An Introduction to Chip-8 Emulation using the Rust Programming Language

by aquova

31 January 2021

Table of Contents

1 An Introduction to Video Game Emulation with Rust

Developing a video game emulator is becoming an increasingly popular hobby project for developers. It requires knowledge of low-level hardware, modern programming languages, and graphics systems to successfully create one. It is an incredibly rewarding and excellent learning project; not only does it have clear goals, but it also very rewarding to successfully play games on an emulator you've written yourself. I am still a relatively new emulation developer, but I wouldn't have been able to reach the place I am now if it weren't for excellent guides and tutorials online. To that end, I wanted to give back to the community by writing a guide with some of the tricks I picked up, in hopes it is useful for someone else.

1.1 Intro to Chip-8

Our target system is the Chip-8. The Chip-8 has become the "Hello World" for emulation development of a sort. While you might be tempted to begin with something more exciting like the NES or Game Boy, these are a level of complexity higher than the Chip-8 (although don't let that stop you if you're dead set on trying it!). The Chip-8 has a 1-bit monochrome display, a simple 1-channel single tone audio, and only 35 instructions (compared to ~500 for the Game Boy), but more on that later. This guide will cover the technical specifics of the Chip-8, what hardware systems need to be emulated and how, and how to interact with the user. This guide will also only focus on the original Chip-8 specification, and will not implement any of the many proposed extensions that have been created over the years, such as the Super Chip-8, Chip-16, or XO-Chip. Many of these were created independently of each other, and thus add contradictory features to one another. If you are curious, an extensive overview of many of these can be found here.

1.2 Chip-8 Technical Specifications

Some additional technical details about the Chip-8, which you will become very acquainted with.

- A 64x32 monochrome display, drawn to via sprites that are always 8px wide and between 1 and 16 pixels tall.
- Sixteen 8-bit general purpose registers, referred to as V0 thru VF. VF also doubles as the flag register for overflow operations.
- 16-bit program counter
- Single 16-bit register used as a pointer for memory access, called the *I Register*.
- An unstandardised amount of RAM, however most emulators allocate 4 KB.
- 16-bit stack used for calling and returning from subroutines.
- 16-key keyboard input.
- Two special registers which decrease each frame and trigger upon reaching zero:
  - Delay timer - Used for time-based game events.
  - Sound timer - Used to trigger the audio beep.

1.3 Intro to Rust

Emulators can be (and have been) written in just about any programming language. This guide will specifically use the Rust programming language, although the steps outlined here are applicable to any language, in a higher level sense. Rust offers a number of great advantages - it is a compiled language with targets for the major platforms and it has an active community of external libraries we can utilize for our project. There is also great support for WebAssembly, which will allow us to recompile our code to work in a browser with a minimal amount of tweaking. This guide will assume you understand the basics of the Rust language and programming as a whole. I will be explaining the code as we go along, so a basic understanding should be enough, but as Rust has a notoriously high learning curve, I would recommend reading and referencing the excellent official Rust book on any concepts that are unfamiliar to you as the guide progresses. This guide also assumes that you have Rust installed and working correctly. Please consult the installation instructions for your platform if need be.

1.4 What you will need

Apart from the Rust language, there are a few items you will need before you begin.

1.4.1 Text Editor

While you are free to use your favorite editor of choice, for those who are looking for an editor well suited for Rust, there are two I recommend which offer features like syntax highlighting, code suggestions, and debugger support.

- Visual Studio Code is the editor I personally use for Rust, in combination with the rust-analyzer extension.
- While JetBrains does not offer a dedicated Rust IDE, there is a Rust extension for many of its other products. In particular, the extension for CLion has additional functionality that the others do not, such as integrated debugger support. Keep in mind that CLion is a paid product, although it offers a 30 day trial as well as extended free periods for students.

If you do not care for any of these, Rust syntax and autocompletion plugins exist for many other editors, and it can be debugged fairly easily with many other debuggers, such as gdb.

1.4.2 Test ROMs

An emulator isn't much use if you have nothing to run! Included with the source code for this book are a number of commonly distributed Chip-8 programs, which can also be found here. Some of these games will be shown as an example throughout this guide, but there are many more games in existence to use as well.

1.4.3 Misc.

There are a few other items that may be helpful as we progress.

- If you are unfamiliar with hexadecimal, we will be using it throughout the project, so it might be worth reading a refresher if its not something you're comfortable with.
- Since Chip-8 games are in a hexadecimal format, it is often helpful to be able to view the raw hex values as you debug. Standard text editors typically don't have support for viewing files in hexadecimal, instead a specialized hex editor is required. Many offer similar features, but I personally like the Reverse Engineer's Hex Editor.

With that set, let's begin!

2 Emulation Basics

This first chapter will be an overview of the concepts of emulation development and the Chip-8. Subsequent chapters will have code implementations in Rust, but for those of you who have no interest in Rust or prefer to work without examples, this chapter will give an introduction to what steps a real system performs, what we need to emulate ourselves, and how all the moving pieces fit together. You may be familiar with some or all of these concepts already, so feel free to skim through. However, if you're completely new to this world, this should provide the information you'll need to wrap your head around what exactly you're about to implement.

2.1 What's in a Chip-8 Game?

Let's begin with a simple question. If I give you a Chip-8 ROM[1] file, what exactly is in it? Somehow it needs to contain all of the game logic and graphical assets needed to make a game run on a screen, but how is it all laid out?

This is a basic, fundamental question, and we're going to need to figure it out at some point as we're going to have to read in this file and make it run. Well, there's only one way to find out so let's try and open a Chip-8 game in a text editor (such as Notepad, TextEdit, or any others you prefer). For this example I'm using the `roms/PONG2` game included with this guide. Unless you're using a very fancy editor, you should probably see something similar to Figure 1.

This doesn't seem very helpful. In fact it seems corrupted, is this okay? Well, don't worry your game is (probably) just fine. All computer programs are, at their core, just numbers saved into a file. These numbers can mean just about anything, it's only with context that a computer is able to put meaning to the values. Your text editor was designed to show and edit text in a language such as English, and thus will automatically try and use a language-focused context such as ASCII.

So we've learned that our Chip-8 file is not written in plain English, which we probably could have assumed already. So what does it look like? Fortunately, programs exist to display the raw contents of a file, and so we will need to use one of those "hex editors" to display the actual contents of our file.

In Figure 2, the numbers on the left of the vertical line are the *offset* values, how many bytes we are away from the start of the file. On the right hand side are the actual values stored in the file. Both the offsets and data values are displayed in hexadecimal (we'll be dealing with hexadecimal quite a bit).

Okay, we have numbers now, so that's an improvement. If those numbers don't correspond to English letters, what do they mean? These are the *instructions* for the Chip-8's CPU. Actually, each instruction is two bytes, which is why I've grouped them in pairs in the screenshot.

2.2 What is a CPU?

Let me take a moment to describe exactly what functionality a CPU provides, for those who aren't familiar. A CPU, for our purposes, does math. That's it. Now, when I say it "does math", this includes your usual items such as addition, subtraction, and checking if two numbers are equal or not. There are also some additional operations that are needed for the game to run, such as jumping to different sections of code, or fetching and saving numbers. The game file is entirely made up of mathematical operations for the CPU to preform.

This is what the numbers represent in our ROM file. All of the mathematical operations that the Chip-8 can perform have a corresponding number, called its *opcode*. A full list of the Chip-8's opcodes can be seen on this page. Whenever it is time for the emulator to perform another instruction (also referred to as a *tick* or a *cycle*), it will grab the next opcode from the game ROM and do whatever operation is specified by the opcode table; add, subtract, update what's drawn on the screen, whatever.

What about parameters? It's not enough to say "it's time to add", you need two numbers to actually add together, plus a place to put the sum when you're done. Other systems do this differently, but two bytes per operation is a lot of possible numbers, way more than the 35 operations that Chip-8 can actually do. The extra digits are used to pack in extra information into the opcode. The exact layout of this information varies between opcodes. The opcode table uses `N`'s to indicate literal hexadecmial numbers. `N` for single digit, `NN` for two digit, and `NNN` for three digit literal values, or for a *register* to be specified via `X` or `Y`.

[1] 'ROM' stands for "Read-only memory". It is a type of memory that is hard written at manufacturing time and cannot be modified by the computer system. For the purposes of this guide "ROM file" will be used interchangeably with "game file".

Figure 1: Raw Chip-8 ROM file

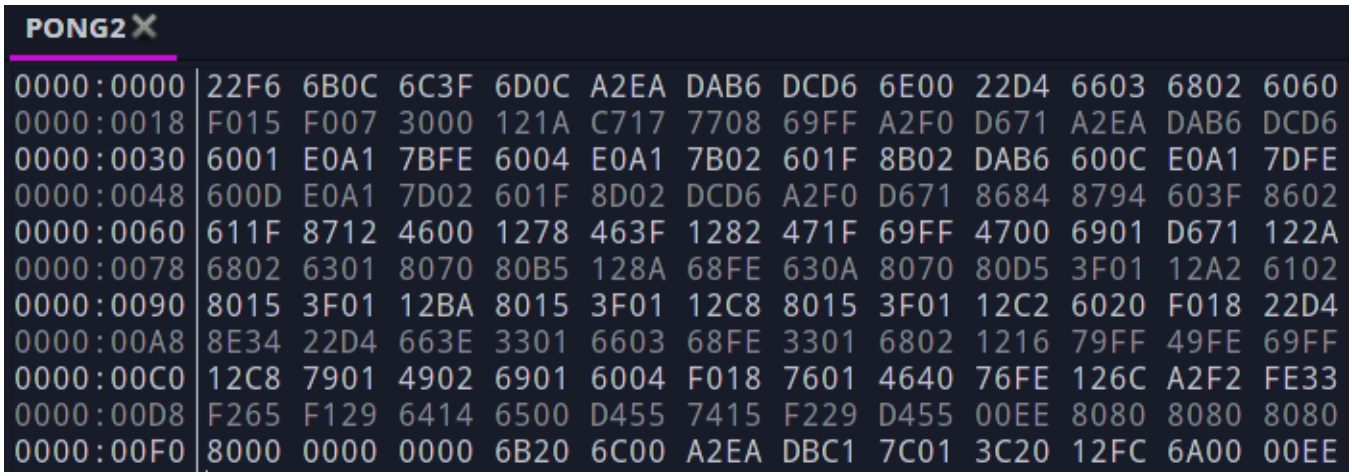

Figure 2: Chip-8 ROM file

2.3 What are Registers?

Which brings us to our next topic. What on earth is a register? A register is a specially designated place to store a single byte for use in instructions. This may sound rather similar to RAM (which we'll cover in a second), and in some ways it is. If you're going to actually build a computer, there are significant hardware differences between the two, but for our purposes while RAM is a big array of numbers, there are usually way fewer registers, they are all named, and they are directly used as part of a CPU operation. For many computers, if you want to use a value in RAM, it has to be copied into a register first, and then used.

The Chip-8 has sixteen registers that can be used freely by the programmer, named V0 through VF (0-15 in hexadecimal). As an example of how registers can work, let's look at one of the addition operations in our opcode table. There is an operation for adding the values of two registers together, VX += VY, encoded as `8XY4`. The `8XY4` is used for pattern matching the opcodes. If the current opcode begins with an 8 and ends with a 4, then this is the matching operation. The middle two digits then specify which registers we are to use. Let's say our opcode is `8124`. It begins with an 8 and ends with a 4, so we're in the right place. For this instruction we will be using the values stored in V1 and V2, as those match the other two digits. Let's say V1 stores 5 and V2 stores 10, this operation would add those two values and replace what was in V1, thus V1 now holds 15.

Chip-8 contains a few more registers, but they serve very specific purposes. One of the most significant ones is the *program counter* (PC), which for Chip-8 can store 16-bit values. I've made vague references to our "current opcode", but how do we keep track of where we are? Our emulator is simulating a computer running a program. It needs to start at the beginning of our game and move from opcode to opcode, doing the instructions as it's told. The PC holds the index of what game instruction we're currently working on. So it'll start at the first byte of the game, then move on to the third (remember all opcodes are two bytes, so opcode two is actually the third bytes), and so on and so forth. Some instructions can also tell the PC to go somewhere else in the game, but by default it will simply move forward opcode by opcode, and that is how the game executes.

2.4 What is RAM?

We have our sixteen V registers, but even for a simple system like a Chip-8, we really would like to be able to store more than 16 numbers at a time. This is now where RAM comes in. You're probably familiar with it in the context of your own computer, but RAM is a large array of numbers for the CPU to do with as it pleases. Chip-8 is not a physical system, so there is no standard amount of RAM it is supposed to have. However, emulation developers have more or less settled on 4096 bytes (4 KB), so our system will have the ability to store up to 4096 8-bit numbers in its RAM, frankly way more than most games will use.

