

## Solution Série 3 (Processeur MIPS)

### Rappel :

**Les registres de MIPS** : la liste des 32 registres programmables de 32 bits sur MIPS, est comme suite :

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

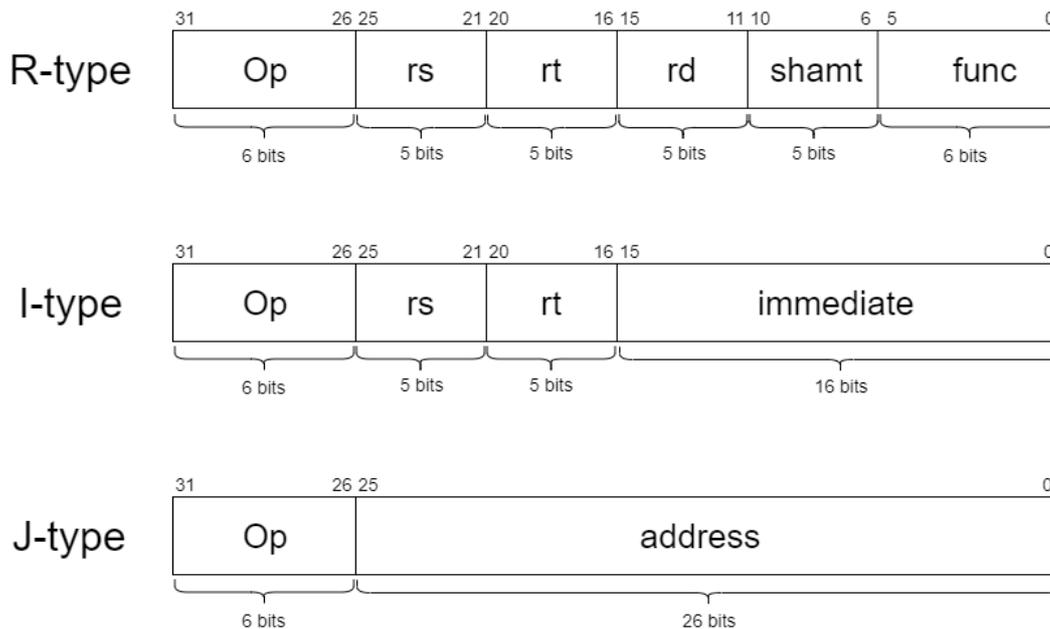
**Remarque 1:** Un registre peut être désigné par son nom ou son numéro.

**Remarque 2:** Un registre doit toujours être précédé par un \$ lors de sont 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*), il 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 possible dans MIPS :



**Op :** *Opcode*, le code de l'instruction.

**rs :** registre source.

**rt :** deuxième registre source.

**rd :** registre destination.

**shamt :** *shift amount*, nombre de bits à décaler dans l'instruction shift.

**func :** *function*, 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'instructions possibles est décrit plus haut. Les instructions de type R (R pour Registre) 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. Le type I (I pour immédiate) utilise 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. Le type J (J pour Jump, ou saut) est réservé pour quelques instructions de 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 immédiate de 16 bits, soit une adresse de 26 bits (Les 5 bits de *shamt* sont particuliers aux instructions de shift).

**Les principales instructions de MIPS :**

**Les instructions arithmétiques et logiques :**

Instruction	Type	Syntaxe	Description
addition	R	add rd,rs,rt	$rd \leftarrow rs + rt$
	I	addi rt,rs,imm	$rt \leftarrow rs + imm$
soustraction	R	sub rd,rs,rt	$rd \leftarrow rs - rs$
	I	subi rt,rs,imm	$rt \leftarrow rs - imm$
and	R	and rd,rs,rt	$rd \leftarrow rs \& rs$
	I	andi rt,rs,imm	$rt \leftarrow rs \& imm$
or	R	or rd,rs,rt	$rd \leftarrow rs   rs$
	I	ori rt,rs,imm	$rt \leftarrow rs   imm$
nor	R	nor rd,rs,rt	$rd \leftarrow \sim(rs   rs)$
	I	nori rt,rs,imm	$rt \leftarrow \sim(rs   imm)$
xor	R	xor rd,rs,rt	$rd \leftarrow rs \wedge rs$
	I	xori rt,rs,imm	$rt \leftarrow rs \wedge imm$

