



Projet Algorithmie I

corewar

42 staff staff@42.fr

Résumé: Ce projet a pour but de vous faire réaliser une "arène" virtuelle dans laquelle vont s'affronter des programmes (les "champions"). Vous allez également réaliser un assembleur permettant de compiler ces champions, ainsi qu'un champion pour montrer que vous savez créer de la vie à partir de café.

Table des matières

| | | |
|-------------|--|-----------|
| I | Préambule | 2 |
| I.1 | Avant toute choses | 2 |
| I.2 | Musiques conseillées | 3 |
| II | Introduction | 4 |
| II.1 | C'est quoi, le Corewar ? | 4 |
| II.2 | Découpage du projet | 4 |
| III | La machine virtuelle | 6 |
| IV | L'assembleur | 8 |
| V | Le champion | 9 |
| VI | Le langage et la compilation | 10 |
| VI.1 | Le langage assembleur | 10 |
| VI.2 | Encodage | 12 |
| VI.2.1 | Exemple complet de compilation | 12 |
| VI.3 | Exécution des champions | 12 |
| VII | Le championnat | 14 |
| VIII | Les bonus | 15 |
| IX | Consignes | 16 |

Chapitre I

Préambule

I.1 Avant toute choses

Nous tenons à vous présenter nos plus plates excuses pour la similarité apparente de ce sujet avec un sac de noeuds, un carton de pelotes de laine après le passage d'une douzaine de chatons surexcités, ou encore le processus de décision de **Bernard Tapie** après avoir pris de la drogue.

En effet, suite à un malheureux concours de circonstances (impliquant notamment **zaz**, une chèvre morte depuis une semaine, une barrique de crevettes, la roulotte d'une bohémienne yougoslave, dix-sept pièces d'or, de la **marmelade d'orange à l'ancienne** et 3.17 milligrammes de **LSD**), ce sujet est devenu complètement incompréhensible, et nos plus grands experts ont été incapables d'en comprendre un traître mot (Le dernier a avoir essayé a été retrouvé trois jours plus tard dans le sous-sol de l'école, nu comme un ver, en train de mâchouiller son pied droit en baragouinant l'**hymne national roumain**).

Voici donc pour vous le moment de faire montre de vos talents de chercheur de secrets. Vous allez certainement devoir avoir recours à des stratagèmes que d'aucuns qualifieraient de malsains pour comprendre exactement de quoi il retourne ici. Voici quelques uns de ces moyens (Âmes sensibles, détournez le regard) : une **lecture** attentive (Berk!), l'**observation** du comportement des programmes de référence (**DÉGUEULASSE!**), ou encore une **réflexion** poussée suivie d'une **discussion** constructive entre vous (Aaaah, **IMMONDE!**).

Nous serons avec vous en pensées pour vous soutenir pendant ce calvaire insoutenable. Si d'aventure vous veniez à ne pas y survivre, sachez que vos noms seront inscrits au feutre rose sur le **Mur des Héros** par un stagiaire privé de sommeil, sous les yeux de 77 jeunes femmes aux cheveux verts, le tout accompagné par un système Hi-Fi nous faisant profiter en boucle des oeuvres d'**Iron Butterfly**.

Bonne chance à tous !

I.2 Musiques conseillées

Afin de vous mettre dans l'état d'esprit moitié transcendantal, moitié tordu, nécessaire à une bonne compréhension de ce sujet, voici quelques groupes à écouter, de préférence en portant une camisole de force, un casque vissé sur vos petites oreilles, et le volume à fond.

Pour une fois, on va varier un peu, et ne pas vous proposer (que) du métal.

