Logiciel de Base : examen de première session

ENSIMAG 1A

Année scolaire 2009–2010

Consignes générales :
- Durée : 2h. Tous documents et calculatrices autorisés.
- Le barème est donné à titre indicatif.
- Les exercices sont indépendants et peuvent être traités dans le désordre.
- **Merci d'indiquer votre numéro de groupe de TD et de rendre votre copie dans le tas correspondant à votre groupe.**

Consignes relatives à l'écriture de code C et assembleur Pentium :
- Pour chaque question, une partie des points sera affectée à la clarté du code et au respect des consignes ci-dessous.
- Pour les questions portant sur la traduction d'une fonction C en assembleur, on demande d'indiquer en commentaire chaque ligne du programme C original avant d'écrire les instructions assembleur correspondantes.
- Pour améliorer la lisibilité du code assembleur, on utilisera systématiquement des constantes (directives `.set` ou `.equ`) pour les déplacements relatifs à `%ebp` (*i.e.* paramètres des fonctions et variables locales). Par exemple, si une variable locale s'appelle `var` en langage C, on y fera référence avec `var(%ebp)`.
- Sauf indication contraire dans l'énoncé, on demande de traduire le code C en assembleur de façon systématique, sans chercher à faire la moindre optimisation : en particulier, **on stockera les variables locales dans la pile** (pas dans des registres), comme le fait le compilateur C par défaut.
- On respectera les conventions de gestions des registres Intel vues en cours, c'est à dire :
  - `%eax`, `%ecx` et `%edx` sont des registres *scratch* ;
  - `%ebx`, `%esi` et `%edi` ne sont pas des registres *scratch*.

Ex. 1 : Exercice préliminaire *(6 pts)*

Il est fortement recommandé de commencer par traiter cet exercice qui porte sur des notions utiles dans la suite de l'examen.

Soit la fonction C suivante :

```
void echanger(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

Question 1 Traduisez cette fonction en assembleur sans chercher à l'optimiser.

```
    .text
// void echanger_asm(int *a, int *b);
    .globl _echanger_asm
_echanger_asm:
    .set a, 8
    .set b, 12
    enter $4, $0
    .set t, -4
    // t = *a;
    movl a(%ebp), %eax
    movl (%eax), %eax
    movl %eax, t(%ebp)
    // *a = *b;
    movl b(%ebp), %eax
    movl (%eax), %eax
    movl a(%ebp), %edx
    movl %eax, (%edx)
    // *b = t;
    movl t(%ebp), %eax
    movl b(%ebp), %edx
    movl %eax, (%edx)
    leave
    ret
```

On rappelle que l'instruction `enter $8, $0` est exactement équivalente à la séquence d'instructions :

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
```

Soit le type de données structuré et la fonction assembleur suivante :

```
1       .text
2   /*
3   struct rationnel_t {
4       int n;
5       unsigned d;
6   };
7   */
8       .set n, 0
9       .set d, 4
10  // void afficher_rat_asm(struct rationnel_t q)
11      .globl afficher_rat_asm
12  afficher_rat_asm:
13      .set q, 8
14      enter $0, $0
15      //
16      cmpl $0, (q + d)(%ebp)
17      jne else
18      //
19      pushl $ch_err
20      call printf
21      addl $4, %esp
22      jmp fin
23  else:
24      //
25      pushl (q + d)(%ebp)
26      pushl (q + n)(%ebp)
27      pushl $ch_aff
28      call printf
29      addl $12, %esp
30  fin:
31      leave
32      ret
33      .data
34  ch_err:     .asciz "Erreur !\n"
35  ch_aff:     .asciz "q = %d / %u\n"
36
```

Question 2 Donner le code C correspondant à cette fonction assembleur. Pour vous aider, on a laissé les `//` dans le code fourni (lignes 15, 18 et 24) qui délimitent la traduction assembleur de chaque ligne de C originale.

```
void afficher_rat(struct rationnel_t q)
{
    if (q.d == 0) {
        printf("Erreur !\n");
    } else {
        printf("q = %d / %u\n", q.n, q.d);
    }
}
```

Soit maintenant la fonction C suivante, qui se base sur le même type de données que précédemment :

```
void afficher_ptr_rat(struct rationnel_t *q)
{
    if (q->d == 0) {
        printf("Erreur !\n");
    } else {
        printf("q = %d / %u\n", q->n, q->d);
    }
}
```

