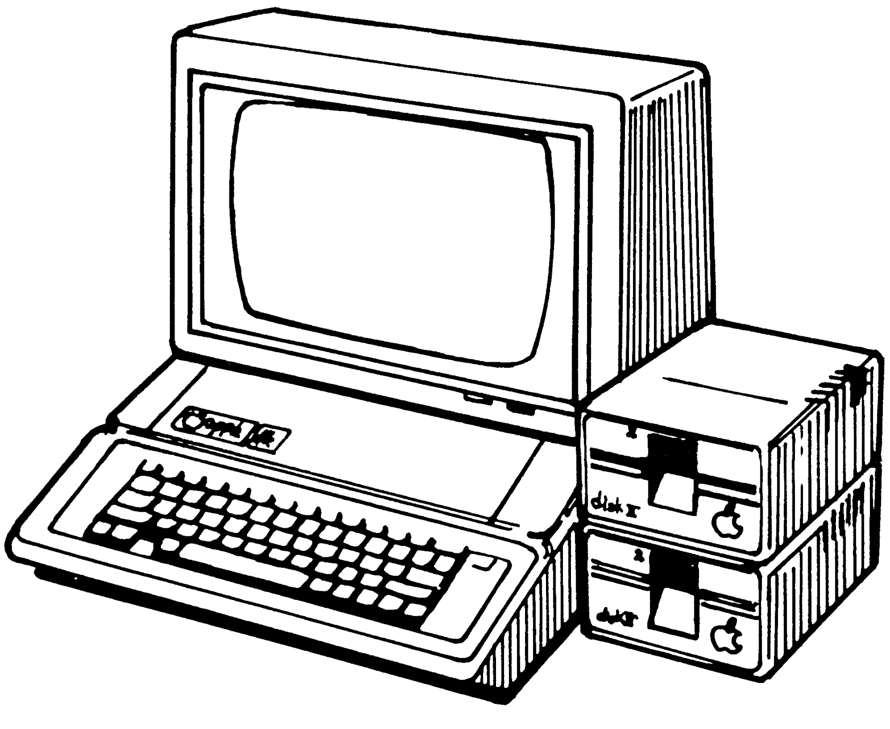
Apple][Computer System Description

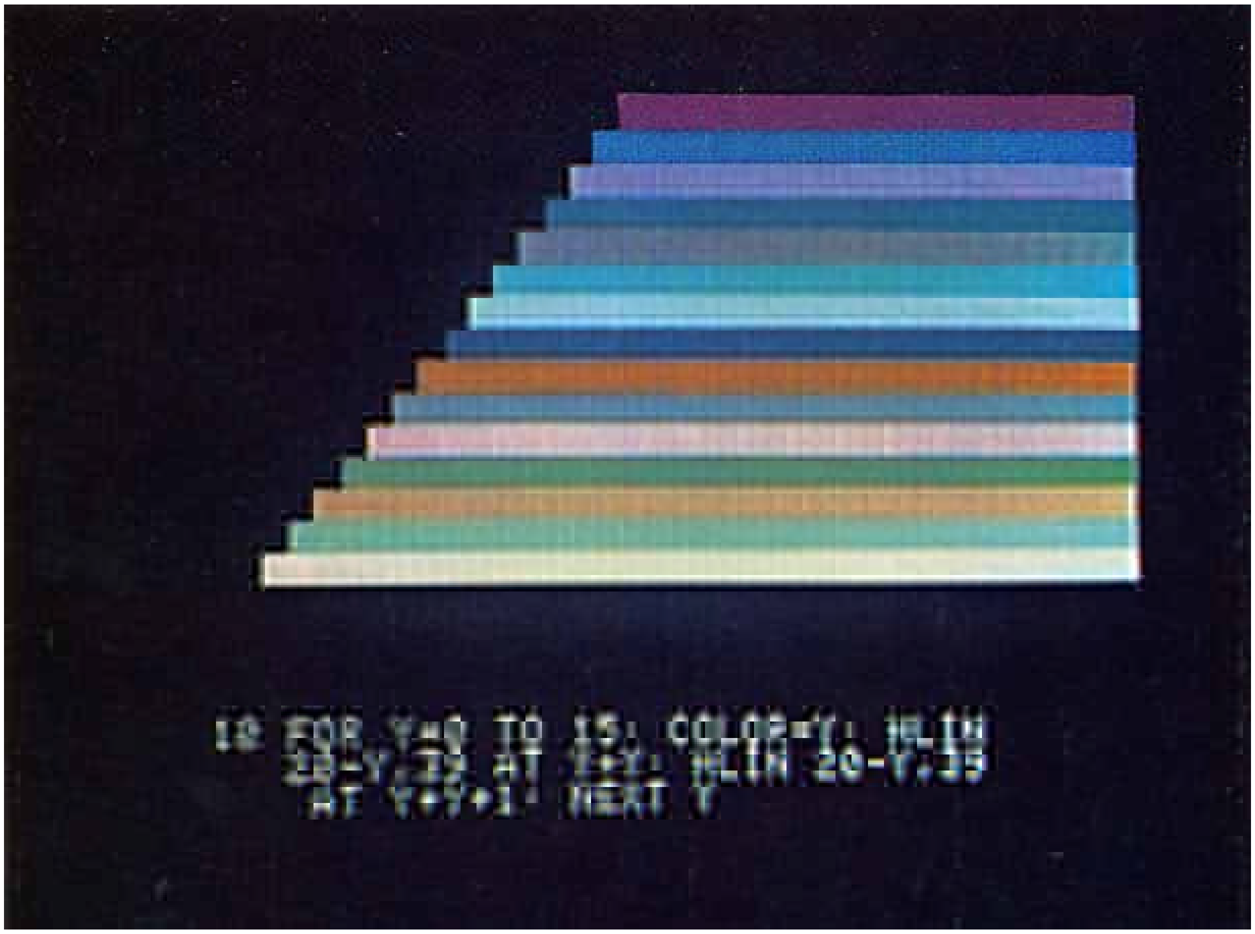
Steve Wozniak • BYTE Magazine • May 1977

http://www.oldcomputers.net/index.html
28 October 2004

System Description: The Apple-II

Stephen Wozniak
Apple Computer Co • 20863 Stevens Creek Blvd B3-C • Cupertino CA 95014





```
10 FOR Y=0 TO 15: COLOR=Y: HLIN
  20-Y,39 AT Y+Y: HLIN 20-Y,39
   AT Y+Y+1: NEXT Y
```

Photo 1: A color test chart showing the 15 shades of hue available from the Apple-II as presented on a typical commercial color set, using one of several RF modulators available on the market. The Apple BASIC program used to generate this color is shown in the text portion of this split screen (graphics and text) display.

