ARCHI – Architecture des ordinateurs

Sylvain Brandel
2025 – 2026
sylvain.brandel@univ-lyon1.fr

Partie 6

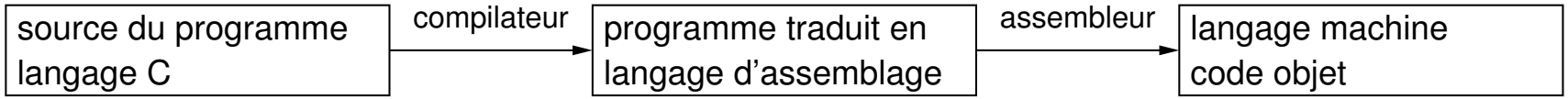
LANGAGE D'ASSEMBLAGE DU LC-3

Instructions arithmétiques et logiques
Accès mémoire
Branchements, boucles
Routines, piles

LC-3

- LC-3 : processeur à but pédagogique
- Mémoire
  - Mots de 16 bits avec adressage sur 16 bits
    → 2^{16} adresses, de $(0000)_H$ à $(FFFF)_H$ → 64Ki adresses
- Registres généraux 16 bits
  - 8 registres généraux : R0 .. R7
  - R6 : reg. spécifique (gestion de la pile d'exécution)
  - R7 : reg. spécifique (adresse de retour d'un appel de fonction)
- Registres spécifiques 16 bits
  - PC : Program Counter
  - IR : Instruction Register
  - PSR : Program Status Register (drapeaux binaires, N, Z, P notamment)
  - USP : User Stack Pointer (sauvegarde R6 pour l'utilisateur)
  - SSP : System Stack Pointer (sauvegarde R6 pour le super-utilisateur)

LC-3



- Programme écrit en langage d'assemblage :
  - Directives d'assemblage
  - Interruptions (macros)
  - Instructions

[label:]{ directive d'assemblage | macro | instruction } [; commentaire]

```
          .ORIG x3000       ; directive pour le début de programme

loop:     GETC              ; macro marquée par une étiquette
          OUT               ; macro
          LD R1, cmpz       ; instruction
          ...
          HALT              ; macro

cmpz:     .FILL xFFD0       ; étiquette et directive
msg:      .STRINGZ "hello"  ; étiquette et directive

          .END              ; directive
```

PennSim

- Lancement du simulateur (terminal) : `java -jar PennSimm.jar`

`as ex0.asm`	Assemblage du fichier `ex0.asm` → `ex0.obj` et `ex0.sym`
`load ex0.obj`	Copie du code machine `ex0.obj` en mémoire
`list x3000`	Se positionne à l'adresse `x3000`
`set PC x3000`	Met la valeur `x3000` dans le registre `PC`
`as lc3os.asm` `load lc3os.obj`	Assemble et charge le mini OS LC3 en mémoire (nécessaire pour les interruptions)

Directives d'assemblage

- Utilisées pour l'assemblage
- Ne figurent pas dans le langage machine

`.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 et y place la chaîne.
`.BLKW nombre`	Réserve le nombre de mots de 16 bits passé en paramètre.

Interruptions

- Appels système via l'OS du LC-3
- Instruction `TRAP {constante 8 bits}` (et macro)

`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
`TRAP x25`	`HALT`	Termine un programme, rend la main à l'OS

Constantes

- Deux types de constantes
- Chaînes de caractères
  - Uniquement après la directive `.STRINGZ`
  - Délimitées par deux caractères " et implicitement terminées par le caractère nul (dont le code ASCII est zéro).
- Entiers relatifs
  - En hexadécimal : précédés d'un `x`
  - En décimal : précédés d'un `#` ou de rien
  - Peuvent apparaître comme
    - Opérandes immédiats des instructions (attention à la taille des opérandes)
    - Paramètres des directives `.ORIG`, `.FILL` et `.BLKW`

```
.ORIG x3000         ; Constante entière en base 16
AND R2,R1,#2        ; Constante entière en base 10
ADD R3,R2,15        ; constante entière en base 10
ADD R6,R5,#-1       ; Constante entière négative en base 10
.STRINGZ "Chaîne"   ; Constante chaîne de caractères
```

Instructions

ISA (Instruction Set Architecture)

	Syntaxe	action	NZP	codage															
				opcode				arguments											
				F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Arith - logique	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				
charg. rang.	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					
branchement	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										

