

# Systèmes d'exploitations

## Cours 2: De l'architecture matérielle au programme

Matthieu Lemerre  
CEA LIST

Année 2020-2021

But de ce cours:

- ① Rappeler le principe de fonctionnement d'un ordinateur
  - En particulier: couple processeur/mémoire
- ② Expliquer pourquoi et comment est organisé le programme pour pouvoir écrire des programmes complexes
  - En particulier: comment est compilé le C?

# Quelques applications de ce cours

- Compréhension et écriture de programmes systèmes: systèmes embarqués, garbage collectors, morceaux de systèmes d'exploitations (pilotes), compilateurs
- Assembleur: Reverse engineering et analyses de sécurité
- Connaissances nécessaires pour savoir bien programmer en C (et C++)

## 1 Cours 2: Organisation de la mémoire

- Architecture matérielle
- Organisation de la mémoire (et correspondance en C)
- Organisation du code
- Organisation des données

Bits

## Bits

Binary digits, la plus petite unité d'information. Peut valoir 0 ou 1.

# Définitions: bit, octets et mots

## Bits

Binary digits, la plus petite unité d'information. Peut valoir 0 ou 1.

## Octet (byte)

# Définitions: bit, octets et mots

## Bits

Binary digits, la plus petite unité d'information. Peut valoir 0 ou 1.

## Octet (byte)

La plus petite unité de mémoire adressable. Généralement 8 bits.



# Définitions: bit, octets et mots

## Bits

Binary digits, la plus petite unité d'information. Peut valoir 0 ou 1.

## Octet (byte)

La plus petite unité de mémoire adressable. Généralement 8 bits.

## Mot (word)

# Définitions: bit, octets et mots

## Bits

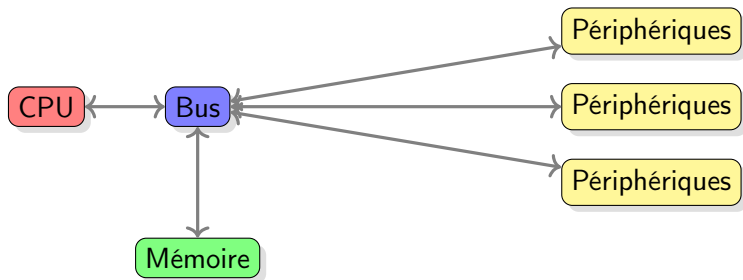
Binary digits, la plus petite unité d'information. Peut valoir 0 ou 1.

## Octet (byte)

La plus petite unité de mémoire adressable. Généralement 8 bits.

## Mot (word)

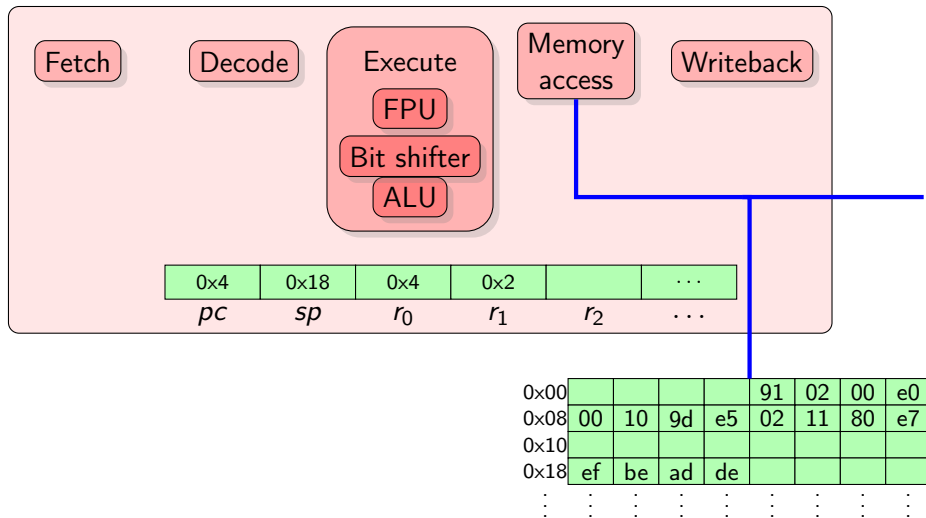
L'unité de mémoire manipulée naturellement par un processeur, composée de plusieurs octets. Généralement 8,16,32,64,ou 128 bits.



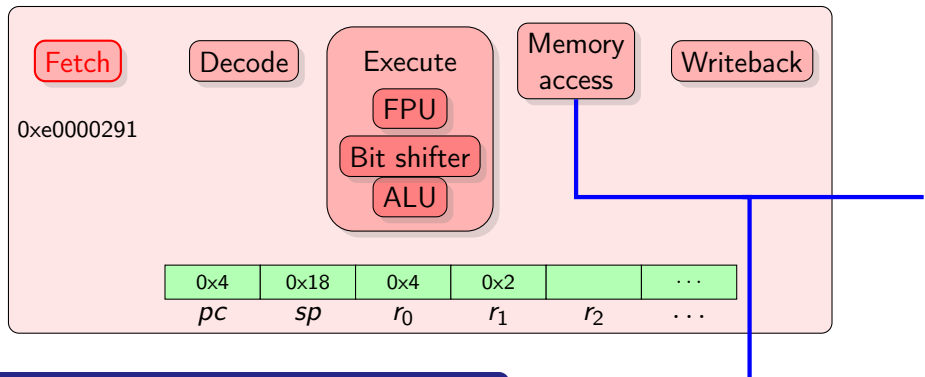
## Architecture Von Neumann

La *même* mémoire est utilisée pour stocker les *instructions* et les *données* des programmes.

# Fonctionnement du processeur



# Fonctionnement du processeur

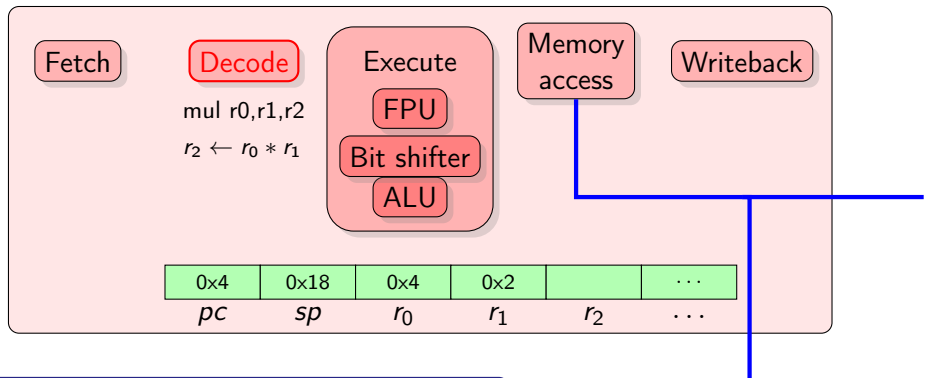


## Fetch (récupération)

Récupération du mot d'instruction pointé par le pointeur d'instruction (pc, program counter).

