8 Programmation

Après un bref rappel du jeu d'instructions du processeur LC-2, on étudie la programmation en langage machine et en assembleur, l'édition de liens pour produire un programme exécutable, et la pile d'exécution.

8.1 Jeu d'instructions

La figure 1 présente un tableau récapitulatif du format des instructions du LC-2.

Quelques remarques importantes.

- Les instructions `ADD`, `AND`, `NOT`, `LD`, `LDI`, `LDR` et `LEA` modifient les codes de condition N, Z et P.
- Les instructions `JMP` et `JSR` (resp. `JMPR` et `JSRR`) ont les mêmes *opcodes* (bits 12 à 15) et ne diffèrent que par le bit 11, appelé L pour *link* (i.e., sauvegarder l'adresse de retour)
- Les instructions `RTI` et `TRAP` seront étudiés dans un chapitre ultérieur : elles sont nécessaires pour la programmation d'appels système et d'interruptions. Nous utiliserons néanmoins l'instruction `TRAP x25`, en supposant que celle-ci appelle une routine du système d'exploitation qui arrête l'exécution du processeur.
- Les instructions `LDI` et `STI` ne respectent pas la philosophie RISC où les seuls accès mémoire sont des opérations *load*/*store* élémentaires ; elles ne sont donc pas indispensables : `LDI` peut être remplacée par `LD` et `LDR`, `STI` par `LD` et `STR`.

Dans toute la suite, on supposera que les programmes en langage machine exécutés par le LC-2 commencent à l'adresse *`x3000`*.

