Contrôle du flux d'exécution

Instructions MIPS pour le saut et le branchement

Comme pour tout processeur informatique, le MIPS possède des instructions pour la mise en œuvre de sauts et branchements conditionnels. Ces instructions sont affichées dans le tableau 1.

Instruction	Signification	Format			
`j label`	Saut à `label`	$op^6 = 2$	imm^{26}		
`beq rs, rt, label`	Brancher si `(rs == rt)`	$op^6 = 4$	rs^5	rt^5	imm^{16}
`bne rs, rt, label`	Brancher si `(rs != rt)`	$op^6 = 5$	rs^5	rt^5	imm^{16}
`blez rs, label`	Brancher si `(rs <= 0)`	$op^6 = 6$	rs^5	0	imm^{16}
`bgtz rs, label`	Brancher si `(rs > 0)`	$op^6 = 7$	rs^5	0	imm^{16}
`bltz rs, label`	Brancher si `(rs < 0)`	$op^6 = 1$	rs^5	0	imm^{16}
`bgez rs, label`	Brancher si `(rs >= 0)`	$op^6 = 1$	rs^5	1	imm^{16}

Tableau 1 : Instructions MIPS pour les sauts et les branchements

Pour le saut (branchement inconditionnel), l'instruction **`j etiq`** est utilisée. **`etiq`** est l'adresse de l'instruction cible comme indiqué ci-dessous :

```
    j etiq      # saut à etiq
    . . .
etiq:
```

Il y a deux instructions de branchement conditionnel dans MIPS qui permettent d'effectuer un saut selon le cas où les deux registres comparés sont égaux ou non :

```
beq rs, rt, etiq          # saut à etiq si (rs == rt)
bne rs, rt, etiq          # saut à etiq si (rs != rt)
```

Quatre instructions MIPS supplémentaires sont fournies pour la comparaison du contenu d'un registre avec la valeur 0 :

```
bltz rs, etiq            # saut à etiq si (rs < 0)
bgtz rs, etiq            # saut à etiq si (rs > 0)
blez rs, etiq            # saut à etiq si (rs <= 0)
bgez rs, etiq            # saut à etiq si (rs >= 0)
```

Notez que l'ISA du MIPS n'implémente pas les instructions **beqz** et **bnez**. Au fait, ces pseudo-instructions assembleurs peuvent être mises en œuvre en utilisant les instructions de l'ISA **beq** et **bne** avec le registre **$0** utilisé comme deuxième opérande.

Le MIPS fourni également quatre instructions « set on less than » comme suit :

```
slt     rd, rs, rt       # si (rs < rt) rd = 1 sinon rd = 0
sltu    rd, rs, rt       # idem mais pour opérandes non-signés
slti    rt, rs, im16     # si (rs < im16) rt = 1 sinon rt = 0
sltiu   rt, rs, im16     # idem mais pour opérandes non-signés
```

Notez que les instructions **slt** et **slti** sont utilisées pour la comparaison signée tandis que les instructions **sltu** et **sltiu** sont utilisées pour la comparaison non signée.

Par exemple, si nous supposons que **$s0 = 1** et **$s1 = -1 = 0xffffffff**, alors les deux instructions suivantes produisent des résultats différents :

```
slt     $t0, $s0, $s1    # résultat dans $t0 = 0
sltu    $t0, $s0, $s1    # résultat dans $t0 = 1
```

Pseudo-instructions Assembleur

Les pseudo-instructions sont des « macros » (groupe d'instructions de base) introduites par un assembleur et se comporte comme s'il s'agissait d'instructions réelles. Les pseudo-instructions sont utiles car elles facilitent la programmation en langage assembleur.

Nous avons vu plus haut un exemple de pseudo-instructions (*i.e.* **bltz - bgez**). Un autre ensemble d'instructions que l'ISA du MIPS n'implémente pas sont les instructions de branchement conditionnel suivantes :

```
blt, bltu        saut si inférieur                (signé/non-signé)
ble, bleu        saut si inférieur ou égale       (signé/non-signé)
```

```
bgt, bgtu        saut si supérieur              (signé/non-signé)
bge, bgeu        saut si supérieur ou égale     (signé/non-signé)
```

Ces instructions ne sont pas implémentées dans l'ISA du MIPS parce qu'elles peuvent être facilement mises en œuvre en utilisant un ensemble réduit d'instructions réelles. Par exemple, la pseudo-instruction **`blt $s0, $s1, etiq`** peut être mise en œuvre en utilisant la séquence suivante d'instructions réelles :

```
slt $at, $s0, $s1
bne $at, $zero, etiq
```

D'une manière similaire, la pseudo-instruction **`ble $s2, $s3, etiq`** sera convertie par l'assembleur en cette séquence d'instructions réelles :

```
slt $at, $s3, $s2
beq $at, $zero, etiq
```

Le Tableau 2 montre d'autres exemples de pseudo instructions. Notez l'utilisation du registre **`$at`** comme registre temporaire lors de la conversion des pseudo-instructions en instructions réelles. Ce registre est réservé par l'assembleur à cet usage.

