Estructura de Computadores

Tema 3. Traducción de Programas

Desplazamientos lógicos

- Shift Left Logical: `sll rd, rt, shamt`
  - Desplaza `rt` a la **izquierda** el número de bits indicado en el operando inmediato `shamt`
- Shift Right Logical: `srl rd, rt, shamt`
  - Desplaza `rt` a la **derecha** el número de bits indicado en el operando inmediato `shamt`
- Las posiciones que quedan vacantes se rellenan con ceros

Desplazamientos lógicos

- Shift Left Logical: `sll rd, rt, shamt`
  - Desplaza `rt` a la **izquierda** el número de bits indicado en el operando inmediato `shamt`
- Shift Right Logical: `srl rd, rt, shamt`
  - Desplaza `rt` a la **derecha** el número de bits indicado en el operando inmediato `shamt`
- Las posiciones que quedan vacantes se rellenan con ceros

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

Desplazamiento aritmético a la derecha

- Shift Right Arithmetic: `sra rd, rt, shamt`
  - Desplaza `rt` a la derecha el número de bits indicado en el operando inmediato `shamt`
- Las posiciones que quedan vacantes a la derecha se rellenan con una copia del bit de signo de `rt`

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

Desplazamiento indicado en registro

- `sllv rd, rt, rs`
- `srlv rd, rt, rs`
- `srav rd, rt, rs`
- El número de bits a desplazar se indica en los 5 bits de menor peso del registro `rs`, el resto de bits se ignoran

Multiplicación y división por potencias de 2

- Desplazar a la izquierda equivale a multiplicar por una potencia de 2
  - `sll rd, rt, shamt`
  - $rd = rt \times 2^{shamt}$
  - Útil para calcular el offset de un vector dado el índice
- Desplazar a la derecha un natural (con `srl`) o un entero (con `sra`) equivale a dividirlo por una potencia de 2
  - `srl rd, rt, shamt`
  - $rd = rt/2^{shamt}$

Ejemplo

```
int vec[100];

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

Ejemplo

```
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
        la $t2, vec
        addu $t2, $t2, $t1
        sw $zero, 0($t2)
```

Conversión de Ca2 a Ca1

- Representar en Ca1 un entero x_s guardado en Ca2 en `$t0`
  - Se puede convertir de Ca2 a Ca1 restando el bit de signo

Conversión de Ca2 a Ca1

- Representar en Ca1 un entero x_s guardado en Ca2 en $t0
  - Se puede convertir de Ca2 a Ca1 restando el bit de signo

```
# Desplazamos el bit de signo a la posicion 0
srl $t1, $t0, 31

# Restamos el bit de signo
subu $t0, $t0, $t1
```

Traducción de los operadores de desplazamiento en C

- C define dos operadores de desplazamiento de bits
  - Desplazamiento a la izquierda: $<<$
    - Se traduce a la instrucción `sll`
  - Desplazamiento a la derecha: $>>$
    - Para naturales, se traduce a la instrucción `srl`
    - Para enteros, se traduce a la instrucción `sra`

Traducción de los operadores de desplazamiento en C

- C define dos operadores de desplazamiento de bits
  - Desplazamiento a la izquierda: <<
    - Se traduce a la instrucción `sll`
  - Desplazamiento a la derecha: >>
    - Para naturales, se traduce a la instrucción `srl`
    - Para enteros, se traduce a la instrucción `sra`
- Traducir a MIPS la siguiente sentencia en C (a y b son enteros y están almacenados en $t0 y $t1)

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

Traducción de los operadores de desplazamiento en C

- C define dos operadores de desplazamiento de bits
  - Desplazamiento a la izquierda: <<
    - Se traduce a la instrucción `sll`
  - Desplazamiento a la derecha: >>
    - Para naturales, se traduce a la instrucción `srl`
    - Para enteros, se traduce a la instrucción `sra`
- Traducir a MIPS la siguiente sentencia en C (a y b son enteros y están almacenados en $t0 y $t1)

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

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

Operaciones lógicas bit a bit en MIPS

- Repertorio de instrucciones lógicas

`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)
`ori rt, rs, imm16`	rt = rs OR ZeroExt(imm16)
`xori rt, rs, imm16`	rt = rs XOR ZeroExt(imm16)

- `imm16` ha de ser un natural

Operaciones lógicas bit a bit - C vs MIPS

- Asumiendo que a, b y c están en $t0, $t1 y $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`

Ejemplo

- Traducir la siguiente sentencia en C:
  - `a = ~(a & b);`
- Operador "~": Negación bit a bit
- Asume que `a` y `b` se almacenan en los registros $t0 y $t1.

Ejemplo

- Traducir la siguiente sentencia en C:
  - `a = ~(a & b);`
- Operador "~": Negación bit a bit
- Asume que `a` y `b` se almacenan en los registros $t0 y $t1.

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

Seleccionar bits

- La instrucción `and` se utiliza para seleccionar determinados bits de un registro, poniendo el resto a 0

Seleccionar bits

- La instrucción `and` se utiliza para seleccionar determinados bits de un registro, poniendo el resto a 0
- Ejemplo: seleccionar los 16 bits de menor peso del registro $t0

