Documentation pour l'utilisateur

Le programme ci-joint est un simulateur d'une machine fictive composée d'une mémoire de 64Ko et d'un microprocesseur contenant 32 registres (de r0 à r31) sur 16 bits, ainsi que d'un registre d'état avec les bits Z, N et C. Le programme est écrit en langage C.

Sommaire

1. Compiler et exécuter le programme

Le dossier contient les fichiers :

- main.c
- conversion.c , conversion.h
- etiquette.c , etiquette.h
- lire_instructions.c , lire_instructions.h
- Makefile
- simulateur

Tout d'abord, assurez-vous que ces fichiers sont bien regroupés dans un même répertoire et que l'invite de commandes affiche bien le chemin de ce répertoire courant.

Afin de compiler, deux options s'offrent à vous.
Vous pouvez soit compiler "à la main" :

```
gcc -Wall main.c lire_instructions.c etiquette.c conversion.c -o simulateur -lm
```

Soit utiliser un Makefile (assurez-vous qu'il se trouve dans le même répertoire que les fichiers à compiler) en faisant :

```
make
```

Le simulateur prend en entrée un fichier texte contenant un programme assembleur. Le programme sera lancé de la façon suivante (avec un fichier en assembleur appelé par exemple *moncode.txt*, sachant que l'exécutable s'appelle *simulateur*) :

```
./simulateur moncode.txt
```

Le programme va alors traduire le fichier moncode.txt en langage machine puis, s'il n'y a pas d'erreurs, générer le fichier de sortie et l'exécuter. Si on rencontre une erreur à la lecture des instructions en assembleur, le fichier d'instructions en langage machine n'est pas généré et on signale l'erreur en affichant le numéro de la ligne erronée ainsi que l'erreur détectée. Le programme s'arrête dès la première erreur trouvée.

Lorsque le programme est lancé, l'utilisateur n'a plus besoin de faire quoi que ce soit, excepté rentrer des données demandées par le programme.

2. Manuel de la syntaxe assembleur

a. Syntaxe générale d'un fichier d'entrée en assembleur

Le fichier texte du programme en langage assembleur doit être composé d'une instruction par ligne. Chaque ligne doit être de la forme suivante :

```
etiquette:   mot_cle_operation    operandes
```

c'est-à dire que chaque ligne est composée d'une étiquette optionnelle, de 0, 1 ou plusieurs espaces ou tabulations, du mot-clé de l'instruction (en minuscule), suivi d'une ou plusieurs espaces, et des opérandes séparées par des virgules et éventuellement des espaces.
Par exemple:

Avec une étiquette : `loop: add r3, r2, #1`
Sans étiquette : `add r3, r2, #1`

S'il y a une étiquette, son nom doit être composé de lettres non accentuées et de chiffres et sera **immédiatement** suivi d'un deux-points lors de sa définition.

Chaque instruction comporte les champs code d'opération, registre de destination (rd), registre source 1 (rn), et source 2 (S); certains sont éventuellement non significatifs selon l'instruction exécutée.

- Le registre de destination s'écrit `rd` où d est un entier compris entre 0 et 31.
- Le registre de source 1 s'écrit `rn` où n est un entier compris entre 0 et 31.
- La source 2 est soit un registre (même syntaxe que les deux précédents), soit une valeur immédiate entière en base 10 (on l'écrira alors `#valeur`, par exemple `#10`), soit une valeur immédiate entière en hexadécimal (on l'écrira alors `#hvaleur`, par exemple `#h7E`).

On notera que le registre r0 vaut toujours zéro même si vous essayez de le modifier.

b. Instructions arithmétiques et logiques

`add rd, rn, S` : met dans rd la valeur rn + S
`sub rd, rn, S` : met dans rd la valeur rn - S
`mul rd, rn, S` : met dans rd la valeur rn x S (l'opération se fait sur les 8 bits de poids faibles de rn et S)
`div rd, rn, S` : met dans rd la valeur rn / S (division entière)
`or rd, rn, S` : met dans rd la valeur rn ou S ("ou" logique)
`xor rd, rn, S` : met dans rd la valeur rn ou exclusif S ("ou exclusif" logique)
`and rd, rn, S` : met dans rd la valeur rn et S ("et" logique)
`shl rd, rn, S` : met dans rd la valeur rn décalée S fois à gauche (ou à droite si S est négatif) et les nouveaux bits sont remplacés par 0

