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

Examen UE INF401 : Architectures des Ordinateurs

Mars 2018, durée 2 h 00
Document autorisé : 1 feuille A4 Recto/Verso de notes 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 Programmation en langage d'assemblage ARM (13 points)

Le but de cet exercice est d'évaluer un polynôme $a_0 + a_1x + ...a_nx^n$ avec une entrée x donnée. Les coefficients entiers $a_0, ..., a_n$ seront récupérés au clavier et stockés dans un tableau `POLY` : pour tout $i = 0, ..., n$, `POLY`$[i]$ contiendra a_i.

Pour cela, on considère les zones `.data` et `.bss` suivantes :

```
   .data
Q1:   .asciz "degre du polynome ?"
Q2:   .asciz "entrez les coefficients"
Q3:   .asciz "X ?"
E:    .asciz "degre trop grand !"
R:    .asciz "resultat :"

  .bss
DEG:  .byte
  .balign 4
POLY: .skip 40
X:    .word
```

Questions.

(a) Le tableau `POLY` est un tableau de 10 éléments. Chaque élément est un double-mot, un mot, un demi-mot, ou un octet ? **(0.25 point)**

(b) Justifiez l'emploi de la directive `.balign` **(0.25 point)**

(c) Rappelez la différence entre les zones `.data` et `.bss`. **(0,5 point)**

Voici la section `.text` à compléter dans les questions suivantes :

```
   .text
   .global main
main:
          @ partie à compléter
          bal exit
ptrQ1:   .word Q1
ptrQ2:   .word Q2
ptrQ3:   .word Q3
ptrE:    .word E
ptrDEG:  .word DEG
ptrPOLY: .word POLY
ptrX:    .word X
ptrR:    .word R
```

1.1 Itération et tableau

On commence par saisir au clavier le degré DEG du polynôme, puis on récupère les coefficients $a_0, ..., a_{\texttt{DEG}}$ en effectuant l'algorithme suivant.

```
Procédure principale()
   EcrChaine("degre du polynome ?")        // Q1
   Lire8(DEG)
   Si DEG >= 10 alors
      EcrChaine("degre trop grand !")      // E
   Sinon
      EcrChaine("entrez les coefficients") // Q2
      Pour i de 0 à DEG
           Lire32(POLY[i])
      FinPour

      @ la suite dans les prochaines questions

   FinSi
```

(d) Traduisez en ARM l'algorithme ci-dessus en tenant compte des indications suivantes :

- EcrChaine, Lire8 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).
- La variable DEG et le tableau POLY sont définis dans le segment .bss
- Pour le reste (en particulier i), vous utiliserez des registres.
- Indiquez en début de programme votre utilisation des registres, par exemple : `@ var i: reg R2`

(3 points)

1.2 Appel de fonction

Dans cette partie, vous supposez l'existence de la fonction

```
int exp(x : entier naturel, y : entier naturel);
```

Cette fonction retourne la valeur x^y. Les paramètres et le résultat de la fonction exp sont placés dans la pile, suivant les conventions adoptées en cours.

Nous allons maintenant compléter la procédure principale avec l'algorithme suivant.

```
   EcrChaine("X ?")                  // Q3
   Lire32(X)
   r := 0
   Pour i de 0 à DEG
        p :=  exp(X,i)
        r := r + POLY[i] * p
   Fin Pour
   EcrChaine("resultat :")           // R
   EcrNdecimal32(r)
```

(e) Traduisez en ARM la ligne avec appel de la fonction exp de l'algorithme ci-dessus (i.e. : `p := exp(X,i)`) en tenant compte des indications suivantes :

- La variable X est définie dans le segment .bss
- Pour le reste (en particulier i et p), vous utiliserez des registres (i dans R2 et p dans R3).
- De plus, nous rappelons que les paramètres et le résultat de la fonction exp sont placés dans la pile, suivant les conventions adoptées en cours.

(4 points)

1.3 Programmation d'une fonction récursive

Voici l'algorithme de la fonction `exp`.

```
int exp(x : entier naturel, y : entier naturel)
    Si y = 0 alors
       retourner 1
    Sinon
       r := exp(x,y/2)
       r := r * r
       Si y est impair alors
          r := r * x
       FinSi
       retourner r
```

(f) Traduisez en `ARM` la fonction `exp` donnée ci-dessus en tenant compte des indications suivantes :

- Nous rappelons que vous devez supposé que l'appelant a stocké les paramètres et réservé une place pour le résultat dans la pile, suivant les conventions adoptées en cours.
- Vous utiliserez le registre `R7` pour `r`.
- Vous utiliserez l'instruction assembleur `MUL Rd, Rn, Rm` pour les multiplications (`Rd` est le registre destination).

