

# Tema 4. Matrius

Joan Manuel Parcerisa



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH  
Facultat d'Informàtica de Barcelona



# Matrius

- Multiplicació d'enters en MIPS
- Declaració i emmagatzematge
- Accés aleatori a un element
- Optimitzacions

# Multiplicació d'enters en MIPS

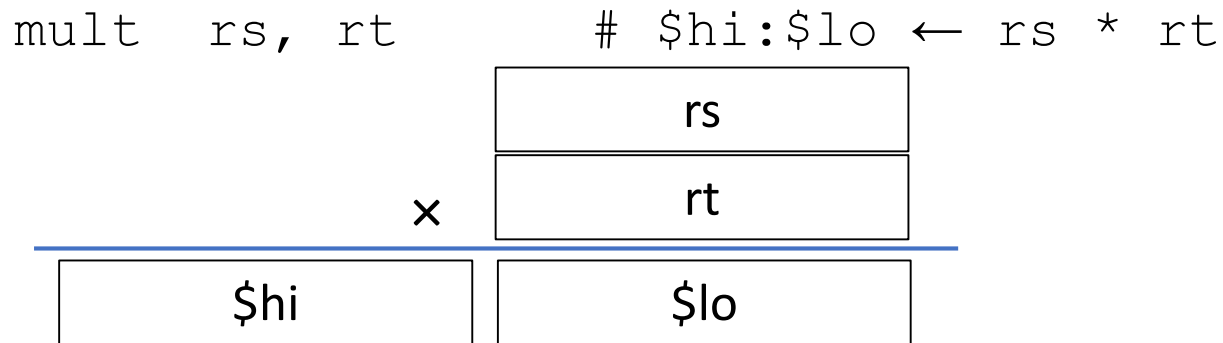
# Multiplicació d'enters en MIPS

- La multiplicació de 2 enters de  $n$  i  $m$  bits pot donar un resultat de  $n+m$  bits. En el cas extrem:

$$(2^n - 1) \times (2^m - 1) = 2^{n+m} - 2^n - 2^m + 1 < 2^{n+m}$$

# Multiplicació d'enters en MIPS

- La multiplicació de 2 enters de  $n$  i  $m$  bits pot donar un resultat de  $n+m$  bits
- En MIPS, 32 bits x 32 bits  $\rightarrow$  64 bits:

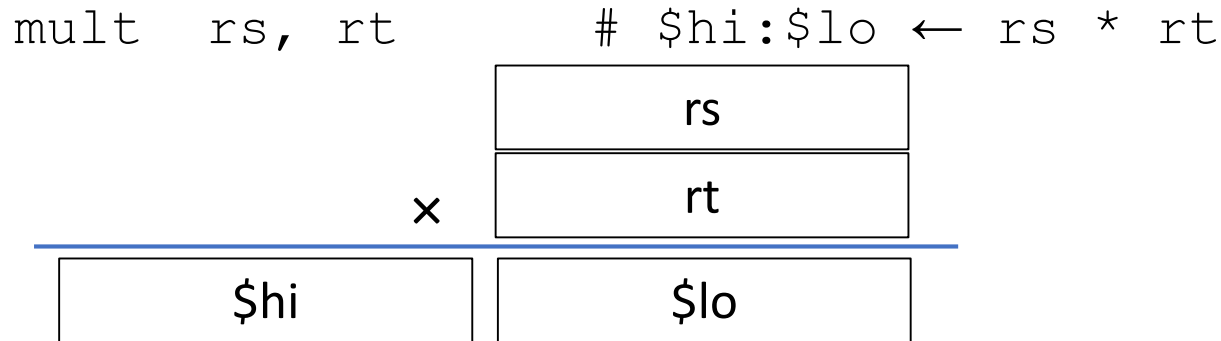


- El resultat es guarda als registres \$hi i \$lo

# Multiplicació d'enters en MIPS

- La multiplicació de 2 enters de  $n$  i  $m$  bits pot donar un resultat de  $n+m$  bits

- En MIPS, 32 bits x 32 bits  $\rightarrow$  64 bits:

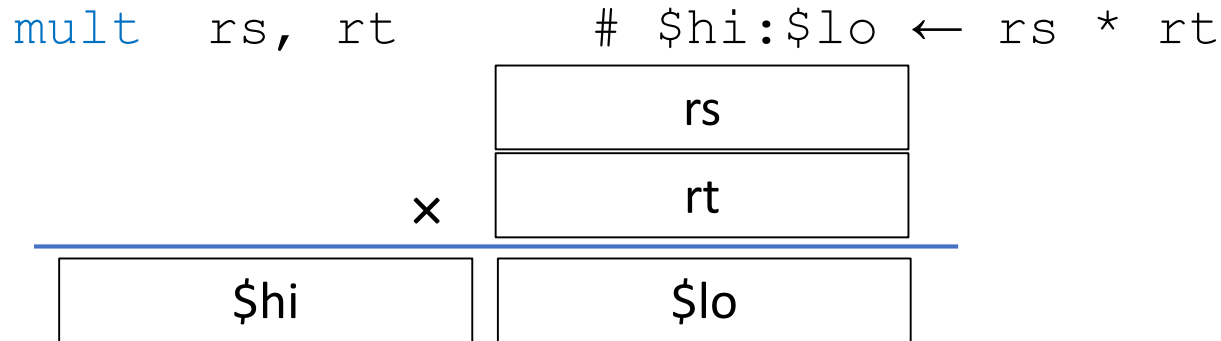


- El resultat es guarda als registres  $\$hi$  i  $\$lo$ 
  - Són registres especials: No es poden usar en les instruccions estudiades fins ara

# Multiplicació d'enters en MIPS

- La multiplicació de 2 enters de  $n$  i  $m$  bits pot donar un resultat de  $n+m$  bits

- En MIPS, 32 bits x 32 bits  $\rightarrow$  64 bits:



- El resultat es guarda als registres `$hi` i `$lo`
  - Són registres especials: No es poden usar en les instruccions estudiades fins ara
  - Per moure el resultat a registres "normals":

`mflo`   `rd`                      # `rd`  $\leftarrow$  `$lo`  
`mfhi`   `rd`                      # `rd`  $\leftarrow$  `$hi`

# Multiplicació d'enters en MIPS

- Suposant el següent codi en C

```
int    a, b, c;
```

```
...
```

```
c = a * b;
```

- El resultat **c té 32 bits**



# Multiplicació d'enters en MIPS

- Suposant el següent codi en C

```
int    a, b, c;
```

```
...
```

```
c = a * b;
```

- El resultat **c té 32 bits**
  - Trunquem els 64 bits del producte (possible overflow, si canvia el resultat)
  - Però en C s'ignora l'*overflow* del producte

# Multiplicació d'enters en MIPS

- Suposant el següent codi en C

```
int    a, b, c;
```

```
...
```

```
c = a * b;
```

- El resultat **c té 32 bits**
  - Trunquem els 64 bits del producte (possible overflow, si canvia el resultat)
  - Però en C s'ignora l'*overflow* del producte
- En MIPS (suposant *a*, *b*, *c* es guarden en \$t0, \$t1, \$t2) :

```
mult    $t0, $t1      # $hi:$lo ← $t0*$t1
```

```
mflo    $t2           # $t2 ← $lo
```

# Matrius

# Matrius. Declaració

- Una matriu és una agrupació multidimensional d'elements de tipus homogeni
  - Els elements s'identifiquen per un índex en cada dimensió
  - La primera fila o columna té índex 0
  - Estudiarem matrius de 2 dimensions

# Matrius. Declaració

- Una matriu és una agrupació multidimensional d'elements de tipus homogeni
  - Els elements s'identifiquen per un índex en cada dimensió
  - La primera fila o columna té índex 0
  - Estudiarem matrius de 2 dimensions
- Exemple: la matriu *mat*, de mida NF x NC

$$\text{mat[NF][NC]} = \begin{vmatrix} \text{mat}[0][0] & \text{mat}[0][1] & \dots & \text{mat}[0][\text{NC}-1] \\ \text{mat}[1][0] & \text{mat}[1][1] & \dots & \text{mat}[1][\text{NC}-1] \\ \dots & \dots & \dots & \dots \\ \text{mat}[\text{NF}-1][0] & \text{mat}[\text{NF}-1][1] & \dots & \text{mat}[\text{NF}-1][\text{NC}-1] \end{vmatrix}$$

# Matrius. Declaració

- En C (les dimensions han de ser constants o literals):

...

```
int mat[NF][NC];
```

```
int mit[2][3] = {{-1, 2, 0}, {1, -12, 4}};
```

# Matrius. Declaració

- En C (les dimensions han de ser constants o literals):

...

```
int mat[NF][NC];
```

```
int mit[2][3] = {{-1, 2, 0}, {1, -12, 4}};
```

- En MIPS:

```
.data
```

...

```
.align 2
```

```
mat: .space NF*NC*4
```

```
mit: .word -1, 2, 0, 1, -12, 4
```

# Matrius. Declaració

- En C (les dimensions han de ser constants o literals):

...

```
int mat[NF][NC];
```

```
int mit[2][3] = {{-1, 2, 0}, {1, -12, 4}};
```

- En MIPS:

```
.data
```

...

```
.align 2
```

```
mat: .space NF*NC*4
```

```
mit: .word -1, 2, 0, 1, -12, 4
```

