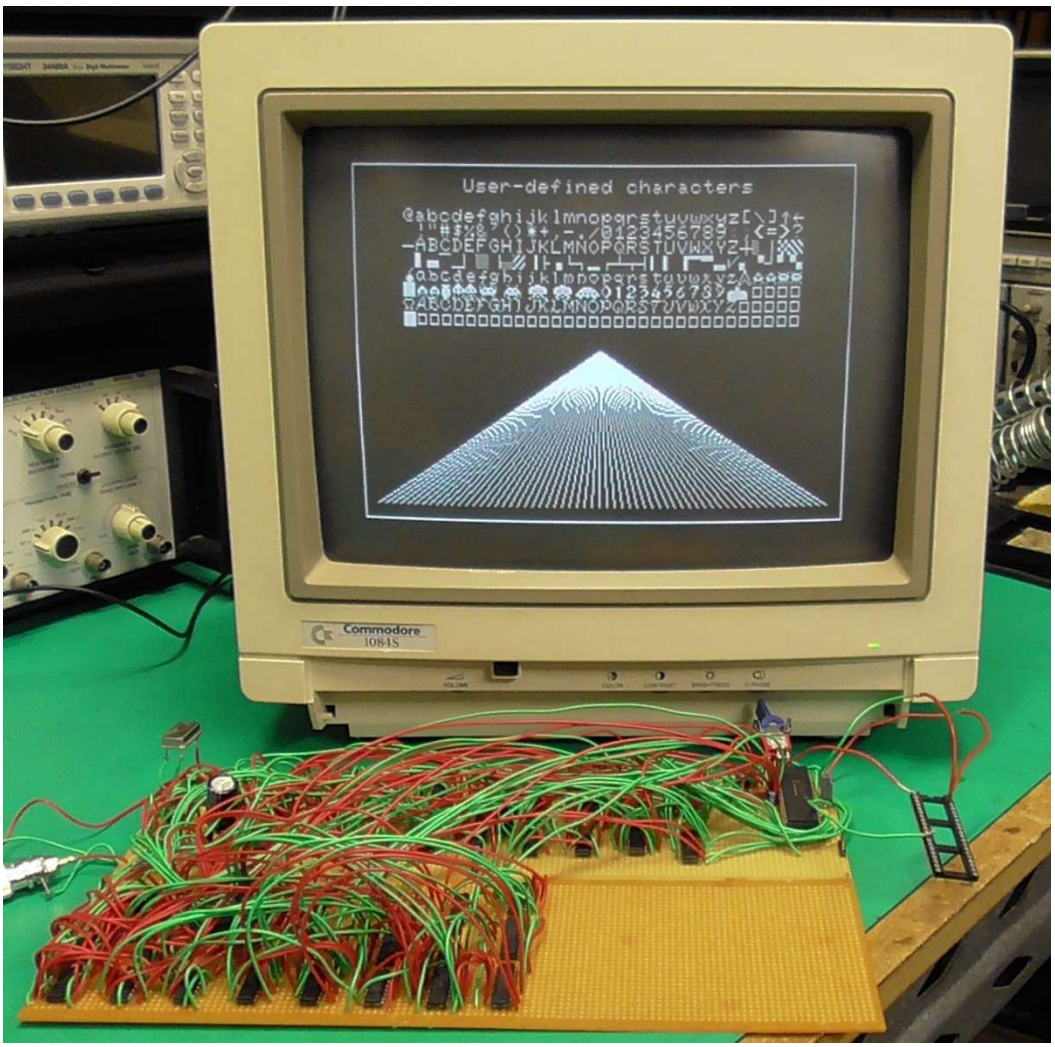
PET “3096” extended graphics computer.



Initial and preliminary specifications, schematics and video graphics-generator hardware prototyping.

Glen Kleinschmidt May/June 2025.

Introduction.

This project is an advance upon my initial PET 2001 clone computer built in 2018. It is another 40-column, non-CRTC, 2001 or 3000-series PET clone with a lot of functionality that the original, unexpanded computer never had.

The project is currently in the PCB-layout phase. I have completed the schematic entry and the full, ***preliminary*** schematics are presented in this document.

At the time of writing, I have only physically prototyped and verified the entirety of the video graphics generation hardware. Aside from the memory chips, the video generator circuitry that I have designed is composed entirely of 74HC and 74HCT-series discrete logic. This is the most complex part of the computer, and the whole of it was assembled on veroboard, as shown and detailed in this document.

I do not intend to veroboard the entire computer. As I immediately needed some kind of readily accessible and practically programmable CPU (so as to not waste eons of time) to program the video memories and thus produce some form of test graphics on screen, I plonked a PIC16F874 microcontroller onto the veroboard and wired it to the video generator hardware control lines, data and address busses. The test program for the PIC was written in C and is given in this document.

All the video graphics screen-shots featured in this document, taken from the Commodore 1084S monitor, were generated by this veroboard video hardware.

The main features of this new PET computer design are:

- BASIC 4.0 operating system.
- "Snow"-free video graphics.
- 320 x 200 screen resolution bitmap graphics
- 128 user-definable characters in addition to the standard PETSCII character ROM
- Full 32 KB system RAM
- +64KB RAM expansion, functionally equivalent to the original CBM 64KB RAM expansion.
- On-board audio DAC with an effective 3.5kHz reconstruction filter for sound synthesis. Inspired by the MTU K-1002-2.

Total RAM is 96KB. However, there is an additional 1KB RAM for the user-defined characters and an additional 8KB RAM for the bitmap graphics. If both bitmap graphics and user-defined characters are not required and thus left disabled, the memories can be utilised for other purposes, giving a total RAM storage capacity of 105KB.

That's enough RAM to play approximately 13 seconds of sampled speech through the audio DAC at a sampling rate of 8kHz!

Custom graphics.

The two biggest limitations of the original machine, as far as its video graphics capabilities are concerned, were being restricted to the fixed "PETSCII" character set stored in ROM and the complete lack of bitmap graphics.

The original PET computer could write to the screen from a set of 256 characters, but only 128, or half of these were truly unique as the top 128 were simply the "reverse" of bottom 128. The "reverse" characters were not defined in the character ROM, but were generated by logically inverting the video signal whenever bit 7 for the selected character was set high. So, in effect, the video generator circuitry produced 256 characters from a basic set of 128.

The 2KB character ROM contained two basic sets of 128 characters, only one of which could be used at a time. The first one of these, which was enabled by default upon power-on, was named the "graphics" set while the other was named the "business" set. The two sets are mostly identical, except that in the business set a bunch of the so-called "PETSCII" graphics characters make way for the inclusion of both upper and lowercase characters. The graphics set only contains uppercase characters, so was not of much use for serious business applications back in the day such as word processing.

My "extended graphics" PET is designed to boot as per a normal PET - that is with the bitmap graphics and user-definable character set disabled by default. The expanded graphics and audio hardware is controlled by a single-byte, write-only register located at 0xFFF1.

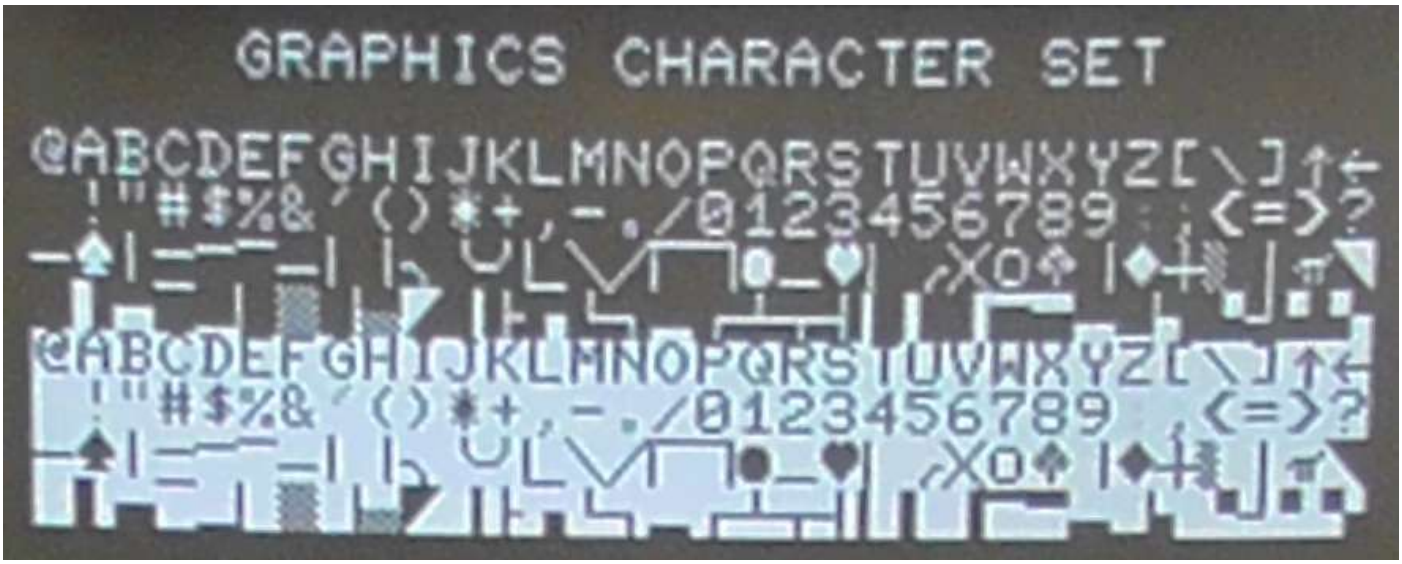
Both bitmap graphics and user-defined characters can be independently enabled by setting specific bits of this register.

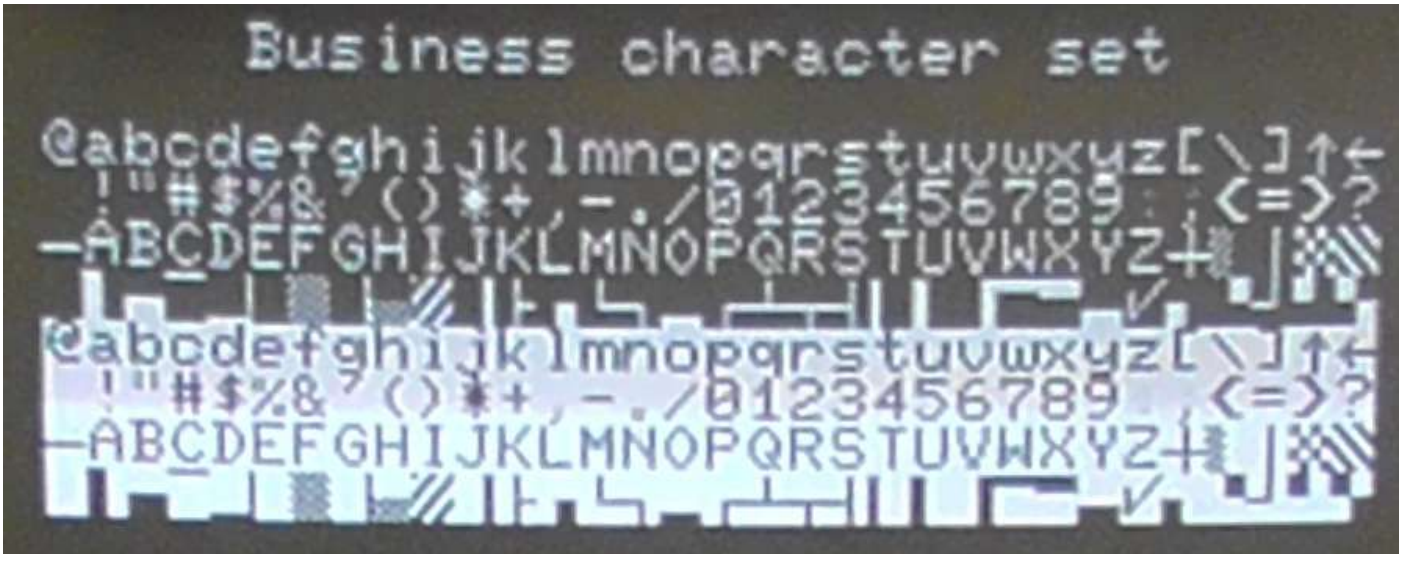
When user-defined characters are enabled, reverse characters are turned off and the video generator instead fetches a video display character from an added character RAM whenever bit 7 for the selected character is set high. The video hardware can therefore address 128 user-defined characters in addition to the 128 accessible from the character ROM.

With user-defined characters turned on, the PET can simultaneously display 256 truly unique characters. Although the function of reversed characters is disabled, any “reverse” character or characters can simply be user-defined if required. The character RAM is both readable and writable, is 1KB in size and is addressed in parallel to the standard 1KB video text (“screen”) memory. Bank switching is controlled by the control register at 0xFFF1.

Like the character RAM, the bitmap RAM is both readable and writable and is addressed in parallel to the standard 1KB video text memory (the latter of which has 4 mirrors as it is only partially decoded). The bitmap RAM is 8KB in size and is split into two separate 4KB pages. Bank switching and page selection of the bitmap video RAM is again controlled the control register at 0xFFF1.

The screen shots immediately below show the standard graphics and business character sets, and then, lastly, user-defined characters enabled when the business set is engaged. Here I have programmed most of the available user-definable characters with something unique. I have included the alpha-numeric character font from the C64 arcade port of *Ghosts ‘n Goblins*. Additionally, there are some characters suitable for video games programming, which include characters from the arcade versions of *Space Invaders* and *Gorf*.



