2018 March 19

Z80 arithmetic: also surprisingly terrible

TAGS

programming

blog

languages

cowgol

all pages / all tags

PUBLISHED

19 March 2018

ABOUT

RSS feed

dg@cowlark.com

@hjalfi

+DavidGiven

davidgiven

Introduction

After getting the 6502 code generator for my Cowgol compiler working pretty well, I've been working on a code generator for the Z80, targeting CP/M. Oh, boy. I thought the *6502* was hard to generate code for…

> **Main Cowgol page**
>
> See the main Cowgol page for more information (and downloads and demo disks).

Problem statement

The 6502's bad enough, but if you squint at it hard enough, it looks like an orthogonal architecture. It's only really got one register, which is A, plus a couple of index registers. Operations work against A and either memory or an immediate. And that's about it.

> **On 6502 arithmetic**
>
> See much the same writeup, but for the 6502.

At first glance the Z80 looks significantly more capable. Seven eight-bit registers! Two dedicated 16-bit index registers! 16-bit arithmetic on register pairs! But then you look more closely, and you find the weirdness…

- Yes, it's got seven eight bit registers. But you can only really do arithmetic on one of them, A. (You can increment and decrement the others.) And only A can be directly written to or from memory! Which is weird, because you can load or store register *pairs*…

register pairs or vice versa. (At least, not without using undocumented instructions.)

- Yes, there's 16-bit arithmetic... well. You can do 16-bit addition and subtraction (and increments and decrements), and only on some registers. Want to do AND or OR? You have to move the values through A.
- There is no variable indexing (like the 6502's `LDA (ptr), Y` addressing mode) at all. The index registers support a constant -128..127 displacement only.
- There are no ALU operations with direct memory addressing modes (like the 6502's `ADC abs` addressing mode) at all (but you can indirect through HL, IX or IY --- and indeed, have to).

Let's try some examples.

Note that Cowgol's a 3op memory machine, so variables live in memory and are only cached in registers. So, all my examples below start and end with values in memory, so that I can compare like with like. In real life, values from one expression will remain in registers for the next, and so won't need to be reloaded; the examples are all deliberately choosing the worst possible case.

Also note that I'm not an expert on the Z80 and there will most likely be stupid bugs (the 6502 version of this had many). This article is essentially me thinking out loud.

Simple 8-bit arithmetic

Let me try `x := y + z`.

```
13   3A 00 ZZ   ld a, (z)
 4   47         ld b, a
13   3A 00 YY   ld a, (y)
 4   80         add a, b
13   32 00 XX   ld (x), a
(11 bytes, 47 cycles.)
```

This immediately shows the problem with registers: the left-hand-side of the addition needs to be in A; so the right-hand-side needs to be somewhere else... but

It actually seems to be cheaper to do this, instead:

```
13   3A 00 YY   ld a, (y)
10   21 00 ZZ   ld hl, z
 7   86         add a, (hl)
13   32 00 XX   ld (x), a
(10 bytes, 43 cycles.)
```

While the Z80 doesn't have direct memory access for ALU operations, it does allow indirection through HL (and also IX and IY); so by loading the location of `z` into HL, we get to do this instead, which avoids using registers other than B at all.

However... while the first version is slightly larger and slower, it may not be as bad as it looks; it has the side effect of leaving B containing `z`. If `z` is used again in the same basic block, it's already there in a register for us, so we don't have to reload it. I'll have to experiment to see whether this actually has any benefits.

(Incidentally, the 6502 can do this in 10 bytes and 14 cycles. The Z80 is *terrifyingly* slow.)

Complex 8-bit arithmetic

Long-term readers (har har) will know that Cowgol supports indirection addressing modes. That is, the backend can do `*x := *y + *z` as a basic operation. This can be faked by manually dereferencing the variables into temporaries before doing the arithmetic, but that's painfully expensive. How do I do this on the Z80?

Well, there are three index registers, and for loads and stores only, you can use BE and DE as index registers too. (But it's slow.) So, this would work:

```
16   2A 00 ZZ      ld hl, (z)
20   ED 4B 00 YY   ld bc, (y)
 7   0A            ld a, (bc)
 7   86            add a, (hl)
16   2A 00 XX      ld hl, (x)
 7   77            ld (hl), a
(13 bytes, 73 cycles.)
```

three parameters loaded for use later (and the result in A).

