

L'objet de cette étude est de comparer la performance de deux réalisations de Mips R3000 pour des algorithmes de recherche de donnée.

La première est la réalisation classique sur un pipeline de 5 étages présenté en cours. Cette réalisation est appelée **Mips**.

La seconde, appelée **P9**, est un superpipeline à 9 étages : IFC1, IFC2, IFC3, DEC, EXE, MEM1, MEM2, MEM3 et WBK. Dans cette réalisation, les accès à la mémoire s'effectuent en trois cycles. Les cycles DEC, EXE et WBK sont similaires aux cycles DEC, EXE et WBK de la réalisation Mips. La fréquence d'horloge de **P9** est 3 fois plus élevée que la fréquence de la réalisation Mips.

Exercice I :

On peut utiliser une liste chaînée pour stocker un ensemble de données. Dans cet exercice on considère que les données sont des nombres entiers relatifs. L'objectif de cet exercice est de comparer la performance d'exécution d'un algorithme de recherche de donnée dans une liste chaînée.

Chaque nœud d'une liste chaînée est une structure de donnée appelée `node_1`. Cette structure comporte 2 champs.

```
struct node_1
{
    int          VALUE ;
    struct node_1 *NEXT ;
};
```

Le champ `VALUE` contient un entier. Le champ `NEXT` est le pointeur sur l'élément suivant de la liste, une structure de type `node_1`. Le champ `NEXT` du dernier élément de la liste pointe sur lui-même.

En mémoire chaque champ occupe un mot. Une structure `node_1` occupe donc 8 octets.

VALUE	adresse X
NEXT	adresse X+4

La fonction de recherche a comme arguments un pointeur sur une liste (une structure `node_t`) et un entier. Elle renvoie `NULL` si la donnée n'a pas été trouvée ou une valeur différente si la valeur a été trouvée.

```
struct node_1 *Search_1 (struct node_1 *list, int val)
{
    if (list != NULL)
    {
        while (list != list->NEXT)
        {
            if (val == list->VALUE)
                break;

            list = list->NEXT;
        }

        if (val != list->VALUE)
            list = NULL;
    }
    return (list);
}
```

Le code de la boucle principale de Search_1 en assembleur Mips (non pipeline) est le suivant. On suppose qu'à l'entrée de la boucle le registre R5 contient le pointeur de list (l'adresse de la structure node_1) et le registre R6 la valeur val.

```

Loop :      Add      r7 , r0 , r5
            Lw       r10, 0 (r7 )      ; list->VALUE
            Beq      r10, r6 , End_Loop
            Lw       r5 , 4 (r7 )
            Bne      r5 , r7 , Loop

```

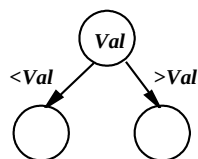
- 1- Modifier le code de la boucle de Search_1 pour qu'il soit exécutable sur la réalisation **Mips**. Analyser l'exécution de cette boucle à l'aide d'un schéma simplifié. On suppose que le branchement Beq ne réussit jamais.

Calculer le nombre moyen de cycles par itération. Calculer le CPI et le CPI-utile.

- 2- Même question pour **P9**.
- 3- Pour la réalisation **Mips**, modifier l'ordre des instructions de Search_1 afin d'éviter au maximum les cycles perdus. Calculer le nombre moyen de cycles par itération.
- 4- Même question pour **P9**
- 5- Pour la réalisation **Mips**, dérouler une fois le code de Search_1 puis optimiser afin d'éviter au maximum les cycles perdus. Calculer le nombre moyen de cycles par itération.
- 6- Même question pour **P9**

Exercice II :

On peut stocker un ensemble de données à l'aide d'un arbre binaire à condition d'établir une relation d'ordre entre les données. Dans cet exercice on considère que les données sont des nombres entiers relatifs. L'objectif de cet exercice est de comparer la performance d'exécution d'un algorithme de recherche de donnée dans un arbre binaire.



VALUE	adresse X
SUB [0]	adresse X+4
SUB [1]	adresse X+8

Chaque noeud de l'arbre contient un entier. La valeur du noeud est la valeur de l'entier stocké dans le noeud. Le sous-arbre droit d'un noeud représente l'ensemble des noeuds dont la valeur est supérieure à la valeur du noeud. Le sous-arbre gauche représente l'ensemble des noeuds dont la valeur est inférieure à la valeur du noeud.

