This assignment will be closed on June 23, 2020 (23:59:59).

You must be authenticated to submit your files

CSE 102 – Project – Core War

Core War

0.03 / 0

The Core War game appeared in article by Alexander Dewdney published in May 1984 in the Scientic American. The principle of this game is extremely simple: two or more computer programs are loaded into the memory of a virtual machine (i.e. of a program that simulates a computer) and are executed concurrently, one instruction at a time. The goal of a process is to be the last one alive. In order to do so, a program has to do everything it can to eliminate the opposing processes. The whole point of the game comes from the fact that a program can write anywhere in memory and can therefore cause the elimination of other players by writing an illegal instruction in the code of the latter. A program can also move and duplicate itself in the memory, to escape from its opponent.

Virtual machine description

Memory

The virtual machine is composed of a 32-bit word memory of size 4096. Each 32-bit word in memory possesses an address that corresponds to its index in the memory seen as a flat array. I.e., the first word of the memory has address `0`, the second one has address `1`, etc... up to the last word that has address `4095`.

Program state

Each player runs not one but several programs, called here processes. Each process owns a state that is composed of:

- 16 32-bit registers (`r0` to `r15`),
- a circular stack of 16 frames of 32 bits each,
- a program counter `PC` (12 bits, unsigned)
- a zero flag `Z`

The player's processes are stored in a queue. Initially, all players' queues are initialised with a single process each, whose state is set as follows:

- all the registers are set to `0`,
- all the frames of the stack are set to `0`,
- the `PC` is set to the location where the player code was loaded, and
- the zero flag `Z` is set to `0`.

During the execution of the programs, some processes might die (because they executed an illegal instruction or executed the `DIE` instruction). In that case, the process is removed from the player's queue. A player is said to be alive if her processes queue is non-empty, otherwise, she is said to be dead and cannot play anymore.

If p represents a process state, then we write $p.\mathrm{r}_i$ for its i-th register, p.stack for its stack, p.PC for its program counter and p.Z for its zero flag.

Execution of programs

Programs execution are concurrent and synchronised. At each turn, the first process of each alive player is dequeued from the processes queues of the players. Let's call $p_1, \ldots, p_n$ these processes. Note that n might be strictly smaller than the number of players if there are dead players.

The processes p_i are then executed in isolation, i.e.:

- the word at address p_i.PC is decoded as an instruction. If the decoding of the instruction fails, the process dies and its execution stops.
- the decoded instruction is then executed and the process state is updated accordingly. However, the instruction is executed in isolation: all memory updates are recorded but are not yet visible to the other concurrent processes. Note that the process might die at this point if it executes the `DIE` instruction.
- if the process is still alive at this point, then it is enqueued in the processes queues of the payers it belongs to.

0.03 / 0

When all the concurrent processes have been executed, all the memory updates of the still alive processes are committed in memory, i.e. they are written in memory and will be visible to the next processes. However, we might have concurrent writes here. The next section explains how concurrent writes are resolved.

Concurrent writes resolution

When committing the processes' memory updates, it may happen that two or more processes want to write at the same memory location. In that case, a voting system will take place. For each bit of the target word, if the number of processes that want to write the bit `b` is greater than the number of processes that want to write the bit `1-b`, then the bit `b` is written. If there is a tie, then the bit is left unchanged.

End of a game

A game ends when there is at most one player alive. The winning player is the last player alive, if any. Otherwise, it's a tie.

However, programs may loop and the virtual machine is going to stop after a maximum number of turns. In that case, we also have a tie.

Instructions

Operands

The processes have access to 16 different instructions. Some of the instructions take one or two parameters, called operands, that are encoded as 12-bit words. However, depending on what we call an *addressing mode*, these operands might denote literals (i.e. integer constants), registers or memory locations.

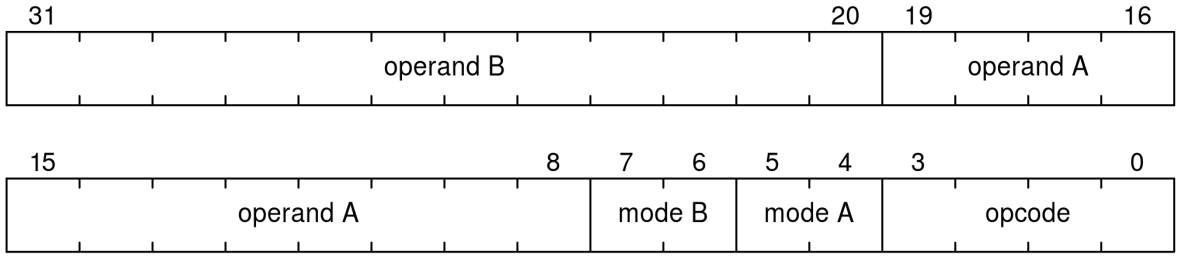
Addressing modes

The virtual machine has 4 addressing modes for its operands. Assume that the operand is encoded using the 12-bit word w. Depending on the addressing mode, w will be interpreted as:

