Columbia University

Embedded Systems 4840

Chip-8 Design Specification

Authors:
Ashley Kling (ask2203)
Levi Oliver (lpo2105)
Gabrielle Taylor (gat2118)
David Watkins (djw2146)

Supervisor:
Prof. Stephen Edwards

March 24, 2016

Introduction

Chip-8 is an interpreted programming language from the 1970s. It ran on the COSMAC VIP, and supported many programs such as Pac-Man, Pong, Space Invaders, and Tetris. We aim to create a processor using SystemVerilog and the FPGA on the SoCKit board that runs these programs. During the boot process of the processor chip8 ROM files will be transferred onto the main memory of the processor. The processor will also allow for save states and restoring of states. The processor will handle keyboard inputs and output graphics and sound.

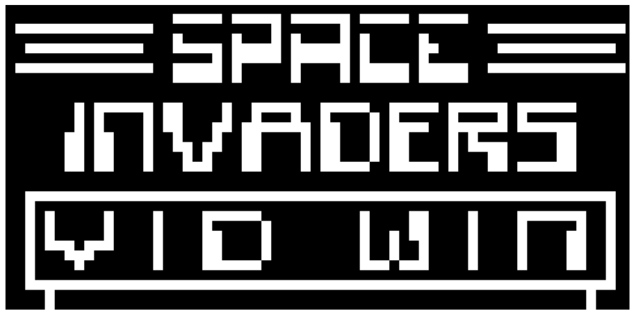

Figure 1: A popular Chip8 game: Space Invaders

Memory Overview

The Chip-8 specification requires the use of sixteen 8-bit registers (V0-VF), a 16-bit index register, a 64-byte stack with 8-bit stack pointer, an 8-bit delay timer, an 8-bit sound timer, a 64x32 bit frame buffer, and a 16-bit program counter. The Chip8 specification also supported 4096 bytes of addressable memory. All of the supported programs will start at memory location 0x200.

- The sound and delay timers sequentially decrease at a rate of 1 per tick of a 60Hz clock. When the sound timer is above 0, the sound will play as a single monotone beep.
- The framebuffer is an (x, y) addressable memory array that designates whether a pixel is currently on or off. This will be implemented with a write address, an (x, y) position, a offset in the x direction, and an 8-bit group of pixels to be drawn to the screen.
- The return address stack stores previous program counters when jumping into a new routine.
- The VF register is frequently used for storing carry values from a subtraction or addition action, and also specifies whether a particular pixel is to be drawn on the screen.

Graphics

Important to the specification is the 64x32 pixel display that is associated with the Chip8. Each pixel only contains the information as to whether it is on or off. All setting of pixels of this display are done through the use of sprites that are always $8 \times N$ where N is the pixel height of the sprite.
Chip8 comes with a font set (sprites) that allows character 0-9 and A-F to be printed directly to the screen. Each one of these characters fit within a 8x5 grid. [1]

