La mémoire centrale

Chapitre 5

J.-C. Dubacq

IUT de Villetaneuse
Université Paris 13

S1D 2009

Situation physique de la mémoire

- Le CPU est connecté à un contrôleur mémoire (ou graphique) : *northbridge*.
- *northbridge* connecté au *southbridge*, connecté aux périphériques plus lents (joue le rôle de tampon).
- Fréquence du FSB=fréquence bus mémoire (de 100 à 1250 MHz).
- Cycle mémoire : inverse, souvent de 1 à 5 ns.
- Mémoire cache L1 : dans le processeur, avec les registres.
- Mémoire cache L2 : BSB, connexion plus rapide que FSB.

La taille d'un processeur

- Différentes tailles d'éléments en bits.
  - taille d'un mot-machine (registre entier).
  - taille d'une instruction (registre d'instruction).
  - taille d'une adresse (PC, *pointeur*).
  - taille d'un transfert entre RAM et processeur (mot-mémoire).
- Potentiellement tous différents, souvent identiques.
- Plus important : mot-machine.
- On parle aussi de la largeur d'un bus ou d'un registre.

Les registres et la mémoire cache

- Un processeur compte des registres généraux (entiers).
  - Taille = taille d'un mot machine = taille (générale).
  - Taille des adresses parfois différente mais toujours dans registre entier.
  - Souvent aussi des registres spécialisés (flottants double).
  - Jeu d'instructions spécialisées (x87, SSE).
- Il y a aussi d'autres registres.
- En C : `register` demande au compilateur de mettre une variable dans un registre.
- La mémoire cache est une copie (partielle) de la mémoire centrale. Ces mécanismes sont invisibles pour nous et seront vus plus tard.

La mémoire principale

- Fonctionne comme un grand tableau linéaire. Une adresse est un index dans ce tableau.
- Tout emplacement de la mémoire peut être lu ou écrit.
- Un emplacement est défini par son *adresse*.
- Selon la taille (en bits) des adresses, on a une quantité de mémoire adressable.
- Espace adressable $= 2^{\text{taille des adresses}}$.
- Taille = 4 Gio en 32 bits, 16 Eio en 64 bits.
- La taille d'une adresse est plus petite qu'un registre entier.
- Exemple : 32 bits pour coder une adresse, mémoire adressable de 2^{32} octets.
- Mémoire physique : quantité de mémoire réellement existante.
- Certaines adresses ne peuvent pas être obtenues : système complexe (mécanisme de translation de pages).

Grand et petit-boutien (little/big-endian)

- Valeur `0x4A3B2C1D`, adresse `0x00000000` ?
- Plusieurs représentations possibles :
  - petit-boutien (*little-endian*)

Contenu	4A	3B	2C	1D
Adresse	03	02	01	00

  - grand-boutien (*big-endian*)

Contenu	1D	2C	3B	4A
Adresse	03	02	01	00

- little-endian : 6502, x86, VAX.
- big-endian : Motorola 68000, SPARC, System/370.
- bi-endian : ARM, PowerPC (sauf G5), MIPS.
- Les bi-boutiens ont un mode par défaut (big-endian pour PowerPC, little-endian pour IA-64).

Mots, demi-mots, octets

- Une opération sur un mot-mémoire doit se faire avec une adresse multiple de sa taille.
- Sinon, erreur dite *d'alignement*.
- Pour mettre dans un registre **moins** qu'un mot-mémoire on utilise des mnémoniques spécifiques en fonction de la taille.
- On utilise souvent les demi-mots *half-word* et les octets.
- Attention au signe : codage usuel, C2.

Exemple (Octets signés et non-signés)

Si une adresse *A* contient `0xC8` (200/-56) et que cet octet est mis dans un registre 4 octets, le registre contiendra `0xFFFFFFC8` (-56 en C2), et non pas `0x000000C8` (200 en NAT/C2). Une mnémonique spécifique (LoadByteUnsigned doit être utilisée).

Mémoire dans un système moderne

- Adresses sur n bits, mots de x octets.
- Notation hexadécimale pour les adresses.
- 2^n octets adressables, soit $2^n/x$ mots.

