

# Représentation des instructions

# Rappel

---

- ISA (Instruction Set Architecture): hardware / software interface
  - Principes de conception/ compromis
- Instructions MIPS
  - Arithmétique: `add/sub $t0, $s0, $s1`
  - Transfert de données (accès mémoire): `lw/sw $t1, 8($s1)`
- Les opérandes doivent être des registres
  - 32 registres de taille 32-bit
  - `$t0 - $t7 => $8 - $15`
  - `$s0 - $s7 => $16 - $23`
- Mémoire: large, tableau d'octets d'une seule dimension  $M[2^{32}]$ 
  - L'adresse mémoire est un indice dans un tableau d'octets
  - Mots Alignés : `M[0], M[4], M[8], ..., M[4,294,967,292]`
  - Ordre des octets Gros/Petit boutiste

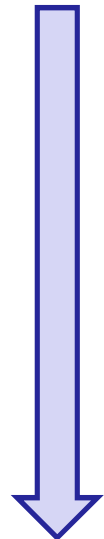
# Langage Machine -- MIPS

- Toutes les instructions ayant la même longueur (32 bits)
- **Une bonne conception signifie un bon compromis**
  - Même longueur d'instruction ou même format
- Trois types formats
  - R: Format des instructions arithmétiques
  - I: Format transfert, branchement, immédiat
  - J: Format des instructions de saut
- `add $t0, $s1, $s2`
  - 32 bits en langage machine
  - Champs pour :
  - Opération (add)
    - Opérandes (\$s1, \$s2, \$t0)

`A[300] = h + A[300];`

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

```
101011010010100000000010010110000
0000000100100100001000000000100000
100011010010100000000010010110000
```



# Formats des instructions

---

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
<b>R:</b>	op	rs	rt	rd	shamt	funct
<b>I:</b>	op	rs	rt	address / immediate		
<b>J:</b>	op	target address				

op: code de l'opération basique de l'instruction (opcode)

rs: Premier registre source de l'opérande

rt: Deuxième registre source de l'opérande

rd: Registre de destination

shamt: shift amount (nombre de décalage)

funct: sélectionne la variante spécifique de l'opcode (function code)

address: offset pour les instructions load/store ( $\pm 2^{15}$ )

immediate: des constants pour les instructions immediates

# Format R

---

add \$t0, \$s1, \$s2 (add \$8, \$17, \$18 # \$8 = \$17 + \$18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

sub \$t1, \$s1, \$s2 (sub \$9, \$17, \$18 # \$9 = \$17 - \$18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	9	0	34
000000	10001	10010	01001	00000	100010

# Format I

---

lw \$t0, 52(\$s3)

lw \$8, 52(\$19)

6 bits	5 bits	5 bits	16 bits
35	19	8	52
100011	10011	01000	0000 0000 0011 0100

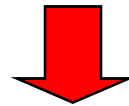
sw \$t0, 52(\$s3)

sw \$8, 52(\$19)

6 bits	5 bits	5 bits	16 bits
43	19	8	52
101011	10011	01000	0000 0000 0011 0100

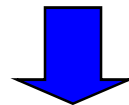
# Exemple

`A[300] = h + A[300];`     `/* $t1 <= base of array A; $s2 <= h */`



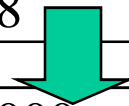
Compilateur

```
lw  $t0, 1200($t1)    # registre temporaire $t0 gets A[300]
add $t0, $s2, $t0      # registre temporaire $t0 gets h + A[300]
sw  $t0, 1200($t1)    # stores h + A[300] retour dans A[300]
```



Assembleur

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		



100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

# Instructions immédiates (Constantes Numériques)

---

- Les petites constantes sont utilisées fréquemment (50% des operands)

- $A = A + 5;$
- $C = C - 1;$

- Instructions MIPS pour les constantes (format I )

