MSX Assembly Page

Multiplications, divisions and shifts

Because the Z80 does not have a built-in multiplication instructions, when a programmer wants to do a multiplication he has to it manually. The programmer might take a multiplication routine from a book or a magazine, which may not necessarily be optimized for the specific case (or optimized at all), or he might even uses a routine that works by adding the same value n times, which is pretty much the worst solution to the problem.

This article presents you with optimized methods for multiplication. First it introduces you to multiplications using shifts, which are very fast and can be used if one of the multiplication parameters is a fixed number. Next, it provides a number of optimized generic multiplication and division routines for various bit-depths, and even a square root routine.

Latest updates:

2009-08-23
Made 16-bit division routine 120 T-states faster, thanks to Juha N.

2009-03-27
Added division by 9 routine provided by Ricardo Bittencourt.

2005-05-12
Optimized the Div16 routine, it is only a humble 7 T-states faster than it was before, but can now be unrolled to be up to 234 T-states faster! It is also 15 bytes smaller. Thanks go to Flyguille.

2005-05-01
Optimized the Mult32 routine, it is now 96 T-states faster and three bytes smaller. Thanks to Colin Douglas Howell.

2005-07-25
Fixed the Mult32, thanks to Nyyrikki for pointing me to a bit that didn’t make sense.

2005-02-06
Put a new (faster) square root routine online, by Ricardo Bittencourt. Also, added the Z80 Bits article kindly provided by Petr Petyovsky from the Sinclair scene, which has some very nice routines as well. I will see if I can improve these multiplication and division routines a little more with those in the future.

2005-02-01
- Did some Div16 bug hunting, fixed two bugs. It should be error-free now. Another thing, the divider of the Div16 routine had to be a 15-bit value, so changed the

8-bit multiplications using shifts

When you shift a register 1 bit to the left, you multiply the value of the register with 2. This shifting can be done using the SLA r instruction. By doing several shifts in sequence you can very easily multiply by any power of 2. For example:

```
        ld b,3          ;Multiply 3 with 4
        sla b           ;x4
        sla b           ;result: b = 12
```

If you use register A you can multiply faster by using the ADD A,A instruction, which is 5 T-states per instruction instead of 8. So ADD A,A is exactly the same as SLA A, or a multiplication by two. On a sidenote, instead of using ADD A,A, you can also use RLCA, which effectively behaves the same.

```
        ld a,15         ;Multiply 15 with 8
        add a,a         ;x8
        add a,a
        add a,a         ;result: a = 120
```

When programming multiplications you must always make sure the result will never exeed 255, in other words a carry may not occur. In the case of that happening, RLCA actually acts different from SLA A or ADD A,A (in some cases more usable, in some cases less usable). But generally that isn’t of any concern because when register A overflows the result will generally not be of much use anymore.

If you want to multiply by another value than a power of two, you can almost always achieve the desired result by storing inbetween values during the shifting and adding or subtracting them up afterwards. A few examples:

```
        ld a,5          ;Multiply 5 with 20 (= A x 16 + A x 4)
        add a,a         ;x16
        add a,a
        ld b,a          ;Store value of A x 4 in B
        add a,a
        add a,a
        add a,b         ;Add A x 4 to A x 16, result: a = 100
```

If you would want to multiply with 22, you would also save the value after the first add in a register and add it to the total afterwards.

name to Div15, and added some additional Div16 code.
- Added square root Sqr16 routine by Arjan Bakker (based on the one published in MCCM).
- Changed Mult12 unrolled version’s routine name to Mult12U, and optimized the `ld l,0` here aswell.
- Re-calculated all timing figures to include M1 wait states

2005-08-28
The MCCW article Multiplication on a Z80 describes a very fast multiplication method.

2005-04-25
Changed `ld hl,0` to `ld l,0` in Mult12, and removed it altogether in the Mult16 routine. Thanks to Sjoerd Mastijn for the tip :).