- O bé:

```
mit: .word -1, 2, 0
```

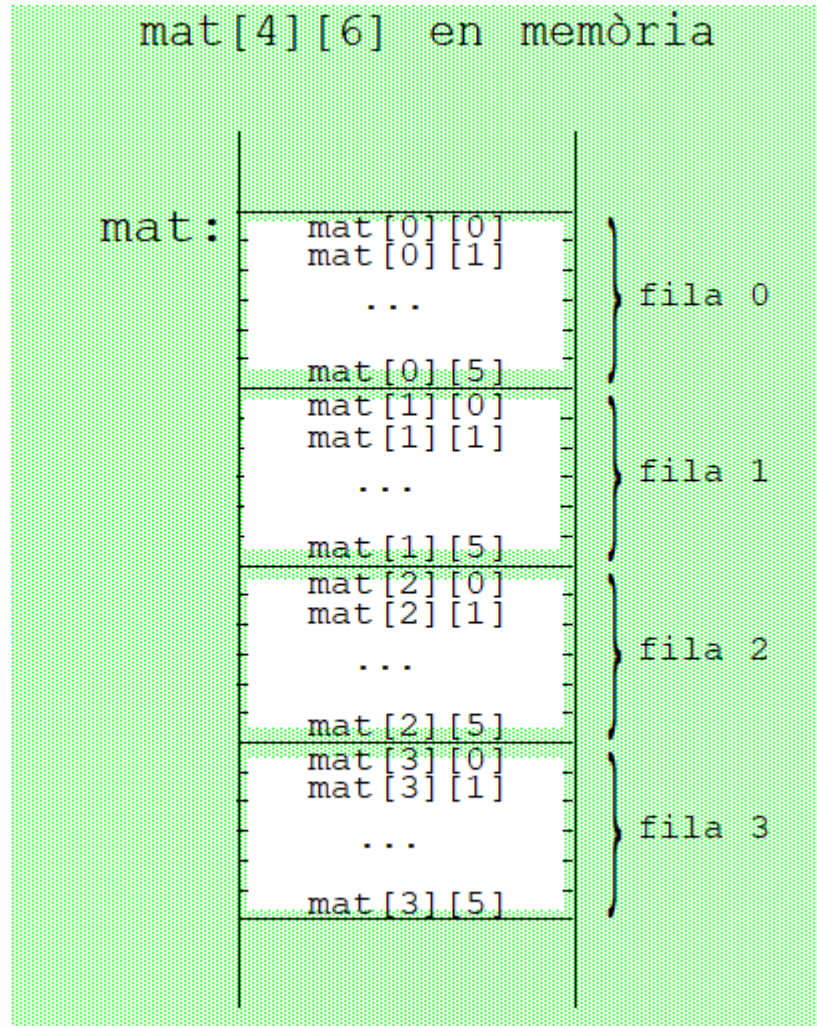
```
.word 1, -12, 4
```

← És més "visual"!



# Matrius. Emmagatzematge en memòria

- En C, les matrius s'emmagatzemen **per files**



# Accès aleatori a un element

# Accés *aleatori* a un element

- Sigui `mat` una matriu d'elements de mida  $T$
- Per accedir a l'element de la fila  $i$ , columna  $j$

