Apunts del Tema 3
Traducció de Programes

Joan Manuel Parcerisa
Rubèn Tous

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Març 2016

Licencia Creative Commons

Tema 3. Traducció de programes

2.6

1. Desplaçaments de bits

1.1 Desplaçaments lògics a esquerra i dreta

Instruccions `sll` (Shift Left Logical) i `srl` (Shift Right Logical):

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

Desplaça *rt* a l'esquerra (`sll`) o a la dreta (`srl`) el nombre de bits indicat a l'operand immediat *shamt* ("shift amount"). Les posicions que queden vacants s'omplen amb zeros.

EXEMPLE 1: Desplaçar a l'esquerra 1 posició els bits de `$t0`

```
li     $t0, 0x88888888
sll    $t0, $t0, 1         # resultat: $t0 = 0x11111110
```

EXEMPLE 2: Desplaçar a la dreta 2 posicions els bits de `$t0`

```
li     $t0, 0x99999999
srl    $t0, $t0, 2         # resultat: $t0 = 0x26666666
```

1.2 Desplaçament aritmètic a la dreta

Instrucció `sra` (Shift Right Arithmetic):

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

Desplaça *rt* a la dreta el nombre de bits indicat a l'operand immediat *shamt*. Les posicions que queden vacants a l'esquerra s'omplen amb una còpia del bit de signe de *rt*.

EXEMPLE 3:

```
li     $t0, 0x88888888
srl    $t0, $t0, 1         #$t0 = 0xC4444444
```

1.3 Sintaxi i repertori d'instruccions de desplaçament

A diferència del processador SISA, el nombre de bits a desplaçar és sempre un número natural, així que s'usen instruccions diferents per desplaçar a la dreta o a l'esquerra. Per altra banda, aquest número es pot especificar com un immediat (camp *shamt* de 5 bits), o bé en un registre, i en aquest cas el número de bits a desplaçar ve expressat només pels 5 bits de menor pes del segon operand ($rs_{4:0}$), la resta de bits del

registre s'ignoren, encara que no siguin zeros:

sll/srl/sra/sllv/srlv/srav		
`sll rd, rt, shamt`	*rd* = *rt* << *shamt*	
`srl rd, rt, shamt`	*rd* = *rt* >> *shamt*	Inserta zeros a l'esquerra
`sra rd, rt, shamt`	*rd* = *rt* >> *shamt*	Extén signe a l'esquerra
`sllv rd, rt, rs`	*rd* = *rt* << $rs_{4:0}$	
`srlv rd, rt, rs`	*rd* = *rt* >> $rs_{4:0}$	Inserta zeros a l'esquerra
`srav rd, rt, rs`	*rd* = *rt* >> $rs_{4:0}$	Extén signe a l'esquerra

1.4 Utilitat dels desplaçaments: multiplicació i divisió per potències de 2

Desplaçar a l'esquerra un natural o un enter (amb `sll`) equival a multiplicar-lo per una potència de dos ($rd = rt \cdot 2^{shamt}$). El cas més freqüent és multiplicar l'índex d'un vector per la mida dels seus elements per calcular l'offset, si aquesta mida és potència de 2.

Desplaçar a la dreta un natural (amb `srl`) o un enter (amb `sra`) equival a dividir-lo per una potència de 2 ($rd = rt / 2^{shamt}$). No obstant, no acostuma a usar-se per a dividir enters negatius, ja que el resultat obtingut no és el de la coneguda "divisió per defecte" ($D = d \cdot q + r$, on el residu és del mateix signe que el dividend), sinó el de la "divisió per excés" ($D = d \cdot q + r$, on el residu ha de ser positiu).

EXEMPLE 4: Comparem la divisió entera -15/4 i el desplaçament -15>>2: la divisió entera -15/4 (divisió per defecte) dóna quocient q = -3 i residu r = -3; en canvi, el desplaçament -15>>2 dóna quocient q = -4 i residu r = 1.

1.5 Altres utilitats dels desplaçaments

Els desplaçaments poden tenir moltes altres utilitats, com la del següent exemple.

EXEMPLE 5: Representar en Ca1 un enter x_s guardat en format de Ca2 en `$t0`. Del tema anterior sabem que en Ca2 és $x_s = x_u - X_{n-1} \cdot 2^n$ (on $X_{n-1}..X_0$ és la cadena de bits, i x_u és el natural que representa). Anàlogament, en Ca1, és $x_s = x_u' - X'_{n-1} \cdot (2^n - 1)$. Igualant les dues expressions i simplificant-les (els bits de signe coincideixen en les dues representacions: $X_{n-1} = X'_{n-1}$) obtenim que la representació en Ca1 en funció de la representació en Ca2 és: $x'_u = x_u - X_{n-1}$. És a dir que la representació en Ca1 x'_u s'obté simplement restant el bit de signe X_{n-1} a la representació en Ca2 x_u:

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

1.6 Traducció dels operadors C de desplaçament a esquerra (<<) i a dreta (>>)

El llenguatge C defineix els operadors de desplaçament `<<` i `>>`. El desplaçament a l'esquerra sempre es tradueix per la instrucció `sll`. En canvi, el desplaçament a la dreta es tradueix per `srl` si es tracta d'un número natural o bé `sra` si es tracta d'un enter.

EXEMPLE 6: Traduir la següent sentència en C, suposant que *a* i *b* són enters:

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

En MIPS, suposant que *a* i *b* es guarden als registres `$t0`, `$t1`:

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

2. Operacions lògiques bit a bit

2.1 Operacions *and*, *or*, *xor*, i *not* bit a bit

Les instruccions `and`, `or`, i `xor` realitzen les corresponents operacions lògiques bit a bit, anàlogues a les estudiades en el processador SISA. Serveixen per traduir del llenguatge C els operadors lògics bit a bit "`&`" (*and*), "`|`" (*or*), i "`^`" (*xor*). No existeix una instrucció MIPS per a traduir directament l'operador lògic bit a bit "`~`" (*not*), però aquesta operació és equivalent a fer una *nor* amb el registre `$zero`.

EXEMPLE 7: Traduir la sentència en C: `a = ~(a & b);`

En MIPS, suposant que *a*, i *b* es guarden als registres `$t0`, `$t1`:

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

2.2 Sintaxi i repertori d'instruccions lògiques

En la següent taula, l'immediat *imm16* és un natural i s'extén a 32 bits amb zeros.

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

2.3 Utilitat de la *and* bit a bit: seleccionar bits (posant la resta a zero)

```
and    rd, rs, rt
```

La instrucció `and` s'usa per seleccionar determinats bits d'un registre, posant a zero la resta. Es posaran a zero aquells bits de *rs* que valguin zero en la **màscara** *rt* .

EXEMPLE 8: Seleccionar els 16 bits de menor pes de *rs*. Usarem la màscara *rt* = 0x0000FFFF per posar a zero la resta de bits (els 16 de major pes):

```
andi   rd, rs, 0xFFFF            # l'immediat s'extén amb zeros
```

EXEMPLE 9: Traduir el següent codi en C, que comprova si la variable *b* té actius els bits 0 i 4, i inactius els bits 2 i 6. Aplicarem la màscara $01\underline{0}1\underline{0}1\underline{0}1\underline{0}1_2$ per seleccionar els bits 0, 2, 4 i 6, i compararem el resultat amb la constant $0\underline{0}0\underline{1}0\underline{0}0\underline{1}_2$ que té actius els bits 0, 4:

```
a = b & 0x55;                    /* màscara  0x55 = 01010101₂ */
if (a == 0x11)                   /* constant 0x11 = 00010001₂ */
     {...}                       /* es_compleix_la_condició   */
```

En MIPS, suposant que *a* i *b* ocupen els registres `$t0` i `$t1`:

```
     andi   $t0, $t1, 0x0055     # seleccionem els bits 0, 2, 4 i 6
     li     $t4, 0x0011
     bne    $t0, $t4, endif      # if (a == 0x11) ...
     ...    # codi a executar si es compleix la condició
endif:
```

2.4 Utilitat de la *or* bit a bit: posar bits a u

```
or     rd, rs, rt
```

La instrucció `or` s'usa per posar a u aquells bits de *rs* que valen u a la màscara *rt*.

EXEMPLE 10: escrivim uns als 16 bits de menor pes de `$t0`:

```
ori    $t0, $t0, 0xFFFF          # l'immediat s'extén amb zeros
```

2.5 Utilitat de la *xor* bit a bit: complementar bits

```
xor    rd, rs, rt
```

La instrucció `xor` s'usa per complementar els bits de *rs* que valen u a la màscara *rt*.

EXEMPLE 11: complementem els bits parells de `$t0`:

```
li     $t1, 0x55555555           # màscara amb uns als bits parells
xor    $t0, $t0, $t1
```

2.7

3. Comparacions i operacions booleanes

En C, les expressions enteres admeten els operadors de comparació, per igualtat o per desigualtat ("==", "!=", "<", "<=", ">", ">="). A diferència d'altres llenguatges, en C no existeix el tipus booleà, i en canvi el resultat de la comparació és un enter: si la comparació és certa el resultat és un 1, altrament és un 0. Les comparacions seran amb signe o sense segons el tipus dels operands.

En C existeixen també expressions lògiques formades pels operadors booleans *and*, *or* i *not* ("&&", "||", "!"). Al igual que per les comparacions numèriques, el resultat de l'operació serà un enter: si el resultat és cert valdrà 1, altrament valdrà 0. Cada operand és també un enter, que s'interpreta com a fals si és 0 o com a cert si és diferent de 0.

Degut a aquesta manera perticular en què C maneja els valors booleans sense definir-los explícitament, cal estar alerta a l'ambigüitat, perquè dues variables poden ser interpretades com a certes sense ser iguals! No obstant, tant les comparacions com els operadors booleans sempre donen com a resultat un enter "normalitzat", 0 o 1. P.ex., suposant que $y = 2$ i $z = 0$, la sentència *"*`x = y>=z;`*"* assigna a x el valor 1. Tant les comparacions com les operacions booleanes serveixen sovint per expressar les condicions de sentències alternatives (*if*, *switch*) i iteratives (*while*, *for*) que veurem més endavant.

3.1 Sintaxi i repertori d'instruccions de comparació

A tots els immediats *imm16* se'ls extén el signe, sigui la comparació amb signe o sense.

