Simulating the Caltech Burroughs B220 on an Apple //e, v2.1

Michael J. Mahon

What's a B220?

The Burroughs 220 was the last commercial vacuum tube computer. It was the successor to the Electrodata/Burroughs 205, which used a magnetic drum for its main memory. The B220 provided a needed technology update by adopting the then-new magnetic core random-access memory while retaining the existing vacuum tube logic modules.

It's likely that you've already seen the console of a B220 with its prodigious quantity of neon lights, since it has often been cast as "the computer" in many TV shows and movies—with all its neon lights replaced with brighter incandescent lights.

Its maximum memory size was 10,000 words, each having ten 4-bit BCD data digits and a BCD sign digit. In the pre-ASCII, pre-lower case days, each machine encoded alphanumeric characters differently, with six bits being a common choice for binary machines, and two BCD digits for decimal machines. Each signed, ten-digit B220 word could hold five alphanumeric characters, indicated by a sign digit value of 2. Positive numbers had an even sign digit, usually 0, and negative numbers had an odd sign digit, usually 1. Other sign values were used for control purposes, such as flagging instructions whose addresses should be relocated upon loading.

S	1	2	3	4	5	6	7	8	9	0

B220 Word Format

The B220 was intended both for commercial markets, with its BCD data representation, and for scientific and engineering uses, with its hardware-supported floating-point arithmetic. Like its B205 predecessor, the B220 had an index register, the B register, named after the "B box" of the Manchester Mark I computer. Index registers were still quite new and greatly simplified operations on tables, arrays of data, and linked lists, and also enabled the B220 to relocate library programs on-the-fly as they were loaded.

B220 input/output devices were paper tape readers and punches, printers, card readers and punches, and magnetic tape units. It could also use a novel magnetic tape device containing 50 loops of magnetic tape, each 50 feet long--an early mass-storage device with significantly lower latency than a magnetic tape.

I've included scans of three handy books that describe the B220 and its instruction set and peripherals in great detail for those who would like to learn more, and perhaps even write some B220 programs to get a feel for it. The first, **B220 Operational Characteristics**, provides a detailed overview of the system and each instruction. The second, **An Introduction to Coding the Burroughs 220**, covers the instruction set from the point of view of a programmer. The third, **B220 Operating Procedures**, is a detailed guide to using the system, with special attention to the various I/O devices.

Why a B220 Simulator?

The B220 was the second computer that I programmed, after the IBM 1620. In 1963 it had already been superseded as Caltech's primary computer system, by an IBM 7090. When the campus computing load moved from the B220, it became a "research computer" for which students and faculty could sign up (on a blackboard) for blocks of time. Since this seemed the ideal environment for learning a machine and experimenting, I was immediately drawn to it. Though I didn't realize it at the time, it was essentially a room-sized personal computer!

When I "met" the Caltech B220, it didn't have a symbolic assembler. It did have an Algol 58 dialect compiler, known as BALGOL, and a "numeric" assembler that allowed "sections" of numeric machine language programs to be combined using section-relative addresses—neither of which provided the kind of programming experience I was accustomed to. (Much later I learned that several B220 symbolic assemblers existed, but they were not available in the Caltech environment.)

So I came to write my first assembler. It was a crash course in scanning and table searching on this venerable machine, and I learned a lot about "wordy" computers. My "Symbolic-Numeric Assembler Program" or SNAP ("Numeric" because it also supported the older numeric assembler pseudo-ops) became pretty popular among the small group of Caltech people still programming the B220. When Caltech decided to augment the B220 by attaching a DEC 340 CRT display, I used it to write interactive graphics programs for visualizing and analyzing the voluminous data from nuclear physics experiments. The appendix describes the Caltech extensions to the B220 in some detail.

Seventeen years later, in 1980, I purchased my first Apple][+ computer. As I programmed it and used it, I realized that the experience I was having on a desk in my home was very similar to my experiences with the Burroughs 220—minus all the blinking lights—and I loved it!

Over the years, I thought several times about writing a simulator for the B220 on my Apple][+. There was enough memory, since the Caltech B220 had just 5000 words—about 30,000 bytes. And the 6502's BCD mode gave me hope that the instructions could be efficiently simulated. The sticking point for me was the decimal address decoding.

I could only see a choice between the memory-wasting method of using the B220's BCD addresses as binary addresses, which would result in 37% of the memory being unused, or the very tedious approach of converting the BCD addresses to binary on each storage access. Then, recently, I realized how I could do it all with tables, quickly and compactly, and I began coding version 1 of B220SIM on the Apple II almost immediately!

