## Introduction

The 8051 is an 8-bit microcontroller. This basically means that each machine language opcode in its instruction set consists of a single 8-bit value. This permits a maximum of 256 instruction codes (of which 255 are actually used in the 8051 instruction set).

The 8051 also works almost exclusively with 8-bit values. The Accumulator is an 8-bit value, as is each register in the Register Banks, the Stack Pointer (SP), and each of the many Special Function Registers (SFRs) that exist in the architecture. In reality, the only values that the 8051 handles that are truly 16-bit values are the Program Counter (PC) that internally indicates the next instruction to be executed, and the Data Pointer (DPTR) which the user program may utilize to access external RAM as well as directly access code memory. Other than these two registers, the 8051 works exclusively with 8-bit values.

For example, the ADD instruction will add two 8-bit values to produce a third 8-bit value. The SUBB instruction subtracts an 8-bit value from another 8-bit value and produces a third 8-bit value. The MUL instruction will multiply two 8-bit values and produce a 16-bit value.

> **Programming Tip**: It could be said that the MUL instruction is a 16-bit math instruction since it produces a 16-bit answer. However, its inputs are only 8-bit. The result is 16-bits out of necessity since any multiplication with two operands greater than the number 16 will produce a 16-bit result. Thus, for the MUL operation to have any value at all it was absolutely necessary to produce a 16-bit result.

As we can see, the 8051 provides us with a number of instructions aimed at performing mathematical calculations. Unfortunately, they are all work with 8-bit input values--and we often find ourselves working with values that simply cannot be expressed in 8-bits.

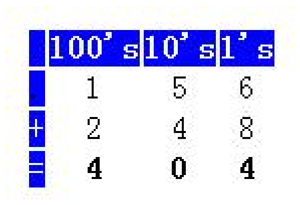
This tutorial will discuss techniques that allow the 8051 developer to work with 16-bit values in the 8051's 8-bit architecture. While we will only discuss 16-bit mathematics, the techniques can be extended to any number of bits (24-bit, 32-bit, 64-bit, etc.). It's just a matter of expanding the code to support the additional bytes. The algorithms remain the same.

These tutorials will explain how to perform 16-bit addition, subtraction, and multiplication with the 8051. For the time being, 16-bit division is outside the scope of this tutorial.

> **Programming Tip**: Compared to addition, subtraction, and multiplication, division is a relatively complicated process. For the time being 16-bit division will not be discussed because the author has not had a need to develop such routines, nor an opportunity to analyze the process in performing the calculation. If you have developed a routine that allows a 16-bit value to be divided by another 16-bit value and would like to contribute the code to 8052.com, along with a tutorial similar to those found in these sections, please contact us..

## How did we learn math in primary school?

Before jumping into multibyte mathematics in machine language, let's quickly review the mathematics we learned as children. For example, we learned to add two numbers, say 156 + 248, as follows:



| | 100's | 10's | 1's |
|---|---|---|---|
| | 1 | 5 | 6 |
| + | 2 | 4 | 8 |
| = | **4** | **0** | **4** |

How do we calculate the above? We start in the 1's column, adding 6 + 8 = 14. Since 14 can't fit in a single column, we leave 4 in the 1's column and carry the 1 to the 10's column. We then add 5 + 4 = 9, add the 1 we carried, to get 10. Again, 10 doesn't fit in a single column. So we leave the 0 in the 10's column and carry the 1 to the 100's column. Finally, we add 1 + 2 = 3, add the 1 we carried to get 4, which is our final answer in the 100's column. The final answer, thus, is 404.

It is important to remember this when working with multibyte math, because the process is going to be the same. Let's start by doing 16-bit addition.

## *16-Bit Addition*

16-bit addition is the addition of two 16-values. First, we must recognize that the addition of two 16-bit values will result in a value that is, at most, 17 bits long. Why is this so? The largest value that can fit in 16-bits is 256 * 256 - 1 = 65,535. If we add 65,535 + 65,535, we get the result of 131,070. This value fits in 17 bits. Thus when adding two 16-bit values, we will get a 17-bit value. Since the 8051 works with 8-bit values, we will use the following statement: *"Adding two 16-bit values results in a 24-bit value"*. Of course, 7 of the highest 8 bits will never be used--but we will have our entire answer in 3 bytes. Also keep in mind that we will be working with unsigned integers.

> ***Programming Tip****: Another option, instead of using 3 full bytes for the answer, is to use 2 bytes (16-bits) for the answer, and the carry bit ( C ), to hold the 17th bit. This is perfectly acceptable, and probably even preferred. The more advanced programmer will understand and recognize this option, and be able to make use of it. However, since this is an introduction to 16-bit mathematics it is our goal that the answer produced by the routines be in a form that is easy for the reader to utilize, once calculated. It is our belief that this is best achieved by leaving the answer fully expressed in 3 8-bit values.*

*Let's consider adding the following two decimal values: **6724** + **8923**. The answer is, of course, 15647. How do we go about adding these values with the 8051? The first step is to always work with hexadecimal values. Simlply convert the two values you wish to add to hexadecimal. In this case, that is equivalent to the following hexadecimal addition: **1A44** + **22DB**.*

*How do we add thes two numbers? Let's use the exact same method we used in primary school, and in the previous section:*

| | 256's | 1's |
|---|---|---|
| | 1A | 44 |
| + | 22 | DB |
| = | **3D** | **1F** |

First, notice the difference. We are no longer working with a 1's, 10's, and 100's columns. We are just working with two columns: The 1's column and the 256's column. In familiar computer terms: We're working with the low byte (the 1's column) and the high byte (the 256's column). However, the process is exactly the same.

