Managing registers/memory effectively on the Z80

Asked 7 months ago Modified 6 months ago Viewed 3k times

I've been writing assembly for the Z80 for some time, lately. Initially, I had no experience with working with 8-bit processors, but after going through a tutorial for my platform and spending a lot of time studying an opcode table, I believe I have a strong grasp of the instruction set: which registers can be used with which operations, their respective sizes and cycles, and so on. In theory, given that I have no problem with assembly in general, I ought to be able to write Z80 programs no problem.

However, every time I try to write non-trivial routines, I find myself struggling with my registers. Most 8-bit operations have to go through A, which isn't a big deal - I just swap things in and out of A as necessary. But I almost always have a pointer sitting in HL, so I'm down to B, C, D, and E for any other values. But if I need to do some pointer arithmetic, I've got to dump that in BC or DE so now I'm down to two registers! If B is a loop counter, I'm down to C. Most of my subroutines need more persistent values than one or two 8-bit registers and a pointer, so I've got to throw in some pushes and pops.

So far, this still seems manageable. But I find myself thinking more about which registers are in use and what the stack looks like than the actual logic of my code. If I realize halfway through the routine that I need an extra value, I have to redesign my register usage. If I ever make a tweak later on, I suddenly need to reorganize my stack usage to get the necessary value on top. Well, perhaps I could save things in RAM more often. Sure, it's a little slower and bigger, but most routines aren't that speed critical. But that has to go through A as well for byte values. More swapping, more pushing. What about index registers? Faaaat and sloooow, and really don't help any more than memory does.

In essence, I struggle with technique. I feel more like a juggler than a programmer because I have to spend so much time deciding where in the CPU to throw all of my values, and my coding speed decreases by an order of magnitude. I've never found a tutorial or article ever deal with this topic when covering the Z80 (or any other 8-bit CPU), but it seems so critical to me. Hence, I'm taking it here to find people who (hopefully) have struggled with this and came up with a way to deal with it.

To sum it up: For people who have actually used the Z80 or CPUs with similar situations, how do you manage your registers, memory, and stack in an effective way? One that allows you to focus on the code without having to constantly decide where to move values and/or reorganize or rewrite existing code.

programming assembly z80

Share Improve this question Follow

edited Feb 17 at 13:46

TonyM
2,995 1 16 27

asked Feb 16 at 23:47

v-rob
403 2 6

6 Don't forget about the alternate register set. When I was a teenager in the '80s I used them a lot. – hippietrail Feb 17 at 4:23

16 Congratulations! You've just arrived in Z80 assembly land. Yes, that's what it is all about. One thing: Don't be afraid of putting stuff into memory - That's probably something you've learned from modern CPUs that are way faster than the Z80. In comparison with the CPU, memory on old systems is generally appropriately fast. – tofro Feb 17 at 12:36

1 For the record, I voted to close as the question seems very broad with no clear single answer. Since it's just me so far, I may be the one out of step. – Tommy Feb 17 at 13:24

7 I guess it's a matter of perspective: I learned Z80 assembly in the early 80s after learning 6502 assembly. I remember thinking "*wow, look at *all those registers!*". – Theodore Feb 17 at 15:52

3 "*that allows you to focus on the code without having to constantly decide where to move values and/or reorganize or rewrite existing code.*" We haven't had a reply to the much-answered question: why are you programming a Z80 then? Programming a Z80 is all about having to hand-craft the source for more complicated algorithms. Tuning it for speed, planning it in your head long before writing it. Doing something within tight constraints really teaches that subject, whether you then get away from it, stick with it or move onto a further form of it. Anyway: why are you programming a Z80? – TonyM Feb 17 at 20:37

6 Answers

Sorted by: Highest score (default)

45

It's quite a while since I did any Z80, but there's an important difference between classic Z80 systems and modern kinds of computer.

Memory is fast.

Unlike modern systems where memory accesses take tens of clock ticks, memory is nearly as fast as registers. Keeping variables in registers is thus largely unnecessary: there aren't enough registers, and their specialised uses make it awkward, as you've noticed.

