Projet d'Informatique

PHELMA 2e année

Année universitaire 2017–2018

Assembleur MIPS

Équipe pédagogique :
Nicolas Castagné, *François* Cayre, *Michel* Desvignes, *Kattel* Morin-Allory, *François* Portet, *Jérémy* Riffet

Résumé

La compilation est une notion fondamentale de l'informatique qui consiste à concevoir des méthodes permettant de transformer des programmes écrits en langages de haut-niveau en code compréhensible directement par la machine cible. La compilation couvre les domaines des langages de programmation, de l'algorithmique, du génie logiciel et de l'architecture des calculateurs. Les techniques employées dans la réalisation d'un compilateur sont si générales qu'on les retrouve régulièrement dans la conception d'autres programmes.

Ce projet a pour but de réaliser un compilateur de programmes écrits en langage assembleur MIPS. Ce programme, que l'on appelle *Assembleur* et qui est beaucoup plus simple qu'un compilateur de langage de haut-niveau, sera écrit en langage C. Durant la réalisation de cet assembleur vous aborderez les notions d'analyse lexicale, d'analyse syntaxique, de gestion des erreurs, de fichiers objets et bien d'autres qui vous permettront d'enrichir votre culture générale en informatique et votre savoir faire en programmation C.

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet informatique est de concevoir puis mettre en œuvre en langage C, un assembleur pour un microprocesseur MIPS.

Le rôle d'un assembleur est de transformer un programme écrit dans un langage informatique accessible à l'homme, **ici le langage assembleur**, en un programme décrivant la même série d'instructions mais cette fois-ci en **langage machine**, c'est-à-dire représenté en code binaire, directement compréhensible par le processeur. Dans notre cas le processeur sera un processeur MIPS 32 bits.

Le logiciel doit donc prendre en entrée un fichier texte contenant un programme écrit dans le langage d'assemblage de la machine MIPS, et produire plusieurs sorties :

— une liste d'assemblage, c'est-à-dire une vision textuelle du programme assemblé ;
— un fichier objet binaire contenant la traduction en langage machine du programme assemblé ;
— un fichier objet au format ELF.

Pour ce projet nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions du MIPS.

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. Les chapitres 3, 4 et 5 présentent quelques notions sur la manière de programmer un assembleur, les considérations importantes pour la mise en œuvre, puis des informations sur l'organisation générale du projet.

Les intérêts pédagogiques de ce projet informatique sont multiples. Il permet tout d'abord de travailler sur un projet de taille importante sous tous ses aspects techniques (analyse d'un problème, conception puis mise en œuvre 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'améliorer votre connaissance et 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, débogueur, etc.). Enfin, il illustre et met en pratique les connaissances relatives aux microprocesseurs, vues notamment dans le cours d'ordinateurs et microprocesseurs de première année et dans certains cours d'architecture ou de micro-électronique.

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 ont notamment été utilisés dans de nombreuses stations de travail et consoles de jeux (Silicon Graphics, Nintendo 64, Sony PlayStation 2...) mais sont maintenant surtout utilisés dans les systèmes embarqués (imprimantes, les routeurs, automobile ...).

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 dans le cas où une instruction dépend de la précédente. Une des difficultés majeures de l'assembleur est de permettre de décharger le programmeur de ces contraintes et de les gérer lors de la compilation. C'est un problème que nous n'aborderons pas mais qu'il est nécessaire de connaître pour interpréter les résultats des assembleurs actuels.

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) :

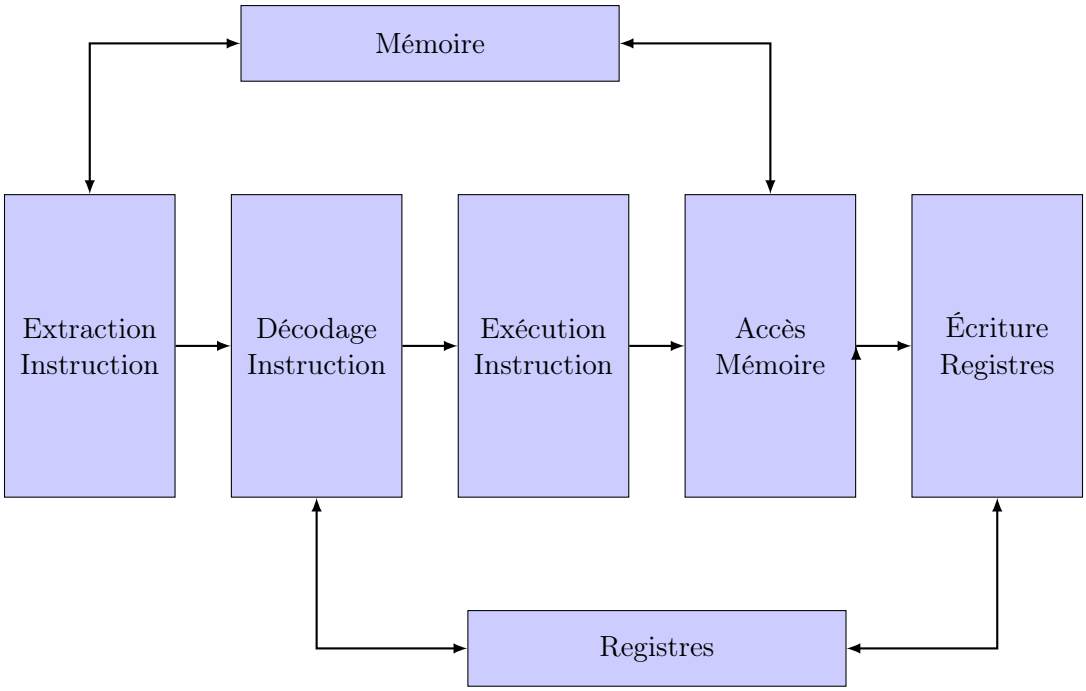


FIGURE 2.1 – Architecture interne simplifiée du microprocesseur RISC 32 bits

- 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 que les processeurs Motorola et certains ARM.
- les systèmes de type *little endian* écrivent l'octet de poids le plus faible à l'adresse la plus basse. Les processeurs Intel et AMD 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) de 32 bits chacun, dénotés `$0` à `$31`. Les registres peuvent également être identifiés par un mnemonic indiquant leur usage conventionel. 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 certains 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` pour les exceptions, registres de valeurs flottantes, ...).

2.1.3 Les Instructions

Bien entendu, comme tout microprocesseur qui se respecte, le MIPS possède une large gamme d'instructions, plus de 280. Les spécifications des instructions étudiées dans ce projet sont données dans l'annexe B. 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]. Dans ce projet nous ne prendrons en compte qu'un nombre restreint d'instructions simples.

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. Toutes les instructions sont codées sur 32bits.

Format: `ADD rd, rs, rt`

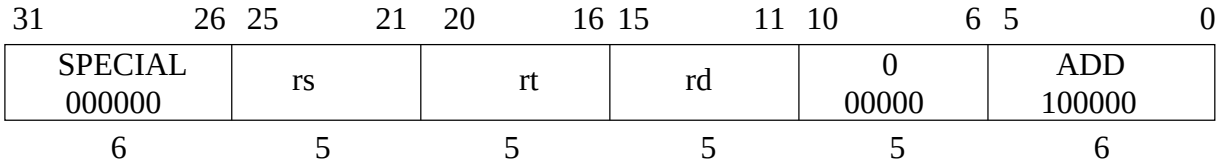


FIGURE 2.4 – Instruction ADD

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.
```

No comment, 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 (sans espace) 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.

Exemple

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

1. En réalité une étiquette peut contenir également les caractères : «.»,«_»,«$». Mais pour simplifier la conception de l'assembleur, seul «_» sera accepté.

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**
`.text`	Ce qui suit doit aller dans le segment TEXT
`.data`	Ce qui suit doit aller dans le segment DATA
`.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
`.asciiz s1, ..., sn`	Met les n chaînes de caractères à la suite en mémoire. Chaque chaîne est terminée par $\backslash 0$.
`.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), trois directives existent en 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` permet de déclarer les données non initialisées. Ces données ne prennent ainsi pas de place dans le fichier binaire du programme. Elles ne seront effectivement allouées qu'au moment du chargement du programme où 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.

Remarque : Dans ce projet, les instructions doivent absolument se trouver dans une section `.text`, la section `.data` ne peut contenir que des directives de données et la section `.bss` uniquement la directive `.space`. Tout écart de ces contraintes entraînera une erreur et fera sortir du programme d'assemblage. Notez que ces contraintes ne sont pas toutes présentes dans la norme (elles dépendent de l'assembleur).

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 normalement dans une section de données `.bss`. Mais `.space` peut également se trouver dans une section data ou l'espace sera effectivement réservé sur le disque.

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 est à l'adresse Tabb de la mémoire, l'autre à l'adresse Tabb+1.

```
Tabb:   .byte -4
        .byte 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. Par exemple, la ligne suivante permet de réserver un mot de 32 bits avec la valeur initiale 32767 à l'adresse Tabw de la mémoire. La deuxième ligne permet de stoker l'adresse du tableau `Tabw` en mémoire. Cette valeur ne sera déterminée que lors du chargement du programme en mémoire.

