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

Examen UE INF401 : Architectures des Ordinateurs

Mai 2022, 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 sur la mémoire (4 points)

(a) Rappeler le schéma d'une machine de von Neumann. Commentez votre schéma en répondant aux questions et indications suivantes. Dans les différentes parties de ce schéma, quels sont les éléments qui permettent de mémoriser de l'information ? Indiquer les caractéristiques communes et les caractéristiques qui distinguent ces éléments. **(2 points)**

(b) Pour cette question, nous nous placerons dans le cas d'une architecture et de programmes écrits en ARM. Faire un nouveau dessin pour préciser le lien entre les différentes zones de la mémoire vive (RAM) et les éléments d'un programme (ARM) **(2 points)** :
- Prendre appui sur un exemple de programme ARM simple de votre choix, par exemple, un programme permettant de faire transiter des données entre les divers éléments stockant de l'information.
- Ajouter explicitement des indications dans votre dessin permettant de faire le lien entre les éléments de mémoire et les éléments de programme.

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

Fonction 91 de McCarthy L'algorithme de la fonction récursive 91 de McCarthy est donné ci-dessous pour des entiers 32 bits (relatifs) :

```
Fonction f91(n : entier) avec résultat entier
         loc : entier
 1:      si n > 100 alors
 2:           loc = n-10
 3:      sinon
 4:           loc = f91(n+11)
 5:           loc = f91(loc)
 6:      finsi
 7:      retourne loc
```

La fonction f91 est utilisée tout simplement de la manière suivante :

```
Programme principal
10:      EcrChaine("Entrer un nombre")
11:      Lire32(&x)
12:      y:=f91(x)
13:      EcrNdecimal32(y)
```

Bonus (à faire en dernier, s'il vous reste du temps)
- Pour comprendre le nom de la fonction, calculer f91(1), f91(91), f91(100).

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

```
.data
        m: .asciz "Entrer un nombre"
.bss
        x: .word
        y: .word

.text
       .global main

main:
        push {lr}

        @ partie à compléter

        pop {lr}
        bx lr

ptr_m: .word m
ptr_x: .word x
ptr_y: .word y
```

Traduction du programme principal

(c) Traduire en ARM avec les fonctions de la bibliothèque es.s (donnée en annexe) les entrées/sorties, lignes 10, 11 et 13. **(2 points)**

(d) Compléter avec la traduction de la ligne 12 **(2 points)**

Attention, vous prendrez en compte les indications suivantes :

- Vous supposerez que la fonction `f91` existe et qu'elle est écrite suivant **la méthode systématique vue en cours** (c-à-d, avec ses paramètres et son résultat passés par la pile).
- Pour les fonctions `Lire32(), 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 91 de McCarthy.

Traduction de la fonction f91 de McCarthy

- Conseil : Dessiner la pile au début de l'exécution du corps de la fonction f91 de McCarthy (ne pas oublier les actions menées dans le prologue de la traduction de la fonction)

(e) Traduire la ligne 2. **(1 point)**

(f) Traduire les lignes 1, 3 et 6 (vous indiquerez les lignes 2, 4 et 5 par un commentaire @ ici ligne 2 (4 ou 5)) **(1 point)**

(g) Donner le prologue, la ligne 7 et l'épilogue de la fonction. **(2 points)**

(h) Traduire les lignes 4 et 5. **(2 points)**

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

- Le paramètre `n` doit être passé par la pile.
- La valeur de retour de la fonction doit aussi être passée par la pile.
- La variable locale `loc` doit être stockée dans la pile.
- Pour les variables temporaires vous utiliserez les registres`r0`, `r1` et `r2`, qui devront être sauvegardés en pile avant utilisation, puis restaurés suivant la convention du cours.

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 :

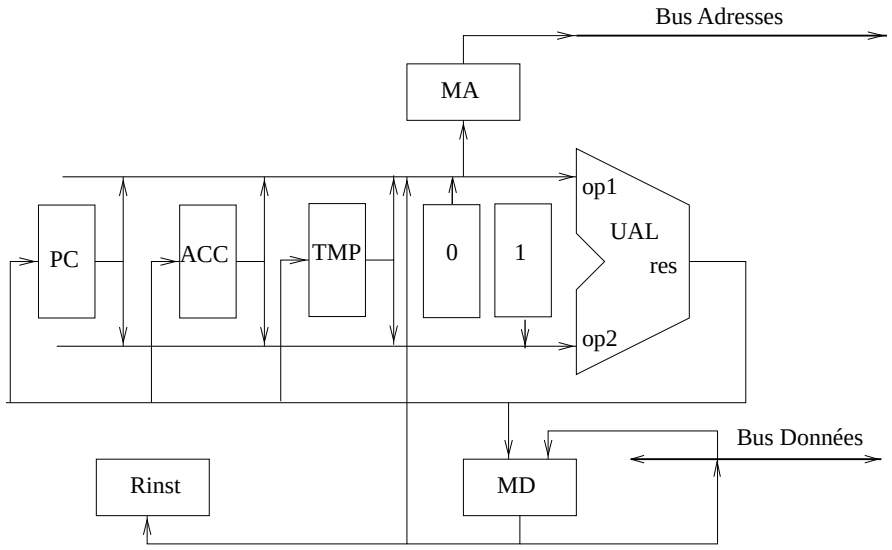

FIGURE 1 – Partie opérative du processeur

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` permet de faire des tests : `Rinst` = entier et l'accumulateur peut être évalué `acc` = entier.

