Emulation Benchmarks

I have a working theory that the feel of a system is defined by it's limits. Meaning, “this is what we can give you, and no more”. Within this space lives the programmer. When restraints are tight the programmer must work hard and come up with something special just to make something in the first place. When restraints are removed, programmers get lazy. Well let's be honest – not lazy but in some ways more productive and in some ways less.

When you have a lot of extra ram and processing speed, things tend to fall into a 3d open world environment. However, CastleVania SOTN on Playstation 1 showed that even in a 30 MIPS, 32 bit environment (the SNES was 8/16 bit, 1.4 MIPS) you could make a credible 2d RPG. The thing is though, Castlevania could do it because Castlevania came from that legacy. To make SOTN without the existing IP behind it would have been considered a mistake at the time.

Therefore, let us attempt to allow emulation of modes and time periods – NOT of hardware itself. The 8516 has a decent period-accurate “feel” ISA that should allow easy porting and in theory would be an excellent target for a C compiler. Those aren't going to be issues. The issues are going to be forcing programmers to comply. Therefore, when being considered for a “Gold Star” award by the SD-8516 foundation (I just made that up) programs must conform to the following specs:

Other Fantasy Computers

- SD-8516 clocks in at 125 MIPS on an i7-12700K (Geekbench 6 baseline 2500)

On the same computer,

- SD-8510 runs at 4.5 MIPS
- XR/Station running ASIX, dhrystone 2.1 shows 9.75 VAX MIPS

The Chart

Legend:

- Green: The SD-8516 is capable of emulating this level of performance on an i7-12700 (Geekbench 6 baseline).
- Light Gray: Although the SD-8516 cannot currently emulate this level of performance, it is expected to once graphics acceleration is emulated.
- Dark Gray: The SD-8516 cannot emulate this system. This is almost always because it is slower than 50% of the required speed. For example, it is unlikely to achieve Dreamcast-level performance even on the new M5 Pro Max chips (4000+ single core). This level of performance would require a C rewrite and some form of multithreading.
- Red: The system is unlikely to emulate this level of performance because it's advanced hardware. It is possible that a C rewrite would get close, but would require enthusiast hardware. These benchmarks are for GB6 baseline systems.

Year	System	CPU	Width	Approx MIPS	RAM	Graphics	Audio	Notes
1977	Atari 2600	MOS 6507	8-bit	0.30	128 B	160×192, ~128 colors (TIA tricks)		No framebuffer; cycle-exact emulation needed
1979	Intellivision	CP1610	16-bit	0.35	1 KB	159×96×16		16-bit CPU but very slow clock
1980	VIC-20	MOS 6502>	8-bit	0.45	5 KB	176×184×16		Simple video chip; CPU-bound
1981	BBC Micro	MOS 6502	8-bit	1.00	32 KB	640×256×2		2 MHz CPU; very tight timing
1982	ColecoVision	Z80 @ 3.58 MHz	8-bit	0.58	1 KB + 16 KB VRAM	256×192×16		Same VDP as MSX
1982	C64	MOS 6510	8-bit	0.36	64 KB	320×200×16	SID	VIC-II steals cycles–slower than VIC-20 CPU-only
1983	NES	Ricoh 2A03	8-bit	0.50	2 KB	256×240×25	2×PWM, 1×triangle, 1×noise, 1× DMC (4-bit PCM samples)	Heavy PPU timing–emulation harder than C64
1983	Apple IIe	MOS 6502	8-bit	0.50	64 KB	280×192×6		No sprites; CPU-driven graphics
1985	C128	MOS 8502	8-bit	0.75	128 KB	320×200×16	SID	Faster CPU but still VIC-II limited
1987	Amiga 500	68000 @ 7 MHz	16/32	4.5	512 KB–1 MB	320×256×32/64/4096	Paula (DMA-driven PCM playback)	Custom chips dominate emulation cost
1991	SNES	Ricoh 5A22	16-bit	1.5	128 KB + VRAM	256×224×256	Advanced Sample Playback	Slow CPU; complex PPU & DMA
1991	386SX	80386SX @ 25 MHz	32-bit	10	4 MB	320×200×256 VGA	SB Pro	Software rendered; cache key for speed
1992	Amiga 1200	68EC020 @ 14 MHz	32-bit	14	2 MB	320×256×256	Paula (DMA-driven PCM playback)	Much easier CPU than SNES to emulate
1992	486 Gamer	486DX2-66	32-bit	54	8 MB	640×480×256 (S3 ViRGE)	SB16	~20 FPS software Quake; 3D accel emerging
1994	PS1	MIPS R3000A	32-bit	30	2 MB + 1 MB VRAM	320×240	SPU	Geometry-heavy; no GPU T&L
1994	Pentium 90	586, 90 MHz	32-bit	90	32 MB	640×480×16M (PCI VGA)	SB AWE32	Smooth Quake @ 50+ FPS; Win95 ready