Adresse	Mémoire
0x00000000	1$^{\text{er}}$mot
0x00000004	2$^{\text{e}}$mot
0x00000008	3$^{\text{e}}$mot
⋮	⋮
⋮	←1 mot→
⋮	= 4 octets
⋮	⋮
0xFFFFFFFC	$2^{30}$$^{\text{e}}$mot

Utilisation de la mémoire

- On distingue dans tous les systèmes modernes plusieurs types d'utilisation de la mémoire.
- Segment de texte *(programme)* contient les instructions.
- Segment de données contient les données.
- Données statiques : début du segment de données.
- Données dynamiques *(tas)* : au-dessus des données statiques.
- Données de fonctionnement *(pile)* : à l'envers, jusqu'au fond de pile.
- Mémoire pour le noyau et emplacements reservés.

Représentation de la mémoire (en MIPS)

Adresses	Zone
0x00000000 – 0x003FFFFF	Reservé
0x00400000 – 0x0FFFFFFF	Segment de texte
0x10000000 – 0x????????	Données statiques
0x???????? – 0x????????	Tas
0x???????? – 0x????????	Espace libre
0x???????? – 0x7FFFFFFF	Pile
0x80000000 – 0xFFFFFFFF	Reservé

Le segment de texte

- Contient le code machine correspondant aux instructions.
- Assembleur : forme textuelle, code machine : codage binaire.
- Les instructions sont consécutives dans la mémoire.
- Exécution consécutive des instructions, sauf sauts et branchements.
- Registre PC : adresse d'instruction courante.
- Point d'entrée déclaré par une étiquette `main`.

Le segment de données statiques

- Déclaration de données dont l'adresse et la taille sont connues du début à la fin du programme.
- Les étiquettes sont les seuls repères dans la mémoire pour l'assembleur.
- Les étiquettes sont transformées en constantes lors de l'assemblage.
- Souvent, une pseudo-instruction `LoadAddress` met la valeur de l'adresse dans un registre.
- Déclaration de types de base avec valeurs initiales.
- Quand pas d'étiquette, adressage par connaissance des tailles des objets.
- En C : `static` met une variable dans la zone de données statiques (comme les variables globales), et les rend donc persistante d'une invocation à l'autre de la fonction.

L'alignement dans le segment de données

- Chaque donnée prend la taille déclarée.
- Espace interstitiel nécessaire : alignement automatique des données (assembleur), ou alignement manuel.

Exemple

Déclarations de données en MIPS

- `.word` (en MIPS) introduit des données sur des adresses multiples de la taille des registres entiers, soit 4 octets.
- `.half` (en MIPS) introduit des données sur des adresses multiples de la taille des registres entiers, soit 2 octets.
- `.byte` ou `.ascii` (en MIPS) introduit des données octet par octet, ainsi que `.space` (espace de taille arbitraire).
- Possibilité de faire manuellement par `.align 4` ou `.align 2`.

La mémoire en VLTN

- Répartition conventionnelle (à part adresse 0).
- Adresses 0 et suivantes : programme.
- Adresses 50 (ou plus) : variables globales (équivalent du segment de données statiques et du tas).
- Adresses 99 et moins : pile (pour plus tard).

Parcourir un tableau

- Un tableau = données consécutives en mémoire.
- Décalage de 1 élément = élément suivant.
- En VLTN : SWAP permet de modifier par calcul PT.
- En VLTN : SETP *yy* donne la valeur *yy* à PT.
- Dans les assembleurs traditionnels : accès plus riche.
- Dans l'IBM 650 : accès par réécriture d'instructions.

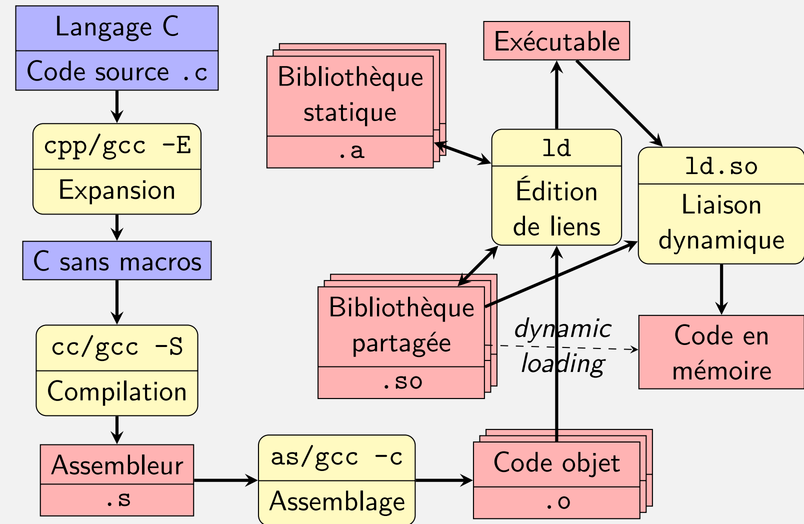
Somme d'un tableau

```
00: SETP 55 01 // Début tableau     _main:SETP _TAB
01: LDIX 00 02 // charger un élt    _bcl: LDIX 00
02: ADD  51 03 // somme partielle         ADD  _A
03: STOR 51 04                            STOR _A
04: SWAP 00 05 // passage élt             SWAP 00
05: ADDI 01 06 // suivant                 ADDI 01
06: SWAP 00 07                            SWAP 00
07: LOAD 50 08 // un élt de               LOAD _LTAB
08: SUBI 01 09 // moins à faire           SUBI 01
09: STOR 50 10                            STOR _LTAB
10: TSTN 49 01 // fini ?                  TSTN _fin _bcl
49: EJECT 0000                      _fin: EJECT 0000
55: ... (tableau)                   _A:   000
...                                 _LTAB:(long. tab.)
64: ... (fin tab)                   _TAB: (tableau)
```

