Projet d'Informatique

PHELMA 2e année

Année universitaire 2014–2015

Émulateur MIPS

Francois.Portet@imag.fr
francois.cayre@gipsa-lab.grenoble-inp.fr

D'après
Nicolas Castagné
Matthieu Chabanas
Laurent Fesquet

Résumé

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

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

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet informatique est de réaliser, en langage C, un émulateur de microprocesseur *MIPS* permettant d'exécuter et de mettre au point des programmes écrits dans le langage binaire du *MIPS*. Le rôle d'un tel émulateur est de lire un programme donné en entrée et d'exécuter chacune des instructions avec le comportement attendu de la machine cible. Les émulateurs permettent notamment de prototyper et déboguer des programmes sans la contrainte de posséder le matériel cible (en l'occurrence, au cas où vous ne l'auriez pas compris, une machine avec un microprocesseur *MIPS*).

Plus précisément, l'émulateur que vous devrez réaliser prendra en entrée un fichier objet binaire au format *E*LF et permettra :

— de le charger en mémoire et de l'exécuter, entièrement ou pas à pas ;
— de modifier et/ou afficher son code assembleur ou la mémoire qu'il utilise pour le mettre au point ;
— de simuler des entrées et sorties basiques depuis et vers le code *MIPS*...

Votre émulateur sera évalué dans l'environnement GNU/Linux de l'école [1]. Pour ce projet, nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions de l'architecture complète *MIPS*. L'exécution de ces instructions sera émulée par des appels de fonctions du langage C dans la machine dite *hôte* [2].

Vous vous référerez à la table des matières pour une organisation générale du sujet.

L'intérêt opérationnel de ce projet est multiple : il permet tout d'abord de travailler sur un projet de taille raisonnable sous tous ses aspects techniques (analyse d'un problème, conception puis implantation d'une solution, et enfin validation du résultat), mais il permet aussi d'aborder la notion de gestion de projet (découpage du travail et respect d'un *planning*). Ce projet vous permettra également d'acquérir une certaine maîtrise du langage C, tel qu'effectivement utilisé ailleurs [3], tant pour la programmation scientifique que pour le développement industriel. Nous en profiterons pour vous présenter quelques notions importantes (*e.g.* machine à état) qui vous manqueraient encore. Vous aurez aussi à intégrer l'utilisation d'outils de développement propres à automatiser les tâches élémentaires de développement. Enfin, ce projet illustre et met en pratique des connaissances relatives aux microprocesseurs ou aux systèmes (*cf.* cours d'architecture, d'ordinateurs et microprocesseurs ou de systèmes d'exploitation),

1. Toute excuse du genre *"Mais je ne comprends pas, ça fonctionnait pourtant chez moi"* ne saurait être prise en compte pour l'évaluation de votre travail. La note minimale, à savoir zéro, vous serait alors attribuée sans autre forme de procès.

2. La machine dite *hôte*, malgré la glorieuse ambiguïté de la langue française au sujet de ce mot, est réputée être celle sur laquelle le code de la machine *MIPS* est émulé.

3. Il n'est donc pas impossible que vous ressentiez comme une légère accélération par rapport à ce que vous avez vécu en première année...

Chapitre 2

Description du MIPS

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

2.1 Exécution d'un programme sur un microprocesseur

Pour la plupart d'entre vous, la transformation du code écrit en texte (programme C) vers un fichier exécutable (suite à une compilation et édition de liens) reste encore une partie très obscure de votre cursus informatique. Pour l'instant, il suffit de rappeler que tout programme saisi dans un langage textuel doit être traduit dans le langage machine de l'ordinateur sur lequel il doit être exécuté (c.-à-d., la machine *hôte*)[1]. Dans notre cas, il s'agit du code MIPS que vous verrons plus en détails dans la section 2.5[2]. Cependant, pour que ce programme s'exécute sur le processeur, il faut qu'il soit placé en mémoire (on dit *chargé* en mémoire) afin que celui-ci puisse être 'lut' par le processeur[3].

Le Schéma de la Figure 2.1 décrit les grandes étapes de l'exécution d'un programme.

1. Lorsque le programme est 'lancé', le fichier exécutable est 'chargé' en mémoire vive par le système d'exploitation[4]. Pendant ce chargement, l'OS, trouve un espace mémoire suffisant pour stocker le programme et ses donnés auxquels il réservera l'accès exclusifs au programme qui s'exécute (c.-à-d., les programmes s'exécutant sur l'OS en même temps n'y auront pas accès)
2. le programme est exécuté proprement dit. C'est à dire que chacune des instructions MIPS est lue en mémoire et exécutée par la CPU (Central Process Unit) séquentiellement. Pour des raisons de rapidité et modularité, les opérations (arithmétique, logique, accès mémoire) sont effectuées sur des registres qui sont des zones mémoires à accès ultra-rapide contenues dans la CPU du microprocesseur. Autrement dit, les opérations ne sont (quasiment) jamais directement effectuées sur la mémoire vive mais toujours à travers des registres. Ces registres sont mis à jour à travers des instructions explicites lisant ou écrivant de/vers la mémoire de/vers les registres. Par ailleurs, pour connaître l'emplacement de la prochaine instruction à exécuter, le microprocesseur utilise un pointeur d'instruction (Programme Counter) qui est un registre mis automatiquement à l'adresse mémoire de la prochaine instruction à exécuter. Enfin, le programme peut également avoir à interagir avec le monde extérieur à travers des 'appels systèmes' (interactions avec l'OS). Le simple fait de vouloir afficher des caractères à

1. Vous aurez l'opportunité, dans ce projet, d'aller ouvrir des fichiers dans le langage machine, ce qui sera, n'en doutons pas, une expérience inoubliable voire même une révélation...
2. Bien entendu un programme compilé sur/pour un système d'exploitation et un processeur particuliers n'est pas exécutable sur un OS ou processeur différent.
3. Pour un nombre conséquent de bonnes raisons que vous découvrirez en cours d'OS, le processeur ne lit pas le programme à partir du disque dur
4. Vous noterez que ce système d'exploitation est lui-même exécuté par le microprocesseur, mais nous ne discuterons pas cet aspect que vous aurez le plaisir de découvrir en cours d'OS

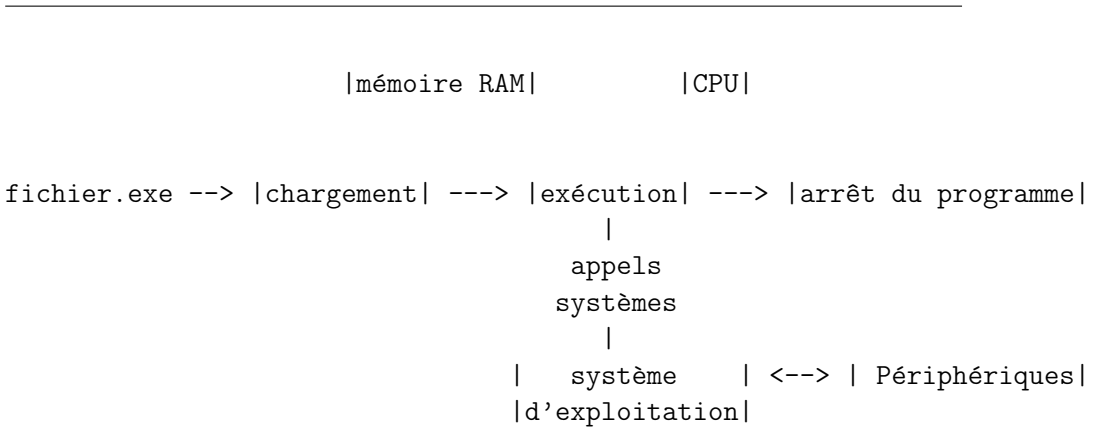


Figure 2.1 – vision simplifiée de l'exécution d'un programme sur un microprocesseur

l'écran nécessite de faire appel à l'OS qui gère les périphériques d'E/S tels que l'écran ou le clavier[5].

3. Une fois le programme terminé (arrivé en fin d'exécution, interrompu par l'OS, etc) l'OS libère la mémoire vive et toutes autres ressources allouées au programme.

Tous ces éléments (code binaire, mémoire, registres) impliqués dans l'exécution d'un programme MIPS sont l'objet du reste de ce chapitre. Ce sont, entre autre, ces éléments qu'il va falloir représenter dans le projet.

2.2 Format des fichiers binaires utilisés dans ce projet (ELF)

Il ne vous aura pas échappé qu'une image n'est pas un son. De la même manière, un fichier MP3 n'est pas non plus un fichier exécutable. Il faut donc qu'il existe des conventions pour décrire des sons, des images, des fichiers exécutables, ou quoi que ce soit d'autre : des vidéos, des objets 3D, des fichiers de paramètres. . .Vous conviendrez aisément qu'il n'aurait aucun sens à ce que des sons soient décrits de la même manière qu'autre chose que des sons. Il faut donc que chaque type de fichier ait ses propres conventions pour décrire son contenu. Nous nous intéressons donc ici aux fichiers décrivant des programmes exécutables.

Nous voulons être capables d'émuler le comportement d'une machine *MIPS* sur une autre machine[6]. Il faut donc trouver une manière de décrire *ce* qui doit être exécuté et *à* quel endroit. De la même manière qu'il existe un format[7] pour décrire les fréquences d'un son, il doit donc en exister un autre pour décrire ce qu'un programme doit exécuter. Un *format* décrit les modalités particulières répondant à la fonction de ce qui ne serait sinon rien d'autre qu'une bête suite d'octets stockés sur un disque. *Cf.* les notions de codeur/décodeur en communications numériques.

Les formats de représentation des programmes informatiques sont multiples. Ils dépendent principalement des systèmes sur lesquels le code machine qu'ils contiennent est censé être exécuté. Dans le monde *Windows*, on parle généralement aujourd'hui de format PE (*Portable Executable*), alors que dans le monde Unix des premiers temps on utilisait le format COFF (*Common Object File Format*[8]).

5. Vous conviendrez aisément qu'exécuter le programme *Mario Kart* sans écran ni manette devient vite frustrant. . .Certains programmes font donc abondamment usage des appels systèmes.
6. Ici, une machine de type i386, mais peu importe.
7. *i.e* Un ensemble de conventions.
8. Le format PE de Windows est une extension du format COFF.

Et aujourd'hui, le monde Unix utilise le format ELF (*Executable and Linkable Format*).

Dans ce projet, nous utiliserons le format ELF pour représenter le code *MIPS* devant être émulé. Autant que faire se peut, nous voudrons vous masquer les détails de cette représentation particulière du code exécutable. Par contre, il nous faut vous expliciter la versatilité de ce format. En effet, le format ELF permet de décrire :

— du code partagé par plusieurs programmes (des *librairies*[9] dites partagées[10]) ;
— du code dit *objet* : la plus simple expression d'un code machine, auquel il manque beaucoup de choses pour s'exécuter sur un système natif ;
— du code dit *exécutable* : du code pouvant être exécuté immédiatement dans un système.

Dans ce projet, nous allons vouloir exécuter du code *objet MIPS* dans un émulateur. Se pose alors la question de savoir comment une vraie machine voit l'exécution d'un code machine, et surtout comment lui faire croire qu'il s'agit d'un authentique code *exécutable* alors qu'il ne s'agit en réalité que d'un code *objet*[11].

2.3 Mémoire et map mémoire

Comme dit précédemment, lorsqu'un fichier exécutable est lancé, il est tout d'abord chargé en mémoire. Dans le cas du microprocesseur MIPS qui possède un bus de 32 bits cette mémoire est au maximum 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.

Même si en pratique la mémoire physique utilisée par les MIPS 32 est bien inférieure à 4Go, pour des raisons qui vous deviendront limpides en cours de Systèmes d'exploitation, il est bon qu'un programme croie qu'il dispose de toute la mémoire de la machine sur laquelle il s'exécute[12]. Quoi qu'il en soit, nous considérerons qu'un programme *voit* effectivement 4Go pour lui seul. Il est donc libre d'organiser la vue de sa mémoire comme bon lui semble.

La vue qu'a un programme en train de s'exécuter (*i.e.* en mémoire) vis-à-vis de lui-même est appelée son *map* mémoire[13]. Ce *map* mémoire doit nécessairement contenir la zone dans laquelle sont contenues les instructions assembleur à exécuter, ainsi que les données sur lesquelles agir. C'est même en réalité les seules zones utiles que contient la description d'un fichier *objet*. Ces zones seront par la suite appelées *section* lorsqu'elles feront référence à leur emplacement dans le fichier *objet*, et

9. Appelées DLL pour *Dynamic Link Libraries* sous *Windows*, mais aussi utilisées pour d'autres buts d'ailleurs.

10. Lorsque deux programmes veulent calculer un cosinus, il paraît légitime de ne pas dupliquer en mémoire le code pour effectuer ce calcul, d'où l'intérêt d'une librairie de mathématiques *partagée*.

11. La complexité des paramètres à prendre en compte pour l'exécution d'un authentique code *exécutable* est telle que vous nous auriez jeté des pierres pour nous punir d'avoir eu cette ambition pour vous. De plus, et c'est aussi encore là que ce projet devient intéressant, nous nous proposons de vous faire toucher du doigt comment ces choses fines s'articulent en pratique. Cette démarche constructive ne saurait trouver d'équivalent intellectuel, et le voyage commence donc avec des fichiers dits *objet*.

12. C'est même pire que ça : même si physiquement une machine dotée d'une largeur de bus d'adresse de 32 bits ne dispose que de 2 Go de mémoire physique (des barrettes de RAM), *chaque* programme croira qu'il dispose de 2^{32}octets=4Go de mémoire pour lui seul.

13. En français : sa *carte* mémoire. Cette même expression est aussi utilisée pour décrire l'ensemble des adresses auxquelles une *machine* a *physiquement* accès – mais notre contexte ne permet pas l'ambiguïté puisque nous parlons de la vue *virtuelle* qu'un *programme* a de lui-même.

segment lorsqu'elle référenceront un espace en *mémoire*. Le *segment* d'instructions exécutables d'un programme est appelé `.text`, et son segment de données est appelé `.data`.

Le paysage est loin d'être complètement décrit. En effet, pour reprendre un exemple décrit précédemment, lorsque'il vous prend la fantaisie d'appeler `sin` pour un calcul vous n'écrivez pas le *code* de `sin`. Vous vous contentez d'inclure `math.h` pour que tout se passe bien. Mais le code de `sin` est ailleurs. Nous ferons l'hypothèse que ce code se trouve dans une zone d'instructions machine appelée `[lib]`. Ce *segment* en mémoire sera réputé contenir la plupart des fonctions usuelles que vous aviez l'habitude d'utiliser [14].

Continuons la description des lieux. Si par exemple, votre code contient un appel à une fonction d'affichage (p.ex., `printf`), fonction située quelque-part dans la zone `[lib]`, il se trouve qu'elle ne fait en réalité que faire appel à une autre fonction permettant d'*écrire* sur un dispositif *matériel* (en l'occurrence : l'écran). En tant que simple *utilisateur* de la machine, vous n'avez en réalité pas le droit de pratiquer des opérations directement sur le *matériel*. Pour ce faire, vous êtes obligés de procéder à ce que l'on appelle un *appel système*. Car, sur une machine bien policée, seul le *système* est autorisé à accéder au matériel, pour le bien de tous et de chacun. La zone mémoire dans laquelle les *appels système* se trouvent est appelée `[vsyscall]`.

Afin de terminer la description du paysage vu par un programme, il nous reste à décrire deux autres zones en mémoire. La plus simple concerne les données qui ne font qu'être lues et qui ne seront jamais modifiées. Ces données résident dans un segment appelé `.rodata` [15]. Un exemple de telles données pourrait être la chaîne de format `"Projet Info 2014"` que votre programme ne manquera de contenir...

Le dernier segment, et un des plus importants, est celui de la pile. La pile contient les données temporairement utiles. Il contient aussi, tout aussi temporairement, les valeurs nécessaires pour que les appels de fonction à fonction se passent correctement : comment passer les paramètres ? comment récupérer la valeur de retour d'une fonction ? Ce segment sera appelé `[stack]` dans la suite. Sa taille est typiquement fixée à 8Mo. Il sert notamment et nécessairement pour les appels récursifs de fonction.

La visite du paysage est maintenant terminée. Néanmoins, pour être certain d'être complet, nous nous devons de mentionner la zone mémoire appelée le *tas*. C'est dans cette zone que sont retournés les pointeurs correspondant aux blocs mémoire alloués par la fonction `malloc` et libérés par `free`. Mais nous n'utiliserons pas le tas car sa gestion, que nous vous aurions fait supporter, se révèle à la fois trop complexe et inutile pour notre propos. Ce segment s'appellerait `[heap]`. La mémoire du MIPS 32 peut être vue comme un tableau d'octets, c'est à dire que l'on ne peut pas accéder directement à un bit particulier en RAM mais seulement à un ensemble de 8 bits (octet). L'adresse d'un octet en mémoire correspond alors au rang qu'il occupe dans le tableau des 4 Go qui la constitue. Pour stocker en mémoire des valeurs sur plusieurs octets, par exemple un mot sur 4 octets, deux systèmes existent (figure 2.3) :

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

La mémoire du MIPS est dite *alignée*. C'est à dire que pour une architecture telle que celle du MIPS 32bits les instructions étant codées sur des mots de 4 octets, le processeur ne lira des instructions que sur des adresses mémoire multiples de 4 octets (32 bits). Les mots (resp. demi-mots) contenus dans

14. Hormis le couple infernal `malloc/calloc` et `free`, pour des raisons qui devraient vous paraître bien plus claire à la fin de ce projet.
15. `ro` comme *read only*.

