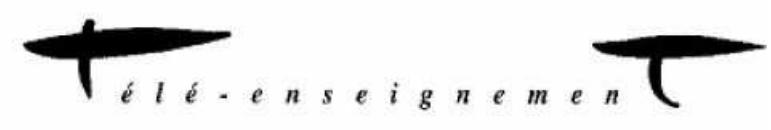


Université de Franche Comté
Centre de Télé-Enseignement Universitaire de Besançon
Licence d'Informatique
Année 2009 / 2010
Didier Teifreto
didier.teifreto@univ-fcomte.fr
18 novembre 2009 à 13:34

Architecture des Systèmes Informatiques

Programmation et Architecture des processeurs MIPS

Table des matières

1 Introduction **5**

2 La programmation en langage machine du MIPS **6**
2.1 Principe de conception d'un jeu d'instructions 7
2.2 Syntaxe des instructions MIPS 8
2.3 Codage des instructions 8
2.4 Mode d'adressage 10
2.4.1 Adressage registre 10
2.4.2 Adressage immédiat 12
2.4.3 Adressage direct 12
2.4.4 Adressage indirect 13
2.4.5 Adressage Indirect immédiat 14
2.5 Instructions arithmétiques et logiques 16
2.6 Instructions de chargement et rangement 17
2.7 Instructions de branchement 18
2.7.1 Valeur immédiate 16 bits signée 18
2.7.2 Valeur immédiate 26 bits non signée 19
2.7.3 Valeur 32 bits non signée 20
2.8 Instructions Diverses 21
2.9 Programmation en assembleur 22
2.9.1 Segments du programme 22
2.9.2 Programmation de l'alternative 24
2.9.3 Programmation des itérations 25
2.9.4 Notion de fonctions 26
2.9.5 Pile LIFO 28
2.10 Exercices 30
2.10.1 Théoriques 30
2.10.2 Programmation 32

3 Organisation du processeur MIPS **34**
3.1 Introduction 34
3.2 Conception des éléments de base 35
3.2.1 Fonction booléenne à une variable 35

3.2.2 Fonctions à deux variables 35
3.2.3 Règles de simplification 36
3.2.4 Méthodologie de conception d'un circuit 36
3.2.5 Élément du chemin de donnée du MIPS 38
3.3 Lecture Instruction 42
3.4 Instructions arithmétiques et logiques 42
3.4.1 Le banc de registre 42
3.4.2 Unité arithmétique et logique 43
3.4.3 Instructions arithmétiques et logiques trois registres 44
3.4.4 Instructions arithmétiques et logiques utilisant une valeur immédiate 44
3.5 Instructions de chargement et de rangement 45
3.6 Instructions de branchement conditionnel 46
3.7 L'assemblage du tout 46
3.8 Exercices 47
3.8.1 Théoriques 47

4 Système d'interruptions et d'entrées/sorties 49
4.1 Organisation 49
4.2 Principe 50
4.3 Gestion des exceptions 51
4.4 Gestionnaire d'interruption 52
4.5 Entrées/Sorties de caractères 52
4.6 Entrées/Sorties par interruption 53
4.7 Exercices pratiques 53

- La simplicité est l'habit de la perfection. **Wladimir Wolf-Gozin**
- La simplicité est la sophistication suprême. **Léonard de Vinci**
- Question : Comment faire pour ne pas perdre son temps ? Réponse : L'éprouver dans toute sa longueur. **Albert Camus**
- Qu'est-ce que la vie, après tout ? La vie, c'est une série d'interruptions !... Dès que quelque chose va bien, il faut changer. **Robert Hollier**
- Exception. Dites qu'elle confirme la règle. Ne vous risquez pas à expliquer comment. **Gustave Flaubert**

Bibliographie :

Sur le chapitre 2 Programmation MIPS :

1. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 3
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre A (en ligne `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`
3. **Architecture des Ordinateur : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 2.8
4. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD A Patterson J Hennessy** Chapitre 5.6, 8.1 et annexe A.10
5. **Architecture de l'ordinateur Nicolas P Carter Schaum's** Chapitre 11.

Sur le chapitre 3 Organisation MIPS :

1. **Architecture logicielles et matérielles : Cours, études de cas et exercices corigés DUNOD Ouvrage collectif** Chapitre 2 pages 23 à 45
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Annexe B.1 à B.3 et B.5
3. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 4.5
4. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 5.1 et 5.2
5. **Architecture des Ordinateur : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 3.1 à 3.5 et 3.10 à 3.13

Chapitre 1

Introduction

MIPS *Microprocessor without interlocked pipeline stages* est une architecture de microprocesseur de type jeu d'instruction réduit *RISC -Reduced Instruction-Set Computer*. Elle fut développée par la compagnie MIPS Computer Systems Inc. Les processeurs fabriqués selon cette architecture sont surtout utilisés dans les systèmes SGI `http://www.sgi.fr/index.shtml`, dans les systèmes embarqués, comme les ordinateurs de poche, les routeurs Cisco et les consoles de jeux vidéo (Nintendo 64 et Sony PlayStation, PlayStation 2 et PSP). Vers la fin des années 1990, on estimait que les processeurs dérivés de l'architecture MIPS occupaient le tiers des processeurs RISC produits.

Les premières implémentations de l'architecture MIPS étaient de 32 bits (registres et chemins de données), puis passèrent à 64 bits.

Le jeu d'instructions de base est simple, efficace ce qui facilite son apprentissage. Pour illustrer les concepts développés ici, nous utiliserons un simulateur multi-plateforme Java nommé Mars (**M**ips **a**rchitecture and **r**untine **s**imulator) en téléchargement sur `http://courses.missouristate.edu/KenVollmar/MARS/` version 3.7.

Vous trouverez d'autres informations sur la genèse de l'architecture MIPS sur cette page en Anglais : `http://en.wikipedia.org/wiki/MIPS_architecture`

Chapitre 2

La programmation en langage machine du MIPS

Le langage machine permet de programmer un ordinateur au plus bas niveau à savoir directement le microprocesseur. Pour cela nous utiliserons des *instructions* groupées dans un *jeu d'instructions*.

Le **compilateur** vérifie la syntaxe et transforme le code écrit en *langage de haut niveau* en un code en *langage assembleur* qui sera traduit ensuite en *code objet binaire* par l'**assembleur**.

L'éditeur de liens inclus les bibliothèques pour obtenir un *fichier exécutable*. En cas de dysfonctionnement, le **débuggeur symbolique** permet de recherche les erreurs logiques faites lors de conception du code de haut niveau ou du code assembleur.

La figure FIG. 2.1 présente les étapes de la production d'un programme exécutable.

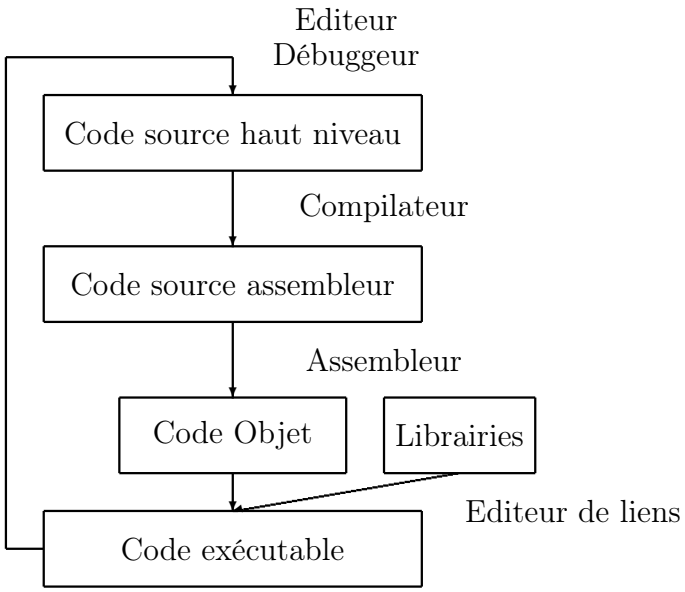


FIG. 2.1: Chaine de production

2.1 Principe de conception d'un jeu d'instructions

Les deux principes d'un ordinateur à programme enregistré sont la **représentation des instructions comme des nombres entiers** et l'utilisation d'une **mémoire modifiable contenant les programmes et les données**.

La figure FIG. 2.2 présente l'architecture simplifiée d'un ordinateur. Cette architecture est nommée de "Von Neumann".

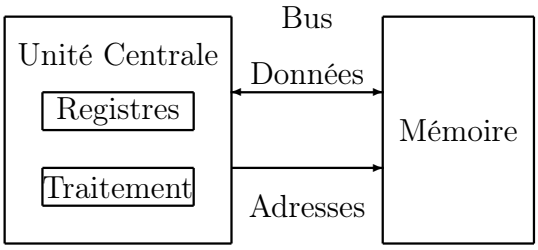


FIG. 2.2: Architecture de Von Neumann

Le choix d'un jeu d'instructions influe sur l'architecture matérielle, mais aussi sur le nombre d'instructions nécessaires pour effectuer une suite d'opération. Le jeu d'instruction conditionne le nombre de cycles d'horloge nécessaires à une instruction ainsi que la vitesse d'horloge séquencent les opérations.

Quatre règles guident les concepteurs dans la recherche d'un compromis entre simplicité et efficacité :

1. **Le plus petit et le plus rapide** Petit nombre de registres (32) et nombre d'instructions limité et petit nombre d'instruction sont des caractéristiques fondamentales de l'architecture *RISC* (acronyme de Reduce Instruction Set Computer).
2. **La simplicité favorise la régularité** La plupart des instruction doivent utiliser la même méthode de codage :
   - taille identique des instructions (32 bits),
   - trois opérandes pour toutes les instructions arithmétiques et logiques
   - taille des données identiques (32 bits)
   - utiliser une même méthode de codage proche pour toutes les autres instructions.
3. **Une bonne conception requiert des compromis** Par exemple, les constantes ne seront pas codés sur 32 bits, puisque toutes les instructions sont elle mêmes codées sur 32 bits (principe de régularité).
4. **Faire en sorte que les cas fréquents soient les plus rapides** Par exemple, les opérations sur les nombres entiers doivent être très rapides, car elles sont utilisées dans tous les types de calcul (pointeurs, caractères, réels).

2.2 Syntaxe des instructions MIPS

Forme générale d'une instruction en langage assembleur MIPS

```
Nom_Instruction Opérande 1, Opérande 2, Opérande 3
```

Nom Instruction représente le nom symbolique de l'instruction.
L'opérande représente une donnée ou le résultat de l'instruction. Les opérandes utilisent les **modes d'adressages** pour savoir comment obtenir la valeur de la donnée recherchée, ou pour savoir où ranger le résultat de l'instruction.

2.3 Codage des instructions

Les instructions sont codées dans des entiers non signés 32 bits. Les opérandes et le code opération sont codés dans des ensembles de bits placés à des endroits spécifiques de l'instruction. Par souci de simplicité, seulement trois types d'instructions existent. Ils sont présentés sur la figure Fig. 2.3 :

type R Trois opérandes registres,

type I Deux opérandes registres et une opérande immédiate sur 16 bits,

Type J Une opérande immédiate sur 26 bits.

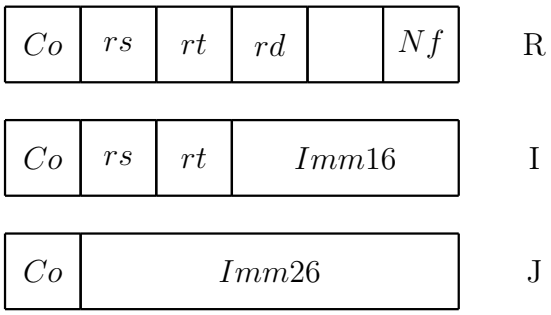


FIG. 2.3: Type d'instructions

Les champs de l'instruction sont :

Code Opération (noté Co) : codé dans les bits 26 à 31 et éventuellement un numéro de fonction bits 5 à 0 (noté Nf). Ce champs n'existent que dans le cas d'instruction à trois registres.

2 ou 3 Registres : codés dans des paquets de 5 bits.

Valeur immédiate 16 bits : Les valeurs immédiates sont donc comprises entre $-32768 \leq \text{Imm} \leq 32767$.

Valeur immédiate 26 bits : Utilisé lors des branchement. Les valeurs immédiates sont donc comprises entre $-2^{25} \leq \text{Imm26} \leq 2^{25} - 1$.

Les instructions avec trois opérandes registres sont codées au format R :

bits 31 à 26 Code opération (noté Co).

bits 25 à 21 Numéro du registre (noté rs).

bits 20 à 16 Numéro du registre (noté rt).

bits 15 à 11 Numéro du registre (noté rd).

bits 10 à 6 Valeur 0

bits 5 à 0 Numéro de fonction (noté Nf).

