MASTERING CHIP-8

By Matthew Mikolay

The following article is a document I've been working on for quite some time. Unfortunately, finishing it always seems to get pushed to the back-burner when other commitments arise. I'm hoping that by posting it here and adding to it over time, I'll be able to finally complete it. If you have any suggestions or comments, feel free to email me!

Hexadecimal values are written in `monospaced` font. Assume all monospaced numbers are hexadecimal unless otherwise stated.

Introduction

CHIP-8 is an interpreted minimalist programming language that was designed by Joseph Weisbecker in the 1970s for use on the RCA COSMAC VIP computer. Due to its hexadecimal format, it was best suited to machines with a scarcity of memory, as minimal text processing had to be performed by the interpreter before a program could be executed. This property inevitably led to its implementation on a variety of hobbyist computers aside from the VIP, such as the COSMAC ELF, Telmac 1800, and ETI 660.

The thirty-one instructions comprising the original CHIP-8 instruction set provide utilities for primitive audio and monochrome video output, user input, and data manipulation. CHIP-8 enjoyed relative success during the late 1970s and early 1980s as a popular language for the development of simple video games, and even spawned a multitude of dialects providing additional features.

Today, CHIP-8 remains popular in the hobbyist communities surrounding the computers on which it was once implemented. CHIP-8 interpreters, often mislabeled as "emulators," are increasingly common on a wide variety of platforms, allowing any individual to run a CHIP-8 program in the absence of an original hobbyist computer. This abundance of interpreters is due to the similarity in design between a CHIP-8 interpreter and a system emulator; those wishing to gain experience in emulator implementation often take on the task of programming a CHIP-8 interpreter.

Despite its simplicity, a variety of enjoyable games and programs have been coded in CHIP-8, proving that a programmer need not be limited by the scope of a language. Though undoubtedly minimal in the features it offers to programmers, CHIP-8 represents a successful attempt at scaling down the complexities of a programming language to the basic level needed to create simple programs and games without difficulty.

CHIP-8 Instructions

CHIP-8 programs are strictly hexadecimal based. This means that the format of a CHIP-8 program bears little resemblance to the text-based formats of higher level languages. Each CHIP-8 instruction is two bytes in length and is represented using four hexadecimal digits. For example, one common instruction is the `00E0` instruction, which is used to clear the screen of all graphics data.

Certain CHIP-8 instructions accept 'arguments' to specify the values which should be read or modified by a given instruction when encountered by the interpreter. An argument is passed to an instruction of this type also as a hexadecimal digit. When an instruction is presented in this document containing non-hexadecimal characters, these locations should be replaced in a program with valid hexadecimal digits depending upon the input data. For example, valid uses of the CHIP-8 instruction `8XY1` include `8001`, `81A1`, `8F21`, etc.

Storage in Memory

CHIP-8 instructions are stored directly in system memory. On many of the hobbyist computers of old, all CHIP-8 code would be entered directly into the system using toggle switches or a memory editing utility. Modern platforms allow files containing the binary data corresponding to the hexadecimal instructions of a CHIP-8 program to be loaded into an interpreter. These modern interpreters are simply automating the entry of CHIP-8 data into the machine. In either case, each CHIP-8 instruction in memory is said to be assigned to a unique memory address.

A small set of CHIP-8 instructions, including those used to generate graphics, require these memory addresses to be specified as arguments. For this reason, it is important to always be aware of the memory location of each instruction when developing a CHIP-8 program.

Consider the `1NNN`, instruction, which is used to jump to a certain address. A valid use of this instruction would be `134A`, which would reference the memory address `34A`.

As standardized by the COSMAC VIP, the programmer should assume that his or her CHIP-8 program is to be loaded into the machine starting at address `200`, even when using a modern interpreter. A few select hobbyist computers require that CHIP-8 programs be loaded starting at a different address, such as the ETI-660, but this should not be considered the norm.

It should be noted that CHIP-8 programs are normally stored in memory in big-endian fashion, with the most significant byte of a two-byte instruction being stored first. The CHIP-8 interpreter will execute a CHIP-8 program starting at the initial address by stepping through the instructions stored in memory one at a time and processing them in a linear manner, unless instructions modifying flow control are encountered.