Constant indexing, such as `x[1] := y[2] + z[3]`? For this I need IX and IY.

```
20   DD 2A 00 ZZ   ld ix, (z)
20   FD 2A 00 YY   ld iy, (y)
19   DD 7E 02      ld a, (ix+2)
19   FD 86 03      add a, (iy+3)
20   DD 2A 00 XX   ld ix, (x)
19   DD 77 01      ld (ix+1), a
(21 bytes, 117 cycles.)
```

177 cycles, ouch.

Simple 16-bit arithmetic

The Z80 supports 16-bit arithmetic! Finally, I can avoid doing things a byte at a time!

Well, not always.

Considering `x := y + z`, but with 16-bit values:

```
16   2A 00 YY      ld hl, (y)
20   ED 4B 00 ZZ   ld bc, (z)
11   09            add hl, bc
16   22 00 XX      ld (x), hl
(11 bytes, 63 cycles.)
```

(For comparison, the naive 6502 16-bit addition was 19 bytes and 26 cycles.)

Which is eminently reasonable. But let's try that with `x := y & z` instead...

```
16   2A 00 YY       ld hl, (y)
20   ED 4B 00 ZZ    ld bc, (z)
 4   7C             ld a, h
 4   A0             and b
 4   67             ld h, a
 4   7D             ld a, l
 4   A1             and c
 4   6E             ld l, a
16   22 00 XX       ld (x), hl
(16 bytes, 76 cycles.)
```

Again, urrgh... but it's not as scary as it looks; all the little 8-bit operations are really cheap 4-cycle instructions. All those together take less than twice the time of

registers for `x` and `y` entirely, by doing this.

```
16   2A 00 ZZ       ld hl, (z)
13   3A 00 YY       ld a, (y+0)
 4   A5             and l
13   32 00 XX       ld (x+0), a
13   3A 01 YY       ld a, (y+1)
 4   A4             and h
13   32 01 XX       ld (x+1), a
(17 bytes, 76 cycles.)
```

Exactly the same time as the previous version, interestingly, but one byte longer.

Complex 16-bit arithmetic

But pointers…

The Z80's 16-bit arithmetic only works on registers, and doesn't have any other addressing modes whatsoever. I have the choice of either dereferencing the pointers into registers, or trying to work a byte at a time.

Consider `*x := *y + *z` for the dereferencing case:

```
16   2A 00 ZZ      ld hl, (z)
 7   4E            ld c, (hl)
 6   23            inc hl
 7   46            ld b, (hl)
16   2A 00 YY      ld hl, (y)
 7   5E            ld e, (hl)
 6   23            inc hl
 7   56            ld d, (hl)
 4   EB            ex de, hl --- swaps DE and HL
11   09            add hl, bc
 4   EB            ex de, hl
16   2A 00 XX      ld hl, (x)
 7   73            ld (hl), e
 6   23            inc hl
 7   72            ld (hl), d
(21 bytes, 127 cycles.)
```

Dereferencing needs to happen a byte at a time; HL is the fastest and smallest register for this. `ex de, hl` lets us very quickly swap DE and HL, which allows me to load the left-hand-side of the expression into DE and then swap in to HL only once I've finished using HL for the dereference. (Remember that physically copying a 16-bit register pair takes eight cycles and two bytes.)


```
20   DD 2A 00 YY   ld ix, (y)
19   DD 6E 00      ld l, (ix+0)
19   DD 66 01      ld h, (ix+1)
20   DD 2A 00 ZZ   ld ix, (z)
19   DD 4E 00      ld c, (ix+0)
19   DD 46 01      ld d, (ix+1)
11   09            add hl, bc
20   DD 2A 00 XX   ld ix, (x)
19   DD 75 00      ld (ix+0), l
19   DD 74 01      ld (ix+1), h
(31 bytes, 185 cycles.)
```

Hell no. It does leave useful values in registers, particularly if I used both IX and IY, but… no.

