Introduction

If you really want to know how a particular computer works, there's no better way to learn than by emulating that computer. In this book, we'll be looking at one of the most loved computers of all time, Nintendo's Game Boy. We'll be going through the process from nothing and building the Game Boy up piece by piece. In the process not only will we learn more about the Game Boy itself, but we'll also get a good glimpse into how computers work in general as well as how to build emulators for other computer systems.

What You Need to Know

We'll be assuming only basic programming knowledge and basic knowledge of the terminal. If you've programmed in almost any other language before you should be good to go. The particular language we'll be using is the Rust programming language, but if you've never used Rust, don't worry; we'll be taking things slow, and I'll do my best to link to relevant learning material as things come up. If you want to get a head start on learning the basics of Rust, the amazing Rust book is a great way to learn.

If you've already build emulators before, and are just looking for a reference specific to the Game Boy, you might find this book to be too detailed. I recommend the Pan Docs as a great place to quickly learn the nity-grity of the Game Boy. You'll find even more resources in the resources guide.

Why Rust?

For many emulation projects performance is a key consideration. This means that emulators are often written in low-level languages that allow the programmer to easily write performant code. While our Game Boy emulator could be written in other languages like JavaScript or Python, it's best to use a language that would be appropriate for more resource intensive emulation (e.g. Sony's PlayStation 2 or Nintendo's Wii) so these skills can be used for future emulation projects. Rust fits the bill perfectly here.

While there are other languages that allow the performance needed for emulation like C and C++, Rust has the added bonus of being much more beginner friendly than C or C++. Unlike these older languages Rust has a solid package manager, testing framework and build tool that will feel familiar to users of more modern languages. Rust's community is

also fantastic and generally very helpful.

Finally, Rust also has a great cross platform story - we'll be focusing on web and on desktop, but in the future, we might also be able to bring our emualtor to mobile platforms and embedded devices!

Setup

In order to get started, you'll only need your favorite text editor and Rust related tooling. Follow the instructions on the Rust website for how to install the rustup tool which gives you access to the Rust compiler, the Rust build tool and package manager called Cargo, as well as some other tools that we'll be using later on in our journey.

If you've successfully been able to install Rust you can create a new project by running the following command in your terminal:

```
cargo new emulator
```

Navigate into your project's directory and have a look around. To run your project run the following:

```
cargo run
```

You're all good to go! Let's get emulating!

Architecture Overview

Before we can begin we need to have a basic understanding of what is needed to build a Game Boy emulator. This starts by having a basic understanding of the Game Boy computer architecture. If you're already familiar with the very basics of CPUs, computer memory and I/O devices, you can skip to the next page.

At a High Level

The Gameboy can be thought to contain the following pieces:

CPU (Central Processing Unit)

The CPU is the "brains" of a computer. The CPU is responsible for the following:

- Executing instructions defined by the game the Game Boy is running
- Reading and writting to memory
- Reacting to "events" (known as interrupts) that come from the I/O (input/output) hardware such as the screen and the gamepad controls
- Keeping track of very small pieces of data in "registers" that it manipulates when it runs instructions

RAM (Random Access Memory)

This is the piece of hardware that allows the Game Boy to remember data while it is running. Without the RAM, the Gameboy's CPU could still execute instructions and keep track of small pieces of data in its registers, but if the data no longer fits in its registers, the CPU would have to throw it away.

ROM (Read Only Memory)

This is memory that has been "hardcoded" into the machine (hence it being read only). This memory is used to tell the machine how to set itself up (a.k.a bootstrap) to be able to begin accepting instructions as well as to play the iconic splash screen on boot.

Game cartridges are also known as ROMs as they are mostly read only memory as well.

I/O (Input/Output)

The Gameboy has several pieces of I/O hardware:

- The screen,
- Hardware dedicated to playing sounds,
- A gamepad that the player uses to interact with the game through pushing various buttons.

We'll discuss how the CPU can interface with this hardware much later in the book.

> **Learn More**
>
> If you're interested in learning more about how computers work all the way down to the electrical level, I suggest watching Ben Eater's series on making an 8-bit Breadboard Computer. Ben does an absolutely wonderful job of explaining how

> CPUs, RAM, ROM and even I/O devices are built from the ground up!

Now that we have a small overview of the pieces we'll be talking about, let's take a closer look at the CPU!

CPU

The Game Boy's CPU is a custom chip called the Sharp LR35902. The chip is very similar to the much more popular Intel 8080 and the Zilog Z80. The 8080 was used by lots of different computers in the 70s and 80s including the very first comercially successful personal computer the Altair 8800. The Z80 was also a very popular chip being used in many home electronic devices including in many Sega home consoles such as the Master System and the Sega Genesis/Mega Drive.

We won't go into the specifics of what makes the LR35902 different from the Intel 8080 or Z80, but in general it's good to know that a large portion of what we'll learn about this custom chip will be applicable to other popular chips from yesteryear.

In the next few sections we'll be looking at the different CPU instructions that the LR35902 can execute as long with how it reads instructions from memory, decodes them and updates its internal state as well as the contents of memory and different I/O devices.

CPU Registers

In the previous chapter we outlined some of things that the CPU is responsible for. In this chapter we'll be focusing on just one of them: saving small amounts of data into registers.

Overview

The Game Boy's CPU is a custom chip made just for the Game Boy. The chip is extremely similar to the Intel 8080 which is itself similar to the Zilog Z80. While the Intel 8080 and Zilog Z80 were used in many different computers in the 70s and 80s, the chip inside the Game Boy was just used for the Game Boy. Most of what's true about how the 8080 and Z80 work is also true of the Game Boy's chip. We won't go into details on how exactly they differ, but it's important to be aware that while they're similar to the Game Boy's chip, they're not exactly the same.

Registers

The CPU is composed of 8 different "registers". Registers are responsible for holding on to little pieces of data that the CPU can manipulate when it executes various instructions. The Game Boy's CPU is an 8-Bit CPU, meaning that each of its registers can hold 8 bits (a.k.a 1 byte) of data. The CPU has 8 different registers labled as "a", "b", "c", "d", "e", "f", "h", "l".

Let's get started building our CPU by specing out the registers in code:

```
struct Registers {
  a: u8,
  b: u8,
  c: u8,
  d: u8,
  e: u8,
  f: u8,
  h: u8,
  l: u8,
}
```

We use the type `u8` for our registers. `u8` are 8-bit unsigned integers. For a refresher on how numbers are stored in computers, checkout the guide on numbers.

While the CPU only has 8 bit registers, there are instructions that allow the game to read and write 16 bits (i.e. 2 bytes) at the same time (denoted as `u16` in Rust - a 16 bit unsigned integer). Therefore, we'll need the ability to read an write these "virtual" 16 bit registers. These registers are refered to as "af" ("a" and "f" combined), "bc" ("b" and "c" combined), "de" ("d" and "e" combinded), and finally "hl" ("h" and "l" combined). Let's implement "bc":

```
impl Registers {
  fn get_bc(&self) -> u16 {
    (self.b as u16) << 8
    | self.c as u16
  }

  fn set_bc(&mut self, value: u16) {
    self.b = ((value & 0xFF00) >> 8) as u8;
    self.c = (value & 0xFF) as u8;
  }
}
```

Here we see our first instance of "bit manipulation" through the use of four bitwise operators: ">>", "<<", "&", and "|". If you're unfamiliar with or feel a bit rusty using these types of operators, check out the guide on bit manipulation.

For reading the "bc" register we first treat the "b" register as a `u16` (this effectively just

adds a byte of all 0s to the most significant position of the number). We then shift the "b" register 8 positions so that it's occupying the most significant byte position. Finally, we bitwise OR the "c" register. The result is a two byte number with the contents of "b" in the most significant byte position and the contents of "c" in the least significant byte position.

Flags Register

We're almost done with our registers, but there's one thing we way we can improve our registers for use later. The "f" register is a special register called the "flags" register. The lower four bits of the register are *always* 0s and the CPU automatically writes to the upper four bits when certain things happen. In other words, the CPU "flags" certain states. We won't go into the specific meanings of the flags just yet, but for now just know that they have the following names and positions:

- Bit 7: "zero"
- Bit 6: "subtraction"
- Bit 5: "half carry"
- Bit 4: "carry"

Here's a diagram of the flags register:

```
   ┌-> Carry
 ┌-+> Subtraction
 | |
1111 0000
| |
└-+> Zero
  └-> Half Carry
```

So while we could continue modeling our flags register as a simple 8-bit number (after all, that's all it is in reality), it might be less error prone to explicitly model the fact that the upper 4 bits (a.k.a the upper "nibble") has specific meaning and the lower 4 bits (a.k.a the lower "nibble") must always be zeros.

