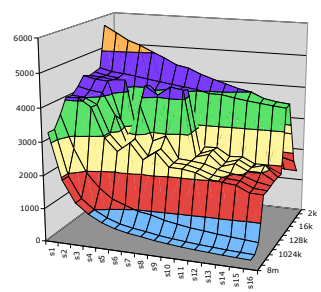
Architecture des systèmes informatiques

2 — Bits, octets et mémoire



Georges-Andre.Silber@ensmp.fr
CRI/ENSMP

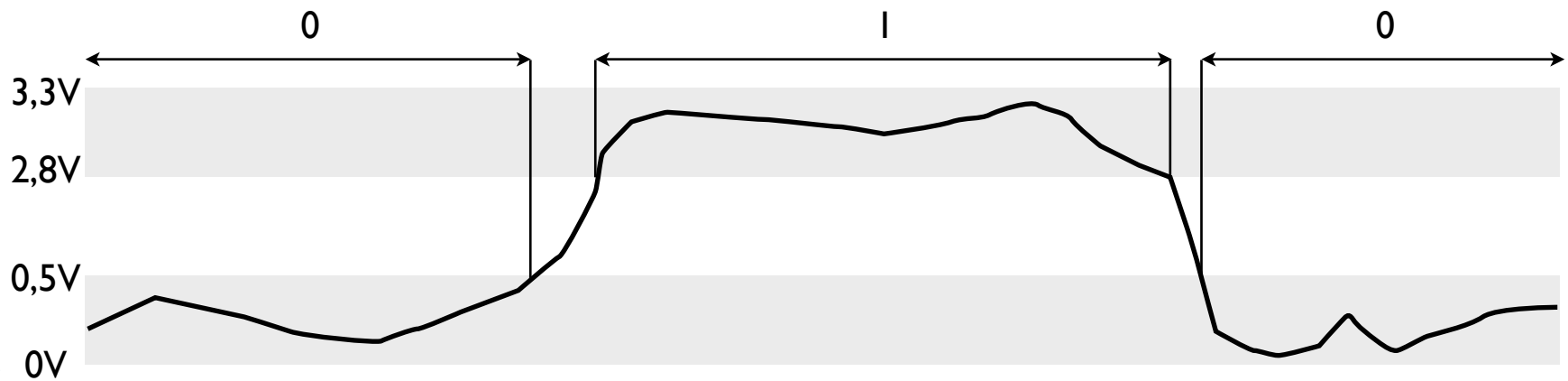
MINES PARIS
CRI

Pourquoi pas la base 10 ?

- Base 10
  - Utile pour compter avec ses doigts
  - Utilisé pour les transaction financières
- Implémentation électronique difficile
  - Stockage : 10 tubes par chiffre (ENIAC)
  - Transmission : 10 niveaux électriques
  - Arithmétique : circuits compliqués

Représentations binaires

- Représentation des nombres en base 2
  - $15213_{10} = 11101101101101_2$
  - $1,20_{10} = 1,00110011001100110011[0011]..._2$
  - $1,5213.10^4{}_{10} = 1,1101101101101_2.2^{13}$
- Implémentation électronique
  - Facile à stocker avec des éléments bistables
  - Transmission fiable sur lignes non fiables



Organisation mémoire

- Les programmes utilisent de la *mémoire virtuelle*
  - Vu comme un gros tableau d'octets
  - Réellement, une hiérarchie de mémoires diverses
  - SRAM, DRAM, disque
  - Seule les régions utilisées sont allouées
- OS modernes : espace d'adressage "privé"
  - Programme en cours = *processus*
  - Les *processus* voient uniquement "leur" mémoire

Gestion de la mémoire

- Compilateur + système d'exécution
  - Variables globales, locales, fonctions, etc...

```
int i = 100;
int v;
int main(int ac, char *av[])
{
  int k;
  int l = 0;
  int *tab;
  tab = (int*) malloc (sizeof(int) * 2);
  printf ("Adresse de main (fonction)                    : %x\n", main);
  printf ("Adresse de i (variable globale initialisee)   : %x\n", &i);
  printf ("Adresse de v (variable globale non initialisee) : %x\n", &v);
  printf ("Adresse de l (variable locale initialisee)    : %x\n", &l);
  printf ("Adresse de k (variable locale non initialisee) : %x\n", &k);
  printf ("Contenu de tab (pointeur memoire dynamique)   : %x\n", tab);
}
```

