



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 3: Traducción de Programas

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



Índice

- ❑ Desplazamientos de bits
- ❑ Operaciones lógicas bit a bit
- ❑ Comparaciones y operaciones Booleanas
- ❑ Saltos
- ❑ Sentencia Alternativa if-then-else
- ❑ Sentencias Iterativas while, for y do-while
- ❑ Sentencia Alternativa switch
- ❑ Subrutinas
- ❑ Estructura de la Memoria
- ❑ Compilación, ensamblado, enlazado y carga

Desplazamientos

- Operación típica: multiplicar por una potencia de 2 (2, 4, 8, ...)

```
void main() {  
    int x = 45;  
    int m;  
  
    m = x * 2;  
    m = x << 1;  
  
    m = x * 8;  
    m = x << 3;  
}
```

C

Operaciones
Equivalentes

```
li $t0, 0x11111111  
sll $t0, $t0, 2  
# $t0 = 0x44444444
```

MIPS

```
li $t0, 0x88888888  
sll $t0, $t0, 1  
# $t0 = 0x11111110 (?)
```

- Instrucción **sll** (Shift Left Logical)

```
sll rd, rt, shamt # rd = rt << shamt
```

Añade 0's por la derecha

shamt = shift amount, natural de 5 bits (0..31)

Desplazamientos

- Operación típica: dividir por una potencia de 2 (2, 4, 8, ...)

```
void main() {
    unsigned int x = 45;
    unsigned int d;

    d = x / 2;
    d = x >> 1;

    d = x / 8;
    d = x >> 3;
}
```

C

Operaciones
Equivalentes

```
li $t0, 0x11111111
srl $t0, $t0, 2
# $t0 = 0x04444444
```

MIPS

```
li $t0, 0x88888888
srl $t0, $t0, 1
# $t0 = 0x44444444
```

- Instrucción **srl** (Shift Right Logical)

```
srl rd, rt, shamt # rd = rt >> shamt
```

Añade 0's por la izquierda

shamt = shift amount, natural de 5 bits (0..31)

Desplazamientos lógicos a la izquierda y a la derecha

- Instrucción **sll** (Shift Left Logical)

```
sll rd, rt, shamt # rd = rt << shamt
```

Añade 0's por la derecha

- Instrucción **srl** (Shift Right Logical)

```
srl rd, rt, shamt # rd = rt >> shamt
```

Añade 0's por la izquierda

shamt = shift amount, natural de 5 bits (0..31)

- ¿Qué pasa si trabajamos con números enteros en ca2?

```
0xFF (-1) << 1 = 0xFE (-2)  
0x0F (15) << 2 = 0x3C (60)  
0xF9 (-7) << 1 = 0xF2 (-14)
```

```
0xFF (-1) >> 1 = 0x7F (127)  
0x0F (15) >> 2 = 0x03 (3)  
0xF9 (-7) >> 1 = 0x7C (124)
```

sll multiplica por 2

srl NO divide por 2 los números negativos

Desplazamientos aritmético a la derecha

- Instrucción **sra** (Shift Right Arithmetic)

sra rd, rt, shamt # rd = rt >> shamt

Extiende signo por la izquierda

shamt = shift amount, natural de 5 bits (0..31)

- ¿Qué pasa si trabajamos con números enteros en ca2?

0xFF (-1) >> 1 = 0xFF (-1)
0x0F (15) >> 2 = 0x03 (3)
0xF9 (-7) >> 1 = 0xFC (-4)

srl SI divide por 2 los números enteros.
Parte entera por abajo.

Repertorio instrucciones de desplazamiento

sll, srl, sra, sllv, srlv, srav		
sll rd, rt, shamt	$rd = rt \ll shamt$	
srl rd, rt, shamt	$rd = rt \gg shamt$	Inserta 0's a la izquierda
sra rd, rt, shamt	$rd = rt \gg shamt$	Extiende signo a la izquierda
sllv rd, rt, rs	$rd = rt \ll rs_{4:0}$	
srlv rd, rt, rs	$rd = rt \gg rs_{4:0}$	Inserta 0's a la izquierda
srav rd, rt, rs	$rd = rt \gg rs_{4:0}$	Extiende signo a la izquierda

- Las instrucciones **sllv**, **srlv** y **srav** permiten que el desplazamiento sea calculado.
 - El desplazamiento son los 5 bits de menor peso del registro rs ,

Repertorio instrucciones de desplazamiento

¿Cómo decidimos que desplazamiento hay que usar?:

- ❑ Operador <<
 - Instrucciones **sll** o **sllv**
- ❑ Operador >>
 - Instrucciones **srl** o **srlv**, si trabajamos con unsigned (naturales)
 - Instrucciones **sra** o **sraev**, si trabajamos con enteros
- ❑ Desplazar un entero/natural (**sll**) equivale a multiplicar por 2^{shamt} .
- ❑ Desplazar un natural (**srl**) o entero (**sra**) equivale a dividir por 2^{shamt} .
 - Pero, si el dividendo es negativo la instrucción **sra** dará resultado diferente que **div**

```
[unsigned] int a, b, c;
b = a << 2; // sll [sll]
b = a << c; // slv [slv]
b = a >> 2; // sra [srl]
b = a >> c; // sraev [srlv]
```

$$\begin{aligned} 1001 \ (-7) \gg 1 &= 1100 \ (-4) \\ 1001 \ (-7) \ / 2 &= 1101 \ (-3) \end{aligned}$$

$$\begin{aligned} -7 &= 2 * (-4) + 1 \\ -7 &= 2 * (-3) - 1 \end{aligned}$$

- En C, la división entera siempre devuelve el resto del mismo signo que el dividendo.
Por eso, sólo usaremos **sra** si estamos seguros que el dividendo no es negativo e impar

Operaciones and, or, xor y not bit a bit

❑ Ejemplos en C

```
a = a & b; // bitwise and  
a = a | b; // bitwise or  
a = a ^ b; // bitwise xor  
a = ~a;     // bitwise not
```

C

a	b	&
0	0	0
0	1	0
1	0	0
1	1	1

a	b	
0	0	0
0	1	1
1	0	1
1	1	1

❑ En MIPS

```
and $t0, $t0, $t1  
or $t0, $t0, $t1  
xor $t0, $t0, $t1  
nor $t0, $t0, $zero
```

MIPS

a	b	^
0	0	0
0	1	1
1	0	1
1	1	0

a	~
0	1
1	0

Repertorio instrucciones lógicas bit a bit

and, or, xor, nor, andi, ori, xori		
and rd, rs, rt	$rd = rs \& rt$	
or rd, rs, rt	$rd = rs rt$	
xor rd, rs, rt	$rd = rs ^ rt$	
nor rd, rs, rt	$rd = \sim(rs rt)$	
andi rt, rs, imm16	$rt = rs \& \text{ZeroExt}(imm16)$	imm16 es un natural
ori rt, rs, imm16	$rt = rs \text{ZeroExt}(imm16)$	imm16 es un natural
xori rt, rs, imm16	$rt = rs ^ \text{ZeroExt}(imm16)$	imm16 es un natural

Usos típicos de las Operaciones Lógicas bit a bit

- AND bit a bit, seleccionar bits dejando el resto a cero

```
andi $t0, $t0, 0xF  # bits 3:0 inalterados, el resto a cero
```

- OR bit a bit, activar determinados bits a 1 dejando el resto inalterado

```
ori $t0, $t0, 0x80  # pone el bit 7 a 1
```

- XOR bit a bit, complementar bits ($0 \rightarrow 1, 1 \rightarrow 0$)

```
xori $t0, $t0, 0xFF  # complementa los bits 7:0
```

- Comprobar si la variable b tiene activos los bits 0 y 4, e inactivos los bits 2 y 6

```
a = a & 0x55;  
if (a == 0x11)  
{ ... }
```

```
andi $t0, $t1, 0x1155  
li $t4, 0x0011  
bne $t0, $t4, endif  
....  
endif:
```

Comparaciones y Operaciones Booleanas

- En C no existe el tipo booleano, se usa un entero

- 0, equivale a FALSO
 - ≠0, equivale a CIERTO

C, **SIEMPRE** devuelve booleanos normalizados:

- CIERTO = 1
- FALSO = 0

- Operadores C que devuelven un valor booleano:

- Comparaciones de enteros/naturales: ==, !=, <, >, <=, >=
 - Operadores booleandos: &&, ||, !

NO confundir & (AND bit a bit) con && (AND lógico)

NO confundir | (OR bit a bit) con || (OR lógico)

NO confundir == (comparador igualdad) con = (asignación)

Instrucciones de Comparación en MIPS

- MIPS sólo implementa la función “<”
 - Devuelve un 0 si FALSO
 - Devuelve un 1 si CIERTO

slt, sltu, slti, sltiu		
slt rd, rs, rt	$rd = rs < rt$	Comparación de Enteros
sltu rd, rs, rt	$rd = rs < rt$	Comparación de Naturales
slti rt, rs, imm16	$rt = rs < \text{SignExt}(imm16)$	Comparación de Enteros
sltiu rt, rs, imm16	$rt = rs < \text{SignExt}(imm16)$	Comparación de Naturales

Instrucciones de Comparación en MIPS

- ❑ Ejemplo de traducción

```
c = a < b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
sltu $t2, $t0, $t1 # si a, b y c son naturales (unsigned)
slt $t2, $t0, $t1 # si a, b y c son enteros
```

```
c = a > b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
sltu $t2, $t1, $t0 # si a, b y c son naturales (unsigned)
slt $t2, $t1, $t0 # si a, b y c son enteros
```

- ❑ Utilizando **slu** (o **sltu**) y operaciones bit a bit podemos implementar todas las comparaciones y operaciones booleanas posibles.

Traducción de la negación booleana: !v

- ❑ Negación booleana:

```
c = !a; // 0 si a es CIERTO(≠0), 1 si a es FALSO(0)
```

```
sltiu $t2,$t0,1 #si $t0 es falso(0) (0<1) $t2=1 (CIERTO)  
#si $t0 es cierto(≠0) (≠0>=1) $t2=0 (FALSO)
```

- Si el valor a negar está normalizado, podemos usar:

```
xori $t2, $t0, 1 # complementa el bit 0 (0→1, 1→0)
```

- ❑ Con la negación booleana podemos implementar:

```
c = !(a < b); // (a >= b)  
c = !(a > b); // (a <= b)
```

Traducción de las operaciones booleanas && y ||

- Normalización de un booleano (en \$t0):

```
sltu $t0, $zero, $t0 # si $t0 es falso(0) (0<0) $t0=0  
                      # si $t0 es cierto(≠0) (0<≠0) $t0=1
```

- AND (&&) booleana

```
c = a && b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
sltu $t0, $zero, $t0 # Normaliza a  
sltu $t1, $zero, $t1 # Normaliza b  
and $t2, $t0, $t1    # c = a && b
```

- OR (||) booleana

```
c = a || b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
or $t2, $t0, $t1    # c = a | b  
sltu $t2, $zero, $t2 # Normaliza c
```

Sólo hay que normalizar el resultado

Traducción de las operaciones booleanas && y ||

- Las operaciones AND y OR suele aparecer en las condiciones de un if o un bucle.

```
if (a && b)
{ COD1 }
```



```
beq $t1, $zero, endif
beq $t2, $zero, endif
COD1
endif:
```

```
if (a || b)
{ COD2 }
```



```
bne $t1, $zero, if
beq $t2, $zero, endif2
if: COD2
endif2:
```

- En estos casos, no hay que guardar el resultado, evaluaremos la condición con saltos.
 - Es una evaluación **lazy (perezosa)**, lo veremos en detalle más adelante.

Traducción de las comparaciones >, <, >=, <=

- Supondremos que a, b, y c son enteros almacenados en \$t0, \$t1 y \$t2 respectivamente.

c = a < b;



slt \$t2, \$t0, \$t1 # a<b

c = a > b;



slt \$t2, \$t1, \$t0 # b<a, a>b

c = a <= b;



slt \$t4, \$t1, \$t0 # b<a, a>b
sltiu \$t2, \$t4, 1 # !(a>b), a<=b

c = a >= b;



slt \$t4, \$t0, \$t1 # a<b
sltiu \$t2, \$t4, 1 # !(a<b), a>=b

Traducción de las comparaciones ==, !=

- Supondremos que a, b, y c son enteros almacenados en \$t0, \$t1 y \$t2 respectivamente.

c = a == 0; → sltiu \$t2, \$t0, 1 # a<1 (natural)

c = a != 0; → sltu \$t2, \$zero, \$t0 # 0<a (natural)

c = a == b;
c = (a-b)==0 → sub \$t2, \$t0, \$t1 # t2=a-b
sltiu \$t2, \$t2, 1 # t2==0

c = a != b;
c = (a-b)!=0 → sub \$t2, \$t0, \$t1 # t2=a-b
sltu \$t2, \$zero, \$t2 # t2!=0

Saltos condicionales relativos al PC (branch)

- ❑ MIPS sólo implementa los saltos condicionales **beq** y **bne**

```
beq $t1, $t2, etiq # salta a etiq si $t1 == $t2
```

```
bne $t1, $t2, etiq # salta a etiq si $t1 != $t2
```

- ❑ Es fácil implementar un salto incondicional

```
beq $zero, $zero, etiq # salta a etiq (macro b)
```

- ❑ ¿Cómo codificamos el salto a una etiqueta determinada?

