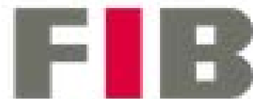
Tema 3. Traducció de programes

Joan Manuel Parcerisa

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



Traducció de programes

- Desplaçaments i operacions lògiques bit a bit
- Comparacions, operacions booleanes i salts
- Sentències alternatives i iteratives
- Subrutines
- Estructura de la memòria
- Compilació, assemblatge, enllaçat i càrrega

Desplaçaments de bits

Desplaçaments lògics de bits

- Shift Left Logical (`sll`) i Shift Right Logical (`srl`)

```
sll     rd, rt, shamt
srl     rd, rt, shamt
```

  - Desplacen `rt` a l'**esquerra** (`sll`) o a la **dreta** (`srl`) el número de bits indicat a l'operand immediat `shamt` ("shift amount")
  - Les posicions vacants s'omplen amb **zeros**

- Exemple

```
li      $t0, 0x88888888
sll     $t1, $t0, 2            # $t1 = 0x22222220
srl     $t1, $t0, 1            # $t1 = 0x44444444
```

Desplaçament aritmètic de bits

- Shift Right Arithmetic (sra)

```
sra     rd, rt, shamt
```

  - Desplaça `rt` a **dreta** `shamt` bits, i omple les posicions vacants amb una còpia del **bit de signe** de `rt`

- Exemple

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

Desplaçament de bits variable

- Desplaçaments indicats en registre

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

  - El nombre de posicions a desplaçar s'indica en els 5 bits de menor pes de `rs` (la resta s'ignoren)

Desplaçament de bits en C

- A l'esquerra: operador `<<`

  Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Desplaçament de bits en C

- A l'esquerra: operador `<<`

Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Traducció a MIPS

```
# Suposem a,b,c guardats en $t0,$t1,$t2

sll     $t0, $t0, 3
sllv    $t2, $t2, $t1
```

Desplaçament de bits en C

- A l'esquerra: operador `<<`

Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Traducció a MIPS

```
# Suposem a,b,c guardats en $t0,$t1,$t2

sll     $t0, $t0, 3
sllv    $t2, $t2, $t1
```

- A la dreta: operador `>>`
  - Si s'aplica a un enter:

Exemple en C

```
int a, b;

...
a = a >> 3;
a = a >> b;
```

Desplaçament de bits en C

- A l'esquerra: operador `<<`

Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Traducció a MIPS

```
# Suposem a,b,c guardats en $t0,$t1,$t2

sll     $t0, $t0, 3
sllv    $t2, $t2, $t1
```

- A la dreta: operador `>>`
  - Si s'aplica a un enter: desplaçament aritmètic

Exemple en C

```
int a, b;

...
a = a >> 3;
a = a >> b;
```

Traducció a MIPS

```
# Suposem a,b guardats en $t0,$t1

sra     $t0, $t0, 3
srav    $t0, $t0, $t1
```

Desplaçament de bits en C

- A l'esquerra: operador `<<`

Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Traducció a MIPS

```
# Suposem a,b,c guardats en $t0,$t1,$t2

sll     $t0, $t0, 3
sllv    $t2, $t2, $t1
```

- A la dreta: operador `>>`
  - Si s'aplica a un enter: desplaçament aritmètic
  - Si s'aplica a un natural (*unsigned*):

Exemple en C

```
int a, b;
unsigned int c;
...
a = a >> 3;
a = a >> b;
c = c >> b;
```

Traducció a MIPS

```
# Suposem a,b guardats en $t0,$t1


sra     $t0, $t0, 3
srav    $t0, $t0, $t1
```

Desplaçament de bits en C

- A l'esquerra: operador `<<`

Exemple en C

```
int a, b, c;
...
a = a << 3;
c = c << b;
```

Traducció a MIPS

```
# Suposem a,b,c guardats en $t0,$t1,$t2

sll     $t0, $t0, 3
sllv    $t2, $t2, $t1
```

- A la dreta: operador `>>`
  - Si s'aplica a un enter: desplaçament aritmètic
  - Si s'aplica a un natural (*unsigned*): desplaçament lògic

Exemple en C

```
int a, b;
unsigned int c;
...
a = a >> 3;
a = a >> b;
c = c >> b;
```

Traducció a MIPS

```
# Suposem a,b guardats en $t0,$t1
# Suposem c guardat en $t2

sra     $t0, $t0, 3
srav    $t0, $t0, $t1
srlv    $t2, $t2, $t1
```

Multiplicació i divisió per potències de 2

- Desplaçar `rt` a la dreta equival a dividir-lo: `rt` / 2^{shamt}

```
srl rd, rt, shamt      # si rt és un natural
sra rd, rt, shamt      # si rt és un enter
```

- Desplaçar `rt` a esquerra equival a multiplicar-lo: `rt` × 2^{shamt}

```
sll rd, rt, shamt
```

Multiplicació i divisió per potències de 2

- Desplaçar `rt` a la dreta equival a dividir-lo: `rt` / 2^{shamt}

```
srl rd, rt, shamt       # si rt és un natural
sra rd, rt, shamt       # si rt és un enter
```

- Desplaçar `rt` a esquerra equival a multiplicar-lo: `rt` × 2^{shamt}

```
sll rd, rt, shamt
```

  - S'usa sovint per calcular l'offset d'un vector, donat un índex:

Exemple en C

```c
int vec[100];

void main() {
    int i; // en $t0
    ...
    vec[i] = 0;
}
```

Multiplicació i divisió per potències de 2

- Desplaçar `rt` a la dreta equival a dividir-lo: `rt` / 2^{shamt}

```
srl rd, rt, shamt       # si rt és un natural
sra rd, rt, shamt       # si rt és un enter
```

- Desplaçar `rt` a esquerra equival a multiplicar-lo: `rt` × 2^{shamt}

```
sll rd, rt, shamt
```

  - S'usa sovint per calcular l'offset d'un vector, donat un índex:

Exemple en C

```c
int vec[100];

void main() {
    int i; // en $t0
    ...
    vec[i] = 0;
}
```

Traducció a MIPS

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

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

Divisió per potències de 2

- Divisió d'enters, en MIPS
  - En general: instrucció `div` (es veurà al Tema 5)
  - Dividir per potències de 2: instrucció `sra`
    - Molt més ràpida que div
  - Les dues satisfan: $D = d \times q + r$

Divisió per potències de 2

- Divisió d'enters, en MIPS
  - En general: instrucció `div` (es veurà al Tema 5)
  - Dividir per potències de 2: instrucció `sra`
    - Molt més ràpida que div
  - Les dues satisfan: D = d × q + r
- Però alerta!
  - Si D < 0 i la divisió no és exacta

    `sra` dóna diferent resultat que `div`

  - Exemple: D = –15, d = 4
    - La divisió amb `div` (`-15/4`) dóna q = – 3, r = – 3
    - La divisió amb `sra` (`-15>>2`) dóna q = – 4, r = 1

Divisió per potències de 2

- Divisió d'enters, en MIPS
  - En general: instrucció `div` (es veurà al Tema 5)
  - Dividir per potències de 2: instrucció `sra`
    - Molt més ràpida que div
  - Les dues satisfan: $D = d \times q + r$
- Però alerta!
  - Si $D < 0$ i la divisió no és exacta

    `sra` dóna diferent resultat que `div`

  - Exemple: $D = -15$, $d = 4$
    - La divisió amb `div` (`-15/4`) dóna $q = -3$, $r = -3$
    - La divisió amb `sra` (`-15>>2`) dóna $q = -4$, $r = 1$
  - Definició de l'operador / en C, i de la instrucció `div`: Sempre retorna un residu (r) del mateix signe que el dividend (D)

➡ **Si el dividend és negatiu, cal traduir l'operador / amb la instrucció `div`**

Operacions lògiques bit a bit

Operacions lògiques bit a bit

- Repertori d'instruccions MIPS

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

  - Les instruccions `andi`, `ori` i `xori` extenen l'operand immediat de 16 a 32 bits afegint zeros a l'esquerra (interpretant-lo com un natural)

Operacions lògiques bit a bit

- Operadors lògics bit a bit en C, i traducció a MIPS
  - Suposant que `a`, `b` i `c` estan en \$t0, \$t1 i \$t2:

En C	En 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`

Operacions lògiques bit a bit

- Operadors lògics bit a bit en C, i traducció a MIPS
  - Suposant que `a`, `b` i `c` estan en \$t0, \$t1 i \$t2:

En C	En 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**. Traduir, suposant que `a` i `b` estan en \$t0 i \$t1 (l'operador ~ significa "negació bit a bit").

```
a = ~(a & b);
```

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

Operacions lògiques bit a bit

- Utilitat de and i andi: posar bits a 0
  - Per seleccionar bits determinats d'un registre, posant la resta a zero

    Exemple: seleccionar bits en posició parell de $t0: 0, 2, 4, etc.

```
li      $t2, 0x55555555   # màscara de selecció
and     $t1, $t0, $t2
```

Operacions lògiques bit a bit

- Utilitat de and i andi: posar bits a 0
  - Per seleccionar bits determinats d'un registre, posant la resta a zero

    Exemple: seleccionar bits en posició parell de $t0: 0, 2, 4, etc.

    ```
    li      $t2, 0x55555555    # màscara de selecció
    and     $t1, $t0, $t2
    ```

    Exemple: seleccionar els 16 bits de menor pes de $t0

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

Operacions lògiques bit a bit

- Utilitat de and i andi: posar bits a 0
  - Per seleccionar bits determinats d'un registre, posant la resta a zero

    Exemple: seleccionar bits en posició parell de $t0: 0, 2, 4, etc.

    ```
    li     $t2, 0x55555555    # màscara de selecció
    and    $t1, $t0, $t2
    ```

    Exemple: seleccionar els 16 bits de menor pes de $t0

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

  - Per calcular el residu de dividir per potències de 2

    Exemple:

    ```
    andi   $t1, $t0, 7        # $t1 = $t0 mod 8
    ```

Operacions lògiques bit a bit

- Utilitat de or i ori
  - Per posar determinats bits a 1

    Exemple: posar a 1 els 16 bits de menor pes de $t0

```
ori   $t1, $t0, 0xFFFF
```

Operacions lògiques bit a bit

- Utilitat de or i ori
  - Per posar determinats bits a 1

    Exemple: posar a 1 els 16 bits de menor pes de $t0

```
ori    $t1, $t0, 0xFFFF
```

- Utilitat xor i xori
  - Complementar determinats bits

    Exemple: complementar els bits parells de $t0

```
li     $t2, 0x55555555   # màscara de selecció
xor    $t1, $t0, $t2
```

Comparacions operacions booleanes

Comparacions i operacions booleanes en C

- Comparacions naturals/enters: ==, !=, <, >, <=, >=
  - Usen operands naturals/enters
  - El resultat és un enter *normalitzat*: 0 si FALS, 1 si CERT
    - En C, NO existeix el tipus booleà!

Comparacions i operacions booleanes en C

- Comparacions naturals/enters: ==, !=, <, >, <=, >=
  - Usen operands naturals/enters
  - El resultat és un enter *normalitzat*: 0 si FALS, 1 si CERT
    - En C, NO existeix el tipus booleà!

- Operacions booleanes: &&, ||, !
  - Usen operands *enters*: 0 es considera FALS, altrament CERT
  - El resultat és un enter *normalitzat*: 0 si FALS, 1 si CERT

Repertori de comparacions en MIPS

- Només implementa la comparació < ("menor que")

slt rd, rs, rt	rd = rs < rt	enters
sltu rd, rs, rt	rd = rs < rt	naturals
slti rd, rs, imm16	rd = rs < sext(imm16)	enters
sltiu rd, rs, imm16	rd = rs < sext(imm16)	naturals

Repertori de comparacions en MIPS

- Només implementa la comparació < ("menor que")

`slt   rd, rs, rt`	`rd = rs < rt`	`enters`
`sltu  rd, rs, rt`	`rd = rs < rt`	`naturals`
`slti  rd, rs, imm16`	`rd = rs < sext(imm16)`	`enters`
`sltiu rd, rs, imm16`	`rd = rs < sext(imm16)`	`naturals`

- Generen un resultat "normalitzat": 0 si fals, 1 si cert
- Alerta!: `sltiu` compara *naturals* però aplica *extensió de signe* a l'immediat

Exemple. Comparació <

- Suposem que a, b, c es guarden en $t0, $t1, $t2
- Traduir a MIPS

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

Exemple. Comparació <

- Suposem que a, b, c es guarden en $t0, $t1, $t2
- Traduir a MIPS

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

- Si a, b, c són enters (int)

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

Exemple. Comparació <

- Suposem que a, b, c es guarden en $t0, $t1, $t2
- Traduir a MIPS

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

- Si a, b, c són enters (int)

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

- Si a, b, c són naturals (unsigned int)

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

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;
```

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;        →     sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;          →      sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;
```

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;          →      sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;          →      sltu  rc,$zero,ra
```

(amb naturals, "diferent de 0" equival a "major que 0")

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;          →      sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;          →      sltu  rc,$zero,ra
```

(amb naturals, "diferent de 0" equival a "major que 0")

- Comparacions entre ra i rb

```
rc = ra==rb;
```

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;        →    sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;        →    sltu  rc,$zero,ra
```

(amb naturals, "diferent de 0" equival a "major que 0")

- Comparacions entre ra i rb

`rc = ra==rb;` equival a: `rc = (ra-rb)==0`

```
                   →    sub   rc,ra,rb
                        sltiu rc,rc,1
```

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;          →      sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;          →      sltu  rc,$zero,ra
```

(amb naturals, "diferent de 0" equival a "major que 0")

- Comparacions entre ra i rb

`rc = ra==rb;` equival a: `rc = (ra-rb)==0`

```
                     →      sub   rc,ra,rb
                            sltiu rc,rc,1
```

```
rc = ra!=rb;
```

Comparacions "==" i "!="

- Suposem ra, rb, rc enters guardats en registres
- Comparació amb zero

```
rc = ra==0;          →      sltiu rc,ra,1
```

(amb naturals, "igual a 0" equival a "menor que 1")

```
rc = ra!=0;          →      sltu  rc,$zero,ra
```

(amb naturals, "diferent de 0" equival a "major que 0")

- Comparacions entre ra i rb

```
rc = ra==rb;  equival a: rc = (ra-rb)==0
                     →      sub   rc,ra,rb
                            sltiu rc,rc,1
rc = ra!=rb;  equival a: rc = (ra-rb)!=0
                     →      sub   rc,ra,rb
                            sltu  rc,$zero,rc
```

Negació booleana !

- La instrucció `not` bit a bit no té resultat "normalitzat"!
- La convertim en una comparació amb zero

```
rc = !ra;
```

Negació booleana !

- La instrucció `not` bit a bit no té resultat "normalització"!
- La convertim en una comparació amb zero

`rc = !ra;` equival a: `rc = ra==0;`

→ `sltiu rc,ra,1`

Negació booleana !

- La instrucció `not` bit a bit no té resultat "normalitzat"!
- La convertim en una comparació amb zero

  `rc = !ra;` equival a: `rc = ra==0;`

  → `sltiu rc,ra,1`
- Si `ra` sabem segur que és 0 o 1, llavors també serveix

  → `xori rc,ra,1`

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;
```

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;        →      slt    rc,rb,ra
```

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;          →      slt   rc,rb,ra
```

- Menor o igual que

```
rc = ra<=rb;
```

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;          →      slt   rc,rb,ra
```

- Menor o igual que

`rc = ra<=rb;` equival a: `rc = !(ra>rb);`

```
                     →      slt   rc,rb,ra
                            sltiu rc,rc,1
```

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;          →     slt   rc,rb,ra
```

- Menor o igual que

```
rc = ra<=rb; equival a: rc = !(ra>rb);
                     →     slt   rc,rb,ra
                           sltiu rc,rc,1
```

- Major o igual que

```
rc = ra>=rb;
```

Comparacions ">", "<=", ">="

- Suposem ra, rb, rc enters guardats en registres
- Major que

```
rc = ra>rb;          →      slt   rc,rb,ra
```

- Menor o igual que

`rc = ra<=rb;` equival a: `rc = !(ra>rb);`

```
                     →      slt   rc,rb,ra
                            sltiu rc,rc,1
```

- Major o igual que

`rc = ra>=rb;` equival a: `rc = !(ra<rb);`

```
                     →      slt   rc,ra,rb
                            sltiu rc,rc,1
```

Salts

- Salts condicionals relatius al PC
- Salts incondicionals

Salts condicionals relatius al PC

- MIPS sols inclou beq i bne:

beq	rs, rt, etiqueta
bne	rs, rt, etiqueta
b	etiqueta

Salts condicionals relatius al PC

- MIPS sols inclou beq i bne:

`beq rs, rt, etiqueta`	if (rs==rt), PC = PC_{up} + Sext(imm16*4)
`bne rs, rt, etiqueta`	if (rs!=rt), PC = PC_{up} + Sext(imm16*4)
`b etiqueta`	

- L'etiqueta és un immediat enter de 16 bits que codifica la distància a saltar
  - Respecte de PC_{up} (PC_{up} = PC+4)
  - En número d'instruccions (paraules de 4 bytes)
  - Permet saltar en el rang $[-2^{15}, +2^{15}-1]$ instruccions

Salts condicionals relatius al PC

- MIPS sols inclou beq i bne:

`beq rs, rt, etiqueta`	if (rs==rt), PC = PC_{up} + Sext(imm16*4)
`bne rs, rt, etiqueta`	if (rs!=rt), PC = PC_{up} + Sext(imm16*4)
`b etiqueta`	`beq $0, $0, etiqueta # és una macro`

- L'etiqueta és un immediat enter de 16 bits que codifica la distància a saltar
  - Respecte de PC_{up} (PC_{up} = PC+4)
  - En número d'instruccions (paraules de 4 bytes)
  - Permet saltar en el rang [-2^{15}, $+2^{15}-1$] instruccions

Repertori de macros convenients

- Macros de salts (comparacions d'enters)

`blt rs,rt,etiq`	`if (rs<rt) salta a etiq`	
`bgt rs,rt,etiq`	`if (rs>rt) salta a etiq`	
`ble rs,rt,etiq`	`if (rs<=rt) salta a etiq`	
`bge rs,rt,etiq`	`if (rs>=rt) salta a etiq`	

- Per a naturals, usarem les macros anàlogues
  `bltu, bgtu, bleu, bgeu`

Repertori de macros convenients

- Macros de salts (comparacions d'enters)

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

- Per a naturals, usarem les macros anàlogues
  `bltu, bgtu, bleu, bgeu`
- S'expandeixen igual, però usant `sltu`

Salts incondicionals (absoluts)

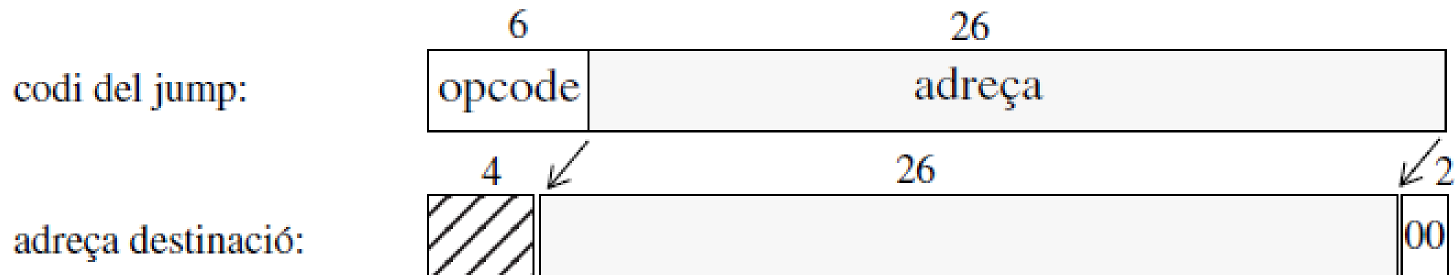
- Els salts **relatius** al PC (inclosa la mcro b) tenen un rang limitat [-2^{15}, $+2^{15}-1$]
- Per a salts majors, usarem salts **absoluts**

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

- Les instruccions jal i jalr s'usen per a cridar subrutines

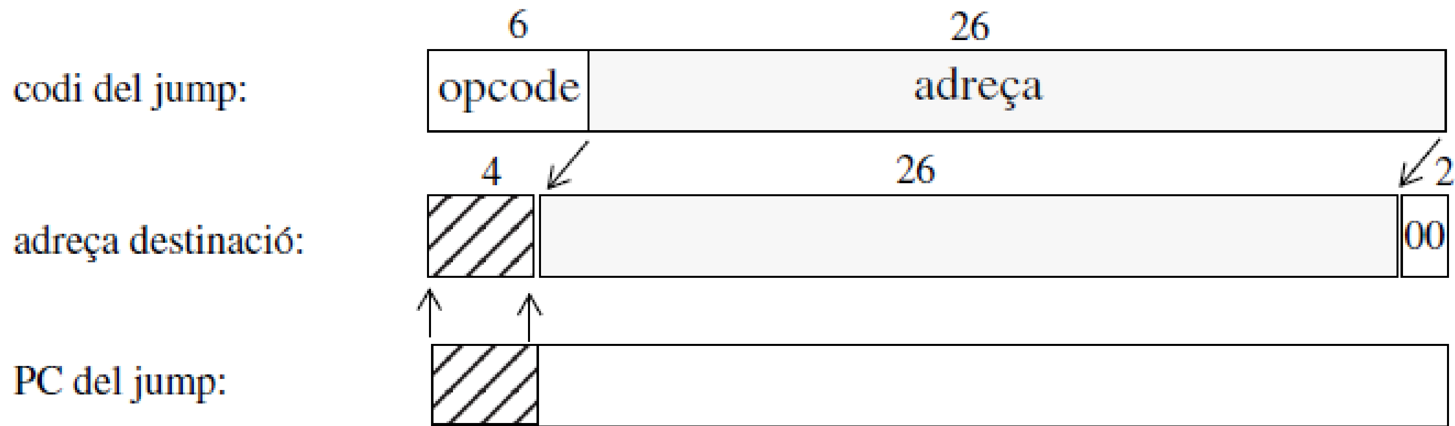
Salts incondicionals en mode pseudodirecte

- Les instruccions *j* i *jal* es codifiquen en el format J
- Adreça destinació en els 26 bits de menor pes



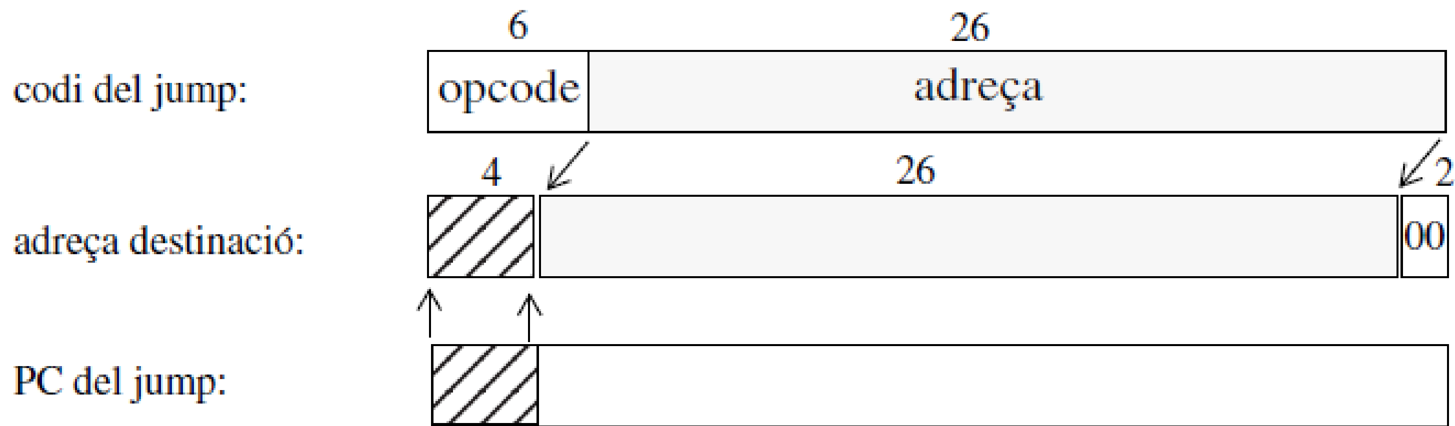
Salts incondicionals en mode pseudodirecte

- Les instruccions *j* i *jal* es codifiquen en el format J
- Adreça destinació en els 26 bits de menor pes



Salts incondicionals en mode pseudodirecte

- Les instruccions *j* i *jal* es codifiquen en el format J
- Adreça destinació en els 26 bits de menor pes



- Rang de salts: dins el bloc de 2^{28} bytes del PC actual
- Per saltar més lluny (qualsevol adreça)

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

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - Mètode per especificar la ubicació d'un operand

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - mètode per especificar la ubicació d'un operand

- Mode registre

