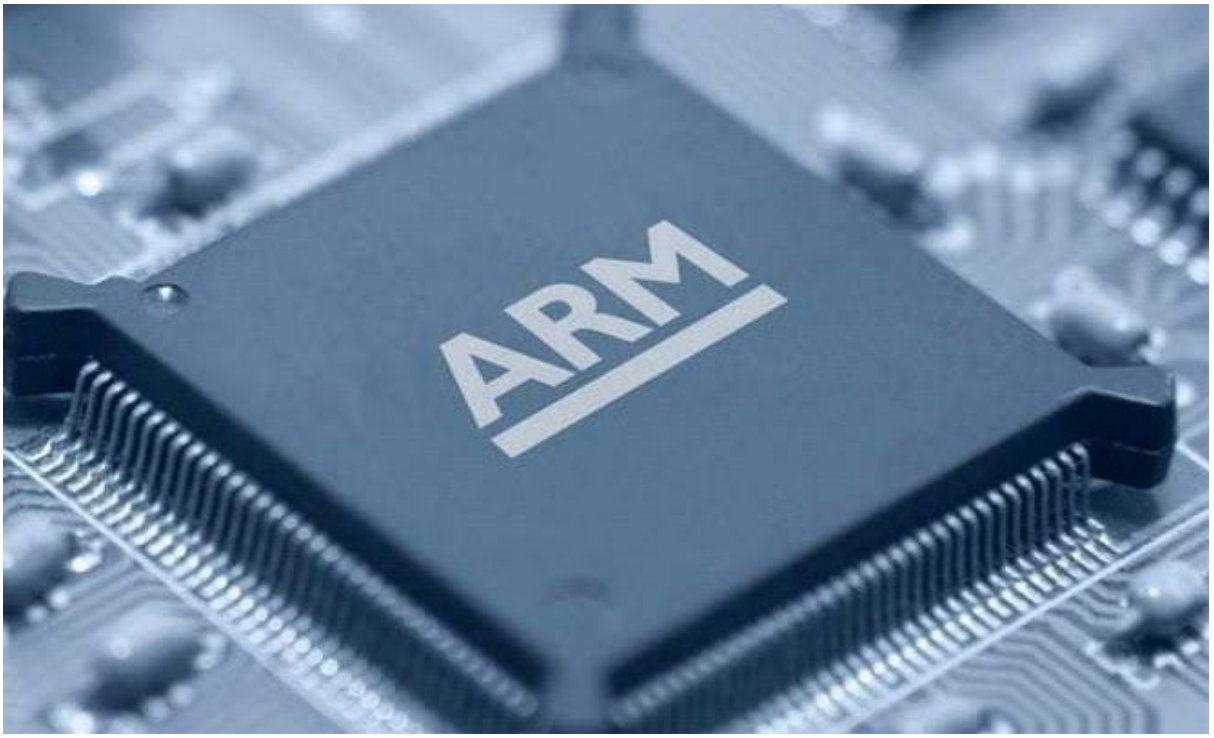
Hugo FETRE - Nicolas GIRARDIER - Vincent BONATO - Yanis MIOLLANY - Mathis VILLARD - Rémi SAMSON
Groupe 12

Compte rendu de projet

Simulateur de carte ARM



1. Déroulement du projet

a. Organisation du travail
b. Ce qui s'est réellement passé
c. Difficultés rencontrées
d. Solutions trouvées

2. Liste des fonctionnalités

a. Les fonctionnalités implémentées
b. Les fonctionnalités manquantes

3. Les tests

Liste des tests effectués pour s'assurer du bon développement.

4. Bogues non résolus

Liste des bogues auxquels nous n'avons pas trouvé de solution.

5. Mode d'emploi du projet

Comment compiler et lancer les programmes développés

6. Journal décrivant la progression du travail

1. Déroulement du projet

a. Organisation du travail

Nous voulions initialement développer individuellement sur chacune des parties à traiter. L'idée était de faire une visioconférence-conférence chaque matin pour faire le point sur ce qu'il y a à faire dans la journée et se concerter sur l'avancement du projet. Nous avons donc prévu d'utiliser Git pour gérer la fusion et les mises à jour des versions de chacun.

b. Ce qui s'est réellement passé

Nous avons eu beaucoup de mal à comprendre l'architecture du projet ainsi que le rôle de chaque fichier. De plus, nous avons eu des difficultés à trouver intuitivement ce qu'il fallait coder dans chaque partie. Le lien entre la finalité du programme et le développement de celui-ci a été très dur à comprendre.

