6502 Homebrew Design

A fun retro-ish project using some current tech to build a relatively simple computer from chips.

The components I'm looking at combining:

CPU

The latest variant on a 6502, keeping close to the original instruction set and electrical interface, the W65C02S:
https://www.jameco.com/z/W65C02S6TPG-14-Western-Design-Center-MPU-8-Bit-14MHz-65KB-Memory-40-Pin-PDIP_2143638.html This is capable of at least 14MHz, but I will only need 12.5875MHz (half the clock speed of VGA 640x480p 60Hz video). So 14 times the speed of the original Apple][(weee!) and with an easy-to-understand set of machine instructions and easy/fun computer design environment, not to mention a big homebrew community such as at 6502.org.

CLOCK

A 25.175MHz clock generator:
https://www.digikey.com/en/products/detail/ecs-inc/ECS-100A-251-7/79229?s=N4IgTCBcDaIKIGEDKBaAjABgwQRWArGgHQDsIAugL5A This matches the VGA 640x480p 60Hz pixel frequency.

RAM

Some fast static ram, to keep the design simple. Ideally we would have a 35ns or faster 1 megabyte memory bank, so we could interleave video reads and CPU r/w to the same RAM. Unfortunately, I don't see any cheap, small set of chips that are available for this. We could get a large set of cheap, 32kx8 15ns chips:
https://www.digikey.com/en/products/detail/renesas-electronics-america-inc/71256SA15TPG/2011041 We would need 32 of these (yuck!). There are expensive 512kx8 25ns parts ($70, yuck!), and a bank of two of these would work great for the ideal video/CPU interleaving design, but grayscale only. A bank of six would be needed for RGB, to get enough bandwidth. So the size is overkill: we only need a bit under 140k per chip instead of 512k. There are cheaper 55ns 512kx8 parts:
https://www.digikey.com/en/products/detail/alliance-memory-inc/AS6C4008-55PIN/4499026
With these, since they are slower, only video could be accessing a bank of six, with no access for the CPU. Double buffering with 12 total ram chips is possible, but not ideal.
Here are some interleaving possibilities for full 8-bit-per-channel RGB:

- 1 RAM chip, 7 11ns cycles (1 CPU, 6 video)
- 2 RAM chips, 4 20ns cycles (1 CPU, 3 video) (can support 2 CPUs, hmm…)
- 6 RAM chips, 2 40ns cycles (1 CPU, 1 video) (can support 6 CPUs)

For 8 bits per pixel (color map lookup or grayscale):

- 1 RAM chip: 3 25ns cycles (1 CPU, 2 video)
- 2 RAM chips: 2 40ns cycles (1 CPU, 1 video) (can support 2 CPUs)

There are 1 megabyte 10ns SRAMs that are cheap, but they are 3.6v and TSOP-44 mounting (yuck!). I found cheap 512k 5v parts in various surface mountings (slightly less yucky): https://www.mouser.com/c/semiconductors/memory-ics/sram/?access%20time=10%20ns~~25%20ns&interface%20type=Parallel&memory%20size=1%20Mbit~~4%20Mbit&organization=512%20k%20x%208&supply%20voltage%20-%20max=5.25%20V~~5.5%20V&instock=y&normallystocked=y&rp=semiconductors%2Fmemory-ics%2Fsram%7C~Access%20Time%7C~Supply%20Voltage%20-%20Max%7C~Memory%20Size There are adapters to DIP mounting out there, and interestingly 6502 homebrewers are a big audience for them! The 2 and 6 RAM chip variants are both possible with these parts (1 and 3 megabytes total, respectively). From Digikey here is a cheap 512kx8 10ns 5v part: https://www.digikey.com/en/products/detail/infineon-technologies/CY7C1049G-10VXI/5247556 This is in a 36-BSOJ (aka 36-SOJ) package, and here is an example adapter for that: https://www.proto-advantage.com/store/product_info.php?products_id=2200264 Note that the adapter costs as much as the memory chip! [Note: this is what I went for, so as to have plenty of margin in the timings for the initial prototyping. Later on a cheaper, more convenient chip could be tried once everything else is working.]

VIDEO WITH CONSUMER INTERFACES

For video output we could go with VGA analog or DVI/HDMI digital [note: see later section a direct digital interfacing with TFT panels]. For the digital route, there is some 8-to-10 bit encoding logic needed, as well as sync counting and voltage setting. The encoding bit might be done with CPLD instead of 74XX series logic, partly for speed and partly to reduce chip counts. The clock frequencies post serialization are huge, 250MHz or some such, which sound not feasible in a breadboard setting. So VGA output it is, unfortunately.

So we need appropriate DACs, and there are cheap triple DACs specifically for this purpose: https://www.digikey.com/en/products/detail/rochester-electronics-llc/ADV7120KN80/12110228 This one is capable of up to 720p (under 80MHz pixel clock), just in case we want to stretch out from 640x480p 60Hz. Unfortunately it requires a bulk purchase of 15 of them on Digikey. Alternatively, a higher-speed model can be ordered one at a time, but has a weird surface mount: https://www.digikey.com/en/products/detail/analog-devices-inc/ADV7125BCPZ170/2606709

The VGA output socket, female, solder individual wires: https://www.digikey.com/en/products/detail/norcomp-inc/171-015-203L001/858126

We could also try generating NTSC ala the original Apple][, and here is a cheap little NTSC LCD monitor that could be fun: https://www.adafruit.com/product/946 Here is a larger, more expensive NTSC-capable display: https://www.adafruit.com/product/2261 We also have some old displays laying around. With the Apple][running with the stock 1MHz clock, I was able to generate NTSC-compatible signals using a couple of memory-mapped bit outputs and a few

resistors. Although the vertical resolution was fine/normal for NTSC, the horizontal resolution was something like 8 pixels. By storing bytes and streaming them through a shift register, we could get up to 1-bit 64 pixels across, and with 12.6875MHz clock we could manage 812 bits per row of video, so maybe even two bits per pixel.

SIMPLE STARTER COMBO

For a super-simple initial project, maybe just a CPU + 1 SRAM chip + oscillator, with some sort of super-minimal I/O/memory writing interface, like some DIP switches memory mapped to the first set of instruction memory, or a CPU-off memory-writing mode using DIP switches for address and data.

LCD CHARACTER DISPLAY

An example 40 character by 4 line LCD display costs $24.39 on Digikey. The characters can be randomly addressed and written with a new ASCII value. Eight custom bitmap characters can also be written and displayed. There are several variants on extended ASCII fonts that can be selected. Overall this would be extremely easy to use and would allow fairly nice development work. The advantage over all the other display options is that you get a decent sized ASCII display with easy coding to drive it. The downside is that you can't really customize/bootstrap the font generation, and the display internally is likely a (small) computer, which we want to develop ourselves (no cheating by using parts with CPUs in them!).

LED SEGMENT/MATRIX DISPLAY

A handy module would be a 2 hex digit display for 8 bits in, based on something like this 2-char alpha LED module: https://www.adafruit.com/product/2153#technical-details A driver circuit could be moderately simple, basically an EEPROM such as https://www.jameco.com/z/28C64A-15-Major-Brands-IC-28C64A-15-EEPROM-64K-Bit-CMOS_74827.html with segment on/off data indexed by the 4-bit input nibbles (so 14 segments times 16 nibble values can be stored in 32 bytes of EEPROM). Add some clocking to alternate between the two nibbles/output characters, and a transistor+resistor per segment, and that is pretty much it. The EEPROM can only sync 5ma on its output pins, not enough to directly drive LEDs. Here is a possible NPN transistor array with common emitters that can save space/number of components: https://www.digikey.com/en/products/detail/stmicroelectronics/ULN2803A/599591 And here is the sort of isolated resistor array that could help as well: https://www.digikey.com/en/products/detail/bourns-inc/4116R-1-331LF/1088636 Since we have 16 bits per character in the EEPROM, and 2x8=16 segment drivers, we might as well use 16-segment LED modules: https://www.mouser.com/ProductDetail/Lite-On/LTP-3862G?qs=gnaPJ2cis70NSLLDN2qJAw%3D%3D With the EEPROM+driver solution, we could even do ASCII display with 8x8 LED arrays: https://www.adafruit.com/product/1079 (or a cheaper one [Note: I went with a cheaper one from Adafruit.]). This is 8 bytes per character for 128 characters, so 1k storage (we have 8k

available in the EEPROM listed above). We could then add a mapping from a nibble to the ASCII for its hex digit, and from there display the ASCII char (kind of a nice modularity/factorization in that). With the 8x8 LEDs we might only need a single pair of transistor-array + resistor array, plus an 8-bit register (maybe we could skip the transistor array if the register can drive the LEDs). Here is a 16-LED constant-current driver chip: https://www.digikey.com/en/products/detail/rochester-electronics-llc/MAX6979ANG/12102650?s=N4IgTCBcDaILIEEAaA2AnAdjQgcgcRAF0BfIA Unfortunately it is serial input, so some extra chip(s)/logic to deal with that. One of these can be the driver for two 8x8 LED arrays (two characters). So we don't need the register+resistor-array+transistor-array. It would not max out the LED brightness at only 55ma instead of the full 100ma. There are higher output current 16-output chips, but not in DIPs, and also higher current 8-output DIP chips, such as https://www.digikey.com/en/products/detail/texas-instruments/TLC5916IN/1906409.

Let's work backwards from the 8x8 LED arrays. The 16-LED driver(s) connects to a pair of these arrays. It will drive one row at a time. We will scan the rows at say 1kHz. So we need a 1kHz clock and a row counter. Every time we move to a new row, we will need to load 16 bits serially into the driver from the EEPROM. The EEPROM will be addressed by the left char plus row, then the right char plus row. We will then use a 8-to1 selector to serialize, or use a shift register. Here is an 8-bit parallel-in, serial-out shift register: https://www.jameco.com/z/74LS165-Major-Brands-IC-74LS165-8-Bit-Parallel-Load-Shift-Register_46877.html

To modularize the design, the output block can be the pair of 8x8 LEDs, with inputs: row index, 8 bits of LED on/off data, and a clock to set. Internally the module will shift the 8 bits of on/off data into the driver. The input is expected to provide the right and left character's data in order. In more detail: turn off LED output, increment row, latch first byte, shift 8 times into driver, latch second byte, shift 8 times into driver, latch 16 bits in driver, turn LED output back on, wait until end of this 1ms interval, then on to next row. Timing-wise, we have 1000us per row. Suppose we use a 1MHz clock. Then we would do the follow per 1us "tick", counting from 0 to 1023 per row:

0: first cycle with LEDs off – NOP otherwise
1: increment row counter
2: read and latch right char on/off bits for this row, going from EEPROM to shift register.
3,4,5,6,7,8,9,10: shift bits into the driver
11: read and latch left char on/off bits for this row
12,13,14,15,16,17,18,19: shift bits into the driver
20: latch data into the driver
21: turn LEDs back on
22-1023: NOP

Note that we can turn LEDs on any time from tick 21 to 1023 in order to modulate output brightness. We could also loosen up the timing to keep the tick number decoding logic simpler. To reduce the number of EEPROMs, we could time multiplex the access to one, since we only read it 2/1024 ticks. Each two-char module could have its or ID in the range 0..511, and offset the ticks by 2 * ID.

Row counter (this is four bits, and has row loading which we don't need, so quite overkill): https://www.jameco.com/z/74LS163-Major-Brands-IC-74LS163-Synchronous-4-Bit-Binary-Counter_46842.html
Cheap 1MHz clock generator (overkill accuracy): https://www.digikey.com/en/products/detail/ecs-inc/ECS-100A-010/20495 This could be an input to the two-char modules, so as to share the clock generator. Or just keep each two-char module maximally self-contained and include this with it.

We could factor the module further into a simple bitmap memory plus driver per 2-char module, and have a module that maps from ASCII to writing the bitmap buffer in the display module. How about something like a cheap static RAM (e.g. https://www.digikey.com/en/products/detail/renesas-electronics-america-inc/71256SA15TPG/2011041) with the 8-bit parallel-in, serial out shift register, plus the driver and LED matrix. The tick map would be something like this:

0: first cycle with LEDs off – NOP otherwise
1: increment (or latch from external) row counter (or just use the external row count?)
2: read and latch right char on/off bits for this row, going from SRAM to shift register.
3,4,5,6,7,8,9,10: shift bits into the driver
11: read and latch left char on/off bits for this row
12,13,14,15,16,17,18,19: shift bits into the driver
20: latch data into the driver
21: turn LEDs back on
22-1023: NOP

Input interface example: 3-bit row, 1-bit char left/right, 8 bits on/off data, write clock, 5v and ground. In this case the output module would need to have its own row and tick counters, tick decoding, tick clock, SRAM buffer etc. The module going from ASCII input would read the EEPROM and write the display buffer.
Second example: LED overall matrix on/off bit, 3-bit row, single LED element on/off bit, driver shift clock line, driver latch line. In this case we are just letting the driver keep the on/off state for a row and otherwise keep the output module dumb. The module going from ASCII input would deal with all the timing and tick logic. The danger is that the LED module could be left in a state with LEDs always on for a row, which they are not rated for at higher current. With our driver maxed out at 55ma per LED element, that would be over double the max continuous rating. We can either run the driver at 25ma, or add some sort of safety logic to the output module, like auto matrix off after 1ms if the row is not changed.

