

Projet L3 S1

Simulateur ARM

Groupe 13 :

Ivane ADAM : ivane.adam@etu.univ-grenoble-alpes.fr

Samuel BRUN : samuel.brun@etu.univ-grenoble-alpes.fr

Tommy COMPTOIS : tommy.comtois@etu.univ-grenoble-alpes.fr

Karim FLISS : karim.fliss@etu.univ-grenoble-alpes.fr

Flavien LACLUQUE : flavien.lacluque@etu.univ-grenoble-alpes.fr

Emile LAFFONT : emile.laffont@etu.univ-grenoble-alpes.fr

Timon ROXARD : timon.roxard@etu.univ-grenoble-alpes.fr

Mode D'emploi

Compilation projet :

make

Compilation des fichiers .s :

arm-none-eabi-gcc -mbig-endian -Wall -Werror -g -c -o <nom_test>.o <nom_test>.s
puis

arm-none-eabi-gcc -mbig-endian -Wall -Werror -g -nostdlib -T ./linker_script -n
--entry main -Wl,-EB -o <nom_test> <nom_test>.o

Exécution:

Ouvrir 2 terminaux :

Un terminal pour le programme arm_simulator :

./arm_simulator

Un terminal pour gdb :

arm-none-eabi-gdb

file <Fichier .s compilé>

target remote localhost:<n° de port>

load

Pour lancer une exception :

Ouvrir un troisième terminal

./send_irq localhost <port_irq> <nom_interruption>

Description De La Structure Du Code Développé

principales fonctions et fichiers correspondants


Module Register :

Ce module a pour objectif de simuler les registres de notre processeur ARMv5.

Les registres sont contenus dans un tableau d'entiers non signés (32 bits) de 37 cases, soit une case pour chaque registre disponible dans chaque mode selon le tableau ci-dessus de la DOC ARM page A2-5.

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Associé à ce type, il y a diverses fonctions permettant l'accès en lecture (read_register) et en écriture (write_register) de chacun des registres ainsi que des fonctions fournissant un accès au mode courant du processeur simulé. La gestion du mode d'exécution se contrôle via le registre CPSR de notre machine et plus précisément via ses 5 bits de poids faible. Le module contient aussi une fonction pour l'allocation mémoire de nos registres simulées et une pour la suppression de ces derniers.

Module Memory :

Ce module permet la gestion de la mémoire de notre simulateur.

La mémoire de notre simulateur est représentée ainsi:

- Un tableau de type `uint8_t` d'une taille `MEM_SIZE` (prédéfinie à 2^{30})
- Une taille (inférieur à 2^{30})
- Un booléen indiquant le boutisme de la machine (Vrai si BE)

Le module contient des fonctions pour la création et la suppression de la mémoire, une fonction d'accès à la taille de cette dernière, ainsi qu'une panoplie de fonctions permettant l'accès aussi bien en lecture qu'en écriture de mot, demi-mot et octets. Ces dernières fonctions prennent en compte un éventuel boutisme différent entre la mémoire et la machine.

Module utils :

Ce module apporte plusieurs outils utilisés tout au long du projet.

En plus des fonctions nativement implémentés dans ce module, nous avons ajouté des fonctions de gestion d'une file a priorité ainsi que la fonction condition. Cette fonction prend en paramètre le registre CPSR de notre machine et une condition (composée des 4 bits de poids fort de l'instruction courante). En fonction de cette condition, un booléen sera renvoyé, il décrit l'état des bits Z,N,C,V du registre CPSR pour la condition (cf tableau ci-dessous extrait de la doc partie A3.2.1).

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See Condition code 0b1111	-

Cette fonction nous sera utile pour les modules décrits ci-dessous.

Module arm_instruction :

Ce module a pour objectif de décoder l'instruction courante afin d'appeler la fonction de traitement adéquate.

En plus de la fonction arm_step fournie, nous avons ajouté la fonction is. Premièrement, la fonction arm_execute_instruction va fetch l'instruction a decodé (via la fonction arm_fetch). Suite à cela, l'instruction va être comparée à chacun des 19 patrons suivants (répertorié doc ARM partie A3.1).