Sometimes, you can also use substractions to achieve your goals faster. For example, the multiplication of A with 15. This can be done by using the method described above, however, in that case you’ll need 4 temporary registers, and four additional adds afterwards. This could better be done as follows, which requires 1 more multiplication but only uses 1 temporary register and 1 subtraction afterwards:

```
        ld a,3          ;Multiply 3 with 15 (= A x 16 - A x 1)
        ld b,a          ;Store value of A x 1 in B
        add a,a         ;x16
        add a,a
        add a,a
        add a,a
        sub b           ;result: a = 45
```

8-bit divisions using shifts

Now, divisions are very much like multiplications. If a multiplication routine is complex, a division routine is even more so. However, using shifts it’s all too easy to divide in Assembly. It is done by simply shifting the other way, to the right. For this, you should use the SRL r instruction. An example:

```
        ld b,3          ;Divide 18 by 4
        srl b           ;x4
        srl b           ;result: b = 4 (rest 2 is lost)
```

There is no real fast alternative to a shift right when using register A. As an alternative, you can use RRCA for that. However, when using RRCA you must make sure there will be no rest, otherwise the result won’t be correct. This can be achieved by ANDing the original value with a value clearing the lower bits (which would otherwise be rotated out), or by making sure you are using values which are always a multiple of the divider.

```
        ld a,153        ;Divide 153 by 8
        and a,%11111000 ;Clear bits 0-2 (equals 256 - 8)
        rrca            ;/8
        rrca
        rrca            ;result: a = 19
```

Dividing with values other than powers of 2 is trickier and is often not possible. If you want to see for yourself, try to construct a routine which divides 100 by 20. To know if a value can be divided, you must look at the amount of trailing zeroes in the binary representation of that value. The maximum number of RRCA’s you can use is then equal to that number. If you look at the beforementioned value 100 in binary (%01100100), you will see there are two trailing zeroes. Therefor, the division routine can only use 2 RRCA’s, while dividing by 20 needs 4 of them.

To be rather blunt about it, unless you have very tight control over the values which are given as a parameter, it is hardly possible to divide by values which are not a power of 2. Using a method like with multiplications, dividing by 3 for example is ofcourse possible, but only if all input values are even and all multiples of 3. If either of these conditions is not met, the result will be flawd.

16-bit shifts

There are also ways to shift 16-bit registers. This is done using a 8-bit shift in combination with an 8-bit rotate and the carry bit. To shift a register DE one bit to the left you should use:

```
        sla e
        rl d
```

To shift it one bit to the right (now with BC as example), use:

```
        srl b
        rr c
```

Unfortunately, generally those 16-bit shifts are rather slow compared to 8-bit shifts (which will most often take place in the fast A register, which makes them almost 4 times as fast). However, just as with the 8-bit shifts, there is also the possibility to do faster 16-bit shifts to the left using the ADD instruction.

```
        add hl,hl       ;shift HL 1 bit left... hl = hl x 2
```

So, the best way to multiply 16-bit values is by using register HL in combination with ADD HL,HL instructions.

Fast generic multiplication routines

Well, that’s about everything there is to tell about shifting and multiplications / divisions. There are a few more minor tricks, but really all I can say about that is to just be a little creative.

Finally I'll give you the as far as I know fastest generic multiplication and division routines possible. These are the routines you should use when:

- both multiplication/division parameters are unknown,
- if you want to divide by another value than a power of 2 or need the rest value,
- if the known value is such a value that it can not feasibly be calculated using the methods described above.

You can also use them if you want your code to look really tidy and don't really care about the speed. As for the inner workings of them, all I will say about it is that they work much like how you learned to solve multiplications and divisions in elementary school. But that really is of no importance. Just copy them into your program as (unrolled) subroutines or, if you feel like removing any notion of structured programming, as inline code or a macro.

Oh, and if you really have very low variable multiplication factors, you might want to use the dreaded multiply-by-adding-x-times-loop method. It's probably faster.