Suppose we have say 3 bytes of data (16-bit address and 8-bit value) to display as hexadecimal on 3 of the dumb output modules (driver plus two LED matrices only). That is 24 wires of input. We can loop through one output row at a time (one every ms). We loop through all the bytes, shifting the row data into the respective drivers after lookup from the EEPROM, but without latching the driver data yet. Then we latch all the driver data and turn the LED matrices back on.

Similarly we could create a display of N bytes of memory in hexadecimal. We would have a start address and N 2-nibble output modules, so 2+N total. We could use the “video” interleaved memory access to scan the N memory bytes and populate the driver buffers, with the same sort of 1kHz row scanning.

Additionally, we could display the ASCII values, one 8x8 LED matrix per byte instead of two hexadecimal digits. Or a combo: scan the memory and output both the ASCII char and the hex.

Finally we could just have a bitmap display, scanning SRAM and loading the driver buffers directly from the 6502. A long 8 or 16 high by 64 wide display could be fun. We could even display grayscale, since we have that option to modulate the duty cycle, but we would need to reload the driver buffer repeatedly (up to 16 times for a pair of 8x8 LED blocks) to independently turn elements off at different times. We could use a scheme where the 6502 writes to a parallel-in, serial-out shift register, which moves data into the driver(s). Since the STA $XXXX op takes 4 cycles at 12.5875 MHz, and the video clock and shift registers are capable of 25.175 MHz, we could keep up with even the tightest possible inline 6502 output code. We would need to auto-trigger 8 shifts per 6502 write to $XXXX. To make this easier on the 6502 programmer working on non display driver stuff, we could make the display work happen in the background using hardware interrupts.

Alternatively, we could use the video half of the 6502 clock cycle to drive the LED arrays in hardware. Reload each row 256 times each ms to allow 256 gray levels. We can read 12.5875MB/s from a single SRAM, so we could in principle get the data for 49 8x8 blocks across (392 pixels wide). Maybe make this 320 pixels across for a 40 character wide display? Readable(ish) characters are possible with 4-wide pixels (1 blank, 3 for characters), allowing an 80 char wide display at the extreme.

```
 *  **   *  * * * *  **  *  * *  *      *       **  *  *
* * * * * * **  * * *   ***   * * *  ** **   ** * * * * * *
*** **  *   *   *** * *  *   *  **  * * * * * *  ** * * **
* * * * * * **  *** * * *** *   * * * * * * **    *  *  **
* * **   *  * * * * *  **  *  * *  **  ** **   ** **    * * *
```

With two 8x8 blocks high, we could have two lines of text with the large font, and almost three with the teensy 3x5 font (need 17 pixels high with space between lines). The duty cycle per row would be 1/16th, which is okay. To get more rows of pixels, we would need to buffer the rows or make the driver updates sparser (not reload 256 times). For example, we would store 8 bytes per 8x8 block and have some simple hardware logic per block-row do the driver reloads.

Some more detailed thoughts on using the 8x8 LED matrices. If we make a complete sweep through the 8 rows of pixels at 125Hz, we would spend 1ms on each row, divvying that 1ms into 256 driver reload cycles to get 8 bit grayscale. So each reload cycle would be 3.90625us. At 12.5875MHz we can read 49 bytes per reload cycle. An appropriate SRAM to handle this would need 49*8=392 bytes of memory with around 70ns or better access time. Some possibilities:

- Jameco 32Kx8 35ns $3.29 in quantity (could handle two rows of 49 8x8 LED modules)

- Digikey 32Kx8 15ns $3.08 in quantity (could handle four rows of 49 8x8 LED modules)

Some LED driver chips are available that offload the PWM per-LED dimming work, allowing 8 bits per LED element, for example this one at Digikey with 16 57ma outputs ($1.75 in quantity). These sort of drivers would be nice since they don't require repeated scanning of the 8-bit-per-pixel row (so 256x less reading of the display SRAM). These chips are complicated enough that I would want to play with them to get a proper feel for their capabilities and limitations. Maybe a set up with two 8x8 blocks of LEDs, and see if they can be output with 8-bit grayscale without noticeable flicker. A 2-block module could be interfaced using ground, 5v, 3-bit row number, 4 serial i/o lines, and maybe the output enable or a couple of other lines. A controller module could have one 32Kx8 SRAM that feeds 256 of these 2x8x8 pixel modules, forming a 256x128 display, for example, giving 42x16 characters with a 5x7 font.

I've done some more digging on the LED matrix modules with e.g. 16-pin ribbon cable connections. I finally found a good page at Adafruit where they describe the reverse engineered pinouts and protocol. Sounds like these modules are *very* similar to what I was coming up with, but the primitive version where you need to scan repeatedly to get the PWM/grayscale effect. They did not use the more advanced internally-PWMing driver chips. I wonder if the modules could be rewired with the better driver chips?

There also is a video on Adafruit indicating some of these sorts of modules have coarser-grained PWM/current control, so the per-pixel PWM can be relative to that, I assume. There is also a notion of adding some small pulses to add a couple more bits precision. There is also per-LED brightness calibration and nonuniformity correction based on this, and in the same intensity mapping logic they can add nonlinearities like gamma. Personally I think having a linear scale display would be great software-simplicity-wise, but I guess human vision is nonlinear. Having a 16.384MHz clock per LED element would allow 16-bit PWM in 1ms. Seems doable, even if it would be a custom IC design (or maybe a CPLD could support this?).

Cheapest per-pixel I've found: 128x64 2mm $59.96 on Amazon [Note: I went with this one, as it is convenient to have a single big, cheap(ish) module with the best-available 2mm dot pitch.]

LCD/TFT DIRECT INTERFACE VIDEO

Maybe a better module is a 320x240 LCD display with a resistive touch screen and multiple i/o options. Adafruit has this module at a good price, along with a connector and simple breakout board to make it easier to solder to. This can be driven from SRAM and not much glue logic (some counters and latches, maybe voltage shifters, not much else). This could manage 320 / 6 = 53 characters by 240 / 8 = 30 lines of 5x7 text font, 64 characters across for a 4x7 font, and 80 across for 3x7 (or 3x5 or whatever). 3 wide fonts get really dicey for some characters, but 4 wide can get every character pretty well, but asterisk and M/W are a bit wonky.

```
X X  X  X X  X  X  X  XX
 X   X  X XXXX   XX  XXXX
X X  X  X X  X  XXXX  XX
     X  X X  X   XX  XXXX
     X  X X  X  X  X  XX
     XXXX X  X
     X  X X  X
```

There are several vaguely similar LCD panels that are 640x480 (this is a small subset):

- Digikey: 3.5", $42.51
- Digikey: 2.4", $48.43
- Digikey: 10.4", $157.06 (only 6 bits per channel color)
- Digikey: 10.4", $164.99 (8 bits per channel, better docs on differential serial LVDS)
- Mouser: 10.4", $234.30

However, some of the interfacing is completely unclear (no real data sheets, e.g. what the heck kind of connector and signaling does it have?). The one from Digikey with good docs may be the best bet. So far the needed connectors I've found are:

- https://www.mouser.com/ProductDetail/Hirose-Connector/DF19-20S-1C?qs=eDUdFcBPps3B3Q3%252BrLePFQ%3D%3D
- https://www.mouser.com/ProductDetail/Hirose-Connector/DF19G-20S-1SD-GND?qs=eDUdFcBPps27xzrqqFyQvg%3D%3D
- Here is a 20 pin 1mm to 0.1in adapter breakout board from Adafruit.

With 640x480, we would get 60 lines of 106 characters each using a 5x7 font, which would be fantastic for coding.

NONVOLATILE STORAGE

Here is a cheap 8kx8 EEPROM from Digikey:
https://www.digikey.com/en/products/detail/microchip-technology/AT28C64B-15PU/1008529
Some pertinent facts about this:

- Can write at most around 100k times (I assume for a given page).
- Write time is 10ms max (that is *milliseconds*, so slow, 100 pages per second maybe?)
- Read time is 150ns, so slower than the SRAM by ~10x. This implies we should treat it like an external disk and read/write pages into/out of SRAM.
- Pages are 64 bytes, and there are 128 of these.
- Read mode: OE low, CE low, WE high, set address, grab data 150ns later.
- Page write: set OE, CE and WE all high, set page address to A6-A12 and hold fixed, then loop A0-A5 from 0 to 63 and for each, set address, bring CE and WE both low at the same time you set the data lines, hold CE and WE low for a bit over 100ns, but less than 150us, then both switch back to high and hold there at least 50ns. Once all 64 bytes are transferred, just keep holding WE high for at least 150us to initiate page write. The page write end can be determined by a couple of different polling schemes. The one that is not data dependent is to keep a fixed address, WE high, CE low, and cycle

low/high on OE. The i/o6 line will alternate low/high while the page write is in progress, and then stabilize to one or the other states.

- For bootstrapping, we need to get enough code to load one page of the EEPROM and jump to this loaded page. The 6502 jumps to the address read from $FFFC-D. We could write to SRAM from EEPROM using external logic before the 6502 reset takes place. Maybe write 64 bytes (one page) to $FFC0-$FFFF and have $FFFC-D jump to $FFC0. This is duplicative with the software though. Maybe start with the EEPROM memory mapped to $FFC0 with a slow start-up clock speed, and run directly from the EEPROM initially. The software can then copy data, revert the memory map to just hitting the SRAM, and bump up the clock to full speed. We will expect access times well under 40ns when clocked at full speed and shared with video out, versus 150ns access times on the EEPROM. So a 4x reduction in clock speed seems about right.

Initially we could just put the EEPROM at $E000 and run with a slower clock for a super-simple system.

To bootstrap the writing of the EEPROM, we could have a simple switch-based address+data+write button (more on this later).

Here is a nice hexadecimal keypad for input:
https://www.digikey.com/en/products/detail/grayhill-inc/88BB2-072/210423

```
; idea on how to copy data from EEPROM to SRAM
; $C000 is the writable page address latch (bit 7 is not used currently)
; $C001 is the writable byte-in-page address (bits 6-7 are ignored)
; put address into EEPROM address latches
        LDA #$12
        STA $C000
        LDX #$40
LOOP  STX $C001
; data will show up at $C002 after 150ns, so delay a bit
        NOP
        LDA $C002
        STA $0FC0,X  ; makes up for the +$40 bias in X
        INX
        BPL LOOP  ; falls through for X=$80
```

The above is 20 bytes of code.

Alternative uses an up counter that does not allow random access for the byte-in-page address:

```
        LDA #$12
        STA $C000  ; sets the page address latch
        LDA $C001  ; clears the byte-in-page counter to zero
        LDX #$40
```

```
LOOP  LDA $C002  ; enough cycles have passed that the EEPROM data is available
      STA $C003  ; increments the byte-in-page counter
      STA $0FC0,X
      INX
      BPL LOOP
```

This is 22 bytes of code. We could combine the byte-in-page increment with the LDA $C002 and eliminate the 3 bytes for the STA $C003, getting us down to 19 bytes of code. We could also combine clearing the byte-in-page counter with the STA $C000, eliminating another 3 bytes and getting us down to 16 bytes of code:

```
      LDA #$12
      STA $C000  ; sets the page address latch and clears the byte-in-page counter
      LDX #$40
LOOP  LDA $C002  ; enough cycles have passed that the EEPROM data is available
                 ; also increments the byte-in-page counter
      STA $0FC0,X
      INX
      BPL LOOP
```

A reset-time hardware loader would need to count from x=0 to 63, read page 0 byte x, write to $FFC0+x (so as to include the RESET vector $FFFC-d).

1. Set CPU pin BE low to put in high-impedance state
2. Set CPU pin RDY low to halt the CPU
3. Hold RESET pin low while copying from EEPROM to SRAM.
4. Clear the page address latch to zero, which will also reset the byte-in-page counter to zero.
5. Latch the byte-in-page counter (needs an extra latch chip for this).
6. Write the EEPROM byte to SRAM at $FFC0+byte-count-latch (bitwise OR).
7. Increment the byte-in-page counter.
8. Goto step 5 if bit 6 of the byte-in-page counter zero.
9. Set BE high, CPU RDY high, and then let RESET go high, triggering the RESET sequence.

Here is a nice, cheap counter chip with parallel outputs, count clock and reset: https://www.digikey.com/en/products/detail/texas-instruments/CD74HCT4040E/38593 It does not have tri-state outputs, but I don't think we need them? We will need to output these to the EEPROM byte-in-page address (not a problem with no tri-state) and the SRAM address lines (tri-state needed to avoid conflict with CPU or video use of the same address lines). But I think we will need a tri-state 16-bit latch for this anyway, so we are fine.

