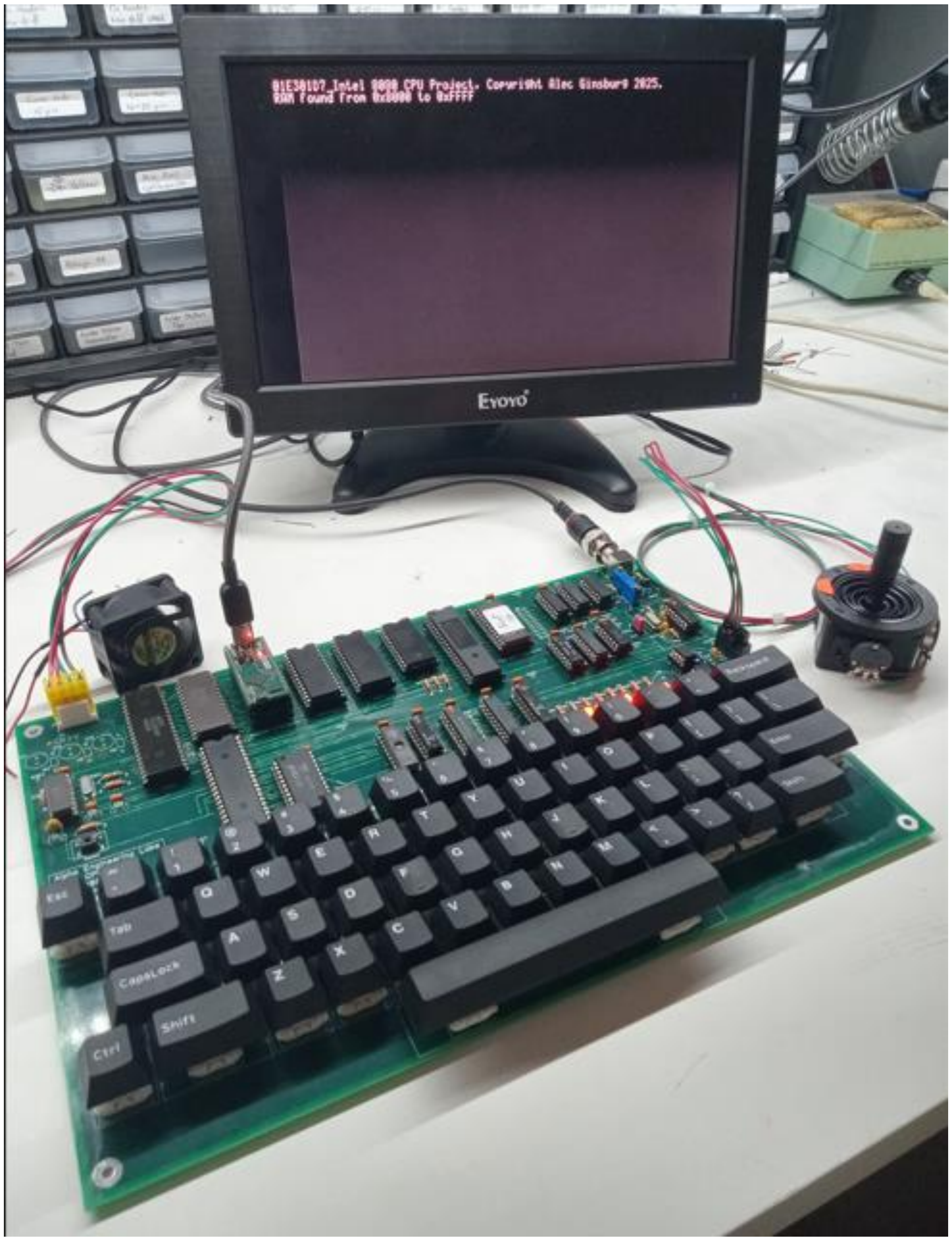
8080 Project

7601900 Board



```
01E301D7_Intel 8080 CPU Project. Copyright Alec Ginsburg 2025.
RAM found from 0x0000 to 0xFFFF
```

Contents

Purpose

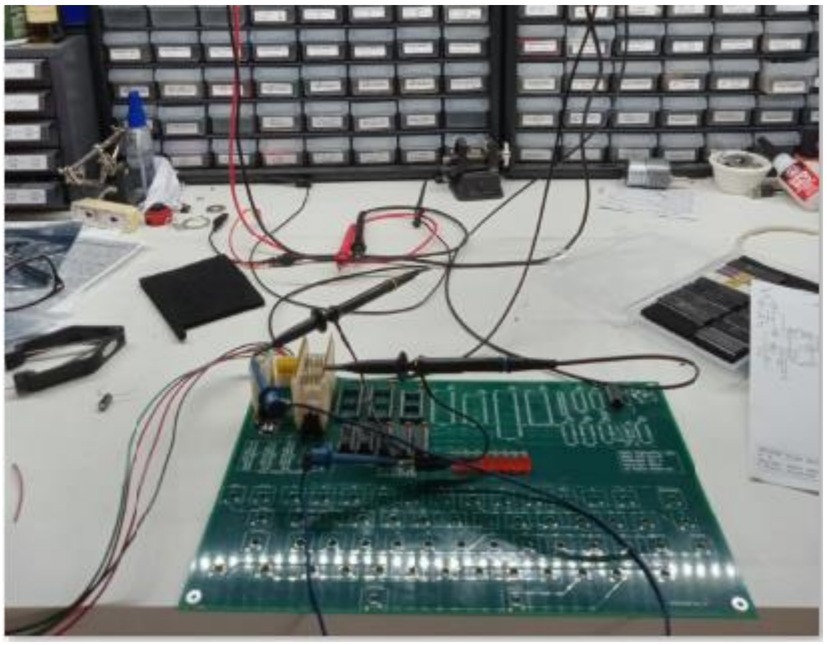
One of my bucket list items was to build something that used a bunch the really old ICs I've been carrying around for decades.

To make it more 'fun' I decided to use the oldest parts I had which turns out to be an 8080 based CPU.

I had to look around online for a few parts that I didn't have (such as the 8224 clock chip), and I also bought a few extra copies of the chips I did have just in case some of them didn't work anymore.

Rev A PCB

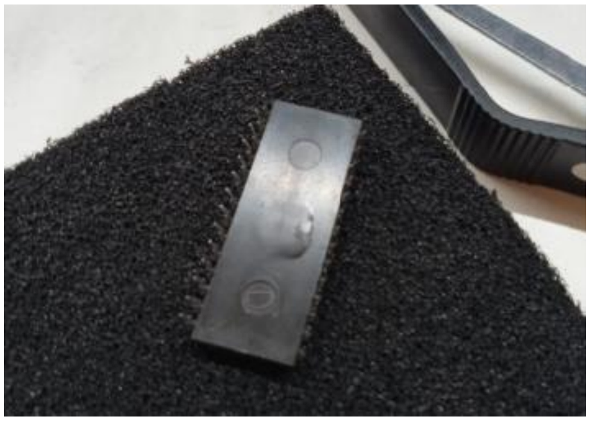
I put together a rough schematic and had a PCB made for a simple CPU board with a keyboard and video display. I spent considerable time making sure that I got all of the keyboard switches in the right place on the board.



The 8224 clock chip (that I got online) and the 8080A chip that I had both came up working! I was pretty amazed because I'd been lugging the 8080A around for decades.

Component Issues

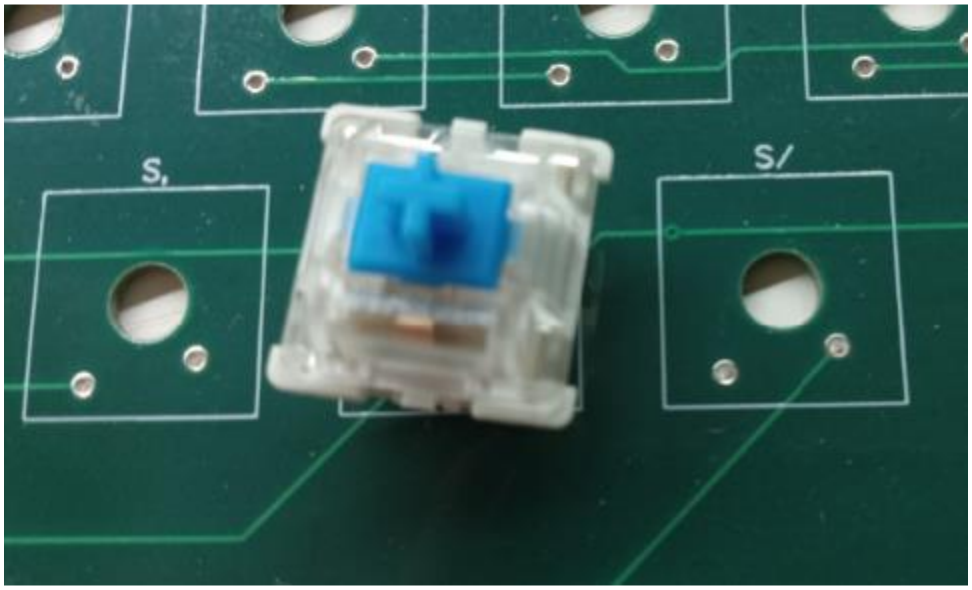
However, when I went to test the 8228 Bus Driver that I had, it did not work. In fact, it caused the 8080A signals to be distorted. After a bit of troubleshooting, I realized that the 8228 had an internal short. The chip would get hot, and when I eventually took it out and looked at the bottom it was very apparent that it was toast. You can see the packaging developed a ‘blister’ in the image below.



Unfortunately, I only had one of the 8228 chips. I was able to order more but it will be a week or two before they arrive.

Layout Issues

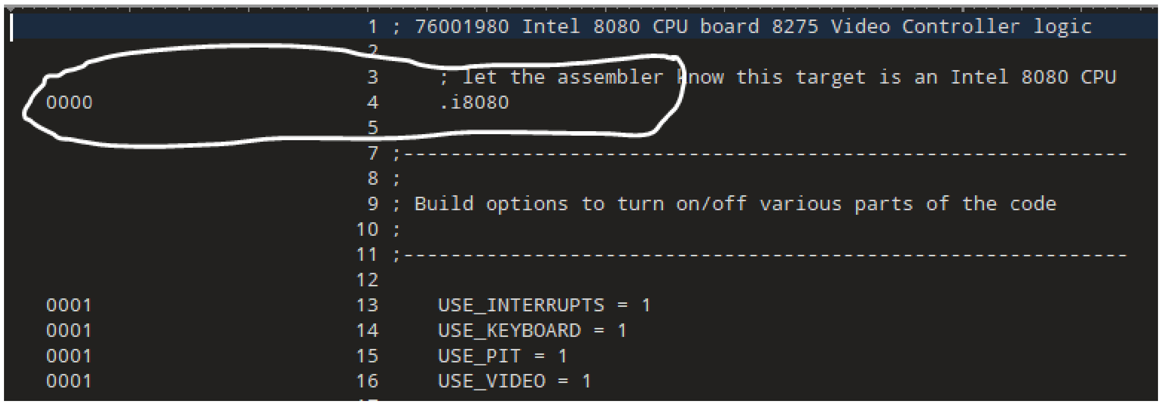
I then moved on to the keyboard, only to find that, while I'd spent lots of time perfectly positioning each key, I completely got the key pattern wrong. As you can see below, the keys end up rotated and unable to be fully inserted! This was a disaster. I was thinking of doing cuts and jumpers to fix stuff I missed, but this made the board virtually unusable – so I had to do another PCB layout to fix this and a number of other small issues that I'd found.



Tool Issues

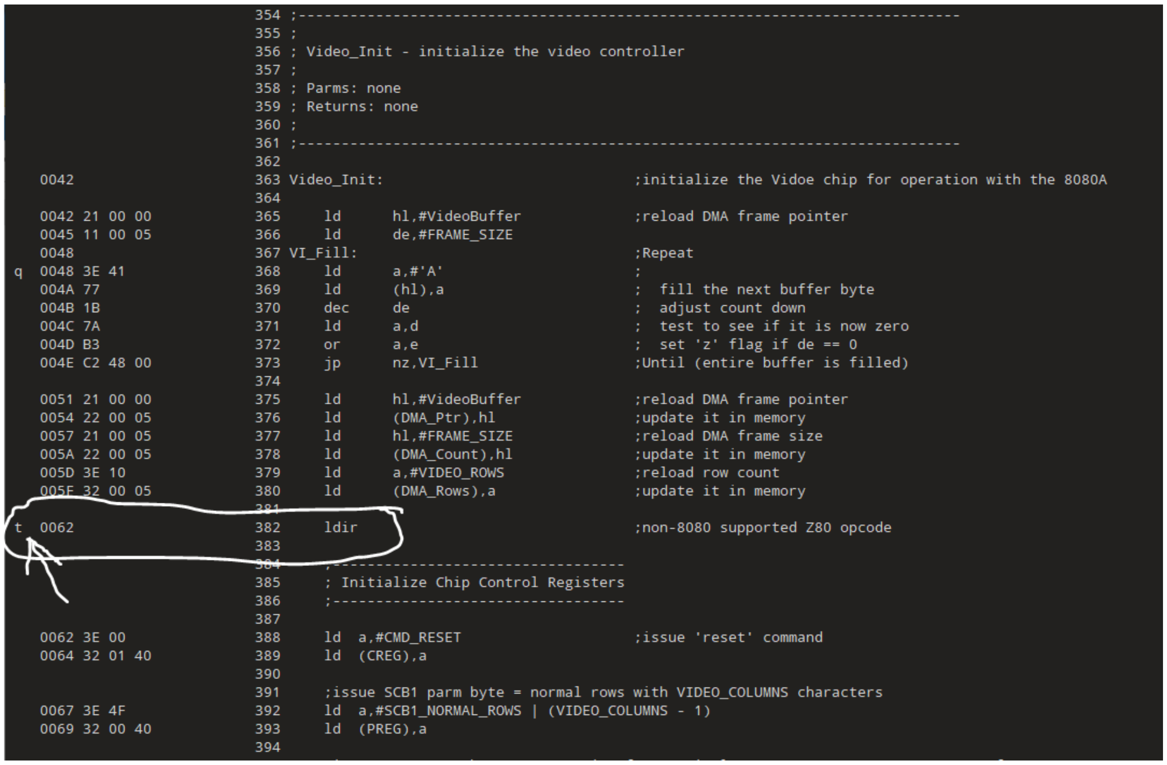
While waiting for parts to come, and between testing hardware, I started writing the bootROM 8080 assembler code. I chose to use the GCCTools (follow link for more info) that I had previously used with my Z80 projects. Although the 8080 mnemonics are a subset of the Z80 mnemonics, it was very difficult to remember 'which' Z80 opcodes could NOT be used on the 8080.

