Chapitre 3 - Instructions, Langage machine et Langage assembleur

Cours de Bernard BoigelotUniversité de Liège

2025

Table des matières

1 Introduction **3**

2 Cycle d'exécution d'une instruction **3**
2.1 Étapes du cycle 3
2.2 Illustration du cycle 4
2.3 Registres utilisés 4
2.4 Exemple simple 4

3 Typologie des instructions **4**
3.1 Instructions arithmétiques et logiques 5
3.2 Instructions mémoire (load/store) 5
3.3 Instructions de contrôle (branchements) 5
3.4 Formats d'instruction 5
3.5 Schéma intégré : Formats d'instruction R, I, J 5
3.6 Exercices types examen 6

4 Langage assembleur vs Langage machine **7**
4.1 Traduction directe (1 :1) 7
4.2 Traduction indirecte : pseudo-instructions 7
4.3 Encodage binaire complet 7

5 Codage des constantes et adresses **9**
5.1 Codage immédiat dans le format I 9
5.2 Décalage pour l'adressage relatif (branchements) 9
5.3 Codage des adresses dans le format J 9
5.4 Résumé pratique : tableau de codage 9

6 Cycle d'exécution des instructions **10**
6.1 Étapes du cycle 10
6.2 Illustration du cycle d'instruction 11
6.3 Exemples d'exécution 11
6.4 Remarques importantes 12

7 Instructions arithmétiques et logiques **13**
7.1 Instructions arithmétiques de base . . . 13
7.2 Instructions logiques . . . 13
7.3 Instructions de décalage . . . 13
7.4 Instructions de comparaison . . . 13
7.5 Schéma : type R en détail . . . 14
7.6 Conseils d'apprentissage . . . 14

8 Instructions de saut et de branchement **15**
8.1 Types de sauts et branchements . . . 15
8.2 Instructions de saut (type J) . . . 15
8.3 Instructions de branchement (type I) . . . 15
8.4 Instructions de saut indirect . . . 15
8.5 Schéma : format des instructions J . . . 15
8.6 Calcul de l'offset pour les branchements . . . 15
8.7 Conseils et erreurs fréquentes . . . 16

9 Appels système et gestion d'entrée/sortie **17**
9.1 Les appels système (syscall) . . . 17
9.2 Principaux appels système . . . 17
9.3 Utilisation de chaînes . . . 17
9.4 Conseils pratiques . . . 17
9.5 Résumé visuel : appels système courants . . . 18

10 Instructions pseudo et synthèse du chapitre **19**
10.1 Instructions pseudo (pseudo-instructions) . . . 19
10.2 Pourquoi les utiliser ? . . . 19
10.3 Résumé du chapitre . . . 19
10.4 Conseils pour réussir les examens . . . 19
10.5 Exercice type examen . . . 20

1 Introduction

Ce chapitre aborde les notions fondamentales de programmation bas niveau, à travers les concepts d'instructions machines, de langage assembleur et de leur lien direct avec l'architecture matérielle. On y détaille le cycle de traitement d'une instruction, les différents types d'opérations (arithmétiques, logiques, de contrôle, de mémoire) et la manière dont elles sont représentées, codées, et exécutées dans une machine réelle.

Objectifs du chapitre

— Comprendre le fonctionnement du cycle instruction exécution.

— Étudier les différentes classes d'instructions (arithmétique, logique, mémoire, branchements).

— Manipuler un jeu d'instructions élémentaire.

— Savoir lire, écrire et traduire du langage assembleur vers le langage machine.

— Visualiser l'impact direct des instructions sur les registres et la mémoire.

— Préparer à la lecture et l'écriture d'un code assembleur simple.