Différences 32/64 bits

- En 1985, 32 bits introduit car suffisant pour numérotation BD.
- 4 Gio était 10^6 fois la mémoire typique.
- Maintenant, les processeurs 64 bits augmentent les données et l'adressage.
- Instructions de taille double.
- Alignement sur taille 8 octets.
- Perte d'espace mémoire.
- Traitement plus rapide pour calcul massif.

Rappel : du haut niveau au langage machine



Les modèles de programmation 64 bits

Besoin de garder la compatibilité des codes sources.
En C et en C++ les types changent de taille.

Modèle	Taille `int`	Taille `long`	Taille adresse	Taille `long long`	Exemples de Compilateurs
32 bits	32	32	32	64	
LP64	32	64	64	64	Tous sauf...
ILP64	64	64	64	64	Exceptions...
LLP64	32	32	64	64	Microsoft...

Pointeurs et données

- Une *donnée* stockée en mémoire occupe un certain nombre d'octets consécutifs.
- Par exemple, un entier en codage C2 (type `int`) utilise 4 octets consécutifs.
- Une *adresse* est une *donnée* comme une autre. Elle peut donc être mise dans une variable !
- Une variable qui contient l'adresse d'une donnée est un *pointeur*.
- En général on parle de pointeur sur un type (par exemple pointeur sur un entier).
- Type = Information non contenue dans la mémoire : que dans l'esprit du programmeur, pas dans l'ordinateur.

Définitions de pointeurs en C

- Si `type` est un type de variable, alors `type *a` sert à déclarer *a* comme un pointeur (sur une donnée de type `type`).
- Exemple : `int *a` déclare un pointeur sur un entier, c'est-à-dire que *a* est une adresse, et qu'à cette adresse le programme s'attend à trouver un entier (type `int`).
- Autre exemple : `char *b,c,d` déclare un pointeur sur un caractère et deux variables de type `char`, c'est-à-dire que *b* est une adresse, et qu'à cette adresse le programme s'attend à trouver un caractère (type `char`). Par exemple, *b* peut être l'adresse où est stockée *c* ou *d*.

Indirection et calcul d'adresse en C

- L'accès à un pointeur signifie récupérer le contenu pointé (ou mettre dans la case pointée) et non pas l'adresse.
- Cette opération s'appelle une *indirection*.
- Par exemple, si *a* contient l'adresse `0x7080FF4C` et que la mémoire à l'adresse `0x7080FF4C` contient l'entier `0x00243653`, on veut récupérer la valeur `0x00243653`.
- En C, l'indirection se fait par l'opérateur `*` : `c=*b;`
- Pour affecter une valeur à une variable de type pointeur, trois origines sont possibles :
  1. Pointeur constant invalide `NULL` : `a=NULL;`
  2. Calcul d'une adresse d'une variable : `b=&d;`
  3. Allocation de mémoire.
- On évitera d'affecter directement une valeur : `a=0xffff005c;` Valable, mais dangereux !

Exemple de pointeurs

Adresse	Mémoire
0x00000000	0x00000000
⋮	⋮
0x40000000	0x40000004
0x40000004	0x00000009
0x40000008	0x4000000C
0x4000000C	0x00000041 (LE)
⋮	⋮
0xFFFFFFFC	2^{30e}mot

- `int *pent,entier=9; char *pchar,car='A';`
- `pent=&entier;pchar=&car;`

Utilisation générale des pointeurs

Les pointeurs sont utilisés de très nombreuses façons :

- pour désigner des *fonctions* ;
- pour construire des structures complexes (qui ne sont pas un type primitif), par exemple des *listes* (tableaux de longueur arbitraire), des *tables de hachage*, des *files d'attente* ou des *piles* ;
- pour lier entre eux des objets de nature différente ;

Un pointeur sur un pointeur s'appelle parfois une *poignée* (en anglais *handle*).

