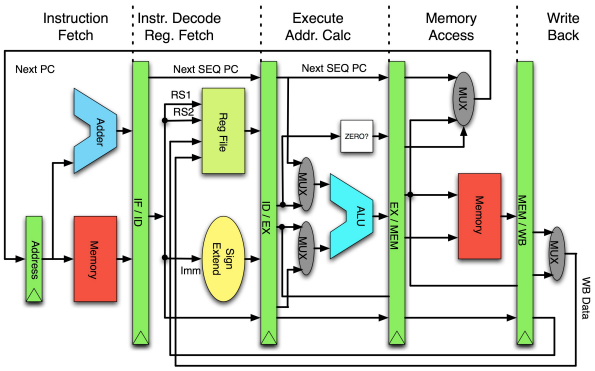
Grenoble INP ensimag

Ensimag — Printemps 2016

Projet Logiciel en C

Simulateur de microprocesseur MIPS



Auteurs : des enseignants actuels et antérieurs du projet C.

Table des matières

Chapitre 1

Présentation générale du projet

L'objectif de ce projet informatique est de concevoir puis implémenter en langage C, sous Linux, un simulateur de microcontrôleur MIPS. Nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions du MIPS.

Comme son nom l'indique, un simulateur (ou émulateur) est un logiciel capable de reproduire le comportement de l'objet simulé, dans le cas qui nous intéresse la machine MIPS lorsqu'elle exécute un programme. Il définit une mémoire et des registres similaires à ceux du processeur, puis doit réaliser l'évolution de leur état selon les spécifications définies par le programme.

Les mêmes résultats doivent être obtenus que lors de l'exécution sur un MIPS réel, mais pas forcément de la même façon : c'est le principe du *faire semblant* (par opposition au *faire comme*). Un tel logiciel permet par exemple de vérifier et déboguer des programmes assembleurs destinés à une machine équipée d'un processeur MIPS, sans avoir réellement ce processeur.

Le simulateur doit permettre de lire des programmes pour l'architecture MIPS décrite au chapitre 2 et de simuler leur exécution. Ces programmes seront écrits dans le langage assembleur décrit au chapitre 3 à partir de l'assembleur `as` de `GNU`. Il devront ensuite être *assemblés*, c'est-à-dire codés en binaire et stockés dans un fichier binaire dit *objet*. Dans ce projet, l'assembleur `mips-elf-as` sera utilisé et les fichiers objets produits seront au format ELF (relogeable ou exécutable).

Dans ce projet, on n'utilisera qu'un sous-ensemble des instructions, décrit dans le chapitre 4, qui explique également le fonctionnement du codage d'une instruction et d'un programme. Lors du chargement d'un programme en mémoire, une phase importante est la *relocation* des symboles, expliquée dans le chapitre 5.

L'interface utilisateur du simulateur sera réalisée sous forme d'un shell Linux, avec les commandes entrées au clavier et affichage dans la console. Le chapitre 6 présente ces spécifications attendues, caractéristiques et commandes du simulateur. Enfin, le chapitre 7 décrit des modules fournis et le travail demandé, ainsi que quelques conseils pour la réalisation de votre projet.

Le sujet est complété par un ensemble d'annexes apportant des exemples ou compléments techniques, la documentation des modules de code fournis et les spécifications, issues du manuel, des instructions du MIPS.

Chapitre 2

Description du microprocesseur MIPS

Cette section contient une description de la machine MIPS : mémoires, registres et modes d'adressage et fonctionnements des entrées/sorties et exceptions.

2.1 Définitions et notations

Commençons par rappeler quelques définitions et notations utiles. Une excellente référence sur l'arithmétique binaire et sur le MIPS peut être trouvée ici [2].

2.1.0.0.1 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. Par exemple voici l'octet représentant la valeur décimale 110 (0x6E en notation hexadécimale).

0_7	1_6	1_5	0_4	1_3	1_2	1_1	0_0

2.1.0.0.2 Représentation hexadécimale

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, $j \in ([0 \ldots 9, A \ldots F]$). Par exemple, la valeur de l'octet vu ci-dessus s'écrit 0x6E en hexadécimal. Pour un demi-mot ou un mot, on aura respectivement 4 ou 8 chiffres hexadécimaux, chacun représentant 4 bits.

2.1.0.0.3 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, 0x00 à

0xFF en notation hexadécimale. Un entier codé sur un demi-mot a une valeur comprise entre 0 et 65535, 0x0000 à 0xFFFF en notation hexadécimale. Enfin, un entier codé sur un mot a une valeur comprise entre 0 et 4294967295, 0x00000000 à 0xFFFFFFFF en notation hexadécimale.

Les adresses d'un processeur MIPS sont des entiers non signés sur 32 bits.

2.1.0.0.4 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, 0x80 à 0x7F en notation hexadécimale. Un entier signé sur un demi-mot est compris entre -32768 et 32767, 0x8000 à 0x7FFF en notation hexadécimale. Enfin, un entier signé sur un mot est compris entre -2147483648 et 2147483647, 0x80000000 à 0x7FFFFFFF en notation hexadécimale.

Il faut remarquer que le bit de plus fort poids d'un octet/demi-mot/mot représentant un entier négatif est toujours égal à 1 alors qu'il vaut 0 pour un nombre positif ou nul (c'est le bit de signe). On peut aussi dire que tout se passe comme si on travaillait modulo 8/16/32 bits respectivement.

2.1.0.0.5 Extension d'une valeur entière

L'extension d'une valeur sur un octet (ou un demi-mot) consiste à coder cette valeur sur un mot entier. Un entier non signé est simplement étendu en ajoutant des bits nuls à gauche. Par exemple la valeur 0xF2 est étendue sur un mot par 0x000000F2. La valeur décimale est toujours $+242$.

Cette règle ne peut par contre pas être appliquée aux valeurs signées. Ainsi 0xF2 sur un octet signé représente la valeur décimale -14, alors que 0x000000F2 sur un mot signé représente la valeur décimale $+242$! L'extension des valeurs signées est donc réalisée en complétant à gauche avec le bit de signe (de poids fort). 0xF2 sur un mot est donc 0xFFFFFFF2.

En anglais, on parle d'une valeur *extended* (complétion avec des zéros) ou bien *sign-extended* (complétion avec le bit de poids fort).

2.2 Généralités

Un MIPS, pour *Microprocessor without Interlocked Pipeline Stages*, est un processeur `RISC` 32 bits. `RISC` signifie qu'il possède un jeu d'instructions réduit (*Reduced Instruction Set Computer*). En contrepartie, il est capable de terminer l'exécution d'une instruction à chaque cycle d'horloge. Pour obtenir ce type de performances, il est nécessaire d'avoir des instructions de taille constante et de construire un *pipeline*. Cette structure permet d'exécuter chaque instruction en plusieurs cycles, mais de terminer l'exécution d'une instruction à chaque cycle. En plus d'un banc de registres, le MIPS possède des unités séparées pour l'extraction des instructions, le décodage des instructions, l'exécution et les accès mémoire (figure 2.1).

Les processeurs MIPS sont notamment utilisés dans des stations de travail (Silicon Graphics, DEC...) de nombreux systèmes embarqués (Palm, modems, imprimantes...), et également des consoles de jeux (Nintendo 64, Sony PlayStation 2, etc.).

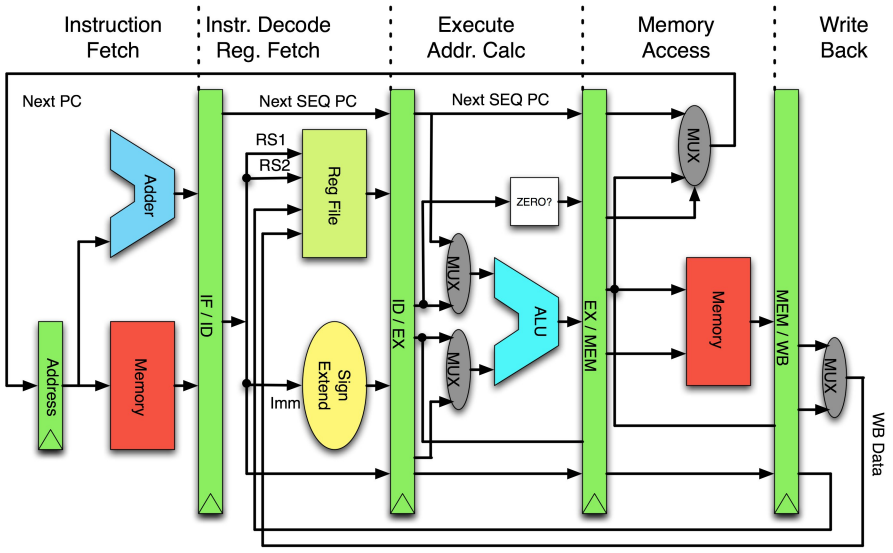


FIGURE 2.1 – Architecture interne du microprocesseur MIPS

2.3 La mémoire

La mémoire d'un ordinateur contient l'ensemble des instructions et données des programmes, toujours codées en binaire. Elle est généralement découpée en différents secteurs, ce qui permet d'accélérer les accès aux données et de leurs affecter des droits de lecture/écriture/exécution spécifiques. Si un système d'exploitation est installé, c'est lui qui a pour rôle (en utilisant le plus souvent une unité matérielle spécialisée `MMU`, pour *Memory Management Unit*) de gérer cette mémoire.

Avant l'exécution, les programmes binaires générés par un assembleur sont d'abord recopiés en mémoire (phase de chargement). Pendant l'exécution le microprocesseur vient y chercher les instructions et données en se repérant notamment grâce au compteur programme `PC` qui contiendra successivement toutes les adresses des instructions à exécuter.

La mémoire peut être vue comme un tableau d'octets (il n'est pas possible d'adresser une donnée plus petite, directement un bit par exemple). Une adresse mémoire correspond alors au rang de l'octet désigné dans ce tableau. Pour stocker en mémoire des valeurs sur plusieurs octets, par exemple un mot de 32 bits, deux solutions existent (figure 2.2) :

- les systèmes de type *gros-boutiste* ou *grand indien* (*big endian* en Anglais) écrivent toujours **l'octet de poids le plus fort à l'adresse la plus basse**.
- à l'inverse, les systèmes de type *petit-boutiste* ou *petit indien* (*little endian*) écrivent l'octet de poids le plus faible à l'adresse la plus basse.

Le MIPS est un processeur big-endian, tout comme les COLDFIRE, DEC ou SUN. Les processeurs ATHLON ou PENTIUM notamment sont *little endian*.

Les adresses du MIPS sont codées sur 32 bits. L'espace d'adressage potentiel est donc de 4Go, soit des adresses comprises entre 0 et $2^{32}-1$. Les mémoires physiques effectivement présentes dans les

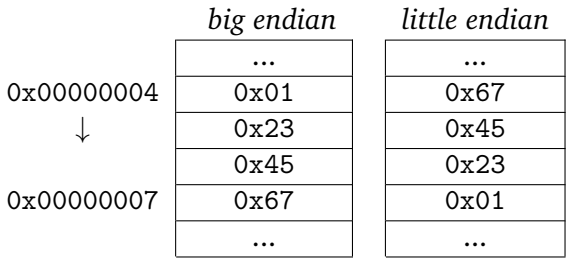


FIGURE 2.2 – Mode d'écriture en mémoire, à l'adresse `0x4`, de la valeur hexadécimale `0x01234567` pour un système *big endian* ou *little endian*.

microcontrôleurs sont généralement bien plus petites, en particulier sur des systèmes embarqués !

La mémoire du MIPS est dite *alignée*. Ceci signifie en particulier que les accès mémoire à des instructions ou données ne peuvent se faire qu'à des adresses "compatibles" avec le type d'information accédée. Par exemple : les instructions étant codées sur 4 octets, le processeur ne pourra lire une instruction qu'à une adresse multiple de 4. Pour les données, les mots (respectivement les demi-mots) devront toujours être placés à des adresses multiples de 4 octets (respectivement 2). Un octet seul pourra par contre être lu n'importe où en mémoire.

Le respect des contraintes d'alignement n'incombe pas à l'assembleur [1] mais au programmeur ! Utilisée à bon escient, la directive `.align` décrite section 3.5.3 permet d'assurer l'alignement correct des données et instructions.

2.4 Les registres

Les registres sont des emplacements mémoire spécialisés utilisés par le processeur pour stocker les opérandes et résultats des opérations. Étant directement câblés au processeur, ils se caractérisent par un temps d'accès très rapide.

2.4.1 Les registres d'usage général

Le MIPS dispose de 32 registres d'usage général (*General Purpose Registers*) de 32 bits chacun, dénotés `$0` à `$31`. Ils peuvent également être identifiés par un mnémonique indiquant leur usage conventionnel (arguments d'une fonction, valeurs temporaires, etc.) ou imposé par l'architecture. Par exemple, le registre `$29` est noté `$sp` (pour *Stack Pointer*), car il est couramment utilisé comme le pointeur de sommet de pile.

1. La seule contrainte d'alignement imposée par l'assembleur et l'éditeur de liens concerne l'alignement des adresses de section. Voir l'annexe C.2.3.

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

Ces conventions devraient être utilisées dans tout programme assembleur[2]. Dans le cadre de ce projet, les restrictions imposées porteront sur $zero (toujours nul, ne peut pas être modifié), $at (utilisé par l'assembleur, notamment pour les pseudo-instructions), $sp et $ra (pour le retour après exécution d'une sous-routine).

2.4.2 Les registres spécialisés

En plus des registres généraux, trois registres spécialisés existent :

- Le compteur programme 32 bits PC (*Program Counter*), qui contient l'adresse mémoire de la prochaine instruction à exécuter. Il est automatiquement incrémenté d'un mot (donc de 32 bits) après la lecture en mémoire de chaque instruction. Les sauts et branchements seront parfois amenés à le modifier directement. Attention, PC a déjà été incrémenté au moment où une instruction est réellement exécutée !
- 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.

D'autres registres existent encore, mais qui ne seront pas utilisés dans ce projet (registres de valeurs flottantes...).

2.5 Les modes d'adressage

Les instructions utilisent toujours zéro, un, deux ou trois opérandes. Le mode d'adressage est la méthode utilisée par le processeur pour localiser un opérande, soit dans un registre soit à une adresse donnée de la mémoire. Tous les modes ne sont pas utilisables pour tous les opérandes d'une opération. Le MIPS compte au total cinq modes d'adressage.

2. Sauf à aimer les comportements non attendus lors de l'exécution...

2.5.1 Adressage registre direct

L'opérande est une valeur sur 32 bits contenue dans un registre.

2.5.1.0.1 Exemple :

```
ADD $2, $3, $4   # Les valeurs des opérandes sont dans les registres 3 et 4
                 # Le résultat (somme) est placé dans le registre 2
```

2.5.2 Adressage immédiat

La valeur numérique de l'opérande est directement encodée dans l'instruction. Les valeurs attendues sont généralement de 16 bits (voir les spécifications des instructions), signées ou non.

2.5.2.0.1 Exemple :

```
ADDI $2, $3, 200         # valeur immédiate décimale
ADDI $2, $3, 0xC8        # la même en hexadécimal
ADDI $2, $3, -200        # valeur immédiate décimale négative
ADDI $2, $3, 0xFFFFFF38  # en hexa négatif, toujours écrire sur 32 bits!
```

2.5.3 Adressage indirect avec base et déplacement : `offset(base)`

L'adresse *en mémoire* de l'opérande est la somme de la valeur contenue dans le registre `base` (interprétée comme une adresse 32 bits) et du déplacement `offset`, entier signé sur 16 bits.

2.5.3.0.1 Exemple :

```
LW $2, 200($3)     # La valeur à l'adresse GPR[3]+200 est placée dans GPR[2]
```

2.5.4 Adressage relatif

Ce mode d'adressage est utilisé par les instructions de branchement. L'opérande est un `offset` signé sur 16 bits.

L'adresse de branchement est déterminée en décalant `offset` de 2 bits à gauche puis en ajoutant cette valeur 18 bits à l'adresse courante du compteur programme. Attention, `PC` a *déjà* été incrémenté au moment du calcul, et contient l'adresse mémoire du mot *suivant* l'instruction de branchement !

2.5.4.0.1 Exemple :

```
B 0xF2      # si cette instruction est à l'adresse 0x20001234 en mémoire,
            # PC vaut 0x20001238 au moment du calcul, et l'adresse de
            # branchement sera: 0xF2<<2 + 0x20001238 = 0x20001600
```

2.5.5 Adressage absolu aligné dans une région de 256Mo

Une adresse 32 bits est calculée à partir de la valeur du compteur programme PC et de l'opérande, un décalage non signé sur 26 bits. Ce mode est utilisé par les instructions de saut (J, JAL...) pour calculer l'adresse de la prochaine instruction à exécuter.

L'adresse de saut est calculée en décalant l'opérande de 2 bits vers la gauche (car les instructions sont toujours alignées sur 4 octets). Les 4 bits de poids forts manquants sont ceux du compteur programme. Comme pour les branchements, PC contient déjà l'adresse de l'instruction de saut + 4 au moment du calcul.