- **immediate**: in that mode, the operand is interpreted as a 2-complement signed 12-bit integer, signed extended to a 32-bit word – i.e. you interpret w as a 12-bit 2-complement signed integer and you encode that signed value as a 32-bit word, again using 2-complement.
- **relative**: in this mode, the operand is interpreted as the 32-bit word in memory at location w (interpreted as a 2-complement signed integer) relatively to the program counter (i.e. the memory cell at address p.PC + signed[w]).
- **computed**: in this mode, the operand is interpreted as the 32-bit word in memory at location l (relatively to the program counter), where l is obtained from interpreting, as a 2-complement 12-bit signed integer, the 12 lowest bits of the memory at location w (relatively to the program counter and where w is interpreted as a 2-complement signed integer),
- **register**: in this mode, w stands for the register number w `& 0b1111` - i.e. we only consider the 4 least significant bits of w.

Encoding of instructions

All instructions are encoded over 32-bit words with the following layout.



Addressing mode

The addressing mode is encoded over 2 bits:

encoding	mode
0b00	immediate
0b01	relative

encoding	mode
0b10	computed
0b11	register

Opcode

The instructions codes are the following (their meaning is given below):

Encoding	Instruction
0b0000	FORK
0b0001	MOV
0b0010	NOT
0b0011	AND
0b0100	OR
0b0101	LS
0b0110	AS
0b0111	ADD
0b1000	SUB
0b1001	CMP
0b1010	LT
0b1011	POP
0b1100	PUSH
0b1101	JMP
0b1110	BZ
0b1111	DIE

Program bytecode

A program is represented by its bytecode, i.e. by an array (or list) of 32-bit words that encode instructions.

Program loading

When loading programs in memory, the virtual machine simply writes their respective bytecode in memory. The location where the n-th program is loaded (the first program is numbered `0`) is $n\lfloor\frac{4096}{K}\rfloor$, where K is the number of programs to be loaded.

If the programs overlapped in memory, then the virtual machine stops immediately.

Instructions semantic

The virtual machine supports 16 instructions, that we describe here. In all instructions `A` denotes the first operand whereas `B` denotes the second one. Note that immediate operands cannot be assigned - such an instruction should be considered as illegal.

`FORK`

This instruction forks the executing process p, i.e. it enqueues a new process in the processes queues from which p originates. Note that the newly created process is enqueued *before* the process p is (re)enqueued itself.

	`FORK`
Description	create a new process
A	*not used*
B	*not used*

0.03 / 0

	FORK
Modifies	`Z`
Z	`Z` ← `0`
PC	`PC` ← `PC + 1`

The new process state is the following:

	value
r_i	$p.r_i$
stack	p.stack
A	*not used*
B	*not used*
Modifies	`Z`
Z	`Z` ← `1`
PC	`PC` ← p.PC + `1`

MOV

Copy the contents of `A` in `B`.

	MOV
Description	`B` ← `A`
A	uninterpreted (32-bit word)
B	*not used*
Modifies	`B`
Z	unchanged
PC	`PC` ← `PC + 1`

ADD

Compute the addition (signed or unsigned) of the two operands and store the result in the second one. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

	ADD
Description	`B` ← `A + B`
A	signed or unsigned
B	signed or unsigned (same as `A`)
Modifies	`B`, `Z`
Z	`Z` ← `1` if `B` is `0` else `0`
PC	`PC` ← `PC + 1`

SUB

Compute the substraction (signed or unsigned) of the two operands and store the result in the second one. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

	SUB
Description	`B` ← `A - B`
A	signed or unsigned

0.03 / 0

	SUB
B	signed or unsigned (same as `A`)
Modifies	`B`, `Z`
Z	`Z` ← `1` if `B` is `0` else `0`
PC	`PC` ← `PC + 1`

NOT

Compute the bitwise `not` of the first operand and store the result in the second one. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

	NOT
Description	`B` ← ~ `A`
A	uninterpreted (32-bit word)
B	*not used*
Modifies	`B`, `Z`
Z	`Z` ← `1` if `B` is `0` else `0`
PC	`PC` ← `PC + 1`

AND

Compute the bitwise `and` of the two operands and store the result in the second one. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

	AND
Description	`B` ← `A` & `B`
A	uninterpreted (32-bit word)
B	uninterpreted (32-bit word)
Modifies	`B`, `Z`
Z	`Z` ← `1` if `B` is `0` else `0`
PC	`PC` ← `PC + 1`

OR

Compute the bitwise `or` of the two operands and store the result in the second one. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

	OR
Description	`B` ← `A` | `B`
A	uninterpreted (32-bit word)
B	uninterpreted (32-bit word)
Modifies	`B`, `Z`
Z	`Z` ← `1` if `B` is `0` else `0`
PC	`PC` ← `PC + 1`

LS

Compute the logical right shift of the second operand. The first operand (interpreted as a signed integer) gives the number of bits to shift. A negative first operand indicates a left shift. The result is stored in the second operand. The `Z` flag is set to `1` if the result is `0`, it is set to `0` otherwise.

