Chapitre 2

Le MIPS et son langage assembleur

MIPS, pour *Microprocessor without Interlocked Pipeline Stages*, est un microprocesseur RISC 32 bits. RISC signifie qu'il possède un jeu d'instructions réduit (*Reduced Instruction Set Computer*) mais qu'en contrepartie, il est capable de terminer l'exécution d'une instruction à chaque cycle d'horloge. Les processeurs MIPS sont notamment utilisés dans des stations de travail (Silicon Graphics, DEC...), plusieurs systèmes embarqués (Palm, modems...), dans les appareils TV HIFI et vidéo, les imprimantes, les routeurs, dans l'automobile et dans de nombreuses consoles de jeux (Nintendo 64, Sony PlayStation 2...).

2.1 Exécution d'une instruction

Les RISC sont basés sur un modèle en pipeline pour exécuter les instructions. Cette structure permet d'exécuter chaque instruction en plusieurs cycles, mais de terminer l'exécution d'une instruction à chaque cycle. Cette structure en pipeline est illustrée sur la Figure 2.1. L'extraction (*Instruction Fetch - IF*) va récupérer en mémoire l'instruction à exécuter. Le décodage (*Instruction Decode - ID*) interprète l'instruction et résout les adresses des registres. L'exécution (*Execute- EX*) utilise l'unité arithmétique et logique pour exécuter l'opération. L'accès en mémoire (*Memory - MEM*) est utilisé pour transférer le contenu d'un registre vers la mémoire ou vice-versa. Enfin, l'écriture registre (*Write Back - WB*) met à jour la valeur de certains registres avec le résultat de l'opération. Ce pipeline permet d'obtenir les très hautes performances qui caractérisent le MIPS. En effet, comme les instructions sont de taille constante et que les étages d'exécution sont indépendants, il n'est pas nécessaire d'attendre qu'une instruction soit complètement exécutée pour démarrer l'exécution de la suivante. Par exemple, lorsqu'une instruction atteint l'étage ID une autre instruction peut être prise en charge par l'étage IF. Dans le cas idéal, 5 instructions sont constamment dans le pipeline. Bien entendu, certaines contraintes impliquent des ajustements comme par exemple lorsqu'une instruction dépend de la précédente. C'est un problème que nous n'aborderons pas mais qu'il est nécessaire de connaître pour interpréter l'exécution de certains programmes.

2.1.1 La mémoire

Le microprocesseur MIPS possède une mémoire de 4 Go (2^{32} bits) adressable par octets. C'est dans cette mémoire qu'on charge la suite des instructions du microprocesseur contenues dans un programme binaire exécutable (ces instructions sont des mots de 32 bits). Pour exécuter un tel programme, le microprocesseur vient chercher séquentiellement les instructions dans cette mémoire, en se repérant grâce à un compteur programme (*PC*) contenant l'adresse en mémoire de la prochaine instruction à exécuter. Les données nécessaires à l'exécution d'un programme y sont également placées (il n'y a pas de séparation en mémoire entre les instructions et les données). Il est à noter que toutes les instructions sont alignées sur 4 octets.

L'adresse d'un octet en mémoire correspond au rang qu'il occupe dans le tableau des 4 Go qui la constitue. Ces adresses sont codées sur 32 bits, et sont contenues dans l'intervalle 0x00000000 à 0xFFFFFFFF.

Pour stocker en mémoire des valeurs sur plusieurs octets, par exemple un mot sur 4 octets, deux systèmes existent (figure 2.2) :