In order to improve this situation, I spent some time modifying the source code for the GCCTools assembler to add an '8080' target architecture opcode.



```
                        ; 76001980 Intel 8080 CPU board 8275 Video Controller logic

                           ; let the assembler know this target is an Intel 8080 CPU
  0000                     .i8080

                        ;------------------------------------------------------------
                        ;
                        ; Build options to turn on/off various parts of the code
                        ;
                        ;------------------------------------------------------------

  0001                     USE_INTERRUPTS = 1
  0001                     USE_KEYBOARD = 1
  0001                     USE_PIT = 1
  0001                     USE_VIDEO = 1
```

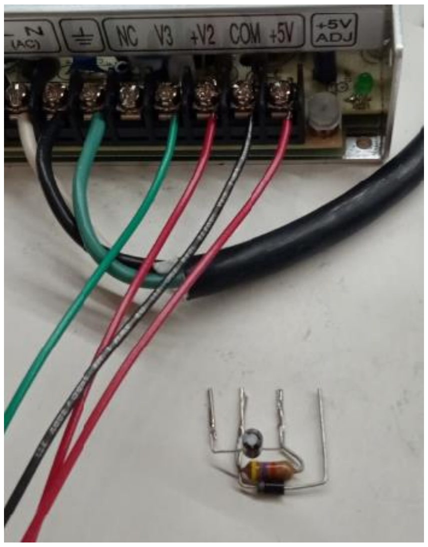
This flags opcodes used that were not supported by the 8080 processor with a 't' – see screen shot below.



```
                   ;---------------------------------------------------------------------------
                   ;
                   ; Video_Init - initialize the video controller
                   ;
                   ; Parms: none
                   ; Returns: none
                   ;
                   ;---------------------------------------------------------------------------

   0042            Video_Init:                          ;initialize the Vidoe chip for operation with the 8080A

   0042 21 00 00       ld      hl,#VideoBuffer          ;reload DMA frame pointer
   0045 11 00 05       ld      de,#FRAME_SIZE
   0048            VI_Fill:                             ;Repeat
q  0048 3E 41          ld      a,#'A'                   ;
   004A 77             ld      (hl),a                   ;  fill the next buffer byte
   004B 1B             dec     de                       ;  adjust count down
   004C 7A             ld      a,d                      ;  test to see if it is now zero
   004D B3             or      a,e                      ;  set 'z' flag if de == 0
   004E C2 48 00       jp      nz,VI_Fill               ;Until (entire buffer is filled)

   0051 21 00 00       ld      hl,#VideoBuffer          ;reload DMA frame pointer
   0054 22 00 05       ld      (DMA_Ptr),hl             ;update it in memory
   0057 21 00 05       ld      hl,#FRAME_SIZE           ;reload DMA frame size
   005A 22 00 05       ld      (DMA_Count),hl           ;update it in memory
   005D 3E 10          ld      a,#VIDEO_ROWS            ;reload row count
   005F 32 00 05       ld      (DMA_Rows),a             ;update it in memory

t  0062                ldir                             ;non-8080 supported Z80 opcode

                       ;----------------------------------
                       ; Initialize Chip Control Registers
                       ;----------------------------------

   0062 3E 00          ld  a,#CMD_RESET                 ;issue 'reset' command
   0064 32 01 40       ld  (CREG),a

                       ;issue SCB1 parm byte = normal rows with VIDEO_COLUMNS characters
   0067 3E 4F          ld  a,#SCB1_NORMAL_ROWS | (VIDEO_COLUMNS - 1)
   0069 32 00 40       ld  (PREG),a
```

Power Issues

Another dilemma was that the 8080 uses 3 power supplies (+12, +5, -5) that have strict rules about how they transition at power up/down. I built up a small circuit that I could attach to the power supply rails to enforce these transitions (see below), but as it turned out it didn't seem necessary as the 8080 turned out to be more robust than advertised.



Design Issues

While I was waiting for the Rev B PCB to arrive, I got to writing the code for the 8275 video controller and ran into some problems.

First of all, the controller datasheet showed the chip being used with a DMA controller (which I didn't have), but I just figured I could get around it and write the data out using the CPU – albeit at a slower rate.

This was a bad call on my part. I had done this because I was used to using PIC processors to do things like this, but I forgot that the PIC parts are about 20x faster than the 8080 CPU!

After roughing out the code and assigning clock cycles to each instruction (see next page), it became apparent that this was not a practical solution.

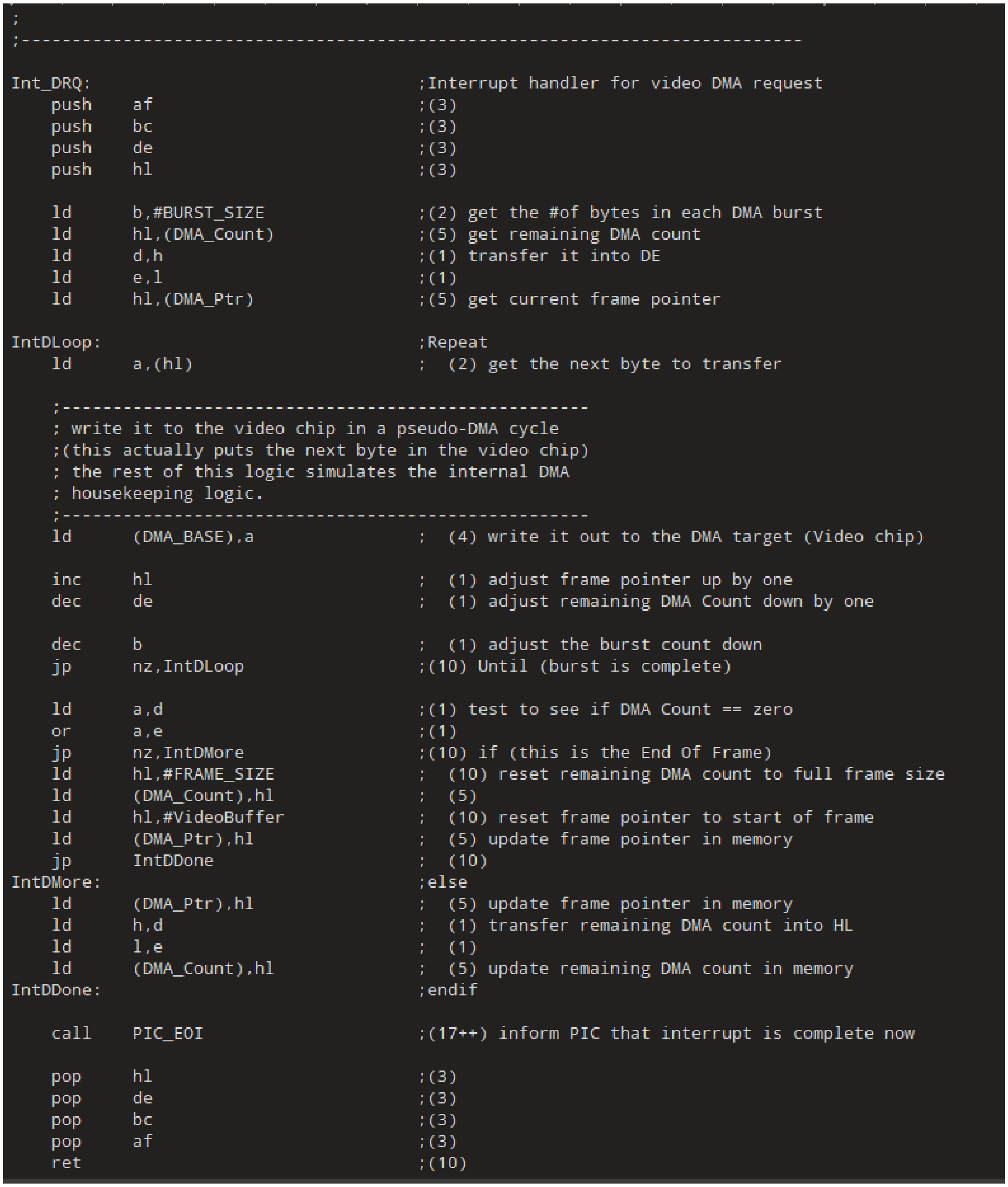
The math works out as follows (? = clock cycles):

> Entry Logic (26) + Exit Logic (44+) = 70+
> Each Loop (19)
>
> Therefore, to execute this logic on a burst of 8 characters requires approximately:
>
> > 70 + (8 * 19) = 222 clock cycles
>
> Or averaging the overhead over a single character time
>
> > 222 / 8 = 27.75 clock cycles or 13.88uS (using an 18MHz crystal / 9)
>
> Therefore, it will take the following amount of time to load a full screen (80x64=5120) of characters from memory into the video chip via this logic:
>
> > 13.88uS * 5120 = 71mS
>
> However, using a dot clock of 11.35MHz and a 7x8 pixel font results in the following frame draw time:
>
> > 5 * 7 *80 * 64 * (1 / 11.35MHz) = 15.79mS (approx 63 frames/second)
>
> So, it is clear that this logic can NOT keep up with the dot clock and will cause FIFO under-run errors at run-time.

I also tried modifying the logic to use the 8080 'HALT' instruction to reduce the entry/exit overhead. This was marginally faster, but not enough to solve the problem.

The only solution to this would be to slow down the dot clock by a factor of about 5 - this would result in the frame draw time going up to 78.94mS which would just barely be slow enough to work with this logic. However, the frame rate would drop to about 12 frames per second which is horribly slow.

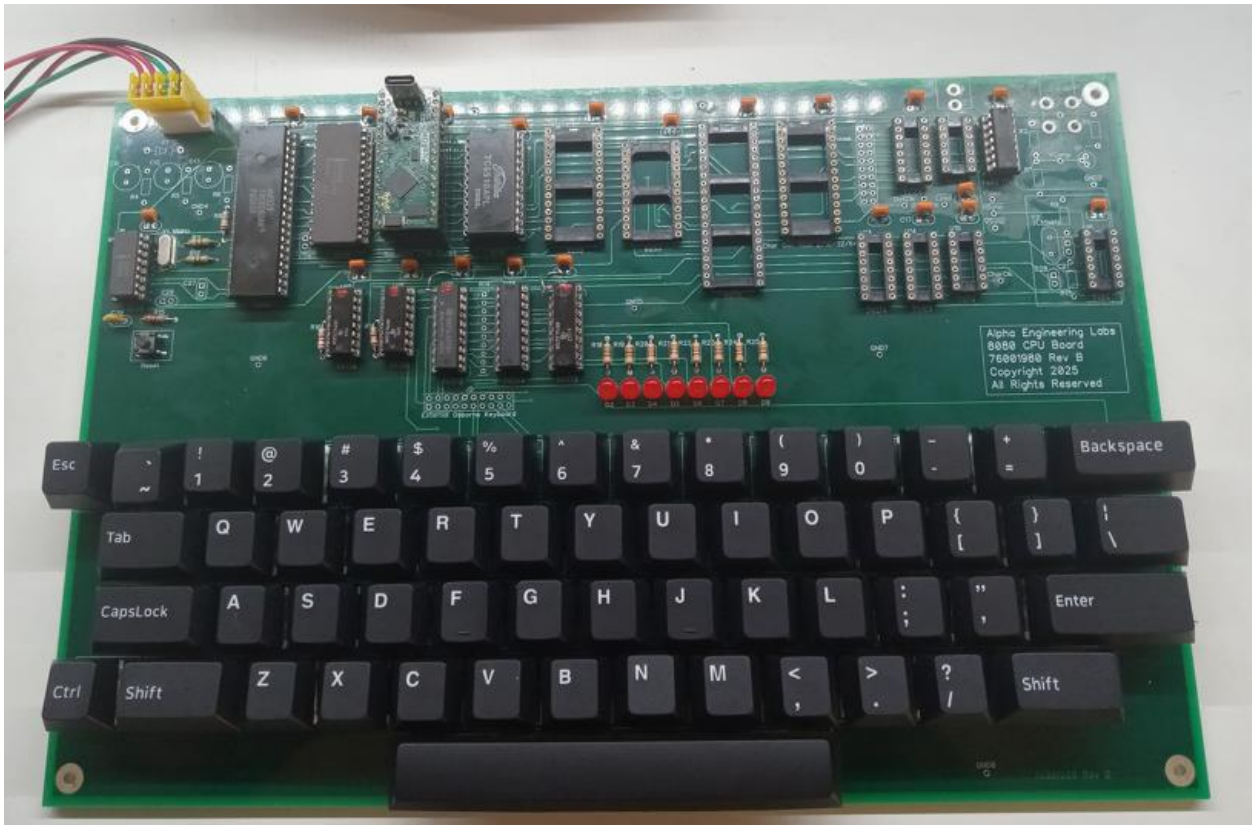
Still, a 12Hz frame rate would probably be OK for simple text. I ordered some 1.8432MHz crystals and I'll try it and see. Just in case, I was also able to find some Intel 8257 DMA and 8212 associated buffer chips so I could potentially re-layout the board again to incorporate a DMA channel if needed.



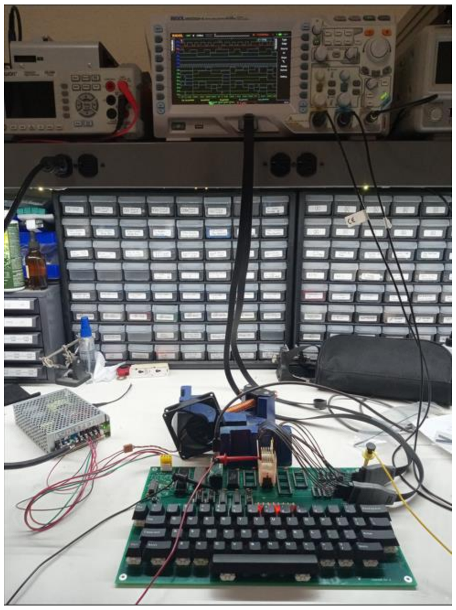
```
;
;------------------------------------------------------------------------------

Int_DRQ:                                ;Interrupt handler for video DMA request
    push    af                          ;(3)
    push    bc                          ;(3)
    push    de                          ;(3)
    push    hl                          ;(3)

    ld      b,#BURST_SIZE               ;(2) get the #of bytes in each DMA burst
    ld      hl,(DMA_Count)              ;(5) get remaining DMA count
    ld      d,h                         ;(1) transfer it into DE
    ld      e,l                         ;(1)
    ld      hl,(DMA_Ptr)                ;(5) get current frame pointer

IntDLoop:                               ;Repeat
    ld      a,(hl)                      ;  (2) get the next byte to transfer

    ;-----------------------------------------------------
    ; write it to the video chip in a pseudo-DMA cycle
    ;(this actually puts the next byte in the video chip)
    ; the rest of this logic simulates the internal DMA
    ; housekeeping logic.
    ;-----------------------------------------------------
    ld      (DMA_BASE),a                ;  (4) write it out to the DMA target (Video chip)

    inc     hl                          ;  (1) adjust frame pointer up by one
    dec     de                          ;  (1) adjust remaining DMA Count down by one

    dec     b                           ;  (1) adjust the burst count down
    jp      nz,IntDLoop                 ;(10) Until (burst is complete)

    ld      a,d                         ;(1) test to see if DMA Count == zero
    or      a,e                         ;(1)
    jp      nz,IntDMore                 ;(10) if (this is the End Of Frame)
    ld      hl,#FRAME_SIZE              ;  (10) reset remaining DMA count to full frame size
    ld      (DMA_Count),hl              ;  (5)
    ld      hl,#VideoBuffer             ;  (10) reset frame pointer to start of frame
    ld      (DMA_Ptr),hl                ;  (5) update frame pointer in memory
    jp      IntDDone                    ;  (10)
IntDMore:                               ;else
    ld      (DMA_Ptr),hl                ;  (5) update frame pointer in memory
    ld      h,d                         ;  (1) transfer remaining DMA count into HL
    ld      l,e                         ;  (1)
    ld      (DMA_Count),hl              ;  (5) update remaining DMA count in memory
IntDDone:                               ;endif

    call    PIC_EOI                     ;(17++) inform PIC that interrupt is complete now

    pop     hl                          ;(3)
    pop     de                          ;(3)
    pop     bc                          ;(3)
    pop     af                          ;(3)
    ret                                 ;(10)
```

Rev B PCB

When I finally got the rev B PCB (it took longer than usual – I think NEXT PCB dropped the ball on this one) I started putting it together and ran into more issues. The good news, of course, was that the keys now fit in the keyboard!



