

Examen UE INF241 : Introduction aux Architectures Logicielles et Matérielles

Première session 2013-2014, 20 mai 2014, 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 Structures de contrôle et fonction en ARM (12 points)

On considère la zone mémoire suivante :

```
1: .data
2: A:  .word 0
3: ch1: .asciz "azbrtybaz"
4: ch2: .asciz "azartyba"
```

- (a) Rappelez la différence entre les directives `.asciz` et `.ascii`. **(0,25 point)**
- (b) Rappelez sur combien d'octets est stocké un seul caractère. **(0,25 point)**
- (c) En supposant que la zone `.data` est stockée en mémoire à partir de l'adresse `0x8000` (en hexadécimal), donnez l'adresse en mémoire des chaînes `ch1` et `ch2`. **(0,25 point)**

On considère le programme principal suivant :

```
5: .text
6: .global main
7: main:

8: ldr r0,ptr_ch1
9: sub sp,sp,#4
10: str r0,[sp]
11: ldr r0,ptr_ch2
12: sub sp,sp,#4
13: str r0,[sp]

14: sub sp,sp, #4

15: bl foo

16: ldr r1,[sp]

17: add sp,sp,#12

18: bl EcrZdecimal32

19: bal exit

20: ptr_ch1: .word ch1
21: ptr_ch2: .word ch2
```

- (d) Justifiez l'usage des lignes 20 et 21. **(0,25 point)**
- (e) Résumez en quelques mots à quelles étapes d'un appel de fonction correspondent les lignes suivantes : **(1,25 point)**
 - Lignes 8 à 13.

- Ligne 14.
- Ligne 15.
- Ligne 16.
- Ligne 17.

- (f) Quels sont les types des différents paramètres de la fonction `foo`? **(0,25 point)**
- (g) Quel est le type de son résultat? **(0,25 point)**
- (h) En reprenant les adresses calculées pour le segment `.data` (question (c)), donnez l'état de la pile au moment de l'exécution de la ligne 15. **(0,5 point)**

Voici le code ARM de la fonction `foo` :

```

22: foo:

23: sub sp,sp,#4
24: str fp,[sp]

25: mov fp,sp

26: sub sp,sp,#4
27: str r0,[sp]
28: sub sp,sp,#4
29: str r1,[sp]
30: sub sp,sp,#4
31: str r2,[sp]
32: sub sp,sp,#4
33: str r3,[sp]

34: ldr r0,[fp,#12]
35: ldr r1,[fp,#8]

36: loop: ldrb r2,[r0]
37:     ldrb r3,[r1]
38:     cmp r2,#0
39:     beq fin
40:     cmp r2,r3
41:     bne fin
42:     add r0,r0,#1
43:     add r1,r1,#1
44:     bal loop
45: fin:
46: sub r2,r2,r3

47: str r2,[fp,#4]

48: ldr r3,[sp]
49: add sp,sp,#4
50: ldr r2,[sp]
51: add sp,sp,#4
52: ldr r1,[sp]
53: add sp,sp,#4
54: ldr r0,[sp]
55: add sp,sp,#4

56: ldr fp,[sp]
57: add sp,sp,#4

58: mov pc,lr

```

- (i) Résumez en quelques mots à quelles étapes principales de la programmation d'une fonction correspondent les lignes suivantes : **(2,25 points)**
- Lignes 23 et 24.
 - Ligne 25.
 - Lignes 26 à 33.
 - Ligne 34 et 35.
 - Lignes 36 à 46.
 - Lignes 47.
 - Lignes 48 à 55.
 - Lignes 56 et 57.
 - Lignes 58.
- (j) Pourquoi n'est-il pas nécessaire de sauvegarder `lr` dans la fonction `foo`? **(0,25 point)**
- (k) Quelle est la valeur affichée par la ligne 18 du programme principal? **(0,25 point)**
- (l) En général, que permet de faire la fonction `foo`? Détaillez en fonction des valeurs que peut renvoyer la fonction. **(0,5 point)**
- Rappel :** le code ascii du caractère 'a' est 97 en base 10.

