

Examen UE INF241 : Introduction aux Architectures Logicielles et Matérielles

Première session 2014-2015, 20 mai 2015, durée 2 h.

Documents, calculatrices, téléphones portables non autorisés. Le barème est donné à titre indicatif.

En annexe, un résumé des instructions du processeur ARM et des procédures de la bibliothèque `es.s`.

1 Fonctions en ARM (11 points)

On considère la zone mémoire suivante :

```
.data
aff: .asciz "resultat :"
```

- (a) Ajoutez, après la chaîne de caractères `aff`, la déclaration d'un tableau `tab` contenant 6 mots. Ce tableau sera initialisé avec les valeurs 40, 30, 10, 20, 12 et 32. N'oubliez pas de régler le problème d'alignement ! (1 point)

Dans un premier temps, nous supposons l'existence des deux fonctions dont les prototypes sont donnés ci-dessous :

```
fonction division(A : entier naturel, B : entier naturel) avec résultat entier naturel
// division retourne le résultat de la division entière de A par B

fonction somme(T : adresse d'un tableau, n : entier naturel) avec résultat entier naturel
// somme retourne la somme des n éléments du tableau T
```

À l'aide de ces deux fonctions, on souhaite écrire le programme principal suivant :

```
1: Procédure principale()
2:   s := somme(tab,6)
3:   r := division(s,6)
4:   afficher "resultat : "
5:   afficher r
```

ATTENTION : Les paramètres et le résultat des fonctions `somme` et `division` sont placés dans la pile, suivant les conventions adoptées en cours.

- (b) Ecrivez le code ARM du programme principal. Les variables `s` et `r` seront réalisées avec les registres `r0` et `r2`, respectivement. Pour les affichages, vous utiliserez les fonctions fournies dans `es.s` (2,5 points)
(c) Dessinez l'état de la pile juste avant l'appel effectif (b1) de la fonction `somme` à la ligne 2. (1 point)

Voici le code ARM de la fonction `division` :

```
6:   sub sp,sp,#4
7:   str fp,[sp]
8:   mov fp,sp

9:   sub sp,sp,#4
10:  str r0,[sp]
11:  sub sp,sp,#4
12:  str r1,[sp]
13:  sub sp,sp,#4
14:  str r2,[sp]
```

```

15:      ldr r0, [fp,#12]
16:      ldr r1, [fp,#8]

17:      mov r2,#0
18: deb: cmp r0,r1
19:      blo fin
20:      add r2,r2,#1
21:      sub r0,r0,r1
22:      bal deb

23: fin: str r2,[fp,#4]

24:      ldr r2,[sp]
25:      add sp,sp,#4
26:      ldr r1,[sp]
27:      add sp,sp,#4
28:      ldr r0,[sp]
29:      add sp,sp,#4

30:      ldr fp,[sp]
31:      add sp,sp,#4

32:      mov pc,lr

```

- (d) Résumez en quelques mots à quelles étapes principales de la programmation d'une fonction correspondent les lignes suivantes : **(2 points)**
- Lignes 6 à 8.
 - Ligne 9 à 14.
 - Lignes 15 et 16.
 - Lignes 17 à 22.
 - Lignes 23.
 - Lignes 24 à 29.
 - Lignes 30 et 31.
 - Lignes 32.

Nous donnons maintenant l'algorithme de la fonction `somme` :

```

33: fonction somme(T : adresse d'un tableau, n : entier naturel) avec résultat entier naturel
34:   si n = 1 alors
35:     retourner T[0]
36:   sinon
37:     retourner T[n-1]+somme(T,n-1)
38:   fin si

```

ATTENTION : Les paramètres et le résultat de la fonction `somme` sont placés dans la pile, suivant les conventions adoptées en cours.

- (e) Donnez le code ARM de `somme`. Vous justifierez les étapes principales avec des commentaires dans le code. **(4,5 points)**

2 Processeur à accumulateur (9 points)

Dans cette partie, nous considérons un processeur à accumulateur fictif proche de celui vu lors du cours 9. La partie opérative de ce processeur à accumulateur est donnée dans la figure 1.

Adresses et données. La principale différence avec le processeur vu lors du cours est qu'ici les tailles du codage d'une **adresse** et d'une **donnée** sont différentes : **16 bits** (un demi-mot) pour **une adresse** et **8 bits** (un octet) pour **une donnée**.

$md \leftarrow mem[ma]$	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
$mem[ma] \leftarrow md$	écriture d'un mot mémoire	C'est la seule possibilité en écriture !
$rcmp \leftarrow reg_0$	affectation	reg_0 peut uniquement être acc ou md !
$reg_0 \leftarrow \#i$	affectation	$\#i$ peut être 0xFFFF ou 0 reg_0 est pc ou sp
$reg_0 \leftarrow reg_1$	affectation	reg_0 est pc , acc , sauv , ma ou md reg_1 est pc , acc , sauv , ma , md ou sp
$reg_0 \leftarrow reg_1 + 1$	incrémentement	reg_0 est pc ou sp reg_1 est pc ou sp
$sp \leftarrow sp - 1$	décrémentement	
$reg_0 \leftarrow reg_1 \text{ op } reg_2$	opération	reg_0 est pc , acc , md , ma ou sauv reg_1 est pc , acc , md , ma ou sauv reg_2 est pc , acc , md ou sauv op : +, -
$ma \leftarrow reg_0 << \#i$	décalage de i bits sur la gauche	reg_0 est md ou sauv $1 \leq \#i \leq 8$

Seul le registre **rcmp** permet de faire des tests : **rcmp** = entier (c'est donc la seule micro-condition).