For this reason we'll make a struct called the `FlagsRegister`:

```
struct FlagsRegister {
    zero: bool,
    subtract: bool,
    half_carry: bool,
    carry: bool
}
```

Since we might need to look at this register as an 8-bit number, we can implement some traits from the standard library that make this easy:

```
const ZERO_FLAG_BYTE_POSITION: u8 = 7;
const SUBTRACT_FLAG_BYTE_POSITION: u8 = 6;
const HALF_CARRY_FLAG_BYTE_POSITION: u8 = 5;
const CARRY_FLAG_BYTE_POSITION: u8 = 4;

impl std::convert::From<FlagsRegister> for u8  {
    fn from(flag: FlagsRegister) -> u8 {
        (if flag.zero       { 1 } else { 0 }) << ZERO_FLAG_BYTE_POSITION |
        (if flag.subtract   { 1 } else { 0 }) << SUBTRACT_FLAG_BYTE_POSITION |
        (if flag.half_carry { 1 } else { 0 }) << HALF_CARRY_FLAG_BYTE_POSITION
|
        (if flag.carry      { 1 } else { 0 }) << CARRY_FLAG_BYTE_POSITION
    }
}

impl std::convert::From<u8> for FlagsRegister {
    fn from(byte: u8) -> Self {
        let zero = ((byte >> ZERO_FLAG_BYTE_POSITION) & 0b1) != 0;
        let subtract = ((byte >> SUBTRACT_FLAG_BYTE_POSITION) & 0b1) != 0;
        let half_carry = ((byte >> HALF_CARRY_FLAG_BYTE_POSITION) & 0b1) != 0;
        let carry = ((byte >> CARRY_FLAG_BYTE_POSITION) & 0b1) != 0;

        FlagsRegister {
            zero,
            subtract,
            half_carry,
            carry
        }
    }
}
```

The `std::convert::From` trait allows us to easily convert our FlagsRegister from a `u8` and back.

Now that we have our special `FlagsRegister`, we can replace the `u8` in our `Registers` struct's `f` field.

And that's it! We have all the functionality we need for our registers. Next we'll be looking at different instructions for manipulating the registers.

Instructions on Register Data

The first instructions we'll be examining are instructions that just operate on and manipulate register data.

ADD

We'll start by looking closely at how the `ADD` instructions work. This is a simple instruction that adds specific register's contents to the A register's contents. Once we know how this instruction works, it won't be much work to extend the CPU to support all other instructions that just operate on register data.

Definition

First we need to define the instruction. We'll get into how the game code actually encodes instructions and where the instructions come from later on. For now we're just focusing on the instruction itself and how it affects the CPU registers.

The first thing to do is to define an enum called `Instruction`. This enum will be the central place where all of our instructions will be defined. Our `ADD` instruction needs to include information on which register they're targeting so we'll make sure to include it by associating the instruction with `ArithmeticTarget` enum that specifies the target register. `ADD` can target all of the 8 bit registers except f.

```
enum Instruction {
  ADD(ArithmeticTarget),
}

enum ArithmeticTarget {
  A, B, C, D, E, H, L,
}
```

Executing the Instruction

Ok, now that we have ths instruction, we'll need a way to execute it. Let's create a method on CPU that takes an instruction and executes it. This method will take a mutable reference to the CPU since instructions always mutate the CPU's state. The method will also take the instruction it will execute. We'll pattern match on the instruction and the target register, and then we'll do the appropriate action according to the instruction and the register:

```
impl CPU {
  fn execute(&mut self, instruction: Instruction) {
    match instruction {
      Instruction::ADD(target) => {
        match target {
          ArithmeticTarget::C => {
            // TODO: implement ADD on register C
          }
          _ => { /* TODO: support more targets */ }
        }
      }
      _ => { /* TODO: support more instructions */ }
    }
  }
}
```

We now have the boiler plate for figuring out which instruction and which target register. Let's see now what we have to do to the actual CPU. The steps for `ADD` for 8 bit target registers are the following:

- Read the current value from the target register
- Add the value to the value in the A register making sure to handle overflow properly
- Update the flags register
- Write the updated value to the A register

Let's implement it with C as the target register:

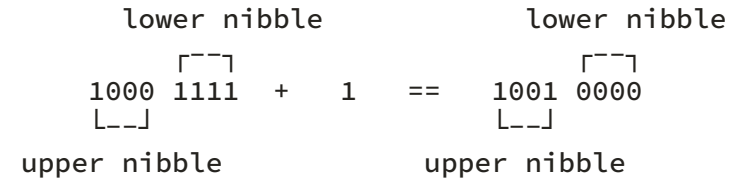
```
impl CPU {
  fn execute(&mut self, instruction: Instruction) {
    match instruction {
      Instruction::ADD(target) => {
        match target {
          ArithmeticTarget::C => {
            let value = self.registers.c;
            let new_value = self.add(value);
            self.registers.a = new_value;
          }
          _ => { /* TODO: support more targets */ }
        }
      }
      _ => { /* TODO: support more instructions */ }
    }
  }

  fn add(&mut self, value: u8) -> u8 {
    let (new_value, did_overflow) = self.registers.a.overflowing_add(value);
    // TODO: set flags
    new_value
  }
}
```

Notice that we use the `overflowing_add` method on our 8 bit value instead of `+`. This is because `+` panics in development when the result of the addition overflows. Rust forces us to be explicit about the behaivor we want, we chose `overflowing_add` because it properly overflows the value, and it informs us if the addition actually resulted in an overflow or not. This will be important information for when we update the flags register.

Setting Flags

There are four flags defined on the flags register:

- Zero: set to true if the result of the operation is equal to 0.
- Subtract: set to true if the operation was a subtraction.
- Carry: set to true if the operation resulted in an overflow.
- Half Carry: set to true if there is an overflow from the lower nibble (a.k.a the lower four bits) to the upper nibble (a.k.a the upper four bits). Let's take a look at some examples of what this means. In the following diagram, we have the byte 143 in binary (0b1000_1111). We then add 0b1 to the number. Notice how the 1 from the lower nibble is carried to the upper nibble. You should already be familiar with carries from elemetry arithmetic. Whenever there's not enough room for a number in a particular digit's place, we carry over to the next digits place.



  If this happens when adding our values, we set the half_carry flag to true. We can test for this by masking out the upper nibble of both the A register and the value we're adding and testing if this value is greater than 0xF.

So let's take a look at the code:

```
impl CPU {
  fn add(&mut self, value: u8) -> u8 {
    let (new_value, did_overflow) = self.registers.a.overflowing_add(value);
    self.registers.f.zero = new_value == 0;
    self.registers.f.subtract = false;
    self.registers.f.carry = did_overflow;
    // Half Carry is set if adding the lower nibbles of the value and register
A
    // together result in a value bigger than 0xF. If the result is larger
than 0xF
    // than the addition caused a carry from the lower nibble to the upper
nibble.
    self.registers.f.half_carry = (self.registers.a & 0xF) + (value & 0xF) >
0xF;
    new_value
  }
}
```

How Do We Know?

Yout might be wondering, "how do we know what to do given a certain the instruction". The short answer is that this is just how the chip was specified and manufactured to worked. We know this because people have either read the original user's manual for the Game Boy's CPU chip (known as a "data sheet"),or they've written test programs for the chip that call specific instructions and see what happens. Luckily you don't need to do this. You can find descriptions of all the instructions in the instruction guide.

> *Side Note*
>
> Most CPU instructions that deal with register data manipulate that data through various bitwise operations. If the likes of logical shifts and bitwise ands aren't super clear to you, check out the guide on bit manipulation.

What are the other types of instructions that act on register data?

