ES102/PC9 : énoncé

Les objectifs pédagogiques de cette PC sont de découvrir :

- l'exécution de quelques fragments de code sur un microprocesseur de type MIPS et leur expression, voire compilation, dans le langage *assembleur* du même nom ;
- les solutions et artifices élaborés pour répondre efficacement à certains besoins logiciels (structuration des données, contrôle de flux, démarche procédurale, etc.) ;
- certains compromis matériels/logiciels pour obtenir la meilleure efficacité de calcul.

1) Multiplication logicielle

1a) Décaler de 1 bit vers la gauche un nombre entier stocké dans un registre 32 bits revient à le multiplier par 2, sous certaines conditions... Lesquelles ? On les supposera vérifiées par la suite.

La séquence d'instructions MIPS ci-dessous est présentée instructions à gauche et explications à droite. Chaque instruction vient avec une liste d'opérandes : le registre destination, le premier registre source et enfin le deuxième registre source ou un paramètre. Trois sortes d'instructions apparaissent ci-dessous : *mv* (move) pour copie, *add* pour addition, et *sll* (shift left logical) pour décalage vers la gauche, avec 0 entrant à droite. Les registres utilisés ont pour nom *s1* et *t1*.

`mv $t1,$s1`	t1 ← $s1
`sll $t1,$t1,2`	t1 ← $t1 << 2 (décalage de 2 bits vers la gauche)
`add $t1,$t1,$s1`	t1 ← $t1+$s1
`sll $t1,$t1,1`	t1 ← $t1 << 1
`add $t1,$t1,$s1`	t1 ← $t1+$s1
`sll $t1,$t1,1`	t1 ← $t1 << 1
`add $t1,$t1,$s1`	t1 ← $t1+$s1

1b) Que réalise le programme ci-dessus ?

1c) Le simplifier par recodage de Booth. L'instruction de soustraction est nommée/notée `sub`.

1d) En réalité, la première instruction, « `mv $t1,$s1` » est réalisée par « `addi $t1,$s1,0` », où *addi* est l'instruction d'addition immédiate. Pourquoi ?

2) Boucle while et tableau de données

Les registres constituent une ressource rare : une donnée stockée en mémoire n'est amenée dans le banc de registres que lorsqu'elle va être utilisée par l'ALU pour un calcul. Ensuite, une autre prendra sa place. De même, un résultat de calcul ne devant pas être réutilisé prochainement sera renvoyé en mémoire pour libérer de la place dans le banc de registres. Ces nécessités sont évidentes lorsqu'on fait des calculs sur les données d'un tableau, dont la taille est généralement bien plus grande que la vingtaine de registres disponibles en pratique pour faire des calculs.

Les instructions MIPS permettant les échanges de données 32 bits entre mémoire et banc de registres sont respectivement notées :

- lw (*load word*) pour un chargement, impliquant une lecture de la mémoire
- sw (*store word*) pour un stockage, impliquant une écriture en mémoire

Lecture ou écriture nécessitent la présentation d'une adresse à la mémoire. La donnée en jeu est souvent un élément d'un tableau dont l'adresse de base aura été préalablement chargée dans un registre. L'adresse de l'élément considéré est alors calculée par l'ALU, à partir de son indice. Soit `data[]` un tableau d'entiers déclaré dans un programme en langage C.

2a) Comment est compilée l'instruction « `i=data[2];` » en assembleur MIPS ?

On continue ci-dessous à accéder au tableau `data[]`, mais les indices des éléments manipulés sont désormais variables, exigeant un calcul à la volée de leur adresse (pour lequel le mécanisme de valeur immédiate n'est d'aucune aide). Un compilateur C a transformé la ligne suivante :

```
while (data[i] == k) { i = i+j; }
```

en la séquence suivante d'instructions en langage assembleur MIPS :

`Loop :`	`sll $t1,$s1,2`	t1 ← $s1 << 2 (décalage de 2 bits vers la gauche)
	`add $t1,$t1,$s4`	t1 ← $t1 + $s4
	`lw  $t0,$t1+0`	t0 ← contenu de la mémoire à l'adresse figurant dans t1
	`bne $t0,$s3,Exit`	si $t0≠$s3, sauter jusqu'à Exit (**b**ranch if **n**ot **e**qual)
	`add $s1,$s1,$s2`	s1 ← $s1 + $s2
	`j Loop`	sauter (retourner) jusqu'à l'instruction étiquetée Loop
`Exit :`	`------`	instruction non précisée