`mat[i][j]`

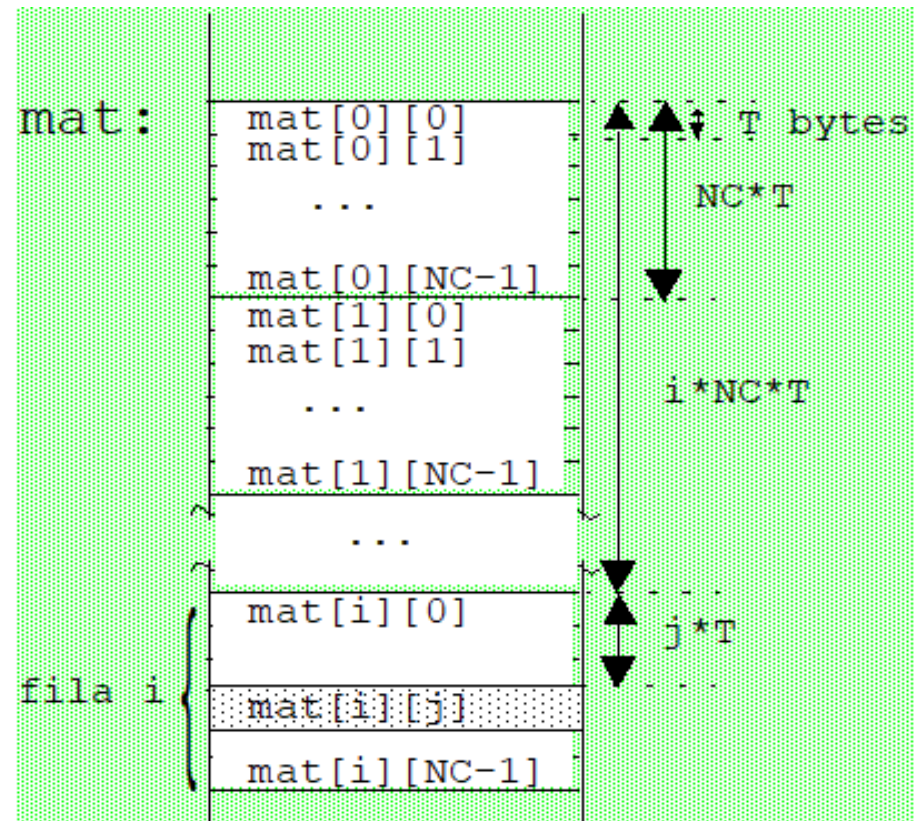
# Accés *aleatori* a un element

- Sigui `mat` una matriu d'elements de mida  $T$
- Per accedir a l'element de la fila  $i$ , columna  $j$

`mat[i][j]`

- Hem de calcular la seva adreça en memòria, així:

```
@mat[i][j] =  
    = mat + i*NC*T + j*T  
    = mat + (i*NC + j)*T
```



# Accés *aleatori* a un element

- Exemple: traduir a MIPS la sentència del requadre
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

```
int mat[NF][NC];  
void func() {  
    int i, j, k;  
    ...  
    k = mat[i][j];  
}
```

# Accés *aleatori* a un element

- Exemple: traduir a MIPS la sentència del requadre
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

```
int mat[NF][NC];  
void func() {  
    int i, j, k;  
    ...  
    k = mat[i][j];  
}
```

- Hem de calcular la següent adreça:

```
@mat[i][j] = mat + (i*NC + j)*4
```

# Accés *aleatori* a un element

- Traduïm a MIPS la sentència `k=mat[i][j];`
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

`@mat[i][j] = mat + (i*NC + j)*4`

```
la    $t3, mat
li    $t4, NC
mult  $t4, $t0
mflo  $t4
```

# Accés *aleatori* a un element

- Traduïm a MIPS la sentència `k=mat[i][j];`
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

`@mat[i][j] = mat + (i*NC + j)*4`

`la $t3, mat`

`li $t4, NC`

`mult $t4, $t0`

`mflo $t4`

`addu $t4, $t4, $t1`



# Accés *aleatori* a un element

- Traduïm a MIPS la sentència `k=mat[i][j];`
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

`@mat[i][j] = mat + (i*NC + j)*4`

`la $t3, mat`

`li $t4, NC`

`mult $t4, $t0`

`mflo $t4`

`addu $t4, $t4, $t1`

`sll $t4, $t4, 2`

# Accés *aleatori* a un element

- Traduïm a MIPS la sentència `k=mat[i][j];`
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

`@mat[i][j] =` `mat + (i*NC + j)*4`

`la`     \$t3, mat

`li`     \$t4, NC

`mult` \$t4, \$t0

`mflo` \$t4

`addu` \$t4, \$t4, \$t1

`sll`    \$t4, \$t4, 2

`addu` \$t4, \$t3, \$t4

# Accés *aleatori* a un element

- Traduïm a MIPS la sentència `k=mat[i][j];`
  - Suposem que *mat* és global i que *i*, *j*, *k* es guarden en \$t0, \$t1, \$t2

`@mat[i][j] = mat + (i*NC + j)*4`

```
la    $t3, mat
li    $t4, NC
mult  $t4, $t0
mflo  $t4
addu  $t4, $t4, $t1
sll   $t4, $t4, 2
addu  $t4, $t3, $t4
lw    $t2, 0($t4)
```

# Accés *aleatori* a un element

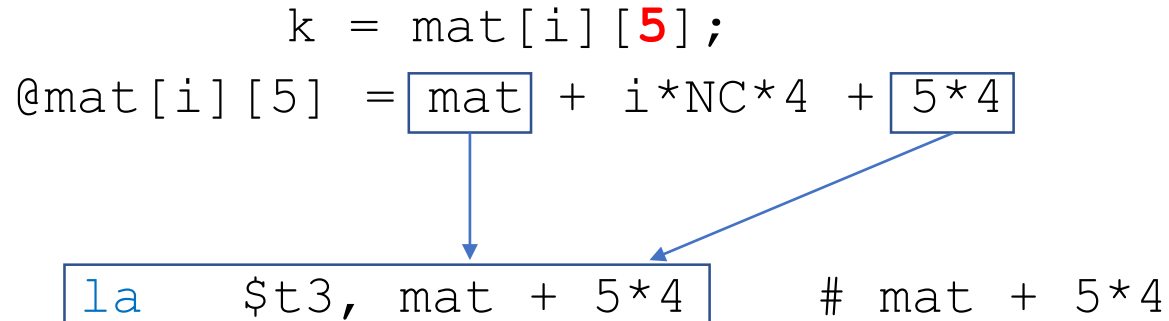
- El codi se simplifica si els índexs són constants

```
k = mat[i][5];  
@mat[i][5] = mat + i*NC*4 + 5*4
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[i][5];  
@mat[i][5] = mat + i*NC*4 + 5*4  
  
la $t3, mat + 5*4      # mat + 5*4
```

