form; a logic diagram of the system is found in Appendix B. The assumption here is that the adders used to compare the exponents provide a binary number (size: 0 to 24; hence 5 bits) which indicates how far the number needs to be shifted in the alignment process. The MSB of this number is then used by the first level of MUXs to shift the number by 16 bits (the I condition), or provide no shift at all (the 0 condition). Similarly, the second MSB of the number is used by the second set of MUXs to shift the number provided by the first set of MUXs by 8 (the I condition) or provide no shift at all (the 0 condition). This process continues, with each level of multiplexers shifting the number by some power of 2, until all 5 bits have been utilized. The result is an

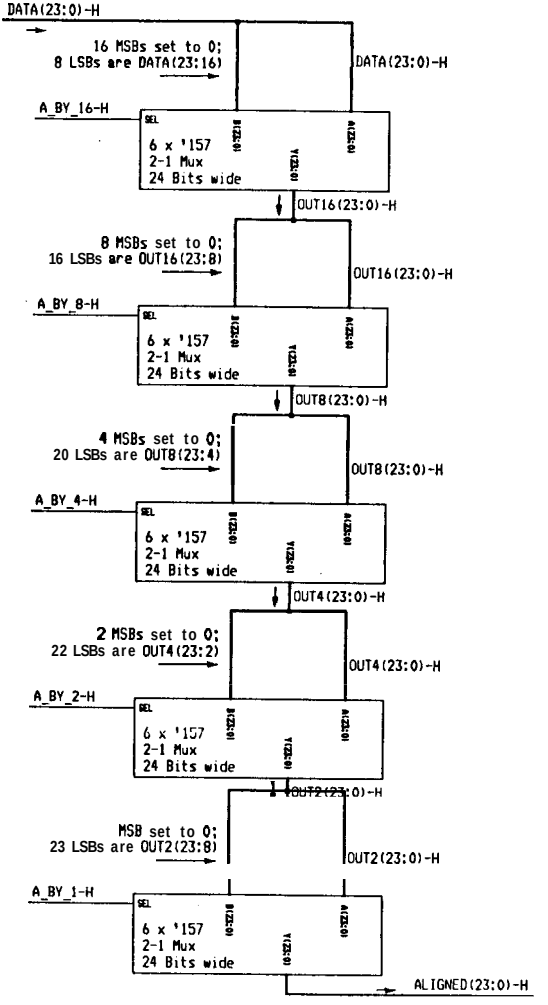


Figure 3.26. Logic for Alignment Shift Network.

output that has been shifted the number of bit positions identified by the 5-bit control number.

This network can be used to illustrate some interesting characteristics of the system. The network of Figure 3.26 has been configured to place zeros to the left of the aligned bits. This could be changed to align with sign bit (not needed here, but possible in some applications of shift networks) by asserting the unspecified inputs of the the multiplexers with the sign bit of the aligned number, rather than forcing them to zero. Another observation concerns the amount of logic needed for the alignment function. This network has been set up to do the alignment required by a base 2 number, such as the DEC or IEEE floating point system. However, if the floating point system has a different base, such as the base 16 IBM floating point system, then not all of the above levels are needed. Notice that the base 16 system does not need to align to each bit position, but rather to each digit position, which is every four bits. Thus, the last two of the five levels of logic shown in the figure would not be necessary, with a resulting in less overall logic and a speed enhancement of 40%. Thus a floating point system that does not use base 2 results in a greater range and smaller logic requirements for *some* of its constituent parts.

3.7.2 Handling the extra bits

Two problems *are* illustrated by the example of floating point addition. both of which deal with what to do with the *extra* bits. The first "extra" bit problem is identified by the following example. Assume a 6-bit mantissa for a base 2 number system, and assume that the second number has been shifted two bit positions to allow the exponents to agree. Then the mantissa addition may be something like:

```
  101010     Larger mantissa.
+ 110010     Smaller mantissa. aligned.
-------
 11011010    Addition results in 8 bits.
```

