RCA

Programming in CHIP-8

H. Kleinberg

PE-721

Programming in CHIP-8

H. Kleinberg

Learning programming on RCA's VIP is relatively easy and painless, and CHIP-8 is one of the major reasons.

This paper introduces one of the most powerful parts of the COSMAC VIP—the CHIP-8 interpreter. The paper is written primarily for those who have not been exposed to a programming language in any depth, but who are familiar with numbering systems and with such basic computer concepts as memory and instructions.

If you don't have that background, you should still be able to get a good idea of what kind of functions are available to the CHIP-8 user. This paper by itself, however, will probably not prepare you to write a program without a bit more coaching. [Try the "Good Guide Special" described in this paper's final section, "Taking the next step."—Eds.]

Descriptive information about the VIP and its accessories is readily available and most of the information appearing in the instruction manual will not be repeated here. But it is worthwhile to review briefly how the system's data structure is organized. The COSMAC VIP handles data in "bytes," each consisting of 8 bits. A byte can, therefore, take on any one of $2^8 = 256$ configurations. For example, 01101101 is equivalent to decimal 109. For ease of handling, this awkward (to people) entity is treated as though it were broken up into 2 subgroups of 4 bits each, although no such fracture occurs in reality. Each subgroup can now assume $2^4 = 16$ configurations, hence the name "hexadecimal," shortened to "hex." These 16 binary configurations, when expressed in hex, are labeled from 0 through 9 (borrowed from our decimal system) and A through F (equivalent to decimal 10 through 15). Our earlier example of decimal 109 now becomes 0110 1101, or 6D in hex, and the byte has now been expressed as two hex digits.

Using the hex system is an acquired skill, and cumbersome in the early learning stages, but it is far easier to cope with than the pure binary which it masks. In all that follows, hex digits are used, so your counting and arithmetic must be based on that system, a fact of which you will be reminded periodically.

You have now been relieved of the burden of coping with long binary strings, but you have not been taken out of the level of machine instructions, where you must concern yourself with the many specific details of the computer's structure. To alleviate that chore, a program called CHIP-8 has been written as part of the VIP system package. CHIP-8 is a special type of program called an interpreter.

Why use an interpreter?

An interpreter provides you with a simplified language that is tailored to a specific application.

In the spectrum of software, interpreters lie between assembly-level programs on the one hand, and the more elaborate and generalized compilers such as Fortran on the other.

The designer of an interpreter starts by picking a set of functions that would be useful in his specific application. For each function he writes a machine-language program, which is later activated when the user calls out the appropriate code in his "higher-level" program. Some of the functions can be quite complex when worked out in machine code.

As with any other product design, an interpreter is a result of certain compromises. There are conflicts between the range of application (flexibility) of the language, the speed of execution, the amount of memory required to store it, the complexity as seen by the user, and so on. In the case of CHIP-8, all these questions were settled in favor of simplicity and economy.

Harry Kleinberg, formerly with RCA's Computer Division, is now Manager of Corporate Standards Engineering. He can't seem to get the 1s and 0s out of his blood, though; besides this article, he recently wrote a book titled *How You Can Learn to Live with Computers.*

Contact him at:
Corporate Standards Engineering
Cherry Hill, N.J.
Ext. 6616

Author Kleinberg with his computer system: the VIP (right), tv display, and cassette recorder (left).

It is important to keep in mind the distinction between the interpreter and the computer on which it is run. By its nature, an interpreter language is designed for a specific application, sacrificing features that would be useful in other, different applications. For example, CHIP-8 is specifically designed for controlling the video display and hex keyboard of the VIP, but is a very poor language for numerical computation. The VIP's 1802 microprocessor, however, does not have that restriction. Using the same 1802, we could write another interpreter that would be excellent for vector calculations and very poor for video games. The interpreter reflects conscious restrictive choices that in no way alter the generality of the computer on which the interpreter is run.

The basics of CHIP-8

CHIP-8 instructions have a simple, consistent format.

