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

Examen UE INF401 : Architectures des Ordinateurs

Mai 2024, 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 (3 points)

Répondre, succinctement, en quelques lignes seulement (1 page en tout, maximum, conseillée) et, si nécessaire, avec des schémas explicatifs commentés.

(a) Rappeler le cycle de base d'exécution des instructions assembleur par le processeur (en 4 temps +/-1 temps [selon le niveau de détail que vous adopterez]). **(1 point)**
(b) Décrire l'optimisation dite d'exécution en "pipe-line" associée à ce cycle, les contraintes matérielles à satisfaire pour qu'elle soit possible et le gain espéré. Explication et schéma. **(1 point)**
(c) En pratique, le gain obtenu par cette optimisation matérielle est souvent plus faible, pourquoi ? Indiquer les facteurs limitant majeurs. **(1 point)**

2 Programmation en langage d'assemblage ARM (10 points)

Important : lire le sujet de l'exercice en entier avant de commencer à rédiger les réponses.

Distance d'édition de Levenshtein La distance de Levenshtein entre deux chaînes de caractères ou tableaux de caractères est égale au nombre minimal de caractères qu'il faut supprimer, ajouter ou modifier pour passer d'une chaîne à l'autre. Ex. entre INM et MIN, il faut au moins 2 opérations (ajouter M au début, supprimer M à la fin).

La distance de Levenshtein peut être définie récursivement par :

— DistLev(a,b) = Max(|a|,|b|), si l'un des textes est vide,
— DistLev(a,b) = DistLev(a-1,b-1), si les 2 textes commencent par le même caractère,
— DistLev(a,b) = 1 + Min (DistLev(fin(a),b), DistLev(fin(a),fin(b)), DistLev(a,fin(b))), sinon.

Ce qui peut se traduire par :

```
Fonction DistLev(a, b : deux tableaux de caractères ) avec résultat entier
         loc, alt : entier
 1:     si (a[0]+b[0]) == 0 alors loc = 0
 2:     sinon si a[0] == 0 alors loc = 1 + DistLev(a, adresse de b[1])
 3:     sinon si b[0] == 0 alors loc = 1 + DistLev(adresse de a[1], b)
 4:     sinon si a[0] == b[0] alors loc = DistLev(adresse de a[1], adresse de b[1])
 5:     sinon
 6:          loc = 1 + DistLev(adresse de a[1], b)
 7:          alt = 1 + DistLev(adresse de a[1], adresse de b[1])
 8:          si alt < loc alors loc = alt finsi
 9:          alt = 1 + DistLev(a, adresse de b[1])
10:          si alt < loc alors loc = alt finsi
11:     finsi ... finsi
12:     retourne loc
```

Cette fonction est utilisée tout simplement de la manière suivante :

```
Programme principal
20:     EcrChaine("Entrer un mot")
21:     LireChaine(&ch1)
22:     EcrChaine("Entrer un mot")
23:     LireChaine(&ch2)
24:     d:=DistLev(ch1,ch2)
25:     EcrChaine("Distance entre les 2 mots ")
26:     EcrNdecimal32(d)
```

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

```
       .data
msg1: .asciz "Entrer un mot"
msg2: .asciz "Distance entre les 2 mots "
d:       .word 0
       .bss
ch1: .skip 80
ch2: .skip 80
       .text
       .global main
main: push {lr}

        @ partie à compléter

        pop {lr}
        bx lr
LD_d: .word d
LD_ch1: .word ch1
LD_ch2: .word ch2
LD_msg1: .word msg1
LD_msg2: .word msg2
```

Traduction d'une partie du programme principal

(d) Traduire en ARM avec les fonctions de la bibliothèque es.s (donnée en annexe) les entrées/sorties, lignes 20, 21, 25 et 26. **(2 points)**

(e) Compléter avec la traduction de la ligne 24 **(2 points)**

Attention, vous prendrez en compte les indications suivantes :

- Vous supposerez que la fonction `DistLev` existe et qu'elle est écrite suivant **la méthode systématique vue en cours** (c-à-d, avec ses paramètres et son résultat placés par la pile dans cet ordre avant l'appel).
- Pour les fonctions `LireChaine(), EcrNdecimal()` et `EcrChaine`, vous appliquerez les conventions de `es.s` utilisées en TP notamment, (*cf.* annexe).
- Vous ferez apparaitre en commentaires (`@`) dans votre code les étapes principales (vues en cours) de l'appel de la fonction DistLev.