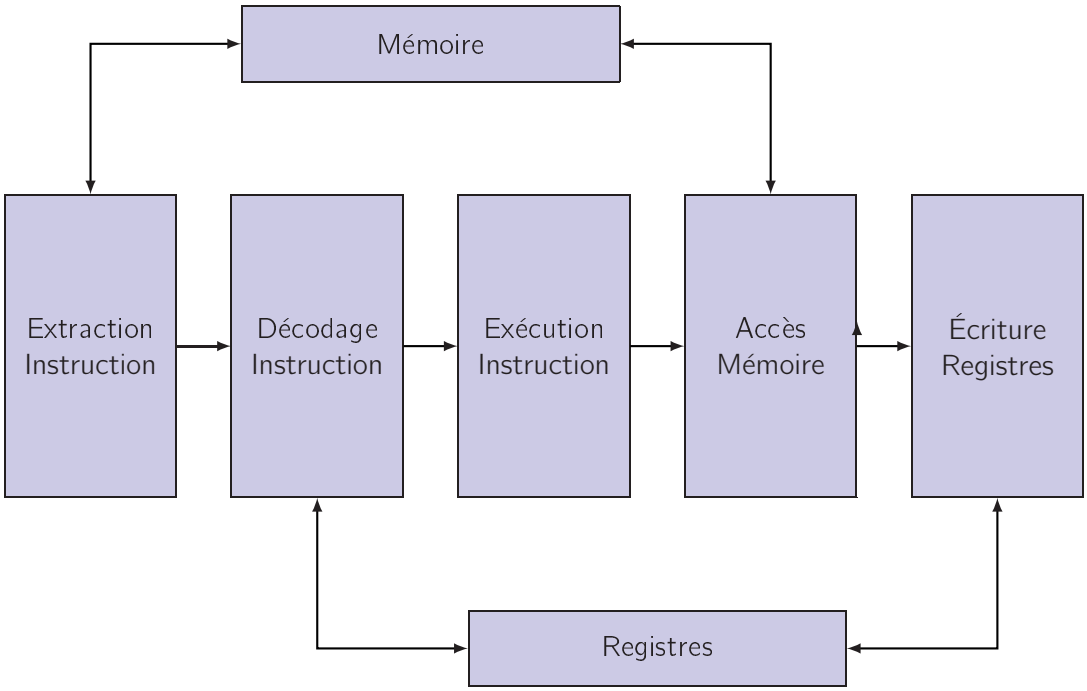


Figure 2.1 – Architecture interne des microprocesseurs RISC

- les systèmes de type *big endian* écrivent l'octet de poids le plus fort à l'adresse la plus basse. Les processeurs MIPS sont *big endian*, ainsi par exemple que les processeurs DEC ou SUN.
- les systèmes de type *little endian* écrivent l'octet de poids le plus faible à l'adresse la plus basse. Les processeurs PENTIUM notamment sont *little endian*.

Adresse :	*Contenu de la mémoire :*	
	big endian	little endian
	...	...
0x00000004	0xFF	0xCC
	0xEE	0xDD
	0xDD	0xEE
0x00000007	0xCC	0xFF
	...	...

Figure 2.2 – Mode d'écriture en mémoire de la valeur hexadécimale *0xFFEEDDCC* pour un système *big endian* ou *little endian*. Le MIPS est un processeur de type *big endian* : l'octet de poids fort se trouve à l'adresse la plus basse.

2.1.2 Les registres

Les registres sont des emplacements mémoire spécialisés utilisés par les instructions et se caractérisant principalement par un temps d'accès rapide.

Les registres d'usage général

La machine MIPS dispose de 32 registres d'usage général (General Purpose Registers, *GPR*) de 32 bits chacun, dénotés $0 à $31. Les registres peuvent également être identifiés par un mnémonique indiquant leur usage conventionnel. Par exemple, le registre $29 est noté `$sp`, car il est utilisé (par convention !) comme le pointeur de pile (sp pour *Stack Pointer*). Dans les programmes, un registre peut être désigné par son numéro aussi bien que son nom (par exemple, `$sp` équivaut à $29).

La figure 2.3 résume les conventions et restrictions d'usage que nous retiendrons pour ce projet.

Mnémonique	Registre	Usage
`$zero`	$0	Registre toujours nul, même après une écriture
`$at`	$1	*Assembler temporary* : registre réservé à l'assembleur
`$v0, $v1`	$2, $3	Valeurs retournées par une sous-routine
`$a0-$a3`	$4-$7	Arguments d'une sous-routine
`$t0-$t7`	$8-$15	Registres temporaires
`$s0-$s7`	$16-$23	Registres temporaires, préservés par les sous-routines
`$t8, $t9`	$24, $25	Deux temporaires de plus
`$k0, $k1`	$26, $27	kernel (réservés !)
`$gp`	$28	Global pointer (on évite d'y toucher !)
`$sp`	$29	*Stack pointer* : pointeur de pile
`$fp`	$30	Frame pointer (on évite d'y toucher !)
`$ra`	$31	*Return address* : utilisé par certaines instructions (`JAL`) pour sauver l'adresse de retour d'un saut

Figure 2.3 – Conventions d'usage des registres MIPS.