**Exemples :**

```
add    $s0 , $s2 , $s3
subi   $t0 , $t5 , 10
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'instructions.

**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 est 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* et *hi*.

Instruction	Type	Syntaxe	Description
mul	R	mul rd,rs,rt	(hi:lo) ← rs * rt ; rd ← lo
mult	R	mult rs,rt	(hi:lo) ← rs * rt
div	R	div rs,rt	rd ← rs & rs
mfhi	R	mfhi rd	rd ← lo
mflo	R	mflo rd	rd ← hi

**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` et `mflo` 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 il 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 instruction de transfert de données sont très importantes, il 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` (load word) et l'inverse, d'un registre vers la mémoire est `sw` (store word), `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 Pseudo-instruction) n'est pas un type à proprement dit, réellement les instructions de ce type sont créés par l'assembleur à partir d'autres instructions équivalente **ex** : `move rd,rs` est réellement `add rd,rs,$zero`.

**Remarque 2:** Les 2 instructions `lw` et `sw` formes leurs l'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 réel, 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 saut 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*.

Instruction	Type	Syntaxe	Description
Jump	J	j address	PC ← address
	R	jr rs	PC ← rs
Jump and link	J	jal address	\$ra ← PC+4 ; PC ← address
	R	jalr rs	\$ra ← PC+4 ; PC ← rs

**Remarque 1:** Les adresses dans l'assembleur sont implémentés 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).

**Remarque 3:** L'opération pour charger le PC avec les 26 bits du champs *address* est relativement complexe, l'explication ne sera pas abordée ici, vous pouvez consulter le livre *Digital Design and Computer Architecture* au chapitre 6 pour en savoir plus.

### 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	I	beq <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> == <i>rt</i> ) PC ← <i>imm</i> <b>else</b> PC ← PC+4
bne	I	bne <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> != <i>rt</i> ) PC ← <i>imm</i> <b>else</b> PC ← PC+4
blt	P	blt <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> < <i>rt</i> ) PC ← <i>imm</i> <b>else</b> PC ← PC+4
ble	P	ble <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> <= <i>rt</i> ) PC ← <i>imm</i> <b>else</b> PC ← PC+4
bgt	P	bgt <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> > <i>rt</i> ) PC ← <i>imm</i> <b>else</b> PC ← PC+4
bge	P	bge <i>rs, rt, imm</i>	<b>If</b> ( <i>rs</i> >= <i>rt</i> ) PC ← <i>imm</i> <b>else</b> 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. La manière détaillée pour le faire est aussi disponible dans le livre.

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

**Remarque 3:** Sur toutes les instructions déjà faite il existe des instructions avec un suffixe U (pour unsigned), c'est pour dire que l'instruction fait l'opération sur des valeur avec la représentation à valeur absolue et non complément-à-2. **ex** : `addu 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 labels (les adresses).

## Exercice 01 :

a.

```
.data          # déclaration de données
entree1: .asciiz "Entrez un entier X: " ;
entree2: .asciiz "Entrez un entier positif N: " # déclaration de 3
sortie: .asciiz "Résultat = "                # chaînes de caractères

.text          # début du programme
li $v0, 4      # v0 ← 4 , choisir le service
la $a0, entree1 # a0 ← @entree1 , le paramètre du service
syscall       # appel de service d'affichage d'une chaînes de caractères

li $v0, 5      # v0 ← 5 , choisir le service
syscall       # appel de service de lecture d'un nombre entier

addu $t0, $0, $v0 # $t0 ← $v0 équivalent de move $t0, $v0

li $v0, 4      # v0 ← 4 , choisir le service
la $a0, entree2 # a0 ← @entree2 , le paramètre du service
syscall       # appel de service d'affichage d'une chaînes de caractères

li $v0, 5      # v0 ← 5 , choisir le service
syscall       # appel de service de lecture d'un nombre entier

addu $t1, $0, $v0 # $t1 ← $v0 équivalent de move $t1, $v0

li $t2, 1      # $t2 ← 1

boucle:
    beq $t1, $0, fin # si ($t1 == 0) on sort de la boucle, c'est la condition d'une boucle tanque
    multu $t0, $t2 # $lo ← $t2 x $t0
    mflo $t2 # $t2 ← $lo
    subi $t1, $t1, 1 # $t1 ← $t1 - 1
    j boucle # sauter au début de la boucle

fin:
li $v0, 4      # v0 ← 4 , choisir le service
la $a0, sortie # a0 ← @sortie , le paramètre du service
syscall       # appel de service d'affichage d'une chaînes de caractères

addu $a0, $0, $t2 # $t1 ← $v0 équivalent de move $a0, $t2

li $v0, 1      # v0 ← 1 , choisir le service
syscall       # appel du service d'affichage d'un nombre entier

li $v0, 10     # v0 ← 10 , choisir le service
syscall       # appel de service d'arrêt du programme
```

Le programme en MIPS fait le calcul de la puissance d'un nombre entier, son équivalent en C++ est comme suite :

```
string  entree1 = "Entrez un entier X: "      ;
string  entree2 = "Entrez un entier positif N: " ;
string  sortie  = "Résultat = "              ;

cout << entree1 ;
cin  >> t0      ;

cout << entree2 ;
cin  >> t1      ;

t2 = 1 ;
while(t1 != 0)
{
    t2 = t2 * t0 ;
    t1 = t1 - 1 ;
}

cout << sortie ;
cout << t2      ;

return 0 ;
```

Les services dans MIPS permettent d'effectuer différentes opérations généralement propre au système d'exploitation, comme la lecture d'un clavier, l'affichage sur écran, l'arrêt du programme...etc. L'utilisation d'un service suit toujours les 3 étapes suivantes :

1. le choix du service dans le registre `$v0` (la liste de quelque service en bas).
2. le passage de paramètres sur par les registres : `$a0..$a4`.
3. l'appel du service par l'instruction `syscall`.
4. Potentiellement un service peut retourner un résultat, dans `$v0`.

**exemple 1:** affichage de l'entier 3.

