Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Modes d'adressage

Introduction

Pendant cette séance, on va travailler sur les différentes modes d'adressage de l'architecture x86_64, c'est à dire les différents mécanismes offerts par le processeur pour accéder à des données en mémoire.
L'architecture x86_64 étant une architecture CISC, il existe beaucoup de façon d'accéder à la mémoire. On ne verra que les plus couramment utilisées ici. On notera que les noms des différents modes d'adressage ne sont pas standardisés et peuvent donc varier d'un auteur à l'autre.

Modes d'adressage utilisés

L'adressage immédiat (constantes)

Ce n'est pas un accès mémoire à proprement parler, car il sert à mettre une constante dans un registre (la constante étant codée directement dans l'instruction comme vous l'avez vu en cours d'architecture).
Par exemple : `movq $5, %rax` charge la constante (sur 64 bits) 5 dans le registre `%rax`.

L'adressage direct (variables globales)

Lorsqu'on souhaite accéder à une variable globale, on utilise un mode d'adressage très simple, qui consiste à utiliser directement l'étiquette correspondant au nom de la variable globale.
Par exemple, si on écrit en C :

```
int64_t x = 5;

void fct(void)
{
    x++;
}
```

on peut traduire ce code comme suit en assembleur :

```
    .text
fct:
    ...
    addq $1, x
    ...

    .data
.comm x, 8
```

Cela suppose bien sûr que la variable `x` a été initialisée quelque-part avant d'appeler la fonction `fct`.

L'adressage indirect

On manipule fréquemment dans les programmes C des variables qui contiennent elles-mêmes des adresses d'autres variables : on parle alors de pointeurs.
On peut manipuler ce type de variables facilement en assembleur grâce à l'adressage indirect, qui consiste à stocker l'adresse de la variable pointée dans un registre, que l'on appelle généralement *registre de base*. Attention, le registre servant de base doit obligatoirement être sur 64 bits sur x86_64 et sur 32 bits sur x86_32.
Par exemple, si `%rax` contient l'adresse d'une variable de type `int64_t x`, on écrira : `movq $5, (%rax)` pour copier 5 dans `x`.
Evidemment, il faut pouvoir copier l'adresse de `x` dans `%rax` initialement. Pour cela, on utilise l'instruction `lea` (*Load Effective Address*) comme dans l'exemple suivant :

```
int64_t x;
int64_t *ptr;

void fct(void)
{
    // on copie l'adresse de x dans ptr
    ptr = &x;
    // on copie 5 dans la case mémoire pointée par ptr
    // cette ligne est donc equivalente a "x = 5"
    *ptr = 5;
}
```

que l'on peut traduire en assembleur par le code suivant (en supposant que la variable ptr est stockée dans le registre `%rax` :

```
    .text
fct:
    ...
    // on copie l'adresse de x dans %rax
    leaq x, %rax
    // on copie 5 dans la case mémoire pointée par %rax
    movq $5, (%rax)
    ...

    .data
.comm x, 8
```

Attention : il faut bien différencier les deux lignes suivantes :

```
// copie L'ADRESSE de x dans %rax
leaq x, %rax
// copie LA VALEUR de x dans %rax
movq x, %rax
```

Il est possible d'ajouter un déplacement à l'adresse contenue dans le registre. Ce déplacement doit être une constante entière signée. On utilisera beaucoup ce mode d'adressage lorsqu'on apprendra comment accéder à des variables locales. Par exemple : `movq -8(%rbp), %rax` copie dans le registre `%rax` l'entier sur 64 bits situé dans la case mémoire dont l'adresse est calculée en enlevant 8 au contenu du registre `%rbp`.

L'adressage indirect avec index

Dans ce mode, on utilise deux registres 64 bits pour stocker l'adresse de la variable que l'on souhaite manipuler.
L'adresse de la variable est calculée selon la formule :

$$adresse = base + index \times type + deplacement$$

où :

- la base est la valeur contenue dans le premier registre (*registre de base*) ;
- l'index est la valeur contenue dans le deuxième registre (*registre d'index*) ;
- le type est une constante entière valant forcément 1, 2, 4 ou 8 ;
- le déplacement est une constante entière quelconque qui, comme indiqué ci-dessus, n'est pas multipliée par le type.

