Z80 Assembly - Multiplication and Division

We take the operations of multiplication and division for granted, because we were taught in the school how to do them. For the Z80 these operations are not as trivial as for us, because they are not built-in functions of the CPU. We already know that the Z80 can perform addition and subtraction. Combined with some kind of loop, this is already enough to create a simple routine for both operations.

First approach

Just as in elementary school, we can start learning multiplication by considering it as a sequence of additions. Let's see how this idea can be realised in assembly language:

```
Multiply:                       ; this routine performs the operation HL=D*E
  ld hl,0                       ; HL is used to accumulate the result
  ld a,d                        ; checking one of the factors; returning if it is zero
  or a
  ret z
  ld b,d                        ; one factor is in B
  ld d,h                        ; clearing D (H is zero), so DE holds the other factor
MulLoop:                        ; adding DE to HL exactly B times
  add hl,de
  djnz MulLoop
  ret
```

The method is quite self-explanatory. Notice that if it wasn't for the verification of the factor to be put into `B`, a value of zero would have resulted in a multiplication with 256. To see this yourself, you should recall how `djnz` works.

Division can be treated as a sequence of subtractions in a similar way:

```
Divide:                         ; this routine performs the operation BC=HL/E
  ld a,e                        ; checking the divisor; returning if it is zero
  or a                          ; from this time on the carry is cleared
  ret z
  ld bc,-1                      ; BC is used to accumulate the result
  ld d,0                        ; clearing D, so DE holds the divisor
DivLoop:                        ; subtracting DE from HL until the first overflow
  sbc hl,de                     ; since the carry is zero, SBC works as if it was a SUB
  inc bc                        ; note that this instruction does not alter the flags
  jr nc,DivLoop                 ; no carry means that there was no overflow
  ret
```

The algorithm is simple: the dividend is decreased by the divisor, and `BC` is increased by one each time the divisor is found to be smaller than the decreasing dividend. The problem is that `BC` is also increased once when the dividend overflows. This is cured by loading -1 initially instead of 0, which compensates for the additional increase.

These little routines look nice and do the job properly, but there is still one important aspect to consider. What happens when `B` is a relatively great number in the first case? Or in the case of the division, what happens if we have a great dividend and a small divisor? The answer is obvious: the result will be calculated very slowly. It will take a lot of time to go through the loop, because the number of iterations will be quite high. The solution to this problem lies in the standard methods for multiplication and division. However, before starting the actual discussion, we have to look at some specific instructions that we need for this.

Shift instructions

Shifting is a bit-level operation. When you write out a number in binary form and perform some kind of shift operation on it, you will see the bits sliding in either direction. The phenomenon is similar to that when you multiply or divide a number with a power of 10: the digits are shifted leftwards or rightwards. The Z80 (and practically all the other processors) can do the same thing in base 2, and the instructions performing this operations are referred to as shift instructions.

There is a number of ways you can shift the contents of the register. You can either drop the bit "falling out" of the register or move it to the other end, making the bits move in some kind of a loop. This is called rotation. Generally the bit leaving the register shifted is put into the `carry` flag. The bit entering the register depends on the Z80 instruction you use: it can be zero or one when shifting, or the value of either the bit just leaving the register or the former content of the `carry`. To see all the possibilities of shifting and rotating on the Z80, you can find a **detailed reference** in the appendix.

As an example, here is an illustration of how these instructions work:

```
  ld a,%11010011                ; A=%11010011, carry=?
  sla a                         ; A=%10100110, carry=1
  rla                           ; A=%01001101, carry=1
  rlca                          ; A=%10011010, carry=0
  sra a                         ; A=%11001101, carry=0
  rra                           ; A=%01100110, carry=1
  ...
```

I strongly recommend that you learn to use the shift instructions very well, because they are essential to perform 99.9% of all the possible data manipulations. Read through the linked reference and look at the above example to get the basic idea, the real skill will come anyway with practice.