Left-rotating multiplication

These are left-rotating multiplication routines. Their speed is basically quite constant, although depending on the number of 1's in the primary multiplier there may be a slight difference in speed (a 1 usually takes 7 states longer to process than a 0).

```
;
;Multiply 8-bit values
;In:  Multiply H with E
;Out: HL = result
;
Mult8:
        ld d,0
        ld l,d
        ld b,8
Mult8_Loop:
        add hl,hl
        jr nc,Mult8_NoAdd
        add hl,de
Mult8_NoAdd:
        djnz Mult8_Loop
        ret
```

```
;
;Multiply 8-bit value with a 16-bit value
;In: Multiply A with DE
;Out: HL = result
;
Mult12: ld l,0
        ld b,8
Mult12_Loop:
        add hl,hl
        add a,a
        jr nc,Mult12_NoAdd
        add hl,de
Mult12_NoAdd:
        djnz Mult12_Loop
        ret
```

```
;
;Multiply 16-bit values (with 16-bit result)
;In: Multiply BC with DE
;Out: HL = result
;
Mult16: ld a,b
        ld b,16
Mult16_Loop:
        add hl,hl
        sla c
        rla
        jr nc,Mult16_NoAdd
        add hl,de
Mult16_NoAdd:
        djnz Mult16_Loop
        ret


;
;Multiply 16-bit values (with 32-bit result)
;In: Multiply BC with DE
;Out: BCHL = result
;
Mult32: ld a,c
        ld c,b
        ld hl,0
        ld b,16
Mult32_Loop:
        add hl,hl
        rla
        rl c
        jr nc,Mult32_NoAdd
        add hl,de
        adc a,0
        jp nc,Mult32_NoAdd
        inc c
Mult32_NoAdd:
        djnz Mult32_Loop
        ld b,c
        ld c,a
        ret
```

Right-rotating multiplication

Right-rotating multiplications are basically not very different from left-rotating multiplications. They use a very similar process, and the number of steps in the calculation is equal. However on the Z80 left-rotating multiplications can be coded much faster and compacter, which is why all the previous routines were using the left rotating variants.

However there is one rather nice advantage to right-rotating divisions, which is that it only has to loop until there are no bits left, after that it can be terminated without any further operations. So if a value actually only uses 4 bits (for example the number 11) the routine only has to loop 4 times and can be terminated afterwards by using a fast conditional check, so there is barely any extra effort involved in this potentially great speedup.

Looking at the table below with speed calculations of Mult12R routine and given the fact that the optimal left-rotating Mult12 (unrolled) takes 268 T-states (M1 states included, basically meaning 1 additional state for each instruction) you can see that in the efficiency boundary is at 4 bits. Above that, a normal Mult12 is faster. However, a 4-bit Mult12 will ofcourse be the faster one again, so to take advantage of this the majority but not all of the values should fall within the 4 bit range.

How long an n-bit multiplication takes (on average):
1-bit: 78.5 T-states
2-bit: 140 T-states
3-bit: 201.5 T-states
4-bit: 263 T-states
5-bit: 324.5 T-states
6-bit: 386 T-states
7-bit: 447.5 T-states
8-bit: 509 T-states

It is important to note that this may seem fast, but do not forget that the number of bits and the possible values are related to eachother on a logarithmic scale. So if you use random byte values, the number of values within the 4 bit range is only 1 out of every 16 values. To indicate a little better what this means, take a look at the following table which specifies the average speed you can expect for a random value within a certain domain.

Average speeds in the specified domain:

<0,255>	447.980 T-states
<0,127>	386.961 T-states
<0,63>	326.422 T-states
<0,31>	266.844 T-states
<0,15>	209.188 T-states
<0,7>	155.375 T-states
<0,3>	109.25 T-states
<0,1>	78.5 T-states

So, concluding, if you make a large number of multiplications of which one of the values is primarily smaller than 6 bits, for example if your data is logarithmic, using a right-rotating algorithm would be the fastest choice. To give a real-world usage example, this routine might for example be applicable to calculations on frequency tables within a music replayer, which are set on a logarithmic scale.