We could, for pure bootstrapping from scratch (no control/downloading from e.g. a Macbook), use the hex keypad and an SRAM and display to fill a page (64 bytes) of EEPROM memory and put at $0000 of the EEPROM address space. Then something like the bootstrap EEPROM-to-SRAM loader circuit could be used to write the page (basically initially

programming in hardware). Once one 6502-based machine is up and running reasonably well, this bootstrap logic can go away and the new machine can be programmed in software to do the future EEPROM programming work. Basically the EEPROMs become our removable permanent storage media.

Since we are treating the EEPROM as a storage drive (akin to the 140KB floppy drives on the Apple][), why not use flash memory chips? Here is one that is 256KB with freaking 70ns(!) access times, and costing only $3.36 in onesies quantities: https://www.digikey.com/en/products/detail/microchip-technology/SST39SF020A-70-4C-PHE/2297831 Here are some relevant facts about this chip:

- Available in PDIP packaging, making these super convenient for breadboarding.
- Organized as 64 4KB sectors.
- 100K write cycles lifetime, maybe per page? Probably not a problem unless this is used for virtual memory a la Linux or frequent saving of text editor state. What was the endurance of the Apple][floppy disk, anyway?
- Reading is simple, like an SRAM: put WE high, CE and OE low, put in an address and read the byte.
- For writing, you send a command sequence on the address and data lines.
- Programming a single byte takes at most 20us (take that, Apple][floppy disks!).
- I don't see anything like the EEPROM where you can fill a page and then initiate a write. I'm not clear what goes on under the hood to avoid repeated sector overwrites as you write every byte in a sector… After searching around for details and definitions, here is a relevant paragraph from https://ww1.microchip.com/downloads/en/Appnotes/S72005.pdf:

  > A flash memory sector can be programmed many times between erases, as long as each byte (word) within the sector is only programmed once. For example, an SST flash device with a 4 KByte sector can have all 4 KByte programmed byte-by-byte in one Program operation time or can have each byte programmed with its own Program operation until all 4,096 locations have been programmed. Each Erase operation plus the Program operations (1 to 4096 times of Byte-Program) after the erase, makes one endurance (Erase and Program) cycle.

  So, while writing one byte at a time with a fancy command sequence seems inefficient, it is not an issue for endurance.

For a really capacious, cheap ($3.38) flash chip, consider this 8MB part: https://www.digikey.com/en/products/detail/microchip-technology/SST26VF064BT-104I-SM/5032913 It has a serial interface, runs at 3.3v, and is available only in surface mount, so it is triply annoying. We *could* design the system to primarily use a 3.3v supply voltage, since the W65C02S will happily run at that, but only up to 8MHz instead of our desired 12.5875MHz.

So flash is indeed weird as I noted: a whole sector must be erased, in general, to rewrite even one bit within it. So: read the sector, erase, modify the copy, write back. Bottom line: it is

probably simplest to treat a sector as an atomic I/O unit. Fancier approaches might look like this: write only to erased pages, keep old/stale pages around, then do the least sector erasing and rewriting possible as needed to get rid of stale pages to allow further writing. A level of indirection is needed to keep track of the physical pages currently associated with the active logical pages. A bit map of erased (writable), stale and active physical pages would also be needed. Since the logical-to-physical page map and physical page status bit maps need to be kept in nonvolatile memory, some clever scheme will be needed to write updates to the maps rather than try to maintain the map state itself directly in flash. In other words, treat flash as append-only storage with as-little-as-possible garbage cleanup. All that complexity goes away if you just treat the 4KB sectors as the units for (re-)writing (read can be random 256-byte pages, no problem).

Back in 6502 land, a 4KB sector is 1/16th of the address space. Of course we can do paging tricks, but still, treating e.g. $2000-$2FFF as an atomic I/O unit is pretty coarse grained.

As a little thought experiment, again comparing to Apple][days, how long would it take to read or write most of the 6502 address space, say 48K, from/to flash? Suppose we implement the serialization hardware for SPI two-wire style communication with the flash chip, with a 25.175MHz per-bit clock (the chip can handle up to 40MHz per-bit clocking). Then reading all 48K would take around 16ms, versus maybe a second for really fast Spiradisc I/O, so around 50x faster on read. Worst case on write is rewriting, with a 4KB sector erase at 25ms and 16 256-byte page programs at 25ms per sector, so 50ms per sector and 600ms total to write all 48KB. So very similar to old floppy disks for write times with the super-fast Spiradisk methods.

As a very expensive but cool alternative to EEPROM/flash, there are NVSRAMs, which sound amazing. For example: https://www.digikey.com/en/products/detail/analog-devices-inc-maxim-integrated/DS1250Y-70IND/1196853 The idea is that the CMOS SRAM has low enough data retention power requirements that it can be paired with a small lithium battery that can hold the data for 10 years with the power off (I assume it recharges when powered up?). This sort of device makes bootstrapping almost trivial! No special pre-loading from nonvolatile storage to SRAM before 6505 RESET starts – this *is* the system memory. Load the NVSRAM with some initial code without the 6502 active, just once, and after that the 6502 can reprogram its own nonvolatile storage.

To go even farther towards a disk drive, there are micro SD cards with storage capacities that are orders of magnitude overkill for a 6502 system, for example, 32GB for $3.50. Evidently an open/free(ish?) interface is available based on the SPI style serial communication. Here is a nice Adafruit tutorial and breakout board. These little disks would be safer/more convenient to pass around than chips, and potentially readable on a wide variety of other hardware. However, making a no-CPU pre-reset reader circuit seems daunting/impractical, so maybe just read the SD i/o logic from a chip and use the SD card from then on. Something to look into.

KEYPAD/KEYBOARD

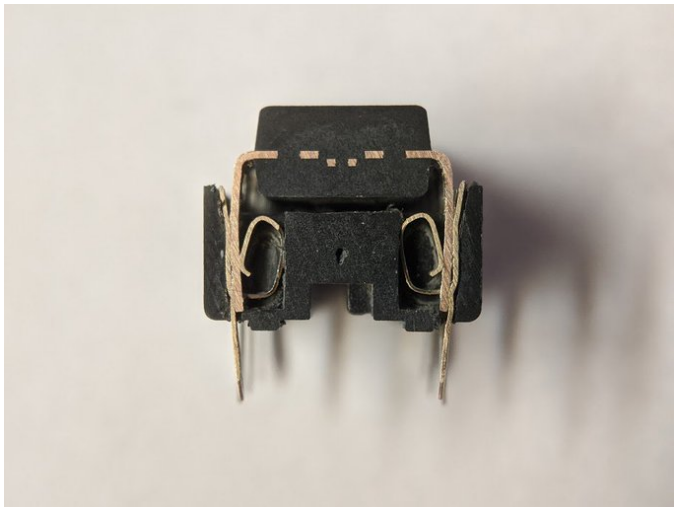
The 4x4 hexadecimal keypad was mentioned earlier, and would be good for the very initial bootstrapping design. It is simple to scan, just a 4x4 matrix.

I'd like to move quickly to a full keyboard. After a lot of digging I don't see any newly-manufactured keyboards that are something simpler than USB to talk to, except maybe the Tipro ones that have their internal I2C bus for interconnecting modules. That thread lead to this really nice post on wiring your own keyboard matrix without a PCB: https://geekhack.org/index.php?PHPSESSID=von3nnn3424qekshpa35uua7jidv8d9s&topic=87689.0;topicseen Another possibility is to take a cheap USB wired keyboard (I have two of these I won't need otherwise) and remove the controller circuit to expose access to the underlying keyswitch matrix wiring. I have not yet found any evidence online that someone has successfully done this. For the Logitech K120 specifically, IFixit has a step-by-step teardown from the back, and a keycap removal page. Evidently their business model is to sell tools like this spudger.

And Aha! With some more digging I found a page where someone hacks the K120 matrix circuit boards: https://www.instructables.com/Hacking-a-USB-Keyboard/ An interesting ingredient to make this easier is electrically conductive epoxy, but it is crazy expensive (~100 bucks for half an ounce): https://www.digikey.com/en/products/detail/mg-chemicals/8331D-14G/13170710 There is a heat-cured alternative that is a bit cheaper, but a hassle to cure (30 min at 100C). I'm not sure how the plastic circuit sheets would react to soldering. Maybe there are some small clips that would work? Or just cannibalize the connectors from the original interface circuit board? Here is a page where someone documents the original connection, essentially just clamping to some pads on the original interface board using a screwed-down metal bar, and their custom PCB to mimic this: https://www.hackster.io/frank-adams/teensy-controls-a-logitech-keyboard-fujitsu-ergotrac-a0e308 I'm thinking a simple solution it to just use solid hookup wire, strip a bit off the end, and form a little j or μ shape and crimp, glue and clamp. Maybe *very* carefully try soldering…

Or just pull the pins out of a cheap DIP socket, the kind that are kind of a spring clip, and use them off label to connect to the plastic PCB contacts. Here is a nice cross section-picture

someone made of these spring clips, in action for their originally-intended use:



Since the keypress can be very brief and we want to catch it right away, some sort of active scanning will be needed. We could code this up on the 6502 itself, or have an auxiliary processor to offload that work. Initially maybe do it on the 6502, then make a little board to do it, buffer the results with time codes, and allow the 6502 to collect the keypresses at its leisure. 1ms ticks on the time codes should be good enough. The events buffered are (timecode,keycode,up-down-bit). It is up to the host 6502 to turn this into ASCII keypress events for higher-level applications. Applications like games probably want the lower-level event details.

A really simple keyboard matrix scanner would have a counter chip repeatedly go from 0 to 127, with the lower 4 bits being the column of the keyboard matrix and the upper three bits interpreted as the row. Use a sense circuit to read the up/down state bit. Put this into a shift register. Every 8 reads put the accumulated byte into an SRAM and increment an address counter. Allow the SRAM to be read by the main host 6502 using two 8-bit address registers and a byte read latch. Also give the 6502 access to the current matrix scan write address. The address encodes the time and the key ids. A 32KB SRAM can hold 2048ms of scan data if we rescan the keyboard every 1ms. The 6502 would need to interpret/access the keyboard SRAM data at least once every two seconds or lose key up/down events. The scanning could use the 12.5875MHz clock divided by 64 or 128 to get close to the desired 1ms scan times.

FAST PROGRAMMABLE LOGIC

For some of the glue logic, I've read that CPLDs are way faster than most discrete, fixed logic blocks, and can save a lot on chip counts. Unfortunately it seems most 5V CPLDs are obsolete. I found a nice discussion thread on the various CPLD families, programming issues, and mainly the 5V tolerance and level shifting possibilities. Evidently the current Lattice ispMACH parts are tolerant of 5V on input, and the 3.3v outputs can adequately drive some 5V TTL families (bipolar, but not CMOS).

BOOTSTRAPPING THE DESIGN

To begin bootstrapping the system without using any existing CPUs (that would be cheating!), we could “program”, in hardware, an NVSRAM/EEPROM/FLASH manual data entry system. Using the hexadecimal keypad and 8x8 LED arrays (initially showing bytes in binary), we could enter address and data and press a “write” button. There could be two data displays, the current read data and the latched data to be written. This would be good enough to put in simple bootstrapping code for the 6502 and ASCII-to-font bitmap data. Once those are in place, we could eventually move to having a bitmap/grayscale video-like display on a set of 8x8 LED matrices, and full-size ASCII keyboard input. For the SST39SF020A-70-4C-PHE 256KB flash memory, it can write one byte at a time in a fairly simple way (no queuing up page writes), without stressing endurance. This might be best for initial bootstrapping, as the other storage options require writing pages of at least 64 bytes as a unit instead of byte-by-byte. The other exception is the NVSRAM with the lithium battery: with that we can write/rewrite bytes at will, but pay through the nose to do so. Thinking about this a bit more, I like the NVSRAM approach: it is for initial bootstrapping only, so cost is not really a factor (it is not like we will be making a bazillion of these). If the NVSRAM were not available prepackaged, we could just hook a regular SRAM up to 5v with battery backup ourselves (the NVSRAMs are not that magical, just convenient).