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

3.2 Comparacions del tipus "<"

La instrucció `slt` (Set Less Than) compara dos enters i dóna com a resultat un 1 si el primer és menor que el segon, o bé 0 en cas contrari. La instrucció `sltu` (Set Less Than Unsigned) fa la mateixa operació però considerant els operands com a naturals. L'operador "<" de C es tradueix per `slt` o `sltu` segons si compara enters *signed* o *unsigned*):

EXEMPLE 12: Traduir, suposant que a, b, c es guarden en `$t0`, `$t1`, `$t2`

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

La traducció en MIPS serà (segons si `a` i `b` són naturals o enters):

```
sltu   $t2, $t0, $t1       # si a, b són naturals
slt    $t2, $t0, $t1       # si a, b són enters
```

3.3 Traducció de les comparacions ">", "<=", ">="

Com s'ha vist, MIPS sols té instruccions amb la comparació "menor que", ja que les altres tres desigualtats es poden traduir usant "menor que", simplement aplicant un canvi d'ordre dels operands o una negació del resultat. Observem les següents equivalències:

$a > b \Leftrightarrow b < a$;

$a \geq b \Leftrightarrow (\overline{a < b})$;

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

EXEMPLE 13: Traduir `c=(a>b);` suposant que *a*, *b*, *c* es guarden en `$t0`, `$t1`, `$t2`:

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

Com que les comparacions han de produir un resultat de 0 o 1, la traducció de la *not* booleana no es pot fer aplicant l'operació *not* bit a bit, ja que aquesta sempre dóna diferent de zero, tant si l'operand és 0 com 1. La solució més simple és la comparació "menor que 1, com a natural", ja que donarà 0 per a qualsevol valor diferent de zero, i 1 altrament.

EXEMPLE 14: Traduir `c=(a<=b);` suposant que *a*, *b*, *c* es guarden en `$t0`, `$t1`, `$t2`:

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

3.4 Traducció de les comparacions "`==`" i "`!=`"

Per traduir les comparacions "`==`" i "`!=`" no calen instruccions especials, tan sols una resta seguida d'una negació lògica del resultat o d'una normalització als valors 0 o 1:

EXEMPLE 15: Traduir `c=(a==b);` suposant que *a*, *b*, *c* es guarden en `$t0`, `$t1`, `$t2`:

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

EXEMPLE 16: Traduir `c=(a!=b);` suposant que *a*, *b*, *c* es guarden en `$t0`, `$t1`, `$t2`:

```
sub    $t4, $t0, $t1        # val zero si són iguals
sltu   $t2, $zero, $t4      # normalitzem el resultat a 0 o 1
```

3.5 Traducció de l'operació booleana "`&&`"

Es pot utilitzar la instrucció `and`, però amb una prevenció: cal assegurar abans que els operands estiguin normalitzats als valors 0 o 1 (per mitjà de la instrucció `sltu`, com hem vist en l'exemple 16).

EXEMPLE 17: Traduir `c = a&&b;` suposant que *a*, *b*, *c* es guarden en `$t0`, `$t1`, `$t2`:

```
sltu   $t3, $zero, $t0      # normalitzem l'operand a
sltu   $t4, $zero, $t1      # normalitzem l'operand b
and    $t2, $t3, $t4
```

3.6 Traducció de l'operació booleana "`!`"

La negació lògica es pot traduir sempre amb una simple instrucció `sltiu`, tant si l'operand està normalitzat com si no. Compte, perquè l'operació not bit a bit no serveix (p. ex. converteix un 1 en un -2, en comptes d'un 0)

EXEMPLE 18: Traduir `b=!a;` suposant que *a*, *b* es guarden en `$t0`, `$t1`:

```
sltiu  $t1, $t0, 1          # negació lògica
```

La traducció de les condicions "`&&`" i "`||`" la veurem a la secció 5.2, ja que impliquen avaluació "lazy": si el resultat de la part esquerra determina el valor de tota la condició, la part dreta ja no s'ha d'avaluar.

2.7

4. Salts

4.1 Salts condicionals relatius al PC (*branch*): `beq` i `bne`

La instrucció `beq` salta a l'adreça especificada per l'etiqueta si els operands són iguals, mentre que la instrucció `bne` salta si són diferents. L'adreça de destinació s'especifica per mitjà d'una etiqueta (*label*), que és una forma simbòlica de designar l'adreça en llenguatge assemblador. No obstant, aquesta adreça té 32 bits i és massa llarga per codificar-la dins la instrucció. El que es codifica en realitat, com un immediat de 16 bits, és la distància a saltar (*offset*16), de manera que aquests salts s'anomenen "relatius al PC". Per ser exactes, *offset*16 és la diferència entre l'adreça destinació i l'adreça de la instrucció següent al salt (que anomenem PC_{up} = PC+4), mesurada en número d'instruccions i codificada en Ca2 amb 16 bits. És a dir, *offset*16 = (*label* - PC_{up})/4), i permet saltar dins el rang de $[-2^{15}, 2^{15}-1]$ instruccions de distància respecte al PC_{up}. Amb la instrucció `beq` també podem implementar un salt incondicional, comparant `$zero` amb `$zero`, i aquesta és precisament la definició de la macro `b`. Convé usar-la, per facilitar la lectura dels programes.

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

4.2 Altres salts condicionals relatius al PC: `blt`, `bgt`, `bge`, `ble`

Els salts que depenen de comparacions <, >, ≤, ≥ no existeixen com a instruccions MIPS, ja que es poden implementar amb una comparació (`slt` o `sltu`) i un salt condicional (`beq` o `bne`). No obstant, resulten tan pràctics per a l'escriptura manual de programes que s'han creat macros amb les quatre combinacions `blt`, `bgt`, `bge`, `ble` (i les corresponents comparacions de naturals `bltu`, `bgtu`, `bgeu`, `bleu`):

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

Observem com hem fet l'expansió de les tres darreres macros:

(1) `rs>rt` equival a `rt<rs`. Intercanviar els operands del `slt`.

(2) `rs>=rt` equival a `!(rs<rt)`. Saltar si condició falsa, amb `beq`.

(3) `rs<=rt` equival a `(rt>=rs)`, que equival a `!(rt<rs)`. Intercanviar els operands del `slt`, i saltar si condició falsa, amb `beq`.

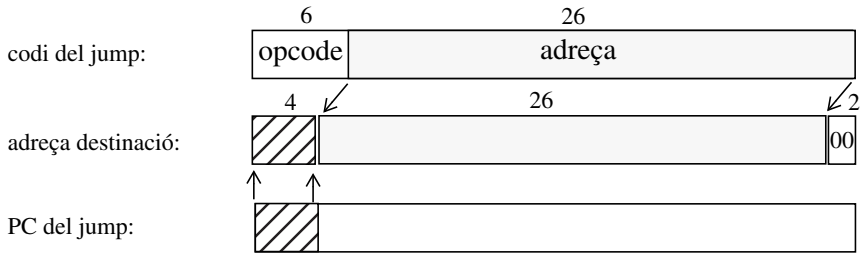
Les macros `bltu`, `bgtu`, `bgeu` i `bleu` s'expandeixen de manera anàloga, usant `sltu` en comptes de `slt`.

4.3 Salts incondicionals en mode registre o pseudodirecte: `j`, `jr`, `jal`, `jalr`

Abans s'ha estudiat com es pot realitzar un salt incondicional relatiu al PC (amb la macro `b` basada en `beq`). Però si la distància del salt en instruccions supera el rang de 16 bits [-2^{15}, 2^{15}-1] llavors haurem d'usar altres instruccions de salt incondicional (*jumps*):

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

Les instruccions j i jal es codifiquen en el format J (veure secció XX), i l'adreça destinació es codifica en els 26 bits de menor pes de la pròpia instrucció, usant l'anomenat mode pseudodirecte. Quan aquesta s'executa, cal completar els 6 bits que falten de la següent manera: primer, es desplacen els 26 bits 2 posicions a l'esquerra fent zeros els 2 bits de menor pes (l'adreça d'una instrucció és sempre múltiple de 4!). Després, se li afegeixen els 4 bits de major pes, copiant-los dels del registre PC actual.



Així doncs, en mode pseudodirecte sols es pot saltar a adreces amb els 4 bits de més pes iguals als del PC actual, és a dir que resideixin dins del mateix bloc de 2^{28} bytes (256MB) de la pròpia instrucció de salt. Sol ser un rang suficient per a la majoria de salts, altrament cal usar *jr* (Jump to Register).

La utilitat de les instruccions *jal* i *jalr* s'explicarà a la secció 7.1 de Subrutines.

4.4 Recapitulació: modes d'adreçament

Un mode d'adreçament és una manera d'especificar un operand en una instrucció. Recapitulant, veiem que en MIPS hi ha 5 modes d'adreçament:

1- Mode Immediat (e.g. `addiu`)

2- Mode Registre (e.g. `addu`, `jr`)

3- Mode Base (e.g. `lw`, `sw`)

4- Mode Relatiu al PC (e.g. `beq`)

5- Mode Pseudodirecte (`j`).

2.7

5. Sentències alternatives *if-then-else* i *switch*

5.1 Sentència *if-then-else*

En C, la condició d'un *if* és una expressió numèrica. Consisteix sovint en una comparació o una operació booleana on el resultat és 0 o 1. Però en ser una expressió entera qualsevol, el resultat potser diferent de 0 o 1. En qualsevol cas, el resultat de l'expressió s'avaluarà com a *fals* si val zero, o bé com a *cert* si val diferent de zero.

Sigui el cas general en C:

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

El patró en MIPS serà:

```
        avaluar condicio
        salta si és falsa a sino
        traducció de sentencia_then
        salta a fisi
sino:
        traducció de sentencia_else
fisi:
```

La clàusula *else* és opcional. Si aquesta no existeix, el primer salt va directament a l'etiqueta `fisi`, que es col·loca just després de la traducció de la `sentencia_then`. La `sentencia_else` pot ser també del tipus *if-then-else* (anidada), però aquesta segueix exactament el mateix patró de traducció que la sentència principal. Per altra part, la condició de l'*if* pot ser tan simple que no requereixi cap instrucció, tan sols el salt condicional. O bé pot ser composta de diverses operacions (exemples 20 i 21 a continuació).

EXEMPLE 19. En C, amb una condició simple:

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

En MIPS serà, suposant que *a*, *b*, *c*, *d* són enters i ocupen `$t0`, `$t1`, `$t2`, `$t3`:

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

5.2 Avaluació "lazy"

En C, els operadors lògics ("`&&`" i "`||`") s'avaluen d'esquerra a dreta de forma "lazy" (si la part esquerra ja determina el resultat, la part dreta NO s'ha d'avaluar).

EXEMPLE 20. Vegem una expressió composta amb l'operador booleà and ("`&&`"):

```
if ( a >= b && a < c )          // && té menys precedència que >= o <
     d = a;
else
     d = b;
```

En MIPS, suposant que *a*, *b*, *c*, *d* són enters i ocupen `$t0`, `$t1`, `$t2`, `$t3`:

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

EXEMPLE 21. Expressió composta amb l'operador booleà or ("||"):

```
if ( a >= b || a < c )          /* || té menys precedència que >= o < */
     d = a;
else
     d = b;
```

En MIPS, suposant que *a*, *b*, *c*, *d* són enters i ocupen `$t0`, `$t1`, `$t2`, `$t3`:

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

5.3 Sentència *switch*

Sigui la forma més típica d'una sentència alternativa múltiple *switch* en C:

```
switch (expressio) {
     case const1: sentencies1; break;
     case const2: sentencies2; break;
     ...
     default:     sentenciesN;
}
```

El *switch* permet seleccionar una d'entre múltiples clàusules *case* alternatives, i executar les sentències que aquesta té associades. La selecció es fa comparant el resultat de l'expressió (que ha de ser de tipus enter), amb les diverses etiquetes enteres constants, les quals no poden estar repetides. Si una d'elles coincideix, s'executen les sentències associades a aquesta clàusula i a les següents, fins a trobar alguna sentència *break*. Si cap etiqueta coincideix, s'executa la clàusula *default* si existeix, o no s'executa res altrament.

