Nom des binaires : asm, corewar
Nom du repo : CPE_corewar_2018
Droits du repo : ramassage-tek
Langage : C
Taille du groupe : 3 – 4
Compilation : via Makefile, règles re, clean et fclean

Fonction	Fichier à include	man
open	fcntl.h	https://linux.die.net/man/2/open
fopen	stdio.h	https://linux.die.net/man/3/fopen
read	unistd.h	https://linux.die.net/man/2/read
write	unistd.h	https://linux.die.net/man/2/write
getline	stdio.h	https://linux.die.net/man/3/getline
lseek	unistd.h	https://linux.die.net/man/2/lseek
fseek	stdio.h	https://linux.die.net/man/3/fseek
close	unistd.h	https://linux.die.net/man/2/close
fclose	stdio.h	https://linux.die.net/man/3/fclose
malloc	stdlib.h	https://linux.die.net/man/3/malloc
realloc	stdlib.h	https://linux.die.net/man/3/realloc
free	stdlib.h	https://linux.die.net/man/3/free
exit	stdlib.h	https://linux.die.net/man/3/exit

Le projet

Introduction

Le Corewar est un jeu dans lequel doit s'affronter deux programmes sur une machines virtuelles.
Le but est d'empêcher le programme adverse de fonctionner correctement.

L'objectif est de survivre, d'éxécuter l'instruction « live », sur une machine virtuelle où les deux programmes conccurents se partagent la même mémoire, il est donc possible pour les programmes d'écrire l'un sur l'autre, ou sur lui-même.

Le gagnant est le dernier qui a executé l'instruction « live ».

Voici un lien de documentation sur Redcode

Et voici la vidéo d'une filnale de Corewar dans une autre école

Les différentes parties

Le projet est séparé en trois parties

- **La machine virtuelle**
  Elle héberge le combat entre les deux programmes (champions), et leur fournit un environnement standard d'execution (cycles, execution des champions, jeu d'instruction). Elle doit pouvoir éxecuter plusieurs programmes en même temps.

- **L'assembleur**
  Permet d'écrire les champions, il génère des binaires que la machine virtuelle peut éxecuter grâce au code (pseudo)assembleur.

- **Les champions**
  Les champions sont les programmes qui vont se battre dans la machine virtuelle, ils sont codés avec le langage spécifique de la machine virtuelle (décrit plus bas).

L'assembleur

Introduction

La machine virtuelle execute du code spécifique au Corewar, appelé « code assembleur » (ce nom sera utilisé dans tout le reste du document à partir d'ici). Le code assembleur se compose de 3 éléments :

- Une « étiquette » optionnelle, suivi du character **LABEL_CHAR** déclaré dans *op.h*

```
19. # define LABEL_CHAR            ':'
```

Le **LABEL_CHAR** peut être n'importe quel character de la suite de caractère **LABEL_CHARS**, aussi déclaré dans *op.h.*

```
23. # define LABEL_CHARS           "abcdefghijklmnopqrstuvwxyz_0123456789"
```

- Une instruction (appelée opcode). Les instructions que la machine virtuelle connaît et qui sont définies dans le tableau **op_tab** donné dans le fichier *op.c*

```
13. op_t    op_tab[] =
14.   {
15.     {"live", 1, {T_DIR}, 1, 10, "alive"},
16.     {"ld", 2, {T_DIR | T_IND, T_REG}, 2, 5, "load"},
17.     {"st", 2, {T_REG, T_IND | T_REG}, 3, 5, "store"},
18.     {"add", 3, {T_REG, T_REG, T_REG}, 4, 10, "addition"},
19.     {"sub", 3, {T_REG, T_REG, T_REG}, 5, 10, "soustraction"},
20.     {"and", 3, {T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG}, 6, 6,
21.      "et (and  r1, r2, r3   r1&r2 -> r3"},
22.     {"or", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 7, 6,
23.      "ou  (or   r1, r2, r3   r1 | r2 -> r3"},
24.     {"xor", 3, {T_REG | T_IND | T_DIR, T_REG | T_IND | T_DIR, T_REG}, 8, 6,
25.      "ou (xor  r1, r2, r3   r1^r2 -> r3"},
26.     {"zjmp", 1, {T_DIR}, 9, 20, "jump if zero"},
27.     {"ldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 10, 25,
28.      "load index"},
29.     {"sti", 3, {T_REG, T_REG | T_DIR | T_IND, T_DIR | T_REG}, 11, 25,
30.      "store index"},
31.     {"fork", 1, {T_DIR}, 12, 800, "fork"},
32.     {"lld", 2, {T_DIR | T_IND, T_REG}, 13, 10, "long load"},
33.     {"lldi", 3, {T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG}, 14, 50,
34.      "long load index"},
35.     {"lfork", 1, {T_DIR}, 15, 1000, "long fork"},
36.     {"aff", 1, {T_REG}, 16, 2, "aff"},
37.     {0, 0, {0}, 0, 0, 0}
38.   };
```

