Généralités MIPS

Syntaxe

- Les commentaires commencent par le symbole # et se terminent à la fin de la ligne.
- Un identificateur est une séquence de caractères alphanumériques, de soulignés (_) et de points (.), qui ne commence pas par un chiffre.
- Les codes opération d'instruction sont des mots réservés qui ne peuvent pas être utilisés comme identificateurs.
- Les étiquettes sont déclarées en les plaçant au début d'une ligne et en les faisant suivre du symbole (:).
- Les nombres sont en base **10** par défaut. S'ils sont précédés de **0x** ils sont interprétés comme hexadécimaux.
- Les chaînes de caractères sont encadrées par des doubles apostrophes (").
- Certains caractères spéciaux dans les chaînes de caractères suivent la convention **C** :
- Retour-chariot : **\n**
- Tabulation : **\t**
- Guillemet : **\"**

Quelques directives

.ascii str	Enregistre en mémoire la chaîne de caractères str, mais ne la termine pas par un caractère nul.
.asciiz str	Enregistre en mémoire la chaîne de caractères **str** et la termine par un caractère nul.
.data<@>	Les éléments qui suivent sont enregistrés dans le segment de données. Si l'argument optionnel **@** est présent, les éléments qui suivent sont enregistrés à partir de l'adresse **@**.
.byte b1; : : : ;bn	Enregistre les **n** valeurs dans des octets consécutifs en mémoire.
.word w1; : : : ;wn	Enregistre les **n** quantités **32 bits** dans des mots consécutifs en mémoire.
.float f1; : : : ;fn	Enregistre les **n** nombres flottants simples précision dans des emplacements mémoire consécutifs.
.text <@>	Les éléments qui suivent sont placés dans le segment de texte de l'utilisateur. Dans **SPIM**, ces éléments ne peuvent être que des instructions ou des mots. Si l'argument optionnel **@** est présent, les éléments qui suivent sont enregistrés à partir de l'adresse **@**.
.globl sym	Déclare que le symbole **sym** est global et que l'on peut y faire référence à partir d'autres fichiers.

Les registres

Il existe **32** registres de **32 bits** numérotés **$0; : : : ; $31**. Les registres peuvent être accédés soit par leur numéro soit par leur nom.

Nom	Numéro du registre	Description
$zero	0	Constante 0
$at	1	Réservé à l'assembleur
$v0,$v1	2-3	Évaluation d'une expression et résultats d'une fonction
$a0,: : :,$a3	4-7	Arguments de sous-programmes
$t0,: : :,$t7	8-15	Valeurs temporaires (non préservées)
$s0,: : :,$s7	16-23	Valeurs temporaires (préservées)
$t8,$t9	24-25	Valeurs temporaires (non préservées)
$k0,$k1	26-27	Réservé pour les interruptions (i.e., système d'exploitation)
$gp	28	Pointeur global
$sp	29	Pointeur de pile
$fp	30	Pointeur de bloc
$ra	31	Adresse de retour

Appel de procédure - Conventions

- Par convention, lors de l'appel de procédure, les registres **$t0,: : :,$t9** sont sauvegardés par l'appelant et peuvent donc être utilisés sans problème par l'appelé. Les registres **$s0,: : :,$s7** doivent quant à eux être sauvegardés et restitués exact par l'appelé.
- La pile croit des adresses hautes vers les adresses basses : on soustrait à **$sp** pour allouer de l'espace dans la pile, on ajoute à **$sp** pour rendre de l'espace dans la pile.
- Les déplacements dans la pile se font sur des mots mémoire entiers (multiples de quatre octets).
- Lors du passage de passage de paramètres : tout paramètre plus petit que **32 bits** est automatiquement promu sur **32 bits**.
- Les quatre premiers paramètres sont passés par les registres **$a0,: : :,$a3**. Les paramètres supplémentaires sont passés dans la pile.
- Toute valeur de format inférieur ou égal à **32 bits** est retournée par le registre **$v0** (sur **64 bits $v1** est utilisé avec **$v0**).

Présentation du Simulateur MIPS (Interface QtSpim)

QtSpim est logiciel de simulateur qui exécute des programmes en assembleur **MIPS**.

L'interface **QtSpim** contient :

1. Une partie pour les registres **«Registers»**, qui affiche le contenu de tous les registres entiers
2. Un segment de texte **«Text Segment»**, qui affiche les instructions MIPS chargées en mémoire pour être exécutées. De gauche à droite, l'adresse mémoire de l'instruction, l'instruction en hexadécimal, les instructions MIPS réelles, l'assembleur MIPS que vous avez écrit ainsi que les commentaires.
3. Un segment de texte **«Data Segment»**, qui affiche les données et leurs valeurs dans les segments de données et la pile en mémoire.
4. La **« Console »** information répertorie les actions effectuées par le simulateur.

Pour une meilleure lisibilité, décochez tout ce qui correspond à *Kernel* dans les menus *Text Segment* et *Data Segment*, et décochez *FP Registers* dans le menu *Window*. Par un clic droit sur un registre ou une adresse mémoire, vous pouvez modifier son contenu dynamiquement.

Comment exécuter un programme MIPS dans QtSpim ?

