Master1 - SEM

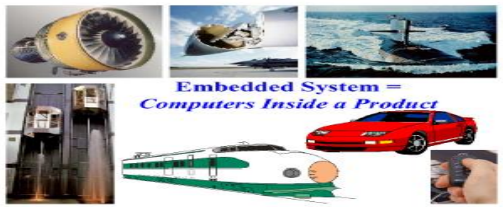
Processeurs Embarqués

Cours 3.2

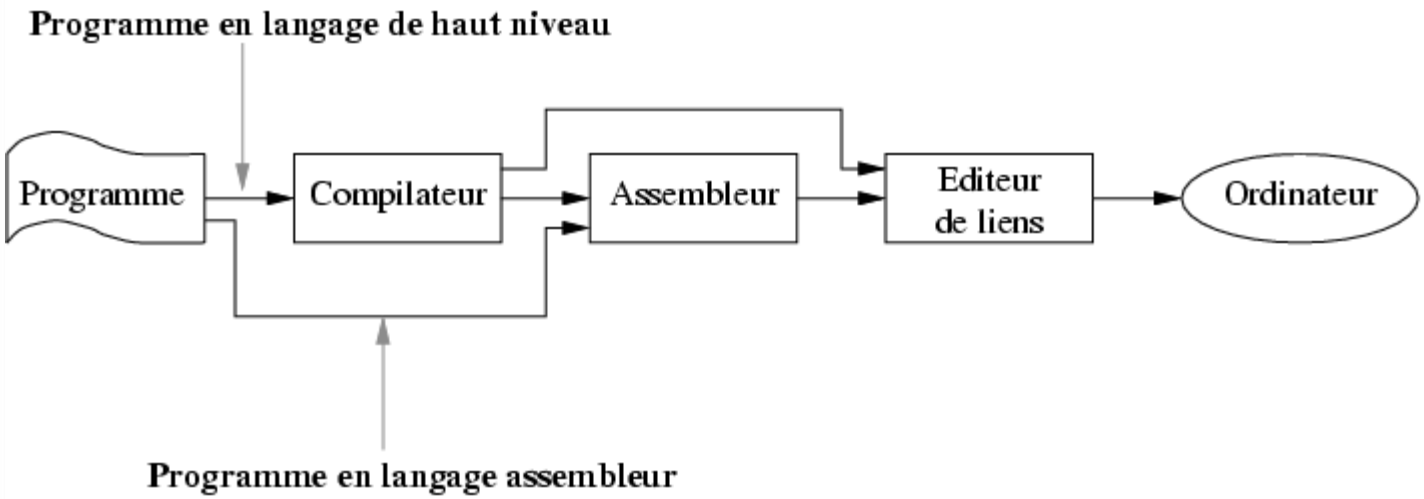
Conception de processeur : chemin de données

Année : 2022-2023

Pr R. BOUDOUR

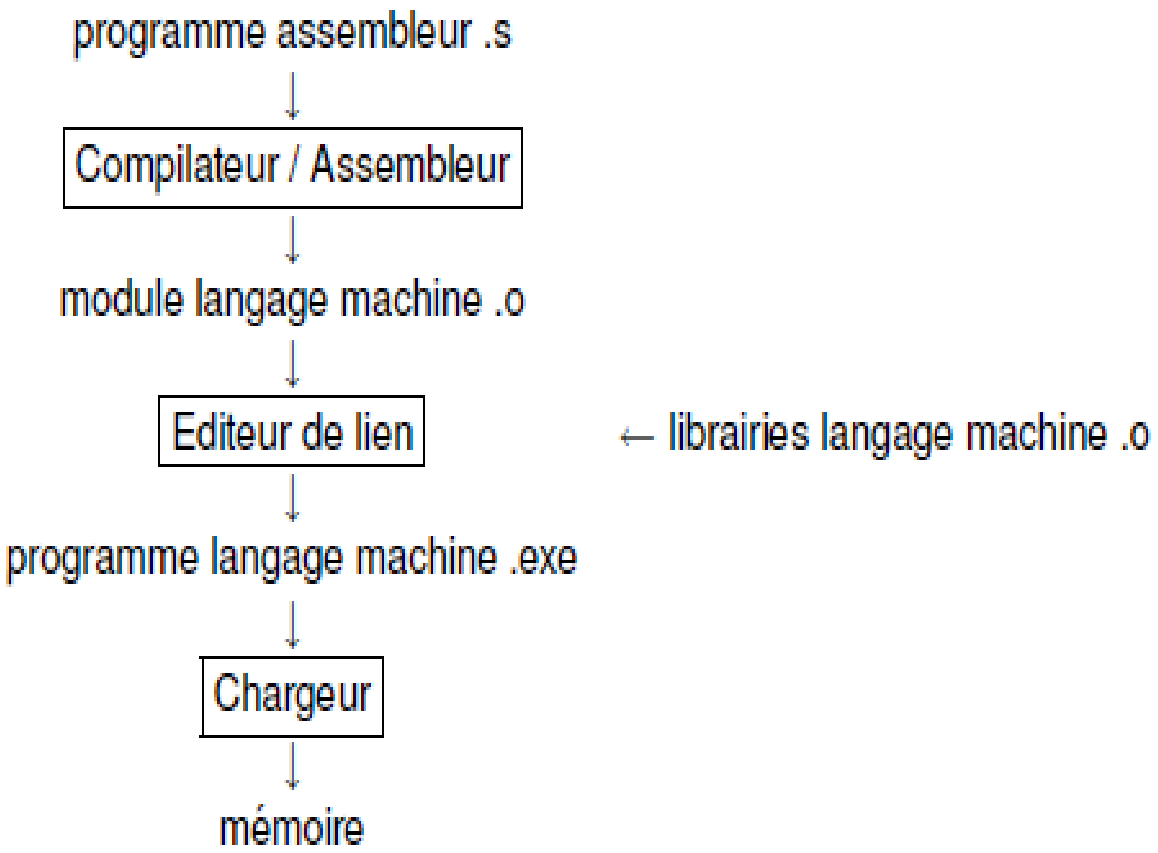


Qui utilise le langage d'assemblage ?



Le langage assembleur est soit écrit par un programmeur, soit généré par un compilateur.

Du langage d'assemblage à l'exécution



Quand utiliser le langage d'assemblage ?

- L'architecture externe représente ce que doit connaitre :
  - un programmeur souhaitant programmer en assembleur,
  - ou la personne souhaitant écrire un compilateur pour ce processeur:
  - Les registres visibles.
  - L'adressage de la mémoire.
  - Le jeu d'instructions.
  - Les mécanismes de traitement des interruptions et exceptions

Quand utiliser le langage d'assemblage ?

- Quand la vitesse d'un programme particulier est extrêmement importante
- Quand la taille d'un programme particulier est extrêmement importante

$\Longrightarrow$ *ordinateur embarqué*

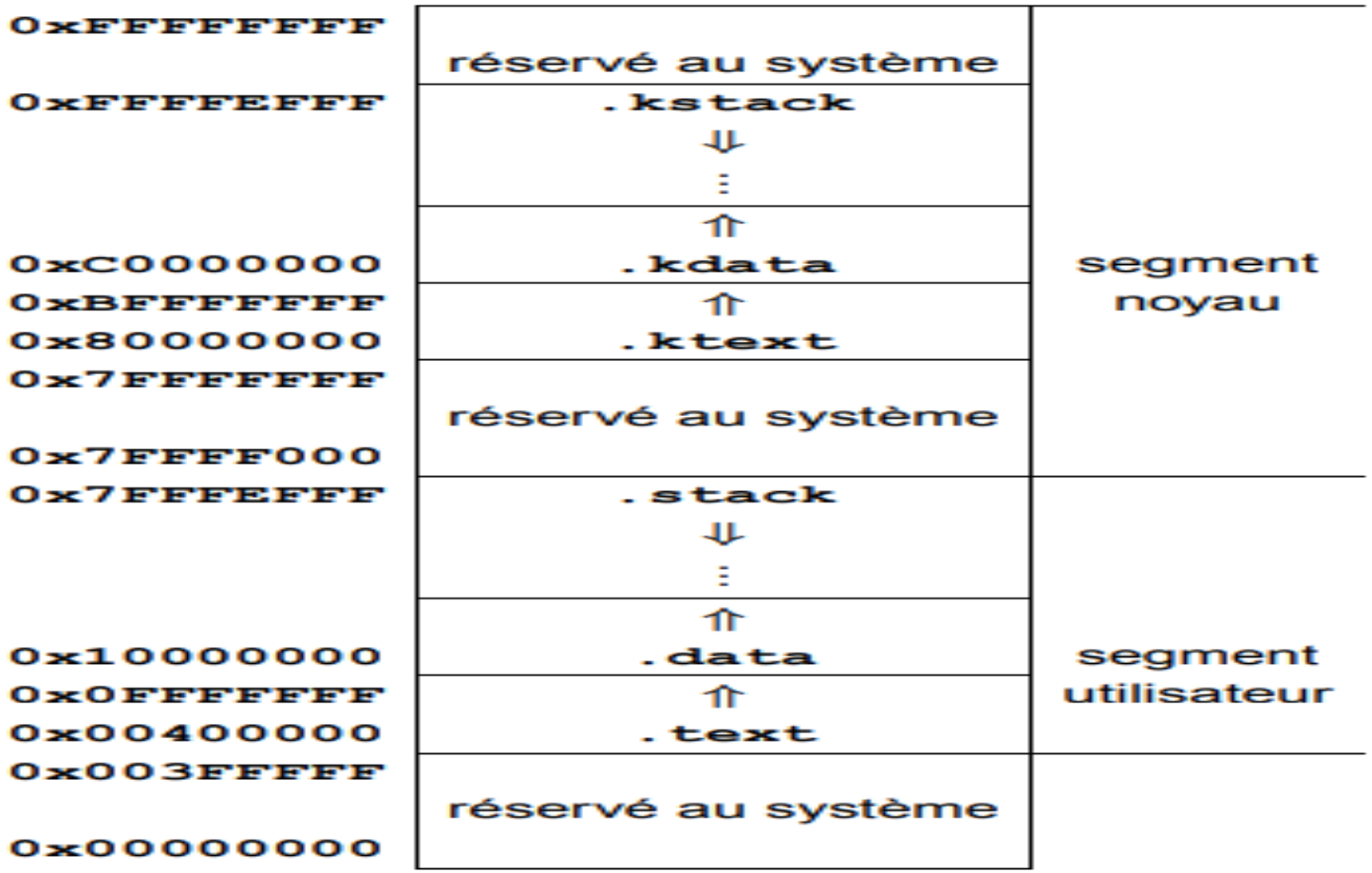
- Approche hybride :
  - la plus grande partie du programme en langage de haut niveau
  - les sections critiques en langage assembleur

MIPS R3000

Registres généraux de MIPS R3000

- Ri (0 ≤ i ≤ 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.