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

[*étiquette*] `.asciiz` ***valeur*** *valeur* doit commencer et terminer par un ". Par exemple, la ligne suivante permet de réserver 19 octets pour une chaîne de 18 caractères « il a dit "bonjour"». Notez le caractère \ d'échappement qui permet d'inclure le " sans que l'assembleur ne l'interprète comme une frontière de chaîne.

```
Tabc:   .asciiz "il a dit \"bonjour\""
```

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) 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 faibles 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. Cette adresse 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 :

```
   # si $2==$3, branchement à l'adresse PC + 0x3F4
BEQ $2, $3, 0x3F4       # code binaire correspondant ->104300FD
```

2.3 Instructions étudiées dans le projet

Cette section présente les instructions et les pseudo-instructions du MIPS qui devront être traitées par l'assembleur. 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

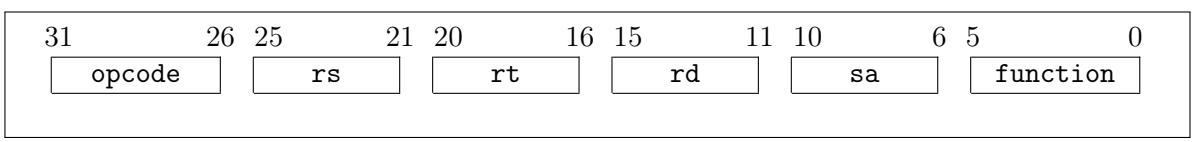
Les processeurs MIPS possèdent des instructions simples de taille constante égale à 32 bits [2]. Ceci facilite notamment les étapes d'extraction et de décodage des instructions, réalisées chacune dans le pipeline en un cycle d'horloge. Les instructions sont toujours codées sur des adresses alignées sur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



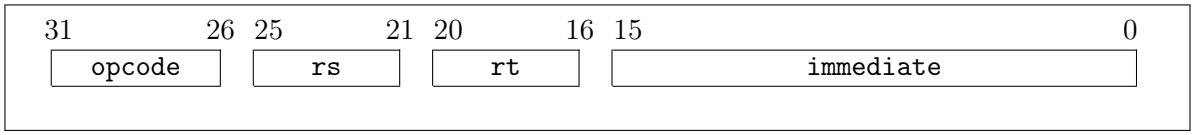
avec les champs suivants :

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

- 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 registre destination (valeur sur 5 bits, donc comprise entre 0 et 31, codant le numéro du registre)
- **rs** est le premier argument source
- **rt** est le second argument source
- **sa (shift amount)** est le nombre de bits de décalage, pour les instructions de décalage.
- **function** 6 bits additionnels pour le code des instructions R-type, en plus du champ **opcode**.

Les instructions de type I

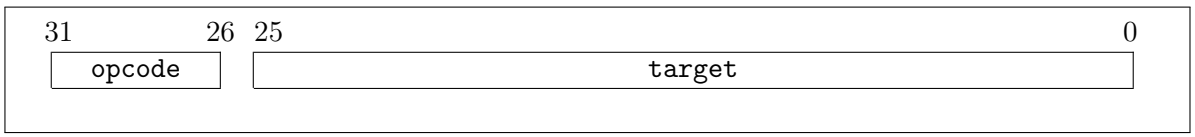
Le codage binaire des instructions *I-type*, pour "immediate type", suit le format :



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
ADD	$rd, $rs, $rt	$rd = $rs + $rt
ADDI	$rt, $rs, immediate	$rt = $rs + immediate
SUB	$rd, $rs, $rt	$rd = $rs - $rt
MULT	$rs, $rt	(HI, LO) = $rs * $rt
DIV	$rs, $rt	LO = $rs div $rt; HI = $rs mod $rt

`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 rt est placée 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
AND	$rd, $rs, $rt	$rd = $rs AND $rt
OR	$rd, $rs, $rt	$rd = $rs OR $rt
XOR	$rd, $rs, $rt	$rd = $rs XOR $rt

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
ROTR	$rd, $rt, sa	$rd = $rt[sa-0] $rt[31-sa]
SLL	$rd, $rs, sa	$rd = $rt « sa
SRL	$rd, $rs, sa	$rd = $rt » sa

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
SLT	$rd, $rs, $rt	if $rs < $rt then $rd = 1, else $rd = 0

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

Les instructions Load/Store

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

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

— `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.
— `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.
— `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.
— `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.
— `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.

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:  1010110101010010101010001100
```

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 + 1010110101010010101010001100 = 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.

Les pseudo-instructions

Les pseudo-instructions ne sont pas définies parmi les instructions du processeur MIPS, mais uniquement au niveau de l'assembleur. Ces pseudo instructions permettent d'augmenter l'expressivité du langage afin de faciliter le travail du programmeur. La conséquence est une plus grande complexité de compilation. Lors de la compilation, l'assembleur doit les remplacer automatiquement par un équivalent (pas forcément unique) composé d'une ou plusieurs instructions en langage machine.

Mnemonic	Opérandes	Opération équivalente
`NOP`		`SLL $0, $0, 0`
`MOVE`	`$rt, $rs`	`ADD $rt, $rs, $zero`
`NEG`	`$rt, $rs`	`SUB $rt, $zero, $rs`
`LI`	`$rt, immediate`	`ADDI $rt, $zero, immediate`
`BLT`	`$rt, $rs, target`	`SLT $1, $rt, $rs` `BNE $1, $zero, target`

— `NOP` n'effectue aucun traitement, seul le compteur programme est incrémenté.
— `MOVE` copie le contenu du registre `$rs` dans le registre destination `$rt`.
— `NEG` copie l'opposé du contenu du registre `$rs` (-`$rs`) dans le registre `$rt`.
— `LI` place la valeur immédiate dans le registre destination `$rt`
— `BLT` permet un branchement suite à la comparaison directe de deux registres, alors qu'on ne peut comparer qu'à zéro avec les instructions. Si le contenu du registre `$rs` est inférieur à celui du registre `$rt`, le branchement vers `target` est effectué après l'instruction.

Chapitre 3

Programmer un assembleur

Le but de ce projet est de concevoir un compilateur de programmes écrits en assembleur. Ce type de compilateur est appelé *assembleur* (confusion entre le langage et le compilateur) et reste beaucoup plus simple à concevoir qu'un compilateur pour langage de haut niveau. Cependant, les principes généraux exposés dans cette section peuvent s'appliquer à tout type de compilateurs (p.ex. compilateur C, Pascal, etc.). Cette section donne quelques règles simples pour développer proprement un assembleur et quelques éléments de réflexion sur la façon de procéder.

3.1 Compilateur

D'une manière générale, un compilateur est un logiciel qui lit un programme écrit dans un premier langage — le langage *source* — et le traduit en un programme équivalent dans un autre langage — le langage *cible*. Dans notre cas, le langage source est l'assembleur MIPS et le langage cible est le code binaire MIPS. La transformation d'un programme source vers un programme cible nécessite plusieurs phases qui sont décrites figure 3.1.

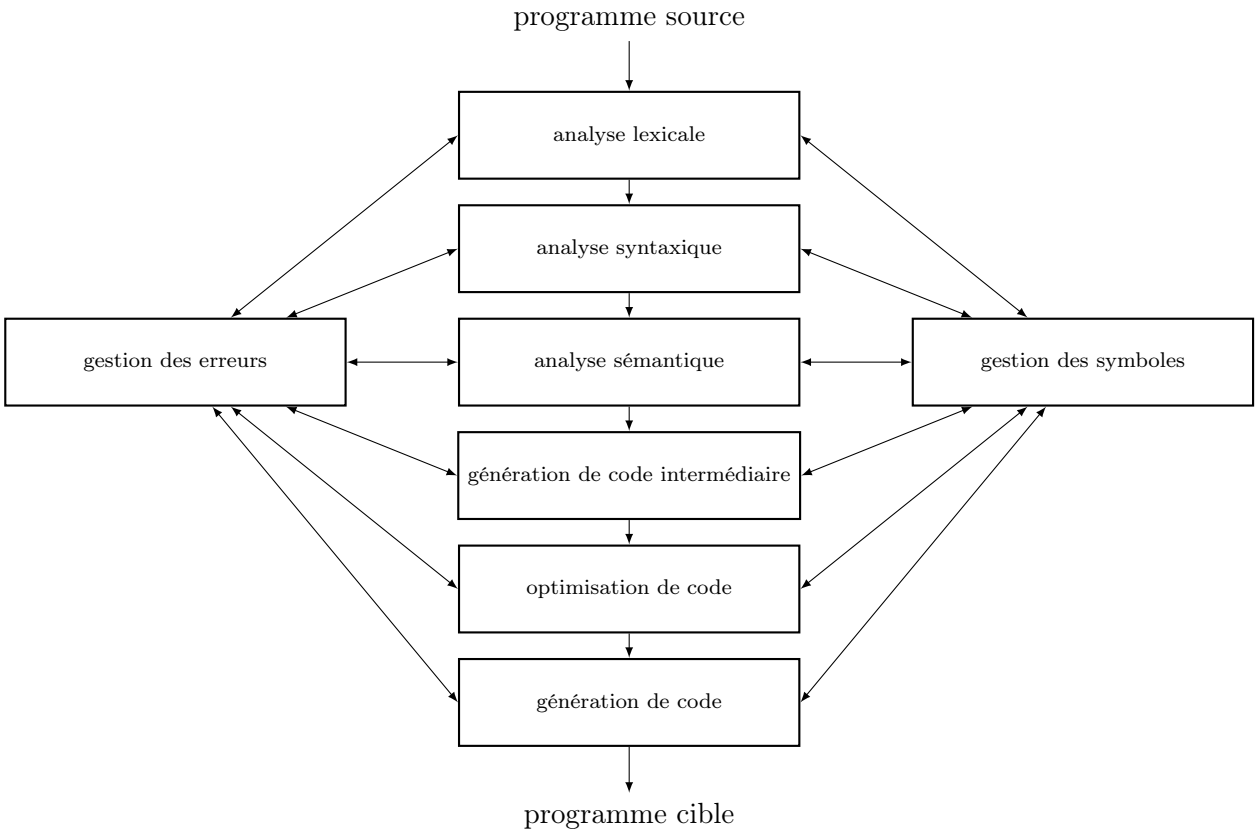


FIGURE 3.1 – Phases courantes d'un compilateur

Toutes les phases sont susceptibles de repérer des erreurs qui doivent être rapportées à l'utilisateur directement ou par un fichier de log. De plus, le compilateur doit maintenir une *liste des*

symboles qui enregistre les identifiants (p.ex. : nom de procédure, étiquette, constante, etc.) et complète l'information au fur et à mesure de l'avancement dans la compilation. Par exemple, dans le code suivant, l'adresse de l'étiquette `etiq` ne peut être connue qu'à la prochaine ligne. Le compilateur lit donc l'instruction `J etiq`, stocke `etiq` comme un nouveau symbole et complète les informations sur ce symbole plus tard.

