Sorbonne-Université Sciences

Architecture Logicielle et Matérielle des Ordinateurs

2019

Architecture Externe du Mips32

Langage d'assemblage du Mips32

licence informatique LU3IN004

Alain Greiner

Quentin Meunier

Franck Wajsbürt

Pirouz Bazargan

Emmanuelle Encrenaz

Processeur MIPS32

Architecture externe

Version 2.6

Septembre 2019

Alain Greiner

A) INTRODUCTION

Ce document présente une version légèrement simplifiée de l'architecture externe du processeur **MIPS32** (pour des raisons de simplicité, tous les mécanismes matériels de gestion de la mémoire virtuelle ont été délibérément supprimés).
L'architecture externe représente ce que doit connaître un programmeur souhaitant programmer en assembleur, ou la personne souhaitant écrire un compilateur pour ce processeur:

- Les registres visibles du logiciel.
- L'adressage de la mémoire.
- Le jeu d'instructions.
- Les mécanismes de traitement des interruptions, exceptions et trappes.

Le processeur **MIPS32** est un processeur 32 bits industriel conçu dans les années 80. Son jeu d'instructions est de type RISC. Il existe plusieurs réalisations industrielles de cette architecture (SIEMENS, NEC, LSI LOGIC, SILICON GRAPHICS, etc...)

Cette architecture est suffisamment simple pour présenter les principes de base de l'architecture des processeurs, et suffisamment puissante pour supporter un système d'exploitation multi-tâches tel qu'UNIX, puisqu'il supporte deux modes de fonctionnement : dans le mode **utilisateur,** certaines zones de la mémoire et certains registres du processeur réservés au système d'exploitation sont protégés et donc inaccessibles; dans le mode **superviseur,** toutes les ressources sont accessibles.

L'architecture interne dépend des choix de réalisation matérielle. Plusieurs implantations matérielles de cette architecture ont été réalisées à l'Université Pierre et Marie Curie dans un but d'enseignement et de recherche : une version microprogrammée, simple mais peu performante (4 cycles par instruction), une version pipe-line plus performante (une instruction par cycle), mais plus complexe, une version superscalaire, encore plus performante (2 instructions par cycle) et beaucoup plus complexe.

La spécification du langage d'assemblage, des conventions d'utilisation des registres, ainsi que des conventions d'utilisation de la pile fait l'objet d'un document séparé.

B) REGISTRES VISIBLES DU LOGICIEL

Tous les registres visibles du logiciel, c'est à dire ceux dont la valeur peut être lue ou modifiée par les instructions, sont des registres 32 bits.

Afin de mettre en oeuvre les mécanismes de protection nécessaires pour un système d'exploitation multi-tâches, le processeur possède deux modes de fonctionnement : utilisateur/superviseur. Ces deux modes de fonctionnement imposent d'avoir deux catégories de registres.

1) Registres non protégés

Le processeur possède 35 registres manipulés par les instructions standard (c'est à dire les instructions qui peuvent s'exécuter aussi bien en mode utilisateur qu'en mode superviseur).

• **Ri** ($0 \leq i \leq 31$) 32 registres généraux
Ces registres sont directement adressés par les instructions, et permettent de stocker des résultats de calculs intermédiaires.
Le registre **R0** est un registre particulier:
- la lecture fournit la valeur constante "0x00000000"
- l'écriture ne modifie pas son contenu.
Le registre **R31** est utilisé par les instructions d'appel de procédures (instructions **BGEZAL**, **BLTZAL**, **JAL** et **JALR**) pour sauvegarder l'adresse de retour.

• **PC** Registre compteur de programme (Program Counter)
Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.

• **HI et LO**
Registres pour la multiplication ou la division
Ces deux registres 32 bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division, qui est un mot de 64 bits.

Contrairement à d'autres processeurs plus anciens, le processeur MIP32 ne possède pas de registres particuliers pour stocker les "codes conditions". Des instructions de comparaison permettent de calculer un booléen qui est stocké dans l'un quelconque des registres généraux. La valeur de ce booléen peut ultérieurement être testée par les instructions de branchement conditionnel.

2) Registres protégés

L'architecture MIPS32 définit 32 registres (numérotés de 0 à 31), qui ne sont accessibles, en lecture comme en écriture, que par les instructions privilégiées MTC0 et MFC0 (ces instructions ne peuvent être exécutées qu'en mode superviseur). On dit qu'ils appartiennent au "coprocesseur système" (appélé aussi CP0). En pratique, cette version du processeur MIPS32 en définit 6. Ils sont utilisés par le système d'exploitation pour la gestion des interruptions , des exceptions, et des appels systèmes (voir chapitre E).

• **SR** Registre d'état (Status Register).
Il contient en particulier le bit qui définit le mode : superviseur ou utilisateur, ainsi que les bits de masquage des interruptions.
(Ce registre possède le numéro 12)

• **CR** Registre de cause (Cause Register).
En cas d'interruption ou d'exception, son contenu définit la cause pour laquelle on fait appel au programme de traitement des interruptions et des exceptions.
(Ce registre possède le numéro 13)

• **EPC** Registre d'exception (Exception Program Counter).
Il contient l'adresse de retour (PC + 4) en cas d'interruption. Il contient l'adresse de l'instruction fautive en cas d'exception (PC).
(Ce registre possède le numéro 14)

• **BAR** Registre d'adresse illégale (Bad Address Register).
En cas d'exception de type "adresse illégale", il contient la valeur de l'adresse mal formée.
(Ce registre possède le numéro 8)

• **PROCID**
Registre en lecture seulement contenant le numéro du processeur.
Cet index « cablé » est utilisé par le système d'exploitation pour gérer des architectures multi-cores.
(Ce registre possède le numéro 15)

• **CYCLECOUNT**
Registre en lecture seulement contenant le nombre de cycles exécutés depuis l'initialisation du processeur.
(Ce registre possède le numéro 16)

C) ADRESSAGE MÉMOIRE

1) Adresses octet

Toutes les adresses émises par le processeur sont des adresses octets, ce qui signifie que la mémoire est vue comme un tableau d'octets, qui contient aussi bien les données que les instructions.
Les adresses sont codées sur 32 bits. Les instructions sont codées sur 32 bits. Les échanges de données avec la mémoire se font par mot (4 octets consécutifs), demi-mot (2 octets consecutifs), ou par octet. Pour les transferts de mots et de demi-mots, le processeur respecte la convention "little endian".
L'adresse d'un mot de donnée ou d'une instruction doit être multiple de 4. L'adresse d'un demi-mot doit être multiple de 2. (on dit que les adresses doivent être "alignées"). Le processeur part en exception si une instruction calcule une adresse qui ne respecte pas cette contrainte.

2) Calcul d'adresse

Il existe un seul mode d'adressage, consistant à effectuer la somme entre le contenu d'un registre général **Ri**, défini dans l'instruction, et d'un déplacement qui est une valeur immédiate signée, sur 16 bits, contenue également dans l'instruction:

adresse = Ri + Déplacement

3) Mémoire virtuelle

Pour des raisons de simplicité, cette version du processeur MIPS32 ne possède pas de mémoire virtuelle, c'est à dire que le processeur ne contient aucun mécanisme matériel de traduction des adresses logiques en adresses physiques. Les adresses calculées par le logiciel sont donc transmises au système mémoire sans modifications.
On suppose que la mémoire répond en un cycle. Un signal permet au système mémoire de "geler" le processeur s'il n'est pas capable de répondre en un cycle (ce mécanisme peut être utilisé pour gérer les MISS du ou des caches).
Si une anomalie est détectée au cours du transfert entre le processeur et la mémoire, le système mémoire peut le signaler au moyen d'un signal d'erreur, qui déclenche un départ en exception.

4) Protection mémoire

En l'absence de mémoire virtuelle, l'espace mémoire est simplement découpé en 2 segments identifiés par le bit de poids fort de l'adresse :

adr 31 = 0 ==> segment utilisateur
adr 31 = 1 ==> segment système

Quand le processeur est en mode superviseur, les 2 segments sont accessibles. Quand le processeur est en mode utilisateur, seul le segment utilisateur est accessible. Le processeur part en exception si une instruction essaie d'accéder à la mémoire avec une adresse correspondant au segment système alors que le processeur est en mode utilisateur.

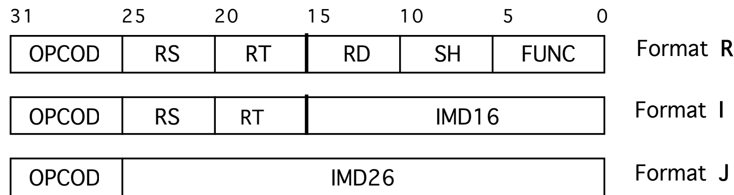
D) JEU D'INSTRUCTIONS

1) Généralités

Le processeur possède 57 instructions qui se répartissent en 4 classes :

- 33 instructions arithmétiques/logiques entre registres
- 12 instructions de branchement
- 7 instructions de lecture/écriture mémoire
- 5 instructions systèmes

Toutes les instructions ont une longueur de 32 bits et possèdent un des trois formats suivants :



Le format **J** n'est utilisé que pour les branchements à longue distance (inconditionnels).
Le format **I** est utilisé par les instructions de lecture/écriture mémoire, par les instructions utilisant un opérande immédiat, ainsi que par les branchements courte distance (conditionnels).
Le format **R** est utilisé par les instructions nécessitant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD.

2) Codage des instructions

Le codage des instructions est principalement défini par les 6 bits du champs **code opération** de l'instruction (**INS 31:26**). Cependant, trois valeurs particulières de ce champ définissent en fait une famille d'instructions : il faut alors analyser d'autres bits de l'instruction pour décoder l'instruction. Ces codes particuliers sont : SPECIAL (valeur "000000"), BCOND (valeur "000001") et COPRO (valeur "010000")

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Ce tableau exprime que l'instruction LHU (par exemple) possède le code opération "100101".

Lorsque le code opération a la valeur SPECIAL ("000000"), il faut analyser les 6 bits de poids faible de l'instruction (**INS 5:0**):

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Lorsque le code opération a la valeur BCOND, il faut analyser les bits 20 et 16 de l'instruction. Lorsque le code opération a la valeur COPRO, il faut analyser les bits 25 et 23 de l'instruction. Les trois instructions de cette famille COPRO sont des instructions privilégiées.

OPCOD = BCOND

INS 20 \ INS 16	0	1
0	BLTZ	BGEZ
1	BLTZAL	BGEZAL

OPCOD = COPRO

INS 25 \ INS 23	0	1
0	MFC0	MTC0
1	RFE	

3) Jeu d'instructions

Le jeu d'instructions est "orienté registres". Cela signifie que les instructions arithmétiques et logiques prennent leurs opérandes dans des registres et rangent le résultat dans un registre. Les seules instructions permettant de lire ou d'écrire des données en mémoire effectuent un simple transfert entre un registre général et la mémoire, sans aucun traitement arithmétique ou logique.