So can I do it a byte at a time? This requires three simultaneous index registers, for the LHS, RHS and destination.

```
16   2A 00 ZZ      ld hl, (z)
20   ED 4B 00 YY   ld bc, (y)
20   ED 5B 00 XX   ld de, (x)
 7   0A            ld a, (bc)
 7   86            add a, (hl)
 7   12            ld (de), a
 6   23            inc hl
 6   03            inc bc
 6   13            inc de
 7   0A            ld a, (bc)
 7   8E            adc a, (hl)
 7   12            ld (de), a
(20 bytes, 116 cycles.)
```

That's shorter, faster *and* easier to read than the example using the native 16-bit arithmetic! And will work just as well for things which aren't addition and subtraction.

(Notice that this is using `add a, (hl)` for the low byte and `adc a, (hl)` for the high byte. Unlike the 6502, the Z80 has both carry-propagating and non-carry propagating adds (and subtracts).)

But now… the hard one. `x[1] := y[2] + z[3]`.

There are two options. Firstly, use IX and IY. Unfortunately I need *three* index registers to make this work, and the Z80 only has two. But we only need two, as we have enough spare registers to stash the result so we can write it back afterwards…

```
20   DD 2A 00 YY   ld ix, (y)
20   FD 2A 00 ZZ   ld iy, (z)
```


```
19   FD 8E 04      adc a, (iy+4)
20   DD 2A 00 XX   ld ix, (x)
19   DD 71 01      ld (ix+1), c
19   DD 77 02      ld (ix+2), a
(31 bytes, 178 cycles.)
```

That's a serious lot of code and cycles, but it does have the big advantage that we don't need to mutate the pointer registers and we get the offsetting for 'free'.

The alternative is to bake the offsets into the pointers. This is surprisingly annoying on the Z80. If the destination is HL, you can do `ld bc, <offset>; ld hl, <pointer>; add hl, bc`, but if it's not HL then you have to physically copy the result or use `ex hl, de`. Plus, you need the third register pair for the offset, so in practice, really only HL and DE are useful here. (Or IX and IY, which work just like HL, but we're trying to avoid those).

It *is* possible to offset a pointer through A…

```
16   2A 00 XX      ld hl, (x)
 4   7D            ld a, l
 7   86 42         add a, 42
 4   6E            ld l, a
 4   7C            ld a, h
 7   8E 00         adc a, 0
 4   67            ld h, a
(11 bytes, 46 cycles.)
```

…but it's usually not cheap enough.

Interestingly, sdcc uses *all* these techniques jumbled together. Trying on the C code `x[10] = y[11] + z[12]` (the offsets chosen to avoid cheap optimisations like using `ld de, (value); inc de; inc de`), I see it do this:

```
13   3A 00 XX      ld a, (x+0)
 7   86 14         add a, 20
 4   5F            ld e, a
13   3A 01 XX      ld a, (x+1)
 7   8E 00         adc a, 0
 4   57            ld d, a

20   FD 2A 00 YY   ld iy, (y)
19   FD 4E 16      ld c, (iy+22)
19   FD 56 17      ld d, (iy+23)

20   FD 2A 00 ZZ   ld iy, (z)
19   FD 6E 24      ld l, (iy+24)
19   FD 66 25      ld h, (iy+25)

11   09            add hl, bc
```


```
 7   12            ld (de), a
 6   13            inc de
 4   78            ld a, b
 7   12            ld (de), a
(40 bytes, 218 cycles.)
```

This is interestingly poor; sdcc's version of the initial setup of DE through A is longer and slower than mine; and it's pointlessly copying the result into BC even though it's not being used. Just blindly using IX/IY is actually faster.

So: my plan is: use register pair indexing if there are no offsets; use index registers if there are; use 16-bit arithmetic only if we can cheaply load the values into register pairs. Combinations are possible, of course.

Beyond 16-bit arithmetic

Cowgol supports 32-bit arithmetic. I could just extend the versions above, but this seems ideally suited for a loop.

