EXERCICE A1 : Programmation en assembleur

Le but de cet exercice est de coder en assembleur MIPS une version récursive de l'algorithme de recherche binaire par dichotomie dans un tableau d'entiers déjà triés. L'algorithme est décrit en C dans la fonction ***binarySearch*** illustré ci-dessous. Les paramètres de la fonction sont : un pointeur sur le premier élément du tableau à analyserr : ***array***, la valeur à trouver ***value***, et deux index dans le tableau définissant la borne inférieure de recherche ***left*** et la borne supérieur de recherche ***right***.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5         int tab[] = {1,12,30,41,55 ,63,70,88,92,105} ;
6         int val = 92 ;
7         int a = 0 ;
8         int b = 9 ;
9         int pos = binarySearch(tab, val, a, b) ;
10        printf(« %d est trouvé dans la position %d\n », val, pos) ;
11        return (0) ;
12 }
13 int binarySearch(int* array, int value, int left, int right)
14 {
15        if (left > right)
16            return -1;
17        int middle = (left + right) / 2;
18        if (array[middle] == value)
19            return middle;
20        else if (array[middle] > value)
21                return binarySearch(array, value, left, middle - 1);
22              else
23                return binarySearch(array, value, middle + 1, right);
24    }
```

A11) A l'entrée de la fonction binarySearch, sur quoi pointe le pointeur de pile contenu dans $29 ?

Le registre $29 pointe dans la pile sur une case qui a été réservée pour contenir le premier paramètre de la fonction binarySearch, c-a-d le paramètre array.

A12) Ecrire le prologue et l'épilogue de la fonction binarySearch en langage assembleur. On rappelle que l'argument array est dans le registre $4, l'argument value est dans le registre $5, l'argument left est dans le registre $6 et l'argument right est dans le registre $7. La fonction utilise une variable locale middle.

```
#prologue :
binarySearch    addi $29,$29, -24       # nv = 1, na = 4 , nr =1
                sw $4, 24($29)          # array
                sw $5, 28($29)          # value
                sw $6, 32($29)          # left
                sw $7,36($29)           # right
                sw $31,16($29)          # $31
#epilogue :
retour :        lw $31,16($29)          # $31
                addi $29,$29, 24
                jr $31
```

A13) En supposant l'association variable – registre suivante : (array , $4) (value , $5) (left , $6) (right , $7) (middle , $13) ($2, valeur de retour), écrivez le code assembleur correspondant aux lignes 20 et 21 de la fonction binarySearch(). On utilisera les registres $14 et $15 comme registres de travail.

```
#
                sll $14, $13, 2         # $14 <= middle*4
                add $14, $4, $14
                lw $15, 0($14)          # $15<= array[middle]
                sub $15, $15, $5
                bgtz  $15, appel        # test
                j next
appel :         subi $7, $13, 1
      :         jal binarySearch        # a
```

EXERCICE A2 : Programmation en assembleur

Dans cet exercice, on veut écrire l'algorithme de tri par sélection en langage d'assembleur du processeur MIPS 32. L'algorithme est codé en C dans la fonction sort() ci-dessous. Attention, l'algorithme itératif proposé ici est différent de l'algorithme récursif présenté en TD.

Le premier argument (a) de la fonction sort() est un pointeur vers le tableau d'entiers à trier. Le deuxième argument (size) est la taille du tableau.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int a[] = {8, 6, 10, 3, 1, 2, 5, 4};
 4 int n = 8;
 5 void swap(int *a, int *b)
 6 {
 7      int temp;
 8      temp = *a;
 9      *a = *b;
10      *b = temp;
11 }
12 void sort(int a[], int size)
13 {
14      int i, m, max;
15      while (size > 1) {
16           m = 0;
17           max = a[m];
18           for (i = 1; i < size; i++) {
19                if (a[i] > max) {
20                     m = i;
21                     max = a[m];
22                }
23           }
24           if (m != size)
25                swap(&(a[m]), &(a[size - 1]));
26           size = size - 1;
27      }
28 }
29 int main()
30 {
31      int i;
32      sort(a, n);
33      for (i = 0; i < n; i++)
34          printf("%d ", a[i]);
35      exit(0);
36 }
```

A21) A l'entrée de la fonction swap, sur quoi pointe le pointeur de pile contenu dans $29 ?