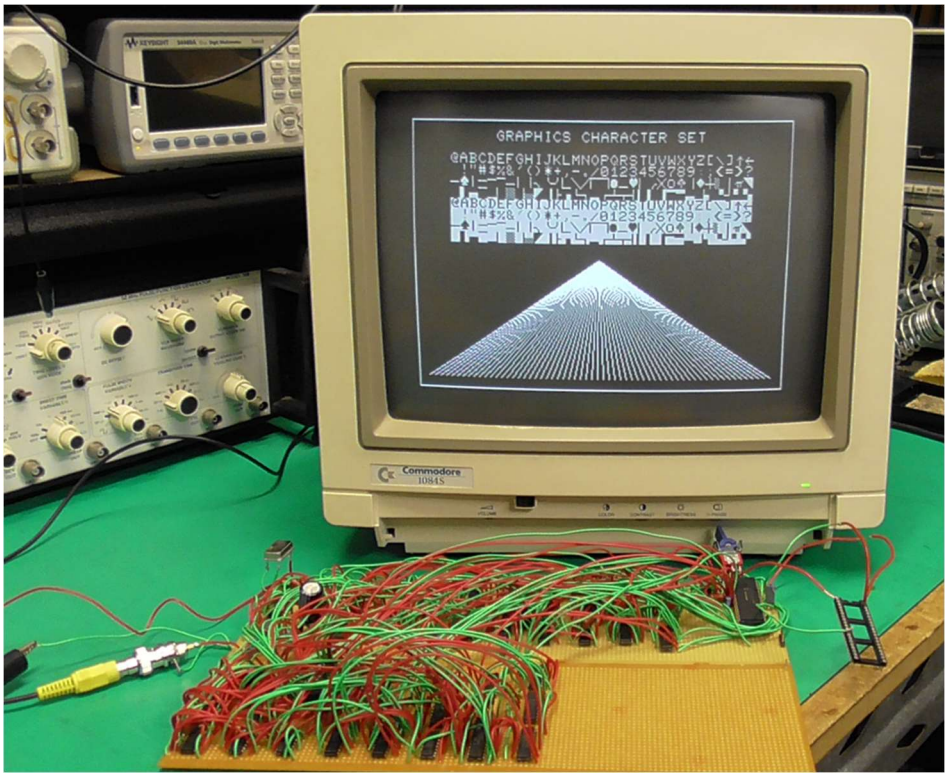


My current plan is to complete this set of 128 unique characters and incorporate it into a set of routines stored in the expansion ROM area. The expansion ROM area is a reserved 8KB chunk of high memory which I have address and pinned-out for a 40-pin DIP socket taking an AT28C256 flash EEPROM.

My plan is to change the 6502-reset vector at 0xFFFC and 0xFFFD in the original BASIC 4.0 ROM to point to a routine stored in the expansion ROM. This routine will clear the bitmap RAM (meaning to set all bytes to 0x00) and copy my custom character set (also stored in the expansion ROM) to the character RAM. The routine will then jump back to the original reset vector.

This means that, initially upon power-on, the PET will initialise both the bitmap and character RAMs before booting as per normal. I might even throw in a memory testing routine as well.

Below is another picture of the prototype video generation hardware.
Here is a link to a video uploaded to Vimeo; a filming of the graphics demonstration and test routines in action.



Memory map											
Address						Bank 1				Bank 2	
Decimal			Hex								
0	-	2047	0	-	7FF	Main RAM 32KB - not effected by bank switching					
2048	-	4095	800	-	FFF						
4096	-	6143	1000	-	17FF						
6144	-	8191	1800	-	1FFF						
8192	-	10239	2000	-	27FF						
10240	-	12287	2800	-	2FFF						
12288	-	14335	3000	-	37FF						
14336	-	16383	3800	-	3FFF						
16384	-	18431	4000	-	47FF						
18432	-	20479	4800	-	4FFF						
20480	-	22527	5000	-	57FF						
22528	-	24575	5800	-	5FFF						
24576	-	26623	6000	-	67FF						
26624	-	28671	6800	-	6FFF						
28672	-	30719	7000	-	77FF						
30720	-	32767	7800	-	7FFF						
32768	-	34815	8000	-	87FF	Screen memory 1 KB (4 mirrors)	128-character custom character set RAM 1 KB (4 mirrors)	320 x 200 pixel bitmap video RAM Block 0 4KB	320 x 200 pixel bitmap video RAM Block 1 4KB	64KB Expansion RAM Block 0 16KB	64KB Expansion RAM Block 1 16KB
34816	-	36863	8800	-	8FFF						
36864	-	38911	9000	-	97FF	Expansion ROM area 8KB					
38912	-	40959	9800	-	9FFF						
40960	-	43007	A000	-	A7FF						
43008	-	45055	A800	-	AFFF						
45056	-	47103	B000	-	B7FF	BASIC 4.0 ROM 12KB					
47104	-	49151	B800	-	BFFF						
49152	-	51199	C000	-	C7FF					64KB Expansion RAM Block 2 16KB	64KB Expansion RAM Block 3 16KB
51200	-	53247	C800	-	CFFF						
53248	-	55295	D000	-	D7FF						
55296	-	57343	D800	-	DFFF						
57344	-	59391	E000	-	E7FF	BASIC 4.0 screen editor ROM 2KB					
59392	-	61439	E800	-	EFFF	I/O operations 2KB					
61440	-	63487	F000	-	F7FF	BASIC 4.0 kernal ROM 4KB					
63488	-	65535	F800	-	FFFF						

Character ROM (no CPU access)						
0	-	1023	0	-	3FF	Upgraded graphics character set (reverse) - 1KB
1024	-	2047	400	-	7FF	Upgraded grahics character set - 1KB
2048	-	3071	800	-	BFF	Upgraded business character set (reverse) - 1KB
3072	-	4095	C00	-	FFF	Upgraded business character set - 1KB
4096	-	5119	1000	-	13FF	Original graphics character set (reverse) - 1KB
5120	-	6143	1400	-	17FF	Original grahics character set - 1KB
6144	-	7167	1800	-	1BFF	Original business character set (reverse) - 1KB
7168	-	8191	1C00	-	1FFF	Original business character set - 1KB

Memory control register. Address = 0xFFF0		
Bit	Function	Power-on-reset/default status
0	Write protect blocks 0 and 1	Write protect off
1	Write protect blocks 2 and 3	Write protect off
2	Select block 0 or 1	Block 0
3	Select block 2 or 3	Block 2
4	Not used	
5	Turn on screen peek-through (0x8000-0x8FFF)	Off
6	Turn on I/O peek-through (0xE800-0xEFFF)	Off
7	Enable 64KB RAM expansion	Off

Video and sound control register. Address = 0xFFF1		
Bit	Function	Power-on-reset/default status
0	Bank-switch video RAM	Off - text RAM selected
1	Select character RAM or Bitmap RAM	Character RAM
2	Select bitmap RAM page 0 or page 1	Page 0
3	Enable user defined characters	Off
4	Enable bitmap graphics	Off
5	Enable original character set	Off
6	Enable audio DAC	Off
7	Disable CB2 audio	Off - CB2 audio enabled

Bitmap graphics programming.

The screen resolution of 320 (H) x 200 (V) pixels maps to 8KB of bitmap RAM. Each byte of RAM defines 8 pixels. The most significant bit of the first byte of the bitmap RAM maps to the topmost and leftmost pixel (coordinate 0*x*, 0*y*). The least significant bit of the last byte of the bitmap RAM maps to the bottommost and rightmost pixel (coordinate 319*x*, 199*y*).

For any given pixel the relevant bitmap RAM byte address is:

$$(y * 40) + \text{INT}\ (x / 8)$$

Turning a pixel on or off is, ordinarily, both a read and a write operation. The relevant bitmap RAM address is first read. It is then either Boolean ORed or ANDed with a mask byte and the result written back to the bitmap RAM. The mask byte determines which bits of the bitmap RAM location are to be modified. Boolean OR sets a bit if the corresponding mask byte bit is set. Boolean AND resets a bit if the corresponding mask byte bit is not set.

Multiplication or division by any value that is a power of two can be achieved with non-circular bit-shifting. There are 40 bytes of bitmap RAM to each horizontal line (320 / 8 = 40).

32 and 8 are both powers of 2 and sum to 40. Therefore, to multiply *y* by 40:

*y*1 = *y* * 8 (Lefthand shift by 3 bits)
*y*2 = *y* * 32 (Lefthand shift by 5 bits)
y * 40 = *y*1 + *y*2

To divide x by 8, righthand shift by 3 bits. The result is automatically an integer as the three least significant bits are discarded.

To generate the mask byte:

1) Multiply the resultant integer value of (*x* / 8) by 8 by lefthand shifting by three bits.
2) Subtract the value from *x*. The result is the remainder.
3) Start with a mask byte set to 0x80 (128 dec) and righthandshift *n* times, where *n* = the remainder.

A worked example:

Suppose that we want to turn on a single pixel, at coordinate position 211*x*, 88*y*.

88 (dec.) = 0101 1000 (bin.)
*y*1 = 0010 1100 0000 (704 dec.)
*y*2 = 1011 0000 0000 (2816 dec.)
*y*1 + *y*2 = 1101 1100 0000 (3520 dec.) = *y* * 40

211 (dec.) = 1101 0011 (bin.)
INT (*x* / 8) = 0001 1010 (26 dec.)

(y * 40) + INT (x / 8) = 3520 (dec.) + 26 (dec.)
= 3546 (dec.)
3546 (dec.) is the bitmap RAM memory address.

26 (dec.) = 0001 1010 (bin.)
26 (dec.) * 8 = 1101 0000 (bin.)
= 208 (dec.)

x (211) – 208 = 3.
As the remainder is 3, the initial mask bit value of 0x80, (1000 0000 [bin.]) is righthand shifted by three bits:

Mask byte = 0001 0000 (bin.)

The set bit of the mask byte now corresponds to our single pixel at coordinate position 211*x*, 88*y*, when applied to the calculated bitmap RAM memory address location of 3546 (dec.). Use Boolean OR with the mask byte to set this pixel.

Video timing.

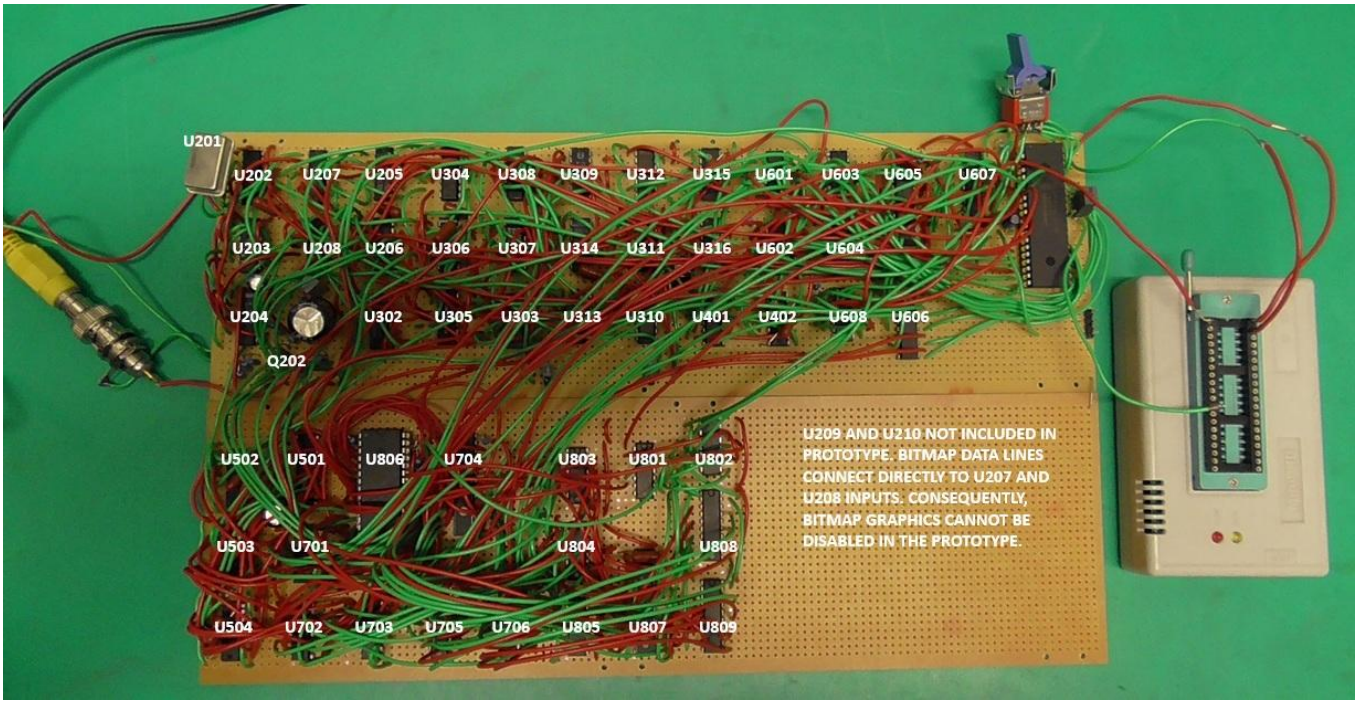
The video generator produces a simplified, non-interlaced and monochrome composite video signal suitable for driving a video monitor or television complying to the old PAL and NTSC timing specifications.

