Supplement to

Logic and Computer Design Fundamentals
4th Edition[1]

MULTIPLIERS AND DIVIDERS

Selected topics not covered in the fourth edition of *Logic and Computer Design Fundamentals* are provided here for optional coverage and for self-study. This material fits well with the desired coverage in some programs but not may not fit within others due to time constraints or local preferences. This supplement introduces unsigned integer multipliers and dividers. Basic algorithms are introduced and two fundamental concepts that support high-speed implementations are presented. One example high-speed implementation is outlined for each operation in order to give a sense of how these operations are handled using combinational circuits in modern VLSI chips. References [3] and [4] provide additional information on this topic.

Introduction

Many digital systems. including computers, require implementation of integer and/or floating point multiplication and division. Further, many of these applications require that multiplication and division for operands of significant size be executed at very high speeds. Historically, the recursive algorithms for these two operations if implemented in a straightforward manner as sequential circuits with maximum hardware reuse have very limited performance. Our goals here are to present these recursive algorithms for integer multiplication and division, introduce two fundamental techniques to improved performance, and to provide an implementation of each algorithm that executes at high speed.

MULTIPLICATION The basic multiplication algorithm recursively applies the following multiply step to each successive bit of the multiplier beginning with the its LSB.

AND the multiplier bit with the entire multiplicand, add the result to the accumulating partial product, and shift the accumulating partial product and multiplier one bit to the right.

Repeat this step until the MSB of the multiplier has been processed, and read the final product.

• EXAMPLE 1 Fixed Point Multiplication

The operation to be illustrated is fixed point unsigned binary multiplication, which is quite simple. The multiplier digits are always 1 or 0. Therefore, the partial products are equal either to the multiplicand or to zero. Multiplication is illustrated by the following example:

Multiplicand:	1011
Multiplier:	× 1010
Initialize Partial Product Z to 0	0000
Add ($Y_0 = 0$) x (X = 1011) to Z	+ 0000
and shift right one position	00000
Add ($Y_1 = 1$) x (X = 1011) to Z	+ 1011
and shift right one position	010110
Add ($Y_2 = 0$) x (X = 1011) to Z	+ 0000
and shift right one position	0010110
Add ($Y_3 = 1$) x (X = 1011) to Z	+ 1011
and shift right one position **Product:**	01101110

This example shows an initialization step followed by four recursive multiply steps, one for each bit of the multiplier. The algorithm illustrated differs from the classic hand multiplication algorithm in two ways. First of all, the left shift of the multiplicand is replaced by a right shift of the partial product which permits an adder of width n, the width of the multiplicand to be employed. Second, the single addition of partial products at the end is replaced by immediate addition of each partial product to form the accumulating partial product to avoid the need for a multioperand adder. This second change, however, is often reversed in modern implementations of the algorithm. •

DIVISION Initially, the dividend, divisor, and quotient are assumed to be n-bit quantities. The initial partial remainder is n 0s followed by the dividend. The basic division algorithm recursively applies the following divide step to find successive bits of the quotient beginning with the its MSB.

Shift the partial remainder one bit to the left. Subtract the divisor from the partial remainder. If the new partial remainder is positive, then set the new quotient bit to 1 and shift the new partial remainder and the quotient one bit to the left. If the new partial remainder is negative, set the new quotient bit to 0, and replace the new partial remainder with the original one and shift it and the quotient one bit to the left.

Repeat this divide step until the all bits of the quotient have been generated to produce the quotient and remainder.

• EXAMPLE 2 Fixed Point Division

The operation to be illustrated is fixed point unsigned binary division. The computation begins with the default initial partial remainder, n 0s followed by the n-bit dividend. Each divide step must perform subtraction of the divisor from the shifted partial remainder in order to determine the value of the quotient bit. Based on this result the quotient bit value is set and if it is 0, the new partial remainder is replace by the prior partial remainder. Note that the least significant half of the partial remainder and the quotient can share the same storage space.

	Dividend: 00110101
Divisor D:	00000101
Zero/Partial Remainder R:	0000000000110101
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000000110101 – 00000101**Quotient**↓ 0000000001101010
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000001101010 – 00000101 0000000011010100
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000011010100 – 00000101 0000000110101000
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000110101000 – 00000101 0000001101010000
Shift R and Q left. Subtract D. Since the new R is +, append 1 to Q.	0000001101010000 – 00000101 0000000110100001
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000110100001 – 00000101 0000001101000010
Shift R and Q left. Subtract D. Since the new R is +, append 1 to Q.	0000001101000010 0000000110000101
Shift R and Q left. Subtract D. Since the new R is –, replace with prior R, and append 0 to Q.	0000000110000101 – 00000101**Quotient**↓
Remainder:	0000000100001010

•

From this example, it is apparent that the step must be applied n times where n is the number of bits to be generated in the quotient.

Speed-up Fundamentals

The algorithms presented and illustrated for multiplication and division are characterized by two common properties: (1) each of them consists primarily of a recursive step, and (2) each execution of the recursive step processes a single bit of the multiplier or produces a single bit of the quotient. respectively, for multiplication

and division. Both algorithms can be implemented by a sequential circuit with storage registers for the operands, a block of combinational hardware that implements the recursive step by processing one bit of the multiplier or quotient per clock cycle.

