

Projet informatique 2A SICOM : Rapport de livrable N°2

Introduction

Pour le second livrable du projet informatique, il s'agira d'implanter un désassembleur. Du point de vue de l'interpréteur, il ne s'agit que d'ajouter une commande (disasm), mais cela implique de nombreuses autres fonctions. Il faudra lire en mémoire les instructions contenues dans le segment .txt, et désassembleur l'instruction inscrite, c'est à dire, en partant du code binaire de l'instruction, retrouver la mnémonique et les opérandes, ainsi que de les afficher de manière convenable.

1. Implantation

1.1 Fonctionnement du logiciel

Actuellement, le logiciel comprend toutes les commandes du livrable 1 terminées et fonctionnelles, ainsi que la commande disasm. Cette commande à été implantée de la manière suivante :

- Tout d'abord, nous utilisons un dictionnaire qui contient toutes les informations nécessaires au désassemblage. Ce dictionnaire a été construit à l'aide d'un tableur, puis exporté au format texte csv afin de manipuler plus facilement les données, et donc de permettre un gain de temps considérable
- Ensuite, ce dictionnaire est chargé au sein d'un tableau de structure, qui est la suivante :

```
struct dico
{
char id_debug[16];
char mnemo[16];
int size; //Size of instruction in bits (16 or 32)
unsigned int sig;
unsigned int mask;
int nb_op; //Number of operands
char registers_index[20];
char immediate_index[20];
int it;
};
```

On utilisera donc un tableau dico[53] qui contiendra la totalité du dictionnaire. Quelques précisions sur son fonctionnement : Les 2 char* registers_index et immediate_index contiennent des chaînes de caractère sous la forme 26-26:19-16:14-12:7-0 par exemple. Ces index nous permettent d'extraire du code objet les opérandes. Ensuite, l'int it peut valoir 0 si l'instruction peut être dans un bloc IT en changeant de syntaxe si c'est le cas, 1 s'il peut être dans un bloc IT sans changer de syntaxe, et 2 s'il ne peut pas être dans un bloc IT.

- La 3ème étape consiste à lire une instruction dans le code objet et à la décoder à l'aide du dictionnaire. Les instructions codées sur 32 bits commencent toujours soit par 0xF soit par 0xE, on fait la distinction de cette manière.
- Enfin, une fois l'instruction décodée, on affiche la mnémonique et les opérandes dans l'interpréteur.

Pour l'instant, le programme compile mais la fonction `disasm` rencontre encore de nombreuses erreurs. L'instruction souffre encore d'erreurs de segmentation mais le débogage est en bonne voie à l'heure de rendu du livrable. De plus, la gestion des blocs IT n'a pas pu être implémentée.