Le registre $29 pointe dans la pile sur une case qui a été réservée pour contenir le premier paramètre de la fonction swap, c-a-d le paramètre a.

A22) Ecrire la fonction swap en langage assembleur. On supposera que l'argument a est dans le registre $4 et l'argument b dans le registre $5. On utilisera deux registres de travail $13 et $14. Attention : cette fonction swap ne possède pas les mêmes arguments que la fonction swap vue en TP !

```
swap:
      # prologue
      add $29, $29, -4          # optionnel
      # corps
      lw $13, 0($4)
      lw $14, 0($5)
      sw $14, 0($4)
      sw $13, 0($5)
      # epilogue
      add $29,$29,4             # optionnel si le pointeur de pile n'a pas été modifié
      jr  $31
```

A23) En supposant l'association variable – registre suivante : (m , $8) (i , $9) (a , $10) (size , $11) écrivez le code correspondant aux lignes 24 à 26

```
# ligne24 : if (m != size)
      beq      $8, $11, ligne26

#ligne25 : swap( &(a[m]) , &(a[size-1]))
      sll      $4, $8 2
      addu     $4, $10, $11
      addiu    $5, $11, -1
      sll      $5, $5, 2
      addu     $5, $10, $11
      jal      swap

ligne26 :  # size = size – 1
      addiu    $11, $11, -1
```

EXERCICE B1 : Caches de Ier niveau

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[128],  B[128], C[128];
int main(){
  register int i, sum = 0;
  for (i=0; i<128; 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 1Ko. Chaque ligne de ce cache a une taille de 32 octets (8 mots de 32 bits).

Les tableaux sont initialisés selon les formules suivantes :

A[i] = I exemple A[10] = 10

B[i] = 2 * i exemple B[12] = 24

C[i] = i+1 exemple C[20] = 21

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 contiguë dans la mémoire, dans l'ordre de leur déclaration. Le premier élément A[0] du tableau A est à l'adresse 0x4000.

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. Les variables **i** et **sum** sont 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 que 1Ko = 1024 octets = 2^10 octets = 0x400 octets.

B11) Donner les adresses de base en mémoire des deux tableaux B et C.

m[A] = 0x4000
m[B] = 0x4200
m[C] = 0x4400

B12) Donner le nombre de cases de ce cache (une case permet de stocker une ligne), le nombre de bits des champ offset, index, et étiquette de l'adresse.

-le nombre de cases 32 = 1024 / 32
-la taille du champ offset 5 = log(32)
-la taille du champ index 5 = log (32)
-la taille du champ étiquette 22 = 32 – 5 - 5

B13) Donner 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 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 champs data i contient le mot d'index i de la ligne de cache. On utilisera la notation hexadécimale, précédée de 0x pour les adresses et pour les données.

index	Valid	Etiquette	data 7	data 6	....	data 1	data 0
0	*1*	*0x4000*	*0x07*	*0x06*		*0x01*	*0x00* ***
	0						
| 0	**1**	**0x4400**	**0x8**	**0x07**		**0x02**	**0x00**
	0						
0x10	**1**	**0x4200**	**0xE**	**0xC**		**0x02**	**0x00**

*** cette ligne a été evincée du cache

A chaque itération, il faut aller lire trois entiers en mémoire : A[i], B[i], C[i], et donc 3 lignes de cache.

La ligne de cache contenant A[0] est rangée dans la case d'index 0

La ligne de cache contenant B[0] est rangée dans la case d'index 4

La ligne de cache contenant C[0] est rangée dans la case d'index 0, (et C[0] est modifié)

La ligne de cache contenant A[0] est écrasée par celle contenant C[0]

B14) 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 2 MISS pour chacune des 2 suivantes (A[1] et C[1] puis A[2] et C[2]. En revanche B[1] et B[2] sont déjà dans le cache.