- [Loituma](#)
- [Punish Yourself](#)
- [Korpiklaani](#)
- [Turmion Kätilöt](#)
- [Centhron](#)
- [Oldelaf](#)
- [Sexy Sushi](#)
- [Infected Mushroom](#)
- [Sabaton](#)
- [Psyclon Nine](#)
- [Lindsey Stirling](#)
- [Phosgore](#)
- [Duke Ellington](#)
- [Venom](#)
- [Grendel](#)
- [Spintronic](#)
- [Wumpscut](#)
- [Les Fatals Picards](#)
- [Grave Digger](#)
- [Nachtmahr](#)
- [Heimataerde](#)
- [Juno Reactor](#)
- [Aesthetic Perfection](#)
- [Les Choeurs de l'Armée Rouge](#)
- [Puppetmastaz](#)
- [Keep of Kalessin](#)
- [Les Svinkels](#)
- [Combichrist](#)
- [Slayer](#)
- [Storm Weather Shanty Choir](#)
- [Dropkick Murphys](#)

Ce projet est plus facile si vous le réalisez en écoutant l'intégralité de cette liste en boucle.

Chapitre II

Introduction

II.1 C'est quoi, le Corewar ?

- Le Corewar est un jeu très particulier. Il consiste à rassembler autour d'une "machine virtuelle" des "joueurs", lesquels vont y charger des "champions" qui vont se battre à l'aide de "processus", dans le but, entre autres, de faire en sorte qu'on dise d'eux qu'ils sont "en vie".
- Les processus s'exécutent séquentiellement au sein de la même machine virtuelle, et du même espace mémoire. Ils peuvent donc, entre autre chose, s'écrire les uns sur les autres afin de se corrompre mutuellement, de forcer les autres à exécuter des instructions qui leur font du mal, de tenter de recréer à la volée l'équivalent logiciel d'un *Côtes du Rhône 1982*, etc ...
- Le jeu se termine quand plus aucun processus n'est en vie. À ce moment là, le gagnant est le dernier joueur à avoir été rapporté comme étant "en vie".

II.2 Découpage du projet

Le projet consiste à rendre trois parties distinctes :

- **L'assembleur** : C'est le programme qui va compiler vos champions et les traduire du langage dans lequel vous allez les écrire (l'assembleur) vers un "bytecode", à savoir un code machine qui sera directement interprété par la machine virtuelle.
- **La machine virtuelle** : C'est l'"arène" dans laquelle les champions vont s'exécuter. Elle offre de nombreuses fonctionnalités, toutes utiles au combat des champions. Il va de soi qu'elle permet d'exécuter plusieurs processus en simultané ; on vous demande une arène, pas un **simulateur de one-man show**.
- **Le champion** : C'est un cas un peu particulier. Plus tard, pour le championnat, vous allez devoir rendre un champion si puissant et effrayant qu'il ferait trembler de peur un bocalien. Cependant, comme cela constitue en soi un travail conséquent, et que pour l'instant on est juste intéressés par votre capacité à réaliser les autres programmes du Corewar, et que votre champion du moment ne sert qu'à nous prouver que vous savez écrire des bouts d'ASM de Corewar, le champion à rendre dans le cadre de ce projet précis n'a besoin d'effrayer qu'un hérisson neurasthénique.

Il se déroulera également un championnat de Corewar, pour lequel vous aller réaliser de nouveaux champions, qui vont s'affronter lors d'une succession de batailles épiques, dont le point culminant fera passer les jeux du cirque pour la sieste à la garderie du coin.

Notez que le championnat est bien un projet *séparé*, pour lequel vous allez rendre un *nouveau* champion. Il serait donc pertinent de garder vos stratégies les plus machiavéliques pour vous, afin de ne pas devenir, pour ainsi dire, le dindon de la farce.