Instructions arithmétiques et logiques

Arith - logique	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				

- C'est tout, pas de soustraction, pas d'affectation directe
- Deux instructions ADD et AND
  - Opérandes : deux registres
  - Opérandes : un registre et une constante sur 5 bits en complément à 2
- NZP
  - N, Z, P = 1 si la dernière valeur rangée dans un registre général est strictement négative, zéro ou strictement positive
- Affectation (ATTENTION R0 ← 15 directement pas possible)
  - `AND R0, R0, 0` puis `ADD R0, R0, 15`
- Soustraction
  - Addition du complément à 1 + 1

```
; exemple
AND R0, R0, 0
ADD R0, R0, 15
AND R1, R1, 0
ADD R1, R1, -4
ADD R4, R0, R1
```

Instructions de chargement et rangement

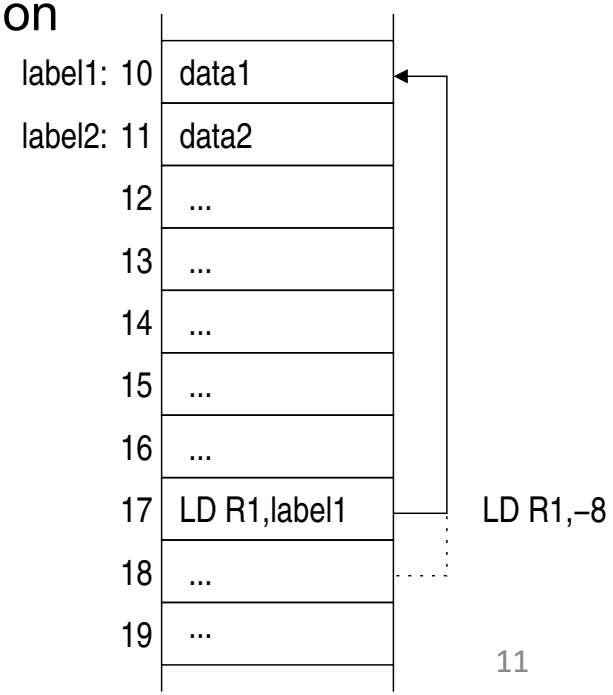
charg. rang.	LEA DR,label	DR <- PC + SEXT(PCoffset9)	*	1	1	1	0	DR	PCoffset9		
	LD DR,label	DR <- mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR	PCoffset9		
	ST SR,label	mem[PC + SEXT(PCoffset9)] <- SR		0	0	1	1	SR	PCoffset9		
	LDR DR,BaseR,Offset6	DR <- mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR	BaseR	Offset6	
	STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] <- SR		0	1	1	1	SR	BaseR	Offset6	

- LEA : Load Effective Address
- LD / ST : Load / Store
- LDR / STR : Load / Store avec adressage relatif

Instructions de chargement et rangement

charg. rang.	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	

- LD
  - DR <- contenu de la case label
  - adM : label
  - adI : adresse de l'instruction
  - PC incrémenté après le chargement de l'instruction
  - PC = adI + 1
  - -256 ≤ SEXT(PCoffset9) ≤ 255
  - adM = PC + SEXT(PCoffset9)
    = adI + 1 + SEXT(PCoffset9)
  - adI – 255 ≤ adM ≤ adI + 256
    → distance entre instruction LD
    et case mémoire limitée
- ST : pareil



Instructions de chargement et rangement

charg. rang.	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	

- LDR
  - Adressage à partir d'un registre et non de l'instruction courante
    → Accès à toute la mémoire
  - Utilisation
    - Manipulation des données sur la pile (paramètres d'une fonction)
    - Accès aux éléments d'un tableau
  - BaseR : pointeur
  - Problème : initialiser BaseR

- STR : pareil

Instructions de chargement et rangement

charg. rang.	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	

- LEA (Load Effective Address)
  - DR <- adresse de la case label
  - L'adresse est calculée comme pour LD mais seule l'adresse est chargée dans DR
  - Utilisation :
    - Charger l'adresse de la pile d'exécution
    - Charger l'adresse d'un tableau dans un registre

  → LEA sert à initialiser un pointeur

Instructions de branchement

branchement	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	0
	JMP BaseR	PC <- BaseR		1	1	0	0	0	0	0	BaseR			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											
	TRAP Trapvect8	R7 <- PC ; PC <- mem[Trapvect8]		1	1	1	1	0	0	0	0	Trapvect8								

- BR : BRanchement
- NOP : No OPeration
- RET : RETour de routine
- JSR : Jump to Sub Routine
- TRAP : interruption logicielle

Instructions de branchement

branchement	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	0
	JMP BaseR	PC <- BaseR		1	1	0	0	0	0	0	BaseR			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											
	TRAP Trapvect8	R7 <- PC ; PC <- mem[Trapvect8]		1	1	1	1	0	0	0	0	Trapvect8								

