Architecture des processeurs et optimisation
Novembre 2011 **Documents autorisés - Durée 2h**

utiliser impérativement les feuilles quadrillées fournies pour les schémas simplifiés

Exercice I :

Un élément d'une liste chaînée est une structure qui contient 2 champs. Le premier, `NEXT`, est un pointeur sur l'élément suivant. Le second, `DATA`, est un pointeur sur la donnée. Chacun des champs est donc un pointeur (une adresse sur 32 bits).

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

La fonction `Reverse` permet d'inverser l'ordre des éléments dans une liste chaînée.

```
struct chain *Reverse (struct chain *list)
  {
  struct chain *prev = NULL;
  struct chain *next = NULL;

  while (list != NULL)
    {
    next       = list->NEXT;
    list->NEXT = prev    ;
    prev       = list    ;
    list       = next    ;
    }
  return (prev);
  }
```

1- Proposer un code en assembleur Mips (pipeline à 5 étages étudié en cours) pour la boucle principale de la fonction `Reverse` en supposant qu'à l'entrée de la boucle, le registre R4 contient le pointeur sur la liste chaînée (`list`).

2- Analyser à l'aide d'un schéma simplifié l'exécution d'une itération de la boucle principale de la fonction `Reverse`. Calculer le nombre moyen de cycles par itération, le CPI et le CPI-utile.

Exercice II :

Une image est composée d'un ensemble de pixels organisés sous forme d'un tableau. Pour une image en 'niveaux de gris', chaque pixel est codé sur un octet. La valeur 0 représente la couleur noire, la valeur 255 la couleur blanche. Les valeurs comprises entre 0 et 255 représentent les différentes nuances de gris. Une fonction de traitement d'images consiste à appliquer un certain algorithme à un tableau de pixels.

On souhaite étudier et comparer la performance d'exécution de la fonction `Seuil` sur deux réalisations de l'architecture Mips.

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 pipeline à 9 étages `IFC1, IFC2, DEC1, DEC2, EXE1, EXE2, MEM1, MEM2, WBK` (vue en TD). Le découpage en étages de ce pipeline a été obtenu à partir

du pipeline de la réalisation Mips. Chacun des étages `IFC`, `DEC`, `EXE` et `MEM` a été divisé en deux étages. Le calcul de l'adresse de l'instruction suivante s'achève dans l'étage `DEC2`. On rappelle que dans cette réalisation, il y a 16 bypass et qu'il n'y a pas de bypass vers l'étage `MEM1`.

La fonction `Seuil` vérifie si la valeur d'un pixel est supérieure à un certain seuil. Dans ce cas, le pixel est remplacé par un pixel blanc (code 255) et dans le cas contraire par un pixel noir.

```
void Seuil (unsigned char *img  ,
            unsigned int   size ,
            unsigned char  seuil )
  {
  for (i=0 ; i!=size ; i++)
     {
    if (img [i] >= seuil)
      img [i] = 255;
    else
      img [i] =   0;
    }
  return ;
  }
```

On dispose de deux versions pour le code de la boucle principale de la fonction `Seuil` en assembleur Mips-32 (**non pipeline**). Conformément aux conventions utilisées par le compilateur Gcc, on suppose qu'à l'entrée de la boucle :

- le registre R4 pointe sur le tableau de pixels
- le registre R5 contient le nombre de pixels de l'image
- le registre R6 contient la valeur du seuil

De plus, le registre R8 contient l'adresse immédiatement après la fin du tableau de pixels (`img + size`) et le registre R7 la valeur 255.

Version 1 :

```
_For1 :          Lbu     r10, 0 (r4 )
                 Sltu    r11, r10, r6
                 Sb      r0 , 0 (r4 )
                 Bne     r11, r0, _Endif
                 Sb      r7 , 0 (r4 )
_Endif :         Addiu   r4 , r4 , 1
                 Bne     r4 , r8 , _For1
```

Version 2 :

```
_For2 :          Lbu     r10, 0 (r4 )
                 Sltu    r11, r10, r6
                 Addiu   r11, r11, -1
                 Sb      r11, 0 (r4 )
                 Addiu   r4 , r4 , 1
                 Bne     r4 , r8 , _For2
```

Dans tout l'exercice, on suppose qu'en moyenne 50% des pixels sont en dessous du seuil et que cette répartition est uniforme en tout point de l'image.

3- Pour la Version 1, modifier le code de la boucle pour qu'il soit exécutable sur la réalisation **Mips**. Analyser l'exécution de cette boucle à l'aide d'un schéma simplifié dans l'hypothèse où le branchement échoue. Calculer le nombre moyen de cycles par itération. Calculer le CPI et le CPI-utile.

4- Même question pour **P9**.

5- Pour la Version 2, modifier le code de la boucle pour qu'il soit exécutable sur la réalisation **Mips**. 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.

6- Même question pour **P9**.

7- Pour chacune des deux réalisations, proposer un code optimisé de la Version 1 en modifiant l'ordre des instructions. Il n'est pas nécessaire de faire un schéma mais d'indiquer simplement les cycles de gel sur le code après optimisation. Pour chacune des deux réalisations, calculer le nombre moyen de cycles par itération. Calculer le CPI et le CPI-utile.

8- Même question pour la Version 2.

9- Pour chacune des deux réalisations, dérouler une fois le code de la Version 1 (traitement de 2 éléments à chaque itération) et optimiser. Il n'est pas nécessaire de faire un schéma mais d'indiquer simplement les cycles de gel sur le code après optimisation. Pour chacune des deux réalisations, calculer le nombre moyen de cycles par itération, le CPI et le CPI-utile.

10- Même question pour la Version 2.