```
andi $t1, $t0, 0xFFFF
```

Seleccionar bits

- La instrucción `and` se utiliza para seleccionar determinados bits de un registro, poniendo el resto a 0
- Ejemplo: seleccionar los 16 bits de menor peso del registro $t0

```
andi $t1, $t0, 0xFFFF
```

- Ejemplo: extraer los bits 0, 2, 4, 6

```
andi $t1, $t0, 0x0055
```

Seleccionar bits

- La instrucción `and` se utiliza para seleccionar determinados bits de un registro, poniendo el resto a 0
- Ejemplo: seleccionar los 16 bits de menor peso del registro $t0

```
andi $t1, $t0, 0xFFFF
```

- Ejemplo: extraer los bits 0, 2, 4, 6

```
andi $t1, $t0, 0x0055
```

- La instrucción `andi` se puede utilizar para calcular el resto de la división por potencias de 2

Utilidad de las operaciones lógicas bit a bit

- Poner bits a 1
  - Ejemplo: poner a 1 los 16 bits de menor peso de $t0

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

Utilidad de las operaciones lógicas bit a bit

- Poner bits a 1
  - Ejemplo: poner a 1 los 16 bits de menor peso de $t0

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

- Complementar bits
  - Ejemplo: complementar los bits pares de $t0

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

Comparaciones y operaciones booleanas

- En C las expresiones enteras admiten las operaciones de comparación
  - ==, !=, <, <=, >, >=

Comparaciones y operaciones booleanas

- En C las expresiones enteras admiten las operaciones de comparación
  - ==, !=, <, <=, >, >=
- No existe el tipo booleano, el resultado de una comparación es un entero
  - 0: falso
  - Distinto de 0: verdadero

Comparaciones y operaciones booleanas

- En C las expresiones enteras admiten las operaciones de comparación
  - ==, !=, <, <=, >, >=
- No existe el tipo booleano, el resultado de una comparación es un entero
  - 0: falso
  - Distinto de 0: verdadero
- También existen expresiones lógicas formadas por operadores booleanos
  - &&, ||, !

Comparaciones y operaciones booleanas

- En C las expresiones enteras admiten las operaciones de comparación
  - ==, !=, <, <=, >, >=
- No existe el tipo booleano, el resultado de una comparación es un entero
  - 0: falso
  - Distinto de 0: verdadero
- También existen expresiones lógicas formadas por operadores booleanos
  - &&, ||, !
- Los operadores booleanos devuelven un valor entero "normalizado": 0 o 1

Repertorio de instrucciones de comparación

`slt rd, rs, rt`	$rd = rs < rt$	enteros
`sltu rd, rs, rt`	$rd = rs < rt$	naturales
`slti rd, rs, imm16`	$rd = rs < Sext(imm16)$	enteros
`sltiu rd, rs, imm16`	$rd = rs < Sext(imm16)$	naturales

- Generan un valor lógico normalizado:
  - Si la comparación es falsa escriben 0 en rd
  - Si la comparación es cierta escriben 1 en rd

Comparaciones de tipo "<"

- Traducir a MIPS, suponiendo que a, b, c se almacenan en $t0, $t1, $t2

  c = a < b;

Comparaciones de tipo "<"

- Traducir a MIPS, suponiendo que a, b, c se almacenan en $t0, $t1, $t2

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

- Si a y b son naturales

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

Comparaciones de tipo "<"

- Traducir a MIPS, suponiendo que a, b, c se almacenan en $t0, $t1, $t2

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

- Si a y b son naturales

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

- Si a y b son enteros

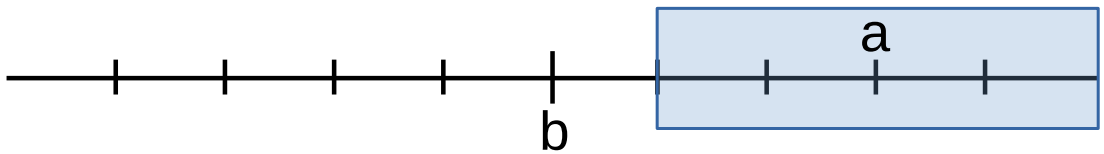
```
slt $t2, $t0, $t1
```

Comparaciones “>”, “>=”, “<=”

- MIPS solo incluye instruciones de comparación “menor que”
- Las otras desigualdades se pueden traducir usando “menor que”
  - $a > b \iff b < a$
  - $a \geq b \iff \overline{(a < b)}$
  - $a \leq b \iff \overline{(a > b)} \iff \overline{(b < a)}$

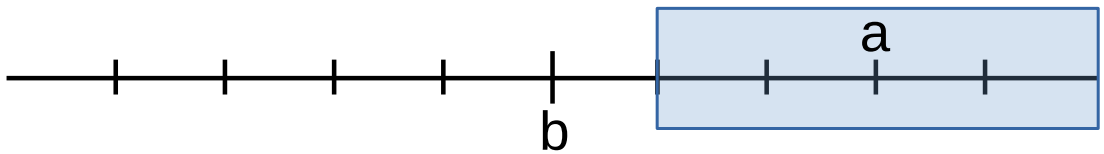
Comparación “>”

- $a > b \iff b < a$



Comparación ">"

- $a > b \iff b < a$



- Traducir c=(a>b); suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

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

Negación lógica "!"

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

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

Negación lógica "!"

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

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

- Traducción a MIPS con comparación "menor que 1" como natural
- `sltiu $t1, $t0, 1 # $t0=a, $t1=b`
  - Si $t0 es cero (falso): $t1 = 1 (cierto)
  - Si $t0 es distinto de cero (cierto): $t1 = 0 (falso)

Negación bit a bit "∼" vs Negación lógica "!"

- La negación bit a bit y la negación lógica puedan dar resultado distinto

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

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

Negación bit a bit "∼" vs Negación lógica "!"

- La negación bit a bit y la negación lógica puedan dar resultado distinto

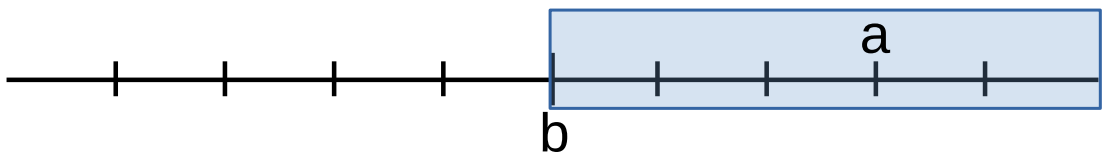
```
unsigned char a = 0x55;
unsigned char b, c;

