EPITECH Nice

Aide Corewar

Guide pour bien débuter

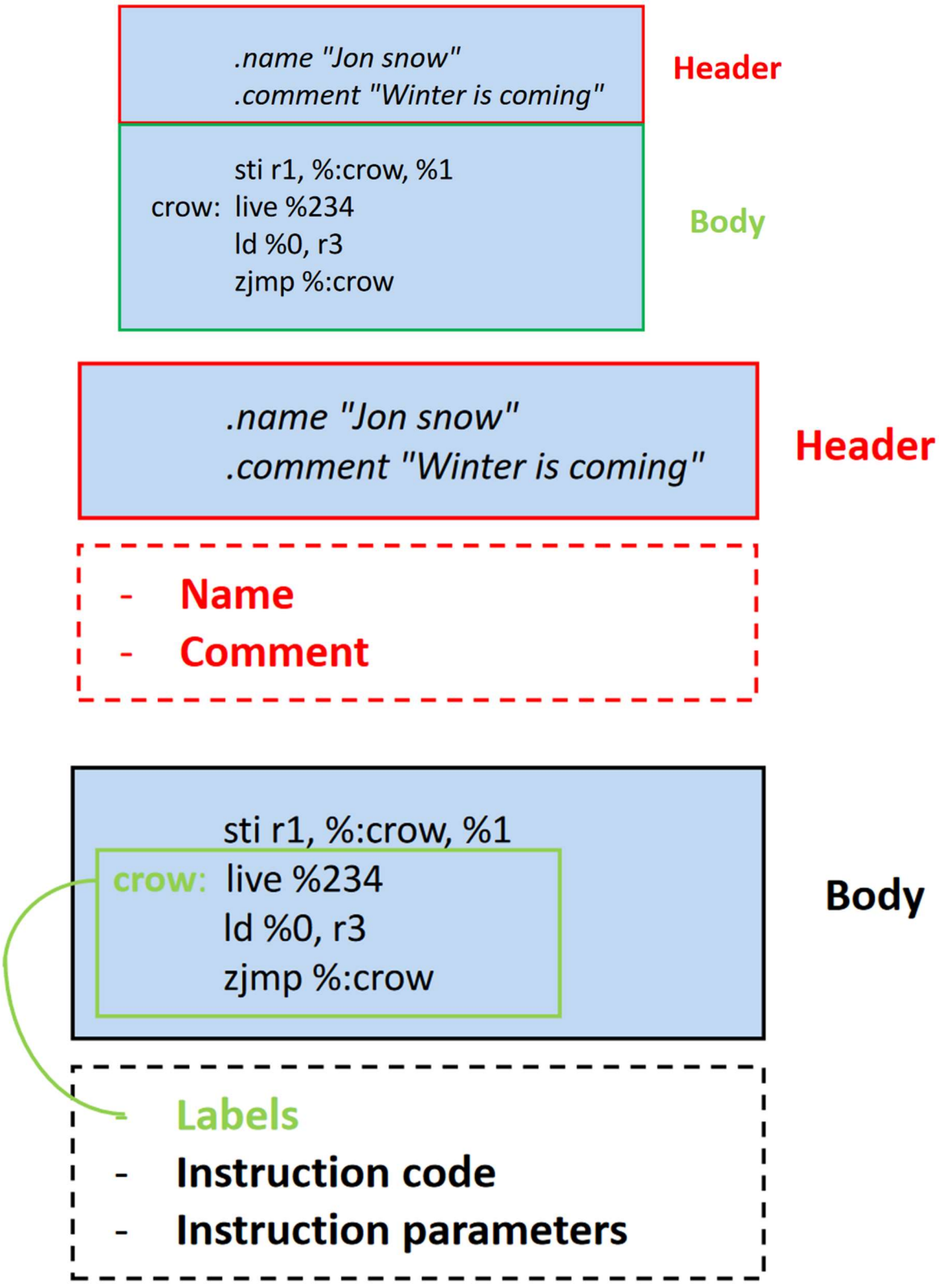
Eric Deschodt
24/04/2023

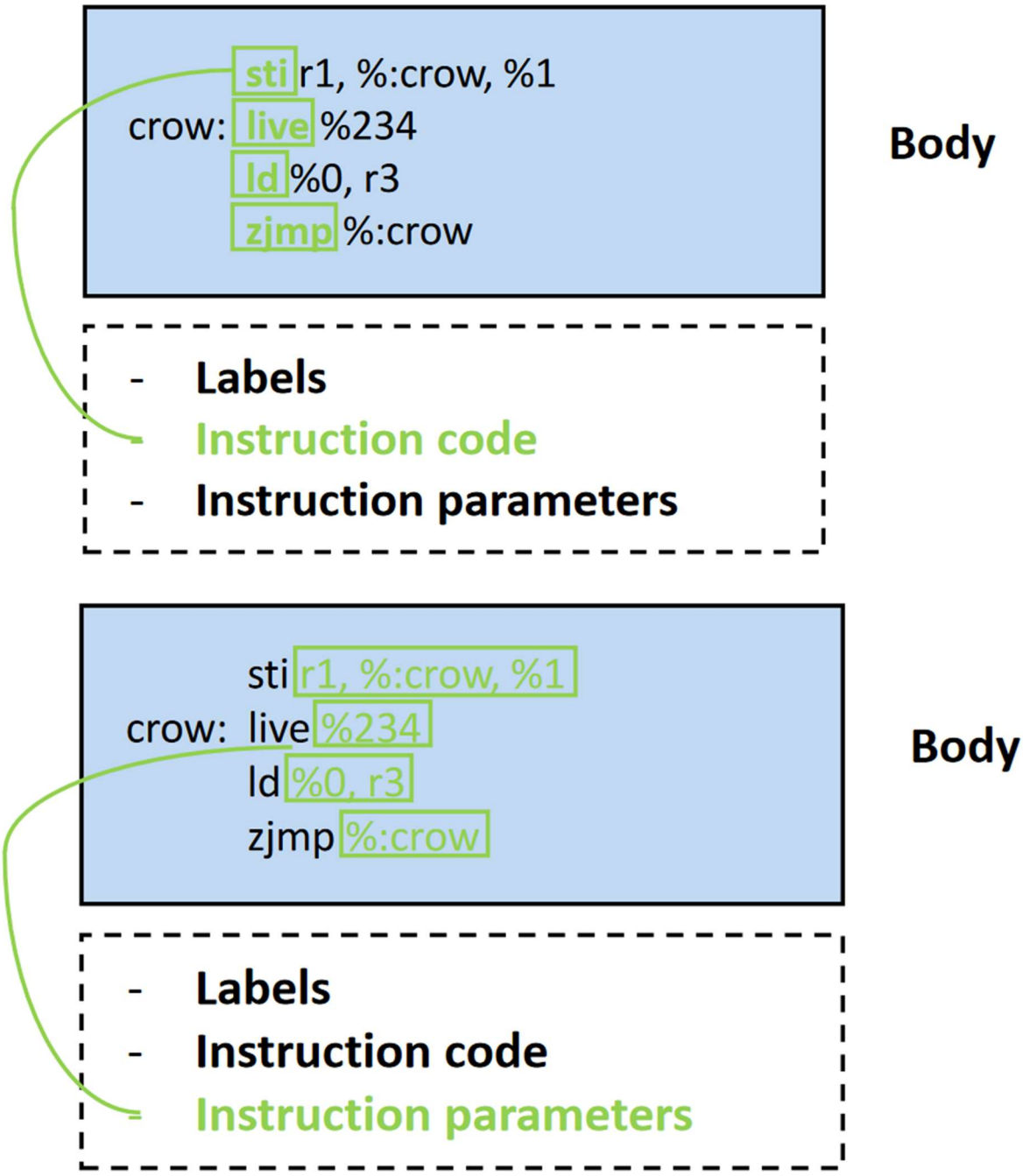
Table des matières

Assembleur 2
1. Le champion .s 2
2. Paramètres des instructions 4
Bytecode 5
1. Transcription 5
2. Instructions et paramètres 6
3. Récapitulatif 8
Machine virtuelle 9
1. Mise en place des champions 9
2. Déroulé d'un tour 10
3. Synchronisation 12
4. Arbitrage 13

Assembleur

1. Le champion .s




sti r1, %:crow, %1
crow: live %234
ld %0, r3
zjmp %:crow
Body
- Labels
- Instruction code
- Instruction parameters
sti r1, %:crow, %1
crow: live %234
ld %0, r3
zjmp %:crow
Body