Voir memoire.c

Valeurs des octets

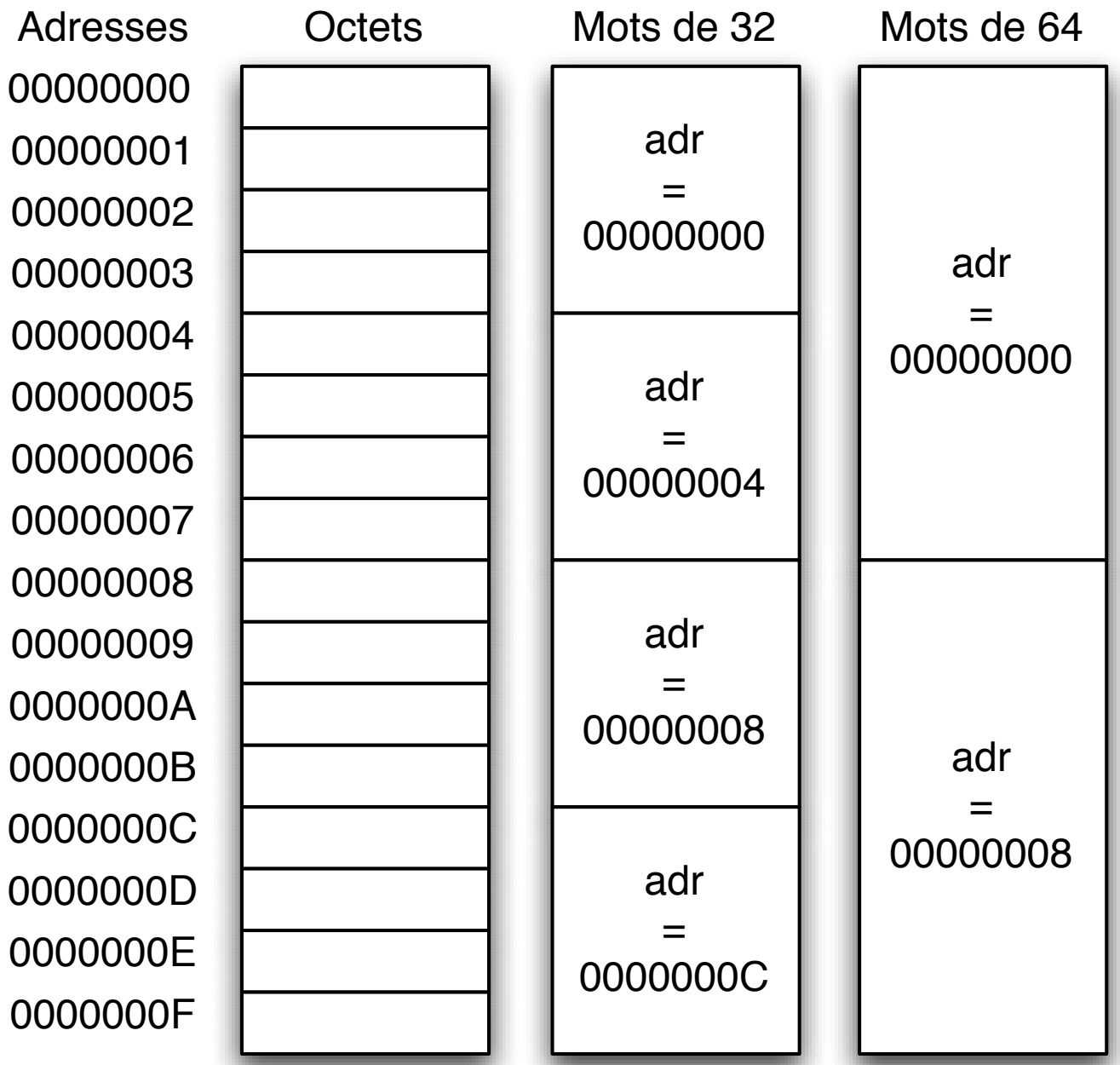
- OCTET (BYTE) = 8 BIT (BInary digiT)
- binaire : de 00000000_2 à 11111111_2
- décimal (*decimus*) : de 0_{10} à 255_{10}
- hexadécimal : de 00_{16} à FF_{16}
  - base 16 (lettres de 'A' à 'F' en plus)
  - En C, FA1D37B16 : `0xFA1DB7B16`