- `addi $t0, $s7, 4`      # \$t0 = \$s7 + 4


8	23	8	4
001000	10111	01000	0000 0000 0000 0100



# Dépassement arithmétique

---

- Les ordinateurs se caractérisent par une précision limitée (32 bits)

$$\begin{array}{r} 15 \\ + 3 \\ \hline 18 \end{array} \qquad \begin{array}{r} 1111 \\ 0011 \\ \hline 10010 \end{array}$$


- Quelques languages détectent le dépassement (overflow) comme : Ada, d'autres non comme , C
- MIPS fournit 2 types d'instructions arithmétiques :
  - Add, sub, and addi: causent le dépassement
  - Addu, subu, and addiu: ne le font pas

# Les instructions des opérations logiques

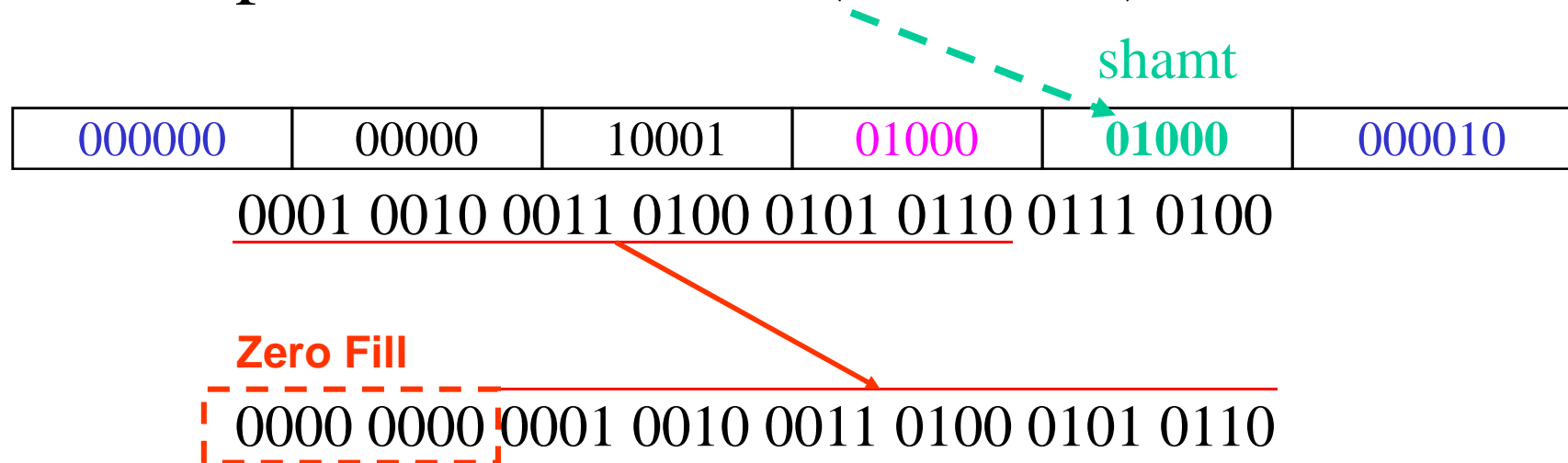
---

- Operations sur les bits
  - Les nombres sont stockés dans les registres sous forme binaires (32 bits)
- Instructions
  - and, or: les 3 operandes sont des registres (format R)
  - andi, ori: le 3ième argument est une constante immediate (format I)
- Exemple: masque (andi \$t0, \$t0, 0xFFF)

1011	0110	1010	0100	0011	1101	1001	1010
0000	0000	0000	0000	0000	1111	1111	1111
<hr/>							
0000	0000	0000	0000	0000	1101	1001	1010

# Les instruction de décalage (Shift)

- Décaler tous les bits dans le registre left/right
  - sll (shift left logical): rajouter des 0s à droite
  - srl (shift right logical): rajouter des 0s à gauche
  - sra (shift right arithmetic): sign extends emptied bits
- Exemple: `srl $t0, $s1, 8` (format R)



