Apple][FPGA replica

I must start stating I hate any modern Apple product. You pay double for half the performance and get tied to a single company that is always looking for new ways to make their products incompatible with anything else, even with older versions of their own products. Said that, I must also admit one exception: the Apple II computer. This old 8-bit machine was where I started programming, first in BASIC, then in assembler, and compared to other similar computers of its time the fact it used floppy disks instead of cassette tapes made it the most hi-tech computer I could put my hands on (well, I also got access to a bulky VAX 11-780 with VT100 terminals, but that was a completely different story). And it was also a quite well documented machine with lots of 3rd-party hardware and software suppliers, something never happened later on that company. I guess that was due to Wozniak still having some weight on the decisions taken by Apple. So, after the FPGA recreation of the ZX Spectrum and the Amstrad CPC it was the time for the Apple II. I must admit this design still needs a lot of debugging and rethinking, but it had to be minimally documented before forgetting its details.

So, lets start presenting some of the peculiarities of this computer, like the use and abuse of NE555s, and also of addressable latches that resulted in many output ports being totally controlled by the address bus only (it seems that Woz had something against the R/W signal of the 6502...).

1 Apple II features

- 6502 NMOS CPU clocked at an average 1.0205MHz (the clock isn't strictly periodic).
- 48 KByte of DRAM memory. This memory is accessed at twice the clock frequency: one cycle is for the CPU and the next for the video refresh logic. There is no memory contention between CPU and video. The video controller provides the needed refresh for the DRAM.
- A TTL-based, 1 bit per pixel, video controller with text and graphics modes. In spite of being inherently a monochrome video signal it can also display colors in a NTSC monitor via color subcarrier modulation.
- A keyboard scanned and encoded by an specific keyboard controller that generates ASCII codes for keypresses.
- 1 bit, software generated, audio output.
- A cassette interface, also handled by software.
- An analog interface for two joysticks (or paddles, as stated) that allows the measurement of 4 variable resistors via pulse length modulation (using a quad NE555, of course).
- 8 expansion slots. These slots allows the connection of more peripheral cards. It was very common to have a floppy disk controller installed in slot #6 (and to forget about cassettes completely).

1.1 Video generation

The video controller in the Apple II has three modes:

- A text mode with 40x24 displayed characters. Each character is 7 pixels wide by 8 lines high and is stored in an specific ROM memory. For older models this was a 512 x 5 ROM, providing up to 64 characters: 32 uppercase letters and 32 symbols plus digits. The character pixels can be inverted outside the ROM via a XOR gate controlled by the two most significant bits of the video data and a 2Hz square wave generator (again, based on the NE555 IC) that makes the selected characters to blink. Later models, like the Apple IIe, included a 2048 x 8 ROM with normal and inverted characters, and also with lowercase letters. In text mode the video memory is located at address $400 or $800.

- A "high resolution" graphics mode where the data to be sent to the display comes directly from RAM instead of the character generator ROM. This mode has a resolution of 280 x 192 pixels and 40 bytes per horizontal line. But wait a minute, 40 bytes are 320 bits, not 280. Where are the remaining 40 bits? The answer is that only the 7 least significant bits of each byte are displayed (LSB on the left), the eighth bit is used to delay the next 7 pixels half a video clock cycle (or 69.84 ns. This can have an effect on color monitors, more about this later) Video memory is located at address $2000 or $4000.

- A "low resolution" graphics mode with 40 x 48 chunky blocks. This mode was almost useless for programmers, but it provided up to 16 colors on color monitors. (I'll bet Woz included this mode after thinking about the "Pong" game) Video memory is located at address $400 or $800, the same area as text.

All modes can have their video memory mapped to two different areas (PAGE 1 or PAGE 2) controlled by a write-only bit in an addressable latch. Also, the graphic and text modes can be mixed together, resulting in a 280 x 160 graphic window and a 40 x 4 character text at the bottom of the screen.

And, what about color? The Apple II video is just a monochrome, 1-bit per pixel, signal but the pixel clock is exactly 2 times the NTSC color subcarrier (7.159MHz). This means that an alternating pattern of ON-OFF pixels is creating the color subcarrier. Also, a color burst signal is included at the beginning of video lines in graphics modes (in text mode the color burst was suppressed on later revisions of the Apple II, the older ones showed a colored text). Therefore, on color NTSC monitors the Apple II is able to create up to six different colors using its single bit per pixel monochrome signal:

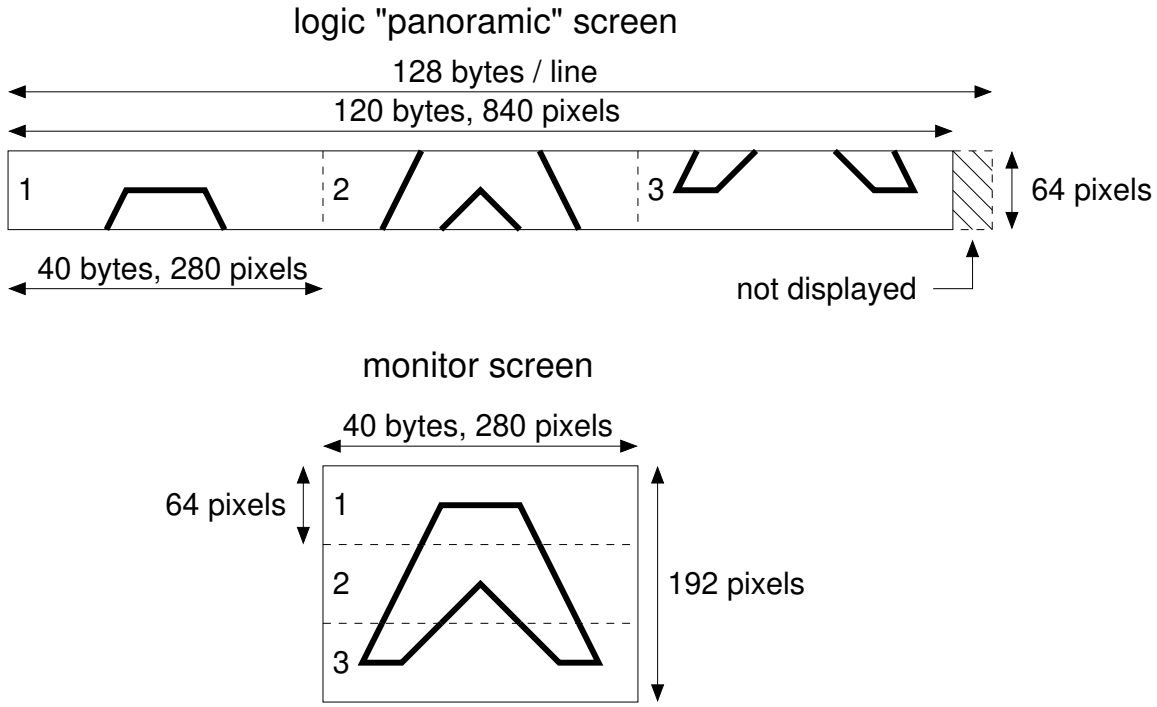
even pixel	odd pixel	half-clock delay	color
0	0	x	Black
0	1	0	Green
1	0	0	Purple
0	1	1	Orange
1	0	1	Blue
1	1	x	White

The half clock delay has the effect of generating the color subcarrier in quadrature with respect the color burst tone and two different colors more. Of course, the half clock delay also means some artifacts on the undelayed-delayed byte boundaries, like different pixel lengths and unexpected colors.

The video modes are selected using four bits of an addressable latch where the data written into the latch is the lowest address bit, A0. Therefore, a memory read or write done to an odd address sets some bit while the same read or write into the preceding, even address, clears the bit. Typically an "LDA $C05x" instruction is used to change one of these bits. The addresses are:

Address	Switch	Address	Switch
$C050	Graphics mode	$C051	Text mode
$C052	No mix	$C053	Mixed Graphics / Text
$C054	PAGE 1	$C055	PAGE 2
$C056	Low-Res	$C057	Hi-Res

Video memory addresses aren't sequential, of course. It is a good idea to think the screen is 120 x 8 characters wide instead of 40 x 24, but divided into three bands of 40 x 8 characters and these bands displayed into the upper, middle, and lower part of the screen:



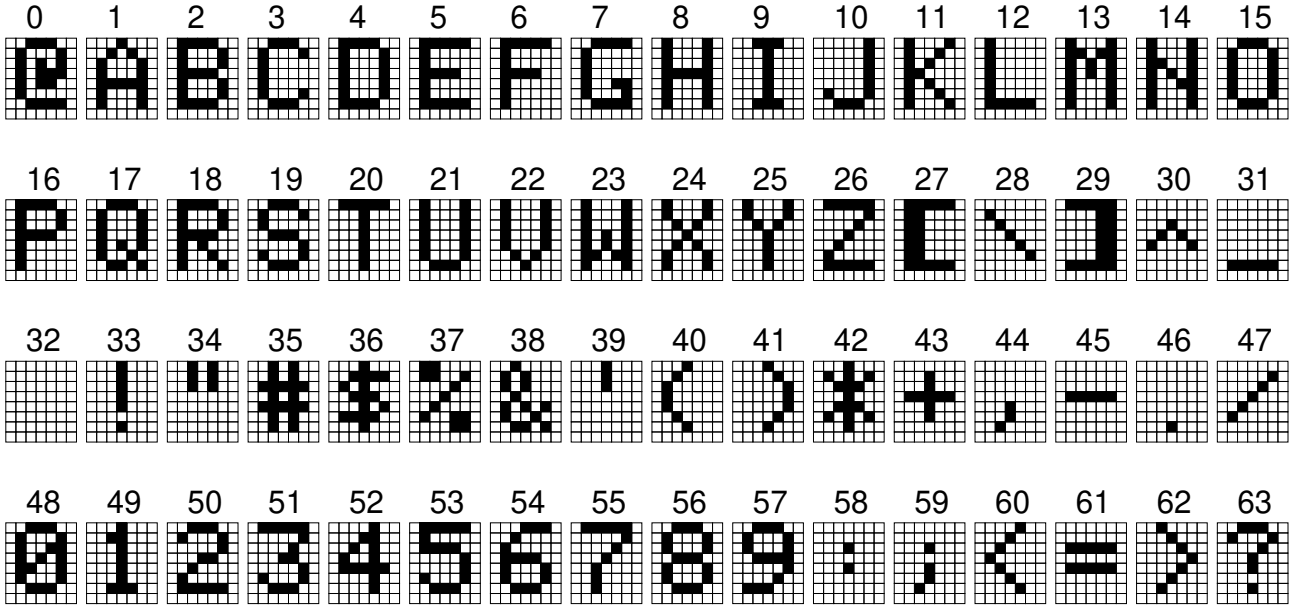
One line of this "panoramic" screen shows 120 characters, but it takes 128 memory bytes, so, the second line starts at offset #128 and there are some video bytes hidden (8 per character line). In high resolution graphics mode the address offsets are the same but the whole scheme is repeated 8 times. In summary, the address offset in Hi-res mode is:

$$Offset = (X/7) + (Y/8) * 128 + (Y\%8) * 1024$$

where X and Y are coordinates of the "panoramic" screen, related with the actual screen coordinates, x and y, by:

$$X = x + (y/64) * 280 \text{ , } Y = y\%64$$

In text mode the bits 5 to 0 of the video bytes addresses the character generator ROM, resulting in the following displayed characters:

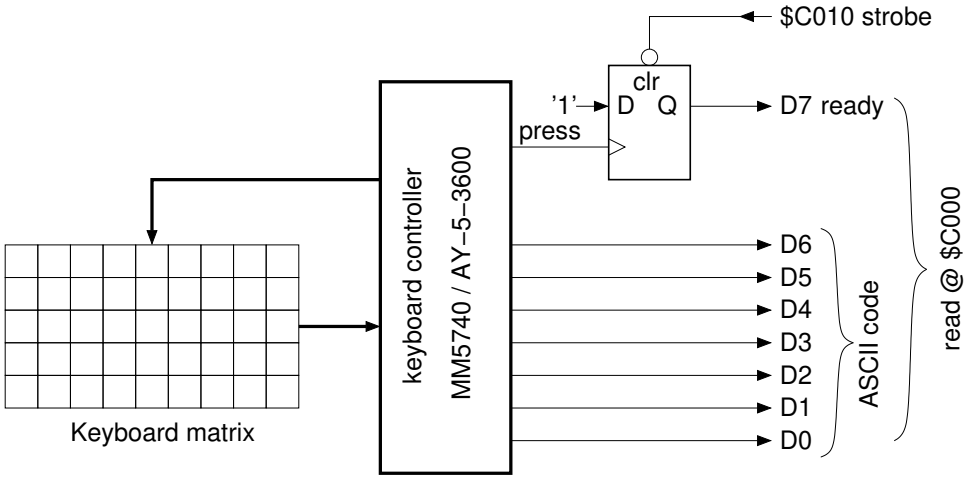


The bits 7 and 6 of the video bytes are used for character attributes:

bit 7	bit 6	Attribute
0	0	Inverse
0	1	Blink
1	x	Normal

1.2 Keyboard

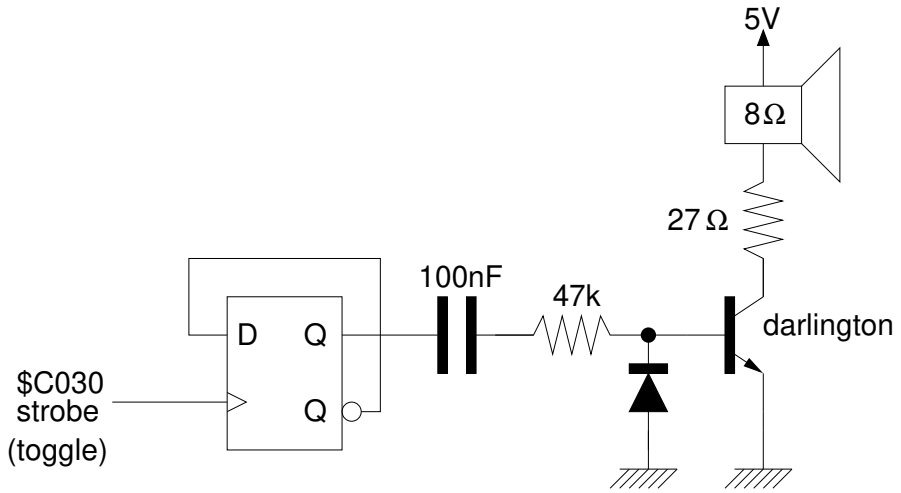
In contrast with many 8-bit computers of the time where the main CPU was actively scanning the keyboard, the Apple II includes an special-purpose controller for the keyboard: the MM5740 from National Semiconductor or the AY-5-3600 from General Instruments, depending on hardware revision. These controllers deal with modifier keys (shift, control,...) and generate a 7-bit ASCII code for the key or combination of keys pressed along with a "strobe / data ready" signal. The functional diagram of the keyboard interface is shown next:



In the Apple II the data ready signal is presented as the bit 7 at the memory address $C000, while the ASCII code of the key is presented on the lower bits. The CPU just has to wait until bit 7 at address $C000 goes high, store the ASCII code, and then reset the ready signal by means of a read or write access at address $C010.

The keyboard interface is so simple from the CPU point of view that is easy to replace the whole keyboard by a serial port, for instance (this was done in the FPGA replica)

1.3 Audio



The audio output of the Apple II is just a flip-flop that toggles every time the CPU reads the address $C030. This flip-flop switches a power transistor on and off creating rectangular voltage waves at the speaker terminals. A series capacitor prevents the transistor from remaining in the on state for too long: after 20ms on the transistor goes off even if the flip-flop output is high. In the on state the current through the speaker is limited by the 27Ω series resistor to 140mA. This results in 40mW of AC power at the speaker for square waves. In spite of this small power the Apple II speaker was quite loud (and there was no volume control)

Certainly, this circuit, while functional and quite common for the computers of that era, can be easily improved. Its most disgusting aspect is the DC current that flows through the speaker. For this small power it is fine, but it should be avoided. Only AC current should flow through the speaker.

The cassette output is basically the same but with a voltage divider instead of a power amplifier. The cassette output flip-flop toggles when the CPU reads the address $C020.

1.4 Other motherboard peripherals

These are mostly related to the paddle interface. Paddles are a sort of analog "joysticks" with variable resistors and buttons, but the connector also includes digital "anunciator" outputs, and even a pulsed strobe signal. Digital inputs are read on different addresses and the data only has the bit 7 as valid. This configuration was intended for the use of the "BIT" instruction. The measurement of the variable resistors starts with a read instruction to address $C070. This triggers four NE555 timers, that set their outputs to high for a time proportional to the value of the measured resistors. The maximum value of the paddle resistor is $150k\Omega$, that results in a pulse length of about 3.3ms. The CPU must stay in a loop incrementing a result value as long as the related timer output is high.

The addresses of these I/O ports are:

Address	Input (bit #7)
$C060	Cassette input
$C061	Paddle button 0
$C062	Paddle button 1
$C063	-
$C064	Timer 0 (paddle resistor ADC)
$C065	Timer 1 (paddle resistor ADC)
$C066	Timer 2 (paddle resistor ADC)
$C067	Timer 3 (paddle resistor ADC)

Address	Outputs
$C058	Clear AN0 (paddle)
$C059	Set AN0 (paddle)
$C05A	Clear AN1 (paddle)
$C05B	Set AN1 (paddle)
$C05C	Clear AN2 (paddle)
$C05D	Set AN2 (paddle)
$C05E	Clear AN3 (paddle)
$C05F	Set AN3 (paddle)

Address	Strobes
$C040	Strobe to paddles
$C070	Trigger the 4 paddle timers

It is worth to remark that in later Apple II models the paddle buttons at $C061 and $C062 were also connected to the "open apple" and "closed apple" keys of the keyboard, and the unused input at $C063 was also connected to the "shift" keys.

1.5 The floppy disk controller

This board is usually found in slot #6, and consequently, its I/O address space is found at $C0E0 - $C0EF. It also includes a 256 byte ROM located at addresses $C600 - $C6FF. This small ROM code only allows the booting from a floppy disk, the rest of the disk operating system is read from the floppy disk to RAM.

The floppy disk controller was included in almost all Apple II computers, and its brilliant design raised Wozniak to the status of genius. I don't want to explain here its internal details, just to comment its main features because this piece of hardware has to be minimally emulated.

The I/O space includes an addressable latch with address line A0 as the written data. Therefore we get 16 read strobes that actually write 8 control bits:

Address	action/selection	address	action/selection
$C0E0	PHS0 off	$C0E1	PHS0 on
$C0E2	PHS1 off	$C0E3	PHS1 on
$C0E4	PHS2 off	$C0E5	PHS2 on
$C0E6	PHS3 off	$C0E7	PHS3 on
$C0E8	Drive off	$C0E9	Drive on
$C0EA	Drive #1	$C0EB	Drive #2
$C0EC	shift (writing)/read	$C0ED	load (write)/ read write protect
$C0EE	Read	$C0EF	Write

PHS0 to PSH3 are the four phases that control the stepper motors of the drives. The proper sequencing and timing of these signals moves the drive head back and forth until it is positioned over the desired disk track. But before this the drive has to be selected (reading $C0EA or $C0EB), and turned on reading $C0E9.

When the head is on the track, the read or write operation is selected by reading $C0EE or $C0EF, and then the data is sent or retrieved through addresses $C0ED and $C0EC. But this isn't a simple task: the data has to be properly encoded and decoded by software, and execution time is also critical. Probably, this is the reason why no Apple II ever used the 6502 interrupts: they would ruin the timing of the floppy drive code.

Yes, this is a poor's man floppy controller, but it was not only a lot cheaper with only 8 simple chips, it also allowed to store more data on disk than other expensive controllers, and also it was quite fast (compare it with the very complex but sluggish Commodore floppy).

Inside the controller lies an state machine built with a 256 byte ROM and 6 D-type flip-flops that controls the shifting of a bidirectional 8-bit shift register. The contents of this register are read when A0 is low (for instance at address $C0EC), so, we must avoid writes to even addresses in the slot I/O range (the R/W signal isn't used at all in the floppy controller card).

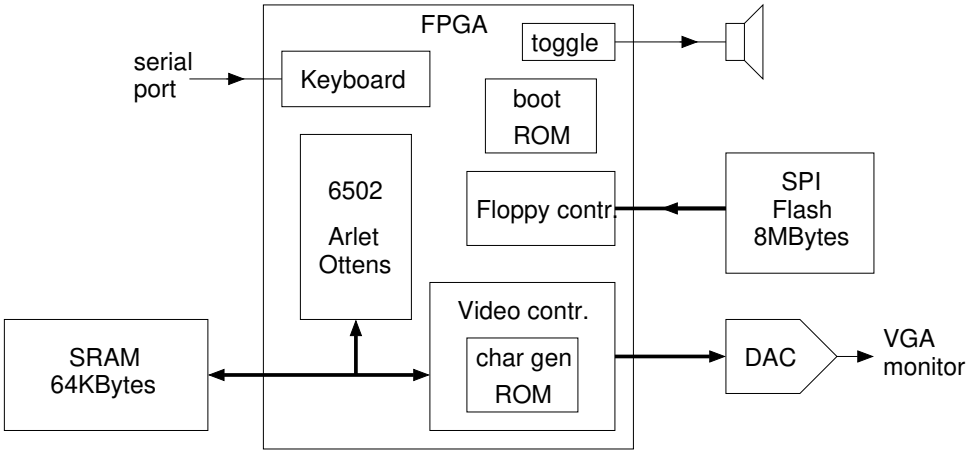
2 Design of the FPGA replica

The Apple II replica is quite different from the original. It targets the SIMRETRO board, that includes an ICE40HX4K FPGA along with an SPI flash memory and 128KB of external SRAM. Of course, the design

includes a 6502 CPU core that executes all the documented NMOS chip opcodes, but many other things differ. It is worth mentioning:

- There is no ROM memory in the replica other than the character generator ROM. A plain 64kB of RAM is used instead. Memory addresses over $C100 are first loaded with the Apple II ROM contents (slot #6 ROM included), and then writes to these locations are inhibited. An small boot ROM (512 bytes) is enabled after reset to allow the loading and then disabled.
- The video controller generates a 12 bit per pixel VGA signal (640x480 VGA mode) instead of an 1 bit per pixel NTSC signal. NTSC color subcarrier modulation can be emulated or not, allowing the user to watch a color screen or a monochrome one (green & black).
- The Lo-res graphics mode is not supported. It would be relatively easy to include but I don't think we are losing too much not having it.
- The keyboard is currently simulated using a serial port. In order to have an autonomous system the keyboard should be the one attached to the PS2 connector, not the serial port. This part is still pending development (the serial port peripheral is a much simpler one because no scancode translation is required at all)
- No cassette, nor paddle controller, are supported.
- The floppy disk is partialy emulated using the SPI flash. It only allows reads and the flash image must contain the encoded image of the data (a sector takes 374 bytes instead of 256). The replica is able to boot from the floppy image.

The basic block diagram of the Apple II replica is presented in the following figure:



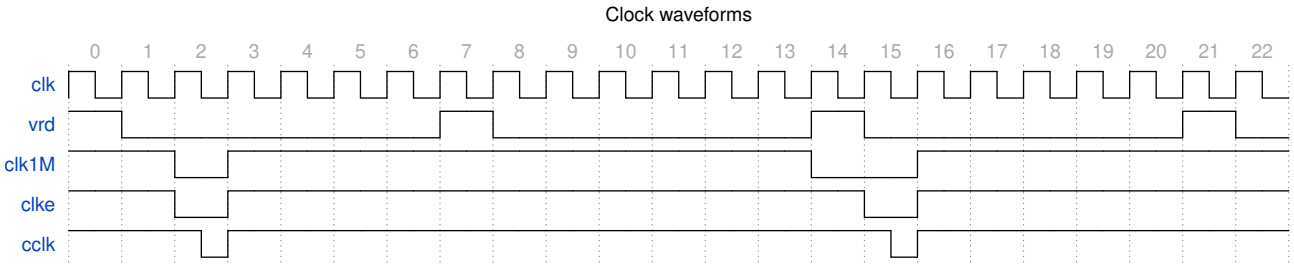
Here the main component is the FPGA that includes many functional blocks of the computer. Outside the FPGA are two memories: An SRAM where the whole address space of the Apple II fits (in fact the external SRAM size is 128 KBytes), and a non-volatile Flash memory that includes the configuration bitstream for the FPGA along with images of the Apple II ROMs and floppy disks.