HEX	DEC	BIN
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Mots machine

- Un ordinateur a une *taille de mot* (word size)
  - Taille nominale des données entières
  - Taille nominale des adresses
- Aujourd'hui, 32 ou 64 bits (4 ou 8 octets)
  - 32 bits : 4 Go de RAM adressable
  - 64 bits : 18 446 744 073 709 600 000 octets
- Plusieurs découpages ou multiples de mots
- Toujours un nombre entier d'octets
- Alignement des données sur des mots

Mémoire et mots



Représentation des données

Voir taille.c

```
#include <stdio.h>

int main (int ac, char *av[])
{
  printf ("Taille int         : %d\n", sizeof(int));
  printf ("Taille long int    : %d\n", sizeof(long int));
  printf ("Taille char        : %d\n", sizeof(char));
  printf ("Taille short       : %d\n", sizeof(short));
  printf ("Taille float       : %d\n", sizeof(float));
  printf ("Taille double      : %d\n", sizeof(double));
  printf ("Taille long double : %d\n", sizeof(long double));
  printf ("Taille char*       : %d\n", sizeof(char*));
}
```

Représentation des données

PowerPC G5 / MacOS X

```
Taille int          : 4
Taille long int     : 4
Taille char         : 1
Taille short        : 2
Taille float        : 4
Taille double       : 8
Taille long double  : 16
Taille char*        : 4
```

Intel Xeon / Linux 32 bits

```
Taille int          : 4
Taille long int     : 4
Taille char         : 1
Taille short        : 2
Taille float        : 4
Taille double       : 8
Taille long double  : 12
Taille char*        : 4
```

