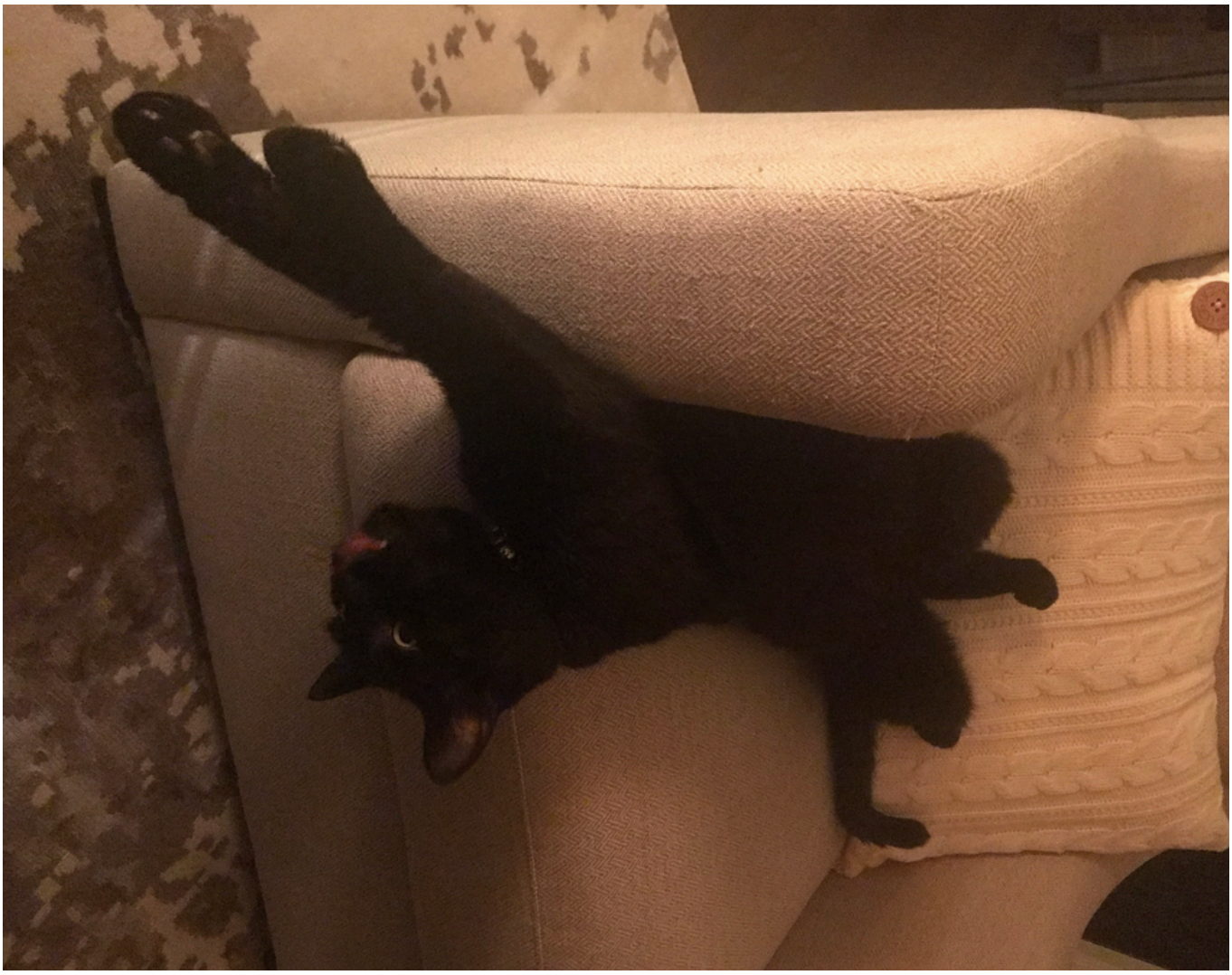
Watson - the guide!

Named Proudly after my cat Watson 🐱.