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

- Traducción a MIPS (a, b y c están en $t0, $t1 y $t2):

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

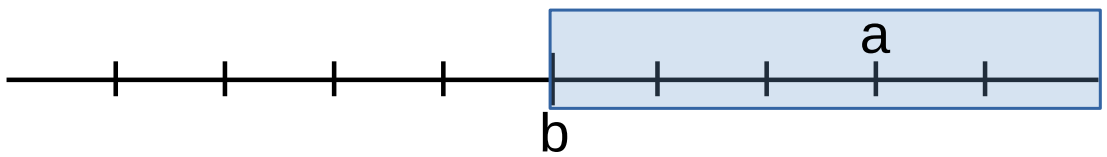
Comparación ">="

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



Comparación ">="

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

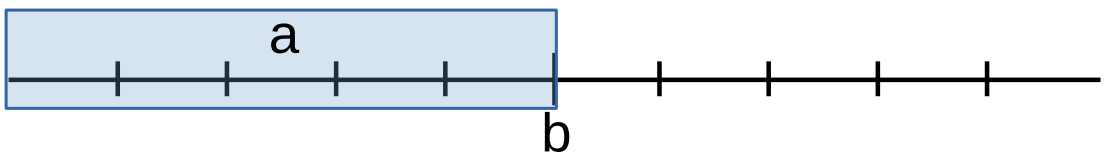


- Traducir `c=(a>=b);` suponiendo que `a`, `b`, y `c` se guardan en `$t0`, `$t1` y `$t2`:

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

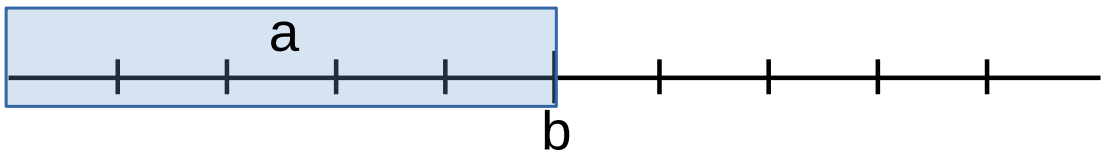
Comparación "<="