Pour exécuter le programme dans **QtSpim**, nous devons suivre les étapes suivantes :

1. Utiliser un éditeur de texte pour créer un programme (par exemple Bloc-notes, Sublime, NotePad,…)
2. Le fichier contenant le programme doit avoir l'extension (le suffixe) ''**.s**''.
3. Utiliser le menu ***File--->Reinitialize and Load File*** pour charger le programme en mémoire. Le simulateur inclut un programme d'assemblage, qui effectue l'encodage en binaire et la traduction des pseudo-instructions et des labels. Il vérifie la correction syntaxique du programme. Si le programme est incorrect, une fenêtre signalant la première erreur s'affiche et le programme ne peut pas être chargé.
4. Exécuter le programme, suivant l'une des deux méthodes suivantes :
   a) **Run** (ou touche **F5**) - toutes les instructions seront exécutées
   b) **Step** (ou touche **F10**) - exécution pas à pas.
5. Pour modifier le programme, il faut :
   a) Revenir dans un éditeur de texte,
   b) Effectuer les modifications,
   c) Recharger le programme.

La structure de base d'un programme MIPS

- La section « **.data** », contient les déclarations de données, c.-à-d. les données globales manipulées par le programme utilisateur. Elle est implantée conventionnellement à l'adresse 0x10000000. Sa taille est fixe et calculée lors de l'assemblage. Les valeurs contenues dans cette section peuvent être initialisées grâce à des directives contenues dans le programme source en langage d'assemblage ;
- La section « **.text** », (code du programme) contient le code exécutable en mode utilisateur. Elle est implantée conventionnellement à l'adresse 0x00400000. Sa taille est fixe et calculée lors de

l'assemblage. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé dans cette section ;

- « **main** », début du programme.
- Pour sortir du programme **MIPS**, on fait un appel system « **li $v0,10 syscall** ».
- Les commentaires permettent de donner plus d'explication sur le code. Ils commencent par un « **#** » ou un « **;** ».

***Exemple* d'un simple programme MIPS**

```
1   # Exemple0
2   .data
3   # Il n'y aura pas de données à déclarer, on utilisera le mode immédiat
4   .text
5   main:
6   li $t0,3 # charger la valeur 3 dans le registre t0
7   li $t1,8 # charger la valeur 8 dans le registre t1
8   add $t2,$t1,$t0 # Faire l'addition de t0+t1 et mettre le résultat dans le registre t2
9   #Fin du programme
10  li $v0,10
11  syscall
```

Déclarations des données

Les données (constantes et variables) doivent être déclarées dans la section « **.data** ».

Les données doivent commencer par une lettre et peuvent contenir des lettres, des chiffres et/ou des caractères spéciaux.

Le format général de la déclaration d'une donnée est le suivant :

<Nom de la variable> <type de la donnée> <valeur initiale>

Exemple de déclaration de données

```
1   .data
2   C    .byte   'a'          # Octet
3   N1   .half   26           # 2 octets
4   N2   .word   353          #4 octets
5   Tap      .space 40        # réservation d'un espace de 40 octets
6   Mess .asciiz    "une chaine"    # déclaration d'une chaîne de caractères terminée par 0
```

Les données en Mips sont de différents types :

Déclaration	Description
.byte	Variables à 8 bits
.half	Variables à 16 bits
.word	Variables à 32 bits
.ascii	Chaine ASCII
.asciiz	Chaine ASCII terminée par un zéro
.float	Nombre réel à 32 bits
.double	Nombre réel à 64 bits
.space <n>	Espace mémoire à n bytes non initialisé

Les instructions du MIPS

Le MIPS propose trois types principaux d'instructions :

- Les instructions de **transfert** entre registres et mémoire ;
- Les instructions de **calcul** ;
- Les instructions de **saut**.

Seules les premières permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

1. Instruction de transfert

- **Lecture** (load word):

lw dest, offset(base)

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké à cette adresse est alors transféré vers le registre *dest*.

On a également: **lb** (load byte)**, lbu** (load byte unisgned)**, lh** (load half)**, lhu** (load half unisgned)**.**

- **Ecriture** (store word):

sw source , offset(base)

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké dans le registre *source* est alors transféré vers cette adresse.

On a également: **sb** (store byte)**, sh** (store half)**.**

2. Instruction de calcul

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits **arguments**, effectuent un calcul, puis écrivent le résultat dans un registre dit **destination**.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

- Les instructions de calcul nullaires

- **Ecriture d'une constante** (Load Immediate):

li dest, constant

Produit la constante *constant*.

On a également : **la** (load adrdess)**.**

- Les instructions de calcul unaires

- **Addition d'une constante** (Add Immediate):

addi dest, source, constant

Produit la somme de la constante (de 16 bits) *constant* et du contenu du registre *source*.

- **Déplacement** (Move):

move dest, source

Produit le contenu du registre *source*. Cas particulier de *addi!*

- **Négation** (Negate):

neg dest, source

Produit l'opposé du contenu du registre *source.* Cas particulier de *sub!*

- Les instructions de calcul binaires

- **Addition** (Add):

add dest, source1, source2

Produit la somme des contenus des registres *source1* et *source2*.

