Assembleur

Jump to bottom

VBrazhnik a modifié cette page le 3 janvier 2019 · 2 révisions

Syntaxe de l'assembleur

Le langage d'assemblage obéit à la règle «Une ligne, une instruction».

> Une instruction ou une instruction est la plus petite partie autonome d'un langage de programmation; commande ou ensemble de commandes. Un programme est généralement une séquence d'instructions.

Source: Opérateur (programmation) - Wikipédia

Les lignes vides, les commentaires, ainsi que les tabulations ou espaces supplémentaires sont ignorés.

Commentaire

Il `op.h` y a une constante dans l'en-tête `COMMENT_CHAR`. Il détermine quel caractère marque le début d'un commentaire.

Dans le fichier fourni, il s'agit d'octotorp - `#`.

Autrement dit, tout ce qui se trouve entre le caractère `#` et la fin de la ligne sera considéré comme un commentaire .

Le commentaire peut être situé n'importe où dans le fichier.

Exemple 1:

```
# UNIT Factory
# is a programming school
```

Exemple n ° 2:

```
ld %0, r2    # And it is located in UNIT City
```

Commentaire alternatif

Dans l'archive fournie le `vm_champs.tar` long du chemin, `champs/examples` vous pouvez trouver un fichier `bee_gees.s` avec le code du champion, que le programme d'origine `asm` traduit en bytecode sans erreur.

Il existe deux types de commentaires dans le code du champion:

- standard, dont il a été question ci-dessus;
- alternative, sur laquelle il n'y a aucune information dans le sujet.

Cette forme alternative diffère de la norme et n'est décrite dans le sujet que par le symbole du début du commentaire. Au lieu d'octotorpa (`#`) apparaît ici `;`.

Un exemple d'utilisation d'un commentaire de ce type:

```
sti r1, %:live, %1    ; UNIT City is placed in Kyiv, Ukraine
```

Comment vivre avec?

Ce type de commentaire n'est pas décrit dans le sujet, mais est pris en charge par le traducteur d'origine. Par conséquent, nous n'avons probablement pas à le traiter.

Mais ajoutons son support à notre projet. Pour ce faire, la `op.h` ligne suivante sera ajoutée au fichier d'en-tête :

```
# define ALT_COMMENT_CHAR    ';'
```

> Ce sera le deuxième (le premier est le cast en Norm) et le dernier changement que nous apporterons au fichier `op.h`.

Nom du champion

Le fichier de code du champion doit définir son nom. Pour cela, il existe une commande en assembleur, dont le nom est défini dans une constante `NAME_CMD_STRING`. Dans le fichier fourni, `op.h` c'est `.name`.

Autrement dit, la commande `.name` doit être suivie d'une ligne avec le nom de notre champion:

```
.name    "Batman"
```

La longueur de la chaîne ne doit pas dépasser le nombre spécifié dans la constante `PROG_NAME_LENGTH`. Dans le fichier fourni, c'est `128`.

À propos, une chaîne vide peut également être utilisée comme nom de champion:

```
.name    ""
```

Mais l'absence totale de ligne est déjà une erreur:

```
.name
```

Commentaire du champion

En outre, un `.s` commentaire de champion doit être défini dans le fichier avec l'extension .

La commande pour vous aider à faire cela est contenue dans la constante de `COMMENT_CMD_STRING` fichier `op.h`. Dans le fichier fourni, c'est `.comment`.

La longueur de la ligne de commentaire est limitée à une constante `COMMENT_LENGTH`. Dans le fichier fourni, `op.h` sa valeur est `2048`.

À la base, la commande est `.comment` très similaire `.name` et se comporte de la même manière dans les cas avec une chaîne vide et dans les cas sans chaîne du tout.

Autres commandes

Dans certains des fichiers avec l'extension `.s` qui nous ont été fournis à titre d'exemple, il y avait une commande telle que `.extend`.

Cette commande, ainsi que toute autre commande autre que `.name` et `.comment`, n'est pas décrite dans le sujet et est définie comme erronée par le traducteur d'origine.

Nous traiterons ces commandes de la même manière.