Nous avons constaté que chaque membre de l'équipe a eu du mal à s'imprégner du fonctionnement du projet et des objectifs à remplir. De plus, le développement d'une partie nécessite d'avoir terminé la précédente, ce qui rend difficile la répartition des tâches. Nous avons donc conclu que se répartir les tâches et travailler individuellement sur chacune des parties à développer était peu productif.

Nous avons donc choisi de travailler tous ensembles sur chaque partie. Nous avons travaillé en suivant des sessions de visioconférence via Discord et de liveshare sur Visual Studio Code. Nous avons tout de même continué d'utiliser Git puisque même si nous réfléchissions ensemble en direct, nous codions chacun une version différente de la partie à faire. Le membre qui avait le code de meilleure qualité effectuait un push de sa version, que les autres récupéraient pour continuer sur cette base. Cette méthode s'est avérée plus productive. En effet, en vue de la difficulté que nous avons eu pour comprendre le fonctionnement du projet, six cerveaux à plein régime sur une seule partie étaient plus efficaces qu'un étudiant par partie, complètement perdu et démotivé. En arrivant à la fin du projet, nous étions suffisamment à l'aise pour se diviser en plusieurs équipes et répartir les tâches par équipes.

c. Difficultés rencontrées

- Difficultés à mettre en place le projet : il y avait des manipulations supplémentaires à faire, telles que passer en gdb-multiarch, installer un certain nombres de packages sur ubuntu.
- Prise en main et utilisation du manuel/guide de référence d'architecture ARM. Cette documentation, bien que très remplie, s'est avérée difficile à utiliser étant donné toutes les informations qui s'y trouvent.

d. Solutions trouvées

- L'aide des professeurs présents nous a permis de résoudre certains problèmes ou de nous débloquer de certaines situations.
- Utiliser gdb-multiarch et ajouter des packages ubuntu (sudo apt-get update -y / sudo apt-get install -y gdb-multiarch**).**

2. Liste des fonctionnalités

a. Les fonctionnalités implémentées

Voici une liste des fonctionnalités implémentées dans chaque fichier et leur rôle:

- **memory.c/h :**
  - ✓ consultation de la mémoire (procédure memory_read_byte)
  - ✓ modification de la mémoire (procédure memory_write_byte)

- **registers.c/h :**
  - ✓ consultation des différents registres (procédure read_register)
  - ✓ modification des différents registres (procédure write_register)

- **arm_instruction.c/h :**
  - ✓ arm_execute_instruction() : Constitue la première partie de la simulation d'un cycle d'exécution d'une instruction. Cette fonction lit à chaque passage la prochaine instruction à effectuer, tout en incrémentant le compteur ordinal et en interprétant le champ d'exécution conditionnelle. Puis cette fonction va sélectionner la classe d'instructions et appeler la fonction correspondante permettant d'exécuter cette instruction.

- **arm_data_processing.c/h :**

  Ce fichier permet de gérer toutes les instructions de type data_processing.

  - **arm_data_processing_shift () :**
    - ✓ Prend en paramètre le cœur arm et l'instruction à effectuer. Calcule le shift correspondant grâce à une fonction shifter_value_calculator. Par la suite, elle va prendre en compte toutes les opérations possibles ainsi que leurs extensions.

  - **arm_data_processing_immediate_msr() :**
    - ✓ Prend en paramètre le cœur arm et l'instruction à effectuer. Calcule le shift correspondant grâce à la fonction *ror()* et les valeurs de l'instructions. Par la suite, elle va prendre en compte toutes les opérations possibles ainsi que leurs extensions.

- **cond_shift_calculator.c/h :**

  - **cond_verifier() :**

    - ✓ Vérifie que les valeurs des indicateurs Z, C, V et N concordent avec le condition code (bits 28 à 31 de l'instruction)

  - **Fonctions de récupération des indicateurs Z C V N :**

    - ✓ Lit le registre CPSR et retourne le bit correspondant à l'indicateur recherché

  - **shifter_value_calculator()** :

    - ✓ Calcule la valeur du *shifter_operand* et du *shifter_carry_out* depuis l'instruction et depuis le registre.