For initial coding, use memory for all variables. Use the registers for arithmetic, and the data you need to access and modify those variables: the most notable example is the usual need to keep a pointer in HL. For this kind of architecture, trying to write your code to use registers heavily from the start is premature optimisation.

When you have your code working, *then* you should look at how to make it smaller and/or faster by using more registers. Given the overheads of fetching long and complex instructions, and the limited RAM of any 8-bit system, optimising for smaller code is usually the best idea. Raffzahn's answer will tell you a lot about how to do that.

Share Improve this answer Follow edited Feb 17 at 9:37 answered Feb 17 at 0:03

John Dallman
10.5k 2 35 49

4 "memory accesses take tens of clock ticks" and that's if you hit on L2 cache, right? If you go all the way to DRAM then it's hundreds of cycles. Yeah, the concept of "all memory is SRAM, no caches" is kind of foreign to modern programmers. – Nate Eldredge Feb 17 at 15:00

2 Store/reload of local vars is still fairly fast on current CPUs in 2023. It costs extra instructions, but store-to-load forwarding latency of about 5 cycles means out-of-order exec can hide it somewhat. Still, debug builds that spill/reload a loop counter instead of keeping it in a register often run simple loops 6x slower than optimized builds even without SIMD vectorization. Except on Zen 2 or Zen 4, or Ice Lake, where memory renaming can store-forward with zero latency (if the addressing mode is the same); agner.org/forum/viewtopic.php?f=1&t=41 – Peter Cordes Feb 17 at 19:53

2 OTOH, with current x86 CPUs being capable of running 4 to 6 instructions per clock cycle (depending on the microarchitecture and details of which instructions), that 5 cycle store-forwarding latency is a window in which the CPU could be executing up to 30 instructions. So yeah it is pretty important to keep your loop-carried dependency chains in registers; reloading other stuff every iteration is fine, though; with 2/clock loads (or 3/clock on Alder Lake and Zen 3), it's not a disaster. – Peter Cordes Feb 17 at 19:58

3 "For this kind of architecture, trying to write your code to use registers heavily from the start is premature optimisation." I think this is what I really needed to understand. It goes contrary to how I've learned any other assembly. I already knew RAM was fast for processors of this speed, but I still felt that The Right Way (TM) is just to keep things in registers--after all, if the registers are used properly with judicious pushes and pops, you wouldn't need to spill out to RAM very much at all, right? Apparently, I'm quite wrong that this is a good way to do things. – v-rob Feb 17 at 23:24

3 @v-rob No, it is a good way to utilize the registers. That's what they are for. And ultimately going there will produce fast code. Just that's the second step after getting your program to work. I'm absolutely nuts about writing compact and fast code - but that's always second to writing working code. – Raffzahn Feb 18 at 3:51

23

Well, there isn't a simple answer. It's a lot is more about philosophy than hard, Z80 specific advice, but let's look at some points:

The most important thing about Z80 is maybe to forget that it's a Z80 and take it as it is, an 8080 with some additions that sound nice but often don't work out as nice. Remember the index registers being clumsy? Yeah, exactly that. The Z80 was designed as an 8080 clone with most instruction set extensions being more targeted at sales than usage. So there's the reason why most CP/M software stayed with 8080 code, despite the Z80 having replaced it in new machines within less than a year.

The 8080 is well-defined and consistent (for most parts). It's an

- 8 bit
- accumulator based CPU with a
- single 16 bit memory pointer (HL)
- single 16 bit stack pointer
- single 16 bit program counter.

This sounds much like the 6800, doesn't it? Except for the second 16-bit accumulator and indexed addressing operations, that is. For both the 8080 makes up by adding two 8-bit pairs that can hold two additional pointers or up to 4 8-bit values.

