DEBUG8

DEBUG8 is a debugger for CHIP-8, based on my CHIP8 interpreter. While fairly basic, it has an assembler, disassembler, sprite editor, multiple breakpoint types, executed instruction history, and more. I wrote it after a friend told me he was going to write a CHIP-8 interpreter (which I hadn't heard of before), and it looked interesting. It's been a while since I've had so much fun writing code.

Like my CHIP8, DEBUG8 is written in C with inline assembly, probably in the most useless way possible. What makes it useless is that it targets a DOS machine to run on -- as an old assembly language programmer, I have a LOT of experience with programming down to the bare metal of a DOS machine and lots of code to draw from.

As a result, on modern machines it needs to be run in DOSBOX (www.dosbox.com). This makes it an interpreter running in an interpreter.

SPEED

Nevertheless, it's quite fast -- on an i7-4770 with DOSBOX running at full speed, non-graphics instructions run at about 1 MIPS (1 million instructions per second). Graphics instructions are slower, but the worst case of constantly drawing a 16 x 16 sprite in Super CHIP-8 graphics (128 x 64) (which updates video memory then re-writes the entire graphics screen) runs at about 1300 instructions per second.

A "mixed instruction set" program like BLINKY.CH8, at full speed, runs at about 43,000 instructions per second.

Speeds like that make most CHIP-8 programs unplayable, so part of the design for my CHIP8 interpreter was an instruction execution clock. I used a default of 2040 instructions per second (2040 Hz) for a couple of reasons:

1) About 2000 Hz sounded like a good target.
2) The DOS environment only has 1 timer available, and I needed to have 3 time bases derived from it:
    a) 2000 Hz execution clock
    b) 60Hz time base for delay timer and sound timer
    c) 18.2Hz

Doing the math, 2040 / 60 = 34 (update the 60 Hz timer every 34th execution tick), and 2040 / 18.2 = 112 (update the system timer every 112th execution tick).

The program allows the user to change both execution ticks per instruction cycle and number of instructions executed every execution cycle. So, for example, changing the execution ticks per instruction cycle to 4 gives a rate of 510 instructions per second. And changing the execution ticks per instruction cycle to 8 and number of instructions executed to 3 would give a rate of 765 instructions per second.

SYSTEM

The important locations in system memory are the sprites and the stack.
The low-res (5 x 5) sprites are at hex locations:

0 : 80
1 : 85
2 : 8A
3 : 8F
4 : 94
5 : 99
6 : 9E
7 : A3
8 : A8
9 : AD
A : B2
B : B7
C : BC
D : C1
E : C6
F : CB

The high-res (8 x 10) sprites are at hex locations:

0 : D0
1 : DA
2 : E4
3 : EE
4 : F8
5 : 102
6 : 10C
7 : 116
8 : 120
9 : 12A
A : 134
B : 13E
C : 148
D : 152
E : 15C
F : 166

(Yes, I know high-res sprites are normally only 0 to 9, but I threw in the sprites for the rest of the hex values too).

The stack (16 levels) is from 1C0 to 1E0.

Instruction Set Problems

There are inconsistencies in the documentation for CHIP-8 instructions.

The Cowgod CHIP-8 technical reference (http://devernay.free.fr/hacks/chip8/C8TECH10.HTM) says that the instructions
  SHL Vx,Vy
and
  SHR Vx,Vy
shift the Vx register, Vy is not used. It also does not say that the I register is updated in the LD [I], Vx and LD Vx, [I] instructions.

Another source, Matthew Mikolay (http://mattmik.com/files/chip8/mastering/chip8.html) says that the instructions
  SHL Vx, Vy
and
  SHR Vx,Vy
shift the Vy register and store the result in the Vx register. It also states that the LD [I], Vx and LD Vx, [I] instructions do update the I register.

Another source, Wikipedia (https://en.wikipedia.org/wiki/CHIP-8) notes that the ADD I, Vx instruction sets CARRY in VF, but this is undocumented.