Now, time for an important detail. The Chip-8 CPU has free access to read and write to RAM as it pleases. It does not, however, have access to our game ROM. Now with a name like "Read-Only Memory", it's safe to assume that we weren't going to be able to overwrite it, but the CPU can't read from ROM either? This whole guide so far has been about how the CPU deals with our ROM data! The CPU of course needs to be able to read the game ROM, and the way it does this is when game first begins running, the entire game is copied into RAM[2]. Think of it this way. It is rather slow and wasteful to open our game file just to read a little bit of data over and over again. Instead, we want to be able to copy as much data as we can into RAM for us to be able to more quickly use. The other catch to this is that somewhat confusingly, the ROM data is not loaded into very start of RAM. Instead, it's offset by 512 bytes (0x200). I.E. the first byte of ROM is loaded at start into RAM address 0x200. The second ROM byte into 0x201, etc.

Why do this? Why doesn't the Chip-8 simply store the game at the start of RAM and call it a day? The short answer is back when the Chip-8 was designed, there was a much more limited amount of RAM, and only one program could run at a given time. That means that the Chip-8 program itself would have to go somewhere, and the first 0x200 were allocated for that reason. Thus, modern Chip-8 emulators need to keep that in mind, as games are still written with that concept. Our system will actually use a little bit of that empty space, but we will cover that later.

That's pretty much it for the conceptual portion of this document. The Chip-8 has various other systems, but they're pretty straight-forward such as the display and keys. The remainder of this guide builds the emulator piece by piece in Rust, with a more depth focus on the various components as they come up.

[2]Therefore, Chip-8 games have a maximum size of 4 KB, any larger and they can't be loaded completed in.

3 Setup

Since the eventual goal of this project is to have an emulator which can be built for both the desktop and a web browser, we're going to structure this project slightly unusually. Between the two builds, the actual emulation of the Chip-8 system should be exactly the same, only things like reading in a file and displaying the screen will be different between desktop and browser. To that end, we will create the backend (which I will call the *core*) as its own crate to be used by both our future frontends.

Move into the folder where you will store your project and run the following command. Do not include the `$`, that is simply to indicate that this is a terminal instruction.

```
$ cargo init chip8_core --lib
```

The `--lib` flag tells `cargo` to create a library rather than an executable module. This is where our emulation backend will go. I called it `chip8_core` for the purposes of this project, but you are free to call it whatever you like.

As for the frontend, we'll create an additional crate to hold that code:

```
$ cargo init desktop
```

Unlike our core, this creates an actual executable project. If you've done it correctly, your folder structure should now look like this:

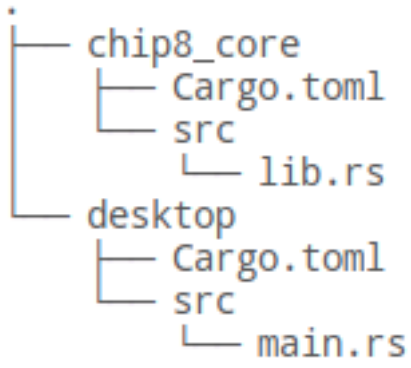

Figure 3: Initial file structure

All that remains is to tell our `desktop` frontend where to find `chip8_core`. Open up `desktop/Cargo.toml` and under `[dependencies]` add the line:

```
chip8_core = { path = "../chip8_core" }
```

Since `chip8_core` is currently empty, this doesn't actually add anything, but it's something that will need to be done eventually. Go ahead and try and compile and run the project, just to be sure everything is working. From inside the `desktop` directory, run:

```
$ cargo run
```

If everything has been setup correctly, "Hello, world!" should print out. Great! Next, we'll begin emulating the CPU and start creating something a little more interesting.

3.1 Defining our Emulator

The most fundamental part of the system is the CPU, so we will begin there when creating our emulator. For the time being, we will mostly be developing the `chip8_core` backend, then coming back to supply our frontend when it's well along. We will begin by working in the `chip8_core/src/lib.rs` file (you can delete the auto-generated `test` code). Before we add any functionality, let's refresh some of the concepts of what we're about to do.

Emulation is simply executing a program originally written for a different system, so it functions very similarly to the execution of a modern computer program. When running any old program, a line of code is read, understood by the computer to perform some task such as modifying a variable, making a comparison, or jumping to a different

line of code; that action is then taken, and the execution moves to the next line to repeat this process. If you've studied compilers, you'll know that when a system is running code, it's not doing it line by line, but instead converts the code into instructions understood by the processor, and then performs this loop upon those. This is exactly how our emulator will function. We will traverse value by value in our game program, **fetching** the instruction stored there, **decoding** the operation that must be done, and then **executing** it, before moving on to the next. This *fetch-decode-execute* loop will form the core of our CPU emulation.

With this in mind, let's begin by defining a class which will manage our emulator. In `lib.rs`, we'll add a new empty struct:

```
pub struct Emu {
}
```

This struct will be the main object for our emulation backend, and thus must handle running the actual game, and be able to pass need information back and forth from the frontend (such as what's on the screen and button presses).

3.2 Program Counter

But what to put in our `Emu` object? As discussed previously, the program needs to know where in the game it's currently executing. All CPUs accomplish this by simply keeping an index of the current instruction, stored into a special *register* known as the *Program Counter*, or PC for short. This will be key for the *fetch* portion of our loop, and will increment through the game as it runs, and can even be modified manually by some instructions (for things such as jumping into a different section of code or calling a subroutine). Let's add this to our struct:

```
pub struct Emu {
    pc: u16,
}
```

3.3 RAM

While we could read from our game file every time we need a new instruction, this is very slow, inefficient, and simply not how real systems do it. Instead, the Chip-8 is designed to copy the entire game program into its own RAM space, where it can then read and written to as needed. It should be noted that many systems, such as the Game Boy, do not allow the CPU to overwrite area of memory with the game stored in it (you wouldn't want a bug to completely corrupt game code). However, the Chip-8 has no such restriction. Since the Chip-8 was never a physical system, there isn't an official standard for how much memory it should have. However, it was originally designed to be implemented on computers with 4096 bytes (4 KB) of RAM, so that's how much we shall give it as well. Most programs won't come close to using it all, but it's there if they need it. Let's define that in our program as well.

```
const RAM_SIZE: usize = 4096;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
}
```

3.4 The display

Next, the display. Chip-8 uses a 64x32 monochrome (1 bit per pixel) display. There's nothing too special about this, however it is one of the few things in our backend that will need to be accessible to our various frontends, and to the user. Unlike many systems, Chip-8 does not automatically clear its screen to redraw every frame, instead the screen state is maintained, and new sprites are drawn onto it (there is a clear screen command however). We can keep this screen data in an array in our Emu object. Chip-8 is also more basic than most systems as we only have to deal with two colors - black and white. Since this is a 1-bit display, we can simply store an array of booleans like so:

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
```

```
pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
}
```

You'll also notice that unlike our previous constant, We've defined the screen dimensions as public constants. This is one of the few pieces of information that the frontend will need, as it will be the component to actually draw the screen.

3.5 V Registers

There's a few other items we'll need to add to get started. While the system has quite a bit of RAM to work with, RAM access is usually considered fairly slow (but still orders of magnitude faster than reading from disc). To speed things up, the Chip-8 defines sixteen 8-bit *registers* which the game can use as it pleases for much faster operation. These are referred to as the *V registers*, and are usually numbered in hex from V0 to VF (I'm honestly not sure what the *V* stands for), and we'll group them together in one array in our Emu struct.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
}
```

3.6 I Register

There is also another 16-bit register known as the *I register*, which is used for indexing into RAM for reads and writes. We'll get into the finer details of how this works later on, for now we simply need to have it.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
}
```

3.7 The stack

The CPU also has a small *stack*, which is an array of 16-bit values that the CPU can read and write to. The stack differs from regular RAM as the stack can only be read/written to via a "First In, First Out (FIFO)" principle (like a stack of pancakes!), when you go to grab (pop) a value, you remove the last one you added (pushed). Unlike many systems, however, the stack is not general purpose. The only times the stack is allowed to be used is when you are entering or exiting a subroutine, where the stack is used to know where you started so you can return after the routine ends. Again, Chip-8 doesn't officially state how many numbers the stack can hold, but 16 is a typical number for emulation developers. There are a number of different ways we could implement our stack, perhaps the

easiest way would be to use the `std::collections::VecDeque` object from Rust's standard library. This works fine for a Desktop-only build, however at the time of writing, many items in the `std` library don't work for WebAssembly builds. Instead we will do it the old fashioned way, with a statically sized array and an index so we know where the top of the stack is, known as the Stack Pointer (SP).

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;
const STACK_SIZE: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
    sp: u16,
    stack: [u16; STACK_SIZE],
}
```

3.8 The timers

This has been a lot to process at once, but we're now at the final items. Chip-8 also has two other special registers that it uses as *timers*. The first, the *Delay Timer* is used by the system as a typical timer, counting down every cycle and performing some action if it hits 0. The *Sound Timer* on the other hand, also counts down every clock cycle, but upon hitting 0 emits a noise. Setting the *Sound Timer* to 0 is the only way to emit audio on the Chip-8, as we will see later. These are both 8-bit registers, and must be supported for us to continue.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;
const STACK_SIZE: usize = 16;
const NUM_KEYS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
    sp: u16,
    stack: [u16; STACK_SIZE],
    keys: [bool; NUM_KEYS],
    dt: u8,
    st: u8,
}
```

3.9 Initialization

That should do it for now! Let's go ahead and implement a `new` constructor for this class before we move onto the next part. Following the `struct` definition, we'll implement and set the default values:

```
impl Emu {
    pub fn new() -> Self {
        Self {
```

```
            pc: 0x200,
            ram: [0; NUM_REGS],
            screen: [false; SCREEN_WIDTH * SCREEN_HEIGHT],
            v_reg: [0; NUM_REGS],
            i_reg: 0,
            sp: 0,
            stack: [0; STACK_SIZE],
            keys: [false; NUM_KEYS],
            dt: 0,
            st: 0,
        }
    }
}
```

Everything seems pretty straightforward, we simply initialize all values and arrays to zero... except for our Program Counter, which gets set to 0x200 (512 in decimal). I mentioned the reasoning behind this in the previous chapter, but the emulator has to know where the beginning of the program is, and it is Chip-8 standard that the beginning of all Chip-8 programs will be loaded in starting at RAM address 0x200. This number will come up again, so let's define it as a constant.

```
// -- Unchanged code omitted --

const START_ADDR: u16 = 0x200;

// -- Unchanged code omitted --

impl Emu {
    pub fn new() -> Self {
        Self {
            pc: START_ADDR,
            ram: [0; NUM_REGS],
            screen: [false; SCREEN_WIDTH * SCREEN_HEIGHT],
            v_reg: [0; NUM_REGS],
            i_reg: 0,
            sp: 0,
            stack: [0; STACK_SIZE],
            keys: [false; NUM_KEYS],
            dt: 0,
            st: 0,
        }
    }
}
```

That wraps up this part! With the basis of our emulator underway, we can begin to implement the execution!

4 Implementing Emulation Methods

We have now created our `Emu` struct and defined a number of variables for it to manage, as well as defined an initialization function. Before we move on, there are a few useful methods we should add to our object now which will come in use once we begin implementation of the instructions.

4.1 Push and Pop

We have added both a `stack` array as well as a pointer `sp` to manage the CPU's stack, however it will be useful to implement both a `push` and `pop` method so we can access it easily.

