

// BUILDERSCON 2016

SIMULATING MINICOMPUTERS ON MICROCONTROLLERS

Hello!

My name is David.

I'm a software programmer and hardware enthusiast from Sydney, Australia.

I want to express my gratitude to Daisuke and the rest of the builderscon organisers for accepting my talk.



Normally I travel around the world and try to convince people to use the Go programming language.

But today I'm here for something entirely different.

I'm delighted today to be able to present a report on two retro computer projects that I recently completed



But before I begin, I want to mention that this talk is dedicated to the memory of my friend, Andrew Stone, who started the Raspberry Pi meetup group in Sydney which gave me the skills and inspiration to attempt the projects you're about to see.



The first project I want to describe a simulator of the PDP-11/40 that I built using Arduino hardware.

// AVR11

WHAT IS THE PDP-11?



In my opinion the PDP-11 is the most important minicomputer of the 1970's.

The PDP-11 was the machine that Ken Thompson and Dennis Ritchie used to develop UNIX and the C programming language.

Every computer you interact with today can trace itself back to the the architectural style of the PDP-11 and the work of Ken and Dennis.



As a historical artefact the PDP-11 has tremendous importance for anyone who is interested in retro computing or computer programming in general.

The PDP-11 was the predecessor to the VAX-11/780, the first 32bit minicomputer, where BSD unix started.

Dave Cutler, the author of VMS, the other operating system for the 11/780 left DEC to join to Microsoft.
Cutler took VMS with him when he left Digital forming the conceptual basis of Windows NT.

So, what better way to learn about the PDP-11, and the history of C and Unix, than to build a simulator of the machine that started it all ?

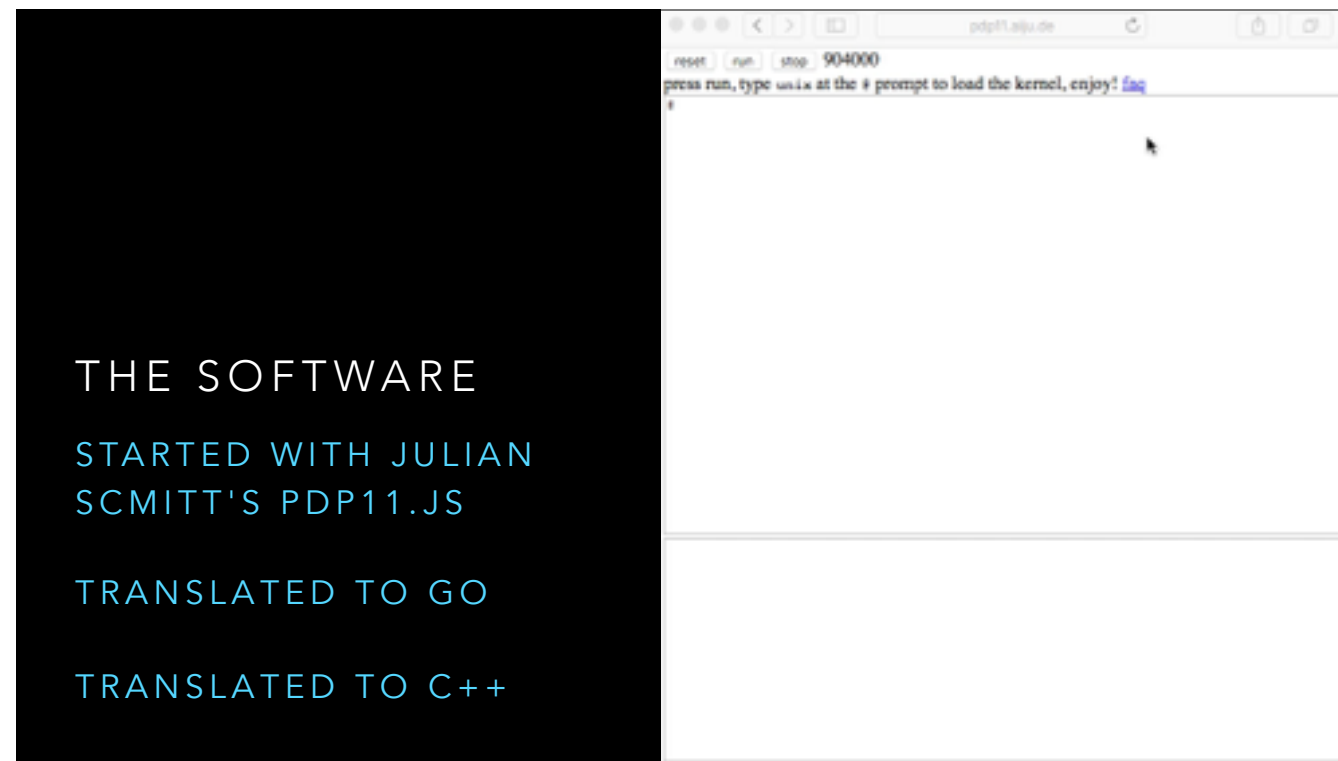
THE DOCUMENTATION IS AMAZING



Another reason to learn more about the PDP-11 is the documentation; it's amazing.

Bitsavers.org has every user manual, every reference manual, and most of the schematics for all of the DEC PDP line of computers.

They also have disk images, tape images, diagnostic programs, everything you will need.



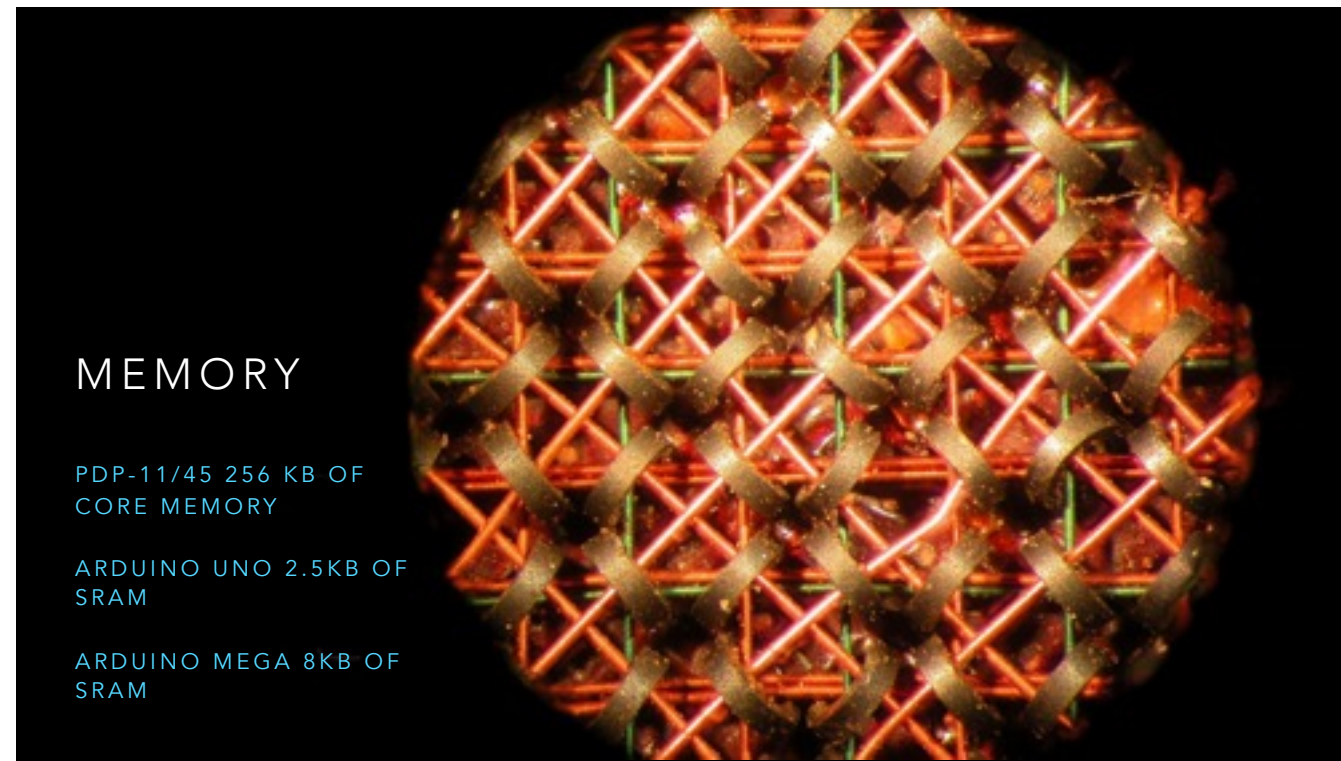
The idea for this project started when I saw Julian Scmitt's pdp11.js project.

(click, click)

This is a short movie of it pdp11.js running inside a web browser running the original UNIX version 6 code on a simulate PDP11.

I translated Julian's code from javascript to Go, because I knew Go better than I knew C++, and when I had the simulator running in Go I ported the code to C++ to run under the arduino environment.

I don't have time to go into the JS -> Go -> C++ translation here, but I gave a presentation about it a few years ago. There is a link at the end of the presentation.



Ok, so now we have a software emulator for the pdp11 running on an arduino, now we have to load a program into our simulated computer.

