The Paradox of the Perfect Virtual Computer

Why?

Why not just write a C program for your desktop or use Emscripten if you want it on the web?

Because then you have to worry about portability and software lifecycles. What happens when they release the next Windows? Software dies after a few generations, by hook or by crook. It's the circle of life.

If you write a program for the SD-8516, it will live forever. And, you can write it in C, if you want. The slowdown is about half. That's the price you pay for an emulated system. But ask yourself, are you really using all of your modern computer's power? If the answer is no, I make the bold statement that **the SD-8516 is a better choice, as a development platform, that *any* other computer system.**

Design Choices

It's a dichotomy to be sure. The duality of man.

On one hand, it *should* be “retro”. It's clear that a throwback is good. However, it *should* also be a serious attempt at computing.

It *must* fulfill the ubiquity requirement. I.E. everyone had a C64. Everyone had a SNES. That kind of thing. It has to have a popularity transcending some kind of niche. That means, in this case, it has to run everywhere and on everything. Two entirely compatable goals, because it means we do not need to compete with modern computers for speed.

This means the system itself is single core. The software can multitask. But this is a single user system. The system is intended to run one application at a time. Not an OS with multiple things going on, like 100 different background theads. That *could* happen, but, it's not the design goal.

It must know itself. It is not real hardware. Attempting to be RISC is wrong. There are no cycles. Cycle accuracy was an artifact of hardware and it was unintentional. We do not need it.

But we are not really trying to invent a completely new programming language, although that is *essentially* what this is; a game library attached to a programming language. However, it is intended to represent the interfaces of yesteryear. This is important because it makes the box general purpose. Thus fulfilling the goals of a) retro and b) general purpose.

We do not aim for speed and power directly. It comes, we do optimize, but the goal is not to get 1000 mips. The goal is to get enough mips to cover anything in the early era. This means 1 to 100 is fine. As a CISC machine with no cycle count this is fine; 10 mips on the SD-8516 is worth 30 mips anywhere else; likely a little more due to instructions like CASETAB, SKPC, CVTAN and so on.

Early Attempts

Early attempts tried to mimic hardware a little too much. The SD-8510 has bank registers. This was a total failure. Later, the SD-8516 had dual register instructions, which evolved into paired registers to get 24 and 32 bit general purpose registers, to be used as pointers. This was also a failure. The organization of the kernal in bank 1, and using memory mapped IO was also problematic (but somehwat workable). The ideal solutions to these were to expand the register file, increase the RAM slightly (to 512k or 1M) and provide an IO bank and/or video bank. This would lift the IO out of main memory without radically changing the IO interface.

The other option was to create some kind of device profile; i.e. a bus of 256 or so devices, which could be read from or written from. These are like registers; a separate memory area for IO. I.E. read from the keyboard status register. It could be done like:

```
READ $56   ; read from port $56 into device/port A
WRITE $56  ; write from A into device/port $56
```

This leads us towards;

```
READ A, [$56]   ; Load from $56 into A -- this looks like a LD...
```

And it is a LD, but from a different, protected memory region. Like a fast IO cache. The alternate is to define functions directly like

```
READKEY A         ; read a key into A
PIXEL X, Y, C     ; as an actual opcode
```

These are like accelerators, but they don't feel like opcodes at all. More like a programming language. The fact is, in the end, that's what this really is. However, also, it isn't. It's a computer. It's just a computer that is running on software. What's so strange about that?

The closer you get, the farther it is

The closer you get to how a modern computer operates, the farther you get from success for a project like this. **This is not a modern computer** nor should it be. That doesn't mean it should be poorly designed. In fact this should be as capable as it could be; including 64 bit registers. But wait, didn't you just say…

“Retro done right.”

Why yes, yesI did. The issue here is constraints. The SD-8516 aims to be very hand-assembly friendly.

8 bit

If you want to copy the 8 bit era:

Use only AL, BL, CL, etc. and don't use too much ram. Just the choice to use 8 bit registers will help you. *Then again, look at what Steve Wozniak did. He wrote Sweet-16, to simulate 16 bit registers on an 8 bit machine.* So keep an eye on your speed. Set MIPS to 1, or, 0.5. Maybe even try 0.25. Deal with it. This is how it was. Make it work. Crashing up against the constraints is what made the old games great. It was about gameplay, not graphics.