```
addu  $t0,$t0,$t1
```

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - mètode per especificar la ubicació d'un operand

- Mode registre

```
addu  $t0,$t0,$t1
```

- Mode immediat

```
addiu $t0,$t0,4
```

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - mètode per especificar la ubicació d'un operand

- Mode registre

```
addu  $t0,$t0,$t1
```

- Mode immediat

```
addiu $t0,$t0,4
```

- Mode memòria

```
lw    $t0,0($t1)
```

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - mètode per especificar la ubicació d'un operand

- Mode registre

```
addu  $t0,$t0,$t1
```

- Mode immediat

```
addiu $t0,$t0,4
```

- Mode memòria

```
lw    $t0,0($t1)
```

- Mode relatiu al PC

```
beq   $t0,$t1,etiqueta
```

Modes d'adreçament MIPS (recapitulació)

- "Mode d'adreçament"
  - mètode per especificar la ubicació d'un operand

- Mode registre

```
addu  $t0,$t0,$t1
```

- Mode immediat

```
addiu $t0,$t0,4
```

- Mode memòria

```
lw    $t0,0($t1)
```

- Mode relatiu al PC

```
beq   $t0,$t1,etiqueta
```

- Mode pseudodirecte

```
j     etiqueta
```

Sentències condicionals i iteratives

- if-then else
- switch
- while
- for
- do-while

Sentència *if-then-else*

C:

```
if (condició)
        cos_then
else
        cos_else
```

Sentència *if-then-else*

C:

```
if (condició)
        cos_then
else
        cos_else
```

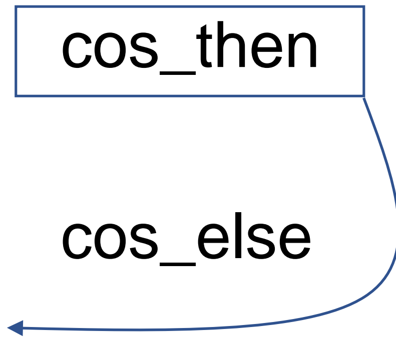
MIPS:

```
        avaluar condició
        salta si és falsa a sino



sino:
```

Sentència *if-then-else*

C:

if (condició)



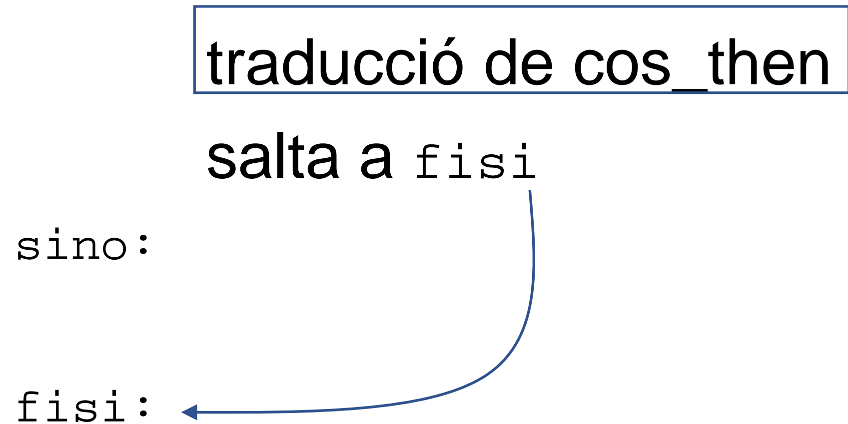
else

cos_else

MIPS:

avaluar condició

salta si és falsa a `sino`



Sentència *if-then-else*

C:

```
if (condició)
        cos_then
else
        cos_else
```

MIPS:

```
        avaluar condició
        salta si és falsa a sino
        traducció de cos_then
        salta a fisi
sino:
        traducció de cos_else
fisi:
```

Sentència *if-then-else*

- Traduir a MIPS
  (a, b, c, d són enters guardats en \$t0, \$t1, \$t2, \$t3)

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

Sentència *if-then-else*

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

```
if (a >= b)                          blt    $t0,$t1,sino
        d = a;
else
        d = b;            sino:
```

Sentència *if-then-else*

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

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


fisi:
```

Sentència *if-then-else*

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

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

Avaluació *lazy* de "&&" i "||"

- En C, els operadors booleans "&&" i "||" s'avaluen d'esquerra a dreta, i de forma *lazy*
  - *Lazy*: si l'operand esquerre determina el resultat, el dret **NO** s'avalua
  - Per exemple:

`x = 0 && func();` No es crida mai a `func()`

`x = 1 || func();` No es crida mai a `func()`

Sentència *if-then-else* amb condició "&&"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

Sentència *if-then-else* amb condició "&&"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

```
blt     $t0,$t1,sino
```

```
sino:
```

Sentència *if-then-else* amb condició "&&"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

```
        blt     $t0,$t1,sino
        bge     $t0,$t2,sino

sino:
```

Sentència *if-then-else* amb condició "&&"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $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:

fisi:
```

Sentència *if-then-else* amb condició "&&"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $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:
```

Sentència *if-then-else* amb condició "||"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

Sentència *if-then-else* amb condició "||"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

```
bge     $t0,$t1,llavors

llavors:
```

Sentència *if-then-else* amb condició "||"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $t3)

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

```
        bge     $t0,$t1,llavors
        bge     $t0,$t2,sino
llavors:


sino:
```

Sentència *if-then-else* amb condició "||"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $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:

fisi:
```

Sentència *if-then-else* amb condició "||"

- Traduir a MIPS
  (a, b, c, d són enters guardats en $t0, $t1, $t2, $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:
```

Sentència *switch*

```
int    val;
char   lletra;
switch (lletra) {
    case 'a':
            val=0;
            break;
    case 'b':
    case 'c':
            val=1;
            break;
 default:
            val=-1;
}
```

Sentència *switch*

```
int    val;
char   lletra;
switch (lletra) {
    case 'a':
            val=0;
            break;
    case 'b':
    case 'c':
            val=1;
            break;
 default:
            val=-1;
}
```

- Es pot traduir com una cadena de if-then-elses

Sentència *switch*

```
int    val;
char   lletra;
switch (lletra) {
    case 'a':
          val=0;
          break;
    case 'b':
    case 'c':
          val=1;
          break;
 default:
          val=-1;
}
```

- Es pot traduir com una cadena de if-then-elses
- Si hi ha molts casos, és més eficient usar una "jump table"

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:
```

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.

```
       [bgt]    $t0, $t1,    [etiq2]
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:
```

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.

```
       bgt   $t0, $t1,   etiq2
etq1:  bne   $t1, $zero, etiq5
etq2:  andi  $t3, $t1,   [    ]
etq3:  [  ]  $t5, $t3,   [    ]
etq4:  ble   $t5, $zero, [    ]
etq5:  li    $t2, 5
etq6:  b     [    ]
etq7:  subu  $t2, $t0, $t1
etq8:
```

Exercici

Donada la següent sentència escrita en alt nivell en C:

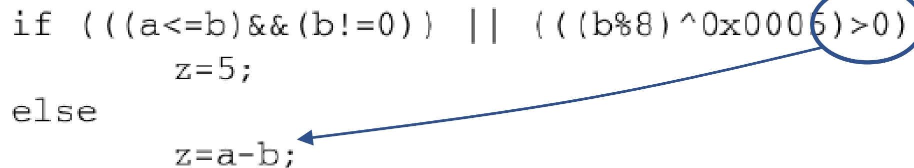
```
if (((a<=b)&&(b!=0)) || (((b%8)^0x0005)>0))
        z=5;
else
        z=a-b;
```

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.

```
        bgt     $t0, $t1,    etiq2
etq1:   bne     $t1, $zero,  etiq5
etq2:   andi    $t3, $t1,    7
etq3:   xori    $t5, $t3,    0x0005
etq4:   ble     $t5, $zero,  [    ]
etq5:   li      $t2, 5
etq6:   b       [    ]
etq7:   subu    $t2, $t0, $t1
etq8:
```

Exercici

Donada la següent sentència escrita en alt nivell en C:



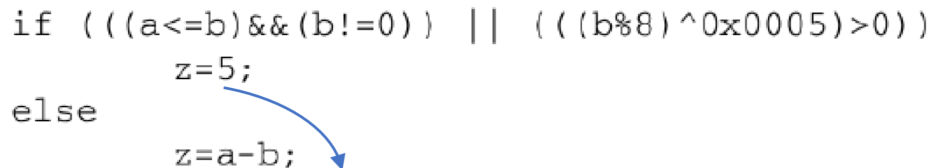
```
if (((a<=b)&&(b!=0)) || (((b%8)^0x0005)>0))
        z=5;
else
        z=a-b;
```

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.

	bgt	$t0, $t1,	etiq2
etq1:	bne	$t1, $zero,	etiq5
etq2:	andi	$t3, $t1,	7
etq3:	xori	$t5, $t3,	0x0005
etq4:	ble	$t5, $zero,	etiq7
etq5:	li	$t2, 5	
etq6:	b		
etq7:	subu	$t2, $t0, $t1	
etq8:			

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.

	bgt	$t0, $t1,	etiq2
etq1:	bne	$t1, $zero,	etiq5
etq2:	andi	$t3, $t1,	7
etq3:	xori	$t5, $t3,	0x0005
etq4:	ble	$t5, $zero,	etiq7
etq5:	li	$t2, 5	
etq6:	b	etiq8	
etq7:	subu	$t2, $t0, $t1	
etq8:			

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.

```
        bgt     $t0, $t1,    etiq2
etq1:   bne     $t1, $zero,  etiq5
etq2:   andi    $t3, $t1,    7
etq3:   xori    $t5, $t3,    0x0005
etq4:   ble     $t5, $zero,  etiq7
etq5:   li      $t2, 5
etq6:   b       etiq8
etq7:   subu    $t2, $t0, $t1
etq8:
```

Sentència *while*

- Executa zero o més iteracions mentre es compleixi la condició

C:

```
while (condició)
        cos_while
```

Sentència *while*

- Executa zero o més iteracions mentre es compleixi la condició

C:

```
while (condició)
        cos_while
```

MIPS:

```
while:
        avaluar condició
        salta si és falsa a fiwhile



fiwhile:
```

Sentència *while*

- Executa zero o més iteracions mentre es compleixi la condició

C:

```
while (condició)
        cos_while
```

MIPS:

```
while:
        avaluar condició
        salta si és falsa a fiwhile
        traducció de cos_while
        salta a while
fiwhile:
```

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

```
        move    $t3,$zero
while:



fiwhile:
```

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

```
         move   $t3,$zero
while:
         blt    $t1,$t2,fiwhile



fiwhile:
```

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

```
        move    $t3,$zero
while:
        blt     $t1,$t2,fiwhile
        subu    $t1,$t1,$t2

fiwhile:
```

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

```
          move   $t3,$zero
while:
          blt    $t1,$t2,fiwhile
          subu   $t1,$t1,$t2
          addiu  $t3,$t3,1
          b      while
fiwhile:
```

Sentència *while*

- Traduir a MIPS
  (dd, dr, q són enters guardats en $t1, $t2, $t3)

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

```
        move   $t3,$zero
while:
        blt    $t1,$t2,fiwhile
        subu   $t1,$t1,$t2
        addiu  $t3,$t3,1
        b      while
fiwhile:
```

Sentència *while*

- Optimització:
  - Estalviar un salt, avaluant la condició al final

Versió bàsica:

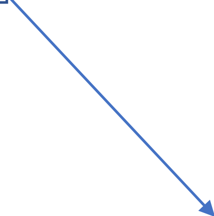
```
      move    $t3,$zero

while:
      blt     $t1,$t2,fiwhile
      subu    $t1,$t1,$t2
      addiu   $t3,$t3,1
      b       while
fiwhile:
```

Sentència *while*

- Optimització:
  - Estalviar un salt, avaluant la condició al final

Versió bàsica:

```
      move    $t3,$zero

while:
      blt     $t1,$t2,fiwhile
      subu    $t1,$t1,$t2
      addiu   $t3,$t3,1
      b       while
fiwhile:
```



Versió optimitzada:

```
      move    $t3,$zero

while:

      subu    $t1,$t1,$t2
      addiu   $t3,$t3,1
      bge     $t1,$t2,while
fiwhile:
```

Sentència *while*

- Optimització:
  - Estalviar un salt, avaluant la condició al final
  - Cal assegurar-se que es manté la semàntica

Versió bàsica:

```
      move    $t3,$zero

while:
      blt     $t1,$t2,fiwhile
      subu    $t1,$t1,$t2
      addiu   $t3,$t3,1
      b       while
fiwhile:
```

Versió optimitzada:

```
      move    $t3,$zero
      blt     $t1,$t2,fiwhile
while:

      subu    $t1,$t1,$t2
      addiu   $t3,$t3,1
      bge     $t1,$t2,while
fiwhile:
```

Sentència *for*

```
for (previ; condició; post)
        cos_for
```

- És equivalent a un while:

```
previ;
while (condició) {
        cos_for;
        post;
}
```

Sentència do-*while*

- Executa una o més iteracions mentre es compleixi la condició (la primera iteració sempre s'executa)

C:

```
do
        cos_do-while
while (condició);
```

Sentència do-*while*

- Executa una o més iteracions mentre es compleixi la condició (la primera iteració sempre s'executa)

C:

```
do
        cos_do-while
while (condició);
```

MIPS:

```
do:
        traducció de cos_do-while
```

Sentència do-*while*

- Executa una o més iteracions mentre es compleixi la condició (la primera iteració sempre s'executa)

C:

```
do
        cos_do-while
while (condició);
```

MIPS:

```
do:
        traducció de cos_do-while
        avaluar condició
        salta si és certa a do
```

Subrutines

- Crida i retorn
- Paràmetres i resultats
- Variables locals. La pila
- Subrutines multinivell

Subrutines

- Peces de codi *reutilitzables* que es poden invocar des de diferents punts
- També conegudes com funcions, mètodes, subprogrames, procediments...
- Executen una tasca determinada segons uns *paràmetres d'entrada* i poden *retornar un resultat*
- Essencials en qualsevol llenguatge

Subrutines

- Cal garantir interoperatbilitat...
  - Entre codi escrit per altres persones
  - Entre codi escrit amb altres compiladors

Subrutines

- Cal garantir interoperatbilitat...
  - Entre codi escrit per altres persones
  - Entre codi escrit amb altres compiladors

- ...adoptant les regles d'un estàndard comú (ABI)
  - Crida i retorn
  - Pas de paràmetres
  - Retorn del resultat
  - Ús de registres i memòria (pila)

Subrutines

- Cal garantir interoperatbilitat...
  - Entre codi escrit per altres persones
  - Entre codi escrit amb altres compiladors

- ...adoptant les regles d'un estàndard comú (ABI)
  - Crida i retorn
  - Pas de paràmetres
  - Retorn del resultat
  - Ús de registres i memòria (pila)

- L'ABI és particular per a cada ISA, llenguatge i/o Sistema Operatiu

Subrutines

- Com s'executa una subrutina?

<u>Programa</u>

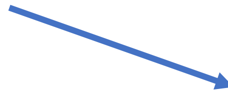
1. Posa els paràmetres on la subrutina hi pugui accedir

Subrutines

- Com s'executa una subrutina?

Programa

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina



Subrutina

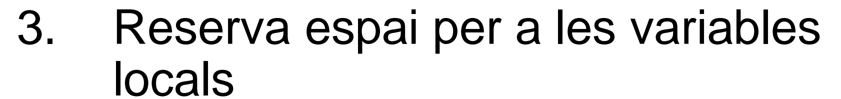
Subrutines

- Com s'executa una subrutina?

Programa

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina

Subrutina



3. Reserva espai per a les variables locals

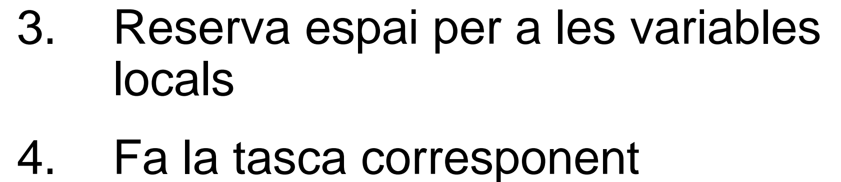
Subrutines

- Com s'executa una subrutina?

<u>Programa</u>

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina

<u>Subrutina</u>



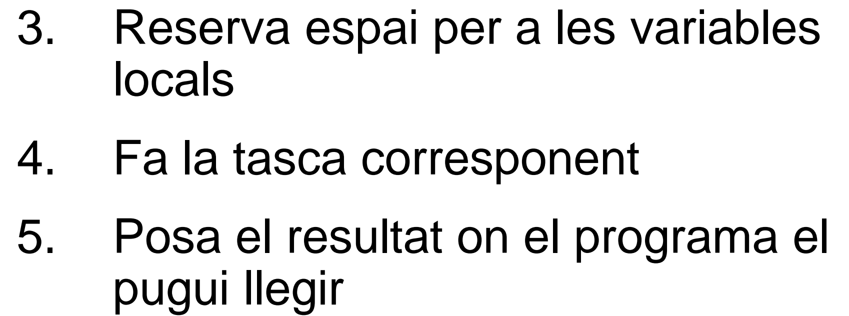
Subrutines

- Com s'executa una subrutina?

Programa

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina

Subrutina



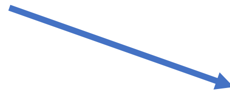
3. Reserva espai per a les variables locals
4. Fa la tasca corresponent
5. Posa el resultat on el programa el pugui llegir

Subrutines

- Com s'executa una subrutina?

Programa

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina



Subrutina

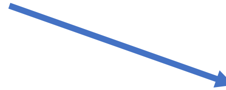
3. Reserva espai per a les variables locals
4. Fa la tasca corresponent
5. Posa el resultat on el programa el pugui llegir
6. Retorna el control al programa

Subrutines

- Com s'executa una subrutina?

Programa

1. Posa els paràmetres on la subrutina hi pugui accedir
2. Transfereix el control a la subrutina



Subrutina

3. Reserva espai per a les variables locals
4. Fa la tasca corresponent
5. Posa el resultat on el programa el pugui llegir
6. Retorna el control al programa

7. Llegeix el resultat

Crida i retorn

- La instrucció `jal` memoritza l'adreça de retorn en $ra
- La instrucció `jr` salta a $ra, adreça de retorn en cada cas

Crida

```
        jal   subr       # $ra = adr1
adr1:

        ...

        jal   subr       # $ra = adr2
adr2:

        ...
```

1

Retorn

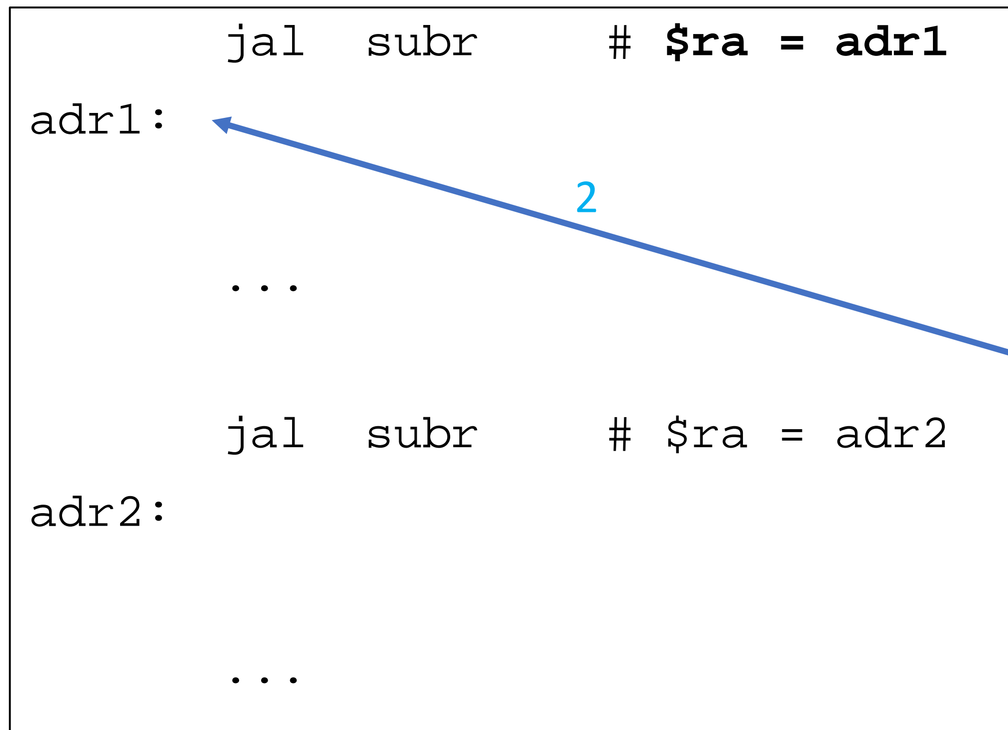
```
subr:
        . . .

        jr   $ra
```

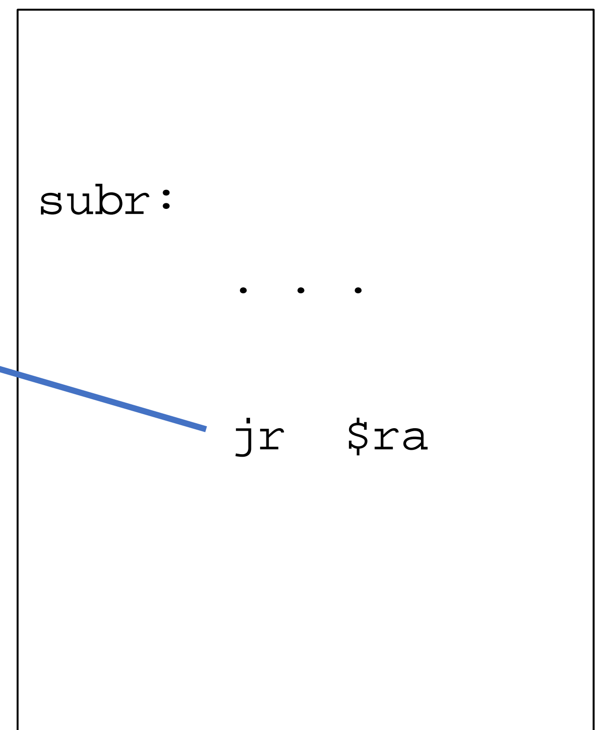
Crida i retorn

- La instrucció `jal` memoritza l'adreça de retorn en $ra
- La instrucció `jr` salta a $ra, adreça de retorn en cada cas

<u>Crida</u>

<u>Retorn</u>



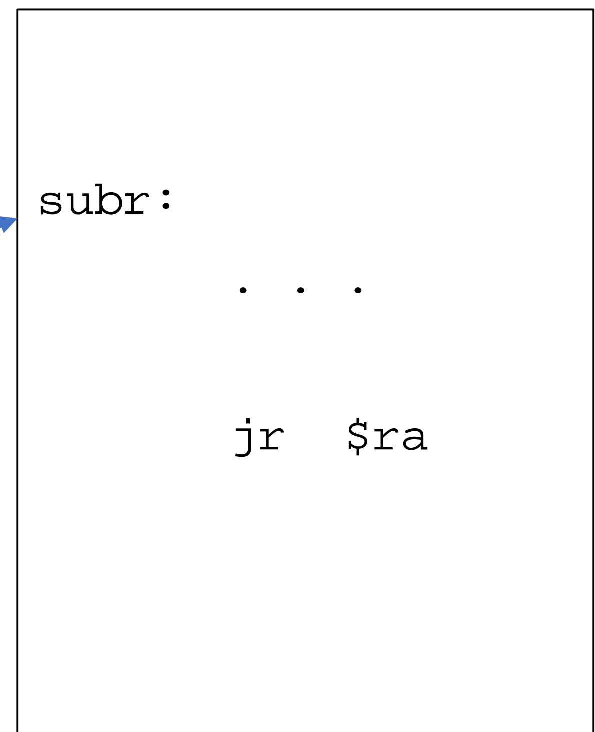


Crida i retorn

- La instrucció `jal` memoritza l'adreça de retorn en $ra
- La instrucció `jr` salta a $ra, adreça de retorn en cada cas

Crida

