Architecture des ordinateurs - Programmation et Architecture MIPS

Didier Teifreto

Université de Franche Comté

23 septembre 2015

Programmation MIPS

1. MIPS *Microprocessor without interlocked pipeline stages*
2. Architecture simple utilisée dans de nombreux systèmes.
3. Simulateur MARS *MIPS Architecture and Runtime Simulator*

Type de jeu d'instructions 1

Architecture à pile Les données sont stockées dans un pile : y ← x +y

- ranger y dans la pile
- ranger x dans la pile
- ajouter données au sommet de la pile
- dépiler la valeur de y

Pourquoi Addition des données sur ou dans la pile, permet l'enchainement des calculs

Utilisation Utiliser opérations nombres réels sur x86

Type de jeu d'instructions - 2

Architecture accumulateur Une donnée et le résultat d'une instruction sont stockées dans l'accumulateur. L'autre donnée est en mémoire : y $\leftarrow$ x +y

- accumulateur $\leftarrow$ y
- accumulateur $\leftarrow$ accumulateur + x
- y $\leftarrow$ accumulateur

Inconvénient Jeu d'instruction destructif (perte d'une donnée pour mettre le résultat

Utilisation Opération sur les entiers dans l'architecture x86.

Type de jeu d'instructions - 3

Architecture à registres Les données et le résultat d'une instruction sont stockée dans les registres. y $\leftarrow$ x +y

- registre 1 $\leftarrow$ y
- registre 2 $\leftarrow$ x
- registre 3 $\leftarrow$ registre 1 + registre 2
- y $\leftarrow$ registre 3

Avantage Jeu d'instructions non destructif, réutilisation des données.

Utilisation Interne dans le architecture x86 et Mips

Application au jeu d'instructions MIPS

1. **Le plus petit et le plus rapide** 32 registres 32 bits, 127 instructions
2. **La simplicité favorise la régularité** Taille instructions 32 bits, trois opérandes par instruction (sauf quelques unes)
3. **Une bonne conception requiert des compromis** instruction codées sur 32 bits, donc constantes sur 16 bits seulement
4. **Faire en sorte que les cas fréquents soient les plus rapides** Manipulation des nombres entiers rapides.

- Jeu d'instruction non destructif (données non modifiées)
- Architecture ISA moderne (depuis année 90's) du type RISC *Reduced Instruction Set Computer*

Jeu d'instructions CICS

1. **Le plus petit est le plus rapide** 8 registres 16 bits, 1000 instructions
2. **La simplicité favorise la régularité** Taille instructions de 8 à 64 bits, jeux d'instruction destructif.
3. **Une bonne conception requiert des compromis** 10 % des instructions utilisées 90 % du temps !
4. **Faire en sorte que les cas fréquents soient les plus rapides** Non implémenté.

Architectures anciennes (avant 70's) du type CISC *Complete Instruction Set Computer*. Ils ne sont plus utilisés.

Abstraction logicielle

Langage de haut niveaux

```
1  int somme(int x,int y,int z){
2  int t;
3  t = x+y+z;
4  return t;
```

Langage assembleur

```
1  somme:
2  add $t0,$zero,$zero
3  add $t0,$a0,$a1
4  add $t0,$t0,$a2
5  jr $ra
```

Code hexadécimal

- **0x**0040 0000 : **0x**0000 4020
- **0x**0040 0004 : **0x**0085 4020
- **0x**0040 0008 : **0x**0106 4020
- **0x**0040 000C : **0x**03e0 0008

Chaîne de développement

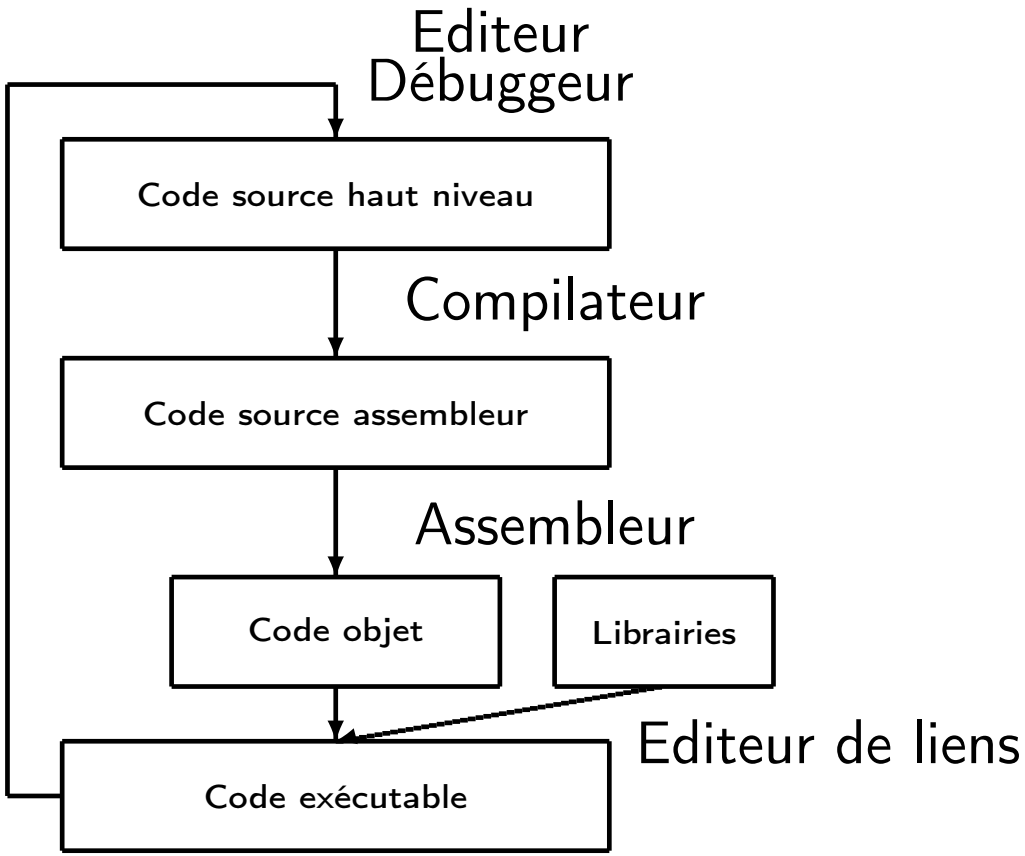


Figure 1 : Chaine de développement

Chaîne de développement- 2

- Compilateur : Analyse syntaxique et transformation code source de langage haut niveau (c, java) en bas niveau (byte code, assembleur) .
- Assembleur : Analyse syntaxique et transformation code source assembleur en code objet.
- Éditeur de lien : intégration des librairies au code objet pour le rendre exécutable.
- Debugger : Mise au point logique du programme (pas à pas, vue interne de l'architecture)

Nous utiliserons les entrées sorties systèmes entrée/sortie (ce n'est pas une bibliothèque, pas d'édition de lien). Ils ne sont pas intégré au code exécutable.

