High–Speed Integer Multiplies and Divides

Donald A. Branson
St. Louis, Missouri

. . .If your processor has multiply/divide commands, no problem. However, if it does not. . .

If you have written any amount of assembler code, you have probably encountered a situation where you wanted to do a multiplication or a division. If your processor has multiply/divide commands, no problem. However, if it does not (as in the case of the 6502–series CPU in Commodore's 8–bit computers), you may have done a multiply by adding an operand to itself, decrementing the operand at the same time until it reaches zero.

This is probably the most straightforward way to do a multiply, but a binary multiply can be done the same way a person multiplies a decimal number by hand. This way is as easy to do and, in all but a few cases, is quite a bit faster.

Multiplication

First, let's look at the way a decimal multiply is done, in order to draw attention to some key elements in any multiply. The procedure is expanded on the right to show more specifically what is happening here.

```
  35      1)   5 * 5   =   25
x 15      2)   5 * 30  =  150
 175      3)  10 * 5   =   50
 350      4)  10 * 30  =  300
 525                      525
```

It can be seen that 35 has not been added to itself 15 times or 15 added to itself 35 times. Doing it that way would take longer and, if you're doing it by hand, more prone to error.

What has been done is to multiply each digit in the first number against all the digits in the other number.

This same method can be applied to binary multiplication.

In order to look at one binary digit (bit) at a time, a combination of the LSR and ROR operations is used to shift the bit into the carry bit, where it can be examined with a BCC or BCS operation. In this way, one digit can be multiplied at a time. If this were a decimal multiply, we would have to have a table of results for each possible combination of one digit multiplies, like this:

```
1 * 1 = 1
1 * 2 = 2
  .
  .
2 * 1 = 2
2 * 2 = 4
  .
  .
```

When we multiply decimal, we have this table in our heads. However, the table is simplified when we do binary multiplication. If fact, it is simplified to the point that the result of a multiply can be one of two things: a zero bit or a one bit.

A result table does not need to be kept for binary multiplication. The table is effectively implied in the operands.

For example:

```
15 = $0F = %0000 1111
35 = $23 = %0010 0011

    00001111    M1
x   00100011    M2
    00001111
   00001111
  00000000
 00000000
00000000
 00001111
 00000000
00000000
000001000001101    R
 ($020D = 525)
```

Now to look at the process in a way that can be easily represented in a computer. We'll call the multiplicand M1, the multiplier M2, and the field where the result is left R, like this:

```
M1: 0000 0000 0000 1111
M2: 0000 0000 0010 0011
R:  0000 0000 0000 0000
```

Since the result will be sixteen bits (525 > 255), all the fields are represented as sixteen bits.

First, M1 is shifted right, so that each bit can be looked at from right to left as in the decimal multiply shown before. As you can see in the example of binary multiplication above, the digit can be either one or zero, and the result of the single-digit multiply is either zero, or a duplicate of M2, properly shifted to the left. After each digit in M1 is shifted into the carry bit, M2 is added to R if the carry is one, then M2 is shifted left, in preparation for the next digit.

Here, 'SR' stands for shift right, 'SL' for shift left.

```
pass                                          carry

1-      SR M1:  0000 0000 0000 0111   1
     R=R+M2:    0000 0000 0010 0011
        SL M2:  0000 0000 0100 0110

2-      SR M1:  0000 0000 0000 0011   1
     R=R+M2:    0000 0000 0110 1001
        SL M2:  0000 0000 1000 1100

3-      SR M1:  0000 0000 0000 0001   1
     R=R+M2:    0000 0000 1111 0101
        SL M2:  0000 0001 0001 1000

4-      SR M1:  0000 0000 0000 0000   1
     R=R+M2:    0000 0010 0000 1101
        SL M2:  0000 0010 0011 0000
```

Note that we can stop here, because M1 is zero, and the carry bit will no longer be set as a result of shifting M1, and therefore M2 will not be added to R any more times. At this point, the result of the multiply is left in R.

Here is how this process looks in assembler:

```
MLOOP EQU *          ;in PAL- MLOOP=*
      LSR  M1        ;SR M1
      ROR  M1+1      ;the carry from M1 → M1+1
      BCC  SHIFT     ;add on carry set
      CLC            ;R=R+M2
      LDA  M2+1
      ADC  R+1
      STA  R+1
      LDA  M2
      ADC  R
      STA  R
SHIFT EQU *
      ASL  M2+1      ;SL M2
      ROL  M2
      LDA  M1+1
      BNE  MLOOP
      LDA  M1        ;if M1 and M1+1=0
      BNE  MLOOP     ;when we are done
      RTS
M1    HEX  0023      ;or .BYTE $00,$23
M2    HEX  000F      ;or .BYTE $00,$0F
R     HEX  0000      ;or .BYTE $00,$00
```

Division

Now we'll take a look at division. Binary division can also be done by mimicking the way decimal division is done. Let's look at an example of how to compute 43/5, giving a result of 8 remainder 3. Here's how it would be done by hand, in decimal.

```
      8   r 3
   ------
5 ) 43
    40
    --
     3

1) 8 * 5 = 40
2) 43 - 40 = 3
```

