Plan

- Représentation des nombres
- Circuits logiques
- Unité Arithmétique et Logique
- Notions de temps et de mémorisation
- Contrôle et jonction des composants
- Evolution des ordinateurs – Historique
- Un microprocesseur simple
- **Programmation d'un microprocesseur**
- Système complet
- Les microprocesseurs actuels
- Exploitation de la performance des microprocesseurs

Programmation

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

00            AND R0, R0, #0      ; i ← 0
01            ADD R2, R0, #-12
02            ADD R2, R2, #-13
03            ADD R2, R2, R2
04            ADD R2, R2, R2      ; R2 ← -100
05 LOOP       LDR R1, R0, #30     ; R1 ← M(30+i), @a=30
06            ADD R1, R1, #5
07            STR R1, R0, #30     ; R1+5 → M(30+i)
08            ADD R0, R0, #1      ; i ← i + 1
09            ADD R3, R0, R2
0A            BRn LOOP            ; si i < 100 goto LOOP
0B            HALT
```

Programmation Assembleur

- Assembleur: un niveau d'abstraction au-dessus du langage machine (e.g., 1000101001100111).

```
01 ;
02 ; Exemple – Multiplication par 6
03 ;
04     .ORIG  x3050
05     LD     R1, SIX
06     LD     R2, NUMBER
07     AND    R3, R3, #0    ; R3 ← 0
08 ;
09 ; Boucle interne
0A ;
0B  AGAIN     ADD    R3, R3, R2
0C            ADD    R1, R1, #-1
0D            BRp    AGAIN
0E ;
0F            HALT
10 ;
11 NUMBER     .BLKW  1
12 SIX        .FILL  x0006
13 ;
14            .END
```

Programmation Assembleur
Syntaxe

N°Ligne	Label	Opcode	Opérandes	Commentaire

- Syntaxe d'une instruction:
  - N°Ligne: ne *pas confondre* avec l'adresse en mémoire de l'instruction (16-bits, 2 octets).
  - Labels (≈ pointeur):
    - Adresse cible (`BRp AGAIN`)
    - Zone mémoire (`LD R1, SIX`)
  - Opcodes: voir définition du jeu d'instruction.
  - Opérandes:
    - Registre (R_i)
    - Zone mémoire: adresse ou label
- Un programme assembleur contient:
  - des instructions
  - et des **données** constantes
- Placer ces données est notamment le rôle des **directives** qui sont interprétées par l'assembleur.

Programmation Assembleur - Directives

Directive	Signification	Exemple
.ORIG ***adresse***	Adresse en mémoire de la première instruction du programme	`.ORIG x3050` L'adresse de `LD R1, SIX` sera 0x3050
.BLKW ***nb***	Les *nb* mots mémoire suivants ne sont pas utilisés (saut); réservation d'une zone mémoire.	`NUMBER .BLKW 1` L'adresse 0x3057 sera utilisée pour la variable NUMBER.
.FILL ***valeur***	Stocker le paramètre *valeur* dans les mots mémoire suivants	`SIX .FILL x0006` L'adresse 0x3058 contient 0x0006.

.STRINGZ ***chaîne de caractères***	Stocker *chaîne de caractères* dans les mots mémoires suivants en codage ASCII et rajouter un 0 en fin de chaîne.	On rajoute
		`ESSAI .STRINGZ «Hello»`
		en fin de programme:
		`0x3059: x0048`
		`0x305A: x0065`
		`0x305B: x006C`
		`0x305C: x006C`
		`0x305D: x006F`
		`0x305E: x0000`

Assemblage

- 2 passes:
  - 1ère passe: déterminer l'adresse des symboles et des instructions (interprétation des directives).
    - Construction d'une **table de symboles**.