Inside the FPGA also reside two small ROMs (512 bytes each): one stores the 6502 code that is executed at startup, that basically loads the Apple II ROM images from the SPI flash to the SRAM, and the other is used for character generation in the video controller. These ROMs are in fact synchronous BRAMs with a known initial content and write disabled. Only 2 of the 32 available BRAMs of the FPGA are used.

2.1 Clocks

The main clock of the replica is a 12.5MHz signal obtained by means of a PLL that generates a 25MHz clock that is then divided by 2. A VGA line last $32\mu s$, resulting in 400 horizontal pixels when using a 12.5MHz clock (only 320 pixels visible). During video refresh the video controller has to read a memory byte every 7 cycles, both in text or graphics modes. An output signal, "vrd" gets asserted in the video controller when a memory read is being performed.

The CPU clock runs at 1/12 of the main clock frequency, or 1.042 MHz. This is an 1.8% faster than the real computer, but I don't think this small difference is going to be noticeable. In order to generate the CPU clock, "cclk", a clock enable signal, "clke" (active low) is generated first. The main clock is divided by 12 using a counter that sets "clk1M" low on overflows. "clk1M" is set high the next clock cycle, unless the "vrd" signal is also active, that results in a delayed rising edge. The clock enable signal is the logical OR of "clk1M" and "vrd", and "cclk" is also the logical OR of "clke" and the main clock, "clk" (see next figure).



In this way, the resulting CPU clock is aperiodic, but the core always gets one clock pulse every 12 "clk" cycles. Also notice that "cclk" has no glitches in spite of the combinational logic involved because all signals are derived from "clk", so they are delayed, and "clk" is high during the time they take for settling.

A fast clock can also be generated by simply replacing "clke" with "vrd". In that case the effective CPU clock is 12.5 MHz during retraces or 6/7 (10.7 MHz) of this value during video refresh. The fast clock is used only during replica startup (memory init and loading of the Apple II ROMs via SPI bitbanging).

2.2 Booting

In addition to the recreated Apple II hardware there is also a boot register that a reset pulse sets to all ones. It includes a "boot" bit that enables the reading of the boot ROM instead of the SRAM in the high memory area, so, the 6502 starts executing code from this internal ROM memory. Also, the "boot" bit enables the writing into the whole SRAM and allows to write the boot register. If this bit is written with zero the boot ROM disappears from the memory space and the boot register and high SRAM memory are no longer writable. The boot register can be read and written at address $C020 (same address as cassette output, not recreated), and its bits are:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ro	-	rw	rw	rw	rw	rw	rw
MISO	-	FAST	BOOT	/CS2	/CS1	SCK	MOSI

Along with the BOOT bit we have a FAST bit that selects a fast CPU clock when one, and all the bits needed for a bitbanging SPI interface. /CS1 is the slave select line for the SPI Flash memory and /CS2 the slave select line for an SD card (not yet used).

SCK and MOSI are ORed with the SPI outputs of the floppy controller and they must end the booting procedure in a low state. Similarly, the /CS1 line is ANDed with the equivalent output of the floppy controller. This signal, and also /CS2, must end in a high state.

The code of the boot ROM starts filling the memory with a random pattern (to avoid warm boots in the Apple II), then issues an SPI read command to the external flash, and reads 16128 bytes starting at address

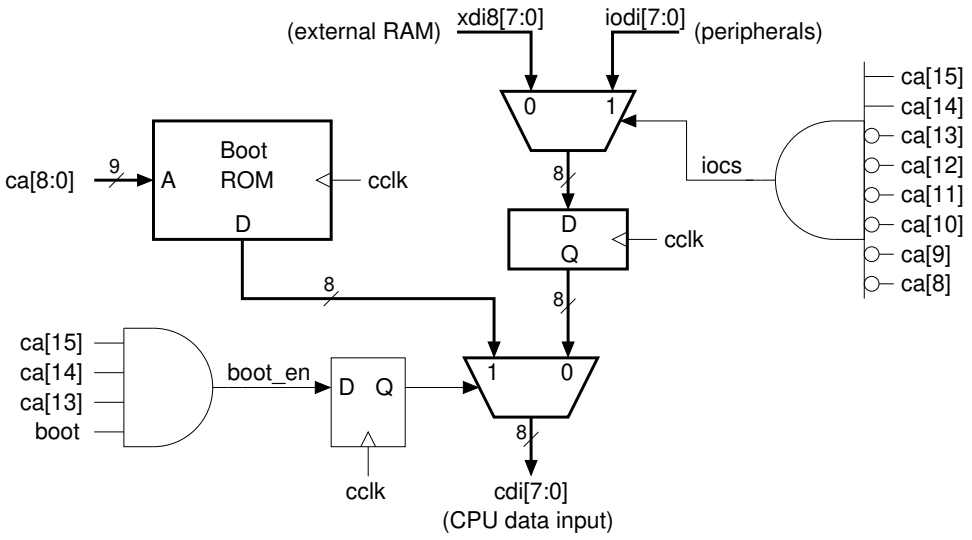
$C100. This copies to the external SRAM the code of all the Apple II ROMs, slots ROMs included. Then it copies an small trampoline code to RAM and jumps to it. This code is:

```
        ; copy trampoline to RAM (to address $0000)
        ldx     #(endtpln-trampoline-1)
cpy1:   lda     trampoline,x
        sta     0,x
        dex
        bpl     cpy1
        jmp     0
        ;Trampoline code (to be copied to RAM)
trampoline:
        lda     #$0C    ; Clear Boot flag (boot ROM no longer mapped)
        sta     $C020
        jmp     ($FFFC) ; jump to Reset vector
endtpln:
```

The trampoline code has to be executed from RAM because it disables the boot ROM. When reaching its last instruction the emulation of the Apple II starts.

The boot code was also capable of loading snapshot files with the contents of the 48KB of RAM made with an emulator. In this case the status of the video mode latches and the CPU registers have to be restored too. Files are loaded using the same serial port of the keyboard interface (reads at $C010 returns the 8-bit serial value in addition to clear the keyboard flag)

2.3 CPU data input and output



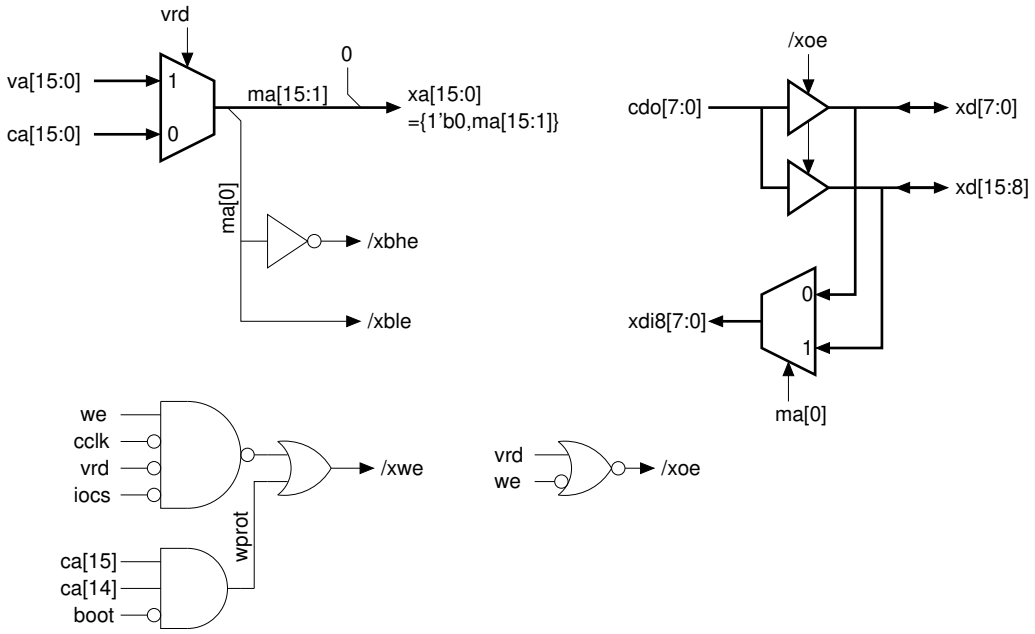
The CPU core I used was developed by Arlet Ottens, and it expects a synchronous memory attached to its input data bus. This means that memory reads are actually two cycles long: in the first cycle the address is presented to the memory and in the second cycle the CPU gets the data. The internal boot ROM does exactly that, but the external SRAM, and also the recreated peripherals, are asynchronous and their data is available at the end of the same cycle the address is presented. This was solved simply by adding a register in series with the data coming from external RAM or peripherals. For the same reason, the multiplexing of the boot ROM data is selected using a delayed boot_enable signal.

The CPU address lines ("ca[15:0]") are used to select the data source. The boot ROM is selected if the address is $E000 or higher, but the ROM has only 512 bytes, not 8 Kbytes, and therefore its contents are

repeated 16 times in the whole boot ROM range (only addresses $FE00 to $FFFF are actually used in the code). Peripherals are selected for any address in the $C000 to $C0FF range. Further address decoding is still required for individual peripherals.

On the other hand, the CPU data output bus is connected directly to the external memory and all the peripherals, and individual write signals are generated depending on the address bus and the write enable "we" signal of the core. If "iocs" is low writes are directed to the external RAM, unless ca[15] and ca[14] are high and the "boot" signal is low. The external RAM write signal is only active when the CPU clock is low. This ensures the address and data buses of the SRAM have valid data before the write pulse.

2.4 External memory interface



The external SRAM has a 16-bit bidirectional data bus. Internally what we need are 8-bit input and output data buses, so, the interface logic of the above figure was designed as an interface. First, the address is selected between that coming from the CPU, "ca[15:0]", or from the video generator, "va[15:0]". Notice that only half of the external memory is actually used, and the address line "xa[15]" is always zero.

The 8-bit output data is presented on the two halves of the external memory data bus during writes, but "/xble" and "/xbhe" are never active simultaneously, so, only one byte gets written. On the other hand, during reads the correct data bus lines have to be routed into the internal 8-bit data bus, "xdi8[7:0]". Tristate outputs are only available on the FPGA pins, and their use is a bit problematic because some vendor-specific cells have to be instantiated.

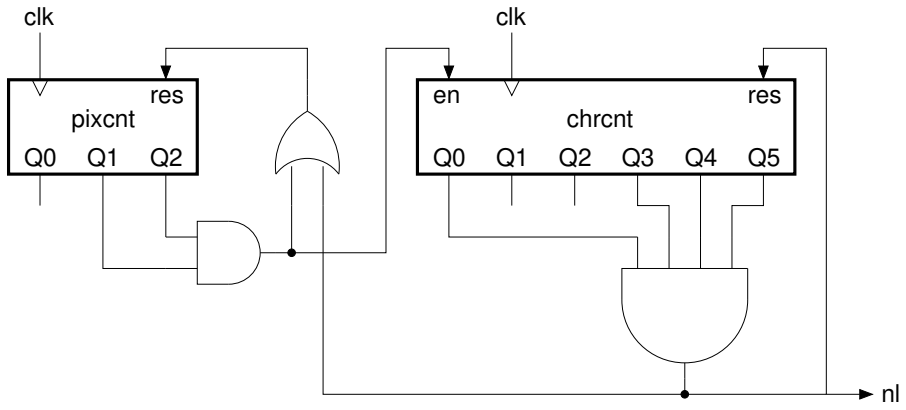
The "wprot" signal inhibits writes to the external RAM if "boot" is deasserted and we are trying to write into a ROM address ($C000 or higher).

2.5 Video generator

This is a quite complex circuit due mainly to the weirdness of the Apple II video. Woz did a great job achieving those capabilities with few chips, but the Apple II video is unlike any other. Lets start presenting the timing.

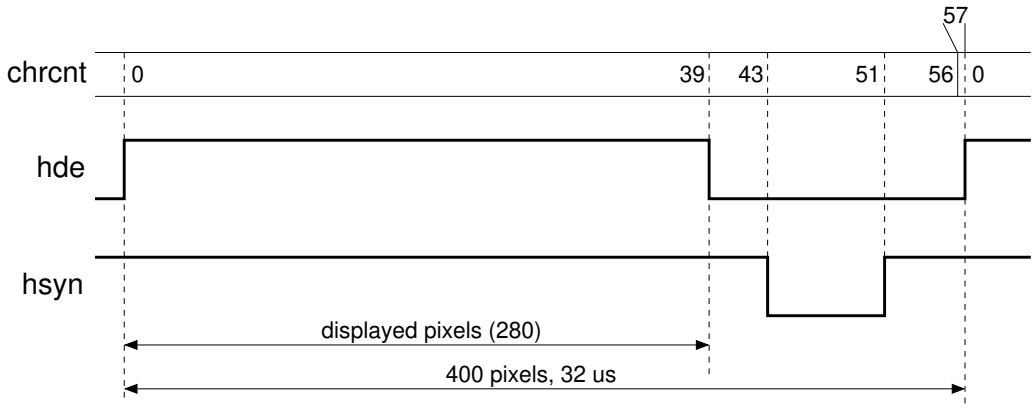
The replica computer is going to use a VGA monitor with separate HSYN and VSYN signals, so, lets begin with the horizontal timing. A video line last 400 pixels and can have up to 320 pixels displayed. But the Apple II reads from memory, either the RAM or the character generator ROM, 7 pixels at a time. Therefore it is very convenient to use two counters: first a modulo 7 pixel counter, and next a "character counter" that gets

incremented when the former overflows. 400 is 57 times 7 plus one, so, both counters get reset one cycle after the char counter reaches the value 57:



The 6-bit value of “chrcnt”, along with “nl”, is used in the generation of the horizontal display enable “hde” and horizontal synchronism pulse, “hsyn”. Both signals are registered:

	hde	hsyn
Set	nl	chrcnt==43
Clear	chrcnt==39	chrcnt==51



The vertical counter, “vc” is a simple 10-bit binary counter clocked by the rising edges of “hde” instead of “clk”, so, it counts lines. This counter gets reset when it reaches the value 524 (vertical total: 525 lines). Three signals are derived from its count: vertical display enable, “vde”, vertical synchronism pulse, “vsyn”, and line over 160, “l160”. This last signal is used in the mixing of graphics and text modes.

	vde	vsyn	l160
Set	vc==524	vc==441	vc==319
Clear	vc==383	vc==443	vc==524

The display has 384 visible lines but each line is displayed twice. This gives 192 display lines for the replica video controller. When both “hde” and “vde” are active the controller reads a byte from memory each 7 cycles. More precisely: when the contents of “pixcnt” are 0:

```
wire de=hde&vde;
assign rd=de&(pixcnt==0);
```

The memory address for the video read is different for graphics and text modes. In text mode the address offset is:

$$txtva = (vc/128)*40 + [(vc/16)\&7]*128 + chrcnt$$

where “vc” is the actual vertical counter for the VGA (384 visible lines, not 192). And taking into account that 40=(32+8) we have to add:

	(vc/128)*32				vc8	vc7	0	0	0	0	0
$\sum$	(vc/128)*8						vc8	vc7	0	0	0
	[(vc/16)&7]*128	vc6	vc5	vc4	0	0	0	0	0	0	0
	chrcnt					ch5	ch4	ch3	ch2	ch1	ch0
=	txtva	ta9	ta8	ta7	ta6	ta5	ta4	ta3	ta2	ta1	ta0

And taking into account that there is no carry after the bit “txtva[6]” (2*40+39 = 119 < 128) we can simplify the logic down to a single 4-bit adder:

```
////////////// video address //////////////
wire [3:0]sum = {vc[8:7],vc[8:7]}+{1'b0,chrcnt[5:3]};
wire [9:0]txtva={vc[6:4],sum,chrcnt[2:0]};
```

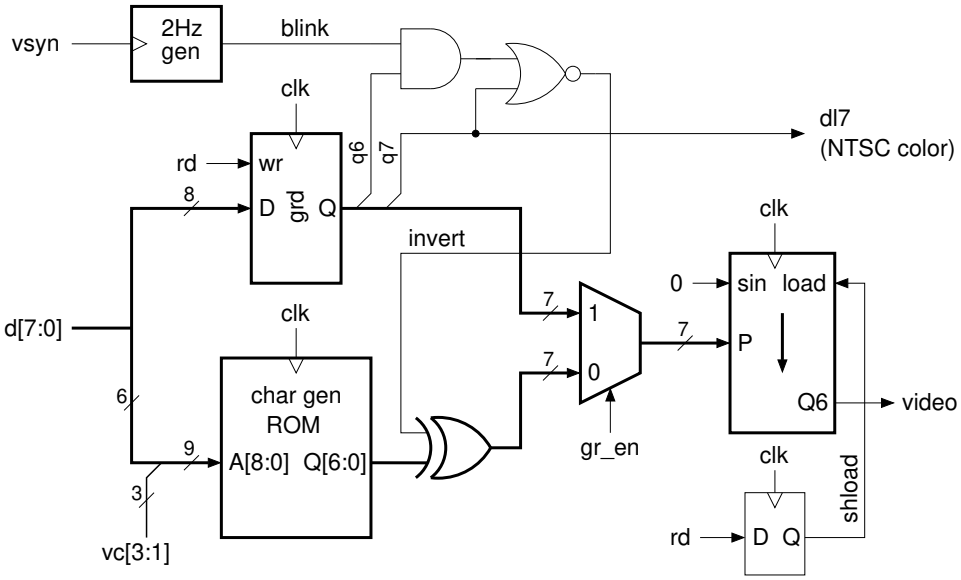
That was for text mode, but for hi-res graphics the ten lower bits of the address are the same, we just has to add three bits of the vertical counter to the MSBs:

```
// address generation
assign a={gr_en?{1'b0,page,~page,vc[3:1]}: // graphics mode MSBs
                {4'b0000,page,~page},      // text mode MSBs
                 txtva};                   // LSBs
```

Here the control bit "page" selects between the two memory areas for the displayed image. Also, the wire "gr_en" can change during the video frame if text and graphics are mixed. In that case text addresses are selected if "l160" is active:

```
// graphics/text switch
wire gr_en = (~text) & (~(mix & l160));
```

After video address generation comes the processing of the retrieved data. The diagram of the video signal logic is presented next:



In text mode the 6 lower bits of the data, plus the three lower bits of the vertical counter, are used as the input address for the character generator ROM. The bit "vc[0]" is not used and, consequently, characters are 16 lines high, but with identical even and odd lines. The ROM is actually a BRAM block with synchronous read, so, its output is delayed until next clock cycle.

The data coming from memory is also stored in a register, "grd", where the output bits "q6" and "q7" hold the character attributes (invert and blink). The lower 7 bits of this register can also be routed to the pixel shifter in graphics mode, thus providing the same delay as the character generator ROM. An square wave about 2Hz is also generated using a 5-bit counter clocked by the vertical synchronism signal, and along the attribute bits, "q6" and "q7", results in an "invert" signal that complements the character generator ROM output when active.

The pixel shift register has 7 bits, shifts out first the lower bit of its contents, and load its most significant bit with zero during shifts, thus avoiding any further blanking logic: If the shift register is not loaded its output ends at zero. Its synchronous load signal is activated one cycle after data is read from memory to account for the delay of the character generator ROM or the "grd" register.

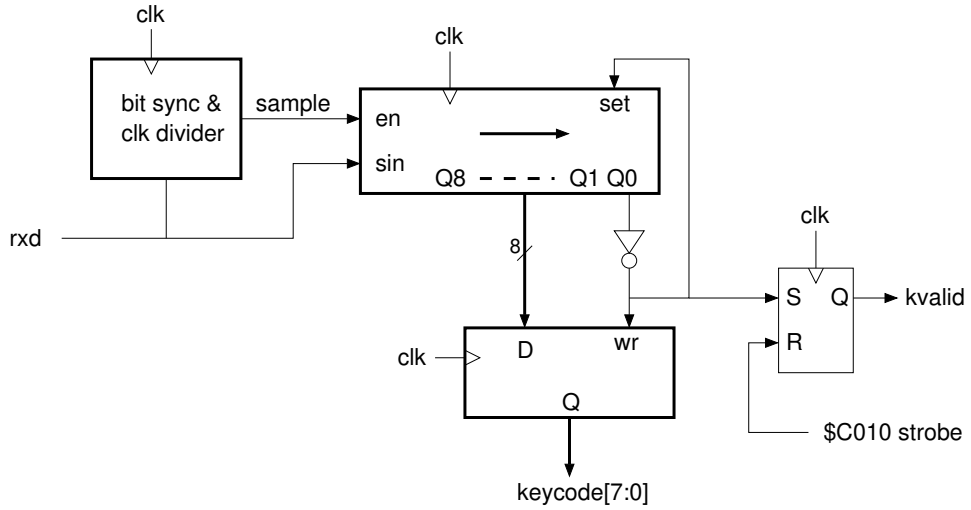
With this last block the monochrome video generator is complete. We just have to route the video output to all the bits of the VGA DACs and we get a black and white picture in the monitor. Or you can just route the video output to the green component if you want to emulate the classic green phosphor Apple monitor.

But, if you want to emulate a color NTSC monitor there are still complications ahead. The main idea is to have six possible colors to route to the VGA DACs, and to chose the proper one depending on the 8 possible combinations of the current pixel pair plus the "dl7" bit. In order to achieve this the video output is delayed two pixels, and the last values stored in a 2-bit register every two clock cycles (notice that horizontal counters are useless because "pixcnt" is modulus 7). Then comes the color multiplexing that includes also a "color" input to select between monochrome or color monitor emulation. We still lack the artifacts of the half-delayed pixels, but I don't think Woz would have objections about perfection.

At the end the video pipeline has a 4 pixel delay. This is less than one character and no adjustment is required in the "hsync" position (modern monitors also have auto adjust capabilities for image centering, so we don't have much to worry).

2.6 Keyboard interface

The keyboard interface is actually a 115200 baud serial port receiver. Its block diagram is:



Apart from the bit synchronizer, the circuit is simply a 9-bit shift register that samples the serial data at the middle of the bit time. When the start bit arrives at Q0 the rest of the received bits are stored in an 8-bit output register, the "kvalid" signal is set, and the shift register is preloaded with all its bits as ones. The "kvalid" signal is cleared by a read to address $C010. The received keycode can be read at two different addresses:

address	D7	D6	D5	D4	D3	D2	D1	D0	effect
$C000	kvalid	keycode[6:0]							-
$C010	keycode[7:0]								clears kvalid

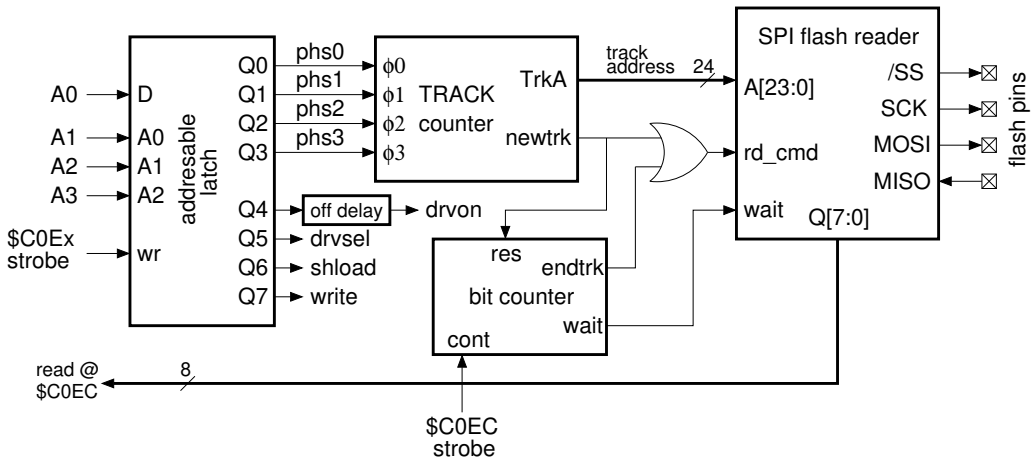
The $C010 address is used in the boot ROM for reading snapshot files (8-bit data)

The "bit sync & clk divider" block is a 7-bit counter that gets reset when it reaches the value 107 (12.5MHz/108 = 115Kbaud), or when the input signal, "rxd", changes. The "sample" signal is asserted when the counter value is 54 (at the middle of the bit time) and last a single clock cycle.

2.7 Floppy disk

The speaker interface is a trivial one, just a toggling flip-flop, so, lets move into the final peripheral of the Apple II recreation: the floppy controller. That card very often is installed on slot #6, and only that possibility was considered here. The media for the storage of the floppy disk data is the same SPI flash that was previously used to load the ROM contents into the external RAM. Only reading is currently supported.

