CS351@ Esisar -- MIPS Emulator Developer Documentation

Romain SERPOLLET - Loan TREHIN

Sommaire

Structure du code

*traduction.[c|h] : Module utilise pour traduire les instructions en hexadecimal

*utile.[c|h] : Module utilise pour stocker les fonctions qui peuvent servir dans tous les autres modules

*memory.[c|h] : Module utilise pour gérer la mémoire, la memoire y est définie

*registre.[c|h] : Module utilise pour gérer les registres, ils y sont définis

*struct.h : définition des structures

*data.h : table de correspondances pour la traduction des instructions

*main.c : fichier main

Les modes de fonctionnement

1. Lecture de fichier

1.1. Traduction du fichier

On va lire ligne par ligne le fichier tant qu'on arrive pas à la fin avec la fonction fget.
Si la ligne n'est pas une ligne vide alors on va la traiter pour en faire une instruction.
On a choisi de créer une structure instruction se composant de :

- un tableau de string pour contenir les mots de l'instruction
- le nombre d'opérandes de l'instruction
- Le numéro de l'instruction correspondant à sa ligne dans nos tables de correspondances
- Un tableau de char de 33 caractères contenant la traduction de l'instruction en binaire
- Un tableau de char de 9 caractère contenant la traduction de l'instruction en hexadécimal
- Un bit pour préciser l'activité de notre instruction (1 si actif 0 sinon)

Lors de la traduction, on vient remplir une é une chaque variable de notre instruction

Puis on rentre un pointeur vers notre instruction dans un tableau pour sauvegarder toutes les instructions du programme

On écrit notre instruction en hexadécimal dans le fichier de destination

1.1.1 Découpage de notre ligne

On utilise la fonction strtok pour découper la ligne en fonction de délimiteurs.
Pour utiliser cette fonction, on lit chaque découpage et on l'écrit dans notre tableau de string de l'instruction

1.1.2 Traduction

On a une table contenant les décompositions de chaque instruction, représentées par des lettres.
Tant qu'on a pas atteint la lettre F, on traduit lit les lettres une à une en plaçant dans le tableau binaire de l'instruction les bits correspondants à la lettre.
S : Special : 000000
Z : Zero : 00000
N : Nom : .code
F : Fin
I : immediate: 16b
J : jump : 26b
O : ofset : 16b
R : 00001
0 : Premiere opérande
1 : Deuxieme opérande
2 : Troisieme opérande

Ensuite on concatène nots mots en binaire en un tableau de 32 bits.
Puis on traduit ce tableau en hexadécimale.

1.1.3 Stockage

On a choisi de stocker tous les pointeurs vers instructions dans un tableau pour pouvoir y accéder depuis notre module main.
A la fin de chaque boucle, une fonction du module memory vient écrire la valeur en binaire de l'instruction dans la mémoire.

A la fin de la traduction, le module de traduction renvoie une valeur qui est la derniere adresse
utilisée dans la mémoire pour stocker le programme.

1.2 Exécution du programme:

1.2.1 Gestion du Programme Compteur:

Le programme compteur est stocké dans un registre, nous avons donc une fonction du module registre qui va lire la valeur du pc au début de chaque boucle d'exécution.
Nous nous servons de cette valeur pour aller lire la valeur de l'instruction dans la mémoire. Lorsque le pc dépasse la valeur renvoyée précédemment pas le module traduction alors c'est que le programme est fini.

A la fin de chaque boucle, on incrémente le programme compteur pour passer à l'instruction suivante.
Les instructions qui modifient le pc agissent directement sur le registre du pc.

1.2.2 Exécution de l'instruction courante:

On utilise le module instruction pour executer les instructions
On passe en paramètre le pointeur vers l'instruction courante.

1.2.3 Affichage:

Pour l'affichage de l'exécution, nous avons choisi d'afficher dans un tableau a 6 colonnes:

- Valeur pc en hexa
- Numero du registre modifié (renvoie X si aucun registre n'a été modifié)
- Valeur entrée dans le registre modifié (renvoie X si aucun registre n'a été modifié)
- Adresse de la mémoire modifiée (renvoie X si aucune mémoire n'a été modifiée)
- Valeur écrite dans la mémoire (renvoie X si aucune mémoire n'a été modifiée)
- Instruction courante

Et a la fin de l'exécution nous affichons la liste de tous les registres et de leur valeur.