2.5.5.0.1 Exemple :

```
J 0xABC3    # Si cette instruction est à l'adresse 0x20001234 en mémoire,
            # PC vaut 0x20001238 au moment du calcul, et l'adresse de
            # saut sera: 0xABC3<<2 + 0x20000000 = 0x2002AF0C
```

2.6 Les entrées/sorties

Le mécanisme d'entrée/sortie de données, par exemple dans la console ou sur un périphérique externe, consiste à interrompre l'exécution du programme pour faire exécuter un service pas le système. Le principe général est :

1. placer un code de service dans le registre $v0
2. appeler l'instruction syscall : l'exécution du programme est interrompu et le système réalise la tâche dont le code est présent dans $v0. Une fois le service effectué, l'exécution reprend.

Les paramètres des services sont lus dans les registres $a0 à $a3 (par exemple les valeurs à afficher), et les résultats des services sont placés dans $v0 et $v1 à la fin de l'exécution (par exemples les valeurs lues). Les services à gérer dans votre simulateurs sont :

Service	Code	Arguments	Sorties
Affiche un entier sur la sortie standard	1	$a0 : l'entier à afficher	-
Affiche une chaîne de caractères sur la sortie standard	4	$a0 : adresse de la chaîne	-
Lit un entier sur l'entrée standard	5	-	$v0 : l'entier lu
Lit une chaîne de caractères sur l'entrée standard	8	$a0 : adresse du buffer où écrire la chaîne $a1 : taille du buffer	-
Exit (termine l'exécution)	10	-	-

L'exemple suivant affiche une chaîne de caractères sur la sortie standard :

```
        .data
chaine: .asciiz "Hello World!\n"  # la chaine à afficher
```

```
.text
LI $v0, 4                # met le code de service dans le registre $v0
LI $a0, chaine           # met l'adresse de la chaine à afficher dans $a0
SYSCALL                  # appel système, qui effectuera l'affichage
```

2.7 Gestion des exceptions

Plusieurs exceptions peuvent arriver lors de l'exécution d'un programme : débordements, mémoire non allouée, etc. Par exemple la somme de deux valeur de 32 bits ne tient pas forcément sur 32 bits. La valeur contenue dans le registre de destination n'est pas alors pas correcte :

```
LI $8, 0xA1234567
LI $9, 0xA1234567
ADD $10, $8, $9     # 0xA1234567 + 0xA1234567 = 0x142468ACE
                    # mais le registre ne peut contenir que 0x42468ACE (32 bits)
                    # il y a eu débordement, et la valeur dans $10 serait fausse!
```

Certains processeurs possèdent un *registre d'état*, dont la valeur est mise à jour après l'exécution de chaque instruction, par exemple pour indiquer un fonctionnement normal ou un débordement. Le MIPS ne possédant pas de registre d'état, les exceptions devront être affichées dans la console de votre simulateur. Selon le type d'exception, l'exécution du programme pourra continuer ou non.

2.8 Exécution et delay slot

Sans rentrer dans les détails de l'architecture de type *pipeline* du processeur MIPS, il est important de mentionner un point concernant l'ordre d'exécution des instructions.

Les instructions de branchement ou de saut (`BEQ`, `J`...) nécessitent des cycles supplémentaires avant de sortir du *pipeline* d'exécution. De ce fait, l'instruction qui suit immédiatement le branchement (dite dans son *delay slot*) aura déjà été entrée dans le *pipeline* et partiellement exécutée, avant même que le branchement soit lui-même exécuté. En particulier :

- l'instruction dans le *delay slot* sera au final **toujours** exécutée, quelle que soit l'évaluation de la condition de branchement ;
- le compteur programme `PC` aura déjà été incrémenté lorsque l'adresse de branchement est calculée (`PC` contient donc alors l'adresse du *delay slot*, i.e., celle du branchement + 4) ;
- le comportement du programme sera (très) incertain si l'instruction de branchement et celle dans le *delay slot* utilisent les mêmes registres, si plusieurs branchements se suivent, si une interruption est déclenchée, etc.

Illustrons ceci sur un petit exemple :

```
debut:
ADDI $10, $0, 0xAB   # Met la valeur 0xAB dans $10
ADD  $11, $0, $10    # Copie cette valeur dans $11
```

```
ADDI $8, $0, 17      # Met la valeur 17 dans $8
BEQ $10, $11, ici    # Branchement si le contenu des registres est identique
ADDI $8, $8, 1       # *DELAY SLOT* du BEQ: La valeur dans $8 est incrémentée,
                     # que le branchement précédent ait lieu ou non!!
ici:
ADDI $8, $8, 1       # Incrémente la valeur dans $8
```

Dans ce programme, la valeur du registre `$8` aura été incrémentée avant que le branchement ait lieu ! Donc la valeur finale dans `$8`, après exécution de l'addition en `ici`, sera 19 et non 18...

Une première solution pour éviter ce problème à l'exécution est de toujours placer une instruction `NOP` (qui ne fait rien) après un saut ou un branchement. Il s'agit d'une « bulle » dans le pipeline. Un cycle est perdu dans le temps d'exécution, mais le fonctionnement est correct :

```
BEQ $10, $11, ici    # Branchement si le contenu des registres est identique
NOP                  # un innofensif NOP dans le delay slot...
ADDI $8, $8, 1       # Désormais, ce ADDI n'est pas exécuté si branchement
```

Une optimisation, réalisée par l'assembleur, est d'*inverser l'ordre* de l'instruction et de *celle qui la précède* (lorsque les registres concernés sont indépendants) :

```
BEQ $10, $11, ici    # Branchement si le contenu des registres est identique
ADDI $8, $0, 17      # Ce ADDI sera en fait exécuté AVANT le branchement!
ADDI $8, $8, 1       # Ici ADDI n'est pas exécuté si branchement
```

Dans le cadre de ce projet, il n'est pas demandé de modéliser le *pipeline*, les instructions doivent être exécutées dans l'ordre où elles se présentent. Si l'assembleur a optimisé le code comme ci-dessus, votre programme simulé n'aura donc pas le bon comportement !

Pour éviter ceci, toujours ajouter la directive `.set noreorder` dans vos fichiers test ! L'assembleur ajoutera des `NOP` si nécessaire, mais n'inversera pas d'instructions.

Chapitre 3

Le langage d'assemblage du MIPS

La syntaxe présentée ici est issue de l'assembleur GNU `as` [8] (notamment utilisé par le compilateur `gcc`), avec certaines options dédiées aux processeurs MIPS. Plusieurs restrictions de syntaxe ont été apportées afin de simplifier et d'éviter de commettre des erreurs difficiles à détecter.

En ignorant les portions de texte en commentaire, un programme se présente comme une liste de lignes séparées par des caractères de fin de ligne. Chaque ligne peut contenir une définition d'étiquette, et éventuellement soit une instruction avec ses paramètres, soit une directive avec son paramètre. Une ligne peut aussi être vide (recommandé pour aérer le texte). Les différentes unités lexicales peuvent être séparées par des combinaisons d'espaces et/ou de tabulations.

3.1 Les commentaires

Il y a deux types de commentaires[1] : ils commencent soit par le caractère « `#` » et se terminent à la fin de la ligne, ou alors commencent par « `/*` » jusqu'au au premier « `*/` ». Par exemple :

```
# ceci est une ligne entièrement commentée
B 0xF2  # ici le commentaire commence après l'instruction
/* Encore un commentaire, mais sur
   plusieurs lignes cette fois */
```

3.2 Les étiquettes

Une étiquette permet de *nommer une adresse mémoire*, et peut servir d'opérande à une instruction ou à une directive.

— Syntaxiquement, une étiquette est une suite de caractères alphanumériques[2] ou « _ », et ne commençant pas par un chiffre.

1. On ne le répétera jamais assez, abusez des commentaires ! Aussi bien dans vos programmes en assembleur que dans le source C de ce projet d'ailleurs...
2. Un caractère alphanumérique est une lettre majuscule ou minuscule, ou un chiffre.

— Une étiquette est toujours définie en début de ligne, puis suivie du caractère « : » (qui ne fait pas partie de son nom).
— Plusieurs étiquettes peuvent être associées à la même opération ou directive.
— Une étiquette ne peut être définie qu'une seule fois dans un programme.
— L'adresse désignée par une étiquette est celle du prochain octet codé lors de l'assemblage. Sa valeur lors de l'exécution est égale à son adresse d'implantation dans la mémoire après le chargement du programme. Elle dépend donc de la section dans laquelle elle est définie et de sa position dans cette section (voir les sections 3.5.1 et 5).

3.3 Les nombres littéraux

Un nombre littéral est une suite de chiffres hexadécimaux précédée de « `0x` », ou une suite de chiffres décimaux.

```
0xFdCba987  # nombre hexadécimal
123456789   # nombre décimal
```

En général, les littéraux hexadécimaux sont interprétés comme des nombres non-signés, et les littéraux décimaux comme des nombres signés en complément à 2 (voir chapitre 2.1). Cette interprétation est en particulier utilisée par l'assembleur pour vérifier le non-débordement des littéraux dans leur contexte d'utilisation. Par exemple, les littéraux signés sur 32 bits doivent avoir des valeurs comprises entre -2147483648 et 2147483647, alors que les littéraux non signés doivent avoir des valeurs entre 0 et 4294967295. La seule exception à cette règle porte sur les littéraux utilisés dans le mode d'adressage avec base et déplacement où le déplacement est toujours considéré comme un signé sur 32 bits, même si le littéral du déplacement est hexadécimal. [3]

Il est possible d'appliquer l'opérateur unaire « - » pour désigner l'opposé d'un littéral signé ou non-signé. Dans le cas non-signé, le nombre alors désigné est toujours considéré comme non-signé et correspond au calcul de l'opposé sur 32 bits.

3.4 Les instructions machine

Une instruction est de la forme champ `opération`, suivi éventuellement par un, deux ou trois champs opérandes séparés par le caractère « , ».

```
opération [op [,op [,op]]]
```

Le champ opération désigne un des mnémoniques d'instruction du MIPS. Pour simplifier, ils sont toujours écrits en majuscules (`ADD`, et pas `add` ni `AdD`).

3. Cette interprétation n'est pas celle de l'assembleur GNU qui interprète les littéraux suivant leur représentation en complément à 2 sur 32 bits. L'assembleur GNU ne vérifie donc pas vraiment les débordements de littéraux.

Les opérandes suivent la syntaxe des modes d'adressage présentés section 2.5. Les registres seront toujours écrits sous la forme `$1` ou via leur mnémonique `$at`. Les opérandes source sont le plus souvent des registres ou des valeurs immédiates quand la destination est un registre ou une donnée en mémoire. Lorsqu'il y a plusieurs opérandes, l'opérande destination précède généralement le ou les opérandes source.

3.5 Les directives

Une directive commence toujours par un point (« . »). Deux types de directives seront principalement utilisés : les directives de sectionnement du programme et les directives de définition de données.

3.5.1 Directives de sectionnement `.text`, `.data` et `.bss`

La mémoire est généralement divisée en blocs physiques de 4 kilo-octets qui peuvent être protégés en lecture, écriture ou exécution, et éventuellement partagés par plusieurs processus, par des dispositifs matériels du microprocesseur. L'intérêt est de pouvoir placer différemment en mémoire le code et les données d'un programme, en fonction du niveau de protection désiré. Les instructions d'un programme seront par exemple placées dans des blocs protégés en écriture, de même que certaines données dont on veut interdire la modification en cours d'exécution. Les autres données seront chargées dans une zone en lecture et écriture permise et exécution interdite.

Pour pouvoir profiter de cette possibilité offerte par le matériel, l'assembleur permet de définir plusieurs sections :

- la *section d'instruction* : toujours chargée dans une zone de la mémoire en exécution et lecture permise, mais en écriture interdite ;
- la *section de données initialisées* : créée et chargée dans une zone de la mémoire en lecture et écriture permise, mais en exécution interdite.
- la *section de données non initialisées* : par rapport à la section précédente, elle n'est définie que par sa taille. Son contenu n'est pas explicitement initialisé.

Ces directives de sectionnement ont pour syntaxe `.text`, `.data` et `.bss`. Elles indiquent à l'assembleur d'assembler les lignes suivantes du programme dans la section correspondante.

En général, les instructions ne sont définies que dans la section `.text`, mais on peut en fait tout à fait les définir dans la section `.data` (dans ce cas, on ne pourra probablement pas les exécuter...). Les données initialisées peuvent être définies dans les deux sections `.text` ou `.data`.

Si aucune directive de sectionnement n'est présente dans un programme, tout est assemblé dans la section `.text`.

3.5.2 Les directives de définition de données

On distingue les définitions de données initialisées, qui ont lieu dans les zones `.text` ou `.data`, des déclarations de données non initialisées qui ont généralement lieu dans la zone `.bss`.

3.5.2.1 Déclaration de données initialisées

3.5.2.1.1 `.byte valeur`

Cette directive réserve un octet en mémoire et lui affecte une valeur initiale donnée par un entier sur 8 bits, en décimal ou hexadécimal. Par exemple, les lignes suivantes permettent de réserver deux octets, aux adresses `tab` et `tab+1` :

```
tab:  .byte 2
      .byte 0xFF
```

L'interprétation de ces valeurs comme signées ou non dépend du contexte d'utilisation, et non de leur définition. Ainsi la valeur `0xFF` en `tab+1` peut être vue comme 255 ou -1, selon l'instruction qui y accède.

3.5.2.1.2 `.word valeur`

Cette directive réserve un mot en mémoire et lui affecte une valeur initiale donnée par un entier sur 32 bits, en décimal ou hexadécimal.

3.5.2.1.3 Directive avec étiquette

La directive précédente peut être utilisée avec pour paramètre `valeur` une étiquette. Dans ce cas, la valeur assemblée est en fait une référence à l'étiquette.

```
mot1: .word mot3      # mot1 désigne un mot contenant l'adresse de mot3
mot2: .word 0xABC
mot3: .word -14
```

A noter que dans ces cas l'assembleur doit toujours générer des données de *relocation*, les adresses des étiquettes n'étant connues que par rapport au début de leur section et non de manière absolue. Tout ceci sera détaillé dans la section 5.

3.5.2.1.4 Directives de déclaration de chaînes de caractères

Il est aussi possible de définir des chaînes de caractères, à l'aide des directives `.string` et `.ascii`. À vous de trouver comment les utiliser !

3.5.2.2 Déclaration de données avec `.skip` *taille*

La directive `.skip` permet de réserver un nombre d'octets égal à *taille* à l'adresse *étiquette*.

```
toto: .skip 13
```

Dans la section `.bss`, ces octets sont simplement réservés mais pas initialisés. Dans les deux autres sections, la zone mémoire correspondant à une directive `.skip` est en fait initialisée avec des zéros.

3.5.3 Directive d'alignement `.align`

Les contraintes d'alignement ont été présentées section 2.3. Leur non respect peut entraîner des erreurs à l'exécution (plantage ou comportement aberrant). Exemple :

```
.text
ici:
ADD $5, $6, $7          # une instruction
.byte 0xAB              # un octet réservé (pas trop de sens ici, mais autorisé)
SUB $5, $6, $8          # une autre instruction
mot: .word 0x1234567    # un mot de 32 bits
```

Les codages binaires des instructions `ADD` et `SUB` étant respectivement `00c72820` et `00c82822`, le contenu de la mémoire à partir de l'adresse `ici` sera : `[00c72820AB00c8282212345678]`.

En supposant que l'adresse `ici` est alignée (multiple de 4), la première instruction `ADD` pourra être lue correctement.

Par contre, l'instruction `SUB` ne pourra jamais l'être, puisque le compteur programme `PC` ne peut contenir que des valeurs multiples de 4. On pourra lire le code `AB00c828` en `ici`+4 ou encore `22123456` en `ici`+8, mais on ne retrouvera jamais l'instruction `SUB` ! De même le mot `0x12345678` ne pourra jamais être lue par une instruction de lecture de mots.

La directive `.align` *puiss* indique à l'assembleur d'incrémenter l'adresse du prochain élément à coder sur la prochaine adresse multiple de 2^{puiss}. Si des octets doivent être sautés, ils sont codés à 0.

```
.text
ici:
ADD $5, $6, $7          # une instruction
.byte 0xAB              # un octet réservé (pas trop de sens ici, mais autorisé)
.align 2                #   ==> Pour aligner le SUB!
SUB $5, $6, $8          # une autre instruction
mot: .word 0x1234567    # un mot de 32 bits
```

En binaire, l'exemple ci-dessus est codé : `[00c72820AB00000000c8282212345678]` Les instructions et le mot sont correctement alignés, et pourront être bien récupérés à l'exécution.

3.5.4 Directive d'exportation des noms

Par défaut, les étiquettes définies dans un fichier source sont considérées comme locales et ne seront pas visibles par d'autres unités de compilation lors de l'édition de liens. Une étiquette utilisée dans un fichier source mais non définie dans ce fichier sera supposée *globale* (exportée par une autre source).

La directive `.global` *`etiq`* indique que l'étiquette *etiq* doit être exportée. Par exemple, le point d'entrée d'un programme s'appelle généralement `_start` et doit être exportée.

```
.global _start
  # ...
_start: # ...
```

3.5.5 Directive de non ré-ordonnancement

Pour éviter que l'assembleur optimise le code binaire en réordonnant des instructions (voir la section 2.8 sur le *delay slot*), il est nécessaire de toujours ajouter la directive `.set noreorder` dans vos fichiers test en langage assembleur.