The problem is that **ALL** the various conflicting definitions are "correct" -- there are programs that rely on a particular implementation to run correctly.

For example, the program BLINKY.CH8 (https://github.com/stianeklund/chip8/blob/master/roms/BLINKY.ch8) relies on the Cowgod definition of SHL and SHR, using Mikolay's definition breaks the program.

The program EATY.CH8 (https://johnearnest.github.io/chip8Archive/play.html?p=eaty) rely on Mikolay's definition of SHL and SHR, using Cowgod's definitions breaks the program.

The program Spacefight 2091! (https://github.com/michaelarnuats/chip8-java/tree/master/Binary/roms) relies on the CARRY out of the ADD I, Vx instruction.

DEBUG8 has both behaviors for each instruction type programmed, controlled by flags that can be set / cleared in the the debugger. The default is the Cowgod definitions and no CARRY from ADD I, Vx (I used BLINKY.CH8 as my standard test while developing DEBUG8).

COMMAND LINE

There are command line parameters available in DEBUG8, mostly leftovers from its CHIP8 origin. The parameters can also be set in the debug
interface.

Command line parameters (numbers are DECIMAL):

-F : execution Freerun (run code as fast as possible)
-Tn : Execution ticks (2040 ticks/second) to wait before executing CHIP8 instructions. Default is 1.
-In : CHIP8 instructions to execute every time execution tick timer expires. Default is 1.
-K : Use Alternate keypad (1..4/q..r/a..f/z..v) instead of PC keypad.
-V : Default High-res video (128 x 64) instead of standard video(64 x 32)
-Sn : Sound frequency. Default is 330Hz. Range is 100 to 1000.
-Ln : Load address for program. Valid values are 512 and 1536. Default value is 512.
-W : Wrap sprites that go beyond the right edge of the screen. Default is to truncate sprites at the right edge of the screen.
-M : Use Mikolay's instruction behaviors (equivalent to using both -X and -Y options).
-X : SHR Vx,Vy and SHL Vx,Vy -- shift Vy register and store it in Vx
Default is that Vx is shifted.
-Y : LD [I],Vx and LD Vx,[I] instructions DO update the I register.
Default is that they DO NOT update the I register.
-Z : ADD I,Vx DOES set carry on overflow
Default is that carry is NOT set on overflow.
program : If given, program will be loaded.

DEBUGGING COMMANDS

One important note on breakpoints: All breakpoints are "break before execution of the instruction that would encounter the breakpoint." Think about a bad pointer over-writing memory. It's nice to stop before the memory value is over-written.

While debugging, all numbers typed and displayed will be in HEX. Parameters in square brackets are OPTIONAL.

Commands are:

G : Go

Run CHIP8 program. The Esc key will stop the interpreter.

The syntax for the G command is

```
G [=pc] [start1[,end1][=count1]] ... [start5[,end5][=count5]]
```

A starting PC can be given, and if valid the PC will be changed before the program starts running.

A list of breakpoints (up to 5) can also be entered in the format:

```
start[,end[=count]]
```

A single breakpoint can cover a range of addresses, from start to end. Also, a "hit" count may be added, so that the breakpoint happens only when count references to the specified address / range have been made.

For example, a 'G' command could look like

```
g =222 100,111=7 200,222=8 300,333=9 400,444=a 500,555=b
```

Breakpoints are only valid for the duration of the G command. If you want "sticky" breakpoints, see the B command below.

P : Proceed

The same as the Go command, but a breakpoint is automatically set after the current instruction. This breakpoint does not count against the 5 allowed breakpoints.

This is typically used to "jump" over a CALL instruction.

T : Trace program

Run the program one instruction at a time. Both keypad and sound are disabled. "Sticky" breakpoints are not active.

The syntax for the T command is

```
T [=pc] [number_of_instructions_to_trace]
```

A starting PC can be given, and if valid the PC will be changed before the program starts running.
The syntax is

```
=pc
```

A count of instructions to trace can also be given, and the program will be traced through that many instructions. Use the Esc key to stop early.

