PROGRAMS
FOR THE
COSMAC ELF
INTERPRETERS

PAUL C. MOEWS
# Programs for the Cosmac Elf Interpreters

Paul C. Moews

## List of Sections

1. Introduction .......................... 3
2. A Demonstration Interpreter .................. 4
3. The CHIP-8 Language .................. 9
4. Hardware Differences between 1802 Computers .................. 13
5. A Complete ELF CHIP-8 Interpreter ................. 13
6. Extending the CHIP-8 Instruction Set .................. 22
7. Appendix ................................ 28

## List of Programs

### Machine Code

1. Demonstration Interpreter .......................... 8
2. Complete CHIP-8 Interpreter .................. 14
3. Additional Skip Instructions .................. 22
4. Multiply, Divide and 16 Bit Display Instructions .................. 23
5. Six Bit ASCII Symbols .......................... 25

### Interpretive Code

1. Addition (Demonstration Interpreter) .......................... 5
2. Subroutine Use (Demonstration Interpreter) .................. 5
3. Addition Problems (Demonstration Interpreter) .................. 6
4. Addition Problems (Full Interpreter) .................. 10
5. Display ASCII Character (Full Interpreter) .................. 26
Introduction

This booklet’s purpose is to explain the construction and operation of an interpreter for the COSMAC 1802 “ELF”. It assumes that the reader has some knowledge of the 1802 instruction set and is able to write simple machine language programs. Mnemonics are not provided because most ELF owners do not have access to assemblers and must work directly in machine language. Instead, programs are explained in a documented, step-by-step fashion, that it is hoped will make the concepts involved easy to follow.

The interpretive language described is “CHIP-8”, the language used by RCA Corporation in its “COSMAC VIP” computer. CHIP-8 is a simple language consisting of about 30 instructions. RCA’s interpreter is elegant and well thought out; once understood it is easily changed and modified.

This booklet contains five sections; in the first section a simple demonstration interpreter is introduced. This demonstration interpreter runs in the basic 1/4 “ELF” and its instructions are a subset of the full CHIP-8 instruction set. While simple, the demonstration interpreter employs methods similar to those in the full interpreter.

Further sections discuss the full CHIP-8 instruction set, hardware differences between the “VIP” and the “ELF”, and provide a listing of a complete ELF interpreter together with suggestions for implementing it on various machines. The final section discusses the extension of the CHIP-8 instruction set. Examples are provided for multiply and divide instructions together with an instruction which displays characters for the 64 six bit ASCII symbols.

I should like to thank RCA Corporation for permission to write about CHIP-8 and to modify it for the ELF. However RCA is not responsible for any of the material in this booklet. The programs described here have been thoroughly tested on a number of versions of the COSMAC “ELF” as described in Popular Electronics articles and are believed to be reliable but there is, of course, still the possibility that they contain unexpected errors. This kind of interpreter is rather hardware dependent and changes in input/output lines or in the use of flag lines will cause failures. An attempt was made to provide sufficient documentation so that the user can make the changes necessary to implement CHIP-8 on a variety of machines.
A Demonstration Interpreter

The surprising power of computers is due to the development of languages which organize programming into different levels of complexity. Perhaps the simplest way to organize programming with a language is to use an interpreter. One can consider an interpreter to be a program that converts the basic instruction set to a new language, a set of instructions that better suits the programmer. Alternately an interpreter can be thought of as a program with a control section and a number of subroutines, the new language now instructs the interpreter as to which subroutines to call and in which order. The subroutines perform "tasks" which are more complicated than those performed by a single machine code operation. The ubiquitous basic interpreter is a good example.

RCA's CHIP-8 language is an interpretive one and it converts the 94 machine language instructions of the 1802 microprocessor to a new set of about 30 more powerful and convenient instructions. Each type of statement in the new language is implemented by a machine code subroutine which carries out the desired operation. It differs from a basic interpreter in that most of the operations carried out by the subroutines are small ones, consisting of only a few machine code instructions, and the language is therefore a simple one without many of the features of basic. However quite powerful programs can be written with a few hundred CHIP-8 instructions.

This section introduces a version of CHIP-8 for the 1/4K Elf. Ten of the instructions are a subset of the full CHIP-8 set and are identical to those in CHIP-8. Two additional instructions, read a byte from the keyboard and display a byte on the hex display, have no exact counterparts in the CHIP-8 set.

CHIP-8 instructions consist of four hex digits. The first hex digit determines the type of instruction; there are therefore 16 basic kinds of CHIP-8 instructions. The next 3 hex digits are used in several different ways. The can be used to specify a memory location, and as there are 3 hex digits available, any memory location from 000 to FFF can be specified. In the demonstration interpreter only the two least significant hex digits are needed for this purpose because it is necessary to address only a single page of memory.

A basic feature of CHIP-8 is that it provides 16 one byte variables, designated V0 through VF. Thus a single hex digit can be used to specify one of these variables. In many pt the CHIP-8 instructions the second most significant hex digit is used for this purpose, leaving the last two hex digits available for other uses. In arithmetic operations the two variables to be added, etc. are specified by the second and third hex digit leaving the last hex digit to designate the type of arithmetic operation to carry out.

Before beginning a discussion of how the interpreter works, it is necessary to have an understanding of the language and its use. The instructions available are shown in Table 1.

Table 1

<table>
<thead>
<tr>
<th>Demonstration Interpreter Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00MM do a machine code subroutine at location MM (The machine code subroutine must end with D4)</td>
</tr>
<tr>
<td>10MM go to MM; control is transferred to location MM in the interpretive code</td>
</tr>
<tr>
<td>20MM do an interpreter subroutine at location MM (The interpreter subroutines must end with 009E)</td>
</tr>
<tr>
<td>4XKK skip if VX≠KK; the next interpreter instruction is skipped over if VX does not equal KK</td>
</tr>
<tr>
<td>6XKK set VX=KK; variable X is made equal to KK</td>
</tr>
<tr>
<td>8XY0 set VX=VY; variable X is made equal to variable Y</td>
</tr>
<tr>
<td>8XY1 set VX=VY or VY; variable X is made equal to the result of VX logically ored against VY (Note that VF is changed)</td>
</tr>
<tr>
<td>8XY2 set VX=VX and VY; variable X is made equal to the result of VX logically anded against VY (Note that VF is changed)</td>
</tr>
<tr>
<td>8XY3 set VX=VX xor VY; variable X is made equal to the result of VX logically xored against VY (note that VF is changed)</td>
</tr>
<tr>
<td>8XY4 set VX=VX+VY; variable X is made equal to the sum of VX and VY (Note that VF becomes 00 if the sum is less than or equal to FF and 01 if the sum is greater than FF)</td>
</tr>
<tr>
<td>8XY5 set VX=VX-VY; variable VX is made equal to the difference between VX and VY (Note that VF becomes 00 if VX is less than VY and 01 if VX is greater than or equal to VY)</td>
</tr>
<tr>
<td>8XY6 set VX equal to VY shifted right 1 bit position, (Note bit 0 is shifted into VF)</td>
</tr>
</tbody>
</table>
| 8XY7 set VX=VY-VX; variable VX is made equal to the difference between VY and VX (Note that VF becomes 00 if the
An easy way to see how these instructions are used is to illustrate them with a simple program. The interpreter is listed at the end of the chapter and can be used to run these sample programs.

To start let's look at the following program. It reads 2 switch bytes, displays them, adds them, and displays the result. If overflow occurs, that is also displayed. The program uses only 10 interpretive instructions (The first instruction 3071 is actually machine code and transfers control on entry to the interpreter; It is not part of the interpretive code.) The interpreter has a program counter for interpretive code (R(5)) which is set on entry to the address of the first instruction (M(0002)). The first interpretive language instruction is 63EE which sets variable number 3 equal to EE.

**Interpretive Addition Program**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>3071</td>
<td>entry to interpreter</td>
</tr>
<tr>
<td>02</td>
<td>63EE</td>
<td>set V3 equal to EE</td>
</tr>
<tr>
<td>04</td>
<td>F400</td>
<td>set V4 equal to switch byte, waits for in on, off</td>
</tr>
<tr>
<td>06</td>
<td>D4FF</td>
<td>display V4 on hex display for about 1.8 seconds</td>
</tr>
<tr>
<td>08</td>
<td>F500</td>
<td>set V5 equal to switch byte</td>
</tr>
<tr>
<td>0A</td>
<td>D5FF</td>
<td>display V5 on hex display</td>
</tr>
<tr>
<td>0C</td>
<td>8454</td>
<td>set V4 equal to V4 + V5</td>
</tr>
<tr>
<td>0E</td>
<td>D4FF</td>
<td>display V4, now the sum of V4 + V5</td>
</tr>
<tr>
<td>10</td>
<td>4F01</td>
<td>skip next instruction if VF ≠ 01, remember VF will be set to 01 by the 8454 instruction if overflow occurs</td>
</tr>
<tr>
<td>12</td>
<td>D3FF</td>
<td>display V3 (V3 was set equal to EE) this instruction is skipped if VF is anything but 01</td>
</tr>
<tr>
<td>14</td>
<td>1004</td>
<td>go back to instruction 04 to wait for next number</td>
</tr>
</tbody>
</table>

The above program illustrates most of the demonstration interpreter instructions; an important exception is the interpreter subroutine call. Unlike the SEP register technique used in simple machine code programs, interpreter subroutines do not have to return to the main program but can be called from other subroutines. A stack is employed to store the return address when a subroutine call is made and successive calls to subroutines, without returns, push the stack further down. In the demonstration interpreter the stack pointer, R(2), points to the last location used and is pushed down one before a new byte is added to the stack. Each time a return from a subroutine occurs the stack pointer is incremented by one.

The next program is a simple illustration of the use of an interpreter subroutine. A switch byte is entered and displayed. It is then counted down by three's until underflow occurs. A subroutine is used to implement the counting down by three.