El *switch* es pot traduir com una cadena de sentències *if-then-else* que comprovin les etiquetes seqüencialment fins a trobar la que coincideixi amb el resultat de l'expressió. No obstant, un mètode més eficient consisteix a definir un vector de punters que apunten a adreces de codi, i que s'indexa amb el resultat de l'expressió. Cada punter del vector apunta al codi d'una clàusula case: en concret, l'element *n*-èssim apunta al codi de la clàusula "*case n*". Si no existeix tal clàusula, llavors apunta a la clàusula *default* si està definida, o al final del *switch* en cas contrari. P.ex. suposant que `$t0` conté el resultat de l'expressió del *switch*, i el vector de punters està a l'adreça *switch_vec*, el salt seria:

```
     la     $t1, switch_vec     # adreça inicial del vector
     sll    $t2, $t0, 2         # offset
     addu   $t1, $t1, $t2       # adreça efectiva
     lw     $t3, 0($t1)         # carreguem el punter en $t3
     jr     $t3                 # saltem al codi apuntat per $t3
```

2.7

6. Sentències iteratives *while*, *for* i *do-while*

6.1 Sentència *while*

La sentència while executa zero o més iteracions mentre es compleixi la condició.

Sigui el cas general en C:

```
while (condicio)
     sentencia_cos_while
```

El patró en MIPS serà:

```
while:
        avaluar condicio
        salta si és falsa a fiwhile
        traducció de sentencia_cos_while
        salta a while
fiwhile:
```

La *condició* és una expressió qualsevol[1] com en el cas del *if*. Pot ser tan simple que no calgui cap instrucció per avaluar-la (l'avalua el propi salt condicional a `fiwhile`).

EXEMPLE 22. Sigui la divisió entera de nombres positius en C:

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

En MIPS serà, suposant que *dd*, *dr*, *q* són enters i ocupen `$t1`, `$t2`, `$t3`:

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

a) Optimització: avaluació de la condició al final del bucle

Observem que podem optimitzar la solució eliminant el salt incondicional final (`b`), si avaluem la condició al final del bucle. Però si ho fem així, ens hem d'assegurar que es manté la semàntica de l'iterador *while*, que requereix que no s'executi cap iteració si la condició és inicialment falsa. Algunes vegades podem garantir que el valor inicial de la condició serà sempre cert (per exemple, si inicialitzem `x=100`, i la condició del *while* és `x>0`), i llavors la transformació del codi ja és correcta. Però sovint (com en l'exemple anterior) el valor inicial de la condició (`dd>=dr`) no es pot predir, i llavors cal afegir una comprovació de la condició just abans d'entrar al bucle:

```
     move   $t3, $zero                # q = 0
     blt    $t1, $t2, fiwhile         # salta si és fals que dd>=dr
while:
     subu   $t1, $t1, $t2             # dd = dd - dr
     addiu  $t3, $t3, 1               # q++
     bge    $t1, $t2, while           # salta si dd>=dr
fiwhile:
```

1. En la terminologia de C, ha de ser una expressió de tipus "enter" (amb signe o sense) o punter.

Si ens volem estalviar escriure dos cops el codi que avalua la condició (p.ex. si és massa llarg), podem substituir la comprovació inicial per un simple salt incondicional a la instrucció on comença l'avaluació de la condició. La versió definitiva queda així:

```
     move   $t3, $zero                # q = 0
     b      test                      # salta a fer comprovació inicial
while:
     subu   $t1, $t1, $t2             # dd = dd - dr
     addiu  $t3, $t3, 1               # q++
test:
     bge    $t1, $t2, while           # salta si dd>=dr
```

6.2 Sentència *for*

```
for (s1; condicio; s2)
     s3;
```

És totalment equivalent a un *while*[2], escrit de la següent manera:

```
s1;
while (condicio) {
     s3;
     s2;
}
```

Encara que *s*1 i *s*2 poden ser sentències simples qualsevols[3], el costum és que *s*1 inicialitzi la variable de la qual depèn la condició del bucle (típicament un comptador) i que *s*2 l'actualitzi després de cada iteració. Quan es fa així, resulta avantatjós utilitzar el *for* en lloc del *while*, ja que agrupa tos els elements que controlen el bucle en una única capçalera, i això el fa més fàcil de llegir.

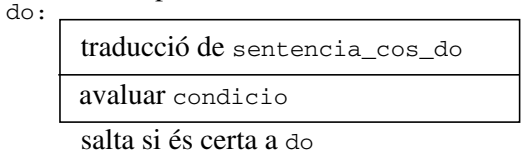
6.3 Sentència *do-while*

La sentència *do-while* executa una o més iteracions mentre es compleix la condició. Observem que la primera iteració s'executa sempre.

Sigui el cas general en C:

```
do
     sentencia_cos_do
while (condicio);
```

El patró en MIPS serà:



Resulta útil quan la condició del bucle depèn d'algun valor que s'ha d'inicialitzar precisament executant el cos del bucle, i així evitem replicar-lo:

```
     do
            dada = obtenir_dada();
     while (dada != dada_esperada);
```

2. Estrictament parlant, són equivalents sempre que el *for* no contingui cap sentència *continue*, que no estudiarem en aquest curs.

3. En la nomenclatura de C, s1 i s2 han de ser "expressions" en el sentit més general. A la pràctica però, sols té sentit usar expressions que produeixin efectes sobre l'estat del programa, tals com assignacions o crides a funcions contenint assignacions. No són "expressions", en canvi, les sentències com if, switch, for, while, etc. ni blocs de sentències enclosos entre claudàtors.

2.8, B-6

7. Subrutines

Anomenem subrutina a la generalització d'una estructura de programació que coneixem normalment com a acció o funció (però també procediment, mètode, subprograma, etc.) i que permet escriure programes estructurats, implementant conceptes de programació com el disseny descendent, reutilització de codi, programació modular, etc. Una subrutina és un subprograma que executa una determinada tasca amb determinats paràmetres podent retornar un resultat, i permet ser invocada des de múltiples punts del programa sense haver de ser reescrita cada vegada.

Per tal que una subrutina i el programa que la crida puguin ser programades en assemblador (o compilades) de forma independent cal que la subrutina declari de forma precisa la seva interfície (nom, paràmetres, resultats i tipus respectius), i que els respectius codis s'ajustin a un conjunt de regles estrictes que s'explicaran en aquesta secció. Aquestes regles formen part d'un conjunt més ampli anomenat ABI (Application Binary Interface). Així, per a cada ISA, SO i llenguatge d'alt nivell es defineix una ABI específica que permet compartir les subrutines emmagatzemades en biblioteques (*libraries*)[4].

EXEMPLE 23. Dues subrutines en C (amb i sense retorn de resultat).

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

void init(int v[], int a) {
     int i;
     for (i=0; i<10; i++) v[i] = a;
}
```

7.1 Crida i retorn

Per cridar a una subrutina podríem usar la pseudoinstrucció *b*:

Codi que fa la crida

```
      b      sub
adr_retorn:
      ...
```

Codi de la subrutina

```
sub:
      ...
      b      adr_retorn
```

Però això no funciona si volem invocar la subrutina des de múltiples punts:

Codi que fa les crides

```
      b      sub
adr1:
      ...
      b      sub
adr2:
      ...
```

Codi de la subrutina

```
sub:
      ...
      b      ¿adr1 o 2?
```

El problema rau en el fet que hi ha una adreça de retorn diferent per a cada punt d'invocació de la subrutina, però les instruccions de salt relatiu al PC només poden codificar una sola adreça, que es determina en temps de compilació, el mateix que passa amb la

4. A fi de concentrar l'estudi en els conceptes bàsics, l'ABI que hem considerat en l'assignatura d'EC solament cobreix els casos de programació més bàsics i freqüents, ignorant deliberadament els casos més complexos

instrucció `j` (Jump). La solució seria que, abans de cridar la subrutina, memoritzéssim l'adreça de retorn en algun lloc prèviament convingut per tal que la subrutina pugui llegir-la per retornar al lloc adequat.

El processador MIPS dedica el registre `$ra`[5] (Return Address, registre `$31`) per guardar aquesta adreça. Per altra banda, el repertori del MIPS té la instrucció `jal` (Jump and Link) que no sols salta a l'adreça indicada per l'operand (expressat com una etiqueta i codificat en mode pseudodirecte), sinó que a més a més copia l'adreça de retorn (PC+4) en el registre `$ra`. I per retornar de la subrutina usarem la ja coneguda instrucció `jr` (Jump to Register) especificant `$ra` com a adreça de destinació del salt.

Codi que fa la crida

```
      jal    sub    # $ra = adr1
adr1:        ...

      jal    sub    # $ra = adr2
adr2:        ...
```

Codi de la subrutina

```
sub:
       ...
       jr     $ra
```

7.2 Paràmetres i resultats

En general, les funcions admeten rebre paràmetres i retornar resultats. El pas de paràmetres i resultats implica una comunicació entre la funció i el codi que la crida. Però la programació modular requereix programar amb independència el codi que fa la crida i el de la funció cridada. Cal doncs que paràmetres i resultats es dipositin en un lloc prèviament convingut, seguint un conveni definit en l'ABI. En concret en MIPS, els paràmetres i el resultat es passen en certs registres. Abans de la crida a una subrutina s'ha de copiar el paràmetre real en un registre determinat, al qual accedirà la subrutina quan vulgui usar el paràmetre formal. Anàlogament, abans de finalitzar, la subrutina escriu el resultat en un registre determinat, i el codi que l'ha invocat hi accedirà després de la crida.

a) Regles de pas de paràmetres (escalars) i resultats

En EC considerarem el següent conjunt reduït de regles[6]:

- Els paràmetres de tipus escalars es passen en els registres `$a0-$a3`, el primer paràmetre en ordre d'escriptura en `$a0`, el segon en `$a1`, etc. (excepte els que són de coma flotant en simple precisió (*float*), que es passen en el registre `$f12` el primer paràmetre i en `$f14` el segon[7]).
- El resultat es passa en el registre `$v0` (excepte si és de coma flotant en simple precisió, que es passa en `$f0`).

5. En processadors amb menys registres (p.ex. x86), l'adreça de retorn es guarda a la pila.
6. Per simplicitat, en l'assignatura d'EC no considerarem el cas en què hi ha més de 4 paràmetres ni el cas en què paràmetres o resultats són de tipus tupla (*struct*, en C) o bé de tipus escalars de doble precisió (enters *long long* o flotants *double*). Tampoc admetrem més de 2 paràmetres flotants (*float*) ni mescla entre paràmetres flotants i altres que no ho són. Tots aquests casos requereixen regles addicionals de l'ABI que no estudiarem. Per a alguns d'aquests casos cal que els paràmetres es guardin a la pila, com també ho fan altres processadors amb menys registres (Intel x86).
7. El processador MIPS incorpora un coprocessador matemàtic CP1 per facilitar els càlculs amb nombres codificats en coma flotant. El CP1 extén el repertori amb noves instruccions per a operar nombres en coma flotant, les quals tenen com a operands un conjunt addicional de registres de 32 bits $f0.. $f31. Els nombres en coma flotant s'estudiaran en el Tema 5.

EXEMPLE 24: Donat el següent codi 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;
}
```

Traduir la funció *suma2* i les sentències visibles de *main* a assemblador MIPS:

```
main:
     ...
     move   $a0, $t0  # passem x
     move   $a1, $t1  # passem y
     jal    suma2
     move   $t2, $v0  # z=resultat
     ...
```

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

Si un paràmetre o resultat té menys de 32 bits (*char* o *short*) llavors s'extén la seva representació convenientment fins a 32 bits, dins el registre que tingui assignat, fent extensió de signe o de zeros segons estigui declarat *signed* o *unsigned*, respectivament.

