Turing Approved - Writeup

> Turing completeness means that I can do everything, right?

Scenario

In this challenge we are provided with the following:

- a `Makefile`
- a textual file called RAM, composed of comma-separated binary digits
- some LaTeX files: `interpreted.tex` and `execution.sty`

Reading `Makefile`

We observe that `make run` calls this line:

```
pdflatex --shell-escape interpreted.tex > /dev/null
```

This is peculiar since it uses a particular option, `--shell-escape`, useful to execute commands in the terminal using the command `\write18`. We'll keep our eyes on it in the LaTeX code.

Reading `interpreted.tex`

Analyzing `interpreted.tex` we discover the following:

- The author left us a very encouraging message;
- The title of the document is "Emulation of a CPU using LaTeX";
- Three commands are called, probably from the `execution.sty` package:
  - `\setup`
  - `\fetchdecodeexecwriteback`
  - `\printCPU`

So yes, definitely we are emulating a CPU with this code. Time to read `execution.sty`!

Reading `execution.sty`

The code is *extremely readable*. Just kidding - it's not. However, we can see the implementation of various commands that we encountered while reading `interpreted.tex`.

- `\setup` does exactly what it says. It creates various counters and then fills some of them using the custom command `\lr` which, according to the descriptions in the code, loads the `RAM` file into the "emulated memory".
- `\fetchdecodeexecwriteback` loads some binary digits from the "memory" into the "IR" (Instruction Register), and then examines it bit-by-bit in order to discover which OPCODE is it, then calling a command that logs the name of the OPCODE, another that parses it and sometimes `\nxti` that increases the Program Counter (PC).
- `\printCPU` prints infos about registers and so on.

We also see that it uses `\write18` commands to output on stderr, in order to avoid the redirect to `/dev/null`.

Executing it

Running `make run` results in the following text:

```
echo "" > DUMP
pdflatex --shell-escape interpreted.tex > /dev/null # runs our "compilation"/"emulation"

LC-2>   Setting up everything, please be patient...

LC-2>   Ready to read memory...

LC-2>   RAM loaded into emulated memory...

[ A LOT OF TABLES DEPICTING THE DATA IN THE REGISTERS ]

tr -d '\n' < DUMP

[ A HUGE DUMP OF BINARY DIGITS, PROBABLY THE RAM AFTER THE EXECUTION ]
```

```
rm interpreted.aux interpreted.log interpreted.output
```

After a bit of research we discover that LC-2 is a CPU described in a book about computer architecture. No, not LC-II, but LC-2, or "Little Computer 2".

Digging a bit more, we find the specs of the ISA, for example ***here***.

At this point we finally have all what we need to read the RAM - or the memory DUMP. We'll read the RAM since it is much easier and faster.

Reading the RAM

A simple python script that does the minimum necessary to understand the RAM file might be the following (`parseRAM.py`):

```python
# read file content
with open("RAM") as f:
    RAM = f.read()
f.close()
# get single instructions
RAM = [''.join(RAM.split(","))[16*i:16*(i+1)] for i in range(len(RAM.split(","))//16)][1:]
# the OPCODES we know
translations = {
    "0001": "ADD  ",
    "0101": "AND  ",
    "0000": "BR   ",
    "0100": "JSR  ",
    "1100": "JSRR ",
    "0010": "LD   ",
    "1010": "LDI  ",
    "0110": "LDR  ",
    "1110": "LEA  ",
    "1001": "NOT  ",
    "1101": "RET  ",
    "1000": "RTI  ",
    "0011": "ST   ",
    "1011": "STI  ",
    "0111": "STR  ",
    "1111": "TRAP "
}
# making it more readable
RAM = '\n'.join([translations[i[0:4]]+i[4:] for i in RAM])
# printing it
print(RAM)
```

We obtain an output like this:

```
BR   111000000010
BR   000000100000
LD   010000000001
LD   001001010010
ADD  000001000010
ST   000001010010
TRAP 000000100101
BR   111011000001
BR   000000000110
BR   000000101001
LD   010000100100
LD   001100110110
ADD  000001000010
ST   000100110110
TRAP 000000100101
BR   111000011110
BR   000001011010
```

```
BR   000000010110
...
```

An analysis of this code gives us a basic understanding of the inner working of this program: it is composed of various parts that follow the same pattern.

```
LD    R2, something          ; loads the value at something into R2
LD    R1, something else     ; loads the value at something else into R1
ADD   R0, R1, R2             ; adds R1 and R2, storing the value in R0
ST    R0, something          ; stores the value of R0 at something
TRAP  x25                    ; halts the execution (!!!)
BRNZP somewhere              ; jumps to the next part
```

These parts repeats, and are interspersed with random instructions that don't appear to make much sense, and we see that their addresses are referenced in the `something`s and `something else`s.

The problem in the code above, that prevents it to run, is the presence of `TRAP x25` instructions: the execution is stopped at each block. There are at least three ways to overcome this limitation.

1. Write a simple interpreter to run the "RAM"
2. Remove them from the RAM by "patching the binary".
3. Edit the LaTeX emulator.

We'll focus on the first solution.

Writing a new interpreter

We can extend the `parseRAM.py` script to provide limited simulation capability:

```
# this is the minimal execution state that we must keep with the soruce code we know
state = {
    "PC": 0,
    "registers": [0,0,0,0,0,0,0,0],
    "RAM" : None
}

def readRAM():
    global state
    # read file content
    with open("RAM") as f:
        state["RAM"] = f.read()
    f.close()
    # get single instructions
    state["RAM"] = [''.join(state["RAM"].split(","))[16*i:16*(i+1)] for i in range(len(state["RAM"].split(","))/

def simulate():
    global state
    # opcodes
    translations = {
        "0001": add,
        "0101": unused,
        "0000": jump,
        "0100": unused,
        "1100": unused,
        "0010": load,
        "1010": unused,
        "0110": unused,
        "1110": unused,
        "1001": unused,
        "1101": unused,
        "1000": unused,
        "0011": store,
        "1011": unused,
        "0111": unused,
        "1111": trap
```

```
    }
    while state["PC"] != len(state["RAM"]):
        curr_instruction = state["RAM"][state["PC"]]
        translations[curr_instruction[:4]](curr_instruction[4:])
    print()

def unused(data):
    # this instruction is unused
    pass

def add(data):
    # the sum is always R0, R1, R2
    global state
    state["registers"][0] = state["registers"][1] + state["registers"][2]
    # we print it immediately to avoid reading the memory dump
    print(chr(state["registers"][0]),end='')
    # next instruction
    state["PC"] += 1

def jump(data):
    global state
    addr = int(data[3:],2)
    if addr:
        state["PC"] = addr
    else:
        state["PC"] += 1

def load(data):
    global state
    DR = int(data[:3],2)
    addr = int(data[3:],2)
    state["registers"][DR] = int(state["RAM"][addr],2)
    state["PC"] += 1

def store(data):
    # we don't really need to implement this one
    global state
    state["PC"] += 1

def trap(data):
    # the TRAP instruction is creating us a lot of problems, so lets just skip it
    global state
    state["PC"] += 1

def print_state():
    global state
    print("PC: {PC} R0: {R0} R1: {R1} R2: {R2}".format(PC=state["PC"],R0=state["registers"][0],R1=state["registe

if __name__ == "__main__":
    readRAM()  # setup
    simulate() # run
```

When we run the script, we obtain the following results:

```
$ python3 parseRAM.py
K1nd4SUS{morality_is_not_turing_complete_42424242}
```

We found the flag!