Chapitre 4

Les instructions du MIPS

Ce chapitre présente le sous-ensemble des instructions et pseudo-instructions du MIPS retenu pour le projet. Les spécifications détaillées des instructions, issues de la documentation de référence de *MIPS Technologies* [1], sont fournies dans un document annexe. Elles indiquent notamment le processus de codage permettant de traduire les instructions du programme source en binaire, et le comportement des instructions à l'exécution.

4.1 Catégories d'instructions

L'architecture MIPS est de type *load/store architecture* : la mémoire n'est accédée que par les instructions de lecture/écriture. Toutes les opérations autres sont effectuées à partir de valeurs contenues dans des registres, ou immédiates (encodées dans l'instruction). Contrairement à bien des processeurs, chaque opérande d'une instruction n'admet ainsi qu'un seul mode d'adressage, et il n'y en a du reste que cinq possibles (décrits section 2.5).

Les processeurs MIPS possèdent un jeu d'instruction réduit (processeur *RISC*) et au codage très régulier. Le codage binaire d'une instruction est *toujours* sur 32 bits [1], alignés en mémoire. Ceci permet d'augmenter la vitesse de transfert des données et facilite l'extraction et le décodage des instructions. Chaque étape est généralement réalisée en un seul cycle d'horloge dans le pipeline.

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

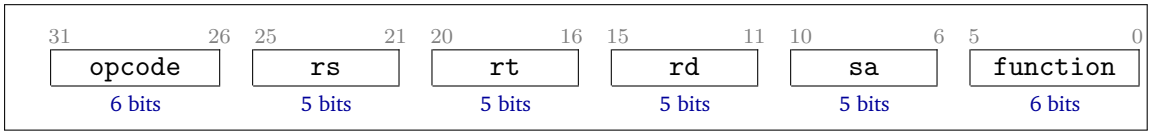
```
R-instruction rd, rs, rt
I-instruction rt, rs, immediate
J-instruction instr_index
```

Certaines instructions ont moins d'opérandes (par exemple `B offset`, qui est en fait de type `I`) auquel cas le codage des opérandes « manquants » est une valeur constante, généralement zéro.

1. Ou 64 bits pour des versions récentes.

4.1.1 Les instructions de type R

Le codage binaire des instructions R-type, pour *Register-type*, suit le format suivant :

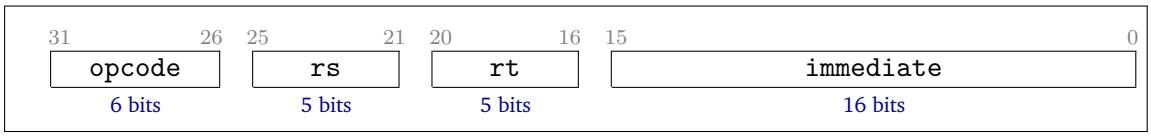


Les champs sont les suivants :
- opcode : (*operation code*) identifie la famille d'instructions (ce champ est commun à tous les types d'instruction) ;
- rd : identifie le registre destination (valeur sur 5 bits, donc entre 0 et 31, codant le numéro du registre) ;
- rs : identifie le registre contenant le premier opérande source ;
- rt : identifie le registre contenant le second opérande source (appelé aussi target) ;
- sa : (*shift amount*) est le nombre de bits de décalage, pour les instructions SLL, SRL...
- function : 6 bits additionnels pour différencier les instructions R-type.

Pour les instructions de type R, c'est l'ensemble des champs opcode et function qui spécifie l'instruction. Ceci permet de coder plus que 64 instructions (6 bits de opcode), ce qui serait peu même pour un processeur RISC.

4.1.2 Les instructions de type I

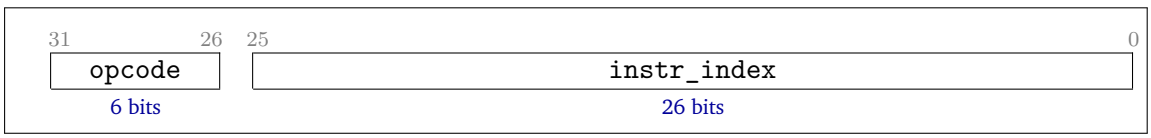
Le codage binaire des instructions I-type, pour *Immediate-type*, suit le format suivant :



avec opcode le code opération, rt le registre cible, rs le registre source et immediate une valeur immédiate signée codée sur 16 bits.

4.1.3 Les instructions de type J

Le codage binaire des instructions J-type, pour *Jump-type*, suit le format suivant :



où opcode est le code opération et instr_index une valeur de saut non signée sur 26 bits.

4.2 Instructions étudiées dans le projet

4.2.1 Instructions arithmétiques

Mnémonique	Opérandes	Opération
ADD	rd, rs, rt	GPR[rd] ← GPR[rs] + GPR[rt]
ADDU	rd, rs, rt	GPR[rd] ← GPR[rs] + GPR[rt]
ADDI	rt, rs, immediate	GPR[rt] ← GPR[rs] + immediate
ADDIU	rt, rs, immediate	GPR[rt] ← GPR[rs] + immediate
SUB	rd, rs, rt	GPR[rd] ← GPR[rs] - GPR[rt]
MULT	rs, rt	(HI, LO) ← GPR[rs] * GPR[rt]
DIV	rs, rt	LO ← GPR[rs] div GPR[rt] HI ← GPR[rs] mod GPR[rt]

La plupart des instructions arithmétiques ont plusieurs « versions » : signée, non signée (U), immédiate (I)... Seules celles de l'addition sont présentées ici : ADD, ADDU, ADDI et ADDIU. Il existe d'autres formats pour les autres opérations. Le suffixe « U » est un faux amis : ADDU est exactement équivalent à ADD sauf qu'il n'y a pas d'exception levée en cas de débordement (voir la documentation).

Pour MULT, les valeurs des deux registres rs et rt sont multipliées, ce qui donne un résultat sur 64 bits. Les 32 bits de poids fort de ce 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 section 4.2.4.

Pour DIV, le quotient de rs divisé par rt est placée dans le registre LO, et le reste de la division entière dans le registre HI.

4.2.2 Les instructions logiques

Mnémonique	Opérandes	Opération
AND	rd, rs, rt	GPR[rd] ← GPR[rs] AND GPR[rt]
OR	rd, rs, rt	GPR[rd] ← GPR[rs] OR GPR[rt]
ORI	rd, rs, immediate	GPR[rd] ← GPR[rs] OR immediate
XOR	rd, rs, rt	GPR[rd] ← GPR[rs] XOR GPR[rt]

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

Il existe aussi des opérations immédiates (ORI ici) : immediate (16 bits) est étendue à gauche avec des 0, puis combinée avec le contenu de rs.

4.2.3 Les instructions de décalage et set

Mnémonique	Opérandes	Opération
SLL	rd, rt, sa	GPR[rd] ← GPR[rt] « sa
SRL	rd, rt, sa	GPR[rd] ← GPR[rt] » sa
LUI	rt, immediate	GPR[rt] ← immediate « 16
SLT	rd, rs, rt	GPR[rd] = (GPR[rs] < GPR[rt])? 1 : 0

Le contenu du registre 32 bits `rt` est décalé de `sa` bits, à gauche pour `SLL` et à droite pour `SRL` (en insérant des zéros sur les bits de poids fort). `sa` est une valeur immédiate sur 5 bits, donc entre 0 et 31. Le résultat est placé dans le registre `rd`.

`LUI` (*Load Upper Immediate*) charge une valeur `immediate` sur 16 bits dans les 2 octets de poids fort du registre `rt`. Les deux octets de poids faible sont mis à zéro.

`SLT` (*Set on Less Than*) Met la valeur 1 dans `rd` si le contenu de `rs` est plus petit que celui de `rt`, 0 sinon. Les valeurs dans `rs` et `rt` sont interprétées comme des entiers 32 bits signés.

4.2.4 Les instructions de lecture/écriture mémoire

Mnémonique	Opérandes	Opération
LW	rt, offset(base)	GPR[rt] ← memory[GPR[base]+offset]
SW	rt, offset(base)	memory[GPR[base]+offset] ← GPR[rt]
LB	rt, offset(base)	GPR[rt] ← memory[GPR[base]+offset]
LBU	rt, offset(base)	GPR[rt] ← memory[GPR[base]+offset]
SB	rt, offset(base)	memory[GPR[base]+offset] ← GPR[rt]
MFHI	rd	GPR[rd] ← HI
MFLO	rd	GPR[rd] ← LO

`LW` (*Load Word*) place le contenu du mot de 32 bits à l'adresse mémoire `GPR[base] + offset` dans le registre `rt`. `offset` est une valeur immédiate signée sur 16 bits.

`SW` (*Store Word*) place le contenu du registre `rt` dans le mot de 32 bits à l'adresse mémoire `GPR[base] + offset`.

`LB` (*Load Byte*) charge un octet en mémoire, l'étend à gauche (`sign-extention`), et place le résultat dans `rt`. Idem pour `LBU`, mais la valeur est `extended` (complétion avec des zéros).

`SB` (*Store Byte*) place l'octet de poids faible du registre `rt` à l'adresse mémoire `GPR[base] + offset`.

`MFHI` (*Move from HI*) : copie le contenu du registre `HI` dans le registre `rd`.

`MFLO` (*Move from LO*) : copie le contenu du registre `LO` dans `rd`.

4.2.5 Les instructions de branchement et de saut, et de contrôle

Mnémonique	Opérandes	Opération
B	offset	branchement: PC ← PC + offset≪2
BEQ	rs, rt, offset	Si (GPR[rs] = GPR[rt]) alors branchement
BNE	rs, rt, offset	Si (GPR[rs] != GPR[rt]) alors branchement
BGTZ	rs, offset	Si (GPR[rs] > 0) alors branchement
BLEZ	rs, offset	Si (GPR[rs] <= 0) alors branchement
J	instr_index	PC ← PC[31:28] | instr_index≪2
JAL	instr_index	GPR[31] ← PC+8, PC ← PC[31:28] | instr_index≪2
JR	rs	PC ← GPR[rs].
SYSCALL		Interrompt l'exécution pour un service

`B` effectue un branchement après l'instruction. Le décalage signé de *18 bits* (`offset` de 16 bits décalés de 2) est ajouté à l'adresse de l'instruction de branchement + 4 pour déterminer l'adresse effective du saut (`PC` a déjà été incrémenté au moment du calcul du décalage).

Les autres instructions de branchement `B[xx]` ne branchent que si la condition sur les registres `rs` et `rt` est vérifiée.

`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 `instr_index` décalé de 2. Les 4 bits de poids forts restant correspondent aux 4 bits de poids fort du compteur `PC` (voir section 2.5).

Attention, l'instruction suivant immédiatement le `J`, dans son *delay slot*, sera exécutée *avant* le saut lui-même (voir la section 2.8) ! On y place généralement un `NOP`.

`JAL` fonctionne exactement comme `J`, mais place en plus l'adresse de retour dans le registre `$31` (`$ra`, *return address*). Il s'agit de l'adresse de la *deuxième instruction* suivant le `JAL` et où l'exécution pourra reprendre après le traitement de la routine (voir l'exemple annexe B). L'instruction dans le *delay slot* sera exécutée avant le saut lui-même.

`JR` effectue un saut à l'adresse spécifiée dans `rs`. Son usage le plus fréquent est `JR $ra`, en fin d'une routine ayant été appelée par un `JAL`.

4.3 Les pseudo-instructions

Les pseudo-instructions ne sont pas définies parmi les instructions du MIPS, mais uniquement au niveau de l'assembleur. Lors de la compilation, l'assembleur les remplace automatiquement par un équivalent (pas forcément unique) composé d'une ou plusieurs instructions en langage machine.

Certaines pseudo-instructions traduisent simplement des opérations courantes :

Mnémonique	Opérandes	Opération équivalente
`NOP`		`SLL $zero, $zero, $zero`
`MOVE`	`rt, rs`	`ADDU rt, rs, $zero`
`LI`	`rt, immediate15`	`ADDIU rt, $zero, immediate15`
`LI`	`rt, immediate16`	`ORI rt, $zero, immediate16`
`LI`	`rt, immediate32`	`LUI rt, hi(immediate32)`
		`ORI rt, rt, lo(immediate32)`
`LA`	`rt, label`	`LUI rt, addendHI16` `+ relocation`
		`ADDIU rt, rt, addendLO16` `+ relocation`

`NOP` (*Not an Operation*) n'effectue aucun traitement, seul le compteur programme est incrémenté (lors du *fetch*, lecture de l'instruction).

`MOVE` copie le contenu du registre `rs` dans le registre destination `rt`.

`LI` (*Load Immediate*) place la valeur immédiate dans le registre destination `rt`. Selon la valeur de `immediate`, sur 15 bits ($\leq$ `0x7FFF`), sur 16 bits (avec bit 15 == 1) ou sur 32 bits, le code correspondant est différent.

`LA` (*Load Address*) charge l'adresse d'un symbole dans `rt`. Cette pseudo-instruction est obligatoirement accompagnée d'informations de relocation liées au symbole.

Les autres pseudo-instructions (très nombreuses, en réalité) reprennent des instructions existantes, mais avec des paramètres différents (nombre, mode d'adressage, etc.). Nous ne gardons ici que les cas où une étiquette est utilisée en paramètre pour un saut, un branchement ou un

accès mémoire.

Mnémonique	Opérandes	Opération équivalente
B[xx]	[rs, rt,] label	offset = label - (adresse instruction + 4) B[xx] [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 étiquette fonctionnent sur le même principe : le décalage `offset` est calculé par différence entre l'adresse mémoire nommée par l'étiquette et celle de l'instruction suivant le branchement (car à l'exécution du branchement, `PC` aura déjà été incrémenté).

Un saut sur une étiquette est obligatoirement accompagné d'informations 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 `LW` et `SW` suit la syntaxe `offset(base)`, où `offset` est codé sur 16 bits. Mais puisqu'une étiquette désigne une adresse mémoire, donc 32 bits, il est nécessaire d'utiliser deux instructions `LUI` puis `LW/SW` pour accéder correctement à cette adresse. `LUI` charge la valeur de 16 bits `addendHI16` dans les 2 octets de poids fort du registre temporaire `$1`, et `LW/SW` rajoute un offset `addendLO16` à cette adresse de base. Ces deux valeurs `addend` ne sont pas absolues, et sont toujours accompagnées d'informations de relocation.

Le mécanisme de **relocation**, partie « avancée » de ce projet, est entièrement décrit au chapitre 5. Un exemple complet utilisant étiquettes et relocation est fourni en annexe B.

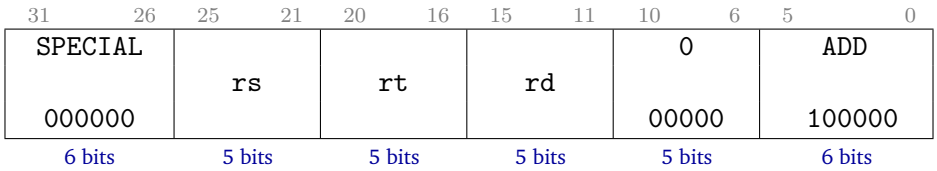
4.4 Codage binaire des instructions

Les spécifications des instructions étudiées dans ce projet sont données dans un document annexe. Elles sont directement issues de la documentation fournie par le *Software User's Manual* de *Architecture For Programmers Volume II* de *MIPS Technologies* [1].

4.4.1 Exemple : l'instruction `ADD`

Nous donnons ici un exemple pour expliciter la spécification d'une instruction, l'opération `ADD`. Les autres instructions ne seront pas détaillées, seules les spécifications du manuel seront données.

Pour chaque instruction, la documentation fournit le format de codage binaire, une description de l'opération effectuée et éventuellement des remarques ou restrictions d'usage.



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] ← GPR[rs] + GPR[rt]`

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

No comment, juste un petit exercice pratique d'anglais... Les descriptions données dans le manuel sont généralement très claires.

Restrictions : `None`

Operation :

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

Exceptions : `Integer Overflow`

Programming Notes :

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

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

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

A vous de retrouver ceci ! Un bon petit exercice pour la compréhension...

Chapitre 5

Relocation

5.1 Cycle de vie d'un programme

Un programme exécutable est obtenu en plusieurs étapes à partir de un ou plusieurs fichiers sources. Chacun de ces fichiers sources décrit l'un des modules du programme dans un langage de programmation donné : langage de haut niveau (Ada, C, etc.) ou langage d'assemblage.

