Programmation d'un simulateur de processeur ARM

THIVOLLE Dorian, GABERT Etienne, RUSSO Lilian,
DUCROS Arthur, CORBILLE Clément, LOREK Dorian

Projet PROG5 - IM2AG Licence Informatique 3° année

Table des matières

1 Mode d'emploi

Le code fournit dans le dépôt git est déjà compilé.
Cependant, voici les étapes à suivre pour setup le Makefile par rapport à votre machine :

1. aller dans le dossier `./arm_simulator-1.4`
2. faire `./configure CFLAGS='-Wall -Werror -g'`
3. faire `make`
4. exécuter en faisant `./arm_simulator --trace-memory --trace-register` pour lancer le simulateur
5. sur un autre terminal, dans le même dossier, lancer `gdb-multiarch`
6. faire `file tests/test1` par exemple pour lancer le test1
7. cibler l'adresse : `target remote localhost:<port gdb donné>`
8. puis faire `load`, et enchaîner avec des stepi pour exécuter les instructions pas-à-pas

Note : Tous les tests sont dans le dossier `./arm_simulator-1.4/tests`. Pour les compiler, il faut faire la commande `make` dans ce dossier. Sauf si vous modifiez le Makefile dans ce dossier, seul le fichier `test.s`, est pris en compte lors de la compilation.
Toutes les étapes de compilation et d'exécution détaillées sont trouvables ici :
`https://github.com/NoxFly/ARM-processor-simulator#execute-the-code`

2 Principales fonctions

— Simulation et interaction avec la mémoire : memory.c/h
— Simulation et interaction avec les registres processeurs : register.c/h
— Lecture et lancement des exécutions des instructions : arm_instruction.c/h
— Implémentation des instructions de branchement : arm_branch_other.c/h
— Implémentation des instructions de traitement de données et calculs : arm_data_processing.c/h
— Implémentation des instructions d'accès à la mémoire : arm_load_store.c/h

3 Liste des fonctionnalités

3.1 Implémentées

Instruction utilisable :
— AND : effectue un " ET logique " entre deux opérandes
— EOR : effectue un " OU exclusif logique " entre deux opérandes
— SUB : effectue une soustraction entre deux opérandes placées en argument
— RSB : effectue une soustraction inversée entre deux opérandes placées en argument
— ADD : effectue une addition entre deux opérandes placées en argument
— ADC : effectue une addition en ajoutant la valeur du carry flag
— SBC : effectue une soustraction en enlevant le complément de la valeur du carry flag
— RSC : effectue une soustraction inversée en enlevant le complément de la valeur du carry flag
— TST : test le " ET logique " entre deux opérandes et met à jour les flags ZNCV
— TEQ : test l'équivalence entre deux opérandes (<=>) et met à jour les flags ZNCV
— CMP : différence entre deux opérandes placées en argument et met à jour les flags ZNCV
— CMN : somme entre deux opérandes et met à jour les flags ZNCV
— ORR : effectue un "OU logique " entre deux opérandes
— MOV : déplace une valeur de 32 bits dans un registre
— BIC : effectue un "ET logique" entre un opérande et le complément d'un autre opérande
— MVN : déplace la négation d'une valeur de 32 bit dans un registre
— LDR : charge une valeur de 32 bits depuis une adresse mémoire (vive) dans un registre
— LDRB : charge une valeur de 8 bits depuis un adresse mémoire dans un registre
— LDM : charge plusieurs valeurs depuis la mémoire
— STR : stock une valeur de 32 bits depuis un registre dans la mémoire (vive)
— SRB : stock une valeur de 8 bits depuis un registre dans la mémoire (vive)

— STM : stock plusieurs registre dans la mémoire.
— B : fait un saut à l'adresse mise en argument
— BL : appelle un sous-programme et place l'adresse de retour dans le registre de liaison et fixe le PC vers l'adresse du sous-programme
— MRS : transfert la valeur de CPSR ou SPSR du mode actuel dans un registre
— MSR : transfert la valeur d'un registre ou un nombre dans le registre CPSR ou SPSR du mode actuel

4 Tests effectués

test_data_processing_1 :
Test de LSL, des instructions de base (MOV, ADD), du carry flag après une opération qui doit mettre C à 1, de la condition CS quand C vaut 1. C est bien mis à 1 et l'instruction avec la condition CS est bien exécutée ce qui est conforme au résultat attendu. Les autres instructions sont aussi bien exécutées.

test_data_processing_2 :
Test du overflow flag après une opération qui doit mettre V à 1, de la condition VS quand V vaut 1. V est bien mis à 1 et l'instruction avec la condition VS est bien exécutée ce qui est conforme au résultat attendu.

test_data_processing_3 :
Test de la condition CS quand C vaut 0. L'instruction avec la condition CS n'est pas exécutée ce qui est conforme au résultat attendu.

test_data_processing_4 :
Test de l'instruction CMP, de la condition LT et de l'instruction MVN. L'instruction MVN avec la condition LT est bien exécutée ce qui est conforme au résultat attendu.

test_data_processing_5 :
Test de l'instruction ADC, de l'instruction CMN, de l'instruction EOR. Les trois instructions sont bien exécutées. CMN met bien le flag C à 1 et ADC prend bien en compte la valeur du flag C. EOR effectue bien l'opération logique du ou exclusif.

test_ldr_str :
Test des fonctions LDR et STR, vérification de leur bon comportement via les traces : observation des registres et adresses chargées puis stockées. Les instructions fonctionnent et produisent les résultats attendus. Un bug de lecture de registre est à noter mais il ne modifie pas les résultats finaux.

test_ldmib_stmia :
Test des fonctions LDR et STR, vérification de leur bon comportement via les traces : observation des registres et des adresses chargées puis stockées.

test_ldrh_strh :
Test des fonctions LDRH et STRH. Test échoué, ces fonctions ne fonctionnent pas encore.