Each instruction is 2 bytes (4 hex digits) long, and each performs a distinct, well defined function. The general format is ABBB, where A is always part of the instruction code, and B is either part of the code or data that you must supply. Fig. 1 lists the instructions in numerical order. (Remember that A,B....F are numbers, not letters.)

In order to understand the instructions, you must first become familiar with a vocabulary of 1 word and 7 symbols.

VX and **VY**, or **X** and **Y**, stand for the program *variables*. A key feature of CHIP-8 is the use of 16 variables, which are actually memory locations containing numbers over which you have complete control. They can signify whatever you want them to signify. You can set them to any value you choose, you can compare them, you can increment them. Each variable is identified by one of the hex digits, 0 through F, but in Fig. 1 and in the descriptive material where the general form of the instructions is used, they are referred to as VX, VY, or simply X or Y, where X and Y, of course, range from 0 to F. Thus, you will be performing such operations as "check if variable 7 (V7) is equal to zero" or "add 1C to the present value of variable B (VB)." Each variable represents one byte of data and its value is, like all other data, expressed in a program as a pair of hex digits.

Only a few other symbols are used in describing the CHIP-8 instructions:

I, called a pointer in the manual, is a memory address. In general, it identifies the beginning of a string of data to be read from or written into the memory. The pointer is your major tool for storing and retrieving data, and it is important to note which instructions use it or modify it.

MI refers to the data stored at the location(s) addressed by I. In a sense, I is the post-office box, MI is the letter stuffed in that box.

MMM indicates memory addresses, other than I, that are to be supplied by the programmer. Since CHIP-8 was designed to run with a small memory, the most significant hex digit in the address will always be 0, i.e. 0MMM.

KK represents a 1-byte value or number that you will supply in your program.

Instruction	*Operation*
***00E0**	Erase display (all 0's)
***00EE**	Return from subroutine
0MMM	Do machine-language subroutine at 0MMM (subroutine must end with D4 byte)
***1**MMM	Go to 0MMM
***2**MMM	Do subroutine at 0MMM (must end with 00EE)
***3**XKK	Skip next instruction if VX = KK
***4**XKK	Skip next instruction if VX ≠ KK
***5**XY**0**	Skip next instruction if VX = VY
***6**XKK	Let VX = KK
***7**XKK	Let VX = VX + KK
***8**XY**0**	Let VX = VY
8XY**1**	Let VX = VX OR VY (VF changed)
8XY**2**	Let VX = VX AND VY (VF changed)
8XY**4**	Let VX = VX + VY (VF = 00 if VX + VY ≤ FF, VF = 01 if VX + VY > FF)
8XY**5**	Let VX = VX – VY (VF = 00 if VX < VY, VF = 01 if VX ≥ VY)
***9**XY**0**	Skip next instruction if VX ≠ VY
***A**MMM	Let I = 0MMM
BMMM	Go to 0MMM + V0
***C**XKK	Let VX = random byte (KK = mask)
***D**XYN	Show n-byte MI pattern at VX, VY coordinates. I unchanged. MI pattern is combined with existing display via EXCLUSIVE-OR function. VF=01 if a 1 in MI pattern matches 1 in existing display.
EX9E	Skip next instruction if VX = hex key (LSD)
EXA1	Skip next instruction if VX ≠ hex key (LSD)
***FX07**	Let VX = current timer value
***FX0A**	Let VX = hex key digit (waits for any key pressed)
***FX15**	Set timer = VX (01 = 1/60 second)
***FX18**	Set tone duration = VX (01 = 1/60 second)
FX1E	Let I = I + VX
***FX29**	Let I = 5-byte display pattern for LSD of VX
***FX33**	Let MI = 3-decimal digit equivalent of VX (I unchanged)
FX55	Let MI = V0:VX (I = I + X + 1)
FX65	Let V0: VX = MI (I = I + X + 1)

Fig. 1
CHIP-8 vocabulary includes 31 instructions. Ones with asterisks are used more commonly than others and so are explained in text. Parts of instructions in **bold type** identify the instruction; the rest of the instruction message tells what variables, memory locations, etc., are to be acted upon. Programs like CHIP-8 are dynamic systems, with new features being continually introduced. If it is important that your information is current, make sure you have the latest copy of the Manual.