```
impl Emu {
    // -- Unchanged code omitted --

    fn push(&mut self, val: u16) {
        self.stack[self.sp as usize] = val;
        self.sp += 1;
    }

    fn pop(&mut self) -> u16 {
        self.sp -= 1;
        self.stack[self.sp as usize]
    }

    // -- Unchanged code omitted --
}
```

These are pretty straightforward. `push` adds the given 16-bit value to the spot pointed to by the Stack Pointer, then moves the pointer to the next position. `pop` performs this operation in reverse, moving the SP back to the previous value then returning what is there. Note that attempting to pop an empty stack results in an underflow panic. You are welcome to add extra handling here if you like, but in the event this were to occur, that would indicate a bug with either our emulator or the game code, so I feel that a complete panic is acceptable.

4.2 Font Sprites

We haven't yet delved into how the Chip-8 screen display works, but the gist for now is that it renders *sprites* which are stored in memory to the screen, one line at a time. It is up to the game developer to correctly load their sprites in before copying them over. However wouldn't it be nice if the system automatically had sprites for commonly used things, such as numbers? I mentioned earlier that our PC will begin at address 0x200, leaving the first 512 intentionally empty. Most modern emulators will use that space to store the sprite data for font characters of all the hexadecimal digits, that is characters of 0-9 and A-F. Strictly speaking, we could store this data at any fixed position in RAM, but this space is already defined as empty anyway. Each character is five bytes long, with each byte making up a row, thus the characters are 8x5 pixels each. To see how this works, make a note of the following diagram.

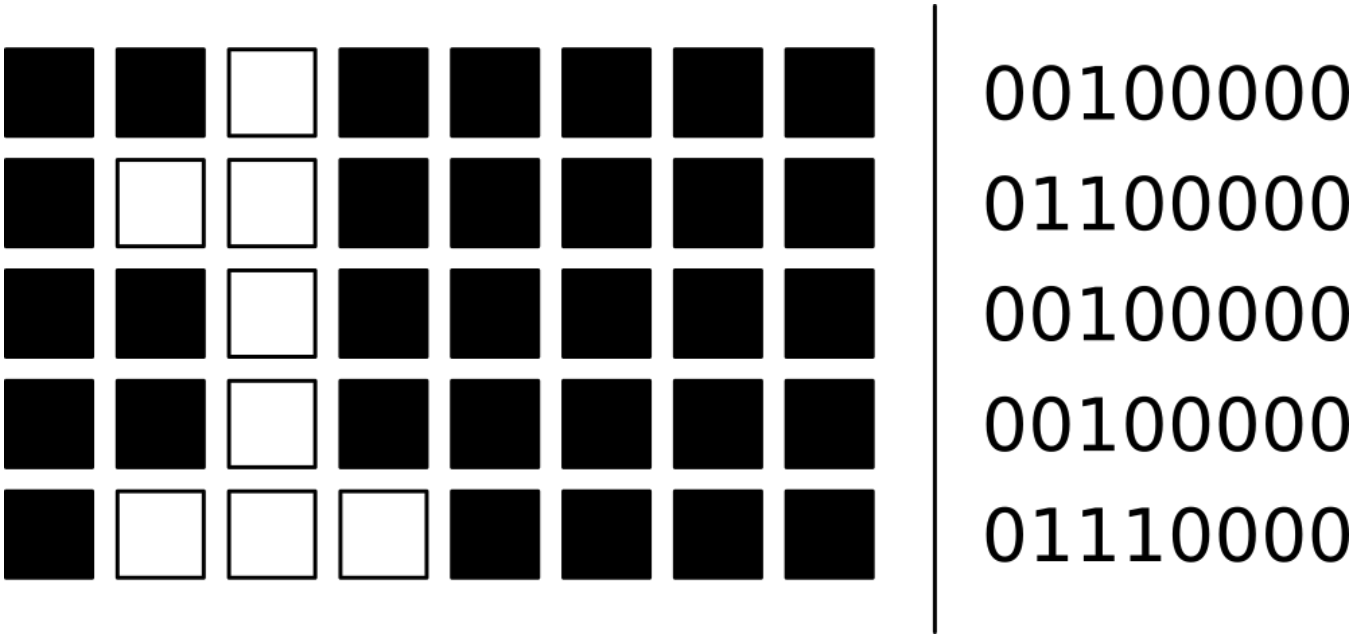

Figure 4: Chip-8 Font Sprite

On the left, we have how the font sprite will appear on screen. On the right, there is each row encoded into binary. Each pixel is assigned a bit, which corresponds to whether that pixel will be white or black. *Every* sprite in Chip-8 is eight pixels wide, which means a pixel row requires 8-bits (1 byte). The above diagram shows the layout of the "1" character sprite. The fonts sprites don't need all 8 bits of width, so they all have black right halves. Sprites have been created for all of the hexadecimal digits, and are required to be present somewhere in RAM for some games to function. Later in this guide we will cover the instruction that handles these sprites, which will show how these are loaded and how the emulator knows where to find them. For now, we simply need to define them. We will do so with a constant array of bytes; at the top of `lib.rs`, add:

```
const FONTSET_SIZE: usize = 80;

const FONTSET: [u8; FONTSET_SIZE] = [
    0xF0, 0x90, 0x90, 0x90, 0xF0, // 0
    0x20, 0x60, 0x20, 0x20, 0x70, // 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3
    0x90, 0x90, 0xF0, 0x10, 0x10, // 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6
    0xF0, 0x10, 0x20, 0x40, 0x40, // 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9
    0xF0, 0x90, 0xF0, 0x90, 0x90, // A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, // B
    0xF0, 0x80, 0x80, 0x80, 0xF0, // C
    0xE0, 0x90, 0x90, 0x90, 0xE0, // D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, // E
    0xF0, 0x80, 0xF0, 0x80, 0x80  // F
];
```

You can see the bytes outlined in the "1" diagram above, all of the other letters work in a similar way. Now that these are outlined, we need to load them into RAM. Modify `Emu::new()` to copy those values in:

```
pub fn new() -> Self {
    let mut new_emu = Self {
        pc: START_ADDR,
```

```
            ram: [0; RAM_SIZE],
            screen: [false; SCREEN_WIDTH * SCREEN_HEIGHT],
            v_reg: [0; NUM_REGS],
            i_reg: 0,
            sp: 0,
            stack: [0; STACK_SIZE],
            keys: [false; NUM_KEYS],
            dt: 0,
            st: 0,
        };

        new_emu.ram[..FONTSET_SIZE].copy_from_slice(&FONTSET);

        new_emu
    }
```

This initializes our `Emu` object in the same way as before, but copies in our character sprite data into RAM before returning it.

It will also be useful to be able to reset our emulator without having to create a new object. There are fancier ways of doing this, but we'll just keep it simple and create a function that resets our member variables back to their original values when called.

```
pub fn reset(&mut self) {
    self.pc = START_ADDR;
    self.ram = [0; RAM_SIZE];
    self.screen = [false; SCREEN_WIDTH * SCREEN_HEIGHT];
    self.v_reg = [0; NUM_REGS];
    self.i_reg = 0;
    self.sp = 0;
    self.stack = [0; STACK_SIZE];
    self.keys = [false; NUM_KEYS];
    self.dt = 0;
    self.st = 0;
    self.ram[..FONTSET_SIZE].copy_from_slice(&FONTSET);
}
```

4.3 Tick

With the creation of our `Emu` object completed (for now), we can begin to define how the CPU will process each instruction and move through the game. To summarize what was described in the previous parts, the basic loop will be:

1. Fetch the value from our game (loaded into RAM) at the memory address stored in our Program Counter.
2. Decode this instruction.
3. Execute, which will possibly involve modifying our CPU registers or RAM.
4. Move the PC to the next instruction and repeat.

Let's begin by adding the opcode processing to our `tick` function, beginning with the fetching step:

```
// -- Unchanged code omitted --

pub fn tick(&mut self) {
    // Fetch
    let op = self.fetch();
    // Decode
    // Execute
}

fn fetch(&mut self) -> u16 {
```

```
}
```

The `fetch` function will only be called internally as part of our `tick` loop, so it doesn't need to be public. The purpose of this function is to grab the instruction we are about to execute (known as an *opcode*) for use in the next steps of this cycle. If you're unfamiliar with Chip-8's instruction format, I recommend you refresh up with the overview from the earlier chapters.

Fortunately, Chip-8 is easier than many systems in two regards. For one, there's only 35 opcodes to deal with as opposed to the hundreds that many processors support. In addition, many systems store additional parameters for each opcode in subsequent bytes (such as operands for addition), Chip-8 encodes these into the opcode itself. Due to this, Chip-8 opcodes are larger than that of other systems, all of them are 2 bytes, but the entire instruction is stored in those two bytes, while other contemporary system might consume between 1 and 3 bytes per cycle.

Each opcode is encoded differently, but fortunately since all instructions consume two bytes, the fetch operation is the same for all of them, and implemented as such:

```
fn fetch(&mut self) -> u16 {
    let higher_byte = self.ram[self.pc as usize] as u16;
    let lower_byte = self.ram[(self.pc + 1) as usize] as u16;
    let op = (higher_byte << 8) | lower_byte;
    self.pc += 2;
    op
}
```

This function fetches the 16-bit opcode stored at our current Program Counter. We store values in RAM as 8-bit values, so we fetch two and combine them as Big Endian. The PC is then incremented by the two bytes we just read, and our fetched opcode is returned for further processing.

4.4 Timer Tick

The Chip-8 specification also mentions two special purpose *timers*, the Delay Timer and the Sound Timer. While the `tick` function operates once every CPU cycle, these timers are modified instead once every frame, and thus need to be handled in a separate function. Their behavior is rather simple, every frame both decrease by one. If the Sound Timer is set to one, the system will emit a 'beep' noise. If they ever hit zero, they do not automatically reset, they will remain at zero until the game manually resets them to some value.

```
pub fn tick_timers(&mut self) {
    if self.dt > 0 {
        self.dt -= 1;
    }

    if self.st > 0 {
        if self.st == 1 {
            // BEEP
        }
        self.st -= 1;
    }
}
```

Audio is the one thing that this guide won't cover, mostly due to increased complexity in getting audio to work in both our desktop and web browser frontends. For now we'll simply leave a comment where the beep would occur, but any curious readers are encouraged to implement it themselves (and then tell me how they did it).

5 Opcode Execution

In the previous section, we fetched our opcode and were preparing to decode which instruction it corresponds to to execute that instruction. Currently, our `tick` function looks like this:

```
pub fn tick(&mut self) {
    // Fetch
    let op = self.fetch();
    // Decode
    // Execute
}
```

This implies a bit that decode and execute will be their own separate functions. While they could be, personally for Chip-8 I think it's easier to simply perform the operation as we determine it, rather than involving another function call. Our `tick` function thus becomes this:

```
pub fn tick(&mut self) {
    // Fetch
    let op = self.fetch();
    // Decode & execute
    self.execute(op);
}

fn execute(&mut self, op: u16) {
}
```

Our next step is to *decode*, which is to determine exactly which operation we're dealing with. The Chip-8 opcode cheatsheet has all of the available opcodes, how to interpret their parameters, and some notes on what they mean (if you don't like my table, there are many similar examples online). You will need to reference this often, in order for the emulator to be complete, each and every one of them must be implemented.

5.1 Pattern Matching

Fortunately, Rust has a very robust and useful pattern matching feature we can use to our advantage. However, we will need to separate out each hex "digit" before we do so.

```
fn execute(&mut self, op: u16) {
    let digit1 = (op & 0xF000) >> 12;
    let digit2 = (op & 0x0F00) >> 8;
    let digit3 = (op & 0x00F0) >> 4;
    let digit4 = op & 0x000F;
}
```

Perhaps not the cleanest code, but we need each hex digit separately. From here, we can create a `match` statement where we can specify the patterns for all of our opcodes:

```
fn execute(&mut self, op: u16) {
    let digit1 = (op & 0xF000) >> 12;
    let digit2 = (op & 0x0F00) >> 8;
    let digit3 = (op & 0x00F0) >> 4;
    let digit4 = op & 0x000F;

    match (digit1, digit2, digit3, digit4) {
        (_, _, _, _) => unimplemented!("Unimplemented opcode: {}", op),
    }
}
```

Rust's `match` statement demands that all possible options be taken into account which is done with the `_` variable, which captures "everything else". Inside, we'll use the `unimplemented!` macro to cause the program to panic if it reaches that point. By the time we finish adding all opcodes, the Rust compiler demands that we still have an "everything else" statement, but we should never hit it.

A brief side note for any developers who might be curious about other systems. While a long `match` statement would certainly work for other architectures, it is usually more common to implement instructions in their own functions, and either use a lookup table or programmatically determine which function is correct. Chip-8 is somewhat unusual as it stores instruction parameters into the opcode itself, meaning we need a lot of wild cards to match the instructions. Since there are a relatively small number of them, a `match` statement works well here.

With the framework setup, let's dive in!

5.2 Intro to Implementing Opcodes

The following pages individually discuss how all of Chip-8's instructions work, and include code of how to implement them. They are there to assist any programmers who are confused on how to proceed or how to interpret some of the more nuanced behavior of the system. You are welcome to simply follow along and implement instruction by instruction, but before you do that, you may want to look forward to the next section and begin working on some of the frontend code. Currently we have no way of actually running our emulator, and it may be useful to some to be able to attempt to load and run a game for debugging. However, do remember that the emulator will likely crash rather quickly unless all of the instructions are implemented. Personally, I prefer to work on the instructions first before working on the other moving parts (hence why this guide is laid out the way it is).

With that disclaimer out of the way, let's proceed to working on each of the Chip-8 instructions in turn.

5.2.1 0000 - Nop

Our first instruction is the simplest one - do nothing. This may seem like a silly one to have, but sometimes it's needed for timing or alignment purposes. In any case, we simply need to move on to the next opcode (which was already done in `fetch`), and return.

