Programmation en assembleur du LC3

Introduction

Le LC-3 est un processeur developpé dans un but pédagogique par Yale N. Patt et J. Patel dans [*Introduction to Computing Systems : From Bits and Gates to C and Beyond*, McGraw-Hill, 2004].

Des sources et exécutables sont disponibles à l'adresse :
`http://highered.mcgraw-hill.com/sites/0072467509/`

Nous allons nous baser sur le LC-3 pour illustrer les points suivants :

- Comment programme t'on en assembleur ?
- Comment mettre en place des routines ?
- Comment mettre en place une pile d'exécution de programme ?

Ce cours est inspiré de celui d'Olivier Carton (Université Paris Diderot - Paris 7), disponible à l'adresse : `http://www.liafa.jussieu.fr/~carton`

Plan

La mémoire centrale et les registres du LC-3

La mémoire est organisée par mots de 16 bits, avec un adressage sur 16 bits : adresses de $(0000)_H$ à $(FFFF)_H$.

Les registres :

- 8 registres généraux 16 bits : R0,. . . ,R7. Toutefois,
  - R6 est utilisé spécifiquement pour la gestion de la pile d'exécution,
  - R7 est utilisé pour stocker l'adresse de retour d'un appel de fonction.

- quelques registres spécifiques 16 bits :
  - PC (*Program Counter*) et IR (*Instruction Register*)
  - PSR (*Program Status Register*) plusieurs drapeaux binaires,

- Les bits N,Z,P du PSR indiquent si la dernière valeur rangée dans un registre général est strictement négative, zéro ou strictement positive.

Plan

Forme générale d'un programme

Un programme source écrit en langage d'assemblage est un fichier texte qui comporte une suite de lignes. Sur chaque ligne, on trouve :

- soit une *instruction* (`ADD`, `LD`, `BR`,...) ;
- soit une *macro* (`GETC`, `OUT`,...) ;
- soit une *directive d'assemblage* (`.ORIG`, `.BLKW`,...) ;

Une ligne peut être précédée par une *étiquette* pour référencer son adresse.

```
        .ORIG x3000             ; directive pour le début de programme
; partie dédiée au code        <- il s'agit d'un commentaire
loop:   GETC                    ; macro marquée par une étiquette
        LD R1,cmpzero           ; instruction
        ...
        HALT                    ; macro
; partie dédiée aux données
cmpzero: .FILL xFFD0            ; étiquette et directive
        .END                    ; directive
```

Constantes

Il existe deux types de constantes.

- Les *chaînes de caractères* qui apparaissent uniquement après la directive `.STRINGZ` : elles sont délimitées par deux caractères " et implicitement terminées par le caractère nul.
- Les *entiers relatifs* en hexadécimal sont précédés d'un `x` ; sinon, ce sont des décimaux (le préfix # est optionnel). Ils peuvent apparaître comme
  - opérandes immédiats des instructions (attention à la taille des opérandes),
  - paramétres des directives `.ORIG`, `.FILL` et `.BLKW`.

Exemple :

```
.ORIG x3000          ; Constante entière en base 16
AND R2,R1,2          ; Constante entière en base 10
ADD R6,R5,#-1        ; Constante entière négative en base 10
.STRINGZ "Chaîne"    ; Constante chaîne de caractères
```

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.

- `.STRINGZ chaine`

  Réserve un nombre de mots égal à la longueur de la chaîne de caractères plus un caractère nul (code ASCII 0) et y place la chaîne.

- `.BLKW nombre`

  Cette directive réserve le nombre de mots de 16 bits passé en paramètre.

Les interruptions prédéfinies : « appels système »

L'instruction TRAP appelle un gestionnaire d'interruptions mis en place par le petit système d'exploitation du LC-3 : il s'agit donc d'un appel système. Chaque appel système est identifié par une constante sur 8 bits.

Dans l'assembleur du LC-3, on peut utiliser les macros suivantes :

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 l'octet de poids faible de R0
`TRAP x21`	`OUT`	écrit à l'écran le caractère ASCII placé dans l'octet de poids faible de 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 l'octet de poids faible de R0

Plan

Les instructions du LC-3

Les instructions du LC-3 se répartissent en trois classes.

1. Instructions arithmétiques et logiques : ADD, AND, NOT.
2. Instructions de chargement et rangement :
   - LD, ST : *load* et *store*
   - LDR, STR : comme LD et ST mais avec un adressage relatif
   - LEA : *load effective address*
3. Instructions de branchement :
   - BR : branch
   - JSR : jump subroutine
   - TRAP : interruption logicielle
   - RET : de retour de routine

Il y à aussi NOP, pour *no operation*, qui ne fait rien... Pour simplifier les choses, d'autres instructions du jeu d'instruction du LC3 sont ici omises.