- **ADDHL** (add to HL) - just like ADD except that the target is added to the HL register
- **ADC** (add with carry) - just like ADD except that the value of the carry flag is also added to the number
- **SUB** (subtract) - subtract the value stored in a specific register with the value in the A register
- **SBC** (subtract with carry) - just like ADD except that the value of the carry flag is also

subtracted from the number
- **AND** (logical and) - do a bitwise and on the value in a specific register and the value in the A register
- **OR** (logical or) - do a bitwise or on the value in a specific register and the value in the A register
- **XOR** (logical xor) - do a bitwise xor on the value in a specific register and the value in the A register
- **CP** (compare) - just like SUB except the result of the subtraction is not stored back into A
- **INC** (increment) - increment the value in a specific register by 1
- **DEC** (decrement) - decrement the value in a specific register by 1
- **CCF** (complement carry flag) - toggle the value of the carry flag
- **SCF** (set carry flag) - set the carry flag to true
- **RRA** (rotate right A register) - bit rotate A register right through the carry flag
- **RLA** (rotate left A register) - bit rotate A register left through the carry flag
- **RRCA** (rotate right A register) - bit rotate A register right (not through the carry flag)
- **RRLA** (rotate left A register) - bit rotate A register left (not through the carry flag)
- **CPL** (complement) - toggle every bit of the A register
- **BIT** (bit test) - test to see if a specific bit of a specific register is set
- **RESET** (bit reset) - set a specific bit of a specific register to 0
- **SET** (bit set) - set a specific bit of a specific register to 1
- **SRL** (shift right logical) - bit shift a specific register right by 1
- **RR** (rotate right) - bit rotate a specific register right by 1 through the carry flag
- **RL** (rotate left) - bit rotate a specific register left by 1 through the carry flag
- **RRC** (rorate right) - bit rotate a specific register right by 1 (not through the carry flag)
- **RLC** (rorate left) - bit rotate a specific register left by 1 (not through the carry flag)
- **SRA** (shift right arithmetic) - arithmetic shift a specific register right by 1
- **SLA** (shift left arithmetic) - arithmetic shift a specific register left by 1
- **SWAP** (swap nibbles) - switch upper and lower nibble of a specific register

Reading through the guide on instructions, should give you enough information to implement all the instructions yourself.

Next, we'll be looking at how the CPU keeps track of which instructions to execute as well as different types of instructions that can change where we are in a particular program.

Program Counter

So far we've seen instructions that can operate on register data. But how does the CPU which instruction to execute? To understand this, we'll first need to understand where our instructions are stored.

Game ROM

So far we know that the Game Boy has a CPU that executes instructions and it has memory. Memory can be thought of as one very large array of 8-bit numbers.

At the beginning of this very long array are 255 bytes (from index 0x0000 to index 0x00FF) that are hard coded into the Game Boy's circuitry. These 255 bytes are instructions that tell the Game Boy how to "bootstrap" itself (i.e. get itself ready to execute a game) as well as display the iconic splash screen. Later in the book we'll be looking at specifically what these instructions do, but for now just imagine them as a collection of instructions many of which we learned in the previous chapter and the rest of which we'll learn in this chapter and the next few to come.

When the user of a Game Boy inserts a game cartridge, the contents of that cartridge become available to the CPU right after these 255 bytes. We'll talk later about where other things such as the contents of the screen and graphics data live in memory later in the book. For now we just need to know that the contents of memory starting at index 0x100 until index 0x3FFF include the contents of the cartridge.

So our memory is simply an long array of 8-bit numbers (0xFFFF or 65,536 of them to be exact). Each of these numbers can be decoded as an instruction that our CPU knows how to run. But how does the CPU know which of these to execute?

The Program Counter

Along with the register data, our CPU also holds on to a 16-bit number called the progam counter (often abbreviated as PC) which tells us which instruction the Game Boy is currently executing. This 16-bit number is capable of addressing of the of 0xFFFF numbers that live in memory. In fact, when we talk about the memory array we don't usually use the term "index", but instead the term "address".

Let's add a program counter to our CPU as well as memory that we can address from the CPU.

```
struct CPU {
  registers: Registers,
  pc: u16,
  bus: MemoryBus,
}

struct MemoryBus {
  memory: [u8; 0xFFFF]
}

impl MemoryBus {
  fn read_byte(&self, address: u16) -> u8 {
    self.memory[address as usize]
  }
}
```

We now have a program counter that can tell us at which address in memory the currently executing instruction is. We won't talk much more about the contents of memory or where certain things in memory live until later in the book. For now, you should just picture memory as a large array that we can read from.

Now we'll need to actually add the method to the CPU that uses the program counter to read the instruction from memory and execute it.

The full set of steps is as follows:

- Use the program counter to read the instruction byte from memory.
- Translate the byte to one of the instances of the `Instruction` enum
- If we can successfully translate the instruction call our `execute` method else panic which now returns the next program counter
- Set this next program counter on our CPU

```
impl CPU {
  fn step(&mut self) {
    let mut instruction_byte = self.bus.read_byte(self.pc);

    let next_pc = if let Some(instruction) =
Instruction::from_byte(instruction_byte) {
      self.execute(instruction)
    } else {
      panic!("Unkown instruction found for: 0x{:x}", instruction_byte);
    };

    self.pc = next_pc;
  }
}
```

So there's two things we'll need to add for the above to work. We'll need to change our execute method to return the next program counter, and we'll need to add a function that takes a byte and returns an `Instruction`. Let's start with latter. Decoding our instruction byte as an `Instruction` is very straight forward. Instructions are uniquely identified by the byte number. For instance, a logical `OR` with the `A` register as its target is identified by the byte 0x87. Want to do an `OR` with the `H` register as the target? That's the number 0xB4. The `SCF` (or Set Carry Flag) instruction is identified by the byte 0x37. We can use our instruction guide to find out which byte value corresponds to which `Instruction`.

```
impl Instruction {
  fn from_byte(byte: u8) -> Option<Instruction> {
    match byte {
      0x02 => Some(Instruction::INC(IncDecTarget::BC)),
      0x13 => Some(Instruction::INC(IncDecTarget::DE)),
      _ => /* TODO: Add mapping for rest of instructions */ None
    }
  }
}
```

And now let's change our `execute` method so that it now returns the next program counter:

```
impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::ADD(target) => {
        match target {
          ArithmeticTarget::C => {
            let value = self.registers.c;
            let new_value = self.add(value);
            self.registers.a = new_value;
            self.pc.wrapping_add(1)
          }
          _ => { /* TODO: support more targets */ self.pc }
        }
      }
      _ => { /* TODO: support more instructions */ self.pc }
    }
  }
}
```

Now we have the ability to read the instruction byte from memory that's pointed to by our program counter, decode that instruction byte as one of the variants of our `Instruction` enum, execute that instruction and get back the new program counter and finally set the new program counter on our CPU. This is how all instructions in the Game Boy get

executed! Well, except...

Prefix Instructions

The process we've laid out for how instructions get executed is true for roughly half of the total instructions the Game Boy can perform. The other half of instructions work the same way except that instead of being identified by a single byte they're first indentified by a prefix byte. This prefix byte tells the CPU, "Hey! The next instruction byte you read shouldn't be interpreted as a normal instruction, but rather as a prefix instruction".

This prefix byte is the number "0xCB". So, we'll need to add logic that first checks to see if the byte we read from memory is 0xCB. If it is, we then need to read one more byte and interpret this byte as an "prefix instruction". For example, if we read 0xCB from memory, we know that we're going to be decoding a prefix instruction. We then read another byte. If that byte is, say, 0xB4, we should *not* interpret this as `OR` with `H` as the target like we normally would but rather as a `RES` instruction with the 6th bit of the `H` register as the target. Again we can use the instruction guide to help us know what a given byte should decode as.

Let's put it in code!

```
impl CPU {
  fn step(&mut self) {
    let mut instruction_byte = self.bus.read_byte(self.pc);
    let prefixed = instruction_byte == 0xCB;
    if prefixed {
      instruction_byte = self.bus.read_byte(self.pc + 1);
    }

    let next_pc = if let Some(instruction) =
Instruction::from_byte(instruction_byte, prefixed) {
      self.execute(instruction)
    } else {
      let description = format!("0x{}{:x}", if prefixed { "cb" } else { "" },
instruction_byte);
      panic!("Unkown instruction found for: {}", description)
    };

    self.pc = next_pc;
  }
}

impl Instruction {
  fn from_byte(byte: u8, prefixed: bool) -> Option<Instruction> {
    if prefixed {
      Instruction::from_byte_prefixed(byte)
    } else {
      Instruction::from_byte_not_prefixed(byte)
    }
  }

  fn from_byte_prefixed(byte: u8) -> Option<Instruction> {
    match byte {
      0x00 => Some(Instruction::RLC(PrefixTarget::B)),
      _ => /* TODO: Add mapping for rest of instructions */ None
    }
  }

  fn from_byte_not_prefixed(byte: u8) -> Option<Instruction> {
    match byte {
      0x02 => Some(Instruction::INC(IncDecTarget::BC)),
      _ => /* TODO: Add mapping for rest of instructions */ None
    }
  }
}
```

The amount the program counter goes forward after each step of execution is determined by how "wide" the instruction - i.e. how many bytes it takes to describe the instruction in its entirety. For simple instructions, this is one byte - the byte the uniquely identifies the instruction. So far all the instructions we've seen either are 1 or 2 bytes wide (prefix instructions are two bytes - the prefix and the instruction identifier - while the other

instructions are only one 1 byte - just for indentifier). In the future we'll see other instructions which have "operands" or data the instruction needs to execute. These instructions can sometimes be even 3 bytes wide.

