readme.md

CHIP8IDE

Introduction

CHIP8IDE is an *interactive development environment* used to write programs for the *CHIP-8 virtual machine*.

CHIP-8 is a virtual machine architecture that was designed for a microcomputer kit released in 1977. Its purpose was to let hobbyists write game programs for their single-board kit computer without having to learn the details of the hardware. In the years since, many people have written emulators for the CHIP-8 so programs written for it could run on later hardware.

CHIP8IDE contains an emulator for executing CHIP-8 code, and it also offers tools for entry, editing, assembly, disassembly, and debugging CHIP-8 programs.

The CHIP-8 machine is programmed using an *assembly language*. That means you program the hardware by writing individual machine instructions to manipulate binary values in the machine's registers and memory. To create output you write instructions that draw pixels on the screen. To get input you write instructions that read the state of the keypad buttons. You control the flow of execution by writing conditional jump instructions.

Although the CHIP-8 virtual machine has many fewer parts and instructions than a modern CPU, the experience of writing code for it is very similar to writing assembly code for a real CPU like the Intel Core i7, ARMv8, or RISC-V. Learning to write and debug CHIP-8 code is excellent preparation for writing assembly code for a real machine.

There is a folder named `extras` included with CHIP8IDE. It contains binary and source programs, and these useful documents:

- `An_Easy_Programming_System.pdf` was a magazine article by the designer of CHIP-8. It clearly explains how the designer meant CHIP-8 to be used.
- `COSMAC_VIP_Manual.pdf` is part of the original manual that you would have received with your new microcomputer kit. It tells how to use CHIP-8 and gives a sense of what it was like to be a computer hobbyist forty years ago.
- `assembler_reference.pdf` is the documentation for the CHIP-8 assembly language. You will want to keep it handy whenever you are writing code.

Those documents provide a complete introduction to the CHIP-8 "machine" and how it is meant to be used. The rest of this document is about CHIP8IDE:

- User interface tells about the windows the app opens.
- Using the editor tells how to write, open, edit, assemble and save programs.
- Executing a program tells about viewing memory and registers and how to run and single-step programs.
- Display and keypad tells about using the emulated screen and keypad.
- History has more on the interesting history of CHIP-8 and SCHIP.
- Design summarizes the code modules of CHIP8IDE for readers.
- Sources acknowledges the many CHIP-8 internet sources that were plundered to make this app.

User interface

When you start it, CHIP8IDE opens three independent windows, the Edit window, the Memory window, and the Display window.

The *Edit window* contains a plain-text editor that you use to write or edit a CHIP-8 program. It supports all the usual editing keystrokes and operations. The Edit window also does syntax-checking and highlights any error statements. You can use the Edit window to set breakpoints so the machine stops when it reaches a certain statement. While you are single-stepping through a program, the Edit window highlights the source line that matches the next instruction to execute.

The *Memory window* displays the CHIP-8 machine memory and registers. It has the RUN/STOP button to start and stop execution, and a STEP button to execute a single instruction. When the machine is stopped you can see exactly what is in its memory and its registers. You can double-click on a memory byte or register and enter a new value in hexadecimal, and so alter the program or its execution.

The *Display window* contains the emulated screen output of the program as an array of square pixels. It has 32x64 pixels in CHIP-8 mode, or 64x128 pixels in SCHIP mode. (The program sets the mode when it executes.)

The Display window also presents the emulated 16-key keypad. When a program is running, it takes input from the keypad. The keys can be clicked with the mouse, or you can map keyboard keystrokes to the 16 keypad keys.

You can change the sizes and positions of the three windows. On a smaller screen you may want to overlap the Edit window and the Display window, because you rarely use them at the same time. The program remembers the window sizes and positions and restores them when it starts again.

Using the editor

You use the Editor to open programs, edit them, and load them into the emulated machine for execution.

Opening a source file

Start CHIP8IDE and select the File>Open command. Navigate to the `extras` folder that is distributed with the program. Navigate inside it to the `source` folder and open the file `draw_the_key.asm`. Now you are looking at a short CHIP-8 program in its source form. It consists of assembler language statements, comments, and white-space.

In the `docs` folder find the `assembler_reference.pdf` file and open it in another window. Read the text under the heading **Statements**. You can see examples of statements now in the Edit window.

Editing

In the Edit window, experiment with the normal edit keys for your system. You will find that you can edit this text with the same keys, like cut, copy and paste, and the same mouse actions, that you would use with NotePad on Windows, TextEdit on MacOS, or gedit on Linux.