First we add the values in the 1's column (low byte): 44 + DB = 11F. Only a 2-digit hexadecimal value can fit in a single column, so we leave the 1F in the low-byte column, and carry the 1 to the high-byte column. We now add the high bytes: 1A + 22 = 3C, plus the 1 we carried from the low-byte column. We arrive at the value 3D.

Thus, our completed answer is 3D1F. If we convert 3D1F back to decimal, we arrive at the answer 15647. This matches with the original addition we did in decimal. The process works. Thus the only challenge is to code the above process into 8051 assembly language. As it turns out, this is incredibly easy.

We'll use the following table to explain how we're going to do the addition:

| | 65536's | 256's | 1's |
|---|---|---|---|
| | | R6 | R7 |
| + | | R4 | R5 |
| = | **R1** | **R2** | **R3** |

Since we're adding 16-bit values, each value requires two 8-bit registers. Essentially, the first value to add will be held in R6 and R7 (the high byte in R6 and the low byte in R7) while the second value to add will be held in R4 and R5 (the high byte in R4 and the low byte in R5). We will leave our answer in R1, R2, and R3.

> **Programming Tip:** Remember that we mentioned above that the sum of two 16-bit values is a 17-bit value. In this case, we'll using 24-bits (R1, R2, and R3) for our answer, even though we'll never use more than 1 bit of R1.

Let's review the steps involved in adding the values above:

1. Add the low bytes R7 and R5, leave the answer in R3.
2. Add the high bytes R6 and R4, adding any carry from step 1, and leave the answer in R2.
3. Put any carry from step 2 in the final byte, R1.

We'll now convert the above process to assembly language, step by step.

Step 1: Add the low bytes R7 and R5, leave the answer in R3.

```
MOV A,R7   ;Move the low-byte into the accumulator
ADD A,R5   ;Add the second low-byte to the accumulator
MOV R3,A   ;Move the answer to the low-byte of the result
```

Step 2: Add the R6 and R4, add carry, leave the answer in R2.

```
MOV A,R6   ;Move the high-byte into the accumulator
ADDC A,R4  ;Add the second high-byte to the accumulator, plus carry.
MOV R2,A   ;Move the answer to the high-byte of the result
```

Step 3: Put any carry from step 2 in the final byte, R1.

```
MOV A,#00h   ;By default, the highest byte will be zero.
ADDC A,#00h  ;Add zero, plus carry from step 2.
MOV R1,A     ;Move the answer to the highest byte of  the result
```

That's it! Combining the code from the three steps, we come up with the following subroutine:

```
ADD16_16:
    ;Step 1 of the process
    MOV A,R7      ;Move the low-byte into the accumulator
    ADD A,R5      ;Add the second low-byte to the accumulator
    MOV R3,A      ;Move the answer to the low-byte of the result

    ;Step 2 of the process
    MOV A,R6      ;Move the high-byte into the accumulator
    ADDC A,R4     ;Add the second high-byte to the accumulator, plus carry.
    MOV R2,A      ;Move the answer to the high-byte of the result

    ;Step 3 of the process
    MOV A,#00h    ;By default, the highest byte will be zero.
    ADDC A,#00h   ;Add zero, plus carry from step 2.
    MOV MOV R1,A  ;Move the answer to the highest byte of  the result

    ;Return - answer now resides in R1, R2, and R3.
    RET
```

And to call our routine to add the two values we used in the example above, we'd use the code:

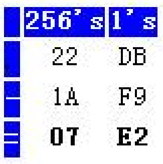
```
   ;Load the first value into R6 and R7
   MOV R6,#1Ah
   MOV R7,#44h

   ;Load the second value into R4 and R5
   MOV R4,#22h
   MOV R5,#0DBh

   ;Call the 16-bit addition routine
   LCALL ADD16_16
```

## *16-Bit Subtraction*

16-bit subtraction is the subtraction of one 16-bit value from another. A subtraction of this nature results in another 16-bit value. Why? The number 65535 is a 16-bit value. If we subtract 1 from it, we have 65534 which is also a 16-bit value. Thus any 16-bit subtraction will result in another 16-bit value.

Let's consider the subtraction of the following two decimal values: 8923 - 6905. The answer is 2018. How do we go about subtracting these values with the 8051? As is the case with addition, the first step is to convert the expression to hexadecimal. The above decimal subtraction is equivalent to the following hexadecimal subtraction: 22DB - 1AF9.

Again, we'll go back to the way we learned it in primary school:



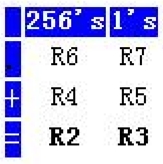
| | 256's | 1's |
|---|---|---|
| | 22 | DB |
| - | 1A | F9 |
| = | **07** | **E2** |

First we subtract the second value in the 1's column (low byte): DB - F9. Since F9 is greater than DB, we need to "borrow" from the 256's column. Thus we actually perform the subtraction 1DB - F9 = E2. The value E2 is what we leave in the 1's column.

Now we must perform the subtraction 22 - 1A. However, we must remember that we "borrowed" 1 from the 256's column, so we must subtract an additional 1. So the subtraction for the 256's column becomes 22 - 1A - 1 = 7, which is the value we leave in the 256's column.

Thus our final answer is 07E2. If we conver this back to decimal, we get the value 2018, which coincides with the math we originally did in decimal.

As we did with addition, we'll use a small table to help us conver the above process to 8051 assembly language:



| | 256's | 1's |
|---|---|---|
| | R6 | R7 |
| + | R4 | R5 |
| = | **R2** | **R3** |

Since we're subtracting 16-bit values, each value requires two 8-bit registers. Essentially, the value to be subtracted from will be held in R6 and R7 (the high byte in R6 and the low byte in R7) while the value to be subtracted will be held in R4 and R5 (the high byte in R4 and the low byte in R5). We will leave our answer in R2, and R3.

