Projet d'Informatique

PHELMA 2e année

Année universitaire 2013–2014

Simulateur MIPS

Francois.Portet@imag.fr
Nicolas.Castagne@imag.fr

D'après
Matthieu Chabanas
Laurent Fesquet

Résumé

Au cours de votre expérience en informatique, il vous est sûrement arrivé d'utiliser un débogueur pour trouver une erreur bien cachée au fond de votre programme. Cet outil bien pratique permet de *simuler* l'exécution d'un programme et d'explorer ou modifier dynamiquement sa mémoire. C'est un outil tellement indispensable pour le développement et l'analyse de logiciels, qu'on en trouve pour tous les langages informatiques.

Ce projet a pour but de réaliser un débogueur de programmes écrits en langage assembleur MIPS. Ce programme, que l'on appelle *Simulateur*, sera écrit en langage C. Durant sa réalisation vous vous familiariserez avec le microprocesseur MIPS et le langage assembleur et vous aborderez les notions d'interpréteur, d'analyse lexicale, de gestion des erreurs, de fichiers objets et bien d'autres qui vous permettront d'enrichir considérablement votre culture générale et votre savoir faire en informatique.

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet informatique est de réaliser sous Linux, en langage C, un simulateur d'une machine MIPS 32bits permettant d'exécuter et de mettre au point des programmes écrits en langage du microprocesseur de la machine MIPS. Le rôle d'un simulateur est de lire un programme donné en entrée et d'exécuter chacune des instructions avec le comportement de la machine cible (ici MIPS 32 bits). Les simulateurs permettent notamment de prototyper et déboguer des programmes sans la contrainte de posséder le matériel cible (ordinateur avec un microprocesseur MIPS 32 bits).

Plus précisément, le simulateur que vous devrez réaliser prendra en entrée un fichier objet au format **ELF** et permettra :

— son *exécution* complète,

— son exécution *pas à pas* à travers un *interpréteur* de commandes,

— son *débogage* (p.ex. : affichage de la mémoire interne, détection d'erreurs, rapport d'erreur...),

Par ailleurs, à la fin du projet, votre simulateur devra pouvoir gérer les fichiers dit *relogeables* qui peuvent être chargés à n'importe quel endroit de la mémoire.

Le simulateur devra être réalisé en langage C et devra fonctionner dans l'environnement Linux de l'école. Pour ce projet nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions du MIPS. Ces instructions du microprocesseur MIPS seront simulées/réalisées par des appels de fonctions du langage C, elles-même compilées en une série d'instructions pour le microprocesseur effectuant la simulation (celui de la machine sur lequel est effectué le travail).

Le chapitre 2 donne une présentation générale du microprocesseur et introduit le langage assembleur considéré et le sous-ensemble des instructions du MIPS à gérer dans le projet. Le chapitre 3 détaille l'organisation et les fonctionnalités attendues du simulateur tandis que le chapitre 4 introduit le format des fichiers d'entrée et le principe de relocation. Les chapitres 5 et 6 présentent quelques considérations importantes pour la mise en œuvre, puis des informations sur l'organisation générale du projet.

L'intérêt pédagogique de ce projet est multiple : il permet tout d'abord de travailler sur un projet de taille importante sous tout ses aspects techniques (analyse d'un problème, conception puis implémentation d'une solution, validation du résultat) mais aborde aussi les notions de gestion de projet et de respect d'un planning. Ce projet vous permettra également d'acquérir la maîtrise du langage C qui est particulièrement utilisé pour la programmation scientifique et le développement industriel, ainsi que des outils de développement associés (systèmes Unix/Linux, outil Make, déboggueur). Enfin, il illustre et met en pratique les connaissances relatives aux microprocesseurs (cf. cours d'architecture ou cours d'ordinateurs et microprocesseurs).

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) :

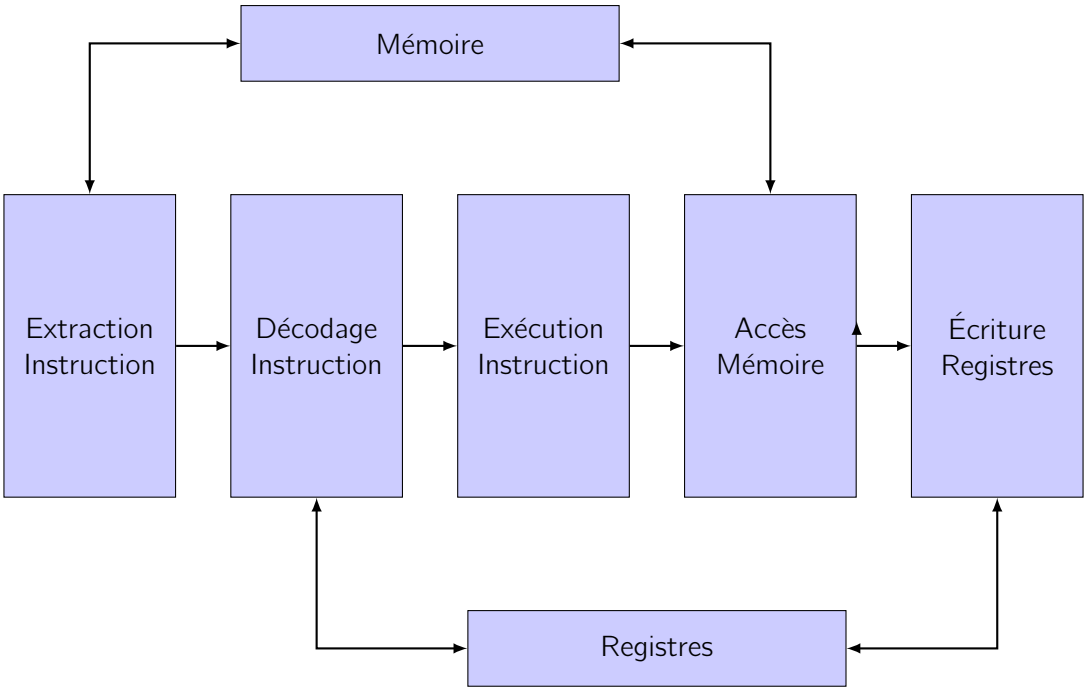


Figure 2.1 – Architecture interne des microprocesseurs RISC

- 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.

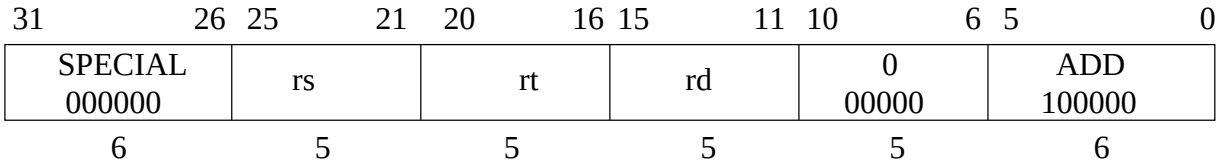


Figure 2.4 – Instruction ADD

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

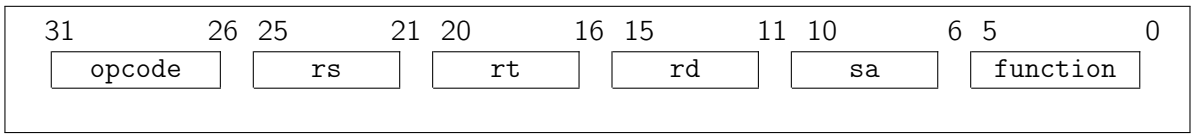
Les processeurs MIPS possèdent des instructions simples de taille constante égale à 32 bits [2]. Ceci facilite notamment les étapes d'extraction et de décodage des instructions, réalisées chacune dans le pipeline en un cycle d'horloge. Les instructions sont toujours codées sur des adresses alignées sur 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 :



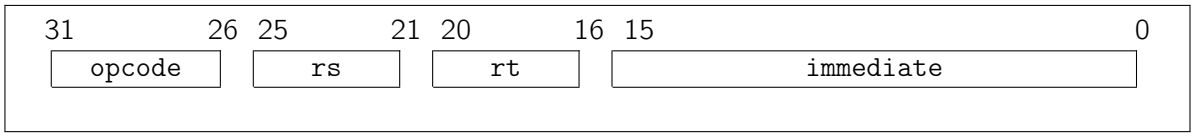
2. pour les séries R2000/R3000 auxquelles nous nous intéressons. Les processeurs récents sont sur 64 bits.

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

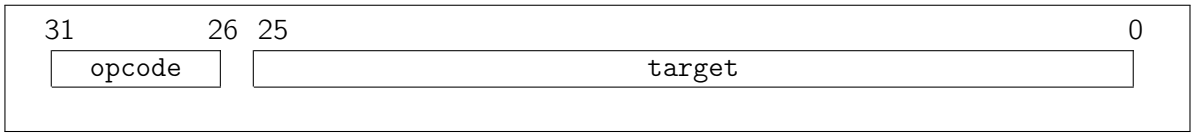
Le codage binaire des instructions *I-type*, pour "immediate type", suit le format :



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.

Chapitre 3

Le simulateur

Comme le nom l'indique, un simulateur (on parle aussi d'émulateur) est un logiciel capable de reproduire le comportement de l'objet simulé, dans le cas qui nous intéresse la machine MIPS lorsqu'elle exécute un programme. *Reproduire le comportement* signifie plus précisément que l'on va définir pour notre simulateur une mémoire et des registres de taille identique à celle de la machine MIPS [3], puis on doit réaliser l'évolution de l'état de cette mémoire et de ces registres selon les instructions du programme. On doit obtenir les mêmes résultats que ceux que l'on obtiendrait avec une exécution sur une machine MIPS réelle mais pas forcément de la même façon : c'est le principe du *faire semblant* (par opposition au *faire comme*).

La Figure 3.1 présente le diagramme des différents composants d'un simulateur et de leur interaction. Le programme assembleur, qui contient non seulement les instructions mais aussi les données, est chargé dans les segments de mémoire du microprocesseur. Un module de chargement lit les fichiers elf et stocke leur contenu dans la mémoire. Les registres contiennent les données représentant les arguments des instructions arithmétiques et logiques. On peut noter la présence du **PC** (*Program Counter* ou *Instruction pointer*) qui est l'indice donnant l'adresse de la prochaine instruction à exécuter. Ce PC est mis à jour par le module de décodage/exécution qui doit interpréter les instructions pour connaître la prochaine à exécuter. Par exemple, une simple opération d'addition (`ADD $9,$10,$11`) impliquera un simple incrément du PC (la prochaine instruction est celle suivant l'addition) mais un branchement tel que `BNE $9,$10, ailleurs` demandera l'exécution complète de l'instruction pour savoir si oui ou non `$9` et `$10` sont égaux avant de savoir s'il faut sauter à l'adresse `ailleurs` ou exécuter l'instruction suivante.

Pour débugger un programme, il faut pouvoir l'arrêter au milieu d'une exécution et pouvoir analyser son état (valeur de registre, valeur du PC, etc.). Pour cela, on utilise des points d'arrêt qui permettent de stopper une exécution à une adresse précise ou une exécution pas-à-pas. L'ajout de point d'arrêt et le mode d'exécution est décidé par l'utilisateur, à travers l'interpréteur de commande. Cet interpréteur est l'interface avec l'utilisateur qui lui permet de contrôler le simulateur.

3.1 Chargement des programmes en mémoire

Pour être *exécuté* par le simulateur, un fichier programme doit d'abord être recopié (on dit *chargé*) dans la mémoire simulant celle de la machine MIPS. Le format des fichiers binaires considérés pour la machine MIPS fait l'objet du chapitre 4. Il faut donc réaliser une interface utilisateur permettant d'effectuer ce chargement, c'est-à-dire la recopie du fichier à exécuter dans la mémoire. En d'autres termes, si l'utilisateur veut exécuter son programme, il faudra qu'il effectue les actions suivantes : lancer le simulateur de la machine, puis taper la commande Charger en fournissant le nom du fichier, et enfin taper la commande Exécuter. Cette partie interface utilisateur du simulateur constitue une partie du programme C à élaborer (le *module d'entrée*, qui offre à l'utilisateur un interpréteur de commande).

3.2 Exécution du programme

La simulation proprement dite d'une instruction du programme, c'est-à-dire l'interprétation du ou des octets consécutifs représentant le code instruction éventuellement suivi d'extensions et d'opérandes,

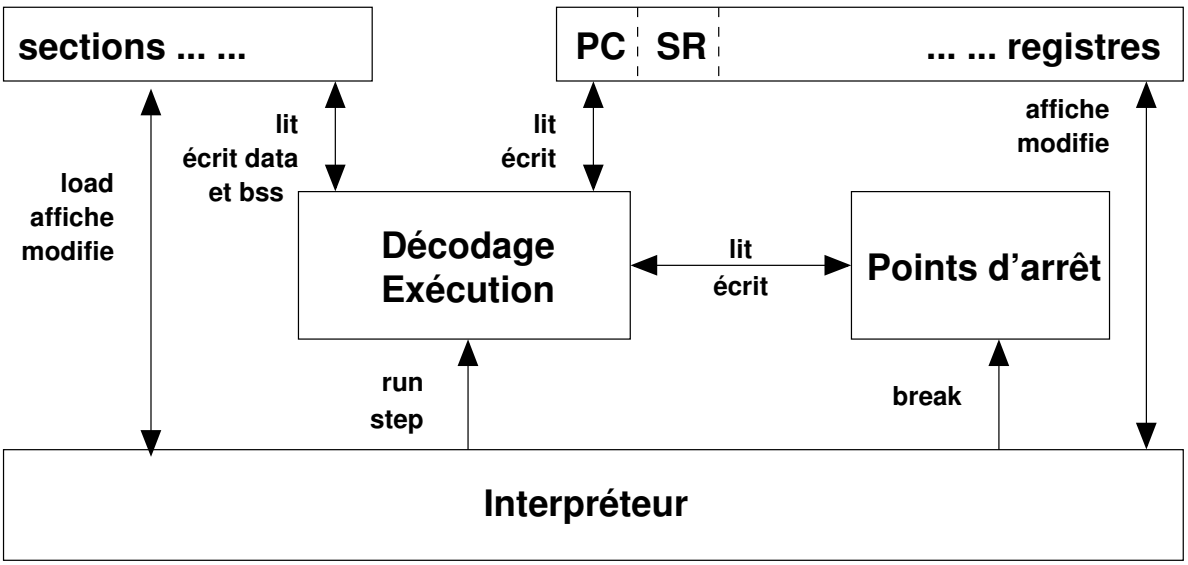


Figure 3.1 – Architecture d'un simulateur MIPS

consiste à décoder l'instruction et à réaliser à l'intérieur de la mémoire et des registres les modifications qu'elle spécifie. Ces étapes de *décodage* puis *d'exécution* constituent la seconde grande partie du programme C à réaliser.