He's a little weird but that's alright.



Created by Benicio Hernandez

I pledge my honor that I have abided by the Stevens Honor System.

First things, first:

The Assembler!

-- Requirements --

Watson's assembler is written in Typescript, and runs on Node.js. Assuming you're using Ubuntu (or another Linux distro), getting these requirements set up is quite easy. Due to Ubuntu

shipping *very* old versions of Node.js, I made a script `nodesetup.sh` to locally setup Node.js version `v21.7.3`. This script depends on `curl` and `tar`, so please make sure those are installed. You can install them easily with the following commands:

- Ubuntu: `sudo apt install curl tar`
- Arch: `sudo pacman -S curl tar`

Next up:

-- Building --

First, `cd` into the `assembler` directory. You should see some directories such as `src` and files like `package.json` and `tsconfig.json`. Once you're in this directory, run the following commands to locally set up Node.js and compile the assembler:

```
mkdir build
chmod +x ./nodesetup.sh

# run node setup (installs to ./.bin)
./nodesetup.sh
source .env

# install dependencies (typescript, type definitions, source map support)
yarn install

# compile the code
yarn run tsc
```

This will output executable JavaScript code in the `build` directory. You can run the assembler using this command:

```
node build/main.js
```

Once you're done with this project, you can just delete the entire folder, which will completely remove this local Node installation.

Now you're ready to assemble some code!

-- Usage --

The assembler itself is quite simple. It accepts two arguments: The path to the input code (.s), the path to the output rom (.rom). You can use the assembler like this (from the assembler directory):

```
node build/main.js program.s program.rom
```

As long as there are no errors in you code, the assembler should cleanly exit (code 0) with no messages in the terminal. I *did* implement some error checking in within the assembler, so feel free to test it out!

Now that you know how to assemble your code, lets learn the language!

Watson Assembly! (watsm)

Watson's assembly code is very similar to ARM64 assembly. The Watson CPU is 16-bit, meaning it allows use of 8 general purpose registers `X0 - X7`, each of which stores one 16-bit unsigned integer value. It also makes use of several instructions which allow you to manipulate data within the CPU and memory. Most instructions can accept values from either immediate values, or other registers. To signify immediate values, I will use the term `imm`. I will also use terms `xDest` `xSrc1` and `xSrc2` to signify the destination of data, the first source and the second source respectively.

Watson provides 8 different instructions:

MOV

`MOV`, the most basic instruction, allows you to either copy values between registers, or assign values to registers directly. Think of this like `a = b`, or `a = x` for an immediate value.

```
MOV    x0, x1    # x0 = x1
MOV    x0, 3     # x0 = 3
```

It's binary representation is as follows: `1 000 001 000 000 011` *(2nd instruction)*
The mod bit (first bit) controls if the command reads the immediate value or a third register.
The next three bits represent the opcode, which is `000` for `MOV`.
The next 9 bits represent Register parameters 1, 2 and the Write register (in bit sets of 3). The final three bits represent the immediate value.

So this binary instruction is equivalent to the assembly instruction: `MOV x1, 3`

Most other instructions will follow this architecture, so I will provide their opcode and any difference they have from this point forward.

ADD

Adding allows you to add two numbers together. Think of it like `a = b + c`. You must specify the destination register (a), the first source register (b) and either a second source register or an immediate value (c). For example:

```
1   ADD    x0, x1, x2    # x0 = x1 + x2
2   ADD    x0, x1, 5     # x0 = x1 + 5
```

The binary representation of instruction 1 would be: `0 011 000 001 010 000`
With `011` as the opcode for `ADD`.

SUB

Similarly, you can use `SUB` to subtract values. This operates the same way as `ADD`, but subtracts values instead. This instruction is equivalent to `a = b - c`.

```
1   SUB    x0, x1, x2    # x0 = x1 + x2
2   SUB    x0, x1, 2     # x0 = x1 - 2
```

Binary of instruction 2 (with immediate): `1 100 000 001 000 010`
Notice how the mod bit is `1` here, and the immediate (last three bits) represent the digit `2`.

