Cours 2 - La mémoire centrale

Halim Djerroud

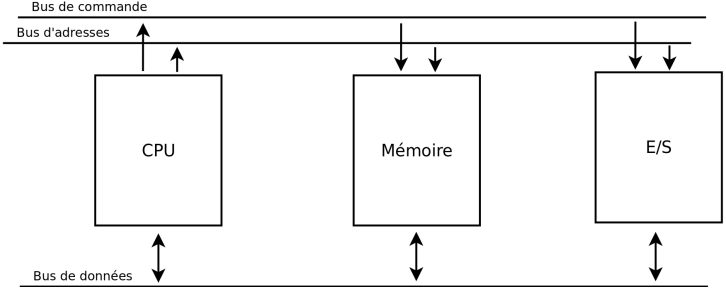
Université Gustave Eiffel
ESIEE PARIS

révision : 2.0

Architecture générale d'un calculateur

Les principaux constituants un calculateur :

- Mémoire (stocker l'information)
- Processeur (exécuter les instructions du programme)
- Entrées / Sorties Communiquer avec l'environnement



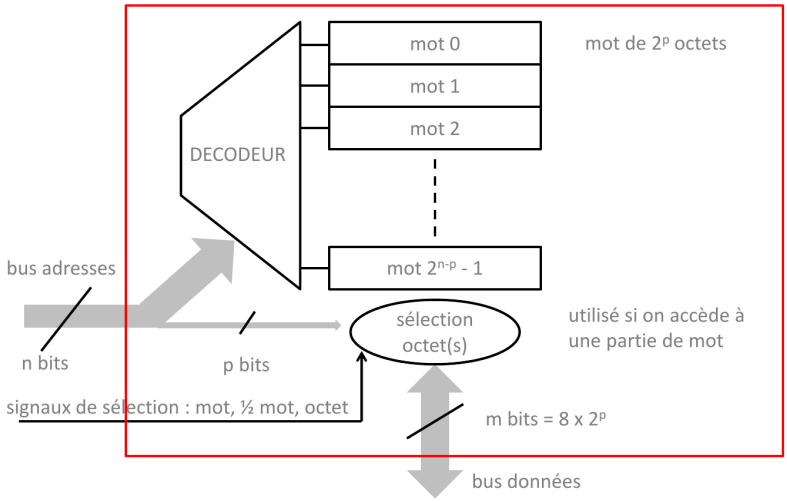
La mémoire centrale

- Mémoire dans laquelle on peut lire et écrire.
- Mémoire volatile (perd son contenu dès la coupure du courant).
- La mémoire vive RAM (Random Access Memory)

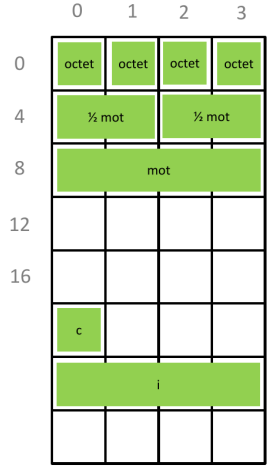
Quelques définitions

- **Mot :** C'est un regroupement de 2^n octets (case). C'est la plus grande quantité d'information transférable en un seul accès (lecture ou écriture).
- **Adresse :** C'est le numéro d'un mot-mémoire (case mémoire) dans la mémoire centrale.
- **Organisation :** La mémoire centrale est organisée en bits et en mots. Chaque mot-mémoire est repéré par son adresse en mémoire centrale.