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing immediate shift	cond [1]	0	0	0	opcode	S		Rn		Rd																						
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0																							
Data processing register shift [2]	cond [1]	0	0	0	opcode	S		Rn		Rd																						
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0																							
Multiplies: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Data processing immediate [2]	cond [1]	0	0	1	opcode	S		Rn		Rd																						
Undefined instruction	cond [1]	0	0	1	1	0	x	0	0																							
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0																							
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L																							
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L																							
Media instructions [4]: See Figure A3-2	cond [1]	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Architecturally undefined	cond [1]	0	1	1	1	1	1	1	1																							
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L																							
Branch and branch with link	cond [1]	1	0	1	L																											
Coprocessor load/store and double register transfers	cond [3]	1	1	0	P	U	N	W	L																							
Coprocessor data processing	cond [3]	1	1	1	0	opcode1				CRn																						
Coprocessor register transfers	cond [3]	1	1	1	0	opcode1	L			CRn																						
Software interrupt	cond [1]	1	1	1	1																											
Unconditional instructions: See Figure A3-6	1	1	1	1																												

La comparaison entre l'instruction et le patron se fait au travers de la fonction is. Cette fonction prend le patron sous forme de chaîne de caractère et l'instruction courante. Les bits non significatifs du patron sont représentés par le caractère X. Quand un patron est reconnu pour l'instruction courante, la fonction arm_execute_instruction exécute la procédure associée au patron. Ces procédures sont contenues dans les modules décrits ci-dessous.

Module arm_data_processing :

Sans aucun doute le module le plus conséquent du projet, il a pour objectif de traiter les instructions de données.

Le module est composé de 3 principales fonctions :

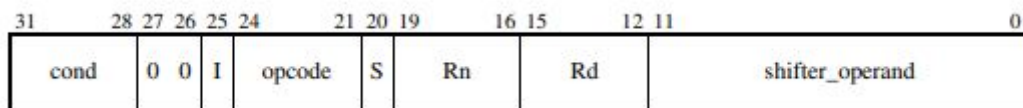
- arm_data_processing_immediate_msr

- arm_data_processing_shift
- processing

La fonction `arm_data_processing_immediate_msr` s'occupe du paterne "move immediate to a status register", commande MSR. Pour traiter ce cas, nous nous sommes appuyés sur la partie dédiée dans la doc, c'est-à-dire A4.1.39. En fonction de l'instruction courante et du mode d'exécution courant, le programme va effectuer une copie dans CPSR ou SPSR de la valeur décrite dans l'instruction (cette valeur peut être immédiate avec ou sans rotation ou être contenue dans un registre).

La fonction `arm_data_processing_shift` est utilisée pour les patrons "data processing immediate shift" et "data processing register shift". L'objectif principal de cette fonction est de déterminer le shifter_opera_{nde} de l'instruction. La partie A5.1 nous informe sur les différentes formes que peuvent prendre le shifter_opera_{nde} et comment traiter chacune d'entre elles. Ainsi donc notre programme filtre le type de shift (immediate ou register puis LSL,LSR,ASR,ROR). Pour chacun de ces cas, une fonction calcul le shifter_opera_{nde} et le shifter_carry_out (en suivant les algorithmes fournis par la doc). Une fois ce travail effectué, le programme peut décrypter le code restant de l'instruction, c'est la fonction `processing` qui s'en occupe.

La fonction `processing` traite pour chaque opéra_{nde} AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR, MOV, BIC, MVN, TST, TEQ, CMP, CMN (la description de chacune de ses opéra_{ndes} se situe à partir de la page A4-4). Nous distinguons 2 types d'instructions, celles qui sont à objectif comparatif (TST, TEQ, CMP, CMN) qui ne vont influencer que sur les ZNCV et les classiques avec un registre de destination. Pour les classiques, chacune d'entre elles existe en version S, qui a pour effet de mettre à jour les ZNCV de CPSR. Pour chaque opération, le programme fait la ou les opérations à effectuer puis met si nécessaire à jour les registres. Cette fonction est appelée dans la continuité de `arm_data_processing_shift` mais aussi pour le paterne data processing immediate. Dans ce cas précis, le shifter_opera_{nde} et le shifter_carry_out sont calculés par la fonction `immediate`. Une instruction de data-processing est encodée de la sorte : (partie A3.4 dans la doc)



Dans des instructions d'addition/soustraction et ces variantes, on prévoit un débordement de son résultat au bit 32 d'indice. on définit le résultat comme `uint64_t` pour ça. On vérifie le débordement, après, on cast le résultat a `uint32_t` pour le mettre dans le registre

Module `arm_branch_other` :

Ce module a pour objectif de gérer les instructions de branchement.

Ce module contient en plus des fonctions déjà implémentées, la fonction `arm_branch`. Cette fonction est appelée pour le paterne "Branch and branch with

link”. La fonction récupère premièrement les différents blocs de l’instruction courante ainsi que le registre d’état de la machine, puis si la condition est vérifiée, procède au branchement en tenant compte de l’alternative L qui sauvegarde PC dans LR avant le branchement. La spécification détaillée des branchements se situe page A4-10 de la doc.

Module arm_load_store :

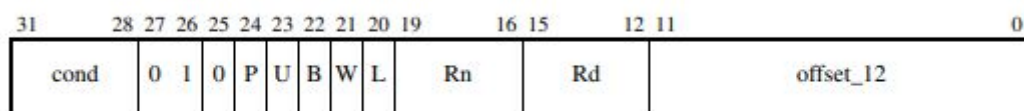
Ce module a pour objectif de gérer les instructions d'accès mémoire.