La plupart des instructions arithmétiques et logiques se présentent sous les 2 formes registre-registre et registre-immédiat:

ADD	:	R(rd)	<---	R(rs) op R(rt)		format R
ADDI	:	R(rt)	<---	R(rs) op IMD		format I

L'opérande immédiat 16 bits est signé pour les opérations arithmétiques et non signé pour les opérations logiques.

Le déplacement est de 16 bits pour les instructions de branchement conditionnelles (Bxxx) et de 26 bits pour les instructions de saut inconditionnelles (Jxxx). De plus les instructions JAL, JALR, BGEZAL, et BLTZAL sauvegardent une adresse de retour dans le registre R31. Ces instructions sont utilisées pour les appels de sous-programme.
Toutes les instructions de branchement conditionnel sont relatives au compteur ordinal pour que le code soit translatable. L'adresse de saut est le résultat d'une addition entre la valeur du compteur ordinal et un déplacement signé.

Les instructions MTC0 et MFC0 permettent de transférer le contenu des registres SR, CR, EPC et BAR vers un registre général et inversement. Ces 2 instructions ne peuvent être exécutées qu'en mode superviseur, de même que l'instruction ERET qui permet de restaurer l'état antérieur du registre d'état avant de sortir du gestionnaire d'exceptions.

E) EXCEPTIONS / INTERRUPTIONS / APPELS SYSTEME

Il existe quatre types d'évènements qui peuvent interrompre l'exécution "normale" d'un programme:

- les exceptions
- les interruptions
- les appels système (instructions **SYSCALL** et **BREAK**)
- le signal **RESET**

Dans tous ces cas, le principe général consiste à passer la main à une procédure logicielle spécialisée (appelée Gestionnaire d'Interrutions, Exceptions et Trappes) qui s'exécute en mode superviseur, à qui il faut transmettre les informations minimales lui permettant de traiter le problème.

1) Exceptions

Les exceptions sont des évènements "anormaux", le plus souvent liés à une erreur de programmation, qui empêche l'exécution correcte de l'instruction en cours. La détection d'une exception entraîne l'arrêt immédiat de l'exécution de l'instruction fautive. Ainsi, on assure que l'instruction fautive ne modifie pas la valeur d'un registre visible ou de la mémoire. Les exceptions ne sont évidemment pas masquables. Il y a 7 types d'exception dans cette version du processeur MIPS32 :

- ADEL Adresse illégale en lecture : adresse non alignée ou se trouvant dans le segment système alors que le processeur est en mode utilisateur.
- ADES Adresse illégale en écriture : adresse non alignée ou accès à une donnée dans le segment système alors que le processeur est en mode utilisateur.
- DBE Data bus erreur : le système mémoire signale une erreur en activant le signal **BERR** à la suite d'un accès de donnée.
- IBE Instruction bus erreur : le système mémoire signale une erreur en activant le signal **BERR** à l'occasion d'une lecture instruction.
- OVF Dépassement de capacité : lors de l'exécution d'une instruction arithmétique (**ADD**, **ADDI** ou **SUB**), le résultat ne peut être représenté sur 32 bits.
- RI Codop illégal : le codop ne correspond à aucune instruction connue (il s'agit probablement d'un branchement dans une zone mémoire ne contenant pas du code exécutable.
- CPU Coprocesseur inaccessible : tentative d'exécution d'une instruction privilégiée (**MTC0**, **MFC0**, E**RET**) alors que le processeur est en mode utilisateur.

Le processeur doit alors passer en mode superviseur, et se brancher au **Gestionnaire d'Interruptions, Exceptions et Trappes** (GIET), implanté conventionnellement à l'adresse "0x80000180". Après avoir identifié que la cause est une exception (en examinant le contenu du registre CR), le GIET se branche alors au **gestionnaire**

d'exception. Toutes les exceptions étant fatales il n'est pas nécessaire de sauvegarder une adresse de retour car il n'y a pas de reprise de l'exécution du programme contenant l'instruction fautive. Le processeur doit cependant transmettre au **gestionnaire d'exceptions** l'adresse de l'instruction fautive et indiquer dans le registre de cause le type d'exception détectée. Lorsqu'il détecte une exception, le matériel doit donc:

- sauvegarder l'adresse de l'instruction fautive dans le registre **EPC**
- passer en mode superviseur et masque les interruptions dans **SR**
- sauvegarder éventuellement l'adresse fautive dans **BAR**
- écrire le type de l'exception dans le registre **CR**
- brancher à l'adresse "0x80000180".

2) Interruptions

Les requêtes d'interruption matérielles sont des évènements asynchrones provenant généralement de périphériques externes. Elles peuvent être masquées. Le processeur possède 6 lignes d'interruptions externes qui peuvent être masquées globalement ou individuellement. L'activation d'une de ces lignes est une requête d'interruption. Elles sont écrites dans le registre **CR,** et elles sont prises en compte à la fin de l'exécution de l'instruction en cours si elles ne sont pas masquées. Cette requête doit être maintenue active par le périphérique tant qu'elle n'a pas été prise en compte par le processeur.
Le processeur doit alors passer alors en mode superviseur et se brancher au GIET. Après avoir identifié que la cause est une interruption (en examinant le contenu du registre CR), le GIET se branche au **gestionnaire d'interruption,** qui doit appeler la routine d'interruption (ISR) appropriée. Comme il faut reprendre l'exécution du programme en cours à la fin du traitement de l'interruption, il faut sauvegarder une adresse de retour. Lorsqu'il reçoit une requête d'interruption non masquée, le matériel doit donc :

- sauvegarder l'adresse de retour (PC) dans le registre **EPC**
- passer en mode superviseur et masque les interruptions dans **SR**
- écrire qu'il s'agit d'une interruption dans le registre **CR**
- brancher à l'adresse "0x80000180".

En plus des 6 lignes d'interruption matérielles, le processeur MIPS32 possède un mécanisme d'interruption logicielle: Il existe 2 bits dans le registre de cause **CR** qui peuvent être écrits par le logiciel au moyen de l'instruction privilégiée **MTC0**. La mise à 1 de ces bits déclenche le même traitement que les requêtes d'interruptions externes, s'ils ne sont pas masqués.

3) Appels système: instructions SYSCALL et BREAK

L'instruction **SYSCALL** permet à une tâche (utilisateur ou système) de demander un service au système d'exploitation, comme par exemple effectuer une entrée-sortie. Le code définissant le type de service demandé au système, et un éventuel paramètre doivent avoir été préalablement rangés dans des registres généraux. L'instruction **BREAK** est utilisée plus spécifiquement pour poser un point d'arrêt (dans un but de déverminage du logiciel): on remplace brutalement une instruction du programme à

déverminer par l'instruction **BREAK**. Dans les deux cas, le processeur passe en mode superviseur et se branche ici encore au GIET. Après avoir identifié que la cause est un appel système (en examinant le contenu du registre CR), le GIET se branche au **gestionnaire d'appels système.** Lorsqu'il rencontre une des deux instructions **SYSCALL ou BREAK**, le matériel effectue les opérations suivantes :

- sauvegarder PC dans le registre **EPC** (l'adresse de retour est PC + 4)
- passer en mode superviseur et masquage des interruptions dans **SR**
- écrire la cause du déroutement dans le registre **CR**
- brancher à l'adresse "0x80000180".

4) Signal RESET

Le processeur possède également une entrée **RESET** dont l'activation, pendant au moins un cycle, entraîne le branchement inconditionnel au logiciel **boot-loader**. Ce logiciel, implanté conventionnellement à l'adresse 0xBFC00000 doit principalement charger le code du système d'exploitation dans la mémoire et initialiser les périphériques. Cette requête est très semblable à une septième ligne d'interruption externe avec les différences importantes suivantes :

- elle n'est pas masquable.
- il n'est pas nécessaire de sauvegarder une adresse de retour.
- le gestionnaire de reset est implanté à l'adresse "0xBFC00000".

Dans ce cas, le processeur doit:

- passer en mode superviseur et masque les interruptions dans **SR**
- brancher à l'adresse "0xBFC00000"

5) Sortie du GIET ou du bootloader

Avant de reprendre l'exécution d'un programme qui a effectué un appel système (instructions **SYSCALL** ou **BREAK**) ou qui a été interrompu par une interruption, ou pour sortir du **bootloade**r, il est nécessaire d'exécuter l'instruction **ERET**.
Cette instruction modifie le contenu du registre **SR,** et effectue un branchement à l'adresse contenue dans le registre **EPC.**

6) Gestion du registre d'état SR

La figure suivante présente le contenu des 16 bits de poids faible du registre **SR**:

IM[7:0]	0	0	0	UM	0	ERL	EXL	IE

Cette version du processeur MIP32 n'utilise que 12 bts du registre SR :

- **IE : Interrupt Enable**
- **EXL : Exception Level**
- **ERL : Reset Level**
- **UM : User Mode**

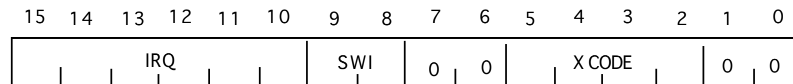
- **IM[7:0] : Masques individuels pour les six lignes d'interruption matérielles (bits IM[7:2]) et pour les 2 interruptions logicielles (bits IM[1:0])**

- Le processeur a le droit d'accéder aux ressources protégées (registres du CP0, et adresses mémoires > 0x7FFFFFFF) si et seulement si le bit UM vaut 0, ou si l'un des deux bits ERL et EXL vaut 1.
- Les interruptions sont autorisées si et seulement si le bit IE vaut 1, et si les deux bits ERL et EXL valent 00, et si le bit correspondant de IM vaut 1.
- Les trois types d'évènements qui déclenchent le branchement au GIET (interruptions, exceptions et appels système) forcent le bit EXL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées.
- L'activation du signal RESET qui force le branchement au Boot-Loader force le bit ERL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées.
- L'instruction ERET force le bit EXL à 0.

Lors de l'activation du RESET, SR contient donc la valeur 0x0004.
Pour exécuter un programme utilisateur en mode protégé, avec interruptions activées, il doit contenir la valeur 0xFF11.
Le code de boot doit écrire la valeur 0xFF13 dans SR et l'adresse du programme utilisateur dans EPC avant d'appeler l'instruction ERET.

