Chapitre IV : Le processeur

II.1 Introduction :

L'architecture externe représente ce que doit connaitre un programmeur souhaitant programmer en assembleur, ou la personne souhaitant écrire un compilateur pour ce processeur:

- Les registres visibles.
- L'adressage de la mémoire.
- Le jeu d'instructions.
- Les mécanismes de traitement des interruptions et exceptions.

MIPS (Microprocessor without Interlocked Pipeline Stages) est un processeur 32 bits industriel conçu dans les années 80. Son jeu d'instructions est de type RISC (Reduced instruction set computer) développée par la compagnie MIPS Computer Systems Inc.
Il existe plusieurs réalisations industrielles de cette architecture (SIEMENS, NEC, LSI LOGIC, SILICON GRAPHICS, etc...)

On les retrouve aussi dans plusieurs systèmes embarqués, comme les ordinateurs de poche, les routeurs Cisco et les consoles de jeux vidéo (Nintendo 64 et Sony PlayStation, PlayStation 2 et PSP).

Le MIPS R4000 sorti en 1991 serait le premier processeur 64 bits. Il a été supporté par Microsoft de Windows NT 3.1 jusqu'à Windows NT 4.0

II.2Registres Visibles

Tous les registres visibles du logiciel, c'est à dire ceux dont la valeur peut être lue ou modifiée par les instructions, sont des registres 32 bits.
Afin de mettre en œuvre les mécanismes de protection nécessaires pour un système d'exploitation multi-tâches, le processeur possède deux modes de fonctionnement : utilisateur/superviseur. Ces deux modes de fonctionnement imposent d'avoir deux catégories de registres.

1) Registres non protégés
2) Registres protégés.

II.2.1 Registres non protégés :

Le processeur possède 35 registres manipulés par les instructions standard (c'est à dire les instructions qui peuvent s'exécuter aussi bien en mode utilisateur qu'en mode superviseur).

• **Ri** ($0 \leq i \leq 31$) 32 registres généraux.
Ces registres sont directement adressés par les instructions, et permettent de stocker des résultats de calculs intermédiaires.
Le registre **R0** est un registre particulier:
- la lecture fournit la valeur constante "0x00000000"
- l'écriture ne modifie pas son contenu.

Le registre **R31** est utilisé par les instructions d'appel de procédures (instructions **BGEZAL**, **BLTZAL**, **JAL** et **JALR**) pour sauvegarder l'adresse de retour.
• **PC** Registre compteur de programme (Program Counter).

Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.
• **HI et LO** Registres pour la multiplication ou la division. Ces deux registres 32 bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division, qui est un mot de 64 bits.
La description des 32 registres est donnée dans le tableau suivant :

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	reserved for the assembler
$v0–$v1	2–3	values for results and expression evaluation
$a0–$a3	4–7	arguments
$t0–$t7	8–15	temporaries
$s0–$s7	16–23	saved
$t8–$t9	24–25	more temporaries
$k0–$k1	26–27	reserved for the operating system
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

II.2.2 Registres protégés
L'architecture **MIPS** 4 registres qui ne sont accessibles, en lecture comme en écriture, que par les instructions privilégiées (c'est à dire les instructions qui ne peuvent être exécutées qu'en mode superviseur) dédiés pour la gestion des interruptions et des exceptions.

• **SR** Registre d'état (Status Register).
Il contient en particulier le bit qui définit le mode : superviseur ou utilisateur, ainsi que les bits de masquage des interruptions. (Ce registre possède le numéro 12)
• **CR** Registre de cause (Cause Register). En cas d'interruption ou d'exception, son contenu définit la cause pour laquelle on fait appel au programme de traitement des interruptions et des exceptions. (Ce registre possède le numéro 13)
• **EPC** Registre d'exception (Exception Program Counter).
Il contient l'adresse de retour (PC + 4) en cas d'interruption. Il contient l'adresse de l'instruction fautive en cas d'exception (PC). (Ce registre possède le numéro 14)
• **BAR** Registre d'adresse illégale (Bad Address Register).
En cas d'exception de type "adresse illégale", il contient la valeur de l'adresse mal formée. (Ce registre possède le numéro 8)

II.3 Adressage Mémoire :
- L'unité adressable est l'octet.
- La mémoire est vue comme un tableau d'octets qui contient les données et les instructions.
- Les adresses sont codées sur 32 bits (CO =32).
- Les instructions sont codées sur 32 bits (RI=32).
- Les échanges de données avec la mémoire se font par mot (4 octets consécutifs), demi-mot (2 octets consécutifs), ou par octet.