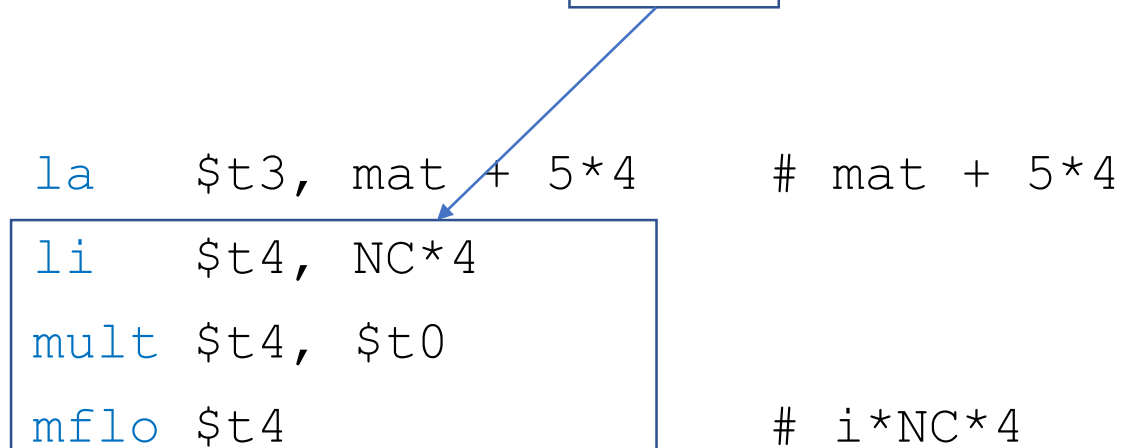


# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[i][5];  
@mat[i][5] = mat + i*NC*4 + 5*4
```

```
la    $t3, mat + 5*4      # mat + 5*4  
li    $t4, NC*4  
mult  $t4, $t0  
mflo  $t4                  # i*NC*4
```



# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[i][5];  
@mat[i][5] = mat + i*NC*4 + 5*4
```

```
la    $t3, mat + 5*4      # mat + 5*4  
li    $t4, NC*4  
mult  $t4, $t0  
mflo  $t4                  # i*NC*4  
addu  $t4, $t3, $t4       # @
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[i][5];
```

```
@mat[i][5] = mat + i*NC*4 + 5*4
```

```
la    $t3, mat + 5*4      # mat + 5*4
```

```
li    $t4, NC*4
```

```
mult  $t4, $t0
```

```
mflo  $t4                      # i*NC*4
```

```
addu  $t4, $t3, $t4
```

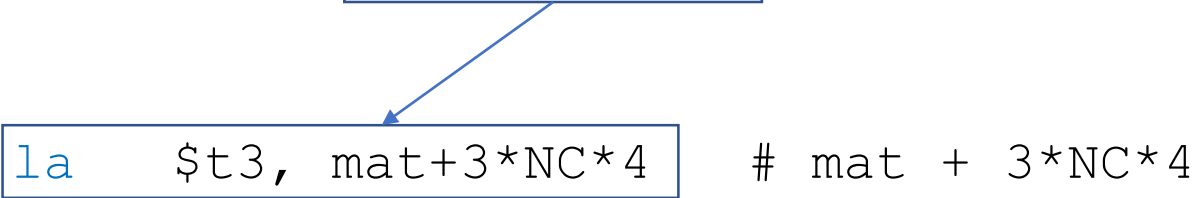
```
lw    $t2, 0($t4)
```



# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][j];  
@mat[i][5] = mat + 3*NC*4 + j*4
```



```
la $t3, mat+3*NC*4    # mat + 3*NC*4
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][j];  
@mat[i][5] = mat + 3*NC*4 + j*4
```

```
la    $t3, mat+3*NC*4    # mat + 3*NC*4  
sll   $t4, $t1, 2        # j*4
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][j];  
@mat[i][5] = mat + 3*NC*4 + j*4
```

```
la    $t3, mat+3*NC*4    # mat + 3*NC*4  
sll   $t4, $t1, 2        # j*4  
addu  $t4, $t3, $t4      # @
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][j];
```