**Program to Illustrate Subroutine use**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>3071</td>
<td>entry to interpreter</td>
</tr>
<tr>
<td>02</td>
<td>F500</td>
<td>set V5 equal to switch byte, waits for in on, off</td>
</tr>
<tr>
<td>04</td>
<td>D5FF</td>
<td>display V5 on hex display for about 1.8 seconds</td>
</tr>
<tr>
<td>06</td>
<td>200A</td>
<td>call interpreter subroutine at location 0A</td>
</tr>
<tr>
<td>08</td>
<td>1002</td>
<td>on return from subroutine go to location 02 to read another switch byte</td>
</tr>
<tr>
<td>0A</td>
<td>6603</td>
<td>set V6 equal to 03</td>
</tr>
<tr>
<td>0C</td>
<td>8565</td>
<td>set V5 equal to V5 - V6</td>
</tr>
<tr>
<td>0E</td>
<td>D540</td>
<td>display V5 for ca 0.4 seconds</td>
</tr>
<tr>
<td>10</td>
<td>4F01</td>
<td>skip next instruction if underflow occurs during the subtraction, VF equals 00 on underflow</td>
</tr>
<tr>
<td>12</td>
<td>100C</td>
<td>transfer to location 0C to subtract three more</td>
</tr>
<tr>
<td>14</td>
<td>009E</td>
<td>return from subroutine</td>
</tr>
</tbody>
</table>

In the above program, the call to the subroutine uses one stack position to store the return address. When the interpreter is entered the stack pointer is set to location 71. On calling the subroutine it is decremented by one, to location 70, and 08, the location the interpreter should execute on return from subroutine, is stored there. If we examine location 70 after running this program 08 will be found there.
Two additional stack locations, 6E and 6F are used by 8565 instruction, these locations become F5 and D3 respectively. An explanation of why this occurs is given in the demonstration interpreter listing.

The interpreter also includes an instruction, 00MM, which executes a machine code subroutine at address MM. This is easily accomplished; the control section of the interpreter treats the machine code subroutine as if it were one of the subroutines written to execute CHIP-8 instruction. All the subroutines which execute CHIP-8 instructions end with a D4 byte.

The following program poses simple addition problems and illustrates most of the demonstration interpreter instructions. It contains a machine language subroutine which generates two random numbers when the in button is pushed. On entry, the program displays AA and the Q light comes on. When the input button is pressed a simple addition problem (base 10) is presented; for example 17AD (for and) 32E0 (for equals) may be displayed. If 00 is entered the problem is shown again, if the correct answer is entered it is displayed followed by AA. However if an incorrect answer is entered EE is shown followed by the correct answer. The program requires 36 interpreter instructions and a machine language subroutine of 25 bytes. An interpreter subroutine is used to generate two random numbers in VD and VE. The displayed numbers are all less than 99 (base 10) to accommodate the hex display and the simple hex to decimal conversion routine which fails for numbers greater or equal to 100 (base 10).

**Program for Addition Problems**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>3071</td>
<td>entry to interpreter</td>
</tr>
<tr>
<td>02</td>
<td>60E0</td>
<td>set V0 equal to E0</td>
</tr>
<tr>
<td>04</td>
<td>61EE</td>
<td>set V1 equal to EE</td>
</tr>
<tr>
<td>06</td>
<td>62AD</td>
<td>set V2 equal to AD</td>
</tr>
<tr>
<td>08</td>
<td>63AA</td>
<td>set V3 equal to AA</td>
</tr>
<tr>
<td>0A</td>
<td>D300</td>
<td>display V3 (AA) on the display but no delay for display</td>
</tr>
<tr>
<td>0C</td>
<td>004A</td>
<td>call machine language subroutine which generates random numbers in VD and VE when in is pushed</td>
</tr>
<tr>
<td>0E</td>
<td>8BE0</td>
<td>set VB equal to VE as preparation for summing the two random numbers</td>
</tr>
<tr>
<td>10</td>
<td>8BD4</td>
<td>set VB equal to VD + VE, sum of the two random numbers</td>
</tr>
<tr>
<td>12</td>
<td>203A</td>
<td>call the interpreter subroutine which converts from hex to decimal, answer is returned in VA and VB is changed</td>
</tr>
<tr>
<td>14</td>
<td>8CA0</td>
<td>save answer on return from subroutine by setting VC equal to VA</td>
</tr>
<tr>
<td>16</td>
<td>8BE0</td>
<td>set VB equal to VE, one of the random numbers</td>
</tr>
<tr>
<td>18</td>
<td>203A</td>
<td>call subroutine to make VA the decimal equivalent of VB</td>
</tr>
<tr>
<td>20</td>
<td>203A</td>
<td>call subroutine to make VA the decimal equivalent of VB</td>
</tr>
<tr>
<td>22</td>
<td>DAFF</td>
<td>display VA, first random number (base 10)</td>
</tr>
<tr>
<td>24</td>
<td>DAFF</td>
<td>display V2 (AD)</td>
</tr>
<tr>
<td>26</td>
<td>F600</td>
<td>make V6 the entered byte</td>
</tr>
<tr>
<td>28</td>
<td>4600</td>
<td>skip the next instruction if V6 is equal to 00</td>
</tr>
<tr>
<td>2A</td>
<td>1016</td>
<td>here only if V6 is 00, back to 16 to repeat display</td>
</tr>
<tr>
<td>2C</td>
<td>D6FF</td>
<td>display V6, the entered byte</td>
</tr>
<tr>
<td>2E</td>
<td>86C5</td>
<td>set V6 equal to V6 - VC, VC is the correct answer (base 10)</td>
</tr>
<tr>
<td>30</td>
<td>4600</td>
<td>skip next instruction unless V6 equals 00, i.e. skip on wrong answer transfer to 0A to show AA if answer is correct display V1 (EE)</td>
</tr>
<tr>
<td>32</td>
<td>100A</td>
<td>display VC, correct answer transfer to 0C to begin next problem</td>
</tr>
</tbody>
</table>
| 34   | D1FF | end of main, begin hex to decimal conversion subroutine, subroutine adds 06 to VB for every time 0A occurs, argument is passed in
VB and returned in VA
set VA equal to VB
3A 8AB0

set V9 equal to 06
3C 6906

set VB equal to VB - V8, i.e. subtract 0A from VB
40 8B85

skip next instruction if VF equals 00, i.e. skip unless underflow
42 4F00

return from subroutine on underflow
44 009E

set VA equal to VA + V9, i.e. add 06 to VA
46 8A94

transfer to location 40 to subtract 0A from VB, this is the end of the subroutine
48 1040

The above program illustrates one of the weaknesses of CHIP-8. There is no way to pass arguments to interpreter subroutines except through the variables and we must execute a number of variable transfer instructions to use the hex to decimal interpreter subroutine. This weakness is partly overcome in the full interpreter by the inclusion of instructions which transfer the variables to and from memory. The full interpreter also includes an instruction which generates random numbers and a hex to decimal conversion routine. In the next section this program has been rewritten for the full interpreter.

Now let’s look at the listing for the demonstration interpreter. It uses the 16 locations F0 through FF to store the 16 variables. The interpreter examines each instruction in turn and carries out the desired operation by calling the correct subroutine. It uses the following registers:

**Demonstration Interpreter Register Use**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(2)</td>
<td>stack pointer</td>
</tr>
<tr>
<td>R(3)</td>
<td>set to address of machine code subroutine that carries out instruction, i.e. subroutine program counter</td>
</tr>
<tr>
<td>R(4)</td>
<td>program counter for control section of interpreter</td>
</tr>
<tr>
<td>R(5)</td>
<td>program counter for interpretive code</td>
</tr>
<tr>
<td>R(6)</td>
<td>VX pointer, points to one of 16 variables</td>
</tr>
<tr>
<td>R(7)</td>
<td>VY pointer, points to one of 16 variables</td>
</tr>
<tr>
<td>R(C)</td>
<td>used to point to a table of addresses</td>
</tr>
</tbody>
</table>

The interpreter is designed for use on a single page of memory and will work in the basic 1/4K Elf as it stands. For expanded systems R(2), R(3), R(4), R(5), R(6), R(7) and R(C) have to have their high order bytes set to the page the interpreter resides on. Perhaps the simplest way to do this initialization for an expanded system is to change the entry point of the interpreter from 71 to 68 and add the following code from locations 68 through 73:

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>F8 00</td>
<td>load page number to D, here 00 but interpreter can be on any page</td>
</tr>
<tr>
<td>6A</td>
<td>B2 B3 B4</td>
<td>initialize registers</td>
</tr>
<tr>
<td>6D</td>
<td>B5 B6 B7 BC</td>
<td>initialize register</td>
</tr>
<tr>
<td>71</td>
<td>F8 68 A2</td>
<td>establish top of stack at M(68) instead of at M(71)</td>
</tr>
</tbody>
</table>

Note that the stack pointer is now initialized at location 68 instead of at location 71. Alternately one can place the interpreter on a higher page in memory, do the initialization of the registers on page 00 and then transfer control to the interpreter. If this method is used the interpretive code can start at location 00 and R(5),0, the
The address of the first interpreter instruction, can be set to 00.