El llenguatge C no determina en quin ordre s'han d'avaluar els paràmetres reals d'una crida (poden ser expressions amb autoincrements, assignacions, o resultats de crides a funcions). Per tant, si es donés el cas que l'avaluació d'un paràmetre produeix "efectes secundaris" (com ara modificacions de variables globals) que afecten al valor d'un altre paràmetre, llavors el resultat del programa podria variar segons l'ordre d'avaluació elegit pel compilador. Aquestes indeterminacions, tot i que indesitjables, són possibles en el llenguatge C, i l'única manera d'evitar-les és a través d'una bona pràctica de programació.

b) Paràmetres de tipus vector o matriu

En C, el nom d'un vector equival a un punter al seu primer element. Per tant, si escrivim el nom d'un vector com a paràmetre real en invocar una funció, estem en realitat passant un punter al seu primer element. El corresponent paràmetre formal a la capçalera de la funció serà del tipus punter, i es pot declarar indistintament com a tal (p. ex. `int *vec`) o com a vector (p. ex. `int vec[]`). Sigui quin sigui el format usat, el codi de la funció podrà aplicar al paràmetre tant els operadors de punters (p.ex. desreferència `*vec`) com de vectors (p.ex. indexat `vec[2]`). Cal notar que amb el segon format de declaració no és necessari especificar-ne el nombre d'elements, de fet és una informació irrellevant que no limita ni afecta en res a la traducció de la funció a llenguatge assemblador.

En C es poden declarar dades de tipus matriu (similars als vectors, però multidimensionals, les estudiarem en el pròxim tema). Al igual que amb els vectors, el nom de la matriu equival a un punter al seu primer element. Per tant, si escrivim el nom d'una matriu com a paràmetre real en invocar una funció, estem en realitat passant un punter al seu primer element. El corresponent paràmetre formal a la capçalera de la funció serà del tipus punter, i es declararà com una matriu (p. ex. `int mat[][100]`). Observem que, anàlogament als paràmetres vectors, aquesta declaració no necessita especificar la primera dimensió (número de files) però sí la segona (número de columnes), a fi de poder traduir expressions que indexin la matriu (p. ex. `mat[i][j]`), com es veurà en el pròxim tema. Alternativament, la declaració de la matriu a la capçalera de la funció pot adoptar també el format de punter (p. ex. `int *mat`) però resulta poc pràctic, ja que aquest format, com que no especifica el nombre de columnes, no permet tampoc usar després expressions que indexin la matriu.

EXEMPLE 25: Siguin les següents declaracions de variables i capçaleres de funcions:

variables

```
short V1[10];
char V2[20];
int a;
int *p;
```

capçaleres de funcions

```
int f(int i);
int f(char c);
int f(short *pi);
int f(char vc[]);
int f(int vi[]);
```

A quines capçaleres corresponen les següents crides (columna esquerra)?

crides

```
a = f(V1);
a = f(p);
a = f(&V2[10]);
a = f(V2[10]);
a = f(*p);
```

solució

```
int f(short *pi);
int f(int vi[]);
int f(char vc[]);
int f(char c);
int f(int i);
```

c) Pas de paràmetres per valor i per referència

En els cursos de programació s'estudia la diferència entre paràmetres passats per valor (la funció rep una còpia del paràmetre real, i pot modificar-la sense que afecti a aquest), i paràmetres passats per referència (les modificacions que faci la funció sobre el paràmetre afecten directament al paràmetre real). Però en llenguatge C sols existeix el pas de paràmetres per valor.

Tot i així, en C es pot recórrer als punters per aconseguir un resultat semblant al dels paràmetres per referència. Si es desitja que una funció pugui modificar la dada original que se li passa com a paràmetre, llavors tan sols cal que se li passi com a paràmetre un punter apuntant a aquesta dada, en lloc de passar-li una còpia de la dada pròpiament dita. Encara que en realitat el punter es passa per valor, la dada apuntada podrà ser modificada per la funció simplement desreferenciant el punter. En el següent exemple, la funció *f* crida dos cops a *sub* passant-li per valor un punter, primer apuntant a la variable *a* i després a la variable *b*, amb la intenció que la funció *sub* les modifiqui:

```
int a, b;
void f() {
int *p = &a;
     sub(p);
     sub(&b);
}
```

```
int sub(int *p) {
     *p = *p + 10;
}
```

d) Un exemple de pas de paràmetres i resultats

EXEMPLE 26: Suposant les següents declaracions de variables globals i de funcions, traduir a MIPS les sentències visibles de les funcions *funcA* i *funcB*:

```
short x[10], y, z;
void funcA(){
       int k;              /* suposem que k es guarda en $t0 */
       ...
       z = funcB(x, y, k);
       ...
}

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

En MIPS serà:

```
funcA:
     ...
     la     $a0, x              # passem l'adreça del vector x (global)
     la     $t4, y
     lh     $a1, 0($t4);        # passem y (global) fent ext. de signe
     move   $a2, $t0            # passem k (local)
     jal    funcB
     la     $t4, z
     sh     $v0, 0($t4)         # escrivim resultat a z (global)
     ...
     jr     $ra

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

7.3 Les variables locals

Per defecte, en C les variables locals d'una funció s'inicialitzen cada cop que s'invoca. Però si no hi ha inicialització explícita, el valor inicial és indeterminat. Són visibles només dins de la funció on estan definides, i tenen reservat un espai d'emmagatzemament temporal, només durant l'execució de la funció. Aquest espai pot ser en registres del processador o en memòria, per decidir-ho l'ABI defineix unes regles que explicarem aquí.

a) El bloc d'activació i la pila del programa

No totes les funcions requereixen guardar variables locals en memòria, però aquelles que ho necessitin (segons les regles del següent apartat) hauran de reservar l'espai necessari en el moment d'iniciar l'execució, i l'hauran d'alliberar novament just abans de finalitzar. Aquest espai rep el nom de *bloc d'activació* (*stack frame*) de la subrutina.

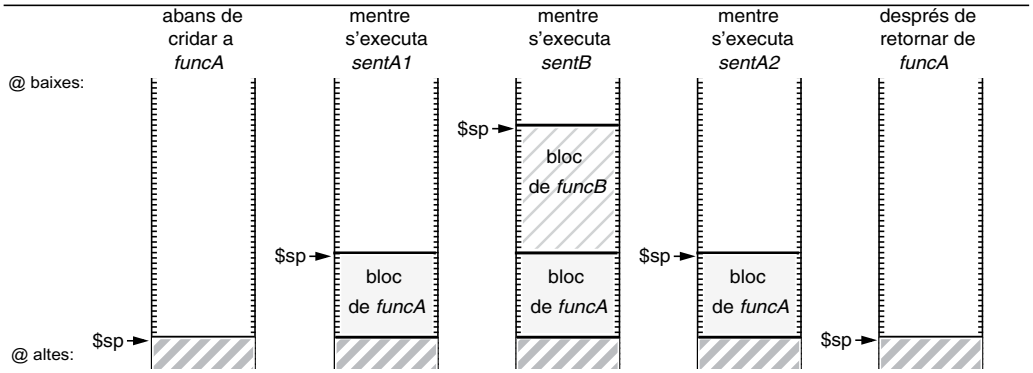
Si una funció A en crida a una altra B, i aquesta a una altra C, i totes tres necessiten guardar variables en memòria, observem que A reserva memòria, després B reserva memòria, i finalment C reserva memòria. Quan C acaba, allibera la seva memòria, quan B acaba allibera la seva, i quan finalment A acaba, allibera també la seva. És fàcil de veure doncs, que l'estructura de dades ideal per guardar les variables locals en memòria és una *pila*, ja que els blocs d'activació de les successives crides anidades s'alliberen en ordre invers a com s'han reservat. La pila creix a mesura que augmenta el nivell d'anidament en cada nova crida i decreix a mesura que disminueix el nivell d'anidament, quan retornem.

En general, tots els programes mantenen una pila en memòria per donar suport a la programació de subrutines, la qual s'anomena *pila del programa*. En MIPS, el *fons* de la pila està situat sempre a l'adreça fixa `0x7FFFFFFC` (és l'adreça més alta múltiple de 4 en l'espai de memòria d'usuari) i creix desplaçant el seu *cim* cap a adreces més baixes. El processador MIPS dedica el registre `$sp` (*stack pointer*, registre `$29`) a apuntar sempre al cim d'aquesta pila, és a dir al primer byte del darrer bloc d'activació creat. Abans de començar el programa, la pila és buida i el sistema inicialitza `$sp=0x7FFFFFFC`. Més endavant, cada funció (inclosa *main*), en iniciar l'execució, pot reservar-hi espai decrementant `$sp` tants bytes com la mida del seu bloc d'activació, una mida que ha de ser sempre un

múltiple de quatre[9]. Abans de finalitzar, la funció allibera l'espai novament incrementant `$sp` en la mateixa quantitat.

EXEMPLE 27: Vegem l'estat de la pila mentre s'executen les funcions *funcA* i *funcB*:

```
funcA()                    funcB()
{       sentA1;            {
        funcB();                  sentB;
        sentA2;            }
}
```



b) Regles de l'ABI per assignar variables locals a registres o a la pila

En MIPS, les variables locals d'una subrutina es guarden en registres o a la pila seguint les següents regles:

- Les variables de tipus escalars es guarden en algun dels registres `$t0..$t9`, `$s0..$s7` o `$v0..$v1` (excepte les que són de coma flotant en simple precisió[8], que es guarden als registres `$f0..$f31`).
- Les variables estructurades (vectors, matrius, tuples, etc.) es guarden a la pila.
- En el cas que no hi hagi suficients registres per emmagatzemar totes les variables locals escalars, les que no càpiguen en registres es guarden també a la pila.
- Si en el cos de la funció apareix una variable local escalar *x* precedida de l'operador unari `&` ("adreça de"), aquesta variable s'ha de guardar a la pila, a fi que tingui una adreça i es pugui avaluar l'expressió `&x`.