Organisation de la mémoire



Organisation de la mémoire

- Dans l'architecture MIPS R3000, l'espace adressable est divisé en deux segments : **le segment utilisateur, et le segment noyau**.
- Un programme utilisateur utilise généralement trois sous-segments (appelés sections) dans le segment utilisateur :
  - la section **text** contient le code exécutable en mode utilisateur. Elle est implantée conventionnellement à l'adresse 0x00400000. Sa taille est fixe et calculée lors de l'assemblage.
  - la section **data** contient les données globales manipulées par le programme utilisateur. Elle est implantée conventionnellement à l'adresse 0x10000000.
  - la section **stack** contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse 0x7FFFEFFF. Contrairement aux sections data et text, la pile s'étend vers les adresses décroissantes

Organisation de la mémoire

- Trois autres sections sont définies dans le **segment noyau** :
  - la section **ktext** contient le code exécutable en mode noyau. Elle est implantée conventionnellement à l'adresse 0x80000000. Sa taille est fixe et calculée lors de l'assemblage ;
  - la section **kdata** contient les données globales manipulées par le système d'exploitation en mode noyau. Elle est implantée conventionnellement à l'adresse 0xC0000000. Sa taille est fixe et calculée lors de l'assemblage ;
  - la section **kstack** contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse 0xFFFFEFFF. Contrairement aux sections data et text, la pile s'étend vers les adresses décroissantes.

Jeu d'instructions MIPS

- MIPS : *Microprocessor without Interlocked Pipeline Stages*
- Le processeur MIPS R3000 possède 57 instructions qui se répartissent en 4 classes :

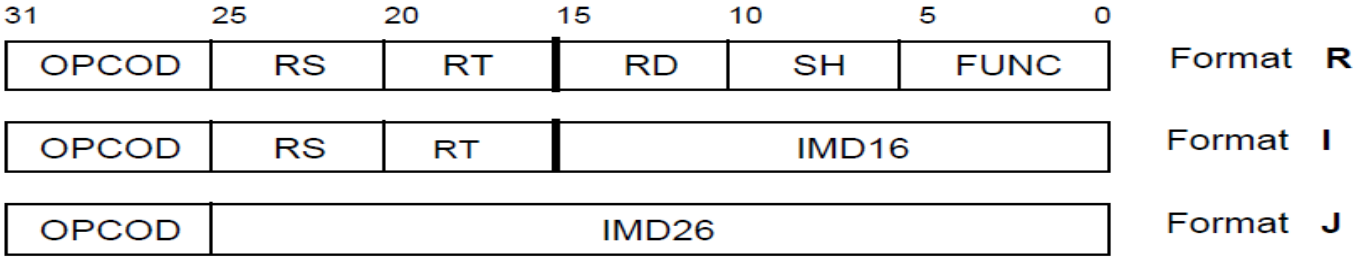
  33 instructions arithmétiques/logiques entre registres

  12 instructions de branchement

  7 instructions de lecture/écriture mémoire

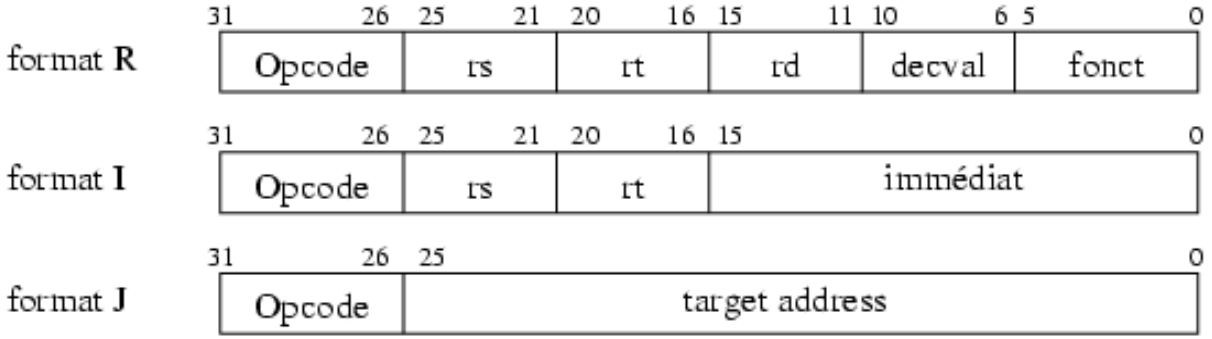
  5 instructions systèmes

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



Les formats des instructions MIPS

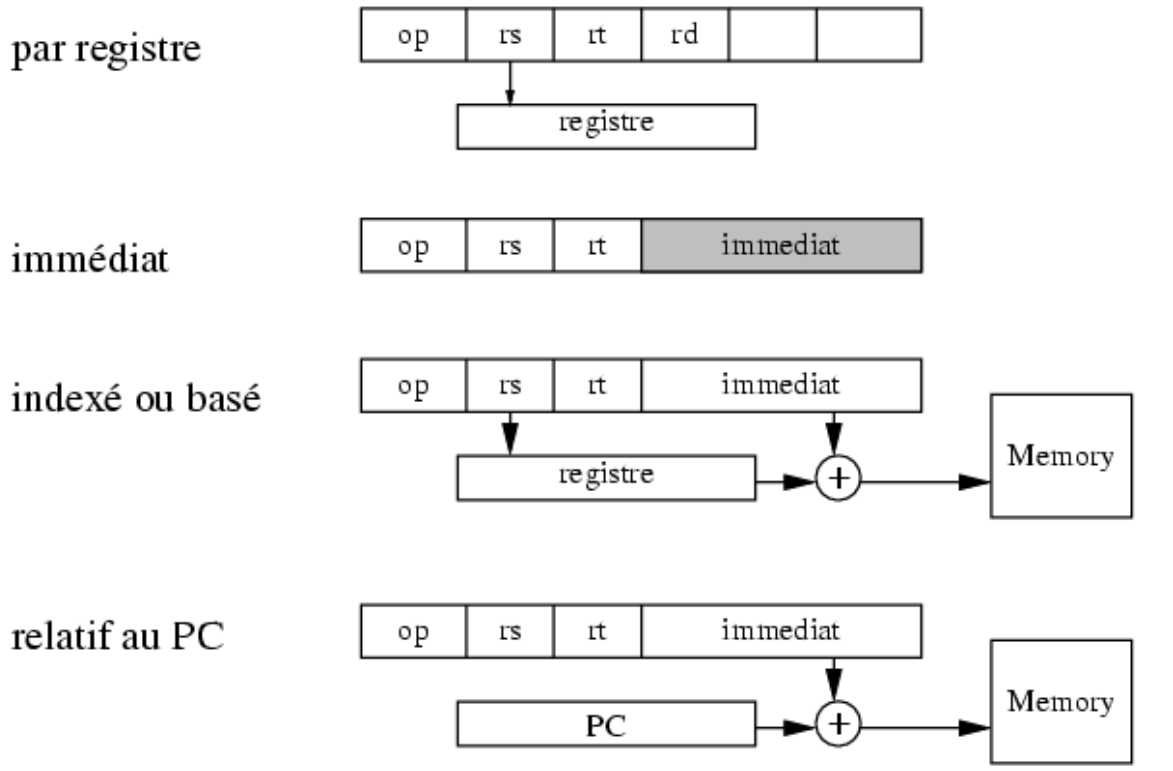
- Toutes les instructions MIPS ont 32 bits de long



- Les différents champs sont :
  - **opcode : opération de l'instruction**
  - **rs, rt, rd : registres sources et destination**
  - **dec : le nombre de décalages**
  - **fonct : les variantes des opérations**
  - **immédiat : valeur immédiate ou de déplacement d'adresse**
  - **target address : adresse cible pour les jump**

Modes d'adressage du MIPS

- Les modes d'adressage MIPS dans les différents modèles



Format d'instructions I

Format I :

	op	*rs*	*rt*	*16 bits*
`lui $1, 100`	15	0	1	100
`lw $1, 100($2)`	35	2	1	100
`sw $1, 100($2)`	43	2	1	100
`beq $1, $2 ,100`	4	1	2	100
`bne $1, $2 ,100`	5	1	2	100

Codage des adresses

Les adresses dans les instructions ne sont **pas sur 32 bits** !

- Pour les instructions de type I : 16 bits

  $\Longrightarrow$ Adresse = PC + signé(16 bits) * 4 **adressage relatif**

- Pour les instructions de type J : 26 bits

  $\Longrightarrow$ On obtient l'adresse d'un mot mémoire (de 32 bits) en ajoutant devant les 26 bits les 4 bits de poids fort de PC (Il faut multiplier par 4 pour l'adresse d'un octet)

  adressage direct restreint

Format d'instructions R

Format R :

	op	*rs*	*rt*	*rd*	*sa*	*fu*
`add $1,$2,$3`	0	2	3	1	0	32
`sub $1,$2,$3`	0	2	3	1	0	34
`slt $1,$2,$3`	0	2	3	1	0	42
`jr $31`	0	31	0	0	0	8

- `sub $1,$2,$3` : soustrait `$3` de `$2` et place le résultat dans `$1`.
- `slt $1,$2,$3` (set less than) : met `$1` à 1 si `$2` est inférieur à `$3` et à 0 sinon.

Codage des opérations

- Le codage des instructions est principalement défini par les 6 bits du champs code opération de l'instruction (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")
  - COPRO (valeur "010000")