A simplified diagram of the recreated floppy controller is shown next:



This circuit recreates the addressable latch of the original card, were 8 bits can be set or reset via read instructions. Among these bits are the four phases that controls the stepper motor of the drive head. These signals are processed in the track counter, where a flash address for the current track data is incremented or decremented depending on the phase activation sequence. Tracks are spaced two steps apart, with the motor resting on "phs0" or "phs2". Also, a new track signal is generated when the current track changes.

Each track holds 16 sectors, each with 256 bytes of useful data, but along that data we get preambles, ID fields, cheksums, and gaps. The data also has to be properly encoded, resulting in 374 total bytes read at address $C0EC for each sector, or 5984 bytes per track. The address presented to the flash reader block is therefore incremented or decremented in 5984 byte steps when the current track changes.

The SPI flash reader is a simple state machine that controls a 32-bit shift register. When "rd_cmd" is asserted "/SS" goes high for a single clock cycle, and the shift register is loaded with the command byte $03 (flash read) along with the 24-bit address. Then the register contents are shifted to the left on every clock cycle. Its most significant bit is routed to "MOSI", and an "SCK" pulse is generated. The data coming through "MISO" is also shifted into the least significant bit of the register. The 8 LSBs of the shift register are the reader output "Q[7:0]".

A bit counter is also present. This counter stops the reader every 8 cycles after the initial 32-bit command sequence, thus forcing it to wait until the 6502 reads the emulated disk byte at address $C0EC. Also, when the number of read bytes reaches 5984 the flash reader, and the counter itself, get reset and the track data starts to get read from the beginning again.

With this crappy emulation the Apple II replica was able to boot from a floppy disk image previously encoded with an application derived from an old Apple II emulator and written into the SPI flash of the SIM-RETRO board. A 140KB disk image requires 205KB of flash space after the encoding.

I must confess I still want to recreate the real Wozniak Machine. That won't affect the image storage too much: Only the preamble bytes $FF have to be changed to 10-bit words, $3FC. But on the other hand the timing is going to be more critical than now. A bit stream would be retrieved directly from the MISO signal, one bit every 4 CPU cycles (250Kbit/s), and ones would result in 1 cycle high followed by 3 cycles low, while zeros are always low.

2.8 Sound emulation

Something I immediately missed after getting the floppy emulation running was the noise made by the real drive. With these drives you can clearly ear the disk spinning and the head changing track, not to mention the loud bangs when the head collides with the track #0 stop during reboots, and these noises provide a nice feedback to the Apple II user (You can predict an incoming read error if the head moves more than usual, for instance). In the FPGA recreation there are no moving parts and the complete silence was... disturbing, so, these noises had to be emulated through the speaker.

This presents two challenges: the generation of audio signals that resemble the real ones, and the mixing of all the signals, Apple II speaker included, into a single digital output.

The mixing was the easy part: we just have to resort to time-domain multiplexing: A 5-bit binary counter runs from the CPU clock and overflows every 32 cycles (about 32 μs, or with a frequency of 32.5 kHz). Then, depending on the counter value a different signal is routed to the speaker:

Count	Sound source
0 and 1	Track noise
2	Spinning noise
3 to 31	Apple II speaker

A short time for a given signal will result in a lower volume. With this configuration the relative volumes are: 1/32 for the spinning noise, 1/16 for the track noise, and 29/32 for the computer speaker.

The head positioning mechanism basically sounds a "click" for every step its motor is moving. In the emulation these "clicks" are 2 ms pulses triggered by the rising edges of any of the four stepper motor phases. This approach generates a quite convincing sound.

The spinning noise required more elaboration. The disk spins at 300 rpm, or 5 Hz. The sound is more or less white noise, but with a clear 5 Hz component. My solution here was to generate a pseudo-random binary signal (PRBS) using a 12-bit linear-feedback shift register (LFSR) clocked by a 32.5kHz signal obtained from the counter of the mixer. But, with this approach the disk was spinning too fast: 32552 Hz /4095 = 7.95 Hz, or 477 rpm. In order to lower the frequency to near 5 Hz the LFSR only changes state 5 out of 8 clock cycles. And, of course, the LFSR output is ANDed with the "drive on" signal of the floppy disk controller before reaching the mixer. The resulting sound was also close to what was heard 40 years ago.

So, now we can type "CATALOG" and listen to the angelical music of a virtual Shugart SA-390 drive in action. Only the alarming noise made when the head bangs against track #0 is missing, and I don't think it is going to be recreated, mainly because that sound seems to depend a lot on the acoustic resonances of the metallic floppy case, and without resorting to actual samples it would be very difficult to reproduce accurately.

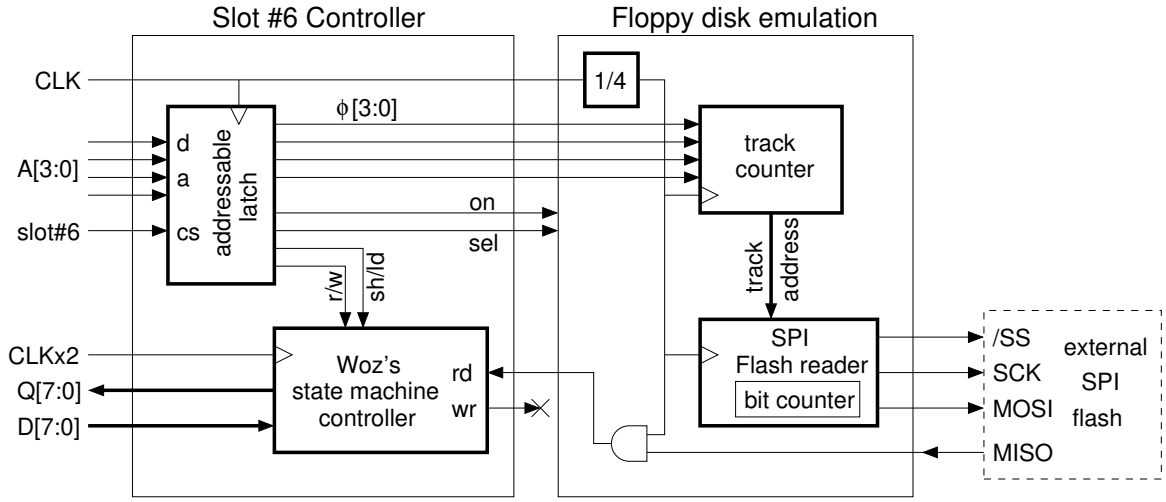
3 Improvements

3.1 Floppy disk revisited

I wasn't very satisfied about the floppy disk emulation, mainly because the chosen solution bypasses the Wozniak state machine completely, and I wanted to have that hardware part working. So, I did some math about floppy disk images: The spinning speed of the disk is 5 revolutions per second (300rpm), and its data rate is 250Kbit/s. This means that a track has a raw data capacity of 50000 bits (or 6250 bytes). Currently, a track takes 47872 bits (374 bytes per sector, including preambles, address fields, data marks, checksums, and the overhead of the 4-to-8 or 6-to-8 encodings). So, the storage of the track as a raw bitstream will only requires a little 4.4% more space in the disk image. In fact, the current disk image only lacks the gaps between sectors and between address and data fields, and the two extra bits of the syncing bytes of the preambles. Thus, I decided to do a new design of the floppy interface with these features:

- Raw bitstreams for each track stored in a SPI flash. Each track will use around 6250 bytes.
- An SPI clock about 250KHz. This is 1/4 of the CPU frequency and will result in a bit rate very close to the actual floppy disk rate. This clock will have a 25% duty cycle (1 μs high, 3 μs low).
- The data coming through MISO is not stored in any word. It is simply ANDed with the 250kHz clock resulting in 1 μs pulses for ones and 4 μs low for zeroes, like for a real MC3470 head amplifier, and presented to the Wozniak's state machine.

- The SPI clock isn't stopped when the controller data isn't read. This differs from the actual emulation and resembles more a real disk. When the bit count reach its maximum, or when the track position changes, a new read command is issued to the SPI flash.



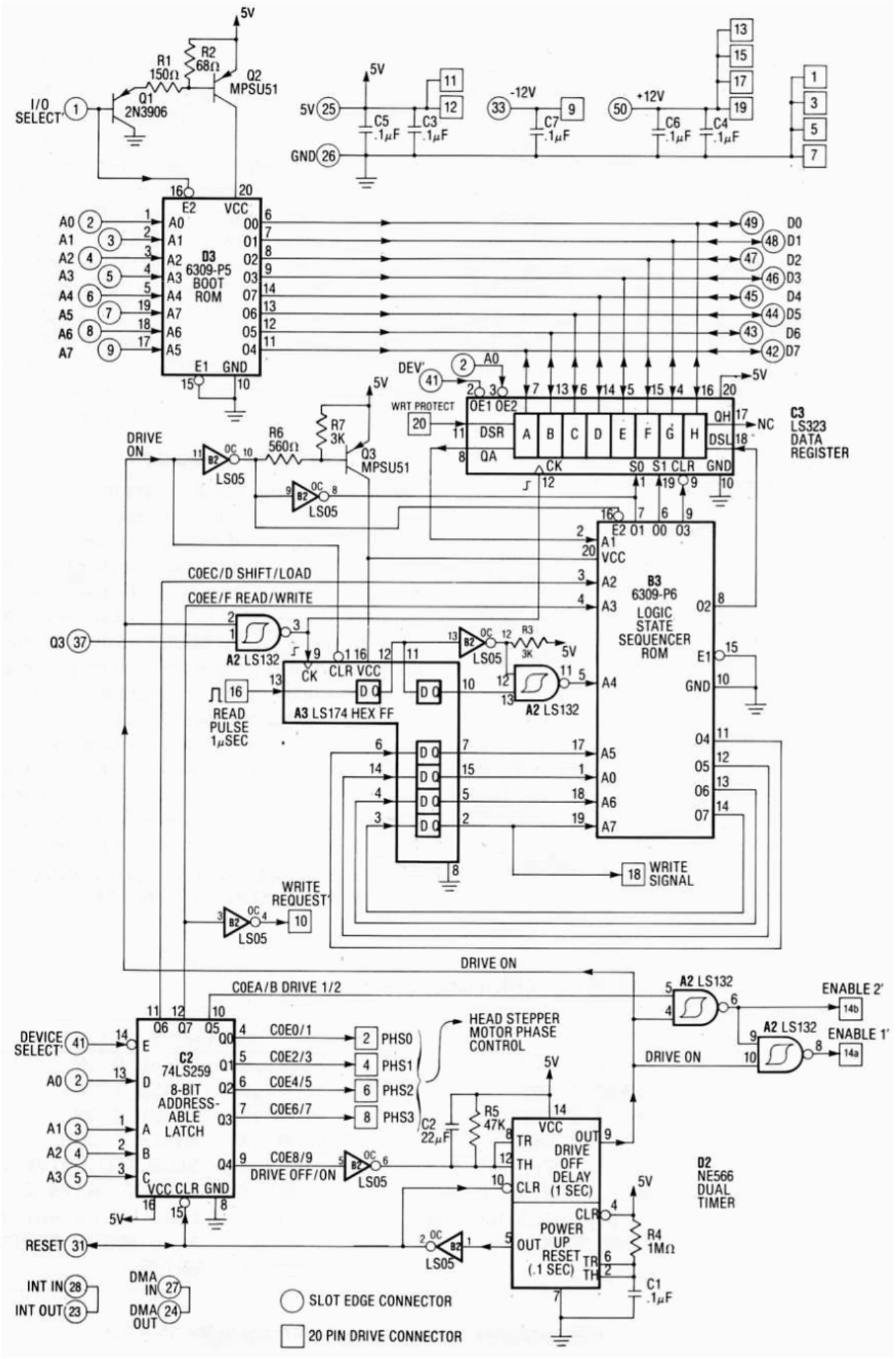
The floppy emulation logic is divided into two main parts: First, the recreation of the controller card, that includes an addressable latch with 8 output bits, and the PROM-based state machine that manages read and write operations. Next , the floppy drive is emulated (only reads) using an external SPI Flash as the real storage for the data, along with its associated logic. This includes a track counter controlled by the 4 stepper motor phases which present the flash address of the current track to the last block: the SPI flash reader, mainly just a shift register for the generation of the MOSI data and a bit counter that issues a Flash read command every 53248 bits.

The controller card schematic includes the famous Wozniak's state machine. It is shown in the next figure (taken from "Undesrtanding the Apple II" by Jim Sather), and includes:

- A 74LS323. An 8-bit bidirectional shift register.
- A 74LS174. A chip containing 6 D-type flip-flops with a common clock.
- A 6309 (-P6). This is a 256 byte PROM that contains the state machine microcode.
- Simple NAND gates (74LS132, a Schmitt-trigger version) and inverters (74LS05: open collector inverters).

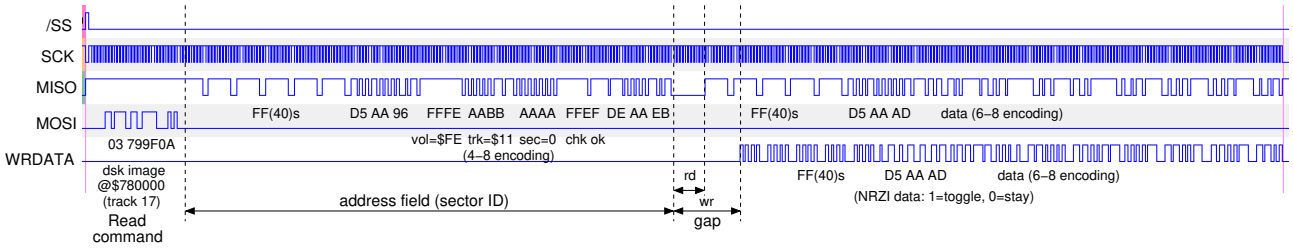
Notice that both the LS174 and the LS323 are running with a double rate clock (Q3: 2 MHz) instead of the main CPU clock. Therefore, a bit time (4 μs) will last 8 clock cycles. This clock signal is synchronous with the CPU clock.

Two flip-flops and two gates are used to detect the falling egdes of the data read signal. A falling edge is presented as a low pulse of just one cycle duration at the PROM's address line A4. The remaing 4 flip-flops hold the current state of the controller. The PROM outputs O[3:0] are used to control the shift register through its serial input, DSL, clear (/CLR, synchronous) , and function-select signals (S0 and S1). The shift register output, QA, is also routed to the Address line A1 of the PROM (This signal is used during disk write operations). The two remaing address lines of the PROM come from the bits 6 and 7 of the addressable latch (bit 6: shift/load, bit 7: read/write).


5V
R1 150Ω
R2 68Ω
Q1 2N3906
Q2 MPSU51
I/O SELECT
-12V
+12V
C5 .1μF
C3 .1μF
C7 .1μF
C6 .1μF
C4 .1μF
GND
D3 6309-P5 BOOT ROM
A0
A1
A2
A3
A4
A5
A6
A7
D0
D1
D2
D3
D4
D5
D6
D7
DEV
WRT PROTECT
C3 LS323 DATA REGISTER
DRIVE ON
R6 560Ω
R7 3K
Q3 MPSU51
LS05
B3 6309-P6 LOGIC STATE SEQUENCER ROM
C0EC/D SHIFT/LOAD
C0EE/F READ/WRITE
A2 LS132
A3 LS174 HEX FF
READ PULSE 1μSEC
WRITE SIGNAL
WRITE REQUEST
DRIVE ON
C0EA/B DRIVE 1/2
ENABLE 2
ENABLE 1
DEVICE SELECT
C2 74LS259 8-BIT ADDRESS-ABLE LATCH
C0E0/1
C0E2/3
C0E4/5
C0E6/7
C0E8/9
PHS0
PHS1
PHS2
PHS3
HEAD STEPPER MOTOR PHASE CONTROL
DRIVE OFF/ON
R5 47K
C2 22μF
D2 NE566 DUAL TIMER
DRIVE OFF DELAY (1 SEC)
POWER UP RESET (.1 SEC)
R4 1MΩ
C1 .1μF
RESET
INT IN
INT OUT
DMA IN
DMA OUT
SLOT EDGE CONNECTOR
20 PIN DRIVE CONNECTOR

The FPGA recreation of the floppy disk controller is now an almost verbatim copy of the former schematic. The PROM is included in a BRAM block with an inverted clock signal for the simulation of an asynchronous memory with half a cycle delay. Alternatively, the PROM can be synthesized as a simple combinational circuit, and in this case its small size, and the regularity of its contents, makes this memory a good candidate for such synthesis, resulting in only 68 logic cells more than in a BRAM implementation (notice the ROM has 2048 bits). The designed substitute for the LS323 only differs from the real one for having separate input and output buses instead of a single bidirectional one.

The recreated controller now generates a write signal with the waveforms to present at the disk head during write operations, but, this signal isn't connected to anything, so no writes are possible. The "wrt protect" signal should be set always high to signal the read-only status of the emulated disks. But, it writes are enabled and the "wrdata" signal routed to a FPGA pin, the following waveforms are observed when executing a SAVE command:

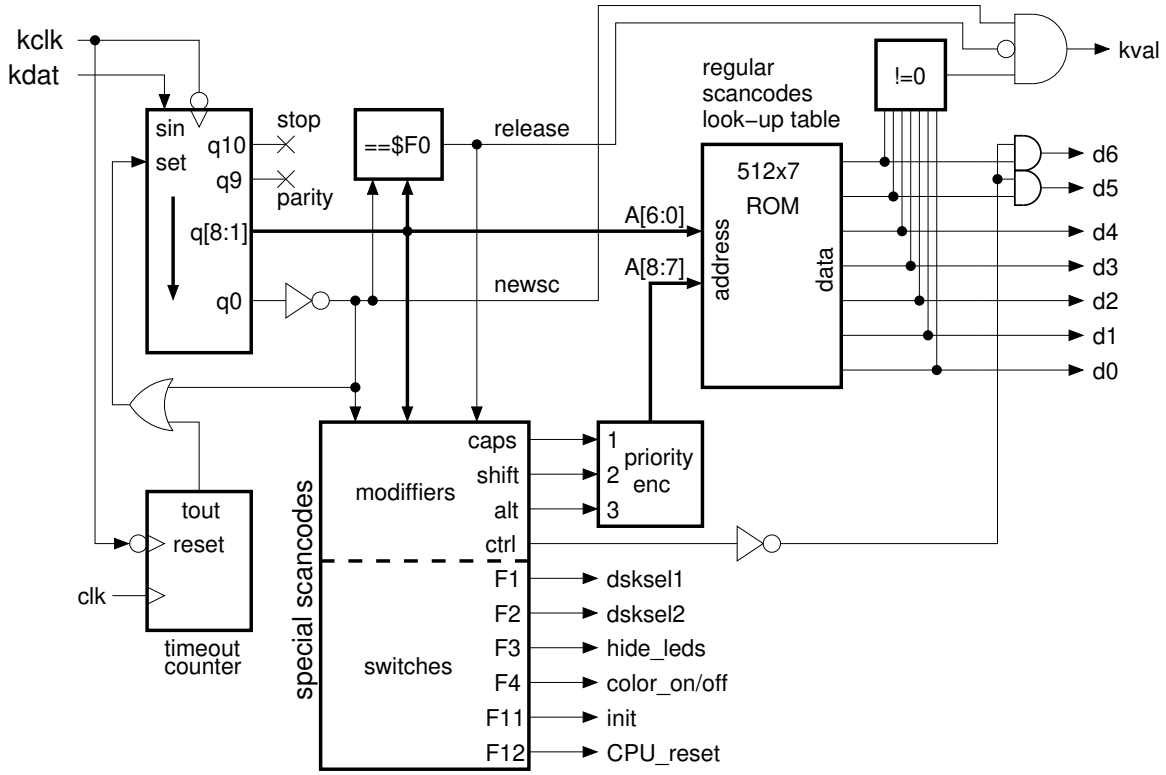


Here the recreated Apple II first waits until it reads the address field of sector #0 at track #17. Then, it switches to write and sends the data field of the sector. Of course, nothing gets written and the old contents of the sector are also simultaneously retrieved through the MISO line of the SPI flash. Notice that the write bitstream is further encoded as an NRZI signal where ones results in a level toggle and zeroes in the same level than the previous bit. This is the kind of signal we expect to have at the disk head during writes. Then, on reads, any polarity change in the magnetic field of the media will generate a read pulse, about 1 μs long, thus signaling a one.

Now the recreated controller is almost the same as in the original Apple II, and it would be quite easy to route all the needed signals out of the FPGA into real floppy drives (if some of these were around, not the case).

As a last addition I included support for two drives. This only required another track counter and a different flash address for the second drive (now "drvsel" is a bit of the track address).

3.2 PS2 keyboard



The PS2 interface was finally designed according with the above diagram, where, as we can see, most of the complexity is related with the scancode to ASCII conversion. The PS2 receiver itself is just an 11-bit shift register where the input bits are shifted on the falling edges of the clock (the register is actually clocked with the CPU clock, but only on the falling edges of "kclk"). When the start bit arrives at Q0 the "newsc" signal gets asserted and the whole register is loaded with ones on the next clock cycle. This circuit is much like the serial receiver, the main difference is the presence of a timeout counter that periodically loads the shift register with ones if no clock pulses are present, so, the shift register is always empty when an scancode starts to being received.

After this simple receiver comes the scancode to ASCII translation. These keyboards sends an scancode when keys are pressed, but also when they are released. In this last case the scancode is preceded by an $F0 code that marks the release of the key. A release signal is asserted if the code $F0 is received, and it holds its state until a new scancode is received. Released keys are basically ignored, and therefore this signal must be zero to enable the rest of the decoding.

There are also some special scancodes that have to be detected. Some of these are related to key modifiers: "caps", "shift", "alt", and "ctrl". These last three scancodes asserts their corresponding signal until the modifier key is released, but "caps" is different: Its modifier signal just toggles on key presses.

"caps","shift", and "alt" are passed through a 2-bit priority encoder before being presented as the two most significant bits of the address of a translation ROM. The lower 7 bits of the address come from the scancode (bit 7 is unconnected as only the key F7 have an scancode higher than 127 and this key isn't used at all). The contents of the ROM are:

modifier key	a[8:7]	address range
—	00	$000 - $07F
caps	01	$080 - $0FF
shift	10	$100 - $17F
alt	11	$180-$ 1FF

Notice that “caps” and “shift” have the same effects on letters but “caps” has no effect on number keys. Also, The “ctrl” modifier isn’t included in the ROM tables because its effect is simply to force the 2 MSBs of the ASCII code low when set. The ROM content was obtained using an ad-hoc application that targeted an Spanish keyboard layout, and was placed into a BRAM block (it was a little too big for a simple combinational synthesis). Unused or nonexistent scancodes will have a zero output, and this condition is checked in the generation of the key valid output, “kval”.

In addition to the modifier keys there are also some convenience keys used as system switches (described later) that are handled in the same way:

key	type	switch
F1	pulse	Increment disk selection for drive 1
F2	pulse	Increment disk selection for drive 2
F3	toggle	Hide / Show virtual LEDs
F4	toggle	Color NTSC / Monochrome monitor
F11	pulse	Init system (reset all)
F12	pulse	Reset CPU (Apple II reset)

The output of the PS2 keyboard block is merged with that of the 115200 baud receiver, so, we can type into the Apple II replica from both interfaces.

3.3 Emulation interface

The last revisions of the design also include some convenience features, like virtual LEDs, that shows the status of some digital signals, and a disc changer to select the actual floppy images used for drives #1 and #2.

The virtual LEDs started as a debugging tool because we have no LEDs in the SIMRETRO board (newer revision boards will have LEDs), but now the virtual LEDs are used as indicators. There are 8 virtual LEDs displayed as rectangles on the left border of the screen. Their are used to show the status of the following signals:

LEDs	Color	Position	Use
0	red	top	Uppercase keys
1	orange		Floppy drive motor
2-4	red		Disk selection for drive 1
5-7	orange	bottom	Disk selection for drive 2

The virtual LEDs can be hidden by pressing F3 on the PS2 keyboard.

The disk changer include two 3-bit counters incremented by keys F1 and F2, that select one of 7 possible disk images previously programmed into the SPI flash (at offset 6MB, last 128Kb are reserved for ROM images).