El bloc d'activació format per les variables locals guardades a la pila ha de respectar unes certes normes de format:

- Les variables de la pila es col·loquen seguint l'ordre en què apareixen declarades al text, començant per l'adreça més baixa que és la del cim de la pila.
- Les variables de la pila han de respectar l'alineació corresponent al seu tipus, igual que les variables globals, deixant sense usar els bytes que calgui.
- L'adreça inicial i la mida total del bloc d'activació han de ser múltiples de 4[9].

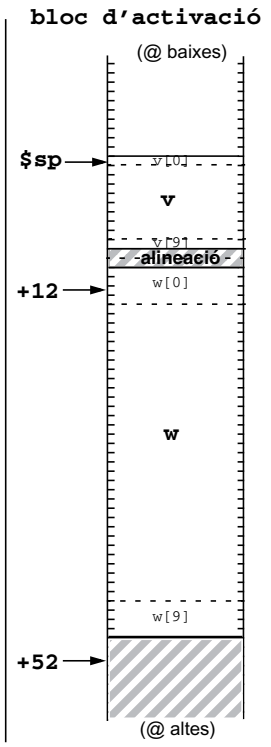
Més endavant, a l'apartat 7.5 ampliarem aquestes regles i donarem una definició completa del bloc d'activació.

8. S'estudiaran en el Tema 5

9. Per simplicitat, en EC no considerarem mai un bloc d'activació amb variables locals de mida 8 bytes (de tipus *long long* o *double*), que requeririen que aquest tingués adreça inicial i mida múltiples de 8.

EXEMPLE 28: Traduir la sentència visible de la següent funció:

```
char func(int i)
{
     char v[10];
     int w[10], k;              /* guardem k en $t0 */
     ...                        /* inicialitzacions */
     return v[w[i]+k];
}
```



En MIPS seria:

```
func:
     # reservem espai a la pila per als vectors v i w
     addiu  $sp, $sp, -52
     ...
     # @w[i] = ($sp + 12) + i*4
     sll    $t4, $a0, 2                 # i*4
     addu   $t4, $t4, $sp               # $sp + i*4
     lw     $t4, 12($t4)                # w[i]
     addu   $t4, $t4, $t0               # w[i] + k

     # @v[$t4] = $sp + $t4*1
     addu   $t5, $sp, $t4               # $sp + $t4*1
     lb     $v0, 0($t5)                 # retorna v[...]

     # alliberem l'espai de pila
     addiu  $sp, $sp, 52
     jr     $ra
```

7.4 Subrutines multinivell: preservar el context del programa

Diem que una subrutina és *multinivell* (*non-leaf*) si el seu codi conté alguna crida a subrutina (les subrutines recursives en són un cas particular). Anàlogament, diem que una subrutina és *uninivell* (*leaf*) si no conté cap crida.

El *context* d'execució d'una subrutina és el conjunt de totes les seves dades locals (incloent-hi paràmetres, adreça de retorn, punter de pila i càlculs intermedis), tant les que es guarden a la pila com en registres. En general, el context d'una subrutina no és visible, ni per tant accessible, a cap altra part del programa, excepte en el cas que la subrutina passi com a paràmetre a una altra subrutina un punter apuntant a una d'aquestes dades.

Si totes les dades del context es guardessin al bloc d'activació a la pila, una subrutina pot evitar fer modificacions en el context d'altres subrutines simplement restringint-se a accedir al seu propi bloc d'activació, ja que aquest té límits ben coneguts. Però si acceptem, per raons d'eficiència, que les variables locals escalars, els paràmetres, l'adreça de retorn, el punter de pila i altres càlculs temporals es poden guardar en registres, ¿com sabrà una subrutina quins registres contenen dades del context de la rutina que l'ha invocat i quins registres estan lliures?

EXEMPLE 29: Vegem el problema a través d'un exemple. Sigui la següent subrutina:

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

a) Una traducció ERRÒNIA

```
multi:
     # a, b c ocupen $a0, $a1, $a2, segons les regles de l'ABI
     # decidim que d, e ocuparan els registres $t0, $t1
     addu   $t0, $a0, $a1       # d = a + b

     move   $a0, $a2            # passem el 1er paràmetre c
     move   $a1, $t0            # passem el 2on paràmetre d
     jal    mcm                 # crida a mcm
     move   $t1, $v0            # assignem el resultat a la variable e

     addu   $t2, $a2, $t0       # c + d
     addu   $v0, $t2, $t1       # return (c + d) + e
     jr     $ra
```

Aparentment, la traducció és correcta. El context de la funció *multi* està format pels registres que contenen paràmetres (`$a0`, `$a1`, `$a2`), variables locals (`$t0`, `$t1`) càlculs temporals (`$t2`), l'adreça de retorn (`$ra`) i el punter de pila (`$sp`). Però ¿què passa si la funció *mcm* modifica algun d'aquests registres? Per exemple, si *mcm* modifiqués el registre `$t0`, alteraria el valor de la variable local *d* i el resultat de *multi* seria incorrecte!

Òbviament, si la rutina que fa la crida i la rutina cridada fossin escrites pel mateix programador, ell ja sap quins registres guarden dades del context de cada rutina i pot evitar els conflictes. Però refiar-se d'aquest mètode, a part de ser propens a errors, no és vàlid per a la programació modular. Per tant, cal establir unes regles d'utilització dels registres per tal que les dues rutines puguin ser escrites per programadors independents, amb l'únic requisit de conèixer la capçalera o interfície de cada subrutina.

b) Primera aproximació: una solució simple, però costosa (que no adoptarem)

Una possible solució per assegurar que no modifiquem el context de la rutina que ens invoca sense conèixer quins registres en formen part, consisteix a obligar que "tota subrutina, abans de retornar, deixi tots els registres en el mateix estat que tenien quan ha estat invocada (excepte `$v0`, que conté el resultat)". Això afecta sols a aquells registres que són modificats dins la subrutina. Pel que fa al punter de pila, no s'altera si alliberem tants bytes com els que reservem. Per a la resta de registres, cal afegir a la subrutina un pròleg i un epíleg: el pròleg guarda ("salva") a la pila l'estat inicial d'aquells registres que després seran modificats per la subrutina; l'epíleg recarrega ("restaura") de la pila el valor prèviament salvat, és a dir el valor inicial que tenien abans d'executar la subrutina.

Per exemple, la rutina *multi* modifica els registres `$a0`, `$a1`, `$t0`, `$t1`, `$t2`, i `$ra`. El pròleg salva a la pila els seus valors inicials i l'epíleg els restaura abans de retornar:

```
multi:
     addiu  $sp, $sp, -24
     sw     $a0, 0($sp)
     sw     $a1, 4($sp)
     sw     $t0, 8($sp)
     sw     $t1, 12($sp)
     sw     $t2, 16($sp)
     sw     $ra, 20($sp)

     addu   $t0, $a0, $a1
     move   $a0, $a2
     move   $a1, $t0
     jal    mcm
```

pròleg

```
     move   $t1, $v0
     addu   $t2, $a2, $t0
     addu   $v0, $t2, $t1

     lw     $a0, 0($sp)
     lw     $a1, 4($sp)
     lw     $t0, 8($sp)
     lw     $t1, 12($sp)
     lw     $t2, 16($sp)
     lw     $ra, 20($sp)
     addiu  $sp, $sp, 24
     jr     $ra
```

epíleg

L'anterior resultat és correccte, no obstant aquesta regla és excessivament estricta ja que obliga a una subrutina a salvar i restaurar tots els registres modificats, i en molts casos salva i restaura registres innecessàriament, simplement perquè ignora si aquests contenen o no alguna dada dels contextos de les rutines que l'han invocat.

c) Regla de l'ABI de MIPS que estableix els registres a salvar/restaurar

A fi de reduir el nombre de registres que cal salvar al principi i restaurar al final, MIPS ha optat pel conveni de dividir els registres en dos grups: uns són els registres que tota rutina ha de preservar obligatòriament i els anomenarem *registres segurs* ("saved" registers), i que són `$s0-$s7`, `$f20-$f31`, `$sp` i `$ra`[10]; mentre que els altres són els registres que tota subrutina pot modificar lliurement i que anomenarem *temporals*, i que són `$t0-$t9`, `$v0-$v1`, `$a0-$a3`, i `$f0-$f19`[11]. Llavors, per assegurar que una subrutina no altera el context de la rutina que l'ha invocat, és suficient observar la següent regla:

- **REGLA: "Quan una subrutina retorna, ha de deixar els registres segurs en el mateix estat que tenien quan ha estat invocada".**

D'aquesta regla en deriven dues conseqüències importants per al programador:

- CONSEQÜÈNCIA 1: Si es vol evitar que una subrutina cridada modifiqui una dada del propi context, cal guardar-la en els registres segurs `$s0-$s7` o `$f20-$f31`.
- CONSEQÜÈNCIA 2: Per evitar modificar dades del context de la rutina que ens ha invocat s'han de salvar a la pila al principi i restaurar al final tots aquells registres segurs que es modifiquin al llarg de la subrutina.

Com que guardar una dada en registre segur comporta feina extra de salvar-lo i restaurar-lo, sols es guarda una dada en un registre segur si és indispensable. Llavors, ¿quina dada és INDISPENSABLE guardar en registre segur? Doncs sols aquella dada que es faci servir DESPRÉS d'una crida i que el seu valor hagi estat generat ABANS d'aquesta crida (els paràmetres es consideren tots generats abans de la crida, ja que els genera el codi que invoca la funció). Òbviament, les subrutines uninivell tendiran a no fer servir registres segurs (excepte que tinguin tantes variables locals que no càpiguen en temporals).

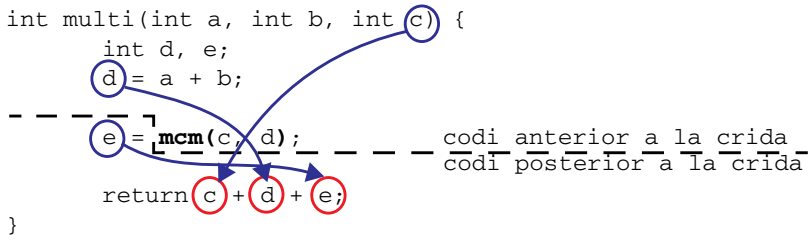
d) Cas pràctic: programar una subrutina multinivell, en tres passos

En primer lloc: Identificar quines dades locals (variables, paràmetres o càlculs intermedis) requereixen ser preservades de possibles modificacions per part de les subrutines cridades. Es tracta de comprovar en el codi d'alt nivell, per a cada una de les dades que es guarden en registres i que es fan servir DESPRÉS d'una crida, si el seu valor ha estat generat ABANS d'aquesta crida. Si és així, caldrà que la dada es guardi en un registre segur (CONSEQÜÈNCIA 1).

Per exemple, en la subrutina *multi* (es mostra a sota), veiem que les dades en registres que es fan servir després de la crida són *c*, *d* i *e* (en vermell). Veiem que *c* és el paràmetre, i el seu valor l'ha generat la subrutina que ha invocat a *multi* (en blau) molt abans de la crida a *mcm*. Veiem que *d* és la variable local, i el seu valor ha estat assignat (en blau) per la suma *a+b* just abans de la crida a *mcm*. Veiem també que *e* és l'altra variable

10. També són registres segurs $gp i $fp, però no els usarem en aquest curs.
11. També són temporals els registres $at, $k0, $k1, però l'ús de $at està reservat a l'expansió de macros i els registres $k0-$k1 estan reservats al S.O..

