by Stephen A. Edwards

Retrocomputing on an FPGA

Reconstructing an 80's-Era Home Computer with Programmable Logic

The author reconstructs a computer of his childhood, an Apple II+.

As a Christmas present to myself in 2007, I implemented an 1980s-era Apple II+ in VHDL to run on an Altera DE2 FPGA board. The point, aside from entertainment, was to illustrate the power (or rather, low power) of modern FPGAs. Put another way, what made Steve Jobs his first million could be a class project for the embedded systems class I teach at Columbia University.

More seriously, this project demonstrates how legacy digital electronics can be preserved and integrated with modern systems. While I didn't have an Apple II+ playing an important role in a system, many embedded systems last far longer than their technology. The space shuttle immediately comes to mind; PDP-8s can be found running some signs for San Francisco's BART system.

What is an Apple II+?

The Apple II+ (Photo 1) was one of the first really successful personal computers. Designed by Steve Wozniak ("Woz") and introduced in 1977 [1, 2, 4], it really took off in 1978 when the 140K Disk II 5.25-inch floppy drive was introduced, followed by VisiCalc, the first spreadsheet.

Fairly simple even by the standards of the day, the Apple II was built around the inexpensive 8-bit 6502 processor from MOS Technology (it sold for $25 when an Intel 8080 sold for $179). The 6502 had an eight-bit data bus and a 64K address space. In the Apple II+, the 6502 ran at slightly above 1 MHz. Aside from the ROMs and DRAMs, the rest of the circuitry consisted of discrete LS TTL chips (Photo 2).

Photo 1: An Apple II+

While the first Apple IIs shipped with 4K of DRAM, this quickly grew to a standard of 48K. DRAMs, at this time, were cutting-edge technology. While they required periodic refresh and three power supplies, their six-times higher density made them worthwhile.

Along with with an integrated keyboard, a rudimentary (one-bit) sound port, and a game port that could sense buttons and potentiometers (e.g., in a joystick), the main feature of an Apple II+ was its integrated video display. It generated composite (baseband) NTSC video that was usually sent through an RF modulator to appear on TV channel 3 or 4.

The Apple II+ had three video modes: a 40×24 uppercase-only black-and-white text display, a 40×48 16-color low-resolution display, and a 140×280 6-color high-resolution display. The Apple II+ can almost be thought of as a video controller that happens to have a microprocessor connected to it. Woz started with a 14.31818 MHz master clock—exactly four times the 3.579545 MHz colorburst frequency used in NTSC video—and derived everything from it.

The CPU and video alternate accesses to memory at 2 MHz. Another Woz trick: the video addresses are such that refreshing the video also suffices to refresh the DRAMs, so no additional refresh cycles are needed.

Figure 1 shows the block diagram of my reconstruction. The 6502 processor on the left generates addresses and output data. The address is fed to the ROMs, an

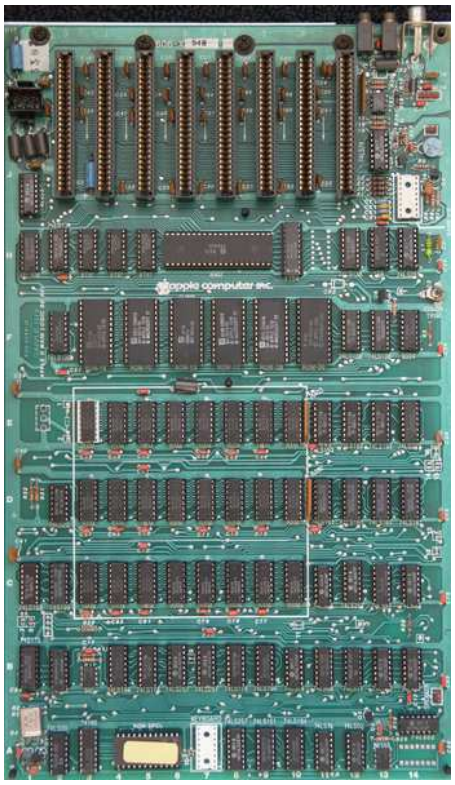

Photo 2: The Apple II+ Motherboard. Expansion slots and analog video circuity dominate the top; the 6502 is above the six large ROM chips. The white rectangle encloses 48K of DRAM. The character ROM is at the bottom; the rest is TTL.