For example, a 'T' command could look like

```
t =222 1000
```

Since "sticky" breakpoints are not active, it's convenient to trace over a breakpoint and then G to continue execution.

The system timer tick is not captured during Trace, so neither DT (Delay Timer) or ST (Sound Timer) will be updated.

B : Sticky breakpoints

Manage sticky breakpoints. Sticky breakpoints are not erased after a Go or Proceed command, although the "hit" count is reset every time a Go or Proceed command is used.

The syntax for the B command is

B
to toggle stickky breakpoints ON / OFF or
B LIST
to show the current sticky breakpoints or
B ["instr"] [start1[,end1][=count1]] ... [start5[,end5][=count5]]

If no parameters are given, the current sticky breakpoints are toggled ON / OFF, and the new status is displayed.

There is a special parameter, "list", that will list the current sticky breakpoints. It must be the first parameter, and if found all other parameters will be ignored.

Setting sticky breakpoints with the B command will clear the current list -- the current list cannot be edited or added to. If you want to retain current breakpoints and add a new location, the entire list must be entered.

In addition to memory breakpoints, a single breakpoint on instruction type may be set by enclosing the instruction name in double-quotes. The instruction string may be as specific as you want. For example, both
"ld"
and
"drw v7,ve,9"
are valid breakpoint strings. The first will stop at any LD instruction, while the second stops on a very specific DRW instruction.

A breakpoint on instruction type must be the first parameter to the B command.

A list of breakpoints (up to 5) can also be entered in the format:
start[,end][=count]
A single breakpoint can cover a range of addresses, from start to end. Also, a "hit" count may be added, so that the breakpoint happens only when count references to the specified address / range have been made.

For example, a B command could look like
b "call 284" 100,111=7 200,222=8 300,333=9 400,444=a 500,555=b

U : Unassemble (Disassemble)
Yes, 'D' for Disassemble would make more sense, but D is being used for Dump memory, and I was being lazy.

The syntax for the U command is
U [starting_pc [number_of_instructions]]

Unassemble (Disassemble) program. At the start of the debugging session, the default staring location is the program load address (200 hex for standard programs, 600 hex for ETI-660 programs). After that, the default location is the next instruction after the last disassembled instruction.

Optionally, the starting location can be given. It must be a valid address (inside memory).

The default is to disassemble 16 (10 hex) instructions. The number to disassemble can be given, but MUST be the second parameter. If you are going to specify the number of instructions, you MUST specify the starting address.

The disassemble command cannot go beyond the end of memory. If the default number of instructions are used near the end of memory, it will be automatically adjusted to the end of memory. If the number of instructions is given and it would go beyond the end of memory, that is an error.

For example, a U command could look like
u 456 5

A : Assemble

Assemble statements to binary.

The syntax for the A command is

A [starting_pc]

At the start of the debugging session, the default staring location is the program load address (200 hex for standard programs, 600 hex for ETI-660 programs). After that, the default location is the next instruction after the last assembled instruction.

The starting location can be given. It must be a valid address (inside program memory).

As a prompt, the assemble location is displayed, and the user enters the instruction. Pressing Esc will erase the current input. If an invalid instruction is entered, it is flagged and the assembly PC does not increment.

When a valid instruction is entered, it is put into memory and the next instruction is prompted for.

End the assembly with a blank line (press Return as the first character).

S : Edit sprite

Edit memory (assumed to be a sprite) with a graphical sprite editor.

The syntax for the S command is

S memory_address num_rows [template_address]

The debug window will be cleared for the sprite editor, and restored after exiting the editor.

In the editor, use the keypad (7,8,9,4,6,1,2,3) to move the cursor in the corresponding direction. Use either the 5 on the keypad or the spacebar to toggle the current pixel ON / OFF.

Also, you can use the I key to Invert the sprite (flip all bits to make a "negative" of the sprite), the C key to clear the sprite, or the F key to Fill the sprite (all pixels on).

Use the S key to save the edited sprite and exit the editor. Use the Esc key to exit the editor without saving the sprite.

