B2- Elementary Programming in C

B-CPE-155

Corewar

Championship

v1.1

Corewar

Championship

binary name:	my_champion.s
repository name:	CPE_$YEAR_corewar_championship
repository rights:	ramassage-tek
language:	Corewar ASM
group size:	1-2
compilation:	none

- 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).

In addition to your delivery in the repository, the SAME champion MUST be live on CodinGame at https://www.codingame.com/hackathon/corewar-2017.

During the time of the project, your name on CodinGame MUST be formatted as such:
[CTY][PROMO]firstname.lastname

The city codes are :

- Bordeaux -> BDX
- Lille -> LIL
- Lyon -> LYN
- Marseille -> MAR
- Montpellier -> MPL
- Nancy -> NCY
- Nantes -> NAN
- Nice -> NCE
- Paris -> PAR
- Rennes -> REN
- Strasbourg -> STG
- Toulouse -> TLS

Leaderboard: https://www.codingame.com/leaderboards/challenge/corewar-2017/global

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...

For now, you only have to do the **Champions** part

The different parts

The project is divided into three seperated parts:

- **The Virtual Machine**
  It houses the fighting binary programs (called champions), and provides them with an 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 generate 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 Virtual Machine

Introduction

o 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 parameter, generating processes.
It must check that each process calls the "live" instruction every CYCLE_TO_DIE cycles.
If, after NBR_LIVE calls, several processes are still alive, CYCLE_TO_DIE is decreased by CYCLE_DATA units. This starts over until there are no live processes left.

The last champion to have said "live" wins.

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 executes the program's numbers 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 he 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
live	takes 1 parameter: 4 bytes that represent the player's number. It indicates that the player is alive.
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 $r3$.
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 $r8$.
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 $r5$.
sub	Similar to `add`, but performing a subtraction.
and	takes 3 parmaters. 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 $r3$.
or	Similar to `and`, but performing a binary OR.
xor	Similar to `and`, but performing a binary XOR (exclusive OR).
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.
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 $r1$.
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.
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.
lld	Similar to `ld` without the %IDX_MOD. This operation modifies the carry.
lldi	Similar to `ldi` without the %IDX_MOD. This operation modifies the carry.
lfork	Similar to `fork` without the %IDX_MOD. This operation modifies the carry.
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` displyas '*' 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 instructins 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 for overwhelming 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"

12:
sti r1,%:live,%1
and r1,%0,r1
live: live %1
zjmp %:live
```