0x00					91	02	00	e0
0x08	00	10	9d	e5	02	11	80	e7
0x10								
0x18	ef	be	ad	de				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

# Fonctionnement du processeur

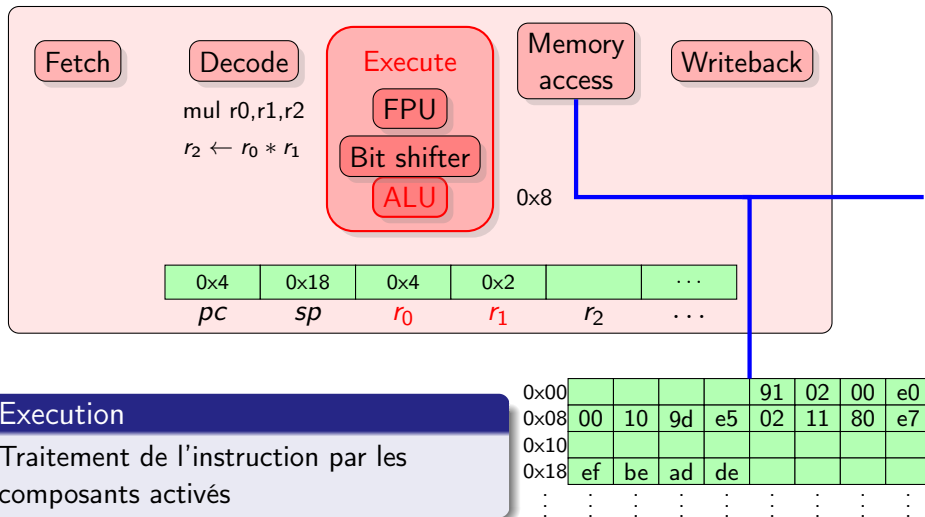


## Décodage

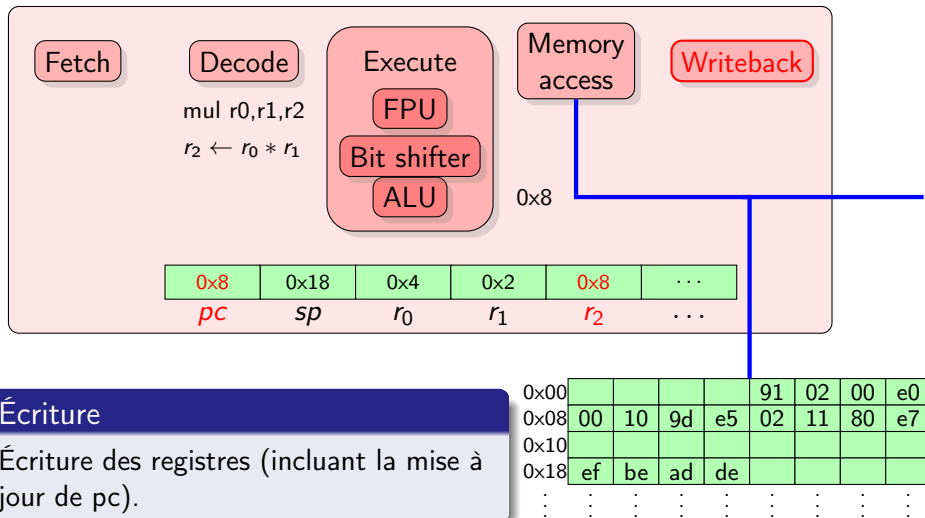
Activation des composants et chemins correspondant au traitement de l'instruction

0x00					91	02	00	e0
0x08	00	10	9d	e5	02	11	80	e7
0x10								
0x18	ef	be	ad	de				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

