

Core War

Version du 13 December 2002

Assistants C/Unix

“The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards – and even then I have my doubts.” –Eugene H. Spafford

“The people of Krikkit have never thought to themselves [as if they were] alone in the Universe. They are surrounded by a huge Dust Cloud, you see, their single sun with its single world, and they are right out on the utmost eastern edge of the Galaxy. Because of the Dust Cloud there has never been anything to see in the sky. At night it is totally blank, During the day there is the sun, but you can't look directly at that so they don't. They are hardly aware of the sky. It's as if they had a blind spot which extended 180 degrees from horizon to horizon.”

“You see, the reason why they have never thought We are alone in the Universe is that until tonight they don't know about the Universe. Until tonight.” – Douglas Adams - Life, the Universe and Everything

Le corewar est une sorte de guerre virtuelle. Elle se déroule dans une zone de mémoire ou des processus évoluent. Ces processus représentent en fait des processeurs virtuels qui exécutent le code situé dans la mémoire et émettent des signaux de vie pour les joueurs. Au cours d'une partie, les processeurs pourront être détruits, se multiplier (leur nombre est variable au cours d'une partie et est théoriquement illimité) et manipuler la mémoire (c'est là tout le coeur du jeu). Bref, c'est la guerre et tous les coups sont permis.

1 Considérations générales

1.1 Indications administratives

- Le projet doit être rendu avant le 20 décembre à 69:42.
- Il doit être réalisé par groupes de 4 personnes, dont les noms et les logins doivent figurer dans le fichier ‘AUTHORS’, et indiqués par mail à acu@epita.fr en utilisant un sujet de la forme :

```
[COREWAR] login1:login2:login3:login4
```

- L’évaluation aura lieu par groupe¹.
- Pour tout renseignement complémentaires, veuillez poser vos questions sur le groupe ‘epita.cours.c-unix.kriquet-wars’.

1.2 Modalités de rendu

- Le programme devra se trouver dans le répertoire :

```
~/c/rendu/proj/corewar
```

- Les sources du projet doivent être conformes au CSS EPITA, notamment concernant les fichiers ‘AUTHORS’ et ‘Makefile’.
- La règle ‘all’ du fichier ‘Makefile’ doit provoquer la création de deux exécutables, celui de l’assembleur, nommé ‘asm’, et celui de la machine virtuelle, nommé ‘corewar’.
- Le rendu doit être effectué en utilisant le script de rendu des ACUs :

```
~/acu/sbin/rendu.sh proj corewar
```

Le rendu ne doit être fait que sur un seul compte.

1.3 Contraintes techniques

- Vous devez coder ce projet en C.
- Les seules fonctions autorisées sont les suivantes :
`open(2)`, `close(2)`, `read(2)`, `write(2)`, `malloc(3)`, `realloc(3)`, `free(3)`, `exit(3)`.
- Les sources doivent être conformes au CSS.
- Le fichier ‘data.h’, contenant les définitions de macros indiquées dans le sujet, sera introduit de force dans votre répertoire lors de la correction. Il est évident que votre programme devra alors tenir compte de ces définitions. Les constantes suivantes seront définies : `COREWAR_EXEC_MAGIC`, `PROG_NAME_LENGTH`, `PROG_COMMENT_LENGTH`, `IDX_MOD`, `MAX_CODE_SIZE`, `MEM_SIZE`, `CYCLE_TO_DEATH`, `CYCLE_DELTA`, `EPOCH`, `MAX_PROCESSORS`, `DELAY_DIE`, `DELAY_LIVE`, `DELAY_ARITH`, `DELAY_LP`, `DELAY_CBRANCH`, `DELAY_LD`, `DELAY_ST`, `DELAY_LC`, `DELAY_LL`, `DELAY_FORK`, `DELAY_STACK`, `DELAY_JMP`, `DELAY_FL`, `DELAY_WRITE`, `DELAY_STAT`, `DELAY_NOP`.
- Votre projet doit fonctionner identiquement sur les trois architectures de l’école auxquelles vous avez couramment accès.

¹ sauf cas exceptionnel

2 La machine virtuelle

La machine virtuelle est une machine multi-processeurs.

2.1 Fonctionnement général

Le rôle de la machine est d'exécuter les programmes qui lui sont données en argument (see Section 2.3 [Interaction], page 4).

L'exécution des programmes sur les processeurs est conditionnée par plusieurs compteurs, associés à des règles:

CYCLES_TO_DEATH

Nombre de cycles d'exécution au bout duquel un processus meurt s'il n'a pas exécuté l'instruction `live`.

EPOCH Nombre d'appels à l'instruction `live` au bout duquel la variable `CYCLES_TO_DEATH` est décrémentée de `CYCLES_DELTA` (précisions plus loin).

CYCLES_DELTA

Décrément pour `CYCLES_TO_DEATH`.

Les règles sont les suivantes:

- Si un processeur manque d'exécuter `live` pendant `CYCLES_TO_DEATH` cycles, il meurt.
- Lorsque l'instruction `live` est exécutée `EPOCH` fois, on dit qu'une *époque* se termine.
- Lorsqu'une époque se termine, la variable `CYCLES_TO_DEATH` est décrémentée de la valeur `CYCLES_DELTA`.
- Lorsqu'il ne reste plus aucun processeur en vie, la partie se termine, et on dit que le joueur ayant bénéficié du dernier `live` a gagné.

