Architecture des ordinateurs
L3 Informatique 2020-2021 Binôme :
TP 4 - Code machine

- Chaque binôme rendra une feuille d'énoncé complétée et la déposera sur *e-learning* avec les programmes écrits pour répondre aux questions.
- Les exercices de ce TP sont à programmer **en langage C.**
- Ce TP a pour objectif de comprendre comment optimiser le code machine produit lors de la compilation à l'aide de connaissances générales sur l'architecture des processeurs.

Exercice 1. ★ Langage machine

Dans cet exercice, nous allons examiner la traduction en langage machine du programme suivant :

```
int main() {
  int i,j=0;
  for (i=0; i<100; i++)
    j += i;
}
```

a. Compiler le programme afin d'en faire un objet (option -c). Visualisez le contenu du fichier `.o` obtenu grâce à la commande `objdump` (options `-d` pour désassembler et `-M intel` pour utiliser la convention intel).

b. Compter le nombre d'instructions assembleur et le nombre d'octets occupés en mémoire par le programme.

c. Donner le code en langage machine de l'instruction `mov rbp,rsp`

d. Quelle(s) instructions occupent le plus d'espace mémoire et pourquoi ?

Exercice 2. ★ Désassemblage

On s'intéresse à la traduction en code assembleur du programme suivant :

```
int sum (int count){
  int s = 0;
  int i;
  for(i = 0; i < count; i++)
    s+=i;
  return s;
}
int main() {
  int count = 100000000;
  sum(count);
}
```

Pour obtenir le code assembleur (avec la syntaxe intel) généré par `gcc`, il suffit de compiler avec l'option `-S -masm=intel`.

a. Récupérer le code assembleur du programme ci-dessus. Si dans le code obtenu, vous avez des lignes qui commencent par `.cfi_`, ce sont des directives pour `GNU AS` qui ne vous seront pas utiles ; éliminez-les en rajoutant l'option `-fno-asynchronous-unwind-tables` à la compilation.

b. Comment sont stockées les variables (dans quel type de mémoire) ?

c. Donner les instructions assembleur qui correspondent à la boucle `for` (lignes 4-5).

d. Indiquer plus précisément les lignes du code assembleur qui correspondent aux trois instructions de la boucle `for` (initialisation, test, incrément).

e. Comment fait-on pour appeler une fonction ? Comment fait-on pour passer des paramètres ? Comment fait-on pour renvoyer un valeur dans une fonction ?

   Donner le code assembleur correspondant à chacune de ces trois actions dans notre exemple.

Exercice 3. ★★ Optimisations

Reprendre le code C de l'exercice précédent et générer le code assembleur, mais cette fois-ci, en compilant avec l'option d'optimisation `-O1` (attention, c'est un 'O' majuscule, pas un zéro).

a. Dans la fonction `sum`, comment sont stockées les variables (dans quel type de mémoire) ? Quel est l'intérêt ?

b. Observer la partie du code qui correspond au `main`... Que s'est-il passé ?

c. Modifier le code en C pour que la fonction `sum` soit exécutée.

d. Comparer le reste du code avec ce que l'on avait obtenu sans optimisation, noter les différences principales et essayer d'expliquer.

e. Même question en utilisant l'optimisation `-O2`.

f. Changer la valeur de `count` pour une valeur (beaucoup) plus petite et compiler avec `-O2`. Qu'observe-t-on dans le code assembleur ?

g. Modifier la fonction `sum` pour faire une somme de carrés et compiler avec `-O2`. Qu'observe-t-on dans le code assembleur ?

Exercice 4. ★★★ Optimiser la récursion

a. On considère le code suivant :

```c
int fac(int n){
  return (n<=1) ? 1 : n * fac(n-1);
}
int fac2_rec(int n, int acc){
  return (n<=1) ? acc : fac2_rec(n-1, n * acc);
}
int fac2(int n){
  return fac2_rec(n,1);
}
```

Quelle est la principale différence entre les fonctions `fac` et `fac2` ?

b. Générer le code assembleur de ces fonctions, en compilant avec l'option `-O1` puis avec l'option `-O2` et comparer.

(a) Y-a-il des différences de traitement entre les deux fonctions ?

(b) Quelle est la principale différence entre les deux optimisations ? Quel est l'intérêt ?

c. Refaire les même tests avec la fonction qui calcule les nombres Fibonacci :

```c
int fib(int n){
  return (n<=2) ? 1 : fib(n-1) + fib(n-2);
}
int fib2_rec(int n, int acc1, int acc2){
  return (n<=2) ? acc2 : fib2_rec(n-1, acc2, acc1 + acc2);
}
int fib2(int n){
  return fib2_rec(n,1,1);
}
```

Exercice 5. ★★★★ Combiner les optimisations
Compiler le programme suivant sans optimisation, puis avec `-O1` et ave `-O2`. Mesurer le temps d'exécution de chacune des trois versions avec la commande `time`. Que constate-t-on ?

```c
int squareMod(int i, int mod){
  return (i*i) % mod;
}

int sumMod (int count, int mod){
    int s = 0, i;
    for(i = 0; i < count; i++)
      s+=squareMod(i,mod);
    return s % mod;
}

int main() {
    int count = 100000000, s = 0, i;
    for(i = 0; i < 10; i++){
      s+=sumMod(count,25);
    }
    printf("%d\n", s);
}
```

Pour comprendre, générer le code assembleur correspondant et observer ce qui se passe dans le main. Expliquer les différences observées.