M1 = 5 = 0000 0101 M2 = 43 = 0010 1011
R1 = 0000 0000 R2 = 0000 0000

In the case of division, two result fields are needed, one for the quotient, the other for the remainder. Specifically:

M1 / M2 = R1 rem R2

M2 will be shifted left into R2, and R2 will then be compared to M1. If R2 is greater than or equal to M1, then M1 will fit into R2. The result of a binary divide can be only a zero or a one, again saving us the trouble of having a result table, as is necessary for decimal division. If M1 will fit into R2, then we subtract M1 from R2, leaving the result in R2. Finally, R1 is shifted left, with the carry bit from the compare going into the least significant bit.

```
pass

1-  SL M2 into R2   R2 = 0000 0000   M2 = 0101 0110
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0000

2-  SL M2 into R2   R2 = 0000 0000   M2 = 1010 1100
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0000

3-  SL M2 into R2   R2 = 0000 0001   M2 = 0101 1000
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0000

4-  SL M2 into R2   R2 = 0000 0010   M2 = 1011 0000
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0000

5-  SL M2 into R2   R2 = 0000 0101   M2 = 0110 0000
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0001
    R2 = R2-M1      R2 = 0000 0000

6-  SL M2 into R2   R2 = 0000 0000   M2 = 1100 0000
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0010

7-  SL M2 into R2   R2 = 0000 0001   M2 = 1000 0000
    CMP R2-M1 (carry clear)
    SL R1           R1 = 0000 0100
```

(. . .cont'd on page 45)

To signal that no secondary address is called for, put 255 (hex FF) into the Y register.

Both of the above calls should be followed soon by a call to OPEN ($FFC0). Because here's the catch: the above setup does NOT work as expected if you're going to call LOAD ($FFD5).

The documentation doesn't tell you this. . . but if you are going to make a call to LOAD, your prior call to SETLFS does **not** set a secondary address. Instead, it sets a flag. A value of binary zero allows the program to relocate as it is loaded; any other (non-zero) value triggers a load to a fixed address, with no relocation. This last is what is most often wanted.

And in the case of LOAD, you do NOT want to add 96. If your objective is relocation, the only value that works in the Y register is zero. . . add 96 and it isn't zero any more, and you will NOT relocate.

I don't use calls to LOAD much. In most cases, it seems to me to be as easy to OPEN and read the bytes myself, stacking them away under program control after examining them. But if you wish to use LOAD, go ahead. . . just remember that the SETLFS secondary address is really not a secondary address at all.

One other related subject, to do with relative files. If you want to position to a particular record in a file, you must use the secondary address of that file in the "P" command. Again, I recommend that you use an added 96 as part of the value. Thus, if you have coded:

```
OPEN 15,8,15
OPEN 1,2,3,"0:RELDATA,L,"+CHR$(75)
```

. . .opening a file, then to position to record number five, I'd suggest:

```
PRINT#15,"P"+CHR$(99)+CHR$(5)+CHR$(0)+CHR$(1)
```

The secondary address for the file is three, as shown in the OPEN 1,2,3. . . statement. But I'd suggest that your position command add 96 as shown to specify a secondary address of 99.

Do you need to carefully do all this, every time? I honestly don't know for sure. Sometimes everything seems to work when you don't worry about the extra bits. But when you carefully trace Basic code (especially the 4.0 and 7.0 versions), you see that Commodore always puts these extra values in. Maybe they know something.

And with the wide variety of disk drives and programs, you can never be sure when you might come up with a combination that needs those extra bits to be exactly right. I don't know about you. . . but my data files are too important for me to take any unnecessary chances.

(Cont'd from page 43)

```
8-  SL M2 into R2   R2 = 0000 0011   M2 = 0000 0000
    CMP R2-M1 (carry clear)
    SL R1          R1 = 0000 1000
```

Now we can stop, because all eight bits of M2 have been processed.

Here is how this process looks in assembler. To compare R2 and M1, a subtract is used. The result of this subtract is saved in registers X and Y, so that the subtract does not have to be repeated, thus saving a few cycles.

```
DLOOP    EQU *
         ASL  M2+1
         ROL  M2
         ROL  M2+1
         ROL  R2
         SEC
         LDA  R2+1
         SBC  M1+1
         TAX             ;save low byte
         LDA  R2
         SBC  M1
         TAY             ;save high byte
         BCC  SKIPSAVE
         STX  R2+1       ;store saved bytes
         STY  R2         ;in R2
SKIPSAVE EQU *
         ROL  R1+1       ;shift carry from
         ROL  R1         ;subtract into R1
         LDA  M2+1
         BNE  DLOOP
         LDA  M2
         BNE  DLOOP
         RTS
M1       HEX  0005
M2       HEX  002B
R1       HEX  0000
R2       HEX  0000
```

Conclusion

To make these routines even faster, use zero page memory for all the fields. They will also take less memory this way.

Where will these routines come in handy? Well, for starters, here are some ideas.

- Graphics programs, for calculating the position of a dot, plotting a line, or drawing a circle, or drawing a circle.
- Compilers, for calculating memory needed for arrays, or for finding a specific element in an array. Also good for fast integer multiply/divides.
- Calculating prime numbers.

There are many places where multiplies or divides are used, and anywhere one is used, one of these routines can make your code from a few times to hundreds of times faster.