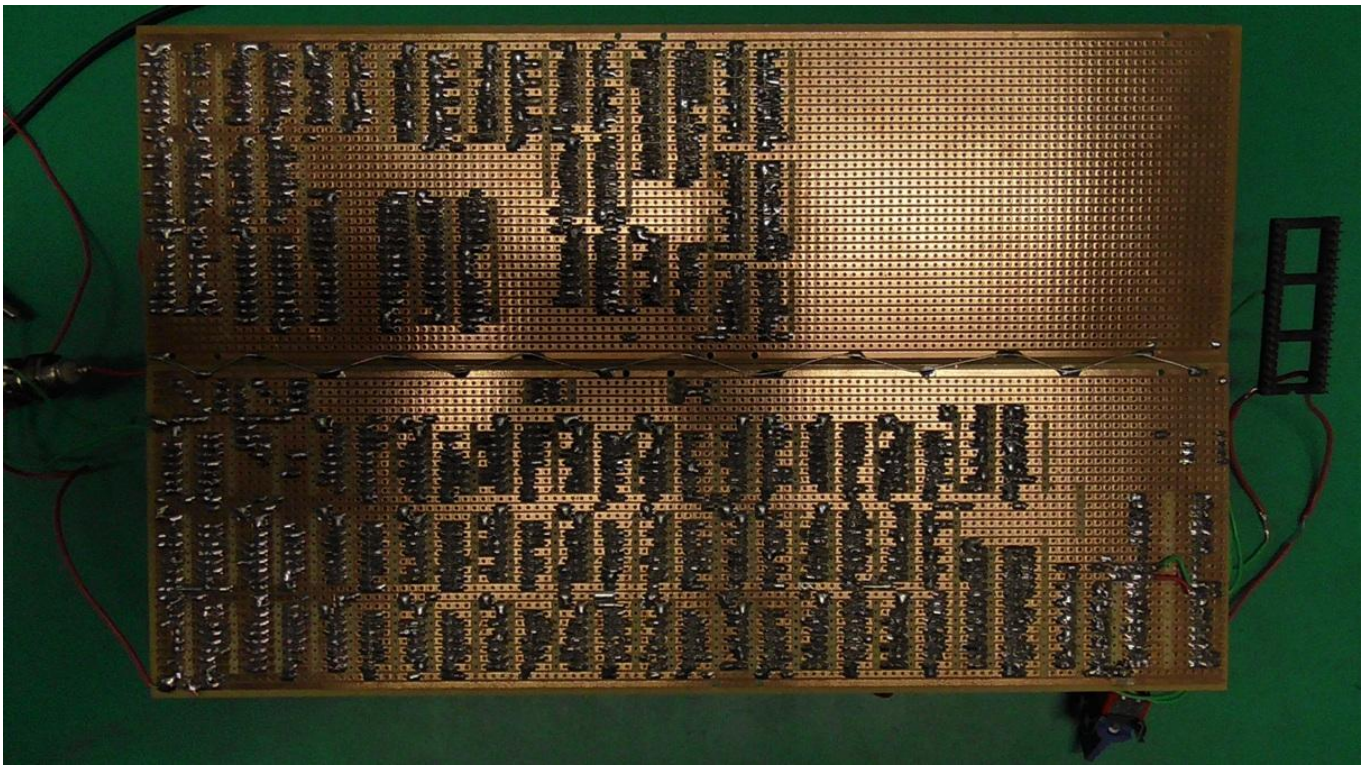
The horizontal and vertical timings are worked out to produce a graphics display centred on the screen of one of these devices. The vertical timing circuitry can be configured via two on-board jumpers for either a 50 Hz or a 60 Hz field frequency or frame rate.

	50 Hz field frequency setting	60 Hz field frequency setting
Bitmap resolution (pixels H & V)	320 x 200	
Character size (pixels H & V)	8 x 8	
Character columns	40	
Character rows	25	
Pixel clock	8 MHz (125 ns)	
Horizontal line period (H)	64 us (15.625 kHz) 512 clock cycles	
Horizontal sync	4 us (32 clock cycles)	
Horizontal front porch	8 us (64 clock cycles)	
Horizontal back porch	12 us (96 clock cycles)	
Horizontal blanking	24 us (192 clock cycles)	
Active horizontal video	40 us (320 clock cycles)	
Field/frame period	19.968 ms (50.08 Hz) 312 H	16.64 ms (60.096 Hz) 260 H
Vertical sync	192 us (3 H)	
Vertical front porch	2.944 ms (46 H)	1.472 ms (23 H)
Vertical back porch	4.032 ms (63 H)	2.176 ms (34 H)
Active vertical video	12.8 ms (200 H)	

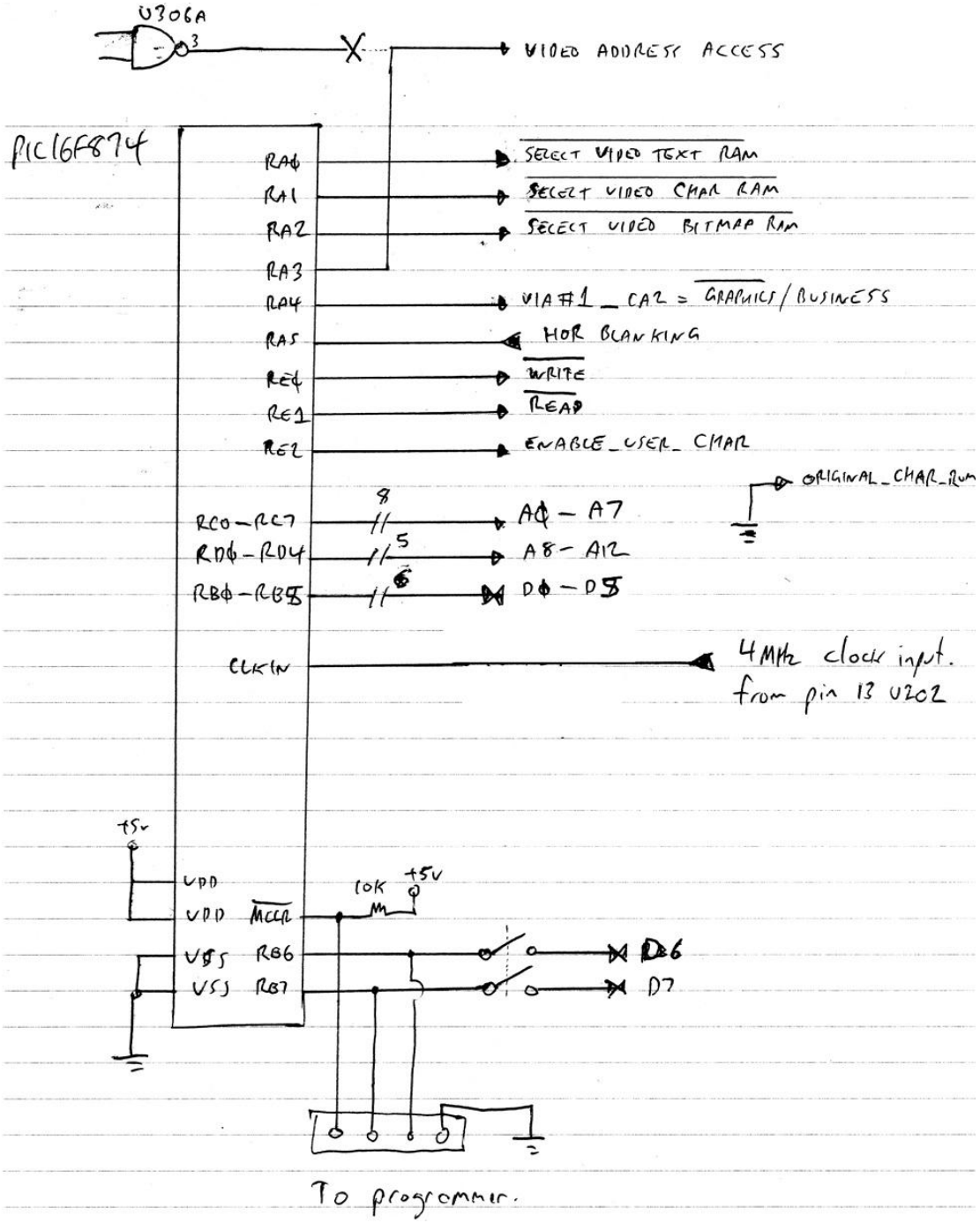
Video generator prototype.

Immediately below are a couple of photos of the prototype video generator. The socketed, 40-pin DIP IC in the top righthand corner in the first photo is the aforementioned PIC16F874A microcontroller used as a CPU to control the video generator. It is wired to an old TL866 Universal Programmer for programming with an improvised ICSP (in circuit serial programming) plug!





Here is the wiring diagram for the PIC16F874A microcontroller:



The VIDEO_ADDRESS_ACCESS control line signal is designed to interleave video RAM access with the 65C02 microprocessor. There are two phases to the 65C02s 1MHz master clock, one of which concludes with a memory access. The video generator always accesses its video memories during the opposite phase. This is how the 65C02 can access the video memories without interfering with the operation of the video generator, thus avoiding the so-called video “snow” that was a feature of the first-generation PET computer.

This memory access phasing, however, does not suit the PIC16F874A, so an alternative method has been used to produce snow-free video memory updates. The VIDEO_ADDRESS_ACCESS control line is instead controlled by the PIC and the horizontal blanking signal is fed to the microcontroller as an input signal. The PIC is programmed to only assert the VIDEO_ADDRESS_ACCESS control line for video RAM reads or writes during the horizontal blanking periods.