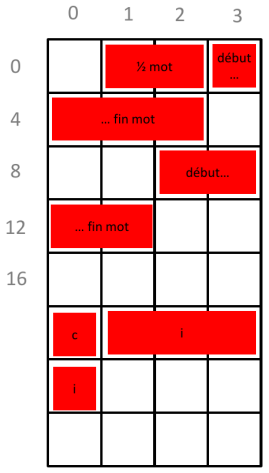
La mémoire centrale



Contraintes d'alignement



possible



impossible / peu performant

Contraintes d'alignement (2)

Contraintes à respecter :

- accès octet : pas de contrainte sur l'adresse
- 2 octets : adresse % 2 = 0
- 4 octets : adresse % 4 = 0
- 8 octets : adresse % 8 = 0

Si la contrainte n'est pas respectée alors :

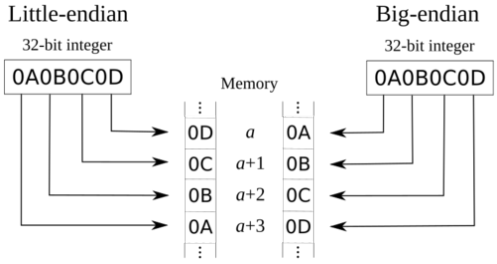
- Pour obtenir un mot il faut 2 accès mémoire.

Pour info

- Le x86 autorise l'accès en deux temps pour récupérer un mot (Pour éviter les pertes de mémoires dans les structures).
- Motorola 68000 interdit cette pratique.

Boutisme (Endianness)

- Little-endian : l'octet du poids le plus faible à l'adresse la plus petite
- Big-endian : l'octet du poids le plus fort à l'adresse la plus petite

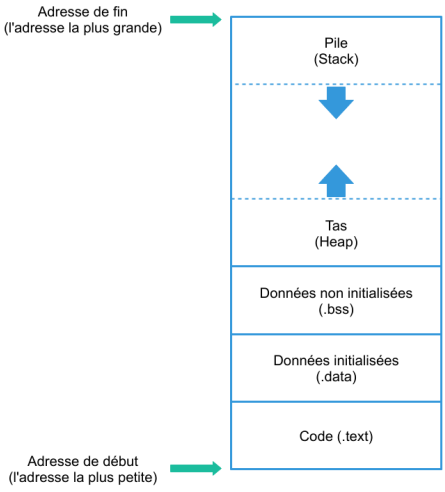


Pour info

- Le x86 autorise utilise l'ordre : Little-endian

Disposition de la mémoire

- Segment de texte (Code)
- Segment de données initialisées
- Segment de données non initialisées
- La pile
- Le tas



Exemple : fichier ELF

- ELF : Executable and Linkable Format format de fichier binaire utilisé pour l'enregistrement de code compilé (objets, exécutables et bibliothèques).

```
$ file  mon-fichier-exe
>> ELF 32-bit LSB pie executable, Intel 80386
$ readelf --sections a.out
Il y a 30 en-têtes de section, débutant à l'adresse de décalage 0x3830:

En-têtes de section :
  [Nr] Nom              Type            Adr      Décala.Taille ES ...
  [ 0]                  NULL            00000000 000000 000000 00 ...
    ...
  [11] .init            PROGBITS        00001000 001000 000020 00 ...
    ...
  [14] .text            PROGBITS        00001060 001060 0002f5 00 ...
  [16] .rodata          PROGBITS        00002000 002000 000028 00 ...
    ...
  [25] .bss             NOBITS          0000401c 00301c 000004 00 ...
    ...
```

Sections assembleur

- **.text** (read only) : la section du code. Elle contient les instructions et les constantes du programme. Un programme assembleur doit contenir au moins la section .text
- **.data** (read-write) : la section des données (data section). Elle décrit comment allouer l'espace mémoire pour les variables initialisables du programme (variables globales)
- **.bss** (read-write) : contient les variables non initialisées. Dans notre code, cette section est vide. On peut donc l'éliminer.

L'ordre des sections dans le code source n'est pas important. N'importe quelle section peut être vide !

