Architecture (avancée) des ordinateurs
L3 Informatique 2016-2017
Examen 1ère session – mardi 28 mars

Nom :
Prénom :
Numéro :

- Cet examen dure 2h. Les réponses sont à écrire directement sur le sujet (qui comporte 8 pages – dont la dernière détachable – et 5 exercices). Le barême est donné à titre indicatif.
- Le seul document autorisé est une feuille A4 recto-verso manuscrite et personnelle. Les systèmes électroniques (calculatrice, téléphone portable, etc.) sont interdits.
- Tous les codes assembleurs fournis adoptent la convention intel.

Exercice 1. (6 points) L'objectif de cet exercice est de tester votre compréhension générale des principaux thèmes abordés dans le cours.

1. Expliquer de quelle manière est codé un nombre flottant (selon la norme IEEE 754 vue en cours). Inutile de décrire les nuances entre simple, double ou quadruple précision.

2. Est-ce que 0.1 est représentable sans approximation en flottant simple précision ? Justifier.

3. Expliquer à quoi sert une mémoire cache.

4. Qu'est-ce qu'une *ligne de cache* et pourquoi la mémoire cache travaille-t-elle par lignes ?

5. Qu'est-ce qu'un *cache-miss* ?

6. Supposons que l'on dispose d'un cache complètement associatif organisé en 1024 lignes de 16 octets. Si on accède successivement à toutes les cases d'un tableau de 512 entiers (codés chacun sur 4 octets), combien de cache-misses va-t-on avoir ? Détailler le calcul.

7. Expliquer pourquoi, lorsque l'on parcourt un très grand tableau (de taille 10^8, par exemple), il est préférable de faire les accès au cases de façon séquentielle plutôt qu'aléatoire.

8. En pratique, pour un tableau de taille 10000, on n'observe pas de différence significative sur le temps d'exécution entre un parcours séquentiel et un parcours aléatoire. Pourquoi ?

9. Expliquer à quoi sert un pipeline dans un processeur.

10. Qu'est-ce qu'une *bulle* lors de l'exécution d'un programme ?

11. Pourquoi est-ce que limiter le nombre de sauts conditionnels peut améliorer le temps d'exécution d'un programme ?

12. Expliquer le rôle d'un prédicteur de branchement.

13. Quelle est la différence entre une instruction *vectorielle* et une instruction *scalaire* ?

14. L'option `-O1` incite le compilateur `gcc` à produire un code plus court. Pourquoi est-ce une amélioration ?

Exercice 2. (5 points) Cet exercice examine le fonctionnement de la mémoire cache au travers du calcul de multiplication entre matrice et vecteur. Étant donné la matrice M et le vecteur V

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \quad \text{et} \quad V = (x, y, z),$$

on s'intéresse aux opérations $M \cdot V$ et $V \cdot M$ définies par

$$M{\cdot}V = (ax{+}by{+}cz, dx{+}ey{+}fz, gx{+}hy{+}iz) \quad \text{et} \quad V{\cdot}M = (ax{+}dy{+}gz, bx{+}ey{+}hz, cx{+}fy{+}iz).$$

Dans tous l'exercice, on supposera que les vecteurs `int* V` et `int* RES` de taille n et la matrice `int** M` de taille $n \times n$ on été initialisés et sont accessibles globalement.

1. Donner un exemple de M et V pour $n = 2$ pour lesquels $V \cdot M \neq M \cdot V$.

2. Le résultat de la multiplication d'un vecteur `V` de taille n par une matrice `M` taille $n \times n$ est un vecteur qui peut être calculé en C de la façon suivante (dans `RES`) :

```
1  void mult_vect_matrix(int n){
2    int i,j;
3    for (i = 0; i < n; ++i){
4        int s = 0;
5        for (j = 0; j < n; ++j)
6            s += V[j] * M[j][i];
7        RES[i] = s;
8    }}
```

   On suppose que `i`, `j`, `n` et `s` sont stockées dans des registres. Expliquer comment se comporte la mémoire cache lorsque cette fonction est appelée avec n très grand (relativement au cache).

3. On souhaite comparer l'efficacité de ce programme à celui qui calcule la multiplication d'une matrice `M` taille $n \times n$ par un vecteur `V` de taille n. Le résultat est également un vecteur qui peut être calculé en C de la façon suivante (dans `RES`) :

```
1  void mult_matrix_vect(int n){
2    int i,j,s;
3    for (i = 0; i < n; ++i){
4        s = 0;
5        for (j = 0; j < n; ++j)
6            s += M[i][j] * V[j];
7        RES[i] = s;
8    }}
```

On suppose que `i`, `j`, `n` et `s` sont stockées dans des registres. Expliquer comment se comporte la mémoire cache lorsque cette fonction est appelée avec n très grand (relativement au cache).

4. Quel résultat s'attend-on à avoir si l'on compare les temps d'exécution de ces deux programmes pour la même valeur de n ?