The complete C language program listing for the PICs video driver/demonstration program follows on the next page. I am using a PIC C compiler by CCS (Custom Computer Services).

```
// PET SUPER GRAPHICS TESTER

#include <16f874A.h>
#include <math.h>
#use delay(clock=4000000)
#use fast_io(A)
#use fast_io(B)
#use fast_io(C)
#use fast_io(D)
#use fast_io(E)
#FUSES HS,NOWDT,PUT,NOPROTECT,NOLVP,NOCPD,NOWRT,BROWNOUT

int8           data, rbyte, mask;
int16          address, count;
signed int16   xx, x, y, x1, y1, x2, y2, deltax, deltay, ddx, ddy, f;

const int usercharacterset[1024] = {
2,4,8,8,28,62,62,28,0,0,60,68,68,76,50,0,0,64,32,60,34,34,92,0,0,0,60,64,64,66,60,0,
0,12,4,60,68,68,58,0,0,0,28,34,62,32,30,0,0,6,8,8,62,8,8,16,0,0,58,68,68,60,4,120,
32,32,32,60,34,34,36,0,0,8,0,24,8,8,8,0,0,8,0,24,8,8,8,48,0,32,32,36,56,36,34,0,
0,48,16,16,16,16,16,0,0,0,108,82,82,82,84,0,0,0,60,34,34,34,36,0,0,0,60,66,66,36,24,0,
0,0,92,34,34,60,32,32,0,0,58,68,68,60,4,2,0,0,76,50,34,32,32,0,0,0,60,64,56,4,56,0,
0,16,62,16,16,18,12,0,0,0,36,68,68,68,58,0,0,0,100,34,34,36,24,0,0,0,68,82,82,82,44,0,
0,0,52,24,16,40,70,0,0,0,34,18,20,12,72,48,0,0,60,72,16,34,124,0,24,24,36,36,66,90,165,195,
0,8,28,42,119,62,65,34,0,8,28,42,119,62,34,65,0,65,62,107,127,62,85,42,0,65,62,107,127,62,42,73,

255,255,255,255,255,255,255,255,0,0,28,62,107,119,42,65,0,0,28,62,107,127,34,20,0,126,60,126,126,126,126,60,
24,60,126,219,255,90,129,66,24,60,126,219,255,36,90,165,8,36,47,59,63,31,8,16,32,72,232,184,248,240,32,16,
8,4,15,27,63,47,40,12,32,64,224,176,248,232,40,192,3,31,63,57,63,6,13,48,192,248,252,156,252,96,176,12,
3,31,63,57,63,14,25,12,192,248,252,156,252,112,152,48,0,7,31,63,109,255,57,16,0,224,248,252,182,255,156,8,
24,44,70,70,70,44,24,0,8,56,24,24,24,24,24,0,60,102,38,12,48,98,124,0,124,8,28,38,6,12,48,0,
8,16,48,44,110,124,12,0,60,32,120,76,12,24,32,0,4,24,48,104,68,76,56,0,62,70,12,8,24,16,16,0,
28,50,18,44,70,70,60,0,28,34,34,22,12,24,32,0,1,3,3,63,127,127,127,127,0,128,128,248,252,252,252,252,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,

0,124,130,130,68,40,108,0,12,20,36,36,124,68,130,0,248,68,120,68,66,98,156,0,56,68,132,128,130,68,56,0,
216,100,66,66,66,68,184,0,220,98,64,112,64,68,248,0,220,98,64,112,64,64,128,0,60,66,128,158,132,76,50,0,
194,68,68,124,68,68,132,0,48,16,16,16,16,16,32,0,6,4,4,68,132,136,112,0,198,72,80,112,72,74,132,0,
96,32,32,32,64,98,156,0,194,102,86,74,74,66,130,0,194,98,82,74,70,70,130,0,56,68,130,130,130,68,56,0,
220,98,66,98,92,64,128,0,56,68,130,130,148,74,52,0,220,98,66,68,120,74,132,0,60,66,48,8,68,132,120,0,
126,136,80,16,32,32,32,0,194,34,34,66,68,68,56,0,194,66,36,36,40,48,16,0,132,66,82,82,82,116,72,0,
198,40,48,16,48,74,132,0,198,68,40,16,16,32,32,0,62,68,8,16,32,114,140,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0

255,255,255,255,255,255,255,255,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,
0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0,0,126,66,66,66,66,126,0
};

const int GRAPHICS_CHARACTER_SET[22] = {7,18,1,16,8,9,3,19,32,3,8,1,18,1,3,20,5,18,32,19,5,20};
const int Business_character_set[22] = {66,21,19,9,14,5,19,19,32,3,8,1,18,1,3,20,5,18,32,19,5,20};
const int User_defined_characters[23] = {85,19,5,18,45,4,5,6,9,14,5,4,32,3,8,1,18,1,3,20,5,18,19};
```

```
void PlotPixel()
{
   xx = x; xx >>=3;                                                  // INT (x/8)
   address = (y * 40) + xx;                                          // Compute memory address
   xx <<=3; xx = x - xx;                                             // INT (x/8)*8 and remainder
   mask = 128; mask >>= xx;                                          // Generate OR mask byte for pixel bit to set
   output_C(make8(address, 0)); output_D(make8(address, 1));         // Assert address bus
   while (input(PIN_A5)); while (!input(PIN_A5));                    // Wait for positive edge of H_BLANK
   output_high(pin_A3);                                              // Assert VIDEO_ADDRESS_ACCESS
   output_low(pin_A2);                                               // Assert !SELECT_VIDEO_BITMAP_RAM
   output_low(pin_E1); rbyte = input_B(); output_high(pin_E1);       // Read bitmap RAM address
   output_B(mask | rbyte);                                           // OR with mask byte and put on data bus
   output_low(pin_E0); output_high(pin_E0);                          // Strobe !WRITE
   output_high(pin_A2);                                              // Unassert !SELECT_VIDEO_BITMAP_RAM
   output_low(pin_A3);                                               // Unassert VIDEO_ADDRESS_ACCESS
}

void QuadrantA()                                                     // Bresenham's line drawing algorithm for DrawLine() function
{
   deltax = x2 - x1; ddx = 2 * deltax; deltay = y2 - y1; ddy = 2 * deltay;
   if (deltay > deltax) {
      f = ddx - deltay;
      while (y <= y2)   {
         PlotPixel();
         if (f > 0)  {
            x++; f = f - ddy;
         }
         f = f + ddx; y++;
      }
      y = y2;
   }
   else  {
      f = ddy - deltax;
      while (x <= x2)   {
         PlotPixel();
         if (f > 0)  {
            y++; f = f - ddx;
         }
         f = f + ddy; x++;
      }
      x = x2;
   }
}

void QuadrantB()                                                     // Bresenham's line drawing algorithm for DrawLine() function
{
   deltax = x1 - x2; ddx = 2 * deltax; deltay = y2 - y1; ddy = 2 * deltay;
   if (deltay > deltax) {
      f = ddx - deltay;
      while (y <= y2)   {
         PlotPixel();
         if (f > 0)  {
            x--; f = f - ddy;
         }
         f = f + ddx; y++;
      }
```

```
      y = y2;
   }
   else  {
      f = ddy - deltax;
      while (x2 <= x)   {
         PlotPixel();
         if (f > 0)  {
            y++; f = f - ddx;
         }
         f = f + ddy;   x--;
      }
      x = x2;
   }
}

void QuadrantC()                                          // Bresenham's line drawing algorithm for DrawLine() function
{
   deltax = x1 - x2; ddx = 2 * deltax; deltay = y1 - y2; ddy = 2 * deltay;
   if (deltay > deltax) {
      f = ddx - deltay;
      while (y2 <= y)   {
         PlotPixel();
         if (f > 0)  {
            x--; f = f - ddy;
         }
         f = f + ddx; y--;
      }
      y = y2;
   }
   else  {
      f = ddy - deltax;
      while (x2 <= x)
      {
         PlotPixel();
         if (f > 0)  {
            y--; f = f - ddx;
         }
         f = f + ddy; x--;
      }
      x = x2;
   }
}

void QuadrantD()                                          // Bresenham's line drawing algorithm for DrawLine() function
{
   deltax = x2 - x1; ddx = 2 * deltax; deltay = y1 - y2; ddy = 2 * deltay;
   if (deltay > deltax)
   {
      f = ddx - deltay;
      while (y2 <= y)   {
         PlotPixel();
         if (f > 0)  {
            x++; f = f - ddy;
         }
         f = f + ddx; y--;
      }
      y = y2;
```

```
   }
   else  {
      f = ddy - deltax;
      while (x <= x2)   {
         PlotPixel();
         if (f > 0)  {
            y--; f = f - ddx;
         }
         f = f + ddy; x++;
      }
      x = x2;
   }
}

void DrawLine()         // (x1, y1) - (x2, y2) Line drawing routine
{
   x = x1;  y = y1;
   if (x1 <= x2)  {
      if (y1 <= y2)
      QuadrantA();
      else
      QuadrantD();
   }
   else  {
      if (y1 <= y2)
      QuadrantB();
      else
      QuadrantC();
   }
}

void WriteToTextRAM()
{
   output_C(make8(address, 0)); output_D(make8(address, 1));      // Assert Address lines
   while (input(PIN_A5)); while (!input(PIN_A5));                 // Wait for positive edge of H_BLANK
   output_high(pin_A3);                                           // Assert VIDEO_ADDRESS_ACCESS
   output_low(pin_A0);                                            // Assert !SELECT_VIDEO_TEXT_RAM
   output_B(data); output_low(pin_E0); output_high(pin_E0);       // Put byte onto data bus and strobe !WRITE
   output_high(pin_A0);                                           // Un-assert !SELECT_VIDEO_TEXT_RAM
   output_low(pin_A3);                                            // Un=assert VIDEO_ADDRESS_ACCESS
}

void WriteToCharacterRAM()
{
   output_C(make8(address, 0)); output_D(make8(address, 1));      // Assert Address lines
   while (input(PIN_A5)); while (!input(PIN_A5));                 // Wait for positive edge of H_BLANK
   output_high(pin_A3);                                           // Assert VIDEO_ADDRESS_ACCESS
   output_low(pin_A1);                                            // Assert !SELECT_VIDEO_CHAR_RAM
   output_B(data); output_low(pin_E0); output_high(pin_E0);       // Put byte onto data bus and strobe !WRITE
   output_high(pin_A1);                                           // Un-assert !SELECT_VIDEO_CHAR_RAM
   output_low(pin_A3);                                            // Un-assert VIDEO_ADDRESS_ACCESS
}

void ProgramCharacterRAM()                                        // Load custom set into Character RAM
{
   for (address = 0; address < 1024; address++) {
      data = usercharacterset[address]; WriteToCharacterRAM(); delay_ms(5);
```

```
   }
}

void ClearBitmapRAM()
{
   output_B(0);                                                  // Zero data bus
   for (address = 0; address <8000; address = address + 1)  {
      output_C(make8(address, 0)); output_D(make8(address, 1));  // Assert Address lines
      while (input(PIN_A5)); while (!input(PIN_A5));             // Wait for positive edge of H_BLANK
      output_high(pin_A3);                                       // Assert VIDEO_ADDRESS_ACCESS
      output_low(pin_A2);                                        // Assert !SELECT_BITMAP_MEMORY
      output_low(pin_E0); output_high(pin_E0);                   // Strobe !WRITE
      output_high(pin_A2);                                       // Un-assert !SELECT_BITMAP_MEMORY
      output_low(pin_A3);                                        // Un-assert VIDEO_ADDRESS_ACCESS
   }
}

void ClearTextRAM()
{
   output_B(32);                                                 // Blank space character code on data bus

   for (address = 0; address <1024; address = address + 1)  {
      output_C(make8(address, 0)); output_D(make8(address, 1));  // Assert Address lines
      while (input(PIN_A5)); while (!input(PIN_A5));             // Wait for positive edge of H_BLANK
      output_high(pin_A3);                                       // Assert VIDEO_ADDRESS_ACCESS
      output_low(pin_A0);                                        // Assert !SELECT_TEXT_MEMORY
      output_low(pin_E0); output_high(pin_E0);                   // Strobe !WRITE
      output_high(pin_A0);                                       // Un-assert !SELECT_TEXT_MEMORY
      output_low(pin_A3);                                        // Un-assert VIDEO_ADDRESS_ACCESS
   }
}

void ClearCharRAM()
{
   output_B(0);                                                  // Zero data bus
   for (address = 0; address <1024; address = address + 1)  {
      output_C(make8(address, 0)); output_D(make8(address, 1));  // Assert Address lines
      while (input(PIN_A5)); while (!input(PIN_A5));             // Wait for positive edge of H_BLANK
      output_high(pin_A3);                                       // Assert VIDEO_ADDRESS_ACCESS
      output_low(pin_A1);                                        // Assert !SELECT_CHAR_MEMORY
      output_low(pin_E0); output_high(pin_E0);                   // Strobe !WRITE
      output_high(pin_A1);                                       // Un-assert !SELECT_CHAR_MEMORY
      output_low(pin_A3);                                        // Un-assert VIDEO_ADDRESS_ACCESS
   }
}

void DrawBorderAndPyramid()
{
   x1 = 0; y1 = 0; x2 = 319; y2 = 0; DrawLine();
   y1 = 199; y2 = 199; DrawLine();
   x2 = 0; y1 = 0; DrawLine();
   x1 = 319; x2 = 319; DrawLine();

   x1=160; y1=100; y2=190;
   for (x2 = 9; x2 <311; x2 = x2 + 5)  {
      DrawLine(); }
}
```

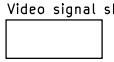
```
void PrintFullCharacterSet()
{
   for (count = 0; count < 32; count++)   {
      address = count + 124;  Data = count;        WriteToTextRAM();
      address = count + 164;  Data = count + 32;   WriteToTextRAM();
      address = count + 204;  Data = count + 64;   WriteToTextRAM();
      address = count + 244;  Data = count + 96;   WriteToTextRAM();
      address = count + 284;  Data = count + 128;  WriteToTextRAM();
      address = count + 324;  Data = count + 160;  WriteToTextRAM();
      address = count + 364;  Data = count + 192;  WriteToTextRAM();
      address = count + 404;  Data = count + 224;  WriteToTextRAM();
   }
}

void InitialiseIO()
{
   set_tris_A(0x20); set_tris_B(0x00); set_tris_C(0x00);          // Setup IO ports
   set_tris_D(0x00); set_tris_E(0x00);                            //    "

   output_high(pin_A0);                                           // !SELECT_VIDEO_TEXT_RAM high
   output_high(pin_A1);                                           // !SELECT_VIDEO_CHAR_RAM high
   output_high(pin_A2);                                           // !SELECT_VIDEO_BITMAP_RAM high
   output_low(pin_A3);                                            // VIDEO_ADDRESS_ACCESS low
   output_low(pin_A4);                                            // GRAPHICS/BUSINESS low
   output_high(pin_E0);                                           // !WRITE high
   output_high(pin_E1);                                           // !READ high
   output_low(pin_E2);                                            // ENABLE_USER_CHAR low
}

void MAIN()
{
   // RA0 (PIN2)  = !SELECT_TEXT_MEMORY (Output)
   // RA1 (PIN3)  = !SELECT_CHAR_MEMORY (Output)
   // RA2 (PIN4)  = !SELECT_BITMAP_MEMORY (Output)
   // RA3 (PIN5)  = VIDEO_ADDRESS_ACCESS (Output)
   // RA4 (PIN6)  = GRAPHICS/BUSINESS (Output)
   // RA5 (PIN7)  = HOR_BLANKING (Input)
   // RE0 (PIN8)  = !WRITE (Output)
   // RE1 (PIN9)  = !READ (Output)
   // RE2 (PIN10) = ENABLE_USER_CHAR (Output)
   // RC0 - RC7   = ADDRESS_0 - ADDRESS_7 (Output)
   // RD0 - RD4   = ADDRESS_8 - ADDRESS_12 (Output)
   // RB0 - RD7   = DATA_0 - DATA_7 (Bidirectional)

   InitialiseIO();

   delay_ms(2000); ClearTextRAM(); ClearBitmapRAM(); ClearCharRAM();          // Clear video RAMs
   delay_ms(1000); DrawBorderAndPyramid(); delay_ms(1000);                    // Draw bitmap graphics

   For (count = 0; count < 22; count++)  {                                    // Graphics character set display routine
      data = GRAPHICS_CHARACTER_SET[count]; address = count + 49;
      WriteToTextRAM(); delay_ms(100);
   }
   delay_ms(1000); PrintFullCharacterSet(); Delay_ms(5000); ClearTextRAM();

   delay_ms(1000); output_high(pin_A4);                                       // Select BUSINESS character set
```

```
   For (count = 0; count < 22; count++)  {                                  // Business character set display routine
      data = Business_character_set[count]; address = count + 49;
      WriteToTextRAM(); delay_ms(100);
   }
   delay_ms(1000); PrintFullCharacterSet(); Delay_ms(5000);

   Data = 32;                                                               // User-defined character set display routine
   For (address = 49; address < 71; address++)   {
      WriteToTextRAM();
   }
   output_high(pin_E2); Delay_ms(1000);                                     // Enable user-defined characters
   For (count = 0; count < 23; count++)  {
      data = User_defined_characters[count]; address = count + 49;
      WriteToTextRAM(); delay_ms(100);
   }
   delay_ms(1000); ProgramCharacterRAM(); Delay_ms(5000);

   ClearBitmapRAM(); Delay_ms(1000); ClearTextRAM();                        // Finish by blanking display
}
```

1) PET "3096" EXTENDED GRAPHICS COMPUTER

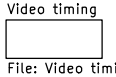
VIDEO GENERATOR SECTION

Video signal shift register and composite video output



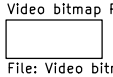
File: Video signal shift register and composite video output.kicad_sch

Video timing



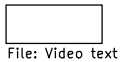
File: Video timing.kicad_sch

Video bitmap RAM address counter



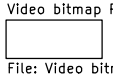
File: Video bitmap RAM address counter.kicad_sch

Video text RAM address counter



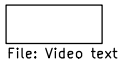
File: Video text RAM address counter.kicad_sch

Video bitmap RAM



File: Video bitmap RAM.kicad_sch

Video text RAM



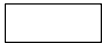
File: Video text RAM.kicad_sch

Video character ROM and RAM

File: Video character ROM and RAM.kicad_sch

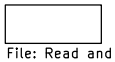
PROCESSOR SECTION

CPU



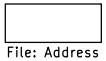
File: CPU.kicad_sch

Read and write strobing



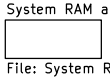
File: Read and write strobing.kicad_sch

Address decoding and memory control registers



File: Address decoding.kicad_sch

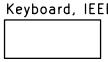
System RAM and ROM



File: System RAM and ROM.kicad_sch

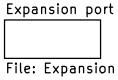
INPUT AND OUTPUT SECTION

Keyboard, IEEE-488 and User ports



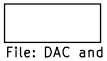
File: Keyboard, IEEE-488 and User ports.kicad_sch