7) Gestion du registre de cause CR

Le registre **CR** contient trois champs. Les 4 bits du champ **XCODE(3:0)** définissent la cause de l'appel au **GIET.** Les 6 bits du champ **IRQ(5:0)** représentent l'état des lignes d'interruption externes au moment de l'appel au **GIET.** Les 2 bits **SWI(1:0)** représentent les requêtes d'interruption logicielle.
La figure suivante montre le format du registre de cause **CR** :



Les valeurs possibles du champ XCODE sont les suivantes :

0000	**INT**	**Interruption**
0001		**Inutilisé**
0010		**Inutilisé**
0011		**Inutilisé**
0100	**ADEL**	**Adresse illégale en lecture**
0101	**ADES**	**Adresse illégale en écriture**
0110	**IBE**	**Bus erreur sur accès instruction**
0111	**DBE**	**Bus erreur sur accès donnée**
1000	**SYS**	**Appel système (SYSCALL)**
1001	**BP**	**Point d'arrêt (BREAK)**
1010	**RI**	**Codop illégal**
1011	**CPU**	**Coprocesseur inaccessible**
1100	**OVF**	**Overflow arithmétique**
1101		**Inutilisé**
1110		**Inutilisé**
1111		**Inutilisé**

Processeur MIPS32

Langage d'assemblage

Version 2.10

Sept 2019

Alain Greiner

1/ INTRODUCTION

Ce document décrit le langage d'assemblage du processeur MIPS32, ainsi que différentes conventions relatives à l'écriture des programmes en langage d'assemblage.

Un document séparé décrit l'architecture externe du processeur, c'est-à-dire les registres visibles du logiciel, les règles d'adressage de la mémoire, le codage des instructions-machine, et les mécanismes matériels permettant le traitement des interruptions, des exceptions et des trappes.

Un troisième document décrit le GIET (Gestionnaire d'Interruptions, Exceptions et Trappes) qui définit le code système utilisé dans l'U.E ALMO.

On présente ici successivement l'organisation de la mémoire, les principales règles syntaxiques du langage, les instructions et les macro-instructions, les directives acceptées par l'assembleur, les quelques appels-système disponibles, ainsi que conventions imposées pour les appels de fonctions et la gestion de la pile.

Les programmes assembleur sources qui respectent les règles définies dans le présent document peuvent être assemblés par l'assembleur MIPS de l'environnement GNU GCC pour générer du code exécutable. Ils sont également acceptés par le simulateur MARS utilisé en TP qui permet de visualiser le comportement du processeur instruction par instruction.

2/ ORGANISATION DE LA MEMOIRE

Rappelons que le but d'un programme source *X* écrits en langage d'assemblage est de fournir à un programme particulier (appelé «assembleur») les directives nécessaires pour générer le code binaire représentant les instructions et les données qui devront être chargées en mémoire pour permettre au programme X d'être exécuté par le processeur.

Dans l'architecture MIPS32, l'espace adressable est divisé en deux zones: la zone accessible aux programmes utilisateur, et la zone réservée au superviseur.

Un programme utilisateur utilise généralement trois segments dans la zone utilisateur:

- Le segment .text (appelé seg_code dans le GIET) contient le code exécutable en mode utilisateur. Il est implanté conventionnellement à l'adresse **0x00400000**. Sa taille est fixe. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé en mémoire dans ce segment.

- Le segment .data (appelé seg_data dans le GIET) contient les données globales manipulées par le programme utilisateur. Il est implanté conventionnellement à l'adresse **0x10000000**. Sa taille est fixe. Les valeurs contenues dans ce segment peuvent être

initialisées grâce à des directives contenues dans le programme source en langage d'assemblage.

- Le segment **.stack** (appelé seg_stack dans le GIET) contient la pile d'exécution du programme utilisateur. Sa taille varie au cours de l'exécution, et s'étend vers les adresses décroissantes. Elle est implantée conventionnellement entre les adresses **0x20000000** et **0xFFFFF000**.

Cinq segments sont définis dans la zone superviseur:

- Le segment .ktext (appelé seg_kcode dans le GIET) contient le code exécutable en mode superviseur. Il est implanté conventionnellement à l'adresse **0x80000000**. Sa taille est fixe. Dans le cas du GIET, ce segment contient le code du GIET, ainsi que le code des pilotes de périphériques et le code des routines d'interruptions.
- Le segment .kdata (appelée seg_kdata dans le GIET) contient les données globales manipulées par le système d'exploitation en mode superviseur. Il est implanté conventionnellement à l'adresse **0x81000000**. Sa taille est fixe. Dans le cas du GIET, il contient par exemple les tableaux de sauvegarde des contextes d'exécution des différentes tâches en cours d'exécution sur la machine.
- Le segment .kunc (appelé seg_unckdata dans le GIET) contient les données non cachables du système d'exploitation. Il est implanté conventionnellement à l'adresse 0X**82000000**. Dans le cas du GIET, il contient les variables partagées permettant la communication entre les périphériques et le système d'exploitation.
- Le segment **reset** (appelé seg_reset dans le GIET) contient le code de boot, qui s'exécute évidemment en mode superviseur. Il est implanté conventionnellement à l'adresse **0xBFC00000**. Sa taille et fixe, et il est généralement stocké dans une ROM.
- Les segments associés aux périphériques sont conventionnellement implantés entre les adresses **0x90000000 et 0x9FFFFFFF.** On définit un segment par périphérique, et le nombre de segments dépend du nombre de périphériques présents dans le système.

	Segment	Adresses
Segment noyau	**Reserved**	**0xFFFFFFFF** **0xFFFFF000**
	seg_reset	**0xBFC00000**
	seg_io	**0x90000000**
	seg_unckdata	**0x82000000**
	.kdata	**0x81000000**
	.ktext	**0x80000000**
Segment utilisateur	**Reserved**	**0x7FFFFFFF** **0x7FFFF000**
	.stack ⇩	**0x20000000**
	.data ⇧	**0x10000000**
	.text	**0x00400000**
	Reserved	**0x003FFFFF** **0x00000000**

Segmentation de l'Espace adressable

3/ RÈGLES SYNTAXIQUES

On définit ci-dessous les principales règles d'écriture d'un programme source.

3.1) Les noms de fichiers

Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixés par «.s». Exemple: **monprogramme.s**

3.2) Les commentaires

ils commencent par un **#** et s'achèvent à la fin de la ligne courante.
Exemple :

```
        ##################################################
        # Source Assembleur MIPS de la fonction memcpy
        ##################################################
  …
    lw     $10, 0($11)    # sauve la valeur copiée dans la mémoire
```

3.3) les entiers

Une valeur entière décimale est notée **250** (sans préfixe), et une valeur entière hexadécimale est notée **0xFA** (préfixée par zéro suivi de x). En hexadécimal, les lettres de A à F peuvent être écrites en majuscule ou en minuscule.

3.4) les chaînes de caractères

Elles sont simplement entre guillemets, et peuvent contenir les caractères d'échappement du langage C. Exemple : "Oh la jolie chaine avec retour à la ligne\n".

3.5) les labels

Ce sont des mnémoniques correspondant à des adresses en mémoire. Ces adresses peuvent être soit des adresses de variables stockées en mémoire (principalement dans la section **data**), soit des adresses de saut (principalement dans la section **TEXT**). Ce sont des chaînes de caractères qui doivent commencer par une lettre, un caractère « _ », ou un caractère « . ». Lors de la déclaration, ils doivent être suffixés par le caractère « : » sans espace. Pour y référer, on supprime le «:».
Exemple :

```
        .data
message:
        .asciiz  "Ceci est une chaîne de caractères...\n"
        .text
__start:
        la    $4,    message          # adresse de la chaine dans $4
        ori   $2,    $0,    4         # code de l'appel système dans $2
        syscall
```

Attention : sont illégaux les labels qui ont le même nom qu'un mnémonique de l'assembleur.

3.6) les immédiats

On appelle immédiat un opérande contenu dans l'instruction. Ce sont des constantes. Ce sont soit des entiers, soit des labels. Les valeurs de ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui l'utilise: 16 ou 26 bits.

3.7) les registres

Le processeur MIPS possède 32 registres de travail accessibles au programmeur. Chaque registre est connu par son numéro, qui varie entre 0 et 31, et est préfixé par un **$**. Par exemple, le registre 31 sera noté **$31** dans l'assembleur. En dehors du registre **$0** qui contient toujours la valeur 0, tous les registres sont identiques du point de vue de la machine.
Afin de normaliser et de simplifier l'écriture du logiciel, des conventions d'utilisation des registres sont définies. Ces conventions sont particulièrement nécessaires lors des appels de fonctions. Les noms entre parenthèses sont des alias utilisés par le compilateur GCC.

$0	Vaut 0 en lecture. Non modifié par une écriture
$1 (at)	Réservé à l'assembleur pour les macros.
$2, $3 (v0, V1)	Utilisés pour les calculs temporaires et la valeur de retour des fonctions,
$4 . $7 (a0 .. a3)	Utilisés pour le passage des arguments de fonctions, les valeurs ne sont pas préservées lors des appels de fonctions. Les autres arguments sont placés dans la pile.
$8..$15, $24, $25 (t0..t9)	Registres temporaires de travail, les valeurs ne sont pas préservées lors des appels de fonctions
$16 ,… $23, $30 (s0 … s8)	Registres persistants dont les valeurs sont préservées par les appels de fonctions
$26,$27 (K0, k1)	Réservés aux procédures noyau.
$28 (gp)	Pointeur sur la zone des variables globales (segment data)
$29 (sp)	Pointeur de pile
$31(ra)	Contient l'adresse de retour d'une fonction

Dans le tableau ci-dessus, les noms entre parenthèses sont les noms symboliques utilisés par gcc, mais xspim ne les comprends pas. $at est équivalent à $1, $v0 est équivalent à $2, $a0 est équivalent à $4, etc.

- at : Assemnbleur Temporary
- v0, v1 : Value 0 et Value 1
- a0 ... a3 : Argument 0 à Argument 3
- t0 ... t9 : Temporary 0 à Temporary 9
- s0 ... s8 : Saved value 0 à Save value 8
- k0, k1 : Kernel temporary 0 et Kernel temporary 1
- gp : Global Pointer
- sp : Stack Pointer
- ra : Return Address

3.8) les arguments

La plupart des instructions nécessitent un ou plusieurs arguments. Si une instruction nécessite plusieurs arguments, ces arguments sont séparés par des virgules. Dans une instruction assembleur, on aura en général comme premier argument le registre dans lequel

est mis le résultat de l'opération, puis ensuite le premier registre source, puis enfin le second registre source ou une constante.

