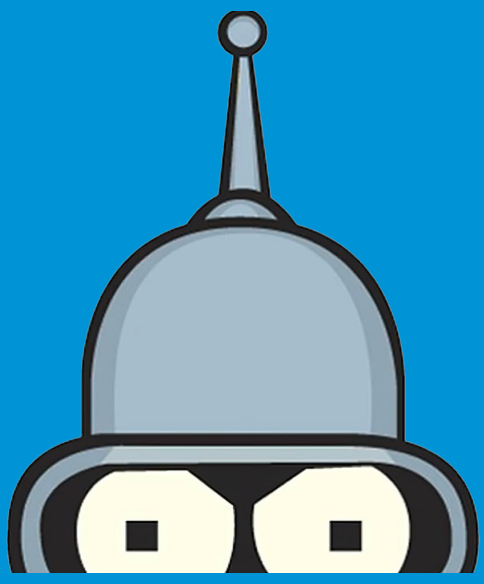
{EPITECH}

COREWAR

ELEMENTARY PROGRAMMING IN C



COREWAR

binary name: corewar
language: C
compilation: via Makefile, including re, clean and fclean rules
Authorized functions: (f)open, (f)read, (f)write, (f)close, (l)stat, lseek, fseek, getline, malloc, realloc, free

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,…), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

> Hey there! You've come a long way since your arrival at Epitech, and now, you stand at the threshold of ultimate glory: the grand Corewar tournament.
>
> Picture yourself in a virtual arena, surrounded by the roar of the crowd as robots clash in an unparalleled battle. This is where your skills will be put to the test, where your robot must prove its worth against other champions of computing.
>
> In this final challenge, you'll put into practice everything you've learned so far. You'll utilize your knowledge of programming, algorithms, and strategy to bring forth an unbeatable robot.
>
> So, are you ready to take on this challenge? To face off against the best minds in computing from around the world? To step into the arena and fight for victory? The Corewar tournament awaits you, future champion. Show the world what you're capable of!
>
> **— Narrator —**

The project

Objectives

The Corewar tournament is a game in which several programs called `Champions` will fight to stay the last one alive. Corewar is actually a computer simulation in which processors will share memory to run on.
The project is based on a virtual machine in which the champions will fight for memory by all possible means in order to win, the last champion being able to signal that he is alive wins the game. In other words, the last champion to execute the `live` instruction is declared winner.

Search “corewars” and “redcode” on the Internet…

The different parts

The project is divided into three seperate parts, with only one to develop:

- ✓ Champions (given, you don’t have to write any)
  The champions are files written in an assembly language specific to our virtual machine. The file is filled with instructions that the champion must follow, it’s his line of conduct. It is in this file that the champion knows when he must attack, defend himself or announce that he is still alive. Examples of champions are given in the attachments.
- ✓ The Assembler (developed in Robot Factory project)
  The purpose of this program, like a program that your own computer would try to run, is to transcribe the champions (the .s files) into a language that the Corewar tournament program can understand. It will therefore be necessary to understand the assembly language in order to translate it byte by byte.
- ✓ **The Virtual Machine** (current project)
  The virtual machine come only after the development of the assembly part because it will ask you to execute the instructions of each of the champions and thus… to understand the machine language that you just translated! The virtual machine is thus a program which will place at the disposal a memory zone so that the champions can share it and fight on it. The importance of the machine is to correctly execute the instructions of each one by respecting an order and a cycle of play in an allotted time or until there remains only one.

op.c and op.h

To get to the end of the Corewar and for each of you to develop champions with similar instructions and architecture, two files are made available to you: `op.c` and `op.h`.
You will have to integrate them into your rendering directory.

All values written in **UPPERCASE** in this subject are variables obtainable in `op.c` or `op.h`.

The coding style will be checked on all the files that you deliver, including the currently non-compliant `op.c` and `op.h`.

Reminder: The instructions

Mnemonic	Parameters / Effects
0x01 (live)	takes 1 parameter: 4 bytes that represent the player's number. It indicates that the player is alive.
0x02 (ld)	takes 2 parameters. It loads the value of the first parameter into the second parameter, which must be a register (not the PC). **This operation modifies the carry**. `ld 34, r3` loads the `REG_SIZE` bytes starting at the address `PC + 34 % IDX_MOD` into `r3`.
0x03 (st)	takes 2 parameters. It stores the first parameter's value (which is a register) into the second (whether a register or a number). `st r4, 34` stores the content of `r4` at the address `PC + 34 % IDX_MOD`. `st r3, r8` copies the content of `r3` into `r8`.
0x04 (add)	takes 3 registers as parameters. It adds the content of the first two and puts the sum into the third one (which must be a register). **This operation modifies the carry**. `add r2, r3, r5` adds the content of `r2` and `r3` and puts the result into `r5`.
0x05 (sub)	similar to `add`, but performing a subtraction.
0x06 (and)	takes 3 parameters. It performs a binary AND between the first two parameters and stores the result into the third one (which must be a register). **This operation modifies the carry**. `and r2, %0, r3` puts `r2 & 0` into `r3`.
0x07 (or)	similar to `and`, but performing a binary OR.
0x08 (xor)	similar to `and`, but performing a binary XOR (exclusive OR).
0x09 (zjmp)	takes 1 parameter, which must be an index. It jumps to this index if the carry is worth 1. Otherwise, it does nothing but consumes the same time. `zjmp %23` puts, if carry equals 1, `PC + 23 % IDX_MOD` into the PC.
0x0a (ldi)	takes 3 parameters. The first two must be indexes or registers, the third one must be a register. **This operation modifies the carry**. `ldi 3, %4, r1` reads `IND_SIZE` bytes from the address `PC + 3 % IDX_MOD`, adds 4 to this value. The sum is named `S`. `REG_SIZE` bytes are read from the address `PC + S % IDX_MOD` and copied into `r1`.
0x0b (sti)	takes 3 parameters. The first one must be a register. The other two can be indexes or registers. `sti r2, %4, %5` copies the content of `r2` into the address `PC + (4+5)% IDX_MOD`.
0x0c (fork)	takes 1 parameter, which must be an index. It creates a new program that inherits different states from the parent. This program is executed at the address `PC + first parameter % IDX_MOD`.
0x0d (lld)	similar to `ld` without the `% IDX_MOD`. **This operation modifies the carry**.
0x0e (lldi)	similar to `ldi` without the `% IDX_MOD`. **This operation modifies the carry**.
0x0f (lfork)	similar to `fork` without the `% IDX_MOD`.
0x10 (aff)	takes 1 parameter, which must be a register. It displays on the standard output the character whose ASCII code is the content of the register (in base 10). A 256 modulo is applied to this ASCII code. `aff r3` displays '*' if `r3` contains `42`.

