The Amazing Zilog Z80

Jun 15, 2016

In the past few months I've become interested (or rather: obsessed) with 8-bit home computer emulation and 'computer archaeology', which is just a fancy term for sifting through early-90's web pages, long abandoned USENET groups and OCR'ed PDFs of old computer manuals and circuit diagrams.

One result of this obsession is a new emulator for East German 8-bit computers, running in the browser via emscripten: http://floooh.github.io/virtualkc/.

Originally I wanted to write a blog post about how fascinating this all is from a non-technical perspective, the cultural and historical aspects, how important it is to preserve the 8-bit heritage, how those early computers with their detailed hardware- and programming-manuals encouraged kids to dive into programming and technology, and how all this was lost during the dark ages of the mid- to late-90's etc etc etc… but I really suck at stuff like this, it would be an endless rambling.

So instead I'll try to write a little series of blog posts about the technical aspects of vintage computer emulation. There's a lot to learn for the modern-day programmer from those old home computers, which basically have been complete game programming frameworks cast into hardware, even if it's just gawking in awe and wonderment how much was achieved with how little code in those days.

Heart and Soul

The heart of each emulator is the CPU. It might not be the most complicated part, but definitely the most tedious and boring to write, mostly because there's nothing to see for a long time. An emulated CPU that's just hooked up to some memory and running small test programs is like a brain in a jar, it may 'live', but without being connected to the world through hands, eyes, ears and mouth it doesn't make a human. The same with an emulated CPU. It may work and process instructions, modifying it's internal state, but without a soul - the system ROM - and a way to communicate with the outside world it won't be an emulator.

And the CPU emulation needs to be nearly complete to successfully boot a system ROM, and this development phase costs many uneventful evenings, but that moment when the emulator first boots up and 'something' happens on the screen, even if it's just flickering pixel garbage… that moment is unforgettable, and from then on everything else seems easy, even if the really tricky problems are still ahead.

The 8-bit Wars

So, where was I… the **Z80**! Heir to the Intel 8080, archnemesis of the M6502, firestarter of the first battles in the eternal fanboy war: Z80 vs 6502, Commodore vs Atari, Intel vs Motorola, PC vs Mac, Xbox vs Playstation!

The Z80 was one of the two 'big' 8-bitters of the late 70's and early 80's, with the 6502 being the other. It is interesting that Intel was pretty much absent from the homecomputer arms race. The world was split into two, the Z80 with Sinclair, Amstrad and MSX machines on one side, and the 6502 with Commodore, Apple and Atari on the other.

The two CPUs had very different design philosophies, everything about the Z80 was big and brute force, it had a lot of registers, a huge instruction set and ran at up to 4 MHz. The 6502 on the other side was small, cheap and (I have to admit even as a Z80 fanboy) - elegant, with the first glimpses of RISC visible on the horizon.

Oh, and the Nintendo Game Boy and Sega Master System had a Z80 too, so I think it's safe to say that the Z80 won the 8-bit wars in the end by sheer number.

To understand the Z80 it makes sense to first look at its illegitimate father, the Intel 8080:

From the i8080 to the Z80

The Intel 8080 was released in 1974, and at first glance it looks like a Z80 cut in half. In reality the Z80 is of course more like two Intel 8080’s duck-taped together.

The i8080 already has the paired 8-/16-bit register set that’s still vaguely perceptible in modern x86 CPUs 42 years later, and which should look strangely familiar to everyone who’s dabbled in x86 assembly coding:

```
A  F
B  C  => BC
D  E  => DE
H  L  => HL

         SP
         PC

flags (F): SZ-H-P-C
```

On the left side are the eight 8-bit registers: A,F,B,C,D,E,H,L. Two of those are special: A is the accumulator, this is where the result of algorithmic instructions is stored. F is the flag register, which contains status bits about the result of previous instructions (for instance if the result is zero or negative, or a carry-over happened).

The remaining 8-bit registers are grouped into 3 16-bit register pairs (BC, DE, HL), with HL being specialized as general memory pointer. This leaves the two special 16-bit registers SP (the stack pointer) and PC (the instruction pointer).

8080 instructions were encoded in a single byte, with all 256 instruction slots used. Instructions could be followed by one or two bytes of immediate operands.