There are more bits than can be *dealt* with in the result, so something must be done with the extra bits. Several ways have been proposed and used to deal with these bits. The first and most obvious method is merely to ignore them; this is called truncation. and the unwanted bits *are* truncated from the result. This results in an error, since the final mantissa (call it M_F) differs from the real result. M_R, by whatever bits happen to be in those bit positions. This results in a truncation error, ERR_{TRUNC}, which will result in an bias, or offset. after a number of operations have been performed. For purposes of comparison with other methods of handling extra bits, let us define the error as the difference between the real result and the final mantissa:

$$ERR_{TRUNC} = M_R - M_F$$

We will also define the bias as the sum of the ERR_{TRUNC} over a span of possible results. The span we will use is all possible combinations of 2 bits, for two iterations. Thus the bias for truncation would be calculated as follows (let the decimal point mark the number of bits storable/usable by the machine):

	M_R	M_F	ERR_{TRUNC}
a	*xx*0.00	*xx*0	0.00
b	*xx*0.01	*xx*0	+0.01
c	*xx*0.10	*xx*0	+0.10
d	*xx*0.11	*xx*0	+0.11
e	*xx*1.00	*xx*1	0.00
f	*xx*1.01	*xx*1	+0.01
g	*xx*1.10	*xx*1	+0.10
h	*xx*1.11	*xx*1	+0.11

The bias is the sum of all the errors over this span. Adding all of the elements in the ERR_{TRUNC} column results in a bias of $+11.0_2$, or $+3_{10}$. Obviously. if we chose fewer elements within a span, such as only one extra bit instead of two, the bias would be less. Or if more points were selected the bias would be greater. Note that, if we included three extra bits instead of two, there would be twice as many values in the above table, all contributing to the error. But as we compare truncation with other methods we will be careful to utilize the same set of M_R so that the comparison will be valid. Truncation always throws away information, which results in a positive bias: the number stored is smaller than the actual number to be represented. Thus, over many calculations results will tend to be smaller than the true value.

Another method of handling the extra bits is to try to reduce the bias by adding half the value of the least significant bit position to the number before truncation. This method is exemplified by the following operation:

```
  101010     Larger mantissa.
+ 110010     Smaller mantissa, aligned.
  11011010   Addition results in 8 bits.
+ 00000010   Now add half of the LSB position.
  11011100   Final result. now truncate.
```

This method is called rounding, and the answers result in errors that have both positive and negative values:

	M_R	$M_R + 1/2$ LSB	M_F	ERR_{ROUND}
a	*xx*0.00	*xx*0.10	*xx*0	0.00
b	*xx*0.01	*xx*0.11	*xx*0	+0.01
c	*xx*0.10	*xx*1.00	*xx*1	−0.10
d	*xx*0.11	*xx*1.01	*xx*1	−0.01
e	*xx*1.00	*xx*1.10	*xx*1	0.00
f	*xx*1.01	*xx*1.11	*xx*1	+0.01
g	*xx*1.10	*xy*0.00	*xy*0	−0.10
h	*xx*1.11	*xy*0.01	*xy*0	−0.01

Note here that the last entries *g* and h above have had a carry propagate into the word, a fact that is indicated by the *xy* in lieu of *xx* for the value in the table. Whatever value was represented by **u** is incremented to be *xy*, and any carry which results continues to propagate into the word. The bias here is −1.0. The error in this method is always smaller than truncation, but the bias does not disappear.

One of the methods utilized to minimize the error of calculations is to create a rounding scheme that will result in a zero bias solution. These schemes have different names. such as round-to-zero or R* rounding. One such method operates according to the following rule: whenever the value to be truncated has a "1" in the most significant bit, and "0" in all other bits, that a "1" is forced into the least significant bit of M_F. This scheme results in a bias which is zero over many calculations:

	M_R	M_R + 1/2 LSB	M_F	$ERR_{ZERO\ BIAS\ RND}$
a	*xx*0.00	*xx*0.10	*xx*0	0.00
b	*xx*0.01	*xx*0.11	*xx*0	+0.01
c	*xx*0.10	*xx*1...	*xx*1	−0.10
d	*xx*0.11	*xx*1.01	*xx*1	−0.01
e	*xx*1.00	*xx*1.10	*xx*1	0.00
f	*xx*1.01	*xx*1.11	*xx*1	+0.01
g	*xx*1.10	*xx*1...	*xx*1	+0.10
h	*xx*1.11	*xx*1.11	*xy*0	−0.01

The two values in the above set that are handled differently from "normal" rounding are entries c and *g*. In both cases, a "1" is forced into the least significant bit position of the value saved. Although both entry c and entry *g* are handled in this way, only entry *g* ends up with a value different from the "normal" rounding system. The bias with this method totals zero, and over many calculations will tend to smaller errors than other techniques.