Exemple: add $8, $4, $5

3.9) l'adressage mémoire

Le MIPS ne possède qu'un unique mode d'adressage pour lire ou écrire des données en mémoire: l'adressage indirect registre avec déplacement. L'adresse est obtenue en additionnant le déplacement (positif ou négatif) au contenu du registre.

```
Exemples:   lw    $12,   13($10)      # $12 <= Mem[$10 + 13]
            Sw    $20,   -60($22)     # Mem[$22 – 60] <= $20
```

S'il n'y a pas d'entier devant la parenthèse ouvrante, le déplacement est nul.

Pour ce qui concerne les sauts, il n'est pas possible d'écrire des sauts à des adresses constantes, par exemple un **j 0x400000** ou **bnez $3, -12**. Il faut nécessairement utiliser des labels.

4/ INSTRUCTIONS

Dans ce qui suit, le registre noté **$rr** est le registre destination, c.-à-d. qui reçoit le résultat de l'opération, les registres notés **$ri** et **$rj** sont les registres source qui contiennent les valeurs des opérandes sur lesquelles s'effectuent l'opération.
Notons qu'un registre source peut être le registre destination d'une même instruction assembleur.
Un opérande immédiat sera noté **imm**, et sa taille sera spécifié dans la
Description : de l'instruction.
Les instructions de saut prennent comme argument une étiquette (ou label), qui est utilisée pour calculer l'adresse de saut. Toutes les instructions modifient le registre **PC** (program counter), qui contient l'adresse de l'instruction suivante à exécuter. Enfin, le résultat d'une multiplication ou d'une division est rangé dans deux registres spéciaux, **HI** pour les poids forts, et **LO** pour les poids faibles.

Ceci nous amène à introduire quelques notations :

Notation	Signification
+	Addition entière en complément à 2
-	Soustraction entière en complément à 2
x	Multiplication entière en complément à 2
/	Division entière en complément à 2
mod	Reste de la division entière en complément à 2
and	Opérateur et logique bit à bit
or	Opérateur ou logique bit à bit
nor	Opérateur non-ou logique bit à bit
xor	Opérateur ou-exclusif logique bit à bit
Mem[ad]	Contenu de la mémoire à l'adresse ad
<=	Assignation
||	Concaténation entre deux chaînes de bits
B^n	Réplication du bit B n fois dans une chaîne de bits
$X_{p...q}$	Sélection des bits p à q dans une chaîne de bits X

add Addition registre registre signée

Syntaxe : add $rr, $ri, $rj

Description : Les contenus des registres $ri et $rj sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre $rr

rr <= ri + rj

Exception : génération d'une exception si dépassement de capacité.

addi Addition registre immédiat signée

Syntaxe : addi $rr, $ri, imm

Description : La valeur immédiate sur 16 bits subit une extension de signe, et est ajoutée au contenu du registre $ri pour former un résultat sur 32 bits qui est placé dans le registre $rr.

rr <= imm_{15}^{16} || $imm_{15...0}$ + ri

Exception : génération d'une exception si dépassement de capacité.

addiu Addition registre immédiat sans détection de dépassement

Syntaxe : addiu $rr, $ri, imm

Description : La valeur immédiate sur 16 bits subit une extension de signe, et est ajoutée au contenu du registre $ri pour former un résultat sur 32 bits qui est placé dans le registre $rr.

rr <= (imm_{15}^{16} || $imm_{15...0}$) + ri

Exception : pas d'exception

addu Addition registre registre sans détection de dépassement

Syntaxe : addu $rr, $ri, $rj

Description : Les contenus des registres $ri et $rj sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre $rr

rr <= ri + rj

Exception : pas d'exception

and Et bit-à-bit registre registre

Syntaxe : and $rr, $ri, $rj
Description : Un et bit-à-bit est effectué entre les contenus des registres $ri et $rj. Le résultat est placé dans le registre $rr.

rr <= ri and rj

Exception : pas d'exception

andi Et bit-à-bit registre immédiat

Syntaxe : andi $rr, $ri, imm

Description : La valeur immédiate sur 16 bits subit une extension de zéros. Un et bit-à-bit est effectué entre cette valeur étendue et le contenu du registre $ri pour former un résultat placé dans le registre $rr.

rr <= (0^{16} || imm) and ri

Exception : pas d'exception

beq Branchement si registre égal registre

Syntaxe : beq $ri, $rj, label

Description : Les contenus des registres $ri et $rj sont comparés. S'ils sont égaux, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

ad <= label
if (ri = rj) pc <= pc + 4 + ad

Exception : pas d'exception

bgez Branchement si registre supérieur ou égal à zéro

Syntaxe : bgez $ri, label

Description : Si le contenu du registre $ri est supérieur ou égal à zéro, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

ad <= label
if (ri >= 0) pc <= pc + 4 + ad

Exception : pas d'exception

bgezal Branchement à une fonction si registre supérieur ou égal à zéro

Syntaxe : bgezal $ri, label

Description : Inconditionnellement, l'adresse de l'instruction suivant l'instruction bgezal est stockée dans le registre $31. Si le contenu du registre $ri est supérieur ou égal à zéro, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

ad <= label
r31 <= pc + 4
if (ri >= 0) pc <= pc + 4 + ad

Exception : pas d'exception

bgtz **Branchement si registre strictement supérieur à zéro**

Syntaxe : bgtz $ri, label

Description : Si le contenu du registre $ri est strictement supérieur à zéro, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

```
ad <= label
if (ri > 0)  pc <= pc + 4 + ad
```

Exception : pas d'exception

blez **Branchement si registre inférieur ou égal à zéro**

Syntaxe : blez $ri, label

Description : Si le contenu du registre $ri est inférieur ou égal à zéro ,le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

```
ad <= label
if (ri <= 0)  pc <= pc + 4 + ad
```

Exception : pas d'exception

bltz **Branchement si registre strictement inférieur à zéro**

Syntaxe : bltz $ri, label

Description : Si le contenu du registre $ri est strictement inférieur à zéro ,le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

```
ad <= label
if (ri < 0)  pc <= pc + 4 + ad
```

Exception : pas d'exception

bltzal **Branchement à une fonction si registre inférieur ou égal à zéro**

Syntaxe : bltzal $ri, label

Description : Inconditionnellement, l'adresse de l'instruction suivant l'instruction bgezal est sauvée dans le registre $31. Si le contenu du registre $ri est strictement inférieur à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

```
ad <= label
r31 <=  pc + 4
if (ri < 0)  pc <= pc + 4 + ad
```

Exception : pas d'exception

bne **Branchement si registre différent de registre**

Syntaxe : bne $ri, $rj, label

Description : Les contenus des registres $ri et $rj sont comparés. S'ils sont différents, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

```
ad <= label
If (ri not= rj )  pc <= pc + 4 + ad
```

Exception : pas d'exception

break **Arrêt et saut à la routine d'exception**

Syntaxe : break imm

Description : Un point d'arrêt est détecté, et le programme saute à l'adresse de la routine de gestion des exceptions.

```
Pc <= 0x80000080
```

Exception : déclenchement d'une exception de type point d'arrêt.

div **Division entière signée**

Syntaxe : div $ri, $rj

Description : Le contenu du registre $ri est divisé par le contenu du registre $rj, le contenu des deux registres étant considéré comme des nombres en complément à deux. Le quotient de la division est placé dans le registre spécial lo, et le reste dans dans le registre spécial hi.

```
lo <=  ri / rj
hi <=  ri mod rj
```

Exception : pas d'exception

divu **Division entière non-signée**

Syntaxe : divu $ri, $rj

Description : Le contenu du registre $ri est divisé par le contenu du registre $rj, le contenu des deux registres étant considéré comme des nombres non signés.
Le quotient de la division est placé dans le registre spécial lo, et le reste dans dans le registre spécial hi.

```
lo <=  (0 || ri) / (0 || rj)
hi <=  (0 || ri) mod (0 || rj)
```

Exception : pas d'exception

eret **Sortie du GIET ou du Boot-Loader**

Syntaxe : eret

Description : Force à 0 le bit EXL du registre SR et branche à l'adresse contenue dans le registre EPC.

sr <= sr & 0xFFFFFFFD
pc <= epc

j **Saut inconditionnel immédiat**

Syntaxe : j label

Description : Le programme saute inconditionnellement à l'adresse correspondant au label, calculé par l'assembleur.

pc <= label

Exception : pas d'exception

jal **Appel de fonction inconditionnel immédiat**

Syntaxe : jal label

Description : L'adresse de l'instruction suivant l'instruction jal est stockée dans le registre $31. Le programme saute inconditionnellement à l'adresse correspondant au label, calculée par l'assembleur.

r31 <= pc + 4
pc <= label

Exception : pas d'exception

jalr **Appel de fonction inconditionnelle registre**

Syntaxe : jalr $ri ou jalr $rr, $ri

Description : Le programme saute à l'adresse contenue dans le registre $ri . L'adresse de l'instruction suivant l'instruction jalr est sauvée dans le registre $rr. Si le registre n'est pas spéficié, alors c'est par défaut le registre $31} qui est utilisé.

rr <= pc + 4
pc <= ri

Exception : pas d'exception

jr **Branchement inconditionnel registre**

Syntaxe : jr $ri

Description : Le programme saute à l'adresse contenue dans le registre $ri.

pc <= ri

Exception : pas d'exception

lb **Lecture d'un octet signé en mémoire**

Syntaxe : lb $rr, imm($ri)

Description : L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. L'octet lu à cette adresse subit une extension de signe et est placé dans le registre $rr.

$rr \Leftarrow (Mem[imm + ri])_7^{24} \; || \; (Mem\}[imm + ri])_{7...0}$

Exceptions : - Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

lbu **Lecture d'un octet non-signé en mémoire**

Syntaxe : lbu $rr, imm($ri)

Description : L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. L'octet lu à cette adresse subit une extension avec des zéros et est placé dans le registre $rr.

$rr \Leftarrow (0)^{24} \; || \; (Mem\}[imm + ri])_{7...0}$

Exceptions : - Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

lh **Lecture d'un demi-mot signé en mémoire**

Syntaxe : lh $rr, imm($ri)

Description : L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. Le demi-mot de 16 bits lu à cette adresse subit une extension de signe et est placé dans le registre $rr.

