Groupe 17

Projet Logiciel
Réalisation d'un simulateur ARM

1. Mode d'emploi :

Ouvrir deux terminaux pour lancer :

- Le simulateur (./arm_simulator), les numéros de deux portes sont ensuite affichés. Le simulateur peut être lancé avec plusieurs options. Pour les connaître, ./arm_simulator –help. Pour afficher les d'accèces aux registres, par exemple, taper ./arm_simulator –trace-registres.
- Le client gdb (arm-none-eabi-gdb) est lancé dans l'autre terminal. Ensuite le fichier souhaité est indiqué avec la commande file Examples/examplei. Puis taper target remote localhost: <port>, le client gdb se connecte au serveur simulateur en prenant le numéro de port précédemment donné .

2. Descriptif de la structure du code développé :

Lors la première étape, Nous avons réalisé les fonctions d'accès aux registres (fichier registre.c) et à la mémoire (fichier memoire.c) de la machine simulée. Ce qui a permis le chargement d'un programme dans le simulateur. La communication entre le simulateur et le client gdb étant faite à l'aide d'un protocole TCP/IP implémenté dans la squelette fournie.

Pour la deuxième étape le fichier, nous avons implémenté les fonctions de décodage et d'exécution d'instruction machine. Le fichier arm_instructions.c a été modifié afin de exécuter les instructions ARM qui se regroupent principalement en:

a) Instructions de traitement de données (data processing)

Nous avons implémenté plusieurs fonctions dans le fichier arm_data_processing.c. Pour chaque instruction sur 32bits les informations nécessaires sont récupérées par la fonction ***set_parameters***. Cette fonction est ensuite appelée dans la fonction **select_operation**, qui, elle, sélectionne la bonne opération à exécuter l'aide de l' *opcode*. Chaque opération (AND , EOR, ADD, …) est implémentée dans une procédure unique. Si nécessaire, les *flags* sont récupérés et sont mis à jour à la fin de chaque opération avec ***update_flags***.
Les modes d'adressages sont pris en compte dans la procédure la fonction ***write_shifter_operand_sco*** *(cfr ARM ARMv5 A5-2 section A5.1 Addressing Mode 1 - Data-processing operands)*

b) Instructions de rupture de séquence

Les instructions de branchement B et BL ont été implémentés, si le bit L de l'instruction est a 1 on écrit la valeur de PC dans le registre LR et ensuite on actualise PC avec l'adresse a laquelle on veut brancher.

c) accès à la mémoire (load_store)

pour traiter ces instructions, nous avons modifié le fichier arm_load_store.c et arm_load_store.h. Pour chaque instruction sur 32bit nous allons recuperer les informations necessaires(Le bit P,B,W,L,... ainsi que offset,rn,rm,...) pour traiter l'instruction. Les

instructions LDR,LDRB,STR,STRB sont gérés par la fonction arm_load_store alors que LDRH,STRH par arm_load_store_miscellaneous, nous realisons l'adressage au cas par cas en traitant chaque bit d'information recuperé par les fonctions get_bit pour effectuer l'adressage adéquat. Nous verifions les flags avec la fonction condition_OK définis dans arm_instruction.

d) Instructions diverses

L'instruction MRS est implémentée, elle permet de déplacer la valeur de CPSR ou SPSR dans un registre général de destination.

Liste des fonctionnalités implémentées :

fichier util.c :

uint32_t lsr(uint32_t value, uint8_t shift)
uint32_t lsl(uint32_t value, uint8_t shift)

fichier arm_instruction.c :

int condition_OK(uint8_t condition, uint8_t flag_N, uint8_t flag_Z, uint8_t flag_C, uint8_t flag_V)
int arm_decode(arm_core p)
int arm_execute_instruction(arm_core p)
int arm_step(arm_core p)

fichier arm_data_processing.c

void update_flags(arm_core p, uint8_t S, uint8_t Rd, uint8_t flag_C, uint8_t flag_V);
void write_flags(arm_core p, uint8_t Rd, uint8_t flag_C, uint8_t flag_V);
uint8_t carryFrom(uint32_t x, uint32_t y);
uint8_t borrowFrom(uint32_t x, uint32_t y);
uint8_t overflowFrom(uint32_t x, uint32_t y, uint32_t z, uint8_t opcode);

uint8_t get_flag_N(arm_core p);
uint8_t get_flag_Z(arm_core p);
uint8_t get_flag_C(arm_core p);
uint8_t get_flag_V(arm_core p);

uint32_t and(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t eor(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t sub(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t rsb(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t add(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t sub(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t adc(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t sbc(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);
uint32_t rsc(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand);

uint32_t tst(arm_core p, uint8_t Rn, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t teq(arm_core p, uint8_t Rn, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t cmp(arm_core p, uint8_t Rn, uint32_t shifter_operand);
uint32_t cmn(arm_core p, uint8_t Rn, uint32_t shifter_operand);
uint32_t orr(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t mov(arm_core p, uint8_t S, uint8_t Rd, uint32_t shifter_operand, uint8_t shifter_carry_out);
uint32_t bic(arm_core p, uint8_t S, uint8_t Rd, uint8_t Rn, uint32_t , uint8_t shifter_carry_out);
uint32_t mvn(arm_core p, uint8_t S, uint8_t Rd, uint32_t shifter_operand, uint8_t shifter_carry_out);

void set_parameters(arm_core p, uint32_t ins,uint8_t *opcode, uint8_t *S, uint8_t *Rn, uint8_t *Rd, uint32_t *shifter_operand, uint8_t *shifter_carry_out);
uint32_t select_operation(arm_core p, uint32_t ins);
uint32_t shift_operation(uint32_t value, uint8_t shift, uint8_t shift_amount);
void write_shift_operand_sco(arm_core p, uint32_t ins, uint32_t *shifter_operand, uint8_t *shifter_carry_out);

int arm_data_processing_shift(arm_core p, uint32_t ins);
int arm_data_processing_immediate_msr(arm_core p, uint32_t ins);

fichier arm_load_store.h/c :
int arm_load_store_miscellaneous(arm_core p, uint32_t ins);
fichier arm_branch_other.h/c :
int arm_branch(arm_core p, uint32_t ins);
int arm_miscellaneous(arm_core p, uint32_t ins);

Liste des bogues connus :
Liste des fonctionnalités manquantes :
int arm_coprocessor_others_swi(arm_core p, uint32_t ins)

Les tests effectués :
memory_test : test de la partie mémoire
test_branch.s : test les instructions de branchement B et BL

Journal du travail :

18 Décembre :
-Lecture et compréhension du sujet / Parcours de la doc ARM
19 Décembre :
-Envoi du code sur github
3 Janvier :
-Répartitions des taches

4-7 Janvier :

-Implémentation des registres par Dan et Benoit

-Implémentation de la mémoire par Mathias et Yann

8 Janvier :

-Implémentation de la phase de décodage et d'exécution par Dan

-Implémentation des instructions de traitement de données par Mehrnaz et Moaz

9 Janvier :

-Implémentation des branchements par Benoit

-Implémentation des fonctions de arm_load_store.c par Mathias et Yann

13 Janvier:

-Test des instructions

-Création des documents de rendu par Mermaz et Moaz