Saltos condicionales relativos al PC (branch)

¿Cómo codificamos el salto a una etiqueta determinada?

- Codificamos la distancia a saltar respecto al PC en un inmediato de 16 bits.

El PC es un registro interno que apunta a la instrucción que se está ejecutando.

- La distancia se codifica en número de instrucciones
- La distancia se calcula respecto al PC de la siguiente instrucción al salto ($PC_{UP} = PC + 4$)
- Permite saltar en un rango de $[-2^{15}..2^{15}-1]$ instrucciones de distancia respecto a PC_{UP} .

beq, bne y la macro b

beq rs, rt, label	si ($rs == rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	
bne rs, rt, label	si ($rs != rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	
b label	$PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	beq \$0, \$0, label

Otros Saltos condicionales relativos al PC (macros)

- ❑ Es muy habitual necesitar instrucciones de salto con condiciones del tipo $>$, $<$, \geq , \leq
- ❑ Usaremos macros

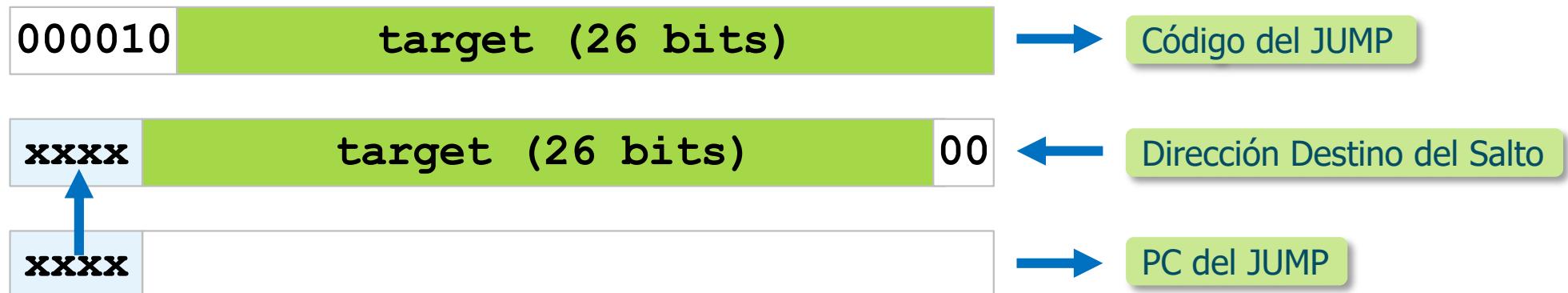
blt, bgt, ble, bge, bltu, bgtu, bgeu, bleu		
blt rs, rt, label	si ($rs < rt$) saltar a label	<code>slt \$at, rs, rt bne \$at, \$zero, label</code>
bgt rs, rt, label	si ($rs > rt$) saltar a label	<code>slt \$at, rt, rs bne \$at, \$zero, label</code>
bge rs, rt, label	si ($rs \geq rt$) saltar a label	<code>slt \$at, rs, rt beq \$at, \$zero, label</code>
ble rs, rt, label	si ($rs \leq rt$) saltar a label	<code>slt \$at, rt, rs beq \$at, \$zero, label</code>

- ❑ Saltos para enteros, para naturales usaremos las macros **bltu, bgtu, bgeu, bleu**

Saltos incondicionales “lejanos”

- Con los saltos bne y beq, sólo podemos saltar a una distancia limitada
 - [$-2^{15}..2^{15}-1$] instrucciones de distancia respecto a la instrucción de salto

j, jr		
j target	PC = target	Jump, modo pseudodirecto, formato J, target valor de 26 bits
jr rs	PC = rs	Jump, modo registro



- Hay 2 instrucciones “cercaas” **jal, jalr**, las explicaremos cuando hablemos de subrutinas.

Saltos incondicionales “lejanos”

- Con los saltos “`j target`”, también estamos “limitados”



- Podemos saltar dentro de un bloque de 256MB (2^{28})
- Si el rango de `j` es insuficiente, podemos utilizar el jump en modo registro

```
la $t0, FarLabel # $t0 <- @FarLabel  
jr $t0                # PC = $t0 = @FarLabel
```

Ejercicio 3.3

3.3 Escriu les seqüències de codi necessàries per escriure en \$t1 el resultat de les següents condicions lògiques, sense fer servir instruccions de salt. El valor final de \$t1 ha de ser 0 si no s'acompleix la condició, o diferent de zero altrament.

- a) L'enter representat per \$t1 és negatiu.
 - b) L'enter representat per \$t1 és múltiple de 4.
 - c) El natural representat per \$t1 és tal que el seu quadrat no es pot escriure amb 32 bits.
 - d) El natural representat per \$t1 és tal que la suma amb si mateix no es pot escriure amb 32 bits.
 - e) Els bits 1, 3 i 5 de \$t1 valen zero.
- \$t1 ← 0 (FALSO), \$t1 ← 1 o ≠0 (CIERTO)

Ejercicio 3.3

a) L'enter representat per \$t1 és negatiu.

- ☐ \$t1 negativo, eso equivale a ver si el bit de mayor peso vale 1.

```
li $t2,0x80000000
and $t1,$t1,$t2      # ya sería suficiente
slt $t1,$zero,$t1    # normaliza
```

```
srl $t1,$t1,31       # alternativa directa
```

Ejercicio 3.3

b) L'enter representat per \$t1 és múltiple de 4.

- \$t1 es múltiplo de 4, eso equivale a comprobar que los bits 1:0 son 00

```
andi $t1,$t1,0x03    # extraer los bits 1:0
sltu $t1,$zero,$t1    # normaliza
xori $t1,$t1,1        # !$t1
```

Ejercicio 3.3

c) El natural representat per \$t1 és tal que el seu quadrat no es pot escriure amb 32 bits.

- \$t1 es un natural tal que $\$t1^2$ no puede representarse en 32 bits
 - Si trabajamos con 8 bits tenemos
 - ✓ $00001111 = 15, 15^2 = 225$ que se puede representar en 8 bits
 - ✓ $00010000 = 16, 16^2=256$ que no puede representarse en 8 bits
 - Por tanto equivale a ver si alguno de los 16 bits de mayor peso es distinto de 0.

```
srl $t1,$t1,16      # obtener los bits 31-16
sltu $t1,$zero,$t1  # normaliza
```

Ejercicio 3.3

d) El natural representat per \$t1 és tal que la suma amb si mateix no es pot escriure amb 32 bits.

- \$t1 es un natural tal que \$t1+\$t1 no puede representarse en 32 bits
 - Equivale a ver si el bit de mayor peso es 1

```
srl $t1,$t1,31
```

Ejercicio 3.3

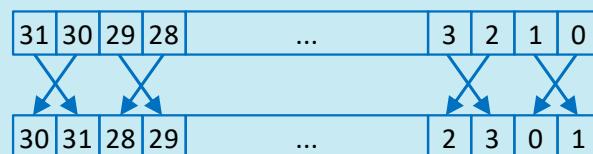
e) Els bits 1, 3 i 5 de \$t1 valen zero.

- ❑ Comprobar que los bits 1, 3 y 5 valen 0
 - Esos bits activos son 00101010 = 0x2A

```
andi $t1,$t1,0x2A    # extraer bits 1,3,5
sltu $t1,$zero,$t1    # normaliza (biti == 1)
xori $t1,$t1,1        # !$t1
```

Ejercicio 3.8

3.8 Escriu un fragment de codi en assemblador MIPS que intercanviï tots els bits parells amb els bits senars del registre \$t0:



```
sll $t1,$t0,1  
li $t3,0xAAAAAAA  
and $t1,$t1,$t3  
srl $t2,$t0,1  
li $t4,0x55555555  
and $t2,$t2,$t4  
or $t1,$t2,$t4
```

t0=

31	30	29	28	...	3	2	1	0
----	----	----	----	-----	---	---	---	---

t1=

30	29	28	27	...	2	1	0	0
----	----	----	----	-----	---	---	---	---

t3=

1	0	1	0	...	1	0	1	0
---	---	---	---	-----	---	---	---	---

t1=

30	0	28	0	...	2	0	0	0
----	---	----	---	-----	---	---	---	---

t2=

0	31	30	29	...	4	3	2	1
---	----	----	----	-----	---	---	---	---

t4=

0	1	0	1	...	0	1	0	1
---	---	---	---	-----	---	---	---	---

t2=

0	31	0	29	...	0	3	0	1
---	----	---	----	-----	---	---	---	---

t1=

30	31	28	29	...	2	3	0	1
----	----	----	----	-----	---	---	---	---

Ejercicio 3.9

3.9 Suposem que denotem el valor inicial de cada bit de \$t4 amb una lletra de la següent manera (en aquest exemple, el valor inicial del bit 2 és n i el bit 5 és k):

\$t4 = ABCD EFGH IJKL MNOP abcd efgh ijkl mnop

- a) Escriu les instruccions necessàries perquè el bit de més pes passi a ser el de menys pes i la resta de bits es desplacin una posició cap a l'esquerra, és a dir que el contingut final del registre \$t4 passi a ser:

\$t4 = BCDE FGHI JKLM NOPa bcde fghi jklm nopA

- b) Seguint la mateixa notació, escriu el contingut del registre \$t4 després d'executar les següents instruccions:

```
li $t1,16
      addiu $t2,$t4,0
bucle: srl $t4,$t4,1
      addu $t2,$t2,$t2
      addiu $t1,$t1,-1
      bne $t1,$zero,bucle
      or $t4,$t4,$t2
```

Ejercicio 3.9

- Rotar \$t4 1 bit a la izquierda, muchos LM tenían esta instrucción.

t4 =

31	30	29	28	...	3	2	1	0
----	----	----	----	-----	---	---	---	---



t4' =

30	29	28	27	...	2	1	0	31
----	----	----	----	-----	---	---	---	----

**sll \$t1,\$t4,1
srl \$t2,\$t4,31
or \$t4,\$t1,\$t2**

t4 =

31	30	29	28	...	3	2	1	0
----	----	----	----	-----	---	---	---	---

t1 =

30	29	28	27	...	2	1	0	0
----	----	----	----	-----	---	---	---	---

t2 =

0	0	0	0	...	0	0	0	31
---	---	---	---	-----	---	---	---	----

t4 =

30	29	28	27	...	2	1	0	31
----	----	----	----	-----	---	---	---	----

Muchos LM tienen
esta instrucción
(p.e.: x86)

Ejercicio 3.9

- ¿Qué hay en \$t4 al acabar? (p.e.: \$t4 = 0x02468ACE)

```
    li $t1, 16          # t1 ← 16
    addiu $t2,$t4,0    # t2 ← t4
bucle: srl $t4,$t4,1      # t4 ← t4 >> 1
    addu $t2,$t2,$t2    # t2 ← t2 << 1
    addiu $t1,$t1,-1    # t1--
    bne $t1,$zero,bucle # 16 iteraciones
    or $t4,$t4,$t2
```

- Al acabar el bucle tenemos:

$$\begin{aligned} \$t4 &= 0x00000246 \\ \$t2 &= 0x8ACE0000 \end{aligned}$$

or
→

$$\$t4 = 0x8ACE0246$$

Traducción Sentencia IF-THEN-ELSE

```
if (cond)
    CUERPO-IF
else
    CUERPO-ELSE
```

Modelo

```
int max(int x, int y) {
    int max;
    if (x>y) max = x;
        else max = y;
    return max;
}
```

Ejemplo

evaluar condición
saltar si es FALSO a **else**
if:
 CUERPO-IF
 saltar a **endif**
else:
 CUERPO-ELSE
endif:

PATRÓN de TRADUCCIÓN

Traducción Sentencia IF-THEN-ELSE

- Ejemplo de traducción, suponiendo que x, y, max son enteros y están en \$t0, \$t1 y \$t2 respectivamente.

```
if (x>y)
    max = x;
else
    max = y;
```

C

```
ble $t0,$t1,else    # salta si x<=y
if:
    move $t2,$t0        # max ← x
    b endif
else:
    move $t2,$t1        # max ← y
endif:
```

