UNIVERSITÉ GRENOBLE ALPES INF401

Licence de Sciences et Technologies - Parcours INM, MIN, MIN-Int Année 2024–2025

Examen UE INF401 :

Introduction aux Architectures Logicielles et Matérielles

Durée = 2h00 **19 mai 2025**

Documents autorisés : 1 A4 R/V personnel manuscrit autorisé ; calculettes et téléphones portables interdits.

Remarques: La plupart des questions sont indépendantes, si vous avez du mal avec l'une, passez à la suivante — faites (bien) attention à votre gestion du temps. Tant que possible, indiquez bien tous les détails et justifiez vos réponses. Le barème est donné à titre indicatif.

Exercice 1 : Questions de cours

4 points

Répondre, succinctement, aux questions suivantes, avec des schémas explicatifs commentés si nécessaire.

Question 1.a

/1

Lorsqu'il exécute une instruction LDR ou STR, le processeur peut-il accéder ailleurs que dans la RAM ? Expliquez comment et pourquoi.

Oui, si des contrôleurs d'entrées-sorties sont branchés sur les bus d'adresses/de données/de contrôle, leurs registres sont projetés en mémoire et donc accessible du processeur. Cela sert à accéder à des périphériques.

Également vrai si il existe de la ROM accessible.

Question 1.b

/1

Listez 3 optimisations que le compilateur peut effectuer pour que les programmes soient plus rapides.

Déroulage de boucle, élimination de la récursivité terminale, propagation de constante, inlining de fonction, calcul constants à la compilation, remplacement d'expression arithmétique et suppression de code non atteignable.

Question 1.c

/1

Qu'est ce qui peut amener un processeur avec un pipeline à 5 étages (tel que vu dans le cours) à ne pas pouvoir exécuter les prochaines instructions dans le pipeline ?

La présence de branchement conditionnels dans les instructions suivantes et/ou des dépendances de données sur les instructions suivantes

Question 1.d

/1

Expliquez sur un exemple le principe de localité spatiale et/ou de localité temporelle, qui permet au cache d'être efficace.

```
int tab[5] = {1, 2, 3, 4, 5}
for (int i = 0; i < 5; i++)
    tab[i]++;
```

Le tableau `Tab` est chargé entièrement en cache au premier accès (à `tab[0]`), ce qui permet aux accès suivants d'être très rapides.

Exercice 2 : Programmation en langage d'assemblage `ARM`

11 points

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

Le but de cet exercice est de calculer le maximum des éléments d'un tableau d'entiers naturels, puis d'afficher cette valeur en binaire. Pour cela, on considère la zone `.data` suivante :

```
.data
msg:  .asciz "Max en binaire :"
             @1234567891111111  <- repères pour compter les caractères
             @.........0123456     de la chaîne ci-dessus
      .balign 4
tab:  .word 12
      .word 11
      .word 14
      .word 44
      .word 99
      .word 452
      .word 1
      .word 4
      .word 42
      .word 241
```

Représentation des données

Question 2.a

/0.5

Rappelez comment est représentée une chaîne de caractères en mémoire et donnez la taille en octets de la chaîne de caractères `msg`.

Un tableau d'octets, un octet par caractère + le caractère de fin de chaîne. Donc, `msg` est stockée sur 17 octets.

Question 2.b

/0.5

Quel est le rôle de la directive `.balign 4` ? Pourquoi s'en sert-on ici ?

Cette directive assure que la prochaine valeur sera stockée à la première adresse libre multiple de 4. Les contraintes d'alignement sur le stockage des *words* imposent que le mot « 12 » (au début de `tab`) soit stocké à une adresse multiple de 4.

Question 2.c */0.5*

Quelle est la taille en octets du tableau `tab` ?

$10 \times 4 = 40$ octets.

Question 2.d */0.5*

En supposant que la zone `.data` est stockée à partir de l'adresse `0xC000`, quelle sera l'adresse (en hexadécimale) du début du tableau `tab` ?

`msg` occupe les positions `0xC000` à `0xC011`.

La première adresse multiple de 4 libre est `0xC014`.