<u>Introduction</u>

To me, a personal computer should be small, reliable, convenient to use and inexpensive.

The Apple-I, my first video oriented single board computer, was designed late in 1975 and sold by word of mouth through-out California and later nationwide through retail computer stores. I think that the Apple-I computer was the first microprocessor system product on the market to completely integrate the display generation circuitry, microprocessor, memory and power supply on the same board. This meant that its owner could run the Apple BASIC interpreter with no additional electronics other than a keyboard and video monitor. The Apple-I video computer board was originally intended as a television terminal product which could also operate in a stand alone mode without much in the way of memory, although it did have a processor, space for 8 K bytes of 4 K dynamic memory chips, and its shared video generation and dynamic memory refresh logic. Apple-I was sold as a completely assembled and tested processor board with a price under $700 at the retail level.

The latest result of my design activities is the Apple-II which is the main subject of this system description article. The Apple-II builds upon this idea by providing a computer with more memory capability, a read only memory (ROM) BASIC interpreter, color video graphics as well as point graphics and character graphics, and extended systems software.

Integral Graphics

A key part of the Apple-II design is an integral video display generator which directly accesses the system's programmable memory. Screen formatting and cursor controls are realized in my design in the form of about 200 bytes of read only memory which are built into the Apple-II's mask programmed 8 K bytes of read only memory. A 1 K byte segment of the processor's main memory is dedicated to the display generator, although it is also accessible to programs. The display transfer rate is the time it takes to fully define the contents of this segment of memory, and averages about 1000 characters per second, limited primarily by the software scrolling routines in the system read only memory. Since the Apple-II incorporates this display generator as a part of its design, its text mode becomes the terminal for the system. The display has 24 rows of 40 characters displayed on an ordinary black and white or color television screen. Each character in the Apple-I I design is a 5 by 7 dot matrix, so the present version of the system only implements upper case characters of the 6 bit ASCII subset, as well as the usual numbers and graphics available in standard character generator read only memory parts.

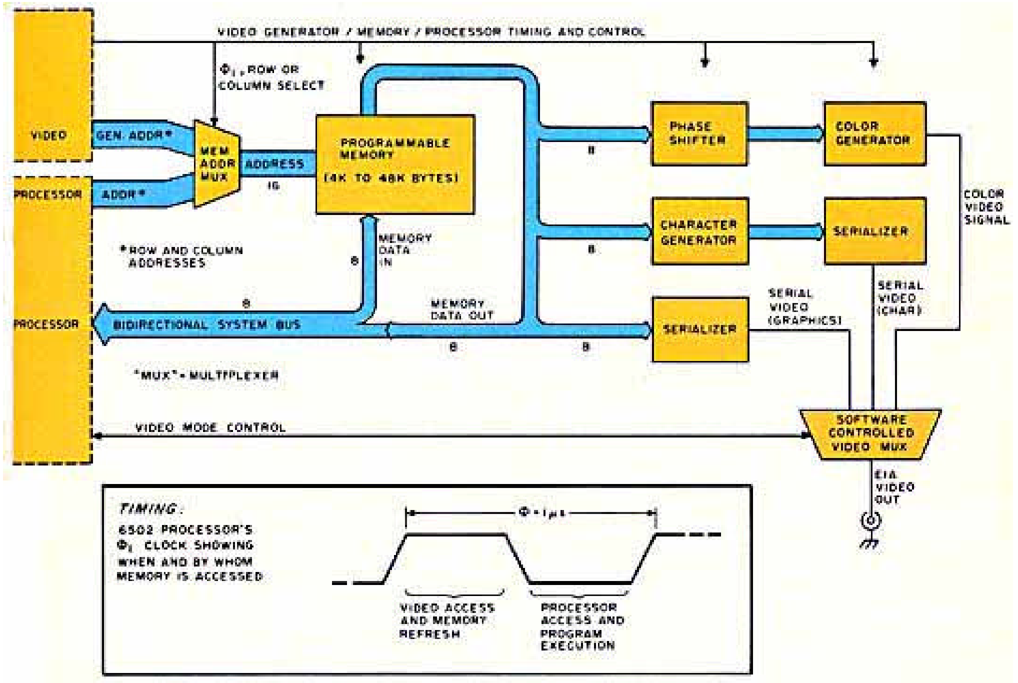

VIDEO GENERATOR / MEMORY / PROCESSOR TIMING AND CONTROL
VIDEO
PROCESSOR
Φ_1, ROW OR COLUMN SELECT
GEN. ADDR*
ADDR*
MEM ADDR MUX
ADDRESS
16
PROGRAMMABLE MEMORY (4K TO 48K BYTES)
MEMORY DATA IN
8
*ROW AND COLUMN ADDRESSES
PROCESSOR
BIDIRECTIONAL SYSTEM BUS
MEMORY DATA OUT
"MUX" = MULTIPLEXER
PHASE SHIFTER
COLOR GENERATOR
CHARACTER GENERATOR
SERIALIZER
SERIALIZER
COLOR VIDEO SIGNAL
SERIAL VIDEO (GRAPHICS)
SERIAL VIDEO (CHAR)
VIDEO MODE CONTROL
SOFTWARE CONTROLLED VIDEO MUX
EIA VIDEO OUT
TIMING:
6502 PROCESSOR'S Φ_1 CLOCK SHOWING WHEN AND BY WHOM MEMORY IS ACCESSED
$\Phi = 1 \mu s$
VIDEO ACCESS AND MEMORY REFRESH
PROCESSOR ACCESS AND PROGRAM EXECUTION

Figure 1: A block diagram of the Apple-II display generator. The generator sneaks into memory on the externally unused phase of the 6502 processor's 2 phase clock. The output of the memory is processed (after a 1 clock cycle delay)

to produce a net video output through a software controlled video multiplexer. The three major modes of operation are:

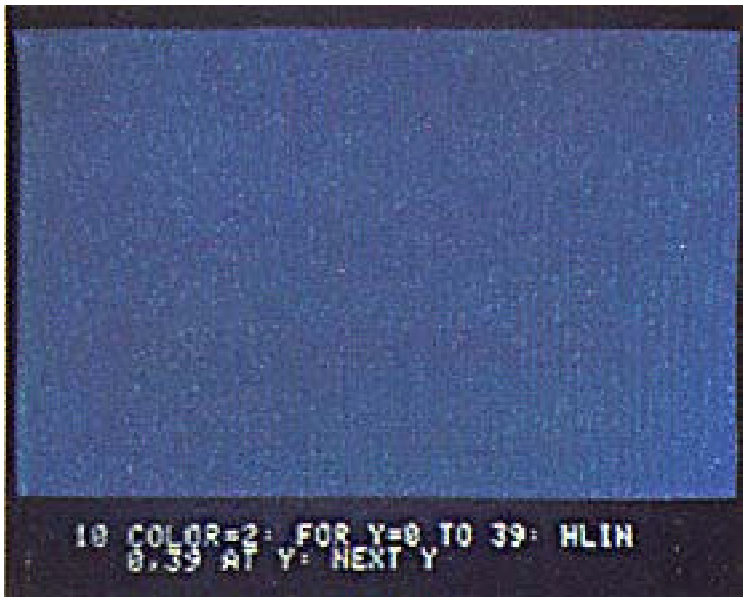
Color graphics, in which each 4 bit nyble of the byte is treated as a color definition code by the color generator.

