Elementary programming

Corewar

Astek in charge astek_resp@epitech.eu

Abstract: This document is the subject of the Corewar Elementary programming project

Contents

Chapter I

Administrative details

- Turn-in on the team leader's account:
  `svn+ssh ://kscm@koala-rendus.epitech.net/corewar-2016ed-2015s-login_x`
- Your executables must be compiled by a Makefile
- Your executables must be named `asm` and `corewar`, and be placed in their respective subdirectories.

`Pay attention to the permissions of your files and directories ...`

Chapter II

Information

- You can find on the "corewar" account an example of the virtual machine and the assembler, both available on `/u/all/corewar/public/bin/$HOSTTYPE/`
(`$HOSTTYPE` representing the architecture of the machine you wish to run the program on).
Example : `/u/all/corewar/public/bin/sun4/AsmX-4.2` on Sun.

- You can compile the champions located in `/u/all/corewar/public/champ/old/src`, or maybe run the `.cor` in `/u/all/corewar/public/champ/old`

- Details of the graphical virtual machine version 4.2
It's the virtual machine that will be used for the championship. It has certain specific characteristics :

  - #define CYCLE_TO_DIE 1536
  - #define CYCLE_DELTA 4
  - #define NBR_LIVE 2048
  - cycle numbers are sometimes different for some instructions
  - one can run 1 to 4 players
  - various options (try running it with no parameters)
  - a system that blocks processes in their own initial memory space for the N first cycles ("-ctmo" option)
  - a "gtmd" instruction, that takes a parameter encoding byte,
sole parameter : a register number
action: puts the number of cycles remaining before "opening" the ram to everyone in the designated register
  - You can implement, if you want to, the process locking system with the associated instruction (this IS optional ...)

Chapter III

Introduction

III.1 What the hell is that thing ?

- The Corewar is a game, a very special game. It consists in making small programs fight in a virtual machine.
- The goal of the game is to prevent other programs from functioning correctly, by any means necessary.
- The game will create a virtual machine in which programs (written by the players) fight. The objective of each process is to "survive". By "survive", we means execute a special instruction (live) that means "hey, i'm still alive !". These programs run simultaneously in the virtual machine, and in the same memory space, so they can write on each other. The winner of the game is the last program who ran the "live" instruction.

III.2 And how does that work ?

- The project will be split in three parts :
  - **The assembler** : It will allow you to write programs destined to fight. It will need to understand the assembly language, and generate binary programs that can be interpreted by the virtual machine.
  - **The virtual machine** : It will house the binary programs (the champions) and provide them with a standard execution environment. It offers a lot of features useful to the fight. It goes without saying that it has to be able to run multiple programs simultaneously (or the fights will be really, really dull ...)

- **The champion** : It's your own personal masterpiece. It has to fight and be the winner of the arena that is the virtual machine. It will be written in the assembly language needed to run on our virtual machine (described later in the subject).

Chapter IV

The virtual machine

- The virtual machine is a multi-program machine. Each program has :
  - REG_NUMBER registers that are all REG_SIZE bytes large. A register is a small memory containing only one value. In a real machine, it's internal to the processor, and thus very fast to access. REG_NUMBER and REG_SIZE are defines available in op.h.
  - A PC (Program Counter) It's a special register that contains the current address in memory where the next intruction to decode and execute is located. Very useful if you want to know where you are located and write things in memory.
  - A flag named "carry" (Yeah, that's a fitting name), that contains one if the last instruction returned zero.
- The role of the virtual machine is to run the programs that are passed as parameters to it.
- It must check that every process calls the "live" instruction every CYCLE_TO_DIE.
- If after NBR_LIVE calls to the "live" instruction all the processes alive still are alive,
- We decrement CYCLE_TO_DIE of CYCLE_DELTA units and we start again until there are no more processes alive
- The last player who said "live" wins
- The machine must then display : "player X(player_name) has won" where X is the player number and player_name the name.
- Example :

"player 3(zork) has won"

At each execution of the "live" instruction the machine must display :

"player X(player_name) is alive"

The player number is generated by the machine and is passed to the programs in register r1 at the process start (all the others will be set to 0, except of course for the PC)

Chapter V

Command line syntax

- SYNOPSIS
  corewar [-dump nbr_cycle] [[-n prog_number] [-a load_address] prog_name] ...