```
Symbol Name       Address
----------------  ------------
AGAIN             3053
NUMBER            3057
SIX               3058
```

  - 2ème passe: traduction des instructions assembleur en code machine (binaire).
    - On substitue aux symboles leur adresse, donnée par la table des symboles.

Assemblage

- Un programme:
  - Est souvent scindé en plusieurs fichiers
  - Fait souvent appel à des routines du système d'exploitation (`read`/`write`).

  ⇒ Phase de lien (*link*) à la compilation, après assemblage.
- Il faut donc:
  - Concaténer les différentes parties du code.
  - Pouvoir référencer des symboles non accessibles lors de l'assemblage.

  ⇒ Les adresses cibles de branchement ne sont fixées qu'après la phase de lien.

  ⇒ On doit disposer d'une directive **`.EXTERN`** ***`symbole`*** pour indiquer qu'un symbole sera déterminé lors de la phase de lien.

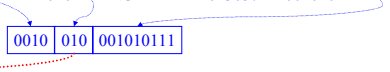
  ⇒ Le code binaire complet n'est donc définitif qu'après la phase de lien.

Assemblage

Adresse	Instruction en binaire	Instruction en hexa	label	Instruction assembleur	
x3050	0010001001011000	x2258		LD	R1, SIX
x3051	0010010001010111	x2457		LD	R2, NUMBER
x3052	0101011011100000	x56E0		AND	R3, R3, x0000
x3053	0001011011000010	x16C2	AGAIN	ADD	R3, R3, R2
x3054	0001001001111111	x127F		ADD	R1, R1, xFFFF
x3055	0000001001010011	x0253		BRP	AGAIN
x3056	1111000000100101	xF025		TRAP	HALT
x3057	0000000000000000	x0000	NUMBER	NOP	
x3058	0000000000000110	x0006	SIX	NOP	

- Le programme de multiplication après assemblage.

LD → 0010 R2 → 010 NUMBER → 0x3057 → 001010111



Exemple sur l'Alpha 21264

```
        .set noat
        .set noreorder
        .text
        .arch   generic
        .align 4
        .file 1 "test01.c"
        .loc 1 2
        .globl  main
        .ent    main
main:
        lda     $sp, -400($sp)
        .frame  $sp, 400, $26
        .prologue 0
        clr     $1
        mov     $sp, $2
        unop
L$2:
        ldl     $4, ($2)
        addl    $1, 1, $1
        cmplt   $1, 100, $5
        lda     $2, 4($2)
        addl    $4, 5, $4
        stl     $4, -4($2)
        bne     $5, L$2
        clr     $0
        lda     $sp, 400($sp)
        ret     ($26)
```

Code Assembleur

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

Code Source

Exemple sur l'Alpha 21264

```
main:
 0x120001100: lda     sp, -400(sp)      ; déplacer pointeur pile
 0x120001104: bis     r31, r31, r1      ; r1 ← 0
 0x120001108: bis     r31, sp, r2       ; r2 ← pointeur pile
                                        (ici, début tableau a)
 0x12000110c: ldq_u   r31, 0(sp)        ; NOP
 0x120001110: ldl     r4, 0(r2)         ; r4 ← @(r2)
 0x120001114: addl    r1, 0x1, r1       ; i ← i + 1
 0x120001118: cmplt   r1, 0x64, r5      ; i < 100 ?
 0x12000111c: lda     r2, 4(r2)         ; r2 ← r2 + 4
 0x120001120: addl    r4, 0x5, r4       ; a[i] + 5
 0x120001124: stl     r4, -4(r2)        ; a[i] ← a[i] + 5
 0x120001128: bne     r5, 0x120001110   ; si i < 100 boucler
 0x12000112c: bis     r31, r31, r0      ; NOP
 0x120001130: lda     sp, 400(sp)       ; rétablir pointeur pile
 0x120001134: ret     r31, (r26), 1     ; PC ← r26
```

Code Machine

Structures de Données

- Variables entières: 1 mot.
- Variables flottantes: 2 mots consécutifs pour LC-2.
- Tableau: *n* mots consécutifs.
- Chaînes de caractères: normalement *n* octets consécutifs (1 octet = 1 caractère); *n* mots consécutifs pour LC-2.

