EXERCICE A : Programmation en assembleur (5 points)

Corrigé

A1) Quelles sont les 3 grandes parties qui composent une fonction F écrite en assembleur ?

Une fonction F en assembleur se compose (dans l'ordre):
- d'un prologue, où l'on décrémente le pointeur de pile pour réserver de la place pour les arguments des fonctions appelées par F, pour sauvegarder les registres persistants, et enfin pour contenir les variables locales de F
- d'un corps de fonction, qui contient le code assembleur correspondant à l'algorithme décrit par la fonction F
- d'un épilogue, où on restaure éventuellement les registres persistants qui ont été modifiés par le corps de fonction, puis on incrémente le pointeur de pile pour le remettre à sa valeur au moment de la premiére instruction du prologue.

Soit le programme C suivant :

```
int fonca(int a)
{
     int la1,la2;

     la1=a+1;
     la2=a+2;
     return (la1+la2)/2;
}
int foncb(int b1, int b2)
{
     int lb1;
     lb1=b1*2 + b2;
     return lb1;
}
int foncc()
{
     int lc1, lc2, lc3, lc4, lc5;
     lc1=1;
     lc2=2;
     lc3=3;
     lc4=fonca(lc1);
     lc5=foncb(lc2,lc3);
     return 0;
}
```

Dans le tableau suivant, **que vous devez remplir**, **nv** représente le nombre de variables locales de la fonction considérée, **na** représente le nombre maximum d'arguments des fonctions appelées par la fonction considérée, **nr** représente le nombre de registres persistants à sauvegarder pour la fonction considérée, en

incluant $31. La colonne « Première instruction du prologue » doit contenir la première instruction de la fonction, faisant partie du prologue.

A2) A partir du code C précédent, remplir le tableau suivant

	nv	na	nr	Première instruction du prologue
fonca()	**2**	**0**	**1**	**addiu $29,$29,-4*(2+0+1)**
foncb()	**1**	**0**	**1**	**Addiu $29, $29, -4*(1+0+1)**
foncc()	**5**	**2**	**1**	**Addiu $29,$29,-4*(5+2+1)**

A3) On s'intéresse à la fonction fonca(int a). On suppose que $4 contient la valeur de l'argument a, et que l'on utilise respectivement $5 et $6 pour les variables locales la1 et la2. Ecrire le code assembleur correspondant au corps de fonction de fonca(int a). La division entière par 2 sera signée et effectuée de manière optimisée. Le résultat sera mis dans $2. Le code sera commenté.

```
     addiu     $5,$4,1   # la1=a+1
     addiu     $6,$4,2   # la2=a+2
     add  $6,$5,$6  # la2=la1+la2
     sra       $2,$6,1   # $2 = la2 >> 1 (division par 2)
```

A4) Dessiner la pile une fois que la première instruction du prologue de foncc() a été effectuée. En particulier, préciser les zones de la pile où vont être stockés les registres persistants, les variables locales et enfin les arguments des fonctions appelées par foncc().

```
          $31        contiendra le registre persistant $31
          lc1        contiendra la variable locale lc1 de foncc()
          lc2        contiendra la variable locale lc2 de foncc()
          lc3        contiendra la variable locale lc3 de foncc()
          lc4        contiendra la variable locale lc4 de foncc()
          lc5        contiendra la variable locale lc5 de foncc()
          arg[1]     contiendra lc3 pour l'appel à foncb()
$29 ->    arg[0]     contiendra lc1 pour l'appel à fonca(), lc2 pour foncb()
```

EXERCICE B : Caches de Ier niveau (5 points)