Architecture de Von Neumann

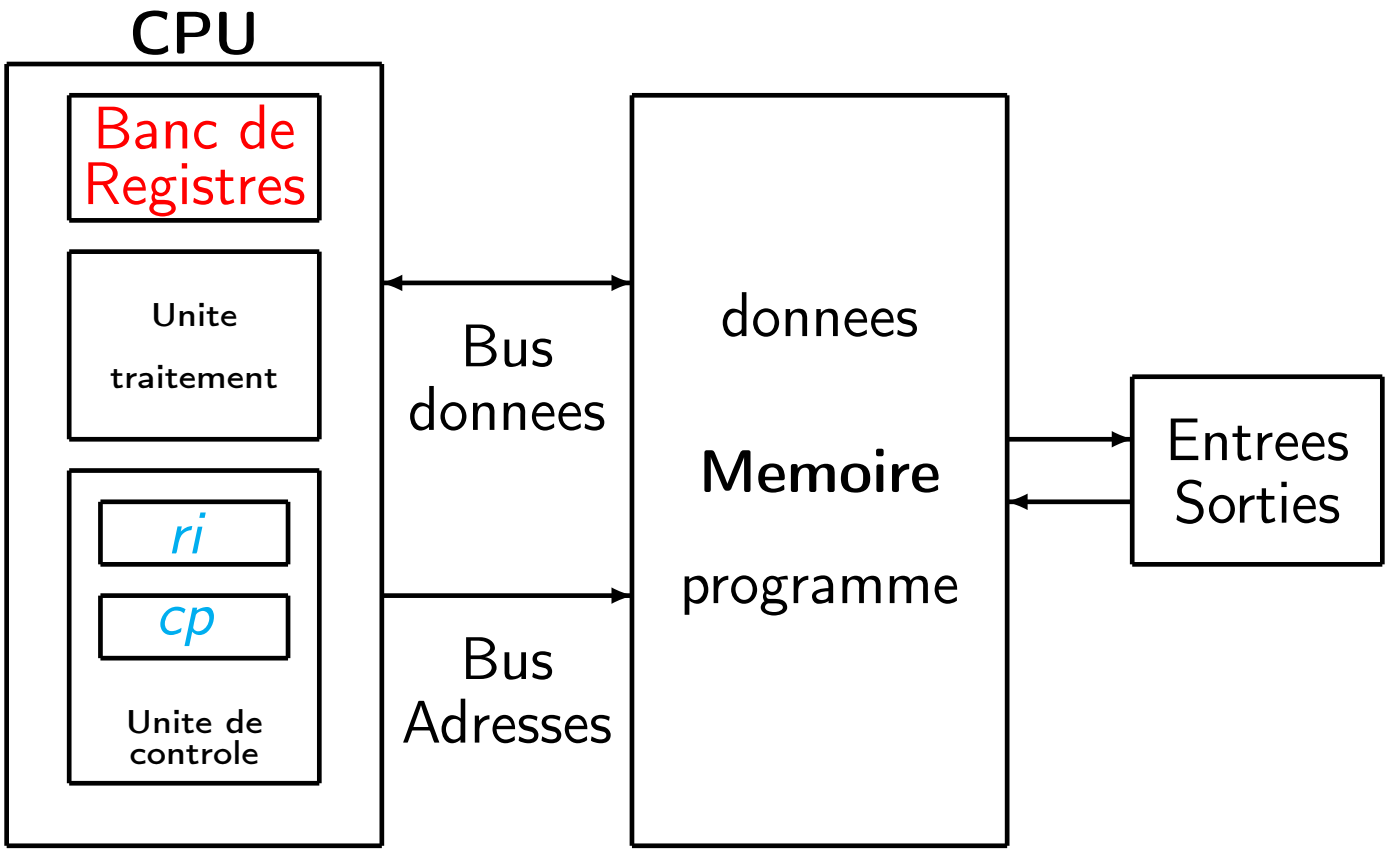


Figure 2 : Architecture de Von Neumann

Architecture de Von Neumann - 2

- Instructions et données en mémoire
- CPU *Central Process Unit*
- Unité de contrôle : séquencement des instructions
- Unité de traitement :effectue les opérations (calculs)
- *cp* : registre compteur de programme : adresse de l'instruction en cours de traitement
- *ri* : registre instruction : valeur hexadécimale représentant l'instruction traitement
- Bus de données et d'adresses : véhicule les informations entre les deux parties
- Deux parties distinctes, goulot d'étranglement

Hello World !

Listing 1 – Hello World

```
1   .data #debute à l'adresse 0x1001 0000
2     Msg:    .asciiz "Hello World \n"
3   .text
4   main: #debute à l'adresse 0x0040 0000
5     #Affichage de la chaine de caractères
6     addi $v0, $zero, 4      # Sycall 4 afficher une chaine
7     la $a0, Msg             # Adresse de la chaine
8     syscall                 # Afficher maintenant
9     # Fin
10    addi $v0, $zero, 10
11    syscall
```

Syntaxe instruction MIPS

1. Syntaxe instruction
   - Nom Instruction Opérande 1, Opérande 2, Opérande 3
   - Nom Instruction Opérande 1, Opérande 2
   - Nom Instruction Opérande 1
   - Nom Instruction
2. Opérandes
   - Opérande 1 : résultat de l'instruction
   - Opérande 2 et 3 : données de l'instruction

Syntaxe instruction MIPS - 2

- Nous devons coder le nom de l'instruction (Code opération entier non signé)
- Nous devons coder les opérandes
  - Coder les registres utilisés : nous utiliserons leur numéro (entier non signé)
  - Coder les constantes (entiers signés)

Type d'instruction

type R (Registres) Trois opérandes de type registres,

type I (Immédiate) Deux opérandes de type registres et une opérande de type immédiate sur 16 bits,

Type J (Jump) Une opérande de type immédiate sur 26 bits.