Les instructions avec deux opérandes registres et une opérande immédiate sont codées au format I :

bits 31 à 26 Code opération (noté Co).

bits 25 à 21 Numéro du registre (noté rs).

bits 20 à 16 Numéro du registre (noté rt).

bits 15 à 0 Valeur Immédiate 16 bits

2.4 Mode d'adressage

Les données d'un instruction peuvent se trouver à différents endroit de l'ordinateur. La figure FIG. 2.4 présente ceux ci :

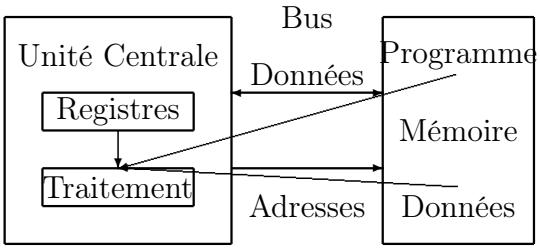


FIG. 2.4: Position des opérandes

L'opérande peut se situer :
- dans les registres,
- dans la mémoire contenant les programmes,
- dans la mémoire contenant les données du programme.

2.4.1 Adressage registre

L'opérande se situe dans un *registre* (mémoire interne au microprosseur). Une valeur utilisant le mode d'adressage registre est obtenue en utilisant la notation i avec $(0 \leq i \leq 31)$ ou un nom symbolique noté NomSymbolique. Le nom symbolique permet d'accéder au registre en utilisant sa catégorie (temporaires, arguments ...). La figure FIG. 2.6 précise l'utilisation des 32 registres de l'architecture MIPS. La figure FIG. 2.5 illustre la détermination d'une opérande registre. Dans l'architecture MIPS, le numéro d'un registre est codé sur 5 bits (car $2^5 = 32$).

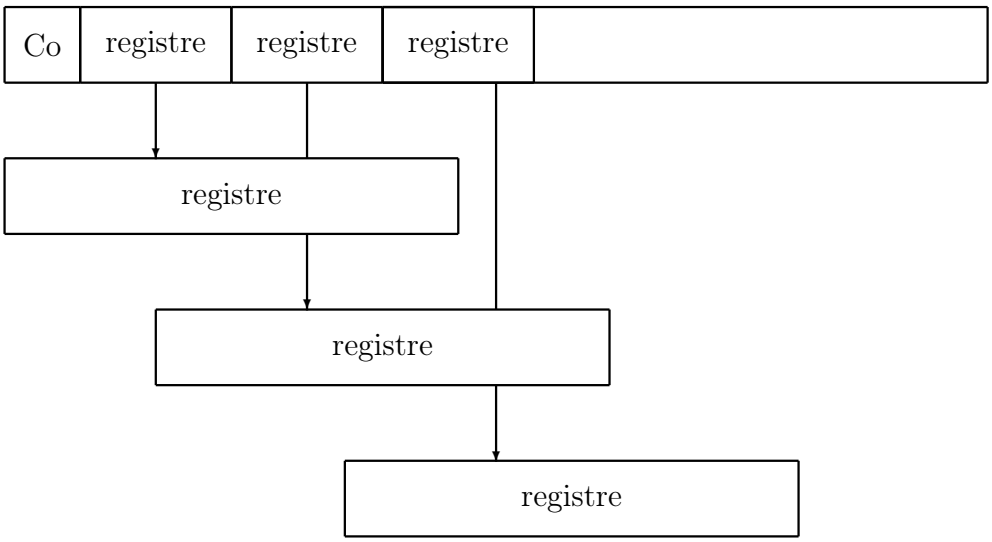


FIG. 2.5: Mode d'adressage registre

Nom symbolique	Numéro	Utilisation
$zero	$0	La valeur 0
$at	$1	réservé assembleur
$v0	$2	résultat de la fonction
$v1	$3	résultat de la fonction
$a0	$4	argument de la fonction
$a1	$5	argument de la fonction
$a2	$6	argument de la fonction
$a3	$7	argument de la fonction
$t0 à $t9 $s0 à $s7	$8 à $25	libres
$k0 et $k1	$26 à $27	OS kernel
$gp	$28	global pointer
$sp	$29	stack pointer
$fp	$30	frame pointeur
$ra	$31	adresse retour de fonction

FIG. 2.6: Numéros et noms explicites des registres

Remarques : Les noms symboliques sont préfixés avec une lettre significative : **a** pour argument, **v** pour value, **t** pour temporary, **k** pour kernel ...

2.4.2 Adressage immédiat

L'opérande est une **constante** située directement dans l'instruction. Dans le cas de l'architecture MIPS cette constante est de taille 16 bits et ne peut pas être 32 bits. Une valeur utilisant le mode d'adressage indirect est obtenue en utilisant la notation Imm16. La figure FIG. 2.7 illustre la détermination d'une opérande immédiate.

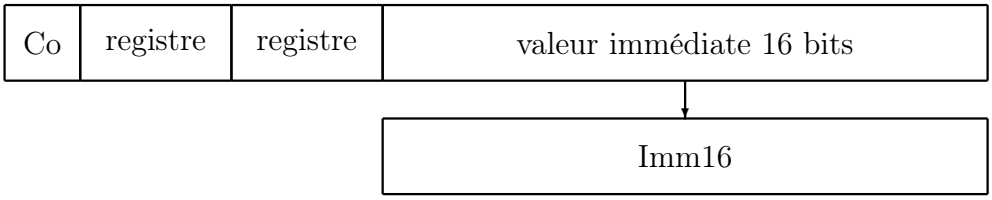


FIG. 2.7: Mode d'adressage immédiat

2.4.3 Adressage direct

L'opérande se trouve dans la mémoire de l'ordinateur pointée par la valeur de l'opérande. Le mode direct n'est pas implémenté dans l'architecture MIPS puisque les constantes 32 bits n'existent pas. Seul l'accès aux mémoires d'adresse inférieure à 0x7FFF est possible avec une instruction de la forme Imm16(zero). La figure FIG. 2.8 illustre la détermination d'une opérande directe.

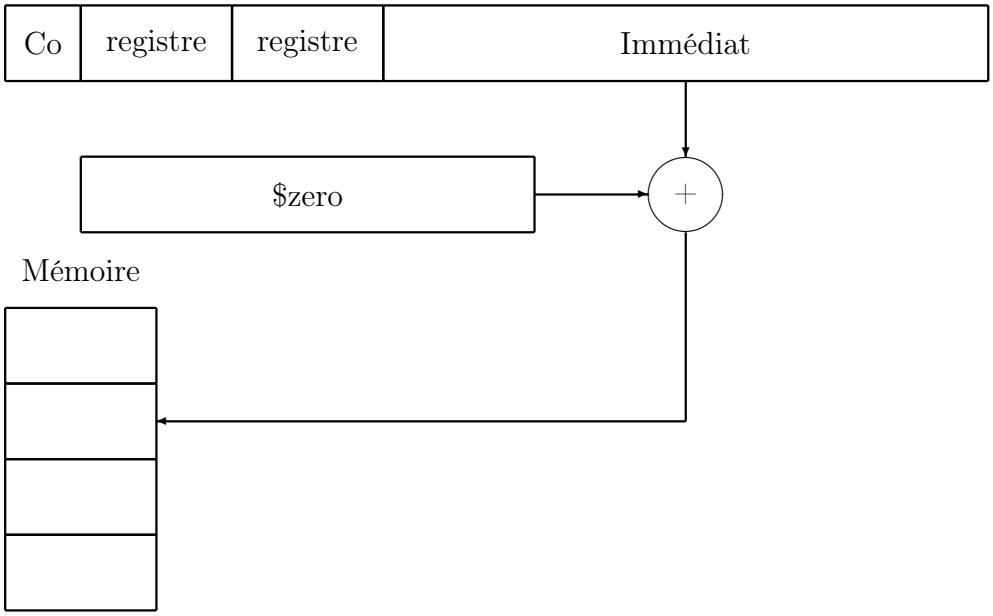


FIG. 2.8: Mode d'adressage direct

2.4.4 Adressage indirect

L'opérande se trouve dans la mémoire de l'ordinateur pointée par un registre. Une valeur utilisant le mode d'adressage indirect est obtenue en utilisant la notation 0($i) ou 0($NomSymbolique).La figure FIG. 2.9 illustre la détermination d'une opérande indirect.

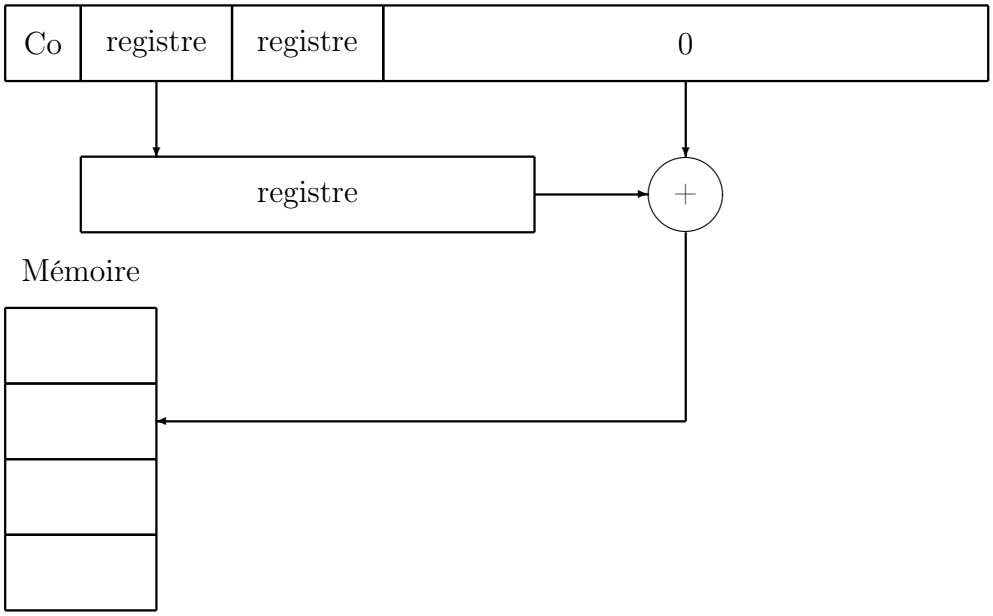


FIG. 2.9: Mode d'adressage indirect

2.4.5 Adressage Indirect immédiat

L'opérande se trouve dans la mémoire de l'ordinateur pointée par un registre auquel nous ajoutons une valeur immédiate 16 bits. Une valeur utilisant le mode d'adressage indirect immédiat est obtenue en utilisant la notation Imm($i) ou Imm($NomSymbolique). La figure FIG. 2.10 illustre la détermination d'une opérande indirect immédiat.

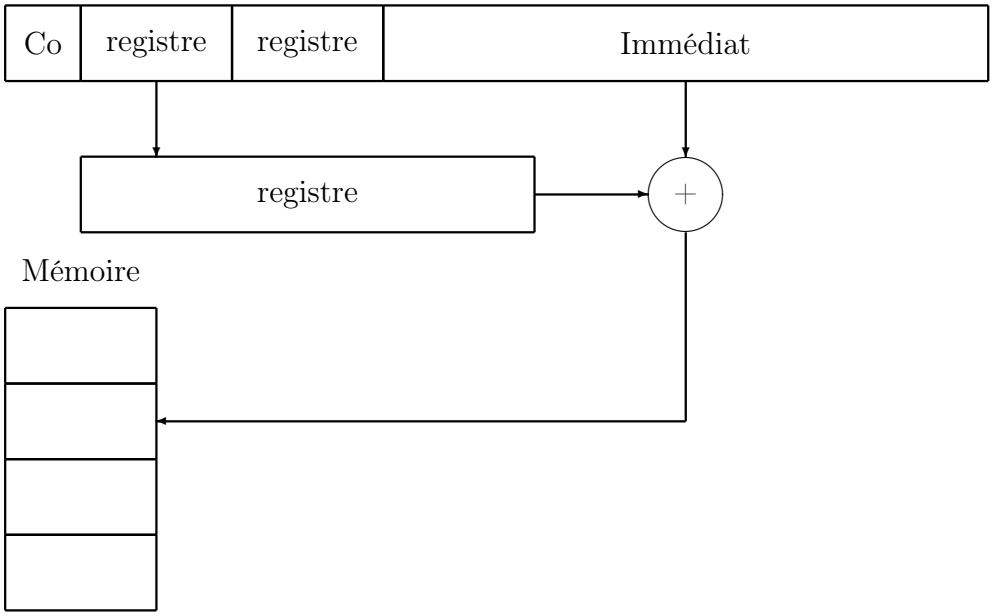


FIG. 2.10: Mode d'adressage indirect immédiat

La figure FIG. 2.11 résume les modes d'adressage ayant accès à la mémoire de donnée.

