**EXERCICE A Assembleur (5 points)** **Numéro :**

On cherche à écrire en assembleur un code calculant la partie entière de la racine carrée d'un nombre positif. Ce code est composé de deux fonctions. La première **sqr()** se contente d'invoquet la fonction récursive **sqrr().**

```
int sqrr (int n, int i)
{
    if (n < i) return 0;
    return 1 + sqrr ((n - i), (i + 2));
}

int sqr (int n)
{
    return sqrr (n, 1);
}
```

Le code de la fonction **main()** utilisée par xspim pour l'invocation de la fonction sqr est le suivant:

```
        .data
msg:    .asciiz "\nEntrez un nombre: "

        .text
        .globl main

main:   addiu   $29, $29, -8
        sw      $31, 4 ($29)

main1:  la      $4, msg
        li      $2, 4
        syscall
        li      $2, 5
        syscall

        beq     $2, $0, main2

        ..... appel de la fonction sqr()

        add     $4, $0, $2
        li      $2, 1
        syscall
        j       main1

main2:  lw      $31, 4 ($29)
        addiu   $29, $29, 8
        jr      $31
```

**A1 (1 point)** Écrivez ici les instructions assembleur permettant l'appel de la fonction **sqr()**.

```
        add     $4, $0, $2
        jal     sqr
```

**A2 (1 point)** quoi sert la ligne **`.globl main`**

la directive .globl informe l'assembleur ou le compilateur que l'étiquette main est globale afin qu'elle soit utilisée par l'éditeur de lien





**A3 (1point)** Pour quelle raison ne peux-t-on pas écrire le code suivant alors que c'est fonctionnellement correct ?

```
addiu   $29, $29, 8
lw      $31, -4 ($29)
jr      $31
```

Il n'est pas autorisé d'écrire sous le pointeur de pile car en cas d'interruption les données s'y trouvant seront effacées par le système d'exploitation

**A4 (1 point)** Écrire le code de la fonction **sqr**. Vous devez justifier la valeur du déplacement du pointeur de pile en indiquant les valeurs de (na, nv, nr) et l'usage de la pile.

```
sqr:
        # na = 2 puisque sqrr à 2 args, nv = 0 pas de variable, nr = 1 pour $31
        addiu   $29, $29, -12
        sw      $31, 8($29)

        li      $5,  1
        jal     sqrr

        lw      $31, 8($29)
        addiu   $29, $29, 12
        jr      $31
```

**A5 (1 point)** Ecrire le code de la fonction **sqrr**

```
sqrr:
        # na = 2 puisque sqrr à 2 args, nv = 0 pas de variable, nr = 1 pour $31
        addiu   $29, $29, -12
        sw      $31, 8($29)

        li      $2,  0
        slt     $8,  $4,  $5
        bne     $8,  $0,  sqrr_fin

        subu    $4,  $4,  $5
        addiu   $5,  $5,  2
        jal     sqrr
        addiu   $2, $2, 1

sqrr_fin:
        lw      $31, 8($29)
        addiu   $29, $29, 12
        jr      $31
```

**Exercice B : Mémoires Cache** **Numéro :**

On considère un cache de données de premier niveau, write-through, à correspondance directe, contenant 512 lignes, et possédant une capacité totale de 32Ko. Les adresses sont sur 32 bits.
Le but de l'exercice est d'analyser le remplissage d'un cache de données de premier niveau puis d'estimer le nombre de cycles nécessaires à l'exécution d'un programme. On suppose que ce cache est initialement vide. On considère la fonction my_sum suivante (on remarquera que 768 = 512+256)

```
int32_t X[768][16];
...
int32_t my_sum(int32_t X[768][16]) {
   register int32_t i, j;
   register int32_t sum = 0;
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 768; j++) { sum += X[j][i]; }
   }
   return sum;
}
```

Dans tout l'exercice, on suppose que le tableau X est placé à l'adresse 0x10010000, et qu'il est passé en paramètre de la fonction my_sum.
**Rappel (i)** 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 implanté sur 4 octets.
**Rappel (ii)** En langage, C les éléments du tableau X à 2 dimensions sont rangés en mémoire dans l'ordre : X[0][0], X[0][1], X[0][2], ... , X[0][15], X[1][0], X[1][1], X[1][2], ... , X[1][15], X[2][0] ...

**B1 (1 point).** Déterminer la taille d'une ligne (en octets) et donner le nombre de bits des champs offset, index et étiquette d'une adresse.

Taille d'une ligne : 32 768 / 512 = 64 octets
Offset : 6 bits
Index : 9 bits
Etiquette : 17 bits

**B2 (1 point).** Représenter dans le schéma ci-dessous les cases valides du cache après 2 itérations de la boucle interne, en précisant les indexes (il y a au plus 2 cases valides).

| Index | Adresse Mot0 | Mot15 | Mot14 | Mot13 | Mot2 | Mot1 | Mot0 |
|---|---|---|---|---|---|---|---|
| 0 | 0x10010000 | X[0][15] | X[0][14] | X[0][13] | X[0][2] | X[0][1] | X[0][0] |
| 1 | 0x10010040 | X[1][15] | X[1][14] | X[1][13] | X[1][2] | X[1][1] | X[1][0] |





**B3 (1 point).** On rappelle qu'une case du cache contient une ligne. Quels sont les éléments de X qui peuvent occuper le mot d'offset 12 de la case d'index 3 du cache ? Même question avec le mot d'offset 16 de la case d'index 256.

Mot d'offset 12 de la case d'index 3 : X[3][3], X[515][3]
Mot d'offset 16 de la case d'index 256 : X[256][4]

**B4 (1 point).** Calculer, en le justifiant, le nombre de miss sur le cache de données rencontrés lors de l'exécution de cette fonction. En déduire le taux de MISS sur le cache de données.

X est aligné sur la taille du cache (l'adresse de base est un multiple de la taille du cache, 0x8000).
Il y a un conflit entre les éléments X[i][j] et X[i + 512][j] pour i dans [0 ; 255] et j dans [0 ; 15].
On aura donc 768 miss pour la première itération externe (miss cold), puis 512 pour chacune des 15 itérations suivantes.
Il y a donc au total 768 + 15 * 512 = 8 448 miss.

Il y a 16 * 768 = 12 288 accès.
Le taux de miss est donc de 8 448 / 12 288 = 68.75%

**B5 (0.5 point).** La compilation de ce code a produit une boucle interne contenant 8 instructions, qui mettrait donc 8 cycles à s'exécuter si le système mémoire était parfait. On néglige l'effet des miss sur le cache d'instructions. En utilisant le résultat de la question B4, calculer le nombre de cycles moyen par itération dans la boucle interne si un miss de données coûte 11 cycles.

Nombre de cycles total = 16 * 768 * 8 + 8 448 * 11 = 191 232
Il y a 12 288 itérations, soit une moyenne de 191 232 / 12 288 = 15.56 cycles/itération

**B6 (0.5 point).** Comment peut-on réduire le taux de miss ?

Solution matérielle :
Disposer d'un cache plus grand (doubler la taille) afin que tout le tableau tienne dedans.
Solution logicielle :
On peut inverser les deux boucles.

Dans les deux cas, il n'y a plus que 768 miss.

Déplacer le tableau dans l'espace d'adressage ne change rien au taux de miss.