Flag Operations are Free

- By Appledog
- January 11th, 2026

Abstract

Discoveries made during profiling to determine what was slowing down my CPU emulator revealed some surprising insights into emulation implementation.

Issue

Sometime during the development of the SD-8516 virtual retro CPU, the processing speed went from 60 mips to under 15 mips. I first thought it was my mac, since I benchmarked it on my mac and only got 20 mips. I was pretty upset and went down a rabbithole of trying to install various browsers, enabling SIMD, and compiling in C. The truth was quite different. During the implementation of various opcodes, decisions I made that struck at the heart of the ISA itself revealed that certain instructions were driving down the general performance of the CPU.

I began by creating a series of profiling programs. I will refer to them by the name of the opcode being profiled, with a number; ex. LDAL-1, LDAL-2, and so forth.

To make a very long story short, here are the programs, the results, and the conclusions:

Here's LDAL-1:

```
; Test program: 1 million LDAL [$1000] operations
; Uses CD (32-bit "count down" counter register)

.address $010100

    SEF ;; fast flags mode : do not perform flag ops for LD and DEC
operations.
    LDCD #1000000        ; Load 1,000,000 into CD (0x989680 is 10 mil)

loop:
    LDAL [$1000]          ; Load AL from address $1000
    DEC CD                ; Decrement CD
    CMP CD, 0
    JNZ loop              ; Jump to loop if CD != 0

    HALT                  ; Halt when done
```

This is a pretty simple take on a simple concept; Execute 1 million LDAL operations and see what happens. The result was a MIPS score of 1.85. I became depressed. How had my beautiful CPU

become so slow? Just a few weeks ago it was pulling over 60 MIPS. Now, it was showing scores that didn't make sense.

This was, in fact, the purpose of adding the SEF instruction you see above. In desperation to find the source of the slowdown, I had commented out all of the debug IF checks and I had set up a fence around most of the flag operations. Changing SEF to CLF above gives us LDAL-2, which turns on flag checks for LD and DEC. It does not change the operation of this program, since we explicitly check for zero with CMP.

The results of LDAL-2 shocked me. Even with fast flag mode turned on, the program remained locked at 1.85 MIPS for multiple runs. In other words, even though there were over 4 million additional checks to set FLAGS data, the processing time did not increase or decrease and remained locked in at 1.85 MIPS.

Next I moved to LDAL-3 where I removed the CMP since it was no longer needed:

```
; Test program: 1 million LDAL [$1000] operations
; Uses CD (32-bit "count down" counter register)

.address $010100

    CLF                 ; fast flags off : perform flag ops for LD and DEC
operations.
    LDCD #1000000       ; Load 1,000,000 into CD (0x989680 is 10 mil)

loop:
    LDAL [$1000]        ; Load AL from address $1000
    DEC CD              ; Decrement CD
;   CMP CD, 0           ; removed since DEC CD will set zero flag if it
DECs from 1 to 0.
    JNZ loop            ; Jump to loop if CD != 0

    HALT                ; Halt when done
```

Now this was a real eye opener. Removing the explicit check and keeping the flag ops ON, resulted in a MIPS score of 2.1! Well now, this was surprising but not entirely unexpected. Well, no, it was unexpected. Removing flag operations for LD and DEC is significant as they are both being executed 1 million times each. Here's the code that we're talking about:

- ZERO_FLAG = (value & 0xFF) === 0;
- NEGATIVE_FLAG = (value & 0x80) !== 0;
- ZERO_FLAG = result === 0;
- NEGATIVE_FLAG = (result & 0x8000) !== 0;
- OVERFLOW_FLAG = value === 0x8000;

That is a significant amount of code to remove, but ONE compare op was killing it. Having this make no impact whatsoever was surprising, so I removed the IF statements blocking these flags on DEC. This produces LDAL-2b, which surprised me by getting again the exact same 2.1 MIPS. So, over 2 million if statements AND two million times the five lines of code above wasn't moving the needle? Wow.

