No more tedious times tables to recite; try binary multiplication in 6502 assembly language.

Multiplying by 1's and 0's

By Leo J. Scanlon

Multiplication. It's a subject that gave most of us untold misery in elementary school. But now, having memorized all of those cursed multiplication tables, we can confidently multiply anything by anything. Right?

Yes, right, as long as the anythings are *decimal* numbers. Unfortunately, if you want to write an assembly language program to perform a multiplication on your Apple, AIM 65, TRS-80 or other microcomputer, you're back at square one, because (a) the numbers being multiplied are *binary* values and (b) none of the popular eight-bit microprocessors has a multiply instruction.

That's the bad news. The good news is that multiplication techniques are covered in virtually every tutorial and textbook on assembly language programming. Regardless of which book you pick up, you'll learn how it's possible to multiply two binary values, using the add and shift instructions of the microprocessor. The discussions of multiplication vary in quality—depending on the author's inclination and the software or hardware orientation of the book—but typically these books give only a summary of the fundamental principles and an example or two (usually, just an eight-bit by eight-bit unsigned multiply). From there on, readers are left to their own devices.

A cop-out? Maybe. But most authors (and, apparently, their publishers) feel that their book must cover so many topics that they can't afford to devote much page space to a "simple" task like multiplication.

Admittedly, a brief once-over will suit the needs of the casual reader/programmer, but what about those who want to multiply numbers that are longer than eight bits, or those who want to multiply *signed* (two's complement) numbers?

This article is intended to serve both kinds of readers: beginners who want an overview of multiplication and serious programmers who need a more detailed treatment than the popular literature provides. The programs are written in 6502 assembly language, but the accompanying text and flowcharts should make them readily convertible for use with any of the other eight-bit microprocessors.

Back to Fourth Grade

Before we discuss multiplying binary numbers, it may be instructive to review the mechanics of multiplying decimal numbers, the way we used to do it in elementary school with pencil and paper. As you will recall (in these days of calculators, it may be a bit hazy), with pencil and paper you write the multiplicand on one line and the multiplier on the line below it, and then grind out a series of multiplications—one for each digit in the multiplier. Each partial product is recorded directly below its multiplier digit, causing it to be offset one digit position to the left of the preceding partial product. When all of the partial products have been calculated, they are added to produce the final product.

For example, multiplying the number 124 by the number 103 looks like this:

124	Miltiplicand
× 103	Multiplier
372	Partial Product #1
000	Partial Product #2
124	Partial Product #3
12772	Final Product

(Of course, you don't normally write down the all-zeroes partial product, but rather just skip to the next digit position.)

It's important to remember why we write the partial products offset from each other: it's because each partial product is associated with a multiplier digit of a different decimal weight. Remember that the preceding problem can also be written in the form

$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$

Or, we could use an equivalent form

$103 \times 124 = (3 \times 1 \times 124) + (0 \times 10 \times 124) + (1 \times 100 \times 124)$

to illustrate that the digits 3, 0 and 1 in the multiplier have decimal weights of 1, 10 and 100, respectively.

In the preceding problem, both the multiplier and the multiplicand happen to be nonnegative (positive or unsigned) numbers. How would the pencil-and-paper operation have changed if one or both was a negative

Address correspondence to Leo J. Scanlon, 23021A Village Drive, El Toro, CA 92630.

```
This subroutine multiplies an 8-bit unsigned multiplicand (MPND)
by an 8-bit unsigned multiplier (MPLR), and returns the 16-bit
product in locations PROD (low byte) and PROD+1 (high byte).

0000                MPLR=$20
0000                MPND=$21
0000                PROD=$22
0000                *=$400
0400 A9 00  MLT8U   LDA  #0       Clear product MSBY
0402 A2 08          LDX  #8       Multiplier bit count = 8
0404 46 20  NXTBT   LSR  MPLR     Get next multiplier bit
0406 90 03          BCC  ALIGN    Multiplier bit = 1?
0408 18             CLC           Yes, add multiplicand
0409 65 21          ADC  MPND      to partial product
040B 6A     ALIGN   ROR  A        Rotate product right
040C 66 22          ROR  PROD
040E CA             DEX
040F D0 F3          BNE  NXTBT    Loop until 8 bits are done
0411 85 23          STA  PROD+1   Store product MSBY
0413 60             RTS
```

Example 1. An eight-bit by eight-bit unsigned multiplication subroutine.

number? Very little, because we all know that if one number is positive and the other is negative, the product will be negative—so you tack a minus sign onto the answer. Similarly, if both multiplier and multiplicand are negative, the product will be positive—so you omit the minus sign from the answer (or put a plus sign on it, if you're a purist). With binary numbers, though, there's quite a bit of difference between multiplying unsigned numbers and multiplying signed numbers, because signed numbers are represented in two's complement form. We'll be looking at both unsigned and signed multiplication later, but for now, let's briefly discuss how binary numbers can be multiplied.