On a également : **sub, mul, div.**

- **Comparaison** (Set On Less Than):

slt dest, source1, source2

Produit 1 si le contenu du registre *source1* est inférieur à celui du registre *source2* ; produit 0 sinon.

On a également : **sle, sgt, sge, seq, sne**.

3. Instruction de saut

On distingue les instructions de saut selon que :

- Leurs destinations possibles sont au nombre de 1 (saut **inconditionnel)** ou bien 2 (saut **conditionnel**);
- Leur adresse de destination est **constante** ou bien **lue** dans un registre ;
- Une **adresse de retour** est sauvegarde´ e ou non.

- Saut inconditionnel

- **Saut** (Jump):

j address

Saute à l'adresse constante *address*. Celle-ci est en général donnée sous forme symbolique par une *étiquette* que l'assembleur traduira en une constante numérique.

- Saut conditionnel

- **Saut conditionnel unaire** (Branch on Greater Than Zero):

bgtz source, address

Si le contenu du registre *source* est supérieur à zéro, saute à l'adresse constante *address*.

On a également **bgez, blez, bltz.**

- **Saut conditionnel binaire** (Branch On Equal):

beq source1, source2, address

Si les contenus des registres *source1* et *source2* sont égaux, saute à l'adresse constante *address*.

On a également **bne.**

- Saut avec retour

- **Saut avec retour** (Jump And Link):

jal address

Sauvegarde l'adresse de l'instruction suivante dans le registre *ra*, puis saute à l'adresse constante *address.*

- Saut vers adresse variable

- **Saut vers adresse variable** (Jump Register)

jr target

Saute à l'adresse contenue dans le registre *target*. L'instruction **jr $ra** est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

4. Instruction spéciale

- **Appel système** (System Call):

  syscall

Provoque un appel au noyau. Par convention, la nature du service souhaité est spécifiée par un code entier stocké dans le registre **v0**. Des arguments supplémentaires peuvent être passés dans les registres **a0-a3**. Un résultat peut être renvoyé dans le registre **v0**.

Les appels système

- Un appel système se fait par l'instruction **syscall**.
- Les simulateurs fournissent un ensemble de services par l'intermédiaire de l'instruction d'appel **syscall**, dont le comportement dépend de la valeur du registre **$v0** :
  - si **$v0 = 1**, alors **syscall** affiche l'entier contenu dans le registre **$a0** ;
  - si **$v0 = 4**, alors **syscall** affiche la chaîne de caractère dont l'adresse est contenue dans le registre **$a0** ;
  - si **$v0 = 5**, alors **syscall** lit un entier à l'écran et met sa valeur dans le registre **$v0**.
- Pour demander un service, on charge le code du service (voir tableau ci-dessous) dans le registre **$v0** et ses arguments dans les registres **$a0,...,$a3** (ou **$f12** pour les valeurs flottantes).
- Les appels système qui retournent des valeurs placent leurs résultats dans le registre **$v0** (ou **$f0** pour les résultats flottants).

Le tableau suivant illustre les différents codes de **syscall**.

Service	Code	Arguments	Résultat
print_int	1	$a0 = entier	
print_float	2	$f12 = flottant simple précision	
print_double	3	$f12 = flottant double précision	
print_string	4	$a0 = chaine de caractères	
read_int	5		Un entier dans $v0
read_float	6		Un flottant simple dans $v0
read_double	7		Un flottant double dans $v0
read_string	8	$a0 = tampon, $a1 = longeur	
sbrk	9	$a0 = quantité	Une adresse dans $v0
exit	10		

Exemple 1 : print_int (Appel système code $v0=1)

Le code suivant imprime « **La réponse = 5** » dans la console.

```
1   .data
2   Str : .asciiz    "La reponse = "
3   .text
4   main:
5   la $a0, Str      # Mettre l'adresse de la chaine de caractère Str dans le registre $a0
6   li $v0, 4        # Charger le registre $v0 avec le code 4 pour affichage caractère (print_string)
7   syscall          # appel système pour affichage caractère
8   li $a0, 5        # Mettre la valeur à afficher dans le registre $a0
9   li $v0, 1        # Charger le registre $v0 avec le code 1 pour affichage entier (print_int)
10  syscall          # appel system pour l'opération affichage entier
11  # Fin du Programme
12  li $v0, 10
13  syscall
```

Exemple2 : print_string (Appel système code $v0 =4)

Le code suivant affiche le message « **Hello Word** » dans la console.

```
.data
Hello: .asciiz "Hello Word\n" # mettre la chaine de caractère « Hello Word » en mémoire
.text
main:           # section <code
la $a0, Hello   # charger le registre $a0 avec l'adresse de la chaine de caractère
li $v0, 4       # Charger le registre $v0 avec le code 4 pour affichage caractère
                #(print_string).
syscall         # appel system pour l'opération d'affichage
# Fin du Programme
li $v0,10
syscall
```

Exemple3 : read_int (Appel système code $v0=5)