Exemple code Assembleur

```
  .data
#  Données initialisées

  .bss
#  Données non initialisées

  .text
#  Code
```

Commentaires

Deux méthodes pour commenter un code assembleur

- /* commentaire sur plusieurs lignes */
- # commentaire jusqu'à la fin de ligne

```
/*
 commentaires
*/

# commentaire
```

Les déclarations

- Une déclaration se termine par le caractère saut de ligne n ou par le caractère « ; »
- Une étiquette peut être suivie d'un symbole clé qui détermine le type de la déclaration
- Si le symbole clé est préfixé par un point, alors la déclaration est une directive assembleur
- Les attributs d'une directive peuvent être un symbole prédéfini, une constante ou une expression

Exemple déclarations

Les déclarations peuvent prendre quatre formes :

```
                .nom directive
label_1 :       .nom directive attribut
label_2 :
                 expression
                 instruction        op1 , op2 , ...
```

Directives

- Les directives sont des pseudo-opérations
- Elles sont utilisées pour simplifier des opérations complexes du programmeur
- Une directive est un symbole préfixé par un point (.)

```
.data
  .directive
  .directive attribut
```

Les constantes

- Un nombre binaire est un 0b ou 0B suivi de zéro ou plusieurs chiffres binaires {0, 1}.
  (ex : 0b10110101)
- Un nombre octal est un 0 suivi de zéro ou plusieurs chiffres octaux 0, 1, 2, 3, 4, 5, 6, 7.
  (ex : 04657)
- Un nombre décimal ne doit pas commencer par 0. Il contient zéro ou plusieurs chiffres décimaux 0..9.
  (ex : 19351)
- Un nombre hexadécimal est un 0x ou 0X suivi de zéro ou plusieurs chiffres hexadécimaux 0..9, A, B, C, D, E, F.
  (ex : 0x4F)

les symboles et les caractères

Les symboles

- Les lettres de l'alphabet : a .. z, A .. Z
- Les chiffres décimaux : 0 .. 9
- Les caractères : (point) . $.

Table ASCII (man ascii dans un terminal) :

- 'A' est le code ASCII 65 du caractère A
- 'a' est le code ASCII 97 du caractère a
- '0' désigne le code ASCII 48 du caractère 0

Les Symboles :

Les chaînes caractères

Table ASCII (man ascii dans un terminal)

- Une chaîne de caractères (dite une string) est une séquence de caractères écrite entre guillemets
- Elle représente un tableau contigu d'octets en mémoire. Exemple : « Hello, World ! »

```
        .data
msg :     .asciz "Hello, World !\n"
```

Directives de la section .data et .bss

- **`.align` *`int`***
  La directive `.align` fait en sorte que les prochaines données soient alignées sur une adresse modulo *`int`*. *`int`* doit être une puissance de 2.

```
.data
                .align 2
var_short:      .word  0x0FFF
                .align 4
tab_3_int:      .int   10,20,30
ma_chaine:      .asciz "Bonjour ASM !"
                .align 8
var_long_long:  .quad  0xff
```

Directives de la section .data et .bss

- **`.ascii "string"`**
  La directive `.ascii` place les caractères de la chaîne *"string"* dans le module objet à l'emplacement actuel mais ne termine pas la chaîne avec un octet nul (`'\0'`). La chaîne doit être placée entre guillemets doubles (") (ASCII 0x22). La directive `.ascii` n'est pas valide pour la section `.bss`.
- **`.asciz "string"`**
  Chaîne de caractères avec le Zero de fin

```
msg1 :    .ascii "Hello, World !\0"
msg2 :    .asciz "Hello, World !"
```

Directives de la section .data et .bss

- **`.byte` *`byte1 ,byte2, ..., byteN`***

  La directive `.byte` génère des octets initialisés dans la section actuelle (un tableau d'octets). La directive `.byte` n'est pas valide pour la section `.bss`. Chaque octet doit être une valeur de 8 bits.

```
pattern:  .byte 0b01010101, 0b00110011, 0b00001111
npattern: .byte npattern - pattern
halpha:   .byte 'A', 'B', 'C', 'D', 'E', 'F'
dummy:    .4byte 0xDEADBEEF
nalpha:   .byte 'Z' - 'A' + 1
```