Let's review the steps involved in subtracting the values above:

1. Subtract the low bytes R5 from R7, leave the answer in R3.
2. Subtract the high byte R4 from R6, less any borrow, and leave the answer in R2.

We'll now convert the above process to assembly language, step by step.

**Step 1: Subtract the low bytes R5 from R7, leave the answer in R3.**

```
MOV A,R7    ;Move the low-byte into the accumulator
CLR C       ;Always clear carry before first subtraction
SUBB A,R5   ;Subtract the second low-byte from the accumulator
MOV R3,A    ;Move the answer to the low-byte of the result
```

**Step 2: Subtract the high byte R4 from R6, less any borrow, and leave the answer in R2.**

```
MOV A,R6    ;Move the high-byte into the accumulator
SUBB A,R4   ;Subtract the second high-byte from the accumulator
MOV R2,A    ;Move the answer to the low-byte of the result
```

Programming Tip: The SUBB instruction always subtracts the second value in the instruction from the first, less any carry. While there are two versions of the ADD instruction (ADD and ADDC), one of which ignores the carry bit, there is no such distinction with the SUBB instruction. This means before you perform the first subtraction, you must always be sure to clear the carry bit. Otherwise, if the carry bit happens to be set you'll end up subtracting it from your first column value -- which would be incorrect.

Combining the code from the two steps above, we come up with the following subroutine:

```
SUBB16_16:
  ;Step 1 of the process
  MOV A,R7   ;Move the low-byte into the accumulator
  CLR C      ;Always clear carry before first subtraction
  SUBB A,R5  ;Subtract the second low-byte from the accumulator
  MOV R3,A   ;Move the answer to the low-byte of the result

  ;Step 2 of the process
  MOV A,R6   ;Move the high-byte into the accumulator
  SUBB A,R4  ;Subtract the second high-byte from the accumulator
  MOV R2,A   ;Move the answer to the low-byte of the result

  ;Return - answer now resides in R2, and R3.
  RET
```

And to call our routine to subtract the two values we used in the example above, we'd use the code:

```
   ;Load the first value into R6 and R7
   MOV R6,#22h
   MOV R7,#0DBh

   ;Load the second value into R4 and R5
   MOV R4,#1Ah
   MOV R5,#0F9h

   ;Call the 16-bit subtraction routine
   LCALL SUBB16_16
```

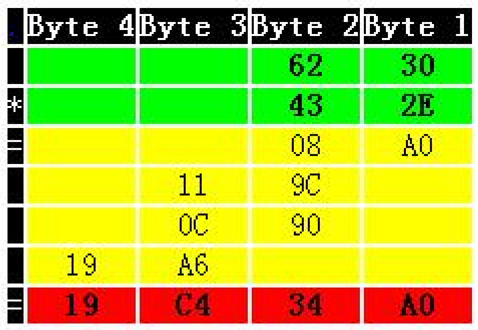
## *16-Bit Multiplication*

16-bit multiplication is the multiplication of two 16-bit value from another. Such a multiplication results in a 32-bit value.

> **Programming Tip:** In fact, any multiplication results in an answer which is the sum of the bits in the two multiplicands. For example, multiplying an 8-bit value by a 16-bit value results in a 24-bit value (8 + 16). A 16-bit value multiplied by another 16-bit value results in a 32-bit value (16 + 16), etc.

For the sake of example, let's multiply 25,136 by 17,198. The answer is 432,288,928. As with both addition and subtraction, let's first convert the expression into hexadecimal: 6230h x 432Eh.

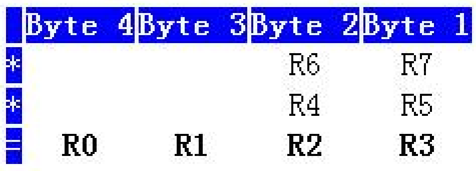
Once again, let's arrange the numbers in columns as we did in primary school to multiply numbers, although now the grid becomes more complicated. The green section represents the original two values. The yellow section represents the intermediate calculations obtained by multipying each byte of the original values. The red section of the grid indicates our final answer, obtained by summing the columns in the yellow area.



| | Byte 4 | Byte 3 | Byte 2 | Byte 1 |
|---|---|---|---|---|
| | | | 62 | 30 |
| * | | | 43 | 2E |
| = | | | 08 | A0 |
| | | 11 | 9C | |
| | | 0C | 90 | |
| | 19 | A6 | | |
| = | 19 | C4 | 34 | A0 |

Remember how we did this in elementary school? First we multiply 2Eh by 30h (byte 1 of both numbers), and place the result directly below. Then we multiply 2Eh by 62h (byte 1 of the bottom number by byte 2 of the upper number). This result is lined up such that the right-most column ends up in byte 2. Next we multiply 43h by 30h (byte 2 of the bottom number by byte 1 of the top number), again lining up the result so that the right-most column ends up in byte 2. Finally, we multiply 43h by 62h (byte 2 of both numbers) and position the answer such that the right-most column ends up in byte 3. Once we've done the above, we add each column, with appropriate carries, to arrive at the final answer.

Our process in assembly language will be identical. Let's use our now-familiar grid to help us get an idea of what we're doing:



| | Byte 4 | Byte 3 | Byte 2 | Byte 1 |
|---|---|---|---|---|
| * | | | R6 | R7 |
| * | | | R4 | R5 |
| = | **R0** | **R1** | **R2** | **R3** |

Thus our first number will be contained in R6 and R7 while our second number will be held in R4 and R5. The result of our multiplication will end up in R0, R1, R2 and R3. At 8-bits per register, these four registers give us the 32 bits we need to handle the largest possible multiplication. Our process will be the following:

1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.
2. Multiply R5 by R6, adding the 16-bit result to R1 and R2.
3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.
4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.