Le type **op_t** est défini dans *op.h*

```
47. struct  op_s
48. {
49.     char        *mnemonique;
50.     char        nbr_args;
51.     args_type_t type[MAX_ARGS_NUMBER];
52.     char        code;
53.     int         nbr_cycles;
54.     char        *comment;
55. };
56.
57. typedef struct op_s     op_t;
```

- Et en dernier les paramètres des instructions.

Une instruction peut avoir entre 0 **et MAX_ARGS_NUMBER** *(op.h)* paramètres, séparés par des virgules.

```
16. # define MAX_ARGS_NUMBER         4     /* this may not be changed 2^*IND_SIZE */
```

Les paramètres peuvent être de trois types :

- Les registres
  De *r*1 à *r*REG_NUMBER

```
32. # define REG_NUMBER      16              /* r1 <--> rx */
```

- Les valeurs directes
  Une valeur directe est écrite en deux parties
  - **DIRECT_CHAR** *(op.h)*
    ```
    20. # define DIRECT_CHAR             '%'
    ```
  - Une valeur ou une étiquette (qui suit le **LABEL_CHAR**)

Elle représente directement la valeur donnée, comme par exemple `%4` ou `%label`.

- Les valeurs indirectes
  Une valeurs indirecte est écrite en deux parties, tout comme une valeur direct.

  Pour un registre :
  - Le caractère *r*
  - Une valeur numérique

  Pour une étiquette :
  - Le **LABEL_CHAR**
  - Le nom de l'étiquette

Il sagit de la valeur qui est trouvée à l'adresse stockée dans la valeur indirecte. (En relation avec le PC, « *Program Counter* », qui contient l'adresse de la prochaine instruction à décoder et executer).

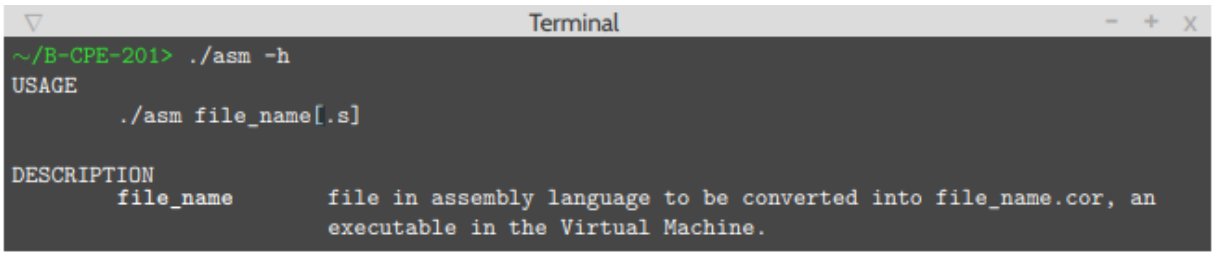
L'assembleur prend un fichier en code assembleur (le champion) comme paramètre, et crée un executable pour la machine virtuelle en convertissant le code assembleur en langage machine.

La machine virtuelle est dit « gros-boutienne » (big endian), c'est-à-dire que l'octet de poids le plus fort est enregistré à l'adresse mémoire la plus petite, il sagit de la notation qui viendrait directement à l'esprit.

Par exemple, le nombre `0x42DAD420` serait stocké en mémoire sous cette forme.