Corrigé

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[1024], B[1024], C[1024];
int main () {
      register int i =0, j=0, k=0;
      while ((i < 1024) && (j<1024)) {
            register int var_a, var_b;
            var_a = (i < 1024) ? A[i] : var_a;
            var_b = (j < 1024) ? B[j] : var_b;
            if (var_a < var_b) {
                  C[k] = var_a;
                  i++;
            } else {
                  C[k] = var_b;
                  j++;
      }
      k++;
}
```

On considère un cache de données L1, suivant une stratégie de correspondance directe, d'une capacité totale de 4 ko. Chaque ligne de ce cache a une taille de 64 octets (16 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 B est initialisé avec les nombres de 1 à 1024, le tableau A avec les nombres de 1025 à 2048. Exemple : A[0]=1025, A[1]=1026, B[0]=1, B[1]=2, ... 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, var_a et var_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 = 0x2000

C = 0x3000

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

Nombre de cases = 4096/64 = 64

Nombre de bits d'offset = log2(64) = 6 bits

Nombre de bits d'index = log2(64) = 6 bits

Nombre de bits d'étiquette = 32 - 6 – 6 = 20 bits

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_15	data_14	...	data_1	data_0
0b000000	1	0x2000	0x10	0xF	…	0x2	0x1

Il y a seulement la ligne de cache qui concerne B, puisqu'elle est lue en dernier et qu'elle aura alors remplacé A (ce qui restera d'ailleurs vrai pour les 16 premières itérations).

B4) Donnez le nombre de MISS sur le cache de données pour les 16 premières itérations de la boucle ; pour la 17ième itération ; de la 18ième itération à la fin de l'exécution. Quel est le nombre total de MISS pour l'exécution complète de la boucle ?

- Pour les 16 premières itérations, les tableaux A et B sont en conflit sur la première case du cache, il y a donc 2 MISS par itération, soit 32 MISS.
- Pour la 17ieme itération, il y a un MISS pour A et un MISS pour B (mais pour la case de cache suivante), soit 2 MISS.
- De la 18ieme itération à la fin du programme, il n'y a plus de MISS pour A, seulement pour B : un MISS tous les 16 itérations, soit 62 MISS.
- Total de MISS : 32 + 2 + 62 = 96 MISS

B5) La compilation de cette boucle génère une séquence de 11 instructions en assembleur MIPS 32 (on ne considère pas la déclaration des variables ni leur initialisation). 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 25 cycles. On considère que le cache instruction se comporte comme un cache parfait (0 MISS). Le nombre total de MISS de données pour l'exécution complète de la boucle est DMISS (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 (indiquez deux chiffres après la virgule) ?

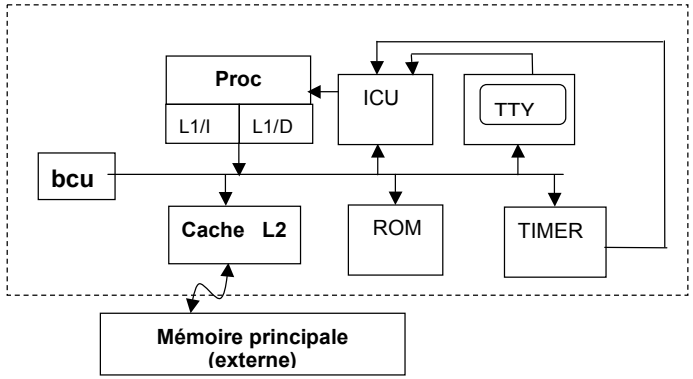
Il y a 1024 itérations de 11 instructions, soit 1024*11 = 11264 cycles d'exécution. Il faut y ajouter les MISS de données, soit 96*25 = 2400 cycles de gel. Le temps total d'exécution est de 11264+2400 = 13664 cycles.

Le CPI réel est donc (nombre de cycles /nombre de d'instructions exécutées) = (13664/11264) = 1,21 cycles/instruction.

EXERCICE C : Mécanisme d'interruption (5points)

Corrigé

On considère l'architecture matérielle ALMO-MONO (utilisée en TP). Cette architecture contient un seul processeur (avec ses caches de 1er niveau), un timer, un terminal écran/clavier de type TTY, une ROM contenant le « code de boot », un cache de 2e niveau permettant d'accéder à la mémoire externe, et un concentrateur d'interruptions ICU.



Le processeur exécute un programme dont le but est d'afficher sur l'écran TTY les caractères saisis au clavier. L'algorithme de ce programme est résumé ci-dessous :

