PROG 5

Projet Logiciel 2020-2021

Réalisation d'un simulateur ARM

BIHAN Baptiste,
CHASSAGNOL Thibault,
GHERRAZ Salah,
LEONARD Valentin,
MORAND Lucas,
REIG Julien

Groupe 2
IM²AG - Université Grenoble Alpes
2020 - 2021

Tables des matières :

I) Structure du code développé 3
II) Fonctionnalités 4
A) Fonctionnalités réalisées 4
B) Fonctionnalités manquantes 5
III) Bogues connus 5
IV) Tests 5
V) Journal de bord 6
VI) Répartition du travail 7
VII) Mode d'emploi 7
A) Compilation 7
1. paquet requis: 7
2. commande: 8
B) lancement du programme 8

I) Structure du code développé

Notre code est structuré autour de deux fichiers pivot : arm_core.c et arm_instructions.c. Le fichier arm_core.c sert à créer et à interagir avec la mémoire et les registres au travers de sa structure de données et des différentes fonctions qui le composent.

Nous avons pu constater dans la documentation ARM qu'il existe 38 registres crées par la présence de différent modes, nous avons décidé de gérer les registres avec un tableau d'entier 32 bits de 38 éléments et nous avons écrit une fonction getRegistre qui en fonction du mode courant et du registre souhaité allant de 1 à 17 nous renvoie l'indice correspondant dans le tableau

Dans memory.c nous avons décidé de travailler sur une liste d'entier 32 bits. Nous avons fait ce choix car un mot fait 4 octets et que les bus sont généralement de 32 bits, nous pensions donc plus simples de travailler avec des entiers de 32 bits.
Cependant, notre méthode utilise beaucoup de calculs d'adresse. Par la suite nous nous sommes rendus compte que notre façon de faire n'était pas optimale. Nous aurions du utiliser une liste d'octet qui aurait permis de supprimer tous les calculs d'adresses.

Le fichier arm_instructions quant à lui sert principalement à traiter les différentes instructions qui lui sont envoyées en appelant la fonction adéquat qui nous fera aller dans un autre fichier. les fichiers appelés étant les suivants :

- arm_branch_others.c
- arm_data_processing.c
- arm_load_store.c
- arm_exceptions.c

Ces fichiers utilisent tous la fonction condition présente dans arm_instructions.c cette fonction sert notamment à vérifier si la condition de l'instruction est respectée.

Le squelette fourni à été modifié, notamment dans le fichier arm_data_processing où nous nous sommes permis de supprimer la deuxième fonction arm_data_processing_immediate_msr pour mettre son contenu dans la fonction arm_data_processing_shift. Le traitement des deux fonctions étant quasiment identique, cela nous semblait être la meilleure solution. De plus nous avons écrit deux fonctions supplémentaires : data_processing_operand et data_processing_immediate_operand. Ces deux fonctions servent à traiter les différents cas d'une instruction data processing. Afin de déterminer laquelle de ces deux fonctions appeler nous utilisons la variable operand_type. Cette variable est un pointeur de fonction et nous permet de choisir laquelle appeler.

Pour rendre générique le code, nous avons écrit une fonction par opération logique (et bit à bit, ou exclusif). Ces dernières sont passées en paramètre de la variable operand_type afin de déterminer laquelle exécuter.