The number of clock cycles required can be reduced from n to n/m for performing multiplication by using m copies of the recursive combinational block interconnected to process m bits, $1 < m \leq n$ of the multiplier in a single clock cycle. The approach is illustrated by use of an adder array in the following example in which $m = n$.

• EXAMPLE 3 Adder Array

Consider the multiplication of two 4-bit numbers, as shown in Figure 1. The multiplicand is X_3, X_2, X_1, X_0, the multiplier is Y_3, Y_2, Y_1, Y_0, and the product is Z_7, Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0. The first partial product is formed by multiplying X_3, X_2, X_1, X_0 by Y_0. The multiplication of two bits such as X_0 and Y_0 produces a 1 if both bits are 1; otherwise it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram. The second partial product is formed by multiplying X_3, X_2, X_1, X_0 by Y_1 and is shifted one position to the left. The sum of the two partial products is produced by a binary adder. Note that the least significant bit of the product does not have to go through an adder, since it is completely formed by the output of the first AND gate. The same approach as used for multiplier bit Y_1 is also used for multiplier bits Y_2 and Y_3 as shown in the final circuit in Figure 1. Note that the Carry Out can be a 1, so it enters the MSB of the adder at the next level down. In general, for J multiplier bits and K multiplicand bits, we need $J \times K$ AND gates and $(J - 1)$ K-bit adders to produce a product of $J + K$ bits. For the Example 1 multiplier in Figure 1, $K = 4$ and $J = 4$, so we need 16 AND gates and three 4-bit adders to produce a product of 8 bits. •

In Example 1, the entire multiplication is completed for all multiplier bits in a single clock cycle using only combinational logic. Thus, in effect, four bits of the multiplier are processed simultaneously. However, because of the cost and the delay of an adder, an additional goal with respect to processing multiple bits of the multiplier or quotient can be to process m multiplier bits *per recursive step.* In this case, only a single adder of length $n + m$ bits is required. for the step. Such an approach requires that multiples of the multiplicand be made available to the adder and additional hardware be made available to select these multiples. This second speedup approach is illustrated by the next example.

• EXAMPLE 4 Multiplier Radix Greater Than 2

Consider again the multiplication of two 4-bit numbers, as shown in Figure 1. Our goal is to use only two recursive steps instead of four to perform this multiply, reducing the number of adders required and the delay through the circuit. The first partial product is to multiply the multiplicand by Y_1, Y_0, and the second partial product is multiply the multiplicand by Y_3, Y_2. The values taken on by these bit pairs are 0, 1, 2, 3, the digits of radix 4. To process multiplier digits in radix 4, we need to provide multiples of X = X_3, X_2, X_1, X_0 by 0, 1, 2, and 3 instead of by just 0 and 1 as for radix 2. This requires that pairs of the AND operation structure from