- Labels
- Instruction code
- Instruction parameters

2. Paramètres des instructions

Registre

Un registre est un emplacement de mémoire interne à un processeur, c'est la mémoire la plus rapide d'un ordinateur.
Pour simplifier les registres dans notre ordinateur simulé, chaque processeur aura un nombre de registre **REG_NUMBER*** qui peut contenir un entier.

Par exemple :
`sti r1, %crow, %1`
r1 signifie que le contenu du **registre 1** est manipulé, dans ce cas l'instruction sti copie le contenu **du registre 1** à un autre endroit.

Direct

Un paramètre de type **direct** est un paramètre dont la valeur est directement celle indiquée.
Par exemple :
`and %4 %3 r3`
Une valeur **directe** est toujours précédée du caractère **DIRECT_CHAR*,** dans ce cas '%'.
Ici, nous plaçons le résultat logique **AND** de **'4'** et **'3'** dans le registre 3.

Indirect

Un paramètre de type **indirect** est une adresse dans la zone de mémoire où se trouve la valeur souhaitée.
Par exemple :
`ld 4, r5`
Dans ce cas, nous chargeons la valeur de la **position 4** dans le registre 5.

Label

Il est possible d'obtenir un label en le faisant précéder de LABEL_CHAR*, dans le cas présent ':'.
Cela permet par exemple dans des 'zjump' de pouvoir spécifier que l'on veut aller dans le code à la position du label.