Code exécutable

Le code exécutable du champion se compose d'instructions.

Pour le langage assembleur, la règle est "Une ligne - une instruction". Et le caractère de fin d'une instruction dans cette langue est un saut de ligne. Autrement dit, au lieu du symbole familier dans le langage C, un symbole `;` apparaît ici `\n`.

Sur la base de cette règle, nous devons nous rappeler que **même après la dernière instruction, un saut de ligne doit suivre** . Sinon, il `asm` affichera un message d'erreur.

Chaque instruction se compose de plusieurs éléments:

Étiquette

L'étiquette se compose de caractères qui ont été définis dans une constante `LABEL_CHARS`. Dans l'exemple de fichier, c'est `abcdefghijklmnopqrstuvwxyz_0123456789`.

Autrement dit, l'étiquette ne peut pas contenir de caractères non spécifiés dans `LABEL_CHARS`.

Et l'étiquette elle-même doit être suivie d'un symbole défini dans une constante `LABEL_CHAR`. Dans le fichier d'exemple, il s'agit d'un symbole `:`.

Pourquoi des étiquettes sont-elles nécessaires?

L'étiquette indique l'opération qui la suit immédiatement. C'est pour une opération, et non pour leur bloc.

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

loop:
        sti r1, %:live, %1    # <-- На эту операцию указывает метка loop
live:
        live %0               # <-- На эту операцию указывает метка live
        ld %0, r2             # <-- А на эту операцию никакая метка не указывает
        zjmp %:loop
```

Le but des étiquettes est de simplifier notre vie, ou plutôt le processus d'écriture de code.

Pour bien comprendre leur rôle, imaginons un monde sans étiquettes.

Comme nous le savons, le code du champion écrit en langage d'assemblage, après l'exécution du programme de traduction, se transformera en un ensemble d'octets représentés dans le système hexadécimal. Et c'est exactement le bytecode que la machine virtuelle exécutera.

Disons que nous devons organiser une boucle dans laquelle l'opération serait répétée encore et encore `live`. Pour ce faire, nous avons une opération `zjmp` qui peut nous renvoyer le N-ième nombre d'octets en avant ou en arrière.

Dans ce cas, nous devons revenir à l'opération après chaque itération de la boucle `live`. Combien d'octets sont-ils en retour? Pour le savoir, vous devez savoir combien d'octets dans le bytecode le code d'opération et son argument prendront.

Comme nous l'apprendrons plus tard, l'opcode `live` est de 1 octet, et son seul argument nécessite 4 octets.

Nous devons donc reculer de 5 octets:

```
live %1
zjmp %-5
```

Ce n'est pas si difficile, mais de tels calculs prennent du temps. Et il serait beaucoup plus facile d'écrire simplement "passer à l'opération `live`". C'est à cela que servent les étiquettes.

Nous créons simplement une étiquette pour l'opération dont nous avons besoin `live` et la passons à l'opération `zjmp`:

```
loop:    live %1
         zjmp %:loop
```

Du point de vue de la traduction du langage d'assemblage en bytecode, les deux exemples sont absolument identiques. Pendant le fonctionnement, le programme `asm` calculera le nombre d'octets en arrière des points d'étiquette `loop` et le remplacera par un nombre `-5`.

Donc, pour le résultat final, peu importe ce qui a été utilisé. Cependant, il est beaucoup plus pratique d'écrire du code à l'aide d'étiquettes.

Formulaires d'enregistrement

Il existe plusieurs approches pour écrire une étiquette:

```
marker:
        live %0
```

```
marker:


        live %0
```

```
marker: live %0
```

Tous les exemples ci-dessus pour un programme de traduction signifient la même chose.

Par conséquent, vous pouvez choisir l'une des options présentées.

Plusieurs étiquettes pour une opération

La forme d'enregistrement suivante est également possible:

```
marker:

label:
        live %0
```

Cela signifie que l'étiquette `marker` et l'étiquette `label` pointent toutes deux vers la même opération.

Sans chirurgie

Ou une situation peut survenir lorsque l'étiquette n'a pas d'opération vers laquelle elle pourrait pointer:

```
marker:
# Конец файла
```

Dans ce cas, l'étiquette pointe vers l'emplacement immédiatement après le code exécutable du champion.

L'essentiel est qu'à la fin de la ligne sur laquelle il se trouve `\n`. Sinon, le traducteur signalera une erreur.

Opérations et leurs arguments

Le langage d'assemblage a un ensemble spécifique de 16 opérations. Chacun d'eux prend un à trois arguments.

Les informations sur le nom de l'opération, son code et les arguments qu'elle prend sont données dans le fichier fourni par la tâche `op.c`.

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
une	`live`	`T_DIR`	-	-
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	-
3	`st`	`T_REG`	`T_REG` / `T_IND`	-
4	`add`	`T_REG`	`T_REG`	`T_REG`
5	`sub`	`T_REG`	`T_REG`	`T_REG`
6	`and`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`
sept	`or`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`
8	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`
neuf	`zjmp`	`T_DIR`	-	-
Dix	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`
Onze	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`
12	`fork`	`T_DIR`	-	-
13	`lld`	`T_DIR` / `T_IND`	`T_REG`	-
14	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`
15	`lfork`	`T_DIR`	-	-
16	`aff`	`T_REG`	-	-

Comprendre quel est le rôle de chaque opération et comment elle interprète différents types d'arguments est la tâche la plus importante pour comprendre les bases d'un projet Corewar.

Opérations et leurs arguments

Arguments

Chaque argument correspond à l'un des trois types suivants:

1. Registre - Registre - `T_REG`

Un registre est une variable dans laquelle nous pouvons stocker toutes les données. La taille de cette variable en octets est indiquée dans une constante `REG_SIZE`, qui est `op.h` initialisée avec une valeur dans le fichier échantillon `4`.

> Un octet en informatique est de huit bits. En russe, un octet est généralement appelé un octet.

Source: Octet (informatique) - Wikipédia

Le nombre de registres est limité par le nombre spécifié dans la constante `REG_NUMBER`. Dans l'exemple - `16`.

Qui est disponible pour nous , il enregistre `r1`, `r2`, `r3`... `r16`.

> **Enregistrer les valeurs**
>
> Lors du démarrage de la machine virtuelle, tous les registres, sauf `r1`, seront initialisés à zéro.
>
> Le `r1` numéro du joueur champion sera inscrit. Seulement avec un signe moins.
>
> Ce numéro est unique dans le jeu et une opération est nécessaire `live` pour informer qu'un joueur particulier est vivant.
>
> Autrement dit, le signe d'insertion qui sera placé au début du code du joueur sous le numéro `2` recevra la valeur `r1` égale à `-2`.
>
> Si l'opération `live` est effectuée avec un argument `-2`, la machine virtuelle considérera que ce lecteur est vivant:
>
> ```
> live %-2
> ```

2. Direct - Direct - `T_DIR`

L'argument direct comporte deux parties: le caractère spécifié dans la constante `DIRECT_CHAR` (`%`) + un nombre ou une étiquette qui représente la **valeur directe** .

Si nous parlons d'une étiquette, alors un symbole de la variable `LABEL_CHAR` (`:`) doit également être spécifié avant son nom :

```
sti r1, %:marker, %1
```

> **Qu'est-ce que la signification directe et indirecte?**
>
> Pour comprendre la différence entre le sens direct et indirect, il vaut la peine de regarder un exemple très simple.

> Imaginons que nous ayons un certain nombre `5`. Dans son sens direct, il se représente. Autrement dit, un nombre `5` est un nombre `5`.
>
> Mais dans un sens indirect, ce n'est plus un nombre, mais une adresse relative qui pointe 5 octets en avant.

> **Étiquette au sens direct et indirect**
>
> Si tout est clair avec des nombres au sens direct et indirect, qu'en est-il des étiquettes? Quelle est la différence?
>
> C'est assez simple. Comme nous le savons, ces opérations seront effectuées par une machine virtuelle. Et c'est pour elle qu'il est important de savoir quel argument a été reçu. Direct ou indirect?
>
> Mais les étiquettes n'atteindront tout simplement pas la machine virtuelle. Au stade de la traduction en bytecode, ils seront tous remplacés par leurs équivalents numériques.
>
> Par conséquent, les étiquettes sont les mêmes numéros. Uniquement écrit sous une forme différente
>
> Le processus de remplacement des étiquettes par des nombres est décrit dans le chapitre "Pourquoi les étiquettes sont-elles nécessaires?"

3. Indirect - Indirect - `T_IND`

Un argument de ce type peut être un nombre ou une étiquette, qui représentent une **valeur indirecte** .

Si l'argument de type `T_IND` est un nombre, aucun caractère supplémentaire n'est nécessaire:

```
ld    5, r7
```

Si un tel argument est une étiquette, alors le symbole de la variable `LABEL_CHAR` (`:`) doit être spécifié avant son nom :

```
ld    :label, r7
```

Caractère séparateur

Afin de séparer un argument d'un autre en une seule opération, l'assembleur utilise un caractère de délimitation spécial. Il est défini par une constante de préprocesseur `SEPARATOR_CHAR` et dans le fichier d'exemple `op.h` c'est un symbole `,`:

```
ld    21, r7
```

Opérations

Opération `live`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
une	`live`	`T_DIR`	-	-

La description

L'opération a `live` deux fonctions:

1. Il compte que le **chariot** effectuant l'opération `live` est vivant.
2. Si le nombre spécifié comme argument de l'opération `live` coïncide avec le numéro du joueur, alors il compte que ce joueur est vivant. Par exemple, si la valeur de l'argument est égale `-2`, alors le joueur avec le numéro est `2` vivant.

> **Qu'est-ce qu'une calèche?**
>
> Une explication détaillée de ce terme sera donnée dans la section "Machine virtuelle" , car le chariot se réfère spécifiquement à cette partie du projet "Corewar".
>
> Mais comme une connaissance de base de ce terme est une condition préalable pour comprendre le fonctionnement des opérations, nous examinerons brièvement ce que c'est.
>
> Le chariot est un processus qui exécute l'opération sur laquelle il se tient.
>
> Disons que nous exécutons une machine virtuelle avec trois joueurs champions qui doivent se battre pour la victoire.
>
> Cela signifie que les codes exécutables des champions seront placés dans la mémoire de la machine virtuelle. Et un chariot sera placé au début de chacune des sections de mémoire.
>
> 3 champions. 3 zones de mémoire avec des codes exécutables situés dessus. 3 voitures.
>
> Chaque chariot contient plusieurs éléments importants:
>
> - `PC` (Compteur de programme)
>
> Une variable qui contient la position du curseur.
>
> - Registres
>
> Les mêmes registres, dont le nombre est déterminé par la constante `REG_NUMBER`.
>
> - Drapeau `carry`
>
> Variable spéciale qui affecte le fonctionnement de la fonction `zjmp` et peut prendre l'une des deux valeurs suivantes: `1` ou `0` (`true` ou `false`).
>
> - Numéro de cycle dans lequel ce chariot a effectué une opération pour la dernière fois `live`
>
> Ces informations sont nécessaires pour déterminer si le chariot est actif.
>
> En fait, le curseur contient beaucoup plus d'éléments, mais ils seront abordés plus tard.

Opération `ld`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	-

La description

La tâche de cette opération est de charger une valeur dans un registre. Mais son comportement diffère selon le type du premier argument:

- Argument n ° 1 - `T_DIR`

Si le type du premier argument est celui-ci `T_DIR`, alors le nombre passé en argument sera traité tel quel.

Objectifs de l'opération:

1. Écrivez le nombre résultant dans le registre qui a été passé comme deuxième argument.
2. Si un nombre a été écrit dans le registre `0`, définissez la valeur `carry` sur `1`. Si une valeur différente de zéro a été écrite, définissez-la `carry` sur `0`.

- Argument n ° 1 - `T_IND`

Si le type du premier argument est `T_IND`, alors dans ce cas, le nombre est l'adresse.

Si un argument de ce type est reçu, il est tronqué modulo - `<ПЕРВЫЙ_АРГУМЕНТ> % IDX_MOD`.