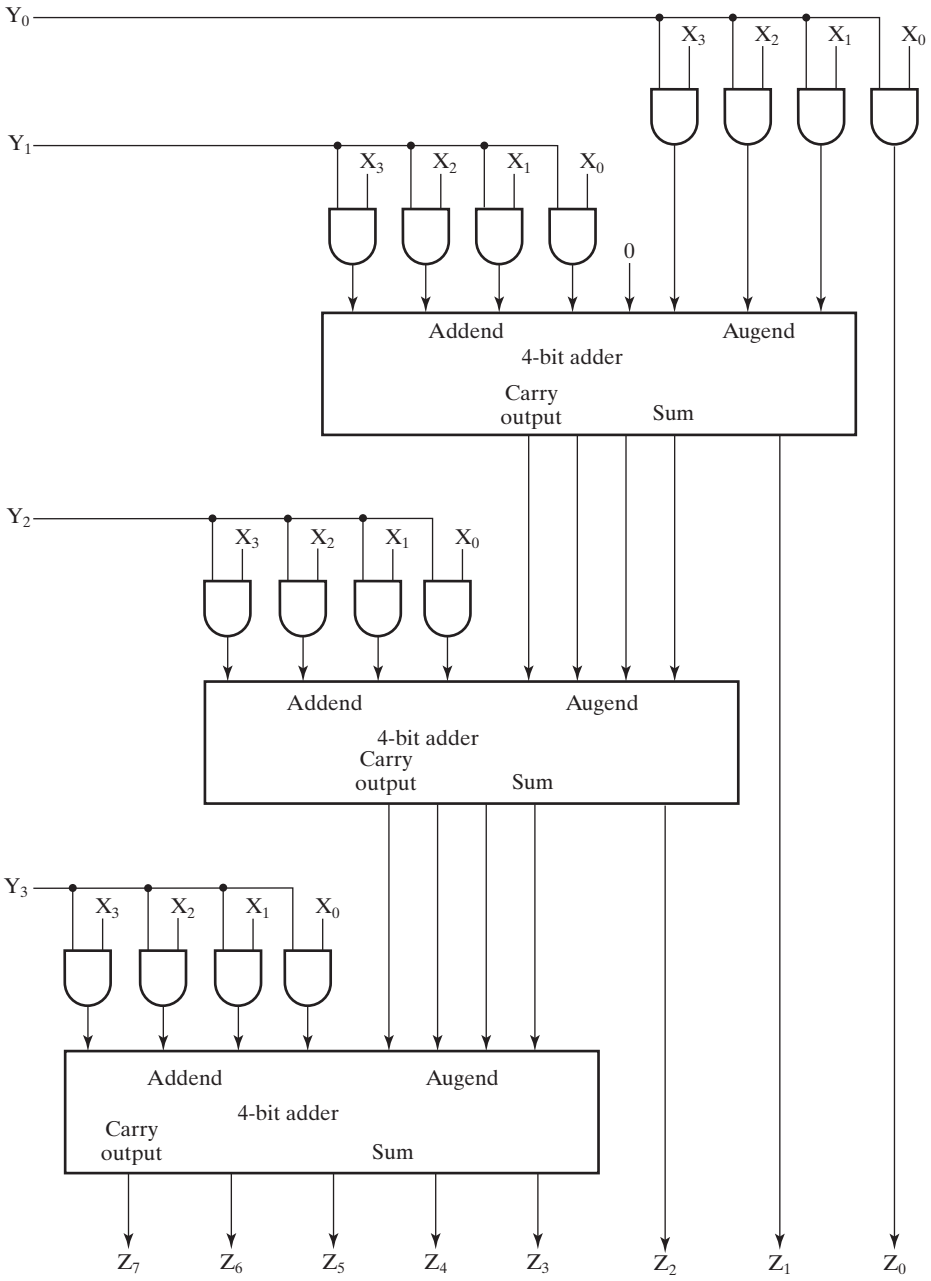

FIGURE 1
A 4-Bit by 4-Bit Binary Multiplier

Example 3 be replace by multiplexers able to select one of these four multiples of the multiplicand. The formation of the multiples and their selection is illustrated in Table 1. From the table, it is apparent that four values must be provided to be

TABLE 1 2-bit Multiplier

Multiplier Bits		Multiple of Multiplicand	
Y_{i+1}	Y_i	Multiples	Implementation
0	0	0	0
0	1	1	X
1	0	2	Shift left X by 1
1	1	3	(Shift left X by 1) + X

selected for each of the recursive steps. The first value 0 is a constant. The second value is simply X. The third value is 2X which is obtained by shifting X one position to the left, filling the vacant position with 0. The final value is 3X which is obtained by using an adder to add X to 2X. The hardware for forming these four values is shown in the upper part of Figure 2.

Since there are only two radix 4 values in the 4 × 4 multiplier, there are only two multiples to be selected from the four values in Table 1, $Y_1Y_0 \times X$ and $Y_3Y_2 \times X$. These two operands require only a single adder as shown in Figure 2. In order to select one of the four values for each adder input, a multiplexer with select inputs Y_1,Y_0 and one with select inputs Y_3,Y_2 are used. Note that bits Z_1 and Z_0 bypass the adder since they depend only on the value selected by Y_1 and Y_0, The 0s inserted are there to show the shifting of operands relative to each other and as fixed inputs can be used to simplify hardware using contraction as presented in Chapter 4. The width of the various operands external and internal to the multiplier are determined by looking at the number of bits in the incoming operands and the number of bits, including outgoing carries that are non-zero, in the results.

High-Speed Multipliers

In this section and the next section, the two speedup approaches will be applied to an example of a contemporary multiplier design and a contemporary divider design. Before presenting these examples, we need to introduce the concept of the carry save adder (CSA) which is used in both of the designs.

Carry Save Adder

The carry save adder consists of full adders as shown in Figure 3. Beginning with a ripple carry adder as a starting point, the ripple carries are disconnected. This pro-

