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

Examen UE INF451 : Introduction aux Architectures Logicielles et Matérielles

Première session 2016-2017, 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 bibliothèque `es.s`.

1 Fonctions en ARM (12 points)

On considère les zones `.data` et `.bss` suivantes :

```
.data
X: .word 15
D: .asciz "Diviseur ?"
Q: .asciz "Quotient :"
R: .asciz "Reste :"
.bss
Y: .word
Z: .word
```

(a) Rappelez la différence entre les zones `.data` et `.bss`. **(0,25 point)**
(b) Si la variable `X` avait été déclarée juste après la variable `R`, quel aurait été le problème et comment pourrait-on le résoudre ? **(0,5 point)**

Nous supposons l'existence de la fonction `division` dont le prototype est donné ci-dessous :

```
Fonction division(A : entier naturel sur 4 octets,
                  B : entier naturel sur 4 octets,
                  C : adresse d'un entier naturel sur 4 octets)
                  avec résultat entier naturel
```

(c) Rappelez l'intérêt du passage par adresse par rapport au passage par valeur. **(0,25 point)**

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

```
1: Procédure principale()
2:      EcrChaine("Diviseur ?")
3:      Y := Lire32()
4:      r0 := division(X,Y,adresse de Z)
5:      EcrChaine("Quotient :")
6:      EcrNdecimal32(r0)
7:      EcrChaine("Reste :")
8:      EcrNDecimal32(Z)
```

ATTENTION : Les paramètres et le résultat de la fonction `division` sont placés dans la pile, suivant les conventions adoptées en cours. En revanche `EcrChaine`, `EcrNdecimal32` 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). Dans ce code, `r0` est un registre, `X` est une variable entière naturelle définie dans le segment `.data`, `Y` et `Z` sont des variables entières naturelles définies dans le segment `.bss`.

(d) Ligne 4, qui est l'appelant ? qui est l'appelé ? **(0,25 point)**
(e) À partir du patron donné ci-dessous, écrivez le code `ARM` du programme principal. **(3 points)**

```
    .text
    .global main
    main:
            @ partie à compléter
            bal exit
    ptrX:  .word X
    ptrY:  .word Y
    ptrZ:  .word Z
    ptrD:  .word D
    ptrQ:  .word Q
    ptrR:  .word R
```

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

```
 9:  Fonction division(A : entier naturel sur 4 octets,
                       B : entier naturel sur 4 octets,
                       C : adresse d'un entier naturel sur 4 octets)
                       avec résultat entier naturel
10: si B > A alors
11:    mem[C] := A
12:    retourner 0
13: sinon
14:    retourner 1+division(A-B,B,C)
15: fin si
```

ATTENTION : Tous les paramètres de la fonction `division` ainsi que son résultat sont placés dans la pile, suivant les conventions adoptées en cours.

(f) Rappelez les étapes principales de la méthode de gestion des fonctions récursives vue en cours, en distinguant le rôle de l'appelant et de l'appelé. **(1,5 point)**

(g) Quel est l'intérêt de sauvegarder les registres utilisés en tant que variables temporaires ? **(0,25 point)**

(h) Donnez le code `ARM` de la fonction `division`. **(4 points)**

(i) Dessinez l'état de la pile lors du premier appel à `division(X,Y, adresse de Z)` juste avant l'exécution de la ligne 10. Vous ferez apparaître les noms de paramètres formels, effectifs ainsi que leur valeur en supposant que `Y` vaut 4 et l'adresse de `Z` est `0x8000`. Dans le dessin, différenciez (par exemple, en utilisant deux couleurs) ce qui est du rôle de l'appelant de ce qui est du rôle de l'appelé **(2 points)**

2 Processeur à accumulateur (8 points)

Dans cette partie, nous enrichissons le processeur fictif vu lors du cours 9. 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 (non signé) 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 :