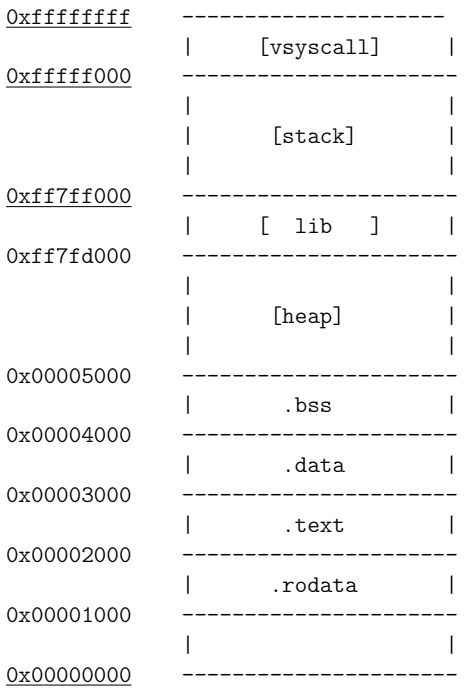


Figure 2.2 – Un *map* mémoire typique de 4Go tel que considéré dans ce projet. Les adresses soulignées sont invariables. La suite du sujet explique comment fixer les autres. Notez qu'il est possible que certains segments (typiquement `.rodata`, `.bss` `.data`) n'existent pas dans certains programmes.

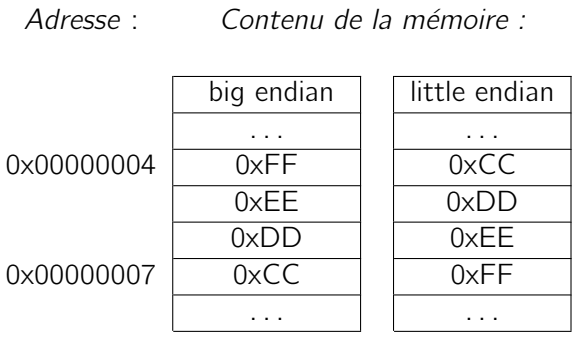


Figure 2.3 – Mode d'écriture en mémoire de la valeur hexadécimale *0xFFEEDDCC* pour un système *big endian* ou *little endian*. Le MIPS est un processeur de type *big endian* : l'octet de poids fort se trouve à l'adresse la plus basse.

des segments mémoire de données devront également être alignés sur 4 (resp. 2) octets.

2.4 Les registres

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

2.4.1 Les registres d'usage général

La machine MIPS dispose de 32 registres d'usage général (General Purpose Registers, *GPR*) de 32 bits chacun, dénotés $0 à $31, et en binaire par les valeurs 0 à 31 (32 valeurs donc 5 bits suffisent). Les registres peuvent également être identifiés par un mnémonique indiquant leur usage conventionnel. Par exemple, le registre $29 est noté `$sp`, car il est utilisé (par convention !) comme le pointeur de pile (sp pour *Stack Pointer*). Dans les programmes, un registre peut être désigné par son numéro aussi bien que son nom (par exemple, `$sp` équivaut à $29).

La figure 2.4 résume les conventions et restrictions d'usage que nous retiendrons pour ce projet.

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

Figure 2.4 – Conventions d'usage des registres MIPS.

2.4.2 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 3.4.2.

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

2.5 Les Instructions

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

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

Nous donnons ici un exemple pour expliciter la spécification d'une instruction : l'opération addition (`ADD`), dont la spécification, telle que donnée dans le manuel, est reportée ci dessous Figure 2.5.

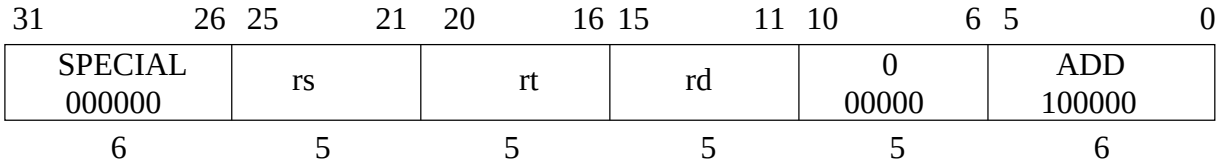


Figure 2.5 – Instruction ADD

Format : `ADD rd, rs, rt`

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

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

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

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

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

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

Restrictions : `None`

Operation :

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

Restriction : `Integer Overflow`

Programming Notes :

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

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

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

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

2.6 Les entrées/sorties

Le mécanisme d'entrée/sortie de données, par exemple afficher une chaîne de caractères à l'écran, consiste à interrompre l'exécution du programme pour faire exécuter un service par le système d'ex-

ploitation. Le principe mis en place dans le MIPS pour les appels systèmes est le suivant :

1. placer un code de service dans le registre `$v0`
2. si nécessaire, passer des paramètres de service en affectant des valeurs dans les registres `$a0` à `$a3`
3. appeler l'instruction `syscall`. L'effet est d'interrompre le programme est d'appeler le service système correspondant au nombre stocké dans `$v0`.
4. si nécessaire, récupérer des valeurs de retour dans les registres `$v0` et `$v1`.

Selon le service appelé, le programmeur peut passer des paramètres (par exemple une valeur à afficher) et récupérer des résultats de ces services dans les registres (par exemple une valeur saisie). Les services à mettre en œuvre dans ce projet sont :

Service	**Code** `$v0`	**Arguments**	**Résultat**
Afficher un entier sur la sortie standard	1	`$a0` = entier à afficher	
Afficher une chaîne de caractères	4	`$a0` = adresse de la chaîne	
Lire un entier sur l'entrée standard	5		`$v0` = entier lu
Lire une chaîne sur l'entrée standard	8	`$a0` = adresse du buffer où ecrire la chaine `$a1` = taille du buffer	
Exit (sort du programme)	10		

Exemple : affichage de la valeur de `$t0` sur la console

```
li  $v0, 1           # service 1 is print integer
add $a0, $t0, $zero  # load desired value into argument register $a0
syscall
```

2.7 Gestion des exceptions

Dans le MIPS une *exception* est une modification soudaine du déroulement de l'exécution quelle qu'en soit la cause (c.-à-d., venant du programme ou d'une source extérieure). C'est un comportement imprévu et donc "exceptionnel", par exemple : débordements, accès à une zone non autorisé [16], division par zéro.

Exemple : débordement (overflow)

```
li  $t0, 0xAFFFFFFF  # t0 est initialisé à une grande valeur négative
li  $t1, -1342177281 # t1 est initialisé à la même grande valeur négative
add $t2, $t1, $t0    # 0xAFFFFFFF + 0xAFFFFFFF = 0x15FFFFFE
                     # résultat allant dans le registre $t2 de 32 bits mais
                     # 0x15FFFFFE ne tient pas sur 4 octet,
                     # il y a débordement
```

Certains processeurs possèdent un registre d'état dont la valeur est modifiée en fonction des exceptions. Dans le cas du MIPS, il n'y a pas de registre d'état et c'est un *exception handler* implanté dans le Coprocesseur 0 (CP0) qui est en charge de gérer les différents cas d'exception. Toutes les exceptions n'ont pas le même impact sur le déroulement du programme et selon les cas, le programme pourra continuer ou non.

16. Si si vous la connaissez celle-la, un indice : `segfault`

2.8 Exécution d'une instruction et delay slot

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

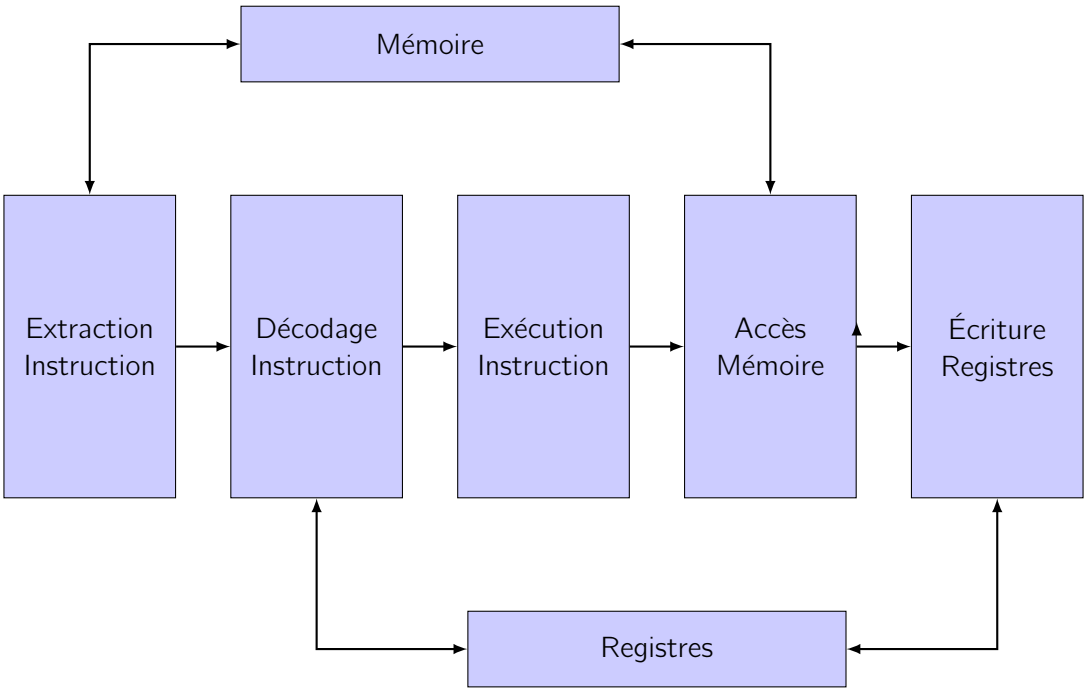


Figure 2.6 – Architecture interne des microprocesseurs RISC

Mais la vie n'est pas toujours aussi rose pour le pipeline. En effet, les instructions de saut et de branchement (p.ex., appel une fonction) nécessitent des cycles supplémentaires avant de sortir du pipeline d'exécution. De ce fait, l'instruction qui suit immédiatement le branchement (dans le *branch delay slot* du branchement) sera déjà partiellement exécutée avant même que l'instruction de branchement ne soit véritablement sortie du pipeline. En particulier :

— l'instruction dans le *delay slot* sera **toujours** exécutée quelque que soit le résultat de l'évaluation du branchement
— le compteur programme PC aura déjà été incrémenté lorsque l'adresse de branchement sera calculée (le PC pointera sur l'instruction suivant l'instruction de branchement, c'est-à-dire celle

dans son *delay slot* soit l'adresse de l'instruction de branchement + 4 octets)

— le comportement du programme sera très incertain si l'instruction à l'adresse résultante du branchement et celle dans son *delay slot* accèdent au mêmes registres, si plusieurs branchements se suivent, si une interruption intervient...

L'exemple ci dessous illustre un comportement incertain dû au *delay slot* :

```
    li  $t0, 12          # met 12 dans t0
    li  $t1, 8           # met 8 dans t1
boucle:                  # étiquette (marque l'adresse de l'instruction suivante)
    addi $t1,$t1,1       # incrémente t1
    BNE $t1,$t0, boucle  # revient à boucle si t1 et t0 sont différents
    li  $t1, 16          # *DELAY SLOT* du BNE t1 est initialisé à 16 en
                         # conséquence le programme ne sortira jamais de la boucle
```

Dans ce programme, la valeur de `$t1` aura été affectée par l'instruction `li $t1, 16` dès le premier passage par `BNE` ainsi lorsque le programme arrivera la deuxième fois sur `addi $t1, $t1,1` `$t1` ne vaudra pas 9 mais 16.

Une manière simple d'éviter ce genre de désagrément est de placer une instruction `NOP` après toutes les instructions de saut et de branchement dans le programme [17]. Dans l'exemple précédent cela donnerait :

```
    ...
 boucle:                 # étiquette (marque l'adresse de l'instruction suivante)
    addi $t1,$t1,1       # incrémente t1
    BNE $t1,$t0, boucle  # revient à boucle si t1 et t0 sont différents
    NOP                  # ajout du NOP (No Operation)
    li  $t1, 16          # t1 est initialisé à 16 en dehors du delay slot du BNE
                         # en conséquence il n'a plus aucun impact sur la boucle
```

17. Il s'agit d'une bulle dans le pipeline. Un cycle est perdu dans le temps d'exécution mais au mois le fonctionnement est correct. Notez que la plupart des compilateurs possèdent une option pour insérer automatiquement les `NOP` derrière les instructions de saut et branchement.

Chapitre 3

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.

3.1 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
```

3.2 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.

3.2.1 Le champ étiquette

Dans un programme assembleur, il est possible de placer des étiquettes avant une instruction ou une directive. Une étiquette ainsi définie peut-être utilisée ailleurs dans le code assembleur pour designer l'instruction ou la directive qui a été étiquetée. Dans le code binaire issu de l'assemblage, les étiquettes prendront la valeur de l'adresse mémoire (relative) de l'instruction ou espace mémoire qu'elles désignent. Dans le code binaire, elles sont regroupées dans la table des symboles.

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

Une étiquette ne peut être définie qu'une seule fois dans une unité de compilation. Sa valeur lors de l'exécution est égale à son adresse d'implantation dans la mémoire après le chargement. Elle dépend donc de la section dans laquelle elle est définie et de sa position dans cette section (cf. section 3.3.1).

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

Exemple

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

3.2.2 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
```

3.2.3 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
```

3.3 Les directives

Les directives sont des commandes à destination de l'assembleur et non des instructions du microprocesseur. Elle sont destinées à être interprétées par l'assembleur [2].

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).

2. on peut faire le parallèle entre les commandes du préprocesseur, par exemple `#define N 5` et les instructions du langage C `int i = N*5;`. Le premier modifiera le code C tandis que le deuxième sera effectivement traduit en suite d'instructions machine

Directive	Description
`.data`	Ce qui suit doit aller dans le segment DATA
`.text`	Ce qui suit doit aller dans le segment TEXT
`.bss`	Ce qui suit doit aller dans le segment BSS
`.set option`	Instruction à l'assembleur pour inhiber ou non certaine options. Dans notre cas seule l'option *noreorder* est considérée
`.word w1, ..., wn`	Met les n valeurs sur 32 bits dans des mots successifs (ils doivent être alignés!)
`.byte b1, ..., bn`	Met les n valeurs sur 8 bits dans des octets successifs
`.asciiz s1, ..., sn`	stocke les n chaînes de caractères à la suite en mémoire en ajoutant un \0 derrière chacune d'elle
`.space` *n*	Réserve *n* octets en mémoire. Les octets sont initialisés à zéro.

3.3.1 Directives de sectionnement

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

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

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

3.3.2 Les directives de définition de données

On distingue les données initialisées des données non initialisées.

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

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

```
toto: .space 13
```

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

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

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

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

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

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

[étiquette] `.asciiz` **chaîne de caractères** *chaîne* est une suite de caractères entre guillemets "'". Cette chaîne est stockée en mémoire et un caractère \0 est ajouté en fin de chaîne dans la mémoire. Par exemple, la ligne suivante permet de stocker la chaîne "Projet Info\n" dans le programme. L'étiquette `String1` référencera l'adresse du premier caractère de la chaîne. La taille totale de la chaîne sera de 12 caractères[3] + le caractère de fin de chaîne ajouté par le compilateur. Au final 13 octets seront stocké en mémoire.

```
String1:   .asciiz "Projet Info\n"
```

3.3.3 Les modes d'adressage

Comme nous le verrons au chapitre 3.4, 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 la valeur de l'opérande. Le langage assembleur MIPS contient 5 modes d'adressage décrit ci dessous.

3.3.4 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.

3. attention, les caractères accentués et certains caractères spéciaux sont codés sur plus d'un octet en UTF-8

Exemple :

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

3.3.5 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)
```

3.3.6 Adressage indirect avec base et déplacement

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

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

Exemple :

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

3.3.7 Adressage absolu aligné dans une région de 256Mo

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

Les 28 bits de poids faible de l'adresse de saut sont contenus dans l'opérande décalé de 2 bits vers la gauche (car les instructions sont alignées tous les 4 octets). Les poids forts manquants sont pris directement dans le compteur programme. Un exemple est donné au paragraphe 3.4.2.