Écrire une procédure

Listing 1 – Code algorithmique pour `affBinaire`

```
1 Procedure affBinaire(a: entier naturel)
2       si (a != 0) alors
3             affBinaire(a/2)
4             si (a est pair) alors
5                   EcrCar('0')
6             sinon
7                   EcrCar('1')
8             fin si
9       fin si
```

Les 3 prochaines questions visent à traduire le corps de la procédure `affBinaire` (Listing 1), en appliquant **la méthode systématique vue en cours** et en tenant compte des indications suivantes :

- `a` est un paramètre passé par la pile.
- `EcrCar` est une procédure du fichier `es.s` utilisé en TP (un rappel des fonctions de `es.s` est donné en annexe 0).
- 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.

Question 2.e */0.5*

A quel endroit est disponible le paramètre `a` de la procédure `affBinaire`, par rapport au sommet de la pile `sp`, **avant le prologue de la procédure** ?

Avant prologue de la procédure, le paramètre `a` se situe au sommet de la pile (`sp + 0`)

Après prologue (sauvegarde de `lr`, `fp` et `r0, r1, r2`), il se situe à :

- `sp + 20`
- `fp + 8`
- voir le code ci-dessous

Question 2.f */2*

Donnez les code du prologue et de l'épilogue de la procédure `affBinaire`, selon la convention vue en cours.

Qu'est ce que la fonction `affBinaire` doit nécessairement sauver dans la pile ? Quel espace est donc nécessaire ?

Voir la correction ci-dessous.

Il faut sauver `lr`, `fp` et les registres `r0-r2` comme indiqué. Il faut donc $4 \times (1 + 1 + 3) = 4 \times 5 = 20$ octets dans la pile.

Question 2.g */2.5*

Traduire en langage `ARM` le corps de la procédure (lignes 2 à 9), sans répéter le prologue et l'épilogue définis précédemment.

N'oubliez pas de commenter votre code.

```
affBinaire:
    @ prologue
    push {lr}
    push {fp}
    mov fp, sp          @ mise à jour de fp
    push {r0, r1, r2}   @ sauvegarder les registres temporaires

    ldr r0, [fp, #8]    @ r0 <- a
    cmp r0, #0          @ (a == 0) ?
    beq finF

    mov r2, r0, lsr #1  @ r2 <- a/2
    push {r2}           @ empiler r2 comme paramètre
    bl  affBinaire      @ appel de affBinaire
    add sp,sp,#4        @ libération de l'espace des param.

    tst r0, #1          @ vérification du LSB de r0
    bne impair          @ si LSB == 1
    mov r1,#'0'         @ sinon afficher 0
    bl EcrCar
    b finF

impair:
    mov r1,#'1'         @ afficher 1
    bl EcrCar

finF:
    @epilogue
    pop {r0, r1, r2}
    pop {fp}
    pop {lr}
    bx lr
```

Question 2.h */1*

En supposant un appel d'`affBinaire(42)` à partir du programme principal (`main`), dessinez l'état de la pile **juste avant le branchement (`bl`)** correspondant au premier appel récursif effectif en ligne 3 de cette procédure.

```
Adresses    Valeurs
           |----------|
00..0      | ...      |
           | ...      |
           | ...      |
           | a=21     | <- SP
           | r0(main) |
           | r1(main) |
           | r2(main) |
 affBin    | fp(main) | <- FP
           | lr(main) |
           ------------
           | a=42     |
 main      | ...      |
           | ...      |
           | ...      |
FF..F      |----------|
```

Manipulation de tableau et appel de procédure

Listing 2 – Code algorithmique pour la procédure principale

```
1 max := T[0]
2 pour i ∈ ⟦1 ; 9⟧ faire
3     si max < T[i] alors
4         max := T[i]
5     fin si
6 fin pour
7 EcrChaine("Max en binaire :")
8 affBinaire(max)
```

Nous allons maintenant compléter la zone `.text` suivante :

```
      .text
      .global main
main:
      @ partie à compléter
      b exit
LD_tab:  .word tab
LD_msg:  .word msg
```

Question 2.i */2*

Traduisez en langage ARM les lignes 2 à 6 de l'algorithme en Listing 2, en tenant compte des indications suivantes :

— Pour les variables `max` et `i`, vous utiliserez les registres `r2` et `r3`.