Binary Multiplicaton vs. Decimal Multiplication

Binary multiplication is much simpler than decimal multiplication, because binary multipliers consist of only the digits 0 and 1, whereas decimal multipliers can be made up of the digits 0 through 9. In binary multiplication, the partial product will always be simply a copy of the multiplicand if the multiplier digit is 1, and it will be 0 if the multiplier digit is 0.

The binary equivalent of our previous 103 x 124 example looks like this:

```
     01111100   Multiplicand (=124)
   × 01100111   Multiplier (=103)
    01111100
   01111100
  01111100
 00000000
00000000
01111100
01111100
00000000
011000111100100   Final Product (=12772)
```

This example gives a good indication of the way in which eight-bit microprocessors must perform multiplication operations. There are some important differences, however. When performing a binary multiplication by hand, you must calculate the final product by adding the partial products, column-by-column. (And if you think that's easy, try it! The carries can drive you bonkers.) The operation is similar on a computer, but instead of waiting until all of the individual partial products are calculated before deriving the final product, *computer programs update the partial product after each multiplier bit is examined.* By doing this, the final product is generated when the last bit of the multiplier has been processed.

Moreover, when multiplying by hand, each partial product is offset one digit position to the left of the preceding partial product, to account for the weight of the multiplier bit. In a computer, it is easier to shift the partial product each time it is updated, thereby aligning it to receive the contribution of the next multiplier bit. The partial product may be shifted either right or left, depending on whether your program is examining the multiplier bits from right to left (low-order to high-order) or from left to right (high-order to low-order). In this article, the multiplier will be processed right-to-left, the way you would do it by hand, so the partial product will be shifted to the right.

In summary, the following applies when multiplying binary numbers by a computer:

If the multiplier bit is a 1, add the multiplicand to the partial product, and then shift the sum one bit position to the right. If the multiplier bit is a 0, shift the current partial product one bit position to the right, with no addition.

In writing a multiplication program for your microcomputer, which instruction would you expect to use to perform the add and right-shift operations? With a 6502-based microcomputer, such as Apple II, KIM-1, SYM-1 or AIM 65, the addition will be performed with the 6502's only add instruction, add to accumulator with carry (ADC). The shifting will be performed with the shift right (LSR) or rotate right (ROR) instruction.

With this background, let's examine the software that will be needed to multiply unsigned or signed numbers.

Multiplying Unsigned Numbers

As you probably know, in an unsigned number, each data bit carries a certain binary weight, according to its position within the number. Data bits are numbered from right to left, with the rightmost bit labeled as bit 0 and the leftmost bit labeled bit 7. The bit numbering scheme has a direct correlation to the binary weights, in that bit 0 has a weight of 2^0 (decimal 1) and bit 7 has a weight of 2^7 (decimal 128). Therefore, a single byte can represent an unsigned number from decimal 0 (binary 00000000) to decimal 255 (binary 11111111).

Single-Precision Unsigned Multiplication

Certainly, the easiest place to begin is by developing a program—a subroutine, actually—to multiply two eight-bit unsigned numbers in memory. If the multiplicand and the multiplier are both eight bits long, how long will the product be? Well, we know that the worst case involves multiplying 255 by 255, which gives a product of 65,025. To hold a value of 65,025, we need 16 bits, or two bytes.

At this point, we can draw the flowchart that will serve as the blueprint for an eight-bit (or single-precision) multiplication subroutine. This flowchart must do five things:

1. Initialize a two-byte product in memory to zero, and a multiplier bit counter to eight.
2. Shift the multiplier right one bit position into carry.
3. Interrogate the state of the multiplier bit that's in carry. If carry = 1, add the multiplicand to the partial product.
4. Rotate the partial product right, into the final product location's least-

```
This subroutine multiplies a 16-bit unsigned multiplicand (MPND,
MPND+1) by a 16-bit unsigned multiplier (MPLR, MPLR+1), and
returns the 32-bit product in locations PROD (low byte), PROD+1,
PROD+2 and PROD+3 (high byte).

0000                MPLR=$20
0000                MPND=$22
0000                PROD=$24
0000                *=$400
0400 A9 00   MLT16  LDA  #0         Clear P2 and P3 of product
0402 85 26          STA  PROD+2
0404 85 27          STA  PROD+3
0406 A2 10          LDX  #16        Multiplier bit count = 16
0408 46 21   NXTBT  LSR  MPLR+1     Get next multiplier bit
040A 66 20          ROR  MPLR        into Carry
040C 90 0B          BCC  ALIGN      Multiplier bit = 1?
040E A5 26          LDA  PROD+2     Yes, fetch P2
0410 18             CLC              and add M0 to it
0411 65 22          ADC  MPND
0413 85 26          STA  PROD+2     Store new P2
0415 A5 27          LDA  PROD+3     Fetch P3
0417 65 23          ADC  MPND+1      and add M1 to it
0419 6A      ALIGN  ROR  A          Rotate product right
041A 85 27          STA  PROD+3
041C 66 26          ROR  PROD+2
041E 66 25          ROR  PROD+1
0420 66 24          ROR  PROD
0422 CA             DEX             Decrement bit count
0423 D0 E3          BNE  NXTBT      Loop until 16 bits are done
0425 60             RTS
```

Example 2. A 16-bit by 16-bit unsigned multiplication subroutine.

significant byte (LSBY).

