Tema 3
Traducció de programes

Estructura de Computadors (EC)

Desplaçament de bits

Desplaçaments lògics (esquerra i dreta)

- Shift Left Logical: `sll rd, rt, shamt`
  - Desplaça `rt` a l'esquerra el número de bits indicat per l'operand immediat `shamt`
- Shift Right Logical: `srl rd, rt, shamt`
  - Desplaça `rt` a la dreta el numero de bits indicat per l'operand immediat `shamt`
- Les posicions que queden vacants s'omplen amb zeros

Desplaçaments lògics (esquerra i dreta)

- Shift Left Logical: `sll rd, rt, shamt`
  - Desplaça `rt` a l'esquerra el número de bits indicat per l'operand immediat `shamt`
- Shift Right Logical: `srl rd, rt, shamt`
  - Desplaça `rt` a la dreta el numero de bits indicat per l'operand immediat `shamt`
- Les posicions que queden vacants s'omplen amb zeros

```
li $t0, 0x33333333
sll $t1, $t0, 2        # $t1 = 0xCCCCCCCC
srl $t2, $t0, 2        # $t2 = 0x0CCCCCCC
```

Desplaçament aritmètic a la dreta

- Shift Right Arithmetic: `sra rd, rt, shamt`
  - Desplaça `rt` a la dreta el número de bits indicat per l'operand immediat `shamt`
- Les posicions vacants a la dreta s'omplen amb una copia del bit de signe de `rt`

```
li $t0, 0x88888888
sra $t1, $t0, 1        # $t1 = 0xC4444444
```

Desplaçament indicat a registre

- `sllv rd, rt, rs`
- `srlv rd, rt, rs`
- `srav rd, rt, rs`

- El número de bits a desplaçar s'indica en els 5 bits de menor pes del registre `rs`, la resta de bits s'ignoren

Aplicacions: Multiplicació i divisió per potències de 2

- Desplaçar a l'esquerra equival a multiplicar per una potència de 2
  - `sll rd, rt, shamt`
  - `rd = rt × 2^shamt`
  - Útil per calcular el offset d'un vector donat un index

- Desplaçar a la dreta un natural (`srl`) o un enter (`sra`) equival a dividir per una potència de 2
  - `srl rd, rt, shamt`
  - `rd = rt / 2^shamt`

Exemple d'ús

```
int vec[100];

void main() {
    int i = 12;
    vec[i] = 0;
}
```

Exemple d'ús

```
int vec[100];

void main() {
    int i = 12;
    vec[i] = 0;
}
```

```
        .data
        .align 2
vec:    .space 400

        .text
main:   li $t0, 12          # $t0 = i
        sll $t1, $t0, 2 # $t1 = i * 4
        la $t2, vec
        addu $t2, $t2, $t1  # $t2 = @vec[i]
        sw $zero, 0($t2)
```

Aplicacions: Conversió de Ca2 a Ca1

- Representar en Ca1 un enter x guardat en Ca2 a $t0
  - Es pot convertir de Ca2 a Ca1 restant el bit de signe
    - Si x és negatiu: restar 1
    - Si x és positiu: queda igual

Aplicacions: Conversió de Ca2 a Ca1

- Representar en Ca1 un enter `x` guardat en Ca2 a `$t0`
  - Es pot convertir de Ca2 a Ca1 restant el bit de signe
    - Si `x` és negatiu: restar 1
    - Si `x` és positiu: queda igual

```
# Desplacem el bit de signe a la posició 0
srl $t1, $t0, 31

# Restem el bit de signe
subu $t0, $t0, $t1
```

Traducció dels operador de desplaçament en C

- C defineix dos operadors de desplaçament de bits
  - Desplaçament a l'esquerra: <<
    - Es tradueix a la instrucció `sll`
  - Desplaçament a la dreta: >>
    - Per naturals es tradueix a la instrucció `srl`
    - Per enters es tradueix a la instrucció `sra`