[1] http://devernayfreefr/hacks/chip8/C8TECH10HTM

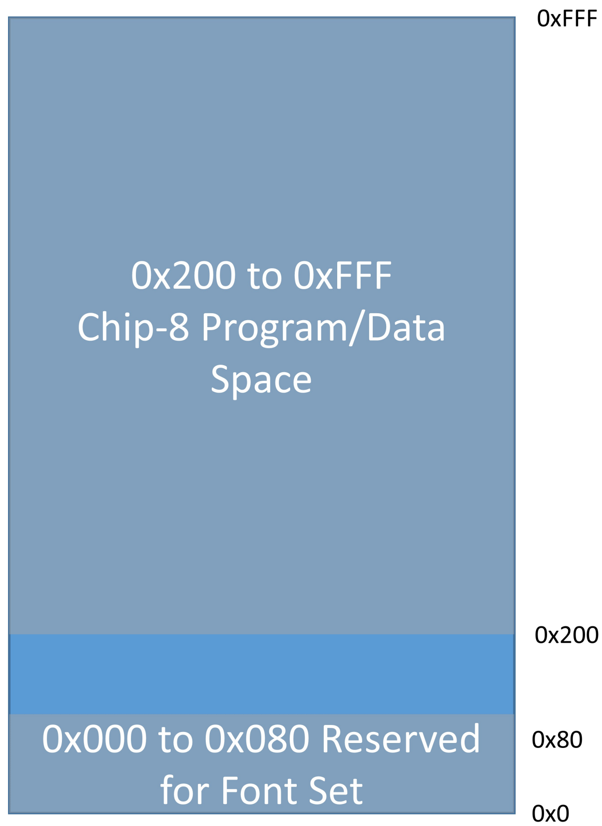

Figure 2: Chip8 4K memory layout

Op Codes

The Chip8 interpreter works by parsing 16 bit opcodes and operating on the data. All supported op codes in the original Chip8 specification are included.

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. This will not be implemented.

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.

"8"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

"9"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"0"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0

"1"	Binary	Hex
*	00100000	0x20
**	01100000	0x60
*	00100000	0x20
*	00100000	0x20
***	01110000	0x70

"A"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90

"B"	Binary	Hex
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0

"2"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

"3"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"C"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80
*	10000000	0x80
****	11110000	0xF0

"D"	Binary	Hex
***	11100000	0xE0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
***	11100000	0xE0

"4"	Binary	Hex
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
*	00010000	0x10

"5"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"E"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

"F"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80

"6"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

"7"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
*	00100000	0x20
*	01000000	0x40
*	01000000	0x40

Figure 3: Chip8 character sprite specification

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 corresponding 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 corresponding 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 corresponding 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 {, Vy}

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 {, Vy}

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 I, 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 - RND 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 for more information on AND.

Dxyn - DRW 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 XOR'd 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.

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. Delay Timer is set equal to the value of Vx.

Fx18 - LD ST, Vx

Set sound timer = Vx. Sound Timer 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, for more information on the Chip-8 hexadecimal font. To obtain this value, multiply VX by 5 (all font data stored in first 80 bytes of memory).

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 - LD [I], Vx

Stores V0 to VX in memory starting at address I. I is then set to I + x + 1.

Fx65 - LD Vx, [I]

Fills V0 to VX with values from memory starting at address I. I is then set to I + x + 1.

Keyboard Input

The keyboard input was a 16-key keyboard with keys $0-9, A-F$. There are a series of op codes (listed in the previous section) that use these key presses. In the design associated with this emulator, the keyboard input will be read in from the ARM processor running Linux and streamed to the emulator.

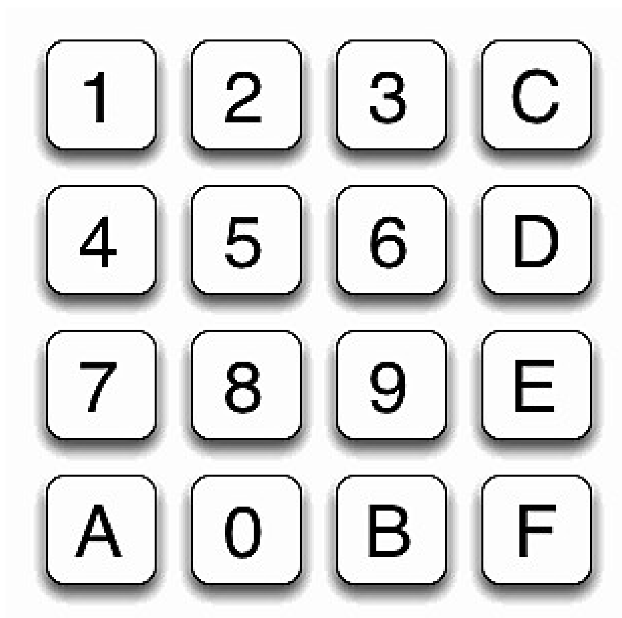

Figure 4: Chip8 16-key keyboard specification

Sound

Chip-8 provides 2 timers, a delay timer and a sound timer. The delay timer is active whenever the delay timer register (DT) is non-zero. This timer does nothing more than subtract 1 from the value of DT at a rate of 60Hz. When DT reaches 0, it deactivates.
The sound timer is active whenever the sound timer register (ST) is non-zero. This timer also decrements at a rate of 60Hz, however, as long as ST's value is greater than zero, the Chip-8 buzzer will sound. When ST reaches zero, the sound timer deactivates.
The output of the sound generator has one tone. In the following implementation it will have a soft tone so as to not aggravate the user. [2]

Design Overview

The design of the Chip8 system will have five major components. The 4K memory chip, the framebuffer, the central processing unit and controller, the 1-bit sound channel, and the VGA out. It will also communicate ROM files and state through the channel created by the ARM processor and the FPGA on the SoCKit board. It will also communicate the current key press value through this same linux-fpga channel.