Character generation in which the 8 bit code is processed with a read only memory to generate a dot matrix pattern which is serialized and sent to the video multiplexer.

Black-white point graphics in which the 8 bit word from memory is used to control the contents of a segment of a 280 by 160 point grid.

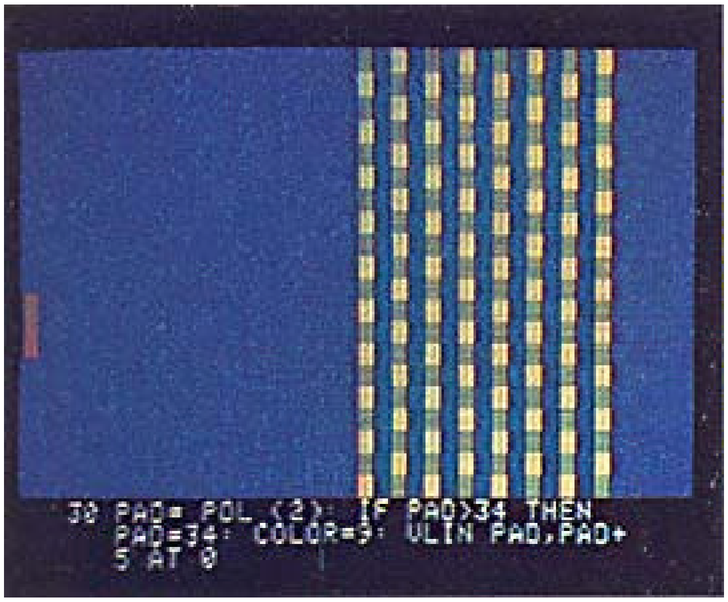
The timing diagram shows how the basic 1 us processor cycle period is split up into a video memory cycle and a microprocessor memory cycle. Since the processor is engaged in internal housekeeping operations during the first (high level) half of a 01 period, this segment of time can be used by the video generator to sneak into memory. Since all of memory is continuously being scanned by the low order bits out of video generator, the entire 48 K byte field (maximum) of dynamic memory is refreshed by the video portion of the cycle. (Refreshing of dynamic memory means scanning through all possible low order addresses to recharge the internal memory capacitors of the chips.)

Photo 2: This series of photos shows the steps in writing an animated BASIC game using the Apple-II computer's BASIC interpreter. This sequence highlights the process of writing a paddle versus "wall" game where the object of play is to knock bricks out of the wall and eventually get the ball to go all the way through. This game is similar to many seen in amusement parks and arcades, and is typical of the kind of game which can be implemented with Apple-II's BASIC software. Using the split screen graphics and text display mode, the BASIC statements are shown at the bottom of each picture.



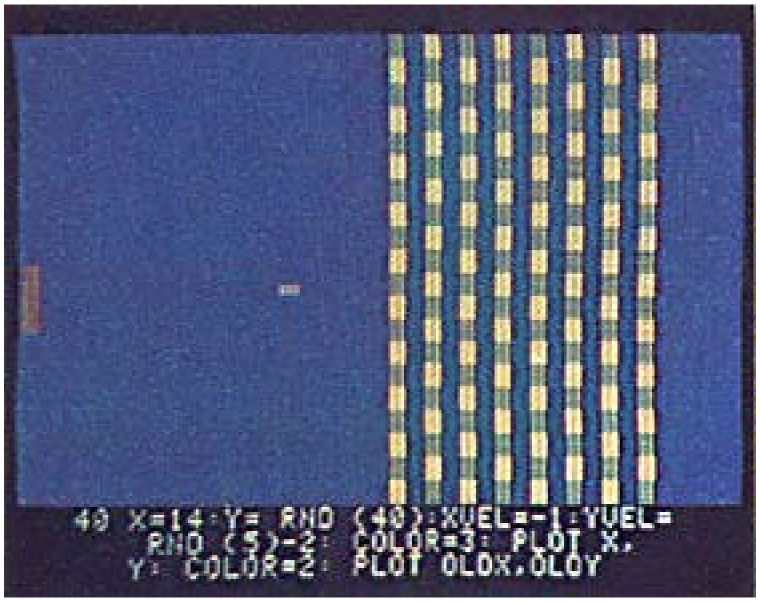
```
10 COLOR=2: FOR Y=0 TO 39: HLIN
    0,39 AT Y: NEXT Y
```

Photo 2a: The first step in any game is to generate the uniform color background for the action of the game. Here we use a blue field.



```
30 PAD= PDL (2): IF PAD>34 THEN
   PAD=34: COLOR=9: VLIN PAD,PAD+
   5 AT 0
```

Photo 2c: Next, we must of course add a paddle, here created with a deeper yellowish orange (color 9) hue.



```
40 X=14:Y= RND (40):XVEL=-1:YVEL=
   RND (5)-2: COLOR=3: PLOT X,
   Y: COLOR=2: PLOT OLDX,OLDY
```

Photo 2d: Then, since no video court game is complete without a ball we must add a square "ball" to the program, and set up some of the parameters of its motion.

Assuming that the video display is the currently assigned system output device, the display is accessed through our system software in read only memory by using a subroutine called COUT which adds text to the screen using an automatic scrolling technique. This is typical of the many read only memory routines which I've incorporated into the ROM to provide complex features with relatively simple user interfaces. Another example of such a software feature is a user definable scrolling window. This means that the user of the system can pick any of four coordinates defining any rectangular subset of the viewing area of the video screen as the current scrolling zone. The remainder of the display will remain frozen and data in the window will scroll normally when COUT is accessed. This is a most useful feature: For example, the user can set up a game background or instruction menu in one part of the screen while using the remainder of the screen for scrolling the variable data.

