Groupe 16 :

BLEJ Mouad

BALADI Ayoub

BOULAGHLA Aderrazzak

EL BOUCHOUARI Zakaria

HOUSSEIN KASSIM Abro

LUIGGI Alexandre

Rapport du projet

Mode d'emloi :

Make : pour compiler tous les programmes.

./memory_test : un test pour les opérations de la mémoire.

./arm_simulator et gdb-multiarch dans deux terminaux pour faire tester tous nos programmes.

Descriptif de la structure du code développé :

arm_branch_other.c :

arm_data_processing.c:

_arm_data_processing_immediate_msr

_arm_data_processing_shift : cette fonction calcule tous la valeur du Registre (Shifter Operand), et aussi le carry des shift (Sco) .

arm_load_store.c :

_ laod_stot_byte_word : cette fonction charge ou écrit dans la mémoire un octet ou mot

_ laod_stot_halfword : cette fonction charge ou écrit dans la mémoire un demi-mot

_ est_immediate : cette fonction s'en occupe da la valeur de l'offset dans le cas d'un octet ou d'un mot

_ half_immediate : cette fonction s'en occupe de la valeur de l'offset dans le cas d'un demi-mot

_ half_word : cette fonction s'en occupe de différents types d'adressage dans le cas d'un demi-mot

_ byte_word : cette fonction s'en occupe de différents types d'adressage dans le cas d'un mot ou d'un octet

_ arm_load_store : la fonction générale qui s'en occupe de load et store dans le cas d'un mot, demi-mot et octet, en prenant en compte des différents types d'adressage

_ creer_reglist : créée un tableau contenant les numéros de registre qu'elle récupère en analysant les 16 premier bits d'une instruction STM ou LDM.

_ check_reg_plag : Vérifie si l'ensemble des registres d'un tableau remplis par créer_reglist vérifie les conditions pour que l'instruction puisse être exécuter.

_ arm_load_store_multiple : fonction d'implémentation pour les instructions STM et LDM prenant en compte les suffixes optionnel (IA, IB, DA, DB)

_ arm_coprocessor_load_store : fonction contenant l'implémentation de la fonction MRS

arm_instruction.c :

utile.c :

La fonction **do_shift**(uint32_t * Sop, uint8_t * sco , uint32_t data , uint32_t val_Rm ,uint8_t shift , int mode , uint8_t c) dans util.c qui prend en paramètre Sop (Shift Operator) , sco (Shift carry out), data soit la valeur d'un registre ou bien une constante (Immediate), val_Rm c'est la valeur du registre Rm (voir la page A5-3 de la documentation) , shift ce sont les bit 5 et 6 de l'instruction, mode soit IMMEDIATE ou REGISTER (0 ou 1) , et le flag c de cspr. La fonction modifie la valeur de Sop et de sco pour que la fonction **execute_instruction** fasse son travail.

add : la somme de deux registre r0 r1 dans un troisième registre r2

la somme d'un registre avec une valeur immédiate

adc : il calcule le carry

adcs : mise ajours des flags test over flow z=1 c =1

adds : mise a jours des flags n=1 v=1

and : et logique entre deux registres

et logique entre registre et valeur immédiat

xor :

ands : mise a jours des flags

eor : mise à jour de n et z

eors : mise à jour de n et z

sub : soustraction de deux registre r0 r1 dans un troisième registre r2

la soustraction d'un registre avec une valeur immédiate

subs : mise a jours des flags

sbc : soustrait le négatif de "carry"

sbcs : mise a jours des flags

rsb : reverse la formule de soustraction

rsbs : mise ajours des flags

Listes de bogues (instructions) :

-STM (tester mais beug)

-MRS (pas tester)

Test :

example1 :

```
.global main
.text
decr:
    subs r0, r0, #1
    mov pc, lr

main:
    mov r0, #5
loop:
    bl decr
    bne loop
end:
    swi 0x123456
.data
```

tester les instruction de branchement

le résultat : tant que r0 diffèrent de 0 on décrément la valeur de r0 par appel de l'étiquette ‘decr ‘

example2 :

```
.global main
.text
main:
    mov r0, #120
    mov r1, #5
    add r2, r1, r0, lsl #8
    add r3, r1, r0
    add r4 , r1, r2
    subs r5, r1, r0, lsr #8
    mov r0,#5
    tsteq r0, r1
    cmpne r1, r2
    mvn r0, #5
    swi 0x123456
```

cet exemple présente le test des instructions de traitement de données avec des shifts.