The HL/BC/DE version is, unfortunately, ruled out here because I need an extra register to keep the loop counter in, and I've run out. So I need to use IX/IY as at least some of the index registers. But I can't make use of the offsetted addressing modes, because the offset is based on the number of iterations… so I end up with the worst of all worlds.

```
16   2A 00 YY      ld hl, (y)
10   01 02 00      ld bc, 2  --- LHS offset
11   09            add hl, bc
 4   EB            ex hl, de --- move LHS to DE

20   DD 2A 00 XX   ld ix, (x)
10   01 01 00      ld bc, 1  --- destination offset
15   DD 09         add ix, bc

16   2A 00 ZZ      ld hl, (z)
10   01 03 00      ld bc, 3  --- RHS offset
11   09            add hl, bc

 7   06 04         ld b, 4   --- number of iterations
 4   B7            or a      --- clear carry flag

                  .loop
 7   1A            ld a, (de)
 7   8E            adc a, (hl)
19   DD 71 00      ld (ix+0), a
 6   13            inc de
```


```
(30 bytes, 12+73+73+73+78 = 422 cycles.)
```

Codewise that's a behemoth, but it's a lot shorter and not much slower than the unrolled version. It's also generalisable to all kinds of arithmetic and can, of course, be easily made shorter if my offsets are zero.

(`djnz` is a useful instruction which decrements B and branches if non-zero. It makes quick-and-dirty loops like above very cheap. Unfortunately it's hard-coded to B.)

Crazy stuff

Of course, part of the problem is that Cowgol is fundamentally a 3op architecture, while the Z80 kinda likes 2op architectures. It doesn't like computing `x := y + z`; it prefers to do `x := y; x := x + z`.

Having only two parameters means I only need two index registers, which fits a lot better; I can now use HL/DE with B as a counter, or IX/IY.

But Cowgol doesn't generate 2op code, and even if it did it would rely heavily on fast copies. So… how fast can I make a copy?

Let's consider `x[1] := y[2] + z[3]` for 32-bit values (the worst case). The first thing we want to do is copy `y[2]` to `x[1]`.

```
16   2A 00 YY      ld hl, (x)
10   01 02 00      ld bc, 1
11   09            add hl, bc
 4   EB            ex hl, de

16   2A 00 ZZ      ld hl, (y)
10   01 03 00      ld bc, 2
11   09            add hl, bc

10   01 04 00      ld bc, 4
21   ED B0         ldir       --- 16 cycles on last iteration
(20 bytes, 80+21+21+21+16=159 cycles.)
```

`ldir` copies BC bytes from HL to DE.

Once we've done this, we know that our destination contains a copy of the LHS, so we can do `x[1] := x[1] + z[3]`. Plus, as we know that DE contains a pointer


```
 6   1B            dec de   --- this is actually
 6   1B            dec de   --- faster and shorter
 6   1B            dec de   --- than trying to
 6   1B            dec de   --- subtract 4

16   2A 00 ZZ      ld hl, (z)
10   01 03 00      ld bc, 3  --- RHS offset
11   09            add hl, bc

 7   06 04         ld b, 4   --- number of iterations
 4   B7            or a      --- clear carry flag

                  .loop
 7   1A            ld a, (de)
 7   8E            adc a, (hl)
 7   12            ld (de), a
 6   13            inc de
 6   23            inc hl
13   10 F9         djnz loop --- 8 cycles last time round
(21 bytes, 68+46+46+46+41 = 247 cycles.)
```

So the two together make up 20+21=41 bytes and 159+247=406 cycles --- the 30p version was 38 bytes and 422 cycles. So this is a little longer, but actually faster! (The index registers are painfully expensive.)

Does this trick work with 16-bit arithmetic? Well, yes, but the results are poor.

Conclusions?

[shrug] Ehhh?

No, I really didn't have anywhere I was going with this. I think I've successfully worked through a bunch of strategies, and I now reckon I know what to do for the code generation.

That doesn't mean I like it, though. Are there any cunning tricks I've missed? Please comment.

Addendum

means I only need two index registers to do the work. Writing back the result, if necessary, happens only after I don't need the two index registers any more, and so I can free them.

This actually works pretty well. There's a huge, horrible, and probably bug-ridden state machine, to make it all happen, but it works.