Le module est composé de 3 fonctions :

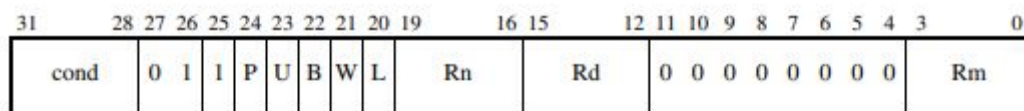
- arm_load_store
- arm_load_store_multiple
- arm_load_store_miscellaneous

La fonction arm_load_store s’occupe des instructions LDR et STR avec leurs options B et H. Dans cette partie on peut dissocier 3 types d’instructions (elles sont répertoriées partie A5.2 de la doc) :

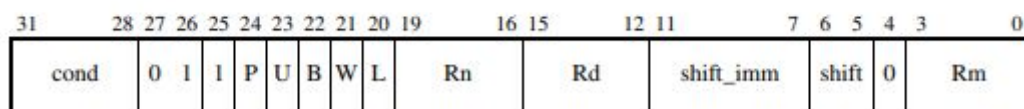
Immediate offset/index



Register offset/index



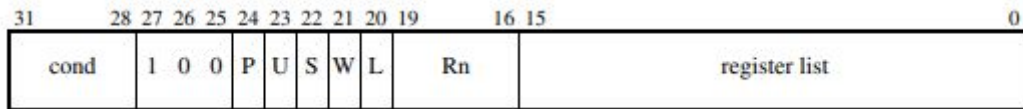
Scaled register offset/index



Chaque type d’instruction ci-dessus est lui-même représenté par 3 variantes qui sont offset, pre-indexed et post-indexed. La distinction entre les sous variantes s’effectue à l’aide des bits P et W. La première étape est de récupérer l’adresse de lecture ou d’écriture, pour cela nous décodons les instructions suivant les 9 modes explicité ci-dessus. L’idée est de récupérer une valeur que l’on va ajouter ou soustraire (en fonction de U) au contenu de rn afin d’obtenir l’adresse souhaitée. La description précise pour obtenir cette valeur dans chacun des 9 cas se situe dans la partie A5.2 de la doc. Une fois ce travail réalisé, le programme n’a plus qu’à distinguer si l’opération est un load ou un store et la taille de la valeur à ranger (byte ou word) via les bits L et B. Cette fonction est utilisée pour le paterne “Load/store immediate offset” et “Load/store register offset”.

La fonction arm load store multiple utilise a peu de choses près le même principe qu’explicité au-dessus si ce n’est qu’au lieu d’y avoir 9 modes il n’y en a que 4 qui

sont incremente/decrement before/after (description partie A5.4 de la doc). Nous cherchons une adresse de début et de fin pour la lecture/écriture. La structure de l'instruction est la suivante (partie A3.12 de la doc) :



Pour chaque bit à 1 dans la liste des registres, un accès mémoire (lecture ou écriture) est effectué avec pour adresse, l'adresse de début +4 par instructions sans dépasser l'adresse de fin. Cette fonction est utilisée pour le paterne "Load/store multiple".

La fonction `arm_load_store_miscellaneous` a sensiblement le même comportement que celles ci-dessus, c'est-à-dire, calcul d'adresse puis opération. Sachant que dans ce cas les opérations sont store de halfword ou load de halfword signé ou non et byte signé. Cette portion de code est décrite partie A5.3 de la doc. Cette fonction est utilisée pour le paterne "Extra load/store".

Nous n'avons pas implémenté les fonctions liées au coprocesseur.

Module `arm_exception` :

Ce module gère les différentes interruptions.

Les principales fonction de ce modules sont :

- `arm_exception`
- `save_state_and_change_mode`

La fonction `arm_exception` est la fonction qui est exécutée quand une interruption apparaît. Les interruptions prises en compte par cette fonction sont RESET, UNDEFINED_INSTRUCTION, SOFTWARE_INTERRUPT, PREFETCH_ABORT, DATA_ABORT, INTERRUPT et FAST_INTERRUPT. Selon la valeur de l'interruption, la fonction appel `save_state_and_change_mode` puis se branche sur le handler de l'interruption (manipulation de PC).

La fonction `save_state_and_change_mode` s'occupe de sauvegarder PC dans LR, CPSR dans SPSR. Une fois les sauvegardes effectuées, PC est modifié pour contenir l'adresse du traitant et le mode est changé. Ces adresses sont en accord avec celles fournies par la doc partie A2.6. Par la suite, si l'interruption est de forte priorité (FIQ ou RESET), les fast interrupts sont désactivés, ainsi que les interruptions normales (dans tous les cas). Ces désactivations se font par manipulation de CPSR.