5. Decrement the multiplier count. If it is zero, store the final product's most-significant byte (MSBY) in memory, and return; otherwise, go back to process the next multiplier bit.

The flowchart in Fig. 1 performs these five tasks.

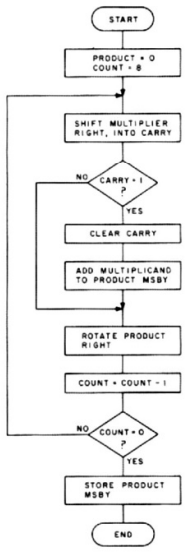

Fig. 1. An eight-bit by eight-bit unsigned multiplication algorithm.

Example 1 is a subroutine (MLT8U) that uses the flowcharted algorithm to multiply the contents of a multiplicand in memory (MPND, assigned to location $21 here) by the contents of a multiplier in memory (MPLR, assigned to location $20 here). The 16-bit product is returned in two consecutive locations, PROD and PROD + 1. The X register is used to hold the multiplier bit count.

In the MLT8U subroutine, the LSR MPLR instruction at NXTBT causes the multiplier (in memory location $20) to be shifted, one bit at a time, into carry. If the shifted multiplier bit is a one, the CLC and ADC MPND instructions add the multiplicand to the most-significant byte of the product, and the two rotate instructions at ALIGN (ROR A and ROR PROD) shift the partial product to the right, into the least-significant byte. If the shifted bit of the multiplier is a zero, BCC ALIGN bypasses the add-multiplicand operation by branching to the rotate-right sequence at ALIGN. The NXTBT loop is executed eight times, once for each bit in the multiplier.

Double-Precision Unsigned Multiplication

We've just concluded a discussion of eight-bit by eight-bit unsigned multiplication, which lets us multiply two numbers as large as 255. Unfortunately, like a three-legged race team in the Boston Marathon, single-precision multiplication is nice, but not very practical, since many applications require numbers larger than 255 to be multiplied. Obviously, the next step is to develop a program that will multiply 16-bit, or double-precision, numbers.

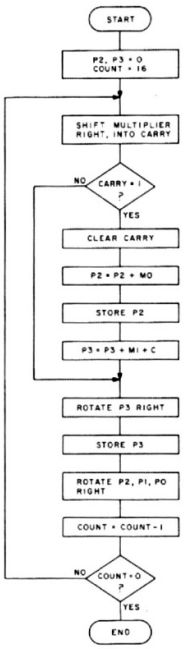

Fig. 2. A 16-bit by 16-bit unsigned multiplication algorithm.

Multiplying 16-bit numbers is somewhat more complex than multiplying eight-bit numbers, due to the additional memory involved, but the add-and-shift procedure still applies. With a double-precision multiplication, the multipler and multiplicand are both 16-bit values, so the product will occupy 32 bits (or four bytes) in memory.

Fig. 2 is a flowchart for a double-precision unsigned multiplication subroutine. The two-byte multiplicand is represented by the symbols M0 (low order byte) and M1 (high-order byte). The four-byte product is represented by the symbols P0 (low-order byte), P1, P2 and P3 (high-order byte). The double-precision algorithm shown in Fig. 2 operates similarly to the single-precision algorithm that has just been discussed. That is, the multiplicand is added to the high-order half of the partial product (P2 and P3, here) if carry is a 1. The result is then rotated one bit to the right,

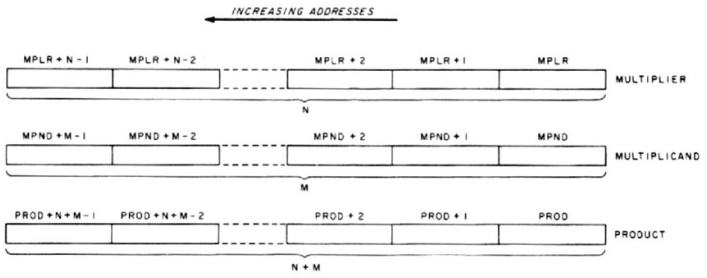

Fig. 3. Multiplier, multiplicand and product in memory.

along with the low half of the partial product.

Example 2 is a subroutine (MLT16) that uses the flowcharted algorithm (Fig. 2) to perform a double-precision unsigned multiplication. In this particular subroutine, the multiplier (MPLR) is in locations $20 and $21, the multiplicand (MPND) is in locations $22 and $23 and the 32-bit product is returned in locations $24 (low byte) through $27 (high byte). Each time a multiplier bit is a one, the MLT16 subroutine must perform two adds, to add the multiplicand to the partial product, and two stores, to update the product in memory. It must also perform four rotate operations (one for each byte in the product). Note that the updated byte P3 is returned to memory *after* the first of the four rotates.