INS 28 : 26

INS 31 : 29	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								

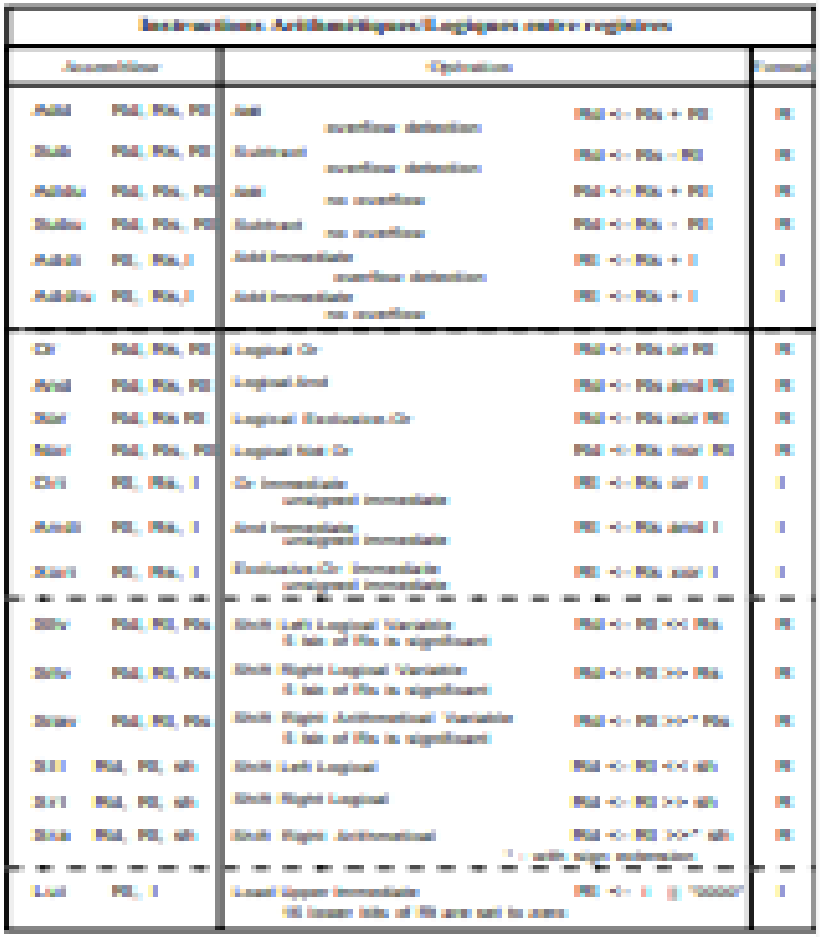
Codage des opérations

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								

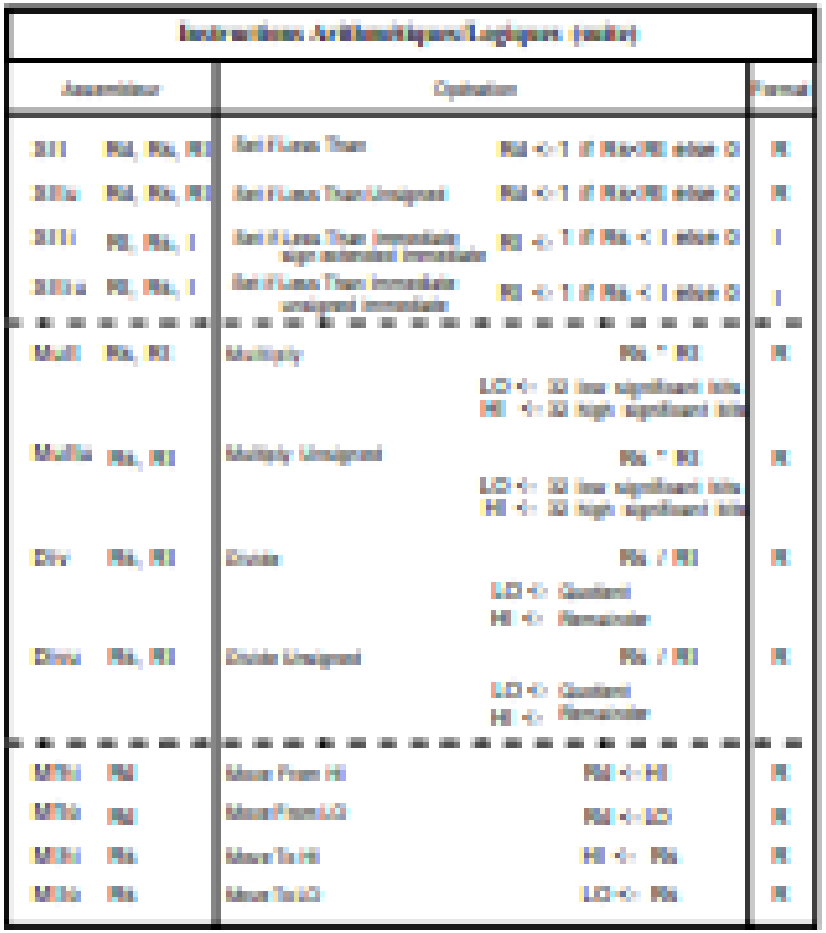
- Lorsque le code opération a la valeur SPECIAL ("000000"),
- Il faut analyser les 6 bits de poids faible de l'instruction (5:0):

Jeu d'instructions



Instructions Arithmétiques/Logiques entre registres

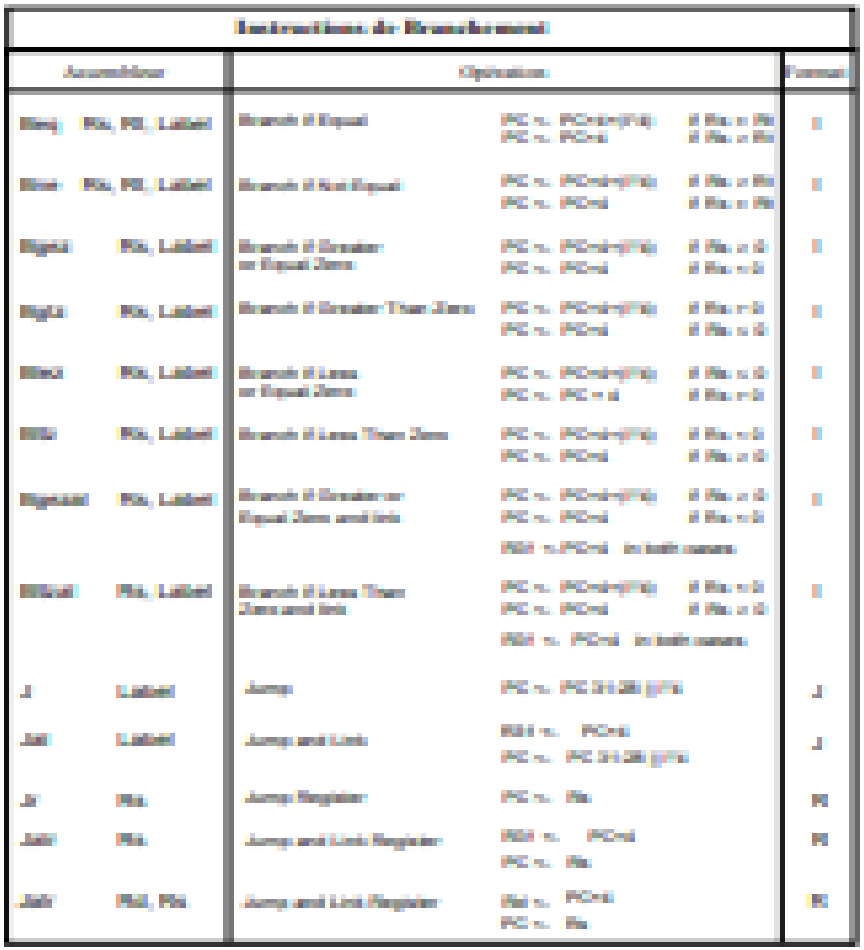
Assembleur		Opération		Format
Add	Rd, Rs, Rt	Add avec/avec détection	Rd <- Rs + Rt	R
Sub	Rd, Rs, Rt	Soustraire avec/avec détection	Rd <- Rs - Rt	R
Addu	Rd, Rs, Rt	Add sans exception	Rd <- Rs + Rt	R
Subu	Rd, Rs, Rt	Soustraire sans exception	Rd <- Rs - Rt	R
Addi	Rt, Rs, I	Add Immediate avec/avec détection	Rt <- Rs + I	I
Addiu	Rt, Rs, I	Add Immediate sans exception	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 bits of Rs is significant	Rd <- Rt << Rs	R
Srlv	Rd, Rt, Rs	Shift Right Logical Variable 5 bits of Rs is significant	Rd <- Rt >> Rs	R
Srav	Rd, Rt, Rs	Shift Right Arithmetical Variable 5 bits 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	R
		* : with sign extension		
Lui	Rt, I	Load Upper Immediate 16 lower bits of Rt are set to zero	Rt <- I * 65536	I



Instructions Arithmétiques/Logiques (suite)

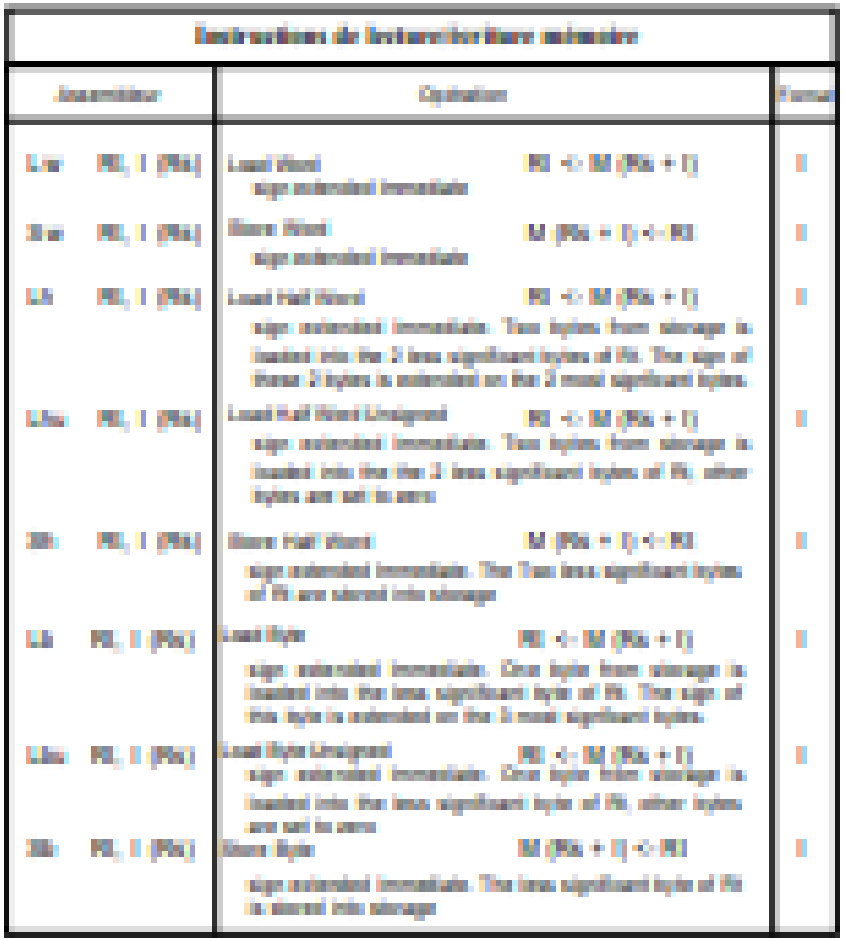
Assembleur		Opération		Format
Slt	Rd, Rs, Rt	Set if Lower Than	Rd <- 1 if Rs<Rt else 0	R
Sltu	Rd, Rs, Rt	Set if Lower Than Unsigned	Rd <- 1 if Rs<Rt else 0	R
Slti	Rt, Rs, I	Set if Lower Than Immediate sign extended Immediate	Rt <- 1 if Rs < I else 0	I
Sltiu	Rt, Rs, I	Set if Lower Than Immediate unsigned 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

Jeu d'instructions



Instructions de Branchement

Assembleur	Opération	Format
Beq Rs, Rt, Label	Branch if Equal	I
Bne Rs, Rt, Label	Branch if Not Equal	I
Bgez Rs, Label	Branch if Greater or Equal Zero	I
Bgtz Rs, Label	Branch if Greater Than Zero	I
Blez Rs, Label	Branch if Less or Equal Zero	I
Bltz Rs, Label	Branch if Less Than Zero	I
Bgezal Rs, Label	Branch if Greater or Equal Zero and link	I
Bltzal Rs, Label	Branch if Less Than Zero and link	I
J Label	Jump	J
Jal Label	Jump and Link	J
Jr Rs	Jump Register	R
Jalr Rs	Jump and Link Register	R
Jalr Rd, Rs	Jump and Link Register	R



Instructions de lecture/écriture mémoire

Assembleur	Opération	Format
Lw Rt, 1 (Rs)	Load Word sign extended Immediate	I
Sw Rt, 1 (Rs)	Store Word sign extended Immediate	I
Lh Rt, 1 (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 significant bytes.	I
Lhu Rt, 1 (Rs)	Load Half Word Unsigned sign extended Immediate. Two bytes from storage is loaded into the the 2 less significant bytes of Rt, other bytes are set to zero	I
Sh Rt, 1 (Rs)	Store Half Word sign extended Immediate. The Two less significant bytes of Rt are stored into storage	I
Lb Rt, 1 (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 significant bytes.	I
Lbu Rt, 1 (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	I
Sb Rt, 1 (Rs)	Store Byte sign extended Immediate. The less significant byte of Rt is stored into storage	I