---

2. Lecture du fichier en mode pas à pas

2.1 La traduction

La traduction du fichier est strictement la même que pour le mode fichier fichier

2.2 Exécution

L'exécution fonctionne de la même manière, seulement, à la fin de chaque boucle on utilise la fonction getchar() pour attendre l'entrée d'un appuis sur la touche entrée. On utilise la même boucle que pour le mode fichier à une différence près; lors du choix du mode, on modifie une variable de type int qui représentera la mode. ainsi, lors de la boucle d'exécution, si la variable est à 1 alors on est en mode pas à pas et donc on appelle cette fonction getchar, sinon on continue normalement.

2.3 Affichage

L'affichage est le même que pour le mode fichier complet.

---

3. Mode console

Le mode console fonctionnera tant que l'on a pas entré la commande "exit" .

3.1 Traduction

On utilise la fonction fget pour récupérer la ligne entrée dans la console. Ici on ne passe pas un fichier en paramètre mais "stdin".
Ensuite on va traduire la ligne de la même manière qu'avec les autres modes, en remplissant notre structure instruction.

3.2 Exécution

Une fois que la traduction est faite on va venir effectuer l'instruction. Ce mode est plus simple à gérer puisqu'il n'est pas nécessaire de gérer le programme compteur.
De plus, toutes les instructions qui touchent au pc, comme les sauts ou les comparaisons, ne seront pas utilisées dans ce mode.

3.3 Affichage

Nous avons choisi d'afficher l'ensemble des registres après chaque commande, pour faciliter le codage pour l'utilisateur.

Les modules

traduction.[c|h]

Le module traduction est utilisé pour traduire les instructions. Ces fonctions utilisent les (*types*) instructions pour les modifier et compléter ces variables.
Nous avons fait ce choix car avec cette méthode, on utilise la même variable instruction durant toute la traduction de la ligne.

Ce module utilise les header *struct.h* et *data.h* qui définissent respectivement les structures et les tables de correspondances.

Il utilise aussi le module *memory* pour pouvoir écrire les instructions qu'il traduit dans la mémoire.
Ce module constitue toute la première partie du projet.

Lors de la traduction, on utilise un système de complément à deux pour encoder les nombre négatifs, ainsi, on ne peut coder que des nombres allant de $-(2^{32}-1)$ à $(2^{32}-1)$.

utile.[c|h]

Nous avons créé un module utile pour y définir toutes les fonctions qui peuvent nous servir dans les autres modules, comme les fonctions de conversion binaire en décimal par exemple.
On y défini aussi les fonctions qui servent à l'éxecution des instructions, comme le *"et bit à bit"* pour l'instruction AND.
Nous avons fait ce module pour éviter d'avoir à redéfinir des fonctions similaires dans chaque module.

memory.[c|h]

Le module memory est le seul qui peut accéder à la mémoire puisqu'elle y est défini, nous avons fait ca pour des raisonde sécurité, pour éviter que d'autres modules viennent écrire des valeurs non correcte dans la mémoire, par ailleurs, il y a une fonction qui vérifie l'intégritée des données écrite dans la mémoire. Avant d'écrire dans la mémoire, le module vérifie que ce que nous voulons écrire est bien des bits et non d'autres caractères.
On a choisi de créer une mémoire comme un tableau de char de 5000 lignes et 9 colonnes, car nos bits sont des chars. La vrai mémoire d'un processeur MIPS est composée de blocs de 8 bits donc chaque ligne fait 9 pour pouvoir mettre '\0' à la fin.

Comme les instructions et les valeurs font 32 bits, notre fonction ecrireMemoire() va écrire sur 4 adresse mémoires consécutives en partant de l'adresse la plus petite.

Ce module permet aussi de nous afficher la mémoire de l'adresse a à l'adresse b.

registre.[c|h]

Ce module est celui où sont défini les registres, il est le seul à pouvoir y accéder.
Nous avons utilisé un tableau de char de 35 lignes de 33 caractères chacune afin de pouvoir y placer des valeurs de 32 bits.
Il y a 32 registres normaux + 3 registres spéciaux : PC, LO, HI
De plus, les registres PC et x0 sont initialisés à 0 au lancement du programme.
Nous avons aussi créer un système de protection des registres, en effet, x0 doit toujours être à 0, il est donc impoosible au programme d'écrire dedans. Il en est de même pour les registres 26, 27 et 30 puisque dans la documentation il nous est écrit que ces registres sont réservés.

Schéma connexions modules