Well, here is the actual routine. Note that it can easily be converted to a Mult8R-routine by inserting an ld d,0 instruction right after the ld hl,0, and also that unrolling this routine will not give you additional speed. Oh, and by the way, although I've been talking about right-rotating all the time, this implementation is actually right-shifting ^_^.

```
;
;Multiply 8-bit value with a 16-bit value (right rotating)
;In: Multiply A with DE
;     Put lowest value in A for most efficient calculation
;Out: HL = result
;
Mult12R:
        ld hl,0
Mult12R_Loop:
        srl a
        jr nc,Mult12R_NoAdd
        add hl,de
Mult12R_NoAdd:
        sla e
        rl d
        or a
        jp nz,Mult12R_Loop
        ret
```

Unrolled left-rotating multiplication

For completeness' sake, here also an example of an unrolled left-rotating multiply routine. It takes a little more space, but is significantly faster. Actually it's still quite compact, only 41 bytes. This one takes 268 T-states on average (M1 waits included). The minimum is 235 states, and the maximum is 301 ticks.

By the way, I looked at the possibility to use the same technique as with the right-rotating multiplication on this routine, it can fairly easy be done by putting jumps inbetween which jump into a list of add hl,hl's. However, loss in speed the additional jumps cause don't outweigh the gain, and it is hardly practical anyway since it would apply to the nr. of bits used from the left (the number 128 would use 1 bit, and 64 two).

```
;
;Multiply 8-bit value with a 16-bit value (unrolled)
;In: Multiply A with DE
;Out: HL = result
;
Mult12U:
        ld l,0
        add a,a
        jr nc,Mult12U_NoAdd0
        add hl,de
Mult12U_NoAdd0:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd1
        add hl,de
Mult12U_NoAdd1:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd2
        add hl,de
Mult12U_NoAdd2:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd3
        add hl,de
Mult12U_NoAdd3:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd4
        add hl,de
Mult12U_NoAdd4:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd5
        add hl,de
Mult12U_NoAdd5:
        add hl,hl
        add a,a
        jr nc,Mult12U_NoAdd6
        add hl,de
Mult12U_NoAdd6:
        add hl,hl
        add a,a
        ret nc
        add hl,de
        ret
```

It is also worthwhile to unroll the other multiplication routines. For the Mult8 routine for example, it saves 115 out of 367 T-states on average, that's 31% faster, at the cost of just 38 bytes (14 → 52).

Fast generic division routines

Before we get to the actual division routines, when dividing you don't necessarily need to use the (slower) division routines. Don't forget, multiplications and divisions are related. If you want to divide by 2, you can also multiply with 0.5. Translating this to these routines, to divide an 8-bit value by another 8-bit value (like Div8 does), you can call a Mult8 routine with parameter 1 being (1 / the first 8-bit value * 256) and parameter 2 being the other 8-bit value. The resulting word will be a fixed-point hexadecimal value, with the comma laying between the high and the low byte.

For example, calculate 55 / 11:

Input A: #18 (1/11*256 = 23.272727, rounded up to 24)
Input B: #37 (55)
Output: #528 (#5.28 or 5,15625 decimal)

Note that the output isn't exactly the correct value (which should have been #500), this is because 1/11 is actually not a very nice number, both in decimal notation (.090909) as in hexadecimal notation (#.1745D1). If we hadn't rounded up the value to #.18 but to #.17 which would be proper rounding, the result would have been #4F1. This would have been more exact than the current result, however our goal is to write optimum code, and to distill the correctly rounded whole number from this result (being 5) would be much easier when we have #528 as a result, because we could then simply take just the upper byte. Also the 'proper rounding' would require some additional code. This is why we rounded up the value.

If you take a base value too large, the error accumulates. If you for example try to divide 2200 by 11, the result will be 206, while it should have been 200. To solve this problem you can either increase the resolution (use a 16-bit division value (#.1746) and a 24- or 32-bit result) or divide by values which are 'tidy' in hexadecimal notation (being the powers of 2). However also remember that no matter which base you use, be it 10 or 16, you will always have divisions which produce errors. The values with which this happens differ, but you will have to deal with resolution limitations and rounding.