L'exécution du programme complet consiste évidemment à exécuter une à une l'ensemble des instructions codées dans l'ordre spécifié par le programme. Par défaut, cet ordre est séquentiel : on exécute les instructions dans l'ordre où elles arrivent dans le programme. Toutefois, certaines instructions dites de branchements (`BEQ`, `J`, `JAL`, ...) induisent des ruptures de séquences à des endroits variés du programme. Cette tache de contrôle du déroulement du programme pourra faire l'objet d'un module du programme C à réaliser.

3.3 Interface utilisateur

Pour que l'utilisateur puisse mettre au point des programmes à l'aide du simulateur, il est nécessaire qu'il garde le contrôle de la simulation. C'est pourquoi l'interface utilisateur introduite précédemment devra comporter une série de commandes permettant ce contrôle.

L'utilisateur pourra par exemple avoir le choix de taper la commande *Exécuter* en lui adjoignant l'adresse de début du programme, ou encore d'utiliser une commande *Modifier registre* pour affecter au PC l'adresse de début du programme et de taper ensuite la commande *Exécuter*. Si le programme comporte une erreur, par exemple un code d'opération invalide ou une opérande invalide, l'utilisateur doit reprendre la main afin de modifier son programme. Pour cela il doit disposer de commandes *Afficher mémoire* et *Afficher registres* qui lui permettent de regarder le contenu de la mémoire et des différents registres, ainsi que *Modifier mémoire* et *Modifier registre* qui lui permettent de modifier les contenus respectifs de la mémoire et des registres. De plus, le simulateur devra fournir une instruction *Pas* qui permet d'exécuter une instruction du programme à la fois, et enfin des instructions *Point d'arrêt* et *Supprimer Point d'arrêt* qui permettent respectivement de positionner et d'enlever des points d'arrêt dans le programme.

Enfin, maintenant qu'on dispose d'un petit langage de commandes pour contrôler le simulateur,

plutôt que de taper ces commandes "à la main", il devient possible... d'écrire des programmes qui utilisent ce langage. Il suffit pour cela que votre programme puise travailler non seulement sur l'entrée standard (i.e. le clavier, avec `scanf(...)` ou `fscanf(stdin, ...)`, etc.) mais aussi sur un fichier texte (avec `fscanf(file, ...)`, etc) qui contiendra la liste des commandes.

On adoptera l'extension ".simcmd" (SIMulateur CoMmanDe) pour les fichiers de commandes. Ainsi, votre programme pourra être lancé de deux manières [1]

— soit sans paramètres, auquel cas il attend des commandes tapées au clavier

— soit avec le nom d'un fichier en paramètre, e.g. dans le Terminal
`simMips monFichierDeCommande.simcmd`, auquel cas il exécutera les commandes présentes dans le fichier `monFichierDeCommande.simcmd`.

Notez que le code qui vous est donné pour démarrer votre projet, décrit dans la suite de ce document, dispose déjà de ce mécanisme. À vous de le comprendre et de l'adapter !

3.4 Fin d'une simulation

Normalement, si le programme ne comporte pas d'erreur ni de boucle infinie, on devrait s'arrêter sur la rencontre d'un code opération non exécutable. En effet, après avoir exécuté la dernière instruction définie dans le programme chargé, le compteur programme va désigner un mot mémoire non initialisé. Un tel mot a peu de chance de correspondre à un code opération valide. Mais même si elle est faible, cette probabilité n'est pas nulle. Il est donc nécessaire que le programme se termine par un code opération invalide non quelconque afin d'être certain de stopper le programme à cet endroit et de distinguer cet arrêt voulu d'une erreur. Ce code invalide sera placé à la fin du programme après le chargement de son contenu en mémoire. Une autre façon de faire est d'insérer un point d'arrêt après la dernière instruction du programme.

3.5 Simulation des exceptions arithmétiques

Comme pour tous microprocesseur, le MIPS peut lever des exceptions lors de calculs arithmétiques. En informatique, une exception et un événement qui sort du déroulement ordinaire d'un programme (c.-à-d., un cas exceptionnel) et qui peut provoquer l'arrêt ou la modification du programme. Beaucoup de types d'exceptions existent que ce soit pour l'arithmétique (p.ex., division par zéro), la mémoire (p.ex., zone non allouée), le système d'exploitation (p.ex., signaux d'arrêt d'un processus), etc. Dans notre cas nous nous contenterons de signaler les exceptions arithmétiques à travers un registre d'état `SR` (pour *Control and Status Register*) de 32 bits qui contient des drapeaux mis à jours après l'exécution de certaines instructions. La figure 3.2 décrit les indicateurs que nous utiliserons dans le projet.

31.....13	12	11				7	6						0
	D	**O**				**S**	**Z**						**C**

Figure 3.2 – Indicateurs utiles du registre `SR`

- **C** : Carry (bit 0 du registre `SR`). Il contient la retenue après une addition ou une soustraction et permet ainsi d'effectuer des calculs avec des nombres qui dépassent la capacité de représentation

1. Notez qu'un tel mécanisme est tout à fait courant dans les langage interprétés. Par exemple, il est adopté par le Shell Linux : pour exécuter un fichier de commandes shell, on tape `sh monNomDeFichier.sh` dans le Terminal. Incidemment, le "script de test" qui vous est fourni est un tel script Shell.

d'un mot. Par exemple, si on additione 0xFFFFFFFF à 0xFFFFFFFF le résultat devrait être 0x1FFFFFFFE qui nécessite 33bits. Le bit de poids fort est stocké dans le **C**arry. Le bit **C** est également utilisé dans les opérations de décalages.
- **Z** : Zéro (bit 6 du registre `SR`). Il contient 1 si le résultat d'une opération arithmétique ou logique est nul et 0 sinon.
- **S** : Signe (bit 7 du registre `SR`). Il prend la valeur du bit de signe du résultat après l'exécution d'une opération arithmétique ou logique.
- **O** : Overflow (bit 11 du registre `SR`). Il indique un débordement après une opération dont les opérandes sont considérées comme des valeurs signées. Pour des opérations sur des valeurs non signées il est non modifié.
- **D** : Division par zéro (bit 12 du registre `SR`). Après l'exécution d'une instruction `DIV`, il contient 1 si la valeur de la deuxième opérande était nulle, 0 sinon.

3.6 Description des commandes du simulateur

Dans cette section, on donne la liste et les spécifications des commandes de l'interface utilisateur du simulateur.

L'interface utilisateur à réaliser est un interpréteur de commande (comme le Shell Linux, si ce n'est que ce ne sont pas de commandes Shell qui sont interprétées). Dans une boucle infinie :
- Le programme se met en attente d'une commande ;
- L'utilisateur tape une commande et termine par "Entrée" $\hookleftarrow$;
- Le programme décode la commande et l'exécute, puis se met en attente de la commande suivante.

Toute les commandes écrivant un résultat utiliseront la sortie standard du programme. À l'inverse, vous prendrez garde à ce que toutes les message de débuggage qu'écrit votre programme soient envoyées sur le flux l'erreur standard du programme. Cela est nécessaire pour pouvoir automatiser les tests de votre programme.

On s'attachera à ce que le programme contrôle le bon format des paramètres des commandes. En particulier, et à titre d'exemple, les adresses mémoires passées en paramètres doivent être des entiers compris entre 0 et la valeur maximale du programme. Ainsi, pour toutes les commandes faisant intervenir un accès en mémoire, tout dépassement doit être signalé par un message d'erreur et la main doit être rendue à l'utilisateur. De même, on contrôlera le bon format des paramètres relatifs aux registres.

Concernant la syntaxe des paramètres, il est à noter les conventions suivantes :
- Les crochets [ ] indiquent que l'argument (ou le groupe d'arguments) à l'intérieur est optionnel.
- L'étoile $*$ collée à un argument indique que cet argument peut être répété un nombre quelconque de fois.
- Le caractère $\|$ séparant des arguments indique que l'un des arguments au choix peut être l'argument de la commande.
- Un argument entre les caractères $<$ et $>$ indique que la description de cet argument est donnée dans les lignes suivantes.

3.6.1 Commandes relatives à la gestion de l'environnement du simulateur

Charger un programme

- Nom de la commande : lp (load program)
- Syntaxe : `lp <nom_du_fichier>`

- Paramètres :

  `nom_du_fichier` : chemin du fichier

- Description :
  Le fichier dont le nom est passé en paramètre doit être un fichier ELF relogeable sans symbole externe. Les fichiers ELF et de la notion de relocation sont décrits en détail à la section 4.
  Un fichier ELF est constitué de trois zones : la section `.text` (la zone contenant les instructions), la section `.data` (la zone contenant les données initialisées) et la section `.bss` (la zone contenant les données non initialisées). Si la valeur du champ correspondant à l'une de ces sections dans le descripteur est nulle, la section ne figure pas dans le fichier. Le chargeur doit prendre chacune des sections présentes dans le fichier et les copier dans la mémoire en respectant les règles suivantes :
  - Les octets constituant la section `.text` seront copiés les uns à la suite des autres à partir de l'adresse 0x0 de la mémoire de la machine simulée.
  - Les octets constituant la section `.data` seront copiés les uns à la suite des autres à partir de la première adresse de la mémoire de la machine simulée suivant l'adresse du dernier octet copié de la section `.text` qui soit multiple de 4096.
  - Les octets constituant la section `.bss` seront réservés les uns à la suite des autres à partir de la première adresse de la mémoire de la machine simulée suivant l'adresse du dernier octet copié de la section `.data` qui soit multiple de 4096.

  Si le fichier ELF contient des sections de relocations `.rel.text` et/ou `.rel.data`, la relocation sera gérée automatiquement à la fin du chargement en mettant à jour les adresses relatives dans le codage des instructions et données.

Quitter le programme

- Nom de la commande : ex (exit)
- Syntaxe : `ex`
- Description : Cette commande provoque la fin du programme simulateur.

Afficher la mémoire

- Nom de la commande : dm (display memory)
- Syntaxe : `dm Adresse`
  avec

```
          Adresse  ==  <adresse> | <adr_nb_octets> | <adr_seg>
    adr_nb_octets  ==  <adresse>":"<nb_octets>
          adr_seg  ==  <adresse>"~"<adresse>
```

- Paramètres :

  `adresse` : valeur hexadécimale
  `nb_octets` : nombre entier d'octets à afficher

- Description : Cette commande affiche sur la console le contenu de la mémoire dont les adresses sont spécifiées en paramètres. L'affichage se fera à raison de 16 octets par ligne séparés par des espaces (un octet sera affiché par deux chiffres hexadécimaux). Chaque ligne commencera par l'adresse hexadécimale de l'octet le plus à gauche de la ligne.
  La combinaison `<adresse>:<nb_octet>` signifie que l'on doit afficher `nb_octets` à partir de l'adresse `adresse`. La combinaison `<adresse1>~<adresse2>` signifie que l'on doit afficher le contenu de la mémoire entre les deux adresses `<adresse1>` et `<adresse2>` comprises. Lorsque

`dm` est appelé avec une seul argument `adresse` on affiche le mot (ici 32 bits) contenu à cette adresse.

- Exemple : affichage de 300 octets de la section TEXT et affichage de 12288 octets à partir de l'adresse 4. Notez que la commande n'affiche que ce qui existe.

```
>> dm 0x0000 : 300
Affichage de la section .text (8 octets)
00000000 21 08 00 11 21 08 00 12

>> dm 0x00004 ~ 0x3000
Affichage de la section .text (8 octets)
00000004 21 08 00 12
Affichage de la section .data (4 octets)
00001000 00 00 0a bc
Affichage de la section .bss (18 octets)
00002000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00002010 00 00

>> dm 0x56000 : 300

>> dm 0x1000:-67
[WARNING] Invalid param : a positive number is expected
```

Afficher le code assembleur

- Nom de la commande : da (display assembler)
- Syntaxe : `da <adresse> ":" <nb_instr>`
- Paramètres :

  `adresse` : valeur hexadécimale
  `nb_instr` : nombre entier d'instructions à afficher

- Description : Cette commande affiche sur la console, en hexadécimal sur 8 chiffres, le contenu de la mémoire qui a été désassemblée. La commande affichera une instruction par ligne avec comme premier élément l'adresse de l'instruction `adresse` en hexadécimal puis le code binaire de l'instruction en hexadécimal puis le code assembleur en chaîne de caractère.
- Exemple : désassemblage de 2 instructions à partir de l'adresse `A0020000`

```
>> da 0xa0020000:2
a0020000 206200C8 ADDI v0, v1, 200
a0020004 00641020 ADD  v0, v1, a0
```

Afficher les registres

- Nom de la commande : dr (display register)
- Syntaxe : `dr [<nom_reg>]*`
- Paramètres :

  `nom_reg` : nom(s) du (des) registre(s) à afficher, par exemple `$1` ou `$at`. En l'absence de paramètre, on affiche tous les registres.