```
        jal   subr       # $ra = adr1
adr1:

        ...

        jal   subr       # $ra = adr2
adr2:

        ...
```



Retorn

```
subr:
        . . .

        jr   $ra
```

Crida i retorn

- La instrucció `jal` memoritza l'adreça de retorn en $ra
- La instrucció `jr` salta a $ra, adreça de retorn en cada cas

Crida

```
        jal   subr       # $ra = adr1
adr1:

        ...

        jal   subr       # $ra = adr2
adr2:

        ...
```

Retorn

```
subr:
        . . .

        jr   $ra
```

4

Paràmetres i resultats

- **Paràmetres**
  - En els registres $a0 - $a3, en ordre
    - Si és char o short, s'extén prèviament a 32 bits
      (extensió de zeros si és unsigned, o extensió de signe si signed)
  - Els de coma flotant es passen en $f12 i $f14 (veure Tema 5)

- **Resultat**
  - En el registre $v0
    - Si és char o short, s'extén prèviament a 32 bits
  - Si és de coma flotant es passa en $f0

Paràmetres i resultats

En C:

```
void main() {
  int x, y, z;
  // en $t0, $t1, $t2
  z = suma2(x, y);
}
```

```
int suma2(int a, int b)
{
  return a+b;
}
```

Paràmetres i resultats

En C:

```
void main() {
   int x, y, z;
   // en $t0, $t1, $t2
   z = suma2(x, y);
}
```

```
int suma2(int a, int b)
{
   return a+b;
}
```

En MIPS:

```
main:
      move    $a0, $t0 # x
      move    $a1, $t1 # y
```

Paràmetres i resultats

En C:

```
void main() {
   int x, y, z;
   // en $t0, $t1, $t2
   z = suma2(x, y);
}
```

```
int suma2(int a, int b)
{
   return a+b;
}
```

En MIPS:

```
main:
        move    $a0, $t0 # x
        move    $a1, $t1 # y
        jal     suma2
```

Paràmetres i resultats

En C:

```
void main() {
   int x, y, z;
   // en $t0, $t1, $t2
   z = suma2(x, y);
}
```

```
int suma2(int a, int b)
{
   return a+b;
}
```

En MIPS:

```
main:
        move    $a0, $t0 # x
        move    $a1, $t1 # y
        jal     suma2
```

```
suma2:
        addu    $v0, $a0, $a1
        jr      $ra
```

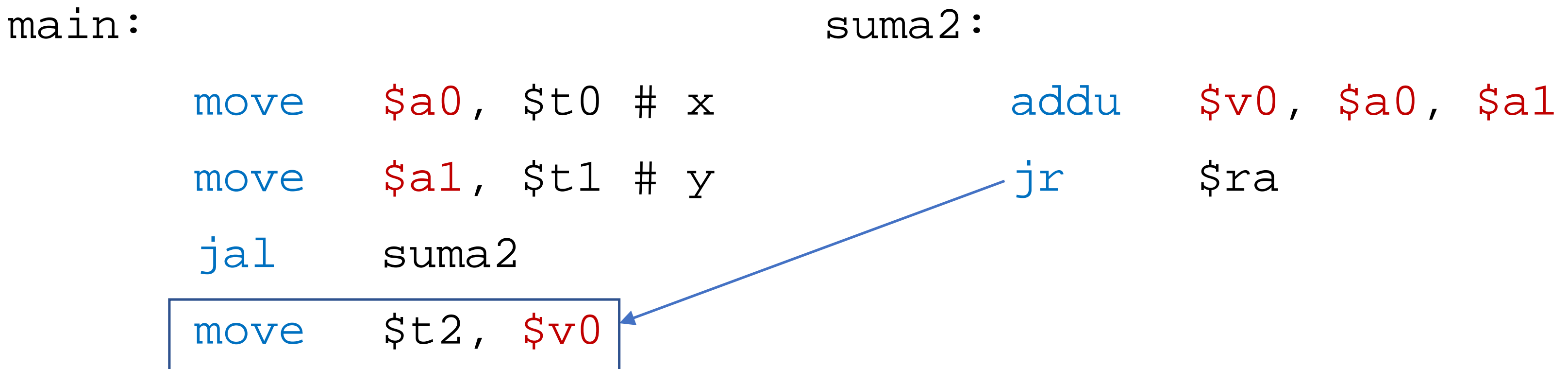
Paràmetres i resultats

En C:

```
void main() {
  int x, y, z;
  // en $t0, $t1, $t2
  z = suma2(x, y);
}
```

```
int suma2(int a, int b)
{
  return a+b;
}
```

En MIPS:



```
main:
        move    $a0, $t0 # x
        move    $a1, $t1 # y
        jal     suma2
        move    $t2, $v0
```

```
suma2:
        addu    $v0, $a0, $a1
        jr      $ra
```

Paràmetres i resultats

- Paràmetres *vector* o *matriu*: es passa el *punter a l'adreça base*
- Amb aquestes declaracions...

```
int v[10]
int m[10][100]
int a, *p;
```

- ... i les següents capçaleres...

```
int f(int *x);
int g(int x[]);
int h(int x[][100]);
```

Paràmetres i resultats

- Paràmetres *vector* o *matriu*: es passa el *punter a l'adreça base*

- Amb aquestes declaracions...

```
int v[10]
int m[10][100]
int a, *p;
```

- ... podem fer les següents crides

```
a = f(p);
a = f(&v[4]);
a = f(v);
```

- ... i les següents capçaleres...

```
int f(int *x);
```

Paràmetres i resultats

- Paràmetres *vector* o *matriu*: es passa el *punter a l'adreça base*

- Amb aquestes declaracions...

```
int v[10]
int m[10][100]
int a, *p;
```

- ... i les següents capçaleres...

```
int f(int *x);
int g(int x[]);
```

- ... podem fer les següents crides

```
a = f(p);
a = f(&v[4]);
a = f(v);
```

```
a = g(p);
a = g(&v[4]);
a = g(v);
```

Paràmetres i resultats

- Paràmetres *vector* o *matriu*: es passa el *punter a l'adreça base*

- Amb aquestes declaracions...

```
int v[10]
int m[10][100]
int a, *p;
```

- ... i les següents capçaleres...

```
int f(int *x);
int g(int x[]);
int h(int x[][100]);
```

- ... podem fer les següents crides

```
a = f(p);
a = f(&v[4]);
a = f(v);

a = g(p);
a = g(&v[4]);
a = g(v);

a = h(m);
```

Paràmetres i resultats

- Exemple. Traduir a MIPS

```
short vec[3] = {5, 7, 9};
short sumav(short v[]);



void main() {
  short res; // guardat en $t0
  res = sumav(vec);
  ...
}
```

Paràmetres i resultats

- Exemple. Traduir a MIPS

```c
short vec[3] = {5, 7, 9};
short sumav(short v[]);


void main() {
  short res; // guardat en $t0
  res = sumav(vec);
  ...
}
```

```
        .data
vec:    .half 5, 7, 9



main:
        la      $a0, vec
```

Paràmetres i resultats

- Exemple. Traduir a MIPS

```c
short vec[3] = {5, 7, 9};
short sumav(short v[]);


void main() {
   short res; // guardat en $t0
   res = sumav(vec);
   ...
}
```

```
        .data
vec:    .half 5, 7, 9


main:
        la      $a0, vec
        jal     sumav
```

Paràmetres i resultats

- Exemple. Traduir a MIPS

```c
short vec[3] = {5, 7, 9};
short sumav(short v[]);

void main() {
  short res; // guardat en $t0
  res = sumav(vec);
  ...
}
```

```
        .data
vec:    .half 5, 7, 9

main:
        la      $a0, vec
        jal     sumav
        move    $t0, $v0
        ...
```

Paràmetres *per referència* en C?

- C no té paràmetres *per referència*, sols *per valor*
- Volem que la subrutina sumi 7 al paràmetre real (a o b)

```
int a, b;



void main() {
  suma7(   ?   );
    . . .
}
```

Paràmetres *per referència* en C?

- C no té paràmetres *per referència*, sols *per valor*
- Volem que la subrutina sumi 7 al paràmetre real (a o b)
  - Li passem un punter que hi apunti

```
int a, b;
int *p = &a;

void main() {
  suma7(p);          // executarà a=a+7
  . . .
}

void suma7(int *ptr) {
      *ptr = *ptr + 7;
}
```

Paràmetres *per referència* en C?

- C no té paràmetres *per referència*, sols *per valor*
- Volem que la subrutina sumi 7 al paràmetre real (`a` o `b`)
  - Li passem un punter que hi apunti

```
int a, b;
int *p = &a;

void main() {
   suma7(p);              // executarà a=a+7
   suma7(&b);             // executarà b=b+7
}

void suma7(int *ptr) {
      *ptr = *ptr + 7;
}
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
  int k; // k guardat a $t0
  ...
  z = funcB(w, x, k);
  ...
}
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```c
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
   int k; // k guardat a $t0
   ...
   z = funcB(w, x, k);
   ...
}
```

```
funcA:
        ...
        la      $a0, w          # w
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```c
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
  int k; // k guardat a $t0
  ...
  z = funcB(w, x, k);
  ...
}
```

```
funcA:
        ...
        la    $a0, w          # w
        la    $t4, x
        lh    $a1, 0($t4)     # x
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
  int k; // k guardat a $t0
  ...
  z = funcB(w, x, k);
  ...
}
```

```
funcA:
      ...
      la    $a0, w        # w
      la    $t4, x
      lh    $a1, 0($t4)   # x
      move  $a2, $t0      # k
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```c
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
  int k; // k guardat a $t0
  ...
  z = funcB(w, x, k);
  ...
}
```

```
funcA:
        ...
        la      $a0, w          # w
        la      $t4, x
        lh      $a1, 0($t4)     # x
        move    $a2, $t0        # k
        jal     funcB
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```c
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
   int k; // k guardat a $t0
   ...
   z = funcB(w, x, k);
   ...
}
```

```
funcA:
        ...
        la      $a0, w          # w
        la      $t4, x
        lh      $a1, 0($t4)     # x
        move    $a2, $t0        # k
        jal     funcB
        la      $t4, z
        sh      $v0, 0($t4)     # z
```

Exemple

- Traduir a MIPS les sentències visibles de *funcA()*

```
short w[10], x, z;
short funcB(short *vec,
            short n, int i);

void funcA(){
   int k; // k guardat a $t0
   ...
   z = funcB(w, x, k);
   ...
}
```

```
funcA:
      ...
      la    $a0, w          # w
      la    $t4, x
      lh    $a1, 0($t4)     # x
      move  $a2, $t0        # k
      jal   funcB
      la    $t4, z
      sh    $v0, 0($t4)     # z
      ...
      jr    $ra
```

Exemple

- Traduir a MIPS la funció *funcB()*

```
short funcB(short *vec, short n, int i){
  return vec[i] - n;
}
```

Exemple

- Traduir a MIPS la funció *funcB()*

```
short funcB(short *vec, short n, int i){
  return vec[i] - n;
}
```

```
funcB:
  sll  $v0, $a2, 1      # 2*i
  addu $v0, $a0, $v0    # @vec[0] + 2*i
  lh   $v0, 0($v0)      # vec[i]
```

Exemple

- Traduir a MIPS la funció *funcB()*

```
short funcB(short *vec, short n, int i){
  return vec[i] - n;
}
```

```
funcB:
 sll   $v0, $a2, 1      # 2*i
  addu $v0, $a0, $v0    # @vec[0] + 2*i
  lh   $v0, 0($v0)      # vec[i]
  subu $v0, $v0, $a1    # vec[i] - n
  jr   $ra
```

Variables locals

- Es creen quan s'invoca la funció, i sols perviuen mentre s'executa
  - Valor indeterminat si no s'inicialitzen explícitament
  - Sols són visibles dins la funció
  - Algunes es guarden en **registres**

```
int funcA(int x, int y) {
      int a, b = 0;
      short vec[8];
      ...
}
```

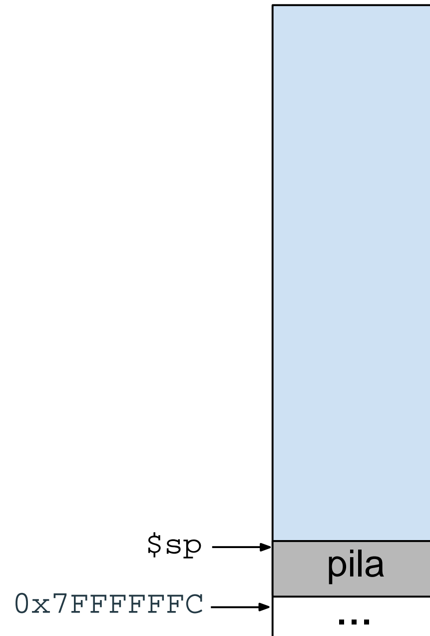
Variables locals

- Es creen quan s'invoca la funció, i sols perviuen mentre s'executa
  - Valor indeterminat si no s'inicialitzen explícitament
  - Sols són visibles dins la funció
  - Algunes es guarden en **registres**
  - Altres es guarden en **memòria**, a la **Pila del Programa**

```
int funcA(int x, int y) {
        int a, b = 0;
        short vec[8];
        ...
}
```

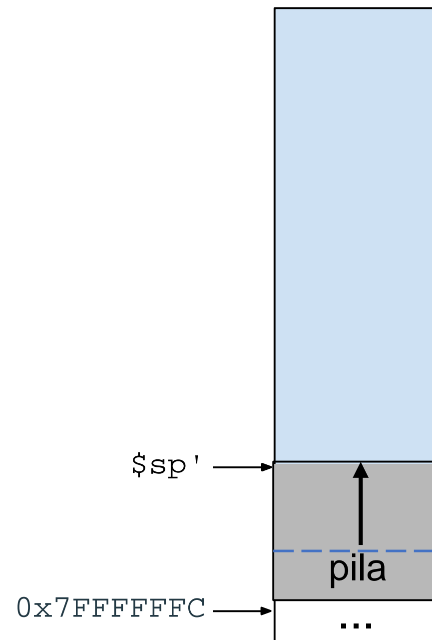
La Pila del Programa

- Regió de memòria que creix/minva dinàmicament
  - Seguint una estructura de pila (LIFO)
  - El registre $sp (stack pointer) apunta sempre al cim de la pila
  - Inicialment, $sp = 0x7FFFFFFC



La Pila del Programa

- Regió de memòria que creix/minva dinàmicament
  - Seguint una estructura de pila (LIFO)
  - El registre $sp (stack pointer) apunta sempre al cim de la pila
  - Inicialment, $sp = 0x7FFFFFFC
  - Creix des d'adreces altes cap a adreces més baixes
  - $sp ha de ser sempre múltiple de 4



La Pila del Programa

- Regió de memòria que creix/minva dinàmicament
  - Seguint una estructura de pila (LIFO)
  - El registre $sp (stack pointer) apunta sempre al cim de la pila
  - Inicialment, $sp = 0x7FFFFFFC
  - Creix des d'adreces altes cap a adreces més baixes
  - $sp ha de ser sempre múltiple de 4

- Cada subrutina manté a la pila el seu Bloc d'Activació
  - On hi guarda variables locals i altres dades privades
  - A l'inici de la subrutina, reserva el BA decrementant $sp
  - Al final, allibera el BA incrementant $sp

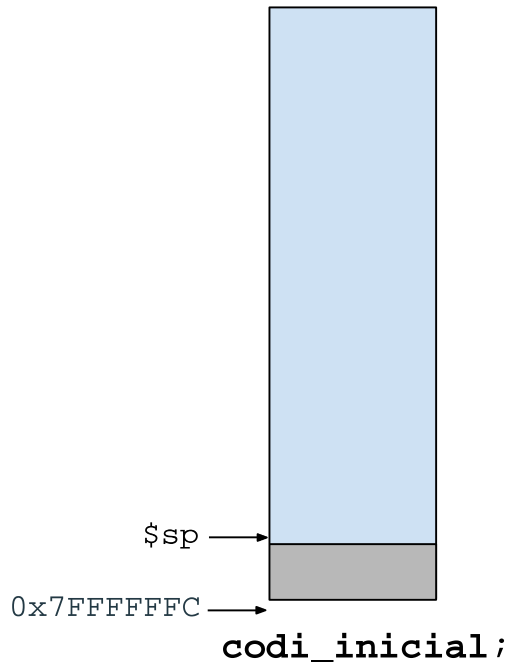
La Pila del Programa

- Exemple. Suposem les funcions `A()` i `B()`
  - Cadascuna crea el seu Bloc d'Activació en ser invocada
  - I l'allibera en retornar

```
codi_inicial;
...
```

```
A()
{
    sent_pre;
    B();
    sent_post;
}
```

```
B()
{
    cos_B;
}
```



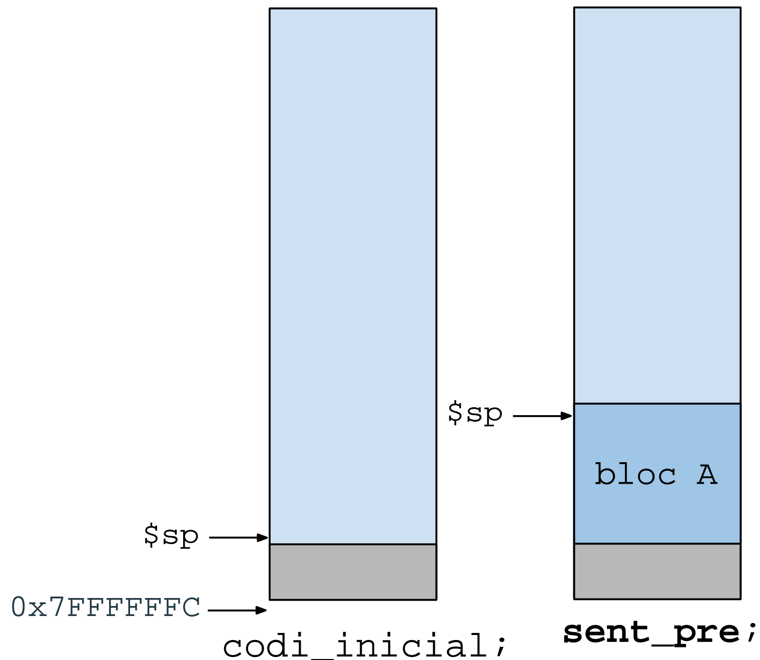
La Pila del Programa

- Exemple. Suposem les funcions `A()` i `B()`
  - Cadascuna crea el seu Bloc d'Activació en ser invocada
  - I l'allibera en retornar

```
codi_inicial;
...
```

```
A()
{
    sent_pre;
    B();
    sent_post;
}
```

```
B()
{
    cos_B;
}
```



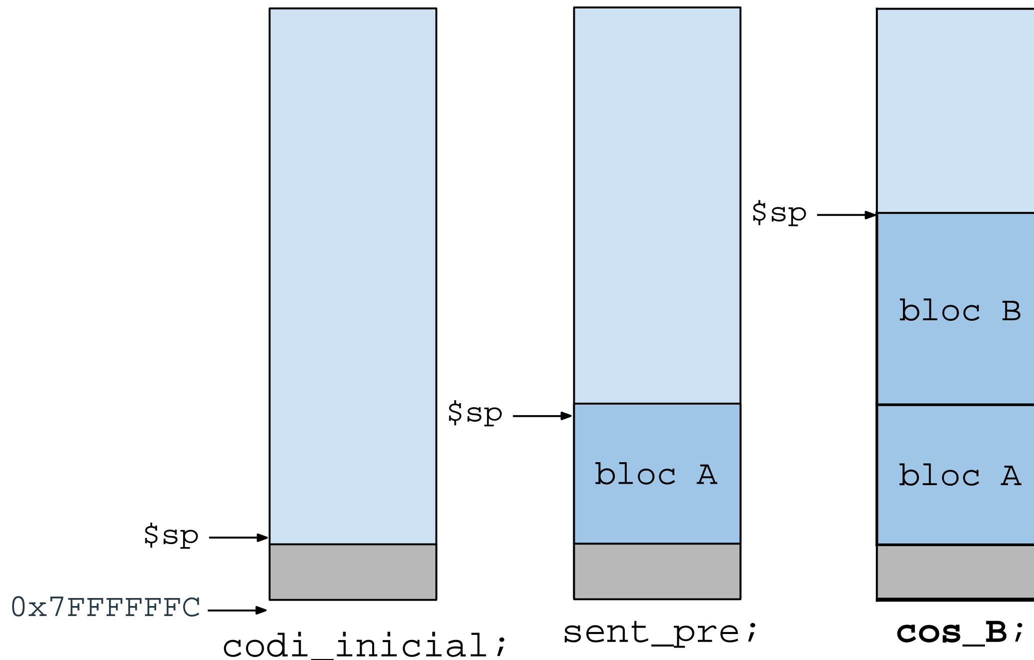
La Pila del Programa

- Exemple. Suposem les funcions `A()` i `B()`
  - Cadascuna crea el seu Bloc d'Activació en ser invocada
  - I l'allibera en retornar