We'll now convert the above process to assembly language, step by step.

acxc44

**Step 1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.**

```
MOV A,R5 ;Move the R5 into the Accumulator
MOV B,R7 ;Move R7 into B
MUL AB   ;Multiply the two values
MOV R2,B ;Move B (the high-byte) into R2
MOV R3,A ;Move A (the low-byte) into R3
```

**Step 2. Multiply R5 by R6, adding the 16-bit result to R1 and R2.**

```
MOV A,R5     ;Move R5 back into the Accumulator
MOV B,R6     ;Move R6 into B
MUL AB       ;Multiply the two values
ADD A,R2     ;Add the low-byte into the value already in R2
MOV R2,A     ;Move the resulting value back into R2
MOV A,B      ;Move the high-byte into the accumulator
ADDC A,#00h  ;Add zero (plus the carry, if any)
MOV R1,A     ;Move the resulting answer into R1
MOV A,#00h   ;Load the accumulator with  zero
ADDC A,#00h  ;Add zero (plus the carry, if any)
MOV R0,A     ;Move the resulting answer to R0.
```

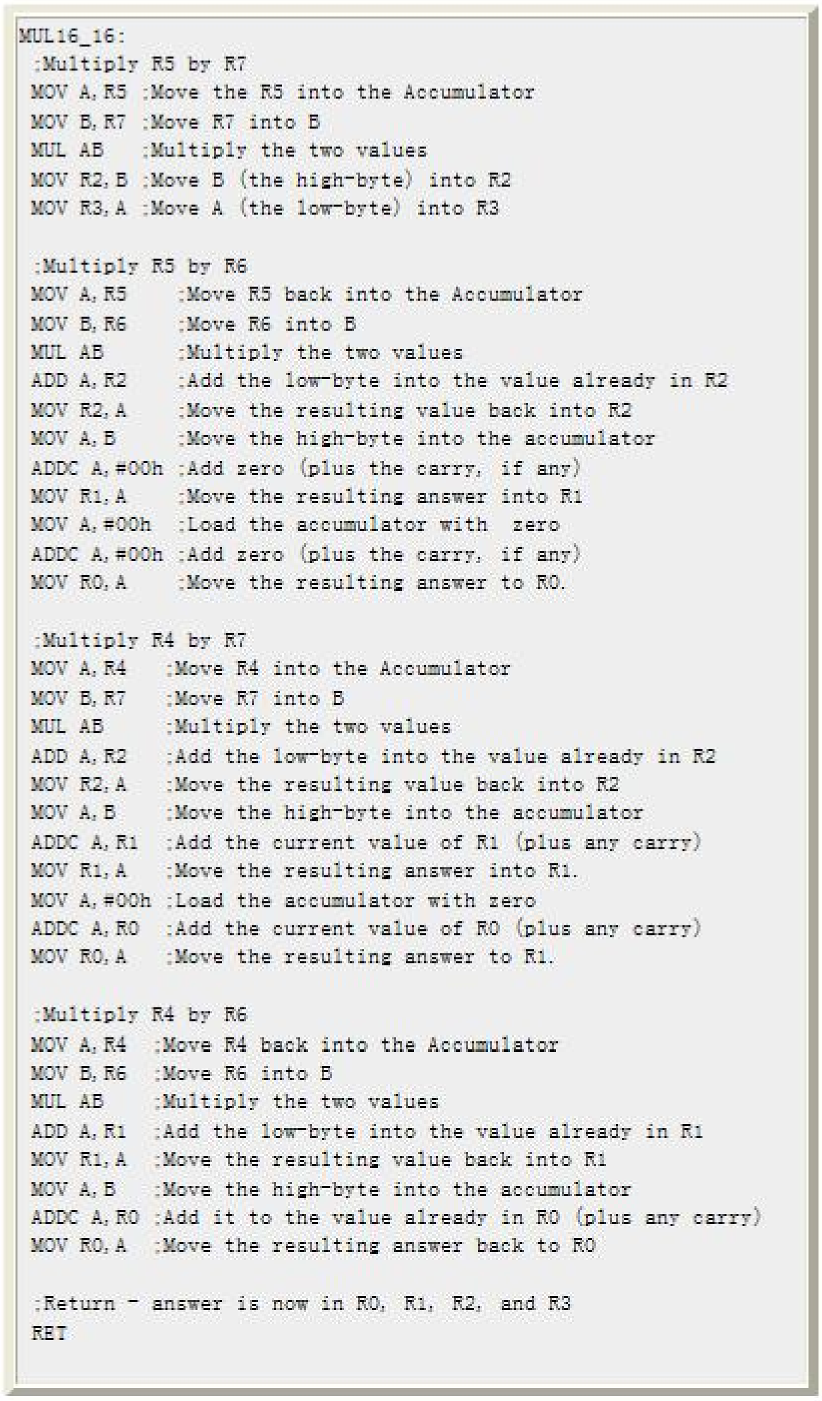
**Step 3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.**

```
MOV A,R4    ;Move R4 into the Accumulator
MOV B,R7    ;Move R7 into B
MUL AB      ;Multiply the two values
ADD A,R2    ;Add the low-byte into the value already in R2
MOV R2,A    ;Move the resulting value back into R2
MOV A,B     ;Move the high-byte into the accumulator
ADDC A,R1   ;Add the current value of R1 (plus any carry)
MOV R1,A    ;Move the resulting answer into R1.
MOV A,#00h  ;Load the accumulator with zero
ADDC A,R0   ;Add the current value of R0 (plus any carry)
MOV R0,A    ;Move the resulting answer to R1.
```

