Architecture des Ordinateurs

2ème Année Licence Informatique

Introduction à MIPS

Travaux Dirigés #3

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 registres `$v0` et `$v1` seront préservés.
Faux. Les registres `$v0` et `$v1` sont souvent utilisés pour stocker les valeurs retournées par une fonction.

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 **`$s0`**.
Faux. Cela est seulement vrai pour les types de données qui sont quatre octets de large comme les entiers ou les réels (float). Pour les types de données comme (**`char`**) qui ne sont qu'un octet de large, **`4($a0)`** retournera l'élément à l'index 4 dans le tableau (ou d'autres données au-delà du tableau, selon la longueur du tableau).

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).
Vrai. Il n'y a pas de différence fondamentale entre un entier, un caractère et une adresse mémoire dans MIPS (ce sont tous des sacs de bits), il est donc possible de manipuler les données de cette façon. (cela n'est pas recommandé, cependant).

1.4. L'instruction « **`j label`** » fait exactement la même chose que l'instruction « **`jal label`** ».
Faux. Les deux instructions effectuent un saut vers l'adresse spécifiée par **`label`**, mais l'instruction « **`jal label`** » permet de sauvegarder dans le registre **`$ra`** l'adresse de l'instruction qui la succède.

2. Instructions de base

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

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

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

[inst]	[registre rt], [registre rs], label
beq	Si `rs == rt` alors aller à `label`.
bne	Si `rs != rt` alors aller à `label`.

[inst]	label
j	Aller à `label`.
jal	Sauvegarder l'adresse de l'instruction suivante $`ra` et aller 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 « `[registre rt]` » devient une étiquette « immédiat » ou un entier au lieu du registre. Notez que la taille (nombre maximum de bits) d'un « immédiat » dans une instruction donnée dépend de quel type d'instruction il s'agit.

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 dans l'extrait suivant de code MIPS (notez que certains codes comportent des erreurs) :

2.1.
```
lw $t0, 12($s0)                                  # 2) lb, lh
add $t1, $t0, $s0
sw $t0, 4($t1)          # arr[2] <- 4            # 2) sb, sh
```

```
2.2.    addiu $s1, $s0, 27
        lh $t0, -3($s1)          #   $t0 <- 0              # 2) lw, lb

2.3.    addiu $s1, $s0, 24
        lh $t0, -3($s1)          #   erreur d'alignement   # 2) lw, lb

2.4.    addiu $t0, $0, 12
        sw $t0, 6($s0)           #   erreur d'alignement   # 2) sh, sb

2.5.    addiu $t0, $0, 8
        sw $t0, -4($s0)          #   erreur d'index        # 2) sh, sb

2.6.    addiu $s1, $0, 10
        addiu $t0, $0, 6         #
        sw $t0, 2($s1)           #   arr[3] <- 6           # 2) sh, sb
```

Dans la question précédente, 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).

Voir la section ci-dessus pour savoir si `lb`, `sb`, `lh`, `sh`, `lw` ou `sw` peuvent fonctionner. D'autre part, comme toutes les valeurs entières dans le tableau `arr` peuvent être représentées avec moins de 8 bits, les instructions `lb` et `sb` peuvent toujours remplacer `lw/lh` et `sw/sh` sans perdre de données.

Si la valeur de l'immédiat dans l'instruction de lecture ou de sauvegarde modulo 4 est égale à 2 (c-à-d. imm % 4 == 2), alors `lh` et `sh` peuvent remplacer `lw` et `sw`.

Si la valeur de l'immédiat dans l'instruction de lecture ou de sauvegarde modulo 4 est égale à 0 (c-à-d. imm % 4 == 0), alors seulement `lw` et `sw` peuvent être utilisés.

3. Conversion C ↔ MIPS

Convertissez instruction pour instruction du/vers langage C vers/depuis l'Assembleur MIPS.

3.1.

C	MIPS

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	`addi $s0, $0, 4`
	`addi $s1, $0, 5`
	`addi $s2, $0, 6`
	`add  $s3, $s0, $s1`
	`add  $s3, $s3, $s2`
	`addi $s3, $s3, 10`

3.2.

```
// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;
```

```
    sw $0, 0($s0)
    addi $s1, $0, 2
    sw $s1, 4($s0)
    sll $t0, $s1, 2
    add $t0, $t0, $s0
    sw $s1, 0($t0)
```

3.3.

```
// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
}
else
{
    b = a - 1;
}
```

```
    addi $s0, $0, 5
    addi $s1, $0, 10
    add $t0, $s0, $s0
    bne $t0, $s1, else
    xor $s0, $0, $0
    j exit
else:
    addi s1, s0, -1
exit:
```

3.4.

```
// calcule s1 = 2^30
// la déclaration « int s1, s0; » est
// supposée déjà faite en dessus

s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
    s1 *= 2;
}
```

```
    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:
```

3.5.

```
// s0 -> n, s1 -> sum
// supposons n > 0 au début
for(int sum = 0; n > 0; n--) {
  sum += n;
}
```

```
    addi $s1, $0, 0
loop:
    beq $s0, $0, exit
    add $s1, $s1, $s0
    add $s0, $s0, -1
    j loop
exit:
```

4. Tableaux et Listes dans MIPS

Soit un tableau déclaré en C :

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

L'adresse mémoire du premier élément dans ce tableau se trouve à l'adresse `0xBFFFFF00`. Soit aussi la déclaration de la liste chaînée :

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

L'adresse mémoire du premier élément dans la liste se trouve à l'adresse `0xABCD0000.` Le champ `next` du dernier nœud de la liste chaînée `lst` contient la valeur `NULL` (c-à-d. la valeur `0x00000000` en 32 bits).