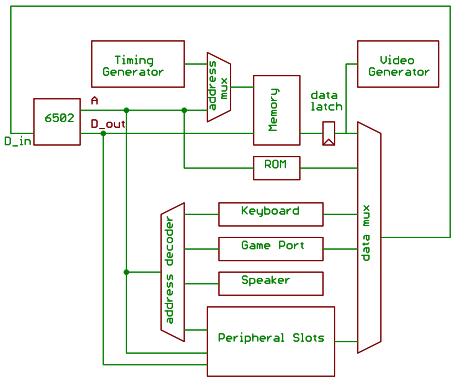

Figure 1: Block Diagram

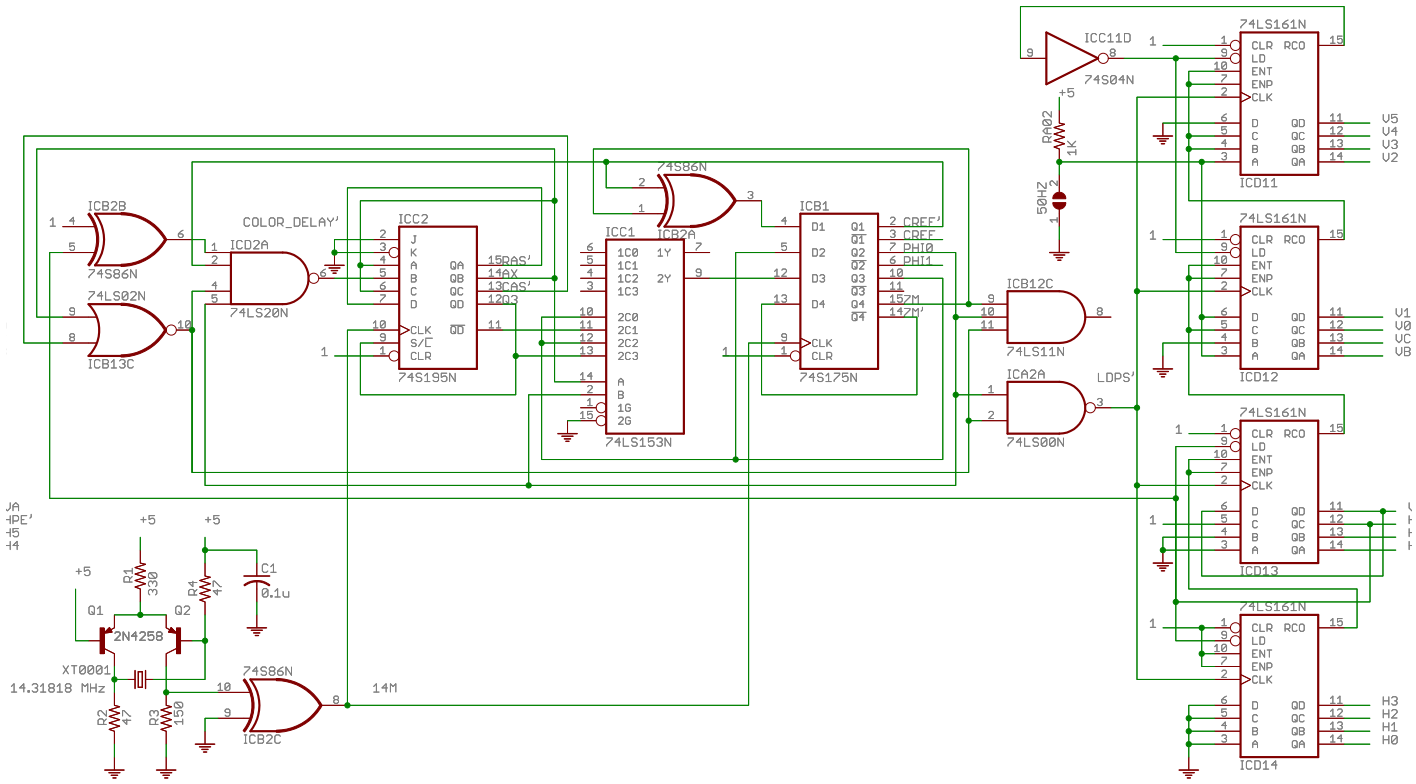

Figure 2: Woz's clock generator circuit. A 14.31818 MHz crystal drives a 4-bit shift register and a quad flip-flop to generate DRAM timing signals and the processor clocks, which in turn feed a bank of horizontal and vertical video counters.

address range decoder, the peripheral slots, and to a mux that selects between processor and video system addresses for the main memory.