	0	1	2	3	
…	42	DA	D4	20	…

Le boutisme dépend de l'architecture du processeur, l'architecture Sun à un gros-boutisme, et l'architecture i386 un petit-boutisme.



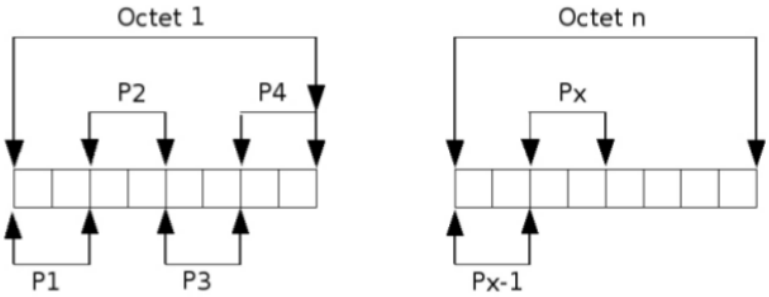
Le code

Chaque instruction est codée en trois partie :

- **Le code d'instruction**, qui peut être trouvé dans le tableau **op_tab** *(op.c)*

- **Les types de paramètres utilisés**, appelé le « coding byte », les instructions live, zjmp, fork et lfokn n'en n'ont pas.
  Les valeurs possibles sont :
  - 01 pour un registre
  - 10 pour une valeur directe
  - 11 pour une valeur indirecte

  Les paramètres sont groupés 4 par 4 pour former des octets complets.



- **Les paramètres**
  - Un octet pour un registre (sa valeur hexadécimal)
  - DIR_SIZE octets pour une valeur directe (sa valeur en hexadécimal)
  - IND_SIZE octets pour une valeur indirecte (sa valeur en hexadécimal)

DIR_SIZE et IND_SIZE sont définis dans *op.h*

```
62. # define IND_SIZE        2
63. # define DIR_SIZE        4
```

Par exemple, `r2, 23, %34` donnera :
- 01 11 10 00, soit 0x78, il y a deux bits par argument, comme décrit au dessus. Pas d'argument donnera 00.
- 02 00 17 00 00 00 22, soit les paramètres

Dans le cas d'une étiquette (label), lorsque celle-ci est définie, elle n'apparaît pas à la compilation, par contre, lorsque celle-ci est appelée, elle est remplacée par la différence entre l'adresse du début de l'instruction en cours et l'adresse du début de l'instriction où l'étiquette a été définie.

Un exemple complet, l'instruction `st r2 r8` : 03 50 04 08

03 : code d'instruction
50 : coding byte
04 08 : Les arguments (le coding byte décrivant deux registres et DIR_SIZE étant égal à 4, les deux arguments sont codés sur 4 bits chacun)

La machine virtuelle

Introduction

La machine virtuelle est capable d'éxecuter plusieurs programmes, chaques programmes contenant :

- REG_NUMBER registres de REG_SIZE octets chacun
  ```
  64. # define REG_SIZE        DIR_SIZE
  ```
  Un registre est une zone mémoire qui contient une variable. Sur une vraie machine, elle est inclus dans le processeur et on peut donc y accéder très rapidement. **REG_NUMBER** et **REG_SIZE** sont définis dans *op.h.*

- Un PC (Program counter)
  C'est un registre spécial qui contient l'adresse mémoire (dans la machine virtuelle) de la prochaine instruction à décoder et à executer. Il est très pratique pour savoir où on se trouve ainsi que pour écrire dans la mémoire.

- Une variable appelée « carry » (retenue) qui est égale à 1 seulement si la dernière opération a retournée 0.

Le rôle de la machine virtuelle est d'executer les programmes qui lui sont donnés en paramètres, générant des processus.
Elle doit vérifier que chaque processus appelle l'instruction « live » tous les **CYCLE_TO_DIE** cyles *(op.h)*.

```
91. # define CYCLE_TO_DIE    1536    /* number of cycle before beig declared dead */
```

Si après **NBR_LIVE** exécutions de l'instruction « live », plusieurs processus sont toujours actifs, **CYCLE_TO_DIE** est baissée de **CYCLE_DELTA**, et la boucle recommence *(op.h)*.

```
92. # define CYCLE_DELTA     5
93. # define NBR_LIVE        40
```

Le dernier champion à avoir executé l'instruction « live » gagne.

```
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
```

La machine virtuelle doit pouvoir être construite et executée sans environnement graphique.