Traducció dels operador de desplaçament en C

- C defineix dos operadors de desplaçament de bits
  - Desplaçament a l'esquerra: `<<`
    - Es tradueix a la instrucció `sll`
  - Desplaçament a la dreta: `>>`
    - Per naturals es tradueix a la instrucció `srl`
    - Per enters es tradueix a la instrucció `sra`
- Traduiu a MIPS la següent sentència en C. On `a` i `b` son enters amb signe i estan emmagatzemats a `$t0` i `$t1`.

```
a = (a << b) >> 2;
```

Traducció dels operador de desplaçament en C

- C defineix dos operadors de desplaçament de bits
  - Desplaçament a l'esquerra: `<<`
    - Es tradueix a la instrucció `sll`
  - Desplaçament a la dreta: `>>`
    - Per naturals es tradueix a la instrucció `srl`
    - Per enters es tradueix a la instrucció `sra`
- Traduiu a MIPS la següent sentència en C. On `a` i `b` son enters amb signe i estan emmagatzemats a `$t0` i `$t1`.

```
a = (a << b) >> 2;
```

```
sllv $t4, $t0, $t1
sra $t0, $t4, 2
```

Operacions lògiques bit a bit

Operacions lògiques bit a bit a MIPS

- Repertori d'instruccions lògiques

and/or/xor/nor/andi/ori/xori		
`and rd, rs, rt`	rd = rs AND rt	
`or rd, rs, rt`	rd = rs OR rt	
`xor rd, rs, rt`	rd = rs XOR rt	
`nor rd, rs, rt`	rd = rs NOR rt = NOT (rs OR rt)	
`andi rt, rs, imm16`	rt = rs AND ZeroExt(imm16)	imm16 ha de ser un natural
`ori rt, rs, imm16`	rt = rs OR ZeroExt(imm16)	imm16 ha de ser un natural
`xori rt, rs, imm16`	rt = rs XOR ZeroExt(imm16)	imm16 ha de ser un natural

- imm16 ha de ser un natural

Operacions lògiques bit a bit - C vs MIPS

- Assumint que `a`, `b` i `c` estan a `$t0`, `$t1` i `$t2`:

Lenguaje C	Ensamblador MIPS
`c = a & b;`	`and $t2, $t0, $t1`
`c = a | b;`	`or $t2, $t0, $t1`
`c = a ^ b`	`xor $t2, $t0, $t1`
`c = ~a;`	`nor $t2, $t0, $zero`
`c = a & 7;`	`andi $t2, $t0, 7`
`c = a | 7;`	`ori $t2, $t0, 7`
`c = a ^ 7;`	`xori $t2, $t0, 7`

Exemple

- Traduiu la següent sentencia de C a MIPS
  - `a = ~(a & b);`
    - Operador “~”: Negació bit a bit
    - Assumeix que `a` i `b` s’emmagatzemen als registres `$t0` i `$t1`

Exemple

- Traduiu la següent sentencia de C a MIPS
  - `a = ~(a & b);`
    - Operador “~”: Negació bit a bit
    - Assumeix que `a` i `b` s’emmagatzemen als registres `$t0` i `$t1`

```
and $t4, $t0, $t1
nor $t0, $t4, $zero
```

Aplicacions: Seleccionar bits

- L'instrucció `and` s'utilitza per seleccionar determinats bits d'un registre, posant la resta a `0`
- Exemple: seleccionar els 16 bits de menor pes del registre `$t0`
  - `andi $t1, $t0, 0xFFFF`
- Exemple: seleccionar els bits `0, 2, 4, 6`
  - `andi $t1, $t0, 0x0055`
- La instrucció `andi` es pot utilitzar per calcular el residu de la divisió per potències de `2`

Altres aplicacions de les operacions lògiques bit a bit

- Posar bits a 1
  - Exemple: posar a 1 els 16 bits de menor per de `$t0`

```
ori $t0, $t0, 0xFFFF
```

- Complementar bits
  - Exemple: complementar els bits parells de `$t0`

```
li $t1, 0x55555555
xor $t0, $t0, $t1
```