Chapitre III

La machine virtuelle

- Chaque processus aura à sa disposition les éléments suivants, qui lui sont propres :
 - REG_NUMBER registres qui font chacun une taille de REG_SIZE octets. Un registre est une petite "case" mémoire, qui ne contient qu'une seule valeur. Sur une vraie machine, elle est interne au processeur et est donc TRÈS rapide d'accès.
 - Un PC ("Program Counter"). C'est un registre spécial, qui contient juste l'adresse, dans la mémoire de la machine virtuelle, de la prochaine instruction à décoder et exécuter. Très utile pour savoir où l'on se trouve dans l'exécution, afin d'écrire des choses en mémoire...
 - Un flag nommé **carry**, qui vaut 1 si la dernière opération a réussi. Seules certaines opérations vont modifier le **carry**.
- Le numéro du joueur est généré par la machine ou spécifié au lancement, et est fourni aux champions via le registre r1 de leur premier processus au démarrage. Tous les autres registres sont mis à 0. Sauf le PC.
- Les champions sont chargés en mémoire de façon à espacer équitablement leurs points d'entrée.
- La machine virtuelle va créer un espace mémoire dédié au combat des joueurs, puis y charger les champions et leurs processus associés, et les exécuter séquentiellement jusqu'à ce que mort s'ensuive.
- Tous les CYCLE_TO_DIE cycles, la machine doit s'assurer que chaque processus a exécuté au moins un **live** depuis la dernière vérification. Un processus qui ne se soumet pas à cette règle sera mis à mort à l'aide d'une batte en mousse virtuelle. (Bonus bruitage!)
- Si au cours d'une de ces vérifications on se rend compte qu'il y a eu au moins NBR_LIVE exécutions de **live** depuis la dernière vérification en date, on décrémente CYCLE_TO_DIE de CYCLE_DELTA unités.
- Quand il n'y a plus de processus en vie, la partie est terminée.
- Le gagnant est le dernier joueur qui a été rapporté comme étant en vie. La machine

va ensuite afficher : "le joueur x(nom_champion) a gagne", où x est le numéro du joueur et nom_champion le nom de son champion.

Exemple : "le joueur 2(rainbowdash) a gagne"

- A chaque exécution valide de l'instruction `live`, la machine doit afficher : "un processus dit que le joueur x(nom_champion) est en vie"
- En tout état de cause, la mémoire est circulaire et fait `MEM_SIZE` octets.
- En cas d'erreur, vous devrez afficher un message pertinent sur la sortie d'erreur.
- Si on n'a pas décrémenté `CYCLE_TO_DIE` depuis `MAX_CHECKS` vérifications, on le décrémente.
- La machine virtuelle se lance de la façon suivante :

```
> ./corewar [-dump nbr_cycles] [[-n number] champion1.cor] ...
```

- `-dump nbr_cycles`
Au bout de `nbr_cycles` cycles d'exécution, dump la mémoire sur la sortie standard, puis quitte la partie. La mémoire doit être dumpée au format hexadécimal, avec 32 octets par ligne.
- `-n number`
Fixe le numéro du prochain joueur. Si absent, le joueur aura le prochain numéro libre dans l'ordre des paramètres. Le dernier joueur aura le premier processus dans l'ordre d'exécution.
- Les champions ne peuvent pas dépasser `CHAMP_MAX_SIZE`, sinon c'est une erreur.

Chapitre IV

L'assembleur



Nous interrompons ce sujet pour vous livrer ce flash d'information :
Selon nos sources, un duoquadragentien sur sept aime les odeurs bleues.

- Votre machine virtuelle va exécuter du code machine (ou "bytecode"), qui devra être généré par votre assembleur. L'assembleur (le programme) va prendre en entrée un fichier écrit en assembleur (le langage), et sortir un champion qui sera compréhensible par la machine virtuelle.
- Il se lance de la façon suivante :

```
> ./asm monchampion.s
```
- Il va lire le code assembleur à traiter depuis le fichier `.s` passé en paramètre, et écrire le bytecode résultant dans un fichier nommé comme l'entrée en remplaçant l'extension `.s` par `.cor`.
- En cas d'erreur, vous devrez afficher un message pertinent sur la sortie d'erreur, et ne pas produire de fichier `.cor`