Expansion port



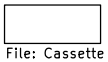
File: Expansion port.kicad_sch

DAC and CB2 audio



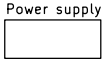
File: DAC and CB2 audio.kicad_sch

Cassette interface



File: Cassette interface.kicad_sch

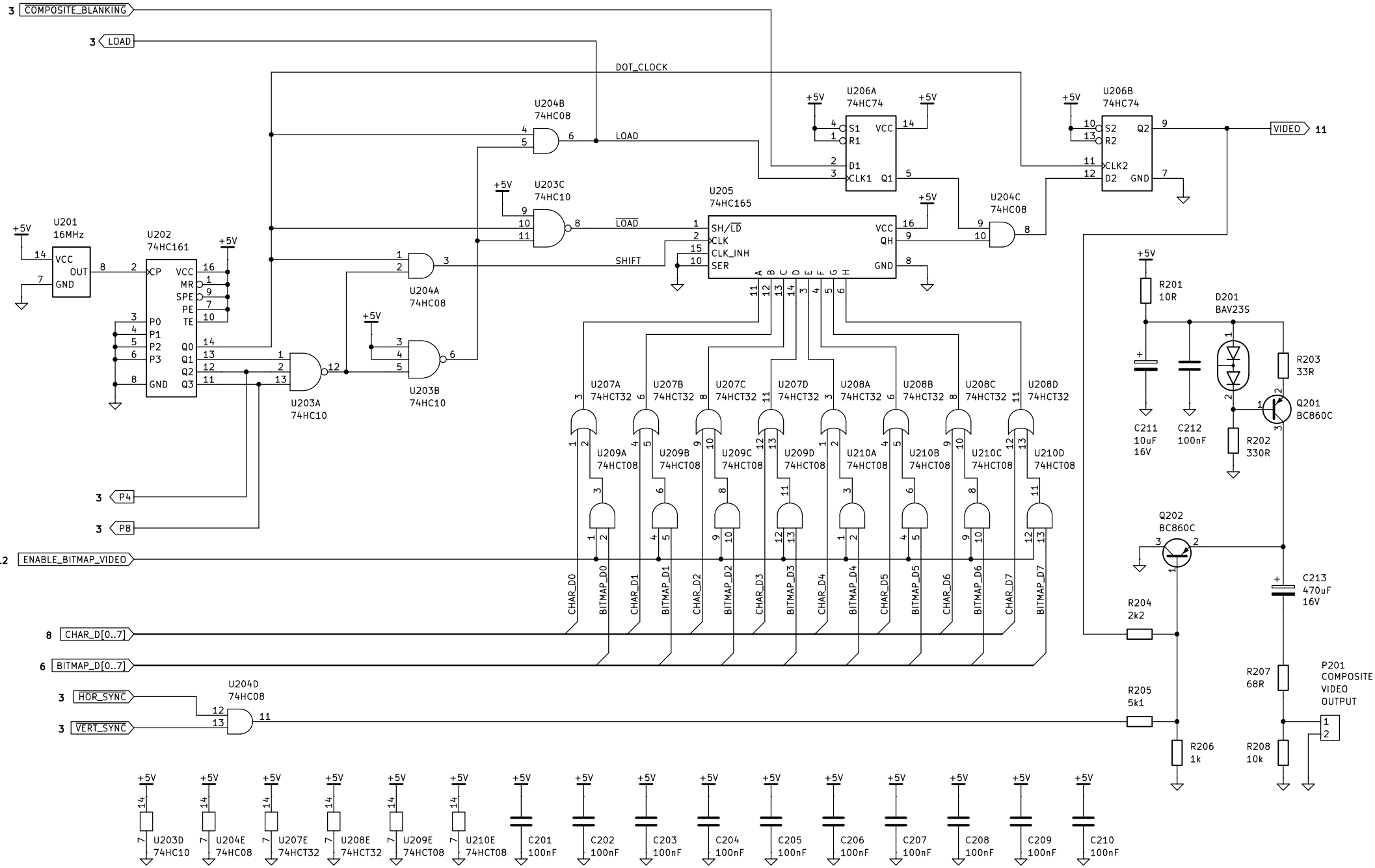
Power supply



File: Power supply.kicad_sch

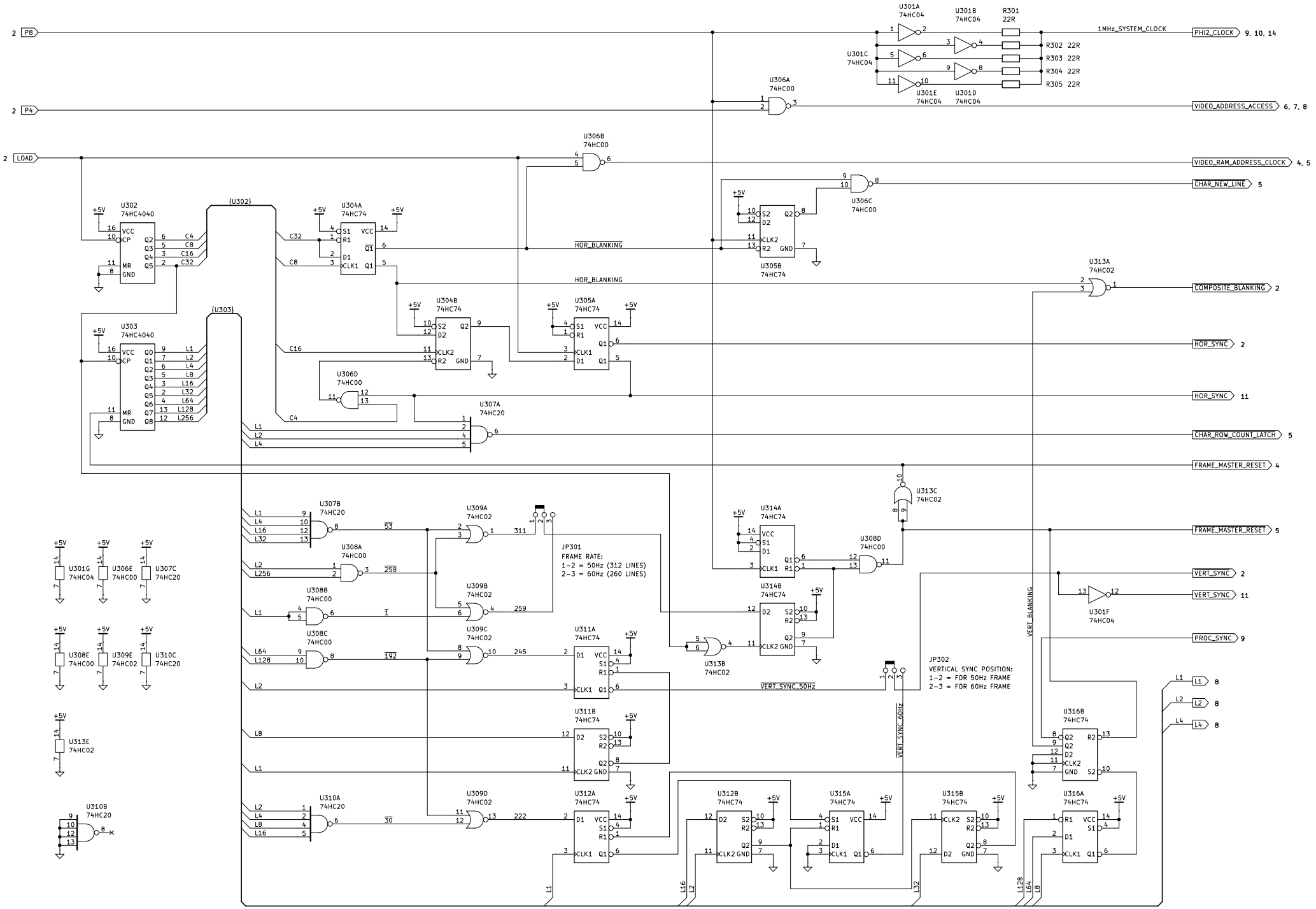
2) VIDEO GENERATOR SECTION

VIDEO SIGNAL SHIFT REGISTER AND COMPOSITE VIDEO OUTPUT