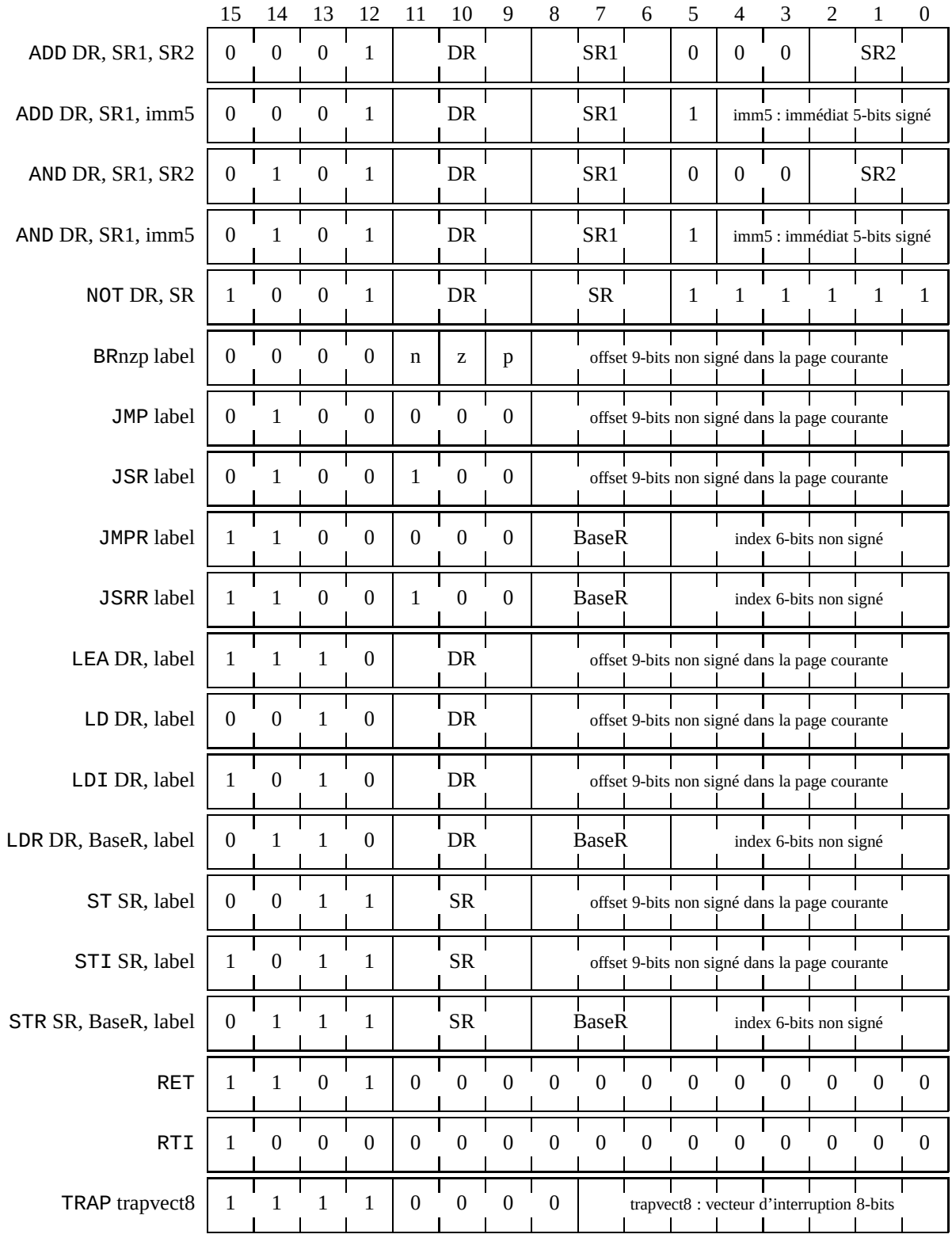

FIG. 1 – Format des instructions du LC-2

8.2 Programmation en assembleur

On considère la boucle de la figure 2 qui ajoute 5 aux 100 premiers éléments d'un tableau A.

```
int i, A[100];
for (i=0; i<100; i++) {
  A[i] = A[i] + 5;
}
```

FIG. 2 – Exemple de programme C à traduire en langage machine LC-2

Le langage *assembleur* — ou langage d'*assemblage* — se situe un niveau d'abstraction au dessus du langage machine. Il propose des instructions élémentaires directement calquées sur les instructions du processeur. L'*assembeur* désigne l'outil de traduction du langage d'assemblage vers le langage machine ; il automatise le codage en binaire des instructions élémentaires.

En particulier, la programmation en assembleur épargne le calcul des adresses de branchement et des offsets d'accès à la mémoire. Par exemple, lors de la traduction en binaire d'une instruction de branchement conditionnel pour le LC-2, l'assembleur vérifie que l'adresse de destination se situe bien dans la page courante du PC.

```
Ligne     Instruction                    Commentaire
1         AND       R0, R0, #0       ; i ← 0
2         ADD       R2, R0, #-12
3         ADD       R2, R2, #-13
4         ADD       R2, R2, R2
5         ADD       R2, R2, R2       ; R2 ← -100
6   Loop  LDR       R1, R0, #30      ; R1 ← Mem[30+i]
7         ADD       R1, R1, #5       ; R1 ← R1+5
8         STR       R1, R0, #30      ; Mem[30+i] ← R1
9         ADD       R0, R0, #1       ; i ← i+1
10        ADD       R3, R0, R2
11        BRn       Loop             ; si (i<100) aller à Loop
12        TRAP      x25              ; arrêter la machine
```

FIG. 3 – Première version en assembleur LC-2

La figure 3 montre un premier exemple de programme en assembleur LC-2. La syntaxe générale des instructions est donnée dans le tableau suivant.

N° de ligne	Label	Opcode	Opérandes	Commentaire

- Le numéro de ligne ne doit pas être confondu avec l'adresse en mémoire de l'instruction (sur 16 bits), déterminée ultérieurement lors de la traduction en langage machine.
- Les labels permettent de nommer des adresses de manière symbolique, ils ressemblent à des pointeurs (constants) en langage C. Ils désignent soit une zone mémoire pour des chargements ou rangements (`LDR R1, Cst`), soit une cible pour une instruction de branchement (`BRn Loop`).

