Année Universitaire 2020-2021

UFR IM²AG Université Grenoble Alpes

PROG5 Projet logiciel

Simulateur ARM

Groupe 10

Membres Titouan Laurent
Ran Yu
Marin Szczepański

Sommaire

Introduction

Ce compte-rendu présente le simulateur ARM tel qu'il a été complété par le groupe 10 (Marin, Ran et Titouan). Il contient les instructions de compilation et d'utilisation, ainsi que les listes des fonctionnalités proposées, des bugs connus et non résolus au moment du rendu, et les tests écrits.

I. Compilation & lancement

Une fois le code source téléchargé, placez-vous dans le dossier et exécutez les commandes suivantes. Il peut parfois être nécessaire d'effectuer un `chmod u+x configure` lors de la première exécution de ce script.

I.1. Compilation

Les options de compilation peuvent être modifiées en changeant le contenu de l'option CFLAGS. Ici nous indiquons celles que nous utilisons habituellement.
Les options `-Wall` et `-Werror` permettent d'obtenir des erreurs détaillées et obligent à corriger chaque *warning*. L'option `-g` rend le débug du programme possible dans gdb, enfin `-std=gnu11` enlève toute ambiguïté quant au standard C utilisé (la différence de versions de gcc entre nos ordinateurs personnels et celle installée sur Mandelbrot introduisait des erreurs de compilation car ces versions n'utilisaient pas le même standard par défaut).

Pour lancer la compilation, exécuter les commandes suivantes :

```
> ./configure CFLAGS='-Wall -Werror -g -std=gnu11'
> make
```

Si l'une de ces commandes renvoie une erreur, il faut « nettoyer » la configuration et réessayer :

```
> autoreconf
> make distclean
> ./configure CFLAGS='-Wall -Werror -g -std=gnu11'
> make
```

I.2. Lancement du simulateur

Dans une première console :

```
> ./arm_simulator --gdb-port:20000
```

Dans une autre console située dans le même dossier :

```
> gdb-multiarch
> file <fichier ARM compilé>
> target remote localhost:20000
> load
```

Puis `stepi` pour avancer d'instruction en instruction, et `info registers` pour afficher l'état des registres du processeur.

I.3. Lancement des jeux de tests pour *memory* et *registers*

Une fois la compilation effectuée avec `make` :

```
> ./memory_test
> ./registers_test
```

II. Structure du code & des fichiers

Ne sont listés ici que les fichiers que nous avons modifiés lors du projet ou qui revêtent une importance pour les tests :

- ➔ `registers.h/c` est l'implémentation de l'interface *registers*, qui représente les registres du processeur et donne les fonctions de lecture et d'écriture sur ces registres,
- ➔ `memory.h/c` implémente le stockage de la mémoire RAM, et donne les fonctions de lectures et d'écriture sur celle-ci,
- ➔ `arm_instruction.h/c` est la partie du programme qui gère l'exécution des instructions. Elle appelle le *fetch* de la prochaine instruction en mémoire, puis évalue son champ conditionnel et effectue la première partie du décodage, avant d'appeler la fonction d'exécution correspondant à la catégorie de l'instruction courante,
- ➔ `arm_load_store.h/c` termine le décodage et exécute les fonctions d'écriture et de lecture en mémoire,
- ➔ `arm_branch_other.h/c` termine le décodage et exécute les fonctions de branchement et diverses,
- ➔ `arm_data_processing.h/c` termine le décodage et exécute les fonctions de calcul, d'affectation et de comparaison, que l'on nomme « instructions de *data processing* »,
- ➔ `memory_test.c` est un jeu de test fourni, permettant de vérifier si l'implémentation de la mémoire par `memory.c` est fonctionnelle,
- ➔ `registers_test.c` est un fichier de test que nous avons écrit, qui vérifie si l'implémentation des registres par *registers.c* est fonctionnelle,
- ➔ `Examples/ex_*.s` sont les fichiers de code source ARM que nous avons écrits pour tester les fonctionnalités du programme.