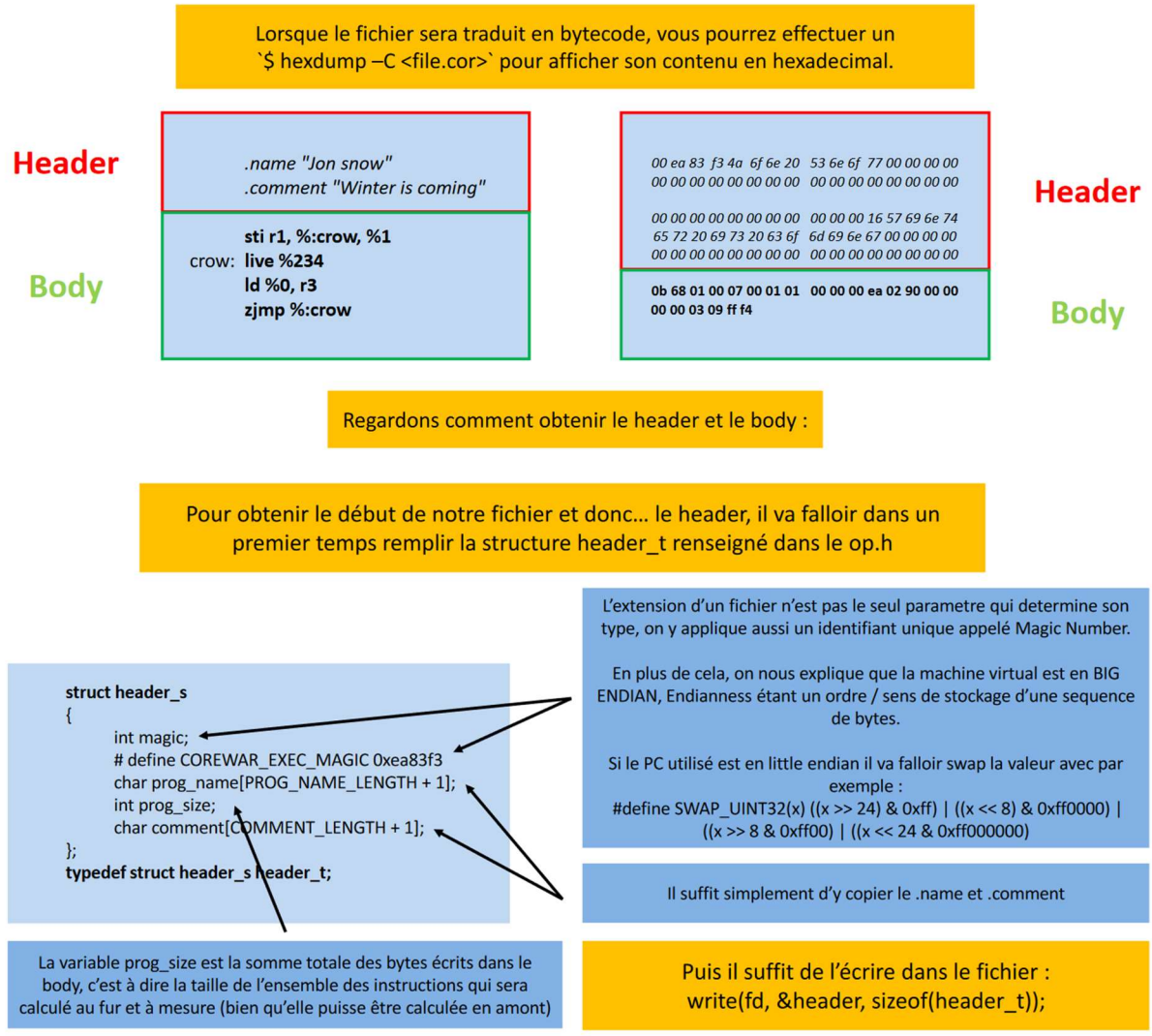
Bytecode

1. Transcription

```
        .name "Jon snow"
        .comment "Winter is coming"

        sti r1, %:crow, %1
crow:   live %234
        ld %0, r3
        zjmp %:crow
```

```
00 ea 83  f3 4a  6f 6e 20   53 6e 6f  77 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00   00 00 00 16 57 69 6e 74
65 72 20 69 73 20 63 6f   6d 69 6e 67 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0b 68 01 00 07 00 01 01   00 00 00 ea 02 90 00 00
00 00 03 09 ff f4
```

Transformer l'asembleur en un code machine en byte-code



Comment obtient-on maintenant le corps du fichier ? La transcription des instructions ?

```
00 ea 83  f3 4a  6f 6e 20   53 6e 6f  77 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00   00 00 00 16 57 69 6e 74
65 72 20 69 73 20 63 6f   6d 69 6e 67 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0b 68 01 00 07 00 01 01   00 00 00 ea 02 90 00 00
00 00 03 09 ff f4
```

```
        .name "Jon snow"
        .comment "Winter is coming"

        sti r1, %:crow, %1
crow:   live %234
        ld %0, r3
        zjmp %:crow
```

Lorsqu'on connait le truc, on arrive facilement à distinguer nos 4 instructions (le nom du label n'est jamais retranscris, on ne traduit que les instructions et leurs paramètres)

sti r1, %:crow, %1	**-**	**live %234**	**-**	**ld %0, r3**	**-**	**zjmp %:crow**
0b 68 01 00 07 00 01	**-**	**01 00 00 00 ea**	**-**	**02 90 00 00 00 00 03**	**-**	**09 ff f4**

2. Instructions et paramètres

Chaque instructions se traduira toujours comme ça : <inst_code> [coding_byte] <params>
Un code d'instruction, un coding_byte (optionnel parfois) et ses params.