**Demonstration Interpreter Listing**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>F8 71 A2</td>
<td>establish stack pointer</td>
</tr>
<tr>
<td>74</td>
<td>F8 7A A4</td>
<td>R(4) will be program counter for control section of interpreter</td>
</tr>
<tr>
<td>77</td>
<td>F8 02 A5</td>
<td>R(5) is program counter for interpretive code, first instruction is at M(02)</td>
</tr>
<tr>
<td>7A</td>
<td>D4</td>
<td>establish program counter for control section</td>
</tr>
<tr>
<td>7B</td>
<td>E2</td>
<td>make R(2) the X register, this is the entry point for return to control section after completing a subroutine call</td>
</tr>
<tr>
<td>7C</td>
<td>45 AF</td>
<td>load first half of instruction and save it in R(F).0</td>
</tr>
<tr>
<td>7E</td>
<td>F6 F6 F6 F6</td>
<td>shift right to get most significant digit, most significant digit determines type of instruction</td>
</tr>
<tr>
<td>82</td>
<td>32 98</td>
<td>if D is zero (type 0 instruction) we have machine code subroutine call, transfer to location 98</td>
</tr>
<tr>
<td>84</td>
<td>F9 A0</td>
<td>else or against A0 to get address from table of subroutine locations (see locations A1 to AF)</td>
</tr>
<tr>
<td>86</td>
<td>AC</td>
<td>save address in R(C).0</td>
</tr>
<tr>
<td>87</td>
<td>8F</td>
<td>bring back instruction</td>
</tr>
<tr>
<td>88</td>
<td>F9 F0</td>
<td>or against F0 to get VX address</td>
</tr>
<tr>
<td>8A</td>
<td>A6</td>
<td>establish R(6) as VX pointer</td>
</tr>
<tr>
<td>8B</td>
<td>05</td>
<td>load second half of instruction, note that R(5) is left pointing to second half of instruction</td>
</tr>
<tr>
<td>8C</td>
<td>F6 F6 F6 F6</td>
<td>shift right to get VY pointer</td>
</tr>
<tr>
<td>90</td>
<td>F9 F0</td>
<td>or against F0 to get VY address</td>
</tr>
<tr>
<td>92</td>
<td>A7</td>
<td>establish R(7) as VY pointer</td>
</tr>
<tr>
<td>93</td>
<td>0C A3</td>
<td>pick up subroutine address from table and point R(3) to subroutine call subroutine to do instruction</td>
</tr>
<tr>
<td>95</td>
<td>D3</td>
<td>on return from subroutine go to 7B for next instruction</td>
</tr>
<tr>
<td>96</td>
<td>30 7B</td>
<td>here for machine code subroutine, load address to D and go to 94 to establish R(3)</td>
</tr>
<tr>
<td>98</td>
<td>45 30 94</td>
<td>begin subroutine for 6XKK instruction</td>
</tr>
<tr>
<td>9B</td>
<td>45 56</td>
<td>load KK to D, store in VX</td>
</tr>
<tr>
<td>9C</td>
<td>D4</td>
<td>return control section</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>subroutine that restores R(5) on return from interpreter subroutine load return address from stack</td>
</tr>
<tr>
<td>9F</td>
<td>A5 D4</td>
<td>restore R(5) and return the next 15 bytes are the subroutine locations, i.e. go to B5 for 10MM instructions, go to B0 for 20MM instructions, etc. illegal instructions go to E5 where they are ignored</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>subroutine for 20MM instructions</td>
</tr>
<tr>
<td>9F</td>
<td>A5 D4</td>
<td>load return address to D</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>save on stack, push stack down first</td>
</tr>
<tr>
<td>9F</td>
<td>A5 D4</td>
<td>restore R(5) so that it points to MM</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>rest of this subroutine is shared with 10MM instructions</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>load MM change R(5) to point to new address return</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>begin subroutine for 4XKK instruction</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>load KK to D</td>
</tr>
<tr>
<td>9E</td>
<td>42</td>
<td>make R(6) the X register, the VX pointer x’or VX against KK</td>
</tr>
</tbody>
</table>
BB 32 BF  
return immediately if D equals 0, i.e. if VX equals KK
BD 15 15  
else increment instruction program counter twice
BF D4  
return
- -  
here begin the 8XYN instructions
C0 45  
load YN to D
C1 FA 0F  
and off N to get 0N in D
C3 3A C8  
go to C8 unless N is zero
C5 07 56  
load VY, write to VX
C7 D4  
return
- -  
here on other 8XYN instructions, makes up FN D3 on stack, transfers control to stack and obeys the two instructions, uses R(2) as program counter
C8 AF  
save 0N
C9 22  
push stack down
CA F8 D3 73  
load D3 to D, write to stack
CD 8F F9 F0  
load 0N, or against F0 to get F1, F2, F3, F4, F5, F6, F7, or FE
D0 52  
write to stack
D1 E6  
make VX pointer the X register
D2 07  
load VY to D
D3 D2  
go to stack to obey FN D3 instructions
D4 56  
on return save result as VX
D5 F8 FF A6  
point R(6) to VF
D8 F8 00  
clear D
DA 7E 56  
shift DF into D and save as VF
DC D4  
return
- -  
begin FX00 subroutine
DD 7B  
Q on to indicate waiting for byte
DE 3F DE  
wait for in on
E0 37 E0  
wait for in off
E2 E6  
make VX pointer the X register
E3 6C  
switch byte to VX
E4 7A  
turn Q off
E5 45 D4  
advance instruction counter, return – also used for illegal instructions
- -  
begin DXKK subroutine
E7 E6  
make VX pointer the X register
E8 64  
display VX
E9 45 BF  
load KK to R(F),1
EB 2F 9F  
decrement R(F), load R(F).1
ED 3A EB  
go to EB unless D is zero, delay loop
EF D4  
return – end of interpreter
F0-FF  
locations where the 16 interpreter variables are stored

The Chip-8 Language

This section contains a brief discussion of the CHIP-8 language and a list of the available instructions. Further information about RCA’s VIP machine and about CHIP-8 can be found in two articles by Joseph Weisbecker (“COSMAC VIP, the RCA Fun Machine”, in the August, 1977 Byte magazine p. 30, and “An Easy Programming System”, in the December, 1978 Byte magazine p.108) and in RCA’s literature. The full CHIP-8 instruction set is listed in the table at the end of this chapter.

Many of the basic features of the CHIP-8 language are explained and illustrated in section 2 and the demonstration interpreter contains ten instructions which are identical to those in the full CHIP-8 set. The complete language is designed for use with low resolution graphics and the display subroutine is the longest and most complex of the subroutines in the interpreter. A number of TV games have been written with CHIP-8 and it is well suited for this purpose. The display instruction is used in conjunction with a memory pointer and the CHIP-8 variables and has the form DXYN. The values of VX and VY indicate where on the video display to show information, and the value of N indicates how many bytes to display. A memory pointer, called I, gives the starting address of the information to be displayed and must be set by other instructions. Positions in the display field are determined by a rectangular coordinate system with the origin in the upper left corner; 64 horizontal positions, designated by VX and 32 vertical positions designated by VY, are available. The bytes to be displayed are exclusively ored against the display field; an important feature for TV games. Portions of memory bytes which extend beyond the display...
Another important feature of the language is the 16 one byte variables, V0 through VF, which are held in random access memory. Two of these variables V0 and VF are used for special purposes. V0 is used in a kind of computed go statement, the BMMM instruction. Control is transferred to location MMM to which has been added the value of V0. As in the demonstration interpreter, VF is used to indicate overflow in arithmetic operations. It is also used to indicate when a display instruction attempts to show a position which is already being displayed. As the display instruction exclusively or’s the data to be displayed against the display field, such an attempt turns off the displayed position. VF is set to 01 to indicate this occurrence. This serves as a simple way to determine if a missile has struck a target in a TV game.

A third important feature of CHIP-8, already mentioned in the discussion of the display routine, is the memory pointer, I. The memory pointer can be set both directly and indirectly; besides its use as a display pointer, it also serves as a pointer for transferring variables to and from memory.

The full CHIP-8 instruction set has six skip instructions all of which follow the principle of the skip instruction included in the demonstration interpreter. That is, the next interpreter instruction is skipped over if on testing a condition it is found to be true.

The instructions which have 8 as the first hexadecimal digit perform arithmetic and logic operations and are all included in the demonstration interpreter. Note again that VF is used to indicate overflow and that the value of VF is changed by 8XY1, 8XY2, 8XY3, 8XY4, 8XY5, 8XY6, 8XY7, and 8XYE instructions.

A number of instructions which are not included in the demonstration interpreter are the “F” instructions. Several of these are used in conjunction with the memory pointer. For example the FX29 instruction points I at a 5 byte memory pattern which corresponds to the least significant hex digit of VX. If V7 were 38 and F729 instruction were executed I would point to the first byte of the series F0, 90, F0, 90, F0 (a pattern for the symbol “8”) and DXY5 instruction would show an “8” on the display. The FX33 instruction is a binary to decimal conversion routine. The value of VX is converted to a 3 digit decimal number with the hundreds digit stored at location I, the tens digit at location I + 1, and the units digit at location I + 2. The FX55 and FX56 instructions use the memory pointer to transfer values from memory to the variables, respectively.

Other “F” instructions include a settable tone generator (FX18) (see the section on Hardware Differences), an instruction to set a timer (FX15), an instruction to read the timer (FX07), and an instruction to read the keyboard (FX0A). An additional “F” instruction has been added for the Elf; FX75, which displays the value of VX on the hex display.

Other useful instructions which are not present in the demonstration interpreter include a random number generator (CXKK where KK is anded against a random byte before being transferred to VX), and an instruction which adds a byte to one of the variables, 7XKK. Two of the CHIP-8 instructions 00E0 (erase display) and 0DEE (return from a CHIP-8 subroutine) are implemented as machine code subroutines resident in the interpreter itself. They are therefore dependant upon the page where CHIP-8 is located and will have to be changed if CHIP-8 is relocated. This also is the reason that the return from a subroutine is 009E in the demonstration interpreter and 00EE in the full CHIP-8 interpreter.

To illustrate the use of the full instruction set, let’s rewrite one of the programs that used the demonstration interpreter, the one involving addition problems. The following program constructs simple addition problems using two randomly chosen numbers between 0 and 127. On entry to the program a problem is presented, e.g. 076 + 093 = ?. An answer is entered through the keyboard one digit at a time (i.e. 1, 6, 3) and when the last digit is entered 163 is displayed. A C flows the entered number if it is correct and an E if it is incorrect. In the case of an incorrect answer the correct answer is also shown. Another problem is given when any key is entered. The program consists of 67 CHIP-8 instructions and also uses 32 bytes for constants and work space.

**Program for Addition Problems**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0200</td>
<td>00E0</td>
<td>erase display</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>first set up problems and answer</td>
</tr>
<tr>
<td>0202</td>
<td>CD7F</td>
<td>VD equals random number</td>
</tr>
<tr>
<td>0204</td>
<td>CE7F</td>
<td>VE equals random number</td>
</tr>
<tr>
<td>0206</td>
<td>8CD0</td>
<td>VC = VD</td>
</tr>
<tr>
<td>0208</td>
<td>8CE4</td>
<td>VC = VD + VE (the answer)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>next convert to decimal and display the problem</td>
</tr>
<tr>
<td>020A</td>
<td>A2A2</td>
<td>point I to work space</td>
</tr>
</tbody>
</table>
020C 6A00 set VA = 00, display pointer
020E 6B00 set VB = 00, display pointer
0210 FD33 M(I) equals 3 digit decimal equivalent of VD
0212 F265 V0, V1, V2 equals M(I)
0214 2276 call CHIP-8 subroutine (displays 3 digit number in V0, V1, and V2)
0216 A288 point I to + pattern
0218 7A07 VA = VA + 07, display pointer
021A DAB display + pattern
021C A292 point I to ? pattern
0220 FE33 M(I) equals 3 digit decimal equivalent of VE
0222 F265 V0, V1, V2 equals M(I)
0224 2276 call subroutine to display VE
0226 A2A5 point I to work space
0228 7A05 VA = VA + 05, display pointer
022A DAB display = pattern
022C A292 point I to ? pattern
0230 6B08 set VB = 08, display pointer
0232 DAB display ? pattern
0234 F00A V0 = least significant digit of switch byte
0236 F10A V1 = switch byte (LSD)
0238 F20A V2 = switch byte (LSD)
023A DAB display ? pattern (erases it)
F
023C 6A15 set VA = 15, display pointer
023E 2276 call subroutine to display entered answer
- - now compute answers, right to 025c, wrong to 0262
0240 A2A5 point I to work space
0242 F255 V0, V1, V2 – correct answer
0244 A2A2 V3, V4, V5 – entered answer
0246 FC33 V3 = V3 – V0
0248 F565 skip to 0262, error
024A 8305 V4 = V4 – V1
024C 3300 skip if V3 = 00
024E 1262 go to 0262, error
0250 8415 V4 = V4 – V1
0252 3400 skip if V4 = 00
0254 1262 go to 0262, error
0256 8525 V5 = V5 – V2
0258 3500 skip if V5 = 00
025A 1262 go to 0262, error
- - here if answer correct
025C 660C set V6 = 0C
025E F618 set tone duration (reward)