MIPS

Evaluación de condiciones con operadores && y ||

- En C, los operadores **&&** y **||** se evalúan **de izquierda a derecha** de forma **lazy** (perezosa).
- Si la parte izquierda ya determina el resultado, la parte derecha NO se ha de evaluar.
- NO es OPCIONAL.**
- Ejemplo evaluación LAZY con AND: a, b, c y d son enteros y están en \$t0, \$t1, \$t2 y \$t3

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

C

```
blt $t0,$t1,else      # salta si a<b
bge $t0,$t2,else      # salta si a>=c
if:
    move $t3,$t0          # d ← a
    b endif
else:
    move $t3,$t1          # d ← b
endif:
```

MIPS

Evaluación de condiciones con operadores && y ||

- ❑ Ejemplo evaluación LAZY con OR: a, b, c y d son enteros y están en \$t0, \$t1, \$t2 y \$t3

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

C

```
bge $t0,$t1,if      # salta si a>=b
bge $t0,$t2,else    # salta si a>=c
if:
    move $t3,$t0        # d ← a
    b endif
else:
    move $t3,$t1        # d ← b
endif:
```

MIPS

- Con **AND (&&)** en cuanto un componente es falso, la condición es falsa
- Con **OR (||)** en cuanto un componente es cierto, la condición es cierta

Evaluación de condiciones con operadores && y ||

- Las condiciones pueden llegar a ser bastante complejas:

```
if ((A&&B) || (C&&D))  
CUERPO-IF;
```

C

```
if ((A||B) && (C||D))  
CUERPO-IF;
```

si A falso saltar a OR
si B cierto saltar a if
OR:
 si C falso saltar a end
 si D falso saltar a end
if:
 CUERPO-IF
end:

MIPS

si A cierto saltar a AND
si B falso saltar a end
AND:
 si C cierto saltar a if
 si D falso saltar a end
if:
 CUERPO-IF
end:

Traducción Sentencia WHILE

```
while (cond) {  
    CUERPO WHILE  
}
```

Modelo

```
int gcd(int a, int b) {  
    while (b!=0) {  
        if (a>b) a = a-b;  
        else b = b-a;  
    }  
    return a;  
}
```

Ejemplo

```
while:  
    evaluar condición  
    saltar si es FALSO a end  
    CUERPO WHILE  
    saltar a while  
end:
```

PATRÓN de TRADUCCIÓN

Traducción Sentencia WHILE

- ❑ Ejemplo de traducción, división entera, supondremos que dd, dr y q son enteros y están en \$t1, \$t2 y \$t3 respectivamente.

```
q = 0;  
while (dd >= dr) {  
    dd = dd - dr;  
    q++;  
}
```

C

```
move $t3, $zero      # q = 0  
while:  
    blt $t1,$t2,end  # si falso end  
    subu $t1,$t1,$t2  # dd = dd-dr  
    addiu $t3,$t3,1   # q++  
    b while  
end:
```

MIPS

- ❑ El cuerpo del WHILE, puede ser cualquier cosa: otro WHILE, un IF-THEN-ELSE, ...

Traducción Sentencia WHILE

- En cada iteración tenemos 2 saltos. Los saltos son una de las cosas que más perjudica el rendimiento de un computador. Podemos reescribir el código para tener 1 solo salto por iteración.

```
q = 0;  
while (dd >= dr) {  
    dd = dd - dr;  
    q++;  
}
```

C

```
move $t3,$zero      # q = 0  
blt $t1,$t2,end    # si falso end  
while:  
    subu $t1,$t1,$t2  # dd = dd-dr  
    addiu $t3,$t3,1    # q++  
    bge $t1,$t2,while  # si cierto while  
end:
```

MIPS

Traducción Sentencia WHILE

- Si queremos nos podemos ahorrar escribir 2 veces la condición (que puede llegar a ser muy compleja), añadiendo un salto.

```
q = 0;  
while (dd >= dr) {  
    dd = dd - dr;  
    q++;  
}
```

C

```
move $t3,$zero      # q = 0  
b test              # goto test  
while:  
subu $t1,$t1,$t2    # dd = dd-dr  
addiu $t3,$t3,1     # q++  
test:  
bge $t1,$t2,while  # si cierto while  
end:
```

MIPS

Traducción Sentencia FOR

```
for (INI; COND; INC) {  
    CUERPO-FOR  
}
```

Modelo

```
int sumV(int V[], int N) {  
    int sum, i;  
    sum = 0;  
    for (i=0; i<N; i++)  
        sum = sum + V[i];  
    return sum;  
}
```

Ejemplo

INI
for:
evaluar COND
saltar si es FALSO a **end**
CUERPO-WHILE
INC
saltar a **for**
end:

```
INI  
while (COND) {  
    CUERPO-FOR  
    INC  
}
```

Traducción Sentencia FOR

- Ejemplo de traducción, suma de los elementos de un vector, supondremos que sum e i son enteros y están en \$t1 y \$t2 respectivamente, y la dirección de v[0] está en \$t0.

```
sum = 0;  
for (i=0; i<N; i++)  
    sum = sum + V[i];
```

C

```
move $t1,$zero      # sum = 0  
move $t2,$zero      # i = 0  
li $t3,N  
for:  
    bge $t2,$t3,end      # si i>=N end  
    lw $t4,0($t0)          # t4 = v[i]  
    addu $t1,$t1,$t4       # sum += v[i]  
    addiu $t0,$t0,4         # t0=&v[i+1]  
    addiu $t2,$t2,1         # i++  
    b for  
end:
```

MIPS



Traducción Sentencia DO-WHILE

```
do {  
    CUERPO-DO  
} while (cond)
```

Modelo

```
int ContA(char v[]) {  
    int i = 0, cont = 0;  
    do {  
        if (v[i] == 'a') cont++;  
        i++;  
    } while (v[i] != '.' );  
    return cont;  
}
```

Ejemplo

do:
 CUERPO-DO
 evaluar condición
 saltar si CIERTO a do
end:

PATRÓN de TRADUCCIÓN

SIEMPRE se ejecuta la 1^a iteración.



Traducción Sentencia SWITCH

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4: days = 30; break;  
    case 6: days = 30; break;  
    case 9: days = 30; break;  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

Ejemplo sin fallthrough

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4:  
    case 6:  
    case 9:  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

Ejemplo con fallthrough

- ❑ Ejemplo, ¿cuántos días tiene el mes X?

Traducción Sentencia SWITCH

```
int month; int TotDays=0;  
switch(month) {  
    case 12: TotDays += 31;  
    case 11: TotDays += 30;  
    case 10: TotDays += 31;  
    case 9: TotDays += 30;  
    case 8: TotDays += 31;  
    ...  
}
```

Ejemplo sin breaks

```
int i, month; int TotDays;  
int v[12] = {31,28,31,...};  
TotDays = 0;  
for (i=0; i<=month; i++)  
    TotDays += v[i];
```

Equivalente

- ❑ Ejemplo, ¿Número de días **hasta** el mes X?

Traducción Sentencia SWITCH

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4: days = 30; break;  
    case 6: days = 30; break;  
    case 9: days = 30; break;  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

Ejemplo sin fallthrough

En general, es poco eficiente.

```
int month; int days;  
if (month == 2)  
    days = 28;  
else if (month == 4)  
    days = 30;  
else if (month == 6)  
    days = 30;  
else if (month == 9)  
    days = 30;  
else if (month == 11)  
    days = 30;  
else  
    days = 31;
```

Se puede traducir con if-then-else

Traducción Sentencia SWITCH

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4:  
    case 6:  
    case 9:  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

Ejemplo con fallthrough

En general, es poco eficiente.

```
int month; int days;  
if (month == 2)  
    days = 28;  
else if (month == 4 ||  
        month == 6 ||  
        month == 9 ||  
        month == 11)  
    days = 30;  
else days = 31;
```

Se puede traducir con if-then-else

Traducción Sentencia SWITCH

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4: days = 30; break;  
    case 6: days = 30; break;  
    case 9: days = 30; break;  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

Ejemplo sin fallthrough

- La forma más eficiente de traducir un **switch** es utilizando un vector de direcciones a etiquetas: **JumpTable**.
- No siempre es posible

.data
JumpTable:
.word default
.word case2
.word default
.word case4
.word default
.word case6
...

MIPS

Traducción Sentencia SWITCH

```
int month; int days;  
switch(month) {  
    case 2: days = 28; break;  
    case 4: days = 30; break;  
    case 6: days = 30; break;  
    case 9: days = 30; break;  
    case 11: days = 30; break;  
    default: days = 31;  
}
```

\$t1 months
\$t2 days

```
JumpTable: .word default,case2,...  
...  
addiu $t0,$t1,-1  
sll $t0,$t0,2  
la $t4,JumpTable  
addu $t0,$t4,$t0  
lw $t0,0($t0)  
jr $t0  
case2: li $t2,28  
b end  
case4: li $t2,30  
b end  
default: li $t2,31  
end:
```

MIPS



Ejercicio: problema 5 del parcial de 2018-19Q2

Donada la següent sentència escrita en alt nivell en C:

```
if ( ((a<=b) && (b!=)) ||  
      ((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 initialitzades i guardades als registres \$t0, \$t1 i \$t2, respectivament.

	\$t0,\$t1,	
etiq1:		\$t1,\$zero,
etiq2:	andi	\$t3,\$t1,
etiq3:		\$t5,\$t3,
etiq4:	ble	\$t5,\$zero,
etiq5:	li	\$t2,5
etiq6:	b	
etiq7:	subu	\$t2,\$t0,\$t1
etiq8:		

Ejercicio: problema 5 del parcial de 2018-19Q2

	bgt	\$t0,\$t1,	etiq2
etiq1:		\$t1,\$zero,	
etiq2:	andi	\$t3,\$t1,	
etiq3:		\$t5,\$t3,	
etiq4:	ble	\$t5,\$zero,	
etiq5:	li	\$t2,5	
etiq6:	b		
etiq7:	subu	\$t2,\$t0,\$t1	
etiq8:			

```
if (((a<=b) && (b != 0)) || (((b%8)^0x0005)>0))
z = 5;
else
z = a-b; // a,b,z están en $t0,$t1,$t2
```

Ejercicio: problema 5 del parcial de 2018-19Q2

	bgt	\$t0,\$t1,	etiq2
etiq1:	bne	\$t1,\$zero,	etiq5
etiq2:	andi	\$t3,\$t1,	
etiq3:		\$t5,\$t3,	
etiq4:	ble	\$t5,\$zero,	
etiq5:	li	\$t2,5	
etiq6:	b		
etiq7:	subu	\$t2,\$t0,\$t1	
etiq8:			

```
if (((a<=b) && (b != 0)) || (((b%8)^0x0005)>0))
z = 5;
else
z = a-b; // a,b,z están en $t0,$t1,$t2
```

Ejercicio: problema 5 del parcial de 2018-19Q2

	bgt	\$t0,\$t1,	etiq2
etiq1:	bne	\$t1,\$zero,	etiq5
etiq2:	andi	\$t3,\$t1,	
etiq3:		\$t5,\$t3,	
etiq4:	ble	\$t5,\$zero,	
etiq5:	li	\$t2,5	
etiq6:	b		etiq8
etiq7:	subu	\$t2,\$t0,\$t1	
etiq8:			

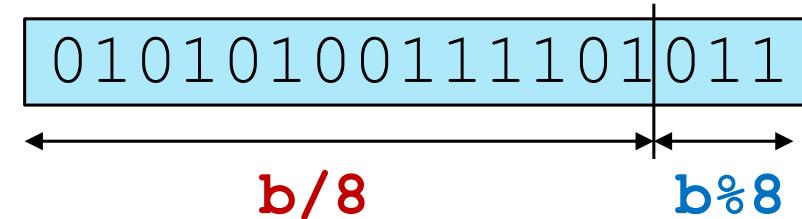
```
if (((a<=b) && (b !=0)) || (((b%8)^0x0005)>0))
    z = 5;
else
    z = a-b; // a,b,z están en $t0,$t1,$t2
```

Ejercicio: problema 5 del parcial de 2018-19Q2

	bgt	\$t0,\$t1,	etiq2	
etiq1:	bne	\$t1,\$zero,	etiq5	
etiq2:	andi	\$t3,\$t1,	0x07	
etiq3:		\$t5,\$t3,		
etiq4:	ble	\$t5,\$zero,		
etiq5:	li	\$t2,5		
etiq6:	b		etiq8	
etiq7:	subu	\$t2,\$t0,\$t1		
etiq8:				

b%8, es b mod 8

Los módulos de potencias de 2 son muy fáciles de calcular.