2.2 Les processeurs

Chaque processeur se voit associé les structures suivantes:

les registres

16 registres de calcul de 16 bits chacun.

la pile

Une pile circulaire de 16 niveaux de 16 bits chacun.

le compteur PC

le pointeur d'exécution de programme (16 bits).

P

un registre de 2 bits (voir plus loin).

Z

un registre de 1 bit (voir plus loin).

Voici une description de ces différents registres :

- La pile est circulaire, ce qui signifie qu'après avoir dépilé 16 valeurs, on peut à nouveau redépiler les mêmes valeurs. Si l'on dépile la valeur 1, puis la valeur 0, la prochaine valeur à être dépilée est celle qui se trouve à la 15e position.
- *P* est un registre spécial, il n'est pas utilisé pour les calculs, mais influe sur le fonctionnement de presque toutes les instructions. Les registres de calcul font 16 bits, c'est-à-dire 4 quartets numérotés de 0 à 3, le quartet 0 représentant les 4 bits de poids faible. *P* fait 2 bits et peut donc prendre les valeurs de 0 à 3. D'une manière générale, les instructions du processeur travaillent uniquement sur les quartets 0 à *P* des registres (les $(P + 1) * 4$ bits de poids faible).

- Z est un autre registre spécial. Il est essentiellement affecté par les instructions de calcul. Après un calcul, Z vaut 1 si le résultat est nul, sinon il prend la valeur 0. Attention: n’oubliez pas que les instructions de calcul travaillent avec les quartets 0 à P, donc Z dépend uniquement du résultat sur ces mêmes quartets 0 à P. Quelques autres instructions modifient Z et cela sera précisé (see [Section 3.1 \[Description des instructions\]](#), page 8).
- On utilise le little-endian¹. Ce concept s’oppose à celui du big-endian². Dans notre cas, il concerne uniquement les accès mémoire et le format des fichiers sur le disque. Ceux-ci auront lieu en little-endian, c’est-à-dire qu’en mémoire les bytes de poids faible seront aux adresses faibles. Par exemple, pour des bytes de 4 bits, si la mémoire contient 0, 0, 0, 0, 0, 0, 0, 0, 0 (en quartets) aux adresses 0x0 à 0x7 et qu’on écrit la valeur 0x1234 sur 2 octets à l’adresse 0x1, la mémoire contiendra après écriture 0 4 3 2 1 0 0 0 à l’adresse 0x0. Si maintenant on lit la valeur sur 1 quartet à l’adresse 0x2, on aura 0x3.
- En ce qui concerne le stockage des données sur le disque, qui est orienté *octets*, on mettra les quartets de poids faibles dans les poids faibles des octets, afin que la représentation en octets de la mémoire quartets 0 4 3 2 1 0 0 0 soit 04 32 10 00.
- Les processeurs fonctionnent entièrement en relatif. Cela signifie que la position du programme dans la mémoire n’influe pas sur son comportement. Les accès mémoire et les sauts se font d’une certaine distance en avant ou en arrière par rapport à la position courante (PC).

2.3 Interaction

2.3.1 Syntaxe de la ligne de commande

```
corewar [-help] [-ms n] [-mp n] [-im n] [-cy n] [-nl n] [-cd n] [-t] [file0 ... [file1
```

-help Affiche un descriptif d’aide pour le programme. Si cette option est rencontrée, le programme ne fait rien d’autre qu’afficher le message.

-dump n Active la génération d’une image de la mémoire au bout de n cycles. Par défaut, cette image n’est jamais générée. Une image de la mémoire est un affichage de son contenu sur la sortie standard, dans le style de `hexdump -C`³.

-ms n Permet d’indiquer la taille de la mémoire (8192 quartets par défaut).

-mp n Permet d’indiquer le nombre maximum de processeurs par joueur (2048 par défaut).

-im n Permet d’indiquer la valeur de la constante `IDX_MOD` (128 par défaut).

-cy n Permet d’indiquer la valeur initiale de `CYCLES_TO_DEATH` (1554 par défaut).

-ep n Permet d’indiquer la constante `EPOCH` (700 par défaut).

-cd n Permet d’indiquer la constante `CYCLES_DELTA` (1 par défaut).

filen Permet d’indiquer un champion dans la partie.

-t (bonus) Active le mode trace, dans lequel le programme affiche les instructions au fur et à mesure de leur exécution.

¹ organisation petit-boutiste

² organisation grand-boutiste

³ voir l’exercice correspondant en piscine

2.3.2 Messages

- Lorsque le programme se termine et qu'un joueur a gagné (c'est lui qui a bénéficié du dernier `live`), il doit afficher:

Le joueur `n`(nom du joueur) a gagné.

Le numro du joueur est celui attribué pendant l'initialisation de la partie (voir plus loin).

Par exemple:

Le joueur `1`(zork) a gagné.

- Lorsque le programme se termine et que lors du dernier `live` plusieurs processeurs ont exécuté `live` en même temps, le programme doit afficher:

Égalité entre les joueurs `n1`(nom1) `n2`(nom2) ...

