Premier livrable – Projet Info SEI

SIMULATEUR MIPS

Nicolas SCHOEMAEKER
Éric FELTRIN

Grenoble INP - Phelma
2A – SEI

Introduction

Ce projet s'inscrit dans le cadre de l'enseignement en informatique et programmation de la deuxième année d'étude ingénieur, dans la filière *SEI*. Ce projet s'effectue en parallèle avec la filière *SICOM*.
Il consiste en la réalisation d'un simulateur de microprocesseur MIPS 32 bits pouvant simuler l'exécution de programmes écrits en assembleur dans un fichier ELF. L'environnement hôte est une distribution Linux basique, à laquelle la librarie *readline* a été rajoutée. Le simulateur est écrit en langage C (version C99).

Le projet est divisé en quatre livrables, qui constituent les quatre grandes étapes de dévelopement :

- Réalisation de l'environnement du MIPS (implémentation des registres, de la mémoire et des différents paramètres relatifs au microprocesseur) et de l'interpréteur de commandes
- Chargement du fichier ELF, décodage des instructions et chargement de la mémoire
- Simulation d'un programme et gestion des points d'arrêt
- Gestion de la relocation

Ce rapport accompagne le premier livrable, à savoir la mise en place d'un environnement de simulation et la réalisation d'un interpréteur de commandes.

Dans une optique de développement en équipe et dans un soucis de propreté du code, nous avons décidé de créer une paire de fichiers .h et .c pour chaque fonction de l'interpréteur de commande (*lp.h, lp.c, dm.h, dm.c, ...*) contenant une fonction de parsing, appelée "parseNom" (en capitalisant la première lettre de la fonction, par exemple "parseLm", "parseDm") et une ou plusieurs fonctions d'exécution.
Toujours dans l'optique d'un travail en équipe, nous avons travaillé avec le système de gestion de versions de fichier *Git*, sur un *repository* externe, hebergé par *Github*.
Enfin, dans une optique ingénieure, nous avons décidé de coder en anglais, donc les noms de variable, de fonction, les commentaires et les chaînes de charactères affichées à l'écran seront écrites en anglais.

L'environnement d'emulation

Les contraintes

La cible (*target*) d'émulation est un microprocesseur MIPS 32 bits, dont le codage des mots se fait en norme *little endian*. Il nous est demandé d'émuler 4Go de memoire RAM, 32 registres géneraux de 32 bits chacun, un registre PC, un registre HI et un registre LO, de 32 bits également.

Notre implémentation

Afin de séparer au mieux notre code, nous avons défini une structure *mips* dans un fichier .h séparé. Pour des raisons d'eventuelle réutilisation du code et de propreté, nous avons décidé d'instancier un *mips* dans la fonction *main*, et de passer un pointeur vers ce *mips* à chacune des fonctions, plutôt que d'utiliser une variable globale.

La structure du *mips* contient :

- une chaine de charactère (*char **) permettant de donner un nom à l'instance (ce qui peut permettre d'émuler plusieurs instances en même temps).
- un tableau de 32 registres (le type *registre* étant un *typedef* d'un *unsigned int*, puisque ce dernier est implementé sur 32 bits sur nos machines)

- trois registres supplémentaires *HI*, *LO*, et *PC*.
- trois pointeurs *unsigned char ** pour implementer les trois segments *.text*, *.data* et *.bss*.
- trois *unsigned int* qui stockent la taille de chacun de ces segments.

Les fonctions de l'interpréteur de commandes

Fonction LP

La fonction *LP* pour *Load Program* devra être chargée de charger un programme conteu dans un fichier ELF afin de simuler son exécution sur la machine hôte.

Dans ce premier livrable, nous avons implémenté la fonction de telle sorte qu'elle affiche un message d'erreur lorsqu'elle est appelée sans aucun paramètre.
Lorsqu'un nom de ficher lui est passé, elle vérifie que ce dernier existe et est accessible en lecture. Elle affiche un message d'information lorsque plusieurs fichiers lui sont passés en paramètres, indiquant qu'elle ne prendra que le premier en compte.

Fonction EX

La fonction *EX* pour *Exit* pemet de quitter proprement l'interpréteur de commande, en retournant un code de retour indiquant que l'exécution du programme s'est déroulée avec succès.