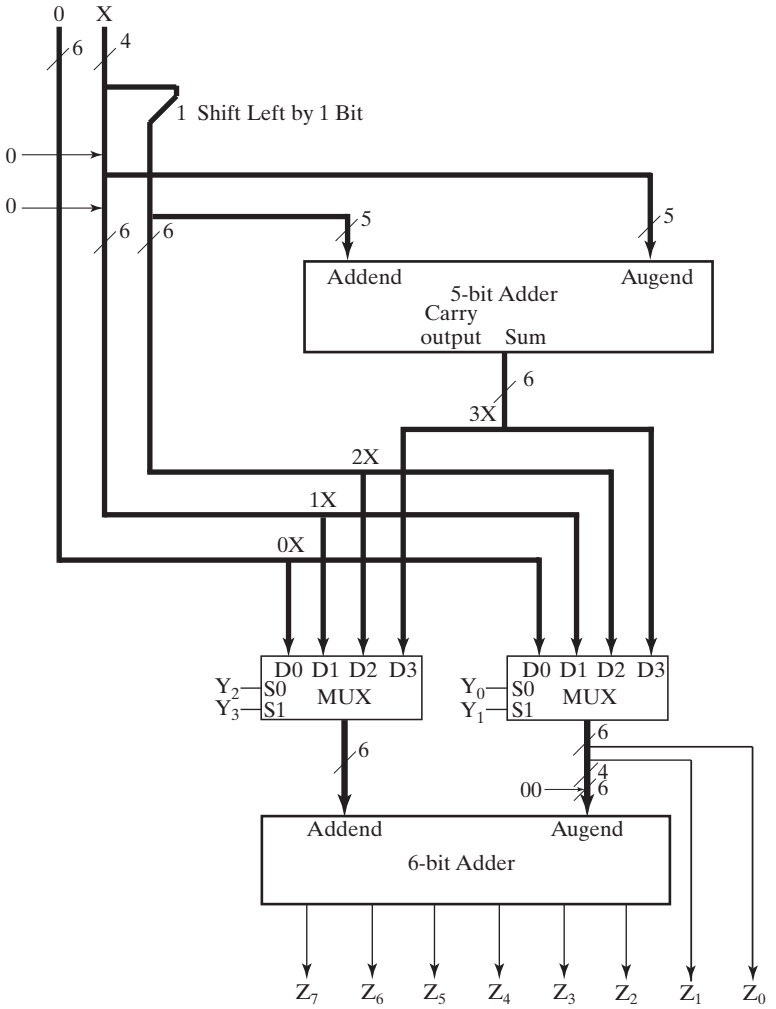

FIGURE 2
A 4-Bit by 4-Bit Binary Multiplier with Radix-4 Multiplication

vides an additional input on each full adder to apply an operand bit, permitting three inputs to the adder. Further, as well as the sum output, the carry output from each full adder is provided. Thus, the sum vector S and the carry vector C represent the actual sum of the three input operands, without the carries fully propagated. Because $2n$ bits are required to represent an n-bit sum, this is sometimes referred to as a redundant representation of the sum. These S and C vectors can be summed by a carry propagate adder to form the overall sum of the three operations. A single carry save adder is not of a great amount of value. However, replacement of all but the final carry propagate adders in Figures 1 or 2 with carefully constructed arrays of carry save adders can be of value in reducing both delay and the amount of digital logic required.

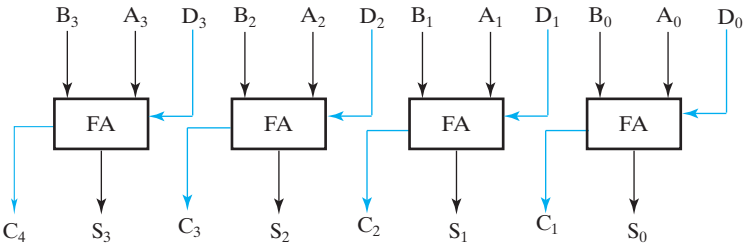

FIGURE 3
4-bit Carry-Save Adder (CSA)

WALLACE TREE MULTIPLIER There are a number of different designs for high-speed adder arrays for multiplication. The one we illustrate as an example is the Wallace tree multiplier. This multiplier consists of an arrangement of CSAs interconnected in the form of an inverted tree with the outputs at the bottom and the partial products at or near the top. The block diagram for an 8 × 8 Wallace tree multiplier for an 8-bit multiplicand X and an 8-bit multiplier Y is shown in Figure 4. Rather than showing the input and output vectors, we show the 8-bit partial products consisting of the AND of a multiplier bit Y_i with the multiplicand X. The partial products are appropriately shifted to the left depending on the value of i. In the tree, full use is made of the three inputs on each CSA. If the tree is built up from the bottom, the structure is quite apparent. The bottom component is a carry propagate adder, typically a carry lookahead adder (CLA) for improved performance. Above the CLA and the first carry save adder (CSA), pairs of C and S

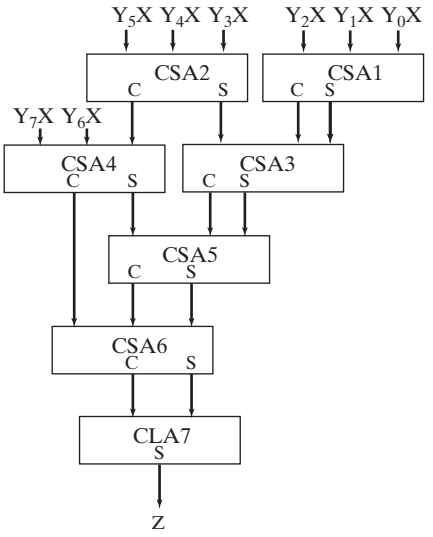

FIGURE 4
8 × 8 Wallace Tree Multiplier

inputs are applied from right to left to the triples of inputs on the row of maximum possible CSAs below. This can be continued until the number of inputs to the top row exceeds the number required. For the Wallace tree multiplier shown, the top row if completed would have 12 inputs which exceeds the eight required, one for each bit of the multiplier. The four excess inputs can be removed by pruning the tree which amounts to removing some of the CSAs in the top row and possibly in lower rows. In this case, to reduce the 12 inputs to eight inputs, two CSAs are removed from the right top. This removes six inputs and opens up two inputs in the second row, for a net reduction of four inputs.