N refers to the number of bytes to be used in setting up a pattern to be displayed.

How to read a CHIP-8 program

A sample working program shows how CHIP-8 works.

Fig. 2 lists a sample program, called "Jumping X and O." Here is what the program is written to do. First, a solid 6x6-spot block appears in the upper right quadrant of the tv display. A 5x5 "X" pattern appears in the center and jumps randomly to a new location every 1/5 second. When the X overlaps the 6x6 block, the X disappears, an "O" pattern appears in the center of the screen, and repeats the process, being replaced by the X when an overlap with the block

Address	Instruction code	Comments
0200	**A24C**	Set I to block pattern
0202	**6530**	V5 = 30 } Coordinates of block
0204	**6604**	V6 = 04 }
0206	**D566**	Show block at V5, V6
0208	**A240**	Set I to X pattern
020A	**2212**	Do subroutine at 0212
020C	**A246**	Set I to O pattern
020E	**2212**	Do subroutine at 0212
0210	**1208**	Go to 0208 (return to X pattern)
0212	**611E**	} Set V1, V2 to center coordinates
0214	**620D**	}
0216	**D125**	Show the pattern
0218	**630C**	Set V3 to 0C (= 1/5 second)
021A	**F315**	Set timer from V3
021C	**F407**	Timer → V4
021E	**3400**	Skip if V4 (timer) = 0
0220	**121C**	Return to 021C if V4 ≠ 0
0222	**4F01**	Skip if VF = 01 (checking for overlap)
0224	**122E**	Go to 022E if VF ≠ 01 (overlap, switch patterns)
0226	**D125**	If no overlap, show old pattern to erase
0228	**C13F**	Random number (6-bit) to V1 } new coordinates
022A	**C21F**	Random number (5-bit) to V2 }
022C	**1216**	Go back to 0216 to show pattern in new location
022E	**D125**	Show old pattern to erase
0230	**00EE**	Return from subroutine (will switch patterns)
0232	**0000**	Space for future changes
0234		
0236		
0238		
023A		
023C		
023E	**0000**	Space for future changes
0240	**8850**	
0242	**2050**	} X pattern
0244	**8800**	}
0246	**F888**	
0248	**8888**	} O pattern
024A	**F800**	}
024C	**FCFC**	
024E	**FCFC**	} Block pattern
0250	**FCFC**	}
0252		
0254		

Fig. 2
"Jumping X and O" program was designed to explain a number of CHIP-8 instructions and entertain *RCA Engineer* editors. Instruction explanations in text use specific lines of the program listing as examples. See Fig. 3 for the program in action.

occurs. The program continues until the machine is stopped; Fig. 3 shows the program in action.

This sample program will be used to illustrate the VIP instruction set, but before proceeding to those details, it would be worthwhile to look at the format of the sample to note some of the conventions that are used.

The first column in the program contains the memory addresses in which the instructions are stored. All instructions occupy two adjacent byte locations, so it is standard practice to write them as complete 4-digit words, using only even-numbered memory locations to identify them. It is understood that the adjacent odd-numbered locations are used for the second byte of each word.

The memory column does not, of course, get entered into the computer—it tells "where," not "what." But still you must keep track of these addresses and list them for each instruction. As you will see, all references made in the program to other parts of the program use these addresses to find the right place. And in using these addresses, *remember hex.*

Note that the first instruction of the sample program is stored in location 0200. This is a mandatory requirement, since CHIP-8 was designed with the assumption that the program starts there. Don't try to surprise the interpreter on this point—it will always have the last laugh.

The next column in Fig. 2 contains the actual 2-byte instructions (described in the next sections). These two bytes are the only part of the program that actually goes into the machine.

The last column contains your comments on each instruction—a memo to yourself and to whoever else studies your program. Each comment should be a brief description, either in mathematical or English form, of what the instruction does. The comment is of no value to the computer and plays no part in executing the program, so you can easily leave it out if you are willing to accept that at some later date you will not be able to understand what your own program is all about. The value of good comments cannot be overstated. Every nontrivial program, without exception, will eventually be revised by the writer or by someone else and, while using good comments will not guarantee easy revision, their lack will guarantee that a simple change becomes a titanic struggle. You will eventually develop your own natural shorthand for comments, but in the beginning, be generous with them.