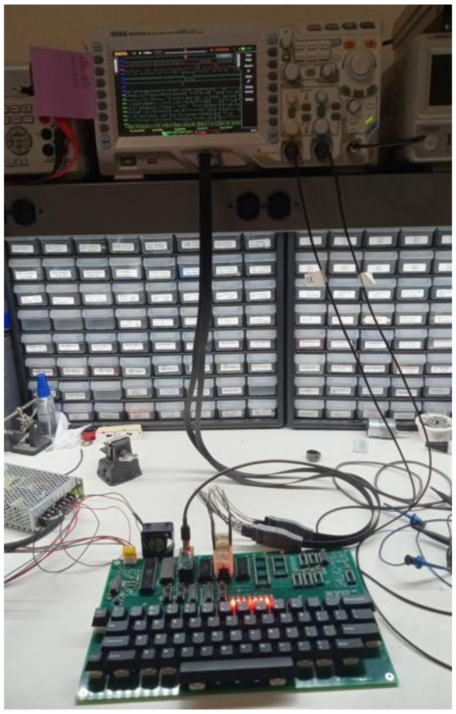
8253 PIT Timer



After spending a couple of hours, I was able to get the PIT timer chip working. I realized that I'd forgotten to add pull-ups on the three gate lines (none of the timers would work without them.)

Additionally, the code that I had roughed out didn't take into account a few details.

8259 Interrupt Controller

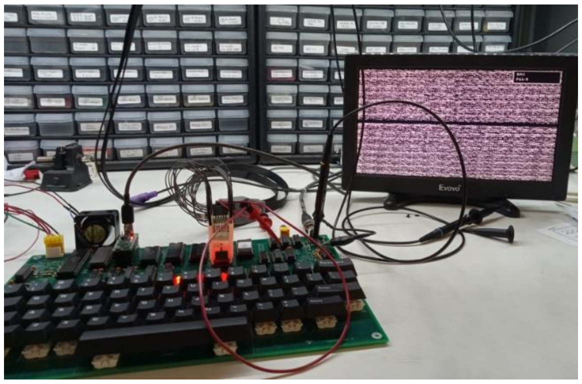


The interrupt controller was a little more complicated.

This was mostly due to my not paying attention to the fact that the interrupt vector table needed to be aligned on a 32/64 byte boundary (depending on whether the vectors were 4 or 8 byte long.

The other issue I had was that I was looking for the RETI (return from interrupt) instruction, but this is not supported in the 8080. I used the RET instruction, but it took me a while to remember that you have to manually enable interrupts (EI) before returning – so the interrupts were just hanging for a while.

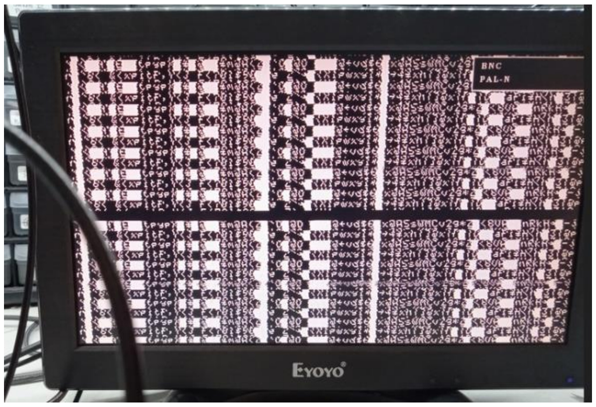
8275 Video Controller



This turned out to be where I would spend the most time.

After initially populating the circuitry for the video, I could see nothing but a blank screen. After some trouble-shooting, I realized that the logic would blank the video if there were any problems with the video stream. Once I found this, I was able to jumper the VSP line so that I could always see video. Now I could see a sloppy mess on the screen.

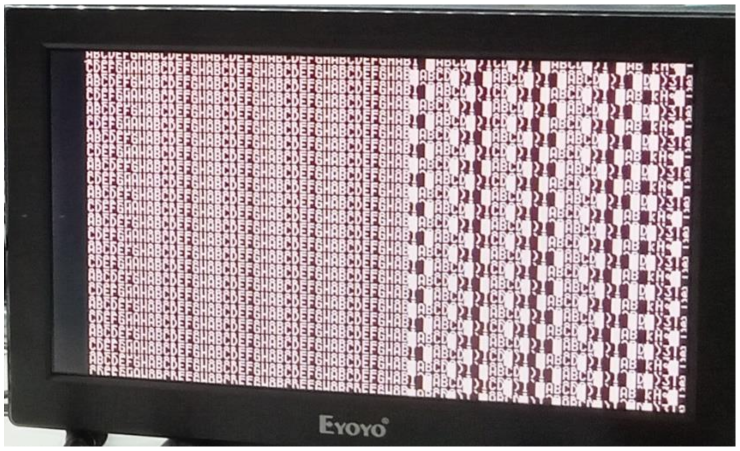
After spending quite some time with a paper, pencil, the video chip data sheet, and a calculator, I was able to get the chip configuration setup close enough so that the monitor was able to sync and recognize the pixel data.



However, the data presented was total garbage – I had filled the display buffer with a repeating alphabet string.

It should be noted that I was attempting to transfer the data to the video chip using only normal 8080 instructions – no DMA hardware. My initial thought was that this would be adequate – even if it made the main application slower – because, hey, I'm just doing this for fun and don't have a real application.

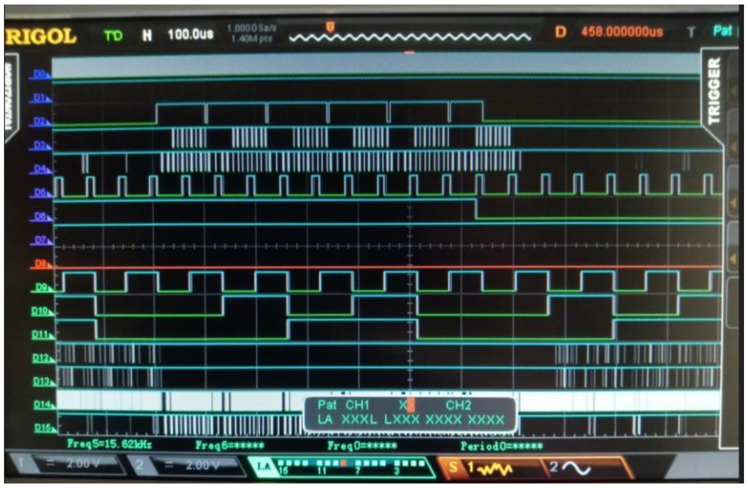
After hours of effort, I was able to get my software pseudo-DMA transfers to start working. You can see the alphabet string starting to show up on the left side of the screen. I was pretty happy at this point because I was finally able to get something to display.



After calming down a bit, I started to look into “why” only the first half of the line was being displayed on the left, and the right was still displaying garbage.

This was probably the most frustrating time of the project because I really wanted the Rev B PCB to work – even if I had to turn the resolution way down to do so.

I tried various tricks, but in the end (as you can see below) I was forced to realize that the reason only the left side of the screen had valid data was because the pseudo-DMA logic was too slow and was only able to push out about 42 characters of data before the end of vertical refresh.



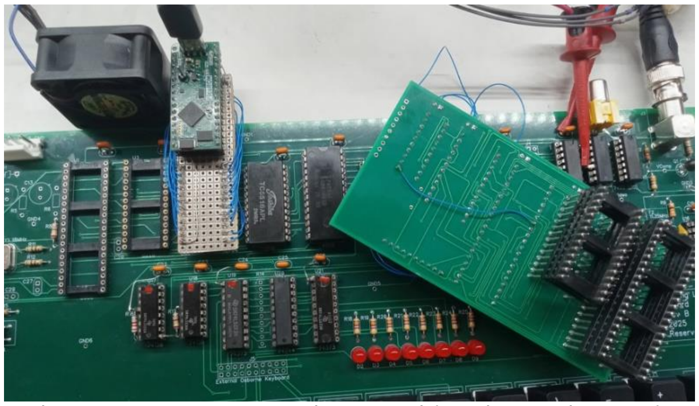
Because of this, the video chip aborted the DMA transfer (because it didn’t have enough data to display even one row.)

Looking at the pseudo-DMA logic, there was absolutely no way to speed it up enough to overcome this issue.

At this point I decided to design a DMA daughter board. This was cheaper and quicker than re-designing the entire PCB to add a DMA circuit, and I didn’t want to spend hours re-soldering all the parts on another board if I could avoid it.

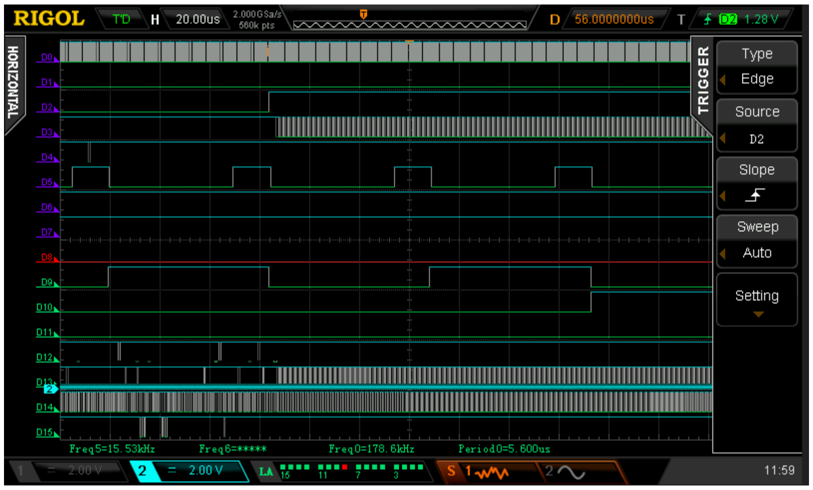
76002080 DMA Daughter Board

I received and built the DMA daughter board (see below left.) Unfortunately, I didn't think about all the



mechanical issues and found out the hard way that the PicoROM ROM emulator could no longer be used with the daughter board. I ended up having to hand build a small adapter board that would shift the EPROM socket out from underneath the daughter board so that the ROM emulator could still be used.

The DMA circuit was MUCH faster than the 8080 logic I had been using. You can see in the image below

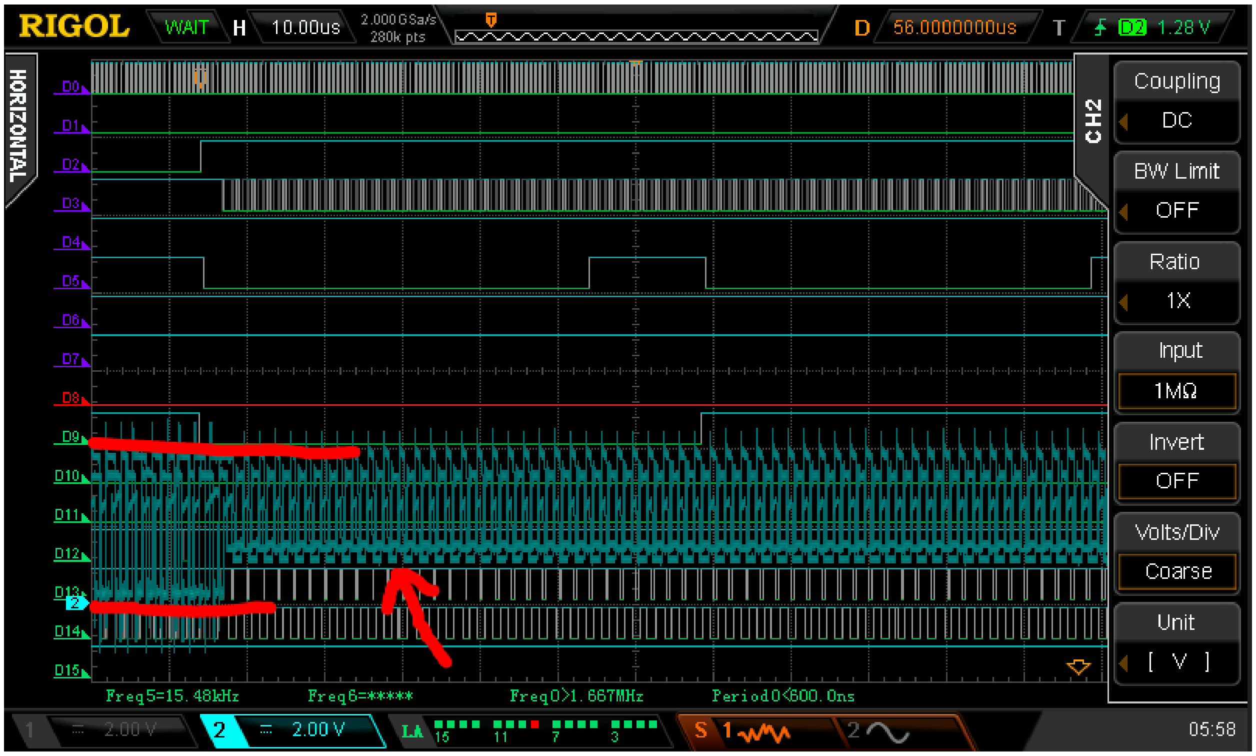


that the DACK pulses are running 10-20 times faster than the software pseudo-DMA logic I had been using.

Another mistake I had made was to map the video chip into memory space. While this worked fine for my software pseudo-DMA logic, it failed completely when using a real DMA. The reason is simple enough if you think about it – the DMA transfers data to/from memory/IO in a single cycle. It can do this because it uses the DACK pin as a substitute for having a second set of target memory addresses. I had to re-work the board for this but it wasn't too hard.

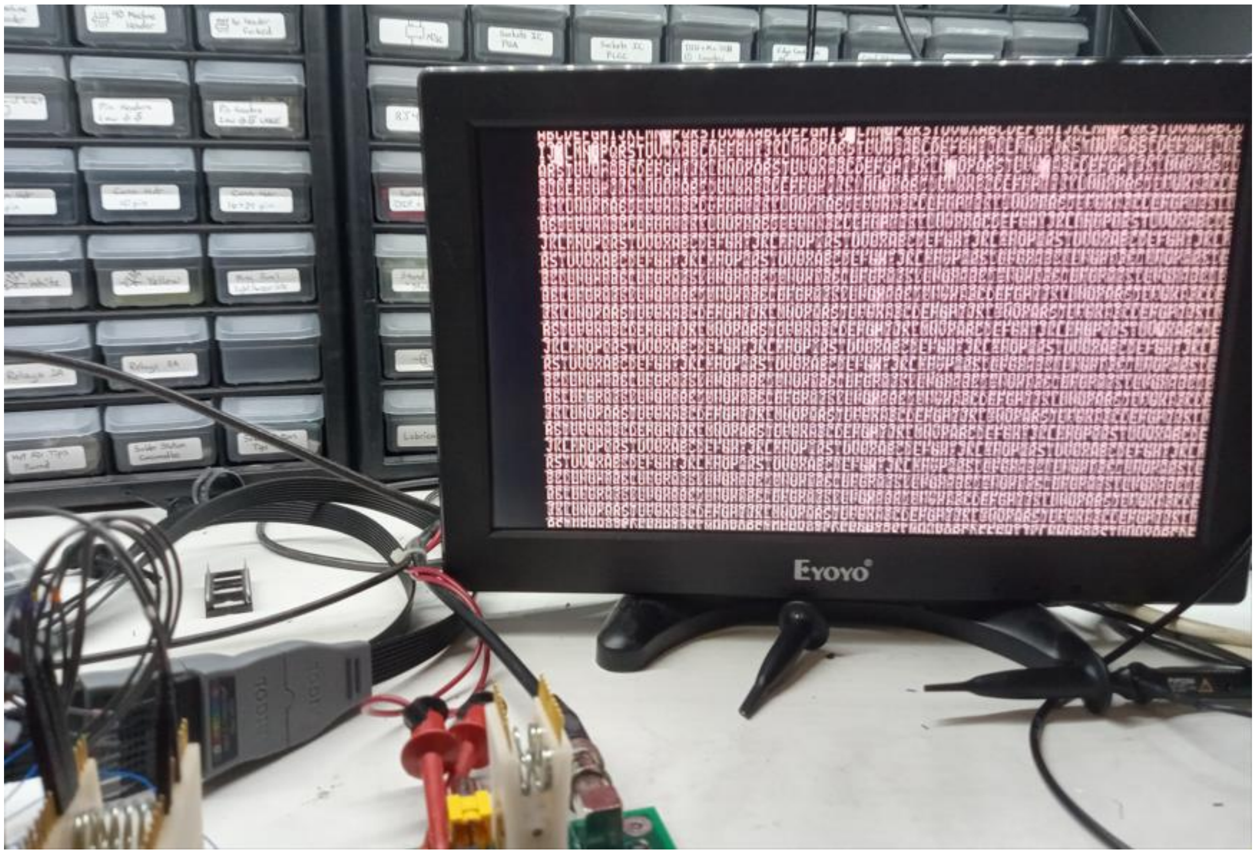
Additionally, my foresight of putting a 16L8 PAL on the daughter board really paid off in being able to easily customize the chip select equations. This really paid off later when I realized that the I/O chip selects have to be disabled while the DMA has control of the bus, otherwise random DMA memory address will be interpreted as I/O addresses and trigger the appropriate chip selects. With the PAL this was easy to fix by qualifying the chip selects with the 8080 HLDA (hold ack) signal inactive.