Les étapes successives permettant d'obtenir un programme exécutable sont les suivantes :

1. Analyse et traduction de chaque fichier source : cette étape est réalisée pour chaque fichier par le compilateur ou l'assembleur du langage utilisé. Le résultat est un *fichier binaire relogeable* (on dit aussi *translatable*). Ce fichier contient des instructions qui font référence à des adresses incomplètes ou inconnues :
   — le module peut utiliser des données ou des sous-programmes définis dans d'autres modules que celui en cours de compilation ou d'assemblage ;
   — l'adresse de chargement en mémoire du programme exécutable n'est pas connue (et pour cause, puisque ce dernier n'existe pas encore). Le compilateur ou l'assembleur place donc à l'adresse 0 chaque section du module, qu'il s'agisse des données ou des instructions.
2. Fusion des différents fichiers binaires relogeables : il s'agit ici de réunir les différents modules du programme. Il faut en particulier « résoudre » les adresses inconnues : si un module `A` utilise un sous-programme `P` défini dans un module `B`, il devient dans cette étape possible de compléter `A` avec l'adresse de `P`. Le résultat est un fichier binaire relogeable unique, ne contenant en principe plus aucune adresse inconnue. [1]
   Cette étape est réalisée par l'éditeur de liens.
3. Implantation du programme en mémoire (ou relocation) : il s'agit de la dernière étape permettant d'obtenir un programme exécutable. Toutes les adresses utilisées par le programme sont relogées (ou translatées) de façon à tenir compte des adresses auxquelles il est prévu de charger en mémoire les différentes sections du programme (instructions, données modifiables, données non modifiables, etc.). Ces adresses de chargement ont une valeur par défaut, mais peuvent aussi être spécifiées par le programmeur.

1. Ceci n'est vrai que dans le cas où le système d'exploitation sous-jacent n'utilise pas les bibliothèques partagées ni l'édition de liens dynamique.

L'étape de relocation est en général confiée, elle aussi, à l'éditeur de liens : le résultat est un *fichier binaire exécutable* qui ne contient que des adresses définies et absolues et donc définitives. La relocation est effectuée une fois pour toutes et le chargement ultérieur en mémoire d'un tel fichier est trivial.

Dans certain cas très particuliers, (comme par exemple pour le projet C !) le processus d'implantation est délégué au chargeur du système. Dans ce cas, c'est au moment du chargement en mémoire du programme, à chaque exécution donc, que les adresses relatives sont transformées en adresses absolues.

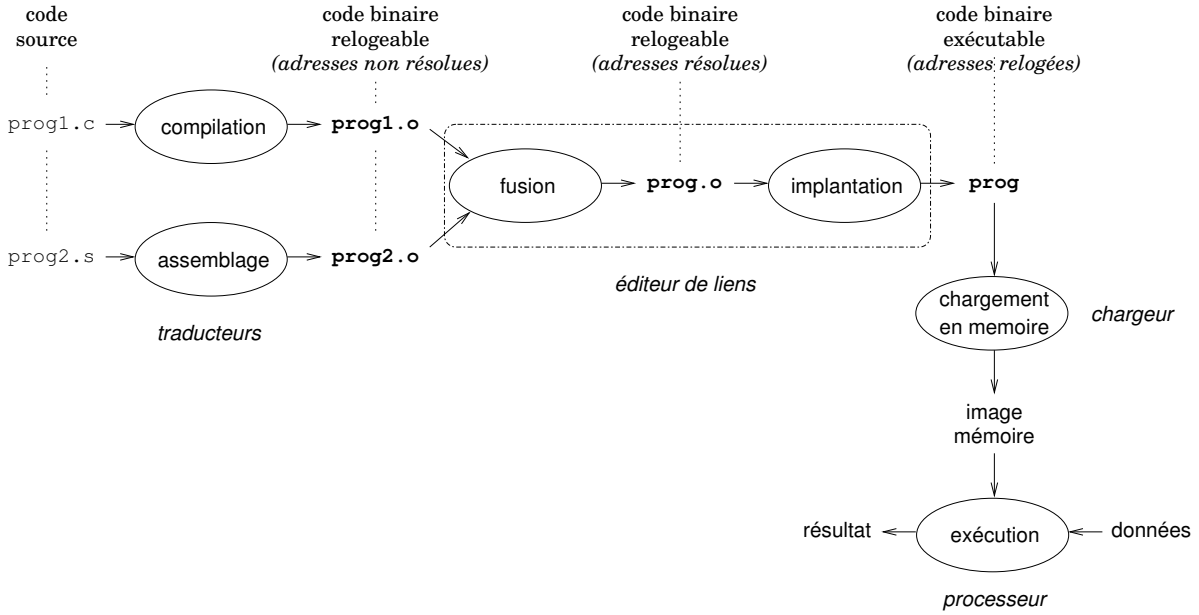


FIGURE 5.1 – Cycle de vie d'un programme

5.2 Les symboles

Un programme en langage d'assemblage (et à plus forte raison un programme en langage de haut niveau) ne manipule jamais ou très rarement des adresses directement. Ces dernières sont représentées sous forme symbolique au moyen de noms. Ce sont les noms de variables ou de sous-programmes dans un programme en langage de haut niveau, et ce sont les étiquettes dans un programme en langage d'assemblage. Chacun de ces noms est associé à une adresse, et on appelle *symbole* cette association.

Chaque symbole est caractérisé par plusieurs propriétés essentielles :

nom : c'est la partie « visible » du symbole, c'est à dire ce qui permet au programmeur de déclarer puis d'utiliser le symbole. Il s'agit bien sûr d'une chaîne de caractères ;

valeur : la valeur d'un symbole est l'adresse associée au nom du symbole. Cette valeur peut être *indéfinie* (inconnue), *relative* au début d'une section (d'une partie) du programme, ou bien *absolue* (relative au début de la mémoire) ;

portée : un symbole peut être *local* ou *global* :
- un *symbole local* est défini dans un fichier source et n'est visible, c'est à dire utilisable, que dans ce fichier source (on pourrait dire aussi que ce symbole est *privé*). La valeur d'un symbole local est toujours définie.
- un *symbole global défini* est un symbole qui est défini dans le fichier source considéré et est *exporté* : ce symbole est visible dans tous les fichiers sources qui constituent le programme (on pourrait dire aussi que ce symbole est *public*).
- un *symbole global indéfini* est un symbole qui est utilisé dans le fichier source considéré mais n'est pas défini dans ce fichier : ce symbole est *importé* par le fichier. La valeur d'un tel symbole est indéfinie.

type : on distingue essentiellement deux types de symboles :
- les *symboles de programme* sont définis et utilisés explicitement par le programmeur. Ce sont principalement les étiquettes (ou les noms de variables et de sous-programmes).
- les *symboles de section* sont associés chacun à une section. La valeur d'un symbole de section est l'adresse en mémoire de la section associée à ce symbole.

section de définition : un symbole défini (local ou global) est défini dans une section, laquelle est donc sa section de définition. Un symbole indéfini ne possède pas bien sûr de section de définition.

5.3 Principe de la relocation

5.3.1 Définition

L'opération dite de **relocation**, ou de relogement ou d'implantation, intervient lors de la fusion de fichiers objet et à la transformation d'un fichier binaire relogeable en un fichier binaire exécutable. En d'autres termes, il s'agit de convertir les *adresses incomplètes* dans les instructions ou les réservations de données en *adresses définitives* (définies et absolues) en respectant des adresses d'implantation finales des différentes sections, qui sont soit fournies soit fixées par convention.

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 ! 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 permettront *d'adapter* ce code lorsque les adresses seront connues de manière absolues.

Un fichier relogeable contient donc (au moins) :
- le code des *sections* `.text` et éventuellement `.data`, assemblé avec des informations liées aux positions *relatives* des instructions et données par rapport au début de chaque section.
- une *table de relocation* pour chaque section, indiquant pour chaque adresse relative à mettre à jour au moment de l'implantation : la position dans la section de l'emplacement à mettre à jour, le mode de relocation (définissant les calculs à effectuer pour déterminer les bonnes adresses effectives), ainsi que le numéro dans la table des symboles du symbole correspondant à l'opérande relogé.
- une *table des symboles* contenant pour chaque étiquette son adresse relative au début de la section dans laquelle elle est définie (cas des symboles locaux ou globaux définis dans

le fichier considéré), ou bien 0 lorsque l'étiquette n'est pas définie dans ce fichier (cas des symboles globaux indéfinis).

5.3.2 Relocation en pratique

Les exemples suivants illustrent les problèmes rencontrés nécessitant de la relocation :

Exemple 1

```
.text
ADDI $3, $0, 12345        # Met la valeur 12345 dans le registre 3
SW $3, X                  # Ecrit le contenu du registre 3 a l'adresse X

.data
X: .word 0x0              # Declaration d'un mot de 32 bits initialise a 0
```

Pas de problème pour l'instruction `ADDI`, toujours codée de la même manière.

En revanche le codage du `SW`[2] dépend de *l'adresse à laquelle correspond l'étiquette « X »* ! En effet il est nécessaire de calculer le décalage entre l'adresse du `SW` et celle de la donnée `X`. Ceci n'est possible qu'à partir du moment où les adresses d'implantation des sections `.text`, `.data` et `.bss` sont connues. Or, ce n'est pas le cas au moment de l'assemblage puisque ces adresses ne seront déterminées que lors de la phase d'implantation, c'est à dire lors de l'édition de liens finale, ou bien lors du chargement du programme en mémoire avant son exécution.

Comment alors coder la pseudo-instruction `SW` si le décalage entre son adresse et celle de la donnée `X` ne peut pas être calculé ?

Exemple 2

```
.data
X: .byte 0xAB       # Declaration d'un octet initialise a 0xAB
Y: .word Z          # Mot de 32 bits Y, en fait une reference a l'adresse Z
Z: .word 0x0        # Declaration d'un mot de 32 bits initialise a 0
```

Dans cet exemple la valeur déclarée derrière l'étiquette `Y` correspond en fait à l'adresse associée à l'étiquette `Z`. Comme dans le cas précédent, il est nécessaire de connaître l'adresse d'implantation de la section `.data` pour déterminer celles des données `X`, `Y` et `Z`.

Exemple 3

```
JAL fonction # Appel a la procedure "fonction" (met PC a la bonne adresse)
NOP          # On ne fait rien ici (rappelez vous du delay slot — section 2.8)
B end        # Retour ici apres execution de "fonction"; branchement sur end
NOP          # Delay slot

fonction:
NOP          # On ne fait... rien!
JR $31       # Fin de la procedure "fonction", retour a l'appelant

end:         # The End!
```

2. 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`. Voir section 4.3.

Ici deux étiquettes sont utilisées par des instructions de branchement (`B`) et de saut (`JAL`). 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 est dans la même section `.text` que le branchement lui-même, le décalage peut être calculé directement lors de l'assemblage. Mais ce n'est pas toujours le cas : dans un programme multi-fichiers, si l'étiquette désignant la destination du saut est un symbole externe non résolu (situé dans la section `.text` d'un autre fichier objet assemblé séparément), alors ce décalage n'est pas calculable.

— Le problème est différent pour l'instruction de saut vers l'adresse désignée par l'étiquette `fonction`. D'après les spécifications de l'instruction `JAL`, lors de son exécution, l'adresse de destination du saut à effectuer est calculée à partir de l'opérande (ce que l'on cherche justement à remplir) *ET* de la valeur courante de `PC`, soit l'adresse de l'instruction `JAL` + 4.[3] En effet, les 4 bits de poids forts du PC déterminent une zone mémoire (un segment) de 256 Mo, dont l'adresse est un multiple de 2^{28} ; le saut est effectué dans les limites de ce segment.
Donc, même si la position de l'étiquette `fonction` est ici connue *par rapport à* la section où se trouve l'instruction `JAL` (la section `.text`), il manque la position effective en mémoire de cette section—l'offset par rapport au début de la section—pour pouvoir déterminer le codage complet de l'opérande.

En résumé, **le codage 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` et `.bss` (donc instructions et données) sont connues**.

5.4 Relocation au format ELF pour le MIPS

Plusieurs formats relogeables existent. L'un des premiers mis au point fut le format `COFF` (Common Object File Format), initialement développé sous Unix et encore utilisé sur les systèmes Windows. Mais le format le plus répandu à ce jour est `ELF` [6], utilisé par de nombreux systèmes d'exploitation dont Linux, Mac OS ou SunOS. Il est conçu pour assurer une certaine portabilité entre différentes plateformes. Il s'agit d'un format standard pouvant supporter l'évolution des architectures et des systèmes d'exploitation.

La suite de cette section présente les informations nécessaires à la relocation au sein du format ELF. Des informations techniques supplémentaires sont décrites dans l'annexe C.

5.4.1 La table des symboles

Tous les symboles définis et/ou utilisés dans un fichier source sont regroupés au sein d'une *table de symboles* dans le fichier binaire issu de la compilation ou de l'assemblage de ce fichier source. La table de symboles d'un fichier binaire est l'élément central de l'architecture interne de ce fichier. En effet, c'est dans cette table qu'on retrouve les adresses associées aux noms

3. Souvenez vous que lors de l'exécution effective d'une instruction, la valeur de `PC` a déjà été incrémentée de la longueur d'un mot.

définis par le programmeur, mais aussi les adresses où sont implantées les différentes sections du programme en vue de son exécution.

Le contenu d'une table de symboles d'un fichier binaire au format ELF peut être affiché sous une forme lisible au moyen de l'outil `readelf`, ou plus précisément de l'outil `mips-elf-readelf` dans le cas du processeur MIPS (annexe A). Par exemple (voir en fait l'annexe B.2) :

```
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: 00000000     0 OBJECT  GLOBAL DEFAULT    1 _start
     7: 00000014     0 NOTYPE  LOCAL  DEFAULT    1 write
     8: 00000024     0 NOTYPE  LOCAL  DEFAULT    1 end
     9: 00000008     0 NOTYPE  LOCAL  DEFAULT    3 Z
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 X
    11: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 Y
```

On remarque dans cet exemple que tous les symboles sont locaux, sauf un défini localement mais exporté (directive `.global`). Certains sont des symboles de section (`.text`, `.data`, etc.). Les autres symboles sont des symboles de programme : leur type est noté `NOTYPE`. La section de définition de chaque symbole figure dans la colonne `Ndx` sous la forme d'un numéro de section : il s'agit d'un numéro dans la *table des sections*, sorte de table des matières du contenu du fichier. Pour plus de détails, voir l'annexe C.2.5, page 69.

5.4.2 Table de relocation

Les informations nécessaires à la relocation sont définies dans des *tables de relocation*, qui seront incluses par l'assembleur dans le fichier objet binaire relogeable. Une table est associée à chaque section contenant des symboles à reloger. La table associée à la section `.text` (respectivement `.data`) est nommée `.rel.text` (respectivement `.rel.data`).

Chaque entrée (ou ligne) d'une table de relocation est définie par les informations suivantes :

- `offset` : la position de l'entrée à modifier (une instruction ou une donnée), en nombre d'octets par rapport au début de la section à laquelle la table est associée ;
- `type` : le mode de relocation (voir plus loin, section 5.4.4) ;
- `symbol` : l'instruction ou la donnée à reloger fait référence à un symbole s d'adresse inconnue. S'il est local, la valeur `symbol` est l'*index* (le numéro), dans la table des symboles, du symbole de section correspondant à la section où s est défini. Si s est un symbole global (défini ou indéfini), `symbol` est l'index, dans la table des symboles, du symbole en question.

Dans l'exemple suivant, l'entrée à reloger est un mot situé 4 octets après le début de la section `.data` (à l'adresse de l'étiquette `Y`). Le symbole `Z` à reloger est local et est défini dans la section `.data`. L'entrée correspondante dans la table de relocation `.rel.data` est :

```
$ mips-elf-readelf -r exemple.o
Relocation section '.rel.data' at offset 0x358 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000202 R_MIPS_32         00000000   .data
```

On constate que le symbole utilisé dans cette entrée (colonne `Sym. Name`) est bien le symbole de section de la section de définition du symbole `Z` (section `.data`).

Dans cet autre exemple, l'entrée à reloger est l'instruction `JAL`, la première de la section `.text`, et le symbole recherché est l'étiquette `fonction`. Ce symbole est local, aussi défini dans la section `.text`. La table `.rel.text` contient donc :

```
Relocation section '.rel.text' at offset 0x350 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000104 R_MIPS_26         00000000   .text
```

Le symbole utilisé dans cette entrée est là encore le symbole de section de la section de définition de `fonction`.

5.4.3 Champ `addend`

Une dernière donnée pour la relocation d'une entrée est le `addend`, c'est-à-dire la valeur binaire présente dans l'emplacement qui va être modifié par la relocation. Cette valeur, contenue dans le fichier binaire relogeable, sera récupérée par l'éditeur de liens ou le chargeur, et utilisée avec les informations de la table de relocation concernée pour déterminer le codage définitif.

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

Dans le cas des symboles définis, mais dont on ne connaît pas encore l'adresse définitive, l'`addend` est en général l'adresse relative du symbole par rapport au début de sa section de définition. Dans le cas des symboles indéfinis (référence à un symbole défini dans un autre fichier), l'assembleur n'a même pas d'adresse relative, et ne peut que laisser une valeur arbitraire pour l'`addend`, en général 0.

5.4.4 Modes de relocation du MIPS

Le *mode de relocation* détermine le calcul spécifique à effectuer pour reloger une entrée (instruction ou donnée). Une quinzaine de modes existent (définis dans la documentation MIPS [7]), mais seul ceux nécessaires à notre projet sont ici décrits (avec quelques restrictions).

R_MIPS_32 Ce mode est utilisé pour reloger une donnée en section `.data` (cf. premier exemple). Les 32 bits de la donnée sont modifiés.