The original Apple II+ used a tri-state data bus, but FPGA cores do not support such complex electrical structures (although they do provide tri-state I/O pins), so my reconstruction breaks the data bus into multiple segments. Most notably, I added a large mux (right side of Figure 1) that selects the source of data fed to the 6502 core, such as main memory or the ROMS.

The Clock Generator

Figure 2 shows the Apple's clock generator circuit. A crystal oscillator drives the clocks on a '195 quad shift register and a '175 quad flip-flop. These generate clocks for the the DRAM (RAS' and CAS') along with the "1 MHz" processor clocks PHI0 and PHI1. A gated version of PHI0 feeds a bank of '161s: four-bit binary counters configured to act as horizontal and vertical counters (H0–H5, VA–VC, and V0–V5) from which the video addresses are generated.

This clever circuit does a lot with few parts. It is at the center of Woz's patent [5], which describes it and his trick of using digital signals to generate color NTSC video.

Woz derived the CPU clock from the 14M clock by dividing by roughly fourteen. "Roughly" because every sixty-fifth CPU cycle (one per horizontal scan line) is stretched by two 14M clock periods to preserve the phase of the 3.58 MHz colorburst frequency. Thus, there are 65 * 14 + 2 = 912 pixel periods per line, or exactly 228 cycles of the 3.58 MHz colorburst per line.

While it would be possible to write a model for each TTL part in VHDL and assemble them according to the schematic, I prefer to try to write the VHDL according to Woz's intentions for the original circuit. This is especially true for combinational "glue" logic, which was often implemented in nonintuitive ways to save parts.

Listing 1 shows my VHDL code for the clock generator. It assumes the 14 MHz clock is provided externally and consists of three main sequential processes. The first models the '195 shift register, which either shifts or loads dependings on its own Q3 output. The second process models the '175 quad flip-flop and the '153 driving it, which selects between PRE_PHI_0 and a combination of Q3 and PHI0 depending on the state of AX.

The third sequential process models the four 4-bit binary counters. In the original circuit, these were clocked by the output of a NAND gate. Such a practice is dangerous because the output of the gate might glitch and cause unpredictable behavior, so instead I chose to clock these counters at 14 MHz and instead carefully control when they count.

Figure 3 shows a timing diagram for the clock generator illustrating how it behaves at the end of a line. The COLOR_DELAY_N signal causes the shift register to delay RAS_N et al. two extra 14M cycles, which also causes PHI0 to be stretched. HCOUNT changes on the rising edge of LDPS_N, just as in the original circuit.

The values taken on by the horizontal counter are a little unusual: the counter is allowed to wrap around from 7F to 00, but is then set to 40 to start the line. These 65 PHI0 periods turn into about 15.70 kHz, close to the NTSC horizontal frequency of 15.734 kHz.