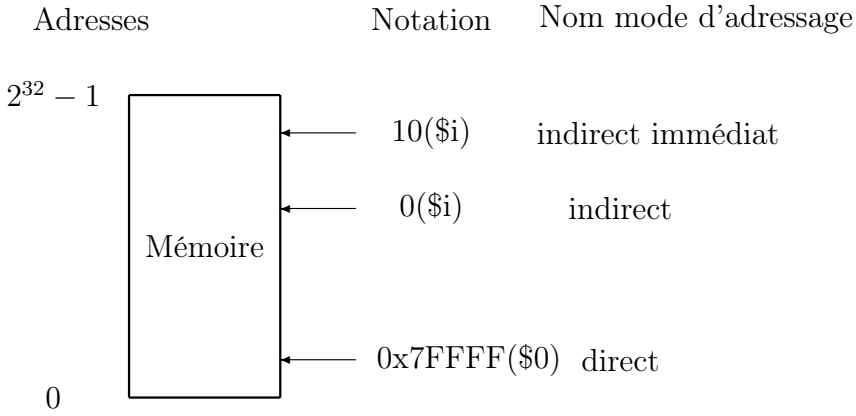


FIG. 2.11: Utilisation de l'adressage mémoire

La figure FIG. 2.12 résumé les modes d'adressage :

Mode d'adressage	Notation	Valeur
Registre	$i	$0 \leq \$i \leq 31$
Immédiat	Imm16	$-2^{15} \leq Imm16 \leq 2^{15} - 1$
Indirect Imm16	Imm16(i)	adresse = Imm16 + valeur registre $i

FIG. 2.12: Notation des modes d'adressage

Nous allons étudiés les principales instructions du jeu d'instruction MIPS. la liste exhaustive des instructions et des pseudo instructions se trouve dans l'aide du simulateur MARS.

2.5 Instructions arithmétiques et logiques

Les deux possibles sont :

1. `Opération rd, rs, rt`
   effectue l'instruction rd ← rs Opération rt
2. `Opérationi rt, rs, Imm16`
   effectue l'instruction rt ← rs Opérationi Imm16

Rôle	Syntaxe	Opération	Codage
addition	add rd,rs,rt	rd ← rs + rt	Co=0 Nf= 0x20
addition immédiate	addi rt,rs,Imm16	rt ← rs + Imm16	Co=8
soustraction	sub rd,rs,rt	rd ← rs - rt	Co=0 Nf=0x22
multiplication	mul rd,rs,rt	rd ← rs × rt	Co=0x1C
et bit à bit	and rd,rs,rt	rd ← rs et rt	Co=0 Nf=0x24
et bit à bit immédiat	addi rt,rs,Imm16	rt ← rs et Imm16	Co=0xC
ou bit à bit	or rd,rs,rt	rd ← rs ou rt	. Co=0 Nf=0x25
ou bit à bit immédiat	ori rt,rs,Imm16	rt ← rs ou Imm16	Co=0xD
décalage à gauche	sll rt,rs,shamt	rt ← rs $<<$ sham	Co=0 Nf=0
décalage à gauche	sllv rd,rs,rt	rd ← rs $<<$ rt	Co=0 Nf=4
décalage droite arithm	sra rt,rs,shamt	rt ← rt $>>$ shamt	Co=0 Nf=3
décalage droite log	srl rt,rs,shamt	rt ← rs $>>>$ shamt	Co=0 Nf=2
charger demi mot poids fort	lui rs,Imm16	rs ← Imm16 $<<$ 16	A chercher

FIG. 2.13: Liste partielle des instructions arithmétiques et logiques

Le jeu d'instruction est **non destructif** car la syntaxe à trois opérandes préserve éventuellement les données en permettant le stockage du résultat dans une autre opérande.

Exemple :

```
addi $t0,$zero,0
add $t1,$zero,$zero
add $t2,$at,$t0
```

Remarques :

- Les **pseudo-instructions** permettant de manipuler des valeurs immédiates codées sur 32 bits. Celles-ci utilisent le registre nommé $at qui est réservé à l'assembleur et une séquence d'instructions pour réaliser cette instruction.
- Une constante 32 bits d'une instruction de type I est décomposée en deux constantes 16 bits. Pour affecter une constante 32 bits nous utiliserons une pseudo instruction. Celle-ci est décomposée en une suite d'instruction ou nous trouvons cette constante divisée en deux parties.

  ```
  addi $t0,$zero,0x1234ABCD
  ```

  est traduite par l'assembleur en la suite d'instructions suivante

  ```
  lui $at,$zero,0x1234
  ori $at,$at,0xABCD
  add $8,$0,$at
  ```

2.6 Instructions de chargement et rangement

Le jeu d'instruction du MIPS est **chargement/rangement** car toutes les données manipulées par les instructions arithmétiques et logiques doivent être placées dans les registres de l'architecture. Les instructions de chargement et de rangement permettent d'effectuer les transferts entre la mémoire centrale et les registres.

Rôle	Syntaxe	Opération	Codage
chargement 32 bits	lw rt,Imm16(rs)	rt ← [Imm16 + rs]	Co=0x23
chargement 16 bits	lh rt,Imm16(rs)	rt ← [Imm16 + rs]	Co=0x21
chargement 8 bits	lb rt,Imm16(rs)	rt ← [Imm16 + rs]	Co=0x20
rangement 32 bits	sw rt,Imm16(rs)	[Imm16+rs] ← rt	Co=0x2B
rangement 16 bits	sh rt,Imm16(rs)	[Imm16+rs] ← rt	Co=0x29
rangement 8 bits	sb rt,Imm16(rs)	[Imm16+rs] ← rt	Co=0x28

FIG. 2.14: Liste partielle des instructions chargement/rangement

Exemple :

```
lw $t0,0($sp)
add $t0,$t0,$t1
sw $t0,0($sp)
```

Remarques :

- Les instructions manipulant des données 32 bits doivent accéder à des adresses mémoires alignées sur des mots de 32 bits (c'est à dire divisibles par 4, car la mémoire physique contient des données 8 bits).

– La pseudo-instruction **la $i,adresse** charge l'adresse 32 bits dans le registre i. Les données d'un programme MIPS sont stockées à partir de l'adresse 0x10010000. Si l'adresse nommé *variable* se trouve à l'adresse 0x10010100, la pseudo instruction **la $t0,variable** est traduite par l'assembleur en la suite d'instructions suivante
  lui $at,$zero,0x1001
  ori $8,$at,0x0100

2.7 Instructions de branchement

Les instructions de branchement rendent non séquentiel le déroulement d'un programme. Elles permettent de se rendre à une **instruction cible** autre que l'instruction suivante du programme. La taille des instructions étant de 32 bits, leurs adresses sont divisibles par 4 ce qui signifie que les deux bits de poids faibles *LSB* sont égaux à 0 et ne sont pas représentés.

L'adresse de instruction cible peut être obtenue de plusieurs façon que nous allons étudier.

2.7.1 Valeur immédiate 16 bits signée

Avec le décalage de deux bits vers la gauche de la valeur signée 16 bits (2 bits LSB à 0), les instructions cibles se situent donc autour de l'instruction suivant le branchement dans la limite de -2^{17} à $2^{17}-4$, ce qui correspond à $\pm$128kilo octets. Ce mode d'adressage est nommé **relatif au compteur de programme**. Dans ce cas les deux registres sont comparés pour prendre la décision de branchement. La figure Fig. 2.15 présente ce mode d'adressage.

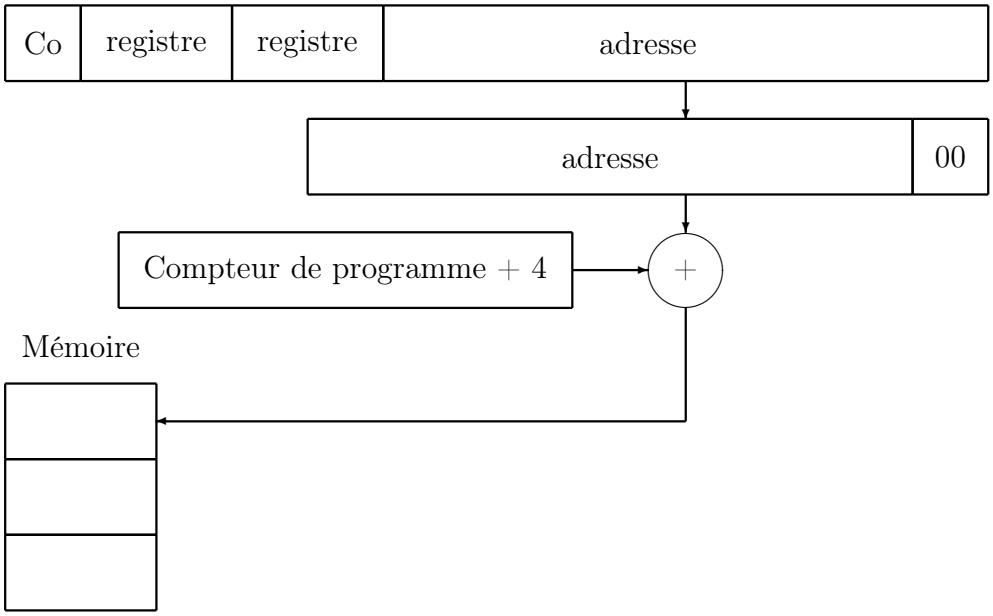


FIG. 2.15: Mode d'adressage relatif au compteur de programme

2.7.2 Valeur immédiate 26 bits non signée

Avec le décalage de deux bits vers la gauche de la valeur non signée 26 bits (2 bits LSB à 0), concaténer avec les quatre bits de poids fort du **compteur de programme** les instructions cibles se situent dans $\frac{1}{16}$ de la mémoire MIPS. Le compteur de programme est un registre du MIPS indiquant l'instruction en cours de traitement. Ce mode d'adressage est nommé **pseudo direct**. Dans ce cas le branchement est inconditionnel (toujours effectué). La figure FIG. 2.16 présente ce mode d'adressage.

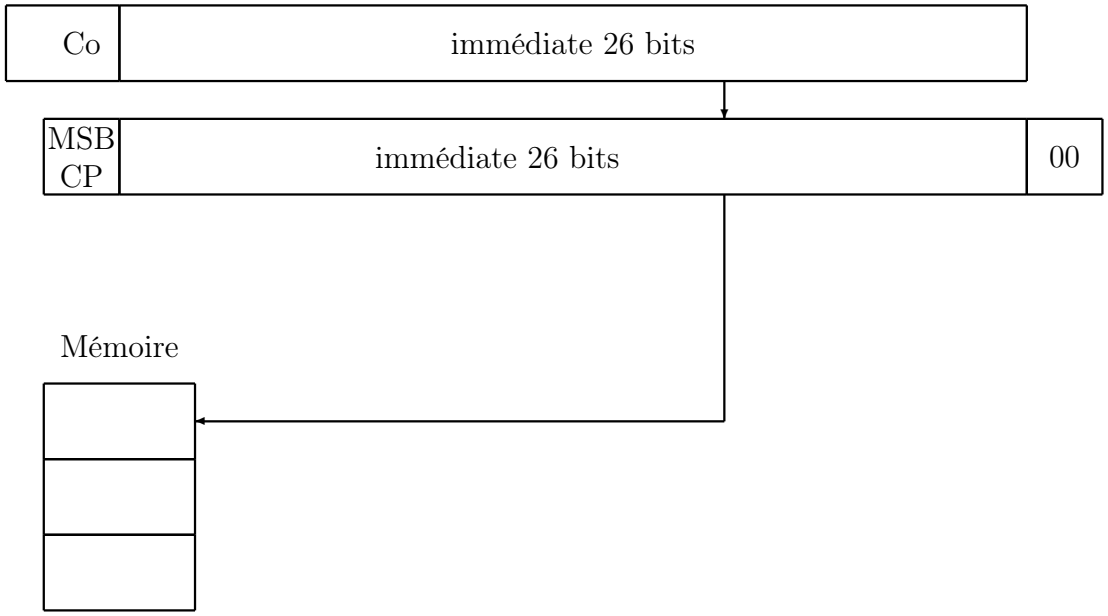


FIG. 2.16: Mode d'adressage pseudo direct

2.7.3 Valeur 32 bits non signée

L'adresse de la cible est codée dans un registre et l'instruction cible se trouve n'importe où en mémoire.

La figure FLISTEBRANCH présente les principales instruction de branchement.

Rôle	Syntaxe	Opération	Codage
branchement si égal	beq rs,rt,Imm16	si rs=rt alors cp ← (cp+4)+Imm16<<2	Co=4
branchement si non zero	bne rs,rt,Imm16	si rs ≠ rt alors cp ← (cp+4)+(Imm16<<2)	Co=5
branchement si positif	bgtz rs,Imm16	si $rs > 0$ alors cp ← (cp+4)+(Imm16<<2)	$Co = 7$
branchement si négatif	blez rs,Imm16	si rs ≤ 0 alors cp ← (cp+4)+(Imm16<<2)	Co=6
saut 26 bits	j Imm26	alors cp ← cp & (0xF0000000) | (Imm26<<2)	Co=2
saut 32 bits	jr rs	cp ← $rs	Co=0 et Nf=8
appel de fonction	jal Imm26	cp ← cp & (0xF0000000) | (Imm26<<2) puis $ra ← cp+4	Co=3

FIG. 2.17: Liste partielle des instructions de branchement

Remarques :
- Il existe des pseudo-instructions de branchement permettant de faire une comparaison par rapport à une valeur immédiate.
  La pseudo instruction bne **$t0,100,etiquette** est traduite par l'assembleur en la suite d'instructions suivante :
  addi $at,$zero,100
  bne $8,$at,etiquette
- L'instruction **jal** permet d'appeler des fonctions en stockant l'adresse de l'instruction suivante celle ayant provoqué le branchement.

