ARCHI - TP4 : Processeur (assembleur.py)

Sommaire

1 - Générer des fichiers de test

Il y a un assembleur en `python3` (`assembleur.py`) qui permet de traduire un code assembleur (`.risc`) en un fichier en hexadécimal (`.hex`) compréhensible par notre processeur picorisc en le mettant dans la mémoire.

Pour compiler un fichier `.risc` il suffit d'exécuter la commande suivante :

```
$ python3 assembleur.py fichier.risc
```

Cette commande va chercher le fichier `fichier.risc` dans le répertoire `tests_code` et produit le fichier `fichier.hex` dans le répertoire `tests` .

2 - Syntaxe risc

Une instructoin se note `op rd rs imm8` (quand les champs sont nécessaires pour l'instruction), et l'on met au plus une instruction par ligne. On peut écrire en minuscule ou en majuscule, l'assembleur transformera automatiquement tous les caractères en leur version majuscule. Les mots clés doivent respecter une certaine syntaxe :

- `op` : le mot clé de l'une des 16 instructions possible,
- `rd` : représente l'un des 4 registres, qui peuvent être nommés come suit :

Registre	A	B	C	PC
Mot clé littéral	`ra`	`rb`	`rc`	`pc`
Mot clé binaire	`00`	`01`	`10`	`11`

- `rs` : même syntaxe que pour `rd` ,
- `imm8` : un entier non signé sur 16 bits, écrit en hexadécimal (donc de longeur 4).

2.1 - Instructions et opérations

Mot clé	op	Opération
`imm`	0	`rd` = sext(`imm8`); `pc` ++
`add`	1	`rd` = `rd` + `rs` ; `pc` ++
`addi`	2	`rd` = `rs` + sext(`imm8`); `pc` ++
`sub`	3	`rd` = `rd` - `rs` ; `pc` ++
`xor`	4	`rd` = `rd` ^ `rs` ; `pc` ++
`or`	5	`rd` = `rd` | `rs` ; `pc` ++
`and`	6	`rd` = `rd` & `rs` ; `pc` ++
`sr`	7	`rd` = `rd` >> 1; `pc` ++
`ori`	8	`rd` = `rd` | `imm8` ; `pc` ++
`oris`	9	`rd` = `rd` | (`imm8` << 8); `pc` ++
`ld`	A	`rd` = mem[`rs`]; `pc` ++
`st`	B	mem[`rs`] = `rd` ; `pc` ++
`lt`	C	if (`rd` < `rs`) then `pc` += sext(`imm8`) else `pc` ++
`eq`	D	if (`rd` == `rs`) then `pc` += sext(`imm8`) else `pc` ++
	E	undef
`end`	F	stop le programme

2.2 - End

L'instruction `end` utilise l'op **F** qui est *undefined* par défaut et est le marqueur de fin du programme et arrête donc l'exécution de l'interpréteur.

Il n'est pas nécessaire de le mettre en fin du code `.risc`, l'assembleur l'ajoute automatiquement.

2.3 - Commentaires

Il y a la possibilité de faire des commentaires, tout ce qui suit un `#` dans une même ligne sera ignoré. Par exemple, le code suivant charge les registres A et B à 3 et 6, puis stocke dans A le résultat de A + B et les commentaires sont bien ignorés :

```
# charger les registres
imm ra 3    # ra = 3
imm rb 6    # rb = b

# addition
add ra rb   # ra = ra + rb
```

3 - Les fichiers de test

3.1 - Tests pour l'ALU

Les fichiers `op_1_3`, `op_4_6`, `op_7_9` fournissent une batterie d'exemples pour les opérations utilisant l'ALU (de 0 à 9 sauf la 7 puisque l'ALU est défectueux pour cette opération). On

peut donc vérifier que notre interpréteur marche correctement en faisant la simulation pas à pas et en vérifiant la valeur des registres.

3.2 - Tests pour la mémoire

Le fichier `op_A_B` fournit un test où l'on écrit une fois dans la mémoire puis on récupère cette valeur dans un autre registre.

3.3 - Tests pour les sauts conditionnels

Le fichier `op_C_D` fournit un test où l'on fait plusieurs sauts consécutifs afin de bien voir le fonctionnement des sauts.

3.4 - Multiplication

Le fichier `mult` implémente la multiplication entre deux entiers. Les entiers doivent être modifié dans le fichier source et sont stocké dans la mémoire aux emplacements 20 et 21 (`mem[20]` et `mem[21]`) et dupliqués aux emplacements 30 et 31 (`mem[30]` et `mem[31]`).

Le résultat est stocké dans la mémoire à l'emplacement 32 (`mem[32]`). Ainsi, on peut lire le calcul en entier, de la case mémoire 30 à 32.

3.5 - Division euclidienne

Le fichier `div` implémente la division euclidienne entre deux entiers. Les entiers doivent être modifié dans le fichier source et sont stocké dans la mémoire aux emplacements 20 et 21 (`mem[20]` et `mem[21]`) et dupliqués aux emplacements 30 et 31 (`mem[30]` et `mem[31]`).

Les résultats sont stockés dans la mémoire aux emplacements 32 et 33, `mem[32]` pour le quotient et `mem[33]` pour le reste. Ainsi, on peut lire le calcul en entier, de la case mémoire 30 à 33.