- Lorsque le programme se termine et qu'aucun joueur n'a jamais bénéficié d'aucun `live`, le programme doit afficher:

Aucun gagnant.

2.4 Initialisation

Au début de chaque championnat:

- L'espace mémoire va être entièrement initialisé à la valeur 0 et le code des différents champions va être copié en mémoire. Le premier sera à l'adresse 0x0000 et les autres tous les (taille mémoire / nombre joueurs) quartets plus loin.
- Un numéro de 1 à 15 va être attribué de manière unique à chacun d'entre eux. Ce numéro va essentiellement servir pour les signes de vie. Au cours de la partie, les champions vont (devoir) émettre des signes de vie. Ceux-ci servent a déterminer le ou les vainqueurs. Comme cela sera expliqué plus loin, du point de vue du fonctionnement, les processeurs n'appartiennent pas aux joueurs, ils évoluent simplement dans la mémoire. Par contre, ils émettent des signaux de vie pour les joueurs, et le vainqueur est le joueur réel (tous les numéros ne sont pas attribués) pour lequel un signal de vie a été émis en dernier (si plusieurs signaux ont eu lieu au même cycle, il y a égalité).
- Enfin, un processeur est crée pour chaque joueur. Chaque processeur est initialisé de la manière suivante:
 - PC contient l'adresse à laquelle le code a été copié ;
 - le premier registre contient le numéro du joueur ;
 - tous les autres registres contiennent la valeur 0 ;
 - P vaut 0 ;
 - Z vaut 1 ;
 - tous les niveaux de la pile contiennent 0.

La partie est ensuite lancée.

2.5 Parallélisme

Pour ne pas favoriser de joueur en fonction de l'ordre de traitement des processeurs par la machine virtuelle, on va évoluer dans un système entièrement parallèle (comme la réalité). À chaque cycle, toutes les opérations vont être exécutées comme si elles avaient lieu exactement en même temps. Cela a une influence sur les signaux de vie et les écritures en mémoire. Nous en reparlerons au cas par cas dans les sections suivantes.

2.5.1 Destruction des processeurs

Au cours de la partie, les processeurs ne sont pas invulnérables, ils peuvent être détruits de deux façons:

minus en exécutant l'instruction `die` (un `0x00` est si vite arrivé...)

minus en n'exécutant pas de `live` pendant un délai trop long

En effet, les signaux de vie ne sont pas uniquement utilisés pour déterminer le gagnant, ils sont aussi indispensables à la survie des processeurs. Il y a un compteur dans la machine virtuelle nommé `CYCLES_TO_DEATH`. Si à un cycle quelconque de la partie un processeur n'a pas exécuté un `live` quelconque (sauf `live 0` bien sûr, qui correspond à l'instruction `die`) depuis `CYCLES_TO_DEATH` cycles ou plus, il est instantanément détruit.

Comme il a été précisé dans See [Section 2.1 \[Fonctionnement general\], page 3](#), pour rendre la chose plus intéressante et pour limiter la durée des parties, `CYCLES_TO_DEATH` va décroître au cours du jeu. En fait, plus les processeurs exécutent `live`, plus `CYCLES_TO_DEATH` décroît. Il existe deux autres compteurs fixes nommés `EPOCH` et `CYCLES_DELTA`: tous les `EPOCH` exécutions d'un `live` par un processeur quelconque, `CYCLES_TO_DEATH` est décrementé de `CYCLES_DELTA`.

Pour respecter le parallélisme, `CYCLES_TO_DEATH` n'évolue pas juste au moment où un `live` est exécuté, mais à la fin du cycle, une fois que tout les `live` à faire ou toutes les lectures des compteurs ont eu lieu.

2.5.2 Concurrence des accès mémoire

Le problème est simple: que se passe-t-il lorsque plusieurs processeurs écrivent en mémoire au même endroit au même moment (dans le même cycle) ? Pour respecter le parallélisme, on décide qu'*aucune* écriture ne réussit : la mémoire n'est pas modifiée.

2.5.3 Fonctionnement de fork

Comme nous venons de le voir, le nombre de processeurs peut décroître, mais il lui est aussi possible d'augmenter. En fait, cela est possible grâce à l'instruction `fork`. Cette instruction a pour but de dupliquer le processeur qui l'exécute. À l'issue de celle-ci (fin du cycle d'exécution), un nouveau processeur va être introduit dans la machine virtuelle entièrement identique à celui qui a exécuté le `fork`. La seule différence entre les 2 est l'état du registre `Z` qui vaut 0 dans un processeur et 1 dans l'autre, ce qui permet de les différencier.

2.5.4 Anti-jeu

À cause du système de `CYCLES_TO_DEATH` pour limiter le temps des parties, il est possible d'appliquer une technique d'anti-jeu qui consiste à multiplier le nombre de processeurs de manière exponentielle puis d'exécuter `live`. La partie se termine alors très rapidement (`CYCLES_TO_DEATH` passe en dessous de 0) et le joueur qui a appliqué cette méthode est quasiment sûr de gagner.