Jeu d'instructions

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 Rt <- M (Rs + I) 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.	I
Lhu	Rt, I (Rs)	Load Half Word Unsigned Rt <- M (Rs + I) sign extended Immediate. Two bytes from storage is loaded into the the 2 less significant bytes of Rt, other bytes are set to zero	I
Sh	Rt, I (Rs)	Store Half Word M (Rs + I) <- Rt sign extended Immediate. The Two less significant bytes of Rt are stored into storage	I
Lb	Rt, I (Rs)	Load Byte Rt <- M (Rs + I) 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.	I
Lbu	Rt, I (Rs)	Load Byte Unsigned Rt <- M (Rs + I) sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt, other bytes are set to zero	I
Sb	Rt, I (Rs)	Store Byte M (Rs + I) <- Rt sign extended Immediate. The less significant byte of Rt is stored into storage	I

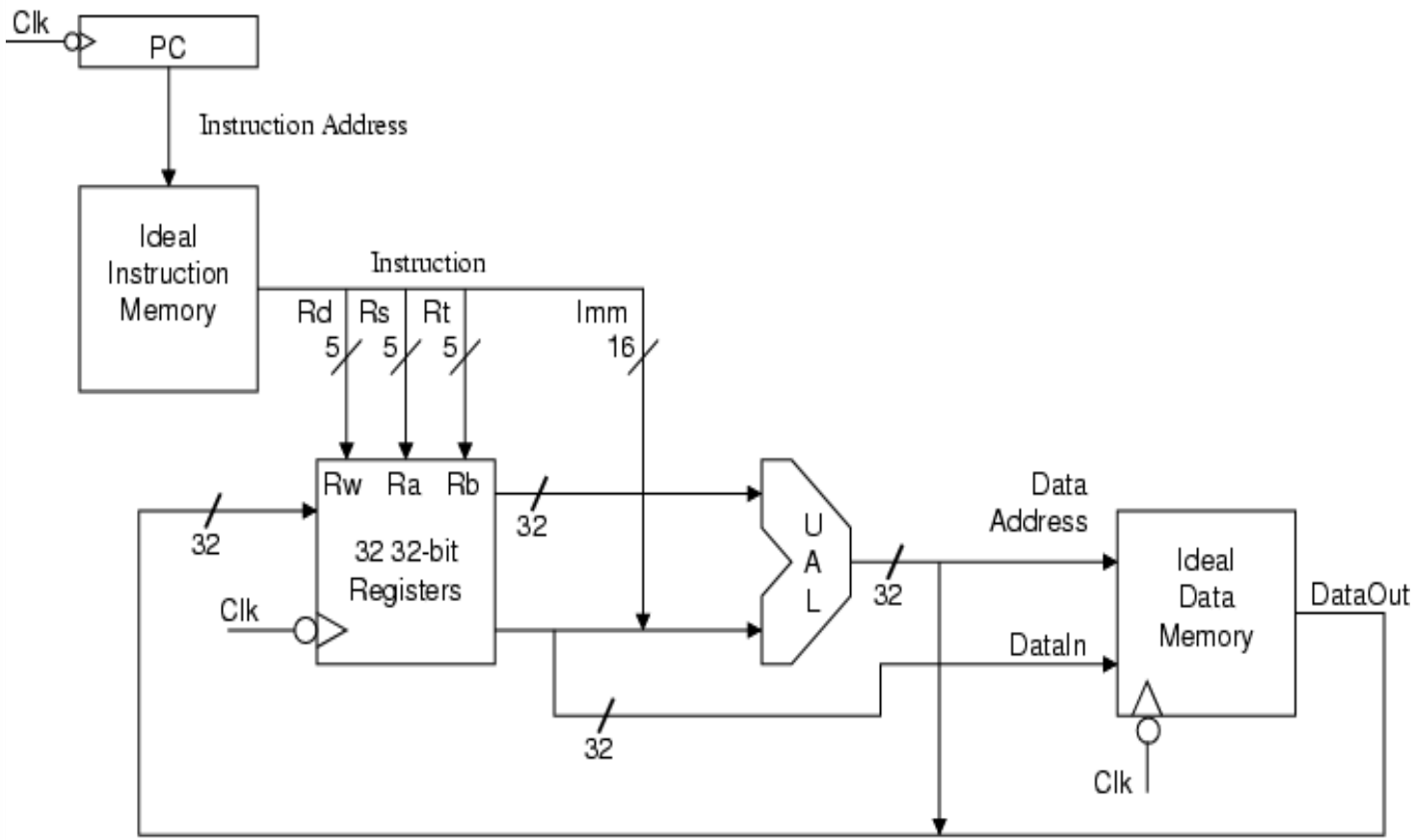
Aspects de performance

- Les performances d'une machine sont déterminées par :
  - le temps de cycle de l'horloge
  - le nombre de cycles par instruction
- La conception du chemin de données et du contrôle détermine :
  - le temps de cycle de l'horloge
  - le nombre de cycles par instruction

Sous-ensemble de MIPS mis en œuvre

- **Addition et Soustraction :**
  - `add rd, rs, rt`
  - `sub rd, rs, rt`
- **Ou immédiat :**
  - `or rt, rs, imm16`
- **Chargement et Rangement :**
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`
- **Branchement :**
  - `beq rs, rt, imm16`
- **Saut :**
  - `j target`

Vue abstraite de l'implémentation



Etapes de conception d'un processeur

- Architecture du jeu d'instruction ⇒ Langage "Transfert de registres" (RTL)
- Langage "Transfert de registres" ⇒
  - composants du chemin de données
  - interconnection des composants
- composants du chemin de données ⇒ Signaux de contrôle
- Signaux de contrôle ⇒ Logique de contrôle

Register Transfer Language (RTL)

- RTL est une représentation intermédiaire d'architecture dépendante proche du langage assembleur principalement utilisée dans les compilateurs modernes. Il est aussi bien implémenté dans des compilateurs certifiés
- En informatique, un **langage intermédiaire** est le langage d'une machine abstraite conçu pour l'analyse d'un programme informatique. Le terme vient de son utilisation dans les compilateurs, où un compilateur transcrit d'abord le code source d'un programme en une forme plus adaptée pour les transformations d'amélioration de code, comme un état intermédiaire avant de générer du code objet ou du langage machine pour une machine cible.

RTL : Instruction ADD

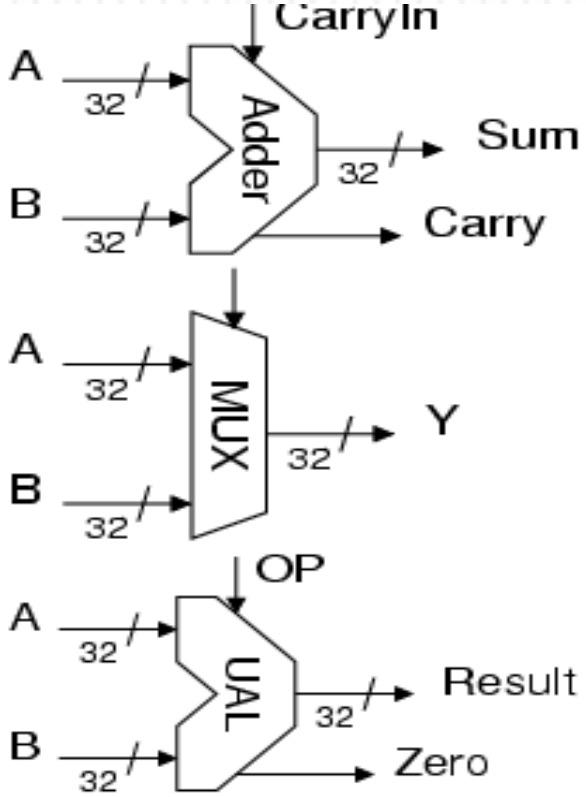
- **add rd, rs, rt**
  - `mem[PC]` — Extraire l'instruction de la mémoire
  - `R[rd] ← R[rs] + R[rt]` — Opération d'addition
  - `PC ← PC + 4` — Calcul de l'adresse de la prochaine instruction

RTL : Instruction Load

- **lw rt, rs, imm16**
  - `mem[PC]` — Extraire l'instruction de la mémoire
  - `Addr ← R[rs] + SignExt(imm16)` — Calcul de l'adresse mémoire
  - `R[rt] ← Mem[Addr]` — Charger la donnée dans le registre
  - `PC ← PC + 4` — Calcul de l'adresse de la prochaine instruction

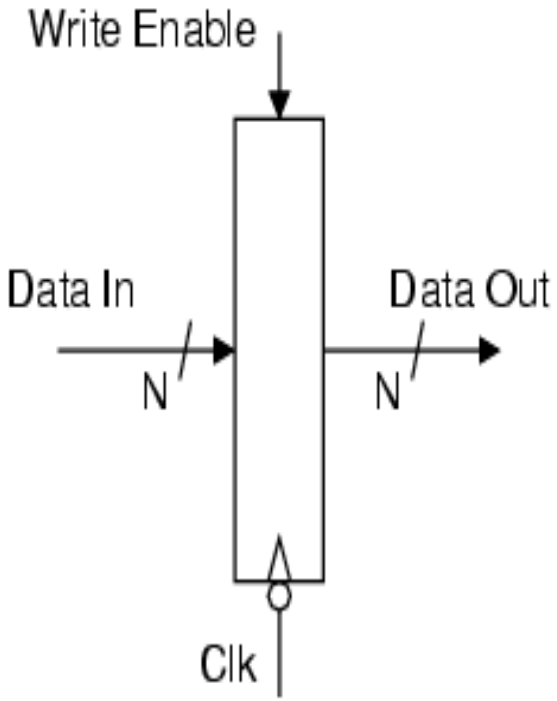
Eléments combinatoires

- **Adder**
- **MUX**
- **UAL**



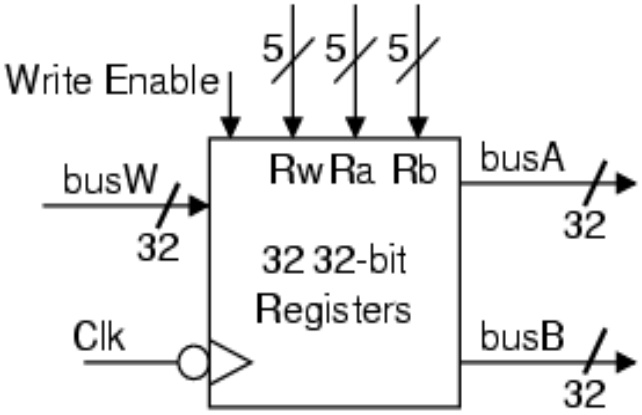
Elément de mémoire : Registre

- Registre
  - Similaire aux bascules D mais
    - N-bits en entrée et en sortie
    - une entrée “Write enable”
  - Write enable :
    - 0 : **DataOut** ne change pas
    - 1 : **DataOut** devient **DataIn**



Elément de mémoire : banc de registres

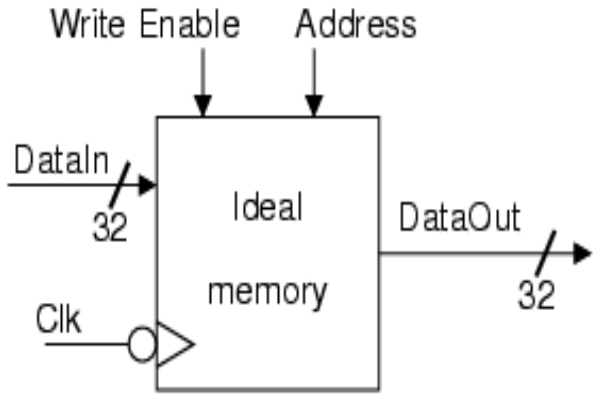
- Banc de registres = 32 Registres
  - 2 bus de sortie de 32 bits ; **busA** et **busB**
  - un bus d'entrée de 32 bits ; **busW**