Le langage. Les instructions sont décrites ci-dessous.

Code	Instruction	Signification	taille (octets)
0	$ld \#vi$	affecte acc à la valeur immédiate vi	2
1	$ld ad$	chargement de l'octet en mémoire à l'adresse ad dans acc	3
2	$add ad$	mise à jour de acc avec la somme du contenu de acc et du mot mémoire d'adresse ad	3
3	$sub ad$	mise à jour de acc avec la soustraction du contenu de acc et du mot mémoire d'adresse ad	3
4	$st ad$	rangement en mémoire à l'adresse ad du contenu de acc	3
5	$jmp ad$	saut à l'adresse ad	3
6	$jeq ad$	si acc = 0, alors saut à l'adresse ad	3
7	push	empile la valeur contenue dans acc	1
8	pop	dépile le sommet de pile dans acc	1

Les instructions sont codées sur **1, 2 ou 3 octets** chacune :

- le premier octet représente le code de l'opération ;
- le deuxième octet, s'il existe, contient l'octet de poids fort d'une adresse ou bien une constante.
- le troisième octet, s'il existe, contient l'octet de poids faible d'une adresse.

Automate d'interprétation. Un automate d'interprétation **incomplet** vous est donné dans la figure 2.

- Expliquez l'interprétation d'une instruction **ld** (vous pouvez utiliser un exemple). **(2 points)**
- Ajoutez l'interprétation de l'instruction **st** à l'automate. **(1 point)**
- Ajoutez l'interprétation de l'instruction **jeq** à l'automate. **(2 points)**

Pour interpréter les instructions **push** et **pop** vous utiliserez le registre **sp** que vous initialiserez à l'adresse 0xFFFF. De plus, vous adopterez les conventions suivantes :

- la pile évolue en direction des adresses décroissantes, et
- Le pointeur de pile (**sp**) pointe vers le sommet de pile (case pleine).

- Ajoutez l'interprétation de l'instruction **push** à l'automate. **(2 points)**
- Ajoutez l'interprétation de l'instruction **pop** à l'automate. **(2 points)**