# Fonctionnement du processeur

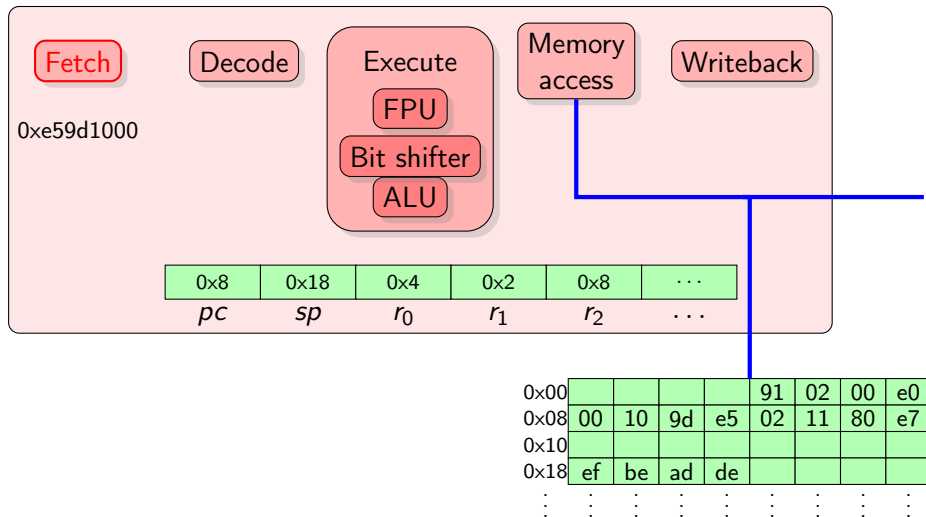


# Fonctionnement du processeur

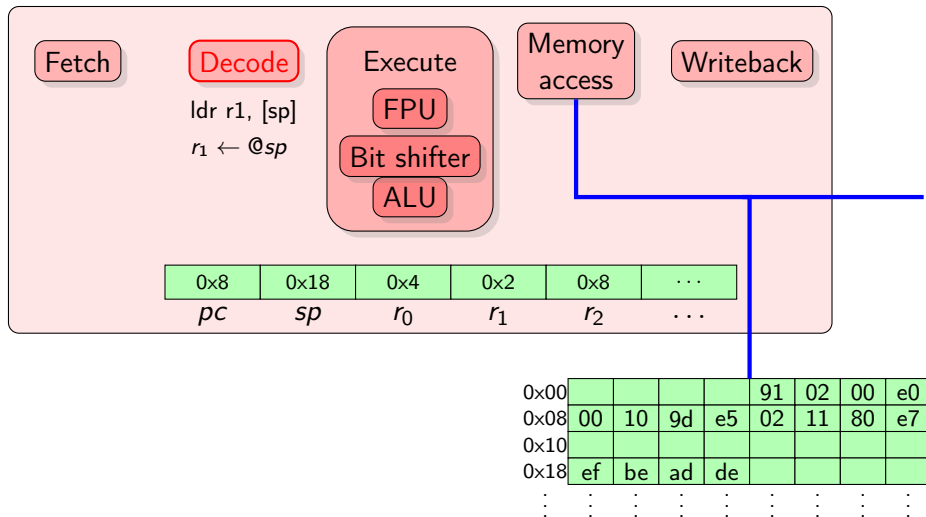




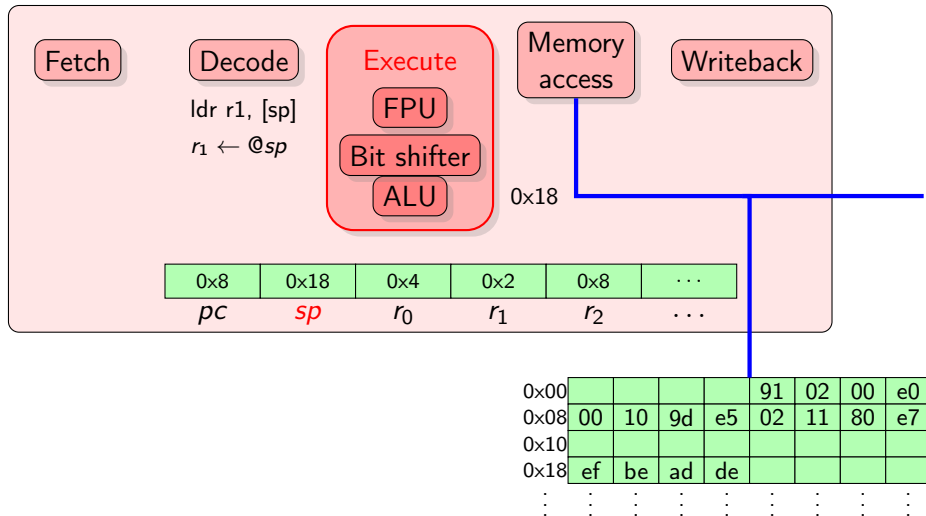
# Fonctionnement du processeur



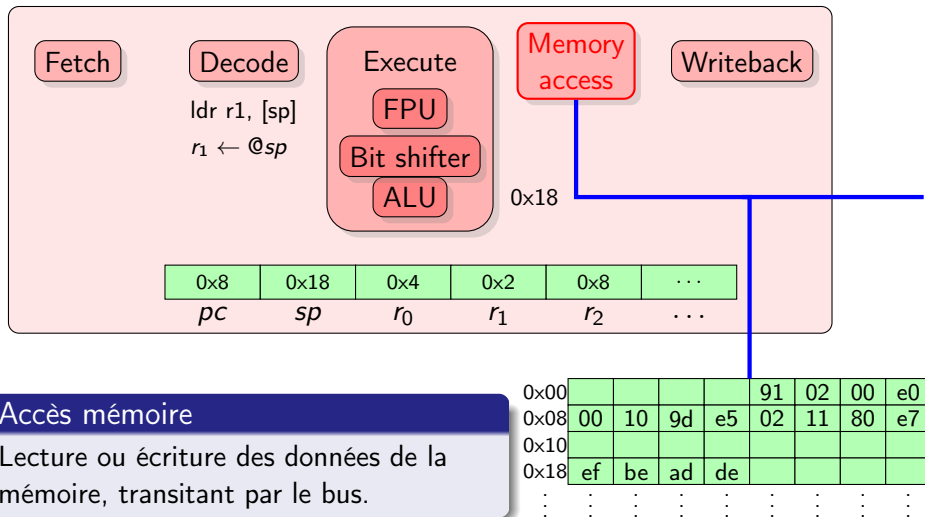
# Fonctionnement du processeur



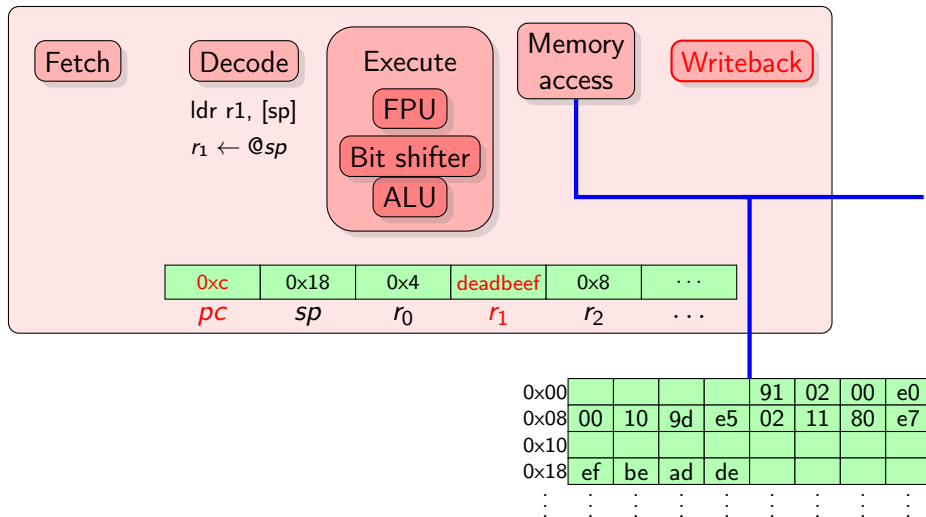
# Fonctionnement du processeur



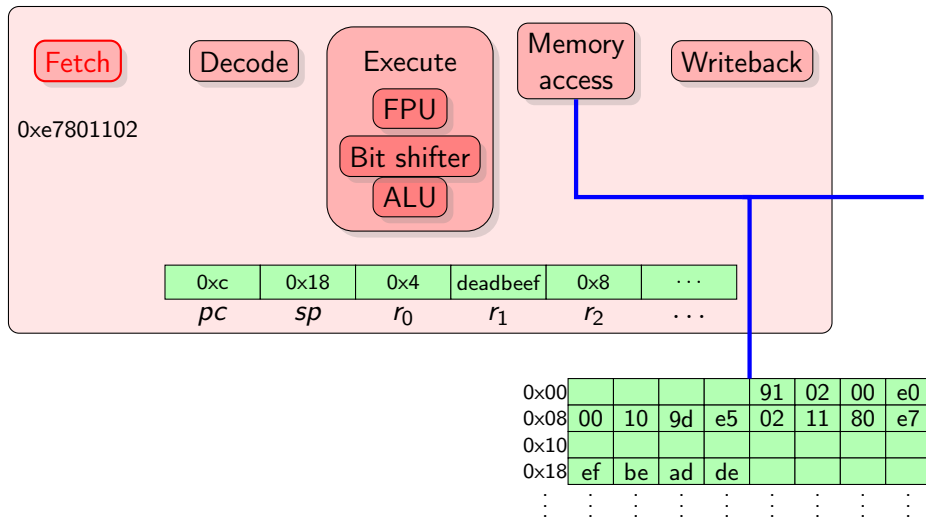
# Fonctionnement du processeur



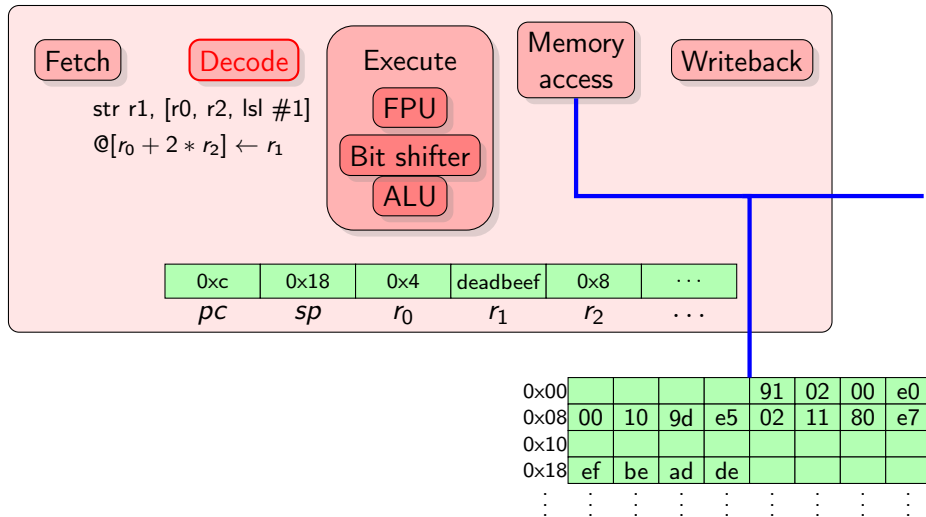
# Fonctionnement du processeur



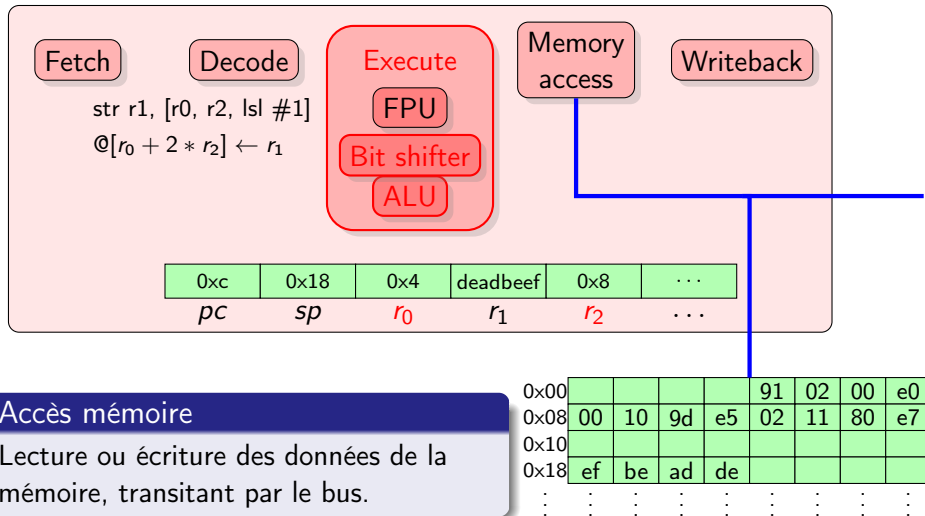
# Fonctionnement du processeur



# Fonctionnement du processeur

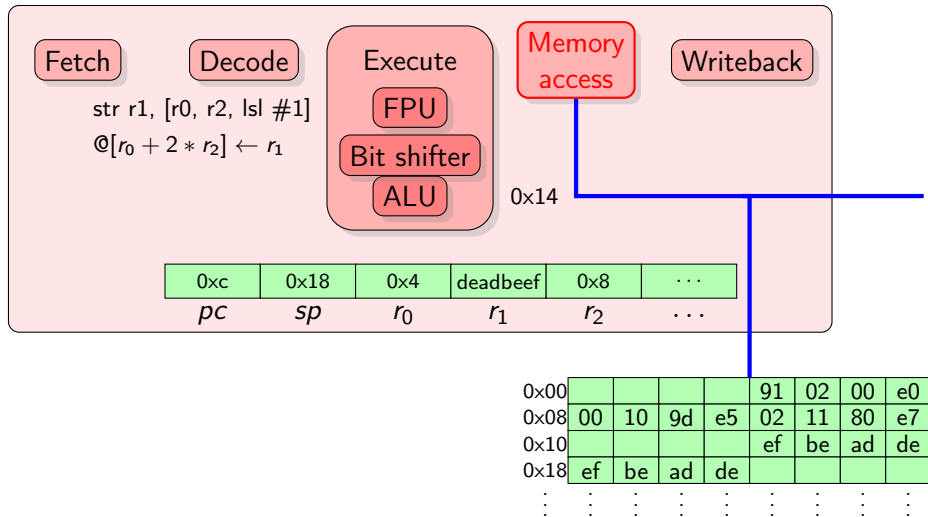


# Fonctionnement du processeur

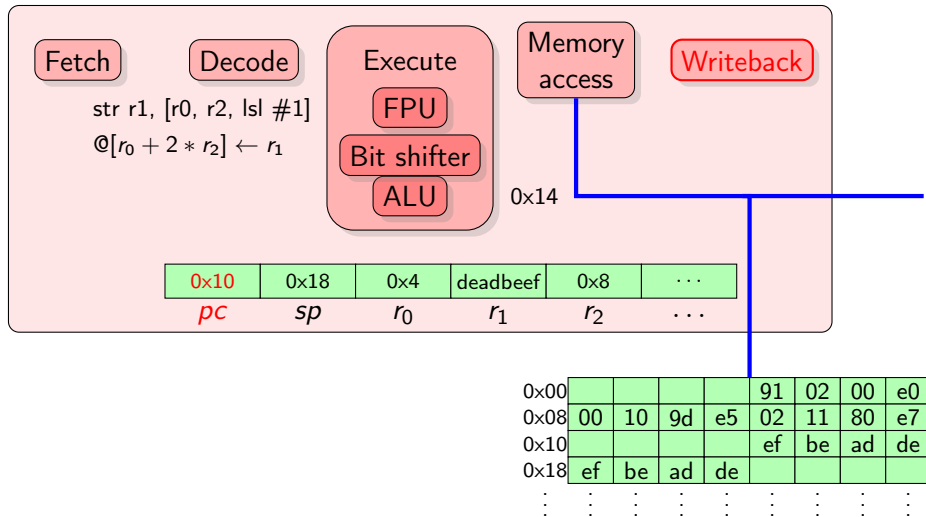




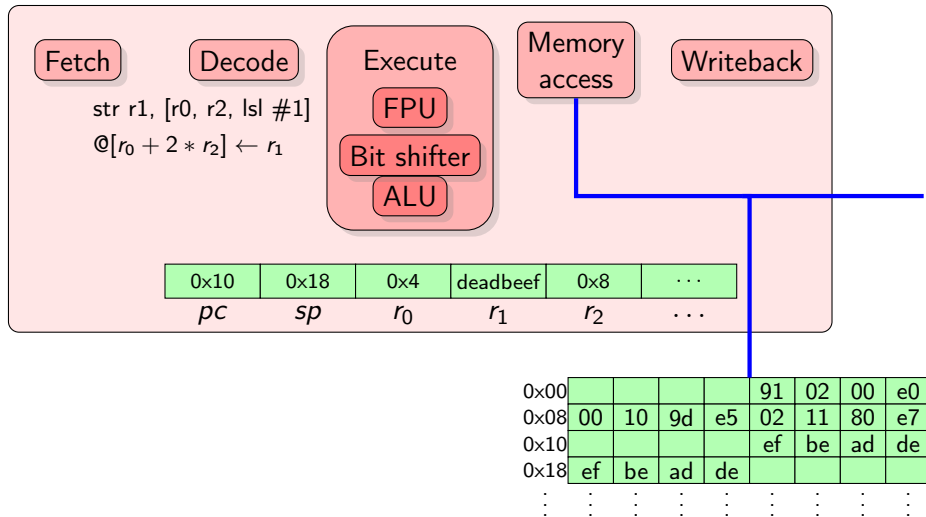
# Fonctionnement du processeur



# Fonctionnement du processeur



# Fonctionnement du processeur



# Catégories d'instructions

- **Manipulation de données :**
  - `mov`:  $r_i \leftarrow \text{const}$ ,  $r_i \leftarrow r_j$
  - `load` et `store`
- **Instructions arithmétiques et logiques :**
  - $r_i \leftarrow r_j \oplus r_k$  pour  $\oplus \in \{ + \ - \ * \ / \ \% \ ^ \ \& \ | \ \ll \ \gg \}$
- **Comparaisons :**
  - `cmp`  $r_i \ r_j$ :  $z \leftarrow r_i == r_j$ ;  $s \leftarrow r_i \leq_u r_j$
- **Sauts:**
  - **Sauts inconditionnels** :  $pc \leftarrow pc + 8$
  - **Saut conditionnel**: `if(z)`  $pc \leftarrow pc + 8$
  - **Saut calculé**:  $pc \leftarrow r_1$
- **Jeux d'instructions étendus**
  - Instructions flottantes
  - Instructions vectorisées
  - Instructions cryptographiques
  - Instructions atomiques
- **Instructions systèmes**