- Registre sélectionné par :
  - **RA** selectionne le registre pour le **busA**
  - **RB** selectionne le registre pour le **busB**
  - **RW** selectionne le registre à écrire via le **busW** quant **WriteEnable** est à 1

Elément de mémoire : Mémoire idéale

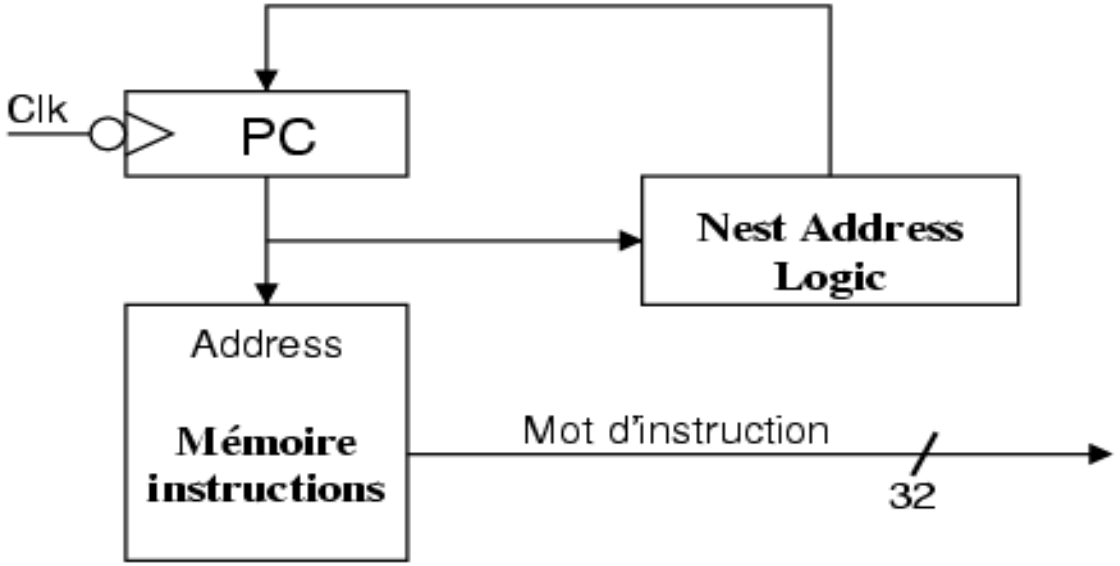
- Mémoire (idéale)
  - Un bus de sortie de 32 bits ; **DataOut**
  - Un bus d'entrée de 32 bits ; **DataIn**



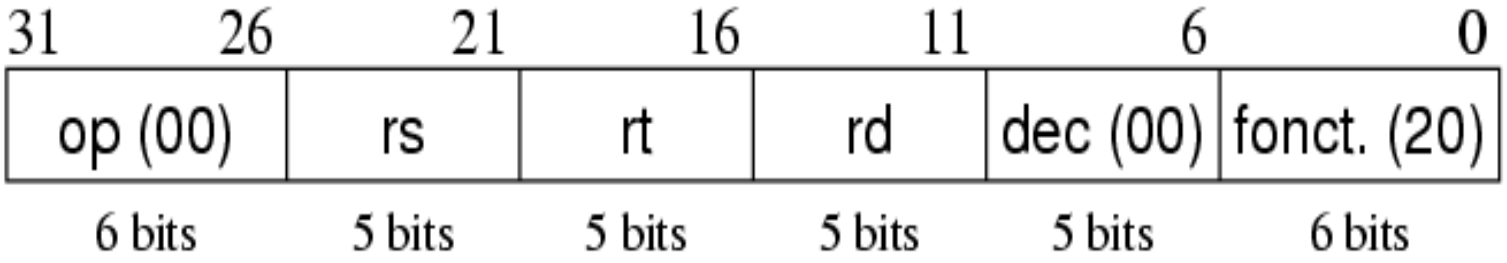
- Un mot mémoire est sélectionné par :
  - **Address** sélectionne le mot pour le bus **DataOut**
  - Si **WriteEnable** $= 1$; **Address** sélectionne le mot mémoire à écrire via le bus **DataIn**

Unité 'Fetch'

- extrait l'instruction : **Mem[PC]**
- met à jour le PC : $PC \leftarrow PC + 4$ (séquentiel) ou $PC \leftarrow$ "*autre*" (branch, jump)

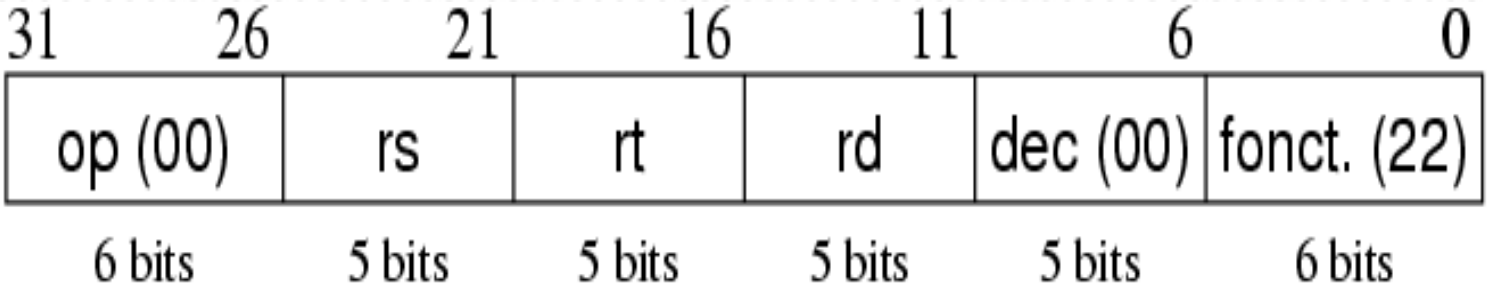


RTL : Instruction Add



- **add rd, rs, rt**
  - `mem[PC]` Extraire l'instruction
  - `R[rd] ← R[rs] + R[rt]` Opération d'addition
  - `PC ← PC + 4` Calcul de l'adresse de la prochaine instruction

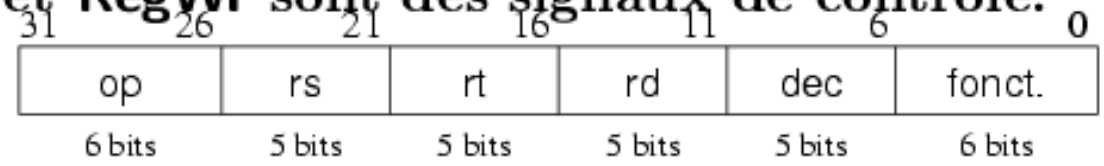
RTL : Instruction Sub

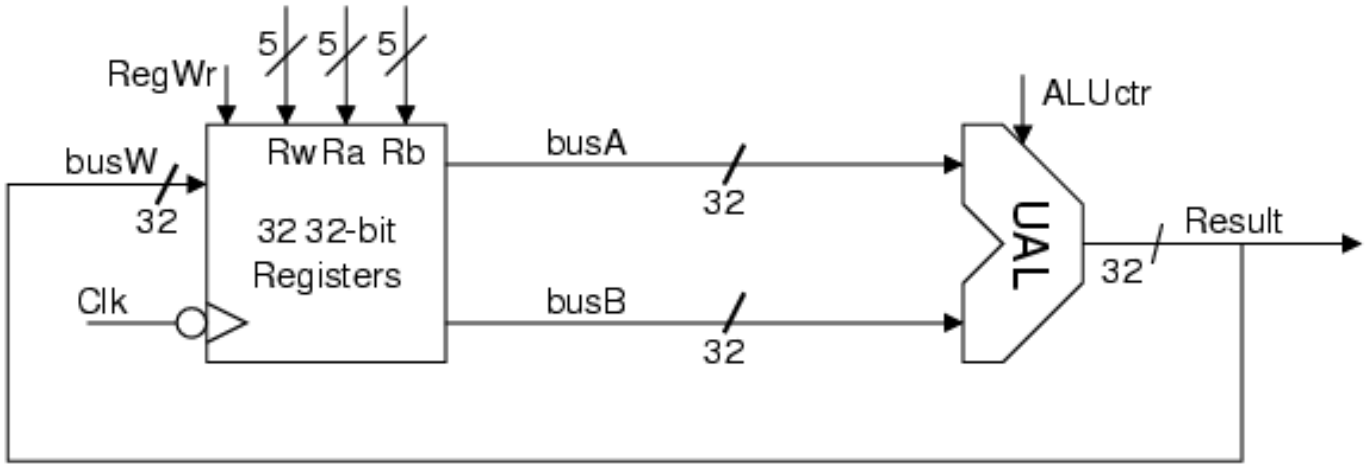


- **sub rd, rs, rt**
  - `mem[PC]` Extraire l'instruction
  - `R[rd] ← R[rs] - R[rt]` Opération de soustraction
  - `PC ← PC + 4` Calcul de l'adresse de la prochaine instruction

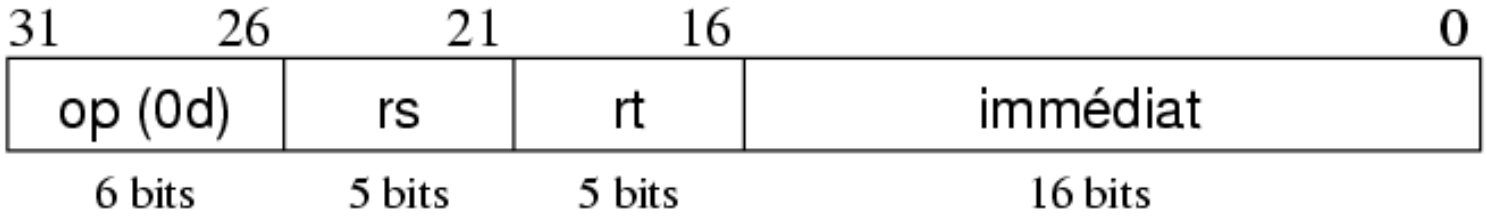
Chemin de données pour opérations Registre-Registre

- **Ra, Rb** et **Rw** sont les champs instructions rs, rt et rd.
- **ALUctr** et **RegWr** sont des signaux de contrôle.