```
codi_inicial;
...
```

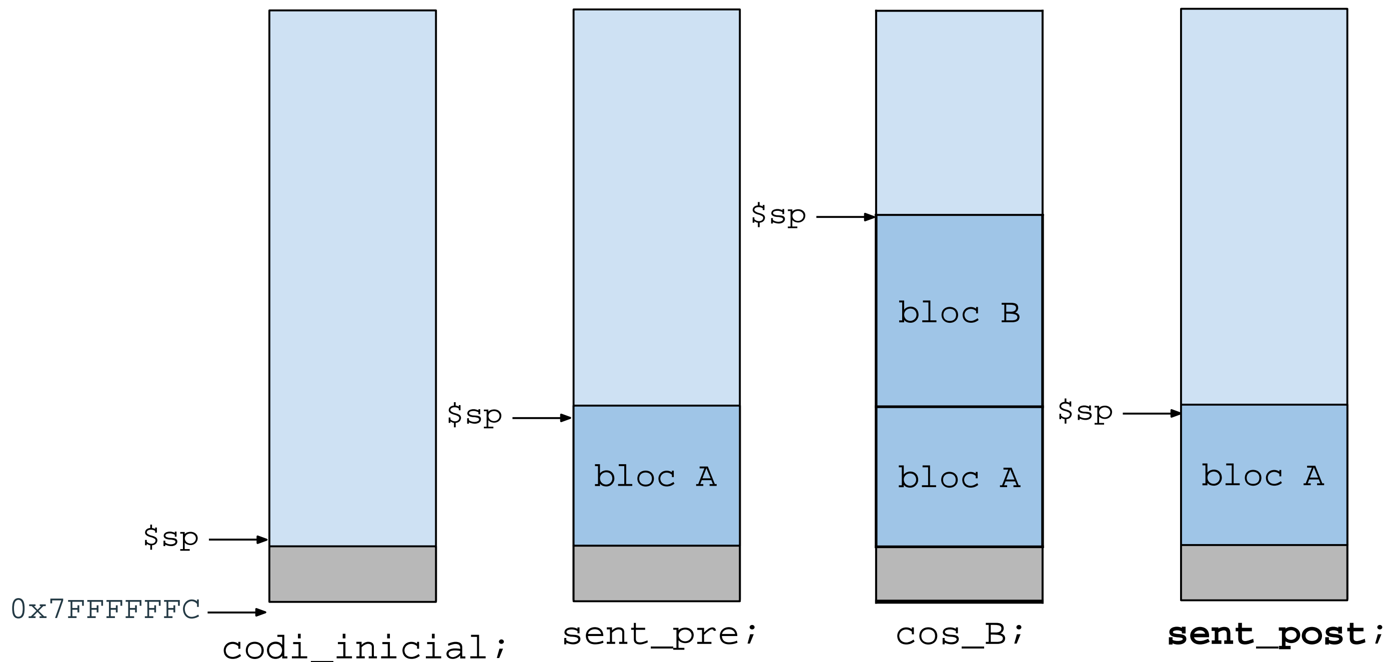
```
A()
{
        sent_pre;
        B();
        sent_post;
}
```

```
B()
{
        cos_B;
}
```



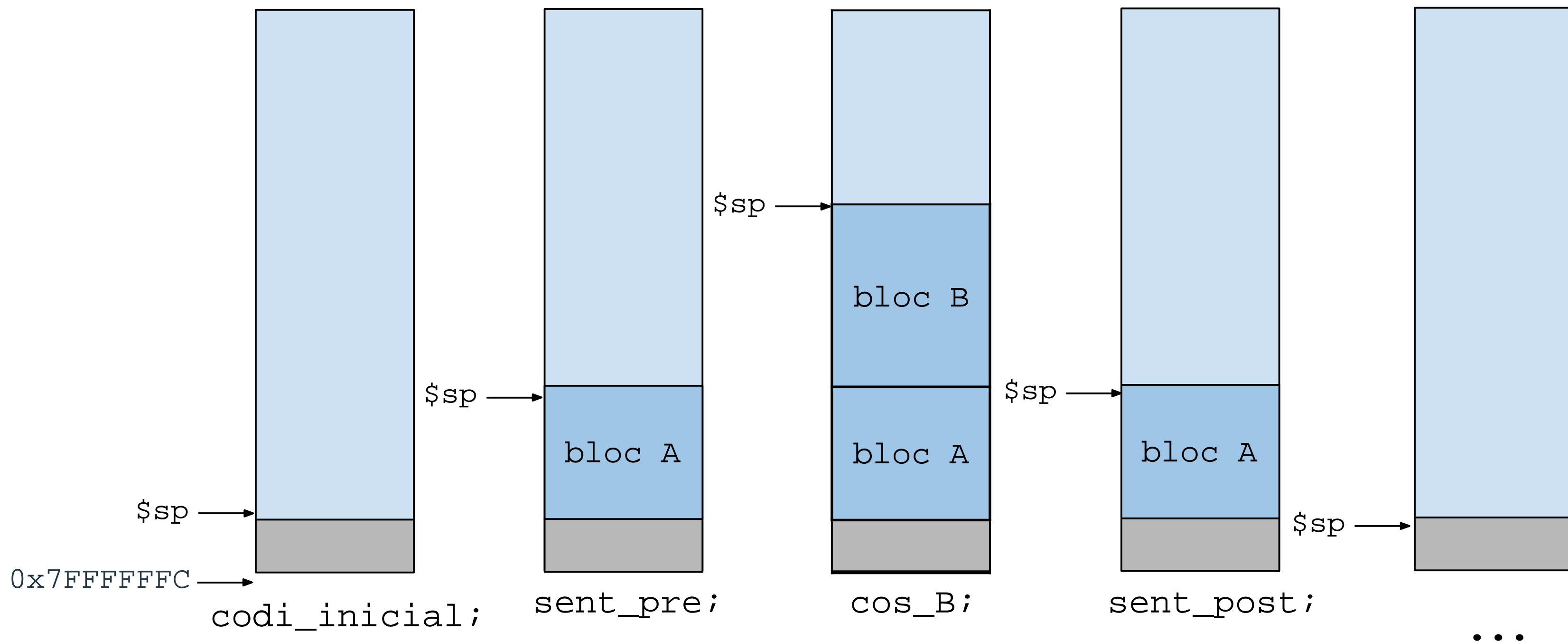
La Pila del Programa

- Exemple. Suposem les funcions `A()` i `B()`
  - Cadascuna crea el seu Bloc d'Activació en ser invocada
  - I l'allibera en retornar



La Pila del Programa

- Exemple. Suposem les funcions `A()` i `B()`
  - Cadascuna crea el seu Bloc d'Activació en ser invocada
  - I l'allibera en retornar



Regles de l'ABI per a variables locals

- Variable estructurada (vector, matriu, struct)
  - Es guarda a la pila, al Bloc d'Activació

Regles de l'ABI per a variables locals

- Variable estructurada (vector, matriu, struct)
  - Es guarda a la pila, al Bloc d'Activació

- Variable escalar
  - Es guarda als registres `$t0-$t9, $s0-$s7, $v0-$v1, $f0-$f31`
  - Si no hi ha prou registres, també es guarda a la pila
  - Si va precedida de l'operador "&" ("adreça de") en alguna sentència del codi, també es guarda a la pila

Regles de l'ABI per a variables locals

- Variable estructurada (vector, matriu, struct)
  - Es guarda a la pila, al Bloc d'Activació

- Variable escalar
  - Es guarda als registres `$t0-$t9, $s0-$s7, $v0-$v1, $f0-$f31`
  - Si no hi ha prou registres, també es guarda a la pila
  - Si va precedida de l'operador "&" ("adreça de") en alguna sentència del codi, també es guarda a la pila

- Regles per al Bloc d'Activació
  - Les variables locals es guarden en el mateix ordre que estan declarades al codi font, començant des del cim del Bloc
  - Respectant les normes d'alineament (afegint *padding* si cal)
  - La mida i l'adreça inicial del Bloc han de ser múltiples de 4

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

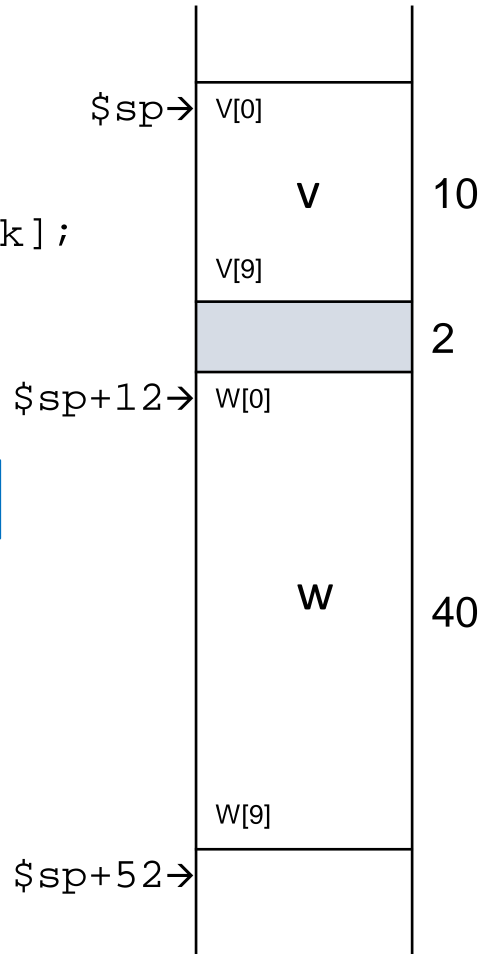
escalar: en registre ($t0)

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

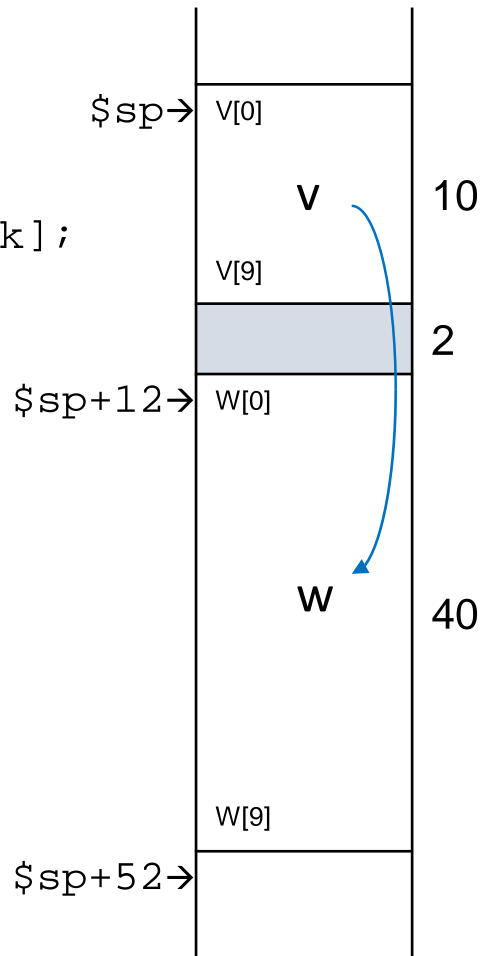
estructurades: a la pila



Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

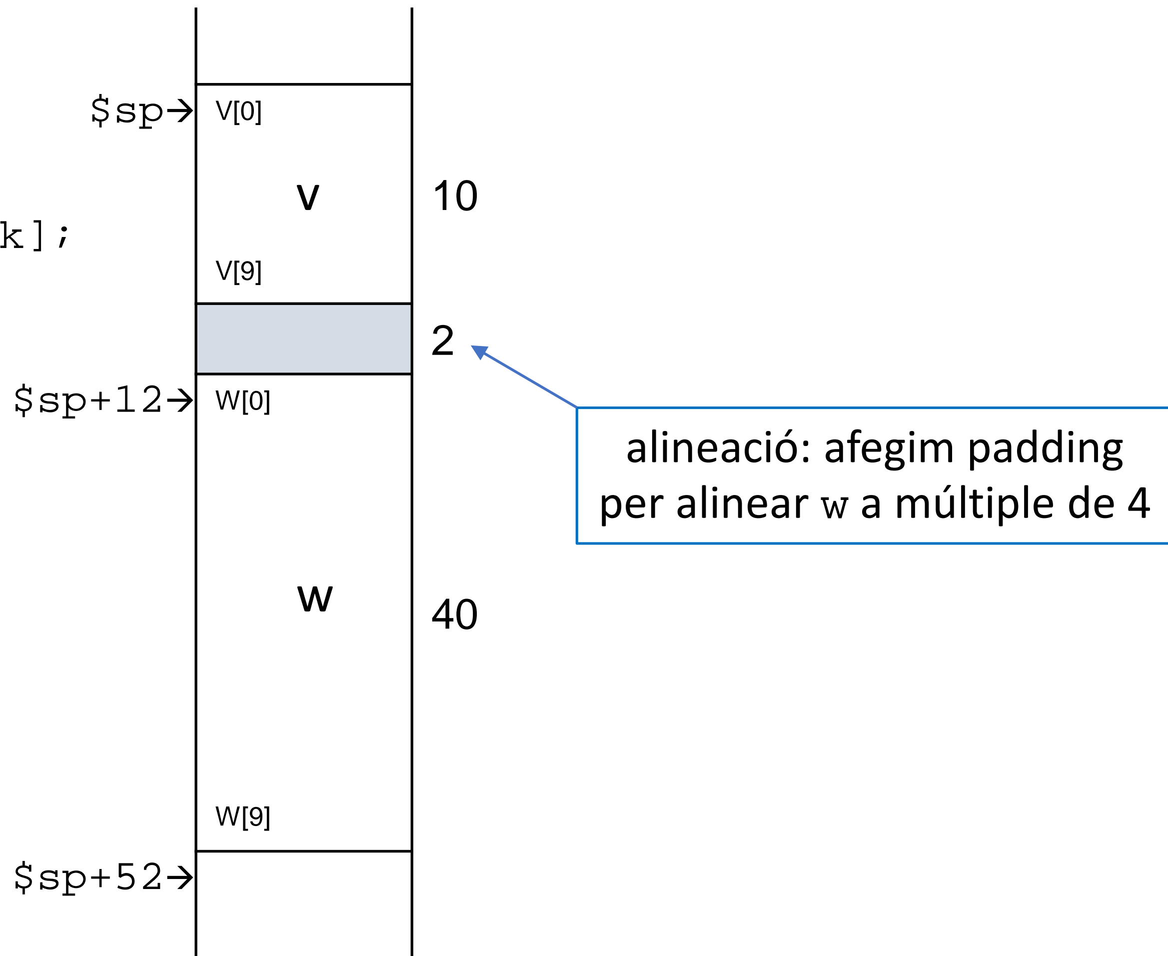


en ordre: primer `v` i després `w`

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

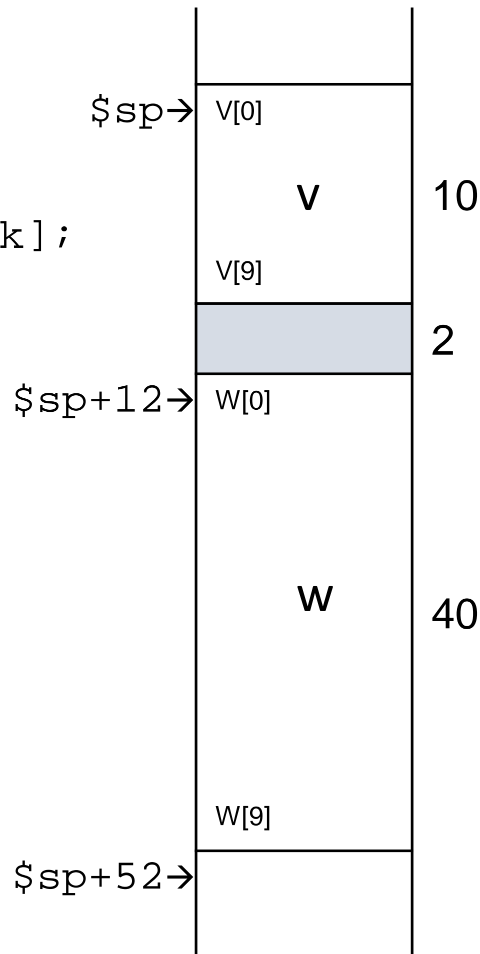
```
char func(int i){
  char v[10];
  int w[10], k;
  ...
  return v[w[i]+k];
}
```



Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

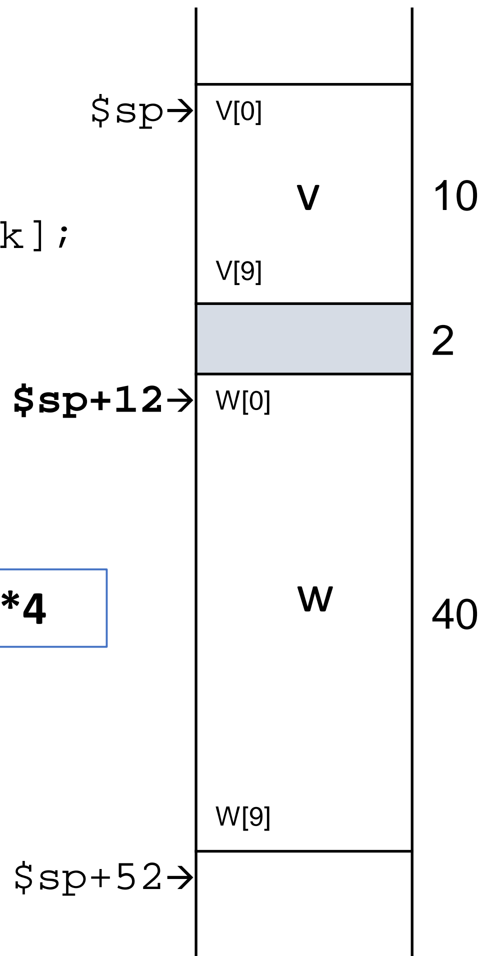



```
func:
    addiu $sp, $sp, -52   # reserva
    ...
```

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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



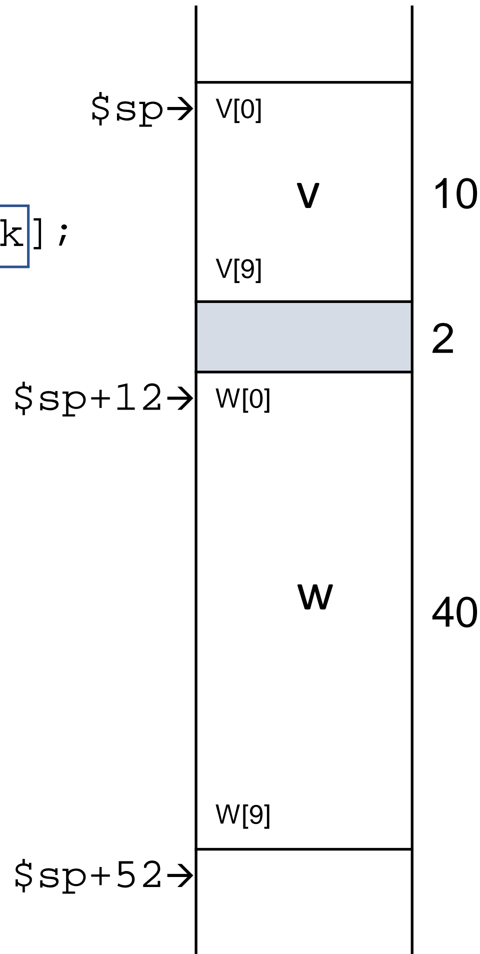
@w[i] = **$sp + 12 + i*4**

```
func:
    addiu $sp, $sp, -52   # reserva
    ...
    sll   $t4, $a0, 2     # i*4
    addu  $t4, $t4, $sp   # @w[i]
    lw    $t4, 12($t4)    # w[i]
```

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

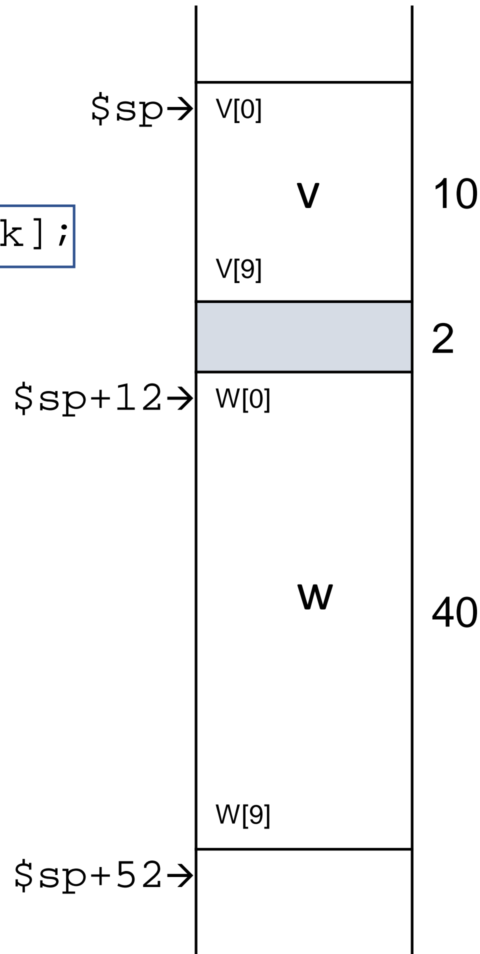



```
func:
   addiu $sp, $sp, -52   # reserva
   ...
  sll    $t4, $a0, 2     # i*4
  addu   $t4, $t4, $sp   # @w[i]
  lw     $t4, 12($t4)    # w[i]
  addu   $t4, $t4, $t0   # w[i]+k
```

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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

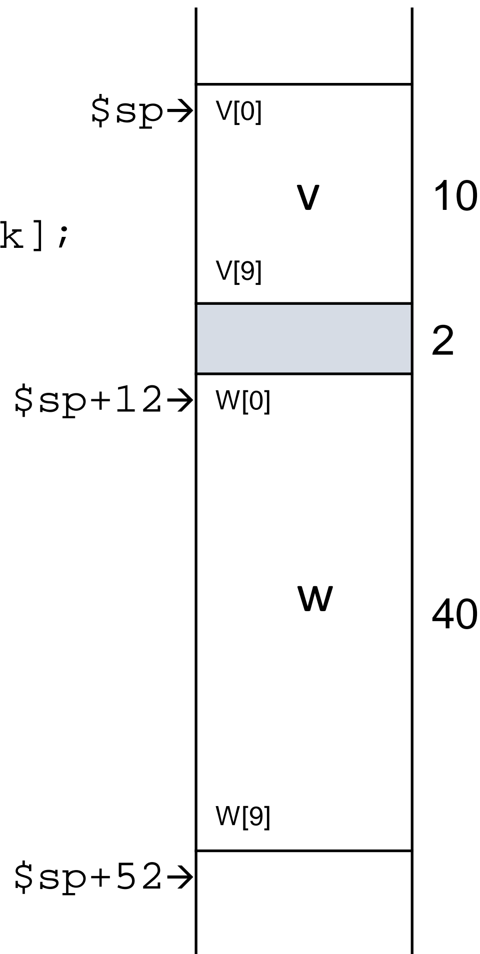



```
func:
   addiu $sp, $sp, -52   # reserva
   ...
   sll   $t4, $a0, 2     # i*4
   addu  $t4, $t4, $sp   # @w[i]
   lw    $t4, 12($t4)    # w[i]
   addu  $t4, $t4, $t0   # w[i]+k

   addu  $t5, $sp, $t4   # @v[..]
   lb    $v0, 0($t5)     # v[..]
```

Exemple

- Traduir la subrutina a MIPS i dibuixar el Bloc d'Activació

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