Question 3 Traduire cette fonction en assembleur sans chercher à l'optimiser (on demande une traduction littérale du code C donné, vous n'avez notamment pas le droit d'appeler le code assembleur de la question précédente, mais vous pouvez bien sûr vous en inspirer).

```
    .text
/*
struct rationnel_t {
    int n;
    unsigned d;
};
*/
    .set n, 0
    .set d, 4
// void afficher_ptr_rat_asm(struct rationnel_t *q)
    .globl _afficher_ptr_rat_asm
_afficher_ptr_rat_asm:
    .set q, 8
    enter $0, $0
    // if (q->d == 0)
    movl q(%ebp), %eax
    cmpl $0, d(%eax)
    jne else
    // printf("Erreur !\n");
    pushl $ch_err
    call _printf
    addl $4, %esp
    jmp fin
else:
    // printf("q = %d / %u\n", q->n, q->d);
    movl q(%ebp), %eax
    pushl d(%eax)
    pushl n(%eax)
    pushl $ch_aff
    call _printf
    addl $12, %esp
fin:
    leave
    ret
```

```
    .data
ch_err:     .asciz "Erreur !\n"
ch_aff:     .asciz "q = %d / %u\n"
```

Ex. 2 : Tableaux et listes chaînées *(6 pts)*

Dans cet exercice, on va travailler sur une fonction

`void decouper(int tab[], struct cellule_t **liste)`

qui prend en entrée un tableau d'entiers signés et construit en sortie une liste chaînée composée de certains des entiers du tableau initial.

Les cellules de la liste chaînée produite en sortie seront du type :

```
struct cellule_t {
  int val;
  struct cellule_t *suiv;
};
```

Soit alors une réalisation en assembleur de cette fonction :

```
1       .text
2   /*
3   struct cellule_t {
4       int val;
5       struct cellule_t *suiv;
6   };
7   */
8       .set val, 0
9       .set suiv, 4
10  // void decouper_asm(int tab[], struct cellule_t **liste);
11      .globl decouper_asm
12  decouper_asm:
13      .set tab, 8
14      .set liste, 12
15      enter $16, $0
16      .set sent, -8
17      .set queue, -12
18      .set i, -16
19      //
20      leal sent(%ebp), %eax // equivalent a : movl %ebp, %eax ; addl $sent, %eax
21      movl %eax, queue(%ebp)
22      //
23      movl $0, i(%ebp)
24  for:
25      //
26      movl tab(%ebp), %eax
27      movl i(%ebp), %ecx
28      cmpl $0, (%eax, %ecx, 4)
29      je fin_for
```

```
30      //
31      movl tab(%ebp), %eax
32      movl i(%ebp), %ecx
33      cmpl $0, (%eax, %ecx, 4)
34      jng else    // Jump if Not Greater
35      //
36      pushl $8
37      call malloc
38      addl $4, %esp
39      movl queue(%ebp), %edx
40      movl %eax, suiv(%edx)
41      //
42      movl queue(%ebp), %eax
43      movl suiv(%eax), %eax
44      movl %eax, queue(%ebp)
45      //
46      movl tab(%ebp), %eax
47      movl i(%ebp), %ecx
48      movl (%eax, %ecx, 4), %eax
49      movl queue(%ebp), %edx
50      movl %eax, val(%edx)
51      //
52      movl queue(%ebp), %eax
53      movl $0, suiv(%eax)
54  else:
55      //
56      addl $1, i(%ebp)
57      jmp for
58  fin_for:
59      //
60      movl queue(%ebp), %eax
61      movl $0, suiv(%eax)
62      //
63      movl (sent+suiv)(%ebp), %eax
64      movl liste(%ebp), %edx
65      movl %eax, (%edx)
66      leave
67      ret
68
```

Question 1 Expliquer en 5 lignes maximum ce que fait la fonction `decouper_asm`.

Cette fonction parcours le tableau `tab` donnée en entrée et construit en sortie la liste chaînée `liste` composée de tous les entiers strictement positifs présents dans le tableau initial. Les entiers sélectionnés apparaissent dans `liste` dans le même ordre que dans `tab`. La fin du tableau initial est indiquée par une case contenant la valeur entière 0.