```
match (digit1, digit2, digit3, digit4) {
    // NOP
    (0, 0, 0, 0) => return,
    (_, _, _, _) => unimplemented!("Unimplemented opcode: {}", op),
}
```

5.2.2 00E0 - Clear screen

Opcode 0x00E0 is the instruction to clear the screen, which means we need to reset our screen buffer to be empty again.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // CLS
    (0, 0, 0xE, 0) => {
        self.screen = [false; SCREEN_WIDTH * SCREEN_HEIGHT];
    },

    // -- Unchanged code omitted --
}
```

5.2.3 00EE - Return from Subroutine

We haven't yet spoken about subroutines (aka functions) and how they work. Entering into a subroutine works in the same way as just a plain jump; we move the PC to the specified address and resume execution from there. Unlike a jump, a subroutine is expected to complete at some point, and we will need to return back to the point where we entered. This is where our stack comes in. When we enter a subroutine, we simply push our address onto the stack, run the routine's code, and when we're ready to return we pop that value off our stack and execute from that point again. A stack also allows us to maintain return addresses for nested subroutines while ensuring they are returned in the correct order.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --
```

```
    // RET
    (0, 0, 0xE, 0xE) => {
        let ret_addr = self.pop();
        self.pc = ret_addr;
    },

    // -- Unchanged code omitted --
}
```

5.2.4 1NNN - Jump

The jump instruction is pretty easy to add, simply move the PC to the given address.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // JMP NNN
    (1, _, _, _) => {
        let nnn = op & 0xFFF;
        self.pc = nnn;
    },

    // -- Unchanged code omitted --
}
```

The main thing to notice here is that this opcode is defined by '0x1' being the most significant digit. The other digits are used as parameters for this operation, hence the _ placeholder in our match statement, here we want anything starting with a 1, but ending in any three digits to enter this statement.

5.2.5 2NNN - Call Subroutine

The opposite of our 'Return from Subroutine' operation, we are going to add our current PC to the stack, and then jump to the given address. If you skipped straight here, I recommend reading the *Return* section for additional context.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // CALL NNN
    (2, _, _, _) => {
        let nnn = op & 0xFFF;
        self.push(self.pc);
        self.pc = nnn;
    },

    // -- Unchanged code omitted --
}
```

5.2.6 3XNN - Skip next if VX == NN

This opcode is first of a few that follow a similar pattern. For those who are unfamiliar with assembly, being able to skip a line gives similar functionality to an if-else block. We can make a comparison, and if true go to one instruction, and if false go somewhere else. This is also the first opcode which will use one of our *V registers*. In this case, the second digit tells us which register to use, while the last two digits provide the raw value.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // SKIP VX == NN
```

```
    (3, _, _, _) => {
        let x = digit2 as usize;
        let nn = (op & 0xFF) as u8;
        if self.v_reg[x] == nn {
            self.pc += 2;
        }
    },

    // -- Unchanged code omitted --
}
```

The implementation works like this: since we already have the second digit saved to a variable, we will reuse it for our 'X' index, although cast to a `usize`, as Rust requires all array indexing to be done with a `usize` variable. If that value stored in that register equals `nn`, then we skip the next opcode, which is the same as skipping our PC ahead by two bytes.

5.2.7 4XNN - Skip next if VX != NN

This opcode is exactly the same as the previous, with the change that we skip if the compared values are not equal.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // SKIP VX != NN
    (4, _, _, _) => {
        let x = digit2 as usize;
        let nn = (op & 0xFF) as u8;
        if self.v_reg[x] != nn {
            self.pc += 2;
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.8 5XY0 - Skip next if VX == VY

A similar operation again, however we now use the third digit to index into another *V Register*. You will also notice that while the least significant digit is not used in the operation, this opcode requires it to be 0.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // SKIP VX == VY
    (5, _, _, 0) => {
        let x = digit2 as usize;
        let y = digit3 as usize;
        if self.v_reg[x] == self.v_reg[y] {
            self.pc += 2;
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.9 6XNN - VX = NN

This operation does not require too much explanation, set the *V Register* specified by the second digit to the value given.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX = NN
    (6, _, _, _) => {
        let x = digit2 as usize;
        let nn = (op & 0xFF) as u8;
        self.v_reg[x] = nn;
    },

    // -- Unchanged code omitted --
}
```

5.2.10 7XNN - VX += NN

This operation adds the given value to the VX register. In the event of an overflow, Rust will panic, so we need to use a different method than the typical addition operator. Note also that while Chip-8 has a carry flag (more on that later), it is not modified by this operation.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX += NN
    (7, _, _, _) => {
        let x = digit2 as usize;
        let nn = (op & 0xFF) as u8;
        self.v_reg[x] = self.v_reg[x].wrapping_add(nn);
    },

    // -- Unchanged code omitted --
}
```

5.2.11 8XY0 - VX = VY

Like the `VX = NN` operation, but the source value is from the VY register.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX = VY
    (8, _, _, 0) => {
        let x = digit2 as usize;
        let y = digit3 as usize;
        self.v_reg[x] = self.v_reg[y];
    },

    // -- Unchanged code omitted --
}
```

5.2.12 8XY1, 8XY2, 8XY3 - Bitwise operations

The `8XY1`, `8XY2`, and `8XY3` opcodes are all similar functions, so rather than repeat myself three times over, I'll implement the *OR* operation, and allow the reader to implement the other two.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX |= VY
    (8, _, _, 1) => {
```

```
        let x = digit2 as usize;
        let y = digit3 as usize;
        self.v_reg[x] |= self.v_reg[y];
    },

    // -- Unchanged code omitted --
}
```

5.2.13 8XY4 - VX += VY

This operation has two aspects to make note of. Firstly, this operation has the potential to overflow, which will cause a panic in Rust if not handled correctly. Secondly, this operation is the first to utilize the `VF` flag register. I've touched upon it previously, but while the first 15 *V* registers are general usage, the final 16th (0xF) register doubles as the *flag register*. Flag registers are common in many CPU processors; in the case of Chip-8 it also stores the *carry flag*, basically a special variable that notes if the last application operation resulted in an overflow/underflow. Here, if an overflow were to happen, we need to set the `VF` to be 1, or 0 if not. With these two aspects in mind, we will use Rust's `overflowing_add` attribute, which will return a tuple of both the wrapped sum, as well as a boolean of whether an overflow occurred.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX += VY
    (8, _, _, 4) => {
        let x = digit2 as usize;
        let y = digit3 as usize;

        let (new_vx, carry) = self.v_reg[x].overflowing_add(self.v_reg[y]);
        let new_vf = if carry { 1 } else { 0 };

        self.v_reg[x] = new_vx;
        self.v_reg[0xF] = new_vf;
    },

    // -- Unchanged code omitted --
}
```

5.2.14 8XY5 - VX -= VY

This is the same operation as the previous opcode, but with subtraction rather than addition. The key distinction is that the `VF` carry flag works in the opposite fashion. The addition operation would set the flag to 1 if an overflow occurred, here if an underflow occurs, it is set to 0, and vice versa. The `overflowing_sub` method will be of use to us here.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX -= VY
    (8, _, _, 5) => {
        let x = digit2 as usize;
        let y = digit3 as usize;

        let (new_vx, borrow) = self.v_reg[x].overflowing_sub(self.v_reg[y]);
        let new_vf = if borrow { 0 } else { 1 };

        self.v_reg[x] = new_vx;
        self.v_reg[0xF] = new_vf;
    },
```

```
    // -- Unchanged code omitted --
}
```

5.2.15 8XY6 - VX »= 1

This operation performs a single right shift on the value in VX, with the bit that was dropped off being stored into the `VF` register. Unfortunately, there isn't a built-in Rust `u8` operator to catch the dropped bit, so we will have to do it ourself.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX >>= 1
    (8, _, _, 6) => {
        let x = digit2 as usize;
        let lsb = self.v_reg[x] & 1;
        self.v_reg[x] >>= 1;
        self.v_reg[0xF] = lsb;
    },

    // -- Unchanged code omitted --
}
```

5.2.16 8XY7 - VX = VY - VX

This operation works the same as the previous VX -= VY operation, but with the operands in the opposite direction.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX = VY - VX
    (8, _, _, 7) => {
        let x = digit2 as usize;
        let y = digit3 as usize;

        let (new_vx, borrow) = self.v_reg[y].overflowing_sub(self.v_reg[x]);
        let new_vf = if borrow { 0 } else { 1 };

        self.v_reg[x] = new_vx;
        self.v_reg[0xF] = new_vf;
    },

    // -- Unchanged code omitted --
}
```

5.2.17 8XYE - VX «= 1

Similar to the right shift operation, but we store the value that is overflowed in the flag register.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX <<= 1
    (8, _, _, 0xE) => {
        let x = digit2 as usize;
        let msb = (self.v_reg[x] >> 7) & 1;
        self.v_reg[x] <<= 1;
        self.v_reg[0xF] = msb;
    },
```

```
    // -- Unchanged code omitted --
}
```

5.2.18 9XY0 - Skip if VX != VY

Done with the 0x8000 operations, it's time to go back and add an opcode that was notably missing, skipping the next line if VX != VY. This is the same code as the 5XY0 operation, but with an inequality.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // SKIP VX != VY
    (9, _, _, 0) => {
        let x = digit2 as usize;
        let y = digit3 as usize;
        if self.v_reg[x] != self.v_reg[y] {
            self.pc += 2;
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.19 ANNN - I = NNN

This is the first instruction to utilize the *I Register*, which will be used in several additional instructions, primarily as an address pointer to RAM. In this case, we are simply setting it to the 0xNNN value encoded in this opcode.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // I = NNN
    (0xA, _, _, _) => {
        let nnn = op & 0xFFF;
        self.i_reg = nnn;
    },

    // -- Unchanged code omitted --
}
```

5.2.20 BNNN - Jump to V0 + NNN

While previous instructions have used the *V Register* specified within the opcode, this instruction always uses the first *V0* register. This operation moves the PC to the sum of the value stored in *V0* and the raw value 0xNNN supplied in the opcode.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // JMP V0 + NNN
    (0xB, _, _, _) => {
        let nnn = op & 0xFFF;
        self.pc = (self.v_reg[0] as u16) + nnn;
    },

    // -- Unchanged code omitted --
}
```

5.2.21 CXNN - VX = rand() & NN

Finally, something to shake up the monotony! This opcode is Chip-8's random number generation, with a slight twist, in that the random number is then AND'd with the lower 8-bits of the opcode. While the Rust development team has released a random generation crate, it is not part of its standard library, so we shall have to add it to our project.

In `chip8_core/Cargo.toml`, add the following line somewhere under `[dependencies]`:

```
rand = "^0.7.3"
```

Note: If you are planning on following this guide completely to its end, there will be a future change to how we include this library for web browser support. However, at this stage in the project, it is enough to specify it as is.

Time now to add RNG support and implement this opcode. At the top of `lib.rs`, we will need to import a function from the `rand` crate:

```
use rand::random;
```

We will then use the `random` function when implementing our opcode:

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX = rand() & NN
    (0xC, _, _, _) => {
        let x = digit2 as usize;
        let nn = (op & 0xFF) as u8;
        let rng: u8 = random();
        self.v_reg[x] = rng & nn;
    },

    // -- Unchanged code omitted --
}
```

Note that specifying `rnd` as a `u8` variable is necessary for the `random()` function to know which type it is supposed to generate.

5.2.22 DXYN - Draw Sprite

This is probably the single most complicated opcode, so allow me to take a moment to describe how it works in detail. Rather than drawing individual pixels or rectangles to the screen at a time, the Chip-8 display works by drawing *sprites*, images stored in memory that are copied to the screen at a specified (x, y). For this opcode, the second and third digits give us which *V Registers* we are to fetch our (x, y) coordinates from. So far so good. Chip-8's sprites are always 8 pixels wide, but can be a variable number of pixels tall, from 1 to 16. This is specified in the final digit of our opcode. I mentioned earlier that the *I Register* is used frequently to store an address in memory, and this is the case here; our sprites are stored row by row *beginning* at the address stored in *I*. So if we are told to draw a 3px tall sprite, the first row's data is stored at *I, followed by *I + 1, then *I + 2. This explains why all sprites are 8 pixels wide, each row is assigned a byte, which is 8-bits, one for each pixel, black or white. The last detail to note is that if *any* pixel is flipped from white to black or vice versa, the *VF* is set (and cleared otherwise). With these things in mind, let's begin.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // DRAW
    (0xD, _, _, _) => {
        // Get the (x, y) coords for our sprite
        let x_coord = self.v_reg[digit2 as usize] as u16;
        let y_coord = self.v_reg[digit3 as usize] as u16;
        // The last digit determines how many rows high our sprite is
        let num_rows = digit4;
```

```
            // Keep track if any pixels were flipped
            let mut flipped = false;
            // Iterate over each row of our sprite
            for y_line in 0..num_rows {
                // Determine which memory address our row's data is stored
                let addr = self.i_reg + y_line as u16;
                let pixels = self.ram[addr as usize];
                // Iterate over each column in our row
                for x_line in 0..8 {
                    // Use a mask to fetch current pixel's bit. Only flip if a 1
                    if (pixels & (0b10000000 >> x_line)) != 0 {
                        // Sprites should wrap around screen, so apply modulo
                        let x = (x_coord + x_line) as usize % SCREEN_WIDTH;
                        let y = (y_coord + y_line) as usize % SCREEN_HEIGHT;

                        // Get our pixel's index for our 1D screen array
                        let idx = x + SCREEN_WIDTH * y;
                        // Check if we're about to flip the pixel and set
                        flipped |= self.screen[idx];
                        self.screen[idx] ^= true;
                    }
                }
            }

            // Populate VF register
            if flipped {
                self.v_reg[0xF] = 1;
            } else {
                self.v_reg[0xF] = 0;
            }
        },

        // -- Unchanged code omitted --
    }
```

5.2.23 EX9E - Skip if Key Pressed

Time at last to introduce user input. When setting up our emulator object, I mentioned that there are 16 possible keys numbered 0 to 0xF. This instruction checks if the index stored in VX is pressed, and if so, skips the next instruction.

