

## 1. Contrôle préalable

Cette partie est conçue comme une vérification pour vous permettre de déterminer si vous comprenez les concepts abordés en cours ou non. Veuillez répondre par « Vrai » ou « Faux » aux questions suivantes et inclure une explication :

- 1.1. Après avoir appelé une fonction dans MIPS. Au retour de cette fonction, les registres \$t0-\$t9 peuvent avoir été modifiés lors de l'exécution de la fonction, mais les valeurs des registres \$v0 et \$v1 seront conservées.
- 1.2. Si \$a0 est un pointeur vers le début d'un tableau **x** en mémoire. L'instruction MIPS « **lw \$s0, 4(\$a0)** » chargera toujours la valeur de **x[1]** en **\$s0**.
- 1.3. En supposant que les entiers sont sur 4 octets, l'addition du caractère ASCII 'd' à l'adresse d'un tableau d'entier vous permettrait d'obtenir l'élément à l'index 25 de ce tableau (en supposant que le tableau est assez grand).
- 1.4. L'instruction « **j label** » fait exactement la même chose que l'instruction « **jal label** ».

## 2. Instructions de base

Pour référence, voici quelques instructions de base dans MIPS pour effectuer des opérations arithmétiques ou des accès mémoire (Remarque : « **rs** » = argument registre 1, « **rt** » = argument registre 2, et « **rd** » = registre de destination) :

[inst]	[registre rd], [registre rs], [registre rt]
<b>add</b>	Additionne les arguments et sauvegarde le résultat dans le registre rd.
<b>xor</b>	Effectue un OU eXclusif entre les arguments, résultat dans rd.
<b>sllv</b>	Décalage logique à gauche de rs par la quantité rt, résultat dans rd.
<b>srlv</b>	Décalage logique à droite de rs par la quantité rt, résultat dans rd.
<b>srav</b>	Décalage arithmétique à droite de rs par la quantité rt, résultat dans rd.
<b>slt/u</b>	Si $rs < rt$ alors $rd = 1$ sinon $rd = 0$ . (slt pour la comparaison non signée).

[inst]	[registre], offset([registre contenant une adresse])
<b>sw</b>	Sauvegarde le contenu du registre dans « adresse + offset » en mémoire.
<b>lw</b>	Copie dans le registre le contenu dans « adresse + offset » en mémoire.

[inst]	[registre rt], [registre rs], label
<b>beq</b>	Si $rs == rt$ alors se brancher (c.-à-d. sauter) à label.
<b>bne</b>	Si $rs != rt$ alors se brancher à label.

[inst]	label
<b>j</b>	Saute à label.
<b>jal</b>	Sauvegarde l'adresse de l'instruction suivante dans \$ra, puis saute à label.

Vous pouvez également vérifier dans la fiche MIPS qu'il y'a un « i » à la fin de certaines instructions, telles que **addi**, **slti**, etc. Cela signifie que l'opérande « [registre rt] » devient une étiquette « immédiat » ou un entier au lieu d'un registre. Notez que la taille (nombre maximum de bits) d'un « immédiat » dans une instruction MIPS dépend du type de cette instruction (type **I** ou type **J**).

Soit un tableau déclaré en C :

```
int arr[] = {1,2,3,4,5,6,0};
```

Supposons que le registre **\$s0** contient l'adresse de l'élément à l'indice 0 dans le tableau arr. Supposons également que la taille des entiers dans notre système est égale à 4 octets. Que font les instructions suivantes dans les extraits de code MIPS ci-dessous (notez que certains codes comportent des erreurs) :

```
2.1.  lw $t0, 12($s0)
      add $t1, $t0, $s0
      sw $t0, 4($t1)
```

- 2.2.    `addiu $s1, $s0, 27`  
          `lh $t0, -3($s1)`
- 2.3.    `addiu $s1, $s0, 24`  
          `lh $t0, -3($s1)`
- 2.4.    `addiu $t0, $0, 12`  
          `sw $t0, 6($s0)`
- 2.5.    `addiu $t0, $0, 8`  
          `sw $t0, -4($s0)`
- 2.6.    `addiu $s1, $0, 10`  
          `addiu $t0, $0, 6`  
          `sw $t0, 2($s1)`

Dans les extraits de code ci-dessus, quelles autres instructions pourraient être utilisées à la place de chaque « lecture depuis / sauvegarde vers » la mémoire pour supprimer les erreurs d'alignement ? (Modifiez UNIQUEMENT l'instruction problématique).

### 3. Conversion C ↔ MIPS

Convertissez instruction pour instruction depuis (resp. vers) le langage C vers (resp. depuis) l'Assembleur MIPS.

