Emulation and Chip-8

Level 0x0f

Quick Overview

- Events
- Chip-8 Background
- Emulation Development
  - Instruction Decoder
  - Disassembler
  - Emulation
- Next Steps

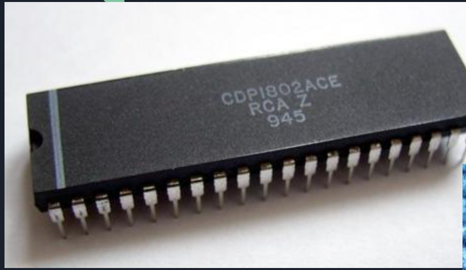
Events

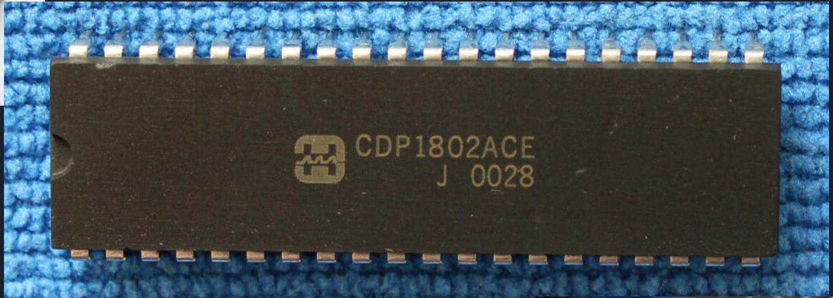
- Code Quest Pictures
    - Lockheed Smug Mug
- Last Meeting
    - April 25, 2025
    - Pizza Party
- Binja Keys

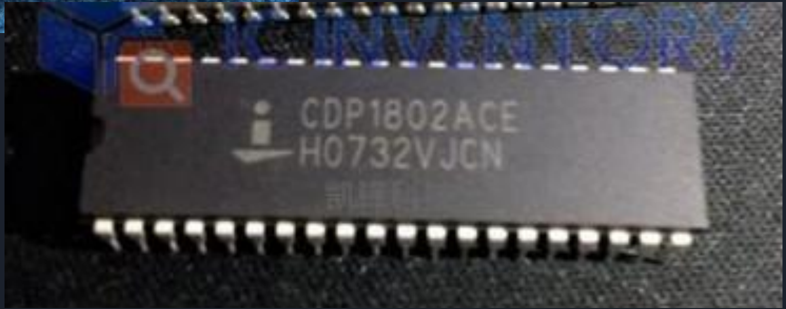
RCA COSMAC 1802 CPU

- Age of early microcomputers / computer kits (1977)
  - COSMAC ELF
  - COSMAC VIP
    - $2.8K on ebay
  - Telmac 1800
  - RCA Studio II
    - $100-$200
- Many of these kits shipped with Chip-8 interpreters and code listing for Chip-8 games
- Pretty obsolete technology right…