Les messages

Un nombre est associé à chaque joueur. Ce nombre est généré par la machine virtuelle et est donné aux programmes dans le registre *r1* au démarrage du système (Tous les autres seront initialisés à 0, sauf le PC).

À chaque execution de l'instruction « live », la machine doit afficher « *The player **NB_OF_PLAYER(NAME_OF_PLAYER)** is alive.*».

Quand un joueur gagne, la machine doit afficher « *The player **NB_OF_PLAYER(NAME_OF_PLAYER)** has won.*».

Pour passer les tests à la moulinette, il est **obligatoire** de respecter très précisement ces messages.
Toutes les options présentées seront utilisées.
Toutes les valeurs du *op.h* doivent avoir les mêmes valeurs que le fichier donné.

Les cycles

La machine virtuelle simule une machine parallèle à celle où elle est lancée.
Pour des raisons d'implémentations, on va assumer que chaque instruction s'execute entièrement à la fin de son dernier cycle et attend pendant toute la durée qui lui est attribuée (Comme une boule de feu dans un jeu, où il faudrait la faire charger un certain nombre de tours avant de devoir la lancer).

Lorsque des instructions commencent au même cycle, elles sont executées dans l'ordre des joueurs.

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)

La machine virtuelle executera les instructions dans cet ordre :

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

Au cycle 21, le compte du nombre de cycle de **1,6** commence, puis **2,5** s'execute puis le compte du nombre de cycle de **3,6** commence.

Le code machine

La machine virtuelle doit reconnaître les instructions suivantes (tous les autres codes ne font rien à part passer un cycle)

N'oubliez pas que la mémoire est circulaire (c'est-à-dire qu'après le dernier octet de mémoire se trouve le premier).
La mémoire disponible est de MEM_SIZE octets *(op.h)*

```
14. # define MEM_SIZE                (6*1024)
```

Le nombre de cycles de chaque instruction ainsi que leur représentation mnemonique, leur nombre de paramètres et les types des possibles paramètres sont décrit dans le tableau **op_tab** *(op.c)*

```
1.  # define T_REG           1       /* register */
2.  # define T_DIR           2       /* direct  (ld  #1,r1  put 1 into r1) */
3.  # define T_IND           4       /* indirect always relative
4.                                      (ld 1,r1 put what's in the address (1+pc)
5.                                      into r1 (4 bytes)) */
```