[click]

The PDP-11 had an address space of 128 kilo words. Each word is 16 bits, so in today's language we'd say the address space is 256 kilo bytes of RAM

If you've played with a microcontroller you'll know they don't have much memory

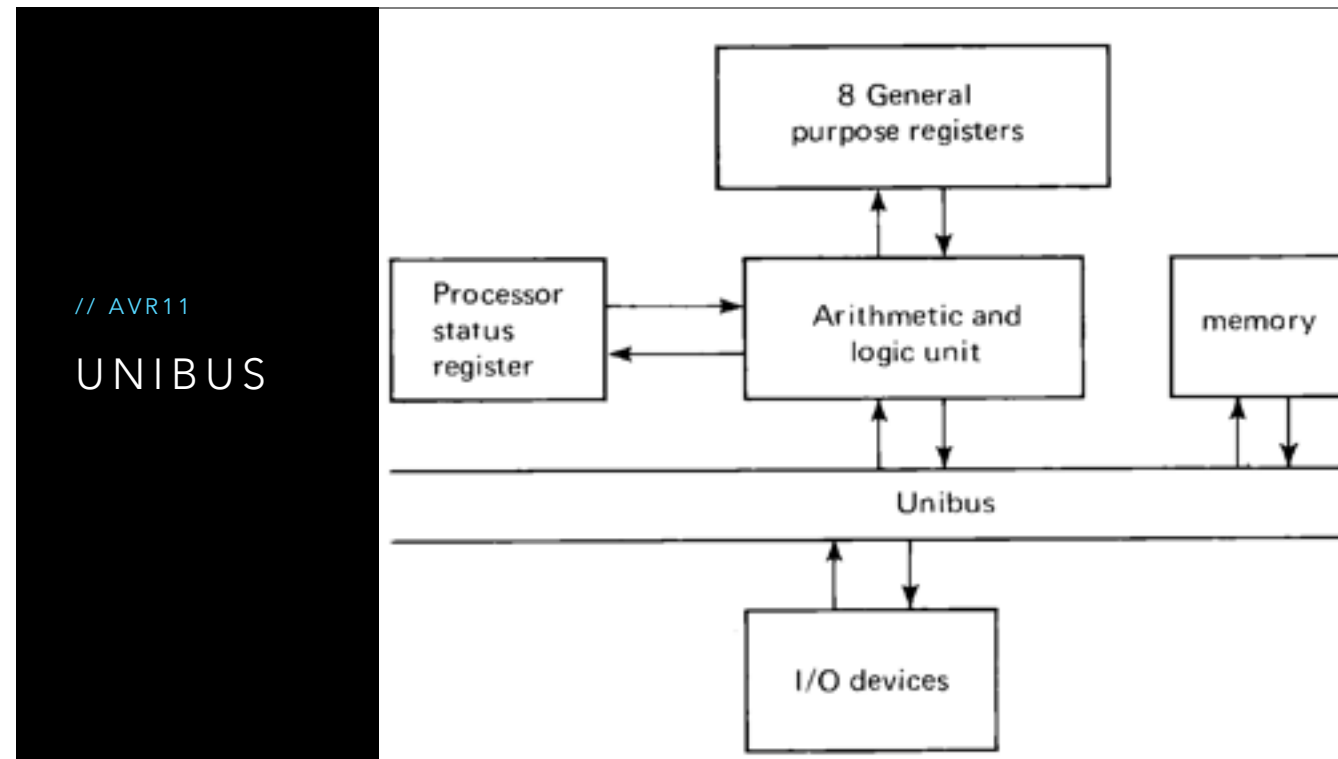
[click]

The arduino uno only has 2.5 kb of SRAM

[click]

The arduino mega, the platform i chose, has 8 kb of SRAM

But it turns out there is a solution, and to talk about it we need to talk about how memory works in the PDP-11.



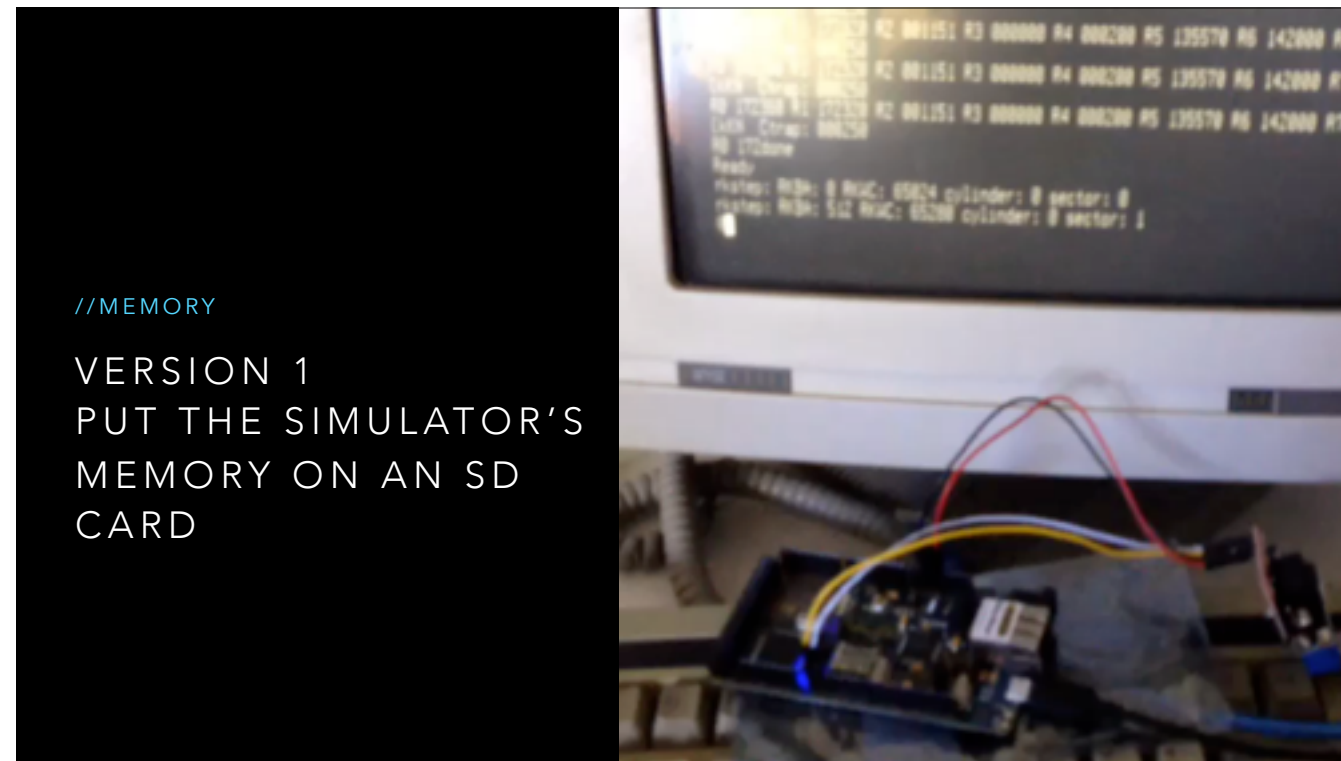
The PDP-11 architecture is very interesting.

All the the major system components, including memory, work asynchronously, co-ordinating access via a shared backplane called the UNIBUS.

This is a little bit different to the way we think about computers today, with the CPU at the top of the tree and everything descending from it.

If you look at this diagram, It almost looks like an ethernet network, with various terminals all communicating across a shared bus. There are no hard guarantees about who will have priority or how much bandwidth you will have.

Because of this property we only need to simulate the instruction decoding and execution which operate on the 8 internal registers of the PDP-11 (plus a few other system registers), and memory can be implemented externally.



The revelation to me was, from the point of view of a CPU, memory is not part of the processor.

Any instruction that references memory always boiled down to a read or write to the UNIBUS.

In short, there is no way for the CPU to observe memory directly, it has to ask the UNIBUS to read or write data on its behalf, and this gave me the opportunity to solve the problem of limited memory available on the arduino devices I had access to.

I was already using the SD card to simulate the RK05 disk pack.

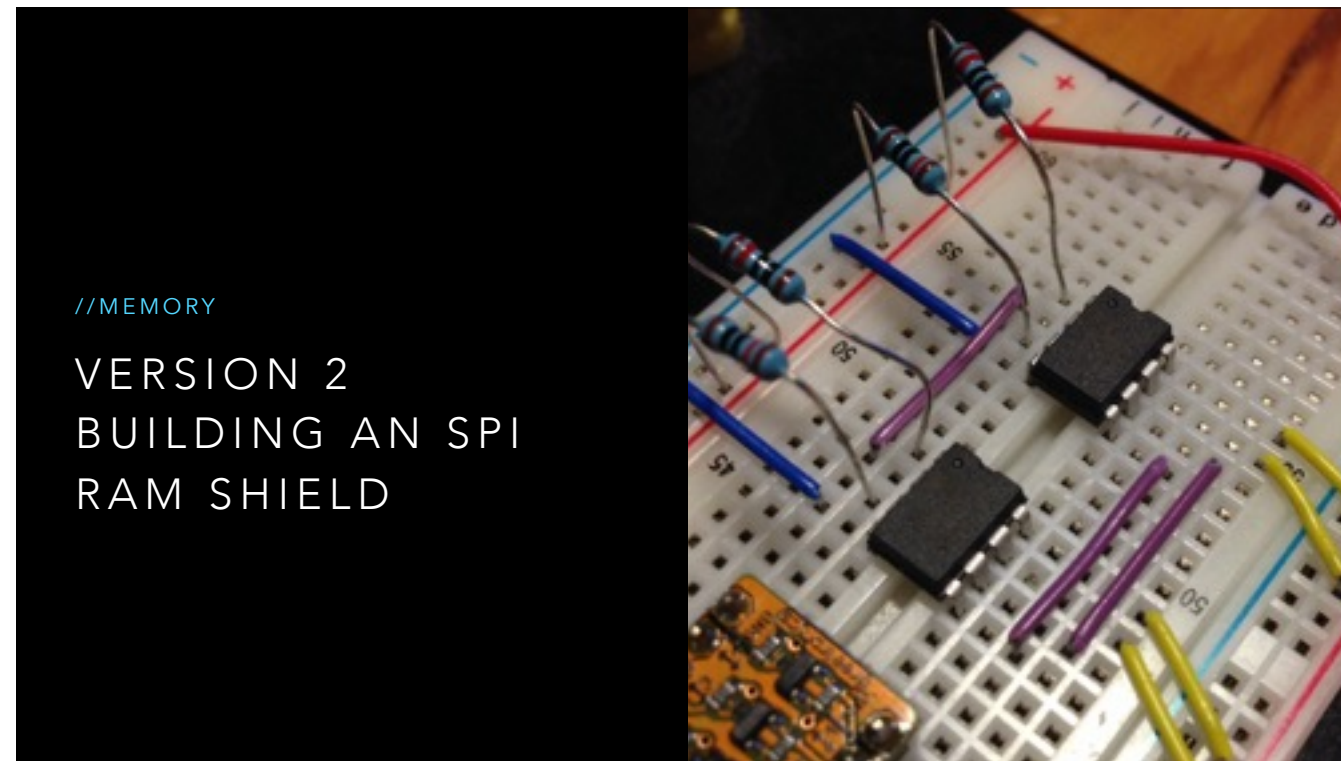
why not just make another image file and use that to back the core memory. The SD card probably wouldn't last very long, but I have a pile of cheap cards so why not try it.

Amazingly it worked, and although it was very slow I was able to use this technique to boot the simulator a very long way into the Unix boot process. This is a little video I took at the time; trust me, this is very slow.