Exemple :

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

3.3.8 Adressage relatif

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

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

Exemple :

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

3.4 Instructions étudiées dans le projet

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

3.4.1 Catégories d'instructions

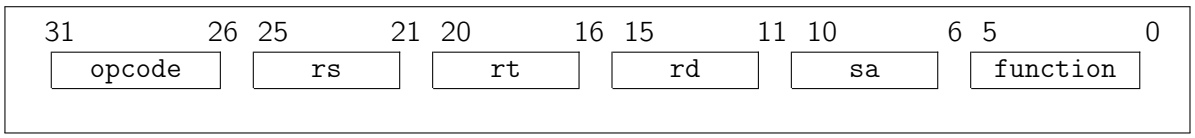
Les processeurs MIPS possèdent des instructions simples de taille constante égale à 32 bits [4]. Ceci facilite notamment les étapes d'extraction et de décodage des instructions, réalisées chacune dans le pipeline en un cycle d'horloge. Les instructions sont toujours codées sur des adresses alignées sur 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 :



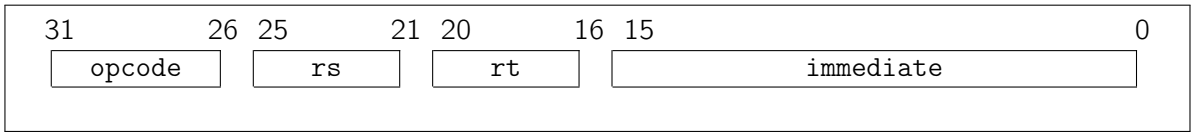
avec les champs suivants :

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

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

Les instructions de type I

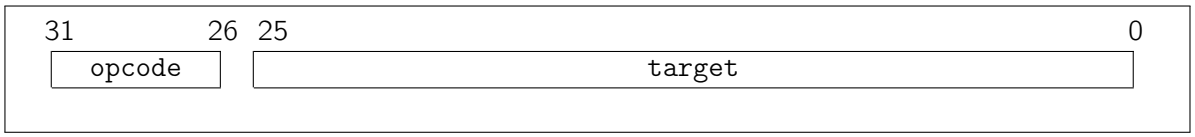
Le codage binaire des instructions *I-type*, pour "immediate type", suit le format :



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.

3.4.2 Détails des instructions à prendre en compte dans le projet

Instructions arithmétiques

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

ADD fait l'addition de deux registres rs et rt. SUB fait la soustraction de deux registres rs et rt. 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.
Attention, la plupart des instructions ont plusieurs versions. Par exemple, le U ajouté à ADD, ADDI et SUB signifie que l'opération ne doit pas lever d'exception même en cas de débordement.

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

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

Les instructions logiques

Mnemonic	Opérandes	Opération	Type
AND	$rd, $rs, $rt	$rd = $rs AND $rt	Type R
ANDI	$rt, $rs, immediate	$rd = $rs AND immediate	Type I
OR	$rd, $rs, $rt	$rd = $rs OR $rt	Type R
ORI	$rt, $rs, immediate	$rd = $rs OR immediate	Type I
XOR	$rd, $rs, $rt	$rd = $rs XOR $rt	Type R

Les deux registres de 32 bits `rs` et `rt` sont combinés bit à bit selon l'opération logique effectuée. Le résultat est placé dans le registre 32 bits `rd`. Pour les versions avec valeur immédiate, (`ANDI, ORI`), `immediate` (sur 16 bits) est étendue à gauche avec des zéros puis combinée avec le registre `rs`.

Les instructions de décalage

Mnemonic	Opérandes	Opération	Type
SLL	$rd, $rt, sa	$rd = $rt << sa	Type R
SRL	$rd, $rt, sa	$rd = $rt >> sa	Type R
SRA	$rd, $rt, sa	$rd = $rt >> sa (signé)	Type R

Le contenu du registre 32 bits `rt` est décalé à gauche pour `SLL` et à droite pour `SRL` et `SRA` de `sa` bits (en insérant des zéros pour `SRL` et en copiant le bit de signe pour `SRA`). `sa` est une valeur immédiate sur 5 bits, donc entre 0 et 31.

Les instructions Set

Mnemonic	Opérandes	Opération	Type
SEB	$rd, $rt	$rd = sign_extend($rt[7..0])	Type R
SLT	$rd, $rs, $rt	if $rs < $rt then $rd = 1, else $rd = 0	Type R
SLTU	$rd, $rs, $rt	if $rs < $rt then $rd = 1, else $rd = 0	Type R
SLTI	$rt, $rs, immediate	if $rs < immediate then $rd = 1, else $rd = 0	Type I
SLTIU	$rt, $rs, immediate	if $rs < immediate then $rd = 1, else $rd = 0	Type I

`SEB` stocke dans `rd` la valeur de l'octet de poids faible de `rt` dont le signe a été étendu sur 32 bits. Dans le cas de `SLT` (resp. `SLTI`), le registre `rd` est mis à 1 si le contenu du registre `rs` est plus petit que celui du registre `rt` (resp de la valeur signée `immediate`), à 0 sinon. Les valeurs `rs` et `rt` sont des entiers signés en complément à 2. Pour les versions `SLTU` et `SLTIU` la comparaison est effectuée en considérant `rt` ou la valeur signée `immediate` étendue sur 32 bits comme non signé.

Les instructions Load/Store

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

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

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

— Store Word (`SW`) place le contenu du registre `rt` dans le mot de 32 bits à l'adresse mémoire (`$rs + offset`). `offset` est une valeur signée sur 16 bits codée en complément à 2, elle est placée dans le champ immediate.
— `LB` et `SB` réalise la même opération que `LW` et `SW` sauf qu'un seul octet est lu/écrit.
— `LBU` réalise la même opération que `LB` sauf que l'octet chargé est considéré non signé.
— Load Upper Immediate (`LUI`) place le contenu de la valeur entière 16 bits `immediate` dans les deux octets de poids fort du registre `$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, de saut et de contrôle

Mnemonic	Opérandes	Opération	Type
BEQ	$rs, $rt, offset	Si ($rs = $rt) alors branchement	Type I
BNE	$rs, $rt, offset	Si ($rs != $rt) alors branchement	Type I
BGEZ	$rs, offset	Si ($rs >= 0) alors branchement	Type I
BGTZ	$rs, offset	Si ($rs > 0) alors branchement	Type I
BLEZ	$rs, offset	Si ($rs <= 0) alors branchement	Type I
BLTZ	$rs, offset	Si ($rs < 0) alors branchement	Type I
J	target	PC=PC[31:28] target	Type J
JAL	target	GPR[31]=PC+8, PC=PC[31:28] target	Type J
JALR	$rd,$rs $rs ($rd vaut 31)	$rd= adresse de retour, PC=$rs	Type R
JR	$rs	PC=$rs.	Type R
BREAK		cause une exception	Type R
SYSCALL		provoque un appel système	Type R

— `BEQ` effectue un branchement après l'instruction si les contenus des registres `rs` et `rt` sont égaux. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
— `BNE` effectue un branchement après l'instruction si les contenus des registres `rs` et `rt` sont différents. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
— `BGEZ` effectue un branchement après l'instruction si le contenu du registre `rs` est positif ou nul. L'offset signé de *18 bits* (16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement pour déterminer l'adresse effective du saut. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.
— `BGTZ` même instruction que `BGEZ` mais effectue un branchement après l'instruction seulement si le contenu du registre `rs` est strictement positif.
— `BLEZ` même instruction que `BGEZ` effectue un branchement après l'instruction seulement si le contenu du registre `rs` est négatif ou nul
— `BLTZ` même instruction que `BLEZ` mais effectue un branchement après l'instruction seulement

si le contenu du registre `rs` est strictement négatif.

— `J` effectue un branchement aligné à 256 Mo dans la région mémoire du PC. Les *28 bits* de poids faible de l'adresse du saut correspondent au champ `target`, décalés de 2. Les 4 bits de poids fort restant correspondent au 4 bits de poids fort du compteur PC. Une fois l'adresse de saut calculée, celle-ci est mise dans le PC.

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

```
0x26767296 --  10101101010100101010100011
```

Comme toutes les instructions sont alignées sur des adresses multiples de 4, les deux bits de poids faible d'une instruction sont toujours 00. On peut donc décaler le champs `offset` de 2 bits, ce qui donne une adresse de saut sur 28 bits :

```
adresse 28 bits:  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.

— `JALR` effectue un saut à l'adresse spécifiée dans `rs` et place l'adresse de retour dans le registre `$rd`. Si `$rd` n'est pas spécifié, le registre par défaut $31 est utilisé.

— `JR` effectue un saut à l'adresse spécifiée dans `rs` Le contenu du registre 32 bits `rs` contient l'adresse du saut.

— `BREAK` provoque une exception qui doit être gérée par le coprocesseur 0 du MIPS.

— `SYSCALL` provoque un appel système qui doit être gérée par le coprocesseur 0 du MIPS afin d'exécuter le service demandé.

Les pseudo-instructions

Les pseudo-instructions ne sont pas (toutes) 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, immediate16	ADDI $rt, $zero, immediate16
LI	$rt, immediate32	LUI $rt, hi(immediate32) ORI $rt, $rt, lo(immediate32)
BLT	$rt, $rs, target	SLT $1, $rs, $rt 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. Selon la valeur de immediate sur 16 ou 32 bits, le code correspondant est different.
— BLT permet un branchement suite à la comparaison directe de deux registres, alors qu'on ne peut comparer qu'à zéro avec les instrutions. Si le contenu du registre $rs est inférieur à celui du registre $rt, le branchement vers target est effectué après l'instruction.

Les autres pseudo-instructions reprennent des instructions existantes mais avec des opérandes différents. Nous nous intéresserons ici aux cas où une étiquette est utilisée comme opérande d'une instruction de saut, branchement ou accès mémoire.

Mnemonic	Opérandes	Opération équivalente
B???	[$rs, $rt,] label	offset = label - (adresse instruction +4) B??? [$rs, $rt,] offset
J/JAL	label	J addend26 + relocation
LW/SW	$rt,label	LUI $1, addendHI16 + relocation LW/SW $rt, addendLO16($1) + relocation

— Tous les branchements sur une même étiquette fonctionnent sur le même principe : le décalage offset est calculé par différence entre l'adresse de mémoire nommée par l'étiquette et celle de l'instruction suivant le branchement (car à l'exécution du branchement PC aura déjà été incrémenté).
— une instruction de saut qui utilise une étiquette comme opérande est obligatoirement accompagnée d'information de relocation. La valeur codée dans le champ instr_index est l'addend de 26 bits nécessaire au calcul de relocation.
— L'adresse mémoire accédée par les instructions de type S??/L?? suit la syntaxe offset(base), où offset est codé sur 16 bits. Cependant une étiquette désigne une adresse mémoire, donc sur 32 bits. Il est donc nécessaire d'utiliser deux instructions LUI puis S??/L??? pour traiter séparément les 16 bits de poids fort (addendHI16) et les 16 bits de poids faible de l'adresse mémoire (addendLO16). Ces deux valeurs de addend ne sont pas absolue est sont toujours accompagnées d'information de relocation.

Le mécanisme de relocation sera abordé plus loin dans le sujet dans le chapitre qui lui est consacré.

Chapitre 4

Relocation

4.1 Principe de la relocation

4.1.1 Nécessité d'un code relogable

Le rôle de l'assembleur est, à partir du fichier source en langage assembleur, de générer le code binaire de l'ensemble des instructions et des données composant le programme. C'est ce code binaire qui sera chargé en mémoire puis exécuté lorsque le programme sera lancé.

Certaines instructions peuvent être codées directement, indépendamment du reste du programme et de l'adresse à laquelle elle seront chargées en mémoire. C'est notamment le cas des opérandes des instructions ne mettant en jeu que des registres ou des valeurs immédiates. Par contre, quand les opérandes des instructions sont des étiquettes, comme dans le cas d'un saut ou d'un accès à la zone `.data`, le code binaire de ces instructions ne peut pas être produit à l'assemblage. En effet, les adresses définitives référencées par les étiquettes ainsi que les adresses des sections ne seront connues que lors du chargement du programme en mémoire.

Les exemples suivants illustrent les problèmes rencontrés dans ce cas.

Exemple 1:

```
.text
ADDI $3,$0,12345   # met 12345 dans le registre 3
SW $3, X           # écrit le contenu de $3 à l'adresse X

.data
X: .word 0         # réservation d'un mot initialisé à 0 dans data
```

Dans cet exemple, l'instruction ADDI ne pose aucun problème. Par contre le SW [a] dépend de l'adresse à laquelle correspond l'étiquette `"X"`. Or cette adresse ne peut être connue avant que l'adresse d'implantation des sections `.text` et `.data` ne soient elles mêmes connues. Or celles-ci ne seront connues qu'au chargement du programme. Mais le programme ne pourra jamais être chargé s'il n'est pas assemblé... Le problème est donc : Comment coder l'instruction SW de l'exemple si l'offset ne peut pas être calculé ?

a. Il s'agit ici de la *pseudo-instruction* SW dont le deuxième opérande est une adresse (32 bits) et non de l'instruction qui attend un offset sur 16 bits. Elle sera remplacée par l'assembleur par les instructions LUI et SW.

Exemple 2:

```
.data
X: .byte 0xAB   # réservation d'un octet initialisé à 0xAB
Y: .word Z      # réservation d'un mot initialisé à l'adresse
                # référencée par l'étiquette Z
Z: .word 0      # réservation d'un mot initialisé à 0xAB
```

Dans cet exemple, la valeur qui suit l'étiquette `Y` est en fait l'adresse référencée par l'étiquette `Z`. Comme dans le cas précédent, il est nécessaire de connaître l'adresse d'implantation de la section `.data` pour déterminer les valeurs de `X, Y, Z`.

Exemple 3:

```
JAL fonction  # Appel de la procédure "fonction"
NOP           # On ne fait rien pour cause de Delay Slot (cf section 2.8)
B end         # Retour ici après la procédure, on se branche sur la fin du programme
NOP           # delay slot

fonction :    # début de la procédure
ADD $t1,$0,$0
JR $31        # fin de procédure fonction, retour à l'appelant

end:
```

Ici, deux étiquettes sont utilisées par des instructions de Branchement et de saut. Ces deux cas ne sont cependant pas équivalents :

- Le codage du branchement nécessite de calculer le décalage entre l'instruction `B` et l'étiquette `end`. Mais puisque cette étiquette se situe dans la même section que le branchement lui-même, ce décalage peut être calculé lors de l'assemblage. Mais ce n'est pas toujours le cas, car cette étiquette aurait très bien pu être déclarée dans un autre fichier assembleur. Ce décalage n'aurait été dans ce cas calculable qu'au moment de l'édition de liens.
- Le problème est différent pour l'instruction `JAL`. En effet, l'adresse de destination du saut est calculée à partir de la valeur de l'opérande (ici l'étiquette) **ET** de la valeur de `PC`, soit l'adresse de l'instruction `JAL` + 4 [a].
  Donc, même si la position de l'étiquette `fonction` est ici connue *par rapport* à l'instruction `JAL`, il manque la position effective en mémoire de cette instruction pour pouvoir déterminer son codage complet.

a. Souvenez-vous que lors de l'exécution d'une instruction, le compteur PC a déjà été incrémenté de 4 octets.

En résumé, **le code d'un programme n'est déterminé de manière absolue qu'à partir du moment où les adresses mémoires auxquelles seront implantées les différentes sections (.text, .data ...) sont connues.**

4.1.2 Codage en position, flat binary file

Une solution est d'assembler le programme en définissant à l'avance, l'adresse mémoire de la section `.text`. Par exemple si `.text` est implanté en 0x5000, la section `.data` sera ensuite placée à la suite et ainsi de suite pour toutes les autres sections. De cette manière, il devient possible de coder l'ensemble des cas des exemples 1, 2 et 3.

Le programme peut ainsi être chargé directement en mémoire **UNIQUEMENT** *à l'adresse prévue* et exécuté en l'état. Par contre le programme n'est plus valide dès que les adresses utilisées ne sont plus disponibles. C'est notamment le cas si la machine ne dispose pas de suffisamment d'espace memoire ou si les plages d'adresses sont déjà utilisées par d'autres programmes.

4.1.3 Code relogeable

Une autre solution consiste à créer des fichiers objets dits *relogeables* qui contiennent des informations permettant de **modifier le code binaire au moment du chargement du programme**, juste

avant l'exécution, pour l'adapter dynamiquement aux plages d'adresse qui lui auront été allouées. On appelle cette opération la *relocation*.

Le principe des codes relogeables repose sur le fait que si les adresses effectives des instructions et étiquettes ne sont pas connues au moment de l'assemblage, leur position *relative* au début de la section où elles se trouvent l'est ! Ainsi dans les exemples ci-dessous :

— l'étiquette `X` de l'exemple 1 correspond en fait à l'adresse de la section `.data`
— l'étiquette `Y` (resp. `Z`) de l'exemple 2 correspond à l'adresse de la section `.data` + 4 octets (resp. 8 octets).
— l'étiquette `fonction` (resp. `end`) de l'exemple 3 correspond à l'adresse de la section `.text` + 16 octets (resp. 24 octets) et l'instruction `JAL` est implantée en début de section.

Lors de l'assemblage, il suffit donc de générer une version *'relative'* du code binaire et d'inclure dans le fichier objet toutes les informations qui permettent d'adapter ce code lors de son positionnement en mémoire. Un fichier relogeable contient donc au moins :

— le code des sections assemblées avec des informations liées au positions relatives des instructions et données par rapport au début de chaque section.
— une *table de relocation* indiquant, pour chaque section, les positions des adresses relatives à mettre à jour au moment du chargement et le mode de relocation (définissant les calculs à effectuer pour déterminer les bonnes adresses effectives).
— une *table des symboles* contenant les labels des étiquettes (chaînes de caractères) associées à leur adresse relative (pour les symboles définis localement).

Le chargement, avant l'exécution du programme est alors précédé d'une phase de relocation :

1. le fichier objet est lu pour déterminer le nombre d'octets nécessaires (taille des sections) pour copier les instructions et données dans la mémoire.
2. l'éditeur de liens (ou sur certains systèmes le chargeur dynamique) détermine à quelles adresses vont être placées les différentes sections composant le programme
3. Le relocation proprement dite a alors lieu. À partir des adresses d'implantation déterminées par le système et des informations de la table de relocation, elle convertit les *adresses relatives* des étiquettes en *adresses absolues*. Selon les modes de relocation à utiliser, cette conversion est plus ou moins simple.
4. le code ainsi modifié est prêt a être exécuté.

Le principal intérêt des codes relogeables est d'être portable d'une machine à une autre puisqu'ils ne sont pas spécifiques à une configuration mémoire. Ils sont également moins encombrants, en particulier dans le cas de programme multi-fichiers ou utilisant des librairies car les procédures ne sont codées qu'une seule fois.

Compilation séparée

Les fichiers relogeables permettent la compilation séparée, c'est-à-dire la création d'un fichier objet qui fait partie d'un programme plus vaste utilisant plusieurs fichiers objets qui seront rassemblés par l'édition de liens.

Au moment de l'assemblage, il est possible que certains symboles soient indéfinis (par exemple, procédures définies dans un autre fichier) et c'est l'éditeur de liens qui ira chercher leur définition. Pour chaque symbole indéfini, l'assembleur va donc laisser un "trou" à l'endroit où ce symbole est référencé et noter dans le fichier binaire à quel symbole correspond ce trou, ainsi que l'action à exécuter pour l'adresse finale au moment où il aura connaissance de l'adresse de ce symbole.

4.2 Informations nécessaires à la relocation

4.2.1 Table de relocation

Les informations nécessaires à la relocation sont définies dans les tables de relocation qui seront incluses par l'assembleur dans le fichier objet. Une table est associée à chaque section contenant des symboles à reloger. La table associée à la section `.text` (resp. `.data`) est appelée `.rel.text` (resp. `.rel.data`). Chaque ligne d'une table de relocation est définie par les lignes suivantes :

- `offset` : la position de l'entrée à modifier, en nombre d'octets par rapport au début de la section à laquelle la table est associée.
- `type` : le mode de relocation
- `value` : le symbole par rapport auquel il faudra faire la relocation. Dans le cas de symboles locaux, `value` est *l'index* de la zone contenant le symbole et dans le cas de symboles globaux (éventuellement non définis à l'assemblage) c'est l'index du symbole en question.

Dans l'exemple 2, l'entrée à reloger est un mot à l'adresse `Y` (soit 4 octets après le début de la section `.data`) et le symbole à reloger est `Z` qui se trouve aussi dans la section `.data`. La table associée est donc :