Pseudo-Assemblers

Because certain CHIP-8 instructions require memory addresses to be passed as arguments, the modification of a pre-existing program often proves to be a hassle when these memory address arguments must be adjusted as the result of instruction shifts. Fortunately, CHIP-8 pseudo-assemblers easily resolve this problem.

A CHIP-8 pseudo-assembler[†] takes in a series of labels, mnemonics, and arguments and outputs the corresponding hexadecimal CHIP-8 code. Each hexadecimal CHIP-8 instruction corresponds to a unique pseudo-assembly mnemonic. When using a pseudo-assembler, locations in memory can be assigned labels, allowing the CHIP-8 mnemonics which would normally accept memory addresses as arguments to reference these labels instead. This substitution of labels for hardcoded addresses allows the programmer to disregard the manual management of addresses, as all addresses to which the labels correspond will be recomputed at assembly time. Therefore, instructions can be inserted and removed in a CHIP-8 program without having to modify other sections of code.

Coding in CHIP-8 pseudo-assembly is strongly encouraged, as it greatly simplifies the process of managing those instructions referencing memory addresses. However, pseudo-assembly was not a part of the original CHIP-8 implementation on the COSMAC VIP, and for this reason, all CHIP-8 code in this document will be presented in standard hexadecimal format.

[†]An assembler accepts a series of mnemonics and outputs the corresponding machine code. Likewise, a pseudo-assembler accepts a series of mnemonics and outputs a program in a language other than machine code. For this reason, the conversion program is identified as a pseudo-assembler.

A Note on Undocumented Instructions

Many documents published shortly after the creation of CHIP-8 fail to describe a select few valid CHIP-8 instructions: the 8XY3, 8XY6, 8XY7, and 8XYE instructions all went undocumented. This is due to the method with which the CHIP-8 interpreter was implemented on the COSMAC VIP.

The VIP interpreter was designed to accept a generic 8XYN instruction, where N is any valid hexadecimal digit, and execute machine language code dependent upon this value N. The functionality of instructions 8XY1, 8XY2, 8XY4, and 8XY5 was documented, but the four aforementioned instructions remained neglected. However, VIP programmers soon learned of these undocumented instructions and used them accordingly. For this reason, they are considered a part of the original CHIP-8 instruction set and are described in the following document.

All About Data Registers

The data register is the primary utility for data manipulation provided by the CHIP-8 language. In order to perform any sort of arithmetic operation, a register must be used. This concept parallels that of registers in a central processing unit or microprocessor. In fact, on the COSMAC VIP, the CHIP-8 registers were linked directly to the registers of the 1802 microprocessor.

CHIP-8 allows for the usage of sixteen eight-bit general purpose registers capable of storing unsigned integers between decimal 0 and 255, or hexadecimal 00 to FF. These registers are referred to as V0 to VF, one for each hexadecimal digit. Any register can be used for data manipulation, but it should be noted that the VF register is often modified by certain instructions to act as a flag.

The most primitive operation involving a data register is to set the value of any given register. Two instructions exist to perform this function: one to set a register to a specific eight bit value, and another to set a register's value to that of another given register:

Opcode	Description
6XNN	Store number NN in register VX
8XY0	Store the value of register VY in register VX

Next, an eight bit value can be added to any given register using the following command:

Opcode	Description
7XNN	Add the value NN to register VX

Be aware that once the supplied number is added, if the value of the register exceeds decimal 255 (the highest possible value that can be stored by an eight bit register), the register will wraparound to a corresponding value that can be stored by an eight bit register. In other words, the register will always be reduced modulo decimal 256.

Likewise, two registers can be added together. To add one register to another, the following instruction can be used:

Opcode	Description
8XY4	Add the value of register VY to register VX Set VF to 01 if a carry occurs Set VF to 00 if a carry does not occur

Similar to the 7XNN instruction, the 8XY4 instruction will cause values too large to be stored in a register to be wrapped around to a modulo equivelent. However, unlike the 7XNN instruction, the 8XY4 instruction will modify the VF register to signal when a carry has taken place. A carry is a term used to

describe the aforementioned action of when a value is too large to be stored in a given register. When a carry takes place, the interpreter will set register VF to `01`. Otherwise, VF will be set to `00`. Therefore, the `8XY4` instruction will always modify the VF register.

