B2- Elementary Programming in C

B-CPE-201

Corewar

Back in the good old days...

v1.11

{EPITECH.}

Corewar

Back in the good old days...

binary name:	asm, corewar
repository name:	CPE_corewar_$SCOLARYEAR
repository rights:	ramassage-tek
language:	C
group size:	3-4
compilation:	via Makefile, including re, clean and fclean rules

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- 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).

The binaries must be found in their respective labeled folders, both compiled by a unique Makefile at the directory's root.

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

Authorized Functions:

- (f)open,
- read,
- write,
- getline,
- lseek,
- fseek,
- (f)close,
- malloc,
- realloc,
- free,
- exit.

The project

Introduction

Corewar is a game. A very special game. It consists of pitting little programs against one another in a virtual machine. The goal of the game is to prevent the other programs from functioning correctly by using all available means.

The game will, therefore, create a virtual machine in which the programs (written by the players) will face off. Each program's objective is to "survive", that is to say executing a special instruction ("live") that means `I'm still alive`. These programs simultaneously execute in the virtual machine and in the same memory zone, which enables them to write on one another.
The winner of the game is the last one to have executed the "live" instruction.

Search "corewars" and "redcode" on the Internet...

The different parts

The project is divided into three seperate parts:

- **The Virtual Machine**
  It houses the fighting binary programs (called champions), and provides them with a standard execution environment. It offers all kind of features that are useful to the champions' fights. It must obviously be able to execute several programs at once...

- **The Assembler**
  It enables you to write programs designed to fight (the champions). It therefore understands the assembly language and generates binary programs that the virtual machine can execute.

- **Champions**
  This is your personal handiwork. They must be able to fight and to victoriously leave the virtual machine arena. They are written in the assembly language specific to our virtual machine (described further on).

The Assembler

Introduction

The virtual machine executes machine code. But in order to write the champions, we will be using a simple language called "assembly code". It is made up of one instruction per line.
The instructions are composed of 3 elements:

- An optional label, followed by the LABEL_CHAR character (here, ":") declared in `op.h`. Labels can be any of the character strings that are composed of elements from the LABEL_CHARS string, which is also declared in `op.h`.
- An instruction code (called opcode). The instructions that the machine knows are defined in the **op_tab** array, which is delcared in `op.c`.
- Instruction parameters.

An instruction can have from 0 to MAX_ARGS_NUMBER parameters, separated by commas. Each parameter can be one of three types:

- **Register**
  From $r1$ to rREG_NUMBER
- **Direct**
  The DIRECT_CHAR character, followed by a value or a label (preceded by LABEL_CHAR), which represents the direct value. For instance, %4 or %:label
- **Indirect** A value or a label (preceded by LABEL_CHAR), which represents the value that is found at the parameter's address (in relation to PC).
  For instance, `ld 4,r5` loads the REG_SIZE bytes found at the PC+4 address into $r5$

The assembler takes such a file in assembly code (the champion) as parameter, and produce an executable for the virtual machine by converting its assembly code into machine code.

The virtual machine is BIG ENDIAN (like the Sun and unlike the i386).

Terminal

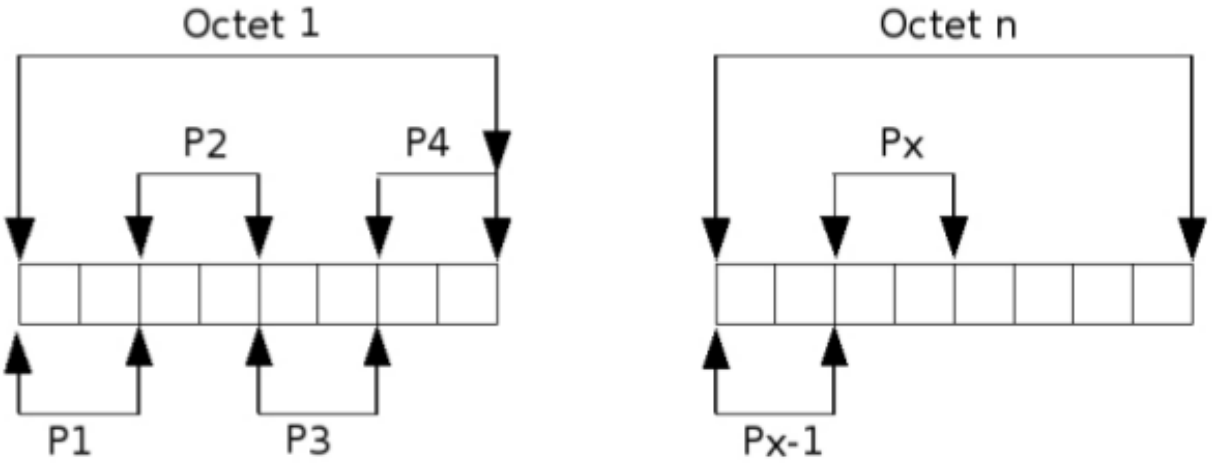
```
~/B-CPE-201> ./asm -h
USAGE
        ./asm file_name[.s]

DESCRIPTION
        file_name       file in assembly language to be converted into file_name.cor, an
                        executable in the Virtual Machine.
```