In the text mode, each character position may be displayed in normal (white character on black background) or inverse, or flashing modes. This information is specified by the high order bits of each character stored in the display memory, The cursor position, for example, is indicated by forcing the character at the cursor location to be in the flashing mode with inverse video.

User application programs may switch the display mode from character to color graphics with a single instruction, dividing the screen instantly into a patchwork of controllable color on a grid of 40 horizontal locations by 48 vertical locations. Each cell in the grid may be one of 15 colors, and software built into the system read only memory can be used to define the color of any point as set by X and Y coordinate integer values. Photo 1 shows a color scale for the 15 colors possible, and a simple BASIC program which generated the display. Here the scrolling window features are used to set the color graphics mode in the fixed portion of the screen (above) and set the text mode of operation in the scrolling portion (below). This mixed mode provides a 40 by 40 color graphics grid plus four lines of scrolling text at the bottom of the screen. A routine in the system read only memory selects this mode and sets up the scrolling window corresponding to the text portion. I've found this mode especially useful to BASIC programmers who can write animation games like Pong while holding a traditional BASIC conversation in the text region of the screen. This split screen mode of viewing is used for all the color graphics of photo 2 as well.

The same display memory region that is used for the text display is used for the color graphics. System software routines supplied in the read only memory of the processor allow users to simply clear the display, select colors, plot points, draw horizontal and vertical lines, and sense the color values presently at specified screen positions. I like to think of these system software subroutines as enhancements to the 6502 instruction set for the purposes of display control. High resolution graphics is the remaining Apple-II display mode. This mode of display is set up by system software routines which are delivered with the computer, but are not built into the system read only memory. (Even with 8 IC bytes for the read only memory space, there sometimes isn't enough room to fit all the needed features.) In the high resolution mode, 8 K bytes of main memory store the data for a display of 180 horizontal dot positions by 192 vertical dot positions; so to allow enough room for some BASIC software to play games with this mode the system requires at least 12 K of memory. If a color television is used with this high resolution mode, the available colors are black, white, violet and green. A mixed mode with 160 rows of 280 dots plus four lines of scrolling text can also be set up. Applications of the high resolution graphics modes include game boards, mazes, maps, plots and histograms, user definable character sets, and games like Space War in its original animation graphics versions.

Some Details

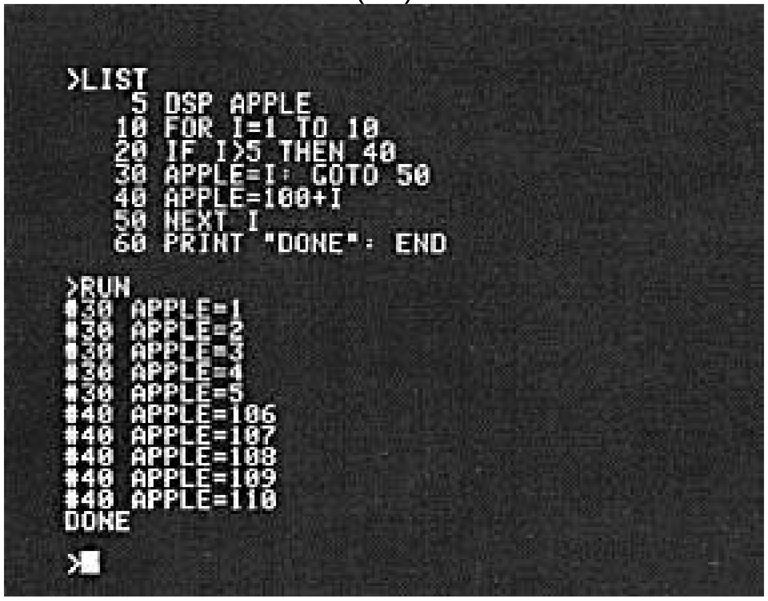
All the Apple-II video modes work identically, using a common clock timing chain which is shared by the processor, memory refresh and video generation logic. During each microprocessor clock cycle's 01 clock pulse, an address is specified by the video circuits and directed to the programmable memory of the system through the address multiplexor (MUX) of figure 1. Display data is received by the three forms of video data generators toward the end of the 02 pulse, and this data is then latched for use during the entire next clock cycle. Since all this action occurs during the 01 pulse which lasts 500 ns, the video generator is able to take over the access to the memory at a time then the 6502 processor is busy with internal housekeeping and processing operations which leave the data bus free. During the 02 pulse, when the processor takes command of the bus, the programmable memory of the system is used by the executing program as if the video generator didn't exist at all. Because the integrated display design uses this direct memory access technique without stealing processor cycles, it is possible to program accurate and predictable timing loops in software as if no DMA were present in the system.

Memory

It is alleged in the Santa Clara (Silicon) Valley that the microprocessor was invented to sell programmable and read only memory chips. It certainly has been the case that one microprocessor in the past would often support hundreds of memory chips, but times change. Technology has since bestowed upon us the 4 K bit and 16 K bit dynamic programmable memory chips. Apple-II was designed to operate with the 16 pin dynamic programmable memory parts, which come in 4 K and 16 K versions which are (with some subtleties) pin for pin compatible. The Apple-II board is supplied with sockets for three blocks of memory, each of which may be configured to use either 4 K or 16 K dynamic programmable memory parts, with intermixing allowed. This means that if you were to purchase an Apple with 4 K bytes of memory and later want to add 16 K bytes, there is no need to scrap the 4 K chips. Dynamic memories have one design characteristic which is not present in the simpler (but more expensive) static memories. This is the fact that they use capacitive storage elements built into the chips which must be periodically recharged ("refreshed") to prevent the information from disappearing. One of the elegant simplifications provided by a system such as the Apple-II with its built-in display is the fact that refreshing the entire memory address space of dynamic memory chips is inherent in the operation of the video display generator. On successive pulses of the video display, it cycles through all the low order addresses of the memories as the memory is scanned to generate the video image. But scanning through the addresses within the maximum allowable time is the algorithm used to accomplish the required refreshing of the memories; so with this video generator integral to the computer, refreshing of the memories happens to come for free and is totally transparent to the user with no extended, missing or delayed cycles. This characteristic is sometimes called "hidden refresh."