ACTUAL PARTS ORDERS

- Amazon: 128x64 2mm LED module Sold by: AZERONE, $59.96, Condition: New [This is coming straight from China via SFExpress, a shipping company headquartered in Shenzhen.]
- Rochester Electronics (rocelec.com, UK!): non-obsolete, fast NVRAM, CY14E116L-ZS25XI, 5 * $57.95 = $289.75 (this is a 5v, 2Mx8 25ns part, pretty amazing)
- Digikey:
    - PA0217, PA0217-ND, TSOP-44 II TO DIP-44 SMT ADAPTER, 10 * 8.39000 = $83.90
    - ECS-100A-251.7, X130-ND, XTAL OSC XO 25.1750MHZ TTL TH, 10 * 3.00900 = $30.09
    - 88BB2-072, GH5016-ND, SWITCH KEYPAD 16 KEY 0.01A 24V, 1 * 61.55000 = $61.55
    - SST39SF020A-70-4C-PHE, SST39SF020A-70-4C-PHE-ND, IC FLASH 2MBIT PARALLEL 32DIP, 10 * 3.36000 = $33.60
    - TLC5916IN, 296-24383-5-ND, IC LED DRIVER LINEAR 120MA 16DIP, 10 * 1.40300 = $14.03
    - CY7C1049G-10VXI, 448-CY7C1049G-10VXI-ND, IC SRAM 4MBIT PARALLEL 36SOJ, 10 * 6.75000 = $67.50
    - CD74HCT4040E, 296-2118-5-ND, IC BINARY COUNTER 12-BIT 16DIP, 10 * 0.70000 = $7.00
    - CD74HCT4040E, 296-2118-5-ND, IC BINARY COUNTER 12-BIT 16DIP, 10 * 0.70000 = $7.00

    - SN74AHCT573N, 296-4759-5-ND, IC OCT TRANSP D-TYP LATCH 20-DIP, 20 * 0.68700 = $13.74
    - SN74F521N, 296-33912-5-ND, IC COMPARATOR IDENTITY 8B, 20DIP, 10 * 0.84000 = $8.40
- Jameco:
    - 2143638 W65C02S6TPG-14 IC,W65C02S6TPG-14,PDIP-40, MPU,8-BIT,14MHz,65KB MEM 4.00 9.95 0.00 9.95 39.80 5/31/2023 0.00
    - 46877 74LS165 IC,74LS165N,DIP-16,8-BIT PARALLEL LOAD SHIFT REGISTER 10.00 0.79 0.00 0.79 7.90 5/31/2023 0.00
    - 2154863 9313-1-R-100 WIRE,22AWG,SOLID,BROWN,100' HOOK-UP WIRE,UL VW-1 1.00 9.45 0.00 9.45 9.45 5/31/2023 0.00
    - 2183752 JMS9313-01D WIRE,HOOK-UP KIT,W/ BOX,22AWG 100',BLK,BLUE,GRN,RED,WHT,YLW 1.00 44.95 0.00 44.95 44.95 5/31/2023 0.00
    - 2154898 9313-8-R-100 WIRE,22AWG,SOLID,GRAY,100' HOOK-UP WIRE,UL VW-1 1.00 8.95 0.00 8.95 8.95 5/31/2023 0.00
    - 2154880 9313-7-100 WIRE,22AWG,SOLID,VIOLET,100' HOOK-UP WIRE,UL VW-1 1.00 9.45 0.00 9.45 9.45 5/31/2023 0.00
    - 2154871 9313-3-100 WIRE,22AWG,SOLID,ORANGE,100' HOOK-UP WIRE,UL VW-1 1.00 9.95 0.00 9.95 9.95 5/31/2023 0.00
    - 36768 9313-LB WIRE,22AWG,SOLID,BLUE,100' LIGHT BLUE (SOLD IN 100'RLS) 1.00 9.95 0.00 9.95 9.95 5/31/2023 0.00
    - 20812 WBU-208-R BREADBOARD,7.25"x7.5",3220pts.RED,BLK,GRN/YLW BDG POST(JE27) 5.00 35.95 0.00 35.95 179.75 5/31/2023 0.00
    - 2318095 4-103185-0 HEADER,STRAIGHT,MALE,1RW,40PIN,.1"CTR,.025"PSTx.23",GOLD 10.00 2.25 0.00 2.25 22.50 5/31/2023 0.00
- Adafruit:
    - 10 x Miniature 8x8 Yellow-Green LED Matrix[ID:861] = $35.60
    - 5 x 3.5mm (1/8") Stereo Audio Jack Terminal Block[ID:2791] = $12.50
- Proto Advantage:
    - SOIC-36 to DIP-36 SMT Adapter (1.27 mm pitch, 10.16 mm body) 10 * xx = $xx
- Jameco:
    - ZIPWIRE,MALE-MALE,20CM LENGTH,40 WIRES,10 COLORS,28 AWG(40) 2260738 10 $4.49 $44.90
    - Tariff Surcharge: $8.53
    - RES,CR-RC10,AXIAL LEADS,50 OHM 10 WATT,5%,CEMENT,WIREWOUND 2246629 10 $0.529 $5.29

    - IC,LM556N,DIP-14, DUAL TIMER (NE556N,UA556PC, LM556CN) 24328 10 $0.49 $4.90
- Amazon:
    - Electronics-Salon 24/20-pin ATX DC Power Supply Breakout Board Module, 2 * $15.99
- Newegg:
    - Seasonic FOCUS GM-650, 650W 80+ Gold, Semi-Modular, Fits All ATX Systems, 2 * $119.99
- Jameco and Digikey: I also ordered a bunch of tools and supplies that will help with this project, but are not specific to it, like a big collection of resistors of various values, pliers and cutters, ZIF sockets, etc.

Parts/tools/etc we may need to order:

- ESD safe small parts cabinets: https://www.jensentools.com/product/429BE9037-551-4ESD
- Only if needed: faster, four-channel oscilloscope (keysight MSOX3054G or Rigol MSO7054).
- [DONE] SN74CBT3257DBQR quad 2:1 mux/demux (at least 10, probably 25 in case we use them in more places).
- [DONE] A variety of ceramic capacitors from 1pf up.
- [Maybe get large ceramics instead?] A variety of metal film capacitors for larger capacitances.
- [DONE] 2N2222 and 2N2907 transistors
- [Note: found a stash of large ceramic caps – need to sort/organize these] Smaller ~30uF capacitors – evidently ceramic capacitors of this size exist: https://www.digikey.com/en/products/detail/tdk-corporation/FG26X5R1E336MRT06/5803046
- More 8x8 LED matrix blocks
- [DONE – F parts] 74LS240 inverting octal buffer with tri-state outputs (maybe some F parts too)
- [DONE – F parts] 74xx08 (LS and F parts)
- {DONE – F parts] 74xx373
- (any others I used and am low on or had to work around if missing)
- {DONE – SN74CBT3245CDWR parts] Maybe use FET switches to connect/disconnect data/address lines, something like SN74CBTD3861PWR. These have sub-nanosecond propagation delays once connected, and less than 10ns to connect/disconnect. You can have a bus-like set of lines that N things can connect to. For example, what is the CPU data writing to/reading from? Which address is going to a particular SRAM? Faster, 24 bits switching (2 independent 12-bit circuits): SN74CBT16211ADGVR Fastest yet, and cheap, 8-bit switch: SN74CBT3245CDWR $0.27 in quantity, <5.3ns enable/disable times.
- [DONE] Hakko FX888D-29BY/P soldering station from Jameco, and some alternate tips
- [DONE] Hot air reflow station and accessories from Adafruit

- [DONE] In addition to the F parts above, I ordered small 33uF ceramic capacitors from Digikey.
- [DONE] After a rough time soldering the surface mount chips with a regular soldering iron, I've ordered stencils and solder paste from Proto Advantage. These should work well with the hot air rework station.

SOLDERING TOOLS/TUTORIALS

Since this project is getting into surface mount stuff, it makes sense to look into tutorials and possible tool upgrades. I have my old Weller soldering station and the nice temp controlled hot air gun with various tips, but who knows if those are suitable.

A reddit post with some helpful-looking tips/leads:
https://www.reddit.com/r/AskElectronics/comments/5mw86x/soldering_iron_recommendation_for_smd_work/

Seems like Hakko is a good brand. Adafruit has their recommended one. Also Adafruit has a hot air rework station and accessories they recommend. I ordered what I could from Adafruit, and what they were out of stock on I ordered from Jameco and Digikey. Now on to watching some tutorial videos…

FORUMS AND POSTINGS

2023-05-31: I joined 6502.org with the username SpiradiscGuy.

GOING DEEPER: HOMEBREW CPU DESIGN

A super deep homebrew would replace the 6502 with an orthogonal RISCish 8-bit instruction set, with one accumulator register, one program counter, 256 bytes address space and uniform opcode+data (16 bits) per instruction:

- 3 bits: Move, add, subtract, rotate, or, and, xor, count matching bits
- 2 bits: control bits that may vary depending on the first three bits. Things like:
    - negate/compliment the result, increment/decrement the result,
    - rotate off the end or wrap around,
    - count zeros or ones, count all or just how many are leading
- 1 bit: Primary register is accumulator or program counter
- 2 bits: source/target code
    - Source is primary register, target is memory with data address
    - Source is primary register, data is immediate, target is other register
    - Source is memory with data address, target is primary register
    - Source is immediate data value, target is primary register

Note that all the instructions with the target register being the program counter are jumps of various flavors. For example, adding/subtracting an immediate value to the program counter is a relative jump, whereas moving an immediate value to it is an absolute jump.

We could also make the address 16 bits, 32 bits, whatever, while keeping the opcode 8 bits, but then the registers would need to be the same number of bits so they can hold a memory address.

For floating point, it would be nice to have log and exp base 2 with 1.0 being in the middle, then mul/div/powers become add/sub. Maybe also have integer mul/div/mod, since that is exact for e.g. indexing work vs. the log/exp thing.

It would also make sense to increase the opcode to 16 bits. That would allow say 8 bits of register index (vs. the one bit we have now), and double the number of ops.

We could use a 6502-based environment to help with the development of the new CPU design. Possible stages: simulate, develop assembler, build hardware prototype in stages.

How about using a fast, big SRAM to implement 8-bit add, subtract, multiply, divide, count bits, whatever, then the op code just becomes more address lines to the SRAM. With the 10ns 512kx8 SRAM we are getting, the 19 address lines could be used for two 8-bit values, a carry/borrow/overflow bit, plus two bits representing four opcodes. More opcodes could be added by adding more SRAMs or getting a bigger one. This could be implemented by adding the propagation delay of an N-to-1 multiplexer to select the right output. Alternatively, using a N-to-2^N demultiplexer with outputs going to the respective SRAM output enable lines could be faster. The demultiplexer propagation delay happens in parallel to the SRAM read delay, but the output enable may add to the SRAM read time. Bitwise logic functions can be performed using the respective 2-input logic gates. For example, a 7F00 NAND gate has a 2.5ns propagation delay. Or just simplify the design and use SRAM for all ops.

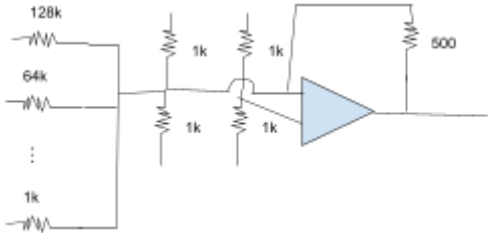
Here is the biggest appropriate SRAM I could find, a 2m x 16 bit 10ns chip. With this, we can have two 8-bit inputs plus carry/borrow/overflow in (17 bits of address), plus 4 bits of opcode (for a total of 21 address bits), and sufficient bits for output to include the full multiply/modulo result or the carry/borrow/overflow and zero/nonzero/bitcount results. So:
+,-,*,/,%,&,|,^,&&,||,<<,>>,bitcount,whatever.

If the instruction opcode is 16 bits, with 4 bits of SRAM/ALU opcode, then the remaining 12 bits could be divided into two 6-bit register indices, where each register is 8 bits (64x8 register space). Or we could have 4kx8 register space and have an instruction to set the target/accumulation register. The output of the SRAM ALU is up to 16 bits, so maybe the two bytes starting at the target register would be written, or just the lower byte and the upper byte goes to a special status register. We could optionally increment the target register index on write, to make N-byte operations easier. In addition, there would need to be load/store from/to main memory, either immediate values located at the next program counter address, or at an

address specified by a number of contiguous register bytes. We would likely need to make the program counter a special register with load/store atomic operations from a contiguous set of register bytes, so as to avoid a read delay to access it when reading the next instruction/data value from main memory.

SOUND GENERATION

Basic audio generation can be as simple as connecting a homebrew resistor-network digital-to-analog converter to some memory-mapped latches (mono 8-bit to 16-bit-per-channel stereo would take one to four latches, respectively). At audio sampling rates, like 48kHz, a 12.6875MHz 6502 would have 264 clock cycles per audio sample output. This type of design would allow (actually force) fully CPU-computed audio. A slight variant is to use chip-based DACs, which becomes especially important with higher bit counts.



Here is a two-channel, fast 16-bit DAC for around $21 in onesies quantities.

Another approach is to have analog audio synthesis modules that can be reconfigured and controlled by the CPU. A simple example of this is the simple piano-like note generation module that I was prototyping, that turns a square wave into a triangle wave into an approximate sine wave. With an amplitude attack/decay circuit to modulate this, and replicating this module several times, a nice little polyphonic synthesizer would result that offloads a lot of work from the CPU. To get a richer experience, general waveforms, plus white noise, with filter banks, would give a generous range of synthesis capabilities with low CPU cost.