Another problem I encountered really makes no sense to me. I was seeing the RD/WR control lines not going to rails – like there was bus contention during the DMA cycles. This turned out to be a requirement that you connect the 8080 HLDA (hold ack) line to the 8228 system controller BUSEN (bus enable) so that the 8228 would tri-state the bus control lines. The confusing part for me, still, is that HLDA is already connected to the 8228 – so why doesn't it do this automatically? I thought for a moment that it could be for cases when the DMA was positioned between the 8080 and 8228 (and thus was using the 8228 lines) but that didn't make sense either because the DMA chip generates its own I/OR, I/OW, MEMR, MEMW lines. So, I'm still confused, but it works now.



I spent a bunch of time after this trying to figure out why the data being displayed was all random garbage. This ended up being a stupid error where I forgot to increment the HL pointer while the RAM video buffer was being initialized – resulting in only the first byte being initialized.

Ultimately, I was able to get text displayed on the monitor so it was a success!



8257 DMA Controller

I had been dealing with a considerable amount of instability in getting the DMA transfers working. At first (for reasons unknown), the issues were somewhat minor and I could reset the board a couple of times to get it working. But later, the problem (why, I still don't know) became insanely annoying because I couldn't get the DMA transfers to work at all no matter how many resets or power cycles I might try.

This was so frustrating that I finally had to abandon the project and watch a movie and get some sleep.

In the morning, I realized what might be causing it. I remembered reading in the 8275 spec that the 16-bit registers were accessed by a single port address, and used an internal flip flop to determine if the write should go to the LSB or MSB of the register. If this flip flop got out of sync, the values written to the DMA registers would be reversed causing all sorts of havoc

In order to prevent this, the specs had recommended that interrupts be disabled while accessing these registers. So, I had written a subroutine to setup the DMA transfer, that was called both by the hardware initialization logic and the video interrupt.

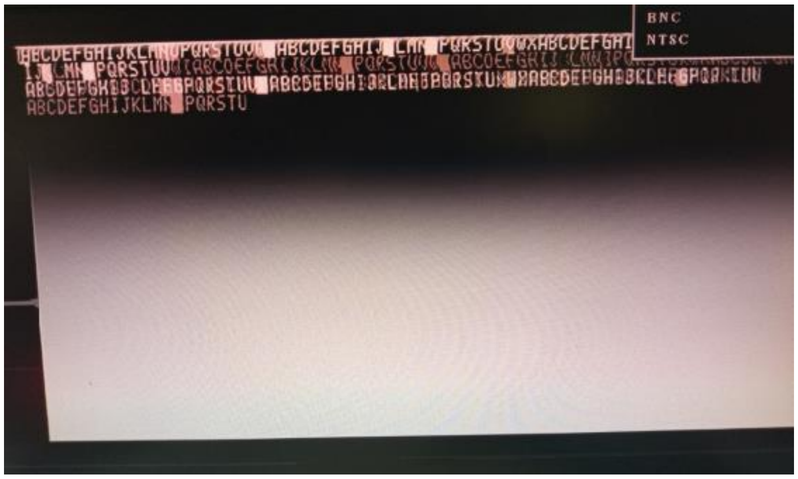
The problem (you probably see it coming) is that calling this routine from the power up logic, will enable interrupts BEFORE all of the hardware is properly setup (because this subroutine is called early in the setup process.) This caused interrupts to start operating BEFORE everything was properly setup.

The fix was simple – don't change the interrupt enable state within the subroutine – because it was already ONLY called from two spots both of which already had interrupts disabled.

DMA RAM Issues

I had been having constant issues where the data displayed on the screen was not always what I expected. After considerable debugging, I was able to isolate this down to RAM access issues.

Below you can see sample video output. As can be seen, several characters are wrong, and the display is terminated prematurely on line 4 – there should be 24 lines of alphabet filling the entire screen.



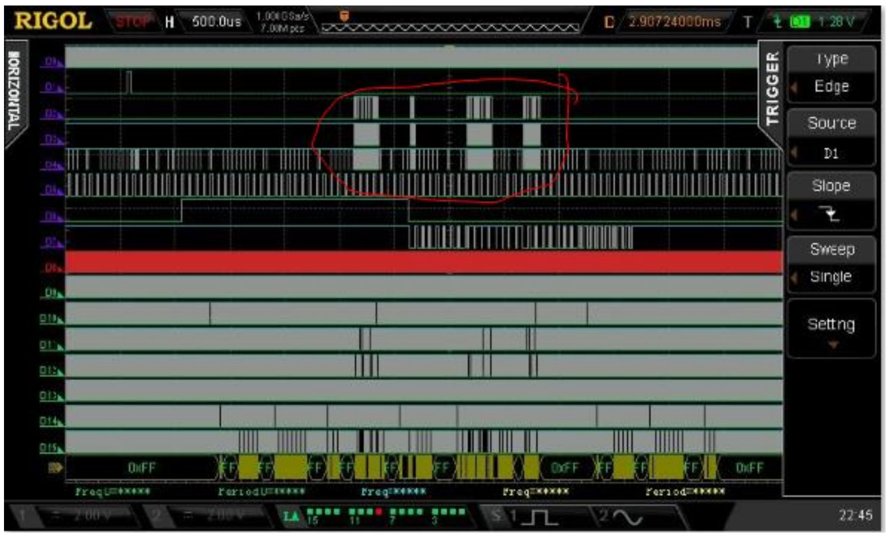
DMA Underrun Check

My initial thought was that this was caused by a DMA underrun condition, which according to the spec would cause the DMA to abort, and the video to be blanked, until the end of the current frame.

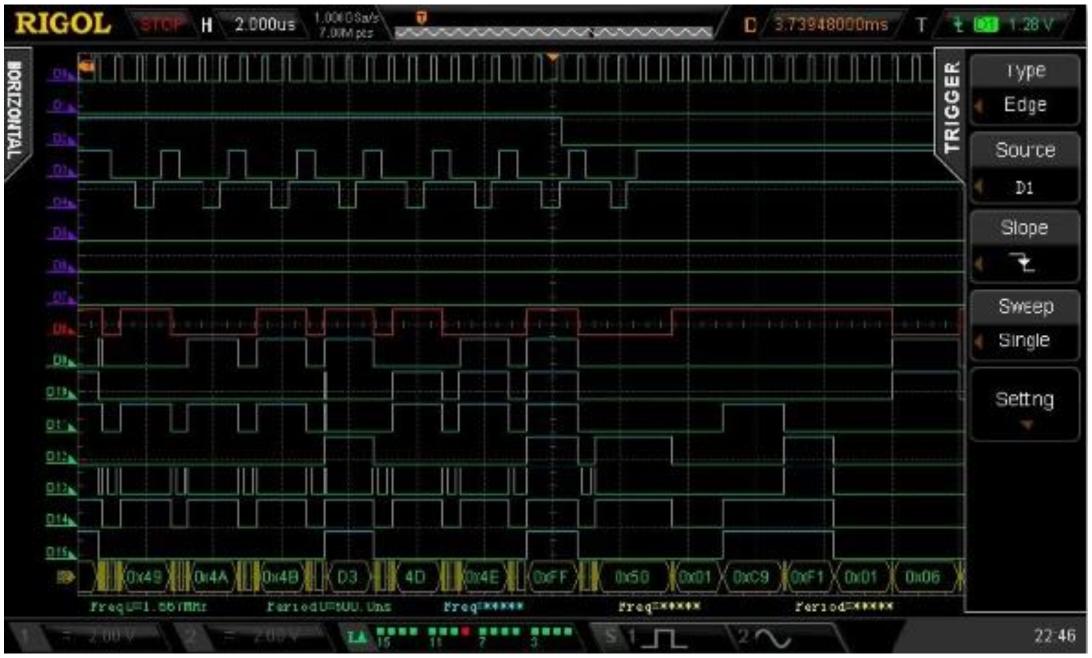
However, after writing some code to be able to trap this condition and display it on the LEDs, I was able to ascertain that there was a DMA underrun, but only a single occurrence at power up, and not repeated cases in the middle of frames to cause what I was seeing.

Video Chip "Special Code" Issue

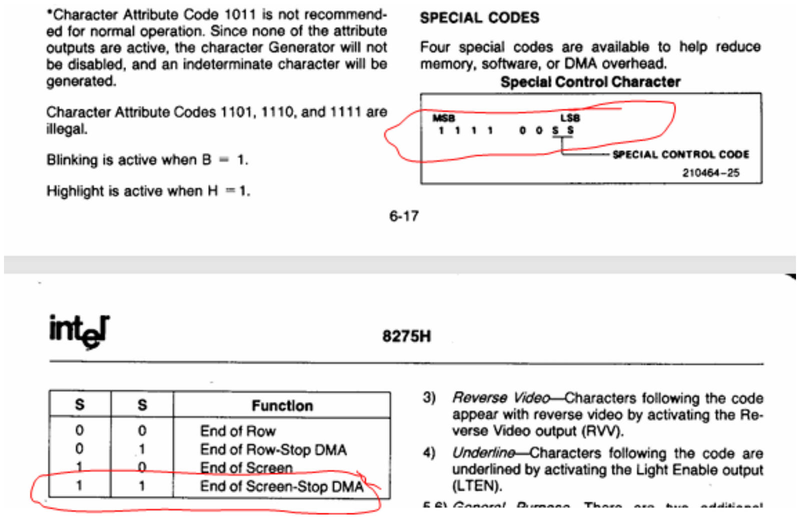
I then spent more time debugging the issue and was able to capture the video chip signals for the screen shot shown on the page above.



If you zoom into the final DMA transfers you will see the following.



At the bottom you can see the actual data transferred by the DMA as: 0x49, 0x4a, 0x4b, 0xd3, 0x4d, 0x4e, and 0xff. Referring to the 8275 video chip spec, you will see that 0xff (technically 0xf3 – but apparently d2-d3 are don't care bits) is a special code that is interpreted to end the screen display and abort the DMA transfer.



*Character Attribute Code 1011 is not recommended for normal operation. Since none of the attribute outputs are active, the character Generator will not be disabled, and an indeterminate character will be generated.

Character Attribute Codes 1101, 1110, and 1111 are illegal.

Blinking is active when B = 1.

Highlight is active when H = 1.

SPECIAL CODES

Four special codes are available to help reduce memory, software, or DMA overhead.

Special Control Character

```
MSB              LSB
 1  1  1  1   0  0  S  S
                     └── SPECIAL CONTROL CODE
                                 210464–25
```

6-17

intel **8275H**

S	S	Function
0	0	End of Row
0	1	End of Row-Stop DMA
1	0	End of Screen
1	1	End of Screen-Stop DMA

3) *Reverse Video*—Characters following the code appear with reverse video by activating the Reverse Video output (RVV).

4) *Underline*—Characters following the code are underlined by activating the Light Enable output (LTEN).

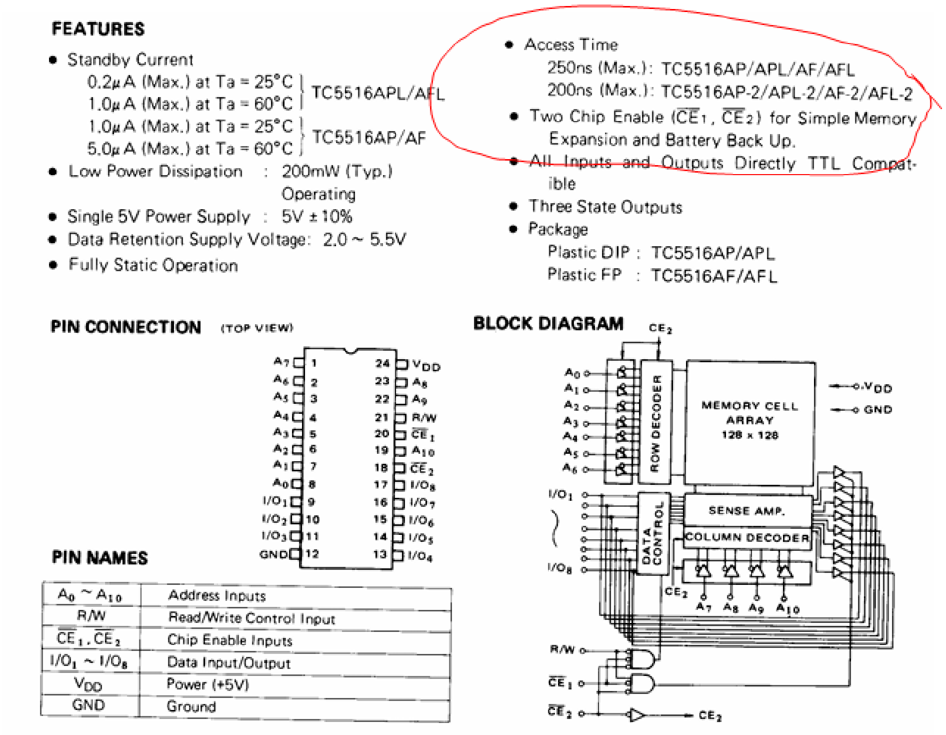
5,6) *General Purpose*—There are two additional

So, it is not that the DMA is failing in some way, but rather that the DMA transfer is being intentionally aborted by the 8275 video chip.

The problem would appear that the RAM chip cannot always keep up with the DMA transfer timing, and that wait states (or some other solution) are required.

Speeding Up RAM Access

I started looking into this and noticed that the RAM was an older Toshiba part that was designed for extremely low power retention and has two chip selects.