2.8 Instructions Diverses

Rôle	Syntaxe	Opération	Codage
No opération	nop		0
Appel Système	syscall	Voir ci dessous	Co=0 et Nf=0xC

FIG. 2.18: Liste des instructions diverses

Les appels système permettent de créer des liens entre le microprocesseur MIPS et l'extérieur. Le système est accessible via une *API* simplifiée implémentant les fonctions

suivantes :

Nom	vo	Arguments	Résultat
Afficher entier	1	$a0	
Afficher chaine	4	$a0=adresse	
Lire entier	5		$v0
Lire chaine	8	$a0=adresse $a1=taille	
Afficher caractère	11	a0	
Lire caractère	12		$v0
Fin programme	10		

FIG. 2.19: Liste partielle des appels système

Exemple :
addi $a0,$zero,10 # mise en place de l'argument
addi $v0,$zero,1 # mise en place du numéro de l'appel système
syscall # Affiche la valeur 10
Remarque : L'aide du simulateur MARS donne la liste complète des fonctions de l'API.

2.9 Programmation en assembleur

Dans cette partie nous allons étudier les directives de l'assembleur. Une directive n'est pas une instruction, ni une pseudo instruction. Ce sont des indications données a l'assembleur pour celui-ci puisse réaliser son travail correctement.

Forme générale d'une instruction en langage assembleur MIPS

```
etiquette: Nom_Instruction Opérande 1, Opérande 2, Opérande 3 \#Commentaire
```

L'*etiquette* est un nom symbolique qui facilite la programmation. Il associe un nom à la ligne du programme assembleur. Ceci évite au programmeur de manipuler les adresses mémoire codées en hexadécimale.

Commentaire est une information non assemblée. Tout ce qui suit le caractère # est ignoré par l'assembleur.

2.9.1 Segments du programme

Comme dans un algorithme et dans tous les langages de programmation, un programme assembleur contient une **partie déclaration des données** et une partie contenant **les instructions**. Le LISTING 2.1présente les deux segments.

Listing 2.1: Fichier assembleur minimal

```
.data
  # Le segment données
# ...
.text
# Le segment text -- instructions du prog
#Le point d'entrée du programme
main:
  # ....
  #  Fin du programme appel système
  addi $v0, $zero, 10  # Syscall 10 fin
  syscall              # Sortir maintenant
```

Comme en Java le symbole *main* est les point d'entrée du programme.

Le symbole # est un commentaire. Tout ce qui est après sera ignoré par l'assembleur.

La figure Fig. 2.20 spécifie les types de données de l'assembleur MIPS.

Contenu	**Type**	**Exemple**
. Chaine	.asciiz str	msg : .asciiz "Bonjour"
Octet	.byte b1,..bn	b : .byte 0x12
Réel 64 bits	.double d1,...,dn	d : .double 1.5
Réel 32 bits	.float f1,...fn	f : .float 1.5
Entier 32 bits	.word i1,...in	w : .word 0xFF
Espace vide	.space n	.space 0xFF
Alignement	.align n	.align 2

Fig. 2.20: Liste des types définis par l'assembleur

Remarques :

- Comme en Java le préfixe 0x représente un nombre hexadécimal.
- La directive **.align** permet de faire commencer la donnée suivante à une adresse divisible par 2^n (indispensable pour utiliser les instructions de chargement/rangement 32 bits qui doit pointer sur une adresse ayant les deux bits *LSB* à 0).
- Une chaîne de caractères est terminée par le caractère de code ascii 0 (non pas le code ascii de 0 !).
- Le caractère $\backslash n$ dans une chaîne de caractères permet d'effectuer un passage à la ligne suivante.

Voici un premier programme : Le classique **Hello World**

Listing 2.2: Hello World

```
.data
  Msg:    .asciiz "Hello World \n"
.text
main:
```

```
#Affichage de la chaine de caractères
addi $v0, $zero, 4     # Sycall 4 afficher une chaine
la $a0, Msg            # Adresse de la chaine
syscall                # Afficher maintenant
# Fin
addi $v0, $zero, 10
syscall
```

Une **étiquette** ou **label** est une information symbolique désignant une ligne du programme (plus simple à manipuler qu'une adresse hexadécimale). Ici *Msg* et *main* sont deux étiquettes. L'étiquette *Msg* désigne une chaîne de caractères et *main* le début des instructions du programme.De façon classique les données débutent à l'adresse 0x1001000 et les instructions débutent à l'adresse 0x00400000.

2.9.2 Programmation de l'alternative

L'instruction algorithmique classique est le **Si** :

Algorithme 1 Alternative

1: **Début**
2: **Si** (condition=VRAI) **Alors**
3: instruction(s)
4: **Sinon**
5: instruction(s)
6: **FinSi**
7: **Fin**

Se traduit en assembleur MIPS par la suite d'instructions :

#instruction qui modifie un registre
b.. sinon # test du registre
alors :
#instruction(s)
j finsi
sinon :
#instruction(s)
finsi

Exemple permettant de tester l'égalité de deux nombres :

Listing 2.3: Programmation de l'alternative

```
.data
  Lecture: .asciiz "Entrez un nombre ?\n"
  Egaux:   .asciiz "Nombres egaux \n"
```

```
  Diff:     .asciiz "Nombre différents \n"
.text
main:
  addi $v0, $zero, 4    # Afficher chaine
  la $a0,Lecture
  syscall
  addi $v0, $zero, 5   # Lire un entier
  syscall
  add $t0, $v0, $zero
  addi $v0, $zero, 4    # Afficher chaine
  la $a0,Lecture
  syscall
  addi $v0, $zero, 5   # LIre entier
  syscall
  bne $t0, $v0, Sinon
Alors:
  la $a0,Egaux
  j FinSi
Sinon:
  la $a0,Diff
FinSi:
  add $v0, $zero, 4    # Afficher chaine
  syscall
  # -- exit
  addi $v0, $zero, 10
  syscall
```

2.9.3 Programmation des itérations

L'instruction algorithmique la plus générale est l'itération nommé **Tant Que** :

Algorithme 2 itération Tant Que

1: **Début**
2: **TantQue** (condition=VRAI) **Faire**
3: instruction(s)
4: **Fin TantQue**
5: **Fin**

Se traduit en assembleur MIPS par la suite d'instructions :

TantQue :
#modifier registre
b.. FinTantQue
Faire :
#instruction(s)
j TantQue

FinTantQue :
Exemple de Calcul de $n!$

Listing 2.4: Factorielle n

```
.data
  Lecture: .asciiz "Entrez un nombre ?\n"
  Fact:  .asciiz "Fact("
  Egal:     .asciiz ")="
.text
main:
  addi $v0, $zero, 4    # Afficher chaine
  la $a0,Lecture
  syscall
  addi $v0, $zero, 5   # Lire entier
  syscall
  add $t1, $v0 , $zero # Garder la valeur lue pour l'afficher
  addi $t0, $zero, 1
TantQue:
  beq $v0,$zero, FinTantQue
  mul $t0,$t0, $v0
  addi $v0, $v0, -1
  j TantQue
FinTantQue:
  addi $v0, $zero, 4    # Afficher chaine
  la $a0, Fact
  syscall
  addi $v0, $zero, 1    # Afficher entier
  add $a0, $t1, $zero
  syscall
  addi $v0, $zero, 4    # Afficher chaine
  la $a0, Egal
  syscall
  addi $v0, $zero, 1    # Afficher entier
  add $a0, $t0, $zero
  syscall
  # Fin
  addi $v0, $zero, 10
  syscall
```

2.9.4 Notion de fonctions

Une fonction contient un bloc de code réutilisable. Elle possède des arguments et elle renvoie une valeur scalaire.

Lors de l'appel d'une fonction, le programme doit réaliser les opérations suivantes :

1. Placer les paramètres d'entrée dans les arguments de la fonction.
2. Sauvegarder l'adresse de l'instruction suivante ayant appelé la fonction (pour pouvoir y revenir ensuite).

3. Exécuter les instructions de la fonction.
4. Revenir à l'instruction suivante du programme ayant appelé la fonction

Exemple algorithmique de fonction et de l'appel :

Algorithme 3 Fonction

Sorties: res entier

```
1: Fonction NomFonction(liste param formels :entier) :entier
2:    Début
3:    instruction(s)
4:    return ...
5:    Fin fonction
6: Début
7: res = NomFonction(liste paramètre réels)
8: Fin
```

En assembleur MIPS, les paramètres sont placés dans les registres $a0 à $a3. Le résultat de la fonction est placé dans le registre $v0 et $v1

```
#mise en place des paramêtres
add $a0, ....
add $a1,
...
#Appel de la fonction
jal NomFonction #aller a fonction mettre l'adresse inst suivante dans $ra
# $v0 contient le résultat
....

NomFonction :
#instructions de la fonction
...
add $v0, ... #placer le résultat dans $v0
jr $ra # aller à l'adresse contenue dans le registre $ra
```

Remarques : En assembleur les étiquettes doivent être unique dans la portée de la fonction. Pour cela, les étiquettes seront préfixées avec le nom de la fonction. Si la fonction s'appelle **factorielle** et que nous désirons appeler l'étiquette **finsi :**, l'étiquette du programme sera nommée **factorielle_finsi :** ou **factorielleFinsi :**

Listing 2.5: n ! avec une fonction

```
.data
  Lecture: .asciiz "Entrez un nombre ?\n"
  MFact:   .asciiz "Fact("
  Egal: .asciiz ")="
.text
```

```
main:
  addi $v0, $zero, 4    # Afficher chaine
  la $a0,Lecture
  syscall
  addi $v0, $zero, 5   # Lire entier
  syscall
  add $a0, $v0, $zero  # Argument de fact dans $a0
  jal Fact
  add $t1, $v0, $zero  # Sauvergarde $a0 et $v0
  add $t0, $a0, $zero
  addi $v0, $zero, 4   # Afficher chaine
  la $a0, MFact
  syscall
  addi $v0, $zero, 1   # Afficher entier
  add $a0, $t0, $zero
  syscall
  addi $v0, $zero, 4   # Afficher chaine
  la $a0, Egal
  syscall
  addi $v0, $zero, 1   # Afficher entier
  add $a0, $t1, $zero
  syscall
  # Fin
  addi $v0, $zero, 10
  syscall
#Fonction Fact Paramétre entrée dans $a0, retour dans $v0
Fact:
  addi $v0, $zero, 1   # Fact(0)=1
Fact_TantQue:
  beq $a0, $zero, Fact_FinTantQue
  mul $v0, $v0, $a0
  addi $a0, $a0, -1
  j Fact_TantQue
Fact_FinTantQue:
  jr $ra # retour au programme appelant
```

2.9.5 Pile LIFO

Une pile est une structure contenant des données ordonnées. La première donnée entrée sera la dernière à en sortir(pile de type *LIFO - Last In Fisrt Out*).

La pile est utilisées lors des appels imbriqués de fonctions, pour stocker les valeurs du registre ra et des éventuels paramètres(si leur nombre est supérieur à 4). Le registre $sp nommé **pointeur de pile** permet d'accéder à cette structure de données.

Lors du rangement d'une donnée 32 bits dans la pile, le registre $sp doit être diminué de 4 (4 octets). Lors de la récupération d'une donnée 32 bits dans la pile, le registre $sp doit être augmenté de 4.

