SIMULATEUR DE PROCESSEUR ARM V5

PROJET PROG5 – L3 INFORMATIQUE

GROUPE 5:
AIT GUENI SSAID ABDERRAHMANE
IMBERT JOSQUIN
SIDORETS KIRILL
NINGYU LI
ZHAN JIZONG
WANG JINNIU

SIMULATEUR DE PROCESSEUR ARM V5

PROJET PROG5 – L3 INFORMATIQUE

INTRODUCTION

Ce projet est dédié à la réalisation d'un simulateur de processeur ARMv5. Notre travail consiste plus précisément à une simulation de la mémoire et des bancs de registres ainsi qu'au développement des étapes de décodage et d'exécution d'une instruction par la manipulation de la mémoire et des registres simulés. L'étape initiale d'un cycle de processeur – récupération de l'instruction (fetch) – nous est déjà fournie.

Nous avons commencé ce projet par la lecture et la compréhension du sujet proposé et de la structure des fichiers. Nous avons d'abord fait la spécification des fichiers d'en-tête memory.h et registers.h. Puis, nous avons développé les fichiers sources associés. Après avoir créé ces structures de registres et de mémoire, nous nous sommes réparti les tâches pour la spécification et le développement des étapes de décodage et d'exécution.

STRUCTURE DU CODE DEVELOPPE

Le fichier principal du projet est arm_instruction dans lequel on fait les étapes principales (fetch, decode, execute) du fonctionnement d'un processeur arm pour exécuter un programme.

Les étapes permettant l'exécution d'une instruction (arm_execute_instruction()) se font dans cet ordre :

1. Chargement de l'instruction de la mémoire (fetch) ;
2. Test si l'instruction est valide (avec la fonction condition_passed()).
3. Décodage du type d'instruction et appel aux fonctions définies dans les autres fichiers (arm_data_processing, arm_load_store et arm_branch_other) pour la suite du décodage.

Les fonctions permettant la suite du décodage et l'exécution de l'instruction sont décrits ci-dessous :

A. arm_data_processing :

1. **arm_data_processing() :** décodage spécialisé pour les instructions de traitement de données AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN.
2. **arm_data_processing_immediate_msr() :** Décodage spécialisé pour l'instruction MSR avec l'opérande immédiate.
   MSR{<cond>} CPSR_<fields>, #<immediate>
   MSR{<cond>} SPSR_<fields>, #<immediate>

B. arm_load_store :

1. **arm_load_store() :** Traite les instructions LDR, STR , LDRB ,STRB ,LDRH ,STRH.
2. **arm_load_store_multiple() :** Traite les instructions LDM , STM.
3. **arm_coprocessor_load_store() :** Récupère une instruction et envoie l'instruction pour traitement.

C. arm_branch_other :

1. **arm_branch() :** Effectue le branchement (B, BL, BLX(1), BLX(2), BX, BXJ) décrit par l'instruction.
2. **arm_coprocessor_others_swi():** Exécute les instructions SWI.
3. **arm_miscellaneous():** Exécute des instructions diverses (En pratique, une seule instruction est traitée puisque c'est la seule « miscellaneous instruction » disponible en ARMv5).

Enfin, la fonction arm_exception (fichier arm_exception.c) permet d'effectuer un branchement sur le vecteur d'exceptions lorsqu'une exception se produit.

LISTE DES FONCTIONNALITES IMPLEMENTEES ET MANQUANTES

Fonctionnalités implémentées :

- Structure de mémoire,
- Structure de registres,
- Décodage d'instructions, incluant les instructions :
    - De branchement et autres (branch) – B, BL, BLX, BXJ, BX, SWI, CLZ,
    - De traitement de données (data-processing) - AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN, MSR, MRS,
    - De chargement en mémoire / depuis la mémoire (load / store) - LDR, STR, LDRB, STRB, LDRH, STRH, LDM, STM.
- Exécution de ces mêmes instructions,
- Traitement des exceptions.

Fonctionnalités manquantes :

- Instruction LDRSH,
- Instruction LDRSB.

LISTE DES EVENTUELS BOGUES CONNUS MAIS NON RESOLUS

Nous avons passé beaucoup de temps sur les derniers jours avant le rendu du projet à corriger des bugs. Tous ceux que nous avions repérés ont donc été corrigés. Cependant, nous n'avons pas eu le temps d'approfondir les tests sur les exceptions et le traitement des interruptions (nous avons uniquement testé l'exception SWI puisqu'il suffisait de la lever avec une instruction SWI).

Sinon, nous avons repéré quelques incohérences comme l'utilisation alternative de variables de type uint8_t et de type int pour une seule et même information. Nous n'avons pas eu le temps de corrigé ces incohérences mais elles n'empêchent pas le bon fonctionnement du simulateur.

LISTE ET DESCRIPTION DES TESTS EFFECTUES

Pour tester, le simulateur, nous avons réalisé des jeux de tests que nous avons exécuté. Nous pouvons ainsi comparer le résultat attendu d'une instruction avec le résultat produit par le simulateur.

Les tests effectués sont les suivants (répertoire Example) :

- example5 : test unitaire de chaque instruction et leurs variations
- example6 : test des conditions, vérifie que les conditions sont bien traitées
- example7 : test de la fonction « load »
- example8 : test de la fonction « store »
- example9 : test des fonctions LDM et STM
- example10 : test d'un programme qui calcule la valeur absolue de -5 et stocke le résultat dans la mémoire. Dans ce test, nous testons les fonctions bl, cmp, b, mvn, add, str, ldr, mov, dans le contexte d'un programme.
- example11 : test des instructions de traitement de données
- example12 : test d'une exception SWI

Les résultats attendus sont décrits en commentaires dans les codes sources des tests.