STR / LDR

These two commands allow you to write to and read from memory. They each accept there parameters: a source / destination register (for the data to be read from or stored to), a base register (for the position in memory) and an offset register (for how far ahead of the base you want to store data). Both `LDR` and `STR` accept immediate values for the offset value.

```
1   STR    x0, x1, x2    # Store data from x0 to address x1 with offset x2
2   LDR    x0, x1, 7     # Load data from address x1 with offset 7
3                                    to register x0
```

`STR` and `LDR` are essentially the same command, with the same paremeters and mod functionality. The difference is that `STR`'s opcode is *one higher* than `LDR`. Being `010` and `001` respectively.

I'll translate both here to showcase the mod functionality a little more:
`STR` (instruction 1, no mod): `0 010 000 001 010 000`
`LDR` (instruction 2, mod): `1 001 000 001 000 111`

CMP

`CMP` allows you to compare two values! Unlike other instructions, `CMP` does not store values to a regular register. Instead, it's result can be used in other commands such as `BEQ`, which I will cover next. Using `CMP`, you can compare either two registers, or a register and an immediate value.

```
CMP    x1, x2    # Compare values in x1 and x2
CMP    x1, 3     # Compare value of x1 to 3
```

`CMP` is an interesting case, because it only uses two arguments and does not need a destination register. Just like other operations, it can still use the mod bit to select immediate values to compare to. It has the highest opcode of all the instructions, being `111`

No mod: `0 111 001 010 000 000`
Mod: `1 111 001 000 000 100`

Labels

Let's take a brief intermission to cover labels. Labels are not instructions like the previous ones. Instead, they represent points in the code you can jump to using instructions like `B` and `BEQ`. Labels are formatted as so `<Label name>:`. For example:

```
MOV    x0, 0    # instruction 1
label:
MOV    x1, 1    # instruction 2
```

Here, jumping to `label` will immediately run instruction 2, without running instruction 1. However, normally running through instructions will not be affected by the presence of a label.

Fun-ish fact: Labels do not have any binary representation! This is because they are actually not part of the generated machine code. They are only used in the assembler to calculate offset values used in branching instructions at the hardware level.

Now, lets `jump` to branching!

B

Branching is an instruction that allows you to "jump" to labels in the code. These are useful for things like loops! `B` is a form of branching called **unconditional branching**. Reaching this instruction will jump to the label *no matter what*. Be careful because this can lead to infinite loops! The branching instruction is formatted as follows: `B label`. For example:

```
MOV    x0, 0          # x0 = 0
loop:                 # loop label
```

```
3   ADD     x0, x0, 1       # x0 = x0 + 1
4   B label                 # branch to loop label
```

Every time this code reaches the `B label` instruction, it jumps to the instruction immediately *after* the label, which in this case is `ADD`. This code starts `x0` as 0, and adds to it forever using the `B` instruction!

Since labels are not used in the actual machine code, branching instructions have a rather unique architecture. The first major difference is they **only** use immediate values. No registers are read when branching. The second is that the mod bit controls the *direction* of the jump taken by the branch. `mod=1` means branching backwards the number of instructions specified by the immediate value, and `0` means branching forwards. Due to the immediate value being 3 bits, it is limited to a maximum value of `7`, meaning you can branch up to `7` instructions away in either direction. You can extend this by chaining together multiple branch instructions and labels! For example

To branch `5` instructions backward: `1 101 000 000 000 101`

But what if you want your loops to end at some? This quite reasonable expectation is addressed by... drum roll please ...