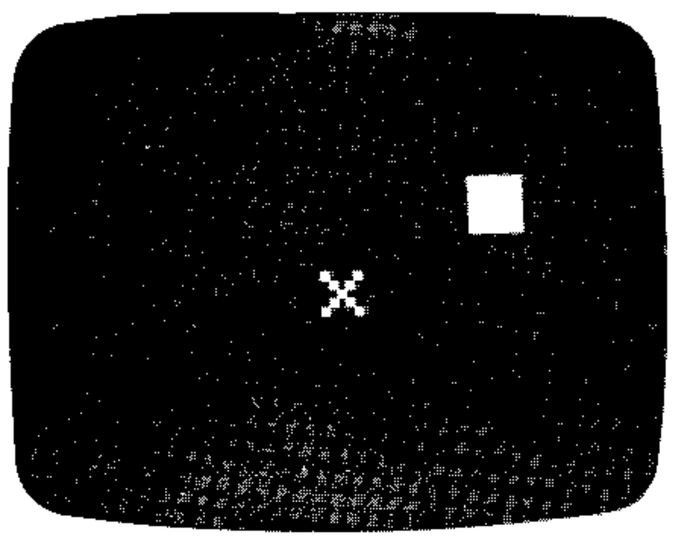
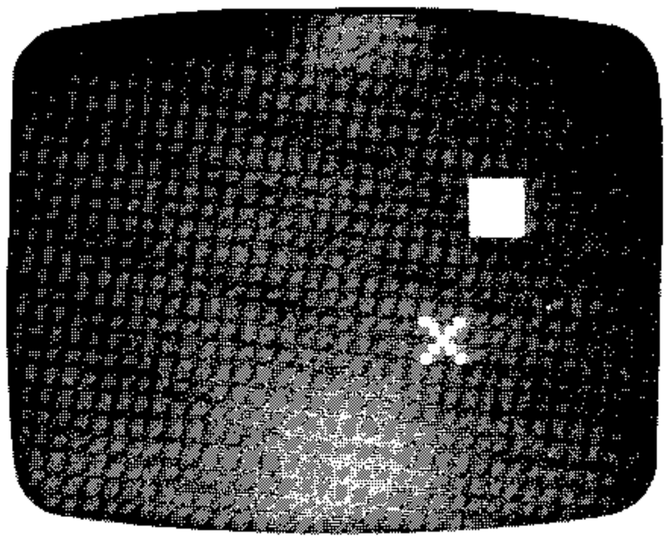
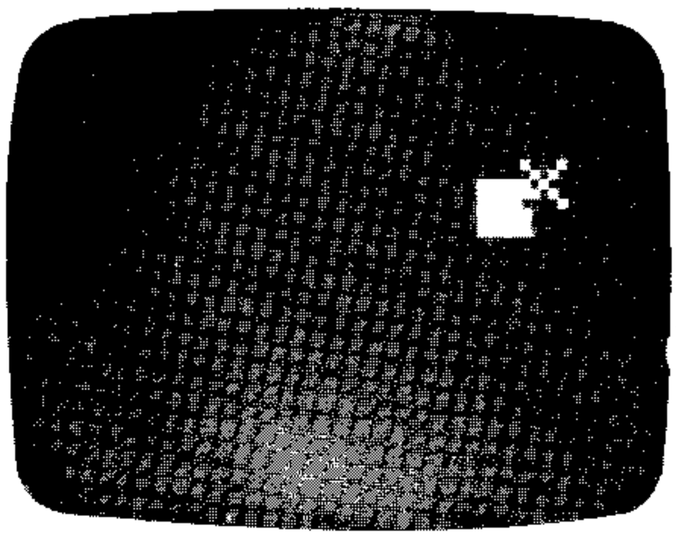


Fig. 3
Program starts with an X in the center of the display and a block at the upper right corner.



The X jumps to a new (random) location every 1/5 second.