```
li $v0, 1      # étape 1 , choisir le service
li $a0, 3      # étape 2 , le passage du paramètre
syscall        # étape 3 , appel du service
               # étape 4 , il n'y a pas de valeur de retour
```

**exemple 2:** lire un entier.

```
li $v0, 5      # étape 1 , choisir le service
               # étape 2 , il n'y a aucun paramètre à faire passer
syscall        # étape 3 , appel du service
move $t0, $v0  # étape 4 , récupérer la valeur de retour dans $v0
```

La table suivante recense quelques service communément utilisés :

Services	Code du service (dans $\$v0$ )	Paramètres ( $\$a0..\$a4$ )	Valeur de retour ( $\$v0$ )
affichage d'entier	1	entier à afficher dans $\$a0$	par de retour
lecture d'entier	5	pas de paramètres	retour dans $\$v0$
affichage de chaîne de caractères	4	adresse (label) de la chaîne dans $\$a0$	par de retour
arrêt du programme	10	pas de paramètres	par de retour

**Remarque :** la liste de tous les services de MIPS peut être consultée dans le menu `help` de l'assembleur MARS.

b.

Registres	Contenus
PC	0x0040 0074 (116)
v0	10
a0	9
t0	3
t1	0
t2	9
lo	9
hi	0

**Remarque 1:** PC n'est pas facile à calculer sans assembleur. le PC doit effectivement contenir l'adresse de la dernière instruction du programme, dans notre cas (`syscall`), en sachant que chaque instruction occupe 4 octets dans le segment des programmes (0x0040), en plus que les pseudo-instruction vont être changées dans leurs forme originale par l'assembleur, il peuvent être changées en plusieurs instructions.

**Remarque 2:** Le panneau `Execute` dans l'assembleur MARS après exécution, permet d'afficher les adresses de chaque instruction dans sa forme originale, et ainsi de déduire l'adresse de PC de notre programme.

c. la console produit l'affichage suivant :

```
Entrez un entier X: 3
Entrez un entier positif N: 2
Résultat = 9
-- program is finished running --
```

d. Ce programme calcule la fonction X à la puissance de N ( $X^N$ ).

**Directives de l'assembleur :** les directives sont des commandes données à l'assembleur pour organiser les programmes, tous les directives commencent par un points (.). La liste des différentes directives sont comme suite :

1. `.data` : cette directive indique à l'assembleur que la partie qui suit est celle de la déclaration des données, Réellement c'est l'indique de l'adresse du segment de données (0x1001 0000).
2. `.text` : cette directive indique que la partie qui suit est celle réservée aux instructions du programme. Réellement c'est l'indique de l'adresse du segment du programme (0x0040 0000).
3. `.byte`, `.half`, `.word` : les 3 directives permettent de déclarer dans le segment de données les entiers signés respectivement de taille 8 bits, 16 bits, 32 bits.
4. `.float` et `.double` : c'est 2 directives sont équivalentes au types réels de même nom dans le C++ float et double.
5. `.ascii` et `.asciiz` : sont aussi des directives de déclaration de données, il permettent de déclarer des chaînes de caractères avec encodage en ASCII, les chaînes de caractères sont déclarées comme tableaux de caractères. Le type `.asciiz` est différent de `.ascii` dans le fait qu'il doit avoir un caractère null (0x00 ou \0 en C++) à la fin de la chaîne de caractères (identique au langage C) pour reconnaître la fin de la chaîne.

**Remarque :** La liste de toutes les directives est présente à travers le menu `help` de MARS.

**Les labels :** Les labels dans MIPS permettent de manipuler les adresses d'une façon facile et agréable au programmeur, ainsi le programmeur peut sauvegarder l'adresse d'une données ou-bien d'une instruction facilement en écrivant un identifiant (nouveau nom dans un programme contrairement aux mots clés) suivi de 2 points (:) avant la donnée ou l'instruction qu'il veut avoir son adresse.

**exemple :** Les 2 instructions suivantes sont prises du code de l'exercice 01, `entree1:` et `boucle:` ce sont 2 labels, la première contient d'adresse de la chaîne de caractères et la deuxième contient l'adresse de l'instruction, ces 2 adresses seront utilisées après dans le code du programme.