0.03 / 0

	LS
Description	`B ← B >>> signed[A]`
A	signed
B	uninterpreted (32-bit word)
Modifies	`B , Z`
Z	`Z ← 1 if B is 0 else 0`
PC	`PC ← PC + 1`

AS

Compute the arithmetic right shift of the second operand. The first operand (interpreted as a signed integer) gives the number of bits to shift. A negative first operand indicates a left shift. The result is stored in the second operand. The `Z` flag is set to `1` if the result is `0` , it is set to `0` otherwise.

	AS
Description	`B ← B >> signed[A]`
A	signed
B	uninterpreted (32-bit word)
Modifies	`B , Z`
Z	`Z ← 1 if B is 0 else 0`
PC	`PC ← PC + 1`

CMP

Compare the two operands and store the result in the `Z` flag.

	CMP
Description	`Z ← A = B`
A	uninterpreted (32-bit word)
B	uninterpreted (32-bit word)
Modifies	`Z`
Z	`Z ← 1 if A = B else 0`
PC	`PC ← PC + 1`

LT

Check if the first operand is strictly smaller than the second one (interpreted as signed integers) and store the result in the `Z` flag.

	LT
Description	`Z ← signed[A] < signed[B]`
A	signed integers
B	signed integers
Modifies	`Z`
Z	`Z ← 1 if signed[A] < signed[B] else 0`
PC	`PC ← PC + 1`

POP

Pop a value from the processor stack and stores it in the first operand. The second operator is ignored.

	POP
Description	`A ← pop()`
A	*not used*
B	*not used*
Modifies	`A`
Z	unchanged
PC	`PC ← PC + 1`

PUSH

Push the operand `A` into the processor stack. The second operator is ignored.

	PUSH
Description	`push(A)`
A	uninterpreted (32-bit word)
B	*not used*
Modifies	*nothing*
Z	unchanged
PC	`PC ← PC + 1`

JMP

Jump to the address designated by `A`, relatively to the `PC`. The second operand is ignored and the `Z` is left unchanged.

	JMP
Description	Jump to `PC + signed[A]`
A	address offset (signed)
B	*not used*
Modifies	*nothing*
Z	unchanged
PC	`PC ← PC + A`

BZ

If `Z` is set, jump to the address designated by `A`, relatively to the `PC`. The second operand is ignored and the `Z` is left unchanged.

	BZ
Description	Jump to `PC + signed[A]` if `Z` is set
A	address offset (signed)
B	*not used*
Modifies	*nothing*
Z	unchanged
PC	`PC ← PC + signed[A]` if `Z` is set
	`PC ← PC + 1` otherwise

DIE

Kills the current process. After the instruction, the process is removed from the program queue.

0.03 / 0

Assembly language

Programs are written in an assembler-type source language: a source program consists of lines that represent operations. A comment might be introduced by the character `;` (all the text after the `;` character is ignored, including the `;` -character)

An operation is a line containing an optional label, an instruction name, an operand and an optional second operand

```
operation    ::= (label ':')? instruction operand operand? comment? '\n'
label        ::= &<a-z>+
instructions ::= FORK | MOV | NOT | AND | OR | LS | AS | ADD | SUB | CMP | LT | POP | PUSH | JMP | BZ |
DIE
comment      ::= ; <text>
operand      ::= mode? value
mode         ::= $ | @ | # | r
value        ::= integer | label
integer      ::= -?<0-9>+ | 0b<0-1>+
```

A label is an identifier, consisting of alphabetic characters. The 16 instructions are written in capital letters. An operand is an optional mode followed by a value. An addressing mode is indicated by the character `$` (immediate), `@` (relative), `#` (computed) or `r` (register). No mode means relative mode. A value is either a signed decimal value, an unsigned binary value or a label.

Labels are a facility to write addresses (always relative to the current instruction) in a symbolic way. The labels of two different instructions must be different. A label appearing in an operand must be the label of a program instruction. The value of a label in an operand is the distance relative to the instruction bearing this label.

Work to be done

We ask you to write an interpreter `run.py` for the virtual machine. Your interpreter should take as first argument the maximum number of turns, and then the filenames that contain the players programs (as bytecode). Then, it should load the programs into memory and execute them until the end of the game.

When the program ends, the program should write `TIE` if there is a tie, or `WINNER = n` if player `n` won (players are numbered from `0`).

E.g.

```
python run.py 1024 program1.cor program2.cor
```

loads the programs `program1.cor` and `program2.cor` into memory and execute them, stopping after a maximum of `1024` turns.

If you work in pair

In that case, we also ask you to write a program `asm.py` that takes as input a program written in the assembly language and that compiles it into bytecode.

E.g.

```
python asm.py program.asm program.cor
```

compiles the contents of `program.asm` and write the result into `program.cor`.

Detailed workplan (virtual machine)

We provide here a detailed plan for implementing the virtual machine. If you follow these steps one-by-one, you will eventually obtain a working final program.

0.03 / 0

Upload form is only available when connected

Cyclic stack

The first thing you are going to implement is the cyclic stack. For that purpose, you will create a class `CyclicStack` with the following methods:

- `.__init__(self, size)` creates a cyclic stack of size `size`. The stack is initialized with `0`s.
- `.__len__(self)` that returns the size of the memory.
- `.pop(self)` that pops an element from the stack.
- `.push(self, n)` that pushed the value `n` on top of the stack.

