# TD 6

# Programmation en langage d'assemblage

Tous les exercices de ce TD sont à faire en utilisant le langage d'assemblage du LC3.

## Description du LC-3

**La mémoire et les registres :** La mémoire du LC-3 est organisée par mots de 16 bits, avec un adressage également de 16 bits (adresses de $(0000)_H$ à $(FFFF)_H$).

Le LC-3 comporte 8 registres généraux 16 bits : R0, ..., R7. R6 est réservé pour la gestion de la pile d'exécution, et R7 pour stocker l'adresse de retour des routines. Il comporte aussi des registres spécifiques 16 bits : PC (*Program Counter*), IR (*Instruction Register*), PSR (*Program Status Register*) qui regroupe plusieurs drapeaux.

Le PSR contient trois bits N, Z, P, indiquant si la dernière valeur (regardée comme le code d'un entier naturel en complément à 2 sur 16 bits) placée dans l'un des registres, R0, ..., R7 est négative strictement pour N, nulle pour Z, ou positive strictement pour P.

**Les instructions :**

| syntaxe | action | NZP |
|---|---|---|
| NOT DR,SR | DR <- not SR | * |
| ADD DR,SR1,SR2 | DR <- SR1 + SR2 | * |
| ADD DR,SR1,Imm5 | DR <- SR1 + SEXT(Imm5) | * |
| AND DR,SR1,SR2 | DR <- SR1 and SR2 | * |
| AND DR,SR1,Imm5 | DR <- SR1 and SEXT(Imm5) | * |
| LEA DR,label | DR <- PC + SEXT(PCoffset9) | * |
| LD DR,label | DR <- mem[PC + SEXT(PCoffset9)] | * |
| ST SR,label | mem[PC + SEXT(PCoffset9)] <- SR | |
| LDR DR,BaseR,Offset6 | DR <- mem[BaseR + SEXT(Offset6)] | * |
| STR SR,BaseR,Offset6 | mem[BaseR + SEXT(Offset6)] <- SR | |
| BR[n][z][p] label | Si (cond) PC <- PC + SEXT(PCoffset9) | |
| NOP | No Operation | |
| RET | PC <- R7 | |
| JSR label | R7 <- PC; PC <- PC + SEXT(PCoffset11) | |

**Directives d'assemblage :**

| | |
|---|---|
| `.ORIG adresse` | Spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit. |
| `.END` | Termine un bloc d'instructions. |
| `.FILL valeur` | Réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre. |
| `.BLKW nombre` | Cette directive réserve le nombre de mots de 16 bits passé en paramètre. |
| `;` | Les commentaires commencent par un point-virgule. |

**Les interruptions prédéfinies :** `TRAP` permet de mettre en place des *appels système*, chacun identifié par une constante sur 8 bits, gérés par le système d'exploitation du LC-3. On peut les appeler à l'aide des macros indiquées ci-dessous.

| instruction | macro | description |
|---|---|---|
| `TRAP x00` | `HALT` | termine un programme (rend la main à l'OS) |
| `TRAP x20` | `GETC` | lit au clavier un caractère ASCII et le place dans R0 |
| `TRAP x21` | `OUT` | écrit à l'écran le caractère ASCII placé dans R0 |
| `TRAP x22` | `PUTS` | écrit à l'écran la chaîne de caractères pointée par R0 |
| `TRAP x23` | `IN` | lit au clavier un caractère ASCII, l'écrit à l'écran, et le place dans R0 |

**Constantes :** Les constantes entières écrites en hexadécimal sont précédées d'un `x` (en décimal elles peuvent être précédées d'un `#` optionnel) ; elles peuvent apparaître comme paramètre : des instructions du LC3 (opérandes immédiats, attention à la taille des paramètres), des directives `.ORIG`, `.FILL` et `.BLKW`.