```
entree1: .asciiz "Entrez un entier X: " ;
```

**et**

```
boucle:
```

```
    beq $t1, $0, fin
```

**Les sauts (Jump) et les branchements** : ces 2 mécanismes permettent dans le code source (`.text`) de ne pas faire une exécution successive de l'instruction suivante et de sauter vers une instruction distante. Ils permettent généralement d'implémenter des instructions équivalentes dans les langages évolués comme les tests (`if`, `if-else`, `switch`), les boucles (`for`, `while`, `do-while`), ou l'implémentation des fonctions.

**exemple 1:** instruction `if` :

```
.text
    li    $s0, 4          # s0 ← 4
    blt   $s0, $0, endif # if(s0 <= 0) sauter à l'adresse endif
    addi  $s0, $s0, 1     # s0 ++
endif:  # le label qui contient l'adresse de l'instruction li $s1, 0
    li    $s1, 0          # autre instruction après le bloque du if
```

Son équivalent en C++ est comme suite :

```
s0 = 4 ;
if(s0 > 0)
{ s0++ ;
}
s1 = 0 ;
```

**Remarque** : Une remarque très importante dans le code précédent, est que la condition dans l'instruction `blt` est l'inverse de la condition du `if` du code C++, en raison que la première fait sortir l'exécution après le bloque de `if`, alors que dans le code C++ elle fait l'inverse, autrement faire entrer l'exécution dans le bloque `if`.

**exemple 2:** instruction `if-else` :

```
.text
    li    $s0, 4          # s0 ← 4
    blt   $s0, $0, else  # if(s0 <= 0) sauter à l'adresse else
    addi  $s0, $s0, 1     # s0 ++
    j     endif          # sauter à l'adresse endif et sortir du bloque if-else
else:   # le label qui contient l'adresse du bloque else
    subi  $s0, $s0, 1     # s0 --
endif:  # le label qui contient l'adresse de l'instruction li $s1, 0 suivante
    li    $s1, 0          # autre instruction après le bloque du if-else
```

Son équivalent en C++ est comme suite :

```
s0 = 4 ;
if(s0 > 0)
{ s0++ ;
}
else
{ s0-- ;
}
s1 = 0 ;
```

## Exercice 02 :

A. Avant de faire le programme de la factorielle en MIPS on va tout d'abord revoir l'algorithme en C++ :

```
int n , F ;
cout << "Donnez S.V.P. un nombre positive: " ;
cin >> n ;
F = 1 ;
while(n > 0)
{
    F = F * n ;
    n-- ;
}
cout << F << "La factorielle est : " << endl ;
```

en MIPS ça donnerait :

```
.data
msg1: .asciiz "Donnez S.V.P. un nombre positive: "
msg2: .asciiz "La factorielle est : "

.text
    li $v0 , 4
    la $a0 , msg1
    syscall                # affichage du msg1

    li $v0 , 5
    syscall
    move $s0 , $v0        # lecture de n dans s0, donc n est s0

    li $s1 , 1            # s1 est F, il est initialisé à 1

loop: ble $s0 , $0 , out  # la condition de while, ça permet de sortir de la boucle

    mul $s1 , $s1 , $s0   # la multiplication  $F \leftarrow F * n$  ;
    subi $s0 , $s0 , 1    # n--

    j loop                # revenir au début de la boucle
out:
    li $v0 , 4
    la $a0 , msg2
    syscall                # affichage du msg2

    li $v0 , 1
    move $a0 , $s1
    syscall                # affichage de F

    li $v0 , 10
    syscall                # arrêter le programme
```

## B. Le programme de Fibonacci en C++ est comme suite :

```
int  n , T0 , T1 , T2  ;
cout <<  "Donnez S.V.P. un nombre positive: "  ;
cin  >>  n  ;
cout <<  "La suite de Fibonacci est comme suite : "  <<  endl  ;

T0 = 1 ;  T1 = 1 ;
cout <<  T0  <<  endl  <<  T1  <<  endl  ;

do
{
    T2 = T1 + T0  ;
    cout <<  T2  <<  endl  ;
    T0 = T1      ;
    T1 = T2      ;
    n--          ;
}
while(n > 0)  ;
```

## L'équivalent en MIPS du programme de Fibonacci est comme suite :

```
.data
msg1: .asciiz  "Donnez S.V.P. un nombre positive: : "
msg2: .asciiz  "La suite de Fibonacci est comme suite : "

.text
li  $v0 , 4
la  $a0 , msg1
syscall          # affichage du msg1

li  $v0 , 5
syscall
move  $s0 , $v0  # lecture de n dans s0, donc n est s0

li  $v0 , 4
la  $a0 , msg2
syscall          # affichage du msg2

li  $t0 , 1      # t0 est T0, il est initialisé à 1
li  $t1 , 1      # t1 est T1, il est initialisé à 1

li  $v0 , 1
move  $a0 , $t0
syscall          # affichage de T0

li  $v0 , 1
move  $a0 , $t1
syscall          # affichage de T1
```

```

loop: add $t2 , $t1 , $t0      # T2 ← T1 + T0 ;

    li    $v0 , 1
    move  $a0 , $t2
    syscall                               # affichage de T2
    move  $t0 , $t1                  # T0 ← T1 ;
    move  $t1 , $t2                  # T1 ← T2 ;
    subi  $s0 , $s0 , 1              # n-- ;