The stack being cyclic, none of these functions can fail: the stack is simply composed of an array of size `size` and a top-stack index that indicates the next cell to be written. When pushing a value, the pointed cell is updated and the top-stack index is incremented. Likewise, when popping a value, the top-stack index is decremented and the contents of the top-stack cell is returned. Since the stack is cyclic, all the arithmetic of the top-stack index is done modulo `size`.

Extracting sub-words

In this implementation, we are going to represent words using Python non-negative integers: a n-bit word is going to be represented as an integer in the range $[0..2^n)$. We have seen in Lecture 7 how one can extract bits from them. For starting, we are going to write a function for extracting a range of bits (or a sub-word) from a word:

- Write a function `extract(w, m, n)` that takes a word `w` (i.e. a non-negative integer) and that returns the subword $w_{m+n-1}w_{m+n-2}\dots w_{m+1}w_m$ of `w` – where w_i is the ith least significant bit of w.

This function must not use string manipulations.

From words to signed/unsigned integers

As we wrote, in this implementation, we are going to represent words using Python non-negative integers: a n-bit word is going to be represented as an integer in the range $[0..2^n)$. A word can be interpreted in different ways. For instance, we can interpret a word as an unsigned integer (in that case, the word value and its interpretation are equal) or as a signed integer (using the 2-complement encoding).

We remind you that, in 2-complement, the word $w = b_{n-1}\dots b_1b_0$ is interpreted as:

- $\sum_i 2^i * b_i$ if the MSB b_{n-1} is 0,
- $(\sum_i 2^i * b_i) - 2^n$ if the MSB b_{n-1} is 1.

(Note that $\sum_i 2^i * b_i$ is simply the value of w as an unsigned integer).

We ask you to:

- write a function `to_signed(w, n)` that takes a `n`-bit word `w` (as an integer in the range $[0..2^n)$) and that returns its interpretation as a `n`-bit signed integer using the 2-complement encoding - the returned integer should hence be in the range $[-2^{n-1}..2^{n-1})$.
- write a function `of_signed(i, n)` that takes an integer in the range $[-2^{n-1}..2^{n-1})$ and that encodes it using 2-complement in a `n`-bit word - the returned integer should hence be in the range $[0..2^n)$.

These functions must not use string manipulations.

Decoding instructions

We have seen that all instructions, along with their operands, are encoded over a single 32-bit word. In this question, we want to implement the decoding of an instruction:

0.03 / 0

- write a function `idecode(w)` that takes a 32-bit word representing an instruction and that decodes it. The function should return a triple of the form `(opcode, (modeA, operandA), (modeB, operandB))` where
  - `opcode` is the instruction opcode,
  - `(modeA, operandA)` is the instruction first operand - it is a pair composed of the operand addressing mode and the operand value itself.
  - `(modeB, operandB)` is the instruction second operand - here again, it is a pair composed of the operand addressing mode and the operand value itself.

These functions must not use string manipulations.

Concurrent writes resolution

When multiple processes want to write at the same location, a voting system will take place: for each bit of the target word, if the number of processes that want to write the bit `b` is greater than the number of processes that want to write the bit `1-b`, then the bit `b` is written. If there is a tie, then the bit is left unchanged.

This question is about writing this poll process:

- write a function `resolve_writes(base, xs)` that takes a 32-bit word `base` and a list memory updates (i.e. of 32-bit words) `xs` for `base`. The function should apply the concurrent process resolution to base and return its new value.

  E.g., if

```
base =   0b00000000_00000000_00000100_10000001
xs   = [
         0b00000000_00000000_00000000_00000010,
         0b00000000_00000000_00000000_00000010,
         0b00000000_00000000_00000001_10000000,
         0b00000000_00000000_00000101_10000010,
]
```

  then the result should be

```
0b00000000_00000000_00000000_10000010
```

These functions must not use string manipulations.

Memory

We are going to encapsulate the memory into a class `Memory`, that should contain the following methods:

- `.__init__(self, size)` that creates a new memory (of 32-bit words) of size `size`. Initially, all the memory cells are set to `0`.
- `.__len__(self)` that returns the size of the memory,
- `.__getitem__(self, idx)` that returns the memory cell `idx` contents. Note that the memory is cyclic, hence `idx` might be outside the range `[0..size)`.
- `.__setitem__(self, idx, value)` that update the memory cell `idx` contents using `value`. Note that the memory is cyclic, hence `idx` might be outside the range `[0..size)`. The memory write should not be immediate. Instead, it should be stored in an internal data attribute dedicated to the storage of pending writes.
- `.writes(self)` that returns all the pending writes as a dictionary from addresses to lists of writes at that address. The addresses should be normalized, i.e. in the range `[0..size)`.
- `.commit(self)` that commits all the pending writes in memory (see `resolve_writes`). Once applied, the list/dictionary of pending writes should be empty.

```
0.0e+0
```

- `.load(self, data, offset)` that copy `data` (a list of 32-bit words) in memory, starting at index `offset` - cycling is necessary. The loading of data should be immediate, i.e. `data` is directly loading into memory, not added to the list of pending writes. The behaviour of the function is not specified is there are pending writes when the function is called.