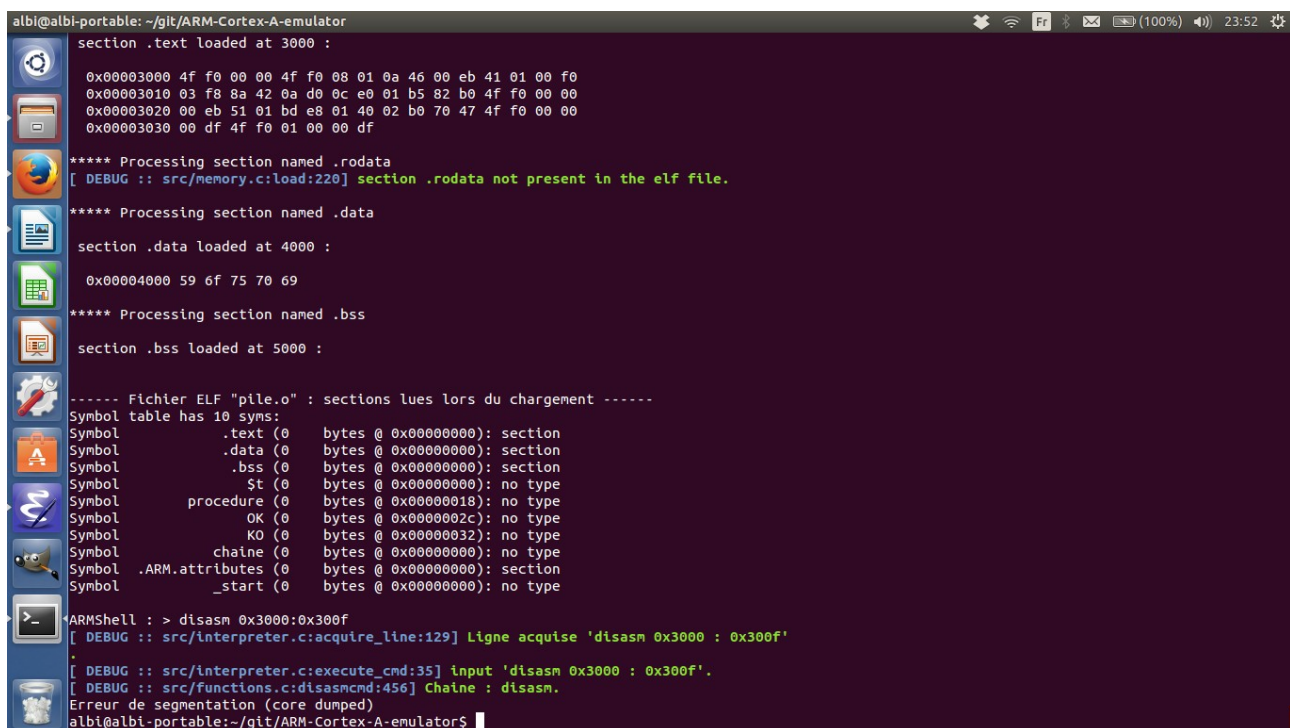
1.2 Tests effectués

Nous avons fait une grosse erreur au cours de ce livrable : nous aurions dû tester chaque fonction pas à pas et faire tourner le compilateur en permanence. Cela n'a pas été fait, et le programme est encore en cours de débogage à la date de rendu de ce livrable. Nous allons adapter nos méthodes de travail pour le prochain livrable.

D'une autre part, tout comme le premier livrable, les nouveaux jeux de tests sont intégrés au livrable dans le dossier "test" et sont au nombre de 6.

1.3 Exemple d'exécution

A l'exécution, le programme se présente sous la forme suivante :



```

albi@albi-portable: ~/git/ARM-Cortex-A-emulator
section .text loaded at 3000 :
0x00003000 4f f0 00 00 4f f0 08 01 0a 46 00 eb 41 01 00 f0
0x00003010 03 f0 8a 42 0a d0 0c e0 01 b5 82 b0 4f f0 00 00
0x00003020 00 eb 51 01 bd e8 01 40 02 b0 70 47 4f f0 00 00
0x00003030 00 df 4f f0 01 00 00 df

**** Processing section named .rodata
[ DEBUG :: src/memory.c:load:220] section .rodata not present in the elf file.

**** Processing section named .data
section .data loaded at 4000 :
0x00004000 59 6f 75 70 69

**** Processing section named .bss
section .bss loaded at 5000 :

----- Fichier ELF "pile.o" : sections lues lors du chargement -----
Symbol table has 10 syms:
Symbol      .text (0)      bytes @ 0x00000000): section
Symbol      .data (0)      bytes @ 0x00000000): section
Symbol      .bss (0)      bytes @ 0x00000000): section
Symbol      $t (0)      bytes @ 0x00000000): no type
Symbol      procedure (0) bytes @ 0x00000018): no type
Symbol      OK (0)      bytes @ 0x0000002c): no type
Symbol      KO (0)      bytes @ 0x00000032): no type
Symbol      chaîne (0)   bytes @ 0x00000000): no type
Symbol      .ARM.attributes (0) bytes @ 0x00000000): section
Symbol      _start (0)   bytes @ 0x00000000): no type

ARMShell : > disasm 0x3000:0x300f
[ DEBUG :: src/interpreter.c:acquire_line:129] Ligne acquise 'disasm 0x3000 : 0x300f'
[ DEBUG :: src/interpreter.c:execute_cmd:35] input 'disasm 0x3000 : 0x300f'.
[ DEBUG :: src/functions.c:disasmcmd:456] Chaîne : disasm.
Erreur de segmentation (core dumped)
albi@albi-portable:~/git/ARM-Cortex-A-emulator$

```

2. Suivi du projet : Problèmes rencontrés

Le principal problème que nous avons rencontré lors du développement de ce second livrable est la prise en main de la nouvelle fonction, `disasm`. En effet, le problème n'a pas été la programmation effective de cette fonction, mais surtout la compréhension de la méthode de codage des instructions en assembleur ARM. Le codage de ces instructions

est très astucieux pour gagner de la place en mémoire, mais leur décodage s'avère assez complexe. De plus, la compréhension de la syntaxe (la documentation ne donne que la syntaxe, mais jamais d'exemple de commande effective) a été assez complexe. Enfin, la syntaxe n'est pas la même suivant que l'on se situe dans un bloc IT ou non, il a donc fallu traiter ce cas en plus. Par la suite, la programmation effective n'a pas posé de problème majeur étant donné qu'elle ne fait pas intervenir de nouvelle notion en termes d'algorithmique.

Nous avons aussi rencontré un problème liés aux outils utilisés. En effet, nous utilisons GitHub afin de collaborer sur le projet et d'assurer un suivi des versions. Cependant, apprendre à utiliser GitHub demande autant (voir plus) de temps que de maîtriser un éditeur de texte performant tel que Emacs ou Vim (que nous utilisons tous les deux : Mickaël utilise Emacs alors que Moctar programme sur Vim). Nous avons fini par rencontrer des erreurs lors de la tentative de fusion des deux branches, et nous avons donc du terminer ce livrable en nous envoyant des bouts de code par mail, ce qui est nettement moins efficace.

Conclusion

Cette fois encore, les objectifs du livrable n'ont pas été atteints. Cependant, les erreurs restant à débbugger sont peu nombreuses, et nous avons tous deux appris à utiliser le débbugger de manière très efficace, ce qui devrait rendre la tâche de finir ce livrable plus aisée. Nous tâcherons surtout de nous améliorer sur l'utilisation des outils à notre disposition pour les prochains livrables, afin d'améliorer notre efficacité.