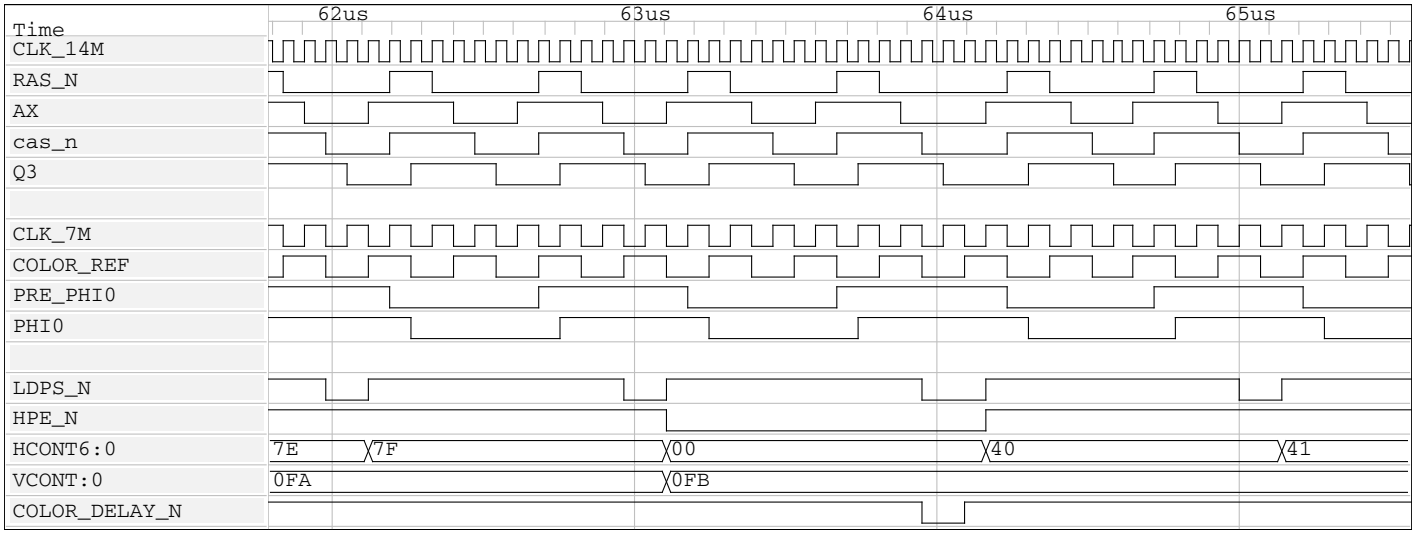

Figure 3: Behavior of the clock generator at the end of a line

The CPU and Memory

Like Woz, I didn't create a 6502 processor from scratch. Instead, I used a 6502 core written by Peter Wendrich for his FPGA-based Commodore 64. The main challenge here was making sure it was clocked properly given the odd way the Apple II+ generates its occasionally stretched processor clock.

Semiconductor memory has changed a lot since 1977. The Apple II+ used 24 4116 16-kilobit DRAM chips with 150 ns access times to provide 48 kilobytes of memory. Today, it is difficult to find memory chips this small.

While it would have been nice to place all of the Apple's memory on the FPGA I was using, it (an Altera Cyclone II 2C35) has about 59K of on-chip RAM, which is just a little too small to fit 48K of RAM plus 12K of ROMs. I chose instead to use off-chip SRAM (the DE2 has 512K) for the 48K of main memory and store the ROMs on-chip. Storing the ROMs in FPGA memory is more convenient because their contents are initialized when the FPGA is programmed.

Asynchronous SRAM is much easier to interface than DRAM. The only real issue is generating an approriately timed write enable signal and making sure the tri-state data pins are only driven when the processor is writing to the RAM.

The Video Generator

The Apple II+ has three main video modes: a 40×24 uppercase-only text display, a 40×48 16-color "lores" graphics mode, and a 280×192 6-color

```
-- To generate the once-a-line hiccup: D1 pin 6
COLOR_DELAY_N <=
  not (not COLOR_REF and (not AX and not CAS_N) and PHI0 and not H(6));

-- The DRAM signal generator
C2_74S195: process (CLK_14M)
begin
  if rising_edge(CLK_14M) then
    if Q3 = '1' then -- shift
      (Q3, CAS_N, AX, RAS_N) <=
        unsigned'(CAS_N, AX, RAS_N, '0');
    else                    -- load
      (Q3, CAS_N, AX, RAS_N) <=
        unsigned'(RAS_N, AX, COLOR_DELAY_N, AX);
    end if;
  end if;
end process;

-- The main clock signal generator
B1_74S175 : process (CLK_14M)
begin
  if rising_edge(CLK_14M) then
    COLOR_REF <= CLK_7M xor COLOR_REF;
    CLK_7M <= not CLK_7M;
    PHI0 <= PRE_PHI0;
    if AX = '1' then
      PRE_PHI0 <= not (Q3 xor PHI0);  -- B1 pin 10
    end if;
  end if;
end process;

LDPS_N <= not (PHI0 and not AX and not CAS_N);
LD194 <= not (PHI0 and not AX and not CAS_N and not CLK_7M);

-- Four four-bit presettable binary counters
-- Seven-bit horizontal counter counts 0, 40, 41, ..., 7F (65 states)
-- Nine-bit vertical counter counts $FA .. $1FF  (262 states)
D11D12D13D14_74LS161 : process (CLK_14M)
begin
  if rising_edge(CLK_14M) then
    -- True the cycle before the rising edge of LDPS_N: emulates
    -- the effects of using LDPS_N as the clock for the video counters
    if (PHI0 and not AX and ((Q3 and RAS_N) or
                             (not Q3 and COLOR_DELAY_N))) = '1' then
      if H(6) = '0' then H <= "1000000";
      else
        H <= H + 1;
        if H = "1111111" then
          V <= V + 1;
          if V = "111111111" then V <= "011111010"; end if;
        end if;
      end if;
    end if;
  end if;

end process;
```

Listing 1: VHDL for the timing generator

"hires" graphics mode. The graphics modes also have a mixed mode in which the bottom four lines of text are displayed instead.