However, the program counter doesn't have to go forward by a set amount. In fact, there are instructions that manipulate the program counter in arbitrary ways sometimes sending the program counter to somewhere far away from its previous location.

Jump Instructions

The real power of computers are their ability to "make decisions" - i.e., do one thing given one condition and do another thing given another condition. At the hardware level this is usually implemented with "jumps" or the ability to change where in our program we are (as indicated by the program counter) based on certain conditions. In the case of the Game Boy's CPU these conditions are specificed by the flags register. For example, there is an instruction that says to "jump" (i.e., set the program counter) to a certain location if the flags register's zero flag is true. This gives the game a way to perform certain instructions and then change to different parts of the game code if the result of the instruction resulted in setting particular flags. Let's list out the types of jumps there:

- JP: Jump to a particular address dependent on one of the following conditions: the zero flag is true, the zero flag is flase, the carry flag is true, the carry flag is false, or always jump.
- JR: Jump a certain amount relative to the current program counter dependent on the same conditions above.
- JPI: Jump to the address stored in HI

You can find the specifics of how these jump instructions work in the instruction guide.

Implementation of jump is pretty trivial:

```
enum JumpTest {
  NotZero,
  Zero,
  NotCarry,
  Carry,
  Always
}
enum Instruction {
  JP(JumpTest),
}

impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::JP(test) => {
        let jump_condition = match test {
            JumpTest::NotZero => !self.registers.f.zero,
            JumpTest::NotCarry => !self.registers.f.carry,
            JumpTest::Zero => self.registers.f.zero,
            JumpTest::Carry => self.registers.f.carry,
            JumpTest::Always => true
        };
        self.jump(jump_condition)
      }
      _ => { /* TODO: support more instructions */ self.pc }
    }
  }

  fn jump(&self, should_jump: bool) -> u16 {
    if should_jump {
      // Gameboy is little endian so read pc + 2 as most significant bit
      // and pc + 1 as least significant bit
      let least_significant_byte = self.bus.read_byte(self.pc + 1) as u16;
      let most_significant_byte = self.bus.read_byte(self.pc + 2) as u16;
      (most_significant_byte << 8) | least_significant_byte
    } else {
      // If we don't jump we need to still move the program
      // counter forward by 3 since the jump instruction is
      // 3 bytes wide (1 byte for tag and 2 bytes for jump address)
      self.pc.wrapping_add(3)
    }
  }
}
```

It's important to note that the address we jump to is located in the two bytes following the instruction identifier. As the comment in the code example explains, the Game Boy is little endian which means that when you have numbers that are larger than 1 byte, the least significant is stored first in memory and then the most significant byte.

```
+-------------+-------------- +--------------+
| Instruction | Least Signif- | Most Signif-
| Identifier  | icant Byte    | icant Byte   |
+-------------+-------------- +--------------+
```

We're now succesfully executing instructions that are stored in memory! We learned that the current executing instruction is kept track of by the program counter. We then read the instruction from memory and execute it, getting back our next program counter. With this, we were even able to add some new instructions that let the game conditionally control exactly where the next program counter will be. Next we'll look at bit closer at instructions that read and write to memory.

Instructions for Reading and Writting to Memory

Now that we've seen how instructions get executed and the very basics of reading from memory in order to fetch the instructions to be read, we'll look now at instructions that are used to read and write from different parts of memory.

Loading Memory

First, when we talk about reading and writing memory, we usually use the term "load". We'll be loading data from some place to some place - for example, loading the contents of register A into memory at location 0xFF0A or loading register C with the contents from memory location 0x0040. Loading doesn't have to be between a register and a place in memory, it can also be between two registers or even two places in memory.

All of the instructions we'll be looking are called `LD` instructions. We'll be differentiating between the types of loads with the `LoadType` enum. The enum will describe what kind of load we're doing.

Let's take a look at the implementation of the `LD` instruction with the `LoadType` of `Byte` which loads a byte from one place to another.

```
            fn write_byte(&self, addr: u16, byte: u8) {} }
enum LoadByteTarget {
    A, B, C, D, E, H, L, HLI
}
enum LoadByteSource {
    A, B, C, D, E, H, L, D8, HLI
}
enum LoadType {
  Byte(LoadByteTarget, LoadByteSource),
}
enum Instruction {
  LD(LoadType),
}

impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::LD(load_type) => {
        match load_type {
          LoadType::Byte(target, source) => {
            let source_value = match source {
              LoadByteSource::A => self.registers.a,
              LoadByteSource::D8 => self.read_next_byte(),
              LoadByteSource::HLI =>
self.bus.read_byte(self.registers.get_hl()),
              _ => { panic!("TODO: implement other sources") }
            };
            match target {
              LoadByteTarget::A => self.registers.a = source_value,
              LoadByteTarget::HLI =>
self.bus.write_byte(self.registers.get_hl(), source_value),
              _ => { panic!("TODO: implement other targets") }
            };
            match source {
              LoadByteSource::D8  => self.pc.wrapping_add(2),
              _                   => self.pc.wrapping_add(1),
            }
          }
          _ => { panic!("TODO: implement other load types") }
        }
      }
      _ => { panic!("TODO: support more instructions") }
    }
  }
}
```

For loads with a register as a source, we simply read the register's value. If the source is a `D8` (meaning "direct 8 bit value"), the value is stored directly after the instruction, so we can simply call `read_next_byte` which reads the byte directly after the byte the program counter is currently pointing to. Lastly, if the source is `HLI` we use the value inside of the `HL` register as an address from which we read an 8 bit value from memory.

The target is merely the reverse of the source (except that we can't have `D8` as a target). If the target is a register, we write the source value into that register, and if the target is `HLI` we write to the address that is stored inside of the `HL` register.

The use of the 16-bit registers `BC`, `DE`, and `HL` to store addresses is very common.

Let's take a look at the other types of loads that there are:

- `Word`: just like the `Byte` type except with 16-bit values
- `AFromIndirect`: load the A register with the contents from a value from a memory location whose address is stored in some location
- `IndirectFromA`: load a memory location whose address is stored in some location with the contents of the A register
- `AFromByteAddress`: Just like `AFromIndirect` except the memory address is some address in the very last byte of memory.
- `ByteAddressFromA`: Just like `IndirectFromA` except the memory address is some address in the very last byte of memory.

For more detail on these instructions checkout the instruction guide.

These instructions have been for writing and writing to anywhere in memory, but there are a set of instructions that deal with a specific piece of memory called the stack. Let's take a look at what the stack is and the instructions that are used to manipulate the stack.

The Stack

Before we can look at the piece of memory in the Game Boy known as the stack, we need to have a good understanding of what a stack is more generally. A stack is a simple data structure that allows you to add values to it (a.k.a "push" values) and then get these values back (a.k.a pop them off the stack). The key thing to remember with a stack is that you pop items off the stack in reverse order from which you pushed the items on - i.e., if you pushed three items "A", "B", "C" on to a stack in that order, the order you will get them back when poping them off is "C", "B", "A".

The Game Boy CPU has built in support for a stack like data structure in memory. This stack lives somewhere in memory (we'll talk about how it's location in memory is set in just a minute), and it holds on to 16 bit values. How is it built?

First, the CPU has an additional 16-bit register on it that indicates the top of the stack. This register is called `SP` or stack pointer because it "points" to where the top of the stack is. Let's add this register to our CPU:

```
struct CPU {
  registers: Registers,
  pc: u16,
  sp: u16,
  bus: MemoryBus,
}
```

We have a stack pointer now so we know where our stack is, but how do we push and pop from this stack?

The Game Boy's CPU understands two insructions for doing just that. `PUSH` will write the contents of any 16-bit register into the stack and `POP` writes the head of stack into any 16-bit register.

