previous:
Realms of Adventure

MCCW issue 92,
March/April/May 2000
Back to contents

next:
De Maiskoek/Bits and Pieces

when speed matters

Multiplication on a Z80

Multiplication is one of those 'difficult' operations for a proccessor, one that eats up a lot of transistors and therefore uses a lot of CPU power and takes up much space on the silicium. So it doesn't surprise us that in the early days technicians didn't directly integrate multiplication instructions into the processor. So it wasn't build in in the — at that time — advanced Z80 8-bit CPU. Multiplication was left to be implemented by the programmers which needed it.

David Heremans

Directory

Most programmers won't care about the fact that multiplication wasn't built into the processor. If you use a higher level programming language it will most certainly provide the necessary commands or code libraries which will solve multiplication arithmetics for us. The only people who will worry about the missing CPU instructions are the people who directly are affected: the assembly language programmers. In our MSX case the only ones who really worry about multiplication are demo-programmers; they need a really fast solution for performance reasons. More normal applications can still use a relatively slow multiplication, since once assembled, ML is still the most performant and for the occasional multiplication one won't see the difference between 40 or 140 microseconds. If you need however thousands of multiplications like in realtime 3D algorithms or a realtime spectrum analyzer this 1000 times 100 microseconds will be a difference of 100 milliseconds between possible screen updates. With the fast code you get at maximum 1/40 ms = 25 Hz, so 25 frames per second. With the slow code you will obtain 7 frames per second. This means that your framerate can vary 357 percent depending on the code!

The solutions

Let's have a look at some of the possible solutions for multiplication on a Z80. Remember, just like in real life there is no best solution, it all depends upon the priorities needed for the programming project at hand.

Solution 1

Let's start off easy. Most novice assembly writers will solve multiplication with something that resembles the following code:

ML-listing: SOLUT1.ASC

```
; INPUT: THE VALUES IN REGISTER B EN C
; OUTPUT: HL = B * C
; CHANGES: AF,DE,HL,B
;
        LD HL,0
        LD A,B
        OR A
        RET Z
        LD D,0
        LD E,C
LOOP:   ADD HL,DE
        DJNZ LOOP
        RET
```

This is almost literally the definition of multiplication like most of us have learned in elementary school. 5 times 3 equals 3+3+3+3+3, 8 times 2 equals 2+2+2+2+2+2+2+2 etc. It is clear that this approach can

be optimized quite a bit, like making sure that register B contains the smallest value. Extra functionality can be easily implemented like adapting the routine so that it can multiply if there are one or two bytes in two's complement format and accordingly outputting a two's complements value in HL.

Advantages
It's simple, it's short. No more memory used than needed to store the code. And it is very easy to understand for outsiders. If you write code it is always nice if somebody else can read and maintain it later.

Disadvantages
Time needed by this routine can vary a lot. When B is zero it will be very fast. 11+4+4+7 is 26 T-states, on a regular 3.5 MHz Z80 this corresponds to ± 7.5 microseconds. In the worst case however this routine will need 11+4+4+7+7+4+254*(11+12)+11+7+7=5904 T-states, meaning milliseconds! Well allright, not really multiple milliseconds, one multiplication can take about 1.45 millisecond considering this worst case scenario. And don't forget on an MSX there is for every instruction an extra T-state wasted to make it even worse. This means that our Z80's 1.455 milliseconds will become 1.834 milliseconds when run on our beloved MSX.

Solution 2
Like in most assembly written applications you will need to make a choice between space and speed. The previous solution is probably the best choice if you were going for a very short solution, that needs the least amount of memory. If you however have some mapperpages free you can precalc all the information in advance in one *huge* matrix and then do lookups. Let's say you need multiplications where number one can be from zero up until 255 and number two is in the range of zero up until 63. Now we can use 32 kB to precalc all this. Let us say we are going to use the memory from #4000 to #BFFF to store this data. We will calculate the data with the following pseudo-code:

Pseudo-listing: PRECALC

```
for a=0 to 63
for b=0 to 255
solution=a*b
poke #4000+a*256+b,low-byte(solution)
poke #8000+a*256+b,high-byte(solution)
next b,a
```

If we have the data stored in this format we can use the following code to multiply:

ML-listing: SOLUT2.ASC

```
; IN: L NUMBER ONE (0-255)
;     A NUMBER TWO (0-63)
; OUT: BC IS RESULT
; CHANGES: AF,BC,HL
;
        OR A,#40
        LD H,A
        LD C,(HL)
        ADD A,#40
        LD H,A
        LD B,(HL)
```

```
        RET
```

Advantages
Fast, this solution will always, no matter what the input is, use 7+4+7+7+4+7+7=43 T-States. If you want to use two's complement numbers the code doesn't change, just make the needed matrix. Or you could alter your routine to take into account the signs of the numbers. Just introduce a litle overhead to transform both numbers to positive numbers and afterwards change HL to its two's complement when the final result had to be negative. This approach would let you multiply numbers from -63 up until 63 with any other number from -255 up until 255 using the same matrix.