```
        J etiq
etiq:   .byte -4
        ADD $2, $3, $4
```

En résumé et pour citer J. S. Rohl, l'écriture d'un compilateur c'est définir des structures de données qui doivent être maintenues par le compilateur et les procédures par lesquelles ces structures sont créées et transformées.

Remarque : la compilation n'est pas concernée par *l'exécution* du programme (sauf dans quelques cas précis d'optimisation). Si le code source contient une addition, le compilateur se contente de traduire cette instruction en langage cible et certainement PAS de réaliser cette addition. Autrement dit, le compilateur doit pouvoir compiler n'importe quel code source qui respecte le langage même s'il est complément farfelu ou erroné à l'exécution (p.e., division par zéro).

La première phase du compilateur est l'*analyse lexicale*, c'est-à-dire extraire les mots du code source et vérifier qu'ils appartiennent bien au langage.

3.2 Analyse lexicale

L'analyse lexicographique peut se définir comme l'analyse des mots (ou «lexèmes») contenus dans un langage. Dans notre cas, il s'agit d'étudier les lexèmes contenus dans le programme source en langage assembleur. Les lexèmes lus peuvent être de nature différente : des mnémoniques, des noms de registres, des nombres, des étiquettes, etc. À titre d'exemple, on montre le découpage du petit programme assembleur suivant en lexème :

```
# Un commentaire...
etiq:  .byte - 4
     ADD $2,$3,$4
     J 0xABCD
```

Le résultat de l'analyse peut se présenter sous la forme suivante (`NL` signifie Nouvelle Ligne codée par le caractère '$\backslash n$' en langage C) :

```
[COMMENT      ]   Un commentaire...
[NL           ]   \n
[SYMBOLE      ]   etiq
[DEUX_PTS     ]   :
[DIRECTIVE    ]   .byte
[VAL_DECIMAL  ]   -4
[NL           ]   \n
[SYMBOLE      ]   ADD
[REGISTRE     ]   $2
[VIRGULE      ]   ,
[REGISTRE     ]   $3
[VIRGULE      ]   ,
[REGISTRE     ]   $4
[NL           ]   \n
[SYMBOLE      ]   J
[VAL_HEXA     ]   0xABCD
```

Cette opération se réalise simplement en mettant en oeuvre un automate à états finis comme celui de la figure 3.2. À chaque étape (décodage d'un lexème), on part de l'état initial `Init` et on est amené vers les différents états de l'automate suivant le prochain caractère rencontré. On boucle alors sur l'état à chaque nouveau caractère jusqu'à ce que ce caractère ne soit pas permis par l'état. On est alors amené vers l'état terminal `Term` où l'on identifie et stocke le lexème avant de retourner à l'état `Init`.

Le schéma présenté est donné à titre d'exemple, il doit être adapté à vos besoins !

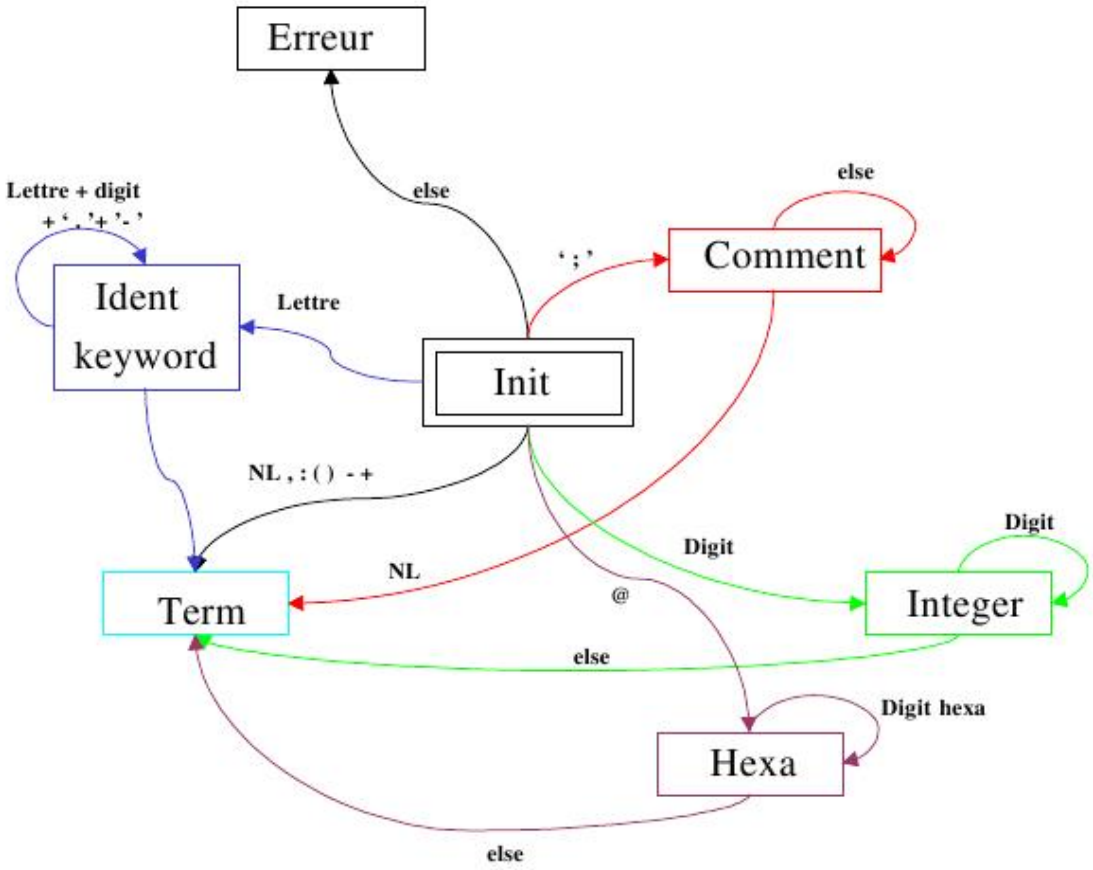


FIGURE 3.2 – Exemple d'automate à états finis

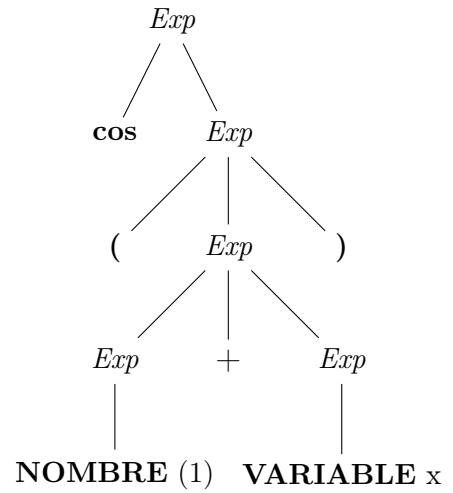
Identifier des lexèmes nécessite de trouver les frontières de ceux-ci, or le langage laisse une grande liberté dans l'utilisation des séparateurs. Ainsi : `etiq : .byte - 4` et `etiq:.byte-4` représentent le même code mais ne sont pas écrits de la même manière. Afin de faciliter l'analyse lexicale les lignes de code à traiter sont préalablement formatées afin que les lexèmes soient plus facile à traiter. Ainsi la ligne `etiq:.byte-4` peut être transformée en `etiq : .byte -4`.

3.3 Analyse grammaticale

L'analyse grammaticale vérifie que les lexèmes extraits lors de l'analyse lexicale forment une séquence valide par rapport au langage assembleur MIPS, autrement dit, que le code respecte la

$Exp \rightarrow$ NOMBRE
$Exp \rightarrow$ VARIABLE
$Exp \rightarrow$ cos Exp
$Exp \rightarrow Exp + Exp$
$Exp \rightarrow$ (Exp)
...

(a) Grammaire réduite des expressions arithmétiques.



(b) Arbre d'analyse de *cos(1+x)*.

FIGURE 3.3 – Exemple de grammaire et d'arbre d'analyse de *cos(1+x)*.

syntaxe MIPS.

Le rôle de l'analyseur syntaxique est de vérifier la conformité syntaxique et de produire l'arbre d'analyse qui sera par la suite exploitable informatiquement. En général, on explicite la syntaxe d'un langage de programmation par une grammaire (d'où le nom d'analyse grammaticale). Une grammaire est composée de

— symboles terminaux (les lexèmes) dans un alphabet A,
— de variables (symboles intermédiaires) qui appartiennent à l'alphabet V,
— règles de grammaire $v \rightarrow w$, où $v \in V$ et $w \in A \cup V$

La figure 3.3.a donne un exemple de grammaire pour l'analyse d'expressions arithmétiques. Les termes en italique sont les variables tandis que les autres sont les symboles terminaux.

L'analyse proprement dite consiste à vérifier si chaque lexème d'entrée peut être associé à un symbole terminal en respectant le chemin de production induit par la grammaire. Cette analyse est soit faite de manière descendante (c'est-à-dire trouver le chemin en partant de la racine qui peut expliquer la séquence de lexèmes d'entrées) ou ascendante (trouver le chemin qui mène à la racine à partir des lexèmes). Le chemin ainsi trouvé permet de constituer l'arbre d'analyse. La figure 3.3.b montre l'arbre d'analyse de l'expression $cos(1 + x)$ ou chaque symbole terminal est affiché en gras. Bien entendu la performance des analyseurs dépend de la complexité des grammaires qui peuvent comprendre un certain nombre d'ambiguïtés, de récursions ou de dépendances contextuelles (p.ex : le symbole '*' en C). L'analyse grammaticale est un ancien et vaste domaine de l'informatique que nous n'aurons pas le loisir d'aborder dans ce projet. Sachez seulement que ce domaine a fourni des outils d'analyse syntaxique tels que Yacc [1] ou Cup [2] pour faciliter l'analyse de la plupart des langages informatiques.

Cependant, la grammaire correspondant au langage MIPS est tellement simple qu'elle nous permettra de nous passer d'outils et de créer notre propre analyseur linéaire spécifique pour le langage MIPS. Cette grammaire peut être recomposée à partir de la syntaxe des différentes instructions, avec leurs opérandes et modes d'adressages décrits dans l'annexe B.

1. http://dinosaur.compilertools.net/
2. http://www2.cs.tum.edu/projects/cup/

3.4 Analyse sémantique

L'analyse sémantique se concentre sur la vérification du sens du programme. Sa tâche concerne notamment la vérification de type de variables dans des expressions (p. ex. : opérateur appliqué à un opérande incompatible), le contrôle d'unicité (p. ex. : un terme déclaré plusieurs fois) ou encore la vérification de la cohérence du programme (p.ex., utilisation d'une variable non initialisée). Le langage assembleur étudié ici étant très simple cette analyse prendra une place relativement peu importante (p.ex. : pas de contrôle de flot d'exécution par exemple). Par conséquent, cette analyse ne nécessitera pas forcément un module indépendant mais pourra être effectuée dans les autres modules (analyse grammaticale et génération de code). Par exemple, la détection d'utilisation multiple d'un nom pour une étiquette pourra être mise en œuvre au sein de la lecture des symboles.

3.5 Code intermédiaire et optimisation/modification de code

L'optimisation de code consiste à modifier le programme source pour le rendre plus efficace ou moins consommateur en ressource. Les compilateurs, tels que GCC, sont capables de modifier un programme pour les rendre plus compact (gain de mémoire) et plus efficace en termes de rapidité d'exécution et d'accès mémoire. Il existe une trop grande variété de types d'optimisation pour les décrire tous en détail. Un exemple d'optimisation classique est le remplacement de toutes les opérations de division ou multiplication par une puissance de deux par des décalages binaires.

Un autre exemple d'optimisation consiste à réécrire une suite non optimale ou inutile d'opérations en une suite d'opérations plus efficace. Dans l'exemple suivant, la deuxième instruction est inutile car $t1 et $t2 sont déjà égaux. Notez cependant que si une étiquette était présente en face de la deuxième instruction celle-ci ne devrait pas être supprimée car elle pourrait être utilisée dans une boucle. De même, la troisième et la quatrième instructions n'ont aucun effet (pas de modification de valeur et écriture dans `$zero`) et peuvent être éliminées.

Exemple :