```
[.rel.data]
Offset          Type            Value
00000004        R_MIPS_32       .data
```

Dans l'exemple 3, l'entrée à reloger est l'instruction `JAL`, la première de la section `.text` et le symbole recherché est l'étiquette `fonction` qui se trouve également dans la section `.text`. La table associée est donc :

```
[.rel.text]
Offset          Type            Value
00000000        R_MIPS_26       .text
```

4.2.2 Champ addend

Une dernière donnée pour la relocation d'une entrée est le `addend`, c'est-à-dire la valeur binaire présente dans l'espace qui va être modifié par le relocation. Cette valeur, contenue dans le fichier objet, va être récupérée par l'éditeur de lien ou le chargeur dynamique et utilisée avec les informations de la table de relocation pour déterminer le nouveau code de l'instruction ou de la donnée. Elle sera finalement écrasée par le nouveau code au moment du chargement en mémoire.

Selon le mode de relocation, le addend correspond aux 32/26/16 bits de poids faible de l'entrée (toujours sur 32 bits) à reloger.

Dans le cas de symboles définis mais dont on ne connaît pas l'adresse absolue, `addend` est en générale l'adresse relative du symbole par rapport au début de section. Dans le cas des symboles indéfinis (référence à un symbole défini dans un autre fichier), l'assembleur, n'ayant pas d'adresse relative laisse généralement le addend à zéro.

4.2.3 Modes de relocation du MIPS

Le *mode de relocation* détermine le calcul spécifique à effectuer pour reloger une entrée (instruction ou donnée). Beaucoup de modes existent mais nous nous restreindrons à ceux définis ci-dessous :

R_MIPS_32 Ce mode est utilisé pour reloger une donnée en section `.data`. Les 32 bits de l'entrée sont modifiés

R_MIPS_26 Ce mode est utilisé pour reloger une instruction de saut de type `J`. Les 6 bits de poids fort (*opcode*) ne seront jamais modifiés par la relocation.

R_MIPS_HI16 &**R_MIPS_LO16** Ces deux modes fournissent les informations pour la relocation d'instructions successives de type I. Dans notre cadre, ils seront utilisés pour les instructions d'accès mémoire (`SW, LB...`). Seuls les 16 bits de poids faibles seront modifiés par la relocation. Des entrées de ces types apparaissent toujours par paire dans une table de relocation : chaque relocation **R_MIPS_HI16** est immédiatement suivie d'une relocation de type **R_MIPS_LO16** [1].

Les notations utilisées pour la description des calculs de relocations sont les suivantes :

— **P**lace désigne l'adresse de l'entrée à reloger, c'est-à-dire l'adresse allouée par l'éditeur ou le chargeur à la section plus la valeur `offset` associée à l'entrée, c'est à dire l'adresse "finale" de l'élément à reloger.

— **A**ddend désigne la valeur à **a**jouter pour calculer la valeur du champ à reloger (c'est la valeur du champ avant relogement). Selon le mode de relocation il s'agit des 32/26/16 bits de poids faible de l'entrée (en fait l'extension signée de `addend`)

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

— **AHL** est une valeur de 32 bits calculée à partir des `addend` des deux entrées de relocation **R_MIPS_HI16** et **R_MIPS_LO16** consécutives. Si `AHI` et `ALO` sont les `addend` d'une paire d'entrées `HI16` et `LO16` de 16 bits chacun alors `AHL = (AHI << 16) + (short)ALO`. Par exemple si `AHI=0xABCD` et `ALO=0x12` alors `AHL=0xABCD0012`.

Avec ceci les calculs de relocation sont :

Mode	Adresse du 1er bit à modifier	Nb de bits à modifier	Valeur à écrire
`R_MIPS_32`	P	32	S+A
`R_MIPS_26`	P+6 bits	26	(—A<<2)—(P & 0xF0000000)—+S)>>2
`R_MIPS_HI16`	P+16 bits	16	((AHL+S)- short(AHL+S))>>16
`R_MIPS_LO16`	P+16 bits	16	AHL+S

4.3 Format `elf`

Plusieurs formats de fichier relogeable existent pour différent systèmes d'exploitation. Dans ce projet nous utiliserons le format ELF qui est celui utilisés par les systèmes UNIX. Pour la description de ce format veuillez vous referer à l'annexe A.

1. En réalité une entrée **R_MIPS_HI16** est suivie d'une ou plusieurs entrées **R_MIPS_LO16**.

Chapitre 5

Spécifications de l'émulateur MIPS32, travail à réaliser

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

La Figure 5.1 présente le diagramme des différents composants d'un émulateur et de leurs interactions. Le programme assembleur, sous forme d'un fichier ELF, contient non seulement les instructions mais aussi les données du programme. Celui-ci est chargé dans les segments de mémoire du microprocesseur. Les registres contiennent les données représentant les arguments des instructions arithmétiques et logiques. On peut noter la présence du **PC** (*Program Counter* ou *Instruction pointer*) qui est l'indice donnant l'adresse de la prochaine instruction à exécuter. Ce PC est mis à jour par le module de décodage/exécution qui doit interpréter les instructions pour connaître la prochaine à exécuter. Par exemple, une simple opération d'addition (`ADD $9,$10,$11`) impliquera un simple incrément du PC (la prochaine instruction est celle suivant l'addition) mais un branchement tel que `BNE $9,$10, ailleurs` demandera l'exécution complète de l'instruction pour savoir si oui ou non `$9` et `$10` sont égaux avant de savoir s'il faut sauter à l'adresse `ailleurs` ou exécuter l'instruction suivante.

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

5.0.1 Modes d'utilisation de l'émulateur

L'émulateur sera un programme C qui sera appelé en tapant sous Linux la commande :

```
emul-mips
```

ou

```
emul-mips script_filename
```

où `script_filename` est le nom d'un fichier de commandes à exécuter. En effet, tout comme le *shell*, l'émulateur MIPS pourra fonctionner sous deux modes.

- Le mode dit *interactif* dans lequel chaque commande de l'utilisateur est interprétée directement, exécutée puis l'invite de commande est rendu en attente d'une nouvelle commande. Dans notre cas, ce mode est utilisé lorsque l'exécutable est lancé sans paramètre.
- Le mode dit *non-interactif* dans lequel l'interpréteur va lire les commandes une à une dans

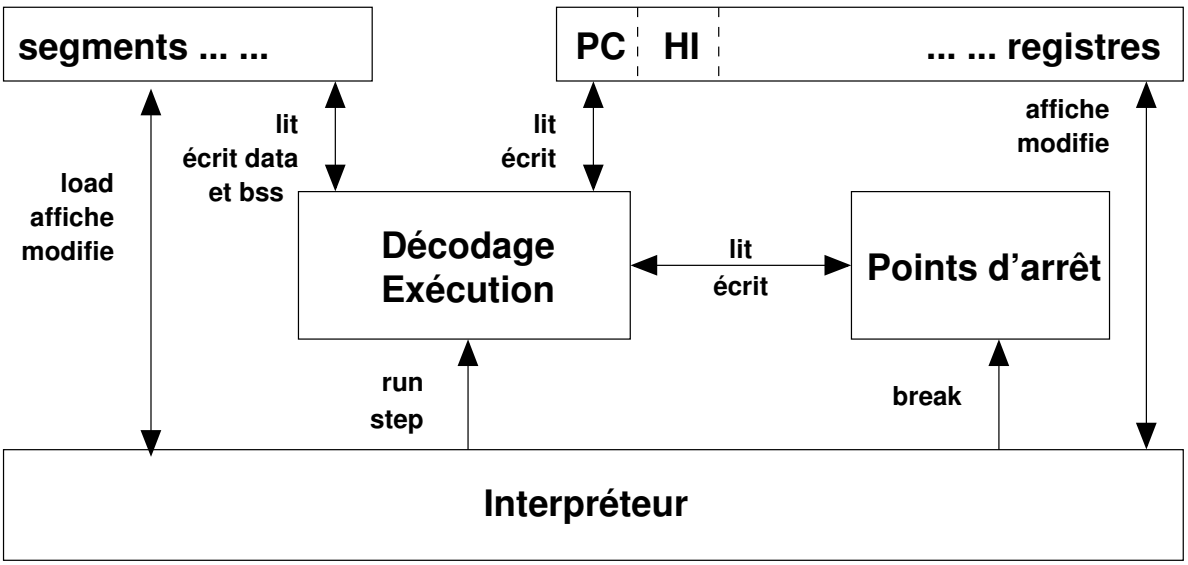


Figure 5.1 – Architecture d'un émulateur MIPS

un fichier[1]. Ce fichier est couramment appelé un *script*[2]. Dans notre cas, ce mode est utilisé lorsque l'exécutable est lancé avec le chemin d'un fichier en paramètre.

Pour faire un parallèle avec les cours de programmation de première année, taper chaque commande `gcc -c fichier.c` correspond au mode *interactif* alors qu'utiliser un Makefile correspond au mode *non-interactif*[3]. Le mode *non-interactif* sera particulièrement utilisé pour automatiser des tests de l'émulateur ainsi que des fichiers objet MIPS.

format des fichiers de script

Nous partirons du principe que les fichiers de script sont composés d'une seule commande par ligne et que les commentaires sont précédés d'un dièse #. L'exemple suivant illustre le contenu d'un fichier de script pour l'interpréteur :

```
#fichier de test de l'émulateur
load fichier.o    # charge en mémoire le fichier objet
disp reg all      # affiche le contenu de tous les registres
exit
#sort du programme
```

1. C'est en fait plus générique car il s'agit plus généralement d'un *flux* d'entrée.
2. Un *script* désigne tous les programmes textuels qui n'ont pas besoin d'être compilés avant d'être exécuté. On dit qu'ils sont *interprétés*...Par exemple, les fichiers de programmes de base de MATLAB sont des scripts.
3. Ceci n'est pas tout à fait exact car c'est l'utilitaire `make` qui interprète le Makefile, mais vous nous excuserez cette approximation car cette comparaison n'est pas dénuée de vertu pédagogique.

5.1 Machine simulée

5.1.1 Adresse des segments de mémoire

Pour charger le programme en mémoire, il faudra (au moins dans un premier temps) respecter les contraintes ci dessous :

- Le premier segment, `.text` (ou `.rodata` si présente) sera placé à une adresse fixée par défaut par le programme. Cette adresse doit également être paramétrable via l'interpréteur.
- Les adresses d'implantation des autres segments dépendent de la taille du premier segment et ne peuvent donc être déterminées *a priori*. La seule contrainte est que les adresses mémoires des segments `.text, .data, .bss, .rodata`, etc soient toujours multiples de 0x1000 (soit 4096 octets ou 4ko). On parle d'alignement de pages. Par exemple, si le segment `.text` est implanté en 0x2000 et fait une taille de 4352 octets alors le segment `.data` devra commencer en 0x3000. Ainsi si le segment `.data` a une taille de 200 octet, la section `.bss` devrait commencer en 0x4000 [4].
- Comme indiqué en section 2.3, la fin de la pile sera implantée 1ko avant la fin de la mémoire et devra avoir une taille parametrable.

Notion d'adresse virtuelle, adresse réelle

Il faut bien comprendre que les valeurs des adresses des segments précités sont *virtuelles*. En effet, nous allons émuler la mémoire, c'est-à-dire, faire croire, par exemple, que le segment `.text` débutera en 0x1000 alors que les données réelles qui lui sont associées seront véritablement stockées dans une zone mémoire allouée sur la machine *hôte* dont l'adresse est inconnue au démarrage du programme et pour laquelle nous n'avons pas prise [5]. Ainsi, si l'on veut accéder à l'octet en 0x100c d'adresse virtuelle, votre programme devra trouver à quelle adresse réelle il se trouve (qui peut très bien être en 0x7fff14b7a484 !). Votre programme devra donc contenir un mécanisme pour traduire les adresses virtuelles en adresses réelles et vice-versa.

5.1.2 Pile d'appel

Une pile est un mécanisme très utilisé pour permettre les appels de fonctions imbriquées (notamment récursifs) en toute sécurité. En effet, prenez l'exemple si dessous :

4. En fait, les adresses d'implantation suivent des lois plus complexes pour permettre l'optimisation de la gestion mémoire. La règle imposée dans le sujet permet de faciliter la mise en œuvre. Vous aurez le loisir d'en apprendre plus sur l'occupation mémoire d'un programme en cours d'OS.

5. Hé oui, c'est le malloc qui décide où l'on peut réserver de la mémoire sur la machine hôte, on ne peut donc lui imposer de trouver de la place en 0x1000 (zone à laquelle vous n'avez de toutes façons pas accès)

