EXERCICE A : Programmation assembleur (5 points) **Numéro d'anonymat :**

On se propose d'écrire une fonction qui prend un tableau "tab" de "size" entiers en argument et qui trouve l'entier qui a le plus grand nombre de bits à 1. Par exemple si tab est défini par :
`Int tab[3] = {0, 4, 7};`
Le nombre de bits à 1 de l'entier 0, c'est 0, le nombre de bit à 1 de l'entier 4, c'est 1, le nombre de bits à 1 de l'entier 7, c'est 3. Le nombre qui a le plus grand nombre de bits à 1 est donc 7.

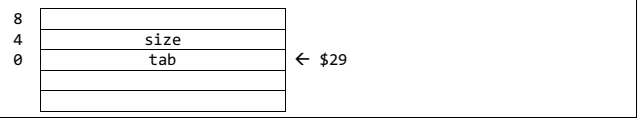
```
1  int nbbits (int n) {
2      int i, res;
3      res = 0;
4      for( i = 32; i != 0; i--) {
5            res = res + (n & 1);
6            n = n >> 1;
7      }
8      return res;
9  }
10 int val_nbbits (int * tab, int size)
{
11     int max, val, i;
12     val = tab[0];
13     max = nbbits(tab[0]);
14     for(i = 1; i != size; i++) {
15           if (max < nbbits(tab[i])) {
16                 max = nbbits(tab[i]);
17                 val = tab[i];
18           }
19     }
20     return val;
21 }
```

```
22 int tab[3] = {14, 29, 1027};
23 main() {
24     int val;
25     val = val_nbbits(tab, 3);
26     printf("%d", max);
27     return 0;
28 }
```

A1) Écrire en assembleur l'allocation du tableau "`tab`" (ligne 22) dans la section des données globales.

```
        .data
tab     .word   14, 29, 1027
```

A2) Représenter l'état de la pile avant l'exécution de la fonction "`val_nbbits`" (dans la fonction main). Indiquez clairement ce que pointe $29 (les adresses petites sont en bas, les adresses grandes sont en haut). Justifiez votre réponse.



A3) En supposant que, dans la fonction **`val_nbbits`**, les variables `tab, size, max, val` et `i` utilisent respectivement les registres `$16, $17, $18, $19` et `$20`. Le code de la fonction utilisera les registres $8, $9 et $10 pour les calculs intermédiaires. Combien de place (en octets) doit-on réserver dans la pile pour le contexte de la fonction val_nbbits. Justifiez votre réponse.

na = 1
nr = 5 + 1
nv = 3
10 mots = 40 octets
On ne réserve pas de place pour les registres temporaires.

A4) écrivez le prologue de la fonction `val_nbbits()`

```
val_nbbit:   addiu $29, $29, -40
             sw    $31, 36($29)
             sw    $20, 32($29)
             sw    $19, 28($29)
             sw    $18, 24($29)
             sw    $17, 20($29)
             sw    $16, 16($29)
```

A5) Écrivez le code des lignes 12 et 13. On suppose que les registres `$16, $17, $18, $19` et `$20` n'ont pas encore été initialisés.

```
L13:   addiu $16, $4, 0
       lw    $19, ($16)
L14:   addiu $4, $16, 0
       jal   nbbits
       addiu $18, $2
```

A6) La fonction `nbbits()` est une fonction terminale. En supposant que les variables `i` et `res` utilisent respectivement les registres `$8` et `$9`, écrivez le code de la fonction `nbbits()`.

```
nbbits:     li    $2,   0
            li    $8,   32
            j     nbbits_test
nbbits_for:
            andi  $9,   $4,   1
            add   $2,   $2,   $9
            srl   $4,   $4,   1
            addiu $8,   $8,   -1
nbbits_test:
            bnez  $8,   nbbits_for
            jr    $31
```