The full logic diagram of the Wallace tree multiplier based on the diagram in Figure 4 is shown in Figure 5. The CSA are made up of full adders (FA) with three inputs and two outputs also called 3-2 adders. In some cases, along the left and right edges of the adder, there are unused inputs to the CSAs, so they degenerate into half adders (HA) with two inputs (2-2 adders) and, in extreme cases, just lines. The process or generating this logic is not insignificant since preserving the proper alignment of the C and S vectors while progressing up or down the tree is critical. In this example, the FAs and HAs are aligned so that the S bits go straight down and the C bits going down move one position to the left.

The Wallace tree shown in Figure 4 processes eight multiplier bits in five levels. In contrast, a conventional series of adders for eight multiplier bits uses seven levels. So the delay in terms of levels is reduced by 29 percent. Using the more detailed diagram in Figure 5, there are nine levels through the CSAs plus the CLA on the longest path. In the series of adders, there are 12 levels through the CLAs, for a delay reduction in terms of levels of 25 percent. So the two evaluation methods yield about the same amount of improvement in delay.

Higher Radix Multiplier Processing As illustrated in Example 2, two or more bits of a multiplier may be retired per multiply step. Just as in that example, this approach has potential for speeding up multiplication and for reducing hardware costs in multipliers using carry save adder trees. In this case, we use a radix-4 multiplier, retiring two multiplier bits per multiply step in the 8×8 multiplier.

The four partial products needed for the four multiply steps are produced by adding two multiplexers to the same circuit used in the upper part of Figure 2. The resulting circuit is shown in the upper part of Figure 6. The four partial products are fed to a 4-input CSA tree requiring just two CSA levels. In addition to dealing with more than two multiplier bits, this circuit also reduces the hardware required by a considerable amount going from a total of seven adders to four adders.

High-Speed Dividers

For each divide step, a single quotient bit is generated. Further, in order to determine the quotient bit, a trial subtraction must be performed in which the quotient bit is temporarily set to 1 and the divisor is subtracted from the current partial remainder to determine the sign of the new partial remainder. If the sign is 0 (non-negative), then the new partial remainder replaces the old one and the quotient bit