Just as values can be added to a register, CHIP-8 provides instructions to subtract one register from another:

Opcode	Description
`8XY5`	Subtract the value of register VY from register VX Set VF to 00 if a borrow occurs Set VF to 01 if a borrow does not occur
`8XY7`	Set register VX to the value of VY minus VX Set VF to 00 if a borrow occurs Set VF to 01 if a borrow does not occur

Observe that the only difference between these two subtraction instructions is which register is subtracted from which: which register is the minuend, and which is the subtrahend.

Similar to the `8XY4` instruction's usage of the VF register as a carry flag, these two subtraction instructions use the VF register to signal when a borrow occurs. Because data registers can only store unsigned values, a borrow will occur when the interpreter is instructed to subtract a value from a given register which would normally force the register to store a negative number. In other words, a borrow occurs whenever the subtrahend is greater than the minuend. The VF register is set to `00` if a borrow occurs, and `01` otherwise. *Therefore, the subtraction instructions will always modify the VF register.*

Aside from arithmetic operations, CHIP-8 allows for the manipulation of registers on the level of individual bits using bitwise operations. The following instructions can be used to AND, OR, and XOR two registers together.

Opcode	Description
`8XY2`	Set VX to VX AND VY
`8XY1`	Set VX to VX OR VY
`8XY3`	Set VX to VX XOR VY

The bits of a data register can also be shifted to the left or to the right using the following instructions:

Opcode	Description
`8XY6`	Store the value of register VY shifted right one bit in register VX Set register VF to the least significant bit prior to the shift
`8XYE`	Store the value of register VY shifted left one bit in register VX Set register VF to the most significant bit prior to the shift

Take note that the register shift instructions modify the VF register to store a value of `00` or `01` depending upon the bit that was shifted out of place. If a register is shifted left, the most significant bit of the register prior to the shift will be placed into VF. If a register is shifted right, the least significant bit of the register prior to the shift will be placed into VF.

Notice that these instructions shift register VY and store the result in register VX. It is a common misconception when programming in CHIP-8 to assume that the VX register is shifted by this instruction, and VY remains unmodified. To have a register perform a shift upon itself, a register VX can be passed as both arguments: `8XX6` or `8XXE`. For example, to shift register V6 right by one bit, the instruction `8666` can be used.

Often it is useful for a program or game to generate a random number for some sort of mathematical application. For this reason, CHIP-8 provides an instruction to set a register to a "random"[†] value.

Opcode	Description
CXNN	Set VX to a random number with a mask of NN

Observe that an additional byte is specified as a byte mask to reduce the size of the size of the set of random numbers capable of being returned by this instruction. When the CHIP-8 interpreter is commanded to generate a random number, it chooses a random value between 00 and FF. It then logical ANDs this value with the byte mask before placing it in the target register.

†The word "random" is placed in quotes to emphasize that a computer can never truly generate a random number. Instead, the values return by random number generation routines will always be pseudo-random and dependant upon some external mathematical operation.

Flow Control with Jumps

Often it is necessary to instruct the interpreter to execute code in a different section of a program, or repeat certain sections of code one or more times. Fortunately, the CHIP-8 language provides utilities to manipulate flow control.

The simplest instructions used to modify flow control are "jumps," which commands the interpreter to continue the execution of a program from another memory address. Two jump commands exist in the CHIP-8 language: one to simply jump to a given address, and another to jump to an address with an offset specified by the value stored in register V0.

Opcode	Description
1NNN	Jump to address NNN
BNNN	Jump to address NNN + V0

When using the jump instructions, it is necessary to make sure that the address to which a jump is made actually contains a valid CHIP-8 instruction. If an attempt is made to jump to an address outside of the memory containing the CHIP-8 program (for example, to the memory containing the CHIP-8 interpreter), the interpreter will most likely crash along with the program.

Care should also be taken to avoid unintentional infinite loops.