— `tab` est défini dans la zone `.data` proposée au début de l'exercice.

```
      ldr r0, LD_tab
      ldr r2,[r0]           @ max <- tab[0]

      mov r3, #1            @ i <- 1
pour: cmp r3, #9            @ si i > 9
      bhi fin
      ldr r4, [r0, r3, LSL #2]  @ r4 <- tab[i]

      cmp r2, r4            @ (max >= tab[i]) ?
      bcs finsi
      mov r2, r4            @ max <- tab[i]

finsi:
      add r3, r3, #1        @ i++
      b pour
fin:
```

Question 2.j */1*

Traduisez en `ARM` les deux appels de fonctions lignes 7 et 8 en tenant compte des indications suivantes :

— On suppose que `max` est stocké dans le registre `r2`.
— `EcrChaine` est une procédure du fichier `es.s` utilisé en TP (un rappel des fonctions de `es.s` est donné en annexe).
— Le paramètre de la procédure `affBinaire` doit être stocké en pile, d'après la convention du cours.

```
@ EcrChaine("Max en binaire :")
      ldr r1, LD_msg
      bl EcrChaine

@ affBinaire(max)
      sub sp, sp, #4 @equivalent à push {r2}
      str r2, [sp]
      bl affBinaire
      add sp,sp,#4
```

Exercice 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 1.

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 :

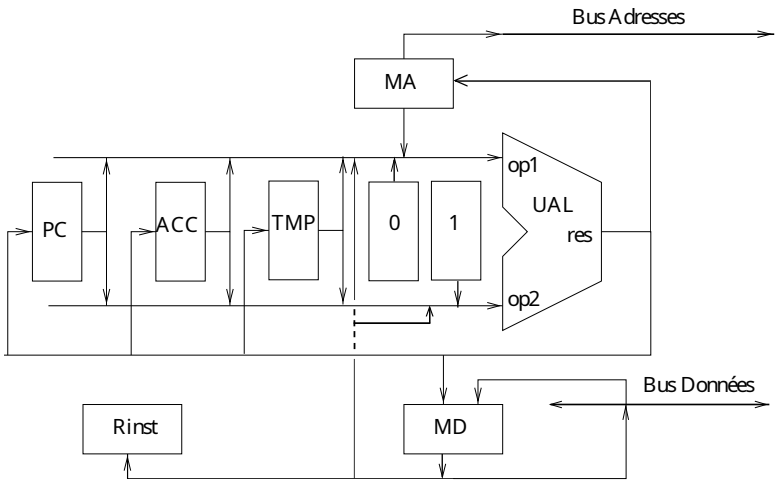

FIGURE 1 – Partie opérative du processeur

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`, `TMP`, `MA` ou `MD`
reg$_0$ ← reg$_1$	affectation	reg$_0$ est `PC`, `ACC`, `TMP`, `MA`, ou `MD` reg$_1$ est `PC`, `ACC`, `TMP`, `MA` ou `MD`
reg$_0$ ← reg$_1$ <<	décalage à gauche d'un bit	reg$_0$ est `PC`, `ACC`, `TMP`, `MA` ou `MD` reg$_1$ est `PC`, `ACC`, `TMP`, `MA` 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`, `MA` ou `MD` reg$_2$ est 1, `PC`, `ACC`, `TMP`, `MA` ou `MD` op : + ou −
reg$_0$ ← (reg$_1$ <<) op reg$_2$	opération avec décalage	reg$_0$ est `PC`, `ACC`, `TMP`, `MA` ou `MD` reg$_1$ est `PC`, `ACC`, `TMP`, `MA` ou `MD` reg$_2$ est 1, `PC`, `ACC`, `TMP`, `MA` 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.

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** (ou d'**une donnée**) est **de 4 bits** — la directive `.word` correspond donc, dans la suite de l'exercice, à la déclaration d'un mot sur **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.

