Chip-8 User Manual

CreateAssembly.app (2021)

Table of Contents

Instruction Set Architecture

00E0 - CLS

Clears the display.

00EE - RET

Returns from the current subroutine.

```
PC = STACK[SP];
 SP = SP - 1;
```

00EE - RET

Returns from the current subroutine.

```
PC = STACK[SP];
 SP = SP - 1;
```

0nnn – SYS addr

Unused.

1nnn – JP addr

Jump to specified address.

```
PC = addr;
```

2nnn – CALL addr

Increment the stack pointer, store the program counter in the stack, then jump to given address.

```
 SP = SP + 1;
STACK[SP] = PC;
  PC = addr;
```

3xkk – SE Vx, byte

Skip the next instruction if the value stored in Vx is equal to the given number.

```
  If (Vx == byte)
Then {PC = PC + 2};
```

4xkk – SNE Vx, byte

Skip the next instruction if the value stored in Vx is not equal to the given number.

```
  If (Vx != byte)
Then {PC = PC + 2};
```

5xy0 – SE Vx, Vy

Skip the next instruction if the value stored in Vx is equal to the value stored in Vy.

```
If (Vx == Vy)
Then {PC = PC + 2};
```

6xkk – LD Vx, byte

Store the given value in register Vx.`Vx = byte;`

7xkk – ADD Vx, byte

Add the given value into register Vx.

```
Vx = Vx + byte;
```

8xy0 – LD Vx, Vy

Store the value of Vy in Vx.

```
Vx = Vy;
```

8xy1 – OR Vx, Vy

Bitwise-OR Vy into Vx.

```
Vx = Vx | Vy;
```

8xy2 – AND Vx, Vy

Bitwise-AND Vy into Vx.

```
Vx = Vx & Vy;
```

8xy3 – XOR Vx, Vy

Bitwise-XOR Vy into Vx.

```
Vx = Vx ^ Vy;
```

8xy4 – ADD Vx, Vy

Add Vy into Vx. Set VF to 1 if there is an overflow, or 0 otherwise.

```
VF = (Vx + Vy) > 255;
Vx = Vx + Vy;
```

8xy5 – SUB Vx, Vy

Subtract Vy from Vx and store the result in Vx. Set VF to 0 if there is a borrow, or 1 otherwise.

```
VF = Vx > Vy;
Vx = Vx - Vy;
```

8xy6 – SHR Vx [, Vy]

Shift Vx right by 1. Vy is deprecated and unused.

```
Vx = Vx >> 1;
```

8xy7 – SUBN Vx, Vy

Subtract Vx from Vy and store the result in Vx. Set VF to 0 if there is a borrow, or 1 otherwise.

```
VF = Vy > Vx;
Vx = Vy - Vx;
```

8xyE – SHL Vx [, Vy]

Shift Vx left by 1. Vy is deprecated and unused.

```
Vx = Vx << 1;
```

9xy0 – SNE Vx, Vy

Skip the next instruction if Vx does not equal Vy.

```
  If (Vx != Vy)
Then {PC = PC + 2};
```

Annn – LD I, addr

Set register I to the given address.

```
I = addr;
```

Bnnn – JP V0, addr

Jump to the value in V0 + the given address.

```
PC = V0 + addr;
```

Cxkk – RND Vx, byte

Store a random value, bitwise-AND'd with the given byte, in Vx

```
Vx = rand(0, 0xFF) & byte;
```

Dxyn – DRW Vx, Vy, nibble

Load n-bytes starting at MEM[I]. XOR them onto the display (each byte occupies its own row) starting at (Vx, Vy). Will wrap horizontally and vertically if necessary. Set VF to 1 if there is a collision, or 0 otherwise.

```
    Load n-byte sprite at MEM[I];
XOR sprite onto the display at (Vx, Vy);
           VF = collision;
```

Ex9E – SKP Vx

Skip the next instruction if the value stored in Vx corresponds to a key that is currently pressed.

```
 If (KEY[Vx] down)
Then {PC = PC + 2};
```

ExA1 – SKNP Vx

Skip the next instruction if the value stored in Vx corresponds to a key that is not currently pressed.

```
  If (KEY[Vx] up)
Then {PC = PC + 2};
```