- Les oppérations sont les noms associés aux opcodes des instructions du langage machine.
- Les opérandes peuvent être :
  - un registre R_i ($0 \leq i \leq 7$) ;
  - une valeur « immédiate » ou un « offset » pour chargement/rangement ou pour un branchement ;
  - un label (i.e., un « offset symbolique ») pour un chargement/rangement ou pour un branchement.

La traduction du programme précédent en langage machine est donnée sur les figures 4 et 5.

```
Adresse   Instruction                  Code machine
x3000     AND       R0, R0, #0       ; 0101 000 000 1 00000
x3001     ADD       R2, R0, #-12     ; 0001 010 000 1 10100
x3002     ADD       R2, R2, #-13     ; 0001 010 010 1 10011
x3003     ADD       R2, R2, R2       ; 0001 010 010 0 00 010
x3004     ADD       R2, R2, R2       ; 0001 010 010 0 00 010
x3005     LDR       R1, R0, #30      ; 0110 001 000 011110
x3006     ADD       R1, R1, #5       ; 0001 001 001 1 00101
x3007     STR       R1, R0, #30      ; 0111 001 000 011110
x3008     ADD       R0, R0, #1       ; 0001 000 000 1 00001
x3009     ADD       R3, R0, R2       ; 0001 011 000 0 00 010
x300A     BRn       x3005            ; 0000 100 000000101
x300B     TRAP      x25              ; 1111 0000 00100101
```

FIG. 4 – Traduction de la première version

Adresse	Binaire (16 bits)	Hexadécimal
x3000	0101000000100000	*x5020*
x3001	0001010000110100	*x14B4*
x3002	0001010010110011	*x1483*
x3003	0001010010000010	*x1482*
x3004	0001010010000010	*x1482*
x3005	0110001000011110	*x621E*
x3006	0001001001100101	*x1265*
x3007	0111001000011110	*x721E*
x3008	0001000000100001	*x1021*
x3009	0001011000000010	*x1602*
x300A	0000100000000101	*x0805*
x300B	1111000000100101	*xF025*

FIG. 5 – Implémentation en langage machine

En plus des instructions correspondant aux opérations implémentées sur le LC-2, le programme assembleur peut contenir des « pseudo-instructions » : il s'agit de raccourcis pour des appels système associés à des vecteurs d'interruption. Par exemple, l'appel système `TRAP x25` correspondant à l'arrêt de la machine peut plus simplement s'écrire `HALT` en assembleur.

Directive	**Signification**	**Exemple**
`.ORIG` *adresse*	Adresse en mémoire de la première instruction du programme	`.ORIG x3000` indique que `AND R0, R0, #0` est à l'adresse *x3000*
`.FILL` *valeur*	Réserver le mot mémoire suivant et y stocker le paramêtre *valeur*	`.FILL -100` réserve un mot juste après l'instruction `HALT` (l'adresse du label `Cst`) et y stocke la valeur -100 en complément à 2 sur 16 bits
`.STRINGZ` *chaîne*	Stocker la *chaîne* de caractères, avec un caractère ASCII par mot 16-bits, suivi du mot nul de fin de chaîne	`.STRINGZ "Salut"` initialise les 6 premiers mots du tableau `A` avec *x0053 x0061 x006c x0075 x0074 x0000*
`.BLKW` *nombre*	Les *nombre* mots mémoire suivants sont réservés comme mémoire de travail	`.BLKW 100` réserve 100 mots consécutifs pour compléter le tableau `A`
`.END`	Fin du code source du programme	

FIG. 6 – Directives de l'assembleur LC-2

Enfin, le programme assembleur contient toujours des directives de deux types : d'une part des données numériques (constantes ou valeurs d'initialisation de variables), et d'autre part des direc-