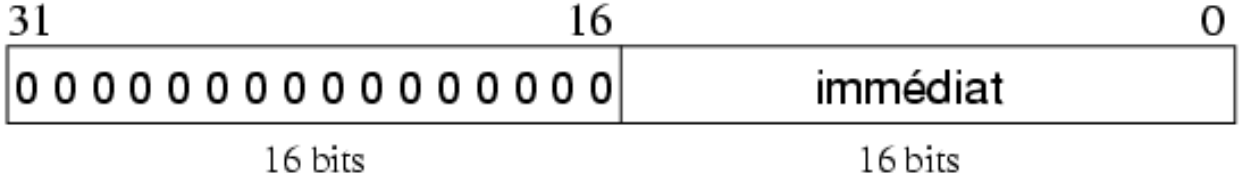




RTL : Instruction Ori (Ou immediat)

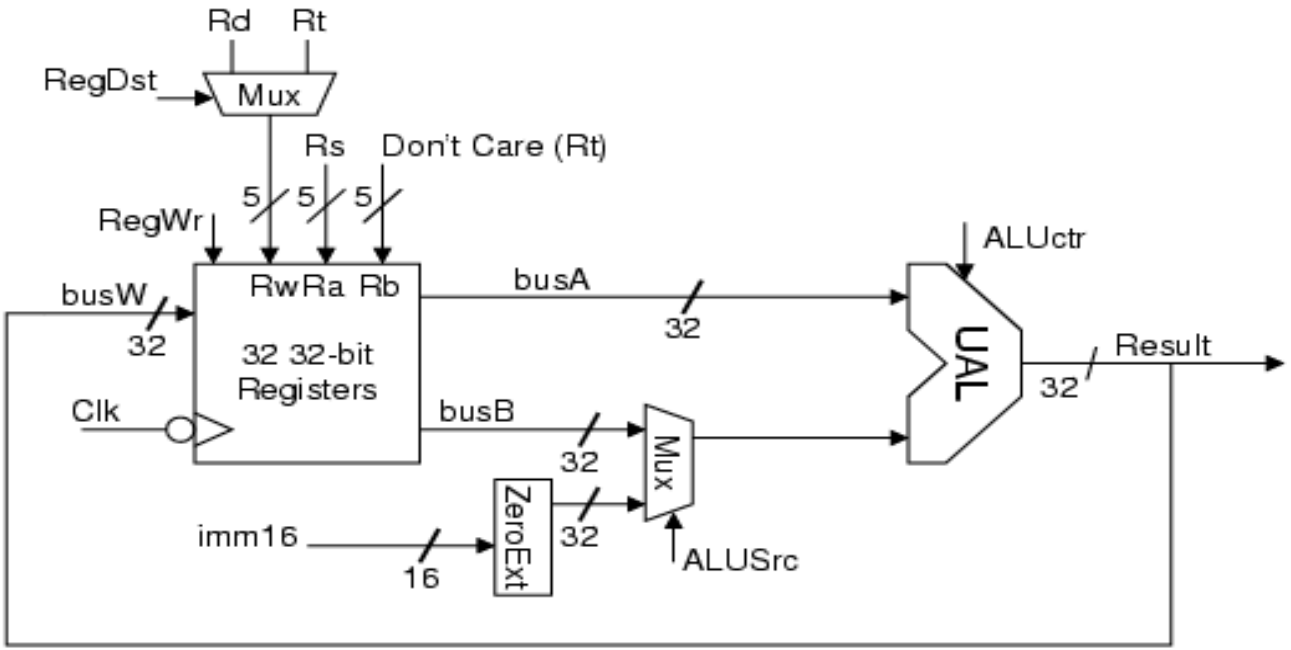


- **ori rt, rs, imm16**
  - `mem[PC]` **Extraire l'instruction**
  - `R[rt] ← R[rs] or ZeroExt(Imm16)` **Opération ou**
  - `PC ← PC + 4` **addr instruction suivante**

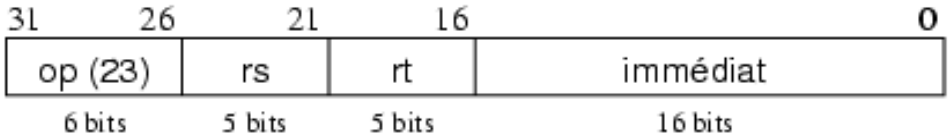


Opérations logiques avec ou immédiat

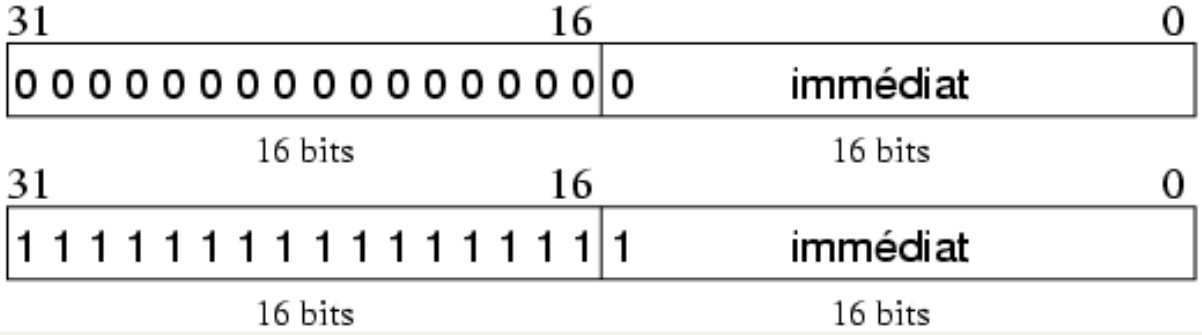
- R[rt] ← R[rs] op ZeroExt(Imm16)



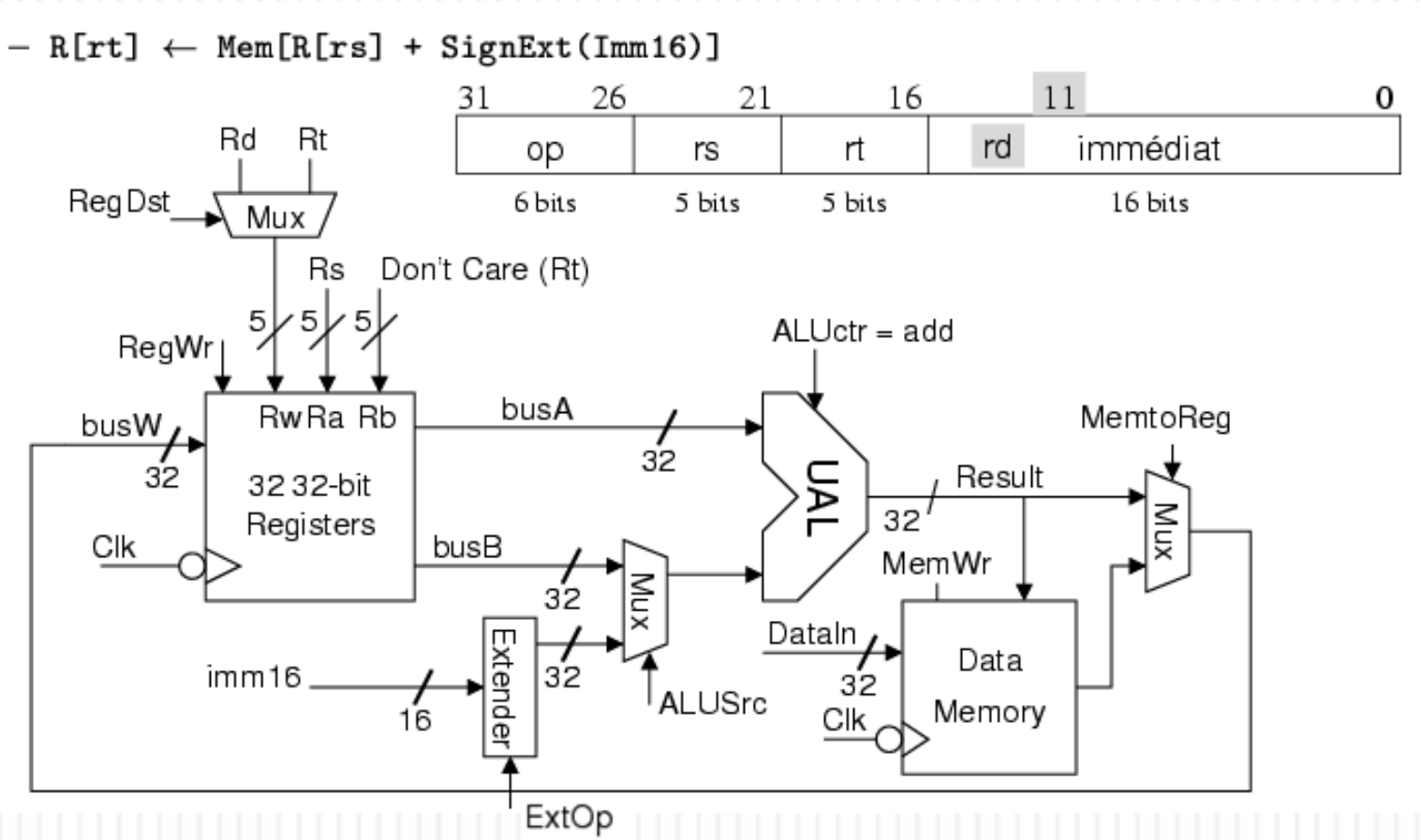
RTL : Instruction Load



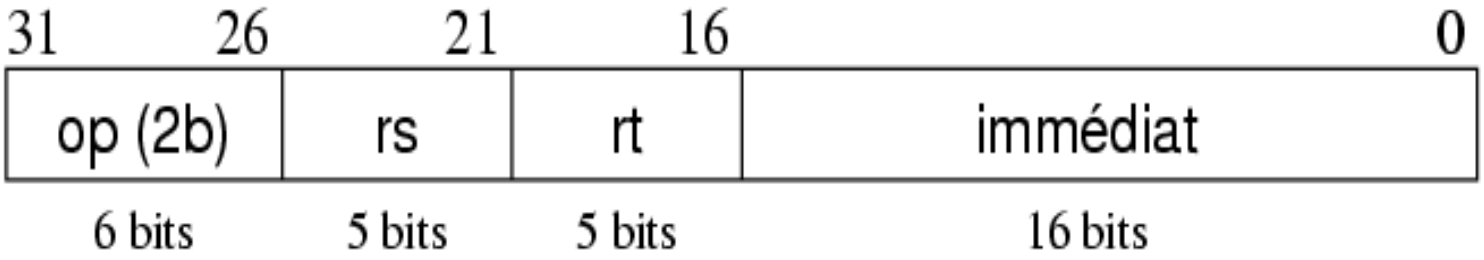
- **lw rt, rs, imm16**
  - `mem[PC]` — **Extraire l'instruction**
  - `Addr ← R[rs] + SignExt(imm16)` — **Calcul de l'adresse mémoire**
  - `R[rt] ← Mem[Addr]` — **Charger la donnée**
  - `PC ← PC + 4` — **adresse suivante**