FEATURES

- Standby Current
  - 0.2μA (Max.) at Ta = 25°C, 1.0μA (Max.) at Ta = 60°C — TC5516APL/AFL
  - 1.0μA (Max.) at Ta = 25°C, 5.0μA (Max.) at Ta = 60°C — TC5516AP/AF
- Low Power Dissipation : 200mW (Typ.) Operating
- Single 5V Power Supply : 5V ± 10%
- Data Retention Supply Voltage: 2.0 ~ 5.5V
- Fully Static Operation
- Access Time
  - 250ns (Max.): TC5516AP/APL/AF/AFL
  - 200ns (Max.): TC5516AP-2/APL-2/AF-2/AFL-2
- Two Chip Enable ($\overline{CE}_1$, $\overline{CE}_2$) for Simple Memory Expansion and Battery Back Up.
- All Inputs and Outputs Directly TTL Compatible
- Three State Outputs
- Package
  - Plastic DIP : TC5516AP/APL
  - Plastic FP : TC5516AF/AFL

PIN CONNECTION (TOP VIEW)

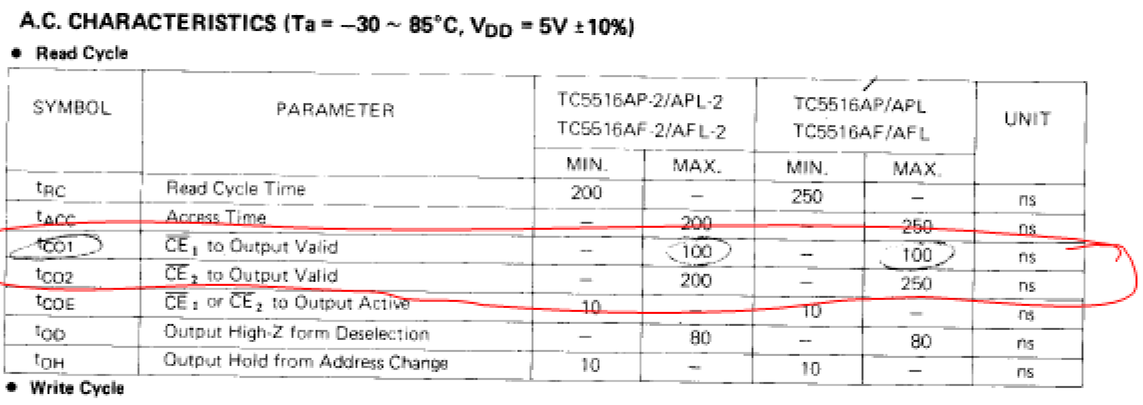
A_7	1	24	V_{DD}
A_6	2	23	A_8
A_5	3	22	A_9
A_4	4	21	R/W
A_3	5	20	$\overline{CE}_1$
A_2	6	19	A_{10}
A_1	7	18	$\overline{CE}_2$
A_0	8	17	I/O_8
I/O_1	9	16	I/O_7
I/O_2	10	15	I/O_6
I/O_3	11	14	I/O_5
GND	12	13	I/O_4

PIN NAMES

$A_0 \sim A_{10}$	Address Inputs
R/W	Read/Write Control Input
$\overline{CE}_1$, $\overline{CE}_2$	Chip Enable Inputs
$I/O_1 \sim I/O_8$	Data Input/Output
V_{DD}	Power (+5V)
GND	Ground

BLOCK DIAGRAM

CE₂
A₀
A₁
A₂
A₃
A₄
A₅
A₆
ROW DECODER
MEMORY CELL ARRAY 128 x 128
VDD
GND
I/O₁
I/O₈
DATA CONTROL
SENSE AMP.
COLUMN DECODER
CE₂
A₇
A₈
A₉
A₁₀
R/W
CE₁
CE₂
CE₂



A.C. CHARACTERISTICS (Ta = –30 ~ 85°C, V_{DD} = 5V ±10%)

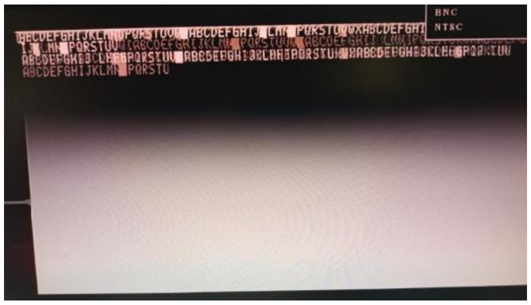
- **Read Cycle**

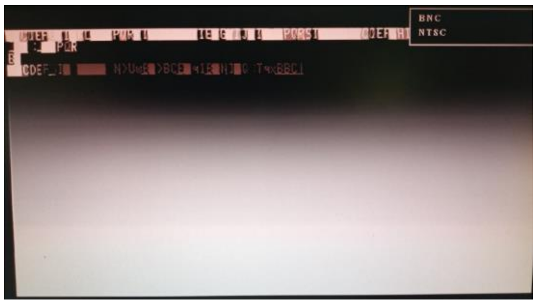
SYMBOL	PARAMETER	TC5516AP-2/APL-2 TC5516AF-2/AFL-2		TC5516AP/APL TC5516AF/AFL		UNIT
		MIN.	MAX.	MIN.	MAX.	
t_{RC}	Read Cycle Time	200	–	250	–	ns
t_{ACC}	Access Time	–	200	–	250	ns
t_{CO1}	$\overline{CE}_1$ to Output Valid	–	100	–	100	ns
t_{CO2}	$\overline{CE}_2$ to Output Valid	–	200	–	250	ns
t_{COE}	$\overline{CE}_1$ or $\overline{CE}_2$ to Output Active	10	–	10	–	ns
t_{OD}	Output High-Z form Deselection	–	80	–	80	ns
t_{OH}	Output Hold from Address Change	10	–	10	–	ns

- **Write Cycle**

I had initially tied the chip selects together and got the results on the left (below) which were mostly correct and only had a few incorrect characters. After looking at the RAM specs closer, it appears that using only CS1 reduces the access time from 200nS -> 100nS.

So, I tried disconnecting CS2 from the chip select logic and tying it to ground. You can see the results below. On the left is the result of both chip selects tied together (with most correct characters), and the right shows the result of CS2 tied to ground (with mostly wrong characters.)

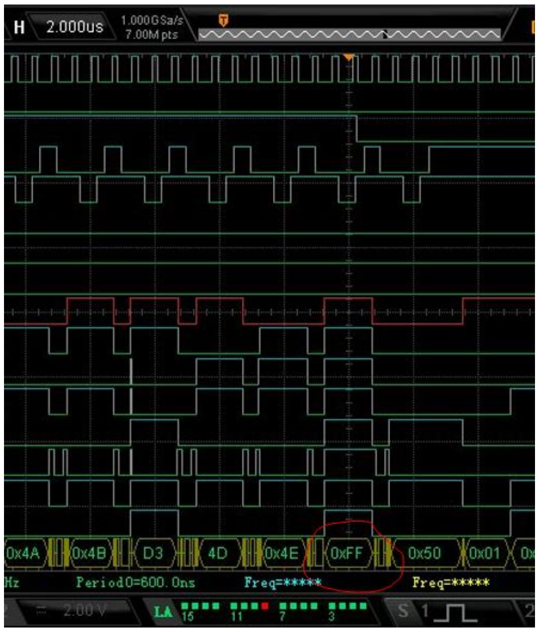


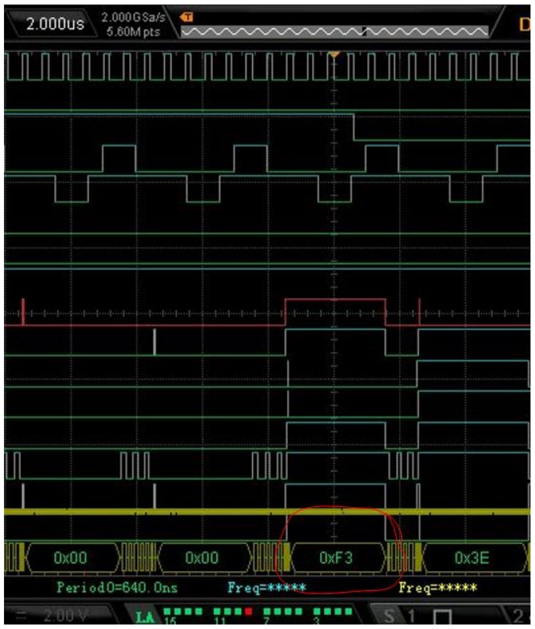


This seems to contradict the spec, as I would have expected the results to be “better” in the right image (not worse.) But it does prove that altering the RAM access times does indeed have an effect on the DMA data transferred.

Reducing DMA Clock to Widen Read Window

I then changed the DMA to use the 2MHz system clock / 2 (by running it through the 8253 Timer chip.) Interestingly, even though this relaxed the memory read timing requirements by 100%, the exact same issue still exists. See below left for issue with 2MHz DMA clock, and right for issue with 1MHz DMA clock. Since doubling the read time didn't fix it, it doesn't appear to be a simple read timing issue.





Pattern Recognition

After looking at so many logic analyzer captures of the problem, I started to notice a pattern.

It appears that, in most cases when the problem occurs, that the data is not the 'A', 'B', 'C'… data that should be there, but rather looks more like 8080 assembler binary – as though it were retrieved from the EPROM instead of the RAM by the DMA.

Take the following hex data (which occurred on the first line after VSync) for example:

> 00 00 C9 00 00 00 00 00 2F D3 00 00 C9 00 4D 4E 4F 00 …

'C9' followed by 7x '00' could be part of the interrupt vector table. 'C9' is the 8080 'RET' opcode, and the vectors are aligned on 8 byte boundaries.

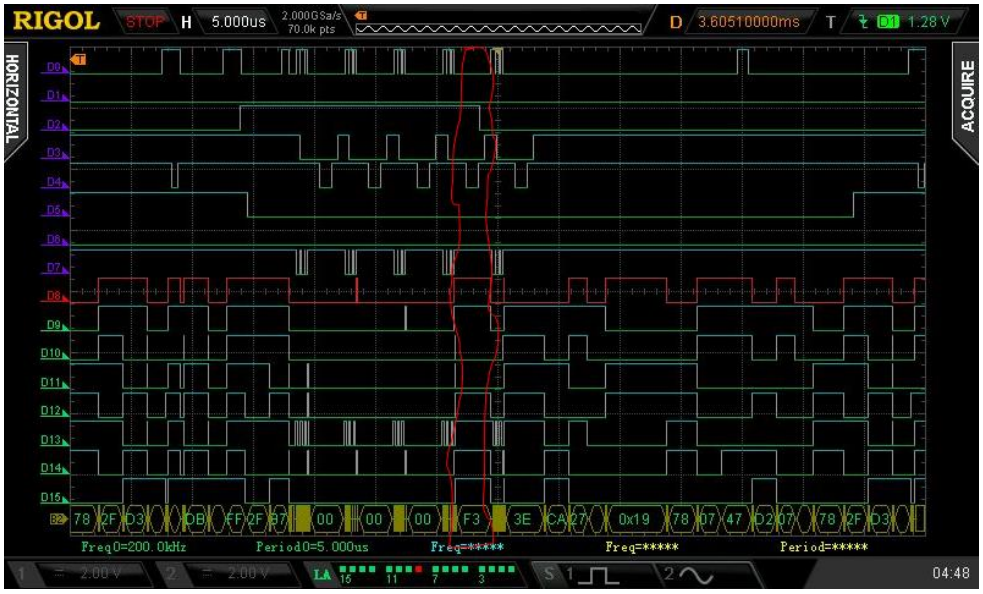
'2F' is the 'CPL' opcode, and 'D3 00' is the 'OUT (0x00),a' opcode both of which are used in the DMA logic (port 0x00 is the base of the DMA chip.)

The 8080 opcodes that would trigger the 1111xx11b code (which is causing the video chip to abort the frame and DMA transfer) are as follows:

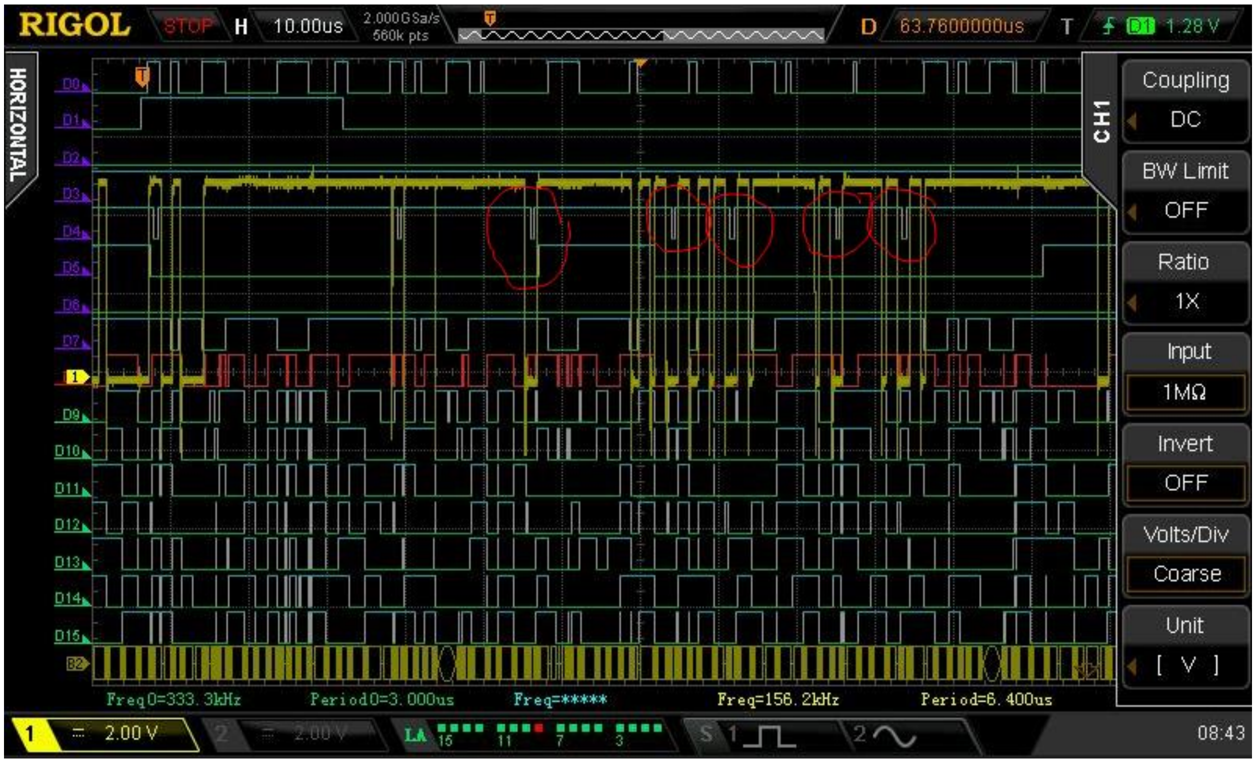
- 0xf3 – DI opcode
- 0xf7 – RST 6 opcode
- 0xfb – EI opcode
- 0xff – RST 7 opcode (AND un-programmed EPROM memory)

'DI', 'EI', and un-programmed memory all exist in the ROM image so it is conceivable that the DMA might be erroneously transferring data from this area of memory instead of the RAM as it should be.

After altering the logic analyzer to include ROM CS on D0 and RAM CS on D7 it can plainly be seen that this is the case – the DMA is transferring from ROM for some reason.



I was able to verify that the video interrupt logic is working properly (see below – yellow scope line is DMA chip select and red circled pulses are IOWR active with DMA CS.) You will have to take my word but if you expand the view, you will see that the data written out in these 5 writes is: 0x01, 0x06, 0x20, 0x7f, 0x87.