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 :

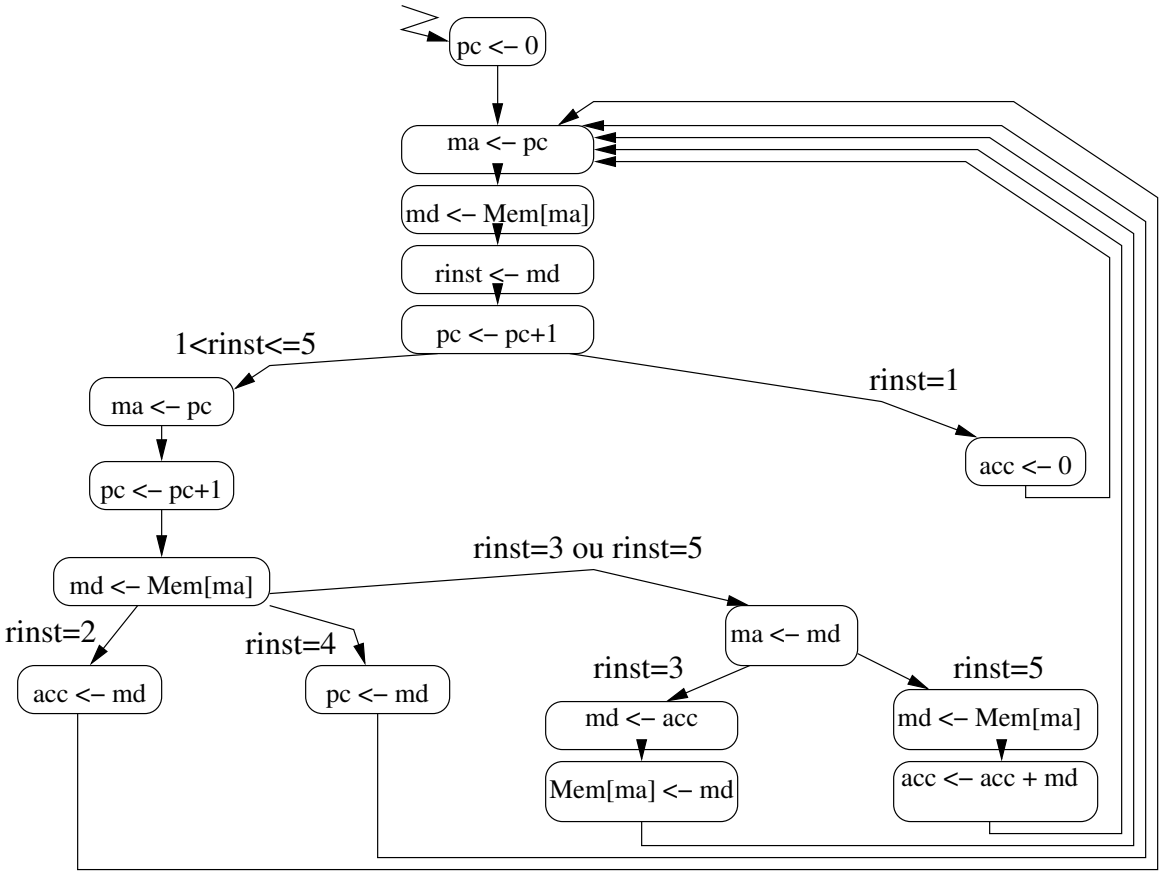

FIGURE 2 – Graphe de contrôle

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'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
`swp ad`	6	échange l'accumulateur et la valeur à l'adresse **ad**	2
`sne ad`	7	échange l'accumulateur et la valeur à l'adresse **ad** si l'accumulateur n'est pas nul	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 **10**.

Adresse	Valeur en mémoire
0	4
1	4
2	2
3	2
4	1
5	5
6	15
7	4
8	5
9	1
10	0
11	3
12	6
13	9
14	12
15	15

Questions.

— Conseil : Traduisez l'état initial de la mémoire en instructions lisibles (décompilez le binaire). Pour la syntaxe des zones, étiquettes, commentaires, pseudo-instruction, directives, etc. vous pouvez vous inspirer du langage ARM.

(i) Simulez l'exécution du début de ce programme au niveau assembleur en remplissant un tableau de simulation adapté. En particulier, vous suivrez la valeur de l'accumulateur. (une ligne par instruction, environ 7 lignes.) **(2 points)**

(j) 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 15 lignes, ligne 0 exclue, la ligne 1 est donnée.) **(2 points)**.

(k) Donnez les modifications à apporter à l'automate fourni par le graphe de contrôle de la figure 2 afin d'interpréter les instructions supplémentaires `swp` et `sne`. **(2 points)** Indication : vous pouvez utiliser le registre `TMP` pour des calculs intermédiaires et des transitions contraintes par les valeurs de l'accumulateur.

	Micro-Action (exécutée)	PC	Rinst	ACC	MA	MD	Mem[15]	Commentaires
0		?	?	?	?	?	15	
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 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`.

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	