Whither Goest the Multiplier?

If you've run either Example 1 or Example 2 on a microcomputer, you've observed that the multiplier has been cleared to zero in the course of the operation. In many applications, it makes no difference whether or not the multiplier is affected. In other applications, the multiplier must be preserved for one reason or another.

How can the multiplier be preserved? Typically, the first impulse is to preserve it by simply changing the LSR instruction at ALIGN to an ROR instruction. Well, that's a good beginning, but you must be careful to observe that for a rotate to work correctly, the sense of the carry bit must be unchanged from one rotate operation to another. Unfortunately, the add operation between labels NXTBT and ALIGN will *always* alter the carry bit!

With these two considerations in mind, we're on our way to a solution. So far, we've discovered that if you want to preserve the multiplier you must

1. Rotate, rather than shift, the multiplier, and
2. Save the state of the carry bit between rotate operations.

Will these two modifications alone preserve the multiplier? Well, almost. They don't quite do the job because the final rotate operation will leave the most-significant multiplier bit in carry, and all other less-significant multiplier bits displaced one bit position to the left. So, in addition, you must

3. Rotate the multiplier one more time, at the end of the operation.

By applying these three rules to Example 2, you may come up with a subroutine that looks like the one in Example 3, a double-precision unsigned multiplication subroutine in which the multiplier is preserved. Since the multiplier must be rotated at two different points in the program (per rules 1 and 3), the rotate instructions are given in a subroutine, called RMPLR. Rule 2 is easily satisfied by executing a push processor status (PHP) instruction after the call to RMPLR, and a complementary pull processor status (PLP) instruction just after the partial product is rotated.

Multiprecision Unsigned Multiplication

Many real-world applications involve multiplying numbers that are longer than two bytes, or have a multiplier and multiplicand of different lengths. For these reasons, it is worthwhile to wind up this discussion of unsigned multiplication by developing a subroutine that can multiply unsigned numbers of *any* length. That is, we will develop a subroutine that multiplies an M-byte multiplicand by an N-byte multiplier, and yields an (N+M)-byte product. Fig. 3 shows how these terms are stored in memory.

The overall approach is unchanged for this general case, but since the multiplier and multiplicand are variable-length values, we will have to compute parameters that were *known* in the eight-bit and 16-bit multiplications. Here, for example, the number

```
This subroutine is a modified version of the subroutine in
Example 2.  It has some additional instructions to return the
multiplier in its original form.

0000                    MPLR=$20
0000                    MPND=$22
0000                    PROD=$24
0000                    *=$400
0400 A9 00      MLT16   LDA  #0        Clear P2 and P3 of product
0402 85 26              STA  PROD+2
0404 85 27              STA  PROD+3
0406 A2 10              LDX  #16       Multiplier bit count = 16
0408 18                 CLC
0409 20 27 04   NXTBT   JSR  RMPLR     Go get next multiplier bit
040C 08                 PHP             and save it (carry)
040D 90 0B              BCC  ALIGN     Multiplier bit = 1?
040F A5 26              LDA  PROD+2    Yes, fetch P2
0411 18                 CLC             and add M0 to it
0412 65 22              ADC  MPND
0414 85 26              STA  PROD+2    Store new P2
0416 A5 27              LDA  PROD+3    Fetch P3
0418 65 23              ADC  MPND+1     and add M1 to it
041A 6A         ALIGN   ROR  A         Rotate product right
041B 85 27              STA  PROD+3
041D 66 26              ROR  PROD+2
041F 66 25              ROR  PROD+1
0421 66 24              ROR  PROD
0423 28                 PLP            Retrieve Carry from stack
0424 CA                 DEX            Decrement bit count
0425 D0 E2              BNE  NXTBT     Loop until 16 bits are done
0427 66 21      RMPLR   ROR  MPLR+1    Rotate multiplier right
0429 66 20              ROR  MPLR
042B 60                 RTS
```

Example 3. A double-precision unsigned multiplication subroutine that preserves the multiplier.

of bits in the multiplier must be determined by multiplying N by eight (with three left-shifts). The length of the product must also be computed, by adding N and M. And whenever the multiplicand needs to be added to the partial product, it will have to be added to the most-significant "M" bytes—still another required computation. These various computations are reflected in Fig. 4, the flowchart for multiprecision unsigned multiplication.

As you can see, the flowchart in Fig. 4 contains the same number of steps as the flowchart for the double-precision case (Fig. 2). That's not surprising, since the same types of operations are being performed, but how do the actual multiplication subroutines compare? That is, what programming overhead is involved in having a general-purpose subroutine rather than a length-specific subroutine? The answer is apparent by comparing Example 4, the multiprecision unsigned multiplication subroutine, with Example 3, the 16-bit multiplication subroutine. The multiprecision subroutine is *twice as long* as the double-precision subroutine!

Although Example 4 looks complicated, it isn't. Its initialization includes two additional input parameters (the multiplier length, N, and the multiplicand length, M) and two symbolic locations (to hold the product index, PINDX, and the multiplier bit count, MBIT). The subroutine itself differs very little from Example 3, except for the various computations and the use of indexes for addressing.