$rr \Leftarrow (Mem[imm + ri])_{15}^{16} \; || \; (Mem\}[imm + ri])_{15...0}$

Exceptions : - Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

lhu **Lecture d'un demi-mot non-signé en mémoire**

Syntaxe : lhu $rr, imm($ri)

Description : L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
. Le demi-mot de 16 bits lu à cette adresse subit une extension avec des zéro et est placé dans le registre $rr.

rr <= $(0)^{16}$ || (Mem}[imm + ri]$)_{15...0}$

Exceptions : - Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

lui **Chargement d'une constante dans les poids forts d'un registre**

Syntaxe : lui $rr, imm

Description : La constante immédiate de 16 bits est décalée de 16 bits à gauche, et est complétée de zéro. La valeur ainsi obtenue est placée dans le registre $rr .

rr <= imm || $(0)^{16}$

Exception : pas d'exception

lw **Lecture d'un mot de la mémoire**

Syntaxe : lw $rr, imm($ri)

Description : L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
Le demi-mot de 32 bits lu à cette adresse est placé dans le registre $rr.

rr <= Mem[imm + ri]

Exceptions : - Adresse non alignée sur une frontière de mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

mfc0 **Copie d'un registre spécial dans d'un registre général**

Syntaxe : mfc0 $rt, $rd

Description : Le contenu du registre spécial $rd --- non directement accessible au programmeur --- est recopié dans le registre général $rt.
Les registres spéciaux servent à la gestion des exceptions et interruptions, et sont les suivants :
$8 pour BAR} (bad address register),
$12 pour SR (status register),
$13 pour CR (cause register)
$14 pour EPC (exception program counter)

rt <= rd

Exception : registre spécial non défini.

mfhi **Copie le registre hi dans un registre général**

Syntaxe : mfhi $rr

Description : Le contenu du registre hi --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général $rr.

rr <= hi

Exception : pas d'exception

mflo **Copie le registre lo dans un registre général**

Syntaxe : mflo $rr

Description : Le contenu du registre lo --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général $rr.

rr <= lo

Exception : pas d'exception.

mtc0 **Copie d'un registre général dans un registre spécial**

Syntaxe : mtc0 $rt, $rd

Description : Le contenu du registre général $rt est recopié dans le registre spécial $rd --- non directement accessible au programmeur --- Les registres spéciaux servent à la gestion des exceptions et interruptions, et sont les suivants :
$8 pour BAR (bad address register),
$12 pour SR (status register),
$13 pour CR (cause register)
$14 pour EPC (exception program counter)

rt <= rd

Exception : registre spécial non défini.

mthi **Copie d'un registre général dans le registre hi**

Syntaxe : mthi $ri

Description : Le contenu du registre général $ri est recopié dans le registre hi.

Exception : pas d'exception.

mtlo **Copie d'un registre général dans le registre lo**

Syntaxe : mtlo $ri

Description : Le contenu du registre général $ri est recopié dans le registre lo.

Exception : pas d'exception.

mult **Multiplication signée**

Syntaxe : mult $ri, $rj

Description : Le contenu du registre $ri est multiplié par le contenu du registre $rj, le contenu des deux registres étant considéré comme des nombres en complément à deux. Les 32 bits de poids fort du résultat sont placés dans le registre hi, et les 32 bits de poids faible dans lo.

$lo <= (ri \times rj)_{31...0}$
$hi <= (ri \times rj)_{63...32}$

Exception : pas d'exception.

multu **Multiplication non-signée**

Syntaxe : multu $ri, $rj

Description : Le contenu du registre $ri est multiplié par le contenu du registre $rj, le contenu des deux registres étant considéré comme des nombres non signés. Les 32 bits de poids fort du résultat sont placés dans le registre hi, et les 32 bits de poids faible dans lo.

$lo <= ((0 || ri) \times (0 || rj))_{31...0}$
$hi <= ((0 || ri) \times (0 || rj))_{63...32}$

Exception : pas d'exception.

nor **Non-ou bit-à-bit registre registre**

Syntaxe : nor $rr, $ri, $rj

Description : Un non-ou bit-à-bit est effectué entre les contenus des registres $ri et $rj. Le résultat est placé dans le registre $rr.

rr <= ri nor rj

Exception : pas d'exception.

or **Ou bit-à-bit registre registre**

Syntaxe : or $rr, $ri, $rj

Description : Un ou bit-à-bit est effectué entre les contenus des registres $ri et $rj. Le résultat est placé dans le registre $rr.

rr <= ri or rj

Exception : pas d'exception.

ori **Ou bit-à-bit registre immédiat**

Syntaxe : ori $rr, $ri, imm

Description : La valeur immédiate sur 16 bits subit une extension de zéros. Un ou bit-à-bit est effectué entre cette valeur étendue et le contenu du registre $ri pour former un résultat placé dans le registre $rr.

$rr <= ((0)^{16} || imm)$ or ri

Exception : pas d'exception.

sb **Écriture d'un octet en mémoire**

Syntaxe : sb $rj, imm($ri)

Description : L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
. L'octet de poids faible du registre $rj est écrit à l'adresse ainsi calculée.

$Mem[imm + ri] <= rj_{7...0}$

Exceptions :
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

sh **Écriture d'un demi-mot en mémoire**

Syntaxe : sh $rj, imm($ri)

Description : L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. Les deux octets de poids faible du registre $rj sont écrit à l'adresse ainsi calculée. Le bit de poids faible de cette adresse doit être à zéro.

$Mem[imm + ri] <= rj_{15...0}$

Exceptions :
- Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

sll **Décalage à gauche immédiat**

Syntaxe : sll $rr, $ri, imm

Description : Le registre est décalé à gauche de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre $rr.

$rr <= ri_{(31 - imm)...0} || (0)^{imm}$

Exception : pas d'exception.

sllv **Décalage à gauche registre**

Syntaxe : sllv $rr, $ri, $rj

Description : Le registre $ri est décalé à gauche du nombre de bits spécifiés dans les 5 bits de poids faible du registre $rj, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre $rr.

rr <= ri $_{(31-rj)...0}$ || (0) ^{rj}

Exception : pas d'exception.

slt **Comparaison signée registre registre**

Syntaxe : slt $rr, $ri, $rj

Description : Le contenu du registre $ri est comparé au contenu du registre $rj, les deux valeurs étant considérées comme des nombres signés.
Si la valeur contenue dans $ri est strictement inférieure à celle contenue dans $rj, alors $rr prend la valeur un, sinon il prend la valeur zéro.

```
If (ri < rj)  rr <= 1
else          rr <= 0
```

Exception : pas d'exception.

slti **Comparaison signée registre immédiat**

Syntaxe : slti $rr, $ri, imm

Description : Le contenu du registre est comparé à la valeur immédiate sur 16 bits qui a subi une extension de signe. Les deux valeurs sont considérées comme des nombres signés. Si la valeur contenue dans $ri est strictement inférieure à celle de l'immédiat, alors $rr prend la valeur un, sinon il prend la valeur zéro.

```
If (ri < imm)  rr <= 1
else           rr <= 0
```

Exception : pas d'exception.

sltiu **Comparaison non-signée registre immédiat**

Syntaxe : sltiu $rr, $ri, imm

Description : Le contenu du registre est comparé à la valeur immédiate sur 16 bits qui a subi une extension de signe.
Les deux valeurs étant considérées comme des nombres non-signés,
Si la valeur contenue dans $ri est strictement inférieur à celle de l'immédiat étendu, alors $rr prend la valeur un, sinon il prend la valeur zéro.

```
if ((0||ri) < (0||imm))  rr <= 1
else                     rr <= 0
```


Exception : pas d'exception

sltu **Comparaison non-signée registre registre**

Syntaxe : sltu $rr, $ri, $rj

Description : Le contenu du registre $ri est comparé au contenu du registre $rj, les deux valeurs étant considérés comme des nombres non-signés.
Si la valeur contenue dans ri est strictement inférieur à celle contenue dans $rj, alors $rr prend la valeur un, sinon il prend la valeur zéro.

```
If ((0||ri) < (0|| rj))   rr <= 1
else                      rr <= 0
```

Exception : pas d'exception

sra **Décalage à droite arithmétique immédiat**

Syntaxe : sra $rr, $ri, imm
Description : Le registre $ri est décalé à droite de la valeur immédiate codée sur 5 bits, le bit de signe du registre $ri étant introduit dans les bits de poids fort.
Le résultat est placé dans le registre .

rr <= (ri $_{31}$) ^{imm} || (ri) $_{31...imm}$

Exception : pas d'exception

srav **Décalage à droite arithmétique registre**

Syntaxe : srav $rr, $ri, $rj

Description : Le registre $ri est décalé à droite du nombre de bits spécifié dans les 5 bits de poids faible du registre $rj, le bit de signe de $ri étant introduit dans les bits de poids fort.
Le résultat est placé dans le registre $rr.

rr <= (ri $_{31}$) ^{rj} || (ri) $_{31...rj}$

Exception : pas d'exception

srl **Décalage à droite logique immédiat**

Syntaxe : srl $rr, $ri, imm

Description : Le registre est décalé à droite de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids fort.
rr <= (0) ^{imm} || (ri) $_{31...imm}$

Exception : pas d'exception

srlv **Décalage à droite logique registre**

Syntaxe : srlv $rr, $ri, $rj

Description : Le registre $ri est décalé à droite du nombre de bits spécifié dans les 5 bits de poids faible du registre $rj des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre $rr .
rr <= $(0)^{rj}$ || $(ri)_{31...rj}$

Exception : pas d'exception

sub **Soustraction registre registre signée**

Syntaxe : sub $rr, $ri, $rj

Description : Le contenu du registre $rj est soustrait du contenu du registre $ri pour former un résultat sur 32 bits qui est placé dans le registre $rr.
rr <= ri - rj

Exception : génération d'une exception si dépassement de capacité.

subu **Soustraction registre registre non-signée**

Syntaxe : sub $rr, $ri, $rj

Description : Le contenu du registre $rj est soustrait du contenu du registre pour former un résultat sur 32 bits qui est placé dans le registre .
rr <= ri – rj

Exception : pas d'exception

sw **Écriture d'un mot en mémoire**

Syntaxe : sw $rj, imm($ri)

Description : L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
Le contenu du registre $rj est écrit en mémoire à l'adresse ainsi calculée.
Les deux bits de poids faible de cette adresse doivent être nuls.
Mem[imm + ri] <= rj

Exceptions :
- Adresse non alignée sur une frontière de mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

syscall **Appel à une fonction du système (en mode noyau).**

Syntaxe : syscall

Description : Un appel système est effectué, par un branchement inconditionnel au gestionnaire d'exception.
Note : par convention, le numéro de l'appel système, c.-à-d. le code de la fonction système à effectuer, est placé dans le registre $2..
Pc <= 0x80000080

xor **Ou-exclusif bit-à-bit registre registre**

Syntaxe : xor $rr, $ri, $rj

Description : Un ou-exclusif bit-à-bit est effectué entre les contenus des registres $ri et $rj. Le résultat est placé dans le registre $rr .
rr <= ri xor rj

Exception : pas d'exception

xori **Ou-exclusif bit-à-bit registre immédiat**

Syntaxe : xori $rr, $ri, imm

Description : La valeur immédiate sur 16 bits subit une extension de zéros.
Un ou-exclusif bit-à-bit est effectué entre cette valeur étendue et le contenu du registre $ri pour former un résultat placé dans le registre $rr.
rr <= $((0)^{16}$ || imm) xor ri

Exception : pas d'exception

5/ MACRO-INSTRUCTIONS

Une macro-instruction est une pseudo-instruction qui ne fait pas partie du jeu d'instructions-machine, mais qui est acceptée par l'assembleur qui la traduit en une séquence de plusieurs instructions-machine. Les macro-instructions utilisent le registre $1 si elles ont besoin de faire un calcul intermédiaire. Il ne faut donc pas utiliser ce registre dans les programmes.

la **Chargement d'une adresse dans un registre**

Syntaxe : la $rr, adr

Description : L'adresse considérée comme une quantité non-signée est chargée dans le registre .