tives de placement du programme en langage machine dans la mémoire du LC-2. Ces directives sont décrites sur le tableau de la figure 6 et illustrées sur la figure 7 par une nouvelle implémentation en assembleur de la boucle de la figure 2.

```
        .ORIG     x3000            ; adresse de début du programme
        ; boucle qui ajoute 5 aux 100 éléments d'un tableau A
        AND       R0, R0, #0       ; i ← 0
        LD        R2, Cst          ; R2 ← -100
Loop    LDR       R1, R0, A        ; R1 ← A[i]
        ADD       R1, R1, #5       ; R1 ← R1+5
        STR       R1, R0, A        ; A[i] ← R1
        ADD       R0, R0, #1       ; i ← i+1
        ADD       R3, R0, R2
        BRn       Loop             ; si (i<100) aller à Loop
        HALT                       ; TRAP x25 pour arrêter la machine
        ; zone de stockage
Cst     .FILL     -100             ; constante -100
A       .STRINGZ  "Salut"          ; Les 6 premiers mots du tableau A
        .BLKW     100              ; réservation de 100 mots
        .END                       ; fi n du programme assembleur
```

FIG. 7 – Deuxième version en assembleur LC-2

8.3 Processus d'assemblage

Le processus d'*assemblage* (i.e., traduction en langage machine) comporte deux passes. En effet, il est nécessaire de calculer au préalable les adresses des labels et des instructions, en interprétant les directives. On construit ainsi une *table des symboles*, associant chaque label à une adresse en mémoire. Dans une deuxième passe, les instructions assembleur sont traduites en binaire, les offsets et les adresses correspondant aux labels étant calculés à partir de la table des symboles. La table des symboles du programme assembleur de la figure 7 est détaillée dans le tableau suivant.

Symbole	Adresse
Loop	*x3002*
Cst	*x3009*
A	*x300A*

Le code binaire issu de l'assemblage s'appelle le *code objet*. À titre d'exercice, on pourra effectuer l'assemblage du programme de la figure 7 et vérifier le résultat avec celui de l'assembleur `lc2asm` fourni en TD.

Ce schéma de traduction en deux passes est compliqué par deux particularités des programmes réels : une application complexe fait appel à des fonctions de bibliothèques ou du système d'exploitationm, et elle est souvent scindée en plusieurs fichiers, suivant un découpage logique inconnu

de l'assembleur. Il en résulte que certains symboles du programme assembleur ne correspondent pas à du code ou des données présents lors de l'assemblage. Ces symboles sont indiqués par une directive `.EXTERN` *symbole* pour indiquer que l'adresse de *symbole* sera déterminée ultiérieurement. Le code objet résultant de l'assemblage contient donc une table des symboles. Certains de ces symboles n'ont pas d'adresse connue, on dit que ces symboles ne sont *pas résolus* lors de l'assemblage.

Pour pouvoir exécuter le programme, plusieurs fichiers objet (et des bibliothèques) doivent être rassemblés lors de la *phase d'édition de liens*. Cette phase consiste à faire correspondre les différentes tables des symboles des codes objet, et à *résoudre* chaque symbole externe, i.e., à leur affecter leur adresse en mémoire.

8.4 Structures de données

La traduction des structures de données d'un langage impératif comme le C en assembleur LC-2 ne pose pas de problèmes particuliers.

Un entier sur 16 bits et stocké sur un mot, un nombre flottant 32 bits sur 2 mots consécutifs, et un tableau d'entiers sur n mots consécutifs. On a vu que les chaînes de caractères subissaient un traîtement particulier (proche du codage UNICODE en Java) : chaque caractère ASCII est codé sur un octet, mais un seul caractère ASCII est stocké dans chaque mot 16 bits (sur l'octet de poids faible), le caractère de fin de chaîne étant le mot *x0000*.

En assembleur, l'accès aux éléments d'un tableau peut être implémenté de deux manières duales, dépendantes de l'adresse de celui-ci et du type de parcours envisagé.

