Projet informatique SICOM 2A : EmulMips

Compte rendu

Table des matières

Introduction

Le dernier livrable avait pour objectif la mise en place de la relocation dans notre programme. La relocation permet de pouvoir écrire des programmes avec des symboles dont on ne connaît pas les adresses au moment de l'écriture du programme, mais qui seront connues lors de la lecture du programme par l'émulateur. Cette dernière partie n'étant pas très longue, le temps disponible a été mis à profit pour finaliser les éléments des livrables précédents et faire des tests. Ammar a travaillé sur la relocation et la libc ; Ambre a fini le codage des instructions et les a testées.

I – Mise en place de la relocation

Partie principale de ce livrable, elle consistait à mettre à jour la mémoire lors du chargement d'un programme pour prendre en compte le principe de relocation.

La relocation se fait dans le fichier load,c, lors du chargement d'un programme dans l'émulateur (commande load nom_fichier,o de l'interpréteur). Une fonction **reloc_segment** a été réalisée afin d'effectuer la relocation d'un segment en mémoire.

Le prototype de la fonction est :

```
void reloc_segment(FILE* fp, segment seg, mem memory,unsigned int endianness,stab symtab)
```

Elle récupère la table de relocation d'un segment puis pour chaque entrée de cette table, va chercher l'adresse du symbole à reloger dans la table des symboles et mettre la mémoire à jour en fonction du type de relocation (nombres de bits à modifier dans l'instruction en mémoire). Il faut donc lors du chargement en mémoire du programme, ne pas oublier de mettre à jour la table des symboles avec l'adresse des segments en mémoires.
Les différents types de relocation sont traités à l'aide d'un switch.

La fonction de relocation semble marcher, elle modifie bien en mémoire les instructions comme attendu. La vérification à été faite en affichant le mot à modifier avant et après relocation et en comparant le résultat attendu avec celui obtenu.

Toutefois, l'exécution du fichier « relocation,o » ne marche pas bien et entraine une boucle infinie. Cela semble logique si on considère le code assembleur avec soit disasm ou mips-objdump -d section=,text programmes/relocation,o :

```
Disassembly of section .text:

        00000000 <write-0x14>:
           0:  20043039   addi   a0,zero,12345
           4:  0c000005   jal 14 <write>
           8:  00000000   nop
           c:  10000005   b   24 <end>
          10:  00000000   nop
```

```
00000014 <write>:
  14:  3c010000   lui at,0x0
  18:  ac240008   sw  a0,8(at)
  1c:  03e00008   jr  ra
  20:  00000000   nop

00000024 <end>:
  24:  0000000c   syscall
```

En faisant un jal <write>, on se retrouve en <write> où on va par la suite exécuter l'instruction jr ra or ra est nul à ce moment, on se retrouve à l'adresse 0 et on recommence. Donc à priori, il est normal de se retrouver dans une boucle infinie.

II – Amélioration des livrables précédents

Plusieurs fonctions du premier livrable ont été modifiées, afin que certaines fonctions de l'interpréteur (assert et set par exemple) puissent prendre comme paramètres des valeurs hexadécimales ou entières indifféremment.

Nous avons décidé que la réinitialisation des registres et des breakpoints n'aurait lieu qu'au chargement d'un nouveau programme, et non au début de l'exécution du programme.

La fonction set, ainsi que les instructions SB et SW, ne modifient pas le contenu non alloué de la mémoire, car pour pouvoir le faire et garder des adresses de segments qui se suivent, il faudrait réallouer tous les segments de la mémoire.

Les fonctions dont le nom commençait par « trouver » ont été regroupées dans le même fichier afin d'alléger les fichiers « interpreteur.c » et « disasm.c ».

Il a été ajouté la fonctionnalité permettant de choisir l'adresse de début du segment .text.

Les fichiers de type assembleur et elf ont été regroupés dans le dossier « programmes ». La fonction load a été modifiée pour faire le chargement à partir de ce dossier.

III – Mise en place de programmes de tests

Nous avons créé des fichiers assembleurs pour tester les instructions MIPS. Afin de pouvoir compiler ces fichiers automatiquement, nous avons ajouté une cible au makefile : make prgm. Ces programmes ont permis de repérer plusieurs erreurs, qui ont été corrigées.

Le masque et le mnémonique de certaines instructions de branchement étaient erronés dans le dictionnaire. Ils ont été modifié.

L'instruction SEB n'a pas pu être testée car l'erreur suivante apparaît lors de la compilation des fichiers assembleur qui la contienne : Error: opcode not supported on this processor: mips1 (mips1).

Les autres instructions ont été testé, y compris dans les cas où des erreurs de dépassement ont lieu. Cependant, les tests ont aussi permis de mettre en évidence le fait que la compilation des fichiers assembleur en fichiers elf n'est pas littérale. Ainsi, lors de la compilation de l'instruction DIV, des instructions supplémentaires apparaissent dans le .o afin d'éviter des erreurs d'exécution, par exemple la division par 0. Cela n'empêche pas le fonctionnement de DIV, mais peut poser problème dans d'autres cas.