<INST_CODE>

Le code d'instruction est renseigné dans le fichier op.c pouvant aller de 1 à 16.

live: 1, ld: 2, st: 3, add: 4, sub: 5, and: 6, or: 7, xor: 8, zjmp: 9, ldi: 10, sti: 11, fork: 12, lld: 13, lldi: 14, lfork: 15, aff: 16

Le code d'instruction sera toujours écrit sur 1 byte. Exemple : ld -> 0x02

Il suffira simplement de : write(fd, &code, 1);

[coding_byte]

Le coding_byte est la valeur permettant de savoir quelles paramètres surviennent après le code une fois en code machine. Pour le calculer il faut se réserver 4 bytes et procéder ainsi :

char codingbyte[8] = [_ _ _ _ _ _ _ _]; L'idée est d'écrire en binaire la valeur de chacun des arguments. Si l'argument est un registre on écrit 01, si c'est un direct 10, si c'est un indirect 11, sinon 00

Pour `sti r1, %:crow, %1` nous avons un registre, 2 direct et aucun 4e argument donc : [10 11 11 00], il n'y a plus qu'à faire un get_nbr_base du char * en int et l'écrire un un seul byte ! -> 0x68

Attention ! Si l'instruction est zjmp, live, fork ou lfork, il n'y pas de coding byte à écrire, ils ne prennent qu'un seul argument qui est toujours un direct, ce n'est donc pas utile !

Il suffira simplement de : write(fd, &coding_byte, 1);

<param>

Il y a 3 types d'arguments à transcrire, les registres, les directs et les indirects.

<param> : registre

Les registres, pas de suprise, s'écrivent sur 1 byte comme les codes d'instructions.

Exemple : `sti r1, %:crow, %1`

On va récupérer sa valeur, qui est 1 et l'écrire sur 1 byte -> 0x01

<param> : direct

Là ça se complique, **par défaut**, un direct s'écrit sur 4 bytes

Exemple : `and %4, %3, r1`

La transcription de %4 serait -> 0x00 0x00 0x00 0x04

Sauf qu'encore il y a une exception, si l'instruction est zjmp, ldi, sti, fork, lldi ou lfork nous ne l'écrivons que sur 2 ! Mais si on l'écrit que sur 2, nous n'aurons pas 0x00 0x04 mais bien 0x00 0x00 ! Il faut donc faire un shift de << 16 pour obtenir le résultat souhaité !

<param> : indirect

Là c'est plutôt simple maintenant, **par défaut**, un indirect s'écrit sur 2 bytes

Exemple : `and 4, 3, r1`

La transcription de 4 serait -> 0x00 0x04

Il faut penser à shift << de 16 pour avoir le résultat sinon il sera encore une fois perdu sur les 2 bytes non écrits !

<param> : label

Dans l'exemple `sti r1, %:crow, %1` par quoi doit-on remplacer le %:crow ?

Et bien il prend la valeur positive ou negative de la position du label par rapport à la position du code d'instruction qu'il l'appelle !

```
00 ea 83  f3 4a  6f 6e 20  53 6e 6f  77 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00  00 00 00 16 57 69 6e 74
65 72 20 69 73 20 63 6f  6d 69 6e 67 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00

0b 68 01 00 07 00 01 01  00 00 00 ea 02 90 00 00
00 00 03 09 ff f4
```