```
if (((a<=b) && (b != 0)) || (((b%8)^0x0005)>0))
    z = 5;
else
    z = a-b; // a,b,z están en $t0,$t1,$t2
```

Ejercicio: problema 5 del parcial de 2018-19Q2

	bgt	\$t0,\$t1,	etiq2
etiq1:	bne	\$t1,\$zero,	etiq5
etiq2:	andi	\$t3,\$t1,	0x07
etiq3:	xori	\$t5,\$t3,	0x05
etiq4:	ble	\$t5,\$zero,	etiq7
etiq5:	li	\$t2,5	if (((a<=b) && (b !=0)) (((b%8)^0x0005)>0))
etiq6:	b		z = 5;
etiq7:	subu	\$t2,\$t0,\$t1	else
etiq8:			z = a-b; // a,b,z están en \$t0,\$t1,\$t2

Subrutinas

SUBRUTINA. Es un conjunto de instrucciones de Lenguaje Maquina que realiza una tarea específica y que puede ser invocada desde cualquier punto del programa o desde la propia subrutina.

Ventajas del uso de subrutinas

- ❑ El código ocupa **menos espacio** en memoria
- ❑ El código está **más estructurado** (programación modular)
 - facilidad de depuración
 - facilidad de **expansión o modificación**
 - posibilidad de **usar librerías públicas**
- ❑ El LM refleja la idea fundamental de los lenguajes estructurados de alto nivel: la existencia de **funciones, procedimientos y métodos**

Subrutinas

SUBRUTINA. Es un conjunto de instrucciones de Lenguaje Maquina que realiza una tarea específica y que puede ser invocada desde cualquier punto del programa o desde la propia subrutina.

- ❑ Ejecutan una tarea determinada utilizando unos parámetros de entrada y pueden devolver un resultado (funciones).
- ❑ Son uno de los elementos esenciales de cualquier Lenguaje de Programación.
- ❑ Para poder utilizar subrutinas escritas por otros (p.e. librerías) necesitamos un estándar.
- ❑ Este estándar indica cómo se pasan los parámetros, qué reglas sigue la función, cómo se recogen los resultados.
- ❑ Estas reglas pueden ser diferentes para cada Lenguaje y Sistema Operativo
- ❑ Estas reglas son el **ABI (Application Binary Interface)**

Subrutinas - Terminología

- ❑ Parámetros
 - Valor
 - Referencia
- ❑ Variables locales
- ❑ Invocación
- ❑ Retorno resultado
- ❑ Cuerpo subrutina

En C todos los parámetros son “por valor”

```
int DOT(int v1[], int v2[], int N) {  
    int i, sum;  
    sum = 0;  
    for (i=0; i<N; i++)  
        sum += v1[i] * v2[i];  
    return sum;  
}  
  
void PDOT(int M[10][10], int *p) {  
    int i;  
    *p = 0;  
    for (i=0; i<10; i++)  
        *p += DOT(&M[0][0], &M[i][0], 10);  
}
```

C

Subrutinas – Llamada y retorno

- Uno de los primeros problemas a resolver es cómo llamamos a una subrutina y cómo volvemos al punto en que la subrutina fue invocada.

```
int max(int a, int b) {  
    if (a>b) return a;  
    else return b;  
}
```

```
...  
y = max(r,s);  
...  
...  
y = max(r,s);  
...
```

C

```
...  
b max  
addr1:  
...  
  
b max  
addr2:  
...
```

MIPS

```
max:  
...  
b addr1
```

¿Qué ocurre si hacemos más de 1 llamada a la misma subrutina?

Hay que buscar otra solución

Subrutinas – Llamada y retorno

- Uno de los primeros problemas a resolver es cómo llamamos a una subrutina y cómo volvemos al punto en que la subrutina fue invocada.

```
int max(int a, int b) {  
    if (a>b) return a;  
    else return b;  
}
```

```
...  
y = max(r,s);  
...
```

```
...  
y = max(r,s);  
...
```

C

Utilizaremos
jal y jr

```
max:  
    ...  
    jr $ra # PC = $ra
```

```
    ...  
    jal max # $ra = addr1
```

addr1:

...

```
    jal max # $ra = addr2
```

addr2:

...

MIPS

Subrutinas – Llamada y retorno

- Instrucciones que usamos para llamar y retornar de subrutinas: **jal**, **jalr**, **jr**

j, jr		
jal target	PC = target, \$ra = PC _{UP}	Jump and Link, modo pseudodirecto
jalr rs, rd	PC = rs, rd = PC _{UP}	Jump and Link, modo registro
jr rs	PC = rs	Jump, modo registro

- La dirección de retorno es el PC de la instrucción que hay después de la llamada ($PC_{UP}=PC+4$)

Subrutinas – Paso de Parámetros y Resultados

- ❑ Pasaremos Parámetros y Resultados de la forma más rápida posible: **REGISTROS**

Registros	Uso
\$a0, \$a1, \$a2, \$a3	Paso de Parámetros (argumentos)
\$v0, \$v1	Retorno resultado (usaremos sólo \$v0)
\$ra	Dirección retorno subrutina

- ❑ Los parámetros a una función se pasan en los registros \$a0–\$a3 **EN ORDEN**
 - Empezando por la izquierda el primero en \$a0, el segundo en \$a1, etc
 - Los parámetros de coma flotante se pasan en \$f12 (1er) y \$f14 (2º), máximo 2.
- ❑ El resultado se devuelve en el registro \$v0 (si es como flotante en \$f0)
- ❑ Si el parámetro/resultado tiene menos de 32 bits, se ha de extender adecuadamente.
- ❑ **Por simplicidad, en EC supondremos que hay 4 parámetros como máximo, no usaremos longs, ni doubles, ni structs, etc**

Subrutinas - Ejemplo

```
int suma2(int a, int b) {  
    return a+b;  
}  
  
void main(){  
    int x,y,z; // en t0,t1,t2  
    ...  
    z = suma2(x,y)  
    ...  
}
```

C

```
main:  
    ...  
    move $a0,$t0 # pasamos x  
    move $a1,$t1 # pasamos y  
    jal suma2  
    move $t2,$v0 # z = resultado  
    ...  
  
suma2:  
    addu $v0,$a0,$a1  
    jr $ra
```

MIPS

- Como conocemos el protocolo (**ABI**), podemos llamar a **suma2** sin necesidad de saber cómo está hecha.

Subrutinas – Tipos de Parámetros

- ❑ Tradicionalmente existen 2 tipos de parámetros
 - Parámetros **POR VALOR**. La función recibe una copia del parámetro y se puede modificar sin que afecte al parámetro en la rutina que llama a la función.
 - Parámetros **POR REFERENCIA**. Las modificaciones que se realizan en la función afectan directamente al parámetro real. Para implementar este tipo de parámetros se pasa la dirección del parámetro, para que la función pueda acceder al dato.
- ❑ En C, todos los parámetros son **por VALOR**.
- ❑ Pero, si el parámetro es un **puntero**, equivale a tener un parámetro **POR REFERENCIA**.

Subrutinas - Ejemplos

```
void swap(int *a, int *b) {  
    int tmp;  
    tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

```
int sumV(int *v, int N) {  
    int i, sum = 0;  
    for (i=0; i<N; i++)  
        sum = sum + v[i];  
}
```

C

```
int sumV(int v[], int N) {  
    int i, sum = 0;  
    for (i=0; i<N; i++)  
        sum = sum + v[i];  
}
```

```
int swapV(int v[100]) {  
    int i;  
    for (i=0; i<100; i=i+2)  
        swap(&v[i], &v[i+1]);  
}
```

```
int f(int M[10][100]);  
int f(int M[][] [90]);
```

iAtención con las matrices!

Subrutinas - Ejemplo

- ¿Qué declaración corresponde con cada llamada?

```
short v1[10];
char v2[20];
int a;
int *p;
```

C

(1) a = f(v1);

(a) int f(int i);

(2) a = f(p);

(b) int f(char c);

(3) a = f(&v2[10]);

(c) int f(short *pi);

(4) a = f(v2[10]);

(d) int f(char vc[]);

(5) a = f(*p);

(e) int f(int vi[]);



Subrutinas - Ejemplo

```
int v1[10];
int initV(int v2[]) {
    int i;
    for (i=0; i<10; i++)
        v2[i] = 0;
}

void main() {
    ...
    initV(v1);
    ...
}
```

C

MIPS

```
main:
    ...
    la $a0, v1          # a0←@v1
    jal initV           # call initV
    ...

initV:
    li $t0,0            # i←0
    li $t1,10             # t1←10
    for: bge $t0,$t1,end # i≥10 goto end
          sll $t2,$t0,2      # t2←i*4
          addu $t2,$t2,$a0     # t2←@v2[i]
          sw $zero,$($t2)       # v2[i]←0
          addiu $t0,$t0,1        # i++
          b for                 # goto for
end: jr $ra               # retorno
```



Subrutinas – Variables Locales

- ❑ Las **variables locales** se declaran dentro de una función y se crean y se destruyen con cada llamada.
- ❑ Las **variables locales** se han de inicializar de forma explícita, sino el valor es indeterminado.

- ❑ ¿Dónde guardamos las **variables locales**?
 - Si son variables escalares se guardan en registros :
 - ✓ Floats de simple precisión: **\$f0-\$f31**
 - ✓ El resto: **\$t0-\$t9 , \$s0-\$s7, \$v0-\$v1**
 - Algunas variables se han de almacenar en memoria [en **la pila (stack)**] :
 - ✓ Si son de tipo estructurado (vectores, matrices, tuplas, ...)
 - ✓ Si una variable local **v** aparece con **&v**
 - ✓ Si nos quedamos sin registros

```
int f(...) {  
    int i, sum = 0;  
    char frase[100];  
    ...  
}
```

Registros

Número	Nombre	Descripción
\$0	\$zero	Valor 0, read only
\$1	\$at	Reservado
\$2, \$3	\$v0, \$v1	Retornos de Subrutinas
\$4, \$5, \$6, \$7	\$a0, \$a1, \$a2, \$a3	Argumentos de Subrutinas
\$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15	\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7	Valores Temporales
\$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23	\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7	Variables Locales
\$24, \$25	\$t8, \$t9	Valores Temporales
\$26, \$27	\$k0, \$k1	Reservado SO
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

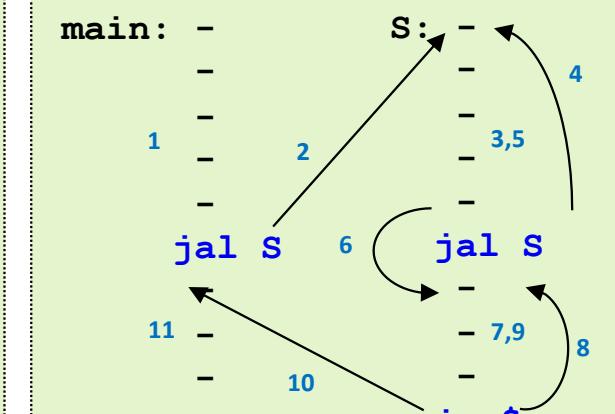
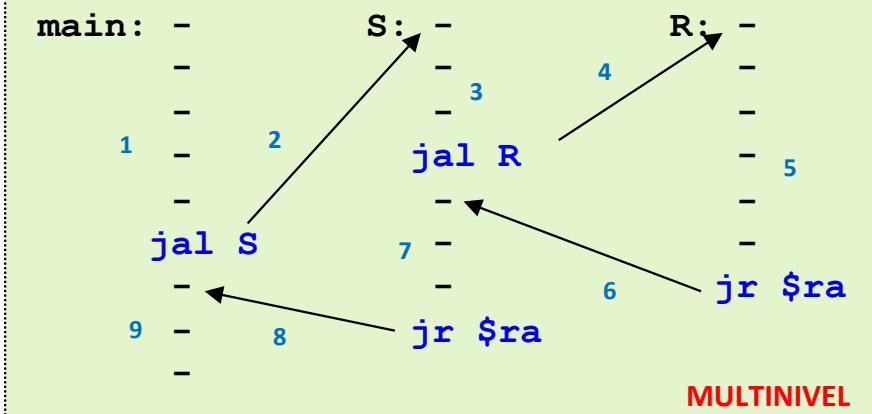
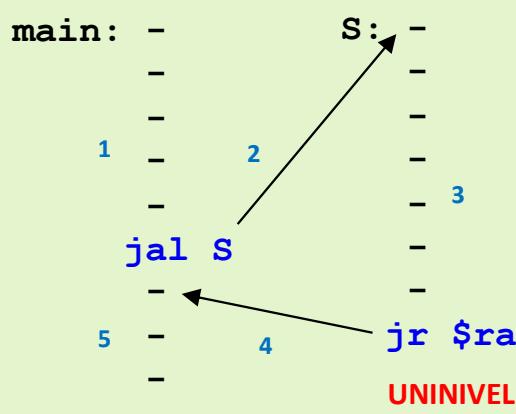
Tipos de subrutinas

```
main() {  
    S();  
}
```

```
S() {  
}
```

```
S() {  
    R();  
}
```

```
S() {  
    S();  
}
```