0260 126A go to 026A
0262 6A15 set VA = 15, display pointer
0264 6B10 set VB = 10, display pointer
0266 2276 call subroutine to display correct answer
0268 660E V6 = 0E
026A 6A26 VA = 26, display pointer
026C 6B08 VB = 08, display pointer
026E F629 point I to C or E pattern
0270 DAB display C or E
5
0272 F00A wait for any input
0274 1200 to 0200 for next problem
0276 F029 point I to pattern for V0
0278 DAB display it
027A 7A05 VA = VA + 05, display pointer
027C F129 point I to pattern for V1
027E DAB display it
5
0280 7A05 VA = VA + 05, display pointer
0282 F229 point I to pattern for V1
0284 DAB display it
5
0286 00EE return from subroutine
- - patterns and work space
0288 2020 pattern for + sign
028A F820
028C 2000
028E 00FF pattern for ? sign
0290 00FF
0292 FFFF
0294 0303
0296 03FF
0298 FFC0
029C C0C0
029E 00C0
02A0 C000
02A2 - work space
02A4 -
02A6 -

Table 2

Full Interpreter Instructions

0MMM do a machine code subroutine at location 0MMM (The machine code subroutine must end with D4)
1MMM go to 0MMM; control is transferred to location 0MMM in the interpretive code
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2MMM</td>
<td>do an interpreter subroutine at location 0MMM (the interpreter subroutine must end with 00EE)</td>
</tr>
<tr>
<td>3XKK</td>
<td>skip if VX = KK; the next interpreter instruction is skipped over if VX equals KK</td>
</tr>
<tr>
<td>4XKK</td>
<td>skip if VX ≠ KK; the next interpreter instruction is skipped over if VX does not equal KK</td>
</tr>
<tr>
<td>5XY0</td>
<td>skip if VX = VY; the next interpreter instruction is skipped over if VX equals VY (see 9XY0)</td>
</tr>
<tr>
<td>6XKK</td>
<td>set VX = KK; variable X is made equal to KK</td>
</tr>
<tr>
<td>7XKK</td>
<td>set VX = VX + KK; add KK to variable X</td>
</tr>
<tr>
<td>8XY0</td>
<td>set VX = VY; variable X is made equal to variable Y</td>
</tr>
<tr>
<td>8XY1</td>
<td>set VX = VX or VY; variable X is made equal to the result of VX logically or'ed against VY (Note that VF is changed)</td>
</tr>
<tr>
<td>8XY2</td>
<td>set VX = VX and VY; variable X is made equal to the result of VX logically anded against VY (Note that VF is changed)</td>
</tr>
<tr>
<td>8XY3</td>
<td>set VX = VX xor VY; variable X is made equal to the result of VX logically xor'ed against VY (Note that VF is changed)</td>
</tr>
<tr>
<td>8XY4</td>
<td>set VX = VX + VY; variable X is made equal to the sum of VX and VY (Note that VF becomes 00 if the sum is less than or equal to FF and 01 if the sum is greater than FF)</td>
</tr>
<tr>
<td>8XY5</td>
<td>set VX = VX – VY; variable X is made equal to the difference between VX and VY (Note that VF becomes 00 if VX is less than VY and 01 if VX is greater than or equal to VY)</td>
</tr>
<tr>
<td>8XY6</td>
<td>set VX = VY shifted right 1 bit position (Note bit 0 is shifted into VF)</td>
</tr>
<tr>
<td>8XY7</td>
<td>set VX = VY - VX; variable X is made equal to the difference between VY and VX (Note that VF becomes 00 if VY is less than VX and 01 if VX is greater than or equal to VX)</td>
</tr>
<tr>
<td>8XYE</td>
<td>set VX = VY shifted left 1 bit position (Note bit 0 is shifted into VF)</td>
</tr>
<tr>
<td>9XY0</td>
<td>skip if VX ≠ VY; the next interpreter instruction is skipped over if VX does not equal VY (see 5XY0)</td>
</tr>
<tr>
<td>AMMM</td>
<td>point I at 0MMM; the memory pointer is set to 0MMM</td>
</tr>
<tr>
<td>BMMM</td>
<td>go to 0MMM + V0, the value of V0 is added to 0MMM and control is transferred to the resulting location</td>
</tr>
<tr>
<td>CXKK</td>
<td>set VX to a random byte; random byte is anded against KK first</td>
</tr>
<tr>
<td>DXYN</td>
<td>display N byte pattern at coordinates VX, VY; I (memory pointer) gives starting locations to be displayed. The displayed locations are exclusively or'ed against display field. VF becomes 01 if some of the display field is already set, 00 if it is not.</td>
</tr>
<tr>
<td>EX9E</td>
<td>skip if VX = hex key; skip next instruction if the least significant digit of VX equals the least significant digit of the keyboard.</td>
</tr>
<tr>
<td>EXA1</td>
<td>skip if VX ≠ hex key; skip next instruction if the least significant digit of VX does not equal the least significant digit of the keyboard.</td>
</tr>
<tr>
<td>FX07</td>
<td>set VX to the value of the timer; timer is counted down in interrupt routine</td>
</tr>
<tr>
<td>FX0A</td>
<td>set VX = hex key; sets VX equal to the least significant digit of the keyboard, waits for in on, off</td>
</tr>
<tr>
<td>FX15</td>
<td>set timer to VX; timer is counted down in interrupt routine so 01 is ca. 1/60th second</td>
</tr>
<tr>
<td>FX18</td>
<td>set tone duration to VX; turns Q on for duration specified by VX, 01 is ca. 1/60th second</td>
</tr>
<tr>
<td>FX1E</td>
<td>set I to I + VX; add the value of VX to the memory pointer</td>
</tr>
<tr>
<td>FX29</td>
<td>point I to pattern for least significant digit of VX</td>
</tr>
<tr>
<td>FX33</td>
<td>convert VX to decimal; 3 decimal digits are stored at M(I), M(I + 1), and M(I + 2), I does not change save V0 through VX in memory at locations specified by I, V0 at M(I), V1 at M(I+1), etc, I becomes I + X + 1</td>
</tr>
<tr>
<td>FX55</td>
<td>transfer memory locations specified by I to variables V0 through VX, V0 becomes M(I), V(1) becomes M(I+1), etc, I becomes I + X + 1</td>
</tr>
<tr>
<td>FX65</td>
<td>display the value of VX on the hex display</td>
</tr>
</tbody>
</table>
Hardware Differences between 1802 Computers

The most important difference between the various versions of the CCOSMAC ELF and the COSMAC VIP is the keyboard. The COSMAC VIP has a hex keyboard; however it is not connected to an input port. Instead the least significant 4 bits of a bus output byte (Out 2, 62) are decoded and the 16 output lines connected to the corresponding hex keys. Each key is connected to one of the flag lines (EF3). To determine which key is depressed requires a software routine which scans the keyboard. Scanning is done by repeatedly outputting the 16 possible least significant hex digits and examining the flag line to see which digits cause it to be pulled low. Debouncing is also carried out within the software routines; there is an approximately 1/15 second software delay to debounce both opening and closing of a keyboard switch.

COSMAC ELF computers on the other hand are variable in design and have a variety of ways to input information from keyboards or switches. Indeed the September, 1976 issue of Popular Electronics describes a way to connect a scanned hex keyboard, much like that contained in the VIP, to the ELF. However most of the commercially available ELFs (e.g. Super Elf and Elf-2) have latched hex keyboards with roll-over. The latches are connected to an input port and one can examine the contents of these latches at any time under software control. A hardware debounced button (the in button) can be used as a device to indicate to a software routine that we wish the switch latches read. An additional feature of the Elf is the ability to carry out direct memory access input from the keyboard by depressing the in button when the computer is in the load mode. This feature is not required by the VIP which has an operating system in ROM.

These different methods in inputting information from the keyboard have different advantages and disadvantages, neither is really totally satisfactory. The VIP’s keyboard has one significant advantage. All of the keys are connected directly to a flag line and it is possible to tell, with software, when a key is being depressed and if so which one. A quick response to keyboard entry is therefore possible and this property is particularly desirable for TV games. It also makes possible an operating system which enters bytes directly from the keyboard to memory without the necessity of pushing an in button. These features are more difficult with a roll-over latched keyboard like that found in many ELFs. Entered bytes can only be read from latches and there is no way, with software, to determine when a single key is repeatedly entered; that is we could never determine if B, B, B was entered because the contents of the latches would never change. This difficulty could, of course, be overcome with some simple hardware changes to the ELF.

The advantage of the ELF keyboard is that the contents of the keyboard latches can be transferred directly to memory by instituting a direct memory access cycle. This, in fact, is what makes the ELF a viable machine without read only memory. However the ELF would be easier to use if the contents of the keyboard latches were displayed and if a signal were provided which made it unnecessary to push the in button.

Another hardware difference is in the treatment of the Q line. In the VIP the Q line is attached to a simple oscillator, and this in turn can be connected to a speaker. Hence in the VIP when the Q line is turned on, a tone is heard in the loudspeaker. This feature can be added to an ELF without much difficulty. It should perhaps be mentioned that the VIP has room on board for one input and one output port, the output uses out-3 (63), and the input port uses in-3 (6B).

Rather than attempt to change the ELF to a VIP by making hardware changes, this booklet accepts the ELF’s as they are and makes the software changes in CHIP-8 to accommodate ELFs. Unfortunately ELF’s are not built to a standard design like the VIP and it is therefore difficult to write software which will suit all ELF users. To compensate for this a detailed listing of the interpreter is presented in the next section. It is hoped that sufficient information is given so that those with ELF’s which differ from commercially available machines will be able to modify the interpreter to suit their machines.

A Complete Elf CHIP-8 Interpreter

This section provides a listing and a discussion of a version of CHIP-8 for COSMAC ELF’s. The main listing of the interpreter is designed for a 4K Elf with memory pages 00 through 0F, the configuration most commonly used by the commercially available ELF’s. It is also possible to use CHIP-8 in the 1 1/4K ELF’s described in the articles in Popular Electronics, but to do so is very tedious unless the switches are replaced with a latched decoded keyboard. This machine has memory pages 00, 04, 05, 06,
and 07 and a version of CHIP-8 for such a machine will also be described. The necessary changes to CHIP-8 will be discussed in the notes included with the full interpreter listing. Similar changes are required when CHIP-8 is relocated in memory and this example may aid those with other styles of machines.