Un noeud de l'arbre est une structure de donnée appelée node_t. Cette structure comporte 3 champs.

```

struct node_t
{
    int      VALUE ;
    struct node_t *SUB [2];
};

```

Le champ VALUE contient un entier. Les champs SUB[0] et SUB[1] sont des pointeurs sur une structure de type node_t. Le champ SUB[0] pointe sur le sous-arbre droit et le champ SUB[1]

sur le sous-arbre gauche. Le champs SUB[0] ou SUB[1] sont NULL si le sous-arbre droit ou le sous-arbre gauche sont vides.

En mémoire chaque champ occupe un mot. Une structure node_t occupe donc 12 octets.

La fonction de recherche a comme arguments un pointeur sur un arbre (une structure node_t) et un entier. Elle renvoie NULL si la donnée n'a pas été trouvée ou une valeur différente si la valeur a été trouvée.

On peut écrire la fonction de recherche de deux manières différentes :

```
struct node_t *Search_t1 (struct node_t *tree, int val)
{
    int less;

    while (tree != NULL)
    {
        if (val == tree->VALUE)
            break;
        if (val < tree->VLAUE)
            less = 1;
        else
            less = 0;

        tree = tree->SUB [less];
    }
    return (tree);
}
```

```
struct node_t *Search_t2 (struct node_t *tree, int val)
{
    while (tree != NULL)
    {
        if (val == tree->VALUE)
            break;
        if (val < tree->VLAUE)
            tree = tree->SUB [1];
        else
            tree = tree->SUB [0];
    }
    return (tree);
}
```

On cherche à savoir quelle est la version la plus performante et sur quelle réalisation.

Le code de la boucle principale de Search_t1 en assembleur Mips (non pipeline) est le suivant. On suppose qu'à l'entrée de la boucle le registre R5 contient le pointeur de tree (l'adresse de la structure node_t) et le registre R6 la valeur val.

```
Loop :      Lw      r10, 0 (r5 )          ; tree->VALUE
            Beq     r10, r6 , End_Loop
            Slt     r11, r6 , r10
            Sll     r11, r11, 2
            Addu    r11, r11, r5
            Lw      r5 , 4 (r11)
            Bne     r5 , r0 , Loop
```

Avec les mêmes hypothèses, le code de la boucle principale de Search_t2 en assembleur Mips (non pipeline) est le suivant.

```

Loop   :      Lw      r10, 0 (r5 )           ; tree->VALUE
          Beq      r10, r6 , End_Loop
          Slt      r11, r6 , r10
          Beq      r11, r0 , Else
          Lw      r5 , 8 (r5 )
          J        Endif
Else   :      Lw      r5 , 4 (r5 )
Endif  :      Bne    r5 , r0 , Loop

```

- 7- Modifier le code de la boucle de Search_t1 pour qu'il soit exécutable sur la réalisation **Mips**. On suppose que le branchement Beq r10, r6, End_Loop ne réussit jamais. Analyser l'exécution de cette boucle à l'aide d'un schéma simplifié.

Calculer le nombre moyen de cycles par itération. Calculer le CPI et le CPI-utile.

- 8- Pour la réalisation **Mips**, modifier l'ordre des instructions de Search_t1 afin d'éviter au maximum les cycles perdus. Calculer le nombre moyen de cycles par itération.
- 9- Pour la réalisation **Mips**, optimiser le code de Search_t1 à l'aide de la technique "pipeline logiciel" afin d'éviter au maximum les cycles perdus. On découpera le traitement en **deux** étages. Calculer le nombre moyen de cycles par itération.
- 10- Modifier le code de la boucle de Search_t2 pour qu'il soit exécutable sur la réalisation **Mips**. On suppose que le branchement Beq r10, r6, End_Loop ne réussit jamais. Calculer le nombre moyen de cycles par itération en supposant que le branchement Beq r11, r0, Else échoue dans 50% des cas. Il n'est pas demandé de faire un schéma simplifié mais d'indiquer les dépendances qui provoquent des cycles de gel. Calculer le CPI et le CPI-utile.
- 11- Pour la réalisation **Mips**, modifier l'ordre des instructions de Search_t2 afin d'éviter au maximum les cycles perdus. Calculer le nombre moyen de cycles par itération.
- 12- Pour la réalisation **Mips**, optimiser le code de Search_t2 à l'aide de la technique "pipeline logiciel" afin d'éviter au maximum les cycles perdus. On découpera le traitement en **deux** étages. Calculer le nombre moyen de cycles par itération.
- 13- Explique pourquoi dans Search_t1 et dans Search_t2, en utilisant la technique d'optimisation "pipeline logiciel", on ne peut pas découper le traitement en plus de deux étages.