Pour empêcher cela et parce que la VM tourne dans un environnement fini, le nombre de processeurs va être limité. Il y a beaucoup de méthodes pour éviter l'anti-jeu, nous avons choisi celle qui semblait la plus simple, la plus efficace et la plus juste.

Entre processeurs et joueurs, du point de vue de l'exécution, il y a un lien d'appartenance assez faible (le premier processeur a le numéro du joueur dans le premier registre et chaque processeur peut demander à qui il appartient), les processeurs ne font qu'exécuter la mémoire. Si on change le code dans la mémoire, le processeur qui passe dessus fait ce qu'on veut.

Par contre, en ce qui concerne les `fork`, le lien va être beaucoup plus fort. Chacun des joueurs va avoir droit à un nombre limité de processeurs. L'appartenance est héréditaire,

c'est à dire que lorsqu'un processeur exécute `fork`, les 2 processeurs resultants appartiendront au même joueur que le processeur qui a fait le `fork`. Si au cours d'un cycle un ou plusieurs processeurs d'un même joueur termine leur `fork` et que cela fait dépasser le plafond, le ou les `fork` en question vont simplement échouer. Le plafond sera fixé par la constante `MAX_PROCESSORS` du `data.h`.

Le vol d'un processeur d'un autre joueur devient alors très intéressant pour empêcher celui-ci de `forker`...

2.6 Le monde du relatif

Comme cela a été précisé au début, les processeurs travaillent intégralement en relatif. Cela signifie que tous les accès mémoire et les sauts se font relativement au PC courant (donc à la fin de l'instruction qui fait une lecture, une écriture ou un saut). De plus, pour que cela fonctionne correctement, la mémoire est circulaire, c'est-à-dire que si elle mesure 1000 quartets, qu'un PC vaut 990 et que le processeur saute de 20 en avant, PC vaudra ensuite 10 et pas 1010.

Or, comme on peut le constater dans la table instructions (see [Section 3.1 \[Description des instructions\]](#), page 8), les instructions classiques d'accès à la mémoire ou de saut utilisent dans leur calcul une constante nommée `IDX_MOD`. Cette valeur qui est précisée au lancement de la VM sert tout simplement à limiter la portée des lectures, écritures ou sauts en mémoire. Pour chacune des instructions classiques, un "modulo" `IDX_MOD` va être appliqué à la distance d'accès. En fait, quand `PC+IDX_MOD` est dépassé, on revient à `PC-IDX_MOD`. Il n'y a aucun moyen d'écrire ou de lire au-delà d'`IDX_MOD` avec les instructions qui y sont assujetties ; cela signifie en particulier que si un processeur écrit 4 quartets à `PC+IDX_MOD-2`, il écrira les deux derniers à `PC-IDX_MOD`. Notez que le quartet `PC-IDX_MOD` est le premier inscriptible, alors que `PC+IDX_MOD` est le premier NON-inscriptible.

Il existe cependant des instructions dites "longues" qui ne tiennent pas compte de cette limite pour la lecture et le saut en mémoire, mais elles sont plus grosses et plus lentes.

3 Tables des instructions

Les sections suivantes décrivent le jeu d'instruction de la machine.

3.1 Description des instructions

Les instructions sont de taille variable et elles peuvent prendre des paramètres qui peuvent être:

- des registres: rx ou ry (il n'y a que 16 registres numérotés de 0 à 15)
- des expressions constantes: n

Dans les tables qui suivent, la première colonne représente les mnémoniques des instructions. Un mnémonique est un identifiant qui représente une instruction dans le langage de l'assembleur. Les mnémoniques sont suivis des paramètres à fournir, s'il y en a, séparés par des virgules. Notez que chaque caractère est important et qu'il devra apparaître dans le code.

<code>die</code>	Tout processeur qui exécute cette instruction est détruit instantanément.
<code>live n</code>	Émet un signe de vie pour le joueur n . Le fonctionnement est indépendant de P et n'affecte pas Z .
<code>mov rx, ry</code>	Copie les quartets 0 à P de ry dans rx . Le fonctionnement n'affecte pas Z .
<code>swp rx, ry</code>	Échange les quartets 0 à P de rx et ry . Le fonctionnement n'affecte pas Z .
<code>not rx, ry</code>	Calcule la négation logique (complément à 1) des quartets 0 à P de ry et stocke le résultat dans les quartets 0 à P de rx .
<code>and rx, ry</code>	Calcule le ET logique bit à bit entre les quartets 0 à P de rx et de ry et stocke le résultat dans les quartets 0 à P de rx .
<code>or rx, ry</code>	Calcule le OU logique bit à bit entre les quartets 0 à P de rx et de ry et stocke le résultat dans les quartets 0 à P de rx .
<code>xor rx, ry</code>	Calcule le OU exclusif bit à bit entre les quartets 0 à P de rx et de ry et stocke le résultat dans les quartets 0 à P de rx .
<code>rol rx, n</code>	Effectue une rotation vers la gauche de n bits des quartets 0 à P de rx . L'instruction <code>ror</code> qui effectue une rotation à droite n'existe pas car on peut la simuler grâce à un <code>rol</code> avec une valeur appropriée de n .
<code>asr rx, n</code>	Effectue un décalage arithmétique à droite de n bits des quartets 0 à P de rx . Ce décalage a pour caractéristique de faire entrer des bits identiques au bit de signe (conservation du signe).
<code>add rx, ry</code>	Additionne les quartets 0 à P de ry à ceux de rx et stocke le résultat dans les quartets 0 à P de rx ($rx = rx + ry$).
<code>sub rx, ry</code>	Soustrait les quartets 0 à P de ry de ceux de rx et stocke le résultat dans les quartets 0 à P de rx ($rx = rx - ry$).
<code>rsb rx, ry</code>	Soustrait les quartets 0 à P de rx de ceux de ry et stocke le résultat dans les quartets 0 à P de rx ($rx = ry - rx$).