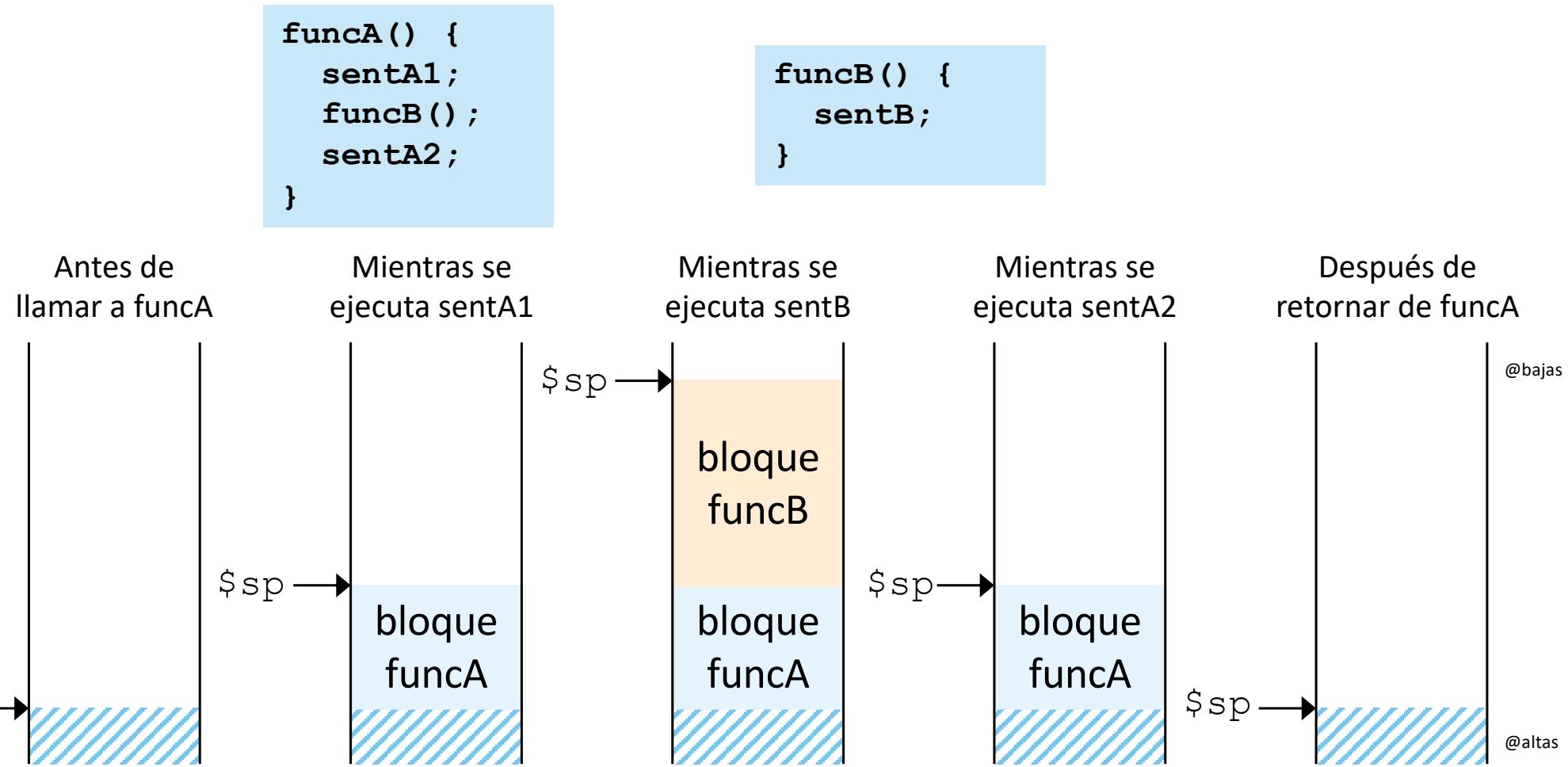


Para gestionar las matrices multinivel (y recursivas), necesitamos una estructura de datos especial: el BLOQUE de ACTIVACIÓN

Subrutinas – La Pila y el Bloque de Activación

- Todos los programas tienen una pila
 - Es una estructura de datos que existe en todos los computadores
 - La pila, en MIPS, comienza en la dirección **0x7FFFFFFC**
 - Las pilas crecen hacia direcciones más bajas.
 - El registro **\$sp** siempre apunta a la cima de la pila
 - ✓ La cima de la pila es el primer byte del último bloque de activación creado.
 - Cuando se inicia un programa la pila está vacía.
 - Cuando se inicia una subrutina reserva espacio en la pila para su bloque de activación, restando al **\$sp** el tamaño en bytes que ocupa el bloque de activación
 - Al acabar la subrutina, libera espacio sumando al **\$sp** la misma cantidad.

Subrutinas – La Pila y el Bloque de Activación



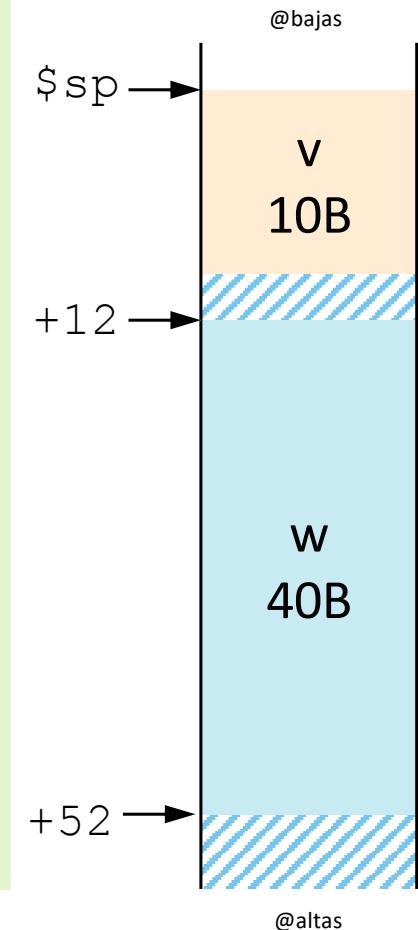
Subrutinas – La Pila y el Bloque de Activación

- ❑ El bloque de activación formado por las variables locales ha respetar ciertas reglas:
 - Las variables locales que van a la pila se colocan siguiendo el mismo orden en que están declaradas, empezando por la dirección más baja en el top de la pila.
 - Las variables que van en la pila, siguen las mismas reglas que las variables globales
 - ✓ alineadas al tamaño de su tipo, dejando sin uso los bytes necesarios
 - La dirección inicial y el tamaño del bloque de activación ha de ser siempre múltiplo de 4.

Subrutinas – Ejemplo

```
char func(int i){  
    char v[10];  
    int w[10], k;  
    ...  
    return v[w[i]+k];  
}
```

```
func:  
    #reserva espacio para v y w, 52B  
    addiu $sp,$sp,-52  
    ...  
    # @w[i] = sp+12 + i*4  
    sll $t4,$a0,2          # i*4  
    addu $t4,$t4,$sp        # sp+i*4  
    lw $t4,12($t4)          # w[i]  
    addu $t4,$t4,$t0        # w[i]+k  
  
    # @v[t4] = sp + t4  
    addu $t5,$sp,$t4        # sp + t4  
    lb $v0,0($t5)           # ret v[]  
  
    # liberar espacio pila  
    addiu $sp,$sp,52  
    jr $ra
```



Subrutinas – Implementación Subrutinas Multinivel

□ Contexto

- ¿Cómo sabemos, en una función, qué registros está usando la rutina que nos ha llamado, y por tanto no podemos tocar?
- ¿Cómo sabemos, al llamar a una función, en qué estado queda la CPU al retornar?
- ¿Cómo podemos gestionar las subrutinas para que el programador no deba preocuparse de las rutinas a las que llama?

□ ABI MIPS

- Tenemos dos tipos de registros: **TEMPORALES** y **SEGUROS**
- Las rutinas han de dejar los registros **SEGUROS** en el mismo estado que antes de la llamada.

Registros Temporales	Registros Seguros
\$t0-\$t9	\$s0-\$s7
\$v0-\$v1	\$sp
\$a0-\$a3	\$ra
\$f0-\$f19	\$f20-\$f31

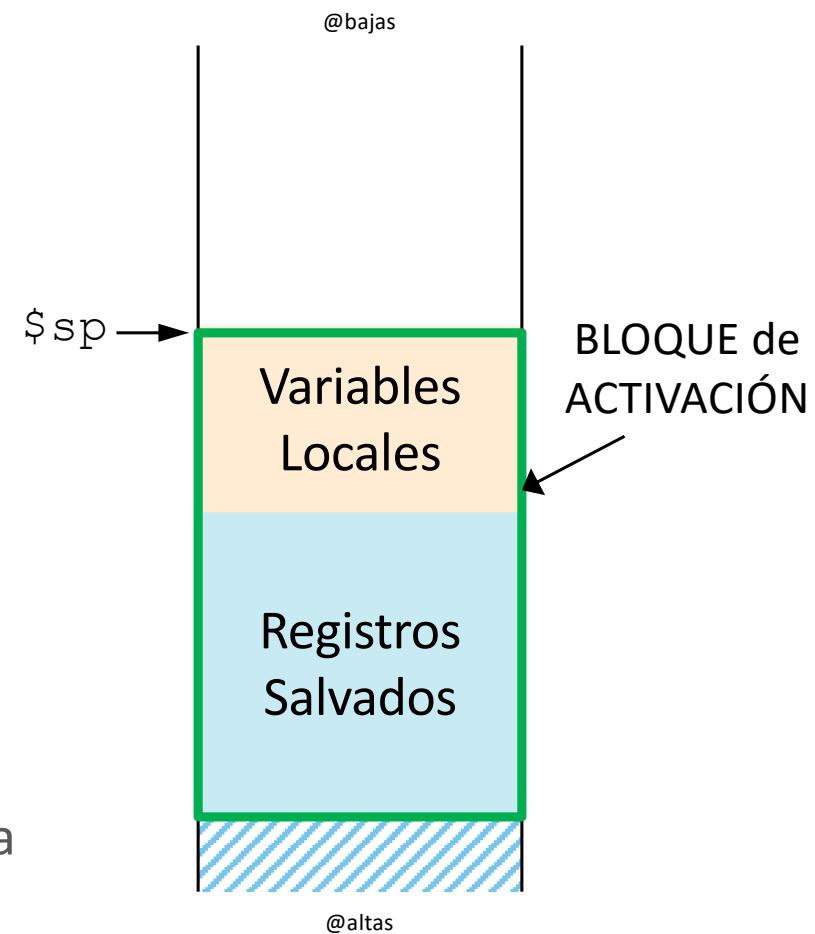
Subrutinas – Guardar y Restaurar en la PILA

Guardar y Restaurar

- ❑ Al escribir una subrutina, si utilizamos algún registro seguro, hemos de guardar una copia de él en la pila.
- ❑ En las subrutinas hemos de incluir:
 - **Prólogo.** Reservamos espacio para las variables globales y salvamos los registros seguros.
 - **Epílogo.** Restauramos los registros seguros y devolvemos el espacio de la pila.

PILA (Stack Frame)

- ❑ En el **BLOQUE de ACTIVACIÓN** tendremos:
 - **Variables Locales**, ordenadas igual que en el programa y alineadas según su tipo.
 - **Registros Salvados**, sin orden prefijado y alineados a 4 (todos los registros ocupan 4 bytes).