Virtual Machine

```
Terminal
~/B-CPE-200> ./corewar -h
USAGE
./corewar [-dump nbr_cycle] [[-n prog_number] [-a load_address] prog_name] ...
DESCRIPTION
-dump nbr_cycle dumps the memory after the nbr_cycle execution (if the round isn't
already over) with the following format: 32 bytes/line in hexadecimal (A0BCDEFE1DD3...)
-n prog_number sets the next program's number. By default, the first free number in the
parameter order
-a load_address sets the next program's loading address. When no address is specified,
optimize the addresses so that the processes are as far away from each other as
possible. The addresses are MEM_SIZE modulo.
```

The **-dump** flag is therefore mandatory for the correction.

The virtual machine is a multi-program machine. Each program contains the following:

- ✓ `REG_NUMBER` registers of `REG_SIZE` bytes each.
  A register is a memory zone that contains only one value. In a real machine, it is embedded within the processor, and can consequently be accessed very quickly. `REG_NUMBER` and `REG_SIZE` are defined in op.h.
- ✓ A PC (Program Counter)
  This is a special register that contains the memory address (in the virtual machine) of the next instruction to be decoded and executed. It is very practical if you want to know where you are and to write things in the memory.
- ✓ A flag badly named "carry" that is worth one if and only if the last operation returned zero.

The machine's role is to execute the programs that are given to it as parameters, generating processes. It must check that each champion calls the `live` instruction every `CYCLE_TO_DIE` cycles. If, after `NBR_LIVE` executions of the instruction `live`, several processes are still alive, `CYCLE_TO_DIE` is decreased by `CYCLE_DELTA` units. This starts over until there are no live processes left.

The last champion to have said `live` wins.

Beware, your virtual machine must be able to be built and run without a graphical environment.

Output

A number is associated to each player. This number is generated by the virtual machine and is given to the programs in the `r1` register at the system startup (all of the others will be initialized at 0, except, of course, the PC).

With each execution of the `live` instruction, the machine must display:

✓ “The player NB_OF_PLAYER(NAME_OF_PLAYER)is alive.”.

When a player wins, the machine must display:

✓ “The player NB_OF_PLAYER(NAME_OF_PLAYER)has won.”

In order to pass tests in the autograder you must respect these messages precisely. Also, the virtual machine’s options presented before will be used.

Obviously, an interesting bonus to develop would be a **graphical interface** to visualize the movement and memory acquisition of champions in the arena!

Have you seen all the graphic versions developed by your elders over the years? Every year, the students outdo themselves in their representation of Corewar!

Scheduling

The Virtual Machine simulates a parallel machine.

But for implementation reasons, it is assumed that each instruction executes entirely at the end of its last cycle and waits throughout its entire duration.
The instructions beginning on the same cycle execute according to the program’s number, in ascending order.

For instance, let’s consider 3 programs (P1, P2 and P3), each comprised of the respective instructions 1.1 1.2 .. 1.7, 2.1 .. 2.7 and 3.1 .. 3.7.
The timing of each instruction being given in the following table, then the execution order according to cycles:

P1	1.1 (4 cycles)	1.2 (5 cycles)	1.3 (8 cycles)	1.4 (2 cycles)	1.5 (1 cycle)	1.6 (3 cycles)	1.7 (1 cycle)
P2	2.1 (2 cycles)	2.2 (7 cycles)	2.3 (9 cycles)	2.4 (2 cycles)	2.5 (1 cycle)	2.6 (1 cycle)	2.7 (2 cycles)
P3	3.1 (2 cycles)	3.2 (9 cycles)	3.3 (7 cycles)	3.4 (1 cycle)	3.5 (1 cycle)	3.6 (3 cycles)	3.7 (1 cycle)

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Instruction	1.1				1.2					1.3								1.4		1.5	1.6			1.7
Instruction	2.1		2.2							2.3									2.4		2.5	2.6	2.7	
Instruction	3.1		3.2									3.3							3.4	3.5	3.6			3.7

During cycle 21, the machine executes 3 instructions in the following order: 1.6, then 2.5, then 3.6

Conclusion

For the rest, think about it and read op.h and op.c.
Your teaching assistants have also been through this, so don't be surprised if one of them looks at you wide-eyed when you ask them for help on certain bytes (spoiler: they'll know how to answer you, but might like to let you do a bit of thinking - that's the point of the project!).
Please verify that your programs are completely up to the coding style.

If you get this far, go back to page 2.

If you've arrived here again, go back to page 2 - it never hurts to reread the subject!

Here again? No but really, I assure you that rereading the subject one more time won't do you any harm.

v1