    bgt  $s0 , $0 , loop            # condition do-while, si c'est true ça boucle au début du bloque

    li  $v0 , 10
    syscall                               # arrêter le programme

```

## OpCodes et encodage des 57 instructions de MIPS (MIPS I):

Opcode	Name	Action	Opcode bitfields						
<b>Arithmetic Logic Unit</b>									
ADD rd,rs,rt	Add	rd=rs+rt	000000	rs	rt	rd	00000	100000	
ADDI rt,rs,imm	Add Immediate	rt=rs+imm	001000	rs	rt	imm			
ADDIU rt,rs,imm	Add Immediate Unsigned	rt=rs+imm	001001	rs	rt	imm			
ADDU rd,rs,rt	Add Unsigned	rd=rs+rt	000000	rs	rt	rd	00000	100001	
AND rd,rs,rt	And	rd=rs&rt	000000	rs	rt	rd	00000	100100	
ANDI rt,rs,imm	And Immediate	rt=rs&imm	001100	rs	rt	imm			
LUI rt,imm	Load Upper Immediate	rt=imm<<16	001111	rs	rt	imm			
NOR rd,rs,rt	Nor	rd=~(rs rt)	000000	rs	rt	rd	00000	100111	
OR rd,rs,rt	Or	rd=rs rt	000000	rs	rt	rd	00000	100101	
ORI rt,rs,imm	Or Immediate	rt=rs imm	001101	rs	rt	imm			
SLT rd,rs,rt	Set On Less Than	rd=rs<rt	000000	rs	rt	rd	00000	101010	
SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	001010	rs	rt	imm			
SLTIU rt,rs,imm	Set On < Immediate Unsigned	rt=rs<imm	001011	rs	rt	imm			
SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt	000000	rs	rt	rd	00000	101011	
SUB rd,rs,rt	Subtract	rd=rs-rt	000000	rs	rt	rd	00000	100010	
SUBU rd,rs,rt	Subtract Unsigned	rd=rs-rt	000000	rs	rt	rd	00000	100011	
XOR rd,rs,rt	Exclusive Or	rd=rs^rt	000000	rs	rt	rd	00000	100110	
XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm	001110	rs	rt	imm			
<b>Shifter</b>									
SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	000000	rs	rt	rd	sa	000000	
SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs	000000	rs	rt	rd	00000	000100	
SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	000000	000000	rt	rd	sa	000011	
SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000111	
SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa	000000	rs	rt	rd	sa	000010	
SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000110	
<b>Multiply</b>									
DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011010	
DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011011	
MFHI rd	Move From HI	rd=HI	000000	0000000000		rd	00000	010000	
MFLO rd	Move From LO	rd=LO	000000	0000000000		rd	00000	010010	
MTHI rs	Move To HI	HI=rs	000000	rs	00000000000000			010001	
MTLO rs	Move To LO	LO=rs	000000	rs	00000000000000			010011	
MULT rs,rt	Multiply	HI,LO=rs*rt	000000	rs	rt	0000000000		011000	
MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt	000000	rs	rt	0000000000		011001	

Branch							
BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	000100	rs	rt	offset	
BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4	000001	rs	00001	offset	
BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4	000001	rs	10001	offset	
BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4	000111	rs	00000	offset	
BLEZ rs,offset	Branch On	if(rs<=0) pc+=offset*4	000110	rs	00000	offset	
BLTZ rs,offset	Branch On < 0	if(rs<0) pc+=offset*4	000001	rs	00000	offset	
BLTZAL rs,offset	Branch On < 0 And Link	r31=pc; if(rs<0) pc+=offset*4	000001	rs	10000	offset	
BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	000101	rs	rt	offset	
BREAK	Breakpoint	epc=pc; pc=0x3c	000000	code			001101
J target	Jump	pc=pc_upper (target<<2)	000010	target			
JAL target	Jump And Link	r31=pc; pc=target<<2	000011	target			
JALR rs	Jump And Link Register	rd=pc; pc=rs	000000	rs	00000	rd	00000 001001
JR rs	Jump Register	pc=rs	000000	rs	0000000000000000 001000		
MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]	010000	00000	rt	rd	000000000000
MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt	010000	00100	rt	rd	000000000000
RFE	Return From Exeption	Update Status register	010000	10X00	rt	rd	000000000000
SYSCALL	System Call	epc=pc; pc=0x3c	000000	000000000000000000000000			001100
Memory Access							
LB rt,offset(rs)	Load Byte	rt=(char*)(offset+rs)	100000	rs	rt	offset	
LBU rt,offset(rs)	Load Byte Unsigned	rt=(Uchar*)(offset+rs)	100100	rs	rt	offset	
LH rt,offset(rs)	Load Halfword	rt=(short*)(offset+rs)	100001	rs	rt	offset	
LBU rt,offset(rs)	Load Halfword Unsigned	rt=(Ushort*)(offset+rs)	100101	rs	rt	offset	
LW rt,offset(rs)	Load Word	rt=(int*)(offset+rs)	100011	rs	rt	offset	
SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	101000	rs	rt	offset	
SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt	101001	rs	rt	offset	
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011	rs	rt	offset	

**Remarque 1:** Les instructions ci-dessus sont les instructions originaux, les pseudo-instructions sont générées par l'assembleur à partir de ces instructions.

**Remarque 2:** Le jeu d'instructions représenté ici est appelé **MIPS I**.

**Remarque 3:** Les instructions avec l'OpCode 000000 regroupe en tous 28 instructions, ce sont des instructions de type R qu'on peut différencier par le champ *fonction* (les 6 premiers bits).

**Remarque 4:** Même chose pour les instructions avec l'OpCode 000001, ça regroupe 4 instructions de type I, différenciable par le bit 16 et le bit 20 (le premier et le dernier bit du champ rt).

**Remarque 5:** Et de même pour les instructions avec l'OpCode 010000 qui regroupe 3 instructions de type R, différenciables par le bit 23 et le bit 25 (le bit au milieu et le dernier bit du champ rs).

**Remarque 6:** Les Actions dans la table des OpCodes décrivent le fonctionnement de l'instruction en utilisant la syntaxe du langage C++.

### Exercice 03 :

A. En utilisant la table des OpCodes en haut on peut extraire la forme assembleur des instructions suivantes écrites en hexadécimal :

Instruction (hexadécimal)	Instruction (binaire)	OpCode	Décodage de l'instruction
0x24040081	0010 0100 0000 0100 0000 0000 1000 0001	LUI	ADDIU \$4, \$0, 129 ⇔ ADDIU \$a0, \$0, 129
0x34020001	0011 0100 0000 0010 0000 0000 0000 0001	ORI	ORI \$2, \$0, 1 ⇔ ORI \$v0, \$0, 1
0x0000000c	0000 0000 0000 0000 0000 0000 0000 1100	SYSCALL	SYSCALL
0x3402000a	0011 0100 0000 0010 0000 0000 0000 1010	ORI	ORI \$2, \$0, 10 ⇔ ORI \$v0, \$0, 10
0x0000000c	0000 0000 0000 0000 0000 0000 0000 1100	SYSCALL	SYSCALL

**Remarque :** Il faut faire attention à l'ordre dans les registres entre la forme binaire et la forme assembleur dans la table des OpCode, car le registre destination est inversé dans les 2 représentation **ex** : dans l'instruction ADDIU le registre rt en binaire est encodé après rs, alors que dans l'assembleur c'est l'inverse, rs s'écrit avant rt.

Ainsi le programme en assembleur est :

