# Assemblage en code d'octet

Jump to bottom

VBrazhnik a modifié cette page on Oct 10, 2019 · 3 révisions

## Structure des fichiers

Pour comprendre à quelles règles le processus de traduction du code source en langage d'assemblage en bytecode obéit, vous devez considérer la structure du fichier avec l'extension `.cor`.

Pour ce faire, nous diffusons ce champion à l'aide du programme fourni par la tâche `asm` :

```
.name       "Batman"
.comment    "This city needs me"

loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

Le fichier résultant aura la structure suivante:


00ea 83f3 4261 746d 616e 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0016 5468 6973
2063 6974 7920 6e65 6564 7320 6d65 0000
MAGIC HEADER
NULL
CHAMPION NAME
CHAMPION EXEC CODE SIZE
CHAMPION COMMENT


## Tête magique

Les 4 premiers octets du fichier sont occupés par le "nombre magique".

Il est défini à l'aide d'une constante `COREWAR_EXEC_MAGIC` dont la valeur dans le fichier d'exemple `op.h` est `0xea83f3`.

### Qu'est-ce qu'un en-tête magique et pourquoi est-il nécessaire?

L'en-tête magique est un nombre si spécial, dont le but est d'indiquer que ce fichier est binaire.

Si le fichier ne contient pas un tel "message", il peut être interprété comme du texte.

## Nom du champion

Les 128 octets suivants du fichier sont occupés par le nom du champion. Pourquoi 128? Il s'agit de la valeur d'une constante `PROG_NAME_LENGTH` qui définit la longueur maximale de la chaîne de nom.

Le nom de notre champion est beaucoup plus court. Mais dans le bytecode, il occupe toujours les 128 octets.

Parce que, selon les règles de traduction, si la longueur de la chaîne avec le nom est inférieure à la limite spécifiée, alors les caractères manquants sont compensés avec zéro octet.

En général, chaque caractère du nom est converti en un code ASCII 1 octet écrit en notation hexadécimale:

| symbole | `B` | `a` | `t` | `m` | `a` | `n` |
|---|---|---|---|---|---|---|
| Code ASCII | `0x42` | `0x61` | `0x74` | `0x6d` | `0x61` | `0x6e` |

Et au lieu de caractères manquants, nous écrivons des octets nuls.

### NUL

Les 4 octets suivants dans la structure du fichier sont alloués pour un certain point d'arrêt - quatre octets zéro.

Ils ne portent aucune charge d'information. Leur travail consiste simplement à être au bon endroit.

### Taille du code exécutable Champion

Ces 4 octets contiennent des informations assez importantes - la taille du code exécutable du champion en octets.

Comme on s'en souvient, la machine virtuelle doit s'assurer que la taille du code source ne dépasse pas la limite spécifiée dans la constante `CHAMP_MAX_SIZE`. Dans le fichier fourni, `op.h` c'est `682`.

### Commentaire du champion

Les 2048 octets suivants sont occupés par le commentaire du champion. Et en substance, cette partie est complètement analogue à la partie "Nom du champion".

Vrai, sauf que maintenant la limite de longueur maximale est définie par une constante `COMMENT_LENGTH`.

### NUL

Et encore 4 octets zéro.

### Code exécutable Champion

La dernière partie du fichier est occupée par le code exécutable du champion.

Contrairement à un nom ou à un commentaire, il n'est pas rempli d'octets nuls.

## Codage des opérations

Afin de comprendre comment fonctionne l'encodage des opérations, nous avons besoin de deux tables.

### Table d'opération

Tout d'abord, nous avons besoin d'un tableau des opérations mis à jour avec les colonnes "Code des types d'argument" et "Taille `T_DIR`".

| Le code | Nom | Argument n ° 1 | Argument n ° 2 | Argument n ° 3 | Code des types d'argument | La taille `T_DIR` |
|---|---|---|---|---|---|---|
| 0x01 | `live` | `T_DIR` | - | - | Non | 4 |
| 0x02 | `ld` | `T_DIR` / `T_IND` | `T_REG` | - | il y a | 4 |
| 0x03 | `st` | `T_REG` | `T_REG` / `T_IND` | - | il y a | 4 |
| 0x04 | `add` | `T_REG` | `T_REG` | `T_REG` | il y a | 4 |
| 0x05 | `sub` | `T_REG` | `T_REG` | `T_REG` | il y a | 4 |
| 0x06 | `and` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` | il y a | 4 |
| 0x07 | `or` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` | il y a | 4 |
| 0x08 | `xor` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` | il y a | 4 |
| 0x09 | `zjmp` | `T_DIR` | - | - | Non | 2 |
| 0x0a | `ldi` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` | `T_REG` | il y a | 2 |