```
    match (digit1, digit2, digit3, digit4) {
        // -- Unchanged code omitted --

        // SKIP KEY PRESS
        (0xE, _, 9, 0xE) => {
            let x = digit2 as usize;
            let vx = self.v_reg[x];
            let key = self.keys[vx as usize];
            if key {
                self.pc += 2;
            }
        },

        // -- Unchanged code omitted --
    }
```

5.2.24 EXA1 - Skip if Key Not Pressed

Same as the previous instruction, however this time the next instruction is skipped if the key in question is not being pressed.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // SKIP KEY RELEASE
    (0xE, _, 0xA, 1) => {
        let x = digit2 as usize;
        let vx = self.v_reg[x];
        let key = self.keys[vx as usize];
        if !key {
            self.pc += 2;
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.25 FX07 - VX = DT

I mentioned the use of the *Delay Timer* when we were setting up the emulation structure. This timer ticks down every frame until reaching zero. However, that operation happens automatically, it would be really useful to be able to actually see what's in the *Delay Timer* for our game's timing purposes. This instruction does just that, and stores the current value into one of the *V Registers* for us to use.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // VX = DT
    (0xF, _, 0, 7) => {
        let x = digit2 as usize;
        self.v_reg[x] = self.dt;
    },

    // -- Unchanged code omitted --
}
```

5.2.26 FX0A - Wait for Key Press

While we already had instructions to check if keys are either pressed or released, this instruction does something very different. Unlike those, which checked the key state and then moved on, this instruction is *blocking*, meaning the whole game will pause and wait for as long as it needs to until the player presses a key. That means it needs to loop endlessly until something in our `keys` array turns true. Once a key is found, it is stored into VX. If more than one key is currently being pressed, it takes the lowest indexed one.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // WAIT KEY
    (0xF, _, 0, 0xA) => {
        let x = digit2 as usize;
        let mut pressed = false;
        for i in 0..self.keys.len() {
            if self.keys[i] {
                self.v_reg[x] = i as u8;
                pressed = true;
                break;
```

```
            }
        }

        if !pressed {
            // Redo opcode
            self.pc -= 2;
        }
    },

    // -- Unchanged code omitted --
}
```

You may be looking at this implementation and asking "why are we resetting the opcode and going through the entire fetch sequence again, rather than simply doing this in a loop?". Simply put, while we want this instruction to block future instructions from running, we do not want to block any new key presses from being registered. By remaining in a loop, we would prevent our key press code from ever running, causing this loop to never end. Perhaps inefficient, but much simpler than some sort of asynchronous checking.

5.2.27 FX15 - DT = VX

This operation works the other direction from our previous *Delay Timer* instruction. We need someway to reset the *Delay Timer* to a value, and this instruction allows us to copy over a value from a *V Register* of our choosing.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // DT = VX
    (0xF, _, 1, 5) => {
        let x = digit2 as usize;
        self.dt = self.v_reg[x];
    },

    // -- Unchanged code omitted --
}
```

5.2.28 FX18 - ST = VX

Almost the exact same instruction as the previous, however this time we are going to store the value from VX into our *Sound Timer*.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // ST = VX
    (0xF, _, 1, 8) => {
        let x = digit2 as usize;
        self.st = self.v_reg[x];
    },

    // -- Unchanged code omitted --
}
```

5.2.29 FX1E - I += VX

Instruction ANNN sets I to the encoded 0xNNN value, but sometimes it is useful to be able to simply increment the value. This instruction takes the value stored in VX and adds it to the *I Register*. In the case of an overflow, the register should simply roll over back to 0, which we can accomplish with Rust's `wrapping_add` method.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --
```

```
    // I += VX
    (0xF, _, 1, 0xE) => {
        let x = digit2 as usize;
        let vx = self.v_reg[x] as u16;
        self.i_reg = self.i_reg.wrapping_add(vx);
    },

    // -- Unchanged code omitted --
}
```

5.2.30 FX29 - Set I to Font Address

This is another tricky instruction where it may not be clear how to progress at first. If you recall, we stored an array of font data at the very beginning of RAM when initializing the emulator. This instruction wants us to take in the number to print on screen (from 0 to 0xF), and store the RAM address of that sprite into the *I Register*. We are actually free to store those sprites anywhere we wanted, so long as we are consistent and point to them correctly here. However, we stored them in a very convenient location, at the beginning of RAM. Let me show you what I mean by printing out some of the sprites and their RAM locations.

Character	RAM Address
0	0
1	5
2	10
3	15
...	...
E (14)	70
F (15)	75

You'll notice that since all of our font sprites take up five bytes each, their RAM address is simply their value times 5. If we happened to store the fonts in a different RAM address, we could still follow this rule, however we'd have to apply an offset to where the block begins.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // I = FONT
    (0xF, _, 2, 9) => {
        let x = digit2 as usize;
        let c = self.v_reg[x] as u16;
        self.i_reg = c * 5;
    },

    // -- Unchanged code omitted --
}
```

5.2.31 FX33 - I = BCD of VX

Most of the instructions for Chip-8 are rather self-explanitory, and can be implemented quite easily just by hearing a vague description. However, there are a few that are quite tricky, such as drawing to a screen and this one, storing the Binary-Coded Decimal of a number stored in VX into the *I Register*. I encourage you to read up on BCD if you are unfamiliar with it, but a brief refresher goes like this. In this tutorial, we've been using hexadecimal quite a bit, which works by converting our normal decimal numbers into base-16, which is more easily understood by computers. For example, the decimal number 100 would become 0x64. This is good for computers, but not very accessible to

humans, and certainly not to the general audience who are going to play your games. The main purpose of BCD is to convert a hexadecimal number back into a pseudo-decimal number to print out for the user, such as for your points or high scores. So while Chip-8 might store 0x64 internally, fetching its BCD would give us `0x1, 0x0, 0x0`, which we could print to the screen as "100". You'll notice that we've gone from one byte to three in order to store all three digits of our number, which is why we are going to store the BCD into RAM, beginning at the address currently in the *I Register* and moving along. Given that VX stores 8-bit numbers, which range from 0 to 255, we are always going to end up with three bytes, even if some are zero.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // BCD
    (0xF, _, 3, 3) => {
        let x = digit2 as usize;
        let vx = self.v_reg[x] as f32;

        // Fetch the hundreds digit by dividing by 100 and tossing the decimal
        let hundreds = (vx / 100.0).floor() as u8;
        // Fetch the tens digit by dividing by 10, tossing the ones digit and the decimal
        let tens = ((vx / 10.0) % 10.0).floor() as u8;
        // Fetch the ones digit by tossing the hundreds and the tens
        let ones = (vx % 10.0) as u8;

        self.ram[self.i_reg as usize] = hundreds;
        self.ram[(self.i_reg + 1) as usize] = tens;
        self.ram[(self.i_reg + 2) as usize] = ones;
    },

    // -- Unchanged code omitted --
}
```

A side note on efficiency here. For this implementation, I converted our VX value first into a `float`, so that I could use division and modulo arithmetic to get each decimal digit. This is not the fastest implementation nor is it probably the shortest. However, it is one of the easiest to understand. I'm sure there are some highly binary-savvy readers who are disgusted that I did it this way, but this solution is not for them. This is for readers who have never seen BCD before where losing some speed for greater understanding is a better trade-off. However, once you have this implemented, I would encourage everyone to go out and look up more efficient BCD algorithms to add a bit of easily optimization into your code.

5.2.32 FX55 - Store V0 - VX into I

We're on the home stretch! These final two instructions populate our *V Registers* V0 thru the specified VX (inclusive) with the same range of values from RAM, beginning with the address in the *I Register*. This first one stores the values into RAM, while the next one will load them the opposite way.

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // STORE V0 - VX
    (0xF, _, 5, 5) => {
        let x = digit2 as usize;
        let i = self.i_reg as usize;
        for idx in 0..=x {
            self.ram[i + idx] = self.v_reg[idx];
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.33 FX65 - Load I into V0 - VX

```
match (digit1, digit2, digit3, digit4) {
    // -- Unchanged code omitted --

    // LOAD V0 - VX
    (0xF, _, 6, 5) => {
        let x = digit2 as usize;
        let i = self.i_reg as usize;
        for idx in 0..=x {
            self.v_reg[idx] = self.ram[i + idx];
        }
    },

    // -- Unchanged code omitted --
}
```

5.2.34 Final Thoughts

That's it! With this, we now have a fully implemented Chip-8 CPU. You may have noticed through this journey that there are a lot of possible opcode values that are never covered, particularly in the 0x0000, 0xE000, and 0xF000 ranges. This is okay. These opcodes are left as undefined by the original design, and thus if any game attempts to use them it will lead to a `panic!`. If you are still curious following the completion of this emulator, there are a number of Chip-8 extensions which do fill in some of these gaps to add additional functionality, but they will not be covered by this guide.

6 Writing our Desktop Frontend

In this section, we will finally connect all the pieces and get our emulator to load and run a game. At this stage, our emulator core is capable of parsing and processing the game's opcodes, and being able to update the screen, RAM, and the registers as needed. Next, we will need to add some public functions to expose some functionality to the frontend, such as loading in a game, accepting user input, and sharing the screen buffer to be displayed.

6.1 Exposing the Core to the Frontend

We are not done with our `chip8_core` just yet. We need to add a few public functions to our `Emu` struct to give access to some of its items.

In `chip8_core/src/lib.rs`, add the following method to our `Emu` struct:

```
impl Emu {
    // -- Unchanged code omitted --

    pub fn get_display(&self) -> &[bool] {
        &self.screen
    }

    // -- Unchanged code omitted --
}
```

This simply passes a pointer to our screen buffer array up to the frontend, where it can be used to render to the display.

Next, we will need to handle key presses. We already have a `keys` array, but it never actually gets written to. Our frontend will handle actually reading keyboard presses, but we'll need to expose a function that allows it to interface and set elements in our `keys` array.

```
impl Emu {
    // -- Unchanged code omitted --

    pub fn keypress(&mut self, idx: usize, pressed: bool) {
        self.keys[idx] = pressed;
    }

    // -- Unchanged code omitted --
}
```

This function is rather straightforward. It takes the index of the key that has been pressed and sets the value. We could have split this function into a `press_key` and `release_key` function, but this is simple enough that I think the intention still comes across. Keep in mind that `idx` needs to be under 16 or else the program will panic. You can add that restriction here, but instead we'll handle it in the frontend and assume that it's been done correctly in the backend, rather than checking it twice.

Lastly, we need some way to load the game code from a file into our RAM so it can be executed. We'll dive into this more deeply when we begin reading from a file in our frontend, but for now we need to take in a list of bytes and copy them into our RAM.

```
impl Emu {
    // -- Unchanged code omitted --

    pub fn load(&mut self, data: &[u8]) {
        let start = START_ADDR as usize;
        let end = (START_ADDR as usize) + data.len();
        self.ram[start..end].copy_from_slice(data);
    }
```

```
    // -- Unchanged code omitted --
}
```

This function copies all the values from our input `data` slice into RAM beginning at 0x200. Remember that the first 512 bytes of RAM aren't to contain game data, and are empty except for the character sprite data we store there.

6.2 Frontend Setup

Finally, let's setup the frontend of the emulator so we can test things out and (hopefully) play some games! Making a fancy GUI interface is beyond the scope of this guide, we will simply start the emulator and choose which game to play via a command line argument. Let's set that up now.

In `desktop/src/main.rs` we will need to read the command line arguments to receive the path to our game ROM file. We could create several flags for additional configuration, but we'll keep it simple and say that we'll require exactly one argument - the path to the game. Any other number and we'll exit out with an error.

```
use std::env;

fn main() {
    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: cargo run path/to/game");
        return;
    }
}
```

This grabs all of the passed command line parameters into a vector, and if there isn't two (the name of the program is always stored in `args[0]`), then we print out the correct input and exit. The path passed in by the user is now stored in `args[1]`. We'll have to make sure that's a valid file once we attempt to open it, but first, we have some other stuff to setup.

6.3 Creating a Window

For this emulation project, we are going to use the SDL library to create our game window and render to it. SDL is an excellent drawing library to use with good support for key presses and drawing. There's a small amount of boilerplate we'll need in order to set it up, but once that's done we can begin our emulation.

