CS351 : Projet MIPS

Ce projet consiste à créer un émulateur de microprocesseur MIPS. Dans ce compte-rendu, nous exposerons le rôle et le fonctionnement de chaque fonction ou groupe de fonctions dans l'ensemble du projet. Puisque le travail a globalement été réalisé en binôme, nous ne différencierons pas les fonctions par rapport à qui les a créées.

Le projet contient :

- des fichiers tests, comprenant des lignes d'instruction en assembleur
- instructiontohex.txt, qui associe des informations pour chaque instruction
- main.c
- parser.c
- instructions.c
- memory.c
- interpreteur.c
- moduletp4.c, un module du tp4 que nous ne développerons pas ici
- Makefile
- tous les headers associés

parser.h :

On déclare une structure **instruction** à 4 champs de type chaîne de caractères. Cette structure permettra de stocker n'importe quelle instruction assembleur. Elle contient 1 champ opcode et 3 champs opérandes. En effet, une instruction MIPS a au maximum tous ces champs.

instructions.h :

On déclare une structure **numinstruction** qui a 9 champs de type entier. Le type numinstruction est utile pour stocker une instruction sous une forme exploitable puisque chaque champ représente une valeur d'une instruction. Une instruction ne peut pas remplir tous les champs de la structure, on les laissera donc vide et ça ne posera pas de problème puisqu'on ne les exploitera pas.

memory.h :

On déclare deux structures semblables : **memory_struct** et **register_struct**. La structure mémoire contient un champ tableau de caractères non signés et nous avons expliqué ce choix dans un document précédent. La structure des registres contient un champ tableau d'entiers. Son choix et son fonctionnement sont également décrit dans un document précédent.

parser.c :

La fonction **cut_instruction** transforme une ligne d'assembleur en un type instruction. On transmet à la fonction une ligne du fichier texte comprenant des instructions en assembleur. Dans la fonction, on lit la ligne jusqu'à un séparateur et on stocke ce qu'on a lu dans le champ correspondant de la structure. On répète l'opération jusqu'à la fin de la ligne.

registerToInt reçoit un registre de type chaîne de caractère (sous la forme $0) et le transforme en un entier correspondant à son numéro de registre.

C'est la même logique pour **immediateToInt** et **targetToInt** qui reçoivent des chaînes de caractères correspondant à des types d'opérandes et les convertissent en entier.

translate_instruction traduit une instruction de type instruction en sa valeur hexadécimale correspondante. Selon le type de l'instruction (correspondance dans instructiontohex.txt), on formate les 32 bits de l'instruction reçue en paramètre. Par la même occasion, on remplit les champs de bit en fonction de la valeur des champs de l'instruction.

ecrit_fichier écrit un tableau de nombre ligne par ligne dans un fichier. On utilisait cette fonction pour mettre les traductions hexadécimales des instructions dans un fichier.

lit_fichier lit une ligne d'un fichier. on utilise la fonction fgets en demandant 100 caractères, ce qui est suffisant pour une instruction. Si il y a un commentaire trop long à la suite de l'instruction, il sera coupé mais c'est sans importance puisqu'on ne les utilise pas et on les coupe plus loin. si la ligne contient moins de 100 caractères, la fonction capturera toute la ligne et n'ira pas plus loin.

mange_blanc enlève les espaces avant les lignes d'instruction, par précaution. Pour ce faire, tant que l'instruction commence par un espace, on décale toute la ligne vers la gauche caractère par caractère.

transformeTotal est une macro fonction qui, en prenant en entrée un fichier texte rempli d'instructions assembleur, crée un fichier contenant la traduction du fichier en hexadécimal. Pour ce faire, on traite chaque ligne du fichier reçu avec les fonctions vues précédemment pour traduire la ligne en hexa, stocker cette valeur dans un tableau puis écrire toutes les valeurs du tableau dans le fichier de sortie. On n'écrit pas directement les traductions dans le fichier car à cause de la boucle, les accès au fichier dans la fonction appelée échouaient.

loadmemory charge en mémoire les traductions hexa des instructions assembleur du fichier reçu. Cette fonction agit de la même manière que la précédente mais stocke en mémoire plutôt que dans un fichier grâce à la fonction swi décrite plus tard.

viderBuffer est une fonction auxiliaire qui permet de vider le buffer pour ne pas avoir de problème avec les scanf. Pour cela, on lit sur l'entrée standard les caractères un à un tant que ce n'est pas la fin d'un fichier ou un retour à la ligne.

instruction.c :

readinstr transforme une instruction qui se trouve en mémoire à l'adresse passée en paramètre en un type numinstruction. On va d'abord lire en mémoire une instruction en hexa grâce à la fonction rw. Ensuite, en fonction du type de l'instruction, on affecte les différents champs de la numinstruction avec des décalages pour placer les champs correctement dans les 32 bits. On utilise les masques par précaution, pour que les champs ne débordent pas les uns sur les autres. On retourne la numinstruction.