Pour accéder à la pile, nous utiliserons les séquences d'instructions :

– Pour ranger une donnée 32 bits dans la pile :

sw ... , 0($sp)
addi $sp, $sp, -4

– Pour récupérer une donnée 32 bits dans la pile :

addi $sp ,$sp, 4
lw ... , 0($sp)

Exemple algorithmique d'une fonction récursive :

Algorithme 4 Fonction recursive

```
1: FONCTION FonctRec(entier n) :entier
2:    DÉBUT FONCTION
3: Si n ≠ 0 Alors
4:       Afficher n
5:       FonctionRec n-1
6: FinSi
7:    FIN FONCTION
8: DÉBUT
9: FonctRec(10)
10: FIN
```

En assembleur, nous obtenons le code suivant :

Listing 2.6: Fonction récursive

```
.data
  MsgProcRec: .asciiz "Valeur du param fonct recursive :  "
.text
main:
  addi $v0, $zero, 4    # Afficher chaine
  la $a0,MsgProcRec
  syscall
  addi $a0,$zero,10
  jal FonctionRec
  # Fin
  addi $v0, $zero, 10
  syscall
#Fonction récursive
FonctionRec:
  #sauvegarde des registres modifiés
  sw  $ra, 0($sp)     # Stocker dans la pile
  addi $sp, $sp, -4   # Modifier stack pointer
  sw  $v0, 0($sp)     # Stocker dans la pile
  addi $sp, $sp, -4   # Modifier stack pointer
  # Aller vers la fin
  addi $a0, $a0, -1
  # Afficher le parametre
  add $v0, $zero, 1   # Afficher entier
  syscall
  beq $a0, $zero, FinFonction
```

```
  #Appel recursif
  jal FonctionRec
FinFonction:
  #restitution des registres modifiés
  addi $sp, $sp, 4  # Modifier stack pointer
  lw   $v0, 0($sp)  # Restituer depuis la pile
  addi $sp, $sp, 4 # Modifier stack pointer
  lw   $ra, 0($sp) # Restituer depuis la pile
  jr $ra
```

Remarques : Les fonctions doivent sauvegarder les registres qu'elles modifient. En effet, ces registres peuvent être utilisés par le programme ayant appelé la fonction. Les fonctions débuteront par un ensemble de sauvegarde dans la pile, et se termineront par un ensemble de restitution depuis la pile. La sauvegarde des paramètres n'est pas indispensable puisqu'ils sont normalement passés par valeur.

2.10 Exercices

2.10.1 Théoriques

Exercice 1 : Code instruction

Donner les valeurs 32 bits représentant les instructions MIPS :

Question 1 : add $t3, $t2, $t0

Question 2 : addi $t3, $t2,-1

Question 3 : bne **$t2,$zero, instruction précédente**

Remarque :

- La valeur immédiate représente l'écart entre l'adresse de l'instruction suivant le branchement et l'adresse de l'instruction cible.

Exercice 2 : Instruction Vax et x86 et algorithmique

Le jeu d'instructions réduit de l'architecture MIPS est capable d'émuler des instructions plus complexes d'un jeu d'instructions complet comme celui du VAX ou du x86 et même les instructions algorithmique . Quelle est l'instruction ou la suite d'instructions MIPS qui réalisent la même opération, sans modifier d'autre registre que ceux de l'instruction. Vous pouvez seulement modifier le registre $at.

Question 1 : L'instruction Vax **decl $8** effectue $8 ← $8 - 1

Question 2 : L'instruction Vax **clrl $8** effectue $8 ← 0

Question 3 : L'instructio Vax **clrl x** effectue mémoire x ← 0

Question 4 : L'instruction **x86 add x, $8** effectue Mémoire x ← Mémoire x + $8
Remarque :
- Vous utiliserez la pile pour sauvegarder le registre modifié.

Question 5 : L'instruction Vax **aoblss $8, $9, étiquette** effectue $8 ← $8 +1 puis si($8 < $9) aller à étiquette.

Question 6 : L'instruction algorithmique c ← a + b

Question 7 : L'instruction algorithmique c ← a + b puis c ← $\frac{c}{2}$

Exercice 3 :

Expliquez pourquoi l'instruction **beq reg,Imm16,etiquette** n'existe pas dans le jeu d'instruction MIPS, mais est une pseudo-instruction.

Exercice 4 : Algorithmique

Question 1 : Initialisation d'une zone mémoire Donnez l'algorithme d'une fonction nommée *InitMem* permettant d'initialiser une zone mémoire nommé **mem** avec une valeur entière nommée **val**. La taille de cette zone mémoire sera exprimée en nombre de valeurs entières à initialiser et placée dans le paramètre **taille** . Vous utiliserez la notation [] pour accéder à un entier du tableau. Le premier entier est positionné à l'indice 0.

```
Fonction InitMem(entier mem[],entier taille, entier val):
```

Question 2 : Copie zone mémoire Donnez l'algorithme d'une fonction nommée *CopieMem* permettant de copier une zone mémoire nommée **src** dans une autre nommée **dest**.

```
Fonction CopieMem(entier src[],entier dest[], entier taille):
```

Question 3 : Copie d'une chaîne de caractères Donnez l'algorithme d'une fonction nommée *CopieChaine* permettant de copier une tableau de caractères nommé **src** terminé par le caractère de code ASCII zéro dans une autre nommé **dest** et qui retourne le nombre de caractères copiés.

```
Fonction CopieChaine(caractere src[],caractere dest[]): entier
```

Question 4 : Donnez l'algorithme d'une fonction nommée *saxpy_ version1* (Scalar A X Plus Y) qui prend en argument les adresses et la taille des tableaux x et y, la constante a et qui effectue pour chaque valeur du tableau l'opération suivante : x[i] ← a × x[i] + y[i]

```
Fonction saxpy(entier taille, entier a, entier x[],entier y[]):
```

2.10.2 Programmation

Pour réaliser vous devez installer le simulateur MARS (Mips Architecture Runtime Simulator). La documentation de la partie Help de ce simulateur est très complète. Vous trouverez les explications concernant le fonctionnement de l'IDE, les instructions et pseudo-instructions, les appels systèmes...

Exercice 5 : Prise en main de l'environnement

Question 1 : Testez les programmes du cours.

Question 2 : Testez le codage des trois instructions.

Question 3 : Testez le code permettant de simuler d'autres instructions.

Exercice 6 : Zone mémoire

Question 1 : Initialisation zone mémoire Donnez le code Java d'une fonction nommée **InitMem** permettant d'initialiser une zone mémoire.

```
# $a0 adresse zone mémoire
# $a1 taille de le zone mémoire
# $a2 valeur à placé en mémoire
InitMem :
```

Question 2 : Copie d'une zone mémoire Donnez le code Java d'une fonction nommée **CopieMem** permettant de copier une zone mémoire dans une autre.

```
# $a0 adresse zone mémoire src
# $a1 adresse zone mémoire dest
# $a2 taille de le zone mémoire
CopieMem :
```

Question 3 : Donnez le code Java d'une fonction nommée *CopieChaine* permettant de copier une chaîne de caractères terminée par le caractère de code ASCII zéro dans une autre et qui retourne le nombre de caractères copiés. Vous utiliserez les instructions de transfert d'octets **lb** et **sb**.

```
# $a0 adresse chaine source
# $a1 adresse chaine destination
# $v0 retourne le nombre de caractère copié
CopieChaine :
```

Question 4 : Donnez le code Java d'une fonction nommée *saxpy_version1* qui prend en argument les adresses et la taille des tableaux x et y, la constante a.

```
# $a0 adresse x
# $a1 adresse y
# $a2 taille du tableau
# $a3 nombre a
saxpy_version1 :
```

Exercice 7 : Affichage et lecture en hexadécimal

Question 1 : Donnez le code Mips des fonctions **digit** et **forDigit** de conversion d'un digit en un caractère et inversement. Elle seront simplifiées pour la base hexadécimale.

```
# $a0 digit à convertir
# $v0 retourne le caractere
digit :
...
# $a0 caractère à convertir
# $v0 retourne le digit correspondant digit ou -1
fordigit :
...
```

Question 2 : Donnez le code MIPS des fonctions nommées **toHexString** et **valueOf** de conversion d'un nombre entier 32 bits en une chaîne de caractères et inversement. La fonction **valueOf** n'effectuera pas le traitement des erreurs.

```
# $a0 entier à convertir
# $a1 adresse de la chaine résultat
toHexString :
...
# $a0 adresse de la chaine à convertir
# $v0 retourne l'entier résultat
valueOf :
```

Chapitre 3

Organisation du processeur MIPS

3.1 Introduction

Dans ce chapitre, nous allons concevoir la mise l'implémentation matérielle d'une architecture permettant de traiter le minimum d'instructions nécessaires à tous les programmes. Ce minimum contient les instructions suivantes :

Instructions de chargement rangement : lw et sw.

Instructions arithmétiques et logiques : add, sub, and, or, addi, ori, andi.

Instruction de branchement : instruction beq, bne.

Pour traiter une instruction, le microprocesseur doit effectuer les opérations suivantes :

1. Envoyer la valeur d'un registre nommé **compteur de programme (cp)** qui contient l'instruction à exécuter à la mémoire et ensuite calculer la position de l'instruction suivante qui se trouve quatre octet plus loin (codage des instructions sur 32 bits)
2. Charger depuis la mémoire contenant les instructions la valeur de l'entier 32 bits correspondant à l'instruction à traiter.
3. Lire la valeur de un (rs) ou deux (rs et rt) registres opérandes.
4. Ensuite en fonction de la nature de l'instruction :

   Instruction arithmétiques et logiques : Calculer le résultat de l'opération et ranger le résultat dans le registre destination rd.

   Instruction chargement/rangement : Calculer l'adresse de la mémoire en fonction du mode d'adressage.

   Instruction de branchement : Tester l'égalité des registres et calculer l'adresse de branchement, puis sélectionner l'adresse de l'instruction suivante en fonction du résultat du test.
5. Lecture ou écriture de la mémoire de données

   Instruction de rangement : écrire le contenu du registre rt à l'adresse calculée ci-dessus

Instruction de chargement : lire le contenu de la mémoire dont l'adresse est calculée ci-dessus puis ranger le résultat dans le registre destination rd

3.2 Conception des éléments de base

Dans l'ordinateur les informations sont codées sous forme de valeurs binaires. Afin de modéliser leurs comportements nous définirons des variables booléennes ayant deux états et un ensemble de fonctions à n variables.

3.2.1 Fonction booléenne à une variable

La seule fonction intéressante est l'inversion logique notée $\overline{a}$ appelée fonction NON.

a	$\overline{a}$
0	1
1	0

FIG. 3.1: Fonction Non

3.2.2 Fonctions à deux variables

Sur les 16 fonctions possibles avec 2 variables, seules 6 sont intéressantes :

a	**b**	$a+b$	$\overline{a+b}$	$a.b$	$\overline{a.b}$	$a \oplus b$	$\overline{a \oplus b}$
		OR	NOR	AND	NAND	XOR	NXOR
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1

FIG. 3.2: Fonctions logiques

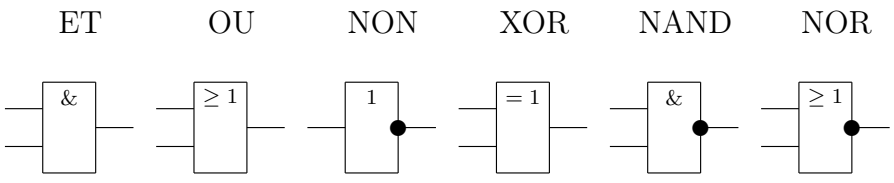


FIG. 3.3: Représentation graphique des portes logiques

3.2.3 Règles de simplification

Les régles de l'arithmétique booléennes permettent de simplifier les expressions.

$$a+b = b+a \text{ Commutativité} \quad (3.1)$$
$$a.b = b.a \text{ Commutativité} \quad (3.2)$$
$$(a+b)+c = a+(b+c) \text{ Associativité} \quad (3.3)$$
$$(a.b).c = a.(b.c) \text{ Associativité} \quad (3.4)$$
$$(a+b).c = a.c+b.c \text{ Distributivité} \quad (3.5)$$
$$(a.b)+c = (a+c).(b+c) \text{ Distributivité} \quad (3.6)$$
$$a.1 = a \text{ Élément neutre} \quad (3.7)$$
$$a.a = a \text{ Élément neutre} \quad (3.8)$$
$$a+0 = a \text{ Élément neutre} \quad (3.9)$$
$$a+a = a \text{ Élément neutre} \quad (3.10)$$
$$\overline{\overline{a}} = a \text{ Élément neutre} \quad (3.11)$$
$$a.0 = 0 \text{ Élément neutralisant} \quad (3.12)$$
$$a.\overline{a} = 0 \text{ Élément neutralisant} \quad (3.13)$$
$$a+1 = 1 \text{ Élément neutralisant} \quad (3.14)$$
$$a+\overline{a} = 1 \text{ Élément neutralisant} \quad (3.15)$$
$$\overline{a+b+c+d...} = \overline{a}.\overline{b}.\overline{c}.\overline{d}... \text{ Théorème de De Morgan} \quad (3.16)$$
$$\overline{a.b.c.d...} = \overline{a}+\overline{b}+\overline{c}+\overline{d}... \text{ Théorème de De Morgan} \quad (3.17)$$

3.2.4 Méthodologie de conception d'un circuit

Pour concevoir un circuit, nous devons le modéliser. Pour cela nous utiliserons une table de vérité. Nous obtiendrons alors une équation que nous simplifierons avec les règles précédentes. Nous pourrons alors tracer le schéma électrique du circuit.

Table de vérité et équations de l'addition de deux bits

La table de vérité présente toutes les combinaisons possibles des n entrées (soit 2^n lignes) et l'état des sorties correspondantes. Ici n=2, donc nous avons quatre lignes. Une fois la table de vérité écrite, nous devons la transformer en une équation logique. C'est une somme (fonction ou) de produit (et).

