DURHONE Coralie
EYMOND LARITAZ Cyprien
MOULAYAT Yacine
VILLALTA Adrien

Projet logiciel

Simulateur ARM

1. Structure du code

1.1. arm_instruction

Pour commencer nous récupérons les bits 28 à 31 qui forment la condition de l'instruction, puis selon la valeur de *cond*, on vérifie les flags concernés et si la condition échoue, on arrête de traiter l'instruction courante, sinon on vérifie les bits 25 à 27 de l'instruction pour traiter l'instruction dans les sous fonctions regroupant les classes d'instructions détaillées ci-dessous.

1.2. arm_data_processing

Le décodage se fait en deux étapes, la première consiste à récupérer les champs fixes pour tout type d'instruction, à savoir l'*opcode*, *rn*, *rd*, *shift*, *rm* et *S*. Ensuite nous faisons un traitement différent pour déterminer le *shifter_operand* selon la valeur du bit 4 de l'instruction. Dans chacun des deux cas, nous calculons le *shifter_operand* grâce aux variables précédemment initialisées et nous retournons à un traitement commun où nous appelons la fonction d'instruction correspondant à l'opcode, puis selon le bit *S*, nous mettons à jour les flags contenus dans *cpsr* puis écrivons le résultat de l'opération dans le registre *rd*.

Les instructions sont déclarées et réalisées avec la signature *uint32_t instr(arm_core, uint32_t rn, uint32_t shifter_operand, int S)*. Elles sont stockées dans un tableau de fonctions dans l'ordre des opcodes de l'instruction, de sorte à ce que *instructions[x]* soit l'adresse de la fonction qui reproduit le comportement de l'instruction d'opcode égal à *x* (soit pour l'instruction *ADC*, d'opcode *0b0101*, elle sera à l'index 5 du tableau). Cette structure nous permet de gagner du temps et des lignes dans le programme, car on peut simplement récupérer l'opcode de l'instruction et d'appeler la fonction dont l'index est l'opcode dans le tableau de fonctions.

1.3. arm_branch_other

Une fois le décodage effectué et la condition validée, nous utilisons la fonction arm_branch dans le cas d'un branchement (B ou BL). Pour effectuer le branchement, nous avons besoin du PC ainsi que les bits de 0 à 23 de l'instruction qui spécifient l'adresse du branchement. Si le bit 24 de l'instruction est à 1 alors il s'agit d'un branchement avec mémorisation de l'instruction de retour dans le registre LR (BL). Dans ce cas nous écrivons dans r14 PC – 4. Le PC a deux instructions davance sur celle qui est exécutée donc c'est celle qui le précède qui nous intéresse ici.

Après dans tous les cas nous récupérons l'adresse du branchement ciblé en trois étapes. La première est d'étendre les 24 bits récupérés précédemment sur 30. Dans le cas où nous avons un négatif les bits de 24 à 29 serons à 1 afin de ne pas perdre le signe et 0 sinon. La deuxième étape est de décaler le résultat de deux bits à gauche pour qu'il soit sur 32 bits. La dernière étape consiste à additionner ces 32 bits avec le PC et nous obtenons l'adresse que doit prendre le PC pour l'instruction à suivre.

1.4. arm_load_store

Le décodage d'une instruction load/store consiste à analyser d'abord si l'on fait un load ou un store, puis le type de load /store (word, byte, half). On effectue ensuite l'analyse un à un des bits qui permettent de savoir le mode d'adressage utilisé dans l'instruction (les bits I, P, U, W, et le champ du bit 11 au bit 0) avec les calculs d'adresse et opérations sur le registre d'adresse correspondants.

Pour le load/store multiple, on analyse les bits 15 à 0, correspondant aux registres chargés ou enregistrés, lorsque leur numéro de bit correspondant est à 1.

2. Fonctionnalités

- structure registers.c/memory.c
- fetch d'instructions, traitement de la condition
- traitement instructions data processing : AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN
- Mise à jour des flags
- traitement instruction load/store : LDR, LDRB, LDRH, LDM(1), STR, STRB, STRH,STM(1)
- traitement instruction branch : B/BL

3. Bugs connus non résolus

- Mise à jour des flags non testée

4. Tests effectués

5. Progression et répartition du travail

18 Décembre

Prise en main du projet, début d'implémentation : Coralie/Cyprien sur memory.c, Adrien/Yacine sur registers.c

19 Décembre

Memory.c/registers.c terminés, début d'implémentation : Coralie/Cyprien sur arm_data_processing.c, Adrien/Yacine sur arm_load_store.c

20 Décembre

Cyprien : continuation du traitement d'instructions data processing
Coralie : début d'implémentation du branch
Adrien : implémentation des instructions store
Yacine : implémentation des instructions load

21 Décembre

Cyprien : débugging data processing
Coralie : débugging branch
Adrien/Yacine : début implémentation load/store multiple et instructions diverse

22 Décembre

Cyprien/Coralie : débugging
Adrien/Yacine : fin de l'implémentation d'instructions load/store et instruction diverse

8 Janvier

Audit de code : ensemble OK, pas de retard, peut mieux faire sur la factorisation du code pour le load/store
Pour tout le groupe, débugging et tests des différentes instructions

9 Janvier

Cyprien/Coralie : début d'implémentation de la mise à jour des flags
Adrien/Yacine : débugging, et tests de toute les options d'adressage pour le load/store

10 Janvier

Cyprien/Coralie : continuation implémentation mise à jour des flags
Adrien/Yacine : préparation des tests pour la soutenance, et vérification du fonctionnement de l'ensemble

11 Janvier

Cyprien : fin de l'implémentation de la mise à jour des flags (non testée), début de rédaction de la synthèse
Adrien : écriture de tests, rédaction de la synthèse
Yacine : écriture de tests
Coralie : rédaction de la synthèse