<code>neg rx, ry</code>	Calcule la négation arithmétique (complément à 2) des quartets 0 a P de <code>ry</code> et stocke le résultat dans les quartets 0 a P de <code>rx</code> .
<code>inc rx, n</code>	Incremente de <code>n</code> les quartets 0 a P de <code>rx</code> ($rx = rx + n$).
<code>dec rx, n</code>	Decremente de <code>n</code> les quartets 0 a P de <code>rx</code> . ($rx = rx - n$).
<code>lsl rx, n</code>	Effectue un décalage à gauche de <code>n</code> bits des quartets 0 a P de <code>rx</code> . Les bits entrants sont nuls.
<code>lsr rx, n</code>	Effectue un décalage à droite de <code>n</code> bits des quartets 0 a P de <code>rx</code> . Les bits entrants sont nuls.
<code>lp n</code>	Charge les 2 bits de poids faible de <code>n</code> dans P ($P = n$). Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>bnz rx</code>	Effectue un saut relatif de <code>rx</code> modulo <code>IDX_MOD</code> quartets si $Z = 0$ ($PC = PC + (rx \% IDX_MOD)$). <code>rx</code> est signé et utilisé en entier. Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>bz rx</code>	Effectue un saut relatif de <code>rx</code> modulo <code>IDX_MOD</code> quartets si $Z = 1$ ($PC = PC + (rx \% IDX_MOD)$). <code>rx</code> est signé et utilisé en entier. Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>ld rx, [ry]</code>	Charge les $P + 1$ quartets stockés a l'adresse $PC + (ry \% IDX_MOD)$ dans les quartets 0 a P de <code>rx</code> . <code>ry</code> est signé et utilisé en entier. Le fonctionnement n'affecte pas Z.
<code>st [rx], ry</code>	Stocke les quartets 0 a P de <code>ry</code> dans les $P + 1$ quartets stockés a l'adresse $PC + (rx \% IDX_MOD)$. <code>rx</code> est signé et utilisé en entier. Le fonctionnement n'affecte pas Z.
<code>lc rx, n</code>	Charge <code>n</code> dans les 2 quartets de poids faible de <code>rx</code> et propage le bit de signe de ces 2 quartets dans les 2 quartets de poids fort de <code>rx</code> . Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>ll rx, n</code>	Charge <code>n</code> dans le registre <code>rx</code> entier. Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>fork</code>	Crée un nouveau processeur en tout point identique à celui qui exécute le <code>fork</code> à l'exception du registre Z qui vaudra 0 dans l'un et 1 dans l'autre. Le fonctionnement est indépendant de P.
<code>push rx</code>	Empile <code>rx</code> entier dans la pile du processeur. Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>pop rx</code>	Depile une valeur de la pile du processeur et la stocke dans <code>rx</code> . Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>jmp rx</code>	Effectue un saut relatif long et inconditionnel de <code>rx</code> quartets ($PC = PC + rx$). <code>rx</code> est signé et utilise en entier. Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>fl rx, [ry]</code>	Charge les $P + 1$ quartets stockés a l'adresse $PC + ry$ dans les quartets 0 a P de <code>rx</code> . <code>ry</code> est signé et utilisé en entier. Le fonctionnement n'affecte pas Z.
<code>write rx</code>	Écrit sur la sortie standard le caractère dont le code ASCII est contenu dans les 2 quartets de poids faible de <code>rx</code> . Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>stat rx, n</code>	Charge dans <code>rx</code> entier la valeur du compteur <code>n</code> (see Section 3.4 [Statistiques] , page 11). Le fonctionnement est indépendant de P et n'affecte pas Z.
<code>nop</code>	Ne fait rien. Le fonctionnement est indépendant de P et n'affecte pas Z.

3.2 Encodage

Cette table fournit le codage bit-à-bit des intructions. Les X représentent des bits non significatifs, c'est-à-dire que leur valeur est quelconque et n'influe pas. Les registres sont stockés sur 1 quartet et sont juste représentés par leur nom depuis la première colonne. Le premier registre sera codé par 0x0, le second par 0x1... Les constantes seront notées par des groupes de nnnn, chacun représentant un quartet de la constante. Attention: ces constantes sont stockées dans le code en little-endian (see [Section 2.2 \[Les processeurs\]](#), page 3).