Exemple d'instruction

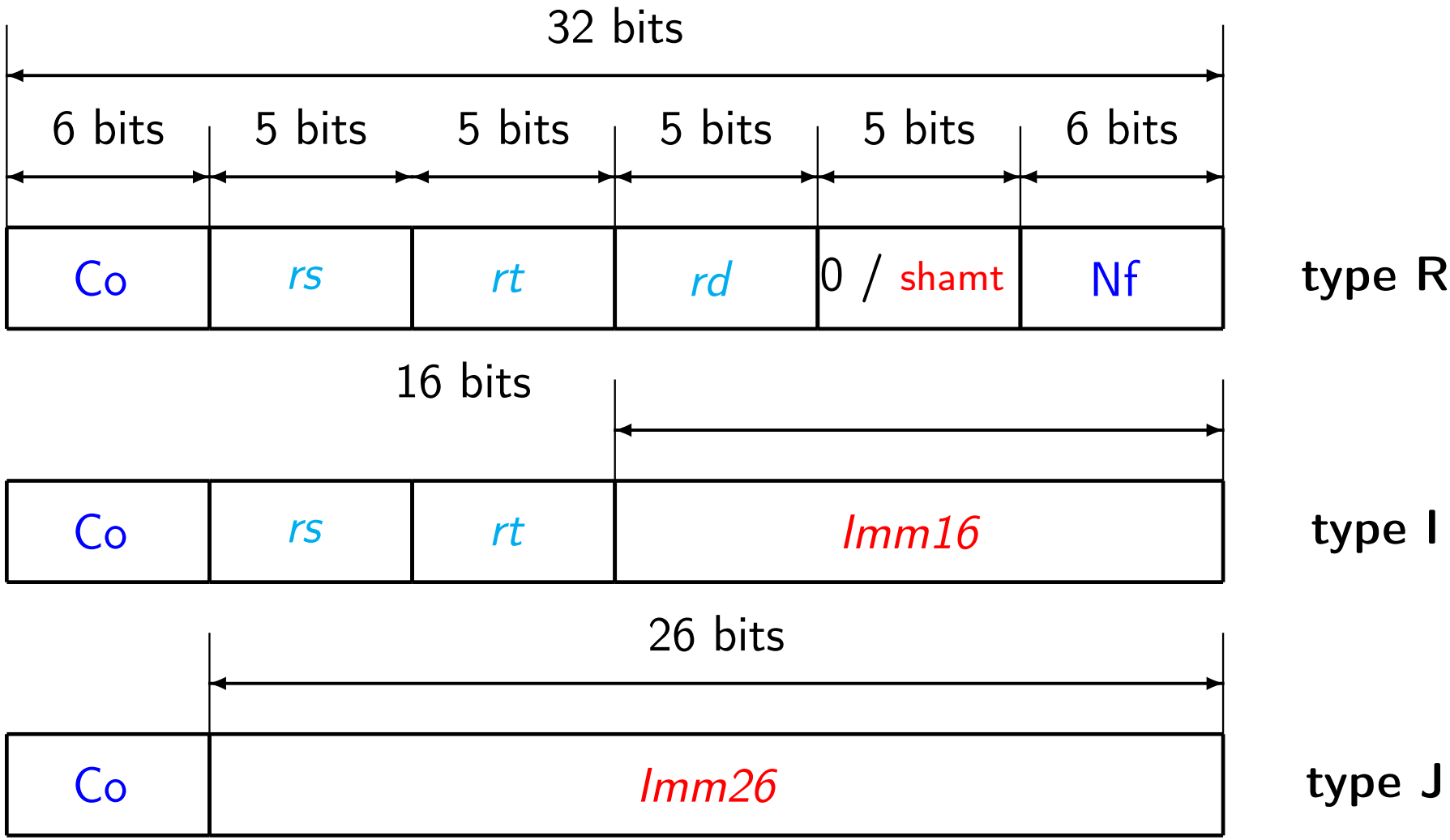
type R Trois opérandes registres, *rd* ← *rs* opération *rt* par exemple add *$t0* , *$t1* , *$t2*

type I Deux opérandes registres et une opérande immédiate sur 16 bits, *rt* ← *rs* opération *Imm16* par exemple add *$t0* , *$t1* ,-1

Type J Une opérande immédiate sur 26 bits. *cp* ← *Imm26* par exemple jal Fact ou jal **0x**40 1100

Dans le projet, nous implémenterons les instructions de type R et I. Vous pourrez aussi en fonction du temps disponible implémentez le type J

Codage instructions MIPS



Codage instructions MIPS - 2

- Code opération : Co codé sur 6 bits, 2^6 instructions, soit 63+1
- Numéro des registres *rs* , *rt* , *rd* : $2^5 = 32$ codé sur 5 bits,
- Numéro de fonction (type R) Nf codé sur 6 bits, 2^6 instructions, soit 64
- Nombre de décalage pour les instruction de décalage (type R même si opérande immédiate) avec shamt codé sur 5 bits $2^5 = 32$.
- Valeur immédiate *Imm16* codé sur 16 et *Imm26* 26 bits.

Au total 63+64 = 127 instructions, jeu RISC

Modes d'adressage

Ou trouver les opérandes ?

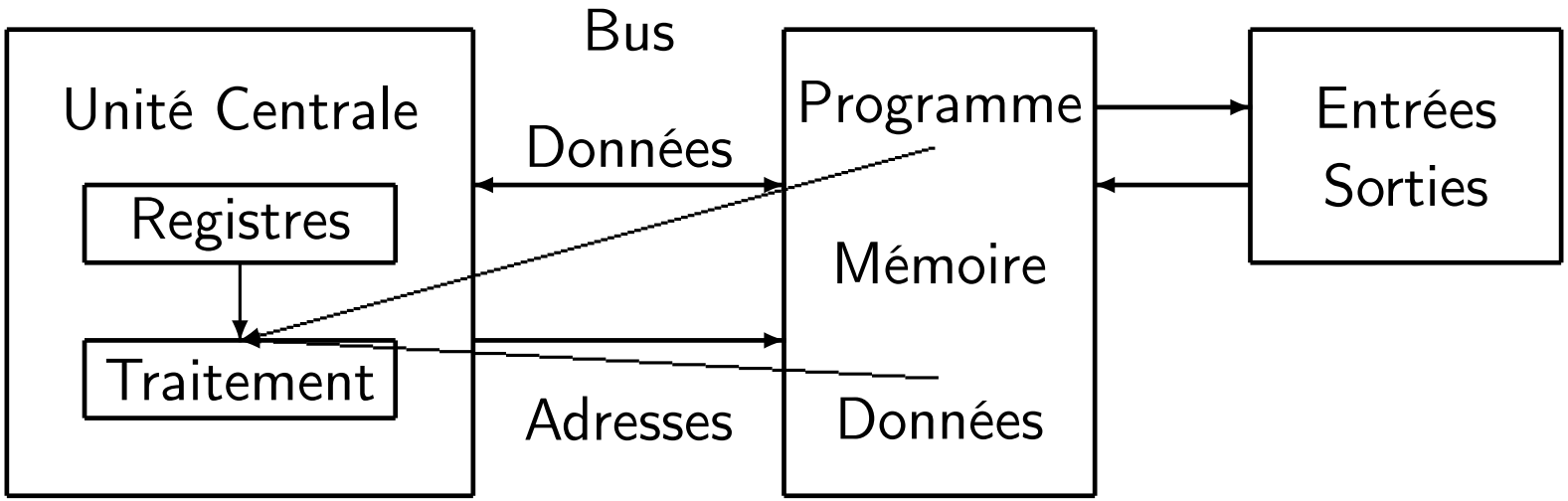


Figure 4 : Positions opérandes

- Registres,
- Constantes : Mémoire de code - instruction,
- variables du programme : Mémoire de données.