CPU Flavors / Identification



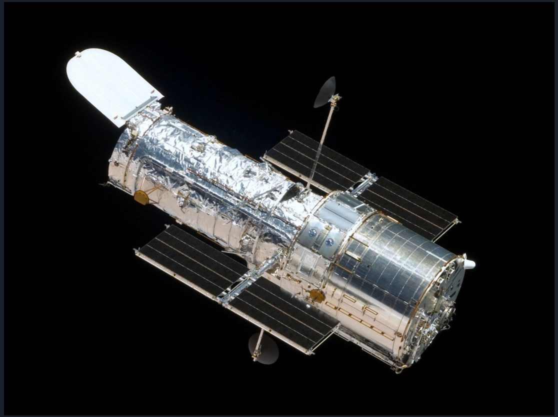




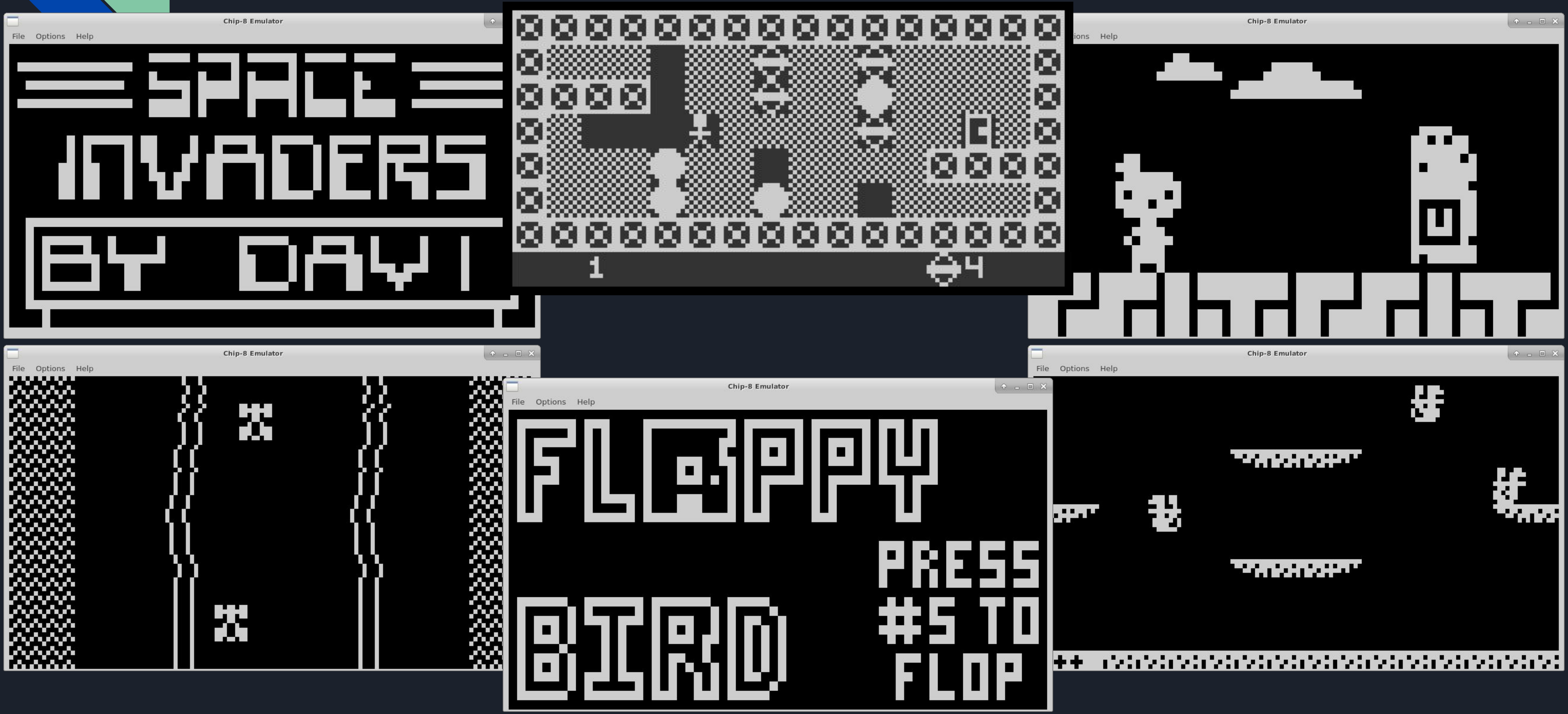
CDP 1802 Status Today

- RCA → GE → Harris Corp → Intersil → Renesas
- RAD Hardened at Sandia National Labs
- Used in many spacecraft
  - Hubble Space Telescope
  - Galileo

- Still in availability @ $146 per 1k in bulk
- Less than $5 on ebay for used parts