It's these commonalities and differences any program needs to be built upon - after all, each and every CPU is built to support a certain coding style, the way the CPU is intended to be used. The Z80 assembler unified the mnemonics to a great extent. Its rather abstract angle simplifies first steps. In contrast 8080 mnemonics are more diverse, if not chatty, presenting more of the idea behind certain instructions. A good example is the already mentioned integrated use of the memory pointer. Where Zilog requires an abstract

```
LD B,(HL)
```

which suggests that there might be ways to use other pointers, Intel marks access via HL by using a pseudo register called M, as in

```
MOV B,M
```

All instructions that accept an 8 bit register as *parameter* will thus also accept a memory pointer in HL. Knowing that makes it clear why some combination in of Z80 parameters will work, others won't.

These are also a large part of the added instructions - when H/L/HL gets replaced by IX(DDh) and IY(FDh) by adding 1 or two bytes and 4 to 12 cycles.

Reading carefully through the instruction section of the original Programming Manual might be more useful than the later Assembler Manual or the Systems Manual, as its instruction section is still organized by encoding and internal working rather than alphabetic sequence.

With that knowledge it will be less confusing to identify which Z80 modifications can be helpful and which are folly.

In addition to above, it might be useful to go for some far more general assembly techniques, foremost among them avoiding premature optimization. This seems to be especially useful when reading your description for registers usage.

- Don't care for input or output ahead or time.
- Use registers always in the same order.
- Don't care for optimization of parameter storage between function calls.
- SP can be loaded into HL to address parameters on stack.
- Stack is cheap.
- Stack is 16 bits wide.
- Push all registers at entry.
- Use 16-bit pointers freely.
- Freeing up a register temporarily is a simple PUSH/op/PULL.
- Remember that *all* 16 bit registers can be added to.
- Remember that adding 0FFFFh is the same as subtracting 0001h.
- All registers used up?
  - Need more storage than registers offers?
  - Need some heap?
  - Set up a stack frame!

    ```
    LD  HL,-40    ; Stack space needed
    ADD HL,SP
    LD  SP,HL
    ```

  - Now there's lots of space to store local data without any stack acrobatics
  - Or set up a stack frame using IX or IY (or both)

  This is one of the few cases where IX/IY becomes really useful, as their implied 8-bit offset is now an fixed address into that stack frame (see below)
- Stack frames may not resolve need for static storage or an independent heap, but does serve all needs for local storage.
- Stack frames are as well a good way to access parameter passing on stack.
  - Except parameter passing on stack is evil.
  - Rather use explicit parameter blocks
  - It's the same effort, but way better readable and less prone to incompatibilities.
- Remember to provide enough stack space
- Don't hesitate to use memory:
  - These are 8 bit CPUs and memory is as fast as the CPU
  - The only down side on a 8080/Z80 is its clumsy addressing
  - IX/IY addressing does resolve a lot of this.

The mentioned stack frames show the importance of understanding the Z80 not as a single CPU design, but as a 8080 design plus a set of modification bolted on.

IX+n or IY+n addressing is a modification of the basic 8080 M addressing. Therefore all of these up to 256 local variables are (at 2 byte and 12 clock penalty) directly accessible in all basic instructions. No need to go through A.

After reading and throughout memorizing the 8080 Programming Manual, a table showing all instructions but making the 8080 to stand out may be helpful. I found this 8080/Z80 Instruction Set table very helpful:

Add Byte Instructions

8080 Mnemonic		Z80 Mnemonic		Machine Code	Operation
ADD	A	ADD	A,A	87	A <- A + A
ADD	B	ADD	A,B	80	A <- A + B
ADD	C	ADD	A,C	81	A <- A + C
ADD	D	ADD	A,D	82	A <- A + D
ADD	E	ADD	A,E	83	A <- A + E
ADD	H	ADD	A,H	84	A <- A + H
ADD	L	ADD	A,L	85	A <- A + L
ADD	M	ADD	A,(HL)	86	A <- A + (HL)
---		ADD	A,(IX+**index**)	DD86**index**	A <- A + (IX+**index**)
---		ADD	A,(IY+**index**)	FD86**index**	A <- A + (IY+**index**)
ADI	**byte**	ADD	A,**byte**	C6**byte**	A <- A + **byte**

It is not only functionally ordered, but also shows both mnemonics and makes thus Z80 only instructions stand out. Noticing them helps to think about added usefulness - or the lack thereof.