# Multiplication & Division

---

- Usage des registres spéciaux (hi, lo)

- Le résultat du produit de 2 nombres de 32-bits = Nombre codé sur 64-bits

000000	10000	10001	00000	00000	011000
--------	-------	-------	-------	-------	--------

- Mult \$s0, \$s1

- hi: la moitié supérieure du produit
- lo: la moitié inférieure du produit

- Div \$s0, \$s1

000000	10000	10001	00000	00000	011010
--------	-------	-------	-------	-------	--------

- hi: Le reste de la division ( $\$s0 / \$s1$ )
- lo: le quotient de la division ( $\$s0 \% \$s1$ )

- Déplacer les résultats vers des registres généraux:

- mfhi \$s0

000000	00000	00000	10000	00000	010000
--------	-------	-------	-------	-------	--------

- mflo \$s1

000000	00000	00000	10001	00000	010010
--------	-------	-------	-------	-------	--------

# Les instructions de décision

---

- Le Branchement conditionnel
  - If-then
  - If-then-else
- Les boucles
  - While
  - Do while
  - For
- Les inégalités
- L'instruction Switch

# Le branchement Conditionnel

---

- Les instructions de prise de décision
- Branchement si Égalité
  - **beq** registre1, registre2, *adresse\_dédestination*
- Branchement si non Egalité
  - **bne** registre1, registre2, *adresse\_dédestination*
- Exemple: beq \$s3, \$s4, 20

6 bits	5 bits	5 bits	16 bits
4	19	20	5
000100	10011	10100	0000 0000 0000 0101

# Les étiquettes (Labels)

---

- Pas besoin de calcul des adresses pour le branchement

Si $(i == j)$ aller à L1;  $f = g + h$ ;  L1: $f = f - i$ ;
---

$f \Rightarrow \$s0$ $g \Rightarrow \$s1$ $h \Rightarrow \$s2$ $i \Rightarrow \$s3$ $j \Rightarrow \$s4$
--

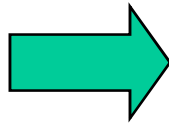
(4000) beq \$s3, \$s4, L1    # Si i égal j aller à L1  (4004) add \$s0, \$s1, \$s2    # $f = g + h$  L1: (4008) sub \$s0, \$s0, \$s3    # $f = f - i$
---

L1 correspond à l'adresse de l'instruction de soustraction

# Les expressions If

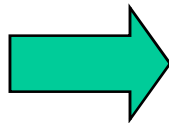
---

```
if (condition)
  Faire1;
else
  Faire2;
```



```
if (condition) aller à L1;
  Faire2;
  aller à L2;
L1: Faire 1;
L2:
```

```
if (i == j)
  f = g + h;
else
  f = g - h;
```



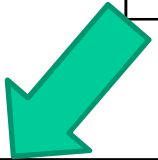
```
beq $3, $4, Vrai
sub $0, $s1, $s2
j Faux
Vrai: add $s0, $s1, $s2
Faux:
```



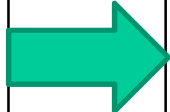
# Les Boucles

```
boucle: g = g + A[i];  
        i = i + j;  
        if (i != h) aller à boucle;
```

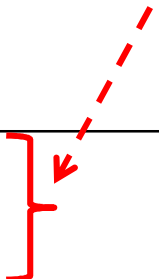
Méthode intelligente  
de multiplication par 4  
pour obtenir un  
décalage d'octet pour  
un mot



```
g: $s1  
h: $s2  
i: $s3  
j: $s4  
Base de A: $s5
```



```
boucle: add $t1, $s3 $s3 # $t1 = 2 * i  
        add $t1, $t1, $t1 # $t1 = 4 * i  
        add $t1, $t1, $s5 # $t1=adresse de A[i]  
        lw  $t0, 0($t1)  # $t0 = A[i]  
        add $s1, $s1, $t0 # g = g + A[i]  
        add $s3, $s3, $s4 # i = i + j  
        bne $s3, $s2, boucle # aller à boucle si i != h
```



