EXERCICE A: Assembleur MIPS32 Numéro :

On cherche à écrire en assembleur MIPS un programme de validation de grille de sudoku 4x4, qui affiche « grille valide » sur le terminal si la grille est valide et « grille invalide » dans le cas contraire.

Par exemple, sur la grille 4x4 suivante : 1 2 4 3
3 4 2 1
4 1 3 2
2 3 1 4

Le programme ici doit afficher « grille valide » car chaque ligne de cette grille contient une occurrence des chiffres 1,2,3 et 4, chaque colonne contient une occurrence des chiffres 1,2,3 et 4, et chaque quadrant (les quatre chiffres au Nord-Ouest, Nord-Est, Sud-Ouest, Sud-Est) également.

Pour vérifier qu'une ligne, qu'une colonne ou qu'un quadrant est valide, il suffit de faire la somme des éléments et de vérifier que celle-ci vaut bien 10=1+2+3+4.

A1 Ecrire les directives assembleur MIPS32 permettant de déclarer la grille précédente dans le segment data. On utilisera l'étiquette « sudoku » pour désigner l'adresse du premier élément du tableau. On suppose que chaque case n'occupe qu'un seul octet.

```
    .data
sudoku:
    .byte   1, 2, 4, 3
    .byte   3, 4, 2, 1
    .byte  4, 1, 3, 2
    .byte   2, 3, 1, 4
```

Le code de la fonction verif_ligne qui vérifie si une ligne est correcte est :

```
1.    verif_ligne:
2.       li       $2,  10
3.       lb       $11, 0($4)
4.       subu     $2,  $2,   $11
5.       lb       $11, 1($4)
6.       subu     $2,  $2,   $11
7.       lb       $11, 2($4)
8.       subu     $2,  $2,   $11
9.       lb       $11, 3($4)
10.      subu     $2,  $2,   $11
11.      jr       $31
```

A2 Combien d'argument a la fonction verif_ligne? Quelle est la valeur de retour de cette fonction ? On ne sauve pas $11 dans la pile, est-ce une erreur ? Pourquoi n'est-il pas nécessaire de sauver $31 ?

- La fonction a 1 paramètre qui est l'adresse d'une ligne.
- Elle rend 0 si la ligne est valide et ≠ 0 sinon.
- $11 est un registre temporaire qu'on ne doit donc pas sauver.
- une fonction terminale (qui n'appelle pas de fonction) ne modifie pas $31

A3 On souhaite modifier la fonction `verif_ligne` pour la transformer en fonction `verif_colonne` qui teste qu'une colonne est valide ? Que prend-elle en argument ? Indiquez les numéros de lignes de code modifiées et leur nouveau contenu.

1 - La fonction prend en paramètre l'adresse du premier octet d'une colonne

2 - Il faut modifier le déplacement dans les instructions lb

5. lb $11, 4($4)
7. lb $11, 8($4)
9. lb $11, 12($4)

A4 Que faut-il modifier dans la fonction `verif_ligne` pour la transformer en fonction `verif_quadrant` qui teste qu'un quadrant est valide. Que prend-elle en argument ? Indiquez les numéros de lignes de code modifiées et leur nouveau contenu.

1 - La fonction prend en paramètre l'adresse du premier octet d'une colonne

2 - Il faut modifier le déplacement dans les instructions lb

5. lb $11, 1($4)
7. lb $11, 4($4)
9. lb $11, 5($4)