- Un processeur:
  - Lit les instructions depuis la mémoire, en suivant le pointeur d'instruction
  - Fait des lectures et écritures entre la mémoire et ses registres
  - Fait des opérations arithmétiques et logiques en combinant ses registres

- Un processeur:
  - Lit les instructions depuis la mémoire, en suivant le pointeur d'instruction
  - Fait des lectures et écritures entre la mémoire et ses registres
  - Fait des opérations arithmétiques et logiques en combinant ses registres

## Note: Interpréteur de bytecode

Les instructions sont souvent similaires pour les machines virtuelles exécutant du bytecode (bytecode java, .NET, OCaml), sauf qu'elles n'utilisent généralement pas de registres (machine à pile).

- Un processeur:
  - Lit les instructions depuis la mémoire, en suivant le pointeur d'instruction
  - Fait des lectures et écritures entre la mémoire et ses registres
  - Fait des opérations arithmétiques et logiques en combinant ses registres

## Note: Interpréteur de bytecode

Les instructions sont souvent similaires pour les machines virtuelles exécutant du bytecode (bytecode java, .NET, OCaml), sauf qu'elles n'utilisent généralement pas de registres (machine à pile).

Comment organiser le programme pour faciliter son écriture?

## 1 Cours 2: Organisation de la mémoire

- Architecture matérielle
- Organisation de la mémoire (et correspondance en C)
- Organisation du code
- Organisation des données



- Comprendre pourquoi et comment est organisé le programme pour pouvoir écrire des programmes complexes
  - En particulier: comment est compilé le C?

---

<sup>1</sup>Attention aux pièges

- Comprendre pourquoi et comment est organisé le programme pour pouvoir écrire des programmes complexes
  - En particulier: comment est compilé le C?
- Pourquoi C?
  - Correspondance directe<sup>1</sup> entre ses concepts et de l'assembleur manuel
  - "C comme un assembleur de haut niveau"

---

<sup>1</sup>Attention aux pièges

- Comprendre pourquoi et comment est organisé le programme pour pouvoir écrire des programmes complexes
  - En particulier: comment est compilé le C?
- Pourquoi C?
  - Correspondance directe<sup>1</sup> entre ses concepts et de l'assembleur manuel
  - "C comme un assembleur de haut niveau"
- Intérêts:
  - Important pour savoir bien programmer en C
    - Compréhension du langage
    - Modèle de coût d'un programme
    - Programmation système
  - Il faut comprendre comment marche un programme avant de comprendre comment en faire fonctionner plusieurs.

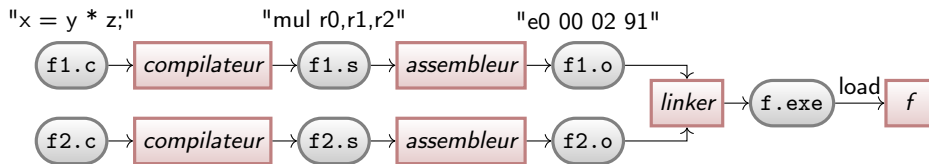
---

<sup>1</sup>Attention aux pièges

# D'un programme à son exécution: compilation et interprétation

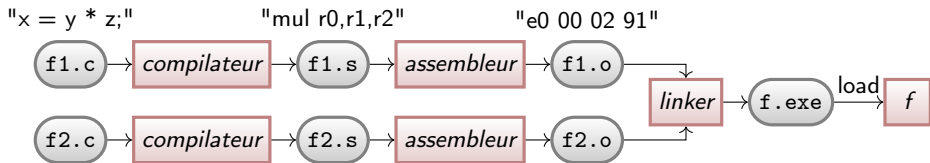
# D'un programme à son exécution: compilation et interprétation

- Le C est un langage **compilé**: il est transformé en instructions machines avant d'être chargé par l'OS pour être exécuté

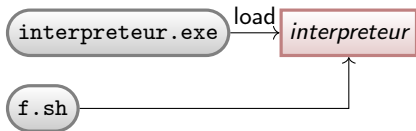


# D'un programme à son exécution: compilation et interprétation

- Le C est un langage **compilé**: il est transformé en instructions machines avant d'être chargé par l'OS pour être exécuté

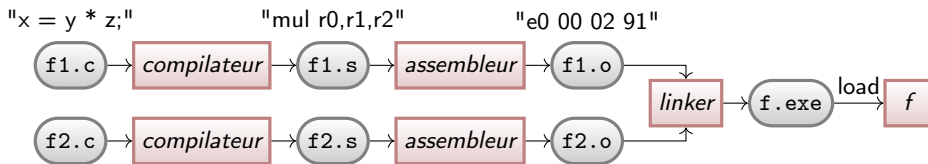


- Certains langages sont **interprétés** (bash): l'interpréteur exécute les instructions sans les traduire.

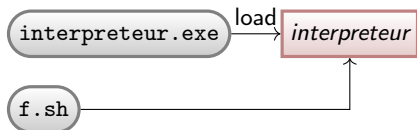


# D'un programme à son exécution: compilation et interprétation

- Le C est un langage **compilé**: il est transformé en instructions machines avant d'être chargé par l'OS pour être exécuté

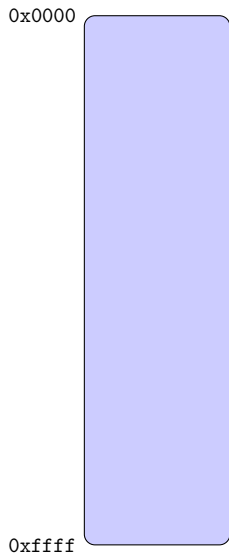


- Certains langages sont **interprétés** (bash): l'interpréteur exécute les instructions sans les traduire.



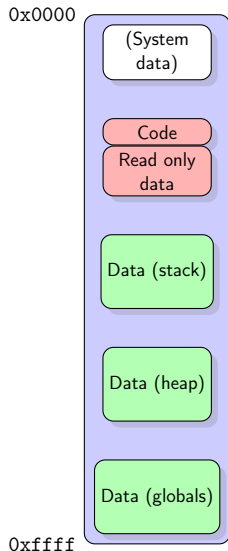
- Il y a aussi des implantations de langage mixtes: compilation vers bytecode, just-in-time compilation...