The first consideration in modifying CHIP-8 for use on the ELF is page use. The following page use was chosen for the 4K Elf’s with memory pages 00 through 0F:

<table>
<thead>
<tr>
<th>Page</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>first half of interpreter</td>
</tr>
<tr>
<td>01</td>
<td>second half of interpreter</td>
</tr>
<tr>
<td>02 – 0D</td>
<td>reserved for interpretive code</td>
</tr>
<tr>
<td>0E(first half)</td>
<td>character table and interrupt routine</td>
</tr>
<tr>
<td>0E(second half)</td>
<td>variables, work space and stack</td>
</tr>
<tr>
<td>0F</td>
<td>display page</td>
</tr>
</tbody>
</table>

This choice of page usage maximizes the similarity of ELF CHIP-8 and VIP CHIP-8. However it is possible to relocate the code to other places in memory and it might be better to accept the changes in CHIP-8 and place the interpreter on pages 0C and 0D. Relocation is necessary to implement the 1 1/4K version. Because of this, some changes in the language are necessary for the 1 1/4K version and the instruction 00E0 becomes 04E0 and 00EE becomes 04EE. Page use for the 1 1/4 K version is as follows:

<table>
<thead>
<tr>
<th>Page</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>display page</td>
</tr>
<tr>
<td>04</td>
<td>first half of interpreter</td>
</tr>
<tr>
<td>05</td>
<td>second half of interpreter</td>
</tr>
<tr>
<td>06 (first half)</td>
<td>character table and interrupt routine</td>
</tr>
<tr>
<td>06 (second half)</td>
<td>variables, work space and stack (There is room for a small operating system in the middle of page 6) interpretive code</td>
</tr>
<tr>
<td>07</td>
<td></td>
</tr>
</tbody>
</table>

Register use is the same as it is in the VIP version of CHIP-8 as follows:

**Use of Registers**

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(0)</td>
<td>DMA address</td>
</tr>
<tr>
<td>R(1)</td>
<td>interrupt address</td>
</tr>
<tr>
<td>R(2)</td>
<td>stack, sometimes X register</td>
</tr>
<tr>
<td>R(3)</td>
<td>program counter for interpreter subroutines</td>
</tr>
<tr>
<td>R(4)</td>
<td>program counter for control section of interpreter</td>
</tr>
<tr>
<td>R(5)</td>
<td>CHIP-8 instruction program counter</td>
</tr>
<tr>
<td>R(6)</td>
<td>variable pointer, the VX pointer</td>
</tr>
<tr>
<td>R(7)</td>
<td>variable pointer, the VY pointer</td>
</tr>
<tr>
<td>R(8)</td>
<td>timer</td>
</tr>
<tr>
<td>R(9)</td>
<td>random numbers</td>
</tr>
<tr>
<td>R(A)</td>
<td>the I pointer</td>
</tr>
<tr>
<td>R(B)</td>
<td>display page pointer</td>
</tr>
<tr>
<td>R(C)</td>
<td>used for scratch but available for machine code subroutines</td>
</tr>
<tr>
<td>R(D)</td>
<td>used for scratch but available for machine code subroutines</td>
</tr>
<tr>
<td>R(E)</td>
<td>used for scratch but available for machine code subroutines</td>
</tr>
<tr>
<td>R(F)</td>
<td>used for scratch but available for machine code subroutines</td>
</tr>
</tbody>
</table>

**Complete CHIP-8 Interpreter Listing**

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>first initialize the registers</td>
</tr>
<tr>
<td>0000</td>
<td>F8 0E B1</td>
<td>high order interrupt address</td>
</tr>
<tr>
<td>03</td>
<td>F8 46 A1</td>
<td>low order interrupt address replace 00E with 06 for 1 1/4K Elf</td>
</tr>
<tr>
<td>06</td>
<td>F8 0F BB</td>
<td>establish display page, replace 0F with 00 for 1 1/4K Elf</td>
</tr>
<tr>
<td>09</td>
<td>F8 0E B2</td>
<td>establish a high order stack address replace 0E with 06 for 1 1/4K Elf</td>
</tr>
<tr>
<td>0C</td>
<td>B6</td>
<td>establish page for variables, work space (same as stack page)</td>
</tr>
<tr>
<td>0D</td>
<td>F8 CF A2</td>
<td>establish low order stack address</td>
</tr>
<tr>
<td>10</td>
<td>F8 01 B5</td>
<td>high order address for first CHIP-8 instruction, replace 01 with 05 for 1 1/4K Elf</td>
</tr>
<tr>
<td>Address</td>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>F8 FC A5</td>
<td>low order address for first CHIP-8 instruction, replace FC with FA for 1 1/4K Elf</td>
<td>38 A7 save in R(7).0, the VY pointer</td>
</tr>
<tr>
<td>00 B4</td>
<td>establish control section program counter, replace 00 with 04 for a 1 1/4K Elf</td>
<td>39 4C B3 interpreter high order subroutine address from table to R(3).1</td>
</tr>
<tr>
<td>1C A4</td>
<td>establish low order address for control section program counter</td>
<td>3B 8C FC 0F AC set up pointer to table of low order subroutine addresses</td>
</tr>
<tr>
<td>D4</td>
<td>make R(4) the program counter, this ends initialization of registers</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>begin control section of interpreter, on return from interpreter subroutine location 1D is entered</td>
<td>41 D3 change to subroutine program counter</td>
</tr>
<tr>
<td>96 B7</td>
<td>establish high order VY pointer</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>establish x-register make R(C).1 the current page</td>
<td>44 8F reload 1st byte of CHIP-8 instruction</td>
</tr>
<tr>
<td>BC</td>
<td>make R(C).1 the current page</td>
<td>45 B3 save in R(3).1, high order machine code subroutine address</td>
</tr>
<tr>
<td>45</td>
<td>load first byte of a CHIP-8 instruction in R(F).0</td>
<td>46 45 load advance – 2nd byte of interpreter instruction</td>
</tr>
<tr>
<td>AF</td>
<td>save 1st byte of instruction to R(F).0</td>
<td></td>
</tr>
<tr>
<td>F6 F6 F6 F6</td>
<td>shift right 4 times to get most significant digit</td>
<td>47 30 40 go to location 40 to set R(3).0 and call subroutine</td>
</tr>
<tr>
<td>32 44</td>
<td>go to 44 if most significant digit is 0, we have a machine language subroutine else or immediate against 50 to make pointer to table of subroutine locations save result in R(C).0, the register used as a pointer</td>
<td></td>
</tr>
<tr>
<td>F9 50</td>
<td>22 69 12 D4</td>
<td>2A 49</td>
</tr>
<tr>
<td>AC</td>
<td>00 00 00 00</td>
<td>2C 4D</td>
</tr>
<tr>
<td>8F</td>
<td>bring back 1st byte of instruction or immediate against F0 to make VX pointer save in R(6).0, the VX pointer</td>
<td>- -</td>
</tr>
<tr>
<td>F9 F0</td>
<td>51 01 01 01</td>
<td>2D 51</td>
</tr>
<tr>
<td>A6</td>
<td>01 00 01 01</td>
<td>2E 55</td>
</tr>
<tr>
<td>05</td>
<td>01 01 01 01</td>
<td>30 59</td>
</tr>
<tr>
<td>F6 F6 F6 F6</td>
<td>shift right to get most significant digit</td>
<td>32 5D</td>
</tr>
<tr>
<td>F9 F0</td>
<td>or immediate against F0 to make VY pointer</td>
<td>36 60</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- 15 -
Now starts the remainder of the interpreter subroutines entry to the display subroutine instruction, DXYN, review material in section 3 to see what it does. R(6) is used to point to work space, R(A) is I (the memory pointer), R(7).0 and R(D).0 are used to store N the number of bytes to display, and R(C) is used as pointer in to display page

70 06 BE
72 FA 3F

load VX, save in R(E).1 and against 3F (only 64 positions across display field)

74 F6 F6 F6

shift right 3 times (gets row address, i.e. 0-7 in display page)

77 22 52

save word address on stack

79 07

load VY

7A FE FE FE

shift left 3 times to make space for row address

7D F1

or on row address by setting R(C).1 to display page address save in R(C).0

7E AC

7F 9B BC

complete address by setting R(C).1 to display page address

81 45

load advance, 2nd half of instruction

82 FA 0F

and off number of bytes to display

84 AD A7

save in R(D).0 and R(7).0

86 F8 D0

load starting address of work space

88 A6

R(6) now points to work space

89 F8 00 AF

establish R(F).0 as a source of 00

8C 87

load number of bytes to display (a reentry point)

8D 32 F3

to location F3 for housekeeping if all done or if no bytes to display
decrement number of bytes to display
load advance, load display byte and save in R(D).1
reload VX and against 07, save in R(E).0, this is position in word – say R(A) pointed to a location containing FF (1111 1111) and least significant 3 bits of VX were (011) – routine from here to A9 would make two adjacent work locations (0001 1111) and (1110 0000) i.e. it would shift the word to be displayed over by 3 bits and fill in to the left and right with 0.