```
.text
debut :
  addi $a0,$zero,15  # on met 15 dans le 1er argument
  addi $a1,$zero, 3  # on met 3 dans le 2e argument
  jal puiss          # on veut 15^3
  move $v0,$t2       # on stocke le résultat dans $t2
  J fin              # on va à la fin du programme

puiss:               # procédure récursive de calcul de puissance
  bgtz $a0, else     # si la puissance vaut zéro
  addi $v0,$zero,1   # on init la valeur de retour à 1
  B sortie           # et on s'en va
else:                # sinon
  addi $a0,$a0, -1   # on décrémente la puissance
  jal puiss          # on relance avec la puissance moins n-1
  mult $a1,$v0       # on calcul à la puissance n
  mflo $v0           # on stocke le résultat dans $v0
sortie:
  jr $31             # on sort de la fonction

fin:
```

Dans ce code où l'on utilise le calcul de puissance par procédure récursive se pose le problème suivant :

— l'appel à JAL stocke automatiquement l'adresse de retour dans le registre `$ra ($31)`. Or si plusieurs JAL se suivent (comme dans l'exemple), l'adresse originale de retour est écrasée (et donc perdue).

Par ailleurs, certaines fonctions imbriquées peuvent utiliser les mêmes registres temporaires et donc écraser des valeurs entres les appels !

Pour résoudre ces problèmes, on utilise couramment un espace mémoire pour stocker les variables intermédiaires entre deux appels afin de pouvoir les rétablir lorsque l'on retourne dans la fonction appelante. Cet espace, c'est la pile. Typiquement, on sauve la valeur d'adresse de retour, les paramètres ainsi que les variables locales (i.e., les registres utilisés par la fonction). On sauve ces informations en les 'empilant' dans l'espace mémoire de la pile avant l'appel d'une fonction puis en les 'dépilant' au retour de celle-ci.

Sur la plupart des architectures, la convention est d'initialiser la pile en 'haut' du segment qui lui est attribué. Ainsi, si la pile occupe l'espace `[0x0f7ff000,0x0ffff000[` la pile devra donc être initialisée à `0x0fffeffc`. En effet, la pile est utilisée pour stocker de l'information de la taille d'un mot, soit 4 octets.

5.2 Description des commandes de l'interpréteur

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

L'interface utilisateur à réaliser est un interpréteur de commandes (comme le Shell Linux, si ce n'est que ce ne sont pas de commandes Shell qui sont interprétées). Le fonctionnement de cet interpréteur est le suivant, dans une boucle infinie :

- Le programme se met en attente d'une commande ;
- L'utilisateur tape une commande et termine par "Entrée" $\hookleftarrow$;

- Le programme décode la commande et l'exécute, puis se met en attente de la commande suivante.

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

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

Concernant la syntaxe des paramètres, il est à noter les conventions suivantes :

- Les crochets [ ] indiquent que l'argument (ou le groupe d'arguments) à l'intérieur est optionnel.
- L'étoile $*$ collée à un argument indique que cet argument peut être répété un nombre quelconque de fois.
- Le symbole $+$ quand à lui signifie que l'argument doit être donné au moins une fois.
- Le caractère $\|$ séparant des arguments indique que l'un des arguments au choix peut être l'argument de la commande.
- Un argument entre les caractères $<$ et $>$ indique que la description de cet argument est donnée dans les lignes suivantes.

La grammaire en BNF de l'interpréteur est donnée en Annexe B. Veuillez vous y référer pour la mise en œuvre de l'interpréteur.

5.2.1 Commandes relatives à la gestion de l'environnement de l'émulateur

Charger un programme

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

  `nom_du_fichier` : chemin du fichier objet à charger.
  `adresse` : une valeur hexadécimale non signée sur 32 bits représentant l'adresse d'implantation du premier segment mémoire. Cette valeur est arrondie au premier multiple de 1ko supérieur.

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Le fichier dont le nom est passé en paramètre doit être un fichier ELF relogeable. Les fichiers ELF et la notion de relocation sont décrits en détail à la section 4. Si le fichier ELF contient des sections de relocations `.rel.text` et/ou `.rel.data`, la relocation sera gérée automatiquement à la fin du chargement en mettant à jour les adresses relatives dans le codage des instructions et données.

Quitter le programme

- Nom de la commande : exit
- Syntaxe : `exit`
- Description : Cette commande provoque la sortie de la boucle infinie et la fin du programme.

Afficher la mémoire, les registres

- Nom de la commande : disp (display)
- Syntaxe :

```
affiche les données mémoire  :  disp mem <plage>+
affiche la carte mémoire     :  disp mem map
affiche des registres        :  disp reg <registre>+
```

- Paramètres :

  `plage` : deux entiers non-signés séparé par un " :"
  `registre` : une chaîne parmi {`$0 ...$31, $zero ...$ra, hi, lo, pc, all`} où `'all'` signifie tous les registres.

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description :
  Cette commande affiche sur la sortie standard le contenu de la mémoire dont les adresses sont spécifiées en paramètres, la carte mémoire du programme ou les valeurs des registres du MIPS émulé.
- Format de sortie
  L'affichage des données mémoire se fera par ligne. Chaque ligne doit débuter par l'adresse virtuelle affichée en hexadécimale sur 4 octets suivie de l'affichage de 16 octets maximum en hexadécimal. Chaque octet sera séparé d'un espace.
  L'affichage des registres se fera à raison de 4 registres par ligne. Chaque valeur de registre affichée en hexadécimal sur 4 octets sera précédée de son mnémonique (i.e., $at et non $1). Les valeurs seront séparées par une tabulation.
  L'affichage de la carte mémoire se fera suivant l'exemple ci-dessous. Chaque ligne donnera les caractéristiques d'un segment comprenant : le nom du segment, les permissions associées, son adresse implantation virtuelle (en hexadécimale sur 4 octets) et sa taille (en octet).

- Exemple :
  Affichage de la mémoire de la zone `0x0000bee0` à `0x0000beef`.

```
>> disp mem 0xbee0:0xbeef
0x0000bee0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

  Affichage de registres spécifiques

```
>> disp reg $at $zero $k1 $t8 $v1
   at :  00000000 zero :  00000000   k1 :  00000000   t8 :  00000000
   v1 :  00000000
```

  Affichage de tous les registres

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

  Affichage du map mémoire.

```
>> disp mem map
Virtual memory map (8 segments)
.text       r-x   Vaddr: 0x00005000   Size: 12 bytes
.data       rw-   Vaddr: 0x00006000   Size: 8 bytes
.bss        rw-   Vaddr: 0x00007000   Size: 0 bytes
[stack]     rw-   Vaddr: 0x0f7ff000   Size: 8388608 bytes
[vsyscall]  r-x   Vaddr: 0x0ffff000   Size: 4096 bytes
```

Afficher le code assembleur

- Nom de la commande : disasm (disassemble)
- Syntaxe : `disasm <plage>+`
- Paramètres :

  `plage` : <adresse> :<adresse> (deux entiers non-signés séparé par un " :")
  `plage` : <adresse>+<décalage non-signé> (deux entiers non-signés séparé par un "+")

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : La commande affichera une instruction par ligne avec comme premier élément l'adresse virtuelle de l'instruction `adresse` en hexadécimal puis le code binaire de l'instruction en hexadécimal puis le code assembleur en chaîne de caractères. Si la zone d'adresse couvre une zone plus grande que le segment `.text`, les zones mémoire en dehors du segment `.text` ne sont pas affichées. Dans la mesure du possible, les instructions, les sauts ou les branchements à une adresse repérée par une étiquette devront afficher l'étiquette correspondante.
- Exemple :
  désassemblage de 2 instructions à partir de l'adresse `0x5000`

```
>> disasm 0x5000+8
5000 :: 2002ABCD   ADDI $v0, $zero, 43981
5004 :: 3C010000   LUI $at, 0
```

  Désassemblage en dehors de la zone `.text` (qui commence ici en `0x5000`).

```
>> disasm 0x4FF8:0x500c
5000 :: 2002ABCD   J 50FC <procedure>
5004 :: 3C010000   LUI $at, 0
5008 :: AC226004   SW $v0, 24580($at)
```

Modifier une valeur en mémoire ou un registre

- Nom de la commande : set

```
modifie la mémoire   :  set mem <type> <adresse> <valeur>
modifie un registre  :  set reg <registre> <valeur>
```

- Paramètres :

  `type` : `byte` ou `word` (i.e., 8 bits ou 32 bits)
  `adresse` : une valeur hexadécimale non signée sur 32 bits
  `registre` : une chaîne parmi {`$0 ...$31, $zero ...$ra, hi, lo, pc`}.
  `valeur` : une valeur entière

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.

- Description : Cette commande écrit la quantité `valeur` dans la mémoire à l'adresse `adresse` ou dans le registre `registre`. Elle n'affiche rien sur la sortie standard.
- Exemple :
  Affecte la valeur 0xAAFFBBDD à 0x5000.
  `>> set mem word 0x5000 0xAAFFBBDD`
  Affecte -15 au registre $t1.
  `>> set reg $t1 -15`

5.2.2 Commandes relatives au test du programme en cours

Évaluer les valeurs en mémoire ou dans les registres

- Nom de la commande : assert
- Syntaxe :

```
test la valeur d'un registre          :  assert reg <registre> <valeur>
test la valeur d'un mot en mémoire    :  assert word <adresse> <valeur>
test la valeur d'un octet en mémoire  :  assert byte <adresse> <valeur>
```

- Paramètres :

  `adresse` : valeur entière non signée hexadécimale
  `valeur` : valeur entière signée

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : La commande `assert` vérifie les valeurs de mémoire et registre de l'émulateur. Cette commande est surtout utile pour automatiser les tests du programme assembleur.
- Exemple :
  Teste si le registre $v0 contient 0xFFFFFFFE
  `>> assert word $v0 -2`
  Teste si l'octet à l'adresse 0x500c contient 12
  `>> assert byte 0x500c 0xC`

Interrompre l'exécution d'un script

- Nom de la commande : debug
- Syntaxe : debug
- Valeur de retour : néant
- Description : Lorsqu'un script est exécuté, si la commande `debug` est rencontrée, l'exécution s'interrompt et la main est rendue à l'utilisateur qui peut interagir avec l'interpréteur.

Reprendre l'exécution d'un script

- Nom de la commande : resume
- Syntaxe : resume
- Valeur de retour : néant
- Description : Si l'interpréteur est en mode DEBUG alors la commande `resume` permet de reprendre l'exécution du script.

5.2.3 Commandes relatives à l'exécution du programme

Exécuter à partir d'une adresse

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

  `adresse` : valeur entière non signée hexadécimale
- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Cette commande charge PC avec l'adresse fournie en paramètre et lance le microprocesseur. Si le paramètre est omis, l'exécution commencera à la valeur courante de PC.

Exécution pas à pas

- Nom de la commande : step

  ```
  Exécute la prochaine instruction dans la procédure  :  step
  Exécute la prochaine instruction                    :  step into
  ```
- Description : Cette commande provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre PC puis rend la main à l'utilisateur. Dans le cas de `step`, si l'on rencontre un appel à une procédure, cette dernière s'exécute complètement jusqu'à l'instruction de retour incluse (par exemple `J $ra`). Par contre, dans le cas de `step into`, l'exécution s'interrompt juste après l'appel (i.e., dans la procédure appelée). La main est alors rendu à l'utilisateur sur l'instruction suivant l'appel.

Gérer les points d'arrêt

- Nom de la commande : break (breakpoint)

  ```
  ajout de points d'arrêt        :  break add <adresse>+
  suppression de points d'arrêt  :  break del <adresse>+|all
  affichage des points d'arrêt   :  break list
  ```
- Paramètres :

  `adresse` : une valeur hexadécimale non signée sur 32 bits
- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Cette commande met un point d'arrêt à (aux) l'adresse(s) fournie(s) en paramètre. Lorsque le registre PC sera égal à cette valeur, l'exécution sera interrompue avant l'exécution de l'instruction et l'utilisateur reprendra la main. Que ce soit pour l'ajout ou la suppression, break n'affiche rien sur la sortie standard. Si un point déjà présent est ajouté ou si un point non présent est supprimé, la commande est sans effet. Si une adresse est non atteignable (i.e., en dehors du segment `.text`), un message d'erreur est affiché. La commande `break` doit également permettre d'afficher l'ensemble des points d'arrêts enregistrés.
- Exemple :
  On ajoute deux points d'arrêt
  ```
  >> break add 0x5000 0x5010
  ```
  Affichage des points d'arrêt
  ```
  >> break list
  #0 0x00005000
  #1 0x00005010
  ```

Suppression de points d'arrêt et affichage

```
>> break del 0x00006000
>> break del 0x5000
>> break list
#0 0x00005010
```

Chapitre 6

À propos de la mise en œuvre

6.1 Méthode de développement

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

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

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

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

6.1.1 Notion de cycle de développement ; développement incrémental

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

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

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

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

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

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

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

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

6.1.2 Développement piloté par les tests

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

Pour chaque incrément, vous commencerez donc par écrire un jeu de tests avant même d'écrire la première ligne de code. Le jeu de test sera constitué de fichiers de script qui seront :

— des fichiers textes **.sml** comprenant le code des commandes de l'émulateur à exécuter (par exemple : charger un fichier elf, lancer le programme, vérifier une valeur en mémoire) ainsi que de variables qui, en particulier, indique si le test est censé échouer ou réussir.

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

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

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

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

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

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

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

6.1.3 À propos de l'écriture des jeux de tests

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

Voici quelques conseils pour l'écriture des jeux de tests.

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

6.1.4 Organisation interne d'un incrément

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

1. réfléchir à la structure de votre programme. Définir en commun les principales structures de données et les signatures et "contrats" (ou "rôle") des principales fonctions.
2. Autant que faire se peut, essayer d'organiser l'incrément... de façon incrémentale ! La notion de développement incrémental peut en effet s'appliquer récursivement : à l'intérieur d'un incrément, il est souhaitable de définir des étapes ou "sous-incréments" qui vous assurent que votre programme est "partiellement fonctionnel" le plus souvent possible.
3. Réfléchir à la répartition du travail. En particulier, éviter que la répartition ne bloque l'un d'entre vous si l'autre prend du retard.
4. Compiler très régulièrement. Il est particulièrement fâcheux de coder pendant plusieurs heures pour ne s'apercevoir qu'à la fin que la compilation génère des centaines d'erreurs de syntaxe. Un développeur expérimenté fait tourner le compilateur (presque) tout le temps en tâche de fond !

5. Exécuter souvent le jeu de tests, afin de mesurer l'avancement et de faire sortir le plus tôt possible les éventuelles erreurs.

6.2 Notions d'interpréteur, de syntaxe et de machine à états finis

Nous décrivons ici brièvement la manière dont l'utilisateur de l'émulateur va pouvoir interagir avec le programme simulé.

6.2.1 Interpréteur

Lorsque vous entrez des commandes dans un terminal, vous utilisez en réalité son interpréteur : en effet, il faut tout d'abord que le terminal ait une idée de ce que vous voulez lui faire faire en *interprétant* la chaîne de caractères entrée par l'utilisateur. Nous prendrons un exemple simple : imaginez que vous entriez la commande `"ls -l *.c"` (destinée, donc, à lister tous les fichiers de code source C présents dans le répertoire courant). L'interpréteur va tout d'abord découper la chaîne de caractères en trois chaînes : `"ls"`, `"-l"` et `"*.c"`. Chacune de ces chaînes sera appelée par la suite un *token*[3]. L'interpréteur va ensuite identifier le *type* du premier *token*. Dans notre cas, il va déterminer qu'il s'agit d'un exécutable à lancer. Le terminal va ensuite lancer cette commande en lui passant en paramètres les deux autres chaînes `"-l"` et `"*.c"`. À charge ensuite à l'exécutable `ls` de remplir sa fonction, en interprétant lui-même les paramètres que l'interpréteur du terminal lui aura transmis. La section 6.2.5 donne des indices pour réaliser l'extraction des tokens en langage C.

Le mécanisme de base d'un interpréteur est simple : c'est un bout de code qui passe son temps à afficher une *invite de commande*[4], à attendre que l'utilisateur daigne entrer une ligne de commande, la lire, puis, donc l'interpréter et, si elle est valide, l'exécuter. Et on recommence ainsi indéfiniment – mais on prévoit généralement une commande `exit` pour sortir de cette boucle infinie !

Nous utiliserons un interpréteur de commandes pour guider l'interaction de l'utilisateur avec l'émulateur. Cet interpréteur devra accepter une dizaine de commandes. Mais avant de vous les présenter, il nous faut d'abord vous expliquer comment *décrire* une commande.

6.2.2 Forme de Backus-Naur (BNF)

Une commande ne va accepter que certains *tokens* : des options ou des paramètres clairement définis, et dont l'enchaînement est lui aussi défini précisément. L'ensemble des enchaînements de *tokens acceptés* par une commande est appelé sa *syntaxe*.

Il serait vain de vouloir lister exhaustivement tous les enchaînements, potentiellement en nombre infini, acceptés par une syntaxe. On va plutôt chercher à *décrire* les enchaînements acceptés par la syntaxe d'une commande. Cette description d'une syntaxe s'effectue en utilisant la forme de Backus-Naur (BNF étant l'acronyme utilisé, depuis l'anglais). En conséquence, le langage BNF, qui sert à décrire une syntaxe et donc un autre langage, est parfois qualifié de méta-langage.

Le langage BNF est composé de différents éléments : les méta-symboles (qui sont des symboles propres au langage BNF), les *terminaux* (ce que l'utilisateur écrit) et les *non-terminaux* (les catégories du langage que l'on est en train de décrire). Ces éléments sont composés entre eux pour décrire des *règles de production* (qui décrivent comment les non-terminaux sont formés). L'ensemble des règles de production forme la syntaxe du langage que l'on est en train de décrire avec le langage BNF.

Les méta-symboles du langage BNF sont :

3. En bon français, un jeton.
4. Typiquement, dans un terminal : `bash-3.2$`. En anglais, on parle de *prompt*.

— : :=, qui signifie “est défini par” ;
— < et >, qui englobent un non-terminal ;
— . . ., qui signifie “jusqu’à” ;
— |, qui signifie “ou” ;
— ’, dont une paire englobe un terminal ;
— +, qui signifie “un ou plus” ;
— *, qui signifie “zéro ou plus”.

Un règle de production consiste à définir un non-terminal à l’aide de terminaux ou d’autres non-terminaux.

Par exemple, imaginons le langage suivant pour un interpréteur, composé de seulement deux commandes :

— `exit`, qui ne prend pas d’argument ;
— `show`, qui prend en argument un ou plusieurs intervalles d’adresses.

