Chapter II

Introduction

What is Corewar?

- Corewar is a very peculiar game. It's about bringing "players" together around a "virtual machine", which will load some "champions" who will fight against one another with the support of "processes", with the objective being for these champions to stay "alive".

- The processes are executed sequentially within the same virtual machine and memory space. They can therefore, among other things, write and rewrite on top of each others so to corrupt one another, force the others to execute instructions that can damage them, try to rewrite on the go the coding equivalent of a `Côtes du Rhône 1982` (that is one delicious French wine!), etc...

- The game ends when all the processes are dead. The winner is the last player reported to be "alive".

Chapter III

Objectives

Breakdown of the project's objectives

This project can be broken down into three distinctive parts:

- `The assembler`: this is the program that will compile your champions and translate them from the language you will write them in (assembly language) into "Bytecode".Bytecode is a machine code, which will be directly interpreted by the virtual machine.

- `The virtual machine`: It's the "arena" in which your champions will be executed. It offers various functions, all of which will be useful for the battle of the champions. Obviously, the virtual machine should allow for numerous simultaneous processes; we are asking you for a gladiator fight, not a `one-man show simulator`.

- `The champion`: This one is a special case. Later, in the championship, you will need to render a super powerful champion, who will scare the staff team to death. However, rendering this kind of champion is serious work. And since, for now, we are mostly interested in your capacity to create Corewar's other programs (i.e. the assembler and virtual machine), your current champion will only need to prove to us that you can write bits and pieces of Corewar ASM. This means that the champion you should render for this project only needs to scare the bejesus out of a neurasthenic hedgehog.

There will also be a Corewar championship, for which you will create new champions that will fight in a series of epic battles. The highpoint of this championship will make `Game of Thrones` look like nap time at kindergarten.

Please note that the championship is a *separate* project for which you will render a *new* champion. It would therefore be wise for you to keep your most twisted strategies to yourself, so that you don't become, so to speak, the laughing stock of this championship of utmost seriousness.

Chapter IV

General Instructions

- This project will be corrected by humans only. You're allowed to organise and name your files as you see fit, but you must follow the following rules.
- The executable file must be named `asm` and `corewar`.
- Your champion needs to have a majestic, epic, glorious name.
- Your `Makefile` must compile the project and must contain the usual rules. It must recompile and re-link the program only if necessary.
- If you are clever, you will use your library for your `RTv1`. Submit also your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.
- Your project must be written in accordance with the Norm. Even if under the powerful influence of some drug, ours isn't. Only norminette is authoritative.
- You have to handle errors carefully. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- Your program cannot have memory leaks.
- You'll have to submit a file called `author` containing your username followed by a '\n' at the root of your repository.

```
$>cat -e author
xlogin$
ylogin$
zlogin$
alogin$
$>
```

- You are Never allowed to submit a code that you did not write yourself. If any doubts exist, you will be invited to recode it to show your good faith.
- Within the mandatory part, you are allowed to use only the following libc functions:

  - `open`
  - `read`
  - `lseek`
  - `write`
  - `close`
  - `malloc`
  - `realloc`
  - `free`
  - `perror`
  - `strerror`
  - `exit`

- You can ask your questions on the forum, on slack...

- We'll provide you with an assembler and a virtual machine, both of which will function properly. However we will not provide you with their source codes. You will need to think about this one carefully.

- Good luck to all!

Chapter V

The virtual machine

- Each process will have available the following exclusive elements available:
    - REG_NUMBER registries, each of which are the size REG_SIZE octets. A registry is a small memory "box" with only one value. On a real machine, it is an internal of the processor and as a result very FAST to access.
    - A PC ("Program Counter"). This is a special registry that only contains, within the memory of the virtual machine, the address of the next set of instructions to code and execute. Very useful to figure out where we are at in the execution, giving us tips on when to write stuff in the memory...
    - A flag named `carry`, if the latest operation was successful. Only certain operations can modify the `carry`.
- The number of the player is generated by the machine or specified at launch and is given to the champions via the r1 registry of their first process at startup. All the other registries are at 0, except PC.
- The champions are charged within the memory so that they can space out evenly their entry points.
- The virtual machine will create a memory space dedicated to the combat of the players, it will then load each champion and their associated processes and execute them sequentially until they die.
- Every CYCLE_TO_DIE cycles, the machine needs to make sure that each process has executed at least one `live` since the last check. A process that does not abide by this rule will be killed immediately with a virtual foamy bat (bonus for sound effect!)
- If during one of those checkup we notice that there has been at least one NBR_LIVE execution of `live` since the latest check up, we will decrease CYCLE_TO_DIE of CYCLE_DELTA units.
- The game is over when all processes are dead.

- The winner is the last player to be reported alive. The machine will then show `"Player X (champion_name) won"`, where X is the player's number and champion_name is its name.
  For example: `"Player 2 (rainbowdash) won"`.

- For each valid execution of the `live` instruction, the machine must display:
  `"A process shows that player X (champion_name) is alive"`.

- In any case, memory is circular and of MEM_SIZE octets.