5. Pour en avoir le cœur net, on exécute le test suivant :

```
unsigned long nb = clock();
mult_matrix_vect(64);
printf("%.10f sec\n", (clock()-nb)/(float)CLOCKS_PER_SEC);

nb = clock();
mult_vect_matrix(64);
printf("%.10f sec\n", (clock()-nb)/(float)CLOCKS_PER_SEC);
```

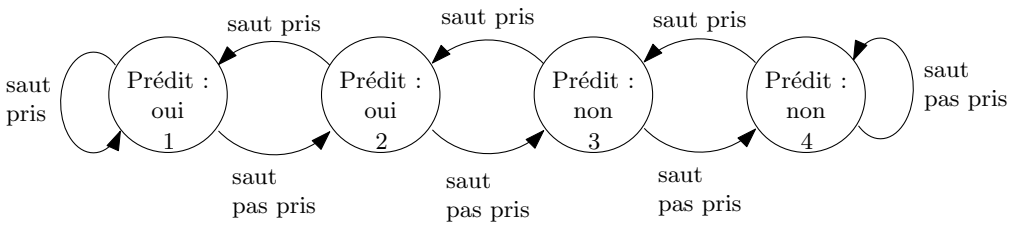
et on observe le résultat suivant :

```
0.0000170000 sec
0.0000150000 sec
```

Expliquer ce qui ne va pas dans ce test.

6. Proposer un algorithme qui réalise la même opération que `mult_vect_matrix` mais utilise le cache plus efficacement.

Exercice 3. (3 points) On s'intéresse dans cet exercice au comportement d'un processeur dont le pipeline utilise un prédicteur de branchement à 4 états.



1. On considère le programme suivant et sa traduction en assembleur :

```
1  int compte(int n, int* tab){
2    int s;
3
4    for (int i = 0; i < n; ++i){
5        if (tab[i]<50){
6            s += 1;
7        }
8    }
9    return s;
10 }
```

```
1  compte: mov     rdx, rsi
2          lea     ecx, [rdi-1]
3          lea     rsi, [rsi+4+rcx*4]
4  .L1:    cmp     DWORD PTR [rdx], 50
5          jge     .L2
6          inc     eax
7  .L2:    add     rdx, 4
8          cmp     rdx, rsi
9          jne     .L1
10         ret
```

Indiquer les numéros de ligne des instructions assembleur de saut conditionnel et, pour chacune d'entre elle, le numéro de ligne de l'instruction C correspondante.

2. On suppose que le processeur consacre un prédicteur de branchement 4 états uniquement à l'instruction assembleur de la ligne 5 (`jge .L2`). On appelle la fonction sur un tableau de 20 cases dont les valeurs provoquent le résultats suivants :

indice dans `tab`	0	1	2	3	4	5	6	7	8	9
saut pris ?	oui	oui	oui	non	non	non	non	non	oui	oui
indice dans `tab`	10	11	12	13	14	15	16	17	18	19
saut pris ?	oui	oui	non	non	oui	oui	non	oui	non	non

   (a) On suppose que le prédicteur est initialement dans l'état 1. Indiquer les indices du tableau pour lesquels le prédicteur de branchement fait une erreur de prédiction.

   (b) Même question en supposant que le prédicteur est initialement dans l'état 4.

Exercice 4. (4 points) On cherche maintenant à comprendre comment est optimisé le code assembleur des deux fonctions suivantes (qui calculent la même chose).

```
1  int f1 (int n){
2      int s = 0, p = 0;
3      for(int i = 1; i <= n; i++){
4        for(int j = 1; j <= n; j++)
5          s+=i;
6        p*=i;
7      }
8      return s+p; }
```

```
1  int f2 (int n){
2      int s = 0, p = 0;
3      for(int i = 1; i <= n; i++){
4        s+=i*n;
5        p*=i;
6      }
7      return s+p;
8  }
```

Les Figures 1 à 3 (dernière page) donnent le code assembleur de `f1` et `f2` pour différentes optimisations de `gcc`.

2. Quelle est la **principale** optimisation effectuée entre le code obtenu sans optimisation et celui obtenu avec `gcc -O1` ? En quoi fait-elle gagner du temps ?

3. Si vous observez le calcul de `p`, une autre optimisation sensible a été effectuée entre le code obtenu sans optimisation et celui obtenu avec `gcc -O1` ? Quelle est cette optimisation et comment est-elle possible ?

4. Quelle est la principale optimisation effectuée entre le code obtenu pour `f1` avec `gcc -O1` et celui obtenu avec `gcc -O2` ? Pourquoi est-ce possible ?

5. Si on compile le même code avec `gcc -O3`, seule `f2` est vectorisée. Pensez-vous qu'il est possible de vectoriser `f1` également ? Justifier.