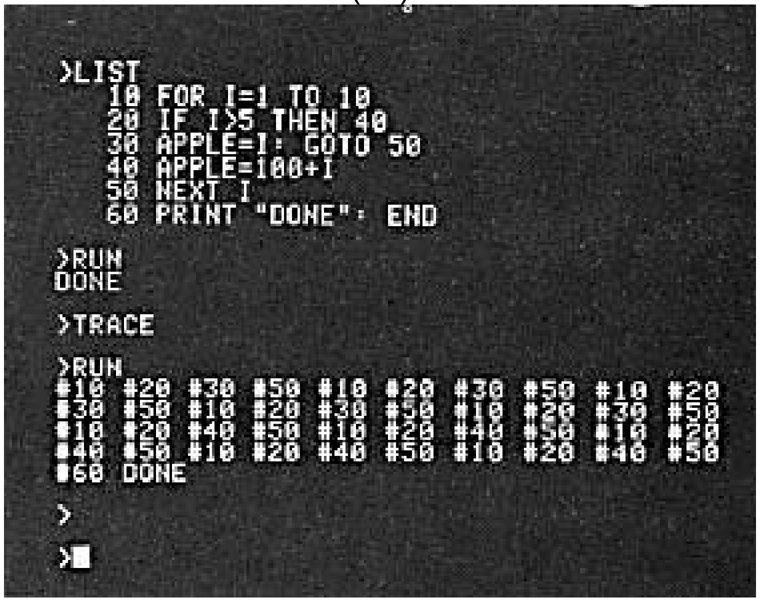
Photo 3: Two examples of the Apple BASIC interpreter, in the form of programs with several lines of execution results. (a) The interpreter has a symbolic trace feature which allows dumping of named variables whenever a change occurs. This simple program illustrates this "DSP" command with a simple computational program. (b) A similar debugging feature of Steve Wozniak's Apple BASIC interpreter is a method of running the interpreter with a statement number trace, by giving a TRACE command instead of RUN in the command mode of the interpreter. This enables one to fairly quickly debug a BASIC program by examining its effect on variables or its course of evolution through statement numbers.

(3a)



```
>LIST
    5 DSP APPLE
   10 FOR I=1 TO 10
   20 IF I>5 THEN 40
   30 APPLE=I: GOTO 50
   40 APPLE=100+I
   50 NEXT I
   60 PRINT "DONE": END

>RUN
#30 APPLE=1
#30 APPLE=2
#30 APPLE=3
#30 APPLE=4
#30 APPLE=5
#40 APPLE=106
#40 APPLE=107
#40 APPLE=108
#40 APPLE=109
#40 APPLE=110
DONE

>
```

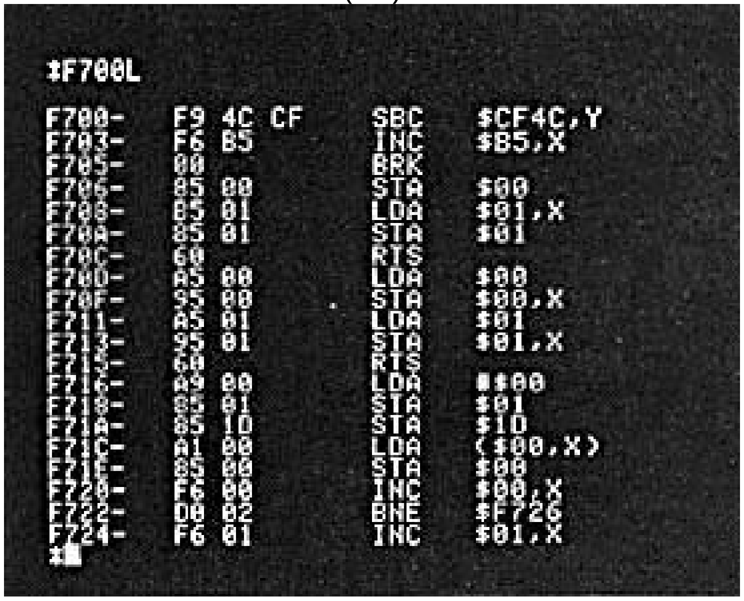
(3b)



```
>LIST
   10 FOR I=1 TO 10
   20 IF I>5 THEN 40
   30 APPLE=I: GOTO 50
   40 APPLE=100+I
   50 NEXT I
   60 PRINT "DONE": END

>RUN
DONE

>TRACE

>RUN
#10 #20 #30 #50 #10 #20 #30 #50 #10 #20
#30 #50 #10 #20 #30 #50 #10 #20 #30 #50
#10 #20 #40 #50 #10 #20 #40 #50 #10 #20
#40 #50 #10 #20 #40 #50 #10 #20 #40 #50
#60 DONE

>

>
```

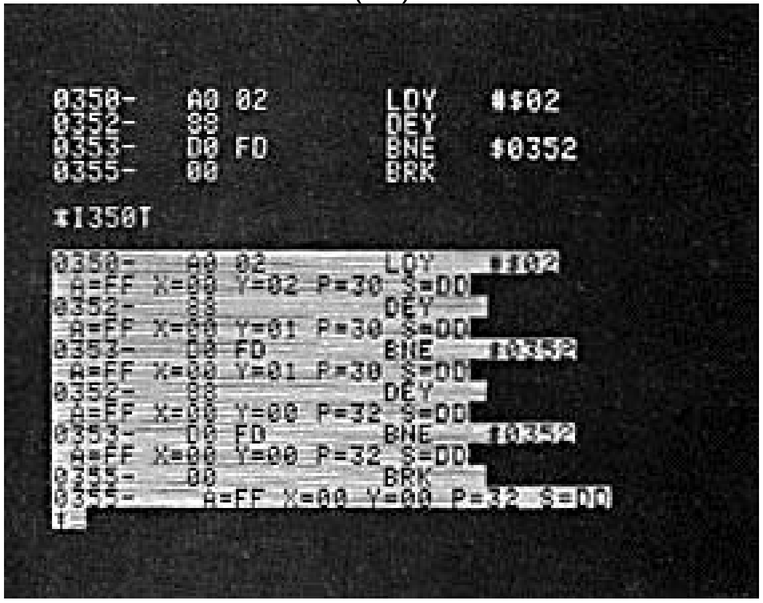
Photo 4: Far from being limited to interpretive integer BASIC, the Apple-II includes some powerful debugging and software development aids at the machine language level. Here at (a) is an example of its dissassembler mode of operation, invoked by the L command following an address in hexadecimal. A corresponding nonsymbolic assembler program will perform transformations in the other direction from text sources. Here at (b) is an example of the instruction trace command, which allows a machine language program to be followed mnemonically via dynamic disassembly, with register and condition code contents indicated after each instruction.