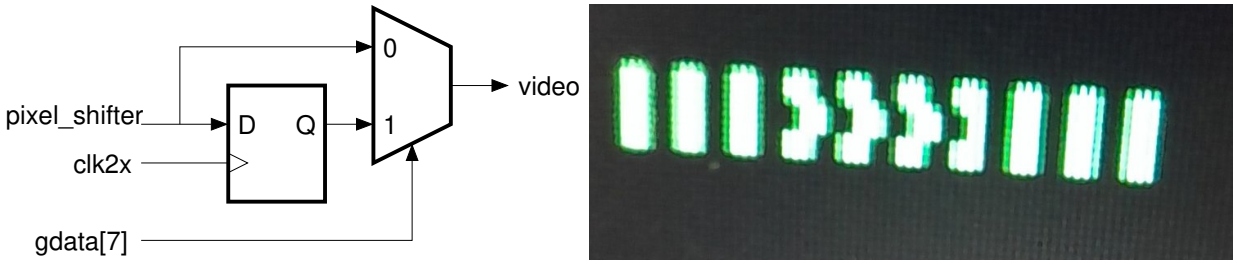
After the design of the Softcard and Floppyton computer the virtual LEDs changed their meaning:

LEDs	Color	Position	Use
0	orange	top	Uppercase keys
1	red		Floppy drive 1 ON
2	red		Floppy drive 2 ON
3	red		Floppy disk Write
4	orange	bottom	Z80 CPU selected (only shown if Softcard included)

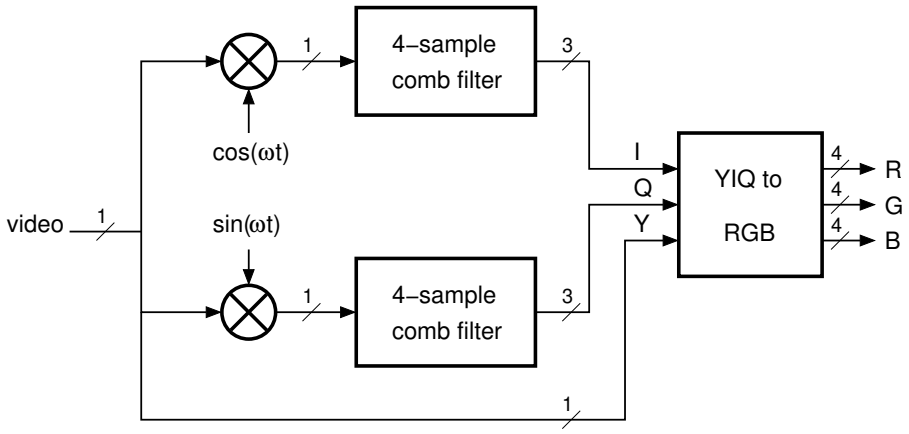
3.4 Video color redesign

As it happened with the floppy disk emulation, I also was a little unsatisfied with the way the color emulation was implemented in the early versions of the machine, so, in the last revision I tried to recreate the actual color processing of a NTSC monitor and the video artifacts of the Apple II computer. This required the use of a clock signal with double the frequency of the pixel shifter, clk2x, at 25MHz. This signal was already being generated by a PLL, so it came at no additional cost.

The first piece of hardware included was the half-pixel delay for the graphics bytes with bit #7 set. It required just a flip-flop clocked by the fast 25MHz clock and a multiplexer for the selection of the normal or delayed video output (D or Q signals of the flip-flop). This hardware is basically the same Woz included in the video controller of the Apple II, and its effects on the video signal, even in monochrome mode, are shown in the following snapshot where the middle byte has its bit #7 set and the half-pixel delay is clearly visible. (Graphic data: lines #0 and #2: $2A, $55, $2A. Line #1: $2A, $D5, $2A)

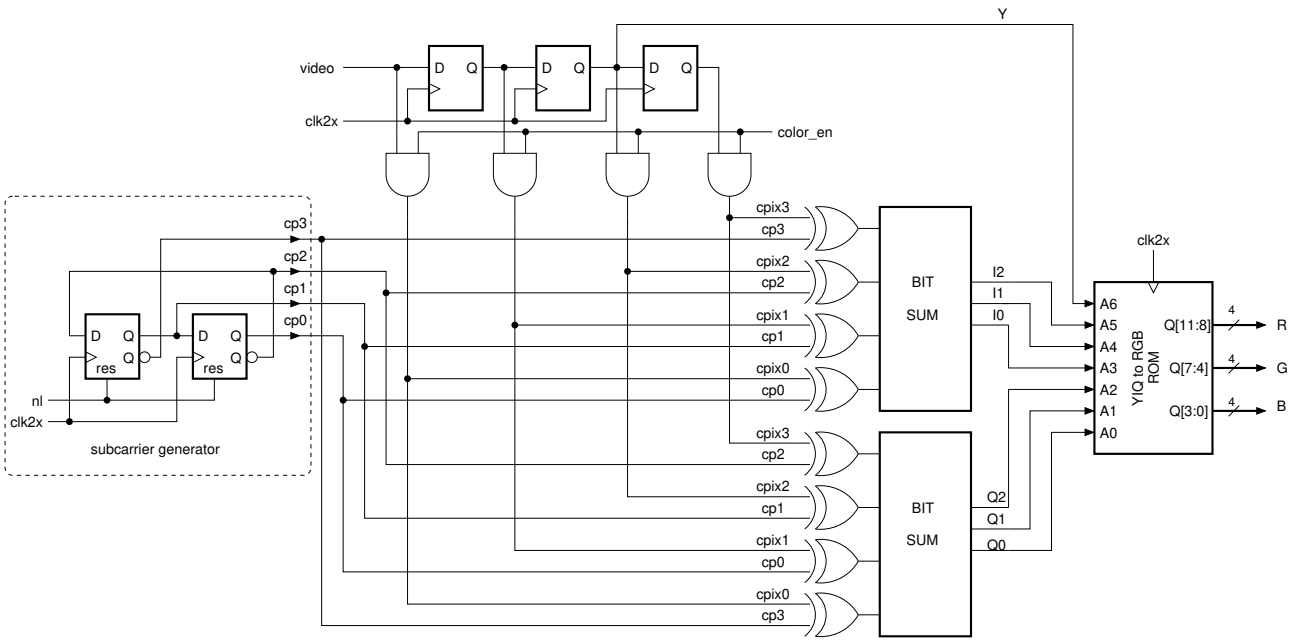


Then the chroma demodulator of an NTSC monitor had to be implemented in the digital domain. The block diagram of such demodulator is:



Where $cos(\omega t)$ and $sin(\omega t)$ are the in-phase and in-quadrature components of the color subcarrier, that are the sequences “1,1,-1,-1...” and “-1,1,1,-1” or in other words square wave signals with a four cycle period. These two signals are obtained from a 2-bit Johnson counter that is reset at the beginning of video lines.

The two mixers are just XOR gates and the comb filters are simply the dynamic addition of the last four input values. All this logic is detailed in the following diagram:



Apart from the generation of the color subcarrier and its four phases, the last three video samples are stored in a shift register and XORed with the delayed subcarrier samples along with the current video data. The delayed subcarrier samples are in fact the same signals as the subcarrier phases. The four 1-bit values of the mixing are added together, resulting in a value range from 0 to 4, that represent 5 different levels for I and Q: [-2, -1, 0, 1, 2]. These bit adders can be built using 6 half-adders, or just three four input, one output, ROMs (logic cells, in fact).

Notice that a color enable signal will force the I and Q levels as zero (binary 010) if not asserted. This signal is disabled in text mode or when a monochrome monitor is selected. The luminance signal, Y, is obtained from the delayed video in order to account for the average delay of the comb filters (two cycles)

The last block of the color demodulator is a translation table from the YIQ color space to the RGB space. This was implemented as a synchronous ROM memory that required an additional BRAM block. Its contents were calculated in the following way:

- First, the Y, I, and Q, components were scaled to the following ranges (https://en.wikipedia.org/wiki/YIQ):

Component	Input range	Output range
Y	[0,1]	[0,1]
I	[0,4]	[-0.5957,+0.5957]
Q	[0,4]	[-0.5226,+0.5226]

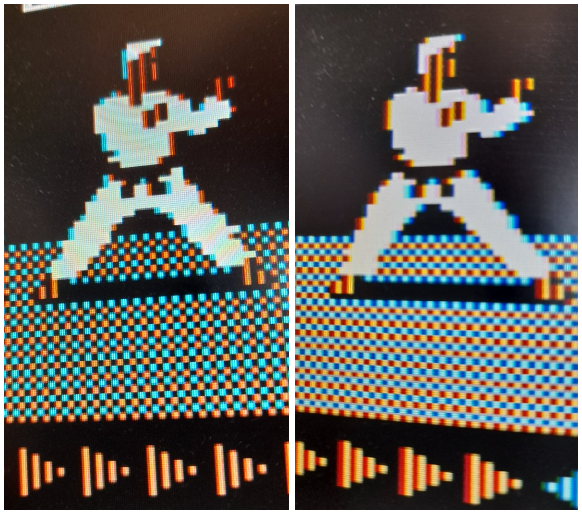
- Then, the normalized R, G, and B values were obtained (https://en.wikipedia.org/wiki/YIQ):

$$R = Y + 0.956I + 0.619Q$$
$$G = Y - 0.272I - 0.647Q$$
$$B = Y - 1.106I + 1.703Q$$

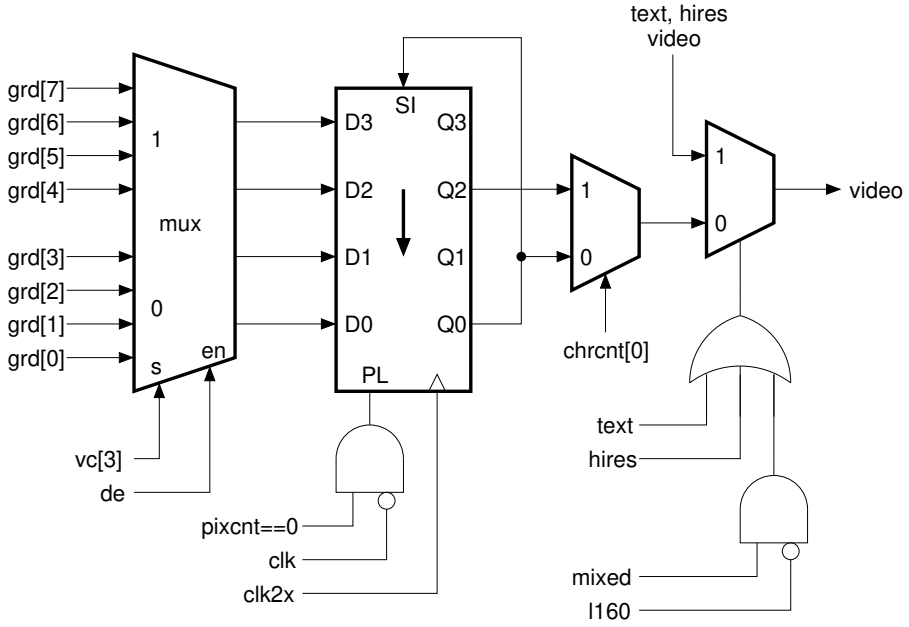
- The R, G, and B values were multiplied by 16, rounded to the nearest integer value, saturated to the [0,15] range, and stored as the initial values of a memory block.

The RBG output of the color translation ROM isn't yet the final video data. These bits can be ANDed with a color mask if a monochrome green or amber monitor is emulated instead of a black and white one, and ORed with the video output of the virtual LED generator.

The resulting design now shows the typical colored blurring of HIRES graphics. The next two snapshots shows an example of the images generated using the old color emulation (left) and the new one (right).

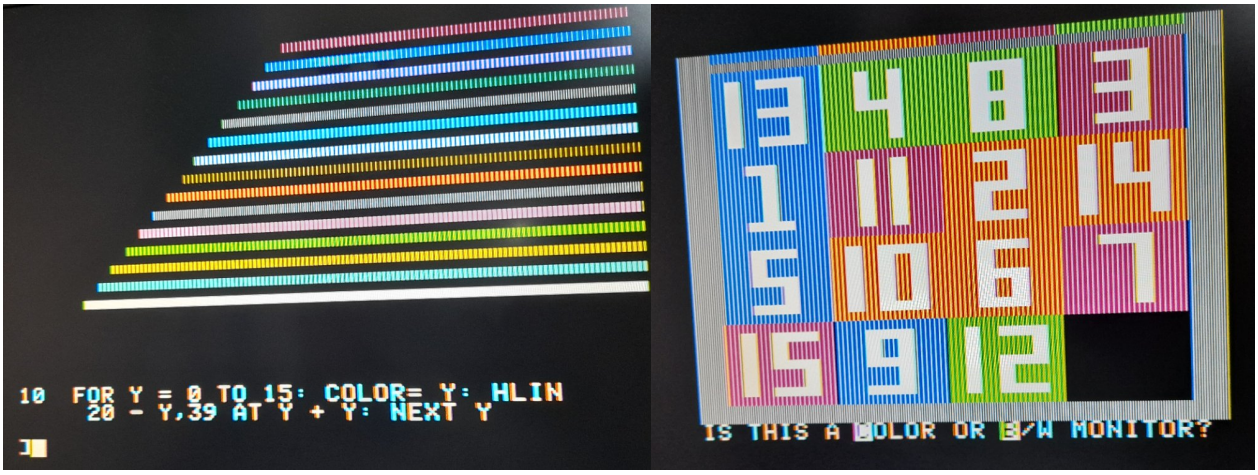


3.5 Low-Res graphics



After the NTSC color emulation was working, the next addition had to be the low resolution graphics video mode, that relies on the same color subcarrier modulation as the original Apple II. The former circuit diagram illustrates the way the low-res video output is generated. The main component here is a 4-bit shift register that is loaded with the low or high 4-bit bits of the video data depending on the current vertical line, or even with zero during blanking intervals. Then, the 4-bit pixel pattern is recirculated inside the shift register at a double clock rate until the register is loaded again 14 cycles later. The output is selected from the bit Q0 on even characters, or from bit Q2 on odd characters in order to avoid pattern disruptions due to characters having 7 pixels instead of 8 (Woz does more or less the same in the Apple II hardware). Finally, the current video output is switched between the low-res signal and the previously discussed video generation logic for the TEXT and HIRES graphics modes depending on the related control signals.

The actual circuit includes some subtle changes in order to get all the signals active at their proper times and with a correct phase relation with respect to the emulated color subcarrier, but its general picture is basically the same.



In the above snapshots the output of a simple Low-Res test program and the presentation screen of the only game I found using this video mode are shown. Notice the text is blurred in mixed video modes. This also happens in the real computer.

3.6 The 16K RAM expansion (Language Card)

This memory expansion is required in order to run Prodos. The original Apple II computer had no paged memory at all, only 48KB of RAM, and 12KB of motherboard ROM. Cards in slots can add more ROM, up to 256 bytes per card. But soon the need for more RAM was realized and the Language Card designed. This card basically replaces the computer ROM with RAM in order to provide support for applications other than BASIC without reducing too much the RAM available for the user. A convenient size for RAM was 16KB, but the ROM space is only 12KB. Therefore only 12KB of RAM can be addressed, but the lower 4KB can be selected from two possible banks, totaling 16KB of RAM.

In contrast with the floppy controller card, that can be described as a piece of electronic art, the Language Card is more or less the opposite and reflects the omission of some critical signals in the expansion slots, namely RAS and CAS, that resulted in a crappy memory card that, in addition to one slot, is also connected to a memory IC socket in the motherboard with a ribbon cable. And not only is the connection ugly, the way the paging is done is also quite weird.

But, anyway, that card became the standard expansion for 64K Apples and further revisions, like the Apple IIe included an equivalent paging circuit directly in the motherboard. In almost all cases the card was connected into slot #0, and therefore this slot is the one emulated in the FPGA replica (and also in the Apple IIe).

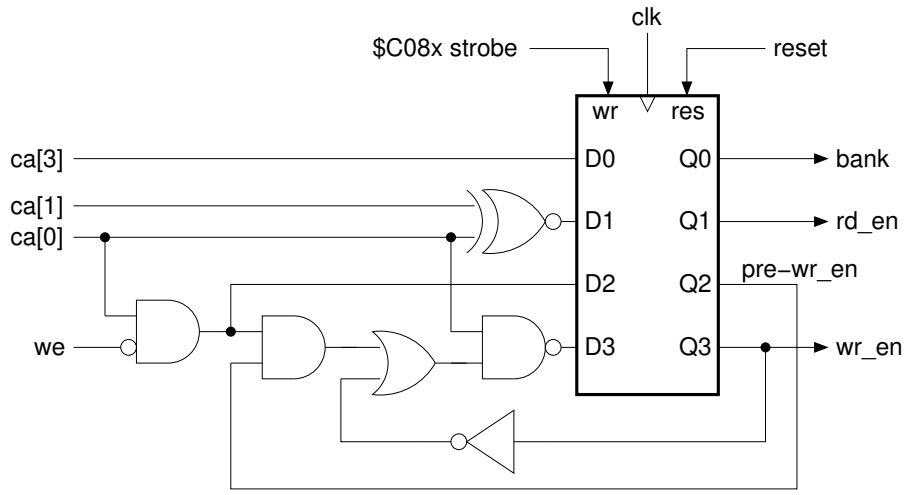
The FPGA board where the Apple II replica is being built has a 128KB SRAM. 64KB were already used, 48KB for the Apple II RAM, and 16KB for all the combined ROMs. This leaves the high 64KB of the SRAM unused, and the emulated Language Card can place its 16KB of memory there.

So, the main idea of the 16KB memory expansion design is to translate the 16-bit CPU address into a 17-bit memory address depending on the area being addressed and the control bits of the emulated card. These bits are stored in a 4-bit write only register (only 3 bits controls the memory, the other is used for a write enable delay), that are set or reset by means of reads to the following addresses:

LDA address	Read selects	Write exp. RAM	$Dxxx Bank	Notes
$C080	exp. RAM	no	2	
$C081	ROM	enabled	2	two LDAs required
$C082	ROM	no	2	
$C083	exp. RAM	enabled	2	two LDAs required
$C088	exp. RAM	no	1	
$C089	ROM	enabled	1	two LDAs required
$C08A	ROM	no	1	
$C08B	exp. RAM	enabled	1	two LDAs required

Notice that reads can be directed to ROM while writes are done into the expansion RAM, and that two consecutive reads of the control register are needed for the enabling of writes.

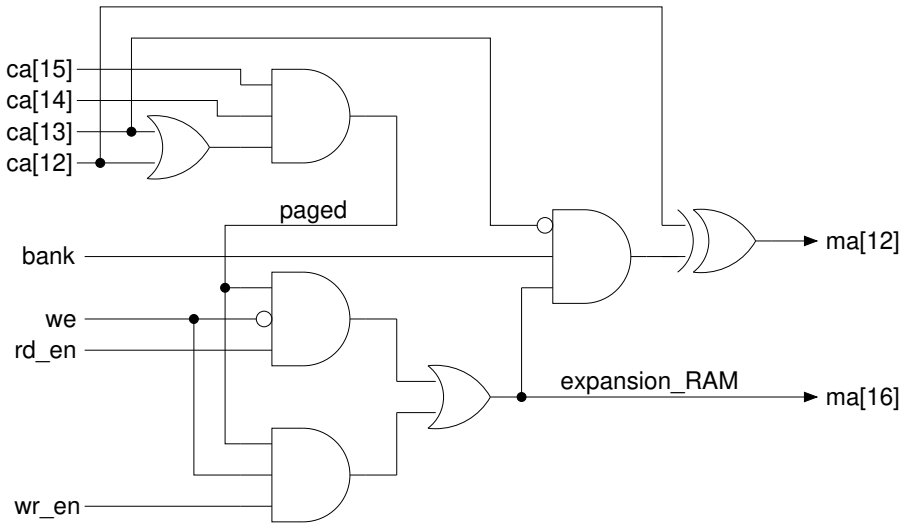
The circuit that implements this control register is basically the same as in the Language Card, the main difference is having all its output signals active high:



The register also includes an asynchronous reset input in order to start the system with all its signals in an inactive state. Then, the CPU address has to be modified in order to read or write the expansion memory areas when enabled:

CPU address	Exp. RAM enabled	BANK	SRAM address	Notes
$0000 to $BFFF	x	x	$00000 to $0BFFF	
$C000 to $C0FF	x	x	x	I/O Page
$C100 to $CFFF	x	x	$0C100 to $0CFFF	Slot ROMs, read only
$D000 to $DFFF	0	x	$0D000 to $0DFFF	ROM, read only
	1	0	$1D000 to $1DFFF	
		1	$1C000 to $1CFFF	
$E000 to $FFFF	0	x	$0E000 to $0EFFF	ROM, read only
	1	x	$1E000 to $1FFFF	

The expansion RAM is enabled if a read or write access is done to the $D000-$FFFF area when the corresponding enable signal is active. Also, notice that only the SRAM address lines ma[16] and ma[12] can differ from the CPU address. The remaining 15 lines are the same. The logic for address translation is:



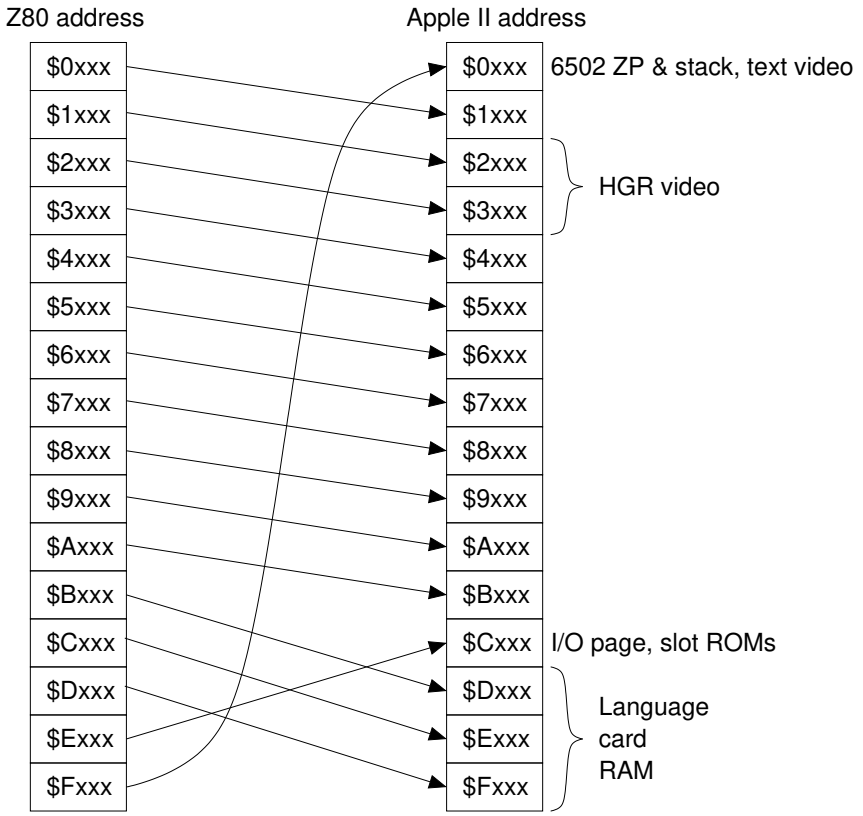
With these additions an effective Language Card is now installed in the Apple II replica. It is detected and tested by programs like Locksmith, and allows the booting of Prodos.