```
func:
    addiu $sp, $sp, -52   # reserva
    ...
    sll   $t4, $a0, 2     # i*4
    addu  $t4, $t4, $sp   # @w[i]
    lw    $t4, 12($t4)    # w[i]
    addu  $t4, $t4, $t0   # w[i]+k

    addu  $t5, $sp, $t4   # @v[..]
    lb    $v0, 0($t5)     # v[..]
    ...
    addiu $sp, $sp, 52    # allibera
    jr    $ra
```

Subrutines multinivell

- Són subrutines que criden altres subrutines
- Context d'una subrutina:
  - Paràmetres ($a0-$a3, $f12, $f14)
  - Adreça de retorn ($ra)
  - Punter de pila ($sp)
  - Càlculs intermedis en registres temporals
  - Variables locals en el bloc d'activació

Subrutines multinivell

- Són subrutines que criden altres subrutines
- Context d'una subrutina:
  - Paràmetres ($a0-$a3, $f12, $f14)
  - Adreça de retorn ($ra)
  - Punter de pila ($sp)
  - Càlculs intermedis en registres temporals
  - Variables locals en el bloc d'activació
- Tota subrutina ha de garantir que no modifica el context de la subrutina que l'ha invocat

Subrutines multinivell

- Són subrutines que criden altres subrutines
- Context d'una subrutina:
  - Paràmetres ($a0-$a3, $f12, $f14)
  - Adreça de retorn ($ra)
  - Punter de pila ($sp)
  - Càlculs intermedis en registres temporals
  - Variables locals en el **bloc d'activació**
- Tota subrutina ha de garantir que no modifica el context de la subrutina que l'ha invocat
- **Problema**: com preservar el context de la rutina que ens ha cridat?
  - El **bloc d'activació** és fàcil: cadascú toca sols el seu!

Subrutines multinivell

- Són subrutines que criden altres subrutines
- Context d'una subrutina:
  - Paràmetres ($a0-$a3, $f12, $f14)
  - Adreça de retorn ($ra)
  - Punter de pila ($sp)
  - Càlculs intermedis en registres temporals
  - Variables locals en el bloc d'activació
- Tota subrutina ha de garantir que no modifica el context de la subrutina que l'ha invocat
- **Problema**: com preservar el context de la rutina que ens ha cridat?
  - El bloc d'activació és fàcil: cadascú toca sols el seu!
  - Però els **registres** són comuns: Com sabem quins registres usa la rutina que ens ha cridat i no podem modificar?

Subrutines multinivell

- Un programador escriu el codi de `f`

```
f:
        ...
        li    $t0, 13
        ...
        jal   g
        ...
        addiu $t0, $t0, -13
        beq   $t0, $zero, etiq
```

beq saltarà? Sí

Subrutines multinivell

- Un programador escriu el codi de `f`
- Un altre programador escriu el codi de `g`

```
f:
        ...
        li     $t0, 13
        ...
        jal    g
        ...
        addiu  $t0, $t0, -13
        beq    $t0, $zero, etiq
```

```
g:
        ...
        li     $t0, 0
        ...
        jr     $ra
```

`beq saltarà? ~~Sí~~ No!!`

`g` ha modificat el context de `f`: `$t0`

Salvar i restaurar registres

- Solució trivial (però ineficient)
  - Garantir que, en retornar, TOTS els registres tenen el mateix estat que quan s'ha invocat la subrutina
  - Salvar una còpia al Bloc d'Activació abans de modificar-los, i restaurar-la abans de retornar

Salvar i restaurar registres

- Solució trivial (però ineficient)
  - Garantir que, en retornar, TOTS els registres tenen el mateix estat que quan s'ha invocat la subrutina
  - Salvar una còpia al Bloc d'Activació abans de modificar-los, i restaurar-la abans de retornar
- Solució del ABI de MIPS
  - Minimitza el nombre de registres a salvar
  - Divideix els registres en temporals i segurs

Temporals	Segurs
$t0 - $t9	$s0 - $s7
$v0 - $v1	$sp
$a0 - $a3	$ra
$f0 - $f19	$f20 - $f31

Salvar i restaurar registres

- Solució trivial (però ineficient)
  - Garantir que, en retornar, TOTS els registres tenen el mateix estat que quan s'ha invocat la subrutina
  - Salvar una còpia al Bloc d'Activació abans de modificar-los, i restaurar-la abans de retornar
- Solució del ABI de MIPS
  - Minimitza el nombre de registres a salvar
  - Divideix els registres en temporals i segurs

Temporals	Segurs
$t0 - $t9	$s0 - $s7
$v0 - $v1	$sp
$a0 - $a3	$ra
$f0 - $f19	$f20 - $f31

Regla: Quan la subrutina acaba ha de deixar els registres segurs en el mateix estat que tenien quan s'ha invocat

Salvar i restaurar registres

Mètode en dos passos:

1. Determinar quins registres segurs s'usaran
   - Identificant quines dades emmagatzemades en registres es generaran ABANS d'una crida a subrutina, i s'usaran DESPRÉS de la crida

Salvar i restaurar registres

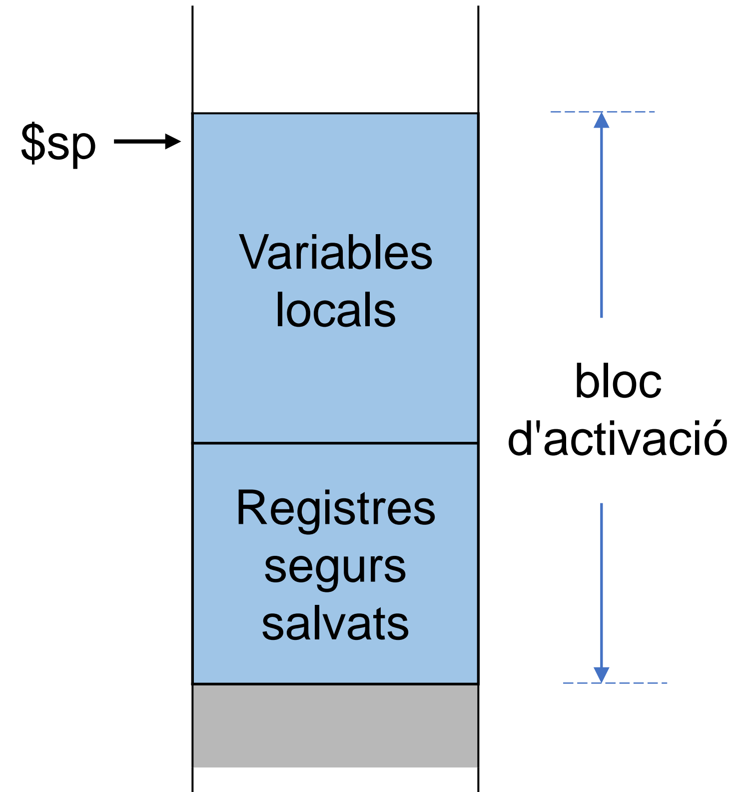
Mètode en dos passos:

1. Determinar quins registres segurs s'usaran
   - Identificant quines dades emmagatzemades en registres es generaran ABANS d'una crida a subrutina, i s'usaran DESPRÉS de la crida
2. Salvar i restaurar registres segurs
   - A l'inici de la subrutina (pròleg), salvar (*store*) el valor anterior dels registres segurs en el Bloc d'Activació
   - Al final de la subrutina (epíleg), restaurar (*load*) el valor original dels registres segurs

Estructura del Bloc d'Activació

- Variables locals
  - Posició: adreces + baixes
  - Ordre: igual que la declaració
  - Alineació: segons el seu tipus

- Registres segurs salvats
  - Posició: adreces + altes
  - Ordre: qualsevol
  - Alineació: múltiples de 4



Exemple: `multi`

- Traduir a MIPS:

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

Exemple: `multi`

- Traduir a MIPS:

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

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e

Exemple: `multi`

- Traduir a MIPS:

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

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e

Exemple: `multi`

- Traduir a MIPS:

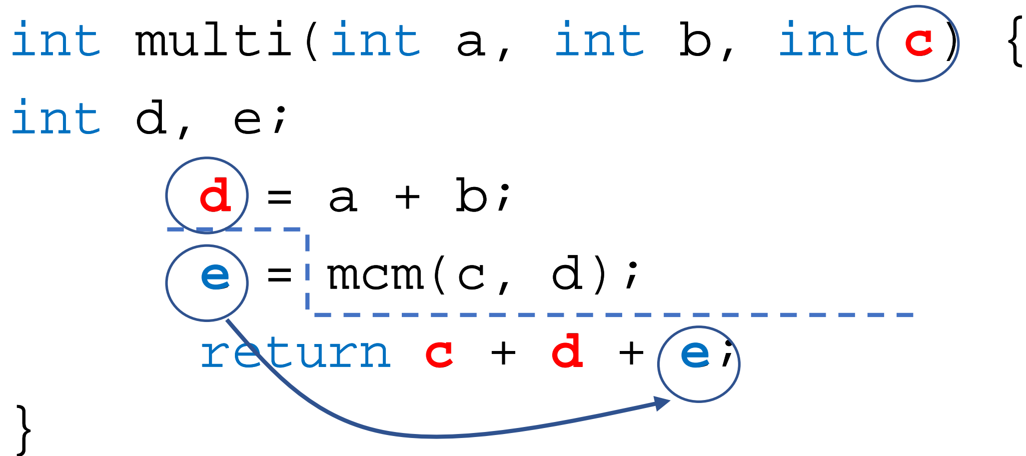
```
int multi(int a, int b, int c) {
int d, e;
        d = a + b;
        e = mcm(c, d);
        return c + d + e;
}
```

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e
3. Generades ABANS de la crida: **c, d**

Exemple: `multi`

- Traduir a MIPS:



- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e
3. Generades ABANS de la crida: c, d (però no **e**!)

Exemple: `multi`

- Traduir a MIPS:

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

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e
3. Generades ABANS de la crida: c, d

→ Copiarem c en $s0

→ La suma d s'escriurà en $s1

Exemple: `multi`

- Traduir a MIPS:

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

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e
3. Generades ABANS de la crida: c, d

→ Copiarem c en $s0

→ La suma d s'escriurà en $s1

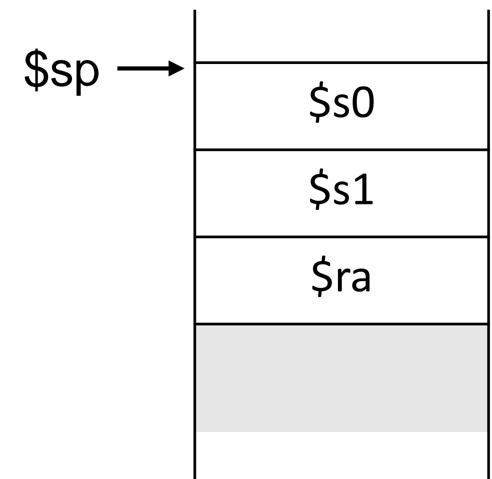
→ Salvarem $s0 i $s1 (i $ra) a la pila

Exemple: `multi`

- Traduir a MIPS:

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

Bloc d'Activació:



- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: a, b, c, d, e
2. Usades DESPRÉS de la crida: c, d, e
3. Generades ABANS de la crida: c, d

→ Copiarem c en $s0

→ La suma d s'escriurà en $s1

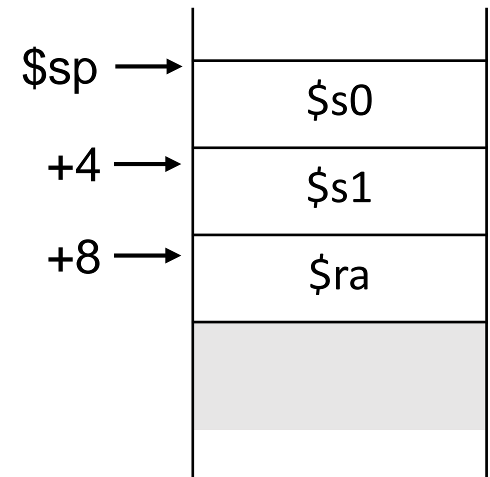
→ Salvarem $s0 i $s1 (i $ra) a la pila

Traduir `multi` a MIPS

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

Pròleg:

reservem espai a la pila
salvem registres segurs
copiem c en $s0



Traduir `multi` a MIPS

```
multi:
      addiu  $sp, $sp, -12
      sw     $s0, 0($sp)
      sw     $s1, 4($sp)
      sw     $ra, 8($sp)
      move   $s0, $a2        # c
      addu   $s1, $a0, $a1   # d
```

```
int multi(int a, int b, int c) {
int d, e;

    d = a + b;

    e = mcm(c, d);

    return c + d + e;
```

Traduir `multi` a MIPS

```
multi:
        addiu  $sp, $sp, -12
        sw     $s0, 0($sp)
        sw     $s1, 4($sp)
        sw     $ra, 8($sp)
        move   $s0, $a2        # c
        addu   $s1, $a0, $a1   # d
        move   $a0, $s0
        move   $a1, $s1
        jal    mcm
```

```
int multi(int a, int b, int c) {
int d, e;

    d = a + b;

    e = mcm(c, d);

    return c + d + e;
```

Traduir `multi` a MIPS

```
multi:
      addiu  $sp, $sp, -12
      sw     $s0, 0($sp)
      sw     $s1, 4($sp)
      sw     $ra, 8($sp)
      move   $s0, $a2        # c
      addu   $s1, $a0, $a1   # d
      move   $a0, $s0
      move   $a1, $s1
      jal    mcm
      addu   $t1, $s0, $s1
```

```
int multi(int a, int b, int c) {
int d, e;

    d = a + b;

    e = mcm(c, d);

    return c + d + e;
```

Traduir `multi` a MIPS

```
multi:
        addiu   $sp, $sp, -12
        sw      $s0, 0($sp)
        sw      $s1, 4($sp)
        sw      $ra, 8($sp)
        move    $s0, $a2        # c
        addu    $s1, $a0, $a1   # d
        move    $a0, $s0
        move    $a1, $s1
        jal     mcm
        addu    $t1, $s0, $s1
        addu    $v0, $t1, $v0
```

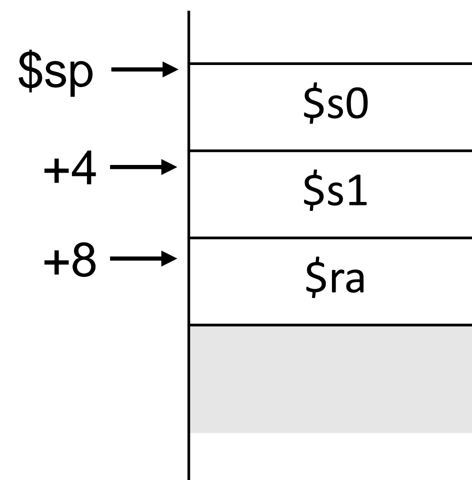
```
int multi(int a, int b, int c) {
int d, e;

    d = a + b;

    e = mcm(c, d);

    return c + d + e;
```

Traduir `multi` a MIPS

```
multi:
      addiu  $sp, $sp, -12
      sw     $s0, 0($sp)
      sw     $s1, 4($sp)
      sw     $ra, 8($sp)
      move   $s0, $a2         # c
      addu   $s1, $a0, $a1    # d
      move   $a0, $s0
      move   $a1, $s1
      jal    mcm
      addu   $t1, $s0, $s1
      addu   $v0, $t1, $v0
      lw     $s0, 0($sp)
      lw     $s1, 4($sp)
      lw     $ra, 8($sp)
```

Epíleg :

restaurem registres segurs



Traduir `multi` a MIPS

```
multi:
      addiu  $sp, $sp, -12
      sw     $s0, 0($sp)
      sw     $s1, 4($sp)
      sw     $ra, 8($sp)
      move   $s0, $a2        # c
      addu   $s1, $a0, $a1   # d
      move   $a0, $s0
      move   $a1, $s1
      jal    mcm
      addu   $t1, $s0, $s1
      addu   $v0, $t1, $v0
      lw     $s0, 0($sp)
      lw     $s1, 4($sp)
      lw     $ra, 8($sp)
      addiu  $sp, $sp, 12
      jr     $ra
```

Epíleg :

restaurem registres segurs

alliberem espai a la pila

saltem a l'adreça de retorn

Exemple

- Traduir a MIPS:

```
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];
}
```

- Hi ha 2 crides a la mateixa línia
  - les reescrivim desdoblades

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

- Hi ha 2 crides a la mateixa línia
  - les reescrivim desdoblades

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: *a*, *b, c, d, e, res_f, res_g*

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: *a*, *b, c, d, e, res_f, res_g,* (però no *q* !!)

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: *a*, *b, c, d, e, res_f, res_g*
2. Usades DESPRÉS i generades ABANS de la crida a *f*: *c, d*

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: *a*, *b, c, d, e, res_f, res_g*
2. Usades DESPRÉS i generades ABANS de la crida a *f*: *c, d*
3. Usades DESPRÉS i generades ABANS de la crida a *g*: *c, d, res_f*

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

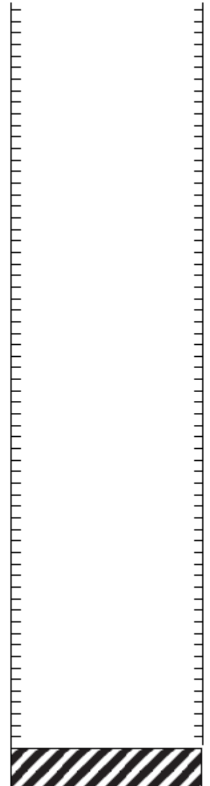
- Identifiquem registres segurs a usar (a més a més de $ra)

1. Dades en registres: *a*, *b, c, d, e, res_f, res_g*
2. Usades DESPRÉS i generades ABANS de la crida a *f*: *c, d*
3. Usades DESPRÉS i generades ABANS de la crida a *g*: *c, d, res_f*

(guardarem *c, d, res_f* als registres segurs $s0, $s1, $s2)

Exemple

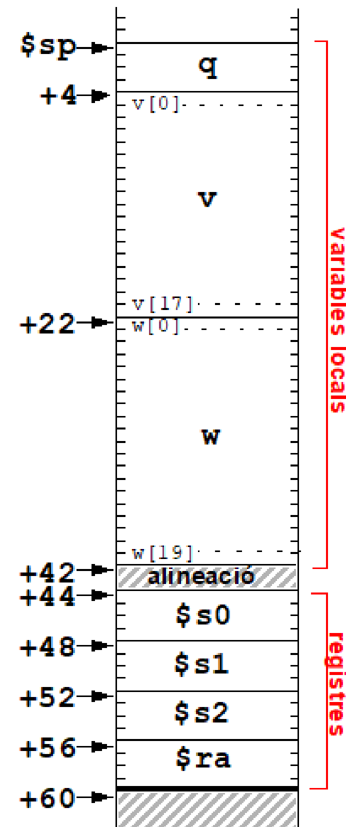
```
char exemple(int a, int b, int c)
{
        int d, e, q;
        char v[18], w[20];
        d = a + b;

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```



- Determinem el Bloc d’Activació

Exemple

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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```



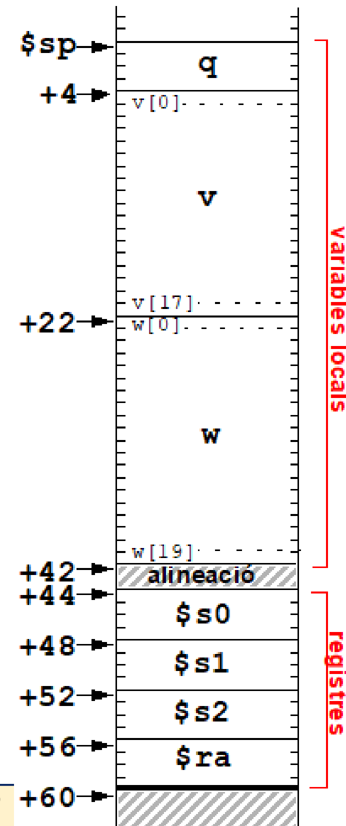
- Determinem el Bloc d'Activació
  - Variables locals: *q* (4 bytes), *v* (18 bytes), *w* (20 bytes)
  - Còpia dels valors originals dels registres segurs: $s0, $s1, $s2, $ra

Exemple

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

Pròleg:

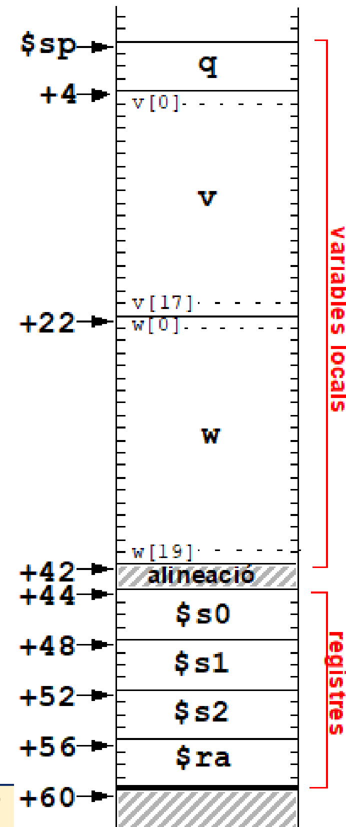
Reservem espai a la pila
Salvem registres segurs
Copiem c en $s0




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

     res_f = f(d, &q);

     res_g = g(v, w);

     e = res_f + res_g;

     return v[e + q] + w[d + c];
}
```

Exemple

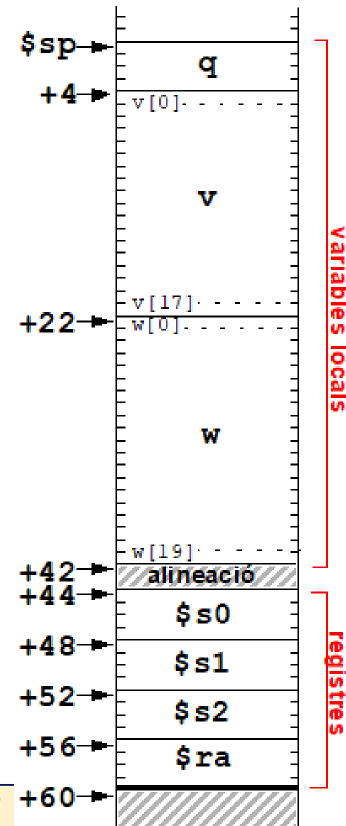
```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1   # d
```




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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

Exemple

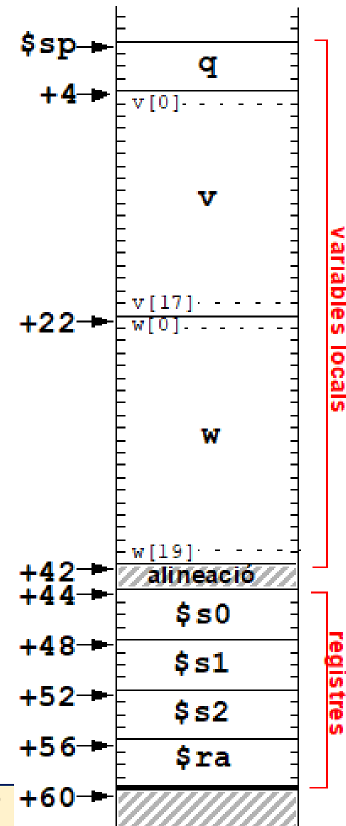
```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0    # res_f
```




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

        res_f = f(d, &q);

        res_g = g(v, w);

        e = res_f + res_g;

        return v[e + q] + w[d + c];
}
```

Exemple

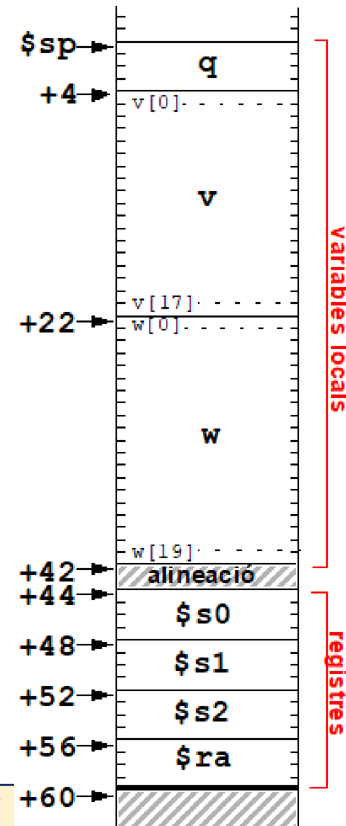
```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2       # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0   # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
```