# Codage des instructions LC3

On donne ici un tableau récapitulatif du codage des instructions LC3.

| **syntaxe** | **action** | **NZP** | **codage** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | opcode | | | | arguments | | | | | | | | | | | |
| | | | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| NOT DR,SR | DR ← not SR | * | 1 0 0 1 | | | | DR | | | SR | | | 1 1 1 1 1 1 | | | | | |
| ADD DR,SR1,SR2 | DR ← SR1 + SR2 | * | 0 0 0 1 | | | | DR | | | SR1 | | | 0 | 0 0 | | SR2 | | |
| ADD DR,SR1,Imm5 | DR ← SR1 + SEXT(Imm5) | * | 0 0 0 1 | | | | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND DR,SR1,SR2 | DR ← SR1 and SR2 | * | 0 1 0 1 | | | | DR | | | SR1 | | | 0 | 0 0 | | SR2 | | |
| AND DR,SR1,Imm5 | DR ← SR1 and SEXT(Imm5) | * | 0 1 0 1 | | | | DR | | | SR1 | | | 1 | Imm5 | | | | |
| LEA DR,label | DR ← PC + SEXT(PCoffset9) | * | 1 1 1 0 | | | | DR | | | PCoffset9 | | | | | | | | |
| LD DR,label | DR ← mem[PC + SEXT(PCoffset9)] | * | 0 0 1 0 | | | | DR | | | PCoffset9 | | | | | | | | |
| ST SR,label | mem[PC + SEXT(PCoffset9)] ← SR | | 0 0 1 1 | | | | SR | | | PCoffset9 | | | | | | | | |
| LDR DR,BaseR,Offset6 | DR ← mem[BaseR + SEXT(Offset6)] | * | 0 1 1 0 | | | | DR | | | BaseR | | | Offset6 | | | | | |
| STR SR,BaseR,Offset6 | mem[BaseR + SEXT(Offset6)] ← SR | | 0 1 1 1 | | | | SR | | | BaseR | | | Offset6 | | | | | |
| BR[n][z][p] label | Si (cond) PC ← PC + SEXT(PCoffset9) | | 0 0 0 0 | | | | n | | z | | p | | PCoffset9 | | | | | |
| NOP | No Operation | | 0 0 0 0 | | | | 0 | | 0 | | 0 | | 0 0 0 0 0 0 0 0 0 | | | | | |
| RET | PC ← R7 | | 1 1 0 0 | | | | 0 0 0 | | | 1 1 1 | | | 0 0 0 0 0 0 | | | | | |
| JSR label | R7 ← PC; PC ← PC + SEXT(PCoffset11) | | 0 1 0 0 | | | | 1 | PCoffset11 | | | | | | | | | | |

# Traduction de programmes en langage d'assemblage

Il vous est demandé de toujours commencer par écrire un pseudo-code pour le programme ou la routine demandé, en faisant apparaître les registres que vous allez utiliser pour effectuer vos calculs, et en ajoutant tous les commentaires utiles. Vous traduirez ensuite votre pseudo-code vers le langage d'assemblage du LC3 en utilisant les règles de traduction suivantes.

**Traduction d'un « bloc if » :** On suppose que la condition d'entrée dans le bloc consiste simplement en la comparaison du résultat d'une expression arithmétique `e` à 0. Dans ce qui suit, `cmp` désigne une relation de comparaison : $<$, $\leq$, $=$ , $\neq$, $\geq$ , $>$. On note `!cmp` la relation contraire de la relation `cmp`, traduite dans la syntaxe des bits `nzp` de l'instruction `BR`. Si par exemple `cmp` est $<$, alors `BR!cmp` désigne `BRpz` (pour « positive or zero »).

```
/* En pseudo-code */
if e cmp 0 {
  corps du bloc
}
```

```
; En langage d'assemblage du LC3
     evaluation de e
     BR!cmp endif    ; branchement sur la sortie du bloc
     corps du bloc
endif:
```

**Traduction d'un bloc « if-else » :**

```
/* En peudo-code */
if e cmp 0 {
  corps du bloc 1
}
else {
  corps du bloc 2
}
```

```
; En langage d'assemblage du LC3
     evaluation de e
     BR!cmp else     ; branchement sur le bloc else
     corps du bloc 1
     BR endif        ; branchement sur la sortie du bloc
else:
     corps du bloc 2
endif:
```