5 Bugs non résolus

Lors de l'exécution d'une instruction il y a parfois des registres lus sans raison apparente mais ça ne change pas le comportement de l'instruction. Adresse de sauvegarde de LDRH et de chargement de STRH apparemment mal calculée par notre ARM. Le problème vient très probablement de notre code mais nous ne l'avons pas trouvée à temps.

6 Répartition des tâches

Dès le premier jour, nous avons mis en place un tableau d'organisation via l'onglet Project sur la page Github. Nous avons, ensemble, listé toutes les tâches à faire, et nous les avons ajoutées dans le tableau. Cet onglet nous a permis de visualiser tout notre travail : Voir quelles tâches sont

— planifiées (TODO),
— en cours de réalisation (IN PROGRESS),
— ou terminées (DONE),
— et quel membre du groupe est assigné à quelle tâche.

L'équipe s'est divisée en trois sous-groupes :
— Dorian L. et Clément,
— Lilian et Arthur,
— Dorian T et Etienne

Pour éviter les problèmes de conflit nous avons décidé de créer une branche git par fichier à compléter et d'y assigner un groupe par branche. Cela nous a permis de travailler en pair programming et donc de disposer des avantages de celui-ci (prise de recul, différentes approches, compétences complémentaires...).

Assez naturellement, Dorian T. a pris le rôle de lead developper dans l'équipe. Nous avons décidé de mettre en place une convention de travail/développement ainsi qu'une répartition des tâches équitables. Cela nous a permis de créer une bonne dynamique de groupe, notamment dans la communication entre les binômes ou lors des réunions.

Intuitivement, nous nous sommes organisés de manière similaire à la méthode agile scrum. En effet, chaque matin, nous nous réunissions afin de décider des missions quotidiennes et d'assigner les binômes au travail à faire. De plus, chaque semaine étant l'équivalent d'un sprint, nous avions aussi un objectif à atteindre à l'issu du sprint. Cela a pour avantage de nous permettre de garder du recul sur les priorités des tâches à accomplir.

Dès que les tâches d'un fichier étaient terminées, nous effectuions un pull request sur la branche dev, où nous testions toutes les fonctionnalités regroupées. Si le programme fonctionnait correctement, nous faisions un pull request sur la branche master, considérée comme branche de production.

7 Journal

Ce journal reflète l'avancée du projet et les soucis rencontrés. Les bugs trouvés et réglés n'ont pas été notés dans le journal puisque c'est une étape qui s'est effectuée chaque jour.

Jeudi 17 Décembre
Le groupe s'est mis d'accord pour que chacun prenne son après-midi pour prendre connaissance du projet.

Vendredi 18 Décembre
Problèmes rencontrés sur les fichiers partagés sur le git (chemins absolus et configurations propre au PC mais partagés sur le dépôt git notamment pour le Makefile).
Mise en accord sur la disposition du git (gestion des branches pour limiter les conflits et simplifier le dépôt) et tous les autres choix importants pour que le groupe fonctionne bien.

Lundi 4 Janvier
Réalisation des fichiers registers.c et memory.c.

Mardi 5 Janvier
Réalisations des fichiers arm_instruction et arm_data_processing.

Mercredi 6 Janvier
Début de réalisation des fichiers arm_branch_other et arm_load_store.

Jeudi 7 Janvier
Audit de code : ce qui l'en est ressorti c'est une bonne progression
mais une possibilité de factoriser le code pour qu'il soit plus lisible.
Factorisation de arm_data_processing et finition de arm_branch_other.

Vendredi 8 Janvier
Factorisation de registers.c, mise en commun de tous les fichiers et test global.

Lundi 11 Janvier
Factorisation et Complétion de arm_load_store.c + premiers tests. Avancement du développement des exceptions et interruptions.

Mardi 12 Janvier
Finition des exceptions et des interruptions. Adaptation de arm_load_store par rapport à ces derniers.

Mercredi 13 Janvier
Finition du code, vérification, et jeux de tests.

Jeudi 14 Janvier
Finitions de la documentation, vérification du code et suite des jeux de tests. Projet rendu à 12h.

Vendredi 15 Janvier
Jour de la soutenance de Projet.

8 Conclusion

Pour finir, ce projet nous a permis de développer des capacités à travailler en petite équipe ; sur un sujet de développement logiciel. Nous avons pu expérimenter, la méthode agile scrum et nous avons dû rechercher des informations dans la documentation ainsi que se concerter pour parvenir à nos fins.

9 Ressources

Le code source est disponible à cette adresse : `https://github.com/NoxFly/ARM-processor-simulator`.