QWASAR
Silicon Valley

Qwasar Master's of Science in Computer Science Programs
Copyright 2023
Engineering Lab 2

Problem Statement

Corewar consists of programs that run in a virtual machine. Each program's objective is to execute a special instruction ("live"), given each program task orientation. These programs simultaneously execute in the virtual machine and in the same memory zone, which enables them to write on one another.

The project must be coded in C - 😎

The project is divided into three separate parts:

Virtual Machine

It's the sandbox environment where the programs will run. It must offer all features that are useful for programs to run. It must be able to execute multiple programs.

Assembler

It's the core of syntax and rules in order to make a valid program. It must be able to generate binary code executable by the Virtual Machine.

Programs

Must be able to run and leave the virtual machine. They are written in the assembly language specific to the virtual machine.

Solution

Virtual Machine

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

REG_NUMBER registers of REG_SIZE byte search

A register is a memory zone that contains only one value. In a real machine, it is embedded within the processor, and can be accessed very fast. REG_NUMBER and REG_SIZE are defined in op.h.

A C(Counter)

This register contains the memory address (in the virtual machine) of the next instruction to be decoded and executed. This is useful to know where you are and to write things in memory.

Carry Flag

This flag is worth one if, and only if, the last operation returned zero.

The machine's role is to execute programs passed as parameters, generating processes. It must check that each process calls the "live" instruction every CYCLE_TO_DIE cycle. 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.

```
Unset

~/QWASAR-MSCS>./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
```

Output

Each program has a number. This number is generated by the virtual machine and is given to the programs in the r1 register at the machine startup (all of the others will be initialized at 0). With each execution of the "live" instruction, the machine must display "The program NBR_OF_PROGRAM(NAME_OF_PROGRAM) is alive." When a player wins, the machine must display "The player NBR_OF_PROGRAM (NAME_OF_PROGRAM) is done.".

Scheduling

The Virtual Machine emulates a parallel machine. As an implementation constraint, it is assumed that each instruction executes entirely at the end of its last cycle and waits for its entire duration. The instructions that start on the same cycle are executed according to the program's number, in ascending order.

Let's consider 3 programs (P1, P2 and P3), each consisting of the respective instructions 1.1 1.2 .. 1.7, 2.1 .. 2.7 and 3.1 .. 3.7. The timing of each instruction would look like this:

Instruction cycles

P1	1.1 1 cycles	1.2 5 cycles	1.3 1 cycles	1.4 1 cycle	1.5 2 cycles
P2	2.1 3 cycles	2.2 1 cycles	2.3 5 cycles	2.4 1 cycles	
P3	3.1 3 cycles	3.2 2 cycles	3.3 3 cycles	3.4 2 cycles	

Virtual Machine Order of execution

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
P1	1.1	1.2					1.3	1.4	1.5	
P2	2.1			2.2	2.3					2.4
P3	3.1			3.2		3.3			3.4	

Machine Code

The machine must recognize the instructions below. If not a valid command, the Virtual machine moves to the next instruction.