Mnémonique	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Offset 5
die	X000	0000				
live n	X000	nnnn				
mov rx, ry	X001	0000	rx	ry		
swp rx, ry	X001	0001	rx	ry		
not rx, ry	X001	0010	rx	ry		
and rx, ry	X001	0011	rx	ry		
or rx, ry	X001	0100	rx	ry		
xor rx, ry	X001	0101	rx	ry		
rol rx, n	X001	0110	rx	nnnn		
asr rx, n	X001	0111	rx	nnnn		
add rx, ry	X001	1000	rx	ry		
sub rx, ry	X001	1001	rx	ry		
rsb rx, ry	X001	1010	rx	ry		
neg rx, ry	X001	1011	rx	ry		
inc rx, n	X001	1100	rx	nnnn		
dec rx, n	X001	1101	rx	nnnn		
lsl rx, n	X001	1110	rx	nnnn		
lsr rx, n	X001	1111	rx	nnnn		
lp n	X010	nnnn				
bnz rx	0011	rx				
bz rx	1011	rx				
ld rx, [ry]	X100	rx	ry			
st [rx], ry	X101	rx	ry			
lc rx, n	0110	rx	nnnn	nnnn		
ll rx, n	1110	rx	nnnn	nnnn	nnnn	nnnn
fork	X111	X000				
push rx	X111	X001	rx			
pop rx	X111	X010	rx			
jmp rx	X111	X011	rx			
fl rx, [ry]	X111	X100	rx	ry		
write rx	X111	X101	rx			
stat rx, n	X111	X110	rx	nnnn		
nop	X111	X111				

3.3 Délais

Chaque instruction est associée au nombre de cycles nécessaires pour mener à terme son exécution :

- la lecture de chaque instruction nécessite autant de cycles qu'il y a de quartets. Attention, à cet effet la lecture de l'instruction se fera au fur et à mesure des cycles, pas en une seule fois au premier cycle ;

- avant que l'exécution aie lieu, le processeur est en pause pendant un certain nombre de cycles ;
- l'exécution elle-même dure un certain nombre de cycles. Lorsqu'une instruction nécessite plusieurs cycles d'exécution à cause d'un transfert mémoire, celui-ci est réparti sur chaque cycle de manière à transférer au plus un quartet par cycle.

Voici la table correspondante :

Mnémonique	Décodage	Délai	Exécution
die	2	DELAY_DIE	0
live n	2	DELAY_LIVE	1
mov rx, ry	4	DELAY_ARITH	P + 1
swp rx, ry	4	DELAY_ARITH	P + 1
not rx, ry	4	DELAY_ARITH	P + 1
and rx, ry	4	DELAY_ARITH	P + 1
or rx, ry	4	DELAY_ARITH	P + 1
xor rx, ry	4	DELAY_ARITH	P + 1
rol rx, n	4	DELAY_ARITH	P + 1
asr rx, n	4	DELAY_ARITH	P + 1
add rx, ry	4	DELAY_ARITH	P + 1
sub rx, ry	4	DELAY_ARITH	P + 1
rsb rx, ry	4	DELAY_ARITH	P + 1
neg rx, ry	4	DELAY_ARITH	P + 1
inc rx, n	4	DELAY_ARITH	P + 1
dec rx, n	4	DELAY_ARITH	P + 1
lsl rx, n	4	DELAY_ARITH	P + 1
lsr rx, n	4	DELAY_ARITH	P + 1
lp n	2	DELAY_LP	1
bnz rx	2	DELAY_CBRANCH	1
bz rx	2	DELAY_CBRANCH	1
ld rx, [ry]	3	DELAY_LD	P + 1
st [rx], ry	3	DELAY_ST	P + 1
lc rx, n	4	DELAY_LC	1
ll rx, n	6	DELAY_LL	1
fork	2	DELAY_FORK	1
push rx	3	DELAY_STACK	1
pop rx	3	DELAY_STACK	1
jmp rx	3	DELAY_JMP	1
fl rx, [ry]	4	DELAY_FL	P + 1
write rx	3	DELAY_WRITE	1
stat rx, n	4	DELAY_STAT	1
nop	2	DELAY_NOP	1

3.4 Statistiques

Comme la table des instructions l'indique, il existe une instruction `stat` qui permet de récupérer la valeur d'un compteur. Les compteurs suivants peuvent être consultés :

- 0 Renvoie la constante EPOCH.
- 1 Renvoie la constante CYCLES_DELTA.
- 2 Renvoie la valeur actuelle de CYCLES_TO_DEATH.
- 3 Renvoie la taille de la mémoire en quartets.
- 4 Renvoie la valeur de IDX_MOD.
- 5 Renvoie les 16 bits de poids faible du compteur de cycles.

- 6 Renvoie les 16 bits de poids fort du compteur de cycles.
- 7 Renvoie 42.
- 8 Renvoie le nombre de joueurs de la partie.
- 9 Renvoie le numéro du joueur qui possède le processeur qui fait le `stat`.
- 10 Renvoie le nombre maximum de processeurs par joueur.
- 11 Renvoie le nombre de processeurs courant pour le joueur qui possède le processeur qui fait le `stat`.
- 12 Renvoie le nombre initial de joueurs.
- 13, 14, 15 Ne sont pas modifiés par l'instruction `stat`.

Remarque: le premier cycle de la partie est le cycle 0.

4 Assembleur

L'assembleur traduit les programmes écrits l'aide de mnémoniques dans leur représentation machine.

