EXERCICE A : programmation assembleur (5 points) **Numéro :**

Un élément d'une liste chainée est une structure de données appelée *chain*. Une structure *chain* est composée de deux pointeurs (un pointeur est un mot de 4 octets qui contient l'adresse d'un autre objet). Le premier pointeur donne l'adresse de la donnée. Le deuxième pointeur donne l'adresse de l'élément de liste suivant. Cette structure de données est organisée en mémoire comme deux mots enregistrés à des adresses consécutives. NULL est une constante égale à 0.

```
struct chain {
    void         *DATA;
    struct chain *NEXT;
};
```

La fonction Reverse inverse l'ordre des éléments de la liste.

```
struct chain *Reverse (struct chain *pt)
{
    struct chain *precedent = NULL;
    struct chain *suivant   = NULL;

    while (pt != NULL)
    {
        suivant   = pt->NEXT;
        pt->NEXT  = precedent;
        precedent = pt;
        pt        = suivant;
    }
    return precedent;
}
```

A1. Écrire en assembleur les directives créant la liste suivante

```
char  s1[] = "module";
char  s2[] = "almo";
chain ch2 = { s2, NULL };
chain ch1 = { s1, &ch2 };
```

```
    .DATA
s1  .asciiz  "module"
s2  .asciiz  "almo"
ch2 .word    s2, 0
ch1 .word    s1, ch2
```

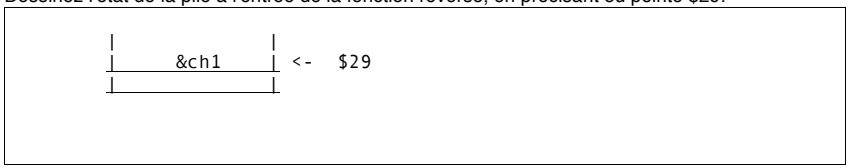
A2. Dites, en quelques mots, ce qu'il y a dans le prologue d'une fonction.

Dans le prologue d'une fonction, on réserve un espace dans la pile pour sauvegarder l'adresse de retour de la fonction et les registres persistants utilisés par la fonction, pour les variables locales de la fonction et pour les arguments des fonctions appelées.
On sauve l'adresse de retour et les registres persistants modifiés.

A3. En supposant que la fonction reverse() est appelée sur la liste ch1, de la manière suivante

```
...
reverse( &ch1 );
...
```

Dessinez l'état de la pile à l'entrée de la fonction reverse, en précisant où pointe $29.



A4. En assignant le registre $8 pour la variable "`precedent`", $9 pour la variable "`suivant`", écrivez le prologue de la fonction `Reverse()`. Vous pouvez tenir compte du fait que `Reverse()` est une fonction terminale ou ne pas en ternir compte.

```
Reverse est une fonction terminale, $8 et $9 sont des registres
temporaires, par conséquent on peut réduire le prologue à rien.

Sinon on doit réserver 1 case pour $31 nr=1 et deux cases pour les
variables locales nv=2

Reverse:
      addiu $29, $29, -3*4
      sw    $31, 8($29)
```

A5. Écrivez le corps de la fonction avec toujours $8 pour "`precedent`" et $9 pour "`suivant`"

```
On utilise
le registre $8 pour la variable precedent et
le registre $9 pour la variable suivant

            li    $8 , 0           # precedent = NULL
            li    $9 , 0           # suivant   = NULL

            beq   $4 , $0 , end_loop

    loop:   lw    $9 , 4 ($4 )     # suivant = pt->NEXT
            sw    $8 , 4 ($4 )     # pt->NEXT = precedent
            or    $8 , $4 , $0     # precedent = pt
            or    $4 , $9 , $0     # pt = suivant
            bne   $4 , $0 , loop

    end_loop:
            or    $2 , $8 , $0     # valeur de retour
```

EXERCICE B : Caches de Ier niveau (5 points) **Numéro :**

Le but de cet exercice est de mesurer le nombre de cycles nécessaires à l'exécution du programme C ci-dessous en tenant compte des effets de cache. Ce programme implémente une partie de l'algorithme de « tri fusion », en fusionnant deux tableaux préalablement triés (A et B) pour créer un seul tableau (C).

```
int A[512], B[512], C[1024];
int main() {
  register int i = 0, j = 0, k = 0;
  while (i < 512 || j < 512) {
    register int a, b;
    a = (i < 512) ? A[i] : a;
    b = (j < 512) ? B[j] : b;
    if (a < b) {
      C[k] = a;
      i++;
    }
    else {
      C[k] = b;
      j++;
    }
    k++;
  }
  return 0;
}
```

On considère un cache de données L1, suivant une stratégie de correspondance directe, d'une capacité totale de 2 ko. Chaque ligne de ce cache a une taille de 32 octets (8 mots de 32 bits). On suppose que le cache de données est initialement vide (tous ses bits de validité sont à 0).