- L'adresse d'un mot de donnée ou d'une instruction doit être multiple de 4. L'adresse d'un demi-mot doit être multiple de 2. (on dit que les adresses doivent être "alignées").

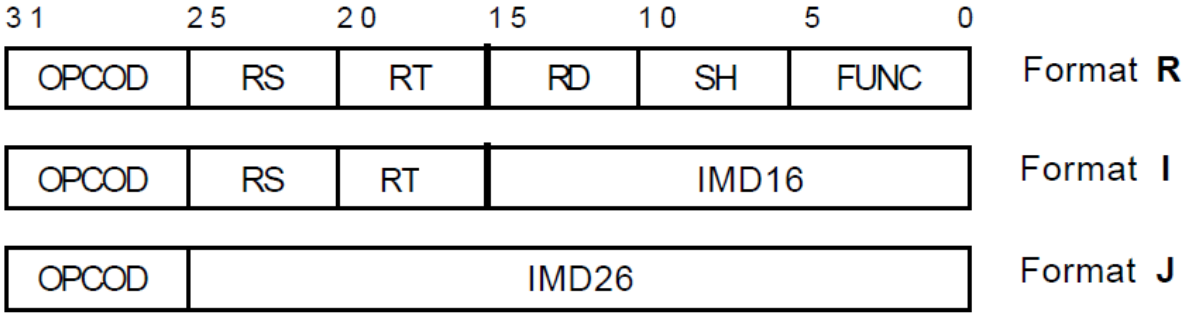
II.4 Jeu d'instructions :

a. Généralités :

Le processeur possède 57 instructions qui se répartissent en 4 classes :

• 33 instructions arithmétiques/logiques entre registres
• 12 instructions de branchement
• 7 instructions de lecture/écriture mémoire
• 5 instructions systèmes

Toutes les instructions ont une longueur de 32 bits et possèdent un des trois formats suivants :



- Le format **J** n'est utilisé que pour les branchements à longue distance (inconditionnels).
- Le format **I** est utilisé par les instructions de lecture/écriture mémoire, par les instructions utilisant un opérande immédiat, ainsi que par les branchements courte distance (conditionnels).
- Le format **R** est utilisé par les instructions nécessitant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD.

Le jeu d'instructions est "orienté registres". Cela signifie que les instructions arithmétiques et logiques prennent leurs opérandes dans des registres et rangent le résultat dans un registre. Les seules instructions permettant de lire ou d'écrire des données en mémoire effectuent un simple transfert entre un registre général et la mémoire, sans aucun traitement arithmétique ou logique.

La plupart des instructions arithmétiques et logiques se présentent sous les 2 formes registre-registre et registre-immédiat:

ADD : R(rd) <--- R(rs) op R(rt) format R
ADDI : R(rt) <--- R(rs) op IMD format I

L'opérande immédiat 16 bits est signé pour les opérations arithmétiques et non signé pour les opérations logiques.
Les instructions JAL, JALR, BGEZAL, et BLTZAL sauvegardent une adresse de retour dans le registre R31. Ces instructions sont utilisées pour les appels de sous programme.

Toutes les instructions de branchement conditionnel sont relatives au compteur ordinal pour que le code soit translatable. L'adresse de saut est le résultat d'une addition entre la valeur du compteur ordinal et un déplacement signé.

Les instructions MTC0 et MFC0 permettent de transférer le contenu des registres SR, CR, EPC et BAR vers un registre général et inversement. Ces 2 instructions ne peuvent être exécutées qu'en mode superviseur, de même que l'instruction RFE qui permet de restaurer l'état antérieur du registre d'état avant de sortir du gestionnaire d'exceptions.