# La boucle While

---

```
while (tab[i] == k)
    i = i + j;
```

```
# i: $s3; j: $s4; k: $s5; base de tab: $s6

boucle: add $t1, $s3, $s3    # $t1 = 2 * i
        add $t1, $t1, $t1    # $t1 = 4 * i
        add $t1, $t1, $s6    # $t1 = adresse de tab[i]
        lw  $t0, 0($t1)     # $t0 = tab[i]
        bne $t0, $s5, quitter # aller à quitter Si tab[i] != k
        add $s3, $s3, $s4    # i = i + j
        j   boucle          # aller à boucle

quitter:
```

Le nombre d'instructions exécuté Si  $\text{tab}[i + m * j]$  est différent de k pour  $m = 10$  et égal à k pour  $0 \leq m \leq 9$  est  $10 \times 7 + 5 = 75$

# Optimisation

Tour partielle  
de la boucle

6  
Instr's

	add \$t1, \$s3, \$s3	# Reg Temp \$t1 = 2 * i
	add \$t1, \$t1, \$t1	# Reg Temp \$t1 = 4 * i
	add \$t1, \$t1, \$s6	# \$t1 = adresse de tab[i]
	lw \$t0, 0(\$t1)	# Reg Temp \$t0 = tab[i]
	bne \$t0, \$s5, Exit	# aller à Exit if tab[i] ≠ k
Loop:	add \$s3, \$s3, \$s4	# i = i + j
	add \$t1, \$s3, \$s3	# Reg Temp \$t1 = 2 * i
	add \$t1, \$t1, \$t1	# Reg Temp \$t1 = 4 * i
	add \$t1, \$t1, \$s6	# \$t1 = adresse de tab[i]
	lw \$t0, 0(\$t1)	# Reg Temp \$t0 = tab[i]
	beq \$t0, \$s5, Loop	# aller à Loop Si tab[i] = k
Exit:		

Le nombre d'instructions exécutées par cette nouvelle forme de la boucle est :  
 $5 + 10 \times 6 = 65 \rightarrow \text{Rendement} = 1.15 = 75/65$ . Si  $4 \times i$  est calculé avant la boucle, d'autres rendements supplémentaires dans le corps de la boucle sont possibles.

# La boucle Do-While

```
do {  
    g = g + A[i];  
    i = i + j;  
} while (i != h);
```

*Réécrire*

```
L1:  g = g + A[i];  
      i = i + j;  
      if (i != h) goto L1
```

```
g: $s1  
h: $s2  
i: $s3  
j: $s4  
Base de A: $s5
```

```
L1: sll  $t1, $s3, 2    # $t1 = 4*i  
     add $t1, $t1, $s5  # $t1 = addr of A  
     lw  $t1, 0($t1)    # $t1 = A[i]  
     add $s1, $s1, $t1  # g = g + A[i]  
     add $s3, $s3, $s4  # i = i + j  
     bne $s3, $s2, L1   # go to L1 if i != h
```

- Le branchement conditionnel est la clé des prises de décision

# Inequalities

---

- Les programmes nécessitant des tests < and >
- *Set on less than* instruction
- **slt** registre1, registre2, registre3
  - registre1 = (register2 < register3)? 1 : 0;
- Example: if (g < h) goto Less;