This sequence should:

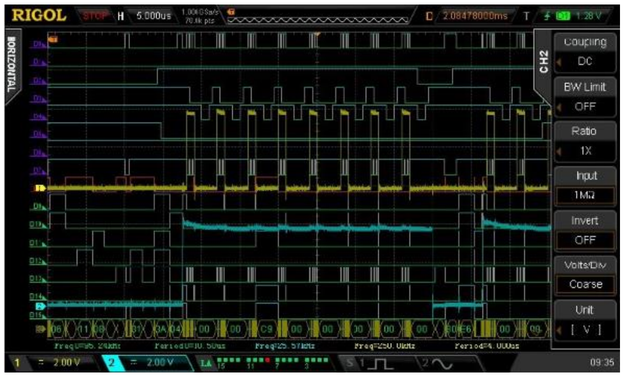
- Enable DMA 0
- Set DMA Address register to 0x2006 (VideoBuffer in RAM)
- Set DMA Count register to 0x0780 (80 x 24), and set transfer mode to 'Read' – memory -> IO

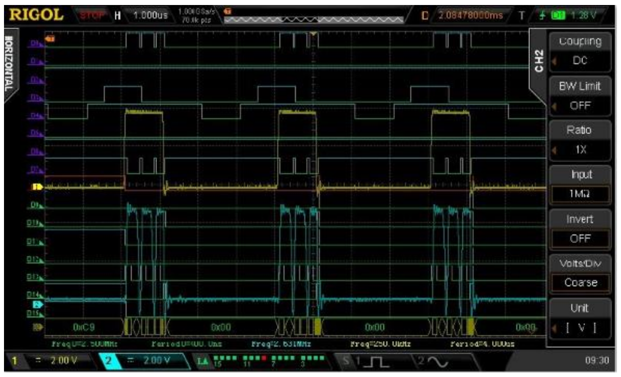
And since the first write (0x01) is to the ModeSet register, the F/L flip flop should be reset and guaranteed to be in the correct state.

So the issue does not appear to be due to improper DMA setup.

So, the only way this can be happening is if the DMA is actually issuing addresses in the range of 0x0000-0x1fff (ROM.)

I was able to capture states of the DMA and 8212 I/O Port below. The image on the left shows an 8-byte DMA transfer with DMA ADSTB (yellow) and AEN (blue) on top of the logic analyzer data. The image on the right shows the same – zoomed in a bit - but with DMA ADSTB (yellow) and A13 (blue.)

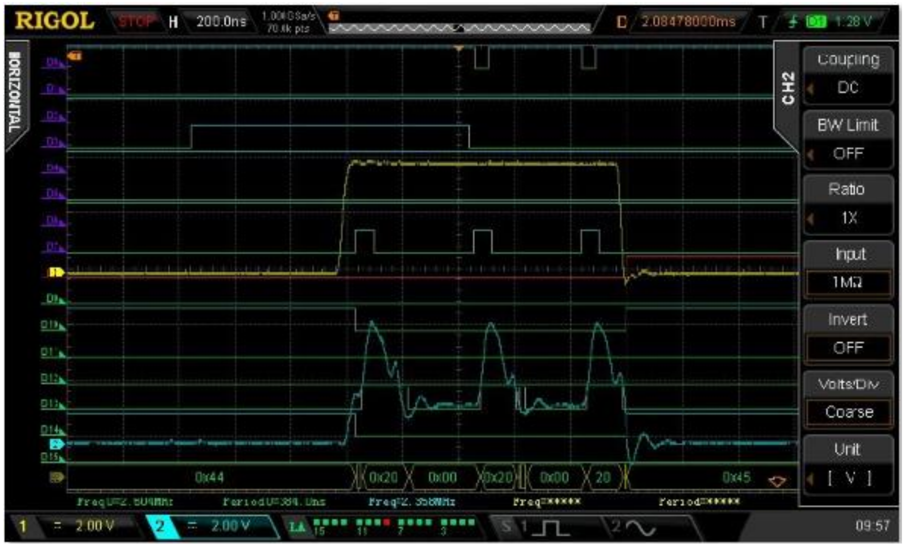




The right image clearly shows A13 low while D3 (DMA DACK) and D4 (IOWR) are both low. This explains why the DMA data is coming from the ROM and not the RAM.

You can see on the right that A13 pulsed high 3 times during ADSTB, but was not actually latched onto the address bus.

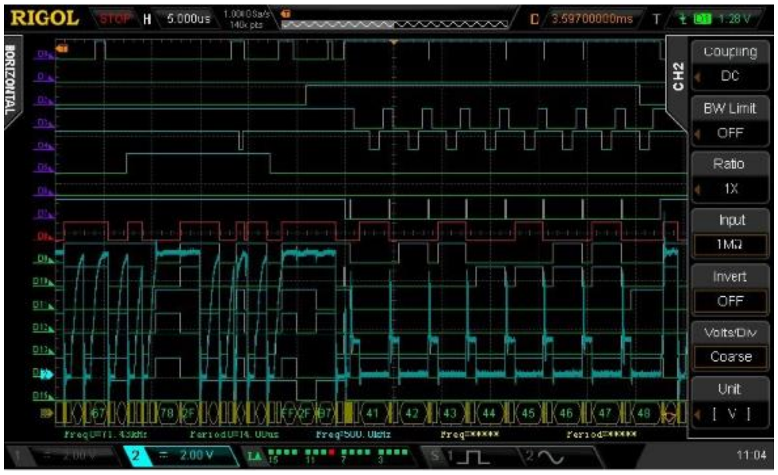
I then captured the same region, except this time instead of looking at the output of the 8212 for A13, I looked at the input – which is the D5 data line which you can see below in blue.



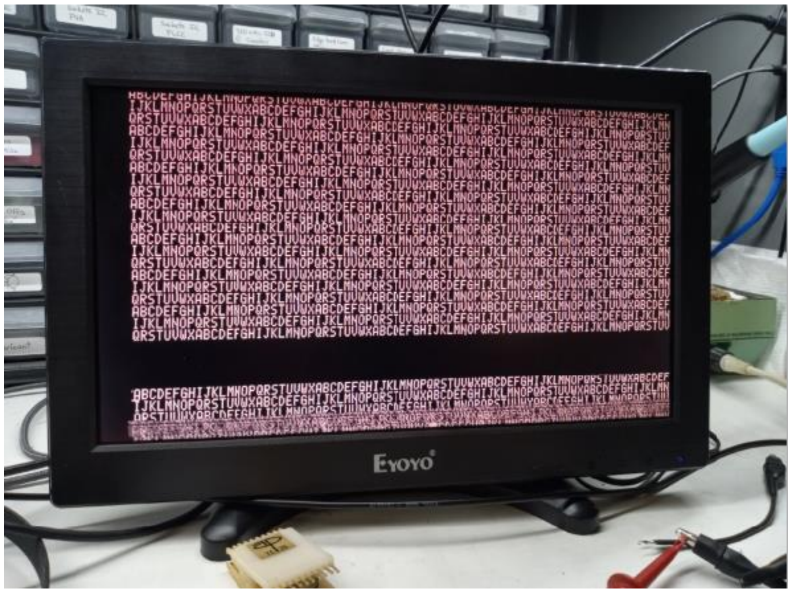
The D5 data line looks horrible and has a very slow rise time compared to the ADSTB above it. It is for this reason that the latch is latching a value of '0' for A13, instead of '1' which would be registered if the D5 data line had a better rise time and cleaner signal.

Initial "Solution" to Problem

By simply adding a 2.2K pull-up to the D5 data line you can see below that the signal is cleaned up, the address decoders are now seeing this as a RAM access (not a ROM access) and the values retrieved on the data bus are now as expected in the alphabet range 0x41 ('A'), 0x42 ('B') …



You can now see below that, not only is the monitor displaying all of the correct characters, but additionally all of the jitter that used to be present is gone and the display looks rock solid.



Footnote: as I was initially adding the pull-up to the circuit, the entire video circuit up and died! There was no video, no DMA requests, and I thought I heard a pop like a sudden short blowing something out. This was so annoying when I thought I was so close to finding the problem, that I almost threw the whole thing in the garbage out of frustration. It took me about an hour to track down that one of the 7474N chips (from the 1970's) had stopped working. This in turn disabled the character clock and prevented the video chip from functioning. After I replaced the chip, things started working again.

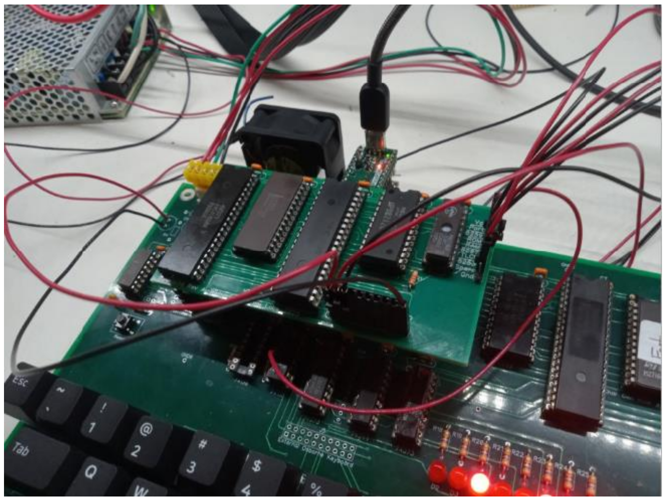
Real Solution to Problem

To be honest, the “Initial” solution presented above was really a band-aid fix. The solution didn’t explain why the problem existed, and only solved the problem when DMA transfers originated from RAM addresses in the 0x2xxx range. The hardware was becoming more and more unstable, as I continued to track down the real issue, to the point where it got so frustrating that I had to put it away and take a couple of days off.

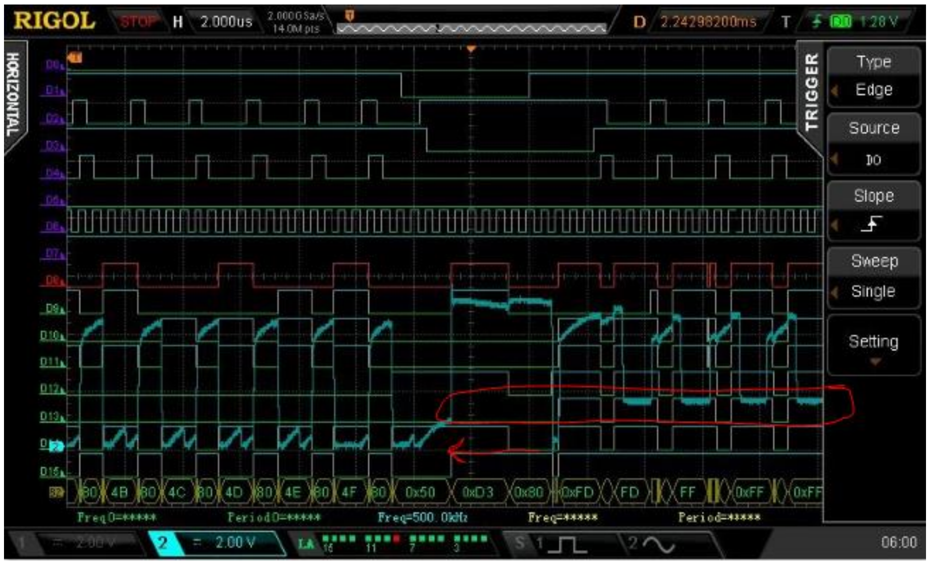
After I came back to the problem, my first job was to clean up the hardware. It took a whole day and I changed the following:

1. Daughter board to main board jumper wires converted from soldered wire wrap wires to 24awg wire with proper connectors so daughter board could be easily removed.
2. Added power and ground wires from main board to daughter board to improve power rails – which were only coming up through the board stand-off pins of the sockets before.
3. Modified both PCB to convert dual 74LS138 address decoding to use PAL16L8 for all critical chips – remaining 74LS138 only decoded LED and keyboard latches. This moved the 8253 and 8259 chips from memory to I/O space and only the ROM and RAM were in memory space now.
    a. This gave me much more flexibility to modify chip select qualifiers.
    b. I ended up adding AEN (active during DMA xfers when DMA has bus) to disable non-RAM chip selects from floating bus conditions, among other things.
4. Updated all prototype changes into new 760019xx schematic and PCB layouts so I am ready to get new boards made when needed.
5. Ordered more DIP socket clips for future so it is easier to do tests on multiple chips.

This DRASTICALLY improved the board reliability and reduced my frustration level. See improved board stability below.

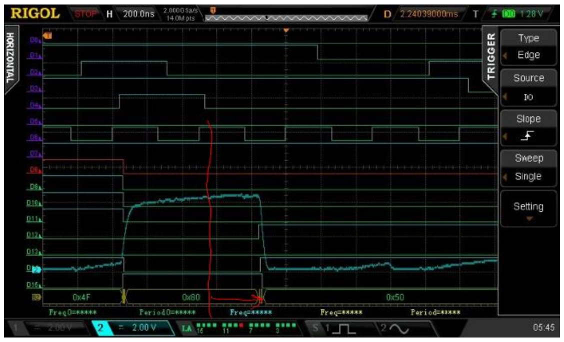


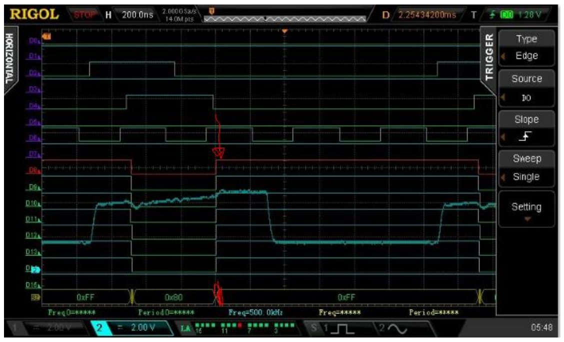
Although it took several hours more of debugging, I finally narrowed it down to the following:



The left side of the screen shot shows correct operation (see 4B, 4C 4D… in bottom showing valid ASCII alphabet characters) vs the right side showing the error state (see FD, FF… characters showing abort DMA xfer commands.)

I had seen this for some time, but I was preoccupied trying to see what was different about the two cases. One thing that seemed likely was that the data was held longer after ASTB (D4) in the working case (left below) than the failed case (right below.)

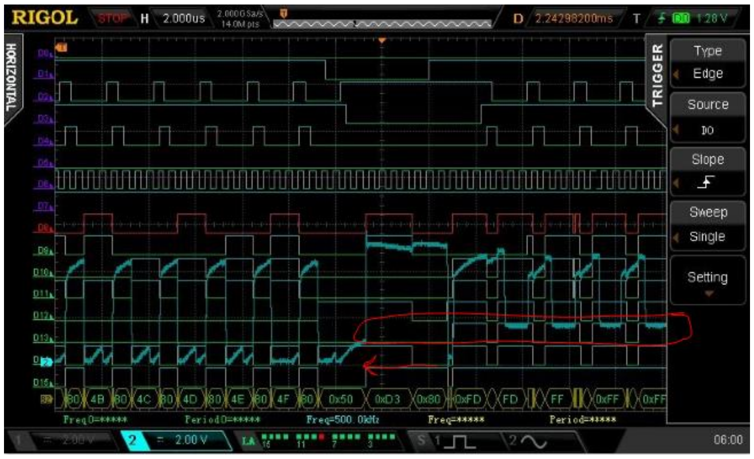




I thought perhaps the DMA state machine might asynchronously lock into the transfers at different points and treat them differently – possibly requiring the DMA clock frequency to be changed or something.

I also spent time working on the possibility that other peripheral chips were being enabled during the DMA transfer. By modifying their chip select equations in the PAL to be more restrictive, I was able to eventually eliminate this possibility.

Ultimately the problem had been looking me right in the face the whole time.



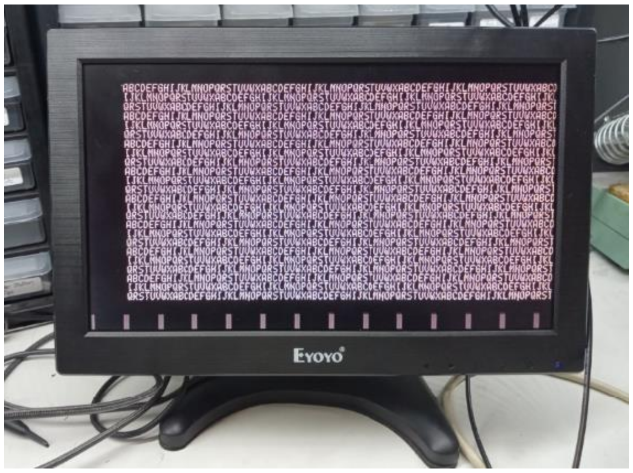
The screen shot above clearly shows what is likely to be bus contention. It couldn't be the peripherals, as I had made the enable logic immune to DMA transfers, so it had to be something else.