- In case of an error, you must display a relevant error message on the standard error output.

- If CYCLE_TO_DIE wasn't decreased since MAX_CHECKS checkups, decrease it.

- The virtual machine should be executed like that:

```
> ./corewar [-dump nbr_cycles] [[-n number] champion1.cor] ...
```

- `-dump nbr_cycles`
  at the end of `nbr_cycles` of executions, dump the memory on the standard output and quit the game. The memory must be dumped in the hexadecimal format with 32 octets per line.

- `-n number`
  sets the number of the next player. If non-existent, the player will have the next available number in the order of the parameters. The last player will have the first process in the order of execution.

- The champions cannot go over CHAMP_MAX_SIZE, otherwise it is an error.

Chapter VI

The assembler

We are interrupting this topic to give you a news flash: According to our sources, one out of seven 42 student loves blue smells.

- Your virtual machine will execute a machine code (or "bytecode") that will be generated by your assembler. The assembler (the program) will get a file written in assembly language as argument and generate a champion that will be understood by the virtual machine.

- It will run like that:

```
> ./asm mychampion.s
```

- It will read the assembly's code processed from the file `.s` given as argument, and write the resulting bytecode in a file named same as the argument by replacing the extension `.s` by `.cor`.

- In case of an error, you will need to display a relevant message on the standard error output and not create the `.cor` file.

Chapter VII

The champion

- Your champion are three intrinsic objectives: Make sure its player is reported "alive", understand the meaning of life, and eradicate its opponents.

- For your player to be qualified as "alive", your champion must make sure that some `live` are achieved with its number. If one of the processes does a `live` with the number of another player... well tough luck, but at least another player will be happy. If the process of another player scores a `live` with your number, you are authorized to make fun of him and you can cash in on his mistake, while insulting his family in binary code.

- All, and absolutely ALL the instructions are useful. All the machine's reactions, described further in the chapter on language can be used to give life to your champion and enable him to win a prize of seventeen euros and fifty three cents in the championship. Yes, even the `aff` instruction is useful to laugh at the uselessness of your opponents.

- Your champion will be graded during defence on its capacity to survive a few basic challenges, such as defeating a champion with the IQ of a banana, managing to eat my grand mother's apple pie, or drawing flowers in a cappuccino.

- Later, you will get the chance to create a new champion, who will be destined to fight in a championship (remember: that's another project!) and fight against your classmates' champions, maybe even against one of the staff's champions, which might quickly transform your own champion into a pile of virtual casings. Yet it's highly possible that after a bit of voodoo, including some dark magic, a `safety pin` and a particular place that moral and decency forbids me to name, you might end up covered in `glory` and `kittens`.

Chapter VIII

The langage and compilation

VIII.1 Assembly language

- The assembly language is composed of one instruction per line.
- An instruction is composed of three elements: a label (optional) composed with a chain of characters amongst LABEL_CHARS followed by LABEL_CHAR; an opcode; and its parameters, separated by SEPARATOR_CHAR. A parameter can be of three different types:
  - Registry: (r1 <–> rx with x = REG_NUMBER)
  - Direct: The character DIRECT_CHAR followed by a numeric value or a label (preceded by LABEL_CHAR) which represents a direct value.
  - Indirect: A value or a label (preceded by LABEL_CHAR), which represents a value located at the address of the parameter, relative to the PC of the current process.
- A label can have no instruction following it or be placed on a line before the instruction it responds to.
- The caracter COMMENT_CHAR starts a comment.
- A champion will also have a name and a description, that should be on a line following the markers NAME_CMD_STRING and COMMENT_CMD_STRING.
- All the addresses are related to PC and to IDX_MOD except for `lld`, `lldi` and `lfork`.
- The number of cycles for each instruction, their mnemonic representations, the associated amount and possible types of arguments are described in the op_tab array declared in op.c. The cycles are always consumed.
- All the other codes have no other action than to pass to the next one and lose a cycle.

- `lfork`: means `long-fork`, to be able to fork abut straw from a distance of 15 meters, exactly like with its opcode. Same as a `fork` without modulo in the address.

- `sti`: Opcode 11. Take a registry, and two indexes (potentially registries) add the two indexes, and use this result as an address where the value of the first parameter will be copied.

- `fork`: there is no argument's coding byte, take an index, opcode 0x0c. Create a new process that will inherit the different states of its father, except its PC, which will be put at (PC + (1st parameter % IDX_MOD)).

- `lld`: Means `long-load`, so it's opcode is obviously 13. It the same as `ld`, but without % IDX_MOD. Modify the carry.

- `ld`: Take a random argument and a registry. Load the value of the first argument in the registry. Its opcode is 10 in binary and it will change the carry.

- `add`: Opcode 4. Take three registries, add the first two, and place the result in the third, right before modifying the carry.

- `zjmp`: there's never been, isn't and will never be an argument's coding byte behind this operation where the opcode is 9. It will take an index and jump to this address if the carry is 1.

- `sub`: Same as `add`, but with the opcode est 0b101, and uses a substraction.