Subroutines

CHIP-8 also allows for the declaration of subroutines, which can then be called from other parts of the executing program. Subroutines are particularly useful when identical or similar code must be executed multiple times in a program. It should be noted that on the COSMAC VIP, enough stack space for twelve successive subroutine calls was allocated, but on many modern implementations, more memory is allocated for this purpose.

CHIP-8 subroutines do not require a specific instruction to signal their start. Instead, the memory address of the first instruction in the subroutine is sent to the call instruction.

Opcode	Description
2NNN	Execute subroutine starting at address NNN

CHIP-8 program execution will then continue from this address until a termination instruction is found. This termination statement informs the interpreter that the end of the currently executing subroutine has been reached, and program execution should proceed at the point from which the last subroutine call occurred.

Opcode	Description
00EE	Return from a subroutine

As previously stated, CHIP-8 was originally implemented on the RCA COSMAC VIP, and it was deemed desirable to include an option to call machine language subroutines from a CHIP-8 program. The following instruction informs the CHIP-8 interpreter to execute a machine language program at a given address, but it should be noted that this instruction is highly considered deprecated, as it often remains unimplemented on modern interpreters. It is included here to provide for a complete CHIP-8 instruction set.

Opcode	Description
0NNN	Execute machine language subroutine at address NNN

On the COSMAC VIP, in order to transfer control back to the CHIP-8 program from the machine language subroutine, it was necessary for the machine language subroutine to end with the byte D4. This would signal the completion of the machine language subroutine.

Conditional Branching using Skips

One of the most powerful aspects of the modern programming language is the ability to execute different sections of code depending upon if a certain condition is met. This is known as conditional branching, and is accomplished in CHIP-8 by combining two types of instructions: skips and jumps.

Before a conditional branch can be coded, a good understanding of instruction skips is needed. CHIP-8 provides a set of instructions that force the interpreter to check a given condition when executed. If this condition is true, then the instruction following immediately after the skip instruction will *not* be executed. Instead, the interpreter will resume execution beginning with the instruction immediately after the one which has just been ignored. Thus, a single instruction has been skipped.

The majority of skip commands operate based on the values of given registers, but a select few are driven by user input. These input-based skips will be described in forthcoming sections.

The following table lists all register-based skip commands.

Opcode	Description
3XNN	Skip the following instruction if the value of register VX equals NN
5XY0	Skip the following instruction if the value of register VX is equal to the value of register VY
4XNN	Skip the following instruction if the value of register VX is not equal to NN
9XY0	Skip the following instruction if the value of register VX is not equal to the value of register VY

In a sense, these skip commands provide a very primitive form of conditional branching: they determine whether a single instruction is executed or not depending upon the value of one or more data registers. Unfortunately, the primary complication with this model is that only a single instruction can be skipped. When the programmer wishes to skip sections of code composed of more than one CHIP-8 instruction, the skip instruction should be followed by one or more jump instructions. These jump instructions will instruct the interpreter where to jump to depending upon what condition is true.

Timers

Often the programmer might require the ability to determine when exactly certain events in a program take place. In this case, some sort of timer would prove useful, as the programmer should be able to read the value of the timer and execute code depending upon this value. Luckily, the CHIP-8 language supplies a single delay timer which can be used for such a purpose.

The delay timer acts similarly to a data register: the programmer can load an eight bit value into the delay timer, and read a value back later. The primary difference is that the delay timer perpetually counts down at a rate of sixty hertz until it reaches zero.

Only two instructions exist in the CHIP-8 language to manipulate the delay timer. The first is used to set the delay timer to a given eight bit value. When this value is non-zero, the delay timer will begin counting down until it reaches zero. The second is used to read the current value of the delay timer into a given register. This register can then be used to conditionally branch in the manner introduced in the previous section, allowing certain actions to occur based on the delay timer's value.

Opcode	Description
`FX15`	Set the delay timer to the value of register VX
`FX07`	Store the current value of the delay timer in register VX

CHIP-8 contains another timer, called the sound timer, which is used for sound output. Similar to the delay timer, the sound timer also counts down at a rate of sixty hertz. Contrastingly, when the sound timer is non-zero, a sound frequency will be generated by the speaker (or emulated by a modern interpreter).