MNEMONIC	CODE BYTE	CYCLES	PARAM	EFFET
0x01 (live)	//	10	1	Prend un paramètre, 4 octets qui indiquent le nombre du joueur. Indique que le joueur est en vie.
0x02 (ld)	T_DIR| T_IND, T_REG	5	2	Il stock la valeur du premier paramètre dans le second, qui ne peut pas être le PC. `ld 34, r3` stock les REG_SIZE premiers octets de l'adresse PC + 34 % IDX_MOD dans r3.
0x03 (st)	T_REG, T_IND | T_REG	5	2	Stock la valeur du premier paramètre dans le deuxième `st r4, 34` stock le contenu de *r4* à l'adresse PC + 34 % IDX_MOD. `st r4, r8` stock le contenu de *r4* dans *r8.*
0x04 (add)	T_REG, T_REG, T_REG	10	3	Ajoute le contenu des deux premiers paramètre et met la somme dans le troisième. Cette opération modifie la retenue. `add r2, r3, r5` met le resultat de l'ajout du contenu de *r2* et *r3 dans* r5.
0x05 (sub)	T_REG, T_REG, T_REG	10	3	Similaire à add mais avec une soustraction.
0x06 (and)	T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG	6	3	Fait un AND binaire entre les deux premiers paramètres et stock le résultat dans le troisième. Cette opération modifie la retenue. `and r2, %0, r3` met `r2 & 0` dans *r3.*
0x07 (or)	T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG	6	3	Similaire à and mais avec un OR binaire.
0x08 (xor)	T_REG | T_DIR | T_IND, T_REG | T_IND | T_DIR, T_REG	6	3	Similaire à and mais avec un XOR binaire (OR exclusif).
0x09 (zjmp)	//	20	1	Prend un index en paramètre. Jump à l'adresse donnée en paramètre si la retenue est égale à 1, sinon, ne fait rien et perd un cycle. `zjmp %23` met, si la retenue est egale à un, PC + 23 % IDX_MOD dans le registre PC.
0x0a (ldi)	T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG	25	3	`ldi 3, %4, r1` lit les IND_SIZE premiers octets de l'adresse PC + 3 % IDX_MOD, et y ajoute 4. La somme est appelée S. REG_SIZE octets sont lu à l'adresse PC + S % IDX_MOD et sont copiés dans *r1*. Cette opération modifie la retenue.
0x0b (sti)	T_REG, T_REG | T_DIR | T_IND, T_DIR | T_REG	25	3	`sti r2, %4, %5` copie le contenu de *r2* à l'adresse PC + (4 + 5) % IDX_MOD.
0x0c (fork)	//	800	1	Crée un nouveau programme qui hérite de l'état du parent. Le program est executé à l'adresse PC + premier paramètre % IDX_MOD
0x0d (lld)	T_DR | T_IND, T_REG	10	2	Similaire à ld sans %IDX_MOD. Modifie la retenue.
0x0e (lldi)	T_REG | T_DIR | T_IND, T_DIR | T_REG, T_REG	50	3	Similaire à ldi sans %IDX_MOD. Modifie la retenue.
0x0f (lfork)	//	1000	1	Similaire à fork sans %IDX_MOD.
0x10 (aff)	T_REG	2	1	Écrit sur stdout la valeur dans le registre donné en paramètre, modulo 256.

Les champions

Introduction

Les champions sont écrits en code assembleur, décrit dans la partie sur l'assembleur.
Quand le jeu, et, par extension, la machine virtuelle se lance, chaque champion va trouver son registre *r1* personnel (le nombre lui est assigné par la machine virtuelle).

Toutes les instructions sont utiles, toutes les réactions de la machine qui sont décrites dans ce document peuvent donner au champion plus de vie, et aider à trouver une stratégie efficace pour gagner.
L'instruction **« fork »** par exemple, sera très utile pour submerger votre adversaire, mais faites attention, elle prend et peut devenir mortelle si **CYCLE_TO_DIE** cycles passent et que l'instruction **« live »** n'a pas été executée.

Si un champion execute une instruction « live » avec un nombre autre que le sien, dommage, mais quelqu'un sera content de votre malheur.

Exemple

```
1.  #
2.  # zork.s for corewar
3.  #
4.  # Bob Bylan
5.  #
6.  # Sat Nov 10 03:24;30 2081
7.  #
8.  .name "zork"
9.  .comment "just a basic living program"
10.
11. l2:
12. sti r1, %:live, %1
13. and r1, %0, r1
14. live %1
15. 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 01 01 |.h......d.......|
000008a0 00 00 00 01 09 ff fb |.......|
000008a7
```

On peut voir que le code assembleur commence par `00 ea 83 f3` il sagit du magic code, qui doit être présent sur chaque champion.
Vient ensuite le nom du champion, écrit sur **PROG_NAME_LENGTH** octets, puis le commentaire sur **COMMENT_LENGTH** octets *(op.h)*, puis le code de l'algo.

```
74. # define PROG_NAME_LENGTH        128
75. # define COMMENT_LENGTH          2048
```

Conclusion

Enfin…

Les fichiers *op.c* et *op.h* sont très utiles, il ne faut pas les négliger.
Des binaires de références sont donnés, ils sont très pratiques et devraient être utilisés pour travailler, ainsi que pour tester vos programmes.

N'oubliez pas les tests unitaires (et fonctionnels).

En cas de problème avec le sujet, contactez moi.

Mise en page du code faite avec http://planetb.ca/syntax-highlight-word.

Bon courage.