Code équivalent
Calcul de adr par l'assembleur

```
Lui   $rr,   adr >> 16
ori   $rr,   $rr,   adr & 0xFFFF
```

li **Chargement d'un opérande immédiat sur 32 bits dans un registre**

Syntaxe : li $rr, imm

Description : La valeur immédiate est chargée dans le registre .$rr .

Code équivalent

```
lui   $rr,   imm >> 16
ori   $rr,   $rr,   imm & 0xFFFF
```


6/ DIRECTIVES SUPPORTÉES PAR L'ASSEMBLEUR MIPS

Les directives ne sont pas des instructions exécutables par la machine, mais permettent de donner des ordres au programme d'assemblage. Toutes ces pseudo-instructions commencent par le caractère « .» ce qui permet de les différencier clairement des instructions.

6.1) Déclaration des sections

Des directives permettent de nommer la section concernée par les instructions, macro-instructions ou directives qui les suivent. Ce nommage des sections est utilisé au moment de l'édition de liens, pour indiquer comment ces sections doivent être regroupées dans les différents segments qui structurent l'espace d'adressage. Les deux sections utilisées en pratique pour le code utilisateur sont les sections .text (qui ont vocation à être regroupées dans le segment seg_code), et l.data (qui ont vocation à être regroupées dans le segment seg_data). Le programme assembleur gère donc deux compteurs d'adresse indépendants pour ces deux sections.

- .text
- .data

Toutes les instructions ou directives qui suivent une de ces six directives concernent la section correspondante.

6.2) Déclaration et initialisation de variables

Les directives suivantes permettent d'initialiser les valeurs contenues dans certaines sections de la mémoire (uniquement TEXT, data, ktext, et kdata).

.align n

Description : Cette directive aligne le compteur d'adresse de la section concernée sur une adresse telle que les n bits de poids faible soient à zéro.

Exemple:

```
.align 2
.byte 12
.align 2
.byte 24
```

.ascii chaine, [autrechaine]•••

Description : Cette directive place à partir de l'adresse du compteur d'adresse de la section concernée la suite de caractères entre guillemets. S'il y a plusieurs chaînes, elles sont placées à la suite. Cette chaîne peut contenir des séquences d'échappement du langage C, et doit être terminée par un zéro binaire si elle est utilisée avec un appel système.

Exemple:

```
message:
```

```
.ascii "Bonjour, Maitre!\n\0"
```

.asciiz chaine, [autrechaine]…

Description : Cette directive opérateur est strictement identique à la précédente, la seule différence étant qu'elle ajoute un zéro binaire à la fin de chaque chaîne.

Exemple:
```
message:
    .ascii "Bonjour, Maître''
```

.byte n, [m]

Description : La valeur de chacune des expressions n,m,… est tronquée à 8 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.

Exemple:
```
table:
        .byte 1, 2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 1
```

.half n, [m]…

Description : La valeur de chacune des expressions n,m,… est tronquée à 16 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.

Exemple:
```
coordonnées:
        .half 0  , 1024
```

.word n, [m]…

Description : La valeur de chaque expression est placée dans des adresses successives de la section active.

Exemple:
```
entiers:
.word -1, -1000, -100000, 1, 1000, 100000
```

.space n

Description : Un espace de taille n octets est réservé à partir de l'adresse courante de la section active.

Exemple:
```
nuls:
    .space 1024        # initialise 1 kilo de mémoire à zéro
```


7/APPELS SYSTÈME SUPPORTES PAR MARS

Certains traitements ne peuvent être exécutés que sous le contrôle du système d'exploitation (typiquement les entrées/sorties consistant à lire ou écrire un nombre, ou une chaîne de caractère sur la console). En assembleur, un programme utilisateur doit effectuer un « appel système », en utilisant l'instruction **syscall**.

Par convention, le numéro de l'appel système est contenu dans le registre $2, et ses éventuels arguments dans les registres $4, $5, $6, $7.

7.1) Appels système supportés par le simulateur MARS

L'environnement de simulation MARS est très limité, car il vise principalement l'apprentissage de la programmation en assembleur. On n'utilise qu'un seul périphérique, qui est un contrôleur de terminal TTY (Écran/Clavier). Le code des appels système n'est pas réellement exécuté par le processeur MIPS, mais il est directement exécuté sur la station de travail qui effectue la simulation. Les appels système suivants sont supportés par MARS :

7.1.1 Écrire un entier sur la console

Il faut mettre l'entier à écrire dans le registre $4 et exécuter l'appel système numéro 1.

```
li      $4,   1234567     # stocke la valeur 1234567 dans $4
ori     $2,   $0,   1     # code de « print_integer » dans $2
syscall                   # affiche  la valeur numérique
```

7.1.2 lire un entier sur la console

La valeur de retour d'une fonction --- système ou autre --- doit être rangée par la fonction appelée dans le registre $2. Ainsi, lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre $2.

```
ori     $2,   $0,   5     # code de « read_integer »
syscall                   # $2 contient la valeur lue
```

7.1.3 Écrire un caractère sur la console

Il faut mettre le code ascii du caractère à écrire dans le registre $4 et exécuter l'appel système numéro 11.

```
li      $4,   0x30        # stocke le caractère dans $4
ori     $2,   $0,   11    # code de « print_char » dans $2
syscall                   # affiche  le caractère
```

7.1.4 lire un caractère sur la console

La valeur de retour d'une fonction --- système ou autre --- doit être rangée par la fonction appelée dans le registre $2. Ainsi, lire un caractère consiste à exécuter l'appel système numéro 12 et récupérer le résultat dans le registre $2.

```
        ori     $2,     $0,     12      # code de « read_char »
        syscall                         # $2 contient la valeur lue
```

7.1.5 Écrire une chaîne de caractères sur la console

Une chaîne de caractères étant identifiée par un pointeur, il faut passer ce pointeur à l'appel système numéro 4 pour l'afficher. La chaîne de caractère doit se terminer par un caractère de valeur nulle.

```
str:    .asciiz "Chaîne à afficher\n"
        la      $4,     str             # charge le pointeur dans $4
        ori     $2,     $0,     4       # code de « print_string » dans $2
        syscall                         # affiche la chaine pointée
```

7.1.6 Lire une chaîne de caractères sur la console

Pour lire une chaîne de caractères, il faut un pointeur définissant l'adresse du buffer de réception en mémoire et un entier définissant la taille du buffer (en nombre de caractères). On écrit la valeur du pointeur dans $4, et la taille du buffer dans $5, et on exécute l'appel système numéro 8.

```
read:   .space 256
        la      $4,     read            # charge le pointeur dans $4
        ori     $5,     $0,     255     # charge longueur max dans $5
        ori     $2,     $0,     8       # code de « read_string »
        syscall                         # copie la chaîne dans le buffer pointé par $4
```

7.1.7 Terminer un programme

L'appel système numéro 10 effectue l' exit du programme au sens du langage C.

```
        ori     $2,     $0,     10      # code de « exit »
        syscall                         # quitte pour de bon
```


7.2) Appels système supportés par le GIET

Pour permettre à un programme utilisateur écrit en C d'accéder aux périphériques, le GIET (Gestionnaire d'Interruptions, Exceptions et trappes) définit dans le fichier **stdio.c** un ensemble de fonctions directement utilisables par un programme utilisateur. Ce sont ces fonctions C qui encapsulent l'instruction assembleur **syscall**, et qui se chargent du passage des arguments ainsi que de la récupération du résultat.
Ces fonctions C "utilisateurs" font elles-mêmes appel à une API de plus bas niveau, c'est-à-dire à d'autres fonctions C "système", définies dans le fichier drivers.c.

Les "appels systèmes" sont donc contenus dans deux fichiers: le fichier **stdio.c** contient le code côté utilisateur, et le fichier **drivers.c** contient le code côté superviseur.

Le GIET supporte actuellement les périphériques suivants:

7.2.1 Terminal TTY

```
int   tty_puts(char* string);
int   tty_putc(char byte);
int   tty_putw(int word);
int   tty_gets(char* buf, int bufsize);
int   tty_getc(char* byte);
int   tty_getw(int* word);
int   tty_printf(char* format,...);
```

7.2.2 Timer programmable

```
int   timer_set_mode(int timer_index, int mode);
int   timer_set_period(int timer_index, int period);
int   timer_reset_irq(int timer_index);
int   timer_get_time(int timer_index, int* time);
```

7.2.3 Contrôleur d'Interruptions ICU

```
int   icu_set_mask(int val);
int   icu_clear_mask(int val);
int   icu_get_mask(int* buffer);
int   icu_get_irqs(int* buffer);
int   icu_get_index(int* buffer);
```

7.2.4 Contrôleur de verrous

```
int   lock_acquire(int lock_index);
int   lock_release(int lock_index);
```

7.2.5 Contrôleur d'entrées/sorties sur disque

```
int   ioc_read(size_t lba, void* buffer, size_t count);
int   ioc_write(size_t lba, void* buffer, size_t count);
int   ioc_completed();
```

7.2.6 Frame Buffer (Ecran Graphique)

```
int   fb_read(size_t offset, void* buffer, size_t length);
int   fb_write(size_t offset, void* buffer, size_t length);
```

```
int    fb_completed();
int    fb_sync_read(size_t offset, void* buffer, size_t length);
int    fb_sync_write(size_t offset, void* buffer, size_t length);
```

7.2.7 Services processeur

```
int    procid();
int    proctime();
int    exit();
int    rand();
```


8/ CONVENTIONS POUR LES APPELS DE FONCTIONS

Les conventions définies ci-dessous sont utilisées par le compilateur GCC.

8.1) Fonction appelante et fonction appelée

Pour expliquer les conventions d'appel de fonction, lors d'un appel nous allons distinguer explicitement la fonction appelante et la fonction appelée. La fonction appelante (nommons-là **F()**) appelle la fonction appelée (nommons-là **g()**) en lui fournissant des arguments. Les propriétés "appelante" et "appelée" sont propres à chaque appel. En effet, la fonction "appelée", **g(),** peut à son tour devenir une fonction "appelante" si son code contient des appels de fonctions.

