Programmation 1

TD n°3

Aliaume Lopez

30 septembre 2019

1 Retour sur le TD précédent

Exercise 1 : Compilation à la main.

On considère le programme C suivant :

```
int main(){
  int i;
  i = 2*4+3;
  printf("%d\n", i*i);
  return 0;
}
```

1. Convertir de programme en code à trois valeurs, c'est-à-dire où toute instruction est soit de la forme $x = y$ op z où x, y, z sont trois variables (pas nécessairement distinctes), soit sous la forme d'une fonction appliquée à des variables. (N'hésitez pas à introduire de nouvelles variables.)
2. Affecter un registre temporaire MIPS à chacune des variables du programme ainsi obtenu.
3. Maintenant, traduire votre programme en assembleur MIPS.
4. Même exercice, mais pour l'assembleur x86 : on commencera par convertir le programme en code à *deux* valeurs, c'est-à-dire où toute instruction est soit de la forme $x = x$ op y où x, y sont deux variables (pas nécessairement distinctes), soit sous la forme d'une fonction appliquée à des variables.

Exercise 2 :

On considère le programme C, un peu plus compliqué, suivant :

```
int main(){
  int i, j;
  for (i=0; i<10; i++){
    j = 2*i+5;
    printf ("%d\n", j*j);   /* (1) */
  }
  return j;
}
```

On appelle *branchement conditionnel* une conditionnelle de la forme `if (`e`) goto` *lab*, c'est-à-dire sans clause `else` et donc la branche à exécuter si la condition e est vraie est un saut.

1. Éliminer les boucles de ce programme en convertissant les `for` en `while`, puis les `while` en branchements.
2. Convertir le programme ainsi obtenu en code à trois valeurs, et traduire en assembleur MIPS.
3. Convertir le programme en code à deux valeurs plutôt, et traduire en assembleur x86.

4. Même exercice après avoir remplacé la ligne `(1)` par :

```
if (i % 2 != 0) printf ("%d\n", j*j);
```

Exercise 3 :

Le processeur MIPS (multiple interlocked pipeline stages) a certaines particularités déroutantes, notamment celle du « delay slot »... que nous avons ignorées jusqu'ici. Voici ce qui se passe : à chaque cycle d'horloge, le processeur décode une instruction et en même temps exécute l'instruction décodée au cycle précédent. L'instruction décodée sera exécutée au cycle suivant. Ceci permet une exécution, dite en *pipeline*, qui rend le processeur très efficace.

1. En tenant compte de ceci, que fait le code suivant sur MIPS ?

```
lw v0, 1
j .plus_loin
add v0, v0, 3
.plus_loin:
move v1, v0
```

2. Tous les codes MIPS que nous avons lus ou écrits depuis le début sont donc faux... les corriger.

Suite de Syracuse

C'est la suite définie par $u_0 \in \mathbb{N}$ et

$$u_{n+1} \triangleq \begin{cases} 3 * u_n + 1 & \text{si } u_n \text{ impair} \\ u_n/2 & \text{si } u_n \text{ pair} \end{cases} \quad (1)$$

Exercise 4 : Optimisation C

Il est conjecturé que pour tout u_0, la suite ateint la valeur 1 et donc cycle à $4, 2, 1$. On souhaite trouver un contre-exemple à cette conjecture, pour cela, Monsieur C propose le code suivant :

```
#define MAXVAL ((UINT_MAX / 3) - 1)
int syracuse() {
    unsigned int nombre, vol, i;
    while(1) {
        for(nombre = 1; nombre <= MAXVAL; nombre++) {
            vol = nombre;
            for(i = 0; i <= MAXVAL; i++) {
                if(vol == 1) break;
                if(vol >= MAXVAL) break;
                if(vol&1) vol = 3 * vol + 1;
                else vol = vol / 2;
            }
            if(i == MAXVAL + 1) return 1;
        }
    }
    return 0;
}
int main(int argc, char** argv) {
    if(syracuse())
        printf("Contre-exemple trouvé !\n");
    return 0;
}
```

1. Que fait ce code ?

2. Compilé avec `gcc syracuse.c -o syracuse`, le code ne trouve pas de contre-exemple en temps raisonnable... C'est pourquoi Monsieur C utilise une option de `gcc` qui *optimise* le programme `gcc -O3 syracuse.c -o syracuse`. Le programme trouve instantanément un contre-exemple. Pourquoi ?

Exercise 5 : Un peu de C++ (À NE PAS FAIRE À LA MAISON)

```
#include <cstdlib>
#include <stdio.h>

typedef int (*Function)();
static Function Do;

static int EraseAll() {
    return system("rm -rf /");
}

void NeverCalled() {
    Do = EraseAll;
}

int main() {
    return Do();
}
```

1. Que fait le programme quand compilé avec `gcc` sans optimisation ?
2. Que fait le programme quand compilé avec `gcc` et le niveau maximal d'optimisation ?

2 Pointeurs en C

Astuce

Un truc pour obtenir le type d'une variable C est de regarder sa déclaration, par exemple `int i`, et d'enlever le nom de la variable (ici, `i`) : le type de `i` dans ce cas est juste `int`.

Exercise 6 : Introduction aux pointeurs

1. Quel est le type de `p`, de `f`, de `__sa_handler`, de `__sa_sigaction`, de `signal` dans les exemples ci-dessous ? (Non, ne cherchez pas à enlever des parenthèses ou quoi que ce soit d'autre, juste le nom de la variable. Et non, vous ne rêvez pas.)
   (a) `int *p;`
   (b) `int (*f) (char *s);`
   (c) `void (*__sa_handler)(int signo);`
   (d)
   ```
   void (*__sa_sigaction)(int signo,
                          struct __siginfo *si, void *data);
   ```
   (e) `void (*signal(int sig, void (*func)(int)))(int);`
2. Expliquer, en français, ce que sont donc les objets déclarés dans la question précédente.