1. La technique la plus simple est utilisée dans le programme de la figure 7. Elle s'applique lorsque l'adresse de base du tableau (l'adresse du premier élément) se situe dans la page du PC courant. Il suffit alors d'une instruction de chargement/rangement avec mode d'adressage indexé par un registre contenant le numéro de l'élément du tableau accédé : par exemple, `LDR R1, R0, A`, `LDR R1, R0, x300A`, `SDR R1, R0, A`, ou `SDR R1, R0, x300A`.
2. Une technique beaucoup plus générale consiste à charger l'adresse de base du tableau dans un registre, et à ajouter un offset à ce registre de référence en fonction de l'élément accédé : par exemple,

```
LEA       R0, A            ; R0 ← adresse de base de A
LDR       R2, R0, #10      ; R2 ← A[10]
ADD       R1, R0, #1
LDR       R2, R1, #10      ; R2 ← A[11]
```

   Cette technique est généralement employée par les compilateurs, ou lorsque la zone mémoire accédée n'est pas à proximité immédiate du programme.

8.5 Structures de contrôle

8.5.1 Conditionnelles

Les tests de la forme `if (···) { ··· }` sont implémentés à l'aide d'un unique branchement conditionnel. Par exemple,

```
if (x > 5)
  x = x + 1;
```

s'écrit simplement

```
        ; en supposant que x est associé à R0
        ADD     R1, R0, #-5     ; x-5
        BRnz    EndIf           ; x-5 ≤ 0
        ADD     R0, R0, #1
EndIf   ; suite du programme
```

Les tests de la forme `if (···) { ··· } else { ··· }` sont implémentés à l'aide d'un branchement conditionnel et d'un branchement inconditionnel. Par exemple,

```
if (x == 5)
  x = x + 1;
else
  x = x - 1;
```

Peut s'écrire

```
        ; en supposant que x est associé à R0
        ADD     R1, R0, #-5     ; x-5
        BRnp    Else            ; x-5 ≠ 0
        ADD     R0, R0, #1
        BRnzp   EndIf           ; saute la branche else
Else    ADD     R0, R0, #-1
EndIf   ; suite du programme
```

8.5.2 Boucles

Les boucles `while (···) { ··· }` et `for (··· ; ··· ; ···) { ··· }` se traduisent à priori par deux branchements. En effet, le test doit avoir lieu avant l'exécution du corps de boucle. Par exemple,

```
while (x < 5)
  x = x + 1;
```

Peut s'écrire

```
        ; en supposant que x est associé à R0
While   ADD     R1, R0, #-5     ; x-5
        BRzp    EndWhile        ; x-5 ≥ 0
        ADD     R0, R0, #1
        BRnzp   While           ; it
EndWhile; suite du programme
```

En revanche, les boucles `do { ··· } while (···)` s'exécutent au moins une fois et peuvent être écrites à l'aide d'un unique branchement.

```
do
  x = x + 1;
while (x < 5)
```

Peut s'écrire

```
        ; en supposant que x est associé à R0
Do      ADD     R0, R0, #1
        ADD     R1, R0, #-5     ; x-5
        BRn     Do              ; x-5 < 0
        ; suite du programme
```

On remarquera que de nombreuses boucles `while` ou `for` se convertissent aisément en boucles `do`. Le compilateur C peut en tirer parti pour optimiser le code assembleur généré.

8.5.3 Appel de procédures et de fonctions

Les aspects procéduraux des langages de programmation sont plus délicats à gérer au niveau assembleur, surtout lorsqu'il s'agit de générer un programme optimisé. Les problèmes rencontrés sont de trois types :

1. le passage des paramètres et de la valeur de retour, s'il y en a ;
2. la sauvegarde de l'adresse de retour, stockée dans `R7` par les instructions `JSR` et `JSRR` et utilisée par `RET` pour retourner au programme appelant ;
3. la coexistence des valeurs stockées dans les registres par le programme appelant et la procédure appelée.

Le premier problème est facilement résolu en choisissant des registres déterminés pour passer les arguments et récupérer la valeur de retour d'une fonction. Par exemple, le programme suivant effectue la multiplication de deux variables `A` et `B` à l'aide d'une fonction :

```
        .ORIG   x3000
        LD      R0, A           ; premier paramètre
        LD      R1, B           ; deuxième paramètre
        JSR     Mult            ; appel de la fonction Mult
        HALT
A       .FILL   2
B       .FILL   3
        ; fonction de multiplication R2←R0× R1
Mult    AND R2, R2, #0
Loop    ADD R2, R2, R1
        ADD     R0, R0, #-1
        BRp     Loop
        RET
```

Ce mécanisme est très efficace car il ne fait pas intervenir la mémoire pour stocker les paramètres.

