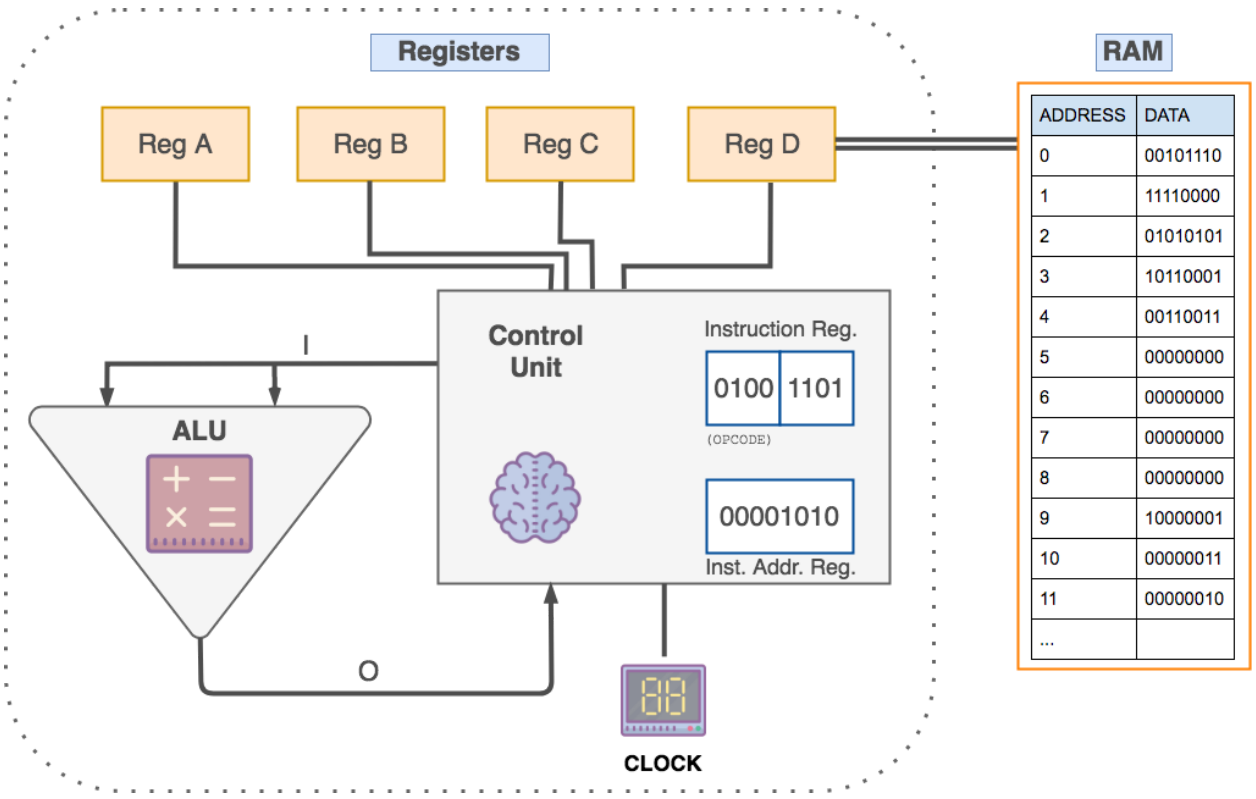
CPU simulator

Information

A CPU (control process unit) is conformed in the most simple way of:



CU:

A Control Unit is responsible of fetching the instruction set, controlling the data flow across the whole CPU, controls the timing of each operation and the interaction with peripheral devices.

ALU:

Arithmetic Logic Unit, is a digital electronic circuit responsible for (depending on the processor) doing all the arithmetic operations such as (but not limited to):

- Add
- Subtraction
- Subtract with borrow

- One’s complement
- Two’s complement
- AND
- OR
- Bit shift operations
- > < <> >=

Registers

An important piece of memory that the CU and the ALU can store temporarily the data, one important register is the program counter which keeps track of where the CPU is reading from the instruction set.