Exercice B : Mémoires Cache **Numéro :**

On considère un cache de données de premier niveau write-through à correspondance directe, d'une capacité totale de 8 Kio. La ligne de cache a une largeur de 32 octets (8 mots de 32 bits). Les adresses sont sur 32 bits.
Le but de l'exercice est d'analyser le remplissage d'un cache de données L1, puis d'estimer le nombre de cycles nécessaire à l'exécution d'une fonction. On suppose que le cache de données est initialement vide.
On considère la fonction suivante :

```
int32_t X[N];
int32_t Y[N];
...
void inv_scan(int32_t X[N], int32_t Y[N]) {
    Y[N - 1] = X[N - 1];
    for (register int32_t i = N - 2; i >= 0; i -= 1) {
        Y[i] = X[i] + Y[i + 1];
    }
}
```

Dans tout l'exercice, on suppose que les tableaux globaux X et Y sont implantés en mémoire respectivement aux adresses 0x1001 0000 et 0x1001 1000, que ces tableaux sont passés à la fonction inv_scan, et que N vaut 1024.

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 étiquette d'une adresse.

Offset : 5 bits
Index : 8 bits
Etiquette : 19 bits

B2 (1 point). Quels sont les éléments de X qui peuvent occuper les mots du cache suivants : case d'index 51, mot 7 ; et case d'index 128, mot 0 ? (une case contient une ligne)

Case d'index 51, mot 7 : X[415]

Case d'index 128, mot 0 : aucun, le dernier élément de X est placé dans la case 127, mot 7.

B3 (1 point). Représenter dans le schéma ci-dessous la ou les case(s) valide(s) du cache après 1 itération, en précisant les index (seules 3 cases sont représentées), et en mettant dans les parties « Mot <x> » les éléments de X ou Y présents (exemple : X[12] ou Y[7]).
Note : dans le code assembleur produit, la lecture de X[i] a lieu avant celle de Y[i + 1].

Index	TAG	Mot 7	Mot 6	Mot 5	Mot 4	Mot 3	Mot 2	Mot 1	Mot 0
. 127	0x08008	X[1023]	X[1022]	X[1021]	X[1020]	X[1019]	X[1018]	X[1017]	X[1016]
. 255	0x08008	Y[1023]	Y[1022]	Y[1021]	Y[1020]	Y[1019]	Y[1018]	Y[1017]	Y[1016]

B4 (1 point). Calculer, en le justifiant, le nombre de miss de données rencontrés lors de l'exécution de cette fonction.

Chaque tableau est lu entièrement (sauf Y[0]) et il n'y a pas de miss de conflit (aucune ligne n'est évincée). Il y a donc 1024 / 8 = 128 miss par tableau, soit 256 au total. A la fin de l'exécution de la fonction, les 256 lignes du cache sont remplies.

B5 (0.5 point). Donner le taux de miss sur le cache de données pour cette fonction.

Le taux de miss pour cette fonction est de 256 / 2047, soit environ 12.5%.

B6 (0.5 point). Peut-on simplement réduire le nombre de miss rencontrés lors de l'exécution ? Si oui, comment ? Si non, pourquoi ?

Non, car tous les miss rencontrés sont obligatoires : tous les lignes doivent être chargées au moins une fois, et chaque ligne n'est chargée qu'une fois.

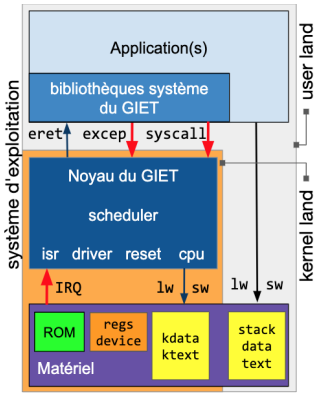
B7 (0.5 point). La compilation de ce code a produit une boucle de 11 instructions (qui met 11 cycles à s'exécuter en présence d'un système mémoire parfait), et des instructions avant la boucle qui mettent 4 cycles à s'exécuter. On néglige l'effet des miss sur le cache d'instructions. Calculer le nombre de cycles total pour l'exécution de la fonction si un miss de données coute 20 cycles (hors prologue et épilogue).

Nombre total de cycles : 4 + 11 * 1023 + 20 * 256 = 16 377

EXERCICE C : GIET (5 points) **Numéro d'anonymat :**

C1) (1 point) Le système d'exploitation GIET est composé d'une bibliothèque système et d'un noyau.
Que trouve-t-on dans les bibliothèques système ? Donnez deux exemples.