```
char      to_print ;
while (1)
{
   if  (tty_get_irq(&  to_print))       tty_printf  ("\t%c\n", to_print);
}
```

Ce programme utilise le mécanisme d'interruption, plutôt qu'une attente active : le terminal envoie une interruption lorsqu'un caractère est saisi au clavier.

C1) Que doit on faire dans la phase d'initialisation du système (code de boot) pour que le mécanisme d'interruption fonctionne correctement ? On ne demande aucune valeur numérique, seulement les principes et les noms de variables/registres/structures utilisés.

Il faut configurer le masque d'interruptions dans le composant ICU, c'est le registre ICU_MASK_SET qui est utilisé.

Il faut ensuite placer l'adresse de la routine de traitement de l'interruption dans le tableau interrupt_vector. L'index dans ce tableau est égal au numéro de l'interruption

C2) Que fait l'appel système tty_get_irq() ? On décrira précisément la suite des appels de fonctions déclenchée par l'exécution de cet appel système, en précisant ce que fait chaque fonction, jusqu'au retour de l'appel système.

L'appel système tty_get_irq() contient une instruction assembleur syscall qui force le branchement au GIET (adresse 0X80000180)

Le GIET analyse le contenu du registre CR, et se branche au « gestionnaire d'appels système » grâce à une première table de sauts.

Celui-ci analyse le numéro de l'appel système contenu dans le registre $2, et se branche à la fonction système *_tty*_read_irq() grâce à une seconde table de sauts.

Cette fonction _tty_read_irq() teste la valeur de la variable de synchronisation _tty_get_full[i] attachée au terminal [i], qui indique l'état du tampon _tty_get_buf[i]

- Si cette variable est non-nulle, la fonction copie le caractère contenu dans le tampon _tty_get_buf[i] dans le tampon du programme utilisateur passé en argument de l'appel système (to_print), et retourne une valeur 1 (succès).
- Si cette variable est nulle, la fonction retourne la valeur 0 (échec).

La fonction _tty_read_irq() retourne au gestionnaire d'appels systèmes, qui rend la main au programme utilisateur par une instruction eret.

C3) L'utilisateur appuie sur une touche du clavier, ce qui active l'interruption provenant du contrôleur TTY. Décrivez précisément ce qui se passe dans la machine entre l'activation de l'interruption et la reprise d'exécution du programme interrompu.

Le TTY envoie une interruption vers le concentrateur d'interruptions ICU.

L'ICU verifie que cette interruption n'est pas masquée et transmet l'interruption vers le CPU.

Le CPU se branche au GIET (0x80000180)

Analyse de CR : le GIET se branche au handler d'interruption.

Lecture du registre ICU_IT_VECTOR, récupération du n° d'interruption

Branchement à la routine d'interruption liée au tty

Récupération de la valeur du caractère saisi dans le registre TTY_READ

Acquittement de l'interruption

Ecriture de ce caractère dans _tty_get_buf[0] et _tty_get_full[0]=1

Sortie du GIET par l'instruction eret et reprise du programme interrompu

Exercice D : Code Système (5 points)

Corrigé

D1) Le composant icu de la plateforme de l'exercice C est connecté aux lignes d'interruption de la manière suivante : le tty est sur l'entrée irq_in[1] de l'ICU, le timer sur l'entrée irq_in[0], comment faut-il programmer le masque de l'ICU pour démasquer les deux interruptions?

☐ 0x00000011

☐ 0x00000000000000000000000000000011

☐ 0x00000033

=> ☐ 3

D2) Un « appel système » est

☐ Une instruction privilégiée exécutable en mode système pour accéder à certains registres.

=> ☐ Une fonction exécutée en mode utilisateur qui provoque l'exécution de code système.

☐ Une fonction qui s'exécute en mode système.

☐ Un type particulier d'interruption géré par le GIET.

D3) Laquelle de ces actions ne nécessite pas que le processeur soit en mode superviseur

=> ☐ exécuter l'instruction assembleur syscall

☐ exécuter l'instruction assembleur eret

☐ masquer les interruptions

☐ accéder à l'adresse 0x80000000

D4) Qu'est-ce que le taux de MISS d'un cache ?

☐ C'est l'accélération observée d'un programme grâce à son cache.

☐ C'est le rapport entre le nombre de MISS et le nombre de transactions sur le bus.

=> ☐ C'est le rapport entre le nombre de requêtes qui font MISS et le nombre total de requêtes.

☐ C'est le rapport entre le nombre de MISS et le nombre de cycles exécutés.

D5) Dans le GIET, les interruptions sont masquées pendant l'exécution des ISR car :

☐ Une ISR est par principe in-interruptible.

☐ Il ne faut pas perdre la cohérence de la pile.

☐ Le code système est par principe in-interruptible.

=> ☐ Les ISRs utilisées sont très courtes, donc on peut se le permettre.

D6) Registres privilégiés

☐ HI / LO sont des registres privilégiés.

☐ $26 et $27 sont des registres privilégiés.

☐ $29 (le pointeur de pile) est un registre privilégié.

=> ☐ Ni HI, ni LO, ni $26, ni $27, ni $29 ne sont privilégiés.

D7) Parallélisme de tâche par multiplexage temporel

☐ On ne parle de parallélisme de tâches que sur les machines multiprocesseurs.

☐ Les tâches partagent la même pile d'exécution.

=> ☐ Les taches utilisent chacune leur propre pile d'exécution.

☐ Sur une machine multiprocesseur, la communication entre tâches utilise le DMA.

D8) A quoi sert la table des pages dans le mécanisme de mémoire virtuelle ?

☐ Connaître à l'avance la quantité de mémoire utilisée par un processus.

=> ☐ Retrouver le numéro de page physique connaissant le numéro de page virtuelle.

☐ Retrouver le numéro de page virtuelle connaissant le numéro de page physique.

☐ Accélérer le calcul de traduction d'adresse.

D9) A quoi sert le fichier ldscript ?

☐ C'est une description de l'ensemble des outils utilisés dans la chaîne de compilation croisée.

=> ☐ C'est un fichier qui définit les adresses de base des différents segments.

☐ C'est un fichier qui contient la définition des tables de page.

☐ C'est un script qui appelle l'éditeur de lien ld.

D10) à propos de l'instruction mfc0

☐ Elle permet de lire le contenu des registres protégés, lorsque le processeur est en mode user

☐ Comme mfhi et mflo, elle permet de récupérer le contenu du registre c0.

☐ C'est l'instruction de masquage Booléen des registres systèmes.

=> ☐ C'est une instruction privilégiée pour lire le contenu des registres systèmes.