Opcode	Description
`FX18`	Set the sound timer to the value of register VX

It should be noted that in the COSMAC VIP manual, it was made clear that the minimum value that the timer will respond to is `02`. Thus, setting the timer to a value of `01` would have no audible effect.

Keypad Input

The CHIP-8 programming language is able to detect input from a sixteen key keypad, with each key corresponding to a single unique hexadecimal digit. Computers like the COSMAC VIP and the ETI-660 actually provided the user with a hex keypad for input, but modern machines often lack such devices. For this reason, CHIP-8 "emulators" often map the key presses of a standard keyboard in order to simulate the key presses of a hex keypad.

CHIP-8 includes three instructions to detect input from the keypad. The simplest instruction halts all program execution until a key on the keypad is pressed, at which point a value corresponding to the key is stored in a given register.

Opcode	Description
`FX0A`	Wait for a keypress and store the result in register VX

The final two instructions do *not* halt the interpreter and wait for a keypress. Instead, they skip the succeeding instruction depending upon the state of the key corresponding to a value in a given register.

Opcode	Description
`EX9E`	Skip the following instruction if the key corresponding to the hex value currently stored in register VX is pressed
`EXA1`	Skip the following instruction if the key corresponding to the hex value currently stored in register VX is not pressed

As previously mentioned, these input-based skips can be combined with jumps to achieve a form of input-based conditional branching.

Graphics

CHIP-8 allows for the generation of monochrome graphics on a black and white† sixty-four by thirty-two pixel screen. By default, the screen is set to all black pixels. The only method of drawing to the screen is using sprites. CHIP-8 sprites are *always* eight pixels wide and between one to fifteen pixels high.

Sprite data is stored in memory, just like CHIP-8 instructions themselves. One byte corresponds to one row of a given sprite. How many rows (bytes) encompass a sprite is specified through the CHIP-8 instructions used to draw the sprites on the screen, and will be covered later. For sprite data, a bit set to one corresponds to a white pixel. Contrastingly, a bit set to zero corresponds to a transparent pixel.

As sprite data is stored in memory just like the actual CHIP-8 program instructions, care should be taken to prevent the interpreter from attempting to execute the sprite data as instructions. For this reason, it is advised to place all sprite data in a section of memory that sits independently of the main program memory. For example, the sprite data could be placed toward the beginning of the program, and preceded by a jump instruction, forcing the interpreter to skip over this data. In another case, sprite data could be placed at the end of the program, and the program could be coded in a way such that the sprite data would never be reached by the interpreter.

†The color of the screen is platform-dependent, but on the VIP, the screen was black and white.

The I Register

In order to specify the memory addresses containing the data for a given sprite, there must be some way to store an address for later use. The sixteen data registers (V0 - VF) provided by CHIP-8 are only eight bits in length, and therefore could only store addresses `00` to `FF`. Therefore, CHIP-8 provides a special register that is used only to store memory addresses. This register, called the I register, proves useful when performing operations involving reading and writing to and from memory, most importantly, drawing graphics.

The two basic CHIP-8 commands involving the I register allow the I register to be set to a given value, or add a value to the I register.

Opcode	Description
`ANNN`	Store memory address NNN in register I
`FX1E`	Add the value stored in register VX to register I

It should be noted that the I register acts very differently from the sixteen data registers of the CHIP-8 language. The data registers are used to manipulate and store data for use in a program, while the I register is used to store a single memory address which *cannot* be modified using any type of arithmetic instruction. *No instructions exist to modify the I register after it is set to a given value.*

Drawing Sprites to the Screen

A single instruction is needed to draw a sprite to the screen.

Opcode	Description
`DXYN`	Draw a sprite at position VX, VY with N bytes of sprite data starting at the address stored in I Set VF to 01 if any set pixels are changed to unset, and 00 otherwise

The two registers passed to this instruction determine the x and y location of the sprite on the screen. If the sprite is to be visible on the screen, the VX register must contain a value between `00` and `3F`, and the VY register must contain a value between `00` and `1F`.