Lorsque nous additionnons deux informations codées sur un bit a et b , nous générons un résultat r et une retenue c.

a	b	c	r
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

FIG. 3.4: Table de vérité de l'addition 2 bits

L'équation de c est $c = a.b$ ($c = 1$ Si $a = 1$ et $b = 1$)

L'équation de r est $r = \overline{a}.b + a.\overline{b} = a \oplus b$. Pour que la sortie soit à 1, $a = 0$ (soit $\overline{a} = 1$) et $b = 1$ ou $a = 1$ et $b = 0$ (soit $\overline{b} = 1$)

Les équations précédentes ne peuvent pas être simplifiées.

Schéma électrique

Si nous ne disposons que de portes logiques nands à deux entrée ($s = \overline{a.b}$) nous devons modifier les équations pour obtenir des produits inversés.

$$\begin{aligned} c &= \overline{\overline{c}} \\ &= \overline{\overline{a.b}} \\ &= \overline{\overline{(a.b)}.\overline{(a.b)}} \end{aligned} \tag{3.18}$$

Pour fabriquer la retenue c, 2 portes nands sont nécessaires.

$$\begin{aligned} r &= \overline{\overline{r}} \\ &= \overline{\overline{a.\overline{b} + \overline{a}.b}} \\ &= \overline{\overline{a.\overline{b}}.\overline{\overline{a}.b}} \\ &= \overline{\overline{(a.\overline{(b.b)})}.\overline{\overline{((a.a)}.b)}} \end{aligned} \tag{3.19}$$

Pour fabriquer le résultat r, 5 portes nands sont nécessaires.

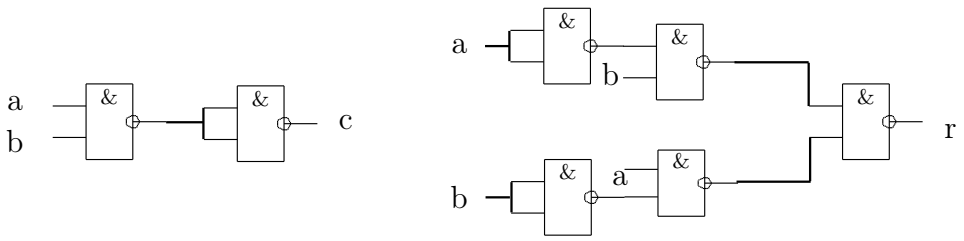


FIG. 3.5: Schéma de l'additionneur 2 bits

3.2.5 Élément du chemin de donnée du MIPS

Multiplexeur

Ce circuit permet d'envoyer sur la sortie c l'état d'une entrée a ou de l'autre b en fonction d'un signal de sélection s.

a	b	s	c
0	X	0	0
1	X	0	1
X	0	1	0
X	1	1	1

FIG. 3.6: Table de vérité du multiplexeur 2 bit

Dans la table de vérité, la notation X signifie que le signal peut prendre les deux valeurs 0 ou 1 sans influence sur le résultat. Cela permet de diminuer la taille de la table de vérité, en éliminant les lignes identiques.

$$\begin{aligned} c &= a.(b+\overline{b}).\overline{s} + (a+\overline{a}).b.s \\ c &= a.\overline{s} + b.s \qquad (3.20) \\ &= \overline{\overline{c}} \\ &= \overline{\overline{a.\overline{s} + b.s}} \\ &= \overline{\overline{a.\overline{s}}.\overline{b.s}} \qquad (3.21) \end{aligned}$$

4 Portes nands nécessaires pour le calcul de c.

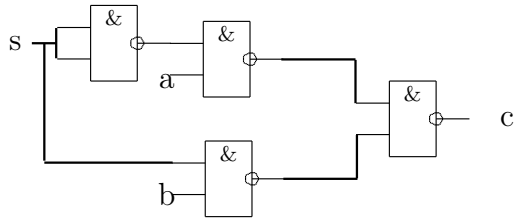


FIG. 3.7: Schéma du multiplexeur 2 bits

Décodeur

Ce circuit permet d'activer une sortie parmi 2^n en fonction de la valeur présente sur les n bits d'entrées $a_i...a_2, a_1, a_0$.

a_1	a_0	s_3	s_2	s_1	$s0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

FIG. 3.8: Table de vérité du décodeur 2 bits

$$
\begin{aligned}
s_3 &= a_1.a_0 \\
s_2 &= a_1.\overline{a_0} \\
s_1 &= \overline{a_1}.a0 \\
s_0 &= \overline{a_1}.\overline{a_0}
\end{aligned}
$$

Unité Arithmétique et Logique 1 bit (UAL)

L'unité arithmétique et logique permet d'effectuer les opérations sur les données. Nous allons construire l'UAL traitant des données codées sur 1 bit.

En fonction des signaux Op_1 et Op_0 permettant de sélectionner une opération, le circuit calcule le résultat de

- a ET b si $Op_1 = 0$ et $Op_0 = 0$
- a OU b si $Op_1 = 0$ et $Op_0 = 1$
- a plus b si $Op_1 = 1$ et $Op_0 = 0$
- a - b si $Op_1 = 1$ et $Op_0 = 1$

Op_1	Op_0	a	b	c	r
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	0

FIG. 3.9: Table de l'UAL 1 bit

$$c = Op_1.\overline{Op_0}.a.b + Op_1.Op_0.\overline{a}.b \tag{3.22}$$

$$\begin{aligned} r &= \overline{Op_1}.\overline{Op_0}.a.b + \overline{Op_1}.Op_0.\overline{a}.b + \overline{Op_1}.Op_0.a.\overline{b} + \overline{Op_1}.Op_0.a.b + \\ &\quad Op_1.\overline{Op_0}.\overline{a}.b + Op_1.\overline{Op_0}.a.\overline{b} + Op_1.Op_0.\overline{a}.b + Op_1.Op_0.a.\overline{b} \\ &= \overline{Op_1}.\overline{Op_0}.a.b + \overline{Op_1}.Op_0.a.b + \overline{Op_1}.Op_0.\overline{a}.b + Op_1.\overline{Op_0}.\overline{a}.b + \\ &\quad Op_1.Op_0.\overline{a}.b + \overline{Op_1}.Op_0.a.\overline{b} + Op_1.\overline{Op_0}.a.\overline{b} + Op_1.Op_0.a.\overline{b} \\ &= \overline{Op_1}.a.b(\overline{Op_0} + Op_0) + \overline{a}.b(\overline{Op_1}.Op_0 + Op_1.\overline{Op_0} + Op_1.Op_0) + a.\overline{b} \\ &\quad (\overline{Op_1}.Op_0 + Op_1.\overline{Op_0} + Op_1.Op_0) \\ &= \overline{Op_1}.a.b + \overline{a}.b(Op_1 + Op_0) + a.\overline{b}(Op_1 + Op_0) \\ &= \overline{Op_1}.a.b + (Op_1 + Op_0)(a.\overline{b} + \overline{a}.b) \end{aligned} \tag{3.23}$$

Une UAL traitant des informations n bits est obtenue en assemblant n UAL traitant 1 bit.

Registre 1 bit

Un circuit est séquentiel lorsque pour la même combinaison des entrées, la sortie peut prendre diverses valeurs en fonction du temps. Ceci permet la mémorisation d'un état passé. Dans la table de vérité nous trouverons, en plus des entrées, la valeur de la sortie à l'instant précédent. Ces entrées seront suivies du signe $-$.

Nous allons définir un circuit qui recopie l'état de l'entrée d sur la sortie q si le signal $h = 1$. Sinon la sortie ne changera pas et la sortie conservera l'état de q_-

h	d	q_-	q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

FIG. 3.10: Table de la mémoire 1 bit

$$\begin{aligned} q &= \overline{h}.\overline{d}.q_- + \overline{h}.d.q_- + h.d.q_-/ + h.d.q_- \\ &= \overline{h}.q_- + h.d \\ &= q// \\ &= \overline{\overline{(\overline{h}.q_- + h.d)}} \qquad (3.24) \\ &= \overline{\overline{(\overline{h}.q_-)}.\overline{(h.d)}} \qquad (3.25) \end{aligned}$$

$q-$ est fabriqué en envoyant la sortie du circuit sur une entrée.

Un registre n bits est l'assemblage de n registres 1 bit. La valeur à mémorisée est placée sur les entrées d. Le signal h est commun et permet de mémoriser la valeur dans le registre.

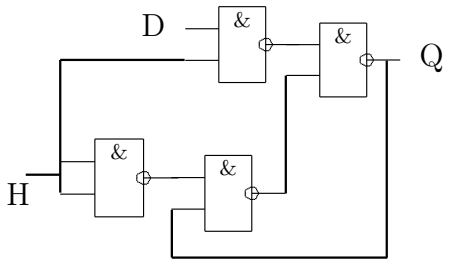


FIG. 3.11: Schéma du registre 1 bit

3.3 Lecture Instruction

Le registre spécial nommé **compteur programme cp** contient l'adresse de l'instruction mémoire à traiter. La mémoire d'instruction est accédée et nous calculerons cp pour qu'il pointe sur la prochaine instruction. La mémoire contenant des informations codées sur 8 bits et l'instruction étant elle même codée sur 32 bits, le registre cp doit être augmenté de 4 pour passer à l'instruction suivante.

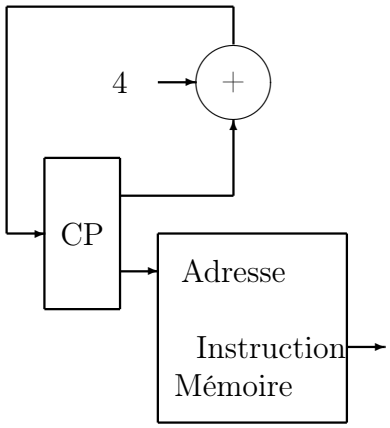


FIG. 3.12: Schéma de la lecture de l'instruction

3.4 Instructions arithmétiques et logiques

3.4.1 Le banc de registre

Le banc de registres contient les 32 registres. Il possède deux ports en lecture et un port en écriture. Pour chaque instruction, nous devons lire un ou deux registres (rs et rt) et écriture un résultat.

RL1 contient le numéro du registre rs à lire et **DL1** la valeur contenue dans ce registre.
RL2 contient le numéro du registre rt à lire et **DL2** la valeur contenue dans ce registre.
RE contient le numéro du registre à écrire et **DE** la valeur à écrire dans ce registre.

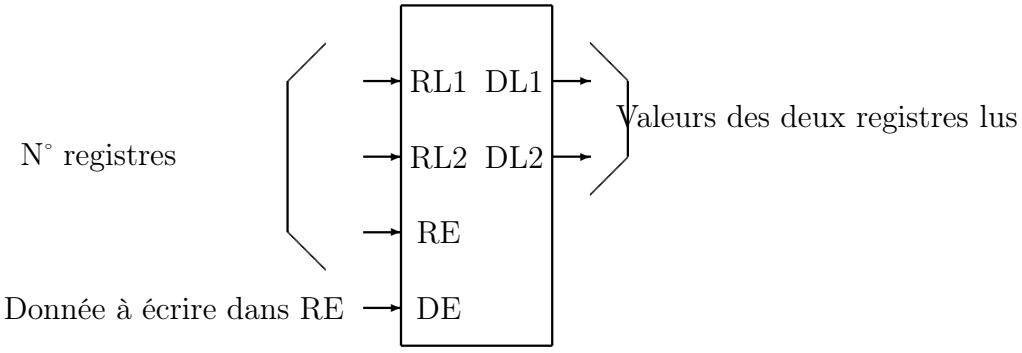


FIG. 3.13: Schéma externe du banc de registres

L'intérieur du banc de registre est composé de 32×32 registres 1 bit, d'un décodeur $5 \rightarrow 32$ qui permet de sélectionner le registre à écrire et deux multiplexeurs $32 \rightarrow 1$ qui permettent de placer sur les sorties les valeurs des deux registres lus.