Cette fonction était déjà implémentée, nous nous somme contentés de la mettre dans une paire de fichiers .h et .c.

Fonction DM

La fonction *DM* pour *Display Memory* prend soit une adresse, soit une adresse et un nombre d'octets, soit une adresse de début et une adresse de fin et affiche le contenu de la mémoire à ces adresses, par lignes de 16 octets, chaque ligne étant préfixée par l'adresse de l'octet le plus à gauche (le choix de 16 octets constitue les "dizaines" de la base héxadécimale, ainsi les indices de lignes auront la forme : 0x000, 0x0010, 0x0020, ...

Le parsing de la fonction *DM* a été implémenté avec un *sscanf* et une vérification sur le nombre de *match* que fait *sscanf* sur les chaînes de type :

- "%x:%d" : tente de parser une adresse (en héxadécimal) et un nombre d'octets (en décimal) séparés par deux-points ":"
- "%x~%x" : tente de parser deux adresses héxadécimales séparées par le tilde "~".
- "%x" : une adresse héxadécimale

Les trois modes d'adressage de la fonction *DM* ont été implémentés, sans support du caractère d'espacement entre les séparateurs ":" et "~".

De plus, une fonction d'initialisation d'un *mips* (appelée dans le *main*) initialise trois segments de mémoire de 4096 octets chacun, ce qui permet à notre implémentation de la fonction *DM* d'afficher le contenu "réel" de segments de mémoire.

Fonction DA

La fonction *DA* pour *Display Assembly* prend une adresse et éventuellement un nombre d'instructions et affiche à l'écran le code assembleur de l'instruction (ou des instructions) présente à l'adresse donnée (et des n instructions suivantes le cas échéant).

Dans le même esprit que la fonction *DM*, la fonction de parsing tente de parser d'abord une combinaison "adresse:nombre", puis seulement une adresse. La but de la fonction *DA* étant d'afficher une instruction de code, il est nécessaire que l'adresse (ou les adresses) donnée(s) soient dans le segment *.text* du *mips*. Ainsi notre fonction *DA* affiche un message d'erreur lorsque l'adresse donnée sort du segment *.text* (mais affiche quand même les premières instructions correspondantes aux premières adresses si celles-ci sont valides).

Fonction DR

La fonction *DR* pour *Display Register* affiche le contenu du registre passé en paramètre, sur 8 chiffres héxadécimaux (32 bits) ou l'ensemble des registres si aucun argument lui est passé en paramètre.

Notre implémentation établit une correspondance entre les noms des registres ($zero, $fp) et leur numéro ($0, $30) à l'aide d'un tableau. Le parsing s'effectue en examinant si le premier charactère est un dollar '$' et en parsant un nombre compris entre 0 et 31 ensuite ou si c'est une chaîne de charactère présente dans le tableau de correspondance. L'appel à la fonction d'execution, en passant l'index du registre à afficher est ensuite effectué, permettant l'affichage de la valeur.

Fonction LM

La fonction *LM* pour *Load Memory* permet d'écrire une valeur passée en paramètre dans une adresse mémoire.

Notre implémentation parse les deux valeurs héxadécimales (et affiche une erreur si l'une des deux valeurs n'est pas dans le bon format), détermine ensuite dans quel segment l'adresse mémoire donnée se trouve, calcule le décalage à effectuer par rapport au pointeur pointant vers le début de cette section et y place la valeur passée en paramètre.

Fonction LR

La fonction *LR* pour *Load Register* charge une valeur passée en paramètre dans un registre, passé en paramètre.

Notre implémentation du parsing est le même que celui pour la fonction *DR* et l'appel à la fonction d'exécution charge la valeur, soit dans le tableau des 32 registres généraux, soit dans l'un des trois autres registres.

Conclusion

L'implementation de ces 7 fonctions de départ permettent d'avoir un interpréteur de commandes basique, autorisant le déroulement de la suite du projet. Certaines de ces fonctions verront leur implementation evoluer au fil du déroulement du projet, notamment lors de la prise en compte de contenu réel issu de fichiers ELF.

Nous avons tenté au maximum de coder de manière générique, en ne codant en dûr que le minimum de choses, de manière à ce que ces futures modifications liées à la prise en compte des fichiers ELF soit la plus minimaliste possible.