Disadvantages
You need a lot of memory! At least you will have to store the result matrix. If you are going to store a 128×256 matrix your entire view range of 64 kB will be filled and you will have to introduce an extra overhead to use the memmory mapper. Making a matrix for 256×256 posibilities is going to use up 128 kB of memory. So for a regular European MSX2 this is not even possible, because there would be no mapperpage left for the program, stack and system variables!

The perfect compromise
Let's introduce a method of which the speed is constant and which doesn't fill the entire memory range. We still need a lookup table of 512 bytes to do our lookups. How are we going to go about it? Let me show you the code.

First, the lookup table. This one must start at #xx00. Most demo programmers will already have the habit of aligning lookup tables this way. It speeds up the indexing since there is no offset to take into account and all lookups can be calculated using boolean operations on addresses. We did this in the previous example also, it was not by coincidence that the lookup table started at #4000.

ML-listing: MULTAB.ASC

```
MULTAB  DB      0, 0, 1, 2, 4, 6, 9, 12, 16, 20
        DB      25, 30, 36, 42, 49, 56, 64, 72, 81, 90
        DB      100, 110, 121, 132, 144, 156, 169, 182, 196, 210
        DB      225, 240, 0, 16, 33, 50, 68, 86, 105, 124
        DB      144, 164, 185, 206, 228, 250, 17, 40, 64, 88
        DB      113, 138, 164, 190, 217, 244, 16, 44, 73, 102
        DB      132, 162, 193, 224, 0, 32, 65, 98, 132, 166
        DB      201, 236, 16, 52, 89, 126, 164, 202, 241, 24
        DB      64, 104, 145, 186, 228, 14, 57, 100, 144, 188
        DB      233, 22, 68, 114, 161, 208, 0, 48, 97, 146
        DB      196, 246, 41, 92, 144, 196, 249, 46, 100, 154
        DB      209, 8, 64, 120, 177, 234, 36, 94, 153, 212
        DB      16, 76, 137, 198, 4, 66, 129, 192

        DB      0, 192, 129, 66, 4, 198, 137, 76, 16, 212
        DB      153, 94, 36, 234, 177, 120, 64, 8, 209, 154
        DB      100, 46, 249, 196, 144, 92, 41, 246, 196, 146
        DB      97, 48, 0, 208, 161, 114, 68, 22, 233, 188
        DB      144, 100, 57, 14, 228, 186, 145, 104, 64, 24
        DB      241, 202, 164, 126, 89, 52, 16, 236, 201, 166
        DB      132, 98, 65, 32, 0, 224, 193, 162, 132, 102
        DB      73, 44, 16, 244, 217, 190, 164, 138, 113, 88
        DB      64, 40, 17, 250, 228, 206, 185, 164, 144, 124
        DB      105, 86, 68, 50, 33, 16, 0, 240, 225, 210
        DB      196, 182, 169, 156, 144, 132, 121, 110, 100, 90
        DB      81, 72, 64, 56, 49, 42, 36, 30, 25, 20
        DB      16, 12, 9, 6, 4, 2, 1, 0
```

```
        DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        DB      0, 0, 1, 1, 1, 1, 1, 1, 1, 1
        DB      1, 1, 1, 1, 1, 1, 2, 2, 2, 2
        DB      2, 2, 2, 2, 2, 2, 3, 3, 3, 3
        DB      3, 3, 3, 3, 4, 4, 4, 4, 4, 4
        DB      4, 4, 5, 5, 5, 5, 5, 5, 5, 6
        DB      6, 6, 6, 6, 6, 7, 7, 7, 7, 7
        DB      7, 8, 8, 8, 8, 8, 9, 9, 9, 9
        DB      9, 9, 10, 10, 10, 10, 10, 11, 11, 11
        DB      11, 12, 12, 12, 12, 12, 13, 13, 13, 13
        DB      14, 14, 14, 14, 15, 15, 15, 15

        DB      16, 15, 15, 15, 15, 14, 14, 14, 14, 13
        DB      13, 13, 13, 12, 12, 12, 12, 12, 11, 11
        DB      11, 11, 10, 10, 10, 10, 10, 9, 9, 9
        DB      9, 9, 9, 8, 8, 8, 8, 8, 7, 7
        DB      7, 7, 7, 7, 6, 6, 6, 6, 6, 6
        DB      5, 5, 5, 5, 5, 5, 5, 4, 4, 4
        DB      4, 4, 4, 4, 4, 3, 3, 3, 3, 3
        DB      3, 3, 3, 2, 2, 2, 2, 2, 2, 2
        DB      2, 2, 2, 1, 1, 1, 1, 1, 1, 1
        DB      1, 1, 1, 1, 1, 1, 1, 0, 0, 0
        DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        DB      0, 0, 0, 0, 0, 0, 0, 0
```