Chemin de données pour Load



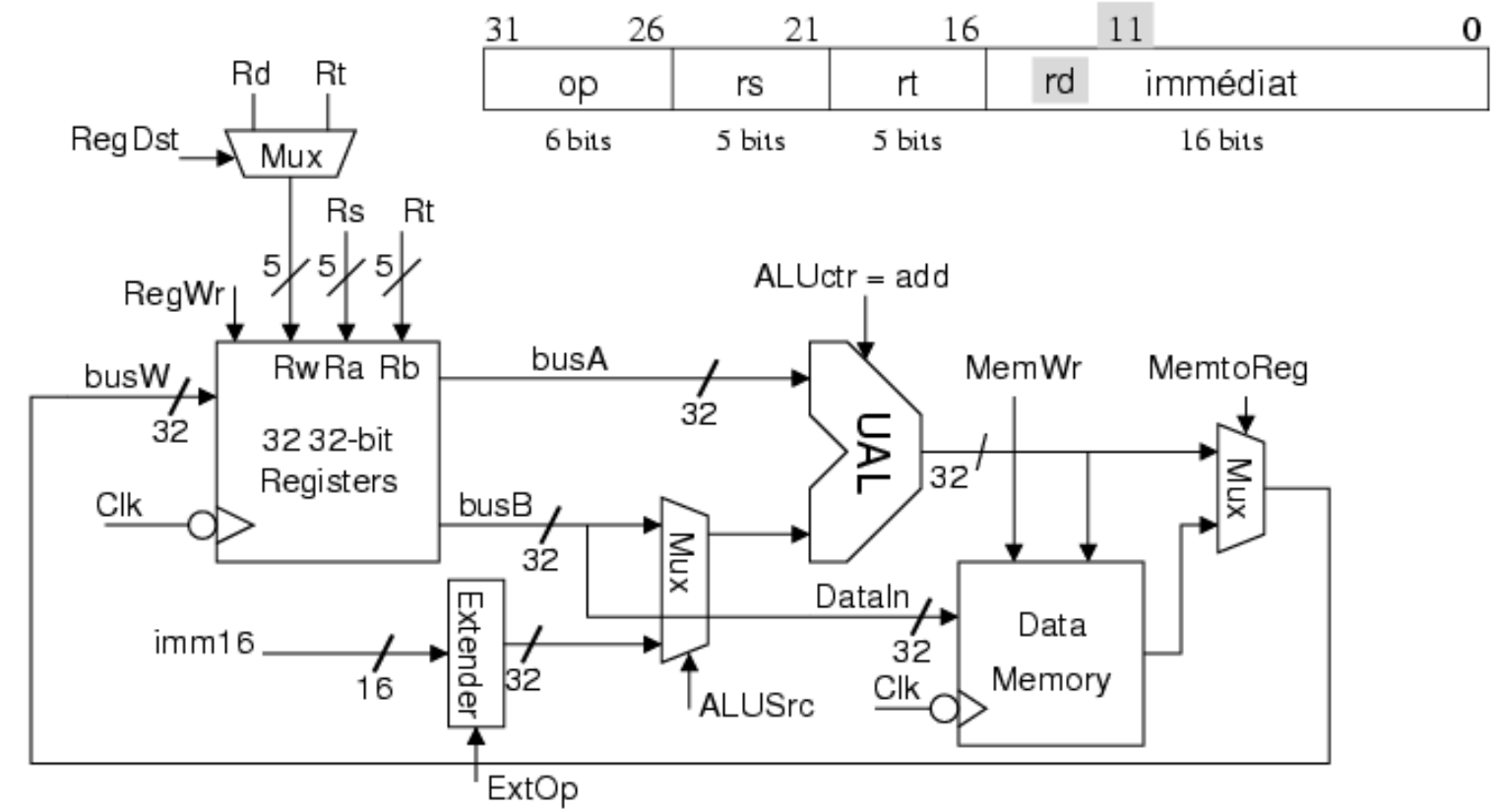
RTL : Instruction Store



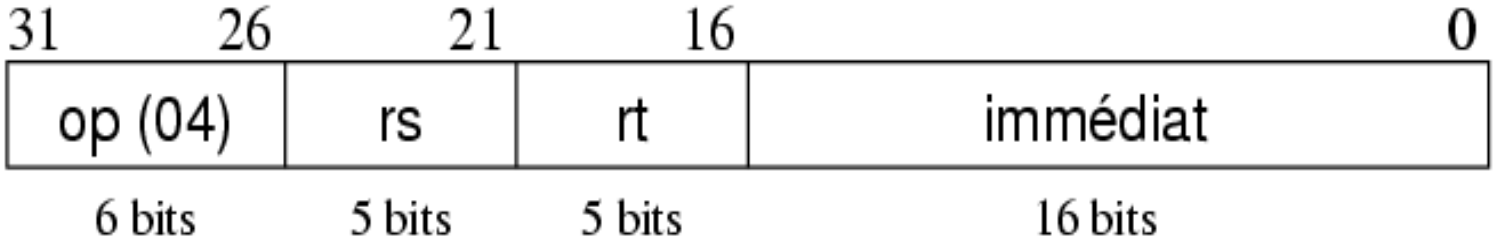
- **sw rt, rs, imm16**
  - `mem[PC]` Extraire l'instruction
  - `Addr ← R[rs] + SignExt(imm16)` Calcul de l'adresse mémoire
  - `Mem[Addr] ← R[rt]` Sauver la donnée
  - `PC ← PC + 4` adresse suivante

Chemin de données pour Store

- Mem[R[rs] + SignExt(Imm16)] ← R[rt]



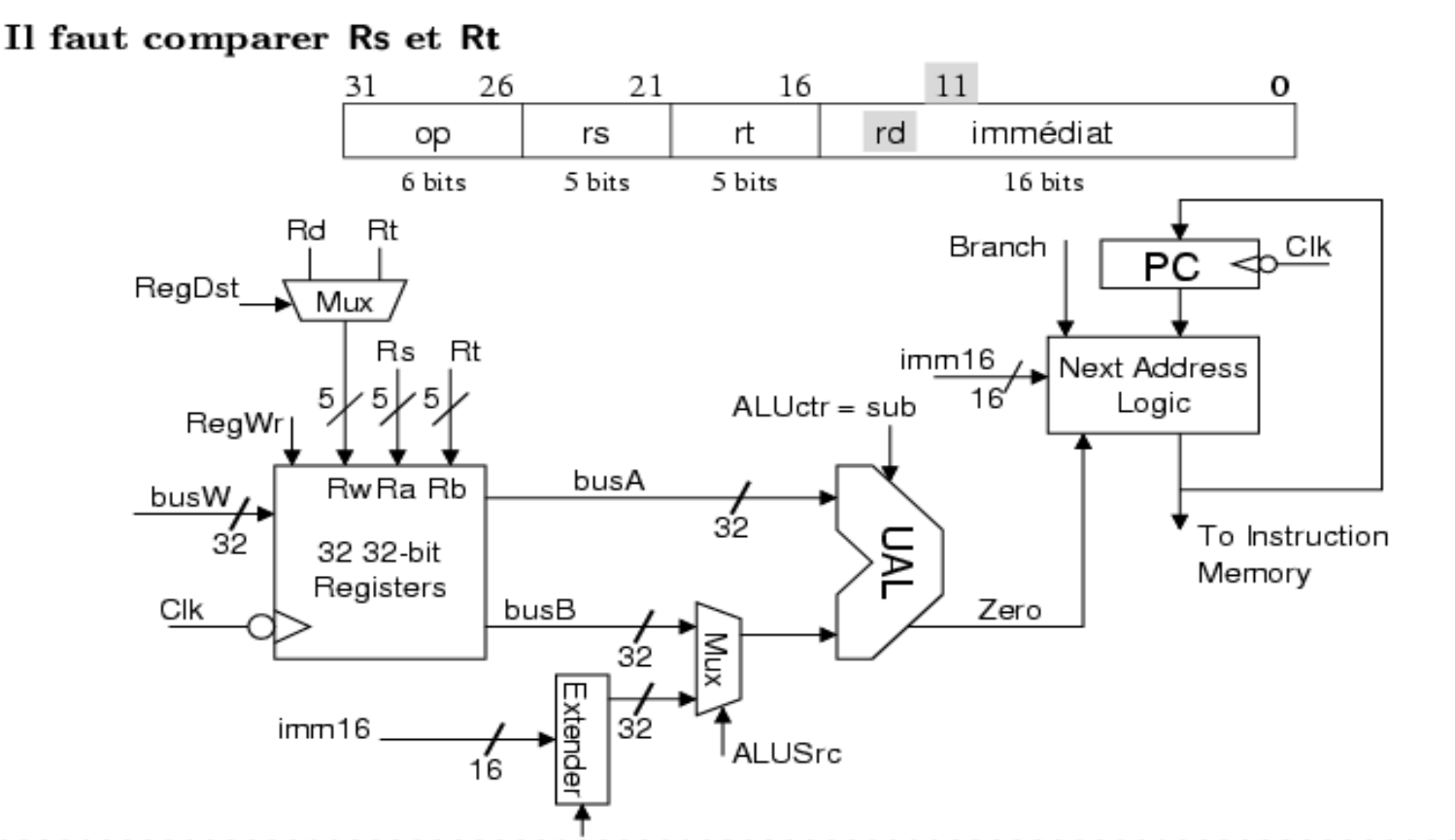
RTL : Instruction branch



- **beq rs, rt, imm16**
  - `mem[PC]` Extraire l'instruction
  - `Cond ← R[rs] - R[rt]` Calcul de la condition
  - `if (Cond == 0)` adresse suivante
    - `PC ← PC + 4 + (SignExt(imm16) × 4)`