You can also select text and drag it, and you can drag text from other apps and drop it on this window. That means if you have a favorite editor, you can use it when writing a longer program. Then just select the program text and paste it, or drag it, into the Edit window.

There is a Find/Replace dialog that comes up when you key *control-f* (*command-f* on Mac). After you find something with the dialog you can use these keys:

- Find next: *control-g* (*command-g*)
- Find previous: *shift-control-g* (*shift-command-g*)
- Replace: *control-=* (*command-=*)
- Replace and find again: *control-t* (*command-t*)

Assembling and errors

If you have changed the file while you were experimenting, get a fresh copy by opening `draw_the_key.asm`. (Do not save if CHIP8IDE asks if you want to save the modified file!)

Click the CHECK button below the Edit window. This button tells CHIP8IDE to *assemble* the statements (convert them into their binary executable form).

Now that you have assembled the program, CHIP8IDE can show you its binary form in the Edit window. Using the mouse or the arrow keys, move to different lines of the program in the Edit window. The address of each instruction and its binary form are displayed in hexadecimal below the window.

Let's make an error. On the first instruction, `LD I, QUERY`, change the opcode `LD` to `LX` which is not a recognized opcode. Hit the down-arrow to move to the next line. When the cursor leaves that line, the line turns red. The red color tells you that statement has some kind of error.

Move the cursor back onto the line. Below the window is the error message "Statement does not make sense". That is the general error statement used when some part of a statement can't be recognized. In this case, the assembler does not recognize `LX` as any sort of opcode or expression.

Change `LX` back to `LD` and move away from that line; it is no longer red. Go back and insert a plus sign after `QUERY` so it reads `QUERY+`. Move away again; the line is red. Move back; the error message is "Invalid expression near 7". Put the cursor at the left margin and hit the right-arrow 7 times to see where the invalid expression begins.

Without fixing the error, click the CHECK button. You get an error dialog telling you there is a "format" error, that is, an error in the format of some statement. Fix the error by removing that plus sign, and click CHECK again. Now the assembly completes without any messages.

Some errors cannot be recognized while you are entering or editing statements. They are only found at assembly time. Near the bottom of the program, change the label `QUERY:` to read `QUERYX:` (or anything that is not `QUERY:`). Move the cursor away from that line: it is not red, so there is nothing wrong with the form of that statement or any other.

Click the CHECK button and a message box tells you there is one "expression error" to correct. Click OK, and note that the `LD I, QUERY` statement is now red. What could be wrong with it? Click in that line and you see the error message "Symbol 'QUERY' is undefined". This statement refers to a name `QUERY` that ought to be defined on some other line. But it isn't! This is an error that should be easy to fix. Fix it and click CHECK again to verify that the program is good.

When you are writing a large program and CHECK reports errors, you can use the key *control-e* (*command-e*). This makes the Editor skip directly to the next line that has an error.

The Edit window has one more useful key, but it is explained in the next section.

Opening a binary file

You have been working with a *source* file, a text file with assembly language statements and (hopefully) comments. However, most CHIP-8 programs on the internet are *binary* files that contain only CHIP-8 instructions in their assembled form as binary bytes. CHIP8IDE can open these, too.

Select File>Open. (You have modified the `draw_the_key.asm` file, so you are asked if you want to save it. Say no!)

Navigate into the `binary` folder and open the file `heart_monitor.ch8`. Its contents appear in the Edit window, but they look quite different from the `draw_the_key.asm` source file. There are no readable comments or white space. This is a *disassembly*.

Each time CHIP8IDE opens a file, it tests to see if the file is entirely text lines encoded in ASCII or Latin-1. If the file is not just text, it assumes the file is a CHIP-8 executable file in binary.

CHIP8IDE *disassembles* the binary code, converting it back into assembly language source statements that express the same program, but which you can read and edit. The only comments in the disassembly are there to show the original hexadecimal values of each instruction.

Executing a program

Let's run the `heart_monitor.ch8` program. Click the LOAD button. This assembles the program, as the CHECK button does, but it also loads the assembled bytes into memory. Look at the Memory window.

Memory display

The CHIP-8 machine has 4,096 bytes of memory. The Memory window shows it in a large scrolling table, 32 bytes per row.

