Chapter 3

Machine Language

Machine Language

Data on its own is useless. We need a way of manipulating it so we can make the computer do anything. The lowest level language a computer can understand is known as *machine language*.

Machine language is a set of instructions and an encoding scheme, which together allow for data manipulation. Each instruction using machine language is known as a machine instruction and can be broken up into two parts: the opcode and the operand. Instructions are formed by creating bit patterns, commonly converted to hexadecimal for simplicity. One such instruction may look like this:

4A 41

In this example, the most significant digit (referred to as a "nibble" as it is half of a byte) is the opcode. It tells the computer what to do with the following numbers, the operand. Our machine language interpreter would read this as "move the contents of register 4 into register 1."

First, we should talk about how data is stored and used by the machine. We can separate our data into two parts, the main memory, and the registers.

Main memory can be thought of as a list of numbers to be read as instructions or data. For our machine language interpreter, there are 256 cells of main memory, thus, we can store 256 bytes here. As

each instruction is made up of 2 bytes, we can have up to 128 different instructions per program. A program counter tells the machine which instruction to look at and execute. We've already discussed that the machine reads two bytes as one instruction, so to do this the counter increments by two after each instruction is read. While instructions can be read from main memory, data cannot be manipulated from there.

Registers work very differently from main memory. While they are also a part of the system's memory, their contents cannot be read as instructions. Instead, data is loaded into them, and operations are then performed using their contents. This is because registers are located inside the CPU, where all the computations take place. Main memory is only connected to this through a bus which is used to transfer data. Our machine utilizes 16 registers, each of which can hold 1 byte.

CPU Architecture

While our machine uses one hexadecimal digit for each opcode, using two bytes per instruction total, this isn't the case for real machines. Real computers must be able to utilize a larger set of instructions. Some deal with these instructions by only using the most basic that are necessary for its function, while some instead opt to include more advanced instructions as well. These advanced instructions aren't necessarily better, as no functionality is added by including them, but they can be far more convenient. CPUs with less instructions, but no redundancy, are known as "Reduced Instruction Set Computers," or RISC, while those with larger, convenient

instruction sets are known as "Complex Instruction Set Computers," or CISC.

RISC CPUs focus on having many, one-step commands. This means multiplying a number may take many instructions, but each one will only take a short time to perform. CISC, on the other hand, may only take one instruction to do this, but it may take several steps to perform this command. Thus, we have a tradeoff consisting largely of RISC taking more of the programmer's time and more memory to store all the instructions, while CISC will take less memory to store, but each instruction will be slower.

Because CISC CPUs can have large instruction sets, they generally will use what are known as variable-length instructions. In other words, one instruction may take 4 bytes to store, while another may take only 3, or may even take 6. RISC CPUs use fixed-length instructions instead, so all instructions will take up the same amount of memory to store.

The Instruction Cycle

For a computer to perform an operation, it must go through what is known as the instruction cycle. This consists of three distinct steps, known as fetch, decode, and execute.

Fetching occurs when the instruction register gets updated with new data. The program counter looks at a byte and sends this and the next to the instruction register. If the program counter is at 00, then the values at cells 00 and 01 are sent to the instruction register.

Decoding occurs after fetching, as the computer must determine what the instruction means and how it will be executed. Here the opcode is separated from the operands.

Finally, the computer can execute the instruction. An operation determined by the opcode runs on the operands, and the program counter gets incremented. This cycle runs until either an error occurs, or a halt instruction is encountered.

Machine Instructions

All instructions in a set can be grouped into three categories: flow control, logic and arithmetic, and data movement.

Flow control, or JUMP instructions, are used to change where the program is being executed from. For instance, jumping from memory cell 00 to memory cell 2E will cause the program to skip over everything from cell 03 to cell 2D, then continue executing instructions at 2E. These jumps can be conditional or unconditional. A conditional jump will look at the value in each register to determine whether a jump should take place or not. An unconditional jump, on the other hand, will always occur when encountered. Both have their unique purposes and are important in their own ways. Finally, the HALT operation tells the machine that execution should stop. At this point, the program counter stops incrementing and the instruction register is no longer updated, thus the instruction cycle is no longer running.

Logical and arithmetic operations are those used to manipulate data within registers. These can largely be thought of like math operations. For these we have three arithmetic operations

and three logical operations. Addition is the easiest operation to understand, as it fits with our classical ideas of math. Our other arithmetic operations are left and right bit shifts. For logical operations, we have our binary operations from the first chapter. These include AND, OR, and XOR.