**Traduction d'une « boucle while » :**

```
/* En peudo-code */
while e cmp 0 {
  corps de boucle
}
```

```
; En langage d'assemblage du LC3
loop:
     evaluation de e
     BR!cmp endloop  ; branchement sur la sortie de boucle
     corps de boucle
     BR loop         ; branchement inconditionnel
endloop:
```

**Quelques « astuces » à connaître :**

— Initialisation d'un registre à 0 : `AND Ri,Ri,#0`

— Initialisation d'un registre à une constante `n` (représentable en complément à 2 sur 5 bits) :

```
AND Ri,Ri,#0
ADD Ri,Ri,n
```

— Calcul de l'opposé d'un entier (on calcule le complément à 2 de Rj dans Ri) :

```
NOT Ri,Rj
ADD Ri,Ri,#1
```

— Multiplication par 2 de Rj, résultat dans Ri : `ADD Ri,Rj,Rj`

— Copie du contenu de Rj dans Ri : `ADD Ri,Rj,#0`

# Exercice 1 : Autour de la sommation

**1**) On considère le programme incomplet ci-dessous. On souhaite pouvoir ajouter les entiers aux adresses `add0`, `add1`, `add2`, et placer le résultat à l'adresse `resultat`. Donnez un code assembleur pour cela.

```
            .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
                            ; R1 sera utilisé comme accumulateur
            ### A COMPLETER ###

; partie dédiée au résultat et aux données
resultat:  .BLKW #1         ; espace pour stocker le résultat (résultat attendu ici : 73 soit x49)
add0:      .FILL #12
add1:      .FILL #45
add2:      .FILL #16
            .END
```

**2**) Supposons que R0 contient un entier positif $x$, et R1 un entier positif $y$. On souhaite affecter à R2 l'entier $x - y$ : proposez un morceau de code pour cela.

**3**) On considère le programme incomplet ci-dessous. On souhaite pouvoir ajouter les entiers compris entre les adresses `debut` et `fin` à l'aide d'une boucle, et placer le résultat de la sommation à l'adresse `resultat` Il vous est demandé de commencer par donner un pseudo-code pour votre programme, en respectant les directives d'utilisation des registres données ci-dessous, avant de le traduire en langage d'assemblage.

```
            .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
; R0 sera utilisé comme un pointeur pour une case du tableau
; R1 contiendra l'opposé de l'adresse fin, pour comparaison avec R0
; R2 servira à l'accumulation des entiers du tableau
; R3 servira de registre temporaire
            ### A COMPLETER ###
; partie dédiée au résultat et aux données
resultat:  .BLKW #1         ; espace pour stocker le résultat (résultat attendu ici : 157 soit x9D)
debut:     .FILL #12
            .FILL #45
            .FILL #16
            .FILL #06
fin:       .FILL #78
            .END
```

# Exercice 2 : Multiplication par 6 des entiers d'un tableau

On considère le programme à compléter ci-dessous.

```
           .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
           LD R6,spinit    ; initialisation du pointeur de pile
           LEA R0,debut    ; charge l'adresse de début du tableau
           LEA R1,fin      ; charge l'adresse de fin du tableau
           JSR mul6tab     ; appel à une routine
           HALT            ; termine le programme

; partie dédiée aux données
mask:      .FILL x000F     ; constante x000F
debut:     .FILL 4         ; adresse de début de tableau
           .FILL 5
           .FILL 6
fin:       .FILL 3         ; adresse de fin de tableau

; pile
spinit:   .FILL stackend
          .BLKW #5
stackend: .BLKW #1      ; adresse du fond de la pile

; Routine mul5tab, pour multiplier les entiers d'un tableau par 6 modulo 16.
; Les multiplications se feront sur place.
; paramètres d'entrée   : R0, adresse de début du tableau
;                         R1, adresse de fin du tableau
mul6tab:
          : *** A COMPLETER ***
          .END
```