Les terminaux seront les chaînes de caractères `exit` et `show`, le marqueur hexadécimal (’0x’), les chiffres hexadécimaux, et le séparateur d’adresses (’-’). Les non-terminaux seront les commandes et les intervalles d’adresses, ainsi que les adresses exprimées en hexadécimal. En effet, nous définissons un intervalle d’adresses comme étant composé d’une adresse, suivie d’un séparateur d’adresses, et d’une seconde adresse. D’autre part, nous définissons une adresse comme étant composée du marqueur hexadécimal suivi d’un ou plusieurs chiffres hexadécimaux. La syntaxe BNF correspondant à cet exemple est donné Fig. 6.1. Notre petite syntaxe contient donc six règles de production, dont la première n’a pour objet que de lister les commandes permises.

```
<wait>  ::= "exit" epsilon
            "show" <inter>+ epsilon
            epsilon

<inter> ::= <blank>+ <adr1> ’-’ <adr2>

<adr1>  ::= 0x<hex>+

<adr2>  ::= 0x<hex>+

<hex>   ::= ’0’...’9’ | ’a’...’f’ | ’A’...’F’

<blank> ::= ’ ’|’\t’
```

Figure 6.1 – Syntaxe BNF de notre petit exemple.

Le langage BNF est un outil fondamental en théorie des langages. Il nous permet de dire que, par exemple, les commandes de la session d’exemple représentée dans la Fig. 6.2 sont valides et doivent être exécutées par l’interpréteur.

Pour aussi puissant que puisse être cet outil, il reste purement théorique et descriptif, et ne dit rien de la manière d’implanter effectivement une syntaxe en machine. Pour cela, nous devons utiliser un automate. Il existe plusieurs sortes d’automates en informatique, mais celui qui suffira amplement pour implanter le genre de syntaxe dont nous aurons besoin est appelé une machine à états finis. Il n’en reste pas moins que nous vous donnons la syntaxe complète de l’interpréteur sous forme de syntaxe BNF en annexe B, et qu’il vous appartiendra de construire et d’implanter l’automate correspondant.

```
interp $ show 0xdeadbeef-0xffffffff
interp $ show      0xdeadbeef-0xffffffff 0x0-0xbeef
interp $ show 0xdeadbeef-0xffffffff    0x0-0xbeef 0x1000-0x100f
interp $ exit
Exiting.
```

Figure 6.2 – Exemple de session d'un petit interpréteur. L'invite de commande est `interp $`. Les chaînes de caractères qui suivent l'invite de commande sont les commandes entrées par l'utilisateur. Les lignes 2 et 3 montrent comment passer plusieurs intervalles à la commande `show`, qui est effectivement censée pouvoir en accepter un ou plus (*cf.* l'utilisation du méta-symbole '+' dans la syntaxe BNF de la Fig. 6.1).

6.2.3 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. Nous allons donc construire un automate (une FSM) pour reconnaître si une entrée de l'utilisateur dans l'interpréteur correspond à quelque-chose de valide du point de vue de la syntaxe. Dans un premier temps, nous allons décrire ce qu'est une machine à états finis, et dans un second temps nous vous donnerons quelques pistes pour implanter efficacement et facilement une FSM en C [5].

Description d'une FSM

Une machine à états finis est définie par la donnée de ses états, et des transitions valides entre ses états. Lorsque, comme nous nous y emploierons, nous souhaiterons utiliser une FSM pour implanter une syntaxe, nous aurons besoin d'attacher à la description des transitions valides le motif ayant causé cette transition entre deux états. Les motifs seront les terminaux ou les non-terminaux de la syntaxe. Les états de notre FSM comprendront les non-terminaux de notre syntaxe, mais pas seulement.

Il faut bien comprendre qu'une syntaxe BNF représente en quelque sorte une description *positive* d'un langage. Elle ne dit rien *explicitement* des erreurs de syntaxe, elle ne fait finalement que les sous-entendre. Notre FSM, en tant qu'elle sera chargée de l'*implantation* de notre syntaxe BNF, représentera un dispositif opérationnel qui devra, à ce titre, être capable d'indiquer à l'utilisateur s'il a commis une erreur de syntaxe et, le cas échéant, laquelle. Il conviendra donc d'ajouter à notre FSM un état correspondant aux erreurs de syntaxe. Une erreur de syntaxe, *in fine*, sera comprise comme une transition invalide depuis un état qui n"autorise pas cette transition. Pour chaque état, nous désignerons par le même nom de *défaut* un motif invalide, qui provoque donc une transition vers l'état d'erreur.

Il est donc possible qu'une entrée de l'utilisateur produise une erreur de syntaxe. Mais, et puisque personne n'est à l'abri d'un coup de chance, il n'est pas impossible non plus que cette entrée soit valide et doive être effectivement exécutée par l'interpréteur. Si vous observez bien la Fig. 6.1, et que vous vous souvenez que les états de notre FSM vont inclure les non-terminaux de notre syntaxe BNF, il convient de s'interroger sur la signification de l'état correspondant au non-terminal <*wait*>. Cet état est en réalité celui d'une *attente* d'une entrée de la part de l'utilisateur. Et lorsqu'aucun motif n'est attaché à une transition, c'est donc qu'il s'agit d'une transition inconditionnelle.

Pour l'exemple, la FSM correspondante est représentée dans la Fig. 6.3.

5. Tellement facilement d'ailleurs, qu'il ne serait pas impossible que vous n'ayez pas besoin de spécifier votre FSM outre mesure, ou que vous soyez en mesure d'écrire son code au fil de l'eau !

6.2.4 Implantation d'une FSM en C

Pour illustrer la mise en œuvre d'un automate à états finis, prenons l'exemple d'un programme qui accepte en entrée une chaîne de caractères et qui doit déterminer si cette chaîne est un entier décimal, octal ou hexadécimal. La première phase du logiciel est de vérifier que la ligne donnée en entrée contient uniquement des éléments acceptés par le langage et de les identifier. Par exemple, il faut pouvoir reconnaître que la chaîne de caractères `0123` est une valeur octale et que `0x123` est une valeur hexadécimale. 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 est complètement déterministe et il est possible de connaître la composition de chaque élément terminal du langage. Par exemple, on sait qu'une valeur hexadécimale est toujours préfixée par `0x` puis un certain nombre de caractères $\in$ `[0,9]` $\cup$ `[a,b,c,d,e,f]` alors qu'une valeur octale n'est composée que de chiffres inférieurs à 8. En tournant les choses de cette façon le problème devient de trouver des *motifs* de caractères dans le texte et non plus des mots prédéfinis. Un autre problème vient du fait que les motifs peuvent partager des caractéristiques communes qui impliquent qu'il faut avoir lu un certain nombre de caractères avant de reconnaître un motif (p.ex. : tant que l'on a pas lu le 'x' après un zéro on ne sait pas si on lit un nombre hexadécimal ou octal).

Une façon d'aborder le problème est de représenter les motifs et leur parcours par un automate à états. Succinctement, l'automate est composé d'états qui dans notre cas représentent la catégorie courante de la chaîne de caractères lue et de transitions entre états étiquetés par les caractères que l'on va lire. L'exemple de la figure 6.4 montre comment on peut représenter un automate faisant la différence entre nombres décimaux, octaux ou hexadécimaux.

Cet automate lit les caractères un par un jusqu'à arriver à l'état terminal ou erreur. Ainsi, en prenant l'exemple de la chaîne `0567`, l'automate passe successivement par `INIT`, `pref hexa`, et reste dans `octal` jusqu'à la fin donnant ainsi la catégorie de la chaîne. La figure 6.5 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 leur motifs, de construire l'automate, et de prévoir les fichiers de tests, avant de commencer à coder...Il est possible que certains traitements à effectuer dans certains états soient réutilisables, n'hésitez donc pas à fragmenter votre code en fonctions.

6.2.5 Découper une chaîne de caractères en token, `strtok`

Un interpréteur lit généralement les commandes ligne par ligne. C'est à dire que chaque ligne est interprétée comme contenant une (et une seule) commande complète [6] autrement dit, le caractère de retour à la ligne `'\n'` marque la fin d'une commande. Si l'on considère que chaque *token* est séparé les

6. Dans la réalité une ligne peut contenir plusieurs commandes (par exemple, `"ls -l; cd .."` liste le contenu d'un répertoire puis va dans le répertoire parent) et dans certains interpréteurs une commande peut occuper plusieurs lignes.

uns des autres par des caractères particulier (dans notre cas des blancs — espace, tabulation, retour de ligne...) alors le langage C fourni un mécanisme implanté dans la la fonction standard `strtok` qui permet de parcourir la chaine de caractères token par token. L'extrait de code ci-dessous illustre son utilisation avec une chaîne de caractères.

La chaîne de caractères `"Dupond 20 \t 76"` est séparée en éléments délimités par les espaces et tabulations. Chaque appel à `strtok()` renvoie le prochain élément. Des informations complètes sont accessibles dans le `man` de `strtok`.

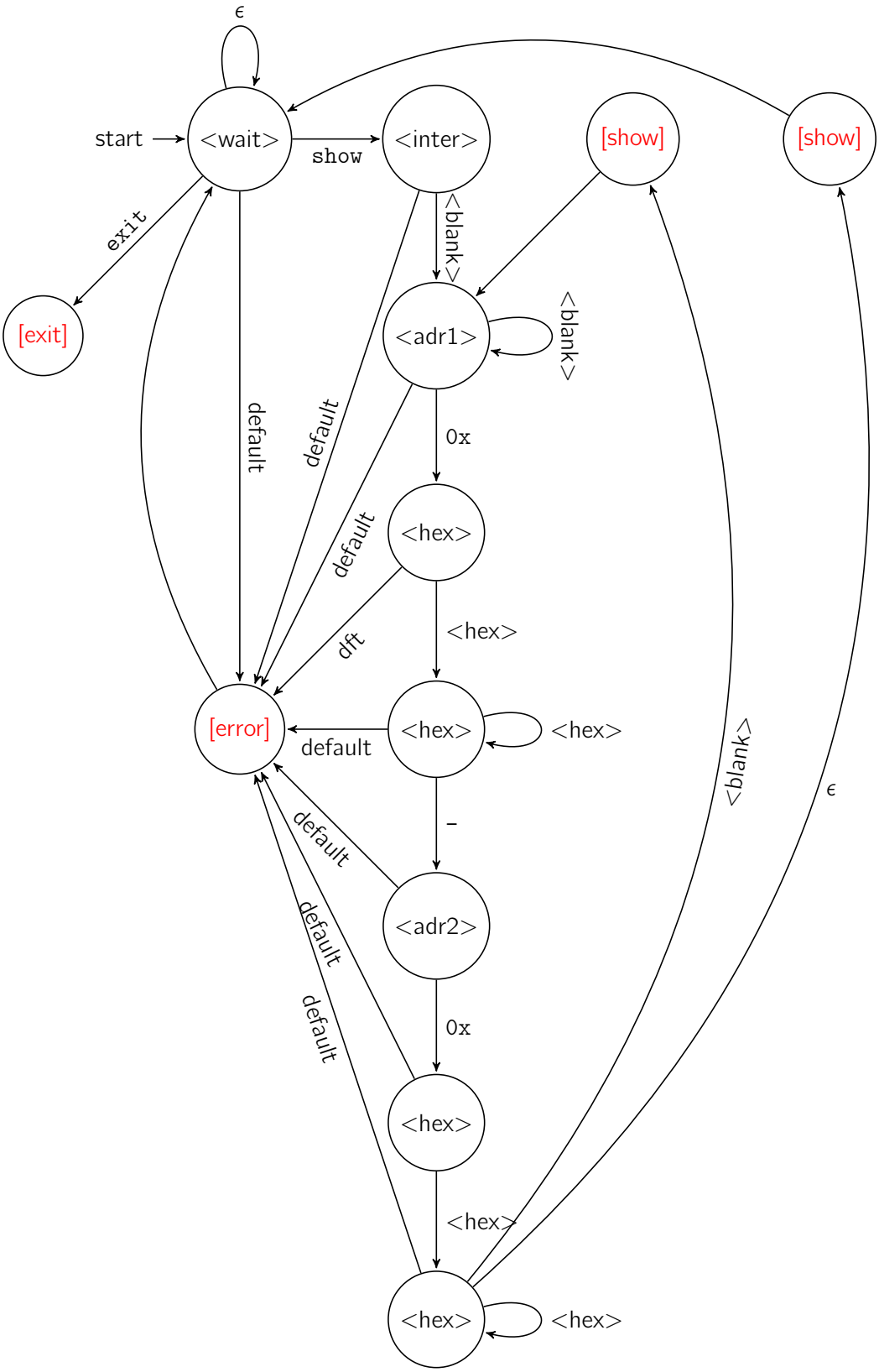


Figure 6.3 – La FSM complète correspondant à notre petit exemple. Pour des raisons évidentes de place manquante, elle est discutée dans le texte.

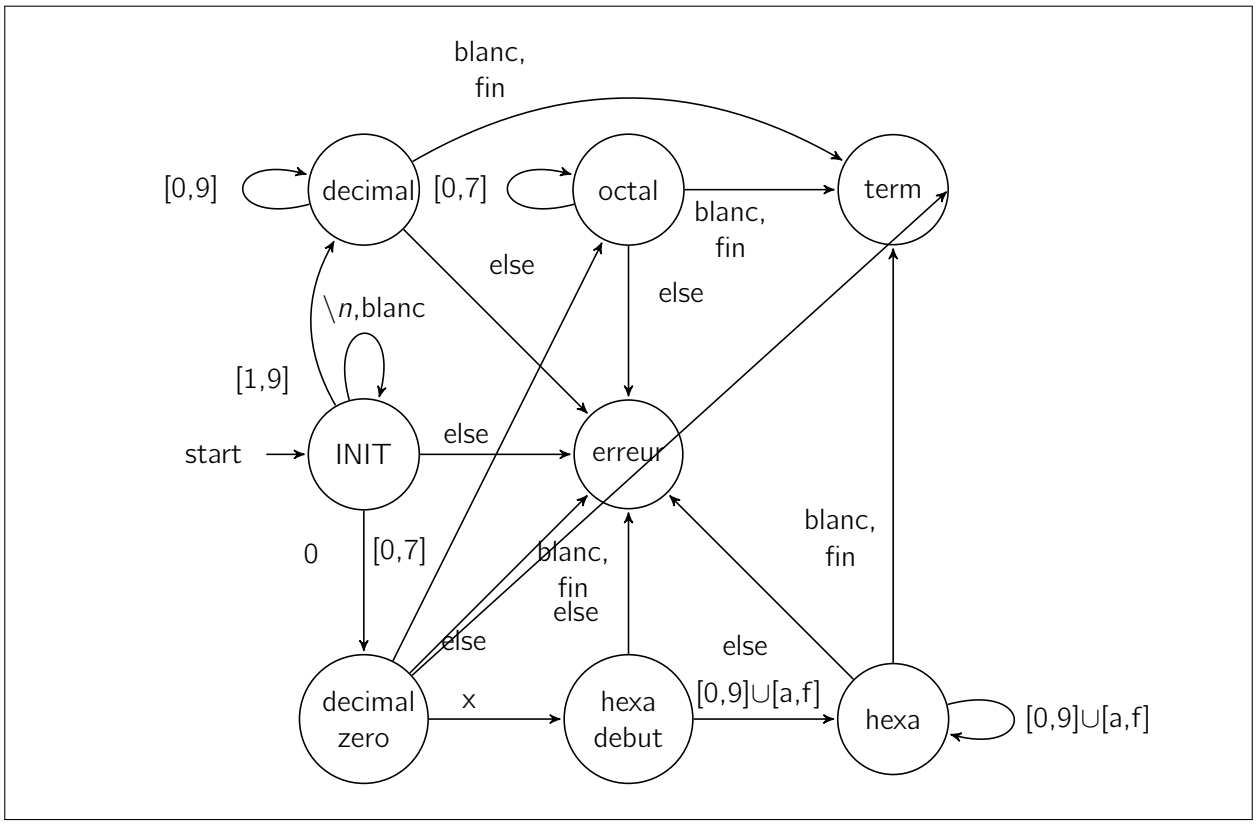


Figure 6.4 – Exemple d'automate faisant la différence entre une valeur décimale, octale et hexadécimale

```
/* 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 6.5 – Exemple de traduction en C de l'automate de la figure 6.4

```
/* 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 6.6 – Extrait de code illustrant l'usage de `strstok()`

Chapitre 7

Organisation du Projet

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.

7.1 Objectif général :

À la fin de ce projet vous devrez avoir réalisé un programme capable d'émuler l'execution d'un fichier objet elf pour l'architecture MIPS32. L'interaction avec cet émulateur se fera à travers un interpreteur de commandes. L'émulateur sera appelé en tapant sous Linux la commande :

```
emul-mips
```

ou

```
emul-mips script_filename
```

où `script_filename` est le nom d'un fichier de commandes à exécuter.