Now get your program working.

Only after that is reached should you start to optimize. This as well can be done in several steps, best done top down:

- Look for groups of functions that serve a purpose entered via a single point of entry.
- Have the stack frame created at entry for all of them.
- Do so with IX.
- Use IY for a second frame needed dynamically or only in some parts.
- Reorder functions within this group for commonalities.
- Reduce register saving between them (that goes bottom up).

- Reorganize register (and memory use).
- Reduce saving allocation again.
- Vary register usage for parameters.
- Inline routines that are called from a single place.
- Remember, a call is also a stack operation, needing 4 bytes of code and 2 bytes of stack.

etc. pp.

Oh, one more, not so secret, hint: 8080 supports all jumps as conditional; that includes `CALL` and `RET`.

Share Improve this answer Follow edited Feb 19 at 22:08 answered Feb 17 at 2:03

Raffzahn
207k 21 582 851

2 You might want to add a word about the alternate register set. With some coding discipline it is quite helpful. – the busybee Feb 17 at 7:07

3 I've seen a fair amount of public domain CP/M software that targeted the Z80 rather than the 8080 (being free, there was no commercial imperative to make it runnable on every potential client's system. What were 8080 users going to do, ask for their money back?). Even if they didn't make use of the extra registers, the 16-bit loads and stores for BC and DE, and the block copy operations, came in useful. – john_e Feb 17 at 9:25

2 I'm not doing Z80 now, but back in the day I would have paid good money for the book that expanded this excellent answer out with examples. – Wayne Conrad Feb 17 at 15:28

5 I think a difficulty for learning Z80 nowadays is that all good tutorials are written by people who are apparently optimization fiends. Every recommendation the tutorials gave is the opposite of what I'm getting here--apparently, by the time the wrote the tutorial, they had forgotten what it was like to be a beginner. (Which does bring into question as to how good the tutorials actually are...) – v-rob Feb 17 at 23:08

2 @Raffzahn - please do, but I don't have a robust test put together yet. In an extremely crude test of 1600 unique COM files I had lying around, 30% of them contain the bytes `ED B0`, aka the Z80-only `LDIR` instruction. By comparison, every ZX Spectrum game image I have contains LDIR – scruss Feb 19 at 1:25

8

Raffzahn's answer is excellent.

Having developed a software product at the time – I don't know how many thousands of lines of assembler but the program was huge, about 96KB in size – here are a couple of points worth bearing in mind.

There is no such thing as the Z80

No sane software developer would have attempted to sell software for the Z80. Here is an accurate breakdown of the CPUs actually used by people using CP/M in the early to mid 80s:

- 8080 chip: 1%
- Z80 chip: 1%
- What's a CPU?: 98%

To require a Z80 is to say to your potential customers, "To know whether this package will work on your computer, go away, open up the box, and look at what is written on the big chip in the middle", they will obey you. They will go away.

The one exception would be if you had some very hot and small code deep down in your software -- perhaps floating-point arithmetic or something. Then you would write write the whole thing twice twice, and decide at runtime whether to call the 8080 version or the Z80 version.

Memory is fast

As Raffzahn points out, the main change in computing over the last 40 years is that memory has got slower and slower and slower. Another way of looking at it is that fast memory has essentially not increased in size since the "64KB" days, but while we called it "memory" in the 1980s, now we call it "cache".

There is no such thing as a stack

Stacks are (a) to make the `RET` instruction work after a `CALL` and (b) to save the contents of a register before restoring them a few lines further down. They are for nothing else. *The fact that modern compilers never use it for (b) is a source of perpetual amazement to anyone who was programming back then.*

The program I wrote never used the stack for data. The great thing about assembler is that one can choose how to pass arguments. For "small" functions it was enough to pass them in registers and get results back in registers. As long as you documented what you were doing, it worked well. For "big" functions I used the IBM 360 method of shoving all the arguments into fixed locations in memory and then passing a pointer to that memory in `HL`.

