Architecture (avancée) des ordinateurs
L3 Informatique 2018-2019
Examen 1ère session – vendredi 31 mai 2019

Nom :
Prénom :
Numéro :

- Cet examen dure 2h. Les réponses sont à écrire directement sur le sujet. Le sujet comporte 8 pages plus une page de convention ("pseudocode vectorisé") détachable.
- Il y a 4 exercices et le barème 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.

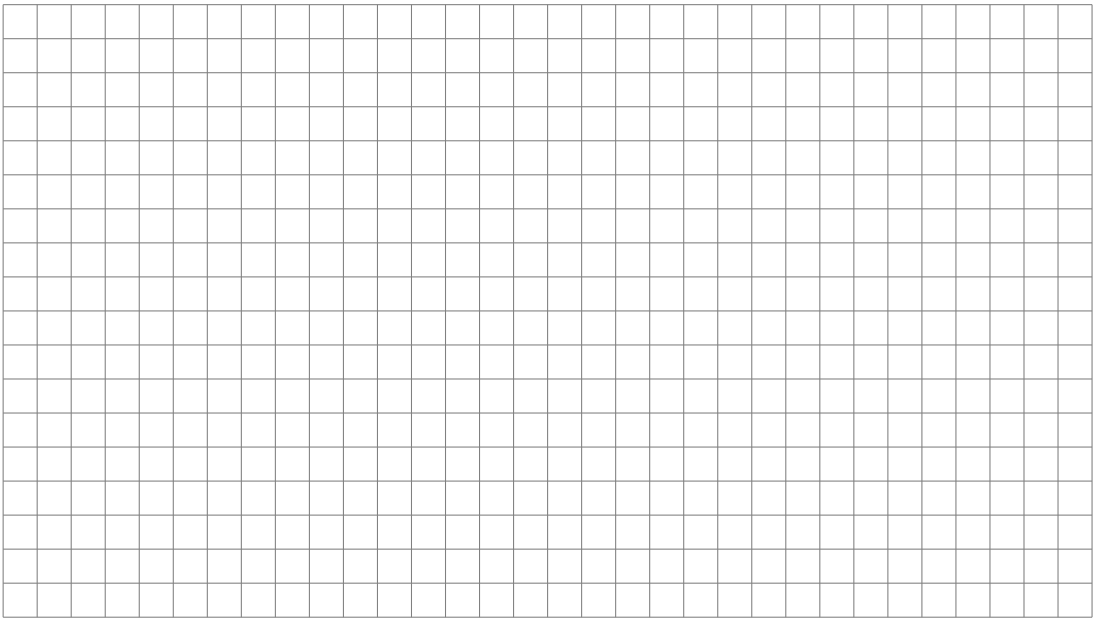
Exercice 1. (6 points) Supposons que l'on dispose d'un cache de 128 octets au total.

1. Dans un premier temps, on considère que **les lignes de cache font 8 octets et qu'il est non associatif en direct mapping**.

   a. Combien y a-t-il de lignes dans ce cache ? Justifier (calcul).

   b. Les adresses en mémoire principale sont codées sur 16 bits. Sachant qu'une adresse correspond à un octet, combien d'octets différents peuvent être stockés en mémoire principale ? Combien d'octets différents correspondent à une même case en mémoire cache ?

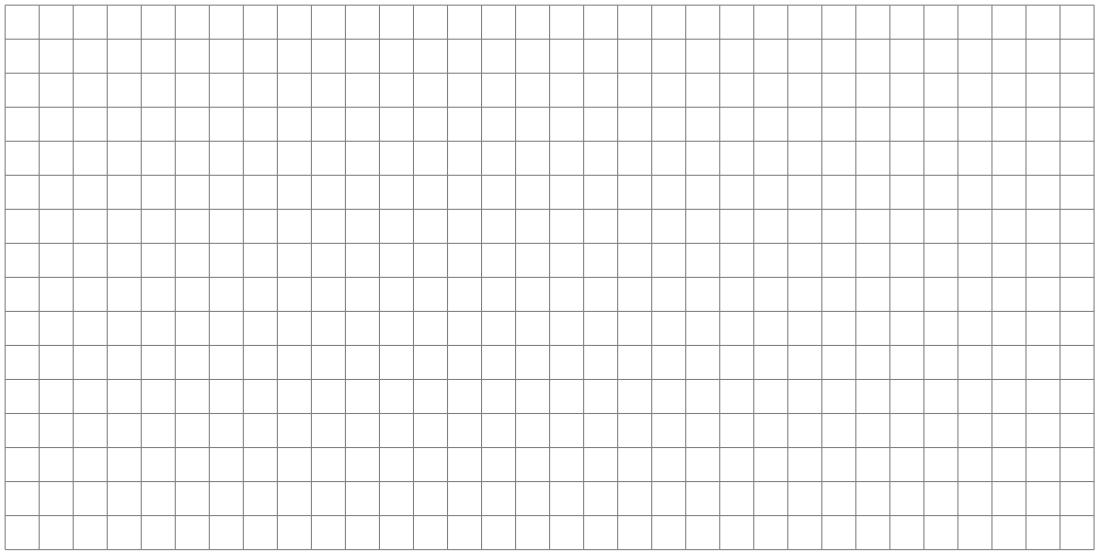
   c. Dessiner l'organisation de ce cache et indiquer (par la lettre A) où va être stocké l'octet d'adresse `0xFFFF`. Justifier (indiquer le calcul d'adresse).