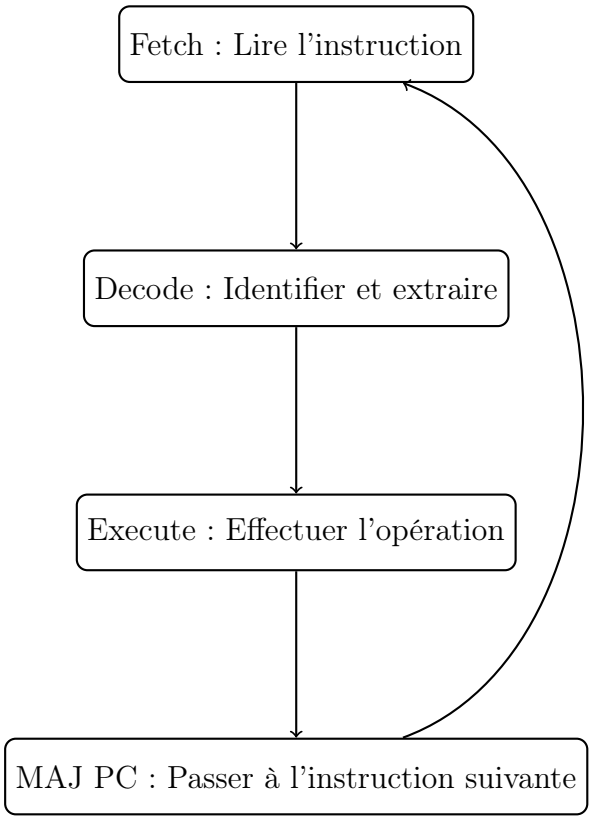
2 Cycle d'exécution d'une instruction

Chaque programme exécuté par un processeur est décomposé en une suite d'instructions élémentaires codées en binaire. Le cycle fondamental de traitement d'une instruction est appelé **cycle fetch-decode-execute**.

2.1 Étapes du cycle

1. **Fetch (chargement)** : lecture de l'instruction à l'adresse pointée par le compteur ordinal (PC).
2. **Decode (décodage)** : identification de l'opération à effectuer et des opérandes nécessaires.
3. **Execute (exécution)** : exécution de l'opération via l'ALU, modification éventuelle des registres ou de la mémoire.

2.2 Illustration du cycle



2.3 Registres utilisés

— **PC (Program Counter)** : contient l'adresse de l'instruction à exécuter.
— **IR (Instruction Register)** : contient l'instruction lue.
— **ALU (Arithmetic Logic Unit)** : réalise les opérations arithmétiques/logiques.
— **Registres généraux** : contiennent les opérandes ou les résultats.

2.4 Exemple simple

— Instruction : `ADD R1, R2, R3` (somme R2 + R3 → R1)
— Étapes :
1. Fetch : PC → mémoire
2. Decode : opcode ADD, R1, R2, R3
3. Execute : ALU effectue l'addition, résultat vers R1
4. Incrément du PC

3 Typologie des instructions

Les instructions en langage machine sont classées en catégories selon leur objectif principal. Chaque catégorie a une structure particulière au niveau des bits utilisés, et chaque instruction implique des registres, des opérandes et parfois une adresse mémoire.

3.1 Instructions arithmétiques et logiques

— Effectuent des calculs sur les registres

— Exemples : `ADD, SUB, MUL, AND, OR, XOR, SLT (Set on Less Than)`

— Format : souvent en **format R** (registre-registre)

Instruction	Opération	Syntaxe	Effet
ADD	Addition entière	ADD R1, R2, R3	R1 = R2 + R3
SUB	Soustraction	SUB R4, R1, R6	R4 = R1 - R6
AND	Et logique	AND R3, R3, R7	R3 = R3 & R7
SLT	Comparaison	SLT R1, R2, R3	R1 = 1 si R2 < R3, sinon 0

3.2 Instructions mémoire (load/store)

— Permettent l'accès à la mémoire principale

— Exemples : `LW, SW` (load word, store word)

— Format : **format I** (immédiat)

— Syntaxe : `LW R1, 0(R2)` (charge depuis [R2+0] dans R1)

Instruction	Action	Exemple
LW	Load Word	LW R5, 4(R6) : R5 ← Mem[R6 + 4]
SW	Store Word	SW R5, 0(R6) : Mem[R6] ← R5

3.3 Instructions de contrôle (branchements)

— Permettent de changer le flux d'exécution

— Exemples : `BEQ, BNE, J`

— **BEQ** : branche si égalité `BEQ R1, R2, offset`

— **J** : saut inconditionnel `J label`

3.4 Formats d'instruction

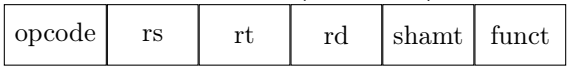
— **Format R** : 6 bits opcode, 3 registres, 5 bits shamt, 6 bits funct

— **Format I** : 6 bits opcode, 2 registres, 16 bits immédiat

— **Format J** : 6 bits opcode, 26 bits adresse

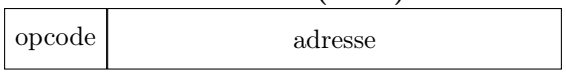
3.5 Schéma intégré : Formats d'instruction R, I, J

Format R (registre)



Format I (immédiat/mémoire)

Format J (saut)



3.6 Exercices types examen

— Q : Quelle est la signification de `ADD R1, R2, R3` ?