Modes d'adressage registre

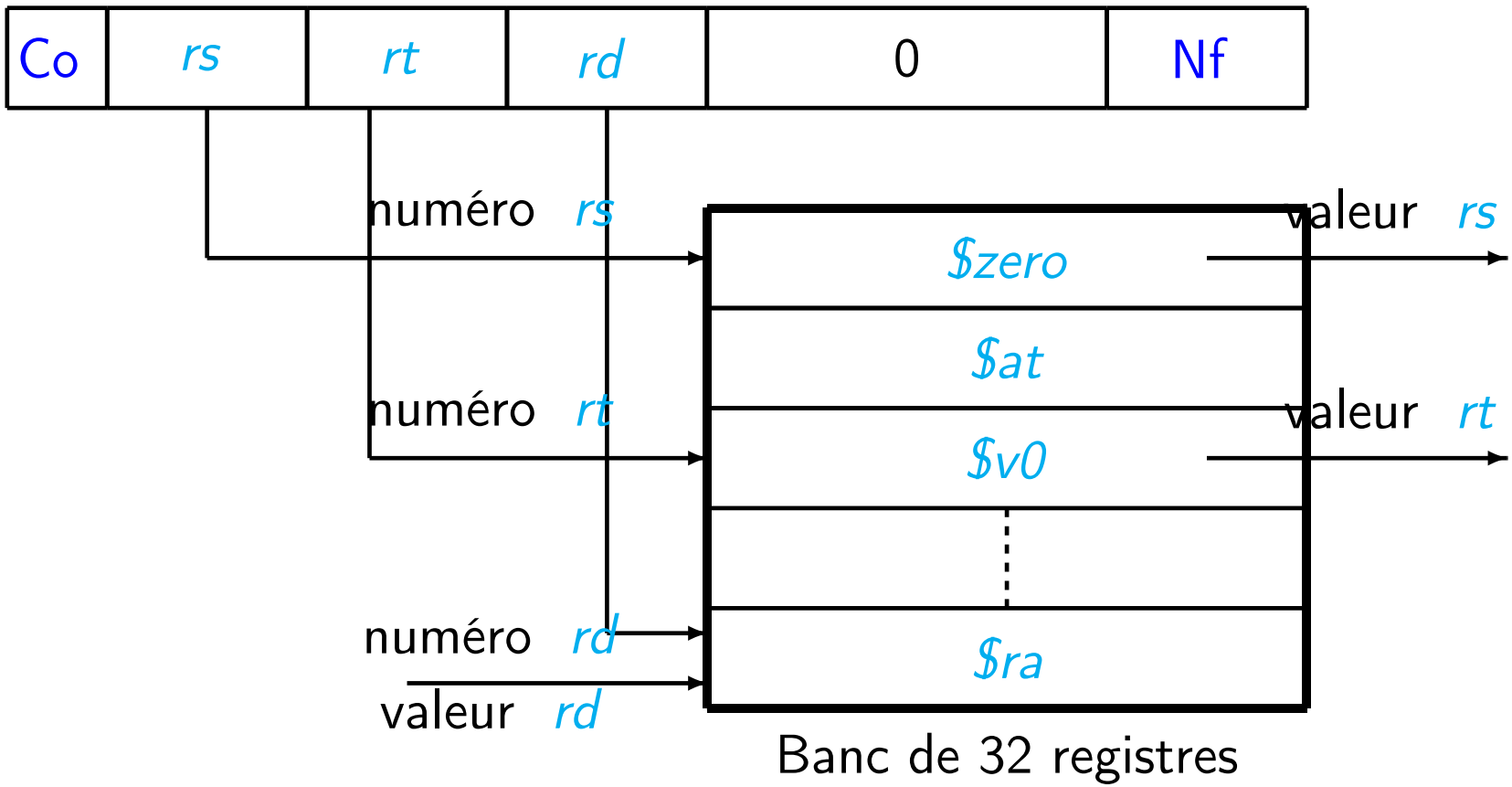


Figure 5 : Mode adressage registre

Modes d'adressage registre

Nom symbolique	Numéro	Utilisation
$zero	$0	La valeur 0
$at	$1	réservé assembleur
$v0 *$v1*	$2 $3	résultats de la fonction
$a0 à *$a3*	$4 à $7	arguments de la fonction
$t0 à *$t8*	$8 à $15	libres
$k0 et *$k1*	$26 à $27	OS kernel
$ra	$31	adresse retour de fonction

Modes d'adressage immédiat

- Mode I et J
- Constante directement dans le code (non modifiable)
- Pas de constante 32 bits (sinon pas de place pour le reste)
- Extension 16 à 32 bits en fonction du type d'instruction (arithmétique, logique) et 26 à 32.

Figure 6 : Mode adressage immédiat

Instructions arithmétiques et logiques

Rôle	Syntaxe	Opération	Codage
addition	add *rd* , *rs* , *rt*	*rd* ← *rs* + *rt*	Co =0 Nf = 0x20
	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 =0 Nf =0x1C
et bit à bit	and *rd* , *rs* , *rt*	*rd* ← *rs* & rt	Co =0 Nf =0x24
	andi *rt* , *rs* ,*Imm16*	*rt* ← *rs* & *Imm16*	Co =0xC
ou bit à bit	or *rd* , *rs* , *rt*	*rd* ← *rs* | *rt*	. Co =0 Nf =0x25
	ori *rt* , *rs* ,*Imm16*	*rt* ← *rs* | *Imm16*	Co =0xD
ou exclusif	xor *rd* , *rs* , *rt*	*rd* ← *rs* ^ *rt*	. Co =0 Nf =0x26
	xori *rt* , *rs* ,*Imm16*	*rt* ← *rs* ^ *Imm16*	Co =0x6
non ou	nor *rd* , *rs* , *rt*	*rd* ← ∼ (*rs* | *rt*)	. Co =0 Nf =0x27
décalage à gauche	sll *rd* , *rt* ,shamt	*rd* ← *rt* << Shamt	Co =0 Nf =0
	sllv *rd* , *rt* , *rs*	*rd* ← *rt* << *rs*	Co =0 Nf =4
décalage droite	sra *rd* , *rt* ,shamt	*rd* ← *rt* >> shamt	Co =0 Nf =3
	srav *rd* , *rt* , *rs*	*rd* ← *rt* >> *rs*	Co =0 Nf =7
	srl *rd* , *rt* ,shamt	*rd* ← *rt* >>> shamt	Co =0 Nf =2
	srlv *rd* , *rt* , *rs*	*rd* ← *rt* >>> *rs*	Co =0 Nf =6
charger demi mot poids fort	lui *rt* ,*Imm16*	*rt* ← *Imm16* << 16	Co =0xF

Codage des instructions arithmétiques et logiques - 2

Exemple :

- add *$t0* , *$zero* , *$zero* # *$t0* ← 0
- addi *$t0* , *$zero* ,**0x**123 # *$t0* ← **0x**123
- addi *$t1* , *$t0* ,**0x**FFFF # *$t1* ← **0x**122 - conversion 32 bits **0x**FFFF FFFF
- ori *$t2* , *$t0* ,**0x**FFFF # *$t2* ← **0x**FFFF conversion 32 bits **0x**0000 FFFF
- addi *$t3* , *$t0* , *$t2* # *$t3* ← *$t0* + *$t2*
- sll *$t4* , *$t0* ,4 # *$t4* ← **0x**1230
- sllv *$t4* , *$t0* , *$t5* # *$t4* ← *$t0* << *$t5*
- lui *$t6* ,**0x**ABCD # *$t6* ← **0x**ABCD 0000

Codage des instructions arithmétiques et logiques - 3

- add *$t0* , *$t1* , *$zero* : Co =0,Nf =**0x**20, le numéro de *$t0* est 8, de *$t1* est 9 et de *$zero* est 0.