The memory layout for all three modes is similar and non-linear. To accommodate 40-character text lines using only a single four-bit binary adder and wasting little memory, Woz divided the screen into three horizontal stripes, each 64 scan lines high (equivalently, eight character rows). Memory for each display mode is divided into 128-byte segments that hold three 40-byte lines (i.e., the last eight bytes in each segment are not displayed). The first line in each segment appears in the top stripe, the second in the middle stripe, and the third in the bottom. The result is that bits 3 to 6 of the video address are a funny sum of horizontal and vertical counter bits.

All three modes fetch one byte from video memory every PHI0 cycle. In text mode, the data is fed to the top six address bits of the character ROM and the output of the ROM is loaded into a '166 eight-bit parallel-to-serial shift register. In lores mode, the byte is loaded into a pair of four-bit recycling shift registers and clocked out repeatedly. In hires mode, the byte is loaded into an eight-bit shift register and clocked out.

The VGA Line Doubler

The Apple II+ generates a composite color NTSC signal that was usually sent through an RF modulator and displayed on a standard television set. Since computers have not used composite color monitors since the early 1980s, one of my goals was to generate an analog color VGA signal (now also obsolete) suitable for a standard computer LCD monitor.

This presented two problems. The first is one of rate: the Apple II+ generates composite color non-interlaced NTSC video: 60 frames a second, 262 lines per frame. This leads to a horizontal refresh rate of about 15.70 kHz.

The VGA standard, which has been around since 1987, is an analog RGB component format associated with a variety of refresh rates, but the most relevant here is essentially NTSC times two: a 31 kHz horizontal sweep rate with a 60 Hz frame rate. By design, this is two VGA lines for every NTSC line.

So to display an NTSC-rate image on a VGA monitor, it is enough to display each NTSC line twice, which is convenient because it only requires buffering a line instead of a whole frame. Rather than redesign Woz's carefully crafted video circuitry, I chose to place a VGA line doubling circuit after his one-bit video output that both doubles the horizontal frequency and interprets color information.

My circuit consists of a dual-ported memory that stores two lines of the 14 MHz 1-bit video signal. At any time, the circuit is filling in one line and displaying the other; the roles of the two lines swap once every NTSC line.

The Color Decoder

Interpreting colors is the bigger challenge in converting the Apple II+ output to color VGA signals. Unlike VGA, which conveys separate red, green, and blue signals, composite (color) NTSC video consists of three signals modulated together. To a high-bandwidth luminance (brightness only) signal (about 3 MHz) called Y, NTSC adds two lower-bandwidth color signals ("I" and "Q") that are quadrature modulated at 3.579545 MHz. A color television demodulates and combines linear ratios of these signals to recover red, green, and blue intensities.

The Apple II+ uses a trick to generate the modulated signal: it produces a digital signal that switches at 14.31818 MHz—exactly four times the colorburst frequency. Figure 4(a) depicts a small patch of this digital video output interpreted as black and white pixels. The sixteen different period-four waveforms (i.e., whose fundamentals are at the 3.58 MHz colorburst frequency) each produce a different color (two produce gray). All 0's is black and all 1's is white since neither has any high-frequency information; the television interprets them as purely luminance. Other patterns produce different levels of Y, I, and Q, and thus different colors.

NTSC demodulation and YIQ-to-RGB colorspace conversion is a linear process, albeit a time-varying one because quadrature modulation uses phase to distinguish two signals. So the digital video signal the Apple II+ produces can be though of as a linear combination of four square wave signals that differ only in their phase. Thus, interpreting groups of four bits as one of sixteen colors produces a reasonable display, especially for solid regions.

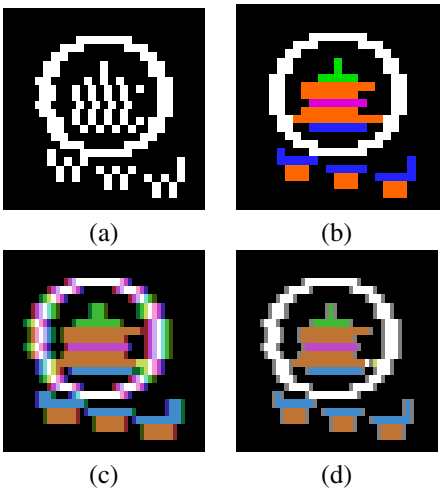

Figure 4: A hires graphics fragment interpreted as (a) monochrome, (b) output from the KEGS software emulator for the Apple IIgs, (c) under a four-bit window algorithm, and (d) under the six-bit window algorithm used in my reconstruction.