Processes

We now need a way to represent processes. We remind you that a process is the data of:

- a set of 16 (32-bit word) registers (`r0` to `r15`),
- a cyclic stack (of 32-bit words) of size `16` ,
- a special register `PC` (Program Counter), and
- a flag (or boolean) `Z` .

We are going to pack all these values in a class `Process` . Your constructor should set the following data attributes:

- `.registers` : for the set of the process' registers, packed in an array, `r0` being the first register and `r15` being the last. Initially, all the registers are set to `0` .
- `.stack` : the process' stack - as a `CyclicStack` instance.
- `.PC` : for the process' program counter, as a single integer that is initially set to `0` .
- `.Z` : for the process' `Z` flag, as a single integer that is initially set to `0` .

Low-level instructions

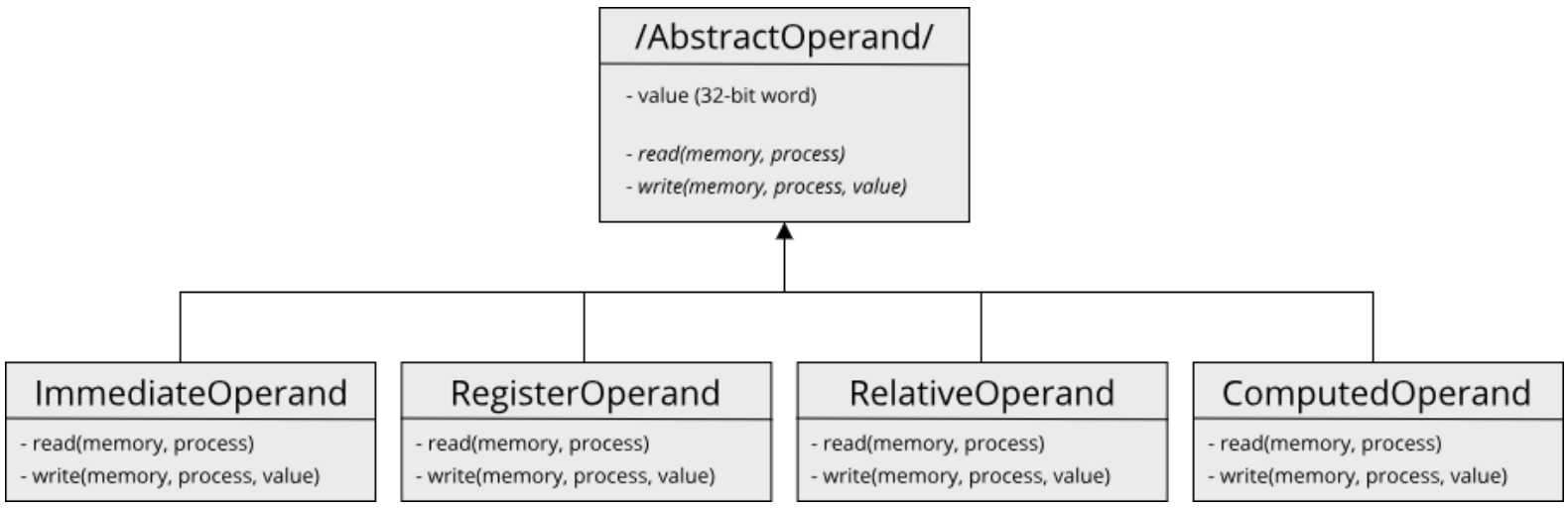
In this section, we are going to implement the computational part of the instructions (i.e. their result) when it makes sense, i.e. for `ADD` , `SUB` , `NOT` , `AND` , `OR` , `LS` , `AS` , `CMP` and `LT` . All functions take uninterpreted 32-bit words and should return an uninterpreted 32-bit word (with the exception of `CMP` and `LT` that return a boolean).

- write a function `eval_ADD(w1, w2)` that returns the 32-bit addition of `w1` and `w2` (the addition has the same behaviour wether you interpret `w1` and `w2` as both signed or unsigned integers).
- write a function `eval_SUB(w1, w2)` that returns the 32-bit substraction of `w1` by `w2` (the substraction has the same behaviour wether you interpret `w1` and `w2` as both signed or unsigned integers).
- write a function `eval_NOT(w)` that returns the 32-bit bitwise negation of `w` .
- write a function `eval_AND(w1, w2)` that returns the 32-bit bitwise AND of `w1` or `w2` .
- write a function `eval_OR(w1, w2)` that returns the 32-bit bitwise OR of `w1` or `w2` .
- write a function `eval_LS(a, w)` that returns the 32-bit logical shift of `w` by `a` bits. The word `a` as to be interpreted as a signed integer `i` . When `i` is non-negative, the shift should be done toward the right by `i` bits. When `i` is negative, the shift should be done toward the left by `abs(i)` bits.
- write a function `eval_AS(a, w)` that returns the 32-bit arithmetic shift of `w` by `a` bits. The word `a` as to be interpreted as a signed integer `i` . When `i` is non-negative, the shift should be done toward the right by `i` bits. When `i` is negative, the shift should be done toward the left by `abs(i)` bits.
- write a function `eval_CMP(w1, w2)` that returns a boolean indicating wether `w1` and `w2` are the same word.
- write a function `eval_LT(w1, w2)` that returns a boolean indicating wether `w1` is strictly smaller than `w2` , when interpreted as signed integers.