> **Qu'est-ce que c'est `IDX_MOD` ?**
>
> `IDX_MOD` c'est une autre constante du fichier `op.h`. Sa valeur est déterminée à l'aide d'une expression `(MEM_SIZE / 8)` où elle `MEM_SIZE` détermine la quantité de mémoire en octets. Dans une machine virtuelle, les `MEM_SIZE` champions se battront dans un morceau de mémoire de la taille de et.
>
> Alors, à quoi sert une constante `IDX_MOD` ? Il est nécessaire pour que le chariot ne puisse pas sauter de trop longues distances en mémoire. Dans le fichier d'exemple, une constante a `MEM_SIZE` été initialisée avec une valeur `(4 * 1024)`. Par conséquent, la valeur `IDX_MOD` dans ce cas correspond `512`.
>
> Il s'avère que le chariot ne peut pas se déplacer de plus de 512 octets en un seul saut.

Une fois que l'argument de type a `T_IND` été tronqué modulo, la valeur résultante est utilisée comme adresse relative - combien d'octets en avant ou en arrière par rapport à la position actuelle du chariot est la position souhaitée.

Tâche d'opération `ld` :

1. Déterminer l'adresse - position actuelle + `<ПЕРВЫЙ_АРГУМЕНТ> % IDX_MOD`.
2. Il est nécessaire de lire **4 octets à** partir de l'adresse reçue .
3. Écrivez le numéro de lecture dans le registre qui a été passé comme deuxième paramètre.
4. Si un nombre a été écrit dans le registre `0`, nous définissons la valeur `carry` sur `1`. Si une valeur différente de zéro a été écrite, définissez-la `carry` sur `0`.

> **Pourquoi lisons-nous exactement 4 octets?**
>
> Comme nous le savons, la taille de chaque registre est de 4 octets. Ou plutôt, le nombre d'octets qui est défini dans le fichier par une `op.h` constante `REG_SIZE`.
>
> Le même fichier définit la taille d'un argument de type `T_DIR`:
>
> ```
> # define REG_SIZE    4
> # define DIR_SIZE    REG_SIZE
> ```
>
> Et c'est aussi 4 octets.
>
> On va à l'adresse donnée par l'argument type `T_IND` pour lire la valeur. Considérez le nombre «tel quel». Autrement dit, obtenez un nombre comme `T_DIR`. Et nous devons écrire ce nombre même dans le registre. Pour que l'écriture soit réussie et que le nombre lu tienne dans le registre, les tailles de ce nombre et les tailles du registre doivent être compatibles.
>
> En outre, lors de l'analyse des opérations suivantes, il deviendra connu que nous pouvons non seulement lire la valeur et l'écrire dans le registre, mais également effectuer l'action inverse - décharger la valeur du registre à l'adresse.

> Par conséquent, les tailles du nombre et du registre doivent être compatibles dans les deux sens. Dans ce cas, la seule solution possible est de rendre le nombre d'octets que nous lisons (ou écrivons) égal au nombre d'octets que nous pouvons stocker dans le registre.
>
> En général, nous lisons autant d'octets que le registre peut contenir.

Opération `st`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
3	`st`	`T_REG`	`T_REG` / `T_IND`	-

La description

Cette opération écrit une valeur du registre qui a été transmise comme premier paramètre. Mais où cette opération écrit cela dépend du type du deuxième argument:

- Argument n ° 2 - `T_REG`

Si le deuxième argument correspond au type `T_REG`, la valeur est écrite dans le registre.

Par exemple, dans ce cas, la valeur du registre numéro 7 est écrite dans le registre numéro 11:

```
st    r7, r11
```

- Argument n ° 2 - `T_IND`

Comme nous nous en souvenons, les arguments de type `T_IND` concernent des adresses relatives. Par conséquent, dans ce cas, la procédure pour l'opération est `st` la suivante:

1. Tronquez la valeur du deuxième argument modulo `IDX_MOD`.
2. Déterminer l'adresse - position actuelle + `<ВТОРОЙ_АРГУМЕНТ> % IDX_MOD`
3. Ecrivez la valeur du registre qui a été passée comme premier argument à la mémoire à l'adresse reçue.

Opération `add`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
4	`add`	`T_REG`	`T_REG`	`T_REG`

La description

Heureusement, pour cette opération, tous les arguments sont du même type. Par conséquent, tout est simple avec elle.

Tâche d'opération `add`:

1. Somme la valeur du registre qui a été passée comme premier argument à la valeur du registre qui a été transmise comme deuxième argument.
2. Écrivez le résultat obtenu dans le registre qui a été passé comme troisième argument.
3. Si le montant reçu que nous avons écrit dans le troisième argument était égal à zéro, nous le définissons `carry` sur `1`. Et si le montant n'était pas nul - c `0`.

Opération `sub`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
5	`sub`	`T_REG`	`T_REG`	`T_REG`

La description

Il n'y a pas non plus d'ambiguïté sur les arguments de cette opération.

Ses tâches:

1. De la valeur de registre passée comme premier argument, soustrayez la valeur de registre qui a été transmise comme deuxième argument.
2. Le résultat obtenu est écrit dans le registre, qui a été passé comme troisième argument.
3. Si le résultat enregistré était égal à zéro, `carry` rendez la valeur égale `1`. Si le résultat n'est pas égal à zéro, rendez-le égal `0`.

Opération `and`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
6	`and`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`

La description

`and` effectue une opération AND au niveau du bit sur les deux premiers arguments et écrit le résultat dans le registre passé comme troisième argument.

Si le résultat enregistré était égal à zéro, la valeur `carry` doit être définie comme égale `1`. Si le résultat n'était pas nul, alors - égal `0`.

Étant donné que le premier et le deuxième arguments peuvent être de trois types, nous allons voir comment obtenir la valeur de chacun d'eux:

- Argument n ° 1 / Argument n ° 2 - `T_REG`

Dans ce cas, la valeur est tirée du registre passé en argument.

- Argument n ° 1 / Argument n ° 2 - `T_DIR`

Dans ce cas, la valeur numérique passée en argument est utilisée.

- Argument n ° 1 / Argument n ° 2 - `T_IND`

Si le type est l'argument `T_IND`, alors il est nécessaire de définir l'adresse à partir de laquelle 4 octets seront lus.

L'adresse est déterminée comme suit - position actuelle + `<АРГУМЕНТ> % IDX_MOD`.

Un nombre de 4 octets lu à cette adresse sera la valeur requise.

Opération `or`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
sept	`or`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`

La description

En substance, cette opération est tout à fait analogue à l'opération `and`. Seulement dans ce cas, "ET au niveau du bit" est remplacé par **"OU au niveau du bit"** .

Opération `xor`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
8	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`

La description

En substance, cette opération est tout à fait analogue à l'opération `and`. Seulement dans ce cas, "ET au niveau du bit" est remplacé par **"OU exclusif au niveau du bit"** .

> Comment fonctionne le OU exclusif bit à bit (XOR)?
>
> | UNE | B | A ^ B |
> |---|---|---|
> | 0 | 0 | 0 |
> | 0 | une | une |
> | une | 0 | une |
> | une | une | 0 |

Opération `zjmp`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
neuf	`zjmp`	`T_DIR`	-	-

La description

C'est la même fonction qui est affectée par la valeur de l'indicateur `carry`.

S'il est égal `1`, la fonction met à jour la valeur `PC` à l'adresse - la position actuelle + `<ПЕРВЫЙ_АРГУМЕНТ> % IDX_MOD`.