Exercice 5. (3 points) On considère la fonction suivante, qui réalise un tri-fusion.

```
1  void merge_sort(int* A, int* aux, int deb, int fin){
2    if(fin - deb < 2)
3      return;
4
5    int mil = (fin+deb)/2;
6    merge_sort(A, aux, deb, mil);
7    merge_sort(A, aux, mil, fin);
8
9    int i = deb, j = mil, t = -1, k = deb;
10   do {
11     if (A[i] <= A[j]){
12       aux[k] = A[i];
13       i++;
14     }
15     else {
16         aux[k] = A[j];
17         j++;
18       }
19     k++;
20   }
21   while (i< mil && j<fin);
22
23   if (i == mil) {t = j;} else {t = i;}
24
25   while (k<fin) {
26       aux[k] = A[t];
27       k++;
28       t++;
29     }
30   for (int k = deb; k < fin; k++)
31     A[k] = aux[k];
32 }
```

1. Analyser, en justifiant, le comportement de cette fonction du point de vue du cache.

2. Analyser, en justifiant, le comportement de cette fonction du point de vue de la prédiction de branchement.

```
f1:
.LFB2:
  push  rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-20], edi
  mov DWORD PTR [rbp-12], 0
  mov DWORD PTR [rbp-16], 0
  mov DWORD PTR [rbp-4], 1
  jmp .L2
.L5:
  mov DWORD PTR [rbp-8], 1
  jmp .L3
.L4:
  mov eax, DWORD PTR [rbp-4]
  add DWORD PTR [rbp-12], eax
  add DWORD PTR [rbp-8], 1
.L3:
  mov eax, DWORD PTR [rbp-8]
  cmp eax, DWORD PTR [rbp-20]
  jle .L4
  mov eax, DWORD PTR [rbp-16]
  imul  eax, DWORD PTR [rbp-4]
  mov DWORD PTR [rbp-16], eax
  add DWORD PTR [rbp-4], 1
.L2:
  mov eax, DWORD PTR [rbp-4]
  cmp eax, DWORD PTR [rbp-20]
  jle .L5
  mov edx, DWORD PTR [rbp-12]
  mov eax, DWORD PTR [rbp-16]
  add eax, edx
  pop rbp
  ret

f2:
  push  rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-20], edi
  mov DWORD PTR [rbp-8], 0
  mov DWORD PTR [rbp-12], 0
  mov DWORD PTR [rbp-4], 1
  jmp .L8
.L9:
  mov eax, DWORD PTR [rbp-4]
  imul  eax, DWORD PTR [rbp-20]
  add DWORD PTR [rbp-8], eax
  mov eax, DWORD PTR [rbp-12]
  imul  eax, DWORD PTR [rbp-4]
  mov DWORD PTR [rbp-12], eax
  add DWORD PTR [rbp-4], 1
.L8:
  mov eax, DWORD PTR [rbp-4]
  cmp eax, DWORD PTR [rbp-20]
  jle .L9
  mov edx, DWORD PTR [rbp-8]
  mov eax, DWORD PTR [rbp-12]
  add eax, edx
  pop rbp
  ret
```

FIGURE 1 – Code assembleur de l'exercice 4 produit avec `gcc` sans optimisation.

```
1  f1:
2    mov ecx, 1
3    mov eax, 0
4    test  edi, edi
5    jg  .L7
6    rep ret
7  .L4:
8    add eax, ecx
9    add edx, 1
10   cmp edi, edx
11   jge .L4
12   add ecx, 1
13   cmp edi, ecx
14   jl  .L3
15 .L7:
16   mov edx, 1
17   jmp .L4
18 .L3:
19   rep ret
20
21 f2:
22   test  edi, edi
23   jle .L11
24   mov esi, edi
25   mov ecx, edi
26   mov eax, 0
27   mov edx, 1
28 .L10:
29   add eax, ecx
30   add edx, 1
31   add ecx, esi
32   cmp edi, edx
33   jge .L10
34   rep ret
35 .L11:
36   mov eax, 0
37   ret
```

FIGURE 2 – Code assembleur de l'exercice 4 produit avec `gcc -O1`

```
1  f1:
2    test  edi, edi
3    jle .L4
4    lea r8d, [rdi-1]
5    lea esi, [rdi+1]
6    mov edx, 1
7    xor eax, eax
8    mov ecx, r8d
9    .p2align 4,,10
10   .p2align 3
11 .L3:
12   add eax, edx
13   add edx, 1
14   add eax, ecx
15   add ecx, r8d
16   cmp edx, esi
17   jne .L3
18   rep ret
19 .L4:
20   xor eax, eax
21   ret
22
23 f2:
24   test  edi, edi
25   jle .L10
26   lea esi, [rdi+1]
27   mov ecx, edi
28   xor eax, eax
29   mov edx, 1
30   .p2align 4,,10
31   .p2align 3
32 .L9:
33   add edx, 1
34   add eax, ecx
35   add ecx, edi
36   cmp edx, esi
37   jne .L9
38   rep ret
39 .L10:
40   xor eax, eax
41   ret
42
```

FIGURE 3 – Code assembleur de l'exercice 4 produit avec `gcc -O2`