The assembled program has been loaded starting at address `#200` (hexadecimal 200, decimal 512). This is a rule for CHIP-8 programs. In the original computer kit, the machine-language code of the *emulator*, the program that implemented the CHIP-8 language, was always loaded into the first 512 bytes of memory, from location `#000` to `#1FF`. So a user program always started at the next available byte, `#200`. Today there is no emulator in those first 512 bytes. They are used instead to store the "sprites" (pixel patterns) for drawing letters and digits. But the convention remains that every program loads at `#200` and begins executing there.

Register display

The machine registers are displayed below the memory display. They have all been initialized to #00 except for the `PC` register. This is the Program Counter that tells the machine what instruction to execute next. It contains `200`, so the next instruction is the one at that address.

The instruction that `PC` points to is always highlighted in the memory display when the program is not running. (Notice that the two bytes at `#200` are highlighted.)

The other CHIP-8 registers are displayed in this row.

- Sixteen data registers `V0` through `VF`. Each can hold one eight-bit byte, an unsigned value from 0 to 255 (`#00` to `#FF`).
- The `I` register holds a memory address. It is used to point to data anywhere in the 4,096 bytes of memory.
- The `DT` or Delay Timer register holds an eight-bit value. When it is not zero, it is automatically decremented every "tick" or 1/60th of a second. Programs use it to time events.
- The `ST` or Sound Timer also holds an eight-bit value that is decremented every tick. When it is not zero, the tone sounds. Programs set it to small numbers to make "blip" noises and longer values to make "beeeeep" sounds.

Running a program

There is a small numeric field in the lower right of the Memory window. This is the "instructions per tick" counter. CHIP8IDE controls how many CHIP-8 instructions can execute in one "tick" (one-sixtieth of a second). Set this value to 10 for now.

Click the RUN! button. The program begins to execute. Its output is on the display window and it also makes beeping noises. (If the Display window is covered up, move it so you can see it.)

Click STOP to stop the program. Whenever you stop the machine, the instruction pointed to by the PC is highlighted in memory, and the source statement for that instruction is highlighted in the Edit window. Start and stop the program several times. Note the changes in the contents of the machine registers each time.

This program spends quite a bit of time in a subroutine. When CHIP-8 executes a `CALL` instruction, the `PC` value is pushed on the Call Stack. That shows where control will return when a `RET` is executed. Depending on where you stop this program, you may see a return address on the Call Stack display.

Start the program running and change the value in the inst/tick field. The more instructions you let it execute, the faster it seems to go. Slow it down to 1/tick. (Our patient is getting worse!) Somewhere around 10 to 15 is roughly the speed of the original microcomputer kit.

Single-step and breakpoints

Stop the program and then repeatedly click the STEP button. Each click executes just one CHIP-8 instruction. Note the register values changing, and how the current line in the Edit window follows along. This is one way to understand a strange program: step through its logic one instruction at a time.

In the edit window, find the `RET` instruction just above `LBL0256:` and click on it. Set a breakpoint by keying *control-b* (*command-b* on Mac). Click on another line and you see that this line now has a violet color, indicating it is a breakpoint.

Click RUN. The program stops immediately, and now the status line reads "Breakpoint on 00EE at 0254". Click RUN several times. Each time the program begins to run until it once again hits the breakpoint you set. Note how the registers and the Call Stack display change. You can tell this subroutine is called from two locations, because sometimes the value on the Call Stack is `#224` and sometimes `#21A`.

You can set as many breakpoints as you like. However, all breakpoints are cleared when click the LOAD button in the Edit window (because it reassembles the program).

There is one more thing to know about single step execution but it will be covered in the next topic.

Display and keypad

Open again open the source program `draw_the_key.asm` and click LOAD and RUN. This tiny program reads the next keypad key and displays that character, from `0` to `F`, on the screen.

Click on the different keypad keys. The program displays the key value. This is a good time to learn the key layout, how the keys for `A` to `F` are wrapped around the decimal numbers.

There are two ways to enter keypad clicks while a program is running. You can click the keys with the mouse, or you can use the keyboard of your computer. In the lower right of the display window is a pop-up menu. Each entry on the menu is a list of 16 keystrokes. These key*board* keys are mapped to the key*pad* keys of the display window. The default reads `1234qewradsfzxcv`. When it is selected, it maps the CHIP-8 keypad keys, from left to right and top to bottom, onto the keyboard keys:

```
1 2 3 4
 q w e r
  a s d f
   z x c v
```

In order for this to work, the Display window must have the "focus". Click on the edge of the Display window. Then type on those keyboard letters. The matching keypad keys should operate and the program should display the matching keypad letters.

There are two other lists of keys in the pop-up menu. Or you can select Enter New Map from the menu, and then enter a list of 16 keys of your choice.