   d. Indiquer sur le même dessin (par la lettre B) où va être stocké l'octet d'adresse `0x4B15`. Justifier. À quelle plage d'adresses correspondent les octets qui seront stockés dans la même ligne de cache ?

2. Dans cette deuxième partie, on considère un cache de la même taille avec **les lignes de cache de 8 octets, 4-associatif et toujours en direct mapping**. On rappelle que cela signifie que 4 lignes de cache différentes sont stockées à la même adresse en cache.
   a. Comment est calculée l'adresse d'un octet dans un tel cache, à partir de son adresse X en mémoire principale ? Justifier.

   b. Dessiner l'organisation de ce cache et indiquer (par la lettre A) les emplacements possibles où vont être stocké l'octet d'adresse `0xFFFF`. Indiquer également (par la lettre B) les emplacements possibles pour l'octet d'adresse `0x4B15`. Justifier.



   c. On suppose que l'on accède dans cet ordre aux cases d'adresses suivantes. Indiquer (en justifiant) pour chacun des accès, s'il provoque ou non un cache miss dans ce modèle de cache, avec un algo de type LRU qui remplace la ligne utilisée le moins récemment.

1. `0x0000`	3. `0x0100`	5. `0x0200`	7. `0x0400`	9. `0x0101`
2. `0x0001`	4. `0x0002`	6. `0x0300`	8. `0x0004`	10. `0x0301`

Exercice 2. (3 points) On souhaite mesurer le nombre de cycles que prend la multiplication d'une case mémoire par un entier. Voici le code proposé pour faire cela :

```
1  void print_mult_cycles(int* tab, long size) {
2    unsigned int ui; register long tic, toc, n = size;
3
4    tic = __rdtscp(&ui);
5    for (int j = 0; j < n; ++j){
6      tab[j] = 64;
7    }
8    toc = __rdtscp(&ui);
9
10   double time1 = toc - tic;
11
12   tic = __rdtscp(&ui);
13   for (int j = 0; j < n; ++j){
14     tab[j] = tab[j] * 64;
15   }
16   toc = __rdtscp(&ui);
17
18   double time2 = toc - tic;
19
20   printf("%.2f cycles", (time2 - time1) / n);
21 }
```

1. À quoi sert la mesure du nombre de cycles de la première boucle ?

2. Donner au moins 3 raisons (distinctes) pour lesquelles cette fonction ne permet pas de déterminer précisément combien de cycles prend la multiplication d'une case mémoire par un entier. Si c'est possible, proposer une façon de rendre la mesure plus précise.

   a.

   b.

   c.

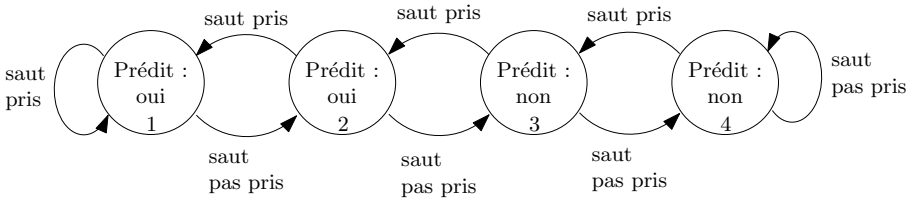
Exercice 3. (4 points) On s'intéresse au comportement des prédicteurs de branchements lors de l'exécution de la fonction `compte` qui prend en paramètre deux tableaux `t1` et `t2` de taille `n`.

```
1  int compte(int n, int* t1, int* t2){
2    int s = 0;
3    for (int i = 0; i < n; ++i){
4      if (t1[i] == 'A' || t1[i] == 'B')
5        s += 1;
6      if (t2[i] == 'A')
7        s += 1;
8    }
9    return s;
10 }
```