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



Comparación "<="

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



- Traducir `c=(a<=b);` suponiendo que `a`, `b`, y `c` se guardan en `$t0`, `$t1` y `$t2`:

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

Conversión a valor lógico normalizado

- Valores lógicos en C:
  - 0: Falso
  - Distinto de 0: Cierto

Conversión a valor lógico normalizado

- Valores lógicos en C:
  - 0: Falso
  - Distinto de 0: Cierto
- Valores lógicos normalizados:
  - 0: Falso
  - 1: Cierto

Conversión a valor lógico normalizado

- Valores lógicos en C:
  - 0: Falso
  - Distinto de 0: Cierto
- Valores lógicos normalizados:
  - 0: Falso
  - 1: Cierto
- Se puede normalizar el valor lógico con instrucción sltu
- `sltu $t1, $zero, $t0`
  - Si $t0 es 0: $t1 = 0
  - Si $t0 es distinto de 0: $t1 = 1

Comparaciones "==" y "!="

- Se traducen a una resta seguida de una negación lógica o de una normalización a 0 o 1

Comparaciones "==" y "!="

- Se traducen a una resta seguida de una negación lógica o de una normalización a 0 o 1
- Traducir `c=(a==b);` suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

```
sub $t4, $t0, $t1  # vale cero si son iguales
sltiu $t2, $t4, 1  # negacion logica
```

Comparaciones "==" y "!="

- Se traducen a una resta seguida de una negación lógica o de una normalización a 0 o 1
- Traducir c=(a==b); suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

```
sub $t4, $t0, $t1  # vale cero si son iguales
sltiu $t2, $t4, 1  # negacion logica
```

- Traducir c=(a!=b); suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

```
sub $t4, $t0, $t1    # vale cero si son iguales
sltu $t2, $zero, $t4 # normalizar a 0 o 1
```

Operador and booleano "&&"

- Se puede usar `and` si normalizamos antes los valores a 0 o 1
- Traducir `c = a && b;` suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

```
sltu  $t3, $zero, $t0
sltu  $t4, $zero, $t1
and   $t2, $t3, $t4
```

and bit a bit "&" vs and lógica "&&"

- El operador and bit a bit y el operador and lógico puedan dar resultado distinto

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

and bit a bit "&" vs and lógica "&&"

- El operador and bit a bit y el operador and lógico puedan dar resultado distinto

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

- Traducción a MIPS (a, b y c están en $t0, $t1 y $t2):

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

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

Operador or booleano "||"

- Usar instrucción `or` y normalizar el resultado
- Traducir `c = a || b;` suponiendo que a, b y c se guardan en $t0, $t1 y $t2:

```
or $t2, $t1, $t0
sltu $t2, $zero, $t2
```

Saltos condicionales relativos al PC

`beq rs, rt, label`	si (rs == rt) $PC = PC_{up} + Sext(offset16)$
`bne rs, rt, label`	si (rs != rt) $PC = PC_{up} + Sext(offset16)$
`b label`	`beq $0, $0, label`

- La dirección de destino del salto se especifica mediante una etiqueta en ensamblador
- La instrucción incluye un inmediato de 16 bits (offset16)
  - El offset16 codifica la distancia a saltar respecto al PC
    - Diferencia entre la dirección de destino y la dirección de la instrucción siguiente
    - $offset16 = (label - PC_{up})/4$
    - $PC_{up} = PC + 4$
    - Permite saltar dentro del rango $[-2^{15}, 2^{15} - 1]$
  - Modo de direccionamiento relativo al PC

Macros de salto

`blt rs, rt, label`	si(rs<rt) salta a label	`slt $at, rs, rt` `bne $at, $zero, label`
`bgt rs, rt, label`	si(rs>rt) salta a label	`slt $at, rt, rs` `bne $at, $zero, label`
`bge rs, rt, label`	si(rs>=rt) salta a label	`slt $at, rs, rt` `beq $at, $zero, label`
`ble rs, rt, label`	si(rs<=rt) salta a label	`slt $at, rt, rs` `beq $at, $zero, label`

- Para naturales, las macros `bltu`, `bgtu`, `bgeu` y `bleu` se expanden de la misma forma, usando `sltu`