When this instruction is processed by the interpreter, `N` bytes of data are read from memory starting from the address stored in register I. These bytes then represent the sprite data that will be used to draw the sprite on the screen. Therefore, the value of the I register determines which sprite is drawn, and should always point to the memory address where the sprite data for the desired graphic is stored. The corresponding graphic on the screen will be eight pixels wide and `N` pixels high.

CHIP-8 also includes an instruction that can be used to clear the screen of all sprite data.

Opcode	Description
`00E0`	Clear the screen

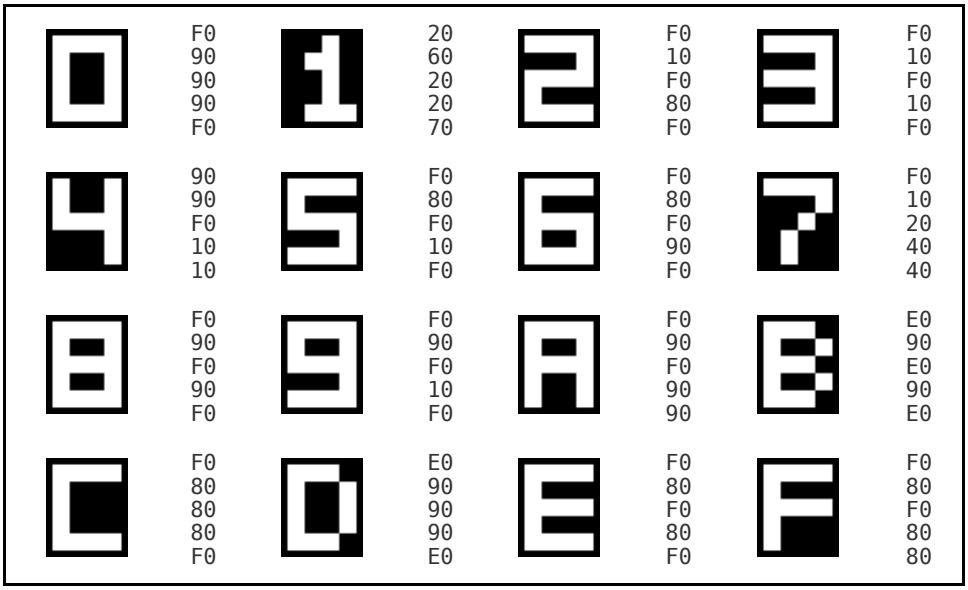
XOR Mode

All sprites are drawn to the screen using an exclusive-or (XOR) mode; when a request to draw a sprite is processed, the given sprite's data is XOR'd with the current graphics data of the screen. This mode of operation allows for the absence of an instruction to erase a sprite at a given position. Instead, when the programmer wishes to erase a sprite, they simply have to redraw that same sprite on the screen at the same location as before.

The `DXYN` instruction modifies the VF data register to reflect any state toggles due to XOR mode behavior. If a pixel on the screen is set to `01`, and the sprite to be drawn contains a `01` for this same pixel, the screen pixel is turned off and VF is set to `01`. If the sprite is simply drawn on the screen without drawing over any pixels set to `01`, VF is set to `00`.

Drawing Fonts

Because many programs often need to output a number to the screen, CHIP-8 contains built-in font utilities to allow for simple output of characters using the `DXYN` instruction. All hexadecimal digits (`0` - `9`, `A` - `F`) have corresponding sprite data already stored in the memory of the interpreter. The following table displays these sprites and their corresponding sprite data.



To set I to the memory address of the sprite data corresponding to one of these characters, a data register containing a single hexadecimal digit must be passed to the following instruction.

Opcode	Description
FX29	Set I to the memory address of the sprite data corresponding to the hexadecimal digit stored in register VX

Binary-Coded Decimal

In order to make good use of a utility such as the built in font sprite data of the CHIP-8 interpreter, it is often desirable to store the decimal equivalent of a binary or hexadecimal number in memory as individual decimal digits. This is called the binary-coded decimal (BCD), and CHIP-8 contains an instruction to convert any value stored in a data register into its BCD equivalent.