Year	System	CPU	Width	Approx MIPS	RAM	Graphics	Audio	Notes
1996	N64	MIPS VR4300	64-bit	125	4–8 MB	320×240–640×480		Hard due to RSP/RDP synchronization
1998	Dreamcast	SH-4 @ 200 MHz	32-bit	360	16 MB	640×480	AICA	Very emulator-friendly architecture
2000	PS2	MIPS R5900	128-bit SIMD	6000 + 40	32 MB	640×448	SPU2	Emotion Engine 6k MIPS; +40 MIPS for PS1 compat.; VUs dominate
2001	GameCube	Gekko @ 485 MHz	32-bit	1125	24 MB	640×480	Flipper DSP	Dolphin emulator gold standard; clean PowerPC arch
2017	Switch	ARM Cortex-A57	64-bit	12,000	4 GB	720p–1080p		GPU & OS dominate emulation cost

Observations

Looking at the above chart, you will see “what happened”. In the early 90s, the 486dx-66 was the last great consumer CPU before the 3d revolution really took off. The death of the Amiga, the rise of the accelerated playstation and the absolute dominance of the Pentium changed how home computing worked. After that moment, graphics acceleration was a must and MIPS did not matter anymore. Thus, the PS-1, even though it is only 30 mips, had enough graphics acceleration to do things a similarly priced PC of the day could not.

> The original Pentium (P5, 1993–1997) provided several key architectural accelerations that Quake's software renderer exploited via hand-tuned x86 assembly (primarily by Michael Abrash), delivering ~2–3x the performance of a 486DX4-100 in rasterization-heavy scenes.

The Days before GPU Acceleration

The Pentium was really the key that launched the GPU acceleration wars.

- Superscalar dual integer pipelines (U/V): Allowed two integer instructions per cycle, allowing quake to draw pixels in bursts
- Pipelined FPU: pipelined floating point ops let Quake fire FDIV for perspective-correct texturing in parallel with integer rasterization (r_alias.s), making slow DIV “free” (~1c effective).
- 64-bit FPU data path: Enabled fast 64-bit FP loads/stores and near-free FXCH (stack rotates, 0-1c), optimizing matrix/dot products for geometry and lighting
- Branch prediction: predicted branches (e.g., bottom-of-loop) increased throughput.

These Pentium-specific traits were exploited via Abrash's hand-tuned ASM (id386.asm) delivered a 3x speedup over 486DX4-100 and AMD/Cyrix 5×86-133 style CPUs, crushing the clones' weaker floatinf point pipelining and marginalizing them in gaming. Pentium began to dominate the 1996 PC market

as Quake's “minimum viable” software 3D benchmark, shifting devs from CPU raster hacks to hardware offload. Next, GLQuake/Voodoo (1996) hit 60+ FPS by rasterizing on GPUs, birthing the 3D acceleration era.

SD-8516 Emulation Strategy

The SD-8516 operates at a GB6 baseline of 70 mips. This implies it has a similar operational power to a 486DX-66. While development is still ongoing, the strategy will be to allow the programmer to control various system modes that allow the “emulation” of various concept systems. Such as the typical 0.5 MIPS, 4k ram, low-res system you might find in 1980.

Then we can allow modes that simulate the C64/128 era, which represents the 1 to 2 MHZ, 64k-128k RAM “basic” era. This will also include consoles like the NES and SNES.

Finally, we can allow an emulation mode similar to an Amiga 500, 386, or Amiga 1200. This means a semi-advanced workstation; old style, probably not suitable for modern work which requires super-high resolution images and advanced sound processing; but more than good enough to represent most games that are not open-world 3d first person shooters.

Next, with some minor graphics acceleration, such as writing commands to a memory mapped javascript game library, we can provide enough acceleration to push operational performance into PS1 and N64 territory. This more than covers the classic era of gaming; We can probably get quake to run decently. And frankly, Quake was really the last great development in gaming, half-life was essentially a kind of quake, and we love hl2 and hl3 for the story. There really does not need to be any further development.

However, rewriting in C would blow the lid off performance, allowing dreamcast and gamecube level performance. There's something to that; rewriting the execution loop alone, and processing the image directly in web assembly, javascript can just project the image. C is viable but for all intents and purposes we are far far away from that need. For the forseeable future we need to:

1. stabilize the ISA
2. stablilize a kernal
3. Write a BASIC
4. INT helper function library
5. profile instructions and try to get MIPS up to 95-100

The ISA and kernal are essentially stable. I don't see a need to make major changes now. Possisbly some block operations or floating point operations can be added. But the very likely way this will be done henceforth is through the INT library. I like to keep a simple ISA. As for the kernal, we have a basic input system that looks and feels like a typical language machine – similar to a BASIC terminal or Python command line.

