Rapport du projet d'informatique Simulateur MIPS

Buléon Benjamin,Ayel FlorianSICOM 2A

17 décembre 2013

Table des matières

Introduction

L'objectif de ce projet informatique est de réaliser sous Linux, en langage C, un simulateur d'une machine MIPS 32bits permettant d'exécuter et de mettre au point des programmes écrits en langage du microprocesseur de la machine MIPS. Le rôle d'un simulateur est de lire un programme donné en entrée et d'exécuter chacune des instructions avec le comportement de la machine cible (ici MIPS 32 bits). Les simulateurs permettent notamment de prototyper et deboguer des programmes sans la contrainte de posséder le matériel cible (ordinateur avec un microprocesseur MIPS 32 bits).

Plus précisément, le simulateur que nous avons du réaliser prend en entrée un fichier objet au format **ELF** et permet :

- 1. Son exécution complète
- 1. Son exécution pas à pas à travers un interpréteur de commandes
- 3. Son débogage (ex. affichage de la mémoire interne, détection d'erreurs, rapport d'erreur...)

Par ailleurs, à la fin du projet, notre simulateurest capable de gérer les fichiers dit relogeables qui peuvent être chargés à n'importe quel endroit de la mémoire.

Le simulateur a été realisé en langage C et fonctionne dans l'environnement Linux. Pour ce projet nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions du MIPS. Ces instructions du microprocesseur MIPS seront simulées/réalisées par des appels de fonctions du langage C, elles-même compilées en une série d'instructions pour le microprocesseur effectuant la simulation (celui de la machine sur lequel est effectué le travail).

Dans ce rapport, nous allons présenter notre travail, étape par étape suivant les 4 incréments du programme que nous avons réalisés.

Chapitre 1

1er Incrément

Le projet informatique que l'on va réaliser a pour but de réaliser un débogueur de programmes écrits en langage assembleur MIPS. Dans l'étape 1 on s'attache à créer le simulateur et l'interpréteur, gérer les premières commandes passées au shell ou par fichier et créer la structure du MIPS.

Ce rapport s'articulera autour du plan suivant :
- 1. Mise en place du projet
- 2. Ce que l'on a fait
- 3. Ce qu'il reste à faire

1.1 Mise en place du projet

Etant répartis en binômes la première tâche à laquelle nous nous sommes employés a été de définir les rôles de chacun et quelques deadlines afin de ne pas se faire surprendre par le temps.

Même si nous avons travaillé chacun de notre côté, nous avons organisé 4 mises en commun de nos codes depuis le début, en dehors des horaires spécialement réservées au projet. Ces "rencontres" nous ont permis de repartir sur la même base, et aussi de réassigner de nouvelles tâches à effectuer avant la rencotre suivante.

Dans l'ensemble on peut dire que notre binôme a connu un bon départ. Nous avons vite appris à connaître les habitudes de l'autre et la cohésion a été instinctive.

1.2 Ce que nous avons fait

En termes de code pur, vous trouverez ci-joint nos fichiers .c dans src, les .h dans le include et les tests dans le dossier test.

Structure.h est le fichier dans lequel nous avons défini la structure du MIPS. Nous avons choisi, sur conseil de notre professeur encadrant, de définir un MIPS comme une structure contenant une mémoire et des registres.

La mémoire est définie quant à elle comme 3 segments DATA, BSS et TEXT caractérisés chacun par leur adresse de départ dans le MIPS (fictive), et leur taille.

Dans le fichier simMips.c nous avons simplement ajouté l'initialisation du MIPS et les reconnaissances des commandes dans les fonctions de parsing.

L'ensemble des fonctions que nous avons créées se trouve dans le fichier fonctionsStep1.c. Le fichier testsStep1.c complète ce fichier puisqu'il contient les tests sur les adresses registre, les adresses mémoire et les valeurs contenues.

Dans le dossier test nous avons tenté de créer tous les cas possibles de commandes qu'un utilisateur pourrait rentrer, et de tester la réponse renvoyée par notre éxécutable. Après maintes retouches de notre code, nous sommes parvenus à le corriger entièrement de sorte que tous les tests passent.

1.3 Ce qu'il reste à faire

Dans l'étape 2 du projet, nous allons devoir coder l'ensemble des commandes offertes par notre simulateur.

1.4 Conclusion

Cette étape 1 n'était sans doute pas la plus difficile du projet mais elle exigeait du sérieux et de l'application. La mise en place du binôme est aussi un moment important de ce genre de projets à mener en groupe. Nous avons géré cette étape sans encombre puisque notre code fonctionne, même s'il n'est sans doute pas partout optimal. La base pour commencer l'étape 2 est solide.

Chapitre 2

2ème incrément

Après avoir démarré le projet et coder quelques instructions, nous nous sommes attachés au cours de cet incrément, à coder les commandes 'da' et 'lp' qui visent à charger un fichier ELF dans la mémoire du MIPS et à le désassembler.
Ce rapport s'articulera autour du plan suivant :

- 1. Répartition du travail
- 2. LP et DA
- 3. Ce qu'il reste à faire

2.1 Répartition du travail

Afin de faciliter le partage d'information au sein de notre binôme nous avons utilisé git et son support internet github. Ceci nous a permis d'avoir toujours sur internet la version la plus avancée et aboutie de notre projet, et d'y avoir accès rapidement (une ligne dans le terminal).