II) Fonctionnalités

A) Fonctionnalités réalisées

- Gestion des registres
- Gestion de la mémoire
- Gestion des instructions de branchement
    - B (Branchement vers une adresse)
    - BL (Branchement avec sauvegarde d'une adresse de retour)
- Gestion des instructions de traitement de données
    - ADC (Addition avec retenue)
    - ADD (Addition sans retenue)
    - AND (Et bit à bit)
    - BIC (Bit clear)
    - CMN (Comparaison négative)
    - CMP (Comparaison)
    - EOR (Ou Exclusif bit à bit)
    - MOV (Affectation)
    - MVN (Affectation inverse)
    - ORR (Ou bit à bit)
    - RSB (Soustraction inversée sans retenue)
    - RSC (Soustraction inversée avec retenue)
    - SBC (Soustraction avec retenue)
    - SUB (Soustraction sans retenue)
    - TEQ (Ou Exclusif bit à bit avec mise à jour des flags ZNCV)
    - TST (Et bit à bit avec mise à jour des flags ZNCV)
- Gestion des instructions d'accès à la mémoire
    - LDM(1) (Lit une liste de valeurs de 32 bits depuis une adresse donnée et la stocke dans une liste de registres)
    - LDR (Lit une valeur de 32 bits en mémoire et la stocke dans un registre)
    - LRDH (Lit une valeur de 16 bits en mémoire, l'étend à 32 bits en ajoutant des 0, et la stocke dans un registre)
    - LRDB (Lit une valeur de 8 bits en mémoire, l'étend à 32 bits en ajoutant des 0, et la stocke dans un registre)
    - LDRSH (Lit une valeur de 16 bits en mémoire, l'étend à 32 bits en ajoutant le bit de signe, et la stocke dans un registre)
    - LDRSB (Lit une valeur de 8 bits en mémoire, l'étend à 32 bits en ajoutant le bit de signe, et la stocke dans un registre)
    - STM(1) (Écrit une liste de valeurs de 32 bits depuis une liste de registres et la stocke dans la mémoire à partir d'une adresse donnée)
    - STR (Écrit une valeur de 32 bits en mémoire depuis un registre)
    - STRH (Écrit une valeur de 16 bits en mémoire depuis un registre)
    - STRB (Écrit une valeur de 8 bits en mémoire depuis un registre)
- Gestion des différents modes d'exécution du processeur
    - USR (User)
    - SYS (System)
    - SVC (Supervisor)
    - UND (Undefined)

    - ABT (Abort)
    - IRQ (Interrupt)
    - FIQ (Fast interrupt)
- Gestion des exceptions
    - RESET (SVC)
    - UNDEFINED_INSTRUCTION (UND)
    - SOFTWARE_INTERRUPT (SVC)
    - PREFETCH_ABORT (ABT)
    - DATA_ABORT (ABT)
    - INTERRUPT (IRQ)
    - FAST_INTERRUPT (FIQ)

B) Fonctionnalités manquantes

Toutes les fonctionnalités demandées dans le sujet ont été réalisées.

III) Bogues connus

Durant la phase de test, nous nous sommes rendus compte qu'il y avait quelques bogues dans notre programme. Nous avons réussi à les corriger, par conséquent tous les bogues connus ont été résolus.

IV) Tests

```
1   .global main
2   .text
3   main:                   Flag de début de programme
4       ldr r2,=donnees     Lit la valeur à l'adresse donnees et la stocke dans r2
5       mov r1,#0           Déplace la valeur immédiate 0 dans r1
6   loop:
7       ldrb r0,[r2, r1]    Stocke dans r0 la valeur lue à l'adresse r2 + r1
8       teq r0,#0xF0        Effectue la comparaison r0 = 0xF0 et met à jour les flags NZCV en conséquence
9       add r1,r1,#1        Ajoute 1 à la valeur de r1
10      bne loop            Effectue un branchement si r0 = 0xF0, soit le bit Z du registre CPSR est à 0
11      swi 0x123456        Instruction de fin de programme
12  .data
13  donnees:
14      .word 0x12345678
15      .word 0x9ABCDEF0
```

Ce programme lit les valeurs de la section .data octet par octet, et s'arrête lorsqu'il lit la valeur 0xF0.