- BR : Branchement
  - Branchements non conditionnels : BR
  - Branchements conditionnels : BRn, BRz, BRp
- Le registre PSR contient – entre autres – 3 drapeaux mis à jour dès qu'une nouvelle valeur est chargée dans un registre général
  - 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
- Si $((\bar{n}\bar{z}\bar{p}) + (n = N) + (z = Z) + (p = P) = true)$

  Alors PC ← label

While

- BR[n][z][p] : branchement si dernière valeur chargée dans un registre est < 0, = 0 ou > 0
- Donc forcément while (n cmp 0)

```
int i = 9;
while (i >= 0) {
  // corps de la boucle
  i = i - 1;
}
```

```
        AND R1, R1, 0     ;
        ADD R1, R1, 9     ; R1 <- 9
loop:   BRn endloop
        ; corps de la boucle
        ADD R1, R1, -1    ; R1 <- R1 - 1
        BR loop
endloop:
```

```
int i = 9;
while (i > 0) {
  // corps de la boucle
  i = i - 1;
}
```

```
        AND R1, R1, 0     ;
        ADD R1, R1, 9     ; R1 <- 9
loop:   BRz endloop
        ; corps de la boucle
        ADD R1, R1, -1    ; R1 <- R1 - 1
        BR loop
endloop:
```

While

- BR[n][z][p] : branchement si dernière valeur chargée dans un registre est < 0, = 0 ou > 0
- Donc forcément while (n cmp 0)

```
int i = -9;
while (i <= 0) {
  // corps de la boucle
  i = i + 1;
}
```

```
          AND R1, R1, 0     ;
          ADD R1, R1, -9    ; R1 <- -9
loop:     BRp endloop
          ; corps de la boucle
          ADD R1, R1, 1     ; R1 <- R1 + 1
          BR loop
endloop:
```

```
int i = -9;
while (i < 0) {
  // corps de la boucle
  i = i + 1;
}
```

```
          AND R1, R1, 0     ;
          ADD R1, R1, -9    ; R1 <- -9
loop:     BRz endloop
          ; corps de la boucle
          ADD R1, R1, 1     ; R1 <- R1 + 1
          BR loop
endloop:
```

Exemple 1

- Afficher les entiers de 9 à 0

```
cout << "Affichage des entiers de 9 à 0 :\n";
int i = 9;
while (i >= 0) {
  cout << i + '0';
  i = i - 1;
}
cout << "\nFin de l'affichage\n";
```

- R0 : affichage ; R1 : compteur ; R2 : '0'

```
print("Affichage des entiers de 9 à 0 :\n");
R2 <- '0';
R1 <- 9;
while(R1 >= 0) {
  R0 <- R1 + R2;
  print(R0);
  R1 <- R1 - 1;
}
print("\nFin de l'affichage\n");
```

Exemple 1

```
          .ORIG x3000
; partie dédiée au code
          LEA R0,msg0       ; charge l'adresse effective désignée par msg0 dans R0
          PUTS              ; affiche la chaîne de car. pointée par R0 (TRAP x22)
          LD R2,carzero     ; charge '0' dans R2
          AND R1,R1,0       ;
          ADD R1,R1,9       ; R1 <- 9
loop:     BRn endloop       ; si R1 < 0, alors on sort de la boucle
          ADD R0,R1,R2      ; R0 <- R1+'0' = (code ascii du chiffre R1)
          OUT               ; affiche le caractère contenu dans R0 (TRAP x21)
          ADD R1,R1,-1      ; R1 <- R1-1, maj de NZP d'après R1
          BR loop           ; retour au début de boucle
endloop: LEA R0,msg1        ; charge l'adresse effective désignée par msg1 dans R0
          PUTS              ; affiche la chaîne de car. pointée par R0 (TRAP x22)
          HALT              ; termine le programme
; partie dédiée aux données
carzero: .FILL x30          ; code ASCII du caractère '0' (48 en décimal)
msg0:    .STRINGZ "Affichage des entiers de 9 à 0 :\n"
msg1:    .STRINGZ "\nFin de l'affichage !\n"
          .END              ; marque la fin du bloc de code
```

Exemple 2

- Compte le nombre de caractères d'une chaîne et met le résultat en mémoire

```
          .ORIG x3000
          ; ici viendra notre code
          HALT
string:  .STRINGZ "Hello World »    ; la chaine dont on veut calculer la
                                    ; longueur
res:      .BLKW #1                  ; le résultat sera mis ici (1 case)
          .END
```

- R0 : pointeur de chaîne ; R1 : compteur ; R2 : caractère courant

```
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
```