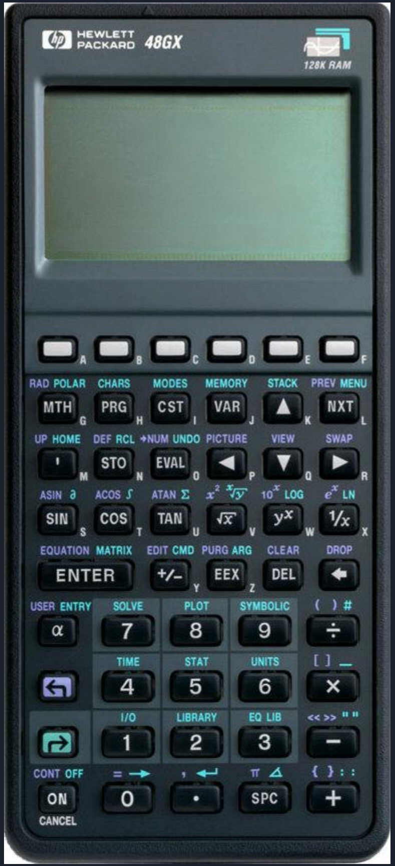


Chip-8 Screenshots



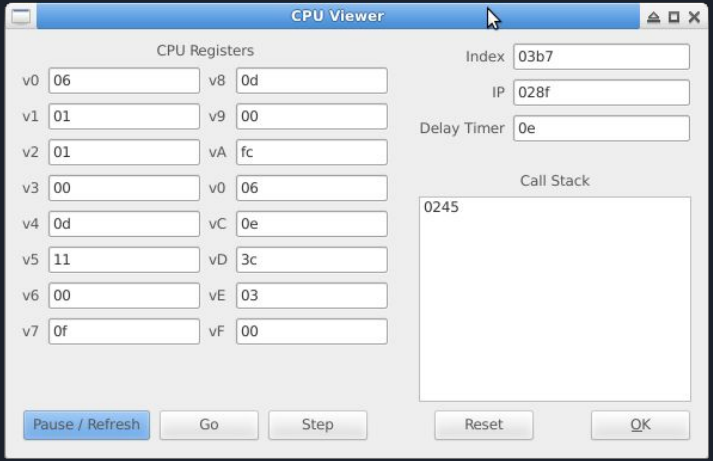
Chip-8 Resurgence in the 90s

- Mid-90s HP Graphing Calculators
  - HP 48SX: Saturn CPU 2.0 MHz 32KB RAM
  - HP 48G: Saturn CPU 3.6 MHz 32KB RAM
  - HP 48GX: Saturn 4.0 MHz 128KB RAM
- Compare to original GameBoy
  - 4.19MHz CPU
  - 8KB S-RAM, 8KB Video RAM
  - ROMs from 256KB+
- Chip-8 Interpreter released for HP
  - Interpreter was small, just over 2KB
  - Chip-8 games were small (500 – 3K typical)
  - Chip-8 assembly is pretty easy
- S-CHIP released with enhanced instructions to take advantage of HP hardware



Chip-8 Specifications

- 4KB RAM (0x0 - 0xFFF)
- 8-bit CPU with 34 2-byte instructions
- 16 8-bit registers (v0 – vF)
- 1 16-bit register (address / index)
- 2 timers (delay and sound) @ 60Hz
- Display is 64x32, B/W
    - Sprites are 8x15 maximum
- Call stack
    - Only stores return addresses
    - >= 16 addresses
- Instruction Pointer



CPU Registers

v0 06
v1 01
v2 01
v3 00
v4 0d
v5 11
v6 00
v7 0f
v8 0d
v9 00
vA fc
v0 06
vC 0e
vD 3c
vE 03
vF 00

Index 03b7
IP 028f
Delay Timer 0e

Call Stack

0245

Instruction (Load, Math)

Op Code	Mnemonic	Operation
8xy0	LD Vx, Vy	Vx = Vy
6xkk	LD Vx, kk	Vx = kk
Annn	LD I, addr	I = addr
Fx55	LD [I], Vx	[I] = V0, ...
Fx65	LD Vx, [I]	V0 = [I], ...
7xkk	ADD Vx, kk	Vx += kk
8xy4	ADD Vx, Vy	Vx += Vy
Fx1E	ADD I, Vx	I += Vx

Op Code	Mnemonic	Operation
8xy1	OR Vx, Vy	Vx |= Vy
8xy2	AND Vx, Vy	Vx &= Vy
8xy3	XOR Vx, Vy	Vx ^= Xy
8xy5	SUB Vx, Vy	Vx -= Vy
8xy7	SUBN Vx, Vy	Vx = Vy - Vx
8x06	SHR Vx	Vx >>= 1
8x0E	SHL Vx	Vx <<= 1

Instructions (Branching)