Dans cette partie on verra le langage d'assembleur qui comporte ce jeu d'instructions.
Avant d'expliquer la syntaxe de certaines instructions on donne certaines règles syntaxiques

Les commentaires :
Ils commencent par un # ou un ; et s'achèvent à la fin de la ligne courante.

Exemple :
```
################################################ ###
# programme qui calcule la somme des n premiers nombres
################################################
.... ; sauve la valeur copiée dans la mémoire
```

Les entiers :
Une valeur entière décimale est notée par exemple 3250, une valeur entière octale est notée 04372 (préfixée par un zéro), et une valeur entière hexadécimale est notée 0xFA (préfixée par zéro suivi de x). En hexadécimal, les lettres de A à F peuvent être écrites en majuscule ou en minuscule.

Les chaînes de caractères :
Elles sont simplement entre guillemets.

Exemple : "la valeur de x est"

Les labels (étiquettes):
Ce sont des chaînes de caractères qui commencent par une lettre, majuscule ou minuscule, un $, un _, ou un .. Ensuite, un nombre quelconque de ces mêmes caractères auquel on ajoute les chiffres. Pour la déclaration, le label doit être suffixé par<< : >>. Exemple : ETIQ1

Attention : sont illégaux les labels qui ont le même nom qu'un mnémonique de l'assembleur ou qu'un nom de registre.

Les immédiats :
Ce sont les opérandes contenus dans l'instruction (constantes :soit des entiers, soit des labels). Ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui l'utilise : 16 ou 26 bits.

Instructions arithmétiques /logiques entre registres			
assembleur	opération		format
Add Rd, Rs, Rt	Add overflow detection	Rd <- Rs + Rt	R
Sub Rd, Rs, Rt	Subtract overflow detection	Rd <- Rs - Rt	R
Addu Rd, Rs, Rt	Add no overflow	Rd <- Rs + Rt	R
Subu Rd, Rs, Rt	Subtract no overflow	Rd <- Rs - Rt	R
Addi Rt, Rs,I	Add Immediate overflow detection	Rt <- Rs + I	I
Addiu Rt, Rs,I	Add Immediate no overflow	Rt <- Rs + I	I

ADDI -- *Add immediate (with overflow)*
Opération : $t=$s+imm
Syntaxe : addi $t, $s, imm

ADDIU -- Add immediate unsigned (no overflow)
Opération : $t=$s+imm
Syntaxe : addiu $t, $s, imm

ADDU -- *Add unsigned (no overflow)*
Operation: $d = $s + $t;
Syntax: addu $d, $s, $t

SUBU -- *Subtract unsigned*
Operation: $d = $s - $t;
Syntax: subu $d, $s, $t

Instructions arithmétiques /logiques entre registres			
assembleur	opération		format
Or Rd, Rs, Rt	Logical Or	Rd <- Rs or Rt	R
And Rd, Rs, Rt	Logical And	Rd <- Rs and Rt	R
Xor Rd, Rs Rt	Logical Exclusive-Or	Rd <- Rs xor Rt	R
Nor Rd, Rs, Rt	Logical Not Or	Rd <- Rs nor Rt	R
Ori Rt, Rs, I	Or Immediate unsigned immediate	Rt <- Rs or I	I
Andi Rt, Rs, I	And Immediate unsigned immediate	Rt <- Rs and I	I
Xori Rt, Rs, I	Exclusive-Or Immediate unsigned immediate	Rt <- Rs xor I	I

And (Et logique) :
Et bit-à-bit registre registre

Syntaxe : and $rd, $rs, $rt

Un **et** bit-à-bit est effectué entre les contenus des registres **$rs** et **$rt**. Le résultat est placé dans le registre **$rd**.

Andi (Et logique immédiat) :
Et bit-à-bit registre immédiat
Syntaxe : andi $rd, $rs, imm

La valeur immédiate sur 16 bits subit une extension de zéros. Un et bit à bit est effectué entre cette valeur étendue et le contenu du registre **$rs** pour former un résultat placé dans le registre **$rd**.