What lengths of numbers can be multiplied by the subroutine shown in Example 4? Well, you can see that the first seven instructions add N and M, and put the result in an eight-bit register (Y) and an eight-bit memory location (PINDX); therefore, the sum of N and M must not exceed decimal 255. The multiplier bit count ($8 \times N$) is also stored in an eight-bit memory location, so the value of N must not exceed 31. These two limitations allow us to conclude that the multiplicand length, M, must not exceed (255 – 31)

```
This subroutine multiplies two variable-length, unsigned
integers.  The multiplier is stored starting at location
MPLR, and is N bytes long.  The multiplicand is stored
starting at location MPND, and is M bytes long.  The
product will be "N + M" bytes long, and will be returned
in memory starting at location PROD.
This subroutine affects the A, X and Y registers.

0000                    N=$20         Multiplier length
0000                    M=$21         Multiplicand length
0000                    MPLR=$30      Multiplier loc.
0000                    MPND=$40      Multiplicand loc.
0000                    PROD=$50      Product loc.
0000                    PINDX *=*+1   Product index
0001                    MBIT *=*+1    Multiplier bit count
0002                    *=$600
0600 18          MMPYU  CLC           Calculate product index
0601 A5 20              LDA  N        (N+M-1)
0603 AA                 TAX
0604 65 21              ADC  M
0606 A8                 TAY           and save it in Y
0607 88                 DEY
0608 84 00              STY  PINDX    and in PINDX
060A A9 00              LDA  #0       Clear the high M bytes
060C 99 50 00   CLRP    STA  PROD,Y   of the product
060F 88                 DEY
0610 C4 20              CPY  N
0612 B0 F8              BCS  CLRP
0614 8A                 TXA           Calculate multiplier bit count
0615 0A                 ASL  A
0616 0A                 ASL  A
0617 0A                 ASL  A
0618 85 01              STA  MBIT     and save it in MBIT
061A 20 4A 06   NXTBT   JSR  RMPLR    Get next multiplier bit
061D 08                 PHP           Save resulting Carry
061E 90 E1              BCC  ALIGN    Multiplier bit = 1?
0620 A4 20              LDY  N        Yes.  Add multiplicand to
0622 A2 00              LDX  #0       high M bytes of product
0624 18                 CLC
0625 B9 50 00   AMPND    LDA  PROD,Y
0628 75 40              ADC  MPND,X
062A 99 50 00           STA  PROD,Y
062D C8                 INY
062E E8                 INX
062F 08                 PHP           Save Carry between adds
0630 C4 00              CPY  PINDX
0632 F0 04              BEQ  PULLC
0634 90 02              BCC  PULLC
0636 B0 04              BCS  GOROT
0638 28          PULLC  PLP
0639 4C 25 06           JMP  AMPND
063C 28          GOROT  PLP
063D 18                 CLC
063E A6 00       ALIGN  LDX  PINDX
0640 76 50       RPROD  ROR  PROD,X
0642 CA                 DEX
0643 10 FB              BPL  RPROD
0645 28                 PLP           Retrieve Carry from stack
0646 C6 01              DEC  MBIT     Multiplier fully processed?
0648 D0 D0              BNE  NXTBT    No.  Loop for next bit
064A A6 20       RMPLR  LDX  N        Yes.  Rotate multiplier right
064C CA                 DEX
064D 76 30       RBYTE  ROR  MPLR,X
064F CA                 DEX
0650 10 FB              BPL  RBYTE
0652 60                 RTS
```

Example 4. A multiprecision unsigned multiplication subroutine.

See List of Advertisers on page 210

=224. In summary, then, *the subroutine in Example 4 can multiply a multiplicand up to 244 bytes long by a multiplier up to 31 bytes long, to yield a product that can be up to 255 bytes long.*

Multiplying Signed Numbers

Subroutines that have been developed in the preceding parts of this article can be used to multiply signed numbers as well as unsigned numbers, provided that the signed multiplier and multiplicand are both positive. In other words, the preceding subroutines can be used to multiply *non-negative* integer numbers. However, many applications require a negative multiplicand to be multiplied by a positive multiplier (or vice versa), or a negative multiplicand to be multiplied by a negative multiplier.

The signs of the multiplier and multiplicand present no problem if you are multiplying decimal numbers with pencil and paper, because you can simply attach a minus sign to the answer if either of the operands was negative! Unfortunately, things don't go that easily if you are multiplying two's complement binary numbers.

Pencil-and-Paper Two's Complement Multiplication

To see the problems you get into

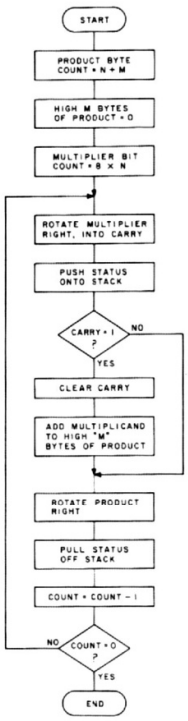

Fig. 4. A multiprecision unsigned multiplication algorithm.