16 bit

The 16 bit era is the main focus here, and will cover the SNES and early PC era. Use 16 bit registers. Ram, you can open up on a bit. RAM over 128k was common here.

32 and 64 bit

The key is not should the SD-8516 allow 32 bit, but how it should allow it so that it does not interfere with 16 bit hand assembly. The method is simple; use 32 bit registers and not 16. The proposed standard for 32 bit is to use register pairs; AB, CD, XY, IJ, KT and so on. There are eight of them. But the second proposal is 16 X registers such as AX, BX, CX.

The 64 bit registers could be called EAX, EBX and so on. We could even go to 128 bit via SA, SB, SC or any other convention.

Another convention is to have b0..b7, then w0..w7 (with b0 accessing it's low byte) and then p0..p7 (24 bit), d0..d7 (32 bit), q0..q7 (64 bit) and even s0..s7 (128 bit). With a cross-width MOV, this system side-by-side with the current system would also be very powerful. The key here is you do not need to use such a system. With some restraint, you can hand-assemble any kind of program you want.

How many cores? Protected mode?

The idea of CORES can be done in two ways. Either is transparent to the underlying system. The software must deal with this if it is allowed; from the standpoint of the state machine how it is implemented is meaningless. However; there must be something atomic about the system or there must be some atomic authority. Not everything can be re-entrant – just most things. Protected mode and memory management is just like this. These are all features of a multi-user operating system that is designed to accomodate management engines and all the *other* modern conveniences required by modern computing in a modern world. So be it, but we defer to the host for this. We do not need to do any of that. So long as we provide the ability to fork(), we are feature complete.

How we do this is irrelevant. I suggest a software solution such as a CPU “class” which we can then instantiate multiple of such, and have them co-ordinate – likely through that shared-memory IO space I proposed – or in some other way (but it will all come down to a shared memory somewhere, I suspect). This can let a protected mode program on one CPU request a second CPU to run certain code (or idle).

Luckily we do not need to worry about this at all. Almost any game can simulate such as thing with an

event que. Polling an event que even several thousand times per second is trivial; even for a program running on the SD-8516, which has a YIELD or otherwise hits the host with 241 batches per second; that alone is around a 4ms resolution and fine for almost any kind of UI except, perhaps, hair-trigger FPS shooters. Which are not really our problem domain. And, again, YIELD fixes that problem cleanly.

But why Retro?

You don't have to do that. That's the point. You can if you want. This is a power that only an emulated computer can have. We can have a very large register file. We are not trying to be a modern computer. The entire system fits in cache! RAM included. *We only need to be a computer.*

And, we are. We are retro done right, and we are also, just a computer.

The path forward

The original b0..b7 system seems more interesting and capable than EA, AX and EAX, precisely because it is separate. That splits 32/64 bit programming away from 8/16 bit based on the mode of the programmer. For C, it is less important; for C or any other language, pick the video mode and sound system that represents your era and set the system speed accordingly.

With this in mind, what changes are to be made?

- Simply the addition of the q0..q7 would aid.
- More video modes.
- Separate IO memory. This is looking more viable every day.
- The kernal has to be moved. Likely to the start of, or end of memory. The kernal interface is based on the INT system. Next, FREE RAM becomes easy to calculate; it's the start of kernal space. The big issue, I suspect, is the INT system.

Separate memory?

The first thing to do is create a special bank (of 64k?) which we can call cache. All IO will be run through this and not main memory. This memory is accessed only via special load and store commands. The key here is I don't know what those commands should be, or how they should work. Maybe a MOV command that can move to and from cache memory. But how does this work exactly?

- a) Since [] is used for a memory location maybe <> or () can be used for a cache memory location.
- b) A separate opcode that moves data between registers and cache memory

```
LDA <$E000>
LDA ($E000)
LDA [[$E000]]
```

This 'parallel memory' for IO and possibly video is interesting. A data-only memory, or, system/kernal-

only memory.

```
MOVC A, [$4000]
MOVC [$4000], A
```

The thing is, using [$4000] makes it look like something from RAM.

```
LDA &$4000
```

Using & could indicate it's from cache, but this means it would need to be a different opcode.