Autrement dit, il `zjmp` définit où le chariot doit se déplacer pour effectuer l'opération suivante. Cela nous permet de sauter à la position souhaitée en mémoire et de ne pas tout faire dans l'ordre.

Si la valeur `carry` est zéro, aucun déplacement n'est effectué.

Opération `ldi`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
Dix	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`

La description

Cette opération écrit une valeur dans le registre qui lui a été transmise en tant que troisième paramètre. La valeur qu'il écrit est de 4 octets. Il lit ces 4 octets à l'adresse, qui est formée selon le principe suivant: position actuelle + `(<ЗНАЧЕНИЕ_ПЕРВОГО_АРГУМЕНТА> + <ЗНАЧЕНИЕ_ВТОРОГО_АРГУМЕНТА>) % IDX_MOD`.

Puisqu'une opération peut prendre différents types du premier et du deuxième arguments, considérez comment obtenir une valeur pour chaque type:

- Argument n ° 1 / Argument n ° 2 - `T_REG`

La valeur est contenue dans le registre qui a été passé en paramètre.

- Argument n ° 1 / Argument n ° 2 - `T_DIR`

Dans ce cas, notre argument contient déjà sa valeur.

- Argument n ° 1 - `T_IND`

Pour obtenir la valeur de cet argument, vous devez lire 4 octets à l'adresse - position actuelle + `<ПЕРВЫЙ_АРГУМЕНТ> % IDX_MOD`.

Opération `sti`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
Onze	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`

La description

Cette opération écrit la valeur du registre passé comme premier paramètre à l'adresse - position actuelle + `(<ЗНАЧЕНИЕ_ВТОРОГО_АРГУМЕНТА> + <ЗНАЧЕНИЕ_ТРЕТЕГО_АРГУМЕНТА>) % IDX_MOD`.

La façon d'obtenir la valeur de chaque type d'argument est décrite ci-dessus.

Opération `fork`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
12	`fork`	`T_DIR`	-	-

La description

L'opération `fork` fait une copie du chariot. Et place cette copie à `<ПЕРВЫЙ_АРГУМЕНТ> % IDX_MOD`.

> **Quelles données sont copiées?**
>
> - Valeurs de tous les registres
> - Valeur `carry`
> - Numéro de cycle dans lequel l'opération a été effectuée pour la dernière fois `live`
> - Et autre chose, mais plus à ce sujet plus tard.

Opération `lld`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
13	`lld`	`T_DIR` / `T_IND`	`T_REG`	-

La description

Cette opération se comporte à peu près de la même manière que l'opération `ld`. Autrement dit, il écrit la valeur obtenue à partir du premier argument dans le registre passé comme deuxième argument.

La seule différence entre les deux est la troncature modulo.

Si le premier argument a un type `T_IND`, alors dans cette opération, nous lirons 4 octets de la valeur à l'adresse - la position actuelle + `<ПЕРВЫЙ_АРГУМЕНТ>`. **Ne pas appliquer la troncature modulo** .

> **Problèmes de machine virtuelle d'origine**
>
> La machine virtuelle d'origine `corewar` ne fonctionne malheureusement pas correctement. Et lit 2 octets, pas 4. Peut-être qu'un bogue similaire est expliqué par les mêmes lignes que les problèmes dans les fichiers fournis:
>
> > ... nous avons peut-être confondu une bouteille d'eau avec une bouteille de vodka.

`T_DIR` Rien ne change pour l'argument type .

Opération `lldi`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
14	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`

La description

En substance, cette opération est similaire à l'opération `ldi`.

Elle écrit la valeur dans le registre qui lui a été transmise comme troisième paramètre. La valeur que cette opération écrit est de 4 octets lus.

Ils sont lus à l'adresse, qui est formée selon le principe suivant: position actuelle + `(<ЗНАЧЕНИЕ_ПЕРВОГО_АРГУМЕНТА> + <ЗНАЧЕНИЕ_ВТОРОГО_АРГУМЕНТА>)`.

Contrairement à l'opération, `ldi` dans ce cas, lors de la formation de l'adresse, vous n'avez pas besoin de tronquer modulo `IDX_MOD`.

> **Pour les arguments de type, `T_IND` tout reste le même.**
>