Le code suivant affiche sur l'écran un entier saisi au le clavier.

```
.data
Str : .asciiz " Saisir un entier "
.text
main:
la $a0,Str  # Mettre l'adresse de la chaine de caractère Str dans le registre $a0
li $v0,4    # Charger le registre $v0 avec le code 4 pour affichage caractère (print_string)
syscall     # Appel système pour affichage caractère
li $v0,5    # Charger le registre $v0 avec le code 5 pour saisir un entier à partir du clavier
syscall     # Appel system pour l'opération lire entier (read_int)
move $a0,$v0 # déplacer la valeur saisie dans le register $a0
li $v0,1    # Charger le registre $v0 avec le code 1 pour affichage entier (print_int)
syscall     # Appel system pour l'opération affichage entier
# Fin du Programme
li $v0,10
syscall
```

Exemple4 : read_string (Appel système code $v0=8)

Le code suivant affiche le message écrit par l'utilisateur.

```
.data
message: .asciiz "Vous avez écrit "
UserInput: .space 20
.text
main:
# Entrer un text de 20 caractères
li $v0, 8
la $a0, UserInput
li $a1, 20
syscall
# Afficher le message « Vous avez écrit »
li $v0,4
la $a0,message
syscall
# Afficher le message
li $v0,4
la $a0,UserInput
syscall
# Fin du programme
li $v0,10
syscall
```

Les sauts et les appels de fonctions ou procédures

1. Sauts inconditionnels

Jump	**j**
Jump **a**nd **l**ink	**jal**
Jump **r**egister	**jr**
Jump **a**nd **l**ink **r**egister	**jalr**

- **j adr** -> va à l'adresse **adr**
- **jal adr** -> sauvegarde en plus **$pc** dans le registre **$ra**
- **jr $t0** -> va à l'adresse contenue dans le registre **$t0**
- **jalr $t0** -> sauvegarde en plus **$pc** dans le registre **$ra**

Saut avec retour (Jump And Link) :

```
jal address
jr $ra
```

- **jal** : Sauvegarde l'adresse de l'instruction suivante dans le registre ***ra***, puis saute à l'adresse constante **address**.
- L'instruction **jr $ra** est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

Exemple

```
1  print_int:
2  li $v0, 1        # # Charger le registre $v0 avec le code 1 pour afficher entier (print_int)
3  syscall          # system call  pour print_int
4  jr $ra           # rendre la main à l'appelant à la fin de la fonction/procédure print_int(return)
5  main:
6  li $a0, 15       # On veut enregistrer la valeur 15 dans le registre$a0
7  jal print_int    # Sauvegarde l'adresse de l'instruction suivante dans le registre ra, puis saute à l'adresse print_int
```

2. Sauts conditionnels :

Branch on **eq**ual	**beq**
Branch on **n**ot **e**qual	**bne**
Branch on **l**ess **t**han	**blt**
Branch on **g**reater **t**han	**bgt**
Branch on **l**ess or **e**qual than	**ble**
Branch on **g**reater or **e**qual than	**bge**

1. **beq r1, r2, adr** **->** si les contenus des registres **r1** et **r2** sont égaux, saute à l'adresse **adr**

Exemple : Instruction conditionnelle if

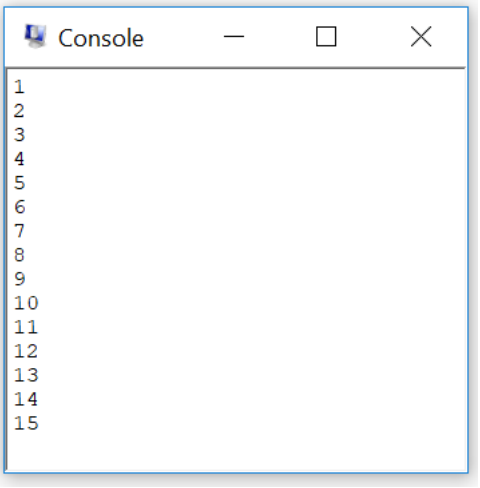
if t1 < t2 then t3 := t1 else t3 := t2

Code MIPS équivalent

```
blt $t1, $t2, Then  # si t1 < t2 saut à Then
move $t3, $t2       # t3 := t2
j End               # saut à End
Then: move $t3, $t1 # t3 := t1
End:                # suite du programme
Li $v0,10
syscall
```

Exercice 1 :

Ecrire le programme MIPS permettant d'afficher les valeurs entières de 0 à 15 comme indiqué ci-dessous sur la console.



 # Solution de l'exercice 1 :

```
1   .data
2   Newln : .asciiz "\n"
3   .text
4   main :
5   li $t0,1
6   li $t1,15
7   loop : move $a0,$t0
8   li $v0,1
9   syscall
10  li $v0,4
11  la $a0,Newln
12  syscall
13  addi $t0,$t0,1
14  ble $t0,$t1,loop #if ($t0<=$t1 aller à loop)
15  li $v0,10
16  syscall
```

Exercice 2 :