example3 :

```
.global main
.text
main:
    ldr r0, =limite              // r0 <- limite /* r0 <- 0x2800 */
    mov r4, #1                   // r4 <- 1
    mov r5, #8                   // r5 <- 8
    ldr r4, [r0]                 // r4 <- *limite   /* r4 <- 0x12345678        */
    str r5, [r0, #4]             // *(r0+4) <- r5   /* *(0x2804) <- 8          */
    strh r4,[r0, #8]             // *(r0+8) <- r4   /* *(0x2808) <- 0x5678     */
    strb r4,[r0, #12]            // *(r0+12) <- r4  /* *(0x2808) <- 0x78       */
                                 // r0 <- r0+12     /* r0 <- 0x280c            */
    str r4,[r0, #16]             // *(r0+4) <- r4   /* *(0x2810) <- 0x12345678 */
    ldrb r1, [r0]                // r1 <- 0x78
    ldrh r2, [r0]                // r2 <- 0x7800
    ldr r3, [r0]                 // r3 <- 0x78000000
    ldr r6, [r0, r5, LSR #1 ]    // r6 <- *(r0+ (r5 >> 1) )  /* r6 <- *(0x2810) <- 0x12345678 */

    swi 0x123456
.data
limite:
    .word 0x12345678
```

```
(gdb) info reg
r0             0x280c              10252
r1             0x78                120
r2             0x7800              30720
r3             0x78000000          2013265920
r4             0x12345678          305419896
r5             0x8                 8
r6             0x12345678          305419896
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x0                 0x0
lr             0x0                 0
pc             0x50                0x50 <main+48>
cpsr           0x1d3               467
(gdb) x 0x2800
0x2800: 0x12345678
(gdb) x 0x2804
0x2804: 0x00000008
(gdb) x 0x2808
0x2808: 0x56780000
(gdb) x 0x280c
0x280c: 0x78000000
(gdb) x 0x2810
0x2810: 0x12345678
(gdb)
```

example4 :

```
.global main
.text
main:
    // on charge les adresses des valeurs étiquetté dans des registres
    ldr r1, =donnee             // r1 <- donnee
    ldr r2, =fin                // r2 <- fin]
    ldr r3, =limite             // r3 <- limite
    ldr r0, =addr               // r0 <- addr
    ldmda r3, {r4, r5, r6}      // r4 <- [r3] puis r5 <- [r3-1] puis r6 <- [r3-2]
                                // on charge les registres dans la liste par les valeurs à l'adresse contenu dan r3
                                // et on décrémente l'adresse quand on passe au suivant
    stmia r0, {r4, r5, r6}      // [r0] <- r4 puis [r0+1] <- r5 puis [r0+2] <- r6
                                // on stock les valeurs des registres r4, r5, et r6 au adresses [r0,..,r0-2]
    swi 0x123456
.data
donnee:
    .word 0x11223344
fin:
    .word 0x55667788
limite:
    .word 0x12345678
addr:
    .word 0x280c
```

Répartition du travail et progression :

Le travail a été répartis entre les trois binômes du groupe. Tout d'abord nous avons due compléter les fichier registers.c et memory.c. Ayoub, Abderrazzak et Mouad pour le fichier registers.c et Abro et Zakaria pour le fichier memory.c. Ensuite chaque binôme a dû implémenter un certain nombre d'instructions en fonction des fichiers à compléter.

arm_branch_other.c et arm_data_processing.c compléter par Abderrazzak et Ayoub
Instructions implémenté : B/BL, AND, EOR, SUB, RSB, ADD, ADC, SBC

arm_data_processing.c compléter par Mouad et Alexandre
Instructions implémenté : RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN

Fonctions implémentée : arm_data_processing_shift, do_shift.

arm_load_store.c compléter par Abro et Zakaria
Instructions implémenté : LDR, LDRB, LDRH, STR, STRB, STRH, LDM(1), STM(1), MRS

La fonction **arm_execute_instruction** du fichier arm_instruction.c a été compléter indépendamment par chaque binôme afin de s'assurer du bon fonctionnement des instructions qu'ils ont ainsi implémentés. La fonction finale est complétée à partir des trois versions.