When the X and the block overlap,

What the instructions mean

CHIP-8 has 31 instructions, but you can write most programs using only 20.

We will concern ourselves with the 20 instructions in Fig. 1 that are marked with asterisks. The remaining 11 are no less legitimate, but tend to be more valuable to the advanced VIP user. Of the 20 sample programs in the VIP manual, all but one or two use only the group of instructions we will examine, so there is no doubt that many interesting programs can be written using them.

Note that the instructions, except the first two, have one, two, or three numbers for you to provide—these are the X, Y, I, K, and M discussed in the "vocabulary" section. It is important to write all the remaining digits—identified by **bold face** type in the following explanation—exactly as they appear in the general form, since the interpreter uses them to identify the operation.

For the sake of cohesiveness, the instructions that we will look at have been arbitrarily divided into six groups:

Group 1 — manipulating the variables

6XKK makes KK become the new value of variable X. As it is the first instruction considered, a brief look at its structure is in order. The "6" is the code that the interpreter will use to translate this operation into the proper set of basic machine instructions. "X" is the identifying number of the variable that you have selected, and "KK" is the hex value to which you want to set variable X. Thus, 6530 (line 0202 in Fig. 2) changes the value of variable 5 to the value 30.

7XKK has the same structure, but the "7" translates to an addition. The instruction adds KK (remember hex) to the present value of VX. For example, 7A1B increases the current value of variable A by 1B.*

8XY**0** Note that in this case the first and last digits are both part of the instruction code. By using this instruction, the value of variable Y also becomes the value of variable X. For example, 8320 copies the value of variable 2 into variable 3.

*This same instruction can be used for decrementing VX by taking advantage of the fact that the variable is a 2-digit number with no carry into a third digit. In such a system (to use a decimal example) you can subtract 1 by adding 99, since n + 99 = n + 100 - 1 and when you throw away the 100 (the 3rd digit) you have n - 1. It is analagous to a 2-digit odometer in an automobile; if you advance it 99 places it brings you back to 1 short of your starting point. In the hex system you add FF (the biggest number) in place of 99 to subtract 1, but the principle remains the same and can be extended to subtract any number you wish.

Group 2—transfer of control

The computer normally steps through a program in sequence. It sets up an instruction from memory, executes it, and then proceeds to the instruction in the next memory location. But the ability to depart from this sequence is one of the essential features of any computer, and CHIP-8 provides a flexible system for doing so.

Breaking out of sequence may be done in two ways—unconditionally (the jump is made whenever the instruction is encountered) or conditionally (the sequence is or is not broken, depending upon whether some condition has been satisfied). This group of instructions deals with the unconditional transfers, often called "go to," or "jump."

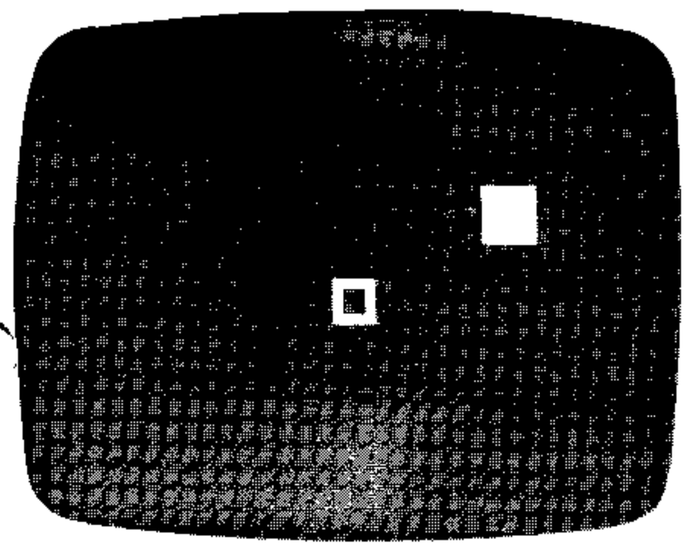
1MMM has the computer perform, not the next instruction in memory, but the one stored at location 0MMM. As an example, see Fig. 2, line 0210. Normally, the next instruction to be done would be 0212 but instead, the 1208 instruction makes the computer go back to do the one at memory location 0208. Jumps may be either forward or backward in the program, and there is no restriction on how many places may be skipped.