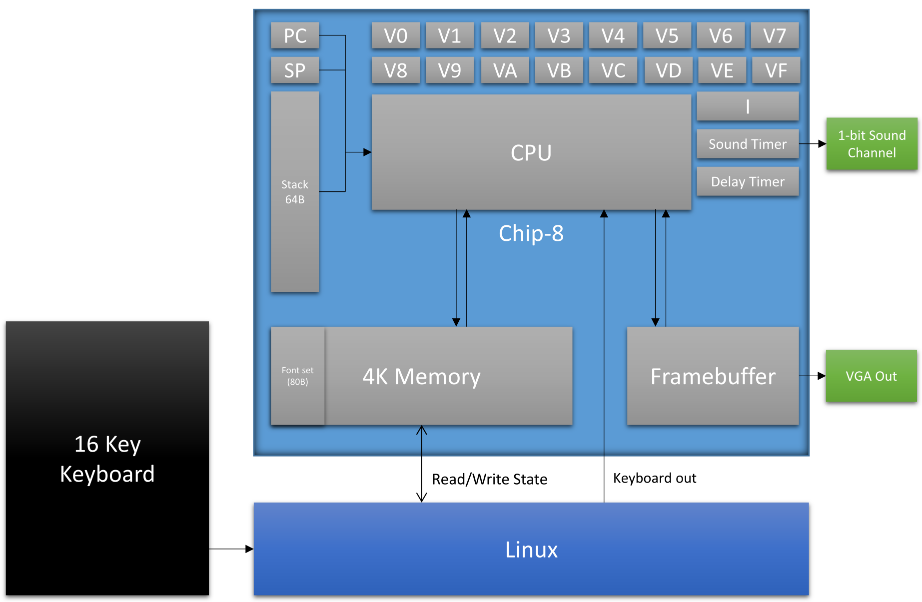

Figure 5: Chip8 emulator design overview

The following is the proposed circuit diagram layout of our chip8 emulator. The majority of the logic will be in the control unit which will specify when certain operations need to be activated. It also receives input directly from the ARM processor on the SoCKit board so that it knows when a particular key is being pressed and whether to load/save the current state of the processor. All of the components that store a value will be able to be read and stored in a file that can then be later loaded to restore a particular state to allow for saving games.

The program counter will be held at its current values many times during execution so that a particular instruction can be held out over the course of 1 CPU cycle. A CPU cycle will likely be much larger than a