      `else`
    - `PC ← PC + 4`

Chemin de données pour branch

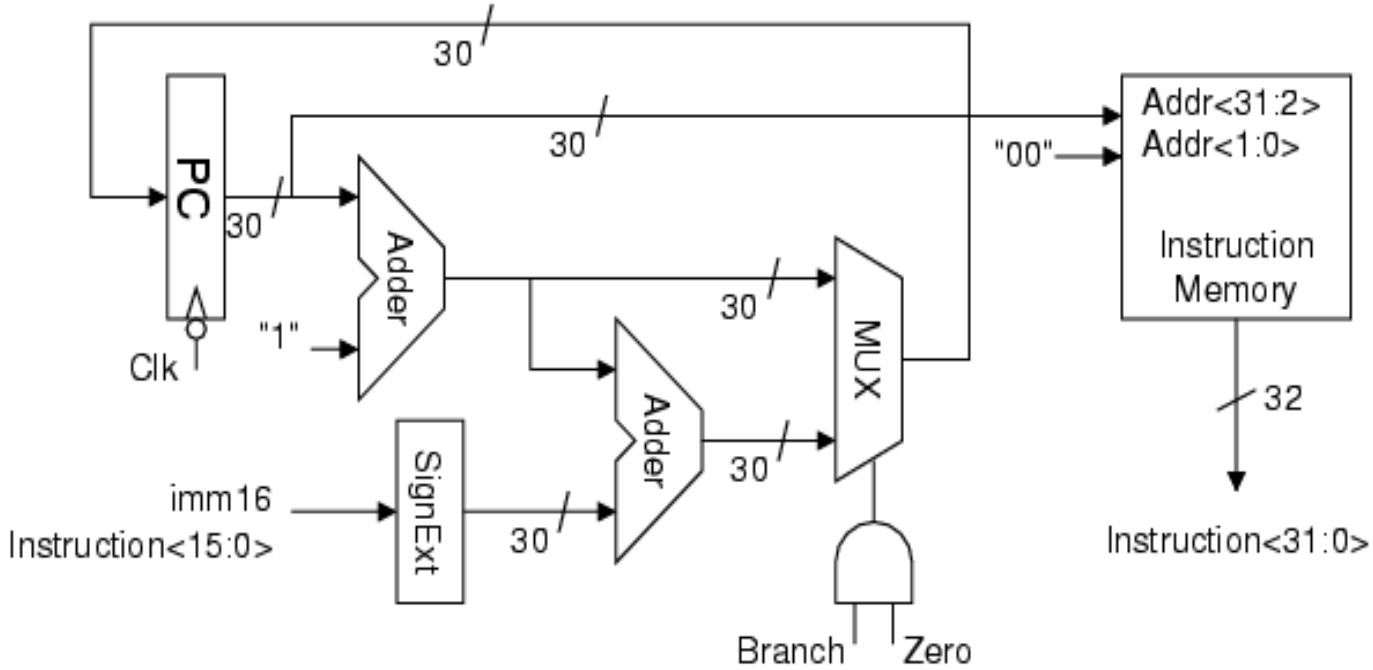


Arithmétique binaire pour l'adresse suivante

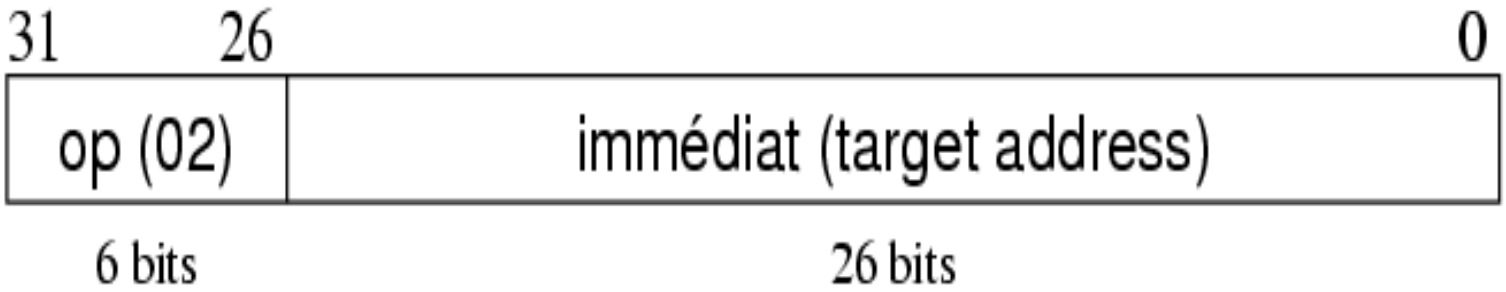
- En théorie, PC est une adresse d'octet sur 32 bits
  - opérations séquentielles : PC<31 :0> = PC<31 :0> + 4
  - branchements :PC<31 :0> = PC<31 :0> + 4 + signExt(Imm16) × 4
- Le nombre "magique" 4 est du au fait que :
  - le PC adresse des octets,
  - toutes les instructions ont 4 octets (32 bits) de long
- En d'autres mots
  - les 2 bits de poids faible du PC sont toujours à O,
  - il n'y a aucune raisons de les mémoriser
- En pratique, on simplifie le matériel en utilisant un PC sur 30 bits : PC<31 :2>
  - opérations séquentielles : PC<31 :2> = PC<31 :2> + 1
  - branchements :PC<31 :2> = PC<31 :2> + 1 + signExt(Imm16)
  - dans tous les cas, l'adresse mémoire est : PC<31 :2> concat ''00''

Logique adresse suivante (PC de 30 bits)

- **opérations séquentielles :** PC<31 :2> = PC<31 :2> + 1
- **branchements :**PC<31 :2> = PC<31 :2> + 1 + signExt(Imm16)
- **dans tous les cas, l'adresse mémoire est :** PC<31 :2> concat ''00''

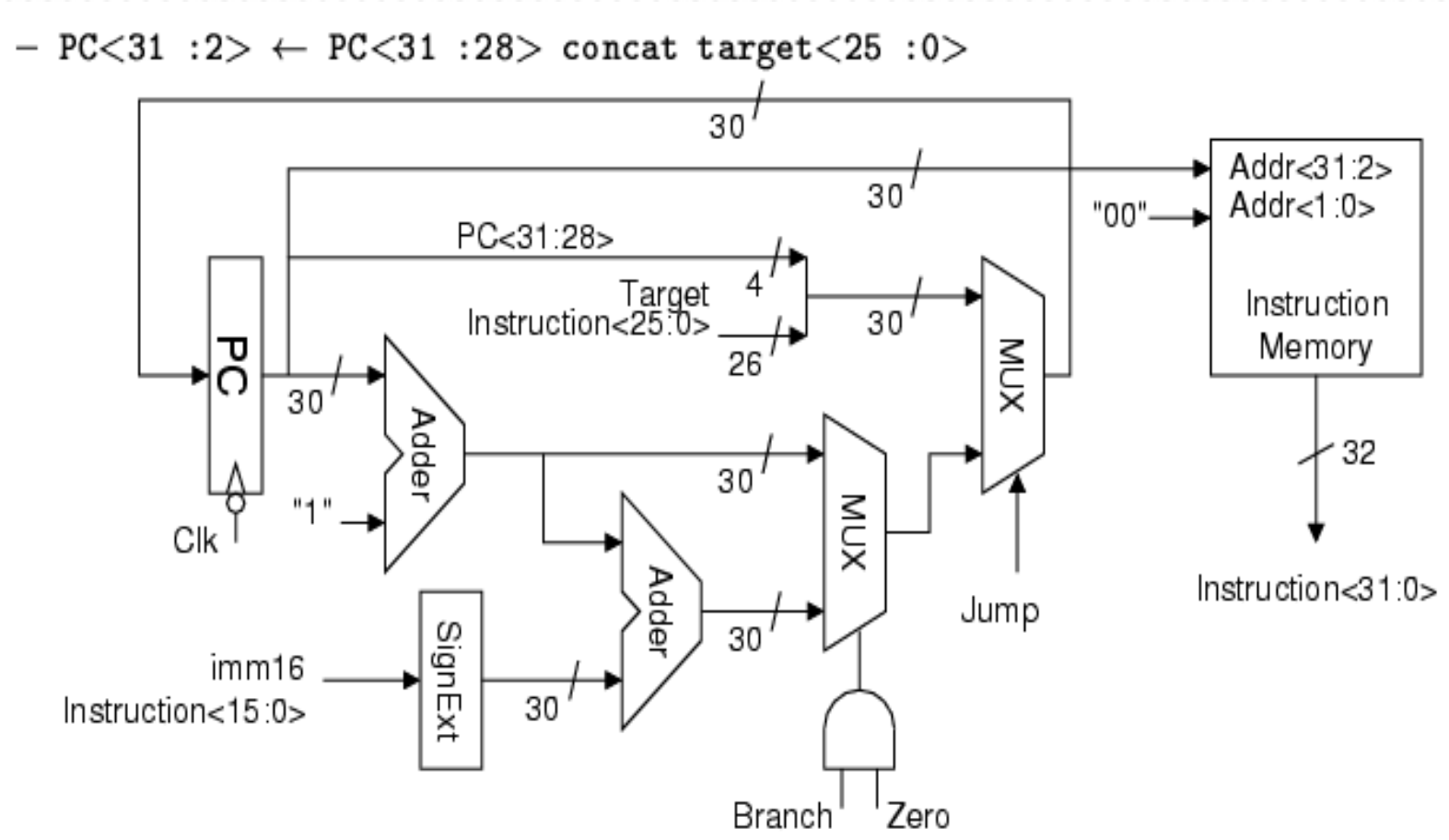


RTL : Instruction Jump

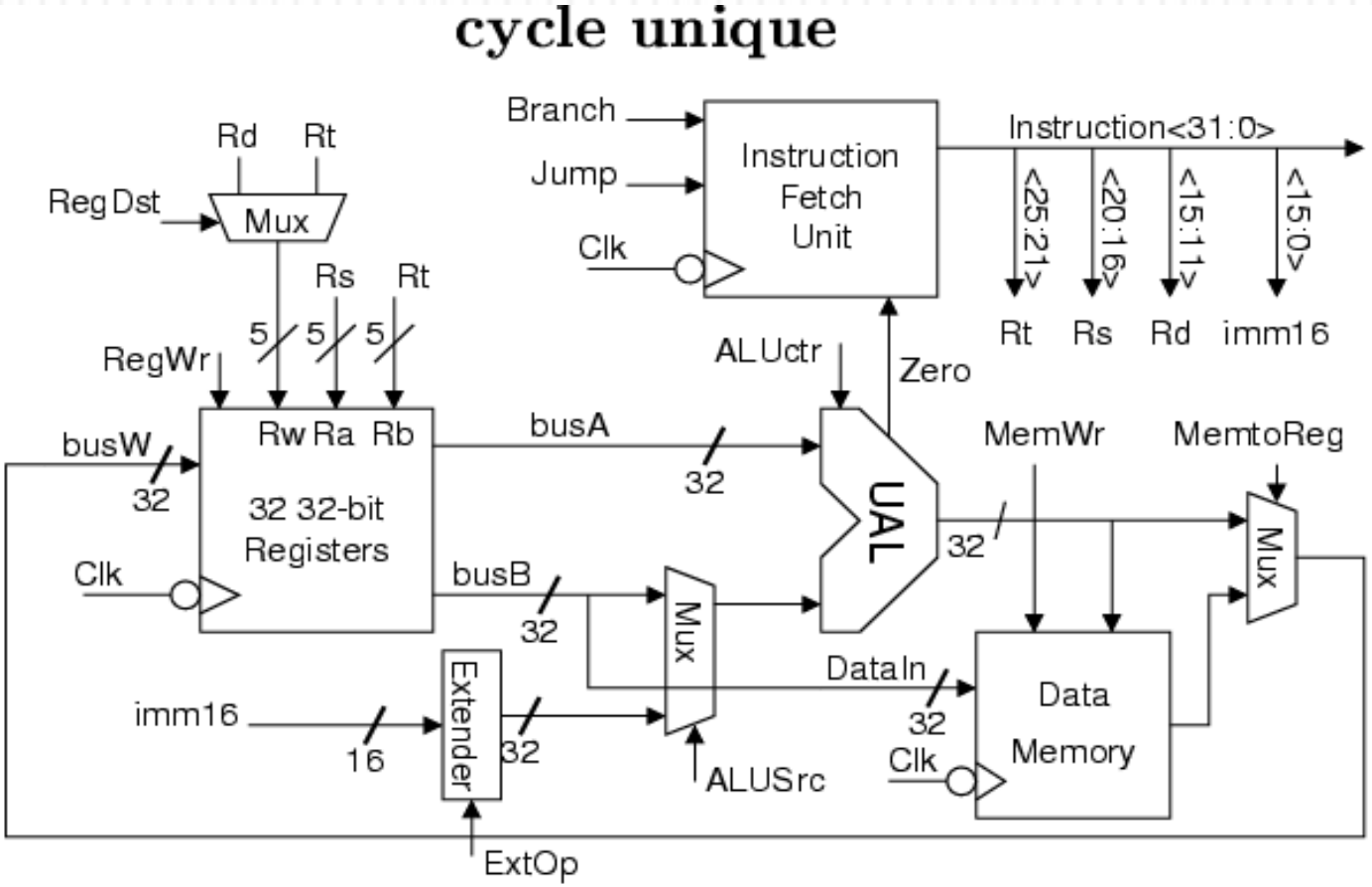


- **j target**
  - mem[PC] Extraire l'instruction
  - PC<31 :2> ← PC<31 :28> concat target<25 :0>
    calcul de la prochaine adresse d'instruction

Unité Fetch instruction



Le tout : un chemin de données à cycle unique



Questions

Q1. Comment obtient-on le cycle horloge dans une implémentation à cycle unique ?

Q2. Quels sont les inconvénients de l'implémentation monocycle ?

Q3. Montrer sur le schéma le chemin de données pendant l'exécution de chacune des instructions


Thank's