The precalced table may seem long and boring — which it is actually — but the code itself is much shorter and cleaner:

ML-listing: MULTROUT.ASC

```
; IN : A and D are to be multiplied
; OUT: HL is result
; CHANGES : AF,BC,E,HL
;
        LD      E,A
        SUB     D
        LD      H,MULTAB/256
        LD      L,A
        LD      C,(HL)
        INC     H
        LD      B,(HL)
        LD      A,E
        ADD     A,D
        LD      L,A
        LD      E,(HL)
        DEC     H
        LD      L,(HL)
        LD      H,E
        OR      A
        SBC     HL,BC
```

Some people would by now feel the urge to take paper and pencil and they would try to figure out how this works. If you are one of those people I'll give you a hint and afterwards you will find it much easier to figure out how it works. Ok, here is the hint. MULTAB is made using the folowing pseudo-code:

Pseudo-listing: PRECALC

```
for i=0 to 255
if i<128 then a=i else a=i-256
solution=int((a*a)/4)
poke i,low-byte(solution)
poke 256+i,high-byte(solution)
```

```
next
```

Those who feel they can tackle the program alone can start working now. For those who didn't find a piece of paper or whose pencils are blunt I will explain the code. Nobody would read this far and then appear too lazy to figure it out himself, right? Now hold on, here we go.

Everybody has once during his scolar career heard about the 'special products'. And we learned that $(a+b)(a-b)=a^2-b^2$ and that $(a+b)^2=a^2+2ab+b^2$. First of all, we make a lookup table for the function $f(x)=(x^2)/4$.

The trick is that now we can say that $a*b=f(a+b)-f(a-b)$. Let's prove that this is true. We are going to work out this function:

$a*b = f(a+b) - f(a-b)$	this is our original function
$a*b = ((a+b)^2)/4 - ((a-b)^2)/4$	we replaced the funtion by it's definition
$a*b = ((a+b)^2 - (a-b)^2)/4$	here we say $a/b - c/b = (a-c)/b$
$a*b = ((a^2 +2ab+b^2) - (a^2-2ab+b^2))/4$	we worked out the special products
$a*b = (a^2-2ab+b^2 - a^2+2ab-b^2)/4$	just dropped the brackets that weren't needed Pay attention to alter the signs if needed, $-(-X)$ becomes $+X$
$a*b = (a^2-a^2 +b^2-b^2 +2ab+2ab)/4$	rearranged the components between the brackets
$a*b = (4ab)/4$	Since $x-x$ can be dropped this simplifies a lot
$a*b = ab$	Voilà, it seems that our initial (complex) function can be reduced to the result of a simple multiplication

Let's use a simple example to make it clearer:

5*8 = f(5+8)-f(5-8)
5*8 = f(13)-f(-3)
5*8 = 13*13/4 - (-3)(-3)/4
5*8 = 169/4 - 9/4
5*8 = 42.25 - 2.25
5*8 = 40

Our pre-built list contains the values for $x^2/4$ for all x's in the range of -128 up until +127. So the constraints for the registers `A` and `D` in our program are -128 <= `A+D` <=127 and -128 <= `A-D` <=127. If we are sure that `A` and `D` are always between -64 and +63 these conditions will always be true.

A little remark however for those who saw the light. Our lookup table doesn't contain $x^2/4$ but $int(x^2/4)$! So for the case of 3 we don't get 2.25 but 2 as result of the lookup. Rest assured however that this doesn't affect the final result.

I hope that this article has made it clear that, even on a slow CPU, fast solutions exist to what seem to be long and tedious problems. Just as in real life, there are multiple solutions to one and the same problem, every solution having its good and bad sides. For those who were looking for a fast multiplying routine, feel free to use the given solution. I'm waiting to see the results of *your* imagination.

previous:
Realms of Adventure

MSX Computer & Club Webmagazine
issue 92, March/April/May 2000

next:
De Maiskoek/Bits and Pieces