An unusual use of this instruction is for stopping the machine. When a program comes to a point demanding some kind of reset or fresh start, a 1MMM instruction referring to its own memory location will put the VIP into an endless cycle until the operator takes the appropriate action.

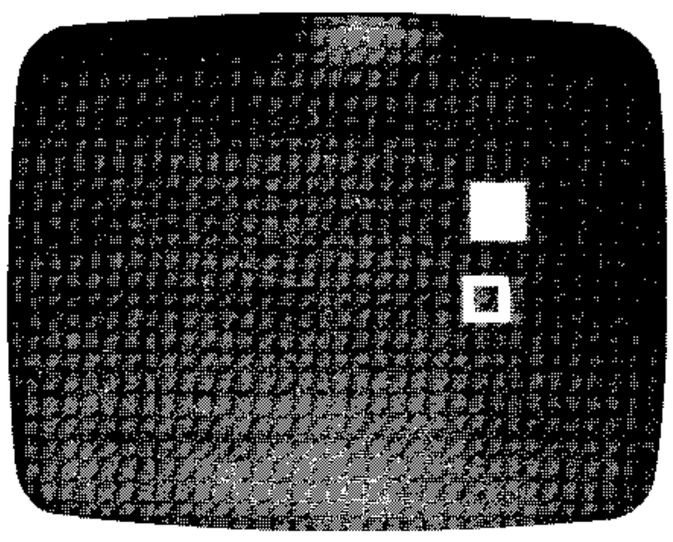
2MMM has the computer do the subroutine at 0MMM, and **00EE** has it return from the subroutine.

These two instructions must be considered together. While the 2MMM instruction also transfers unconditionally, it is not the same as the previous case. The difference lies in the nature of a subroutine, which is a package of instructions performing some function that will be used at more than one point during the course of a program. (For example, computing sin x or the roots of a quadratic equation might be subroutines in a scientific program.) When a subroutine is finished, you want the program to return to the point from which the jump was made and pick up its normal sequence. Since the subroutine may be entered (called) from any place in the program, the obvious question is "How do I know which place is the right returning point?"

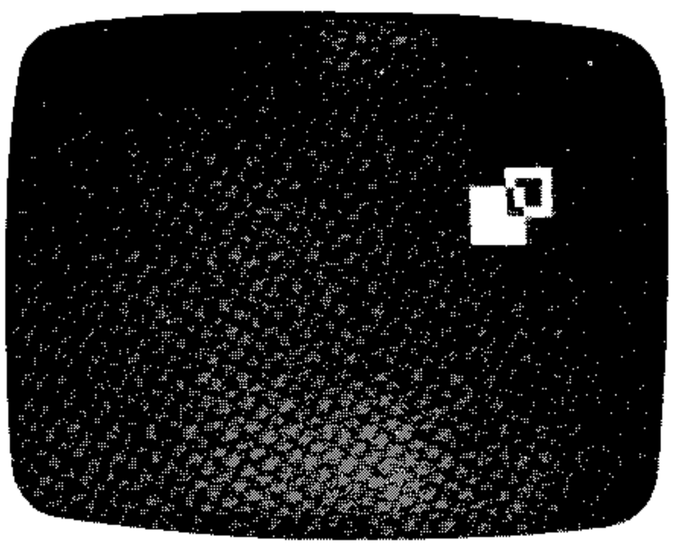
In Fig. 2, lines 020A and 020E present a typical case, in that they both have the program jump to the same point, 0212, the beginning of a subroutine that makes the X or O jump



the X disappears and an O appears in the center of the display.



The O then starts jumping to random locations



until it overlaps the block, and the whole process begins over again.

every 1/5 second. Clearly, before this program transfer is made, the computer must store the location of the "jumping-off" point so that at the end of the subroutine (line 0230) it can correctly return to 020C or 0210 as appropriate. The 00EE—return from subroutine—instruction makes the proper transfer back by retrieving the stored "jumping-off" address.