MNEMONIC	EFFECT
0X01 (live)	Takes 1 parameter: 4 bytes that represent the player's number. Indicates the player is alive.
0x02 (ld)	Takes 2 parameters: loads the first parameter to the second parameter. Second parameter is a register. Ex: **ld 34, r3** loads the REG_SIZE bytes starting from the Counter + 34 % IDX_MOD into **r3**.
0x03 (st)	Takes 2 parameters: Stores first parameter (a register) into the second parameter (it can be another register or a number). Ex: **st r4, 34** stores r4 at the address of Counter + 34 % IDX_MOD; **st r3, r8** copies **r3** into **r8.**
0x04 (add)	Takes 3 parameters: 3 registers. Add the first to the second, and store the result to the third. Modifies the carry.
0x05 (sub)	Same as **add**, but Subtracting. Modifies the carry.
0x06 (and)	Same as **add**, and **sub,** but does a binary operation AND between the first and the second, and storing the result in the third parameter. Modifies the carry.
0x07 (or)	Same as **and**, but performing an **OR.**
0x08 (xor)	Same as **and** and **or**, but performing an **XOR.**
0x09 (zjmp)	Takes 1 parameter, that must be an index. Jumps to this index if carry is 1. Else, does nothing, but still consume time. *Zjmp %23* -> if carry == 1, Counter + 23 % IDX_MOD to Counter
0x0a (ldi)	Takes 3 parameters. First two are indexes and the third one is a register Ex: *ldi 3, %4, r1* -> reads IND_SIZ bytes from address Counter + 3 % IDX_MOD, add 4 to this value (SumResult). REG_SIZE byte are read from the Counter + SumResult % IDX_MOD and copies to *r1.*
0x0b (sti)	Takes 3 parameters. The first one is a register. The other two can either be indexes or registers. Ex: *sti r2, %4, %5* -> copies the content for *r2* into Counter + (4 + 5) % IDX_MOD.
0x0c (fork)	Takes 1 parameter, an index. Creates a new program, inheriting states from the parent, and starting execution at COunter + parameter % IDX_MOD
0x0d (lld)	Same as **ld** without the %IDX_MOD. Modifies the carry
0x0e (lldi)	Same as **ldi** without the %IDX_MOD. Modifies the carry
0x0f (lfork)	Same as **fork** without the %IDX_MOD.
0x10 (aff)	Takes 1 parameter, a register. Displays to stdout the character corresponding to the ASCII code of the content of the register (in base 10). A 256 modulo is applied to this ASCII code. Ex; a*ff r3* -> outputs '*' if r3 contains 42.

Assembler

The virtual machine job is to execute machine code. The assembler is responsible for setting the rules and syntax for writing the programs. We shall call it simply GAC (General Assembly Code). Instructions are made of 3 elements:

- **Label**

  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.

- **Opcode**

  An instruction code. The instructions that the machine knows are defined in the op_tab array, which is declared in op.c.

- **Parameters**

Instructions can have from 0 to MAX_ARGS_NUMBER parameters, separated by comma. There are three types available:

- **Register**

  From *r1* to *r*REG_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 the **C**ounter). For example: **ld 4,r5** loads the REG_SIZE bytes found at the **C**ounter+4 address into **r5**.

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

```
Unset
~/QWASAR-MSCS>./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 composed of three elements:

- **Instruction code**
- **Description of parameter types**, except for the following instructions: **live, zjmp, fork, fork.**

  01 - register
  10 - direct
  11 - indirect

- **Parameters**
  1 byte for register (in hexadecimal)
  DIR_SIZE bytes for direct (in hexadecimal)
  IND_SIZE bytes for indirect (in hexadecimal)

  Parameters are grouped 4 by 4, to form a full byte.

Examples:

sti r2, 23, %34

- Instruction code: 00 00 10 11 = 0x0b (Hex. representation)
- Param. Description: 01 11 10 00 = 0x78 (Hex. representation)
  r2 23 %34
- Values: 0x02 0x00 0x17 0x00 0x00 0x00 0x22
  | r2 | 23 | %34 |
- Full Instruction (in Hexadecimal): 0b 78 02 00 17 00 00 00 22

Programs

Programs are written in GAC (General Assembly Code) assembly language, described in the **The Assembler** section. When the virtual machine starts, each program is going to find its personal r1 register (the number assigned to it by the Virtual Machine).

Scenarios

Simple

```
Unset
.name "Simple"
.comment "Let's get started"

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

live:  live %1
              zjmp %:live
```

Complex

```
Unset
.name "Complex"
.comment "Let's do it!"

sti    r1,%:live,%1
sti    r1,%:live2,%1
ld     %1,r3
ld     %33,r6
#While (r2 < 10)
forks:
add    r2,r3,r2            #increment r2
xor    r2,%15,r4           #if (r4) {carry = 0}
live2:
       live   %4
zjmp   %:endwhile          #if (carry)
fork   %:forks
ld     %0,r4               #carry = 1
```