Les registres spécialisés

En plus des registres généraux, plusieurs autres registres spécialisés sont utilisés par le MIPS :

- Le compteur programme 32 bits `PC`, qui contient l'adresse mémoire de la prochaine instruction. Il est incrémenté après l'exécution de chaque instruction, sauf en cas de sauts et branchements.
- Deux registres 32 bits `HI` et `LO` utilisés pour stocker le résultat de la multiplication ou de la division de deux données de 32 bits. Leur utilisation est décrite section 2.3.2.

D'autres registres existent encore, mais qui ne seront pas utilisés dans ce projet (`EPC`, registres des valeurs à virgule flottante, ...).

2.1.3 Les Instructions

Bien entendu, comme tout microprocesseur qui se respecte, le MIPS possède une large gamme d'instructions (plus de 280). Toutes les instructions sont codées sur 32bits.

Dans ce projet nous ne prendrons en compte qu'un nombre restreint d'instructions simples. Les spécifications des instructions étudiées dans ce projet sont données dans l'annexe A. Elles sont directement issues de la documentation du MIPS fournie par le *Software User's Manual* de *Architecture For Programmers Volume II* de *MIPS Technologies* [3].

Nous donnons ici un exemple pour expliciter la spécification d'une instruction : l'opération addition (`ADD`), dont la spécification, telle que donnée dans le manuel, est reportée ci dessous Figure 2.4.

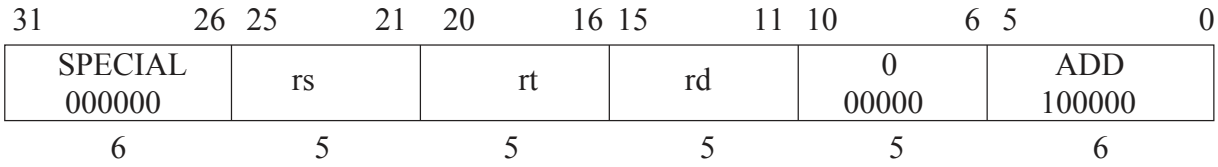


Figure 2.4 – Instruction ADD

Format: `ADD rd, rs, rt`

Purpose: `To add 32-bit integers. If an overflow occurs, then trap.`

Additionne deux nombres entiers sur 32-bits, si il y a un débordement, l'opération n'est pas effectuée.

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

```
The 32-bit word value in GPR rt is added to the 32-bit value in
GPR rs to produce a 32-bit result.
. If the addition results in 32-bit 2's complement arithmetic
  overflow, the destination register is not modified and an
  Integer Overflow exception occurs.
. If the addition does not overflow, the 32-bit result is placed
  into GPR rd.
```

rd, *rp* et *rt* désignent chacun l'un des 32 *General Purpose Registers GPR*. Comme il y a 32 registres, le codage d'un numéro de registre n'occupe que 5 bits.

Pour le reste, pas de commentaire : il s'agit juste un petit exercice pratique d'anglais .... Les descriptions données dans le manuel sont généralement très claires.

Restrictions: `None`

Operation:

```
temp <- (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 != temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] <- temp
endif
```

Restriction: `Integer Overflow`

Programming Notes:

`ADDU performs the same arithmetic operation but does not trap on overflow.`

Exemple de codage pour les instructions `ADD` et `ADDI` :

```
ADD $2, $3, $4                  00641020
ADDI $2, $3, 200                206200C8
```

À vous de retrouver ceci à partir de la doc ! Un bon petit exercice pour bien comprendre...

2.2 Le langage d'assemblage MIPS

Pour programmer un MIPS ont utilise un langage assembleur spécifiquement dédié au MIPS. La syntaxe qui est présentée ici est volontairement moins permissive que celle de l'assembleur *GNU*. On

se contente ici de présenter la syntaxe de manière intuitive.

Un programme se présente comme une liste d'unités, une unité tenant sur une seule ligne. Il est possible (et même recommandé pour aérer le texte) de rajouter des lignes blanches. Il y a trois sortes de lignes que nous allons décrire maintenant.