High-level operands

We are now interested in writing a class hierarchy for representing operands. The general layout will be the following:

0.03 / 0



i.e. we have an abstract top-class `AbstractOperand` for operands, that contains the operand value (but not the operand mode) and then, we have one subclass of `Operand` per operand mode. The `value` attribute is the 12-bit (non-interpreted) word that has been extracted from the instruction. All these classes are going to share two methods `.read(self, memory, process)` and `.write(self, memory, process, value)` for resp. reading from and writing to the operand - these two methods are abstract in the top-class and are going to be redefined in the subclasses. Note that the `.write` operation is not supported by all operands.

We here provide the base abstract class:

```
class InvalidOperation(Exception):
    pass

class AbstractOperand:
    def __init__(self, value):
        self.value = value

    def read(self, memory, process):
        raise InvalidOperation

    def write(self, memory, process, value):
        raise InvalidOperation
```

We ask you to:

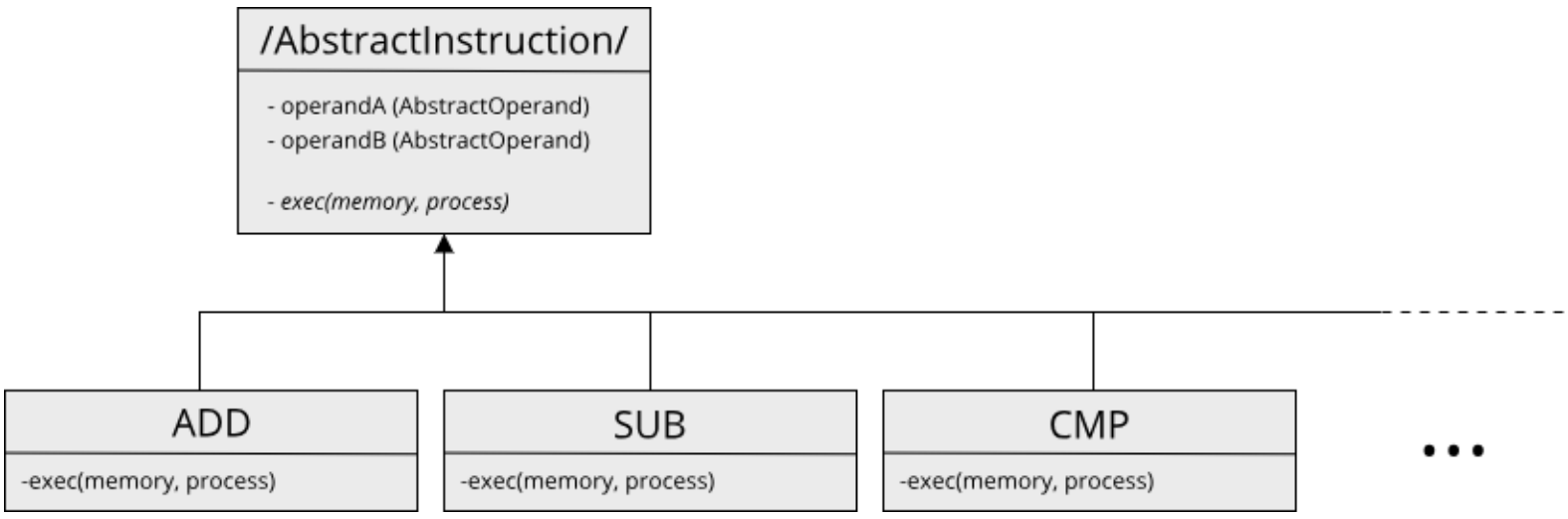
- define the 4 subclasses `ImmediateOperand`, `RegisterOperand`, `RelativeOperand` and `ComputedOperand`, that all should inherit from `AbstractOperand`.
- implement their `.read(self, memory, process)` and `.write(self, memory, process, value)` methods – if some operation is meaningless (i.e. `.write` for an immediate operand), the method body should be left as-is.
- add a *static* method `.create(opmode, opvalue)` to `AbstractOperand` that takes an operand mode `opmode` and an operand value `opvalue` and that creates, based on `opmode`, the relevant instance of `AbstractOperand`.

Note that you should not commit memory writes at this point!

High-level instructions

We are now interested in writing a class hierarchy for representing operands. The general layout will be the following:

0.03 / 0



i.e. we have an abstract top-class `AbstractInstruction` for instructions, that contains the instructions operands and then, we have one subclass of `AbstractInstruction` per instruction. All these classes are going to share one method `.exec(self, memory, process)` that executes the actual instruction – this method is abstract in the top-class and is going to be redefined in the subclasses.