- `ldi`: `ldi`, ldi, as per the name, does not imply to go swimming in chestnut cream, even if its code is 0x0a. Instead, it will use 2 indexes and 1 registry, adding the first two, treating that like an address, reading a value of a registry's size and putting it on the third.

- `or`: This operation is an bit-to-bit OR, in the same spirit and principle of `and`, its opcode is obviously 7.

- `st`: take a registry and a registry or an indirect and store the value of the registry toward a second argument. Its opcode is 0x03. For example, `st r1, 42` store the value of r1 at the address (PC + (42 % IDX_MOD))

- `aff`: The opcode is 10 in the hexadecimal. There is an argument's coding byte, even if it's a bit silly because there is only 1 argument that is a registry, which is a registry, and its content is interpreted by the character's ASCII value to display on the standard output. The code is modulo 256.

- `live`: The instruction that allows a process to stay alive. It can also record that the player whose number is the argument is indeed alive. No argument's coding byte, opcode 0x01. Oh and its only argument is on 4 bytes.

- `xor`: Acts like `and` with an exclusive OR. As you will have guessed, its opcode in octal is 10.
- `lldi`: Opcode 0x0e. Same as `ldi`, but does not apply any modulo to the addresses. It will however, modify the carry.
- `and`: Apply an & (bit-to-bit AND) over the first two arguments and store the result in a registry, which is the third argument. Opcode 0x06. Modifies the carry.

VIII.2 Encoding

Each instruction is encoded by:

- The instruction code (you find it in `op_tab`).
- The argument's coding byte if appropriate. To be done as per the following examples:
  - `r2,23,%34` gives the coding byte `0b01111000`, hence `0x78`
  - `23,45,%34` gives the coding byte `0b11111000`, hence `0xF8`
  - `r1,r3,34` gives the coding byte `0b01011100`, hence `0x5C`
- The arguments, based on the following examples:
  - `r2,23,%34` gives the ACB `0x78` then `0x02 0x00 0x17 0x00 0x00 0x00 0x22`
  - `23,45,%34` gives the ACB `0xF8` then `0x00 0x17 0x00 0x2d 0x00 0x00 0x00 0x22`

Some important notes:

- The executable will always start with a header, defined in op.h by the `header_t` type.
- The virtual machine is BIG ENDIAN. Ask `Google` what it means.

VIII.2.1 Complete example of compilation

```
.name "zork"
.comment "just a basic living prog"

l2:     sti r1,\%:live,\%1
        and r1,\%0,r1

live:   live \%1
        zjmp \%:live

# Executable compilation:
#
```

```
# 0x0b,0x68,0x01,0x00,0x0f,0x00,0x01
# 0x06,0x64,0x01,0x00,0x00,0x00,0x00,0x01
# 0x01,0x00,0x00,0x00,0x01
# 0x09,0xff,0xfb
```

VIII.3 Champion's execution

- The virtual machine is supposed to emulate a a machine perfectly parallel.
- However, for implementation purposes, we will suppose that each instruction will execute itself (completely) at the end of its last cycle and wait for its entire duration. The instructions ending at the same cycle will execute themselves in in decreasing order of the processes' number.
- Yes, the last born (youngest) champion plays first.

Chapter IX

The championship

- At some point in time the Corewar championship will start. The specific date and time will be determined by a cabalistic ritural.
- Your champions will fight each other in an epic battle and will most likely have to defeat staff member's champions...
- The winners will be covered with:
  - `glory`;
  - `booze`;
  - `goodies`;
  - `kittens`.
- The losers, poor things, will be:
  - dragged in `mud`;
  - booed;
  - laughed at by `street workers`.
  - forced to watch as their champion gets slapped.
- This championship is a separate project, all you'll have to do is present a champion (not necessarily the same one as for your Corewar, for obvious strategic and secret reasons...). It will be executed on our own virtual machine, so the configuration will be the one you will describe in your file `op.h` that will be attached. Be careful, the files `op.c` and `op.h` will be given as an example and you will most definitely need to modify them. It's quite possible that they won't work as we might have mistaken a bottle of water for a bottle of `vodka`.

Chapter X

Bonus part

We will look at your bonuses if and only if your mandatory part is EXCELLENT. This means that your must complete the mandatory part, beginning to end, and your error management must be flawless, even in cases of twisted or bad usage. If that's not the case, your bonuses will be totally IGNORED.

After you've successfuly coded a complete `Corewar` worthy of immortality and leveraged your community service to have its entire code sculpted in gold plate over precious wood, you can of course start considering bonuses. The possibilities are endless! However, keep in mind that any mistakes in the bonuses will risk to invalidate the entire project. You will need to be very meticulous!

Here are a few ideas:

- A graphic interface for the virtual machine using the library of your choice. (OpenGL, SDL, nCurses... whatever floats your boat!)
- A network game mode
- New instructions
- The support of mathematical operations in the .s files.

Once again be cautious about the quality of your bonus, and keep your priorities straight. `Corewar` is not your average project. It is somewhat easy to consume a bit too much illegal substances and invalidate a whole month of work with a single mistake...