Comme on a 128 itérations, et qu'une ligne de cache contient 8 entiers, il faudra charger au total 16 lignes de cache pour la tableau B, tandis que les tableaux A et C font MISS à chaque lecture.. On aura au total 16 + 128 + 128 = 272 MISS

B15) La compilation de cette boucle génère une séquence de 20 instructions en assembleur MIPS 32 (on ne considère pas la déclaration des variables ni leur initialisation). Grâce à la technique de pipe-line, 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 128 itérations de la boucle. Quel est le CPI réel?

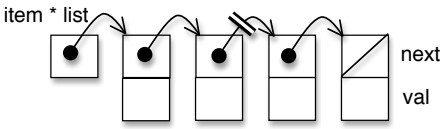
Il faut exécuter 20 * 128 instructions = 2560 cycles.

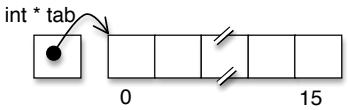
Comme on a 272 MISS, on a un surcoût de 2720 cycles.

Le CPI effectif est de (2560 + 2720) / 2560 = 2.06 cycles / instruction

EXERCICE B2 : Caches de Ier niveau

Le but de cet exercice est de mesurer le nombre de cycles nécessaires à l'exécution des fonctions C ci-dessous, en tenant compte des effets de cache. Les deux fonctions réalisent la somme des éléments d'un tableau. Dans la première fonction, le tableau est réalisé sous la forme d'une liste simplement chainée. Dans la seconde fonction, le tableau est une zone contiguë de la mémoire.






```
int sum_list (item * list)
{
      register int s = 0;
      while (list != NULL) {
            s = s + list→val;
            list = list→next;
      }
      return s;
}
```

```
int sum_tab (int size, int *tab)
{
      register int s = 0;
      while (size != 0){
            s = s + *tab;
            tab = tab + 1;
            size = size - 1;
      }
      return s;
}
```

```
sum_list: li   $2, 0
          j       test1
loop1:    lw      $3,4($4)
          lw      $4,($4)
          add     $2,$2,$3
test1:    bnez    $4,loop1
          jr      $31
```

```
sum_tab:  li   $2, 0
          j       test2
loop2:    lw      $3,($5)
          add     $4,$4,-1
          add     $5,$5,4
          add     $2,$2,$3
test2:    bnez    $4,loop2
                  jr
      $31
```

On considère un cache de données L1, « write-through » et à correspondance directe, d'une capacité totale de 4 ko. Chaque ligne de cache a une taille de 64 octets. On suppose que le cache de données est initialement vide (tous ses bits de validité sont à 0). Les adresses sont sur 32 bits et chaque adresse référence un octet en mémoire.

La variable `s`, préfixée par le mot clé « `register` », est placée dans un registre plutôt que sur la pile du programme. Durant toute la durée d'exécution du programme, sa lecture ne provoque pas d'accès au cache de données.

Le tableau `tab` est rangé en 0x10000040. Les éléments de la liste `list` sont placés en mémoire de telle sorte que chaque élément est dans une ligne de cache différente. La fonction `sum_list` est à l'adresse 0x400100, fonction `sum_tab` est à l'adresse 0x400200.

B21) 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 = 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

B22) On suppose que les fonctions n'ont encore jamais été exécutées. Donnez, **en le justifiant**, le nombre de miss instruction lors de l'exécution des fonctions sum_list et sum_tab, respectivement NB_MISS_INST_sum_list et NB_MISS_INST_sum_tab

NB_MISS_DATA_sum_list = 1

NB_MISS_DATA_sum_tab = 1

Chaque fonction commence en début de ligne de cache, et fait moins de 16 mots (la taille d'une ligne). Il n'y a donc qu'un seul miss par fonction.

