UNIVERSITE GRENOBLE ALPES
Licence de Sciences et technologie - Parcours INM, MIN

Examen UE INF401 : Architectures des Ordinateurs

Mai 2023, durée 2 h 00
Document autorisé : 1 feuille A4 Recto/Verso de notes personnelles manuscrites.
Les calculettes et téléphones portables sont interdits.
La plupart des questions sont indépendantes, si vous avez des difficultés avec une question, passez à la suivante.
Le barème est donné à titre indicatif.

1 Questions de cours (4 points)

Répondre, succinctement, en quelques lignes seulement (1 page en tout, maximum, conseillée). Les deux premières questions peuvent être répondue dans un tableau avec une ligne par classe (préciser les colonnes par des noms d'entête clairs).

(a) Donner 3 classes d'instructions de rupture de séquence. **(1 point)**
(b) Pour chacune de ces classes, indiquer la syntaxe et les éléments caractéristiques de la classe. **(2 points)**
(c) Donner un exemple d'usage représentatif et complet pour une classe. **(1 point)**

2 Programmation en langage d'assemblage ARM (11 points)

La course à 100 La course à 100 (récursive) cherche à obtenir une somme valant 100 avec une liste d'entiers donnés dans un tableau, chaque entier pouvant être utilisé au maximum une fois. Par exemple, une solution avec l'ensemble d'entiers 1 5 12 28 15 20 3 50 9 est 12+15+20+3+50.

Pour le travail demandé pendant votre examen, il n'est pas nécessaire de savoir comment résoudre ce problème ni même de comprendre comme il est résolu par le programme ci-dessous, votre travail concerne seulement la traduction systématique des programmes déjà écrits.

L'algorithme de résolution de la course à 100 est donné ci-dessous pour des entiers 32 bits (relatifs) :

```
Fonction course(TE : adresse d'un tableau d'entiers; i, taille, somme : 3 entiers) avec résultat entier
       valeur, suivant, res : 3 entiers non initialisés
       ok = 1, ko = 0 : 2 entiers initialisés
 01:     si somme == 0 :
 02:          afficheChaine("Solution : ")
 03:          res = ok
 04:     sinon-si i >= taille :
 05:          res = ko
 06:     sinon
 07:         valeur = TE[i]
 08:         suivant = i+1
 09:         si course(TE, suivant, taille, somme-valeur) == ok :
 10:             afficheEntier(valeur)
 11:             res = ok
 12:         sinon-si course(TE, suivant, taille, somme) == ok :
 13:             res = ok
 14:         sinon
 15:             res = ko
 16:         finsi
 17:     finsi
 18:     retourne res
```

La course à 100 est utilisée tout simplement de la manière suivante :

```
Programme principal
20:     afficheChaine("Entrer une liste de nombres positifs")
21:     N=LireTableauNombres(@Tab,100) //renvoie le nombre d'entiers lus
22:     afficheEntier(N)
23:     afficheChaine("nombres positifs lus")
24:     si course(@Tab,0,N,100) == 0
25:         afficheChaine("Pas de solution")
26:     finsi
```

Pour écrire les deux programmes précédents l'ébauche de code suivante est fournie :

```
  .data
MsgTab:  .asciz "Entrer une liste de nombres positifs"
MsgEnt:  .asciz "Entrer un nombre positif"
MsgLu:   .asciz "nombres positifs lus"
MsgSol:  .asciz "Solution : "
MsgEch:  .asciz "Pas de solution."
  .bss
Tab:     .skip 100*4
N:       .skip 4

  .text
  .global main

main:
        push {lr}

        @ partie à compléter

        pop {lr}
        bx lr

LD_MsgTab: .word MsgTab
LD_MsgEnt: .word MsgEnt
LD_MsgLu:  .word MsgLu
LD_MsgSol: .word MsgSol
LD_MsgEch: .word MsgEch
LD_Tab:    .word Tab
LD_N:      .word N
```

Traduction du programme principal

(d) Traduire en ARM avec les fonctions de la bibliothèque es.s (donnée en annexe) les affichages du programme principal (lignes 20, 22, 23 et 25. Indiquez les lignes ou l'affichage que vous traduisez). **(1 points)**

Attention, vous prendrez en compte les indications suivantes :

- Vous supposerez que la fonction `LireTableauNombres` existe et qu'elle est écrite avec les conventions à suivre suivantes :
  - Le premier paramètre est dans `R0`, le second dans `R1`.
  - Le résultat est dans `R0`.
- Vous supposerez que la fonction `course` sera écrite avec les conventions de traduction systématiques vues en cours.
  - Passage des paramètres et du résultat sur la pile.
  - Utilisation de la pile pour les variables locales et sauvegarde/restitution des valeurs des registres temporaires utilisés.