Récapitulatif des instructions du LC-3

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
RET (JMP R7)	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										

Notez par exemple que l'addition (ADD) se décline de deux façons :

- ADD DR, SR1, SR2
- ADD DR, SR1, Imm5

La colonne NZP indique les instructions qui mettent à jour les drapeaux NZP.

Plan

LD et ST

On ne va décrire que LD, mais le principe est symétrique pour ST.

- syntaxe : LD DR,label
  $\hookrightarrow$ le label servira à désigner une adresse mémoire AdM.
- action : DR $\leftarrow$ Mem[PC + Sext(PCOffset9)]
  - valeur de PC après son incrémentation lors du chargement ;
    si adI est l'adresse de l'instruction, PC = adI+1.
  - PCOffset9 est un décalage en complément à 2 sur 9 bits.
  - La case mémoire chargée est donc celle à l'adresse :
    $$\text{adM} = \text{adI} + 1 + \text{Sext(PCOffset9)}, \quad \text{adI} - 255 \leq \text{adM} \leq \text{adI} + 256.$$

C'est l'assembleur qui va se charger du calcul de PCOffset9 : le programmeur en langage d'assemblage utilise des labels. Notez que la distance entre une instruction LD et la case mémoire dont elle peut charger le contenu est limitée.

LDR et STR

On ne va décrire que LDR, mais le principe est symétrique pour STR.

- syntaxe : LDR DR,baseR,Offset6
- action : DR $\leftarrow$ Mem[baseR + Sext(PCOffset6)]

Avec LDR et STR, on peut accéder à toutes les adresses de la mémoire, ce qui permet de lever les limitations de LD/ST. On utilise ces instructions pour

- manipuler des données sur la pile, comme les variables locales des fonctions ;
- accéder aux éléments d'un tableau ou d'une chaîne de caractères.

baseR est utilisé comme un pointeur. Par contre, comment initialiser baseR ?

LEA

LEA permet de charger une adresse dans un registre général.

- syntaxe : LEA DR,label
- action : DR $\leftarrow$ PC + Sext(PCOffset9)

L'adresse est calculée comme pour LD, mais seule l'adresse est chargée dans DR.

Cette instruction est utile pour :

- charger l'adresse d'un tableau dans un registre,
- charger l'adresse de la base de la pile d'exécution.

On se sert donc de LEA pour initialiser un pointeur.

Plan

BR

On réalise avec BR des *branchements inconditionnels ou conditionnels*.

Trois drapeaux N, Z et P (majuscules) du PSR sont mis à jour dès qu'une nouvelle valeur est chargée dans l'un des registres généraux :

- N passe à 1 si cette valeur est strictement négative,
- Z passe à 1 si cette valeur est zéro,
- P passe à 1 si cette valeur est strictement positive.

L'instruction BR contient trois bits n, z et p (minuscules) :

- syntaxe : BR[n][z][p] label
- action : si cond alors PC $\leftarrow$ PC+Sext(PCOffset9)
  avec cond $= (\overline{n}\,\overline{z}\,\overline{p}) + (n = N) + (z = Z) + (p = P)$

L'assembleur se charge de calculer PCOffset9 d'après le label fourni : il faut néanmoins garder à l'esprit le fait que la distance du saut est limitée...

Plan

Un exemple

On veut écrire un programme qui range à l'adresse désignée par `res` la longueur d'une chaîne de caractères se trouvant à l'adresse `string`. Point de départ :

```
        .ORIG x3000
        ...       ; Ici viendra notre code
        HALT      ; Pour mettre fin à l'exécution
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

On va traduire le pseudo-code suivant :

```
R0 <- string;  // R0 pointe vers le début de la chaîne
R1 <- 0;       // Le compteur R1 est initialisé à 0
while((R2 <- Mem[R0]) != '\0') {
  R0 <- R0+1;  // Incrémentation du pointeur
  R1 <- R1+1;  // Incrémentation du compteur
}
res <- R1;     // Rangement du résultat
```

Cela donne au final le programme suivant :

```
        .ORIG x3000
        LEA R0,string   ; Initialisation du pointeur R0
        AND R1,R1,0     ; Le compteur R1 est initialisé à 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractere pointé par R0
        BRz end         ; Test de sortie de boucle
        ADD R0,R0,1     ; Incrémentation du pointeur
        ADD R1,R1,1     ; Incrémentation du compteur
        BR loop
end:    ST R1,res
        HALT
; Chaîne constante
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

Plan

Les routines

En assembleur, une routine est juste une portion de code :

- l'adresse de la première instruction de cette portion de code est connue, et marquée par un label ;
- on peut effectuer un saut vers cette portion de code (instruction JSR), qui constitue un appel à la routine ;
- à la fin de l'exécution de la routine, une instruction de retour (RET) fait repasser PC à l'adresse suivant celle de l'instruction d'appel de la routine.