Or (Ou logique)
Ou bit-à-bit registre registre
Syntaxe : or $rd, $rs, $rt

Un **ou** bit-à-bit est effectué entre les contenus des registres **$rs** et **$rt**. Le résultat est placé dans le registre **$rd**.

Ori (Ou logique immédiat)
Ou bit-à-bit registre, immédiat
Syntaxe ori $rd, $rs, imm

La valeur immédiate sur 16 bits subit une extension de zéros. Un **ou** bit-à bite est effectué entre cette valeur étendue et le contenu du registre **$rs** pour former un résultat placé dans le registre **$rd**.

Instructions arithmétiques /logiques entre registres		
assembleur	opération	format
Sllv Rd, Rt, Rs	Shift Left Logical Variable Rd <- Rt << Rs 5 lsb of Rs is significant	R
Srlv Rd, Rt, Rs	Shift Right Logical Variable Rd <- Rt >> Rs 5 lsb of Rs is significant	R
Srav Rd, Rt, Rs	Shift Right Arithmetical Variable Rd <- Rt >>* Rs 5 lsb of Rs is significant	R
Sll Rd, Rt, sh	Shift Left Logical Rd <- Rt << sh	R
Srl Rd, Rt, sh	Shift Right Logical Rd <- Rt >> sh	R
Sra Rd, Rt, sh	Shift Right Arithmetical Rd <- Rt >>* sh * : with sign extension	R

Décalage logique à gauche.
Décalage à gauche immédiat
Syntaxe : sll $rd, $rt, imm

Le registre **$rt** est décalé à gauche de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre **$rd**.

Décalage à gauche registre(variable)
Syntaxe sllv $rd, $rs, $rt

Le registre **$rs** est décalé à gauche du nombre de bits spécifiés dans les 5 bits de poids faible du registre **$rt**, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre **$rd**

Décalage à droite logique immédiat
Syntaxe : srl $rd, $rt, imm

Le registre **$rt** est décalé à droite de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids fort. Le résultat est placé dans le registre **$rd**.

Décalage à droite logique registre
Syntaxe srlv $rd, $rs, $rt

Le registre **$rs** est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre **$rt**, des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre **$rd**.

Exercice :
Ecrire un programme en assembleur qui permet de lire un entier au clavier et qui calcule 2 à la puissance ce nombre, vous aurez besoin de sll, beq, b et addi.

Solution:
```
.data
nombre1 :.word 1
nombre2: .word 2
entree : .asciiz "donnez la valeur du nombre SVP:"
sortie : .asciiz "la puissance est :"
.text
lw $t1, nombre1
lw $t2, nombre2

li $v0, 4
la $a0, entree
syscall

li $v0, 5
syscall
move $t3, $v0

calcul : beq $t3, $t1, fin
sll $t2, $t2, 1
addi $t1, $t1, 1
b calcul

fin :
li $v0, 4
la $a0, sortie
```

```
syscall
li $v0, 1
move $a0, $t2
syscall
li $v0, 10
syscall
```

Instructions arithmétiques /logiques entre registres		
assembleur	opération	format
Slt Rd, Rs, Rt	Set if Less Than — Rd <- 1 if Rs<Rt else 0	R
Sltu Rd, Rs, Rt	Set if Less Than Unsigned — Rd <- 1 if Rs<Rt else 0	R
Slti Rt, Rs, I	Set if Less Than Immediate sign extended Immediate — Rt <- 1 if Rs < I else 0	I
Sltiu Rt, Rs, I	Set if Less Than Immediate unsigned immediate — Rt <- 1 if Rs < I else 0	I

Instructions de comparaison

Comparaison signée registre registre

Syntaxe slt $rd, $rs, $rt

Le contenu du registre **$rs** est comparé au contenu du registre **$rt**, les deux valeurs étant considérées comme des quantités signées. Si la valeur contenue dans **$rs** est inférieure à celle contenue dans **$rt**, alors **$rd** prend la valeur un, sinon il prend la valeur zéro.