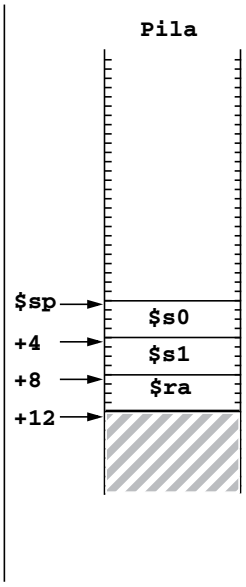
local, i el seu valor ha estat assignat (en blau) al resultat de *mcm*, per tant just després de la crida. Així doncs, solament en el cas de *c* i *d* els seus valors han estat generats abans de la crida i usats després, (les fletxes travessen la línia de punts) i caldrà guardar-los en registres segurs:



En segon lloc: Assignem registres a totes les dades locals. El paràmetre *c* està inicialment guardat en `$a2`, i caldrà copiar-lo a `$s0`. La variable local *d* no té un valor inicial, l'escriurem al registre `$s1` en el moment que calculem el seu valor, a la primera suma. La variable local *e* la podem guardar en un registre temporal, p.ex. `$t0`, i així no ens cal salvar-lo a la pila. Per al càlcul intermedi *c+d* també usarem un registre temporal, p.ex. `$t1`.

En tercer lloc: Escrivim el codi de la subrutina amb aquestes assignacions, i li afegim un pròleg i un epíleg (CONSEQÜÈNCIA 2) per salvar i restaurar els registres segurs modificats `$s0` i `$s1`, així com el registre `$ra` (que és modificat per la instrucció `jal`):

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

    move   $s0, $a2               # Copiem c en $s0
    addu   $s1, $a0, $a1          # Escrivim d en $s1
    move   $a0, $s0
    move   $a1, $s1
    jal    mcm
    move   $t0, $v0               # Escrivim e en $t0
    addu   $t1, $s0, $s1          # Escrivim c+d en $t1
    addu   $v0, $t1, $t0

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



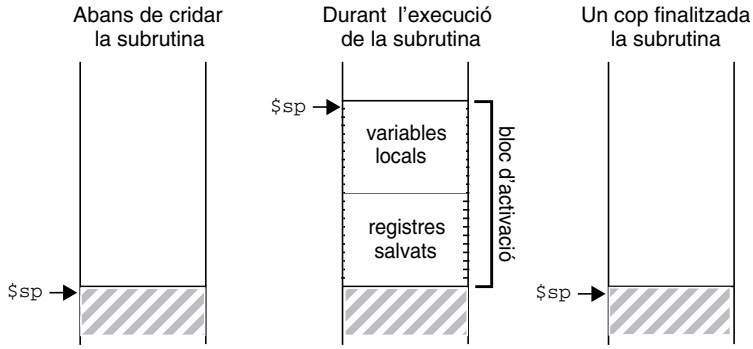
7.5 Estructura del bloc d'activació

Recapitulant les idees exposades fins ara, el bloc d'activació d'una subrutina està ubicat a la pila i pot incloure les següents informacions:

- **Variables** locals de tipus estructurats. O bé de tipus escalars, si resulta que no hi ha prou registres on guardar-les o si se'ls aplica l'operador unari `&`.
- Valors inicials de **registres** segurs, salvats al principi de la subrutina, en concret d'aquells que es modificaran al llarg de la subrutina.
- Valors de registres temporals salvats a la pila per alliberar-los temporalment, si ens quedem sense registres temporals lliures suficients per a càlculs intermedis.

La disposició dels diversos elements dins del bloc d'activació acostuma a determinar-se de forma estricta en l'ABI, de manera que els programes depuradors puguin analitzar-ne el contingut amb precisió (vegeu el diagrama a sota):

- **Posició**: Les variables locals ocupen les adreces més baixes del bloc d'activació a partir del cim de la pila (part alta del diagrama). Els registres salvats ocupen les adreces més altes, a continuació de les variables (part baixa del diagrama).
- **Ordenació**: Les variables locals es col·loquen a la pila en el mateix ordre en què apareixen declarades en el text, començant per l'adreça més baixa del bloc (que és la del cim de la pila, a on apunta $sp un cop reservat l'espai per al bloc). Els registres salvats no segueixen cap ordre en particular entre ells.
- **Alineació**: De manera anàloga a com s'alineen les variables globals, cada una de les variables locals s'alinea segons el seu tipus (si és escalar) o segons el tipus dels seus elements (si és estructurada, com els vectors i matrius), deixant lliures els espais intermedis necessaris. Els registres salvats aniran alineats a adreces múltiples de 4. La mida total del bloc d'activació i el valor del registre `$sp` han de ser sempre múltiples de 4[12].



7.6 I per acabar, un exemple de revisió, traduït pas a pas

EXEMPLE 30: Analitzar i després traduir a MIPS la subrutina *exemple*:

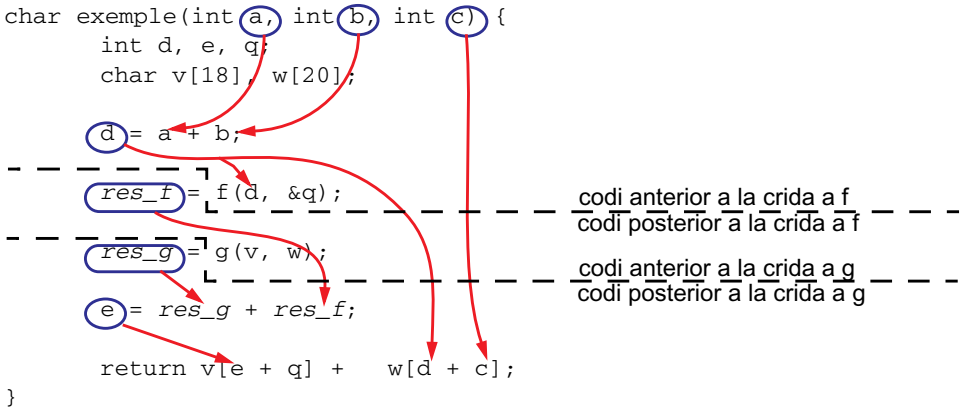
```
int f(int m, int *n);
int g(char *y, char *z);
char exemple(int a, int b, int c) {
       int d, e, q;
       char v[18], w[20];

       d = a + b;
       e = f(d, &q) + g(v, w);
       return v[e + q] + w[d + c];
}
```

En primer lloc decidirem quines variables locals es guarden al bloc d'activació: seran els vectors *v* i *w*, però també l'escalar *q*, ja que apareix precedit de l'operador unari '&' en la crida a *f*. Es guardaran en el mateix ordre que apareixen declarats: *q*, *v*, *w*.

12. Per simplicitat, en l'ABI estudiat en EC no considerarem mai un bloc d'activació que contingui variables locals de mida 8 bytes (tipus *double* o *long long*) i per tant considerarem que la mida i adreça inicial del bloc d'activació ha de ser simplement un múltiple de 4. Sense aquesta restricció, com succeeix en la vida real, la mida i alineació del bloc d'activació haurien de ser múltiples de 8.

A continuació, cal fer una inspecció del codi en alt nivell per determinar quins dels valors cal preservar en registres segurs per evitar que siguin modificats per les subrutines cridades (veure CONSEQÜÈNCIA 1). Possibles candidats són els valors que es guarden en registres. Descartant *q*, *v*, i *w* que es guarden a la pila, queden: (1) paràmetres *a*, *b*, *c*, (2) variables locals de tipus escalars *d*, *e*, i (3) càlculs intermedis de subexpressions, com ara els resultats de les crides a *f* i *g*. Tots ells poden ser candidats a usar registres segurs.

A continuació determinem quins d'aquests candidats són dades que es necessiten després de la crida, i el valor de les quals ha estat generat abans de la crida. A fi de visualitzar millor quin codi és anterior i posterior a cada crida, hem desglossat (veure el codi a sota) les dues crides en línies diferents afegint-hi les variables temporals fictícies *res_f* i *res_g* per al resultat de cada funció (i que un cop traduït el codi es guardaran en registres), i hem separat amb línies de punts el codi anterior i posterior a cada crida. Llavors hem marcat amb cercles blaus els punts on es generen els valors, i amb cercles vermells els punts on són utilitzats, i els hem unit amb fletxes. Així és fàcil veure quines fletxes travessen les "fronteres" de cada crida, indicant doncs que són valors a preservar en registres segurs. Un detall addicional en aquest exemple és que l'ordre en què es fan les dues crides no està establert pel llenguatge C (si està ben programat, ha de ser indiferent), de manera que nosaltres suposarem aquí que primer cridem a *f* i després a *g*.



Analitzem en detall cada un dels candidats a ser preservat en registre segur:

- Els paràmetres *a*, *b* i *c* estan definits des del principi, ja que els calcula el codi que ha invocat a la funció *exemple*. En el cas d'*a* i *b,* només s'usen en la primera suma, abans de les dues crides a *f* i *g*, per tant no cal preservar-los. En canvi, *c* es necessita per a la sentència final, després de les crides, i s'ha de preservar.
- La variable *d* es calcula en la primera sentència, abans de les crides, i es necessita com a paràmetre de *f* (abans de la crida) però també per a la sentència final, després de les dues crides, per tant s'ha de preservar.
- El resultat intermedi *res_f* es genera abans de la crida a *g*, i es necessita després, per calcular la suma que assignarem a *e*, per tant s'ha de preservar. En canvi el resultat *res_g* es genera després de les dues crides, per tant no s'ha de preservar.
- La variable *e* no es calcula fins després de les crides, tampoc s'ha de preservar.

Així doncs, concloem que hem de preservar *c*, *d*, i el resultat de *f* (primera crida) guardant-los als registres segurs `$s0`, `$s1` i `$s2` respectivament:

- Abans de la crida a *f,* copiarem `$a2` (paràmetre *c*) en `$s0`.
- Quan escrivim el resultat de la primera suma en *d*, ho farem en `$s1`.
- Abans de la segona crida, copiarem `$v0` (resultat de *f*) en `$s2`.

En canvi, *a* i *b* seguiran ocupant `$a0` i `$a1`, i la variable *e* l'escriurem en `$t0`.