```
int    var1          = 21;
double var2          = 3.5;
int    var4[3]       = {1, 2, 3};
char   var5[6]       = «essai»;
```

0x4000	x0015	var1
0x4001	x0000	var2 (poids faible)
0x4002	x4060	var2 (poids fort)
0x4003	x0001	var4[0]
0x4004	x0002	var4[1]
0x4005	x0003	var4[2]
0x4006	x0065	e
0x4007	x0073	s
0x4008	x0073	s
0x4009	x0061	a
0x400A	x0069	i
0x400B	x0000	\0

Structures de Données

- Manipulation de données complexes (e.g., tableaux, chaînes de caractères...).

```
main() {
int tab[3];
       ...
       tab [2] += 1;
       mult(tab);
       ...
}

void mult(int *tab) {
       a = tab[0]
              + tab[1]
              + tab[2];
       return a;
}
```

```
MAIN
                      ; adresse de
                      tab dans R6
...
LDR R0, R6, #2 ; R0←tab[2]
ADD R0, R0, #1
STR R0, R6, #2 ; R0←tab[2]+1
ADD R0, R6, #0 ; Calcul
                      adresse de tab
                      et
                      transmission
                      de l'adresse
                      tab à MULT
...
JSR MULT
...

MULT
...
                      ; adresse de
                      tab dans R0
LDR R1, R0, #0 ; R1←tab[0]
LDR R2, R0, #1 ; R2←tab[1]
ADD R1, R2, R1
LDR R2, R0, #2 ; R1←tab[2]
ADD R1, R2, R1
...
```

Structures de Contrôle

- Le test: `if (...) {...} else {...}`

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

```
                    ...
                    LDR R0, R6, #3   ; R0←x
                    ADD R1, R0, #-5  ; x-5
                    BRnz ELSE        ; x-5 ≤ 0 ?
                    ADD R0, R0, #1
                    BR FIN
             ELSE
                    ADD R0, R0, #-1
                    ...
             FIN
                    ...
```

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

```
                    ...
                    LDR R0, R6, #3   ; R0←x
                    ADD R1, R0, #-5  ; x-5
                    BRnp ELSE        ; x-5 ≠ 0 ?
                    ADD R0, R0, #1
                    BR FIN
             ELSE
                    ADD R0, R0, #-1
                    ...
             FIN
                    ...
```

Structures de Contrôle

- La boucle while: `while (...) {...}`

```
while (x > 5) {
       x += 1;
}
```

```
                    ...
             BOUCLE LDR R0, R6, #3   ; R0←x
                    ADD R1, R0, #-5  ; x-5
                    BRnz FIN         ; x-5 ≤ 0 ?
                    ADD R0, R0, #1
                    STR R0, R6, #3
                    BR BOUCLE
             FIN
                    ...
```

- La boucle for: `for (init; test; update) {...}`

```
for (i=0; i < 5; i++) {
       x += 1;
}
```

```
                    AND R0, R0, #0   ; i=0
                    STR R0, R6, #4   ; i→R0
             BOUCLE LDR R0, R6, #4   ; R0←i
                    ADD R1, R0, #-5  ; i-5
                    BRnp FIN         ; i-5 ≥ 0 ?
                    LDR R0, R6, #3   ; R0←x
                    ADD R0, R0, #1
                    STR R0, R6, #3   ; R0→x
                    LDR R0, R6, #4   ; R0←i
                    ADD R0, R0, #1
                    STR R0, R6, #4   ; R0→i
                    BR BOUCLE
             FIN    ...
```

Appel de Fonctions