(4a)



```
*F700L

F700-   F9 4C CF    SBC   $CF4C,Y
F703-   F6 B5       INC   $B5,X
F705-   00          BRK
F706-   85 00       STA   $00
F708-   B5 01       LDA   $01,X
F70A-   85 01       STA   $01
F70C-   60          RTS
F70D-   A5 00       LDA   $00
F70F-   95 00       STA   $00,X
F711-   A5 01       LDA   $01
F713-   95 01       STA   $01,X
F715-   60          RTS
F716-   A9 00       LDA   #$00
F718-   85 01       STA   $01
F71A-   85 1D       STA   $1D
F71C-   A1 00       LDA   ($00,X)
F71E-   85 00       STA   $00
F720-   F6 00       INC   $00,X
F722-   D0 02       BNE   $F726
F724-   F6 01       INC   $01,X
*
```

(4b)



```
0350-   A0 02       LDY   #$02
0352-   88          DEY
0353-   D0 FD       BNE   $0352
0355-   00          BRK

*I350T

0350-   A0 02       LDY   #$02
 A=FF X=00 Y=02 P=30 S=DD
0352-   88          DEY
 A=FF X=00 Y=01 P=30 S=DD
0353-   D0 FD       BNE   $0352
 A=FF X=00 Y=01 P=30 S=DD
0352-   88          DEY
 A=FF X=00 Y=00 P=32 S=DD
0353-   D0 FD       BNE   $0352
 A=FF X=00 Y=00 P=32 S=DD
0355-   00          BRK
0355-     A=FF X=00 Y=00 P=32 S=DD
*
```

Standard Peripherals

I designed the Apple-II to come with a set of standard peripherals, in order to fit my concept of a personal computer. In addition to the video display, color graphics and high resolution graphics, this design includes a keyboard interface, audio cassette interface, four analog game paddle inputs (for user supplied potentiometers which vary a resistance which the processor measures), three switch inputs, four 1 bit annunciator outputs, and even an audio output to a speaker. Also part of the Apple-II design is an 8 slot motherboard for IO which has a fully buffered bus, prioritized interrupts, two prioritized direct memory access (DMA) schemes, and address decoding at the individual slots so that multiple bit address decoders are not required on peripheral boards.

The Apple-II cassette interface is simple, fast, and I think most reliable. The data transfer rate averages over 180 bytes per second, and the recording scheme is compatible with the interface used with the Apple-I. This tape recording method can be used with any inexpensive recorder, but as with any such use of audio media only high quality tapes should be used in order to avoid problems due to dropouts from poor oxide coatings on the tapes. In the Apple audio cassette interface, timing is performed by software which is referenced to the system clock. A zero bit is defined as a full cycle of a 2000 Hz signal (500 us long), while a one bit is defined as a full cycle of a 1000 Hz signal (1 ms long). While reading data, full cycles are sampled, never half cycles, a method which tends to provide immunity to DC offset and other forms of distortion. All the cassette management routines are available to user programs as subroutine calls from assembly language directly, or through hooks in the BASIC interpreter.

The Apple-II analog game control paddle circuits are based upon inexpensive timer chips of the 555 type. I've used a quad timer of this type, called the 553, as shown in figure 2. To read the value of resistance on the paddle's potentiometer, the timer is strobed under software control using routines in the system read only memory. The input routine then enters a loop which counts the length of the timer output pulse, which is a function of the paddle potentiometer's setting. To prevent endless loops if a wire breaks, the paddle scan routines exit at the maximum count of 255. The resolution of the loop is 12 us per count.

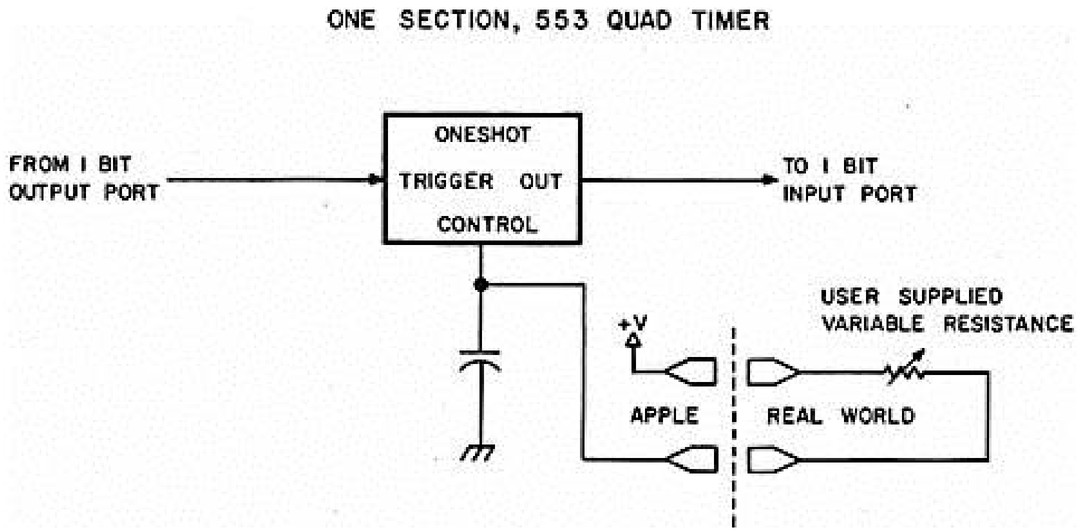


Figure 2: How to make a 1 bit measurement of an analog parameter for games (or perhaps we should say "2 bit "). Basically, a 555 style timing element is set up so that it can be triggered by a 1 bit output port. After triggering the oneshot, the processor enters a timing loop continuously testing the 1 bit input port until the end of the oneshot's cycle, which is controlled by the game parameter potentiometer. The result is an integer count developed hy the timing loop which gives a measure of how long the oneshot pulse lasted, and hence a measure of the position of the input potentiometer. Apple-II implements four of these resistance measuring ports (which have plenty of accuracy for game

contexts with graphics display feedback but are hardly not to be interpreted as having any absolute accuracy independent of hand-eye coordination).

One memory address is dedicated to the audio output port which drives a speaker. When this memory location is referenced from a program, with either a read or a write operation, the speaker drive line is toggled. Generating tones requires continuous speaker toggling by this method, at an audible rate. The cassette output port works in a similar (toggle) fashion to generate audio tones for the tape. The annunciator outputs each have two corresponding addresses, with one used to set the output and the second used to clear the outputs. Switch, paddle and cassette inputs place their data on the system bus in the sign bit position when their corresponding addresses are referenced; this choice of wiring enables software to test the state of the bit directly with a conditional branch instruction of the 6502 processor.