Here's what's actually happening when a `PUSH` is performed:

- *Decrease* the stack pointer by 1.
- Write the most significant byte of the 16 bit value into memory at the location the stack pointer is now pointing to
- *Decrease* the stack pointer by 1 again.
- Write the least significant byte of the 16 bit value into memory at the location the stack pointer is now pointing to

Notice that the stack pointer is decresed by 1 and not increased. This is because the stack grows downward in memory. This is extra helpful since the normal place for the stack to live is at the very end of memory. In a later chapter we'll see that it's actually the Game Boy's boot ROM that sets the stack pointer to the very end of memory. Thus, when the stack grows it grows away from the end of memory towards the beginning of memory.

Let's implement `PUSH`:

```
impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::PUSH(target) => {
        let value = match target {
          StackTarget::BC => self.registers.get_bc(),
          _ => { panic!("TODO: support more targets") }
        };
        self.push(value);
        self.pc.wrapping_add(1)
      }
      _ => { panic!("TODO: support more instructions") }
    }
  }

  fn push(&mut self, value: u16) {
    self.sp = self.sp.wrapping_sub(1);
    self.bus.write_byte(self.sp, ((value & 0xFF00) >> 8) as u8);

    self.sp = self.sp.wrapping_sub(1);
    self.bus.write_byte(self.sp, (value & 0xFF) as u8);
  }
}
```

We can now push elements on to the stack. Here's what's actually happening when a `PUSH` is performed:

- Read the least significant byte of the 16 bit value from memory at the location the stack pointer is pointing to
- *Increase* the stack pointer by 1.
- Read the most significant byte of the 16 bit value from memory at the location the stack pointer is now pointing to
- *Increase* the stack pointer by 1 again.
- Return the value with the most and least significant byte combined together

Let's write `POP`:

```
impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::POP(target) => {
        let result = self.pop();
        match target {
            StackTarget::BC => self.registers.set_bc(result),
            _ => { panic!("TODO: support more targets") }
        };
        self.pc.wrapping_add(1)
      }
      _ => { panic!("TODO: support more instructions") }
    }
  }

  fn pop(&mut self) -> u16 {
    let lsb = self.bus.read_byte(self.sp) as u16;
    self.sp = self.sp.wrapping_add(1);

    let msb = self.bus.read_byte(self.sp) as u16;
    self.sp = self.sp.wrapping_add(1);

    (msb << 8) | lsb
  }
}
```

And there we have it! We have a working stack that we can used. But what sort of things is the stack used for? One built in use for the stack is creating a "call" stack that allows the game to "call" functions and return from them. Let's see how that works.

Calling Functions

In most programming languages when you call a function, the state of the calling function is saved somewhere, execution is allowed to happen for the called function, and then when the called function returns, the state of the called function is restored. It turns out the Game Boy has built in support for this mechanism where the state that is saved is simply just what the program counter was when the called function was called. This means we can "call a function" and that function itself can call functions, and when all of that is done, we'll return right back to the place we left off before we called the function.

This functionality is handled by two types of instructions `CALL` and `RET` (a.k.a return). The way `CALL` works is by using a mixture of two instructions we already know about `PUSH` and `JP` (a.k.a jump). To execute the `CALL` instruction, we must do the following:

- `PUSH` the next program counter (i.e. the program counter we would have if we were not jumping) on to the stack
- `JP` (a.k.a jump) to the address specified in the next 2 bytes of memory (a.k.a the function).

And that's it! We've called into our function. But what happens when we call `RET` (a.k.a return) from our called function? Here's what will happen:

- `POP` the next program counter off the stack and jump back to it.

Well that's easy! Let's see it in code:

```
impl CPU {
  fn execute(&mut self, instruction: Instruction) -> u16 {
    match instruction {
      Instruction::CALL(test) => {
          let jump_condition = match test {
              JumpTest::NotZero => !self.registers.f.zero,
              _ => { panic!("TODO: support more conditions") }
          };
          self.call(jump_condition)
      }
      Instruction::RET(test) => {
          let jump_condition = match test {
              JumpTest::NotZero => !self.registers.f.zero,
              _ => { panic!("TODO: support more conditions") }
          };
          self.return_(jump_condition)
      }
      _ => { panic!("TODO: support more instructions") }
    }
  }

  fn call(&mut self, should_jump: bool) -> u16 {
    let next_pc = self.pc.wrapping_add(3);
    if should_jump {
      self.push(next_pc);
      self.read_next_word()
    } else {
      next_pc
    }
  }

  fn return_(&mut self, should_jump: bool) -> u16 {
    if should_jump {
      self.pop()
    } else {
      self.pc.wrapping_add(1)
    }
  }
}
```

Now we can easily call functions and return from them. We're now done with the vast majority of our CPU instructions!

Finishing Up the CPU

We've nearly reached the end of building our CPU. There are a few instructions left to implement, many of which we won't touch on here, since they're closely tied with other parts of the Game Boy that we've yet to talk about.

Remaining Instructions

In this chapter, we'll look at two more instructions: `NOP` and `HALT`

NOP

`NOP` is perhaps the simplest of the Game Boy's instructions. It stands for no-operation and it effectively does nothing except advance the program counter by 1.

HALT

`HALT` is a big more complicated than `NOP`. When the Game Boy is running, it is constantly in a loop executing instructions. The `HALT` instruction gives the game the ability to stop the CPU from executing any more instructions. How the Game Boy eventually continues executing instructions will be discussed later in the book, but for now, we have the ability to stop the Game Boy dead in its tracks. Reasons a game might want to do this include saving battery. If the game doesn't have anything to do, it can halt the CPU and save a bit energy.

For now, we'll implement `HALT` by adding a `is_halted` boolean to the CPU. At the beginning of `execute` we can check if the CPU is halted. If it is, we simply return. The `HALT` instruction simply sets this field to true.

Where We Are

So far we've learned about the CPU and the many different instructions that the CPU can execute. We learned that these instructions live in memory which is just a long array of 8-bit numbers. The CPU reads in these bytes, decodes them as instructions and executes them. Some instructions simply do different arithmetic operations on the contents of the CPU's registers. Some instructions can cause the CPU to change its program counters, effectively "jumping" it to a different place in the game code. Some instructions read from and write to memory including a special part of memory we call the stack which behaves like a stack data structure. Finally, we learned about two special instructions: `NOP` which does nothing and `HALT` which stops the CPU from executing more instructions.

In the next section of the book, we'll be leaving the comfort of the CPU and exploring memory more closely.

Graphics

Up until this point, we've been reading and writting to memory and performing different operations in the CPU. Our CPU has lots of work that it is capable of doing and in many senses it can already 'run' games. But, of course, one of the main things that defines a video game is the video or graphics aspect of it.

In the next section of the book, we'll be deep diving into how graphics work in the Game Boy. First, we'll start with how the Game Boy's background graphics work. This will be enough for us to get the iconic boot screen displaying. Once we're done with this, we'll move on to sprite graphics that allow things like characters and enemies to appear.

So without further delay, let's get started!

Tile Ram

Before we can display background graphics to the screen, we have to have a good understanding of how background graphics actually work and where those graphics are stored in memory.

Game Boy games do not have direct control over what appears in the background. This is because the Game Boy is limited in how much it can store for graphics. The Game Boy has 0x1FFF (8191) bytes worth of storage for background graphics. Unlike more modern systems that employ a direct "frame buffer" (i.e., a long array of bytes where each byte or set of bytes describes how a corresponding pixel should be displayed on screen), the Game Boy uses a tiling system. This system allows the game to build 8 pixel by 8 pixel tiles and then place each tile on the screen at a certain index.

TODO: Graphic showing the difference between the two

Creating Tiles

So before we look at how pixels on the screen are shown, we first have to see how games manipulate and store tiles.

As we've seen before in our overview of the memory map of the Game Boy, tile data is stored between 0x8000 and 0x97FF (0x1800 or 6144 bytes worth of memory). This area actually contains two seperate tile sets. This allows the game to very quickly switch between two differnt graphic styles without having to switch the tiles out in the time between two screens. We'll explore how the game switches between the two tile sets a bit

later.

For now, we'll be focusing on the first tile set in memory that resides at 0x8000 to 0x8FFF (for a total of 0x1000 or 4096 bytes worth of data). Each tile is encoded in 16 bytes (we'll talk about exactly what this encoding looks like below). So if we 0x1000 bytes worth of memory and each tile is encoded in 16 bytes, then we have 0x1000 / 0x10 or 0x100 (256) different tiles.