- Description : Cette commande affiche sur la console de visualisation le contenu des registres. Chaque registre sera affiché sous la forme nom : valeur, valeur étant définie par 8 chiffres hexadécimaux. Les registres seront affichés à raison de 4 par ligne.
- Exemple : affichage de 5 registres et affichage de tous les registres

```
>> dr $at $zero $k1 $t8 $v1
   at :  00000000 zero :  00000000   k1 :  00000000   t8 :  00000000
   v1 :  00000000

>> dr
 zero :  00000000   at :  00000000   v0 :  00000000   v1 :  00000000
   a0 :  00000000   a1 :  00000000   a2 :  00000000   a3 :  00000000
   t0 :  00000000   t1 :  00000000   t2 :  00000000   t3 :  00000000
   t4 :  00000000   t5 :  00000000   t6 :  00000000   t7 :  00000000
   s0 :  00000000   s1 :  00000000   s2 :  00000000   s3 :  00000000
   s4 :  00000000   s5 :  00000000   s6 :  00000000   s7 :  00000000
   t8 :  00000000   t9 :  00000000   k0 :  00000000   k1 :  00000000
   gp :  00000000   sp :  00000000   fp :  00000000   ra :  00000000
   HI :  00000000   LO :  00000000   SR :  00000000   PC :  00000000
```

Modifier une valeur en mémoire

- Nom de la commande : lm (load memory)
- Syntaxe : `lm <adresse> <valeur>`
- Paramètres :

  `adresse` : valeur hexadécimale
  `valeur` : une valeur entière exprimable sur 32 bits (1 à 8 chiffres hexadécimaux)
- Description : Cette commande écrit la quantité `valeur` dans la mémoire à l'adresse `adresse` et affiche le mot modifié.
- Exemple : modification d'une donnée dans BSS.

```
>> lm 0x2000  0xFF00FF00
Affichage de la section .bss (18 octets)
00002000 ff 00 ff 00
```

Modifier une valeur dans un registre

- Nom de la commande : lr (load register)
- Syntaxe : `lr <nom_registre> <valeur>`
- Paramètres :

  `nom_registre` : un nom d'un des registres 32 bits valides
  `valeur` : une valeur entière exprimable sur 32 bits (1 à 8 chiffres hexadécimaux)
- Description : De façon analogue à la commande précédente, cette commande écrit la valeur `valeur` donnée en paramètre dans le registre `nom_registre` sauf si celui-ci est le registre 0. La commande affiche la valeur du registre après affectation.
- Exemple : on set le registre `$v1` à -71 et on essaye de mettre le registre `$zero` a une valeur non nulle.

```
>> lr $3 -71
   v1 :  fffffb9
```

```
>> lr $zero 0x1000
 zero :  00000000
```

3.6.2 Commandes relatives à l'exécution du programme

Exécuter à partir d'une adresse

- Nom de la commande : run
- Syntaxe : `run [<adresse>]`
- Paramètres :

  `adresse` : valeur hexadécimale

- Description : Cette commande charge PC avec l'adresse fournie en paramètre et lance le microprocesseur. Si le paramètre est omis, on se contente de lancer le microprocesseur, qui commencera son exécution à la valeur courante de PC.

Exécution pas à pas (ligne à ligne)

- Nom de la commande : s (step)
- Syntaxe : `s`
- Paramètres : néant
- Description : Cette commande provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre PC puis rend la main à l'utilisateur. Si l'on rencontre un appel à une procédure, cette dernière s'exécute complètement jusqu'à l'instruction de retour incluse (par exemple `J $ra`). La main est alors rendu à l'utilisateur sur l'instruction suivant l'appel.

Exécution pas à pas (exactement)

- Nom de la commande : si (step into)
- Syntaxe : `si`
- Paramètres : néant
- Description : Cette commande provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre PC puis rend la main à l'utilisateur. Si l'on rencontre un appel à une procédure, cette dernière s'exécute alors pas à pas.

Mettre un point d'arrêt sur une adresse

- Nom de la commande : bp (break point)
- Syntaxe : `bp <adresse>`
- Paramètre :

  `adresse` : valeur hexadécimale

- Description : Cette commande met un point d'arrêt à l'adresse fournie en paramètre. Lorsque le registre PC sera égal à cette valeur, l'exécution sera interrompue avant l'exécution de l'instruction et l'utilisateur reprendra la main.
- Exemple : on ajoute un point d'arrêt à l'adresse 12 et on ajoute un point d'arrêt en dehors de la zone mémoire.

```
>> bp 12
   bp at 0X0000000c added

>> bp 0X800000
 [WARNING] 00800000 is not a valid address for a breakpoint in the TEXT section.
```

Supprimer un point d'arrêt

- Nom de la commande : er (erase)
- Syntaxe : `er <adresse>`
- Paramètres :

  `adresse` : valeur hexadécimale
- Description : Cette commande ôte le point d'arrêt à l'adresse fournie en paramètre. Si l'adresse est erronée, la commande le signale.
- Exemple : on ôte le point d'arrêt à l'adresse 12 et on ôte un point d'arrêt inexistant.

```
>> er 12

>> er 12
[WARNING] 0000000c is not within the breakpoint list.
```

Afficher les points d'arrêt

- Nom de la commande : db (display breakpoint)
- Syntaxe : `db`
- Paramètres : néant
- Description : Cette commande affiche sur la console de visualisation l'adresse des points d'arrêt positionnés dans la mémoire.
- Exemple :

```
>> db
breakpoints
00000004  add $t2,$zero,$zero
0000000c  bne $t2,$t1,65534
```

Chapitre 4

ELF : Executable and Linkable Format

Ce chapitre décrit "brièvement" le format ELF (*Executable and Linkable Format*) et explique comment en extraire les informations nécessaires pour le projet. Le format ELF est le format des fichiers objets dans la plupart des systèmes d'exploitation de type UNIX (GNU/Linux, Solaris, BSD, Androïd...). Il est conçu pour assurer une certaine portabilité entre différentes plates-formes. Il s'agit d'un format standard pouvant supporter l'évolution des architectures et des systèmes d'exploitation.

Le format ELF est manipulable, en lecture et écriture, par utilisation d'une bibliothèque C de fonctions d'accès `libelf`. Cette librairie suffit pour lire et écrire des fichiers ELF, même quand ELF n'est pas le format utilisé par le système d'exploitation (par exemple, on peut l'utiliser sous MacOs X).

Dans ce chapitre, nous nous limitons aux fichiers objets, dits *à lier*, fichiers contenant du *code binaire relogeable* (ou *translatable*), c'est-à-dire du code binaire dont l'affectation en mémoire n'est déterminée qu'au moment de l'exécution. Ce chapitre décrit tout d'abord la structure d'un fichier objet, puis s'intéresse à la notion de relocation.

4.1 Fichier objet au format ELF

Les fichiers ELF sont composés d'un ensemble de sections (éventuellement vides) comme indiqué par la figure 4.1.

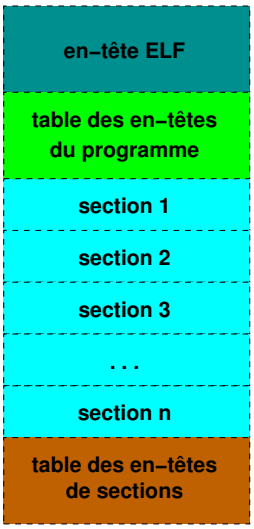


Figure 4.1 – Structure d'un fichier ELF

:

— En-tête du fichier ELF
— Table des entêtes de sections
— Table des noms de sections ("".text"", "".data"", "".rel.text"",...)
— Table des chaînes (noms des symboles)
— Table des symboles (informations sur les symboles)

— Section de données (.text, .data, .bss)
— Tables de relocations (.rel.text, .rel.data)

Les seules sections qui contiennent des données sont les sections :

— `.text` qui contient l'ensemble des instructions exécutables du programme.
— `.data` qui contient les données initialisées du programme.
— `.bss` qui contient les données non initialisées du programme. Ces données ne prennent pas de place dans le fichier ELF : seules les tailles des zones mémoire à réserver sont spécifiées et ces zones sont remplies avec des zéros au début de l'exécution du programme.

Les autres sections servent à décrire le programme et le rendre portable et relogeable.

4.2 Structure générale d'un fichier objet au format ELF et principe de la relocation

Un fichier objet à lier au format ELF est formé d'un en-tête donnant des informations générales sur la version, la machine, etc., puis d'un certain nombre de pointeurs et de valeurs décrits ci-dessous. Ce que nous appelons ici pointeur est en fait une valeur représentant un déplacement en nombre d'octets par rapport au début du fichier. Les tailles, elles aussi, sont exprimées en nombre d'octets. La figure 4.2 donne une idée de la structure d'un fichier au format ELF. Le fichier est constitué d'un *en-tête* donnant les caractéristiques générales du fichier, puis d'un certain nombre de *sections* contenant différentes formes de données, ces sections étant détaillées par la suite (par exemple, section "`.text`", section "`.data`", section des "relocations en zone text", section de la table des symboles, etc).

Lors de la fabrication d'un fichier objet, les instructions du programme sont logées dans une section binaire correspondant à une zone `.text` alors que certaines des opérandes de ces instructions peuvent appartenir à une section binaire différente, par exemple la zone `.data`. Lors du chargement du fichier en mémoire en vue de son exécution (de sa simulation dans notre cas), ces différentes zones sont placées en mémoire par le chargeur. Ce n'est qu'après cette étape que les adresses des opérandes seront connues. Il faut donc mettre à jour la partie adresse effective des instructions afin que ces dernières accèdent correctement aux opérandes (notons qu'avant cette mise à jour, les adresses des opérandes dans les instructions sont des adresses relatives définies par rapport au début de la zone `.data` du fichier initial). Par ailleurs, cette situation peut être généralisée si ce fichier objet fait partie d'un programme plus vaste utilisant plusieurs fichiers objets susceptibles d'être rassemblés en un seul programme par l'éditeur de liens entre fichiers. Par exemple, une opérande en zone `.data` peut être utilisée par des instructions contenues dans des fichiers objets différents. Cette remarque est également valable pour les zones `.text`, par exemple l'appel d'une fonction déclarée dans un fichier externe. Il faut donc indiquer pour chaque section `.text` concernée, un moyen de retrouver l'adresse de cette opérande. Pour cette raison chaque section `.text` ou `.data` possède une section associée dite de relocation (`.rel.text` et `.rel.data`) contenant les informations nécessaires aux calculs des adresses. Comme explicité précédemment, toutes les adresses non définies avant le chargement, sont finalement mise à jour à l'issue du chargement. La partie du code des instructions correspondant à l'adresse des opérandes en question ne peut donc pas être figée au moment de la compilation du fichier objet initial mais seulement à l'issue du chargement du programme. Du fait de cette relocation, les fichiers ELF sont dits "relogeables".

Dans ce projet, pour des raisons de simplicité, nous n'aborderons pas le traitement de l'édition de liens entre plusieurs fichiers ELF. On se contentera de réaliser la simulation d'un fichier objet simple mais susceptible de nécessiter une relocation lors du chargement dans la mémoire du simulateur. Dans la suite de ce chapitre, nous donnons un exemple précis permettant d'illustrer en détails les principes de la relocation.

Entête d'un fichier ELF L'en-tête est décrite par le type C `struct ELF32_Ehdr` dont voici la description des principaux champs :

- `e_ident` : identification du format et des données indépendantes de la machine permettant d'interpréter le contenu du fichier (Cf. exemple dans le paragraphe suivant).
- `e_type` : un fichier relogeable a le type `ET_REL` (constante égale à 1).
- `e_machine` : le processeur MIPS qui nous intéresse est identifié par la valeur `EM_MIPS` (constante égale à 8).
- `e_version` : la version courante est 1.
- `e_ehsize` : taille de l'en-tête en nombre d'octets.
- `e_shoff` : pointeur sur la *table des en-têtes de sections* (ou plus simplement "*table des sections*"). Chaque entrée de cette table est l'en-tête d'une section qui décrit la nature de cette section, et donne sa localisation dans le fichier (voir ci-dessous). La table des sections est appelée *shdr* dans la documentation ELF. Dans le reste du format ELF, les sections sont généralement désignées par l'index de leur en-tête dans cette table. Le premier index (qui est 0) est une sentinelle qui désigne la section "non définie".
- `e_shnum` : nombre d'entrées dans la table des sections.
- `e_shentsize` : taille d'une entrée de la table des sections.
- `e_shstrndx` : index de la table des noms de sections (dans la table des en-têtes de sections).
- Les autres champs de l'en-tête ne sont pas utilisés dans le cadre du projet. On les met à 0.

En-têtes des sections Un en-tête de section est décrit par le type C `struct Elf32_shdr`. Il définit les champs suivants :

- `sh_name` : index dans la table des noms de sections (section "`.shstrtab`").
- `sh_type` : type de la section. Les types qu'on utilise dans ce projet sont les suivants :
  - `SHT_PROGBITS` (constante 1) : type des sections "`.text`" et "`.data`". Ce type indique que la section contient une suite d'octets correspondant aux données ou aux instructions du programme.
  - `SHT_NOBITS` (constante 8) : type de la section "`.bss`" (données non initialisées). La section ne contient aucune donnée. Elle sert essentiellement à déclarer la taille occupée par les données non initialisées (voir ci-dessous).
  - `SHT_SYMBTAB` (constante 2) : type des tables des symboles. Dans le projet, on en utilise une seule, appelée "`.symtab`".
  - `SHT_STRTAB` (constante 3) : type des tables de chaînes. Dans le projet, deux sections ont ce type : la table des noms de sections, appelée "`.shstrtab`", et la table des noms de symboles, appelée "`.strtab`" (elles sont souvent désignées par "table des chaînes").
  - `SHT_REL` (constante 9) : type des tables de relocations. Dans le projet, on aura : "`.rel.text`" pour les relocations en zone `.text` et "`.rel.data`" pour les relocations en zone `.data`.
  - `SHT_REGINFO` : type spécifique aux fichiers ELF de processeurs MIPS 32 bits, pour la section "`.reginfo`" (Register Information Section).
- `sh_offset` : pointeur sur le début de la section dans le fichier.
- `sh_size` : taille qu'occupera la section une fois chargée en mémoire (en octets). Si le type de la section n'est pas `SHT_NOBITS`, la section doit correspondre effectivement `sh_size` octets dans le fichier, puisque la section sera chargée "telle quelle" en mémoire. Si le type de la section est `SHT_NOBITS`, ce champ sert à déclarer la taille de la zone non initialisée qui sera finalement allouée en mémoire.
- `sh_addralign` : contrainte d'alignement sur l'adresse finale de la zone en mémoire. L'adresse finale doit être un multiple de ce nombre.