Question 2 Donner le code C équivalent à la fonction `decouper_asm`. N'oubliez pas les déclarations des variables utilisées.

```
void decouper(int tab[], struct cellule_t **liste)
{
    struct cellule_t sent;
    struct cellule_t *queue = &sent;
    for (unsigned i = 0; tab[i] != 0; i++) {
        if (tab[i] > 0) {
            queue->suiv = malloc(sizeof(struct cellule_t));
            queue = queue->suiv;
            queue->val = tab[i];
            queue->suiv = NULL;
        }
    }
    queue->suiv = NULL;
    *liste = sent.suiv;
}
```

Question 3 Ecrire une implantation assembleur optimisée de la fonction `decouper_asm`. On cherchera donc à minimiser le nombre d'accès mémoire en utilisant au mieux les registres disponibles.

Il est utile de chercher à supprimer les accès mémoires ayant lieu dans la boucle : ceux à l'extérieur ne seront exécutés qu'une fois ce qui ne changera pas grand-chose au temps total d'exécution sur un grand tableau. De plus, l'appel à `malloc` dans le corps de la boucle nous empêche de mieux utiliser les registres *scratch* : si on essayait de stocker par exemple `tab[i]` dans `%edx`, il faudrait le sauvegarder dans la pile avant d'appeler `malloc` et le restaurer après, ce qui ne nous ferait rien gagner.

```
    .text
/*
struct cellule_t {
    int val;
    struct cellule_t *suiv;
};
*/
    .set val, 0
    .set suiv, 4
// void decouper_asm_opt(int tab[], struct cellule_t **liste);
    .globl _decouper_asm_opt
_decouper_asm_opt:
    .set tab, 8
    .set liste, 12
    enter $8, $0
    pushl %ebx // registres non-scratch utilises
    pushl %esi
    pushl %edi
    .set sent, -8
    // tab = %edi, i = %esi, queue = %ebx
    movl tab(%ebp), %edi
    // queue = &sent;
```

```
    leal sent(%ebp), %ebx
    // i = 0
    xorl %esi, %esi
for:
    // tab[i] != 0
    cmpl $0, (%edi, %esi, 4)
    je fin_for
    // if (tab[i] > 0)
    jng else
    // queue->suiv = malloc(sizeof(struct cellule_t));
    pushl $8
    call _malloc
    addl $4, %esp
    movl %eax, suiv(%ebx)
    // queue = queue->suiv;
    movl suiv(%ebx), %ebx
    // queue->val = tab[i];
    movl (%edi, %esi, 4), %eax
    movl %eax, val(%ebx)
    // queue->suiv = NULL
    movl $0, suiv(%ebx)
else:
    // i++
    addl $1, %esi
    jmp for
fin_for:
    // queue->suiv = NULL;
    movl $0, suiv(%ebx)
    // *liste = sent.suiv;
    movl (sent+suiv)(%ebp), %eax
    movl liste(%ebp), %edx
    movl %eax, (%edx)
    popl %edi
    popl %esi
    popl %ebx
    leave
    ret
```

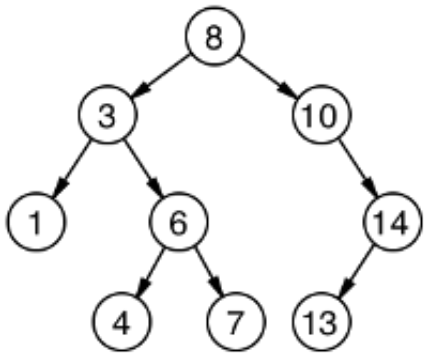
Ex. 3 : Arbres binaires de recherche *(8 pts)*

On va travailler dans cet exercice sur des arbres binaires de recherche (ABR) dont les nœuds contiennent une simple valeur entière (naturelle). On rappelle les principales propriétés de cette structure de données :

- chaque nœud de l'arbre a au plus 2 fils ;
- le sous-arbre gauche d'un nœud N donné ne contient que des nœuds dont la valeur est strictement inférieure à la valeur de N ;
- le sous-arbre droit d'un nœud N donné ne contient que des nœuds dont la valeur est strictement supérieure à la valeur de N ;
- les sous-arbres gauche et droit d'un nœud donné sont aussi des arbres binaires de

recherche.
Il vient naturellement de ses propriétés que chaque clé est unique (*i.e.* il n'existe pas plusieurs nœuds de même valeur).