**Step 4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.**

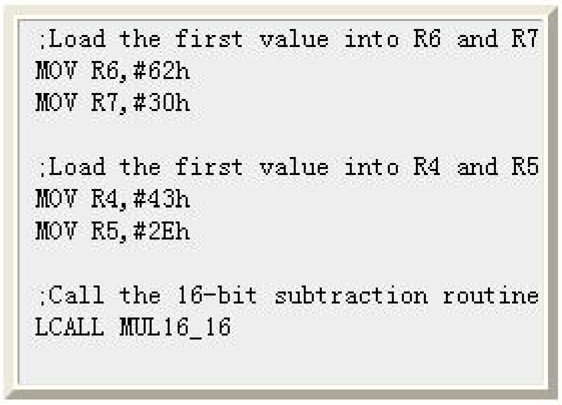
```
MOV A,R4   ;Move R4 back into the Accumulator
MOV B,R6   ;Move R6 into B
MUL AB     ;Multiply the two values
ADD A,R1   ;Add the low-byte into the value already in R1
MOV R1,A   ;Move the resulting value back into R1
MOV A,B    ;Move the high-byte into the accumulator
ADDC A,R0  ;Add it to the value already in R0 (plus any carry)
MOV R0,A   ;Move the resulting answer back to R0
```

Combining the code from the two steps above, we come up with the following subroutine:



```
MUL16_16:
 ;Multiply R5 by R7
 MOV A,R5 ;Move the R5 into the Accumulator
 MOV B,R7 ;Move R7 into B
 MUL AB   ;Multiply the two values
 MOV R2,B ;Move B (the high-byte) into R2
 MOV R3,A ;Move A (the low-byte) into R3

 ;Multiply R5 by R6
 MOV A,R5    ;Move R5 back into the Accumulator
 MOV B,R6    ;Move R6 into B
 MUL AB      ;Multiply the two values
 ADD A,R2    ;Add the low-byte into the value already in R2
 MOV R2,A    ;Move the resulting value back into R2
 MOV A,B     ;Move the high-byte into the accumulator
 ADDC A,#00h ;Add zero (plus the carry, if any)
 MOV R1,A    ;Move the resulting answer into R1
 MOV A,#00h  ;Load the accumulator with  zero
 ADDC A,#00h ;Add zero (plus the carry, if any)
 MOV R0,A    ;Move the resulting answer to R0.

 ;Multiply R4 by R7
 MOV A,R4   ;Move R4 into the Accumulator
 MOV B,R7   ;Move R7 into B
 MUL AB     ;Multiply the two values
 ADD A,R2   ;Add the low-byte into the value already in R2
 MOV R2,A   ;Move the resulting value back into R2
 MOV A,B    ;Move the high-byte into the accumulator
 ADDC A,R1  ;Add the current value of R1 (plus any carry)
 MOV R1,A   ;Move the resulting answer into R1.
 MOV A,#00h ;Load the accumulator with zero
 ADDC A,R0  ;Add the current value of R0 (plus any carry)
 MOV R0,A   ;Move the resulting answer to R1.

 ;Multiply R4 by R6
 MOV A,R4  ;Move R4 back into the Accumulator
 MOV B,R6  ;Move R6 into B
 MUL AB    ;Multiply the two values
 ADD A,R1  ;Add the low-byte into the value already in R1
 MOV R1,A  ;Move the resulting value back into R1
 MOV A,B   ;Move the high-byte into the accumulator
 ADDC A,R0 ;Add it to the value already in R0 (plus any carry)
 MOV R0,A  ;Move the resulting answer back to R0

 ;Return - answer is now in R0, R1, R2, and R3
 RET
```

And to call our routine to multiply the two values we used in the example above, we'd use the code:



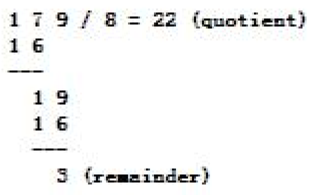
```
;Load the first value into R6 and R7
MOV R6,#62h
MOV R7,#30h

;Load the first value into R4 and R5
MOV R4,#43h
MOV R5,#2Eh

;Call the 16-bit subtraction routine
LCALL MUL16_16
```

## *16-Bit Division*

16-bit division is the division of one 16-bit value by another 16-bit value, returning a 16-bit quotient and a 16-bit remainder. I used r1/r0 for dividend/remainder and r3/r2 for divisor/quotient.

> **Programming Tip:** The number of bits in the quotient and the remainder can never be larger than the number of bits in the original divident. For example, if you are dividing a 16-bit value by a 2-bit value, both the quotient and the remainder must be able to handle a 16-bit result. If you are dividing a 24-bit value by a 16-bit value, the quotient and remainder must both be able to handle a 24-bit result.

So, again, let's remember how we did division in elementary school. For example, 179 divided by 8:

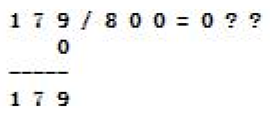


```
1 7 9 / 8 = 22 (quotient)
1 6
---
  1 9
  1 6
  ---
    3 (remainder)
```

It's necessary necessary to follow this same process step by step. There is a 3-digit-dividend, so we expect 3 digits maximum for quotient. We "shift left" the divisor 2 digits (3-1) such that the number of digits in the divisor is the same as the number of digits in the dividend. So we get:

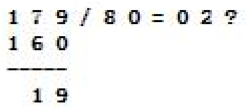
1 7 9 / 8 0 0 = ? ? ?