Single step and the keypad

Click STOP to halt the program (`draw_the_key.asm`) that is running. The next instruction is at #203, `LD V5, K`. This instruction tells the machine to wait until a key is pressed, then load its number into register `V5`, then wait until the key is released before going to the next instruction.

Click the STEP button a few times. Nothing happens. The machine is waiting to find a key pressed. But you can't click on STEP and also click on a keypad key at the same time! So there is a problem trying to single-step through a load-keyboard instruction or any other instruction that tests the keypad.

This is solved by "latching" a key. Go to the Display window and *shift-click* on a key, for example hold the Shift key and click on the D key. The key is highlighted and remains highlighted afterward. It is "latched down" and will remain pressed until the machine samples it.

Now you can go back to the Memory window and click STEP. The `LD V5, K` instruction finds a key down, and it can complete. Click STEP again and note the value `0D` in register V5.

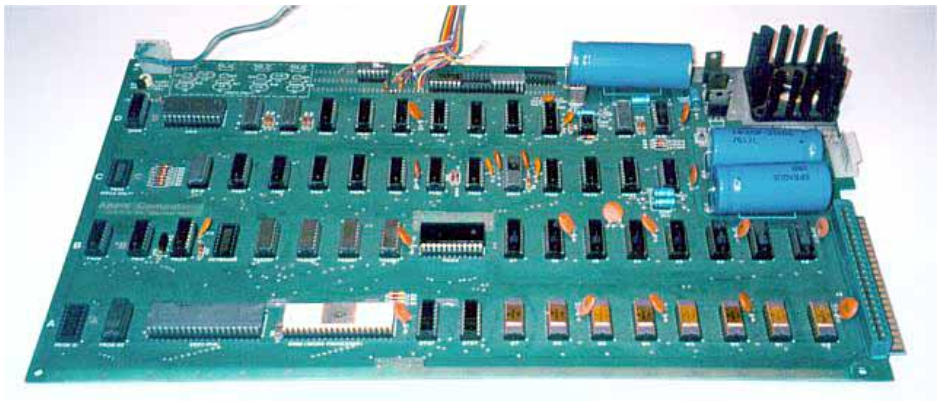
You now have the tools to load, edit, run and debug CHIP-8 programs. Look for the file `CONTENTS` in the `extras` folder. It lists all the programs and documents in that folder, with suggestions on how some of them might be improved.

History

The personal computer industry was born in the years 1974-77. In 1974, Intel announced the 8080 chip. In Albuquerque, Ed Roberts read about it, realized it could be the basis of a cheap minicomputer, and designed the Altair 8800 kit around it. It was featured on the cover of the January 1975 issue of *Popular Electronics*.

Two Harvard undergrads, Bill Gates and Paul Allen, were so inspired by that article that they dropped out of school, moved to Albuquerque, and in April 1975 they founded a software company then called Micro-Soft.

Two months later, June 1975, Steve Wozniak first booted a prototype of what would be the Apple I, which he and his pal Steve Jobs began selling in June 1976. Wozniak used the MOS Tech 6502 microprocessor chip, largely because of its lower cost.

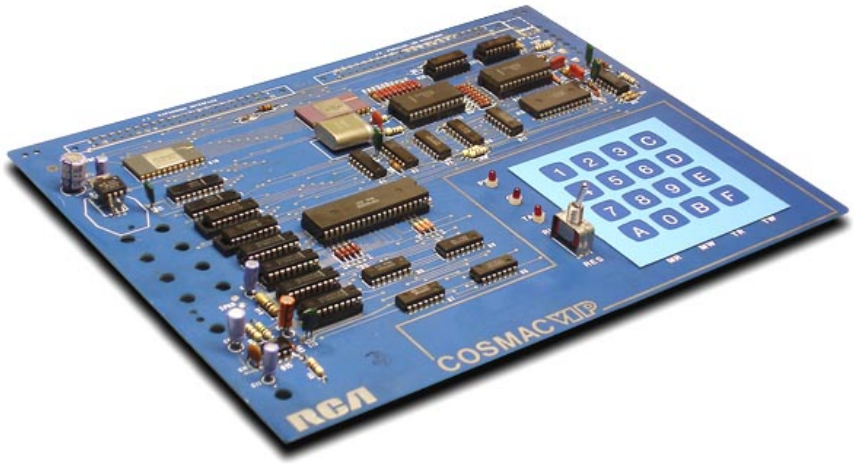


Apple I single-board computer sold for $666.