```
MOVE $t1, $t2
MOVE $t2, $t1
ADD $t1,$t1,$0
ADDI $zero,$t1, 120
```

Parmi la multitude d'optimisations possibles, une technique simple d'amélioration du code est l'optimisation à lucarne (*peephole optimization*). Celle-ci consiste à améliorer une courte séquence d'instructions (appelée lucarne) en une séquence plus efficace. Une des caractéristiques de l'optimisation à lucarne est que chaque amélioration est susceptible d'engendrer d'autres optimisations. Cela implique que plusieurs passes sont nécessaires pour atteindre un code optimal. L'optimisation à lucarne est particulièrement adaptée à l'élimination de code mort (instructions ne pouvant jamais être exécutées), aux simplifications algébriques (une opération logique est souvent plus rapide qu'une multiplication) et au remplacement de tous motifs de code prédéfini (p. ex. : les deux `MOVE` de l'exemple ci-dessus). Nous verrons que cette dernière capacité pourra être mise à profit dans la gestion des pseudo-instructions.

La génération de code intermédiaire est présente dans la plupart des compilateurs. Cette étape permet une grande ré-utilisabilité du compilateur pour la génération de code sur différentes cibles car, dans ce cas, seul l'étape de génération de code final est à réécrire. C'est aussi pour cette raison que la plupart des optimisations sont effectuées sur le code intermédiaire qui sert de pivot entre

le langage source et le langage cible. Dans le cadre de l'assembleur, l'étape de génération de code intermédiaire est souvent absente étant donné sa grande simplicité. En effet, certains compilateurs, tel que le GCC de GNU, utilisent même un langage assembleur comme code intermédiaire. Par conséquent, nous n'utiliserons pas de code intermédiaire et travaillerons directement sur le code assembleur d'entrée pour générer le code de sortie.

3.6 Génération du code binaire

Durant les phases d'analyse lexicale et grammaticale, l'information doit être stockée dans une structure afin de l'exploiter ultérieurement pour la génération du code binaire. En effet, à partir de cette structure, il devient possible de coder les instructions les unes après les autres et de déterminer les adresses correspondantes. Par exemple, dans le cas d'instructions dont les opérandes sont des registres, des valeurs immédiates ou des adresses absolues, les instructions peuvent être codées directement, indépendamment du reste du programme.

Par contre, lorsqu'un opérande est une étiquette, i.e. l'adresse d'une autre instruction ou d'une donnée, les choses se compliquent un peu. En fait, il serait nécessaire de calculer l'adresse effective de l'opérande, puis de l'intégrer dans le codage binaire des instructions du programme. Or ceci n'est pas possible car on ne connaît pas, **au moment de l'assemblage**, les adresses mémoires auxquelles seront implantées les différentes sections `.text`, `.data` et `.bss`. En effet, celles-ci ne seront connues qu'au moment du chargement du programme où le *chargeur* devra *loger* le programme en mémoire. Les assembleurs génèrent donc des instructions *incomplètes* avec les informations permettant au chargeur de les compléter. Le code devient ainsi *relogeable*.

3.6.1 Notion de fichier relogeable

Reprenons les principes permettant de trouver les adresses effectives des opérandes des instructions. Il est possible de programmer avec des *adresses absolues* : utiliser l'opérande à l'adresse 0x1000A, effectuer un branchement vers l'adresse 0xB600, etc. Ce mode de programmation est assez limité, et ce pour au moins deux raisons :

- Il faut que ces adresses existent (mémoire suffisante), et qu'elles soient accessibles (certaines plages sont réservées par le micro-contrôleur). Ce qui est vrai sur une machine ne l'est pas forcément sur une autre, or il est souhaitable que le programme puisse être utilisé indépendamment de la configuration spécifique d'une machine.
- Il faut également que, au moment où l'exécution du programme démarre, ces adresses ne soit pas déjà utilisées par le système d'exploitation ou une autre application. Si les variables d'un processus A sont modifiées par un processus B sans que cela soit attendu, il est peu probable que l'exécution des deux processus se déroule normalement...

Une autre solution est de programmer avec des *adresses relatives à l'instruction courante*, c'est-à-dire la valeur du compteur programme `PC` à un instant donné. Dans ce cas, un branchement est vu comme "avancer/reculer `PC` de `x` octets", la valeur d'un opérande est à situer `y` octets plus loin que l'instruction...Ceci fonctionne bien, mais est limité quand la taille des programmes augmente, et ne permet pas d'utiliser des opérandes situées dans une autre section, par exemple la section `.data`, puisque son adresse d'implantation n'est pas connue au moment de l'assemblage.

La solution retenue est donc de créer un fichier objet **relogeable**, c'est-à-dire qui ne contient pas d'adresses effectives finales mais simplement toutes les informations nécessaires pour recalculer ces adresses au moment où le programme sera chargé (logé) en mémoire.

Ce principe est utilisé pour l'assemblage de programme à un seul fichier, mais peut être généralisé 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. Par exemple, un 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.

Prenons l'exemple de la figure 3.4. Dans cet exemple le symbole `boucle` est défini dans la section `.text`. Or, pour les raisons évoquées précédemment, cette adresse n'est pas encore connue au moment où le programme est assemblé. Le principe est donc de coder les adresses **relativement au début de la section à laquelle elles appartiennent**. En effet, même si on ne connaît pas l'adresse finale de `boucle` en mémoire, on sait la situer par rapport au début de la section `.text` à laquelle elle appartient. Dans cet exemple, `boucle` est situé 4 octets après le début de la section `.text` car il est le deuxième mot déclaré. Lorsque l'adresse de la section `.text` sera connue il suffira d'ajouter cette valeur au 4 qui est déjà connu. De la même manière le code `.word boucle` de la section data va devoir être relogé pour que l'espace réservé contienne l'adresse de `boucle` lorsque le programme sera chargé.

```
.text
     ADDI $3 , $0 , $1
boucle:
     ADDI $3 , $0 , 12345
     J boucle
.data
    .word boucle
    .word 0x12345
```

FIGURE 3.4 – Extrait de code nécessitant une relocation

La partie codant l'opérande destination de l'instruction `J` ne contiendra donc pas directement l'adresse effective de l'opérande, mais les informations permettant de la calculer. Ces informations, dites de **relocation**, consistent en

— l'adresse relative du code à reloger par rapport au début de sa section
— un type de relocation
— le symbole (dans la table des symboles) auquel est lié le mot

Ainsi le programme de la Figure 3.4 générera les deux entrées de relocation suivantes :