- DESCRIPTION

  - -dump nbr_cycle
    Dumps the memory after nbr_cycle execution cycles (if the game isn't already over) at 32 bytes per line, coded in hexadecimal : A0BCDEFE1DD3..... once the memory is dumped, we exit.

  - -n prog_number
    Sets the number of the next program. By default, it will be the next available number, in parameter order.

  - -a load_address
    Sets the load address of the next program. When no address is specified, we'll choose the addresses so that the programs are the farthest possible from each other. Addresses are modulo MEM_SIZE.

  - In the case of a syntax error of any kind, return an error message.

Chapter VI

Machine code

- The machine must recognize the following instructions :

Mnemonic	Effects
0x01 (live)	Followed by 4 bytes representing the player name. This instruction indicates that the player is alive. (No parameter encoding byte).
0x02 (ld)	This instruction takes 2 parameters, the 2nd of which has to be a register (not the PC) It loads the value of the first parameter in the register. This operation modifies the carry. ld 34,r3 loads the REG_SIZE bytes from address (PC + (34 % IDX_MOD)) in register r3.
0x03 (st)	This instruction takes 2 parameters. It stores (REG_SIZE bytes) the value of the first argument (always a register) in the second. st r4,34 stores the value of r4 at the address (PC + (34 % IDX_MOD)) st r3,r8 copies r3 in r8
0x04 (add)	This instruction takes 3 registers as parameter, adds the contents of the 2 first and stores the result in the third. This operation modifies the carry. add r2,r3,r5 adds r2 and r3 and stores the result in r5
0x05 (sub)	Same effect as add, but with a substraction
0x06 (and)	p1 & p2 -> p3, the parameter 3 is always a register This operation modifies the carry. and r2, %0,r3 stores r2 & 0 in r3.
0x07 (or)	Same as and but with an or (| in C)
0x08 (xor)	Same as and but with an xor (în C)
0x09 (zjmp)	This instruction is not followed by any parameter encoding byte. It always takes an index (IND_SIZE) and makes a jump at this index if the carry is set to 1. If the carry is null, zjmp does nothing but consumes the same amount of time. zjmp %23 does : If carry == 1, store (PC + (23 % IDX_MOD)) in the PC.
0x0a (ldi)	This operation modifies the carry. ldi 3,%4,r1 reads IND_SIZE bytes at address: (PC + (3 % IDX_MOD)), adds 4 to this value. We will name this sum S. Read REG_SIZE bytes at address (PC + (S % IDX_MOD)), which are copied to r1. Parameters 1 and 2 are indexes.
0x0b (sti)	sti r2,%4,%5 sti copies REG_SIZE bytes of r2 at address (4 + 5) Parameters 2 and 3 are indexes. If they are, in fact, registers, we'll use their contents as indexes.

0x0c (fork)	This instruction is not followed by a parameter encoding byte. It always takes an index and creates a new program, which is executed from address : (PC + (first parameter % IDX_MOD)). Fork %34 creates a new program. The new program inherits all of its father's states.
0x0d (lld)	Same as ld, but without the % IDX_MOD This operation modifies the carry.
0x0e (lldi)	Same as ldi, without the % IDX_MOD This operation modifies the carry.
0x0f (lfork)	Same as fork, without the % IDX_MOD This operation modifies the carry.
0x10 (aff)	This instruction is followed by a parameter encoding byte. It takes a register and displays the character the ASCII code of which is contained in the register. (a modulo 256 is applied to this ascii code, the character is displayed on the standard output) Ex: ld %52,r3 aff r3 Displays '*' on the standard output

- Every address is relative to the PC, IDX_MOD except for "lld", "lldi" and "lfork".
- In any case, the memory is circular and is MEM_SIZE bytes in size.
- The number of cycles of each instruction, their mnemonic representation, and the number and possible types of parameters are described in the op_tab array declared in op.c.
- Any other code has no action, except going to the next one and losing a cycle.