Les pointeurs en assembleur

- Un pointeur est une adresse, c'est-à-dire un nombre.
- On peut donc le manipuler dans un registre, comme un nombre.
- Calcul d'adresse : valeur constante.
- Variable en C = emplacement mémoire en asm.
- Indirection par adressage indirect ou indexé.
- Pas de typage !

Tableaux et pointeurs

- En C comme en assembleur, un tableau est une suite de valeurs de même type rangées côte à côte.
- Pour désigner un tableau, on utilise l'adresse de son premier élément.
- Pour accéder au premier élément, on fait donc une indirection.
- Pour accéder au n-ième élément, on calcule son adresse, puis on fait une indirection.
- Comment calculer l'adresse du n-ième élément ?

Arithmétique des pointeurs

- Ajouter à un pointeur la taille totale d'un élément permet de trouver l'adresse de l'élément suivant.
- En C : `type *p,*q;` puis plus loin `q=p+1;` pour que `q` pointe sur l'élément suivant celui pointé par `p`.
- En assembleur : on doit connaître la taille, et ajouter exactement le bon nombre (attention à la taille de l'élément pointé).
- On peut bien sûr faire des soustractions aussi (on recule d'un certain nombre d'éléments).
- En C : `a[n]` est en fait `*(a+n)`.

Maquillage de type (cast) en C

- En C, il est possible de prendre une donnée et de faire croire qu'elle est d'un autre type.
- La donnée n'est pas transformée (sauf cas particuliers, par exemple `int` vers `float`).
- Exception : cast d'expression.
- Par contre, un pointeur est n'est pas transformé : une adresse reste une adresse.
- Il faut se méfier du cast : supprime les avertissements, pas forcément les problèmes !
- Existence de pointeurs génériques de type `void *`.

Allocation dynamique de mémoire

- Réserver un espace mémoire pour une variable.
- L'allocation se fait à partir du *tas* (espace mémoire destiné à l'allocation dynamique).
- En C : fonction `malloc(size)` qui renvoie soit l'adresse du début d'un espace mémoire reservé de taille *size* octets, soit NULL.
- La taille d'un type est donnée par l'instruction `sizeof`.
- Un appel typique est donc pour *n* élements de type `type` : `type *a=malloc(n*sizeof(type));`.
- Pour libérer un espace : `free(pointeur)` libère la zone mémoire située à l'adresse donnée par `pointeur` **si elle a été allouée par un malloc**.

Le tas

- Cette zone de la mémoire est située juste après le segment de données statiques.
- L'utilisation de cette zone passe forcément par l'utilisation d'un service du système d'exploitation, l'allocateur de mémoire (appel système `malloc`).
- Le service attribue des plages de mémoire de taille demandée.
- Il repère quand elles sont libérées par l'appel système `free`, et les note comme à nouveau disponibles.
- Il doit anticiper la fragmentation de la mémoire.
- La zone est aussi grande que nécessaire, mais ne doit pas dépasser la limite inférieure de la *pile*.
- La conception de l'allocateur de mémoire rentre beaucoup dans les performances d'un système d'exploitation et dépend beaucoup du profil d'usage.

Utilisation d'un débogueur C

- Pour utiliser un débogueur, il faut compiler avec l'option `-g` partout.
- Exemple : `gcc -g toto.c -o toto`
- Pour *exécuter* le programme, on fait précéder la commande du nom du débogueur : `ddd toto` ou `gdb toto`.
- Les débogueurs affichent le code source du programme, et peuvent afficher le contenu de variables à n'importe quel moment de l'exécution du programme.
- Il est possible d'exécuter le programme et de l'arrêter à une instruction spécifique : *point d'arrêt*.

Utilisation d'un débogueur C (2)

- On utilisera `ddd`, un outil graphique.
- On peut insérer des points d'arrêt où on veut.
- Il est possible de continuer l'exécution après, ou d'avancer instruction par instruction.
- Il est possible d'afficher des variables, ou d'afficher des expressions.
- Pour afficher plusieurs éléments de tableau (une *tranche*) : on tape `graph display tab[depart]@longueur`