Dans tout l'exercice, on utilise les deux tableaux de caractères suivants :

X :	D	A	B	A	D	C	A	A	B	C	A	D	D	B	D	C
Y :	B	A	D	C	B	A	D	A	C	D	C	A	B	B	A	A

Et on suppose que les prédicteurs de branchement utilisés correspondent à l'automate suivant (un saut est pris si la condition est évaluée à VRAI) :



1. Pour cette question, on considère que **chaque branchement** (lignes 4 et 6) **a son propre prédicteur** (on dit qu'il est "local"). Indiquez, pour chaque branchement, la suite des états pris par le prédicteur (y compris l'état initial, qui est fixé à 1) lors de l'appel à la fonction `compte(16,X,X)` et indiquez les erreurs de prédictions par une étoile entre deux états. Par exemple, pour le branchement de la ligne 4, on commence par 1*211...

   a. Branchement de la ligne 4 :

   b. Branchement de la ligne 6 :

2. Désormais, on utilise **un seul prédicteur pour les deux branchements** (on dit qu'il est "global"). Indiquez, **pour chaque appel de la fonction** `compte`, la suite des états pris par le prédicteur de branchement et indiquez les erreurs de prédictions par une étoile entre deux états (pour chaque appel, le prédicteur est initialement dans l'état 1).

   a. Appel de la fonction `compte(16,X,X)` :

   b. Appel de la fonction `compte(16,X,Y)` :

3. À votre avis, est-ce que l'un des deux prédicteurs (local ou global) est meilleur que l'autre ? Argumentez.

Exercice 4. (8 points) On considère le code des deux fonctions suivantes écrites en C, ainsi que la fonction `main`. La variable `size` est une variable globale déclarée en début de programme. Le résultat de l'exécution du programme est donné ensuite, pour des compilations avec `gcc` sans optimisation (`-O0`), avec `-O1` ou `-O3`.

```c
int mysteryA(int* tabX, int* tabY){
  register int i, a, b, myst = 0;
  for(i = 1; i < size; i++){
    a = tabX[i];
    b = tabY[a];
    myst+=b;
  }

  return myst;
}
```

```c
int mysteryB(int* tabX, int* tabY){
  register int i, a, b, myst = 0;
  for(i = 1; i < size; i++){
    a = tabX[i];
    b = tabY[i];
    if (a<b)
      myst+=a;
    else myst+=b;
  }
  return myst; }
```

```c
int main(){
  size = atoi(argv[1]); // l'argument de la ligne de commande converti en int
  int* tab1; if((tab1 = (int*)malloc(size*sizeof(int))) == NULL) exit(1);
  int* tab2; if((tab2 = (int*)malloc(size*sizeof(int))) == NULL) exit(1);
  int* tab3; if((tab3 = (int*)malloc(size*sizeof(int))) == NULL) exit(1);

  for(int i=0; i<size; i++){
    tab1[i] = rand()%size;
    tab2[i] = size/2;
    tab3[i] = i;
  }
  printf("mysteryA - tab1 - tab2 "); print_timing(tab1, tab2, mysteryA);
  printf("mysteryA - tab3 - tab2 "); print_timing(tab3, tab2, mysteryA); printf("\n");
  printf("mysteryB - tab1 - tab2 "); print_timing(tab1, tab2, mysteryB);
  printf("mysteryB - tab3 - tab2 "); print_timing(tab3, tab2, mysteryB);
}
```

Compilation avec gcc -O0 :

```
mysteryA - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 1.828887
mysteryA - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.190705

mysteryB - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.636256
mysteryB - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.213098
```

Compilation avec gcc -O1 :

```
mysteryA - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 1.818390
mysteryA - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.056819

mysteryB - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.056456
mysteryB - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.054223
```

Compilation avec gcc -O3 :

```
mysteryA - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 1.581774
mysteryA - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.045305

mysteryB - tab1 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.036185
mysteryB - tab3 - tab2 temps moyen sur 10 exécutions (size = 80000000) : 0.037228
```

On suppose que la fonction `print_timing(tabX, tabY, f)` mesure correctement (comme en TP) le temps d'exécution de la fonction `f` qui prend en paramètre deux tableaux d'entiers `tabX` et `tabY` de taille `size`. Dans tous les exemples ci-dessus, **la valeur de `size` est** 80 000 000.