A third approach is to use the first, fully CPU-compute approach, but have a second CPU module offloading that work. The most general way to talk to such a module is to send it code to execute, for example, a blob of machine code, although more specialized communication protocols are of course possible.

A fourth approach is to create a digital synthesis module with waveform memory, random number generators, multiply-add modules and the like, essentially making a specialized programmable audio processing unit. I'm not sure how much of a win this is over just plopping down another general CPU, except maybe in performance metrics like the number of voices it can support (but we can always just plop down lots of CPU modules).

A fifth approach is to create pulse-width modulation style audio output, either directly on the CPU or using a simple external counter circuit. The PWM output will need to be band-pass filtered to the 20Hz-20kHz audio range. A 16-bit counter would effectively be equivalent to a

16-bit DAC. Unfortunately a pure PWM scheme with a 16 bit counter and 48kHz sampling rate would need a bit of 3gHz for the counter clock.

A sixth approach is to use a bit of a hybrid design. Use the 8-bit resistor-network DAC, but send its output to an op amp integrator, so the DAC output is interpreted as a slope of the signal. Keep a model of the integrator error and offset the DAC inputs to correct for these over time, essentially dithering in derivative space. This may result in some drift due to an imperfect model prediction. Maybe use a second 8-bit DAC and voltage comparator (or low bit depth ADC) to do drift correction, and maybe feed this back to help improve the model calibration on the fly.

Basic iterations:

- delta0_1 = (V1_desired - running_error) - V0_modeled, rounded to the 8 most significant bits
- V1_modeled = V0_modeled + delta0_1
- running_error = (1 - alpha) running_error + alpha * (V1_desired - V1_modeled)
- delta1_2 = (V2_desired - running_error) - V1_modeled, rounded to the 8 most significant bits
- V2_modeled = V1_modeled + delta1_2
- running_error = (1 - alpha) running_error + alpha * (V2_desired - V2_modeled)
- …

This sort of scheme does not require an extreme clock rate. Even running at 48kHz (no oversampling) the scheme would correct nicely for lower-frequency signals, on average. To be accurate at higher frequencies, maybe run at a 16x clock rate, so 16*48kHz = 768kHz. The desired 48kHz samples could be linearly interpolated to get the Vn_desired sequence. The output would be band-pass filtered to 20Hz-20kHz.

TIMING AND MEMORY ACCESS

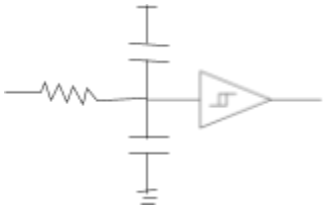
We will be running a 25.175MHz oscillator to match the 640x480 60Hz pixel clock rate, and dividing this in half to run the W65C02S CPU at 12.5875MHz. We will alternate video and CPU access to SRAM, where video data will be read when the CPU clock is low, and CPU read/write will occur when the CPU clock is high (with some minor quibbles about the access going 10ns or so past the end of these intervals).

Dividing the 25.175MHz clock by powers of two will be performed by a fast 4-bit counter. The propagation delay from the video clock low-to-high transition to the counter output changes is 2.7ns to 10ns, so the video clock will be a bit early in its transitions compared to the CPU clock. This is likely a good thing that we will take advantage of.

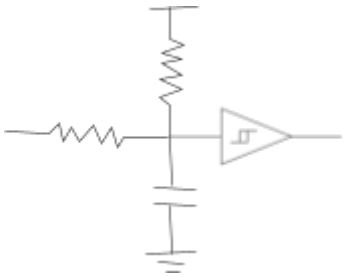
We need to pick whether the video address or the CPU address is input to the SRAM address. A pair of simple but fast 8:4 multiplexers will do this with a propagation delay of 1.2 to 11ns. The CPU clock will feed into the A-low/B-high selection input line, with the A inputs hooked to the video address, and the B inputs to the CPU address, and the Y outputs connected to the SRAM address. Due to timing requirements of the SRAM, we may need to delay the CPU clock edges

by at least 8.8ns. For example, a SN74F32N OR gate has a propagation delay of 2.2ns to 6.6ns. To get >=8.8ns we would need to put four of these OR gates in series. That could lead to up to a 4*6.6=26.4ns delay, which would mess with the next SRAM cycle's address requirements. So a puzzle: how to get a predictable propagation delay of a certain size?

How about a circuit like this?



The basic relation between current and voltage for a capacitor is $I(t) = \frac{\mathrm{d}Q(t)}{\mathrm{d}t} = C\frac{\mathrm{d}V(t)}{\mathrm{d}t}$. We would want the voltage input to the Schmitt trigger to be changing at something like one volt per nanosecond. With a 100Ω resistor, that would imply 1 picofarad capacitors. Those are about the smallest capacitances available in ceramic capacitors with through-hole mounting, and they only come in very low precision (+/- 0.25 picofarads). With a 10Ω resistor we are at more reasonable 10pf capacitors, but the instantaneous currents may be ridiculous. V=I*R with 5v and 10Ω gives I=500ma, way more than the counter chip outputs should be driving, although this is an extremely short transient thing so maybe this doesn't matter? The clock output evidently drives hard low (20ma) but weakly high (1ma). So maybe we need a pull-up resistor to help balance the rise and fall times at the input of the Schmitt trigger. Also, we don't really need two capacitors. Here is a revised design:



Another option is just to use a long-enough wire. The length of the wire is a tuning parameter: longer wire → longer delay. From wikipedia:

> Wires have an approximate propagation delay of 1 ns for every 6 inches (15 cm) of length.

Hence we would need something like 60 inches (five feet) of wire to get a 10ns delay.

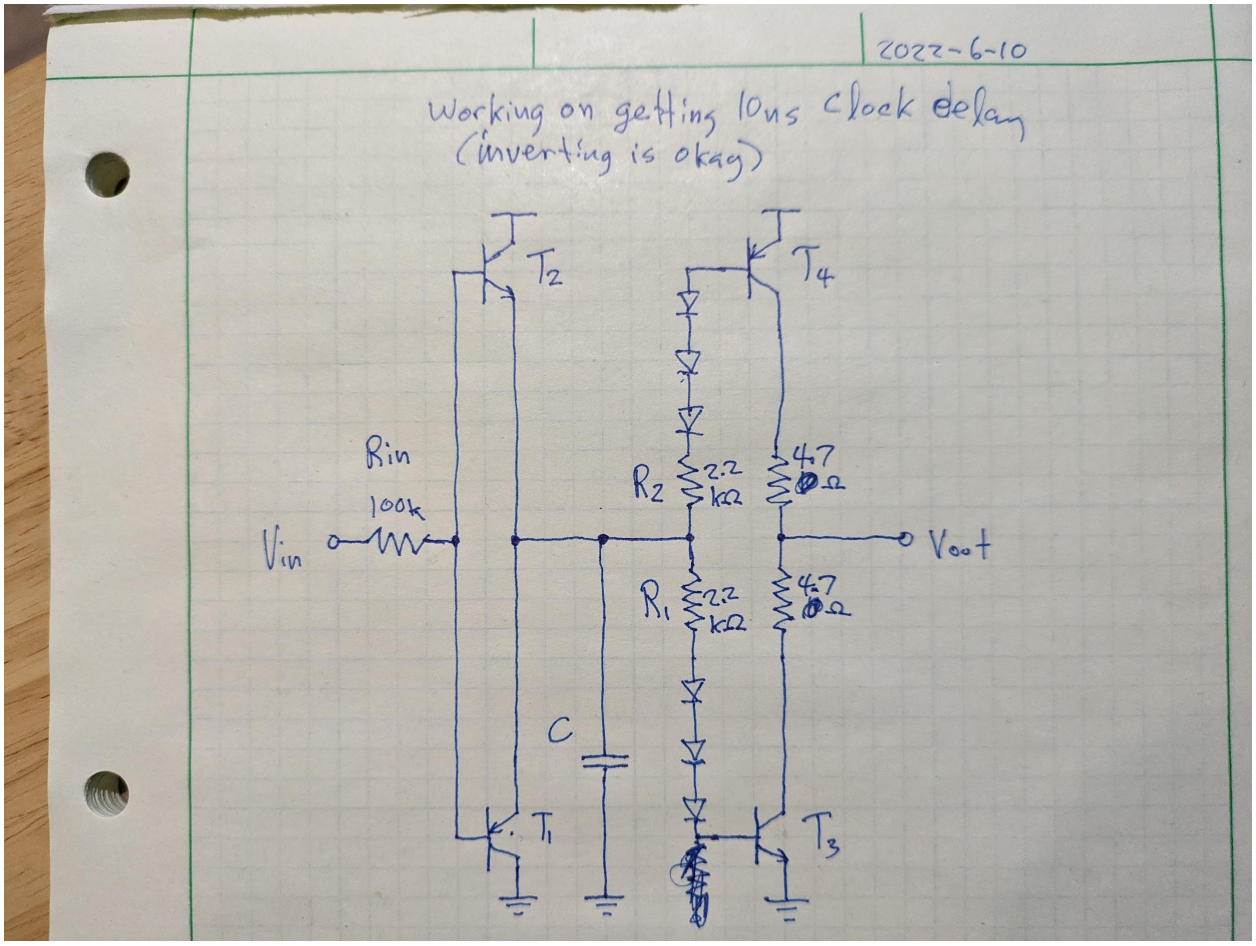
Since this is getting messy, maybe we can find a better part to switch the address lines input to the SRAM from video or CPU. Here are some options:

- 74FST3257DTR2G FET quad 2:1 mux/demux, 1.0-4.7ns select $0.63
- ADG3257BRQZ FET quad 2:1 mux/demux, 4-12ns, $3.03
- PI5C3257QEX quad 2:1 mux/demux, 0.5-5.2ns, $0.44
- QS3257QG quad 2:1 mux/demux, 0.5-5.2ns, $0.77
- SN74CBT16233DGG 16 2:1 mux/demux, 1.6-5.3ns, $0.87 (bulk only)

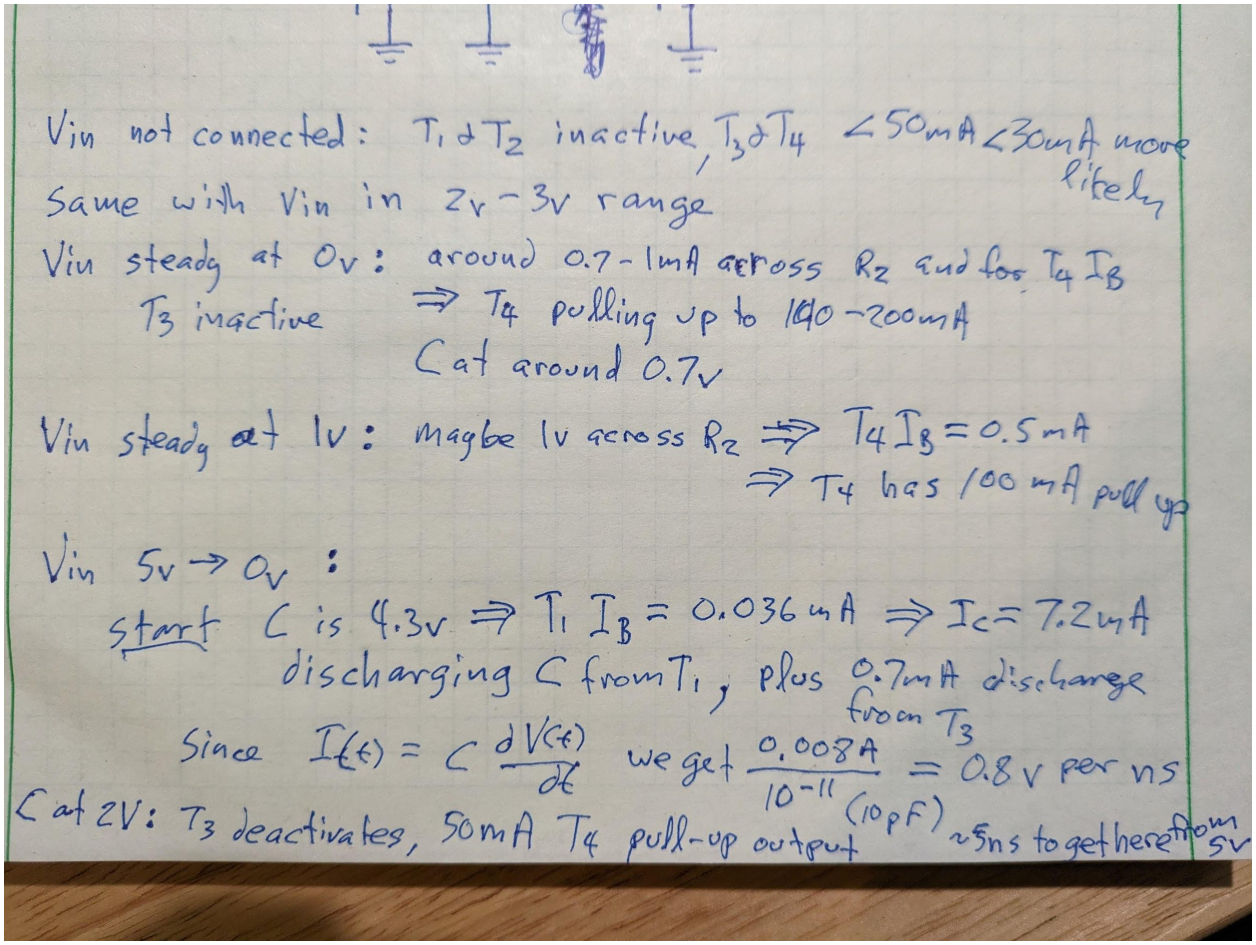
- ADG3257BRQZ-REEL, quad 2:1 mux/demux, 8-12ns, $3.03
- SN74CBT3257CDBR quad 2:1 mux/demux, 1.5-5.8ns, $0.61
- CBT3257APW,118 quad 2:1 mux/demux, 1.4-5ns, $0.64
- NLV74FST3257DR2G quad 2:1 mux/demux, 1.0-4.7ns, $0.85
- SN74CBT16233DGGR 16 2:1 mux/demux, 1.6-5.3ns, $2.09 ($1 in quantity)
- SN74CBT3257DBQR quad 2:1 mux/demux, 1.6-5ns, $0.88 ($0.30 in quantity) [tightest spread: 3.4ns from min to max propagation delay]