The only reasons for using the stack for data are (a) with recursive function calls, which are never needed in real software, and (b) to add unreliability. If you know how much space is needed for data, reserve the space. If you don't know how much space is needed for data then you have no business writing software in the first place. "If it crashes, double the stack space" may be all right in today's culture of amateurism but it was not appropriate in the days when people expected their software to work.

Of course if you really needed recursion for something, you could allocate a parameter block dynamically and pass **its** address in `HL`. But even memory allocation isn't really needed: just

another way of adding indeterminism and unreliability.

In conclusion

As the aircraft designer said, "Simplicate and add lightness".

Share Improve this answer Follow

answered Feb 17 at 9:02

Martin Kochanski

552 2 3

11 "No sane software developer would have attempted to sell software for the Z80. " - Maybe not for generic CP/M, but the original question never mentioned generic CP/M. For software targeting a specific computer that was built around a Z80 (such as games for a number of widely used 8-bit micros), the developer would have used any features of the Z80 that the platform allowed them to. – john_e Feb 17 at 9:19

2 ... and not just computers. Sega sold at least 10 million Master Systems, which sounds like a potential market that a sane developer might be interested in. – Tommy Feb 17 at 13:23

1 @manassehkatz-Moving2Codidact: If you compare SRAM to DRAM and look at latency, I think it's kind of true. Based on en.wikipedia.org/wiki/Random-access_memory, early 1980s SRAM had latency of about 20-50 ns. That's as good or better than modern DRAM, I believe. – Nate Eldredge Feb 17 at 15:12

1 @NateEldredge From the same page: *From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory speed only improved at 10%.* which supports my statement. If average speed limit (US) increased at the same rate as memory speed, starting at 55 MPH in 1986, we'd be driving at more 200 MPH in 2000. Not the same as MHz to GHz CPU increase, but not insignificant.
– manassehkatz-Moving 2 Codidact Feb 17 at 15:16

1 @manassehkatz-Moving2Codidact: It improved within the SRAM and DRAM categories individually, sure. But we transitioned from mainly using one (on small systems) to mainly using the other. Huge gain in bits per dollar, and bandwidth is great, but we traded off latency. – Nate Eldredge Feb 17 at 15:30

7

Thank you for a lovely question. All answers so far seem to come from the perspective of programming Z80 as 8080 with a few extra commands, so, as *an optimization fiend*, I'd like to add some balance and talk about the practice of writing software for Z80 as the primary target.

In your question you pretty much nailed it regarding register allocation as the main source of difficulty when writing code. It is true. You see it as a bad thing, but a lot of people, including myself, tend to see it as an interesting challenge. So how does one rise up to this challenge? By working in a way that is somewhat counter-intuitive, esp. compared with many modern good software practices. You can be guided by heuristics, some of which I'll try to formulate here:

- Your CPU is weak and you don't have many registers. So, you don't think about your algorithm as a whole, you focus on your innermost loop and try to find solution for your most critical data manipulation. It is easier and it addresses 90% of your needs. The rest of the code can save data onto stack, save/load directly to RAM and even use frame pointers (not really, no-one does it actually). It does not matter much.

- Specific patterns for using registers are not very well defined. You need to learn some important patterns, but they are sometimes contradictory, so you pick a specific pattern for each situation. These are some key ideas:
  - A is your key register, most direct manipulations should get it involved. In other words, think what is the most manipulation intensive part of your inner loop, it should probably done via A. Most maths libraries use A **a lot**. Fastest data decompressors use A to store bit reservoir, because any alternative slows things down.
  - HL tends to be used in several ways. It can be a source pointer for `LDI` / `LDIR`, `LDD` / `LDDR` etc, so if you are planning to use one of these commands, you are pretty much stuck. HL is also very good for setting up LUTs, esp. with two levels of indirection, because you can load values from memory and into any register via HL. I mean, given something in A, you can