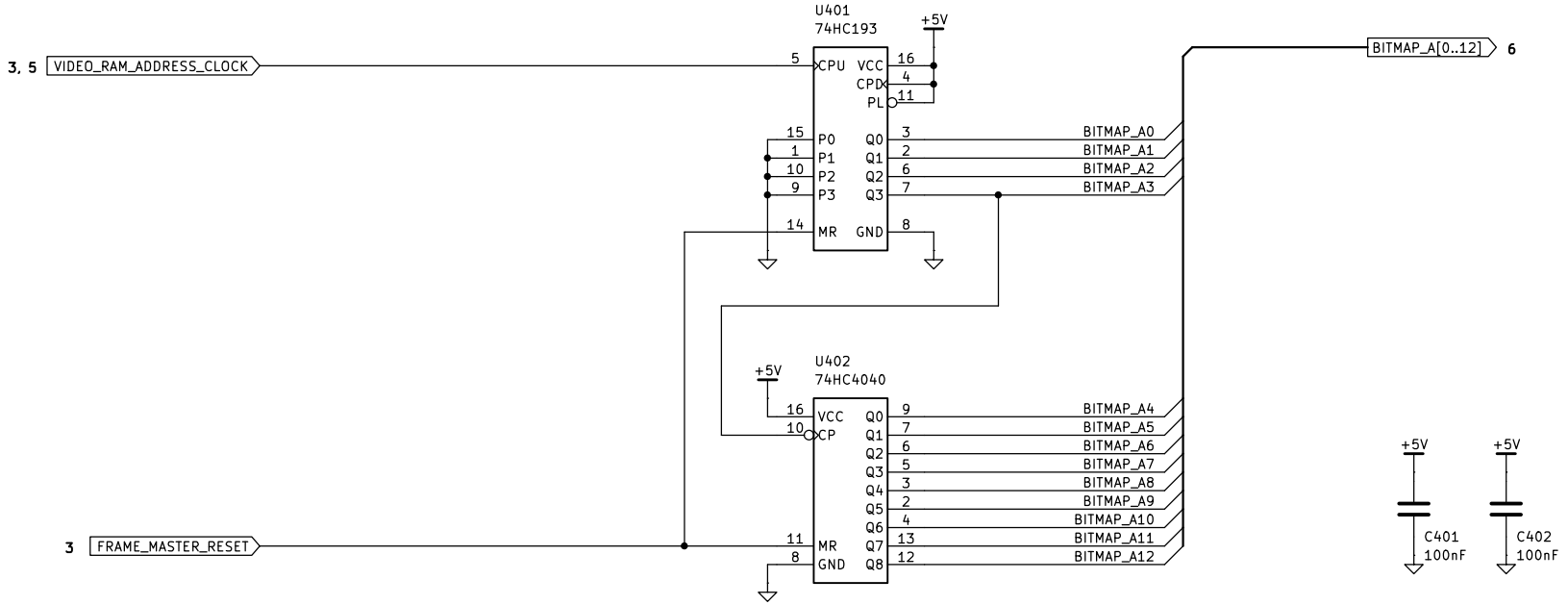
3) VIDEO GENERATOR SECTION

VIDEO TIMING



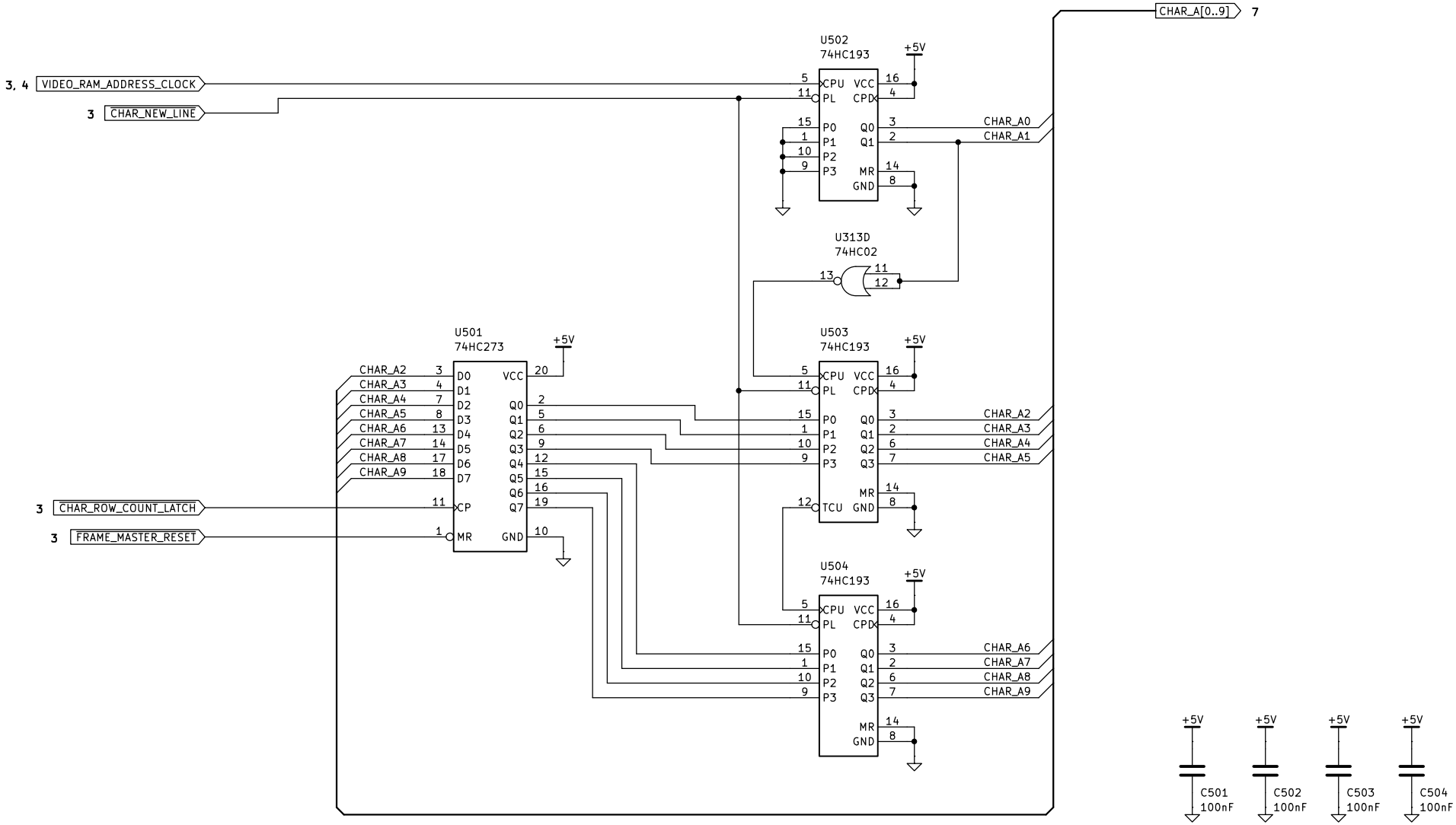
4) VIDEO GENERATOR SECTION

VIDEO BITMAP RAM ADDRESS COUNTER



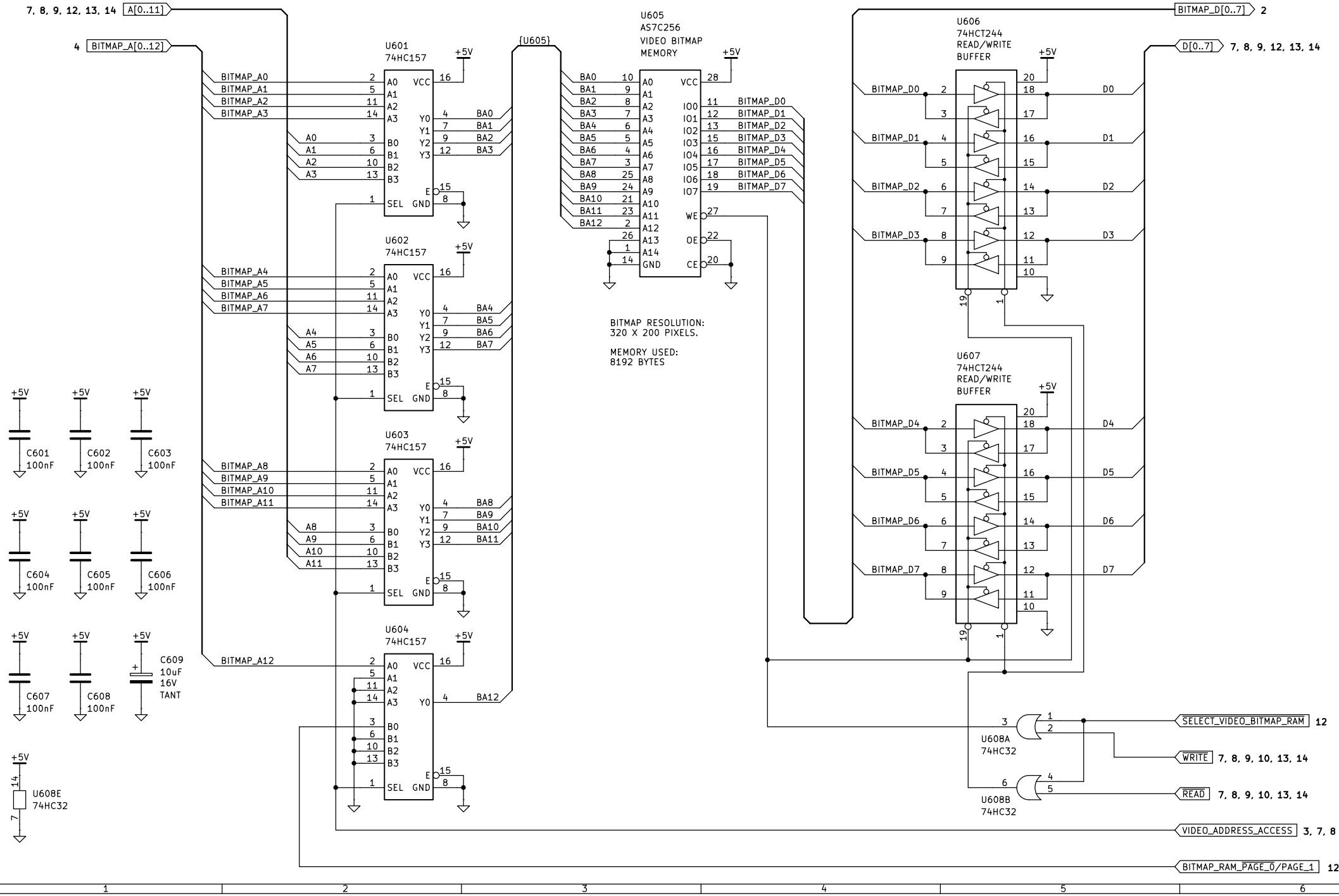
5) VIDEO GENERATOR SECTION

VIDEO TEXT RAM ADDRESS COUNTER



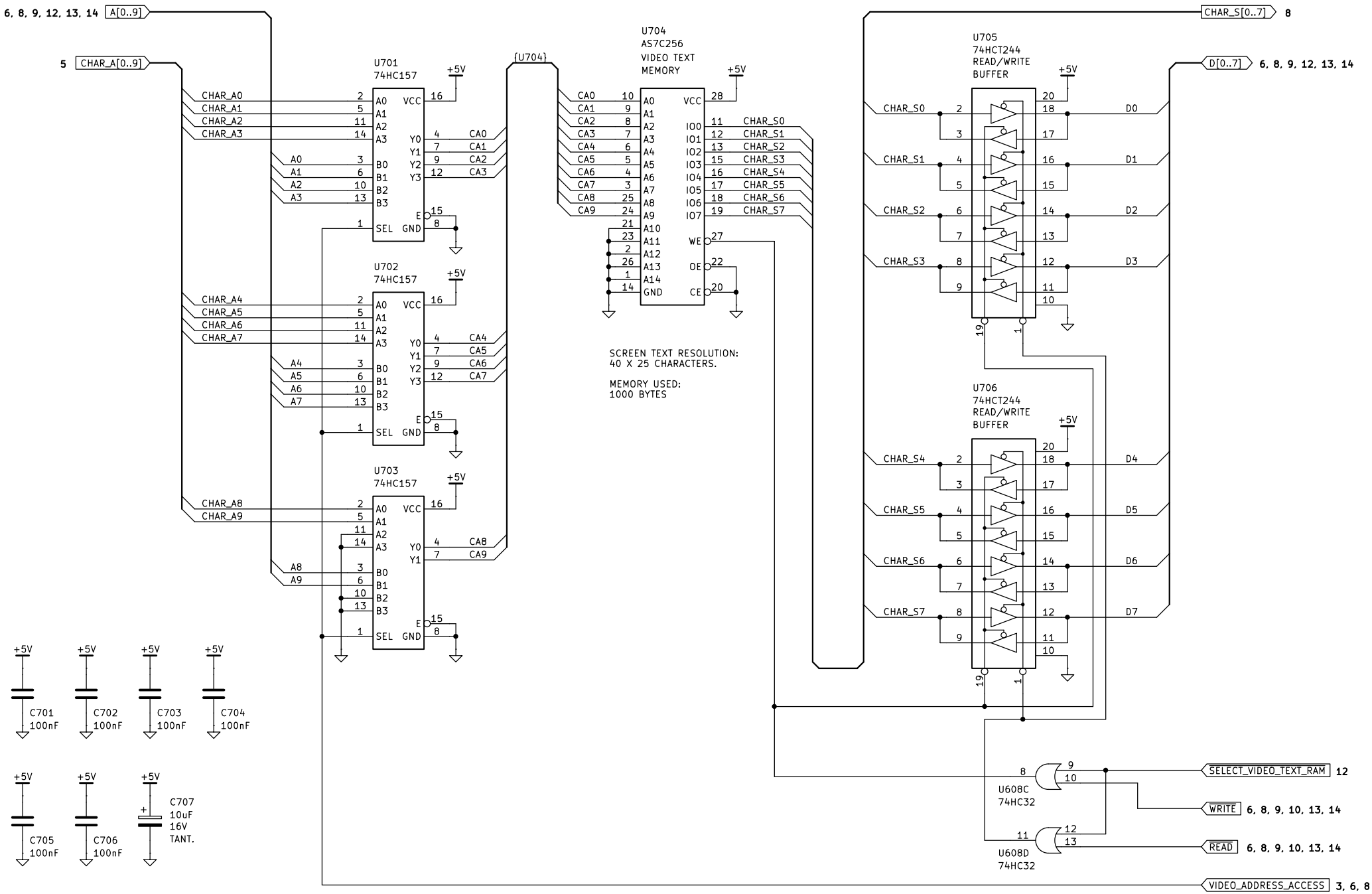
6) VIDEO GENERATOR SECTION

VIDEO BITMAP RAM



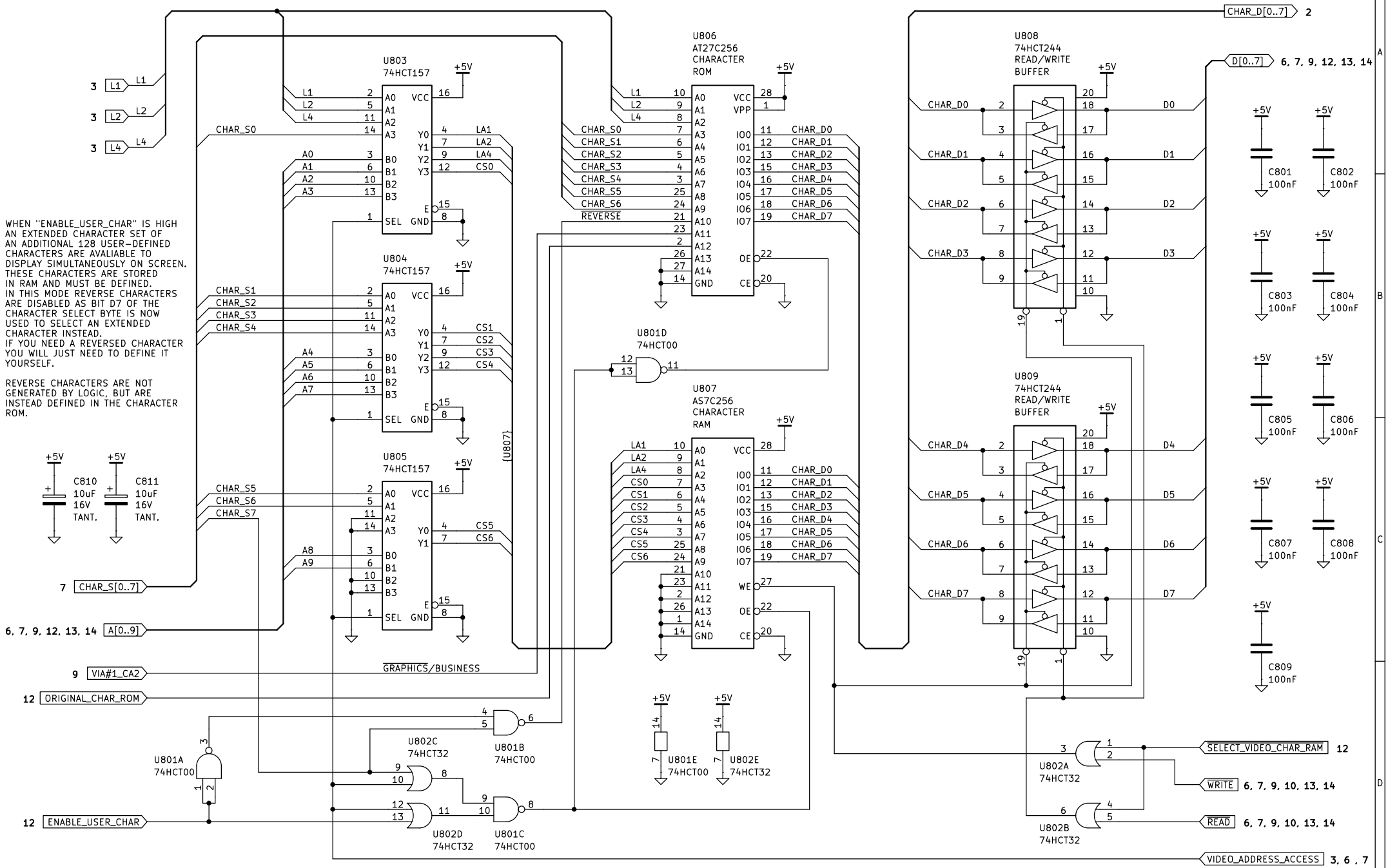