```
rel.text
00000008 R_MIPS_26 : .text:00000004   boucle
rel.data
00000000 R_MIPS_32 : .text:00000004   boucle
```

Pour la section `.text`, à l'adresse 8 se trouve un mot à reloger. La relocation est de type `R_MIPS_26` car pour les instructions de type J et JAL l'opérande est sur 26 bits. Le symbole `boucle` est celui qui est visé par l'instruction. Cependant, le symbole étant local (c'est-à-dire dans le fichier assemblé) l'instruction a pu coder dans l'opérande l'adresse du symbole relative à la section qui est ici 4. C'est ce que l'on appelle le *Addend*. Ainsi, la seule valeur nécessaire pour résoudre la relocation est celle de l'adresse d'implantation du début de la section `.text`. Pour information, le code de l'instruction dans le fichier binaire est `08000001` où `08` est le code de J et $000001 << 2 = 4$.

Pour la section `.data`, à l'adresse 0 se trouve un mot à reloger. La relocation est de type `R_MIPS_32`. Le symbole `boucle` est à nouveau celui qui est visé par l'instruction. Cependant, le symbole étant local (c'est-à-dire dans le fichier assemblé) l'instruction a pu coder l'adresse du symbole relative à la section qui est ici 4 (le *Addend*). Ainsi la seule valeur nécessaire pour résoudre la relocation est celle de l'adresse d'implantation du début de la section `.text`. Pour information, le code du mot dans la section data est `00000004`.

Si nous détaillons 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 nous devons d'abord préciser les notations 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 (dans les exemples précédents il s'agit de 8 et 0).

S désigne l'adresse du **S**ymbole par rapport auquel on reloge (ici les sections .text et .data).

A désigne le **A**ddend à ajouter pour calculer la valeur du champ à reloger (c'est la valeur du champ avant relogement, ici il s'agit de 1 et 4).

AHL désigne un autre type de valeur à **a**jouter qui est calculée à partir de la valeur provisoire A[3].

Les adresses finales des sections `.text`, `.data`, `.bss` sont normalement déterminées lors du chargement. Les principaux modes de calcul dépendent du type de relocation et 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.

Lors de l'exécution du programme, celui-ci sera d'abord placé dans la mémoire par un *chargeur*. C'est lui qui, en fonction de la configuration de la machine (capacité de mémoire) et surtout de son état actuel (où y a-t-il de la place disponible en mémoire pour loger le code du programme ?), déterminera où vont être placées les différentes sections composant le programme. Ce n'est qu'après cette étape que les adresses des opérandes seront connues de manière absolue. La partie adresse effective des instructions sera donc mise à jour afin que ces dernières accèdent correctement aux opérandes. Cette mise à jour sera faite grâce aux informations de relocations.

Par exemple, si le programme de la Figure 3.4 est chargé en mémoire avec la section `.text` à l'adresse 0AF08 et `.data` à l'adresse BCDE0 les deux entrées de relocation permettront de remplacer le code des mots suivants

— `.word boucle` → `00000004` → relocation par rapport à .text → on a S=0AF08, A = 00000004 en appliquant la formule on obtient la nouvelle valeur V = (4 + 0AF08) = 0AF0C ce qui conduit bien à la valeur de boucle. Le code devient `0000AF0C`.

— `J boucle` → `08000001` → relocation par rapport à .text → on a S=0AF08, P = 08, A = 000001 en appliquant la formule on obtient la nouvelle valeur V = ((4 | 0) + 0AF08)>>2 = 2BC3 ce qui conduit bien à la valeur de boucle (2BC3<<2 = 0AF0C). Le code devient `08002BC3`.

Dans le projet, vous devrez générer les informations de relocation pour les 4 types de relocations présentées ci-dessus.

Le format `ELF`

Ce principe de fichier relogeable se retrouve notamment dans le format de fichier `ELF` (Executable and Linkable Format). Ce format est celui des fichiers objets et exécutables dans nombreux systèmes d'exploitation, dont Linux, SunOs, MacOs X, etc. 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.

Un fichier `ELF` commence par une *en-tête* donnant les caractéristiques générales du fichier, la version, la machine, etc. Il contient ensuite un ensemble de sections, notamment une table des symboles, la section `.text` des instructions, la section `.data` des données, la section `.bss` et les sections `.rel.text` et `.rel.data` qui contiennent des informations de relocation. Elles seront utilisées pendant l'édition de liens ou le chargement en mémoire dans le cas d'un fichier exécutable.

Vous trouverez plus d'informations sur le format `ELF` et la relocation dans [4]. Et si vous voulez des exemples, rien de plus simple : pour programmer votre assembleur en langage C, vous allez générer un ensemble de fichiers objets `.o` ainsi qu'un exécutable, votre programme `as-mips`. Ils seront tous au format `ELF` ![4]

4. Pour étudier leur contenu, utilisez la commande `readelf`

Chapitre 4

À propos de la mise en œuvre

4.1 Quelques conseils sur la programmation

4.1.1 Programmation incrémentale

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 d'éviter les erreurs (ou les résoudre facilement) et de gagner en efficacité. 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. On pourra utilement s'inspirer de la programmation incrémentale qui consiste principalement à :

- 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 [13] ! (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*[1] qui fourmille de conseils intéressants.

4.1.2 Se familiariser avec le domaine d'application

Réalisez des expériences pour mieux appréhender les différents aspects du projet et préparer des fichiers de tests que vous utiliserez pour corriger et valider votre programme.

- Écrivez des programmes en assembleur, qui vous permettront de mieux appréhender ce langage. Vous pouvez commencer par quelques instructions puis compliquer la chose en gérant des données, des branchements (instructions conditionnelles), un appel à une procédure, un tableau, ...
- Constituez-vous une base de programmes tests en langage assembleur couvrant les différentes instructions, les modes d'adressage, etc.
- Ces fichiers pourront servir de tests pendant le projet pour déboguer vos programmes.
- Déterminez manuellement, à partir des spécifications, le codage de quelques instructions.

1. http://www.extremeprogramming.org

— Vous pouvez ensuite jouez avec les différents outils fournis pour compiler et exécuter du code assembleur (cf. annexe A). Toujours instructif !

4.1.3 Conception et développement

Après avoir pris connaissance du langage assembleur MIPS, vous pouvez vous lancer dans la conception du programme : identification des différentes tâches, choix des structures de données intermédiaires, découpage modulaire du code (quels fichiers ? quelles fonctions ? quelles entrées-sorties ?), etc.

Par exemple, sous quelle forme vont être représentés les lexèmes (quelle structure de données) ? Quelles sont les fonctionnalités nécessaires pour réaliser l'analyse syntaxique ?

Il faut prévoir une décomposition du développement de manière à pouvoir tester et corriger le programme au fur et à mesure que vous l'écrivez. Sinon, vous risquez d'avoir un programme très difficile à corriger ou vous risquez de devoir réécrire de grandes portions de code. La programmation modulaire permet en outre d'avoir un code plus concis et donc plus facile à déboguer.

Programmez de manière **défensive**. Pour les cas que votre programme ne devrait jamais rencontrer générez un message compréhensible du type `Erreur Ouverture Fichier : fonction analyse lexicale` et arrêtez proprement le programme, afin de pouvoir déboguer plus facilement. Placez des traces d'exécutions dans vos programmes de manière à pouvoir suivre le déroulement du programme. Il est fortement recommandé de se familiariser avec l'utilisation d'un débogueur (`gdb` et `valgrind`).

Pensez à concevoir le programme de manière à pouvoir le modifier facilement. Par exemple, il peut être intéressant de prévoir une représentation des données sous forme de structure pour pouvoir ajouter des champs facilement, ou encore d'utiliser des fonctions pour isoler et réutiliser au maximum du code indépendant. De cette manière, une correction/amélioration sur une fonction générale sera bénéfique pour l'ensemble du code. Il peut aussi être intéressant de penser à rendre l'ajout d'une nouvelle instruction aisée.

De manière générale, on code d'abord les cas les plus généraux et les plus simples, avant de coder les cas particuliers et compliqués. Gardez-comme ligne directrice d'avoir le plus tôt possible **un programme qui fonctionne**, même s'il ne gère pas tout. Ensuite, améliorez-le au fur et à mesure.

4.1.4 Développement dirigé par les tests

Quand un programme dépasse les quelques lignes, qu'il est conçu par plusieurs personnes et qu'il a un objectif d'utilisation générale (c.-à-d., d'être utilisé par des clients), plusieurs problèmes vont devoir être résolus. Comment parvenir à faire évoluer un code de plusieurs milliers de lignes sans effets de bord ? Comment corriger un bug sur le code de Dupond qui est parti en weekend prolongé à Ibiza ?

Mise à part une gestion rigoureuse, un bon moyen de s'assurer, dès la conception, du bon fonctionnement d'un bout de code est de prévoir des tests. Dans un programme informatique, chaque fonction prend en entrée un certain nombre de paramètres, effectue un calcul et renvoie une valeur et peut modifier les paramètres d'entrées. Durant la conception de cette fonction (avant le codage), le programmeur a en tête quelques scénarios d'utilisation. Le développement dirigé par les tests consiste à utiliser ces scénarios pour *tester* la validité de la fonction. Un *test*, consiste donc à fournir des valeurs en entrée à une fonction et à vérifier que le résultat est bien celui attendu. L'extrait de code ci-dessous fournir un exemple de test.

Dans ce test, écrit avant le codage de la fonction, le programmeur est sûr de ne pas avoir oublié de prendre en compte les cas particuliers (le zéro) et les cas d'erreurs (nombres négatifs). Sans ces

```
/* test factorielle */
#include "factoriel.h"  /* prototype de la fonction factorielle:
                           int fact(int)*/

void main(){
        /* test d'un cas isole*/
        if(fact(3)==6) printf("1-OK\n") else printf("1-KO\n")

        /* test du zero*/
        if(fact(0)==1) printf("2-OK\n") else printf("2-KO\n")

        /* test des nombres negatifs : renvoie -1 lorsque erreur*/
        if(fact(-2)==-1) printf("3-OK\n") else printf("3-KO\n")