La compilation des instructions de sauts en particulier est problématique. En effet, un NOP est automatiquement ajouté derrière l'instruction de saut, ce qui empêche les erreurs dues à l'exécution de l'instruction du delay slot. Cet ajout décale les adresses des instructions suivantes, mais comme on lit toujours l'instruction à l'adresse PC + 4 sauf pour la première instruction du programme, l'exécution continue comme prévue. Cependant, cela pose problème si le programme contient d'autres instructions de saut. Le tableau ci-dessous reprend les résultats d'un des tests effectués :

Adresse de l'instruction	Fichier assembleur	Fichier désassemblé
0x3000	J 0x300c	J 0x300c
0x3004	ADDI $t0, $zero, 12	NOP
0x3008	ADDI $t1, $zero, 15	ADDI $t0 $zero 12
0x300c	ADDI $t2, $zero, 17	ADDI $t1 $zero 15
0x3010	ADDI $t3, $zero, 19	ADDI $t2 $zero 17
0x3014	JAL 0x3024	JAL 0x3024
0x3018	ADDI $t4, $zero, 20	ADDI $t3 $zero 19
0x301c	ADDI $t5, $zero, 21	ADDI $t4 $zero 20
0x3020	ADDI $t6, $zero, 22	ADDI $t5 $zero 21
0x3024	ADDI $t7, $zero, 24	ADDI $t6 $zero 22
0x3028	ADDI $a0, $zero, 0x3038	ADDI $t7 $zero 24
0x302c	JALR $a1, $a0	ADDI $a0 $zero 123448
0x3030	ADDI $t8, $zero, 26	JALR $a1 $a0
0x3034	ADDI $t9, $zero, 28	NOP
0x3038	ADDI $s1, $zero, 30	ADDI $t8 $zero 26
0x303c		ADDI $t9 $zero 28
0x3040		ADDI $s1 $zero 30

Le premier NOP décale les instructions suivantes. L'instruction JAL reste à la même adresse, car le NOP est remplacé par l'instruction précédant JAL. Ce n'est pas le cas de l'instruction JALR, qui est décalée de 4.

Le même phénomène se produit lorsque plusieurs instructions de branchements apparaissent dans le même code.

Ne pouvant modifier la compilation des fichiers assembleurs, nous avons testé les quatre instructions de saut et les six instructions de branchements indépendamment les unes des autres.

IV – A propos de la Libc

La gestion de la Libc a pris beaucoup de temps et pas mal d'efforts. En effet, en intégrant la libc le programme entraînait systématiquement une erreur de segmentation lors de la sortie du programme sans que l'on ne comprenne penddant longtemps pourquoi. Il s'agissait en fait d'une

erreur dans l'initialisation des registres : la dernière case mémoire du tableau de registre se retrouvait être la même qu'une des cases mémoire de l'interpréteur suite à une mauvaise définition du nombre de registres. Cela a eu pour conséquence la « pollution » de l'interpréteur et ainsi entrainait une erreur lors de la libération mémoire de celui-ci.

Le programme réalisé met bien en mémoire la table des symboles, les segments de la libc et fait bien la relocation de la libc. Toutefois, faute de temps pour débugger en détail, nous n'avons pas pu vérifier et valider le comportement des programmes utilisant celle-ci.

Les programme, test_printf.o et rodata.libc.o ne fonctionnent pas. Si on regarde le code assembleur de ceux-ci, ils comment tous les deux de la même manière :

Pour rodata.libc.o

```
0:    addiu  sp,sp,-24
4:    afbf0014   sw  ra,20(sp)
```

Pour test_printf.o

```
0:    27bdffe0   addiu  sp,sp,-32
4:    afbf001c   sw  ra,28(sp)
```

On voit que la première instruction met 0xfffffffe0 dans sp, ce qui ne se trouve nul part dans la mémoire (la libc commence à 0xff7fc000 et finit à 0xff7fef24) et donc l'instruction sw ne sera pas exécutée.

Conclusion

Nous avons réussi à réaliser un émulateur Mips qui répond à la majeure partie des exigences du sujet. Toutefois, nous avons vu que la compilation des fichiers assembleurs en fichiers elf pose problème, car le compilateur ajoute des instructions pour sécuriser l'exécution. Pour créer un émulateur le plus proche possible du MIPS, il faudrait modifier le compilateur afin qu'il fasse une traduction littérale des fichiers .s en fichiers .o.

La gestion du temps a été un problème tout au long du projet, et le temps nous a manqué pour finir correctement toutes les fonctionnalités de l'émulateur (notemment la libc). Toutefois, ce travail s'est avéré très intéressant et a été très formateur.