Comparaison signée registre immédiat

Syntaxe slti $rd, $rs, imm

Le contenu du registre **$rs** est comparé à la valeur immédiate sur 16 bits qui à subit une extension de signe. Les deux valeurs étant considérées comme des quantités signées, si la valeur contenue dans **$rs** est inférieure à celle de l'immédiat étendu, alors **$rd** prend la valeur un, sinon il prend la valeur zéro.

Instructions arithmétiques /logiques entre registres		
assembleur	opération	format
Mult Rs, Rt	Multiply — Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Multu Rs, Rt	Multiply Unsigned — Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Div Rs, Rt	Divide — Rs / Rt LO <- Quotient HI <- Remainder	R
Divu Rs, Rt	Divide Unsigned — Rs / Rt LO <- Quotient HI <- Remainder	R

La multiplication
Multiplication signée
Syntaxe : mult $rs, $rt
Le contenu du registre $rs est multiplié par le contenu du registre $rt, le contenu des deux registres considérés comme des nombres en complément à deux. Les 32 bits de poids fort du résultat sont placés dans le registre hi et les 32 bits de poids faible dans le registre lo.

Multiplication non-signée
Syntaxe : multu $ri, $rj
Meme comportement que mult sauf que les nombres sont non signés

La division
Division entière registre registre et reste signé
Syntaxe : div $rs, $rt
Le contenu du registre **$rs** est divisé par le contenu du registre **$rt**, le contenu des deux registres étant considéré comme des nombres en complément à deux. Le résultat de la division est placé dans le registre spécial **$lo**, et le reste dans **$hi**.

Division entière et reste non-signé registre registre
Syntaxe : Divu $rs, $rt
Le contenu du registre **$rs** est divisé par le contenu du registre **$rt**, le contenu des deux registres étant considéré comme des nombres non signés. Le résultat de la division est placé dans le registre spécial **$lo**, et le reste dans **$hi**.

Instructions arithmétiques /logiques entre registres			
assembleur	opération		format
Mfhi Rd	Move From HI	Rd <- HI	R
Mflo Rd	Move From LO	Rd <- LO	R
Mthi Rs	Move To HI	HI <- Rs	R
Mtlo Rs	Move To LO	LO <- Rs	R

Copie du registre $hi dans un registre général
Syntaxe mfhi $rd

Le contenu du registre spécialisé **$hi** —qui est mis à jour par l'opération de multiplication ou de division —est recopié dans le registre général **$rd**.

Copie du registre $lo dans un registre général
Syntaxe : mflo $rd

Le contenu du registre spécialisé **$lo** — qui est mis à jour par l'opération de multiplication ou de division est recopié dans le registre général **$rd**.

Instructions de Branchement			
assembleur	opération		format
Beq Rs, Rt, Label	Branch if Equal	PC <- PC+4+(I*4) if Rs = Rt PC <- PC+4 if Rs ≠ Rt	I
Bne Rs, Rt, Label	Branch if Not Equal	PC <- PC+4+(I*4) if Rs ≠ Rt PC <- PC+4 if Rs = Rt	I
Bgez Rs, Label	Branch if Greater or Equal Zero	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0	I
Bgtz Rs, Label	Branch if Greater Than Zero	PC <- PC+4+(I*4) if Rs > 0 PC <- PC+4 if Rs ≤ 0	I
Blez Rs, Label	Branch if Less or Equal Zero	PC <- PC+4+(I*4) if Rs ≤ 0 PC <- PC + 4 if Rs > 0	I
Bltz Rs, Label	Branch if Less Than Zero	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0	I
Bgezal Rs, Label	Branch if Greater or Equal Zero and link	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0 R31 <- PC+4 in both cases	I
Bltzal Rs, Label	Branch if Less Than Zero and link	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0 R31 <- PC+4 in both cases	I
J Label	Jump	PC <- PC 31:28 || I*4	J
Jal Label	Jump and Link	R31 <- PC+4 PC <- PC 31:28 || I*4	J
Jr Rs	Jump Register	PC <- Rs	R
Jalr Rs	Jump and Link Register	R31 <- PC+4 PC <- Rs	R
Jalr Rd, Rs	Jump and Link Register	Rd <- PC+4 PC <- Rs	R

