SD-8516 ISA Review

Why was VC-3 Created?

- VC-0 was NetWhack, a roguelike game I had written in Java, and had rewritten in various other languages, like Python and Javascript. I wasn't happy with the lack of blocking input in JavaScript, so I wrote VC-1.
- VC-1 was a JavaScript terminal simulation to demonstrate how to do blocking input. It took me a very long time to come up with this method. It is based on how I did it in VC-0 under LWJGL and in a PyGame framework I wrote in Python.
- VC-2 was a proof of concept. VC-2 is the SD-8510. It is a simple 8 bit CPU with 16 bit address pointers and some bank pointers. It was a bit of a mess, but it worked. It was also written in JavaScript.
- VC-3 is the direct evolution of the SD-8510, called the SD-8516. It is written in AssemblyScript and is 20x faster than the SD-8510, reaching peak sustained speeds of 100 MIPS on a GeekBench 6 Baseline system.

What Changed from the SD-8510?

The SD-8510 was an 8 bit CPU so LDA was an 8 bit load operation. We also had a highly orthogonal MOV operation that was the “real” operation, and LD/ST operations were converted to MOV before assembly. There were some other oddities, primarily in the way banks and pointers worked. We had several bank pointers. I didn't like the system, it was a constant mess. So they were removed and I adopted a register pairing system instead.

1. The Register-Indirect Conditional Jump block (JZRI, JNZRI, JNRI, JNNRI, JCRI, JNCRI, JORI, JNORI — opcodes 119–126)Eight opcodes consuming encoding space, each requiring a memory load to fetch the target address from RAM. The use case — “jump if zero to the address stored at the memory location pointed to by register” — is a double indirection that virtually never arises in practice. If you ever needed this, LD ELM, [reg] followed by JZR ELM accomplishes the same thing in two instructions, and you can actually debug it because the intermediate address is visible. The unconditional JMPRI (118) is marginally defensible for jump tables, but even that is better served by CALLR with a 24-bit register. That's 9 opcodes (118–126) you could reclaim; at minimum, the 8 conditional variants should go.

Register Pairing

This is something that some CPUs do– they pair registers to get a larger addressing mode. For example, AL and AH become the AX register, and the AX register is the low word part of the EAX register. Well, imagine that “BX is the high word part of the EAX register” and that EAX is called AB. And, that the word register is called A, not AX, similarly with B. Then, EAX is AB, instead of EAX being [AX:high word]. That is register pairing in a nutshell. The convention is different than on an 8086 or a 6502. Here, A is a 16 bit register, while AL and AH are still 8 bit.

Could I have done it differently? Well, imagine 24 different byte registers. I and S, of course, being IP

and SP and not available for use. Then, we could string them together as in [A:B:C] for a 24 bit pointer (C being the hi-byte). Or we could just say ABC, or AB for 16 bit. That's possible. This would keep a look and feel similar to 8 bit 6502 code. In the end there are a lot of different things I could have done, but the way I did it seems OK in the end. I do not like LDAL and such, the L seems strange compared to LDA being 16 bit. Perhaps I would have preferred to have A and A' as registers going into AX. A and A' (a-hi). Or A and AH (in AX). Then EAX would be 32 bit, and we could have an AX'. But how do you address AX'? XA and XA'? Well, maybe, but it gets a bit confusing.

AB is 32 bit, ABCD is 64 bit. That's simple. it also encourages you to use 16 bit as the word length. For pointers, its BLX, ELM, FLD, GLK and the like. Actually you get used to it. In the end, it's all about convention anyways. I'm happy I broke with convention and created my own. It tastes better.

Non-orthogonal MOV

I set MOV aside for register to register moves ONLY.

LD/ST is for immediate or memory pointer to register ONLY. For example, you cannot do LDA B. Neither can you do MOV A, $20. You must do LDA $20 and MOV A, B. That's how it works. I like this system.

Other than that, there are a precious few operations I added. One is TXS and TSX which puts the stack pointer in a named 24 bit pointer. This is so you can do stack frames, which is a fancy way of saying you put local variables and calling parameters on the stack when you call a function. It's a convention beloved by compilers, but if you are programming in assembly, we *do* have 16 general purpose word registers to play with!

TESTF is another one. Not every flag op has a JZ or a CLZ/SEZ. With TESTF you do a non-destructive AND which sets the Z flag. Think of it like testing a set of flags and then use JZ to jump if those flags are set.

The 3 most useless instructions on the SD-8516 and why I can't remove them.

Register-Indirect Conditional Jumps

- JZRI, JNZRI, JNRI, JNNRI, JCRI, JNCRI, JORI, JNORI

Why I hate them

Eight opcodes consuming encoding space, each requiring a memory load to fetch the target address from RAM. The use case, “jump if zero to the address stored at the memory location pointed to by register”, is a double indirection that virtually never arises in practice.

If you ever needed this, LD ELM, [reg] followed by JZR ELM accomplishes the same thing in two instructions, and you can actually debug it because the intermediate address is visible. The unconditional JMPRI is marginally defensible for jump tables, but even that is better served by CALLR

with a 24-bit register. That's 9 opcodes of wasted space; 8 if you like JMPRI.

ROLC / RORC

I've never used them, but I hear it's useful for crypto. Apparently? Multi-precision shift chains where you rotate through carry across multiple registers.

SHLC/SHRC already cover the “shift with carry output” pattern that's actually useful (bit extraction, serial protocol work). If multi-precision arithmetic ever becomes necessary, ADDC/SUBC handle it more naturally.

ROL/ROR (the non-carry versions) are at least conceptually simple and might see use in checksums or hash mixing, so they're borderline. But ROLC/RORC are strictly for a programming pattern that I don't need.

Actually the kicker here is you can in fact simulate it if you really need to. This CPU is not intended to be used for cryptography where massive optimizations are needed. If you really need a RCL or RCR style instruction, just keep a variable somewhere and test the MSB (most significant bit), then ROL (for example) and replace the LSB with what was in the variable, *then set the variable with what the MSB was*. This simulates a RCL/ROLC, and is fine.

SETF / CLRF (opcodes 222–223)

TESTF has proven its worth, but SETF and CLRF are redundant with the dedicated flag instructions (SEZ/CLZ, SEC/CLC, etc.) The fact is, the other flags, we don't really need. But I'm not going to remove it yet. If I remove it I need to replace it with flag set and get instructions. It's a meh case at best – and at worst.

CMPB

Byte versions like this should have been removed already, but I think I saw a reference somewhere in the code and I'm going to have to hunt it down. All operations (like CMP) work contextually by their register size.

On the Review block

SHR and SHL need a by-n form. Ok, I admit it, but I just don't care. Maybe later.

To Add

I should probably add indexed addressing.

Another mode for LDA? or call it LEA (load effective address)?

```
  LDA [BLX+I]           ; base pointer BLX, index pointer I
```

or

```
  LEA A, [BLX+I]
```

The LD looks more compact and familiar? Looks like the LDA version fits the look and feel of the ISA so far, but I like LEA too!