Apple BASIC

Apple-II comes with an Apple BASIC interpreter in the mask programmed read only memories of the system. There is no need to load it off tape, nor to dedicate any programmable memory for it. It's always there and it is impossible to accidentally clobber it. This BASIC is essentially similar to any BASIC with the exceptions that it only implements 16 bit fixed point arithmetic. It also features some unique language extensions to take advantage of the Apple-II hardware features such as color graphics and to provide coveniences in the form of debugging aids. It is intended primarily for games and educational uses.
A monitor command puts you into BASIC mode, which is indicated on the screen by a prompt character, " > ". Memory limits for BASIC source programs and data are set automatically at the time of entry, but these limits may be varied by user commands. While in BASIC mode, statements are entered on the current system input device, which is normally the keyboard.

Apple-II BASIC is implemented as a translator-interpreter combination. When a line is read from the input device, the translator analyzes it and generates a more efficient internal language facsimile. Syntax errors are detected at this time. The "nouns" of this internal language are variable names, integer constants (preconverted to binary for execution speed enhancement), and string constants. The "verbs" are 1 byte tokens substituted for keywords, operators and delimiters. Because the translator distinguishes syntax, different verbs are assigned to different usages of the same symbol, For example, three distinct verbs represent the word PRINT, depending on whether it is immediately followed by a string source, an arithmetic expression or nothing. Thus this distinction need not be made at execution time. For each verb there exists a subroutine to perform that specific action. Listing a program actually involves decompiling the internal language back to BASIC source code. Those statements with line numbers are stored as part of the user program, while those without line numbers are executed immediately. If desired, the Apple BASIC interpreter's editing functions can be set to generate line numbers automatically. Although some commands are valid only for immediate execution and others only for programmed execution, most can be employed in both ways. In the BASIC source programs, multiple statements may reside on the same line, separated by colons (':').

BASIC language statements are stored in user memory as they are accepted and variables are allocated space the first time they are encountered during immediate or programmed execution. When a program terminates, whether by completion, interruption or error conditions, all variables are preserved.

Programs may be interrupted in execution by typing an ASCII control C; it is then possible to examine and modify a few variables in immediate mode, then continue execution at the point of interruption by typing the CONtinue command. BASIC provides the line number of the statement as the point of interruption when this sequence is used. The entire variable space is cleared to zero when BASIC is initialized by the CLR command, and prior to executing the RUN command. (It is possible to carry variables from one program to another, but to initiate the second program a GOTO command must be used instead of RUN in order to override the automatic clear at the beginning of execution of a new program.) The interpreter consists of a standard expression evaluator and a symbol table routine for allocating variable storage similar to those described by Prof Maurer in his 2 part series in the February and March 1976 issues of BYTE. As statements are scanned, nouns and verbs are encountered. Variable names result in calls to the symbol table routine which pushes address and length information on the noun stack (operand stack). Constants are pushed directly onto this stack. Verbs are pushed onto the verb stack (operator stack) after popping and executing any verbs of greater priority. A verb is executed by calling its associated subroutine. Tables define priorities and routine entry addresses for all verbs. Keywords such as THEN or STEP, and delimiters such as commas and parentheses, are dealt with just as though they were arithmetic operators. Verb routines obtain their arguments from the noun stack. Because verbs such as parentheses tend sometimes to be of low, and other times of high priority, each verb is actually assigned two priorities (left hand-right hand). One represents its tendency to force execution of other verbs, the second its tendency to be executed.

Interactive Monitor

The entry into BASIC, as well as other user oriented features of the Apple-II, is provided by an interactive keyboard monitor which serves as an aid to writing and debugging machine language programs for the 6502 processor of the system. The user enters commands from the keyboard specifying data and address parameters in hexadecimal. Multiple commands are permitted on the same line and editing features facilitate error correction. I completely wrote and debugged Apple BASIC using the monitor as my only software development tool. It was of course the first hand assembled program I wrote for the system. In addition to the direct monitor commands, a number of subroutines were included in the Apple-II's mask programmed system read only memory to provide easy access to hardware features. These are the service routines which are used by the monitor, as well as BASIC and any user routines you care to code.

The Story of Sweet Sixteen

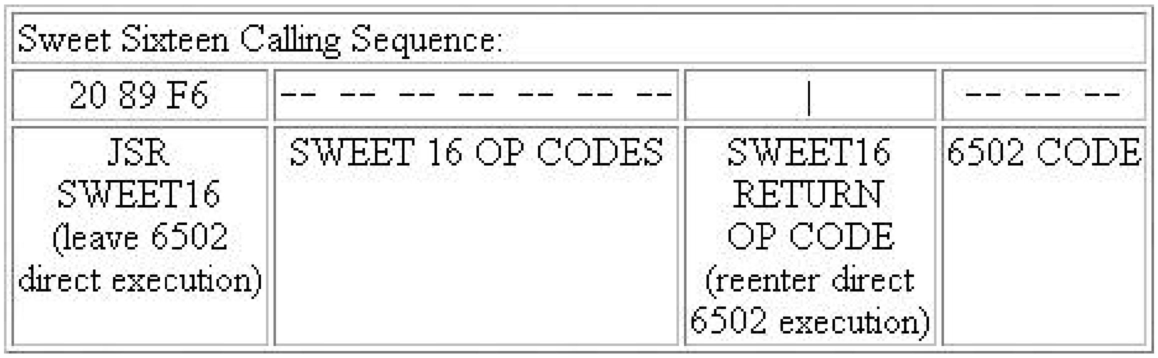
The Apple-II monitor read only memory also contains an interpreter program called SWEET16 which can be used from machine language programs to implement 16 bit arithmetic operations. This facility can prove quite useful, for example, in calculating addresses, and serves as on extension of the instruction set of the 6502 which is reached by the JSR SWEET16 escape sequence in code.

While writing Apple BASIC, I ran into the problem of manipulating the 16 bit pointer data and its arithmetic in an 8 bit machine.

My solution to this problem of handling 16 bit data, notably pointers, with an 8 bit microprocessor was to implement a non-existent 16 bit processor in software, interpreter fashion, which I refer to as SWEET16.

SWEET16 contains sixteen internal 16 bit registers, actually the first 32 bytes in main memory, labelled RO through R15. RO is defined as the accumulator, Rl5 as the program counter, and R14 as a status register. R13 stores the result of all COMPARE operations for branch testing. The user accesses SWEETl6 with a subroutine call to hexadecimal address F689. Bytes stored after the subroutine call are thereafter interpreted and executed by SWEET16. One of SWEET16's commands returns the user back to 6502 mode, even restoring the original register contents.