We here provide the base abstract class:

```
class AbstractInstruction:
    def __init__(self, operandA, operandB):
        self.operandA = operandA
        self.operandB = operandB

    def exec(self, memory, process):
        raise NotImplementedError
```

We ask you to:

- define one subclass or `AbstractInstruction` per instruction – the name of the class should be the name of the instruction (in upper case).
- implement their `.exec(self, memory, process)` method, that executes the actual instruction using memory `memory` and process state `process`. If you want to indicate that the instruction execution leads to a runtime error (e.g. executing `DIE` or writing to an invalid location), you can raise the `InvalidOperation` exception. The method should return a list of the processes that have been created by the instruction (the current executed process should not be oart of that list) - see for instance the `FORK` instruction. In most cases, this list is going to be empty.
- add a *static* method `.create(opcode, operandA, operandB)` to `AbstractInstruction` that takes an instruction opcode (as an integer) and two operands (as instances of `AbstractOperand`) and that creates, based on `opcode`, the relevant instance of `AbstractInstruction`.

Note that you should not commit memory writes at this point!

Packing all together

We are now ready to pack everything. For that purpose, we are going to create a class `Machine` that will load the programs and execute them in an endless loops (or until all the processes but one die).

Define a class `Machine` that has the following methods:

- a constructor `.__init__(self, program1, program2)` that takes two programs `program1` and `program2` (as lists of 32-bit words) and create a virtual machine by:
  - creating a memory of size `4096`,
  - loading the programs in memory at resp. offset `0` and `2048`. If the programs overlap in memory, your constructor should raise a `ValueError` exception.
  - creating two processes queues for each program. Initially, these queues contain one process each (see the `Process` class). Initially, the PC of the first process (resp. the second process) is `0` (resp. `2048`).

0.03 / 0

- three data-attribute accessors:
  - `.memory` : a reference to the actual memory,
  - `.player1` : a reference to the first player processes queue (as a list of `Process`, the front of the queue being at index `0`),
  - `.player2` : a reference to the second player processes queue (as a list of `Process`, the front of the queue being at index `0`).
- a method `.status(self)` that returns, **for the current machine state**:
  - `1` (resp. `2`) if the first player wins (resp. the second player wins)
  - `0` if there is a tie (i.e. all the players' processes queues are empty)
  - `None` if the game still continues (i.e. all the payers's have at least one process in their queues)
- a method `.step(self)` that executes one round, i.e.
  1. for each player, pop the front process for the player's processes queue,
  2. for each poped process, decode the instruction pointed by the process `PC` and execute it. If the instruction raises `InvalidOperation`, the process dies. Otherwise, the process is pushed-back in its respective queue. Moreover, if the instruction created new processes, these processed are then pushed back in the same queue.
  3. commit the memory writes
- a method `.run(self)` that runs the machine until there is a winner or a tie.

Adding an entry point to your program

We are now ready to add an entry point (i.e. a main function) to your program.

Write a function `main()` that loads two programs from two files given as command line arguments (you can get them using `sys.args`) and run the virtual machine until there is a tie or a winner.

You can load a file in binary format using:

```
with open(filename, 'rb') as stream:
    contents = stream.read()
if len(contents) % 4 != 0:
    raise ValueError
contents = [
    int.from_bytes(contents[i:i+4], 'little')
        for i in range(0, len(contents), 4)
]
```

Detailed workplan (assembler)

We provide here a detailed plan for implementing the assembler. If you follow this steps one-by-one, you will eventually obtain a working final program.

Upload form is only available when connected

The assembler is going to work by successive passes that we will eventually chain.

We remind you that you can get iterate over all the lines of a file with the folllowing code:

0.03 / 0

```
def readlines(filename):
    with open(filename, 'r') as stream:
        for line in stream:
            yield line.rstrip('\r\n')
```

For the automated tests, when asked to raise an exception, use a user defined exception (i.e. an exception that you define in your Python module).

The automated tests are only triggered when your Python file contains a function named `assembler` - such a function is given at the end of the workplan. This function is not used by the automated tests but indicates that you are done with the workplan.

As a running example, we will consider the following program that runs a loop, counting numbers from 127 to 0.

```
        MOV $127 r1  ; Initialize r1 to 127

&loop:  ADD $-1 r1   ; Decrement  r1 by 1
        BZ  $&end    ; If r1 is 0, move to end of program
        JMP $&loop   ; Otherwise, jump to loop

&end:   DIE
```

Removing comments / stripping empty lines

The first function is simply going to remove comments from the input line and to strip empty lines (i.e. lines composed only of spaces and comments) from the file.

Write a function `strip_comment(line)` that takes a single line of source code and that:

- remove its comment (i.e. the `; ...` part) if any, then
- remove any leading or trailing space (there is a method in the `str` class for that – search for it),
- return the obtained line if it is not empty – otherwise, the function returns `None`.