- **arm_load_store.c/h :**

  Accès à la mémoire : opérations load/store entre un registre et la mémoire avec différents niveaux de granularité (mot, demi-mot, byte), transferts multiples entre un ensemble de registres et la mémoire.

  - **number_of_set_bit_in() :**
    - ✓ Compte le nombre de bit à 1 dans un champ de bit.
  - **arm_load_store() :**
    - ✓ load and store word / unsigned byte
    - ✓ load and store half/double word
    - ✓ miscellaneous load/store
  - **arm_load_store_multiple() :**
    - ✓ load and store multiple

  Nous avons implémenté 3 modes d'adressage différents grâce à différentes fonctions :

  - adressage 2 : load and store word/unsigned byte
  - adressage 3 : miscellaneous load/store
  - adressage 4: load/store multiple

  Voici les différentes fonctions que nous avons créés puis implémentées afin de calculer les adresses pour chaque mode d'adressage différent :

  - **adress_calculator() :**
    - ✓ Calcul de l'adresse à partir de l'instruction et du registre pour le stockage ou le chargement d'un mot ou d'un octet non signé dans la mémoire.
  - **misc_adress_calculator() :**
    - ✓ Calcul de l'adresse à partir de l'instruction et du registre pour le stockage ou le chargement d'un demi-mot, d'un octet signé ou d'un double mot dans la mémoire.
  - **mult_adress_calculator() :**
    - ✓ Calcule et modifie les valeurs des paramètres *start_address* et *end_address* grâce aux valeurs de l'instruction et du registre. Cette fonction permet de charger ou stocker un ensemble de registre dans la mémoire ou inversement.

- **arm_branch_other.c/h :**
    - **arm_branch() :**
        - ✓ Écrit dans le registre PC (R15), la *target_address* calculé depuis le registre (transformation de 24 bits non signés en 32 bits signés). Dans le cas d'une instruction BL (bit 24 à 1) on enregistre la valeur du registre PC dans le registre link (R14).

- **instructions ARM :**
    - traitement de données (data processing) : opérations arithmétiques et logiques, transferts entre registres ;
    - rupture de séquence : branchement avec sauvegarde éventuelle de l'adresse de retour ;
    - autres instructions : instructions ne rentrant pas dans les catégories précédentes, par exemple transfert entre le registre d'état et un registre généraliste.

b. Les fonctionnalités manquantes

Dû au fait que nous avions rencontré des difficultés à comprendre le projet, nous avons mis plus de temps que prévu pour développer chaque partie. Par conséquent, certaines fonctionnalités n'ont pas pu être implémentées à temps. Voici une liste des fonctionnalités que nous n'avons pas réussi à développer, par manque de temps.

- ruptures de séquence
- autres instructions (arm_exception.c / .h)

3. Les tests

Nous avons vérifié l'efficacité de notre code en utilisant quelques programmes tests, notamment en utilisant ceux déjà fournis. Par la suite, nous avons utilisé des méthodes plus rapides permettant de vérifier l'intégrité des données et le bon déroulement du simulateur grâce aux fonctions *printf* et fprintf. L'utilisation de gdb s'est avérée utile pour exécuter notre programme en mode débug et ainsi suivre les valeurs des différentes variables du programme. Nous avons ainsi pu repérer les erreurs et les corriger sans trop de difficultés. Voici les fonctionnalités testées et leurs tests :

a. Test de registers.c

Pour ce fichier de test, nous avons repris le principe de memory_test.c en modifiant légèrement les fonctions déjà présentes dans ce dernier (print_test(), compare() et compare_with_sim()).