On suppose que le tableau A est initialisé avec les nombres de 1 à 512, le tableau B avec les nombres de 1024 à 1536. Exemple : A[0] = 1, A[1] = 2, B[0] = 1024, B[1] = 1025, ...

Les données du programme sont stockées de façon contigüe dans la mémoire, dans l'ordre de leur déclaration. Le premier élément A[0] est rangé à l'adresse 0x00001000.

Le mot clé « register », utilisé dans le programme C, est une directive passée au compilateur pour qu'il place les variables i, j, k, a et b dans des registres plutôt que sur la pile du programme. Ces variables sont donc contenues dans des registres durant toute la durée d'exécution du programme et leur lecture ne provoque pas d'accès au cache de données.

Les adresses sont sur 32 bits et chaque adresse référence un octet en mémoire.
On rappelle que 1 ko = 1024 octets = 2^{10} octets = 0x400 octets.

B1. Donnez les adresses de base en mémoire des deux tableaux B et C.

B = 0x00001800
C = 0x00002000

B2. Donnez le nombre de cases de ce cache (une case permet de stocker une ligne) et le nombre des bits respectifs des champs « offset », « index » et « étiquette » de l'adresse.

Nombre de cases = 64
Nombre de bits d'offset = 5
Nombre de bits d'index = 6
Nombre de bits d'étiquette = 21

B3. Donnez l'état de ce cache de données après le premier tour de boucle en remplissant le tableau ci-dessous. Le champ « index » contient l'index de case du cache. Pour faciliter la compréhension, le champ « étiquette » contiendra l'adresse complète (sur 32 bits) du mot d'indice 0 de la ligne de cache correspondante. Le champ « data_i » contient le mot d'indice « i » de la ligne de cache. On utilisera la notation hexadécimale, précédée de 0x pour l'étiquette **et les données.**

index	valid	étiquette	data_7	data_6	...	data_1	data_0
0	1	0x00001800	0x407	0x407		0x401	0x400

B4. Donnez le nombre de MISS sur le cache de données pour les 8 premières itérations de la boucle ; pour la 9ième itération, la 10^e itération, et la dernière itération. Quel est le nombre total de MISS pour l'exécution complète de la boucle ?

$1^{ère}$ itération : 1 miss sur A[0] et un miss sur B[0] (évince A[0]).
2^e à 8^e itération : 2 miss par itération (situation identique de conflit)
9^e itération : 1 miss sur A[8], hit de B[0]
10^e à 16^e : pas de miss
Puis : un miss à la 17^e (A[16]), et toutes les huit itérations.
A partir de l'itération 1017, on a à nouveau la situation de conflit.
Dernière itération : 2 miss (A[511] et B[511])
Au final : chaque tableau provoque 512 / 8 = 64 miss utiles
Et il y a une situation de conflit 2 fois sur une ligne complète (2 fois 15 miss)
Soit au total : 64 * 2 + 15 * 2 = 158 miss

B5. La compilation de cette boucle génère une séquence de 11 instructions en assembleur MIPS 32, qu'on passe dans le then ou le else. Grâce à la technique de pipeline, le CPI (nombre de cycles par instruction) avec un système mémoire parfait est de 1 cycle par instruction. Le coût d'un MISS est de 15 cycles. On considère que le cache instruction se comporte comme un cache parfait (0 MISS). Le nombre total de MISS de donnée pour l'exécution complète de la boucle est NB_Miss (calculé à la question précédente).

- Donnez le temps total (en nombre de cycles horloge) nécessaire à l'exécution des 1024 itérations de la boucle.
- Quel est le CPI réel ?

Il y a 1024 itérations de 11 instructions, soit 1024*11 = 11264 cycles d'exécution. Il faut y ajouter les MISS de données, soit 158*15 = 2370 cycles de gel. Le temps total d'exécution est de 11264+2370 = 13634 cycles.