Nous considérons maintenant la fonction récursive suivante :

```

59: fonction bar(adresse paramA, adresse paramB) résultat entier relatif
60:   si mem[paramA] = 0 ou mem[paramA] != mem[paramB] alors
61:     retourner mem[paramA] - mem[paramB]
62:   sinon
63:     retourner bar(paramA+1,paramB+1)
64:   fin si

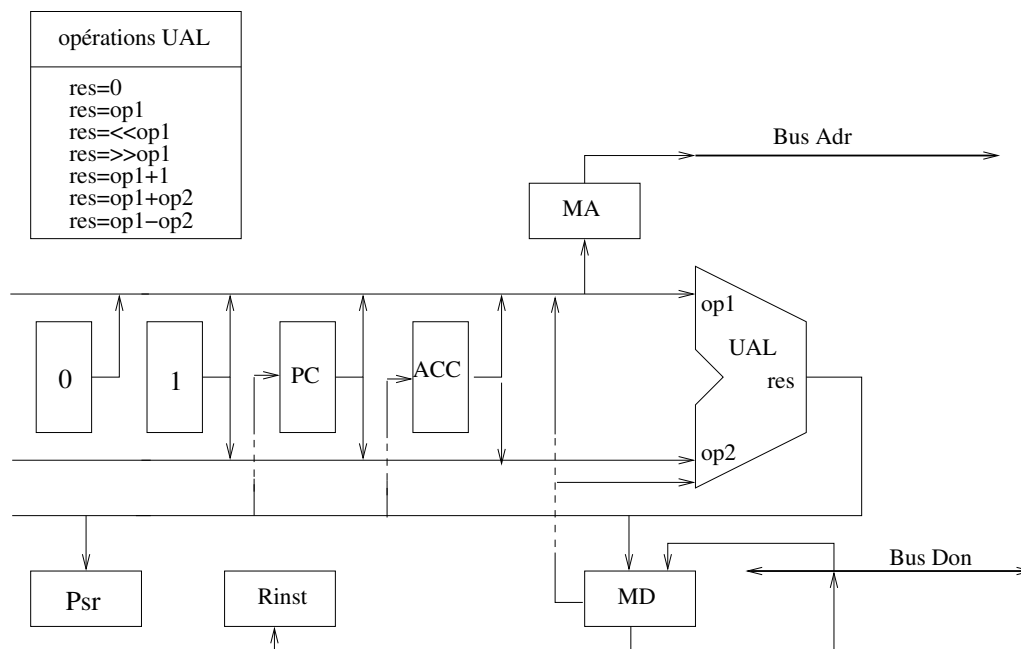
```

ATTENTION : Les paramètres et le résultat de la fonction `bar` sont placés dans la pile.

- (m) En supposant que la fonction `foo` est stockée en mémoire à l'adresse `0x9000` (en hexadécimal) et que la fonction `bar` est placée juste après, à partir de quelle adresse est stockée la fonction `bar`? **(0,5 point)**
- (n) En supposant que l'appel à la fonction `foo` dans le programme principal est remplacé par un appel à `bar`, quel est l'état de la pile lors du premier appel récursif de la fonction `bar` (c'est-à-dire, lors du premier `bl` dans `bar`)? **(1 point)**
- Le dessin de la pile n'a pas besoin de détailler la liste des registres « temporaires » sauvegardés.
- (o) Donnez le code ARM de `bar`. Vous justifierez les étapes principales avec des commentaires dans le code. **(4 points)**

2 Processeur à accumulateur (10 points)

Dans cette partie, nous enrichissons le processeur fictif vu lors du cours 9 et dont la partie opérative est représentée dans la figure ci-dessous :



Nous rappelons que ce processeur comporte un seul registre de données directement visible par le programmeur : ACC (pour accumulateur). La taille du codage d'une adresse et d'une donnée est un mot de 4 bits.

Le langage. Les instructions sont décrites ci-dessous. On donne pour chacune une syntaxe de langage d'assemblage et l'effet de l'instruction.