The Fetch Routine

Of course, there is another way of providing a dense BCD-to-binary address mapping that is also quite efficient: indirect mapping through tables. The natural mapping of B220 word memory to 6502 memory is to allocate 6 bytes to hold the B220's 11 BCD digits, with one wasted digit—the high order digit of the byte containing the sign digit. This mapping causes the BCD address part of each instruction to occupy two bytes, one containing the upper two digits and one containing the lower two digits.

Un-used	S	Variant				Op		Address			
-	S	1	2	3	4	5	6	7	8	9	0

B220 Instruction Format

In this scheme, converting a 4-digit B220 BCD address to the binary address of a 6-byte “word” requires:

1. Checking the B220 address for invalid BCD digits,
2. Converting the B220 address to binary,
3. Multiplying by 6 (bytes per B220 word), and
4. Adding the base address of simulated B220 memory.

That seems like a pretty tall order to do quickly on a 6502, but it turns out that with a couple of tables, one indexed by the two low-order BCD digits and the other indexed by the two high-order digits, plus some careful arranging, the complete binary address formation can be performed by a 6502 in just 38 cycles, including range and validity checking!

The table-lookup method of converting addresses combined with a straightforward BCD opcode lookup table pretty well handles the common “instruction fetch” cycle of the B220. There’s some additional logic related to “run” mode versus “single step”, error halts, periodic simulated register display, and B-register address modification, but that’s all pretty straightforward, though it does add cumulatively to the fetch path.

Take a look at the simulator’s ‘fetch’ routine on pages 76-78 of the B220SIM listing to see the result. Note that the program counter is kept in both BCD (rP) and binary format (instptr) to avoid conversions when fetching sequential instructions.

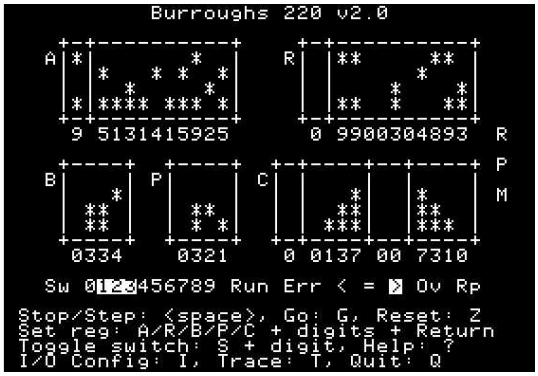
The Execute Routines

In general, there is an execute routine for each opcode. Since several B220 instructions are minor variations of each other, some routines are combined, or share significant implementation code. It’s a good thing that some combination is possible, because on a 64KB Apple //e with ProDOS and BASIC.SYSTEM resident there is less than 4KB left for the execute routines after allocating 30K of simulated B220 memory and about 2KB for the fetch, display, and human interface routines for the simulator. B220SIM version 2 runs on a 128KB Apple //e, so this version 1 constraint is lifted, allowing for a more complete implementation and numerous additional features.

Initially, I had only planned a “skeleton” implementation—just enough instructions to convince myself that it would work. But, like many other projects, it took on a life of its own, and I set about fleshing it out with, first, all basic instructions without the floating-point instructions, then, all but the mag-tape instructions, and currently, all but the Cardatron instructions (since the Caltech system had no card devices). The latest version, 2.1, includes a complete implementation of the rather extensive B220 mag-tape subsystem and buffering for all I/O devices, a trace routine for debugging 220 programs, and the Caltech CRT functionality, complete with its light pen and keyboard. These features require a //e or IIc with 128KB RAM.

You can see all the instruction execute routines in the B220SIM v2.1 listing starting at page 84.

Actual B220 panel (simulated portions highlighted)



B220 simulator panel

Implementation

Although the B220 is a decimal, word-oriented architecture, many of its instructions permit operations on “partial fields” of words—contiguous groups of digits treated as unsigned integers. Such fields are specified by two decimal digits, one of which designates the low-order “start” digit of the field, from digit 1 to 10 (represented as 0), and one of which is the length of the field extending to the left, again from 1 to 10, but never extending beyond the sign digit of the word.

S	1	2	3	4	5	6	7	8	9	0

B220 Word Format (04 field highlighted)

The fact that the simulator groups digits two to a byte implies that the “even numbered” digits are low-order digits in a byte, while the odd numbered digits are high-order in a byte. Since partial fields can start and end at any digit, digit alignment may require a one-digit (four-bit) shift. Similar shifts are required for multiply, divide, and (of course) shifts, as well as several instructions that use a two-digit immediate in the instruction as data to operate on an arbitrary partial field.