L'objectif de ce test est de tester les branchements conditionnels, les lectures depuis la mémoire et la modification des ZNCV.
Pour lancer ce test et les autres tests contenus dans le dossier Examples/ , il faut suivre la démarche du mode d'emploi.
Trace commentée du test:

```
main:
Cycle 1, Register read, PC_SVC, val: 00000024                       Lecture de pc pour récupérer l'instruction courante
Cycle 1, Mem read (4 bytes, fetch) addr: 00000020, val: E59F2014    Instruction ldr r2,=donnees
Cycle 1, Register write, PC_SVC, val: 00000024                      Ecriture de pc pour aller à l'instruction suivante
Cycle 1, Register read, PC_SVC, val: 00000028                       On lit le registre pc pour avoir l'adresse de la lecture
Cycle 1, Mem read (4 bytes) addr: 0000003C, val: 00002800           On lit la valeur à l'adresse pc + 0x14, soit l'adresse de la balise donnees
Cycle 1, Register write, R02_SVC, val: 00002800                     On stocke cette valeur dans r2

Cycle 2, Register read, PC_SVC, val: 00000028
Cycle 2, Mem read (4 bytes, fetch) addr: 00000024, val: E3A01000    Instruction mov r1,#0
Cycle 2, Register write, PC_SVC, val: 00000028
Cycle 2, Register read, CPSR, val: 000001D3                         On lit le registre CPSR pour vérifier si l'instruction doît être exéctuée
Cycle 2, Register read, R00_SVC, val: 00000000                      On lit r0 inutilement pour simplifier le code
Cycle 2, Register read, CPSR, val: 000001D3                         On lit le registre CPSR pour mettre à jour les flags si besoin
Cycle 2, Register write, R01_SVC, val: 00000000                     On écrit la valeur immédiate dans r1

loop:
Cycle 3, Register read, PC_SVC, val: 0000002C
Cycle 3, Mem read (4 bytes, fetch) addr: 00000028, val: E7D20001    Instruction ldrb r0,[r2,r1]
Cycle 3, Register write, PC_SVC, val: 0000002C
Cycle 3, Register read, R02_SVC, val: 00002800                      On lit la valeur de r2
Cycle 3, Register read, R01_SVC, val: 00000000                      On lit la valeur de r1
Cycle 3, Mem read (1 bytes) addr: 00002800, val: 00000012           On récupère un octet à l'adresse r2 + r1,
Cycle 3, Register write, R00_SVC, val: 00000012                     Et on le stocke dans r0, en ajoutant des 0 pour que ce soit une valeur 32 bits

Cycle 4, Register read, PC_SVC, val: 00000030
Cycle 4, Mem read (4 bytes, fetch) addr: 0000002C, val: E33000F0    Instruction teq r0,#0xF0
Cycle 4, Register write, PC_SVC, val: 00000030
Cycle 4, Register read, CPSR, val: 000001D3                         On lit le registre CPSR pour vérifier si l'instruction doît être exéctuée
Cycle 4, Register read, R00_SVC, val: 00000012                      On lit le registre r0, et on vérifie si r0 = 0xF0
Cycle 4, Register read, CPSR, val: 000001D3                         On lit le registre CPSR pour mettre à jour les flags si besoin
Cycle 4, Register read, CPSR, val: 000001D3                         On sauvegarde la valeur de CPSR
Cycle 4, Register write, CPSR, val: 200001D3                        Et on écrit la valeur stockée avec mise à jour des NZCV

Cycle 5, Register read, PC_SVC, val: 00000034
Cycle 5, Mem read (4 bytes, fetch) addr: 00000030, val: E2811001    Instruction add r1,r1,#1
Cycle 5, Register write, PC_SVC, val: 00000034
Cycle 5, Register read, CPSR, val: 200001D3                         On lit le registre CPSR pour vérifier si l'instruction doît être exéctuée
Cycle 5, Register read, R01_SVC, val: 00000000                      On lit la valeur de r1
Cycle 5, Register read, CPSR, val: 200001D3                         On lit le registre CPSR pour mettre à jour les flags si besoin
Cycle 5, Register write, R01_SVC, val: 00000001                     On écrit dans r1 la valeur lue + 1

Cycle 6, Register read, PC_SVC, val: 00000038
Cycle 6, Mem read (4 bytes, fetch) addr: 00000034, val: 1AFFFFFB    Instruction bne loop
Cycle 6, Register write, PC_SVC, val: 00000038
Cycle 6, Register read, PC_SVC, val: 0000003C                       On lit le registre pc pour effectuer ou non un branchement
Cycle 6, Register read, CPSR, val: 200001D3                         On lit le registre CPSR pour vérifier si l'instruction doît être exéctuée
Cycle 6, Register write, PC_SVC, val: 00000028                      On écrit dans pc l'adresse de la balise loop

On effectue les instructions de la boucle 7 fois de plus

Cycle 35, Register read, PC_SVC, val: 0000003C
Cycle 35, Mem read (4 bytes, fetch) addr: 00000038, val: EF123456   Instruction swi 0x123456, qui stoppe l'exécution du programme
Cycle 35, Register write, PC_SVC, val: 0000003C
```