# Organisation générale de la mémoire d'un processus





# Organisation générale de la mémoire d'un processus



- Code: read-only, contient les instructions
- Données read-only: contient constantes, chaîne de caractères. . .
- Données read-write: séparées en plusieurs zones
  - Piles (stack): variables locales d'un thread (allocation sur *pile*)
  - Globaux: variables globales (allocation *statique*)
  - Tas (heap): malloc (allocation *dynamique*)

# Exemple réel

```
> readelf -a file.exe
```

ELF Header:

```
...
Type:                EXEC (Executable file)
Machine:             Intel 80386
Version:             0x1
Entry point address: 0x8048730
```

```
...
```

```
...
```

Program Headers:

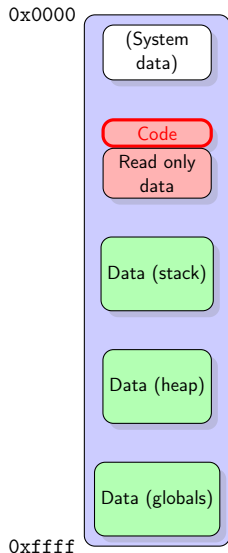
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x08048000	0x08048000	0x8f0f8	0x8f0f8	R E	0x1000
LOAD	0x08f6dc	0x080d86dc	0x080d86dc	0x02c40	0x1e03914	RW	0x1000

```
...
```

## 1 Cours 2: Organisation de la mémoire

- Architecture matérielle
- Organisation de la mémoire (et correspondance en C)
- Organisation du code
- Organisation des données

# Organisation du code



# Traduction des sauts (goto,break,continue)

## Exemple (C)

```
goto next;  
...  
next:  
...
```

## Exemple (ARM assembly language)

```
current:  
    b next  
...  
...  
...  
next:  
...
```

# Traduction des sauts (goto,break,continue)

## Exemple (C)

```
goto next;  
...  
next:  
...
```

## Exemple (ARM assembly language)

```
current:  
    b next    ; pc := pc + 16  
...  
...  
...  
next:  
...
```

C'est le rôle du **linker** (éditeur de lien) de calculer l'offset correspondant à une cible de saut.

# Traduction des instructions conditionnelles

## Exemple (C)

```
if(x != 3){  
  ...  
}  
else {  
  ...  
}
```

## Exemple (ARM assembly language)

```
    cmp r0, 3    ; update flags  
    beq ???     ; if(z) pc := pc + ...  
then:  
    ...  
    ???  
else:  
    ...  
    ???  
end:  
    ...
```

# Traduction des instructions conditionnelles

## Exemple (C)

```
if(x != 3){  
  ...  
}  
else {  
  ...  
}
```

## Exemple (ARM assembly language)

```
    cmp x, 3    ; update flags  
    beq else    ; if(z) pc := pc + ...  
then:  
    ...  
    ???  
else:  
    ...  
    ???  
end:  
    ...
```



# Traduction des instructions conditionnelles

## Exemple (C)

```
if(x != 3){  
  ...  
}  
else {  
  ...  
}
```

## Exemple (ARM assembly language)

```
    cmp x, 3    ; update flags  
    beq else    ; if(z) pc := pc + ...  
then:  
    ...  
    b end      ; pc := pc + ...  
else:  
    ...  
    ???  
end:  
    ...
```

# Traduction des instructions conditionnelles

## Exemple (C)

```
if(x != 3){  
  ...  
}  
else {  
  ...  
}
```

## Exemple (ARM assembly language)

```
    cmp x, 3    ; update flags  
    beq else    ; if(z) pc := pc + ...  
then:  
    ...  
    b end      ; pc := pc + ...  
else:  
    ...  
    ...        ; no need to jump here  
end:  
    ...
```

# Traduction des boucles

## Exemple (C)

```
int x = 0;
while(x < 12){
    x++;
}
```

## Exemple (ARM assembly language)

```
    mov r0, 0
loop:
    cmp r0, 12    ; update flags
                  ; branch if result
                  ; of signed comparison
                  ; was >=
    bge end
    add r0,r0,1   ; r0 := r0 + 1
    b loop        ; pc := pc - 12
end:
```

# Traduction des appels de fonctions (1)

## Exemple (C)

```
void f(void) {}  
  
void g(void) {  
    f();  
    ...  
}
```

## Exemple (ARM assembly language)

```
f:  
    b after      ; pc <- &after  
  
g:  
    b f          ; pc <- &f  
after:  
    ...
```

Pourquoi est-ce que cette solution ne marche pas en général?

# Traduction des appels de fonctions (1)

## Exemple (C)

```
void f(void) {}

void g(void) {
    f();
    ...
}

void h(void) {
    f();
    ...
}
```

## Exemple (ARM assembly language)

```
f:
    b after      ; pc <- &after

g:
    b f          ; pc <- &f
after:
    ...

h:
    b f          ; pc <- &f
after2:
    ...
```

# Traduction des appels de fonctions (1)

## Exemple (C)

```
void f(void) {}

void g(void) {
    f();
    ...
}

void h(void) {
    f();
    ...
}
```

## Exemple (ARM assembly language)

```
f:
    bx lr          ; pc <- lr

g:
    mov lr,after ; lr <- &after
    b f          ; pc <- &f
after:
    ...

h:
    mov lr,after2; lr <- &after2
    b f          ; pc <- &f
after2:
    ...
```

# Traduction des appels de fonctions (1)

## Exemple (C)

```
void f(void) {}

void g(void) {
    f();
    ...
}

void h(void) {
    f();
    ...
}
```

## Exemple (ARM assembly language)

```
f:
    bx lr          ; pc <- lr

g:
    bl f           ; lr <- pc + 4; pc <- &f
after:
    ...

h:
    bl f           ; lr <- pc + 4; pc <- &f
after2:
    ...
```

# Traduction des appels de fonctions (2): appels imbriqués

## Exemple (C)

```
void f2(void) {}  
void f1(void) {f2();}  
void main(void){  
    f1();  
}
```

## Exemple (ARM assembly language)

```
f2:  
    bx lr    ; pc <- lr  
f1:  
  
    bl f2    ; lr <- pc + 4; pc <- &f1  
  
    bx lr    ; pc <- lr  
main:  
    bl f1    ; lr <- pc + 4; pc <- &f1
```

Que fait ce code?



# Traduction des appels de fonctions (2): appels imbriqués

## Exemple (C)

```
void f2(void) {}  
void f1(void) {f2();}  
void main(void){  
    f1();  
}
```

## Exemple (ARM assembly language)

```
f2:  
    bx lr    ; pc <- lr  
f1:  
    push lr  ; sauvegarde lr sur la pile  
    bl f2    ; lr <- pc + 4; pc <- &f2  
    pop lr   ; restaure lr  
    bx lr    ; pc <- lr  
main:  
    bl f1    ; lr <- pc + 4; pc <- &f1
```

- Besoin de sauvegarder l'adresse de retour
- Besoin mémoire dépend de la profondeur d'appel ( $\Rightarrow$  **pile**)
- Peut on optimiser le retour de f1?

# Traduction des appels de fonctions (2): appels imbriqués

## Exemple (C)

```
void f2(void) {}  
void f1(void) {f2();}  
void main(void){  
    f1();  
}
```

## Exemple (ARM assembly language)

```
f2:  
    bx lr    ; pc <- lr  
f1:  
    push lr  ; sauvegarde lr sur la pile  
    bl f2    ; lr <- pc + 4; pc <- &f2  
    pop lr   ; restaure lr  
    bx lr    ; pc <- lr  
main:  
    bl f1    ; lr <- pc + 4; pc <- &f1
```

- Besoin de sauvegarder l'adresse de retour
- Besoin mémoire dépend de la profondeur d'appel ( $\Rightarrow$  **pile**)
- Peut on optimiser le retour de f1?