III. Fonctionnalités

III.1. Fonctionnalités implémentées

III.1.a. Registres

Les registres sont pleinement implémentés et suivent la spécification jusqu'au bout. En plus des registres communs aux différents modes d'exécution, ceux spécifiques aux modes d'interruption (notamment SPSR) sont correctement accessibles et séparés : chaque mode d'interruption procède son propre SPSR.
De plus, les limitations imposées à chaque mode sont bien présentes : par exemple, il est impossible d'écrire dans les bits [4-0] de CPSR si l'on est en mode USER (ces bits déterminent le mode d'exécution courant).

III.1.b. Mémoire

Il en est de même pour l'implémentation de la mémoire qui permet de gérer des accès (lecture/écriture) de 3 tailles différentes : 8 bits (*byte*), 16 bits (*half*) et 32 bits (*word*). Ces fonctions d'accès intègrent également le contrôle du type d'*endianness* utilisé (*little endian* ou *big endian*), de la détection des adresses non alignées (pour les type *half* et *word*) et ainsi que de celle des adresses hors-limite (gestion de la taille de la mémoire, et donc de la plage d'adresse disponible).

III.1.c. Décodage & exécution des instructions

➔ Décodage des instructions

Pour le décodage des instructions, il a besoin d'utiliser la fonction *arm_fetch* pour obtenir les bits de l'instruction. En utilisant la fonction *inst_cond*, il peut vérifier que c'est la condition de l'instruction est vérifiée, et que l'instruction doit être exécutée. Ensuite, les instructions sont diffusées dans trois parties : les instructions de branchement, les instructions de mémoire et les instructions de *data processing*. Chaque catégorie d'instruction présente des bits en commun, donc nous analysons les bits pour décider quelle fonction d'instruction appeler.

➔ Exécution des instructions

L'instruction est exécutée seulement si sa condition est vérifiée. Les fonctions qui exécutent les instructions renvoient 0 s'il n'y a pas d'erreur, ou retournent un entier positif s'il y a une exception.

➔ Instructions prises en charge

Nous avons implémenté le décodage et l'exécution des instructions suivantes :

- B/BL pour les branchements,
- AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN pour le *data processing*,
- LDR, LDRB, LDRH, STR, STRB, STRH, LDM(1), STM(1) pour les accès mémoire,
- MRS pour l'accès à CPSR, traitée comme instruction « à part ».

III.2. Fonctionnalités non implémentées

Il était prévu, une fois les principales fonctionnalités prises en charge, d'étendre les capacités du simulateur en ajoutant une gestion des exceptions grâce au fichier `arm_exception.c`. Cependant, nous déplorons un manque d'investissement de la part de nos camarades Édouard, Rania et Dac Gia Phúc qui nous a peu ou prou contraint à compléter l'ensemble du projet à trois personnes sur six. C'est la principale raison pour laquelle nous n'avons pas pu aller plus loin que les fonctionnalités de base.

IV. Bugs connus

Bug spécifique aux instructions LDMDA et LDMDB :

Lorsque l'on inclut le registre PC (R15) dans la liste des registres à charger, le simulateur a un comportement inattendu : celui se met à ne plus fournir d'informations permettant le débug (dans `gdb-multiarch`), malgré tout l'exécution du programme continue "normalement" (mais impossible de connaître l'état courant des registres par exemple).

Nous n'avons pas réussi à expliquer l'origine de ce comportement, car les instructions LDMDA et LDMDB n'incluant pas R15 fonctionnent normalement. Il s'agirait peut-être d'un conflit avec la fonction *cont* du fichier fourni `gdb_protocol.c`, qui exploite l'instruction *undefined* d'ARM comme point d'arrêt.

Pour reproduire le bug, un fichier de test commenté `bug_LDM_STM.s` est mis à disposition.

V. Tests effectués

Chaque test que nous avons écrit voit son code source commenté, afin d'indiquer la nature du test ainsi que les résultats attendus.