Opcode	Description
FX33	Store the binary-coded decimal equivalent of the value stored in register VX at addresses I, I+1, and I+2

When this instruction is executed by the interpreter, the value stored in register VX is converted to its decimal equivalent. Because each register is only eight bits in length, there will be three decimal digits (including any leading zeros). The most significant decimal digit is then stored in at the address found in I, the next in I + 1, and the least significant digit in I + 2. These values might then be used along with the font utility to output a decimal number to the screen.

Register Values and Memory Storage

Because CHIP-8 only allows the use of sixteen data registers, it is often desirable to store the current values of the registers in memory for later usage. Register values can be written into memory using the following instruction.

Opcode	Description
FX55	Store the values of registers V0 to VX inclusive in memory starting at address I I is set to I + X + 1 after operation

Just as register values can be stored in memory, the values of memory addresses can be read back into the registers. To accomplish this, the following instruction can be used.

Opcode	Description
FX65	Fill registers V0 to VX inclusive with the values stored in memory starting at address I I is set to I + X + 1 after operation

Notice that after both of these operations, the value of the I register will be incremented by X + 1. This is due to the changing of addresses by the interpreter.

Conclusion

Despite its minimalist design, the CHIP-8 language provides the programmer with thirty-five powerful commands with which complexity can be built. Please check back in the future as more is added to this tutorial!

Instruction Reference Table

The following is a reference table containing all thirty-five of the original CHIP-8 instructions. NNN refers to a hexadecimal memory address, NN refers to a hexadecimal byte, N refers to a hexadecimal nibble, and X and Y refer to registers.

Opcode	Description
0NNN	Execute machine language subroutine at address NNN
00E0	Clear the screen
00EE	Return from a subroutine
1NNN	Jump to address NNN
2NNN	Execute subroutine starting at address NNN
3XNN	Skip the following instruction if the value of register VX equals NN
4XNN	Skip the following instruction if the value of register VX is not equal to NN
5XY0	Skip the following instruction if the value of register VX is equal to the value of register VY
6XNN	Store number NN in register VX
7XNN	Add the value NN to register VX
8XY0	Store the value of register VY in register VX
8XY1	Set VX to VX OR VY
8XY2	Set VX to VX AND VY
8XY3	Set VX to VX XOR VY
8XY4	Add the value of register VY to register VX Set VF to 01 if a carry occurs Set VF to 00 if a carry does not occur
8XY5	Subtract the value of register VY from register VX Set VF to 00 if a borrow occurs Set VF to 01 if a borrow does not occur
8XY6	Store the value of register VY shifted right one bit in register VX Set register VF to the least significant bit prior to the shift
8XY7	Set register VX to the value of VY minus VX Set VF to 00 if a borrow occurs Set VF to 01 if a borrow does not occur
8XYE	Store the value of register VY shifted left one bit in register VX Set register VF to the most significant bit prior to the shift
9XY0	Skip the following instruction if the value of register VX is not equal to the value of register VY
ANNN	Store memory address NNN in register I
BNNN	Jump to address NNN + V0

CXNN	Set VX to a random number with a mask of NN
DXYN	Draw a sprite at position VX, VY with N bytes of sprite data starting at the address stored in I Set VF to 01 if any set pixels are changed to unset, and 00 otherwise
EX9E	Skip the following instruction if the key corresponding to the hex value currently stored in register VX is pressed
EXA1	Skip the following instruction if the key corresponding to the hex value currently stored in register VX is not pressed
FX07	Store the current value of the delay timer in register VX
FX0A	Wait for a keypress and store the result in register VX
FX15	Set the delay timer to the value of register VX
FX18	Set the sound timer to the value of register VX
FX1E	Add the value stored in register VX to register I
FX29	Set I to the memory address of the sprite data corresponding to the hexadecimal digit stored in register VX
FX33	Store the binary-coded decimal equivalent of the value stored in register VX at addresses I, I+1, and I+2
FX55	Store the values of registers V0 to VX inclusive in memory starting at address I I is set to I + X + 1 after operation
FX65	Fill registers V0 to VX inclusive with the values stored in memory starting at address I I is set to I + X + 1 after operation