Chapter VII

Scheduling

- The virtual machine is supposed to emulate a perfectly parralel machine.
- For implementation reasons, we will assume that every instruction is executed at the end of its last cycle, and waits for its entire duration. Instructions that start at the same cycle execute in ascending program number order (cf. figure)
- Example:
  Consider three programs (P1, P2, P3) respectively constituted of instructions 1.1 1.2 .. 1.7 , 2.1 .. 2.7, 3.1 .. 3.7 . The timings of each instruction being given in the following table:

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

- The virtual machine will execute them in the following order :

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
Instructions	1,1				1,2					1,3			
Instructions	2,1		2,2							2,3			
Instructions	3,1		3,2									3,3	

Cycles	14	15	16	17	18	19	20	21	22	23	24	25
Instructions					1,4		1,5	1,6			1,7	
Instructions						2,4		2,5	2,6	2,7		
Instructions						3,4	3,5	3,6				3,7

`At cycle 21, the machine executes 1.6 (instruction 6 of program 1) then instruction 2.5 (instruction 5 of program 2) then 3.6.`

Chapter VIII

The assembler

The machine executes machine code. But to write the programs, we will use a simple language called assembly. It is composed of an instruction per line. Instructions are composed of three elements :

- An optional label followed by character LABEL_CHARS (here ":") declared in op.h.
- Labels can be any string composed of elements from the string LABEL_CHARS declared in op.h.
- An instruction code (opcode). (Instructions that the machine knows are defined in the array op_tab declared in op.c.)
- The instruction's parameters

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

- Register : r1 <–> rx with x = REG_NUMBER
  Example : ld r1,r2 (load r1 in r2)
- Direct : The character DIRECT_CHAR followed by a value or a label (preceded by LABEL_CHARS), which represents a direct value.
  Example : ld $4,r5 (load 4 in r5)
  Example : ld %:label, r7 (load label in r7)
- Indirect : A value or a label (preceded by LABEL_CHARS) which represents the value contained at the address of the parameter, relative to the PC.
  Example : ld 4,r5 (load the 4 bytes at address (4+PC) in r5).

Chapter IX

Command line syntax

- SYNOPSIS
  asm file_name[.s] .....

- DESCRIPTION
  transforms file_name[.s] in file_name.cor (an executable for the virtual machine).

Chapter X

Encoding

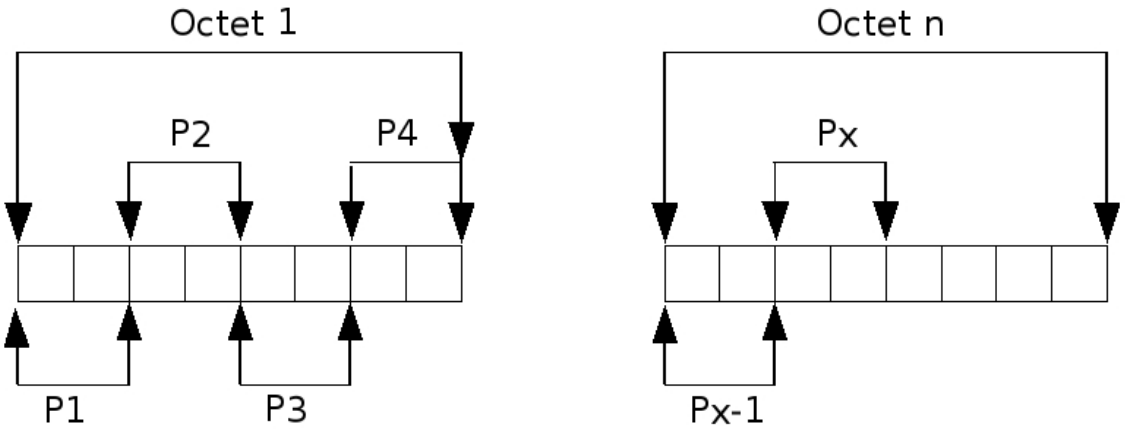
The encoding is rather simple:

Each instruction is encoded by the instruction code, the description of the types of the parameters, then the parameters.