At this point, we will mention two other techniques. The first is called jamming, and was proposed by von Neumann as a good method to reduce overall errors; that is, it is better than truncation. The method is to "jam" a 1 into the least significant bit of the result. regardless of the values of the extra bits. This method results in larger errors than other methods, but over time it has the same bias as rounding. Thus, it is as fast as truncation (no time required for rounding step. since LSB is always forced to 1), but has a smaller bias.

Another method centers on the ability to look at the extra bits and the least significant bits to be retained, and using this information make an educated decision as to the value to be added. This step is carried out by using a ROM or other method of looking at several bits for the decision process. The reason for doing this is to construct the value added in the rounding step in such a way that there is no carry to propagate into the higher bit positions. This will speed the rounding step, since the method guarantees no carry beyond the least significant bits. But since the choice of the value to add in this step is made judiciously. the bias is controlled, and again over time the bias should be zero.

The errors resulting from the various methods of handling extra bits are graphically depicted in Figure 3.27. Note that the shape of the envelope of error is the same for truncation and munding, one being offset from the other. However, the rounding process has made the overall bias smaller. Note also that jamming has the same shape, but that the variations are greater. The zero bias schemes, round-to-zero and ROM rounding, have shapes that reflect their approaches to achieving their results. In both cases. the bias is minimized by intelligent handling of the extra bits involved in the action.

The second "extra bit" problem deals with the number of bits that need to be retained in the alignment process. That is, if the resulting mantissa is going to be 24 bits, must we construct adders and alignment networks capable of 48 bits or more? If the difference in the exponents is greater than 24, what should happen

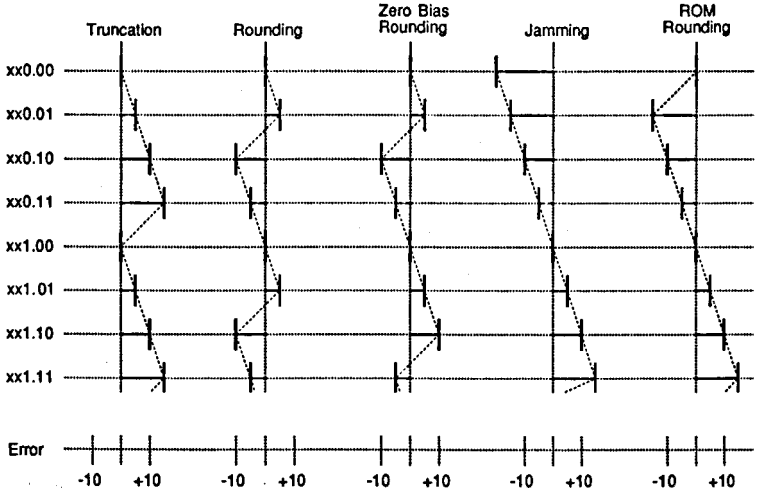


Figure 3.27. Errors in Handling Extra **Bits.**

to the aligned operand? These questions must be addressed by the designer to create a properly functioning system. **Let** us look at the problem with some examples.

As we have noted before, the alignment process takes the mantissa of smaller significance and shifts (aligns) it the proper number of places, which is **the** difference in exponents. Let the amount the alignment be represented by a, and then consider some cases. We will use mantissas which consist of 5 bits. First of all, if a = 0, then no alignment is necessary in the problem setup, but post normalization may be necessary. such **as:**

```
  0.10000
- 0.10001
- 0.00001   Post normalization
           necessary of 4 places left.
```

Now consider some examples where alignment is necessary. We will consider subtracting an aligned version of the largest mantissa representable from the smallest mantissa. The smallest mantissa for this system is just 0.10000, while the largest mantissa has a value of 0.11111. Thus for the problem $0.10000 \times 2^{1} - 0.11111 \times 2^{0}$, the value of a will be I. and the addition problem can be represented:

```
  0.10000 0
- 0.01111 1
  0.00000 1   Post normalization
           necessary of 5 places left.
```

This is perhaps the worst case for post normalization. However, note that the problem required a single bit wider than the 5 bits of the normal mantissa. The situation when o = 2 is depicted in Figure 3.28, as is the situation with other values of a. First we point out that in each of the situations depicted in Figure 3.28 there is a leading zero in the result, which will need to be removed in post normalization. The next observation concerns the bits retained by the system in the computation. These bits are underlined in the figure. Note that, for any rounding scheme (except jamming) to work properly. at least one more bit than the (end of the) underlined bits must be retained. For example, if truncation is to be used, which is the simplest of the methods mentioned above, the answer would be different if that one additional bit is not included in the calculation. Finally. we observe that the answers would all be the same if only one 1 bit were retained to the right of the vertical lines in the figure. We call this bit a "sticky" bit, and it has the characteristic that if any 1 bit were to be shifted through that position in the process of alignment, then the bit is set to a 1. This allows the results to turn out as expected.