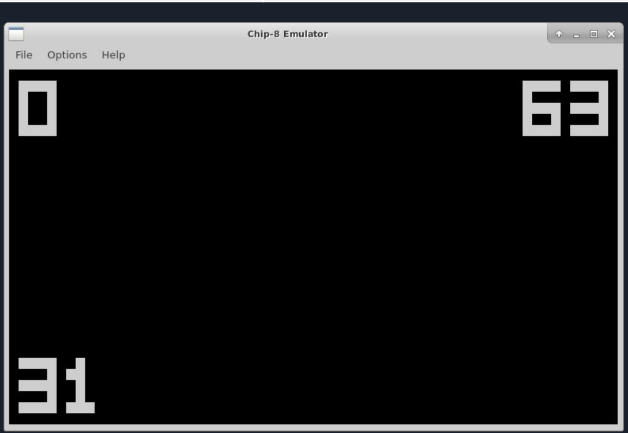
Op Code	Mnemonic	Operation
1nnn	JP nnn	PC = nnn
2nnn	CALL nnn	Push PC; PC = nnn
00EE	RET	Pop PC
3xkk	SE Vx, kk	Skip next inst if Vx == kk
4xkk	SNE Vx, kk	Skip next inst if Vx != kk
5xy0	SE Vx, Vy	Skip next inst if Vx == Vy
9xy0	SNE Vx, Vy	Skip next inst if Vx != Vy
Ex9E	SKP Vx	Skip next inst if Vx == key pressed
ExA1	SKNP Vx	Skip next inst if Vx != key pressed
Bnnn	JP V0, nnn	Jump with variable offset

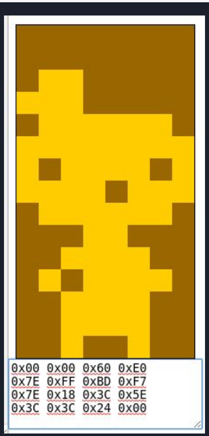
Instructions (Misc)

Op Code	Mnemonic	Operation
Cxnn	RND Vx, nn	Vx = random & nn
Fx0A	LD Vx, Key	Vx = key pressed (blocking)
Fx15	LD DT, Vx	DT = Vx (counts down 60Hz)
Fx18	LD ST Vx	ST = Vx (buzzes while != 0)
Fx07	LD Vx, DT	Vx = DT
Fx33	LD B, Vx	Converts Vx in base 10. [I] = 100s, [I+1] = 10s, [I+2] = 1s

Instructions (Graphics)

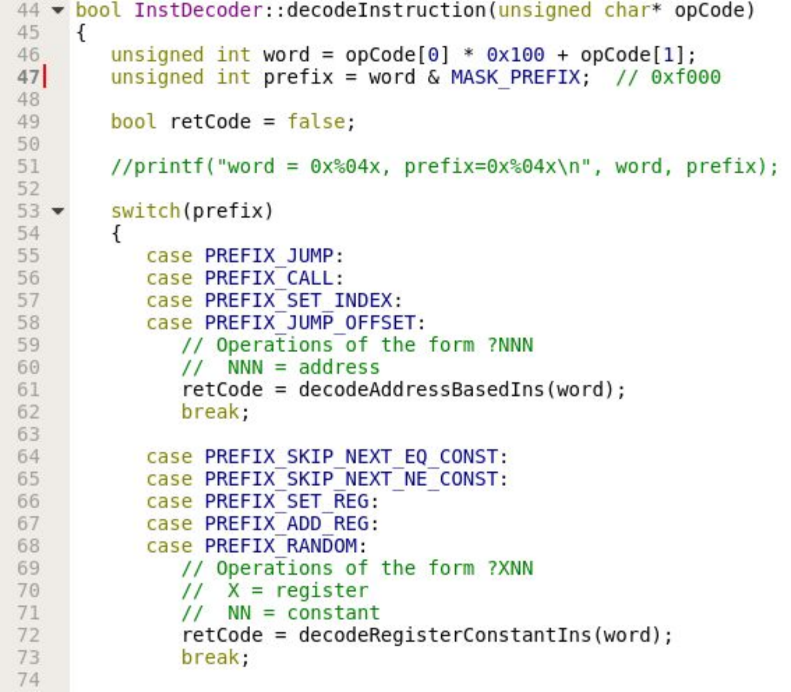
Op Code	Mnemonic	Operation
Fx29	LD F, Vx	I = addr of sprite character Vx
Dxyn	DRW Vx, Vy, n	Draw Sprite from memory location I at Vx,Vy. n = sprite height
00E0	CLS	Clear Screen