| 0x0b | `sti` | `T_REG` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` | il y a | 2 |
| 0x0c | `fork` | `T_DIR` | - | - | Non | 2 |
| 0x0d | `lld` | `T_DIR` / `T_IND` | `T_REG` | - | il y a | 4 |
| 0x0e | `lldi` | `T_REG` / `T_DIR` / `T_IND` | `T_REG` / `T_DIR` | `T_REG` | il y a | 2 |
| 0x0f | `lfork` | `T_DIR` | - | - | Non | 2 |
| 0x10 | `aff` | `T_REG` | - | - | il y a | 4 |

> **Pourquoi avons-nous besoin de «Size `T_DIR`» pour les opérations qui n'acceptent pas d'arguments de ce type?**
>
> Ce sont vraiment des informations inutiles à ce stade. Mais ils seront nécessaires pendant l'exécution de la machine virtuelle.
>
> Ce qui sera discuté dans la section correspondante .

Le tableau complet des transactions peut être consulté sur Google Sheets.

## Table d'arguments

Le deuxième tableau dont nous avons besoin contient des informations sur les codes de type des arguments et leurs tailles.

| Un type | Signe | Le code | La taille |
|---|---|---|---|
| `T_REG` | `r` | `01` | 1 octet |
| `T_DIR` | `%` | `10` | La taille `T_DIR` |
| `T_IND` | - | `11` | 2 octets |

Le tableau complet des arguments peut être consulté sur Google Sheets.

> **Registres et leurs tailles**
>
> Il est important de distinguer deux caractéristiques de taille concernant les registres.
>
> Le nom du registre ( `r1`, `r2`...) dans le bytecode est de 1 octet. Mais le registre lui-même contient 4 octets, comme indiqué dans la constante `REG_SIZE`.

> **À propos de la taille des arguments de type `T_DIR`**
>
> Comme vous pouvez le voir dans le tableau des opérations, la taille des arguments de type n'est `T_DIR` pas fixe et dépend de l'opération. Mais `op.h` il y a une constante de préprocesseur dans le fichier qui dit que la taille `T_DIR` est égale à la taille du registre - 4 octets:
>
> ```
> # define IND_SIZE     2
> # define REG_SIZE     4
> # define DIR_SIZE     REG_SIZE
> ```
>
> Cela soulève la question "Où est la logique ici?"
>
> Il y a une certaine logique ici, mais plutôt illusoire.
>
> Comme nous l'avons déjà dit, pour écrire un numéro de type `T_DIR` dans un registre et pour décharger ce nombre en mémoire, les tailles des registres et des arguments de type `T_DIR` doivent correspondre.
>
> Tout va bien avec cette déclaration. C'est correct. Si vous regardez les opérations qui chargent une valeur dans un registre, leur taille `T_DIR` est `4`.
>
> De plus, la taille `T_DIR` est égale `4` pour l'opération `live`.
>
> Quant aux opérations dont la taille `T_DIR` est égale `2`, alors dans ces cas l'argument de ce type ne participe qu'à la formation de l'adresse. Et pour une telle tâche, 4 octets sont redondants. Après tout, la quantité de mémoire n'est que de 4096 octets, comme indiqué dans la constante `MEM_SIZE`. Et toute adresse numérique dépassant cette limite sera tronquée modulo `MEM_SIZE`, si ce n'est déjà fait `IDX_MOD`.
>
> Dans cette situation, l'argument type `T_DIR` joue le rôle d'une adresse relative. Autrement dit, l'argument type `T_IND`. Et donc sa taille est `IND_SIZE` (2 octets).

## Algorithme de codage

Chaque opération représentée en bytecode a la structure suivante:

1. Code d'opération - 1 octet
2. Code de type d'argument (non requis pour toutes les opérations) - 1 octet
3. Arguments

> **Code des types d'argument**

> Comme mentionné ci-dessus, la structure de l'opération codée peut manquer du deuxième composant - le code des types d'argument.
>
> Sa disponibilité dépend de l'opération spécifique. S'il ne prend qu'un seul argument et que son type est uniquement défini comme `T_DIR`, alors le code avec des informations sur les types des arguments n'est pas nécessaire. Pour toutes les autres opérations, ce composant est requis.
>
> Vous pouvez vérifier si ce code est nécessaire pour une opération spécifique dans la colonne "Code de type d'argument" de la table des opérations.

Voyons comment nous encodons les instructions du code exécutable:

```
loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

## Instruction n ° 1

La première instruction à diffuser est:

```
loop:
        sti r1, %:live, %1
```

Commençons par définir les dimensions de chaque composant.

| Code d'opération | Code des types d'argument | Argument n ° 1 | Argument n ° 2 | Argument n ° 3 |
|---|---|---|---|---|
| 1 octet | 1 octet | 1 octet | 2 octets | 2 octets |

Voyons maintenant comment nous trouvons le bytecode de chaque partie de l'instruction.