c. Instructions de transfert

`ldb rd, (rn)S` : met dans rd le contenu de l'adresse (rn+S) sur 1 octet
`ldw rd, (rn)S` : met dans rd le contenu de l'adresse (rn+S) sur 2 octets
`stb (rd)S, rn` : met dans l'adresse (rd + S) le contenu de rn sur 1 octet
`stw (rd)S, rn` : met dans l'adresse (rd + S) le contenu de rn sur 2 octets

d. Instructions de saut

`jmp S` : la prochaine instruction à exécuter est à l'adresse S
`jzs S` : la prochaine instruction à exécuter est à l'adresse S, si le bit Z est à 1
`jzc S` : la prochaine instruction à exécuter est à l'adresse S, si le bit Z est à 0
`jcs S` : la prochaine instruction à exécuter est à l'adresse S, si le bit C est à 1
`jcc S` : la prochaine instruction à exécuter est à l'adresse S, si le bit C est à 0
`jns S` : la prochaine instruction à exécuter est à l'adresse S, si le bit N est à 1
`jnc S` : la prochaine instruction à exécuter est à l'adresse S, si le bit N est à 0

Le bit Z est mis à 1 lorsque le résultat précédent est nul, 0 sinon.
Le bit C est mis à 1 lorsque le résultat précédent comportait une retenue, 0 sinon.
Le bit N est mis à 1 lorsque le résultat précédent est négatif, 0 sinon (c'est le bit de poids fort).

e. Instructions d'entrée/sortie

`in rd` : met dans rd une valeur rentrée par l'utilisateur au clavier
`out rd` : affiche à l'écran en décimal le contenu du registre rd

f. Instructions diverses

`rnd rd, rn, S` : met dans rd un nombre aléatoire entier entre rn et S-1 inclus
`hlt` : termine l'exécution du programme

3. Liste non exhaustive de ce que le simulateur considère comme une erreur :

Afin que vous partiez sur de bonnes bases avec le simulateur, voici quelques erreurs que vous pouvez éviter facilement :

- Veillez à n'utiliser que des mots clés d'opération valides, seuls ceux cités dans les listes ci-dessus existent !
- De la même façon, il est essentiel de bien respecter le nombre d'opérandes attendu selon l'opération à effectuer, ni plus, ni moins. Accordez également de l'importance à la syntaxe de chaque opérande : une valeur immédiate doit impérativement commencer par un '#', le numéro d'un registre doit être compris entre 0 et 31, etc.
- Le programme n'autorise pas d'espaces ou de tabulations avant le nom de l'étiquette.
- Pour les instructions de transfert, Il faut toujours un S, et cela peut être un registre (rX) ou une valeur immédiate (#n). On pourra donc par exemple trouver l' instruction : `LBD r3,(r4)#0`.
- Dans les instructions de saut, on ne peut pas avoir de valeur immédiate négative à la lecture du fichier assembleur. Et à l'exécution, si la valeur lue dans le registre (S=src2) est négative (car représentation signée [-32768,32767]) alors on la repasse en positif ([0,65000]).
- Une division par zéro entraîne une erreur et un arrêt du programme.
- La génération d'une valeur aléatoire avec `rnd` entre rn et S-1 avec rn>=S est impossible et entraîne une erreur et l'arrêt du programme.

- Pour l'opération d'entrée `in`, si la valeur rentrée dépasse de l'intervalle [-32768,32767] (valeur attendue normalement car les registres sont sur 16bits signés), le programme n'affiche pas de message d'erreur et "convertit" la valeur pour qu'elle rentre dans le registre. De cette manière, l'utilisateur peut rentrer une adresse à la main pour un futur saut s'il le souhaite (puisqu'une fois à l'instruction de saut, on repasse le contenu du registre en positif /ie non signé).
- De la même manière, pour chaque opération, si le résultat dépasse l'intervalle sur 16 bits signés des registres ([-32768,32767]), alors le résultat est converti automatiquement et le bit C est mis à 1. Attention, aucun message d'erreur n'est généré.

A vous de jouer !