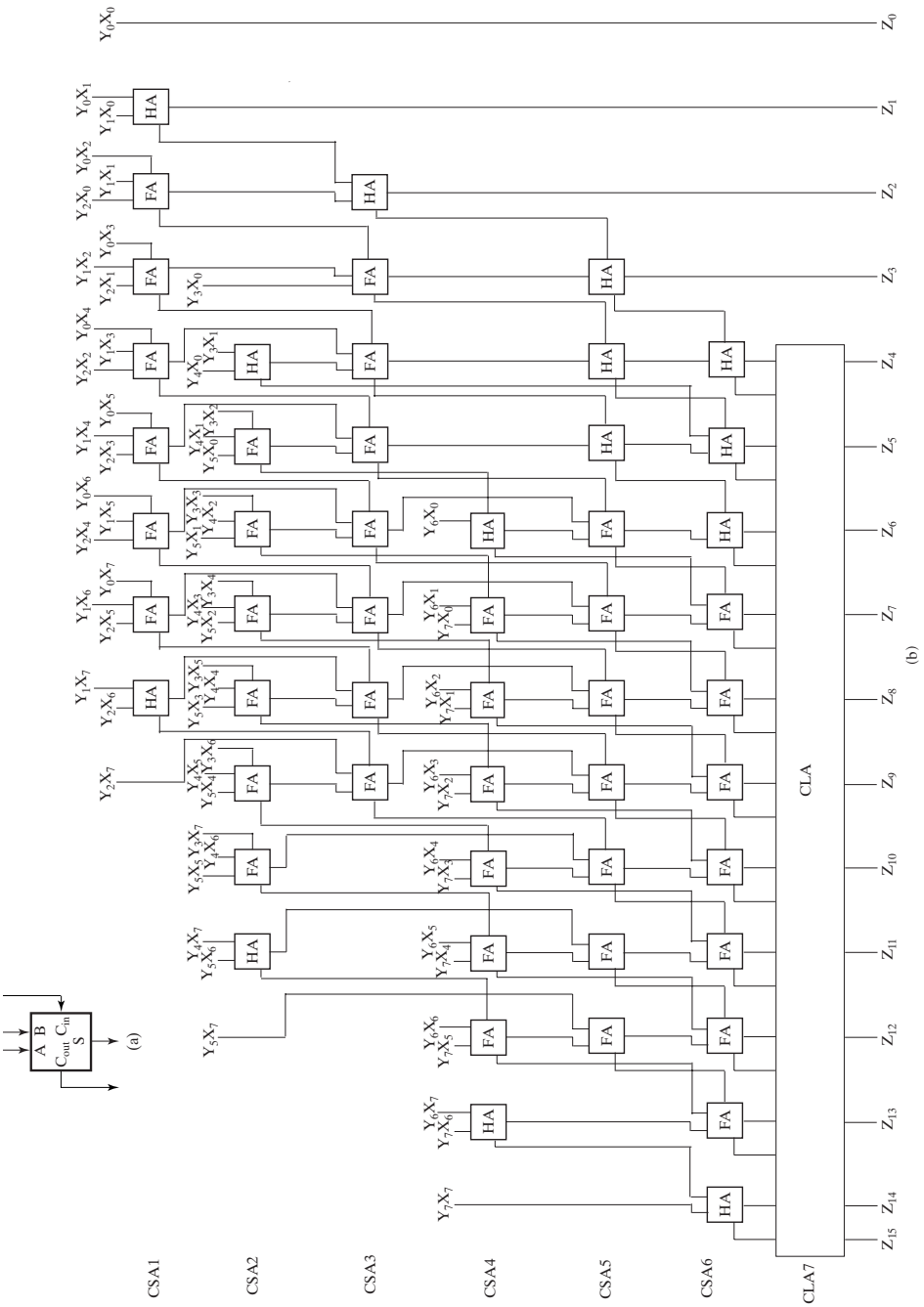

FIGURE 5
Figure 5Wallace Multiplier Details

is 1. If this sign is negative, then the quotient bit is changed to 0 and the current partial remainder retained.

To implement the above for a divide step, a subtractor is used to subtract the divisor from the appropriately shifted partial remainder. Assuming that these are

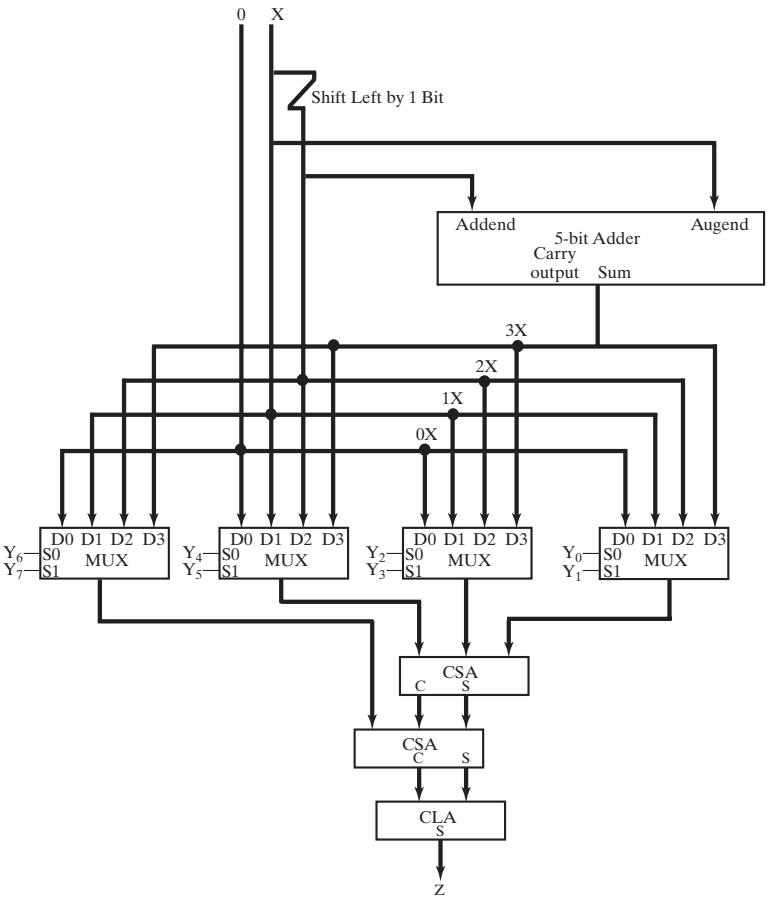

FIGURE 6
Radix-4 Multiplier

unsigned numbers, the subtraction is performed by adding the two's complement of the divisor to the partial remainder. If no carry out occurs, then the result is negative, the quotient bit is set to 0, and the old partial remainder is shifted one position to the left. If a carry occurs, then the result is non-negative, the quotient bit is set to 1, and the new partial remainder shifted one bit to the right replaces the old partial remainder.

Just as was done for multiplication, the combinational logic used to perform the above divide step can be cascaded to build a combinational divider that processes multiple quotient bits, one at a time. The requires n such circuits for n divide steps and corresponds to the cascaded adders for multiplication in Figure 1. There will be a multiplexer between adders in order to select between the new and old partial remainders. CSA adders can be used to add the inverted divisor and a 1 to the most significant n bits of the partial remainder. This performs a 2s-complement unsigned subtraction of the divisor from the partial remainder. This can also be

converted to a CSA adder cascade with multiplexers for both C and S inserted between the adders. However, due to the necessity that the subtractions be performed for each quotient bit sequentially, this cannot be converted to a CSA tree. In order to achieve greater speed up and potential reduction of hardware, the logical next step is to attempt multiple bit quotient generation.

