B2 - Elementary Programming in C

B-CPE-200

Corewar

Back in the good old days

Corewar

binary name: asm/asm, corewar/corewar
repository name: CPE_corewar_*$ACADEMICYEAR*
language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), 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).

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

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

Introduction

The Corewar project 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:

- **Champions**
  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**
  The assembler is the first of the two binaries that you are asked to do. 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 virtual machine can understand. It will therefore be necessary to understand the assembly language in order to translate it byte by byte.
- **The Virtual Machine**
  In most cases, the second binary, the virtual machine, will 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 the same machines with similar machine code, two files are made available to you: **op.c** and **op.h**.
These two files contain all the information you will need and will give you the specs of your "mini simulated computer". 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`.

CHAMPIONS

Here's what a champion might look like:

```
        .name "Jon Snow"
        .comment "Winter is coming"

        sti r1, %:crow, %1
crow:   live %234
        ld %0, r3
        zjmp %:crow
```

A champion (an .s file), consists of two parts: the header and the body.

THE HEADER

The header contains the information that will be used to display the game.

- A name
  `.name "Jon Snow"`
- A description
  `.comment "winter is coming"`

There can be comments anywhere as long as the comment starts with a `'#'`.

THE BODY

The body is a succession of lines which are for each of them a different instruction. In the example we have 4 distinct instructions like `"sti r1, %:crow, %1"`. These instructions are themselves decomposable into 3 sub-parts:

- There are so-called **instruction codes** (called **opcode**), which allow to define which instruction to execute.
  `"sti", "live", "ld", "zjmp"`
- And the **parameters** that go with them. Each instruction depends on parameters to be used.
  `"r1, %:crow, %1", "%234", "ld %0, r3", "%:crow"`
- But there are also **labels**. followed by the `LABEL_CHAR` character (here, `':'`). Labels can be any of the character strings that are composed of elements from the `LABEL_CHARS` string. It is a kind of function that allows to encompass several instructions. It serves as a coordinate point in the instructions of a champion.
  `"crow:"`

THE INSTRUCTIONS

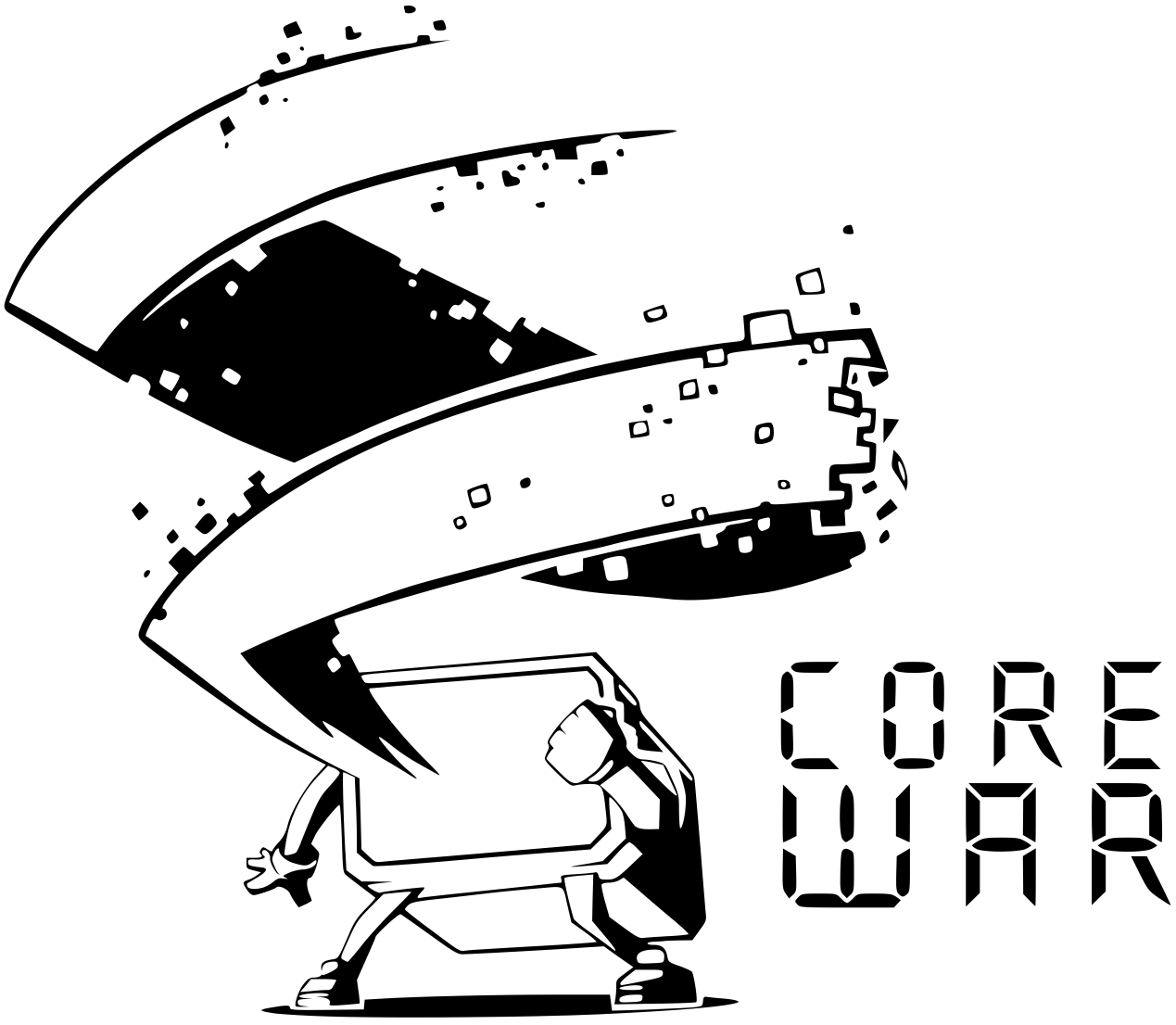
In our Corewar, the champions have the possibility of executing one of our 16 instructions (detailed and established in the op.c). Below you will find a table explaining their parameters and operation.
Don't worry ! You still don't understand everything written there, but the following will help you to understand them!