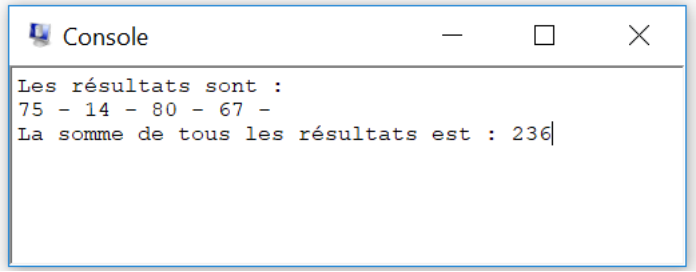
En utilisant l'appel fonction *affiche*, faire un programme MIPS qui calcule et affiche les résultats de : **x+y+z** - **x+4** - **x*8** - **x+y+(z/2)** Tels que x, y, z sont des variables déclarées sur **8 bits** et leurs valeurs sont respectivement : **10, 50, 15.**

Pour l'affichage il faut avoir : (Voir l'image de la console ci-dessous)

Message1- **Les résultats sont :** *retour à la ligne*

Résultat des calculs : **résultat1 - résultat2 - résultat 3 - résultat 4 -**

Message 2 - **La somme de tous les résultats est :** ***Résultat de la somme***



 # Solution de l'exercice 2 :

```
1   .data
2   x: .byte 10
3   y: .byte 50
4   z: .byte 15
5   message1: .asciiz"Les résultats sont : \n"
6   separateur:.asciiz " - "
7   retour:.asciiz "\n"
8   message2 : .asciiz"La somme de tous les résultats est : "
9   .text
10  main:
11  li$v0,4
12  la $a0,message1
13  syscall
14  lb $t1,x
15  lb $t2,y
16  lb $t3,z
17  add $a0,$t1,$t2
18  add $a0,$a0,$t3
19  move $t0,$a0
20  jal affiche
21  addi $a0,$t1,4
22  move $t5,$a0
23  jal affiche
24  li $t4,8
25  mul $a0,$t1,$t4
26  move $t6,$a0
27  jal affiche
28  add $a0,$t1,$t2
29  li $t4,2
30  div $t4,$t3,$t4
31  add $a0,$a0,$t4
32  move $t7,$a0
33  jal affiche
34  li$v0,4
35  la $a0,retour
36  syscall
37  li$v0,4
38  la $a0,message2
39  syscall
40  add $t4,$t0,$t5
41  add $a0,$t4,$t6
42  add $a0,$a0,$t7
43  li $v0,1
44  syscall
45  j exit
46  exit:
47  li $v0,10
48  syscall
49  affiche:
50  li $v0,1
51  syscall
52  li$v0,4
53  la $a0,separateur
54  syscall
55  jr $ra
```

Les registres de MIPS :

La liste des **32** registres programmables de **32 bits** sur **MIPS**, est comme suit :

Nom	Numéro	Description
$zero	$0	constante zéro
$at	$1	réserve pour l'assembleur
$v0	$2	retour de fonction
$v1	$3	retour de fonction
$a0	$4	argument de fonction
$a1	$5	argument de fonction
$a2	$6	argument de fonction
$a3	$7	argument de fonction
$t0	$8	temporaire
$t1	$9	temporaire
$t2	$10	temporaire
$t3	$11	temporaire
$t4	$12	temporaire
$t5	$13	temporaire
$t6	$14	temporaire
$t7	$15	temporaire

Nom	Numéro	Description
$s0	$16	sauvegardé
$s1	$17	sauvegardé
$s2	$18	sauvegardé
$s3	$19	sauvegardé
$s4	$20	sauvegardé
$s5	$21	sauvegardé
$s6	$22	sauvegardé
$s7	$23	sauvegardé
$t8	$24	temporaire
$t9	$25	temporaire
$k0	$26	pour noyau système
$k1	$27	pour noyau système
$gp	$28	pointeur global
$sp	$29	pointeur de pile
$fp	$30	pointeur de frame
$ra	$31	registre d'adresse

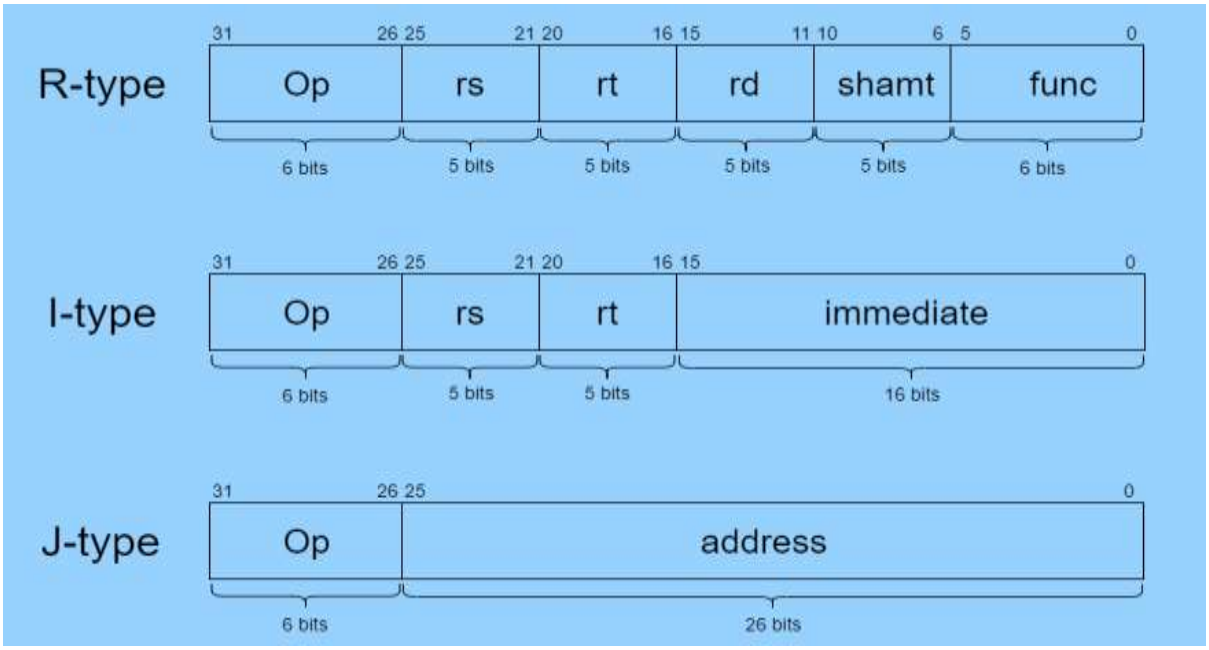
Remarque 1 : Un registre peut être désigné par son nom ou son numéro.