We divide the two numbers, multiply the result by the divisor and substract this result from the dividend. In this first step 179 can't be divided by 800, so the the result is 0. We subtract 0 from 179 and still have 179:



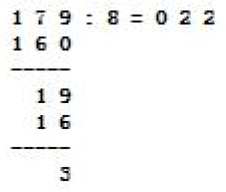
```
179/800=0??
  0
-----
179
```

We then "shift right" the divisor 1 digit and repeat the process. 179 divided by 80 results in an answer of 2. After we subtract 160 (2x80) we are left with a remainder of 19:



```
179/80=02?
160
-----
 19
```

We repeat the process again until the divisor has shifted into its original position:



```
179:8=022
160
-----
 19
 16
-----
  3
```

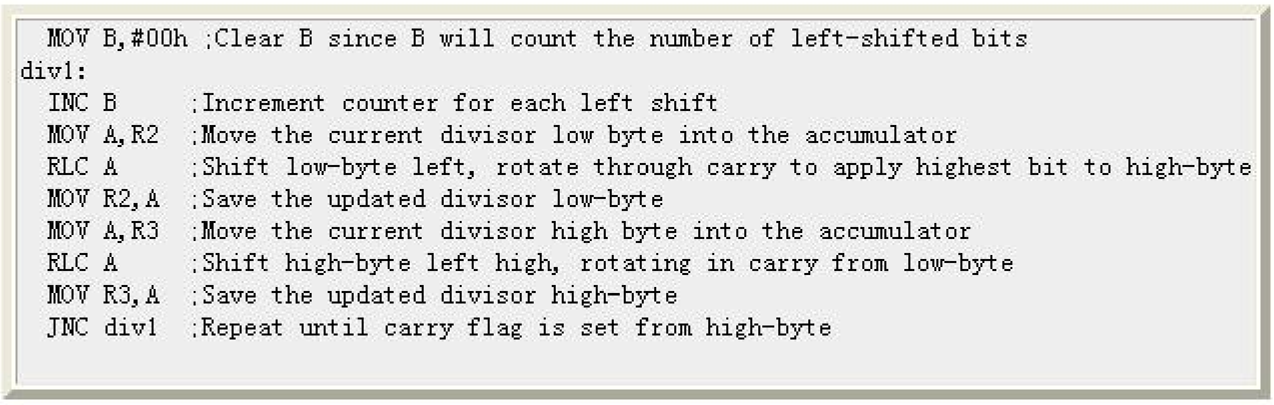
This may have been an unnecessary review of elementary school math, but it is important to remember exactly how the process is performed because we do *exactly* the same with the 8052 in binary system.

In this routine we will place the original dividend into R1 (high-byte) and R0 (low-byte) and the divisor in R3 (high-byte) and R2 (low-byte).

In the case of our example (179 divided by 8), the initial registers would be:

```
R1/R0 00000000 10110011
R3/R2 00000000 00001000
```

**Step 1. Shift left the divisor.**



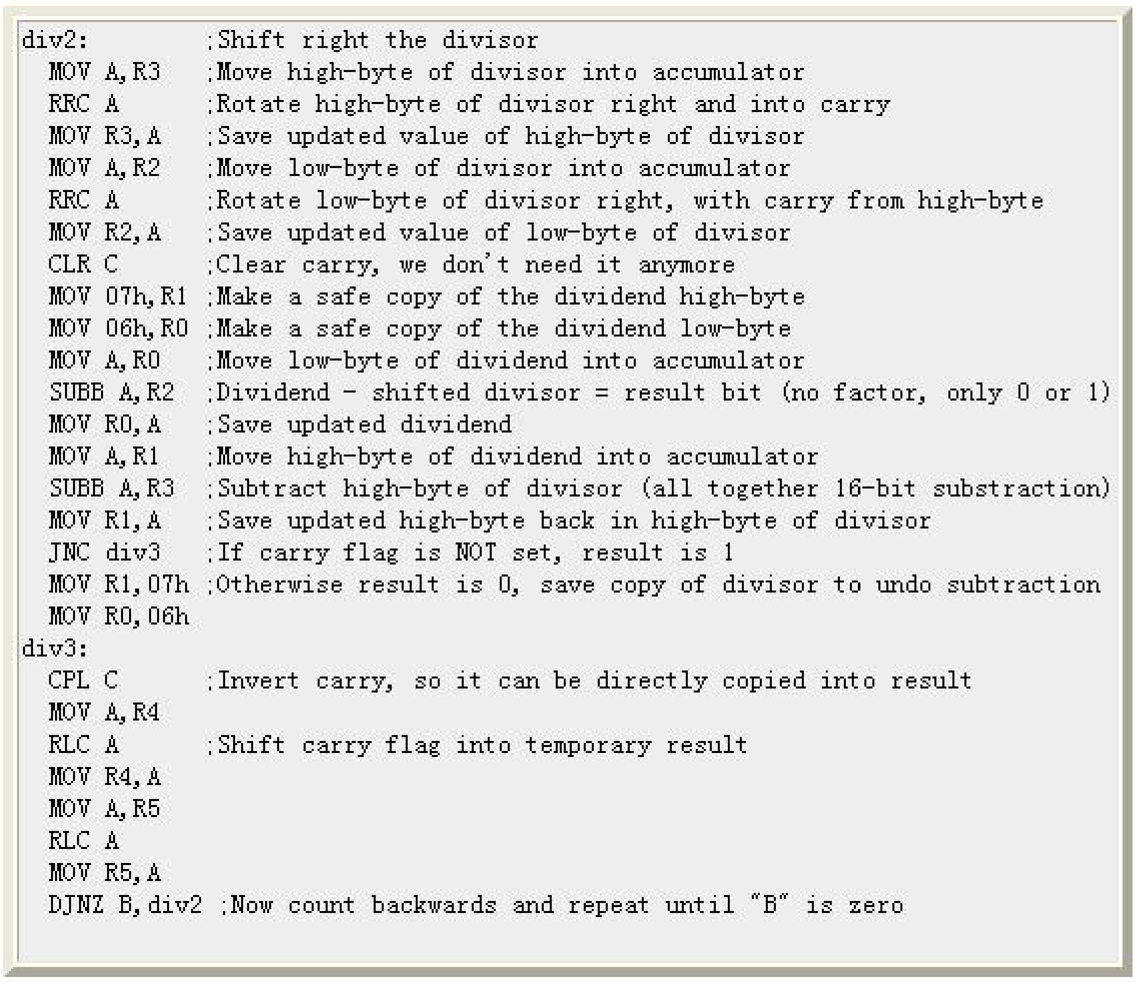
```
  MOV B,#00h ;Clear B since B will count the number of left-shifted bits
div1:
  INC B      ;Increment counter for each left shift
  MOV A,R2   ;Move the current divisor low byte into the accumulator
  RLC A      ;Shift low-byte left, rotate through carry to apply highest bit to high-byte
  MOV R2,A   ;Save the updated divisor low-byte
  MOV A,R3   ;Move the current divisor high byte into the accumulator
  RLC A      ;Shift high-byte left high, rotating in carry from low-byte
  MOV R3,A   ;Save the updated divisor high-byte
  JNC div1   ;Repeat until carry flag is set from high-byte
```