8.2) Type de registres

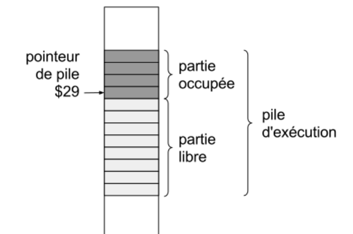
Lorsque l'on programme une fonction **F()**, on distingue deux types de registres utilisables (cf. section 3.7 page 6). D'une part, les registres temporaires que la fonction **F()** peut modifier sans en restaurer la valeur initiale en sortant (sortir signifie un retour vers la fonction appelante de la fonction **F()** à la fin de son exécution) et, d'autre part, les registres persistants que la fonction **F()** peut utiliser, mais qu'elle doit sauvegarder pour pouvoir les restaurer dans leur état initial avant de sortir. Seuls les registres $16 à $23 et $28 à $31 sont persistants et doivent être restaurés s'ils ont été modifiés. Les registres $1 à $15 et $24, $25 sont temporaires et donc utilisables sans sauvegarde préalable. Certains registres temporaires ont un rôle spécifique : $1 est réservé pour les macro-instructions, $2 et $3 sont utilisés pour la valeur de retour de la fonction, enfin $4 à $7 sont utilisés pour le passage des arguments.

8.3) Contexte d'exécution d'une fonction

Pour s'exécuter, une fonction **F()** utilise une zone en mémoire que l'on nomme **contexte d'exécution**. Un contexte d'exécution contient 3 zones distinctes correspondant à trois usages : la première zone permet à **F()** de sauvegarder les registres persistants qu'elle utilise, la deuxième zone permet à **F()** de stocker ses variables locales et la troisième zone permet à **F()** de passer les arguments aux fonctions qu'elle appelle.
Attention : les arguments que la fonction **F()** a reçus de sa fonction appelante ne se trouvent pas dans le contexte de **F()**, ils se trouvent dans le contexte de la fonction appelante de **F()**.

Les contextes d'exécution des fonctions sont alloués dynamiquement dans la **pile d'exécution**.

- La pile d'exécution est la structure de données contenant les contextes d'exécution de toutes les fonctions.
- La pile est découpée en deux parties, une partie occupée et une partie libre.
- La pile s'étend vers les adresses décroissantes.
- Le pointeur de pile contient toujours l'adresse de la dernière case occupée dans la pile. Ceci signifie que toutes les cases d'adresse inférieure au pointeur de pile sont libres.
- Le pointeur de pile se trouve dans registre **$29** par convention.



La zone de sauvegarde des registres persistants

- Le processeur MIPS32 impose l'utilisation du registre $31 pour stocker l'adresse de retour lors d'un appel de fonction. Cela signifie que lorsque le processeur se branche à la première instruction d'une fonction, le registre $31 contient l'adresse de retour de la fonction.
- La fonction **F()** est chargée de sauvegarder le registre $31, ainsi que les registres persistants qu'elle utilise de façon à pouvoir restaurer leur valeur avant de revenir à la fonction appelante. Les registres sont placés dans la pile suivant l'ordre croissant, le registre d'indice le plus petit à l'adresse la plus petite. Ainsi le registre $31, contenant l'adresse de retour, est toujours stocké à l'adresse la plus grande de la zone de sauvegarde.
- Dans la suite de ce document, on note **nr** le nombre de registres sauvegardés en comptant le registre $31.

La zone des variables locales de la fonction

- Les valeurs stockées dans cette zone sont toujours lues et écrites par la fonction appelée. Elle est utilisée pour stocker les variables et structures de données locales à la fonction (c'est à dire l'ensemble des variables déclarées à l'intérieur de la fonction).
- Dans la suite de ce document, on note **nv** le nombre de mots de 32 bits constituant la zone des variables locales.

La zone des arguments des fonctions appelées

- La fonction courante doit réserver de place pour le passage des arguments des fonctions qu'elle appelle.
- Dans la suite de ce document, on note **na** le nombre de mots de 32 bits nécessaires pour le passage des arguments.
- Dans le cas général, une fonction **F()** appelle plusieurs fonctions (nommons ces fonctions les **g0()**, **g1()**, … **gn()**). Ces fonctions ont généralement des nombres d'arguments différents ($\mathbf{na}_{g0}$, $\mathbf{na}_{g1}$, … $\mathbf{na}_{gn}$). Les $\mathbf{na}_{gi}$ sont les nombres d'arguments des fonctions appelées.
  Le nombre **na** est la valeur maximale des $\mathbf{na}_{gi}$: **na** = MAX($\mathbf{na}_{gi}$).
- Le MIPS impose une optimisation pour le passage des arguments. La fonction **F()** a réservé dans la pile l'espace nécessaire pour tous les arguments de la fonction appelée, **mais elle n'y écrit pas les 4 premiers**. Les valeurs des 4 premiers arguments sont écrits dans les registres $4, $5, $6 et $7. Si la fonction appelée contient plus de 4 arguments, les valeurs des arguments suivants (à partir du 5e) sont écrites dans la pile, à la place qui leur a été réservée.
- Quand on entre dans une fonction, le pointeur de pile $29 pointe sur la case de la pile où le premier argument devrait être (devrait puisqu'en réalité il est dans $4).

8.4) Accès à la pile

Le processeur MIPS32 ne possède pas d'instructions spécifiques à la gestion de la pile. On utilise les instructions **lw** et **sw** pour lire et écrire dans la pile. L'instruction **addiu** est utilisée pour modifier la valeur du pointeur de pile.
L'accès à la pile est toujours relatif par rapport au pointeur de pile $29:

- Pour lire dans la pile, on utilise : **lw $i, d($29)** avec **d ≥ 0** et **$i € [$1...$31]**
- Pour écrire dans la pile, on utilise : : **sw $i, d($29)** avec **d ≥ 0** et **$i € [$0...$31]**

8.5) Organisation de la pile

À l'entrée de la fonction **F()**, le pointeur de pile pointe sur la case réservée pour le premier argument de **F()**. La fonction va exécuter une séquence d'instructions nommées **prologue** qui contient les étapes suivantes :

- Allocation du contexte d'exécution de **F()** en décrémentant le pointeur de pile :
  $29 <= $29 – 4*(**nv** + **nr** + **na**)
- Écriture dans la pile des valeurs des **nr** registres persistants que **F()** va modifier en commençant par le registre $31 (à l'adresse la plus grande).
- Écriture éventuelle dans la pile des 4 premiers arguments reçus dans les registres $4 à $7. En effet, nous avons vu qu'à l'entrée de la fonction **F()**, les quatre premiers arguments sont placés dans les registres $4 à $7, mais si **F()** appelle d'autres fonctions, elle peut avoir besoin d'écraser les registres $4 à $7. Elle doit alors les écrire dans la pile dans les cases réservées par la fonction appelante de **F()**.

À la sortie de la fonction **F()**, il faut exécuter une séquence nommée **épilogue** qui sert à :

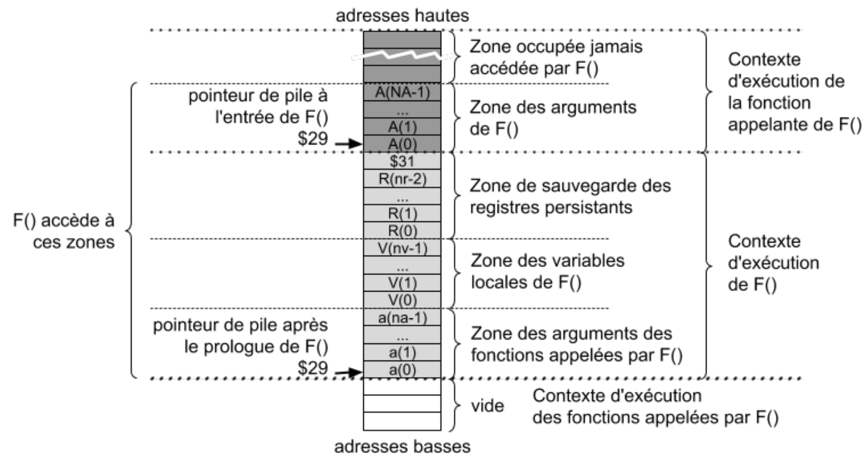
- Restaurer les valeurs des **nr** registres dont $31.
- Restaurer le pointeur de pile à sa valeur initiale $29 <= $29 + 4*(**nv** + **nr** + **na**)
- sauter à l'adresse contenue dans $31

Entre le prologue et l'épilogue se trouve le **corps de la fonction** qui réalise les calculs en utilisant les registres temporaires et les registres persistants sauvés dans le prologue ainsi que les variables locales stockées dans la pile. Elle écrit la valeur de retour dans le registre $2. Il est préférable de choisir des registres temporaires pour les calculs.

Lors de l'appel d'une fonction **g()** par la fonction **F()**, le pointeur de pile $29 n'est pas modifié par **F()**, il faut juste :

- Placer les 4 premiers arguments dans les registres $4 à $7 et les autres dans la pile au-delà de la place réservée pour les 4 premiers.
- Effectuer le branchement à la première instruction de la fonction **g()** en utilisant une instruction de type jal (jump and link) pour enregistrer l'adresse de retour dans $31.

La figure suivante représente l'état de la pile après l'exécution du prologue de la fonction **F()**. On peut voir que la zone des arguments de la fonction **F()** a bien été allouée dans la pile par la fonction appelante de la fonction **F()**.
C'est dans le prologue que la fonction **F()** alloue son propre contexte.
F() accède donc à son contexte **ET** au contexte de sa fonction appelante pour ses propres arguments.



8.6) Exemple complet

On traite ici l'exemple d'une fonction calculant la norme d'un vecteur (x,y), en supposant qu'il existe une fonction int isqrt(int x) qui retourne la racine carrée d'un nombre entier. Les coordonnées du vecteur sont des variables globales initialisées dans le segment « data ». Ceci correspond au code C ci-dessous :

```
int     x = 5 ;
int     y = 4 ;

int main()
{
    printf (« %x », norme(x,y) ); /* réalisé par l'appel système n° 1 */
    exit() ;                      /* réalisé par l'appel système n° 10 */
}

int norme (int a, int b)
{   int somme, val;       /* Variables locales */
    somme = a * a + b * b;
    val = isqrt(somme);
    return val;
}
```

Quelques remarques :

- La fonction *main* n'a pas de variables locales (nv = 0), n'utilise pas de registres persistants (nr = 1 pour $31) et appelle la fonction *norme* avec deux arguments (na = 2). Il faut donc réserver 3 cases dans la pile (1 case pour $31 et 2 cases pour les arguments de *norme*).
- **Attention, même si** la fonction *main* est **ici** une fonction spéciale qui se termine par un appel système *exit* qui met fin définitivement à l'exécution du programme. Il faut restaurer l'état des registres ou de la pile, parce que c'est ce qu'aurait fait le compilateur qui ignore que l'appel système *exit* est sans retour.
- La fonction *norme* déclare deux variables locales *somme* et *val* (nv = 2), utilise deux registres de travail temporaires $8 et $9 et pas de registre persistants (nr = 1 pour

$31) et appelle la fonction *isqrt,* qui a 1 argument. Il faudra donc réserver 4 cases dans la pile (1 pour $31, 2 pour les variables locales, 1 pour l'argument de *isqrt)*
- Les deux fonctions *isqrt* et *norme* renvoient leur résultat dans le registre $2.

