

Projet informatique 2A SICOM : Rapport de livrable N°3

Introduction

Pour le troisième livrable de ce projet, nous allons nous occuper de l'exécution. Il s'agira d'exécuter les instructions stockées en mémoire dans le segment .txt, de gérer des breakpoints pour le débogage ainsi que l'exécution des instructions pas à pas. Il va donc falloir gérer cette exécution de la même manière que sur un processeur réel, c'est à dire en modifiant les valeurs des registers APSR et PC en suivant l'évolution du programme.

1. Implantation

1.1 Etat du logiciel

Actuellement, le logiciel comprend, comparé au livrable précédent, la commande `disasm` presque entièrement fonctionnelle. Elle ne prend pas encore en compte les POP, PUSH ainsi que les ADD et SUB avec SP, ni l'affichage des étiquettes correctement. En revanche, les blocs IT sont gérés ainsi que tous les encodages possible. En ce qui concerne l'exécution, nous avons choisi de traiter chaque encodage différemment du fait des trop nombreuses variations qu'il y a entre elles. En effet, nous avons besoin pour l'exécution de certains bits de l'instruction qui ne sont pas nécessaires à l'affichage. De plus, traiter le problème de manière "générique" serait également source d'erreurs, car nous pouvons difficilement être sûrs que tous les cas soient correctement traités. La structure qui permet l'exécution est présente, cependant les instructions ne sont pas encore implémentées.

Etant donné le travail effectué depuis le précédent livrable, nous allons traiter aussi bien des solutions techniques particulières apportées au livrable 2 qu'au livrable 3.

En ce qui concerne la gestion des blocs IT, nous utilisons 2 variables, `in_it` et `it_state`. La première est une simple variable de comptage qui compte le nombre d'instructions du bloc IT, est donc inférieure ou égale à 0 hors d'un bloc IT. On la décrémente à chaque instruction dans un bloc IT. La seconde contient les opérandes de l'instruction IT, avec `firstcond` et `mask`. Via un décalage du masque d'un bit vers la gauche, on va pouvoir trouver à quelle condition obéit chaque bloc.

Ensuite, afin de traiter de manière générique les instructions, nous avons deux colonnes dans le dictionnaire, `TreatImm` et `TreatReg`. Ces variables donnent le traitement à appliquer à l'instruction afin de décoder les valeurs des opérandes. Parmi les traitements, nous avons par exemple `ThumbExpandImm` ou `SignExtend`.