# Traduction des appels de fonctions (2): appels imbriqués

## Exemple (C)

```
void f2(void) {}  
void f1(void) {f2();}  
void main(void){  
    f1();  
}
```

## Exemple (ARM assembly language)

```
f2:  
    bx lr    ; pc <- lr  
f1:  
    push lr  ; sauvegarde lr sur la pile  
    bl f2    ; lr <- pc + 4; pc <- &f2  
    pop pc   ; pc <- ancien lr
```

```
main:  
    bl f1    ; lr <- pc + 4; pc <- &f1
```

- Schémas de traduction classique
  - Fonctions "feuilles" et "non-feuille".

# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:  
int f(int a, int b)  
{ return a - 2 * b }  
-----
```

```
main.c:  
...  
int x = 33;  
int y = x + f(13,1)  
...
```

## Exemple (ARM assembly language)

```
f:  mul r?,r?,2   ; r? <- r? * 2  
    sub r?,r?,r? ; r? <- r? - r?  
    bx lr
```

```
main: ...  
    mov r?, 33    ; r? <- 33  
    mov r?, 13    ; r? <- 13  
    mov r?, 1     ; r? <- 1  
    bl f          ; appelle f  
    add r?,r?,r?  ; r? <- r? + r?  
    ...
```

## Convention d'appel (ABI): défini

# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:  
int f(int a, int b)  
{ return a - 2 * b }  
-----
```

```
main.c:  
...  
int x = 33;  
int y = x + f(13,1)  
...
```

## Exemple (ARM assembly language)

```
f:  mul r?,r1,2   ; r? <- r1 * 2  
    sub r?,r0,r? ; r? <- r0 - r?  
    bx lr
```

```
main: ...  
    mov r?, 33    ; r? <- 33  
    mov r0, 13    ; r0 <- 13  
    mov r1, 1     ; r1 <- 1  
    bl f          ; appelle f  
    add r?,r?,r?  ; r? <- r? + r?  
    ...
```

## Convention d'appel (ABI): défini

- Comment sont passés les arguments (ARM: les premiers dans r0...r3)

# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:  
int f(int a, int b)  
{ return a - 2 * b }  
-----
```

```
main.c:  
...  
int x = 33;  
int y = x + f(13,1)  
...
```

## Exemple (ARM assembly language)

```
f:  mul r?,r1,2   ; r? <- r1 * 2  
    sub r0,r0,r? ; r0 <- r0 - r?  
    bx lr
```

```
main: ...  
    mov r?, 33    ; r? <- 33  
    mov r0, 13    ; r0 <- 13  
    mov r1, 1     ; r1 <- 1  
    bl f          ; appelle f  
    add r?,r?,r0  ; r? <- r? + r0  
    ...
```

## Convention d'appel (ABI): défini

- Comment sont passés les arguments (ARM: les premiers dans r0...r3)
- Comment sont renvoyés les résultats (ARM: dans r0)

# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:
int f(int a, int b)
{ return a - 2 * b }
```

-----

```
main.c:
```

```
...
int x = 33;
int y = x + f(13,1)
...
```

## Exemple (ARM assembly language)

```
f:  mul r2,r1,2   ; r2 <- r1 * 2
    sub r0,r0,r2  ; r0 <- r0 - r2
    bx lr
```

```
main: ...
```

```
    mov r2, 33    ; r2 <- 33
    mov r0, 13    ; r0 <- 13
    mov r1, 1     ; r1 <- 1
    bl f          ; appelle f
    add r?,r2,r0  ; r? <- r2 + r0
    ...
```

## Convention d'appel (ABI): défini

- Comment sont passés les arguments (ARM: les premiers dans r0...r3)
- Comment sont renvoyés les résultats (ARM: dans r0)
- Erreur

# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:  
int f(int a, int b)  
{ return a - 2 * b }  
-----
```

```
main.c:  
...  
int x = 33;  
int y = x + f(13,1)  
...
```

## Exemple (ARM assembly language)

```
f:  mul r2,r1,2   ; r2 <- r1 * 2  
    sub r0,r0,r2 ; r0 <- r0 - r2  
    bx lr
```

```
main: ...  
    mov r2, 33    ; r2 <- 33  
    mov r0, 13    ; r0 <- 13  
    mov r1, 1     ; r1 <- 1  
    push r2  
    bl f  
    pop r2  
    add r?,r2,r0 ; r? <- r2 + r0
```

## Convention d'appel (ABI): défini

- Comment sont passés les arguments (ARM: les premiers dans r0...r3)
- Comment sont renvoyés les résultats (ARM: dans r0)
- Quels registres sont préservés par l'appellant (ARM: r4...r11 + sp)



# Traduction des appels de fonctions (3): convention d'appel

## Exemple (C)

```
f.c:
int f(int a, int b)
{ return a - 2 * b }
```

-----

```
main.c:
```

```
...
int x = 33;
int y = x + f(13,1)
...
```

## Exemple (ARM assembly language)

```
f:  mul r2,r1,2   ; r2 <- r1 * 2
    sub r0,r0,r2  ; r0 <- r0 - r2
    bx lr
```

```
main: ...
```

```
    mov r4, 33    ; r4 <- 33
    mov r0, 13    ; r0 <- 13
    mov r1, 1     ; r1 <- 1
    bl f          ; appelle f
    add r?,r4,r0  ; r? <- r4 + r0
    ...
```

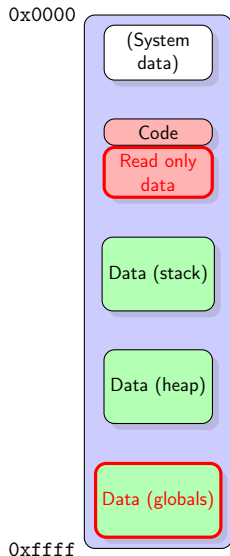
## Convention d'appel (ABI): défini

- Comment sont passés les arguments (ARM: les premiers dans r0...r3)
- Comment sont renvoyés les résultats (ARM: dans r0)
- Quels registres sont préservés par l'appellant (ARM: r4...r11 + sp)

## 1 Cours 2: Organisation de la mémoire

- Architecture matérielle
- Organisation de la mémoire (et correspondance en C)
- Organisation du code
- Organisation des données

# Organisation des variables globales



# Traduction des variables globales

## Exemple (C)

```
struct point {  
    int x;  
    int y;  
} p = {3,4};  
char c = 'q';
```

```
main() {  
    int r = p.y;  
}
```

## Exemple (ARM assembly language)

```
p:  
    .word 3  
    .word 4  
c:  
    .char 'q'
```

```
main:  
    mov32 r2, &p  
    ldr r0, [r2, #4]
```

# Traduction des variables globales

## Exemple (C)

```
struct point {  
    int x;  
    int y;  
} p = {3,4};  
char c = 'q';
```

```
main() {  
    int r = p.y;  
}
```

## Exemple (ARM assembly language)

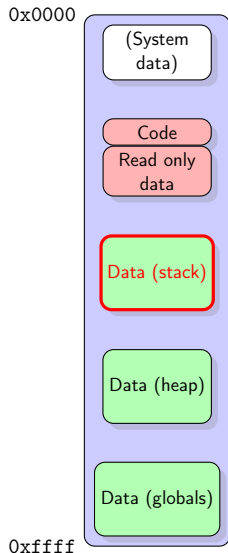
```
0x804022a0:  
        03 00 00 00  
        04 00 00 00
```

