UNIVERSITE Joseph FOURIER, Grenoble
Licence de Sciences et technologie - Parcours MIN, INF, BIN et MIN international

Examen UE INF241 : Introduction aux Architectures Logicielles et Matérielles

Première session 2015-2016, 17 mai 2015, durée 2 h.
Documents, calculettes, téléphones portables non autorisés. Le barême est donné à titre indicatif.
En annexe, un résumé des instructions du processeur ARM et des procédures de la bibliotheque `es.s`.

1 Procédures en ARM (12 points)

On considère la zone mémoire suivante :

```
.data
CH: .asciz "entrez un nombre"
.bss
X: .word
```

(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) Rappelez la différence entre les segments `.data` et `.bss`. **(0,25 point)**
(d) En supposant que la zone `.data` est stockée en mémoire à partir de l'adresse `0x8000` (en hexadécimal) et que la zone `.bss` est stockée immédiatement après la zone `.data`, donnez l'adresse mémoire de la variable `X` en hexadécimal. **(0,5 point)**

Tou d'abord, nous supposons l'existence de la procédure `syracuse` dont le prototype est donné ci-dessous :

```
procédure syracuse(A : entier naturel sur 4 octets) sans résultat
```

Avec cette procédure, on souhaite écrire le programme principal suivant :

```
1: Procédure principale()
2:     EcrChaine("entrez un nombre")
3:     X := Lire32()
4:     syracuse(X)
```

ATTENTION : Le paramètre de la procédure `syracuse` est placé dans la pile, suivant les conventions adoptées en cours. En revanche `EcrChaine` et `Lire32` suivent les conventions adoptées dans le fichier `es.s` utilisé en TP (un rappel de ces conventions est donné en annexe du sujet).

(e) À partir du patron donné ci-dessous, écrivez le code `ARM` du programme principal. **(2,5 point)**

```
.text .global main
main:
       @ à compléter
bal exit
ptrX: .word X
ptrCH: .word CH
```

Ci-dessous nous donnons le code ARM de la procédure `foo` dont le prototype est :

`procédure foo(adresse Z d'un entier sur 4 octets) sans résultat`

ATTENTION : L'adresse Z en paramètre de la procédure `foo` est placés dans la pile, suivant les conventions adoptées en cours.

```
 5: foo:
 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,#4]
16:    ldr r1,[r0]
17:    mov r2,r1,lsl #1
18:    add r2,r2,r1
19:    add r2,r2,#1
20:    str r2,[r0]
21:    ldr r2,[sp]
22:    add sp,sp,#4
23:    ldr r1,[sp]
24:    add sp,sp,#4
25:    ldr r0,[sp]
26:    add sp,sp,#4
27:    ldr fp,[sp]
28:    add sp,sp,#4
29:    mov pc,lr
```

(f) Résumez en quelques mots à quelles étapes principales de la programmation d'une procédure correspondent les lignes suivantes : **(2 points)**
- Lignes 6 et 7.
- Ligne 8.
- Lignes 9 à 14.
- Lignes 15 à 20.
- Lignes 21 à 26.
- Lignes 27 et 28.
- Lignes 29.

(g) Pourquoi n'est-il pas nécessaire de sauvegarder `lr` dans la fonction `foo` ? **(0,5 point)**

(h) Quel calcul effectue la fonction `foo` ? **(0,5 point)**

Nous donnons maintenant l'algorithme de la procédure `syracuse` :

```
30:  procédure syracuse(A : entier naturel sur 4 octets) sans résultat
31:  b : variable locale entière naturelle sur 4 octets
32:  EcrNdecimal32(A)
33:  si A != 1 alors
35:     si A pair alors
36:        b = A / 2
37:     sinon
38:        foo(adresse de A)
39:        b = A
40:     fin si
41      syracuse(b)
42:  fin si
```

ATTENTION : Le paramètre A de la procédure `syracuse` ainsi que la variable locale `b` sont placés dans la pile, suivant les conventions adoptées en cours En revanche, `EcrNdecimal` suit les conventions adoptées dans le fichier `es.s` utilisé en TP (un rappel de ces conventions est donné en annexe du sujet).

(i) Rappelez comment détermine-t-on la parité d'un entier naturel écrit en binaire. **(0,25 point)**

(j) Donnez le code `ARM` de la procédure `syracuse`. Vous justifierez les étapes principales avec des commentaires dans le code. **(4 points)**

(k) Dessinez l'état de la pile lors du premier appel à `syracuse(5)` juste avant l'appel effectif (`bl`) de la procédure `foo` à la ligne 38. **(1 point)**