```
        .ORIG   x3000
        LD R0, A
        LD R1, B
        JSR MULT
        ADD R5, R2, #0
        LD R0, C
        LD R1, D
        JSR MULT
        ADD R5, R5, R2
        HALT
; --- Donnees
A       .FILL   x0002
B       .FILL   x0002
C       .FILL   x0002
D       .FILL   x0003
; --- Procedure de multiplication
MULT    AND     R2, R2, #0
LOOP    ADD     R2, R2, R1
        ADD     R0, R0, #-1
        BRp     LOOP
        RET
        .END
```

- Passage de paramètres par les registres:
  - Rapide
  - Nombre de paramètres limité
- Retour des valeurs:
  - Un seul mot (return)
  - Convention: par exemple, résultat retourné dans R2.
- Rappel:
  - Adresse de retour dans R7.

Appel de Fonctions

```
        .ORIG   x3000
        LD R0, A
        LD R1, B
        JSR MULT
        ADD R5, R2, #0
        LD R0, C
        LD R1, D
        ST R5, BAK_R5
        JSR MULT
        LD R5, BAK_R5
        ADD R5, R5, R2
        HALT
; --- Sauvegarde
BAK_R5  .BLKW   1
; --- Donnees
A       .FILL   x0002
B       .FILL   x0002
C       .FILL   x0002
D       .FILL   x0003
; --- Procedure de multiplication
MULT    AND     R2, R2, #0
LOOP    ADD     R2, R2, R1
        ADD     R0, R0, #-1
        BRp     LOOP
        RET
```

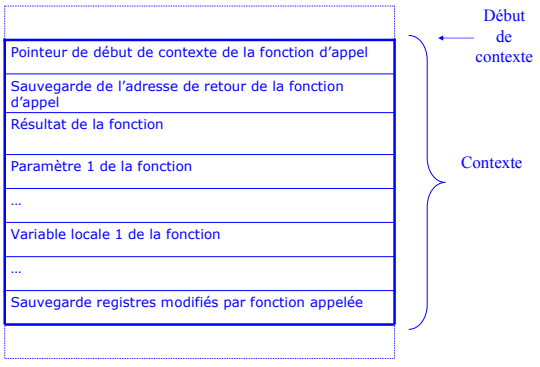
- Registres utilisés dans fonction souvent inconnus (routines externes…):
  - Sauvegarde des registres avant appel de fonctions.
- Ou:
  - Sauvegarde des registres modifiés dans la fonction appelée.
- Passage de paramètres par mémoire.

Appel de Fonctions
La Pile

- Pour une fonction, il est nécessaire:
  - de sauver les registres modifiés par la fonction appelée pour les restaurer en fin d'appel.
  - d'avoir une zone pour stocker les variables locales (durée de vie = exécution de la fonction).
  - de passer des paramètres à la fonction par la mémoire.
  - de renvoyer le résultat de la fonction.

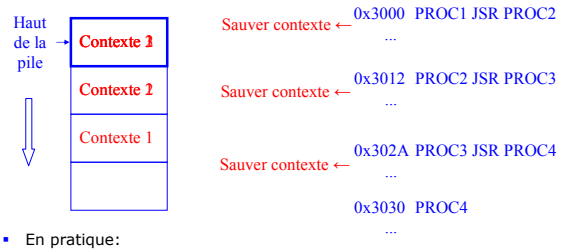
  ⇒ **contexte** de la fonction.
- 1ère solution: zones mémoire statiques (`BAK_R2`) à insérer dans chaque routine et utilisées seulement en cas d'appel → inefficace.
- 2ème solution: zone mémoire dynamique allouée à l'exécution, libérée en fin d'exécution: la **pile**.
- Remarque:
  - On peut souvent utiliser le passage de paramètres par registre afin de réduire l'utilisation de la mémoire/pile.
  - La pile est nécessaire pour le passage d'un grand nombre de paramètres.

Le contexte dans la Pile



La Pile - Fonctionnement

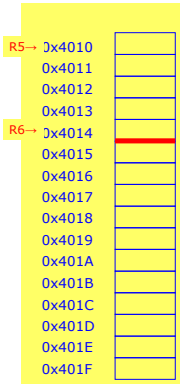
- LIFO (*Last In First Out*)
- Seul le premier élément est visible et accessible.



- En pratique:
  - Zone mémoire de taille importante
  - Les données sauvées ne sont pas déplacées en mémoire
  - Un pointeur indique le haut de la pile (en général, un registre)
  - Un pointeur indique le début du contexte courant (pas toujours nécessaire)
- On choisit un pointeur de pile: `R6`, un pointeur de contexte: `R5`

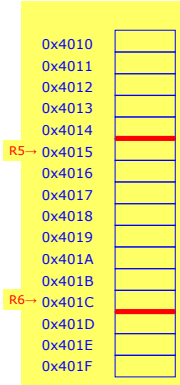
La Pile - Implémentation