- Instruction Register (Current instruction loaded)
- Instruction Address Register (program counter)

RAM

Random Access memory:

- DATA
- INSTRUCTIONS

Clock

In charge of control the fetch-decode-execute cycle.

Instruction Set **Table**

OP CODE (4bit)	Instruction (mnemonic)	Description	Address Or Regs (operands)
0000	OUTPUT, OUT	Wires OUTPUT register directly to RAM data address location (writes to console)	4-bit RAM data address
0001	LOAD_R0, LD_R0	Reads RAM location into register R0	4-bit RAM data address
0010	LOAD_R1, LD_R1	Reads RAM location into register R1	4-bit RAM data address
0011	AND	Performs **AND** between 2-bit registers ID	2 bit register ID
0100	ILD_R0	Immediate Read constant into register R0	4-bit constant
0101	STR_R0, STORE_R0	Write from Register R0 into RAM location	4-bit RAM data address
0110	STR_R1, STORE_R1	Write from Register R1 into RAM location	4-bit RAM data address
0111	OR	Performs **OR** between 2-bit registers ID values	2 bit register ID
1000	ILD_R1	Immediate Read constant into register R1	4-bit constant
1001	ADD	Add two registers, store result into second register	2 bit registers ID
1010	SUB	Subtract two registers, store result into second register	2 bit register ID
1011	JMP, JUMP	update Inst. Addr. Reg to new address	4-bit code address
1100	JMP_N, JUMP_NEG	IF **ALU** result was negative , update Inst. Addr. Reg to new address	4-bit code address
1101	*	*	*
1110	*	*	*
1111	HALT, HLT	Program done. Halt computer	NA

* Developer will create and define his own operation and addressing mode. See Phase two.3

Project Description

We will create a CPU simulator, that will read a program and it will execute it using what we learned in the course. We will do a Register file machine 4-bit CPU with 16 bytes of RAM and 4 registers + extra registries, one ALU.

Phase one:

1. We will read the program from a ".cpufm" file, which will contain the program in machine code. (Code will be loaded to our RAM.insutrctions),
2. Comments in .cpufm files will start with semicolon char **;**, any **line** starting with it will be ignored by your parser.
3. Instructions will be in "1100 0011" string format. But ints[1] in our Memory objects.
4. We will "wire" the output of our program to a Special register, OReg (output register), which indeed is our console, so whenever you want to print you will do something like 'OR.out("")'
5. We will fetch-decode-execute each instruction as fast as we can.
6. It is required that you implement the following Classes (be creative and intuitive for the object attributes and methods):
    a. **IC** (integrated circuit): parent class (examples: manufacturer, build-date, purpose, etc we already built something similar in class)
    b. **Memory**: parent class, (but inherits from IC), remember it is a 4 bit CPU
    c. **ALU** (inherits from IC), MUST have 3 attributes **Z**ero[2], **O**verflow, **N**egative, but not limited to those three. OP-Code, Input, and it **MUST** have all the arithmetic /logic operations supported as functions.
    d. **CU** (inherits from IC) is the main controller for everything, it is where everything is orchestrated. It should be the one responsible for Fetch, Decode , Execute (send instruction decoded to ALU if needed).

[1] You will probably need a "radixconvert"
[2] Store the result of the operations, Zero: if the result is 0; Overflow: if the result is bigger than X; Negative: if the result was < 0

e. **Register** (parent class, but inherits from IC) implement general purposes methods and attributes. (DRY=dont repeat yourself)
   i. fields: data
f. **Registers**: R0,R1,R2,R3,PC (IAR), IR, OR
g. **RAM:**
   i. fields: data & instructions
   ii. stores: RAM (scratchboard 16 bytes) & code (cpufm instructions)
h. **ROM:**
   i. **fields:** bios & instructions set table
   ii. **stores**: bios.yaml & instruction set table.