g: \$s0

h: \$s1

slt \$t0, \$s0, \$s1

bne \$t0, \$0, Less

- **slti**: Utile dans les boucles For if (g >= 1) goto Loop

slti \$t0, \$s0, **1**      # \$t0 = 1 Si g < 1

beq \$t0, \$0, Loop      # goto Loop Si g >= 1

- Versions non signées: **sltu** and **sltiu**

# Conditions Relatives

- $==$   $!=$   $<$   $<=$   $>$   $>=$ 
  - MIPS ne support directement ces instructions
  - Les compilateurs utilisent **slt**, **beq**, **bne**, **\$zero**, and **\$at**
- Pseudoinstructions
  - `blt $t1, $t2, L` # if ( $\$t1 < \$t2$ ) go to L
    - $\left\{ \begin{array}{l} \text{slt } \$at, \$t1, \$t2 \\ \text{bne } \$at, \$zero, L \end{array} \right.$
  - `ble $t1, $t2, L` # if ( $\$t1 \leq \$t2$ ) go to L
    - $\left\{ \begin{array}{l} \text{slt } \$at, \$t2, \$t1 \\ \text{beq } \$at, \$zero, L \end{array} \right.$
  - `bgt $t1, $t2, L` # if ( $\$t1 > \$t2$ ) go to L
    - $\left\{ \begin{array}{l} \text{slt } \$at, \$t2, \$t1 \\ \text{bne } \$at, \$zero, L \end{array} \right.$
  - `bge $t1, $t2, L` # if ( $\$t1 \geq \$t2$ ) go to L
    - $\left\{ \begin{array}{l} \text{slt } \$at, \$t1, \$t2 \\ \text{beq } \$at, \$zero, L \end{array} \right.$

# L'expression Switch en C

```
switch (k) {  
    case 0: f = i + j; break;  
    case 1: f = g + h; break;  
    case 2: f = g - h; break;  
    case 3: f = i - j; break;  
}
```

```
if (k==0) f = i + j;  
else if (k==1) f = g + h;  
else if (k==2) f = g - h;  
else if (k==3) f = i - j;
```

```
# f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
```

```
    bne    $s5, $0, L1      # branch k != 0  
    add    $s0, $s3, $s4    # f = i + j  
    j      Exit            # end of case  
L1: addi   $t0, $s5, -1     # $t0 = k - 1  
    bne    $t0, $0, L2     # branch k != 1  
    add    $s0, $s1, $s2    # f = g + h  
    j      Exit            # end of case  
L2: addi   $t0, $s5, -2     # $t0 = k - 2  
    bne    $t0, $0, L3     # branch k != 2  
    sub    $s0, $s1, $s2    # f = g - h  
    j      Exit            # end of case  
L3: addi   $t0, $s5, -3     # $t0 = k - 3  
    bne    $t0, $0, Exit   # branch k != 3  
    sub    $s0, $s3, $s4    # f = i - j  
Exit:
```

# Tables des sauts

- Instruction de saut vers le contenu d'un registre
  - **jr** <registre>
  - branchement inconditionnelle vers une adresse contenue dans un registre

Table des étiquettes

L0	
L1	
L2	
L3	

```
# f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
# $t2 = 4; $t4 = base address of JT

slt    $t3, $s5, $zero    # test k < 0
bne    $t3, $zero, Exit   # if so, exit
slt    $t3, $s5, $t2      # test k < 4
beq    $t3, $zero, Exit   # if so, exit
add    $t1, $s5, $5        # $t1 = 2*k
add    $t1, $t1, $t1       # $t1 = 4*k
add    $t1, $t1, $t4       # $t1 = &JT[k]
lw     $t0, 0($t1)         # $t0 = JT[k]
jr     $t0                # jump registre

L0: add $s0, $s3, $s4      # k == 0
j      Exit               # break
L1: add $s0, $1, $s2       # k == 1
j      Exit               # break
L2: sub $s0, $s1, $s2      # k == 2
j      Exit               # break
L3: sub $s0, $s3, $s4      # k == 3
Exit:
```



# Conclusions

---

- Format d'instructions MIPS – 32 bits
- Code d'assemblage: Dest = premier opérande
- Langage Machine: Dest = dernier opérande
- **Trois Formats MIPS:**
  - R (arithmetic)**
  - I (immediate)**
  - J (jump)**
- Instructions de Décision – utilise *jump* (goto)
- Amélioration des performances – déroulement des boucles