2.2 LP et DA

En termes de code pur, vous trouverez ci-joint nos fichiers .c dans src, les .h dans le include et les tests dans le dossier test.

La principale difficulté pour la fonction LP a été d'imbriquer la fonction mips-loader dans notre code en renommant nos structures préalablement créées dans l'incrément 1.

A l'entrée de DA, si l'adresse passée en paramètre n'est pas multiple de 4 (par exemple, da 0x00000003 :12), nous avons pris le parti de décaler l'adresse et de se ramener au début d'une instruction (dans l'exemple on se ramène à 0x00000000).

Pour les commandes nous avons choisi de passer par des fichiers. A la lecture de la chaîne instruction binaire, on détecte grâce à l'opcode le type de commande (I, J ou R). Grâce au dictionnaire et à typeIJ.desc (ou typeR.desc selon le cas)

on détecte la commande correspondante. Ensuite l'ouverture de cmd.desc (par exemple ADDI.desc) permet de déterminer le format de l'instruction et affiche les arguments en conséquence.

Pour DA, nous avons utilisé deux fonctions annexes :
getbits : elle prend en paramètres un entier sur 32 bits, un bit start et un bit stop, elle sélectionne seulement la partie comprise entre start et stop et retourne donc un entier de taille stop-start bits.
cmdSearch : elle prend en paramètres le type de la commande et son opcode si c'est un type I ou J (ou sa fonction si c'est un type R d'opcode nul), ouvre les flux du dictionnary et retourne l'instruction correspondante à l'opcode (ou fonction) passée en paramètre.

Pour faciliter l'utilisation, nous avons mis en place un rapide manuel d'utilisation des commandes donnant les formats attendus par le simulateur. On y accède en tapant la commande «man» dans le shell.

Nous avons compléter le dossier test permettant de vérifier les réponses de notre programme dans différents cas.

2.3 Ce qu'il reste à faire

Dans l'étape 3 du projet, il s'agira principalement de créer le code correspondant à chaque instruction, de sorte à traiter vraiment le code assembleur et plus juste l'afficher à l'écran. Il reste à gérer aussi le problème de la relocation mais ce sera fait dans l'incrément 4.

2.4 Conclusion

L'étape 2 était sans doute plus difficile que l'étape 1 puisqu'elle nécessitait de notre part une certaine imagination pour gérer la reconnaissance des commandes. Nous avons fait en sorte de trouver une méthode qui nous permette de rajouter des commandes aisément, sans devoir modifier tout notre code. Malgré deux ou trois problèmes mineurs dont nous vous avons fait part par mail, nous sommes plutôt satisfaits de cette étape, et on espère pouvoir commencer l'étape 3 sur une bonne base.

Chapitre 3

3ème incrément

Chapitre 4

4ème incrément

Pour la quatrième étape de ce projet, l'objectif était de s'occuper de la relocation et de terminer la mise en forme du code.

Ce rapport s'articulera autour du plan suivant :
- 1. relocation
- 2. Nettoyage du code et amélioration

4.1 relocation

Le coeur de ce dernier incrément était la mise en place de la relocation. En effet, lorsque le fichier est compilé en fichier objet, on ne sait pas dans quelles zones mémoire vont être placées les différentes sections. Il est donc nécessaire d'effectuer des relogements, une fois que les zones mémoires sont définies. Dans ce projet, nous ne nous sommes intéressé qu'a 4 types de relocation, R_MIPS_32, R_MIPS_24, R_MIPS_LO16 et R_MIPS_HI16, parmi la trentaine existante. Nous avons dans cette partie effectué des changements directement en mémoire pour modifier les valeurs des éléments en mémoire pour les adapter aux emplacements attendus.

4.2 Nettoyage du code et amélioaration

Nous avons profité de ce dernier incrément pour nettoyer le code des précédents incréments pour pouvoir y voir plus clair, en supprimant les bout de code inutile. Nous avons aussi pris le parti d'afficher nos instructions de façon a ressembler le plus possible à la sortie de mips-objdump. Nous avons donc mis en place un affichage prenant en compte l'affichage des étiquettes pour segmenter notre affichage, et le rendre plus lisible. En effet, l'appel à une procédure est clairement délimité lors d'une utilisation de la fonction Run.

D e plus, nous avons voulu nous amusé un peu, en écrivant SimMips en couleur, au demarrage de l'application. Pour cela, nous avons créé une nouvelle macro dans notifiy.h, en utilisant les séquences d'échappement pour ajouter de la couleur, et pour ne pas avoir à modifier tout nos tests, nous l'écrivons dans stderr.

E nfin, nous avons implémenté une quatrième section : la pile. La mémoire lui est allouée, le point difficile consiste ensuite à l'utiliser dans le code assembleur.

4.3 Conclusion

Cette dernière étape marque la conclusion de notre projet. Nous arrivons à un stade ou notre programme est capable de lire et debugger un fichier ELF. Depuis ce dernier incrément, notre programme gère également les fichiers relogeables.

C ependant ce programme est loin d'être parfait, même si il répond au cahier des charges imposé. Pour l'améliorer, il serait interessant de gérer les plus de 200 instructions, ou les 30 types de relocation. Pour les plus courageux, il pourrait être interessant d'ajouter une interface graphique à l'image de DDD, à notre logiciel pour l'instant en console.

Chapitre 5

Retour sur l'organisation du projet

Conclusion