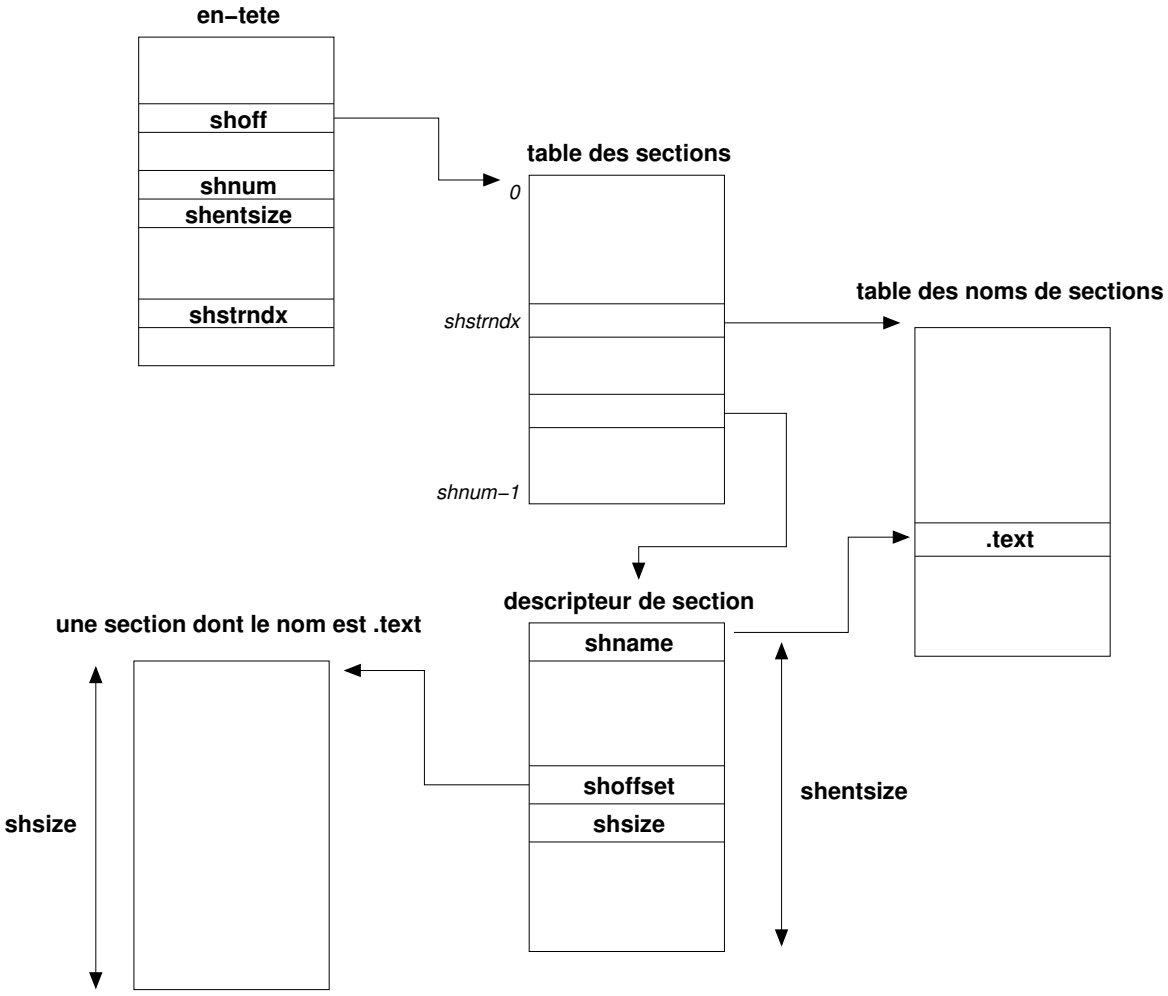


Figure 4.2 – Structure d'un fichier relogeable au format ELF

- `sh_entsize` : certaines sections ont des entrées de taille fixe. Cet entier donne donc cette taille. Dans le projet, seules les sections de type `SHT_SYMBTAB` et `SHT_REL` sont concernées par ce champ.
- `sh_link` et `sh_info` : ont des interprétations qui dépendent du type de la section. Dans tous les cas, le champs `sh_link` est l'index d'un en-tête de section (dans la table des en-têtes de sections). Dans le cadre du projet, on a :

`sh_type`	`sh_link`	`sh_info`
`SHT_REL`	l'index de la table de symbole associée	l'index (dans la table des en-têtes de sections) de la section à reloger
`SHT_SYMTAB`	l'index de la table des noms de symboles	l'index (dans la table des symboles) du premier symbole global [1]

Remarque le contenu d'un fichier objet *exécutable* ressemble à celui d'un fichier objet relogeable. Il est formé de segments au lieu de sections et on y trouve ainsi une table des segments (au lieu d'une table des sections). Dans l'en-tête, les informations décrivant la table des segments sont données par les champs dont les noms commencent par `ph` au lieu de `sh`.

4.3 Exemple de fichier relogeable

Pour étudier le format ELF en détaillant le format de chacune des sections considérées dans le projet, considérons le programme en langage d'assemblage `reloc_miam.s` donné Figure 4.3.

```
# allons au ru
.set noreorder

.text
    ADDI $t1,$zero,8
    Lw $t0 , lunchtime
boucle:
    BEQ $t0 , $t1 , byebye
    NOP
    addi $t1 , $t1 , 1
    J boucle
    NOP
byebye:

.bss
tableau: .space 16

.data
debut_cours: .word 8
lunchtime: .word 12
adresse_lunchtime : .word lunchtime
```

Figure 4.3 – Programme assembleur nécessitant une relocation.

1. On verra en effet en sous-section 4.4.6 que tous les symboles globaux doivent être rassemblés à la fin de la table des symboles.

La commande `mips-as reloc_miam.s -o reloc_miam.o` produit un fichier objet `reloc_miam.o` dont nous donnons en figure 4.4 le contenu affiché par la commande Unix `od -t xC reloc_miam.o`.

```
0000000 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
0000020 00 01 00 08 00 00 00 01 00 00 00 00 00 00 00 00
0000040 00 00 00 bc 00 00 00 01 00 34 00 00 00 00 00 28
0000060 00 0b 00 08 20 09 00 08 3c 08 00 00 8d 08 00 04
0000100 11 09 00 04 00 00 00 00 21 29 00 01 08 00 00 03
0000120 00 00 00 00 00 00 00 08 00 00 00 0c 00 00 00 04
0000140 10 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00
0000160 00 00 00 00 00 00 00 00 00 2e 73 79 6d 74 61 62
0000200 00 2e 73 74 72 74 61 62 00 2e 73 68 73 74 72 74
0000220 61 62 00 2e 72 65 6c 2e 74 65 78 74 00 2e 72 65
0000240 6c 2e 64 61 74 61 00 2e 62 73 73 00 2e 72 65 67
0000260 69 6e 66 6f 00 2e 70 64 72 00 00 00 00 00 00 00
0000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
```

Figure 4.4 – Résultat de `od -t xC reloc_miam.o`.

C'est assez difficile à lire. . . On va donc utiliser les programmes standards `objdump` et `readelf` sous Linux pour analyser les fichiers objets. Il s'agit d'outils capable d'analyser la séquence de bits du fichier pour y retrouver la structure d'un fichier objet, et d'afficher en clair les informations intéressantes. La commande `readelf -a reloc_miam.o` produit :

```
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          188 (bytes into file)
  Flags:                             0x1, noreorder, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000020 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000394 000018 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000054 00000c 00  WA  0   0  4
```

```
  [ 4] .rel.data         REL             00000000 0003ac 000008 08      9   3  4
  [ 5] .bss              NOBITS          00000000 000060 000010 00  WA  0   0  1
  [ 6] .reginfo          MIPS_REGINFO    00000000 000060 000018 01      0   0  4
  [ 7] .pdr              PROGBITS        00000000 000078 000000 00      0   0  4
  [ 8] .shstrtab         STRTAB          00000000 000078 000042 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000274 0000c0 10     10   6  4
  [10] .strtab           STRTAB          00000000 000334 00005e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x394 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000205 R_MIPS_HI16       00000000   .data
00000008  00000206 R_MIPS_LO16       00000000   .data
00000018  00000104 R_MIPS_26         00000000   .text

Relocation section '.rel.data' at offset 0x3ac contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000202 R_MIPS_32         00000000   .data

There are no unwind sections in this file.

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT    3 .data
     3: 00000000     0 SECTION LOCAL  DEFAULT    5 .bss
     4: 00000000     0 SECTION LOCAL  DEFAULT    6 .reginfo
     5: 00000000     0 SECTION LOCAL  DEFAULT    7 .pdr
     6: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 lunchtime
     7: 0000000c     0 NOTYPE  LOCAL  DEFAULT    1 boucle
     8: 00000020     0 NOTYPE  LOCAL  DEFAULT    1 byebye
     9: 00000000     0 NOTYPE  LOCAL  DEFAULT    5 tableau
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 debut_cours
    11: 00000008     0 NOTYPE  LOCAL  DEFAULT    3 adresse_lunchtime

No version information found in this file.
```

4.4 Détail des sections

4.4.1 L'en-tête

```
0000000 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
0000020 00 01 00 08 00 00 00 01 00 00 00 00 00 00 00 00
```

```
0000040 00 00 00 b8 00 00 00 01 00 34 00 00 00 00 00 28
0000060 00 0b 00 08
```

Les 16 premiers octets constituent le champ `e_ident` (taille définie par la constante `EI_IDENT`). Les quatre premiers identifient le format : notons que `0x45, 0x4c` et `0x46` sont les codes ASCII des caractères `'E', 'L', 'F'`. Le cinquième octet `01` donne la classe du fichier, ici `ELFCLASS32`, ce qui signifie que les adresses sont exprimées sur 32 bits. Le sixième donne le type de codage des données, ici 2, ce qui signifie que les données sont codées en complément à deux, avec les bits les plus significatifs occupant les adresses les plus basses (big endian).

Par ailleurs on repère : la taille de l'en-tête (0x34) ; le déplacement par rapport au début du fichier donnant accès à la table des sections (0xb8 octets = 184 en décimal) ; la taille d'une entrée de la table des sections (0x28 = 40 octets), le nombre d'entrées dans la table des sections (0x0b = 11). L'entrée numéro 8 dans la table des sections est celle du descripteur de la table des noms de sections `.shstrtab` (celle-ci est indispensable pour savoir quelle section décrit un autre descripteur).

La commande `readelf -S reloc_miam.o` permet d'obtenir l'en-tête des sections du ELF et de savoir ainsi quelles sont les sections qui le cosntitue. On peut ainsi voir qu'il existe une section `.text` de type `.PROGBITS` (bits appartenant à un programme) se trouvant à l'offset `0x34` et faisant `0x20` octets.

```
There are 11 section headers, starting at offset 0xbc:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000020 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000394 000018 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000054 00000c 00  WA  0   0  4
  [ 4] .rel.data         REL             00000000 0003ac 000008 08      9   3  4
  [ 5] .bss              NOBITS          00000000 000060 000010 00  WA  0   0  1
  [ 6] .reginfo          MIPS_REGINFO    00000000 000060 000018 01      0   0  4
  [ 7] .pdr              PROGBITS        00000000 000078 000000 00      0   0  4
  [ 8] .shstrtab         STRTAB          00000000 000078 000042 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000274 0000c0 10     10   6  4
  [10] .strtab           STRTAB          00000000 000334 00005e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Figure 4.5 – Table des section : résultat de `readelf -S reloc_miam.o`.

4.4.2 La table des noms de sections (`.shstrtab`)

La table des noms de section est accessible par la commande `readelf --hex-dump=8 reloc_miam.o`. En effet, la section `.shstrtab` est à l'indice 8. La table des noms de sections est du type table de chaînes de caractères. Elle contient les noms suivants :

```
0000 002e7379 6d746162 002e7374 72746162  .symtab .strtab
0010 002e7368 73747274 6162002e 72656c2e  .shstrtab .rel.
```

```
0020 74657874 002e7265 6c2e6461 7461002e text .rel.data .
0030 62737300 2e726567 696e666f 002e7064 bss .reginfo .pd
0040 7200                               r.
```

C'est dans cette table que `readelf` va chercher les noms des sections pour remplir la table donnée figure 4.5. Le format ELF impose certaines règles à respecter. La première chaîne doit être la chaîne vide (0x00). Chaque chaîne doit se terminer par le caractère de code ASCII 0 (comme en "C").

4.4.3 La section table des chaînes (`.strtab`)

Cette section (qui porte l'index 10) rassemble tous les noms des symboles utilisés (directement ou implicitement) dans le code. Les règles concernant cette section sont les mêmes que pour `.shstrtab` ci-dessus.