3.7 The Z80 Softcard

This card adds an alternative second CPU to the Apple II, in this case a Z80. This card was sold by Microsoft and allowed the Apple II to run CP/M. That computer has now two processors running, but not simultaneously: when one CPU is running the other is sitting in a waiting state. Both CPUs have access to all the memory and peripherals, but, at least in the CP/M implementation, the 6502 is executing all I/O related routines in an slave loop while the Z80 is in charge of the main code. The active CPU is selected by a flip-flop that toggles on writes to the following addresses (assuming the Z80 card is connected to its usual slot, #4):

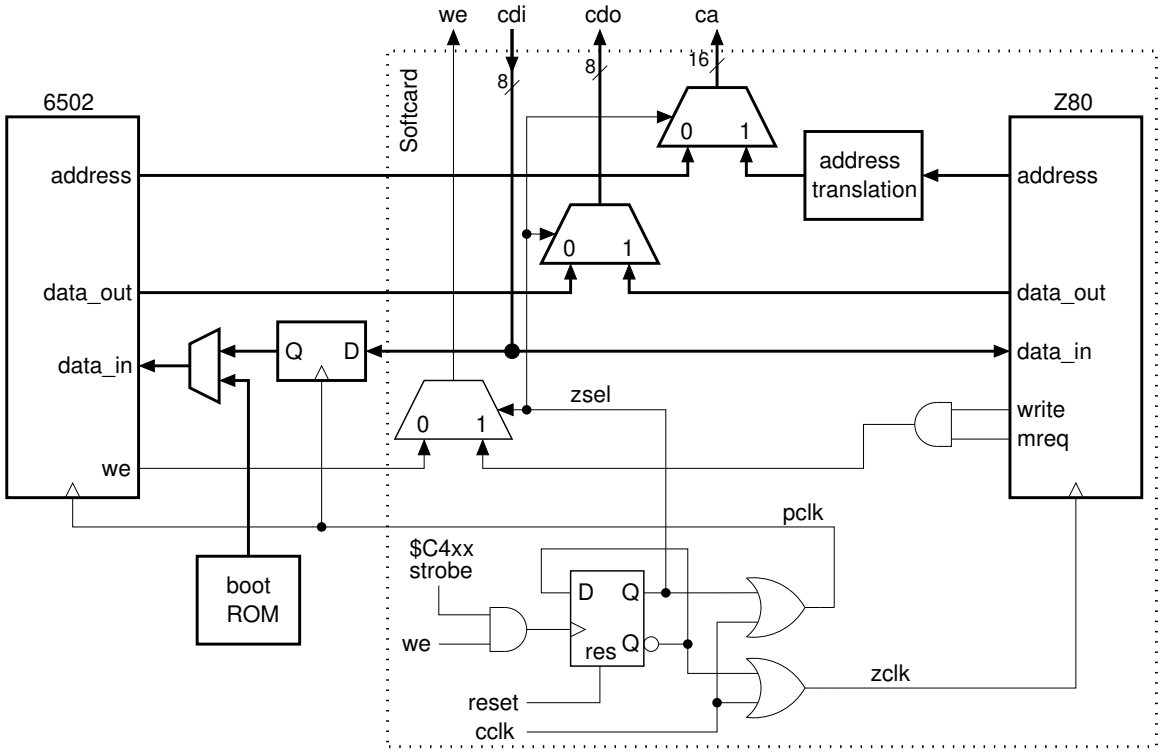
6502	Z80
$C4xx	$E4xx

The Z80 address is modified before being presented to the Apple II bus, and that explains the different addresses for the same register in the 6502 and Z80 cases. This avoids having the Z80 startup code located into the same memory region as the 6502 zero page, and tries to present the biggest possible block of free RAM to CP/M, also taking into account the memory expansion of the Language card. The actual address translation is shown next:



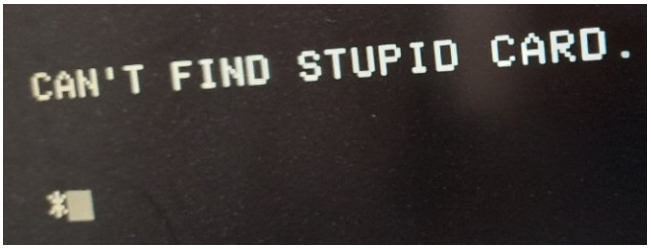
In the Softcard the Z80 clock runs at double the 6502 frequency, or 2.045MHz, resulting in more or less the same computing power as the main 6502.

The recreation of the Z80 card follows the next diagram:

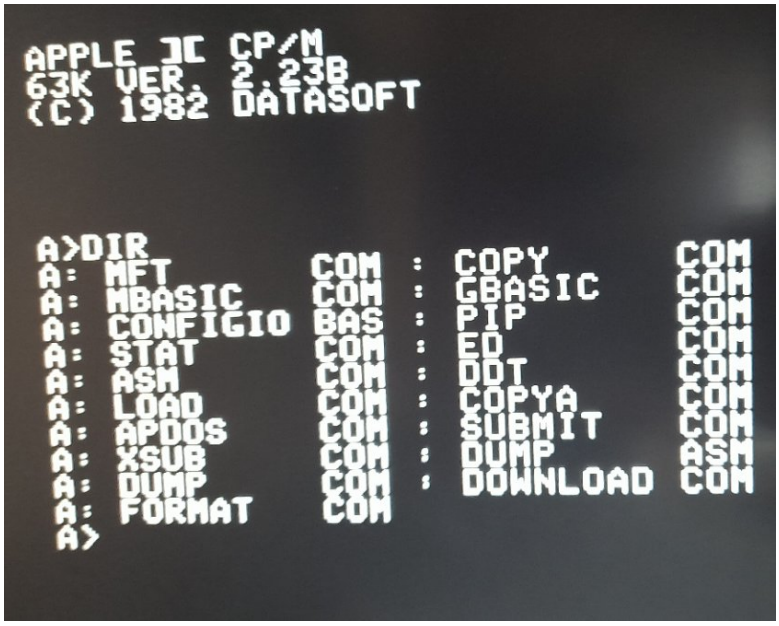


Here, the clocks of the CPUs are enabled only when selected, thus, forcing the unselected CPU to wait. Also, in addition to this logic, the clock generator changes its division factor when the Z80 is selected, resulting in a 2MHz clock for the Z80.

In the Verilog sources the Z80 card logic was made optional and only included if the symbol “SOFTCARD” is defined. When not included all the the Apple II bus signals come directly form the 6502 CPU and its clock is just “cclk”. In that case the CP/M disk fails to boot with a funny message:



It seems that back on those days some people had the same appreciation for Microsoft than today ;) If the Z80 is enabled the synthesis takes quite a long time and the number of allocated logic cells more than doubles (3648 LCs), but the recreated Apple II can now boot into CP/M:



3.8 Sound improved

The original sound emulation had some inconveniences:

- It can generate very short pulses in its output when the track or motor noises are active, meaning we have to turn the power transistors on and off very often, and this isn't very desirable (electromagnetic interferences, power consumption...).
- There are some time intervals (9.37% of the time) when the speaker is muted. This has a bad effect if a fast changing audio wave is played. In fact, a particular test program that used a sigma-delta modulation for audio generated much more noise than when the speaker flip-flop was routed directly to the output.

So, the sound emulation was redesigned in order to address these problems. Now, the main idea is to use a true PWM wave for the output with 8 bit resolution. With a 12.5MHz clock this results in a PWM sampling rate of 49KHz. The speaker output is integrated in an accumulator register during the 255 cycles of each sample and its resulting value is mixed with the other noise sources using adders. The resulting PWM level is:

$$PWM = 0.75 \sum_{i=0}^{254} speaker[i] + 16 \cdot tracknoise + 4 \cdot motornoise + 22$$

$speaker[i]$, $tracknoise$, and $motornoise$ are binary signals with values of 0 or 1.

The motor noise is now emulated by using a 13-bit PRBS generator clocked 5 out of 6 PWM clock cycles, and the track noise are pulses with a duration of 6 * 15 PWM cycles, or 1.84ms.

With these changes the sound is clearly better, specially that of the sigma-delta stream player, and the shortest output pulse is now 22 cycles long or 1.76μs.

4 The Apple IIe

4.1 New features

The 'e' version (for 'Enhanced') was a serious redesign of the Apple II, and a very successful one, indeed. Its main improvements were:

- 80 column text if a memory expansion card was installed. (1KB expansion was enough)
- Lowercase text font.
- Double Hi-res graphics (560x192) if a 64KB memory expansion was installed.
- More memory. 64KB on motherboard, and up to 128KB of RAM with a 64KB memory expansion.

The 80 column text and the Double Hi-res graphics demanded double the memory bandwidth than the old Apple II could provide and, consequently, the IIe includes an alternate memory bank that can be read in parallel with the normal one during video display (video data has an effective 16-bit data bus). That alternate memory bank is installed in a new slot connector specific for that memory. No high-resolution video modes are possible if there is no memory connected on that slot, and therefore, most Apple IIes came from the factory with a ridiculously small memory card of just one kilobyte that was enough for 80 column text (that memory stores the character codes for even columns, the odd columns characters reside in the main memory)

The motherboard came with 64KB of RAM where the upper 16KB are the equivalent of a Language Card installed on slot #0. Also, there are 16KB of ROM, with the lower 4KB located at the same address space as the slot ROMs ($C100 to $CFFF), so, in order to read the slot ROMs some switching is included. And there is also an specific ROM switch for the card in slot #3, where 3rd-party video cards were usually connected, because a virtual slot #3, 80 column, card is being emulated in the onboard ROM.

These new video modes and memory switching are controlled from a new addressable latch that only changes its bits during writes to the $C00x address range (the actual data written is ignored):

Switch	OFF	ON	Function
80STORE	$C000	$C001	Enables Alt RAM for PAGE2 video ($400 to $7FF) if TEXT / LORES ($2000 to $3FFF) if HIRES
RAMRD	$C002	$C003	Read Alt RAM ($200 to $BFFF)
RAMWR	$C004	$C005	Write Alt RAM ($200 to $BFFF)
INTCXROM	$C006	$C007	Slot ROMs if 0, Internal ROM if 1
ALTZP	$C008	$C009	Enables Alt RAM for zero page, stack, and high RAM ($0 to $1FF) and ($D000 to $FFFF)
SLOTC3ROM	$C00A	$C00B	Internal ROM if 0, slot #3 ROM if 1
80COL	$C00C	$C00D	Enables 80 column text / Double Hires
ALTCHAR	$C00E	$C00F	Enables lowercase font if 1

These control bits are not really intuitive and the documentation from Jim Shatter's "Understanding the Apple IIe" was really useful for their correct recreation. Also, the IIe allows the reading of many of these control bits back, something impossible on the older Apples. These bits are routed to the data bit #7 when reading the addresses reserved for keyboard flag reset:

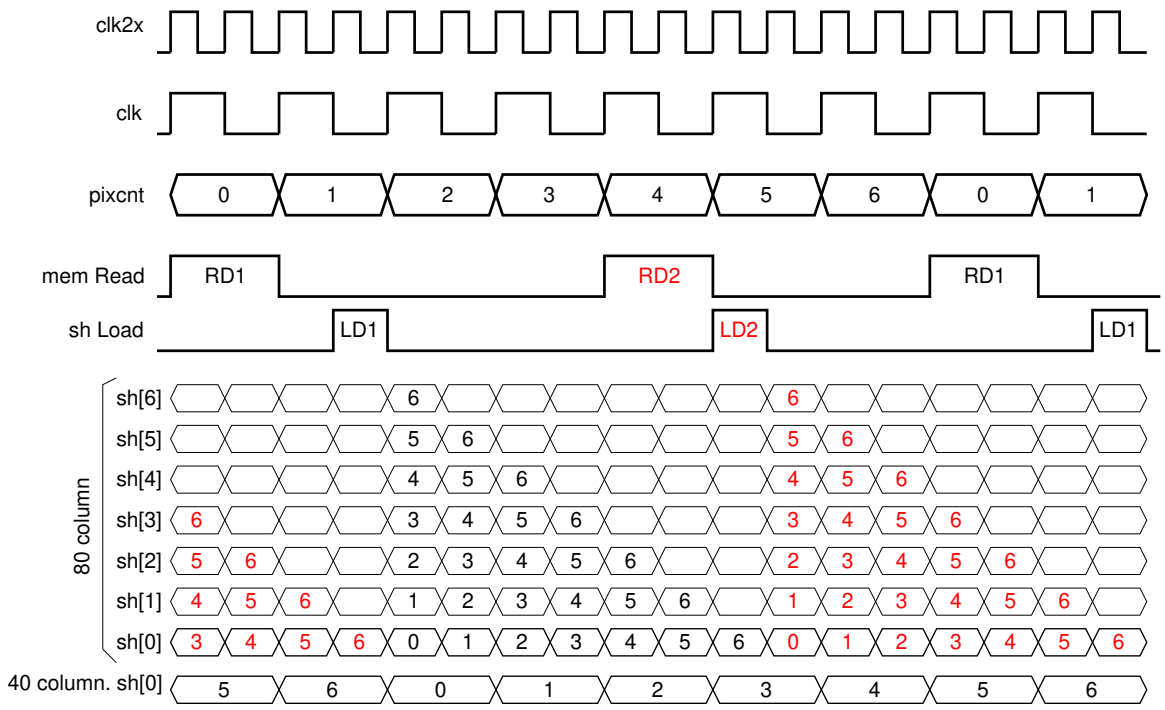
Address	Bit / action	Address	Bit
$C010	keyboard flag reset	$C018	80STORE
$C011	Lang. Card BANK (inverted)	$C019	VBLANK (inverted)
$C012	Lang. Card RD enable	$C01A	TEXT
$C013	RAMRD	$C01B	MIXED
$C014	RAMWR	$C01C	PAGE2
$C015	INTCXROM	$C01D	HIRES
$C016	ALTZP	$C01E	ALTCHAR
$C017	SLOTC3ROM	$C01F	80COL

4.2 IIe recreation

4.2.1 Double rate video (80 column)

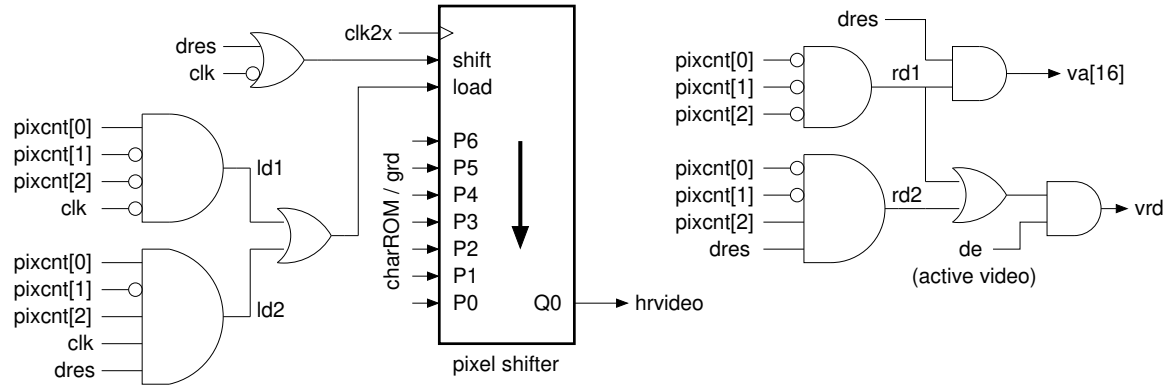
Without doubt the biggest attractive of the IIe was its 80 column text mode that turned a gaming computer into a serious one, at least if it was attached to a monochrome monitor. So, the recreation of this Apple II variant must start facing the problem of the generation of a video signal with double the pixel rate. This wasn't really as hard as it seems because the double rate clock is already in use for the low-res graphic mode, and we don't have to resort to a double width data bus because our RAM is a lot faster than that of the original Apple II (A 16-bit data bus is available anyway, but it isn't used in this design). We just have to insert a new memory read cycle into the time it takes for the displaying of a 40 column character row, or 7 clock cycles at 12.5MHz. The odd number of cycles is a nuisance, of course, but not really a serious problem. With two video read cycles every 7 we still have a lot of free memory cycles for the recreated CPU that only needs one memory cycle every 12 clock pulses.

With these considerations, the following chronograph for the video shifting was devised:



As we can see here, in 80 column mode we must add a second video read pulse, RD2, and a shift register load pulse, LD2. In 40 column modes these pulses aren't present and the shift register only shifts when CLK is low. So, in order to add support for 80 column video modes we only have to add the logic for the generation of these pulses, to change the shift register clock to 'clk2x', and to signal a memory address in the alternate space

for the RD1 pulse if in 80 column mode (alternate video memory data is displayed first). All of this looks like very little extra logic and it really is:



Here 'dres' signals a double pixel rate mode and this signal is generated from '80COL', 'TEXT', 'HIRES', 'MIXED', 'AN3', and also 'l160' because we can have a single resolution graphics window mixed with an 80 column text at the bottom of the screen. As we can see, in double res modes the pixel shifter is loaded when 'pixcnt' is 1 and 'clk' low, or when 'pixcnt' is 5 and 'clk' high, and the register shifts on all 'clk2x' cycles. If 'dres' is low 'ld2' is inhibited and the register only shifts when 'clk' is low. The logic for the generation of the video read signal, 'vrd', and the 17th video address line is also shown.

Notice that the pixel shifter, and the logic for 'rd1' and 'ld1', were already there, so, the generation of the double resolution video modes requires very little extra logic. Now, the available video modes are:

mode	TEXT	HIRES	80COL	AN3
40 column text	1	x	0	x
80 column text	1	x	1	x
Low Res (40x48)	0	0	x	x
Hi Res (240x192)	0	1	0	x
			x	1
Double Hi Res (560x192)	0	1	1	0

The double Hi res mode requires an additional enable signal that comes from the paddle interface (anunciator bit AN3). That mode was unavailable on the earliest revisions of the Apple IIe but soon a hardware mod was developed using that paddle bit and it became a 'de facto' standard.

Just another pair of details: In double res modes the 'PAGE2' signal is ignored by the video logic, so, only the main video page is available (but both on the main and the alternate memories, that BTW, are selected using PAGE2). And the same happens with the bit #7 of video bytes in graphics modes. There are no additional video delays depending on that bit. In fact, all pixels have to be delayed 1 cycle in order to obtain the correct colors.

4.2.2 Enhanced character ROM

The Apple IIe comes with a 2KB ROM for character generation instead of the 512 bytes ROM of the early Apples. This is 4 times the amount of memory, but its use is a bit inefficient because half the memory stores just the same pixels inverted. That was surely done in order to save a few XOR gates in the mainboard because 1KB ROMs were already obsolete or non existent when the IIe was designed. The recreation this ROM is going to use some FPGA BRAMs and I don't want to use more memory than really needed, so, this ROM is going to be 1KB and the pixel inversion is going to be done with XOR gates. This will keep the size of the new character generator ROM at just 2 BRAMs, that is enough for the displaying of the following 128 characters:

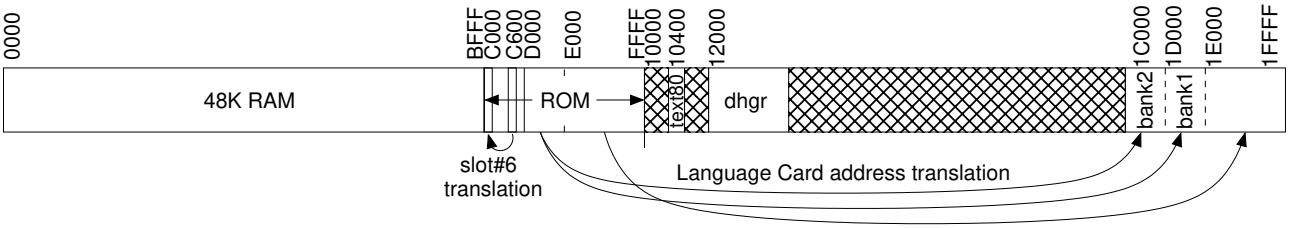
Where, as we can see, the complete ASCII character set is included, lowercase symbols too. But this means we now need 7 bits to address the character and there is no longer a blink attribute, just an invert one. The old behavior, with blink attributes can also be achieved, but at the expense of no lowercase symbols. The ALTCHAR switch will enable the whole ASCII set without blink characters when one, or maintains the text style of old Apples, when zero.

4.2.3 Memory allocation and management

It would be desirable to recreate a 128KB Apple IIe but the bare truth is there isn't enough memory for that. I'm short of just 3KB! and that's a pity. Well, I could recreate a IIe with up to 125KB, but all the software around assumes there are just two kind of Apple IIes: ones with 65KB (1 extra KB for 80 column text) and others with 128KB, and there are no cases in between, period. If the recreated computer is identified as an 128KB machine it's very probable for the code to try to use some nonexistent RAM area and to crash miserably, so, the amount of the alternate memory is limited to just the video pages and no more. This will still allow to play with the double hi-res graphics, but only when using our own code, other software will refuse to use this mode thinking the machine only has 65KB of RAM.

And there is still the problem with slot ROMs, in particular with the floppy disk controller ROM. In the previous Apple recreation these ROMs were merged with the mainboard ROMs and placed in the $C100 to $FFFF RAM area. But now all this area is occupied with the mainboard ROM. All of it? No! A village populated by irreducible Gauls still resists the invader. Well... I mean the first 256 bytes of the onboard ROM that are filled with $00 because that area is never addressed as ROM: this is the I/O page. And 256 bytes is just what we need for the floppy controller ROM, so I replaced the first 256 bytes of the IIe ROM with the Floppy ROM, and when a read is done in the range $C6xx the bits #9 and #10 of the memory address are forced low, thus translating the address from $C6xx to $C0xx.

With all these considerations this is the actual memory map:



All of these memory areas use the external 128KB SRAM of the FPGA board, while the internal BRAMs are used for other ROMs:

	Size (BRAMs)
Boot ROM	1
Character generator	2
Floppy state machine	1
PS2 keyboard mapping	1
YIQ to RGB color mapping	1

The FPGA has 32 BRAMs (16KB), and 6 are already used, totaling 3KB. There are still 13KB of unused internal memory, but in order to recreate an Apple IIe with 128KB I would need 3KB more.

4.2.4 Special keys

The Apple IIe added two new special keys to the keyboard: the open apple key and the solid apple key. But these keys weren't connected to the keyboard controller. They ended up being connected to the paddle interface as the paddle buttons #0 and #1. (It seems that the paddle interface was used for anything but paddles) The main use of these keys is for the modification of the reset behavior:

combination of keys	Effect
reset	warm reset (stop)
open-apple + reset	cold reset (reboot)
solid-apple + reset	enter diagnostics

In the IIe replica these keys are simulated by pressing F9 and F10 in the PS2 keyboard (F12 is the computer reset). In the Apple IIe the reset key usually had to be pressed along the control key because they are connected in series. This behavior was, no doubt, intended as a safeguard against accidental resets and is easy to replicate, it just requires a single AND gate, but I see no need for a three finger salute, and BTW, in the Apple keyboard the reset key can be connected directly to ground if some jumper wires are changed.

In the original computer the shift keys can also be optionally connected to the paddle button #2, but beware: If a paddle is connected and its button #2 is pressed at the same time than some shift key an electrical short between +5V and ground happens (shift key is active low, the paddle button high, but paddles usually have only buttons #0 and #1). This connection is also replicated (but without shorts ;)

And also there are two new regular keys on the IIe keyboard: the up and down arrow keys (ASCII codes 11 and 10). These keys are now included in the PS2 translation table.

5 Floppies on SD cards

It is very desirable to store floppy images on an SD card. The FPGA board already includes an SD connector that allows the use of these cards through an SPI bus. My solution here was to design an small computer that is synthesized along the Apple II replica and provides the interfacing between the SD card and the Apple disk controller in addition to allow the interactive selection of floppy images from the SD card. The main idea is to

extract the raw content of a disk track from the card into a memory area and to play it in an endless loop using DMA, thus simulating the bitstream of a floppy read head. A convenient size for a raw track bitstream is 6.5KB (53248 bits, or 13 SD card sectors), and we still have 13KB of the available internal FPGA RAM unused.

The floppy disk emulation computer, Floppyton-II (a ton of floppies for Apple II. No kidding, a 16GB card has the same storage capacity as 73000 single density 5 1/4" floppy disks, and that's more than one ton of floppies ;) is going to use much of this free memory, and these are its main features:

- 12KB of RAM memory with a known initial content. There is no ROM.
- A 25MHz, Z80, CPU. This processor core was selected in order to reuse some code for SD cards that I initially wrote for a ZX Spectrum. With almost 2200 logic cells this CPU is a bit bulky, but there are enough logic cells in the FPGA for the Apple IIe, the Z80 softcard, and the Floppyton.
- An OSD video display with a 32x24 text mode and transparent characters.
- A DMA channel for floppy read and write.
- A fast SPI controller.
- A PS2 keyboard controller.
- Interrupt logic for the generation of interrupts on track phase changes.