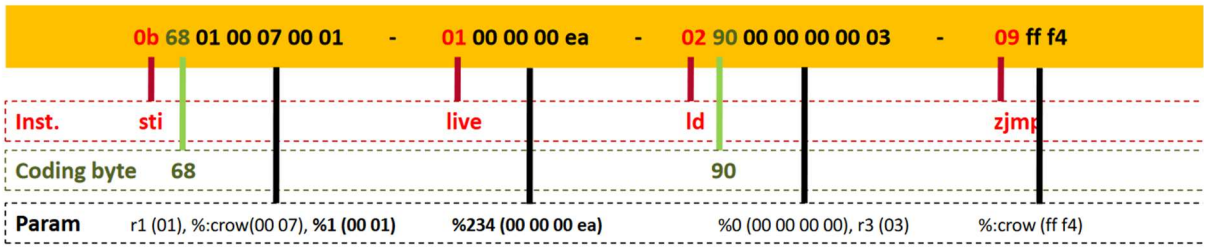
```
        .name "Jon snow"
        .comment "Winter is coming"

        sti r1, %:crow, %1
crow:   live %234
        ld %0, r3
        zjmp %:crow
```

La première fois c'est 0x00 0x07 parce qu'on est à 7 case avant le label et à la fin c'est 0xff 0xf4 par ce qu'on est à -12 devant le label ! A partir de là il est écrit sur un nombre de byte dépendant de son type de variable et de code d'instruction comme jusqu'à maintenant !

3. Récapitulatif

```
00 ea 83  f3 4a  6f 6e 20   53 6e 6f  77 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00   00 00 00 16 57 69 6e 74
65 72 20 69 73 20 63 6f   6d 69 6e 67 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0b 68 01 00 07 00 01 01   00 00 00 ea 02 90 00 00
00 00 03 09 ff f4
```

```
        .name "Jon snow"
        .comment "Winter is coming"

        sti r1, %:crow, %1
crow:   live %234
        ld %0, r3
        zjmp %:crow
```



Machine virtuelle

1. Mise en place des champions

L’arène est juste un array d’élément vide au début du programme.

Attention : Les champions ne possèdent par leurs instructions ! Les champions sont des curseurs qui vont lire les instructions écrites dans l’arène (même si elle se font corrompre par d’autres champions).

Il faut donc prendre le « body » de chacun des champions et écrire leurs instructions directement dans l’arène en byte code de façon équidistante si vous avez 2 / 4 champions.

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

Chaque champion possède des registres, qui est la mémoire du champion accessible uniquement par lui. Le registre 1 contient toujours le numéro du champion.

2. Déroulé d'un tour

Il faut ensuite placer les curseurs sur le début de leurs instructions

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

Champion 1 – Champion 2 – Champion 3 – Champion 4

La machine virtuelle est un comme un RPG au tour par tour. Chaque tour de jeu, le champion va faire une action, puis le 2, puis le 3, puis le 4.

TOUR 1 : Action du champion 1

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

Le curseur lit l'instruction 0b => ce qui correspond à l'instruction ld. Le curseur va donc lire le byte de codage « 68 »

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

Le 68 indique qu'on va avoir 3 paramètres : un registre, et deux distances qui vont s'ajouter.

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

La fonction ld va prendre la valeur contenue dans le premier paramètre, c'est-à-dire le registre 1 (le numéro du champion).
En ajoutant 7 et 1 (les autres paramètres), on indique qu'on veut copier le registre 8 octets plus loin.

	1	2	3	4	5	6	7	8										
0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

Une fois copié le curseur se place sur l'instruction suivante et il entre en cooldown.