```
0000 002e7465 7874002e 64617461 002e6273  .text .data .bs
0010 73002e72 6567696e 666f002e 70647200 s .reginfo .pdr.
0020 6c756e63 6874696d 6500626f 75636c65 lunchtime boucle
0030 00627965 62796500 7461626c 65617500  byebye tableau
0040 64656275 745f636f 75727300 61647265 debut_cours adre
0050 7373655f 6c756e63 6874696d 6500     sse_lunchtime
```

C'est dans cette table que `readelf` va chercher les noms des symboles, aucun symbole n'est codé "en dur" dans les autres sections.

4.4.4 La section `.text`

La portion de fichier objet correspondant à la zone TEXT (les instructions) est le résultat de la commande `readelf --hex-dump=8 reloc_miam.o` :

```
0000 20090008 3c080000 8d080004 11090004 ....<...........
0010 00000000 21290001 08000003 00000000 ....!)..........
```

Une version plus lisible de la zone TEXT peut être obtenue avec `mips-objdump -d --section=.text reloc_miam.o` :

```
Disassembly of section .text:

00000000 <boucle-0xc>:
   0: 20090008  addi t1,zero,8
   4: 3c080000  lui t0,0x0
   8: 8d080004  lw t0,4(t0)

0000000c <boucle>:
   c: 11090004  beq t0,t1,20 <byebye>
  10: 00000000  nop
  14: 21290001  addi t1,t1,1
  18: 08000003  j c <boucle>
  1c: 00000000  nop
```

On peut par exemple constater que l'instruction `Lw $t0 , lunchtime` a été remplacé par `lui $t0 , 0x0 == 3c080000` et `lw $t0 , 0x4($t0) == 8d080004`.

Pour chaque instruction, les deux premiers octets codent le numéro de l'instruction et le registre $t0 ; les deux derniers correspondent à la valeur numérique de `lunchtime`. En fait la valeur numérique

pour `lui` devrait être les 16 bits de poids fort de `lunchtime` et la valeur numérique pour `lw` devrait être les 16 bits de poids faible de `lunchtime`. Cependant, la valeur de `lunchtime` n'est pas connue. En effet, `lunchtime` est une adresse est cette adresse ne sera connue que lors du chargement final en mémoire, les 16 bits sont donc à zéro ou une valeur temporaire (ici 4 pour lw) en attendant d'être replacés par une valeur lors du chargement grâce à une donnée de relocation `rel.text` qui est associée à cette instruction (cf. section 4.4.7).

4.4.5 La section `.data`

La portion de fichier objet correspondant à la zone DATA (les données initialisées) est donnée par la commande `readelf --hex-dump=3 reloc_miam.o` :

```
0000  00000008 0000000c 00000004
```

La valeur 8 indiquée (pointée par) `debut_cours` est codée sur les 4 premiers octets (c'est un `.word`). La valeur 12 (`0x0000000c == 12`) indiquée (pointée par) `lunchtime` est codée sur les 4 octets suivants (c'est un `.word`). Les 4 octets suivants devraient contenir la valeur immédiate d'adresse correspondant à l'étiquette `lunchtime`. Mais comme l'adresse de `lunchtime` ne sera connue que lors du chargement final en mémoire, les 4 octets codent une valeur, 4, qui est en fait une donnée de translation. Cette information permettra, avec celles contenues dans la zone `.rel.data`, de calculer l'adresse finale de `lunchtime` et donc la valeur de `adresse_lunchtime`.

4.4.6 La section table des symboles

La table des symboles d'un fichier objet est donnée par la commande `readelf -s reloc_miam.o` :

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT    3 .data
     3: 00000000     0 SECTION LOCAL  DEFAULT    5 .bss
     4: 00000000     0 SECTION LOCAL  DEFAULT    6 .reginfo
     5: 00000000     0 SECTION LOCAL  DEFAULT    7 .pdr
     6: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 lunchtime
     7: 0000000c     0 NOTYPE  LOCAL  DEFAULT    1 boucle
     8: 00000020     0 NOTYPE  LOCAL  DEFAULT    1 byebye
     9: 00000000     0 NOTYPE  LOCAL  DEFAULT    5 tableau
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 debut_cours
    11: 00000008     0 NOTYPE  LOCAL  DEFAULT    3 adresse_lunchtime
```

Le type décrivant une entrée de la table des symboles est `struct Elf32Sym`. Une entrée occupe 16 octets, il y a ici 12 entrées. La première entrée est à zéro (elle sert de sentinelle). Les entrées de numéros 1 à 5 sont des symboles spéciaux qui représentent en fait respectivement les sections `.text`, `.data`, `.bss`, `.reginfo` et `.pdr`. On remarque en effet qu'elles ont le type `SECTION` (constante 3). Le champ `Ndx` de ces entrées indique dans quelle section le symbole est défini (par exemple `tableau` est défini dans la section `5 == .bss`). Le champ `Value` indique l'offset a appliquer à partir de la section des symboles pour trouver l'adresse des symboles. Les entrées numéros 6 à 11 sont de vrais symboles

de l'utilisateur, correspondants aux étiquettes `lunchtime`, `boucle`, `byebye` , `tableau`, `debut_cours` et `adresse_lunchtime`.
Les différents champ de la structure `Elf32Sym` sont les suivants :

- `st_name` : index du symbole dans la table des noms de symboles. Lorsque le symbole est de type `STT_SECTION`, aucun nom ne lui est associé. La valeur de l'index est alors 0 (qui désigne donc la chaîne vide, d'après les contraintes sur `.strtab`).
- `st_value` : il vaut 0 pour un symbole non défini ; pour un symbole défini localement, `st_value` est le déplacement (en octets) par rapport au début de la zone de définition.
- `st_shndx` : indique l'index (dans la table des en-têtes de sections) de la section où le symbole est défini. Si le symbole n'est défini dans aucune section (symbole externe), cet index vaut 0. Ainsi **l'index 0 de la table des en-têtes de section est une sentinelle qui sert à marquer les symboles externes**. Si le symbole est de type `STT_SECTION`, cet index est directement l'index de la section.
- `st_size` et `st_other` : non utilisés dans le projet (`st_size` sert à associer une taille de donnée au symbole).
- `st_info` : ce champ codé sur un octet sert en fait à coder 2 champs : le champ "bind" et le champ "type". Les macros suivantes (définie dans les fichiers d'en-tête du format ELF) permettent d'encoder ou de décoder le champ `st_info` :

  ```
  #define ELF32_ST_BIND(i)   ((i)>>4)              /* de info vers bind */
  #define ELF32_ST_TYPE(i)   ((i)&0xf)             /* de info vers type */
  #define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))  /* de (bind,type) vers info */
  ```

  Le champ "bind" indique la portée du symbole. Dans le cadre du projet, on considère les 2 cas suivants :
  - `STB_LOCAL` (constante 0) indique que le symbole est défini localement et non exporté.
  - `STB_GLOBAL` (constante 1) indique que le symbole est soit défini et exporté, soit non défini et importé. Il faut noter qu'à l'édition de lien, il est interdit à 2 fichiers distincts de définir deux symboles de même nom ayant la portée `STB_GLOBAL`.

  Dans le cadre du projet, on ne considère que les 2 cas suivants pour le champ "type" :
  - `STT_SECTION` (constante 3) indiquant que le symbole désigne en fait une section.
  - `STT_NOTYPE` (constante 0), dans les autres cas.

Le format ELF impose certaines contraintes sur l'ordre des symboles dans la table : le premier symbole n'est pas utilisé. Tous les symboles globaux doivent être regroupés à la fin de la table.

4.4.7 Les sections de relocation

Les données de relocation `.rel.text` décrivent comment réécrire certains champs d'instruction incomplets le codage des instructions de la zone `.text`. Ces instructions sont partiellement codées, car contenant des références à des symboles (étiquettes) dont on ne peut pas connaître l'adresse au moment de la fabrication du fichier objet. L'instruction est codée en réservant les 4 octets nécessaires au stockage de la valeur d'adresse mais la valeur indiquée dans le champ est provisoire. Cependant, cette valeur est très importante, comme nous allons le voir. Elle dépend du mode de relocation, et permet de calculer la valeur finale du champs. Les données de relocation `.rel.date`, selon le même principe, décrivent comment réécrire certaines valeurs dans la section data.

Format de la table de relocation

Une entrée de la table de relocation est représentée par le type `struct Elf32_Rel` :

- `r_offset` : contient un décalage en octets par rapport au début de la section. Cette valeur indique la position dans la zone à reloger de l'instruction qui contient un champ incomplet.
- `r_info` : est un champ codé sur 4 octets composé en fait de deux champs :.
  - Le champ "`sym`" est l'index du symbole à reloger dans la table des symboles.
  - Le champ "`type`" indique le mode de calcul de la relocation. Dans le cadre du projet, il prend par exemple les valeurs `R_MIPS_32` (constante 2) ou `R_MIPS_LO16` (constante 6).

Les macros de codage/décodage du champ `r_info` défini dans les fichiers d'en-têtes de la librairie ELF sont :

```
#define ELF32_R_SYM(i)    ((i)>>8)                      /* info vers sym */
#define ELF32_R_TYPE(i)   ((unsigned char)(i))          /* info vers type */
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t)) /* (sym,type) vers info */
```

Modes de calcul de la relocation

Détaillons maintenant le mode de calcul de la relocation, c'est-à-dire la valeur que l'éditeur de lien (ou le chargeur de notre simulateur) met finalement dans les champs incomplets des instructions. Les notations sont les suivantes :

V désigne la **V**aleur "finale" du champ accueillant le relogement.

P désigne la **P**lace, c'est à dire l'adresse "finale" de l'élément à reloger.

S désigne l'adresse "finale" du **S**ymbole à reloger fournie par le chargeur.

A désigne la valeur à **a**jouter pour calculer la valeur du champ à reloger (c'est la valeur du champ avant relogement).

AHL désigne un autre type de valeur à **a**jouter qui est calculée à partir de la valeur provisoire A[2].

Les adresses finales des sections `.text`, `.data`, `.bss` sont normalement déterminées lors de l'édition de liens. Dans notre simulateur, elles sont calculées lors du chargement des sections en mémoire suivant les contraintes définies section 3.6.1.

Le mode de calcul dépend du type de relocation, codé dans le champs `type` de `r_info`. Les principaux modes de calcul sont :

- `R_MIPS_32` (constante 2) : la valeur mise à l'adresse P vaut $V = S + A$. Ce mode sert pour les adressages directs.
- `R_MIPS_26` (constante 4) : le calcul se décompose en plusieurs étapes :
  - calcul de l'adresse de saut (comme décrit section 2.3.2) : $(P\&0xf0000000)+S$
  - ou logique avec A décalé de 2 à gauche : $(A << 2) \mid ((P\&0xf0000000)+S)$
  - résultat décalé de 2 à droite : $V = ((A << 2) \mid ((P\&0xf0000000)+S)) >> 2$

  Ce mode sert pour les adressages absolus alignés sur 256Mo (pour les *J-instructions*).
- `R_MIPS_HI16` (constante 5) : la valeur mise à l'adresse P vaut $V = (AHL + S - (short)AHL + S) >> 16$. Ce mode sert pour remplacement des accès à la section data par étiquette (`lw, sw, lb, sb`...). Une relocation `R_MIPS_HI16` est toujours suivi d'une relocation `R_MIPS_LO16` car la valeur AHL est calculée par $(AHI << 16)+(short)(ALO)$ ou AHI est le A de l'instruction ayant une relocation `R_MIPS_HI16` et ALO est le A de l'instruction ayant une relocation `R_MIPS_LO16`.
- `R_MIPS_LO16` (constante 6) : la valeur mise à l'adresse P vaut $AHL + S$. Ce mode sert pour les adressages immédiats avec une valeur sur 16 bits.

La section de relocation `.rel.data`

La relocation est peut-être plus simple à comprendre en regardant le résultat de la commande `readelf -r reloc_miam.o`. Sur l'exemple, la table des relocations en data texte est :

```
Relocation section '.rel.data' at offset 0x3ac contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000202 R_MIPS_32         00000000   .data
```

Ici on va chercher à calculer $V = S + A$. Le champ `info` vaut 0x00000202, donc se décompose en :

```
sym = (info)>>8 = 0x00000202 >> 8 = 0x00000002 = 2
type = (unsigned char)(info) = (unsigned char)(0x00000202) = 0x02
```

Le champ `type` vaut 2, ce qui signifie que le mode de relocation est `R_MIPS_32`. Il faut donc déterminer A et S. Le champ `sym` vaut également 2 ce qui signifie qu'il s'agit de l'adresse de la zone `.data`. On a donc S = adresse de `.data` = $0x0$. A est la valeur du champ présent dans le code à l'adresse de l'instruction. Le champ `r_offet` vaut `0x8`, si on se réfère à la section 4.4.5, la valeur sur 32 bits à l'adresse 8 de la section data est 0x00000004. Nous avons donc $A = 0x4$ et $V = S + A = 0x0 + 0x4 = 0x4$. On va trouver que le contenu de P sera égal à l'adresse finale de la zone `.data` + 4. C'est normal, on fait ici référence à `lunchtime`, qui est le deuxième mots de 32 bits de la zone data.

La section de relocation `.rel.text`

Pour la zone TEXT, la relocation fonctionne de la même manière :

```
Relocation section '.rel.text' at offset 0x394 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000205 R_MIPS_HI16       00000000   .data
00000008  00000206 R_MIPS_LO16       00000000   .data
00000018  00000104 R_MIPS_26         00000000   .text
```

Pour la première entrée le champs `r_offet` vaut `0x4`, ce qui signifie que l'instruction à reloger est la deuxième instruction de la zone `.text`. Effectivement, il s'agit bien de `lui $t0, 0x0`.

Le champ `info` vaut 0x00000205, donc se décompose en :

```
sym = (info)>>8 = 0x00000205 >> 8 = 0x00000002 = 2
type = (unsigned char)(info) = (unsigned char)(0x00000205) = 0x05
```

Le champ `type` vaut 5, ce qui signifie que le mode de relocation est `R_MIPS_HI16`. Il faut donc déterminer S et AHL. Pour calculer ce dernier il faut récupérer AHI et ALO à partir des instructions. Le AHI est le A de l'instruction en adresse $0x4$ soit `lui $t0, 0x0 == 3c080000`. Dans le cas d'une relocation `R_MIPS_HI16` se sont les 16 bits de poids faible que l'on récupère soit $AHI = A = 0x0000$. Le ALO est le A de l'instruction en adresse $0x8$ soit `lw $t0,4($t0) == 8d080004`. Dans le cas d'une relocation `R_MIPS_LO16` se sont les 16 bits de poids faible que l'on récupère soit $ALO = A = 0x0004$. Nous pouvons donc calculer $AHL = (AHI << 16) + (short)(ALO) = 0x0 << 16 + (short)0x4 = 0x000004$

Le champ `sym` vaut 2 et correspond au début de la zone `.data`. Ici S = `Val.-sym` = 00000000.

V vaut donc $V = (AHL + S - (short)AHL + S) >> 16 = (0x04 + 0x0 - (short)0x4 + 0x0) >> 16 = (0x0) >> 16 = 0x0$, les 16 bits de poids faible de l'instruction prennent donc la valeur 0000. C'est bien ce qui était codé à la fin de 3c080000

Essayez maintenant d'effectuer la relocation des entrées 2 et 3 de la table.

4.4.8 Autres sections

Il reste trois sections à décrire :

- La section `.bss` contient uniquement la taille à réserver en mémoire pour les données non initialisées.
- La section `.reginfo` (*Register Information Section*) indique l'usage des registres dans le fichier objet. On ne s'y intéressera pas dans le projet.
- La section `.pdr` (*Procedure Descriptor*). On ne s'y intéressera pas dans le projet.

Chapitre 5

À propos de la mise en œuvre

5.1 Méthode de développement

La conduite d'un projet de programmation de taille "importante" dans un contexte de travail en équipe (qui n'offre pas que des avantages !) et multitâches (autres cours en parallèles) nécessite une méthodologie qui vous permettra de parvenir efficacement à un code qui, par exemple : réponde au problème, soit robuste (absence de plantage), fiable (absence d'erreur, ou gestion appropriée des erreurs), clair et lisible, maintenable (facilité de reprise et d'évolution du code), etc.