(click, keep talking)

Even more amazingly I didn't wear out the mini SD card, and still haven't. This is due to a property of the SD card itself

All SD cards mandate that you read and write to them in units of pages. Pages happen to be 512 bytes



So that was version one, using an SD card to simulate the entire address space. This was a very 1950's solution and came with matching performance.

I needed a way to connect real memory to my arduino; but the arduino doesn't have enough pins to drive a real static ram chip.

There is a common SRAM chip, the Microchip 23K256, which is a 32 kilobyte chip with an SPI interface.

The 23K256 isn't as common in Arduino designs because of one major flaw; it's a 3v3 part which is incompatible with most of the arduino ecosystem.

There was also the problem of capacity. To get to 256 kilobytes I would need 8 chips on the same SPI bus each with their own chip select line, consuming the scarce IO pins available.

Luckily Microchip sell another chip, the 23LC1024 which has 4 times the capacity, and can operate at 5 volts. This meant I would only need two chips to get 256kb.

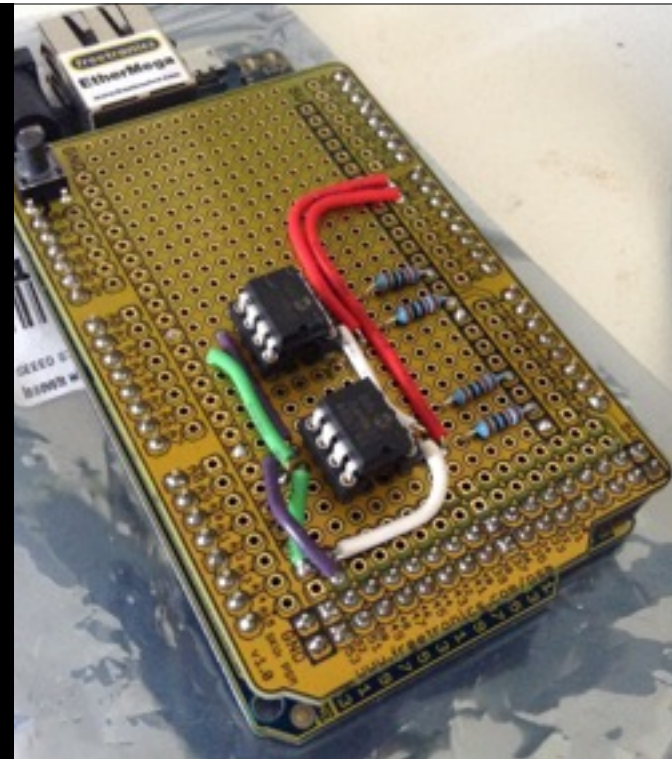
Lucky I found the last two chips in stock at a Element 14, and ordered them straight away.

This is a picture of my first attempts to integrate the 23LC1024s on the breadboard which weren't very successful. Even though I followed the application note I wasn't able to get the chips to reliably pass the SRAM test. Sometimes the data would be written perfectly, other times it would just be garbage.

By default the 16Mhz Atmel parts drive the SPI pins at 4Mhz. From reading other blogs it was clear that this sort of frequency is outside what the breadboard is designed for, not to mention the noise and capacitance of the large patch leads.

// MEMORY

VERSION 2 SPI RAM ARDUINO SHIELD



So I did. This is a standard arduino prototyping shield which I built at my local hacker space.

I took the shield home, plugged in the chips and both banks worked first time! So that was the memory problem solved.

Next, I loaded the avr11 sketch and discovered that the micro SD card had failed to initialise. Reloading the sketch, the SD card worked fine, but the SRAM test showed garbage.

The source of the problem turned out to be the default state of the digital pins on the Arduino. The way SPI works is all the components on the SPI bus share three lines, MISO (master in, slave out), MOSI (master out, slave in), and SCLK (a clock line driven by the master). Additionally every device has its own Chip Select line which must be held high to inhibit the device unless you want to talk to it.

To talk to an individual device, you lower the CS line connected to that chip and read and write data on MOSI/MISO, toggling the SCLK line. All the other devices which have their CS lines high are supposed to hold their MISO and MOSI at a high impedance and ignore transactions on the bus.

The problem is, when the Arduino resets, all the digital lines are set to input and are low; you don't want an Arduino with no sketch loaded suddenly sending 5volts out of every digital pin. In effect all the Chip Select lines could be active, meaning all the components are listening to the transaction and trying to interact with the master.

The solution I came up with was to ensure that all the digital pins are set to output and held high before calling any of the SD.begin() or SPI.begin() functions. In effect this disables all the SPI devices until their various begin() functions were called to configure them.



The RK05 disk drive is a 2.5 megabyte removable disk drive which stores the root file system for UNIX.

Like most minicomputers of the time, the PDP-11 processor used memory mapped IO.

A range of addresses, at the top of the address space, called the IO Page holds addresses that represent IO cards connected to the UNIBUS.

The RK05 controller is a UNIBUS device that responds to commands over a few memory addresses in the IO page.

From the point of view of our PDP simulator, the CPU writes a command to an IO address, then waits for an interrupt telling it the disk operation is complete.

The simulator catches that write to the IO address on the UNIBUS, sends it to the sd card where it reads the block from a file stored on the FAT file system into memory, and sets the interrupt flag which checked each time the simulator simulates an instruction; just like a real CPU



Our simulator also needs a way to communicate with the user via serial console.

Just like the RK05 disk drive, the console works the same way, the PDP-11 writes a word to the memory address where the serial console is expected to be present; we catch that word and copy it to the arduino serial port library.

Similarly, when a character is received over the arduino serial port, we store that character in the memory the PDP and send an interrupt.

// INSTRUCTION TIMING FOR PDP-11
WITH SEMICONDUCTOR MEMORY

HOW FAST IS IT?

SOURCE ADDRESS TIME

Instruction	Source Mode	SRC Time (A)	Memory Cycles
	0	0.00 μ sec	0
	1	.78	1
	2	.84	1
Double	3	1.74	2
Operand	4	.84	1
	5	1.74	2
	6	1.46	2
	7	2.36	3

NOTE (A): For Source Modes 1 thru 7, add 0.34 μ sec for Odd Byte instructions.

DESTINATION ADDRESS TIME

Instruction	Destination Mode	DST Time (B)	Memory Cycles
Single	0	0.00 μ sec	0
Operand,	1	.78 (.90)	1
and	2	.84 (.90)	1
Double	3	1.74 (1.80)	2
Operand	4	.84 (.90)	1
(except	5	1.74 (1.80)	2
MOV, JMP, JSR)	6	1.46 (1.74)	2
	7	2.36 (2.64)	1

NOTE (B): For Destination Modes 1 thru 7, add 0.34 μ sec for Odd Byte instructions. Use higher values in parentheses () for ADD, SUB, CMP, BIT, BIC, or BIS and a Source Mode of 0.

So, the big question is, how fast was my simulator.

The PDP-11 doesn't have a "clock speed", in the way we're used to these days. The "speed" of the PDP-11 is defined by how fast its memory works; faster memory, faster computer.

I recently came across the 1972 PDP-11/40 processor handbook which provides formulas for calculating instruction timings.

So, now we can compute how long a PDP-11/40 took to execute an instruction, maybe this could be used to give some idea of how well avr11 was performing in simulation.

Because avr11 runs directly on the Arduino, there is no simple way to measure the performance of various pieces of code externally, you don't have an operating system or a profiler, so you can't tell how fast my simulator is running.

// SPEED

TIMING MY SIMULATOR BY TOGGLING A PIN



A reader of my blog suggested this, each time through the instruction loop, I raise a pin at the start and lower it at the end. This would give me a "clock" of sorts that I could time with a frequency counter.

This is a picture of an early version of the frequency counter I built using another arduino, I'll talk more about frequency counters later.

So, how fast does my simulator run?

```
// ARDUINO MEGA 2560 SPEED
```

11,000 HZ

11,000 hertz, and that took a lot of optimisation to reach even that number.

```
// AVR11
```

GOING FASTER: UPGRADE TO THE ARDUINO DUE



As much as I love the minimalist idea of building a '70's era mini computer on an 8 bit microcontroller, it looks like this just wasn't going to be practical to build a usable simulator on the 16mhz Atmel 2560.

So, I ordered the Arduino Due, which has a full 32bit ARM processor and runs at a much higher clock rate, 84Mhz. The due also has more on board ram, although it's not quite the full 256kb required for UNIX.

The night the Arduino Due arrived I modified avr11 to run on it. The result, with just a recompilation of the code for the SAM3X processor; 85,000 instructions/second.

```
// ARDUINO DUO SAM3X SPEED
```

85,000 HZ

~8X SPEEDUP

Depending on how you cut it, this is between 5 and 8 times faster

PERFORMANCE COMPARISON

INSTRUCTION	PDP-11/40	ARDUINO DUE	RELATIVE PERFORMANCE
ADD R0, R1	1,000,000 HZ	85,344 HZ	8.5%
ADD (R0), (R1)	301,204 HZ	63,493 HZ	21%

So, how does that compare to a real PDP-11?