Memory is the main concern here. The DMA buffer is going to requite 6.5KB, but there are more thigns requiring storage:

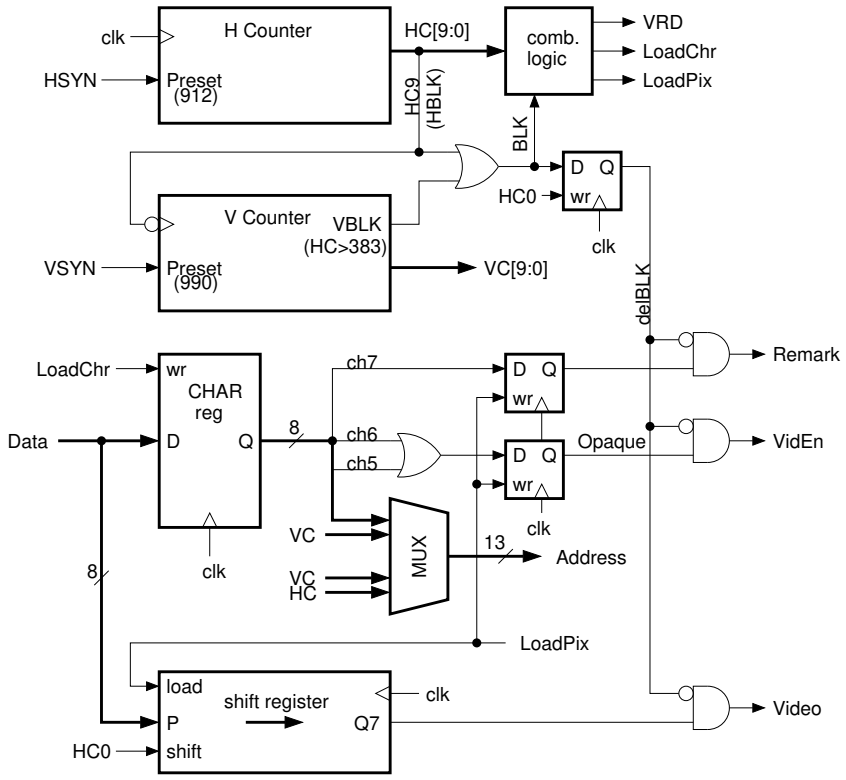
storage area	size (bytes)
disk DMA	6656
framebuffer	768
character generation	768
FAT buffer	512
Code & vars	3584

So, the memory available for the Floppyton's code and its variables is only 3.5KB. I hope this is still enough for an assembler implementation of the required functions if things are keept as simple as possible, like providing support only for SD-HC cards with FAT32 filesystems, DOS8.3 file names, and assuming disk image files aren't fragmented.

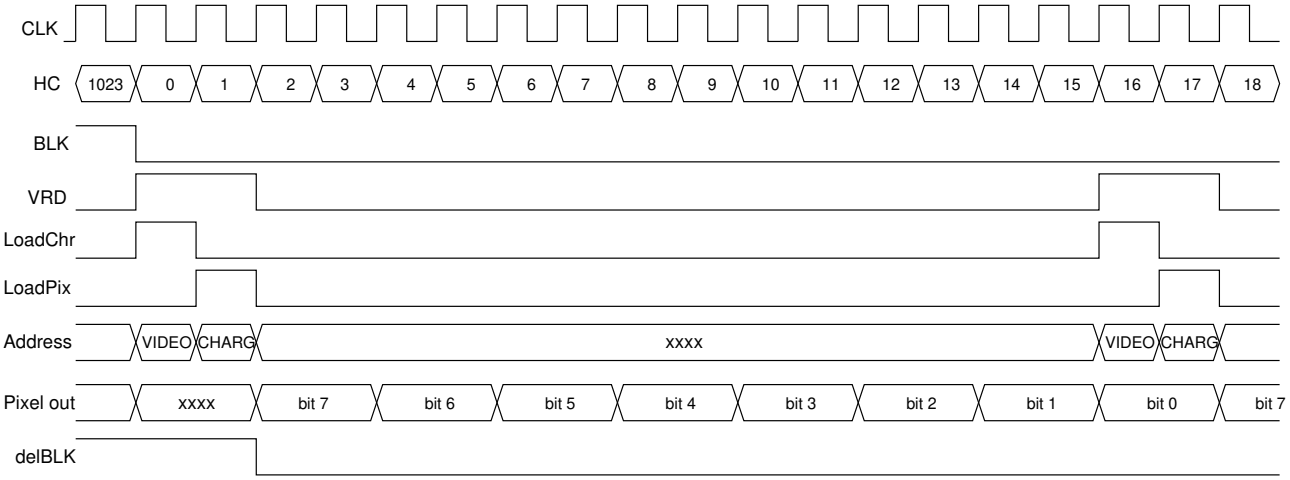
The video controller can display up to 24 lines of 32 characters each, but it also has some other peculiarities: first, it is an On-Screen Display (OSD) or in other words: an slave one. That means that instead of generating the horizontal and vertical sync pulses these signals are inputs and the internal horizontal and vertical counters gets preset by them. Also, the MSB of the character code is used as a remark bit that can change the color of the selected text. Not only that, characters with ASCII codes below 32 are transparent. This means that the video signal of the Apple replica is passed to the monitor instead of the text of the Floppyton computer. In this way the size and place of the floppy emulator window can be selected as desired, we only have to write all the other characters with zeros or any other value under 32.

The Floppyton also has a simple keyboard interface for the receiving of key scancodes. This interface is connected in parallel with the keyboard of the replica computer and this could cause interference when typing to the floppy emulator. In order to avoid this the Floppyton is able to "mute" the keyboard of the main computer when it pops on the screen. The <sys_req> key, that is unused by the replica, has the effect to show and hide the screen of the floppy player computer.

5.1 Text mode video



The block diagram of the text-mode video controller is shown above, where we should notice the *CHAR* register. This is where the ASCII codes of the text characters are stored. Other text mode video controllers include a character generator ROM with the pixels of the characters, but in this case the character generator data is stored in the same RAM memory as everything, and therefore, the memory has to be read twice for each 8-bit segment of a character: first the ASCII code is read from the video region of the memory and stored in the temporary *CHAR* register. Then, the memory is read again using the previously loaded ASCII code as part of the address and a 8-pixel data is finally transferred to the shift register.



In order to illustrate the inner workings of the video generator the chronograph of the previous figure shows the beginning of a video line. Here the horizontal counter, *HC*, was previously loaded with the value 924 when the HSYN pulse was active. Then it spent some time incrementing, and when it rolled over to zero the blank signal, *BLK*, is deasserted and the visible part of the line begins. But before shifting out any data we have to retrieve it from memory. The video read signal is thus asserted for the duration of two clock cycles. During this time the clock of the Z80 is stopped and the address presented to the memory comes from the video generator. During the first cycle the address is for video memory and the destination register for the data is the *CHAR* register. The video address is:

A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	0	1	1	VC[8:4]					HC[8:4]				

Where *VC* and *HC* are bits of the vertical and horizontal counters. Then, in the next cycle the address presented to memory is for the character generator table, and it is:

A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	ASCII code (7 bit)							VC[3:1]		

The data read during this second cycle is stored in the shift register of the video generator, and pixels can start to come out towards the screen.

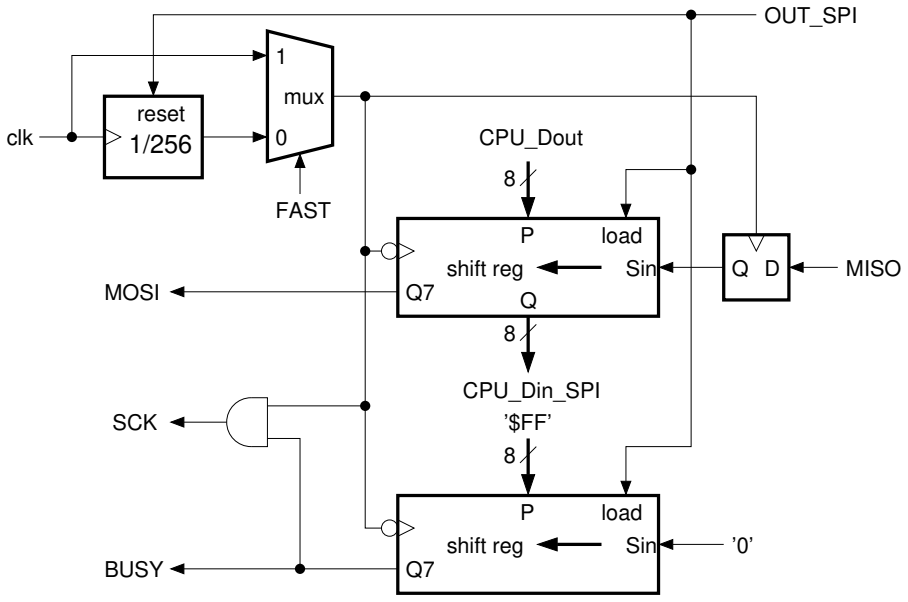
Notice that VC[0] isn't used in the generation of memory addresses and that means that video lines are displayed twice. HC[0] is also used as an enable signal for the shift register, so, 8 pixels take 16 cycles to be shifted out. The net result is to have each pixel of the floppy player as a 2x2 VGA pixel block. The 4 MSB bits of these addresses are 1011 for the video region and 0000 for the character generator. Therefore, the character generator is located at the beginning of the memory, but when the character code is lower than 32 the *VidEn* output is low, disabling the floppy player video. So, the first 256 bytes of the memory are never displayed on the screen and they can be used for other purposes, like for the starting code of the Z80 or its interrupt routine.

Also, the video memory starts at the last kilobyte of the memory but only 768 bytes are displayed, leaving 256 unused bytes at the end that can be assigned for instance as the stack area.

So, at the end we got the following memory map:

Start Address	Length	Use
$0000	256	Startup code, Interrupt routine
$0100	768	Character generator table
$0400	~	Application code
~	~	Application variables
~	~	Free
$1200	6656	DMA buffer
$2C00	768	Video memory
$2F00	256	Stack

5.2 The SPI controller



The block diagram of the SPI peripheral is shown above. It is a simple but very effective interface. Only SPI mode #0 is supported and that means output data have to be shifted on the falling edges of the clock but input data have to be sampled on rising edges. Therefore an additional flip-flop is included to sample the *MISO* signal. The clock can be selected from the main clock (25MHz) or from an 1/256 prescaler. The slow clock is going to be used during the initialization phase of the SD card, while the fast clock is intended for the rest of the SD card operation. The controller includes two 8-bit shift registers, one is for the actual data while the other is used for control. This last register gets loaded with all ones when an OUT (SPI) instruction is executed, and consequently, sets the *BUSY* signal during 8 *SCK* cycles. The status of the *BUSY* signal can be read from an input register, but when the clock is fast there is no need for any polling because an SPI transfer takes less time (8 cycles) than the fastest polling period, so, an SPI transfer could be performed with just two assembler instructions:

```
OUT (SPI),A     ; Send data in Acc
IN  A,(SPI)     ; Received data in Acc
```

Of course, with the slow clock the polling is absolutely necessary because now the SPI transfer takes 2048 cycles.

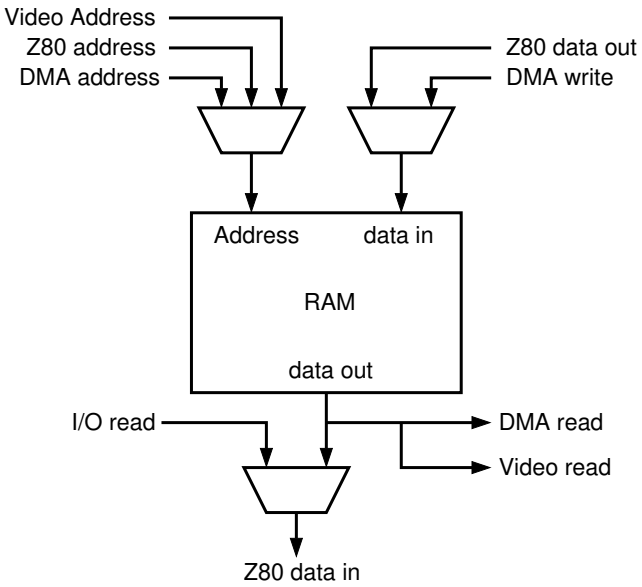
5.3 DMA

In the Floppyton computer there are three memory masters, that, in priority order are:

memory master	Priority	master when
Video controller	high	*vrd* active
Z80		*vrd* inactive, *mreq* active
DMA	low	*vrd* and *mreq* inactive

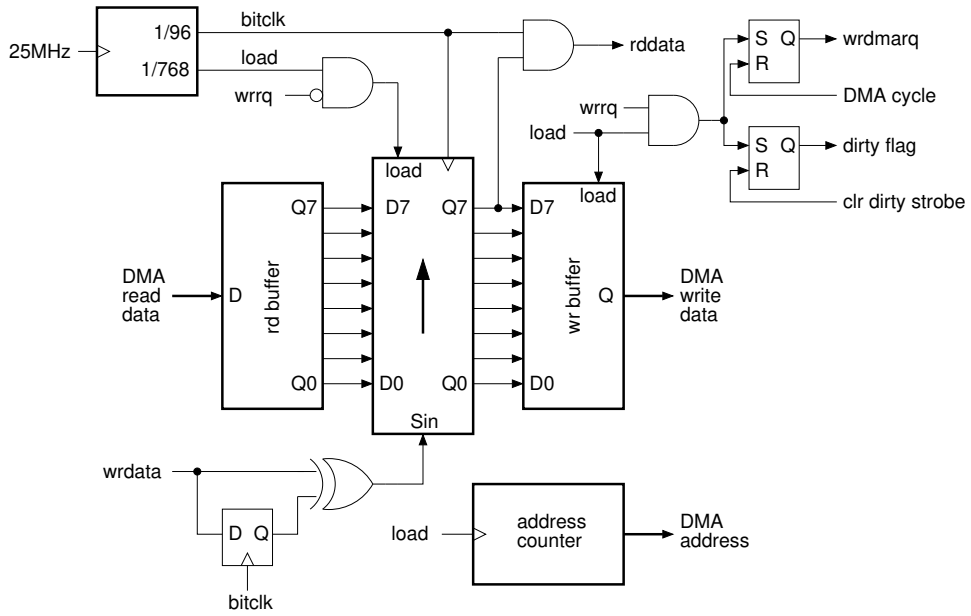
As we can see the DMA controller can only access the memory when neither the video controller nor the Z80 are accessing the RAM, but there are still a lot of cycles available for DMA that only requires a read or write cycle every 8 floppy disk bits, that take more or less $31\mu s$ or 768 main clock cycles.

with respect to the memory interface, it follows the next diagram:



The DMA interface for disk reading includes a 14-bit address counter, an 8-bit buffer register, a shift register, a clock divider, and a simple AND gate for the translation of ones into pulses. In the case of disk writing the address counter, the clock divider, and the shift register are the same, but there is a separate 8-bit

buffer for writes along with a dma-write request bit and a *DIRTY* flag that notifies the change of the memory contents due to DMA. The input bitstream is also decoded from NRZI to NRZ by means of a simple delay flip-flop and a XOR gate. The address counter is always incrementing, and when it reaches the end of the DMA buffer it gets reloaded with the base address automatically. A simplified diagram for this logic is shown next:



The programmer's model for the DMA controller is very simple: it consist of a single enable bit, a dirty flag, and that's all folks. The DMA can be disabled for the time it takes to reload or to save the content of the track data from or to the SD card, or when the memory of the DMA buffer is used for other purposes, like the storage of directory entries during the interactive selection of files.

5.4 I/O map

In the floppy player there are only 6 I/O addresses used:

Addr[1:0]	type	register
0	R/W	CONTROL1 / STATUS1
1	R/W	SPI
2	RO	KEYBOARD
3	R/W	CONTROL2 / STATUS2
4	WO strobe	CLEAR EDGES
5	WO strobe	CLEAR DIRTY

The last two I/O addresses are simple write strobes for the clearing of flags and the actual value written to these addresses is ignored. The *CONTROL1 / STATUS1* register needs further description. Its bits are:

	7	6	5	4	3	2	1	0
OUT	GRAB	FAST	EN_DMA	/CS2	x	x	x	x
IN	GRAB	FAST	EN_DMA	/CS2	DRVSEL	DRVON	BUSY	KVAL

Where *GRAB* is an output bit that mutes the keyboard in the replica computer, *FAST* and *BUSY* are used in the SPI controller, *EN_DMA* activates the disk DMA, */CS2* is the chip select signal of the SD card, *DRVSEL* is the selection of the floppy drive 1 or 2, *DRVON* is the signal of the replica computer that activates the floppy motors, and *KVAL* is a bit that signals a received keyboard scancode when high. This last bit is cleared when the *KEYBOARD* register is read.

The *CONTROL2 / STATUS2* register have the following bits:

	7	6	5	4	3	2	1	0
OUT	x	x	x	x	x	x	WROK2	WROK1
IN	RPHS3	RPHS2	RPHS1	RPHS0	x	DIRTY	WROK2	WROK1

RPHS[3:0] are bits that get set when there is a rising edge on the corresponding stepper motor phase signal, and reset when writting the *CLEAR EDGES* strobe. If any of these four bits is set an interrupt is requested to the Z80 and, in this way, the Floppyton keeps the current drive track updated. *WROK1* and *WROK2* signal the read-write status of the two emulated drives when set, or their read-only status when reset. And finally, the *DIRTY* bit gets set when a DMA write cycle is performed, meaning we have to write the track data back to the SD card. This bit is cleared by means of a write to the *CLEAR DIRTY* strobe.

The keyboard interface is little more than a simple shift register and I don't think it deserves much explanation. I only want to remember that the reading of its register, for example with the "`IN a,(2)`" instruction, also clears the *KVAL* flag.

5.5 Firmware details

The Floppyton code can be split into three main components:

1. An interrupt routine that samples the rising edges of the stepper motor phases and updates the track position variables *trk1* and *trk2* (one per drive).

2. An interactive loop where the directory listing of the SD card is shown to the user and the selection of floppy image files performed. The Apple II keyboard is muted during this time.

3. A floppy emulation loop with the Floppyton's screen hidden. During the execution of this code any track change will result in the reading of a whole track (13 sectors) from the image file in the SD card to the DMA buffer, that can be also preceded by the storing of the buffer in the image file if the *DIRTY* flag is set (this only happens on write-enabled disks). The *DIRTY* flag is also checked when the *DRVON* signal goes low in order to flush the modified track data to the disk image.

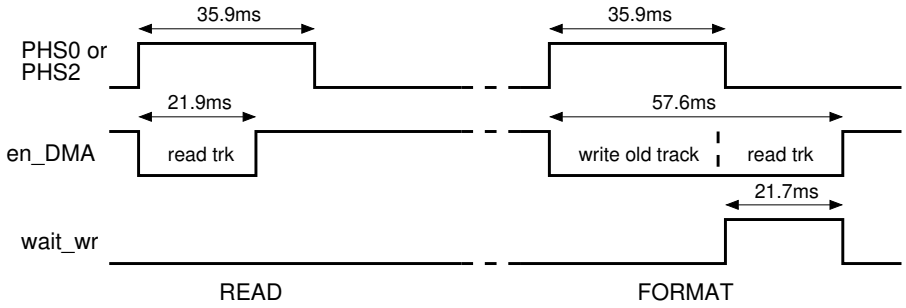
The following PS2 keys are used to control the Floppyton computer:

Key	Action	
F11	Reset	
Sys Req.	Show / Hide Floppyton screen	
Cursor keys	Select file / navigate current folder	
Enter	Enter subdirectory	Assign RAW image file to drive #1
Space		Assign RAW image file to drive #2
1	Toggle Read-only flag for drive #1	
2	Toggle Read-only flag for drive #2	

5.6 Flow control

The SD card when used in SPI mode is quite slow (~300KB/s measured during reads), but it is still an order of magnitude faster than the real floppy disk, meaning it can read a whole track in about 20ms, and this is less than the settling time of the disk head. But that was an average figure, SD cards can have latencies as long as 100ms as stated in the SD Card Specification. But for reads this isn't a problem because the Apple II is waiting for the data and it simply takes a little longer if some big latency happens during reads.

This isn't a problem for sector writes too because the Apple has to read the address field of the sector before writing, and this forces the Apple to wait until the Floppyton has updated the track data, but for format commands it can really be a problem because nothing gets read before a write, and some track data can be lost.



In the previous figure the waveforms measured in a logic analyzer during read and format commands are shown. The first trace is the logical OR of the phase #0 and phase #2 signals for the stepper motors, so, when a track changes we get a pulse in this signal. The last pulse is almost 36ms long, and the Apple II waits until its falling edge before performing any read of write (well, some programs like Locksmith can generate shorter pulses, around 20ms).

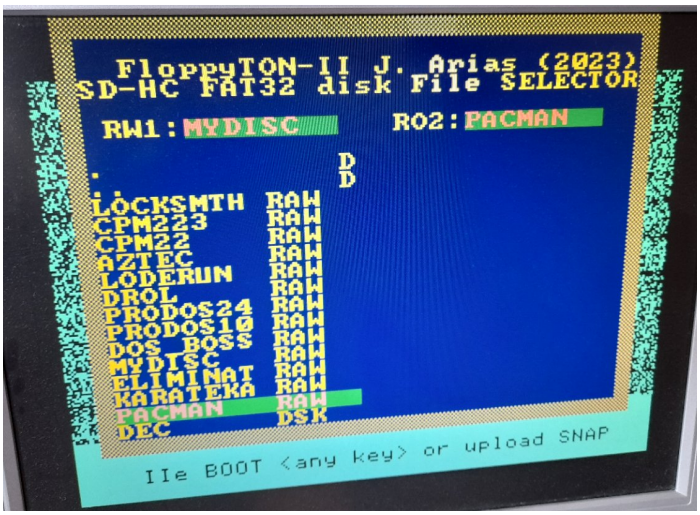
In the Floppyton a rising edge on PHS0 or PSH2 will signal a track change and the contents of the DMA buffer have to be updated. During this update the DMA is disabled, as we can see in the middle trace. If no writes were done to the buffer the old track data is discarded and a new track is read from the SD card. The reading takes about 22ms (but in a worst case it could take more than 100ms), and when the phase signal goes low the new track is already loaded in the buffer and the DMA enabled.

But if the old track was modified its data has to be saved on the SD card before reading the new track content and the update takes a longer time, about 58ms, meaning the Apple II has to wait until the new track is loaded. In the case of reads there is nothing special to do, but for format commands the Apple II will start to write data into the new track as soon as the phase signal goes low (well, about 58μs later), and any data sent to Floppyton while its DMA is disabled will be lost.

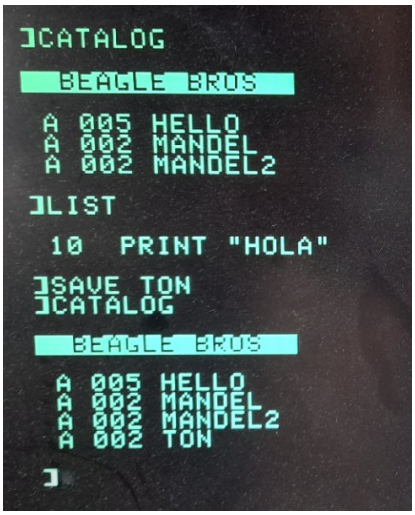
So, a wait for write signal was generated when the DMA is disabled and the write-request signal asserted. This signal is shown as the third trace and last about 22ms. As expected, it only gets asserted during format commands and it is used to stop the Apple II clocks (both the CPU and the 2MHz disk controller clocks) when active, thus forcing the recreated Apple II to wait until the Floppyton finishes the track update.

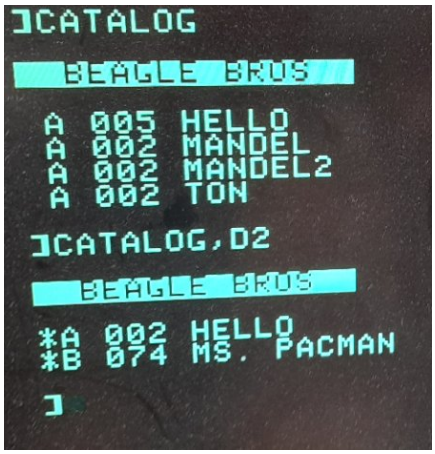
Well, this kind of flow control is really disgusting. It would be way better to do the track write and read in less than 20ms. This would demand faster read and write commands in the Floppyton code, like “read multiple sectors”, but, even if this is achieved it is only for the average case, while we can still have long latencies from time to time (I got a capture where the track update took 143ms instead of the usual 57ms). So, we have to live with this sort of flow control, but fortunately this only happens during format commands.

5.7 Examples



And now some snapshots. The previous one shows the selection window of the Floppyton computer where the file listing of a card folder is displayed. Under this window is the boot screen of the recreated Apple IIe that is waiting for a key press, but that Apple won't receive any key press until the Floppyton window is closed (pressing <sys req>). The disk image files have the extension .RAW, meaning they store the raw bitstream of the floppy tracks. These files are very similar to .WOZ files, in fact if you remove all metadata from a .WOZ file (in some cases by just skipping the first 256 bytes) you'll get a .RAW image suitable for Floppyton, but the files in the snapshot were generated by means of the encoding of .DSK images with a custom application. The image file "MYDISK" is selected for drive #1 and write enabled, while for drive #2 we have the "PACMAN" image as read only.