2.2.1 Les commentaires

C'est un texte optionnel non interprété par l'assembleur. Un commentaire commence sur une ligne par le caractère # et se termine par la fin de ligne.

Exemple

```
# Ceci est un commentaire. Il se termine à la fin de la ligne
ADD $2,$3,$4 # Ceci est aussi un commentaire, qui suit une instruction ADD
```

2.2.2 Les instructions machines

Elles ont la forme générale ci-dessous, les champs entre crochets indiquant des champs optionnels. Une ligne peut ne comporter qu'un champ étiquette, une opération peut ne pas avoir d'étiquette associée, ni d'opérande, ni de commentaire. Les champs doivent être séparés par des séparateurs qui sont des combinaisons d'espaces et/ou de tabulations.

```
[étiquette] [opération] [opérandes] [# commentaire]
```

Les sections suivantes présentent la syntaxe autorisée pour chacun des champs.

Le champ étiquette

C'est la désignation symbolique d'une adresse de la mémoire qui peut servir d'opérande à une instruction ou à une directive de l'assembleur. Une étiquette est une suite de caractères alphanumériques qui ne doit PAS commencer par un chiffre[1]. Cette chaîne est suivie par le caractère « : ». Le nom de l'étiquette est la chaîne de caractères alphanumériques située à gauche du caractère « : ». Plusieurs étiquettes peuvent être associées à la même opération ou à la même directive.

Une étiquette ne peut être définie qu'une seule fois dans une unité de compilation. Sa valeur lors de l'exécution est égale à son adresse d'implantation dans la mémoire après le chargement. Elle dépend donc de la section dans laquelle elle est définie et de sa position dans cette section (cf. section 2.2.3).

Exemple

```
etiq1:
_etiq2:
etiq3:  ADD $2,$3,$4  # les trois étiquettes repèrent la même instruction ADD
```

Le champ opération

Il indique soit un des mnémoniques d'instructions du processeur MIPS, soit une des directives de l'assembleur.

1. En réalité une étiquette peut contenir également les caractères : «.», «_», «$».

Exemple

```
ADD $2,$3,$4   # le champ opération à la valeur ADD
.space 32      # le champ opération à la valeur .space
```

Le champ opérandes

Le champ *opérandes* a la forme : `opérandes = [op1] [,op2] [,op3]`

Ce sont les opérandes éventuels si l'instruction ou la directive en demande. S'il y en a plusieurs, ces opérandes sont séparés par des virgules.

Exemple

```
ADD $2,$3,$4   # les opérandes sont $2, $3 et $4
.space 32      # l'opérande est 32
```

2.2.3 Les directives

Une directive commence toujours par un point (≪.≫). Il y a trois familles de directives : les directives de sectionnement du programme, les directives de définition de données et la directive d'alignement (nous n'aborderons pas cette dernière).

Directive	Description
`.data`	Ce qui suit doit aller dans le segment DATA
`.text`	Ce qui suit doit aller dans le segment TEXT
`.bss`	Ce qui suit doit aller dans le segment BSS
`.set option`	Instruction à l'assembleur pour inhiber ou non certaine options. Dans notre cas seule l'option *noreorder* est considérée
`.word w1, ..., wn`	Met les n valeurs sur 32 bits dans des mots successifs (ils doivent être alignés!)
`.byte b1, ..., bn`	Met les n valeurs sur 8 bits dans des octets successifs
`.space n`	Réserve *n* octets en mémoire. Les octets sont initialisés à zéro.

Directives de sectionnement

Bien que le processeur MIPS n'ait qu'une seule zone mémoire contenant à la fois les instructions et les données (ce qui n'est pas le cas de tous les microprocesseurs), deux directives existent en langage assembleur pour spécifier les sections de code et de données.

- la section `.text` contient le code du programme (instructions) .

- la section `.data` est utilisée pour définir les données du programme.
- la section `.bss` est utilisée pour définir les zones de données non initialisées du programme (qui réservent juste de l'espace mémoire). Ces données ne prennent ainsi pas de place dans le fichier binaire du programme. Elles seront effectivement allouées au moment du chargement du processus. Elles seront initialisées à zéro.