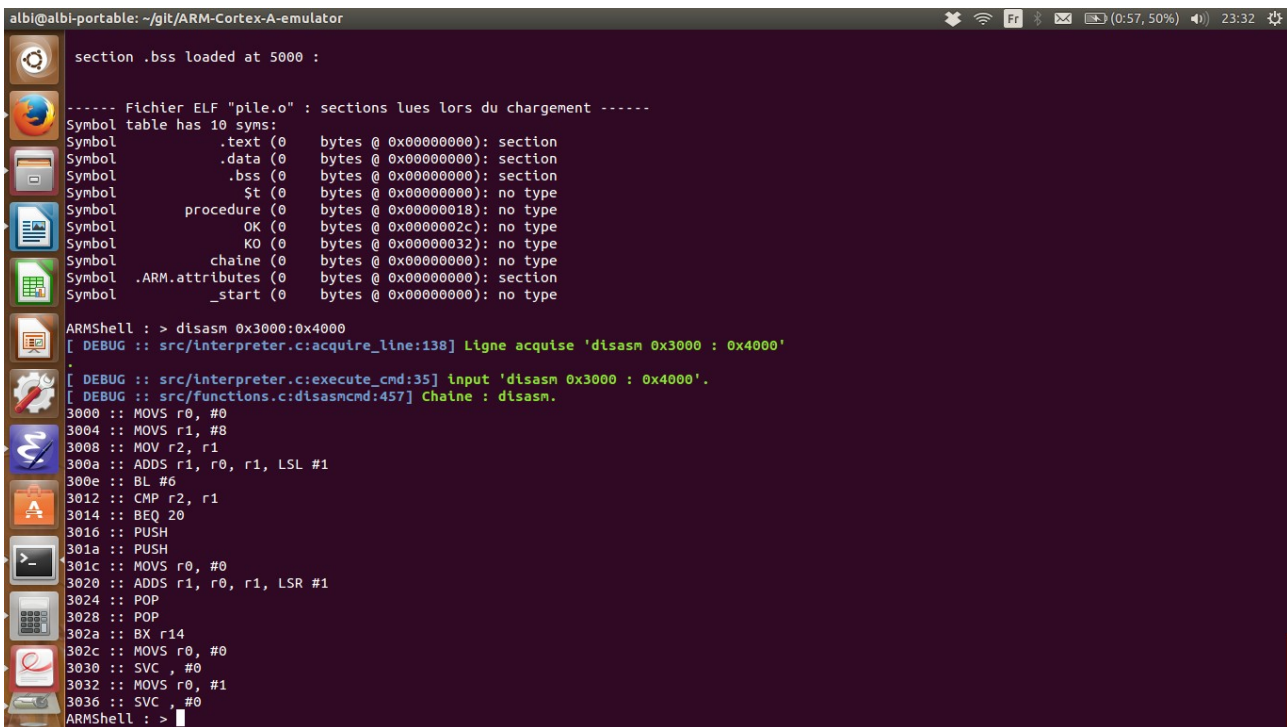
La gestion des breakpoints est très simple : lors du chargement du fichier, nous allouons un tableau d'int de la taille du segment .txt, chaque case correspond donc à une adresse mémoire. Chacune de ces cases prend la valeur 0 ou 1, 1 correspondant à un breakpoint, et 0 à l'absence de breakpoint. Il suffit donc de comparer l'adresse d'exécution et la valeur dans le tableau pour arrêter l'exécution. Lorsque l'on atteint un breakpoint, celui-ci est retiré du tableau afin de pouvoir continuer l'exécution du programme.

1.2 Tests effectués

Pour ce livrable, nous avons appris de nos erreurs et avons testé le programme à chaque nouveau bloc d'instructions. Cela nous a permis un gain de temps précieux, mais cela ne nous a tout de même pas permis de terminer ce livrable à temps

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 .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 0x3000:0x4000
[ DEBUG :: src/interpreter.c:acquire_line:138] Ligne acquise 'disasm 0x3000 : 0x4000'
.
[ DEBUG :: src/interpreter.c:execute_cmd:35] input 'disasm 0x3000 : 0x4000'.
[ DEBUG :: src/functions.c:disasmcmd:457] chaîne : disasm.
3000 :: MOVS r0, #0
3004 :: MOVS r1, #8
3008 :: MOV r2, r1
300a :: ADDS r1, r0, r1, LSL #1
300e :: BL #6
3012 :: CMP r2, r1
3014 :: BEQ 20
3016 :: PUSH
301a :: PUSH
301c :: MOVS r0, #0
3020 :: ADDS r1, r0, r1, LSR #1
3024 :: POP
3028 :: POP
302a :: BX r14
302c :: MOVS r0, #0
3030 :: SVC , #0
3032 :: MOVS r0, #1
3036 :: SVC , #0
ARMShell : >
```

L'exécution ne donne aucun résultat pour l'instant car les fonctions ne sont pas implémentées. Le simple ajout de cette « brique » logicielle permettrait de terminer le livrable.

2. Suivi du projet : Problèmes rencontrés

Nous avons rencontré moins de problème que précédemment lors de ce livrable. Cependant, il a fallu résoudre tous les bugs du second livrable et terminer les désassemblage. Pour cela, il a fallu revoir le fonctionnement quasiment entièrement afin d'implémenter correctement la gestion des blocs IT. Cela a pris un temps considérable étant donné le très grand nombre de manières d'encoder les opérandes dans les instructions. De ce fait, nous n'avons pas pu terminer le 3ème livrable à temps.

Conclusion

Une fois encore, nous n'avons pas pu atteindre les objectifs requis pour ce 3ème livrable. Cependant, il est en bonne voie et ne nécessite que l'ajout des différentes instructions. Ce travail, bien que fastidieux, ne demande que très peu de réflexion étant donné qu'il suffit de suivre les algorithmes donnés dans la documentation constructeur.