0b	68	01	00	07	00	01	01	00	00	00	01	02	90	00	00	00	03	09
ff	14																	

C'est ensuite au tour des autres champions. L'exemple vous montre 4 fois le même champion, sauf que chacun va écrire son propre numéro en écrasant ce qui est écrit sur l'arene

	1	2	3	4	5	6	7	8										
0b	68	01	00	07	00	01	01	00	00	00	02	02	90	00	00	00	03	09
ff	14																	

	1	2	3	4	5	6	7	8										
0b	68	01	00	07	00	01	01	00	00	00	03	02	90	00	00	00	03	09
ff	14																	

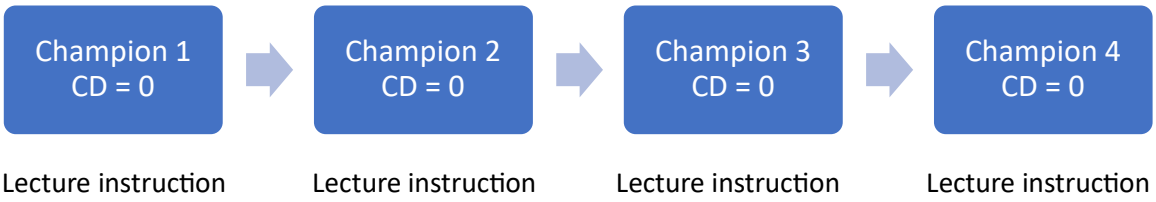
	1	2	3	4	5	6	7	8										
0b	68	01	00	07	00	01	01	00	00	00	04	02	90	00	00	00	03	09
ff	14																	

Les champions peuvent écrire et modifier ce qui est écrit sur le sol de l'arène.
Une des stratégies des champions est de repérer les instruction « live » de leurs adversaires et écrire leur propre numéro dessus !

3. Synchronisation

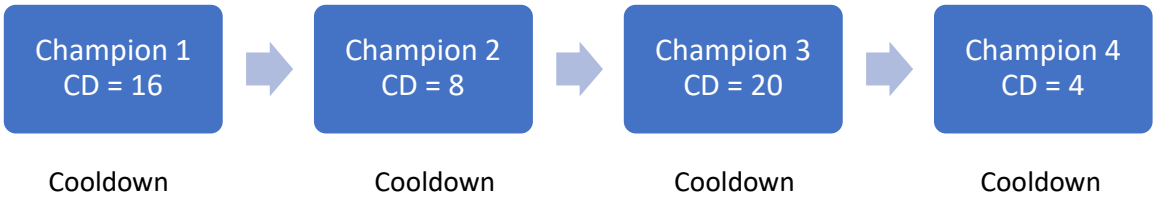
Chaque action applique un Cool down en nombre de tours. Une fois une instruction accomplie, on déplace le curseur sur le byte après les paramètres, et pendant X nombre de tours, le champion ne va pas agir. Certaines instructions sont très longues (par exemple, fork).

Tour 0 :



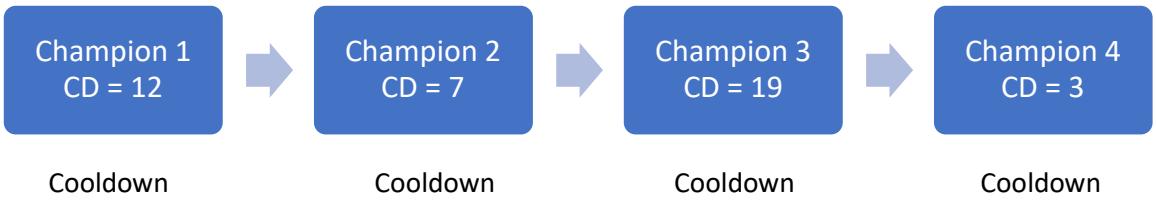
Les champions exécutent leurs instructions, qui ont sûrement des cool down différents.