Le but est de compléter la routine `mul6tab` pour qu'elle multiplie les entiers d'un tableau par 6 modulo 16 : en entrée R0 contient l'adresse de la première case du tableau, R1 l'adresse de la dernière case. Vous traduirez pour cela le pseudo-algorithme suivant :

```
; R2 <- R0
; while( R2 <= R1 ) {  // (R2 <= R1) <=> (R2-R1 <= 0)
;   R3 <- mem[R2];
;   R3 <- 2*R3+4*R3;   // R3 <- 6*R3
;   R3 <- R3 & 0x000F; // R3 <- R3 modulo 16
;   mem[R2] <- R3;
;   R2++;
; }
}
```

**1**) En utilisant uniquement R4 comme registre intermédiaire, montrez comment traduire la ligne `R3 <- 2*R3+4*R3`.

**2**) A la ligne `R3 <- R3 & 0x000F`, le & désigne le AND bit-à-bit : justifier le fait que l'opération `R3 & 0x000F` calcule bien le reste dans la division euclidienne de R3 par 16. Comment traduirez-vous cette ligne en langage d'assemblage ?

**3**) Traduisez l'algorithme proposé. Vous pouvez utiliser R4 et/ou R5 pour les calculs intermédiaires.

## Exercice 3 : Programme Mystère LC3

On (source : `http://castle.eiu.edu/~mathcs/mat3670/index/index.html`, avec l'aimable autorisation des auteurs) fournit un programme LC3 sous forme d'un bout de mémoire dont on donne le contenu en hexadécimal. Il est demandé de décoder ce programme et de dire ce qu'il fait. On fera attention aux sauts de PC (le cas échéant, on rajoutera des *labels* à certaines adresses mémoires).

| Adresse | Contenu | Contenu binaire | Détails des instructions | pseudo-code |
|---|---|---|---|---|
| x3000 | x5020 | 0101 000 000 1 00000 | AND (mode cst), DR=SR=R0, Imm5=x00 | $R_0 \leftarrow R_0 \& 0 = 0$ |
| x3001 | x1221 | | | |
| x3002 | xE404 | | | |
| x3003 | x6681 | | | |
| x3004 | x1262 | | | |
| x3005 | x16FF | | | |
| x3006 | x03FD | | | |
| x3007 | xF025 | | HALT | HALT |
| x3008 | x0006 | donnée | - | |

## Exercice 4 : Saisie d'une chaîne de caractères

Le système d'exploitation du LC3 fournit une interruption (pour simplifier, disons que c'est une routine) permettant d'afficher une chaîne de caractères (`PUTS` ≡ `TRAP x22`), mais on n'a pas la possibilité de saisir une chaîne de caractères. Le but est ici d'écrire une routine permettant cela. On considère le programme à compléter ci-dessous.

```
 .ORIG x3000
; Programme principal
         LEA R6,stackend ; initialisation du pointeur de pile
         ; affiche la chaîne à l'adresse msg1
         LEA R0,msg1
         PUTS
         ; saisie d'une chaîne à l'adresse ch1
         LEA R1,ch1
         JSR saisie
         ; affiche la chaîne à l'adresse msg2
         LEA R0,msg2
         PUTS
         ; affiche la chaîne à l'adresse ch1
         LEA R0,ch1
         PUTS
         HALT

; partie dédiée aux données
msg1:    .STRINGZ "Entrez une chaîne : "
msg2:    .STRINGZ "Vous avez tapé : "
ch1:     .BLKW #8

; pile
stack:    .BLKW #5
stackend: .FILL #0

; Sous-routine pour saisir une chaîne de caractères
; paramètre d'entrée : l'adresse R1 du début de chaîne
saisie:
          ; *** A COMPLETER ***
          .END
```

**1**) Dans une routine, avant d'appeler une autre routine, ou de déclencher une interruption vers le système d'exploitation, il est important de sauvegarder l'adresse de retour contenue dans R7 : pourquoi?