Exemple 2

```
        .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 caractère 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
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

Instructions

ISA (Instruction Set Architecture)

	Syntaxe	action	NZP	codage															
				opcode				arguments											
				F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Arith - logique	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				
charg. rang.	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					
branchement	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										

Exemple 2

- Compte le nombre de caractères d'une chaîne et met le résultat en mémoire

```
          .ORIG x3000
          ; ici viendra notre code
          HALT
string:  .STRINGZ "Hello World »    ; la chaine dont on veut calculer la
                                    ; longueur
res:      .BLKW #1                  ; le résultat sera mis ici (1 case)
          .END
```

- R0 : pointeur de chaîne ; R1 : compteur ; R2 : caractère courant

```
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
```

Exemple 2

```
          .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 caractère 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
string:   .STRINGZ "Hello World"
res:      .BLKW #1
          .END
```

- Maintenant, on a envie de réutiliser le calcul de la longueur

Routines

- Plusieurs fois la même opération / même code ? → routine.

- Routine : bloc de code accessible
  - Étiquette à la première instruction de la routine
  - Appel par saut (`JSR`)
    → Modification de PC
  - Retour (`RET`) à l'instruction suivant l'appel
    → Connaitre l'adresse de retour Mémorisation (R7)

- Routine : fonction dans un langage de haut niveau

Routines

branchement	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
	JMP BaseR	PC <- BaseR		1	1	0	0	0	0	0	BaseR			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										
	TRAP Trapvect8	R7 <- PC ; PC <- mem[Trapvect8]		1	1	1	1	0	0	0	0	Trapvect8							

- JSR
  - Stocke l'adresse de retour dans R7
  - Modifie PC ← adresse de label

branchement

```
; programme principal
...
JSR sub  ; adresse de l'instruction suivante dans R7
...
```

- RET (équivalent à JMP R7)
  - Retour à la routine appelante
  - Modifie PC ← R7

ATTENTION AUX TRAP

```
sub: ...
     ...
     RET      ; retour au programme principal
```

Exemple 2 bis

- On reprend le calcul de la longueur d'une chaine

```
        .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 caractère 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
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

- Routine pour le calcul de la longueur
  - Adresse de la chaine dans R0
  - Résultat dans R0

Exemple 2 bis

```
        .ORIG x3000
        LEA R0,string ; Initialisation du pointeur R0
        JSR length ; saut vers la routine
        ST R0,res
        HALT
; données
string:  .STRINGZ "Hello"
res:     .BLKW #1

; Routine pour calculer la longueur d'une chaîne (terminée par '\0')
; Entrée : R0 adresse de la chaîne
; Sortie : R0 longueur de la chaîne
length:  AND R1, R1, 0
loop:    LDR R2, R0, 0
         BRz end
         ADD R0, R0, 1
         ADD R1, R1, 1
         BR loop
end:     ADD R0, R1, 0
         RET
```

Routines imbriquées

- Appel à une routine dans une routine ? TRAP dans une routine ?

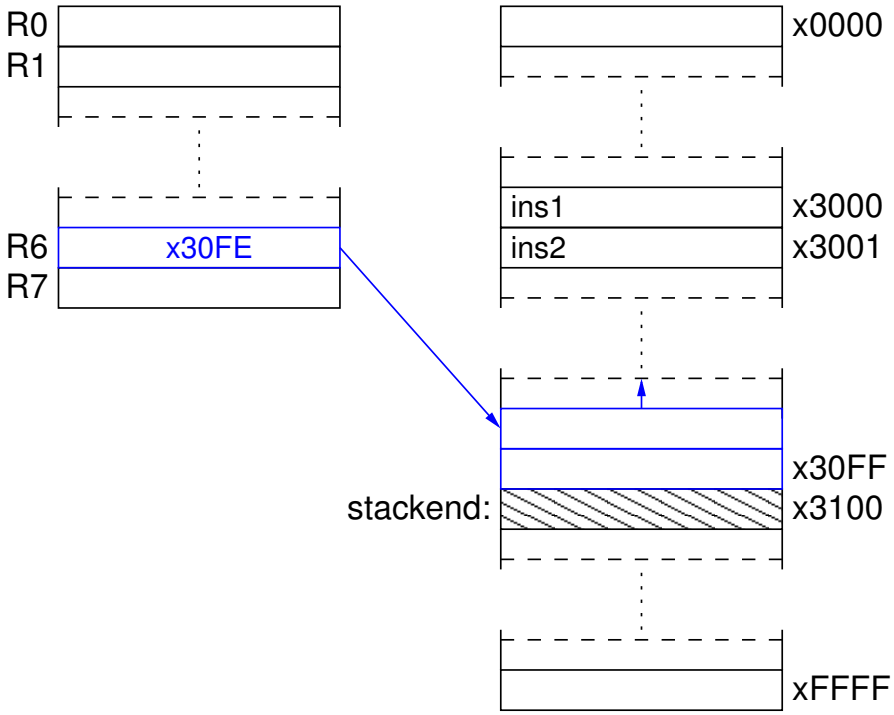
```
; programme principal
          ...
          JSR sub1
          ...

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

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

- Pareil pour les registres …

  → Pile d'exécution

Pile d'exécution

- Appel à une routine dans une routine ? R7 change → pile
- Pile : structure pour mémoriser adresses et registres
  - Sommet (dernier ajouté) pointé par R6 (fond : stackend)
  - Push → nouveau sommet → modification de R6 + mémorisation
  - Pop → lecture + nouveau sommet → modification de R6
  - La pile croît dans le sens des adresses décroissantes
  - Installation :
    - De la place (assez, pas trop)
    - Un fond
    - Initialisations



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
```