2 Processeur à accumulateur (8 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.

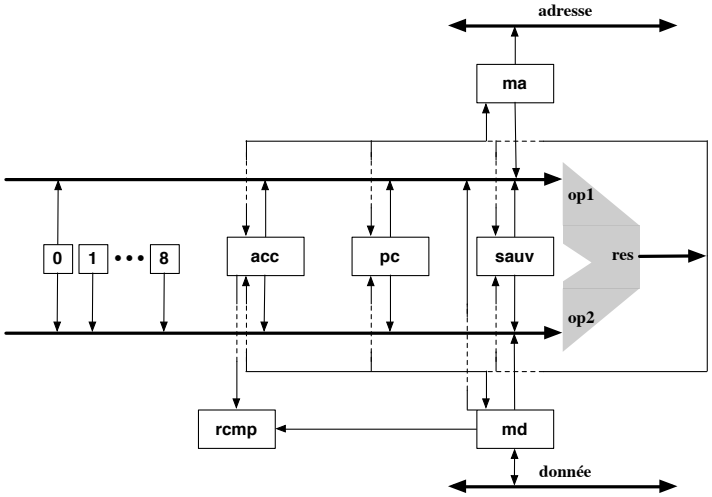

FIGURE 1 – Processeur à accumulateur

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**.

(l) Quelle est la taille maximale de la mémoire ? **(0,5 point)**

Registres. Le processeur contient les 6 registres suivants :

Nom	Taille	Visible par l'utilisateur ?	Remarque
`acc`	8	**oui**	
`md`	8	non	
`rcmp`	8	non	
`sauv`	8	non	
`pc`	16	non	compteur de programme
`ma`	16	non	

ATTENTION :

— `acc` (pour accumulateur) est le seul registre de données directement visible par le programmeur.
— Deux tailles de registres sont possibles 8 et 16 bits. Les registres de données sont sur 8 bits et les registres d'adresses sont sur 16 bits.

L'Unité Arithmétique et Logique (UAL). L'UAL comporte deux entrées de 16 bits **op1** et **op2** ainsi qu'une sortie **res** de 16 bits. Les opérations possibles sur cette UAL sont les suivantes :

Opération	**Remarque**
res ← **op1**	
res ← **op1** + **op2**	
res ← **op1** − **op2**	
res ← **op1** << **op2**	Décale **op1** de **op2** bits sur la gauche
res ← **op1** >> **op2**	Décale **op1** de **op2** bits sur la droite

Remarques :

- Les accès possibles à **op1**, **op2** et **res** sont donnés par les flèches dans la figure 1.
- Lorsque la valeur de **res** (16 bits) est envoyée vers un registre rg de 8 bits, ses 8 bits de poids faible sont affectés à rg (les 8 bits de poids fort sont perdus !).
- Les valeurs négatives sont codées en complément à deux.
- Lorsque la valeur vi d'un registre 8 bits est envoyée à **op1** ou **op2**, vi est affectée aux 8 bits de poids faibles, les bits de poids forts restants sont complétés avec la valeur du 8ème bit de vi (son bit de poids fort). Ainsi, l'UAL ne fait que des calculs signés.

Micro-actions et micro-conditions. 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 criture !
`rcmp` ← $\texttt{reg}_0$	affectation	$\texttt{reg}_0$ peut uniquement être `acc` ou `md` !
`pc` ← ♯0	initialisation	
$\texttt{reg}_0$ ← $\texttt{reg}_1$	affectation	$\texttt{reg}_0$ est `pc`, `acc`, `sauv`, `ma` ou `md` $\texttt{reg}_1$ est `pc`, `acc`, `sauv`, `ma` ou `md`
`pc` ← `pc + 1`	incrémentation	
$\texttt{reg}_0$ ← $\texttt{reg}_1$ **op** $\texttt{reg}_2$	opération	$\texttt{reg}_0$ est `pc`, `acc`, `md`, `ma` ou `sauv` $\texttt{reg}_1$ est `pc`, `acc`, `md`, `ma` ou `sauv` $\texttt{reg}_2$ est `pc`, `acc`, `md` ou `sauv` op : +, −, >> (décalage de bits à droite)
`ma` ← $\texttt{reg}_0$ << ♯i	décalage de i bits sur la gauche	$\texttt{reg}_0$ est `md` ou `sauv` $1 \leq \sharp i \leq 8$

Seul le registre `rcmp` permet de faire des tests, par exemple : `rcmp` = entier ou `rcmp` < entier .

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	`bal` dpl	saut à l'adresse $pc + dpl$	2
6	`beq` dpl	si `acc` = 0, alors saut à l'adresse $pc + dpl$	2
7	`blt` dpl	si `acc` < 0, alors saut à l'adresse $pc + dpl$	2
8	`lsr` vi	`acc` ← `acc` >> vi (`acc` est décalé de vi bits sur la droite)	2

Les instructions sont codées sur **2 ou 3 octets chacunes** :

- 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 (valeur immédiate ou déplacement).
— le troisième octet, s'il existe, contient l'octet de poids faible d'une adresse.

ATTENTION : Lors d'un saut (inconditionnel ou conditionnel), le compteur de programme `pc` **repère l'instruction qui suit le saut** (c'est-à-dire, `pc` a **deux octets d'avance** !).