Chapitre V

Le champion

- Votre champion a trois objectifs indissociables : Faire en sorte que son joueur soit rapporté comme "en vie", comprendre le sens de ladite vie, et annihiler ses adversaires.
- Pour que son joueur soit dit comme "en vie", votre champion doit faire en sorte que des **live** soient faits avec son numéro. Si l'un de ses processus fait un **live** avec le numéro d'un autre joueur ... eh bien, c'est dommage, mais au moins un autre joueur sera content. Si un processus d'un autre joueur fait des **live** avec votre numéro, vous avez l'autorisation de vous moquer de lui et de profiter éhontément de son erreur, tous en insultant sa famille en binaire.
- Absolument TOUTES les instructions sont utiles. Toutes les réactions de la machine, décrites plus loin dans le chapitre sur le langage, peuvent être utilisées pour donner vie à votre champion et lui permettre de remporter dix-sept euros cinquante-trois centimes au cours du championnat. Oui, même l'instruction **aff** est utile, par exemple pour se moquer de l'ineptitude de vos adversaires.
- Il sera noté, à la soutenance, sur sa capacité à survivre à quelques challenges élémentaires, comme vaincre un champion au Q.I de pot de fleurs, réussir à finir une assiette de tarte aux pommes de ma grand-mère, ou dessiner des fleurs dans un cappuccino.
- Vous pourrez, plus tard, réaliser un nouveau champion qui sera destiné à participer au championnat, (Rappel : C'est un autre projet !) et se battre contre les champions de vos camarades, et peut-être même ceux du Bocal, chose qui transformera probablement assez rapidement votre champion en tas informe de boyaux virtuels, mais qui pourrait bien, au prix d'une ou deux cérémonies vaudou impliquant des **épingles à nourrice** et un endroit que l'honnêteté et la décence m'interdisent de préciser davantage, vous couvrir de **gloire** et de **chatons**.

Chapitre VI

Le langage et la compilation

VI.1 Le langage assembleur

- Le langage assembleur est composé d'une instruction par ligne.
- Une instruction se compose de trois éléments : Un label (optionnel), composé d'une chaîne de caractères parmi LABEL_CHARS suivi par LABEL_CHAR ; un opcode ; et ses paramètres, séparés par SEPARATOR_CHAR. Un paramètre peut être de trois types :
 - Registre : (r1 <-> rx avec x = REG_NUMBER)
 - Direct : Le caractère DIRECT_CHAR suivi d'une valeur numérique ou d'un label (précédé par LABEL_CHAR), ce qui représente une valeur directe.
 - Indirect : Une valeur ou un label (précédé de LABEL_CHAR), ce qui représente la valeur qui se trouve à l'adresse du paramètre, relativement au PC du processus courant.
- Un label peut n'avoir aucune instruction à sa suite, ou être placé sur la ligne d'avant l'instruction qu'il concerne.
- Le caractère COMMENT_CHAR démarre un commentaire.
- Un champion comportera également un nom et une description, qui sont présents sur une ligne après les marqueurs NAME_CMD_STRING et COMMENT_CMD_STRING.
- Tous les adressages sont relatifs au PC et à IDX_MOD sauf pour lld, lldi et lfork.
- Le nombre de cycles de chaque instruction, leur représentation mnémonique, leur nombre de paramètres et les types de paramètres possibles sont décrits dans le tableau op_tab déclaré dans op.c. Les cycles sont toujours consommés.
- Tous les autres codes n'ont aucune action à part passer au suivant et perdre un cycle.
- lfork : Ca signifie long-fork, pour pouvoir fourcher de la paille à une distance de 15 mètres, exactement comme son opcode. Pareil qu'un fork sans modulo à

l'adresse.