— Q : Convertir `LW R4, 8(R5)` en format binaire I.

— Q : Quelle différence entre `BEQ` et `J` ?

4 Langage assembleur vs Langage machine

Le **langage machine** correspond à une séquence de bits directement interprétée par le processeur. C'est un langage binaire, difficilement lisible pour un humain.

Le **langage assembleur** est une notation symbolique proche du langage machine, mais compréhensible pour les programmeurs. Chaque instruction assembleur correspond en général à une ou plusieurs instructions machine (sauf dans le cas des pseudo-instructions).

4.1 Traduction directe (1 :1)

- Exemple :
  - Assembleur : `ADD R1, R2, R3`
  - Machine (format R) : `000000 00010 00011 00001 00000 100000`
- Le code binaire est structuré selon le format R :
  - opcode : 000000
  - rs : 00010 (R2)
  - rt : 00011 (R3)
  - rd : 00001 (R1)
  - shamt : 00000
  - funct : 100000 (ADD)

4.2 Traduction indirecte : pseudo-instructions

Certaines instructions d'assembleur ne correspondent pas directement à une instruction machine unique. Ce sont des **pseudo-instructions**, traduites par l'assembleur en une séquence d'instructions de base.

- Exemple : `LI R1, 5` (load immediate)
  - Ce pseudo-code peut être traduit en :

```
LUI R1, 0        ; Charge le haut de l'immédiat à 0
ORI R1, R1, 5    ; Ajoute le bas de l'immédiat
```

- Autres exemples de pseudo-instructions :
  - `MOVE R1, R2` → `ADD R1, R2, R0`
  - `NOP` → `SLL R0, R0, 0`

4.3 Encodage binaire complet

- Convertir une instruction assembleur en code binaire consiste à :
  1. Identifier le format (R, I, J)
  2. Utiliser la table d'opcode et fonctions
  3. Encoder les registres, décalages ou adresses
- Exemple : `LW R4, 8(R5)`
  - opcode : 100011
  - rs = R5 → 00101

  - rt = R4 → 00100
  - offset = 8 → 0000 0000 0000 1000
- Binaire : `100011 00101 00100 0000000000001000`

5 Codage des constantes et adresses

5.1 Codage immédiat dans le format I

Le champ **immédiat** est une valeur codée sur 16 bits, souvent signée (représentation en complément à deux). Utilisé dans les instructions `LW`, `SW`, `ADDI`, etc.

Exemple :

— `ADDI R1, R2, -4`

— Immédiat = -4 → `1111 1111 1111 1100` (en binaire signé sur 16 bits)

5.2 Décalage pour l'adressage relatif (branchements)

— Instructions `BEQ`, `BNE` utilisent un offset relatif à l'adresse `PC+4`.

— L'offset est exprimé en nombre d'instructions (non en octets).

Exemple :

Adresse source = `0x1000`, Adresse cible = `0x100C` $\Rightarrow$ Offset = $\frac{0x100C - 0x1000}{4} = 3 =$ `0000 0000 0`

5.3 Codage des adresses dans le format J

— L'adresse est codée sur 26 bits.

— Le processeur reconstitue une adresse 32 bits en : `PC[31..28] || adresse || 00`

Exemple :

— `J 0x00400000` → les 26 bits significatifs de l'adresse sont extraits : `000100 000000 000000 000000`

5.4 Résumé pratique : tableau de codage

Type	Taille	Utilisation	Représentation
Immédiat	16 bits	Format I	Complément à deux
Offset de branchement	16 bits	Format I	Relatif à PC+4
Adresse de saut	26 bits	Format J	Concaténé avec PC[31 :28]