Nous avons testé toutes les autres instructions en utilisant la même procédure, de plus nous avons utilisé et écrit deux fichiers de test : memory_test.c et registers_test.c servant à tester les fichiers éponymes. Ces deux tests étant automatisés, il suffit de les lancer en tapant les commandes : ./registers_test et ./memory_test dans la source du projet.

V) Journal de bord

17/12/2020 : mise en place du github, du trello, début de la rédaction du compte rendu. Installation des composants nécessaires au codage. Découverte du code fourni. Lecture du

sujet en groupe et définition des étapes du projet et des fonctions à réaliser. Répartition des tâches sur trello afin de faciliter le contrôle de l'avancement.

18/12/2020 : analyse et compréhension du code fourni. Début programmation de memory.c et de registre.c.

Vacances

04/01/2021 : fin de la programmation de registre.c et début du chargement des instructions (partie 2). Reprise à 0 de la partie mémoire à cause d'erreurs.

05/01/2021 : fin de la reprise de la mémoire et du chargement des instructions. Premiers tests sur de petits programmes.

06/01/2020 : fin des instructions load, store et création de fichiers tests pour les registres. Fin des instructions merge dans le main, vérification du fonctionnement des instructions. Création d'un fichier test pour les registres et correction de registre.c.

07/01/2021 : vérification des traces grâce aux différents exemples fournis. reprise à zéro des registres pour prendre en compte les modes.

08/01/2021 : Correction de la gestion des registres et load multiple. Début de la rédaction du dossier de projet.

11/01/2021 : Ajout des instructions optionnelles LDRSH et LDRSB, début de l'écriture des tests, correction des conditions et avancement sur la rédaction du rapport.

12/01/2021 : Ajout de nouveau tests, correction des bogues trouvés grâce à ces tests et avancement du rapport

13/01/2021 : Préparation de la démo du 15/01

VI) Répartition du travail

du 17/12/2020 au 04/01/2021 : Travail en groupe pour la phase d'analyse. Puis mise en place de petit groupes (deux groupes de 3) pour écrire les fichiers memory.c et registers.c.

Par la suite, nous avons décidé de faire les corrections en groupe complet.
Les instructions ont été réalisées par petits groupes (trois groupes de 2).
Enfin, le dossier à été réalisé en groupe complet.

VII) Mode d'emploi

A) Compilation

1. paquet requis:

- gcc-arm-none-gnueabi
- gdb-multiarch
- automake
- flex

2. commande:

Se placer dans le répertoire du projet puis exécuter la commande suivante dans un terminal :

./configure CFLAGS='-Wall -Werror -g'

make.

B) lancement du programme

Il est nécessaire de lancer deux terminaux.

Terminal 1 exécuter la commande :

./arm_simulator

Deux lignes devrait apparaître :

Listening to gdb connection on port <port gdb>

Listening to irq connections on port <port irq>

Terminal 2 exécuter la commande :

gdb-multiarch

puis une fois gdb lancé

file <chemin jusqu'au fichier executable ARM à ouvrir>

target remote localhost:<port gdb>

load

Ensuite utilisation classique de gdb