}
```

FIGURE 4.1 – Extrait de code illustrant le test de la fonction factorielle

tests, il y aurait eu de fortes chances pour que ces cas aient été oubliés lors du codage [2] ce qui aurait pu avoir des conséquences sur le reste du programme (par exemple beaucoup de temps perdu pour retrouver l'origine d'un bug). Par ailleurs, si la fonction doit être réécrite — d'impératif en récursif par exemple — les tests n'ont pas besoin d'êtres modifiés, ils sont réutilisables à l'infini.

4.2 Quelques conseils sur le projet assembleur

4.2.1 Réflexions sur le format des données

La compilation se compose de plusieurs phases de traitement bien définies (cf. figure 3.1) qui vont produire de l'information nouvelle (p.ex., le code final) ou enrichir de l'information produite par les traitements précédents (p.ex., ajouter l'adresse à une instruction traitée par l'analyse syntaxique). Le choix de représentation des structures de données doit donc se tourner tout naturellement vers des solutions permettant de relier ensemble les informations à propos d'une même instruction. Pour cela, les structures sont un choix tout indiqué. Par ailleurs, les contraintes de temps liées au projet (livrables à rendre fréquemment) vont faire qu'il ne sera pas possible de prévoir d'avance toutes les subtilités du programme. Ainsi, il est fort possible que les structures de données soient amenées à être modifiées au cours du projet. Là aussi, les structures sont un choix pertinent. En effet, il est très facile d'ajouter un champ à une structure avec un minimum d'impact sur le code original. Par exemple, la fonction `void affiche_personne(char *nom, int age, int poids)` sera beaucoup plus dur à modifier que `void affiche_personne(Personne p)` (ou `Personne` est une structure ayant les champs `nom, age` et `poids`) si on décide après coup d'afficher aussi le numéro de sécurité sociale. Dans le premier cas, il faudra modifier tous les endroits ou la fonction est utilisée en ajoutant un argument alors que dans le deuxième cas il sera simplement nécessaire d'ajouter un champ dans la structure et de modifier le code dans la fonction `affiche_personne`. D'une manière générale, essayez de toujours concevoir un code facile à faire évoluer.

2. si, si, souvenez-vous de votre première année...

Les différentes phases de la compilation vont toutes manipuler le programme original mais à un niveau différent. L'analyse lexicale va manipuler des lexèmes, l'analyse syntaxique des motifs syntaxiques, la génération de code des instructions et la plupart de ces phases vont devoir gérer les symboles contenus dans le code source. Concernant l'analyse lexicale, le nombre de lexèmes n'étant pas connus à l'avance on pourra éviter l'utilisation d'un tableau en se référant au type abstrait vu en première année (liste, pile, file). L'analyse syntaxique est typiquement conçue comme manipulant les instructions sous forme d'arbres syntaxiques auxquels sont appliqués des transformations et ajoutés des informations au fur et à mesure des traitements (on parle d'arbres décorés). Cependant, la syntaxe de l'assembleur est suffisamment simple est contrainte pour utiliser une approche moins élaborée, plus proche de vos connaissances en programmation et algorithmique. En effet, les motifs des instructions sont de trois types (cf. 2.3.1) avec un nombre fixe d'opérandes. Une instruction pourra donc être représentée par une liste composée des chacun des éléments (étiquette, opérateur, opérandes...) voire une structure. L'ensemble des instructions pourra être représenté soit par une liste soit par un tableau (car on connaît à ce stade le nombre de lignes du code — utile pour la génération des messages d'erreur). Enfin, il conviendra de prendre en compte la table des symboles qui pourrait être codée par une table de hachage. Quel que soit le mode de représentation choisi, il sera important d'écrire les fonctions qui permettront l'accès, la modification, la création, l'affichage et la destruction de ces données[3].

4.2.2 Machine à états finis (*FSM*)

Une machine à états finis (ou FSM pour *finite state machine* en anglais) décrit les états dans lesquels un automate est autorisé à être. Une FSM décrit aussi quoi faire lorsque l'on atteint un état donné, ou même ce qu'îl faut faire lorsque l'on passe de tel à tel autre état. Pour les différentes analyses, nous allons être amené à construire des types d'automate. Par exemple, la première phase de l'assembleur est de vérifier que le code donné en entrée contient uniquement des éléments acceptés par le langage et de les identifier. Il faut notamment pouvoir reconnaître que la chaîne de caractères `0123` est une valeur octale et que `0x123` est une valeur hexadécimale, que `ADDI` est un symbole et que `.text` est une directive. 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 trop 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 assembleur 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 n'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 FSM. 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 4.2 montre comment on peut représenter un automate faisant la différence entre nombres décimaux, octaux ou hexadécimaux.

3. Ainsi que les tests, bien entendu...

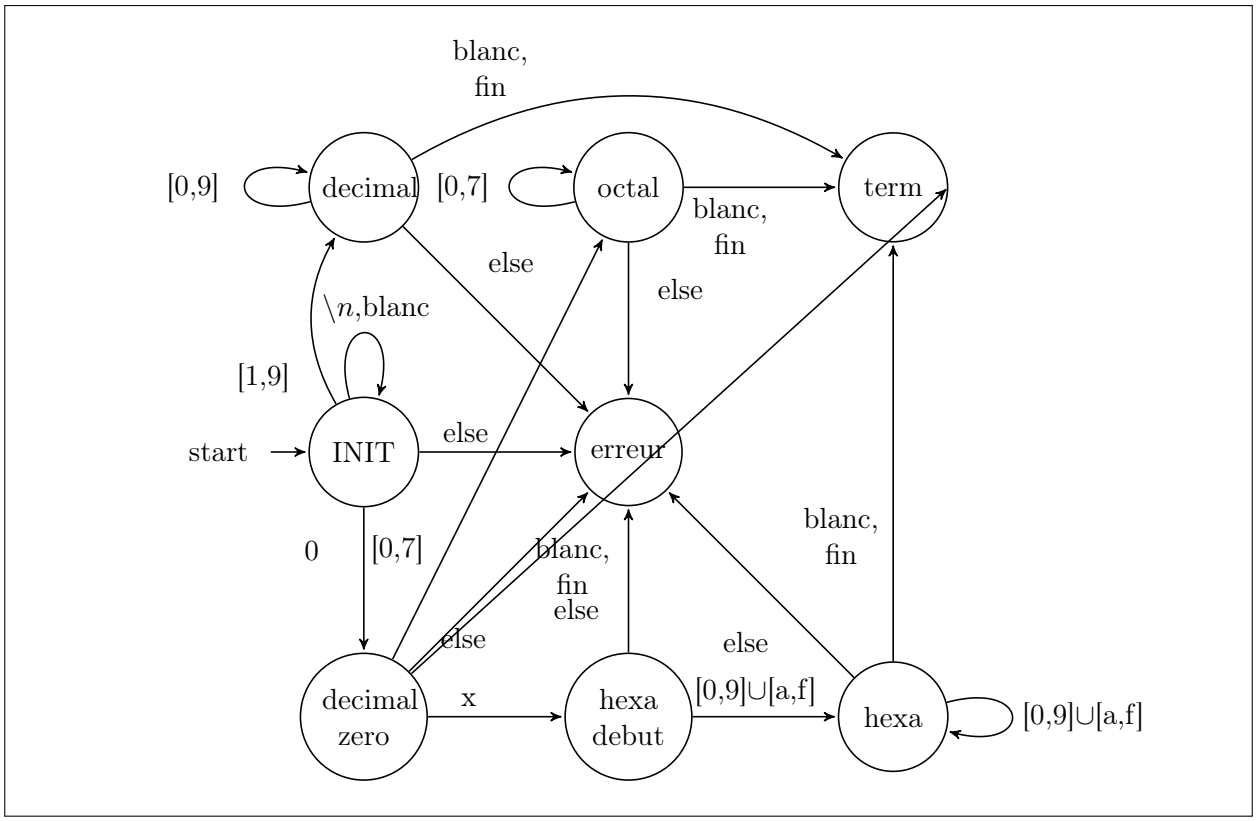


FIGURE 4.2 – Exemple d'automate faisant la différence entre une valeur décimale, octale et hexadécimale

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 4.3 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 général sera bien plus conséquent et il sera plus judicieux de travailler les chaînes au niveau des tokens. La démarche est d'étudier les différents éléments du langage, d'identifier leurs 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.

4.2.3 Découper une chaîne de caractères en *token*, `strtok` pour la représentation des instructions MIPS

Votre assembleur doit vérifier un programme d'entrée et traduire chaque instruction en code machine. Cependant, où cette connaissance sera-t-elle stockée ? Comment la représenter ? Comment y accéder ?

```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 4.3 – Exemple de traduction en C de l'automate de la figure 4.2

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.

```
/* 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␣\t␣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 4.4 – Extrait de code illustrant l'usage de `strstok()`

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`.

Chapitre 5

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.

5.1 Objectif général :

À la fin de ce projet vous devrez avoir réalisé un assembleur qui prend en entrée un fichier source `file.s` et génère trois fichiers :

1. une liste d'assemblage : `file.l`
2. un fichier objet binaire : `file.obj`
3. un fichier objet au format ELF : `file.o`

```
# allons au ru
.set noreorder
.text
    Lw $t0 , lunchtime
    LW $6, -0x200($7)
    ADDI $t1,$zero,8

boucle:
    BEQ $t0 , $t1 , byebye
    NOP
    addi $t1 , $t1 , 1
    J boucle
    NOP
byebye:
    JAL viteviteauru

.data
lunchtime:
    .word 12
    .word menu

.bss
menu:
    .space 24
```

FIGURE 5.1 – Exemple de fichier assembleur `miam.s`

L'assembleur sera appelé en tapant sous Linux la commande :

```
as-mips source_filename [-elf]
```

où `source_filename` est le nom du fichier texte contenant le programme à assembler. L'option `-elf` sert à indiquer que l'on souhaite générer un fichier ELF relogeable

5.1.1 Liste d'assemblage

La liste d'assemblage devra contenir pour chaque ligne du fichier source une ligne incluant :
— le numéro en décimal de la ligne du fichier source
— l'adresse en hexadécimal de l'instruction ou de la donnée
— le codage en hexadécimal de l'instruction ou de la donnée
— le contenu de la ligne source
— sur la ligne suivante, les erreurs éventuellement détectées !
En C ce type de ligne peut être affiché par la commande
`fprintf(fp,"%3u␣%08X␣%08X␣%s\n",nl,address,code,source_line)`.

Après le programme, la liste d'assemblage contiendra une table des symboles définis dans le programme, triée par ligne de définition, avec pour chacun d'eux :
— le numéro de la ligne où il est défini
— la section à laquelle il appartient (`.text`, `.data` ou `.bss`)
— le déplacement du symbole dans sa section
— son adresse relative en mémoire
En C ce type de ligne peut être affiché par la commande
`fprintf(fp,"%3d\t.%-4s:%08X\t%s\n",nl,section,address,sym_name)`.

Enfin, la liste d'assemblage contiendra les entrées de relocation pour la section `.text` puis la section `.data` :
— le décalage par rapport au début de la section
— le type de relocation
— le symbole à partir duquel reloger avec sa section, son adresse relative et son nom
En C ce type de ligne peut être affiché par la commande
`fprintf(fp,"%08x\t%s\t.%-4s:%08x\t%s\n",address,type_reloc,sym_section,sym_address,sym_name)`.

Remarque : le respect de ce format est impératif !

5.1.2 Fichier objet binaire

Le fichier binaire sera composé du codage des sections `.text` puis `.data` puis `.bss`. Les deux premières sections contiendront les informations suivantes :
— la taille des informations à implanter (nombre d'octets), en entier sur 4 octets
— les informations elles-mêmes, c'est-à-dire la traduction binaire des instructions ou des données.
L'information concernant la section `.bss` ne contiendra que la taille de celle-ci. La taille sera codée sur un mot de 32 bits.