Le programme assembleur est le suivant :

```
        .data

x :     .word       5
y :     .word       4

        .text

main :  addiu       $29,  $29,  -12   # décrémente pointeur de pile pour x et y
        sw          $31,  8($29)      # sauvegarde de $31
        la          $4,   x           # Ecriture 1er parametre
        lw          $4,   0($4)
        la          $5,   y           # Écriture 2e  paramètre
        lw          $5,   0($5)
        jal         norme
        or          $4    $2    $0    # récupération résultat norme dans $4
        ori         $2,   $0,   1     # code de « print_integer » dans $2
        syscall                       # affichage résultat sur la console
        ori         $2, $0, 10        # code de « exit » dans $2
        syscall                       # sortie du programme
        lw          $31, 8($29)       # restaure adresse de retour
        addiu       $29, $29,   12    # restaure l'état du poineur de pile
        jr          $31               # retour de la fonction main

norme :
# prologue nv=2 nr=1 na = 1
        addiu       $29,  $29,  -16   # décrémente pointeur de pile (2+1+1)
        sw          $31,  12($29)     # sauvegarde adresse de retour

# corps de la fonction
        mult        $4,   $4          # calcul a*a
        mflo        $4
        mult        $5,   $5          # calcul b*b
        mflo        $5
        addu        $4,   $4,   $5    # calcul somme dans $4 (1er argument)
        jal               isqrt       # appel de isqrt (résultat dans $2)

# épilogue
        lw          $31,  12($29)     # restaure adresse de retour
        addiu       $29,  $29,  16    # restaure l'état du pointeur de pile
        jr          $31               # retour de la fonction norme
```

8.7) Optimisation possible, mais non obligatoire dans le cas des fonctions terminales sans variables locales

Une fonction qui n'appelle pas d'autres fonctions est nommée fonction terminale. Si, en outre, elle n'a pas de variables locales et si elle utilise que des registres temporaires ($1 à $15 et $24, $25). Dans ce cas, **nr**=0, **nv**=0 et **na**=0.

Dans ce cas, il n'est pas utile de déplacer le pointeur de pile à l'entrée dans la fonction, ni de sauver $31. Il n'y a donc ni prologue, ni épilogue.

Aide mémoire ALMO
Jeu d'instructions MIPS

Instructions Arithmétiques/Logiques entre registres

Assembleur		Opération		Format
Add	Rd, Rs, Rt	Add overflow detection	Rd <- Rs + Rt	R
Sub	Rd, Rs, Rt	Subtract overflow detection	Rd <- Rs - Rt	R
Addu	Rd, Rs, Rt	Add no overflow	Rd <- Rs + Rt	R
Subu	Rd, Rs, Rt	Subtract no overflow	Rd <- Rs - Rt	R
Addi	Rt, Rs,I	Add Immediate overflow detection	Rt <- Rs + I	I
Addiu	Rt, Rs,I	Add Immediate no overflow	Rt <- Rs + I	I
Or	Rd, Rs, Rt	Logical Or	Rd <- Rs or Rt	R
And	Rd, Rs, Rt	Logical And	Rd <- Rs and Rt	R
Xor	Rd, Rs Rt	Logical Exclusive-Or	Rd <- Rs xor Rt	R
Nor	Rd, Rs, Rt	Logical Not Or	Rd <- Rs nor Rt	R
Ori	Rt, Rs, I	Or Immediate unsigned immediate	Rt <- Rs or I	I
Andi	Rt, Rs, I	And Immediate unsigned immediate	Rt <- Rs and I	I
Xori	Rt, Rs, I	Exclusive-Or Immediate unsigned immediate	Rt <- Rs xor I	I
Sllv	Rd, Rt, Rs	Shift Left Logical Variable 5 lsb of Rs is significant	Rd <- Rt << Rs	R
Srlv	Rd, Rt, Rs	Shift Right Logical Variable 5 lsb of Rs is significant	Rd <- Rt >> Rs	R
Srav	Rd, Rt, Rs	Shift Right Arithmetical Variable 5 lsb of Rs is significant	Rd <- Rt >>* Rs	R
Sll	Rd, Rt, sh	Shift Left Logical	Rd <- Rt << sh	R
Srl	Rd, Rt, sh	Shift Right Logical	Rd <- Rt >> sh	R
Sra	Rd, Rt, sh	Shift Right Arithmetical	Rd <- Rt >>* sh * : with sign extension	R
Lui	Rt, I	Load Upper Immediate 16 lower bits of Rt are set to zero	Rt <- I ll "0000"	I

Instructions Arithmétiques/Logiques (suite)

Assembleur		Opération		Format
Slt	Rd, Rs, Rt	Set if Less Than	Rd <- 1 if Rs<Rt else 0	R
Sltu	Rd, Rs, Rt	Set if Less Than Unsigned	Rd <- 1 if Rs<Rt else 0	R
Slti	Rt, Rs, I	Set if Less Than Immediate sign extended Immediate	Rt <- 1 if Rs < I else 0	I
Sltiu	Rt, Rs, I	Set if Less Than Immediate unsigned immediate	Rt <- 1 if Rs < I else 0	I
Mult	Rs, Rt	Multiply	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Multu	Rs, Rt	Multiply Unsigned	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Div	Rs, Rt	Divide	Rs / Rt LO <- Quotient HI <- Remainder	R
Divu	Rs, Rt	Divide Unsigned	Rs / Rt LO <- Quotient HI <- Remainder	R
Mfhi	Rd	Move From HI	Rd <- HI	R
Mflo	Rd	Move From LO	Rd <- LO	R
Mthi	Rs	Move To HI	HI <- Rs	R
Mtlo	Rs	Move To LO	LO <- Rs	R

Instructions de lecture/écriture mémoire

Assembleur		Opération		Format
Lw	Rt, I (Rs)	Load Word sign extended Immediate	Rt <- M (Rs + I)	I
Sw	Rt, I (Rs)	Store Word sign extended Immediate	M (Rs + I) <- Rt	I
Lh	Rt, I (Rs)	Load Half Word sign extended Immediate. Two bytes from storage is loaded into the 2 less significant bytes of Rt. The sign of these 2 bytes is extended on the 2 most signficant bytes.	Rt <- M (Rs + I)	I
Lhu	Rt, I (Rs)	Load Half Word Unsigned sign extended Immediate. Two bytes from storage is loaded into the 2 less significant bytes of Rt, other bytes are set to zero	Rt <- M (Rs + I)	I
Sh	Rt, I (Rs)	Store Half Word sign extended Immediate. The Two less significant bytes of Rt are stored into storage	M (Rs + I) <- Rt	I
Lb	Rt, I (Rs)	Load Byte sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt. The sign of this byte is extended on the 3 most signficant bytes.	Rt <- M (Rs + I)	I
Lbu	Rt, I (Rs)	Load Byte Unsigned sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt, other bytes are set to zero	Rt <- M (Rs + I)	I
Sb	Rt, I (Rs)	Store Byte sign extended Immediate. The less significant byte of Rt is stored into storage	M (Rs + I) <- Rt	I

Instructions de Branchement

Assembleur		Opération			Format
Beq	Rs, Rt, Label	Branch if Equal	PC <- PC+4+(I*4) PC <- PC+4	if Rs = Rt if Rs ≠ Rt	I
Bne	Rs, Rt, Label	Branch if Not Equal	PC <- PC+4+(I*4) PC <- PC+4	if Rs ≠ Rt if Rs = Rt	I
Bgez	Rs, Label	Branch if Greater or Equal Zero	PC <- PC+4+(I*4) PC <- PC+4	if Rs ≥ 0 if Rs < 0	I
Bgtz	Rs, Label	Branch if Greater Than Zero	PC <- PC+4+(I*4) PC <- PC+4	if Rs > 0 if Rs ≤ 0	I
Blez	Rs, Label	Branch if Less or Equal Zero	PC <- PC+4+(I*4) PC <- PC + 4	if Rs ≤ 0 if Rs > 0	I
Bltz	Rs, Label	Branch if Less Than Zero	PC <- PC+4+(I*4) PC <- PC+4	if Rs < 0 if Rs ≥ 0	I
Bgezal	Rs, Label	Branch if Greater or Equal Zero and link	PC <- PC+4+(I*4) PC <- PC+4 R31 <-PC+4 in both cases	if Rs ≥ 0 if Rs < 0	I
Bltzal	Rs, Label	Branch if Less Than Zero and link	PC <- PC+4+(I*4) PC <- PC+4 R31 <- PC+4 in both cases	if Rs < 0 if Rs ≥ 0	I
J	Label	Jump	PC <- PC 31:28 II I*4		J
Jal	Label	Jump and Link	R31 <- PC+4 PC <- PC 31:28 II I*4		J
Jr	Rs	Jump Register	PC <- Rs		R
Jalr	Rs	Jump and Link Register	R31 <- PC+4 PC <- Rs		R
Jalr	Rd, Rs	Jump and Link Register	Rd <- PC+4 PC <- Rs		R

Instructions Systèmes

Assembleur		Opération		Format
Rfe		Restore From Exception Privileged instruction. Restore the prevous IT mask and mode	SR <- SR 31:4 II SR 5:2	R
Break	n	Breakpoint Trap Branch to exception handler. n defines the breakpoint number	SR <- SR 31:6 II SR 3:0 II "00" PC <- "8000 0080" CR <- cause	R
Syscall		System Call Trap Branch to exception handler	SR <- SR 31:6 II SR 3:0 II "00" PC <- "8000 0080" CR <- cause	R
Mfc0	Rt, Rd	Move From Control Coprocessor Privileged Instruction . The register Rd of the Control Coprocessor is moved into the integer register Rt	Rt <- Rd	R
Mtc0	Rt, Rd	Move To Control Coprocessor Privileged Instruction . The integer register Rt is moved into the register Rd of the Control Coprocessor	Rd <- Rt	R