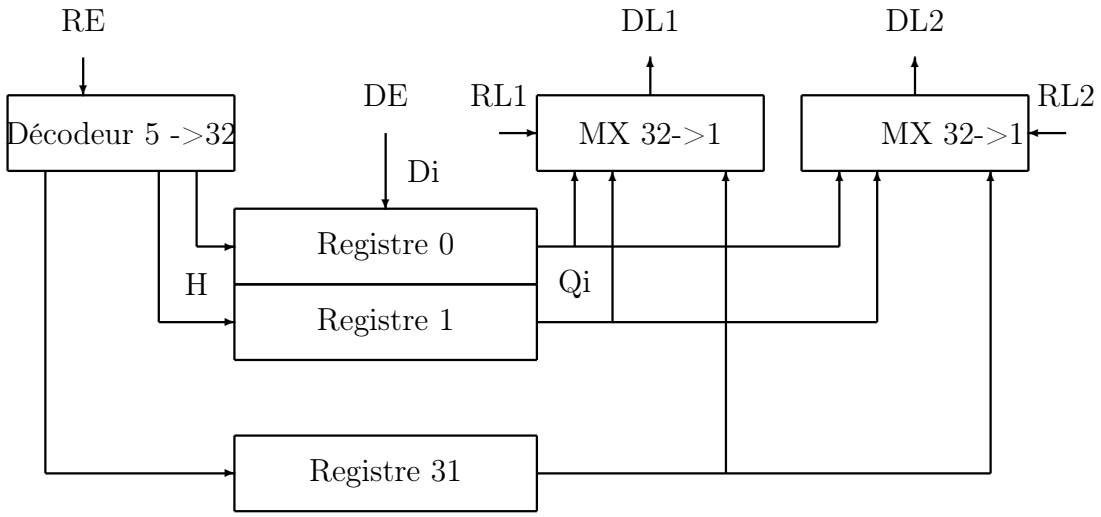
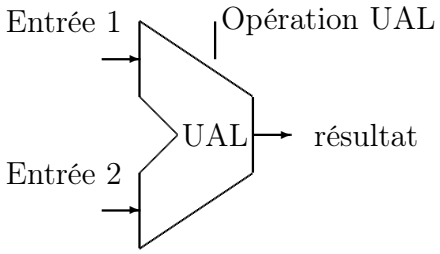


FIG. 3.14: Schéma interne du banc de registres

3.4.2 Unité arithmétique et logique

Cet élément permet d'effectuer une opération sur deux données. L'entrée nommée **Opération UAL** permet de sélectionner l'opération $+, -$,ou,et qui sera appliquée aux deux données nommées **Entrée 1** et **Entrée2**. Le résultat sera placé sur la sortie nommée **Résultat**.



L'unité arithmétique et logique

3.4.3 Instructions arithmétiques et logiques trois registres

Les instructions à trois registres utilisent le banc de registres et l'UAL. Les champs correspondants aux numéros des registres rs rt et rd sont placés sur les entrées du banc de registre. A la sortie de celui ci, les valeurs de rs et rt sont placées sur les entrées de l'UAL qui effectue l'opération codée dans les champs Co et du Nf de l'instruction. Le résultat est placé dans le registre dont le numéro est rd.

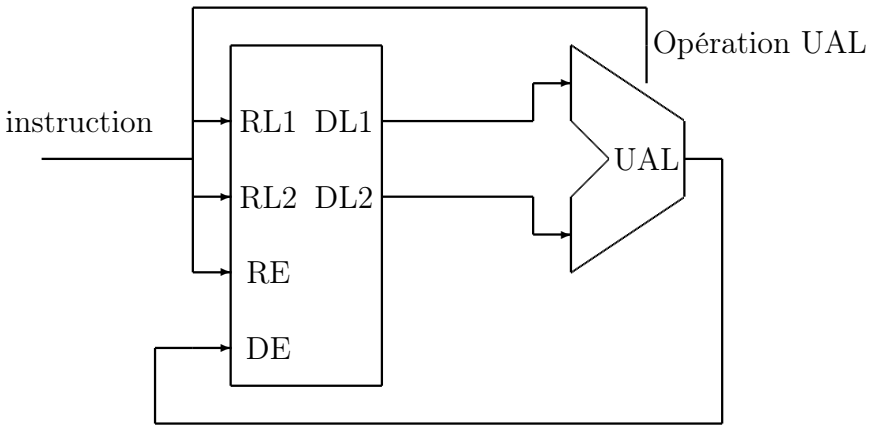


FIG. 3.15: Chemin de données pour les opérations UAL trois registres

3.4.4 Instructions arithmétiques et logiques utilisant une valeur immédiate

L'instruction à deux registres et une valeur immédiate utilise le banc de registres et l'UAL. La valeur immédiate étant codée sur 16 bits, elle doit être transformée en une valeur signée codée sur 32 bits avec une extension du signe.

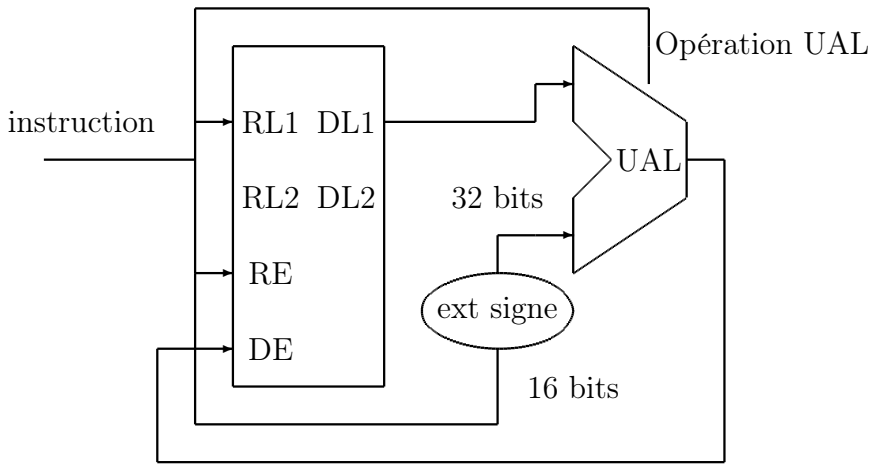


FIG. 3.16: Chemin de données pour les opérations UAL immédiate

3.5 Instructions de chargement et de rangement

Ces instructions accèdent à la mémoire de données. L'UAL calcule l'adresse de la mémoire qui sera lue ou écrite en ajoutant la valeur immédiate à la valeur du registre rs.

La donnée sera écrite en mémoire ou lue et écrite dans un registre rt.

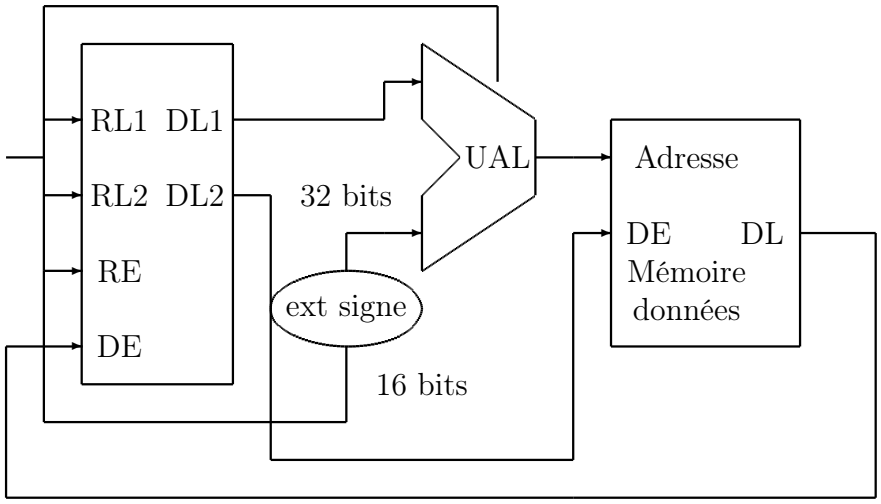


FIG. 3.17: Chemin de données pour les opérations de chargement/rangement

3.6 Instructions de branchement conditionnel

Lors du branchement conditionnel, l'UAL est utilisée pour effectuer le calcul de l'adresse de branchement. Un circuit supplémentaire noté **Cond** teste l'égalité des deux opérandes rs et rt.

Le multiplexeur piloté par le résultat de Cond positionne sur sa sortie la valeur calculée par l'UAL dans le cas d'un branchement pris, ou (cp+4) dans le cas d'un branchement non pris.

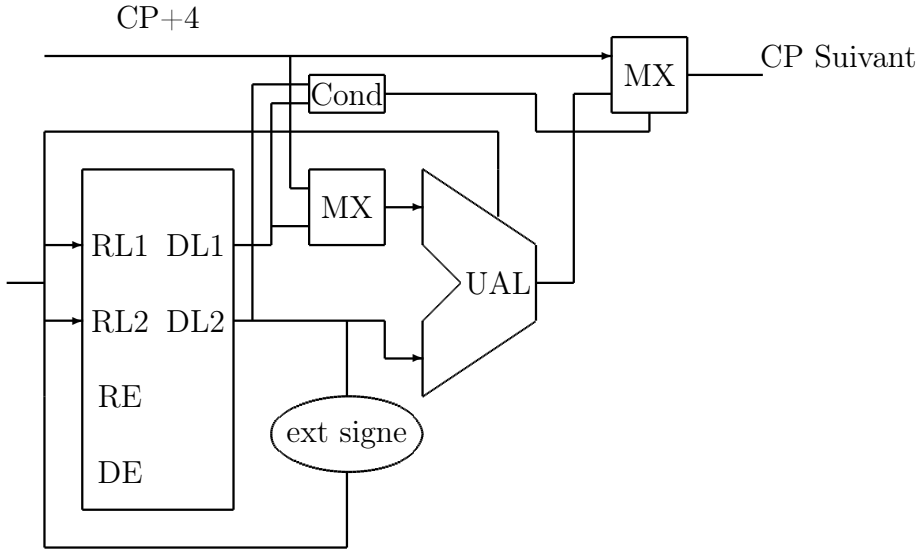


FIG. 3.18: Chemin de données pour l'opération UAL

3.7 L'assemblage du tout

Le processeur MIPS complet est obtenu en mettant ensemble les différentes parties. Trois multiplexeurs sont ajoutés pour :

- sélectionner la source placée sur l'entrée de l'UAL (soit la valeur (cp+4) soit la valeur du registre rs),
- sélectionner la source placée sur l'entrée de l'UAL (soit la valeur immédiate soit la valeur du registre rt),
- sélectionner ce qui doit être écrit dans le registre (soit le résultat de l'UAL, soit la donnée lue en mémoire).

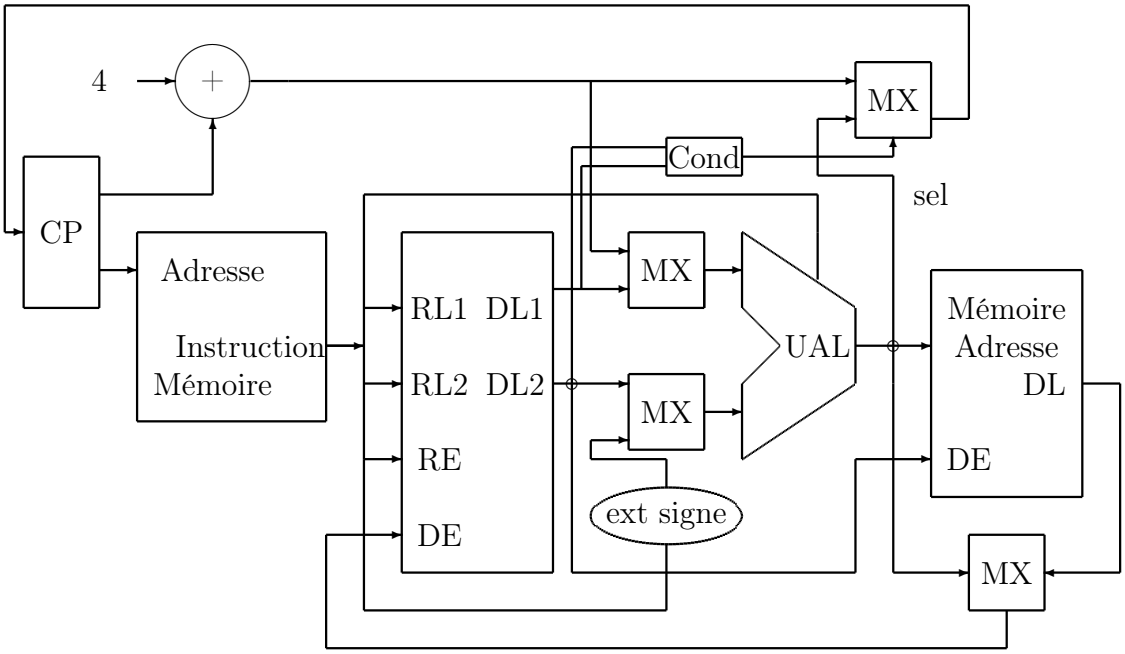


FIG. 3.19: Chemin de données complet

3.8 Exercices

3.8.1 Théoriques

Exercice 8 : Application de la logique

Question 1 : Donnez l'équation de la sortie du bloc Cond nommée $zero? = f(i_{31}, ...i_0)$ permettant de piloter le multiplexeur qui sélectionne l'adresse de l'instruction cible. L'équation utilisera des uniquement des portes nands.