92 9E

93 FA 07 AE

96 8E

97 32 A2

load word position to A2 if 00, no shift needed

99 9D F6 BD

shift 1 bit to DF, 0 to MSB of D

9C 8F 76 AF

transfer DF to R(F).0, DF to MSB, LSB to DF

repeat number of times in word address
save 1st word in work
save 2nd word in work
point R(6) to next work space
repeat till all display words treated

A2 9D 56

A4 16 8F 56

A7 16

A8 30 89

A9 00

AA 00

idles here after housekeeping, sees locations F3 through FB, still here to transfer work to display – R(C) points to first word to change in display field make R(C) the X register

AB EC

load starting address of work

AC F8 D0

R(6) points to work

AE A6

00 to R(7).0 and eventually to VF

AF F8 00 A7
B2 8D load number of bytes to display, reenters here until done
B3 32 D8 all done?, to D8 to set VF and exit
B5 06 load byte from work and against display field
B7 2D decrement bytes to display
B8 32 BD to BD if result of and is 00, i.e. no points already set
BA F8 01 A7 if points set make R(7).0 and eventually VF, 01
BD 46 reload work to D (load advance)
BE F3 x’or against display field
BF 5C write result to display field
C0 02 reload VX
C1 FB 07 are we at the end of the row?
C3 32 D1 if we are quit, no wrap around
C5 1C else increment R(C)
C6 06 load next word from work
C7 F2 32 CD repeat test for already set bits
CA F8 01 A7 01 to R(7).0 if bits set
CD 06 load from work again
CE F3 5C x’or against filed and write to field
D0 2C 16 decrement R(C), increment R(6)
D2 8C FC 08 load R(C).0 add 08
D3 AC load new address to R(C).0
D6 3B B2 if DF is 0 go to B2 to do more, else we’ve run over bottom and should return
D8 F8 FF A6 load VF address to R(6).0
DB 87 56 load R(7).0 (either 00 or 01) and store in VF
DD 12 D4 fix up stack and return to control section

- - comes here when all done
D8 F8 FF A6 load VF address to R(6).0
DB 87 56 load R(7).0 (either 00 or 01) and store in VF
DD 12 D4 fix up stack and return to control section

DF 00 unused – done with main part of display routine se F3 – FB, a patch for housekeeping entry point for 00E0 instruction (04E0 for 1/4k Elf) a machine code subroutine that erases the display page load display page address to R(F).1
E0 9B BF load FF to R(F).0
E2 F8 FF AF load 00 to D store via F
E5 F8 00 load R(F).0, return from subroutine if D is 00, all done
E7 5F else decrement R(F) and go back to blank another memory location
EB 2F 30 E5 entry point for 00EE instruction (04EE for 1 1/4k Elf) retrieves interpretive code address from stack retrieve high order address
EE 42 B5 then low order address R(5) now set return to control section part of display routine, resets memory pointer
F0 42 A5 load number bytes to display, save in R(7).0 load R(7).0 to D if 00 done, go to AA to wait for DMA decrement R(A) (memory pointer) and R(7)
F2 D4 get back to check if done
F4 30 F5 entry for 6XXK subroutine
FC 45 load KK to D
FD 56 D4 write to VX and return
FF 00 unused, end of page 00 (04 for 1 1/4k Elf)
FA 30 FF load KK to D
FB 30 45 begin page 01 (05 for 1 1/4k Elf)
FD 56 D4 entry for 7XXK subroutine
FF 00 make R(6), VX, the X register
02 F4 add KK to VX
03 56 write result to VX
04 D4 return to control section
05 45 load advance – 2nd byte of F instruction is location to transfer to on this page
06 A3 change R(3) subroutine program counter to correct address
07 98 load timer value to D (see interrupt routine)
08 56 D4 write VX and return entry for FX0A subroutine
0A 3F 0A 37 0C wait for in on, off
0E 22 push down stack
0F 6C read switch byte
10 FA 0F and against 0F to get least significant digit (This corresponds to original Chip-8, could and against FF to read complete byte)
12 12 56 restore stack, write to VX
14 D4 return to control section entry for FX15 subroutine
15 06 load VX to D
16 B8 D4 save in R(8).1 and return
18 06 load VX to D
19 A8 D4 save in R(8).1 and return (see interrupt routine for FX15 and FX18 explanation)
1B 64 100 (base 10)
1C 0A 10 (base 10)
1D 01 1 (base 10)
1E E6 make R(6), VX pointer, the X register

54 00 unused
- - entry for FX55 subroutine transfer variables to memory
55 22 push down stack
56 86 52 load contents of R(6).0 to stack (one of F0-FF)
58 F8 F0 A7 point R(7) to V0
5B 07 load V0, on later entry V1, etc.
5C 5A write to M(R(A))
5D 87 F3 load R(7).0 and x'or against stack byte - passed VX pointer - if result is 00 we're done increment R(7) and memory pointer
5F 17 1A go to 5B to transfer next VX unless done
61 3A 5B else restore stack pointer, return
- - entry for FX65 subroutine transfer memory to variables push down stack
65 22 transfer contents of R(6).0 to stack, on of F0-FF
68 F8 F0 A7 point R(7) to V0
6B 0A load M(R(A)) to D, enters here later write in V0, V1, V2, etc.
6C 57 load R(7).0 and x'or against stack byte - if result is 00 we're done increment R(7) and memory pointer
6D 87 F3 go to 5B to transfer next byte unless done
6F 17 1A else restore stack pointer, return
- - entry for FX75 subroutine transfer VX to hex display
75 E6 make VX pointer the X register
76 12 D4 output VX and return
- - entry for 2MMM subroutine, go to interpreter subroutine store return interpreter code
78 15 85 address on stack
7A 22 73
7C 95 52

7E 25 restore R(5) to point to 2nd half of instruction entry for 1MMM subroutine rest of code through location 85 is shared
7F 45 A5 load MM to D and transfer to R(5).0
81 86 FA 0F retrieve M (most significant part) from R(6).0
84 B5 D4 set R(5).1 and return entry for 3XKK subroutine - skip if VX equals KK
86 45 load KK to D make VX pointer X register, x'or VX against KK
87 E6 F3 return if D does not equal zero
8B 15 15 else skip return to control section entry for 4XKK subroutine
8D D4 load KK to D make VX pointer X register, x'or VX against KK
8E 45 skip if D does not equal zero
8F E6 F3 else return entry for 9XY0 subroutine, skip if VX does not equal VY
91 3A 8B set R(5) to next instruction
93 D4 load VY to D transfer to 8F to complete instruction
94 45 set R(5) to next instruction load VY to D transfer to 8F to complete instruction
95 07 entry for 5XY0 subroutine set R(5) to next instruction
96 30 8F load VY to D transfer to 87 to complete instruction
98 45
entry for E subroutine
EX9E - skip if VX equals keys (LSD),
EXA1 - skip if VX does not equal keys (LSD),
see Section 4 Hardware Differences. Designed
to be as close as possible to original use
in VIP

9C 22 push down stack
9D 6C switch byte to stack, D
9E 06 F3 load VX, x'or against
switch byte
A0 FA 0F and off least significant
digit of answer
A2 52 write result to stack
A3 45 F6 load advance - shift
right 0 to DF for EX9E
instruction, I to DF for
EXA1 instruction
A5 42 load back stack byte,
restore stack
A6 3B AD to AD for EX9E
instruction, carry on for
EXA1 instruction
A8 3F 8B skip if in not depressed
skip if in depressed but
wrong key
AC D4 else return
AD 3F B1 skip if in not depressed
skip if in depressed but
wrong key
AF 32 8B else return

- - entry for BMMM
instruction, go to
0MMM plus V0
B2 F8 F0 A7 point R(7) to V0
B5 E7 make R(7) the X
register
B6 45 load MM
B7 F4 add V0 and D
B8 A5 save it in R(5).0
B9 86 FA 0F load R(6).0 to retrieve
most significant part of
MMM, and off
BC 3B C0 to C0 if no overflow on
addition, all done
BE FC 01 else add 01 to D
C0 B5 D4 set R(5).1 and return
entry for 8XYN
instructions, identical to
those in demonstration
interpreter
C2 45 load YN to D
C3 FA 0F and off N to get 0N

- - entry for BMMM
instruction, go to
0MMM plus V0
B2 F8 F0 A7 point R(7) to V0
B5 E7 make R(7) the X
register
B6 45 load MM
B7 F4 add V0 and D
B8 A5 save it in R(5).0
B9 86 FA 0F load R(6).0 to retrieve
most significant part of
MMM, and off
BC 3B C0 to C0 if no overflow on
addition, all done
BE FC 01 else add 01 to D
C0 B5 D4 set R(5).1 and return
entry for 8XYN
instructions, identical to
those in demonstration
interpreter
C2 45 load YN to D
C3 FA 0F and off N to get 0N
F1 45 AA  load MM - transfer to R(A).0
F3 86 FA 0F  retrieve M from R(6).o
F6 BA  complete memory pointer
F7 D4  end of interpreter subroutines

remaining 8 locations are used for interpretive code, starting address of interpretive code is 01
FC for 4k interpreter, 05FA for 1 1/4k interpreter

F8 00 00  unused, this is 4K version
FA 00 00  unused
FC 00 E0  erase display page
FE 00 49  turn on TV
02 00  start of interpreter code
05 F8 00 00  unused
FA 04 E0  erase display page
FC 04 49  turn on TV
FE 17 00  transfer to page 7 for interpreter code

Character Table and Interrupt Routine

Add. Code Notes
- - This code could go on any page, as written it is on page 0E for the 4k version and page 06 for the 1 1/4k version
0E 00 30 39 22 2A  pointers to 0, 1, 2, 3
04 3E 20 24 34  pointers to 4, 5, 6, 7
08 26 28 2E 18  pointers to 8, 9, A, B
0C 14 1C 10 12  pointers to C, D, E, F
- - next 51 bytes are the display symbols for the characters, 5
bytes/symbol
10 F0 80  start E display
12 F0 80  start F display
14 F0 80  start C display
16 80 80
18 F0 50  start B display
1A 70 50
1C F0 50  start D display
1E 50 50
20 F0 80  start 5 display
22 F0 10  start 2 display

Character Table:

Add.  Code  Notes
0E 00 30 39 22 2A  pointers to 0, 1, 2, 3
04 3E 20 24 34  pointers to 4, 5, 6, 7
08 26 28 2E 18  pointers to 8, 9, A, B
0C 14 1C 10 12  pointers to C, D, E, F
- - next 51 bytes are the display symbols for the characters, 5

Add.  Code  Notes
10 F0 80  start E display
12 F0 80  start F display
14 F0 80  start C display
16 80 80
18 F0 50  start B display
1A 70 50
1C F0 50  start D display
1E 50 50
20 F0 80  start 5 display
22 F0 10  start 2 display

- 21 -
AB 2B 8B B8 else subtract 01 from timer, method used does not disturb the DF flag. DF is not changed by the interrupt routine.

Load R(8).0 tone duration, see FX18 instruction.

If tone duration is over go to 43.

Continue with or start tone.

Decrement R(8).0, tone duration.

Return, leaving tone on end of interpreter.

### Extending the CHIP-8 Instruction Set

The CHIP-8 interpreter is well organized and constructed and as a result it is easy to modify and extend. If a specific task, for example the control of a robot, is to be programmed the interpretive language can be changed to suit the application. Let's look at how we might extend the current CHIP-8 instructions. There are two main types of instructions one might wish to add, those which involve pointers to two of the CHIP-8 variables, (e.g. like 8XYN) and those which require a pointer to a single CHIP-8 variable (e.g. 6XKK).

The first group of instructions might be created by expanding either the 5XY0 instruction or the 9XY0 instruction. Say we chose to expand the 5XY0 instruction. The entry point for the 5XY0 instruction would be changed to point to a third CHIP-8 page. The least significant hex digit of the instruction would be examined and if it was 00 the instruction would have its usual meaning. However if the last hex digit was 1, 2, etc., the new operations would be performed.

