One Instruction Per Cycle

When discussing emulators, people often bring up the concept of “cycle accurate” emulation. This means that some instructions take longer to execute than others. On a C64 for example, it does not execute one instruction operation per cycle, but rather it takes an average of 3.1 to 3.3 cycles to complete the average instruction. Therefore, on a 1.024 MHz C64, it executes approximately 320,000 instructions per cycle. This can be higher or lower, with some accounts saying up to 350,000. It depends how you program.

However, the SD-8516 is not trying to emulate anything, and is thus unconstrained by the need to impose artificial limitations on things like cycles per instruction. It has no need to perform at a certain speed. Therefore, how long does it take to execute one instruction operation?

One cycle.

Every cycle, or “cpu_step()” or “execute_instruction()” if you prefer, takes one “loop”. It fetches the opcode, and enters a rather large switch statement which finds the opcode and executes it. This takes about the same time no matter which instruction it is executing. There is however, a very small variation in how long it takes to execute an instruction. In fact, some operations can be up to three times slower than others! What's the diff?

The issue is touching memory. Every time you need to touch memory, it slows the system down. One cycle is one cycle. if it has to run through an IF, it takes about 20-30% extra time. And for every memory access, it slows down about as much.

If I had to guess, I would say that if each operation had a base speed of 10, then a memory access would add 4 and an if would add 4. This a very rough estimate. So, an instruction that has to fetch an extra word, such as `LDAB $00112233` has to fetch the additional memory $2233. It also has to put it into AB, which is very fast. But, overall, it's about 70% the speed of two word operations. So it's faster than doing two word operations, because in the 2nd word operation, you have to fetch a second opcode and a second register byte and this pushes the time over 2.3 or 2.4 of a normal instruction.

```
LDA $2233     ; takes 10 units of time
LDB $0011     ; takes 10 units of time
LDAB $00112233 ; takes 16 units of time
```

Something like this, it's very rough.

Where this gets interesting is in certain algorithms that do something like SHL to “quickly” multiply a number. Our multiply is just as fast as a SHL, maybe faster. So doing this:

```
; Multiply A by 16:
SHL A
SHL A
SHL A
SHL A
```

This takes 40 units of time, 10 for each op. But this takes 10 units of time (maybe 11):

```
  MUL A, #16
```

So one must always remember that since the execution loop as a fixed amount of time, fewer instructions is almost always faster than multiple.

It also pays huge to remember which operations set flags. On a lower-memory computer such as an Apple][, BBC Micro, C64, TRS-80 etc. you would want to know this to save space. Here, you need to know it to save time as well!

Example:

```
  loop:
     DEC X
     CMP X, #0
     JZ @finish
     JMP @loop
```

```
  finish:
     RET
```

This example program takes 4 instuctions and 8 bytes of memory access. Can we do better?

```
  loop:
     DEC X
     JNZ @loop
```

```
  finish:
     RET
```

This is much faster. We know that DEC will set the zero flag if it hits a zero. Second, we invert the check and use a fallthough to get to the continue point. So this inner loop is more than twice as fast as the last one. This works with LDA as well:

```
  str_loop:
      LDAL [ELM]
      LDBL [FLD]
      JZ @str_end      ; If LDBL loaded a zero, then zero flag is set-- the
string has ended.
      CMP AL, BL       ; Are the strings equal?
      JNZ @str_loop
```

```
  str_end:
      CMP AL, BL       ; Just perform the check again to make sure that if
BL was 0 they both ended.
      ; Now we can return the strcmp results.
```

This idea shows that if LD load a zero, we can immediately check. We do not need a CMP.

The idea is that fewer instructions are faster.

- On normal CPUs MUL and DIV are very slow.

    - On this, it's faster than several SHLs.
- DIV is also faster than several SHRs.
- Remember which operations set flags so you don't waste time on a CMP.

Oh help, i'm running out of registers!

There is a thing where it's possible to get confused over register pairing and run out of registers for your convention. In this case there's a couple things you can do. One, is you can use memory as a free register. Consider:

```
  LDA $500 ; loads $500 into A.
```

The above compiles to: 00 00 05 00. Thats four bytes that need to be read in. Lets say then that the cost of this instruction is to load four bytes from WASM memory. That is the essential cost; the CPU fetch-execute cycle itself has a fixed cost, so we can ignore it or add some constant.

Now consider:

```
  MOV A, B
```

This costs three bytes to fetch and execute; one byte for the opcode and one for each register. What about from memory?

```
  LDA [$000000] ; This is a 5 byte instruction.
```

At five bytes, it's 30-40% slower, on average, than LDA $500. But for scratch values, that are infrequently used, and not in a hot path, this frees up registers. In other words, feel free to use memory!

Now, if you consider the number of operations and bytes a simple push/pop would use (to juggle a register) maybe it's better to keep certain kinds of pointer or scratch value in memory versus in a register? It would need to be done with consideration, so that its not in the hot path, but even so it should be of nearly acceptable speed. For example, we can make up for it by avoiding that wastrel CMP A, #0 and JZ on a LDAL! With a little effort, the tradeoff is not so bad. Lets use memory locations for certain kinds of scratch or temporary values, if no registers are available!