# Project: The LS-8 Emulator

## Implementation of the LS-8 Emulator

*Objective*: to gain a deeper understanding of how a CPU functions at a low level.

We're going to write an emulator for the world-famous LambdaSchool-8 computer, otherwise known as LS-8! This is an 8-bit computer with 8-bit memory addressing, which is about as simple as it gets.

An 8 bit CPU is one that only has 8 wires available for addresses (specifying where something is in memory), computations, and instructions. With 8 bits, our CPU has a total of 256 bytes of memory and can only compute values up to 255. The CPU could support 256 instructions, as well, but we won't need them.

For starters, we'll execute code that stores the value 8 in a register, then prints it out:

```
# print8.ls8: Print the number 8 on the screen

10000010 # LDI R0,8
00000000
00001000
01000111 # PRN R0
00000000
00000001 # HLT
```

The binary numeric value on the left in the `print8.ls8` code above is either:

- the machine code value of the instruction (e.g. `10000010` for `LDI`), also known as the *opcode*

or

- one of the opcode's arguments (e.g. `00000000` for `R0` or `00001000` for the value `8`), also known as the *operands*.

This code above requires the implementation of three instructions:

- `LDI`: load "immediate", store a value in a register, or "set this register to this value".
- `PRN`: a pseudo-instruction that prints the numeric value stored in a register.
- `HLT`: halt the CPU and exit the emulator.

See the LS-8 spec for more details.

The above program is already hardcoded into the source file `cpu.py`. To run it, you will eventually:

```
python3 ls8.py
```

but you'll have to implement those three above instructions first!

## Step 0: IMPORTANT: inventory what is here!

- Make a list of files here.
- Write a short 3-10-word description of what each file does.
- Note what has been implemented, and what hasn't.
- Read this whole file.
- Skim the spec.

## Step 1: Add the constructor to `cpu.py`

Add list properties to the `CPU` class to hold 256 bytes of memory and 8 general-purpose registers.

> Hint: you can make a list of a certain number of zeros with this syntax:
>
> ```
> x = [0] * 25  # x is a list of 25 zeroes
> ```

Also add properties for any internal registers you need, e.g. `PC`.

Later on, you might do further initialization here, e.g. setting the initial value of the stack pointer.

## Step 2: Add RAM functions

In `CPU`, add method `ram_read()` and `ram_write()` that access the RAM inside the `CPU` object.

`ram_read()` should accept the address to read and return the value stored there.

`ram_write()` should accept a value to write, and the address to write it to.

> Inside the CPU, there are two internal registers used for memory operations: the *Memory Address Register* (MAR) and the *Memory Data Register* (MDR). The MAR contains the address that is being read or written to. The MDR contains the data that was read or the data to write. You don't need to add the MAR or MDR to your `CPU` class, but they would make handy parameter names for `ram_read()` and `ram_write()`, if you wanted.

We'll make use of these helper function later.

Later on, you might do further initialization here, e.g. setting the initial value of the stack pointer.

## Step 3: Implement the core of `CPU`'s `run()` method

This is the workhorse function of the entire processor. It's the most difficult part to write.

It needs to read the memory address that's stored in register `PC`, and store that result in `IR`, the *Instruction Register*. This can just be a local variable in `run()`.

Some instructions requires up to the next two bytes of data *after* the `PC` in memory to perform operations on. Sometimes the byte value is a register number, other times it's a constant value (in the case of `LDI`). Using `ram_read()`, read the bytes at `PC+1` and `PC+2` from RAM into variables `operand_a` and `operand_b` in case the instruction needs them.

Then, depending on the value of the opcode, perform the actions needed for the instruction per the LS-8 spec. Maybe an `if-elif` cascade...? There are other options, too.

After running code for any particular instruction, the `PC` needs to be updated to point to the next instruction for the next iteration of the loop in `run()`. The number of bytes an instruction uses can be determined from the two high bits (bits 6-7) of the instruction opcode. See the LS-8 spec for details.

## Step 4: Implement the `HLT` instruction handler

Add the `HLT` instruction definition to `cpu.py` so that you can refer to it by name instead of by numeric value.

In `run()` in your if-else block, exit the loop if a `HLT` instruction is encountered, regardless of whether or not there are more lines of code in the LS-8 program you loaded.

We can consider `HLT` to be similar to Python's `exit()` in that we stop whatever we are doing, wherever we are.

## Step 5: Add the `LDI` instruction

This instruction sets a specified register to a specified value.