```
CALLER ...
       ADD R6, R6, #1 ; réservation de 1 mot sur
                        la pile
       STR R5, R6, #0 ; sauvegarde du registre de
                        contexte courant
       ADD R6, R6, #7 ; réservation de 7 mots sur
                        la pile
       ADD R5, R6, #-7; nouveau contexte pour la
                        fonction Mult
       LD R0, A       ;
       STR R0, R5, #3 ; passage du paramètre A
       LD R0, B       ;
       STR R0, R5, #4 ; passage du paramètre B
       JSR Mult       ; appel de la fonction Mult
       LDR R0, R5, #2 ; lecture de A*B
       LDR R5, R5, #0 ; restauration du registre
                        de contexte courant
       ADD R6, R6, #-8; libération de 8 mots sur
                        la pile
       ...
```



La Pile - Implémentation

```
MULT   ...
       STR R7, R5, #1 ; sauvegarde de
                        l'adresse de retour
       STR R0, R5, #5 ; sauvegarde de R0
       STR R1, R5, #6 ; sauvegarde de R1
       STR R2, R5, #7 ; sauvegarde de R2
       LDR R0, R5, #3 ; lecture du premier
                        paramètre
       LDR R1, R5, #4 ; lecture du deuxième
                        paramètre
       AND R2, R2, #0
Loop   ADD R2, R2, R1
       ADD R0, R0, #-1
       BRp Loop
       STR R0, R5, #2 ; valeur de retour
       LDR R0, R5, #5 ; restauration de R0
       LDR R1, R5, #6 ; restauration de R1
       LDR R2, R5, #7 ; restauration de R2
       LDR R7, R5, #1 ; restauration adresse
                        retour
       RET
       ...
```



Appels Récursifs de Fonctions

```
        Somme (int n) {
                if (n > 1) return (n + Somme(n-1));
                else return 1;
        }
SOMME
        STR R7, R5, #1    ; sauvegarde de l'adresse de retour
        STR R0, R5, #4    ; sauvegarde de R0
        STR R1, R5, #5    ; sauvegarde de R1
        LDR R0, R5, #3    ; lecture de n sur la pile
        ADD R1, R0, #-1   ; n > 1 ?
        BRnz FIN
        ADD R6, R6, #1 ; réservation de 1 mot
        STR R5, R6, #0 ; sauvegarde du registre de
                       contexte courant
        ADD R6, R6, #5 ; réservation de 6 mots
        ADD R5, R6, #-5; nouveau contexte
        STR R1, R5, #3    ; Transmission de n-1 par la pile
        JSR SOMME
        LDR R1, R5, #2    ; Récupération de la valeur de retour
        ADD R0, R1, R0    ; n + Somme(n-1)
        LDR R5, R5, #0    ; restauration du registre
                          de contexte courant
        ADD R6, R6, #-6   ; libération de 6 mots sur la pile
FIN     STR R0, R5, #2    ; sauvegarde valeur de retour
        LDR R0, R5, #4    ; restauration de R0
        LDR R1, R5, #5    ; restauration de R1
        LDR R7, R5, #1    ; restauration adresse retour
        RET
```