First, we'll need to include it into our project. Open `desktop/Cargo.toml` and add `sdl2` to our dependencies:

```
[dependencies]
chip8_core = { path = "../chip8_core" }
sdl2 = "^0.34.3
```

Now back in `desktop/src/main.rs`, we can begin bringing it all together. We'll need the public functions we defined in our core, so let's tell Rust we'll need those with `use chip8_core::*`.

```
use chip8_core::*
use std::env;

fn main() {
    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: cargo run path/to/game");
        return;
    }
}
```

Inside `chip8_core`, we created public constants to hold the screen size, which we are now importing. However, a 64x32 game window is really small on today's monitors, so let's go ahead and scale it up a bit. After experimenting with some numbers, a 15x scale works well on my monitor, but you can tweak this if you prefer something else.

```
const SCALE: u32 = 15;
const WINDOW_WIDTH: u32 = (SCREEN_WIDTH as u32) * SCALE;
const WINDOW_HEIGHT: u32 = (SCREEN_HEIGHT as u32) * SCALE;
```

Recall that `SCREEN_WIDTH` and `SCREEN_HEIGHT` were public constants we defined in our backend and are now included into this crate via the `use chip8_core::*` statement. SDL will require screen sizes to be `u32` rather than `usize` so we'll cast them here.

It's time to create our SDL window! The following code simply creates a new SDL context, then makes the window itself and the canvas that we'll draw upon.

```
fn main() {
    // -- Unchanged code omitted --

    // Setup SDL
    let sdl_context = sdl2::init().unwrap();
    let video_subsystem = sdl_context.video().unwrap();
    let window = video_subsystem
        .window("Chip-8 Emulator", WINDOW_WIDTH, WINDOW_HEIGHT)
        .position_centered()
        .opengl()
        .build()
        .unwrap();

    let mut canvas = window.into_canvas().present_vsync().build().unwrap();
    canvas.clear();
    canvas.present();
}
```

This is a lot to take in, but the gist is this. We'll initialize SDL and tell it to create a new window of our scaled up size. We'll also have it be created in the middle of the user's screen. We'll then get the canvas object we'll actually draw to, with VSYNC on. Then go ahead and clear it and show it to the user.

If you attempt to run it now (give it a dummy file name to test, like `cargo run test`), you'll see a window pop up for a brief moment before closing. This is because the SDL window is created briefly, but then the program ends and the window closes. We'll need to create our main game loop so that our program doesn't end immediately, and while we're add it, let's add some handling to quit the program if we try to exit out of the window (otherwise you'll have to force quit the program from your task manager).

SDL uses something called an *event pump* to poll for events every loop. By checking this, we can cause different things to happen for given events, such as attempting to close the window or pressing a key. For now, we'll just have the program break out of the main game loop if it needs the window to close.

We'll need to tell Rust that we wish to use SDL's `Event`:

```
use sdl2::event::Event;
```

And modify our `main` function to add our basic game loop:

```
fn main() {
    // -- Unchanged code omitted --

    // Setup SDL
    let sdl_context = sdl2::init().unwrap();
    let video_subsystem = sdl_context.video().unwrap();
    let window = video_subsystem
        .window("Chip-8 Emulator", WINDOW_WIDTH, WINDOW_HEIGHT)
        .position_centered()
        .opengl()
        .build()
        .unwrap();
```

```
    let mut canvas = window.into_canvas().present_vsync().build().unwrap();
    canvas.clear();
    canvas.present();

    let mut event_pump = sdl_context.event_pump().unwrap();

    'gameloop: loop {
        for evt in event_pump.poll_iter() {
            match evt {
                Event::Quit{..} => {
                    break 'gameloop;
                },
                _ => ()
            }
        }
    }
}
```

This addition sets up our main game loop, which checks if any events have been triggered. If the `Quit` event is detected (by attempting to close the window), then the program breaks out of the loop, causing it to end. If you try to run it again via `cargo run test`, you'll see a new black window pop-up, with the title of 'Chip-8 Emulator'. The window should successfully close without issue.

It lives! Next up, we'll initialize our emulator's `chip8_core` backend, and open and load the game file.

6.4 Loading a File

Our frontend can now create a new emulation window, so it's time to start getting the backend up and running as well. Our next step will be to actually read in a game file, and pass its data to the backend to be stored in RAM and executed upon. Firstly, we need to actually have a backend object to pass things to. Still in `frontend/src/main.rs`, let's create our emulation object. Our `Emu` object has already been included with all the other public items from `chip8_core`, so we're free to initialize.

```
fn main() {
    // -- Unchanged code omitted --

    let mut chip8 = Emu::new();

    'gameloop: loop {
        // -- Unchanged code omitted --
    }
}
```

Creating the `Emu` object needs to go somewhere prior to our main game loop, as that is where the emulation drawing and key press handing will go. Remember that the path to the game is being passed in by the user, so we'll also need to make sure this file actually exists before we attempt read it in. We'll need to first `use` a few items from the standard library in `main.rs` to open a file.

```
use std::fs::File;
use std::io::Read;
```

Pretty self-explanatory there. Next, we'll open the file given to us as a command line parameter, read it into a buffer, then pass that data into our emulator backend.

```
fn main() {
    // -- Unchanged code omitted --
    let mut chip8 = Emu::new();

    let mut rom = File::open(&args[1]).expect("Unable to open file");
    let mut buffer = Vec::new();
```

```
    rom.read_to_end(&mut buffer).unwrap();
    chip8.load(&buffer);
    // -- Unchanged code omitted --
}
```

A few things to note here. In the event that Rust is unable to open the file from the path the user gave us (likely because it doesn't exist), then the `expect` condition will fail and the program will exit with that message. Secondly, you may be asking why don't we simply give the file path to the backend and load the data in there? Opening and reading a file is in the standard library after all. This is for two reasons. Firstly, reading a file is a more frontend-type behavior and it better fits here. Secondly, and more importantly, our eventual plan is to make this emulator work in a web browser with little to no changes to our backend. How a browser reads a file is very different to how your file system will do it, so we will allow the frontends to handle the reading, and pass in the data once we have it.

6.5 Running the Emulator and Drawing to the Screen

Here's the big moment! The game has been loaded into RAM, our main loop is running, now we just need to tell our backend to begin processing its instructions, and to actually draw to the screen. If you recall, the emulator runs through a clock cycle each time its `tick` function is called, so let's add that to our loop.

```
fn main() {
    // -- Unchanged code omitted --

    'gameloop: loop {
        for event in event_pump.poll_iter() {
            // -- Unchanged code omitted --
        }

        chip8.tick();
    }
}
```

Now every time the loop cycles, the emulator will progress through another instruction. This may seem too easy, but we've set it up so `tick` moves all the pieces of the backend, including modifying our screen buffer. Let's add a function that will grab the screen data from the backend and update our SDL window. First, we need to `use` a few additional elements from SDL:

```
use sdl2::pixels::Color;
use sdl2::rect::Rect;
use sdl2::render::Canvas;
use sdl2::video::Window;
```

Next, the function, which will take in a reference to our `Emu` object, as well as a mutable reference to our SDL canvas. Drawing the screen requires a few steps. First, we clear the canvas to erase the previous frame. Then, we iterate through the screen buffer, drawing a white rectangle anytime the given value is true. Since Chip-8 only supports black and white, if we clear the screen as black, we only have to worry about drawing the white squares.

```
fn draw_screen(emu: &Emu, canvas: &mut Canvas<Window>) {
    // Clear canvas as black
    canvas.set_draw_color(Color::RGB(0, 0, 0));
    canvas.clear();

    let screen_buf = emu.get_display();
    // Now set draw color to white, iterate through each point and see if it should be drawn
    canvas.set_draw_color(Color::RGB(255, 255, 255));
    for (i, pixel) in screen_buf.iter().enumerate() {
        if *pixel {
            // Convert our 1D array's index into a 2D (x,y) position
            let x = (i % SCREEN_WIDTH) as u32;
            let y = (i / SCREEN_WIDTH) as u32;
```

```
            // Draw a rectangle at (x,y), scaled up by our SCALE value
            let rect = Rect::new((x * SCALE) as i32, (y * SCALE) as i32, SCALE, SCALE);
            canvas.fill_rect(rect).unwrap();
        }
    }
    canvas.present();
}
```

To summarize this function, we get our 1D screen buffer array and iterate across it. If we find a white pixel (a true value), then we calculate the 2D (x, y) of the screen and draw a rectangle there, scaled up by our `SCALE` factor.

We'll call this function in our main loop, just after we `tick`.

```
fn main() {
    // -- Unchanged code omitted --

    'gameloop: loop {
        for event in event_pump.poll_iter() {
            // -- Unchanged code omitted --
        }

        chip8.tick();
        draw_screen(&chip8, &mut canvas);
    }
}
```

Some of you might be raising your eyebrows at this. The screen should be updated at 60 frames per second, or 60 Hz. Surely the emulation needs to happen faster than that? It does, but hold that thought for now. We'll start by making sure it works at all before we fix the timings.

If you have a Chip-8 game downloaded, go ahead and try running your emulator with an actual game via:

```
$ cargo run path/to/game
```

If everything has gone well, you should see the window appear and the game begin to render and play! This is a tremendous step, and you should feel accomplished for getting this far with your very own emulator.

As I mentioned previously, the emulation `tick` speed should probably run faster than the canvas refresh rate. If you watch your game run, it might feel a bit sluggish. Right now, we execute one instruction, then draw to the screen, then repeat. As you're aware now, it takes several instructions to be able to do any meaningful changes to the screen. To get around this, we will allow the emulator to tick several times before redrawing.

Now, this is where things get a bit... experimental. The Chip-8 specification says nothing about how quickly the system should actually run. Even leaving it is now so it runs at 60 Hz is a valid solution (and you're welcome to do so). We'll simply allow our `tick` function to loop several times before moving on to drawing the screen. Personally, I (and other emulators I've looked at) find that 10 times is a nice sweet spot.

```
const TICKS_PER_FRAME: usize = 10;
// -- Unchanged code omitted --

fn main() {
    // -- Unchanged code omitted --

    'gameloop: loop {
        for event in event_pump.poll_iter() {
            // -- Unchanged code omitted --
        }

        for _ in 0..TICKS_PER_FRAME {
            chip8.tick();
        }
        draw_screen(&chip8, &mut canvas);
```

```
    }
}
```

Some of you might feel this is a bit hackish, and to be honest I somewhat agree with you. However, this is also how more 'sophisticated' systems work, with the exception that those CPUs usually have some way of notifying the screen that it's ready to redraw. Since the Chip-8 has no such mechanism nor any defined clock speed, this is a easier way to accomplish this task.

If you run again, you might notice that it doesn't get very far before pausing. This is likely due to the fact that we never update our two timers, so the emulator has no concept of how long time has passed for its games. I mentioned earlier that the timers run once per frame, rather than at the clock speed, so we can modify the timers at the same point as when we modify the screen.

```
fn main() {
    // -- Unchanged code omitted --

    'gameloop: loop {
        for event in event_pump.poll_iter() {
            // -- Unchanged code omitted --
        }

        for _ in 0..TICKS_PER_FRAME {
            chip8.tick();
        }
        chip8.tick_timers();
        draw_screen(&chip8, &mut canvas);
    }
}
```

There's still a few things left to implement (you can't actually control your game for one) but it's a great start, and we're on the final stretches now!

6.6 Adding User Input

We can finally render our Chip-8 game to the screen, but we can't get very far into playing it as we have no way to control it. Fortunately, SDL supports reading in inputs to the keyboard which we can translate and send to our backend emulation.

As a refresher, the Chip-8 system supports 16 different keys. These are typically organized in a 4x4 grid, with keys 0-9 organized like a telephone with keys A-F surrounding. While you are welcome to organize the keys in any configuration you like, some game devs assumed they're in the grid pattern when choosing their games inputs, which means it can be awkward to play some games otherwise. For our emulator, we'll use the left-hand keys of the QWERTY keyboard as our inputs, as shown below.