```
 1
 2                    # allons au ru
 3
 4
 5                    .set noreorder
 6                    .text
 7 00000000 3C010000      Lw $t0 , lunchtime
 7 00000004 8C280000
 8 00000008 8CE6FE00      LW $6, -0x200($7)
 9 0000000C 20090008      ADDI $t1,$zero,8
10
11                    boucle:
12 00000010 11090004      BEQ $t0 , $t1 , byebye
13 00000014 00000000      NOP
14 00000018 21290001      addi $t1 , $t1 , 1
15 0000001C 08000004      J boucle
16 00000020 00000000      NOP
17                    byebye:
18 00000024 0C000000      JAL viteviteauru
19
20                    .data
21 00000000 0000000C lunchtime: .word 12
22 00000004 00000000 .word menu
23
24                    .bss
25 00000000 000...    menu:.space 24

.symtab
 11      .text:00000010      boucle
 17      .text:00000024      byebye
 18      [UNDEFINED]         viteviteauru
 21      .data:00000000      lunchtime
 25      .bss :00000000      menu

rel.text
00000000      R_MIPS_HI16      .data:00000000      lunchtime
00000004      R_MIPS_LO16      .data:00000000      lunchtime
0000001c      R_MIPS_26        .text:00000010      boucle
00000024      R_MIPS_26        [UNDEFINED]         viteviteauru

rel.data
00000004      R_MIPS_32        .bss :00000000      menu
```

FIGURE 5.2 – Liste d'assemblage correspondant au code de la figure 5.1

Pour valider votre fichier binaire, vous pouvez regarder le contenu à l'aide d'un éditeur de fichier binaire, ou en utilisant la commande `od -t xC` qui affiche le contenu binaire d'un fichier, octet par octet.

5.1.3 Fichier objet au format ELF

Le format ELF pour *Executable and Linkable Format* est un format de fichiers objets, bibliothèques et exécutables, utilisé dans de nombreux systèmes d'exploitations (Linux, Solaris, BSD, ...). Les fichiers ELF sont composés d'un ensemble de sections (éventuellement vides) :

- Entête
- 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 librairies C standards (libelf, gelf) permettent la lecture et l'écriture d'un fichier ELF.

5.2 Étapes de développement du programme

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

5.2.1 [5 pts] Étape 1 : Analyse lexicale

But

À la fin de cette étape, vous devrez avoir validé l'analyse lexicale.

Marche à suivre

1. Définition des catégories de lexèmes et du format interne
2. Écriture des tests
3. Canonisation du fichier assembleur
4. Écriture de l'automate à états finis
5. Génération des traces
6. Gestion des erreurs

5.2.2 [5 pts] Étape 2 : Analyse syntaxique – 1

But

À la fin de cette étape, vous devrez avoir validé le chargement des instructions et le décodage des directives.

Marche à suivre

1. Définition des catégories d'instructions, des types d'adressage et du format interne
2. Définition des structures de représentation des instructions
3. Écriture des tests
4. Chargement du dictionnaire d'instructions
5. Définition de la table des symboles
6. Décodage des instructions
7. Décodage des directives
8. Décodage des définitions de symbole
9. Vérification des instructions (nombre opérandes)
10. Génération des traces
11. Gestion des erreurs

5.2.3 [5 pts] Étape 3 : Analyse syntaxique – 2

But

À la fin de cette étape, vous devrez avoir validé la vérification de la syntaxe des instructions et la génération des entrées de relocation.

Marche à suivre

1. Écriture des tests
2. Décodage des opérandes et des modes d'adressage
3. Génération de la table des symboles
4. Vérification des instructions
5. Vérification des directives
6. Génération des entrées de relocation
7. Mécanisme de réécriture
8. Génération des traces
9. Gestion des erreurs

5.2.4 [5 pts] Étape 4 : Génération de code

But

À la fin de cette étape, vous devrez avoir validé la génération de code.

Marche à suivre

1. Écriture des tests
2. Calcul des adresses des instructions
3. Génération de la liste d'assemblage
4. Génération du fichier objet

5. Définition et génération de la table de relocation
6. Génération du fichier ELF
7. Tests de fonctionnement de l'assembleur

5.3 Bonus : extensions du programme

Plusieurs extensions possibles du programme peuvent être envisagées, telle que la prise en compte des *float*, des variables globales, des fichiers multiples, etc. Une extension intéressante peut être d'inclure certaines pseudo-instructions (cf. 2.3.2). Toute extension menée de manière satisfaisante amènera un bonus dans la notation.

5.4 Agenda et organisation du projet

Consulter le site web du projet info : http://tdinfo.phelma.grenoble-inp.fr/2Aproj/

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

Compilation d'un programme en assembleur MIPS

Ce chapitre décrit les principales commandes dont vous aurez besoin pour assembler un programme MIPS, générer les fichiers ELF qui vous serviront de tests et les analyser.

Nous allons utiliser l'assembleur `as` de GNU, qui peut être compilé pour traiter les programmes en assembleur MIPS et générer des fichiers binaires compatibles avec un processeur MIPS, et ce même si votre machine a effectivement un autre processeur.

A.1 Installation

Par défaut, `gcc` et d'autres outils de la librairie binutils (assembleur `as`, linker `ld`, désassembleur `objdump`,...) sont compilés pour la machine et le système d'exploitation que vous utilisez. À PHELMA, il s'agit essentiellement de PC Intel Pentium ou Athlon sous Linux.

Pour notre projet, il est nécessaire de compiler ces outils pour qu'ils soient adaptés au microprocesseur MIPS. On parle de "cross-compilation". Tout a été fait à l'école, bien sûr, et il vous suffira d'utiliser les commandes `mips-gcc`, `mips-as`, `mips-ld`, `mips-readelf`, etc. `gcc` (sans le préfixe `mips-`) sera bien sûr encore disponible pour que vous puissiez développer votre simulateur en langage C !

A.2 Compilation et étude des fichiers

Soit le programme `exemple.s` présenté figure A.1, écrit en langage assembleur MIPS [1].

A.2.1 Assemblage

Pour assembler ce fichier et créer un fichier objet binaire relogeable au format ELF, vous avez deux solutions :

1. Ou bien utiliser directement l'assembleur `mips-as`

   ```
   mips-as exemple.s -o exemple.o
   ```

2. Utiliser `mips-gcc` [2] avec l'option `-c` :

   ```
   mips-gcc -c exemple.s -o exemple.o
   ```

Vérifiez, celà revient exactement au même !

A.2.2 Désassemblage

Pour étudier le contenu du fichier `exemple.o` ainsi généré, vous pouvez utiliser différents outils :

1. Au passage, vous aurez un exemple de code assembleur avec appel d'une procédure...
2. `mips-gcc` est en fait un alias de la commande `mips64-gcc -mabi=32 -mgp32 -march=R3k`
Les options permettent de spécifier que l'on souhaite utiliser une architecture 32 bits, avec des *general purpose register* de 32 bits, pour un processeur de type MIPS R3000. Aspirine ?

```
.text
.globl __start
__start:
   li $a0, 1 # fib(n): parameter n
   move $v0, $a0 # n < 2 => fib(n) = n
   blt $a0, 2, done
   li $t0, 0 # second last Fibâ number
   li $v0, 1 # last Fibâ number
   fib: add $t1, $t0, $v0 # compute next Fibâ number in sequence
   move $t0, $v0 # update second last
   move $v0, $t1 # update last
   sub $a0, $a0, 1 # more work to do?
   bgt $a0, 1, fib # yes: iterate again
   done: sw $v0, result # no: store result, done
.data
result: .word 0x11111111
```

```
## Fichier exemple.s : un programme principal, avec appel d'une procédure Min
.text
.globl __start                # Pas obligé pour vous...

__start:

##-------------------------------------------------------------------------
# Programme principal
  li   $2, 10                 # $2 <- 10
  li   $3, 0xff               # $3 <- 0xff

# appel de la procedure min
  jal  Min                    # $ra (return adress) <- pc+1; puis saut à Min
  sw   $4, 0x1000($zero)      # ecrit le resultat a l'adresse 0x1000
  j    Exit                   # saut à la fin du programme


##-------------------------------------------------------------------------
# fonction Min qui calcule le minimum de $2 et $3 et le stocke dans $4.
# On pourrait aussi utiliser la pile...
Min:
  blt   $2, $3, Then          # si $2 >= $3 saut à Then
  move  $4, $3                # $4 <- $3 (else)
  j     End                   # saut à Fin
  Then: move  $4, $2          # $4 <- $2 (then)
  End:                        # fin de la procedure min
  jr    $ra                   # saut à l'adresse contenue dans $ra,
                              # (mise à jour auto lors de l'appel avec jal)
                              # on retourne ainsi dans le programme principal

Exit:
## The End
```


FIGURE A.1 – Fichier `exemple.s` en langage assembleur MIPS.

- `od -t xC exemple.o` permet de voir le contenu du fichier binaire. Différentes options d'affichage sont possible. Ici `-t xC` affiche les valeurs octet par octet en hexadécimal. `man od` pour plus d'informations.
- `mips-objdump exemple.o` désassembleur standard.
- `mips-readelf exemple.o` permet de voir les différentes sections, symboles, etc. d'un fichier ELF.