I modified avr11 to execute ADD R0, R1 over and over again (effectively disabling the program counter increment) and timed the results. Looking at the table from the previous slide, we know this instruction takes one cycle for the instruction itself, and no cycle for the operands; as they are registers

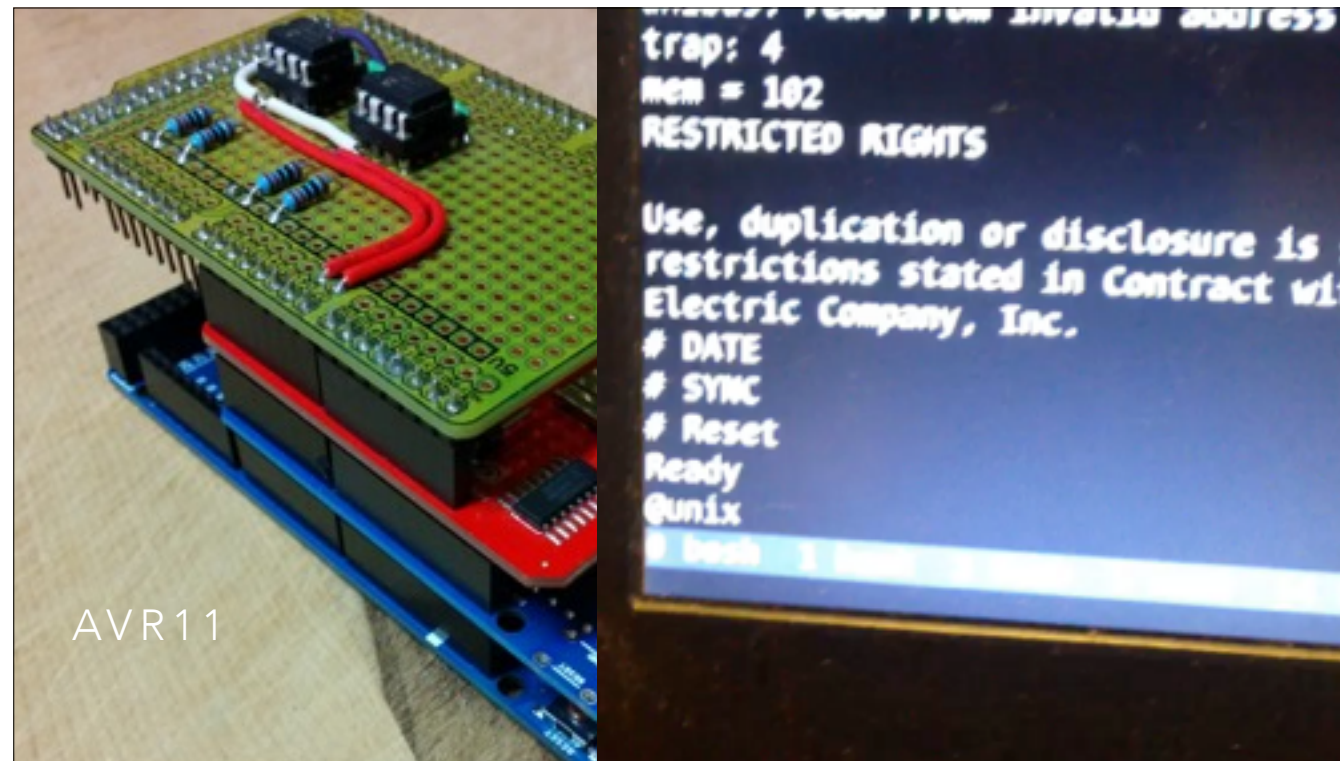
The PDP-11 execute this at 1 million hertz, a megahert.

Compared to the Due, which is barely 8.5% of that, which isn't great.

However, that was for a best case instruction with no operand overhead. What if the instruction was more complex, for example ADD (R0), (R1), add the value at the address stored in R0 to the value in the address at R1. Using the tables above the timing on a real PDP-11/40 would have been 3.32 microseconds because it takes three read cycles, and one write for the result, 3.32x times slower, just over 300,000 instructions a second.

The due is still much slower, but relative to the PDP-11 speed, it's over 20%.

So, perhaps all is not lost. Being able to deliver 25%, 30% of a real PDP-11/40, depending on the instruction stream this might be a useable simulator.



This is the final form of the Avr11, using an Arduino Duo, SPI Ram board and Sparkfun SDCard shield.

// APPLE 1

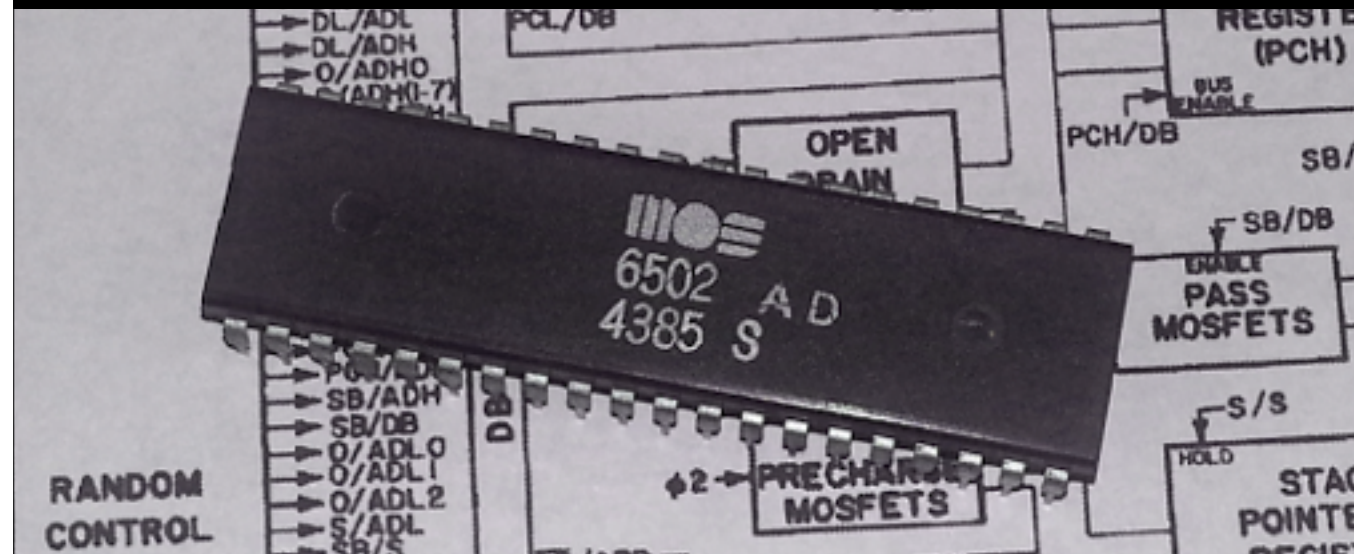
SIMULATING THE APPLE 1



The second project I want to talk about is an Apple I simulator i built, using an arduino compontents.

// 1975 MOS TECHNOLOGY

MOS 6502



The heart of the apple 1 was the legendary 6502 microprocessor design by Chuck Peddle and introduced in 1975 by MOS technology.

The 6502 was _the_ CPU for the 1970's and well into the 1980's.



The 6502 powered the Nintendo Famicom

The VIC-20 and C-64

The Atari 2600, Atari 400 and 800

Apple II, and before it the Apple-I

This is what we're going to focus on.

// APPLE 1

61 CHIPS



The Apple 1 was essentially a 6502 computer with 4k of RAM and 256 bytes of ROM.

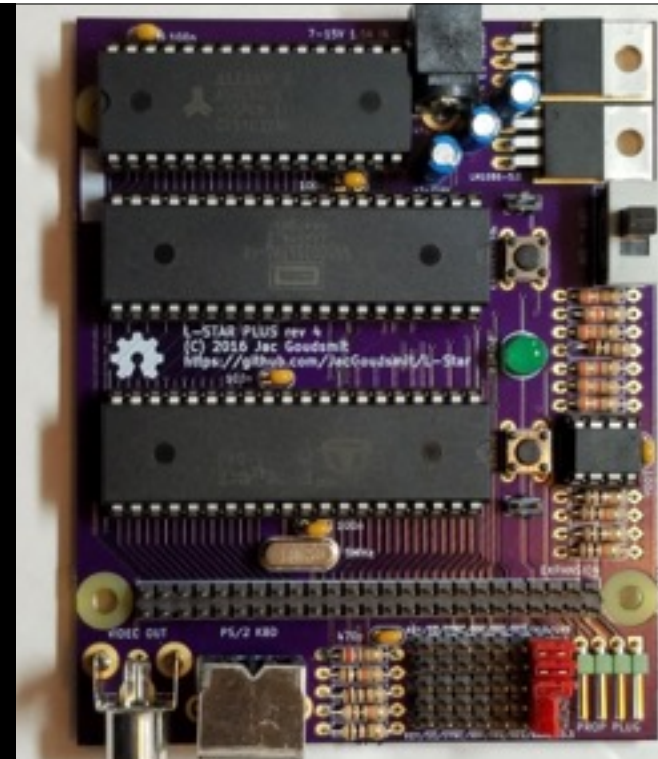
Just supply your own power supply, keyboard, composite monitor, and you were in business.

The Apple I used 61 discrete chips

The good news is we can emulate the RAM, ROM, PIA, and all the glue logic with a single microcontroller.

// L-STAR SOFTWARE DEFINED COMPUTER

4 CHIPS

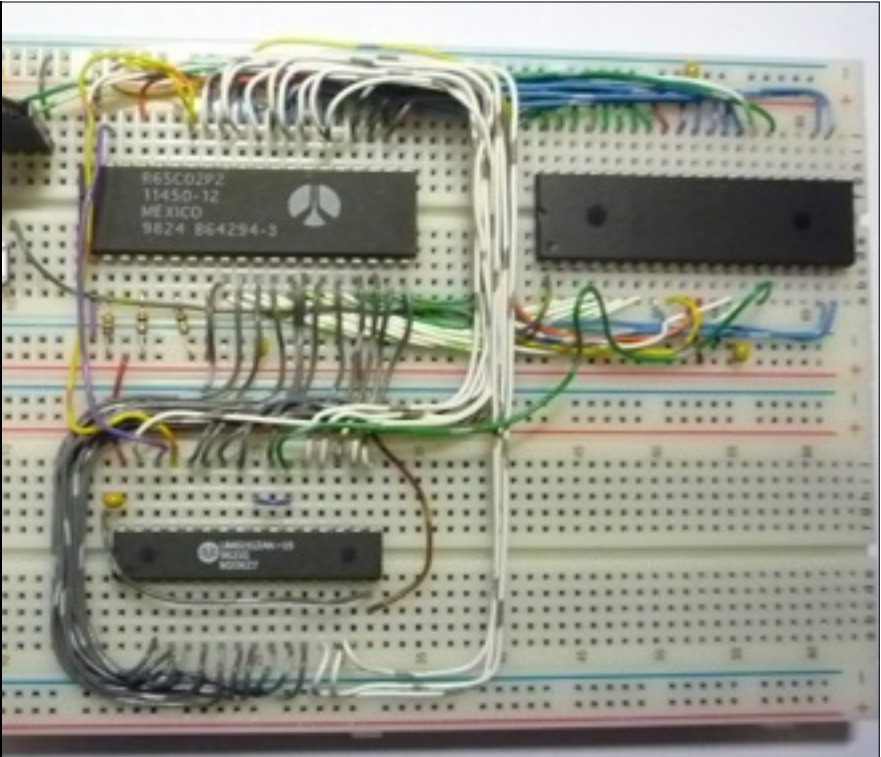


I'm not the first person to try to do this, This is the l-star software defined computer (which has been featured on hack a day a few times)

In addition to the 6502, it uses a propellor micro computer, and EEPROM to store the code for the propeller, and an SRAM chip.