Les instructions ci-dessus, chacune codées sur 32 bits, sont stockées l'une derrière l'autre en mémoire d'instructions. « Loop » et « Exit » sont des étiquettes attachées chacune à une instruction, vers lesquelles d'autres instructions peuvent pointer. C'est une commodité de notation pour désigner l'adresse en mémoire de l'instruction étiquetée.

2b) Ci-dessus, que contiennent les registres *s1*, *s2*, *s3*, *s4* ? Qui sert de pointeur sur `data[i]` ?

2c) Comment appelle-t-on la dernière instruction du segment de code ci-dessus ? Quelles particularités de fonctionnement du processeur exige-t-elle ? Quelle est sa « portée » ?

2d) Le segment de code ci-dessus comporte également une instruction *bne*. De quoi s'agit-il ? Quelles particularités de fonctionnement du processeur exige-t-elle ? Quelle est sa « portée » ? Comment l'adresse correspondant à l'étiquette Exit est-elle calculée ?

2e) Un compilateur plus performant produit le nouveau code ci-dessous. Expliquer et apprécier.

```
         j Begin
Loop :   add $s1,$s1,$s2
Begin :  sll $t1,$s1,2
         add $t1,$t1,$s4
         lw  $t0,$t1+0
         beq $t0,$s3,Loop
```

3) Fonctions, passage de paramètres, appels imbriqués

Parmi les 32 registres disponibles sur processeur MIPS I, il y en a 6 servant spécifiquement en tant qu'**a**rguments et **v**aleurs de fonctions : *a0*, *a1*, *a2*, *a3*, *v0* et *v1*. Le code suivant, qui utilise *a0*, *a1* et *v0*, est celui d'une fonction min(), qui retourne le minimum de ses deux arguments.

```
Min2 : sub $t0,$a0,$a1
       bgtz $t0,Arg1     bgtz : branch if greater than zero
       mv $v0,$a0
       j Fin
Arg1 : mv $v0,$a1
 Fin : jr $ra
```

3a) Hormis la dernière instruction, quelle structure de contrôle reconnaît-on ?

L'appel de la fonction min() apparaîtra en un ou plusieurs autres endroits du programme, chargeant d'abord les registres *a0* et *a1* avec les deux arguments souhaités, puis exécutant un *jump and link*, comme suit :

```
mv $a0,...
mv $a1,...
jal Min2
```

3b) Quels sont les rôles des instructions `jal` et `jr` ci-dessus ? Que signifie la notation *`ra`* ?

3c) Le code MIPS ci-dessous semble être celui d'une fonction min3() à 3 arguments. Expliquer en quoi. Toutefois, ce code présente un grave dysfonctionnement. Lequel ?

```
Min3 : jal Min2
       mv $a0,$v0
       mv $a1,$a2
       jal Min2
       jr $ra
```

Dans le cadre d'un appel imbriqué de fonction, tout registre utilisé à la fois par la fonction appelante et la fonction appelée doit être sauvegardé en mémoire juste avant ou juste après l'appel, pour être restauré ensuite. Une pile LIFO (Last In, First Out) est une structure de données efficace pour réaliser de telles sauvegardes et restaurations, puisque tout appel postérieur à un autre doit effectuer son retour avant lui. En pratique, on utilise la queue de la mémoire de données pour réaliser une pile LIFO logicielle : on la remplit en partant de l'adresse la plus élevée. Le sommet de la pile (correspondant à l'adresse la plus petite) est pointé par la valeur du registre spécifique nommé *sp* (stack pointer).

3d) Doter les codes des fonctions min() et min3() des bonnes sauvegardes dans la pile.

3e) Comment passe-t-on des paramètres à une fonction qui en compte plus que 4 ?

3f) Comment passe-t-on des paramètres à une fonction qui peut en compter un nombre variable, telle que `printf()` ?