```

ADDIU $a0, $0, 129    # équivalent à li $a0, 129
ORI $v0, $0, 1       # équivalent à li $v0, 1
SYSCALL              # affichage du nombre 129

ORI $v0, $0, 10      # équivalent à li $v0, 10
SYSCALL              # arrêt du programme
    
```

B. Le programme affiche le nombre 129 puis il s'arrête.

### Exercice 04 :

A. La conversion des instructions de la forme binaire (hexadécimal) vers la forme assembleur en suivant la table des OpCodes est comme suite :

Instruction (hexadécimal)	Instruction (binaire)	OpCode	Décodage de l'instruction
0x3c051001	0011 1100 0000 0101 0001 0000 0000 0001	LUI	LUI \$5, 0x1001 ⇔ LUI \$a1, 4097
0x80a60000	1000 0000 1010 0110 0000 0000 0000 0000	LB	LB \$6, 0(\$5) ⇔ LB \$a2, 0(\$a1)
0x10c00007	0001 0000 1100 0000 0000 0000 0000 0111	BEQ	BEQ \$6, \$0, 7 ⇔ BEQ \$a2, \$0, 7
0x30c6000f	0011 0000 1100 0110 0000 0000 0000 1111	ANDI	ANDI \$6, \$6, 15 ⇔ ANDI \$a2, \$a2, 15
0x00043880	0000 0000 0000 0100 0011 1000 1000 0000	SLL	SLL \$7, \$4, 2 ⇔ SLL \$a3, \$a0, 2
0x00872020	0000 0000 1000 0111 0010 0000 0010 0000	ADD	ADD \$4, \$4, \$7 ⇔ ADD \$a0, \$a0, \$a3
0x00042040	0000 0000 0000 0100 0010 0000 0100 0000	SLL	SLL \$4, \$4, 1 ⇔ SLL \$a0, \$a0, 1
0x00862020	0000 0000 1000 0110 0010 0000 0010 0000	ADD	ADD \$4, \$4, \$6 ⇔ ADD \$a0, \$a0, \$a2
0x24a50001	0010 0100 1010 0101 0000 0000 0000 0001	ADDUI	ADDIU \$5, \$5, 1 ⇔ ADDIU \$a1, \$a1, 1
0x08100001	0000 1000 0001 0000 0000 0000 0000 0011	J	J 0x0040_0004