Phase two

1. Separate in **modules**. Every class can be a module, a set of functions, etc.
2. The initial configuration will be read from a **bios.yml (yaml files) and load into ROM.bios**
3. Create and define your own 2 new operations and addressing modes. Instruction 13 and 14. Please be creative.
4. We will add a **"Clock"** class that we will attach its frequency (~0Hz to 2Hz) to the CU and will determine the frequency at which we do each instruction-cycle. `clock: 0.5`
   a. Remember that 0.5 Hz => 2 seconds
   b. `Fetch [Sleep (1/Freq) seconds]`[3] `=> Decode [Sleep (1/Freq) seconds] => Execute [Sleep (1/Freq) seconds] =>` Repeat
5. Bios.yml RAM.data array, ability to Load from bios.yml a 16 position array with pre filled values in order to test your code. `RAM: [1,2, 15, 0 , 4, …… ]`
6. Bios.yml RAM.instructions, ability to load the program itself in binary or decimal mode (unlimited amount of lines)
7. We will read now code in mnemonic (0001 = LOAD_R0) as well as machine code. So our .cpufm files can be a mix of machine code or mnemonic
   a. Note that some instructions support multiple mnemonics
8. We will add a "Step" debug mode, which will let the user press Enter to continue to the next instruction. `clock: 0`

[3] Sleep in between just for demo/display/readability purposes.

Phase three

1. Pretty print (console table? Graphic? UI? Be creative) the current state of our CPU, print out:
    a. Registers
    b. RAM
    c. Current Clock speed
    d. ALU bit flags
    e. Take into account the Radix in the bios.yml, it will define if the numbers will be displayed in dec, bin or hex format, defaults dec.

Phase four

1. Create 7 example .cpufm files that do something different than the ones below. Be sure to mix mnemonic and machine code as well.

Here is an example of bios.yml

Additional Info

- Use OOP principles in the whole project.
- Try to split responsibilities across the team
- If any operation returns negative we will turn on Negative flag in ALU
- Overflow if a value is greater than ? and it cannot be stored in our 4 bit registers
- And if any operation is Zero turn on Zero flag in ALU
- Project in a personal/organizational git repository (recommended are: **Gitlab**, **Bitbucket**, it already includes pipelines or **Github** + Travis or Circle CI)
- All team members have commits (in history) and/or merge/pull requests.
- We will not accept any project that commits from yesterday or three days before yesterday.

Extra Points:

- Project has a README.md.
- It has CI.
- Use pipenv

- Put the project in Docker.
- Any graphical enhancements.
- Use Functions as FCO

Delivery Format:

- Git repository
- Create a **git tag** (release) and send the repo URL and git tag ID. (preferable)

Appendix

Radix

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Example programs:

Simple Add (with halt):

RAM:

...	
14	**3**
15	**14**

Program:

```
; ---- Simple ADD ----
; LOAD RAM 14 to REG A
LOAD_R0 14
; THIS IS A COMMENT
LD_R1 15
; THIS IS ANOTHER COMMENT
ADD R1 R0
STORE_R0 13
OUTPUT 13
HLT
```

Loop with jump:

RAM

...	
13	3
14	1
15	1

Program:

```
; ---- LOOP with JUMP if ----
LOAD_R0 14
LOAD_R1 15
ADD R1 R0
STORE_R0 13
; SEE HOW WE CHANGE PROGRAM COUNTER (IAR)
JUMP_NEG 2
HALT
```

Loop (with JUMP_NEG | divide):

RAM:

...	
14	**11**
15	**5**

Program:

```
; ---- DIVIDE ;) ----
LOAD_R0 14
LOAD_R1 15
SUB R1 R0
; JUMP IF NEGATIVE
JUMP_NEG 5
JUMP 2
ADD R1 R0
STORE_R0 13
OUTPUT 13
HALT
```