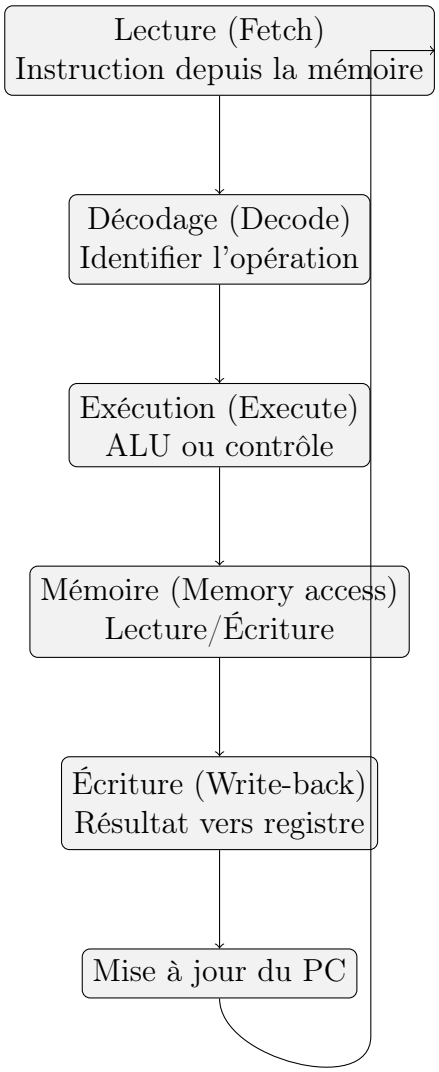
6 Cycle d'exécution des instructions

6.1 Étapes du cycle

Le cycle d'exécution d'une instruction dans un processeur se décompose en plusieurs étapes clés, généralement appelées le « cycle de traitement ». Ces étapes sont essentielles pour comprendre le fonctionnement d'un processeur et son interaction avec la mémoire.

1. **Fetch (Lecture de l'instruction)** : le processeur lit l'instruction située à l'adresse pointée par le compteur ordinal (PC - Program Counter).
2. **Decode (Décodage)** : l'instruction est analysée pour déterminer l'opération à effectuer et les opérandes à utiliser.
3. **Execute (Exécution)** : l'opération est réalisée par l'ALU (Unité arithmétique et logique), comme une addition ou un test de condition.
4. **Memory access (Accès mémoire)** : si l'instruction implique une lecture ou une écriture en mémoire, cette opération est effectuée.
5. **Write-back (Écriture du résultat)** : le résultat de l'instruction est stocké dans un registre (ou la mémoire).
6. **Update PC (Mise à jour du PC)** : le compteur ordinal est mis à jour pour pointer vers l'instruction suivante.

6.2 Illustration du cycle d'instruction



6.3 Exemples d'exécution

Exemple 1 : Instruction `add $t0, $t1, $t2`

- Fetch : Récupérer `add $t0, $t1, $t2` depuis la mémoire
- Decode : Type R, opérateurs $t1 et $t2, destination $t0
- Execute : Addition $t1 + $t2
- Memory : Pas d'accès mémoire
- Write-back : Stocker le résultat dans $t0
- Mise à jour du PC : PC = PC + 4

Exemple 2 : Instruction `lw $t0, 4($sp)`

- Fetch : Lire l'instruction `lw`
- Decode : Type I, base $sp, offset 4
- Execute : Calculer $sp + 4
- Memory : Lire à l'adresse obtenue

— Write-back : Stocker la valeur en $t0
— Mise à jour du PC : PC = PC + 4

Exemple 3 : Instruction `beq $t0, $t1, offset`
— Fetch : Lire `beq`
— Decode : Type I, comparer $t0 et $t1
— Execute : Comparaison
— Memory : Pas d'accès
— Write-back : Rien à écrire
— Mise à jour du PC : saut si égalité $\Rightarrow$ PC = PC + 4 + offset

6.4 Remarques importantes

— Certaines instructions peuvent sauter des étapes (ex : `j` ne fait pas de write-back).
— Le cycle peut être étendu dans un pipeline (chapitre 4).

7 Instructions arithmétiques et logiques

7.1 Instructions arithmétiques de base

Les instructions arithmétiques permettent de réaliser des opérations comme l'addition, la soustraction, etc. Elles sont de type R lorsque les trois opérandes sont des registres.

- `ADD $d, $s, $t` : Additionne $s et $t, stocke le résultat dans $d.
- `SUB $d, $s, $t` : Soustrait $t de $s, stocke le résultat dans $d.
- `ADDU`, `SUBU` : Versions non signées.
- `ADDI $t, $s, imm` : Addition avec immédiat (imm sur 16 bits).

Exemples :

- `ADD $t0, $t1, $t2` → $t0 = $t1 + $t2
- `ADDI $t0, $t1, -5` → $t0 = $t1 - 5
- `SUBU $s0, $s1, $s2` → $s0 = $s1 - $s2 (sans détection d'overflow)

7.2 Instructions logiques

Elles permettent de manipuler les bits dans les registres.