    ```
    LD L,A : LD H,high LUT1
    LD E,(HL) : LD D,high LUT2
    LD A,(DE)
    ```

    Last, but not least, if you need to fill memory with fixed data

    ```
    LD HL,... : LD (..),HL
    ```

    is very flexible and one of the faster patterns you can employ.
  - DE is often a destination (because of `LDI` / `LDIR`, `LDD` / `LDDR`, etc), but can also be a useful extra pointer.
  - BC can be a pointer, but given how useful `DJNZ` tends to be, it is often B for the counter and C for storing intermediate values.
  - IX, IY are very rarely pointers, because the indirect access of memory via index registers is ridiculously expensive. Usually, IX and IY are useful as 16 bit values you need to add to (`ADD IX,dd` can be useful, esp. when HL is not available). `JP (IX)` is faster than `JP` and also keeps HL free. Last but not least, undocumented commands operating on halves of index registers are a bit slow, but not drasticly so, hence you tends to use them too, for constants that need to be defined/change around your inner loop and also as counters for outer loops. They are supported by every clone of Z80, so don't worry about their support, much software uses them.
  - `EX AF,AF'` is *insanely* useful, because it lets you use A for something intensive, but if need arises, do something else too.
  - `EXX` is very useful too, for more complex algorithms, but requires a bit of practice to get right. Common patterns involve either having 3-4 16-bit values for whatever computation and then using one of the HLs as the destination pointer (to save results). Another common pattern is to use first set of registers to read off your source data and then use the shadow set to write processed data elsewhere.

- Since stack was mentioned, you need to re-think the way you use it. Basically, stack operations are the fastest way to read/write data to memory, and in fact, it is done in 16 bit chunks, which is a massive win for an 8-bit CPU. So, stack is your best friend, but not in a way you normally think for modern machines:
  - The fastest way to fill memory with a constant value is to deploy stack:

    ```
    LD (SAVED_SP), SP : LD SP,end_addr
    LD HL,value*(256 + 1) : PUSH HL : PUSH HL : PUSH HL ...
    LD SP,(SAVED_SP)
    ```

  - The fastest way to copy a large chunk of data is to use stack.
  - Reading/writing data from memory is also often fastest via stack. A typical pattern with source and destination pointers is almost often improved by replacing one of pointers with the SP. I.e., instead of

    ```
    LD A,(DE) : AND (HL) : INC HL
    OR (HL) : INC HL
    LD (DE),A : INC DE
    ```

    it is faster to

    ```
    POP BC
    LD A,(DE) : AND C : OR B : LD (DE),A : INC DE
    ```

  - Saving/restoring intermediate values on stack tends to be an anti-pattern in inner loops, it is usually too slow.

I would be happy to expand on any of these points/examples, or maybe explore some more heuristics. I think that my real message would be that the limitation in the number of registers that you absolutely correctly identified is a pain, but can also be seen as an inspiring challenge, which leads you to consider fairly unusual (and very fun) patterns of programming.

Share Improve this answer Follow

edited Feb 19 at 22:40

user3840170

21k 4 86 140

answered Feb 18 at 22:55

introspec

4,012 1 18 27

5

What you are describing is a tricky problem. Especially for human brains. We may never be able to say the problem is truly *solved*, but we have software (compilers and other code generation) that mechanically apply some simple steps to get pretty close with it. And of course, the problem has had a lot of attention in academia as well, and not only for the Z80. So the way you can solve your problem is thinking about how a compiler would do it.

I would advise you to create the routine first, imagining that there is an infinite amount of space for your variables*. This means that allocating a variable is not difficult: just stick it in a vacant memory location somewhere.

You should also be sure to know about your variables' lifetimes. That means, from your variables first use (including of course initialization) to its last use (for example, the variable has gone out of scope**), the variable needs to be backed by memory. If a variable is handed to your function, and the caller expects to be able to use this same value afterward, then the variable is said to outlive the function.