[2] http://devernay.free.fr/hacks/chip8/C8TECH10.HTM

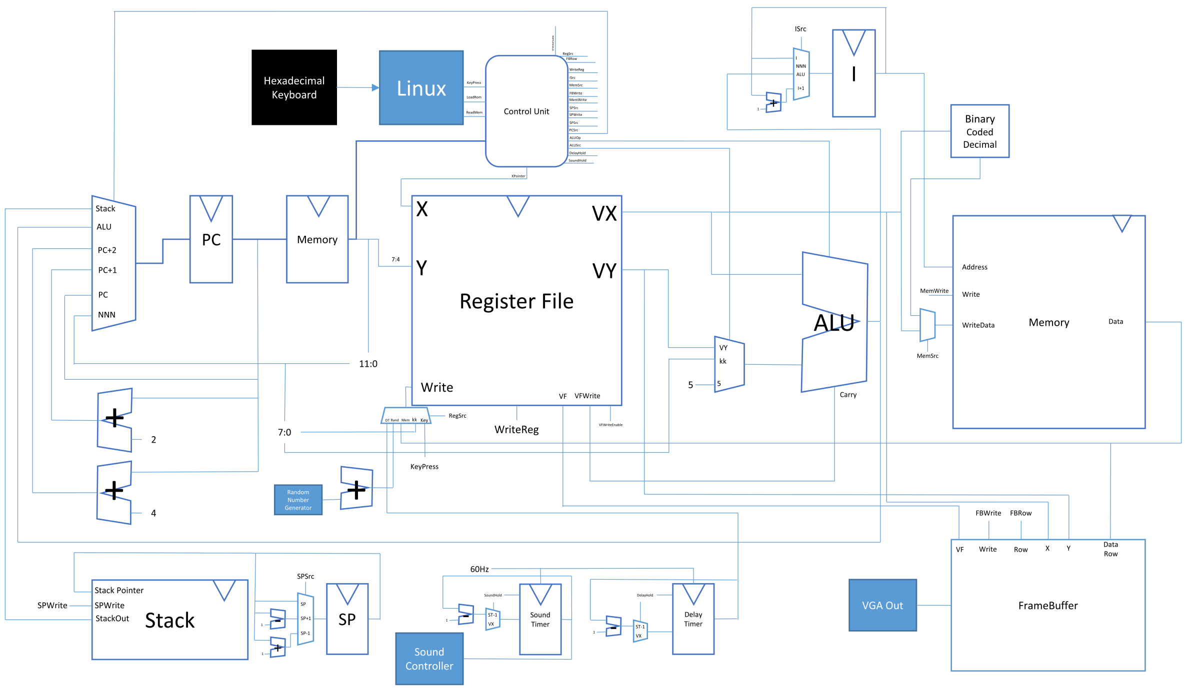

Figure 6: Chip8 emulator circuit design

clock cycle due to the enormous speed that an FPGA can run at compared to the 3.58MHz that the 1802 would run at on the COSMAC VIP, slowed down further by the interpreter's extra instructions. The final clock speed cannot yet be determined but will be evaluated as part of the second milestone (see milestones below).

The stack will be a 64 byte memory block that only reads out the current value stored in the stack pointer. The stack pointer will be either incremented or decremented by the current instruction being evaluated.

The framebuffer is a 64x32 bit memory array that is written two in 8-bit chunks by reading memory locations. The framebuffer will feature a wraparound that causes the pixels to be written from the position $Y + 0$ in the y axis, all the way until $(Y + N)\%32$. This will allow for proper wraparound of the sprites that need to be drawn.

The binary coded decimals module will take in the value stored in VX and then write out three values corresponding to the hundred's, ten's, and one's digits of the binary number. This will be written to memory over the course of 3 clock cycles as each of these values are stored in I, $I + 1$, and $I + 2$, respectively.

The keyboard presses will be synchronously written to the control unit from the ARM processor. This will allow the control unit to properly handle varying input without worrying about varying input signals affecting the resulting code.

There is also a random number generator in the specification for a Chip8 emulator. This random number generator will likely use a 16-bit gray code counter so that a 16-bit result can be obtained.

The following timing diagram is an example of a program that has four instructions, not including code surrounding the execution of these instruction. The timing of a particular operation is slightly off because it