Anyways. These are the general division routines. Slower than the multiplication routines but still as fast as possible, and probably very useful.

```
;
;Divide 8-bit values
;In: Divide E by divider C
;Out: A = result, B = rest
;
Div8:   xor a
        ld b,8
Div8_Loop:
        rl e
        rla
        sub c
        jr nc,Div8_NoAdd
        add a,c
Div8_NoAdd:
        djnz Div8_Loop
        ld b,a
        ld a,e
        rla
        cpl
        ret


;
;Divide 16-bit values (with 8-bit result)
;In: Divide HL by divider D
;Out: L = result, H = rest
;
Div12:  ld b,8
Div12_Loop:
        and a
        sbc hl,de
        jp p,Div12_NoAdd
        add hl,de
        add hl,hl
        djnz Div12_Loop
        ret
Div12_NoAdd:
        scf
        adc hl,hl
        djnz Div12_Loop
        ret


;
;Divide 16-bit values (with 16-bit result)
;In: Divide BC by divider DE
;Out: BC = result, HL = rest
;
Div16:  ld hl,0
        ld a,b
        ld b,8
Div16_Loop1:
        rla
        adc hl,hl
        sbc hl,de
        jr nc,Div16_NoAdd1
        add hl,de
Div16_NoAdd1:
        djnz Div16_Loop1
        ld b,a
        ld a,c
        ld c,b
        ld b,8
Div16_Loop2:
        rla
        adc hl,hl
        sbc hl,de
        jr nc,Div16_NoAdd2
        add hl,de
Div16_NoAdd2:
        djnz Div16_Loop2
        rla
        cpl
        ld b,a
        ld a,c
        ld c,b
        rla
        cpl
        ld b,a
        ret
```

Thanks to Flyguille for the Div16 routine, taken from his MNBIOS source code.

The division routines can be unrolled as well to gain a nice speed increase. If you take the Div16 routine for example, it takes 1297 T-states to complete. When unrolled however, it only needs 1070 T-states, that's an 18% speed increase. The additional cost in bytes is 146 (original routine is 27 bytes).

Division by 9

Ricardo Bittencourt provided us with a fast routine for division by 9. It is built for .dsk or Disk ROM routines. It's very fast, but only works in the range 0-1440.

```
;
; division by nine
; enter     HL = number from 0 to 1440
; exit      A = HL/9
; destroy   HL,DE
;                                  Z80  R800
DIV9:   INC     HL              ;  7    1
        LD      D,H             ;  5    1
        LD      E,L             ;  5    1
        ADD     HL,HL           ;  12   1
        ADD     HL,HL           ;  12   1
        ADD     HL,HL           ;  12   1
        SBC     HL,DE           ;  17   2
        LD      E,0             ;  8    2
        LD      D,L             ;  5    1
        LD      A,H             ;  5    1
        ADD     HL,HL           ;  12   1
        ADD     HL,HL           ;  12   1
        ADD     HL,DE           ;  12   1
        ADC     A,E             ;  5    1
        XOR     H               ;  5    1
        AND     03FH            ;  8    2
        XOR     H               ;  5    1
        RLCA                    ;  5    1
        RLCA                    ;  5    1
        RET             ; total  = 157  22
```

Square root routine

This is a faster square root routine than the one that was previously here, test results say that it’s 26% faster. It is written by Ricardo Bittencourt, so many thanks to him :).

```
;
;Square root of 16-bit value
;In:  HL = value
;Out:  D = result (rounded down)
;
Sqr16:  ld de,#0040
        ld a,l
        ld l,h
        ld h,d
        or a
        ld b,8
Sqr16_Loop:
        sbc hl,de
        jr nc,Sqr16_Skip
        add hl,de
Sqr16_Skip:
        ccf
        rl d
        add a,a
        adc hl,hl
        add a,a
        adc hl,hl
        djnz Sqr16_Loop
        ret
```

Well, that’s it. Thanks go to mr. Rodnay Zaks :), for writing his book “Programming of the Z80”, which taught me about how to do multiplications on Z80, and has been the inspiration for the routines listed here. If you have any suggestions for speed improvements, or know of another method which might be faster under certain conditions, please let me know.

~Grauw