Thus, three digits are needed beyond the number required by the number system. (This has been shown in binary, but is true in any radix.) One digit is needed for post normalization, at least one digit is needed for the rounding method, and one digit is used as the "sticky bit."

Handling the additional bits involves making reasonable decisions about the bits that result when operations generate more bits than can he retained in a result. This involves bits generated in multiplication and division, since both of these operations generate more bits than can be retained in a floating point number with the same characteristics as the input values. For example, multiplication of two 24-bit mantissas will result in a 48-bit value, which must then be reduced lo 24 bits by an appropriate algorithm. Additional bits to be concerned about in the design process include the bits in the alignment process for floating point addition. In each case, the system architect and designer need to identify the goals of the system, and based on those goals make appropriate decisions on the number of bits to retain and the rounding algorithm to produce a desired result.

With the adoption of the IEEE floating point number system, many of these decisions have been dealt with by the specification. That is, different types of rounding schemes are available, and the user has the option of specifying the mechanism that will be most appropriate for the calculations to be done.

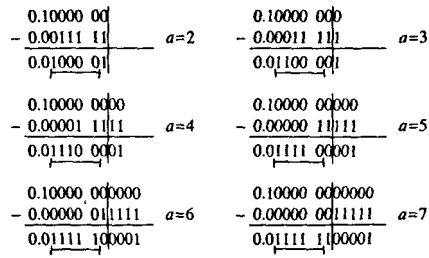


Figure 3.28. Subtraction with Alignment of Operands. (Alignment is done by a shift of a bits.)

3.7.3 Floating Point Multiplication

Floating point multiplication is perhaps the simplest floating point operation in terms of the required operations. That is, there is no alignment of operands required before initiating the operation, and minimal normalization is required at the end of the transaction. The required operations are simply stated:

$$A = B \times C$$

$$= M_B \times r_s^{E_B} \times M_C \times r_s^{E_C}$$

$$= (M_B \times M_C) \times r_s^{E_B + E_C}$$

That is, the mantissa of the result is the product of the mantissas of the two input operands, and the exponent of the result is the sum of the exponents of the input operands. A block diagram of this operation is shown in Figure 3.29. The basic operations shown in the block diagram are identical to those indicated in the above equations: the operands are separated into their constituent parts, the exponents are added, and the mantissas are multiolied The only difficulties are implementation specific. once the floating point representation has been selected. For example, the IEEE 32-bit floating point system calls for representing the exponent in an excess 127 code; therefore, the exponent adder must be so designed to correctly present the result in excess 127 code. The other block in Figure 3.29 that is not obvious from the above equations is the post normalization block. This block has the responsibility of checking the output of the multiplier to ascertain if the result is a normalized number. If it is not, then it must be adjusted accordingly, and the exponent modified. To identify the number of digit positions that can be involved in this process, let's look at the two extremes: the

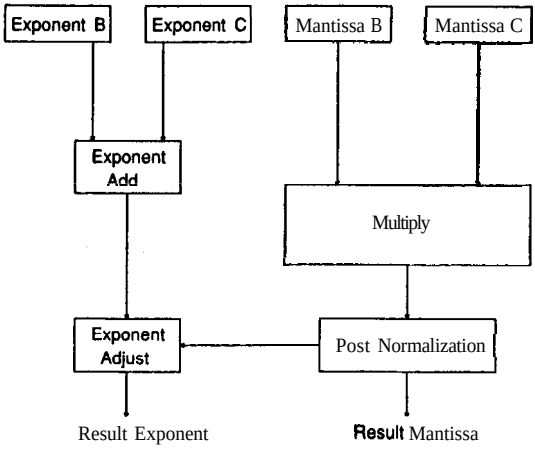


Figure 3.29. Block Diagram for a Floating Point Multiply

product of the largest legal mantissas, and the product of the smallest legal mantissas.