To illustrate multiple quotient bit generation, the circuit in Figure 7 is an alternative implementation of a division method appearing in the patent reference [4]. This particular method generates the quotient radix 16, i. e., four bits at a time. Initially, we consider conceptually the problem of generating four bits of quotient in a divide step. Recall that in the one-bit-at-a-time quotient generation, a trial subtraction of the divisor from the appropriately shifted dividend is performed to determine whether the quotient is 0 or 1. Extending this approach, to generate a divisor four bits at a time could potentially require 15 trial subtractions simultaneously, a procedure that is too costly in hardware. By examining a few of the most significant divisor bits and several of the most significant shifted partial remainder bits, this can be reduced to a maximum of five trial subtractions. In the upper left of Figure 7, two divisor bits and four partial remainder bits are examined by using a lookup table[2] implemented by a combinational circuit or a high-speed memory. The outputs of the lookup table consist of five hexadecimal values that are know to contain the representation of the actual four quotient bits.

The five hexadecimal values are used to generate five multiples of the divisor that are to be subtracted from the shifted partial remainder. The shifting operations and CSA adder in the upper center of Figure 7, produce multiples of 12 and of 3 of the divisor. Shifting operators alone at the inputs to the first pair of multiplexers produce multiples of 0, 4, and 8 and 0, 1, and 2, respectively of the divisor. The pair of dual multiplexers that follow combine these multiples with the multiples of 12 and 3, respectively produced by the CSA in C and S form. The multiplexers must be dual because they have to past the C and S form. For the multiples from the first pair of multiplexers, a 0 is applied to each of the second multiplexers for the C vector. Table 2 shows how the multiples of 0, 4, 8, 12 and 0, 1, 2, 3 are selected by the four quotient bits to produce multiples of 0 through 15. These multiples enter a CSA adder tree consisting of two CSA adders to produce the divisor multiple. In order to not have to deal with taking the twos complement of the divisor multiple in C and S form, we subtract the n most significant bits of the shifted partial remainder from the divisor multiple using a cascade of a final CSA followed by a CLA. The operation performed is qD + P + 1 = $qD - P$. The carry out is that of $qD - P$. which is 0 for $P - qD$ negative. The entire circuit, encircled by a dashed

[2]This implementation uses a pre-shift of the dividend and divisors so that they are *normalized* as done for floating point numbers. Each of these two operands are shift to the left so that the MSB has value 1. Now that the divisor always contains a 1 as its MSB, the two divisor bits used for the lookup table are the two bits following the MSB since the fixed 1 provides no information. Thus the MSB is implicitly being used giving three divisor bits. The amount that each of the operands is shifted to the left is saved. For fixed point division, this shift information controls post-shifts applied to the quotient and the remainder to change them from floating point to fixed point format.

• **TABLE 2 Formation of Multiples of 0 through 15**

Multiple	Most Significant Half	Least Significant Half
0	0	0
1	0	1
2	0	2
3	0	3
4	4	0
5	4	1
6	4	2
7	4	3
8	8	0
9	8	1
10	8	2
11	8	3
12	12	0
13	12	1
14	12	2
15	12	3

line, is called a Trial Divide Step (TDS) circuit and is replicated five times to perform the five trial divisions. The carry outs of the five TDSs enter a priority encoder with TD5's carry out highest priority D (TD0 through TD4 have the divisors ordered from smallest to largest). The priority encoder selects the largest divisor that results in a carry out of 1, corresponding to the largest quotient that produces a positive partial remainder, which is the actual quotient. This quotient is selected from the output of the multiplexer below the lookup table using the priority encoder outputs. Likewise, the 2s-complement of n bits of the new partial remainder is selected by the multiplexer driven by the TDSs. This is inverted and +1 is added to obtain the new partial remainder bits which are loaded into the most significant n bits of the new partial remainder which is also shifted by four to the left. For an n-bit divisor, this circuit is used k times sequentially where k is an integer, $k \geq n/4$.

References

1. MANO, M. M. AND C. R. KIME. *Logic and Computer Design Fundamentals, 4th* ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.
2. MANO, M. M. AND C. R. KIME. *Logic and Computer Design Fundamentals, 3rd* ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.
3. HENNESSY, J. L. AND D. A. PATTERSON, *Computer Architecture, a Quantitative Approach*, 4th ed. Morgan Kaufmann/Elsevier, Inc., 2007.
4. SHEAFFER, G. S., *Computer for Performing Non-Restoring Division,* United States Patent 5,818,745, United States Patent and Trademark Office, http://www.uspto.gov/, October 6, 1998.