(e) Traduire en ARM l'appel ligne 21. **(2 points)**

(f) Traduire en ARM la fin du programme principal avec l'appel à `course` (lignes 24 et suivantes). **(2 points)**

Traduction de la fonction `course`

(h) Dessiner la pile au début de l'exécution du corps de la fonction `course` (ne pas oublier les actions menées dans le prologue de la traduction de la fonction et prévoir l'utilisation des 2 registres temporaires) **(1 point)**

(i) Traduire en ARM le début du corps de la fonction (lignes 1 à 3, ajouter une étiquette pour la ligne 4). **(1 point)**

(j) Traduire en ARM l'accès au tableau (ligne 7). **(1 point)**

(k) Traduire en ARM le début de la conditionnelle imbriquée avec l'appel récursif à `course` (ligne 9 et 10, ajouter une étiquette pour la ligne 12). **(2 points)**

(l) Traduire en ARM le prologue, la ligne 18 et l'épilogue de la fonction. **(1 point)**

3 Automate, microprogrammation et processeur (5 points)

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

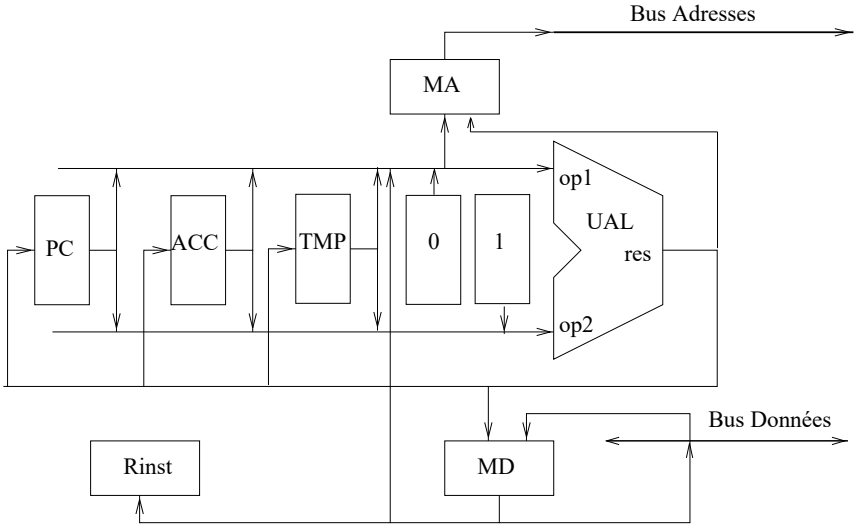

FIGURE 1 – Partie opérative du processeur

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) et `TMP`. 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 spécifique	Affectation spécifique à `Rinst`
reg_0 ← reg_0 op 1	incrémentation ou décrémentation spécifique	Incrémentation ou décrémentation hors UAL spécifique à certains registres spéciaux reg_0 est `PC` ou `MA` op : + ou −
reg_0 ← 0	mise à zéro	reg_0 est `PC`, `ACC`, ou `TMP`
reg_0 ← reg_1	affectation	reg_0 est `PC`, `ACC`, `TMP`, `MA`, ou `MD` reg_1 est `PC`, `ACC`, `TMP`, ou `MD`
reg_0 ← reg_1 op reg_2	opération	reg_0 est `PC`, `ACC`, `TMP`, `MA`, ou `MD` reg_1 est 0, `PC`, `ACC`, `TMP`, ou `MD` reg_2 est 1, `PC`, `ACC`, `TMP`, ou `MD` op : + ou −

Pour permettre la mise en place de choix dans l'automate de contrôle, la partie opérative peut faire des tests sur le registre `Rinst`, de la forme : `Rinst` = entier.