- Push
  - Bouger le pointeur de sommet
  - Stocker le registre à sauver
    → Avant appel de la routine

```
ADD R6, R6, -1
STR Rx, R6, 0
```

- Pop
  - Récupérer la donnée au sommet
  - Bouger le pointeur de sommet
    → Après retour de la routine

```
LDR Rx, R6, 0
ADD R6, R6, 1
```

Routines imbriquées

```
; programme principal
          ...
         LD R6, spinit      ; initialisation du pointeur de pile
         JSR sub1
         ...

; routine 1
sub1:    ...
          ADD R6, R6, -1    ; sauvegarde R7
         STR R7, R6, 0
         JSR sub2
         LDR R7, R6, 0      ; restauration R7
         ADD R6, R6, 1
          ...
         RET

; routine 2
sub2:    ...
         RET

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

Exemple 3

- Somme des entiers de 1 à n version récursive

```
int sum(int n) {
  if (n==0) return 0; else return n + sum(n-1);
}
```

- R1 : paramètre d'entrée, R0 : paramètre de sortie

```
routine sum(R1) {
  push(R7);            // Sauvegarde de R7 sur la pile
  push(R1);            // Sauvegarde de R1 sur la pile
  if (R1 == 0) {
    R0 <- 0;
  }
  else {
    R1 <- R1 - 1;      // On prépare le paramètre de l'appel récursif
    call sum(R1);      // Appel récursif
    R1 <- top();       // R1 reprend sa valeur depuis le sommet de pile
    R0 <- R0 + R1;
  }
  R1 <- pop();         // Restauration de R1 depuis la pile
  R7 <- pop();         // Restauration de R7 depuis la pile
  return;              // Résultat retourné via R0
}
```

Exemple 3

```
; Programme principal
main:    LD R6,spinit      ; on initialise le pointeur de pile
         LD R1,n           ; on initialise R1 avant l'appel à la routine sum
         JSR sum           ; appel à la routine sum
         ST R0,r           ; on stocke le résultat à l'adresse r
         HALT

; Données
n:       .FILL #5
r:       .BLKW #1

; Routine sum
; paramètre d'entrée : R1
; paramètre de sortie : R0
sum:     ...
         RET               ; on retourne à l'appelant

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

Exemple 3

```
; Routine sum.
; Paramètre d'entrée  : R1
; Paramètre de sortie : R0
sum:    ADD R6,R6,#-1
        STR R7,R6,#0     ; On sauvegarde l'adresse de retour R7.
        ADD R6,R6,#-1
        STR R1,R6,#0     ; On sauvegarde l'adresse de retour R1.

if:     ADD R1,R1,#0     ; Mise à jour du PSR d'après R1.
        BRnp else        ; Si R1 != 0, branchement sur le bloc else
        AND R0,R0,#0     ; Si R1 == 0, la valeur de retour est 0,
        BR endif         ;             et on ne fait pas d'appel récursif.

else:   ADD R1,R1,#-1    ; Dans le cas R1 == 0, on prépare l'appel récursif.
        JSR sum          ; Appel récursif.
        LDR R1,R6,#0     ; On récupère la valeur de R1 avant l'appel récursif.
        ADD R0,R0,R1     ; On met le résultat dans le paramètre de sortie R0.

endif:  LDR R1,R6,#0     ; On restaure le registre R1.
        ADD R6,R6,#1
        LDR R7,R6,#0     ; On restaure le registre R7.
        ADD R6,R6,#1
        RET              ; On retourne à l'appelant
```