- `clr` : mise à zéro du registre ACC.
- `ld# vi` : chargement de la valeur immédiate `vi` dans ACC.
- `st ad` : rangement en mémoire à l'adresse `ad` du contenu de ACC.
- `jmp ad` : saut à l'adresse `ad`.
- `add ad` : mise à jour de ACC avec la somme du contenu de ACC et du mot mémoire d'adresse `ad`.

Les instructions sont codées sur **1 ou 2 mots de 4 bits** chacuns :

- le premier mot représente le code de l'opération (`clr`, `ld`, `st`, `jmp`, `add`);
- le deuxième mot, s'il existe, contient une adresse (`ad`) ou bien une constante (`vi`).

Le codage est le suivant :

<code>clr</code>	1	
<code>ld# vi</code>	2	<code>vi</code>
<code>st ad</code>	3	<code>ad</code>
<code>jmp ad</code>	4	<code>ad</code>
<code>add ad</code>	5	<code>ad</code>

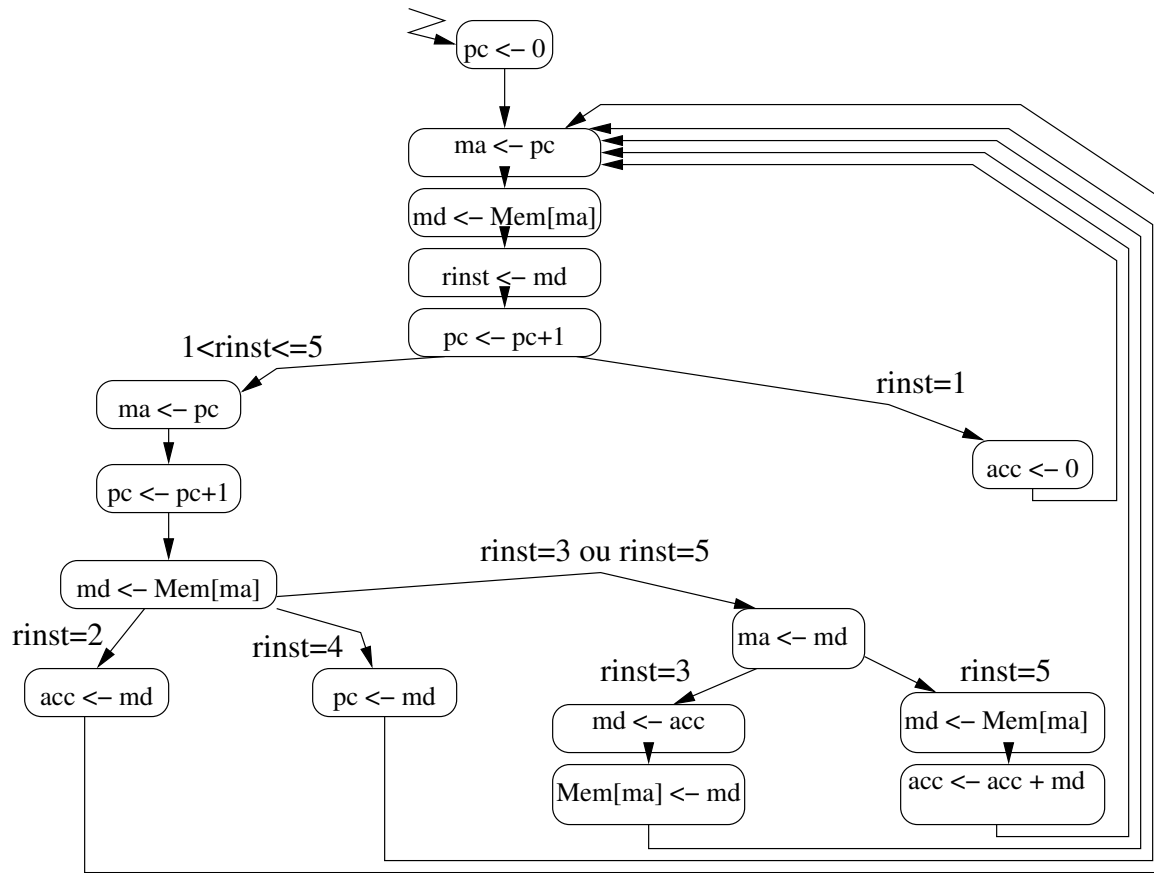


FIGURE 1 – Graphe de contrôle

Structure de la partie opérative : micro-actions et micro-conditions. Dans la partie opérative on a les registres suivants : **pc**, **acc**, **rinst**, **ma** (memory address), **md** (memory data). Les transferts possibles sont les suivants :