The edited sprite may be anywhere in memory, not just program memory. This allows you to edit the default system sprites.

The sprite may be between 1 and 16 rows (1 and 10 hex). A 16 row sprite is also 16 columns wide.

The sprite editor also allows you to enter a location for a template, which is copied as the default sprite image.

If you want to use a template, you must specify the number of columns, even if it is the default value.

For example, the command

s 400 5 80

would edit a sprite in user memory that is 5 rows high, 8 bits wide,
and is a copy of the system 5 x 5 sprite '0' as its initial image.

D : Dump memory

The syntax for the D command is
  D [memory_address [len]]

At the start of the debugging session, the default staring location is the program load address (200 hex for standard programs, 600 hex for ETI-660 programs). After that, the default location is the next address after the last dumped memory. The default length is 32 bytes (20 hex).

The starting location and length can be specified. If the start address of the dump plus length is beyond the end of memory, it is an error. All memory, including the system area starting at 0, can be dumped, not just program memory.

The starting location is specified first, then the length. Even if you want the default location, if you want to specify a length you must specify the starting location.

R : Change Register value

The syntax for the R command is
  R reg_name value

Where reg_name is V0 .. VF, PC, I, SP, DT, or ST.

The value must be appropriate for the register. V0 .. VF, DT, and ST must be in the range 0 .. FF, SP must be EVEN and in the range 1C0 to 1E0, PC must be in the program memory, and I must be in memory (0 .. FFF).

E : Enter value(s) into memory

The syntax for the E command is
  E starting_address value1 [value2 [value3 ...]]]

The starting address must be in program memory. If the starting address plus number of bytes is beyond the end of memory, it is an error.

F : Fill memory with value(s)

The syntax for the F command is
  F starting_address ending_address value1 [value2 [value3 ...]]]

The starting and ending addresses must be in program memory, and the ending address must be greater than or equal to the starting address.

The sequence of values is repeated until the specified memory is filled.

W : Where was I?

The syntax for the W command is
  W

Show the last 16 (10 hex) instructions (or number of instructions since program start, whichever is less) executed.

L : Load program

The syntax for the L command is
L [program_name]

Load the CHIP-8 program and reset the interpreter. If no program name is given, the current program is reset.

When the interpreter is reset the PC, SP, assemble, disassemble, and dump addresses are reset. The graphics are also cleared.

C : Create program (write current program to a file)

The syntax for the C command is
C program_name [start_address [end_address]]

If the starting address is not given, the start of program memory will be used. If the ending address it not given, the end of memory will be used. Note that writing extra memory does not have any effect on a program.

If the program name already exists, you will be prompted to make sure you want to over-write it.

Note that the saved program is only program memory, not system memory. If you have modified the system sprites and want to save them, you need a copy in program memory.

SAVE : Save debugger state

The syntax for the SAVE command is
SAVE filename

Save the current memory (all of it, including system memory (in case you edited the system sprites)), graphics, registers, and internal variables that contain the current state of the interpreter.

This allows you to debug multiple times starting from a known state, or come back another day and continue your debugging session.

Parameters like "Freerun" or which SHR / SHL instructions to use are not saved.

LOAD : Load debugger state

The syntax for the LOAD command is
LOAD filename

Load the current memory (all of it, including system memory (in case you edited the system sprites)), graphics, registers, and internal variables that contain the current state of the interpreter.

This allows you to debug multiple times starting from a known state, or come back another day and continue your debugging session.

Parameters like "Freerun" or which SHR / SHL instructions to use are not loaded.

H : Help. Show brief descriptions of commands

Q : Quit the debugger.

The command line parameters can also be set or modified in the debugger.
Note that values need to be given in HEX, not the decimal of the command line.

-F : Toggle execution Freerun (run code as fast a possible)

The syntax for the -F command is
-F

In execution Freerun, instructions are run without regards to the execution timer ticks or instructions per execution cycle, so that code runs as fast as possible.

See the -T and -I commands for details on execution timer ticks and instructions per execution cycle.