Birth of the VIP

Meanwhile on the East Coast, the RCA Corporation had developed a series of microprocessor chips, largely under the direction of a multi-talented engineer named Joseph Weisbecker. In 1976 RCA released the 1802, a CPU with a different manufacturing process and architecture that RCA called "complementary symmetry monolithic array computer", or COSMAC. (The 1802 is still manufactured today for use in high-reliability embedded systems.)

Weisbecker, who had already designed a number of game and hobby boards, now designed a new single-board hobby kit around this CPU and called it the COSMAC VIP.



COSMAC VIP single-board computer kit assembled and ready for use.

Weisbecker realized that hobbyists could not be expected to begin programming the kit in 1802 machine language. First, the only means of entering code was to key in hexadecimal digits with the keypad, and second, the only means of output was to write binary commands to the separate chip that controlled the television interface.

Weisbecker set out to flatten the steep learning curve by designing a much simpler virtual machine that he dubbed CHIP-8. The CHIP-8 machine was still programmed by entering machine instructions as hexadecimal digits, but it had many fewer and simpler instructions than a "real" microprocessor. Most important, it had single instructions for reading the keypad and for writing a pattern of pixels, called a "sprite", onto the screen.

Weisbecker hoped that the CHIP-8 architecture and language were simple enough that users could get enjoyment out of creating simple game programs for the VIP. He first described CHIP-8 in the manual that shipped with the kit when it was released in 1977. (This manual is included in the `extras` folder.) A year later, an article by Weisbecker was published in *BYTE* magazine describing CHIP-8, including a sample game program. (A copy of this article is also in the `extras` folder.)

The COSMAC VIP sold well enough to create an active community of users who kept in touch through a fanzine, *VIPer*, which published 39 issues between 1978 and 1984. You can read all the issues of *VIPer* at Matthew Mikolai's Archive site. They contain a number of contributed programs for the VIP.

The HP-48 and SCHIP

In the years following, many people wrote extended versions of CHIP-8 for various hardware platforms.

In 1990, Hewlett-Packard released the HP-48 Graphing Calculator. It had a black on white screen of 131x64 pixels. Andreas Gustafsson wrote a CHIP-8 emulator for this machine so that users could write and play the kinds of simple games that the CHIP-8 language made easy. He added support for the double-size 64x128 display.

In 1991, Erik Bryntse extended Gustafsson's emulator with instructions for scrolling the screen and support for higher-resolution character sprites. This extension became known as Superchip, or SCHIP. This extended version is supported by CHIP8IDE.

Undocumented instructions

There are three original sources of documentation for CHIP-8: the VIP manual, the BYTE article, and issue 1 of *VIPer*. All omit to mention three instructions that were supported by Weisenbecker's emulator code: shift-left, shift-right, and exclusive-or of machine registers.

It is impossible to know if that omission was intentional or a mistake. Perhaps the instructions were added in a last-minute update before the VIP shipped.

At any rate, they were soon discovered by fans who reverse-engineered the 512-byte machine language emulator program out of curiosity. The first description of the added instructions appeared in *VIPer* issue 2 in a letter to the editor from Peter K. Morrison that describes the operation of the emulator. (The same issue has another analysis of the emulator code, with flowcharts. These were diligent hobbyists!)

Morrison's letter correctly describes the operation of the SHR and SHL instructions: the value from the source register, V*s*, is shifted one bit left or right and the result is placed in the target register V*t* (`Vt = Vs<<1` and `Vt = Vs>>1`).

Somehow this became lost over time and an incorrect interpretation has propogated online which assumes that the value was shifted in-place (`Vs = Vs<<1` etc). This may have been because most programs that used the instructions *intended* for the shift to happen in-place, so they coded the same register number for V*t* and V*s* (e.g. `SL V5,V5`). At least two of the emulators I've looked at have this incorrect implementation, and CowGod's influential CHIP-8 Technical Reference also has it wrong.

The correct interpretation of the shift instructions was clarified in a recent exchange at the Yahoo VIP Group.

CHIP8IDE design

This app is written in Python 3 and uses the PyQt5 and Qt5.x packages for its user interface. All functions and global variables have PEP 484-style type hints and the program has been checked by the mypy type hint checker.

You are welcome to read the code on Github. If you see a way to improve it, by all means enter a pull request!

These are the source modules in the order you might want to read them.