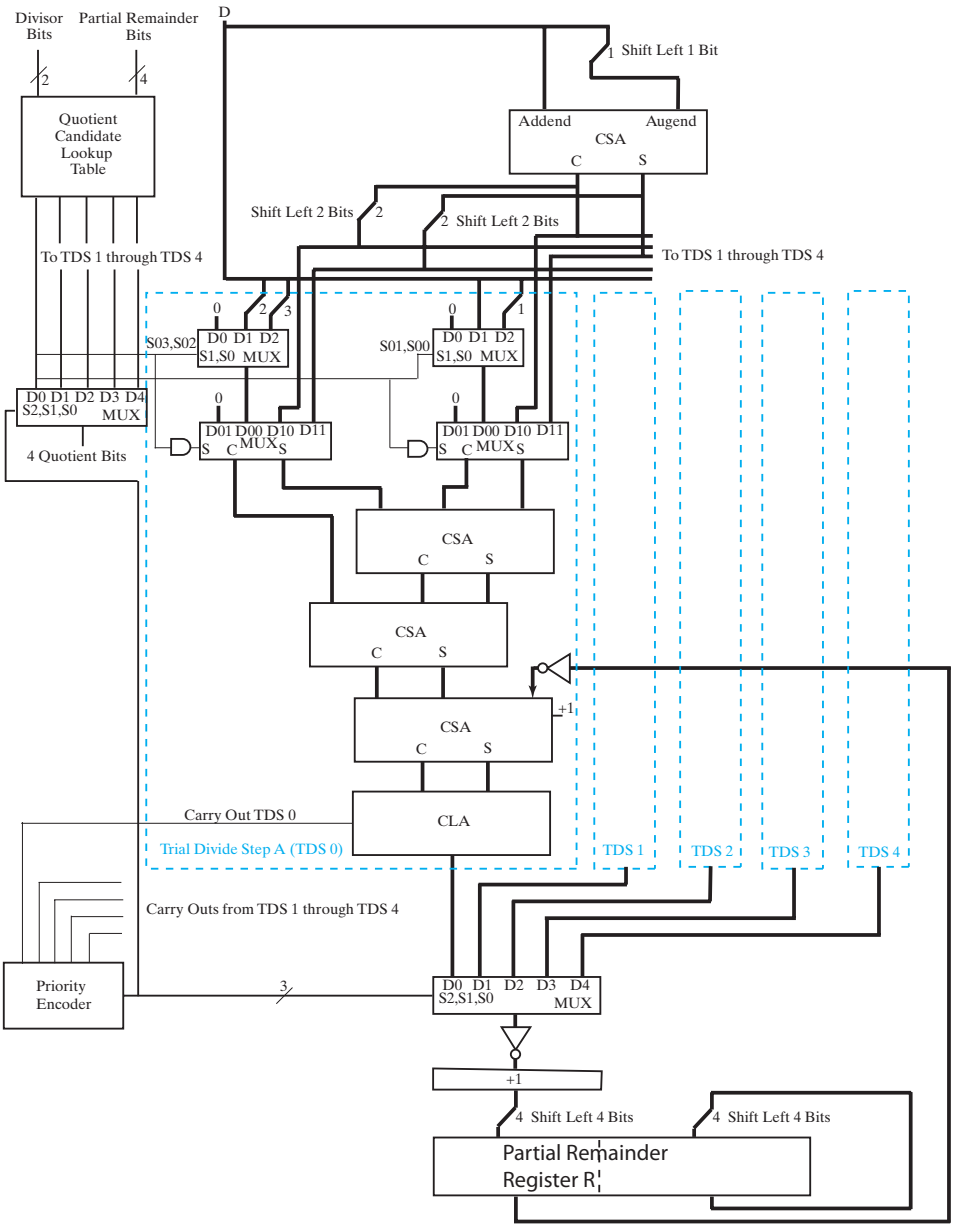

FIGURE 7
Radix-16 Divider

Problems

1. Perform a fixed point multiplication with multiplier 11001001 and multiplicand 11100011.

2. Perform a fixed point division with divisor 00001011 and dividend 11100011.

3. Construct a Wallace CSA block diagram for an 8×8 multiplier.

4. Many multiple-bit processing algorithms for multiplication and division use both positive and negative coefficients times the multiplicand or divisor, respectively. These methods avoid the use of adders in forming multiples, requiring only multiplicand or divisor shifts to generate multiples. For multiplication, one of these schemes is based on the Booth algorithm, which uses signed 2s-complement operations and result, and 2s-complement subtraction. In this algorithm for processing of two multiplier bits at a time, a bit position m_{-1}, initialized to 0, is added to the right of the LSB of the multiplier. For each step, the multiplier is shifted right by two positions with its LSB going into m_{-1}. Using the 3-bit combination (m_1, m_0, m_{-1}) as input, the multiple of the multiplicand M to be used for processing bit pair (m_1, m_0) is determined using Table 3. After the operation specified by the table, the product and multiplier are shifted two bit positions to the right.

TABLE 3 2-bit Booth Algorithm Recoding

Multiplier Bits			Multiple of Multiplicand	
m_1	m_0	m_{-1}	Multiple	Implementation
0	0	0	0	Constant 0
0	0	1	+1	Add M
0	1	0	+1	Add M
0	1	1	+2	Add M left-shifted by 1
1	0	0	−2	Subtract M left-shifted by 1
1	0	1	−1	Subtract M
1	1	0	−1	Subtract M
1	1	1	0	Constant 0

(a) Show the multiple of M required for each pair of bits in the multiplier 1011100010. Remember to append the 0 to the right of the LSB before applying Table 3.

(b) Sketch an implementation of the hardware required for a multiplication step.