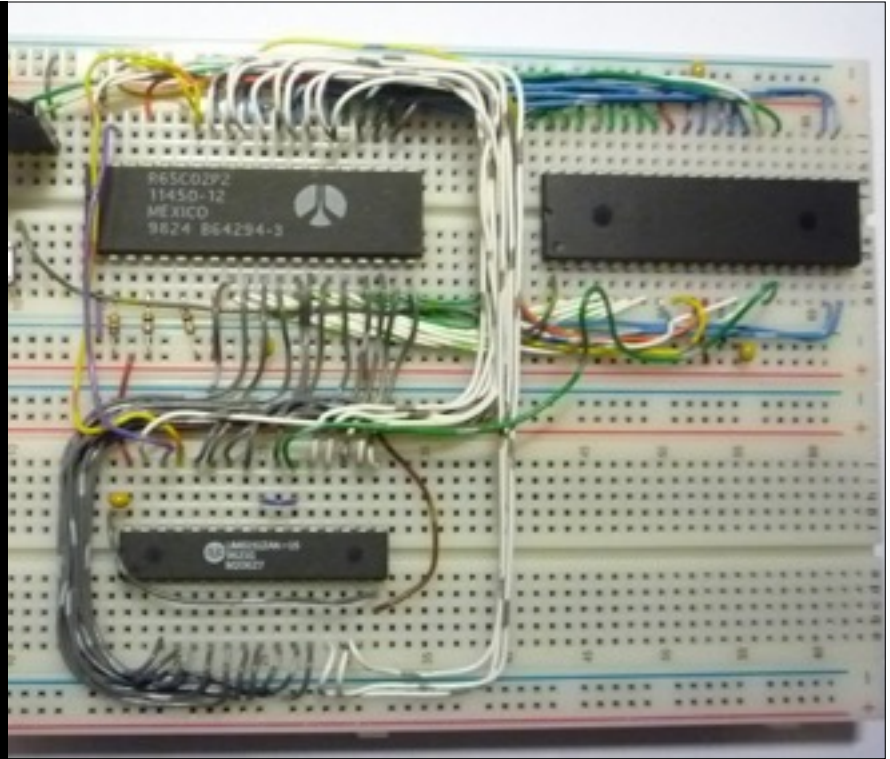
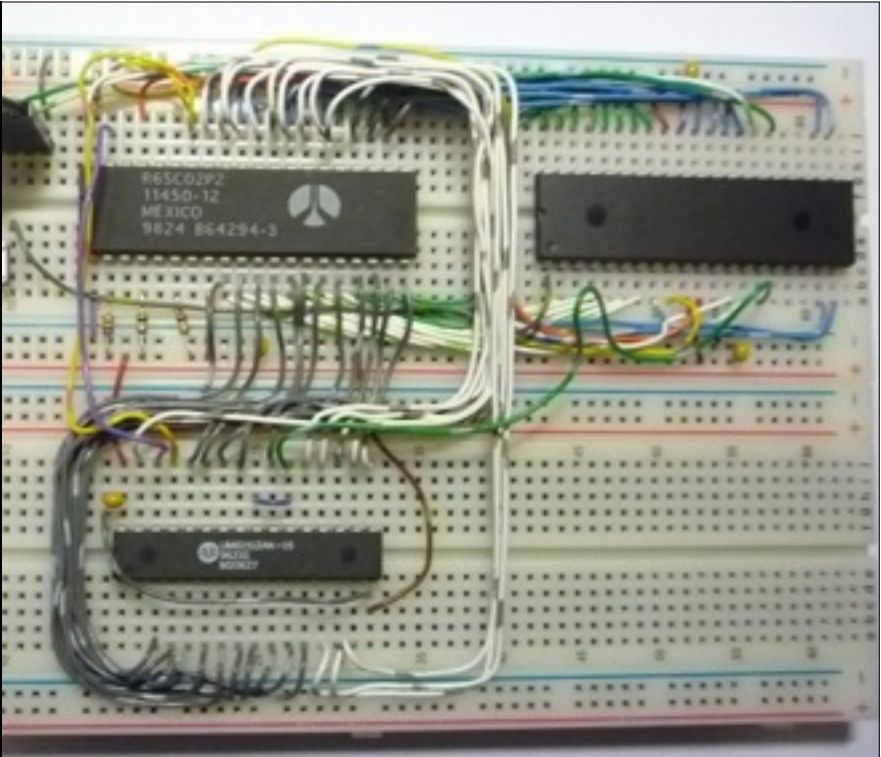
// PIC MICROCONTROLLER

3 CHIPS!

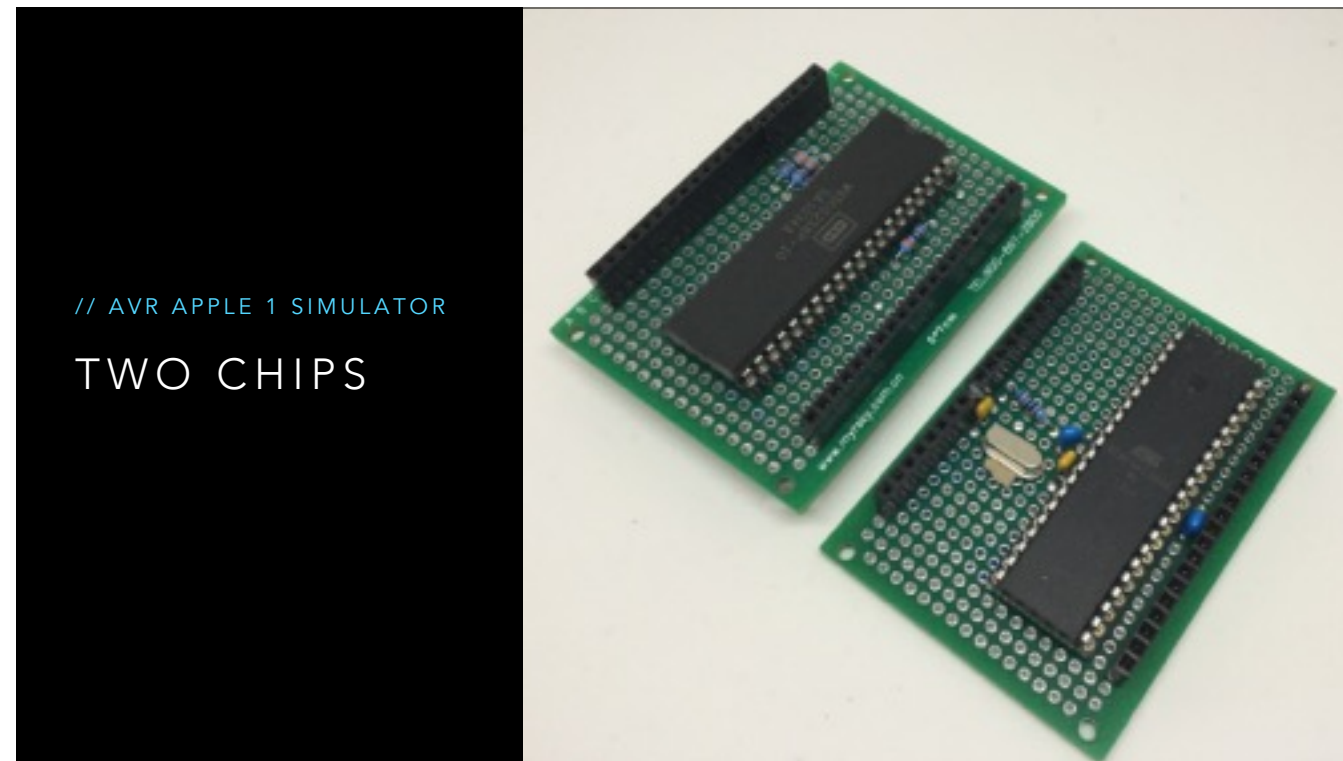


// PIC MICROCONTROLLER

3 CHIPS!



Here is another example that uses a PIC micro controller to replace the glue logic between the 6502 and the memory on the bottom left of the picture



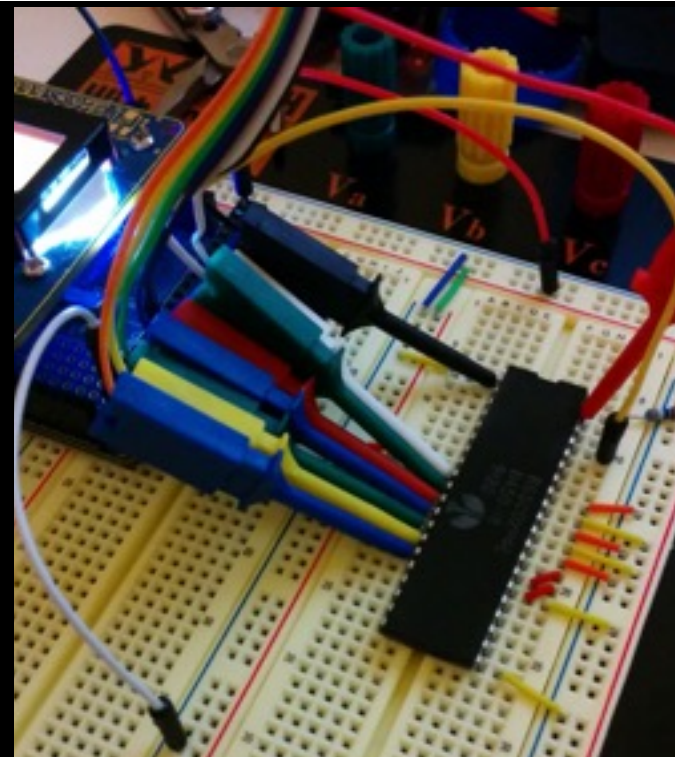
This is my simulator. It uses two chips, an atmega 1284p and a rockwell 65c02 on two boards

It runs at about a third of the speed of the original apple 1

But i'm skipping ahead a bit, lets take a step back.

// HARDWARE

ARDUINO PROVIDES THE 6502 CLOCK



Unlike the previous project, this is not a simulator. This is a real 6502 executing real machine code.