AMD Opteron 175 / Linux 64 bits

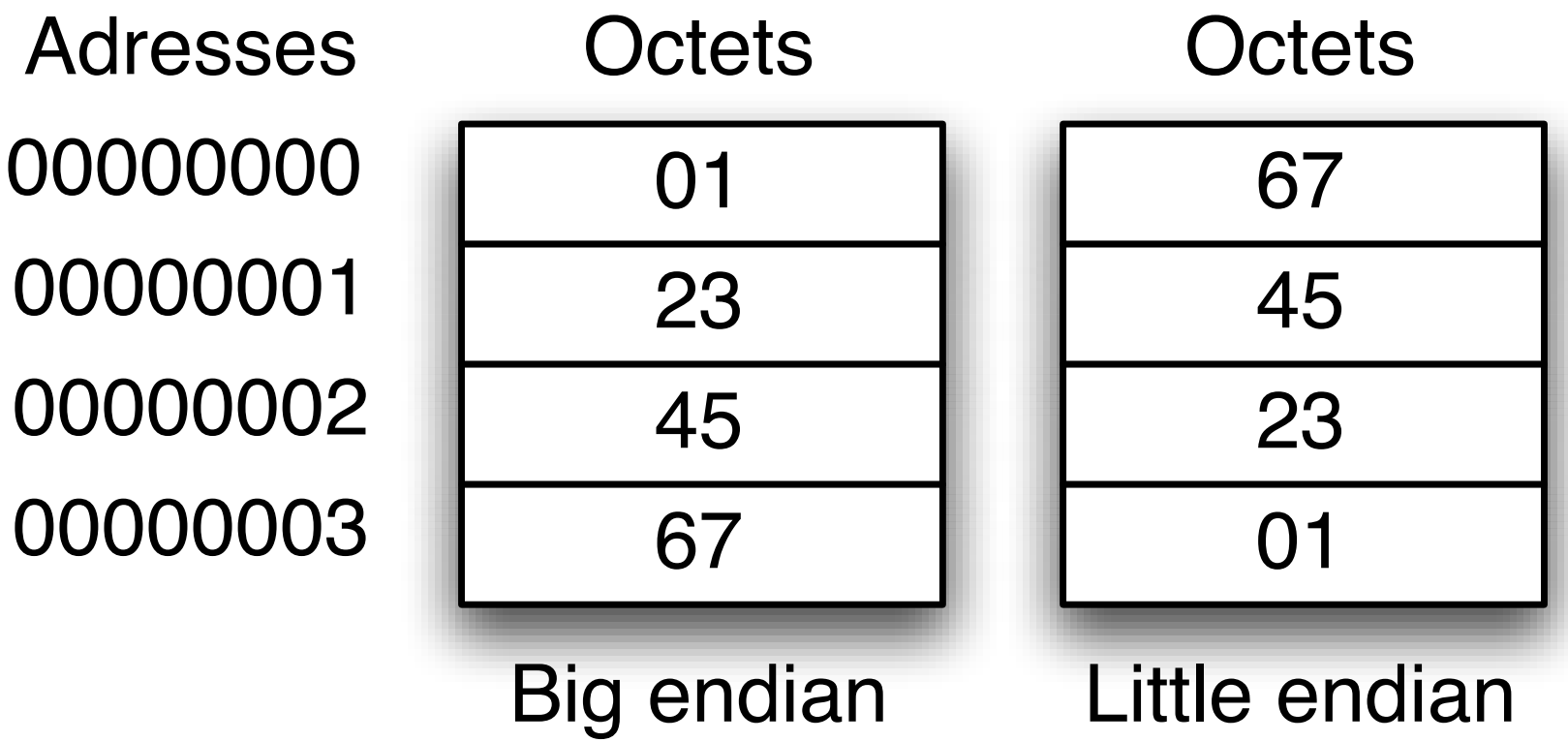
```
Taille int          : 4
Taille long int     : 8
Taille char         : 1
Taille short        : 2
Taille float        : 4
Taille double       : 8
Taille long double  : 16
Taille char*        : 8
```

Ordre des octets

- Ordre des octets dans les mots ?
  - Conventions
- PowerPC, SPARC
  - Big endian
  - Octet de “poids faible” avec la plus grande adresse
- x86, Alpha
  - Little endian
  - Octet de “poids fort” avec la plus grande adresse

Ordre des octets

- Exemple
  - Variable `x` avec une représentation `0x01234567`
  - Adresse de `x` : `0x00000000`



Afficher la représentation

Voir showbytes.c

```
#include <stdio.h>

typedef unsigned char *pointer;

void show_bytes (pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf ("%p\t0x%.2x\n", start+i, start[i]);
  printf ("\n");
}

int main (int ac, char *av)
{
  int a = 15213;
  printf ("int a = 15213 (0x3B6D);\n");
  show_bytes ((pointer) &a, sizeof (int));
}
```

Afficher la représentation

PowerPC G5 / MacOS X

Big endian

```
int a = 15213 (0x3B6D);
0xbffff778      0x00
0xbffff779      0x00
0xbffff77a      0x3b
0xbffff77b      0x6d
```

```
                                3    b    6    d
0000 0000 0000 0000 0011 1011 0110 1101
```

Intel Xeon / Linux 32 bits

Little endian

```
int a = 15213 (0x3B6D);
0xbffff1f4      0x6d
0xbffff1f5      0x3b
0xbffff1f6      0x00
0xbffff1f7      0x00
```

```
 6    d    3    b
0110 1101 0011 1011 0000 0000 0000 0000
```

Représentation des entiers

Voir showall.c

```
int a = 15213;
int b = -15213;
long int c = 15213;
...
printf ("int a = 15213 (0x00003B6D);\n");
show_bytes ((pointer) &a, sizeof (int));

printf ("int b = -15213 (0xFFFFC493);\n");
show_bytes ((pointer) &b, sizeof (int));

printf ("long int c = 15213 (0x00003B6D);\n");
show_bytes ((pointer) &c, sizeof (long int));
```