Now let's present some use of the emulated floppy drives. In the left snapshot a new file, "TON" is created in the floppy image after booting. In the right snapshot the contents of the same disk are listed again after a cold reboot to prove the new file is indeed stored in the card, and also the catalog of disk #2 is displayed.

As we can see, Floppyton deals quite well with disk writing, at least if writes are performed at the sector level. But, if a whole track is written as a single block the resulting disk image gets somehow corrupted. This will happens if a disk copy application (like Locksmith) is used, or when a floppy is formatted with an "INIT" command. So, a little more debugging was still required. The Locksmith's speed test tool was really useful in finding the tracks were too long. 6656 bytes per track was a convenient size for the storage because it is an integer number of card sectors (13), but the emulated track has to be shorter, with 6392 bytes giving a good 300 rpm in the speed test. Thus, tracks are still stored as 6656 bytes but the last 264 bytes are never played by the Floppyton DMA. With this mod, and the previously commented method for flow control, Locksmith is now able to format blank disks and so does the "INIT" command.

Another interesting thing I discovered after I got the format commands working was related with the interleaving of sectors. .DSK floppy images assume the sector sequence {0, 13, 11, 9, 7, 5, 3, 1, 14, 12, 10, 8, 6, 4, 2, 15}, and I though that was the default order of sectors on tracks. But at least with DOS 3.3, after an INIT command the sectors are arranged in a simple sequential order, 0 to 15, and the resulting disk is much faster to read than the one with the weird interleaving. So, now my application for the creation of .RAW images rearranges the sectors into a sequential order, and sure, booting is way faster than before.

6 A worm in the apple

My initial though about the Floppyton computer, and also for the closely related Amstrad tape player, was to use my own CPU design, the GUS16, for their cores. But at the end I resorted to use a Z80 instead because I already had the FAT32 code written for a ZX Spectrum. It wasn't until I got everything working that I reconsidered the move to the original plan.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags				operation
0	0	0	0	0	RD			0	RA			–	RB			ADD	C	V	N	Z	RD= RA+RB
0	0	0	0	0	RD			1	RA			ulit4				ADDI	C	V	N	Z	RD= RA+ulit4
0	0	0	0	1	RD			0	RA			–	RB			ADC	C	V	N	Z	RD=RA+RB+Cflag
0	0	0	0	1	RD			1	RA			ulit4				ADCI	C	V	N	Z	RD=RA+ulit4+Cflag
0	0	0	1	0	RD			0	RA			–	RB			SUB	C	V	N	Z	RD=RA−RB
0	0	0	1	0	RD			1	RA			ulit4				SUBI	C	V	N	Z	RD=RA−ulit4
0	0	0	1	1	RD			0	RA			–	RB			SBC	C	V	N	Z	RD=RA−RB−(~Cflag)
0	0	0	1	1	RD			1	RA			ulit4				SBCI	C	V	N	Z	RD=RA−ulit4−(~Cflag)
0	0	1	0	0	–	–	–	0	RA			–	RB			CMP	C	V	N	Z	RA−RB
0	0	1	0	0	–	–	–	1	RA			ulit4				CMPI	C	V	N	Z	RA−ulit4
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			ulit4				ANDI	–	–	N	Z	RD=RA&ulit4
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			ulit4				TSTI	–	–	N	Z	RA&ulit4
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			ulit4				ORI	–	–	N	Z	RD=RA|ulit4
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			ulit4				XORI	–	–	N	Z	RD=RA^ulit4
0	1	0	0	1	RD			0	–	–	–	–	RB			NOT	–	–	N	Z	RD=~Rb
0	1	0	0	1	RD			1	–	–	–	–	RB			NEG	–	–	N	Z	RD=−RB
0	1	0	1	0	RD			0	–	–	–	–	RB			SHR	C	?	N	Z	RD=RB/2, Cflag=RB.0
0	1	0	1	0	RD			1	–	–	–	–	RB			SHRA	C	?	N	Z	RD=RB/2, Cflag=RB.0 (signed)
0	1	0	1	1	RD			0	–	–	–	–	RB			ROR	C	?	N	Z	RD=(RB>>1)|(Cflag<<15), Cflag=RB.0
0	1	0	1	1	–	–	–	1	–	–	–	–	–	–	–	ILEG					
0	1	1	0	0	RD			0	RA			d[3:0]				LD	–	–	N	Z	RD=Mem[RA+d]
0	1	1	0	0	d[2:0]			1	RA			d[3]	RB			ST	–	–	–	–	Mem[RA+d]=RB
0	1	1	0	1	RD			0	–	–	–	ulit4				ADPC	–	–	–	–	RD=PC+ulit4
0	1	1	0	1	–	–	–	1	–	–	–	0	RB			JIND	–	–	–	–	PC=RB (indirect jump)
0	1	1	0	1	–	–	–	1	–	–	–	1	–	–	–	RETI	–	–	–	–	Interrupt return
0	1	1	1	0	RD			–	–	–	–	–	–	–	–	LDPC	–	–	–	–	RD=Mem[PC++]
0	1	1	1	1	RD			ulit8								LDI	–	–	–	–	RD=ulit8
1	0	0	0	lit12 (signed)												JZ	–	–	–	–	jump if Zflag=1
1	0	0	1	lit12 (signed)												JNZ	–	–	–	–	jump if Zflag=0
1	0	1	0	lit12 (signed)												JC	–	–	–	–	jump if Cflag=1
1	0	1	1	lit12 (signed)												JNC	–	–	–	–	jump if Cflag=0
1	1	0	0	lit12 (signed)												JMI	–	–	–	–	jump if Nflag=1 (negative)
1	1	0	1	lit12 (signed)												JPL	–	–	–	–	jump if Nflag=0 (positive)
1	1	1	0	lit12 (signed)												JV	–	–	–	–	jump if Vflag=1 (overflow)
1	1	1	1	lit12 (signed)												JR	–	–	–	–	unconditional jump

The GUS16-V5 instruction set

The GUS16 is a quite compact core requiring only 673 logic cells, less than the 6502 core (789) and much less than the Z80 core (2247), while it easily outperforms any of these 8-bit processors. Of course, I chose the latest core version that includes the very useful load and store instructions with small address displacements. Lets present the main features of this core:

- 8 general purpose registers, R0 to R7. The instruction set is orthogonal: these registers are completely equivalent to each other, but I assigned some defined functions for two registers, namely:
  - R7 is used as an stack pointer
  - R6 is used as a link register (it stores the return address for subroutines)
- A 16-bit program counter, PC. This special register can be used as a source operand for the instruction ADPC Rx,lit4 (add a positive 4-bit literal to PC and store the result in Rx), for the LDPC Rx, (load Rx from the memory pointed by PC instead of executing that data as an instruction) and, of course, for relative jumps.
- A 4-bit flags register (Carry, Zero, Negative, and oVerflow). These bits are modified by arithmetic and logic instructions and their values are checked during the execution of conditional jumps.
- Alternate PC and Flags registers for their use during interrupts. The normal PC and flags remain with their values unchanged during the execution of an interrupt routine. The return from interrupt instruction, RETI, switches the PC and flags registers back to the normal set. Up to 7 interrupts with independent vectors can be routed to the core, (vector addresses: base address (core parameter) +4, +8, +12, +16, +20, +24, and +28)
- An instruction set that includes three-operand versions of the usual arithmetic, logic, and arithmetic with carry instructions. The same instructions also have an immediate variant where the second source operand is an unsigned 4-bit literal value. Other instructions include complements, comparisons, right shifts and rotations, load and store instructions, 8-bit literal load, PC-relative jumps (conditional and unconditional), jump indirect (JIND Rx), where PC is loaded with the content of an Rx register (this is the usual way to return from a subroutine), and return from interrupt. These instructions are summarized in the preceding table.

The GUS16 is more programmer's friendly than the Z80 and a lot more than the 6502. But there are also some issues related to its design that have to be addressed:

- The core knows nothing about bytes, its only data type is the 16-bit integer, and its memory space comprises 65536 16-bit words. This resulted in some complications, mainly when dealing with character strings.
- As there are no bytes it makes no sense to talk about endianess. We can store two bytes in a 16-bit word any way we like, but we have to select an endian convention in our software in order to deal with bytes always in the same way. I chose a big-endian type because in the new computer there is also a 16-bit SPI controller and SPI data comes with its most significant bit first. This was very convenient for character strings, but it also resulted in 16-bit and 32-bit values (like those found in FAT tables) having their bytes swapped. In order to simplify the reading of these values, and taking into account the small memory of this computer (only 6 Kwords), I resorted to present to the core the input data bus with its bytes swapped if the address line A15 is high. This avoids having to use a lot of rotations to swap the bytes, we just have to read the same data in the high memory space. As an example lets present what is read at the following addresses:

Address	Value
0x0084	0x3078
0x8084	0x7830

And also it is included next the code of an small routine that reads a byte from memory using that memory swapped area. (The byte address is just the memory address multiplied by 2 for word aligned bytes (MSB) or the same address plus one for the LSB):

```
;-------------------------------------------------------------------------
;                       read a byte
;-------------------------------------------------------------------------
; Parameters:   R1: byte address (=mem_address *2 +odd/even)
; Returns:      R0: byte read
; Modiffies:    R1++
; Notes:        Big endian memory assumed

; using byte swapped memory
rdbyte: subi    r7,r7,1     ; save R6 on the stack
        st      (r7),r6
        ldi     r0,0xff     ; byte mask
        xori    r6,r1,1     ; LSB at 0 means high byte (big endian)
        ror     r6,r6       ; ~LSB goes to carry
        ror     r6,r1       ; Now carry is on the MSB of R6
        ld      r6,(r6)     ; read memory (swapped bytes if R1's LSB was 0)
        and     r0,r0,r6    ; mask upper byte
        addi    r1,r1,1     ; increment byte pounter
        ld      r6,(r7)     ; restore R6 from stack
        addi    r7,r7,1
        jind    r6          ; and return
```

The byte swap is done only for memory reads. Words are always written without swapping. Therefore the write byte counterpart is a more complex routine with a read-modify-write code and 8 data rotations.

- There are no IN / OUT instructions in the GUS-16 core. The pheripheral's registers have to be memory mapped. The address range 0x20 to 0x3F is reserved for I/O registers:

Read register		Bits								
addr	name	15	14-8	7-6	5	4	3	2	1	0
0x20	IRQEN	x								IRQEN
0x21	PFLAGS	DIRTY	x				KVAL	SPIBUSY	UTXBUSY	URXVAL
0x22	UART	0	UART RX							
0x23	CONTROL	DRVSEL	x		WROK2	WROK1	GRAB	SPIFAST	EN_DMA	CSB
0x24	SPI	SPI_RX_MSB / 0	SPI_RX_LSB							
0x25	KEYBOARD	0	KEY_SCANCODE							
0x26	FLAGS2	DRVON	x				PHASE_RISING			
rest	-	x								

Write register		Bits							
addr	name	15-8	7-6	5	4	3	2	1	0
0x20	IRQEN	x							IRQEN
0x21	CLREDGE	x							
0x22	UART	x	UART TX						
0x23	CONTROL	x		WROK2	WROK1	GRAB	SPIFAST	EN_DMA	CSB
0x24	SPI16	SPI TX (16-bit)							
0x25	SPI8	x	SPI TX (8-bit)						
0x25	CLRDIRTY	x							
rest	-	x							

Here, most of the flags and control bits are grouped into the lower 4 bits of registers in order to use the ANDI, ORI, and TSTI instructions. The exceptions are the control bits WROK1 and WROK2, and some flags located at bit 15 that can be tested directly with JPL or JMI.

This memory area isn't really lost because the I/O registers are only selected when the address lines A[15:13] are zero but these lines aren't connected to the memory (A12 is the most significant address line used by the memory). Therefore the memory contents at address 0x20-0x3F can be accessed at base addresses 0x2020, 0x4020, and 0x6020. This trick is used in the interrupt routine that starts at address 0x2004 and ends at 0x202A.

The computer includes a very simple UART into its peripherals that was intended as a debugging aid. This peripheral isn't needed for the normal use of the FloppyGUS and could be later removed from the design. This UART uses about 70 logic cells.

- The GUS-16 access the memory all the clock cycles, so, it has to be the lowest priority memory master or other devices couldn't read nor write the memory. In our case the disk emulation DMA has now more priority than the CPU, meaning the GUS-16 core can only get a clock cycle if neither the video controller nor the DMA are active. The video controller steals 2 cycles every 16 during the visible part of video lines while the DMA only steals a cycle every 1536.

- The SPI controller now can transfer 16-bit values, but it also allows to transfer 8-bit bytes as in the original Floppyton. The data lenght is selected by means of the writing to two different addresses: one for 16-bit transfers and other for 8-bit. The data retrieved is always 16-bit wide, but it will have its 8 MSB bits as 0 if the transfer was 8-bit wide.

- The DMA shift register and its read and write buffers are now 16-bit wide and DMA reads or writes are requested after the transfer of 16 floppy bits instead of 8.

- The video controller is the same as in the Floppyton and it reads bytes (16-bit words aren't very useful for a text mode video controller). The least significant bit of the video address selects between the high 8 bit of the memory data when zero, or the lower 8 bits when one.

- The code size was a constant worry because the available memory is small and the GUS16 processor uses 16 bits for each op-code (one instruction, LDPC, uses 32 effective bits). The Z80 have variable length op-codes, many of them only 8-bit wide, so, I was expecting a quite bigger code for the GUS16 case. But, as it turned out, both codes have more or less the same size. Doing some statistics the average instruction sizes were:

Z80	GUS16-V5
16.18 bits	16.12 bits

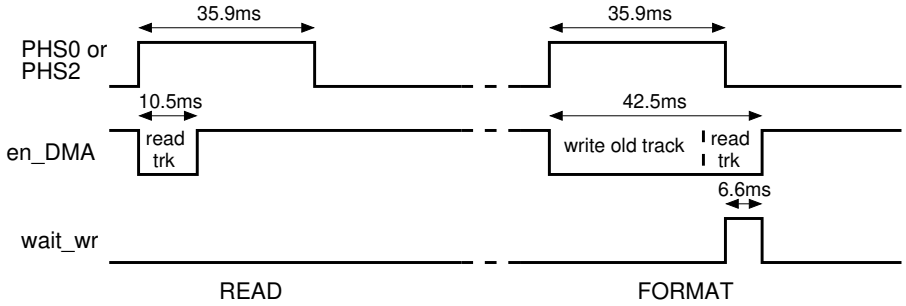
The Z80 code makes an extensive use of the IX and IY registers, requiring instruction prefixes and resulting in a bigger code. Also, the GUS16 is able to uses less instructions than the Z80 in many situations, specially when dealing with 16 or 32 bit arithmetic.

6.1 GUS16 port results

It took quite a lot of work to rewrite the Floppyton code into a GUS16 equivalent, but I finally got it working and the effort paid off mainly in the logic cell count of the synthesized computer:

System	GUS16	Z80
//e, no floppies	1356	
//e + Floppyton	2641	3931
//e + Floppyton + Softcard	4783	6118

As we can see the GUS16 version of the Floppyton requires a little less space than the Apple IIe, while the original Floppyton with a Z80 almost multiplied the system size by 3. And not only the cell count is way lower when using the GUS16, the time required for the synthesis is also substantially less.



And what about performance? Well, I attached the logic analyzer again and recorded the times of the previous figure. Here we can see the GUS16 version of the Floppyton reads a whole track in about 10ms. This is twice as fast as in the original Floppyton with the Z80. Writes also take less time, but they aren't yet fast enough to avoid stopping the Apple CPU during format commands. The speedup can be attributed to both the faster CPU, that executes one instruction every clock cycle, and also to the 16 bit mode of the SPI controller.

And with respect to the code size, these are the values for the used memory, including the code and tables:

Floppyton	Size	Size in bytes
Z80	3522	
GUS16	1789 words	3708

The GUS16 ends up using a 5% more memory due mainly to the byte handling routines that aren't present in the Z80 case. The average code size for a given function is usually larger in the GUS16 code, but, if there is some 16 bit or 32 bit arithmetic involved, it can be noticeably shorter, so, at the end the memory usage is surprisingly similar in both cases.

In summary, after a lot of work the new Floppyton does the same as the older. Not a big deal... But it does it faster, using half of the logic, and without demanding more memory. So, the GUS16 proved to be a

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags				operation
0	0	0	0	0	RD			RA			RB			0	0	ADD	C	V	N	Z	RD= RA + RB
0	0	0	0	0	RD			RA			RB			0	1	SUB	C	V	N	Z	RD= RA – RB
0	0	0	0	0	RD			RA			RB			1	0	ADC	C	V	N	Z	RD= RA + RB+Cflag
0	0	0	0	0	RD			RA			RB			1	1	SBC	C	V	N	Z	RD= RA– RB –~cflag
0	0	0	0	1	RD			RA			RB			0	0	AND	–	–	N	Z	RD= RA & RB
0	0	0	0	1	RD			RA			RB			0	1	OR	–	–	N	Z	RD= RA | RB
0	0	0	0	1	RD			RA			RB			1	0	XOR	–	–	N	Z	RD= RA ^ RB
0	0	0	0	1	RD			RA			RB			1	1	BIC	–	–	N	Z	RD= RA & (~RB)
0	0	0	1	0	RD			ulit8								ADDI	C	V	N	Z	RD= RD + ulit8
0	0	0	1	1	RD			ulit8								SUBI	C	V	N	Z	RD= RD – ulit8
0	0	1	0	0	RD			ulit8								ADCI	C	V	N	Z	RD= RD + ulit8 +Cflag
0	0	1	0	1	RD			ulit8								SBCI	C	V	N	Z	RD= RD – ulit8 –~Cflag
0	0	1	1	0	RD			ulit8								ANDI	–	–	N	Z	RD= RD & ulit8
0	0	1	1	1	RD			ulit8								ORI	–	–	N	Z	RD= RD | ulit8
0	1	0	0	0	RD			ulit8								XORI	–	–	N	Z	RD= RD ^ ulit8
0	1	0	0	1	RD			ulit8								CMPI	C	V	N	Z	RD – ulit8
0	1	0	1	0	RD			ulit8								LDI	–	–	–	–	RD= ulit8
0	1	0	1	1	RD			0	ulit4h		RB			ulit4l		RORI	–	–	N	Z	RD = (RB>>uli4) | (RB<<16–ulit4)
0	1	0	1	1	RD			1	0	0	RB			0	0	RORC	C	?	N	Z	RD = {Cflag,RB>>1}, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			0	1	SHR	C	?	N	Z	RD = RB>>1, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			1	0	SHRA	C	?	N	Z	RD = RB>>1 (signed) , Cflag=RB0
0	1	0	1	1	RD			1	0	1	RB			0	0	NOT	–	–	N	Z	RD = ~RB
0	1	0	1	1	RD			1	0	1	RB			0	1	NEG	C	V	N	Z	RD = –RB
0	1	0	1	1	RD			1	1	1	–	–	–	0	0	LDPC	–	–	–	–	RD = Mem(PC++)
0	1	0	1	1	–	–	–	1	1	1	RB			1	0	JIND	–	–	–	–	PC = RB
0	1	0	1	1	–	–	–	1	1	1	–	–	–	1	1	RETI	–	–	–	–	return from interrupt
0	1	1	0	0	RD			RA			udisp5					LD	–	–	N	Z	RD= Mem(RA+udisp5)
0	1	1	0	1	RB			RA			udisp5					ST	–	–	–	–	Mem(RA+udisp5)=RB
0	1	1	1	sdisp12												JAL	–	–	–	–	PC = PC+sdisp12, Rlink=PC
1	0	0	0	sdisp12												JZ	–	–	–	–	PC = PC+sdisp12 if Zflag
1	0	0	1	sdisp12												JNZ	–	–	–	–	PC = PC+sdisp12 if ~Zflag
1	0	1	0	sdisp12												JC	–	–	–	–	PC = PC+sdisp12 if Cflag
1	0	1	1	sdisp12												JNC	–	–	–	–	PC = PC+sdisp12 if ~Cflag
1	1	0	0	sdisp12												JMI	–	–	–	–	PC = PC+sdisp12 if Nflag
1	1	0	1	sdisp12												JPL	–	–	–	–	PC = PC+sdisp12 if ~Nflag
1	1	1	0	sdisp12												JV	–	–	–	–	PC = PC+sdisp12 if Vflag
1	1	1	1	sdisp12												JR	–	–	–	–	PC = PC+sdisp12

The GUS16-V6 instruction set

very worthy replacement. In the above snapshot it's file selection window is shown as the already loaded game (Lode Runner) runs in the background. This was the first really serious application for the GUS16 core and, with 2400 lines of code it was complex enough to expose the strengths and weakness of its design. And talking about weakness, there are a few of these worth mentioning:

- It would be very desirable to be able to read and write bytes to memory in addition to 16 bit words. The lack of LDB and STB instructions resulted in the use of slow and complex emulation routines. The inclusion of these instructions into the set will require some routing logic for the high and low halves of data buses and also the presence of byte-select signals for the memory (the memory has to be able to write single bytes). And, of course, some way to fit the new op-codes into the instruction set. One important aspect of this possible change is the meaning of address lines: The least significant bit would be selecting odd or even bytes, not words, so, the total addressable memory would be 32768 words, half the current memory space. The PC will lack its lower bit, jump displacements will have to be multiplied by 2, unaligned word reads will probably swap the bytes, and so on... ¡What a mess! I'll probably keep the address space as it is and include some kind of SWAP instruction to ease the handling of bytes instead...

- The call to a subroutine involves two instructions: First, the returning address is stored in a register (always R6), and then a jump follows. It would be desirable to merge these two instructions into a single one, like the RiscV's "JAL" or the ARM's "BL". In the datapath of the processor this would result in an extra 16 bit multiplexer for routing the PC to the register bank input and a few gates, not too much. How to encode this instruction without rearranging the whole instruction set is another question.

- Some instructions weren't used at all in the code. These were: NOT, NEG, SHRA, and JV. I don't think SHRA and JV are useless, these could be really needed for other programs, but the complement instructions are good candidates for removal because their functions can also be accomplished using XOR and SUB.

- Three operand instructions very often have the same source and destination register, so, it's tempting to reduce the op-codes to only 2 operands. This would allow wider literal values at the expense of more data moves between registers in the programs.

6.2 GUS16-V6

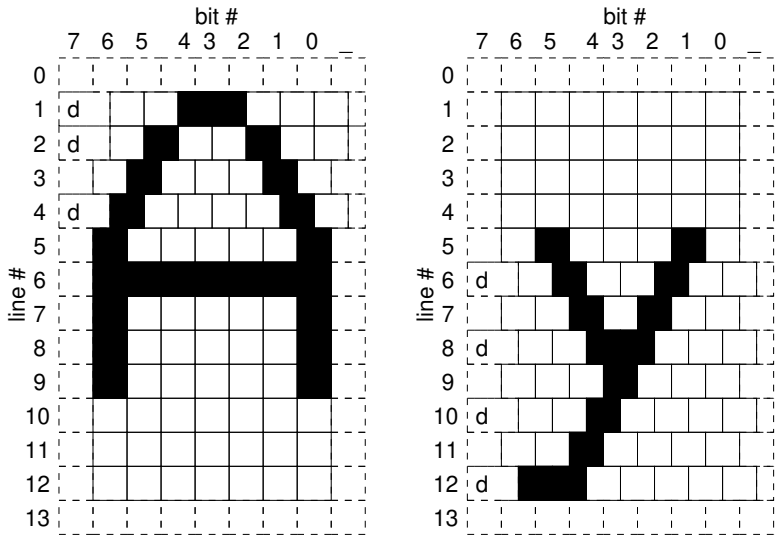
The GUS16 instruction set was changed after the experience gained while programming the Floppyton firmware. The new instruction set is shown in the preceding table.

There, we must remark two new instructions: JAL, a jump that also copies the value of PC into R6 and allows subroutine calls with a single instruction, and RORI, a multi bit rotation that eases the handling of bytes and mismatched endianness words without resorting to external data swappers. Another important change are the instructions with immediate operands that now have the same source and destination register and 8-bit literals. The new encoding allowed less instructions than in the V5 core, so, a few instructions are no longer present, like CMP, TST, TSTI, and ADPC. That last instruction was used for subroutine calls and is now replaced with JAL. On the other hand, the lack of CMP and TSTI was a little annoying. These instructions were replaced by SUB and AND / ANDI respectively.

With the new V6 core the size of the Floppyton firmware was reduced to 1703 words, 150 words less than before, and resulted in even less memory than for the original Z80 version (34 bytes less). Subtracting the size of data tables and strings, the V6 core shows a 12% reduction in the size of the actual code.