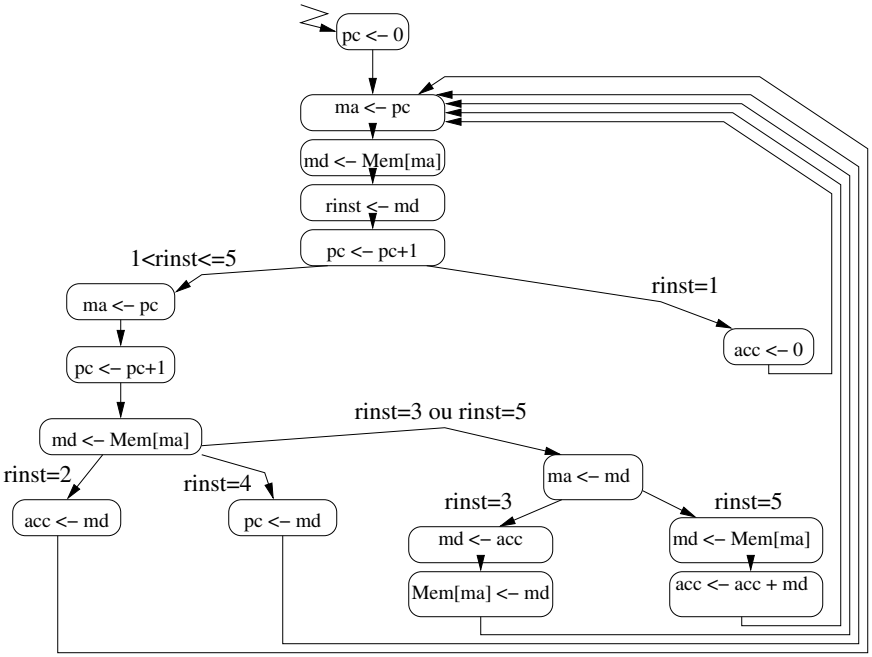


FIGURE 1 – Graphe de contrôle

`clr`	1	
`ld# vi`	2	vi
`st ad`	3	ad
`jmp ad`	4	ad
`add ad`	5	ad

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 $\leftarrow$ **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 !
rinst $\leftarrow$ **md**	affectation	C'est la seule affectation possible dans `rinst`
$\mathbf{reg_0} \leftarrow \mathbf{0}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1} + 1$	incrémentation	reg_0 est `pc`, `acc`, ou `md` reg_1 est `pc`, `acc`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1}$ **op** $\mathbf{reg_2}$	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.

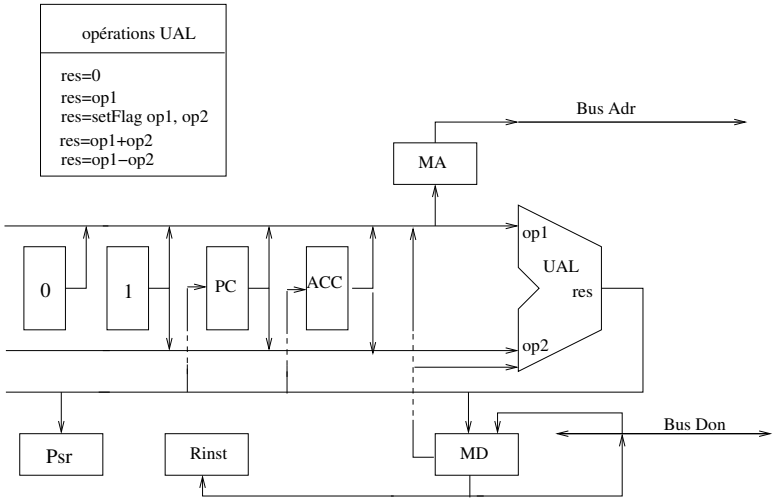
Enrichissement du langage. Nous souhaitons enrichir le langage de notre processeur en y ajoutant des instructions. Pour cela, nous avons besoin de micro-actions et micro-conditions supplémentaires. Nous supposons donc l'existence des indicateurs Z et C (stockés dans un registre de deux bits, `psr`),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

l'automate d'interprétation (comme pour le registre `rinst`). De plus, nous ajoutons la micro-action :

$$\texttt{setFlag}\ \mathrm{reg}_1,\ \mathrm{reg}_2$$

qui met à jour les indicateurs Z et C en fonction du résultat de l'opération $\mathrm{reg}_1 - \mathrm{reg}_2$ (où reg_1 est `pc`, `acc`, ou `md` ; et reg_2 est `pc`, `acc`, ou `md`).

La (nouvelle) partie opérative complète est représentée dans la figure ci-dessous :



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

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

Questions. On suppose que le programme suivant est stocké en mémoire à partir de l'adresse **0**.

```
deb: ld#  1
 si: cmp# 7
     bhs  fin
     st   X
     add  X
     jmp  si
fin: jmp  deb
  X:
```

(j) Par quelles adresses seront remplacées les étiquettes `deb`, `si`, `fin` et `X` ? **(1 point)**

(k) Que fait le code précédent ? (déroulez un exemple) **(1 point)**

(l) Modifier directement le graphe de contrôle donné en figure 1 afin d'ajouter l'interprétation des instructions suivantes :
- `ld ad` **(0,75 point)**
- `sub ad` **(0,75 point)**
- `cmp# vi` **(2 points)**
- `beq ad` **(2 points)**

(m) Si on souhaite maintenant ajouter un saut conditionnel qui s'effectue seulement lorsque la condition « supérieure stricte » est vérifiée, quelle condition sur les indicateurs faut-il vérifier ? **(0,5 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`}.

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`.