A continuació escriurem aquestes assignacions sobre el codi en C, per tenir-les presents mentre escrivim en assemblador (anotacions en vermell i en negreta):

```
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]];
}
```

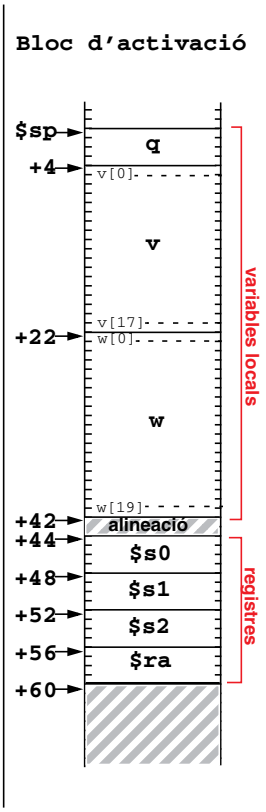
a → $a0, b → $a1, c → $a2 → $s0; d → $s1, e → $t0; f(d, &q) → $v0 → $s2

Finalment, establim el bloc d'activació: en primer lloc *q*, *v* i *w* ocupant en total 44 bytes (4 de *q*, 18 de *v*, 20 de *w*, i 2 més per alinear els registres que van a sota). Just a continuació, se salvaran els registres `$s0`, `$s1`, `$s2` i `$ra`, ocupant 16 bytes més. En total, el bloc d'activació es compon de 60 bytes. El codi MIPS quedarà definitivament així:

```
exemple:
    # pròleg: salvar registres segurs a la pila
    addiu  $sp, $sp, -60
    sw     $s0, 44($sp)
    sw     $s1, 48($sp)
    sw     $s2, 52($sp)
    sw     $ra, 56($sp)

    # primera sentència
    move   $s0, $a2            # copiar c a $s0
    addu   $s1, $a0, $a1       # d=a+b, en $s1

    # segona sentència
    move   $a0, $s1            # passem d
    move   $a1, $sp            # passem &q
    jal    f                   # cridem f
    move   $s2, $v0            # copiar res_f a $s2

    addiu  $a0, $sp, 4         # passem @v[0]
    addiu  $a1, $sp, 22        # passem @w[0]
    jal    g                   # cridem g
    addu   $t0, $s2, $v0       # e = res_f + res_g

    # tercera sentència
    lw     $t1, 0($sp)         # q
    addu   $t1, $t0, $t1       # e + q
    addu   $t1, $t1, $sp       # (e + q) + $sp
    lb     $t2, 4($t1)         # v[e + q]

    addu   $t1, $s1, $s0       # d + c
    addu   $t1, $t1, $sp       # (d + c) + $sp
    lb     $t3, 22($t1)        # w[d + c]
    addu   $v0, $t2, $t3       # return v[e+q]+w[d+c]

    # epíleg: restaurar registres de la pila
    lw     $s0, 44($sp)
    lw     $s1, 48($sp)
    lw     $s2, 52($sp)
    lw     $ra, 56($sp)
    addiu  $sp, $sp, 60
    jr     $ra
```

pròleg

epíleg



2.8

8. Estructura de la memòria

En general, les variables d'un programa poden tenir un mode d'emmagatzemament estàtic o dinàmic. Diem que el mode d'emmagatzemament és estàtic quan la variable es guarda al mateix lloc durant tot el temps d'execució del programa. En C, pertanyen a aquesta categoria, per exemple, les variables externes (o globals), que es defineixen a la secció `.data` d'un programa en assemblador. La regió de memòria dedicada a l'emmagatzemament estàtic té una mida fixa per a cada programa.

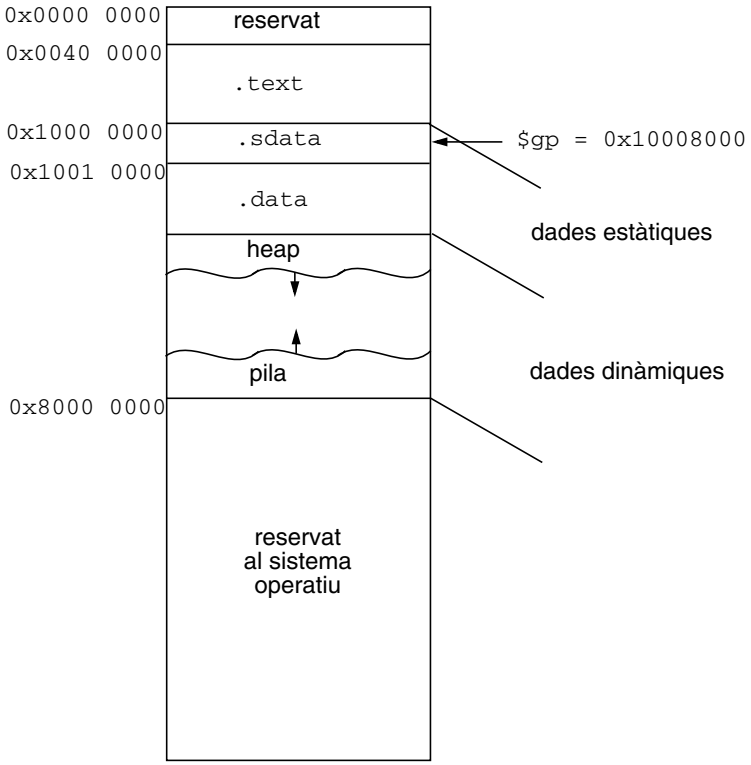
Diem que el mode d'emmagatzemament és dinàmic quan la variable es guarda en una àrea de memòria de manera temporal. Pertanyen a aquesta categoria, per exemple, les variables automàtiques (o locals) en C. Com hem vist en les seccions anteriors, algunes variables locals es poden guardar en registres, però la resta es guarden a la regió de memòria que anomenem *pila*.

Però les variables que solem anomenar pròpiament com a *dinàmiques* són aquelles on la reserva i alliberament de l'espai de memòria la fa explícitament el programador per mitjà de funcions de la llibreria estàndar tals com *malloc* o *free*. Com que poden invocarse en qualsevol punt del programa, aquest espai no es reserva i s'allibera de forma ordenada com succeeix amb la pila. Per tant, la gestió d'aquest espai de memòria requereix una estructura de dades que anomenem *heap*. Aquestes dues funcions fan d'interfície amb el sistema operatiu, que és qui gestiona realment la reserva de l'espai de memòria al heap. Per crear una variable dinàmica en C cal declarar un punter (local o global, segons l'abast que hagi de tenir), cridar a la funció *malloc*, i assignar al punter l'adreça que retorna la crida com a resultat.