Tour 1 :



Aucun champion ne prendra d'action ce tour, car tout le monde est en cooldown

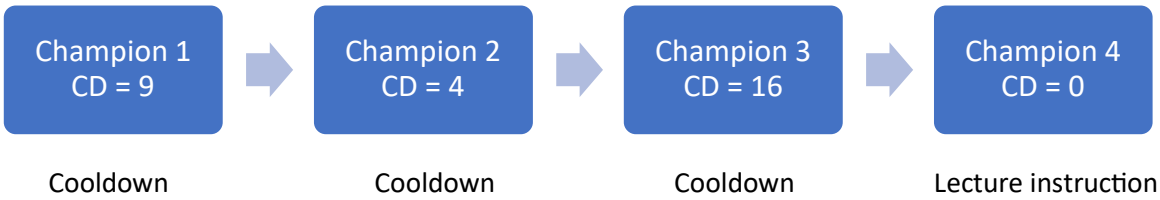
Tour 2 :



Les cooldowns de tous les champions sont toujours atifs => pas d'actions

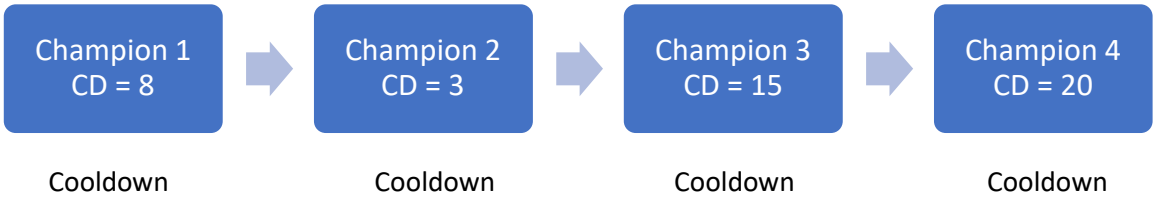
Il ne se passera donc rien pour encore 3 tours

Tour 5 :

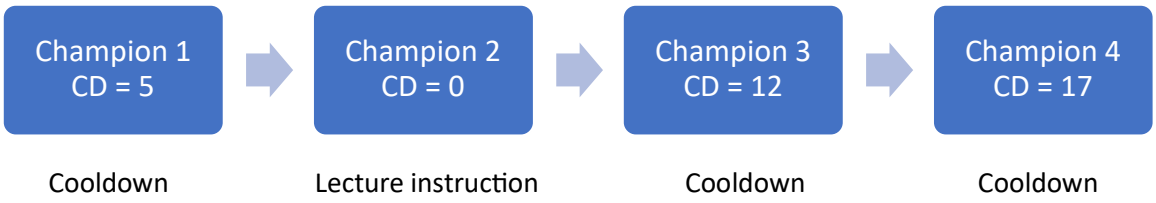


Au tour 5, le champion 4 peut de nouveau lire une instruction qui va lui appliquer un nouveau cooldown, les autres champions ne peuvent encore rien faire

Tour 6 :



Tour 9 :



Le champion 2 a fini son cooldown, il peut lire et appliquer une nouvelle instruction.

4. Arbitrage

Vous devez coder l'arbitre dans la machine virtuelle.
Le DELTA_CYCLE a pour but de vérifier qui est encore en vie.

Exemple : Pour un DELTA_CYCLE qui commence à 100. Cela signifie que au bout de 100 tours, l'arbitre verifie qui est encore en vie en regardant les instructions live qui se sont exécutées avec le numéro correspondant.

Attention : il n'est pas nécessaire que ce soit le curseur du champion 1 qui lance la commande live 00 00 00 01, l'arbitre regarde juste s'il a recu dans le DELTA_CYCLE des commandes live avec le numéro de champion. C'est-à-dire qu'il est possible que le processus d'un champion lance l'instruction « live » pour un autre champion !

S'il n'y a jamais eu d'instruction live avec leur numéro, le champion et tous ses forks disparaissent.

Pour éviter que les parties soient infinie, le DELTA_CYCLE diminue à chaque arbitrage. Au début il pourrait valoir 100, puis 90, puis 80, etc… A vous d'équilibrer votre jeu.

En cas de mort simultanée, c'est le champion avec le plus petit nombre qui gagne (injuste mais il faut bien les départager).