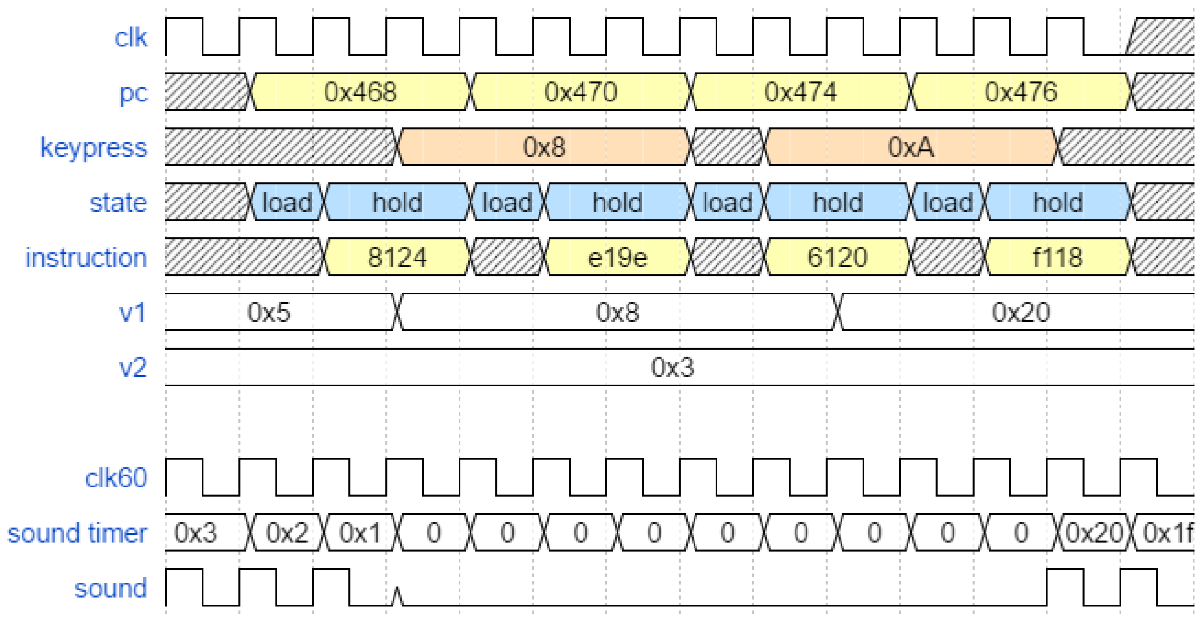

Figure 7: Chip8 example program timing diagram

is not taking into account the execution of the serial read and write to register instructions, or the draw sprite instruction, which will likely take roughly 16-20 clock cycles. When this window of operation is determined, the "CPU Cycle" will be determined to be x clock cycles. On each clock cycle, it is possible to read the instruction from memory, read from the register file, perform an ALU operation, and write to either the sound timer, delay timer, framebuffer, or the memory block.

In this example, the four instructions being executed are the add instruction (8xy4), the skip next instruction on keypress instruction (Ex9E), the load constant instruction (6xkk), and the set sound timer instruction (Fx18). The *state* indicates what state the processor is currently in. A load state is when the instruction is being read from memory and executing a read from the register file. The hold phase is either when the processor is doing nothing or continuing to execute a particular instruction. The *keypress* designates which of the 16 possible keys are currently pressed. During this example only the 8 and A keys are pressed. In a real execution of the emulator it is unlikely that the keys would actually get pressed that fast considering one clock cycle will likely be a lot smaller. The *sound timer* is also indicated as a decreasing value. Whenever the *sound timer* is greater than 1, the sound output is producing output. The clock for the *sound timer* is different from the clock for the processor, which means that the exact timing is not necessarily accurate for both sound and CPU.

During the execution of this program, first the instruction at location $0x468$ is loaded which is instruction 8124. This means that register $V1$ and register $V2$ are going to be added together and stored in register $V1$. This is why at the end of the execution of that instruction the value at $V1$ is $0x8$. Then the next instruction, $E19E$, checks to see if the key pressed during that instruction is equal to the value stored in register $V1$. Because they are equal, the next instruction is skipped, therefore the PC is updated to $0x474$. The next instruction 6120 denotes that $V1$ will be loaded with value $0x20$. The instruction after that, $F118$, sets the value in *sound timer* equal to $V1$, which then causes the audio to play again.

Timeline

Milestone 1 - March 31st

We will have implemented a semi functional framebuffer module that will allow us to debug the output of the Chip8 emulator. The layout of the other modules will also be completed, such as the interfaces to using the stack, sound timer, delay timer, binary coded decimal, random number generator, program counter, and control unit.

Milestone 2 - April 12th

The processor will be able to execute all instructions pertaining to the ALU. We will also support keyboard presses and random number generation. For this to work, the register file must also be implemented as well as any components of the control unit that are required to control access to certain elements. We will also determine the length of a CPU cycle in terms of clock cycles.

Milestone 3 - April 26th

The cpu will be able to draw sprites to the screen. It will also be able to draw elements from the font set. The memory instructions will also work, allowing for multiple reads/writes from register files.

Deliverables - May 12th

The Chip8 emulator will be completed with saving and writing state as well as any bugs will be fixed by this time.