As an example let's expand the 5XY0 instruction to the following set:

- **5XY0**
  - skip if VX=VY; the next interpreter instruction is skipped over if VX equals VY (original meaning).

- **5XY1**
  - skip if VX>VY; the next interpreter instruction is skipped over if VX is greater than VY.

- **5XY2**
  - skip if VX<VY; the next interpreter instruction is skipped over if VX is less than VY.

- **5XY3**
  - skip if VX≠VY; the next interpreter instruction is skipped over if VX does not equal VY.

We will place the new subroutines in the middle of page 0E between the interrupt routine and the bottom of the CHIP-8 stack. The entry point of the new interpreter subroutine will be 0E 70 (06 70 for the 1 1/4k Elf). CHIP-8 must be modified so that the 5 instructions transfer control to this address in the interpreter. Replace the 01 at location 00 55 with 0E (06 in the corresponding place for the 1 1/4k Elf) and replace the 98 at location 00 65 with 70.

### Additional Skip Instructions

#### Expansion of the 5XY0 Instruction

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>70</td>
<td>set R(C).1 to current page</td>
</tr>
<tr>
<td>72</td>
<td>45</td>
<td>load advance 2nd CHIP-8 byte, now VY and off 00, 01, 02, or 03 depending on instruction</td>
</tr>
<tr>
<td>73</td>
<td>FA</td>
<td>add starting address of table of locations</td>
</tr>
<tr>
<td>75</td>
<td>FC</td>
<td>point R(C) to proper entry in table</td>
</tr>
<tr>
<td>77</td>
<td>AC</td>
<td>pick up table entry, point R(C) to proper subroutine address</td>
</tr>
<tr>
<td>7A</td>
<td>07</td>
<td>load VY, make R(6) the X register</td>
</tr>
<tr>
<td>7C</td>
<td>DC</td>
<td>go to one of four subroutines</td>
</tr>
<tr>
<td>7D</td>
<td>81</td>
<td>address for 5XY0 instruction</td>
</tr>
<tr>
<td>7E</td>
<td>8B</td>
<td>address for 5XY1 instruction</td>
</tr>
<tr>
<td>7F</td>
<td>8F</td>
<td>address for 5XY2 instruction</td>
</tr>
<tr>
<td>80</td>
<td>87</td>
<td>address for 5XY3 instruction</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>entry for 5XY0</td>
</tr>
<tr>
<td>81</td>
<td>F3</td>
<td>x’or VX against VY</td>
</tr>
<tr>
<td>82</td>
<td>3A</td>
<td>return if D does not equal 00</td>
</tr>
<tr>
<td>84</td>
<td>15</td>
<td>15 D4 else skip and return</td>
</tr>
<tr>
<td>85</td>
<td>-</td>
<td>entry for 5XY3</td>
</tr>
<tr>
<td>87</td>
<td>F3</td>
<td>x’or VX against VY</td>
</tr>
<tr>
<td>88</td>
<td>3A</td>
<td>skip if D does not equal 00</td>
</tr>
<tr>
<td>8A</td>
<td>D4</td>
<td>else return</td>
</tr>
<tr>
<td>8B</td>
<td>F7</td>
<td>entry for 5XY1</td>
</tr>
<tr>
<td>8C</td>
<td>3B</td>
<td>84 8D subtract VX from VY</td>
</tr>
<tr>
<td>8E</td>
<td>D4</td>
<td>skip if DF equals zero</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>entry for 5XY2</td>
</tr>
<tr>
<td>8F</td>
<td>F5</td>
<td>subtract VY from VX</td>
</tr>
<tr>
<td>90</td>
<td>3B</td>
<td>84 8D skip if DF equals zero</td>
</tr>
</tbody>
</table>
Among the instructions that the interpreter lacks are simple multiply and divide instructions to go along with its addition and subtraction instructions. Let's expand the 9XY0 instruction to add these instructions to CHIP-8. Multiply and divide instructions are necessarily 16 bit ones, the product of two 8 bit numbers may be up to 16 bit long and of course we need 16 bits to represent the quotient and remainder from the division of two 8 bit numbers. An additional variable will be required to hold the most significant byte from a multiplication and the remainder form a division. VF is already a special variable and will be used to hold the most significant part of a product in a multiplication and the remainder in division. As well it would be nice to be able to represent the product of a multiplication as a decimal number and a 16 bit hex to decimal conversion routine will also be added.

The new "9" instructions will be located starting at the beginning of page 0D and we shall have to change the address of the "9" instructions in the interpreter. Memory location 00 59 should be changed from 01 to 0D and memory location 00 69 should be changed from 94 to 00.

The new instructions are:

9XY0 skip if VX≠VY; the next interpreter instruction is skipped over VX does not equal VY (unchanged)
9XY1 set VF, VX equal to VX times VY where VF is the most significant part of a 16 bit word
9XY2 set VX equal to VX divided by VY where VF is the remainder
9XY3 let VX, VY be treated as a 16 bit word with VX the most significant part and convert to decimal; 5 decimal digits are stored at M(I), M(I+1), M(I+2), M(I+3), and M(I+4), I does not change

Multiply, Divide and 16 Bit Display Instructions Expansion of 9XY0 Instruction

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D 00</td>
<td>93 BC</td>
<td>set R(C).1 to current page</td>
</tr>
<tr>
<td>02 45</td>
<td></td>
<td>load 2nd CHIP-8 byte, YN</td>
</tr>
<tr>
<td>03 FA 03</td>
<td></td>
<td>add off 00, 01, 02, or 03</td>
</tr>
<tr>
<td>05 FC 18</td>
<td></td>
<td>add starting address of table of locations</td>
</tr>
<tr>
<td>07 AC</td>
<td></td>
<td>point R(C) to proper entry in table</td>
</tr>
<tr>
<td>08 0C AC</td>
<td></td>
<td>pick up table entry, point R(C) to proper subroutine address before calling</td>
</tr>
<tr>
<td>0A E7</td>
<td></td>
<td>subroutine get ready for multiply and divide</td>
</tr>
<tr>
<td>0B 96 BE</td>
<td></td>
<td>R(7), VY pointer the X register</td>
</tr>
<tr>
<td>0D F8 FF AE</td>
<td></td>
<td>set VF to 00</td>
</tr>
<tr>
<td>10 F8 00 5E</td>
<td></td>
<td>clear DF flag</td>
</tr>
<tr>
<td>13 F6</td>
<td></td>
<td>initialize counter for shifts to 09</td>
</tr>
<tr>
<td>14 F8 09 AD</td>
<td></td>
<td>now call subroutines</td>
</tr>
<tr>
<td>17 DC</td>
<td></td>
<td>go to one of 4 subroutines</td>
</tr>
<tr>
<td>18 80</td>
<td></td>
<td>address for 9XY0 instruction</td>
</tr>
<tr>
<td>19 1C</td>
<td></td>
<td>address for 9XY1 instruction, multiply</td>
</tr>
<tr>
<td>1A 2D</td>
<td></td>
<td>address for 9XY2 instruction, divide</td>
</tr>
<tr>
<td>1B 46</td>
<td></td>
<td>address for 9XY3 instruction, hex to decimal conversion</td>
</tr>
<tr>
<td>1C 0E 76 5E</td>
<td></td>
<td>multiply routine entry, works by shift and add method like pencil and paper multiplication</td>
</tr>
<tr>
<td>1F 06 76 56</td>
<td></td>
<td>shift double length bit to the left</td>
</tr>
<tr>
<td>22 2D 8D</td>
<td></td>
<td>decrement and load counter</td>
</tr>
<tr>
<td>24 32 34</td>
<td></td>
<td>done when counted out</td>
</tr>
<tr>
<td>26 3B 1C</td>
<td></td>
<td>back if DF is 00, nothing to add</td>
</tr>
<tr>
<td>28 0E F4 5E</td>
<td></td>
<td>else add VY to VF,</td>
</tr>
<tr>
<td>2B 30 1C</td>
<td></td>
<td>before going back</td>
</tr>
<tr>
<td>30 F8 FF</td>
<td></td>
<td>end of multiply routine, begin divide routine - first check for division by zero</td>
</tr>
<tr>
<td>32 56 5E D4</td>
<td></td>
<td>load VF, subtract VY</td>
</tr>
<tr>
<td>35 0E F7</td>
<td></td>
<td>here if divisor greater than 0, division method similar to multiplication</td>
</tr>
<tr>
<td>37 3B 3A</td>
<td></td>
<td>load VF, subtract VY to 3A on overflow</td>
</tr>
</tbody>
</table>
If one has an ASCII device connected to an ELF, perhaps a keyboard, it would be convenient to have a CHIP-8 instruction which would create symbols for the characters in ASCII code. Such an instruction is presented last, the FX94 instruction. This instruction uses the space left unused in the interpreter by the expansion of the "5" and "9" instructions and creates symbols for the 64 characters in 6 bit ASCII. In operation it works exactly like the FX29 instruction except that the memory pointer is set to the address of one of the 64 ASCII symbols corresponding to VX instead of to the address of one of the 16 symbols 0-F. If the "5" and "9" instructions have not been expanded this instruction can, as well, replace the FX29 instruction and ways to implement either alternative will be given.

The instruction fits on a single page; each of the 64 ASCII symbols are coded by 3 bytes which requires 192 memory locations and the remainder of the subroutine fits in the 64 locations remaining. The construction of this instruction is quite simple. The first 16 locations on the page are patterns which are available to construct the symbols. Each ASCII symbol is designated by 5 hex digits which correspond to the patterns needed to construct the symbol. The sixth hex digit in the three words used to code each symbol serves as an indicator of the length of the symbol. When an FX94 (FX29) instruction is carried out this value is transferred to V0 where it can be used to get a pleasing spacing of the symbols.
The symbols are relatively crude, both because of the poor resolution of the ELF graphics and also because they consist of combinations of only 16 patterns. However they are easily recognized and make the presentation of ASCII data relatively with the aid of a very simple interpreter program.

The method used to transfer control from the interpreter to the new subroutine is to change the program counter from R(3) to R(C). This change has to be done in the interpreter and the address of the new subroutine must first be loaded to R(C). If the ASCII subroutine is located on page 0C the proper entry point is 0C D0. To make an FX94 instruction add the following code to the interpreter on page 01 (4k version):

```
01 94 F8 D0 AC point R(C).0 to D0
97 F8 0C BC point R(C).1 to page 0C
9A DC make R(C) the program counter
```

This code overwrites the locations which were used for the "5" and "9" instructions. The same code, but located starting at address 01 29, would change the FX29 instruction to the ASCII instruction.