See the LS-8 spec for the details of what this instructions does and its opcode value.

## Step 6: Add the `PRN` instruction

This is a very similar process to adding `LDI`, but the handler is simpler. See the LS-8 spec.

*At this point, you should be able to run the program and have it print `8` to the console!*

## Step 7: Un-hardcode the machine code

In `cpu.py`, the LS-8 programs you've been running so far have been hardcoded into the source. This isn't particularly user-friendly.

Make changes to `cpu.py` and `ls8.py` so that the program can be specified on the command line like so:

```
python3 ls8.py examples/mult.ls8
```

(The programs `print8.ls8` and `mult.ls8` are provided in the `examples/` directory for your convenience.)

For processing the command line, checkout `sys.argv`. Try running the following Python program with different arguments on the command line to see how it works:

```python
import sys

print(sys.argv)
```

Note that `sys.argv[0]` is the name of the running program itself.

If the user runs `python3 ls8.py examples/mult.ls8`, the values in `sys.argv` will be:

```
sys.argv[0] == "ls8.py"
sys.argv[1] == "examples/mult.ls8"
```

so you can look in `sys.argv[1]` for the name of the file to load.

> Bonus: check to make sure the user has put a command line argument where you expect, and print an error and exit if they didn't.

In `load()`, you will now want to use those command line arguments to open a file, read in its contents line by line, and save appropriate data into RAM.

As you process lines from the file, you should be on the lookout for blank lines (ignore them), and you should ignore everything after a `#`, since that's a comment.

You'll have to convert the binary strings to integer values to store in RAM. The built-in `int()` function can do that when you specify a number base as the second argument:

```
x = int("1010101", 2)  # Convert binary string to integer
```

## Step 8: Implement a Multiply and Print the Result

Extend your LS8 emulator to support the following program:

```
# mult.ls8: Multiply 8x9 and print 72

10000010 # LDI R0,8
00000000
00001000
10000010 # LDI R1,9
00000001
00001001
10100010 # MUL R0,R1
00000000
00000001
01000111 # PRN R0
00000000
00000001 # HLT
```

One you run it with `python3 ls8.py examples/mult.ls8`, you should see:

```
72
```

Check the LS-8 spec for what the `MUL` instruction does.

> Note: `MUL` is the responsiblity of the ALU, so it would be nice if your code eventually called the `alu()` function with appropriate arguments to get the work done.

## Step 9: Beautify your `run()` loop

Do you have a big `if-elif` block in your `cpu_run()` function? Is there a way to better modularize your code? There are plenty of them!

> What is the time complexity of the `if-elif` cascade? In the worst case, we're going to have to check the value in `IR` against all of the possible opcode values. This is `O(n)`. It would be a lot better if it we an `O(1)` process...

One option is to use something called a *branch table* or *dispatch table* to simplify the instruction handler dispatch code. This is a list or dictionary of functions that you can index by opcode value. The upshot is that you fetch the instruction value from RAM, then use that value to look up the handler function in the branch table. Then call it.

Example of a branch table:

```python
OP1 = 0b10101010
OP2 = 0b11110000

class Foo:

    def __init__(self):
        # Set up the branch table
        self.branchtable = {}
        self.branchtable[OP1] = self.handle_op1
        self.branchtable[OP2] = self.handle_op2

    def handle_op1(self, a):
        print("op 1: " + a)

    def handle_op2(self, a):
        print("op 2: " + a)

    def run(self):
        # Example calls into the branch table
        ir = OP1
        self.branchtable[ir]("foo")

        ir = OP2
        self.branchtable[ir]("bar")

c = Foo()
c.run()
```

## Step 10: Implement System Stack

All CPUs manage a *stack* that can be used to store information temporarily. This stack resides in main memory and typically starts at the top of memory (at a high address) and grows *downward* as things are pushed on. The LS-8 is no exception to this.

Implement a system stack per the spec. Add `PUSH` and `POP` instructions. Read the beginning of the spec to see which register is the stack pointer.

- Values themselves should be saved in the ***portion of RAM*** *that is allocated for the stack*.
  - Use the stack pointer to modify the correct block of memory.
  - Make sure you update the stack pointer appropriately as you `PUSH` and `POP` items to and from the stack.

If you run `python3 ls8.py examples/stack.ls8` you should see the output:

```
2
4
1
```

## Step 11: Implement Subroutine Calls

Back in the old days, functions were called *subroutines*. In machine code, subroutines enable you to jump to any address with the `CALL` instruction, and then return back to where you called from with the `RET` instruction. This enables you to create reusable functions.