In **1974**, a special-projects engineer escaped Intel’s secret underground facilities and defected to the Soviet Union, where he founded OKB-Zilog to work on a secret weapons program leading to the development of the first bipedal tank, a ‘metal gear’.

Ok I made that up, the Intel engineer was actually Federico Faggin, who lead the i4004 design group (the world’s first commercially available microprocessor), and he didn’t defect to the Soviet Union of course, but founded Zilog as an American company.

The Z80 was released in 1976 and built to be an 8080 killer: fully backward compatible in order to run the dominating CP/M operating system, but with a much more powerful instruction set and more registers:

```
A  F           AF'
B  C => BC     BC'
D  E => DE     DE'
H  L => HL     HL'

I       IX
R       IY
        SP
        PC

flags (F): SZ-H-PNC
```

There's a new "shadow register bank" AF',BC',DE' and HL'. These registers are not directly accessible, but can be swapped very quickly with their main bank counterparts, for instance, the EXX instruction swaps BC,DE and HL with BC',DE' and HL', all at once in just 4 cycles which is way faster then writing or reading even a single register from memory.

Next there's two new 16-bit 'index registers' IX and IY. These enable a powerful new 'indirect with offset' addressing mode. The IX or IY register would serve as base pointer, and an 8-bit offset which is directly encoded in the instruction is added to yield the final 16-bit address.

The remaining register bank enhancements are fairly obscure: the I register is the 'interrupt vector' to locate the interrupt handler jump table, and the 'R' register is the 'memory refresh counter', and there's a new flag bit 'N' indicating whether the last instruction was a substraction. All of these are not very interesting at the moment.

The next big change was the enhanced instruction set, and this is where things get interesting, especially for emulator writers:

Remember that the Intel 8080 had single-byte opcodes, and all 256 possible instruction codes were used. How would the Z80 go about cramming additional instruction slots into an already completely filled up bucket? And how would all those additional instructions be added without increasing the chip’s transistor count multiple times?

The answer looks easy and straightforward on the surface, but under the hoods awaits a mind-blowing (at least to me) engineering marvel of how to make the most out of very limited resources, hidden in the dark and undocumented corners of the CPU.

A Hack Within A Hack Within A Hack

The simple and obvious answer how to extend the instruction set is: through multi-byte instructions. Instead of encoding all instructions into a single byte, there are now opcodes with 2 or even 3 instruction bytes. These could be followed by up to 2 immediate-operand bytes, the longest instructions can be 4 bytes long and take 23 cycles to complete (the fastest 1-byte instructions take 4 cycles).

But wait, how would such a multi-byte instruction work if the Intel 8080 already had all 256 possible instruction slots filled?

The answer is the first inception level: some of those slots were redundant and implemented the same instructions. They were ‘alternative opcodes that should not be used’.

The 8080 had a number of those redundant slots: 00 is the offical NOP instruction (doing nothing), but 08, 10, 18, 20, 28, 30 and 38 were also NOPs. All of these alternative NOPs were filled with new 1-byte opcodes on the Z80 (mostly relative jumps), more interesting were 4 alternative opcode bytes in the upper instruction range:

The jump-to-address instruction **JMP nn** is at opcode hex C3 (both on the i8080 and Z80), but on the i8080, the opcode CB *also* implements JMP nn. And the **CALL nn** instruction (which calls into a subroutine) at CD even has 3 alternatives at DD, ED and FD.

On the Z80, those 4 byte values CB, DD, ED and FD are used as lead bytes for the extended multi-byte instructions (of course if an Intel 8080 program made use of those alternative opcodes it wouldn’t run on a Z80).

The Zilog designers made good use of the multibyte instructions:

The **CB** prefix added a complete set of bit shift and rotate instructions (today these are common but totally absent in the 8080), and filled the rest with the BIT, RES and SET instructions which tested, cleared and set individual bits.

The **DD** and **FD** prefixes added instruction that used the new index-register plus offset addressing modes (DD was for the IX register and FD for the IY register).

Finally, the **ED** multi-byte instruction group added the so called 'block instructions', and a smaller number of left-overs, like 16-bit addition, subtraction and load/store instructions.