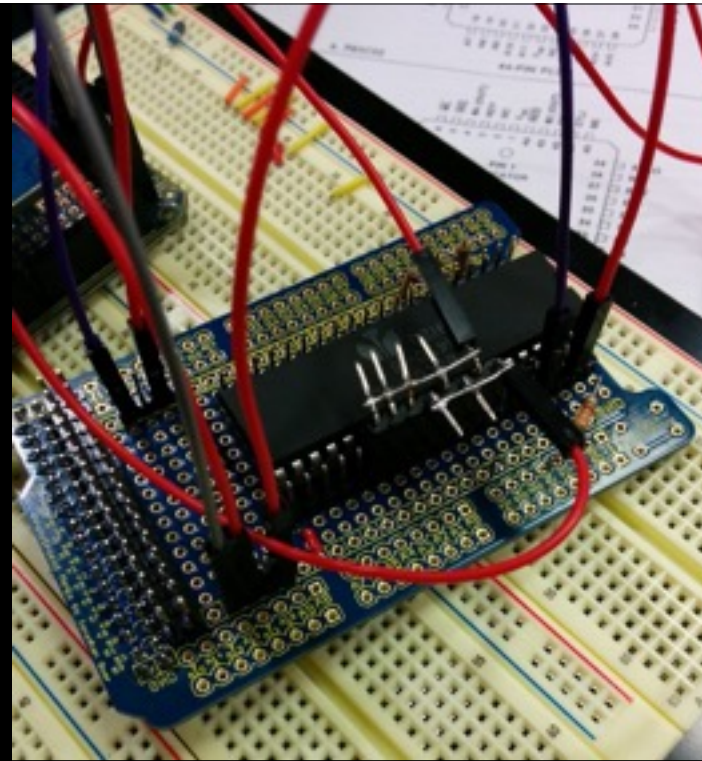
The plan was the rest of the “computer”, the ram, rom, display, keyboard, would be provided by the Arduino.

To validate the idea that an Arduino could provide a stable clock for the 6502, I started by breadboarding the project.

The result was a success, with a tight assembly loop I was able to generate a 1Mhz clock with a 50% duty cycle and I could see the address bus of the 6502 counting up.

// HARDWARE

PROTOTYPE ARDUINO SHIELD



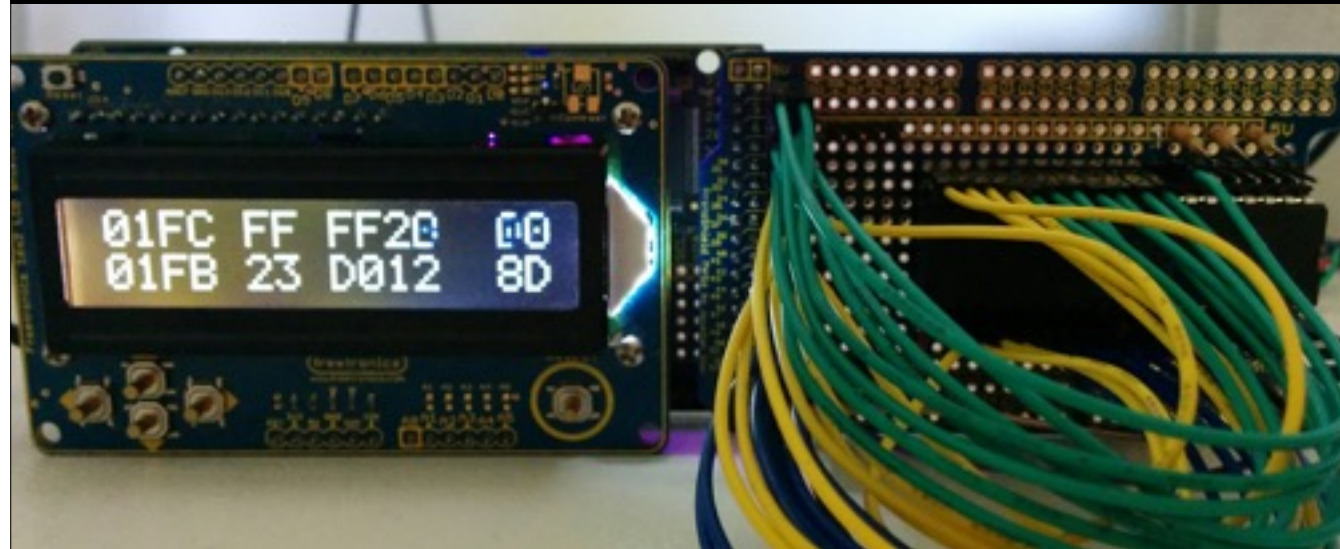
Next I built a prototype using an arduino shield.

The shield has connectors for the 40 pins on the 6502 and the 40 something pins on the Ardunio Mega's expansion header allowing me to jumper between the 6502 and the Arduino.

The strange jumper block presents the NOP instruction on the data bus unconditionally, this is called free running mode.

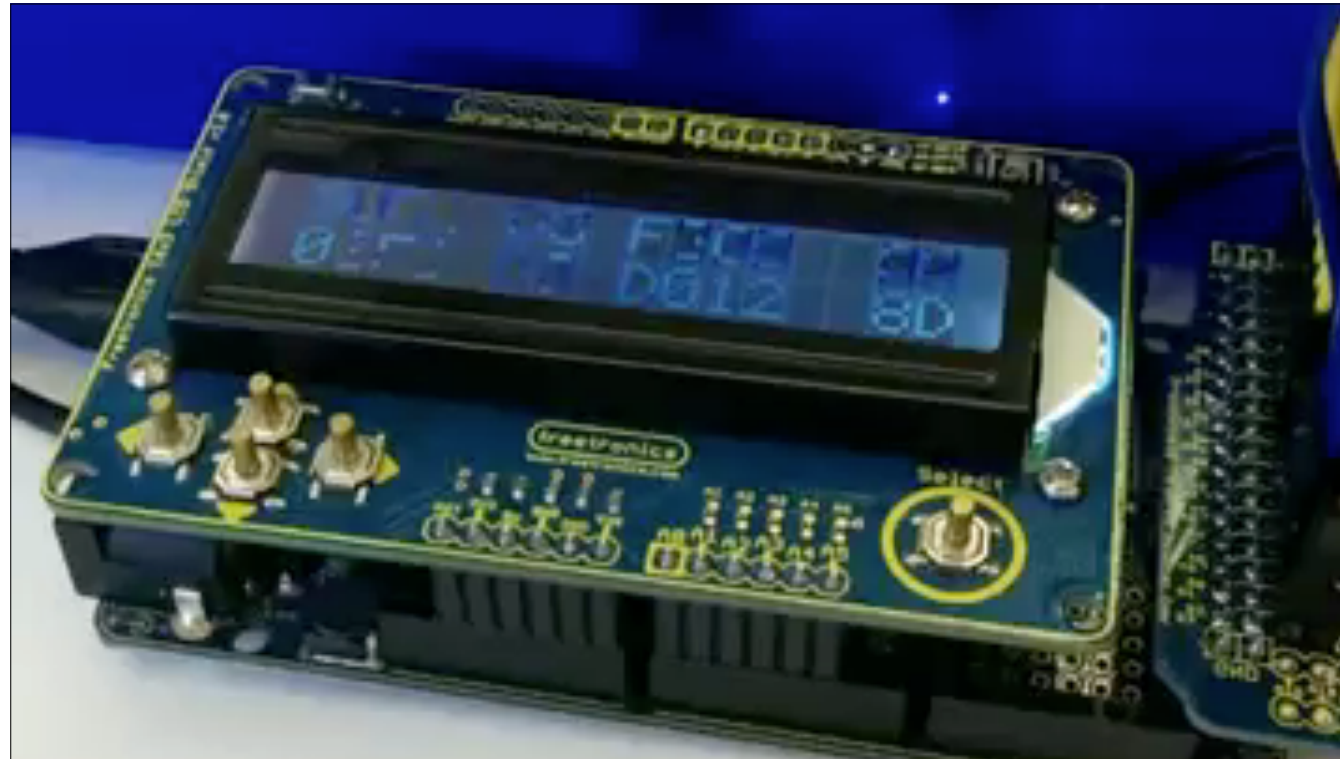
// HARDWARE

THE SIDECAR



Because I wanted to use an LCD panel for debugging and the patch wires on the protoshield would not fit under the LCD shield I mounted the shield backwards and upside down. I called this prototype design the “sidecar”.

In this picture you can see some status from the arduino reporting the values of the last few memory reads and writes.



Here's a small video I took when I got it all working.

The cpu is operating at a very low clock speed, but you can see it's running a basic program and the output is coming out to my laptop over the arduino serial terminal.

ROM \$FF00

6502 HEX MONITOR LISTING				
FF00	D8	RESET	CLD	Clear decimal arithmetic mode.
FF01	58		CLI	
FF02	A0 7F		LDY #\$7F	Mask for DSP data direction register.
FF04	8C 12 D0		STY DSP	Set it up.
FF07	A9 A7		LDA #\$A7	KBD and DSP control register mask.
FF09	8D 11 D0		STA KBD CR	Enable interrupts, set CA1, CB1, for
FF0C	8D 13 D0		STA DSP CR	positive edge sense/output mode.
FF0F	C9 DF	NOTCR	CMP #\$DF	"←"?
FF11	F0 13		BEQ BACKSPACE	Yes.
FF13	C9 9B		CMP #\$9B	ESC?
FF15	F0 03		BEQ ESCAPE	Yes.
FF17	C8		INY	Advance text index.
FF18	10 0F		BPL NEXTCHAR	Auto ESC if > 127.
FF1A	A9 DC	ESCAPE	LDA #\$DC	"\".
FF1C	20 EF FF		JSR ECHO	Output it.
FF1F	A9 8D	GETLINE	LDA #\$8D	CR.

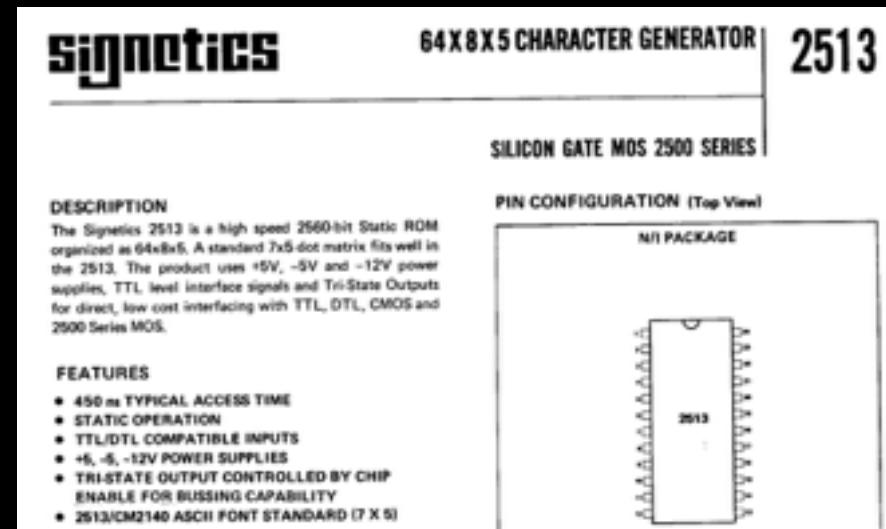
The Apple 1 shipped with a 256 byte assembly language monitor, called the Woz rom.