This article deals with multiplication operations on signed and unsigned binary numbers. In an unsigned number, each data bit carries a certain binary weight, according to its position within the number. Within each byte, data bits are numbered from right to left, with the rightmost bit labeled as bit 0 and the leftmost bit labeled as bit 7.

This bit numbering scheme has a direct correlation to the binary weights in that bit 0 has a weight of 2^0 (decimal 1), bit 1 has a weight of 2^1 (decimal 2), and so on. Thus, bit 7 has a weight of 2^7 (decimal 128). The assignments can be summarized as follows:

7	6	5	4	3	2	1	0	Bit position
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Binary weight
128	64	32	16	8	4	2	1	Equivalent decimal weight

As you can see, a single byte can represent an unsigned number from 0 (binary 00000000) to decimal 255 (binary 11111111).

In a signed number, the seven low-order bits (bits 0 through 6) represent data, and have the same relative weights as the bits in unsigned numbers. The most-significant bit (bit 7) represents the sign of the number. If the number is positive or zero, bit 7 is a logic 0. If the number is negative, bit 7 is a logic 1. A single byte can represent a positive signed number from 0 (binary 00000000) to +127 (binary 01111111), or a negative signed number from −1 (binary 11111111) to −128 (binary 10000000).

Why is −1 represented by binary 11111111, rather than by 10000001? The answer is that negative signed numbers are represented in two's complement form. The two's complement form was introduced to eliminate the problems that are associated with allowing zero to be represented in two separate forms, binary combination 00000000 (the positive form) and binary combination 10000000 (the negative form). Using two's complement, zero is represented by only one form, the binary combination 00000000.

To derive the negative two's complement form of a binary number, you simply take the positive form of the number, reverse the sense of each bit (change each 1 to a 0, and each 0 to a 1) and add 1 to the result. The following example shows the steps required in deriving the binary representation of −32 in two's complement form:

```
+00100000   +32
 11011111   One's complement
+       1   Add 1
 11100000   −32 in two's complement form
```

From the book 6502 Software Design *by Leo J. Scanlon. It is reproduced here with permission of the publisher, Howard W. Sams & Co., Inc.*

multiplying with two's complement negative numbers, let's work out our 103 times 124 example once more, but with a negative multiplier (−103). The pencil-and-paper version will look like this:

```
     01111100   Multiplicand (= +124)
   × 10011001   Multiplier (= −103)
    01111100
   00000000
  00000000
 01111100
01111100
00000000
00000000
01111100
0100101000011100   Product (= +18972)
```

Not only is this answer too large (recall that the correct magnitude is 12772), but it has the wrong sign as well! Incidentally, the situation does not improve if you multiply −124 by −103. That multiplication will give you a product of +20196.

What, then, can be done to obtain a correct product when we want to multiply negative numbers? Certainly, one valid solution would be to take the two's complement of the negative operand(s), then multiply these two now-positive numbers. If just one of the operands was negative, the resulting product must be two's complemented. If both of the operands were negative, the (positive) product is correct as it stands.

Booth's Algorithm for Signed Multiplication

A much faster solution, and one that does not require either operand to be altered nor the product to be adjusted, is to use a method called Booth's Algorithm. This algorithm is implemented in many of the multiplier chips on the market, and is fully described in Advanced Micro Devices' *Digital Signal Processing Handbook* (John R. Mick, "Understanding Booth's Algorithm in 2's Complement Digital Multiplication," pp. 5–23 through 5–27).

Booth's Algorithm takes advantage of the fact that a string of zeroes in the multiplier requires no additions, but just shifting the partial product, and that a string of ones running from binary weight 2^r to weight 2^s represents a multiplier of $2^{s+1} - 2^r$. For example, if the multiplier = 00001110, then r = 1 and s = 3 and $2^4 - 2^1 = 14$. Note that whereas our previously-described "add-and-shift" algorithm requires three additions for this example, Booth's Algorithm requires only two operations: an addition at weight 2^{s+1} and a subtraction at weight 2^r.

So, when a multiplier is being processed in right-to-left (least-significant bit to most-significant bit) order, Booth's Algorithm boils down to this:

Subtract the multiplicand from the partial product when you find the

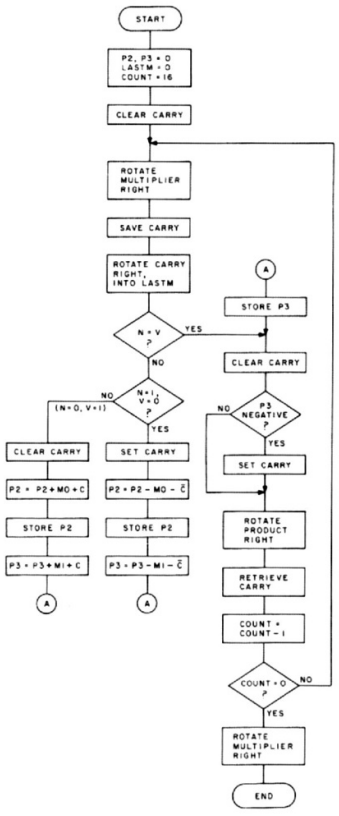

Fig. 5. A double-precision signed multiplication algorithm.

```
This subroutine multiplies a 16-bit signed multiplicand (MPDN,
MPND+1) by a 16-bit signed multiplier (MPLR, MPLR+1), and
returns the 32-bit product in locations PROD (low byte), PROD+1,
PROD+2 and PROD+3 (high byte).

0000                        MPLR=$20      Multiplier
0000                        MPND=$22      Multiplicand
0000                        PROD=$24      Product
0000                        LASTM *=*+1   Previous multiplier bit
0000                        *=$400
0400 A9 00      MLT16S      LDA  #0       Clear P2 and P3 of product
0402 85 26                  STA  PROD+2
0404 85 27                  STA  PROD+3
0406 85 00                  STA  LASTM    Clear LASTM
0408 A2 10                  LDX  #16      Multiplier bit count = 16
0408 18                     CLC
040B 20 46 04   NXTBT       JSR  RMPLR    Go get next multiplier bit
040E 08                     PHP            and save it (as Carry)
040F 66 00                  ROR  LASTM     and put it in LASTM
0411 24 00                  BIT  LASTM    This bit (N) = last bit (V)?
0413 10 10                  BPL  CHKPOS
0415 70 1D                  BVS  ALIGN
0417 38                     SEC           No. N = 1 and V = 0, so
0418 A5 26                  LDA  PROD+2    subtract multiplicand
041A E5 22                  SBC  MPND
041C 85 26                  STA  PROD+2
041E A5 27                  LDA  PROD+3
0420 E5 23                  SBC  MPND+1
0422 4C 32 04               JMP  SP3
0425 50 0D      CHKPOS      BVC  ALIGN
0427 18                     CLC           No. N = 0 and V = 1, so
0428 A5 26                  LDA  PROD+2    add multiplicand
042A 65 22                  ADC  MPND
042C 85 26                  STA  PROD+2
042E A5 27                  LDA  PROD+3
0430 65 23                  ADC  MPND+1
0432 85 27      SP3         STA  PROD+3   Store product MSBY (P3)
0434 18         ALIGN       CLC
0435 24 27                  BIT  PROD+3   P3 negative?
0437 10 01                  BPL  RPROD    No.  Continue with Carry = 0
0439 38                     SEC           Yes.  Set Carry = 1
043A 66 27      RPROD       ROR  PROD+3   Rotate product right
043C 66 26                  ROR  PROD+2
043E 66 25                  ROR  PROD+1
0440 66 24                  ROR  PROD
0442 28                     PLP           Retrieve Carry from Stack
0443 CA                     DEX           Decrement bit count
0444 D0 C5                  BNE  NXTBT    Loop until 16 bits are done
0446 66 21      RMPLR       ROR  MPLR+1   Rotate multiplier right
0448 66 20                  ROR  MPLR
044A 60                     RTS
```

Example 5. A double-precision signed multiplication subroutine.

first 1 in a string of 1's, *add* the multiplicand to the partial product when you find the first 0 in a string of 0's, and do nothing when the bit is identical to the previous multiplier bit.

Mathematically, the basic algorithm as developed by Booth is as follows: y_i is the i-th most significant bit of an n-bit multiplier. y_0 is the least-significant bit and y_{n-1} is the most-significant (sign) bit. So that we can perform a comparison on y_0, we will also define an imaginary less-than-least-significant bit, y_{-1}, and give it a value of 0. X is the multiplicand. Starting with i = 0, bit pairs y_i and y_{i-1} are compared:

1) If $y_i = y_{i-1}$, do nothing.

2) If $y_i = 1$ and $y_{i-1} = 0$, subtract X (the multiplicand) from the partial product.

3) If $y_i = 0$ and $y_{i-1} = 1$, add X to the partial product.

As with our unsigned "add-and-shift" multiplications, once the partial product has received the contribution of a particular multiplier bit, it must be right-shifted one bit position. Here, since two's complement numbers are being multiplied, *the partial product's sign must be preserved during the shift*. In other words, the most-significant bit of the partial product must have the same sense after the shift as it did before the shift.

Signed Multiplication Subroutines

With this groundwork, let's see how Booth's Algorithm can be applied to a couple of "real" signed multiplication subroutines.