Question 2 : UAL 1 bit L'UAL proposée est incomplète. En effet une UAL n bits doit tenir compte de la retenue générée par le bit de rang précédent (notée c_p). Donnez les équations de $c = f(a, b, c_p, Op_1, Op_0)$ et de $r = f(a, b, C_p, Op_1, Op_0)$

Question 3 : Les signaux Op_1 et Op_0 sont obtenus à partir du code opération de l'instruction. Donnez les équations utilisant l'opérateur d'égalité de $Op_0 = f(Co, Nf)$ et de $Op_1 = f(Co, Nf)$

Question 4 : Ajoutez au chemin de données l'instruction **jal**

Question 5 : Ajoutez au chemin de données précédent l'instruction **jr**

Chapitre 4

Système d'interruptions et d'entrées/sorties

4.1 Organisation

Un système informatique est organisé de la façon suivante :

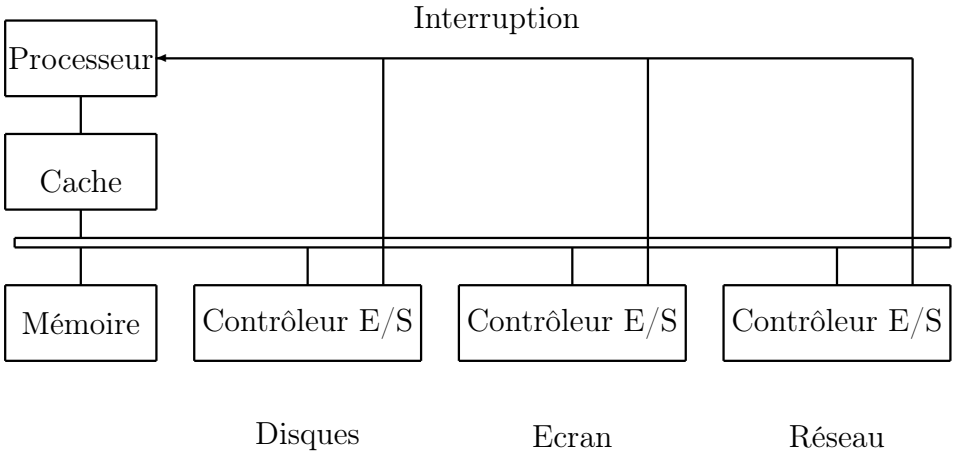


FIG. 4.1: Organisation d'un système

Nous avons étudié le processeur et nous étudierons la mémoire cache dans la partie précédente. Le système informatique contient aussi des dispositifs d'entrées/sorties (nommé aussi I/O) et un système d'interruptions.

Les interruptions permettent de gérer le système d'entrées/sorties de façon efficace. Prenons par exemple la gestion du périphérique d'entrée (clavier). Nous ne devons pas perdre de caractère en provenance de ce périphérique si le processeur est occupé à une autre activité.

Deux techniques de gestion des événements sont possibles :

Interrogation utilisant le principe suivant :

Algorithme 5 Gestion par interrogation du clavier

```
1: DÉBUT
2: TantQue VRAI Faire
3:    Si Touche clavier appuyée ? Alors
4:       Stocker en mémoire le code ascii du caractère lu
5:    FinSi
6:    Attendre 100 ms
7: Fin TantQue
8: FIN
```

Interruption utilisant le principe suivant :

Algorithme 6 Gestion par interruption du clavier

```
1: DÉBUT
2: Si Touche clavier appuyée ? Alors
3:    stocké dans le buffer le code ascii du caractère lu
4: FinSi
5: FIN
```

L'interrogation effectue une lecture de l'état du clavier alors qu'avec la technique d'interruption, c'est le clavier qui signale au processeur qu'il doit traiter un caractère et ne pas le perdre.

4.2 Principe

Une exception est un événement inattendu qui est généré par le processeur. Le débordement de la capacité de calcul, les erreurs d'accès à la mémoire sont des exceptions.

Les interruptions sont générées par le système d'entrées/sorties pour signaler qu'une donnée qu'un périphérique est prêt et requiert l'attention du processeur.

Les interruptions et les exceptions contrairement au branchement peuvent arriver à n'importe quel endroit du programme. Elle ne doivent pas modifier les registres du programme en cours. Pour cela les sous programmes gérant les interruptions disposent de deux registres nommé $k0 et $k1.

Voici quelques exemples de sources d'interruptions et d'exceptions

Nature de l'événement	**Provenance**	Type
Requête E/S	Externe	Interruption
Sycall	Interne	Exception
Débordement calcul	Interne	Exception
Instruction invalide	Interne	Exception
Probléme matériel	Les deux	Les deux

FIG. 4.2: Sources des interruptions

Lors de l'arrivée d'un événement le MIPS doit terminer l'instruction en cours de traitement, puis exécuter un sous-programme nommé d'exception ou d'interruption qui effectue un traitement spécifique.

4.3 Gestion des exceptions

Dans l'implémentation MIPS, une partie appelée le **coprocesseur 0** gère les exceptions et les interruptions. Ce coprocesseur est accessible via quatre registres.

Nom	Numéro	Usage
AdrErrRef	8	Contient l'adresse mémoire où l'erreur s'est produite
Etat	12	Bits masquage et validation interruptions
Cause	13	Type exceptions et interruption en attente
CEP	14	Adresse de l'instruction ayant provoqué l'exception

FIG. 4.3: Exceptions du MIPS

Pour accéder à ces quatre registres du coprocesseur, le MIPS possède deux instructions.

Rôle	**Syntaxe**	**Opération**	**Codage**
déplacer depuis c0	mfc0 rt,rd	rt ← rd de c0	Co=0x10
déplacer vers c0	mtc0 rd,rt	rd de c0 ← rt	Co=0x10

FIG. 4.4: Liste des instructions du coprocesseur c0

Après une exception le registre **CEP** contient l'adresse de l'instruction en cours d'exécution lors de l'arrivée de l'événement. Si l'instruction a fait un accès mémoire qui a provoqué l'exception, le registre **AdrErrRef** contient l'adresse de la mémoire référencée. Le registre **Etat** ne sera pas utilisé ici. Les bits 2 à 6 du registre **Cause** indique la nature de l'exception (voir la figure FIG. 4.5)

Numéro	Nom	Description
0	I/O	dispositif d'entrées/sorties
4	ADDRL	Erreur d'adresse Load
5	ADDRS	Erreur d'adresse Store
7	DBE	Erreur sur le bus données
8	SYSCALL	Appel système
10	RI	Instruction erronée
12	OVF	dépassement calcul

FIG. 4.5: Causes de l'exception

Lors de l'arrivée d'une exception ou d'une interruption le MIPS exécute le programme se trouvant à l'adresse 0x80000180 (dans l'espace du noyau, pas de l'utilisateur), appelé **gestionnaire d'interruptions**. Le système effectuera alors l'opération voulue pour répondre au problème ou souvent terminer le programme.

4.4 Gestionnaire d'interruption

Le programme doit être écrit dans un segment spécial nomme **.ktext 0x80000180**, et utilise les données stockée dans **.kdata**. Les registres permettant de transférer les données entre le gestionnaire d'interruption et le programme utilisateurs sont nommé $k0 et $k1 .

Le gestionnaire doit effectuer les opérations suivantes :

- Charger les registres du coprocesseur
- Analyser la cause de l'exception
- traiter le problème
- retour au programme utilisateur ou le terminer.

4.5 Entrées/Sorties de caractères

Dans une architecture MIPS la console est "mappée" en mémoire. Cela signifie que les entrées/sorties sont accessibles via des adresses mémoires spécifiques. Cet espace mémoire est nommé MMIO (Memory Manager IO).

Pour la réception de caractères :

- l'adresse de reception du caractère est 0xFFFF0004,
- l'adresse du controleur de réception est 0xFFFF0000. Si le bit 0 est à 1 cela signifie qu'un caractère est en attente. Ce bit repassera à 0 lors de l'instruction de lecture mémoire à l'adresse de réception.

Pour l'envoi de caractères :

- l'adresse d'envoi du caractère est 0xFFFF000C,

– l'adresse du controleur d'emission est 0xFFFF0008. Le bit 0 passera à 0 lors de l'écriture mémoire à l'adresse d'envoi. Il reviendra à 1 après un temps nécessaire au traitement du caractère.

4.6 Entrées/Sorties par interruption

Le bit numéro 1 des deux contrôleurs permet de valider les interruptions lors de l'envoi ou de la réception d'un caractère. Lors d'un événement, un interruption de cause numéro 0 est générée. Le bit 8 (respectivement 9) du registre cause indique la reception d'un caractère (respectivement la fin de l'envoi).

4.7 Exercices pratiques

Exercice 9 : Exceptions

Question 1 : Donnez le programme qui génère une erreur de lecture puis d'écriture mémoire (adresse invalide) et ensuite un débordement arithmétique (débordement d'un nombre signé)

Question 2 : Donnez le programme de gestion des exceptions 4,5 et 12 qui affiche "Exception ... sur l'instruction ... ", puis un message indiquant de façon lisible le problème, et en cas d'accès mémoire l'adresse ayant causée cette l'exception. Le programme sera stoppé dans tous les cas. Ce fichier source sera chargé dans le simulateur MARS avec le menu Setting, exception handler.

Exercice 10 : Entrés/Sorties

Pour tester les entrées/sorties de la console, vous utiliserez l'outils intégré à MARS. Cette outils se trouve dans le menu **tools** et se nomme **keyboard and display simulator**. L'outil doit être connecté pour fonctionner. Dans le segment de données vous afficherez le segment MMIO. Le bouton reset de l'outil permet de réinitialisé les contrôleurs (le bit de 0 du contrôleur d'envoi est mis à 1 pour dire qu'un caractère peut être envoyé)

Question 1 : En utilisant l'interrogation, écrivez un programme qui attend un caractère et l'envoie.

Remarque : L'affichage dans la fenêtre d'envoi se fait avec une latence de cinq instructions (temps de traitement du caractère). Votre programme contiendra donc au moins 5 instructions après l'envoi du caractère à écrire.

Question 2 : En utilisant la technique de l'interruption, écrivez un gestionnaire qui stocke les caractères reçus dans une chaîne de caractères placée dans le segment de donnée du noyau. Le programme principal incrémente une variable.

Remarques :

- Vous lancerez le programme en pas à pas et observerez la chaîne de caractères contenu les caractères lu dans le segment **kdata** du simulateur.
- Vous terminez le gestionnaire d'interruption avec la suite d'instructions suivantes, détaillée dans le fichier HP_AppaA.pdf.

```
mtc0,$zero,$13
mfc0 $k0, $12
andi $k0, 0xfffd
ori $k0, 0x1
mtc0 $k0, $12
eret
```

Table des figures

2.1 Chaine de production 7
2.2 Architecture de Von Neumann 7
2.3 Type d'instructions 9
2.4 Position des opérandes 10
2.5 Mode d'adressage registre 11
2.6 Numéros et noms explicites des registres 11
2.7 Mode d'adressage immédiat 12
2.8 Mode d'adressage direct 13
2.9 Mode d'adressage indirect 14
2.10 Mode d'adressage indirect immédiat 15
2.11 Utilisation de l'adressage mémoire 15
2.12 Notation des modes d'adressage 16
2.13 Liste partielle des instructions arithmétiques et logiques 16
2.14 Liste partielle des instructions chargement/rangement 17
2.15 Mode d'adressage relatif au compteur de programme 19
2.16 Mode d'adressage pseudo direct 20
2.17 Liste partielle des instructions de branchement 21
2.18 Liste des instructions diverses 21
2.19 Liste partielle des appels système 22
2.20 Liste des types définis par l'assembleur 23

3.1 Fonction Non 35
3.2 Fonctions logiques 35
3.3 Représentation graphique des portes logiques 35
3.4 Table de vérité de l'addition 2 bits 37
3.5 Schéma de l'additionneur 2 bits 38
3.6 Table de vérité du multiplexeur 2 bit 38
3.7 Schéma du multiplexeur 2 bits 39
3.8 Table de vérité du décodeur 2 bits 39
3.9 Table de l'UAL 1 bit 40
3.10 Table de la mémoire 1 bit 41
3.11 Schéma du registre 1 bit 41
3.12 Schéma de la lecture de l'instruction 42

3.13 Schéma externe du banc de registres 43
3.14 Schéma interne du banc de registres 43
3.15 Chemin de données pour les opérations UAL trois registres 44
3.16 Chemin de données pour les opérations UAL immédiate 45
3.17 Chemin de données pour les opérations de chargement/rangement 45
3.18 Chemin de données pour l'opération UAL 46
3.19 Chemin de données complet 47

4.1 Organisation d'un système 49
4.2 Sources des interruptions 51
4.3 Exceptions du MIPS 51
4.4 Liste des instructions du coprocesseur c0 51
4.5 Causes de l'exception 52

Liste des algorithmes

1 Alternative 24
2 itération Tant Que 25
3 Fonction 27
4 Fonction recursive 29
5 Gestion par interrogation du clavier 50
6 Gestion par interruption du clavier 50

Listings

2.1 Fichier assembleur minimal 23
2.2 Hello World 23
2.3 Programmation de l'alternative 24
2.4 Factorielle n 26
2.5 n ! avec une fonction 27
2.6 Fonction récursive 29