- **sti** : Opcode 11. Prend un registre, et deux index (potentiellement des registres). Additionne les deux derniers, utilise cette somme comme une adresse ou sera copiée la valeur du premier paramètre.
- **fork** : Pas d'octet de codage des paramètres, prend un index, opcode 0x0c. Crée un nouveau processus, qui hérite des différents états de son père, à part son PC, qui est mis à $(PC + (1er\ paramètre \% IDX_MOD))$.
- **lld** : Signifie **long-load**, donc son opcode est évidemment 13. C'est la même chose que **ld**, mais sans $\% IDX_MOD$. Modifie le carry.
- **ld** : Prend un paramètre quelconque et un registre. Charge la valeur du premier paramètre dans le registre. Son opcode est 10 en binaire, et il changera le carry.
- **add** : Opcode 4. Prend trois registres, additionne les 2 premiers, et met le résultat dans le troisième, juste avant de modifier le carry.
- **zjmp** : Il n'y a jamais eu, n'y a pas, et n'y aura jamais d'octet de codage des paramètres derrière cette opération dont l'opcode est de 9. Elle prendra un index, et fait un saut à cette adresse si le carry est à 1.
- **sub** : Pareil que **add**, mais l'opcode est 0b101, et utilise une soustraction.
- **ldi** : **ldi**, comme son nom l'indique, n'implique nullement de se baigner dans de la crème de marrons, même si son opcode est 0x0a. Au lieu de ça, ça prend 2 index et 1 registre, additionne les 2 premiers, traite ça comme une adresse, y lit une valeur de la taille d'un registre et la met dans le 3eme.
- **or** : Cette opération est un OU bit-à-bit, suivant le même principe que **and**, son opcode est donc évidemment 7.
- **st** : Prend un registre et un registre ou un indirect, et stocke la valeur du registre vers le second paramètre. Son opcode est 0x03. Par exemple, **st r1, 42** stocke la valeur de r1 à l'adresse $(PC + (42 \% IDX_MOD))$
- **aff** : L'opcode est 10 en hexadécimal. Il y a un octet de codage des paramètres, même si c'est un peu bête car il n'y a qu'un paramètre, qui est un registre, dont le contenu est interprété comme la valeur ASCII d'un caractère à afficher sur la sortie standard. Ce code est modulo 256.
- **live** : L'instruction qui permet à un processus de rester vivant. A également pour effet de rapporter que le joueur dont le numéro est en paramètre est en vie. Pas d'octet de codage des paramètres, opcode 0x01. Oh, et son seul paramètre est sur 4 octets.
- **xor** : Fait comme **and** avec un OU exclusif. Comme vous l'aurez deviné, son opcode en octal est 10.
- **lldi** : Opcode 0x0e. Pareil que **ldi**, mais n'applique aucun modulo aux adresses. Modifiera, par contre, le carry.

- **and** : Applique un & (ET bit-à-bit) sur les deux premiers paramètres, et stocke le résultat dans le registre qui est le 3ème paramètre. Opcode 0x06. Modifie le carry.

VI.2 Encodage

Chaque instruction est encodée par :

- Le code de l'instruction (on le trouve dans `op_tab`).
- L'octet de codage des paramètres, si c'est approprié. À faire comme le montrent les exemples suivants :
 - `r2,23,%34` donne l'octet de codage `0b01111000`, soit `0x78`
 - `23,45,%34` donne l'octet de codage `0b11111000`, soit `0xF8`
 - `r1,r3,34` donne l'octet de codage `0b01011100`, soit `0x5C`
- Les paramètres, directement, selon le modèle suivant :
 - `r2,23,%34` donne l'OCP `0x78` puis `0x02 0x00 0x17 0x00 0x00 0x00 0x22`
 - `23,45,%34` donne l'OCP `0xF8` puis `0x00 0x17 0x00 0x2d 0x00 0x00 0x00 0x22`

Quelques notes importantes :

- L'exécutable commence toujours par un header, défini dans `op.h` par le type `header_t`
- La machine virtuelle est BIG ENDIAN. Demandez à Google ce que cela veut dire.

VI.2.1 Exemple complet de compilation

```
.name "zork"
.comment "just a basic living prog"

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

live:  live %1
      zjmp %:live

# Executable compile:
#
# 0x0b,0x68,0x01,0x00,0x0f,0x00,0x01
# 0x06,0x64,0x01,0x00,0x00,0x00,0x00,0x01
# 0x01,0x00,0x00,0x00,0x01
# 0x09,0xff,0xfb
```

VI.3 Exécution des champions

- La machine virtuelle est supposée émuler une machine parfaitement parallèle.
- Mais pour des raisons d'implémentation, on supposera que chaque instruction s'exécute entièrement à la fin de son dernier cycle et attend durant toute sa durée. Les instructions qui se terminent à un même cycle s'exécutent dans l'ordre décroissant des numéros de processus.

- Oui, le dernier né joue en premier.