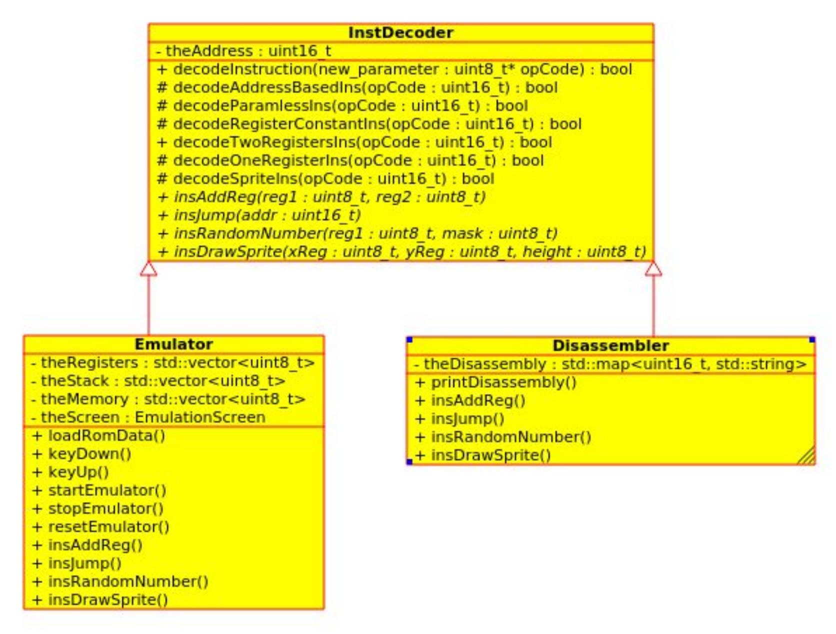


```
0x00 0x00 0x60 0xE0
0x7E 0xFF 0xBD 0xF7
0x7E 0x18 0x3C 0x5E
0x3C 0x3C 0x24 0x00
```

Instruction Decoding



```cpp
bool InstDecoder::decodeInstruction(unsigned char* opCode)
{
    unsigned int word = opCode[0] * 0x100 + opCode[1];
    unsigned int prefix = word & MASK_PREFIX;  // 0xf000

    bool retCode = false;

    //printf("word = 0x%04x, prefix=0x%04x\n", word, prefix);

    switch(prefix)
    {
        case PREFIX_JUMP:
        case PREFIX_CALL:
        case PREFIX_SET_INDEX:
        case PREFIX_JUMP_OFFSET:
            // Operations of the form ?NNN
            //  NNN = address
            retCode = decodeAddressBasedIns(word);
            break;

        case PREFIX_SKIP_NEXT_EQ_CONST:
        case PREFIX_SKIP_NEXT_NE_CONST:
        case PREFIX_SET_REG:
        case PREFIX_ADD_REG:
        case PREFIX_RANDOM:
            // Operations of the form ?XNN
            //  X = register
            //  NN = constant
            retCode = decodeRegisterConstantIns(word);
            break;

```

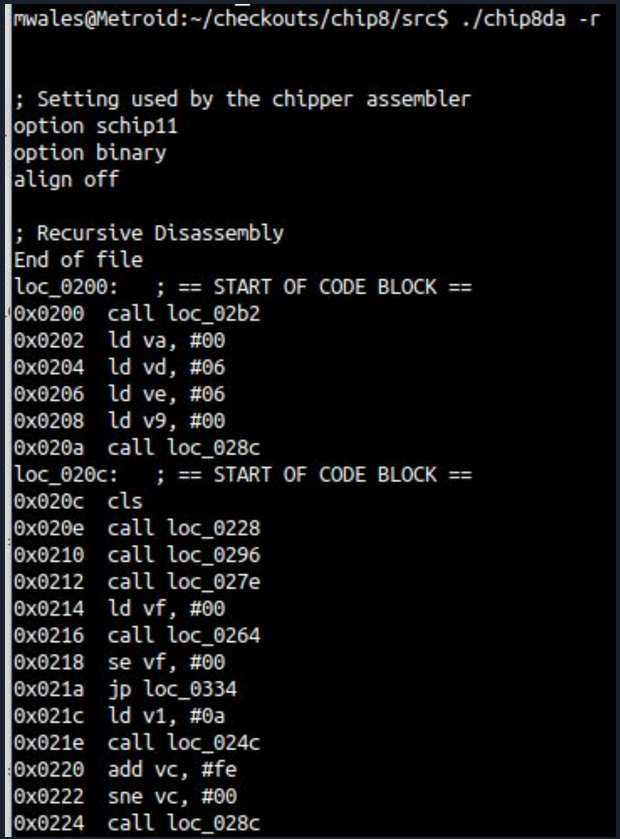


```
InstDecoder
- theAddress : uint16_t
+ decodeInstruction(new_parameter : uint8_t* opCode) : bool
# decodeAddressBasedIns(opCode : uint16_t) : bool
# decodeParamlessIns(opCode : uint16_t) : bool
# decodeRegisterConstantIns(opCode : uint16_t) : bool
+ decodeTwoRegistersIns(opCode : uint16_t) : bool
# decodeOneRegisterIns(opCode : uint16_t) : bool
# decodeSpriteIns(opCode : uint16_t) : bool
+ insAddReg(reg1 : uint8_t, reg2 : uint8_t)
+ insJump(addr : uint16_t)
+ insRandomNumber(reg1 : uint8_t, mask : uint8_t)
+ insDrawSprite(xReg : uint8_t, yReg : uint8_t, height : uint8_t)

Emulator
- theRegisters : std::vector<uint8_t>
- theStack : std::vector<uint8_t>
- theMemory : std::vector<uint8_t>
- theScreen : EmulationScreen
+ loadRomData()
+ keyDown()
+ keyUp()
+ startEmulator()
+ stopEmulator()
+ resetEmulator()
+ insAddReg()
+ insJump()
+ insRandomNumber()
+ insDrawSprite()

Disassembler
- theDisassembly : std::map<uint16_t, std::string>
+ printDisassembly()
+ insAddReg()
+ insJump()
+ insRandomNumber()
+ insDrawSprite()
```