To get precise but discrete clock shifts, we could use a 100.7 MHz oscillator. We could use this to count with different count start times, and even use the positive and negative clock edges to get double the precision. With this we would have a selection of shifts that are multiples of 5ns. For example, after selecting between connecting the video or CPU address lines to the SRAM, we need a 10.2ns shift +/- 4.8ns. We could choose the 10ns shift and be off by only 0.2ns, well within tolerance. However, the parts needed for this sort of high frequency work are getting into ECL territory – all the TTL logic chips have very wide ranges of propagation delay, making it hard/impossible to get within tolerance without a lot of measurement/calibration that could get out of date with manufacturing changes.

So I went with designing my own analog/TTL delay circuit. After several iterations, I came to this:



Here are some notes on how this functions:



It looks like the common 2N2222 (NPN) and its companion 2N2907 (PNP) are pretty nice, popular components that might work for this application, so we should start there and measure results. If needed, we can go with higher frequency/faster switching parts.

Note: I am likely to just use 74F04 hex inverters to get close enough to the right delay. If that is not good enough, I will try the analog option above.

PROGRESS ON ACTUAL BUILD

I've now got a critical mass of parts to start prototyping modules in some bootstrapping order. It seems like a super fundamental module that is depended on but does not depend on other modules is a clock/time counter module.

CLOCK MODULE

I started by using the 12-bit ripple counters tied to the 25.175MHz crystal oscillator module, but realized the ripple counters have nasty glitches in the count values if you use their outputs directly. While it is possible to do some sort of latching to fix this, that seemed overly

complicated. Searching around, there is a sweet 74xx163 counter chip that deals with this latching nicely. I also played with using 74xx175 flip flops directly, maybe combined with some logic similar to what the 74xx163 has. This would have the advantage of providing normal and inverted outputs per clock counter bit.

My second build-out after the ripple counter demo was using straight 74F163's clocked directly from the 25.175MHz clock. This worked when chaining together around six of them (4*6=24 bits count out of the 32 bits I was aiming for), then the seventh counter chip would not count. It turns out this was due to the chained propagation delays of the carry lookahead logic: it just could not manage that many chips in the chain at the high clock speed. So I placed a one-bit flip-flop counter using a 74F175 at the beginning to divide the fast video clock in two, to 12.5875MHz, and was able to chain eight 4-bit 74F163 counters off of that. With that fix, I think the clock module is adequate, and will supply the following interface lines: GND, +5v, clear input, 25.175MHz (640x480 VGA dot clock), 12.5875MHz (CPU clock), and count bits 0 to 31. Total lines used: 37. It turns out that the big soldered proto boards I have include 37 pins on the edge connector array. What luck!

The clock module counter has 32 bit, with cycles time going from 6.29375MHz in powers of two divisions:

- Bits 0-3:
    - 6.29375mHz/158.9ns, 3.146875mHz/317.8ns, 1.5734375mHz/635.6ns, 786.71875kHz/1.271us
- Bits 4-7:
    - 393.359375kHz/2.542us, 196.6796875kHz/5.084us, 98.33984375kHz/10.169us, 49.169921875kHz/20.34us
- Bits 8-11:
    - 24.5849609375kHz/40.68us, 12.2924804688kHz/81.35us, 6.14624023438kHz/162.70us, 3.07312011719kHz/325.40us
- Bits 12-15:
    - 1.53656005859kHz/650.80us, 768.280029297Hz/1.30ms, 384.140014648Hz/2.60ms, 192.070007324Hz/5.21ms
- Bits 16-19:
    - 96.0350036621Hz/10.41ms, 48.0175018311Hz/20.83ms, 24.0087509155Hz/41.65ms, 12.0043754578Hz/83.30ms
- Bits 20-23:
    - 6.00218772888Hz/166.61ms, 3.00109386444Hz/333.2ms, 1.50054693222Hz/666.4ms, 0.75027346611Hz/1.333s
- Bits 24-27:
    - 2.666s, 5.33s, 10.66s, 21.33s
- Bits 28-31:
    - 42.65s, 85.30s, 170.60s, 341.2s

8x8 LEDs DISPLAY MODULE

For the LED display module based on the 8x8 matrix units, the first test with the combo of TBD62783APG driving the 8 anode rows and TLC5916IN driving the per-8x8-unit cathode columns seems to work well. The TBD62783APG will drive just one row at a time, switching rows at up to 1.57MHz for a 1 bit per pixel display. A simple scheme is to read one byte from SRAM every CPU clock cycle (the video half of the cycle), place that into a parallel-in, serial-out shift register, and shift the bits into the chain of up to 7 TLC5916IN's. After all TLC5916IN's are full we simultaneously latch them and switch to the respective row with the TBD62783APG. If we waste one of 8 SRAM reads, we could use that time slot to latch and not shift, and do the row change timed to match the latch output change taking effect (thus avoiding needed to blank the display driving using the TLC5916IN's output enable, instead leaving it enabled all the time).

We can use bits 12,13,14 for the row index (row changes at 3kHz). Use bits 9,10,11 for the 8x8 block index, bits 6,7,8 for the bit-shift index, and bit 5 for the bit shift clock. Mainly we will leave the latching off, but turn it on briefly when changing to a new row (this will be on a much shorter time scale than the bit shifting clock). Bits 9,10,11,12,13,14 will form the SRAM memory address. Actual reading of the SRAM will happen in a tiny time window (one video half of a CPU clock cycle) near the beginning of each block time period, to get the byte for that block/row.

Thinking about the bigger picture, I'm not sure going to a full bitmap display from SRAM is helpful at this early stage of the design/build bootstrapping. So I will pivot to focusing on writing to the NVRAM, and visualizing that process using maybe just one row of the 8x8 LED blocks showing bytes verbatim/1:1 as binary 8 bits ⇔ 8 LEDs.

HEXADECIMAL KEYPAD

Inputs: 10 consecutive clock lines (bits 6-15), ground and +5v
Outputs: positive-edge clock line, 4 data lines, for hexadecimal keypress events

The keypad is scanned 192 times per second. For each keypad row, four R-S latches are set initially, and selectively cleared should a key down be detected for the respective key in the row. Near the end of the per-row time slot, these R-S latches are copied/clocked-out to a set of four serial-in/parallel-out shift registers, and before the row increments, the shift registers are scanned one-at-a-time to look for key up-to-down (keypress) transitions. Each keypress event found generates a positive-going pulse on the output clock line, and the four associated data lines are the row and column bits appended together to form the hexadecimal value of the key being pressed.

NVRAM DATA ENTRY MODULE

Based on keypress events from the hexadecimal keypad module, the data entry module allows setting up a 16-bit address and 8 bit data value, visualized 1:1 as an LED per bit, and has a 'write' button to store the byte at the address. The NVRAM reads the byte at the selected address and visualizes this in a separate 1:1 LED display.

The data entry module is fully functional and tested on a cheap 32kB SRAM as of 2023-06-30. Next step: try getting the expensive NVRAM working. This will require delicate soldering to the proto adapter board, plus hardwiring the extra unused address lines.

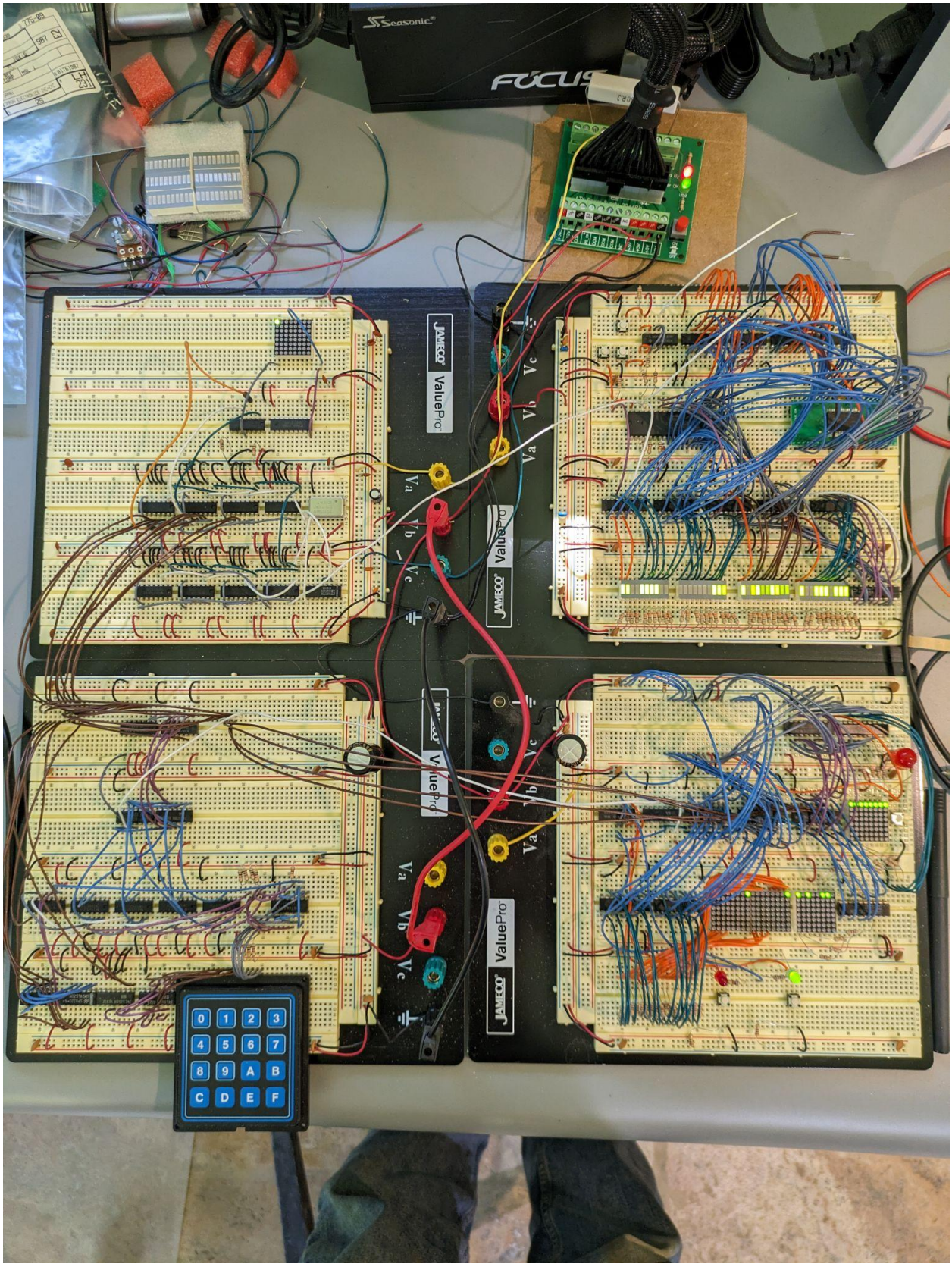
By mid July 2023 the NVRAM editing was working. The first NVRAM module got borked, the data MSB stuck high, likely due to the difficulty of soldering the dang TSOP II chip to the DIP adapted board. My second try soldering a second NVRAM worked. The NVRAM indeed remembers the data I have written to it, despite power downs.

INITIAL 6502 MODULE

To start, I am making the 6502 setup run off or a manual half-cycle-step clock, with an S-R latch set high/low with respective pushbuttons. I've also added a reset pushbutton and lots of 1x10 LED arrays to visualize all the state, including address, data and clock/control/output lines. I'll put in a simple clock/counting program in the NVRAM, add memory-mapped output to an 8-bit latch, and see how it goes.