Instruction	Equivalent en instructions réelles
`move $s1, $s2`	`addu $s1, $zero, $s2`
`not $s1, $s2`	`nor $s1, $s2, $zero`
`li $s1, 0xabcd`	`ori $s1, $zero, 0xabcd`
`li $s1, 0xabcd1234`	`lui $at, 0xabcd` `ori $s1, $at, 0x1234`
`sgt $s1, $s2, $s3`	`slt $s1, $s3, $s2`
`blt $s1, $s2, label`	`slt $at, $s1, $s2` `bne $at, $zero, label`

Tableau 2 : Exemples de pseudo-instructions MIPS.

Traduction des structures de contrôle de haut-niveau vers l'assembleur MIPS

Nous pouvons traduire n'importe quelle structure de contrôle de haut niveau en langage assembleur à l'aide des instructions de saut, de branchement et des instructions **`set-less-than (slt)`**.

Considérons le test **`if`** suivant dans le langage C :

```
if (a == b) c = d + e; else c = d – e;
```

Si nous supposons que les variables **a**, **b**, **c**, **d**, **e** sont stockées dans les registres **$s0** jusqu'à **$s4** respectivement. Alors le code assembleur suivant implémente ce test :

```
        bne   $s0, $s1, ELSE
        addu  $s2, $s3, $s4
        j     EXIT
ELSE:   subu  $s2, $s3, $s4
EXIT:   . . .
```

Nous pouvons aussi implémenter une condition composée impliquant l'opérateur logique ET :

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

L'instruction IF est implémentée efficacement en utilisant le code assembleur suivant qui utilise le concept de chute (fall through) en algorithmique qui permet de sauter l'exécution de l'instruction si la condition est fausse sinon l'instruction suivante est exécutée :

```
   blez    $s1, next          # skip if s1 <= 0
   bgez    $s2, next          # skip if s2 >= 0
   addiu   $s3, $s3, 1        # $s3++ ; les deux conditions sont vraies
next:
```

De façon similaire, nous pouvons aussi traduire un test IF avec une condition composée impliquant l'opérateur logique OU. Par exemple :

```
if (($sl > $s2) || ($s2 > $s3)) {$s4 = 1;}
```

L'instruction IF est implémentée efficacement en utilisant le code assembleur suivant qui vérifie la première condition si elle est vraie et saute le test de la deuxième condition dans ce cas :

```
   bgt $s1, $s2, L1           # oui : execute le corp du if
   ble $s2, $s3, next         # non : saute le corp du if
L1: li $s4, 1                 # $s4 = 1
next:
```

Nous pouvons également implémenter tout type de boucle sous MIPS. Considérons la mise en œuvre de la boucle **for** suivante :

```
for (i=0; i<n; i++) {
  loop body
}
```

Supposons que la variable **i** est stockée dans le registre **$s0** et que **n** est stockée dans le registre **$s1**. Dans ce cas, la boucle **for** peut être implémentée à l'aide du code assembleur suivant :

```
    li $s0, 0                 # i = 0
ForLoop:
    bge $s0, $s1, EndFor
    loop body
    addi $s0, #s0, 1          # i++
    j ForLoop
EndFor:
```

Considérons l'implémentation de la boucle **while** suivante :

```
  i=0;
  while (i<n) {
    loop body
    i++;
  }
```

Notez qu'une boucle **for** et une boucle **while** sont conceptuellement identiques, et par conséquent leurs codes en assembleur sera identique.

Enfin, considérons l'implémentation de la boucle **do-while** suivante :

```
  i=0;
  do {
    loop body
    i++;
  } while (i<n);
```

La boucle **do-while** peut être traduite à l'aide du code assembleur suivant :

```
    li $s0, 0                 # i = 0
WhileLoop:
    loop body
    addi $s0, $s0, 1          # i++
    blt  $s0, $s1, WhileLoop
```