For efficiency, it is desirable to minimize the number of partial-byte (one-digit) shifts, and to maximize full-byte (two-digit) shifts, since they do not require iterated 6502 bit shift instructions, only data movement.

For example, multiply is implemented by first making a one-digit-shifted copy of the multiplicand (x10), and then handling two digits of multiplier per iteration, with all subsequent shifting done by byte moves.

Floating-Point

The B220 was unusual in providing both BCD data representation for commercial applications and floating-point arithmetic instructions for scientific applications. Floating-point add and subtract are implemented directly, while floating multiply and divide are implemented by using the fixed-point multiply and divide implementations as subroutines, just as the real B220 did.

Simulator I/O

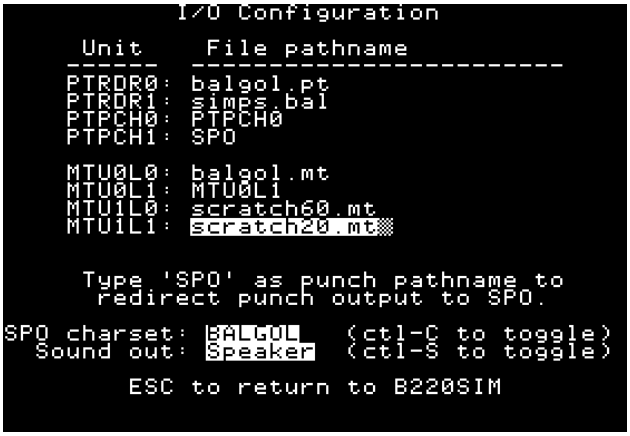
Although the B220 could attach to a pretty wide variety of I/O devices, the Caltech machine was intended for scientific applications and so had a relatively limited I/O configuration. It had one 1000 character-per-second paper tape photoreader, one 60 cps paper tape punch, one 10 cps teletype printer, and one electrostatic printer that could print about 200-300 lines per minute, depending on line length. It also had two B220 mag-tapes, which could transfer about 1900 B220 words per second. I based the simulator I/O configuration on the Caltech machine.

B220 magnetic tapes are a bit unusual. Each drive mounts one tape, but each tape can be read or written in two independent “lanes”—independent, that is, except for positioning, since the lanes are parallel on the tape. So, for now, consider that each tape unit is like two units in terms of capacity.

I decided to implement both paper tape and mag-tape as ProDOS binary disk files, with the data being simply the B220 memory image of the record. Version 2 implements buffering for all I/O devices (although, since Apple I/0 is not overlapped, “blocked” I/O is the more accurate term). Physical I/O for paper tape readers and punches is done in blocks of 100 words, and mag-tape I/O is done in blocks of 1000 words, which reduces the number of physical I/O operations considerably. On-screen logical I/O activity indicators are provided for all devices by displaying an inverse unit number next to the R(eader), P(unch), and M(ag-tape) letters on the right side of the panel display (see above).

I also chose to provide the simulator with an additional paper tape reader, since I had always wanted readers for both a boot device and a data reader. So B220SIM provides two paper tape readers: PTRDR0 and PTRDR1, two paper tape punches: PTPCH0 and PTPCH1, and two magnetic tapes, each with two lanes: MTU0L0, MTU0L1, MTU1L0, and MTU1L1.

Each of these units can be associated with a ProDOS file using the I/O Configuration screen. If no file name is configured, the default file name is the unit name, e.g.: “PTPCH0”, for example:



The I/O Configuration screen allows the user to change the ProDOS filename associated with each I/O unit from its default, which is the unit name, and to change certain B220SIM options. Editing is rudimentary but sufficient. The up and down arrow keys select the desired unit file name, and left arrow and delete keys delete the character to the left of the cursor. Typing appends characters. Escape returns to B220SIM.

The only B220 output instruction that results in formatted text output is SPO (Supervisory Print-Out). It prints to the 4-line “help” region at the bottom of the simulator’s screen. If desired, SPO output can be redirected to a printer (or a text file if running on an Apple II emulator) by typing “PR#1” at the Applesoft BASIC prompt. For convenience, paper tape punches can be redirected to the SPO by specifying “SPO” as their file name.

The default character set for the SPO corresponds to the Burroughs “Whippet” electrostatic printer used at Caltech, which differs slightly from other B220 printers. BALGOL was designed to work with a more standard character set. The I/O Configuration screen provides for switching the SPO character set between the Caltech set and the BALGOL set by pressing control-C to toggle between the two choices.

This screen also allows switching the sound output (one click per instruction executed) between the speaker and the cassette output by pressing control-S (which is also a handy way to silence the speaker if desired).