Le deuxième problème ne se pose que dans le cas d'appels imbriqués. Si ces appels ne sont *pas récursifs*, il est possible de sauvegarder R7 dans d'autres registres où dans des emplacements mémoire prédéterminés lors des appels successifs.

Le troisième problème nécessite la plupart du temps de faire appel à la mémoire pour sauvegarder les données qui ne « tiennent pas » dans les 8 registres du LC-2 (7 en pratique, car R7 est déjà associé à l'adresse de retour). On a alors le choix entre la sauvegadre des registres de l'appelant — juste avant le saut — ou de l'appelé — juste au début de la procédure. La sauvegarde dans l'appelant permet d'optimiser le nombre de registres néccessitant une sauvegarde, alors que la sauvegarde dans l'appelé facilite grandement le recensement des registres à protéger (et le debuggage) lors d'appels imbriqués.[1] Si les appels ne sont pas récursifs, la sauvegarde proprement dite peut utiliser des emplacements mémoire prédéterminés pour chaque appel ou pour chaque fonction. Voici un exemple de sauvegarde dans l'appelant ou la formule `A*B + C*D` est calculée à l'aide de la fonction `Mult` précédente :

```
        .ORIG     x3000
        LD        R0, A             ; premier paramètre
        LD        R1, B             ; deuxième paramètre
        JSR       Mult              ; appel de la fonction Mult
        ADD       R2, R5, #0        ; sauvegarde de R2
        LD        R0, C             ; premier paramètre
        LD        R1, D             ; deuxième paramètre
        JSR       Mult              ; appel de la fonction Mult
        ADD       R5, R5, R2        ; A*B + C*D
        HALT
A       .FILL     2
B       .FILL     3
C       .FILL     4
D       .FILL     5
        ; fonction de multiplication R2←R0× R1
Mult    AND R2, R2, #0
Loop    ADD R2, R2, R1
        ADD       R0, R0, #-1
        BRp       Loop
        RET
```

8.5.4 La pile

Dans le cas d'appels récursifs, les solutions précédentes ne sont pas suffisantes. De plus, lorsque l'imbrication des appels est suffisante, sans nécessairement être récursive, ces mécanismes de sauvegarde « statiques » se montrent rapidement très complexes ; plus grave, ils conduisent à un gaspillage important de mémoire car de nombreuses zones de la mémoire sont inoccupées la plus grande partie du temps (lorsque la fonction correspondante n'est pas en cours d'exécution). Enfin, lorsque les paramètres sont nombreux ou comportent des données complexes, leur passage par la mémoire est rendu obligatoire.

[1] Les registres contenant les paramètres ne peuvent évidement pas être sauvegardés dans l'appelé.

On appelle *contexte d'une fonction* les données constituées
– de l'adresse de retour,
– des paramètres,
– de la valeur de retour,
– des registres de l'appelant à protéger,
– et des variables locales de cette fonction.

La *pile* est une structure de données[2] offrant une solution générale au problèmes liés à la *sauvegarde du contexte* lors de l'appel de fonctions.

Comme son nom l'indique, elle fonctionne sur un mode LIFO : *Last In First Out*, voir la figure 8. En pratique, il s'agit d'une zone mémoire de taille importante (extensible au besoin, sur les processeurs modernes). Les données ne sont pas déplacées en mémoire : un pointeur/registre dédié indique l'adresse du sommet de la pile ; sur le LC-2, il s'agit du registre `R6`.