A5 Ecrivez en assembleur le code de la fonction C qui vérifie toutes les lignes d'une grille.

```
int verif_lignes(char * grille) { int res = 0;
                                  res = res + verif_ligne(grille);
                                  res = res + verif_ligne(grille+4);
                                  res = res + verif_ligne(grille+8);
                                  res = res + verif_ligne(grille+16);
                                  return res;}
```

```
verif_lignes:
1.       addiu   $29, $29, -8
2.       sw      $31, 4($29)
3.       sw      $16, 0($29)
4.       sw      $4,  8($29)
5.       li      $16, 0
6.       jal     verif_ligne
7.       add     $16, $16, $2
8.       lw      $4,  8($29)
9.       addiu   $4,  $4, 4
10.      jal     verif_ligne
11.      add     $16, $16, $2
12.      lw      $4,  8($29)
13.      addiu   $4,  $4, 8
14.      jal     verif_ligne
15.      add     $16, $16, $2
16.      lw      $4,  8($29)
17.      addiu   $4,  $4, 12
18.      jal     verif_ligne
19.      add     $2,  $16, $2
20.      lw      $16, 0($29)
21.      lw      $31, 4($29)
22.      jr      $31
```

Exercice B : Mémoires Cache(5points) **Numéro :**

On considère un cache de données de premier niveau write-through, à correspondance directe, d'une capacité totale de 1Ko. La ligne de cache a une largeur de 16 octets (4 mots de 32 bits). Les adresses sont sur 32 bits, et chaque adresse référence un octet en mémoire. Le but de l'exercice est d'analyser le remplissage du cache de données puis d'estimer le nombre de cycles nécessaires à l'exécution d'un programme. On suppose que le cache de données est initialement vide. On considère la fonction `tabdiff` suivante :

```
int32_t X[512];
...
void tabdiff(int32_t X[512]) {
   register int32_t i;
   for (i = 1; i < 512; i++) {
      X[i - 1] = X[i] - X[i - 1];
   }
}
```

Dans tout l'exercice, on suppose que le tableau X est placé à l'adresse 0x10010000, et qu'il est passé à la fonction `tabdiff`.

Rappels : le mot clé "register" est une directive passée au compilateur pour qu'il place la variable **`i`** dans un registre plutôt que sur la pile ; la variable **`i`** reste donc en registre durant toute la durée d'exécution de la fonction et ni sa lecture ni son écriture ne provoquent d'accès au cache de données. Un `int32_t` est codé sur 4 octets.

B1 (0.5 point). Donner le nombre de bits des champs offset, index, et tag d'une adresse.

Offset : 4 bits

Index : 6 bits

Etiquette : 22 bits

B2 (1 point). Quels sont les éléments de X qui peuvent occuper les cases d'index 0 et 17 ?

Tableau X aligné sur la taille du cache.

Case d'index 0 : X[0], X[1], X[2], X[3], X[256], X[257], X[258], X[259]

Case d'index 17 : X[68], X[69], X[70], X[71], X[324], X[325], X[326], X[327]

B3 (0.5 point). Donner dans l'ordre les 6 premiers éléments de X lus en cache par la fonction `tabdiff`.

X[1], X[0], X[2], X[1], X[3], X[2]

B4 (1 point). Représenter dans le schéma ci-dessous les cases valides du cache après 2 itérations (1 case contient 1 ligne), en précisant les indexes (il y a au plus 3 cases valides).

Index	adresse	Mot 3	Mot 2	Mot 1	Mot 0
0	0x10010000	X[3]	X[2]	X[1]	X[0]

B5 (1 point). Calculer, en le justifiant, le nombre de miss de données rencontrés lors de l'exécution de cette fonction.

On charge tous les éléments du tableau sans conflit. Il y a 4 éléments de X par ligne et X est aligné sur la taille du cache. Il y a donc 512 / 4 = 128 miss de données.

B6 (0.5 point). Donner le taux de miss de données de cette fonction.

Il y a 511 * 2 = 1022 accès.

Le taux de miss est donc de 128 / 1022 = 12.5%

B7 (0.5 point). La compilation de ce code a produit une boucle contenant 7 instructions, qui mettrait donc 7 cycles à s'exécuter si le système mémoire était parfait (jamais de miss). Calculer le nombre de cycles moyen par itération si un miss de données coûte 9 cycles, et si on néglige l'effet des miss sur le cache d'instructions.

Nombre de cycles total = 511 * 7 + 128 * 9 = 4729
Il y a 511 itérations, soit une moyenne de 4729 / 511 = 9.25 cycles/itération