An observant reader might wonder why the first tile set takes up 0x1000 of the 0x1800 or two thirds worth of space alloted for tile memory. The truth is that the second tile set starts at 0x8800 and goes to 0x97FF. The chunk between 0x8800 and 0x8FFF is therefore shared by the two tile sets.

TODO: Make nicer chart

```
8000-87FF: First part of tile set #1
8800-8FFF: Second part of tile set #1
           First part of tile set #2
9000-97FF: Second part of tile set #2
```

So how are each of the tiles encoded? First, we need to understand how many different colors a pixel of the Game Boy can display. The Game Boy is capable of displaying 4 different colors: white, light gray, dark gray, and black. The minimal number of bits that we need to encode this information is 2 bits since two bits can encode 4 different numbers: 0b00, 0b01, 0b10, and 0b11.

> *Learn More*
>
> The way the Game Boy hardware displays the 4 different colors is simply by emitting 4 different levels of white light. For "white" for instance the light is fully on, while for black the light is fully off. Light and dark gray are at 33% light and 66% light respectively. In fact, calling these colors white, gray and black isn't really true since the screen of the original Game Boy was green so the colors players see are actually shades of green.

The bit value to color mapping is as follows:

```
+------+------------+
| 0b11 | white      |
| 0b10 | dark-gray  |
| 0b01 | light-gray |
| 0b00 | black      |
+------+------------+
```

So each pixel of our 8x8 pixel (i.e., 64 pixels in total) tile will take 2 bits to represent. That means we'll need 64 * 2 or 128 bits total to represent all the pixels. In terms of number of bytes that's 128 / 8 or 16 bytes total as we've said above.

So this shouldn't be too hard to encode right? Just start from the top left most pixel and every two bits we encode that pixels value right? Unfortunately not. The actual encoding scheme is a little bit more complicated.

Each row of a tile is 2 bytes worth of data (8 pixels with 2 bits per pixel equals 16 bits or 2 bytes). Instead of each pixels value coming one after the other, each pixel is split between the two bytes. So the first pixel is encoded with the left most (i.e., most significant bit a.k.a bit 7) of each byte.

For example, let's imagine that the first two bytes of our tile set memory were 0xB5 (0b10110101) and 0x65 (0b01100101). These two bytes together will encode the data for the first tile. Byte 1 contains the value of the upper (a.k.a most significant) bit and byte 2 contains the value of the lower (least significant) bit.

Let's take a look at how this looks. In the following diagram that colors are represented by 1 letter "B" for black, "D" for dark-gray, "L" for light-gray and "W" for white.

```
              Bit Position
A             7 6 5 4 3 2 1 0
d           +-----------------+
d  0x8000   | 1 0 1 1 0 1 0 1 |
r           |-----------------|
e  0x8001   | 0 1 1 0 0 1 0 1 |
s           +-----------------+
s             D L W D B W B W
                  Color
```

Since reading the tile data happens much more often than writing it, we can store the tile data internally in a more friendly way.

Let's write some code to see what we need. First, we're going to create a new struct that will be responsible for all the graphics needs of the Game Boy. This loosely mimics the set up of actual hardware where the CPU knows nothing about graphics and all. There's no one chip responsible for graphics instead there is dedicated video RAM and the screen hardware. It would over complicate things if we tried to too closely mimic this set up.

Instead we'll create the `GPU` or "Graphic Processing Unit" to model all of our video needs.

For now, our GPU will hold on to video RAM and our tile set data. Our video ram is just a long array which holds on to raw byte values. The tile set will also be an array of tiles. A tile is simply an array of 8 rows where a row is an array of 8 `TileValue`s.

```
const VRAM_BEGIN: usize = 0x8000;
const VRAM_END: usize = 0x9FFF;
const VRAM_SIZE: usize = VRAM_END - VRAM_BEGIN + 1;

#[derive(Copy,Clone)]
enum TilePixelValue {
    Zero,
    One,
    Two,
    Three,
}

type Tile = [[TilePixelValue; 8]; 8];
fn empty_tile() -> Tile {
    [[TilePixelValue::Zero; 8]; 8]
}

struct GPU{
    vram: [u8; VRAM_SIZE],
    tile_set: [Tile; 384],
}
```

Let's go back to our memory bus to redirect any of writes in memory to our video ram to go to the GPU:

```
            fn write_vram(&self, addr: usize, value: u8) {  } }

impl MemoryBus {
    fn read_byte(&self, address: u16) -> u8 {
        let address = address as usize;
        match address {
            VRAM_BEGIN ... VRAM_END => {
                self.gpu.read_vram(address - VRAM_BEGIN)
            }
            _ => panic!("TODO: support other areas of memory")
        }
    }

    fn write_byte(&self, address: u16, value: u8) {
        let address = address as usize;
        match address {
            VRAM_BEGIN ... VRAM_END => {
                self.gpu.write_vram(address - VRAM_BEGIN, value)
            }
            _ => panic!("TODO: support other areas of memory")
        }
    }
}
```

Notice how from the `MemoryBus` we don't directly access the vram but instead go through two methods `read_vram` and `write_vram`. This is so we can easily cache our tile set in the `tile_set` field of our CPU. Let's take a look at how these are implemented.

`read_vram` is very simple as it actually just reads from the vram array:

```
impl GPU {
  fn read_vram(&self, address: usize) -> u8 {
    self.vram[address]
  }
}
```

However, `write_vram` is much more involved. Let's take a look at the code and go line by line to see what's happening:

```
impl GPU {
    fn write_vram(&mut self, index: usize, value: u8) {
        self.vram[index] = value;
        // If our index is greater than 0x1800, we're not writing to the tile
set storage
        // so we can just return.
        if index >= 0x1800 { return }

        // Tiles rows are encoded in two bytes with the first byte always
        // on an even address. Bitwise ANDing the address with 0xffe
        // gives us the address of the first byte.
        // For example: `12 & 0xFFFE == 12` and `13 & 0xFFFE == 12`
        let normalized_index = index & 0xFFFE;

        // First we need to get the two bytes that encode the tile row.
        let byte1 = self.vram[normalized_index];
        let byte2 = self.vram[normalized_index + 1];

        // A tiles is 8 rows tall. Since each row is encoded with two bytes a
tile
        // is therefore 16 bytes in total.
        let tile_index = index / 16;
        // Every two bytes is a new row
        let row_index = (index % 16) / 2;

        // Now we're going to loop 8 times to get the 8 pixels that make up a
given row.
        for pixel_index in 0..8 {
            // To determine a pixel's value we must first find the
corresponding bit that encodes
            // that pixels value:
            // 1111_1111
            // 0123 4567
            //
            // As you can see the bit that corresponds to the nth pixel is the
bit in the nth
            // position *from the left*. Bits are normally indexed from the
right.
            //
            // To find the first pixel (a.k.a pixel 0) we find the left most
bit (a.k.a bit 7). For
            // the second pixel (a.k.a pixel 1) we first the second most left
bit (a.k.a bit 6) and
            // so on.
            //
            // We then create a mask with a 1 at that position and 0s
everywhere else.
            //
            // Bitwise ANDing this mask with our bytes will leave that
particular bit with its
            // original value and every other bit with a 0.
```

```
            let mask = 1 << (7 - pixel_index);
            let lsb = byte1 & mask;
            let msb = byte2 & mask;

            // If the masked values are not 0 the masked bit must be 1. If
they are 0, the masked
            // bit must be 0.
            //
            // Finally we can tell which of the four tile values the pixel is.
For example, if the least
            // significant byte's bit is 1 and the most significant byte's bit
is also 1, then we
            // have tile value `Three`.
            let value = match (lsb != 0, msb != 0) {
                (true, true) => TilePixelValue::Three,
                (false, true) => TilePixelValue::Two,
                (true, false) => TilePixelValue::One,
                (false, false) => TilePixelValue::Zero,
            };

            self.tile_set[tile_index][row_index][pixel_index] = value;
        }

    }
}
```

We now have a cache of our tile ram so that not only do we have the information directly in VRAM but we also have it in a more accessible format.

Next we'll get into the details of rendering.