Remarque 2 : Un registre doit toujours être précédé par un **$** lors de son utilisation dans une instruction assembleur.

Remarque 3 : Il existe aussi **2** autres registres qui ne sont pas listés ici, c'est ***lo*** (pour ***low***) et ***hi*** (pour ***high***). Ils sont, exclusivement, utilisés par les instructions de multiplication et division.
Dans le langage assembleur MIPS, les registres généraux utilisés pour contenir les données sont les registres temporaires et les registres sauvegardés, la différence entre les deux est que les registres temporaires ne doivent pas être sauvegardés par le programmeur lors d'un appel de fonction. Les registres ***a***, ***v*** et ***ra*** sont utilisés pour le mécanisme des appels de fonction, les registres ***a*** doivent contenir les arguments, les registres ***v*** la valeur de retour, et ***ra*** l'adresse de retour.

Les formats d'encodage des instructions :

3 formats d'encodage sont possibles dans MIPS :



Op : ***Op****code*, le code de l'instruction.
rs : registre **s**ource.
rt : deuxième **r**egistre **s**ource.
rd : registre destination.
shamt : ***sh****ift* ***am****oun****t***, nombre de bits à décaler dans l'instruction shift.
func : ***func****tion*, le code de la fonction/opération.
address : adresse mémoire sur **26 bits**.

Chaque instruction MIPS est contenue sur **32 bits**, l'encodage des **3** types d'instruction possibles est décrit plus haut.

1. **Les instructions de type R (R** pour **R**egistre**)** utilisent **3** registres pour les opérations arithmétiques et logiques, **2** registres sources comme opérandes et un registre destination pour le résultat, **5 bits** sont utilisés pour coder le numéro du registre utilisé.
2. **Les instructions de type** I (**I** pour **i**mmédiate) utilisent aussi **2** opérandes et un registre pour le résultat, la différence avec le type **R** est que l'un de ces opérandes est une immédiate sur **16 bits**, ils sont utilisés pour différents types d'instructions incluant aussi des opérations arithmétiques et logiques.
3. **Les instructions de type J (J** pour **J**ump, ou saut**)** sont réservées pour le saut.

Remarque : On peut observer sur les **3** formats, que seulement **3** manières d'utiliser l'information dans une instruction MIPS sont possibles, soit l'information est dans un registre de **32 bits**, soit une valeur immédiate de **16** bits, soit une adresse de **26 bits** (Les **5 bits** de ***shampt*** sont particuliers aux instructions de **shift**).

Les instructions arithmétiques et logiques :

Instruction	Type	Syntaxe	Description
Addition (Addition)	R	**add** rd,rs,rt	rd ← rs + rt
	I	addi rt,rs,imm	rt ← rs + imm
Substraction (Soustraction)	R	**sub** rd,rs,rt	rd ← rs - rs
	I	subi rt,rs,imm	rt ← rs - imm
And (ET)	R	**and** rd,rs,rt	rd ← rs & rs
	I	andi rt,rs,imm	rt ← rs & imm
Or (OU)	R	**or** rd,rs,rt	rd ← rs | rs
	I	ori rt,rs,imm	rt ← rs | imm

Exemples :

```
add   $s0, $s2, $s3
subi $t0, $t5, 3
and   $s3, $t5, $t1
```

Remarque 1 : Les opérations logiques sont des opérations binaires (**bitwise** : bit-par-bit).

Remarque 2 : Les autres opérations logiques comme le ***not***, ***nand*** ou ***xnor*** sont réalisées en composition à partir des instructions présentes dans le jeu d'instruction.

Les instructions de multiplication et division

Les instructions concernant la division et la soustraction sont des instructions particulières dans le sens où leur résultat nécessite **64 bits** de mémoire, la **multiplication** de **2** nombres de **32 bits** exige **64 bits** d'espace pour le résultat, et **la division entière** exige aussi **32 bits** pour le **quotient** (résultat de division entière), et **32 bits** pour le **reste de division**. Pour cela ces **2** opérations utilisent les registres **32 bits lo** (pour **lo**w) et **hi** (pour **hi**gh) qui sont exclusivement utilisés par les instructions de **multiplication** et de **division**.