The Apple 1 manual even provided the source of the ROM

The Arduino intercepts memory reads for the \$FF00 page and looks up the rom bytes from a table compiled into the program.

I have tested a few of the popular ROM images like A1Assembler and Applesoft-lite and they work nicely as well.

INPUT // OUTPUT



The Apple 1 interfaces to the keyboard and screen via four registers from the 6821 PIA chip mapped into the address space at \$D000.

In the apple 1, when a key is pressed on the keyboard, the high bit of \$D011 is latched high, this can be detected by the 6502 ROM monitor which then reads \$D010 to fetch the keycode, which is conveniently encoded as 7bit ASCII.

This is similar to the way the console worked in the PDP-11.

Output is similar, the 6502 polls \$D013 until the PIA reports that the video encoder is not busy then the character to write to the screen is placed in \$D012.

It is straight forward to map these reads and writes to these addresses to the Arduino serial port.

CLOCK SPEED COMPARISON

MOS 6502

1 MHZ CLOCK

ATMEL 1284P

16 MHZ CLOCK

The original Apple 1 ran at a clock speed of 1 Mhz

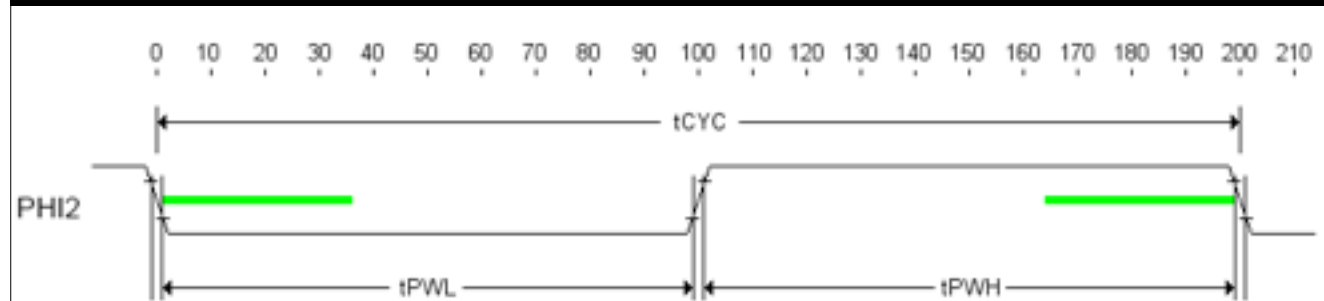
Most arduino's have a clock speed of 16Mhz. In this project the Arduino is responsible for providing the clock signal to the 6502.

This means that for every cycle we want the 6502 to execute, we have only 16 arduino cycles to

- lower the clock signal
- wait a few cycles (otherwise we overlock the 6502)
- raise the clock signal
- decode the the address and depending on the state of the read/write line either fetch or place data on the data bus
- lower the clock so the 6502 reads the data we just placed on the data bus

The problem is, even at 16 mhz, there simply isn't enough time because most of the arduino instructions take two or three clock cycles to complete.

// CLOCK STRETCHING



But there is a solution.

Different 6502 models have different requirements for the minimum and maximum length of their clock pulse.

The original NMOS 6502 required a clock of at least 100kHz to avoid losing internal CPU state, which made single stepping more complicated.

With the Rockwell 65c02 I am using the requires the clock low phase must not exceed 5 μ s, but the clock signal can remain high indefinitely.

We can use this property to generate a short low clock, then raise the clock and do our processing, even take an interrupt, at our leisure

Because I have the 4Mhz 65c02 version, we can even make the clock low period shorter, to allow our high pulse to take longer in an effort to reach the 1Mhz clock target.

This is called clock stretching.

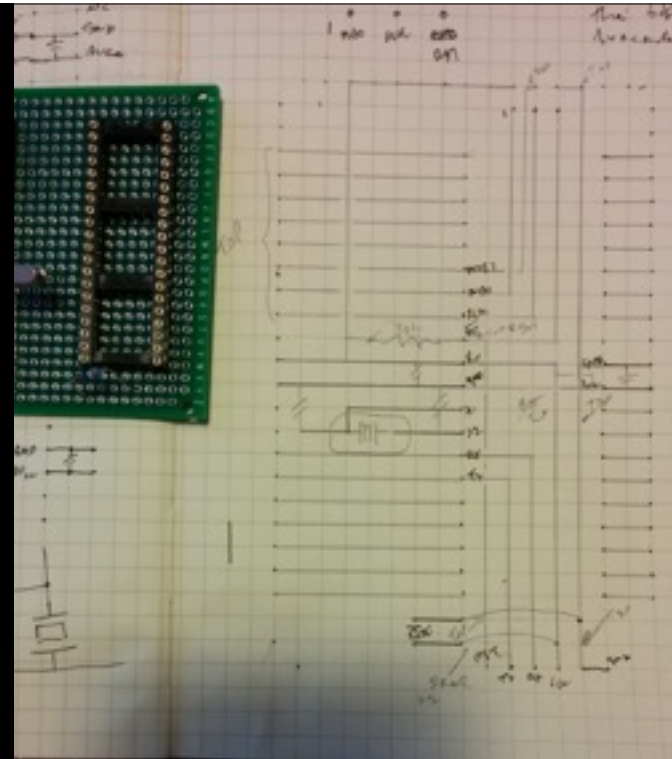


Because the arduino is providing the clock, a nice strong 5volt signal, this gave me an easy way to measure how fast my computer is running.

So, similar to the last project, I built a frequency counter, this time it has an LCD shield for the display.

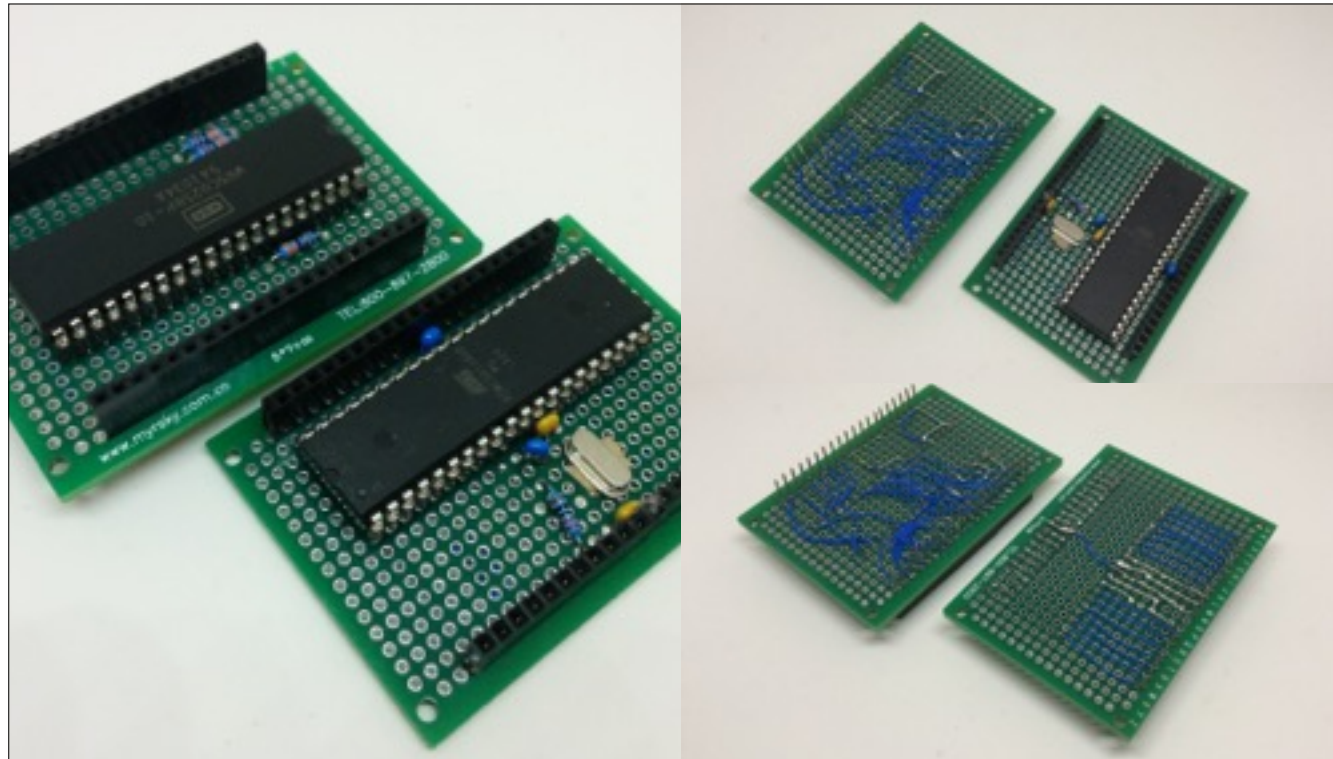
The frequency counter is good up to a few Mhz, which is way faster than my apple 1 runs.