Une routine correspond un peu à une fonction dans un langage de haut niveau : en fait, le programmeur doit gérer beaucoup de choses lui-même. . .

JSR et RET

Lorsqu'une routine est appelée, il faut mémoriser l'adresse à laquelle doit revenir s'exécuter le programme à la fin de la routine.

JSR stocke l'adresse de retour dans R7, puis effectue un branchement à l'adresse passée comme opérande. RET permet de retourner à la routine appelante, par un saut à l'adresse contenue dans R7.

Dans le programme, on fera par exemple appel à la routine `sub` :

```
        ...
        JSR sub     ; R7 <- PC ; PC <- PC + SEXT(PCoffset11)
        ...
```

Dans la routine, le retour au programme principal se fera avec `RET` :

```
; routine sub
sub:    ...
        RET         ; PC <- R7
```

Un exemple de routine

On reprend le programme pour calculer la longueur d'une chaîne de caractères :

```
        .ORIG x3000
        LEA R0,string   ; Initialisation du pointeur R0
        AND R1,R1,0     ; Le compteur R1 est initialisé à 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractere pointé par R0
        BRz end         ; Test de sortie de boucle
        ADD R0,R0,1     ; Incrémentation du pointeur
        ADD R1,R1,1     ; Incrémentation du compteur
        BR loop
end:    ST R1,res
        HALT
; Chaîne constante
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

On va écrire une routine `strlen` pour calculer la longueur d'une chaîne de caractères dont l'adresse est placée dans R0, et qui rend son résultat dans R0.

Solution

```
; Programme principal
        LEA R0,chaine1  ; Chargement dans R0 de l'adresse de la chaîne
        JSR strlen      ; Appel de la routine
        ST R0,lg1
        HALT
; Données
chaine1: .STRINGZ "Hello World"
lg1:    .BLKW #1

; Routine pour calculer la longueur d'une chaîne terminée par '\0'.
; paramètre d'entrée  : R0 adresse de la chaîne
; paramètre de sortie : R0 longueur de la chaîne
strlen: AND R1,R1,0     ; Mise à 0 du compteur : c = 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur
        ADD R1,R1,1     ; Incrémentation du compteur
        BR loop
fini:   ADD R0,R1,0     ; R0 <- R1
        RET             ; Retour à l'appelant
```

Plan

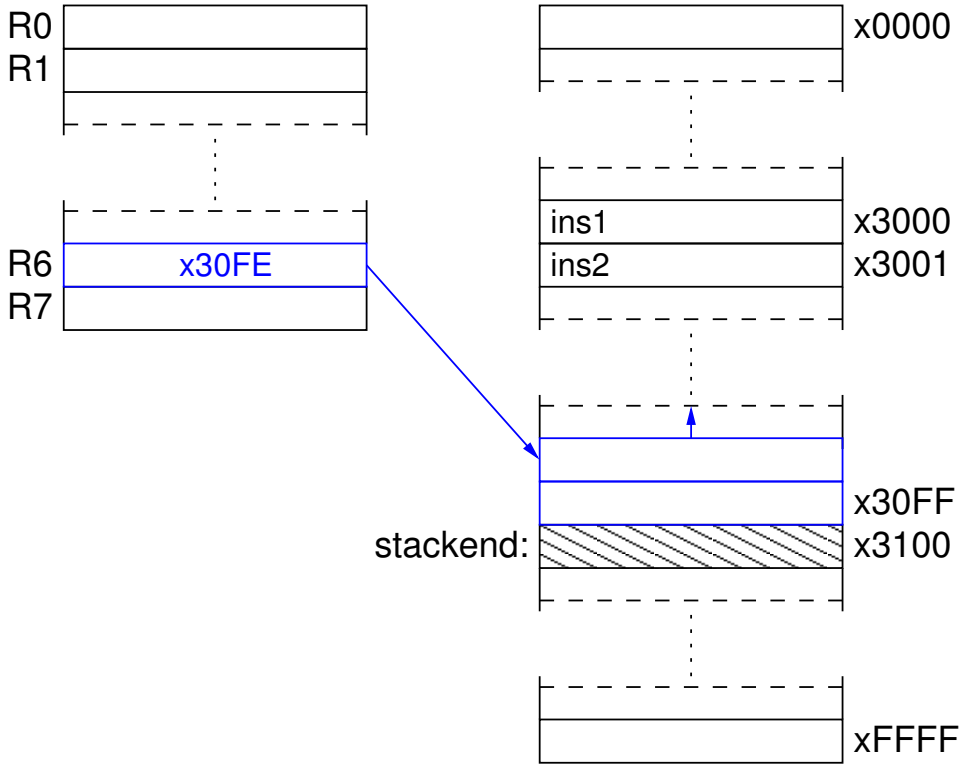
Quand une routine en appelle une autre, elle doit sauvegarder son adresse de retour avant l'appel, et la restaurer après.