Instruction	Type	Syntaxe	Description
Divide (Division)	R	**DIV** rs,rt	HI=rs%rt (Modulo); LO=rs/rt (Quotient)
Divide **U**nsigned	R	**DIVU** rs,rt	HI=rs%rt (Modulo); LO=rs/rt (Quotient)
Move **F**rom **HI**	R	**MFHI** rd	rd=HI
Move **F**rom **LO**	R	**MFLO** rd	rd=LO
Move **T**o **HI**	R	**MTHI** rs	HI=rs
Move **T**o **LO**	R	**MTLO** rs	LO=rs
Multiply (Multiplication)	R	**MUL** rd,rs,rt	HI,LO=rs*rt ; rd=LO
Multiply	R	**MULT** rs,rt	HI,LO=rs*rt
Multiply **U**nsigned	R	**MULTU** rs,rt	HI,LO=rs*rt

Remarque 1 : Les **2 points** dans la description (**hi:lo**) indiquent une concaténation, ça veut dire que les **2** registres de **32 bits hi** et **lo** sont mis côte à côte pour construire un registre de **64 bits**.

Remarque 2 : Les **2** instructions **mfhi** (**M**ove **f**rom **hi**) et **mflo** (**M**ove **f**rom **lo**) sont des instructions qui permettent de copier les valeurs de **hi** et **lo** dans l'un des **32** registres programmables. Il faut rappeler que **hi** et **lo** ne sont pas des registres programmables donc ils ne peuvent pas être utilisés directement dans les instructions MIPS.

Remarque 3 : L'instruction **mul** est identique à l'instruction **mult** mais elle fait en plus copier la valeur de **lo** dans **rd**, ça favorise son utilisation lorsque la multiplication est sur des petites grandeurs.

Les instructions de transfert de données :

Les instructions de transfert de données sont très importantes. Elles permettent de faire le transfert de l'information d'un emplacement vers un autre. Par exemple :

- le transfert de la mémoire vers un registre est fait par l'instruction **lw** (**l**oad **w**ord)
- et l'inverse, d'un registre vers la mémoire est **sw** (**s**tore **w**ord),
- **move** est pour un transfert d'un registre vers un autre,
- **li** le transfert d'une immédiate vers un registre,
- et **la** d'une adresse vers un registre.

Instruction	Type	Syntaxe	Description
move	P	move rd,rs	rd ← rs
li	P	li rd,imm	rd ← imm
la	P	la rd,label	rd ← label
lw	I	lw rt,imm(rs)	rt ← MEM[rs+imm]
sw	I	sw rt,imm(rs)	MEM[rs+imm] ← rt

Remarque 1 : Le type **P** (pour **P**seudo-instruction) n'est pas un type proprement dit. Réellement, les instructions de ce type sont créées par l'assembleur à partir d'autres instructions équivalentes.
Exemple : move rd,rs est réellement add rd,rs,$zero.

Remarque 2 : Les **2** instructions **lw** et **sw** forment leurs adresses mémoires d'une manière analogue à la notion **segment/offset**, dans le sens où l'adresse est formée par l'ajout d'un décalage **imm** à une adresse de départ **rs**.

Remarque 3 : L'instruction ***load*** possède d'autres variantes en plus du **lw**, il y'a **lb** pour *load **byte*** qui fait charger une valeur de la mémoire sur **1 octet** (**8bits**), et **lh** pour *load **half*** qui fait charger une valeur sur **16 bits** (**demi-mot**).

Les instructions de comparaison :

Instruction	Type	Syntaxe	Description
slt	R	slt rd,rs,rt	**If**(rs < rt) rd ← 1 **else** rd ← 0
	I	slti rt,rs,imm	**If**(rs < imm) rt ← 1 **else** rt ← 0
sle	P	sle rd,rs,rt	**If**(rs <= rt) rd ← 1 **else** rd ← 0
	P	slei rt,rs,imm	**If**(rs <= imm) rt ← 1 **else** rt ← 0
seq	P	seq rd,rs,rt	**If**(rs == rt) rd ← 1 **else** rd ← 0
	P	seqi rt,rs,imm	**If**(rs == imm) rt ← 1 **else** rt ← 0
sne	P	sne rd,rs,rt	**If**(rs != rt) rd ← 1 **else** rd ← 0
	P	snei rt,rs,imm	**If**(rs != imm) rt ← 1 **else** rt ← 0
sgt	P	sgt rd,rs,rt	**If**(rs > rt) rd ← 1 **else** rd ← 0
	P	sgti rt,rs,imm	**If**(rs > imm) rt ← 1 **else** rt ← 0
sge	P	sge rd,rs,rt	**If**(rs >= rt) rd ← 1 **else** rd ← 0
	P	sgei rt,rs,imm	**If**(rs >= imm) rt ← 1 **else** rt ← 0

Remarque : À l'exception de **slt** et **slti**, les autres instructions de comparaison ne sont pas réelles. Ce sont des pseudo-instructions fournies par l'assembleur pour faciliter la programmation.