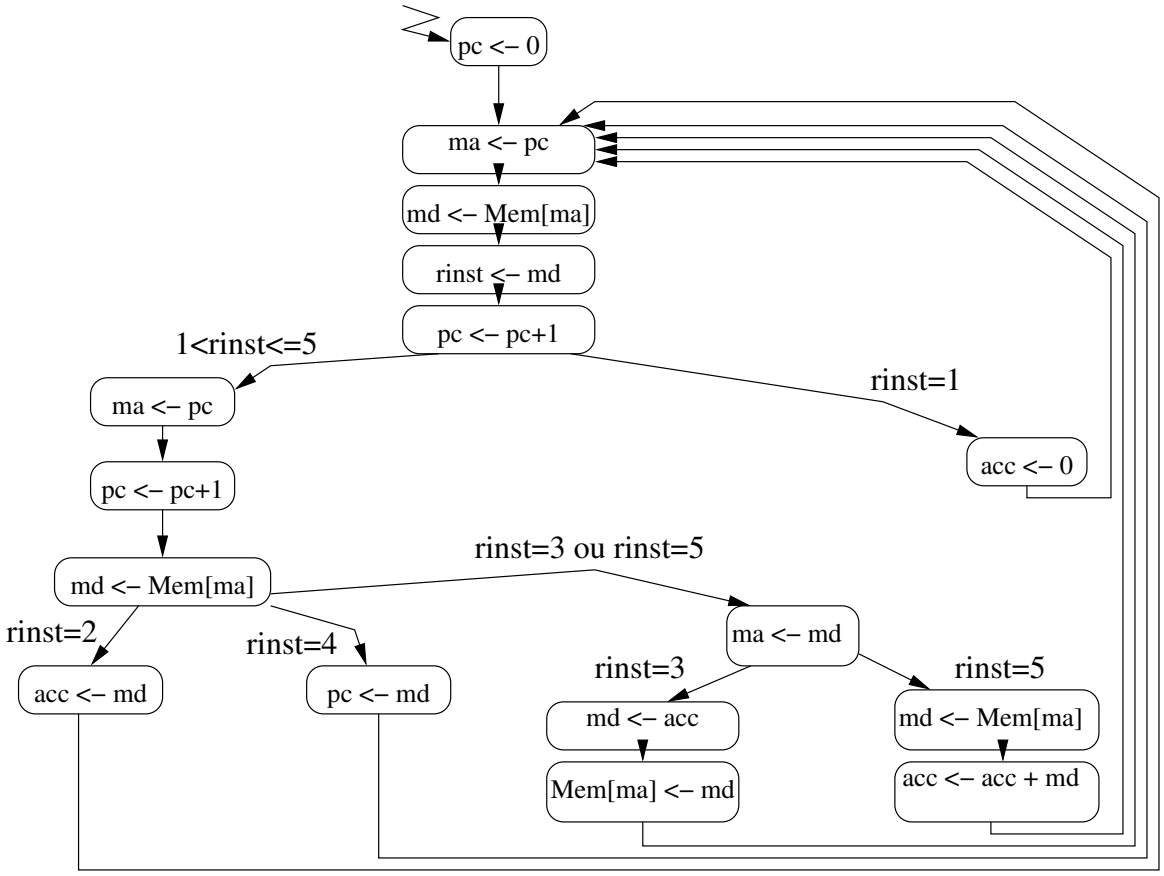

FIGURE 2 – Graphe de contrôle du processeur

Enrichissement du langage. Nous souhaitons enrichir le langage de notre processeur en ajoutant une instruction qui permet d'incrémenter deux valeurs en mémoire d'un seul coup.

Sémantique opérationnelle de l'instruction à ajouter. L'instruction à ajouter, son code, sa sémantique et la taille de son codage sont données dans la table ci-dessous :

instruction	code	signification	mots
`addv ad`	6	les deux mots à l'addresse **ad** et **ad + 1** sont incrémentés (en mémoire) de la valeur **ACC**	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 **12**.

```
    .text           @ zone text à l'adresse 0x0
main:   ld 1
        add A
        add B
        add C
        st Total
end:    jmp end

    .data           @ zone data à l'adresse 0xC
A:      .word 2
B:      .word 4
C:      .word 6
Total:  .word 0
```

Question 3.a */1*

Donnez l'état initial de la mémoire en binaire.

```
    @ section .text
0000      ld:   0010
0001       1:   0001   @ vi <- 1
0010     add:   0101
0011       A:   1100   @ adresse de A : 12
0100     add:   0101
0101       B:   1101   @ adresse de B : 13
0110     add:   0101
0111       C:   1110   @ adresse de C : 14
1000      st:   0011
1001   Total:   1111   @ adresse de Total : 15
1010     jmp:   0100
1011     end:   1010   @ adresse de jmp : 10
   @ section .data
1100       A:   0010   @ A <- 2
1101       B:   0100   @ B <- 4
1110       C:   0110   @ C <- 6
1111   Total:   0000   @ Total <- 0
```