- Test de write_register() et read_register() nous avons donc pu vérifier que ces 2 fonctions fonctionnaient correctement.
- Test de write_cpsr() et read_cpsr() nous pouvons également voir que ces fonctions sont correctes.
- Test de write_usr_register() et read_usr_register().
- Test de write_spsr() et read_spsr(). Pour tester ces 2 fonctions, nous avons dû changer le mode pour qu'il soit différent de USR et SYS car ces deux modes ne possèdent pas de registre SPSR. Le test de ces fonctions ont permis de déceler une erreur dans celle-ci. L'erreur venait de la fonction read_spsr(), en effet SPSR est le 17 ème registre de chaque mode, mais ce n'est pas du tout le registre numéro 17 par exemple pour le mode UND le registre SPSR porte le numéro 25. Donc au lieu de récupérer le registre 17 nous récupérons le registre se trouvant à la colonne 17 de la matrice matriceMode.

b. Test d’arm_data_processing.c

Ce fichier nous a posé quelques problèmes. En effet, nous ne savons pas pourquoi mais arm_data_processing_test.c n’arrive pas à utiliser les fonctions se trouvant dans arm_data_processing.c. Nous avons une erreur de type undefined reference. Nous avons donc décidé de réécrire les fonctions que nous voulions tester sur le fichier de test.

Nous n'avons donc fait qu’un seul test dans ce fichier à cause du problème rencontré ci-dessus.

- Test de la fonction carryFrom() cette fonction permet de vérifier s’il n’y a pas de débordement lors d’une opération entre deux uint32_t.
  Les tests réalisés sur cette fonction ont permis de trouver plusieurs erreurs.
  - Tout d’abord nous récupérions le mauvais bit nous faisions return get_bit(res >> 32, 1) or nous voulons récupérer le 32ème bit et pas le 33ème nous avons donc changé pour mettre return get_bit(res >> 32, 0).
  - Ensuite il y avait également une seconde erreur nous utilisions des uint32_t dans la fonction or la fonction demande un uint64_t pour aller chercher le bit 32 nous avons donc dû faire des casts sur tous les opérandes pour que la fonction soit correcte.

4. Les bogues non résolus

- Boucle infinie lors de l’instruction stepi
- Impossibilité de récupérer les fonctions de arm_data_processing.c dans arm_data_processing_test.c

5. Mode d'emploi du projet

Veuillez trouver les procédures à suivre pour lancer le simulateur et lui faire parvenir des instructions.

1. Ouvrez 2 terminaux Linux.
2. Dans l'un des deux, compilez le programme : *make*.
3. Dans un terminal, lancez "*gdb" ou "gdb-multiarch"* (ce sera le terminal A)
4. Dans l'autre, lancez "*./arm_simulator" (ou "gdb ./arm_simulator" si vous souhaitez déboguer le simulateur)* (ce sera le terminal B). Dans ce terminal, 2 numéros de port (gdb et irq) apparaîtront.
5. Dans le terminal A, vous pouvez définir un programme à charger en suivant les commandes : *file répertoire/programme* (par exemple, pour lancer Exemple1 situé dans le répertoire exemples : "*file exemples/Exemple1*".
6. Ensuite connectez le terminal A au serveur local du simulateur: *target remote localhost:<port gdb indiqué par le simulateur (terminal B)>*
7. Toujours dans le terminal A, chargez votre programme dans le simulateur avec la commande : *load*.
8. Puis utilisez *stepi* pour commencer la lecture des instructions du fichier chargé.

6. Journal décrivant la progression du travail

Avant les vacances: Lecture du sujet et prise en main de la doc.

17/12/2020 : Création du GitHub.

4/01/2021 : Prise en main du projet (compréhension de la structure).

5/01/2021 : Mise en place des fonctions de memory.c

6/01/2021 : Finalisation de memory.c et début de registre.c

7/01/2021 : Finalisation de registre.c et amélioration du code déjà effectué suite à la visioconférence avec M. Huard.

8/01/2021 : Début des fonctions de arm_instruction.c

11/01/2021 : Finalisation de arm_instruction.c et début de arm_data_processing

12/01/2021 : Suite des instructions dans arm_data_processing et début de code dans arm_load_store. Et développement de arm_branch_other.c

13/01/2021 : Finalisation de arm_load_store pour les trois types d'adressage demandés (word/unsigned byte, miscellaneous, multiple) et débogage du projet.

14/01/2021 : Réalisation de jeux de test pour register.c et débuts de tests sur arm_data_processing (tests sur la fonction carryFrom). Finalisation du compte rendu et débogage du projet.