Unfortunately, this four-bit-at-a-time approach produces more color fringing around the edges of white objects than a television would because of the bandwidth limits on I and Q, as shown in Figure 4(c). My solution was to look at one bit to the left and right of the four-bit window and generate color only when these extra bits follow the same pattern as the middle four. (Figure 4(d))

Figure 5 shows an abstract view of my color generator. At the top is a six-bit shift register that amounts to a sliding window into the video signal. Each bit consumes 90 degrees of phase; the circuit mostly considers the middle four bits.

The main color circuitry comprises a "permute" block that rotates the four (constant) basis colors depending on which of the four phases a pixel can be in relative to the colorburst frequency. Then each of the four basis colors are ANDed with the four middle bits of the sliding window filter and added together to form a 24-bit RGB value.

At the top right of Figure 5 are three gates that guess when we are in the middle of a solid color region. When

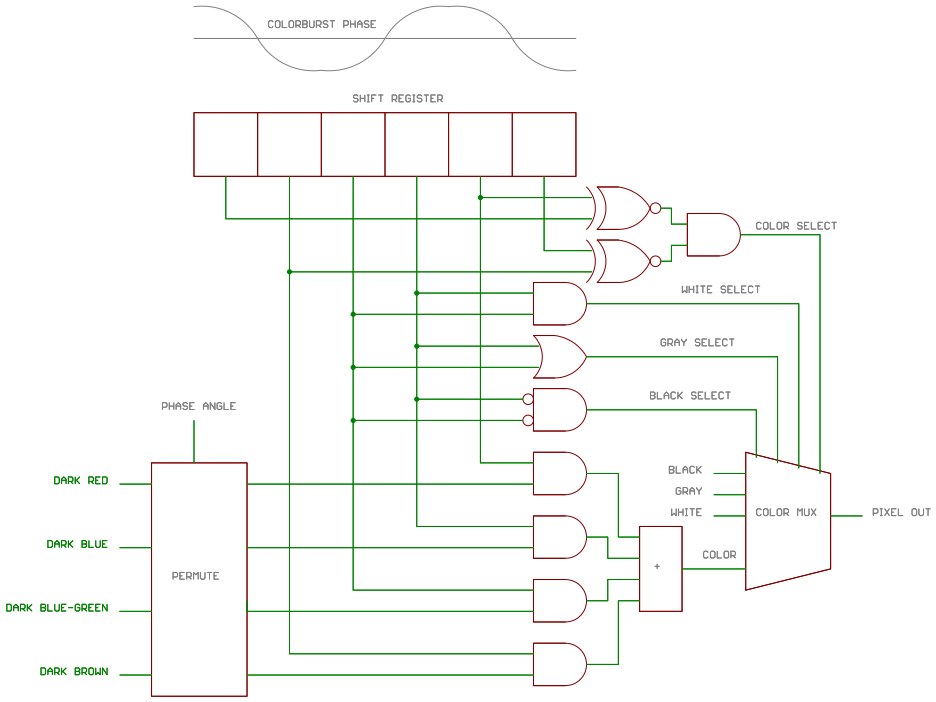

Figure 5: Abstract View of the Color Generator

bits 0 and 4 in the filter are equal, and bits 1 and 5 are also equal, the "color select" signal is true and the solid color value generated as described above is selected as the color for this pixel.

Otherwise, my circuit colors the pixel black, gray, or white depending on how many bits are set in the middle two positions in the shift register. This approximates the effect of the lower I and Q bandwidth: when the signal suddenly changes from dark to light, the luminance changes more quickly; the color information changes slower.

It took some experimentation for me to arrive at this approximation. To evaluate the algorithms, I wrote a simple C program that converted a memory dump of a hires image into a PPM file, which I then evaluated. Figure 4(d) is the output I finally implemented.

The Disk II Emulator

Introduced about a year after the Apple II itself, the Disk II 5.25" floppy disk drive was another remarkably svelte piece of hardware [2, 3]. The system consisted of a digital controller board connected to the peripheral bus, an analog board in the drive itself that handled things like controlling the stepper motor and conditioning the read signal, and a bare Shugart SA400 drive mechanism.

My goal was to make it possible for my reconstruction to boot images of 5.25" floppy disks. Years ago I converted my own collection of physical disks to such images; many more can be found on the web. Thus, my goal was to make the software think it was talking to a floppy drive instead of attempting to reconstruct the drive and its controller exactly.