Applying the function to all the lines of the sample program, you should obtain:

```
program0 = r'''
        MOV $127 r1  ; Initialize r1 to 127

&loop:  ADD $-1 r1   ; Decrement  r1 by 1
        BZ  $&end    ; If r1 is 0, move to end of program
        JMP $&loop   ; Otherwise, jump to loop

&end:   DIE
'''

program0 = program0.splitlines()
program1 = [strip_comment(x) for x in program0]
program1 = [x for x in program1 if x is not None]

# We now have:
#   program1 = ['MOV $127 r1', '&loop:   ADD $-1 r1', 'BZ  $&end', 'JMP $&loop', '&end:     DIE']
```

Extracting labels

We are now interested in extracting the labels. For that purpore, we implement a second phase that will extract all the labels from the source file and associate to each label the index of the line where it is defined.

0.03 / 0

Write a function `extract_label(line, lineno, labels)` that takes a single line of source code and that:

- extract its labels (i.e. the `&xxx:` part) if any, then
- associate the current line number `lineno` to `&xxx` in `labels` – i.e. the line number where it is defined. Note that if `&xxx` is already defined, this is an error and your function should raise an exception,
- return the line pruned from it label.

Applying the function to all the elements of `program1`, starting with an empty dictionary, you should obtain:

```
labels   = {}
program2 = [extract_label(x, i, labels) for i, x in enumerate(program1)]

# We now have:
#   program2 = ['MOV $127 r1', 'ADD $-1 r1', 'BZ  $&end', 'JMP $&loop, 'DIE']
#   labels   = { '&loop': 1, '&end': 4 }
```

Extracting mnemonics & operands

We are now going to extract the mnemonic (i.e. the name of the instruction) and the operands.

Write a function `parse_operand(oprd, index, labels)` that takes a single operand (in source form, e.g. `$10` or `@address` where `address` is a label), an `index` (designated the instruction index where this operand is used) and a labels definitions dictionary, and that decompose it into a pair composed of the operand mode (as a string equal to `$`, `@`, `#` or `r`) and the operand value (which should now be an integer at this point - if the operand value is a label, you should resolve if using the dictionary `labels`). Note that operands' values are encoded on 12-bits: you should check that literals are not too big (i.e. are in the range `[0..2^12)` when considering them unsigned). For signed operands (i.e. prefixed with the sign `-`), use a 2-complement encoding.

Write a function `parse_instruction(txt, index, labels)` that takes a line of code (pruned from any comments or label definition), the index of this line of code and that returned a triplet composed of:

- the instruction (as a string equal to the instruction name as 'FORK', 'DIE'). If the instruction name is invalid, the function should raise an exception.
- the first operand (or `None` is there is no such operand), and
- the second operand (or `None` is there is no such operand).

Applying the function to all the elements of `program2`, starting with an empty dictionary, you should obtain:

```
program3 = [parse_instruction(x, i, labels) for i, x in enumerate(program2)]

# We now have:
#   program3 = [
#     ('MOV', ('$', 127), ('r', 1)),
#     ('ADD', ('$', 4095), ('r', 1)),
#     ('BZ' , ('$', 2), None),
#     ('JMP', ('$', 4094), None),
#     ('DIE', None, None),
#   ]
```

Validating instructions

We now need to validate instructions. E.g. `DIE $10` is invalid because `DIE` does not take any argument. Likewise, `MOV r2 $10` is invalid because `MOV` writes the content of its first operand in its second operand and a literal (here `$10`) is now writable.

Write a function `validate_instruction` that takes an instruction as returned by `parse_instruction` and that check:

- if the instruction name is valid,
- if the number of operands matches the expected one,

- if the the operands make sense for the instruction (essentially, you should check wether an operand is writable when the instruction needs to write it).

The function should raise an exception if the validation fails.

Encoding instructions

We are now ready to encode instructions:

- Write a function `instruction_name_code(name)` that takes an instruction name `name` and that returns its 4-bit code (as an integer).
- Write a function `operand_mode_code(mode)` that takes an operand mode (as a string in `$`, `@`, `#` or `r`) and that returns its 2-bit code (as an integer).
- Write a function `instruction_code(instr)` that takes an instruction (as returned by `parse_instruction`) and that returns its 32-bit code (as an integer).

Applying the function to all the elements of `program3`, you should obtain:

```
program4 = [instruction_code(x) for x in program3]

# We now have:
#   program4 = [1081281, 2097095, 526, 1048077, 15]
```

Packing everything

You can now pack everything together with the following code:

```
def assembler(filename):
    labels, lineno, src, codes, aout = {}, 0, [], [], bytearray()
    with open(filename, 'r') as stream:
        for line in stream:
            line = strip_comment(line)
            if line is not None:
                src.append(extract_label(line, lineno, labels))
                lineno += 1
    for i, line in enumerate(src):
        instr = parse_instruction(line, i, labels)
        validate_instruction(instr)
        codes.append(instruction_code(instr))
    for i in codes:
        aout.extend(int.to_bytes(i, 4, 'little'))
    return bytes(aout)
```

Note that you can save a byte-array in binary form in a file with the following code (note the `wb`):

```
with open('myfile.cor', 'wb') as stream:
    stream.write(my_byte_array)
```

0.03 / 0