1. Proposer une explication détaillée et justifiée de la différence de temps d'exécution observée entre `mysteryA(tab1, tab2)` et `mysteryA(tab3, tab2)` compilées avec `-O0` (lignes 1 et 2).

2. Proposer une explication détaillée et justifiée de la différence de temps d'exécution observée entre `mysteryB(tab1, tab2)` et `mysteryB(tab3, tab2)` compilées avec `-O0` (lignes 4 et 5).

3. Pourquoi n'observe-t-on plus de différence de temps d'exécution entre `mysteryB(tab1, tab2)` et `mysteryB(tab3, tab2)` compilées avec `-O1` (lignes 9 et 10) ? Proposer une explication argumentée.

4. Est-ce que c'est cohérent avec le fait que le temps d'exécution de `mysteryA(tab3, tab2)` soit à peu près le même que pour `mysteryB(tab1, tab2)` et `mysteryB(tab3, tab2)` (toujours en `-O1` : lignes 7, 9 et 10) ? Justifier.

5. Qu'est-ce qui peut justifier le gain en temps d'exécution observé pour `mysteryB` entre les compilations avec `-O1` et `-O3` (lignes 9 et 10 vs. 14 et 15) ? Justifier.

6. Est-ce que c'est cohérent avec le fait que le temps d'exécution de `mysteryA(tab3, tab2)` soit plus élevé que pour `mysteryB(tab1, tab2)` et `mysteryB(tab3, tab2)` (toujours en `-O3` : lignes 12, 14 et 15) ? Justifier.

7. Proposer une expérience utilisant le même programme que ci-dessus (sans modification) et qui permet de déterminer la taille des différents niveaux de cache.

8. Écrire l'algorithme vectorisé correspondant à `mysteryB`. Le pseudo-code utilisera les conventions de la page 9 pour les instructions vectorisées.

9. On suppose maintenant que l'**on ne dispose pas de l'instruction de minimum vectorisé**. Ré-écrire le corps de la boucle de l'algorithme vectorisé correspondant à `mysteryB`.

Pseudo code vectorisé. On utilise des vecteurs pouvant contenir 4 entiers (qu'on appelle des *composantes*). Chaque composante est traitée comme un `int`. Il est inutile de déclarer les variables vecteurs. On autorise les opérations suivantes sur les vecteurs.

Pour le transfert :

- initialiser un vecteur (ex : `u = 1,1,1,1` et `v = 0,33,42,806`)
- copier un vecteur dans un autre (ex : `w = v`)
- charger un vecteur depuis 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `w = (char) TAB[i:i+3]` indique que l'on charge 4 cases de la taille d'un `char` dans `w` depuis l'adresse `TAB[i]`)
- charger un vecteur dans 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `TAB[0:3] = (char) w` indique que l'on place les 4 composantes de `w` dans 4 cases mémoire consécutives de la taille d'un `char` à l'adresse `TAB`)

Pour le calcul :

- additionner (ou soustraire, multiplier, diviser) deux vecteurs composante par composante (ex : `w = u + v`; avec les valeurs précédentes, on obtient que `w` vaut désormais `1,34,43,807`)
- faire des opération logiques bit à bit sur les vecteurs (ex : `z = u AND v`, `z = u OR v` ou `z = NOT u`, ...)
- comparer deux vecteurs composante par composante (ex : `z = u < v` signifie que si la i^{eme} composante de `u` est strictement inférieure à la i^{eme} composante de `v`, alors la i^{eme} composante de `z` vaut `111...111` et sinon elle vaut `000...000` (en binaire). Avec les valeurs précédentes, on obtient que `z` vaut `0...0,1...1,1...1,1...1` (ici aussi en binaire).
- calculer le minimum de deux vecteurs composante par composante (ex : `z = min(u ,v)` signifie que si la i^{eme} composante de `u` est strictement inférieure à la i^{eme} composante de `v`, alors la i^{eme} composante de `z` vaut la i^{eme} composante de `u` et sinon elle vaut la i^{eme} composante de `v`. Par exemple, avec `u = 1,100,1,27` et `v = 0,33,42,806`, si on fait `z = min(u ,v)` on obtient que `z` vaut `0,33,1,27`.

Pour la conversion d'un vecteur en `int` :

- faire la somme des 4 composantes d'un vecteur (ex : `int i = sum_comp(w)`; avec les valeurs précédentes, on obtient que `i` vaut `885`)