The only other thing on the bus was the 8080 and its 8228 bus buffer.

It was apparent immediately when I looked at my notes on the schematic. I had previously been using HLDA to disable the 8228 bus buffer, but I had recently changed that to be the DMA AEN line (as that is what was shown in the Intel app notes.) However, I screwed up and connected the 8228 ~BUSEN to the DMA ASTB line (which was the pin right next to AEN.) So, instead of the host bus being disabled during the whole DMA transfer, it was only disabled as the transfer MSB address was being setup for each byte!

This is why the data in the image above only looks good during ASTB, and looks like bus contention otherwise.

To be honest, I'm still confused that this worked at all – I would have expected ALL bytes transferred to be wrong, not just now and then. However, since this is such a blatant and obvious mistake that does explain why it was failing, I am now happy with the solution.



Intel – Screw Up?

As I dig deeper into the DMA instability issue, it gets stranger and stranger.

The 8257 DMA Controller is clearly "designed" to work with the 8212 latch chip – see Intel DMA System Interface Schematic below:

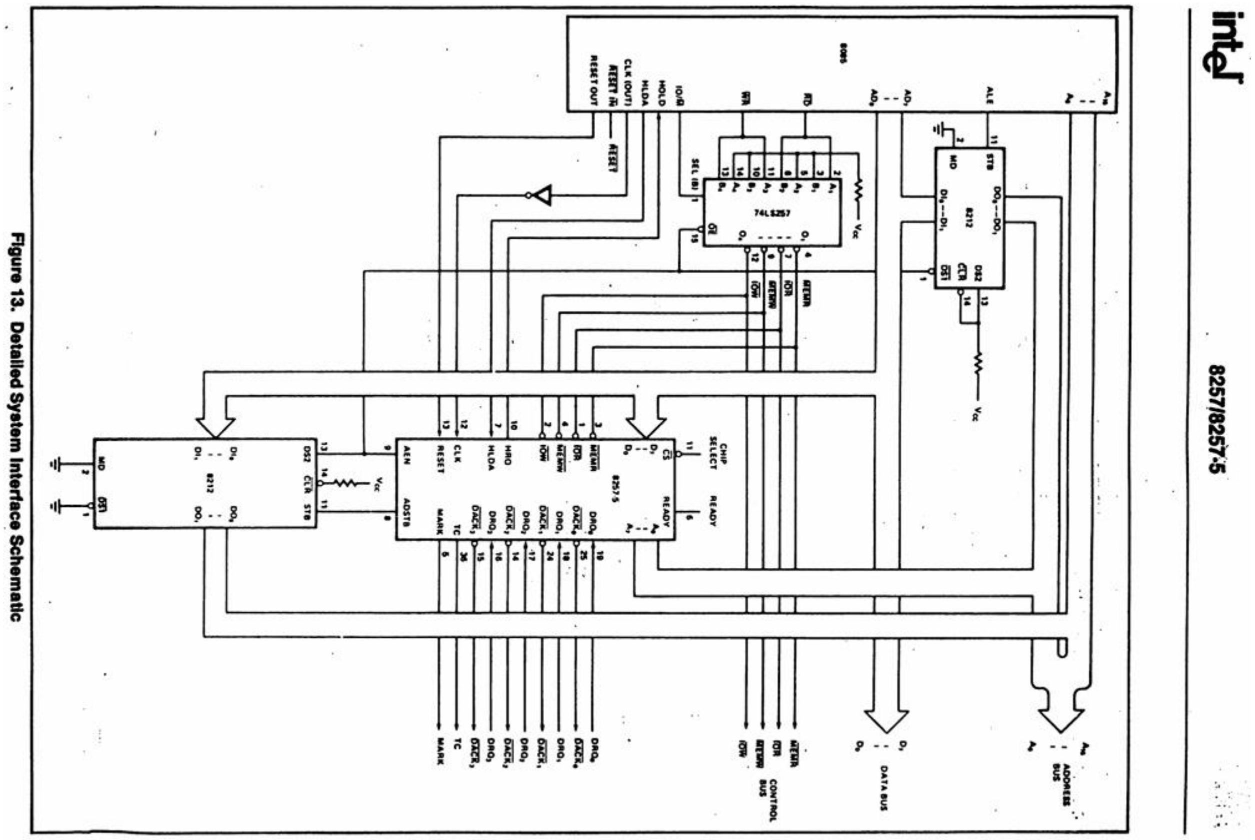
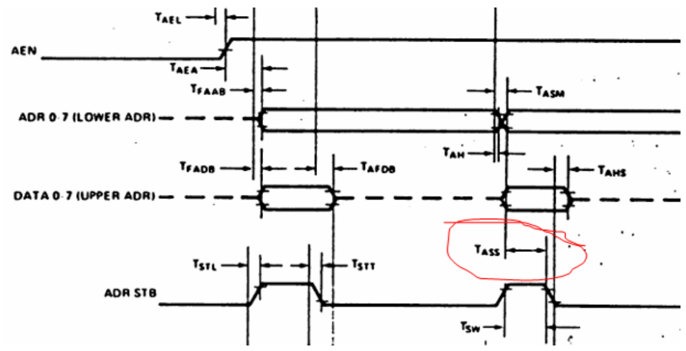
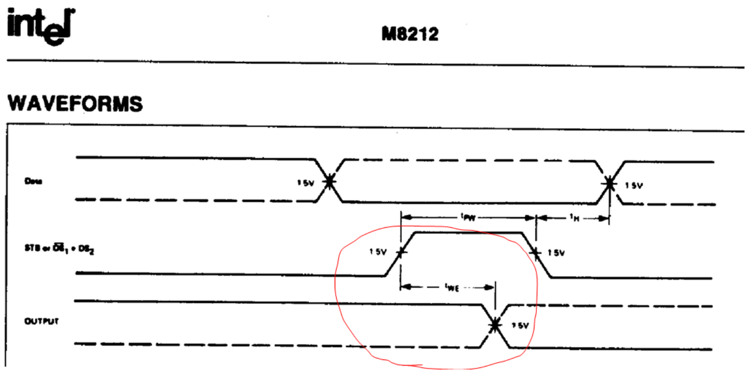


Figure 13. Detailed System Interface Schematic

However, when you dig deeper, you will see that the DMA spec guarantees that the LSB address bus data will be stable on D0-D7 Tass (>100nS) before the FALLING edge of ASTB.

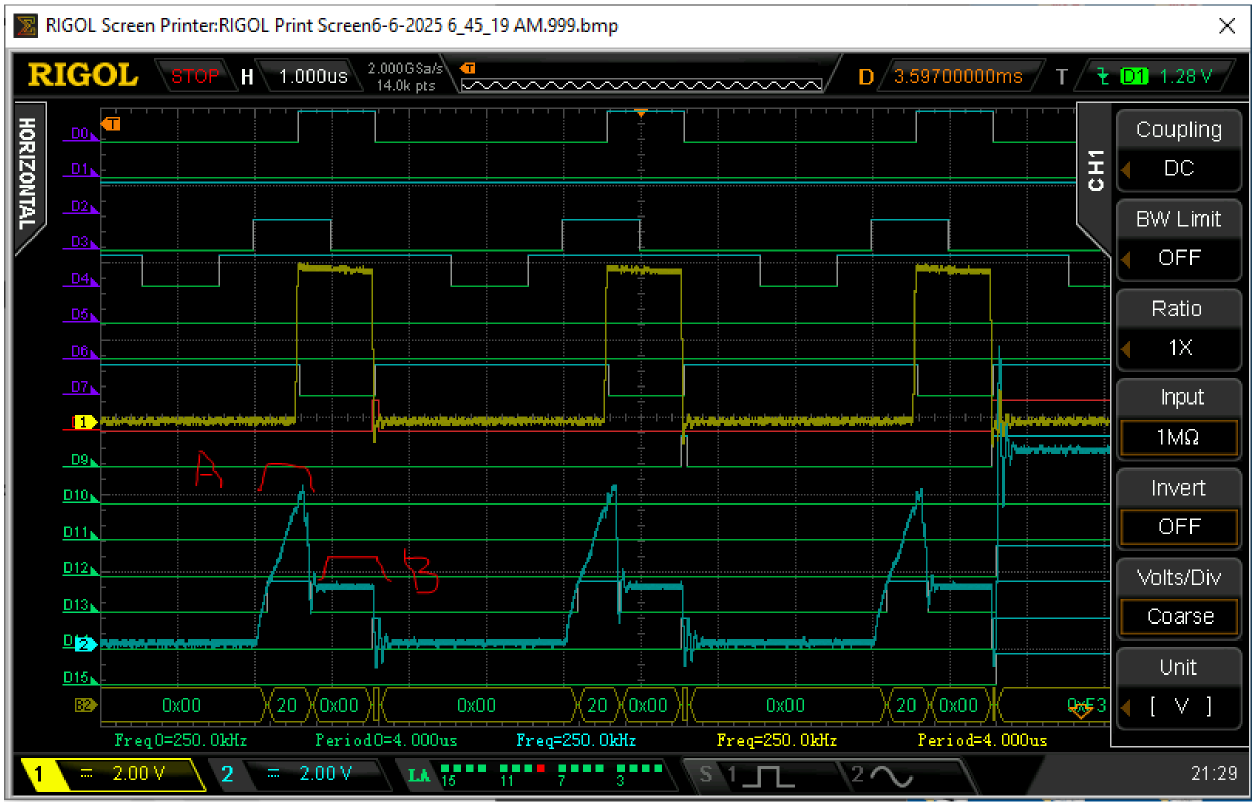


Whereas, the 8212 spec shows that the data bus inputs are transferred to the address outputs within Twe (<50nS) after the RISING edge of ASTB.



So, if the ASTB pulse is greater than Tass (100nS) the data latched into the 8212 will be wrong!

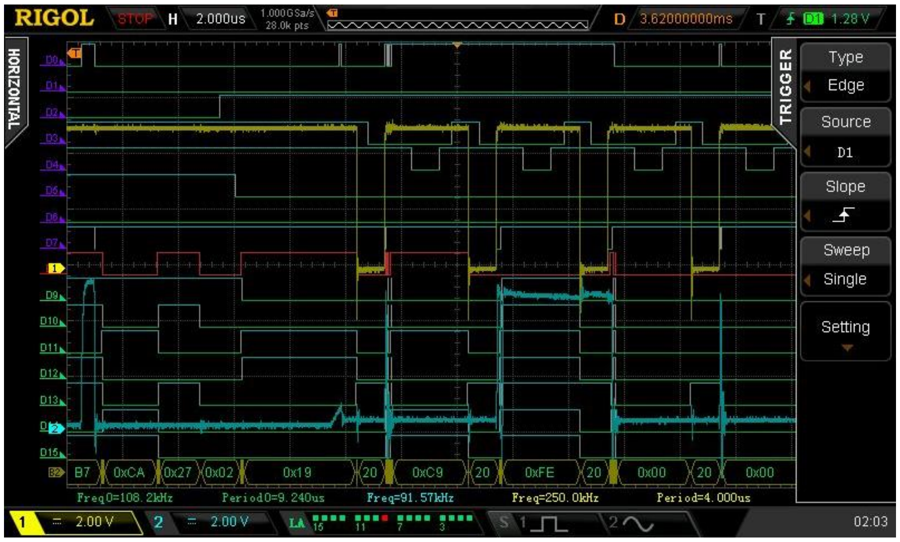
This is, in fact, the problem I was experiencing. Adding the pull-up resistor will only work in my very specific case where the RAM transfer address is in the 0x2xxx range. You can see this below – the yellow scope line is ASTB, and the blue is D5 from the DMA chip.


RIGOL Screen Printer:RIGOL Print Screen6-6-2025 6_45_19 AM.999.bmp
RIGOL STOP H 1.000us 2.000GSa/s 14.0k pts
D 3.59700000ms T D1 1.28 V
HORIZONTAL
D0 D1 D2 D3 D4 D5 D6 D7 D9 D10 D11 D12 D13 D14 D15
A
B
0x00 20 0x00 0x00 20 0x00 0x00 20 0x00
Freq0=250.0kHz Period0=4.000us Freq=250.0kHz Freq=250.0kHz Period=4.000us
CH1 Coupling DC BW Limit OFF Ratio 1X Input 1MΩ Invert OFF Volts/Div Coarse Unit [V]
1 2.00 V 2 2.00 V LA 15 11 7 3
21:29

You can see in section 'A' adding the pull-up resistor "fixed" the problem by pulling up the floating data bus so that the value can be correctly read by the 8212 on the rising edge of ASTB. You can then see in section 'B' where the DMA actually starts driving the data bus (in this case D5) to a value of '1' so that it is stable before the falling edge of ASTB.

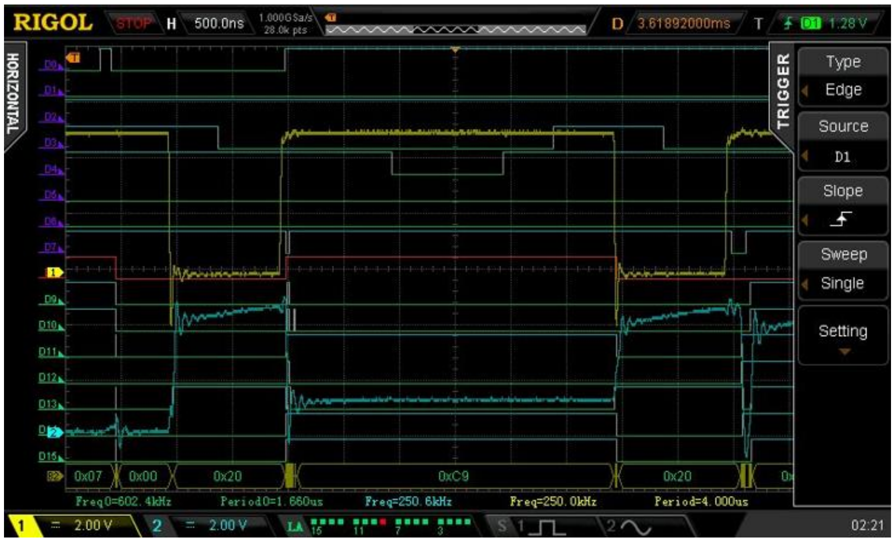
Inverting ASTB Test

I spent the time to modify the circuit to invert the DMA ASTB signal (yellow is inverted ASTB, blue is D5 8212 latch output or A13) on its way to the 8212 latch. You can see the results below.



H 2.000us
1.000GSa/s 28.0k pts
D 3.62000000ms
T D1 1.28 V
Type Edge
Source D1
Slope
Sweep Single
Setting
Freq0=108.2kHz
Period0=9.240us
Freq=91.57kHz
Freq=250.0kHz
Period=4.000us
1 = 2.00 V
2 = 2.00 V

And here is another (zoomed in) capture where the blue is the 8212 D5 input.



H 500.0ns
1.000GSa/s 28.0k pts
D 3.61892000ms
T D1 1.28 V
Type Edge
Source D1
Slope
Sweep Single
Setting
Freq0=602.4kHz
Period0=1.660us
Freq=250.6kHz
Freq=250.0kHz
Period=4.000us
1 = 2.00 V
2 = 2.00 V

As you can see, the results don't look any better which is rather confusing.

