Instructions for using the assembler &

Possible commands for our basic assembly language

Quick start guide:

* In the same folder as this document, you’ll find a .jar file and an intellij project folder.
* To run assembler, navigate to your java environment and run something like the following command: java -jar Chip8 Assembler.jar “fileDirectoryForAssemblyLangFile.txt”
* The assembler will place your opcode file in the same folder as the .jar file. It will be named programFile.txt
* This assembler will follow a similar instruction set as this link:
  + <http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#memmap>

What it does:

* The assembler transforms assembly language instructions into chip8 supported opcodes.
* Ex: SE V3 VA 🡪 53A0
* Because we have created our own assembler, the assembly language used has been slightly modified from the reference noted above.

How it works:

* The assembler reads in 1 line at a time, so only place 1 instruction per line.
* It then splits the line into tokens, using the space “ “ as a separator.
* When writing instructions, only place 1 per line and separate each part of the instruction by a space, nothing else (no commas).
* Please see the exampleAssembly.txt file for an example of each instruction being used.

For the list below, please refer to these variable definitions:

* nnn or addr - A 12-bit value, the lowest 12 bits of the instruction (3 digits)
* n or nibble - A 4-bit value, the lowest 4 bits of the instruction (1 digit)
* x - A 4-bit value, the lower 4 bits of the high byte of the instruction (1 digit)
* y - A 4-bit value, the upper 4 bits of the low byte of the instruction (1 digit)
* kk or byte - An 8-bit value, the lowest 8 bits of the instruction (2 digits)
* Vx refers to register x.

Supported instructions:

**0nnn - SYS addr**  
Jump to a machine code routine at nnn.  
This instruction is only used on the old computers on which Chip-8 was originally implemented. It is ignored by modern interpreters.

**00E0 - CLS**  
Clear the display.

**00EE - RET**  
Return from a subroutine.  
The interpreter sets the program counter to the address at the top of the stack, then subtracts 1 from the stack pointer.  
  
**1nnn - JP addr**  
Jump to location nnn.  
The interpreter sets the program counter to nnn.

**2nnn - CALL addr**  
Call subroutine at nnn.  
The interpreter increments the stack pointer, then puts the current PC on the top of the stack. The PC is then set to nnn.  
  
**3xkk - SE Vx byte**  
Skip next instruction if Vx = kk.  
The interpreter compares register Vx to kk, and if they are equal, increments the program counter by 2.  
  
**4xkk - SNE Vx byte**  
Skip next instruction if Vx != kk.  
The interpreter compares register Vx to kk, and if they are not equal, increments the program counter by 2.  
  
**5xy0 - SE Vx Vy**  
Skip next instruction if Vx = Vy.  
The interpreter compares register Vx to register Vy, and if they are equal, increments the program counter by 2.  
  
**6xkk - LD Vx byte**  
Set Vx = kk.  
The interpreter puts the value kk into register Vx.  
  
**7xkk - ADD Vx byte**  
Set Vx = Vx + kk.  
Adds the value kk to the value of register Vx, then stores the result in Vx.   
  
**8xy0 - LD Vx Vy**  
Set Vx = Vy.  
Stores the value of register Vy in register Vx.  
  
**8xy1 - OR Vx Vy**  
Set Vx = Vx OR Vy.  
Performs a bitwise OR on the values of Vx and Vy, then stores the result in Vx. A bitwise OR compares the corrseponding bits from two values, and if either bit is 1, then the same bit in the result is also 1. Otherwise, it is 0. 

**8xy2 - AND Vx Vy**  
Set Vx = Vx AND Vy.  
Performs a bitwise AND on the values of Vx and Vy, then stores the result in Vx. A bitwise AND compares the corrseponding bits from two values, and if both bits are 1, then the same bit in the result is also 1. Otherwise, it is 0.   
  
**8xy3 - XOR Vx Vy**  
Set Vx = Vx XOR Vy.  
Performs a bitwise exclusive OR on the values of Vx and Vy, then stores the result in Vx. An exclusive OR compares the corrseponding bits from two values, and if the bits are not both the same, then the corresponding bit in the result is set to 1. Otherwise, it is 0.   
  
**8xy4 - ADD Vx Vy**  
Set Vx = Vx + Vy, set VF = carry.  
The values of Vx and Vy are added together. If the result is greater than 8 bits (i.e., > 255,) VF is set to 1, otherwise 0. Only the lowest 8 bits of the result are kept, and stored in Vx.  
  