- `AND $d, $s, $t` : ET logique
- `OR $d, $s, $t` : OU logique
- `XOR $d, $s, $t` : OU exclusif
- `NOR $d, $s, $t` : NON OU
- `ANDI, ORI, XORI` : versions avec immédiat

Exemples :

- `AND $t0, $t1, $t2` : $t0 = $t1 & $t2
- `ORI $t0, $t1, 0x0F` : $t0 = $t1 | 0x0000000F

7.3 Instructions de décalage

Elles décalent les bits d'un registre vers la gauche ou la droite.

- `SLL $d, $t, shamt` : décalage à gauche logique
- `SRL $d, $t, shamt` : décalage à droite logique
- `SRA $d, $t, shamt` : décalage à droite arithmétique

Exemples :

- `SLL $t0, $t1, 2` : $t0 = $t1 « 2 (multiplie par 4)
- `SRL $t0, $t1, 1` : $t0 = $t1 / 2 (si unsigned)

7.4 Instructions de comparaison

- `SLT $d, $s, $t` : $d = 1 si $s < $t, sinon $d = 0
- `SLTI $t, $s, imm` : $t = 1 si $s < imm

Exemples :

— `SLT $t0, $t1, $t2` : $t0 = 1 si $t1 < $t2, sinon 0

— `SLTI $t0, $t1, 5` : $t0 = 1 si $t1 < 5

7.5 Schéma : type R en détail

Champ	Bits	Signification
Opcode	6	000000 (fixe pour R)
Rs	5	Registre source $s
Rt	5	Registre source $t
Rd	5	Registre destination $d
Shamt	5	Décalage (souvent 0 sauf SLL/SRL)
Funct	6	Spécifie l'opération (ex : 100000 pour ADD)

7.6 Conseils d'apprentissage

— Apprenez les fonctions `funct` pour les instructions R (ex : ADD = 100000, SUB = 100010).

— Pratiquez avec des conversions binaires et hexadécimales.

— Testez en simulateur MIPS comme QtSPIM.

— Les instructions avec immédiat sont très fréquentes en réalité (optimisation du code).

8 Instructions de saut et de branchement

8.1 Types de sauts et branchements

Ces instructions modifient le compteur ordinal (PC) pour modifier le flot d'exécution du programme. Elles permettent d'implémenter les boucles, les conditions, etc.

— **Sauts inconditionnels (type J)** : saut direct à une adresse.

— **Branchements conditionnels (type I)** : saut si une condition est vraie.

— **Appels de procédure** : saut avec sauvegarde du retour.

— **Retour de procédure** : revenir à l'instruction suivante après un appel.

8.2 Instructions de saut (type J)

— `J label` : saut inconditionnel vers l'étiquette spécifiée.

— `JAL label` : saut vers le label et sauvegarde de l'adresse de retour dans $ra.

Exemples :

— `J loop` : saute à l'étiquette `loop`

— `JAL function` : saute à `function`, $ra = PC + 4

8.3 Instructions de branchement (type I)

— `BEQ $s, $t, offset` : saut si $s == $t

— `BNE $s, $t, offset` : saut si $s $\neq$ $t

Exemples :

— `BEQ $t0, $t1, label` : saute à `label` si $t0 == $t1

— `BNE $s1, $s2, next` : saute à `next` si $s1 $\neq$ $s2

8.4 Instructions de saut indirect

— `JR $ra` : saut à l'adresse contenue dans $ra (retour de fonction)

— `JALR $d, $s` : saute à $s, stocke retour dans $d

8.5 Schéma : format des instructions J

Champ	Bits	Signification
Opcode	6	Ex : 000010 pour J, 000011 pour JAL
Address	26	Adresse cible (concaténée avec bits PC)

8.6 Calcul de l'offset pour les branchements

L'offset utilisé dans les instructions BEQ et BNE est relatif :

— On prend la différence entre l'adresse cible et l'adresse suivante (PC + 4)

— On divise par 4 (taille d'une instruction)

Exemple :

— PC courant = 0x1000, label = 0x100C

— Offset = (0x100C - 0x1004) / 4 = 2 → codé sur 16 bits

8.7 Conseils et erreurs fréquentes

— Toujours utiliser un label défini sinon erreur à l'assemblage.

— Les sauts de type J ne contiennent pas l'adresse complète (seulement 26 bits).

— Attention à l'alignement : les adresses doivent être multiples de 4.

— Le registre $ra contient l'adresse de retour après un JAL.