**2**) Complétez la routine `saisie` de manière à ce qu'elle permette de saisir une chaîne de caractères au clavier, en rangeant les caractères lus à partir de l'adresse contenue dans R1. La saisie se termine lorsqu'un retour chariot (code ASCII 13) est rencontré, et la chaîne de caractères doit être terminée par un caractère `'\0'` (de code ASCII 0). Vous utiliserez `GETC`, qui lit un caractère au clavier, et place son code dans `R0`.

## Exercice 5 : Décompte de bits non-nuls

Écrire une fonction dans l'assembleur du LC3 afin de compter le nombre de bits non-nuls dans un entier. Votre programme principal doit appeler votre fonction afin de compter le nombre de bits non-nuls de l'entier à l'adresse `n`, et placer le résultat à l'adresse `r`. Vous compléterez le code ci-dessous, en le commentant.

```
         .ORIG x3000
; Programme principal
         ...
; Données
n:       .FILL #78
r:       .BLKW #1

; Sous-routine pour calculer le nombre de bit non-nuls dans R1
; paramètre : R1, inchangé
; retour : R0, le nombre de bits non nuls dans R1
; registres temporaires : R2, R3, R4
cmptbits: ...
.END
```

## Exercice 6 : Débogage

On considère le programme ci-dessous, qui s'assemble sans message d'erreur, est s'exécute sur le LC3 ; par contre, l'exécution n'aboutit jamais au `HALT` : le programme boucle indéfiniment…

```
        .ORIG x3000
; Programme principal
        LD R0, n        ; R0 <- mem[n]
        JSR fois20      ; appel à la routine fois20 : R1 <- 20 * R0
        ST R1, r        ; mem[r] <- R1
        HALT            ; rend la main à l'OS
n:      .FILL 7         ; Donnée, entier 7
r:      .BLKW 1         ; Résultat

; Routine pour effectuer R1 <- 20 * R0
; Attention, R2 est écrasé par l'appel.
fois20: JSR fois10     ; R1 <- 10 * R0
        ADD R1, R1, R1 ; R1 <-  2 * R1 = 20 * R0
        RET            ; retour à l'appelant

; Routine pour effectuer R1 <- 10 * R0
; Attention, R2 est écrasé par l'appel.
fois10: ADD R1, R0, R0  ; ......................................
        ADD R2, R1, R1  ; ......................................
        ADD R2, R2, R2  ; ......................................
        ADD R1, R1, R2  ; ......................................
        RET             ; retour à l'appelant
        .END
```

**1**) Identifiez la raison pour laquelle le programme entre dans une boucle infinie : expliquez clairement les raisons du problème.

**2**) Quelle solution classique permet de remédier à ce problème ?

**3**) `fois10` effectue `R1 <- 10 * R0` : ajouter des commentaires dans le programme pour justifier cela.

## Exercice 7 : Nombre d'occurrences

Il s'agit de compléter la routine `nboccs` pour qu'elle compte le nombre d'occurrences d'un caractère dans une chaîne de caractères (par exemple, le nombre d'occurrences de `'o'` dans `"Toto fait du vélo"` est 3). Rappel : le code du caractère de fin de chaine, noté généralement `'\0'`, est tout simplement l'entier 0. Il vous est demandé de commencer par donner un pseudo-code pour votre routine, en respectant les directives d'utilisation des registres données ci-dessous, avant de le traduire en langage d'assemblage.

```
 .ORIG x3000
; Programme principal
         LEA R0,ch
         LD R1,car
         JSR nboccs
         HALT

; Une chaîne de caractères
ch:      .STRINGZ "Toto fait du vélo"
car:     .FILL x006F    ; code ASCII du caractère 'o'

; Sous-routine pour pour compter le nombre d'occurrences
; dans une chaîne de caractères terminée par un '\0'.
; paramètres d'entrée   : R0 contient l'adresse du début de la chaîne
;                         R1 contient le code ASCII d'un caractère
; paramètre de sortie   : R2 le nombre d'occurrences du caractère
; registres temporaires : R3 contiendra l'opposé de R1
;                         R4 servira de pointeur pour parcourir la chaîne
;                         R5 recevra successivement chacun des caractères de la chaîne
nboccs:
### PARTIE A COMPLETER ###
         .END
```