Fast 8-bit multiplication

Knowing the shift instructions we can directly translate the standard multiplication process into Z80 assembly. Let's look at this process as we were taught to do it:

```
1872 * 216    (factors)
----
3744          (auxiliary products)
 1872
 11232
------
404352        (product)
```

What happens exactly? For each digit of the multiplier we calculate the digit times the multiplicand and write it to the corresponding place, which is determined by the place-value of the digit currently in question. At the end, we add these products (which are multiplied by the corresponding powers of 10 due to the place we wrote them to) to get the final product.

This seems to be a bit more complex than a simple sequence of additions, but things start looking less complicated in base 2 (with the corresponding numbers in base 10 in brackets):

```
1011 * 101    (11 * 5)
----
1011          (auxiliary products)
  1011
------
110111        (55)
```

The main difference is that each digit can be either zero or one, which means that multiplying with a digit means either taking zero or the other factor itself as the product. So we can conclude that the auxiliary products can be obtained by shifting the original multiplicand. The digit of the multiplier decides whether we add the current product to the final result or not. Let's see a possible code to carry out this task:

```
Mul8:                           ; this routine performs the operation HL=DE*A
  ld hl,0                       ; HL is used to accumulate the result
  ld b,8                        ; the multiplier (A) is 8 bits wide
Mul8Loop:
  rrca                          ; putting the next bit into the carry
  jp nc,Mul8Skip                ; if zero, we skip the addition (jp is used for speed)
  add hl,de                     ; adding to the product if necessary
Mul8Skip:
  sla e                         ; calculating the next auxiliary product by shifting
  rl d                          ; DE one bit leftwards (refer to the shift instructions!)
  djnz Mul8Loop
  ret
```

The routine is not really bigger than the original addition loop, yet it is considerably faster. It is important to notice that the speed of the operation does not depend on the absolute value of the multiplier any more. However, its binary weight, i. e. the number of 1's in its binary representation will affect the time needed considerably. As a trick you can note that I used `rrca` to get the next bit of the `A` register. This way the `A` register is preserved after the loop is over. I could have used `rra` instead, but then the value of the accumulator would be more or less the old value of `D` upon leaving the loop.

The result will be correct if the product is less than 65536, or if you take the number in `DE` as signed integer. For example, if you write 65534 in `DE`, which is equivalent to -2 if considered as a signed number, the resulting product will be the value of -2*`A` in 16 bits. `A` is always treated as an unsigned 8-bit integer.

You might not need 16 bits for either factor. In this case using `DE` as one of the factors seems to be redundant. Here is a tricky solution to this problem:

```
Mul8b:                          ; this routine performs the operation HL=H*E
  ld d,0                        ; clearing D and L
  ld l,d
  ld b,8                        ; we have 8 bits
Mul8bLoop:
  add hl,hl                     ; advancing a bit
  jp nc,Mul8bSkip               ; if zero, we skip the addition (jp is used for speed)
  add hl,de                     ; adding to the product if necessary
Mul8bSkip:
  djnz Mul8bLoop
  ret
```

This code is faster than the previous one, but it is limited to 8-bit unsigned factors. I'm sure it looks a bit embarrassing at first, but it is nothing more than a little trick. (I know I am slightly overusing this expression...)

What's interesting in this solution is that `HL` is used to hold both one of the factors and the final product at the same time. The instruction `add hl,hl` is actually a shift to the left (which is equivalent to a multiplication by two). When it is encountered, the upmost bit of `HL` is put into the `carry`, all the 16 bits are shifted, and a zero appears in the last bit. I. e. after its first execution the lower 9 bits of `HL` are available for the product. When we add `DE`, we add an 8-bit number in the worst case, therefore it can be done without disturbing the contents of the factor originally residing in `H`.

You can still ask when the auxiliary product is calculated (which was done by shifting `DE` in the previous version). You have to realise that this is done with the same addition (`add hl,hl`) as the verification of the next bit of the factor in `H`! To see this, imagine that `H` was equal to 128 at the beginning. This is 2^7, i. e. a one followed by 7 zeroes in base 2. Therefore multiplying by 128 would mean shifting by 7 bits leftwards. Let's see if this happens.