Largest x largest

Base 2	*Base 10*	
0.1111	0.9999	
x 0.1111	x 0.9999	
0.1110	0.9998	Aligned properly. no postnormalization.

Smallest x smallest

Base 2	*Base 10*	
0.1000	0.1000	
x 0.1000	xO.lOOO	
0.0100	0.0100	Not aligned properly. postnormalization of one digit position.

For many of the multiplications performed, no alignment will be needed in the post normalization stage. The worst case will be a post normalization of one digit position. If this occurs, then the exponent must be decremented by one before the operation is complete. It is interesting to note that the base 10 and base 2 problems are exactly the same for the smallest case (this would be true of any radix). but that the number of bits required to represent these values is not the same.

The above calculations also point out the fact that the final mantissa is composed of only portions of the result out of the multiplier. For example, the complete bit pattern resulting from the largest base two multiplication above is 11100001. But since the result is handled in the same number of bits as the original operands, the same questions arise as those discussed in connection with floating point addition: should the result be rounded? Truncated? Or what? Also, need all of the partial product array be created in the process of generating the result, or only portions of it? These questions must be addressed by the system designer in the creation of an appropriate multiplication unit.

3.7.4 Floating point division

The division operation in floating point contains almost the same steps as the multiplication operation:

$$A = B / C$$

$$= (M_B \times r_s^{E_B}) / (M_C \times r_s^{E_C})$$

$$= (M_B / M_C) \times r_s^{E_B - E_C}$$

A block diagram of the hardware required to accomplish this would look very similar to the multiplication system of Figure 3.29. The only differences are that the exponent addition would actually be subtraction, and that the multiplication block would be replaced by a divider. This division could be handled by either direct or iterative methods. The result of the mantissa division may then require post normalization in the opposite direction of the multiplier:

Largest / smallest

Base 2	*Base 10*	
0.1111	0.9999	
+ 0.1000	+ 0.1000	
1.1110	9.9990	Not aligned properly, postnormalization of digit position.

Smallest / largest

Base 2	*Base 10*	
0.1000	0.1000	
+ 0.1111	+ 0.9999	
0.1000	0.1000	Aligned properly. no postnormalization.

Again the questions of rounding' methods and number of places to calculate are raised, and the system decisions made will reflect the resource constraints placed on the system.

3.7.5 Floating point status

We discussed earlier the various status bits normally found in the status register of a computer. In general, these bits are controlled by the "normal" instructions in a computer, floating point instructions have their own conditions that add additional system status information. That is. the bits discussed previously do not form a sufficient set to reflect the conditions associated with floating point arithmetic. Thus, floating point systems often provide for indication of the following conditions:

- *Overflow.* This is similar to the overflow discussed earlier: the result has exceeded the ability of the system to represent information. because the result to be represented is too large. This can result from adding two numbers already at the maximum representable by the system, or, more generally. by multiplying two numbers whose exponents add to an exponent not representable in the system. Division can also cause overflow, dividing a very large number by a very small one.
- Underflow. This results when a number is too small to represent in the number system. This will occur when two very small numbers are multiplied, and the resulting exponent cannot be represented in the system. Similarly, division of a small number by a large one can cause the same condition to exist.
- Zero. Like the integer counterpart, this condition indicates that the specified operation resulted in a value of zero.
- Sign. The sign of the result can be the MSB of the word. like the integer case, or accessed by whatever method is indicated by the number system. This can then be used in the same fashion as the sign of an integer number.

Some manufacturers also provide additional information when building a floating point arithmetic unit:

- ***NAN (not a number).*** After the hardware performs the operation requested by the instruction. the result is not a legal number in the floating point number system. This could be an operand resewed by the manufacturer, or the IEEE Not-a-Number value.
- *Inexact.* This condition arises when the operation specified results in a value not infinitely precise, due to rounding. This can be used as an indication of truncation or mundoff error.
- ***Invalid.*** The IEEE floating point system utilizes specific patterns for representation of $+-$ and $-:$ The invalid flag of a system indicates that an operation has been performed which was invalid, such as $\infty \times 0$.

These status conditions can be incorporated in a register with the "normal" status bits, or they can form a separate status register accessible in a different manner. The implementation details will differ with design constraints and system definition.