Fig. 5 is a flowchart for a double-precision (16-bit by 16-bit) signed multiplication subroutine, based on Booth's Algorithm. The initialization box of this flowchart includes a new parameter, LASTM, which is a memory location that will hold the value

Example 6. A multiprecision signed multiplication subroutine.

```
This subroutine multiplies two variable-length, signed integers:
an N-byte multiplier, starting at location MPLR, and an M-byte
multiplicand, starting at location MPND.  The product will be
returned in memory starting at location PROD.  The A, X and Y
registers are affected by the subroutine.

0000                    N=$20          Multiplier length
0000                    M=$21          Multiplicand length
0000                    MPLR=$30       Multiplier location
0000                    MPND=$40       Multiplicand location
0000                    PROD=$50       Product location
0000                    LASTM *=*+1    Previous multiplier bit
0000                    PINDX *=*+1    Product index
0000                    MBIT *=*+1     Multiplier bit count
0000                    *=$600
0600 18          MMPYS  CLC            Calculate product index
0601 A5 20              LDA  N          (N+M-1)
0603 AA                 TAX
0604 65 21              ADC  M
0606 A8                 TAY             and save it in Y
0607 88                 DEY
0608 84 01              STY  PINDX      and in PINDX
060A A9 00              LDA  #0
060C 85 00              STA  LASTM     Clear LASTM and
060E 99 50 00  CLRP     STA  PROD,Y     high M bytes of product
0611 88                 DEY
0612 C4 20              CPY  N
0614 B0 F8              BCS  CLRP
0616 8A                 TXA            Calculate multiplier bit count
0617 0A                 ASL  A
0618 0A                 ASL  A
0619 0A                 ASL  A
061A 85 02              STA  MBIT       and save it in MBIT
061C 18                 CLC
061D 20 79 06  NXTBT    JSR  RMPLR    Get next multiplier bit
0620 08                 PHP             and save it (as Carry)
0621 66 00              ROR  LASTM      and in high bit of LASTM
0623 24 00              BIT  LASTM    This bit (N) = last bit (V)?
0625 10 20              BPL  CHKPOS
0627 70 3E              BVS  ALIGN
0629 A4 20              LDY  N        No.  N = 1 and V = 0, so
062B A2 00              LDX  #0         subtract multiplicand from
062D 38                 SEC             high M bytes of product
062E B9 50 00  SMPND    LDA  PROD,Y
0631 F5 40              SBC  MPND,X
0633 99 50 00           STA  PROD,Y
0636 C8                 INY
0637 E8                 INX
0638 08                 PHP
0639 C4 01              CPY  PINDX
063B F0 06              BEQ  PULLC
```

More →

See List of Advertisers on page 210

Example 6 continued.

```
063D 90 04            BCC  PULLC
063F 28               PLP
0640 4C 67 06         JMP  ALIGN
0643 28        PULLC  PLP
0644 4C 2E 06         JMP  SMPND
0647 50 1E     CHKPOS BVC  ALIGN
0649 A4 20            LDY  N          No.  N = 0 and V = 1, so
064B A2 00            LDX  #0          add multiplicand to
064D 18               CLC              high M bytes of product
064E B9 50 00  AMPND  LDA  PROD,Y
0651 75 40            ADC  MPND,X
0653 99 50 00         STA  PROD,Y
0656 C8               INY
0657 E8               INX
0658 08               PHP
0659 C4 01            CPY  PINDX
065B F0 06            BEQ  PULLC1
065D 90 04            BCC  PULLC1
065F 28               PLP
0660 4C 67 06         JMP  ALIGN
0663 28        PULLC1 PLP
0664 4C 4E 06         JMP  AMPND
0667 18        ALIGN  CLC
0668 A6 01            LDX  PINDX
066A B5 50            LDA  PROD,X   Partial product negative?
066C 10 01            BPL  RPROD    No.  Continue with Carry = 0
066E 38               SEC           Yes.  Set Carry = 1
066F 76 50     RPROD  ROR  PROD,X   Rotate product right
0671 CA               DEX
0672 10 FB            BPL  RPROD
0674 28               PLP           Retrieve Carry from stack
0675 C6 02            DEC  MBIT     Multiplier fully processed?
0677 D0 A4            BNE  NXTBT    No.  Loop for next bit
0679 A6 20     RMPLR  LDX  N        Yes.  Rotate multiplier right
067B CA               DEX
067C 76 30     RBYTE  ROR  MPLR,X
067E CA               DEX
067F 10 FB            BPL  RBYTE
0681 60               RTS
```

of the current multiplier bit in bit 7 and the value of the previous multiplier bit in bit 6. These particular bits were selected because they're readily accessible with the 6502's BIT instruction, which puts bits 7 and 6 into the status register's N (Negative) and V (Overflow) flags, respectively.

Example 5 shows a subroutine that follows Fig. 5's flowchart to multiply two 16-bit signed numbers. Similarly, Example 6 shows a subroutine that multiplies two multiprecision signed numbers, an N-byte multiplier and an M-byte multiplicand. Note that Examples 5 and 6 represent the signed multiplication counterparts of the unsigned multiplication subroutines in Examples 3 and 4, respectively. And, incidentally, there's no reason you could not use the subroutines in Examples 5 and 6 to multiply unsigned numbers as well as signed numbers. Booth's Algorithm is applicable to either type of data, and if your multiplier contains long strings of 1's, the longer Booth's Algorithm program will probably perform an unsigned multiplication faster than the "add-and-shift" algorithm program.■