

ARCHI – Architecture des ordinateurs

Sylvain Brandel

2024 – 2025

sylvain.brandel@univ-lyon1.fr



CM 9

LANGAGE D'ASSEMBLAGE DU LC-3

Partie 1

Instructions arithmétiques et logiques

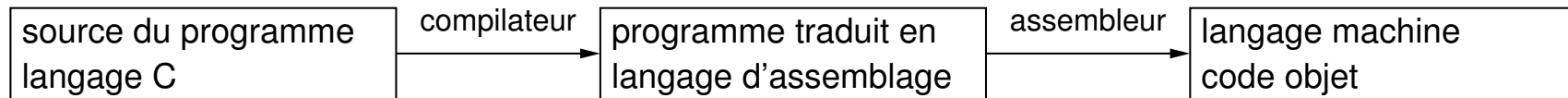
Accès mémoire

Branchements / boucles

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`

<code>as ex0.asm</code>	Assemblage du fichier <code>ex0.asm</code> → <code>ex0.obj</code> et <code>ex0.sym</code>
<code>load ex0.obj</code>	Copie du code machine <code>ex0.obj</code> en mémoire
<code>list x3000</code>	Se positionne à l'adresse <code>x3000</code>
<code>set PC x3000</code>	Met la valeur <code>x3000</code> dans le registre PC
<code>as lc3os.asm</code> <code>load lc3os.obj</code>	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

<code>.ORIG adresse</code>	Spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit
<code>.END</code>	Termine un bloc d'instructions.
<code>.FILL valeur</code>	Réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre
<code>.STRINGZ chaîne</code>	Réserve un nombre de mots égal à la longueur de la chaîne de caractères plus un caractère nul et y place la chaîne.
<code>.BLKW nombre</code>	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	N/Z/P	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	
	RET	PC <- R7		1	1	0	0	0	0	0	1	1	1	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

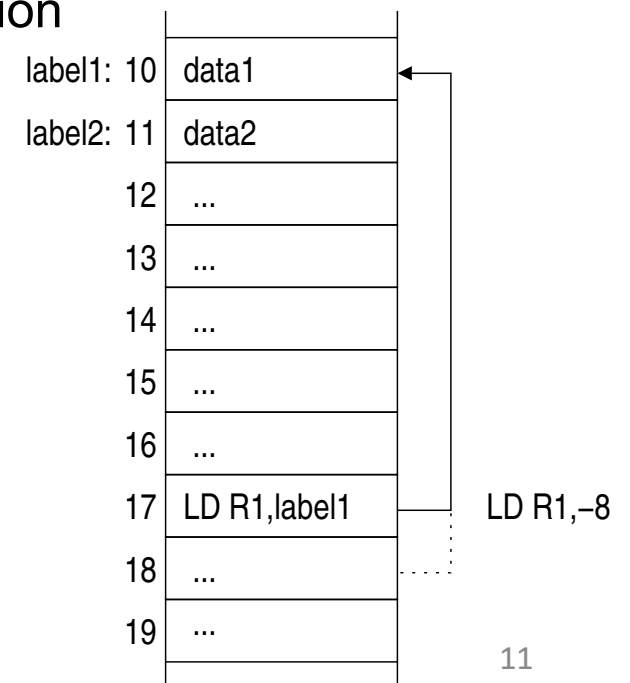
Instructions de chargement et rangement

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
- $-256 \leq \text{SEXT}(\text{PCoffset9}) \leq 255$
- adM = PC + SEXT(PCoffset9)
= adI + 1 + SEXT(PCoffset9)
- $\text{adI} - 255 \leq \text{adM} \leq \text{adI} + 256$
→ 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
	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
	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	
	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
	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 **N**égative
 - **Z** passe à 1 si cette valeur est **Z**éro
 - **P** passe à 1 si cette valeur est strictement **P**ositive
- Si $((\bar{n}\bar{z}\bar{p}) + (n = N) + (z = Z) + (p = P) = true)$
 Alors $PC \leftarrow label$

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; // Incrémenter le pointeur
    R1 <- R1+1; // Incrémenter le 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
```