Comparacions i operacions booleanes

Comparacions i operacions booleanes

- En C les expressions enteres admeten les operacions de comparació:
  - `==`, `!=`, `<`, `<=`, `>`, `>=`
- No existeix el tipus bolea, el resultat d'una comparació és un enter
  - 0: fals
  - Diferent de 0: cert
- També existeixen expressions lògiques formades per operadors booleans
  - `&&`, `||`, `!`
- Els operadors booleans retornen un valor enter "normalitzat": `0` o `1`

Repertori d'instruccions de comparació

- Generen un valor lògic normalitzat
  - Si la comparació és falsa escriuen un `0` a `rd`
  - Si la comparació és certa escriuen un `1` a `rd`

slt/sltu/slti/sltiu		
`slt rd, rs, rt`	rd = rs < rt	comparació d'enters
`sltu rd, rs, rt`	rd = rs < rt	comparació de naturals
`slti rt, rs, imm16`	rt = rs < Sext(`imm16`)	comparació d'enters
`sltiu rt, rs, imm16`	rt = rs < Sext(`imm16`)	comparació de naturals

Comparacions de tipus “<”

- Tradueix a MIPS suposant que `a`, `b` i `c` s’emmagatzemen a `$t0`, `$t1` i `$t2`:

  ```
  c = a < b;
  ```

- Si `a` i `b` son naturals

  ```
  sltu $t2, $t0, $t1
  ```

- Si `a` i `b` son enters

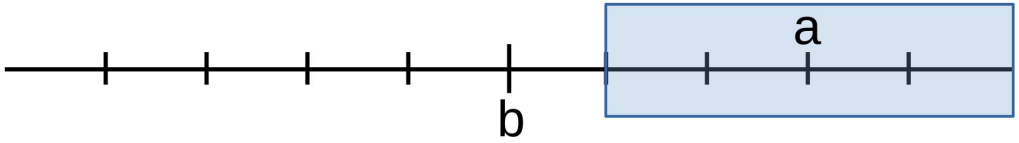
  ```
  slt $t2, $t0, $t1
  ```

Comparacions “>”, “>=”, “<=”

- MIPS només inclou instruccions de comparació “menor que”
- Les altres desigualtats es poden traduir utilitzant “menor que”
  - $a > b \equiv b < a$
  - $a \geq b \equiv \overline{(a < b)}$
  - $a \leq b \equiv \overline{(a > b)} \equiv \overline{(b < a)}$

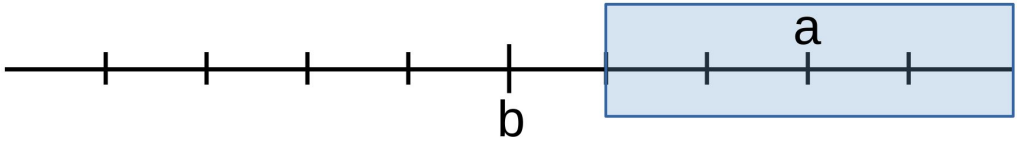
Exemple Comparació “>”

- a > b ≡ b < a



Exemple Comparació “>”

- `a > b` ≡ `b < a`



- Tradueix `c = (a > b)` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`

```
slt $t2, $t1, $t0      # c = (b < a)
```

Negació lògica “!”

```
unsigned char a = 0x55;
unsigned char b;

b = !a;    // b = 0
```

Negació lògica “!”

```
unsigned char a = 0x55;
unsigned char b;

b = !a;     // b = 0
```

- Traducció a MIPS amb comparació “menor que 1” com a natural
  - `sltiu $t1, $t0, 1` `# $t0 = a, $t1 = b`
    - Si `$t0` és zero (fals): `$t1 = 1` (cert)
    - Si `$t0` és diferent de zero (cert): `$t1 = 0` (fals)

Negació bit a bit “~” vs Negació lògica “!”

- La negació bit a bit i la negació lògica poden donar resultat diferent

```
unsigned char a = 0x55;
unsigned char b, c;

b = !a;  // b = 0
c = ~a;  // c = 0xAA
```

Negació bit a bit “~” vs Negació lògica “!”