Les instructions de saut :

Les instructions de saut (**jump**) et de branchement (**branch**) permettent de faire des sauts dans l'exécution d'un programme. Ça permet réellement d'implémenter au niveau assembleur les instructions de tests comme le **if**, **if-else**, **switch** et les instructions de boucle comme **for**, **while**, **do-while**.

Exemple : Le code MIPS correspondant au fragment du programme suivant ;

```
t2 := 0;
while t1 > 0 do begin t2 := t2 + t1; t1 := t1 - 1 end
```

est le suivant :

```
  li $t2, 0            # t2 := 0
  while :              # début de la boucle
  blez $t1, done
  add $t2, $t2, $t1    # t2 := t2 + t1
  sub $t1, $t1, 1      # t1 := t1 - 1
  j while              # retour vers l 'étiquette while
  done:                # suite du programme
```

Instruction	Type	Syntaxe	Description
Jump	J	**j** address	PC ← address
	R	jr rs	PC ← rs
Jump **a**nd **l**ink	J	**jal** address	$ra ← PC+4 ; PC ← address
	R	**jal**r rs	$ra ← PC+4 ; PC ← rs

Remarque 1 : Les adresses dans l'assembleur sont implémentées par le mécanisme des labels.

Remarque 2 : Les instructions ***jump** and **link*** sont utilisées lors d'appel de fonction. En plus d'un saut, ces instructions sauvegardent aussi le **PC+4** dans le registre **$ra**, faisant office d'adresse de retour lors de la fin de la fonction. **PC+4** est l'adresse suivante de l'adresse actuelle sachant qu'une adresse est représentée sur **32 bits** (**4 octet**).

Les instructions de branchement :

Ce sont des instructions semblables aux instructions de saut, la principale différence est qu'ils doivent satisfaire une condition pour que le saut soit effectué.

Instruction	Type	Syntaxe	Description
beq (**b**ranch on **eq**ual)	I	beq rs,rt,imm	**If**(rs == rt) PC ← imm **else** PC ← PC+4
bne (**b**ranch on **n**ot **e**qual)	I	bne rs,rt,imm	**If**(rs != rt) PC ← imm **else** PC ← PC+4
blt (**b**ranch on **l**ess **t**han)	P	blt rs,rt,imm	**If**(rs < rt) PC ← imm **else** PC ← PC+4
ble (**b**ranch on **l**ess or **eq**ual)	P	ble rs,rt,imm	**If**(rs <= rt) PC ← imm **else** PC ← PC+4
bgt (**b**ranch on **g**reat **t**han)	P	bgt rs,rt,imm	**If**(rs > rt) PC ← imm **else** PC ← PC+4
bge (**b**ranch on **g**reat or **eq**ual)	P	bge rs,rt,imm	**If**(rs >= rt) PC ← imm **else** PC ← PC+4

Remarque 1 : L'adresse de branchement ici est implémentée dans l'immédiate, c'est le même mécanisme de labels que va utiliser l'assembleur pour réaliser cette implémentation.

Remarque 2 : Il existe des variantes pour ces instructions avec un **Z** comme suffixe indiquant que la comparaison se fait par rapport à **z**éro.
Exemple : beq**z** rs,imm est une instruction qui applique la comparaison entre ***rs*** et **zéro** comme condition de saut.

Remarque 3 : Sur toutes les instructions déjà faites, il existe des instructions avec un suffixe **U** (pour **u**nsigned). C'est pour dire que l'instruction fait l'opération sur des valeurs avec la représentation à valeur **absolue** (non signé).
Exemple : add**u** rd,rs,rt.

Remarque 4 : Il faut savoir que dans MIPS on a globalement **3** types de données à manipuler, les **immédiates**, les **registres** et les **adresses** (les labels).

Les Fonction dans MIPS :

L'implémentation des fonctions dans MIPS implique l'utilisation de **7 registres spécifiques**, **$a0**, **$a1**, **$a2**, **$a3** pour le passage d'arguments, **$v0**, **$v1** pour le retour de fonction, et **$ra** pour l'adresse de retour de la fonction. Les étapes d'exécution de l'appel d'une fonction sont :

- remplir les registres d'argument **$a0**, **$a1**, **$a2**, **$a3** avec les paramètres à faire passer à la fonction.
- appel de la fonction avec l'instruction **jal**. Ça permet de sauter au début de la fonction, et de sauvegarder l'adresse de retour.
- la fonction va toujours commencer par récupérer les paramètres des registres **$a0**, **$a1**, **$a2**, **$a3**.
- à la fin d'exécution de la fonction, elle doit remplir la valeur de retour dans **$v0**, **$v1**.
- L'instruction de retour est **jr $ra**, (**$ra** contient l'adresse l'instruction juste après **jal**).
- la valeur retournée par la fonction est dans **$v0**, **$v1**.

Remarque : L'instruction **jal** (**jump and link**) est l'instruction d'appel d'une fonction. Elle fait un saut vers le label de l'adresse de la première instruction de la fonction, et le **link** est l'opération de sauvegarder l'adresse de l'instruction juste après l'instruction **jal (PC+4)** dans **$ra**.