Directives de la section .data et .bss

- **`.word word1, word2, ..., wordN`**
  La directive `.word` définit et initialise un tableau de mots binaires (16 bits ou word). La directive `.word` n'est pas valide pour la section `.bss`. Chaque octet doit être une valeur de 16 bits.

```
w_tab:      .word    0x08FF
            .word    0x123F
            .word    0xFF00
            .word    0x0000
```

Directives de la section .data et .bss

- **`.quad` *quad1, quad2, ..., quadN***
  La directive `.quad` définit et initialise un tableau de mots binaires (64 bits ou quad). La directive `.quad` n'est pas valide pour la section `.bss`. Chaque octet doit être une valeur de 64 bits.

```
q_tab:      .quad    0x0000000000000000
            .quad    0x0000000000000000
            .quad    0x00C09200ffffffff
            .quad    0x0000000000000000
```

Directives de la section .data et .bss

- **`.int` *`int1, int2, ..., intN`***

  La directive `.int` définit et initialise un tableau de mots binaires (32 bits ou int). La directive `.int` n'est pas valide pour la section `.bss`. Chaque octet doit être une valeur de 32 bits.

```
int_tab:    .int 10, 20, 30, 40, 5*10
mat:        .int 2,4,6
            .int 8,4,9
            .int 2,2,1
```

Directives de la section .data et .bss

- **`.long` *long1, long2, ..., longN***
  La directive `.long` définit et initialise un tableau de mots binaires (32 bits ou long). La directive `.long` n'est pas valide pour la section `.bss`. Chaque octet doit être une valeur de 32 bits.

```
long_tab:    .long 10, 20, 30, 40, 5*10
mat:         .long 2,4,6
             .long 8,4,9
             .long 2,2,1
```

Directives de la section .data et .bss

- **`.fill` *`repeat, size, value`***
  Réserve `repeat` adresses contiguës dans la mémoire et les charge avec les valeurs `value` de taille `size`.

```
buffer :    .fill 1024,4,0
```

Définit un tableau contenant 1024 `int` (4 octets) à l'adresse buffer. Chaque élément du tableau est initialisé à zéro.

Directives de la section .data et .bss

- **`.lcomm` *`symbol, length`***
  Déclare un symbole local et lui attribue `length` octets sans les initialiser.

```
.bss
.lcomm buffer,1024
```

La directive `.lcomm` est uniquement valide pour la section `.bss`.

Étiquette (Labels)

- Si dans la section **.bss** ou **.data** alors ce label fait référence à une adresse mémoire
- Si dans la section **.text** alors il fait référence au compteur de programme (On peut alors utiliser l'étiquette pour se référer dans le programme)

```
    .data
msg : .asciz "Hello, World !\n"
len = . - msg
        .text
f1:
  /* code f1 */
main:
  f1
```

Symbole point (.)

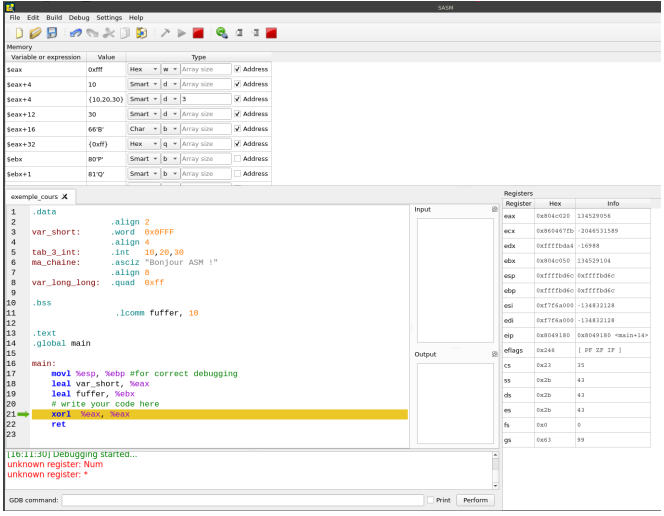
- Le symbole spécial « . » peut être utilisé comme une référence à une adresse au moment de l'assemblage.

```
        .data
msg :   .asciz "Hello, World !\n"
len = . - msg
```