Question 3.b */1*

Simulez l'exécution du début de ce programme (au niveau assembleur). Donnez les valeurs de l'accumulateur jusqu'à ce que le programme atteigne **end**.

```
  ACC <- 1       @ ld 1
  ACC <- 3       @ add A := 2
  ACC <- 7       @ add B := 4
  ACC <- 13      @ add C := 6
```

Question 3.c */1.5*

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 (deux premières instructions, jusqu'à la fin de l'exécution de l'instruction `add A`). 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.)

	Micro-Action (exécutée)	PC	Rinst	ACC	MA	MD	Commentaires
0		?	?	?	?	?	
1	pc ← 0	0					Début du fetch
2	ma ← pc				0		
3	md ← Mem[ma]					2	
4	rinst ← md		2				`Rinst <- 2 (ld)`
5	pc ← pc + 1	1					Fin du fetch
6	ma ← pc				1		
7	pc ← pc + 1	2					
8	md ← Mem[ma]					1	
9	acc ← md			1			`ACC <- 1`
10	ma ← pc				2		Début du fetch
11	md ← Mem[ma]					5	
12	rinst ← md		5				`Rinst <- 5 (add)`
13	pc ← pc+1	3					Fin du fetch
14	ma ← pc				3		Début récup. opérande
15	pc ← pc+1	4					
16	md ← Mem[ma]					12	
17	ma ← md				12		
18	md ← Mem[ma]					2	`md <- Mem[A]`
19	acc ← acc + md			3			`ACC += 2`

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

Question 3.d */1.5*

Donnez les modifications à apporter à l'automate fourni par le graphe de contrôle de la Figure 2 afin d'interpréter l'instruction supplémentaire `addv`.

Attention, les modifications nécessitent d'ajouter plusieurs états à l'automate d'origine. De plus, vous pouvez utiliser le registre `TMP` présent dans la partie opérative (Figure 1) pour stocker des valeurs intermédiaires utiles pour vos calculs.

Au lieu de partir vers Rinst = 3 (ou 5), on part, si Rinst = 6, vers :

```
1.  ma <- md
2.  md <- Mem[ma]
```

```
3.  md <- md + ACC
4.  Mem[ma] <- md
5.  ma <- ma + 1
6.  md <- Mem[ma]
7.  md <- md + ACC
8.  Mem[ma] <- md
```

Question 3.e */1 (Bonus)*

Remarque : Cette question ne sera corrigée que si vous avez traité (correctement ou non) TOUTES les autres questions de l'exercice 3.

Écrivez un nouveau programme qui **ajoute 4** aux valeurs stockées aux adresses `A`, `B`, `C`, grâce (entre autres) à l'instruction additionnelle `addv`. On suppose que la zone .data est placée consécutivement à la section `.text`. Donnez la traduction en binaire de votre programme.

0.5 point pour le programme :

```
    .text           @ zone .text à l'adresse 0x0
main: ld 4
      addv A        @ incrémente A et B
      add  C        @ ACC <- ACC (4) + *C
      st C          @ met à jour C
end:  jmp end       @ optionnel

    .data           @ zone .data à l'adresse 0x8
A:    .word 2
B:    .word 4
C:    .word 6
```

0.5 points pour la traduction binaire :

```
    @ section .text
0000:   0010    @ ld
0001:   0100    @ 4
0010:   0110    @ addv
0011:   1000    @ @A
0100:   0101    @ add
0101:   1010    @ @C
0110:   0011    @ st
0111:   1010    @ @C
0110:   0100    @ jmp       @ optionnel
0111:   0110    @ @end      @ (mais attention aux adresses !)

    @ section .data
1000:   0010    @ A := 2
1001:   0100    @ B := 4
1010:   0110    @ C := 6
```

Annexes

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 EcrCar` affiche le caractère ASCII dont le code est contenu dans `r1`.
- `bl EcrChaine` affiche la chaîne de caractères dont l'adresse est dans `r1`.

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`}.

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	