DESIGNING A FIRST PROTOTYPE



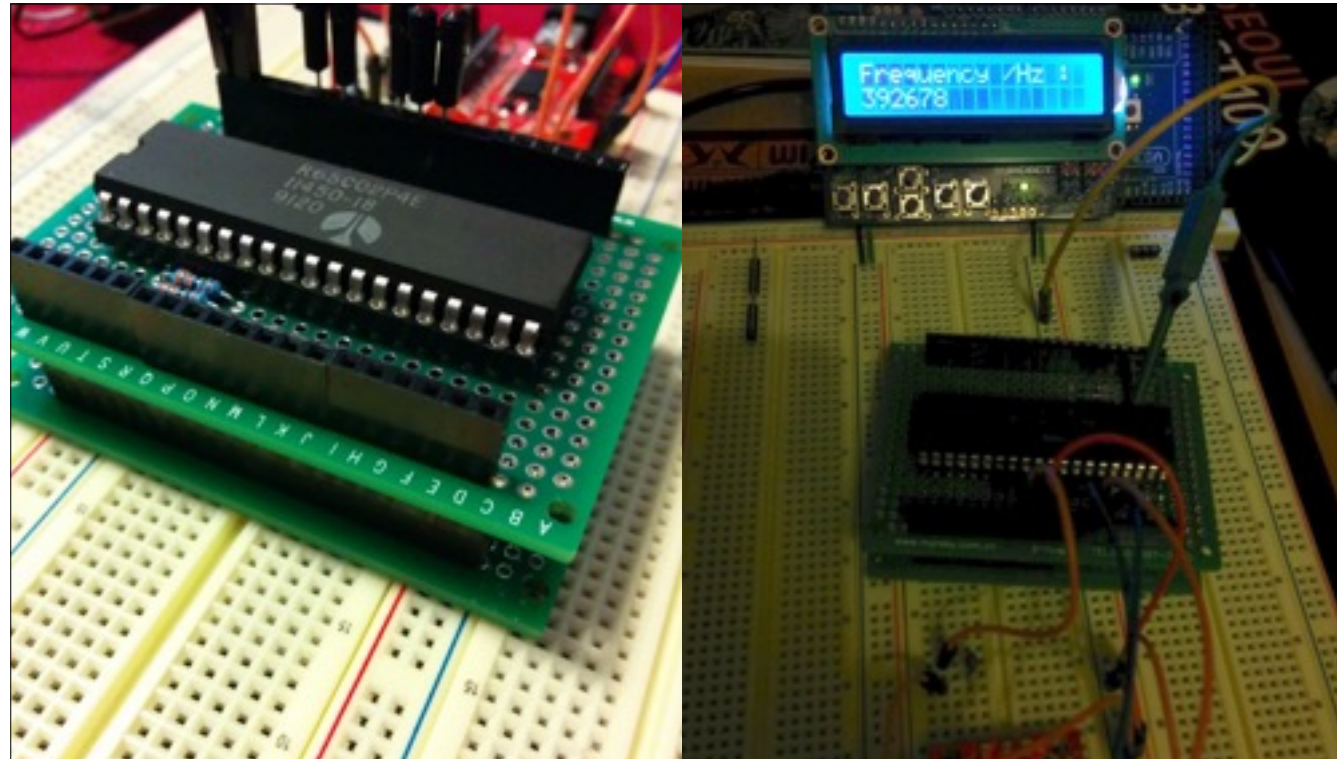
For the next step in my Apple 1 replica project I decided I wanted to replace the Arduino Mega board with a single Atmel 1284p with the goal of producing a two chip solution

I had been stockpiling parts for this phase of the project for a while, so I sat down to lay out the board based on a small 5×7 cm perfboard, using graph paper, just like we used to do with sprite designs in the 80's.



I had always intended that the design would be on two boards, that could be developed and tested independently, that would clip together like an arduino shield.

The trickiest piece was fitting the crystal and load capacitors into the design without disrupting too many of the other traces.



And this is the final product.

On the left are the two boards stacked together.

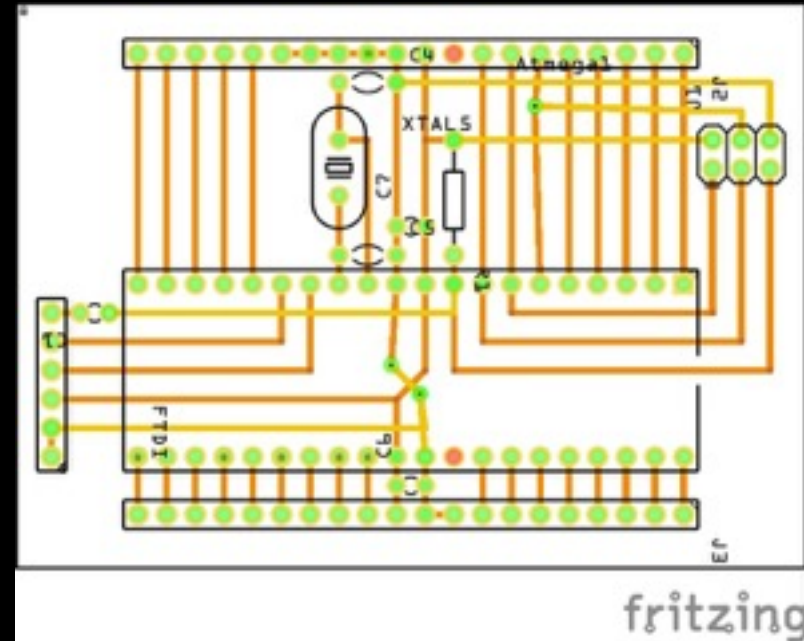
On this right you can see the frequency counter attached to the clock, we're running at roughly 400 kilohertz, that's about 1/3rd of the speed of the apple 1

[click]

This is a video of my terminal screen with the computer loaded with David Schmenk's 30th birthday demo for the Apple 1.

// FRIZZING

MAKING A REAL PRINTED CIRCUIT BOARD



I'm smitten with the atmega 1284p.

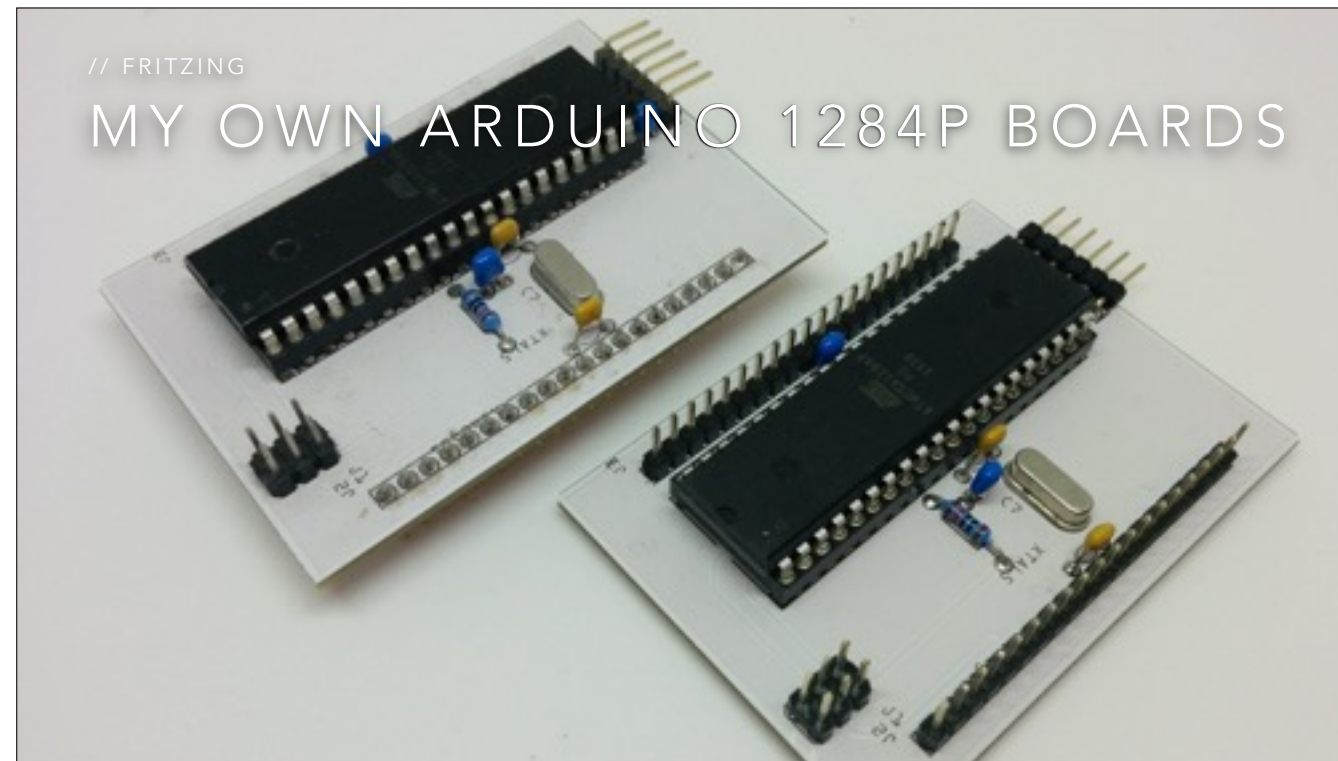
It feels like the right compromise between the pin starved 328 and the unfriendly 2540 series.

The 1284p supports more SRAM than either of its counterparts and ships in a package large enough that you get a full 24 pins of IO.

Now that I had a prototype working, I decided to try my hand at laying out a real PCB.

There are two popular PCB cad software programs available, Eagle Cad, and Fritzing. Eagle is more well known, but isn't very friendly for beginners.

The picture above is one of several designs I tried in Fritzing.



And here are the two PCB boards that I got back, both work.

Looking at these projects, they were fun, and challenging, and I learnt new skills and new tools, but ultimately both projects have sat on my shelf for over a year now. Neither of them are useful computers in the way that this computer here in front of my IS useful.

But I don't think that this matters, I learnt a huge amount just by doing; circuit design, spi, hardware debugging. I built a frequency Counter and learnt how to lay out a PCB and practiced my soldering.

I also learnt more about how computers work, at a fundamental level

Electronics is a hobby, not my job.

It's something I do for fun, when my job is not fun.

I don't get paid to work on my hobbies, but that also means that I won't let anyone down if they do not work.

I WANT YOU TO FEEL SAFE TO
EXPERIMENT.

I WANT YOU TO EXPLORE HOW
SOMETHING WORKS BY TRYING TO BUILD
IT YOURSELF.

DO NOT STOP YOURSELF BECAUSE YOU
ARE AFRAID THAT IT MAY NOT BE
PERFECT.

And this is the message I want you to take away from my talk.

I want you to feel safe to experiment,

to explore how something works by trying to build it yourself

and most of all,

please, do not stop yourself building a project because you are afraid that it may not be perfect.

Thank you.

THANK YOU

[HTTPS://DAVE.CHENEY.NET/PROJECTS/AVR11](https://dave.cheney.net/projects/avr11)

[HTTPS://DAVE.CHENEY.NET/2014/12/26/MAKE-YOUR-OWN-APPLE-1-REPLICA](https://dave.cheney.net/2014/12/26/make-your-own-apple-1-replica)

[HTTPS://TALKS.GODOC.ORG/GITHUB.COM/DAVECHENEY/GOSYD/PDP11A.SLIDE](https://talks.godoc.org/github.com/davecheney/gosyd/pdp11a.slide)