Co 6 bits	*rs* 5 bits	*rt* 5 bits	*rd* 5 bits	0/ shamt 5 bits	Nf 6 bits
0	9	0	8	0	**0x**20
0000 00	01 001	0 0000	0100 0	000 00	10 0000

0b 0000 0001 001 0 0000 0100 0000 0010 0000 = **0x**0120 4020

- addi *$t0* , *$zero* ,**0x**123 : Co =8,

Co 6 bits	*rs* 5 bits	*rt* 5 bits	*Imm16* 16 bits
8	0	8	**0x**123
0010 00	00 000	0 1000	0000 0001 0010 0011

0b 0010 0000 000 0 1000 0000 0001 0010 0011 = **0x**2008 0123

Codage des instructions arithmétiques et logiques - 4

- sll *$t0* , *$t1* ,4 : Co =0,Nf =0, le numéro de *$t0* est 8, de *$t1* est 9.

Co 6 bits	*rs* 5 bits	*rt* 5 bits	*rd* 5 bits	0/ shamt 5 bits	Nf 6 bits
0	0	9	8	4	0
0000 00	00 000	0 1001	0100 0	001 00	00 0000

0b 0000 0000 000 0 1001 0100 0001 0000 0000 = **0x**0009 4100

Instructions et pseudo-instruction

1. Pseudo-instruction
   - Pseudo-instruction est une instruction interprétée
   - addi *$t0* , *$zero* ,**0x**1234 ABCD
   - la *$ra* ,**0x**0040 0010 (chargement adresse)

   sont traduite par l'assembleur en les suites d'instructions suivantes qui utilisent *$at* (assembler temporary)

   ```
   lui $at,0x1234 # $at = 0x1234 0000
   ori $at,$at,0xABCD # $at = 0x1234 ABCD
   add $t0,$zero,$at

   lui $at,0x0040 # $ra = 0x0040 0000
   ori $ra,$at,0x10 # $ra = 0x0040 0010
   ```

2. extension 16 - 32 bits
   - Extension 32 bits signée pour les opérations arithmétiques.
   - Extension 32 bits non signé (à 0) pour les opérations logiques.