B23) Si on suppose que la liste chainée contient 16 éléments. Donnez, **en le justifiant**, le nombre de miss data provoqué par l'exécution de la fonction sum_list et combien de miss data provoquées par l'exécution de la fonction sum_tab, respectivement NB_MISS_DATA_sum_list et NB_MISS_DATA_sum_data.

NB_MISS_DATA_sum_list = 16

NB_MISS_DATA_sum_tab = 1

Pour la fonction sum_list c'est simple, il y en a un par élément puisque tous les éléments sont dans des lignes différentes. Pour sum_tab, il n'y en a qu'un car le tableau tab commence en début de ligne.

B24) On suppose que le CPI est de 1 lors de l'exécution de chacune des fonctions `sum_list` et `sum_tab`. Déterminer, le nombre de cycles nécessaires pour exécution de chacune des 2 fonctions, sans tenir compte des miss instructions et des miss data, respectivement NBCYC_sum_list_parfait et NBCYC_sum_tab_parfait.

NB_CYCLE_sum_list_parfait = 2 + 4*16 + 2 = 68

NB_CYCLE_sum_tab_parfait = 2 + 5*16 + 2 = 84

B25) En supposant que le coût d'un miss instruction et le coût d'un miss data est de 40 cycles. Calculer la durée d'exécution des deux fonctions, respectivement NBCYC_sum_list et NBCYC_sum_tab

NB_CYCLE_sum_list =

NB_CYCLE_sum_list_parfait + 40*(NB_MISS_INST_sum_list + NB_MISS_DATA_sum_list)

= 67 + 40 * (1+16) = 747

NB_CYCLE_sum_tab =

NB_CYCLE_sum_tab_parfait + 40*(NB_MISS_INST_sum_tab + NB_MISS_DATA_sum_tab)

= 83 + 40 * (1+1) = 163

Exercice C : Interruptions, GIET

Cochez une seule réponse pour chaque question

C1) Concernant les interruptions, laquelle de ces assertions est vraie ?

☐ Grâce au composant ICU, il est impossible à deux périphériques, le TTY et le timer par exemple, d'activer simultanément la même ligne d'interruption.

=> ☐ Le composant ICU permet à plusieurs périphériques de partager, et donc d'utiliser le même signal d'interruption.

☐ Le délai qui sépare l'activation d'une l'interruption et l'exécution de l'ISR (Interrupt Service Routine) associée à cette interruption est toujours borné.

☐ Dans un système multi-processeurs, tous les processeurs reçoivent les interruptions, mais l'ISR est exécutée par le premier processeur qui reçoit l'interruption.

C2) Concernant le code de « boot », laquelle de ces assertions est vraie ?

=> ☐ Le code de boot s'exécute en mode superviseur car ce code doit accéder aux périphériques et aux registres protégés.

☐ Chaque processeur (quand il y en a plusieurs) dispose de son propre code de boot.

☐ Seuls les registres $26 et $27 peuvent être utilisés par le code de boot .

☐ Le code de boot doit initialiser tous les registres internes du processeur.

C3) Concernant le GIET, laquelle de ces assertions est vraie ?

☐ Tous les registres sont sauvés sur la pile avant d'exécuter la première instruction du GIET.

☐ Seuls les registres $26 et $27 sont sauvés automatiquement dans la pile avant d'exécuter la première instruction du GIET.

=> ☐ Un champ du registre CR (Cause Register) est initialisé automatiquement avec un numéro dont la valeur définit la cause de l'appel au GIET.

☐ Le GIET ne peut être exécuté que si le processeur est en mode utilisateur.

C4) 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

C5) 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.

C6) 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

☐ accèder à l'adresse 0x80000000

C7) 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.

C8) 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.

C9) 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.

C10) A quoi sert le fichier ldscript ?

☐ C'est une description de l'ensemble des outils utilisés dans la chaine 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.

C11) 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 système

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