This has made me rethink 32-bit arithmetic. I *could* stash the result in two register pairs. At least one source register would have to be an index register but it's actually doable. I'd end up with (for `x[1] := y[2] + z[3]`):

```
20   DD 2A 00 YY   ld ix, (y)
20   FD 2A 00 ZZ   ld iy, (z)

19   DD 7E 02      ld a, (ix+2)
19   FD 86 03      add (iy+3)
 4   6F            ld l, a

19   DD 7E 03      ld a, (ix+3)
19   DD 8E 04      adc (iy+4)
 4   67            ld h, a

19   DD 7E 04      ld a, (ix+4)
19   DD 8E 05      adc (iy+5)
 4   5F            ld e, a

19   DD 7E 05      ld a, (ix+5)
19   DD 8E 06      adc (iy+6)
 4   57            ld d, a

20   DD 2A 00 XX   ld ix, (x)
19   DD 75 01      ld (ix+1), l
19   DD 74 02      ld (ix+2), h
19   DD 73 03      ld (ix+3), e
19   DD 72 04      ld (ix+4), d
(52 bytes, 304 cycles.)
```

That's worst case. If one of the source registers or the destination can be indexed with HL, BC or DE, or can be loaded directly, it's a lot cheaper. Consider `x := y + z`:

```
10   21 00 ZZ      ld hl, z

13   3A 00 YY      ld a, (y+0)
 7   86            add (hl)
 4   4F            ld c, a
 6   23            inc hl

13   3A 01 YY      ld a, (y+1)
 7   8E            adc (hl)
 4   47            ld d, a
```


```
 7   8E             adc (hl)
 4   5F             ld e, a
 6   23             inc hl

13   3A 03 YY       ld a, (y+3)
 7   8E             adc (hl)
 4   57             ld d, a

10   21 00 XX       ld hl, x
 7   71             ld (hl), c
 6   23             inc hl
 7   70             ld (hl), b
 6   23             inc hl
 7   73             ld (hl), e
 6   23             inc hl
 7   72             ld (hl), d
(36 bytes, 180 cycles.)
```

(In this specific case I could do the addition using two 16-bit adds, but I'm not going to consider that here because of complexity.)

The worst case is two thirds of the cycles of the looped version, at the expensive of being ten bytes longer. The best case is faster or shorter than any of them. The two biggest advantages for me, though, is that not only do I get to reuse exactly the same logic needed for 16-bit arithmetic (which I already have), but I get offsets for 'free'.

It's tempting to try and load the RHS into the destination registers using cheap 16-bit instructions ahead of time:

```
20   ED 43 00 ZZ    ld bc, (z+0)
20   ED 53 02 ZZ    ld de, (z+0)

13   3A 00 YY       ld a, (y+0)
 4   81             add a, c
 4   4F             ld c, a
 6   23             inc hl

13   3A 01 YY       ld a, (y+1)
 4   8A             adc a, d
 4   47             ld d, a
 6   23             inc hl

13   3A 02 YY       ld a, (y+2)
 4   8B             adc a, e
 4   5F             ld e, a
 6   23             inc hl

13   3A 03 YY       ld a, (y+3)
 4   8A             adc a, d
 4   57             ld d, a
```


```
 6   23            inc hl
 7   73            ld (hl), e
 6   23            inc hl
 7   72            ld (hl), d
(41 bytes, 198 cycles.)
```

But this actually ends up *more* expensive! Those cheap 16-bit loads aren't nearly as cheap as they look.

User comments on this page (add your own below!)

Marcin Ciura

Apr 13, 2018
06:56 PM GMT+10

Nice article. Your "offset a pointer through A" code can be improved from 11 bytes, 46 cycles to 10 bytes, 43 cycles if the offset fits in one unsigned byte:

ld hl, (x)

ld a, l

add a, 42

ld l, a

adc a, h

sub l

ld h, a

Also, have you thought about optimizing for size by moving every snippet longer than 3 bytes (the size of "call xxxx") from the generated code into your standard library? The user might decide whether to optimize for speed or for size by a command-line flag.

Anonymous Cow

Jun 10, 2018
11:13 PM GMT+10

Have you thought about supporting the 6809, It would be interesting to see how that went...