- chip8ide.py is the top level module. It sets up logging and the Qt environment, loads all the other modules to initialize themselves, and starts the Qt Application event loop.
- chip8.py has the actual emulator. For reasons explained in the comments, it is written "Pascal-style", with all subroutines first. The actual code to decode and execute a CHIP-8 instruction appears almost a thousand lines in.
- source.py implements the Edit window. It is mostly PyQt5 class definitions to implement the window and the widgets inside it, interesting only if you need to understand Qt. One key part is the Syntax Highlighter which is entered after every single user edit action to check for errors in the statement.
- assembler1.py inspects a line every time the user does an edit. It uses regular expressions to "tokenize" the line; then verifies that it makes sense as an assembler statement.
- assembler2.py is called when you click CHECK or LOAD. It performs the actual assembly, converting opcodes and expressions into binary values.
- disassemble.py performs the disassembly of a binary file. It is interesting to compare this to `chip8.py` ; they both decode the same binary instruction values, one to execute them and one to convert them to source.
- display.py implements the Display window with its emulated screen and keypad. Like `source.py` it is a whole lot PyQt5 class definitions.
- memory.py is the code that actually runs the emulator. The bulk of it is code to create and manage three Qt Tables using Qt's Model-View architecture. The tables display memory, the call stack, and the registers. Down at the bottom is the asynchronous QThread that runs when you click the RUN button, so the CHIP-8 emulator can go full speed while Qt still handles mouse clicks and keyboard actions.

All this code is written in "literate" style, with a narrative about what the code is doing interspersed with the Python statements that actually do it. If you wonder why something is being done a certain way, there is probably a long boring explanation (or an apology!) in the comments somewhere.

Sources

The internet is just stuffed with information about the CHIP-8, including many well-written CHIP-8 emulators. These are some of the sources I read and stole ideas from:

- CHIP-8 article on Wikipedia with overview and links.
- The COSMAC VIP page at OldComputers.net has good pictures of the original machine.
- A full online copy of the original COSMAC VIP manual documents the entire single-board computer in detail, including its hardware and ROM. The VIP manual distributed with CHIP8IDE was extracted from this source. (Another online copy is incomplete, lacking most of the pages with CHIP-8 game listings.)
- BYTE magazine for December 1978 (large PDF) contains the article by Joseph Weisbecker describing CHIP-8. A copy of this article is distributed with CHIP8IDE.
- CowGod's CHIP-8 page is probably the most frequently cited internet reference, and covers both CHIP-8 and SCHIP features (but is incorrect on the shift instructions).
- David Winter's CHIP-8 emulation page has a trove of CHIP-8 games and emulators.
- Mastering CHIP-8, an essay by Matthew Mikolay, has a good technical description of the CHIP-8 but omits the SCHIP features.
- Matthew Mikolay's Retrocomputing page has documents, some program sources, and PDF copies of all issues of the fanzine *VIPer*.
- *VIPer* issue 1 has a lengthy review of the CHIP-8 instructions with a tutorial.

There are many CHIP-8 emulators to be found around the internet. Writing one is a favorite project for people studying computer science. I have seen several of these, but in creating CHIP8IDE I have taken explanations, ideas, and some code from following:

Hans Christian Egeberg wrote CHIPPER, a CHIP-8/SCHIP assembler with output to a binary file for execution in the HP48 calculator. It is still available from the HP Calculator archives. The C source of CHIPPER is completely uncommented, but the CHIPPER.DOC file included with it has a good review of the instructions.

David Winter created a CHIP-8 emulator for MS-DOS. His CHIP-8 page has the source of Egeberg's CHIPPER, and the source and/or executable of a number of CHIP-8 and SCHIP games. Although the source of his own emulator CHIP8.EXE is not included, his CHIP8.DOC file also documents the instruction set. I have used the games from this page as test vehicles and have included some of them in the `extras` folder.

Craig Thomas's Chip8Python is an elegantly coded CHIP-8 emulator in Python. I followed his method of dispatching decoded instructions. (But his does SHR/SHL incorrectly.)

"Mudlord" (Brad Miller) wrote a CHIP-8/SCHIP emulator in C++ which I have consulted. (It too has the SHR/SHL instructions wrong.)

Jeffrey Bian wrote MOCHI-8, a CHIP-8 assembler, disassembler, and emulator, all in Java. I have followed his assembly language syntax, and I referred to his code and documentation in creating the assembler and disassembler in this project. (But his emulator also has the SHL/SHR instructions wrong.)

Github user "mir3z" has a JavaScript CHIP-8 emulator which I did not actually read. However, the rom folder at that site is stuffed with many S/CHIP-8 executable files.