One paper tape read instruction, PRB, has unusual, but very useful, behavior. Instead of specifying a fixed number of words to read in, it reads paper tape until it encounters a specially flagged word which, instead of being stored in memory, is passed immediately to the C (instruction) register to be executed. This permits “scatter loading” of code and data, and typically finishes with a flagged branch to the start of the program, making it the B220’s natural boot instruction.

Unlike version 1 of B220SIM, version 2 provides a complete implementation of the B220 mag-tape subsystem. In additiion to the usual tape read and write instructiions, the mag-tape subsystem can SEARCH for blocks with ordered initial words and SCAN for blocks with specified content in any of the first ten words of a block. Version 2.1 extends the simulation further, with an approximate simulation of the 1000x1000 point DEC 340 CRT display using the Apple II high-resolution graphics mode. See the appendix for more information on the Caltech extensions. Version 2 also implements a trace feature which outputs each executed instruction and the register and indicator state to the SPO. It's not on the Help menu, but "T" toggles tracing on and off.

Testing the Simulator

When I wrote version 1 of the simulator, I had no machine-readable code to run on it, so I had to type in some code listings that I had saved from when I had access to the real machine. The most suitable of these was the assembler I had written for the B220, called SNAP, version 1E. This was primarily a paper tape assembler, which used magnetic tape only to save the source read from paper tape during the first pass so that it could be re-read from magnetic tape during pass 2 to save time and rewinding the paper tape reader.

Since its use of magnetic tape was so straightforward, B220SIM version 1 got by with a very simplified mag-tape implementation—only read, write, position forward and backward, and rewind were needed. This was fortunate, since there was very little of the 64KB memory space left to implement the mag-tape operations.

This limited implementation of the B220 mag-tape subsystem was adequate for SNAP and other simple uses, but it was insufficient for running more complex programs, like the Burroughs Algebraic Compiler 220, also known as BALGOL--an early but quite capable implementation of the ALGOL 58 programming language.

Another admirer of the early Burroughs/Electrodata machines, Paul Kimpel, has also written a Burroughs 220 emulator, in JavaScript, and has painstakingly typed in the entire listing for the 1961 BALGOL compiler, together with its libraries and the "generator" program that customized the compiler for a particular installation's memory size and I/O configuration. He very kindly supplied me with a mag-tape image of the compiler customized for paper tape input and output. The compiler runs on a 5000-word 220 with two mag-tape drives, one for the compiler and library and one for "scratch" use. Many thanks to Paul for his work and for permission to include BALGOL.MT on my web site. His extensive Burroughs 220 web site is: https://github.com/pkimpel/retro-220/tree/master/software/BALGOL/BALGOL-INPUTMEDIA-OUTPUTMEDIA/PaperTape-Media.

Running this compiler on B220SIM became my "holy grail," and started me down the path to writing B220SIM version 2.

Paul also supplied several paper tape images of B220 hardware diagnostics to test the simulator just as the real machine was tested! This was indispensable for catching some subtle discrepancies in arithmetic exception behavior. I combined four of these diagnostics (after removing redundant looping behavior) into a regression test called "regress.obj", which should be run from PTRDR1.

Simulator Speed

The B220 was not fast by today's standards. Its basic clock speed was 200kHz, and data was transmitted in the machine as serial streams of 4-bit digits. Fetching an instruction from memory to the C register required 90 microseconds. A fixed-point ADD executed in 185 or 245 microseconds, depending on the need to decomplement the result. Therefore, if we could fetch and execute a B220 instruction in about 200 cycles of a 1MHz 6502, we would achieve approximate speed parity with a B220. This was an encouraging result.

Unsurprisingly, there are quite a few error checks, corner cases, and mode options to be handled, and the current simulator runs at approximately a third of B220 speed with a 1MHz processor. The good news is that a moderately accelerated Apple (8MHz) runs considerably **faster** than the native B220. Most Apple II emulators can run at an equivalent 6502 clock speed of tens of megahertz, so B220SIM can really fly under emulation!

Future Plans

The B220 Simulator has met all my short-term expectations: running BALGOL, SNAP 1E, and my various diagnostic and demo programs. It certainly demonstrates that an Apple II computer can effectively simulate one of the last early "mainframes".

When the Caltech B220 was decommissioned in 1967, I managed to rescue the library mag-tape, which contained numerous relocatable library routines, plus the SNAP assembler and the BALGOL compiler and its libraries. I'm quite sure that no working B220 tape drives exist, but, in principle, this ¾" 12-track tape could be "read" using special equipment, allowing recovery of a substantial collection of B220 code. This remains "an idea in search of a project", but it is a possibility!