4.1 Fonctionnement

```
asm [-help] [-nc] [file1 ... [filen]]
```

-help Affiche un descriptif d'aide pour le programme. Si cette option est rencontrée, le programme ne fait rien d'autre qu'afficher le message.

-nc Ne génère pas les fichiers des champions.

file1 ... Les noms des fichiers sources à assembler.

L'assembleur s'occupe d'assembler des fichiers sources (que l'on nommera de préférence '*.s'), séparément. Pour chaque fichier, il tentera d'assembler et affichera les erreurs qu'il trouvera. Si aucune erreur n'est rencontrée, il générera un fichier '*.cor' qui contiendra le code du champion.

Une erreur rencontrée pendant la compilation d'un fichier source n'empêche pas le programme de tenter de compiler les fichiers source suivants.

4.2 Les séparateurs et les commentaires

Pour qu'un source soit assemblable, les différents éléments qui le composent doivent être séparés les uns des autres. Un séparateur sera juste un groupe de caractères formé d'espaces ou de tabulations.

L'assembleur gère les commentaires. Le caractère '#' marque le début d'un commentaire, et celui-ci se terminera à la fin de la ligne ('\n' suivant).

Exemple:

```
blablabla # Ceci est un commentaire, j'y mets ce que je veux!!!
```

4.3 Les directives

L'assembleur reconnaît des directives. Les directives ne sont pas des instructions au sens où elles ne laissent pas de trace directe dans le code généré. Elles servent en fait à transmettre des requêtes ou des informations à l'assembleur, celles-ci pouvant modifier son comportement.

Chaque directive est un mot commençant par un '.' et constitué de lettres (voire de chiffres et d'underscores).

L'assembleur contient seulement 2 directives: `.name` et `.comment`. Elles sont toutes les deux suivies d'une chaîne de caractères délimitée par des '"', et permettent de fournir à l'assembleur le nom et la description du champion (ceux-ci seront inclus dans le '*.cor'). Les chaînes de caractères pourront contenir n'importe quels caractères à l'exception du '".

Exemple:

```
.name "Terminator"
.comment "Ceux qui me croisent n'en reviennent pas."
```

4.4 Les labels

Un label est un identifiant qui sert à indiquer une position dans le code. Ils sont utiles pour calculer automatiquement les distances des sauts et des accès mémoire. Un label peut être constitué de lettres, de chiffres et de underscores, mais il ne devra pas commencer par un chiffre.

Les labels peuvent être déclarés n'importe où dans le code: au début, à la fin et entre les deux. Ils seront utilisés dans les expressions pour faire des calculs, même plus haut que leur déclaration. De plus, on ne devra pas pouvoir déclarer plusieurs fois le même label, l'assembleur devra générer une erreur dans ce cas.

Chaque label désigne l'instruction suivante.

Exemple:

```
to42:   # Declaration ok
3com:   # Label invalide
:a--a   # Label invalide
to42:   # Erreur, le label a déjà été déclaré
```

4.5 Les expressions

Partout où une instruction a besoin d'une valeur constante (un `n` dans le mnémotique), on devra pouvoir y mettre une expression. Une expression est une formule mathématique qui fournit une valeur en résultat d'évaluation.

Dans l'assembleur qui nous concerne, ces expressions sont très simples. L'assembleur ne supporte que les opérateurs `+` (addition) et `-` (soustraction et `-` unaire) et il n'y a pas de priorité, les opérations sont effectuées dans l'ordre exact où elles apparaissent.

De plus, comme le résultat doit être constant, ces opérations ne doivent pouvoir s'appliquer qu'avec des valeurs constantes. L'assembleur reconnaît les 2 types de constantes suivantes :

- Les constantes numériques (des nombres). Elles sont par défaut dans la base décimale courante. De plus, l'assembleur gère la base octale classique si elles commencent par `'0'`, la base hexadécimale classique si elles commencent par `'0x'`.
- Les labels. Les labels n'ont pas réellement de valeur. En fait, le seul intérêt est de pouvoir effectuer des différences entre 2 labels, ce qui doit donner la taille du code situé entre les deux labels en quartets. Nous allons supposer que le codeur est assez intelligent et attentif et qu'il ne fera pas n'importe quoi avec les labels, l'assembleur ne vérifie pas les opérations effectuées.

Exemples (on supposera que les labels `a`, `b`, `c` et `d` existent) :

```
lc r0, 5
lc r1, 0x42
lc r2, -19
lc r3, -0x666
lc r4, 1+2-3-0xBC+7
lc r5, a-b
lc r6, a-b+2
lc r7, a-b+c-d # Addition de 2 differences de labels -> pas de problème.
lc r8, a+2+b-0x2A-c-d # Idem, avec quelques operations en plus.
lc r9, -c+d # Idem.
```

4.6 Indications supplémentaires

—

On pourra considérer que qu'il y a au plus une instruction ou un label par ligne de source. Le traitement de plusieurs labels / instructions par ligne, ou des instructions étalées sur plusieurs lignes, sont des bonus.