Les directives de sectionnement s'écrivent par leur nom de section : `.text`, `.data` ou `.bss`. Elles indiquent à l'assembleur d'assembler les lignes suivantes dans les sections correspondantes.

Les directives de définition de données

On distingue les données initialisées des données non initialisées.

Déclaration des données non initialisées Pouvoir réserver un espace sans connaître la valeur qui y sera stockée est une capacité importante de tout langage. Le langage assembleur MIPS fournit la directive suivante.

[étiquette] `.space` **taille** La directive `.space` permet de réserver un nombre d'octets égal à *taille* à l'adresse *étiquette*. Les octets sont initialisés à zéro.

```
toto: .space 13
```

La directive `.space` se trouve généralement dans une section de données `.bss`.

Déclaration de données initialisées L'assembleur permet de déclarer plusieurs types de données initialisées : des octets, des mots (32 bits), des chaînes de caractères, etc. Dans ce projet, on ne s'intéressera qu'aux directives de déclaration suivantes :

[étiquette] `.byte` **valeur** *valeur* peut être soit un entier signé sur 8 bits, soit une constante symbolique dont la valeur est comprise entre -128 et 127, soit une valeur hexadécimale dont la valeur est comprise entre 0x0 et 0xff. Par exemple, les lignes ci-dessous permettent de réserver deux octets avec les valeurs initiales -4 et 0xff sous forme hexadécimale. Le premier octet sera créé à une certaine adresse de la mémoire, que l'on pourra ensuite manipuler avec l'étiquette $Tabb$. Le second octet sera lui à l'adresse $Tabb + 1$.

```
Tabb:   .byte -4, 0xff
```

[étiquette] `.word` **valeur** *valeur* peut être soit un entier signé sur 32 bits, soit une constante symbolique dont la valeur est représentable sur 32 bits, soit une valeur hexadécimale dont la valeur est comprise entre 0x0 et 0xffffffff. Par exemple, la ligne suivante permet de réserver un mot de 32 bits avec la valeur initiale 32767 à une adresse de la mémoire, adresse que l'on manipulera ensuite avec l'étiquette Tabw.

```
Tabw:   .word 0x00007fff
```

2.2.4 Les modes d'adressage

Comme nous le verrons au chapitre 2.3, les instructions du microprocesseur MIPS ont de zéro à quatre opérandes. On appelle mode d'adressage d'un opérande la méthode qu'utilise le processeur pour déterminer où se trouve l'opérande, c'est-à-dire pour déterminer son **adresse**. Le langage assembleur MIPS contient 5 modes d'adressage décrit ci dessous.

Adressage registre direct

Dans ce mode, la valeur de l'opérande est contenue dans un registre et l'opérande est désigné par le nom du registre en question.

Exemple :

```
ADD $2, $3, $4   # les valeur des opérandes sont dans les registres 3 et 4
                 # le résultat est placé dans le registre 2
```

Adressage immédiat

La valeur de l'opérande est directement fournie dans l'instruction.

Exemple :

```
ADDI $2, $3, 200   # valeur immédiate entière signée sur 16 bits
ADDI $2, $3, 0x3f  # idem avec une valeur immédiate hexadécimale
ADDI $2, $3, X     # ajout $2 à la valeur (et non le contenu) de X (adresse mémoire)
```

Adressage indirect avec base et déplacement

Dans ce mode, interviennent un registre appelé *registre de base* qui contient une adresse mémoire, et une constante signée (décimale ou hexadécimale) codée sur deux octets appelée *déplacement*. La syntaxe associée par l'assembleur à ce mode est `offset(base)`.

Pour calculer l'adresse de l'opérande, le processeur ajoute au contenu du registre de base `base` la valeur sur 2 octets du déplacement `offset`.

Exemple :

```
LW $2, 200($3)     # $2 = memory[($3) + 200]
```

Adressage absolu aligné dans une région de 256Mo

Un opérande de 26 bits permet de calculer une adresse mémoire sur 32 bits. Ce mode d'adressage est réservé aux instructions de sauts (`J`, `JAL`).

Les 28 bits de poids faible de l'adresse de saut sont contenus dans l'opérande décalé de 2 bits vers la gauche (car les instructions sont alignées tous les 4 octets). Les poids forts manquants sont pris directement dans le compteur programme. Un exemple est donné au paragraphe 2.3.2.