EXEMPLE 31: Vegem els tres tipus de variables esmentats en un programa:

```
int gvec[100];                                  → globals: dades estàtiques
int *pvec;

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

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

El següent diagrama mostra les diferents àrees de memòria, en MIPS:



La regió dedicada a dades estàtiques conté diverses seccions. La secció `.data` és on solem guardar les variables globals. L'accés a una d'aquestes variables comporta sovint que el programa hagi d'inicialitzar un registre amb la seva adreça de 32 bits. Per exemple:

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

Per tal d'optimitzar l'accés, els compiladors solen usar un punter (*global pointer*) que apunti a una adreça fixa, i que s'emmagatzema en el registre $gp (registre $28) durant tot el programa. D'aquesta manera, totes les dades que estiguin a distàncies pròximes d'aquesta adreça es poden accedir de forma simple amb desplaçaments relatius a $gp.

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

Òbviament, com que el desplaçament d'una instrucció de memòria es codifica amb un enter de 16 bits, per mitjà del global pointer sols podem accedir a un màxim de 2^{16} bytes, els que formen l'anomenada secció `.sdata` (*small data*). Aquesta secció de 64KB es destina a guardar-hi variables globals d'ús freqüent de petites dimensions i està situada a l'adreça 0x10000000. El global pointer s'inicialitza apuntant al seu punt mig, a l'adreça 0x10008000. La secció `.data` que ve a continuació ja comença a l'adreça 0x10010000.

La regió dedicada a dades dinàmiques té 2 seccions, el *heap* i la *pila*, i se situa a continuació de les dades estàtiques. Com que les dues seccions poden créixer dinàmicament, ocupen els dos extrems d'aquesta regió a fi de no fragmentar l'espai de memòria lliure: el heap ocupa l'espai a partir de l'adreça més baixa i creix cap a adreces més altes, mentre que la pila ocupa l'espai a partir de l'adreça més alta i creix cap a adreces baixes.

Les adreces més enllà de la pila es caracteritzen per tenir el bit de més pes a u, i estan reservades a guardar el codi i les dades del sistema operatiu.

2.12, B2-B5

9. Compilació, assemblatge, enllaçat i càrrega

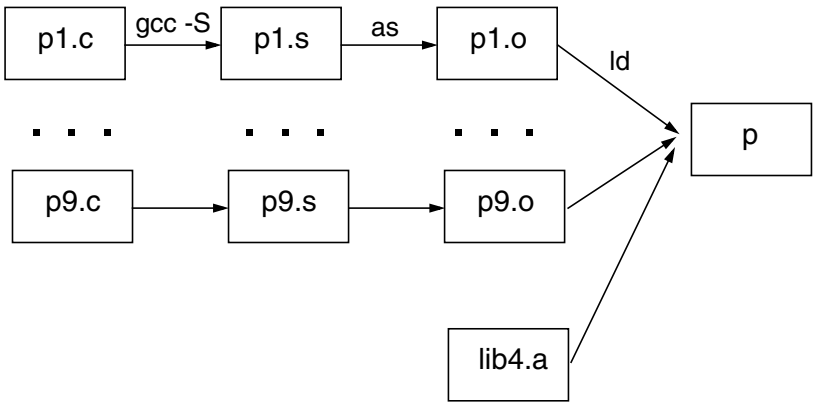
En general, el procés pel qual un programa escrit en un llenguatge d'alt nivell com C es converteix en un programa que s'executa en el computador consta de 4 passos:

- Compilació: El compilador (compiler) és un programa que pren com a entrada el codi font en C i el tradueix a llenguatge assemblador.
- Assemblatge: L'assemblador (assembler) és un programa que pren com a entrada el fitxer en llenguatge assemblador i el tradueix a codi màquina en un fitxer objecte.
- Enllaçat: L'enllaçador (linker) combina múltiples fitxers objecte i possiblement fitxers de biblioteca en un sol fitxer executable.
- Càrrega: El carregador (loader) és un programa del sistema operatiu que llegeix un fitxer executable i copia els seus elements a la memòria del computador per a ser executat.

9.1 Compilació separada

Quan es construeixen programes de grans dimensions, resulta convenient separar-ne el codi i les dades en mòduls per diverses raons: per una part, per estructurar racionalment un projecte complex i abordar-lo de manera eficient desenvolupant les diverses parts a càrrec d'equips independents de programadors. Per una altra, per poder reutilitzar codi, escrivint biblioteques de funcions que puguin ser usades en més d'un programa.

En aquests programes resulta desitjable que si fem una petita modificació en un sol dels mòduls no hàgim de compilar i assemblar tots els mòduls sinó solament el que s'ha modificat. Però per a això cal que els mòduls es compilin i assemblin per separat generant fitxers objecte (o de biblioteca) independents que més tard s'enllaçaran per crear l'executable. D'aquesta manera, un cop compilat i assemblat el mòdul modificat, ja sols cal enllaçar-lo amb la resta de fitxers objecte i/o de biblioteca.



9.2 Assemblatge

Una de les tasques de l'assemblador és expandir les macros en les seqüències d'instruccions equivalents, i més tard traduir cada instrucció al seu corresponent codi binari, d'acord amb el seu format.

A més a més, el llenguatge assemblador usa etiquetes per a referir-se a adreces, ja siguin dins el codi o en les dades. En assemblador, aquestes etiquetes poden ser usades dins el fitxer fins i tot abans de ser definides (referència anticipada). Per tant, el programa assemblador ha de llegir tot el codi font en una primera passada, anotant en una taula (taula de símbols) la correspondència entre cada etiqueta i la seva adreça. En la segona passada al fitxer, ja es podran codificar les instruccions.

No obstant, les adreces d'etiquetes que assigna l'assemblador no són pròpiament les adreces definitives en memòria, ja que no coneix el codi i dades que puguin estar definits en altres mòduls. Les adreces definitives les assignarà l'enllaçador. Durant l'assemblatge, les adreces es representen simplement com *offsets* o distàncies en bytes de la posició de l'etiqueta en relació a la secció on està definida. Per a codificar instruccions de salt relatiu al PC (*beq* i *bne*) les adreces així definides són suficients, ja que aquestes instruccions codifiquen la distància a saltar, la qual no depèn d'on s'ubiqui el codi en memòria finalment. Però en canvi, algunes instruccions necessiten l'*adreça absoluta*: és el cas dels salts en mode pseudodirecte (*j* i *jal*), i també la pseudoinstrucció *la* (o les instruccions en què s'expandeix). Així doncs, al camp de l'adreça en aquestes instruccions s'hi escriu provisionalment un zero, ja que no pot codificar-se fins que l'enllaçador assigni una adreça definitiva a l'etiqueta. El procés de completar la codificació d'aquestes instruccions amb les adreces definitives es coneix com *reubicació* i la fa l'enllaçador. L'assemblador ha d'elaborar una llista amb la posició de cada instrucció a reubicar, el seu tipus, i l'adreça provisional a què fa referència.

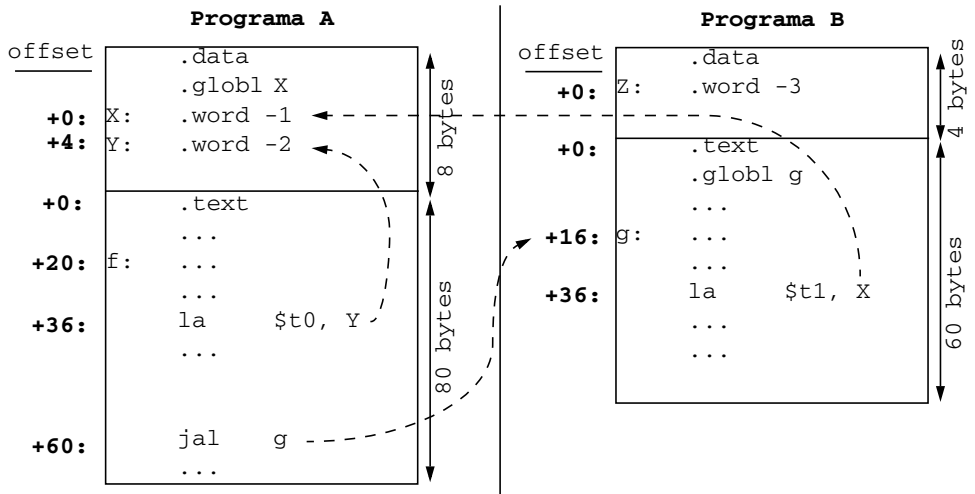
Si el programa consta de més d'un mòdul, pot passar que alguna instrucció faci referència a una etiqueta que s'ha definit en un altre mòdul. En aquest cas, l'assemblador no pot codificar-la (hi codifica un zero), però pren nota de la posició d'aquesta instrucció i de l'etiqueta de la qual depèn, perquè sigui l'enllaçador qui s'encarregui de codificar-la. Al final del fitxer, l'assemblador haurà generat una llista de referències externes no-resoltes.

Si alguna de les etiquetes del fitxer ha de ser referenciada per un altre mòdul, cal que s'hagi declarat global (amb la directiva `.globl`). L'assemblador genera una taula de símbols globals amb les seves corresponents adreces provisionals, que li servirà a l'enllaçador per resoldre referències en altres mòduls.

En UNIX, el fitxer objecte generat per l'assemblador consta dels següents components:

- Capçalera, amb informació sobre la ubicació dels restants components del fitxer
- Secció de text amb el codi màquina de les subrutines del mòdul
- Secció de dades estàtiques, amb les dades globals definides al mòdul
- Llista de reubicació (posició de la instrucció, tipus, i adreça provisional)
- Taula de símbols globals, i Llista de referències externes no-resoltes
- Informació de depuració (per exemple, números de línia del codi font a què correspon cada instrucció).

EXEMPLE 32: Suposem que assemblem els programes A i B següents. Els respectius fitxers objecte inclouran la informació de reubicació, de símbols globals i de referències no-resoltes que apareix a sota de cada un:



Reubicar:
• posició +36 (text), tipus la, offset +4 (data)

Símbols globals:
• posició +0 (data), label="X"

Referències no-resoltes:
• posició +60 (text), tipus jal, label="g"

Reubicar:
•

Símbols globals:
• posició +16 (text), label="g"

Referències no-resoltes:
• posició +36 (text), tipus la, label="X"

9.3 Enllaçat (o muntatge o "linkatge")

En primer lloc l'enllaçador s'assegura que el programa no té cap etiqueta sense definir. Si n'hi ha alguna, busca la seva definició en els fitxers de biblioteca que tingui disponibles (del programa o del sistema), i si la troba afegeix el fitxer corresponent (aquest pot contenir altres referències externes que donin lloc a noves búsquedes). Si no troba la definició, l'enllaçat finalitza amb un error.

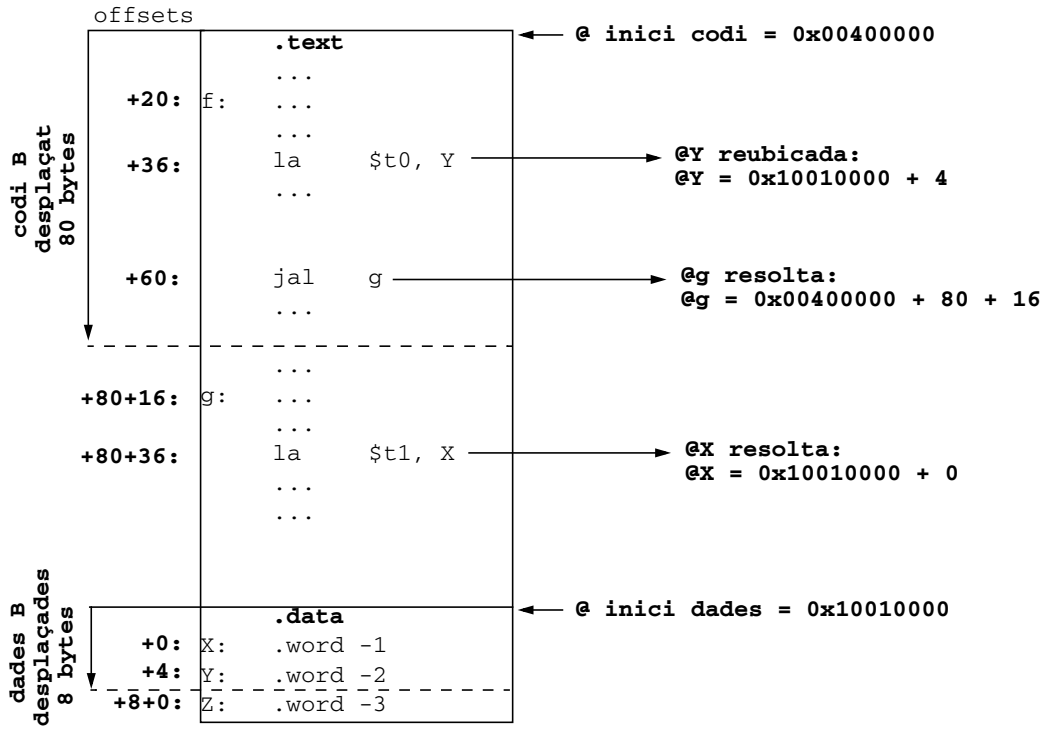
En segon lloc, ajunta el codi i les dades dels diversos mòduls. La majoria de seccions de codi i dades resulten desplaçades, de manera que s'assigna un desplaçament a cada secció. També s'assignen adreces definitives a totes les etiquetes que figuren en les taules de símbols globals, sumant-los-hi els desplaçaments de les seves corresponents seccions.

En tercer lloc, es procedeix a reubicar les instruccions amb adreces absolutes, les que figuren a les llistes de reubicació. L'adreça definitiva a codificar-hi s'obté sumant l'adreça inicial de la secció (0x00400000 per al `.text` o 0x10010000 per al `.data`, en MIPS) amb el desplaçament sofert per la secció al combinar els mòduls, i amb l'offset provisional dins la secció que havia assignat l'assemblador.

En quart lloc, es resolen les referències creuades no-resoltes, és a dir que es corregeixen les instruccions que contenen referències a etiquetes externes, consultant les adreces definitives a la taula de símbols globals. Si l'etiqueta en qüestió no es troba a la taula, l'enllaçador acaba amb un error ("unresolved reference to symbol X"): això pot ser degut a haver oblidat declarar l'etiqueta o haver oblidat la directiva `.globl`, o haver equivocat el nom de l'etiqueta, o haver oblidat enllaçar el fitxer que conté la declaració, etc.

Finalment, s'escriu al disc el fitxer executable, el qual té una estructura similar a un fitxer objecte, excepte que no conté informació de reubicació ni referències no-resoltes.

EXEMPLE 33: Suposem que enllacem els programes A i B de l'exemple anterior. El programa definitiu quedaria:



9.4 Càrrega en memòria

En aquest punt, el programa resideix en un fitxer del disc. El pas final per executar-lo consisteix a carregar-lo a la memòria principal del computador, i d'això se n'encarrega el "Loader" del sistema operatiu. En el cas de Unix, se segueixen els següents passos:

1. Llegir la capçalera del fitxer executable per determinar la mida dels segments de codi i dades.
2. Reservar memòria principal i crear un espai d'adreces suficientment gran per al codi i les dades.
3. Copiar les instruccions i dades del fitxer executable a la memòria.
4. Copiar a la pila els paràmetres del programa principal, expressats a la línia de comandes (si n'hi ha).
5. Inicialitzar els registres del processador, deixant $sp apuntant al cim de la pila.
6. Saltar a una rutina de "startup" dins l'executable, la qual copia els parametres (si n'hi ha) de la pila als registres de pas de paràmetres ($a0, etc.): en C són els paràmetres *argv* i *argc* de la funció "main". A continuació, la rutina startup fa una crida a "main".

Cal observar que quan la funció "main" retorna, se segueix executant el codi de la rutina "startup", el qual simplement invoca la crida al sistema *exit*[13]. La rutina *exit* del sistema operatiu allibera tots els recursos assignats al programa, com per exemple la memòria, i es queda en espera d'executar el següent programa.

13. Les crides al sistema s'estudiaran en el darrer tema del curs.