Complément à 2

AMD Opteron 175 / Linux 64

```
long int c = 15213
(0x00003B6D);
0x7fbffffbe0    0x6d
0x7fbffffbe1    0x3b
0x7fbffffbe2    0x00
0x7fbffffbe3    0x00
0x7fbffffbe4    0x00
0x7fbffffbe5    0x00
0x7fbffffbe6    0x00
0x7fbffffbe7    0x00
```

Little endian

```
int b = -15213 (0xFFFFC493);
0x7fbffffbe8    0x93
0x7fbffffbe9    0xc4
0x7fbffffbea    0xff
0x7fbffffbeb    0xff
```

Représentation des pointeurs

Voir showall.c

```
int *p = &b;
...
printf ("int *p = &b; (%p)\n", p);
show_bytes ((pointer) &p, sizeof (int *));
```

AMD Opteron 175 / Linux 64

```
int *p = &b; (0x7fbffffbe8)
0x7fbffffbd8    0xe8
0x7fbffffbd9    0xfb
0x7fbffffbda    0xff
0x7fbffffbdb    0xbf
0x7fbffffbdc    0x7f
0x7fbffffbdd    0x00
0x7fbffffbde    0x00
0x7fbffffbdf    0x00
```

Représentation des flottants

Voir showall.c

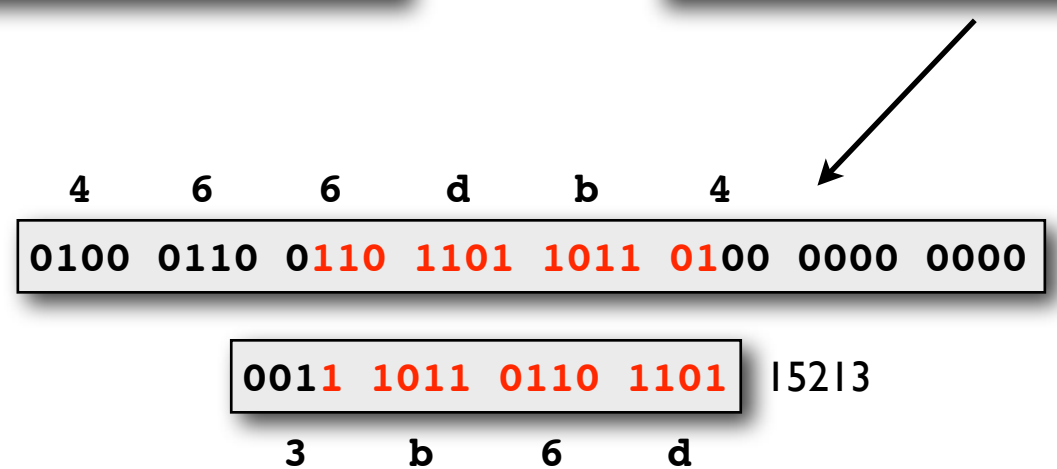
```
int *p = &b;
...
printf ("int *p = &b; (%p)\n", p);
show_bytes ((pointer) &p, sizeof (int *));
```

AMD Opteron 175 / Linux 64

```
float f = 15213.0;
0x7fbfffbd4    0x00
0x7fbfffbd5    0xb4
0x7fbfffbd6    0x6d
0x7fbfffbd7    0x46
```

PowerPC G5 / MacOS X

```
float f = 15213.0;
0xbffff798      0x46
0xbffff799      0x6d
0xbffff79a      0xb4
0xbffff79b      0x00
```



MINES PARIS CRI

Représentation des chaînes

Voir showstring.c

```
void show_chars (pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf ("%p  0x%.2x %2d '%c'\n",
       start+i, start[i], start[i], start[i]);
  printf ("\n");
}

int main (int ac, char *av)
{
  char s[6] = "15213";
  printf ("char s[6] = \"%s\";\n", s);
  show_chars ((pointer) s, sizeof (s));
}
```

```
char s[6] = "15213";
0xbffff778  0x31 49 '1'
0xbffff779  0x35 53 '5'
0xbffff77a  0x32 50 '2'
0xbffff77b  0x31 49 '1'
0xbffff77c  0x33 51 '3'
0xbffff77d  0x00  0 ''
```