- The instruction code (found in op_tab which is in op.h).
- The description of the type of the parameters (see figure). For the live, zjmp, fork and lfork instructions, it is not there.
- Example of parameter encoding byte :

r2,23,%34	Will give the following encoding byte : 01 11 10 00, so 0x78
23,45,%34	Will give the following encoding byte : 01 11 10 00, so 0xF8
r1,r3,34	Will give the following encoding byte : 01 01 11 00, so 0x5C

Parameter type encoding :



01 Register, followed by a byte (the register number)
10 Direct, followed by DIR_SIZE bytes (the direct value)
11 Indirect, followed by IND_SIZE bytes (the value of the indirection)

- The parameters.

After the parameter types, we will directly put the parameters :

- for a register, its number on a byte
- for a direct, its value on DIR_SIZE bytes
- for an indirect, its value on IND_SIZE bytes

R2,23,%34	Will give 0x78 then 0x02 0x00 0x17 0x00 0x00 0x00 0x22
23,45,%34	Will give 0xF8 then 0x00 0x17 0x00 0x2d 0x00 0x00 0x00 0x22

Example :

```
#
# ex.s for corewar
#
# Alexandre David
#
Sat Nov 10 22:24:30 2201
#
.name "zork"
.comment "just a basic living prog"

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

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

- The program must, from the assembly file passed as a command-line parameter, produce an executable for the virtual machine. This executable starts by the header defined in the header_t type in op.h

- Important note : The virtual machine is BIG ENDIAN (Like a Sun, and not an i386 for example).

Chapter XI

Modus operandi of a champion

- One objective : There can be only one left.
- At the beginning of the game, each champion will find in its r1 register the number that it has been alloted. It will have to use it for every "live" instruction.
- If a champion does a "live" with a number that is not its own ... Well, tough luck. At least that's not a loss for everyone !
- You should look at the code given as example in the instruction encoding part, it's very useful.
- Every instruction is useful, every reaction of the machine described in the subject is exploitable to give life to your champion.
- For example, when the machine encounters an unknown opcode, what does it do ? How could you use that to your advantage ?
- Look at the /u/all/corewar/public/ directory, there are champions from previous years. Use them as inspiration !
- Note that the machine has a "fork" instruction, very useful to overwhelm your adversaries, but it takes time and can become fatal if the end of the CYCLE_TO_DIE happens before it has a chance to finish and allow for a "live" !

Chapter XII

Messages

The following messages do not have to be respected to the letter (You'll be graded by a human ...), but they have to stay on topic. (See the examples for the machine and the assembler).

XII.1 For the machine

1. "file_name is not a corewar executable" (where file_name is the name of an argument)
2. "prog number the_number already used" (where the_number is the number requested).
3. "player x(nom_du_joueur) is alive"
4. "player x(nom_du_joueur) has won"

Messages 1 and 2 have to stop the program.

XII.2 For the assembler

1. "Syntax error line x" (Where x is the line number, first line is 1)
2. "Warning Indirection to far line x" (indirection > IDX_MOD)
3. "label the_label undefine line x"
4. "no such register line x"
5. "Warning Direct too big line x"

Messages 1, 3 and 4 will stop the program.

XII.3 For the both of them

1. "File file_name not accessible"

2. "Can't perform malloc"

Both these messages will stop the program.

If you need other messages, ask your assistants, new messages and every relevant changes will be posted in the forum's relevant section. All notifications on this forum will be considered part of the subject and you have to take them into account.

Chapter XIII

Conclusion

- For the rest, think, read op.h and op.c, and if something seems ambiguous, ask your assistants or on the forum.
- We stronhly urge you to check that all your programs comply with the norm defined in the lecture.
- Good luck, work well !

Chapter XIV

Allowed functions

- open
- read
- write
- lseek
- close
- malloc
- realloc
- free
- exit