Let's create a helper function to convert SDL's key type into the values that we will send to the emulator. We'll need to bring SDL keyboard support into `main.rs` via:

```
use sdl2::keyboard::Keycode;
```

Now, we'll create a new function that will take in a `Keycode` and output an optional `u8` value. There are only 16 valid keys, so we'll wrap a valid output valid in `Some`, and return `None` if the user presses a non-Chip-8 key. This function is then just pattern matching all of the valid keys as outlined in the image above.

```
fn key2btn(key: Keycode) -> Option<usize> {
    match key {
        Keycode::Num1 =>    Some(0x1),
        Keycode::Num2 =>    Some(0x2),
        Keycode::Num3 =>    Some(0x3),
        Keycode::Num4 =>    Some(0xC),
        Keycode::Q =>       Some(0x4),
        Keycode::W =>       Some(0x5),
        Keycode::E =>       Some(0x6),
```

```
        Keycode::R =>       Some(0xD),
        Keycode::A =>       Some(0x7),
        Keycode::S =>       Some(0x8),
        Keycode::D =>       Some(0x9),
        Keycode::F =>       Some(0xE),
        Keycode::Z =>       Some(0xA),
        Keycode::X =>       Some(0x0),
        Keycode::C =>       Some(0xB),
        Keycode::V =>       Some(0xF),
        _ =>                None,
    }
}
```

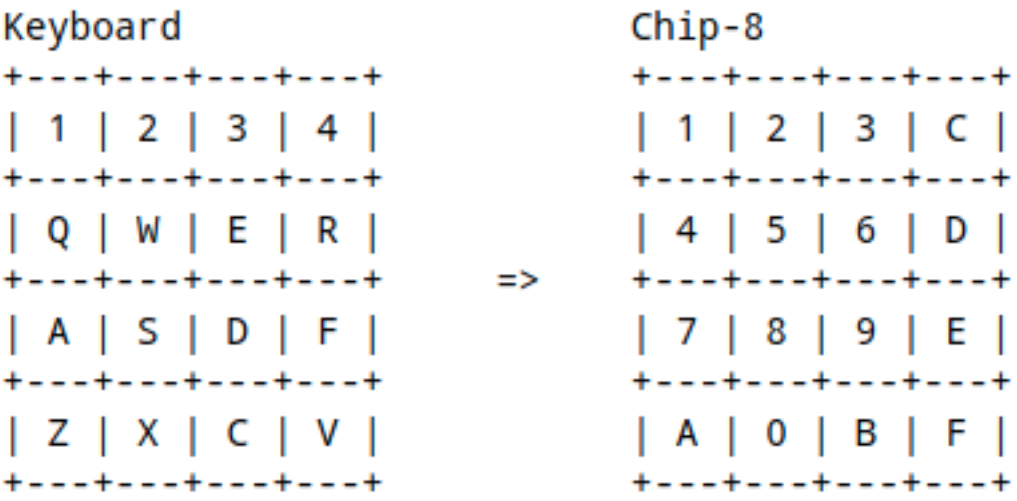

Figure 5: Keyboard to Chip-8 key layout

Next, we'll add two additional events to our main event loop, one for `KeyDown` and the other for `KeyUp`. Each will check if the pressed key gives a `Some` value from our `key2btn` function, and if so pass it to the emulator via the public `keypress` function we defined earlier. The only difference between the two will be if it sets or clears.

```
fn main() {
    // -- Unchanged code omitted --
    'gameloop: loop {
        for evt in event_pump.poll_iter() {
            match evt {
                Event::Quit{..} => {
                    break 'gameloop;
                },
                Event::KeyDown{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, true);
                    }
                },
                Event::KeyUp{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, false);
                    }
                },
                _ => ()
            }
        }
```

```
        chip8.tick();
        draw_screen(&chip8, &mut canvas);
    }
    // -- Unchanged code omitted --
}
```

If you haven't seen it before, the `if let` statement is only satisfied if the value on the right matches that on the left, namely that `key2btn(key)` returns a `Some` value. The unwrapped value is then stored in `k`.

Let's also add a common emulator ability - quitting the program by pressing Escape. We'll add that alongside our `Quit` event.

```
fn main() {
    // -- Unchanged code omitted --
    'gameloop: loop {
        for evt in event_pump.poll_iter() {
            match evt {
                Event::Quit{..} | Event::KeyDown{keycode: Some(Keycode::Escape), ..}=> {
                    break 'gameloop;
                },
                Event::KeyDown{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, true);
                    }
                },
                Event::KeyUp{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, false);
                    }
                },
                _ => ()
            }
        }

        chip8.tick();
        draw_screen(&chip8, &mut canvas);
    }
    // -- Unchanged code omitted --
}
```

Unlike the other key events, where we would check the found `key` variable, we are simply looking for the Escape key to quit. If you don't want this ability in your emulator, or would like some other key press functionality, you're welcome to do so.

That's it! The desktop frontend of our Chip-8 emulator is now complete. We can specify a game via a command line parameter, load and execute it, display the output to the screen, and handle user input. I hope you were able to get an understanding of how emulation works. Chip-8 is a rather basic system, but the techniques discussed here form the basis for how all emulation works.

However, this guide isn't done! In the next section I will discuss how to build our emulator with WebAssembly and getting it to run in a web browser.

7 Introduction to WebAssembly

This section will discuss how to take our finished emulator and configure it to run in a web browser via a relatively new technology called *WebAssembly*. I encourage you to read more about WebAssembly, as it is an interesting and rapidly growing technology. A brief gist is that it is a format for compiling programs into a binary executable, similar in scope to an .exe, but are meant to be run within a web browser. It is supported by all of the major web browsers, and is a cross-company standard being developed between them. This means that instead of having to write web code in JavaScript or other web-centric languages, you can write it in any language that supports compilation of .wasm files and still be able to run in a browser. At the time of writing, C, C++, and Rust are the major languages which support it, fortunately for us.

7.1 Setting Up

While we could cross-compile to the WebAssembly Rust targets ourselves, a useful set of tools called wasm-pack has been developed to allow us to easily compile to WebAssembly without manually adding the appropriate targets and dependencies. You will need to install it via:

```
$ cargo install wasm-pack
```

I also mentioned that we will need to make a slight adjustment to our `chip8_core` module to allow it to compile correctly to the `wasm` target. Rust uses a system called `wasm-bindgen` to create hooks that will work with WebAssembly. All of the `std` code we use is already fine, however we also use the `rand` crate in our backend, and it is not currently set to work correctly. Fortunately, it does support the functionality, we just need to enable it. In `chip8_core/Cargo.toml` we need to change

```
[dependencies]
rand = "^0.7.3"
```

to

```
[dependencies]
rand = { version = "^0.7.3", features = ["wasm-bindgen"] }
```

All this does is specifies that we will require `rand` to include the `wasm-bindgen` feature upon compilation, which will allow it to work correctly in our WebAssembly binary.

Note: In the time between writing this tutorial's code and finishing the write-up, the `rand` crate updated to version 0.8. Among other changes is that the `wasm-bindgen` feature has been removed. If you are wanting to use the most up-to-date `rand` crate, it appears that WebAssembly support has been moved out into its own separate crate. Since we are only using the most basic random function, I didn't feel the need to upgrade to 0.8, but if you wish to, it appears that additional integration would be required.

That's the last time you will need to edit your `chip8_core` module, everything else will be done in our new frontend. Let's set that up now. First, lets crate another Rust module via:

```
$ cargo init wasm --lib
```

This command may look familiar, it will create another new Rust library called `wasm`. Just like `desktop`, we will need to edit `wasm/Cargo.toml` to point to where `chip8_core` is located.

```
[dependencies]
chip8_core = { path = "../chip8_core" }
```

Now, a big difference between our `desktop` and our new `wasm` is that `desktop` was an executable project, it had a `main.rs` that we would compile and run. `wasm` will not have that, it is meant to be compiled into a .wasm file that we will load into a webpage. It is the webpage that will serve that the frontend, so let's add some basic HTML boilerplate, just to get us started. Create a new folder called `web` to hold the webpage specific code, and then create `web/index.html` and add basic HTML boilerplate.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Chip-8 Emulator</title>
        <meta charset="utf-8">
```

```
    </head>
    <body>
        <h1>My Chip-8 Emulator</h1>
    </body>
</html>
```

We'll add more to it later, but for now this will suffice. However, our web program will not run if you simply open the file in a web browser, you will need to start a web server first. If you have Python 3 installed, which all modern Macs and many Linux distributions do, you can simply start a web server via:

```
$ python3 -m http.server
```

Then navigate to `localhost` in your web browser. If you ran this in the `web` directory, you should see the our `index.html` page displayed. Now, I've tried to find a simple, built-in way to start a local web server on Windows, and I haven't really found one. I personally use Python 3, but you are welcome to use any other similar service, such as `npm` or even some Visual Studio Code extensions. It doesn't matter which, just so they can host a local web page.

7.2 Defining our WebAssembly API

Here are the broad steps we are going to take to create our browser emulator. We have our `chip8_core` created already, but we are now missing all of the functionality we added to `desktop`. Loading a file, handling key presses, telling it when to tick, etc. On the other hand, we have a web page that (will) run JavaScript, which needs to handle inputs from the user and display items. Our `wasm` crate is what goes in the middle. It will take inputs from JavaScript and convert them into the data types required by our `chip8_core`.

Most importantly, we also need to somehow create a `chip8_core::Emu` object and keep it in scope for the entirety of our web page. This is also where that happens.

To begin, let's include a few external crates that we will need to allow Rust to interface with JavaScript. Open up `wasm/Cargo.toml` and add the following dependencies:

```
[dependencies]
chip8_core = { path = "../chip8_core" }
js-sys = "^0.3.46"
wasm-bindgen = "^0.2.69"

[dependencies.web-sys]
version = "^0.3.46"
features = []
```

You'll notice that we're handling `web-sys` differently than other dependencies. That crate is structured in such a way that instead of getting everything it contains simply by including it in our `Cargo.toml`, we also need to specify additional "features" which come with the crate, but aren't available by default. Keep this file open, as we'll be adding to the `web_sys` features soon enough.

Since this crate is going to be interfacing with another language, we need to specify how they are to communicate. Without getting too deep into the details, Rust can use the C language's ABI to easily communicate with other languages that support it, and it will greatly simplify our wasm binary to do so. So, we will need to tell `cargo` to use it. Add this in `wasm/Cargo.toml` as well:

```
[lib]
crate-type = ["cdylib"]
```

Excellent. Now to `wasm/src/lib.rs`. Let's create a struct that will house our `Emu` object as well as all the frontend functions we need to interface with JavaScript and operate. We'll also need to include all of our public items from `chip8_core` as well.

```
use chip8_core::*;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct EmuWasm {
```

```
    chip8: Emu,
}
```

Note the `#[wasm_bindgen]` tag, which tells the compiler that this struct needs to be configured for WebAssembly. Any function or struct that is going to be called from within JavaScript will need to have it. Let's also define the constructor.

```
use chip8_core::*;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct EmuWasm {
    chip8: Emu,
}

#[wasm_bindgen]
impl EmuWasm {
    #[wasm_bindgen(constructor)]
    pub fn new() -> EmuWasm {
        EmuWasm {
            chip8: Emu::new(),
        }
    }
}
```

Pretty straight-forward. The biggest thing to note is that the `new` method requires the special `constructor` inclusion so the compiler knows what we're trying to do.

Now we have a struct containing our `chip8` emulation core object. Here, we will implement the same methods that we needed by our `desktop` frontend, such as passing key presses/releases to the core, loading in a file, and ticking. Let's begin with ticking the CPU and the timers, as it's the easiest.

```
#[wasm_bindgen]
impl EmuWasm {
    // -- Unchanged code omitted --

    #[wasm_bindgen]
    pub fn tick(&mut self) {
        self.chip8.tick();
    }

    #[wasm_bindgen]
    pub fn tick_timers(&mut self) {
        self.chip8.tick_timers();
    }
}
```

That's it, these are just thin wrappers to call the corresponding functions in the `chip8_core`. These functions don't take any input, so there's nothing fancy for them to do except, well, tick.

Remember the `reset` function we created in the `chip8_core`, but then never used? Well, we'll get to use it now. This will be a wrapper just like the previous two functions.

```
#[wasm_bindgen]
pub fn reset(&mut self) {
    self.chip8.reset();
}
```

Key pressing is the first of these functions that will deviate from what was done in `desktop`. This works in a similar fashion to what we did for `desktop`, but rather than taking in an SDL key press, we'll need to accept one from JavaScript. I promised we'd add some `web-sys` features, so lets do so now. Back in `wasm/Cargo.toml` add the

`KeyboardEvent` feature

```
[dependencies.web-sys]
version = "^0.3.46"
features = [
    KeyboardEvent
]
```

```
use web_sys::KeyboardEvent;

// -- Unchanged code omitted --

impl EmuWasm {
    // -- Unchanged code omitted --

    #[wasm_bindgen]
    pub fn keypress(&mut self, evt: KeyboardEvent, pressed: bool) {
        let key = evt.key();
        if let Some(k) = key2btn(&key) {
            self.chip8.keypress(k, pressed);
        }
    }
}

fn key2btn(key: &str) -> Option<usize> {
    match key {
        "1" => Some(0x1),
        "2" => Some(0x2),
        "3" => Some(0x3),
        "4" => Some(0xC),
        "q" => Some(0x4),
        "w" => Some(0x5),
        "e" => Some(0x6),
        "r" => Some(0xD),
        "a" => Some(0x7),
        "s" => Some(0x8),
        "d" => Some(0x9),
        "f" => Some(0xE),
        "z" => Some(0xA),
        "x" => Some(0x0),
        "c" => Some(0xB),
        "v" => Some(0xF),
        _ =>   None,
    }
}
```

This is very similar to our implementation for our `desktop`, except we are going to take in a JavaScript `KeyboardEvent`, which will result in a string for us to parse. Note that the key strings are case sensitive, so keep everything lowercase unless you want your players to hold down shift a lot.

A similar story awaits us when we load a game, it will follow a similar style, except we will need to receive and handle a JavaScript object.

```
use js_sys::Uint8Array;
// -- Unchanged code omitted --

impl EmuWasm {
    // -- Unchanged code omitted --
```

```
    #[wasm_bindgen]
    pub fn load_game(&mut self, data: Uint8Array) {
        self.chip8.load(&data.to_vec());
    }
}
```

The only thing remaining is our function to actually render to a screen. I'm going to create an empty function here, but we'll hold off on implementing it for now, instead we'll turn our attention back to our web page, and begin working from the other direction.