(5 points)

2 Automate, microprogrammation et processeur (7 points)

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

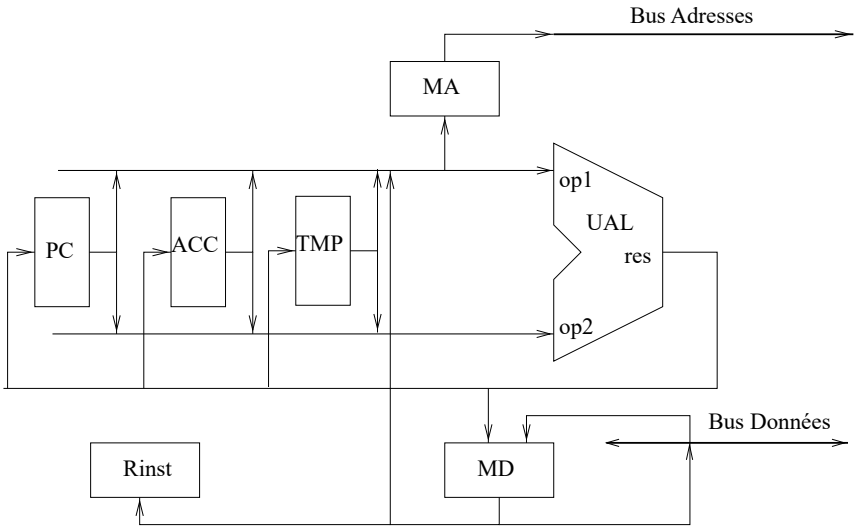

FIGURE 1 – Partie opérative du processeur

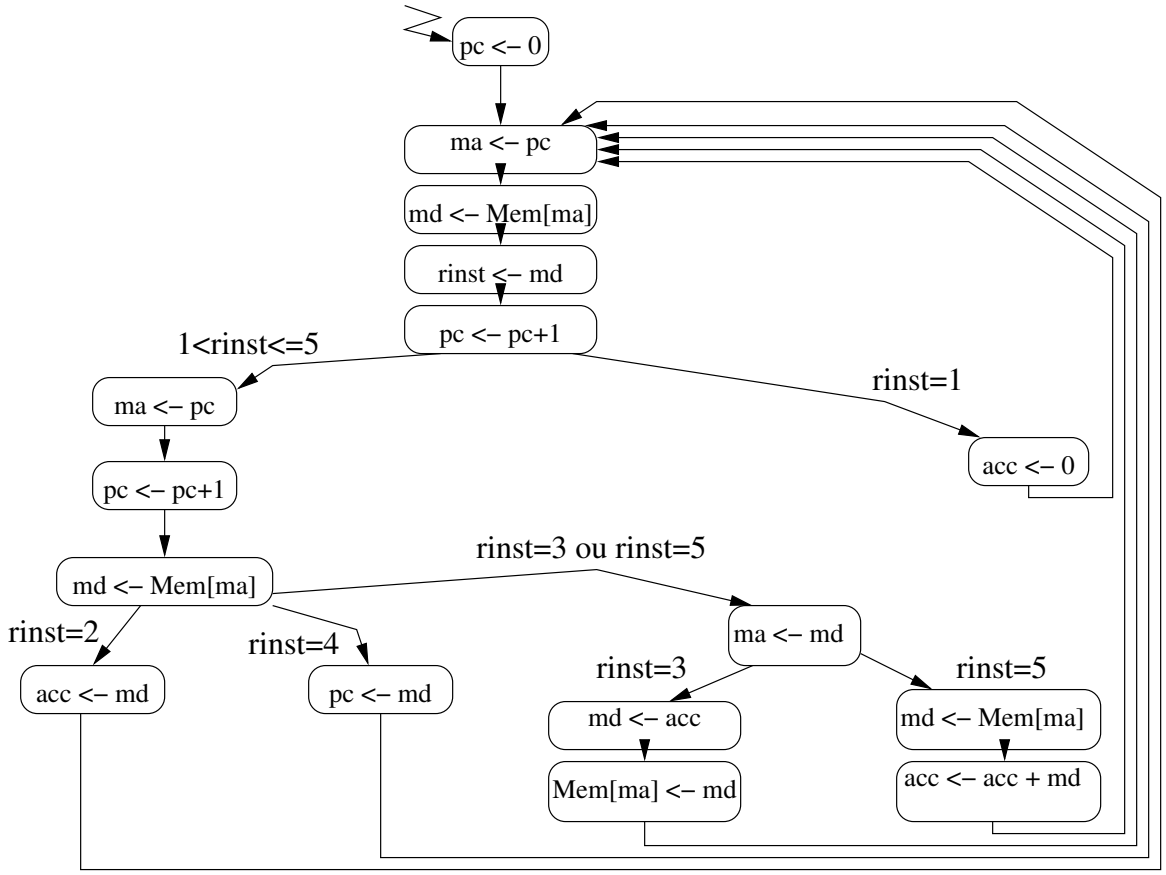

FIGURE 2 – Graphe de contrôle

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`
$\mathbf{reg}_0 \leftarrow \mathbf{0}$	mise à zéro	reg_0 est `PC`, `ACC`, ou `TMP`
$\mathbf{reg}_0 \leftarrow \mathbf{reg}_1$	affectation	reg_0 est `PC`, `ACC`, `TMP`, `MA`, ou `MD` reg_1 est `PC`, `ACC`, `TMP`, ou `MD`
$\mathbf{reg}_0 \leftarrow \mathbf{reg}_1$ **op** $\mathbf{reg}_2$	opération	reg_0 est `PC`, `ACC`, `TMP`, ou `MD` reg_1 est `PC`, `ACC`, `TMP`, ou `MD` reg_2 est `PC`, `ACC`, `TMP`, ou `MD` op : + ou −

Seul le registre `Rinst` permet de faire des tests : `Rinst` = entier (c'est donc la seule micro-condition).

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 à l'adresse `ad`	2
`add ad`	5	mise à jour de `ACC` avec la somme de `ACC` et du mot d'adresse `ad`	2

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*).

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 un mode d'adressage direct pour le chargement `ld`, un mode conditionnel pour le saut `jmp` et une nouvelle instructions d'échange de valeurs entre le registre `ACC` et la mémoire (instruction similaire à l'instruction `swp` du processeur ARM).

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
`ld@ ad`	6	chargement du mot à l'adresse mémoire `ad` dans `ACC`	2
`jacc ad`	7	saut à l'adresse `ad` si `ACC` est non nul	2
`swp ad`	8	échange symétrique des valeurs de `ACC` et du mot à l'adresse `ad`	2