In the case of our example, once the above code is executed the registers will be as follows (including the carry bit 'C'):

```
      C/R1/R0 0 00000000 10110011
      C/R3/R2 1 00000000 00000000
```

At this point we can do the division itself. As we are in binary mode there is no need for a real division--it's just a comparison. At this point it's important to know the steps from above.

**Step 2. Shift left the divisor.**



```
div2:          ;Shift right the divisor
  MOV A,R3     ;Move high-byte of divisor into accumulator
  RRC A        ;Rotate high-byte of divisor right and into carry
  MOV R3,A     ;Save updated value of high-byte of divisor
  MOV A,R2     ;Move low-byte of divisor into accumulator
  RRC A        ;Rotate low-byte of divisor right, with carry from high-byte
  MOV R2,A     ;Save updated value of low-byte of divisor
  CLR C        ;Clear carry, we don't need it anymore
  MOV 07h,R1   ;Make a safe copy of the dividend high-byte
  MOV 06h,R0   ;Make a safe copy of the dividend low-byte
  MOV A,R0     ;Move low-byte of dividend into accumulator
  SUBB A,R2    ;Dividend - shifted divisor = result bit (no factor, only 0 or 1)
  MOV R0,A     ;Save updated dividend
  MOV A,R1     ;Move high-byte of dividend into accumulator
  SUBB A,R3    ;Subtract high-byte of divisor (all together 16-bit substraction)
  MOV R1,A     ;Save updated high-byte back in high-byte of divisor
  JNC div3     ;If carry flag is NOT set, result is 1
  MOV R1,07h   ;Otherwise result is 0, save copy of divisor to undo subtraction
  MOV R0,06h
div3:
  CPL C        ;Invert carry, so it can be directly copied into result
  MOV A,R4
  RLC A        ;Shift carry flag into temporary result
  MOV R4,A
  MOV A,R5
  RLC A
  MOV R5,A
  DJNZ B,div2  ;Now count backwards and repeat until "B" is zero
```

To see how the loop works here are the registers after each step:

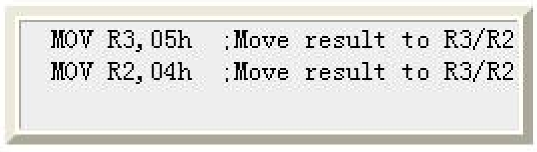
```
1 r1/0 00000000 10110011 ;dividend
  r3/2 10000000 00000000 ;divisor
  r5/4 00000000 00000000 ;result

2 r1/0 00000000 10110011 ;dividend
  r3/2 01000000 00000000 ;divisor
  r5/4 00000000 00000000 ;result

...
```

```
8  r1/0 00000000 10110011 ;dividend
   r3/2 00000001 00000000 ;divisor
   r5/4 00000000 00000000 ;result

9  r1/0 00000000 00110011 ;dividend
   r3/2 00000000 10000000 ;divisor
   r5/4 00000000 00000001 ;result

10 r1/0 00000000 00110011 ;dividend
   r3/2 00000000 01000000 ;divisor
   r5/4 00000000 00000010 ;result

11 r1/0 00000000 00010011 ;dividend
   r3/2 00000000 00100000 ;divisor
   r5/4 00000000 00000101 ;result

12 r1/0 00000000 00000011 ;dividend
   r3/2 00000000 00010000 ;divisor
   r5/4 00000000 00001011 ;result

13 r1/0 00000000 00000011 ;dividend
   r3/2 00000000 00001000 ;divisor
   r5/4 00000000 00010110 ;result
```

STOP!

Register "B" is zero at this point. The remainder is already in R1/R0, and it is 3 decimal, same as above. The result is still in R5/R4, but we can see it's correct, too (10110b=22d). To finish the routine, we just "clean up" by moving R5/R4 to R3/R2.