R_MIPS_26 Ce mode correspond à la relocation des instructions de saut de type `J` (cf. deuxième exemple). Les 6 bits de poids fort de l'instruction (opcode) ne seront jamais changés 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 (`LW`, `SW`, `LUI`). Seuls les 16 bits de poids faible de chaque

entrée sont mis à jour. Des entrées de ces types apparaissent toujours par couple dans une table de relocation : chaque relocation `R_MIPS_HI16` et immédiatement suivie d'une relocation de type `R_MIPS_LO16`. [4]

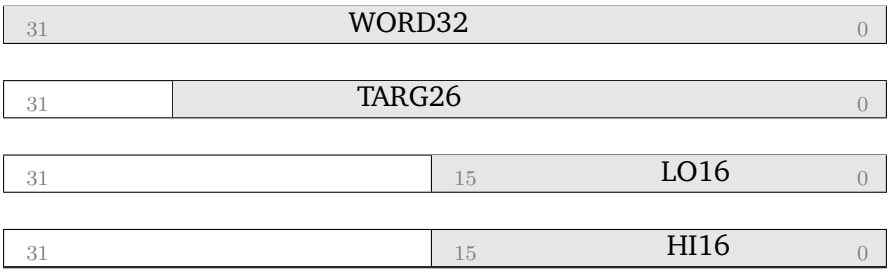


FIGURE 5.2 – Selon le mode de relocation 32, 26 ou 16 bits de poids faible sont mis à jour. Les bits de poids fort (codage de l'instruction) restent inchangés.

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

- `P` désigne l'adresse de l'emplacement (instruction ou réservation de donnée) à reloger, c'est-à-dire l'adresse d'implantation de la section + l'adresse relative de l'emplacement dans la section (`offset`).
- `A` désigne la valeur de l'`addend` : il s'agit de la valeur présente provisoirement dans l'emplacement à reloger, plus précisément dans les 32, 26 ou 16 bits de poids faible de l'emplacement, selon le mode de relocation. Cette valeur peut dans certains cas être négative (bit de poids le plus fort à 1), il faut donc dans les calculs utiliser l'extension signée à 32 bits de cette valeur, si nécessaire.
- `S` désigne l'adresse (c'est à dire la valeur) du symbole référencé par l'entrée de relocation. On rappelle que ce symbole peut être un symbole de section (cas des symboles locaux) ou bien un symbole de programme (cas des symboles globaux).
- `AHL` est une valeur 32 bits calculée à partir des `addend` de deux entrées de relocation `R_MIPS_HI16` et `R_MIPS_LO16` consécutives. Si `AHI` et `ALO` sont les `addends` d'une paire d'entrées `HI16` et `LO16`, de 16 bits chacun, `AHL` est alors calculé par :
  `AHL = (AHI` $\ll$ `16) + (short)ALO`.
  Par exemple, si `AHI = 0xABCD` et `ALO = 0x12`, alors `AHL = 0xABCD0012`.
  De même, si `AHI = 0xABCD` et `ALO = 0xFFF0`, alors `AHL = 0xABCCFFF0`.

Avec ceci, les calculs de relocation sont les suivants :

4. En réalité non, une entrée `HI16` est suivie par une *ou plusieurs* entrées `LO16`.

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 ou bien : (((A ≪ 2) + S) | (P & 0xF0000000)) ≫ 2
R_MIPS_HI16	P + 16 bits	16	((AHL + S) - (short)(AHL + S)) ≫ 16 ou bien : ((AHL + S) ≫ 16) & 0x0000FFFF
R_MIPS_LO16	P + 16 bits	16	AHL + S ou bien : (AHL + S) & 0x0000FFFF

5.4.5 ...

Pour passer le mal de tête, rien de tel qu'un exemple complet et commenté, non ?

Allez donc vous coucher, puis faites demain un tour en annexe B et tout ira mieux !

Chapitre 6

Spécifications du simulateur

Ce chapitre présente les spécifications du simulateur à réaliser, concernant les caractéristiques de la machine MIPS à simuler et les commandes de contrôle (interface utilisateur).

6.1 Machine simulée

6.1.1 Adresses des sections

Des contraintes sur les adresses mémoire des sections devront toujours être vérifiées :

- Les adresses des sections `.text` et `.data` doivent être alignées sur des pages. Concrètement, elles seront toujours des multiples de 0x1000 octets (soit 4096 octets ou 4Ko).
- A moins qu'elle soit définie explicitement, l'adresse de la section `.data`, si elle est présente, est alignée sur la page suivant le programme. [1] Si par exemple la section `.text` est en 0x1000, les données seront implantées en 0x2000 si la taille de la section d'instructions est de 200 octets, en 0x3000 si elle est de 6000 octets, etc.
- La section `.bss` (réservation d'espaces mémoire) commencera toujours immédiatement après la section `.data`.

Plusieurs cas d'utilisation du simulateur se présentent :

- Les adresses de sections peuvent être paramétrées au lancement du simulateur. Par exemple, la commande `simips -t 0x4000 -d 0x1000` placera les sections `.text` et `.data` aux adresses respectives `0x4000` et `0x1000`. Si seule l'option `-t` est présente, la section `.data` sera positionnée selon les contraintes décrites ci-dessus. L'option `-d` seule n'est par contre pas autorisée.
- Par défaut, si aucune option n'est spécifiée au lancement du programme, la machine simulée utilisera l'adresse 0x400000 [2] pour la section `.text` (première instruction).
- Si le fichier chargé est exécutable, les adresses de ses sections sont définies de manière absolues. Ces adresses seront alors utilisées.

1. Cette adresse dépend de la taille du programme et ne peut être déterminée que lors de la lecture du fichier.
2. Cette adresse est en fait celle utilisée par défaut par l'éditeur de liens `mips-elf-ld` pour placer la section `.text` d'un programme exécutable. Les adresses peuvent aussi être spécifiées à l'édition de liens avec les options `-Ttext` `-Tdata` et `-Tbss` de `mips-elf-ld`.

En cas d'incompatibilité des adresses (chevauchement, pas assez de place...) un comportement et un message appropriés seront les bienvenus !

6.1.2 Représentation de la mémoire

L'espace d'adressage d'un processeur 32 bits est a priori de 4Go (mémoire logique). Évidemment il ne s'agit pas de représenter physiquement l'ensemble cet espace adressable, charge à vous de trouver une structure permettant de représenter les données nécessaires pour simuler la mémoire. Le système simulé gérera *deux types de mémoire* : la mémoire centrale, dédiée au programme et à ses données, et la mémoire du framebuffer qui gère l'affichage.

L'option `-s <taille>` de `simips` (pour *size*) spécifie la taille de la mémoire *centrale* de la machine simulée, qui sera implantée de façon contiguë à partir de l'adresse `0x00000000`. Par exemple `-s 0x10000` signifie une mémoire de 64Ko, ayant une plage d'adresses allant de `0x0000` à `0xFFFF`.

En l'absence de ce paramètre, la taille par défaut sera de `0x1000000` (16 Mo).

6.1.3 Convention sur la pile

Pour simplifier l'usage du simulateur, la valeur du pointeur de pile est initialisée par défaut à l'adresse du mot le plus haut de la mémoire centrale (car elle doit nécessairement être multiple de 4).

6.1.4 Point d'entrée

Le point d'entrée d'un programme est l'adresse de sa première instruction à exécuter. L'exécution de la commande `run` après le chargement devra démarrer à cette adresse.

Si le fichier ELF chargé est exécutable, le point d'entrée est défini de manière explicite. [3]

Dans le cas où le fichier est relogeable, le point d'entrée est par défaut l'adresse de la section `.text` (l'exécution commence par la première instruction de la section).

6.1.5 Le framebuffer

Un framebuffer est une zone de mémoire contiguë dédiée à l'affichage d'une image par un périphérique vidéo. Cette zone mémoire peut donc être vue comme un tableau de pixels. Le framebuffer simulé fourni dans ce projet a une taille de 320x200 pixels. Chaque pixel est codé sur 8 bits et sa valeur correspond au niveau de gris affiché dans le framebuffer (0 = noir, 255 = blanc). Son utilisation est décrite section 7.1.3.

Pour pouvoir piloter ce framebuffer depuis vos programmes MIPS simulés, certaines adresses de l'espace d'adressage de votre machine simulé doivent correspondre (être *mappées*) à la zone

3. `mips-elf-ld` défini par défaut le point d'entrée comme l'adresse du symbole `_start`, s'il est présent, ou comme l'adresse de `.text`. Un autre symbole peut être spécifié comme point d'entrée à l'aide de l'option `-e`.

mémoire du framebuffer. Il vous est demandé de mapper le début du framebuffer à l'adresse `0xffff0600`, c'est-à-dire à la fin de l'espace d'adressage comme indiqué sur la figure 6.1. Cette adresse est indépendante de l'option `-s` qui spécifie uniquement la taille de la mémoire centrale et n'impacte en aucune façon le fonctionnement du framebuffer.

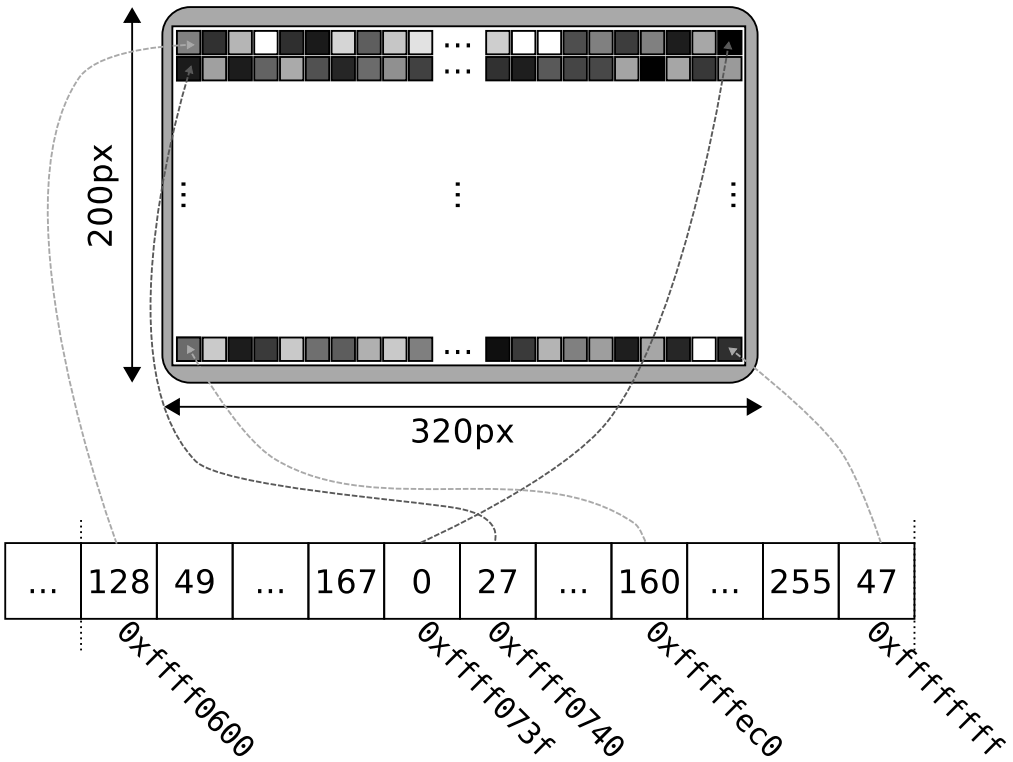


FIGURE 6.1 – Le framebuffer

Si un programme écrit dans la zone `0xffff0600-0xffffffff`, il écrira donc dans la mémoire du framebuffer ce qui modifiera l'affichage.

Vous pouvez enfin ajouter les options `-fb` et `-no-fb` pour activer/désactiver le framebuffer du simulateur. [4]

6.2 Interface utilisateur

L'interface utilisateur du simulateur doit être sous forme d'un shell linux, avec les commandes entrées au clavier et affichage dans la console. Elle doit comporter au minimum les commandes explicitées dans cette section. Ces commandes permettent de charger un fichier objet en mémoire, d'afficher le code assembleur correspondant (désassemblage), d'afficher et de modifier l'état virtuel de la machine (valeurs des registres, de la mémoire) et de simuler l'exécution du programme.

4. Cette petite fenêtre est parfois très pénible quand on n'a pas à s'en servir...

Outre un respect scrupuleux de la syntaxe des commandes exigées, [5] votre shell devra vérifier si le format des paramètres d'une commande est correct. Par exemple, si un nom de registre est incorrect (`$41`) ou que des valeurs numériques ne correspondent pas aux spécifications, un message d'erreur doit être affiché et la main rendue à l'utilisateur.

Pour l'interface, commencez par quelque-chose de simple. Lorsque vous aurez rempli le cahier des charges minimal, vous pourrez améliorer votre shell en lui ajoutant vos propres commandes ou en améliorant son ergonomie. Sur ce dernier point, l'utilisation de la librairie `GNU Readline` est une option intéressante [6] car elle permet entre autre la gestion d'un historique, l'édition des lignes de commandes et la gestion de raccourci.

6.2.1 Charger un programme : `load`

— Syntaxe : `load <filename>`
— Paramètres :
`filename` : nom du fichier objet à charger
— Description : Les données lues dans le fichier `filename` sont placées en mémoire en respectant les contraintes sur les adresses de sections vues en 6.1.1.
Le fichier `<filename>` doit être au format ELF, exécutable ou relogeable sans symbole externe.
Dans le cas relogeable, la relocation devra être effectuée avec les adresses de section définies au lancement du simulateur (voir section 7.1.2).

6.2.2 Afficher le code désassemblé : `dasm` (display assembler)

— Syntaxe : `dasm [n | all]`
— Description : Affiche dans la console le code désassemblé, c'est-à-dire les instructions en langage assembleur correspondant au code binaire.
— Paramètres :
  — Sans option, `dasm` désassemble l'instruction courante à l'adresse `$pc`.
  — Avec un paramètre entier `n` : désassemble `n` instructions à partir de `$pc`.
  — Avec `all` : désassemble la totalité des trois zones `.text`, `.data` et `.bss`.
  Les informations affichées, une instruction ou donnée par ligne, seront : l'adresse d'implantation en mémoire, le code binaire puis le code désassemblé. L'affichage des symboles en lieu et place des adresses correspondantes sera apprécié.
  Pour les données (sections `.data` et `.bss`), il n'est pas possible de remonter aux directives [7] et seuls l'adresse et le code binaire seront affichés.
  Inspirez-vous du résultat de `objdump -D` (voir exemple B.2).

6.2.3 Afficher les registres : `dreg` (display register)

— Syntaxe : `dreg [<registerName>]`

5. Sans quoi il sera beaucoup plus difficile de le tester efficacement.
6. Avec un gros point noir, la mémoire qui fuit en sortie...grr.
7. Comment savoir si `0xABCDEF12` correspond à `.word 0xABCDEFGH` ou à `.byte 0xAB`, `.byte 0xCD`, `.byte 0xEF` puis `.byte 0x12` ? ? Impossible, à moins qu'il y ait une étiquette à chaque directive...

— Paramètres :
`registerName` (optionnel) : nom d'un des registres
— Description : Affiche dans la console, au format hexadécimal, la valeur du registre donné en paramètre. S'il n'y a aucun paramètre, `dreg` affiche les valeurs de chacun des registres. Formattez correctement (plusieurs valeurs par lignes), pour voir toutes les valeurs en même temps dans la console.

6.2.4 Modifier une valeur dans un registre : `sreg` (set register)

— Syntaxe : `sreg <registerName> <value>`
— Paramètres :
`registerName` : nom d'un registre.
`value` : valeur numérique valide, entrée au format hexadécimal (préfixe `0x`) ou décimale.
— Description : Écrit la valeur donnée dans le registre passé en paramètre.

6.2.5 Afficher la mémoire : `dmem` (display memory)

— Syntaxes : `dmem <address>` ou `dmem <address1> <address2>`
— Paramètres :
`address` : valeur hexadécimale représentant une adresse sur 32 bits.
— Description : Cette primitive affiche sur la console le contenu de la mémoire. L'affichage se fera à raison de 16 octets par ligne séparés par des espaces (un octet sera affiché par deux chiffres hexadécimaux). Chaque ligne commencera par l'adresse hexadécimale (32 bits) de l'octet le plus à gauche de la ligne. [8] La commande `dmem <address>` affiche 3 (par exemple) lignes de 16 octets à partir de l'adresse donnée. `dmem <address1>-<address2>` affiche tous les octets entre les deux adresses en paramètres.

6.2.6 Modifier une valeur en mémoire : `smem` (set memory)

— Syntaxe : `smem <address> <value> [nbOctets]`
— Paramètres :
`address` : valeur représentant une adresse mémoire valide
`value` : valeur numérique, au format hexadécimale ou décimale.
`nbOctets` : paramètre optionnel indiquant le nombre d'octets à écrire en mémoire (1,2 ou 4). S'il n'est pas présent, un seul octet est écrit.
— Description : Écrit dans la mémoire, à l'adresse fournie en paramètre, la valeur fournie en paramètre. Selon la valeur de l'argument `nbOctets`, 1, 2 ou 4 octets seront écrits en mémoire à partir de `address`. Les valeurs `value` et `nbOctets` devront être compatibles. Par exemple :
  — `smem 0x1000 0xAB` écrit 0xAB en 0x1000
  — `smem 0x1000 0xABC 2` écrit 0x0A en 0x1000 et 0xBC en 0x1001
  — `smem 0x1000 0xABC 1` lève une erreur.

8. Format similaire au résultat de la commande `od -t xC --address-radix=x` sur un fichier binaire. Si si, essayez voir...