Par ailleurs, le développement de programmes nécessite de nos jours de plus en plus de réactivité aux modifications de toutes sortes (p.ex. : demandes des clients, changement d'équipe, bugs, évolutions de technologie) ce qui a conduit à des méthodes de conception s'écartant des schémas classiques de conception/implémentation pour adopter un processus plus souple, plus facile à modifier en cours de développement.

La mise au point de méthodologies de développement est l'une des activités du "Génie Logiciel", une branche de l'informatique. Une telle méthodologie définit par exemple le cycle de vie qu'est censé suivre le logiciel, les rôles des intervenants, les documents intermédiaires à produire, etc.

À l'occasion du projet, vous expérimenterez non pas une méthodologie complète – ce serait trop lourd – mais deux méthodes, ou moyens méthodologiques, qui sont promus par plusieurs méthodologies : le *développement incrémental* et le *développement piloté par les tests*.

5.1.1 Notion de cycle de développement ; développement incrémental

La notion de cycle de vie du logiciel correspond à la façon dont le code est construit au fur et à mesure du temps, depuis la rédaction des spécifications jusqu'à la livraison du logiciel, en passant par l'analyse, l'implantation, les tests, etc. Plusieurs cycles de vie sont possibles.

Dans le cadre du projet, vous adopterez un *cycle incrémental* - appelé également *cycle en spirale*. Il s'agit de découper les fonctionnalités du logiciel de telle sorte qu'il soit construit progressivement, en plusieurs étapes. À l'issue de chaque étape, le logiciel ne couvre pas toutes les fonctionnalités attendues, mais doit être fonctionnel - et testable - pour la sous-partie des fonctionnalités réalisées jusqu'ici.

Les avantages du développement incrémental dans le cadre du projet sont nombreux. Par exemple, les risques sont réduits : au lieu de ne pouvoir tester votre logiciel qu'à la fin du projet et de risquer que rien ne marche, vous aurez à chaque étape quelque chose de partiel, certes, mais qui fonctionne. Incidemment, cela tend à garantir une plus grande implication dans l'activité de développement, puisqu'on voit la chose - le logiciel - se construire au fur et à mesure.

À l'inverse, le développement incrémental peut nécessiter une plus grande dextérité : durant un incrément, il peut être nécessaire de développer une "fausse" partie du logiciel pour "faire comme si" cette partie existait... Puis être capable de la remplacer intégralement lors de l'incrément suivant. Une autre des difficultés du développement incrémental tient à la définition des incréments et de l'ordre de leur réalisation – c'est-à-dire du découpage temporel de l'activité de développement. Fort heureusement pour vous, le découpage est déjà défini pour le projet !

Vous réaliserez votre projet en quatre incréments. Chaque incrément vous occupera de une à trois semaines (typiquement : une séance de TD, une séance de TP et une séance en temps libre).

Par ailleurs, on ne résiste pas à vous donner les conseils/rappels suivants inspirés de la programmation structurée et incrémentale :

— Séparer le projet en modules indépendants de petites tailles en fonction des fonctionnalités désirées du programme.
— Choisir des solutions simples pour les réaliser (ce qui ne veut pas dire les SEULES solutions que vous connaissez).
— Concevoir les tests des modules AVANT leur écriture (concevoir les tests avant permet de bien réfléchir sur le comportement attendu).
— Intégrer la génération de traces pour faciliter le débogage.
— Commenter le code pendant l'écriture du code (après, c'est trop tard).
— Bien définir les rôles de chaque membre de l'équipe (p.ex. : écriture des tests, des structures de données, des rapports, etc.).
— Discuter du projet avant chaque phase de travail.
— ! Se mettre d'accord sur les standards de programmation [1] ! (p.ex. : organisation des dossiers, include, makefile, éditeurs, commentaires, nom des variables, etc.)
— ...

À toute fin utile vous pouvez consulter le site de la communauté de l'*eXtreme Programming* [2] qui fourmille de conseils intéressants.

5.1.2 Développement piloté par les tests

Pour développer un logiciel, il est possible de travailler très précisément ses spécifications, de s'en imprégner, de beaucoup réfléchir, de développer... Et de ne tester le logiciel qu'à la fin, lorsqu'il est fini. Cette démarche est parfois appropriée mais, dans le cadre du projet, vous renversez le processus pour vous appuyer sur la méthode dite de "développement piloté par les tests".

Pour chaque incrément, vous commencerez donc par écrire un jeu de tests avant même d'écrire la première ligne de code. Chaque test du jeu de tests sera constitué :

— d'un fichier texte **.simcmd** comprenant le code des commandes du simulateur à exécuter (par exemple : charger un fichier elf, lancer le programme, vérifier une valeur en mémoire) ;
— d'un fichier texte **.res** comprenant le résultat qu'est censé fournir (afficher) votre programme ;
— d'un fichier texte **.info** qui décrit le test - et qui, en particulier, indique si le test est censé échouer ou réussir.

Ce n'est qu'après avoir écrit votre jeu de tests que vous réfléchirez à la structure de votre programme et que vous le développerez.

Au début de l'incrément, aucun des tests du jeu de tests ne doit "passer". À l'issue de l'incrément, tous les tests que vous avez écrits doivent "passer" : votre programme doit produire le résultat attendu pour chacun des tests.

Voici quelques-uns des avantages du *développement piloté par les tests*.

— Cette méthode garantit que le programme sera accompagné d'un jeu de tests - alors que, trop souvent, les développeurs "oublient" d'écrire des tests.
— écrire les jeux de tests est un très bon moyen pour assimiler les spécifications du programme et s'assurer qu'on les a comprises. Le jeu de tests doit faire apparaître les situations simples, mais aussi faire ressortir les cas particuliers et plus complexes. Ce faisant, le développement lui même peut être guidé par une vue d'ensemble de ce que doit faire le programme, dans

1. https://computing.llnl.gov/linux/slurm/coding_style.pdf
2. http://www.extremeprogramming.org

laquelle, dès le début, on a en tête non seulement les cas simples, mais aussi les situations plus délicates.

— L'existence d'un jeu de tests permet de très vite détecter les *problèmes de régression*, c'est à dire les situations ou, du fait de modifications introduites dans le code, une partie du programme qui fonctionnait jusqu'ici se met à poser problème. Or, la méthode incrémentale que vous adopterez tend à ce que le code soit souvent modifié pour tenir compte des nécessités d'un nouvel incrément. . .et donc à ce que des problèmes de régression surgissent.

Plus concrètement, vous vous astreindrez à faire tourner votre programme très souvent – le plus souvent possible, en fait – sur l'ensemble de vos tests ; cela vous permettra de détecter rapidement les problèmes et de vous assurer que votre programme, petit à petit, se construit dans la bonne direction.

5.1.3 À propos de l'écriture des jeux de tests

L'écriture d'un jeu de tests pertinent – c'est-à-dire à même de raisonnablement "prouver" que votre logiciel fait ce qu'il doit faire – n'est pas chose évidente.

Pour vous mettre en jambe, nous vous donnons un jeu de tests pour le premier incrément. Il vous faudra le lire et l'analyser avec précision.

Pour les incréments suivants, voici quelques conseils pour l'écriture des jeux de tests.

1. un bon jeu de tests comprend nécessairement de nombreux tests - par exemple entre 20 et 100 pour chaque incrément.
2. commencer par écrire des tests très courts et très simples. Par exemple, pour le premier incrément, lancer une commande inconnue et vérifier que le programme renvoie bien une erreur du bon type. La simplicité de ces tests facilitera la recherche du problème si un test échoue.
3. penser également à écrire des tests complexes : les problèmes peuvent surgir du fait de l'enchaînement d'instructions !
4. se remuer les méninges face aux spécifications ; essayer d'imaginer les cas qui risquent de poser problème à votre programme ; pour chacun d'eux, écrire un test !
5. **important** : un jeu de tests se doit de tester ce qui doit *fonctionner*. . . mais aussi ce qui doit *échouer* et la façon dont votre programme gère les erreurs ! Pour ce faire, un jeu de tests doit *aussi* inclure des tests qui font échouer le programme. Par exemple, un premier test pourrait être de charger un elf corrompu afin de s'assurer que votre programme renvoie une erreur.

5.1.4 Organisation interne d'un incrément

On a déjà indiqué que la réalisation de chacun des quatre incréments commencera par l'écriture du jeu de tests, et devra se terminera par un test du programme dans lequel tous les tests du jeu de tests "passent". Mais comment s'organiser à l'intérieur de l'incrément ? Voici quelques conseils. . .

1. réfléchir à la structure de votre programme. Définir en commun les principales structures de données et les signatures et "contrats" (ou "rôle") des principales fonctions.
2. Autant que faire se peut, essayer d'organiser l'incrément... de façon incrémentale ! La notion de développement incrémental peut en effet s'appliquer récursivement : à l'intérieur d'un incrément, il est souhaitable de définir des étapes ou "sous-incréments" qui vous assurent que votre programme est "partiellement fonctionnel" le plus souvent possible.
3. Réfléchir à la répartition du travail. En particulier, éviter que la répartition ne bloque l'un d'entre vous si l'autre prend du retard.

4. Compiler très régulièrement. Il est particulièrement fâcheux de coder pendant plusieurs heures pour ne s'apercevoir qu'à la fin que la compilation génère des centaines d'erreurs de syntaxe. Un développeur expérimenté fait tourner le compilateur (presque) tout le temps en tâche de fond !
5. Exécuter souvent le jeu de tests, afin de mesurer l'avancement et de faire sortir le plus tôt possible les éventuelles erreurs.

5.2 Quelques conseils sur le projet

5.2.1 Automate à états

La première phase du logiciel est de vérifier que la ligne donnée en entrée contient uniquement des éléments acceptés par le langage et de les identifier. Par exemple, il faut pouvoir reconnaître que la chaîne de caractères `0123` est une valeur octale et que `0x123` est une valeur hexadécimale, que `exit` est une commande. Un moyen brutal serait de comparer les caractères du fichier avec toutes les chaînes possibles du langage (avec par exemple `strcmp`). Ceci est bien entendu impossible car : les possibilités sont trops importantes, les tailles des chaînes peuvent varier, il est nécessaire de bien identifier le début et la fin des chaînes d'intérêt pour éviter les recouvrements (p.ex. : trouver `.text` dans le commentaire `# la section .text`)...

Heureusement, le langage est complètement déterministe et il est possible de connaître la composition de chaque élément terminal du langage. Par exemple, on sait qu'une valeur hexadécimale est toujours préfixée par `0x` puis un certain nombre de caractères $\in$ `[0,9]` $\cup$ `[a,b,c,d,e,f]` alors qu'une valeur octale n'est composée que de chiffres inférieurs à 8. En tournant les choses de cette façon le problème devient de trouver des *motifs* de caractères dans le texte et non plus des mots prédéfinis. Un autre problème vient du fait que les motifs peuvent partager des caractéristiques communes qui impliquent qu'il faut avoir lu un certain nombre de caractères avant de reconnaître un motif (p.ex. : tant que l'on a pas lu le 'x' après un zéro on ne sait pas si on lit un nombre hexadécimal ou octal).

Une façon d'aborder le problème est de représenter les motifs et leur parcours par un automate à états. Succinctement, l'automate est composé d'états qui dans notre cas représentent la catégorie courante de la chaîne de caractères lue et de transitions entre états étiquetés par les caractères que l'on va lire. L'exemple de la figure 5.1 montre comment on peut représenter un automate faisant la différence entre nombres décimaux, octaux ou hexadécimaux.

Cet automate lit les caractères un par un jusqu'à arriver à l'état terminal ou erreur. Ainsi, en prenant l'exemple de la chaîne `0567`, l'automate passe successivement par `INIT`, `pref hexa`, et reste dans `octal` jusqu'à la fin donnant ainsi la catégorie de la chaîne. La figure 5.2 donne une traduction en langage C de l'automate (il existe bien d'autre moyens de traduire l'automate en C).

Dans cette traduction, le fichier est parcouru caractère par caractère (boucle `while`). L'automate est tout d'abord dans l'état `INIT`. La lecture de chaque caractère peut provoquer une transition vers un autre état (par exemple si `c` est un chiffre), laisser l'automate dans l'état présent (p.ex., si `c` est un saut de ligne), faire terminer la tâche (p.ex., `EOF` *End Of File*) ou encore détecter une erreur. Ce programme est capable de traiter de gros fichiers textes en peu de temps.

Dans le cadre du projet, l'automate sera bien entendu plus conséquent. Il serait très judicieux de commencer par étudier les différents éléments du langage, d'identifier leur motifs, de construire l'automate, et de prévoir les fichiers de tests, avant de commencer à coder...Il est possible que certains traitements à effectuer dans certains états soient réutilisables, n'hésitez donc pas à fragmenter votre code en fonctions.