Subrutinas Multinivel – Ejemplo paso a paso

PASO 1. Registros Seguros

- ❑ Hay que poner en registros seguros aquellos datos que tienen un valor útil **ANTES** de una llamada y que se utilizan **DESPUÉS** de la llamada.
 - **c** es un parámetro que se utiliza después de la llamada a **mcm**.
 - **d** se calcula antes de llamar a **mcm** y se utiliza después.
 - Si no hacemos nada, dejamos **c** y **d** en registros temporales, nadie garantiza que **mcm** no modifique esos valores.
 - Pondremos **c** en **\$s0** y **d** en **\$s1**.

```
int multi(int a, int b, int c) {  
    int d, e;  
    d = a + b;  
    e = mcm(c, d);  
    return c + d + e;  
}
```

C

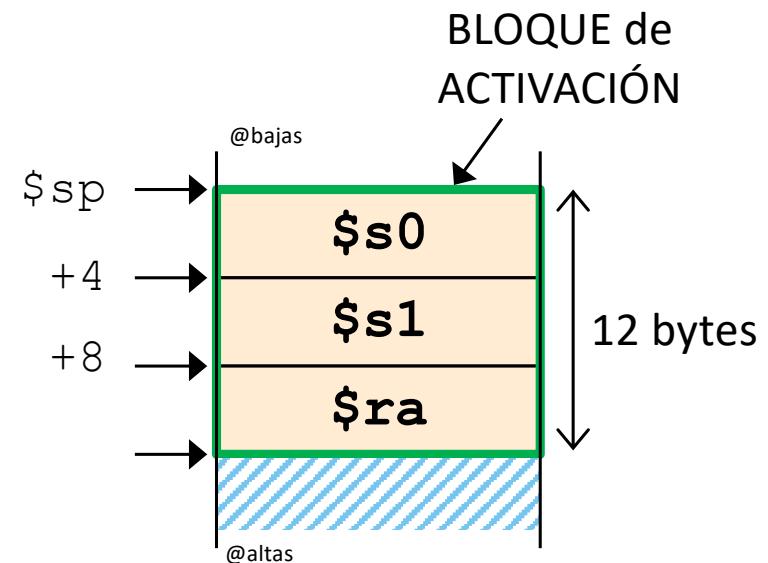
Registros Temporales	Registros Seguros
\$t0-\$t9	\$s0-\$s7
\$v0-\$v1	\$sp
\$a0-\$a3	\$ra
\$f0-\$f19	\$f20-\$f31

Subrutinas Multinivel – Ejemplo paso a paso

PASO 2. Bloque de Activación

- ❑ Determinar gráficamente la estructura del bloque de activación:
 - indicando la distancia en bytes a cada elemento desde el inicio del bloque.
 - indicando el tamaño total en bytes.
- ❑ En el ejemplo, sólo se guardan los registros salvados, y no hay variables locales en la pila
 - **\$s0** y **\$s1**
 - **\$ra**, este registro guarda la dirección de retorno y lo guardaremos **siempre** que tengamos una subrutina multinivel.

```
int multi(int a, int b, int c) {  
    int d, e;  
    d = a + b;  
    e = mcm(c, d);  
    return c + d + e;  
}
```



Subrutinas Multinivel – Ejemplo paso a paso

PASO 3. Programar la rutina

```
multi:  
    addiu $sp,$sp,-12  
    sw $s0,0($sp)  
    sw $s1,4($sp)  
    sw $ra,8($sp)  
    move $s0,$a2
```

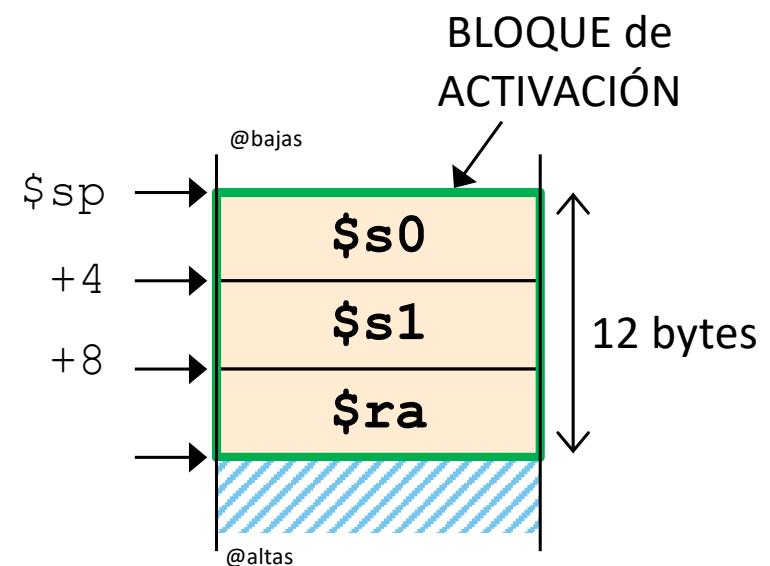
PRÓLOGO

```
    addu $s1,$a0,$a1  
    move $a0,$a2  
    move $a1,$s1  
    jal mcm  
    addu $t0,$s0,$s1  
    addu $v0,$v0,$t0
```

```
    lw $s0,0($sp)  
    lw $s1,4($sp)  
    lw $ra,8($sp)  
    addiu $sp,$sp,12  
    jr $ra
```

EPÍLOGO

```
int multi(int a, int b, int c) {  
    int d, e;  
    d = a + b;  
    e = mcm(c, d);  
    return c + d + e;  
}
```



Subrutinas Multinivel – Un Ejemplo más complejo

PASO 1. Registros Seguros

- ❑ Hay que poner en registros seguros aquellos datos que se calculan **ANTES** de una llamada y que se utilizan **DESPUÉS** de la llamada.
 - **a**, **b** y **c** vienen en **\$a0**, **\$a1** y **\$a2**
 - **d** y **e** se guardarán en registros
 - El resultado de **f()** y **g()** se guardará en registros
 - Necesitamos que **c**, **d** y el resultado de **f()** se guarden en registros seguros.
 - ✓ Guardaremos **c** en **\$s0**, **d** en **\$s1** y el resultado de **f()** en **\$s2**
 - ✓ La variable **e** estará en **\$t0**

```
int f(int m, int *n);  
int g(char *y, char *z);  
  
char exemple(int a, int b, int c) {  
    int d, e, q;  
    char v[18],w[20];  
    d = a + b;  
    e = f(d, &q) + g(v,w);  
    return v[e+q] + w[d+c];  
}
```

Variables en registros:

- a: \$a0
- b: \$a1
- c: \$a2 → \$s0
- d: \$s1
- e: \$t0
- resultado f(): \$s2

Variables en la pila:

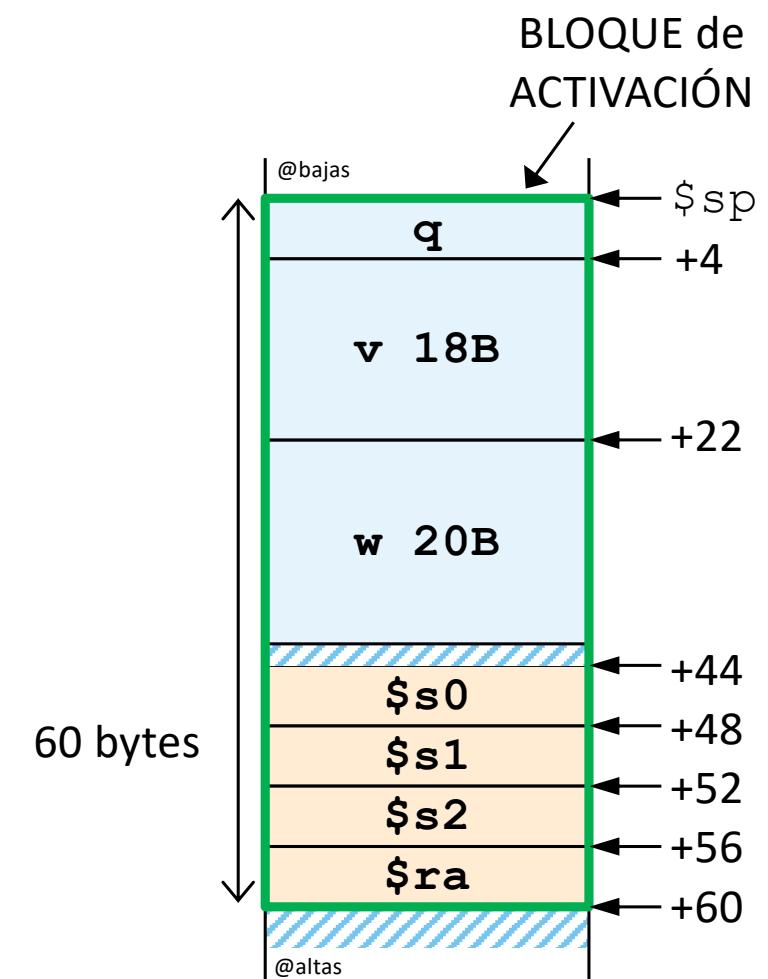
- q, v y w

Subrutinas Multinivel – Ejemplo paso a paso

PASO 2. Bloque de Activación

- ❑ Variables Locales:
 - **q, v** y **w**
- ❑ Registros que hay que salvar:
 - **\$s0, \$s1, \$s2** y **\$ra**

```
char exemple(int a, int b, int c){  
    int d, e, q;  
    char v[18],w[20];  
    d = a + b;  
    e = f(d, &q) + g(v,w);  
    return v[e+q] + w[d+c];  
}
```



Subrutinas Multinivel – Ejemplo paso a paso

PASO 3. Programar la rutina

multí:

```
addiu $sp,$sp,-60  
sw $s0,44($sp)  
sw $s1,48($sp)  
sw $s2,52($sp)  
sw $ra,56($sp)  
move $s0,$a2
```

PRÓLOGO

```
lw $s0,44($sp)  
lw $s1,48($sp)  
lw $s2,52($sp)  
lw $ra,56($sp)  
addiu $sp,$sp,60  
jr $ra
```

EPÍLOGO

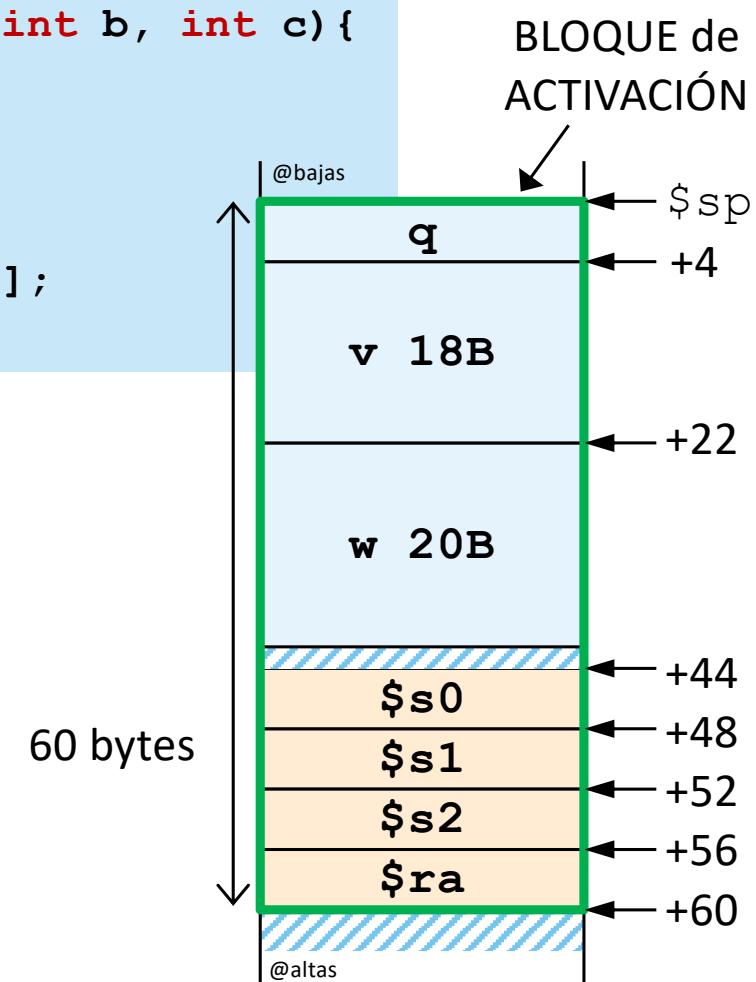
```
char exemple(int a, int b, int c) {  
    int d, e, q;  
    char v[18],w[20];  
    d = a + b;  
    e = f(d,&q)+g(v,w);  
    return v[e+q]+w[d+c];  
}
```

Variables en registros:

- a: \$a0
- b: \$a1
- c: \$a2 → \$s0
- d: \$s1
- e: \$t0
- resultado f(): \$s2

Variables en la pila:

- q, v y w



Subrutinas Multinivel – Ejemplo paso a paso

PASO 3. Programar la rutina

multi:

PRÓLOGO

```
# 1a sentencia  
addu $s1,$a0,$a1  
  
#2a sentencia  
move $a0,$s1  
move $a1,$sp  
jal f  
move $s2,$v0  
  
addiu $a0,$sp,4  
addiu $a1,$sp,22  
jal g  
addu $t0,$s2,$v0  
...
```