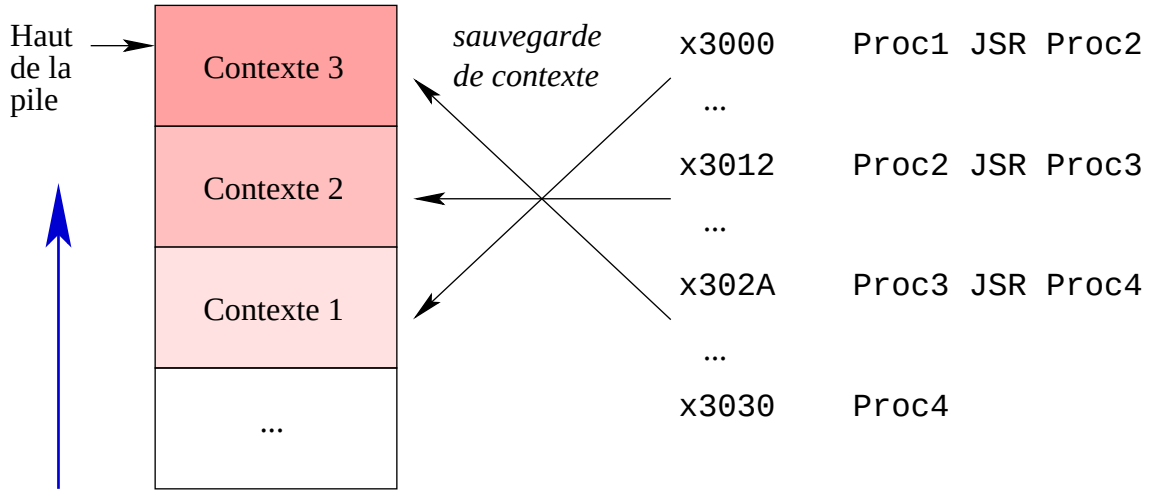

FIG. 8 – Sauvegarde de contexte dans la pile

Par convention, le registre `R6` est incrémenté lors de la sauvegarde de contexte et décrémenté lors de sa libération. On reprend l'exemple de la formule `A*B + C*D` en utilisant la pile :

```
.ORIG     x3000
LD        R0, A           ;
STR       R0, R6, #0      ; passage du paramètre A
LD        R0, B           ;
STR       R0, R6, #1      ; passage du paramètre B
JSR       Mult            ; appel de la fonction Mult
LD        R0, R6, #0      ; lecture de A*B
LD        R1, C           ;
STR       R1, R6, #0      ; passage du paramètre C
LD        R1, D           ;
STR       R1, R6, #1      ; passage du paramètre D
JSR       Mult            ; appel de la fonction Mult
```

[2]En partie gérée par le processeur, voir le chapitre suivant sur le système d'exploitation.

```
        LD      R1, R6, #0      ; lecture de C*D
        ADD     R0, R0, R1      ; A*B + C*D
        HALT
A       .FILL   2
B       .FILL   3
C       .FILL   4
D       .FILL   5
        ; fonction de multiplication R2←R0× R1
Mult    STR     R7, R6, #2      ; sauvegarde de l'adresse de retour
        STR     R0, R6, #3      ; sauvegarde de R0
        STR     R1, R6, #4      ; sauvegarde de R1
        STR     R2, R6, #5      ; sauvegarde de R2
        LDR     R0, R6, #0      ; lecture du premier paramètre
        LDR     R1, R6, #1                                  ; lecture du deuxi
        ADD     R6, R6, #6      ; réservation de 6 mots sur la pile
        AND R2, R2, #0
Loop    ADD R2, R2, R1
        ADD     R0, R0, #-1
        BRp     Loop
        ADD     R6, R6, #-6  lib
        STR     R0, R6, #0      ; valeur de retour
        LDR     R0, R6, #3      ; restauration de R0
        LDR     R1, R6, #4      ; restauration de R1
        LDR     R2, R6, #5      ; restauration de R2
        LDR     R2, R6, #2      ; restauration de R7
        RET
```

En raison du mode d'adressage particulier du LC-2 — seuls les offsets positifs sont autorisés — on notera que l'incrémentation de `R6` s'effectue *après* la sauvegarde du contexte, et que la décrémentation de `R6` s'effectue *avant* la libération de celui-ci.