Exemple :

```
J 10101101010100101010100011    # l'adresse de saut est calculée à partir
                                # de l'opérande et de la valeur de PC
```

Adressage relatif

Ce mode d'adressage est utilisé par les instructions de branchement. Lors du branchement, l'adresse de branchement est déterminée à partir d'un opérande `offset` sur 16 bits.

`offset` est compté en *nombre d'instructions*. Comme une instruction MIPS occupe 4 octets, pour obtenir le saut à réaliser en mémoire, l'`offset` est d'abord décalée de 2 bits vers la gauche puis ajoutée au compteur PC courant. Par exemple, un offset codé 0xFD dans l'instruction correspond en réalité à un offset de 0x3F4 ! La valeur sur 18 bits est ensuite ajoutée au compteur programme pour déterminer l'adresse de saut.

Exemple :

```
BEQ $2, $3, 0xFD # si $2==$3, branchement à l'adresse PC + 0x3F4
```

2.3 Instructions étudiées dans le projet

Cette section présente les instructions du MIPS qui devront être traitées dans le projet. Toutes les instructions MIPS ne seront pas traitées (en particulier les entrées-sorties, la gestion des valeurs flottantes...). La syntaxe des instructions en langage assembleur est donnée, ainsi qu'une description et le codage binaire des opérations.

2.3.1 Catégories d'instructions

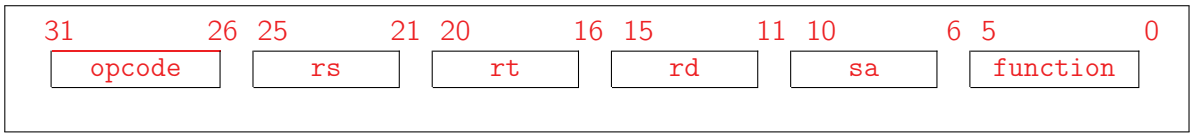
Les processeurs MIPS possèdent des instructions simples de taille constante égale à 32 bits[2]. Ceci facilite notamment les étapes d'extraction et de décodage des instructions, réalisées chacune dans le pipeline en un cycle d'horloge. Les instructions sont toujours codées sur des adresses alignées sur un mot, c'est-à-dire divisibles par 4. Cette restriction d'alignement favorise la vitesse de transfert des données.

Il existe seulement trois formats d'instructions MIPS, *R-type*, *I-type* et *J-type*, dont la syntaxe générale en langage assembleur est la suivante :

```
R-instruction $rd, $rs, $rt
I-instruction $rt, $rs, immediate
J-instruction target
```

Les instructions de type R

Le codage binaire des instructions *R-type*, pour "register type", suit le format :



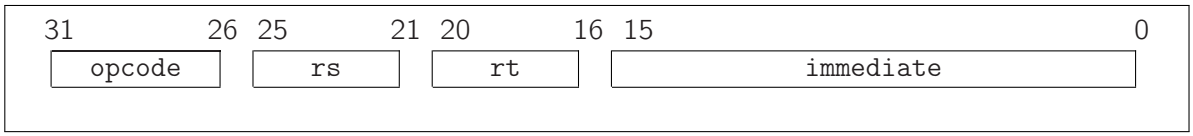
2. pour les séries R2000/R3000 auxquelles nous nous intéressons. Les processeurs récents sont sur 64 bits.

avec les champs suivants :

- le code binaire **opcode** (operation code) identifiant l'instruction. Sur 6 bits, il ne permet de coder que 64 instructions, ce qui même pour un processeur RISC est peu. Par conséquent, un champ additionnel **function** de 6 bits est utilisé pour identifier les instructions R-type.
- **rd** est le nom du registre destination (valeur sur 5 bits, donc comprise entre 0 et 31, codant le numéro du registre)
- **rs** est le nom du registre dans lequel est stocké le premier argument source.
- **rt** est le nom du registre dans lequel est stocké le second argument source ou destination selon les cas.
- **sa (shift amount)** est le nombre de bits de décalage, pour les instructions de décalage de bits.
- **function** 6 bits additionnels pour le code des instructions R-type, en plus du champ **opcode**.