> Si nous recevons un argument de type comme premier ou deuxième argument `T_IND`, alors nous lisons toujours 4 octets de la valeur à l'adresse - la position actuelle + `<АРГУМЕНТ> % IDX_MOD`. Ici, la troncature modulo est préservée.

Opération `lfork`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
15	`lfork`	`T_DIR`	-	-

La description

En substance, cette opération est similaire à l'opération `fork`.

Sauf le fait que le nouveau curseur dans ce cas est créé à l'adresse - position actuelle + `<ПЕРВЫЙ_АРГУМЕНТ>`. Vous `lfork` n'avez pas besoin d'effectuer une troncature modulo dans l' opération .

Opération `aff`

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
16	`aff`	`T_REG`	-	-

La description

Cette opération prend une valeur du registre qui a été passée comme seul argument. Le conduit à taper `char`. Et l'affiche sous forme de caractère ASCII.

> Dans le texte de l'affectation, le fonctionnement de l'opération aff est décrit par les lignes suivantes:
>
> > **aff** : l'opcode est `10` au format hexadécimal. Il y a l'octet de codage d'un argument, même si c'est un peu idiot car il n'y a qu'un seul argument qui est un registre, qui est un registre, et son contenu est interprété par la valeur ASCII du caractère à afficher sur la sortie standard. Le code est modulo `256`.
>
> Mais en réalité, `256` il n'est pas nécessaire d'effectuer une troncature modulo supplémentaire .
>
> Après tout, le résultat de ces deux calculs sera identique:
>
> ```
> (char)(value % 256)
> ```
>
> ```
> (char)(value)
> ```

> **Mode d'affichage de sortie `aff` dans l'original `corewar`**
>
> Dans la machine virtuelle d'origine `corewar`, le mode d'affichage de la sortie d'opération est `aff` désactivé par défaut . Pour l'activer, vous devez utiliser l'indicateur `-a`.

Mise à jour du tableau des opérations

Le code	Nom	Argument n ° 1	Argument n ° 2	Argument n ° 3	Changements `carry`	La description
une	`live`	`T_DIR`	-	-	Non	vivant
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	-	Oui	charge
3	`st`	`T_REG`	`T_REG` / `T_IND`	-	Non	boutique
4	`add`	`T_REG`	`T_REG`	`T_REG`	Oui	une addition
5	`sub`	`T_REG`	`T_REG`	`T_REG`	Oui	soustraction
6	`and`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Oui	ET au niveau du bit (`&`)

Le code	Nom	Argument n ° 1	Argument n ° 2	Argument n ° 3	Changements `carry`	La description
sept	`or`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Oui	OU au niveau du bit (`|`)
8	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Oui	XOR au niveau du bit (`^`)
neuf	`zjmp`	`T_DIR`	-	-	Non	sauter si non nul
Dix	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	Non	indice de charge
Onze	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	Non	index du magasin
12	`fork`	`T_DIR`	-	-	Non	fourchette
13	`lld`	`T_DIR` / `T_IND`	`T_REG`	-	Oui	longue charge
14	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	Oui	indice de charge long
15	`lfork`	`T_DIR`	-	-	Non	longue fourche
16	`aff`	`T_REG`	-	-	Non	aff

Boucles avant exécution

Mais ce n'est pas tout ce qu'il y a à savoir sur les opérations.

Il y a un autre paramètre important - les cycles avant l'exécution.

C'est le nombre de cycles que le chariot doit attendre avant de démarrer l'opération.

Par exemple, s'il se trouve en fonctionnement pour les `fork` 800 prochains cycles, il attendra. Et pour l'opération, l' `ld` attente n'est que de 5 cycles.

Ce paramètre a été introduit pour créer des mécanismes de jeu dans lesquels les fonctions les plus efficaces et utiles ont la plus grande valeur.

Le code	Nom	Boucles avant exécution
une	`live`	Dix
2	`ld`	5
3	`st`	5
4	`add`	Dix
5	`sub`	Dix
6	`and`	6
sept	`or`	6

Le code	Nom	Boucles avant exécution
8	`xor`	6
neuf	`zjmp`	vingt
Dix	`ldi`	25
Onze	`sti`	25
12	`fork`	800
13	`lld`	Dix
14	`lldi`	cinquante
15	`lfork`	1000
16	`aff`	2

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

▸ **Des pages** neuf

Cloner ce wiki localement