7) VIDEO GENERATOR SECTION

VIDEO TEXT RAM



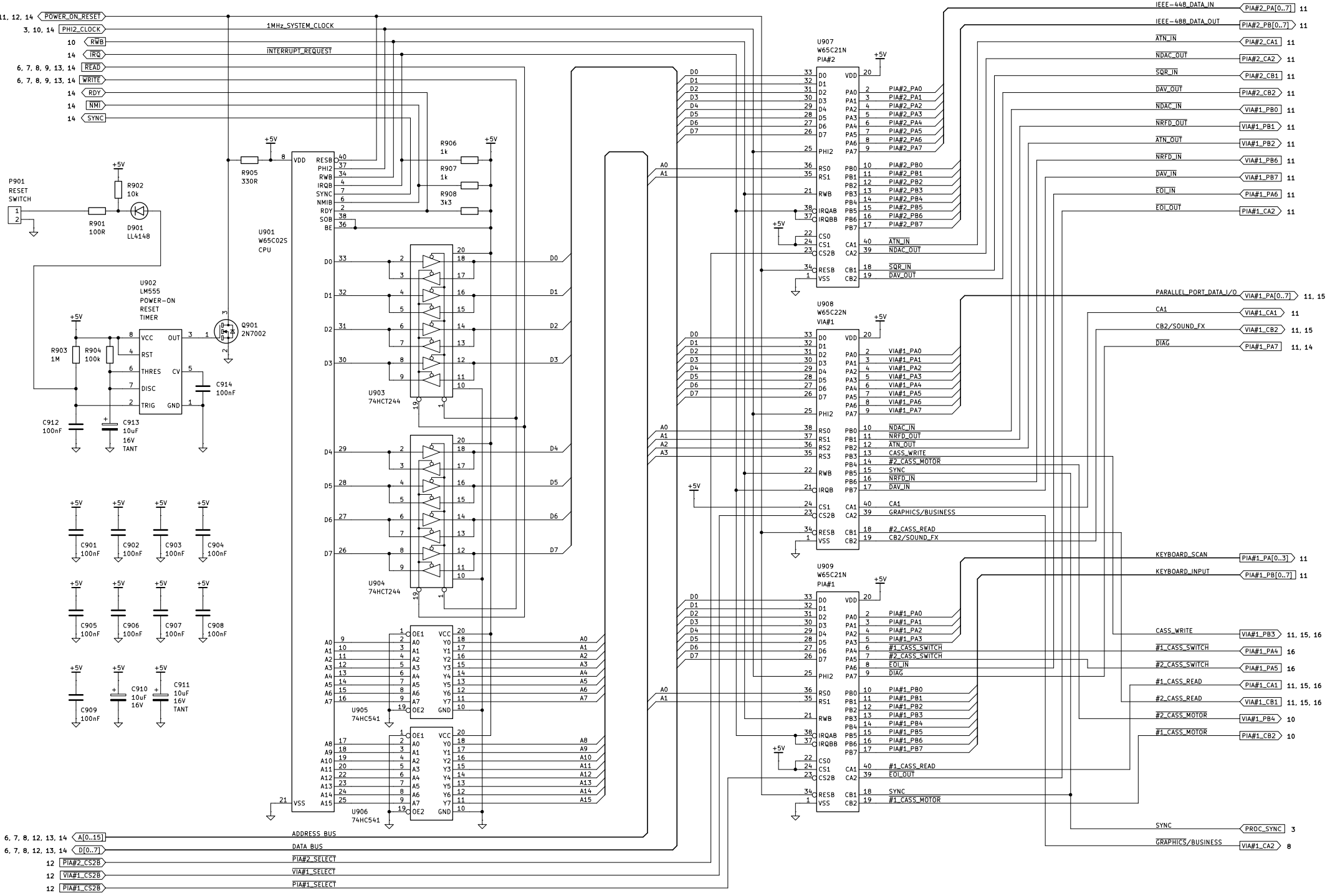
8) VIDEO GENERATOR SECTION

VIDEO CHARACTER ROM AND RAM



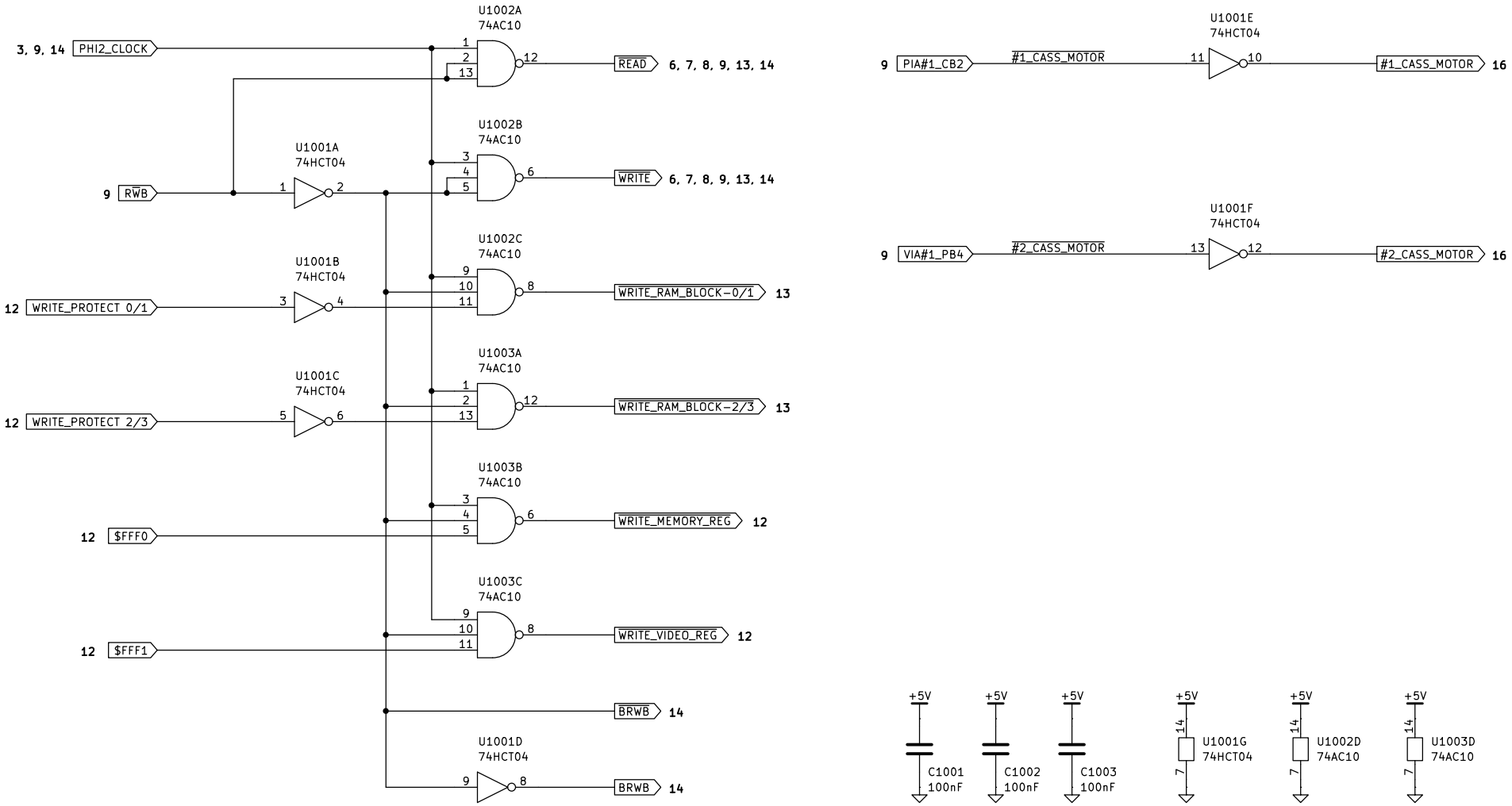
9) PROCESSOR SECTION

CPU



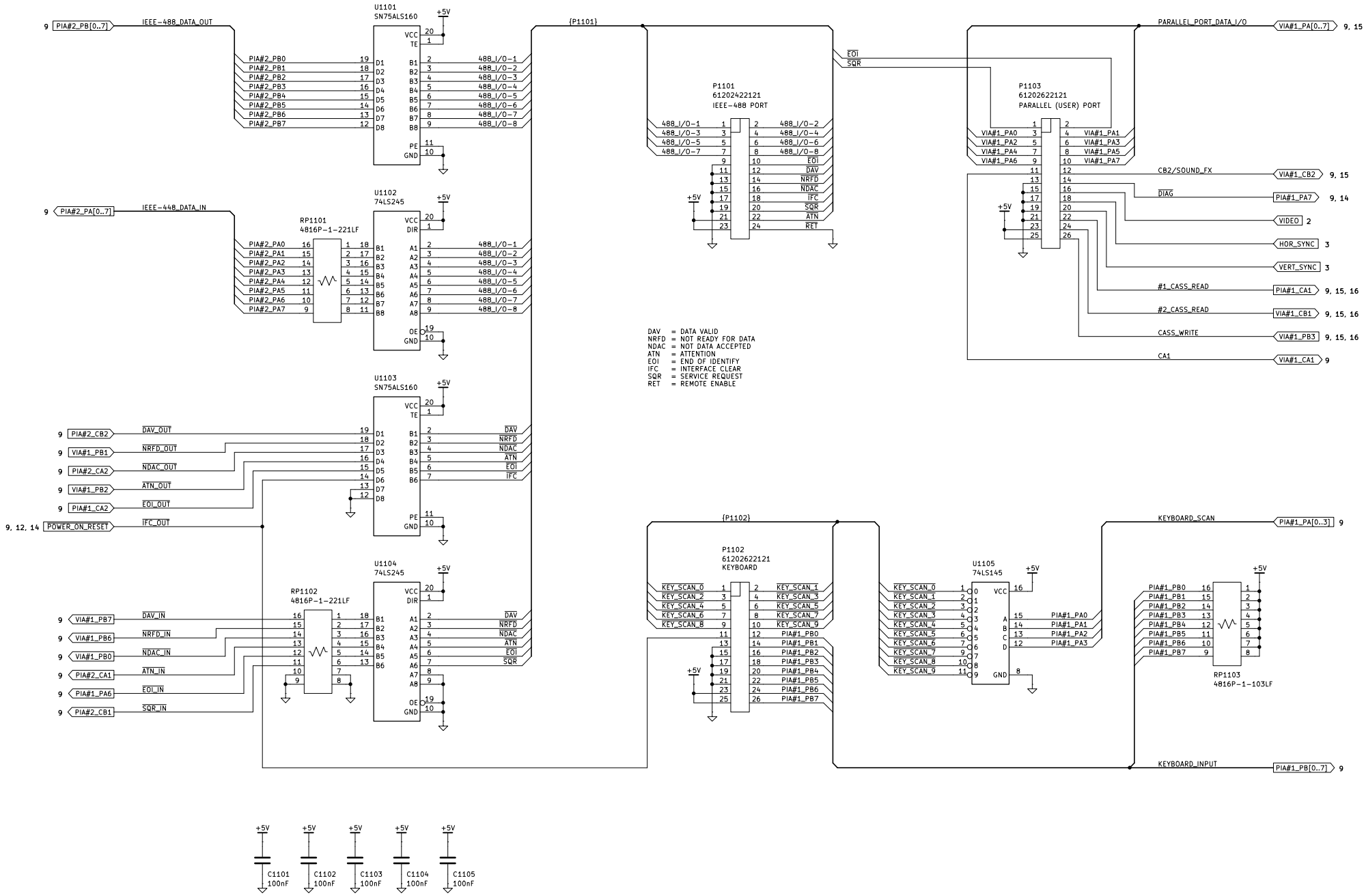
10) PROCESSOR SECTION

READ AND WRITE STROBING



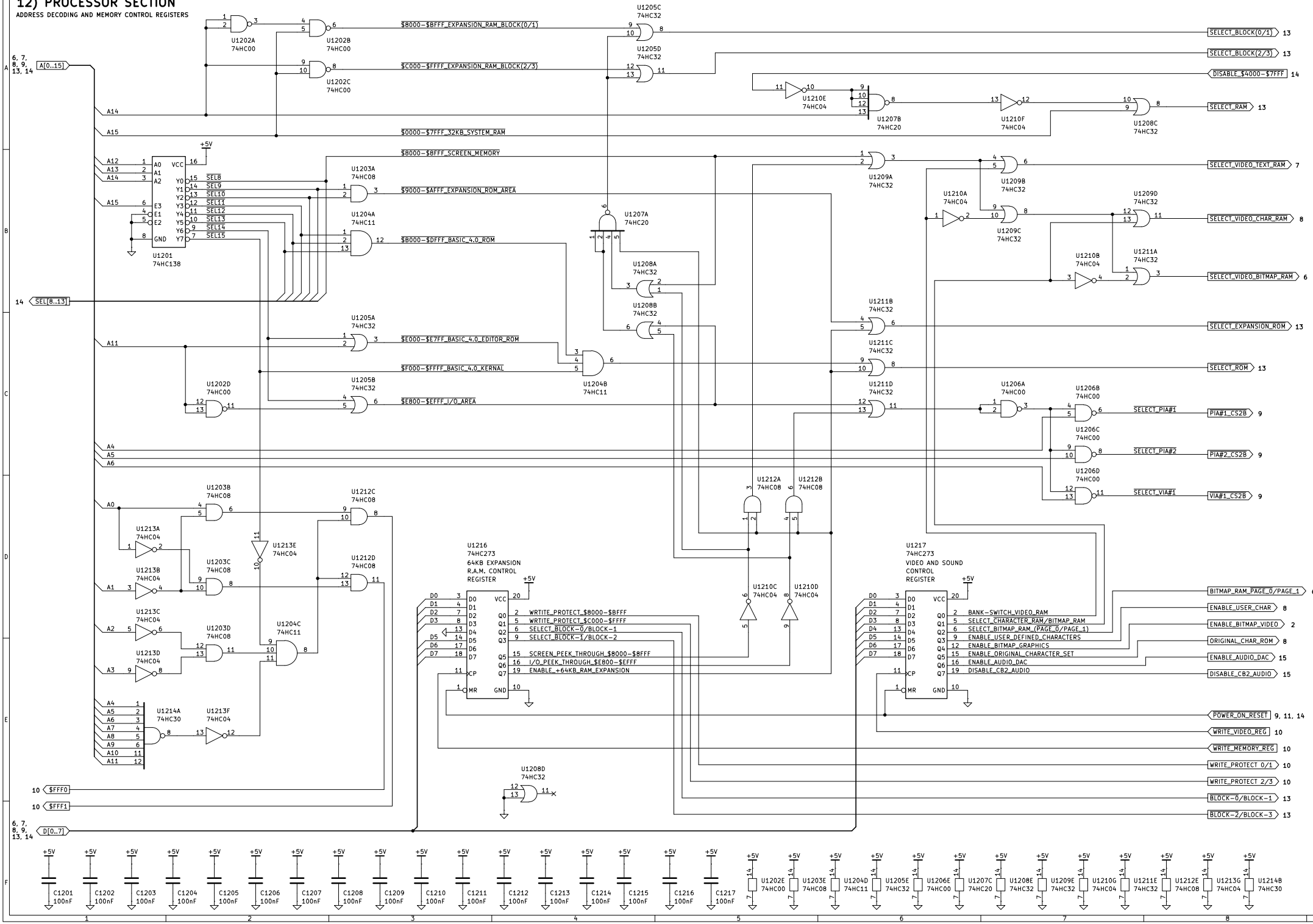
11) INPUT AND OUTPUT SECTION

KEYBOARD, IEEE-488 AND USER PORTS



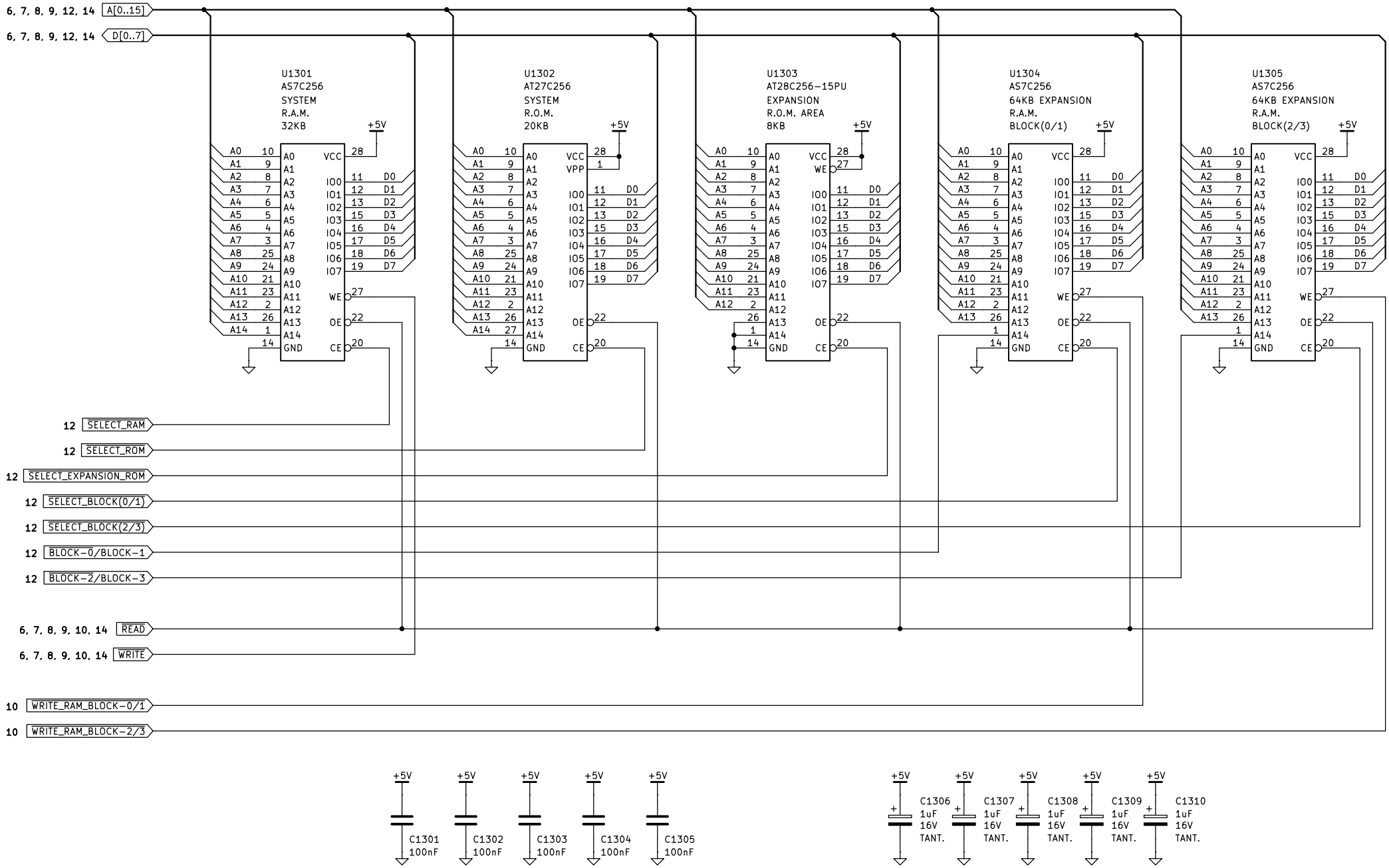