Modes d'adressage direct - Très limité

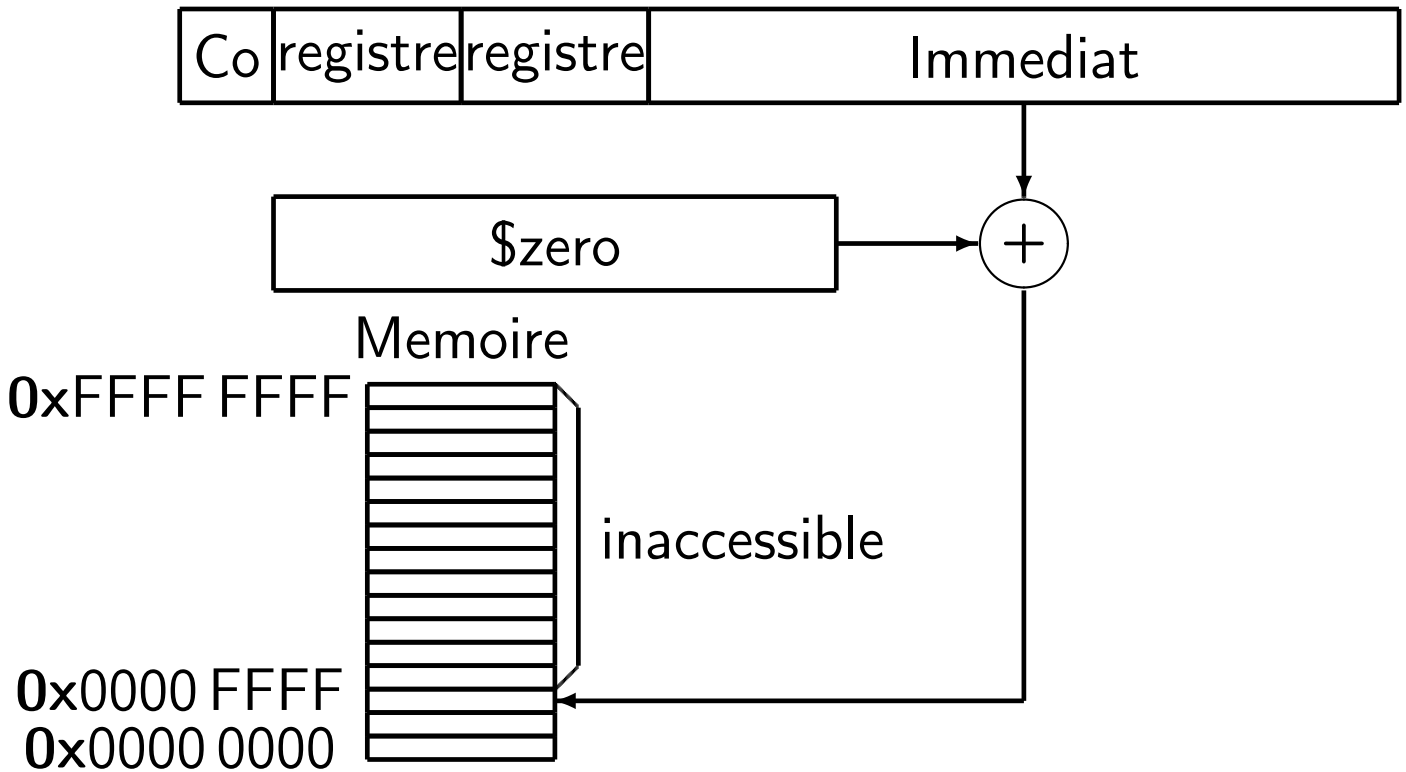


Figure 7 : Mode adressage direct

Accès a 64 Ko de données

Modes d'adressage indirect immédiat

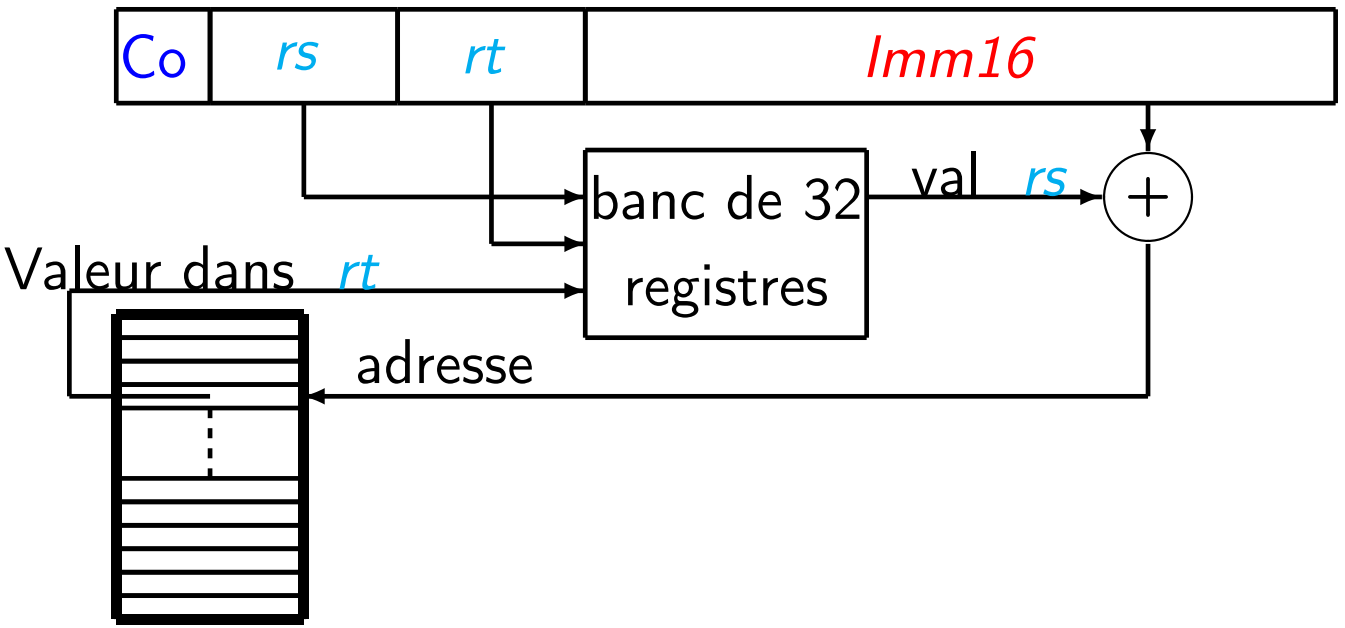


Figure 8 : Mode adressage indirect immdeiat

Accès a la mémoire complète

Instructions chargement et rangement

Rôle	Syntaxe	Opération	Codage
chargement 32 bits	lw rt,Imm16(rs)	rt ← [Imm16 + rs]	Co=0x23
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 8 bits	sb rt,Imm16(rs)	[Imm16+rs] ← rt	Co=0x28

- pour lw et sw l'adresse mémoire doit être divisible par 4 (alignement 32 bits - 4 octets)
- pour lb et sb l'adresse mémoire peut être quelconque
- début du segment de données à l'adresse **0x**1001 0000

Exemple chargement et rangement

Exemple : lw $t1 ,0x8($t0)

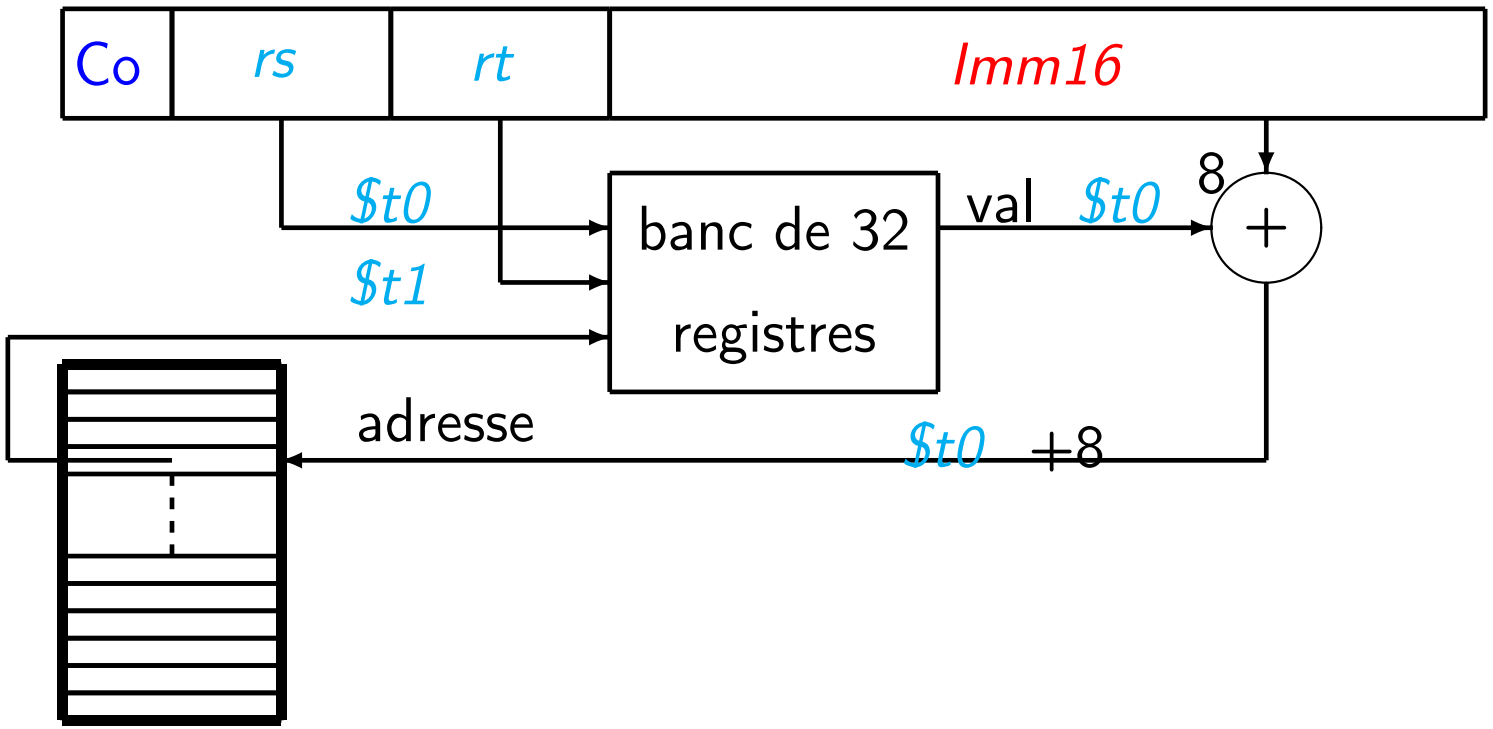
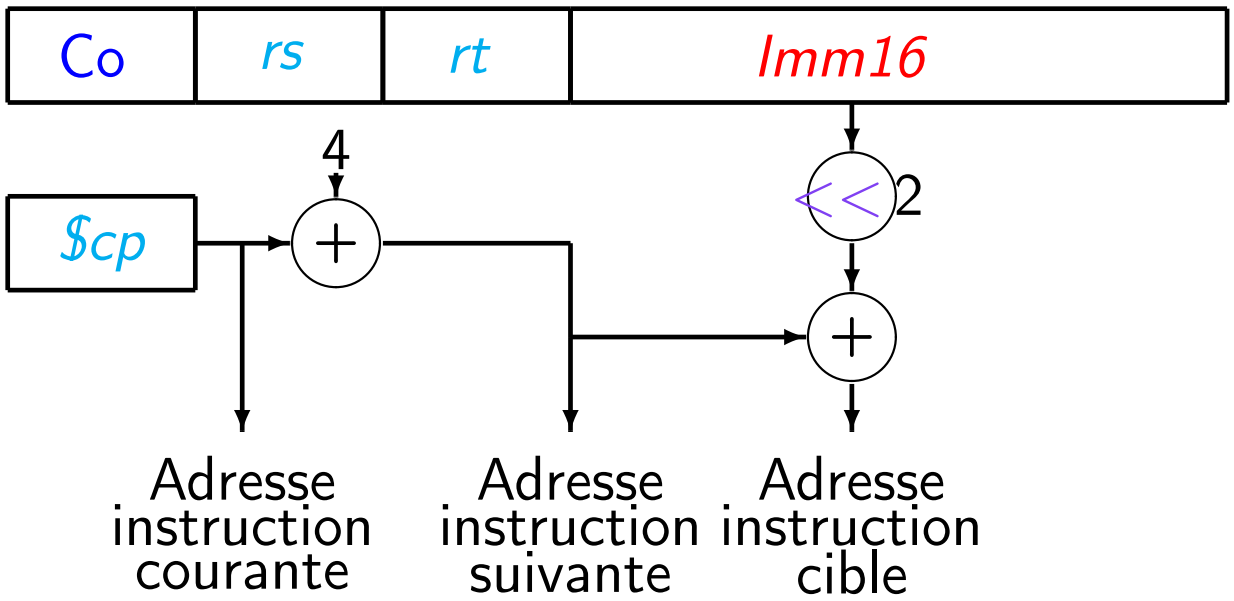