Memory Map

Up until now we've been treating the Game Boy's memory as one long array with 0xFFFF entries in it. While this was a helpful simplification for our work on the CPU (which doesn't know anything more about memory than that), in reality things are bit more complex. Sections of memory are actually used for very specific purposes. In fact there are parts of "memory" that are not actually backed by a RAM chip but instead are directly tied to things like the Game Boy screen or the Game Boy's audio device. Below we'll be "mapping" out memory and talking at a high level about what the different sections of the Game Boy's memory map do. We'll start at the very beginning of memory with address 0x0000 and work our way all the up to the top. This chapter should serve as the jumping off point for the rest of the book, so if you don't understand exactly how something works don't worry, we'll be going in much more detail later on.

0x0000 - 0x00FF: Boot ROM

When the Game Boy first boot's the very bottom 256 bytes of memory is occuppied with the boot ROM. We've talked a little bit about the boot ROM before - how it's responsible for bootstrapping the Game Boy to be able to run a game as well as for playing the iconic splash screen on boot. Later in the book, we'll be examining the boot ROM very closely.

0x0000 - 0x3FFF: Game ROM Bank 0

Once the Game Boy is done booting, it unmaps the boot ROM from memory so it is no longer available. From this point the area once occupied by the boot ROM all the way up to address 0x3FFF is occupied by game code loaded in from the cartridge. Inside of this memory are two areas worth noting:

- 0x0000 - 0x00FF - the area once occupied by the Boot ROM now holds memory called the "Interrupt Table". We'll be talking at length in the future about interrupt's, but for now the best way to think about them is just like "events" in higher level programming. When specific things happen the hardware automatically looks inside the interrupt table at specific locations for how to handle those events.
- 0x0100 - 0x014F - This area is known as the cartridge header area. It contains data about the cartridge that was loaded into Game Boy including its name, the size of cartridge ROM and even the nintendo logo. We'll talk about more about the contents of the cartridge when we talk about the boot ROM since it directly references this header area. If you want to really dive into specifics checkout the cartridge header guide

After these special areas is plain game code. The reason this area is refered to as Bank 0 is explained in the next section.

0x4000 - 0x7FFF: Game ROM Bank N

Game ROMs can be quite large - much larger than what can fit into the memory area available. To handle this, the Game Boy allows for ROM "bank switching". A ROM bank is simply a chunk of the cartirdge ROM. The game can switch in these chunks at run time into the area between 0x4000 - 0x7FFF. The first bank, bank 0, is always the same memory and cannot be switched out. Only the area of memory between 0x4000 and 0x7FFF is capable of being switched out. Later in the book we'll go into specifics of how this works.

0x8000 - 0x97FF: Tile RAM

This area of memory contains data about the graphics that can be displayed to the screen. The Game Boy uses a tiling system for grapics meaning that a game doesn't control the specific pixels that get drawn to the screen, at least not directly. Instead, the game creates a set of tiles which are square chunks of pixels. It can then place these tiles on the screen. So instead of saying "draw pixel 438 light green", it first says "create a tile with these pixels values" and then later "place that tile I made earlier in positon 5". The placement of tiles at certain positions happens in the next chunk of memory...

0x9800 - 0x9FFF: Background Map

As we described above, the Game Boy uses a tiling system for graphics. In memory 0x8000 to 0x97FF the game creates different tiles. These tiles however don't show up on screen. That happens in this section of memory where the game can map tiles to sections of the screen.

0xA000 - 0xBFFF: Cartridge RAM

Cartridges (being physical devices) sometimes had extra RAM on them. This gave games even more memory to work with. If the cartridge had this extra RAM the Game Boy automatically mapped the RAM into this area of memory.

0xC000 - 0xDFFF: Working RAM

This is the RAM that the Game Boy allows a game to use. Our idea of RAM really just being a plain old array where the game could read and write bytes to really only applies to this section of memory.

0xE000 - 0xFDFF: Echo RAM

This section of memory directly mirrors the working RAM section - meaning if you write into the first address of working RAM (0xC000), the same value will appear in the first spot of echo RAM (0xE000). Nintendo actively discouraged developers from using this area of

memory and as such we can just pretend it doesn't exist.

0xFE00 - 0xFE9F: OAM (Object Atribute Memory)

This area of memory contains the description of graphical sprites. The tiles we talked about above were used for backgrounds and levels but not for characters, enemies or objects the user interacted with. These entities, known as sprites, have extra capabilties. The description for how they should look lives here.

0xFEA0 - 0xFEFF: Unused

This area is completely unmapped: reading from it just returns 0s and writing to it does nothing.

0xFF00 - 0xFF7F: I/O Registers

This is one of the most dense areas of memory. Practically every byte has a special meaning. It's used by both the screen and the sound system to determine different settings. We'll be talking a lot about this in the future.

0xFF80 - 0xFFFE: High RAM Area

This area is also just normal RAM but is used a lot because some of the `LD` instructions we've already seen can easily target this area in memory. This area is also sometimes used for the stack, but the top of working RAM is also used for this purpose.

0xFFFF: Interrupt Enabled Register

The very last byte of memory has special meaning used for handling interrupt events. We'll be looking at this closely later in the book.

Now that we have an idea of what all the areas of memory are used for we can start diving

more in detail. The first area we'll be looking at is at tile RAM and the background map along with a few of the I/O registers that will help us finally get some graphics on a screen!

Appendix

Numbers in Computers

In this guide, we'll look at how numbers are stored in the Gameboy's CPU, RAM, and ROM. In this guide we'll be using different types of number notations: binary, decimal and hexadecimal. If you're unfamiliar with these different ways of writing numbers, check out our guide on number notations.

Bits

At a very basic level, computers can only read and write two different values that we'll call 1 and 0. This piece of data is called a bit. Computer memory is really just a long array of bits that the computer can read or write.

> **Learn More**
>
> Computers normally represent bits as either one voltage (e.g., five volts) or as some other, typically lower, voltage (e.g., zero volts). Again, a great resource for learning about how computers actually deal with bits, check out Ben Eater's series on making an 8-bit Breadboard Computer.

Bits can represent only two different values 0 or 1. If we want to numbers larger than one we need to compose bits together. To get to three for instance we would write 0b11. The total count of numbers we can represent is equal to 2^(# of bits). So one bit can represent 2^1 a.k.a two numbers and 7 bits can represent 2^7 a.k.a 128 numbers.

Since being able to manipulate numbers larger than 1 is pretty useful, we normally talk about and the computer typically reads and writes bits in large chunks called bytes.

Bytes

Bytes are defined as a collection of 8 bits. Our Gameboy, as an 8-bit machine, typically

deals with one byte at a time and each compartment in memory stores one byte. However, the Game Boy also has 16-bit instructions which act on two bytes at a time. A byte can represent numbers 2^8 a.k.a 256 numbers (0 to 255) while 8 bytes (composed of 64 bits) and can represent 2^64 a.k.a 9,223,372,036,854,775,808 numbers (0 to 9,223,372,036,854,775,807).

> **Learn More**
>
> Some times we'll actually only deal with half a byte (i.e., 4 bits) at a time. This is usually referred to as a "nibble".

Since writing out bytes in binary can be quite tedious, we normally write out bytes in hexadecimal notation: So while we could write out the byte representing the number 134 as "0b10000110" we typically write it as "0x86". These two notations specify the same number, "0x86" is just shorter so it's more often used.

When disucssing numbers composed of multiple bytes, for example 0xFFA1 (composed of three bytes), we'll often need to talk about which byte is "most significant" (MSB - most significant byte) and which is "least significant" (LSB - least significant byte). Going back to math class, you may remember that when writing numbers like "178", the digit on the right (i.e., the "8") is the least sigificant, it adds the least amount to the total sum of the number (just eight) while the digit on the left (i.e., the "1") is the most significant since it adds the most to the sum of the number (one hundred!). Bytes work the same way - in 0xFFA1, 0xFF is the most significant byte and 0xA1 is the least significant.

Endianess

Let's take the example of two bytes sitting next to each other in memory: first at address 0 there is 0xFF and then at address 1 there is 0x16. If we want to read these two bytes together as a 16 bit number, should it be read as 0xFF16 or as 0x16FF? Even if one way or the other makes more sense to you, the answer is: it depends on the machine. In the case of the Game Boy the order is 0xFF16 - in other words the least significant byte is first in memory. This scheme is known as little-endian and its opposite is known as big-endian.

Signed Numbers

Ok so we know how to conceivably represent any number from 0 to some very large

positive number. We can just keep adding bytes until we have enough to represent our number. But what about negative numbers? Well one way we could chose to do it (and the way the Game Boy does it) is using something called the "two's complement".