Supposons maintenant que le registre `$s0` contient l'adresse du tableau `arr` (donc `$s0 = 0xBFFFFF00`) ; et que le registre `$s1` contient l'adresse du premier élément de la liste chaînée `lst` (c.-à-d. `$s1 = 0xABCD0000`). Enfin, la taille des entiers et des pointeurs dans notre système est de 4 octets (32 bits).

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)
```
-> arr[1] = arr[0] + arr[2]

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:
```
-> incrémente le champ val de 1 dans chaque noeud de la liste chaînée lst.

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:
```
-> inverse le signe de tous les éléments du tableau arr.

5. Conventions d'appels dans MIPS

5.1. Comment pouvons-nous transmettre des arguments aux fonctions dans MIPS ?
Utiliser les quatre registres pour arguments `$a0-$a3.`

5.2. Comment les valeurs sont-elles retournées par les fonctions dans MIPS ?
Utiliser les registres `$v0` et `$v1`.

5.3. Qu'est-ce que `$sp` et comment doit-il être utilisé dans le contexte des fonctions MIPS ?
`sp` signifie « stack pointer » ou pointeur de pile en français. `$sp` est le registre MIPS utilisé pour représenter le sommet d'une pile matérielle. Le contenu de `$sp` est décrémenté pour créer plus d'espace (l'adresse mémoire contenue dans `$sp` décroit dans ce cas). Pour libérer de l'espace dans la pile, le registre `$sp` est incrémenté (l'adresse mémoire contenu dans `$sp` augmente). La pile matérielle est principalement utilisée pour sauvegarder (et restaurer ultérieurement) les valeurs des registres qui peuvent être écrasés.

5.4. Si une fonction à 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 ?
Les registres `$v0-$v1`, `$t0-$t9` et `$a0-$a3`.

5.5. Quels registres la fonction appelée doit conserver (ou restaurer après utilisation) avant de retourner à la fonction appelante ?
Les registres `$s0-$s7`, `$gp`, `$sp`, `$fp`, et `$ra`.

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 ?
Les registres `$v0-$v1`, `$t0-$t9` et `$a0-$a3` ne sont pas garantis d'être les mêmes après un appel de fonction (c'est la raison pour laquelle ils doivent être sauvegardés si nécessaire pas la fonction appelante). Les registres `$s0-$s7`, `$gp`, `$sp`, `$fp`, et `$ra` sont garantis d'être les mêmes après un appel de fonction (La fonction appelée **DOIT** conserver les valeurs de ces registres ou les restaurer si nécessaire).

6. Écrire des Fonctions MIPS

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

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

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` ?

```
        add $s0, $a0, $0     # Affecter à $s0 la valeur du paramètre n
        add $s1, $0, $0      # Initialiser l'accumulateur ($s1) à la valeur 0.
loop:   beq $s0, $0, end     # Si $s0 == 0 alors allez à end (nous avons fini).
        add $a0, $s0, $0     # Initialiser $a0 avec la valeur dans $s0, nous
                             # préparons les args pour appeler la fonction
                             # « square » qui calculera le carré du paramètre
                             # passé.
        jal square           # Appel de la fonction « square »
        add $s1, $s1, $v0    # Ajouter la valeur retournée à l'accumulateur
        addi $s0, $s0, -1    # Décrémenter $s0 par la valeur 1
```

```
        j loop                  # Boucler sur loop !
end:    add $v0, $s1, $0        # Affecter $s1 à $v0 (la valeur à retourner)
```

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.

```
prologue:  addi  $sp, $sp, -12  # Allouer de l'espace pour 3 mots dans la pile
           sw $ra, 0($sp)       # Sauvegarder l'adresse de retour
           sw $s0, 4($sp)       # Sauvegarder le contenu du registre $s0
           sw $s1, 8($sp)       # Sauvegarder le contenu du registre $s1

###  mettre ici la réponse de la question précédente ###

epilogue:  lw $ra, 0($sp)       # Restaurer $ra
           lw $s0, 4($sp)       # Restaurer $s0
           lw $s1, 8($sp)       # Restaurer $s1
           addi  $sp, $sp, 12   # Libérer l'espace dans la pile (3 mots)
           jr $ra               # Retour à la fonction appelante
```

On notera ici que le registre `$ra` est sauvegardé/restauré dans le prologue et l'épilogue de la fonction « appelée » `sumSquare` même s'il s'agit d'un registre qui est normalement enregistré par la fonction « appelante ». Cela est dû au fait que la fonction `sumSquare` devient une fonction « appelante » vis-à-vis la fonction `square` appelée dans le corps. Dans ce cas, les règles et conventions d'appels pour la fonction appelante doivent être appliquées (ici, sauvegarde et restauration du registre `$ra`).