Traduction d'une partie du calcul de la distance d'édition de Levenshtein

- Conseil : Dessiner la pile au début de l'exécution du corps de la fonction DistLev (ne pas oublier les actions menées lors de l'appel et dans le prologue de la traduction de la fonction)

(f) Donner le prologue de la fonction. **(1 point)**

(g) Traduire la ligne 1 (en ajoutant et utilisant des étiquettes ligne.2 et ligne.11) **(2 points)**

(h) Traduire la ligne 9 **(2 points)**

(i) Donner la ligne 12 et l'épilogue de la fonction. **(1 point)**

Rappel, vous prendrez en compte les indications suivantes conséquences de la méthode systématique de traduction vue en cours :

— Les paramètre sont placés par la pile dans l'ordre de lecture de l'entête.
— La valeur de retour de la fonction se trouve ensuite dans la pile.
— Les variables locales `loc` et `alt` doivent ensuite être stockées dans la pile.
— Pour l'utilisation de registres temporaires, vous prendrez `r0`, `r1` et `r2`, qui devront être sauvegardés en pile avant utilisation.
— Vous ferez apparaitre en commentaires (`@`) dans votre code les étapes principales (vues en cours) de la traduction de la fonction DistLev.

3 Automate, microprogrammation et processeur (6 points)

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

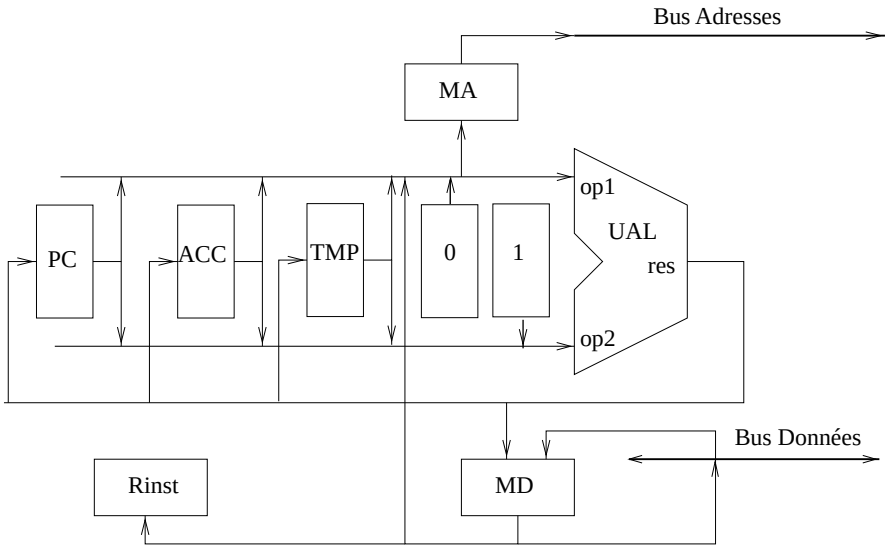

FIGURE 1 – Partie opérative du processeur

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	Affectation spécifique à `Rinst`
PC ← PC + 1	incrémentation	Incrémentation spécifique à `PC`
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$_1$ <<	décalage à gauche d'un bit	reg$_0$ est `PC`, `ACC`, `TMP`, ou `MD` reg$_1$ est `PC`, `ACC`, `TMP`, ou `MD`
reg$_0$ ← reg$_1$ op reg$_2$	opération	reg$_0$ est `PC`, `ACC`, `TMP`, ou `MD` reg$_1$ est 0, `PC`, `ACC`, `TMP`, ou `MD` reg$_2$ est 1, `PC`, `ACC`, `TMP`, ou `MD` op : + ou −
reg$_0$ ← (reg$_1$ <<) op reg$_2$	opération avec décalage	reg$_0$ est `PC`, `ACC`, `TMP`, ou `MD` reg$_1$ est `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, le registre `Rinst` peut servir à faire des tests : `Rinst` == entier, de même pour l'accumulateur avec des tests : `ACC` == entier.