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 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_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. `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.



It is not up to you to write your own champions! That's could be a great bonus and help you to understand their behavior but champions are already provided in project side files

THE ASSEMBLER

```
Terminal
~/B-CPE-200> ./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.
```

Now that you know what a champion is made of, you will have to transcribe them into a language that the virtual machine can understand, i.e. into machine code.
To do this, you will have to translate line after line (instruction after instruction).

PARAMETERS

As seen before, the structure of an instruction is thus: **<opcode> <parameters>**. It seems that the opcode is understood, it is the instruction to be executed, but what does the parameters that follow mean? An instruction can have from 0 to `MAX_ARGS_NUMBER` parameters, separated by commas.
These parameters can be of 3 different types:

- **Register**
  Each champion will have to `REG_NUMBER` registers in which it can store integers (`r1` to rREG_NUMBER). Besides, the first register (`r1`) will contain the identifier of the champion in question.
- **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).

So now you understand `sti r1, %:crow, %1`? The instruction `sti` is executed with the direct value of :crow and 1.

For your interest, a register is a memory location internal to a processor, it is the fastest memory in a computer.

Example

Before continuing further, here is what the champions should look like once transcribed.

Using the command `"$ hexdump -C <file>"` will come to you as frantically as the revelio spell in Hogwarts Legacy.

Terminal

```
~/B-CPE-200> ./asm jon.s && hexdump -C jon.cor
00000000 00 ea 83 f3 4a 6f 6e 20 53 6e 6f 77 00 00 00 00 |....Jon Snow....|
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 16 57 69 6e 74 |............Wint|
00000090 65 72 20 69 73 20 63 6f 6d 69 6e 67 00 00 00 00 |er is coming....|
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00000890 0b 68 01 00 07 00 01 01 00 00 00 ea 02 90 00 00 |.h..............|
000008a0 00 00 03 09 ff f4 |......|
000008a6
```

Transcription

With hexdump, you have the ability to see each of the bytes transcribed by the assembly. For example you can see that the 4th byte `4a` is the hexadecimal value of the character `J`, the same `J` as in `Jon Snow`! This means two things:

- It is possible to take byte after byte a .cor file with its original .s file to understand how the transcription is made (to understand the particular cases it is very practical!)
- Before transcribing the instructions, the header (with name and comment) must be transcribed using the `header_t` structure provided in op.h.

Each instruction is coded through 3 elements:

- **the instruction code** (opcode)
- **the coding byte**, which is a value to describe the type of the parameters that follow (after all, once in machine code, how do you know what type the parameters are?) The byte coding will not be present for the `live`, `zjmp`, `fork` and `lfork` instructions since they have only one parameter which will always be of the same type
- **the parameters**

But how to write them?