La suite des fonctions d'opérations réalisent les opérations attendues en utilisant les structures et fonctions mises en place. Pour la majorité d'entre elles, la logique est triviale. Cependant, le temps de test de ces fonctions était surprenamment long puisque les écritures sont compliquées.

La fonction **operation** permet d'effectuer l'opération correspondant à l'instruction lue. Selon la valeur hexa de l'opération, on décide quelle fonction appliquer ensuite. Toutefois, certaines instructions ont le même opcode. Dans ces cas-là, on les différenciait en fonction du type qu'on leur affectait dans readinstr. Mais là encore, pour différencier ROTR et SRL, il fallait aller plus loin. En regardant la composition de ces instructions, nous avons remarqué qu'on pouvait les différencier sur le bit 21.

memory.c :

lw charge un mot en mémoire dans un registre. On stocke dans un registre la valeur contenue à l'adresse contenue dans un registre passé en paramètre. On peut décaler la valeur de l'adresse par un offset.

De la même manière, **sw** charge un mot d'un registre vers la mémoire.

rw lit un mot en mémoire à une adresse donnée et le retourne sans transformation. La valeur retournée est un entier long non signé.

swi charge un mot passé en paramètre en tant qu'entier long en mémoire, c'est-à-dire qu'on écrit directement en mémoire la valeur passée en paramètre.

wr écrit une valeur dans un registre. Certains registres sont protégés : on ne peut pas modifier leur valeur de cette façon. Ce sont les registres 0, GP, HI et LO.

printRegisters et **printMemory** affichent respectivement les valeurs des structures de registre et de mémoire.

init_mem et **init_reg** initialisent simplement les tableaux des structures mémoire et registres à 0.

interpreteur.c :

lit_ligne est une fonction pour le mode pas à pas qui permet d'afficher l'instruction exécutée. Elle lit le fichier texte ligne par ligne comme lit_fichier. Elle affiche ensuite la ligne lue et sa traduction hexa. Pour lire le fichier, on replace le curseur au début du fichier à chaque appel puis on se déplace jusqu'à la ligne voulue grâce au GP.

step_by_step est la fonction qui propose à l'utilisateur les choix en mode pas à pas. Elle appelle des fonctions en fonction du choix de l'utilisateur. On peut quitter ce mode en tapant exit.

interactif est la fonction du mode interactif. Comme certaines instructions sont indisponibles en mode interactif puisqu'on n'utilise pas la mémoire, on trie déjà les instructions entrées par l'utilisateur. Ensuite, on applique simplement les fonctions utiles pour transformer l'instruction en résultat. On déclare une structure mémoire invisible pour l'utilisateur pour utiliser les fonctions déjà existantes, qui en ont besoin dans leurs paramètres.

interprete prépare les structures et initialise les registres, puis charge le programme en mémoire. Ensuite, on différencie les modes pas à pas et pas pas à pas : on lit les instructions dans le fichier, exécute l'opération correspondante puis écrit dans les registres son effet. Si l'option pas est spécifiée, alors on appelle la fonction step_by_step, sinon on boucle jusqu'à la fin du fichier. On affiche finalement l'état des registres et de la mémoire.

main.c :

En utilisant les arguments argc et argv de la fonction main, on récupère les informations du mode et le nom du fichier pour émuler. On agit donc en fonction du nombre d'arguments et de leur valeur en appelant les fonctions adéquates.

Globalement sur le projet, nous n'avons pas eu beaucoup besoin de travailler individuellement chez soi. Dans ces cas-là, toutefois, nous prenions le temps d'expliquer ce que nous avions fait et comment à notre binôme. En outre, nous passions parfois jusqu'à plusieurs heures pour corriger un problème.