I replaced the flag fences and I created LDAL-3; this time, I had 100,000 runs of 10 LDAL operations. My heart lept for joy when I saw the score; 7.55 MIPS! This meant that LDAL was executing much faster than the other instructions. I immediately created LDAL-4 which had 1,000 lines of LDAL and loaded CD with 1 million. The goal was simple: execute 1 billion LDAL instructions and time the result. The results were spectacular. 78 MIPS. I did try with CMP,0 and SEF mode, and it was slower (73 MIPS). The immediate conclusion is that SEF mode was useless. CMP was dragging everything down. But I didn't know why.

It was only when I began profiling the LDBLX and LDAB addressing modes did that 78 mips fall all the way back down to 1.5. I was in shock, and then I realized what was happening. Inside the LDBLX and LDAB code were array allocations being performed in the hot path! I removed them into a switch statement and performance shot back up to about 40 MIPS.

```
; Test program: 1 million LDAL [$1000] operations
; Uses CD (32-bit "count down" counter register)

.address $010100

    LDCD #1000        ; Load 1,000 into CD (0x989680 is 10 mil)

loop:
    LDAB [$1000]          ; Many of these; x1000
    DEC CD                ; Decrement CD
    CMP CD, 0
    JNZ loop              ; Jump to loop if CD != 0

    HALT                  ; Halt when done
```

Finally, I wanted to find out why they were operating so slowly compared to LDAL. This led me to discover that load<u8> was the same speed or sometimes slower than an aligned load<u32>. I began batching and pre-loading values and masking out what I needed and this was enough to boost speed from 40 MIPS up to about 55 MIPS.

In closing, here's the 87.5 MIPS version of LDA16 [$addr], the workhorse of this 16 bit system:

```
        case OP.LD_MEM: {
            // Load reg (1 byte) + addr (3 bytes) = 4 bytes total
            let instruction = load<u32>(RAM + IP);
            let reg:u8 = instruction as u8;                    // Extract
low byte
            let addr = (instruction >> 8) & 0x00FFFFFF;      // Extract
upper 3 bytes
            // Pre-load 32 bits from target address
            let value = load<u32>(RAM + addr);
            let reg_index = reg & 0x0F;  // Extract physical register 0-15
            IP += 4;

            if (reg < 16) {
                set_register_16bit(reg, value as u16);
```

```
                ZERO_FLAG = value === 0;
                NEGATIVE_FLAG = (value & 0x8000) !== 0;
                //if (DEBUG) log(`$${hex24(IP_now)}    LD${reg_names(reg)}
[$${hex24(addr)}] ; = ${hex16(value)}`);
            } else if (reg < 48) {
                set_register_8bit(reg, value as u8);
                ZERO_FLAG = value === 0;
                NEGATIVE_FLAG = (value & 0x80) !== 0;
                //if (DEBUG) log(`$${hex24(IP_now)}    LD${reg_names(reg)}
[$${hex24(addr)}] ; = ${hex8(value)}`);
            } else if (reg < 80) {
                set_register_24bit(reg, value & 0x00FFFFFF);
                ZERO_FLAG = value === 0;
                NEGATIVE_FLAG = (value & 0x800000) !== 0;
                //if (DEBUG) log(`$${hex24(IP_now)}    LD${reg_names(reg)}
[$${hex24(addr)}] ; = ${hex24(value)}`);
            } else {
                set_register_32bit(reg, value);
                ZERO_FLAG = value === 0;
                NEGATIVE_FLAG = (value & 0x80000000) !== 0;
                //if (DEBUG) log(`$${hex24(IP_now)}    LD${reg_names(reg)}
[$${hex24(addr)}] ; = ${hex32(value)}`);
            }
            break;
        }
```

The Prefetch System

Combining the loads and bit-shifting things out worked so well I had the brilliant idea to create a prefetch system. Why not batch loads across instructions? This would allow us to capitalize on aligned loads but still have long instructions (5 bytes and up) without running into alignment problems.