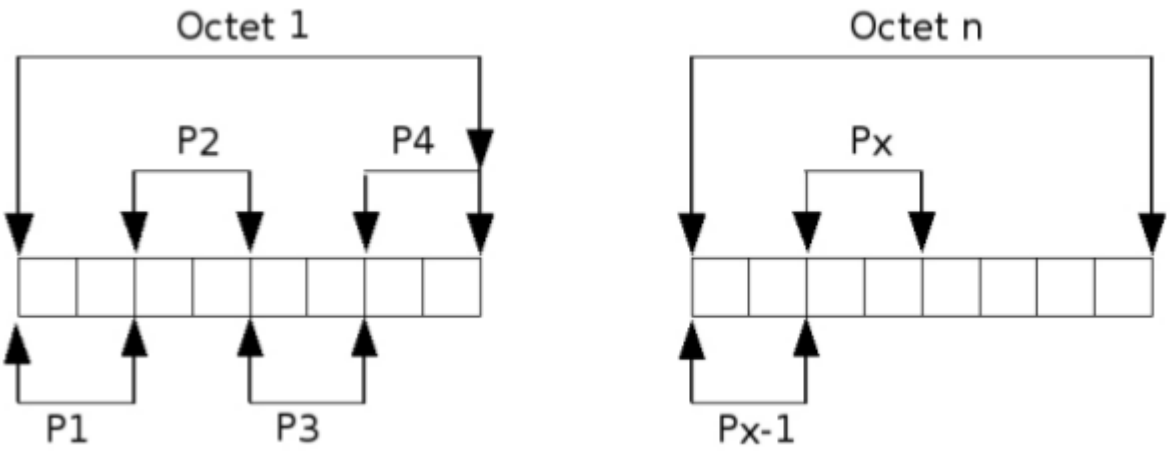
Instruction code

The instruction code can be found in `op_tab` in op.h. This instruction code is written on a single byte. This will give `0x0b` for the `sti` instruction in our example.

Coding byte

The byte coding is calculated according to the following parameters. In a byte we will fill each of its 8 bits from the nature of the parameters. Bits 1 and 2 will be the type of parameter 1, bits 3 and 4 the type of parameter 2 etc.

- If the parameter is a register: 01
- If the parameter is a direct: 10
- If the parameter is an indirect: 11
- Otherwise: 00



The whole thing then forms a single byte ready to be written. This will give for example :
- The parameters `r2, 23, %34` will give: `01 11 10 00` and therefore worth: `0x78`
- The parameters `23, 45, %34` will give: `11 11 10 00` and therefore worth: `0xf8`
- The parameters `r1, r3, 34` will give: `01 01 11 00` and therefore will be worth: `0x5c`

Parameters

The parameters are transcribed one by one by writing their value directly on a number of bytes equivalent to their type.

- The value is written (in hexadecimal) on 1 byte for a register
- The value is written (in hexadecimal) on `DIR_SIZE` bytes for a direct
- The value is written (in hexadecimal) on `IND_SIZE` bytes for indirect

Using the examples from before:
- The parameters `r2, 23, %34` will give: `0x02`, `0x00 0x17` and `0x00 0x00 0x00 0x22`
- The parameters `23, 45, %34` will give: `0x00 0x17`, `0x00 0x2d` and `0x00 0x00 0x00 0x22`
- The parameters `r1, r3, 34` will give: `0x01`, `0x03` and `0x00 0x22`

What happened when the parameter is a label like `%:crow`? You should let's compare .s and .cor file to see the result!

Those examples are only valid when none of the parameters are indexes

Did you say indexes?

Instructions will sometime describe the parameters they take as `indexes`. This does not change the rules for the description of the parameters type but the size of the parameter itself might change.
Indexes' size will always be `IND_SIZE` bytes, even when the parameter is a direct.
Read the Instructions section carefully to identify the function that have indexes as parameters.

The virtual machine is **BIG ENDIAN** (like the Sun and unlike the i386). This changes the order of storage, and thus perhaps the way you should transcribe your information

.

THE 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 virtual machine will be evaluated by your pedagos and not in automated tests. 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.

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

Conclusion

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.
Please verify that your programs are completely up to the coding style.
Concerning the output messages, they are not expected to be an exact character length, but they must be relevant. Ideally, reproduce reference binary behavior.