**8xy5 - SUB Vx Vy**  
Set Vx = Vx - Vy, set VF = NOT borrow.  
If Vx > Vy, then VF is set to 1, otherwise 0. Then Vy is subtracted from Vx, and the results stored in Vx.  
  
**8xy6 - SHR Vx**  
Set Vx = Vx SHR 1.  
If the least-significant bit of Vx is 1, then VF is set to 1, otherwise 0. Then Vx is divided by 2.  
**8xy7 – SUBN Vx Vy**  
Set Vx = Vy - Vx, set VF = NOT borrow.  
If Vy > Vx, then VF is set to 1, otherwise 0. Then Vx is subtracted from Vy, and the results stored in Vx.  
  
**8xyE - SHL Vx**  
Set Vx = Vx SHL 1.  
If the most-significant bit of Vx is 1, then VF is set to 1, otherwise to 0. Then Vx is multiplied by 2.  
  
**9xy0 - SNE Vx Vy**  
Skip next instruction if Vx != Vy.  
The values of Vx and Vy are compared, and if they are not equal, the program counter is increased by 2.  
  
**Annn - LD addr**  
Set I = nnn.  
The value of register I is set to nnn.  
  
**Bnnn - JP V0 addr**  
Jump to location nnn + V0.  
The program counter is set to nnn plus the value of V0.



**Cxkk - RAND Vx byte**  
Set Vx = random byte AND kk.  
The interpreter generates a random number from 0 to 255, which is then ANDed with the value kk. The results are stored in Vx. See instruction [8xy2](http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#8xy2) for more information on AND.

**Dxyn - DRAW Vx Vy nibble**  
Display n-byte sprite starting at memory location I at (Vx, Vy), set VF = collision.  
The interpreter reads n bytes from memory, starting at the address stored in I. These bytes are then displayed as sprites on screen at coordinates (Vx, Vy). Sprites are XORed onto the existing screen. If this causes any pixels to be erased, VF is set to 1, otherwise it is set to 0. If the sprite is positioned so part of it is outside the coordinates of the display, it wraps around to the opposite side of the screen. See instruction [8xy3](http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#8xy3) for more information on XOR, and section 2.4, [Display](http://devernay.free.fr/hacks/chip8/2.4), for more information on the Chip-8 screen and sprites.  
  
**Ex9E - SKP Vx**  
Skip next instruction if key with the value of Vx is pressed.  
Checks the keyboard, and if the key corresponding to the value of Vx is currently in the down position, PC is increased by 2.  
  
**ExA1 - SKNP Vx**  
Skip next instruction if key with the value of Vx is not pressed.  
Checks the keyboard, and if the key corresponding to the value of Vx is currently in the up position, PC is increased by 2.

**Fx07 - LD Vx DT**  
Set Vx = delay timer value.  
The value of DT is placed into Vx.  
  
**Fx0A - LD Vx K**  
Wait for a key press, store the value of the key in Vx.  
All execution stops until a key is pressed, then the value of that key is stored in Vx.  
  
**Fx15 - LD DT Vx**  
Set delay timer = Vx.  
DT is set equal to the value of Vx.  
  
**Fx18 - LD ST Vx**  
Set sound timer = Vx.  
ST is set equal to the value of Vx.  
  
**Fx1E - ADD I Vx**  
Set I = I + Vx.  
The values of I and Vx are added, and the results are stored in I.  
  
**Fx29 - LD F Vx**  
Set I = location of sprite for digit Vx.  
The value of I is set to the location for the hexadecimal sprite corresponding to the value of Vx. See section 2.4, [Display](http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#2.4), for more information on the Chip-8 hexadecimal font.  
  
**Fx33 - LD B Vx**  
Store BCD representation of Vx in memory locations I, I+1, and I+2.  
The interpreter takes the decimal value of Vx, and places the hundreds digit in memory at location in I, the tens digit at location I+1, and the ones digit at location I+2.  
  
**Fx55 - DUMP Vx**  
Store registers V0 through Vx in memory starting at location I.  
The interpreter copies the values of registers V0 through Vx into memory, starting at the address in I.

**Fx65 - READ Vx**  
Read registers V0 through Vx from memory starting at location I.  
The interpreter reads values from memory starting at location I into registers V0 through Vx.