3.8. Summary

Many books and articles have been written about performing arithmetic on computers, and designing hardware- lo do the actual arithmetic. What we have looked at are some of the basic concepts utilized in the design of arithmetic units. Addition is perhaps the most basic. since it is used in the other types of operations. We found that addition can be done in a time linear in the number of bits to be added (with full adders) or in a time that is logarithmic in the number of bits to be added (with carry look-ahead). Thus, the addition process can be made faster at the expense of additional gates or integrated circuit real estate.

Multiplication is a simple operation that can be done in a fashion similar to paper and pencil methods, using a single adder and a register to maintain the sum of the partial products. However, if speed is a major consideration, then other methods can be utilized to reduce the time required at the expense of additional hardware. We looked at methods using carry-save adders and mw reduction techniques. as well as methods that would reduce the number of mws actually needed in the partial product array. This latter method utilized parts that not only performed the generation of partial product bits, but combined those bits into partial results. The amount of useful parallelism will be decided by the system designer as he or she considers the relative cost of system resources.

Division is another operation that can be done with direct methods, such as paper and pencil methods or with iterative techniques. We have looked at some of each of these techniques. One feature of the iterative methods is the ability to use the multiplication hardware in performing the division. This justifies some of the additional design effort and hardware costs of a high speed multiplier.

Finally. we looked at some of the considerations introduced by combining the adders, multipliers, and dividers into systems for floating point arithmetic. The floating point systems introduced a number of issues related to the storage and manipulation of information. The manner in which a designer addresses these issues will have an impact in the complexity of the hardware constructed, and it will also have an impact in the complexity of any software required to effectively utilize the hardware.

3.9. Problems

3.1 Design a circuit that will accept as input a BCD digit and produce a 7-bit output that is the square of the input digit.

3.2 Design a circuit that accepts as input two 2-bit numbers. *A* and *6*. The output is a 3-bit number, which is the sum of the two input values. modulo 5.

3.3 Design a 2-bit adder that functions in no more than 3 gate delays. Inputs include two 2-bit numbers and a carry in. Outputs are the 2-bit sum, a carry generatc, and a carry propagate.

3.4 Design a 2-bit subtractor. Inputs are two 2-bit numbers and a borrow. Outputs include the 2-bit difference out and the borrow output.

35 Create the logic equations that demonstrate the look-ahead process for subtraction. That is, show (with logic equations) how a subtractor could be built so that it uses a "look-ahead borrow" technique.

3.6 Design a circuit that accepts as input two 2-bit numbers. *A* and *B*, and produces three outputs: *A>B*, *A=B*, and *A<B*. Assume an unsigned binary representation for the numbers.

3.7 Repeat Problem 3.6, but include *A>B*, *A=B*, and *A<B* inputs. How should these devices cascade? Show how these devices could be used to compare 8-bit numbers.

3.8 Prove that the overflow bit for a two's complement addition is the exclusive OR of the carry in and the carry out of the most significant stage of the addition.

3.9 Design a carry look-ahead generator circuit for 4 bits. Inputs include a carry in. as well as propagate and generate signals from four adders. Outputs are three carries, a propagate out, and a generate out. Compare your solution with the 74S182. How are they the same? How are they different? Why?

3.10 Design the logic necessary to create the status bits for a system that requires the following bits in the status register: zero, overflow, carry. sign. Assume that the carry bit out of the ALU is available.

3.11 Row reduction can be used to speed up the multiplication process. A 3-2 row reduction unit for a single bit position is a carry-save-adder, which has the same logic equation as a full adder. A 7-3 row reduction unit can be created from 3-2 row reduction units, or from random logic. Design a 7-3 row reduction unit using both methods and compare the result from the aspect of gate count and speed of operation.

3.12 Give a logic diagram for the data path of a multiplier that will produce the product of two 24-bit numbers. Use the standard shift-and-add algorithm (partial products added least significant to most significant). Use a shift register for the product register and no AND gates. Also, create a flow chart that specifies the action of the system. Be sure you know which lines go where and why.

3.13 Give a logic diagram for a multiplier system that uses the shift-and-add algorithm for partial products added in the reverse order (from most significant partial product to least significant). Use '283s for adders; use

'198s for the register functions needed. Identify the control signals on the individual parts that must be asserted to do the work. and the levels (or edges) that cause the action to occur. Include a flow chart for the action of the system.

3.14 Create a logic diagram for a 16x16 multiplier using Booths algorithm. Use '382s for the arithmetic element, and whatever registers and shift registers are needed. Include the logic required to control the function lines for the addition/subtraction/do nothing performed by the '382s.