```
@mat[i][5] = mat + 3*NC*4 + j*4
```

```
la    $t3, mat+3*NC*4    # mat + 3*NC*4
```

```
sll   $t4, $t1, 2        # j*4
```

```
addu  $t4, $t3, $t4      # @
```

```
lw    $t2, 0($t4)
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][5];  
@mat[i][5] = mat + 3*NC*4 + 5*4  
  
la    $t4, mat + 3*NC*4 + 5*4    # @
```

# Accés *aleatori* a un element

- El codi se simplifica si els índexs són constants

```
k = mat[3][5];
```

```
@mat[i][5] = mat + 3*NC*4 + 5*4
```

```
la    $t4, mat + 3*NC*4 + 5*4    # @
```

```
lw    $t2, 0($t4)
```

# Example

## 1. Traduir a MIPS (versió bàsica)

```
void clear(int array[], int nelem){  
    int i;  
    for(i=0; i<nelem; i+=1)  
        array[i] = 0;  
}
```

clear:

```
move    $t0, $zero
```

# Example

## 1. Traduir a MIPS (versió bàsica)

```
void clear(int array[], int nelem){  
    int i;  
    for(i=0; i<nelem; i+=1)  
        array[i] = 0;  
}
```

clear:

move \$t0, \$zero

loop:

bge \$t0, \$a1, end1



# Example

## 1. Traduir a MIPS (versió bàsica)

```
void clear(int array[], int nelem){  
    int i;  
    for(i=0; i<nelem; i+=1)  
        array[i] = 0;  
}
```

clear:

move \$t0, \$zero

loop:

bge \$t0, \$a1, end

sll \$t1, \$t0, 2

addu \$t2, \$a0, \$t1

sw \$zero, 0(\$t2)

# Example

## 1. Traduir a MIPS (versió bàsica)

```
void clear(int array[], int nelem){  
    int i;  
    for(i=0; i<nelem; i+=1)  
        array[i] = 0;  
}
```

clear:

move     \$t0, \$zero

loop:

bge     \$t0, \$a1, end

sll     \$t1, \$t0, 2

addu    \$t2, \$a0, \$t1

sw      \$zero, 0(\$t2)

addiu   \$t0, \$t0, 1

b       loop1

end:

# Optimitzacions

- Accés seqüencial
- Eliminar la variable d'inducció
- Avaluar la condició al final del bucle
- Recorreguts d'una matriu amb accés seqüencial
- Exemple de revisió

# Accés seqüencial

- És una optimització per a bucles que recorren elements d'un vector o matriu

$$V_0, V_1, V_2, \dots V_{i-1}, V_i$$

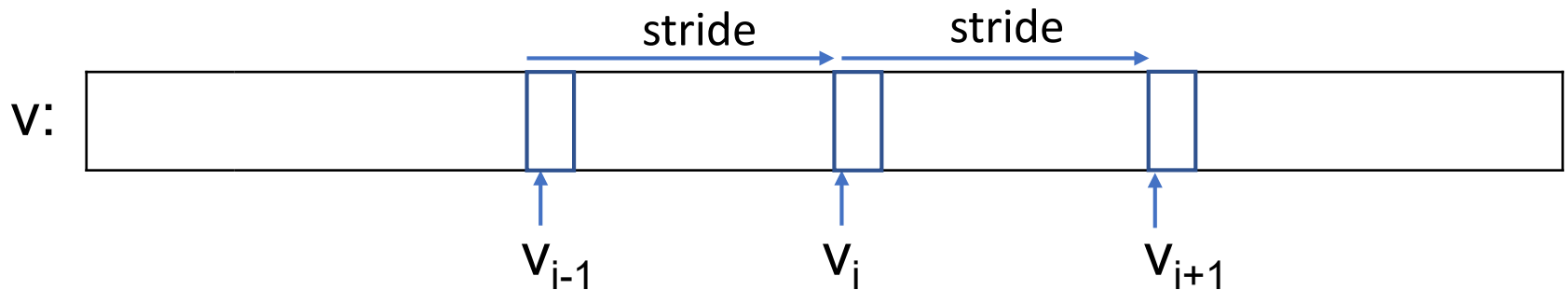
# Accés seqüencial

- És una optimització per a bucles que recorren elements d'un vector o matriu

$$V_0, V_1, V_2, \dots V_{i-1}, V_i$$

- Si les adreces d'elements d'iteracions consecutives estan a distància constant (s'anomena **stride**), l'adreça de la iteració  $i$  s'obté amb l'adreça de la iteració  $i-1$

$$@v_i = @v_{i-1} + \text{stride}$$



# Accés seqüencial

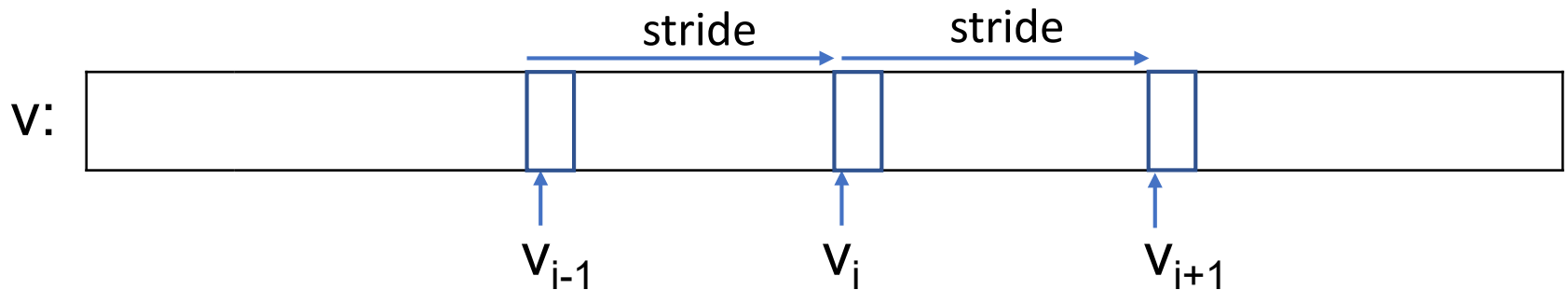
- És una optimització per a bucles que recorren elements d'un vector o matriu

$$V_0, V_1, V_2, \dots V_{i-1}, V_i$$

- Si les adreces d'elements d'iteracions consecutives estan a distància constant (s'anomena **stride**), l'adreça de la iteració  $i$  s'obté amb l'adreça de la iteració  $i-1$

$$@v_i = @v_{i-1} + \text{stride}$$

- No cal una multiplicació en cada iteració!



# Accés seqüencial: mètode

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Abans de programar, calculem

1. Adreça del primer element a recórrer (@array[0]): **\$a0**
2. El stride: diferència entre adreces d'elements en iteracions consecutives

$$\begin{aligned}\text{stride} &= \text{@array}[i+1] - \text{@array}[i] \\ &= (\text{array} + (i + 1)*4) - (\text{array} + i*4) \\ &= 4\end{aligned}$$

# Accés seqüencial: mètode

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Consells de programació

1. Inicialitzem un punter (\$t1) apuntant al primer element

**move \$t1, \$a0**

2. En cada iteració: accedim a l'element desreferenciant \$t1

**sw \$zero, 0(\$t1)**

3. En cada iteració: actualitzem el punter sumant-li el stride

**addiu \$t1, \$t1, 4**



# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
move    $t0, $zero
```