Fx07 – LD Vx, DT

Load the value of the delay timer into Vx.

```
Vx = DT;
```

Fx0A – LD Vx, K

Block until a key is pressed. Store which key was pressed in Vx.

```
Block until key down;
      Vx = key;
```

Fx15 – LD DT, Vx

Load the value of Vx into DT.

```
DT = Vx;
```

Fx18 – LD ST, Vx

Load the value of Vx into ST.

```
ST = Vx;
```

Fx1E – ADD I, Vx

Add the value of Vx into I.

```
I = I + Vx;
```

Fx29 – LD F, Vx

Store the location of the Vx'th sprite in I. The exact location may vary from emulator to emulator. However, most sprite data starts at MEM[0] and each sprite has a length of 5 bytes.

```
I = &SPRITE[Vx];
```

Fx33 – LD B, Vx

Store the BCD representation of the given register in memory, starting at address I.

```
  MEM[I] = floor(Vx / 100);
MEM[I+1] = floor(Vx % 100 / 10);
    MEM[I+2] = Vx % 10;
```

Fx55 – LD [I], Vx

Store contents of registers V0 through Vx in memory, starting at address I.

```
MEM[I+0] = V0;
MEM[I+1] = V1;
MEM[I+2] = V2;
      …
MEM[I+x] = Vx;
```

Fx55 – LD [I], Vx

Load x bytes of data from memory, starting at address I, and store them in registers V0 through Vx

```
V0 = MEM[I+0];
V1 = MEM[I+1];
V2 = MEM[I+2];
      …
Vx = MEM[I+x];
```

Assembler

Numbers

Representation

```
; All of the following instructions load 19 into their respective registers

        LD      V0, 19          ; Decimal
        LD      V1, 0x13        ; Hexadecimal
        LD      V3, 0b10011     ; Binary
        LD      V5, 0o23        ; Octal
```

Overflow

All numbers and registers are unsigned. If an operand is assigned an out-of-range constant *at assembly-time*, there will be an error. If a register goes out of range during run time, there will be an integer overflow (or underflow) and no error will be produced.

```
        LD      V0, 0x101    ; Error during assembly
        LD      V1, 0xFF
        ADD     V1, 2        ; Due to overflow, V1 = 1
```

Math

The assembler is capable of resolving basic mathematical equations during assembly-time, so long as all operands are constants. The result is always truncated down to an integer. Operators have the same precedence as in C. Overflow / underflow will still raise an assembly-time error.

```
DEFINE  x 10

        LD      V0, x - 5                       ; Sets V0 to 5
        LD      V1, (x * 2) + (0x12 / 0b10)     ; Sets V1 to 28
        LD      V2, 3 ** 2                      ; Sets V2 to 9
        LD      V3, V1 + 2                      ; Error, V3 not a constant
        LD      V4, 0xFF + 2                    ; Error, 0x101 out of range
```

Labels

A label is a case-insensitive bookmark for a memory address. All labels must be unique, and may not shadow any keywords or register names. Its value is computed during assembly, and resolves to the location in memory the next instruction is located. Most Chip-8 emulators begin program data at memory location 0x200. For the following example, remember that each instruction is 2 bytes long.

```
label1:                 ; label1 = 0x200
label2:                 ; label2 = 0x200
        LD      V0, 5
label3:                 ; label3 = 0x202
        JP      label1  ; Jump to 0x200
```

The DEFINE Macro

The DEFINE macro is a label with an explicit value associated with it. All define macros should be placed at the top of the file, before any instructions are executed. While defines may be placed anywhere in the code, they will still affect any instructions that came before them. Notice that there is no comma between the macro's key and value.

```
DEFINE  x V0
DEFINE  value 5
DEFINE  JUMP jp

start:
        LD      x, value  ; Sets V0 to 5
        LD      y, value  ; Sets V1 to 5, even though y is defined later
        JUMP    start     ; Labels can even replace instructions


DEFINE  y V1

DEFINE  0xF 15            ; Error – Not in format /[A-Z_][0-9A-Z_]*/
DEFINE  I 6               ; Error – 'I' is a restricted keyword
```