Using B220SIM

B220SIM is a standalone binary program that runs under ProDOS and BASIC.SYSTEM. The program is started by typing "-B220SIM" at the Applesoft prompt.

When the simulated "panel" is displayed, the simulator is ready to run. Since its memory is initially cleared to zeroes, a B220 program must be entered before it can do anything besides execute Halt instructions (00). Programs and data may be entered manually, by placing the desired instruction or data into the A or R register and putting an appropriate Store instruction into the C register, then "stepping" one instruction by keying the space bar. This gets quite tiresome for more than a few words, so programs are usually loaded from paper tape.

After starting B220SIM, type "I" to enter the I/O Configuration screen, where you can enter the boot file name for the desired paper tape reader unit. You can also configure the file names for data input and data output. If you wish to send output to a printer, the paper tape punch units can be redirected by entering "SPO" as their "file name". This screen also permits selection of the SPO character set and the sound output port. When the I/O configuration is complete, press ESCape to return to the simulator.

To boot the simulator from paper tape reader 0, enter "00000040000" into the C register (leading zeroes are unnecessary if the C register is already 0), then press "G" (Go) to start the simulator, which will load the program and immediately start running it. To boot from reader 1, the unit number "1" should be entered into the C register in place of the second "0".

At any time, you can press the space bar to stop the running program, press it again to single-step through the program, or press "G" to resume continuous execution. While the simulator is stopped, you can alter any register by typing its name ("A", "R", "B", "P", or "C"), followed by any number of digits. As each digit is typed, the register is shifted left and the new digit enters at the right. To "accept" the value in the register, press RETURN. This provides an easy way to examine and change the simulated processor state, just as the registers could be changed manually at the real B220 console

The B220 also has 10 switch inputs that can be used to control the running program. The simulated switches are indicated by the title "Sw" and a string of the ten digits. If a digit is inverse, that switch is on, and if a digit is displayed normally, that switch is off. Any switch can be toggled by typing "S" followed by the desired digit.

If it is desired to zero the registers of the simulated B220 and "rewind" all the simulated paper and magnetic tape units, type "Z". The B220 memory is not changed by this command.

While the simulator is stopped, you can also type "Q" to quit to Applesoft. This can be used to allow disk commands to be typed, for example, to CATALOG files or to use "PR#1" to direct SPO output to a printer. Most BASIC commands will fail, since B220SIM takes over much of page zero. To return to the simulator with its memory intact, type "mtr", then "803g".

All of these commands are summarized in a 4-line help window below the simulated B220 panel. These lines are also where Supervisory Print-Out (SPO) instructions write their output during simulation. If SPO commands have scrolled the help lines off, typing "?" will re-display them.

An Example: Compiling and Running a program with BALGOL

Let's examine the process of using B220SIM to compile and run a program written in BALGOL called SIMPS.BAL, which uses Simpson's Rule to integrate a function to compute the value of pi to eight significant digits.

The first step is to start BASIC.SYSTEM and use CVT2B220 to convert SIMPS.BAL.TXT to SIMPS.BAL, which is the B220-coded source file to be read by BALGOL. CVT2B220 is versatile enough to extract the source file from a listing file, and it has a number of options to facilitate this. Type in "SIMPS.BAL.TXT" as the input file, and "SIMPS.BAL" as the output file. Then specify a BALGOL conversion ("4"), producing 14 B220 word records as output (again, this is the input format expected by BALGOL). In a few seconds, CVT2B220 will complete, having created "SIMPS.BAL" containing the desired source.

If you want to produce a text listing of the assembly, type "PR#1", then start the simulator by typing "-B220SIM". After the simulator loads, you will see the simulated panel of the B220.

To run BALGOL to compile the source, configure the B220 boot file ("PTRDR0"), and the source input file ("PTRDR1") and object code output file (PTPCH1). Type "I" to enter the I/O Configuration screen and enter "BALGOL.PT" as the file name associated with "PTRDR0", the B220SIM boot device. Also enter "SIMPS.BAL" for PTRDR1 and "SPO" for PTPCH1. Configure mag-tape 0 (MTU0L0) as "BALGOL.MT" and mag-tape 1, lane 0 (MTU1L0) as "scratch60.mt" and unit 1, lane 1 (MTU1L1) as "scratch20.mt". The unit 1 assignments provide scratch tapes for BALGOL to use for intermediate storage. You will also want to set the SPO character set to "BALGOL". (The I/O Configuration screen shot three pages back illustrates the proper setup.) When you've finished, type ESCape to return to the simulator.