Premier programme assembleur

```
.data
                .align 2
var_short:      .word  0x0FFF
                .align 4
tab_3_int:      .int   10,20,30
ma_chaine:      .asciz "Bonjour ASM !"
                .align 8
var_long_long:  .quad  0xff
.bss
    .lcomm buffer, 10
.text
.global main
main:
    movl %esp, %ebp #for correct debugging
    leal var_short, %eax
    # write your code here
        xorl  %eax, %eax
    ret
```

Premier programme assembleur : SASM



Utilisation de gdb

- GDB : GNU Debugger

```
$ gcc -m32 cours_mem_exemple.s  -g -o mon_prog -no-pie
$ gdb mon_prog
```

Utilisation de gdb

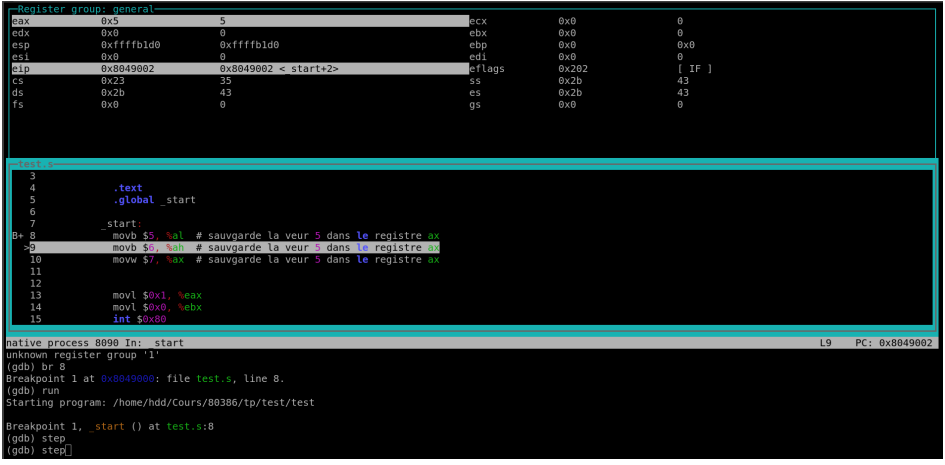


FIGURE – Exemple après exécution de la première instruction

Utilisation de gdb

```
hdd@lea:~/Cours/asm$ gdb mon_pro -q
Reading symbols from mon_pro...
(gdb) break main
Breakpoint 1 at 0x8049152: file cours_mem_exemple.s, line 17.
(gdb) run
Starting program: /home/hdd/Cours/asm/mon_pro

Breakpoint 1, main () at cours_mem_exemple.s:17
17        movl %esp, %ebp #for correct debugging
(gdb) break +4
Breakpoint 2 at 0x8049160: file cours_mem_exemple.s, line 21.
(gdb) continue
Continuing.

Breakpoint 2, main () at cours_mem_exemple.s:21
21        xorl  %eax, %eax
(gdb) x $eax
0x804c018:      0x00000fff
(gdb) x/d $eax + 4
0x804c01c:      10
(gdb) x/3d $eax + 4
0x804c01c:      10      20      30
(gdb) x/14c $eax + 16
0x804c028:      66 'B'  111 'o' 110 'n' 106 'j' 111 'o' 117 'u' 114 'r' 32 ' '
0x804c030:      65 'A'  83 'S'  77 'M'  32 ' '  33 '!'  0 '\000'
(gdb) x/10x $ebx
0x804c048 <fuffer>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804c050 <fuffer+8>:   0x00    0x00
(gdb)
```

Commandes utiles

- `file` : Déterminer le type d'un fichier
- `readelf` : Afficher des informations sur les fichiers ELF
- `hexdump` : Afficher le contenu du fichier en hexadécimal, décimal, octal ou ascii
- `objdump` : Afficher les informations des fichiers objets
- `strings` : Afficher les séquences de caractères imprimables dans les fichiers
- `GDB` : Le débogueur GNU