It worked! The memory-mapped output latch shows the count incrementing correctly during half-step execution, and the latch bits show the correct frequencies on the oscilloscope when running at various CPU clocks.

Here is the result:



The upper left is the clock module, with a 25.175mHz crystal oscillator/initial clock signal, a

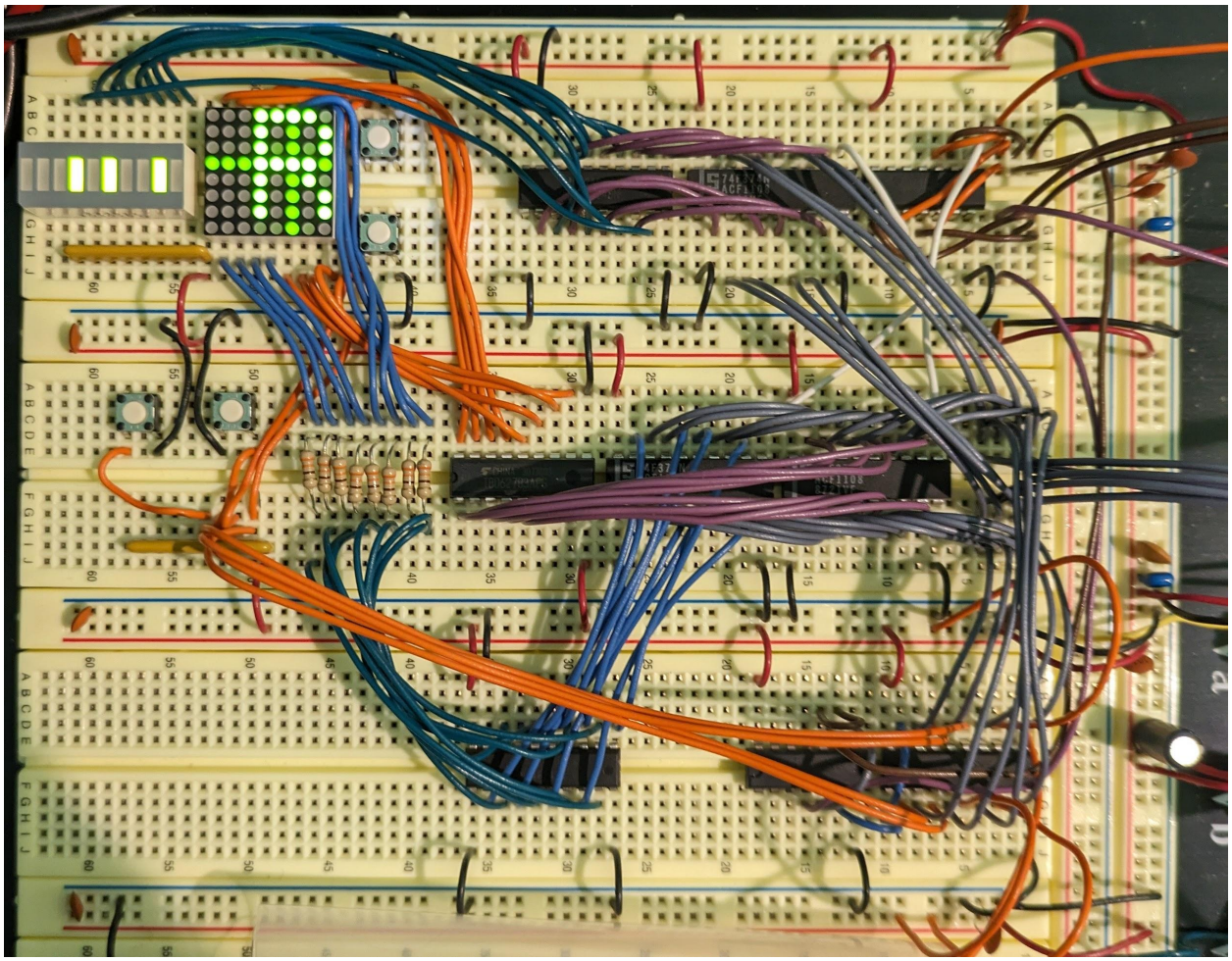
74f175 dividing that in half to 12.5875mHz, and then 8 74F163 4-bit clock/latch counters, for a total of 34 different clock frequencies. The lower left is the hexadecimal keypad scanning logic, going from keypresses on the keypad 4x4 matrix to keypress events on five wires, one for clocking a keypress event and four representing the hex value. The lower right is the nonvolatile SRAM manual visualizer/programmer that takes hexadecimal keypresses and forms a 16-bit address and 8-bit data value. A “write” button allows that value to be placed into the SRAM. On power off the SRAM is automatically copied to an identically-sized non-volatile storage area on the same chip. I move the chip over to the upper right module, which is the 6502 CPU centered circuit, and run whatever I had programmed in. Finally, the upper right is the 6502 module, with the CPU on the left, the NVSRAM on the right, and the LEDs visualizing all the state in the bottom half. The upper row of chips is dealing with clock, address and memory-mapped I/O logic. The actual memory-mapped I/O register is just to the right of the LED banks. The LED banks show, from left to right, reset, CPU clock, NC, NC, A15 through A0, R/W, NC, D7 through D0, memory-mapped latch clock, NC, and latch D7 through D0.

2023-08-02: I’ve converted the clock module to a soldered breadboard. Testing it isolated from the other modules was fine, all power-or-two count bits measured correctly. However, when connecting to the hexadecimal keypad module and the NVRAM byte editor module the keypresses went haywire. It turns out the long wires were getting too noisy, picking up some clock-switching spikes, and creating false keypress signals on the keypress event line. Adding a 22pF capacitor on the keypad module end of the event line cleaned that up. Here is the state of the modules with the soldered clock module in place, and the upper left breadboard cleared and starting to be populated for a font editing module.

UP
ENTER
Seasonic
FOCUS
JAMECO
ValuePro
Va
Vb
Vc

FONT EDITOR

The next module I built is the font editor. Since I can now program the W65C02S by byte editing the NVRAM, it made sense to go with minimal hardware and put most of the logic into software. The finished solderless version as of 2023-08-23 looks like this while in use:



I tried to make the user interface as intuitive and self-explanatory as possible. The use of grayscale for highlighting what you are editing means you can get a feel for what the button presses do quickly. My son, with no instructions, was able to figure it out and make a heart shape in 5 minutes of playing around.

The software interface from the font editor module to/from the CPU consists of:

1) Output latch at $C000 that drives a bar of LEDs (8 of 10 are used). For the font editing, this shows the ASCII value of the character being displayed and edited.
2) Output latch at $C001 that drives the 8 row anodes of the 8x8 LED font matrix display. We typically activate (set high) just one row at a time, with the row scan clocking at 12.3 kHz. The whole array is thus scanned at 1537.5 Hz, and with 8-bit grayscale via PWM we end up with full grayscale scanning effectively at 192.2 Hz.
3) Output latch at $C002 that drives the column cathodes (bit set means LED on).

4) Input buffer at $C003 that allows reading of bit 10 of the clock module in the high bit and four push button states in the lower four bits (the remaining bits are unused for now). The buttons are:
    a) Bit 0: down button
    b) Bit 1: up button
    c) Bit 2: right button
    d) Bit 3: left button

The hardware interface of the font module to previous modules is:

- The clock module bit 10 signal,
- the 8 CPU data R/W lines,
- the CPU RWB line,
- the active-low write enable line (combination of RWB and CPU clock), and
- four memory-mapped I/O address decode lines for $C000-3.

The hardware specific to the font editor module is:

- 3x 74F374 8-bit latches
- 3x 74F240 inverter/buffers
- 1x TBD62783APG 8-bit anode driver
- 8x 330Ω current-limiting resistors to avoid burning out the 8x8 LED matrix
- 1x SIP bank of 8 330Ω resistors connected together on one end to limit current to the LED bar.
- 1x 74F32 quad positive OR gates to generate the latch clock signals from the write enable and respective address selection lines.
- 1x 74F00 quad NAND gates to generate the output enable of one of the `240s to output to the CPU data lines, based on the $C003 address decode selection line and the CPU RWB line.
- A 10 LED bar (currently using only 9 of them)
- An 8x8 LED matrix
- 4x push buttons

I'll work on putting out some schematics.

After shoring up the power bus wiring, I'm now running stably at 12.5875MHz with all five modules hooked together: clock/counter, hexadecimal keypad scanner, NVRAM byte editor, CPU module, and not the font editor.

To keep my sanity, I modularized the code into moderately small subroutines, since I am only using a byte editor to code in machine language right now, and don't have the ability to add or remove code from a routine except at the end of it, and thus end up rewriting the whole thing if there is an error or modification needed.

The memory state for the font editor consists of:

- $00,$01: low and high byte of a 16-bit loop counter for timing. One application loop takes 81.4 usec (12.3 kHz). 256 loops takes 20.82 msec, and the full 65536 loops before wrapping takes 5.333 seconds.
- $02: the ASCII value of the character being edited.
- $03: indices of the row and selection of ASCII/row/column being edited at the moment.
    - Bits 0-2: index of row being edited
    - Bits 3-6: 4-bit code of what is being edited
        - 0000: up/down add/subtracts $10 from the ASCII value.
        - 0001: up/down add/subtracts 1 from the ASCII value
        - 0010: up/down moves the row being edited up/down.
        - 0011: leftmost/bit 7 pixel column for row is turned on/off with the up/down buttons.
        - 0100-1010: Bits 6 through 0 are similarly edited using the up/down buttons.
    - Bit 7: always 0
- $04-$07: up/down state and timers for buttons 0-3 respectively.
- $08-$0B: The auto-repeat countdown timer for buttons 0-3 respectively.
- $E000-$E7FF: the font itself, with 8 bytes per character (8x8 bit mask), 2kB total for 256 character bitmaps.

The font-editing code is broken up into the following routines:

1. Generic main loop that does general 6502 reset initialization (stack pointer to $1FF, $00 into status register), calls the application-specific initialization routine, and then in a forever loop calls the application per-loop work routine.
2. Font-editor init routine: clears $00-$07 to zeros, a reasonable initial state, then calls the clock sync routine so that the first-loop timing relative to the clock edges is the same as the Nth loop, in case that matters.
3. Get column mask in A from $03 edit selection variable.
4. A = A >> Y
5. A = 1 << Y
6. Wait for the clock line to change (used for synchronization/timing).
7. $10,$11 = address low,high of first byte of font memory for ASCII value in $02
8. Font editor per-loop work:
    a. Handle key press events
        i. Handle button indexed by X for X=3,2,1,0, generating key press events on first down press with debouncing, and for key press auto repeats for long button holds.
        ii. Handle button X key press event. The left/right buttons navigate what is being edited, and the up/down buttons change state on whatever is being edited.
    b. Update display. The ASCII value is put in the LED bar using $C000, the active row in the 8x8 LED matrix is set using $C001, and the highlighted font row is output using $C002.

        i. Highlight font row bits in A based on frame number modulo 8, to get grayscale visual cues as to what is being edited.
    c. Increment loop count $10,$11
    d. Clock sync

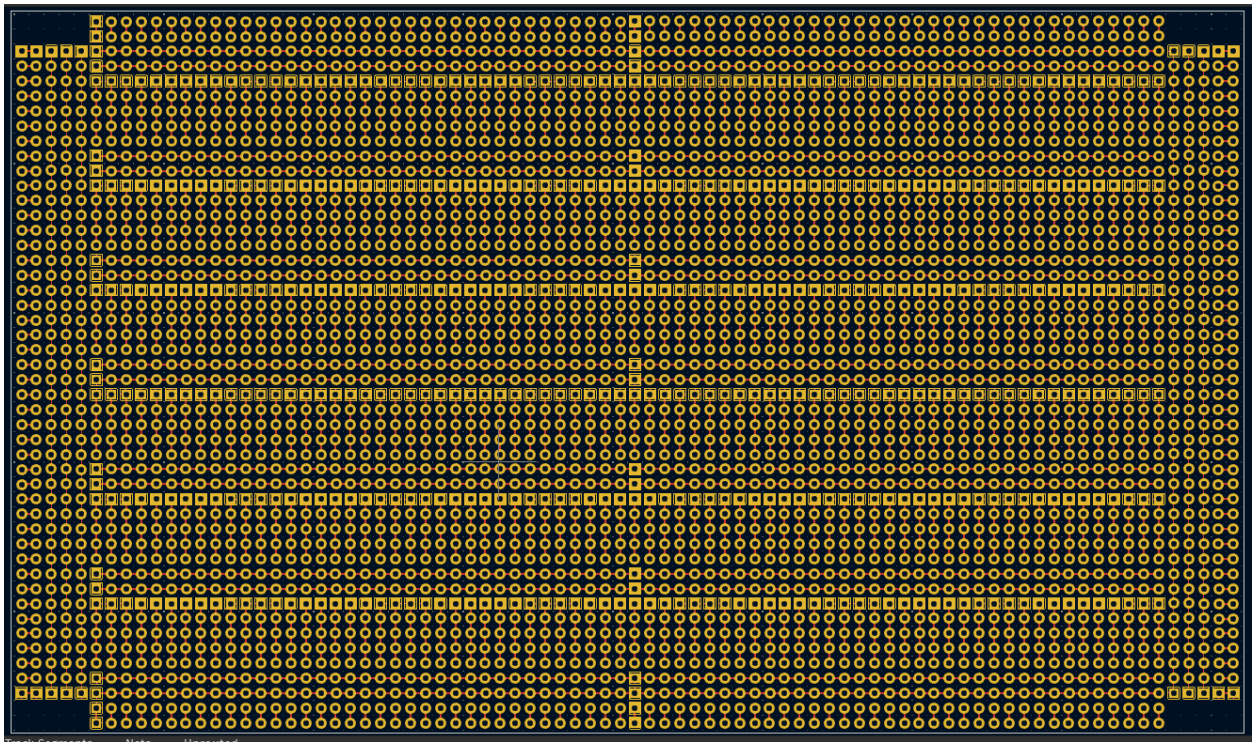
Overall the font editor code comes in at under 768 bytes, and lives at $F000-$F2FF. Boy, that was a lot of tedious hexadecimal data entry! I'm really looking forward to getting a more proper display, full keyboard input, and assembler going.