La figure ci-dessous représente un arbre binaire de recherche typique :



En C, on défini simplement le type nœud d'un ABR par :

```
struct noeud_t {
    unsigned val;
    struct noeud_t *fg;
    struct noeud_t *fd;
};
```

Question 1 Ecrire en C une fonction **récursive**

```
int rechercher(unsigned v, struct noeud_t *arbre)
```

qui renvoie faux (*i.e.* 0) si la valeur `v` n'est pas présente dans l'ABR `arbre` et vrai (*i.e.* 1) sinon.

```
int rechercher(unsigned v, struct noeud_t *arbre)
{
    if (arbre == NULL) {
        return 0;
    } else if (v == arbre->val) {
        return 1;
    } else if (v < arbre->val) {
        return rechercher(v, arbre->fg);
    } else {
        return rechercher(v, arbre->fd);
    }
}
```

Question 2 Traduire cette fonction de recherche en assembleur sans chercher à l'optimiser.

```
    .text
/*
struct noeud_t {
    unsigned val;
    struct noeud_t *fg;
    struct noeud_t *fd;
};
*/
    .set val, 0
    .set fg, 4
    .set fd, 8
// int rechercher_asm(unsigned v, struct noeud_t *arbre)
    .globl _rechercher_asm
_rechercher_asm:
    .set v, 8
    .set arbre, 12
    enter $0, $0
    // if (arbre == NULL)
    cmpl $0, arbre(%ebp)
    jne L0
    // return 0;
    xorl %eax, %eax
    jmp fin
L0:
    // if (val == arbre->val)
    movl arbre(%ebp), %eax
    movl val(%eax), %eax
    cmpl %eax, v(%ebp)
    jne L1
    // return 1
    movl $1, %eax
    jmp fin
L1:
    // if (val < arbre->val)
    movl arbre(%ebp), %eax
    movl val(%eax), %eax
    cmpl %eax, v(%ebp)
    jnb L2
    // return rechercher_asm(v, arbre->fg);
    movl arbre(%ebp), %eax
    pushl fg(%eax)
    pushl v(%ebp)
    call _rechercher_asm
    addl $8, %esp
    jmp fin
L2:
    // else
    // return rechercher_asm(v, arbre->fd);
    movl arbre(%ebp), %eax
    pushl fd(%eax)
    pushl v(%ebp)
    call _rechercher_asm
    addl $8, %esp
fin:
```

```
    leave
    ret
```

Soit la fonction C suivante :

```
void arbre_vers_tab(struct noeud_t *arbre, unsigned *tab[])
{
    if (arbre != NULL) {
        arbre_vers_tab(arbre->fg, tab);
        **tab = arbre->val;
        (*tab)++;
        arbre_vers_tab(arbre->fd, tab);
    }
}
```

Question 3 Traduire cette fonction en assembleur sans chercher à l'optimiser.

```
    .text
/*
struct noeud_t {
    unsigned val;
    struct noeud_t *fg;
    struct noeud_t *fd;
};
*/
    .set val, 0
    .set fg, 4
    .set fd, 8
// void arbre_vers_tab_asm(struct noeud_t *arbre, unsigned *tab[]);
    .globl _arbre_vers_tab_asm
_arbre_vers_tab_asm:
    .set arbre, 8
    .set tab, 12
    enter $0, $0
    // if (arbre != NULL) {
    cmpl $0, arbre(%ebp)
    je fin
    // arbre_vers_tab(arbre->fg, tab);
    pushl tab(%ebp)
    movl arbre(%ebp), %eax
    pushl fg(%eax)
    call _arbre_vers_tab_asm
    addl $8, %esp
    // **tab = arbre->val;
    movl arbre(%ebp), %eax
    movl val(%eax), %eax
    movl tab(%ebp), %edx
    movl (%edx), %edx
    movl %eax, (%edx)
```

```
    // (*tab)++;
    movl tab(%ebp), %eax
    addl $4, (%eax)
    // arbre_vers_tab(arbre->fd, tab);
    pushl tab(%ebp)
    movl arbre(%ebp), %eax
    pushl fd(%eax)
    call _arbre_vers_tab_asm
    addl $8, %esp
fin:
    leave
    ret
```

Question 4 Décrire en 2 lignes maximum le contenu du tableau `tab` en sortie de cette fonction.

Le tableau contient les valeurs des nœuds présents dans l'arbre initial, triées par ordre croissant.