```

0x0040_0000 : LUI  $a1, 0x1001
0x0040_0004 : LB   $a2, 0($a1)
0x0040_0008 : BEQ  $a2, $0, 7
0x0040_000C : ANDI $a2, $a2, 15
0x0040_0010 : SLL  $a3, $a0, 2
0x0040_0014 : ADD  $a0, $a0, $a3
0x0040_0018 : SLL  $a0, $a0, 1
0x0040_001C : ADD  $a0, $a0, $a2
0x0040_0020 : ADDIU $a1, $a1, 1
0x0040_0024 : J    0x0040 000C

```

ainsi avec l'affectation des labels ça devient :

```

                LUI  $a1, 0x1001
                LB   $a2, 0($a1)
                BEQ  $a2, $0, label_2
label_1:        ANDI $a2, $a2, 15
                SLL  $a3, $a0, 2
                ADD  $a0, $a0, $a3
                SLL  $a0, $a0, 1
                ADD  $a0, $a0, $a2
                ADDIU $a1, $a1, 1
                J    label_1
label_2:

```

le segment des instructions (.text = 0x0040)	le segment des données (.data = 0x1001)
<pre> label_1:        LUI  \$a1, 0x1001                 LB   \$a2, 0(\$a1)                 BEQ  \$a2, \$0, label_2                 ANDI \$a2, \$a2, 15                 SLL  \$a3, \$a0, 2                 ADD  \$a0, \$a0, \$a3                 SLL  \$a0, \$a0, 1                 ADD  \$a0, \$a0, \$a2                 ADDIU \$a1, \$a1, 1                 J    label_1 label_2: </pre>	<pre> 0x1001_0000 : 0x31 (0011 0001) 0x1001_0001 : 0x36 (0011 0110) 0x1001_0010 : 0x30 (0011 0000) 0x1001_0011 : 0x33 (0011 0011) </pre>

**Remarque 1:** Il existe 3 types d'instructions de décalage, la différence entre les 3 ces dans le remplissage du bit vide après décalage, pour les instructions SLL (Shift Left Logical) et SRL (Shift Right Logical) inserent toujours le zéro dans le bit vide après décalage. Par-contre le SRA (Shift Right Arithmetic) il insert le bit de signe (ça peut être 0 ou 1), la raison est de garder l'opération division par 2 correct même pour les nombres négatives complément-à-2.

**Remarque 2:** Il est évident de faire noter qu'une instruction de type SRA (Shift Right Logical) n'a pas lieu d'être, en raison qu'il n'a pas de propagation de signe pour un décalage à gauche.

**Remarque 3:** La formule pour calculer l'adresse d'une instruction de branchement (comme BEQ) est  $(PC+4) + (imm*4)$ . Ainsi on constate que dans une instruction de branchement l'immédiate représente réellement le nombre d'instructions à décaler par rapport à l'instruction actuelle, c'est ce qu'on appelle adresse relative.

**Remarque 4:** Par-contre pour l'instruction de Jump c'est une adresse réelle, elle est obtenue par la formule  $(PC+4)_{[31,28]} : (address \ll 2)$ , c'est ce qu'on appelle une adresse absolue. ( : c'est la concaténation, et [31,28] sont les 4 derniers bits de PC+4).

## B. Trace d'exécution :

pas	Instruction	PC	\$a0 (\$4)	\$a1 (\$5)	\$a2(\$6)	\$a3 (\$7)
1	LUI \$a1, 0x1001	0x0040_0000	0x0000_0000	0x0000_0000	0x0000_0000	0x0000_0000
2	LB \$a2, 0(\$a1)	0x0040_0004	-	0x1001_0000	-	-
3	BEQ \$a2, \$0, label_2	0x0040_0008	-	-	0x0000_0031	-
4	ANDI \$a2, \$a2, 15	0x0040_000C	-	-	-	-
5	SLL \$a3, \$a0, 2	0x0040_0010	-	-	0x0000_0001	-
6	ADD \$a0, \$a0, \$a3	0x0040_0014	-	-	-	0x0000_0000
7	SLL \$a0, \$a0, 1	0x0040_0018	0x0000_0000	-	-	-
8	ADD \$a0, \$a0, \$a2	0x0040_001C	0x0000_0000	-	-	-
9	ADDIU \$a1, \$a1, 1	0x0040_0020	0x0000_0001	-	-	-
10	J 0x0040_0004	0x0040_0024	-	0x1001_0001	-	-
11	LB \$a2, 0(\$a1)	0x0040_0004	-	-	-	-
12	BEQ \$a2, \$0, label_2	0x0040_0008	-	-	0x0000_0036	-
13	ANDI \$a2, \$a2, 15	0x0040_000C	-	-	-	-
14	SLL \$a3, \$a0, 2	0x0040_0010	-	-	0x0000_0006	-
15	ADD \$a0, \$a0, \$a3	0x0040_0014	-	-	-	0x0000_0004
16	SLL \$a0, \$a0, 1	0x0040_0018	0x0000_0005	-	-	-
17	ADD \$a0, \$a0, \$a2	0x0040_001C	0x0000_000A	-	-	-
18	ADDIU \$a1, \$a1, 1	0x0040_0020	0x0000_0010	-	-	-
19	J 0x0040_0004	0x0040_0024	-	0x1001_0002	-	-
20	LB \$a2, 0(\$a1)	0x0040_0004	-	-	-	-
21	BEQ \$a2, \$0, label_2	0x0040_0008	-	-	0x0000_0030	-
22	ANDI \$a2, \$a2, 15	0x0040_000C	-	-	-	-
23	SLL \$a3, \$a0, 2	0x0040_0010	-	-	0x0000_0000	-
24	ADD \$a0, \$a0, \$a3	0x0040_0014	-	-	-	0x0000_0040
25	SLL \$a0, \$a0, 1	0x0040_0018	0x0000_0050	-	-	-
26	ADD \$a0, \$a0, \$a2	0x0040_001C	0x0000_00A0	-	-	-
27	ADDIU \$a1, \$a1, 1	0x0040_0020	0x0000_00A0	-	-	-
28	J 0x0040_0004	0x0040_0024	-	0x1001_0003	-	-

29	LB \$a2, 0(\$a1)	0x0040_0004	-	-	-	-
30	BEQ \$a2, \$0, label_2	0x0040_0008	-	-	0x0000_0033	-
31	ANDI \$a2, \$a2, 15	0x0040_000C	-	-	-	-
32	SLL \$a3, \$a0, 2	0x0040_0010	-	-	0x0000_0003	-
33	ADD \$a0, \$a0, \$a3	0x0040_0014	-	-	-	0x0000_0280
34	SLL \$a0, \$a0, 1	0x0040_0018	0x0000_0320	-	-	-
35	ADD \$a0, \$a0, \$a2	0x0040_001C	0x0000_0640	-	-	-
36	ADDIU \$a1, \$a1, 1	0x0040_0020	0x0000_0643	-	-	-
37	J 0x0040_0004	0x0040_0024	-	0x1001_0004	-	-
38	LB \$a2, 0(\$a1)	0x0040_0004	-	-	-	-
39	BEQ \$a2, \$0, label_2	0x0040_0008	-	-	0x0000_0000	-
40	NOP	0x0040_0028	-	-	-	-
		<b>0x0040_0028</b>	<b>0x0000_0643</b>	<b>0x1001_0004</b>	<b>0x0000_0000</b>	<b>0x0000_0280</b>

**Les Tableaux dans MIPS:** 2 manières différentes sont possibles pour la déclaration des tableaux :

**exemple 1:** la déclaration d'un tableaux de 3 entiers est faite comme suite :