Les bibliothèques contiennent des fonctions d'accès aux ressources de la machine.
Par exemple :
- printf() pour afficher une chaine de caractère
- getc() pour lire un caractère



C2) (1 point) Le noyau gère 3 types d'événements : les appels système, les interruptions et les exceptions. Donnez deux exemples de chaque type de demande. De quoi est composé le vecteur de syscall ?

syscall :
- write, read, set_timer_period

interruption
- interruption du timer, du TTY, de l'IOC

Exception
- division par 0, lecture non-alignée, instruction illégale, …

Le vecteur de syscall est un tableau de pointeur de fonctions.
Chaque fonction implémente un appel système.

C3) (0.5 point) Le code du noyau s'exécute dans un mode d'exécution privilégié du processeur : le mode *kernel*. Que permet l'exécution en mode *kernel* qui est impossible en mode *user* (non-*kernel*) ? Est-ce que l'application peut accéder la mémoire sans passer par le noyau ?

Le mode kernel permet d'accéder à tous l'espace d'adressage.
Il permet aussi l'exécution des instructions mfc0 et mtc0.

Evidemment l'application accède directement aux segments d'adresse mappés en mémoire qui sont dans la partie autorisée à l'utilisateur < 0x80000000

C4) (1 point) Le scheduler (ordonnanceur) est le service du noyau en charge des changements de contexte des tâches. Dans le cas du GIET, quand ce service est-il appelé ? Que signifie changement de contexte des tâches ?

Le scheduler est appelé lors des interruptions du TIMER.

Changement de contexte signifie
- choix d'une nouvelle tâche à exécuter,
- sauvegarde de l'état du processeur de la tâche courante,
- restauration de l'état du processeur de la tâche future.

C5) (1 point) Sur une machine avec deux cœurs de calcul, chaque cœur dispose de deux caches (données et d'instructions) ayant une stratégie *write-through* sans cohérence. Il y a une perte de cohérence de cache lorsqu'un cache contient une ligne dont le contenu n'est plus valide. Expliquez en détails comment cela peut arriver.

Cela peut arriver si les deux cœurs lisent tous les deux la même ligne.
Ils auront donc tous les deux la ligne dans les cases de leur cache de données.
Si l'un des deux modifie un mot dans cette ligne, son propre cache sera mise à jour, la mémoire également grâce à la stratégie *write-through*. Le cache de l'autre processeur ne sera pas mise-à-jour entraînant une perte de cohérence pour la ligne concernée.

C6) (0.5 point) Si on veut que deux tâches *user* partagent un même tableau en mémoire, les deux peuvent lire et écrire dans le tableau pour s'échanger des données. Puisque les caches L1 ne garantissent pas la cohérence matérielle de leur contenu, comment éviter le problème de cohérence ?

Il faut simplement demander à invalider le contenu des caches avant de lire les lignes partagées.

EXERCICE D Accès aux périphériques (5 points) **Numéro :**

0n s'intéresse dans cette partie aux opérations d'entrée/sortie permettant à un programme utilisateur d'accéder aux périphériques, tels qu'un terminal écran/clavier, un contrôleur réseau, ou un dispositif de stockage externe (disque magnétique ou clé USB).

D1) (1 point) Dans le cas du GIET, quels sont les trois arguments qu'un programme utilisateur doit transmettre au système d'exploitation lorsqu'il effectue un appel système pour demander un accès au disque en lecture ou en écriture ?

Le disque étant un périphérique « orienté blocks », le GIET transfert toujours un nombre entier de blocs de 512 octets entre le disque et la mémoire. Il faut donc trois arguments :

1. buffer : adresse du tampon mémoire qui doit recevoir les données (pour une lecture sur disque), ou qui contient les données (pour une écriture sur disque).
2. lba : numéro du premier bloc à transférer sur le disque.
3. count : Nombre de blocs à transférer.

D2) (1 point) Expliquez pourquoi un programme utilisateur ne peut pas directement envoyer une commande d'entrée ou de sortie à un périphérique, sans passer par un appel système. Quelles sont les deux vérifications que doit effectuer le système d'exploitation avant de démarrer une opération d'entrée sortie de lecture ou d'écriture vers un fichier sur disque.