- Tableau de caractères
- Chaque caractère codé en ASCII
- 0 a le code 0x30 (48)
- Doit se finir par 0x00 (en C '\0')
- Portable Little/Big endian (sauf '\n')

MINES PARIS
CRI

Alignement

- Instructions mémoire du processeurs : mots
  - Donnée alignée : une opération
  - Donnée non alignée : une ou deux opérations
- Les compilateurs essayent d'aligner les données
  - Ajout d'espace vide
  - Voir le code C `alignement.c`
- Optimisation de la taille à la demande
  - Option `-Os` de GCC

Afficher l'alignement

Voir alignement.c

```
char a;
int i;
char b;

struct data0 {
  char a;
  char i;
} d0;

struct data1 {
  char a;
  int i;
} d1;

struct data2 {
  int i;
  char a;
} d2;

struct data3 {
  char a;
  double d;
} d3;
```

```
printf ("Taille de data0 : %d octets\n",
        sizeof(struct data0));
printf ("Taille de data1 : %d octets\n",
        sizeof(struct data1));
printf ("Taille de data2 : %d octets\n",
        sizeof(struct data2));
printf ("Taille de data3 : %d octets\n",
        sizeof(struct data3));
printf ("Adresse de a    : %x\n", &a);
printf ("Adresse de i    : %x\n", &i);
printf ("Adresse de b    : %x\n", &b);
printf ("Adresse de d0   : %x\n", &d0);
printf ("  Adresse de d0.a: %x\n", &(d0.a));
printf ("  Adresse de d0.i: %x\n", &(d0.i));
printf ("Adresse de d1   : %x\n", &d1);
printf ("  Adresse de d1.a: %x\n", &(d1.a));
printf ("  Adresse de d1.i: %x\n", &(d1.i));
printf ("Adresse de d2   : %x\n", &d2);
printf ("  Adresse de d2.i: %x\n", &(d2.i));
printf ("  Adresse de d2.a: %x\n", &(d2.a));
printf ("Adresse de d3   : %x\n", &d3);
printf ("  Adresse de d3.a: %x\n", &(d3.a));
printf ("  Adresse de d3.d: %x\n", &(d3.d));
```

Alignements

PowerPC G5 / MacOS X

```
Taille de data0 : 2 octets
Taille de data1 : 8 octets
Taille de data2 : 8 octets
Taille de data3 : 12 octets
Adresse de a    : bffff758
Adresse de i    : bffff75c
Adresse de b    : bffff760
Adresse de d0   : bffff761
  Adresse de d0.a: bffff761
  Adresse de d0.i: bffff762
Adresse de d1   : bffff764
  Adresse de d1.a: bffff764
  Adresse de d1.i: bffff768
Adresse de d2   : bffff76c
  Adresse de d2.i: bffff76c
  Adresse de d2.a: bffff770
Adresse de d3   : bffff774
  Adresse de d3.a: bffff774
  Adresse de d3.d: bffff778
```

AMD Opteron 175 / Linux 64 bits

```
Taille de data0 : 2 octets
Taille de data1 : 8 octets
Taille de data2 : 8 octets
Taille de data3 : 16 octets
Adresse de a    : bfffbdf
Adresse de i    : bfffbd8
Adresse de b    : bfffbd7
Adresse de d0   : bfffbd0
  Adresse de d0.a: bfffbd0
  Adresse de d0.i: bfffbd1
Adresse de d1   : bfffbc0
  Adresse de d1.a: bfffbc0
  Adresse de d1.i: bfffbc4
Adresse de d2   : bfffbb0
  Adresse de d2.i: bfffbb0
  Adresse de d2.a: bfffbb4
Adresse de d3   : bfffba0
  Adresse de d3.a: bfffba0
  Adresse de d3.d: bfffba8
```