Le CPI réel est donc (nombre de cycles / nombre de d'instructions exécutées) = (13634 / 11264) = 1,21 cycles/instruction.

EXERCICE C : Interruptions et périphériques (5 points) **Numéro :**

Dans ce questionnaire à choix multiples, chaque bonne réponse vous rapporte ½ point, mais attention, chaque mauvaise réponse vous coûte ¼ de point. Il n'y a qu'une réponse valide par question.

C1/ Le composant ICU est connecté aux lignes d'interruptions de la façon suivante. Le tty est connecté sur l'entrée irq_in[2] de l'ICU, le timer est connecté sur l'entrée irq_in[3] et le DMA est connecté sur l'entrée irq_in[7]. Comment faut-il programmer le masque de l'ICU pour démasquer uniquement ces trois lignes d'interruptions ?

- 0x00000111
- (1 << 8) | (1 << 4) | (1 << 3)
- ⇨ 0x0000008C
- 141

C2/ On suppose qu'une tâche qui s'exécute sur le processeur est interrompue par une interruption matérielle. Donnez la succession des événements qui se produisent dans le système (l'ordre est important).

- Acquittement interruption; Sauvegarde par le processeur du compteur ordinal dans EPC Saut du processeur à l'adresse int_handler ; Saut du processeur à l'adresse 0x80000180.
- Saut du processeur à l'adresse 0x80000180 ; Sauvegarde par le processeur du compteur ordinal dans EPC ; Saut du processeur à l'adresse int_handler ; Acquittement interruption.
- Saut du processeur à l'adresse int_handler ; Acquittement interruption ; Sauvegarde par le processeur du compteur ordinal dans EPC ; Saut du processeur à l'adresse 0x80000180.
- ⇨ Sauvegarde par le processeur du compteur ordinal dans EPC; Saut du processeur à adresse 0x80000180; Saut du processeur à l'adresse int_handler; Acquittement interruption.

C3/ On suppose que le processeur se branche à l'adresse 0x80000180. On constate que le register CR contient la valeur 0x00000018. Quelle est la cause de l'appel au GIET ?

- ⇨ C'est une exception de type "instruction bus error".
- C'est une exception de type "data bus error".
- C'est un appel système.
- C'est une interruption matérielle.

C4/ Lors d'une interruption, d'une exception ou d'un appel système, quels registres sont automatiquement modifiés de manière matérielle par le processeur lui-même ?

- Les registres $29 et $31.
- ⇨ Les registres EPC, CR et SR.
- Les registres $26 et $27.
- Les registres $1 à $15, EPC et CR.

C5/ Lorsqu'un programme utilisateur exécute un appel système, quel registre utilise le GIET pour déterminer le numéro de l'appel à effectuer ?

- $4
- ⇨ $2
- $31
- $29

C6/ Quelle affirmation concernant l'instruction mfc0 est-elle vraie ?

- Elle doit toujours être suivie d'une instruction "nop", afin de vider le contenu du pipeline.
- De la même façon que mflo, elle permet de récupérer le contenu du registre nommé c0.
- C'est l'instruction de masque booléen de tous les registres systèmes.
- ⇨ C'est une instruction habituellement privilégiée permettant de lire les registres systèmes.

C7/ Dans le code du GIET, à quoi sert la ligne de code "_isr_func_t _interrupt_vector[32] = { [0…31] = & _isr_default }", extraite du fichier irq_handler.c ?

- À initialiser les numéros des interruptions avec les valeurs 0 à 31.
- À initialiser la fonction _isr_default avec la bonne valeur.
- ⇨ À installer une ISR par défaut, utilisée lorsqu'une interruption non prévue se produit.
- À vérifier que les indices des interruptions sont bien compris entre 0 et 31.

C8/ Laquelle de ces actions ne nécessite pas que le processeur soit en mode superviseur ?

- ⇨ Éxecuter l'instruction assembleur "syscall" .
- Éxecuter l'instruction assembleur "eret" .
- Masque les interruptions matérielles.
- Accéder à l'adresse 0xA0A0A0A0.

C9/ Dans quel registre est stocké l'information qui code le mode courant du processeur (user/kernel) ?

- Dans le registre système CR uniquement.
- ⇨ Dans le registre système SR seulement.
- Dans les deux registres systèmes CR et SR.
- Dans aucun de ces registres puisqu'on passe en mode kernel par une interruption.