6.2.7 Effectuer une copie d'écran : `sshot` (screenshot)

— Syntaxe : `sshot <fichier.ppm>`
— Paramètres :
`fichier.ppm` : nom du fichier utilisé pour sauvegarder la copie d'écran. La sauvegarde est réalisée au format ppm.
Par exemple :
— `sshot screen1.ppm` sauvegarde le framebuffer courant dans le fichier screen1.ppm.

6.2.8 Exécuter à partir d'une adresse : `run`

— Syntaxe : `run [<address>]`
— Paramètres :
`address` : valeur hexadécimale (facultatif)
— Description : Charge `PC` avec l'adresse fournie en paramètre et lance le microprocesseur. Si le paramètre est omis, l'exécution commence à la valeur courante de `PC`.

6.2.9 Exécution pas à pas (ligne à ligne) : `step`

— Syntaxe : `step`
— Description : Provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre `PC` puis rend la main à l'utilisateur. Si l'on rencontre un appel à une procédure avec retour (`JAL`), cette dernière s'exécute complètement jusqu'à l'instruction de retour incluse (`JR $ra`). La main est alors rendue à l'utilisateur sur l'instruction suivant le saut (donc celle *après* son *delay slot*).

6.2.10 Exécution pas à pas (exactement) : `stepi` (step into)

— Syntaxe : `stepi`
— Description : Cette primitive provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre `PC` puis rend la main à l'utilisateur. Si l'on rencontre un saut, seule l'instruction dans le *delay slot* puis le saut sont exécutés.

6.2.11 Ajouter un point d'arrêt : `addbp` (add breakpoint)

— Syntaxe : `addbp <address>`
— Paramètre :
`address` : valeur hexadécimale d'adresse
— Description : Ajoute un point d'arrêt à l'adresse fournie en paramètre. Lorsque le compteur ordinal `PC` sera égal à cette valeur, l'exécution sera interrompue et la main rendue à l'utilisateur.

6.2.12 Supprimer un point d'arrêt : `rmbp` (remove breakpoint)

— Syntaxe : `rmbp [<address>]`

— Paramètres :
`address` : valeur hexadécimale d'adresse (facultative).
— Description : Enlève le point d'arrêt à l'adresse fournie en paramètre, s'il y en a un. Si le paramètre est omis, cette commande efface tous les points d'arrêt.

6.2.13 Afficher les points d'arrêt : `dbp` (display breakpoint)

— Syntaxe : `dbp`
— Description : Affiche sur la console de visualisation l'adresse de tous les points d'arrêt positionnés dans la mémoire.

6.2.14 Afficher l'aide : `help`

— Syntaxe : `help [<command>]`
— Paramètres :
`command` : nom de la commande sur laquelle l'aide est demandée.
— Description : Affiche l'aide d'une commande et sa syntaxe. Si aucun argument n'est fourni, affiche la liste de toutes commandes du simulateur avec une description très brève (moins d'une ligne).

6.2.15 Quitter le programme : `exit`

— Syntaxe : `exit`
— Description : Quitte le simulateur.

Chapitre 7

Travail à réaliser

Après une présentation des modules fournis, ce chapitre décrit le cahier des charges minimal de ce projet. Les extensions que vous devrez y apporter sont ensuite proposées, essentiellement la réécriture des modules fournis. Enfin, quelques conseils vous sont donnés pour la bonne réalisation de votre simulateur.

7.1 Code et modules fournis

Trois modules sont fournis dans ce projet, sous forme d'une spécification (fichier `.h` docummenté) et d'un fichier objet binaire (extension `.o`). Le code source n'est pas distribué.

Les deux premiers modules concernent l'extraction des informations d'un fichier ELF et la relocation. Le dernier permet de créer une fenêtre graphique pour afficher le contenu du framebuffer.

Vous devrez en premier temps *utiliser* ces modules dans votre projet. Leur réécriture fait partie des extensions, et ne devra donc pas être commencée avant d'avoir terminé la partie minimale des spécifications.

7.1.1 Module de lecture du ELF : `elf_reader`

Ce module fourni des fonctions simples d'utilisation pour extraire les différentes informations contenues dans un fichier ELF. Les formats de données sont ceux définis dans le fichier `elf.h` de votre système.

Les prototypes des fonctions fournies sont :

```
struct elf_descr *read_elf(const char *filename);
void close_elf(struct elf_descr *elf);

bool get_text_section(const struct elf_descr *elf, uint8_t **data,
                      size_t *size, uint32_t *addr, uint8_t *align);
bool get_data_section(const struct elf_descr *elf, uint8_t **data,
                      size_t *size, uint32_t *addr, uint8_t *align);
bool get_bss_section(const struct elf_descr *elf,
```

```
                    size_t *size, uint32_t *addr, uint8_t *align);
bool get_rel_text_section(const struct elf_descr *elf,
                          Elf32_Rel **data, size_t *size);
bool get_rel_data_section(const struct elf_descr *elf,
                          Elf32_Rel **data, size_t *size);
bool get_string_table(const struct elf_descr *elf,
                      char **strtab, size_t *size);
bool get_symbol_table(const struct elf_descr *elf,
                      Elf32_Sym **symtab, size_t *size);
uint16_t get_elf_type(const struct elf_descr *elf);
uint32_t get_entry_point(const struct elf_descr *elf);
```

La structure détaillée d'un fichier ELF (ses principales tables ou sections) ainsi qu'une explication détaillée du module sont disponibles dans l'annexe C. L'API complète des fonctions est disponible directement dans le fichier `elf_reader.h` fourni.

Une partie importante de votre travail sera de *comprendre* la structure ELF et les spécifications du module pour l'utiliser correctement, afin de charger des fichiers tests dans votre simulateur.

La réécriture de ce module fait partie des extensions du projet (section 7.2.3).

7.1.2 Module de relocation

Ce module est à utiliser après `elf_reader`. Il permet de réaliser, lorsque c'est nécessaire, l'opération de relogement des sections `.text` et `.data`. Il contient trois fonctions seulement, qui travaillent à partir des informations extraites du fichier ELF et des adresses auxquelles implanter les sections :

```
struct symbole {
    char* nom;
    uint32_t adresse;
};

void traduit_table_symboles(const Elf32_Sym *symtab, size_t symtab_size,
                            const char *strtab,
                            struct symbole **table_symboles,
                            size_t *nb_symboles);

void reloge_symboles(const Elf32_Sym *symtab, size_t symtab_size,
                     const char *strtab,
                     uint32_t addr_text, uint32_t addr_data, uint32_t addr_bss,
                     struct symbole **table_symboles, size_t *nb_symboles);

void reloge_section(uint32_t addr, uint8_t *data,
                    const Elf32_Rel *rel_table, size_t rel_size,
                    const struct symbole *table_symboles);
```

L'API complète du module est disponible directement dans le fichier `relocation.h` fourni.

Là encore, votre travail consistera dans un premier temps à *utiliser* ce module pour réaliser la relocation dans le cas où un fichier ELF relogeable est chargé, ou pour simplement traduire la table des symboles dans un format clair.

Comme pour le module précédent, la réécriture de ce module fait partie des extensions du projet.

7.1.3 Module du framebuffer

Le module du framebuffer (composé des fichiers `framebuffer.h` et `framebuffer.o`) est très simple d'utilisation.

```
uint8_t *framebuffer_init_display();
void framebuffer_close_display();
```

La fonction `framebuffer_init_display` initialise l'affichage graphique en ouvrant une nouvelle fenêtre. Cette fonction doit être appelée au démarrage de l'application. Une fois crée, le contenu de la fenêtre est automatiquement réactualisé toutes les 0.5 secondes. Cette valeur est choisie relativement basse afin de ne pas saturer certaines machines qui auraient une puissance (graphique) limitée.

Le contenu du framebuffer est accessible via l'adresse retournée par la fonction d'initialisation. Chaque pixel tient sur un octet qui code le niveau de gris correspond (0 = noir, 255 = blanc). Les pixels sont placés séquentiellement en mémoire, ligne par ligne. `framebuffer[0]` contient ainsi la valeur du pixel situé dans le coin en haut à gauche de la fenêtre, `framebuffer[319]` le pixel du coin en haut à droite et `framebuffer[320*200-1]` le pixel du coin en bas à droite.

À noter qu'une modification d'un pixel ne force pas une réactualisation immédiate de l'image affichée. Il peut donc y avoir un délai d'une demi-seconde avant la propagation de la mise-à-jour à l'écran.

A la fin du programme, la méthode `framebuffer_close_display` permet de fermer la fenêtre [1].

7.2 Cahier des charges minimal et extensions

7.2.1 Fonctionnalités minimales

Les **objectifs minimum** du projet sont de mettre en oeuvre un simulateur d'une machine MIPS que l'on appellera `simips`. L'ensemble des spécifications décrites au chapitre 6 devront être intégrées, c'est-à-dire :

- la machine simulée (*processeur*, *mémoire centrale* et *mémoire vidéo*) respectant les spécifications de la section 6.1.
- une interface utilisateur respectant les spécifications de la section 6.2.

Cette implantation minimale devra utiliser les modules fournis pour la lecture d'un fichier ELF, la relocation et le framebuffer.

7.2.2 Validation & qualité du code

Il est important de **valider** correctement les différents éléments de votre projet. Pour cela, un point important sera de vous constituer une base de fichiers tests écrits en assembleur. L'utilisation des outils pour le MIPS décrits en annexe A (`mips-elf-as`, `mips-elf-ld`, `mips-elf-objdump`,

1. Normalement, toute la mémoire devrait être libérée. Hélas, il semble que la librairie SDL utilisée entraîne de nombreuses fuites mémoire, même utilisée correctement (grrr...). Un Carambar si vous arrivez à fermer SDL proprement sans fuite !

`mips-elf-readelf...`) sera particulièrement importante tout d'abord pour bien comprendre les informations contenues dans les fichiers ELF, puis pour comparer le résultat des opérations de votre simulateur (désassemblage, relocation...) avec ceux de ces outils. Pour la simulation, déterminez les sorties attendues de vos programmes, et comparez-les à celles obtenues !

De même la **qualité de votre implantation** sera évaluée : un code bien conçu, clair et facile à faire évoluer est toujours préférable à plus de fonctionnalités (par exemple une extension) mais mal codé (compliqué, redondances... ce qui tourne très vite en bugs).

7.2.3 Extensions

À condition d'avoir terminé proprement le cahier des charges minimal, les extensions suivantes pourront être réalisées. [2] Il s'agit essentiellement de remplacer les fichiers objets fournis par les enseignants par vos propres modules.

Extension	Difficulté	Priorité
Écriture du module de framebuffer	+	+
Écriture de votre propre module de relocation	++	++
Écriture de votre propre module de lecture d'un fichier ELF	+++	++
Vous manquez d'idées ? Nous non !	+ à +++	-

Cas particulier de `elf_reader.o`

La réécriture de ce module sera réalisée à partir de la spécification détaillée du format ELF, claire et complète, qu'on trouvera en [6] (il est vraiment intéressant d'apprendre à utiliser un « vrai » document de travail). La première partie "*Object files*" est nécessaire et suffisante. Se référer également à l'annexe C.

Remarque : il existe une librairie `libelf` fournissant une interface de plus bas niveau que `elf_reader` pour la lecture et l'écriture de fichiers ELF. Cependant cette librairie est difficile d'utilisation et très peu (voire mal) documentée. Dans ce projet, il est explicitement demandé de ne pas utiliser `libelf` [3].

7.3 Quelques conseils de programmation

7.3.1 Programmation modulaire et tests autonomes

Par rapport à la plupart des TPs réalisés cette année, une des complexité de ce projet est sa *taille*, i.e. la quantité des tâches à réaliser : représentation des éléments du MIPS (registres, mémoire...), instructions à exécuter, chargement d'un fichier ELF, contrôle du simulateur, etc.

2. Elles seront bien entendu prises en compte dans l'évaluation du projet !
3. et en toute honnêteté, il est plus intéressant mais aussi plus simple (si si !) de ne pas utiliser `libelf`.

Avant de partir tête baissée dans du codage, il est essentiel de bien définir un découpage en *modules*. Ils doivent être clairement spécifiés : rôle, structures de données éventuelles, fonctions proposées avec une signature précise.

Dans le partage de votre travail, vous serez amenés à *utiliser* les modules programmés par vos collègues. Il est donc nécessaire de bien comprendre leur usage, même si vous ne connaissez ou maîtrisez pas leur contenu. Une bonne spécification est donc fondamentale pour que la mise en commun des différentes parties du projet se fasse sans trop de heurts (vous verrez, ce n'est pas toujours simple...).

Dans la mesure du possible (pas toujours facile), chaque module devra être testé de manière autonome c'est-à-dire hors contexte de son utilisation dans le simulateur. Il s'agit de tester avec des entrées contrôlées et de vérifier que les sorties sont bien conformes à la spécification.

7.3.2 Approche incrémentale

Il est fortement conseillé d'adopter une *approche incrémentale* dans la réalisation de votre projet. L'erreur à faire serait de vouloir « tout faire » d'abord, puis seulement ensuite d'attaquer la vérification. Cette stratégie [4] est très risquée et ne vous garanti pas d'être en mesure de fournir un rendu convenable (même incomplet) à la fin du projet.

Un scénario pertinent serait par exemple de terminer le plus rapidement possible un simulateur couvrant toutes les étapes principales, mais avec un sous-ensemble de fonctionnalités ou ne gérant que des cas simples. Ceci pourrait être :

- être capable de lire un fichier objet très simple : une section `.text` (en 0x0) contenant une ou quelques instructions `ADD`, pas de relocation, aucun symbole ;
- désassembler ce programme : affiche les instructions reconnues. Retrouvez-vous le programme ?
- afficher les registres, exécuter trois fois pas à pas (`step`), réafficher : le nouvel état est-il correct ?
- le tout sans interface en ligne de commande, mais en appelant les fonctions en dur dans un programme de test.

Ce mode de fonctionnement est bien plus pertinent que de fournir au final un magnifique programme capable de charger tous les programmes de monde et gérant tous les modes de relocation, mais sans qu'on puisse exécuter la 1ère instruction...

Bien entendu, chacune des étape sera ensuite complétée (voire complètement reprise) pour couvrir les spécifications demandées, mais il est important d'avoir toujours une base fonctionnelle minimale à présenter à votre client (nous !), à tout instant. Imaginez que le rendu soit subitement avancé de deux jours, [5] et bien vous rendrez dans l'état courant, sans (trop) de stress ; et hop.

7.3.3 Factorisation

Très fréquemment dans ce projet, vous retrouverez des portions de codes communes ou similaires à plusieurs éléments : traitement des instructions du MIPS, des commandes du shell, etc. Il est

4. Parfois appelée *big bang*, allez savoir pourquoi...
5. Vos enseignants sont parfois très joueurs !

fondamental de **factoriser** le code commun, à tous les niveaux de votre simulateur. Par exemple, les paragraphes suivants s'intéressent à la factorisation des instructions.

7.3.3.1 Instructions

Toutes les instructions ont bien sûr un champs `opcode` qui sera utilisé pour identifier une instruction à partir de 4 octets de la section `.text`, et un mnémonique pour l'afficher lors du désassemblage. Il n'y a également que trois formats de codage `R`, `I` et `J`. Mais vous pouvez aller beaucoup plus loin !

7.3.3.2 Instructions : récupération des opérandes

Un niveau très important se situe au niveau des modes d'adressage. Par exemple :

- les instructions `ADD`, `AND` ou `SLT` ont trois opérandes situés dans des registres (`rd`, `rs`, `rt`) ;
- d'autres instructions de type `R` prennent deux registres (`rd`, `rt`) et une valeur `sa` (*shift-amont*), ou bien deux registres seulement (`rs`, `rt`), etc. ;
- presque toutes les instructions de lecture/écriture (`LW`, `SW`, `LB`, `SB`...) ont deux opérandes : un registre (`rt`) et une adresse calculée à partir d'un `offset(base)` ;
- d'autres instructions de type `I` prennent deux registres (`rt`, `rs`) et un `immediate`, ou encore deux registres (`rs`, `rt`) puis un `offset` ;
- et ainsi de suite...

Étudiez soigneusement les différentes instructions du projet, pour identifier les principaux « modèles » qui se présentent pour récupérer les opérandes d'une instruction, et ainsi limiter votre simulateur au traitement de ces uniques cas. [6]

7.3.3.3 Instructions : opérations

Une fois les opérandes récupérés, un autre niveau de factorisation porte sur les opérations à réaliser. Fondamentalement, `ADD`, `ADDU` ou `ADDI` réalisent toutes l'addition de deux valeurs et l'ajout du résultat dans une destination. Que dire de `SUB` ? À vous de voir...

Là encore, cherchez à factoriser au maximum les opérations à réaliser dans votre simulateur.

7.3.3.4 Et en C ?

Plusieurs « techniques » de C peuvent être utilisées pour prendre en compte la factorisation : `union` dans des structures, extension de structures et cast, pointeurs de fonctions... Par ailleurs, l'utilisation d'énumération permet de rendre un code très lisible, par rapport à des constantes numériques.

À vous de vous renseigner pour tirer parti au mieux des possibilités du langage !