Enter the boot instruction into the C register by typing "C" followed by enough zeroes (six is plenty, but none are needed if the C register is zero), "4", and four more zeroes, followed by ENTER. The C register should now be "00000040000", which is a "Paper tape Read and Branch" (PRB) instruction used to boot the short compiler callout file associated with PTRDR0.

Now type "G" (Go), and BALGOL will load and compile SIMPS.BAL. If it compiles without errors, the compiler will come to an O-K halt (C = 0 0757 00 7250). To run the program, type "G" again. The compiler output, followed by the program output, will be sent to paper tape punch unit 1 (PTPCH1), which is redirected to the SPO, and therefore to the printer or an emulator's printer output file.

When the program completes at the "X-X" halt, type "Q" to quit the simulator.

If you would like to see the code produced by the compiler, turn on switches 1,2, and 3. The compiled code and libraries will be interleaved with the source program listing. The BALGOL manual has additional information on console switch settings.

You can view a B220 format file by writing a short B220 program to read in a block of words and then print them out using the SPO opcode, which converts to ASCII text. I leave this as an exercise for the reader. ;-)

Now, refer to the BALGOL manual to write your own BALGOL programs! Note that the compiler only sees columns 2-70 of each source file line, and it uses period (".") for multiply and asterisk ("*") for exponentiation and string quotes. It also supports implied multiplication, so that "4S" is equivalent to 4.S (that is, 4 times S). As always, reading the manual is advised!

Longer BALGOL programs may require larger "scratch" mag tapes than those provided. They can be created easily by keying in a simple loop to do "initial writes" (MIW) of the desired number of 100-word blocks.

Have fun writing and running programs in this surprisingly sophisticated 60-year-old language!

Appendix: Caltech Interactive Graphics Extensions to the Burroughs 220

In 1965, Caltech augmented their B220 by attaching a DEC 340 graphics display to enable early prototyping for a real-time experimental support system. The display was able to display individual points on a 1000x1000 coordinate system. A variant was added to the B220 clear and add instruction that caused it to send successive words to the display, with each word causing the momentary illumination of a single point on the screen, until an end word was reached. To produce a stable display, the list of points to be shown had to be constantly refreshed. This resulted in a program structure which consisted of a main loop which repeatedly refreshed the "display list" while detecting interactive events requiring additional processing—a structure which has since become the norm for interactive graphical programs.

The primary purpose of the extensions was to provide a high-resolution graphical display terminal configured for interactive use, inspired by the then-novel Sketchpad at MIT and the Culler-Freed system at U. C. Santa Barbara.

The CRT display was the output device, supported by modifications to two op codes, and input was provided by: 1) a light pen which could interrupt the display when a displayed point was "seen", 2) ten pushbuttons which also interrupted the display and provided a digit identifying the button pushed, 3) a general-purpose keyboard which interrupted the computer by causing a "Reset/Transfer" operation, and whose key code could be read as a pseudo paper tape reader, and 4) a second keyboard with thirty-two "function keys" which were to be used to invoke programmed functions.

Here's a 1965 photo of me holding the light pen in front of the CRT display!

The following sections describe each of these modifications in more detail and how they are simulated by B220SIM.

The Opcode extensions

The main instruction for displaying points on the CRT is an extension of the Clear and Add (CAD) instruction, which was given a new variant specified by the 4:2 variant field equal to 10. When this variant is used, the CAD instruction becomes the **Display (DIS)** instruction. The operand address, which may be B-modified, specifies the address of the first point to be displayed, and additional points continue to be fetched and displayed from ascending memory addresses, until either a light pen detection or a pushbutton causes the DIS instruction to complete with the currently displayed point, or a word is fetched which has the "continue" bit off. B220SIM maps the 1000x1000 point CRT display to a 256x192 section of the Apple HGR screen which can be zoomed (with "+" and "-") and panned (with the arrow keys). This View mode is enabled by pressing "V" and ended by pressing ESC. The format of a displayed point is shown below:

Display Word Format

The 1:1 field is the **intensity** digit. A value of 1 indicates zero intensity and a value of 7 indicates maximum intensity. Points with an intensity of 0 display in the intensity of the previous point. (This field is ignored by B220SIM, which displays all points in the same intensity.)

The 1-bit of the 2:1 field is the **continue** bit. The DIS instruction continues to display points in ascending addresses until it encounters a word with this bit off, the light pen detects a point, or a pushbutton is pressed, whereupon it loads the final word into the A register, completing the execution of the CAD variant.