The DE2 board has a SD/MMC card interface, which is just a connector with a few pins connected directly to the FPGA and some pull-up resistors. This plus the quickly falling prices of SD flash memory cards made it the natural choice.

My emulation circuit consists of two parts: a module that emulates the behavior of the Disk II controller, which interprets CPU access to the relevant I/O addresses, and a SPI module that fetches blocks of data from an SD card based on commands from the first module.

SD/MMC flash memory cards can be operated in a variety of modes. The simplest is SPI, a simple, well-documented, four-wire synchronous serial protocol. Furthermore, the wiring on the DE2 was clearly set up to operate SD cards in such a mode.

The Disk II presented an extremely low-level interface to software. Head positioning was performed by directly activating the stepper motor phases in sequence. And although the hardware did provide a facility for clock recovery and framing, the software was presented with just a raw stream of encoded bytes from the disk.

Instead of the FM scheme used by the Shugart controller, which placed a clock pulse between every data pulse, the Disk II used a group code recording scheme that allowed up to two consecutive 0's before a 1 was mandatory, making it possible to store six bits instead of four in the space of eight transitions. This improved formatted capacity to 140K per diskettee over the 90K possible with FM encoding, but it fell to the software to decode this data.

My Disk II emulator consists of an SPI controller responsible for initializing and reading data from the SD card, a bus device that interprets and responds to the 6502 like the Disk II controller, and a dual-ported RAM that holds a single unformatted track's worth of data. At 300 rpm at 4 μs per bit, this is 50,000 bits or 6250 bytes. However, the standard file format for Apple II raw disk images (".nib") uses 6656 bytes (26×256) per track, so I chose to use that.

The SA400 had a single read/write head whose position over the floppy was controlled by a stepper motor. My Disk II controller observes how the software activates the four phases of the stepper motor and responds to each track change by reading a track's worth of data into the track buffer. Once in the buffer, the controller simply cycles through the track data, emulating the movement of the head over the track.

The stepper motor has four phases, and every two phases corresponds to a distinct track (of which there are 35), but because the software is free to turn on two (or more) phases simultaneously, my controller models both when the head is at a particular phase and when it is between two adjacent phases. It constantly monitors the state of the four phases and updates the head position based on its current position. When it observes a track change, it signals the SPI controller to fetch the new track and transfer it into the track buffer.

I added a rudimentary user interface for selecting different disk images: ten

switches supply the image number in binary, which I displayed in hex on two of the seven-segment LEDs. On the SD card, the images are laid out one after the other, i.e., not in a file system. To create such a collection, I wrote a script that finds all the .dsk files in a directory, converts each to the "nibblized" format, and adds it to an image file. All 500 of the 5.25″ floppies I owned fit into 112 MB, which now resides comfortably on a $5 SD card. How times have changed.

The PS/2 Keyboard Interface

The Apple II plus had an integrated keyboard consisting of an array of discrete keyswitches scanned by a General Instruments AY-5-3600 keyboard encoder that produced a seven-bit ASCII code. When a key was pressed, it would latch the code and send a pulse that indicated a new key was pressed. The Apple II would latch the pulse as bit 7 of the keyboard I/O location and clear it when another I/O location was accessed, providing a simple handshake.

Instead of directly connecting a keyswitch array to the FPGA, I decided to employ one of the many PS/2-compatible keyboards littering my office. This was especially attractive since the DE2 board board already had a PS/2 connector.

The PS/2 keyboard interface is a simple but idiosyncratic synchronous serial protocol that sends and receives data a byte at a time. The usual message is "make," which indicates a particular key has been pressed. Other messages include "break" followed by a code for a key that has been released. Unfortunately, the scan codes are not ASCII (perhaps reflecting the wiring of an early keyboard) and use "extended codes" for keys such as the arrows, since they were not on the original keyboard.

My solution uses the free PS/2 controller distributed by ALSE, which speaks the low-level protocol and performs the serial-to-parallel conversion, and a simple state machine that looks at the returned messages and interprets them as ASCII. The code is sloppy but works. Because all of this was never part of the Apple II, I was not concerned with being faithful to the original design, or even elegant.

Sound

The Apple II+'s sound system is simultaneously humorous and amazing: a speaker connected to a Darlington transistor driven by a flip-flop configured to toggle when a particular I/O address is accessed. The amazing part is that programmers managed to drive such a trivial circuit to generate four-voice synthesized sound and even speech. Emulating the audio address decoding and flip-flop was trivial; doing something useful with the resulting signal was more of a challenge.