6. À quelques très rares exceptions près, les instructions spécifiées section 4 couvrent en réalité tous les cas possibles dans l'ensemble du jeu d'instruction d'un mips 32 bits—hors calcul flottant. Il serait donc très simple d'étendre la couverture d'instructions gérée par votre simulateur.

Bibliographie

[1] MIPS Technologies, *MIPS32 Architecture For Programmers Volume II : The MIPS32 Instruction Set*, Rev 2.50, July 1, 2005. http://www.mips.com 4, 4.4

[2] Bradley Kjell, *Programmed Introduction to MIPS Assembly langage*. https//chortle.ccsu.edu/AssemblyTutorial/index.html 2.1

[3] *Emulators - LinuxMIPS*. http://www.linux-mips.org/wiki/Emulators

[4] *Wikipedia : MIPS architecture*. http://en.wikipedia.org/wiki/MIPS_architecture

[5] Charles Lin, *Computer Organization*, University of Maryland. http://www.cs.umd.edu/class/spring2003/cmsc311/Notes

[6] Tools Interface Standard (TIS), *Executable and Linkable Format (ELF)*, Portable Formats Specification, Ver 1.1. http://www.skyfree.org/linux/references/references.html 5.4, 7.2.3, 3, C, C.1

[7] *SYSTEM V APPLICATION BINARY INTERFACE, MIPS RISC Processor Supplement*. http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf 5.4.4

[8] Free Software Foundation & TIGCC Team, *The* GNU *Assembler*. http://tigcc.ticalc.org/doc/gnuasm.html 3

Annexe A

Suite d'outils spécifiques au MIPS

Pour assembler des fichiers de tests en langage assembleur et étudier leur contenu, les outils de la suite `binutils` ont été cross-compilés pour une architecture MIPS. [1]

Ces programmes sont disponibles sur les machines de l'Ensimag. Pour les utiliser, il faut ajouter dans votre .bashrc la ligne `export PATH=$PATH:/opt/mips-tools-cep/bin`.

A.1 Génération de fichiers ELF

A.1.1 L'assembleur `mips-elf-as`

`mips-elf-as test.s -o test.o` permet de créer le fichier objet binaire `test.o` au format ELF à partir du fichier texte `test.s` contenant le source assembleur.

A.1.2 L'éditeur de liens `mips-elf-ld`

Il permet de réaliser les deux phases décrites figure 5.1, page 32 : la fusion de fichiers objets (avec relocation des symboles externes) puis l'implantation des trois sections `.text`, `.data` et `.bss` à des adresses mémoires déterminées.

Ainsi `mips-elf-ld test.o -o test` génère un *programme exécutable*. Les adresses des sections dans `test` sont déterminées, de même que le point d'entrée, et il n'y a plus d'informations de relocation. [2] Dans le cas multi-fichiers, `mips-elf-ld toto1.o toto2.o toto3.o -o totoexe` fusionne les trois fichiers objets puis crée un exécutable.

Avec l'option `-r`, il est possible de ne réaliser que la phase de "fusion" de plusieurs fichiers objet, mais sans la phase d'implantation. Par exemple, `mips-elf-ld -r toto1.o toto2.o toto3.o -o totos.o` génère un unique fichier objet mais qui n'est pas un exécutable. Seule la relocation des symboles externes à un fichier et définis dans un des deux autres a été faite, et certaines autres informations de relocation remises à jour (les adresses relatives ont pu changer du fait

1. Tiens, il exite aussi de nombreux. . .simulateurs de MIPS ! ! Déjà vu ? Comme quoi, ce projet n'est pas là que pour [rigoler | vous embêter] ! :)
2. Sauf en cas de liaison dynamique de librairies, mais ceci sort du cadre de ce projet.

de la fusion des sections). `totos.s` ne contient plus de symbole externe, mais reste un fichier relogeable. Les adresses de ses sections sont toujours indéterminées (donc 0x0) et il peut encore contenir des tables de relocation.

Les autres options de `mips-elf-ld` les plus utiles dans ce projet seront :

- `-Ttext` pour spécifier l'adresse d'implantation mémoire de la section `.text`, par exemple `-Ttext=0x10000`. Si elle n'est pas spécifiée, l'adresse par défaut est 0x400000.
- `-Tdata` (respectivement `-Tbss`) pour spécifier l'adresse d'implantation de la section `.data` (respectivement `.bss`). Si elle n'est pas spécifiée, cette adresse est positionnée juste après la fin de la section d'instructions [3] (respectivement `.data`).
- `-e entry` spécifie le symbole définissant le point d'entrée du programme exécutable. Si cette option n'est pas présente, l'éditeur de lien défini par défaut le point d'entrée comme l'adresse du symbole global `_start`, s'il existe. S'il n'existe pas, le point d'entrée sera placé au début de la section `.text` (et vous aurez gagné un *warning*).

L'assembleur comme l'éditeur de liens vous serviront à créer les fichiers de test de votre simulateur.

Pour aller plus loin, vous pourriez aussi utiliser directement un compilateur cross-compilé pour le MIPS (`mips-elf-gcc`) pour générer des binaires directement à partir de programmes C. Nettement plus compliqué néanmoins à utiliser dans votre simulateur (jeu d'instructions complet, librairies, ...).

A.2 Etude du contenu de fichiers ELF

Plusieurs outils permettent d'afficher les informations contenues dans un fichier ELF :

- `mips-elf-objdump test.o` permet d'étudier le contenu d'un fichier binaire au format ELF. Plusieurs options sont possibles : `-d` pour désassembler la section `.text`, `-r` pour voir la table de relocation, etc.
- `mips-elf-readelf test.o` permet également de lire les infos sur toutes les sections d'un fichier ELF. Présentation et informations parfois différentes par rapport à `mips-elf-objdump` (souvent plus clair, mais pas de désassemblage).
- `od -t xC -address-radix=x test.o` pour ceux qui aiment lire un fichier ELF binaire directement en hexadécimal...(rare, mais ponctuellement utile)
- `hexdump -C test.o` est également très intéressant, avec un affichage dual binaire et `ascii`

Ces outils vous serons très utiles, d'abord pour mieux comprendre la structure d'un fichier ELF puis pour valider le désassemblage et la relocation de votre simulateur !

3. Donc sans respecter les contraintes d'alignement spécifiées section6.1.1 ! En fait la phase de chargement d'un exécutable est plus complexe (adresses virtuelles, pagination...) et sort du cadre de ce projet. Voir par exemple le chapitre "Program loading and dynamic linking" du document [6] pour plus d'informations.

Annexe B

Exemple complet avec relocation

B.1 Programme `exempleElf.s`

Soit le programme assembleur complet suivant, qui reprend tous les types de relocation abordés dans les exemples du chapitre 5.

```
# Pour compiler le fichier objet:
#     mips-as exempleElf.s -o exempleElf.o
# Pour avoir un exe:
#     mips-ld exempleElf.o -o exempleElf

.set noreorder
.global _start      # Exportation du point d'entree (symbole global)

.text
_start:             # Point d'entree
ADDI $3, $0, 12345  # Met la valeur 12345 dans le registre 3
JAL write           # Appel a la procedure "write" (met PC a la bonne adresse)
NOP                 # branch delay slot
B end               # On revient ici apres le "write"; branchement sur la fin
NOP

write:
SW $3, Z            # Ecrit le contenu du registre 3 a l'adresse Z
JR $31              # fin de la procedure "write", retour a l'appelant
NOP

end:                # The End!

.data
X: .byte 0xAB       # Declaration d'un octet initialise avec la valeur 0xAB
Y: .word Z          # Mot de 32 bits, en fait une reference a l'adresse Z
Z: .word 0x0        # Declaration d'un mot de 32 bits initialise a 0

.bss
.skip 9             # Reservation de 9 octets, non initialises
```

B.2 Désassemblage, tables de relocation et de symboles

Le fichier peut être désassemblé à l'aide de la commande `mips-elf-objdump exempleElf.o -D` :

```
exempleElf.o:     file format elf32-bigmips

Disassembly of section .text:

00000000 <_start>:
   0:   20033039    addi    v1,zero,12345
   4:   0c000005    jal 14 <write>
   8:   00000000    nop
   c:   10000005    b   24 <end>
  10:   00000000    nop

00000014 <write>:
  14:   3c010000    lui at,0x0
  18:   ac230008    sw  v1,8(at)
  1c:   03e00008    jr  ra
  20:   00000000    nop
Disassembly of section .data:

00000000 <X>:
   0:   ab000000    swl zero,0(t8)

00000004 <Y>:
   4:   00000008    jr  zero

00000008 <Z>:
   8:   00000000    nop
Disassembly of section .bss:

00000000 <.bss>:
    ...
Disassembly of section .reginfo:

00000000 <.reginfo>:
   0:   9000000a    lbu zero,10(zero)
    ...
```

La commande `mips-elf-readelf -r exempleElf.o` fournit les informations de relocation :

```
Relocation section '.rel.text' at offset 0x370 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000104 R_MIPS_26         00000000   .text
00000014  00000205 R_MIPS_HI16       00000000   .data
00000018  00000206 R_MIPS_LO16       00000000   .data

Relocation section '.rel.data' at offset 0x388 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000202 R_MIPS_32         00000000   .data
```

Et `mips-elf-readelf -s exempleElf.o` la table des symboles :

```
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: 00000000     0 OBJECT  GLOBAL DEFAULT    1 _start
     7: 00000014     0 NOTYPE  LOCAL  DEFAULT    1 write
     8: 00000024     0 NOTYPE  LOCAL  DEFAULT    1 end
     9: 00000008     0 NOTYPE  LOCAL  DEFAULT    3 Z
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 X
    11: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 Y
```

Plusieurs choses à remarquer :

1. La pseudo-instruction `SW $3, Z` a en fait été remplacée par les deux instructions

```
LUI $1, 0x0
SW $3, 8($1)
```

2. Toutes les adresses (instructions, étiquettes, données) sont toujours définies par rapport au début de la section à laquelle il appartient. Les adresses absolues ne seront fixées qu'à l'édition de liens ou au chargement du fichier dans votre simulateur.

3. Toutes les entrées nécessitant une relocation sont quand même codées ! La partie de ce code qui sera modifiée au chargement (les 32, 26 ou 16 bits de poids faible) contient le `addend` nécessaire au calcul de relocation (section 5.4.3).

4. D'autres encore mais à vous de fouiller un peu... C'est le meilleur moyen de comprendre !

B.3 Calculs de relocation

Détaillons maintenant le fonctionnement de la relocation qui est effectuée au moment du chargement. Pour cet exemple, les adresses fournies par l'éditeur de liens sont les suivantes :
— `0x0AA1B008` pour la section `.text`
— `0x26004` pour la section `.data`

B.3.1 `JAL write`

B.3.1.0.1 Relocation

Les variables de calcul introduites section 5.4.4 ont les valeurs suivantes :

Variable	Valeur	Signification
`offset`	`0x4`	Décalage de l'entrée par rapport au début de sa section
`Type`	`R_MIPS_26`	Mode de relocation
`Value`	`.text`	Section contenant l'étiquette `write`
`P`	`0x0AA1B00C`	Adresse de l'entrée : `0xAA1B008` + `offset`
`A`	`0x00000005`	`addend` : `0x0C000005` & `0x3FFFFFF` (26 bits de poids faible)
`S`	`0x0AA1B008`	Adresse de la section contenant le symbole

La valeur du champ `instr_index` du `JAL` est donc :

```
(P & 0xF0000000)                          = 0x0
(A ≪ 2) | (P & 0xF0000000)                = 0x14
[(A ≪ 2) | (P & 0xF0000000)] + S          = 0xAA1B01C
([(A ≪ 2) | (P & 0xF0000000)] + S) ≫ 2    = 0x2A86C07
```

Finalement, **le code écrit en mémoire à l'adresse `0x0AA1B00C` est `0x0EA86C07`**, en lieu et place du précédent (`0x0C000005`, rappel : les 6 bits de poids fort sont conservés).

B.3.1.0.2 Exécution

Pour vérifier ce calcul, il suffit de simuler l'exécution de l'instruction `JAL` située à l'adresse `0x0AA1B00C` :

D'après les spécifications de l'instruction `JAL`, l'adresse destination du saut est obtenue en décalant de 2 bits vers la gauche l'offset de 26 bits, ce qui donne une adresse sur 28 bits. L'adresse finale est obtenue en complétant cette adresse avec les 4 bits de poids fort de l'adresse contenue dans `PC`, c'est à dire avec les 4 bits de poids fort de l'adresse de l'instruction suivante (située en `0x0AA1B010`).

```
0x02A86C07 ≪ 2     = 0x0AA1B01C
PC & 0xF0000000    = 0x00000000
adresse de saut    = 0x0AA1B01C
```

Est-ce bien l'instruction référencée par l'étiquette `write` ? Oui, puisqu'elle se trouve `0x14` octets après le début de la section `.text`, donc bien `0x0AA1B008` + `0x14` = `0x0AA1B01C` ![1]

B.3.2 `SW $3, Z`

La pseudo-instruction `SW` a été décomposée en deux instructions successives `LUI` et `SW` :

```
0x00000014  3C010000     LUI $1, 0x0
0x00000018  AC230008     SW $3, 8($1)
```

chacune associée à une entrée de relocation. Les adresses de chargement des sections sont les mêmes que précédemment (`0x0AA1B008` et `0x26004`).

1. *Eh eh eh eh, c'est pas beautiful ça, hein ?*

B.3.2.0.1 Chargement

Les variables de calcul sont les suivantes :

Variable	Valeur pour LUI	Valeur pour SW
offset	0x14	0x18
Type	R_MIPS_HI16	R_MIPS_LO16
Value	.data	.data
P	0x0AA1B01C	0x0AA1B020
A	0x0000	0x0008
AHL	0x00000008	0x00000008
S	0x00026004	0x00026004

Le résultat est donc le suivant :

Entrée	Calcul	Adresse mémoire	Codage fichier	Codage mémoire
LUI	((AHL + S) - (short)(AHL + S)) $\gg$ 16	0x0AA1B01C	3C010000	3C010002
SW	(AHL + S) & 0x0000FFFF	0x0AA1B020	AC230008	AC23600C

Après relocation, le code correspondant est :

```
0x0AA1B01C  3C010002     LUI $1, 0x2
0x0AA1B020  AC23600C     SW $3, 0x600C($1)
```

B.3.2.0.2 Exécution

À nouveau une petite vérification :

— l'exécution de `LUI $1, 0x2` affecte la valeur `0x20000` au registre `$1` [2] ;
— l'exécution de `SW $3, 0x600C($1)` écrit la valeur contenue dans `$3` à l'adresse mémoire `0x20000 + 0x600C` = `0x2600C`, qui est bien l'adresse de `Z`.

B.3.3 Étiquettes `Y` et `Z`

À vous de jouer et de vérifier, c'est très simple si vous avez compris ce qui précède.

B.3.4 Et moi dans tout ça ?

Lorsque c'est nécessaire, votre travail consistera à récupérer les informations de relocation et à reloger correctement le code binaire lu en fonction des adresses de sections.

Le plus dur/long est de comprendre, ensuite c'est très simple !

2. Au passage, c'est une des raisons pour laquelle le registre `$1` ne devrait pas être utilisé dans les programmes mais réservé à l'assembleur. Si un programme l'utilise mais que l'assembleur s'en sert aussi pour stocker des données temporaires, l'exécution a très peu de chances de se dérouler normalement...

B.4 Cas d'un fichier exécutable

Créez un fichier exécutable : `mips-elf-ld exempleElf.o -o exempleElf`.

Étudiez ensuite son contenu par rapport à celui du fichier objet (désassemblage, symboles, tables de relocation, entête de fichiers, etc.).

Annexe C

Format ELF et module `elf_reader`

Un fichier ELF (*Executable and Linkable Format*) est un fichier binaire contenant toutes les informations relatives à un fichier objet ou un programme assemblé : son codage binaire, la liste des symboles définis, l'ensemble des données de relocation, et d'autres informations encore. Toutes ces informations seront utilisées pour l'édition de liens (fusion entre plusieurs fichiers objets, création d'un fichier exécutable), puis par le chargeur du système d'exploitation avant l'exécution d'un programme. Dans ce projet, votre simulateur devra lire un fichier au format ELF pour charger un programme et le reloger en mémoire.

Pour avoir un exemple de fichier, c'est très facile : créez un petit programme et assemblez-le avec l'assembleur `mips-elf-as` fourni, `as` cross-compilé pour une cible MIPS. Ensuite, étudiez son contenu à l'aide des commandes `mips-elf-objdump` ou `mips-elf-readelf` ! [1]

Le format ELF est défini par le comité *Tool Interface Standards* (TIS) [6]. Les constantes et structures C correspondants aux différents types de données (entête, section, symbole...) sont définies dans le fichier `elf.h` de votre système (généralement situé dans `/usr/include`).

L'objectif de cette annexe est de décrire les principales notions sur la structure d'un fichier ELF puis le module de haut-niveau `elf_reader` distribué par les enseignants. La première section sera surtout utile pour *comprendre* la structure ELF et *réécrire* vous-même votre propre module `elf_reader`. La seconde (en fait les spécifications détaillées des fonctions et les types de données du format ELF) est elle suffisante pour *utiliser* ce module dans votre simulateur.