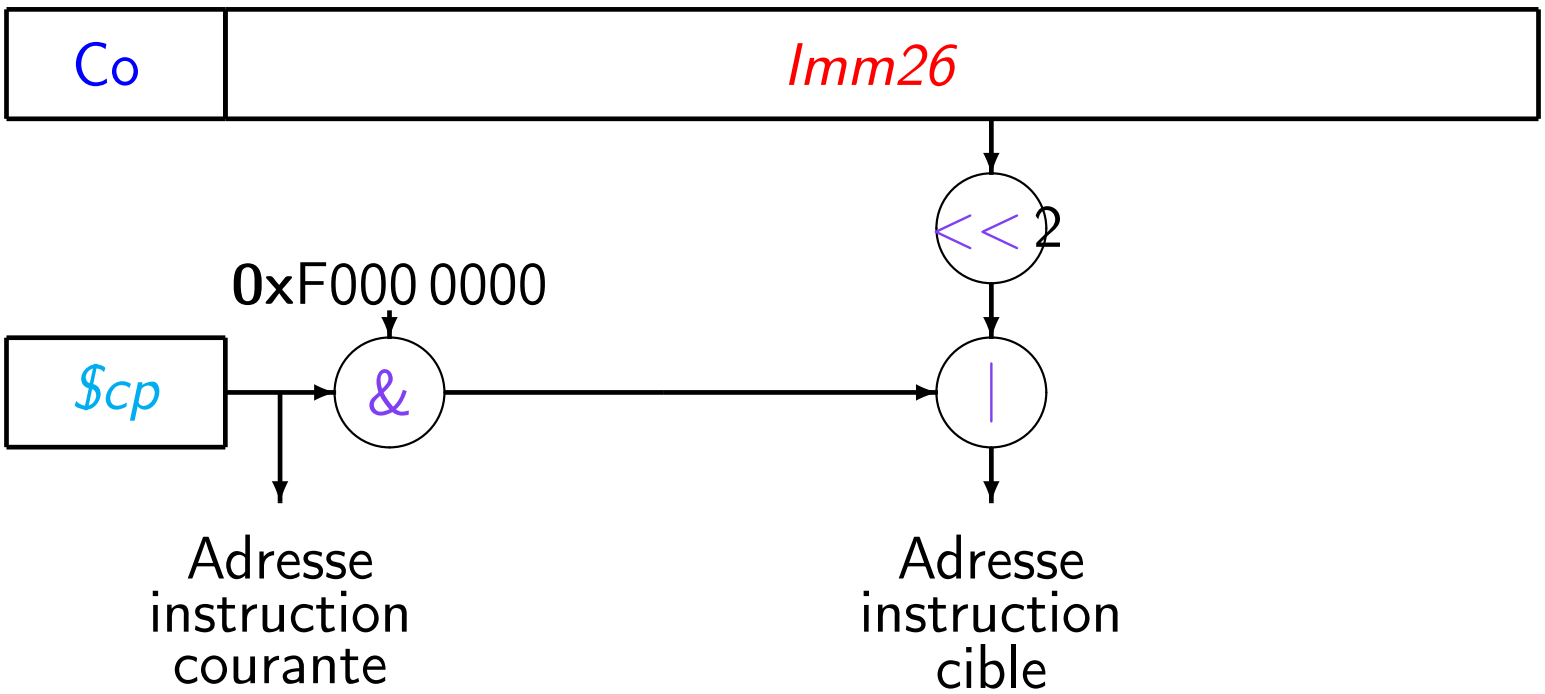
Figure 9 : Exemple Mode adressage indirect immédiat

Mode d'adressage relatif au compteur de programme



- instructions sur 32 bits, alignées sur les adresses divisibles par 4, se termine donc par **0b**00
- Adresse cible $\leftarrow$ *$cp* + (*Imm16* << 2)
- Accès à l'espace -2^{15} à $2^{15} - 1$ autour de *$cp* soit $\pm$ 32k instructions

Mode d'adressage immédiat 26 bits



- Adresse cible ← $cp & 0xF000 0000 + (Imm16 << 2)
- Accès au 64 Méga instructions du segment