Un ordinateur peut exécuter plusieurs programmes simultanément, pour le compte de plusieurs utilisateurs différents. Or les périphériques sont des ressources partagées par les différents programmes, et il faut éviter qu'un programme A lise ou écrive des données appartenant au programme B.
Dans le cas d'un disque, le système doit donc protéger les utilisateurs les uns contre les autres, et il doit également se protéger lui-même.

1. il doit vérifier que le fichier appartient bien au programme demandeur.
2. Il doit vérifier que le tampon mémoire (source pour une écriture) ou destination pour une lecture) n'appartiennent pas à la zone protégée de l'espace adressable, réservée au système d'exploitation.

D3) (1 point) Comment le système d'exploitation déclenche-t-il une opération d'entrée/sortie sur un périphérique particulier ? Pour un type de périphérique donné, il existe évidemment de très nombreux fabricants. Deux contrôleurs de disque fournissent les mêmes services, mais utilisent des commandes différentes . Comment un système d'exploitation généraliste comme LINUX s'adapte-t-il à cette grande variété de périphériques matériels ?

Tous les périphériques possèdent des registres adressables dans lesquels le processeur peut aller écrire pour définir les paramètres d'une opération d'entrée/sortie, et déclencher son exécution. Tous les périphériques sont donc des cibles pour recevoir les commandes envoyées par l'OS.
Pour permettre au système d'exploitation de s'adapter a différents matériels, chaque fabricant de périphérique fournit un pilote logiciel adapté à ce périphérique. Ce pilote contient le code qui doit être exécuté pour écrire les bonnes valeurs dans les bons registres, et démarrer l'opération d'entrée sortie souhaitée (une fonction à appeler pour chaque type d'opération).

D4) (1 point) Une opération d'entrée/sortie peut avoir une durée d'exécution très variable (quelques milliers à quelques millions de cycles), suivant le type et l'état courant du périphérique. L'exécution du programme demandeur est donc généralement suspendue par le système d'exploitation, en attendant la fin du transfert demandé, pour ne pas gaspiller inutilement du temps processeur. Comment le périphérique signale-t-il à l'OS qu'il a terminé le transfert ? Comment le programme utilisateur demandeur est-il re-activé ?

Pour signaler à l'OS la fin de l'opération d'entrée/sortie, un périphérique doit envoyer un signal logiciel à l'OS, en écrivant une valeur dans une case mémoire (boîte aux lettres) réservée pour cette signalisation.
Comme tous les périphériques n'ont pas forcément de capacité DMA pour écrire en mémoire, la technique utilisée consiste à déclencher une interruption : le périphérique interrompt le programme pour forcer le processeur à exécuter une fonction (Interrupt Service Routine) qui effectue cette écriture dans la boîte aux lettres.
Cette boîte aux lettres est régulièrement inspectée par l'OS, qui peut alors re-activer le programme qui avait demandé l'opération d'entrée/sortie.

D5) (1 point) Puisque chaque périphérique matériel possède sa propre ligne d'interruption, le nombre de lignes d'interruption sortant des périphériques est beaucoup plus grand que le nombre d'interruptions entrant dans le processeur (certains processeurs n'ont qu'une seule ligne entrante). Lorsqu'un processeur est interrompu, et se branche au gestionnaire d'interruptions de l'OS, comment celui détermine-t-il quel périphérique est la source de l'interruption, et quelle ISR doit être exécutée ?

L'OS utilise un composant matériel particulier appelé « Contrôleur d'interruption », qui rassemble les N lignes d'interruptions des N périphériques, et qui effectue un OU logique entre ces N valeurs, pour envoyer un seul bit au processeur. Ce composant contient un registre adressable, qui peut être lu par le gestionnaire d'interruption et contient le numéro (compris entre 0 et N-1) de la ligne d'interruption active la plus prioritaire.
L'OS utilise ce numéro pour rechercher dans une table de saut l'adresse de la routine d'interruption associée à la ligne d'interruption sélectionnée par le contrôleur d'inerruption. Cette table de saut s'appelle le vecteur d'interruption, et doit être initialisée par l'OS lors du démarrage de la machine, en fonction du nombre de périphériques disponibles.