C10/ Comment un périphérique signale t-il au système qu'il a terminé le traitement qu'on lui a demandé ?

- En générant une exception.
- ⇨ En générant une interruption.
- En utilisant un appel système.
- Le logiciel applicatif doit toujours vérifier régulièrement si le traitement est terminé.

EXERCICE B : Caches de 1er niveau (5 points) Numéro :

Le but de cet exercice est de mesurer le nombre de cycles nécessaires à l'exécution du programme C ci dessous en tenant compte des effets de cache.

```
int A[256], B[256], int C[256];
int main() {
   register int i, sum = 0;
   for (i = 0; i < 256; i++) {
      C[i] = A[i] + B[i];
      sum = sum + C[i];
   }
   return 0;
}
```

On considère un cache de données L1, à correspondance directe, d'une capacité totale de 4Ko. Chaque ligne de ce cache a une taille de 16 octets (4 mots de 32 bits).
Les tableaux sont initialisés selon les formules suivantes :

```
A[i] = i     exemple A[10] = 10
B[i] = i + 1       exemple B[12] = 13
C[i] = 2 * i       exemple C[20] = 40
```

On suppose que le cache de données est initialement vide (tous ses bits de validité sont à 0).
Les données du programme sont stockées de façon contigüe dans la mémoire, dans l'ordre de leur déclaration. Le premier élément A[0] du tableau A est à l'adresse 0x10000.
Le mot clé "register" est une directive passée au compilateur pour qu'il place les variables i et sum dans des registres plutôt que sur la pile : ces variables sont donc dans des registres durant toute la durée d'exécution du programme et ni leur lecture ni leur écriture ne provoquent d'accès au cache de données.
Les adresses sont sur 32 bits, et chaque adresse référence un octet en mémoire. On rappelle de plus que 1Ko = 1024 octets = 0x400 octets.

B1) Donner les adresses de base en mémoire des deux tableaux B et C.

&B[0] = 0x10400
&C[0] = 0x10800

B2) Donner le nombre de cases de ce cache (une case permet de stocker une ligne), le nombre de bits des champs offset, index, et étiquette de l'adresse.

-le nombre de cases 256 = 4096 / 16
-la taille du champ offset 4 = log(16)
-la taille du champ index 8 = log (4096/16)
-la taille du champ étiquette 20 = 32 – 8 - 4

B3) Donner l'état de ce cache de données après le premier tour de boucle en remplissant le tableau ci-dessous (pour les cases non vides, c'est-à-dire valides).

Le champ index contient l'index de la case du cache. Pour faciliter la compréhension, le champ étiquette contiendra l'adresse complète (sur 32 bits) du mot d'indice 0 de la ligne de cache concernée. Le champ data i contient le mot d'index i de la ligne de cache. On utilisera la notation hexadécimale, précédée de 0x pour les adresses.

Index	Valid	Étiquette	Data 3	Data 2	Data 1	Data 0
0	1	0x10000	3	2	1	0
64	1	0x10400	4	3	2	1
128	1	0x10800	6	4	2	1

B4) Donner le nombre de MISS sur le cache données pour les 3 premières itérations de la boucle. Quel est le nombre total de MISS pour l'exécution complète de la boucle ?

On a 3 MISS pour la première itération (A[0], B[0], C[0]). On a 3 MISS pour la quatrième (A[4] et B[4], C[4]) et 3 MISS pour toutes les iterations où I est un multiple de 4. On a donc au total (256 / 4) * 3 = 192 MISS.

B5) La compilation de cette boucle génère une séquence de 18 instructions en assembleur MIPS 32 (on ne considère pas la déclaration des variables ni leur initialisation). Grâce à la technique de pipeline, le CPI avec un système mémoire parfait est de 1 cycle par instruction. Le coût d'un MISS est de 10 cycles. On considère que le cache instruction se comporte comme un cache parfait (0 MISS). Donner le temps total (en nombre de cycles horloge) nécessaire à l'exécution des 256 itérations de la boucle. Quel est le CPI réel ?

Il faut exécuter 18 * 256 instructions = 4608 cycles.
Comme on a 192 MISS, on a un surcoût de 1920 cycles.
Le CPI effectif est de (4608+ 1920) / 4608 = 1.42 cycles / instruction