The first `add hl,hl` puts one into the carry and results in `HL`=0 (remember, the original value was $8000). Since we have a `carry`, we add `DE` to `HL` at this point. During the following iterations `HL` will be shifted leftwards bit by bit, and the `carry` will be constantly zero, meaning that we do not add `DE` any more. Consequently, the final value of `HL` will be `DE`*128: DE shifted leftwards by 7 bits. Notice that we are processing the factor (`H`) from its upper end contrary to the other solution, meaning that we start by adding `DE`*128, then `DE`*64 etc. until reaching `DE`*1 itself (of course only if the corresponding bit in the other factor is 1). However, the these multiplications are performed later than the additions, so it might be a bit harder to see how it works.

Fast 8-bit division

Before going into the details, let's revise division in base 2 (the operation is 216/7=30, with 6 remaining):

```
    11011000/111=11110 (quotient)
   - 111
   -----
     1101
    - 111
    -----
      1100
     - 111
     -----
       1010
      - 111
      -----
         110 (remainder: already less than the divisor)
```

If we were in a base greater than 2 we would need to do small divisions all the way along. But in base 2—similarly to the case of multiplication—we only have two-way decisions instead. The two possibilities are the following: either the divisor divides the upper digits currently chosen, then the next digit of the quotient will be 1, and a subtraction must be performed; or the divisor does not divide the digits chosen, in which case the next digit of the quotient will be zero, and we add the next digit of the dividend to the ones we have without prior subtraction.

The best assembly language representation of this task uses the same trick as the second multiplication above: the same place is used to store one of the inputs and the output at the same time. In this case, this place is a "24-bit register", **AHL**. Of course, these three registers are separate, but we will consider them as one this time. The code follows here:

```
Div8:                           ; this routine performs the operation HL=HL/D
  xor a                         ; clearing the upper 8 bits of AHL
  ld b,16                       ; the length of the dividend (16 bits)
Div8Loop:
  add hl,hl                     ; advancing a bit
  rla
  cp d                          ; checking if the divisor divides the digits chosen (in A)
  jp c,Div8NextBit              ; if not, advancing without subtraction
  sub d                         ; subtracting the divisor
  inc l                         ; and setting the next digit of the quotient
Div8NextBit:
  djnz Div8Loop
  ret
```

It's time to have a hard time. I will let you analyse this routine on your own. Try to understand it without any help, because understanding others' code is also a skill one must practice—again, I know this is not the first time I am saying this, but it is part of the training programme. :)

Fast 16-bit multiplication

The following solution for this problem is a direct rewrite of the second multiplication routine (the one which did **HL**=**H***E**), therefore I am not commenting it too much:

```
Mul16:                          ; This routine performs the operation DEHL=BC*DE
  ld hl,0
  ld a,16
Mul16Loop:
  add hl,hl
  rl e
  rl d
jp nc,NoMul16
  add hl,bc
  jp nc,NoMul16
  inc de                        ; This instruction (with the jump) is like an "ADC DE,0"
NoMul16:
  dec a
  jp nz,Mul16Loop
  ret
```

If you have two unsigned 16-bit numbers in **BC** and **DE**, a call to ***Mul16*** will return their (unsigned) product in 32 bits. The higher half of the result is in **DE**, while the lower half can be found in **HL**. You should realise that this routine can be used to multiply two signed 16-bit integers as well. If you consider the lowest 16 bits of the result (i. e. the contents of **HL**), it will be the correct signed 16-bit product of the two numbers. Of course, this only holds if the product remains within 16 bits. (For example, try to calculate $(-1)^2$, which is actually 65535^2. The lowest 16 bits of the result will contain 1.)

Other aspects

The pieces of code presented above are certainly not the best solutions. Sometimes you need speed, in other cases you want to make your routines as short as possible. You can make these operations faster if you unroll the loops (i. e. instead of repeating them using **djnz** or any other conditional jump instruction, you write them down as many times as you want them to execute), but then you have to sacrifice some memory to achieve a higher speed.

Variations on the subject can be found on the site of **Macross** (check the "universal code" section), on the **page of Milos Bazelides** and presumably in your head. I encourage you to use the latter, you will be surprised how much potential it has. :-)

Back to the index