The DE2 board includes a Wolfson MW8731 CODEC, a CD-quality stereo audio chip capable of driving an audio amplifier, complete overkill for Apple II+ audio, but already there on the board. Using it presented two challenges: generating the appropriate set of signals to feed its serial interface and initializing its registers through an I^2C bus.

I implemented one module that generates the various square waves for the codec's clocks (a bit clock and a word or channel clock) and shifts out sixteen bits of amplitude data. The main trick here was choosing the proper divider values and sending out each bit at the right time.

The I^2C bus controller was more tricky. While I only needed to support a small part of the bus protocol, it still required three state machines: one to handle the low-level details of clock and data bit generation, one to transmit single packets, and one to prepare the proper sequence of packets to initialize the Wolfson chip's registers.

The Top Level

My reconstruction actually has two "top-level" modules. The "apple2" module contains the timing generator, video generator, processor, ROMs, address decoder, and various minor peripheral devices, i.e., all the original parts of the Apple II+. A second module is the actual top level, consisting of the "apple2" module along with the VGA line doubler, the PS/2 keyboard interface, Disk II emulator, audio components, a PLL that divides the DE2's 50 MHz clock down to about 28 MHz (i.e., not exactly the right frequency, but close enough), and connections for switches and LEDs on the DE2 board.

I brought out the CPU's PC to four of the seven-segment displays on the DE2 and the drive's current track on another two. While the PC is usually changing so fast it becomes a blur, patterns do often emerge. For example, the PC remains highly focused when the computer is waiting at the prompt. Similarly, I have found a lot of software, including the operating system when it is moving the drive head, calls the monitor's "delay" routine to slow things down.

Comparing Implementations

This project demonstrates how little power modern hardware consumes and how much more efficient it can be than software. I compared the power consumed by an actual Apple II+ with that consumed by my reconstruction as well as a software emulator running on ten-year-old x86-based Linux box. I used an inexpensive "Kill A Watt" power meter, which only claims 0.2% accuracy, but this was enough to demonstrate what was going on.

The results were dramatic. My real Apple II+ nominally consumed 22 watts, which rose to 31 watts when the disk was rotating; my FPGA reconstruction only consumed 5 watts, even with all its extra unused peripherals. The Dell Optiplex GXa (running a now-modest 233 MHz Pentium II) consumed 62 watts when running the emulation software.

Project Files

Included with all the VHDL files are project files for Altera's Quartus software, a utility program for converting the more common 140K .dsk files to the .nib files my reconstruction uses.

For copyright reasons, I did not include a copy of the Apple ROMs. They are easy to obtain from an existing computer or from the Internet. I included the script I used to convert the binary files into VHDL files that hold the same data.

But the project will function as it stands: I wrote a "fake BIOS" that clears the screen, displays some messages, then cycles through a simple pair of graphics demos. I included the 6502 assembly source, which I compiled with the xa65 cross-assembler. My "BIOS" is not able to boot any Apple disks, however.

A Slippery Slope

Like most projects, this one could continue without end. Many important features are still missing. Many Apple II games used a joystick, but I have not emulated it. The DE2 board has a USB host controller, so in theory I could use a standard USB joystick to it, but even a USB controller chip still demands a processor control it.

The disk emulation presents the most opportunities for improvement. For example, it is read-only, which is enough for running plenty of software, but there are plenty of reasons to want to write to a disk. Also, my emulator uses an SD card but does not support a filesystem. It would be much easier to manage disk images if they could be named and stored in a standard hierarchical filesystem (e.g., FAT32). It might be possible to do this with the 6502 processor, but a separate processor for managing this might also be in order. Along the same lines, my emulator could also support the more standard 140K disk images if it included logic to perform the encoding used by Apple DOS; most software emulators do this.

There are myriad peripheral cards that could also be emulated. The 16K memory expansion card would be a first step, but it would also be nice to have others that provided serial ports, printers, and improved sound.

Perhaps next Christmas I'll have time.

References

[1] Winston Gayler. *The Apple II Circuit Description*. Howard W. Sams & Co., 1983.

[2] Jim Sather. *Understanding the Apple II*. Quality Software, Reseda, CA, 1983.

[3] Don Worth and Pieter Lechner. *Beneath Apple DOS*. Quality Software, Reseda, CA, 1981.

[4] Stephen Wozniak. System description: The Apple-II. *Byte Magazine*, 2(5):34–43, May 1977.

[5] Stephen G. Wozniak. Microcomputer for use with video display. US Patent 4,136,359, January 1979.