Saltos incondicionales

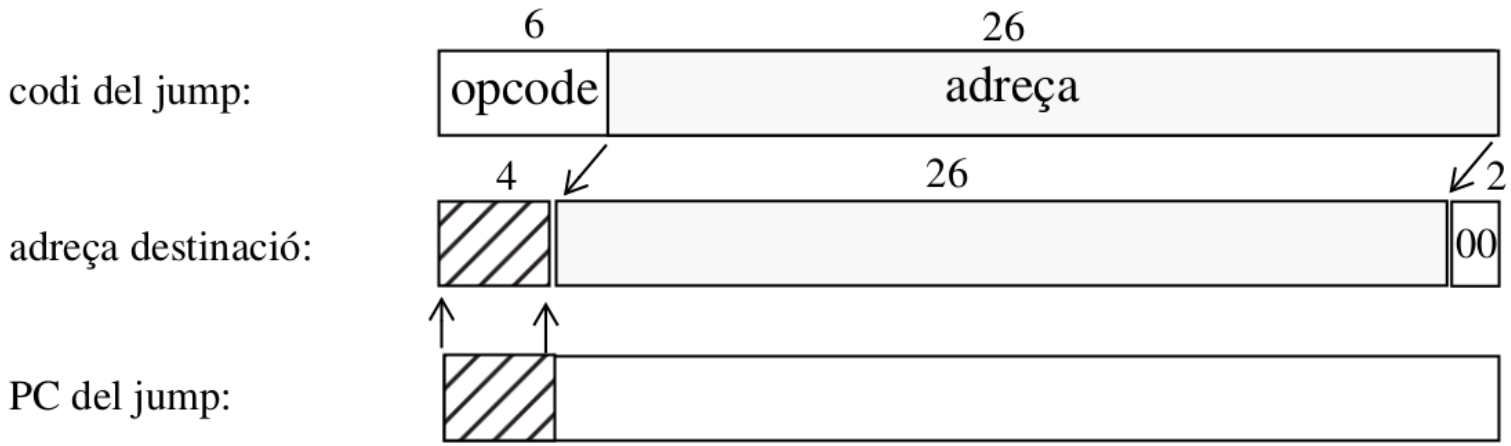
- La macro `b` está restringida al rango $[-2^{15}, 2^{15} - 1]$
- Para saltos que superen el rango, se utilizan las siguientes instrucciones:

`j target`	`PC = target`
`jr rs`	`PC = rs`
`jal target`	`PC=target; $ra =` PC_{up}
`jalr rs, rd`	`PC=rs; rd=`PC_{up}

- Se utilizan para implementar subrutinas

Saltos incondicionales en modo pseudodirecto

- Las instrucciones `j` y `jal` se codifican en el formato J
  - La dirección de destino se codifica en los 26 bits de menor peso de la instrucción



Modos de direccionamiento

- Modo registro
  - `addu $t0, $t0, $t1`
- Modo immediato
  - `addiu $t0, $t0, 4`
- Modo memoria
  - `lw $t0, 0($t1)`
- Modo pseudodirecto
  - `j label`
- Modo relativo al PC
  - `beq $t0, $zero, label`

Sentencia *if-then-else*

```
if (condicion)
    sentencia_then
else
    sentencia_else
```

Sentencia *if-then-else*

```
if (condicion)
    sentencia_then
else
    sentencia_else
```

```
    evaluar condición
    salta si es falsa a sino
    traducción de sentencia_then
    salta a fisi
sino:
    traducción de sentencia_else
fisi:
```

Sentencia *if-then-else*

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c`, y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

Sentencia *if-then-else*

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c`, y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

```
        blt $t0, $t1, sino
        move $t3, $t0
        b fisi
sino:
        move $t3, $t1
fisi:
```

Evaluación "lazy"

- En C, los operadores lógicos "&&" y "||" se evaluan de izquierda a derecha de forma "lazy"
  - Si la parte izquierda determina el resultado, la parte derecha NO se evalúa

Evaluación "lazy"

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c` y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

Evaluación "lazy"

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c` y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

```
    blt $t0, $t1, sino
    bge $t0, $t2, sino
    move $t3, $t0
    b fisi
sino:
    move $t3, $t1
fisi:
```

Evaluación "lazy"

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c` y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

Evaluación "lazy"

- Traduce a MIPS el código en C, suponiendo que `a`, `b`, `c` y `d` son enteros almacenados en $t0, $t1, $t2 y $t3:

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

```
        bge $t0, $t1, llavors
        bge $t0, $t2, sino
llavors:
        move $t3, $t0
        b fisi
sino:
        move $t3, $t1
fisi:
```

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:
```