NTSC Standards and Monitors

The NTSC defines the standard by which the older analog video data is delivered to your TV/monitor – usually over an antenna, BNC, or RCA jack.

The standard dictates that your video signal should have:

- Horizontal sync pulses at 15.734KHz
- Vertical sync pulses at 59.94Hz
- 525 lines of pixels
- And many other parameters

Each TV/Monitor has a set of chips that process the video that you generate and either accept and display it, or reject it if it is out of range. Each of these chip sets enforce the standards to a different degree.

This is critical to understand while you are developing and getting your video “tuned in.” If you are using a monitor with strict timing requirements, you will see nothing on the screen while debugging. Whereas, with a laxer monitor you will see video data but it may be shifted out of the center of the screen when your video signal is less than perfect.

For example, with this Miktver monitor (Model 278e4493-6ca4-4787-8df1-bec7a554e1a5), I couldn’t see any video at all while developing. Whereas this Eyoyo monitor (model EM101AD, name K1223) would display my video even when my signal was way out of spec.

Sadly, I have not found an easy way to determine which monitor is more tolerant – other than to purchase one and see if it works or not.

Naturally, once you get your design completed and the video signal adheres to the standard, it really doesn’t matter which monitor you use.

One thing I learned from this project was to not limit the font rows/character. My video chip supported up to 16 pixel rows per character, but I chose to not connect the upper row line and only use 8 rows per character (because that was all I needed.) However, having extra blank font rows available below the character makes it easier to fine tune the video frequencies to match the NTSC standard, and to provide extra spacing between character rows.

8275 Inadequacy

I spent dozens of hours trying to get the 8275 video chip to properly format the video stream so that it would conform to NTSC standards and be accepted by standard monitors.

My mistake was thinking that the 8275 was a video interface controller (something that will interface with various industry standard video interfaces such as NTSC, CGA, VGA, etc.) whereas it is, in reality, a CRT controller (meant to directly drive CRT hardware, such as the internal guts of a dumb terminal – see example image below.)

There are many video interface features missing on the 8275 chip, such as:

1. Support for front/back porch timing
2. Support for interlaced video
3. Support for NTSC/PAL ½ line VSync pulses
4. Etc…

The fact that I got it to work at all with a standard monitor is kind of surprising.

The fact that the video is shifted right about 1/2” no longer bothers me.

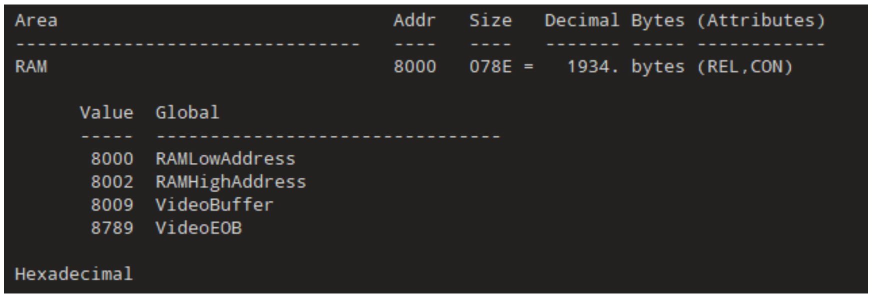
So, I’m not going to waste any more time trying to get the video to look perfect on the monitor.

Memory (or lack of) Issues

Another problem I ran into early on was that the processor would go berserk as I typed characters that were near the bottom of the screen. I didn't think too much about it in the beginning because I had bigger issues to deal with.

Later on, as I was trying to change the video setup to match NTSC timing specifications, I ran into another problem where the program stopped running if I changed the setup to anything bigger than 80 columns x 24 rows (specifically I was trying to change rows from 24 to 32.)

You can see below that the bootrom RAM usage is very minimal.



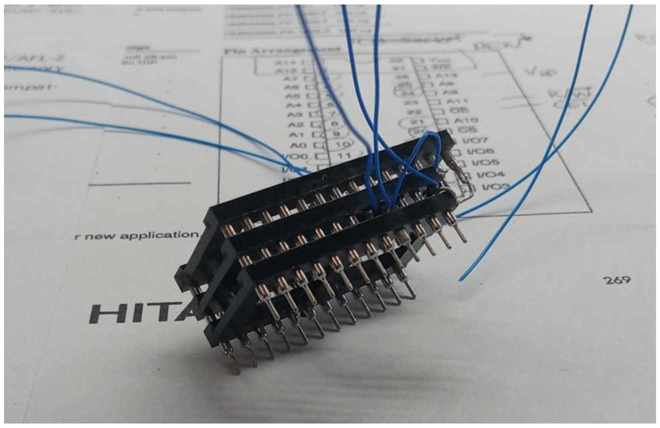
```
Area                             Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
RAM                              8000   078E =    1934. bytes (REL,CON)

      Value  Global
      -----  --------------------------------
       8000  RAMLowAddress
       8002  RAMHighAddress
       8009  VideoBuffer
       8789  VideoEOB

Hexadecimal
```

I was using the Toshiba TC5516 SRAM chips that I'd been lugging around for decades – to satisfy my project requirement of building something using these old parts. For some reason, I had mistakenly thought that these were 8K chips and that there was plenty of RAM to go around, especially for such a minimal application.

Sadly, I was mistaken. These are 2K SRAM chips – I should have paid more attention! And to make matters worse, the 80x24 video buffer occupied 1920 bytes of the RAM – leaving only 128 bytes free for the other variables, and the stack.

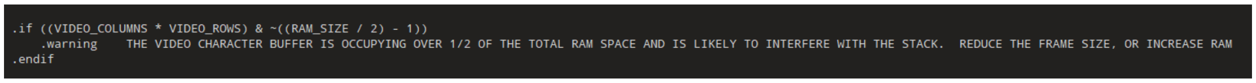
As the stack grew down, it would overlap the end of the video buffer. This explains why I would see random characters at the bottom of the screen sometimes. And as I typed characters near the bottom of the screen, they would corrupt the stack causing the program to crash! In a way this was kind of interesting, as it was the first time in my career that I actually 'saw' the stack changing with my own eyes!

I initially thought about using 2x TC5516 chips (soldered together, with the top chip select jumpered to a new chip select output) but I didn't have a spare output pin on the PAL for another chip select. So instead, I bought some old Hitachi HM62256 SRAM parts and built a socket adapter.



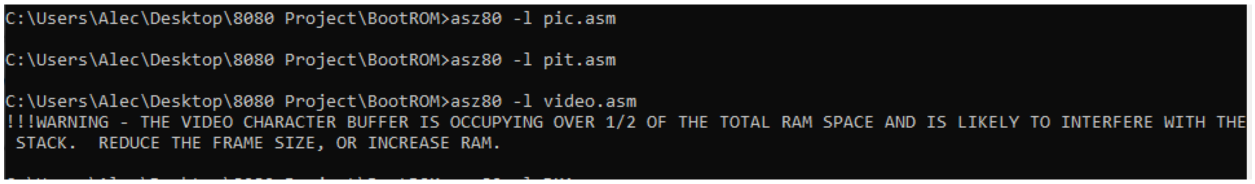
As a side issue, I also added a pre-processor warning to detect this sort of excessive memory usage and warn the user during build time. However, the ZCC tools that I'm using have no support for the standard #warning and #error directives – all you can do is use an invalid opcode which gives you a cryptic error message that you have to waste time tracking down.

In order to improve this situation, I spent some time modifying the ZCC assembler source code to add support for a '.warning' pre-processor command. So now, you can do things like this:



```
.if ((VIDEO_COLUMNS * VIDEO_ROWS) & ~((RAM_SIZE / 2) - 1))
    .warning    THE VIDEO CHARACTER BUFFER IS OCCUPYING OVER 1/2 OF THE TOTAL RAM SPACE AND IS LIKELY TO INTERFERE WITH THE STACK.  REDUCE THE FRAME SIZE, OR INCREASE RAM
.endif
```

And see a warning issued at build time as follows:



```
C:\Users\Alec\Desktop\8080 Project\BootROM>asz80 -l pic.asm

C:\Users\Alec\Desktop\8080 Project\BootROM>asz80 -l pit.asm

C:\Users\Alec\Desktop\8080 Project\BootROM>asz80 -l video.asm
!!!WARNING - THE VIDEO CHARACTER BUFFER IS OCCUPYING OVER 1/2 OF THE TOTAL RAM SPACE AND IS LIKELY TO INTERFERE WITH THE
 STACK.  REDUCE THE FRAME SIZE, OR INCREASE RAM.
```

I also increased the ZCC assembler line length from 128 to 250. This was very annoying before, because any comment that ran over the limit would cause an error – because the end of the comment would be treated as a new line.

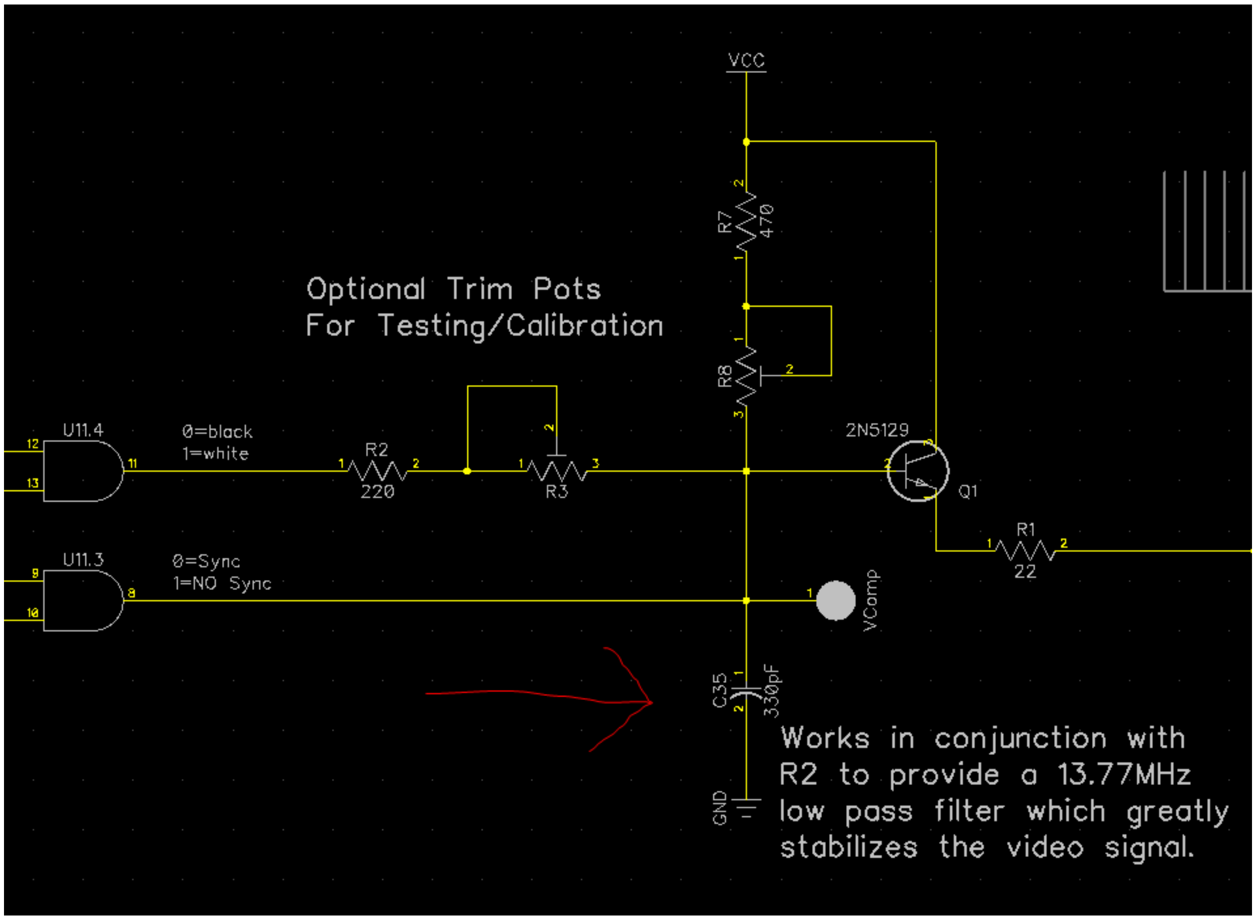
I updated the tool changes on my website so they are publicly available.

Video Instability Issue

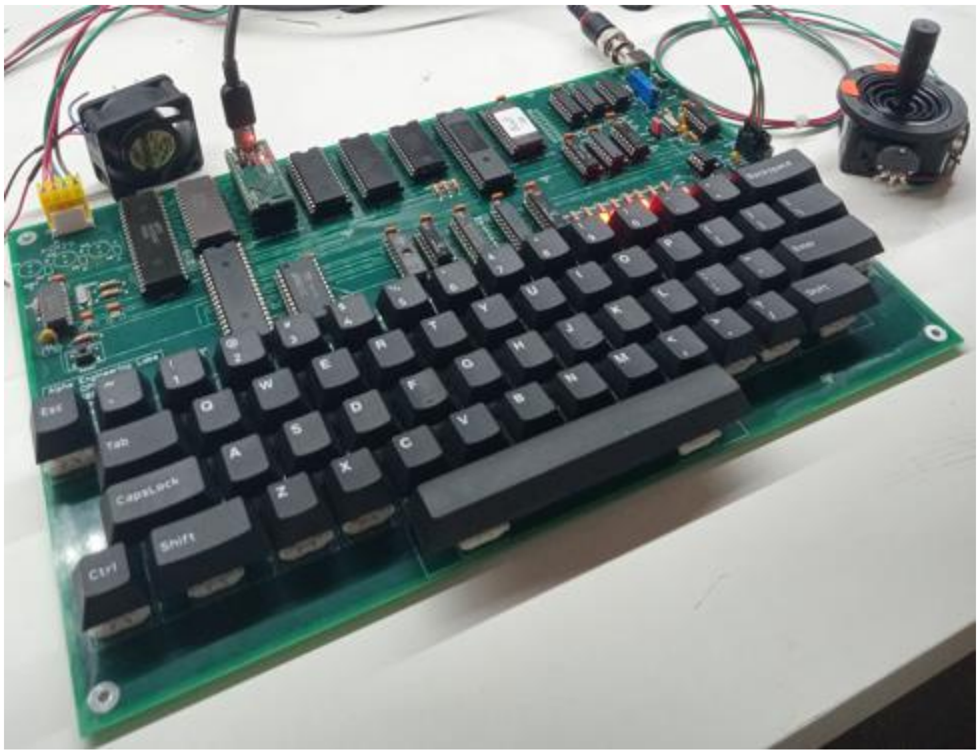
Throughout the design process there was video instability – tearing, jitter, loss of sync, etc…

I finally found that this was due mainly to high frequency noise on the video signal.

This was solved by adding a 330pF cap as shown below in the video output section.



Rev C PCB



The rev C PCB was generated after all of the preceding debug was completed – to resolve the issues found as well as to add a joystick interface so that simple video games could be programmed.

Important notes about the rev C board:

1. The Font ROM now requires 16 bytes/char (to support up to 16 rows.) Use Font16.asm instead of Font.asm to build the Font ROM.
2. The RAM has been changed from 2K to 32K.
3. There is no longer any need for the 76002980 daughter-board as all of that logic has been moved onto the main board.
4. A new joystick interface has been added to the rev C board.
5. An option to use a Piezo buzzer instead of LED d7 has been added to provide sound.

The rev C board works great and is much more stable than the rev B board.

Source Files for Project

Important – prior to clicking on the link below, please clear your browser cache to ensure you get the latest version (this is necessary only if you have previously downloaded this file.)

You can download the source files, tools, schematics & Gerber files for the project here: download

You can download the ZCC toolset (that I modified) here: download

You can see the steps required to build the PAL files here: link