Branchement conditionnel
Branchement si registre égal registre
Syntaxe beq $rs, $rt, label

Les contenus des registres **$rs** et **$rt** sont comparés. S'ils sont égaux, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.
Addr ← label

Branchement si registre supérieur ou égal à zéro
Syntaxe bgez $ri, label

Si le contenu du registre **$ri** est supérieur ou égal à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

Addr ← label

Branchement si registre différent de registre
Syntaxe bne $rs, $rt, label
Les contenus des registres **$rs** et **$rt** sont comparés. S'ils sont différents, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.
Addr ← label

Branchement inconditionnel
Branchement inconditionnel immédiat
Syntaxe j label

Le programme saute inconditionnellement à l'adresse correspondant au label,
calculé par l'assembleur.

Instructions de lecture/écriture		
assembleur	opération	format
Lw Rt, I (Rs)	Load Word Rt <- M (Rs + I) sign extended Immediate	I
Sw Rt, I (Rs)	Store Word M (Rs + I) <- Rt sign extended Immediate	I

Instruction de chargement rangement dans la mémoire principale
(Lecture-Ecriture)
Les deux instructions lw (load word = lecture) et sw (store word =écriture) permettent les échanges entre la mémoire centrale et les registres.

Lecture d'un mot de la mémoire
Syntaxe lw $rd, imm($rs) rd← mem[imm + rs]

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits et du contenu du registre **$rs**. Le contenu de cette adresse est placé dans le registre **$rd**.

Écriture d'un mot en mémoire
Syntaxe sw $rd, imm($rs) mem[imm + rs]← rd
L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec et du contenu du registre **$rs**. Le contenu du registre **$rd** est écrit à l'adresse ainsi calculée.

Exemple : ,
lw $t2, 10($t3) copie dans le registre $t2 la valeur située dans la mémoire principale à l'adresse *m* obtenue en ajoutant 10 au nombre stocké dans la registre $t3.

sw $t2, 15($t1) copie la valeur présente dans le registre $t2 dans la mémoire principale à l'adresse *m* obtenue en ajoutant 15 au nombre stocké dans la registre $t1.

Structures de contrôle :
Les structures de répétition en assembleur s'effectuent en utilisant les instructions de branchements.

Exercice 1 :

Ecrire un programme en assembleur qui permet d'afficher la suite des nombres de 0 à 10 séparés par des virgules, l'exécution doit ressembler à ceci :

```
0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,la boucle while est faite
-- program is finished running --
```

Correction :

```
.data
message :.asciiz "la boucle while est faite"
espace:.asciiz " ,"
.text
addi $t0, $zero, 0

while:
bgt $t0, 10, fin
jal afficher_nombre
addi $t0, $t0, 1
j while


fin:
li $v0, 4
la $a0, message
syscall
li $v0, 10
syscall

afficher_nombre:
li $v0, 1
add $a0, $t0, $zero
syscall
li $v0, 4
la $a0, espace
syscall

jr $ra
```

Exercice 2 :

Ecrire un programme en assembleur qui permet de lire votre nom au clavier et d'afficher le message suivant à l'écran : salut votre nom.

Correction :

```
.data
message1: .asciiz "Entrez votre nom SVP: "
message2: .asciiz "Salut "
entree_utilisateur: .space 20
.text
li $v0, 4
la $a0, message1
```

```
syscall

li $v0, 8
la $a0, entree_utilisateur
li $a1, 20
syscall

li $v0, 4
la $a0, message2
syscall

li $v0, 4
la $a0, entree_utilisateur
syscall
```

Exercice 3 :

Ecrire un programme en assembleur qui permet de lire 10 notes d'examen ainsi leurs coefficients respectifs au clavier, de calculer la moyenne de l'étudiant et de l'afficher à l'écran.