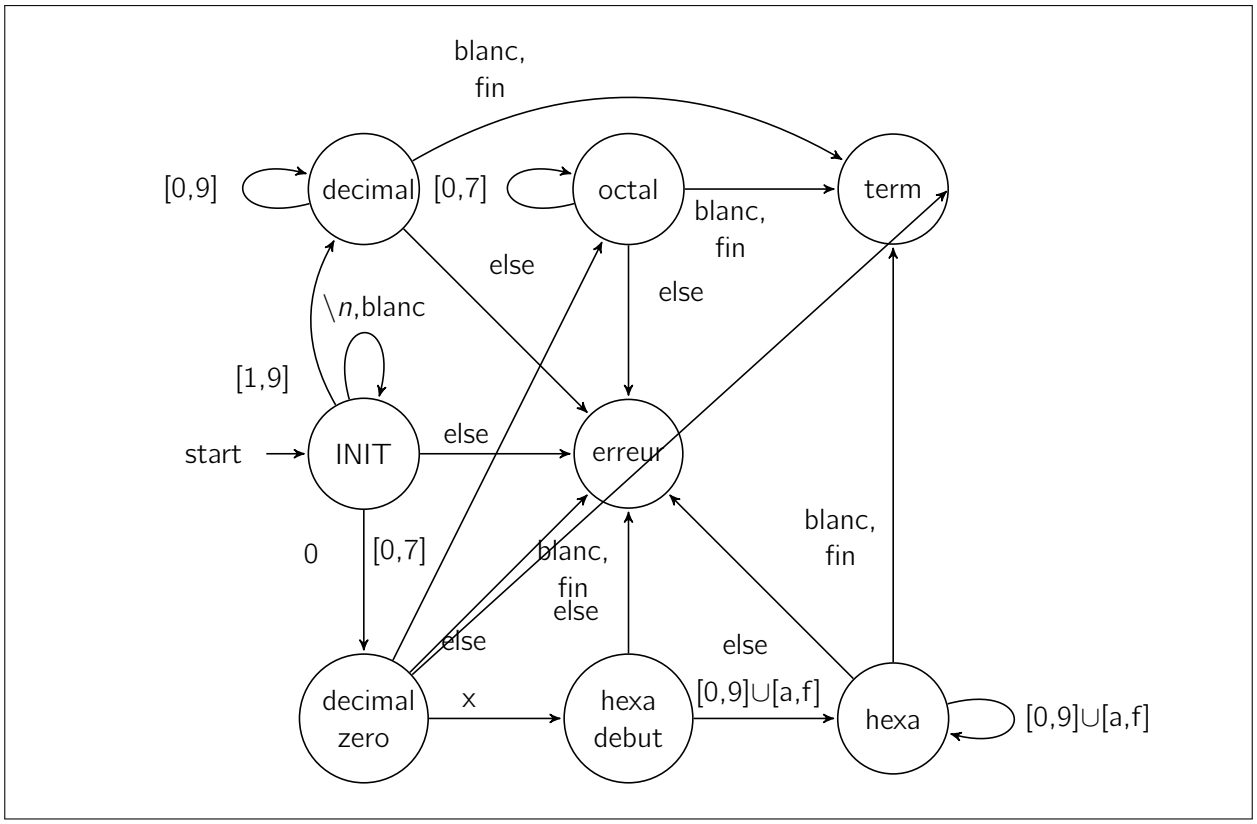


Figure 5.1 – Exemple d'automate faisant la différence entre une valeur décimale, octale et hexadécimale

5.2.2 Représentation des instructions MIPS

Votre simulateur doit charger un code binaire et extraire chaque instruction de l'assembleur (c'est le désassemblage). Cependant, où cette connaissance sera-t'elle stockée ? Comment la représenter ? Comment y accéder ?

La solution de coder *en dur* les instructions une par une dans le code est bien évidement à rejeter. D'une manière générale, on ne mélange pas la connaissance opérationnelle (c.-à-d., le code) et les données. La meilleure option pour le projet est de représenter les instructions dans un fichier texte séparé et de charger les instructions au début de l'exécution de l'assembleur. Le flot peut être ensuite parcouru pour chercher les informations avec la fonction standard `strtok`. L'extrait de code ci-dessous illustre son utilisation avec une chaîne de caractères.

La chaîne de caractères `"Dupond 20 76"` est séparée en éléments délimités par les espaces. Chaque appel à strtok() renvoie le prochain élément. Ainsi, on peut stocker des données sous forme de chaîne de caractères dans un fichier et lire (charger) ces informations dans des structures de données au démarrage du programme pour un accès rapide en mémoire (l'accès au fichier est lent). Des informations complètes sont accessibles dans le `man` de `strtok`.

```c
/* definition des etats*/
enum {INIT, DECIMAL_ZERO, DEBUT_HEXA, HEXA, DECIMAL, OCTAL};
/* mise en oeuvre de l'automate*/
int main() {
    int c;              /*caractere analyse courant*/
    int S=INIT;         /*etat de l'automate*/
    FILE *pf;           /*pointeur du fichier à analyser*/

    if((pf=fopen("nombres.txt","rt"))==NULL) {
        perror("erreur d'ouverture fichier");return 1;}

    while(EOF!=(c=fgetc(pf))) {
        switch(S) {
        case INIT:
            i=0;
            if(isdigit(c)) {    /* si c'est un chiffre*/
                S = (c=='0')? DECIMAL_ZERO : DECIMAL;
            }
            else if (isspace(c)) S=INIT;
            else if (c==EOF) return 0; /* fin de fichier*/
            else return erreur_caractere(string,i,c);
            break;
        case DECIMAL_ZERO: /*reperage du prefixe de l'hexa*/
            if (c == 'x' || c == 'X') S=HEXA;
            else if (isdigit(c) && c<'8') S=OCTAL;/* c'est un octal*/
            else if (c==EOF || isspace(c)){ S=INIT;
                printf("la chaine est sous forme édcimale\n");
            }
            else return  erreur_caractere(string,i,c);
            break;
        case DEBUT_HEXA: /* il faut au moins un chiffre apres x*/
            if(isxdigit(c)) S=HEXA;
            else return erreur_caractere(string,i,c);
            break;
        case HEXA: /* tant que c'est un chiffre hexa*/
            if(isxdigit(c)) S=HEXA;
            else if(c==EOF || isspace(c)) { S=INIT;
                printf("la chaine est sous forme éhexadcimale\n");
            }
            else return erreur_caractere(string,i,c);
            break;
        case DECIMAL: /*tant que c'est un chiffre*/
            if(isdigit(c))  S=DECIMAL;
            else if(c==EOF || isspace(c)) {  S=INIT;
                printf("la chaine est sous forme édcimale\n");
            }
            else  return erreur_caractere(string,i,c);
            break;
        case OCTAL: /*tant que c'est un chiffre*/
            if(isdigit(c)&& c<'8') S=OCTAL;
            else if(c==EOF || isspace(c)) { S=INIT;
                printf("la chaine est sous forme octale\n");
            }
            else  return erreur_caractere(string,i,c);
            break;
        }
    }
    return 0;
}
```

Figure 5.2 – Exemple de traduction en C de l'automate de la figure 5.1

```
/* test strtok */
#include <string.h>/* prototype de la fonction strtok*/
#include <stdio.h>
typedef struct {char* nom; int age; int poids;} Personne;

void main(){
char *token;
char *texte = strdup("Dupond␣20␣76");
char *delimiteur = "␣";
Personne pers;

/*renvoie un pointeur vers "Dupond". */
printf("%s\n", pers.nom=strdup(strtok(texte, delimiteur)));

/* renvoie l'entier "20". */
printf("%d\n", pers.age=atoi(strtok(NULL, delimiteur)));

/* renvoie l'entier "76". */
printf("%d\n", pers.age=atoi(strtok(NULL, delimiteur)));
}
```

Figure 5.3 – Extrait de code illustrant l'usage de `strstok()`

Chapitre 6

Travail à réaliser

La page web du projet informatique est disponible à l'adresse suivante : http://tdinfo.phelma.grenoble-inp.fr/2Aproj/. Veuillez vous y référer pour tous ce qui concerne l'organisation et l'évaluation.

6.1 Objectif général :

À la fin de ce projet vous devrez avoir réalisé un simulateur qui prend en entrée un fichier objet elf pour l'architecture MIPS32 `file.o` et simule son exécution.

Le simulateur sera appelé en tapant sous Linux la commande :

```
sim-mips elf_filename
```

ou

```
sim-mips (avec chargement à travers l'interface)
```

où `elf_filename` est le nom du fichier objet à simuler.

6.2 Étapes de développement du programme

Le projet est découpé en 4 étapes principales que vous aurez à achever dans un délai imparti. Au début de chaque étape, une séance globale de tutorat sera utilisée pour la préparation puis quelques séances de codages seront consacrées à la mise en œuvre. Les quatre étapes seront :

1. Le simulateur et son interpréteur de commandes : À la fin de cette étape, vous devrez avoir l'environnement du simulateur initialisé et un interpréteur exécutant des commandes simples.
2. Décodage des instructions : À la fin de cette étape, vous devrez avoir un simulateur qui est capable de charger un ficher objet elf (sans relocation) et de lire les instructions.
3. Simulation : À la fin de cette étape, vous devrez avoir un programme capable d'exécuter le code machine pas à pas ainsi qu'en utilisant des points d'arrêt.
4. Gestion de la relocation : À la fin de cette étape, votre simulateur devra être capable de charger et d'exécuter du code relogeable.

Les détails de chacune de ces étapes sont à recueillir sur le site web du projet.

6.3 Bonus : extensions du programme

Plusieurs extensions possibles du programme peuvent être envisagées, telles que la prise en compte d'un plus grand nombre de relocations, la prise en compte des informations de débogage du elf, d'une simulation du pipeline d'exécution du MIPS, de la prise en compte des instruction sur les *float*, etc. Toute extension menée de manière satisfaisante amènera un bonus dans la notation.

Bibliographie

[1] B. W. Kernighan et D. M. Ritchie *Le langage C, Norme ANSI* http://http://cm.bell-labs.com/cm/cs/cbook/

[2] Bradley Kjell. *Programmed Introduction to MIPS Assembly langage.* http://chortle.ccsu.edu/AssemblyTutorial/TutorialContents.html

[3] *MOPS32 Architectur For Programmers Volume II, Revision2.50*. 2001-2003,2005 MIPS Technologies Inc. http://www.mips.com/products/product-materials/processor/mips-architecture/

[4] *Executable and Linkable Format (ELF)*. Tools Interface Standard (TIS). Portable Formats Specification, Ver 1.1. http://www.skyfree.org/linux/references/references.html

[5] *64-bit ELF Object File Specification*. http://techpubs.sgi.com/library/manuals/4000/007-4658-001/pdf/007-4658-001.pdf

[6] *System V Application Binary Interface - MIPS® RISC Processor.* http://www.caldera.com/developers/devspecs

[7] Amblard P., Fernandez J.C., Lagnier F., Maraninchi F., Sicard P. et Waille P. *Architectures Logicielles et Matérielles.* Dunod, collection Siences Sup., 2000. ISBN 2 10 004893 7.

[8] Dean Elsner, Jay Fenlason and friends. *Using as, the* GNU *Assembler*. Free Software Foundation, January 1994. http://www.gnu.org/manual/gas-2.9.1/

[9] David Alex Lamb. Construction of a peephole optimizer, *Software : Practice and Experience,* **11(6)**, *pages 639–647, 1981*.

[10] FSF. *GCC online documentation*. Free Software Foundation, January 1994. http://gcc.gnu.org/onlinedocs/

[11] Steve Chamberlain, Cygnus Support. *Using ld, the* GNU *Linker*. Free Software Foundation, January 1994. http://www.gnu.org/manual/ld-2.9.1/

[12] Linux Assembly http://linuxassembly.org/

[13] Linus Torvalds. *Linux Kernel Coding Style.* https://computing.llnl.gov/linux/slurm/coding_style.pdf

[14] Bernard Cassagne. *Introduction au langage C.* http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html

Annexe A

Spécifications détaillées des instructions

Cette annexe contient les spécifications des instructions étudiées dans ce projet. Elles sont directement issues de la documentation du MIPS fournie par le *Architecture For Programmers Volume II* de *MIPS Technologies* [3].

A.1 Définitions et notations

Commençons par rappeller quelques définitions et notations utiles.

Octet/Mot Un *octet* (byte en anglais) est une suite de 8 bits qui constitue la plus petite entité que l'on peut adresser sur la machine. La concaténation de deux octets forme un *demi-mot* (half-word) de 16 bits, et la concaténation de quatre octets, ou de deux demi-mots, forme un *mot* (word) de 32 bits. Les bits sont numérotés de la droite (poids faible) vers la gauche (poids fort) de 0 à 7 pour l'octet, de 0 à 15 pour un demi-mot et de 0 à 31 pour un mot.

0	1	1	0	1	1	1	0

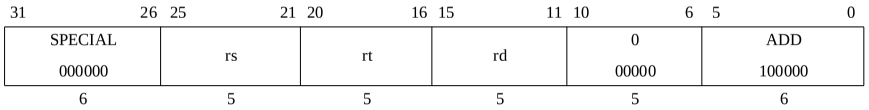
Figure A.1 – Octet

Représentation hexadécimale d'un octet/mot On représente par 0xij la valeur d'un octet dont les 4 bits de poids fort valent i et les 4 bits de poids faible j (avec $i, j \in ([0...9, A...F])$. Par exemple, la valeur de l'octet de la figure A.1 s'écrit 0x6E en hexadécimal. Pour un demi-mot ou un mot, on aura respectivement 4 ou 8 chiffres hexadécimaux, chacun représentant 4 bits.

Codage binaire d'un entier non signé Quand on parle d'un entier non signé codé sur n bits ou plus simplement d'un entier codé sur n bits, il s'agit de sa représentation en base 2 sur n bits, donc d'une valeur entière comprise entre 0 et $2^n - 1$. Un entier codé sur un octet a donc une valeur comprise entre 0 et 255 correspondant aux images binaires 0x00 à 0xFF, un entier codé sur un demi-mot a une valeur comprise entre 0 et 65535 correspondant aux images binaires 0x0000 à 0xFFFF, et un entier codé sur un mot a une valeur comprise entre 0 et 4294967295 correspondant aux images binaires 0x00000000 à 0xFFFFFFFF. Les adresses du processeur de la machine MIPS sont des entiers non signés sur 32 bits.

Codage binaire d'un entier signé Les entiers signés sont représentés en complément à 2. Le codage sur n bits du nombre i est la représentation en base 2 sur n bits de $2^n + i$, si $-2^{n-1} \leq i \leq -1$, et de i, si $0 \leq i \leq 2^{n-1} - 1$. Un entier signé codé sur un octet est compris entre -128 à 127 correspondant aux images binaires 0x80 à 0x7F. Un entier signé sur un demi-mot est compris entre -32768 et 32767 correspondant à l'intervalle binaire 0x8000 à 0x7FFF. Enfin, un entier signé sur un mot est compris entre -2147483648 et 2147483647 correspondant à l'intervalle binaire 0x80000000 à 0x7FFFFFFF. On remarque que le bit de plus fort poids d'un octet/mot/long mot représentant un entier négatif est toujours égal à 1 alors qu'il vaut 0 pour un nombre positif (c'est le bit de signe).

Add Word — ADD



Format: `ADD rd, rs, rt` **MIPS32**

Purpose:

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

Description: `GPR[rd] ← 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*.