**Six-Bit ASCII Symbols Subroutine**

```
Add. Code Notes
01 94 F8 D0 AC point R(C).0 to D0
97 F8 0C BC point R(C).1 to page 0C
9A DC make R(C) the program counter
```

A diagram giving the order in which the patterns are assembled from the bytes is:

```
XX XX XX
45 23 61
```

where the 6th hex digit contains the width of the character, at most 5 bits. The first ASCII character (hex 00) is @, its coding is 46, 3E, 56 which gives:

- pattern 6 - ******
- pattern 3 - * * *
- pattern E - ****
- pattern 4 - * * *
- pattern 6 - ******

The character is 5 bits long

```
00 - @
01 - A
02 - B
03 - C
04 - D
05 - E
06 - F
```

```
10 46 3E 56
A9 AC 49
3E 88 4F
4F 99 49
5B 22 53
6E CF 12 4F
6A 30 50
6D 06 00 50
73 00 00 40
7C 46 46 56
```

```
11 - Q
12 - R
13 - S
14 - T
15 - U
16 - V
17 - W
18 - X
19 - Y
00 - @
01 - A
02 - B
03 - C
04 - D
05 - E
06 - F
07 - G
```

```
08 - H
27 22 47
3A 22 52
40 89 4F
43 9F 9B 4F
46 A9 9F 4F
49 1F 8F 4F
4C 22 22 56
4F 9F 99 49
52 22 55 53
55 44 54
58 53 52 53
5B 22 52 53
5E CF 12 4F
61 8C 88 3C
64 10 C2 40
67 2E 22 3E
6A 30 25 50
6D 06 00 50
70 00 00 40
73 0C CC 2C
76 00 50 45
79 65 65 55
7C 46 46 56
```

```
10 - P
11 - Q
12 - R
13 - S
14 - T
15 - U
16 - V
17 - W
18 - X
19 - Y
```

A diagram giving the order in which the patterns are assembled from the bytes is:

```
XX XX XX
45 23 61
```

where the 6th hex digit contains the width of the character, at most 5 bits. The first ASCII character (hex 00) is @, its coding is 46, 3E, 56 which gives:

- pattern 6 - ******
- pattern 3 - * * *
- pattern E - ****
- pattern 4 - * * *
- pattern 6 - ******

The character is 5 bits long

```
00 - @
01 - A
02 - B
03 - C
04 - D
05 - E
06 - F
```

```
10 46 3E 56
A9 AC 49
3E 88 4F
4F 99 49
5B 22 53
6E CF 12 4F
6A 30 50
6D 06 00 50
73 00 00 40
7C 46 46 56
```

```
11 - Q
12 - R
13 - S
14 - T
15 - U
16 - V
17 - W
18 - X
19 - Y
```
The reader would probably like to see what these characters look like when displayed. Here is an interpretive program which can be used to display all of the ASCII symbols. The program waits for a switch byte (0-F) and when it is entered displays the corresponding ASCII symbol in the upper left of the screen followed by as many ASCII symbols as the screen has room for. If the byte in the interpreter (4K) at location 01 11 is changed from 0F to FF complete switch bytes (00-FF) can be entered.

Program to Display ASCII Characters

<table>
<thead>
<tr>
<th>Add.</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0200</td>
<td>F50A</td>
<td>V5 equals keys - waits for in button</td>
</tr>
<tr>
<td>02</td>
<td>6600</td>
<td>V6 = 00</td>
</tr>
<tr>
<td>04</td>
<td>6700</td>
<td>V7 = 00, display pointers</td>
</tr>
<tr>
<td>06</td>
<td>6B3F</td>
<td>VB = 3F, line length</td>
</tr>
<tr>
<td>08</td>
<td>F594</td>
<td>(F529?) set I to V5</td>
</tr>
<tr>
<td>0A</td>
<td>7501</td>
<td>ASCII symbol, V0 = symbol length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V5 = V5 + 01</td>
</tr>
</tbody>
</table>
display the symbol at V6, V7
V6 = V6 + V0
V6 = V6 + 01, space between symbols
VD = V6
(F529?) set I, V0 for next symbol
VD = VD + V0, add length of next symbol to VD
VD = VD - VB, check will it extend past line end?
skip if VF is 01, over the end of line
O.K. go back and display
reset to new line
V7 = V7 + 06, set line down
skip unless V7 is 1E, we're off bottom
stop - screen is full
return to do another line

It is hoped that these examples demonstrate the ease with which the CHIP-8 interpreter can be extended and modified. One of the limitations of CHIP-8, the fact that only memory locations 0000 through 0FFF are available to it, can be overcome by redesigning the interpreter to address memory in 4k fields. A field designation instruction is used to change from one 4k field to another. A relocatable 1k interpreter which includes all of the material presented in this booklet, as well as a field instruction, is listed in the Appendix. The field instruction is a four byte one which has the form, FFFF, MMMM. M is the new field and MMM is the address of the first instruction to be obeyed in the new field. For example to transfer to a new field:

<table>
<thead>
<tr>
<th>Add. Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0FD0 6300</td>
<td>set V3 to 0D</td>
</tr>
<tr>
<td>D2 6400</td>
<td>set V4 to 00</td>
</tr>
<tr>
<td>D4 650a</td>
<td>set V5 to 0A</td>
</tr>
<tr>
<td>D6 FFFF</td>
<td>field instruction go to</td>
</tr>
<tr>
<td>D8 1004</td>
<td>field 1, 004</td>
</tr>
<tr>
<td>10 04 F529</td>
<td>point to symbol for A</td>
</tr>
<tr>
<td>06 D345</td>
<td>display A</td>
</tr>
</tbody>
</table>

More ambitious programs can be written with the 4K memory restraint removed. The field designation is stored in R(B).0 and is set on entry
Appendix

The interpreter listed below is relocatable and can be placed on any four contiguous pages (e.g. 0A00 - 0DFF for 4k Elf). It must be entered with R(3) as the program counter. Enter at location 0000 for default values for the first interpreter instruction (01FE), the display page (0F), and the page for variables and constants (0E). To change the default values set R(5) to the address of the first interpreter instruction, set R(B).1 to the display page, set R(6).1 to the page for variables and constants, and enter the interpreter at location 000C. The default value for the location of the first interpreter instruction (01FE) allows space for an erase display instruction (00E0) before a program which starts at location 0200. The FX29 instruction in this interpreter does not alter the value of V0.

```
0000  F8  01  B5  F8  FE  A5  F8  0F
0008  BB  F8  0E  B6  95  FA  F0  AB
0010  96  B2  F8  CF  A2  E3  70  23
0018  93  B4  FC  02  B1  F8  D3  A1
0020  F8  25  A4  69  D4  96  B7  45
0028  AF  F6  F6  F6  F6  32  4D  FC
0030  69  AC  8F  F9  F0  A6  05  F6
0038  F6  F6  F6  F9  F0  A7  94  BC
0040  EC  F4  B3  8C  FC  0F  AC  0C
0048  A3  E2  D3  30  25  8F  32  54
0050  B3  45  30  48  94  FC  02  B3
0058  05  EB  EE  32  F6  0E  32
0060  64  8F  30  50  FC  05  FC  07
0068  30  48  01  01  02  02  02  02
0070  01  01  02  01  01  00  01
0078  01  7F  7B  1B  1F  27  23  00
0080  C4  4F  F3  AD  E1  88  96  05
0088  06  BE  FA  3F  F6  F6  F6  22
0090  52  07  FE  FE  FE  F1  AC  9B
0098  BC  45  FA  0F  AD  A7  F8  D0
00A0  A6  F8  00  AF  87  32  F7  27
00A8  4A  BD  9E  FA  07  AE  8E  32
00B0  BA  9D  F6  BD  8F  76  AF  2E
00B8  30  AE  9D  56  16  8F  56  16
00C0  30  A1  00  EC  F8  D0  A6  F8
00C8  00  A7  8D  32  F0  06  F2  2D
00D0  32  D5  F8  01  A7  46  F3  5C
00D8  02  FB  07  32  E9  1C  06  F2
00E0  32  E5  F8  01  A7  06  F3  5C
00E8  2C  16  8C  FC  08  AC  3B  CA
00F0  F8  FF  A6  87  56  12  D4  8D
00F8  A7  87  32  C2  2A  27  30  F9
0100  45  E6  F4  56  D4  45  A3  98
0108  56  D4  3F  0A  37  OC  22  6C
0110  FA  0F  12  56  D4  06  B8  D4
0118  06  A8  D4  64  0A  01  E6  8A
0120  F4  AA  3B  28  9A  FC  01  BA
0128  D4  F8  B0  30  8E  00  00  00
0130  15  15  D4  E6  06  BF  93  BE
0138  F8  1B  AE  2A  1A  F8  00  5A
0140  0E  F5  3B  4B  56  0A  FC  01
0148  5A  30  40  4E  F6  3B  3C  9F
0150  56  2A  2A  D4  00  22  86  52
0158  F8  F0  A7  07  5A  87  F3  17
0160  1A  3A  5B  12  D4  22  86  52
0168  F8  F0  A7  0A  57  87  F3  17
0170  1A  3A  6B  12  D4  E6  64  D4
0178  15  95  22  73  85  52  25  45
0180  A5  86  FA  0F  22  52  8B  F1
0188  B5  12  D4  00  F8  C0  AC  93
0190  FC  02  BC  DC  30  BC  22  6C
0198  06  F3  FA  0F  52  45  F6  42
01A0  3B  A7  3F  30  3A  30  D4  3F
01A8  AB  32  3D  D4  00  F8  F0  A7
01B0  47  45  F4  A5  86  FA  0F  3B
01B8  BB  FC  01  E2  22  52  8B  F1
01C0  B5  12  D4  00  45  FA  0F  3A
01C8  CC  07  56  D4  0F  22  F8  D3
01D0  73  8F  F9  F0  52  E6  07  D2
01D8  56  F8  FF  A6  F8  00  7E  56
01E0  D4  19  89  AE  93  BE  99  EE
01E8  F4  56  76  E6  F4  B9  56  45
01F0  F2  56  D4  45  AA  86  FA  0F
01F8  22  52  8B  F1  BA  12  D4  45
```
Notes

The FX00 and FX75 instructions cause failures when X is F because R(6) "turns" a page; R(6) should be decremented after the use of an output (64) instruction.

When using the relocatable interpreter place all the machine code subroutines in field 0 (0000 to 0FFF); they are accessible to calls from any of the 16 fields.
Additional copies of this booklet can be ordered from:
Paul C. Moews
16 B Yale Road
Storrs, CT 06268

The price, $5.50, includes first class postage and handling.
Two other booklets with programs for the basic 1/4k Elf are also available:
1. Music and Games
2. Graphics
for $3 each, postpaid.