Chapitre VII

Le championnat

- À un certain moment, déterminé par un rituel cabalistique, se déroulera le championnat de Corewar.
- Vos champions vont s'affronter les uns les autres au cours de cet événement épique, et vont sûrement se frotter aux champions du Bocal...
- Les gagnants seront couverts :
 - de gloire;
 - d'alcool;
 - de goodies;
 - de chatons.
- Les perdants, quant à eux, seront :
 - traînés dans la boue;
 - hués;
 - la risée des ouvriers/ouvrières de rue.
 - contraints d'observer leur champion se faire bifler.
- Ce championnat sera un projet à part, et vous n'aurez qu'à y rendre un champion (Pas forcément le même que pour votre Corewar, pour des raisons évidentes de secret et de stratégie...). Il sera exécuté sur notre propre machine virtuelle, donc la configuration est celle qui est décrite dans le fichier `op.h` fourni en annexe. Faites attention, les fichiers `op.c` et `op.h` sont fournis à titre indicatif et vous aurez certainement besoin de les modifier. Il est fort possible qu'ils ne marchent pas, suite à une malencontreuse méprise entre une bouteille d'eau et une bouteille de vodka.

Chapitre VIII

Les bonus



Les bonus ne seront évalués que si votre partie obligatoire est EXCELLENTE. On entend par là qu'elle est entièrement réalisée, que votre gestion d'erreur est au point, même dans des cas vicieux, ou des cas de mauvaise utilisation.

Après avoir avec succès réalisé un **Corewar** complet et digne d'être immortalisé en ayant l'intégralité de son code gravé dans une plaque d'acajou par un TIG, vous pouvez tenter de rajouter des bonus. Les possibilités sont innombrables ! Cependant, gardez bien en tête que toute erreur dans l'un de ces bonus risque d'invalider l'intégralité de votre travail. Il convient donc d'être rigoureux !

Voici quelques idées :

- Une interface graphique pour la machine virtuelle, à votre choix. (OpenGL, SDL, nCurses, ... ce qui vous amuse !)
- Un mode de jeu en réseau
- De nouvelles instructions
- Le support d'opérations mathématiques dans les fichiers .s

Encore une fois, soyez vigilants quant à la qualité de votre bonus, et gardez de bonnes priorités. Le **Corewar** n'est pas un projet trivial, et il est très facile de prendre malencontreusement un peu trop de drogue et d'invalider un mois de travail en faisant une bêtise...

Chapitre IX

Consignes

- Ce projet ne sera corrigé que par des humains. Vous êtes donc libres d'organiser et nommer vos fichiers comme vous le désirez, en respectant néanmoins les contraintes listées ici.
- Vos exécutables doivent s'appeler **asm** et **corewar**
- Votre champion doit avoir un nom majestueux, épique et glorieux.
- Vous devez rendre un Makefile.
- Votre Makefile devra compiler le projet, et doit contenir les règles **libft** qui devra être à la racine de votre dépôt de rendu. Votre **Makefile** devra compiler la librairie, en appelant son **Makefile**, puis compiler votre projet.
- Votre projet doit être à la Norme. Même si, grâce à la puissance de la drogue, le notre ne l'est pas.
- Vous devez gérer les erreurs de façon pertinente. En aucun cas votre programme ne doit quitter de façon inattendue (Segmentation fault, etc...).
- Vous devez rendre, à la racine de votre dépôt de rendu, un fichier **auteur** contenant vos logins, à raison d'un par ligne, de cette façon :

```
$>cat -e auteur
xlogin$
ylogin$
zlogin$
alogin$
$>
```

- Vous avez le droit d'utiliser les fonctions suivantes :
 - open
 - read
 - write
 - lseek
 - close
 - malloc

- realloc
 - free
 - exit
 - perror / strerror (et, du coup, errno)
- Vous pouvez poser vos questions sur le forum, sur jabber, IRC, ...
- Nous vous fournirons un assembleur et une machine virtuelle qui fonctionnent exactement comme il faut. Par contre, vous n'aurez jamais leur code source. Il va falloir réfléchir.
- Bon courage à tous !