Functions can get pulled into the kernal maybe, but right now the main direction forward is to get BASIC up and operational by writing as few INT helper functions as needed. Once we have the BASIC system, video/audio mode switching will be slowly implemented by working on creating environments in various video modes, each of which will attempt to recreate the feel of an era. But, the actual operation will be kept separate, so you can mix and match. it will be beautiful!

Profiling Experiments

Writing assembly language programs on the SD-8516 is similar, but different to writing them on the 8510. For one, programs are entered in much the same way:

Profiling Example

The following program illustrates a baseline:

```
.address $010100

    LDA $1010
    LDB $0101

    LDCD $0FFFFFFF          ; Load 1,000,000 into CD (0x989680 is 10 mil)

loop:
    DEC CD                ; Decrement CD
    JNZ @loop              ; Jump to loop if CD != 0

    HALT                  ; Halt when done
```

This program executes at a certain speed we can call X. It doesn't matter what the speed is for now, suffice to say it is X in terms of MIPS or some other benchmark. When discussing the profiling of a command, we have to determine if it pulls the execution of this loop up or down from X. In this manner we can judge the relative speed of the instruction; if A is the speed of DEC, and B is the speed of JNZ, then the portion remaining goes to the instruction being profiled. However, when adding just one instruction, it is difficult to judge the true speed of the instruction in question. The solution is to increase the number of instructions per loop, which is known in a way as unrolling the loop.

One idea is to increase the number DEC instructions relative to JNZ and see what happens. In the regular run I got a score of 77 MIPS on my 12600k. Increasing the DEC:JNZ ratio to 10:1 brought us down to 56 mips. At 100:1 we got 54 MIPS.

On the other side, a program that tests JNZ to DEC 10:1 brings MIPS up to 91. In either case, a nearly 20 MIPS difference. Therefore clearly, JNZ is a much faster operation than DEC, although you would expect DEC to be a lot faster than JNZ! The reason why is that DEC CD is very slow, as it is a dual register DEC. Moving to single register DEC increases the speed by 50-100%:

```
.address $010100

    LDC #10000
    LDD #25000

loop:
    DEC C
```

```
    DEC C
    DEC C
    DEC C
    DEC C
    DEC C
    DEC C
    DEC C
    DEC C
    DEC C
    JNZ loop

    ; C reached zero, decrement D
    LDC #10000
    DEC D
    JNZ loop

    ; done
    HALT
```

This version runs at 90 MIPS. Considering all of the results so far, we'll use the double C counter version with 20 executions of the profiling instruction unrolled inside the loop. We'll also take the C loop down to 10,000 from 30,000 seeing as how we will be unrolling instructions in the loop, and they are almost surely bound to be slower.

The following chart indicates the best results out of several runs:

LDA

Instruction	Execution time	Notes
Empty Loop	97 MIPS	
LDA [$1000]x10	90 MIPS	
LDA [$1000]x100	95 MIPS	
LDAL [$1000]x20	85 MIPS	Not native word size
LDAB [$1000]x20	76 MIPS	unexpected! will check code
LDBLX [$1000]x20	25 MIPS	array method method
LDBLX [$1000]x20	45 MIPS	switch method
LDBLX [$1000]x20	64 MIPS	unified memory reads
LDBLX [$1000]x20	73 MIPS	inlined acceess

Notes on LDA/LDAL

This is likely a branch prediction and instruction cache artifact in the Web Assembly/JavaScript JIT compiler. With the empty loop, the CPU's branch predictor may be working against speculative execution overhead. Adding a single LDA gives the pipeline something productive to do between branches, potentially hiding some of the branch misprediction penalty or better aligning the instruction stream. At 10-20 instructions, you're hitting different bottlenecks:

Increased loop body size may cause instruction cache pressure More register pressure in the

generated machine code Loop overhead becomes proportionally smaller but absolute instruction decode cost increases

The LDAL slowdown confirms this - non-native 32-bit operations require more complex codegen, putting additional pressure on the optimizer. This is classic JIT behavior: a tiny amount of work can sometimes improve performance by giving the CPU's execution units better scheduling opportunities, but too much work overwhelms those benefits. You might also be seeing alignment effects - the single instruction could be placing the loop branch at an optimal address boundary.

Finally, using LDBLX as a proxy for the process we went through earlier, we achieved a 3x speedup by using a switch versus a map, unifying <u8> memory reads into <u32>, and inlining the the load() calls into the opcode handler.

I wouldn't want to do this for every instruction because it produces ugly, hard to maintain code, but it works like a charm!

DEC

A loop with 20xDEC had a high mark of 104.7 MIPS.

PUSH/POP

- PUSH and POP are slower operations, in the 80-85 MIPS range.
- But PUSHA/POPA are noticeably slow, in the 27 MIPS range.
- Using PUSHA/POPA everywhere will kill performance. We saw a 25% increase in speed after moving from PUSHA to push (reg).