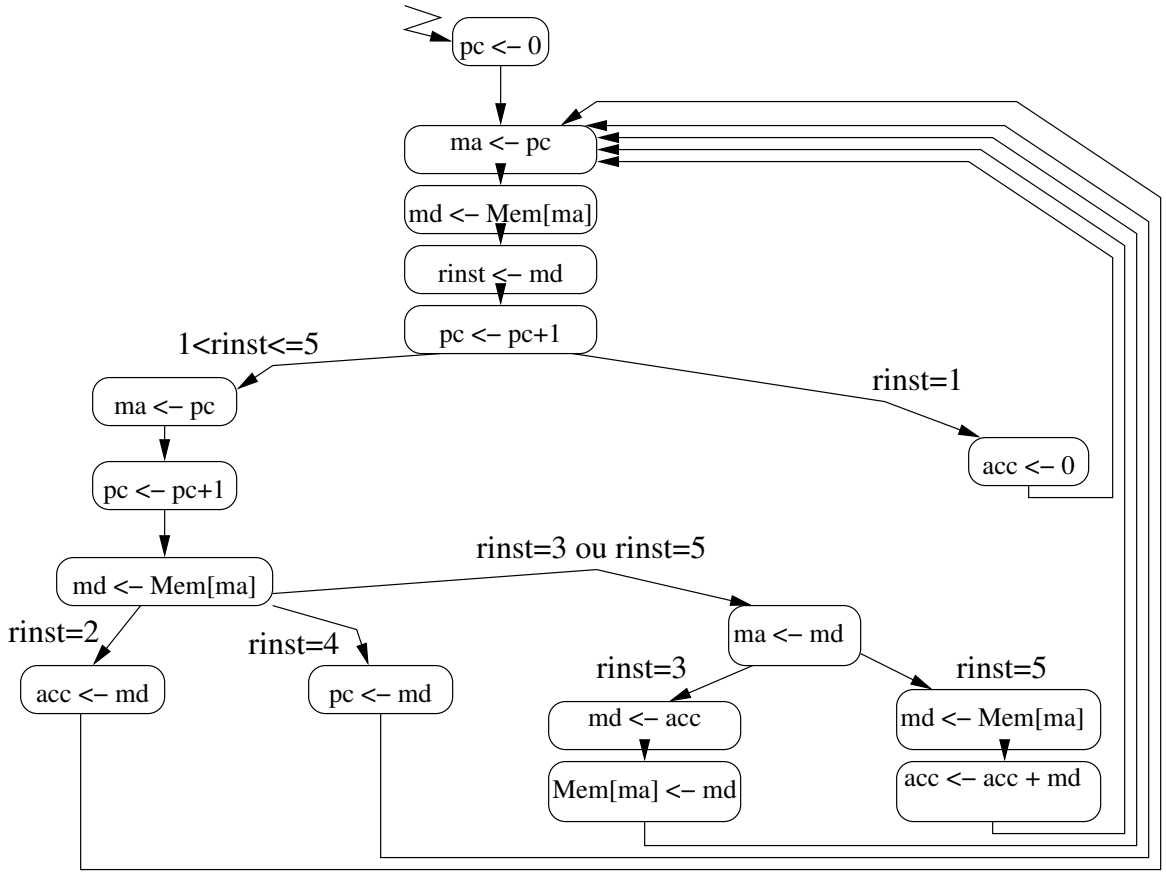

FIGURE 2 – Graphe de contrôle

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.

Enrichissement du langage. Nous souhaitons enrichir le langage de notre processeur en ajoutant deux instructions d'accès à la mémoire avec déplacement.

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**
`ldd dep`	6	l'accumulateur reçoit la valeur à l'adresse **ACC+dep**	2
`std dep, vi`	7	la mémoire à l'adresse **ACC+dep** reçoit la valeur **vi**	3

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
main: ld 1
      add  A
bcl:  add  B
      st  B
      jmp bcl
  .data
A: .valeur 4
B: .valeur 9
```

Questions.

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

(k) 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 0 (s'il l'atteint, sinon indiquer que le programme ne l'atteint pas). **(1 point)**

(l) 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 défini ci-après (une ligne par micro-action, environ 20 lignes, ligne 0 exclue, la ligne 1 est donnée.) **(2 points)**.

(m) Donnez les modifications à apporter à l'automate fourni par le graphe de contrôle de la figure 2 afin d'interpréter les instructions supplémentaires `ldd` et `std`. **(2 points)**
Indication : vous pouvez utiliser le registre `TMP` pour des calculs intermédiaires et si vraiment nécessaire d'autres temporaires `TMP2`, `TMP3`, etc.

	Micro-Action (exécutée)	PC	Rinst	ACC	MA	MD	Mem[15]	Commentaires
0		?	?	?	?	?	81	
1	pc ← 0	0						
2								
3								
4								

4 ANNEXE 0 : fonctions d'entrée/sortie

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

- `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 LireCar` récupère au clavier une chaine de caractères terminée par un espace et la stocke à partir de l'adresse contenue dans `r1`.

5 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`
`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`}.

6 ANNEXE II : codes conditions du processeur ARM

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

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	