md ← mem[ma]	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
mem[ma] ← md	écriture d'un mot mémoire	C'est la seule possibilité en écriture !
rinst ← md	affectation	C'est la seule affectation possible dans rinst
reg₀ ← 0	affectation	reg ₀ est pc , acc , ma , ou md
reg₀ ← reg₁	affectation	reg ₀ est pc , acc , ma , ou md reg ₁ est pc , acc , ou md
reg₀ ← reg₁ + 1	incréméntation	reg ₀ est pc , acc , ou md reg ₁ est pc , acc , ou md
reg₀ ← reg₁ op reg₂	opération	reg ₀ est pc , acc , ou md reg ₁ est pc , acc , ou md reg ₂ est pc , acc , ou md op : + ou -

Pour le moment, seul le registre **rinst** permet de faire des tests : **rinst** = entier (c'est donc la seule micro-condition).

L'automate d'interprétation est donné dans la figure 1.

Enrichissement du langage. Nous souhaitons enrichir le langage de notre processeur en y ajoutant des instructions. Pour cela, nous avons besoin de micro-actions et micro-conditions supplémentaires. Nous supposons donc l'existence des indicateurs Z et C (stockés dans un registre de deux bits, **psr**), avec la même signification que dans le langage ARM. Ces deux indicateurs peuvent être testés pour faire des transitions conditionnelles dans l'automate d'interprétation (comme pour le registre **rinst**). De plus, nous ajoutons la micro-action :

setFlag reg_1, reg_2

qui met à jour les indicateurs Z et C en fonction du résultat de l'opération $reg_1 - reg_2$ (où reg_1 est **pc**, **acc**, ou **md**; et reg_2 est **pc**, **acc**, ou **md**).

Enfin, nous supposons que l'UAL peut effectuer deux nouvelles opérations : $<<$ et $>>$ où $<<$ (respectivement $>>$) permet de décaler l'opérande **op1** d'un bit sur la gauche (respectivement sur la droite).

Nouvelles instructions. Les nouvelles instructions et leur sémantiques sont données dans la table ci-dessous :

code instruction	instruction	signification	mots
6	ld <i>ad</i>	chargement du mot à l'adresse mémoire <i>ad</i> dans ACC	2
7	cmp# <i>vi</i>	mise à jour de Z et C avec le résultat de $ACC - vi$	2
8	lsr	$ACC \leftarrow ACC/2$	1
9	lsl	$ACC \leftarrow ACC \times 2$	1
10	beq <i>ad</i>	si $Z = 1$ alors $PC \leftarrow ad$	2
11	bne <i>ad</i>	si $Z = 0$ alors $PC \leftarrow ad$	2
12	bhs <i>ad</i>	si $C = 1$ alors $PC \leftarrow ad$	2
13	blo <i>ad</i>	si $C = 0$ alors $PC \leftarrow ad$	2
14	sub <i>ad</i>	mise à jour de ACC avec la différence entre le contenu de ACC et le mot mémoire d'adresse <i>ad</i>	2

Questions. On suppose que le programme suivant est stocké en mémoire à partir de l'adresse 1.

```
ld# 1
cmp# 7
blo 9
jmp 1
st 0
add 0
jmp 3
```

- Donnez l'image en mémoire (adresse et code binaire) de chaque instruction du code précédent (**2 points**).
- Que fait le code précédent (**1 point**) ?
- Donnez les modifications à apporter au graphe de contrôle donné en figure 1 afin d'interpréter les instructions suivantes :
 - lsr (**0,5 point**)
 - ld *ad* (**1 point**)
 - sub *ad* (**1 point**)
 - cmp# *vi* (**2 points**)
 - beq *ad* (**2,5 points**)

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**.

mnémorique	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**.