	C	MIPS
3.1.	<pre>// \$s0 -&gt; a, \$s1 -&gt; b // \$s2 -&gt; c, \$s3 -&gt; z int a = 4, b = 5, c = 6, z; z = a + b + c + 10;</pre>	

3.2.	<pre>// \$s0 -&gt; int *p = intArr; // \$s1 -&gt; a; *p = 0; int a = 2; p[1] = p[a] = a;</pre>	
3.3.	<pre>// \$s0 -&gt; a, \$s1 -&gt; b int a = 5, b = 10; if(a + a == b) {     a = 0; } else {     b = a - 1; }</pre>	
3.4.		<pre>addi \$s0, \$0, 0 addi \$s1, \$0, 1 addi \$t0, \$0, 30 loop:     beq \$s0, \$t0, exit     add \$s1, \$s1, \$s1     addi \$s0, \$s0, 1     j loop exit:</pre>
3.5.	<pre>// \$s0 -&gt; n, \$s1 -&gt; sum // supposons n &gt; 0 au début for(int sum = 0; n &gt; 0; n--) {     sum += n; }</pre>	

## 4. Tableaux et Listes dans MIPS

Soit un tableau déclaré en C :

```
int arr[] = {3,1,4,1,5,9};
```

Le premier élément dans ce tableau se trouve à l'adresse mémoire 0xBFFFFFF0. Soit aussi la déclaration suivante d'une liste chaînée :

```
struct ll {
    int val;
    struct ll* next;
}
struct ll* lst;
```

Le premier nœud dans la liste chaînée se trouve à l'adresse mémoire 0xABCD0000. Le champ next du dernier nœud de la liste chaînée `lst` contient la valeur NULL (c-à-d. la valeur 0 en 32 bits).

Supposons maintenant que le registre `$s0` contient l'adresse du tableau `arr` (donc `$s0 = 0xBFFFFFF0`) ; et que le registre `$s1` contient l'adresse du premier élément de la liste chaînée `lst` (c-à-d. `$s1 = 0xABCD0000`). Supposons enfin, que la taille des entiers et des pointeurs dans notre système est de 4 octets.

Indiquez pour chaque question ci-dessous, ce que fait le bloc d'instructions MIPS (Les blocs de code dans les questions et leurs résultats sont indépendants les uns des autres).

```
4.1.      lw $t0, 0($s0)
           lw $t1, 8($s0)
           add $t2, $t0, $t1
           sw, $t2, 4($s0)
```

```
4.2.  loop: beq $s1, $0, end
           lw $t0, 0($s1)
           addi $t0, $t0, 1
           sw $t0, 0($s1)
           lw $s1, 4($s1)
           j loop
           end:
```

```
4.3.      add $t0, $0, $0
           loop: slti $t1, $t0, 6
              beq $t1, $0, end
              sll $t2, $t0, 2
              add $t3, $s0, $t2
              lw $t4, 0($t3)
              sub $t4, $0, $t4
              sw $t4, 0($t3)
              addi $t0, $t0, 1
              j loop
           end:
```

## 5. Convention d'appel dans MIPS

5.1. Comment pouvons-nous transmettre des arguments aux fonctions dans MIPS ?

5.2. Comment les valeurs sont-elles retournées par les fonctions dans MIPS ?



5.3. Qu'est-ce que `$sp` et comment doit-il être utilisé dans le contexte des fonctions MIPS ?

5.4. Si une fonction a besoin de conserver les valeurs de certains registres au travers de l'appel d'une autre fonction, quels registres la fonction appelante doit enregistrer avant l'appel et restaurer après retour de la fonction appelée ?

5.5. Quels registres la fonction appelée doit conserver (ou restaurer après utilisation) avant de retourner à la fonction appelante ?

5.6. Dans un programme sans bogues, quels registres sont garantis d'être les mêmes après un appel de fonction ? Quels registres ne sont pas garantis d'être les mêmes ?

## 6. Écrire des Fonctions MIPS

Écrire une fonction `sumSquare` dans MIPS qui calcule la somme ci-dessous.

$$n^2 + (n - 1)^2 + (n - 2)^2 + \dots + 1^2$$

Où  $n$  est un paramètre passé à la fonction. Si  $n$  n'est pas positif alors la fonction doit retourner la valeur zéro.

6.1. Commençons par implémenter le corps de la fonction : la sommation de quadratures. Nous respecterons les conventions d'appel entre fonctions appelante et appelée, alors dans quel registre le paramètre  $n$  est attendu ? Quels registres utiliser pour le calcul des quadratures et de la sommation ? Dans quel registre devrions-nous retourner le résultat de `sumSquare` ?

6.2. Comme `sumSquare` est une fonction appelée, nous devons nous assurer que, à son retour, elle ne modifie aucun des registres que la fonction appelante pourrait utiliser. En tenant compte de votre réponse pour la question ci-dessus, écrivez un prologue et un épilogue à la fonction `sumSquare` pour être en conformité avec les règles et conventions d'appels dans MIPS.