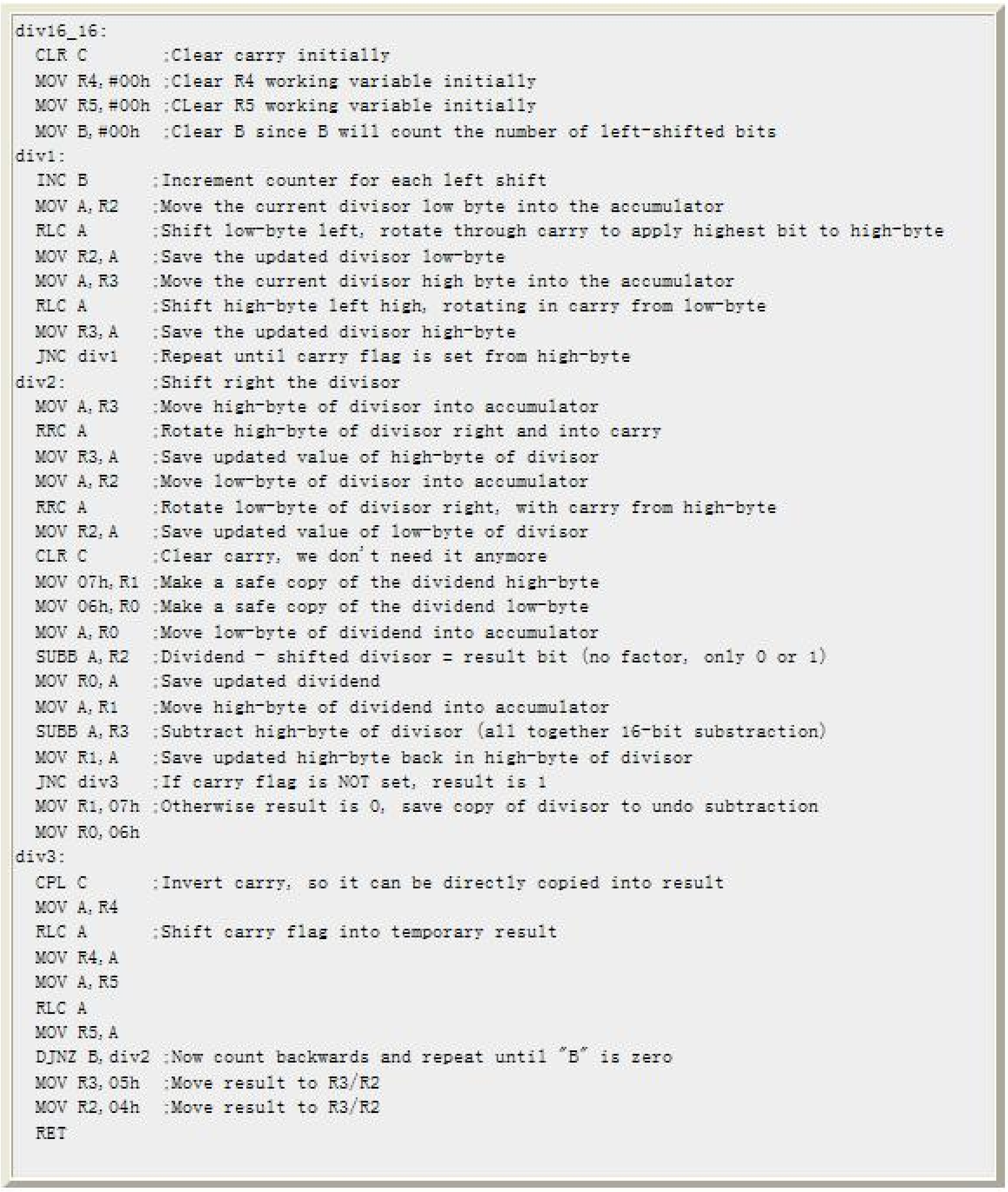
**Step 3. Final Clean-up.**



```
MOV R3,05h  ;Move result to R3/R2
MOV R2,04h  ;Move result to R3/R2
```

We used small numbers here for easier explanation. Of course it works also with 16-bit numbers, that's what it was designed to do.

Taken as a whole, the above division algorithm can be converted into an easy-to-use function that can be called from your program. To call this function, you should pre-load R1/R0 with the high/low value to be divided, and R3/R2 with the high/low value that the number is to be divided by.



```
div16_16:
  CLR C         ;Clear carry initially
  MOV R4,#00h  ;Clear R4 working variable initially
  MOV R5,#00h  ;CLear R5 working variable initially
  MOV B,#00h   ;Clear B since B will count the number of left-shifted bits
div1:
  INC B        ;Increment counter for each left shift
  MOV A,R2     ;Move the current divisor low byte into the accumulator
  RLC A        ;Shift low-byte left, rotate through carry to apply highest bit to high-byte
  MOV R2,A     ;Save the updated divisor low-byte
  MOV A,R3     ;Move the current divisor high byte into the accumulator
  RLC A        ;Shift high-byte left high, rotating in carry from low-byte
  MOV R3,A     ;Save the updated divisor high-byte
  JNC div1     ;Repeat until carry flag is set from high-byte
div2:          ;Shift right the divisor
  MOV A,R3     ;Move high-byte of divisor into accumulator
  RRC A        ;Rotate high-byte of divisor right and into carry
  MOV R3,A     ;Save updated value of high-byte of divisor
  MOV A,R2     ;Move low-byte of divisor into accumulator
  RRC A        ;Rotate low-byte of divisor right, with carry from high-byte
  MOV R2,A     ;Save updated value of low-byte of divisor
  CLR C        ;Clear carry, we don't need it anymore
  MOV 07h,R1 ;Make a safe copy of the dividend high-byte
  MOV 06h,R0 ;Make a safe copy of the dividend low-byte
  MOV A,R0     ;Move low-byte of dividend into accumulator
  SUBB A,R2    ;Dividend - shifted divisor = result bit (no factor, only 0 or 1)
  MOV R0,A     ;Save updated dividend
  MOV A,R1     ;Move high-byte of dividend into accumulator
  SUBB A,R3    ;Subtract high-byte of divisor (all together 16-bit substraction)
  MOV R1,A     ;Save updated high-byte back in high-byte of divisor
  JNC div3     ;If carry flag is NOT set, result is 1
  MOV R1,07h ;Otherwise result is 0, save copy of divisor to undo subtraction
  MOV R0,06h
div3:
  CPL C        ;Invert carry, so it can be directly copied into result
  MOV A,R4
  RLC A        ;Shift carry flag into temporary result
  MOV R4,A
  MOV A,R5
  RLC A
  MOV R5,A
  DJNZ B,div2 ;Now count backwards and repeat until "B" is zero
  MOV R3,05h  ;Move result to R3/R2
  MOV R2,04h  ;Move result to R3/R2
  RET
```