Coding

Each instruction is coded through 3 elements:

- **the instruction code**, which can be found in **op_tab** (itself in `op.h`)
- **the description of the parameters types**, called the coding byte, except for the following instructions: live, zjmp, fork, lfork.
  01 for the register, 10 for a direct and 11 for an indirect.
  The parameters are grouped 4 by 4 to form a full byte, the following way:



- **the parameters**
  1 byte for a register (its number in hexadecimal),
  DIR_SIZE bytes for a direct (its value in hexadecimal),
  IND_SIZE bytes for a indirect (its value in hexadecimal).

For instance, `r2, 23, %34` as parameters will give:
01 11 10 00, which is 0x78 as coding byte,
then 0x02 0x00 0x17 0x00 0x00 0x00 0x22 for the parameters.

`23, 45, %34` as parameters will give:
11 11 10 00, which is 0xF8 as coding byte,
then 0x00 0x17 0x00 0x2d 0x00 0x00 0x00 0x22 for the parameters.

`r1, r3, 34` as parameters will give:
01 01 11 00, which is 0x5c as coding byte,
then 0x01 0x03 0x00 0x22 for the parameters.

A full example can be found in the **Champions** section.

The Virtual Machine

Introduction

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 process 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.

```
Terminal
~/B-CPE-201> ./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
```

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.

Every "variables" (RED_NUMBER, CYCLE_TO_DIE, CYCLE_DELTA, MEM_SIZE, ...) **must** have the same value as the one in the given **op.h** file.

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:

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)

The virtual machine will execute the instructions in the following order:

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

Machine Code

The machine must recognize the instructions below (any other code have no action aside from passing to the next and losing a cycle).

Don't forget: the memory is circular and makes MEM_SIZE bytes.

The number of each instruction's cycles, their mnemonic representation, the number of parameters and the types of possible parameters are described in the **op_tab** table in `op.c`

MNEMONIC	EFFECT
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 ans 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, the third one must be a register. This operation modifies the carry and functions as follows: `ldi 3,%4,r1` reads IND_SIZ 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. It functions as follows: `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 register, 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.

Champions

Introduction

Champions are written thanks to the assembly language, described in the *The Assembler* section.
When the game, and therefore the virtual machine, starts, each champion is going to find its personal $r1$ register (the number assigned to it by the Virtual Machine).

All of the instructions are useful; all of the machine's reactions that are described in this project description are able to be used to give your champions more life, and to find an efficient strategy to win!
The `fork` instruction for instance is very useful to overwhelm your opponent. But be careful, it takes time and can become lethal if the end of CYCLE_TO_DIE comes without the machine having been able to terminate and be able to perform a "live"!
Another example: when the machine runs into an unknown opcode, what will it do? How can you reroute this behaviour?

If a champion makes a live with a number other than its own, too bad.
At least it won't be ruined for everyone...

Example

```
#
# zork.s for corewar
#
# Bob Bylan
#
# Sat Nov 10 03:24:30 2081
#
.name "zork"
.comment "just a basic living prog"

l2:
sti r1,%:live,%1
and r1,%0,r1
live: live %1
zjmp %:live
```

```
Terminal
~/B-CPE-201> ./asm zork.s && hexdump -C zork.cor
00000000 00 ea 83 f3 7a 6f 72 6b 00 00 00 00 00 00 00 00 |....zork........|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00000080 00 00 00 00 00 00 00 00 00 00 00 17 6a 75 73 74 |............just|
00000090 20 61 20 62 61 73 69 63 20 6c 69 76 69 6e 67 20 | a basic living |
000000a0 70 72 6f 67 72 61 6d 00 00 00 00 00 00 00 00 00 |program.........|
000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00000890 0b 68 01 00 0f 00 01 06 64 01 00 00 00 00 01 01 |.h......d.......|
000008a0 00 00 00 01 09 ff fb |.......|
000008a7
```

Conclusion

at last...

For the rest, think about it and read `op.h` and `op.c`.
If ever something is unclear, ask the assistants and/or the unit coordinator for more precise directions on Yammer.
Please verify that your programs are completely up to the norm.

Concerning the output messages, they are not expected to be an exact character length, but they must be relevant.
Ideally, reproduce reference binary behavior.