Implemented in only 300 bytes of code, SWEET16 has a very simple instruction set tailored to operations such as memory moves and stack manipulation. Most op codes are only one byte long, but since she runs approximately ten times slower than equivalent 6502 code, SWEET16 should be employed only when code is at a premium or execution speed is not. As an example of her usefulness, I have estimated that about l K bytes could be weeded out of my 5 K byte Apple-II BASIC interpreter with no observable performance degradation by selectively applying SWEET16.



Sweet Sixteen Calling Sequence:			
20 89 F6	-- -- -- -- -- -- --	|	-- -- --
JSR SWEET16 (leave 6502 direct execution)	SWEET 16 OP CODES	SWEET16 RETURN OP CODE (reenter direct 6502 execution)	6502 CODE

SWEET16 OP CODES (16 Bit Operands, 2's Complement Arithmetic)

Op Code	Instr Length	Description	Op Code	Length	Description
00	1	Return to 6502 mode	--	-	- - - - - - - - - -
01	2	Branch always	1R	3	R<-2 byte constant (Load register immediate)
02	2	Branch no carry	2R	1	ACC<-R
03	2	Branch on carry	3R	1	ACC->R
04	2	Branch on positive	4R	1	ACC<-@R, R<-R+1
05	2	Branch on negative	5R	1	ACC->@R, R<-R+1
06	2	Branch if equal	6R	1	ACC<-@R double
07	2	Branch not equal	7R	1	ACC->@R double
08	2	Branch on negative 1	8R	1	R<-R-1, ACC<-@R (pop)
09	2	Branch not negative 1	9R	1	R<-R-1, ACC->@R
0A	1	Break to monitor	AR	1	ACC<-@R (pop) double
0B	1	No operation	BR	1	COMPARE ACC to R
0C	1	No operation	CR	1	ACC<-ACC+R
0D	1	No operation	DR	1	ACC<-ACC-R
0E	1	No operation	ER	1	R<-R+1
0F	1	No operation	FR	1	R<-R-1

Notes.

1. All branches are followed by a 1 byte relative displacement. Works identically to 6502 branches.
2. Only ADD, SUB and COMPARE can set carry.
3. Notation:

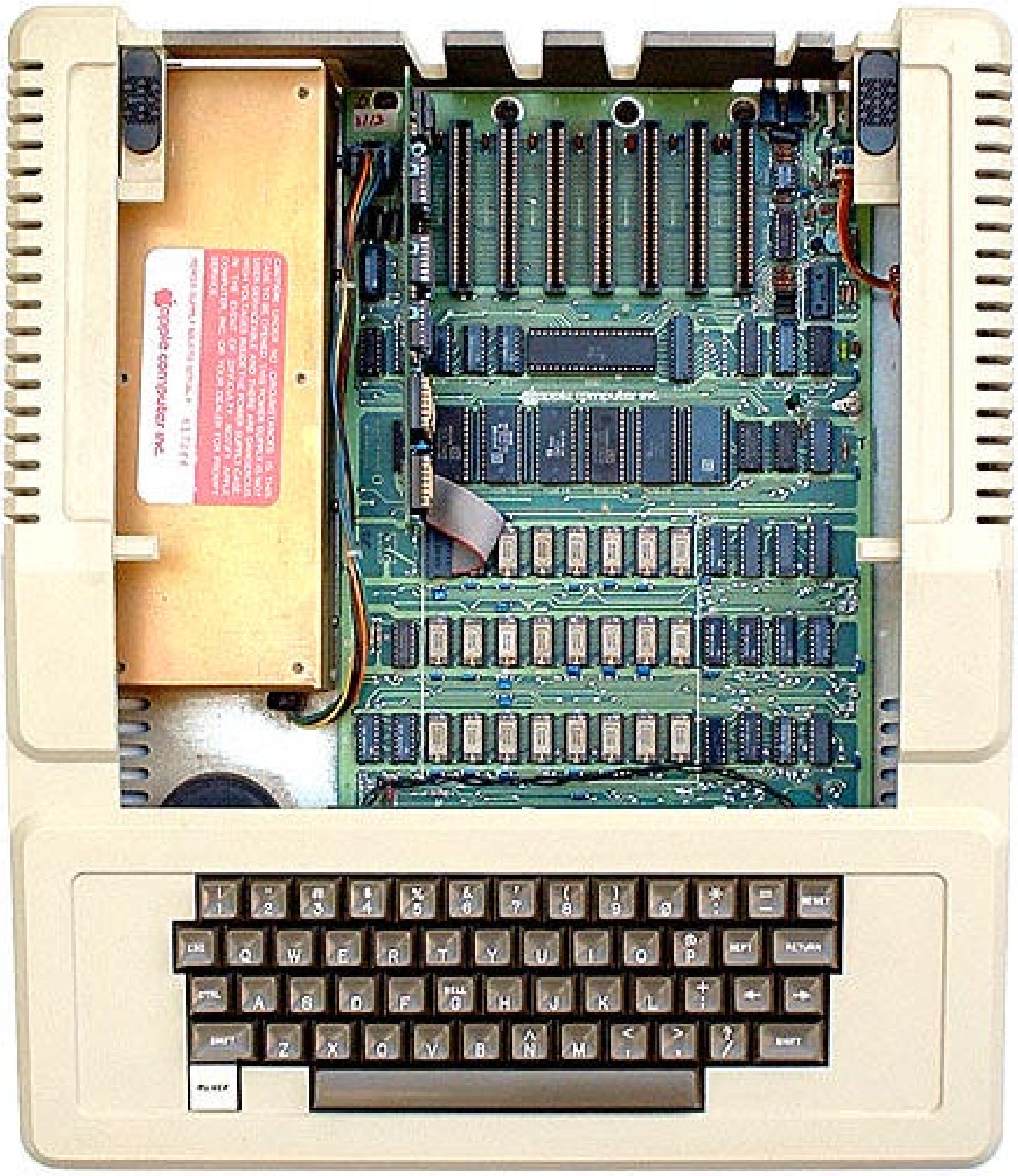
    R = a 16 bit "register" operand designation, one of 16 labelled 0 to 15 (decimal),0 to F (hexadecimal).
    ACC = register operand RO. @R = indirect reference, using the register R as the pointer.
    -> and <- = assignment of values.

4. Length of instructions:

    Branches are always two bytes: op code followed by relative displacement.
    Load register immediate (1R) is three bytes: the hexadecimal op code 10 to 1F followed by the 2 byte literal value of a 16 bit number.
    All other instructions are one byte in length.

Author's Note

So as to not slight their efforts, I would like to thank Allen Baum for originating the Apple-II debug software, Doug Kraul for helpful suggestions on the I/O structure, and Randy Wigginton and Chris Espinosa for many long and late hours testing the Apple BASIC. . . SW



###