Intel Xeon / Linux 32 bits

```
Taille de data0 : 2 octets
Taille de data1 : 8 octets
Taille de data2 : 8 octets
Taille de data3 : 12 octets
Adresse de a    : bffff1ef
Adresse de i    : bffff1e8
Adresse de b    : bffff1e7
Adresse de d0   : bffff1e4
  Adresse de d0.a: bffff1e4
  Adresse de d0.i: bffff1e5
Adresse de d1   : bffff1d8
  Adresse de d1.a: bffff1d8
  Adresse de d1.i: bffff1dc
Adresse de d2   : bffff1d0
  Adresse de d2.i: bffff1d0
  Adresse de d2.a: bffff1d4
Adresse de d3   : bffff1c0
  Adresse de d3.a: bffff1c0
  Adresse de d3.d: bffff1c4
```

Instructions du processeur

- Programme = séquence d'opérations
  - Opération arithmétique
  - Lecture ou écriture mémoire
  - Branchement conditionnel
- Une opération : suite d'octets
  - taille fixe (PowerPC, SPARC) : RISC
  - taille variable (x86) : CISC
- Opérations spécifiques à un processeur

Exemple x86

sum.c

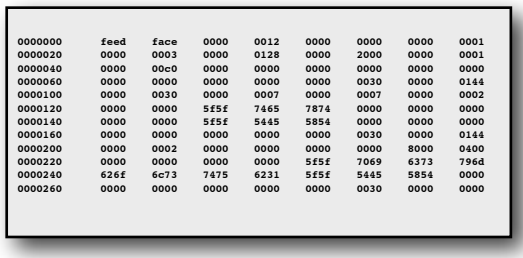
```
int sum (int x, int y)
{
  return x + y;
}
```

gcc -S sum.c

sum.s (Assembleur x86)

```
sum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret
```

gcc -c sum.c



sum.o

gdb sum.o

```
(gdb) disassemble 0
Dump of assembler code for function sum:
0x00000000 <sum+0>:     push   %ebp
0x00000001 <sum+1>:     mov    %esp,%ebp
0x00000003 <sum+3>:     mov    0xc(%ebp),%eax
0x00000006 <sum+6>:     add    0x8(%ebp),%eax
0x00000009 <sum+9>:     pop    %ebp
0x0000000a <sum+10>:    ret
End of assembler dump.
```

Code binaire “désassemblé”

Exemple PowerPC

sum.c

```
int sum (int x, int y)
{
  return x + y;
}
```

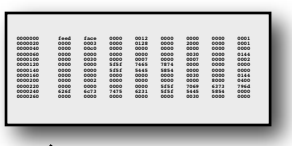
gcc -S sum.c

sum.s (Assembleur PowerPC)

```
_sum:
        stmw r30,-8(r1)
        stwu r1,-48(r1)
        mr r30,r1
        stw r3,72(r30)
        stw r4,76(r30)
        lwz r2,72(r30)
        lwz r0,76(r30)
        add r0,r2,r0
        mr r3,r0
        lwz r1,0(r1)
        lmw r30,-8(r1)
        blr
```

gcc -c sum.c

sum.o



gdb sum.o

```
(gdb) disassemble 0
Dump of assembler code for function sum:
0x00000000 <sum+0>:     stmw    r30,-8(r1)
0x00000004 <sum+4>:     stwu    r1,-48(r1)
0x00000008 <sum+8>:     mr      r30,r1
0x0000000c <sum+12>:    stw     r3,72(r30)
0x00000010 <sum+16>:    stw     r4,76(r30)
0x00000014 <sum+20>:    lwz     r2,72(r30)
0x00000018 <sum+24>:    lwz     r0,76(r30)
0x0000001c <sum+28>:    add     r0,r2,r0
0x00000020 <sum+32>:    mr      r3,r0
0x00000024 <sum+36>:    lwz     r1,0(r1)
0x00000028 <sum+40>:    lmw     r30,-8(r1)
0x0000002c <sum+44>:    blr
End of assembler dump
```

Code binaire “désassemblé”

MINES PARIS
CRI