Write Disassembler

- Easy to test and debug decoder
- Create helpers to pull opcode apart
- Disassembly printout
- Simple printfs

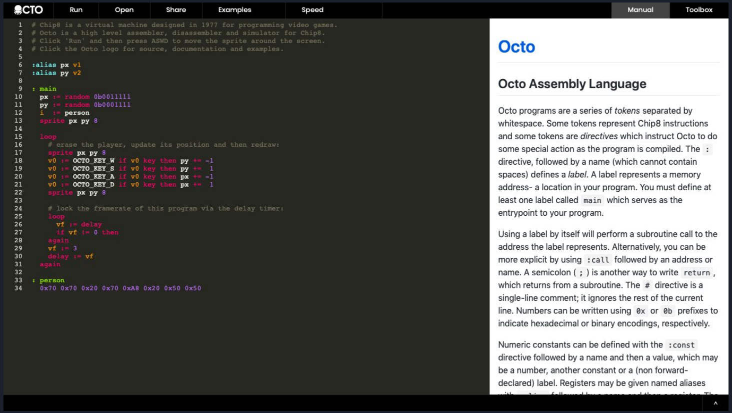
  `printf(“add v%X, v%X\n”, regX, regY);`
- Recursive vs Linear disassembly
  - What if program jumps to odd address…



```
mwales@Metroid:~/checkouts/chip8/src$ ./chip8da -r


; Setting used by the chipper assembler
option schip11
option binary
align off

; Recursive Disassembly
End of file
loc_0200:   ; == START OF CODE BLOCK ==
0x0200  call loc_02b2
0x0202  ld va, #00
0x0204  ld vd, #06
0x0206  ld ve, #06
0x0208  ld v9, #00
0x020a  call loc_028c
loc_020c:   ; == START OF CODE BLOCK ==
0x020c  cls
0x020e  call loc_0228
0x0210  call loc_0296
0x0212  call loc_027e
0x0214  ld vf, #00
0x0216  call loc_0264
0x0218  se vf, #00
0x021a  jp loc_0334
0x021c  ld v1, #0a
0x021e  call loc_024c
0x0220  add vc, #fe
0x0222  sne vc, #00
0x0224  call loc_028c
```

Testing

- Write simple test cases
  - Test basic operations (add, load, store)
  - Single step through CPU execution
- Chipper assembler for Chip-8 (on David Winters page)
- Ambiguous Implementation Details
  - How do you think it should work?
  - Design test case
  - Test on real hardware (other emulators)
- Reference test cases
  - BestCoder test roms for Chip-8
  - NES 6502 has many well known test cases
  - Avoid using full ROMS / games initially



```
# Chip8 is a virtual machine designed in 1977 for programming video games.
# Octo is a high level assembler, disassembler and simulator for Chip8.
# Click 'Run' and then press ASWD to move the sprite around the screen.
# Click the Octo logo for source, documentation and examples.

:alias px v1
:alias py v2

: main
  px := random 0b0011111
  py := random 0b0001111
  i  := person
  sprite px py 8

  loop
    # erase the player, update its position and then redraw:
    sprite px py 8
    v0 := OCTO_KEY_W if v0 key then py += -1
    v0 := OCTO_KEY_S if v0 key then py +=  1
    v0 := OCTO_KEY_A if v0 key then px += -1
    v0 := OCTO_KEY_D if v0 key then px +=  1
    sprite px py 8

    # lock the framerate of this program via the delay timer:
    loop
      vf := delay
      if vf != 0 then
    again
    vf := 3
    delay := vf
  again

: person
  0x70 0x70 0x20 0x70 0xA8 0x20 0x50 0x50
```