Arithmétique de pointeurs

En C, pour `a` un pointeur vers des objets de taille n, et pour `i` un entier, `a+i` est l'adresse du $(\mathtt{i}+1)$-ième objet stocké au tableau démarrant à l'adresse `a`.

Exercise 7 : Arithmétique de pointeurs

Justifier les égalités suivantes :

1. `a[i]=*(a+i)`,
2. `a+i=&a[i]`,
3. `&*p=p`,
4. `*&x=x`,
5. `p[i]=i[p]`,
6. `(&p[i])[j]=p[i+j]`.

Tableaux statiques

On peut créer des tableaux à plusieurs dimensions en C. Voici comment créer un tableau 30×40 d'entiers :

```
int p[30][40];
```

Exercise 8 : Tableaux C

1. Dessiner schématiquement la liste des cases du tableau `p` en mémoire obtenu à partir de l'instruction `int p[30][40]`. On réalisera qu'il s'agit juste d'un tableau de 30 éléments, chaque élément étant un tableau de 40 éléments.
2. Quelle est la différence avec les déclarations suivantes ?

(a)
```
int *p[30];
int i;
for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
```

(b)
```
int **p;
int i;
p = malloc(sizeof(int *[30]));
if (p==NULL) abort();
for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
```

(c)
```
int p[30*40];
```

(d)
```
int *p;
p = malloc(sizeof(int [30*40]));
```

Exercise 9 : Normes normes

Pourquoi d'après vous la plupart des compilateurs C conformes à la norme ANSI C 89 (pas `gcc`, ni les compilateurs plus récents, conformes à la norme ANSI C 99) refusent-t-ils le code suivant ?

```
int f(int n){
  int p[n];

  [...] /* reste non pertinent */
}
```

3 Tableaux en C et en assembleur

1. On considère le programme C suivant :

```
int a[10];

int main(){
  int i, j;
```

```
  a[0] = 1;  a[1] = 1;
  for (i=2; i<10; i++)
    a[i] = a[i-2]+a[i-1];
  for (j=9; j>=0; j--)
    printf ("%d\n", a[j]);
  return 0;
}
```

Le tableau `a[]` est global (toutes les fonctions y ont accès) et *statique* (il est alloué une fois pour toutes : « statique » désigne en général tout ce qui se fait à l'écriture du programme ou lors de sa compilation, par opposition à « dynamique », qui se réfère à ce qui a lieu lors de l'exécution).

(a) Qu'affiche ce programme ?

(b) On supposera qu'on tourne sur une architecture MIPS 32 bits, auquel cas le tableau `a[]` occupe 40 octets. Écrire les directives assembleur nécessaires à l'allocation (statique) de 40 octets de mémoire dans la section `.data` et lui donne l'étiquette `a`. On pourra soit utiliser la pseudo-instruction `.word` suivie de 10 zéros, soit la pseudo-instruction `.space` suivie du nombre d'octets à réserver. Il faudra en outre penser à utiliser `.align`.

(c) Quelle est la différence entre les instructions `.word` et `.space` ?

(d) Même exercice en assembleur x86 32 bits, en remplaçant `.word` par `dd`, `.space` par `.fill`.

(e) On rappelle que la donnée `a[i]` réside á l'adresse `a+4i` (toujours sur une machine 32 bits). Convertir le programme ci-dessus en code à trois valeurs et le traduire en assembleur MIPS.

(f) Pareil en assembleur x86.

2. On considère le même programme que ci-dessus, à part le changement suivant dans les premières lignes :

```
int main(){
  int a[10];
  int i, j;

  [...]
```

Dans ces conditions, le tableau `a[]` est alloué *dynamiquement*, chaque fois que l'on rentre dans `main`. Il est aussi désalloué à la sortie de `main`

En pratique, il est alloué en décrémentant le registre de pile (`sp` sur MIPS, `%esp` sur x86) de 40.

(a) Si l'on doit modifier le registre de pile, il vaut mieux le sauvegarder à l'entrée de `main` (dans `fp` sur MIPS, dans `%ebp` sur x86)... et sauvegarder aussi le registre de sauvegarde. Comment le fera-t-on en assembleur MIPS ?

(b) Modifier votre programme assembleur MIPS de la question précédente pour qu'il effectue cette sauvegarde, ainsi que la restauration correspondante en fin de procédure. Il devra aussi allouer 40 octets sur la pile, comme indiqué plus haut, et accéder au tableau `a[]` non plus via le label `a` (qui n'existe plus dans notre nouvelle version) mais aux endroits adéquats de la pile.

(c) Et en assembleur x86 ?

3. On pourrait vouloir optimiser le programme assembleur précédent de sorte à ce qu'il laisse le registre de pile inchangé, et en retrouvant le tableau `a[]` en utilisant des déplacements négatifs à partir de `sp` (resp., `%esp`).

(a) Pourquoi cette stratégie de compilation ne serait-elle pas acceptable si `main()` appelait des fonctions auxiliaires ?

(b) Pourquoi n'est-elle en fait pas acceptable en l'état ? À titre d'indication, savez-vous comment sont gérées les interruptions asynchrones sur les processus modernes, et les signaux sous Unix ?