- La negació bit a bit i la negació lògica poden donar resultat diferent

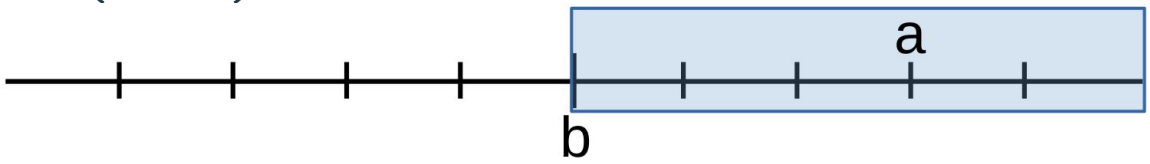
```
unsigned char a = 0x55;
unsigned char b, c;

b = !a;  // b = 0
c = ~a;  // c = 0xAA
```

- Traducció a MIPS (`a`, `b` i `c` estan a `$t0`, `$t1` i `$t2`):

```
sltiu $t1, $t0, 1 # b = !a;
nor $t2, $t0, $zero    # c = ~a;
```

Comparació “>=”

- $a \geq b \equiv \overline{(a < b)}$

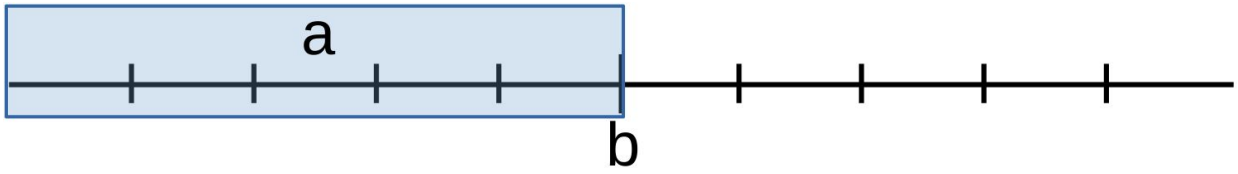


- Traduir `c = (a >= b);` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
slt $t4, $t0, $t1 # aux = (a < b)
sltiu $t2, $t4, 1 # c = !aux
```

Comparació “<=”

- $a \leq b \equiv \overline{(a > b)} \equiv \overline{(b < a)}$



- Traduir `c = (a <= b);` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
slt $t4, $t1, $t0 # aux = (b < a)
sltiu $t2, $t4, 1 # c = !aux
```

Conversió a valor lògic normalitzat

- Valors lògics en C:
  - 0: Fals
  - Diferent de 0: Cert
- Valors lògics normalitzats:
  - 0: Fals
  - 1: Cert
- Es pot normalitzar un valor lògic amb la instrucció `sltu`
  - `sltu $t1, $zero, $t0`
    - Si `$t0` és 0: `$t1 = 0`
    - Si `$t0` és diferent de 0: `$t1 = 1`

Comparacions “==” i “!=”

- Es tradueixen a una resta seguida de una negació lògica o de una normalització a `0` o `1`
- Traduir `c = (a == b);` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
sub $t4, $t0, $t1 # val zero si son iguals
sltiu $t2, $t4, 1 # negació lògica
```

- Traduir `c = (a != b);` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
sub $t4, $t0, $t1 # val zero si son iguals
sltu $t2, $zero, $t4   # normalitzar a 0 o 1
```

Operador and boolea “&&”

- Es pot utilitzar la instrucció `and` si abans normalitzem els valors a 0 o 1

- Traduir `c = a && b;` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
sltu $t3, $zero, $t0       # normalitzar a
sltu $t4, $zero, $t1       # normalitzar b
and $t2, $t3, $t4      # and bit a bit
```

and bit a bit “&” vs and lògica “&&”

- L’operador and bit a bit i l’operador and lògic poden donar resultat diferent

```
unsigned char a = 0x55; b = 0xAA, c;
c = a & b;    // c = 0
c = a && b;   // c = 1
```

- Traducció a MIPS suposant que `a`, `b` i `c` estan a `$t0`, `$t1` i `$t2`:

```
and $t2, $t1, $t0 # c = a & b;

sltu $t3, $zero, $t0
sltu $t4, $zero, $t1  } # c = a && b;
and $t2, $t3, $t4
```

Operador or boolea “||”

- Usar la instrucció `or` i normalitzar el resultat

- Traduir `c = a || b;` suposant que `a`, `b` i `c` es guarden a `$t0`, `$t1` i `$t2`:

```
or $t2, $t1, $t0          # or bit a bit
sltu $t2, $zero, $t2      # normalitzar
```

Salts

Salts condicionals relatius al PC

beq/bne i la macro b		
`beq rs, rt, label`	si (rs==rt) $PC = PC_{up} + Sext(offset16*4)$	
`bne rs, rt, label`	si (rs!=rt) $PC = PC_{up} + Sext(offset16*4)$	
`b label`	$PC = PC_{up} + Sext(offset16*4)$	`beq $0,$0 label`

- En assemblador, l'adreça de destí del salt s'especifica amb una etiqueta
- El format de la instrucció inclou un immediat de 16 bits (offset)
  - *offset16* codifica la distància a saltar respecte el PC
    - Diferencia entre la direcció de destí i la direcció de la següent instrucció
      - PC_{up} = PC + 4
      - offset16 = (label - PC_{up}) / 4
    - Permet saltar dintre del rang $[-2^{15}, 2^{15} - 1]$
  - Mode de direccionament relatiu al PC

Macros de salt

macros blt/bgt/bge/ble/bltu/bgtu/bgeu/bleu		
`blt rs, rt, label`	si (rs<rt) saltar a label	`slt $at, rs, rt` `bne $at, $zero, label`
`bgt rs, rt, label`	si (rs>rt) saltar a label[1]	`slt $at, rt, rs` `bne $at, $zero, label`
`bge rs, rt, label`	si (rs>=rt) saltar a label[2]	`slt $at, rs, rt` `beq $at, $zero, label`
`ble rs, rt, label`	si (rs<=rt) saltar a label[3]	`slt $at, rt, rs` `beq $at, $zero, label`

- Per naturals les macros `bltu, bgtu, bgeu` i `bleu` s'expandeixen de la mateixa manera, però usant `sltu`

Salts incondicionals

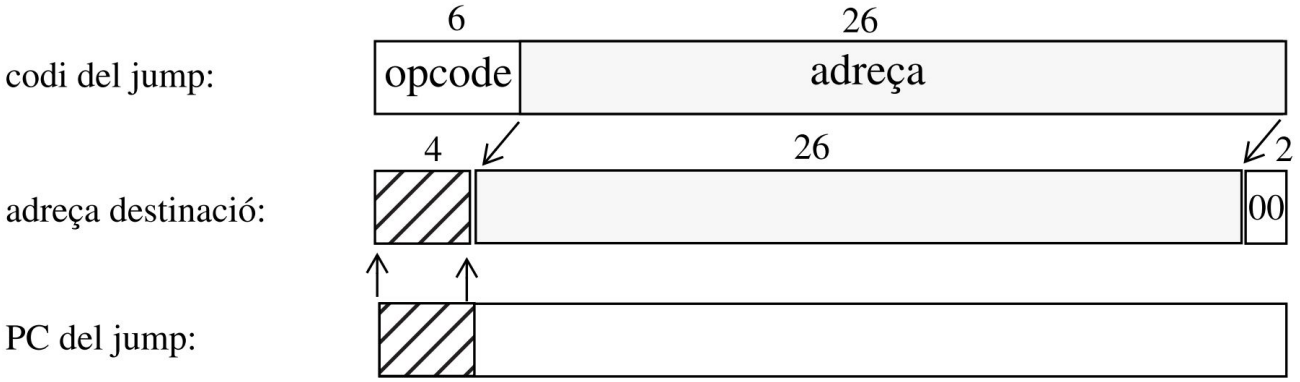
- La macro b està restringida al rang $[-2^{15},\ 2^{15} - 1]$
- Per salts que superen aquest rang s'utilitzen les següents instruccions

j/jr/jal/jalr		
`j target`	PC = target	Jump, mode pseudodirecte
`jr rs`	PC = rs	Jump, mode registre
`jal target`	PC = target; $ra = PC_{up}	Jump and Link, mode pseudodirecte
`jalr rd, rs`	PC = rs; rd = PC_{up}	Jump and Link, mode registre

- S'utilitzen per implementar subrutines

Salts incondicionals en mode pseudodirecte

- Les instruccions `j` i `jal` es codifiquen en format `J`
  - La direcció de destí es codifica en els 26 bits de menor pes de la instrucció



- Saltar a adreces que resideixen en un bloc de 2^{28} bytes
  - Altrament cal usar:

```
la $t0, etiqueta_llunyana
jr $t0
```

Recapitulació: modes d'adreçament

- Mode registre
  - `addu $t0, $t0, $t1`
- Mode immediat
  - `addiu $t0, $t0, 4`
- Mode memòria
  - `lw $t0, 0($t1)`
- Mode pseudodirecte
  - `j label`
- Mode relatiu al PC
  - `beq $t0, $zero, label`

Sentències alternatives: if-then-else

Sentència *if-then-else*

Sigui el cas general en C:

```
if (condicio)
      sentencia_then
else
      sentencia_else
```

El patró en MIPS serà:

avaluar `condicio`

salta si és falsa a `sino`

traducció de `sentencia_then`

salta a `fisi`

`sino:`

traducció de `sentencia_else`

`fisi:`

Exemple *if-then-else*

- Tradueix a MIPS el codi en C suposant que `a`, `b`, `c`, i `d` son enters emmagatzemats a `$t0`, `$t1`, `$t2` i `$t3`:

```
if (a >= b)
     d = a;
else
     d = b;
```

```
     blt $t0, $t1, sino     # salta si a<b
     move $t3, $t0
     b fisi
sino:
     move $t3, $t1
fisi:
```

Avaluació “lazy” dels operadors booleans “&&” i “||”

- Els operadors booleans `&&` i `||` s’avaluen d’esquerra a dreta de manera *lazy*
  - Si la part esquerra determina el resultat, la part dreta NO s’avalua

Exemple avaluació lazy amb &&

- Tradueix a MIPS el codi en C suposant que `a`, `b`, `c`, i `d` son enters emmagatzemats a `$t0`, `$t1`, `$t2` i `$t3`:

```
if (a >= b && a < c)
     d = a;
else
     d = b;
```

```
     blt $t0, $t1, sino     #salta si a<b
     bge $t0, $t2, sino     #salta si a>=c
     move $t3, $t0
     b fisi
sino:
     move $t3, $t1
fisi:
```

Exemple avaluació lazy amb ||

- Tradueix a MIPS el codi en C suposant que `a`, `b`, `c`, i `d` son enters emmagatzemats a `$t0`, `$t1`, `$t2` i `$t3`:

```
if (a >= b || a < c)
    d = a;
else
    d = b;
```

```
    bge $t0, $t1, then    #salta si a>=b
    bge $t0, $t2, sino    #salta si a>=c
then:
    move $t3, $t0
    b fisi
sino:
    move $t3, $t1
fisi:
```

Exercici

Donada la següent sentència escrita en alt nivell en C:

```
if (((a<=b)&&(b!=0)) || (((b%8)^0x0005)>0))
        z=5;
else
        z=a-b;
```

Completa el següent fragment de codi MIPS, que tradueix l'anterior sentència, escrivint en cada calaix un mnemònic d'instrucció o macro, etiqueta, registre o immediat. Les variables `a`, `b` i `z` són de tipus `int` i estan inicialitzades i guardades als registres `$t0`, `$t1` i `$t2`, respectivament.

```
        [          ] $t0, $t1, [          ]
etq1:   [          ] $t1, $zero, [          ]
etq2:   andi       $t3, $t1, [          ]
etq3:   [          ] $t5, $t3, [          ]
etq4:   ble        $t5, $zero, [          ]
etq5:   li         $t2, 5
etq6:   b          [          ]
etq7:   subu       $t2, $t0, $t1
etq8:
```