Let's say we have the number 0b00111011 a.k.a. 59 and we want to represent -59 instead. In two's complement, we do the following:

- Invert every digit - 1s become 0s and 0s become 1s
    - 0b00111011 becomes 0b11000100
- Add 1 to the number
    - 0b11000100 becomes 0b11000101

So -59 is 0b11000101. But wait is 0b11000101 already 197? Yes it is! Whether we chose to interpret a byte as a number from 0 to 255 or as two's complement number capable of representing -128 to 127 is up to programmer! Interpreting a number as only positive means it is "unsigned" and interpeting as being possibly negative with two's complement means it is "signed".

Overflow and underflow

When doing arithmetic on numbers, sometimes the result is too large or small to be represented. For example if you add two 8 bit numbers 253 and 9 together you would expect to get 262. But 262 cannot be represented by 8 bits (it requires 9 bits). When this happens the number simply is what the first 8 bits of 262 would be just with the final 9th bit missing: 0b0000_0110 a.k.a 6. This phenomenon is called overflow. The opposite can occur when subtracting. This is called underflow

Rust

In Rust, the various number types tell us both how many bits are used to represent that particular integer and whether the integer is in two's complement or not. For example, the number type `u8` is a number composed of 8 bits (i.e., 1 byte) and is unsigned while `i64` is a number composed of 64 bits (i.e., 8 bytes) and is signed.

Number Notations

In this guide we'll deal with the various ways that we can think about and write down numbers.

Number Notations

Binary

The smallest amount of data that computers, including the Game Boy, deal with are bits. A bit is either one of two values. When we talk about bits we say that a bit is one of two distinct values: "1" or "0". Bits in other words are "binary" (i.e. relating to, composed of, or involving two things - "bi" is a latin prefix meaning two).

So we can talk about bits by using either the symbol "1" or the symbol "0". This way of talking about numbers is called "binary notation". This is different from the way we normally talk about numbers where we have ten different possibilities composed of ten distinct symbols: "0", "1", "2", "3", "4", "5", "6", "7", "8", or "9" - a.k.a. decimal notation ("deci" is a latin prefix meaning ten). When talking binary we'll never need any other symbols besides "0" and "1".

> **Side Note**
>
> We don't *have* to use the symbols "1" and "0" for binary. People sometimes use other ways of representing the two distinct values including: "yes" or "no", true or false, on or off, high or low, and more! In fact, computers normally represent bits as either one voltage (e.g. five volts) or as some other, typically lower, voltage (e.g. zero volts). Again, a great resource for learning about how computers actually deal with bits, check out Ben Eater's series on making an 8-bit Breadboard Computer.

So in binary notation we can represent the number zero with the symbol "0" and the number one with the symbol "1". What if we want to represent the number two? We just said we don't have the symbol "2" in binary - we only have either "1" or "0". Well let's think about what we do in our "normal" decimal system when we run out of symbols to use? In other words what happens when we're going from nine (represented by "9") to ten? Well, we reset the first digit to "0" and add a new one starting with "1" and we end up with "10" for ten.

This is exactly what we do in binary. So, to represent the number two, we reset the first digit to "0" and a "1" to the left of it. The number two is therefore "10".

So what's the number three then? You guessed it! "11". Let's count to ten in binary!

Zero "0" One "1" Two "10" Three "11" Four "100" Five "101" Six "110" Seven "111" Eight "1000" Nine "1001" Ten "1010"

We can now count as high as we want in binary!

Since binary numbers can look an awful lot like decimal numbers, it's helpful to have a way to say "this number is written in binary not in decimal!". A common way this is done (and how both we will do it in this book and how Rust also does it) is by prefixing numbers written in binary with "0b". So, the number three would be written "0b11"

> **Side Note**
>
> In order to avoid confusion, the best way to read binary numbers outloud (or in your head) is by reading each digit instead of using the decimal name for the number. So the number "0b1001" should either be read as "nine" or as "one zero zero one" never as "one thousand and one".

Hexadecimal

One interesting and important thing to note is at what numbers we add an extra digit when using binary. Let's list them: two, four, eight, sixteen, etc. Can you spot the pattern? Each number is a power two! Another way to call a counting system that works like this is a "base two" counting system. Our decimal system on the other hand gains a digit at every number that is a power of ten - so it is a "base ten" counting system.

The natural question might arise: "are there any other types of counting systems besides base two and base ten". And the answer is yes! While there are many different systems, the only other one we'll make use of in our emulator is hexadecimal - base sixteen!

So, base two has two different symbols and base ten has ten different symbols, which means base sixteen should have sixteen different symbols. And here they are: "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e", "f". So, instead of writing "10" for ten, we would now write "a" and eleven would be "b" and so on. Once we run out of symbols (i.e. after fifteen - "f"), we do what we've always done: reset our number to "0" and add a digit. Sixteen is written "10".

Just like how in binary, we use "0b" to make it obvious we're talking binary and not decimal, we use "0x"for hexdecimal. So "0x18" is twenty-four not eighteen.

Bit and Byte Manipulation

In this guide, we'll be discussing bit and byte manipulation through different bit-wise

operators. If you're a little rusty on bits, bytes, binary and hexadecimal, check out our guide on number notation and our guide on numbers

CPUs normally do very simple operations on their registers. Most of these operations should be familiar to everyone, but some involve manipulating the bits of a byte in some structured way. Let's take a look.

AND (&)

Bitwise AND (&) is similiar to the boolean AND (&&) operator you're probably already familiar with except that it operates on each bit of a byte or set of bytes. It's important to remember that boolean values are equivalent to a single bit: 1 is true and 0 is false.

If we have two boolean values we already know how to AND them:

```
true  && true  == true
false && true  == false
true  && false == false
false && false == false
```

For bitwise operators we simply do the same operation on the the bits of two numbers that are the same digit place. If we encounter a `1` and `1` it becomes `1`. All other combinations are `0`.

Let's take a look at an example:

```
  1001
& 1100
------
  1000
```

Of course, we can do the same thing with hexadecimal numbers. The best way to think of these when you're getting used to them is to convert the hexadecimal number to binary and then do the AND:

```
  0x8  1000
& 0x3  0011
------ ----
  0x0  0000
```

Bitwise AND is often used to get the value at a specific bit or byte number. For example, if you have a 4 byte value and you just want the value in the third byte position you can AND the number with another value that has zeros in every position but the thrid byte - this

practice is known as "bit masking".

OR (|)

Just like bitwise AND, Bitwise OR (|) is similiar to the boolean OR (||) operator except that it operates on each bit of a byte or set of bytes.

If we have two boolean values we already know how to || them:

```
true  || true  == true
false || true  == true
true  || false == true
false || false == false
```

Let's take a look at a bitwise example:

```
  1001
| 1100
------
  1101
```

The same advice around hexadecimal applies for bitwise OR as it did for AND. When in doubt, convert to binary and go bit by bit.

Bitwise OR can be used to combine two binary values together. For example, if you want to be sure that a value has it's least significant bit set to 1, you can OR it with 0b1.

Shift (<< >>)

Bit shifting is the practice of moving bits in order to different digit positions.

To accomplish this first we look at the operator to see what direction we're going: `<<` for left and `>>` for right. We then take each bit and shift it over as many places as the number to the right of the operator tells us to. Any numbers that no longer fit (i.e. they've been shift off the edge of the number) disappear. All digit places that are left empty by the shifting get replaced by 0s.

Let's take a look at some examples:

```
1001 >> 1 == 0100
1001 >> 2 == 0010
1001 >> 3 == 0001
1001 >> 4 == 0000

1101 << 1 == 1010
1101 << 2 == 0100
1101 << 3 == 1000
1101 << 4 == 0000
```

Again, if you have a number in hexadecimal notation, you can first convert it to binary and go bit by bit.

One thing bit shifting is used for is to get the values of certain bytes in a number composed of many bytes. For example, in the following snippet, we're trying to get the value of the most significant byte:

```
0x1c74a3 >> 16 == 0x1c
```

Resources

Instruction Guide

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8
0x0									
0x1									
0x2									
0x3									
0x4									
0x5									
0x6									
0x7									
0x8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A, (HL)	ADD A,A	
0x9									
0xa									
0xb									

0xc	ADD A,d8
0xd	
0xe	
0xf	

Arithmetic Instructions
ADD
1
4
?
0
?
?
Add the value stored in a target register to the value in register A
Targets: A, B, C, D, E, H, L, BC, DE, HL, SP, d8, SP

Cartridge Header