Les instructions de type I

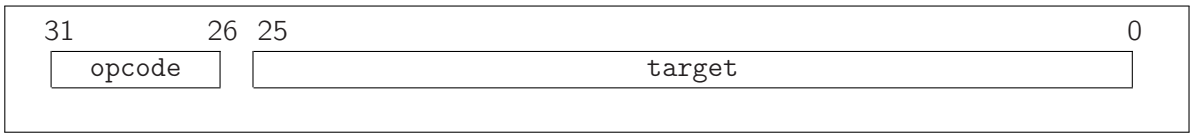
Le codage binaire des instructions *I-type*, pour "immediate type", suit le format :



avec **opcode** le code opération, **rt** le registre destination, **rs** le registre source et **immediate** une valeur numérique codée sur 16 bits.

Les instructions de type J

Le codage binaire des instructions *J-type*, pour "jump type", suit le format :



où **opcode** est le code opération et **target** une valeur de saut codée sur 26 bits.

2.3.2 Détails des instructions à prendre en compte dans le projet

Instructions arithmétiques

Mnemonic	Opérandes	Opération	Type
ADD	$rd, $rs, $rt	$rd = $rs + $rt	type R
ADDI	$rt, $rs, immediate	$rt = $rs + immediate	type I
SUB	$rd, $rs, $rt	$rd = $rs - $rt	type R
MULT	$rs, $rt	(HI, LO) = $rs * $rt	type R
DIV	$rs, $rt	LO = $rs div $rt; HI = $rs mod $rt	type R

`ADD` fait l'addition de deux registres `rs` et `rt`. Le résultat sur 32 bits de ces opérations est placé dans le registre `rd`.

`ADDI` est l'addition avec une valeur immédiate, `SUB` la soustraction. Le résultat sur 32 bits de ces opérations est stocké dans le registre `rd`. Les opérandes et le résultat sont des entiers signés sur 32 bits.

Pour la multiplication MULT, les valeurs contenues dans les deux registres rs et rt sont multipliées. La multiplication de deux valeurs 32 bits est un résultat sur 64 bits. 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. Les valeurs de ces registres sont accessibles à l'aide des instructions MFHI et MFLO définies ci dessous.

La division DIV fournit deux résultats : le quotient de rs divisé par rt est placé dans le registre LO, et le reste de la division entière dans le registre HI. Les valeurs de ces registres sont accessibles à l'aide des instructions MFHI et MFLO.

Les instructions logiques

Mnemonic	Opérandes	Opération	Type
AND	$rd, $rs, $rt	$rd = $rs AND $rt	Type R
OR	$rd, $rs, $rt	$rd = $rs OR $rt	Type R
XOR	$rd, $rs, $rt	$rd = $rs XOR $rt	Type R

Les deux registres de 32 bits rs et rt sont combinés bit à bit selon l'opération logique effectuée. Le résultat est placé dans le registre 32 bits rd.

Les instructions de décalage

Mnemonic	Opérandes	Opération	Type
ROTR	$rd, $rt, sa	$rd = $rt[sa-0] $rt[31-sa]	Type R
SLL	$rd, $rt, sa	$rd = $rt << sa	Type R
SRL	$rd, $rt, sa	$rd = $rt >> sa	Type R

Le contenu du registre 32 bits rt est décalé à gauche pour SLL et à droite pour SRL de sa bits (en insérant des zéros). sa est une valeur immédiate sur 5 bits, donc entre 0 et 31. Pour ROTR le mot contenu dans le registre 32 bits rt subi une rotation par la droite. Le résultat est placé dans le registre rd.

Les instructions Set

Mnemonic	Opérandes	Opération	Type
SLT	$rd, $rs, $rt	if $rs < $rt then $rd = 1, else $rd = 0	Type R

Le registre rd est mis à 1 si le contenu du registre rs est plus petit que celui du registre rt, à 0 sinon. Les valeurs rs et rt sont des entiers signés en complément à 2.