The V6 Floppyton required 53 logic cells more for its synthesis, a quite small increment.

6.3 HP-150 text font



Another cosmetic improvement was the change of the text font in the Floppyton window. When I knew about the HP-150 computer and the way its gorgeous text font was rendered I had to do the same. In that computer each character is 7x12 pixels wide but lays inside a 9x14 pixel array. And not only that, the most significant bit of the character ROM data has the interesting effect of delaying half a pixel the rest of the character line when one. This trick sounds familiar, it is the same thing Woz did in Hires mode, and its effect on the text is equivalent to doubling the horizontal resolution, but without increasing the size of the ROM for character lines. Well, in fact the character table in the Floppyton was increased a 50% because now a character has 12 lines instead of 8. In the HP-150 each character needs 16 bytes inside the ROM, but I'm going to use only 12 bytes in order to avoid wasting more memory than really needed, even if this results in some complications in the generation of the character row address, like multiplications by 12 and additions.

The horizontal and vertical counters now are divided into two sections each:

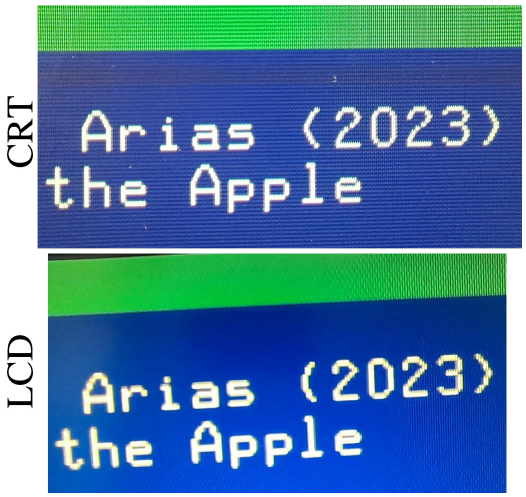
- A modulus-9 pixel counter, plus a 6-bit horizontal character counter that get preset to #40 by the "hsyn" input and stops counting when it reaches the value #32. The horizontal blanking signal is its most significant bit.

- A modulus-14 line counter, plus a 5-bit vertical character counter that gets preset to #29 by the "vsyn" input and stops counting when it reaches the value #24. The vertical blanking signal is the logic AND of its two most significant bits. The line counter actually counts from #15 to #12, and it signals a non-visible character line if its two most significant bits are one, thus disabling the loading of the pixel shift register. (but the "char" register is loaded with the character ASCII code and its attributes: transparent and remark)

The HP-150 text display also includes an analog way to stretch the pixels horizontally (an open-collector gate with a loading capacitor and a pull-up resistor). This makes the on-pixels about a 60% wider than nominal, and was intended as a way to correct the rectangular aspect ratio of the CRT pixels. Here, the VGA screen has square pixels and I don't think this trick is really needed.

I must confess (Nobody expects the Spanish Inquisition!) the results were a little disappointing, with many characters being a bit distorted. But the problem was related with my LCD monitor. When I attached an old CRT screen to the board the text was as gorgeous as expected (images made with electron beams ;) :

The problem with LCD monitors is they are sampling the analog video signal and don't expect more than 640 pixels on a line. But, with half-pixel delays you would need to sample the line 1280 times or some details could be lost, just what happens with my LCD display. Look at the "A"s, or "0"s, in the following comparison:



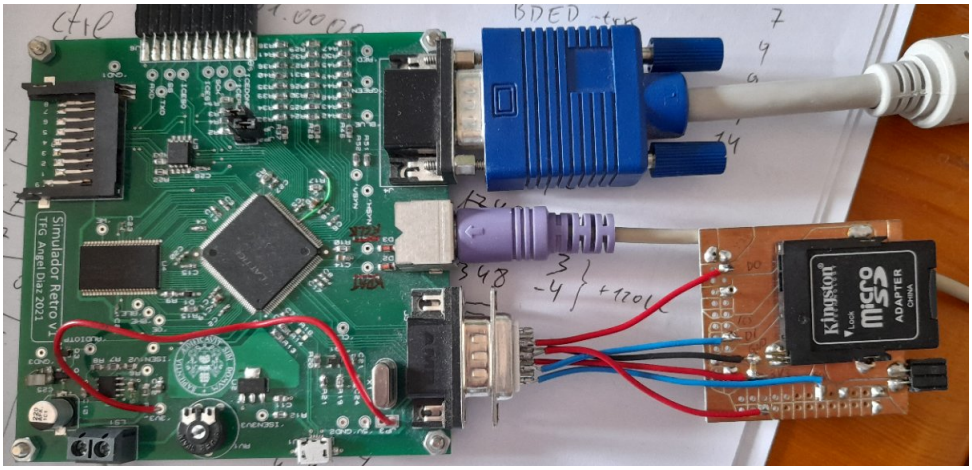
But, anyway, on LCD screens the text is still very readable and I had enough memory for it, so, I think this last mod is going to stay.

7 The //e with 128K

Probably the last improvement of this recreation, the 128KB //e required some big modifications. The most obvious was the use of the board SRAM only for the Apple II RAM because it has just the needed size, 128KB. And, what do we do with the 16KB of ROM? The internal BRAM is not enough: we have just 16KB and some blocks are already in use for the character ROM and other functions, And I also want to keep the Floppyton, with its 12KB of memory. The only solution is to map the Apple ROM into the SPI flash directly. The SPI flash is slow, it takes no less than 40 clock cycles to read a byte in a random location: 8 cycles for the command, 24 cycles for the address, and other 8 cycles for the data. But our Flash chip can run with a 54MHz clock

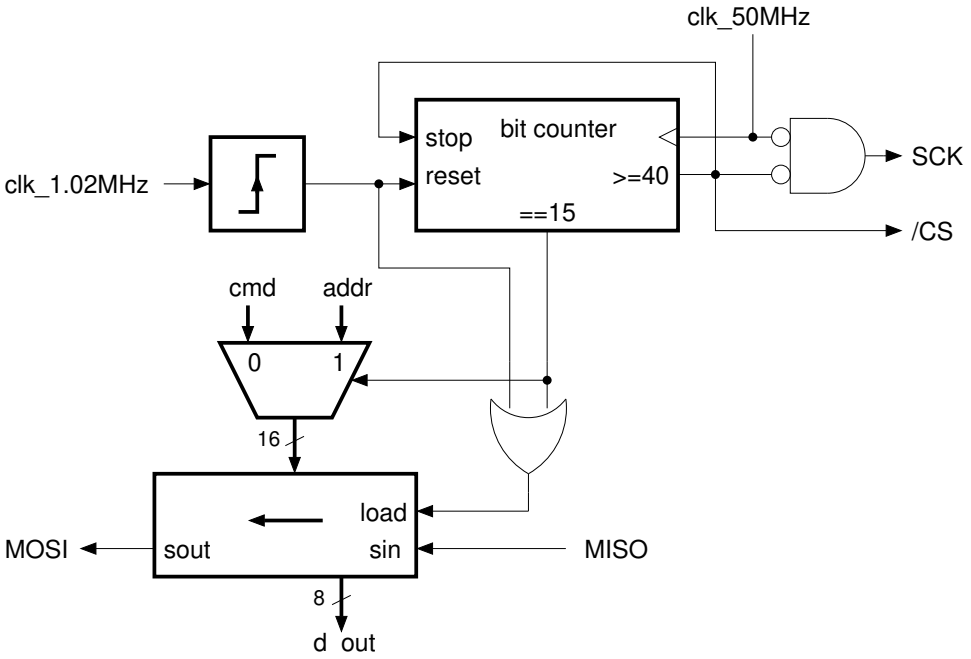
frequency while the Apple CPU runs at only 1.020MHz, so, it is still possible to do the reading if a fast clock is provided for the SPI bus.

But there is still another problem here: the on board SPI bus is shared between the SPI flash and the SD card, and if the ROM is accessed through the Flash this SPI bus is going to be very busy and the SD card will simply be unreadable. The simplest solution was to use a separate SPI bus for the SD card, and this was possible thanks to a DB9 connector that was initially intended for Atari-style joysticks, but that also included two power pins. So, I only had to attach an SD card connector to that DB9 and to route the used pins to the Floppyton logic in the Verilog sources.



In the new design there are no longer any boot ROM nor boot register, and the computer boots from the Apple ROM directly. The PLL frequency was changed from 25MHz to 49MHz, and one 6502 cycle takes 48 fast clock cycles. This results in a speed that matches the original Apple II almost perfectly (1.0208MHz vs 1.0205MHz), but beware: some CPU cycles can be shorter or longer due to video memory arbitration, ranging from 44 cycles to 52, but we still have more than 40 cycles for the worst case. The VGA video is expecting a 50MHz clock, but this 2% deviation lies still into the syncing range of monitors and the image remains perfect.

And, of course, an SPI flash reader has to be included in order to emulate the system's ROM. Its diagram is shown in the next figure:



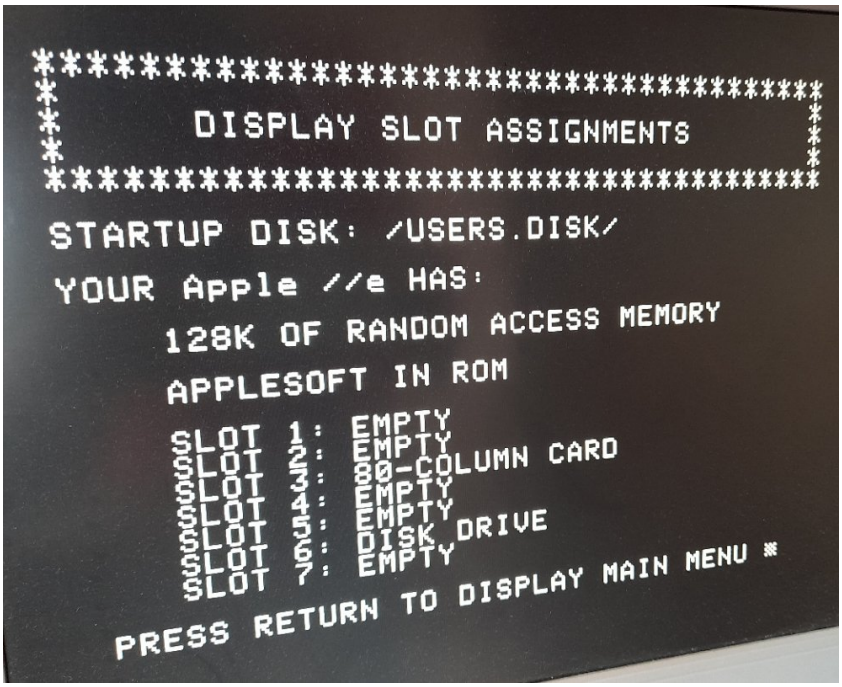
The reader uses mainly a 16-bit shift register and a bit counter. A rising edge in the CPU clock triggers a reading and loads the shift register with the read command and the fixed higher 8 bits of the flash address (0x037E). Also, the bit counter is reset, asserting /CS, and starting the generation of SCK pulses.

The shift register is loaded again when the bit counter reaches the value #15, in this case with the lower 16 bits of the address (the 14 LSBs are the emulated ROM address). In this way, the 6502 have a quite long time (326ns) for settling the memory address. The shift register isn't loaded again, but it keeps shifting the input data into its lower bits, and after 40 cycles its bits zero to seven holds the data returned by the Flash. At that time the bit counter stops, deasserts /CS, and no more pulses are generated in SCK.

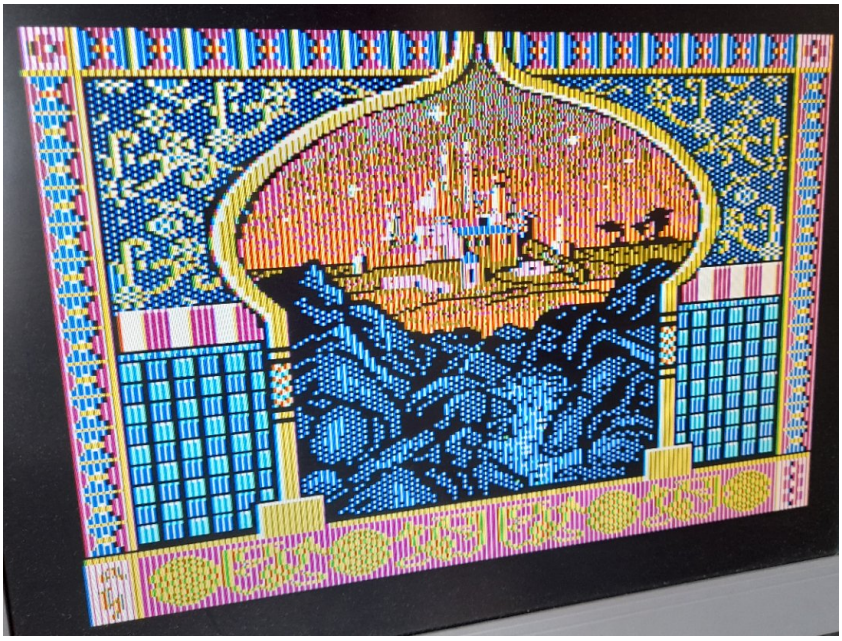
Notice that MISO is sampled on the falling edges of SCK. At this clock frequency reading the input bits on the rising edges resulted in unreliable data, probably due to the external SCK signal being delayed with respect to its internal equivalent and not meeting the setup time requirements.

With this approach we can now boot the Apple IIe directly, but only after fixing a few glitches. It is absolutely necessary to avoid CPU cycles shorter than 40 fast clock cycles, and I found the hard way that these short cycles were present during reset until I passed my reset signal through a flip-flop clocked by the 1.02MHz clock. And in this respect the Softcard is also another troublemaker because the Z80 runs with a 2.04MHz clock, and we haven't enough time for the ROM reading. But, a ROM filled with 6502 code isn't very usable by a Z80, anyway, so I resorted to disable it completely when the Z80 is selected, and CPM keept running as fine as before. It seems that CPM only uses RAM.

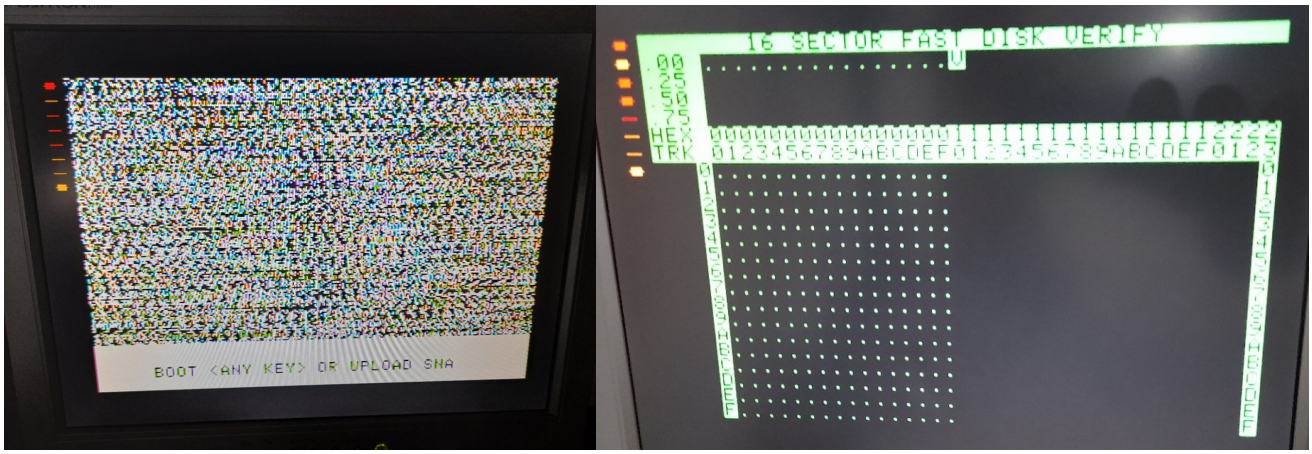
And now, we have a 128-KB Apple //e running, as we can see in this Prodos snap:



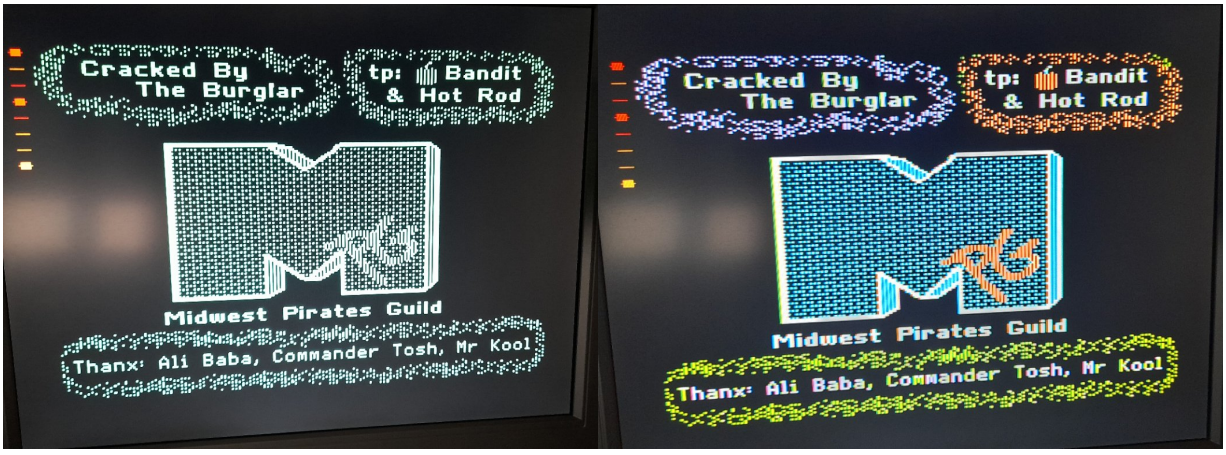
And, of course, it can play "Prince of Persia" with its double-Hires init screen, that BTW, it also allowed to detect a color bug in double-Hires mode:



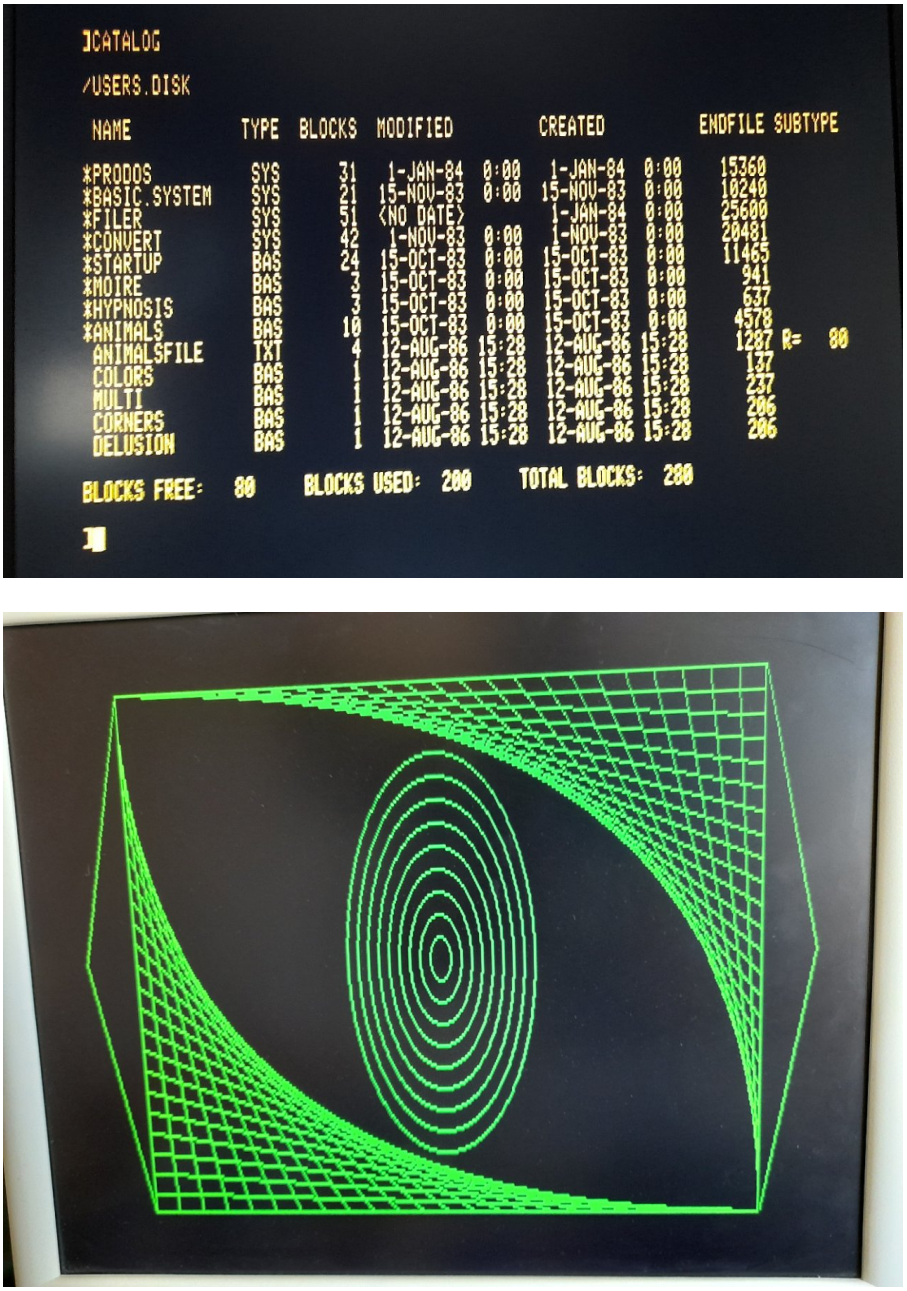
8 Results, conclusions, more improvements



The recreated computer can boot from raw floppy images stored first in the SPI flash, and then in an SD card, and the list of working programs include, among others, Locksmith 6.0, Lode Runner, Aztec, Eliminator, Ms Pacman,... In the above snapshots we show the boot screen (left) and the Locksmith utilities verifiying a floppy disk. On both cases the virtual LEDs are visible on the upper left (old style).



In the above snapshots the same image was displayed in monochrome green and in color mode, also with the virtual LEDs still visible.



And finally, the Apple IIe recreation is also shown in the previous snapshots, where an 80 column screen with a Prodos directory is displayed on the upper image with a monochrome amber monitor emulation, and also a double hi-res monochrome screen on the bottom image. This last figure was in fact computed by the Softcard's Z80 running an sdcc-compiled ad-hoc program. Notice that in this mode the aspect ratio of pixels is quite bad, with almost twice the size in the vertical axis than in the horizontal.

These computers were quite simple, yet they required many hours of work, most of then for debug. Like when, after days of getting stuck with the floppy controller, thinking about spurious read errors and changing details of its design many times, I finally decided to read back the disk image from the SPI flash, and it differed from the original! The controller was perfectly fine, the flash image not, it got corrupted due to the presence of an SD card that shouldn't be inserted during flash programming because it can contend with the data on the SPI bus. After this discovery about human stupidity everything started to run fine until I tripped again on the same stone when I downloaded a new Apple II emulator and found the colors it displayed on the screen were different than those I had so much trouble to reproduce in the replica. The old emulator that I chose as a reference had in fact a bug regarding NTSC colors, and I stubbornly tried to reproduce it even when the color decoding hardware was pointing to the correct colors.

And talking about improvements, many extensions of the basic computer are not yet supported, like the

Super serial card and several others. And I don't think they are going to be included soon.

And, of course, an FPGA board with more RAM would be a very desirable hardware to have because there are several vintage computers with 128KB of RAM that simply doesn't fit in the current board.

The basic Apple II design fits in only 1424 logic cells plus 5 BRAMs, mainly thanks to the 6502 core being quite small when compared to other CPUs of its time. And with a total of 7680 logic cells and 32 BRAMs we still have plenty of FPGA space for improvements. Even with the Z80 card recreation almost half the FPGA space is still unused. The Apple IIe only requires a few logic cells more. A simplified Apple IIe with monochrome video, no PS2 keyboard, no funny sounds, and only one floppy disk was made to fit into a 1280 logic cell FPGA. On the other hand, a computer with the Z80 Softcard and the Floppyton will require no less than 5948 logic cells and 30 BRAMs.

Recreating ancient computers is basically a waste of time comparable to putting little boats inside bottles. But it can be a very rewarding time, and this certainly was. Not forgetting to mention that now I finally understand how magnetic recording works in deep detail, and you got to know technology well if you want to become a Morlock.

Best regards.