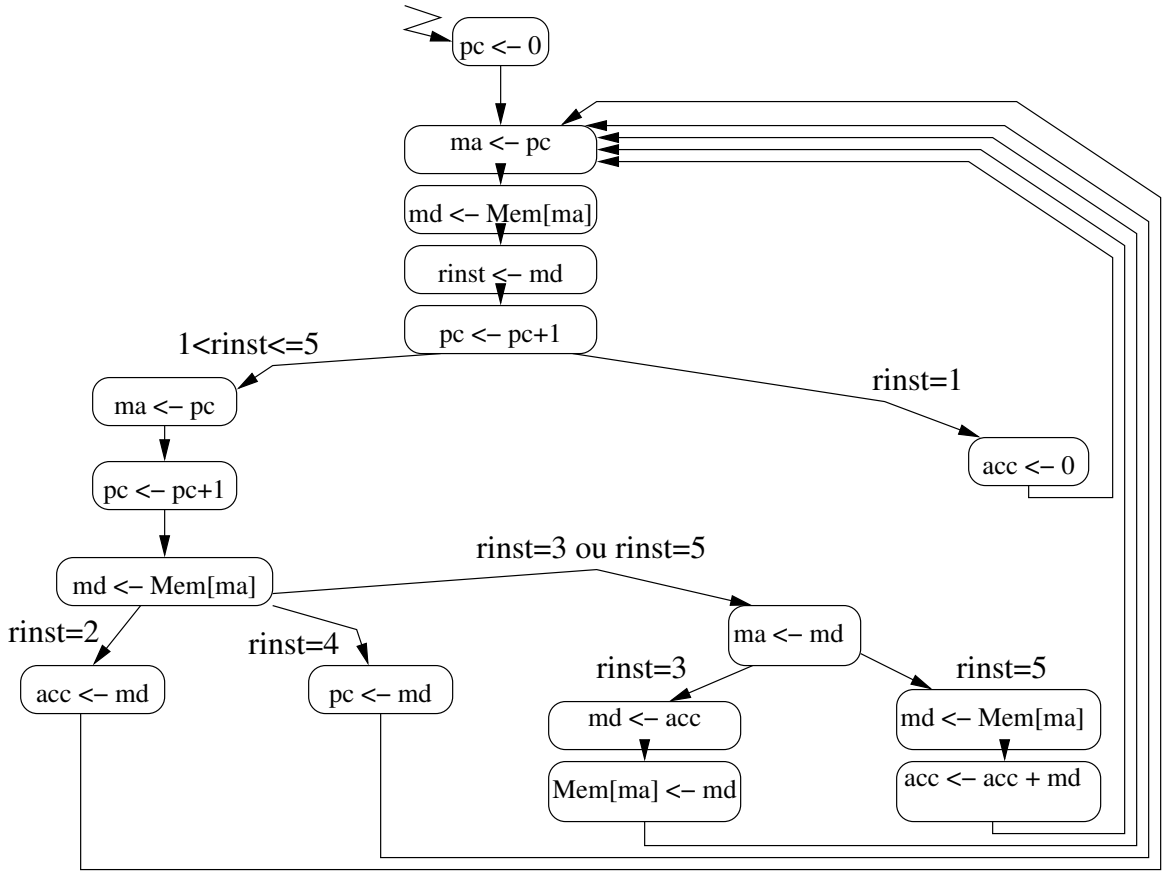

FIGURE 2 – Graphe de contrôle

Le langage d'assemblage. 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**. La mémoire comporte donc 16 mots adressables.

Les instructions sont décrites ci-dessous. On donne pour chacune une syntaxe de langage d'assemblage, le code machine, la sémantiques et la taille du codage :

instruction	code	signification	mots
`clr`	1	mise à zéro du registre `ACC`	1
`ld vi`	2	chargement de la valeur immédiate `vi` dans `ACC`	2
`st ad`	3	rangement en mémoire à l'adresse `ad` du contenu de `ACC`	2
`jmp ad`	4	saut inconditionnel à l'adresse `ad`	2
`add ad`	5	mise à jour de `ACC` avec la somme de `ACC` et de la valeur à l'adresse `ad`	2

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

— 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*).

L'automate d'interprétation de ce langage est donné dans la figure 2.

État initial de la mémoire. On suppose que le programme suivant est stocké en mémoire, la zone `.text` commence à l'adresse **0** et la zone `.data` commence à l'adresse **14**.

```
  .text
l.0  main: ld 6
l.1  rep:  add A
l.2        add B
l.3        st A
l.4        clr
l.5        jmp rep
```

```
1.6    .data
1.7  A: .valeur 4
1.8  B: .valeur -2
```

Questions.