12) PROCESSOR SECTION

ADDRESS DECODING AND MEMORY CONTROL REGISTERS



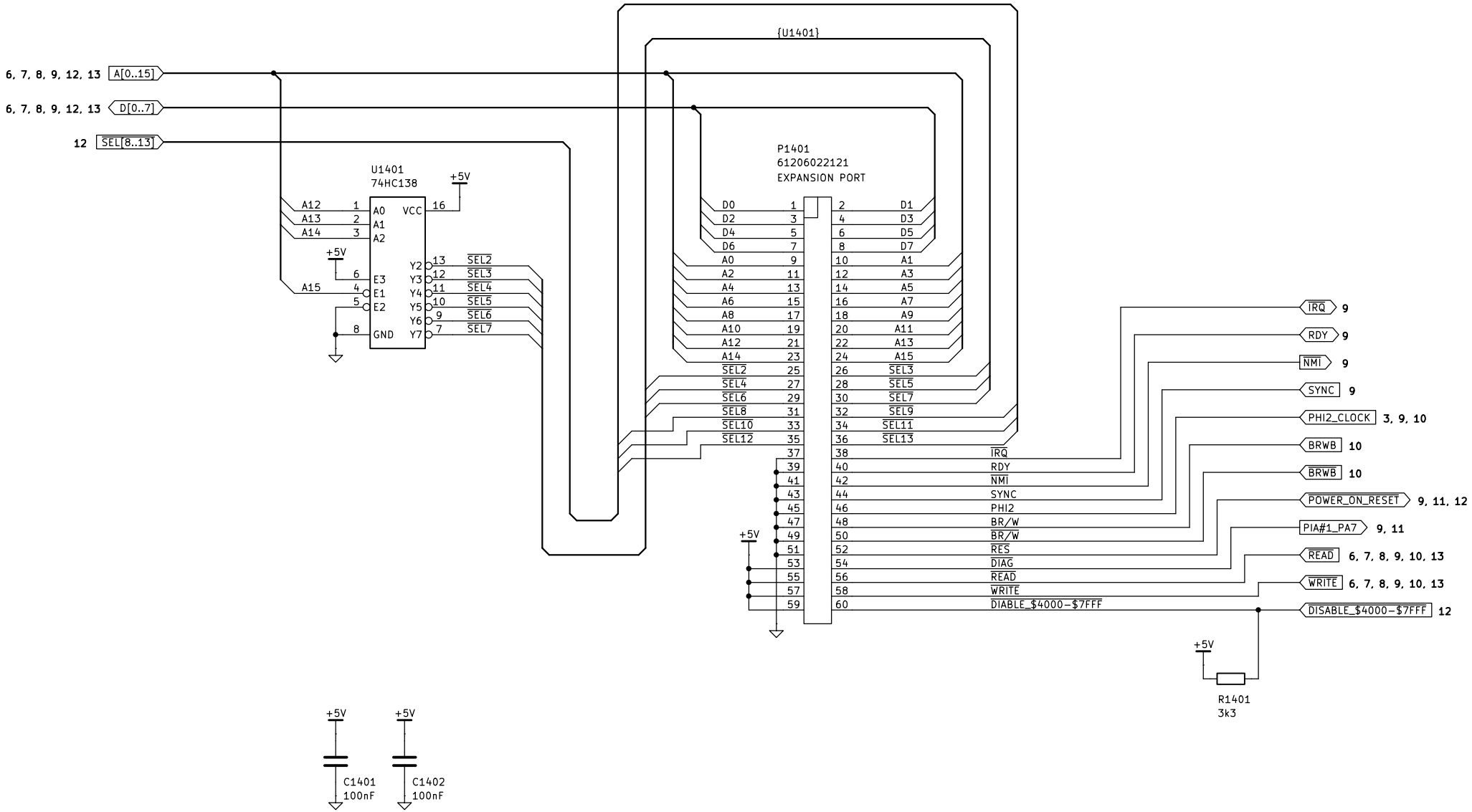
13) PROCESSOR SECTION

SYSTEM RAM AND ROM



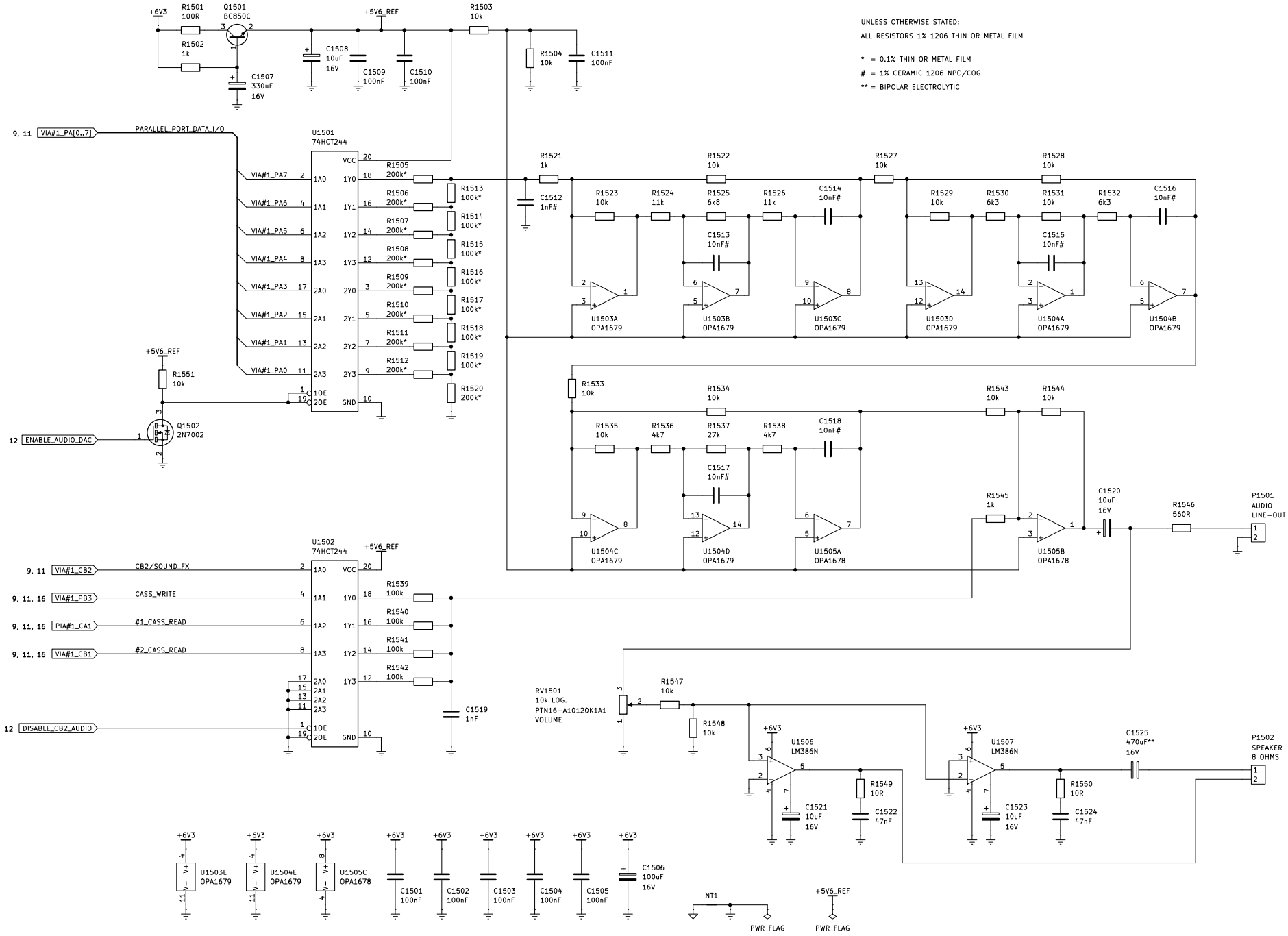
14) INPUT AND OUTPUT SECTION

EXPANSION PORT



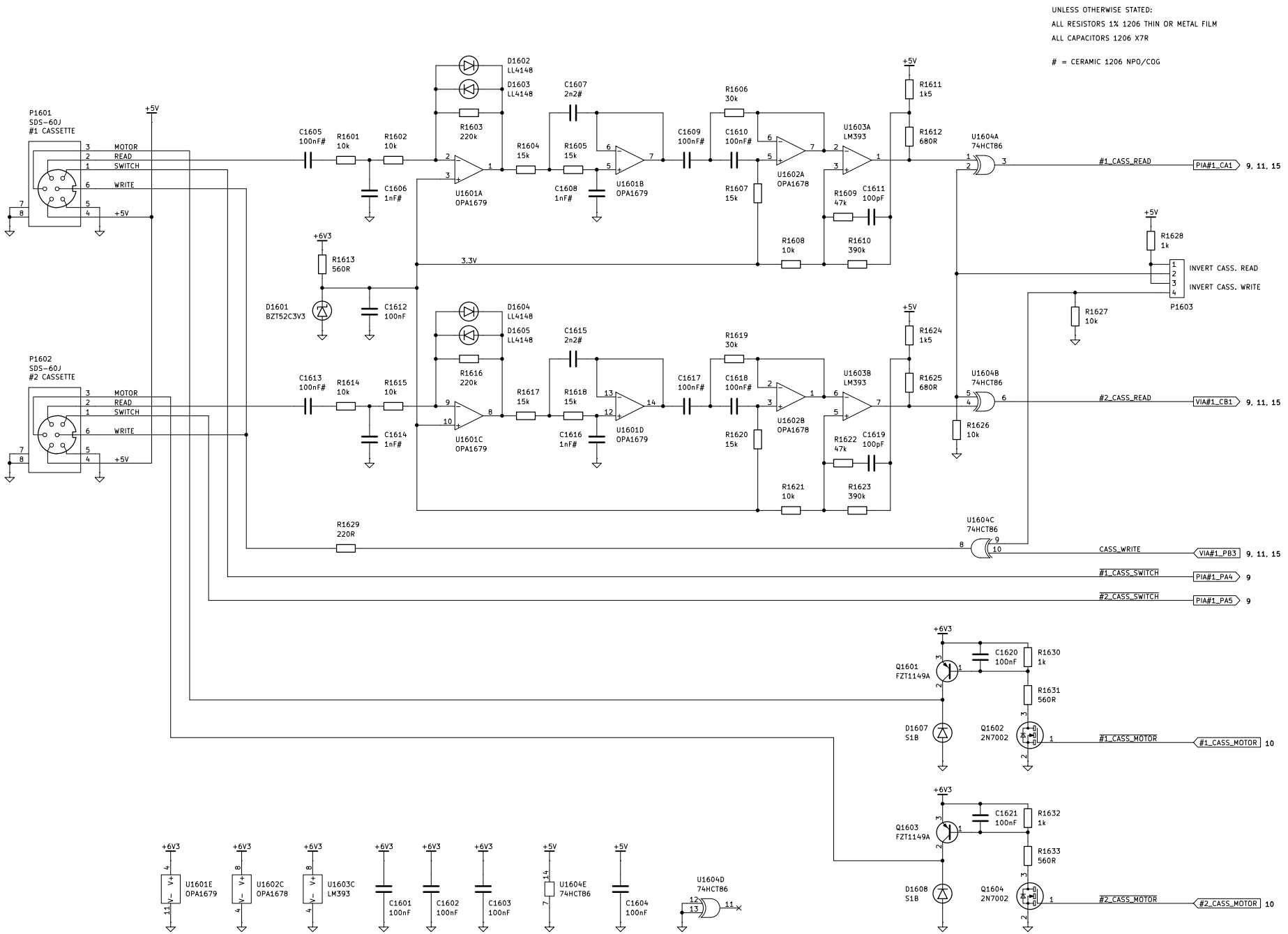
15) INPUT AND OUTPUT SECTION

DAC AND CB2 AUDIO



16) INPUT AND OUTPUT SECTION

CASSETTE / ANALOGUE DATA INTERFACE



17) POWER SUPPLY