CHIP-8 provides enough storage to "nest" up to 12 subroutines. This means that, in the course of executing one subroutine, another may be called and executed, and so on. The situation is analogous to the use of nested parentheses in an algebraic statement, and the same care must be taken so that each unit is begun and terminated properly with the 2MMM and 00EE instructions.

Group 3—conditional skips

This group of instructions alters the sequence of events in a program depending on whether or not some stated condition has been met. They give the computer its most powerful function, the ability to follow one course of action or another, depending on the value of data in the machine.

In these CHIP-8 instructions, if the stated condition *is* met, then the computer skips the next instruction in line and executes the following one. If the stated condition is *not* met, then the computer continues in its normal sequence, i.e., it does not skip. In a program, it looks like this:

Line n states the condition to be met.
Line n+1 is executed if the condition is *not* met.
Line n+2 is executed if the condition *is* met.

In Fig. 2, lines 021E through 0226 show examples. Note that when the condition is not met, there is only space for one instruction (line n+1). In most cases that instruction will have to be an unconditional jump to the place where the relevant code has been stored.

The instructions in this group are:

3XKK Skip next instruction if VX=KK
4XKK Skip next instruction if VK ≠ KK
5XY**0** Skip next instruction if VX = VY
9XY**0** Skip next instruction if VX ≠ VY

Group 4—memory control

AMMM sets the pointer, I, to the memory address 0MMM. All following instructions that use memory will go to this address until you do something that changes I. See Fig. 2, lines 0200 and 0208, for an example.

FX29 sets up the pointer for displaying hex digits. Part of the VIP is an "operating system," which takes care of loading memory from the keyboard, reading to and from the tape cassette, etc. The bit patterns for displaying any of the hex digits are already stored in memory as part of this operating system. The FX29 instruction provides access to those patterns, so that you do not have to work them out for yourself. It sets I to the correct address of the 5-byte pattern for the hex digit in the least-significant digit of VX.

FX33 translates the value of VX into its 3-digit decimal equivalent and stores the result in the memory starting at I. This instruction is very useful for presenting video-game scores in decimal form.

Group 5—miscellaneous control

The VIP has a timer and an audible tone generator. The timer, which automatically counts down to zero when it is set by the programmer, is useful for such things as making a display flash, timing a permissible period for some keyboard action to take place, etc. The audible tone, whose duration you can set, can be used to celebrate collisions, end of program, etc.

FX15 sets the timer to the value of VX. The smallest value, (01 in VX) is equivalent to 1/60 second, so the maximum time (FF in VX) is 255x1/60 = 4.3 seconds.

FX07 transfers the current timer value into VX. This instruction lets a program monitor the progress of a countdown. Fig. 2, lines 021A through 0220, demonstrates a typical timing loop. V3, which stores the hex number equivalent to 1/5 second, is transferred to the timer (line 021A), and from there to V4 (line 021C), where it is tested (lines 021E, 0220) until it is found to equal zero.

FX18 turns on the audible tone for the length of time specified by the value of VX. Again, the smallest increment is 1/60 second.

Caution—In the above three instructions, since 1/60 is a decimal number, you will be going back and forth between the hex and decimal systems. Watch your arithmetic.

CXKK sets a random byte into VX. KK acts as "mask"—where KK has a 0 bit, no entry is made into VX. The mask is a way of controlling the range of the random number; for example, C703 would set into V7 a 2-bit random number (since 03 has two bits) with equal probability of any $2^2=4$ values. In Fig. 2, lines 0228 and 022A provide further examples.

Group 6—input and output

FX0A sets the least-significant digit of VX to the value of the next key pressed. The program will pause here until some key is pressed.

00E0 erases the entire display.

DXYN displays a pattern on the screen. Most of your attention in programming the VIP will center around the display. For full information, you should refer to the *VIP Instruction Manual,* since we can concern ourselves here only with those characteristics that reflect into this CHIP-8 instruction.