```
mips-gcc -S toto.c -o toto.s
```

A.2.3 Edition de lien

L'édition de lien (commande `ld`) permet normalement de créer, à partir d'un ou plusieurs fichiers objets `.o`, un unique fichier binaire exécutable sur une machine MIPS (ou avec un simulateur !). Par exemple :

```
mips-ld exemple.o -o exemple.bin
```

Dans le cadre de ce projet, on ne s'intéresse pas à l'édition de liens ! On se contentera de charger des fichiers ELF relogeables.

Cependant, la commande `ld` pourra tout de même être utilisée pour lier deux fichiers objets. La commande `ld` avec l'option `-r` permet en effet de produire à partir de plusieurs fichiers relogeables, un seul fichier relogeable. Ainsi la commande suivante lie ensemble `toto1.o` et `toto2.o` dans un seul fichier relogeable `toto.o` :

```
mips-ld -r toto1.o toto2.o -o toto.o
```

Cette forme d'édition de lien, appelée édition de lien statique, recopie le code des 2 fichiers en entrée dans le fichier en sortie. Vous pouvez faire un essai, avec par exemple le code d'une procédure dans un fichier et l'appel dans un autre. De jolis tests de relocation en perspective !

Annexe B

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].

B.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

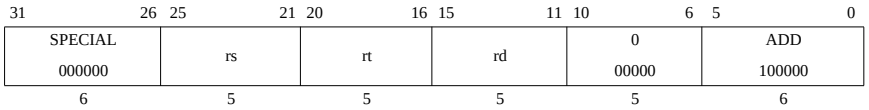
FIGURE B.1 – Octet

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 B.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
```

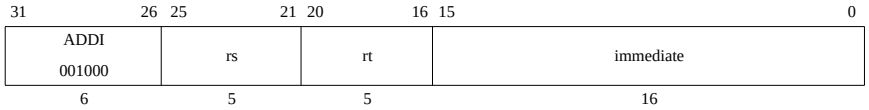
Exceptions:

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
```

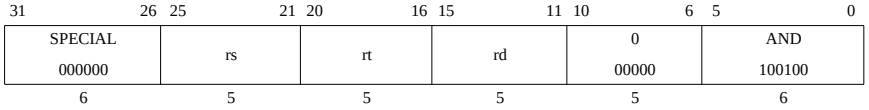
Exceptions:

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

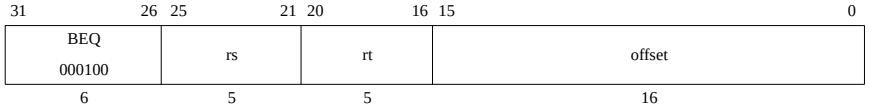
Operation:

```
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:

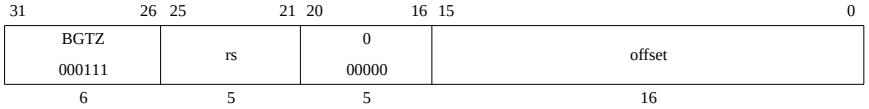
None

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
```

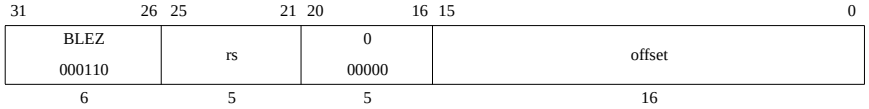
Exceptions:

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
```

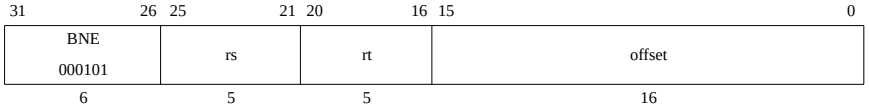
Exceptions:

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
```

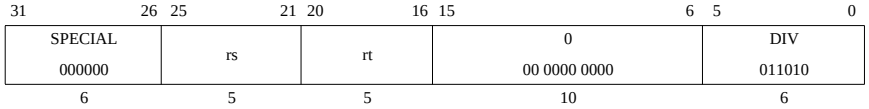
Exceptions:

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:

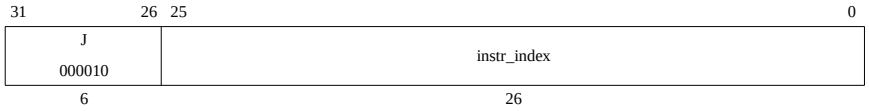
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor 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:

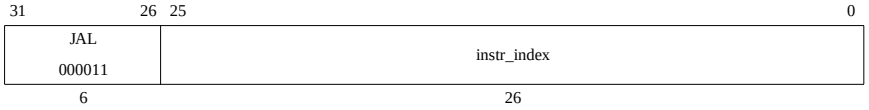
None

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:

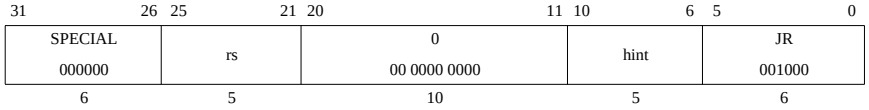
None

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:

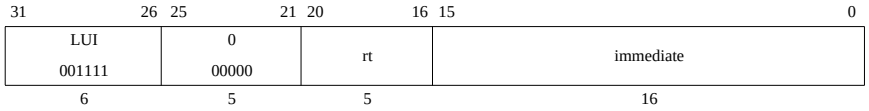
```
I:  temp ← GPR[rs]
I+1:if Config1_CA = 0 then
       PC ← temp
    else
       PC ← temp_GPRLEN-1..1 || 0
       ISAMode ← temp_0
    endif
```

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

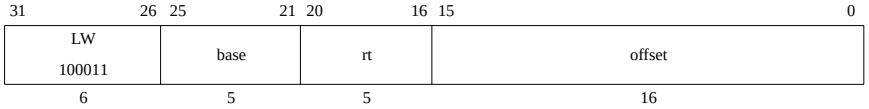
Operation:

```
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.

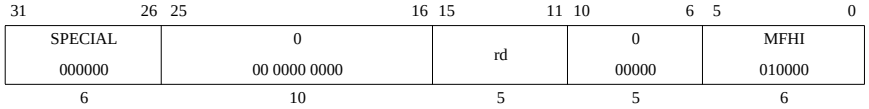
Operation:

```
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
```

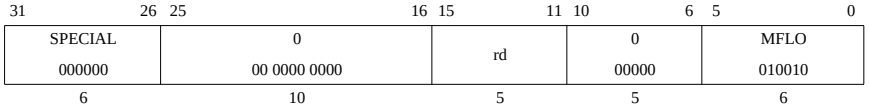
Exceptions:

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
```

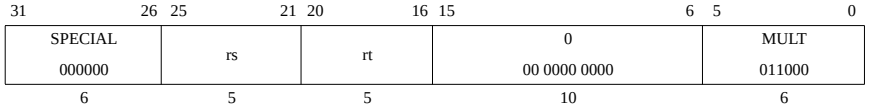
Exceptions:

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

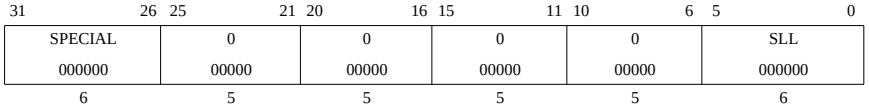
Programming Notes:

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

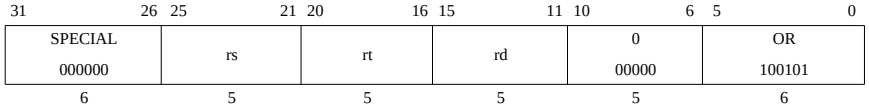
Exceptions:

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

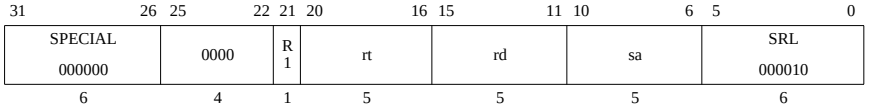
Operation:

```
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:

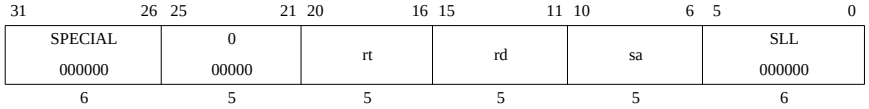
Operation:

```
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:

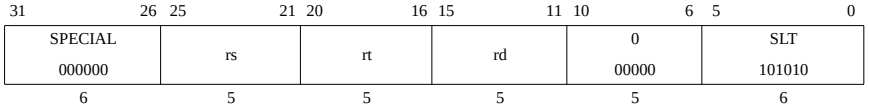
None

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

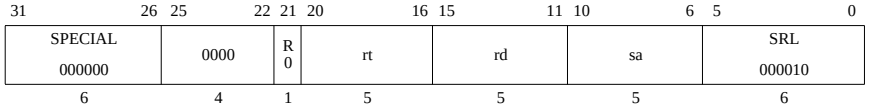
Operation:

```
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

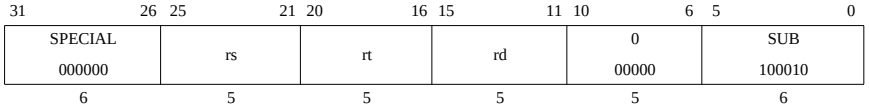
Operation:

```
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
```

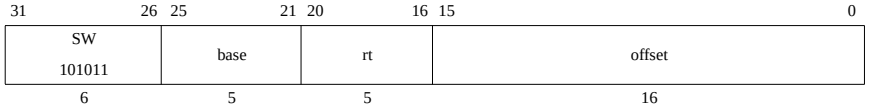
Exceptions:

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.

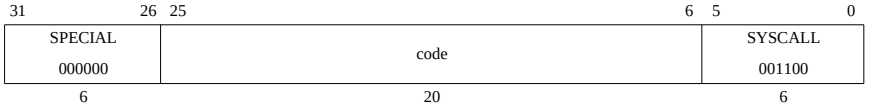
Operation:

```
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

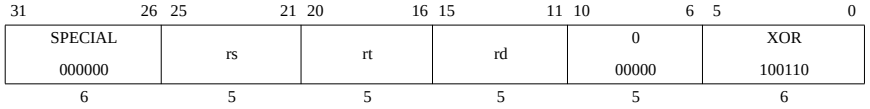
Operation:

```
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