The 7:3 field is the Y coordinate of the point to be plotted, in the range of 000 to 999.

The 0:3 field is the X coordinate of the point to be plotted, in the range of 000 to 999.

The 4:2 field and all but the 1-bit of the 2:1 field are ignored, and are often used to "tag" display points for programmatic purposes.

The second instruction extension is a variant of Store A (STA), specified by the 4:2 variant field value of 20. When this variant is used, the STA instruction becomes the **Store Display (STD)** instruction. Its only effect is to store the value in the A register while **or**-ing in the 1-bit of the 2:1 (continue) digit. (This instruction is seldom needed, since it is usually trivial for the programmer to ensure that the continue bit is already on.)

The Light Pen

The light pen is a handheld device with a lens that can be pointed at the CRT display and a "trigger" which enables it to detect the next displayed dot which it can "see." When a detection occurs, the display word for the detected point is loaded into the A register with its sign forced to 0, and the Display instruction terminates after incrementing the P register. This causes the instruction immediately following the Display instruction to be skipped, which allows the program to know that the light pen (or a pushbutton, as we shall see) has detected the point contained in the A register.

Pressing "L" while in View mode enables the light pen in B220SIM. The simulated light pen's position is indicated by a crosshair cursor which is moved by a joystick or paddles. The light pen trigger is the Open-Apple key, which enables the simulated light pen to detect the next point displayed within two pixels of the cursor position.

The Pushbuttons

When one of the ten display pushbuttons, labeled 0 to 9, is pressed while a Display instruction is executing, the current display word is loaded into the A register with its sign forced to the digit labelling the pushbutton and the instruction terminates after incrementing the P register, causing the next sequential instruction to be skipped. Since the button press is asynchronous with the displayed points, the actual point in the A register is usually irrelevant except for the sign, which indicates which button was pushed. Note that pushbutton 0 is indistinguishable from a light pen detection.

In B220SIM, a pushbutton press is simulated in View mode by pressing a digit key with Open-Apple held down..

The Keyboards

The first keyboard is a custom keyboard much like a conventional computer keyboard, plus three special keys. The second keyboard is the Function Key keyboard containing 32 keys that can be assigned programmed meanings. They are presented to the B220 as a pseudo “paper tape reader” which performs a Reset/Transfer (interrupt) operation on each key press.

The B220 Reset/Transfer is normally initiated by pressing a console switch. Its intended purpose was to allow an operator to cause a program to recover from an error halt or to interrupt a running program to signal some configuration change, such as the mounting of a magnetic tape, or the completion of some manual error recovery operation. It was unused by any program run at Caltech, and so it was simply appropriated by the keyboard designers as a keyboard interrupt.

The keyboard Reset/Transfer function occurs after the program counter has been incremented past the current instruction but before the instruction’s execution has completed. The incremented P register is stored in the 0:4 (address) field of memory location 0000, and the 0:4 field of the C register is stored in the 6:4 field of location 0000, then control is passed to location 0001. The P register value (minus 1) is used to restart the interrupted instruction and the value of the C register address field can be used to determine the progress of certain magnetic tape operations (though it is seldom used).

The keyboard is used by putting its service routine (or a branch to its service routine) at location 0001, so that it is entered on each key press. The code word for the key pressed is read inside the service routine by a special Paper Tape Read (PRD) operation, with the unit number = 7, length field = 01, and variant digit = 4. Thus the instruction to read the key code is: s 7014 03 aaaa, where s is the sign digit and aaaa is the address field.

The key code word is all zeroes except for the 9:2 field, which is two octal digits, and the 0:1 field, whose 2-bit is set if the Upper Case shift is pressed, and whose 1-bit is set if the key pressed was on the Function Key keyboard. The “conventional” keyboard supports upper and lower case letters and a lot of special characters which have no equivalents in the B220’s 2-digit character code, so many of the key codes have no translation to B220 internal codes, and all lower case letters are usually “folded” into their upper case equivalents. The key codes actually used are shown in the table on the next page.