```
impl EmuWasm {
    // -- Unchanged code omitted --

    #[wasm_bindgen]
    pub fn draw_screen(&mut self, scale: usize) {
        // TODO
    }
}
```

We'll come back here once we get our JavaScript setup and we know exactly how we're going to draw.

7.3 Creating our Frontend Functionality

Time to get our hands dirty in JavaScript. First, let's add some additional elements to our very bland web page. When we created the emulator to run on a PC, we used SDL to create a window to draw upon. For a web page, we will use an element HTML5 gives us called a *canvas*. We'll also go ahead and point our web page to our (currently non-existent) JS script.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Chip-8 Emulator</title>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>My Chip-8 Emulator</h1>
        <label for="fileinput">Upload a Chip-8 game: </label>
        <input type="file" id="fileinput" autocomplete="off"/>
        <canvas id="canvas">If you see this message, then your browser doesn't support HTML5</canvas>
    </body>
    <script type="module" src="index.js"></script>
</html>
```

We added three things here, first a button which when clicked will allow the users to select a Chip-8 game to run. Secondly, the `canvas` element, which includes a brief message for any unfortunate users with an out of date browser. Finally we told our web page to also load the `index.js` script we are about to create. Note that at the time of writing, in order to load a .wasm file via JavaScript, you need to specify that it is of `module` type.

Now, let's create `index.js` and we'll define some items we'll need. First, we need to tell JavaScript to load in our WebAssembly functions. Now, we aren't going to load it in directly here. When we compile with `wasm-pack`, it will generate not only our .wasm file, but also an auto-generated JavaScript "glue" that will wrap each function we defined around a JavaScript function we then can use here.

```
import init, * as wasm from "./wasm.js"
```

This imports all of our functions, as well as a special `init` function that will need to be called first before we can use anything from `wasm`.

Let's define some constants and do some basic setup now.

```
import init, * as wasm from "./wasm.js"
```

```
const WIDTH = 64
const HEIGHT = 32
const SCALE = 15
const TICKS_PER_FRAME = 10
let anim_frame = 0

const canvas = document.getElementById("canvas")
canvas.width = WIDTH * SCALE
canvas.height = HEIGHT * SCALE

const ctx = canvas.getContext("2d")
ctx.fillStyle = "black"
ctx.fillRect(0, 0, WIDTH * SCALE, HEIGHT * SCALE)

const input = document.getElementById("fileinput")
```

All of this will look familiar from our `desktop` build. We fetch the HTML canvas and adjust its size to the dimension of our Chip-8 screen, plus scaled up a bit (feel free to adjust this for your preferences).

Now to the meat of it! Let's create a main `run` function that will load our `EmuWasm` object and handle the main emulation.

```
async function run() {
    await init()
    let chip8 = new wasm.EmuWasm()

    document.addEventListener("keydown", function(evt) {
        chip8.keypress(evt, true)
    })

    document.addEventListener("keyup", function(evt) {
        chip8.keypress(evt, false)
    })

    input.addEventListener("click", function(evt) {
        // Handle file loading
    }, false)
}

run().catch(console.error)
```

Here, we called the mandatory `init` function which tells our browser to initialize our WebAssembly binary before we use it. We then create our emulator backend by making a new `EmuWasm` object.

We will now handle loading in a file when our button is pressed.

```
input.addEventListener("click", function(evt) {
    // Stop previous game from rendering, if one exists
    if (anim_frame != 0) {
        window.cancelAnimationFrame(anim_frame)
    }

    let file = evt.target.files[0]
    if (!file) {
        alert("Failed to read file")
        return
    }

    // Load in game as Uint8Array, send to .wasm, start main loop
    let fr = new FileReader()
```

```
    fr.onload = function(e) {
        let buffer = fr.result
        const rom = new Uint8Array(buffer)
        chip8.reset()
        chip8.load_game(rom)
        mainloop(chip8)
    }
    fr.readAsArrayBuffer(file)
}, false)

function mainloop(chip8) {
}
```

This function adds an event listener to our `input` button which is triggered whenever it is clicked. Our `desktop` frontend used SDL to manage not only drawing to a window, but only to ensure that we were running at 60 FPS. The analogous feature for canvases is the "Animation Frames". Anytime we want to render something to the canvas, we request the window to animate a frame, and it will wait until the correct time has elapsed to ensure 60 FPS performance. We'll see how this works in a moment, but for now, we need to tell our program that if we're loading a new game, we need to stop the previous animation. We'll also reset our emulator before we load in the ROM, to ensure everything is just as it started, without having to reload the webpage.

Following that, we look at the file that the user has pointed us to. We don't do any fancy checking to see if it's actually a Chip-8 program, but we do need to make sure that it is a file of some sort. We then read it in and pass it to our backend via our `EmuWasm` object. Once the game is loaded, we can jump into our main emulation loop!

```
function mainloop(chip8) {
    // Only draw every few ticks
    for (let i = 0; i < TICKS_PER_FRAME; i++) {
        chip8.tick()
    }
    chip8.tick_timers()

    // Clear the canvas before drawing
    ctx.fillStyle = "black"
    ctx.fillRect(0, 0, WIDTH * SCALE, HEIGHT * SCALE)
    // Set the draw color back to white before we render our frame
    ctx.fillStyle = "white"
    chip8.draw_screen(SCALE)

    anim_frame = window.requestAnimationFrame(() => {
        mainloop(chip8)
    })
}
```

This should look very similar to what we did for our `desktop` frontend, minus the SDL specific stuff, so I'll only briefly mention it. We tick several times before clearing the canvas and telling our `EmuWasm` object to draw the current frame to our canvas. Here is where we tell our window that we would like to render a frame, and we also save its ID for if we need to cancel it above. The `requestAnimationFrame` will wait to ensure 60 FPS performance, and then restart our `mainloop` when it is time, beginning the process all over again.

7.4 Compiling our WebAssembly binary

Before we go any further, let's now try and build our Rust code and ensure that it can be loaded by our web page without issue. `wasm-pack` will handle the compilation of our .wasm binary, but we also need to specify that we don't wish to use any web packing systems like `npm`. To build, change directories into the `wasm` folder and run:

```
$ wasm-pack build --target web
```

Once it is completed, the targets will be built into a new `pkg` directory. There are several items in here, but the only ones we need are `wasm_bg.wasm` and `wasm.js`. `wasm_bg.wasm` is the combination of our `wasm` and `chip8_core` Rust

crates compiled into one, and `wasm.js` is the JavaScript "glue" that we included earlier. It is mainly wrappers around the API we defined in `wasm` as well as some initialization code. It is actually quite readable, so it's worth taking a look at what it is doing.

Running the page in a local web server should allow you to pick and load a game without any warnings coming up in the browser's console. However, we haven't written the screen rendering function yet, so let's finish that so we can see our game actually run.

7.5 Drawing to the canvas

Here is the final step, rendering to the screen. We created an empty `draw_screen` function in our `EmuWasm` object, and we call it at the right time, but it currently doesn't do anything. Now, there are two ways we could handle this. We could either pass the frame buffer into JavaScript and render it, or we could obtain our canvas in our `EmuWasm` binary and render to it in Rust. Either method would work fine, but personally I found that handling the rendering in Rust is actually easier personally.

We've used the `web_sys` crate to handle JavaScript `KeyboardEvents` in Rust, but it has the functionality to manage many more JavaScript elements. Again, the ones we wish to use will need to be defined as features in `wasm/Cargo.toml`.

```
[dependencies.web-sys]
version = "^0.3.46"
features = [
    "CanvasRenderingContext2d",
    "Document",
    "Element",
    "HtmlCanvasElement",
    "ImageData",
    "KeyboardEvent",
    "Window"
]
```

Here is an overview of our next steps. In order to render to an HTML5 canvas, you need to obtain the canvas object and its *context* which is the object which gets the draw functions called upon it. Since our WebAssembly binary has been loaded by our webpage, it has access to all of its elements just as a JS script would. We will change our `new` constructor to grab the current window, canvas, and context much like you would in JavaScript.

```
use wasm_bindgen::JsCast;
use web_sys::{CanvasRenderingContext2d, HtmlCanvasElement, KeyboardEvent};

#[wasm_bindgen]
pub struct EmuWasm {
    chip8: Emu,
    ctx: CanvasRenderingContext2d,
}

#[wasm_bindgen]
impl EmuWasm {
    #[wasm_bindgen(constructor)]
    pub fn new() -> Result<EmuWasm, JsValue> {
        let chip8 = Emu::new();

        let document = web_sys::window().unwrap().document().unwrap();
        let canvas = document.get_element_by_id("canvas").unwrap();
        let canvas: HtmlCanvasElement = canvas
            .dyn_into::<HtmlCanvasElement>()
            .map_err(|_| ())
            .unwrap();

        let ctx = canvas.get_context("2d")
            .unwrap().unwrap()
```

```
            .dyn_into::<CanvasRenderingContext2d>()
            .unwrap();

        Ok(EmuWasm{chip8, ctx})
    }

    // -- Unchanged code omitted --
}
```

This should look pretty familiar to those who have done JavaScript programming before. We grab our current window's canvas and get its 2D context, which is saved as a member variable of our `EmuWasm` struct. Now that we have an actual context to render to, we can update our `draw_screen` function to draw to it.

```
#[wasm_bindgen]
pub fn draw_screen(&mut self, scale: usize) {
    let disp = self.chip8.get_display();
    for i in 0..(SCREEN_WIDTH * SCREEN_HEIGHT) {
        if disp[i] {
            let x = i % SCREEN_WIDTH;
            let y = i / SCREEN_WIDTH;
            self.ctx.fill_rect(
                (x * scale) as f64,
                (y * scale) as f64,
                scale as f64,
                scale as f64
            );
        }
    }
}
```

We get the display buffer from our `chip8_core` and iterate through every pixel. If set, we draw it scaled up to the value passed in by our frontend. Don't forget that we already cleared the canvas to black and set the draw color to white before calling `draw_screen`, so it doesn't need to be done here.

That does it! The implementation is done. All that remains is to build it and try it for ourself.

Rebuild by moving into the `wasm` directory and running:

```
$ wasm-pack build --target web
$ mv pkg/wasm_bg.wasm ../web
$ mv pkg/wasm.js ../web
```

Now start your web server and pick a game. If everything has gone well, you should be able to play Chip-8 games just as well in the browser as on the desktop!

8 Opcodes Table

Understanding the Opcode column:

- Any hexadecimal digits (0-9, A-F) that appear in an opcode are interpreted literally, and are used to determine the operation in question.
- The X or Y wild card uses the value stored in VX/VY.
- N refers to a literal hexadecimal value. NN or NNN refer to two or three digit hex numbers respectively.

Example: The instruction 0xD123 would match with the `DXYN` opcode, where VX is V1, VY is V2, and N is 3 (draw a 8x3 sprite at (V1, V2))

Opcode	Description	Notes
0000	Nop	Do nothing, progress to next opcode
00E0	Clear screen	
00EE	Return from subroutine	
1NNN	Jump to address 0xNNN	
2NNN	Call 0xNNN	Enter subroutine at 0xNNN, adding current PC onto stack so we can return here
3XNN	Skip if VX == 0xNN	
4XNN	Skip if VX != 0xNN	
5XY0	Skip if VX == VY	
6XNN	VX = 0xNN	
7XNN	VX += 0xNN	Doesn't affect carry flag
8XY0	VX = VY	
8XY1	VX |= VY	
8XY2	VX &= VY	
8XY3	VX ^= VY	
8XY4	VX += VY	Sets VF if carry
8XY5	VX -= VY	Clears VF if borrow
8XY6	VX »= 1	Store dropped bit in VF
8XY7	VX = VY - VX	Clears VF if borrow
8XYE	VX «= 1	Store dropped bit in VF
9XY0	Skip if VX != VY	
ANNN	I = 0xNNN	
BNNN	Jump to V0 + 0xNNN	
CXNN	VX = rand() & 0xNN	
DXYN	Draw sprite at (VX, VY)	Sprite is 0xN pixels tall, on/off based on value in I, VF set if any pixels flipped
EX9E	Skip if key index in VX is pressed	
EXA1	Skip if key index in VX isn't pressed	
FX07	VX = Delay Timer	
FX0A	Waits for key press, stores index in VX	Blocking operation
FX15	Delay Timer = VX	
FX18	Sound Timer = VX	

Opcode	Description	Notes
FX1E	I += VX	
FX29	Set I to address of font character in VX	
FX33	Stores BCD encoding of VX into I	
FX55	Stores V0 thru VX into RAM address starting at I	Inclusive range
FX65	Fills V0 thru VX with RAM values starting at address in I	Inclusive

9 Changelog

9.1 Version 1.0

- Initial release
- Includes source code for Chip-8 emulator as well as PDF document discussing its development