```
; programme principal
      ...
      JSR sub1
      ...

; routine sub1
sub1: ...
      JSR sub2
      ...
      RET       ; Problème : ici R7 ne contient l'adresse de retour vers sub1 !
                ; (puisque sub2 a été appelée juste avant...)

; routine sub2
sub1: ...
      RET
```

Une méthode est de mettre en place une *pile d'exécution*, qui permet de gérer même les appels récursifs. Il faut aussi sauvegarder les registres dont on souhaite retrouver le contenu après l'exécution d'une routine.

On donne ici une manière de gérer une pile d'exécution sur le LC-3 :

- R6 est réservé à la gestion de la pile.
- R6 est initialisé à un adresse de « fond de pile » désignée par stackend.
- R6 donne toujours l'adresse de la dernière valeur mise sur la pile.
- La pile croît dans le sens des adresses décroissantes.



Mise en place de la pile d'exécution :

```
          .ORIG x3000
; Programme principal
main:     LD R6,spinit   ; on initialise le pointeur de pile
          ...
          HALT

; Gestion de la pile
spinit:   .FILL stackend
          .BLKW #15
stackend: .BLKW #1       ; adresse du fond de la pile

          .END
```

Empilement d'un registre Ri :

```
ADD R6,R6,-1   ; déplacement du sommet de pile
STR Ri,R6,0    ; rangement de la valeur
```

Dépilement du sommet de pile dans un registre Ri :

```
LDR Ri,R6,0    ; récupération de la valeur
ADD R6,R6,1    ; restauration du sommet de pile
```

Un exemple utilisant la pile

On reprend notre exemple, d'abord sous la forme d'un pseudocode :

```
main(void) {
  ...
  sub1();
  ...
}

sub1(void) {
  ...
  sub2();
  ...
  return;
}

sub2(void) {
  ...
  return;
}
```

On ajoute la gestion de la pile d'exécution :

```
main(void) {
  ...
  sub1();
  ...
}
sub1(void) {
  push(R7);              // sauvegarde de l'adresse de retour R7
  ...
  sub2();
  ...
  R7 <- pop();           // restauration de l'adresse de retour
  return;
}
sub2(void) {
  ...
  return;
}
```

Rem : si sub2 faisait appel à d'autres routines, il serait nécessaire qu'elle sauvegarde aussi son adresse de retour !

En langage d'assemblage du LC3, cela donne pour le programme principal :

```
; Programme principal
main:     LD R6,spinit   ; on initialise le pointeur de pile
          ...
          JSR sub1       ; appel à la routine sub1
          ...
          HALT

; Mémoire réservée à la pile
spinit:   .FILL stackend
          .BLKW #15
stackend: .BLKW #1       ; adresse du fond de la pile
```

Et pour les deux routines sub1 et sub2 :

```
; routine sub1
sub1: ADD R6,R6,#-1      ;
      STR R7,R6,#0       ; sauvegarde de l'adresse de retour R7.
      ...
      JSR sub2
      ...
      LDR R7,R6,#0       ; restaure le registre R7.
      ADD R6,R6,#1       ;
      RET                ; retour à l'appelant

; routine sub2
sub2: ...
      RET                ; retour à l'appelant
```

Conclusion

Nous avons vu :

- un exemple de langage d'assemblage ;
- comment peuvent être mis en place les appels de routines et la pile ;
  ↪ utile pour sauvegarder variables, paramètres et adresse de retour...

Les étapes suivantes seraient de voir comment transformer :

- du code en langage d'assemblage vers le langage machine
  ↪ assemblage ;
- du code en langage de haut niveau vers un langage d'assemblage
  ↪ compilation.