Then, as a separate step, variables are loaded into registers before a section of code that uses that variable. The reason to do that is the limitations of the instruction set. For example, the Z80 cannot add a memory location to another memory location, so memory has to be read into the accumulator first. The nature of the instruction set, and precisely what you're doing with each variable will inform the decisions about which register to *temporarily* move your variable into. After working with a variable, the register needs to be written back to the memory, because after all, memory is where the variable is allocated.

After this step, you will have a routine that spends a lot of its time unnecessarily reading and writing to memory. It will be very inefficient. But it will be very clear to you where the loads and stored can be removed. Examples:

- a variable does not need to be stored to memory to be immediately read back (this is a peephole optimization).
- Or, a variable does not need to be read from memory if it's the first use in the variable's lifetime.
- If the variable's lifetime has ended then you don't need to write it back to memory (this is dead code elimination).

After this step, you will probably end up with unused memory locations, where you *used* to keep a variable, but you've proven that it doesn't need to live in memory. And of course, if two of your variables have lifetimes that don't overlap, then they can share a memory location, which can free up more memory.

Which part of your program should you optimize first? Keep in mind that the parts you optimize first will also be the parts you optimize best. So try and prioritize the parts you expect will run the most often (inside loops) and deprioritize parts you don't need to run very fast (e.g. error handling)

This whole problem *is* a really hard one for human brains. But by breaking the problem down you can get pretty close. And of course, you can test or debug the same program after each optimization step, which will show you if your bug comes from your logic, or from a mutation you applied on the way to improving your code.

*: Of course, not actually infinite, but theoretically limited to the amount of addressable memory (not registers) on the target machine.

**: For example, after a loop, you don't usually need the loop counter any more.

Share Improve this answer Follow edited Feb 17 at 11:51

answered Feb 17 at 9:44

Lorraine

37.9k 10 130 268

5

The limitations that you describe drove the development of some virtual machines during the 8-bit era:

- SWEET16 was a virtual machine implemented by Steve Wozniak for the 6502. It took up about 300 bytes of the Apple II ROM.
- CHIP-8 was developed for the COSMAC 1802. Unlike SWEET16, it includes instructions for video, audio, keyboard, and timers; they don't have to be implemented if they are not going to be used.

Although neither of these were created for the Z80, there is nothing inherent about them that would prevent them from being implemented on the Z80. Otherwise, you could use them as inspiration to develop your own virtual machine.

A few conclusions to be drawn from these virtual machines:

- They implemented 16 registers, any of which can be used as the source or destination of an operation. This illustrates just how inadequate an accumulator-based architecture is -- you spend more instructions getting data into and out of the accumulator than actually performing operations.
- SWEET16 registers were 16 bits, enough to hold an address. The native processor simply does not have enough address registers, and the operations that can be performed on addresses are too limited.
- Pointer manipulation is further enhanced by instructions with additional addressing modes such as autoincrement and autodecrement.
- Both virtual machines support subroutines in the virtual codespace, as well as returning to native assembly language.
- It's not as inefficient as you might think. A well-designed virtual instruction set can actually use fewer bytes of code in the long run, and the accumulator-based architecture is already a speed bottleneck.

Share Improve this answer Follow

answered Feb 17 at 13:41

DrSheldon

15.7k 5 48 111

Agree on the virtual computer as solution. The Apollo Computer, used in moon landings in the 60-s, had one part with a virtual computer "featured double precision trigonometric, scalar and vector arithmetic (16 and 24-bit), even an MXV (matrix × vector) instruction" (text from wikipedia). – ghellquist Feb 17 at 21:14

Having an accumulator which can be accessed more readily than other registers isn't a bad thing if it's possible to write operation results to either the accumulator or back to the other source operand and--for a slight speed penalty--to use something else in place of the accumulator for an instruction. The PIC architecture actually works pretty well, and would work even better if instead of "movff" there were an instruction that would cause the value of a memory operand to be used in place of the accumulator for the following instruction. – supercat Feb 21 at 5:51