7.2 Étapes de développement du programme

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

1. L'émulateur et son interpréteur de commandes : À la fin de cette étape, vous devrez avoir l'environnement de l'émulateur initialisé et un interpréteur exécutant des commandes simples.
2. Désassemblage des instructions : À la fin de cette étape, vous devrez avoir un émulateur qui est capable de désassembler les instructions d'un fichier objet .
3. Simulation : À la fin de cette étape, vous devrez avoir un programme capable d'exécuter le code machine pas à pas ainsi qu'en utilisant des points d'arrêt.
4. Relocation et appels systèmes : À la fin de cette étape, votre émulateur devra être capable de charger et d'exécuter du code relogeable ainsi que de gérer des appels systemes pour certaines opérations d'entrées/Sorties.

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

7.3 Bonus : extensions du programme

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

Bibliographie

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Annexe A

ELF : Executable and Linkable Format

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

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

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

A.1 Fichier objet au format ELF

Les fichiers ELF sont composés d'un ensemble de sections (éventuellement vides) comme indiqué par la figure A.1.

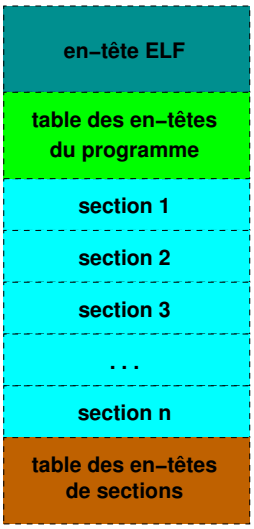


Figure A.1 – Structure d'un fichier ELF

:

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

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

Les seules sections qui contiennent des données sont les sections :

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

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

A.2 Structure générale d'un fichier objet au format ELF et principe de la relocation

Un fichier objet à lier au format ELF est formé d'un en-tête donnant des informations générales sur la version, la machine, etc., puis d'un certain nombre de pointeurs et de valeurs décrits ci-dessous. Ce que nous appelons ici pointeur est en fait une valeur représentant un déplacement en nombre d'octets par rapport au début du fichier. Les tailles, elles aussi, sont exprimées en nombre d'octets. La figure A.2 donne une idée de la structure d'un fichier au format ELF. Le fichier est constitué d'un *en-tête* donnant les caractéristiques générales du fichier, puis d'un certain nombre de *sections* contenant différentes formes de données, ces sections étant détaillées par la suite (par exemple, section "`.text`", section "`.data`", section des "relocations en zone text", section de la table des symboles, etc).

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

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

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

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

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

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

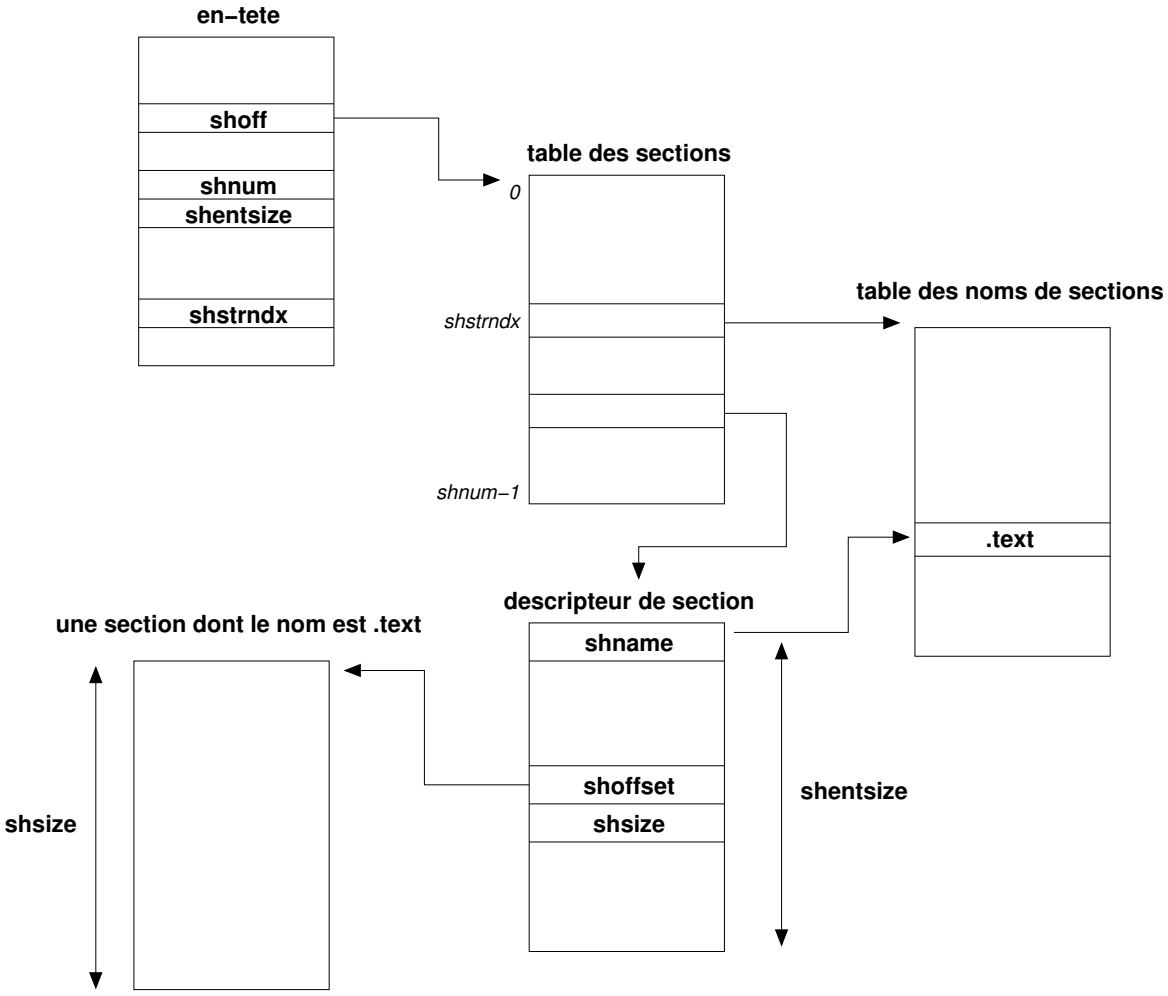


Figure A.2 – Structure d'un fichier relogeable au format ELF

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

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

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

A.3 Exemple de fichier relogeable

Pour étudier le format ELF en détaillant le format de chacune des sections considérées dans le projet, considérons le programme en langage d'assemblage `reloc_miam.s` donné Figure A.3.

```
# allons au ru
.set noreorder

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

.bss
tableau: .space 16

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

Figure A.3 – Programme assembleur nécessitant une relocation.

La commande `mips-as reloc_miam.s -o reloc_miam.o` produit un fichier objet `reloc_miam.o`

1. On verra en effet en sous-section A.4.6 que tous les symboles globaux doivent être rassemblés à la fin de la table des symboles.

dont nous donnons en figure A.4 le contenu affiché par la commande Unix `od -t xC reloc_miam.o`.

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

Figure A.4 – Résultat de `od -t xC reloc_miam.o`.

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

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

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

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

There are no section groups in this file.

There are no program headers in this file.

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

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

There are no unwind sections in this file.

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

No version information found in this file.
```

A.4 Détail des sections

A.4.1 L'en-tête

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

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

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

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

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

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

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

Figure A.5 – Table des section : résultat de `readelf -S reloc_miam.o`.

A.4.2 La table des noms de sections (`.shstrtab`)

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

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

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

C'est dans cette table que `readelf` va chercher les noms des sections pour remplir la table donnée figure A.5. Le format ELF impose certaines règles à respecter. La première chaîne doit être la chaîne vide (0x00). Chaque chaîne doit se terminer par le caractère de code ASCII 0 (comme en "C").

A.4.3 La section table des chaînes (`.strtab`)

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

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

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

A.4.4 La section `.text`

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

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

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

```
Disassembly of section .text:

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

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

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

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

pour `lui` devrait être les 16 bits de poids fort de `lunchtime` et la valeur numérique pour `lw` devrait être les 16 bits de poids faible de `lunchtime`. Cependant, la valeur de `lunchtime` n'est pas connue. En effet, `lunchtime` est une adresse est cette adresse ne sera connue que lors du chargement final en mémoire, les 16 bits sont donc à zéro ou une valeur temporaire (ici 4 pour lw) en attendant d'être replacés par une valeur lors du chargement grâce à une donnée de relocation `rel.text` qui est associée à cette instruction (cf. section A.4.7).

A.4.5 La section `.data`

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

```
0000  00000008 0000000c 00000004
```

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

A.4.6 La section table des symboles

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

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

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

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

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

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

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

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

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

A.4.7 Les sections de relocation

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

Format de la table de relocation

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

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

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

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

Modes de calcul de la relocation

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

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

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

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

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

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

Les adresses finales des sections `.text`, `.data`, `.bss` sont normalement déterminées lors de l'édition de liens. Dans notre simulateur, elles sont calculées lors du chargement des sections en mémoire suivant les contraintes définies section 5.2.1.

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

- `R_MIPS_32` (constante 2) : la valeur mise à l'adresse P vaut $V = S + A$. Ce mode sert pour les adressages directs.
- `R_MIPS_26` (constante 4) : le calcul se décompose en plusieurs étapes :
  - calcul de l'adresse de saut (comme décrit section 3.4.2) : $(P\&0xf0000000)+S$
  - ou logique avec A décalé de 2 à gauche : $(A << 2) \mid ((P\&0xf0000000)+S)$
  - résultat décalé de 2 à droite : $V = ((A << 2) \mid ((P\&0xf0000000)+S)) >> 2$

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

La section de relocation `.rel.data`

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

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

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

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

Le champ `type` vaut 2, ce qui signifie que le mode de relocation est `R_MIPS_32`. Il faut donc déterminer A et S. Le champ `sym` vaut également 2 ce qui signifie qu'il s'agit de l'adresse de la zone `.data`. On a donc S = adresse de `.data` = $0x0$. A est la valeur du champ présent dans le code à l'adresse de l'instruction. Le champ `r_offet` vaut 0x8, si on se réfère à la section A.4.5, la valeur sur 32 bits à l'adresse 8 de la section data est 0x00000004. Nous avons donc $A = 0x4$ et $V = S + A = 0x0 + 0x4 = 0x4$. On va trouver que le contenu de P sera égal à l'adresse finale de la zone `.data` + 4. C'est normal, on fait ici référence à `lunchtime`, qui est le deuxième mots de 32 bits de la zone data.

La section de relocation `.rel.text`

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

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

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

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

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

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

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

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

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

A.4.8 Autres sections

Il reste trois sections à décrire :

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

Annexe B

Grammaire des commandes de l'émulateur

La grammaire des commandes de l'interpreteur doit respecter les règles de production ci-dessous :

```
<commande> ::=  <assert> | <break> | "debug" | <disasm> | <disp> | "exit" | <help> |
                <load> | "resume" |<run> | <set> | <step>

<assert> ::= "assert" "reg" <reg> <integer32>
<assert> ::= "assert" "word" <@> <integer32>
<assert> ::= "assert" "byte" <@> <integer8>

<break> ::= "break" "add" <@>+
<break> ::= "break" "del" <@>|"all"
<break> ::= "break" "list"

<disasm> ::= "disasm" <range>

<disp> ::= "disp" "mem" "map"|<range>+
<disp> ::= "disp" "reg" "all"|<reg>+

<help> ::=  "help" "assert" | "break" | "debug" | "disasm" | "disp" | "exit" | "help" |
                   "load" | "resume" | "run"  | "set" | "step"

<load> ::= "load" <filename> [<@>]

<run> ::= "run" [<@>]

<set> ::= "set" "mem" (<@> (("byte" <integer8>) | ("word" <integer32>)))+
<set> ::= "set" "reg" (<reg> <integer32>)+

<step> ::= "step" ["into"]
```

Les symboles terminaux sont

```
<reg>       ::= un parmi tous les mnemoniques et numéros de registre du MIPS
<integer32> ::= entier signé sur 32 bits
<integer8>  ::= entier signé sur 8 bits
<integer>   ::= integer32|integer8
<@>         ::= entier non-signé sur 32 bits représenté en hexadécimal
<range>     ::= @+integer | @:@
```

Annexe C

La `libc` ou comment créer un fichier objet MIPS sans connaître l'assembleur...

À de rares exceptions prês, l'assembleur n'est plus un langage utilisé par les développeurs de logiciels. En effet, les développeurs actuels utilisent des langages de plus haut niveau tels que le C pour écrire des programmes qu'ils peuvent compiler pour un ensemble d'architectures cibles (c'est-à-dire, dans notre cas particulier le MIPS 32) [1]. Autrement dit, il est possible d'écrire du code dans votre langage préféré pour une architecture MIPS sans avoir à apprendre l'assembleur !

Avant de nous jeter des pierres pour avoir osé vous faire faire de l'assembleur, sachez qu'il y a quand même un sérieux désavantage. Pour pouvoir utiliser du code écrit en C sur votre émulateur, il faut que votre programme charge en mémoire une `libc` [2]. En effet, lorsqu'il vous prend la fantaisie d'appeler `printf` pour afficher la valeur, par exemple, d'un entier, vous n'écrivez pas le *code* de `printf`. Vous vous contentez d'inclure `stdio.h` pour que tout se passe bien. Mais le code de `printf` est ailleurs. Nous ferons l'hypothèse que ce code se trouve dans une zone d'instructions machine appelée {`libc`}`.text`. Ce *segment* en mémoire sera réputé contenir la plupart des fonctions usuelles que vous aviez l'habitude d'utiliser [3]. Si les fonctions de la librairie standard ont besoin elles-mêmes de données, elles seront accédées dans une zone mémoire appelée {`libc`}`.data`.

Que se passe t il lorsque votre code appelle une fonction d'affichage (p.ex., `printf`) ? Cette fonction est située quelque-part dans la zone `libc.text` donc en définitive le programme fera un saut de la zone `.text` à la zone {`libc`}`.text` pour exécuter le code de `printf`. Il se trouve que printf ne fait en réalité que faire appel à une autre fonction permettant d'*écrire* sur un dispositif *matériel* (en l'occurrence : l'écran). En tant que simple *utilisateur* de la machine, vous n'avez en réalité pas le droit de pratiquer des opérations directement sur le *matériel*. Pour ce faire, vous êtes obligés de procéder à ce que l'on appelle un *appel système*. Les fonctions de la zone {*libc*}*.text* telles que `printf` passent typiquement leur temps à mettre en forme les données de l'utilisateur pour les transmettre au matériel à travers des appels au système.

Pour résumer, si votre fonction `main` fait un appel à `printf`, *in fine* cette dernière fonction ne fait que mettre en forme ce qu'il faut écrire sur un dispositif matériel, et l'écriture des caractères issus de *printf* sur le dispositif matériel qu'est l'écran se fait à travers des *appels système*. Car, sur une machine bien policée, seul le *système* est autorisé à accéder au matériel, pour le bien de tous et de chacun. La zone mémoire dans laquelle les *appels système* se trouvent est appelée `[vsyscall]`.

En clair, pour faire fonctionner l'émulateur avec du code compilé à partir d'un programme écrit en langage C, votre programme devra :

1. charger la zone `[vsyscall]` avec le code permettant d'effectuer les appels systèmes.
2. charger une `libc` en mémoire dans la zone `[lib]` (cf. section 2.3) et la reloger
3. charger votre programme et le reloger. Dans cette phase, il faudra mettre en place un mécanisme de résolution de symboles non définis. En effet, lorsque le programme sera compilé les fonctions

1. Il faut quand même souligner que le compilateurs, notamment GCC, transforme souvent le code de haut niveau en assembleur pour ensuite le transformer en code machine. L'assembleur est donc extrêmement utilisé... mais peu par les humains...
2. http://www.gnu.org/software/libc/
3. Hormis le couple infernal `malloc`/`calloc` et `free`, pour des raisons qui devraient vous paraître bien plus claire à la fin de ce projet.

telles que `printf` seront des symboles inconnus (c'est-à-dire, définis dans un autre fichier). Pour chaque symbole non-défini, l'émulateur devra chercher dans les tables des symboles du segment {`llibc`}`.text` si celui-ci s'y trouve.

4. lancer l'exécution et regarder, les yeux ébahis, le programme interagir avec vous...

Il ne reste plus qu'à écrire une `libc` ! Fort heureusement, la génération de cette librairie sort du cadre du projet. Il vous sera donc généreusement fourni une libc sous forme de fichier objet directement chargeable dans votre émulateur. Cette bibliothèque donne accès aux fonctions `printf, scanf, getchar, exit, strlen, strcpy, puts, gets`. Attention celles sont restreintes à la manipulation exclusive des entiers et des caractères.

Annexe D

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].

D.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

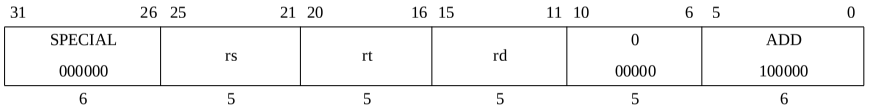
Figure D.1 – Octet

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 D.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
```

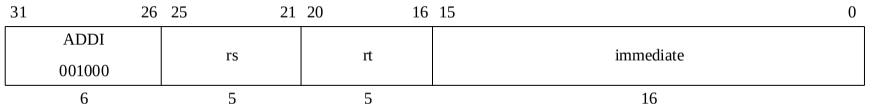
Exceptions:

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
```

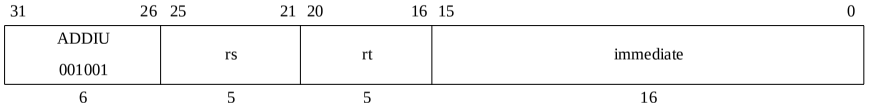
Exceptions:

Integer Overflow

Programming Notes:

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

Add Immediate Unsigned Word — ADDIU



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

Purpose:

To add a constant to a 32-bit integer

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

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

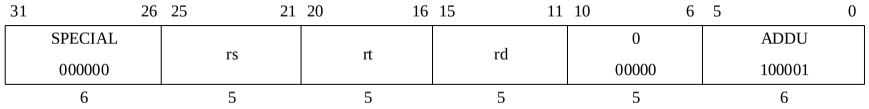
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

Add Unsigned Word — ADDU



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

Purpose:

To add 32-bit integers

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* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

Exceptions:

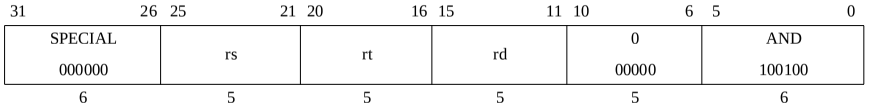
None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

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:

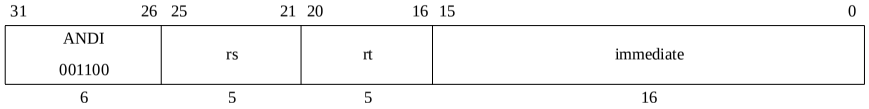
```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

And Immediate

ANDI



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

Purpose:

To do a bitwise logical AND with a constant

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

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

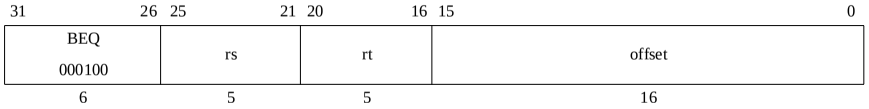
Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

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:

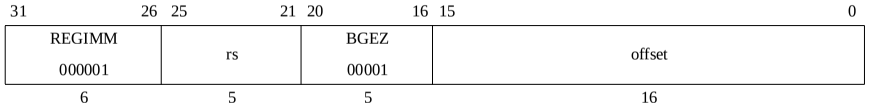
None

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 or Equal to Zero — BGEZ



Format: `BGEZ 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 or equal to zero (sign bit is 0), 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
```

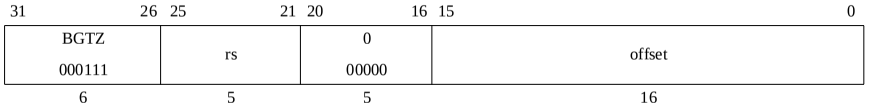
Exceptions:

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 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
```

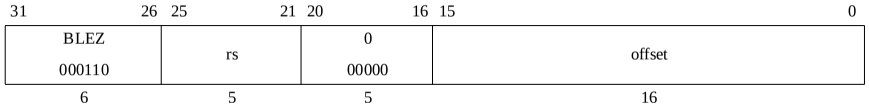
Exceptions:

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
```

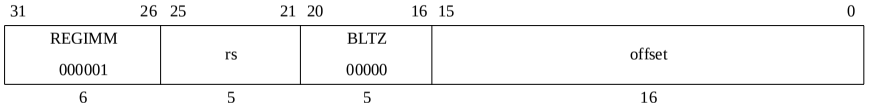
Exceptions:

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 Less Than Zero — BLTZ



Format: `BLTZ 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 zero (sign bit is 1), 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
```

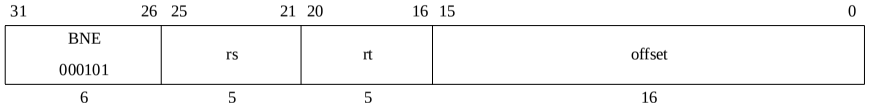
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls 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
```

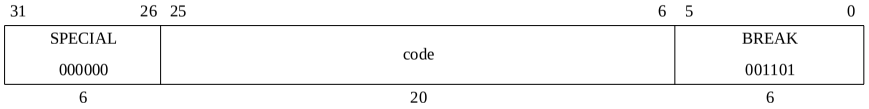
Exceptions:

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.

Breakpoint — BREAK



Format: `BREAK` **MIPS32**

Purpose:

To cause a Breakpoint exception

Description:

A breakpoint 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

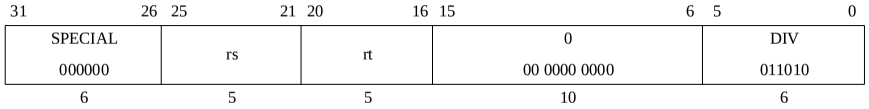
Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

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:

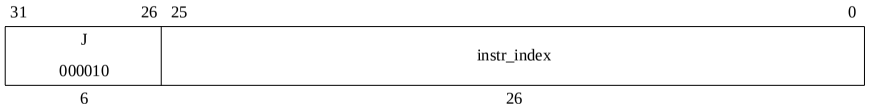
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor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

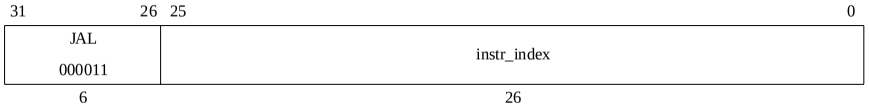
None

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:

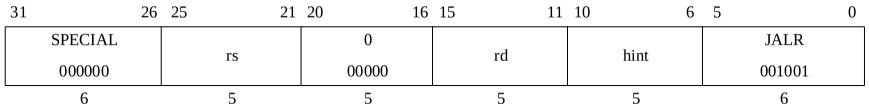
None

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 and Link Register JALR



Format: `JALR rs (rd = 31 implied)` **MIPS32**
`JALR rd, rs` **MIPS32**

Purpose:

To execute a procedure call to an instruction address in a register

Description: `GPR[rd] ← return_addr, PC ← GPR[rs]`

Place the return address link in GPR *rd*. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

For processors that do not implement the MIPS16e ASE:

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

For processors that do implement the MIPS16e ASE:

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

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

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

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.

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:

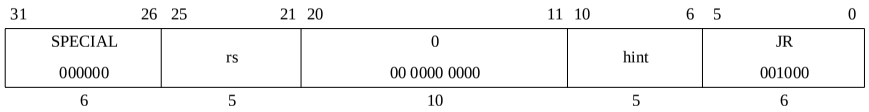
```
I:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

Exceptions:

None

Programming Notes:

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR *rd*, if omitted in the assembly language instruction, is GPR 31.



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:

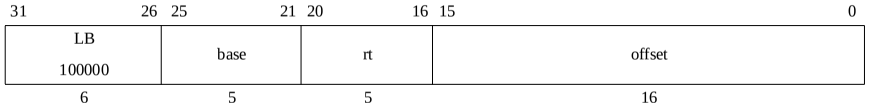
```
I:  temp ← GPR[rs]
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

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: `LB rt, offset(base)` **MIPS32**

Purpose:

To load a byte from memory as a signed value

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

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

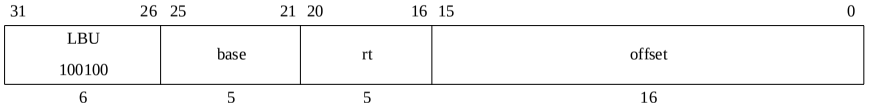
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr(PSIZE-1..2) || (pAddr(1..0) xor ReverseEndian(2))
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr(1..0) xor BigEndianCPU(2)
GPR[rt]← sign_extend(memword(7+8*byte..8*byte))
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Load Byte Unsigned — LBU



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

Purpose:

To load a byte from memory as an unsigned value

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

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

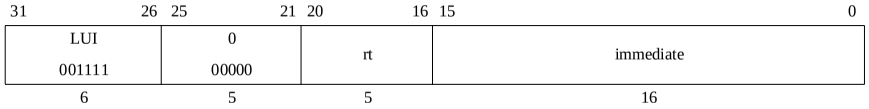
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_1..0 xor BigEndianCPU^2
GPR[rt]← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Load Upper Immediate — LUI



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

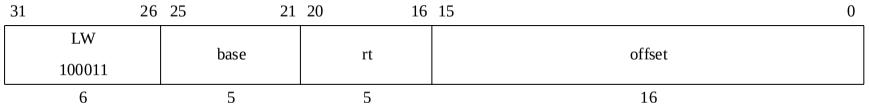
Operation:

```
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.

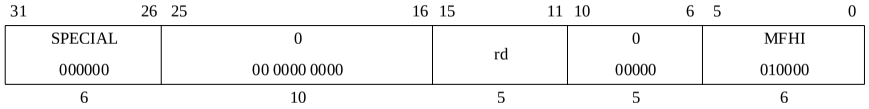
Operation:

```
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
```

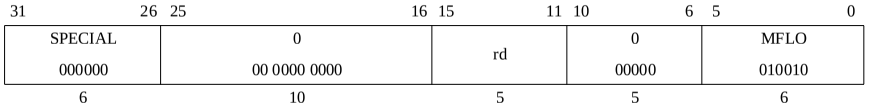
Exceptions:

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
```

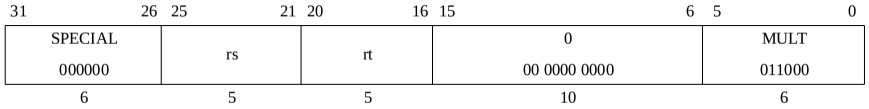
Exceptions:

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

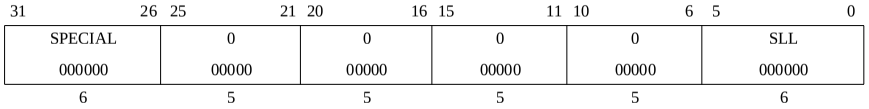
Programming Notes:

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

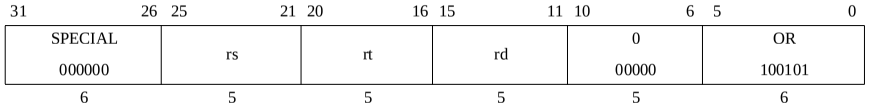
Exceptions:

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

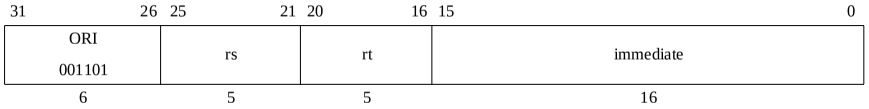
Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Or Immediate — ORI



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

Purpose:

To do a bitwise logical OR with a constant

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

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

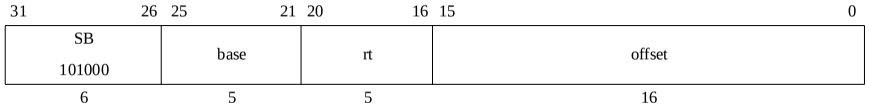
Operation:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None

Store Byte — SB



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

Purpose:

To store a byte to memory

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

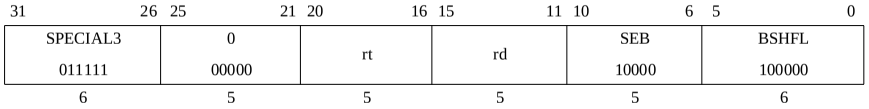
Operation:

```
vAddr     ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr     ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
bytesel   ← vAddr_1..0 xor BigEndianCPU^2
dataword  ← GPR[rt]_31-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Sign-Extend Byte — SEB



Format: `seb rd, rt` **MIPS32 Release 2**

Purpose:

To sign-extend the least significant byte of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← SignExtend(GPR[rt]7..0)`

The least significant byte from GPR *rt* is sign-extended and stored in GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

```
GPR[rd] ←sign_extend(GPR[rt]7..0)
```

Exceptions:

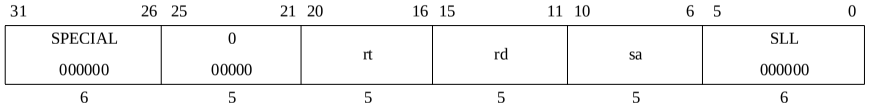
Reserved Instruction

Programming Notes:

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
`ZEB rx,ry`	Zero-Extend Byte	`ANDI rx,ry,0xFF`
`ZEH rx,ry`	Zero-Extend Halfword	`ANDI rx,ry,0xFFFF`

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 || 0^s
GPR[rd]← temp
```

Exceptions:

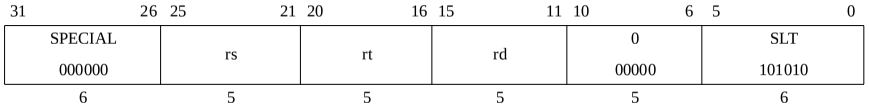
None

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

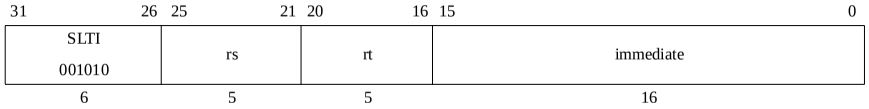
Operation:

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

Exceptions:

None

Set on Less Than Immediate SLTI



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

Purpose:

To record the result of a less-than comparison with a constant

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

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

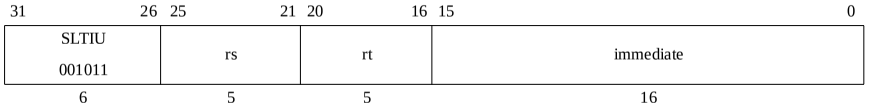
Operation:

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

Exceptions:

None

Set on Less Than Immediate Unsigned — SLTIU



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

Purpose:

To record the result of an unsigned less-than comparison with a constant

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

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

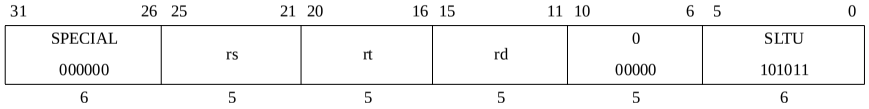
Operation:

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

Exceptions:

None

Set on Less Than Unsigned — SLTU



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

Purpose:

To record the result of an unsigned less-than comparison

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

Compare the contents of GPR *rs* and GPR *rt* as unsigned 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

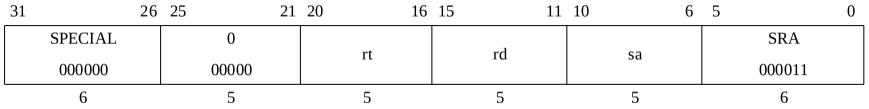
Operation:

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

Exceptions:

None

Shift Word Right Arithmetic SRA



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

Purpose:

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

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

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

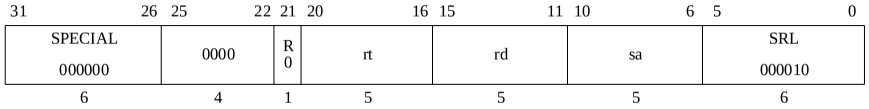
None

Operation:

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

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:

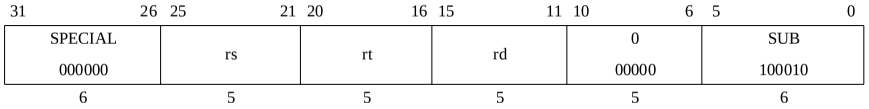
```
s      ← sa
temp   ← 0s || GPR[rt]31..s
GPR[rd]← temp
```

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:

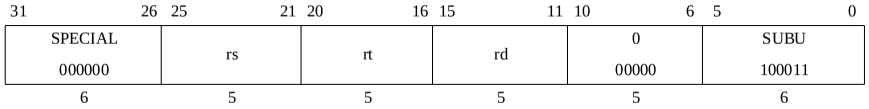
Integer Overflow

Programming Notes:

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

Subtract Unsigned Word

SUBU



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

Purpose:

To subtract 32-bit integers

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* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← temp
```

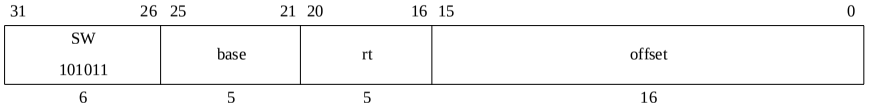
Exceptions:

None

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

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.

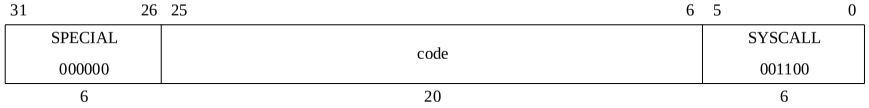
Operation:

```
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

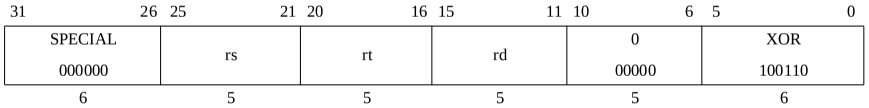
Operation:

```
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