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

    res_f = f(d, &q);

    res_g = g(v, w);

    e = res_f + res_g;

    return v[e + q] + w[d + c];
}
```

Exemple

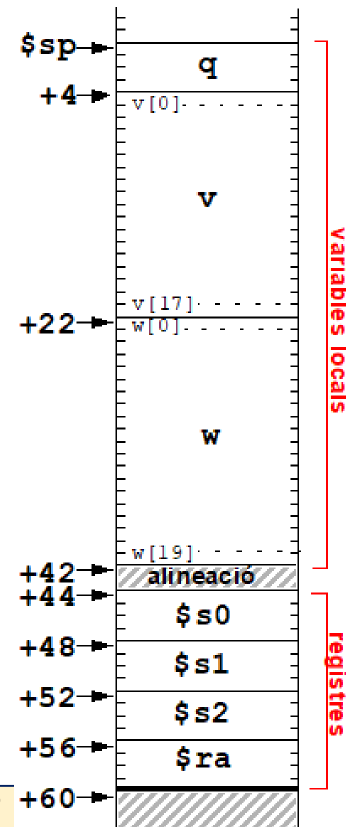
```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0   # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
    addu    $t0, $s2, $v0
```




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

    res_f = f(d, &q);

    res_g = g(v, w);

    e = res_f + res_g;

    return v[e + q] + w[d + c];
}
```

Exemple

```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2       # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0   # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
    addu    $t0, $s2, $v0
    lw      $t1, 0($sp)    # q
    addu    $t1, $t0, $t1
    addu    $t1, $t1, $sp
    lb      $t2, 4($t1)
```

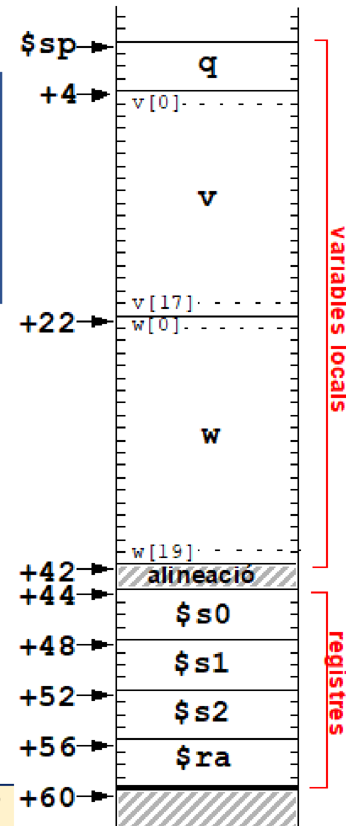



```
char exemple(int a, int b, int c)
{
    int d, e, q;
    char v[18], w[20];
    d = a + b;

    res_f = f(d, &q);

    res_g = g(v, w);

    e = res_f + res_g;

    return v[e + q] + w[d + c];
}
```

Exemple

```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0   # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
    addu    $t0, $s2, $v0
    lw      $t1, 0($sp)     # q
    addu    $t1, $t0, $t1
    addu    $t1, $t1, $sp
    lb      $t2, 4($t1)
```

```
addu    $t1, $s1, $s0
addu    $t1, $t1, $sp
lb      $t3, 22($t1)
addu    $v0, $t2, $t3
```



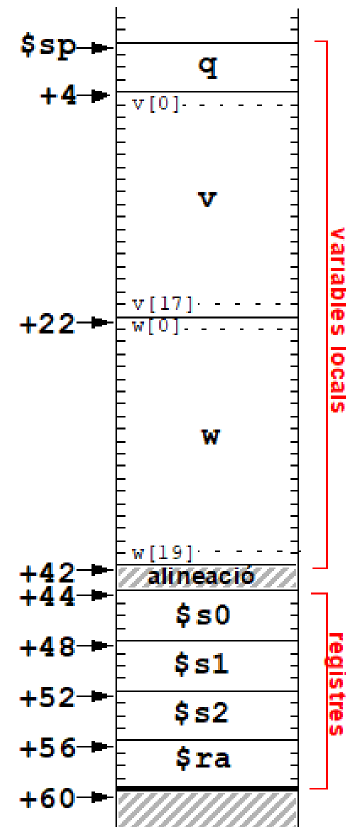

```
char exemple(int a, int b, int c)
{
    int d, e, q;
    char v[18], w[20];
    d = a + b;

    res_f = f(d, &q);

    res_g = g(v, w);

    e = res_f + res_g;

    return v[e + q] + w[d + c];
}
```

Exemple

```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0    # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
    addu    $t0, $s2, $v0
    lw      $t1, 0($sp)     # q
    addu    $t1, $t0, $t1
    addu    $t1, $t1, $sp
    lb      $t2, 4($t1)
    addu    $t1, $s1, $s0
    addu    $t1, $t1, $sp
    lb      $t3, 22($t1)
    addu    $v0, $t2, $t3
    lw      $s0, 44($sp)
    lw      $s1, 48($sp)
    lw      $s2, 52($sp)
    lw      $ra, 56($sp)
```

Epíleg:

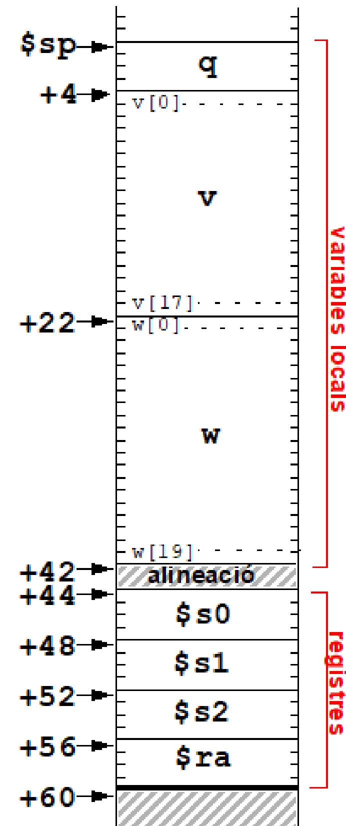
Restaurem registres segurs



Exemple

```
exemple:
    addiu   $sp, $sp, -60
    sw      $s0, 44($sp)
    sw      $s1, 48($sp)
    sw      $s2, 52($sp)
    sw      $ra, 56($sp)
    move    $s0, $a2        # c
    addu    $s1, $a0, $a1 # d
    move    $a0, $s1
    move    $a1, $sp
    jal     f
    move    $s2, $v0   # res_f
    addiu   $a0, $sp, 4
    addiu   $a1, $sp, 22
    jal     g
    addu    $t0, $s2, $v0
    lw      $t1, 0($sp)    # q
    addu    $t1, $t0, $t1
    addu    $t1, $t1, $sp
    lb      $t2, 4($t1)
```

```
    addu    $t1, $s1, $s0
    addu    $t1, $t1, $sp
    lb      $t3, 22($t1)
    addu    $v0, $t2, $t3
    lw      $s0, 44($sp)
    lw      $s1, 48($sp)
    lw      $s2, 52($sp)
    lw      $ra, 56($sp)
    addiu   $sp, $sp, 60
    jr      $ra
```



Epíleg:

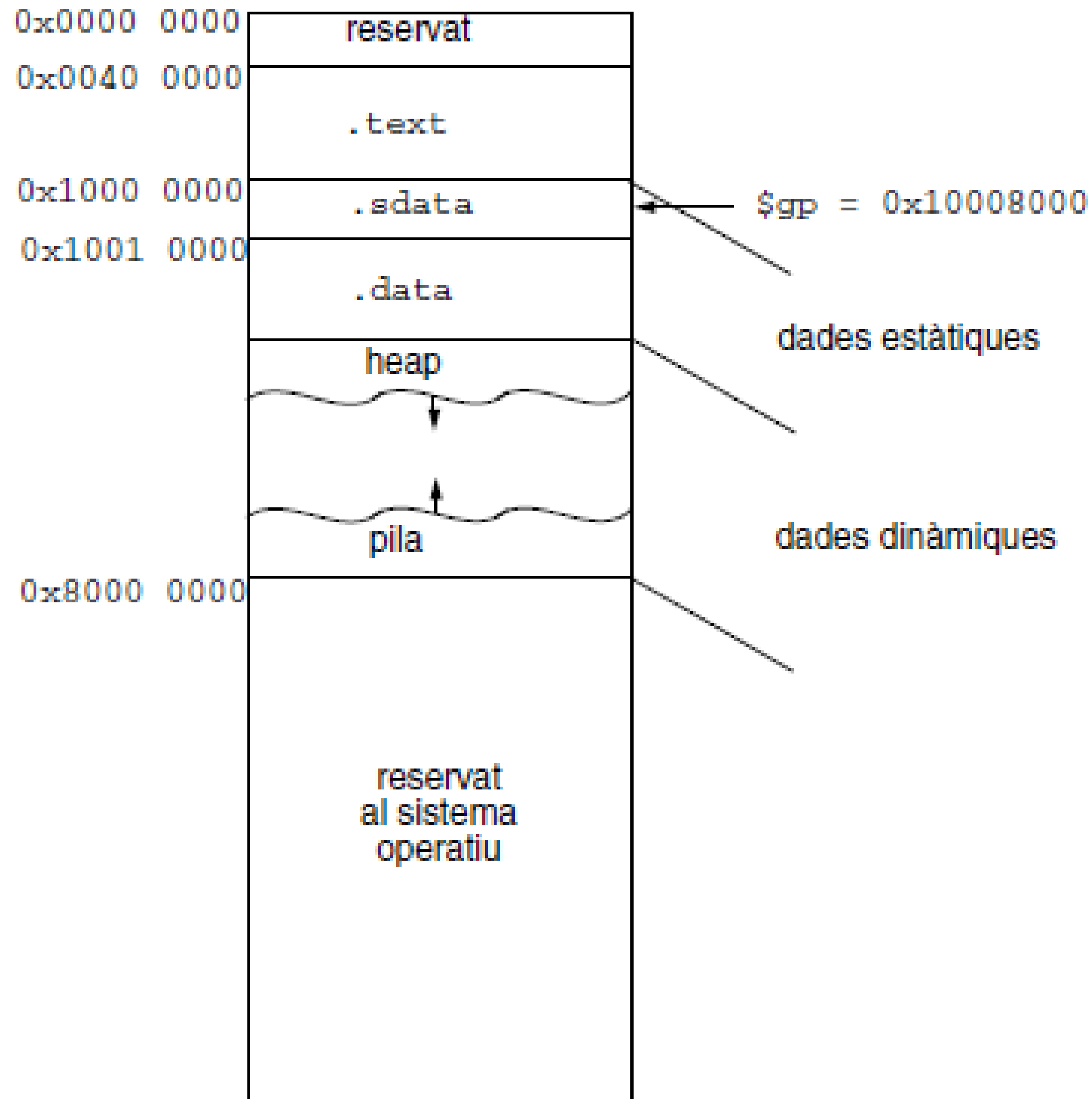
Restaurem registres segurs

Alliberem espai a la pila

Saltem a $ra

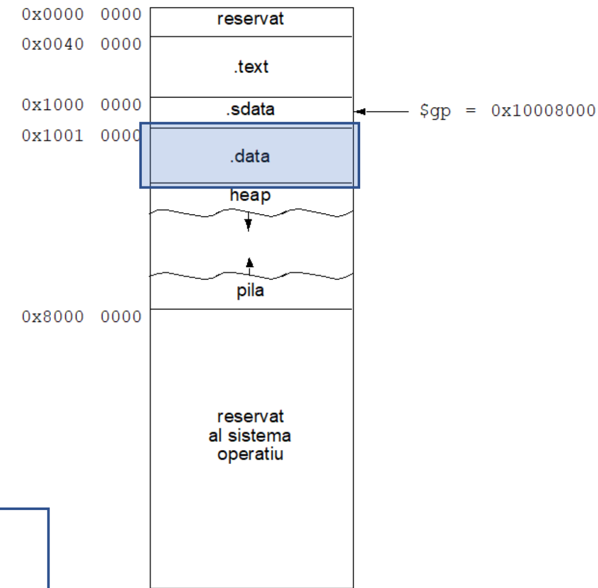
Estructura de la memòria

Estructura de la memòria en MIPS



Emmagatzemament estàtic

- La secció `.data`
  - Mida fixa per a cada programa
  - Guarda variables globals, en C
  - Ocupen el mateix lloc durant tota l'execució



C:

```c
int a[2] = {8, 7};
short b = -7;

int main(void){
        ...
}
```

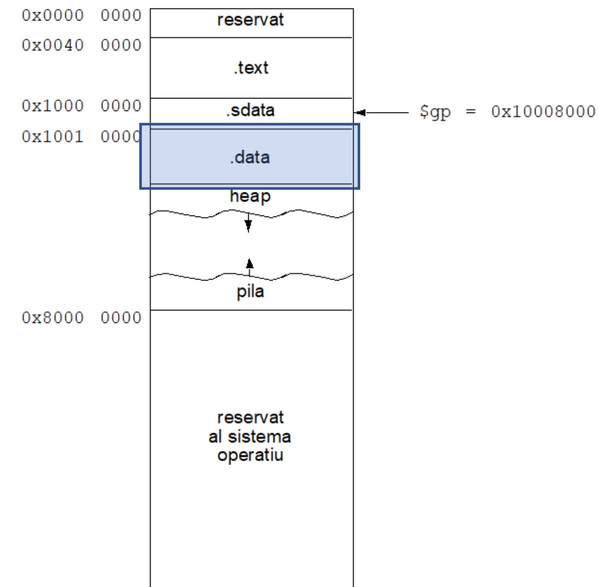
MIPS:

```
        .data
a:      .word 8, 7
b:      .half -7

        .text
main:
        ...
```

Emmagatzemament estàtic

- Accés a variables globals de `.data`
  - Requereix carregar l'adreça en un registre, i usar 3 instruccions

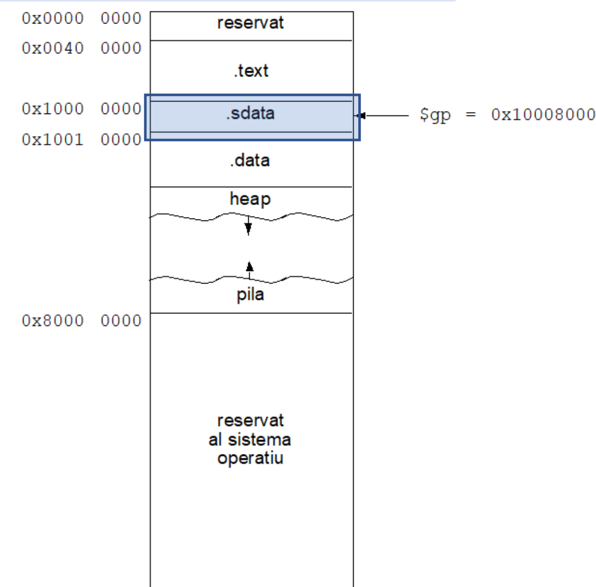
```
la      $t0, etiqueta_var_global
lw      $t0, 0($t0)
```



Emmagatzemament estàtic

- Accés a variables globals de `.data`
  - Requereix carregar l'adreça en un registre, i usar 3 instruccions

```
la      $t0, etiqueta_var_global
lw      $t0, 0($t0)
```



- La secció `.sdata` (*small data*)
  - Mida = 2^{16} bytes
  - Guarda variables globals, en C
  - $gp (*global pointer*) apunta al mig de `.sdata`
  - Accés amb 1 instrucció (offsets < 2^{15} del $gp)

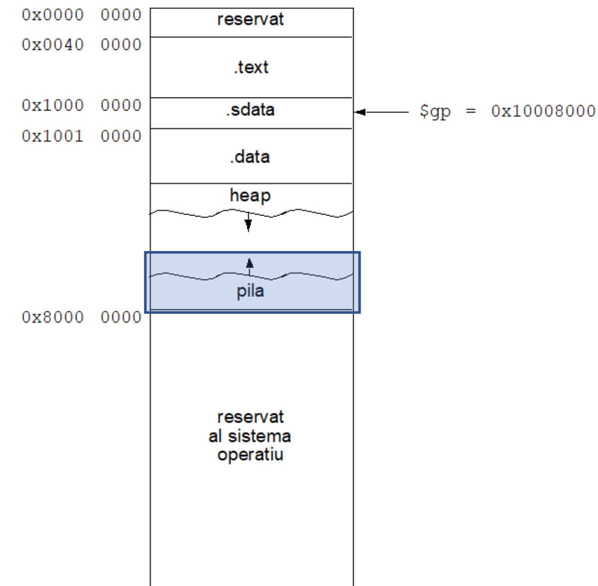
```
lw  $t0, offset_var_global($gp)
```

Emmagatzemament dinàmic

Dinàmic: "la variable ocupa temporalment un lloc en memòria "

- La secció *Pila*
  - Guarda variables locals
  - Creix dinàmicament cap a adreces baixes
  - Reserva/alliberament ordenats a l'inici/final de les subrutines

```
subrutina:
    addiu $sp, $sp, -mida_bloc

    ...

    addiu $sp, $sp, +mida_bloc
    jr    $ra
```

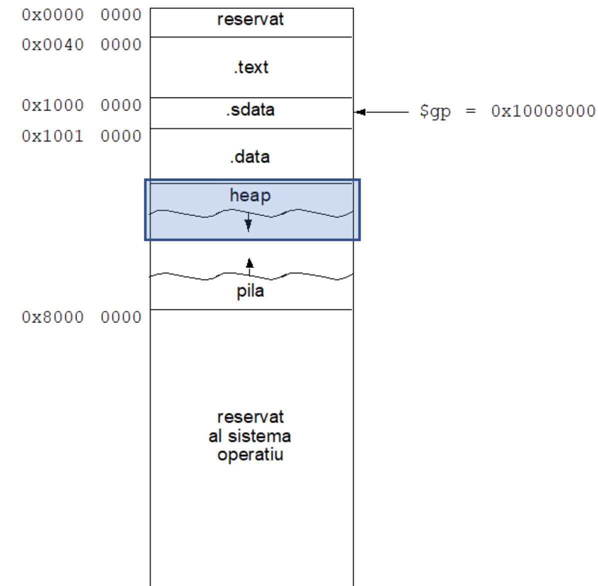


Emmagatzemament dinàmic

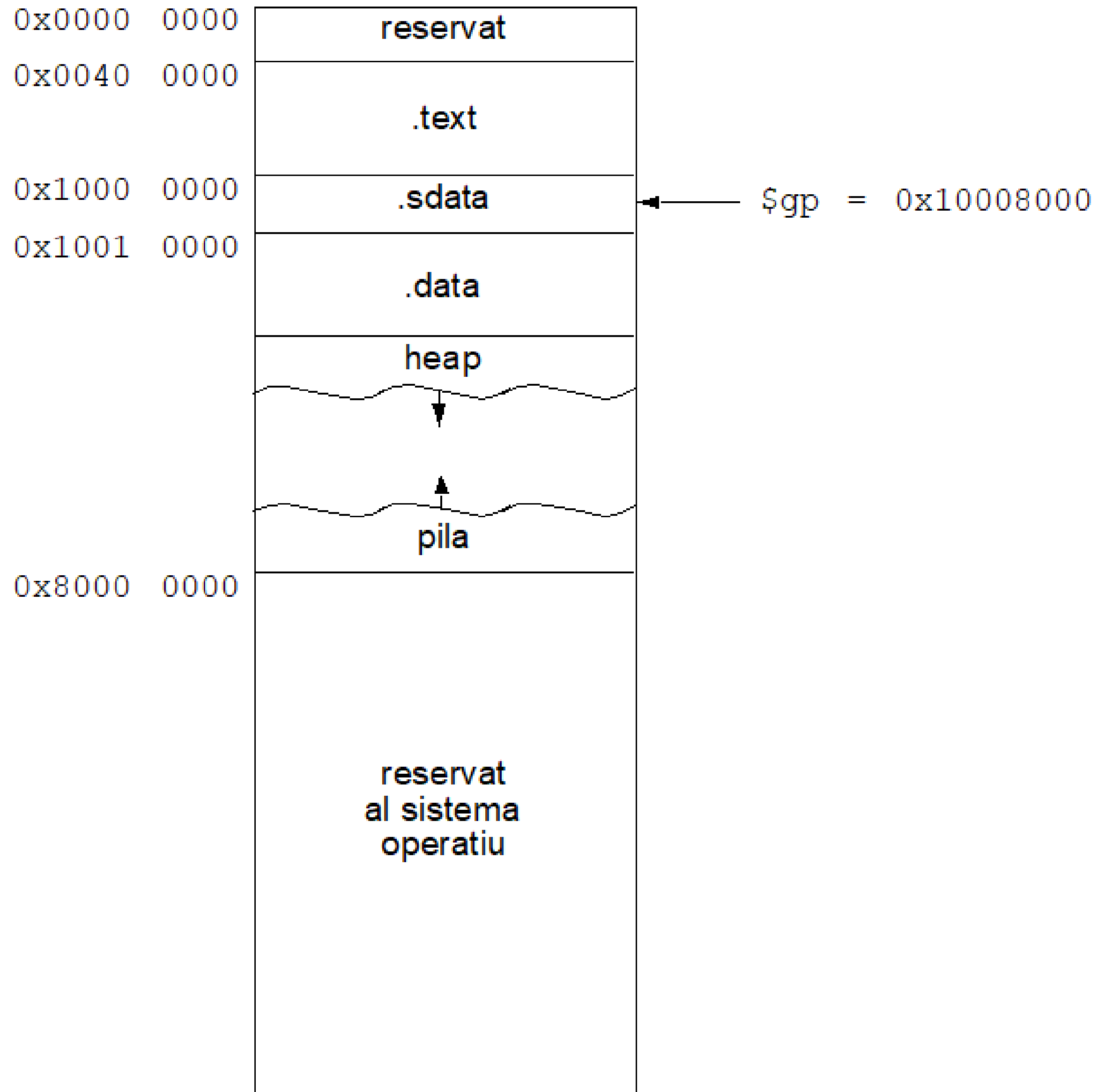
Dinàmic: "la variable ocupa temporalment un lloc en memòria"

- La secció *Heap*
  - Guarda variables que es creen i destrueixen explícitament
  - Creix dinàmicament cap a adreces altes
  - Reserva/alliberament invocant el SO amb `malloc()` i `free()`

```
int main (void) {
        int *ptr;
        ...
        ptr = malloc(400);
        ...
        free (ptr);
        ...
}
```



Estructura de la memòria



On s'emmagatzemen les diferents variables?

```
int gvec[100];
int *pvec;

int f()
{
        int lvec[100];
        ...
        pvec = malloc(400);
        ...
        pvec[10] = gvec[10] + lvec[10];

}

int g()
{
        ...
        free(pvec);
        ...
}
```

On s'emmagatzemen les diferents variables?

```
int gvec[100];
int *pvec;

int f()
{
        int lvec[100];
        ...
        pvec = malloc(400);
        ...
        pvec[10] = gvec[10] + lvec[10];

}

int g()
{
        ...
        free(pvec);
        ...
}
```

globals: dades estàtiques

On s'emmagatzemen les diferents variables?

```
int gvec[100];
int *pvec;

int f()
{
        int lvec[100];        ──────────► local: dades dinàmiques
                                            (reservem espai a la pila)
        ...
        pvec = malloc(400);
        ...
        pvec[10] = gvec[10] + lvec[10];
        ...
} ────────────────────────────────────►      (alliberem espai a la pila)

int g()
{
        ...
        free(pvec);
        ...
}
```

On s'emmagatzemen les diferents variables?

```
int gvec[100];
int *pvec;

int f()
{
        int lvec[100];
        ...
        pvec = malloc(400);         → global: dades dinàmiques
        ...                               (reservem espai al heap)
        pvec[10] = gvec[10] + lvec[10];

}

int g()
{
        ...
        free(pvec);                 → (alliberem espai al heap)
        ...
}
```