Si le déplacement est 0, on peut ne pas l'écrire. Par exemple `movq 0(%r10, %r11, 4), %rax` est équivalent à `movq (%r10, %r11, 4), %rax`.
Si on ne précise pas le type, il vaut 1 par défaut. Par exemple, `movq 5(%r10, %r11, 1), %rax` est équivalent à `movq 5(%r10, %r11), %rax`.
Ce mode d'adressage est très utile pour accéder à des tableaux : le registre de base contient l'adresse du début du tableau, le type représente la taille d'un élément du tableau et le registre d'index contient l'indice de la case à accéder. Si c'est un tableau de structures, l'index permet de passer d'une structure à la suivante, et le déplacement permet d'accéder directement au champ désiré de la structure.
Attention, dans tous les cas, les registres de base et d'index doivent obligatoirement être des registres 64 bits sur x86_64, et des registres 32 bits sur x86_32.
Par exemple, si on écrit en C :

```
int64_t tab[10];
int64_t i;

void fct(void)
{
    ...
    if (tab[i] == tab[i + 1]) {
        ...
    }
    ...
}
```

la comparaison entre `tab[i]` et `tab[i + 1]` pourra s'écrire :

```
...
leaq tab, %r10
movq i, %r11
movq (%r10, %r11, 8), %rax
cmpq 8(%r10, %r11, 8), %rax
jne fin_if
...
```

Exercices

Comme à la séance précédente, vous ne pouvez pas utiliser n'importe-quel registre dans ces exercices. Vous utiliserez uniquement les registres `%rax`, `%r10`, `%r11` (ou leurs fractions comme `%al` sur 8 bits par exemple), sauf pour le dernier exercice concernant l'optimisation de code.

Petit rappel préliminaire

Supposons qu'on a en C le code ci-dessous :

```
#include <stdio.h>

char *x = "abc";
char y[] = "abc";

int main(void)
{
```

```
    puts(x);
    puts(y);
    x++;
    puts(x);
//      y++;
    puts(y);
    return 0;
}
```

Quelle est la différence entre `x` et `y` ? Pourquoi ne peut-on pas compiler si on décommente `// y++;` ? Ecrivez la zone `data` correspondant à la définition de `x` et `y`.

Manipulation de chaînes de caractères

Dans cet exercice, on va écrire des petits programmes de manipulation de chaînes de caractères.
Commencez par récupérer l'archive contenant les sources pour cette séance. Dans ce premier exercice, on va utiliser le fichier `chaines.c` pour tester les fonctions écrites en assembleur dans le fichier `fct_chaines.s`.
Dans cet exercice, on va travailler sur une chaine de caractères définie comme une variable globale dans le fichier `chaines.c`. On peut accéder à la chaine depuis les fonctions assembleur exactement comme si elle était définie dans la zone `.data` du module assembleur, puisque les variables globales sont par défaut publiques en C.

Calcul de la longueur de la chaine

Compléter la fonction `taille_chaine` dans le fichier `fct_chaines.s` pour calculer la taille de la chaine. On rappelle qu'une chaine de caractères est terminée par le caractère `'\0'` (c'est à dire l'octet zéro) et qu'on ne compte pas ce caractère de fin dans la taille de la chaine. Attention, les caractères sont codés sur 8 bits : on ne manipule pas ici des entiers.
On demande une traduction littérale du code C ci-dessous :

```
uint64_t taille;

uint64_t taille_chaine(void)
{
    for (taille = 0; chaine[taille] != '\0'; taille++);
    return taille;
}
```

Cela signifie que vous devez déclarer une variable globale `taille` dans la zone `.data` et faire des accès mémoires systématiques : le fait de laisser la variable `taille` dans un registre est une optimisation qu'un compilateur ne ferait pas par défaut.