DEFINE macros are even allowed to contain other symbols.

```
DEFINE  x 10
DEFINE  y x * x

        ADD V0, y  ; Evaluates to 100
```

Label Recursion Limit Exception

To prevent circular or never-ending DEFINE replacements, there is a hard limit (255) on how deeply nested the macro is. This limit is almost guaranteed to only be reached through unintentional circular references. Note that this limit does not apply to the number of simultaneous replacements, but rather the consecutive replacements that occur when the value of a DEFINE macro is the key of another.

```
; Standard circular reference will loop infinitely until the limit
; is reached and an exception is thrown
DEFINE  x y
DEFINE  y x

; While this will result in hundreds of replacements, it only has
; a recursion level of 5 and therefore will not throw an exception
DEFINE a1 1
DEFINE a2 a1 + a1 + a1 + a1 + a1
DEFINE a3 a2 + a2 + a2 + a2 + a2
DEFINE a4 a3 + a3 + a3 + a3 + a3
DEFINE a5 a4 + a4 + a4 + a4 + a4
DEFINE a6 a5 + a5 + a5 + a5 + a5
```

Comments

Comments are immediately discarded during the preprocessing phase of the assembler, and have no effect on the program. However, they are integral to making Chip-8 code readable. For this assembler, comments begin with a semicolon (;) and may be placed anywhere on a line. Other assemblers may use double slash (//) or a pound sign (#) to denote a comment. Every source file should start with a comment block following this format:

```
; FILE: Adder.asm
; AUTHOR: Michael Bianconi
; CREATED: 12 December 2020
; DESCRIPTION: Performs simple addition based on user input
```

When commenting subroutines, it may be prudent to place a comment after every instruction. These comments should be structured such that reading only the comments will provide a good overview of what is happening. The following snippet is the main loop of a program that displays all 16 fonts.

```
loop:
        SNE     spriteIndex, numSprites  ; If no more sprites,
        JP      end                      ; Jump to end
        DRW     x, y, spriteHeight       ; Else, draw the current sprite
        ADD     spriteIndex, 1           ; Set spriteIndex to next sprite
        LD      F, spriteIndex           ; Move I to next sprite
        ADD     x, spriteWidth           ; Go to next column
        SE      x, displayWidth          ; If not end of line,
        JP      loop                     ; Go draw the next sprite
        LD      x, 0                     ; Else, set x back to 0
        ADD     y, spriteHeight          ; Move y down to the next line
        ADD     y, 1                     ; Add 1 pixel of padding
        JP      loop                     ; Go draw the next sprite
```

Indentation, Spacing, and Newlines

Each label and define should be placed *un-indented* on its own line. Instructions should be indented 8 spaces. The first operand should be indented another 8 spaces. Operands will be spaced one space apart. Comments should be aligned with each other *within their subroutine*.

```
DEFINE  current V0  ; Holds the current sprite index
DEFINE  x       V1  ; Where to display sprite (x-axis)
DEFINE  y       V2  ; Where to display sprite (y-axis)
DEFINE  _       VE  ; Unused register

start:
        LD      F, current    ; set I to the location of the current sprite
        CLS                   ; clear previous sprite from screen
        DRW     x, y, 5       ; draw new sprite
        ADD     current, 1    ; advance to next sprite
        SNE     current, 0xF  ; if looped through all sprites
        JP      end           ; jump to end
        LD      _, K          ; else wait for key press
        JP      start         ; key was pressed, go to start


end:
        SYS     0             ; Done
```

Warnings

Instructions are 2 bytes long, and always start on an even-numbered memory address. The process of reading the next instruction is:

```
OP = (MEM[PC] << 8) | MEM[PC+1]
```

This has a very important implication: the program counter should always be an even number. However, there is no restriction from jumping the program counter to an odd-numbered address. When this happens, the lower 8 bits of the current instruction, and the upper 8 bits of the next instruction, will bitwise-OR'd together to create an entirely new instruction.

Additional References

The Chip-8 Wikipedia page

Cowgod's Chip-8 Technical Reference