Finally, we have data transfer operations. These encompass our LOAD functions, which move data from main memory to registers, our STORE function, which moves data from registers to main memory, and our COPY operation, which copies data from one register to another. We must use these functions because, as we discussed, registers and main memory are separate. Arithmetic and logic operations cannot happen in main memory, and instructions cannot be executed in registers, so we move them back and forth.

Example of Machine Instructions

Let's take a look at each of the specific instructions at our disposal.

First, we have the two LOAD instructions, 1RXY and 2RXY. These appear to be the same, just with different opcodes, but they have slight differences which are very important. 1RXY loads a *value*, while 2RXY loads the *value at a memory cell*. This means the code 1023 will *LOAD* the hexadecimal value 23 into register 0. On the other hand, 2023 means *LOAD* the value located at memory cell [23] into register 0. So, 1RXY loads the *value* XY into register R, while 2RXY loads the value *at* [XY] into register R.

Next, we have our STORE instruction, 3RXY. This again looks like our LOAD instructions. In this case, we will *STORE* the value

located in register R into memory cell [XY]. If we had the code 3110, we would *STORE* the value from register 1 into cell [10].

Lastly for our transfer instructions, we have COPY, 40ST. Here, our instruction looks very different from the ones before it. What this means is *COPY* the value from register S into register T. Also, important to note is that the "0" can be any value. Thus, the codes 4012 and 4F12 will execute exactly the same. In both cases, we would *COPY* the value from register 1 into register 2.

Now we have our arithmetic and logical operators.

First is our ADD function, 5RST. This *ADD*s the values from registers S and T, then places the result into register R. If we had 5012, we would *ADD* the value of register 1 to the value of register 2, then place this sum into register 0.

We then have OR, 6RST, AND, 7RST, and XOR, 8RST. These all work similarly, using the different logic operators. In each case, we take the value in register S, use the logical operation on it with register T, then store the result in register R. 6112 would give us the value at register 1 *OR* the value at register 2, stored into register 1. 701A would give the value at register 1 *AND* the value at register A, placed into register 0. Finally, 8501 gives us the value at register 0 *XOR* the value at register 1, stored into register 5.

After these are our SHIFT operations, AR0X and BR0X. Like with the COPY command, the 0's can be any value, as they aren't used in the instruction. AR0X is used to *SHIFT* register R right by X bits, and BR0X *SHIFT*s register R left by X bits. Thus, A094 would *shift* register 0 right by four bits, and B004 would *shift* register 0 *left* by four bits.

Lastly, we have our flow control instructions.

The first of these are our JUMP commands, CRXY and DRXY. The first of these, CRXY, checks to see if register R is equal to register 0. If it is, the program counter will *JUMP* to cell [XY] and the program continues. If it is not, then the program counter simply increments and continues as if no instruction occurred. The instruction C100 will check to see if the value in register 1 is equal to the value in register 0. If it is, then the program will *JUMP* to cell [00], otherwise it will continue normally. As a helpful side effect of comparing register R to register 0, we can make our JUMP always occur by setting R to 0. In other words, C010 will *always JUMP* to cell [10]. This is because the computer will check if the value at register 0 is equal to the value at register 0, which must always be true. In other words, it checks if the value in register 0 is equal to itself.

DRXY will instead check to see if the value in register R is *less than* the value at register 0. If this is true, we will *JUMP* to cell [XY]. Like with the CRXY operation, this also has an interesting side effect, although this one is far less useful. In this case, using D025 will *never JUMP* to cell [25]. This is because the value in register 0 will never be less than the value in register 0. In other words, the value in register 0 can never be less than itself. As a result, the instruction can never cause a JUMP. If we instead used the instruction D125, the computer would check to see if the value in register 1 is less than the value in register 0. If it is, then the program counter will *JUMP* to cell [25].

Our final operation is the HALT instruction, E000. When the opcode "E" is encountered, the machine stops reading new instructions, and the program terminates, or *HALT*s. This is how we can safely end a program without getting an error. Once again, the 0's can be replaced with any values, so the instructions E000 and E9F1 will both just *HALT* the machine.

All our remaining possible opcodes (**0**, **9**, and **F**) will only give us errors, as they do not correspond to any instructions.

Programs and Data

Main memory is used to store more than just instructions. We may also want to simply store data to be accessed and written to. This data does not need to be read by the instruction register, but instead can be used to store the data our program might use. For instance, here we could store things such as the values we are using for a complex equation.

Data and programs can be stored *anywhere* in main memory and are essentially indistinguishable. This makes programs easy to write to and read from during execution. While this can be very useful for having a dynamic program, this also makes running into errors far easier, as we can no longer determine exactly how the program should be laid out as easily.