Compilació, assemblatge, enllaçat i càrrega

Compilació de programes

1. Compilació
   - El compilador tradueix de C a MIPS

Compilació de programes

1. Compilació
   - El compilador tradueix de C a MIPS

2. Assemblatge
   - L'assemblador tradueix de MIPS a codi màquina

Compilació de programes

1. Compilació
   - El compilador tradueix de C a MIPS

2. Assemblatge
   - L'assemblador tradueix de MIPS a codi màquina

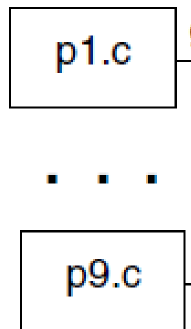
3. Enllaçat
   - L'enllaçador (linker) combina múltiples fitxers amb codi màquina en un únic fitxer executable

Compilació de programes

1. Compilació
   - El compilador tradueix de C a MIPS
2. Assemblatge
   - L'assemblador tradueix de MIPS a codi màquina
3. Enllaçat
   - L'enllaçador (linker) combina múltiples fitxers amb codi màquina en un únic fitxer executable
4. Càrrega
   - El carregador (loader) llegeix el fitxer executable i el carrega a memòria per a ser executat

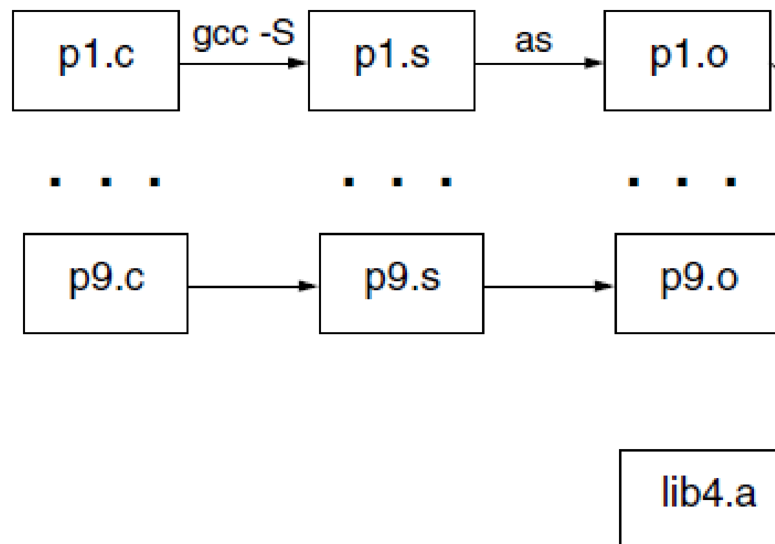
Compilació separada

- Estructurar el codi en diversos mòduls
  - Facilita la gestió de projectes complexos
  - Fomenta la reutilització de codi
  - Recompilació parcial quan es modifica un mòdul



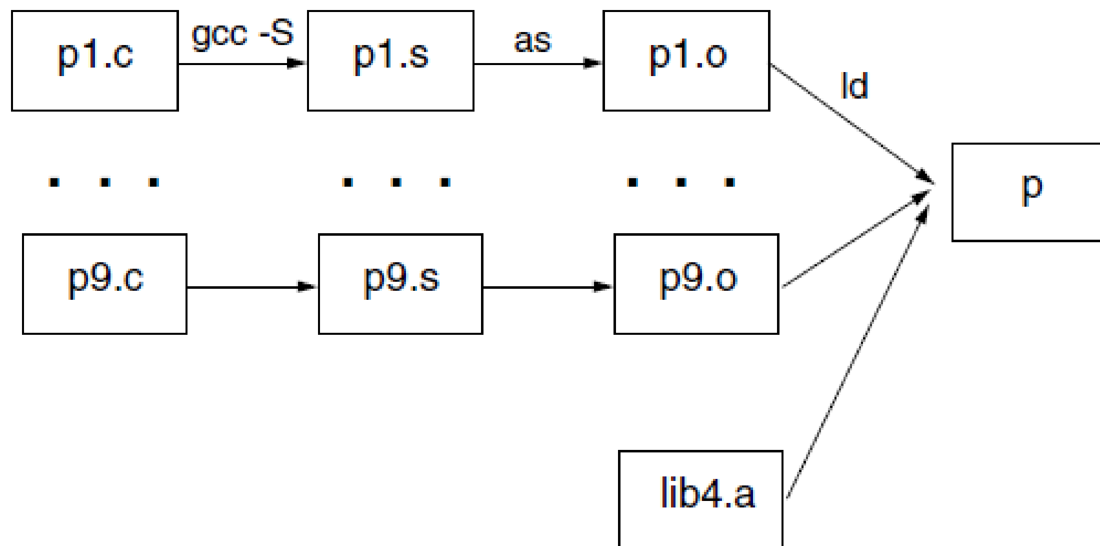
Compilació separada

- Estructurar el codi en diversos mòduls
  - Facilita la gestió de projectes complexos
  - Fomenta la reutilització de codi
  - Recompilació parcial quan es modifica un mòdul
- Els mòduls (fitxers) es compilen i assemblen per separat



Compilació separada

- Estructurar el codi en diversos mòduls
  - Facilita la gestió de projectes complexos
  - Fomenta la reutilització de codi
  - Recompilació parcial quan es modifica un mòdul
- Els mòduls (fitxers) es compilen i assemblen per separat
- Els diferents mòduls s'enllacen per generar l'executable

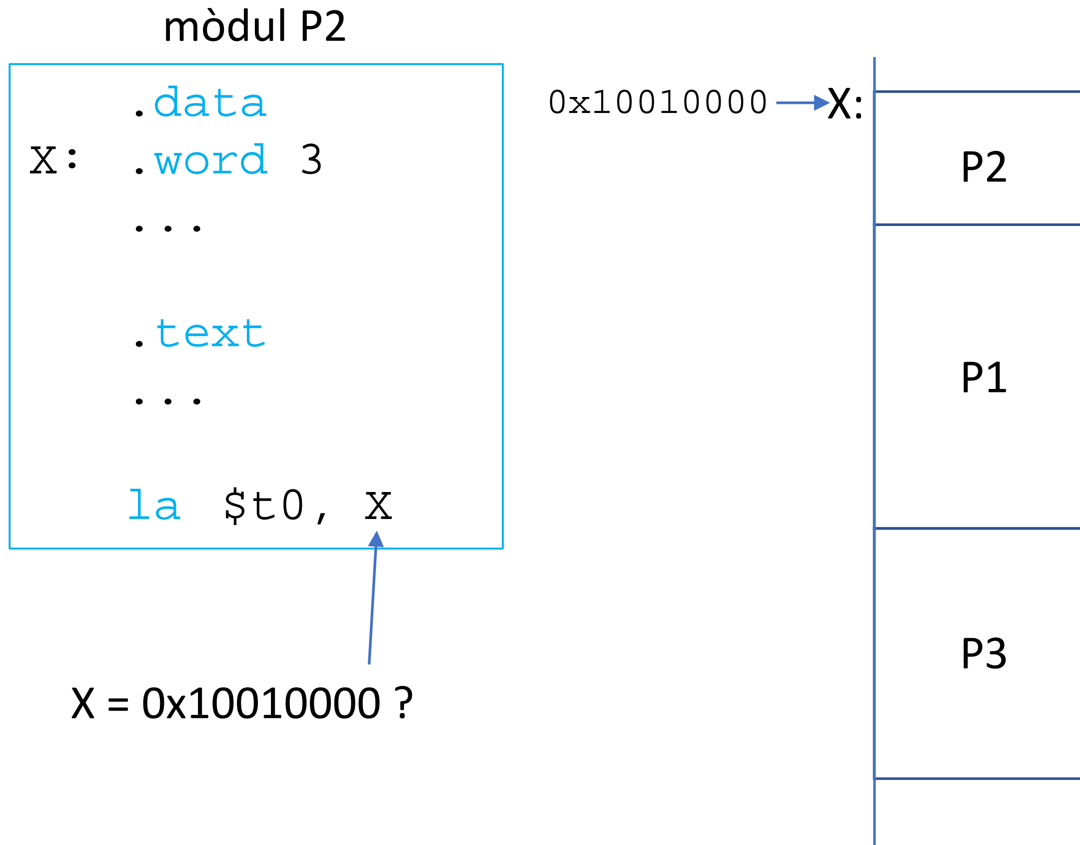


Problema: reubicació

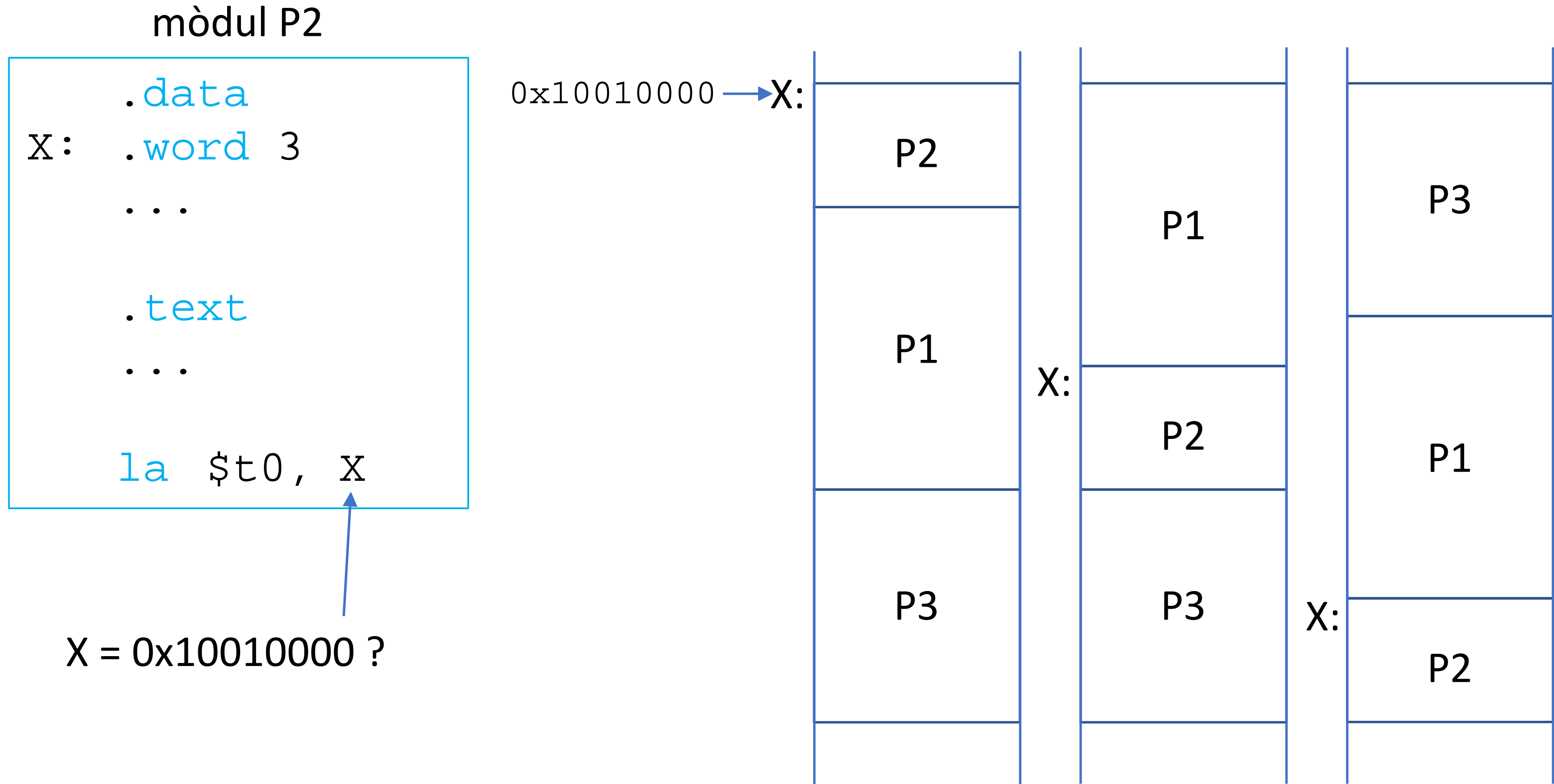
mòdul P2

```
        .data
X:      .word 3


        .text
        ...

        la  $t0, X
```

Problema: reubicació



Problema: reubicació



De moment, hi codificarem un simple offset:
secció = "data", offset = 0
Quan s'enllacin tots els mòduls sabrem l'adreça definitiva i REUBICAREM la instrucció

Problema: referències creuades

mòdul P2

```
      .data
X:    .word 3
      ...

      .text
      ...

      la  $t0, X
      ...
      la  $t1, Y
```

Y no és una etiqueta definida en P2
(No és un error, podria estar definida en un altre mòdul)

Problema: referències creuades

mòdul P2

```
     .data
X:   .word 3
     ...

     .text
     ...

     la  $t0, X
     ...
     la  $t1, Y
```

Y no és una etiqueta definida en P2
(No és un error, podria estar definida en un altre mòdul)

mòdul P3

```
     .data

Y:   .word 3
     ...

     .text
     ...

     la  $t2, Y
```

Però per defecte, les etiquetes són LOCALS a cada mòdul

Problema: referències creuades

mòdul P2

```
      .data
X:    .word 3
      ...

      .text
      ...

      la  $t0, X
      ...
      la  $t1, Y
```

Y no és una etiqueta definida en P2
(No és un error, podria estar definida en un altre mòdul)

mòdul P3

```
      .data
      .globl Y
Y:    .word 3
      ...

      .text
      ...

      la  $t2, Y
```

`.globl` la fa global a tots els mòduls

Però per defecte, les etiquetes són LOCALS a cada mòdul

Assemblatge en 2 passades

Primera passada

- Expandir macros
- Generar una taula de símbols amb l'adreça de cada etiqueta
  - No són adreces definitives, són relatives (offsets) a la secció on està definida l'etiqueta
- Generar una taula de símbols globals per a les etiquetes declarades amb la directiva `.globl`

Assemblatge en 2 passades

Segona passada: codificar les instruccions

- Amb adreces relatives al PC (`beq`, `bne`): calcular la distància de salt usant l'offset guardat a la taula de símbols

Assemblatge en 2 passades

Segona passada: codificar les instruccions

- Amb adreces relatives al PC (`beq`, `bne`): calcular la distància de salt usant l'offset guardat a la taula de símbols
- Amb adreces absolutes (`la`, `j`, `jal`): codificar-hi un zero provisional (les resoldrà l'enllaçador, en la *reubicació*)

Assemblatge en 2 passades

Segona passada: codificar les instruccions

- Amb adreces relatives al PC (`beq`, `bne`): calcular la distància de salt usant l'offset guardat a la taula de símbols

- Amb adreces absolutes (`la`, `j`, `jal`): codificar-hi un zero provisional (les resoldrà l'enllaçador, en la *reubicació*)
  - Si l'etiqueta està a la *taula de símbols*
    - Afegir la instrucció a una llista de reubicació, especificant posició, tipus, adreça provisional (l'offset que figura a la taula de símbols)

Assemblatge en 2 passades

Segona passada: codificar les instruccions

- Amb adreces relatives al PC (`beq`, `bne`): calcular la distància de salt usant l'offset guardat a la taula de símbols

- Amb adreces absolutes (`la`, `j`, `jal`): codificar-hi un zero provisional (les resoldrà l'enllaçador, en la *reubicació*)

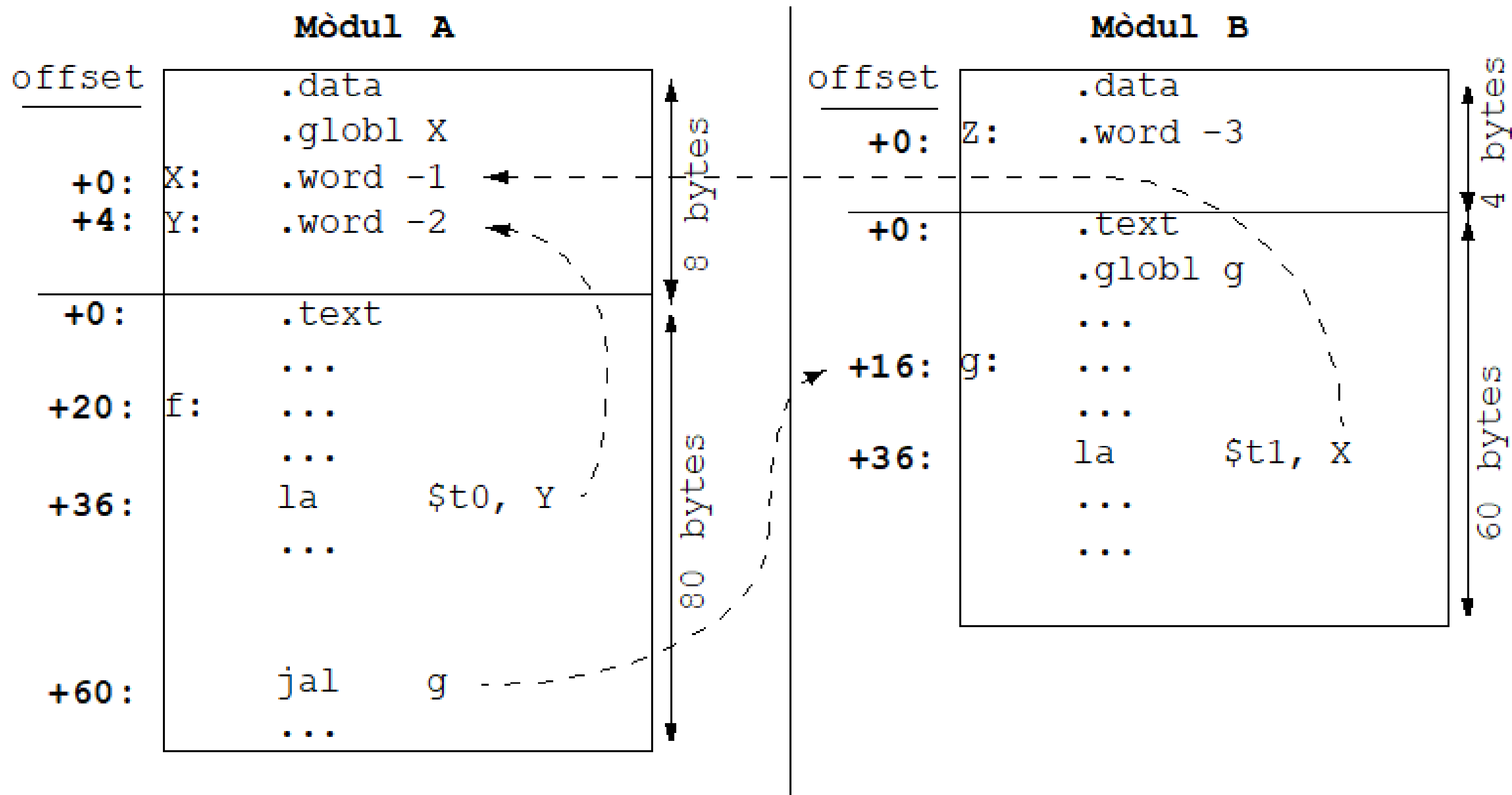
    - Si l'etiqueta està a la *taula de símbols*
        - Afegir la instrucció a una llista de reubicació, especificant posició, tipus, adreça provisional (l'offset que figura a la taula de símbols)

    - Sinó (l'etiqueta potser està definida a la *taula de símbols globals* d'un altre mòdul)
        - Afegir la instrucció a una llista de referències externes no resoltes

Assemblatge

- Fitxer objecte resultant, en UNIX:
  - Capçalera (una mena d'índex del fitxer)
  - Codi màquina (secció .text)
  - Dades estàtiques globals (secció .data)
  - Llista de reubicació
    - Posició de la instrucció, tipus i adreça provisional
  - Taula de símbols globals
  - Llista de referències no resoltes
  - Informació de depuració (p.ex. núms de línia del codi font)

Assemblatge. Exemple amb 2 mòduls



Exemple: Assemblatge del mòdul A

Programa A

offset		
		.data
		.globl X
+0:	X:	.word -1
+4:	Y:	.word -2
+0:		.text
		...
+20:	f:	...
		...
+36:		la $t0, Y
		...
+60:		jal g
		...

1.- Taula de símbols (TS)

- labels X (.data + 0), Y (.data + 4), f (.text + 20)

Exemple: Assemblatge del mòdul A

Programa A

offset	label	code
		.data
		.globl X
+0:	X:	.word -1
+4:	Y:	.word -2
+0:		.text
		...
+20:	f:	...
		...
+36:		la $t0, Y
		...
+60:		jal g
		...

1.- Taula de símbols (TS)

- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)

- label X (.data + 0)

Exemple: Assemblatge del mòdul A

Programa A

```
offset
                .data
                .globl   X
 +0:    X:      .word  -1
 +4:    Y:      .word  -2

 +0:            .text
                ...
 +20:   f:      ...
                ...
 +36:           la       $t0, Y
                ...

 +60:           jal      g
                ...
```

1.- Taula de símbols (TS)

- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)

- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)

- instrucció tipus **`la`** (.text + 36): @ = .data + 4

Exemple: Assemblatge del mòdul A

Programa A

offset		
		.data
		.globl X
+0:	X:	.word -1
+4:	Y:	.word -2
+0:		.text
		...
+20:	f:	...
		...
+36:		la $t0, Y
		...
+60:		jal g
		...

1.- Taula de símbols (TS)

- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)

- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)

- instrucció tipus `la` (.text + 36): @ = .data + 4

4.- Llista referències no-resoltes (no estan en TS)

- instrucció tipus `jal` (.text + 60): label = "g"

Exemple: Assemblatge del mòdul B

Programa A

```
offset
          .data
          .globl   X
+0:   X:  .word -1
+4:   Y:  .word -2

+0:       .text
          ...
+20:  f:  ...
          ...
+36:      la      $t0, Y
          ...

+60:      jal     g
          ...
```

Programa B

```
offset
+0:   Z:  .word -3   (.data)

+0:       .text
          .globl g
          ...
+16:  g:  ...
          ...
+36:      la      $t1, X
          ...
          ...
```

1.- Taula de símbols (TS)
- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)
- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)
- instrucció tipus `la` (.text + 36): @ = .data + 4

4.- Llista referències no-resoltes (no estan en TS)
- instrucció tipus `jal` (.text + 60): label = "g"

1.- Taula de símbols (TS)
- labels Z (.data + 0), g (.text + 16)

Exemple: Assemblatge del mòdul B

Programa A

offset	label	code
		.data
		.globl X
+0:	X:	.word -1
+4:	Y:	.word -2
+0:		.text
		...
+20:	f:	...
		...
+36:		la $t0, Y
		...
+60:		jal g
		...

Programa B

offset	label	code
		.data
+0:	Z:	.word -3
+0:		.text
		.globl g
		...
+16:	g:	...
		...
+36:		la $t1, X
		...
		...