In this instruction, N defines how many bytes make up the patterns you want to display, X and Y describe where on the screen you want it to appear and, while I does not appear in the code, its current value determines where the computer will find the first byte to be shown. Therefore, before this instruction can be used you must have taken care of the

following functions: 1) the bit pattern to be displayed must have been set up in the memory; 2) I must have been set (be pointing) to the address of the first byte in the pattern; and 3) the two variables defining the pattern's screen location must have been set to the proper values, X for horizontal (increasing left to right), Y for vertical (increasing top to bottom).

For example, in Fig. 2, line 0206, D566 means "show a 6-byte pattern taken from the memory starting with I." The upper left-hand corner of this pattern will be at coordinates specified by V5 and V6. Note that the three preceding instructions have set the values of the coordinates (lines 0202 and 0204) and have set I to 024C (line 0200). At location 024C, the beginning of the pattern has been stored, so everything is all set to go.

In the VIP, the pattern that is being displayed is superimposed over any existing pattern on the display, with the feature that where a new spot overlays an existing spot, the screen is erased. When this happens, the computer puts 01 into variable F, making possible the programmed detection of overlaps or collisions between the old and new patterns. Line 0222, in Fig. 2 for example, senses VF to see whether the jumping pattern has overlapped the fixed block. If it hasn't (VF=0), line 0226 starts the procedure to move the X or O randomly again. If there has been an overlap (VF=1), line 0224 causes the program to switch from X to O, or vice versa, and start over. You can also use the "blankout on overlap" feature in selectively erasing all or part of a pattern by rewriting it in the same place.

Writing a program

Familiarity with the CHIP-8 instructions does not, by itself, make you a programmer, any more than learning the alphabet made you literate.

Learning the instructions is a necessary first step which, with the exercise of some practice and patience, can lead to whatever level of proficiency you may want to achieve in writing programs. In fact, one of the most important uses of the VIP is in providing a fun way of acquiring exactly that skill.

While the actual design of a program is something you must do for yourself, a few paperwork tools and helpful hints can make the job easier and more organized. Here are some suggestions to help you get started.

Many people find it useful to make a flowchart of their programs before they start coding. It's a good way of organizing your thoughts, and it lets you test the program on paper before you have invested a lot of time in coding. It is also very helpful in identifying those parts of the program that could be done as subroutines, and so reducing the amount of coding you have to do. But with or without a flowchart, think about what you want to do before you plunge ahead; a half hour of planning will more than pay for itself in easier coding.

Keep a list of the 16 variables and what meanings you have assigned to them, i.e., what they represent in your program. Note that VO and VF are occasionally set aside for special use. Make sure they're clear and available when the instruction will be using them.

Be sure to keep track of the memory addresses in which your program is stored. For the first rough drafts you might want to leave spaces every so often in the program to allow for changes. These spaces can later be bridged by unconditional jumps. If you decide to close up the gaps later, or if you find you have to squeeze in another line of code, make sure that you have properly modified all the "jump to" addresses that have been affected. Another trick is to label "jump" points with arbitrary designations (e.g., Greek letters) until you are satisfied that you have progressed to the point of specifying accurate addresses.

Make a chart of the display to help you plan the program, work out the bit patterns, and establish the coordinates. It should also carry the memory addresses in which the data to be displayed is stored.

If you have occasion to put data, such as display patterns, in the memory, keep a list of where you have put them. You may never find them again otherwise.

Don't forget that CHIP-8 expects to find your program starting in location 0200. If for some reason you want to violate that rule, then 0200 must have an unconditional jump to whatever starting point you have selected.

Be generous with your comments. It's surprising how difficult it becomes, even after a short coffee break, to remember or reconstruct the brilliant trick you thought up while you were deeply engrossed in your creative approach to the problem.

Don't forget to return properly from a subroutine, have fun, and *remember hex.*

Acknowledgments

Ann Merriam, J.W. Wentworth.

References

1. COSMAC VIP instruction Manual, RCA Solid State Division, Somerville, N.J.
2. Microcomputer Fundamentals (CEE Course C-51) RCA Corporate Engineering Education, Cherry Hill, N.J. See Study Guide, Chapter 8.

Final manuscript received December 7, 1977.