9 Appels système et gestion d'entrée/sortie

9.1 Les appels système (syscall)

L'appel système permet d'interagir avec l'environnement externe (affichage, saisie, fichiers, etc.) en utilisant le simulateur MIPS (QtSPIM ou Mars).

Fonctionnement général :

- Mettre le numéro de l'appel système dans $v0.
- Fournir les arguments nécessaires dans $a0, $a1, etc.
- Utiliser l'instruction `syscall`.
- Le résultat (si applicable) est renvoyé dans $v0.

9.2 Principaux appels système

Code	Description	Arguments
1	Afficher un entier	$a0 = entier
4	Afficher une chaîne	$a0 = adresse chaîne
5	Lire un entier	(résultat dans $v0)
8	Lire une chaîne	$a0 = adresse, $a1 = taille max
10	Quitter le programme	aucun

Exemples :

- Affichage :

```
li $v0, 1
li $a0, 42
syscall
```

- Lecture :

```
li $v0, 5
syscall
move $t0, $v0  # stocke le résultat dans $t0
```

9.3 Utilisation de chaînes

- Les chaînes sont définies dans la section `.data` avec l'instruction `.asciiz`.
- Exemple : `msg: .asciiz "Hello world!"`
- Pour afficher une chaîne, mettre son adresse dans $a0.

9.4 Conseils pratiques

- Bien distinguer les appels système par leur numéro.
- Toujours initialiser les registres utilisés avant chaque syscall.
- Utilisez `.data` pour les chaînes, `.text` pour le code.
- Préférez des labels explicites (`msg`, `buffer`, etc.).

9.5 Résumé visuel : appels système courants

Opération	**$v0**	**Registres utilisés**
Afficher un entier	1	$a0 = valeur à afficher
Afficher une chaîne	4	$a0 = adresse de la chaîne
Lire un entier	5	résultat dans $v0
Lire une chaîne	8	$a0 = adresse, $a1 = taille max
Quitter	10	aucun

10 Instructions pseudo et synthèse du chapitre

10.1 Instructions pseudo (pseudo-instructions)

Les pseudo-instructions sont des instructions simplifiées offertes par l'assembleur pour faciliter l'écriture du code, bien qu'elles ne soient pas directement exécutables par le processeur. Elles sont converties en une ou plusieurs instructions réelles.

Exemples courants :

Pseudo-instruction	Instructions réelles générées
`LI $t0, 5`	`ORI $t0, $zero, 5`
`MOVE $t1, $t2`	`ADD $t1, $t2, $zero`
`BGT $s1, $s2, label`	`SLT $at, $s2, $s1; BNE $at, $zero, label`

10.2 Pourquoi les utiliser ?

— Simplifie la lecture et l'écriture du code.

— Rend le code plus intuitif.

— Évite les erreurs manuelles de gestion de registres.

Astuce : On peut utiliser le compilateur MIPS pour voir comment une pseudo-instruction est traduite en instructions natives.

10.3 Résumé du chapitre

— MIPS est une architecture à jeu d'instructions réduit (RISC).

— Trois types d'instructions : R, I, J.

— Utilisation des registres $t, $s, $a, $v, etc.

— Appels système via `syscall` et code $v0.

— Instructions de branchement/saut pour les conditions/boucles.

— Instructions arithmétiques/bit à bit selon les formats R et I.

— Pseudo-instructions pour simplifier la programmation.

10.4 Conseils pour réussir les examens

— Savoir identifier le type d'instruction (R, I, J).

— Comprendre comment calculer les offsets.

— Connaître les registres standards $a0-$a3, $v0-$v1, $t0-$t9, $s0-$s7, $ra, $sp.

— Utiliser les schémas pour vérifier les champs d'instruction.

— Apprendre à utiliser les syscalls les plus fréquents (1, 4, 5, 8, 10).

— S'exercer avec QtSPIM ou Mars sur des petits programmes (affichage, boucles, conditions).

10.5 Exercice type examen

Écrivez un programme MIPS qui lit un entier, le double si c'est un nombre pair, sinon affiche le triple. Utilisez des syscalls et affichez le résultat.

Solution :

```
li $v0, 5       # lecture
syscall
move $t0, $v0
andi $t1, $t0, 1
beq $t1, $zero, pair
# impair
mul $t2, $t0, 3
j fin
pair:
mul $t2, $t0, 2
fin:
li $v0, 1
move $a0, $t2
syscall
```