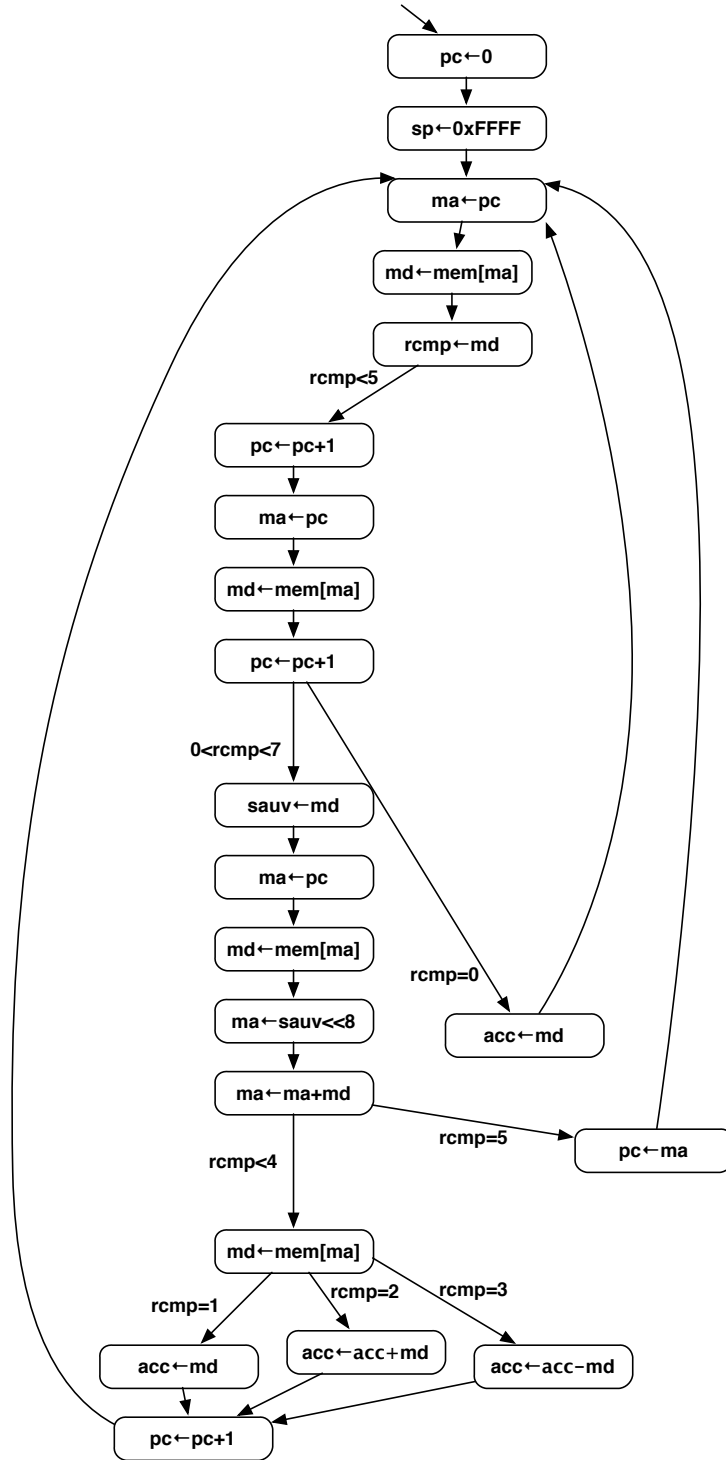


FIGURE 2 – Automate d'interprétation incomplet

3 ANNEXE I : instructions du processeur ARM

Nom	Explication du nom	Opération	Remarque
AND	AND	et bit à bit	
EOR	Exclusive OR	ou exclusif bit à bit	
SUB	SUBstract	soustraction	
RSB	Reverse SuBstract	soustraction inversée	
ADD	ADDition	addition	
ADC	ADdition with Carry	addition avec retenue	
SBC	SuBstract with Carry	soustraction avec emprunt	
RSC	Reverse Substract with Carry	soustraction inversée avec emprunt	
TST	TeST	et bit à bit	pas rd
TEQ	Test EQuivalence	ou exclusif bit à bit	pas rd
CMP	CoMPare	soustraction	pas rd
CMN	CoMpare Not	addition	pas rd
ORR	OR	ou bit à bit	
MOV	MOVe	copie	pas rn
BIC	BIt Clear	et not bit à bit	
MVN	MoVe Not	not (complément à 1)	pas rn
Bxx	Branchement		xx = condition Cf. table ci-dessous
BL	Branchement à un sous-programme		adresse de retour dans r14=LR
LDR	“load”		
STR	“store”		

L'opérande source d'une instruction MOV peut être une valeur immédiate notée #5 ou un registre noté Ri, i désignant le numéro du registre. Il peut aussi être le contenu d'un registre sur lequel on applique un décalage de k bits; on note Ri, DEC #k, avec DEC ∈ {LSL, LSR, ASR, ROR}.

4 ANNEXE II : codes conditions du processeur ARM

La table suivante donne les codes de conditions arithmétiques xx pour l'instruction de branchement Bxx.

mnémonique	signification	condition testée
EQ	égal	Z
NE	non égal	$\overline{Z}$
CS/HS	$\geq$ dans N	C
CC/LO	$<$ dans N	$\overline{C}$
MI	moins	N
PL	plus	$\overline{N}$
VS	débordement	V
VC	pas de débordement	$\overline{V}$
HI	$>$ dans N	$C \wedge \overline{Z}$
LS	$\leq$ dans N	$\overline{C} \vee Z$
GE	$\geq$ dans Z	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	$<$ dans Z	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
GT	$>$ dans Z	$\overline{Z} \wedge ((N \wedge V) \vee (\overline{N} \wedge \overline{V}))$
LE	$\leq$ dans Z	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
AL	toujours	

5 ANNEXE III : fonctions d'entrée/sortie

Nous rappelons les principales fonctions d'entrée/sortie du fichier **es.s**.

Les fonctions d'affichages :

- **b1 EcrHexa32** affiche le contenu de **r1** en hexadécimal.
- **b1 EcrZdecimal32** affiche le contenu de **r1** en décimal sous la forme d'un entier relatif de 32 bits.
- **b1 EcrZdecimal16** affiche le contenu de **r1** en décimal sous la forme d'un entier relatif de 16 bits.
- **b1 EcrZdecimal8** affiche le contenu de **r1** en décimal sous la forme d'un entier relatif de 8 bits.
- **b1 EcrNdecimal32** affiche le contenu de **r1** en décimal sous la forme d'un entier naturel de 32 bits.
- **b1 EcrNdecimal16** affiche le contenu de **r1** en décimal sous la forme d'un entier naturel de 16 bits.
- **b1 EcrNdecimal8** affiche le contenu de **r1** en décimal sous la forme d'un entier naturel de 8 bits.
- **b1 EcrChaine** affiche la chaîne de caractères dont l'adresse est dans **r1**.

Les fonctions de saisie clavier :

- **b1 Lire32** récupère au clavier un entier 32 bits et le stocke à l'adresse contenue dans **r1**.
- **b1 Lire16** récupère au clavier un entier 16 bits et le stocke à l'adresse contenue dans **r1**.
- **b1 Lire8** récupère au clavier un entier 8 bits et le stocke à l'adresse contenue dans **r1**.
- **b1 LireCar** récupère au clavier un caractère et stocke son code ASCII à l'adresse contenue dans **r1**.