```
zjmp   %:forks
#EndWhile
endwhile:
ld     %0,r4               #carry = 1

live:
live %4
zjmp %:live
```

op.h

```
Unset
#ifndef _OP_H_
#define _OP_H_

#define MEM_SIZE (6 * 1024)
/* modulo of the index < */
#define IDX_MOD 512
/* this may not be changed 2^*IND_SIZE */
#define MAX_ARGS_NUMBER 4

#define COMMENT_CHAR '#'
#define LABEL_CHAR ':'
#define DIRECT_CHAR '%'
#define SEPARATOR_CHAR ','

#define LABEL_CHARS "abcdefghijklmnopqrstuvwxyz_0123456789"

#define NAME_CMD_STRING ".name"
#define COMMENT_CMD_STRING ".comment"

/* r1 <--> rx */
#define REG_NUMBER 16

typedef char args_type_t;
typedef unsigned char code_t;
```

```
enum parameter_types {
  T_REG = 1,
  T_DIR = 2,
  T_IND = 4,
  T_LAB = 8
};

typedef struct champion champion_t;
typedef struct core_s core_t;

struct op_s {
  char *mnemonique;
  char nbr_args;
  args_type_t type[MAX_ARGS_NUMBER];
  char code;
  int nbr_cycles;
  int (*inst)(champion_t *, core_t *, code_t, int *);
};

enum op_types {
  OP_LIVE,
  OP_LD,
  OP_ST,
  OP_ADD,
  OP_SUB,
  OP_AND,
  OP_OR,
  OP_XOR,
  OP_ZJMP,
  OP_LDI,
  OP_STI,
  OP_FORK,
  OP_LLD,
  OP_LLDI,
  OP_LFORK,
  OP_AFF,
  OP_NOTHING,
  OP_NB
};
typedef struct op_s op_t;

/* size (in bytes) */
#define IND_SIZE 2
#define DIR_SIZE 4
#define REG_SIZE DIR_SIZE
```

```
/* op_tab */
extern const op_t op_tab[];

/* HEADER */
#define PROG_NAME_LENGTH 128
#define COMMENT_LENGTH 2048
#define COREWAR_EXEC_MAGIC 0xea83f3

typedef struct header_s {
  int magic;
  char prog_name[PROG_NAME_LENGTH + 1];
  int prog_size;
  char comment[COMMENT_LENGTH + 1];
} header_t;

/* live */

#define MAX_CHAMPIONS 4
/* number of cycle before being declared dead */
#define CYCLE_TO_DIE 1536
#define CYCLE_DELTA 5
#define NBR_LIVE 40

#endif
```

op.c

Unset

```
#include "corewar.h"

const op_t op_tab[] = {
  {"live", 1, {T_DIR}, 1, 10, inst_live},
  {"ld", 2, {T_DIR | T_IND, T_REG}, 2, 5, inst_ld},
  {"st", 2, {T_REG, T_IND | T_REG}, 3, 5, inst_st},
  {"add", 3, {T_REG, T_REG, T_REG}, 4, 10, inst_add},
  {"sub", 3, {T_REG, T_REG, T_REG}, 5, 10, inst_sub},
  {"and", 3, {T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG}, 6, 6,
inst_and},
  {"or", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 7, 6,
inst_or},
  {"xor", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 8, 6,
inst_xor},
```

```
  {"zjmp", 1, {T_DIR}, 9, 20, inst_zjmp},
  {"ldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 10, 25, inst_ldi},
  {"sti", 3, {T_REG, T_REG | T_DIR | T_IND, T_DIR | T_REG}, 11, 25, inst_sti},
  {"fork", 1, {T_DIR}, 12, 800, inst_fork},
  {"lld", 2, {T_DIR | T_IND, T_REG}, 13, 10, inst_lld},
  {"lldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 14, 50,
inst_lldi},
  {"lfork", 1, {T_DIR}, 15, 1000, inst_lfork},
  {"aff", 1, {T_REG}, 16, 2, inst_aff},
  {0, 0, {0}, 0, 0, 0}
};
```

References

- [CoreWar - Origins]

Authorized Functions

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