Le retour à l'état d'avant interruption se fait grâce aux instructions tels que `movs pc, lr` qui sont exécuté à la fin de chaque handler et qui restaurent PC et CPSR.

Les adresses des traitants doivent être définies du côté assembleur. Pour chaque type d'interruption, il faut définir les handler et les chargés dans le vecteur d'interruption aux adresses comme définis dans notre code.

Liste Des Fonctionnalités

Fonctionnalités implémentées	Fonctionnalités manquantes
<p>Instructions de rupture de séquence: B : Branchement BL : Branchement avec lien</p> <p>Instructions de traitement de données: AND : ET logique EOR : OU exclusif logique SUB : Soustraction RSB : Soustraction sens inverse ADD : Addition ADC : Addition avec retenue SBC : Soustraction avec retenue RSC : Soustraction sens inverse avec retenue TST : Test TEQ : Test d'équivalence CMP : Comparaison CMN : Comparaison inversée ORR : OU logique MOV : Déplacement de valeur BIC : ET logique complémentaire MVN : Déplacement de valeur inversée</p> <p>Instructions d'accès à la mémoire : LDR : Chargement d'un mot LDRB : Chargement d'un octet LDRH : Chargement d'un demi-mots LDRSB : Chargement d'un octet signé LDRH : Chargement d'un demi-mots LDRSH : Chargement d'un demi-mots signé STRB : Stockage d'un octet STRH : Stockage d'un demi-mots STR : Stockage d'un mot LDM : Chargement de plusieurs valeurs STM : Stockage de plusieurs valeurs</p> <p>Instructions diverses : MRS : Chargement d'un registre d'état du programme MSR: Modifier une zone spécifique (selon les flags f, s, x, c) de CPSR/SPSR</p>	<p>BLX et BX LDC et STC</p>

Liste Des Bugs Connues Non Réglés

Problème:

Priorité entre DATA_ABORT et PREFETCH_ABORT, l'origine du problème est qu'on ne peut pas déterminer quel ABORT on est entrain d'exécuter avec juste l'information de CPSR. l'information est qu'on est en mode ABT.

Conséquence:

on ne peut pas partir en interruption DATA_ABORT et interrompre PREFETCH_ABORT en cascade car on sait juste que le proc en mode ABT, il peut être en DATA_ABORT ou PREFETCH_ABORT. pas de décision dans ce cas. au pire le simulateur attend que le processeur décente en priorité pour l'exécuter

Liste Des Tests

Pour compiler le test :

Pour lancer le test, le compiler puis l'exécuter (démarche de compilation et d'exécution dans la partie "Mode D'emploi").

Pour vérifier le résultat du test, le protocole est spécifié dans le .s en plus du tableau ci-dessous.

Nom du test	Descriptif	Comment vérifier le résultat
test_data_proc	test toutes les instructions de data processing a la suite	A la fin du test, les registres 0 à 8 contiennent leurs valeurs d'index.
test_interruption	charge les traitants dans le vecteur interruption	La valeur de R1 a une valeur précise en fonction de la dernière interruption faite
test_ldm_stm	test load store multiple	r5 à r8 ont les mêmes valeurs que r1 à r4
test_ZNCV	test les branchements conditionnels	Les branchements A a C ne sont pas fait, le D est fait et R1 est le seul reg modifié et contient 0xFFFFFFFF
test_msr_load_signed	test mrs, ldrsb et ldrsh	CPSR a la valeur 0xF0000010 R2 et R3 valent 0xFFFFFFFF

Tests irq

Il faut ouvrir un 3ème terminal et lancer la commande suivante:

`./send_irq localhost port nom_exception` (dans le répertoire arm_simulator)(si le nom de l'interruption contient une espace il faut le protéger avec un \ ex:prefetch\ abort)

Mais il faut: (par exemple pour l'exception prefetch abort)

```
@ put prefetch_abort on interrupt vector (0xC)
mov r3, #0xC
ldr r2, =prefetch_abort
str r2, [r3]
```

Il faut avoir fait dans le gdb les trois instructions pour pouvoir charger le vecteur d'interruption. Les instructions sont dans le fichier test_interruption.s .

Journal De Bord

Date	Travail réalisé
18-19 Dec	Appropriation du sujet + démarrage avec les modules register et memory
Vacances	Avance du projet pour les membres le pouvant
04 janv	Concertation entre les membres + choix du code le plus avancé
05 janv	début du travail sur la base la plus avancée fournit par Karim (avec les modules data_proc, load_store, instruction et branch)
06-11 janv	Réalisation du module exception + tests sur les différents fichiers et correction de bugs + factorisation
12 janv	Preparation Oral
13-14 janv	Tests du code fin de la preparation oral