Here's the code, preserved for posterity, may it serve a lesson to all that too much of a good thing actually sucks:

```
//
================================================================================
// INSTRUCTION PREFETCH BUFFER SYSTEM
//
================================================================================
//
// Purpose: Reduce memory operations by batching instruction fetches.
// Instead of loading 1 byte at a time from RAM, we load 4 bytes at once
// and extract individual bytes from the buffer as needed.
//
// Trade-off: Adds overhead from buffer management, but amortizes memory
// loads across multiple instruction bytes.
//
```

```
// Performance note: This system was intended to improve performance by
// reducing memory operations, but in practice added overhead that reduced
// MIPS from ~90 to ~60. WebAssembly's memory access and CPU prefetching
// are already well-optimized for sequential loads.
//
================================================================================

// Prefetch buffer holds up to 4 bytes of instruction stream
let prefetch_buffer: u32 = 0;        // The 4-byte buffer holding fetched
data
let prefetch_valid: u8 = 0;          // How many valid bytes are in buffer
(0-4)
let prefetch_offset: u8 = 0;         // Current read position within buffer
(0-3)

/**
 * Fetch a single byte from the instruction prefetch buffer.
 *
 * Automatically refills the buffer when exhausted by loading 4 new bytes
 * from RAM at the current instruction pointer (_IP).
 *
 * @returns The next instruction byte
 */
function prefetch_byte(): u8 {
    // Check if buffer needs refilling
    if (prefetch_offset >= prefetch_valid) {
        // Load 4 new bytes from instruction stream
        prefetch_buffer = load<u32>(RAM + _IP);
        prefetch_valid = 4;
        prefetch_offset = 0;
        _IP += 4;  // Advance physical IP by 4 bytes
    }
    // Extract byte at current offset (0, 8, 16, or 24 bits from right)
    let byte = (prefetch_buffer >> (prefetch_offset * 8)) as u8;
    prefetch_offset++;
    return byte;
}
```

You can imagine the rest of the code – this is enough to understand the problem:

```
  if (prefetch_offset >= prefetch_valid) is very bad.
```

Let me put it this way. If I run an IF on every opcode, it slows the program by 50%, turning a 90 mips LDA benchmark into a 45 MIPS benchmark. Now, I tried a lot of different methods to try and get a better score than just using load<u8> when needed. The closest I got was a branchless <u64> prefetch version of the above, which got around 84 MIPS. Even just trying to load everything at the top of the loop (a u32 load) and bit-roating out the operands was no more than a 1% improvement (i.e. not worth the trouble, frankly).

In fact, the way I got to a 95 MIPS benchmark for unrolled LDA operations was to simply use

fetch_byte(). If I did anything else, *including inliling the load<> operations*, the program would run slower.

At first glance you may wonder what on earth is happening. How did I beat the 87.5 MIPS implementation above? Simple, by not trying to cheat the system. As it turns out, load<> is already as optimized as it is going to get in Web Assembly. The epiphany is, we're running a simulation. And the host computer is actually doing a good job of helping us access memory. Any abstraction we put on top ends up getting in the way.

It's strange but true. If you factor out the load operations, trying to load everything at once adds complexity:

```
  let addr = (instruction >> 8) & 0x00FFFFFF;      // Extract upper 3 bytes
```

You're adding a bit shift, a bitwise AND, plus you're creating an intermediary variable access. In the end this is almost 10% slower than just calling fetch_byte().

Conclusion

The grass is green, the sky is blue.

```
export function cpu_step(): void {
    let IP_now:u32 = _IP;

    // Pre-fetch 4 bytes but only commit to using opcode initially
    let opcode = fetch_byte();

    switch(opcode) {
////////////////////////////////////////////////////////////////////////
        // LDR/STR load and store architexture
//
////////////////////////////////////////////////////////////////////////
        case OP.LD_IMM: {
            let reg = fetch_byte();

            // Determine width based on register number
            if (reg < 16) {
                // 16-bit register
                let value:u16 = fetch_word();
                set_register(reg, value);

...
```

Nothing beats this. This is it. You can't even inline it, it messes with the compiler.

If you want to get faster than this, you need to rewrite the entire switch in C or maybe Rust.

As a result of all this, I now know that flag operations are free inside an opcode and I don't need to

have a “fast flags” bit. Simply checking that bit every instruction was slowing down the system by far more than it saved.

Moral: “Premature optimization is the root of all evil.”