C.1 Tables

Un fichier ELF est un fichier binaire structuré sous forme de « tables » ou « sections ». Le contenu de ces tables est accessible via les programmes `mips-elf-objdump` ou `mips-elf-readelf` (voir section A), avec les options adaptées : `-s` pour afficher la tables des symboles, `-r` pour les tables de relocation, etc. Des exemples sont disponibles dans les annexes.

1. Vous pouvez aussi faire ceci avec le code de votre propre simulateur, quelque que soit l'architecture sur laquelle vous exécutez (Pentium, Athlon...). Si `simips` est le nom de l'exécutable, tapez `objdump simips -d`, et vous comprendrez pourquoi vous avez écrit votre projet en `C` et pas en langage assembleur...

Entête Le fichier commence toujours par un entête donnant les caractéristiques générales du fichier (processeur cible, version, etc.) puis les caractéristiques de la *table des entêtes de sections* : sa taille (nombre de sections) et sa position (décalage, en nombre d'octets par rapport au début du fichier).

Table des entêtes de sections C'est en fait un tableau d'entêtes, chacun indiquant notamment :
- le nom de la section (en fait un index dans la *table des noms de sections*)
- sa taille (en octets) et sa position dans le fichier (décalage, toujours en nombre d'octets par rapport au début).
- son type, lié à la nature des informations de la section (données du programme, symboles, chaînes...)
- l'adresse mémoire de la section, lorsqu'elle est déterminée (cas d'un exécutable), et une contrainte d'alignement de la section (dont l'adresse doit être un multiple).

Un ficher objet ELF contient généralement : une section par zone (`.text`, `.data` ou `.bss`) et sa section de relocation associée (`.rel.text` ou `.rel.data`), une *table des symboles*, une *table des chaînes* et une *table des noms de sections*. Toutes les sections ne sont pas forcément présentes, selon le contenu du code assemblé (par exemple il peut ne pas y avoir de zone `.data`, ni de symboles à reloger, aucune donnée n'est associée à `.bss`, etc.).

La première section (indice 0) est toujours vide, c'est une sentinelle.

Table des noms de sections C'est une table de chaînes de caractères contenant les noms des sections : ".text", ".data", ".symtab", etc. La première chaîne est toujours la chaîne vide.

Table des chaînes Encore une table de chaînes de caractères, contenant cette fois les noms des symboles de sections et de tous les symboles définis ou utilisés dans le fichier source.

Table des symboles C'est la table des symboles du code, caractérisés notamment par :
- son nom, en fait son index dans la table des chaînes.
- l'index (dans la table des entêtes de sections) de la zone dans laquelle il est défini. Pour un symbole externe, cet index vaut 0 (la sentinelle dans la table des entêtes de sections).
- sa valeur, i.e. le déplacement en nb d'octets par rapport au début de sa zone de définition (0 pour les symboles externes).

Les sections de données `.text` et `.data`, qui contiennent le code assemblé, binaire, du programme (il faut bien qu'il soit quelque part !).

Les sections de relocation La section `.rel.text` contient par exemple les informations de relocation des entrées situées dans la section `.text`. Le contenu d'une entrée de relocation a en fait déjà été expliqué section 5.4.2 : la position du code à reloger (par rapport au début de la zone), le mode de relocation, et la section dans laquelle chercher le symbole référencé (indice de zone ou 0 pour un symbole externe).

Autres sections D'autres sections peuvent encore être présentes, notamment le *Program Header*, section qui contient les informations de pagination d'un fichier exécutable, ou des sections d'informations spécifiques à une architecture donnée. Elles ne seront pas traitées dans ce projet.

Cette description a juste pour objectif de vous présenter rapidement le contenu et la structure d'un fichier ELF, en partie shématisée figure C.1.

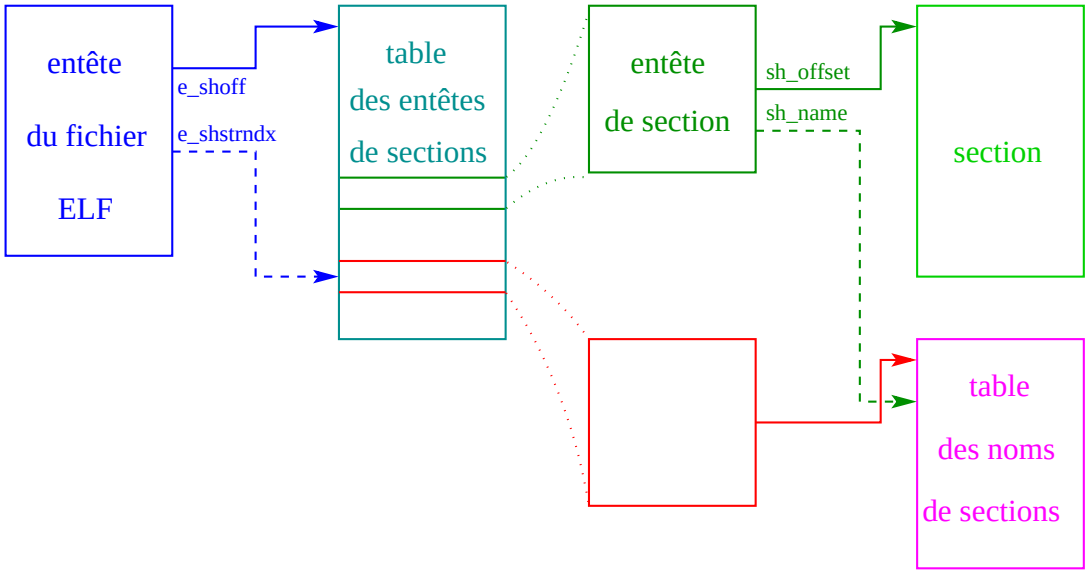


FIGURE C.1 – Exemple de relations entre les “tables” d’un fichier ELF (qui sont en fait stockées séquentiellement dans le fichier).

Une description complète est surtout disponible dans le document du TIS [6] (ne regardez que la première partie, sur les fichiers objet). **Il s’agit DU document de référence, clair et complet, indispensable pour réécrire vous-même votre chargeur de fichiers ELF !**

C.2 Module `elf_reader` et structure des principales tables

Cette section décrit de manière détaillée :
- certaines des structures de données du format ELF, utilisées par les principales tables. Seules celles directement utiles pour votre projet sont décrites.
- le principe de l’utilisation du module `elf_reader` pour lire ces tables.

C.2.1 Utilisation générale

Le principe général d’utilisation d’ `elf_reader` est
- de débuter la lecture d’un fichier à l’aide de la fonction `read_elf`, qui retourne un descripteur (de format privé) contenant toutes les informations du fichier.
- de lire les informations dans ce descripteur, à l’aide des fonctions `get_XXX`
- de terminer avec `close_elf`.

Les données récupérées en argument des fonctions `get_XXX` seront libérées lors de l’appel à `close_elf`.

C.2.2 Structures de données

Le format ELF définit (via le fichier `elf.h` installé sur les systèmes d'exploitation) des structures de données pour tous les types d'informations manipulés : `Elf_Scn` (section), `Elf_Data` (données), `Elf32_Sym` (symbole), `Elf32_Rel` (entrée de relocation), ... De nombreuses constantes sont aussi déclarées, par exemple pour représenter différents types de fichier : `ET_REL` (fichier objet relogeable), `ET_EXE` (exécutable), etc.

Le module fourni déclare une structure (privée) `struct elf_descr`, un descripteur retourné lors de l'appel à `read_elf` et qui est passé aux fonctions `get_XXX` pour accéder aux informations.

C.2.3 Lecture des sections `.text`, `.data` et `.bss`

La fonction `get_text_section` permet de récupérer le codage binaire des instructions à partir d'un descripteur ELF.

```
bool get_text_section(const struct elf_descr *elf, uint8_t **data,
                      size_t *size, uint32_t *addr, uint8_t *align);
```

En entrée, `elf` est un descripteur préalablement crée via l'appel à la fonction `read_elf`. Si ce descripteur n'est pas valide, la fonction retourne `false`

En sortie (les adresses des variables où stocker le résultat sont passées) :

- `size` est le nombre d'octets de la section et `data` le tableau de ces octets. A noter que le tableau est allouée dynamiquement au début de la lecture, et sera libéré lors de l'appel à `close_elf`.
  Avec l'exemple de l'annexe B, le tableau contiendra les 36 octets `[200330390C00000500 ... AC23000803E0000800000000]`
- `addr` est l'adresse d'implantation en mémoire de la section, qui est nulle dans le cas d'un fichier relogeable et absolue pour un programme exécutable.
- `align` est la contrainte d'alignement de la section. L'adresse absolue de la section (que vous devrez fixer vous-même dans le cas relogeable) doit nécessairement être un multiple de cette valeur. Pour la section `.text`, `align` vaut nécessairement 4. Pour les sections de données, elle vaut 1 si seule la directive `.byte` est utilisée, 2 (respectivement 4) dès qu'une directive `.half` (respectivement `.word`) est utilisée.

Le fonctionnement est identique pour la section `.data`. La section `.bss` ne contenant aucune donnée (pas de tableau de "0"), seules sont renseignées sa taille, son adresse et sa contrainte d'alignement.

C.2.4 Lecture de la table des chaînes

Cette table contient les noms de tous les symboles du fichier (voir section 5.4.1), d'abord les symboles de section puis ceux de programme.

La fonction `get_string_table` permet de lire cette table et sa taille :

```
extern bool get_string_table(const struct elf_descr *elf,
                             char **strtab, size_t *size);
```

La table des chaînes ELF est formatée comme un unique tableau de caractères contenant une concaténation de toutes les chaînes, séparées par des caractères nuls ('\0'). Par exemple, si un fichier ELF ne contient qu'une section `.text` avec deux symboles `"toto"` et `"X"`, la table des chaînes sera constituée de :

\0	.	t	e	x	t	\0	t	o	t	o	\0	X	\0

- Le premier caractère est une sentinelle, toujours égale à '\0'.
- La chaîne `".text"` (un nom de section est aussi un symbole !) aura l'indice 1, la chaîne `"toto"` l'indice 7 et la chaîne `"X"` l'indice 12.
- La taille totale de la table est de 14 octets.

C.2.5 Lecture de la table des symboles

La table des symboles et sa taille (nombre de symboles) sont accessibles via la fonction

```
bool get_symbol_table(const struct elf_descr *elf,
                      Elf32_Sym **symtab, size_t *size);
```

La table des symboles d'un fichier ELF n'est jamais vide, elle contient toujours au moins 4 symboles, d'indice 0 à 3 : une sentinelle en 0 puis les symboles de noms de zones (`.text`, `.data` et `.bss`). Ensuite, la table peut contenir les symboles de programme locaux (ou externes) définis (ou utilisés) dans le programme lu.

La fonction `get_symbol_table` permet d'extraire le tableau des symboles contenus dans le fichier lu. Le type de données `Elf32_Sym` représentant un symbole est défini dans `elf.h` :

```
/* Symbol table entry.  */
typedef struct
{
    Elf32_Word    st_name;      /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;     /* Symbol value */
    Elf32_Word    st_size;      /* Symbol size */
    unsigned char st_info;      /* Symbol type and binding */
    unsigned char st_other;     /* Symbol visibility */
    Elf32_Section st_shndx;     /* Section index */
} Elf32_Sym;
```

Plusieurs macros `C` sont associées, pour coder/décoderle 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 */
```

Les différents champ de la structure `Elf32_Sym` sont les suivants :

- `st_name` : index du symbole dans la table des chaînes.
- `st_value` : pour un symbole défini localement, il s'agit du déplacement (en octets) par rapport au début de la zone de définition. Il vaut 0 pour un symbole non défini localement. Pour un symbole de section, il vaut l'adresse absolue de la section (cas d'un exécutable) ou 0 (cas d'un fichier relogeable).
- `st_size` et `st_other` : non utilisés dans le projet, ces champs seront toujours nuls `st_size` sert à associer une taille de donnée au symbole).

— st_info : cette valeur codée sur un octet sert en fait à coder 2 champs : le champ “bind” et le champ “type”, décrits ci-dessous.
— st_shndx : indique l’index (dans la table des en-têtes de sections) de la section où le symbole est défini. Si le symbole est de type STT_SECTION, cet index est directement l’index de la section. Si le symbole n’est défini dans aucune section (symbole externe), cet index vaut 0.

Le champ “bind” indique la portée du symbole. Dans ce projet, seuls les 2 cas suivants seront considérés :

— 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 cas suivants pour le champ “type” :

— STT_SECTION (constante 3) indiquant que le symbole désigne en fait une section.
— STT_OBJECT (constante 1) indiquant que le symbole est exporté (directive .global, voir section 3.5.4).
— STT_NOTYPE (constante 0), dans les autres cas.

Le champ st_info est généré à l’aide de la macro définie précédemment, par exemple pour un symbole local non exporté :

```
sym_local->st_info = ELF32_ST_INFO (STB_LOCAL, STT_NOTYPE);
```

Pour un exemple, on peut reprendre la table des symboles de l’annexe B.2 :

```
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: 00000000     0 OBJECT  GLOBAL DEFAULT    1 _start
     7: 00000014     0 NOTYPE  LOCAL  DEFAULT    1 write
     8: 00000024     0 NOTYPE  LOCAL  DEFAULT    1 end
     9: 00000008     0 NOTYPE  LOCAL  DEFAULT    3 Z
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 X
    11: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 Y
```

C.2.6 Entrées de relocation

Les deux fonctions get_rel_text_section et get_rel_text_section permettent d’obtenir un tableau des entrées de relocation de chaque zone du fichier, et sa taille.

```
bool get_rel_text_section(const struct elf_descr *elf,
                          Elf32_Rel **data, size_t *size);
```

Chaque entrée de relocation est de type Elf32_Rel, défini comme :

```
typedef struct
{
    Elf32_Addr r_offset;        /* Address */
    Elf32_Word r_info;          /* Relocation type and symbol index */
} Elf32_Rel;
```

Les champs de cette table se réfèrent directement aux explications de la section 5.4.2 :

- `r_offset` : contient le décalage en octets de l'entrée de relocation par rapport au début de la zone considérée (`.text` pour une entrée de la table `.rel.text`, etc.)
- `r_info` : champ composé des deux informations "type" et "sym"
  - "type" : le mode de relocation, codé sur 1 octets
  - "sym" : codé sur 3 octets, il permet de localiser le symbole à reloger. Si le symbole est défini localement et non exporté, "sym" est l'index dans la table des symboles *de la section contenant le symbole* (1 pour un symbole en zone `.text`, 2 en zone `.data`, 3 en zone `.bss`). Si le symbole est externe ou défini localement et exporté, "sym" est directement l'index du symbole dans la table des symboles.
    Cette différence de traitement entre les symboles locaux et les symboles globaux permet en fait d'omettre tous les symboles locaux (excepté les symboles de section) de la table des symboles et de la table des chaînes : on réduit ainsi la taille du fichier généré.[2] Toutefois, il peut être commode de garder les symboles locaux lorsqu'on utilise un débogueur.

Les macros de codage/décodage du champ `r_info` sont :

```
/* Macros for accessing the fields of r_info. */
#define ELF32_R_SYM(info)       ((info) >> 8)
#define ELF32_R_TYPE(info)      ((unsigned char)(info))

/* Macro for constructing r_info from field values. */
#define ELF32_R_INFO(sym, type) (((sym) << 8) + (unsigned char)(type))
```

Les modes de relocation sont définis, dans `elf.h` par des constantes entières. Ceux utilisés dans ce projet sont :

```
#define R_MIPS_32       2
#define R_MIPS_26       4
#define R_MIPS_HI16     5
#define R_MIPS_LO16     6
```

Pour un exemple, on peut reprendre la table de relocation de l'annexe B.2 :

```
Relocation section '.rel.text' at offset 0x370 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000104 R_MIPS_26         00000000   .text
00000014  00000205 R_MIPS_HI16       00000000   .data
00000018  00000206 R_MIPS_LO16       00000000   .data

Relocation section '.rel.data' at offset 0x388 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000202 R_MIPS_32         00000000   .data
```

2. Pour omettre les symboles locaux avec l'assembleur du projet `mips-elf-as`, il faut l'invoquer avec l'option `-n`.

La première entrée de relocation concerne l'instruction située 4 octets après le début de la section `.text` (`JAL write`). Le type de relocation est 4, soit `R_MIPS_26`. La valeur du symbole est 1, qui est ici l'index de la section `.text` dans la table des entêtes de section (c'est la valeur de `Ndx` associée au symbole de section `.text`).

C.2.7 Type de fichier, point d'entrée

Pour simplifier le module, toutes les tables du fichier ELF ne sont pas accessibles (par exemple l'entête de fichier ou la table des entêtes de sections). Pour accéder à certaines données nécessaires au projet, des fonctions supplémentaires ont été ajoutées qui retournent directement ces informations :

```
uint16_t get_elf_type(const struct elf_descr *elf);
```

permet de lire de type de fichier : `ET_EXEC` s'il est *exécutable* (les adresses des sections sont absolues) ou `ET_REL` s'il est *relogeable* (les adresses n'ont pas été fixées, il peut y avoir des tables de relocation).

```
uint32_t get_entry_point(const struct elf_descr *elf);
```

lit le point d'entrée du programme (voir 6.1.4). Cette adresse est 0 pour un fichier relogeable, absolue pour un exécutable.