CRT Keyboard Octal Code Mapping

Octal	F Key	LC B220	LC ASCII	LC Char	UC B220	UC ASCII	UC Char	Octal	F Key	LC B220	LC ASCII	LC Char	UC B220	UC ASCII	UC Char
0	**Attn**	99	12	**Rstrt**	99	03	**Cold Rstrt**	40	32	20	2D	**-**	35		
1	1	81	31	**1**	33	3D	**=**	41		51	6A	**j**	51	4A	**J**
2	2	82	32	**2**	35			42		52	6B	**k**	52	4B	**K**
3	3	83	33	**3**	35			43		53	6C	**l**	53	4C	**L**
4	4	84	34	**4**	35			44		54	6D	**m**	54	4D	**M**
5	5	85	35	**5**	35			45		55	6E	**n**	55	4E	**N**
6	6	86	36	**6**	34	27	**'**	46		56	6F	**o**	56	4F	**O**
7	7	87	37	**7**	35			47		57	70	**p**	57	50	**P**
10	8	88	38	**8**	14	2A	*****	50		58	71	**q**	58	51	**Q**
11	9	89	39	**9**	10	28	**(**	51		59	72	**r**	59	52	**R**
12	10	80	30	**0**	4	29	**)**	52		99			99		
13	11	35			35			53		27	24	**$**	35		
14	12	35			35			54		99			99		
15	13	99			99			55		99			99		
16	14	99			99			56		99			99		
17	15	35			35			57		99			99		
20	16	35			13	2B	**+**	60		0	20	**space**	0	20	**space**
21	17	41	61	**a**	41	41	**A**	61		21	2F	**/**	32	3F	**?**
22	18	42	62	**b**	42	42	**B**	62		62	73	**s**	62	53	**S**
23	19	43	63	**c**	43	43	**C**	63		63	74	**t**	63	54	**T**
24	20	44	64	**d**	44	44	**D**	64		64	75	**u**	64	55	**U**
25	21	45	65	**e**	45	45	**E**	65		65	76	**v**	65	56	**V**
26	22	46	66	**f**	46	46	**F**	66		66	77	**w**	66	57	**W**
27	23	47	67	**g**	47	47	**G**	67		67	78	**x**	67	58	**X**
30	24	48	68	**h**	48	48	**H**	70		68	79	**y**	68	59	**Y**
31	25	49	69	**i**	49	49	**I**	71		69	7A	**z**	69	5A	**Z**
32	26	99			99			72		99			99		
33	27	3	2E	**.**	35			73		23	2C	**,**	35		
34	28	99			99			74		1			1		
35	29	99			99			75		39	8	**right arrow**	39	8	**right arrow**
36	30	99			99			76		40	15	**left arrow**	40	15	**left arrow**
37	31	99			99			77		15	5E	**enter**	15	5E	**enter**

Note that this table is the result of reverse-engineering a B220 program that used the CRT keyboard, so it does not show the characters for all of the keys that were on the keyboard. B220 codes of 35 or 99 indicate that these keys are invalid for the program. If the Function Key bit is set, the first 32 octal codes correspond to the 32 keys on the function keypad. The Shift key can also be used as a modifier for Function Keys. There are three keys with special functions and no internal representations: ctl-R (Restart), ctl-@ (Attention), and ctl-C (Cold Restart). These functions may be implemented by the B220 program and are usually processed in the Reset-Transfer trap routine.

B220SIM's simulation of the CRT keyboards is enabled by pressing "K" while in View mode. Keyboard mode is ended by typing ESC, which returns to normal View mode. Function keys (unshifted) are simulated in keyboard mode by holding Open-Apple while pressing "1", "2", "3" … "0" and "a", "b", "c" … "v" for function keys 1 through 32 respectively.

VIEW Mode Implementation Note

The Digital Equipment Corporation type 340 display was a circular CRT display that provided a display area of approximately 10"x10". Under B220 software control, it repeatedly displayed single points on a coordinate grid of 1000x1000 at a rate of about 8,000 points per second. Multiple displayed points were visible because of the persistence of the phosphor and the persistence of human vision.

In contrast, the Apple high-resolution graphics system is raster-based, with a resolution of 280x192 points, and the raster is automatically refreshed at a rate of 60 fields per second.

B220SIM maps the DEC display to a 256x192 subset of the Apple HGR display by dividing the DEC coordinates by 5 (default), but allows zooming from 5:1 to 1:1, and panning both horizontally and vertically to view any portion of the DEC display in detail.

To simulate the dynamic nature of the DEC display, B220SIM erases points after 256 new points have been plotted. This is only an approximation of the effect of phosphor persistence, and can, in some cases, result in parts of the display being erased almost immediately after they are redrawn. To overcome this simulation artifact, the erasing of points can be temporarily defeated by holding down the Closed-Apple key. Of course, this causes the path of any moving points to be accumulated, so it is most useful with essentially static displays.

Below is a screenshot taken in View Mode while running ViewKbdTst.obj and after entering Keyboard Mode ("K") and pressing lower case "d". The 240*D* is displayed by a call to CCONV (loaded as CCONV.obj). 240 is the octal character code for lower case d, and the translated B220 code ("D") is shown between asterisks.