The 'block instructions' were little CISC monsters. Take for instance the LDIR instruction, which means 'Load, Increment and Repeat':

It takes a source address in HL, a target address in DE, and a 16-bit count in BC. When executed, LDIR will load a byte from the address pointed to by HL, and store it at the address pointed to by DE. It will then increment HL and DE, decrement BC and repeat the whole load/store/increment/decrement dance until BC is 0. An entire memory block could be copied in a single instruction! Copying one byte this way cost 21 cycles though so it wasn't exactly fast, for a few dozen kilobytes LDIR would take multiple seconds to complete.

And now the last question of the puzzle: How could the Z80 designers afford to increase the instruction count several times without creating a monster chip? According to Wikipedia the 8080 had about 4.5k and the Z80 had 8.5k transistors, not even twice as many (unrelated side note: the 6502 had around 3.5k transistors).

The answer lies in the undocumented instruction set:

The Dark Side of the Z80

The undocumented parts of the Z80 are huge and manifold. While the Intel 8080 had just 11 'undocumented' instructions, the Z80 instead introduced **hundreds** of new undocumented instructions along with 2 undocumented flag bits (at position 3 and 5), which haven't been fully understood by emulator writers until well into the 2000s.

The undocumented instructions in the extended FD and DD ranges are especially interesting, they explain how all those indexed instructions using the IX and IY registers could be implemented with minimal additional chip space:

Looking at the *documented* index instruction it is clear that the index registers IX and IY are somehow related to the HL register.

For instance the opcodes to load a 16 bit value into HL, IX and IY look quite similar:

```
LD HL,nn       => 21 n n
LD IX,nn       => DD 21 n n
LD IY,nn       => FD 21 n n
```

The opcode byte 21 is the same, the extended versions just have the prefix byte DD and FD in front.

The same relation exists for instructions that use the new index addressing modes, for instance:

```
LD (HL),n      => 36 n
LD (IX+d),n    => DD 36 d n
LD (IY+d),n    => FD 36 d n
```

Again, the base opcode 36 is identical, just the prefix byte, and the offset ‘d’ are added.

All basic 1-byte opcodes that somehow use HL have their counterparts where IX and IY are used instead of HL.

But those DD and FD instruction groups have huge holes in them. Big areas seem to be simply unused in the official Z80 documentation. What lies beneath those unused areas?

The answer is the second inception level: all those ‘unused’ instruction slots are redundant alternatives of ‘unextended’ base instructions!

For instance the hex opcode C6 is ADD A,n (add an 8-bit immediate value to the accumulator A):

```
ADD A,n    => C6 n
```

The prefixed opcode DD C6 is undocumented, what does it do? The answer is: ADD A,n! It is absolutely identical to the 1-byte instruction which adds an 8-bit immediate value to A, the same is true for FD C6:

```
DD C6 n     => ADD A,n
FD C6 n     => ADD A,n
```

Those instructions burn more cycles than their official 1-byte counterpart, so why do they even exist, why did the engineers 'put them there'? Why waste transistors for hundreds of completely useless instructions?

The answer can only be that the designers made clever use of existing resources and those redundant instructions run on exactly the same transistors as the documented 'main instruction'. It seems that the only purpose of the prefix byte is to 're-wire' the normally used HL register to IX or IY, and whenever HL would be used as memory pointer, fetch and add an additional offset byte to IX or IY. Apart from that the indexed instructions are completely identical to their unextended counterparts, they just take longer because loading the prefix byte and loading and adding the offset needs more cycles.

The whole cleverness of that instruction re-use becomes much clearer when actually implementing those parts in an emulator, since the prefix byte for extended instructions becomes quite literally 'prefix code' which in the end calls into the same code that actually implements the instruction logic as the unprefixed 'main instruction', but more on that in the next blog post which will be about the Z80 emulator itself!

Oh, and there's also the fascinating story of the Z80's mysterious East German brother, the U880, but this will also have to wait for another time :)

The Brain Dump

The Brain Dump
floooh@gmail.com

floooh
flohofwoe

This is the blog and personal web page of Andre Weissflog (Floh, floooh, flohofwoe) mostly about programming stuff.