(m) En supposant que l'adresse de chargement est 0x0200 (en hexadécimal), donnez la représentation hexadécimale en mémoire (adresses et instructions) du programme ci-dessous. **(2 points)**

```
      ld# 10
      st 0x0102
      ld 0x0100
bcle: sub 0x0102
      blt fin
      bal bcle
fin:  add 0x0102
      st 0x0100
```

(n) En supposant qu'initialement la valeur 45 est stockée à l'adresse 0x0100, quelle est la valeur stockée à l'adresse 0x0100 à la fin de ce programme ? **(0,5 point)**
(o) Plus généralement, quelle opération arithmétique est effectuée par ce programme sur `mem[0x0100]` si `mem[0x0100]` est initialement positif ? **(0,5 point)**

Automate d'interprétation. Un automate d'interprétation **incomplet** vous est donné dans la figure 2.
(p) Expliquez l'interprétation d'une instruction `ld` (vous pouvez utiliser un exemple). **(1 point)**
(q) Augmenter l'automate pour interpréter l'instruction `st` à l'automate. **(1,5 point)**
(r) Augmenter l'automate pour interpréter l'instruction `beq` à l'automate. **(1 point)**
(s) Augmenter l'automate pour interpréter l'instruction `lsr` à l'automate. **(1 point)**

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` $\in$ {`LSL`, `LSR`, `ASR`, `ROR`}.

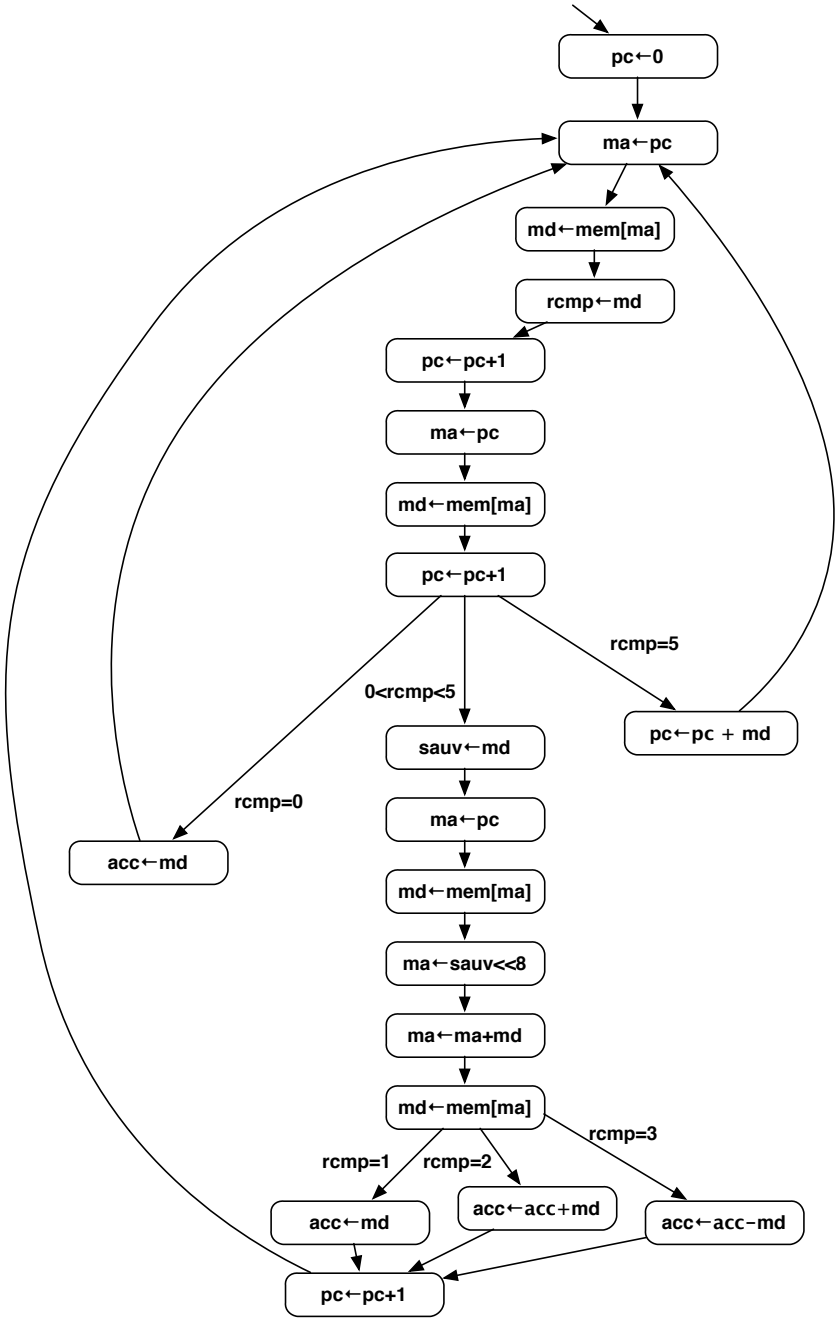

FIGURE 2 – Automate d'interprétation incomplet

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 :

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

Les fonctions de saisie clavier :

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