(m) Donnez l'état initial de la mémoire en binaire (hexadécimal). **(1 point)**

(n) Simulez l'exécution du début de ce programme (au niveau assembleur). Donnez les valeurs de l'accumulateur jusqu'à ce que l'accumulateur atteigne la valeur 2 (s'il l'atteint, sinon indiquer que le programme ne l'atteint pas). Pour répondre, vous remplirez un tableau de simulation similaire à celui défini annexe 1 (une ligne par instruction, environ 20 lignes, ligne 0 exclue, la ligne 1 est en partie donnée.) **(1 point)**

(o) Simulez, en suivant le graphe de contrôle de la figure 2, l'exécution au niveau des micro-actions du début du programme stocké en mémoire. Pour répondre, vous remplirez un tableau de simulation similaire à celui donné à l'annexe 2 (une ligne par micro-action, environ 20 lignes, ligne 0 exclue, la ligne 1 est en partie donnée.) **(1 points)**.

Enrichissement du langage. Nous souhaitons enrichir le langage de notre processeur en ajoutant deux instructions d'échange entre la mémoire et l'accumulateur.

Sémantique opérationnelle des instructions à ajouter. Les instructions à ajouter, leur code, leur sémantiques et la taille de leur codage sont données dans la table ci-dessous :

instruction	**code**	**signification**	**mots**
`dup ad`	6	duplique la valeur à l'adresse **ad** vers l'adresse **ad+1**	2
`swp ad`	7	échange la valeur à l'adresse **ad** et la valeur à l'adresse **ad+1**	2

Questions.

(p) Donnez les modifications à apporter à l'automate fourni par le graphe de contrôle de la figure 2 afin d'interpréter l'instruction `dup`. **(1 point)**

(q) Donnez les modifications à apporter à l'automate fourni par le graphe de contrôle de la figure 2 afin d'interpréter l'instructions `swp`. **(1 point)**

— Indication : vous pouvez utiliser le registre `TMP` pour des calculs intermédiaires. Si vous avez besoin de plus de registres pour des calculs intermédiaires le signaler et le justifier.

4 ANNEXE 1 : Tableau pour l'exécution au niveau assembleur

	instruction (exécutée)	Acc	A	B	prochain PC	Commentaires
0		?	4	-2	0	
1	ld 6					
2						
3						

5 ANNEXE 2 : Tableau pour l'exécution au niveau de l'automate de contrôle

	Micro-Action (exécutée)	PC	Rinst	ACC	MA	MD	Mem[14]	Mem[15]	Commentaires
0		?	?	?	?	?	4	-2	
1	pc $\leftarrow$ 0								
2									
3									

6 ANNEXE 3 : fonctions d'entrée/sortie

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

- `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 EcrNdecimal32` affiche le contenu de `r1` en décimal sous la forme d'un entier naturel de 32 bits.
- `bl EcrChaine` affiche la chaîne de caractères dont l'adresse est dans `r1`.
- `bl EcrCar` affiche le caractère dont le code ASCII est dans `r1`.
- `bl Lire32` récupère au clavier un entier 32 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`.

7 ANNEXE 4 : instructions du processeur ARM

Nom	Explication du nom	Opération	Remarque
`AND`	AND	et bit à bit	
`SUB`	SUBstract	soustraction	
`RSB`	Reverse SuBstract	soustraction inversée	
`ADD`	ADDition	addition	
`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`
`MUL`	MULtiplication	multiplication tronquée	pas `r15`
`B**`	Branch	rupture de séquence conditionnelle	** = condition, cf. ci-dessous
`BL`	Branch and Link	appel sous-programme	adresse de retour dans r14
`LDR`	Load Register	lecture mémoire	
`STR`	Store Register'	écriture mémoire	

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`}.

8 ANNEXE 5 : codes conditions du processeur ARM

La table suivante donne quelques codes de conditions arithmétiques `**` pour l'instruction de rupture de séquence `B**`.

mnémonique	signification	condition testée
EQ	égal	Z
NE	non égal	$\overline{Z}$
GE	$\geq$ dans Z	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	$<$ dans Z	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
AL	toujours	