3.15 Create a logic design for the data path to divide a 16-bit number by an 8-bit number to give an 8-bit result and an 8-bit remainder. Be sure you know why the connections are made as you specify in your design. Use '382s to perform the arithmetic. Give a Row chart that identifies the work to be done and the assertion levels of the signals required to do the work.

3.16 Design a 2x4-bit multiplier with a maximum delay from input to output of 3 gate delays.

3.17 Create an 8x8-bit multiplier system using 2x4-bit multipliers, carry save adders, and adder systems as needed.

3.18 Give a block diagram for a 32x32-bit multiply system using 7-3 row reduction units. 3-2 row reduction units, with the final stage being a carry propagate add system. Estimate the speed of the system in gate delays.

3.19 Design a floating point adder system for the floating point format given in Problem 2.10.

3.20 Obtain a data sheet for the Am29C325 floating point multiplier, and identify the steps which can be used to perform a divide operation.

3.21 **Create the logic diagrams needed for the data path of a 32-bit floating point multiplication system. Assume that the inputs have been loaded into two 32-bit registers, and that the output will be loaded into a third 32-bit register. Assume that the floating point format is a normalized format with the radix of the system equal to 2, the mantissa stored in fractional form using the hidden bit technique, and the 8-bit exponent stored in excess 128. The multiplier must use a shift-and-add algorithm. In addition, provide a status register with bits for the sign of the result, underflow, overflow, and result equal to zero. Identify the control points, and the levels of the control signals to do the work. Give a flow chart that identifies the proper levels for the signal assertions.

3.22 Multiply problem. Design a multiplier for a 24x24-bit multiply. You have three types of parts to work with: 3-2 mw reduction elements, 4-bit carry look-ahead adders. and 4-bit carry look-ahead generators. Construct a data path block diagram of the multiply process. starting with rows of the partial product array. Show all of the interconnections necessary at the row reduction stage, but not at the CLAA stage. Assuming two gate delays for all of the functions (that is, assume that the row reduction elements. the CLAAs, and the CLAGs all take two gate delays to do their work), how much time is required for the multiply? How many individual CSAs are needed for this function?

3.23 One method for performing the iterative divide operation is described as follows:

$$Q = \frac{D_D}{D_S}$$

can be calculated by:

$$Q = \frac{D_D \times f_0 \times f_1 \times f_2 \times f_3 \times \cdots}{D_S \times f_0 \times f_1 \times f_2 \times f_3 \times \cdots}$$

if the successive f_k's are chosen so that the denominator approaches one. The numerator iteration for this method is $D_{D_{n+1}} = D_{D_n} \times f_n$. The denominator iteration is used to calculate the f's, and is $f_{n+1} = 2 - D_S \times f_n$. Assume that a ROM is provided to choose an appropriate f_0, which is correct to 8 bits. Create a block diagram of a system that will follow the iteration system. Assume that you have one multiplier available, and one two's complement unit available. as well as the initial value ROM and whatever registers you need. With the block diagram include a description of how a divide will proceed. How many steps to get a result correct to 56 bits?

3.10. References and Readings

[AMD85] Advanced Micm Devices. *Bipolar Microprocessor Logic and Interface Data Book.* Sunnyvale, CA: Advanced Micm Devices, 1985.

[AnLe81] Anderson, T.. and P. A. Lee, *Fauh Tolerance. Principles and Practice.* Englewood Cliffs. NI: Prentice Hall International. 1981.

[Arms81] Armstrong, R. A., "Applying CAD to Gate Arrays Speeds 32 bit Minicomputer Design," *Electronics.* Vol. 54, No. 1, January 13. 1981. pp. 167–173.

[Baer84] Baer, J. L., "Computer Architecture." *Computer.* Vol. 17, No. 10, October 1984, pp. 77–87.

[Baer80] Baer. J. L., *Computer Systems Architecture.* Rockville, MD: Computer Science Press, 1980.

[Bart85] Bartee, T. C.. *Digital Computer Fundamentals, 6th edition.* New York: McGraw-Hill Book Company. 1985.

[Boot84] Booth, T. L., *Introduction to Computer Engineering: Hardware and Software Design.* New York: John Wiley & Sons, 1984.