07:05 AM GMT+11

Well, that's pretty funny way to code for old 8-bit CPUs indeed. They're effectively stack machines. All those extra registers are a tiny optimization on top of that. How one does 16-bit binary + arith for 8080 (or z80) is:

; compute left arg in HL

push hl

; compute right arg in HL

pop de

add hl, de

Cycle counts? Well, it takes how much it takes. We got spoiled by RISC CPUs where an instruction takes one cycle, and operands are loaded over 32-bit, not over 8-bit bus.

And we should enjoy that spoilment, look at older 8-bits with disgust of wasted time/effort, and proclaim once again: BURN, BURN IN HELL, CISC!

kktos

Jan 14, 2019
07:04 PM GMT+11

6502 is more or less a low cost 6800. 6800 has a neat architecture, 6809 is even better :)

I remember my days when I tried programming the Z80 coming from the 6502..... Let's say I didn't do it for long ! :oD

Then I went straight to 6809 and later I discovered the 68000..... ASM programmer dream !

kktos

Jan 14, 2019
07:06 PM GMT+11

oops, I missed the most important point :

David, thanx for this excellent article !

example, with the last parts destination registers what you could do is

ex hl, sp

ld hl, x

push bc

push de

ex hl, sp

with 6502 and z80 I've always felt that you should avoid doing any calculations and focus on precalculating as much as possible, hopefully you have at least 32kb ram or rom to store it in.

Jeremy Barnes

Jan 14, 2019
11:23 PM GMT+11

It has been a long while and I was just a kid, but as I recall from thousands of hours writing z80 assembly was that I used self modifying code all the time. Offsets can be spilled into the immediate bytes of instructions to save registers, as well as things like strides.

Anonymous Cow

Jan 15, 2019
12:15 AM GMT+11

The Z80 is much slower than the 6502 because it does less in each cycle, but typically runs faster (classically, ~4MHz vs ~1MHz). Thus you need a fudge factor when comparing cycle counts - which comes out to about 4, which means it breaks pretty even.

mtkd

Jan 15, 2019
03:59 AM GMT+11

Not looked at Z80 in decades but some of those memory writes could be done faster by moving the stack pointer and using the 11 cycle PUSH BC, PUSH DE etc. then moving it back (it saved a lot of time on video write loops)

Philip

Jan 15, 2019
05:44 AM GMT+11

Osvaldo Doederlein

Jan 15, 2019
08:17 AM GMT+11

Great stuff. However, when you complain that the Z80 was much slower, did you factor in the frequently that each chip was able to reach? I had a ZX Spectrum running at 3.56MHz, while losers on the Apple II or C64 had to crawl at 1MHz ;-)

Looks like that was an early skirmish of the CISCxRISC wars; the Z80 had a very large instruction set with complex encoding, but its decoder and execution logic weren't good enough to make the trade-off unequivocally positive...

Anonymous Cow

Jan 15, 2019
09:08 AM GMT+11

The cycle counts look awful for the Z80 but it was typically clocked at four times the 6502.

I.e. 4 cycles on the Z80 is roughly equivalent to 1 cycle on the 6502.

Anonymous Cow

Aug 27, 2019
04:29 AM GMT+10

Z80 not "slow", but takes several clock cycles by design/intent.

Look carefully: at the same time 6502 clocked at 1Mhz, whereas Z80 clocked at 4Mhz. In 1975, when you counted each transistor on silicon die and when you drew transistors by hands, you end up with transistors limit and your physical abilities to design such complex machine at transistor level. All is not free. Z80 reuses logic in time, needing more complex sequencer. You either have complex design with sequenced logic, or simple design as 6502 with simple sequencer. Choose one.

?

Anonymous Cow

G+
Login

B I U Post

If you log in, you get to use your own name and avatar. Otherwise you're anonymous.

The rules: this is not a public forum; it is my private space. I decide what gets posted here. All comments are held for moderation and may be rejected for any reason, no matter how arbitrary, and there is no appeal. (Complaints about moderation will be discarded out of hand.) If you have a problem about any comment posted by someone else, email me.