Inversion de chaine

Compléter la fonction `inverse_chaine` pour inverser la chaine de caractère (e.g. "azerty" donne "ytreza").
On utilisera la taille de la chaine calculée précédemment pour écrire ce code.
Le code C correspondant à traduire est :

```
int64_t dep;
char *ptr;
char tmp;

void inverse_chaine(void)
{
    dep = taille - 1;
    ptr = chaine; // attention : ici, on copie dans ptr l'adresse de la chaine
    while (dep > 0) {
```

```
        tmp = *ptr;
        *ptr = ptr[dep];
        ptr[dep] = tmp;
        dep = dep - 2;
        ptr++;
    }
}
```

On conseille de commencer par compléter la zone `.data` du module assembleur et de bien faire attention aux types et aux tailles des données manipulées.
Mettre au point le programme en utilisant GDB et en affichant la chaine résultat à la fin. On rappelle que si par exemple `%al` contient un caractère, on peut facilement afficher son contenu grâce à la commande `display /c $rax`. Pour afficher la chaine au fur et a mesure de son inversion, le plus simple est d'utiliser la commande `display /6bc (char[])chaine` (si la chaine contient 6 caractères).

Manipulation de tableaux d'entiers

On va maintenant travailler sur des tableaux non-vides d'entiers signés sur 32 bits. Le fichier `tableaux.c` contient un programme principal de test.

Tri par recherche du minimum

Compléter le fichier `fct_tableaux.s` pour implanter une fonction de tri par recherche du minimum, correspondant au programme C ci-dessous (à traduire le plus litteralement possible) :

```
uint64_t i, j, ix_min;
int32_t tmp;
void tri_min(void)
{
    for (i = 0; i < taille - 1; i++) {
        for (ix_min = i, j = i + 1; j < taille; j++) {
            if (tab[j] < tab[ix_min]) {
                ix_min = j;
            }
        }
        tmp = tab[i];
        tab[i] = tab[ix_min];
        tab[ix_min] = tmp;
    }
}
```

Le tableau `tab` et sa `taille` sont définis comme des variables globales dans le fichier `tableaux.c`, par contre vous devez définir les autres variables dans la zone `.data` du fichier `fct_tableau.s`.
Dans cette question, on vous demande de traduire chaque ligne de C indépendamment les unes des autres, c'est à dire en « oubliant » les valeurs stockées dans des registres lors de la traduction des lignes précédentes : c'est ce que ferait un compilateur non-optimisant. On verra à la question suivante comment optimiser du code assembleur.
Vous pouvez utiliser GDB pour mettre au point votre programme. Pour afficher le contenu du tableau `tab`, vous pouvez utiliser la commande `display /10d (int[])tab` (en remplaçant 10 par le nombre d'éléments que vous avez choisi).

Optimisation de code

On demande dans les exercices précédents de traduire litteralement le C en assembleur, comme le ferait un compilateur non-optimisant.
En pratique, on voit bien que l'on fait beaucoup d'accès mémoire inutiles car on recharge sans arrêt les variables (`i`, `ix_min`, etc.) dans des registres pour pouvoir les manipuler.

Une façon rapide d'optimiser le code consiste à garder des copies des variables dans des registres disponibles, et de travailler directement sur ces registres. Par contre, il n'est évidemment pas possible d'éviter les accès aux cases du tableau à trier.
Compléter la fonction `tri_min_opt` pour écrire une version optimisée du tri par recherche du minimum, en se basant sur la méthode suivante :

1. commencer par copier-coller la version non-optimisée, pour garder la même structure de l'algorithme ;
2. copier en début de fonction les variables `tab` et `taille` dans des registres : cela évitera d'avoir à les recharger lorsqu'on en aura besoin (sachant qu'on ne modifie pas ces variables dans l'algorithme démandé) ;
3. remplacer les variables globales `i`, `j`, etc. par des registres.

On verra à la prochaine séance qu'on ne peut pas utiliser n'importe-quel registre comme on veut sur l'architecture x86_64. Vous devrez donc utiliser les registres suivants :

— `%rcx` : `ix_min`
— `%edx` : `tmp`
— `%r8` : `i`
— `%r9` : `j`
— `%r10` : `taille`
— `%r11` : `tab`

En faisant ces remplacements, vous devriez pouvoir supprimer des lignes qui ne sont plus nécessaires quand on travaille directement sur des registres.
Bien entendu, un compilateur n'est pas capable de modifier un algorithme inefficace pour le rendre performant : comparer le temps d'exécution de votre version optimisée avec le tri de référence (un tri en $n \times log(n)$) en jouant sur le nombre d'éléments du tableau (la différence sera d'autant plus visible que vous augmenterez la taille du tableau).