```
.data
tab: .word 0:3 # équivalent en C++ à : int tab[3] ;
```

**exemple 2:** la même déclaration d'un tableaux de 3 entiers d'une autre manière :

```
.data
tab: .space 12 # on déclare un espace mémoire de 12 octets, chaque entier en 4 octets
```

**Remarque :** Réellement la directive `.space` n'est pas réservée uniquement pour la déclaration des tableaux, elle est prédestinée pour la déclaration de n'importe quel structure de donnée complexe, comme les *structures*, les *unions* et les *classes*.

L'utilisation des tableaux est aussi un peu particulière dans MIPS, dans le sens où l'accès aux cellules se fait à travers une adresse mémoire qui va représenter l'*indice* du tableau :

**exemple 3:** le code suivant en C++ pour accéder aux cellules du tableau est traduit par le code qui le suit en MIPS :

```
int tab[3] ;

for(int i = 0 ; i < 3 ; i++)
{
    tab[i] = i ;
}
```

en langage MIPS ça donnerait :

```
.data
tab: .word 0 : 3          # équivalent à int tab[3] ;

.text
    la    $t0, tab        # mettre l'adresse de tab dans t0
    li    $t1, 0          # équivalent à i = 0 ; t1 représente la variable i
    li    $t2, 3          # t2 reçoit la valeur 3 pour la comparaison dans la boucle
                                # sachant que bge fait la comparaison entre 2 registres

for:  bge $t1, $t2, end_for # le test de la boucle for. c'est l'inverse du C++

    sw    $t1, 0($t0)     # équivalent à tab[i] = i;
    addi  $t1, $t1, 1     # équivalent à i++;
    addi  $t0, $t0, 4     # on doit incrémenter l'adresse par 4 pour pointer sur la
                                # cellule suivante

    j     for             # retour inconditionnel au début de la boucle
end_for:                  # le label de sortie de la boucle
```

**Remarque :** Il impératif lors de la manipulation des tableaux de créer un registre qui contient les adresses des cellules pour avoir toujours un pointeur sur la cellule à traiter dans le tableau, dans cette exemple c'était t0.