-T : Set ticks per execution cycle

The syntax for the -T command is
-T [cycle_count]

DEBUG8 uses a 2040 Hz "tick" as an instruction execution timebase. When the 2040Hz "tick" counter equals the ticks per execution cycle count, "instructions per execution cycle" are executed.

The cycle count cannot be 0.

If no parameter is given, the current execution cycle tick count is displayed.

By using both ticks per execution cycle and instructions per execution cycle, a wide range of program execution speeds can be set. For example, with ticks per execution cycle set to 3 and instructions per execution cycle set to 2, the interpreter will run at
(2040 / 3) * 2 = 680 * 2 = 1360 IPS.

-I : Set instructions per execution cycle

The syntax for the -I command is
-I [instruction_count]

Every execution cycle, execute the specified number of instructions.

The instructions per execution cycle cannot be 0.

If no parameter is given, the current instructions per execution cycle is displayed.

By using both ticks per execution cycle and instructions per execution cycle, a wide range of program execution

speeds can be set. For example, with ticks per execution cycle set to 3 and instructions per execution cycle set to 2, the interpreter will run at
(2040 / 3) * 2 = 680 * 2 = 1360 IPS.

-K : Toggle normal / alternate keypad

The syntax for the -K command is
-K

The normal keypad is the PC keypad, the alternate keypad is
1..4/q..r/a..f/z..v

-V : Toggle startup mode High-res / Standard video

The syntax for the -V command is
-V

The default is Standard video (128 x 32) at program start. -V will toggle between the standard and High-res video (256 x 64) at program start. The new mode is displayed.

-S : Sound freqency

The syntax for the -S command is
-S frequency

The default for system sound is 330Hz (14A hex). The valid range is 100Hz (64 hex) to 1000Hz (3E8 hex).

-L : Load address for program

The syntax for the -L command is
-L address

Address must be either 200 (512 decimal, standard CHIP-8 load location) or 600 (1536 decimal, ETI660 load location).

When the load location is given, a full system reset takes place (clear memory and video, reset registers and defaults for Assemble, Unassemble, Dump).

-W : Toggle Wrap Sprites

The syntax for the -W command is
-W

Toggle whether sprites wrap or truncate if they go beyond the edge of the screen. The default to truncate sprites.

-M : Toggle instruction behaviors

The syntax for the -M command is

-M

Toggle between:
Mikolay's instruction behavior:
SHR Vx, Vy -- shift Vy register and store it in Vx
SHL Vx, Vy -- shift Vy register and store it in Vx
LD [I], Vx -- increment I after storing registers
LD Vx, [I] -- increment I after loading registers
Cowgod's instruction behavior:
SHR Vx, Vy -- shift Vx register, Vy not used
SHL Vx, Vy -- shift Vx register, Vy not used
LD [I], Vx -- I unchanged after storing registers
LD Vx, [I] -- I unchanged after loading registers

This is the same as using the -X and -Y commands together.

-X : Toggle SHL / SHR instruction behaviors

The syntax for the -X command is
-X

Toggle between:
Mikolay's instruction behavior:
SHR Vx, Vy -- shift Vy register and store it in Vx
SHL Vx, Vy -- shift Vy register and store it in Vx
Cowgod's instruction behavior:
SHR Vx, Vy -- shift Vx register, Vy not used
SHL Vx, Vy -- shift Vx register, Vy not used

-Y : Toggle I update

The syntax for the -Y command is
-Y

Toggle between:
Mikolay's instruction behavior:
LD [I], Vx -- increment I after storing registers
LD Vx, [I] -- increment I after loading registers
Cowgod's instruction behavior:
LD [I], Vx -- I unchanged after storing registers
LD Vx, [I] -- I unchanged after loading registers

-Z : Toggle carry from ADD I,Vx

The syntax for the -Z command is
-Z

Toggle between:
Undocumented behavior:
ADD I, Vx -- if I + Vx > 0x0FFF, set VF to 1 else set VF to 0

Default behavior

ADD I, VX -- does not change VF