PCB MANUFACTURING

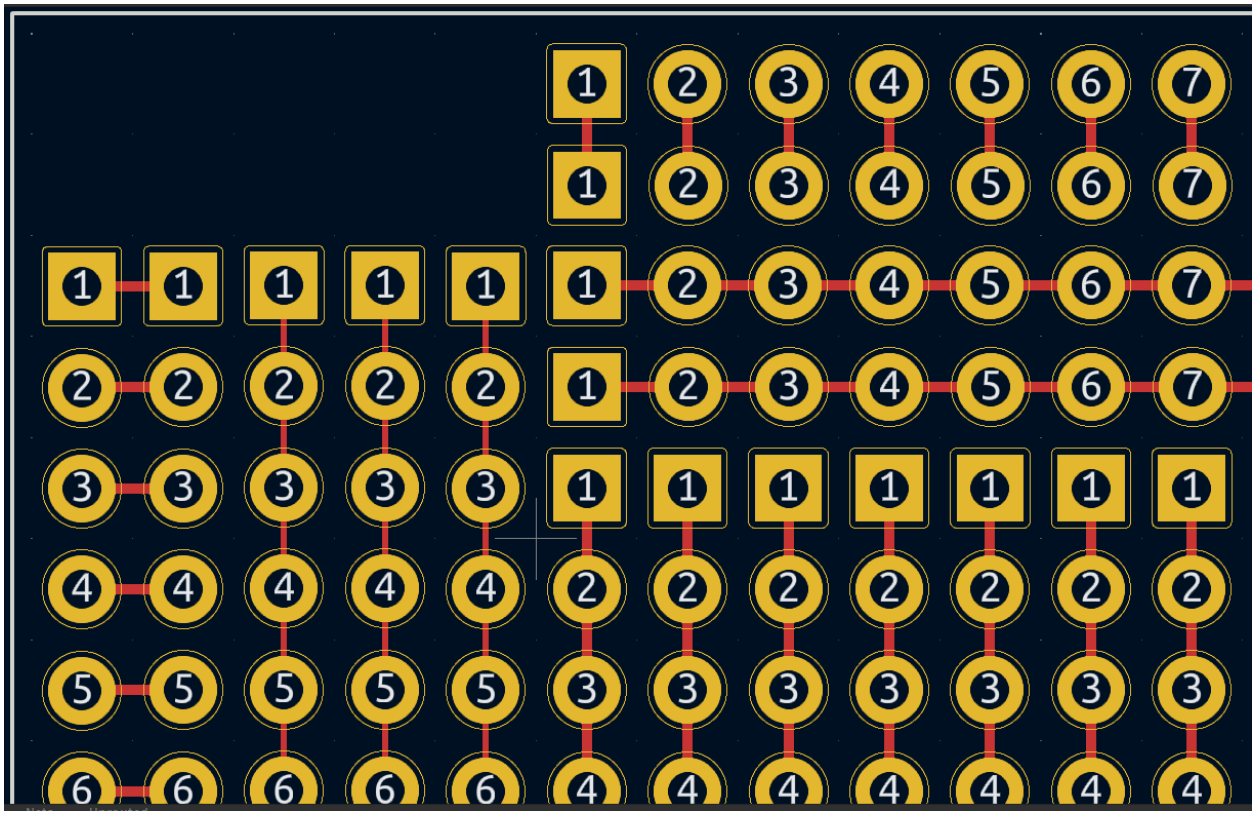
An in-depth comparison of ~10 manufacturers:
https://camptechii.com/the-2021-list-of-top-8-printed-circuit-board-manufacturers-serving-north-america/
A reddit discussion thread on remaining US PCB manufacturers:
https://www.reddit.com/r/PrintedCircuitBoard/comments/lmvdop/any_good_pcb_manufacturers_still_left_in_canadausa/?onetap_auto=true
A Kicad forum discussion on US-based Kicad-friendly manufacturers:
https://forum.kicad.info/t/usa-board-house/27856
A more recent reddit thread on quick-turn US-based manufacturers:
https://www.reddit.com/r/PrintedCircuitBoard/comments/10g623u/usa_quick_turn_pcb_recommendations/

I've been looking into designing my own perf/strip board. Something like three rows of ICs with power busses down the middle and stacking shield type connectors on three out of four of the outside edges. Something like 6x11 1x5 strips in a row on either side of the bus strips. 3x14 pins on the cross-bus-strip direction for the IC rows plus bus strips. And the 2x66 and 2x42 sets of pads on the edges for stacking connectors and wiring for them. Maybe a set of bus strips going perpendicular to the IC row bus strips.

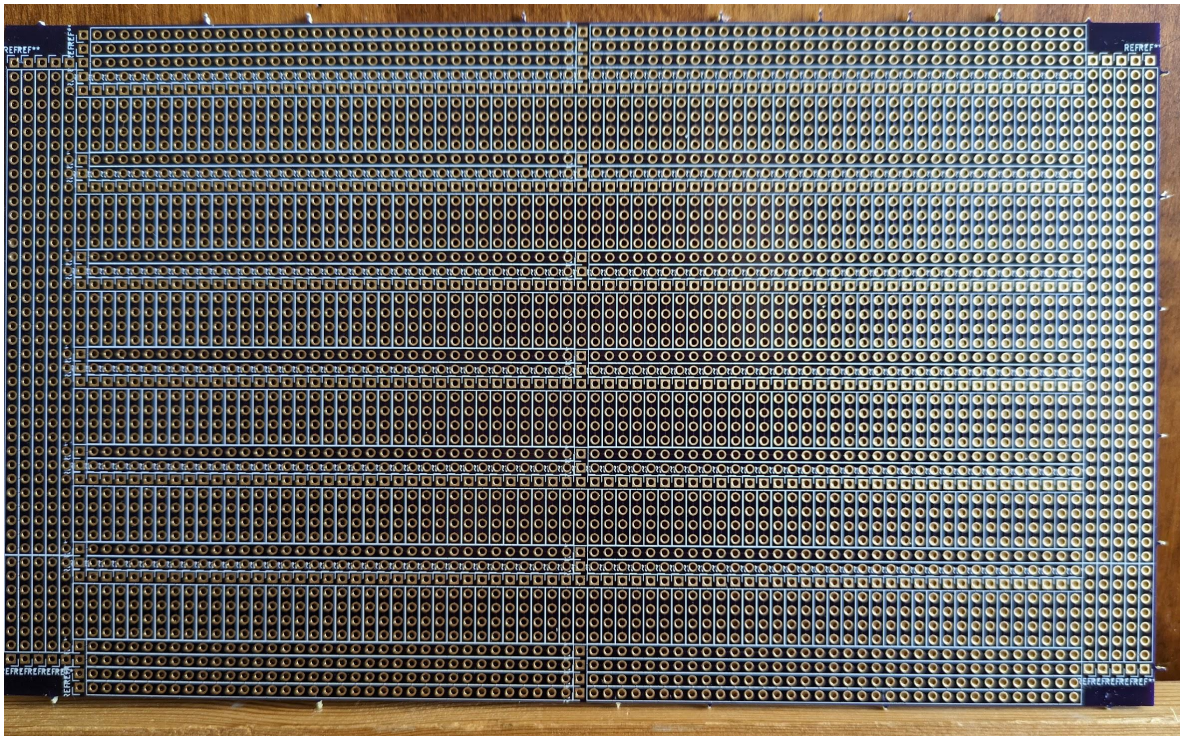
I created a PCB design and manufacturing files in Kicad. I got the board where I would like electrically, but the silk screen is a horrid mishmash. I will need to make my own footprints for 1x5 and 1x72 and 1x44 header strips. I sent the design to OSH Park, which was quite easy, and they will be making three copies for around $200. This is a big board, and costs are proportionate to area as I understand it.

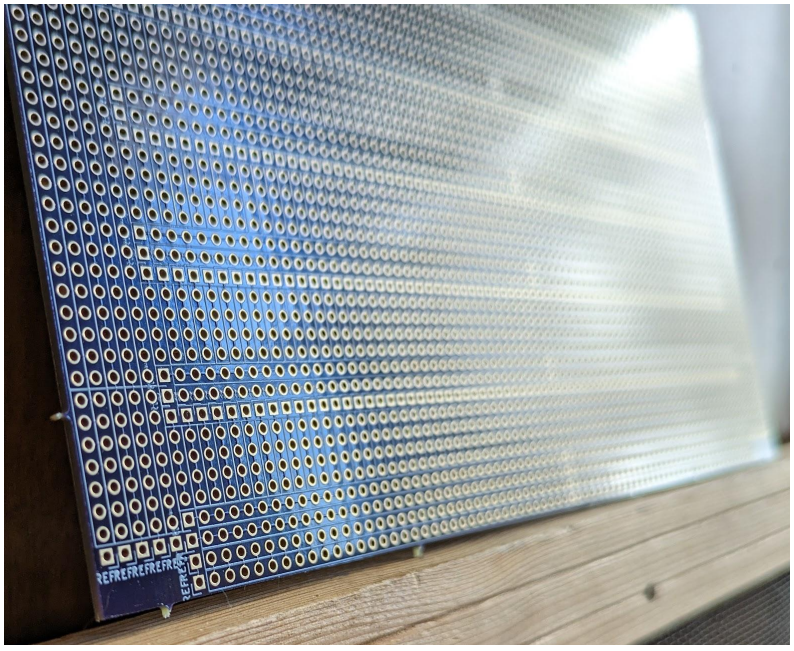


Upper-left detail:

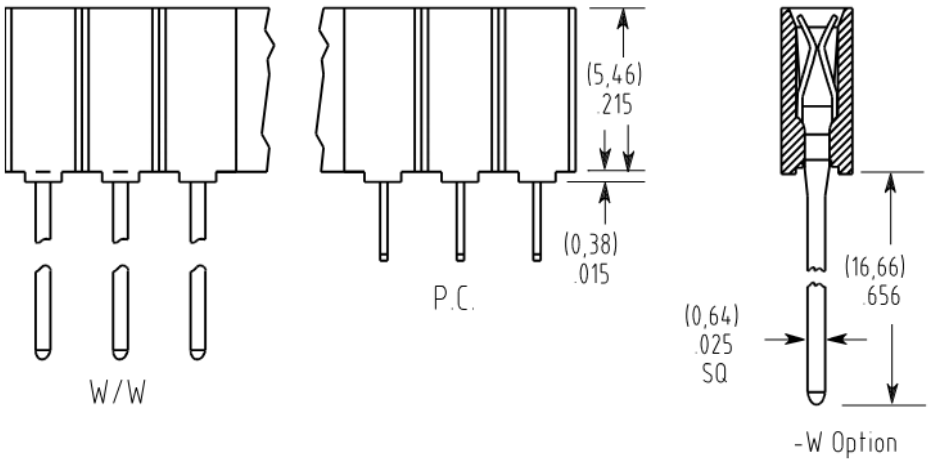


Resulting boards that came back from OSH Park:





The best headers I could find so far to stack this board is a Samtec SSA-124-S-G, which looks like this:



The best price I could find for quantity 100 was Sager: https://www.sager.com/ssa-124-s-g-93868.html?utm_source=OEMSecret&utm_medium=click&utm_campaign=sager-brand
On 2023-08-02 the per-item cost at quantity 100 was $3.49. Pretty pricey but these are nice parts (good solid design, breakable, gold plated). Two of these will be needed for our proto board's smaller side, and three for the longer side. For two small and one long side, as planned, a total of 7 connectors per board are needed, for a total of $24.43 per board.