### Code d'opération

Chaque code d'opération est répertorié dans le tableau d'opération. Car `sti` c'est égal `0x0b`.

### Code des types d'argument

Afin de former ce code, vous devez représenter 1 octet **dans le système binaire** . Les deux premiers bits à gauche seront occupés par le code de type d'argument # 1. Les deux suivants iront au code de type du deuxième argument. Etc. La dernière quatrième paire sera toujours égale `00`.

Les codes de chaque type sont répertoriés dans le tableau des arguments.

| Argument n ° 1 | Argument n ° 2 | Argument n ° 3 | - | Code final |
|---|---|---|---|---|
| `T_REG` | `T_DIR` | `T_DIR` | - | - |
| `01` | `10` | `10` | `00` | `0x68` |

### Argument de type `T_REG`

Dans ce cas, le numéro de registre est traduit en code hexadécimal. Pour le registre, `r1` c'est `0x01`.

### Argument d'étiquette de type `T_DIR`

Comme nous le savons déjà, l'étiquette doit se transformer en un nombre contenant l'adresse relative en octets.

Étant donné que l'étiquette `live` pointe vers l'instruction suivante et que nous connaissons déjà la taille en octets de l'instruction actuelle, nous pouvons facilement calculer la distance requise - 7 octets.

L'adresse numérique résultante devra tenir dans 2 octets - `0x0007`.

**Argument numéro de type `T_DIR`**

Dans ce cas, tout est encore plus simple. Il vous suffit d'écrire le nombre obtenu dans le système décimal sous la forme d'un bytecode hexadécimal - `0x0001`.

Le bytecode de l'instruction qui en résulte est `0b 68 01 0007 0001`.

## Instruction n ° 2

Pour l'instruction suivante, tout est pareil:

```
live:
        live %0
```

Le seul changement significatif est que cette opération ne nécessite pas de code de type d'argument:

| Code d'opération | Argument n ° 1 |
|---|---|
| 1 octet | 4 octets |

Le bytecode final de la deuxième instruction est `01 00000000`.

## Instruction n ° 3

Troisième instruction:

```
        ld %0, r2
```

| Code d'opération | Code des types d'argument | Argument n ° 1 | Argument n ° 2 |
|---|---|---|---|
| 1 octet | 1 octet | 4 octets | 1 octet |

Il n'y a pas non plus de surprises ici - `02 90 00000000 02`.

## Instruction n ° 4

```
        zjmp %:loop
```

L'opcode `zjmp` est `0x09`.

Le code de type d'argument n'est pas requis pour cette opération.

| Code d'opération | Argument n ° 1 |
|---|---|
| 1 octet | 2 octets |

L'étiquette `loop` pointe 19 octets en arrière. Mais comment représenter un nombre en système hexadécimal `-19`?

Pour ce faire, vous devez écrire le numéro `19` **dans le système binaire** en code direct:

```
0000 0000 0001 0011
```

A partir d'un nombre `19` écrit en code direct, on peut obtenir un nombre `-19` en complément à deux.

> C'est dans le code du complément qu'il est d'usage de représenter des entiers négatifs dans les ordinateurs.

Afin d'obtenir un code numérique supplémentaire `-19`, nous devons effectuer les étapes suivantes:

1. Inverser tous les chiffres

Autrement dit, remplacez un par zéro et vice versa:

```
1111 1111 1110 1100
```

2. Ajouter un à un nombre

```
1111 1111 1110 1101
```

Terminé. Nous avons donc le numéro `-19` dans le code complémentaire.

Convertissons-le du système binaire en système hexadécimal - `0xffed`.

Le bytecode final de l'instruction n ° 5 est `09 ffed`.

## Résultat

L'ensemble du code exécutable du champion en question ressemblera à ceci:

```
0b68 0100 0700 0101 0000 0000 0290 0000
0000 0209 ffed
```


▸ **Des pages** neuf

1. introduction
2. Fichiers fournis
3. Assembleur
4. Analyse lexicale
5. Assemblage en Bytecode
6. Démontage
7. Machine virtuelle
8. Visualisation