Les instructions Load/Store

Mnemonic	Opérandes	Opération	Type
LW	$rt,offset($rs)	$rt = memory[$rs+offset]	Type I
SW	$rt, offset($rs)	memory[$rs+offset] = $rt	Type I
LUI	$rt,immediate	$rt = immediate << 16	Type I
MFHI	$rd	$rd = HI	Type R
MFLO	$rd	$rd = LO	Type R

– Load Word (LW) place le contenu du mot de 32 bits à l'adresse mémoire ($rs + offset) dans le registre rt. offset est une valeur signée sur 16 bits codée en complément à 2, elle est placée dans le champ immediate.

Exemple : LW $8, 0x60($10)

- Store Word (SW) place le contenu du registre rt dans le mot de 32 bits à l'adresse mémoire ($rs + offset). offset est une valeur signée sur 16 bits codée en complément à 2, elle est placée dans le champ immediate.
- Load Upper Immediate (LUI) place le contenu de la valeur entière 16 bits immediate dans les deux octets de poids fort du registre $rt et met les deux octets de poids faible à zéro.
- L'instruction Move from HI (MFHI) : Le contenu du registre HI est placé dans le registre rd. HI contient les 32 bits de poids fort du résultat 64 bits d'une instruction MULT ou le reste de la division entière d'une instruction DIV.
- L'instruction Move from LO (MFLO) est similaire à MFHI : le contenu du registre LO est placé dans le registre rd. LO contient les 32 bits de poids faible du résultat 64 bits d'une instruction MULT ou le quotient de la division entière d'une instruction DIV.

Les instructions de branchement et de saut

Mnemonic	Opérandes	Opération	Type
BEQ	$rs, $rt, offset	Si ($rs = $rt) alors branchement	Type I
BNE	$rs, $rt, offset	Si ($rs != $rt) alors branchement	Type I
BGTZ	$rs, offset	Si ($rs > 0) alors branchement	Type I
BLEZ	$rs, offset	Si ($rs <= 0) alors branchement	Type I
J	target	PC=PC[31:28] target	Type J
JAL	target	GPR[31]=PC+8, PC=PC[31:28] target	Type J
JR	$rs	PC=$rs.	Type R

- BEQ effectue un branchement après l'instruction si les contenus des registres rs et rt sont égaux. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
- BNE effectue un branchement après l'instruction si les contenus des registres rs et rt sont différents. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
- BGTZ effectue un branchement après l'instruction si le contenu du registre rs est strictement positif. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
- BLEZ effectue un branchement après l'instruction si le contenu du registre rs est négatif ou nul. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
- J effectue un branchement aligné à 256 Mo dans la région mémoire du PC. Les *28 bits* de poids faible de l'adresse du saut correspondent au champ target, décalés de 2. Les 4 bits de poids fort restant correspondent au 4 bits de poids fort du compteur PC. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.

  Exemple : Soit l'instruction J 10101101010100101010100011, localisée à l'adresse 0x56767296.
  Quelle est l'adresse du saut ?
  L'offset est :

  ```
  0x26767296 --  10101101010100101010100011
  ```

  Comme toutes les instructions sont alignées sur des adresses multiples de 4, les deux bits

de poids faible d'une instruction sont toujours 00. On peut donc décaler le champs `offset` de 2 bits, ce qui donne une adresse de saut sur 28 bits :

```
adresse 28 bits:  10101101010100101010100011 00
```

Les quatre bits de poids fort de l'adresse de saut sont ensuite fixés comme les 4 bits de poids fort de l'adresse de l'instruction de saut, c'est-à-dire du compteur PC.

```
adresse de l'instruction
PC:  0x56767296 == 01010110011101100111001010010110
```

L'adresse finale de saut est donc :

```
0101 + 10101101010100101010100011 00 = 01011010110101010010101010001100
```

- `JAL` effectue un appel à une routine dans la région alignée de 256 Mo. Avant le saut, l'adresse de retour est placée dans le registre `$ra` (= `$31`). Il s'agit de l'adresse de l'instruction qui suit immédiatement le saut et où l'exécution reprendra après le traitement de la routine. Cette instruction effectue un branchement aligné à 256 Mo dans la région mémoire du PC. Les *28 bits* de poids faible de l'adresse du saut correspondent au champ `offset`, décalés de 2. Les poids forts restant correspondent au bits de poids fort de l'instruction.
- `JR` effectue un saut à l'adresse spécifiée dans `rs` Le contenu du registre 32 bits `rs` contient l'adresse du saut.