Université de Bordeaux
Licence STS

université de BORDEAUX

ARCHITECTURE DES ORDINATEURS

TP : **04plus**

Optimisations du x86

Pour effectuer ce TP, nous allons utiliser le site `https://godbolt.org/`, qui propose un outil web permettant de traduire facilement un programme C en différents assembleurs.

Par ailleurs, si vous passez la souris sur une instruction assembleur (pas toutes), une brève description de l'instruction apparaît.

Nous allons nous intéresser aux optimisations du compilateur x86-64 gcc 7. On decochera le bouton `Intel` pour privilégier la syntaxe de l'assembleur GNU (vue en cours) à celle d'Intel. On rajoutera le drapeau de compilation `-m32` pour générer de l'assembleur 32 bits, plutôt que 64 bits.

Pour modifier le niveau d'optimisation du code produit, vous pouvez ajouter un des drapeaux de compilation suivants : de `-O0` (pas d'optimisations) à `-O3` (optimisations très agressives), ou encore `-Os` (optimisation en taille).

N.B. : Les options de compilation peuvent être saisies en haut à droite de la fenêtre.

1 Multiplication/division par une constante

La multiplication et la division sont des opérations *a priori* coûteuses. gcc est cependant capable d'optimiser grandement dans le cas où le facteur (resp. le quotient) est connu.

Considérons le code de la fonction suivante, avec les drapeaux `-m32` et `-O1` :

```
unsigned int foo (unsigned int x) {
    return (x * 2);
}
```

Dans l'assembleur généré, remarquez l'utilisation d'une instruction addition `addl`, plutôt que multiplication `mull`.

Considérez maintenant le calcul `4*x`. Remarquez l'utilisation d'une instruction de décalage, ici de 2 bits, vers la gauche (`sal` ou `shl`) pour effectuer la multiplication par 4. Il est donc inutile, lorsque vous écrivez des programmes C, de remplacer une multiplication par 4 par un décalage à gauche de 2 bits : il vaut mieux garder une version la plus lisible possible, et laisser le compilateur s'occuper de l'optimiser.

Remplacez 4 par 5. Remarquez que, selon le niveau d'optimisation utilisé, gcc va utiliser l'instruction `leal` (« `leal (a,b,i),c` » effectue `c=a+b*i`), ou bien simplement une multiplication. Essayez avec la valeur 7, et observez le code produit.

Considérez maintenant le calcul `x/2`, et observez le code produit (`sar` ou `shr` décalent vers la droite). Remplacez 2 par 3, et observez le code produit avec `-O1`.

Vous remarquerez que la division par une constante (très coûteuse) est en fait traduite en multiplication par une constante « magique » qui, par débordement, réalise le calcul de la division ! Cherchez la preuve. ;-)

De manière générale, on notera que gcc sait très bien optimiser des calculs. Il vaut donc mieux écrire ses programmes de la manière la plus lisible possible plutôt que chercher à optimiser « à la main ».

2 Une boucle for

Considérons le code suivant :

```
int boucle (int x) {
  int sum = 0;
  for (int i = 0 ; i < x ; i ++)
    sum += i;
  return sum;
}
int test () {
  return boucle (100);
}
```

Testez le résultat de la compilation avec le drapeau `-m32` en essayant successivement les différentes optimisations de `-O0` à `-O3`. Que pouvez vous dire de l'appel de fonction *test* à partir de `-O2` ?

Ajoutez maintenant le drapeau `-mavx` en plus de `-O3`. Que se passe-t-il ? Qu'est-ce que AVX ? Repérez l'instruction `vpaddd` dans le code assembleur ; que fait-elle ?

3 Pour aller plus loin : `switch`

Considérons le programme suivant :

```
char myswitch (int x) {
  switch (x) {
    case 1 : return 'A';
    case 2 : return 'Z';
    case 3 : return 'E';
  }
}
```

Observez dans le code produit le rôle de l'instruction `jmp`. Note : `ja` est presque la même chose que `jg` ; de même pour `jb` et `jl`. Quelle est la méthode utilisée ?

Modifiez maintenant le programme :

```
void myswitch (int x) {
  switch (x) {
    case 1 : printf ("foo\n"); break;
    case 2 : printf ("bar\n"); break;
    case 3 : printf ("baz\n"); break;
  }
}
```

Observez le résultat de la compilation et notamment la transformation de `printf` en `puts`, ainsi que la factorisation de l'appel.

4 Bonus : `strlen`

Considérons le programme suivant :

```
int mystrlen (char * p) {
  char * c;
  for (c = p; *c != '\0'; c ++) ;
  return (c - p);
}
```

Observez le résultat de la compilation avec `-O1` et `-Os` de la fonction *mystrlen* ; remarquez l'astuce utilisée dans le deuxième cas.