- L’assembleur ne s’arrête pas à la première erreur qu’il rencontre dans le source, il continue l’assemblage jusqu’à la fin du fichier. Lorsqu’il détecte une erreur, il tente de reprendre de son mieux. Au pire, il saute tout ce qui reste sur la ligne et reprend l’assemblage. Il écrira donc la liste des erreurs rencontrées à l’écran avec leur numéro de ligne.

Attention : si ce système fait gagner du temps, il n’est pas parfait, cela peu provoquer la génération de fausses erreurs (ex: si un label est invalide, il n’est pas déclaré et l’assembleur dira que ce label n’existe pas pendant le link). Prenez garde.

- Il est capable de définir l’état des bits indéterminés des instructions. L’assembleur va associer une lettre de l’instruction à chacun de ces bits dans l’ordre. Un bit indéterminé est mis a 1 si la lettre correspondante est en majuscule, sinon il prend la valeur 0. Toutes les autres lettres de l’instruction doivent être en minuscule.

Exemple: `fork` s’encode `X111 X000` (see [Section 3.2 \[Encodage\], page 10](#)). Ainsi, le premier X va être associée au “f” et le deuxième au “o”. `fork` donnera `0111 0000` alors que `FOrk` sera traduit en `1111 1000` et `fOrk` en `0111 1000`.

- Il accepte des données pures. Vous pouvez écrire dans le code une série de quartets en hexadécimal qui sera intégrée directement au code. Les lettres doivent être en majuscule.

4.7 Format des fichiers

Les fichiers ‘.cor’ sont les fichiers gérés par l’assembleur. Il s’agit de fichiers binaires qui vont contenir le code exécutable du champion et toutes les informations relatives. Ils sont composés de 2 parties: un en-tête et le code.

Attention : Comme il s’agit de fichiers UNIX, il vont être orientés octets, et comme nous ne voulons pas vous depayer, on va y accéder en little-endian.

L’en-tête a la structure suivante :

- un "magic number" sur 32 bits valant `COREWAR_EXEC_MAGIC` (stocké en little-endian orienté octet)
- la taille du code (en nombre de quartets) exécutable sur 32 bits (idem)
- une zone de `PROG_NAME_LENGTH` octets contenant le nom du champion (directive ‘.name’) complété par des `0x00`
- une zone de `PROG_COMMENT_LENGTH` octets contenant la description du champion (directive ‘.comment’) complète par des `0x00`
- Le code est un champ d’octets contenant le code généré complété par un `0x0` si le nombre de quartets est impair (ce `0x0` n’est pas comptabilisé dans la taille du code). Ces octets sont obtenus comme si on avait lu une zone de mémoire contenant le code. On lit donc les quartets 2 par 2 en little-endian (orienté quartet) dans l’ordre jusqu’à la fin du code. Ainsi, l’écriture dans la mémoire de la machine virtuelle se fera en écrivant (toujours en little-endian pour ne pas modifier l’ordre) les octets (groupes de 2 quartets) lus dans le fichier 1 par 1 dans la mémoire.

Voici, à titre indicatif, des définitions possibles pour ces constantes :

```
#define COREWAR_EXEC_MAGIC 0x726f6342
#define PROG_NAME_LENGTH 64
#define PROG_COMMENT_LENGTH 256
```

Index et Table des matières

A

adressage relatif.....	4
affichage.....	5
arguments.....	4
'AUTHORS'.....	2

B

big-endian.....	4
-----------------	---

C

compteurs.....	3
CYCLES_DELTA.....	3, 6
CYCLES_TO_DEATH.....	3, 6

D

date de rendu.....	2
début de partie.....	5

E

endianness.....	4
EPOCH.....	3, 6

F

fork.....	6
-----------	---

G

groupes.....	2
--------------	---

I

IDX_MOD.....	7
--------------	---

L

ligne de commande.....	4
little-endian.....	4, 10
live.....	3, 5, 6

M

'Makefile'.....	2
mémoire, accès simultanés.....	6
messages.....	5

N

newsgroup.....	2
----------------	---

P

P.....	3, 8, 11
PC.....	3
pile.....	3

R

registres.....	3
rendu.....	2

Z

Z.....	3, 4, 9
--------	---------

Table of Contents

1	Considérations générales	2
1.1	Indications administratives	2
1.2	Modalités de rendu	2
1.3	Contraintes techniques	2
2	La machine virtuelle	3
2.1	Fonctionnement général	3
2.2	Les processeurs	3
2.3	Interaction	4
2.3.1	Syntaxe de la ligne de commande	4
2.3.2	Messages	5
2.4	Initialisation	5
2.5	Parallélisme	5
2.5.1	Destruction des processeurs	6
2.5.2	Concurrence des accès mémoire	6
2.5.3	Fonctionnement de <code>fork</code>	6
2.5.4	Anti-jeu	6
2.6	Le monde du relatif	7
3	Tables des instructions	8
3.1	Description des instructions	8
3.2	Encodage	10
3.3	Délais	10
3.4	Statistiques	11
4	Assembleur	13
4.1	Fonctionnement	13
4.2	Les séparateurs et les commentaires	13
4.3	Les directives	13
4.4	Les labels	14
4.5	Les expressions	14
4.6	Indications supplémentaires	14
4.7	Format des fichiers	15
	Index et Table des matières	16