1.- Taula de símbols (TS)
- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)
- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)
- instrucció tipus `la` (.text + 36): @ = .data + 4

4.- Llista referències no-resoltes (no estan en TS)
- instrucció tipus `jal` (.text + 60): label = "g"

1.- Taula de símbols (TS)
- labels Z (.data + 0), g (.text + 16)

2.- Taula de símbols globals (TS-globals)
- label = g (.text + 16)

Exemple: Assemblatge del mòdul B

Programa A

```
offset
                .data
                .globl  X
+0:     X:      .word -1
+4:     Y:      .word -2

+0:             .text
                ...
+20:    f:      ...
                ...
+36:            la      $t0, Y
                ...

+60:            jal     g
                ...
```

Programa B

```
offset
                .data
+0:     Z:      .word -3

+0:             .text
                .globl g
                ...
+16:    g:      ...
                ...
+36:            la      $t1, X
                ...
                ...
```

1.- Taula de símbols (TS)
- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)
- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)
- instrucció tipus `la` (.text + 36): @ = .data + 4

4.- Llista referències no-resoltes (no estan en TS)
- instrucció tipus `jal` (.text + 60): label = "g"

1.- Taula de símbols (TS)
- labels Z (.data + 0), g (.text + 16)

2.- Taula de símbols globals (TS-globals)
- label = g (.text + 16)

3.- Llista de reubicació (adreces en TS)
- (cap)

Exemple: Assemblatge del mòdul B

Programa A

```
offset
              .data
              .globl  X
+0:     X:    .word -1
+4:     Y:    .word -2

+0:           .text
              ...
+20:    f:    ...
              ...
+36:          la      $t0, Y
              ...

+60:          jal     g
              ...
```

Programa B

```
offset
              .data
+0:     Z:    .word -3

+0:           .text
              .globl g
              ...
+16:    g:    ...
              ...
+36:          la      $t1, X
              ...
              ...
```

1.- Taula de símbols (TS)

- labels X (.data + 0), Y (.data + 4), f (.text + 20)

2.- Taula de símbols globals (TS-globals)

- label X (.data + 0)

3.- Llista de reubicació (adreces en TS)

- instrucció tipus `la` (.text + 36): @ = .data + 4

4.- Llista referències no-resoltes (no estan en TS)

- instrucció tipus `jal` (.text + 60): label = "g"

1.- Taula de símbols (TS)

- labels Z (.data + 0), g (.text + 16)

2.- Taula de símbols globals (TS-globals)

- label = g (.text + 16)

3.- Llista de reubicació (adreces en TS)

- (cap)

4.- Llista referències no-resoltes (no estan en TS)

- instrucció tipus `la` (.text + 36): label = "X"

Enllaçat (o muntatge o "linkatge")

1. Buscar, en cascada, fitxers de biblioteca necessaris (del programa o del sistema)

Enllaçat (o muntatge o "linkatge")

1. Buscar, en cascada, fitxers de biblioteca necessaris (del programa o del sistema)
2. Emparellar referències no-resoltes amb definicions de símbols globals

Taula de símbols globals de A
- label X (.data + 0)

Taula de símbols globals de B
- label g (.text + 16)

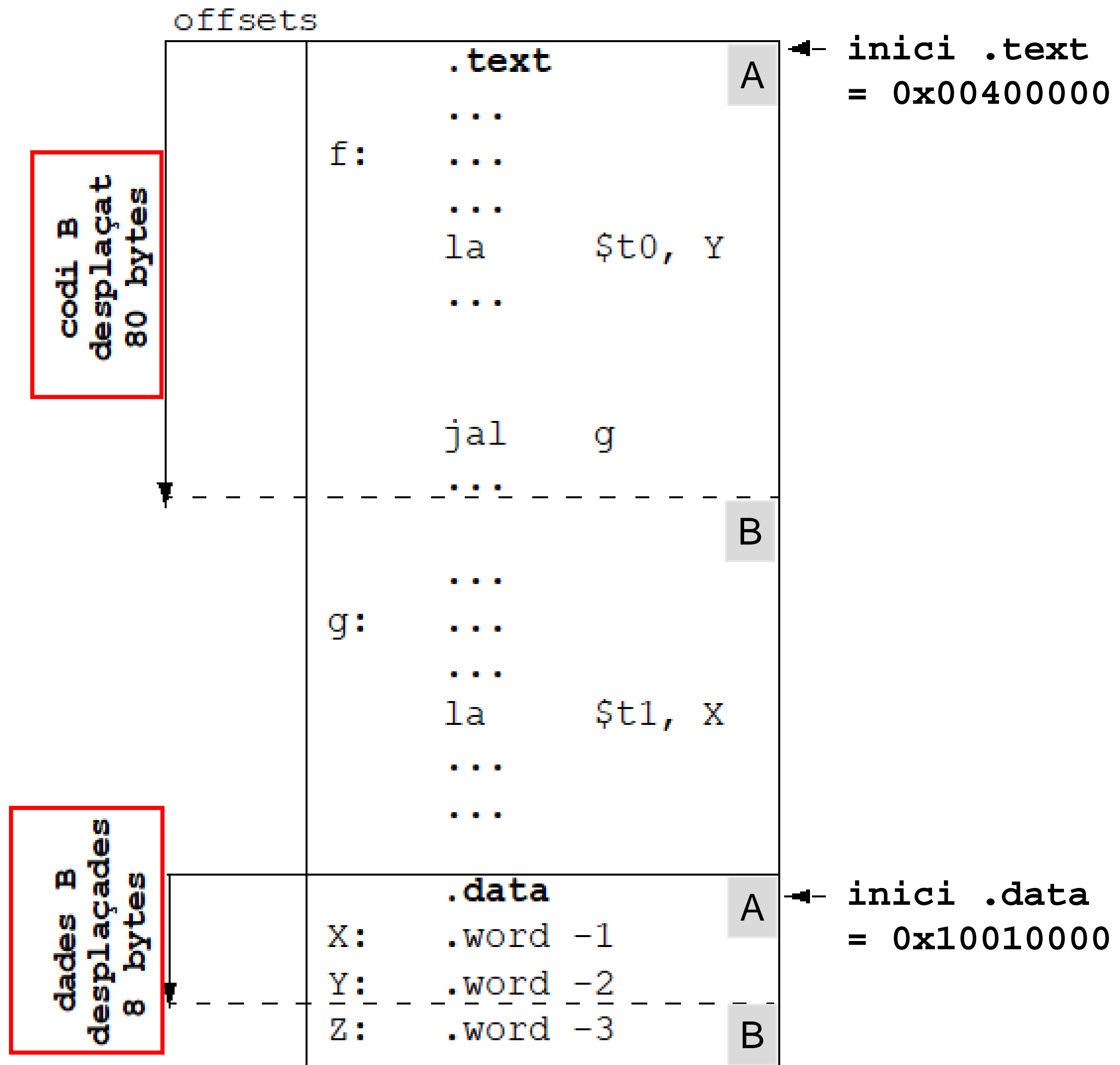
Referències no-resoltes de A
- instrucció tipus **`jal`** (.text + 60): label = "g"

Referències no-resoltes de B
- instrucció tipus **`la`** (.text + 36): label = "X"

- Si queda una referència sense emparellar, l'enllaçat falla: "Unresolved reference!"

Enllaçat (o muntatge o "linkatge")

3. Concatenar codi i dades del mòduls
   - Anotant el desplaçament de cada secció



Enllaçat (o muntatge o "linkatge")

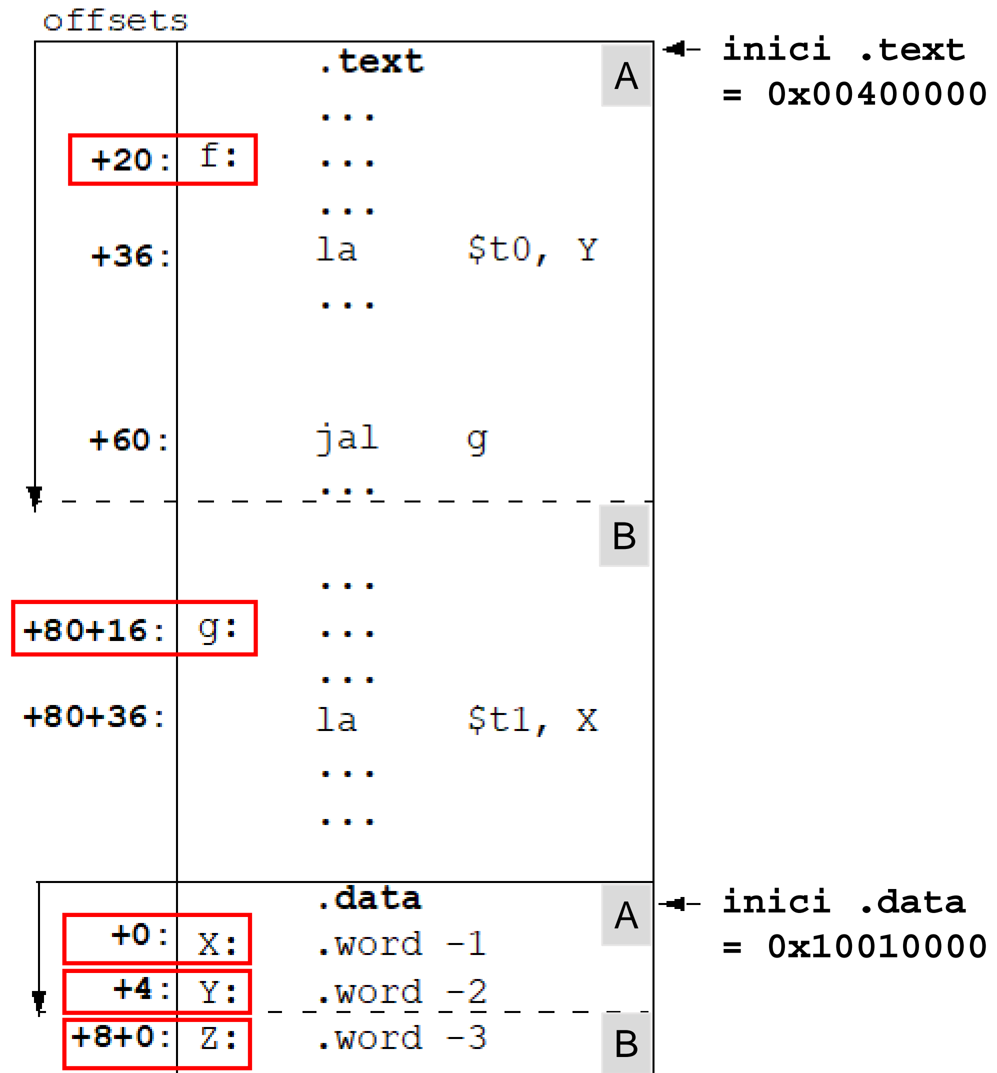
3. Concatenar codi i dades del mòduls
   - Anotant el desplaçament de cada secció
   - Assignar adreces definitives d'etiquetes a les taules de símbols:

TS:

```
f = 0x00400000 + 20
Y = 0x10010000 + 4
Z = 0x10010000 + 8 + 0
```

TS-globals:

```
g = 0x00400000 + 80 + 16
X = 0x10010000 + 0
```



Enllaçat (o muntatge o "linkatge")

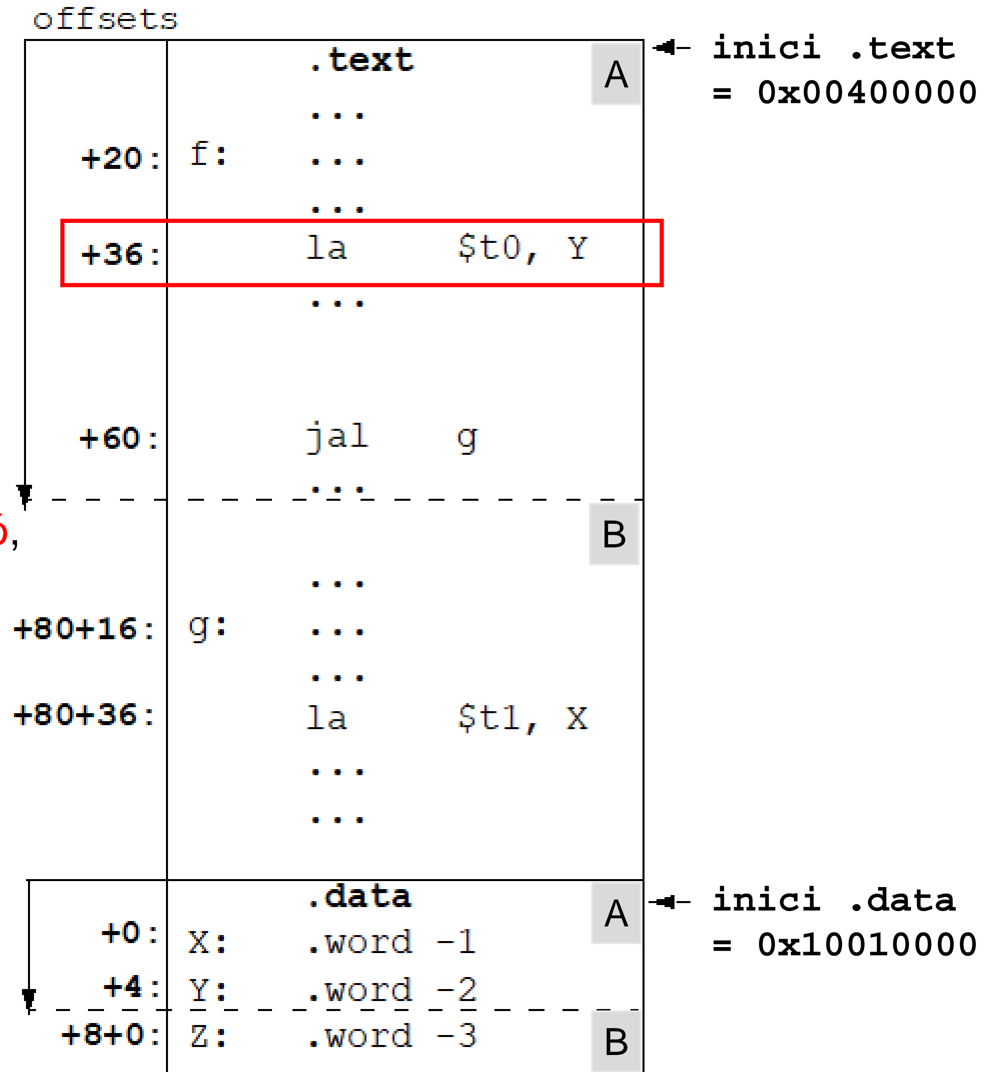
TS:

```
f = 0x00400000 + 20
Y = 0x10010000 + 4
Z = 0x10010000 + 8 + 0
```

TS-globals:

```
g = 0x00400000 + 80 + 16
X = 0x10010000 + 0
```

4. Reubicar instruccions
   - Corregir les de la Llista de Reubicació, amb les adreces definitives:
     - instrucció tipus **la** (.text + 36): @ = .data + 4



Enllaçat (o muntatge o "linkatge")

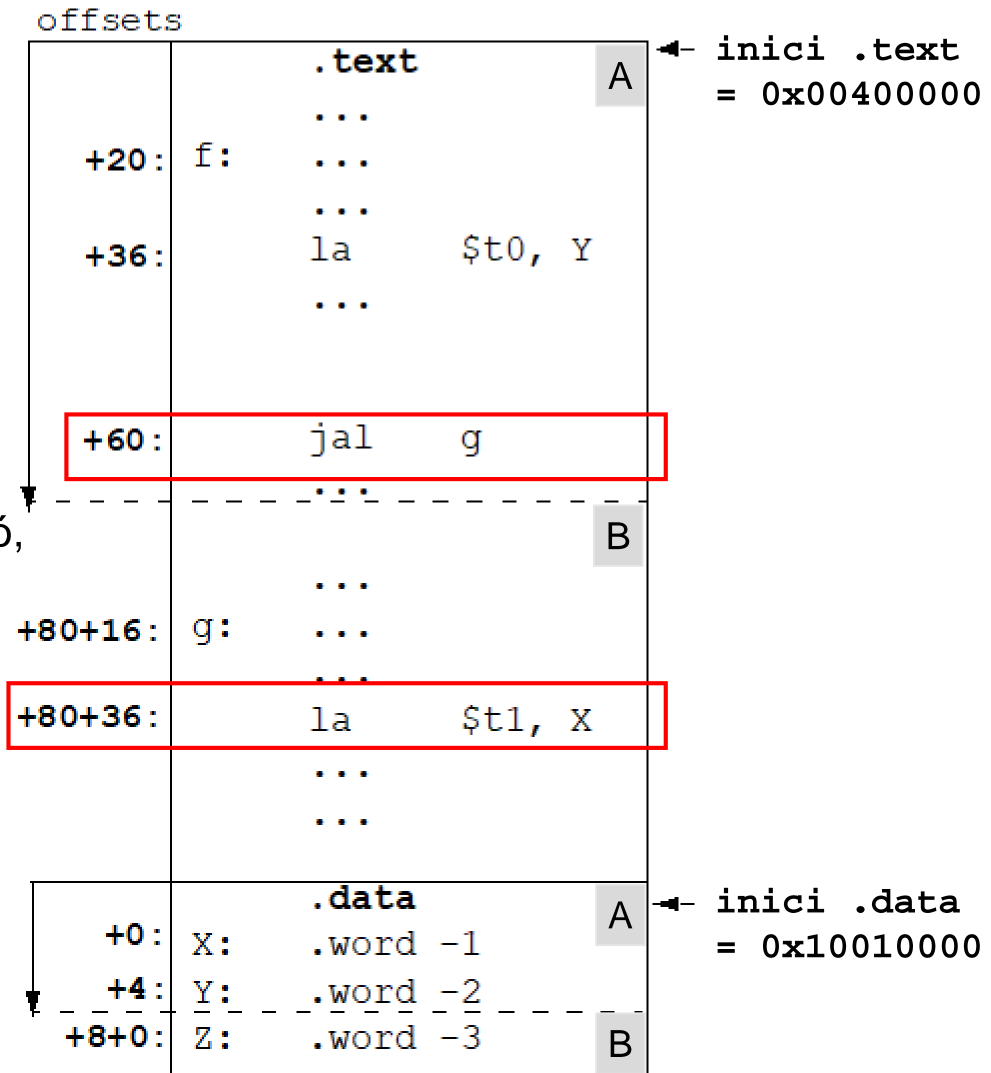
TS:

```
f = 0x00400000 + 20
Y = 0x10010000 + 4
Z = 0x10010000 + 8 + 0
```

TS-globals:

```
g = 0x00400000 + 80 + 16
X = 0x10010000 + 0
```

4. Reubicar instruccions
   - Corregir les de la Llista de Reubicació, amb les adreces definitives:
     - instrucció tipus **la** (.text + 36): @ = .data + 4
   - Corregir les de la Llista de Referències no-resoltes:
     - instrucció tipus **jal** (.text + 60): label = "g"
     - instrucció tipus **la** (.text + 36): label = "X"



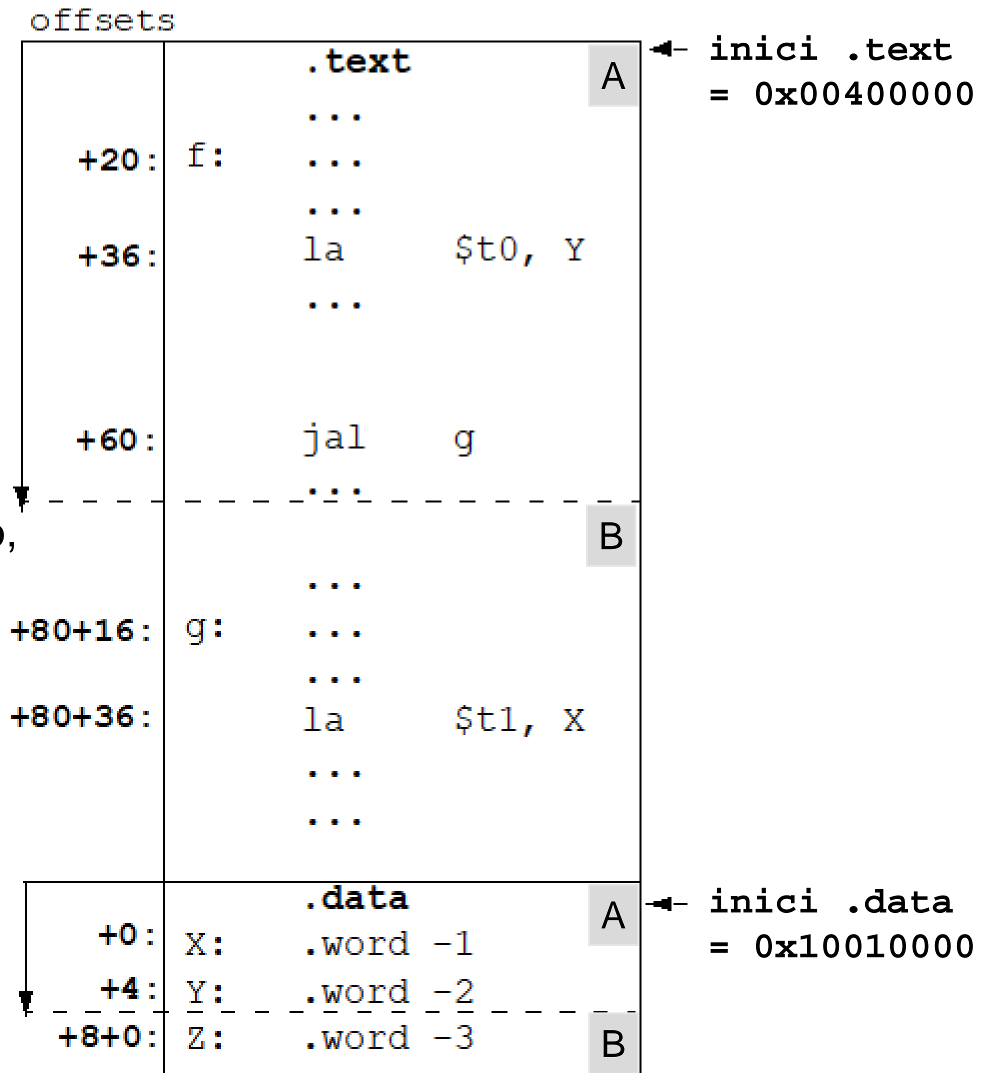
Enllaçat (o muntatge o "linkatge")

TS:

```
f = 0x00400000 + 20
Y = 0x10010000 + 4
Z = 0x10010000 + 8 + 0
```

TS-globals:

```
g = 0x00400000 + 80 + 16
X = 0x10010000 + 0
```



4. Reubicar instruccions
   - Corregir les de la Llista de Reubicació, amb les adreces definitives:
     - instrucció tipus **la** (.text + 36): @ = .data + 4
   - Corregir les de la Llista de Referències no-resoltes:
     - instrucció tipus `jal` (.text + 60): label = "g"
     - instrucció tipus `la` (.text + 36): label = "X"
5. Escriure el fitxer executable
   - Estructura similar al fitxer objecte

Càrrega en memòria

- El loader del SO carrega el programa en memòria
  1. Llegeix la capçalera per determinar la mida de les seccions
  2. Reserva memòria principal
  3. Copia instruccions i dades del fitxer a la memòria (al tema 7 veurem que no cal copiar-ho tot)
  4. Copiar a la pila els arguments en línia de comandes del programa
  5. Inicialitzar registres, deixant $sp apuntant al cim de la pila
  6. Saltar a la rutina *startup* (l'enllaçador l'ha inclòs a l'executable)

Càrrega en memòria

- El loader del SO carrega el programa en memòria
  1. Llegeix la capçalera per determinar la mida de les seccions
  2. Reserva memòria principal
  3. Copia instruccions i dades del fitxer a la memòria (al tema 7 veurem que no cal copiar-ho tot)
  4. Copiar a la pila els arguments en línia de comandes del programa
  5. Inicialitzar registres, deixant $sp apuntant al cim de la pila
  6. Saltar a la rutina *startup* (l'enllaçador l'ha inclòs a l'executable)

- Què fa *startup*?
  1. Copia els arguments del *main*(*int argc, char *argv*[]) de la pila als registres $a0, $a1
  2. Crida a la subrutina `main()`
  3. Quan `main()` retorna, *startup* invoca la rutina `exit()` del sistema per alliberar els recursos assignats al programa