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
```

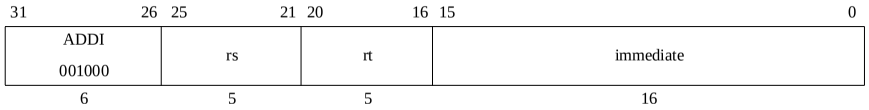
Exceptions:

Integer Overflow

Programming Notes:

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

Add Immediate Word — ADDI



Format: `ADDI rt, rs, immediate` **MIPS32**

Purpose:

To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: `GPR[rt] ¨ GPR[rs] + immediate`

The 16-bit signed *immediate* 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 *rt*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp
endif
```

Exceptions:

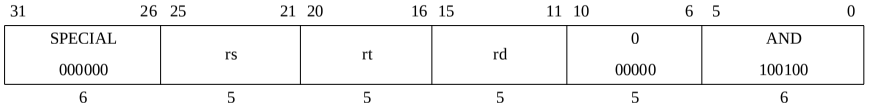
Integer Overflow

Programming Notes:

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

And

AND



Format: `AND rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical AND

Description: `GPR[rd] ← GPR[rs] AND GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

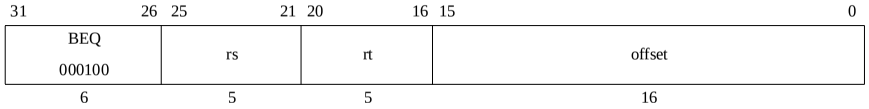
```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

Branch on Equal

BEQ



Format: `BEQ rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if GPR[rs] = GPR[rt] then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:      target_offset ← sign_extend(offset || 0^2)
           condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Exceptions:

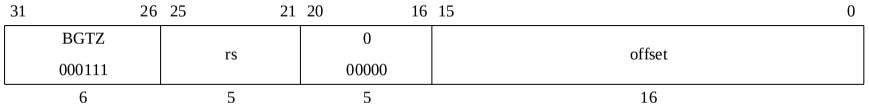
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

Branch on Greater Than Zero — BGTZ



Format: `BGTZ rs, offset` **MIPS32**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[rs] > 0 then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] > 0^GPRLEN
I+1:  if condition then
         PC ← PC + target_offset
      endif
```

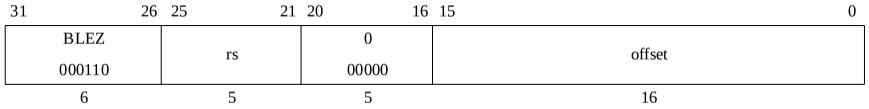
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Less Than or Equal to Zero — BLEZ



Format: `BLEZ rs, offset` **MIPS32**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[rs] ≤ 0 then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] ≤ 0^GPRLEN
I+1:  if condition then
         PC ← PC + target_offset
      endif
```

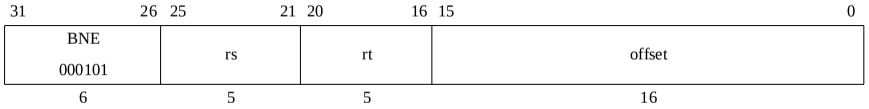
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Not Equal — BNE



Format: `BNE rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if GPR[rs] ≠ GPR[rt] then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← (GPR[rs] ≠ GPR[rt])
I+1:  if condition then
         PC ← PC + target_offset
      endif
```

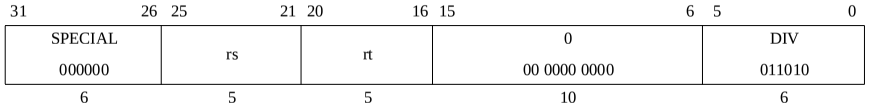
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Divide Word — DIV



Format: `DIV rs, rt` **MIPS32**

Purpose:

To divide a 32-bit signed integers

Description: `(HI, LO) ← GPR[rs] / GPR[rt]`

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder isplaced into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is **UNPREDICTABLE**.

Operation:

```
q  ← GPR[rs]31..0 div GPR[rt]31..0
LO ← q
r  ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← r
```

Exceptions:

None

Programming Notes:

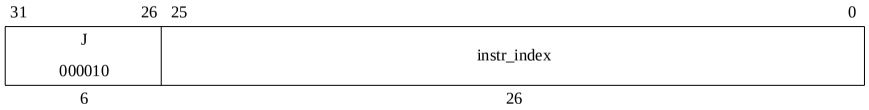
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself, or more typically within the system software; one possibility is to take a BREAK exception with a *code* field value to signal the problem to the system software.

As an example, the C programming language in a UNIX® environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if a zero is detected.

In some processors the integer divide operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

Historical Perspective:

In MIPS 1 through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.



Format: `J target` **MIPS32**

Purpose:

To branch within the current 256 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:
I+1:PC ← PC_GPRLEN-1..28 || instr_index || 0^2
```

Exceptions:

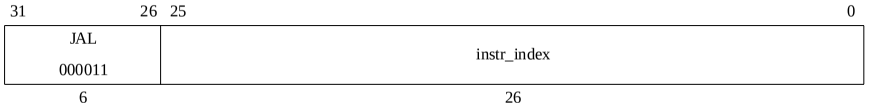
None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

Jump and Link JAL



Format: `JAL target` **MIPS32**

Purpose:

To execute a procedure call within the current 256 MB-aligned region

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:   GPR[31]← PC + 8
I+1:PC     ← PC_GPRLEN-1..28 || instr_index || 0^2
```

Exceptions:

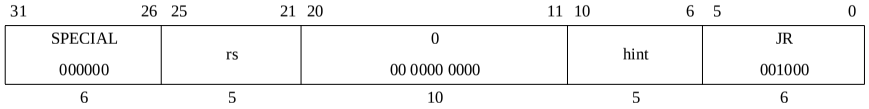
None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

Jump Register JR



Format: `JR rs` **MIPS32**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rs]`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16e ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

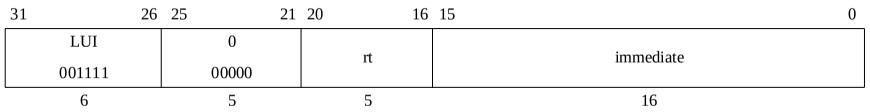
```
I:   temp ← GPR[rs]
I+1:if Config1_CA = 0 then
       PC ← temp
    else
       PC ← temp_GPRLEN-1..1 || 0
       ISAMode ← temp_0
    endif
```

Exceptions:

None

Programming Notes:

Software should use the value 31 for the *rs* field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.



Format: `LUI rt, immediate` **MIPS32**

Purpose:

To load a constant into the upper half of a word

Description: `GPR[rt] ← immediate || 0^16`

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR *rt*.

Restrictions:

None

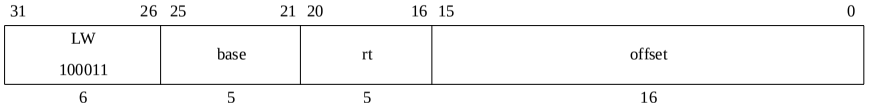
Operation:

```
GPR[rt] ← immediate || 0^16
```

Exceptions:

None

Load Word LW



Format: `LW rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

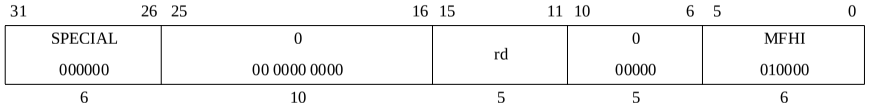
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Move From HI Register MFHI



Format: `MFHI rd` **MIPS32**

Purpose:

To copy the special purpose *HI* register to a GPR

Description: `GPR[rd] ← HI`

The contents of special register *HI* are loaded into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← HI
```

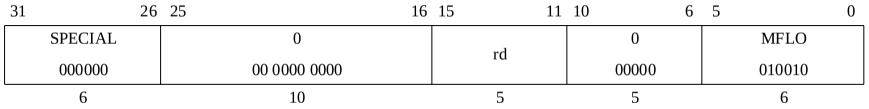
Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not moodify the HI register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.

Move From LO Register — MFLO



Format: `MFLO rd` **MIPS32**

Purpose:

To copy the special purpose *LO* register to a GPR

Description: `GPR[rd] ← LO`

The contents of special register *LO* are loaded into GPR *rd*.

Restrictions: None

Operation:

```
GPR[rd] ← LO
```

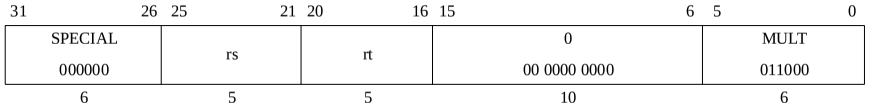
Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not moodify the HI register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.

Multiply Word — MULT



Format: `MULT rs, rt` **MIPS32**

Purpose:

To multiply 32-bit signed integers

Description: `(HI, LO) ← GPR[rs] × GPR[rt]`

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is splaced into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

```
prod   ← GPR[rs]31..0 × GPR[rt]31..0
LO     ← prod31..0
HI     ← prod63..32
```

Exceptions:

None

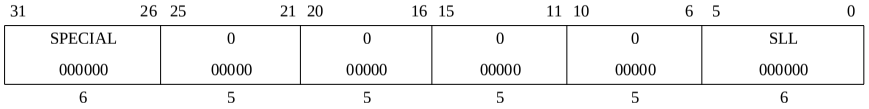
Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

No Operation NOP



Format: `NOP` **Assembly Idiom**

Purpose:

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

Restrictions:

None

Operation:

None

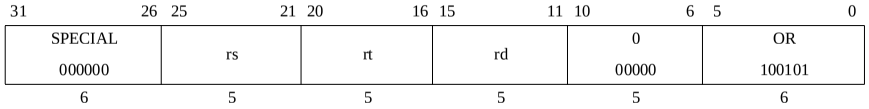
Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.

Or OR



Format: `OR rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical OR

Description: `GPR[rd] ← GPR[rs] or GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

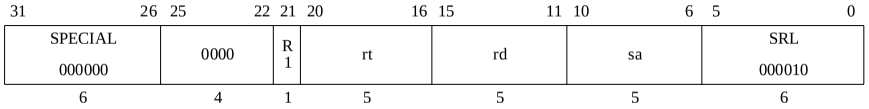
Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Rotate Word Right — ROTR



Format: `ROTR rd, rt, sa` **SmartMIPS Crypto, MIPS32 Release 2**

Purpose:

To execute a logical right-rotate of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[rt] ↔(right) sa`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is placed in GPR *rd*. The bit-rotate amount is specified by *sa*.

Restrictions:

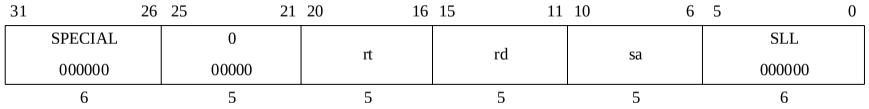
Operation:

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s      ← sa
temp   ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd]← temp
```

Exceptions:

Reserved Instruction

Shift Word Left Logical — SLL



Format: `SLL rd, rt, sa` **MIPS32**

Purpose:

To left-shift a word by a fixed number of bits

Description: `GPR[rd] ← GPR[rt] << sa`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s      ← sa
temp   ← GPR[rt](31-s)..0 || 0s
GPR[rd]← temp
```

Exceptions:

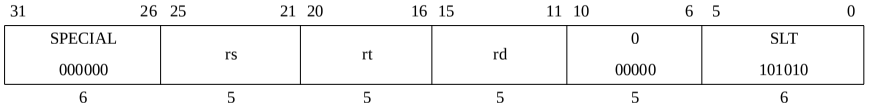
None

Programming Notes:

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

Set on Less Than SLT



Format: `SLT rd, rs, rt` **MIPS32**

Purpose:

To record the result of a less-than comparison

Description: `GPR[rd] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

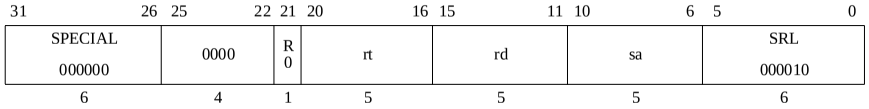
Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Shift Word Right Logical SRL



Format: `SRL rd, rt, sa` **MIPS32**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[rt] >> sa (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

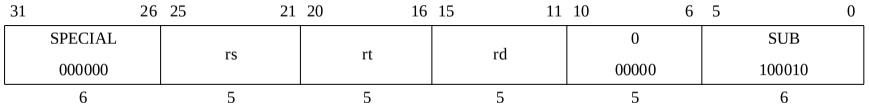
Operation:

```
s      ← sa
temp   ← 0^s || GPR[rt]_31..s
GPR[rd]← temp
```

Exceptions:

None

Subtract Word — SUB



Format: `SUB rd, rs, rt` **MIPS32**

Purpose:

To subtract 32-bit integers. If overflow occurs, then trap

Description: `GPR[rd] ← GPR[rs] - GPR[rt]`

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

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] ← temp31..0
endif
```

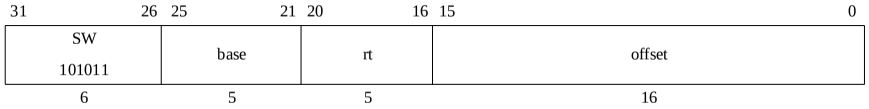
Exceptions:

Integer Overflow

Programming Notes:

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

Store Word — SW



Format: `SW rt, offset(base)` **MIPS32**

Purpose:

To store a word to memory

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

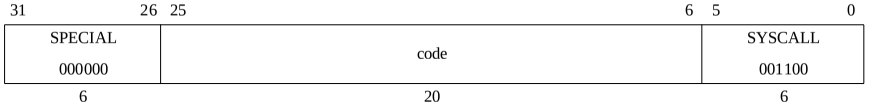
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

System Call SYSCALL



Format: `SYSCALL` **MIPS32**

Purpose:

To cause a System Call exception

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

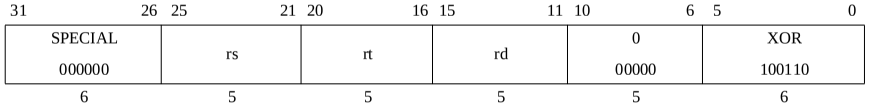
Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

Exclusive OR XOR



Format: `XOR rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical Exclusive OR

Description: `GPR[rd] ← GPR[rs] XOR GPR[rt]`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None