Etat initial. On suppose que le programme suivant est stocké en mémoire, zone `.text` commence à l'adresse **0** et la zone `.data` commence à l'adresse **13**.

```
  .text
  .org 0
main :  clr
        add 14
        st 15
        clr
        add 13
        st 14
fin:   jmp fin
  .data
  .org 13
X:    .valeur 3
Y:    .valeur 9
Z:    .valeur 15
```

Questions.

(a) Donnez l'image en mémoire (valeur des 16 adresses et mots mémoire) du programme précédent **(1 point)**.
(b) Simuler l'exécution du début du programme précédent : compléter un tableau similaire à celui défini ci-après (une ligne par micro-action) avec les 19 premières micro-actions exécutées selon l'automate donné en figure 2 **(2 points)**.
(c) Combien de micro-actions sont nécessaires pour exécuter un saut `jmp fin` à la fin du programme précédent **(1 point)** ?
(d) Donnez les modifications à apporter à l'automate donné par le graphe de contrôle de la figure 2 afin d'interpréter les instructions supplémentaires suivantes :
   — `ld@ ad` **(1 point)**
   — `jacc ad` **(1 point)**
   — `swp ad` **(1 point)**

Indication pour `swp ad` *: utiliser le registre* `TMP`

Tableau de simulation

	Micro-Action (exécutée)	PC	Rinst	ACC	MA	MD	Mem[13]	Mem[14]	Mem[15]
0		?	?	?	?	?	3	9	15
1	pc ← 0	0							
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									

3 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 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`.
- `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`.

4 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`}.

5 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	