BEQ

The `BEQ` instruction, standing for **B**ranch if **EQ**ual, works similarly to `B`, with the difference of only branching if a certain condition is met. The condition in question is that of the `CMP` instruction! Using `CMP` *immediately before* `BEQ` allows `BEQ` to branch depending on the comparison made in `CMP`. Specifically, `BEQ` will branch if the two values in the `CMP` instruction are **equal**. **It is important that `BEQ` is used right after `CMP` is called, otherwise the result of `CMP` will no longer be valid!** For example (adding on to the previous example):

```
1   MOV     x0, 0           # x0 = 0
2   loop:                   # loop label
3   ADD     x0, x0, 1       # x0 = x0 + 1
4   CMP     x0, 5           # x0 = 5?
5   BEQ     byebye          # if so, branch to byebye
6   B label                 # otherwise, branch to loop label
```

As you can see here, the combination of `CMP`, `BEQ` and `B` allows you to conditionally branch out of a loop! In this case, `x0` is being added to, and compared against the immediate value `5`. So, this loop will run `5` times, and then branch away to the label `byebye`. You can think of this almost like a `for` loop, where `x0` is i, the comparison is `i != 5`, and the operation is `i++`. To further clarify, here is some equivalent `C` code:

```
```

```
for(int i = 0; i != 5; i++) { // <-- loop label
        ...
}
// byebye label
```

`BEQ` is syntactically identical to `B`, with the exception of it's opcode being *one higher* than `B` - `011`. It's conditional branching will depend on a `CMP` instruction being run previously. This result is stored in the **EQ Register** next to the `ALU` within the CPU. It stores the comparison result from the `ALU` for **One extra cycle**, making it accessible by the instruction immediately after `CMP`, which in this case would be `BEQ`. After one instruction, the EQ Register's value is no longer predictable and should be considered random.

The binary representation of `BEQ` would be: `0 011 000 000 000 111`
This will branch you `7` instructions forward if the previous comparison was true.

This concludes the instructions available in Watson. Wahooo!! 🥳

Next up at bat is the ...

Language Format!

Watson's language format is essentially the same as ARM64's format. That being: `[Operation] Register1 Register2 Register3/Immidiate`.
The assembler will trim lines automatically, so feel free to indent your code with tabs, spaces, or whatever you prefer. The parser separates instruction components based on non-ascii values. This leads to the language having a high level of flexability in how it is written.
For example:

```
MOV    x0, 1
ADD    x0, 2
```

is equivalent to:

```
MOV x0 1
ADD x0 2
```

which is equivalent to:

```
MOV🗿x0🛸1
ADD🍷x0👽2
```

All of these instructions will assemble to the same machine code.

-- Comments --

The assembler also supports comments, via use of the `#` character. When parsing, the assembler will ignore all text *after* a `#`. This allows you to have entire lines as comments, as well as in-line comments. You can also use these to temporarily disable lines of code with comments.

For example:

```
#This is a comment
MOV    x0, 0
ADD    x0, 5 #This is also a comment!
STR    x0, x2, 0
#LDR    x0, x2, 1    Believe it or not, also a comment!
# The instruction above will not run!
```

Examples

Here are some example programs to show you more of how this program works!

- You can also find these in `assembler/examples`.

Example 1: Fibonacci numbers!

This program calculates the first 7 digits of the Fibonacci sequence, and stores it in the data memory. Once it is done, it loops itself in the `bye` label to prevent further execution.

```
MOV x0, 0 # a
MOV x1, 1 # b
MOV x2, 0 # c
MOV x4, 0 # iterator
B fib

fibmath:
        ADD x2, x0, x1
        MOV x0, x1
        MOV x1, x2
        B fibstr

fib:
        B fibmath

fibstr:
        STR x2, x4, 0
```

```
18          ADD x4, x4, 1
19          CMP x4, 7
20          BEQ bye
21          B fib
22
23  bye:
24          B bye
```

Example 2: 2+2!

This program adds 2 and 2 to get 4! I am very tired!

```
1   MOV    x0, 2
2   MOV    x1, 2
3   ADD    x2, x0, x1
4
5   MOV    x3, 0
6   STR    x2, x3, 0
```

This concludes the guide to the Watson CPU. Thanks for reading!