```
0x804022a8:  
        71
```

```
main:  
    mov  r2, 0x22a0 ; poids faible  
    movt r2, 0x8040 ; poids fort  
    ldr  r0, [r2]
```

- Les variables globales ont une adresse constante.
- C'est le rôle du **linker** (éditeur de lien) de placer les variables et d'insérer leur adresse dans le code.

# Organisation de la pile

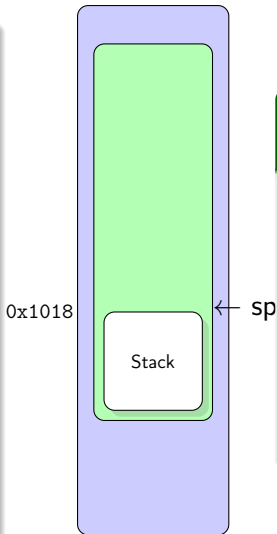


# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



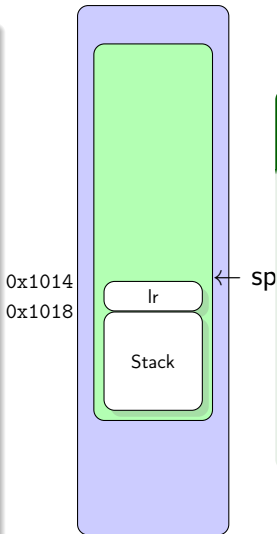
## Exemple (ARM assembly language)

# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



## Exemple (ARM assembly language)

```
g:  
    push lr                ; *sp := lr
```

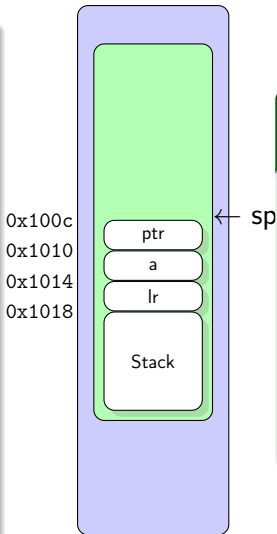


# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



## Exemple (ARM assembly language)

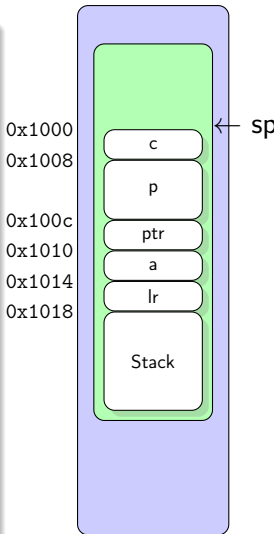
```
g:  
    push lr                ; *sp := lr  
    sub sp, sp, #8         ; sp := sp - 8  
    mov r0, #3             ; r0 := #3  
    str r0, [sp]           ; *sp := r0  
    str sp, [sp, #4]        ; *(sp + 4) := sp  
    bl f                   ; lr := pc
```

# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



## Exemple (ARM assembly language)

f:

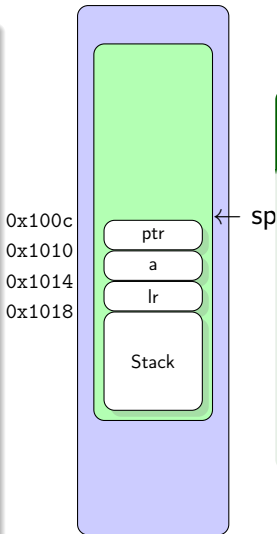
```
sub sp, sp, #12    ; sp := sp - 12  
mov r0, #3          ; r0 := 3  
str r0, [sp, #4]    ; *(sp + 4) = r0  
add r1, r0, #2      ; r1 := r0 + 2  
str r1, [sp]         ; *sp = r1  
add sp, sp, #12     ; sp := sp + 12
```

# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



## Exemple (ARM assembly language)

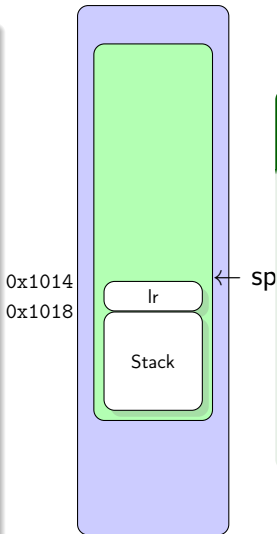
```
f:  
    sub sp, sp, #12    ; sp := sp - 12  
    mov r0, #3         ; r0 := 3  
    str r0, [sp, #4]   ; *(sp + 4) = r0  
    add r1, r0, #2     ; r1 := r0 + 2  
    str r1, [sp]        ; *sp = r1  
    add sp, sp, #12    ; sp := sp + 12  
    bx lr
```

# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```



## Exemple (ARM assembly language)

```
g:  
    push lr                ; *sp := lr  
    sub sp, sp, #8         ; sp := sp - 8  
    mov r0, #3             ; r0 := #3  
    str r0, [sp]           ; *sp := r0  
    str sp, [sp, #4]        ; *(sp + 4) := sp  
    bl f                   ; lr := pc  
    add sp, sp, #8         ; sp := sp + 8
```

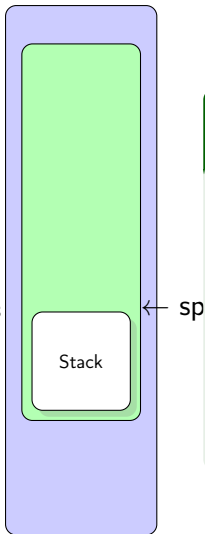
# Traduction des variables locales

## Exemple (C)

```
void f(void){  
    struct point {  
        int x;  
        int y;  
    } p;  
    char c;  
    p.y = 3;  
    p.x = p.y + 2;  
}
```

```
void g(void){  
    int a = 3;  
    int *ptr = &a;  
    f();  
}
```

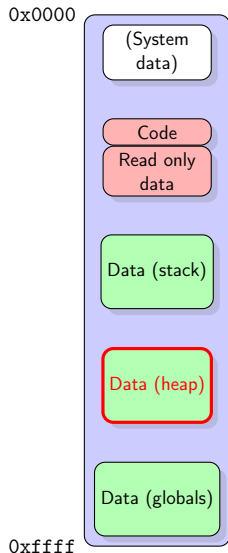
0x1018



## Exemple (ARM assembly language)

```
g:  
    push lr                ; *sp := lr  
    sub sp, sp, #8         ; sp := sp - 8  
    mov r0, #3             ; r0 := 3  
    str r0, [sp]           ; *sp := 3  
    str sp, [sp, #4]        ; *(sp + 4) := sp  
    bl f                   ; lr := pc  
    add sp, sp, #8         ; sp := sp + 8  
    pop pc                 ; sp := lr
```

# Organisation du tas



- Allocation de la mémoire du tas avec `malloc` et `free`
- Nécessité de différencier les zones libres des zones déjà allouées
- On verra comment en TP

- Un processeur:
  - Lit des instructions en mémoire
  - Manipule des registres
  - Fait des lectures et écritures
- La mémoire d'un programme est organisée en 5 grandes zones
  - Code
  - Données constantes
  - Variables globales
  - Piles
  - Tas
- Le compilateur, l'assembleur et le linker:
  - Transforme le code C en code machine en suivant des schémas pré-établis
  - S'occupent de l'allocation statique du code et des données globales