EPÍLOGO

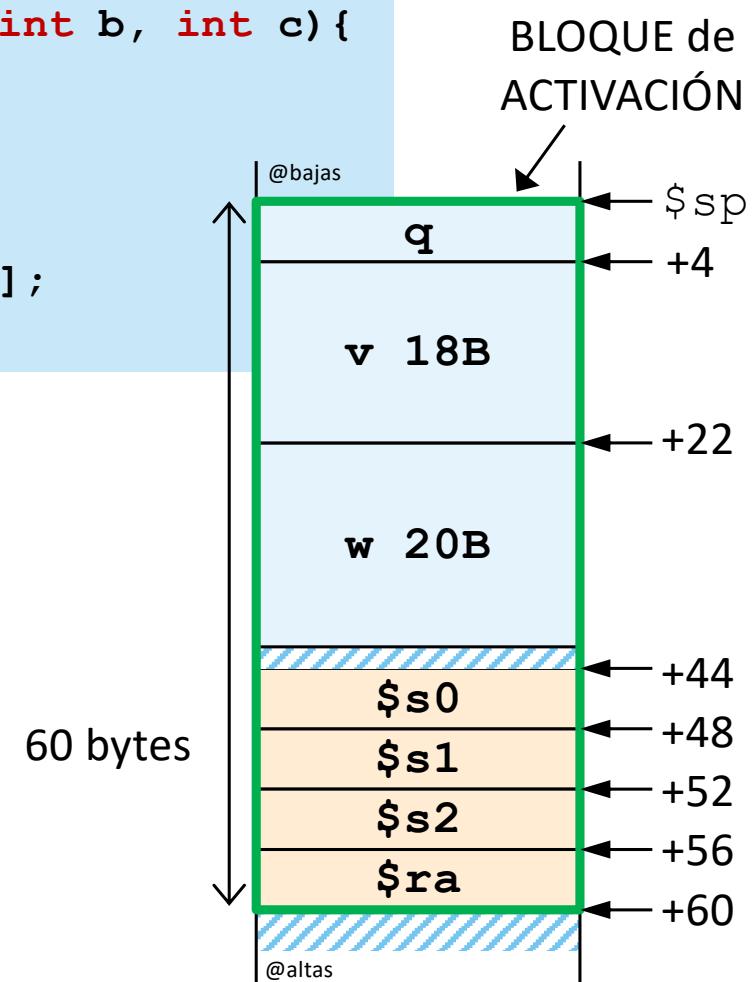
```
char exemple(int a, int b, int c) {  
    int d, e, q;  
    char v[18],w[20];  
    d = a + b;  
    e = f(d,&q)+g(v,w);  
    return v[e+q]+w[d+c];  
}
```

Variables en registros:

- a: \$a0
- b: \$a1
- c: \$a2 → \$s0
- d: \$s1
- e: \$t0
- resultado f(): \$s2

Variables en la pila:

- q, v y w



Subrutinas Multinivel – Ejemplo paso a paso

PASO 3. Programar la rutina

multi:

PRÓLOGO

```
...  
#3a sentencia  
lw $t1,0($sp)  
addu $t1,$t0,$t0  
addu $t1,$t1,$sp  
lb $t2,4($t1)  
  
addu $t1,$s0,$s1  
addu $t1,$t1,$sp  
lb $t3,22($t1)  
addu $v0,$t2,$t3
```

EPÍLOGO

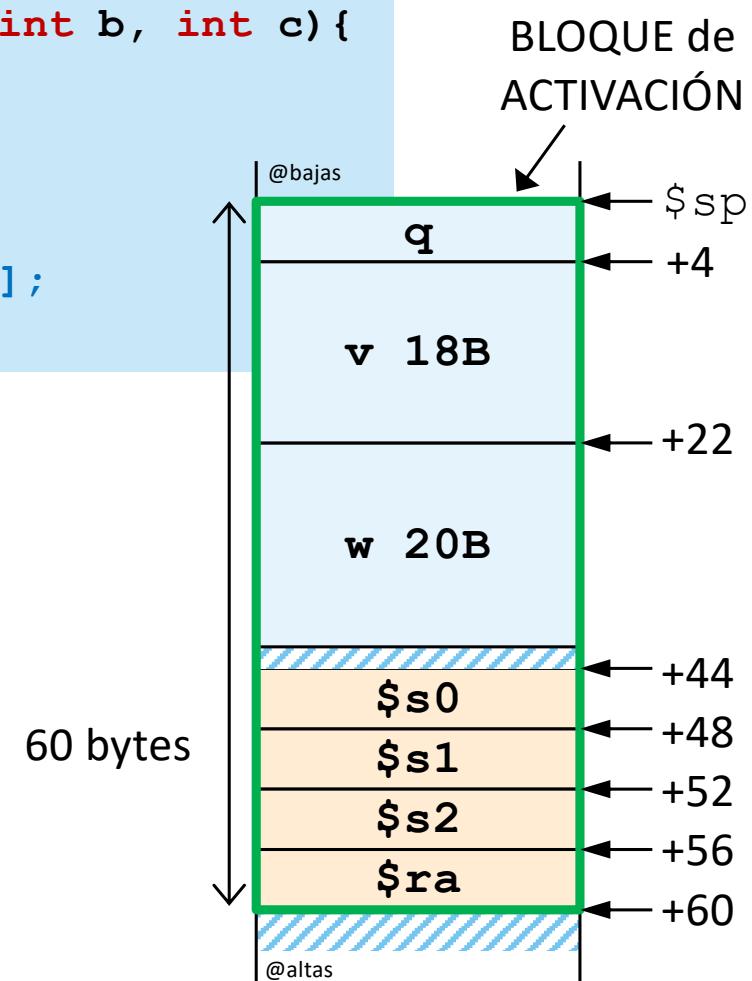
```
char exemple(int a, int b, int c) {  
    int d, e, q;  
    char v[18],w[20];  
    d = a + b;  
    e = f(d,&q)+g(v,w);  
    return v[e+q]+w[d+c];  
}
```

Variables en registros:

- a: \$a0
- b: \$a1
- c: \$a2 → \$s0
- d: \$s1
- e: \$t0
- resultado f(): \$s2

Variables en la pila:

- q, v y w



Subrutinas Repaso

Invocar a una subrutina

1. Paso de parámetros
 - Los parámetros se pasan de izquierda a derecha en los registros **\$a0-\$a3**, por orden.
 - Tendremos como máximo 4 parámetros
2. Llamada a la subrutina
 - **jal SubName**
3. Recoger/Usar resultado
 - El resultado siempre vendrá en **\$v0**

Antes de escribir la subrutina

1. Evaluar Registros Seguros
2. Dibujar Bloque de Activación

Escribir la subrutina

1. Reservar espacio en la pila para el BA
 - Restar a **\$sp** el tamaño del BA
2. Salvar registros
 - Incluyendo siempre **\$ra** [multinivel]
3. Mover parámetros a registros seguros
4. CUERPO de la SUBRUTINA
 - El resultado siempre se deja en **\$v0**
5. Restaurar registros
6. Eliminar espacio variables locales
 - Sumar a **\$sp** el tamaño del BA
7. Retorno de subrutina
 - **jr \$ra**

Subrutinas Repaso

Antes de escribir la subrutina

1. Evaluar Registros Seguros
2. Dibujar Bloque de Activación
 - En la parte alta estarán las variables locales que no puedan ir en registros.
 - En la parte baja guardaremos **\$ra** [multinivel] y los registros seguros que utilicemos.
 - Para cada elemento indicaremos la distancia desde el inicio del BA.
 - Indicaremos el tamaño total del BA.
 - ✓ Siempre ha de ser múltiplo de 4.

Variables Locales

- Siempre que podamos las variables locales se guardarán en registros.
 - **\$t0-\$t9**
- Se almacena en la pila:
 - matrices vectores, tuplas, ...
 - Las variables locales con **&** .
 - variables para las que no hay registros suficientes.
- Los datos que van a la pila se almacenan en orden, alineados siguiendo los mismos criterios que las variables globales.
- El bloque de activación siempre ha de estar alineado a 4 bytes.

Ejercicio 3.29

3.29 Donades les següents accions en C:

```
void swap(int *a, int *b) {  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}  
void examen(int v[], char c, int j){  
    swap(&v[j+1], &v[j]);  
}
```

- a) Tradueix a assemblador MIPS la subrutina **swap**.
- b) Tradueix a assemblador MIPS la subrutina **examen**.

- ❑ En **swap**, sólo tenemos la variable local **temp**, la pondremos en **\$t0**. Ni siquiera hay que salvar **\$ra**.
- ❑ Los parámetros vendrán en **\$a0** y **\$a1**.

- ❑ En **examen** no hay variables locales, sólo hay que salvar **\$ra** en la pila.,
- ❑ Los parámetros vendrán en **\$a0**, **\$a1** y **\$a2**.

Ejercicio 3.29

```
void swap(int *a, int *b) {  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
void examen(int v[], char c, int j){  
    swap(&v[j+1], &v[j]);  
}
```

```
swap:  
    lw $t0,0($a0)  
    lw $t1,0($a1)  
    sw $t1,0($a0)  
    sw $t0,0($a1)  
    jr $ra
```

```
examen:  
    addiu $sp,$sp,-4  
    sw $ra,0($sp)  
  
    sll $t1,$a2,2  
    addu $a1,$a0,$t1  
    addiu $a0,$a1,4  
    jal swap  
  
    lw $ra,0($sp)  
    addiu $sp,$sp,-4  
    jr $ra
```

Ejercicio 3.28

3.28 Donades les següents declaracions, tradueix a assemblador MIPS la funció **s1**:

```
int s2(int c, long long *d, long long w[]);
int s1(int x, long long v[], int *p) {
    long long *k;
    k = &v[x];
    x = s2(*p, k, v);
    return *p + x;
}
```

- ❑ Los parámetros, en las 2 subrutinas, van en **\$a0**, **\$a1** y **\$a2**
- ❑ Variable local **k**, irá en el registro **\$t0**, podría ser **\$t1**, **\$t2**, ...
- ❑ El resultado de ambas funciones se devuelve en **\$v0**
- ❑ El puntero **p** se ha de preservar en un registro seguro: **\$s0**
 - Hay que salvar **\$s0** y **\$ra** en la pila al principio de la rutina
 - Hay que restaurar **\$s0** y **\$ra** de la pila al final de la rutina

Ejercicio 3.28

```
s1:  
addiu $sp,$sp,-8  
sw $s0,0($sp)  
sw $ra,4($sp)  
move $s0,$a2
```

PRÓLOGO

```
# k = &v[x]  
sll $t1,$a0,3  
add $t0,$t1,$a1  
...
```

```
lw $s0,0($sp)  
lw $ra,4($sp)  
addiu $sp,$sp,8  
jr $ra
```

EPÍLOGO

```
int s2 (int c, long long *d, long long w[]);  
int s1 (int x, long long v[], int *p) {  
    long long *k;  
    k = &v[x];  
    x = s2(*p, k, v);  
    return *p+x;  
}
```

Variables en registros:

- x: \$a0
- v: \$a1
- p: \$a2 → \$s0
- k: \$t0
- resultado: \$v0

BLOQUE de
ACTIVACIÓN



Ejercicio 3.28

```
s1:  
addiu $sp,$sp,-8  
sw $s0,0($sp)  
sw $ra,4($sp)  
move $s0,$a2
```

PRÓLOGO

```
...  
# x = s2(*p, k, v);  
lw $a0,0($s0)  
move $a2,$a1  
move $a1,$t0  
jal s2  
move $a0,$v0  
...
```

```
lw $s0,0($sp)  
lw $ra,4($sp)  
addiu $sp,$sp,8  
jr $ra
```

EPÍLOGO

```
int s2 (int c, long long *d, long long w[]);  
int s1 (int x, long long v[], int *p) {  
    long long *k;  
    k = &v[x];  
    x = s2(*p, k, v);  
    return *p+x;  
}
```

Variables en registros:

- x: \$a0
- v: \$a1
- p: \$a2 → \$s0
- k: \$t0
- resultado: \$v0

BLOQUE de
ACTIVACIÓN



Ejercicio 3.28

```
s1:  
addiu $sp,$sp,-8  
sw $s0,0($sp)  
sw $ra,4($sp)  
move $s0,$a2
```

PRÓLOGO

```
...  
# return *p+x;  
lw $t1,0($s0)  
addu $v0,$t1,$a0
```

```
lw $s0,0($sp)  
lw $ra,4($sp)  
addiu $sp,$sp,8  
jr $ra
```

EPÍLOGO

```
int s2 (int c, long long *d, long long w[]);  
int s1 (int x, long long v[], int *p) {  
    long long *k;  
    k = &v[x];  
    x = s2(*p, k, v);  
    return *p+x;  
}
```

Variables en registros:

- x: \$a0
- v: \$a1
- p: \$a2 → \$s0
- k: \$t0
- resultado: \$v0

BLOQUE de
ACTIVACIÓN



Ejercicio, pregunta 3 examen final enero 2019

Donades les següents declaracions en C:

```
int *pglob;                      // variable global
int f2(int x, char *y, char *z); // prototipus de f2
char f1(int a, char b[][][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```

A continuació es mostra una traducció de la funció **f1** a llenguatge MIPS que està incompleta. Llegiu-la amb atenció, i completeu les caixes per tal que la traducció sigui correcta.

- Vamos a traducir la rutina completa

```
.data
pglob: .word 0
```

Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;
int f2(int x, char *y, char *z);
char f1(int a, char b[][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```

La variable c se utiliza después de llamar a f2, hay que guardarla en un registro seguro.

Pondremos c en \$s0.

1. Registros Seguros
2. Bloque de Activación
3. Programar la subrutina

Bloque de activación

Variables Locales

- v, ocupará 10 + 2 bytes

Registros salvados

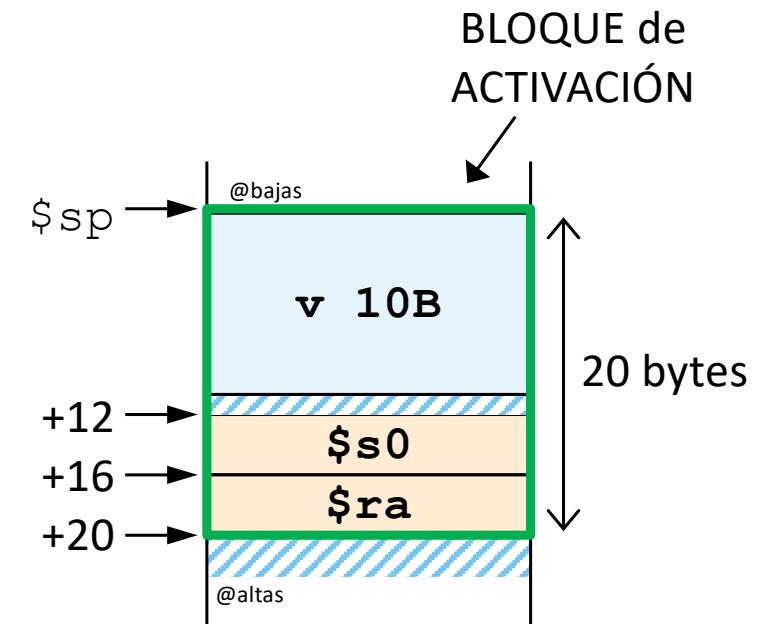
- \$s0 y \$ra

El Bloque de activación ocupará 20 bytes



Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;
int f2(int x, char *y, char *z);
char f1(int a, char b[][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```



1. Registros Seguros
2. Bloque de Activación
3. Programar la subrutina

Bloque de activación
Variables Locales

- v, ocupará 10 + 2 bytes

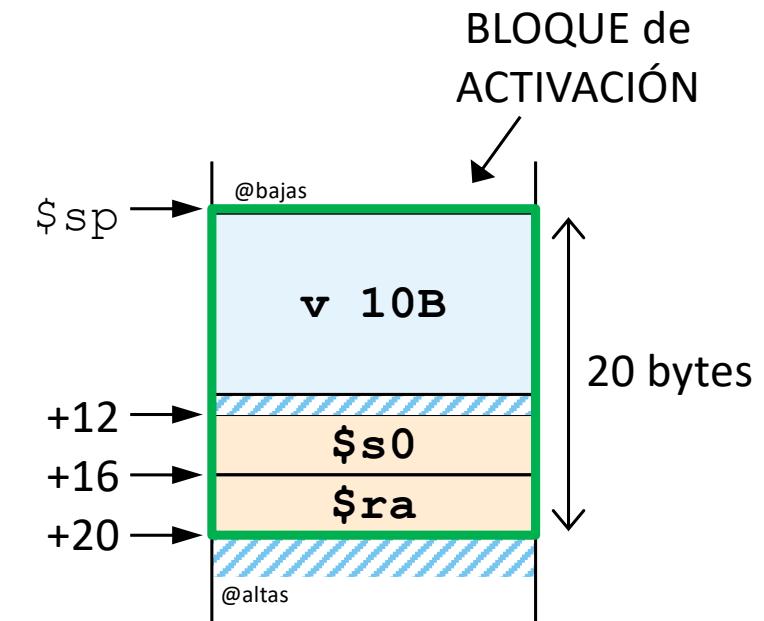
Registros salvados

- \$s0 y \$ra

El Bloque de activación ocupará 20 bytes

Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;  
int f2(int x, char *y, char *z);  
char f1(int a, char b[][5], int c, int *d) {  
    char v[10];  
    if ((c < 0) || (c >= a))  
        c = 0;  
    *pglob = f2(*d, &b[3][0], v);  
    return v[c];  
}
```



```
f1:  
    addiu $sp,$sp,-20  
    sw $s0,12($sp)  
    sw $ra,16($sp)  
    move $s0,$a2  
    ...
```

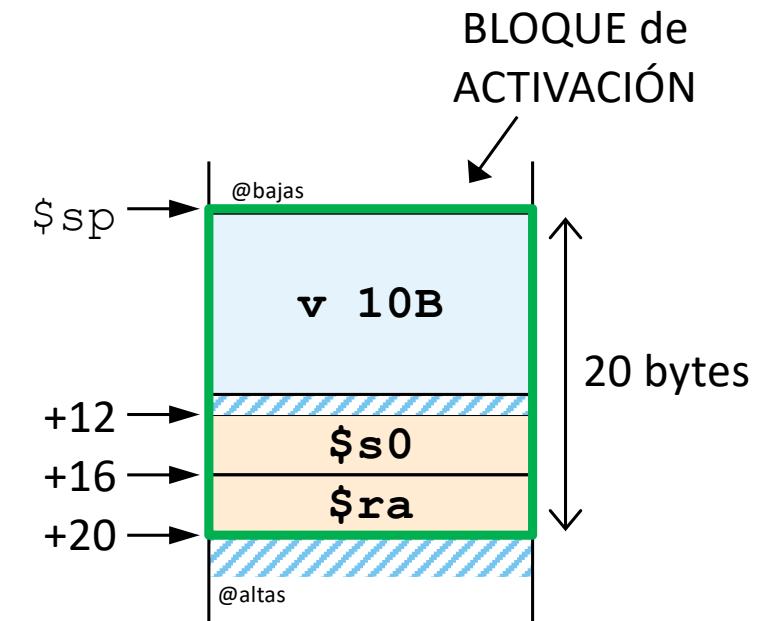
PRÓLOGO

```
...  
    lw $s0,12($sp)  
    lw $ra,16($sp)  
    addiu $sp,$sp,20  
    jr $ra
```

EPÍLOGO

Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;
int f2(int x, char *y, char *z);
char f1(int a, char b[][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```

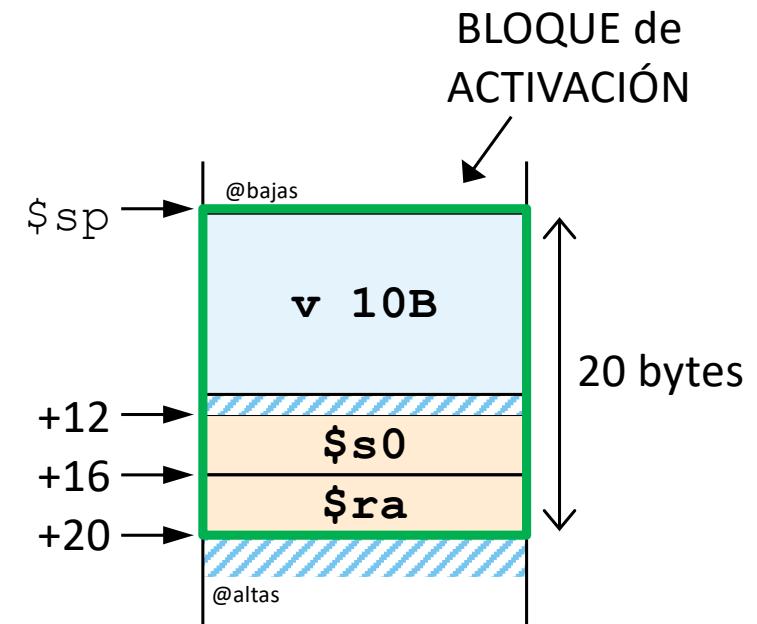


```
        blt $s0,$zero,then
        blt $s0,$a0,endif
then:
        move $s0, zero
endif:
```

Ejercicio, pregunta 3 examen final enero 2019

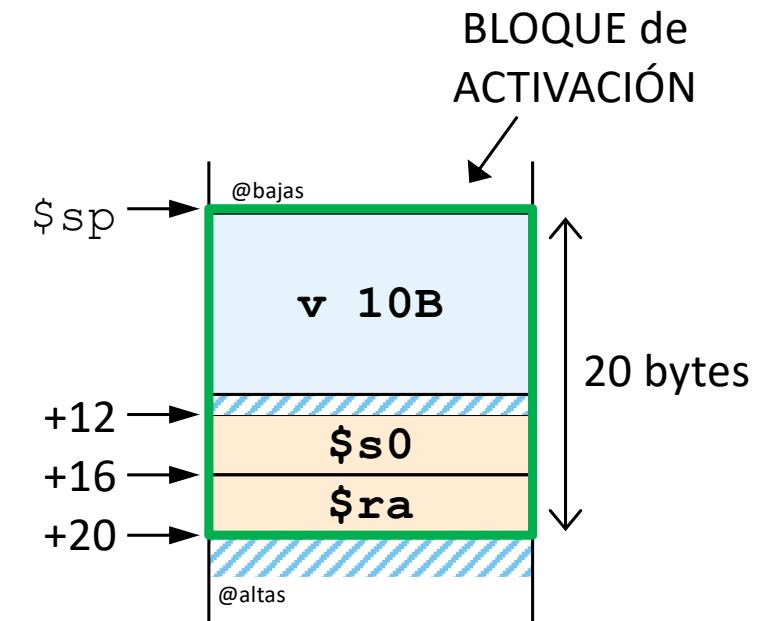
```
int *pglob;  
int f2(int x, char *y, char *z);  
char f1(int a, char b[][5], int c, int *d) {  
    char v[10];  
    if ((c < 0) || (c >= a))  
        c = 0;  
    *pglob = f2(*d, &b[3][0], v);  
    return v[c];  
}
```

```
lw $a0,0($a3)  
addiu $a1,$a1,15  
move $a2,$sp  
jal f2  
la $t0,pglob  
lw $t1,0($t0)  
sw $v0,0($t1)
```



Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;
int f2(int x, char *y, char *z);
char f1(int a, char b[][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```

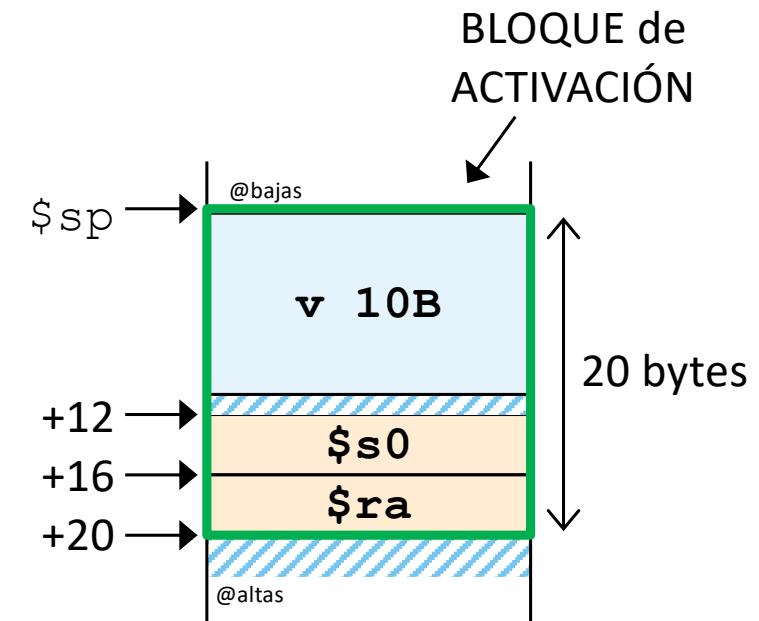


```
addu $t0,$sp,$s0
lb $v0,0($t0)
```

Ejercicio, pregunta 3 examen final enero 2019

```
int *pglob;
int f2(int x, char *y, char *z);
char f1(int a, char b[][5], int c, int *d) {
    char v[10];
    if ((c < 0) || (c >= a))
        c = 0;
    *pglob = f2(*d, &b[3][0], v);
    return v[c];
}
```

```
addu $t0,$sp,$s0
lb $v0,0($t0)
```





UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 3: Traducción de Programas

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