[BoTj78] Borgerson, B. R., G. S. Tjaden, and M. L. Hanson. "Mainframe Implementation with Off-the-Shelf LSI Modules." *Computer.* Vol. 11, No. 7, July 1978, pp. 42–48.

[Bree89] Breeding. K. J., *Digital Design Fundamentals.* Englewood Cliffs. *NJ:* Prentice Hall. 1989.

[BuCa84] Burger. R. M., R. K. Calvin. W. C. Holton, et al., "The Impact of ICs on Computer Technology," *Computer,* Vol. 17, No. 10, October 1984, pp. 88–96.

[Cava84] Cavanagh, I. J. P.. *Digiral Computer Arithmetic: Design and Implementation.* New York: McGraw-Hill Book Company. 1984.

[ErLa85] Ercegovac, M. D.. and T. Lang, *Digital Systems and Hardware/Firmware Algorithms*. New York: John Wiley & Sons. 1985.

[Flet80] Fletcher, W. I., *An Engineering Approach to Digital Design*. Englewood Cliffs, NJ: Prentice Hall, 1980.

[Gosl80] Gosling, J. B., *Design of Arithmetic Units for Digital Computers*. Springer-Verlag, 1980.

[Haye88] Hayes, I. P.. *Computer Architecture and Organization*. 2nd Edition. New York: McGraw-Hill Book Company. 1988.

[JaSm85] lames, M. L., G. M. Smith. and I. C. Welford, *Applied Numerical Methodsfor Digital Computers*. New York: Harper & Row, 1985.

[Knut73] Knuth, D. E.. *The Art of Computer Programming: Volume 1. Fundamental Algorithms*. Reading. MA: Addison-Wesley, 1973.

[Knut69] Knuth, D. E., *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Reading, *MA:* Addison-Wesley, 1969.

[KuLa77] Kuck. D. J., D. H. Lawrie, and A. H. Sameh (Eds.), *High Speed Computer and Algorithm Organization*. New York: Academic Press, Inc., 1977.

[Kuck78] Kuck. D. I., *The Structure of Computers and Computations*. New York: John Wiky & Sons. 1978.

[KuMi81] Kulisch, U., and W. L. Miranker, *Computer Arithmetic in Theory and Practice*. New York: Academic Press, 1981.

[Lang82] Langdon, G. G.. Jr.. *Computer Design*. San Jose, CA: Computeach Press Inc, 1982.

[Leis83] Leiserson, C. E., *Area-Efficient VLSI Computation*. Cambridge. MA: MIT Press, 1983.

[Mano79] Mano, M. M., *Digital Logic and Computer Design*. Englewood Cliffs, NJ: Prentice Hall, 1979.

[Mano88] Mano, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs. NJ: Prentice Hall, 1988.

[McCl86] McCluskey, E. I., *Logic Design Principles, with Emphasis on Testable Semicustom Circuits*. Englewood Cliffs, NJ: Prentice Hall, 1986.

[MeCo80] Mead, C. A., and L. Conway, *Introduction to VLSI Systems*. Reading. MA: Addison-Wesley. 1980.

[Prad86] Pradham, D. K. (Ed.), *Fault Tolerant Computing: Theory and Techniques*. Englewood Cliffs, NJ: Prentice Hall, 1986.

[Prep85] Preparata, F. P.. *Introduction to Computer Engineering*. New York: Harper & Row. 1985.

[Schm74] Schmid. H., *Decimal Computation*. New York: John Wiley & Sons. 1974.

[Shiv85] Shiva, S. G.. *Computer Design and Architecture*. Boston. *MA:* Little, Brown, 1985.

[Stal87] Stallings, W., *Computer Organization ond Architecture*. New York: Macmillan Publishing Co., 1987.

[Swar80] Swartzlander, E. E., Jr. (Ed.), *Computer Arithmetic*. Dowden, Hutchinson & Ross. 1980.

[Tane84] Tanenbaum. A. S., *Structured Computer Organization.* Englewd Cliffs. NJ: Prentice Hall. 1984.

[TI85] Texas Instruments, *The TTL Data Book. Volume 2.* Dallas. TX: Texas Instruments, 1988.

[WaFl82] Waser, S.. and M. I. Flynn, *Introduction to Arithmetic for Digital Systems Designers.* New York: CBS College Pub.. 1982.

[Wilk87] Wilkinson. B., *Digital System Design.* Englewd Cliffs. NJ: Prentice Hall International, 1987.