Octo

Octo Assembly Language

Octo programs are a series of *tokens* separated by whitespace. Some tokens represent Chip8 instructions and some tokens are *directives* which instruct Octo to do some special action as the program is compiled. The `:` directive, followed by a name (which cannot contain spaces) defines a *label*. A label represents a memory address- a location in your program. You must define at least one label called `main` which serves as the entrypoint to your program.

Using a label by itself will perform a subroutine call to the address the label represents. Alternatively, you can be more explicit by using `:call` followed by an address or name. A semicolon (`;`) is another way to write `return`, which returns from a subroutine. The `#` directive is a single-line comment; it ignores the rest of the current line. Numbers can be written using `0x` or `0b` prefixes to indicate hexadecimal or binary encodings, respectively.

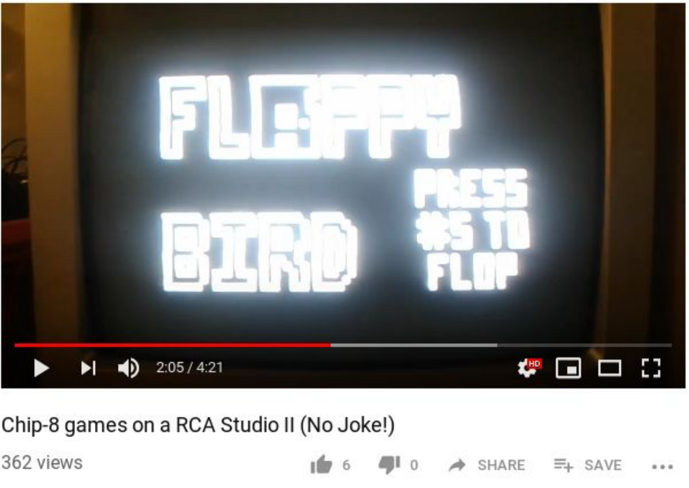
Numeric constants can be defined with the `:const` directive followed by a name and then a value, which may be a number, another constant or a (non forward-declared) label. Registers may be given named aliases

Adding Graphics / Keyboard

- SDL
  - Multiplatform C Gaming Library
  - Traditionally used for OSX and Linux ports
  - Graphics, Inputs, Audio, Threading / Locking
- Qt
  - Powerful multiplatform C++ Toolkit
  - Traditionally used for widget based GUIs
  - Threading / Locking, Containers
- SFML
- OpenGL
- ImGUI

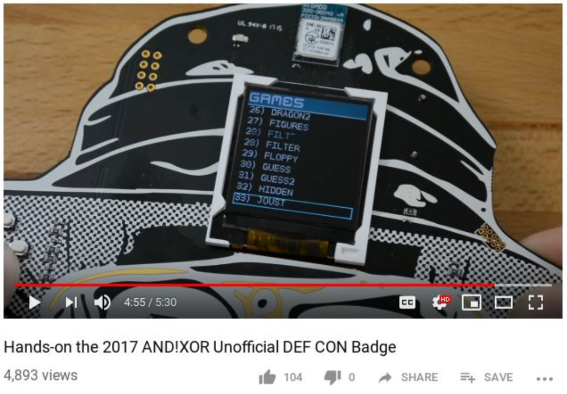
Make your own game!

- Octojam
- Held during October
- Entries run in Octo online emulator
- Get published!?



Chip-8 games on a RCA Studio II (No Joke!)

362 views



Hands-on the 2017 AND!XOR Unofficial DEF CON Badge

4,893 views

Attributions

- https://en.wikipedia.org/wiki/Telmac_1800
- https://github.com/JohnEarnest/Octo?tab=readme-ov-file
- https://johnearnest.github.io/chip8Archive/
- https://github.com/Timendus/chip8-test-suite
- https://github.com/mwales/chip8