Instructions de branchement

Rôle	Syntaxe	Opération	Codage
branchement =	beq rs,rt,Imm16	si rs=rt alors cp ← (cp+4)+(Imm16< < 2)	Co=4
branchement ≠	bne rs,rt,Imm16	si rs ≠ rt alors cp ← (cp+4)+(Imm16< < 2)	Co=5
branchement > 0	bgtz rs,Imm16	si *rs* > 0 alors cp ← (cp+4)+(Imm16< < 2)	*Co* = 7
branchement ≤ 0	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,Nf=8
appel de fonction	jal Imm26	cp ← cp & (0xF0000000) | (Imm26< < 2) puis $ra ← cp+4	Co=3

- etiquette : beq $zero,$zero,etiquette
- jal destination
- jr $ra
- début des instruction en **0x**0040 0000
- instructions débutent au adresses divisibles par 4
- *Imm16* représente la distance (en instructions)

Codage des instructions de branchement

- etiquette : beq $zero,$zero,etiquette. Si l'adresse de l'instruction est 0x0040 0000, le compteur de programme suivant vaut 0x0040 0004. Pour revenir sur l'instruction nous devons ajouter -4 $>>$ 2=-1, soit **0x**FFFF et Co=4

Co 6 bits	*rs* 5 bits	*rt* 5 bits	*Imm16* 16 bits
4	0	0	**0x**FFFF
0001 00	00 000	0 0000	1111 1111 1111 1111

0b 0001 0000 000 0 0000 1111 1111 1111 1111 $=$ **0x**1000 FFFF

Instructions diverses

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

Nom	v0	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		

```
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
```

Directives

1. etiquette : Instruction Opérande 1, Opérande 2, Opérande 3 #Commentaire
2. Directives (commence par .) :

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

Les directives ne sont pas traitées par le processeur mais par l'assembleur (elles ne produisent pas de code)

Hello World

Listing 2 – Hello World

```
1   .data #debute à l'adresse 0x1001 0000
2     Msg:    .asciiz "Hello World \n"
3   .text
4   main: #debute à l'adresse 0x0040 0000
5     #Affichage de la chaine de caractères
6     addi $v0, $zero, 4      # Sycall 4 afficher une chaine
7     la $a0, Msg             # Adresse de la chaine
8     syscall                 # Afficher maintenant
9     # Fin
10    addi $v0, $zero, 10
11    syscall
```

Alternative

Algorithme 1 Alternative

```
1:  Début
2:    $v0  ← 1
3:    $t0  ← 2
4:  Si  $v0  ==  $t0  Alors
5:      $a0  ← Adr "Egaux"
6:  Sinon
7:      $a0  ← Adr "Différent"
8:  FinSi
9:  Afficher chaîne pointée  $a0
10: Fin
```

```
                    .data
0x1001 0000         Egaux : .asciiz "Egaux"
0x1001 0005         Different : .asciiz "Differents"
                    .text
0x40 0000           addi  $t0 , $zero ,1
0x40 0004           addi  $t1 , $zero ,2
0x40 0008           bne  $t0 , $t1 ,Sinon
0x40 000C           la  $a0 ,Egaux
0x40 0014           j FinSi # Type J
                    Sinon :
0x40 0018           la  $a0 ,Differents
                    FinSi :
0x40 0020           # Afficher la chaîne ...
      Ou encore
0x40 0008           beq  $t0 , $t1 ,Sinon
0x40 000C           la  $a0 ,Differents
0x40 0014           beq  $zero , $zero , FinSi # Type I
                    Sinon :
0x40 0018           la  $a0 ,Egaux
                    FinSi :
```

Tant Que

Algorithme 2 Itération

1: **Début**
2: $a0 ← 5 # avec $a0 ≥ 0
3: $v0 ← 0
4: **Tant Que** $a0 ≠ 0 **Faire**
5: $v0 ← $v0 + $a0
6: $a0 ← $a0 - 1
7: **Fin Tant Que**
8: **Fin**

```
                .text
0x40 0000       addi  $a0 , $zero ,5
0x40 0004       addi  $v0 , $zero ,0
                TantQue :
0x40 0008       beq   $a0 , $zero ,FinTantQue
0x40 000C       add   $v0 , $v0 , $a0
0x40 0010       addi  $a0 , $a0 ,-1
0x40 0014       beq   $zero , $zero ,TantQue # Type J
                FinTantQue :
0x40 0018       # Afficher la  $v0

        Ou encore si  $a0  > 0
                TantQue :
0x40 0008       add   $v0 , $v0 , $a0
0x40 000C       addi  $a0 , $a0 ,-1
0x40 0010       beq   $a0 , $zero ,TantQue
```

Fonctions

- Paramètres dans les registres *$a0* à *$a3*
- Résultat dans les registres *$v0* et *$v1*
- Possibilité de modifier les registres temporaires *$ti*
- Appel de la fonction avec l'instruction jal
- Retour de la fonction avec l'instruction jr *$ra*
- Si fonctions imbriquées, sauvegarder la valeur de *$ra* avant l'appel des fonctions internes.
- Pas de mécanisme de récursivité
- Si nombre de paramètres insuffisant, les placer dans la pile (hors cours)

Fonctions avec jal

```
            .text
0x40 0000   addi  $a0 , $zero ,5
0x40 0004   jal SommeNpremiers
0x40 0008   # Afficher résultat ...
...         # suite et Fin du programme IMPORTANT
            SommeNpremiers :
0x40 0030   add  $v0 , $v0 , $a0
0x40 0034   addi  $a0 , $a0 ,-1
0x40 0038   beq  $a0 , $zero ,TantQue
0x40 003C   jr  $ra
```

Fonctions- 2

Déroulement du programme précédent.

Adresse	Instruction	*$cp*	*$ra*
0x40 0000	addi *$a0* , *$zero* ,5	**0x**40 0000	?
0x40 0004	jal SommeNpremiers	**0x**40 0004	?
0x40 0030	add *$v0* , *$v0* , *$a0*	**0x**40 0034	**0x**40 0008
0x40 003C	jr *$ra*	**0x**40 0034	**0x**40 0008
0x40 0008	# Afficher résultat ...	**0x**40 0008	**0x**40 0008

Fonctions sans jal et sans la

```
          .text
0x40 0000   addi  $a0 , $zero ,5
0x40 0004   lui  $ra , 0x0040
0x40 0008   ori  $ra , $ra ,0x18 # Adresse retour 0x0040 0018
0x40 000C   lui  $t0 , 0x0040
0x40 0010   ori  $t0 , $t0 ,0x30 # Adresse fonction 0x0040 0030
0x40 0014   jr  $t0
0x40 0018   # Afficher résultat- retour de la fonction
...         # suite et Fin du programme IMPORTANT
          .text 0x40 0030
0x40 0030 SommeNpremiers :
0x40 0034   add  $v0 , $v0 , $a0
0x40 0038   addi  $a0 , $a0 ,-1
0x40 003C   beq  $a0 , $zero ,TantQue
0x40 0040   jr  $ra
```