Subroutines have many similarities to functions in higher-level languages. Just as a function in C, JavaScript or Python will jump from the function call, to its definition, and then return back to the line of code following the call, subroutines will also allow us to execute instructions non-sequentially.

The stack is used to hold the return address used by `RET`, so you **must** implement the stack in step 10, first. Then, add subroutine instructions `CALL` and `RET`.

- For `CALL`, you will likely have to modify your handler call in `cpu_run()`. The problem is that some instructions want to execute and move to the next instruction like normal, but others, like `CALL` and `JMP` want to go to a specific address.

  > Note: `CALL` is very similar to the `JMP` instruction. However, there is one key difference between them. Can you find it in the specs?

  - In **any** case where the instruction handler sets the `PC` directly, you *don't* want to advance the PC to the next instruction. So you'll have to set up a special case for those types of instructions. This can be a flag you explicitly set per-instruction... but can also be computed from the value in `IR`. Check out the spec for more.

If you run `python3 ls8.py examples/call.ls8` you should see the output:

```
20
30
```

```
36
60
```

## Stretch Goal: Timer Interrupts

Add interrupts to the LS-8 emulator.

**You must have implemented a CPU stack before doing this.**

**You must have implmented the `ST` instruction before doing this.**

See the LS-8 spec for details on implementation.

The LS-8 should fire a timer interrupt one time per second. This could be implemented by calling `datetime.now()` (in the `datetime` module) each iteration of the main loop and checking to see if one second has elapsed.

When the timer is ready to fire, set bit 0 of the IS register (R6).

Later in the main instruction loop, you'll check to see if bit 0 of the IS register is set, and if it is, you'll push the registers on the stack, look up the interrupt handler address in the interrupt vector table at address `0xF8`, and set the PC to it. Execution continues in the interrupt handler.

Then when an `IRET` instruction is found, the registers and PC are popped off the stack and execution continues normally.

## Example

This code prints out the letter `A` from the timer interrupt handler that fires once per second.

```
# interrupts.ls8

10000010 # LDI R0,0XF8
00000000
11111000
10000010 # LDI R1,INTHANDLER
00000001
00010001
10000100 # ST R0,R1
00000000
00000001
10000010 # LDI R5,1
00000101
00000001
10000010 # LDI R0,LOOP
00000000
00001111

# LOOP (address 15):
01010100 # JMP R0
00000000
```

```
# Timer interrupt Handler
# When the timer interrupt occurs, output an 'A'
# INTHANDLER (address 17):
10000010 # LDI R0,65
00000000
01000001
01001000 # PRA R0
00000000
00010011 # IRET
```

The assembly program is interested in getting timer interrupts, so it sets the IM register to `00000001` with `LDI R5,1`.

The interrupt timer gets to 1 second, and sets bit #0 in IS.

At the beginning of each `cpu_run()` loop, the CPU checks to see if interrupts are enabled. If not, it continues processing instructions as normal. Otherwise:

Bitwise-AND the IM register with the IS register. This masks out all the interrupts we're not interested in, leaving the ones we are interested in:

```
masked_interrupts = cpu.reg[IM] & cpu.reg[IS]
```

Step through each bit of `masked_interrupts` and see which interrupts are set.

```
for i in range(8):
  # Right shift interrupts down by i, then mask with 1 to see if that bit was set
  interrupt_happened = ((masked_interrupts >> i) & 1) == 1

  # ...
```

(If the no interrupt bits are set, then stop processing interrupts and continue executing the current instruction as per usual.)

If `interrupt_happened`, check the LS-8 spec for details on what to do.

## Stretch Goal: Keyboard Interrupts

This gets tricky because you have to see if a key has been pressed without stopping the program from running otherwise. The easiest way to do this is with *polling*. ("Was a key hit? What about now? What about now?")

Google for `python keyboard poll` to get some ideas on how to do this. Windows does it differently than Unix/Mac.

## Stretch Goal: Curve Histogram

Write an LS-8 assembly program that prints this curve on the screen:

```
*
**
****
********
****************
********************************
****************************************************************
```

Each subsequent line has two-times the number of asterisks as the previous line.

**Use loops to get this done.**

Doing this correctly requires implementing `CMP`, and some comparative forms of `JMP`, such as `JLT` or `JNE` or `JEQ`.

Hint: Look in the `asm/` directory and learn how to use the `asm.js` assembler. This way you can write your code in assembly language and use the assembler to build it to machine code and then run it on your emulator.