V.1. Registres

Le fichier `registers_test.c` permet de tester l'implémentation des registres. Il propose des tests les plus exhaustifs possibles, mais tester tous les cas n'est pas réaliste. Le fichier source C est commenté pour permettre de comprendre la teneur des tests.

La création de registres, leur lecture et écriture dans différents modes ainsi que l'accès aux registres spécifiques à un mode (les registres *banked*) sont testés à l'aide de la fonction *assert*. Si le test réussit, le programme l'indique. Sinon, l'exécution s'arrête au *assert* ayant échoué.

V.2. Mémoire

Le fichier `memory_test.c` fourni avec le squelette du projet permet de tester la validité de l'implémentation de la mémoire. Il est suffisamment exhaustif pour que nous n'ayons pas eu à le modifier. Le code source n'est pas commenté, mais il indique clairement lors de l'exécution quels sont les tests effectués.

V.3. Instructions de branchement & MRS

Les instructions de branchement B et BL sont testées dans le fichier `example1.s`. Celui-ci était déjà fourni, et propose un test satisfaisant. Il a été commenté pour rendre le test plus clair.

Le fichier `ex_MRS.s` permet quant à lui de tester l'instruction à part MRS. Comme tous les fichiers de test, les résultats attendus sont en commentaires.

V.4. Instructions de *data processing*

Chacune des 15 instructions de *data processing* possède un fichier qui la teste, de nom ex_XXX.s, où XXX est le nom de l'instruction testée. Ces fichiers sont commentés pour expliquer le résultat attendu (souvent une valeur précise dans un registre).

Ce type de test n'est pas exhaustif mais au vu de la relative simplicité d'implémentation de ces instructions, nous ne pouvions pas nous permettre de créer des jeux de tests complets ; nous aurions alors manqué de temps.

Les fichiers de test pour ces instructions sont : `ex_ADC.s`, `ex_ADD.s`, `ex_AND.s`, `ex_EOR.s`, `ex_RSB.s`, `ex_RSC.s`, `ex_SBC.s`, `ex_SUB.s`, `ex_MVN.s`, `ex_ORR.s`, `ex_BIC.s`, `ex_TST.s`, `ex_TEQ.s`, `ex_CMP.s`, `ex_CMN.s`.

Vous remarquerez l'absence de ex_MOV.s. Cela est dû au simple fait que l'instruction MOV est appelée dans tous les autres tests, et que d'elle dépend le succès des tests. On considère donc qu'elle est testée partout.

V.5. Instructions d'accès à la mémoire

Pour les instructions d'accès mémoire, il faut se référer aux fichiers de test `ex_LDR_STR_LDRB_STRB.s` et `ex_LDRH_STRH.s`. Ceux-ci ne permettent pas de couvrir tous les cas possibles et imaginables, mais au moins toutes les variantes d'adressage, de taille de la donnée accédée ainsi que les fonctionnalités d'adressage post- et pré-indexé.

Pour les instructions LDM(1) et STM(1) le fichier `ex_LDM_STM.s` permet d'en tester les fonctionnalités (avec les 4 différents modes d'incrémentation de l'adresse notamment). Cette suite de tests se limite à la sauvegarde (STM) des 15 premiers registre (R0 à PC inclus) et à la restauration (LDM) des 14 premiers registres seulement (R0 à PC exclu). En effet restaurer le registre PC dans le cas des instruction LDMDB et LDMDA produit un bug (cf. *IV. Bugs connus*), et pour ce qui est des instructions LDMIB et LDMIA cela produit une boucle infinie (comportement normal, mais qui n'aide pas à l'évaluation des tests suivants).

V.6. Évaluation du champ conditionnel d'une instruction

Pour tester l'évaluation des 15 conditions possibles du champ *cond* d'une instruction, nous avons utilisé un unique fichier de test, qui doit être suivi pas à pas par gdb. En commentaire sont indiqués les valeurs attendues dans les registres à la fin de chaque test.

VI. Journal du projet

Le journal du projet, qui représente la répartition des tâches et le calendrier de l'avancement du travail, est fourni avec le présent document sous le nom `Journal du projet.pdf`.