loop:

```
bge     $t0, $a1, end
```

```
sll     $t1, $t0, 2
```

```
addu    $t2, $a0, $t1
```

```
sw      $zero, 0($t2)
```

```
addiu   $t0, $t0, 1
```

```
b       loop
```

end:

```
jr      $ra
```

clear1:

```
move    $t1, $a0
```

Inicialitzem el punter **\$t1** apuntant a **array[0]**

# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
    move    $t0, $zero
```

loop:

```
    bge     $t0, $a1, end
```

```
    sll     $t1, $t0, 2
```

```
    addu    $t2, $a0, $t1
```

```
    sw      $zero, 0($t2)
```

```
    addiu   $t0, $t0, 1
```

```
    b       loop
```

end:

```
    jr      $ra
```

clear1:

```
    move    $t1, $a0
```

```
    move    $t0, $zero
```

loop1:

```
    bge     $t0, $a1, end1
```

El control del bucle queda igual

# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
move    $t0, $zero
```

loop:

```
bge     $t0, $a1, end
```

```
sll     $t1, $t0, 2
```

```
addu    $t2, $a0, $t1
```

```
sw      $zero, 0($t2)
```

```
addiu   $t0, $t0, 1
```

```
b       loop
```

end:

```
jr      $ra
```

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
```

```
sw      $zero, 0($t1)
```

Accedim al vector amb una simple indirecció de **\$t1**!

# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
move    $t0, $zero
```

loop:

```
bge     $t0, $a1, end
```

```
sll     $t1, $t0, 2
```

```
addu    $t2, $a0, $t1
```

```
sw      $zero, 0($t2)
```

```
addiu   $t0, $t0, 1
```

```
b       loop
```

end:

```
jr      $ra
```

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

Sumem el stride al punter **\$t1**

# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
move    $t0, $zero
```

loop:

```
bge     $t0, $a1, end
```

```
sll     $t1, $t0, 2
```

```
addu    $t2, $a0, $t1
```

```
sw      $zero, 0($t2)
```

```
addiu   $t0, $t0, 1
```

```
b       loop
```

end:

```
jr      $ra
```

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
addiu   $t0, $t0, 1
```

```
b       loop1
```

end1:

La resta del bucle queda igual

# Exemple: accés seqüencial

- Optimització 1: accés seqüencial

clear:

```
move    $t0, $zero
```

loop:

```
bge     $t0, $a1, end
sll     $t1, $t0, 2
addu    $t2, $a0, $t1
sw      $zero, 0($t2)
addiu   $t0, $t0, 1
b       loop
```

end:

```
jr      $ra
```

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
sw      $zero, 0($t1)
addiu   $t1, $t1, 4
addiu   $t0, $t0, 1
b       loop1
```

end1:

```
jr      $ra
```

Resultat: el bucle loop2 té 1 instrucció menys!

# Eliminar la variable d'inducció

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Com es controla el bucle?
  - Amb la variable d'inducció: el bucle itera entre  $i = 0$  i  $i = nelem$

# Eliminar la variable d'inducció

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Com es controla el bucle?
  - Amb la variable d'inducció: el bucle itera entre  $i = 0$  i  $i = nelem$
- Podem controlar-lo amb el punter?
  - Sí, el punter itera entre  $@array[0]$  i  $@array[nelem]$



# Eliminar la variable d'inducció

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Com es controla el bucle?
  - Amb la variable d'inducció: el bucle itera entre  $i = 0$  i  $i = nelem$
- Podem controlar-lo amb el punter?
  - Sí, el punter itera entre  $@array[0]$  i  $@array[nelem]$
- Llavors, podem prescindir de la variable  $i$ ?
  - Si, a condició que només s'utilitzi per al control del bucle
- Optimització 2: *eliminar la variable d'inducció*

# Eliminar la variable d'inducció

```
void clear(int array[], int
nelem) {
    int i;
    for(i=0; i<nelem; i+=1)
        array[i] = 0;
}
```

- Consells de programació:

1. Calculem l'adreça del punter després de l'últim increment

**$\$t3 = @array[nelem] = array + nelem * 4$**

2. En lloc de comparar " $i < nelem$ " comparem

**$\$t1 < \$t3$**

Atenció! Si estem comparant adreces (naturals)  
en lloc de *bge* hem d'usar **bgeu**

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
addiu   $t0, $t0, 1
```

```
b       loop1
```

end1:

```
jr      $ra
```

clear2:

```
move    $t1, $a0
```

L'inici és igual

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

move \$t1, \$a0

~~move \$t0, \$zero~~

loop1:

bge \$t0, \$a1, end1

sw \$zero, 0(\$t1)

addiu \$t1, \$t1, 4

addiu \$t0, \$t0, 1

b loop1

end1:

jr \$ra

clear2:

move \$t1, \$a0

sll \$t2, \$a1, 2

addu \$t3, \$a0, \$t2

Calculem en \$t3 el darrer valor del punter

$\$t3 = \text{array} + 4 * \text{nelem}$

$= \$a0 + 4 * \$a1$

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

move \$t1, \$a0

~~move \$t0, \$zero~~

loop1:

bge \$t0, \$a1, end1

sw \$zero, 0(\$t1)

addiu \$t1, \$t1, 4

addiu \$t0, \$t0, 1

b loop1

end1:

jr \$ra

clear2:

move \$t1, \$a0

sll \$t2, \$a1, 2

addu \$t3, \$a0, \$t2

loop2:

bgeu \$t1, \$t3, end2

Comparem el punter \$t1 amb \$t3

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

move \$t1, \$a0

~~move \$t0, \$zero~~

loop1:

bge \$t0, \$a1, end1

sw \$zero, 0(\$t1)

addiu \$t1, \$t1, 4

addiu \$t0, \$t0, 1

b loop1

end1:

jr \$ra

clear2:

move \$t1, \$a0

sll \$t2, \$a1, 2

addu \$t3, \$a0, \$t2

loop2:

bgeu \$t1, \$t3, end2

sw \$zero, 0(\$t1)

addiu \$t1, \$t1, 4

L'accés al vector queda igual

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

```
move    $t1, $a0
```

```
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
addiu   $t0, $t0, 1
```

```
b       loop1
```

end1:

```
jr      $ra
```

clear2:

```
move    $t1, $a0
```

```
sll     $t2, $a1, 2
```

```
addu    $t3, $a0, $t2
```

loop2:

```
bgeu    $t1, $t3, end2
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
b       loop2
```

end2:

Ja no hem d'incrementar la variable *i*

# Exemple: eliminar la variable d'inducció

- Optimització 2: eliminar la variable d'inducció (*i*)

clear1:

```
move    $t1, $a0
move    $t0, $zero
```

loop1:

```
bge     $t0, $a1, end1
sw      $zero, 0($t1)
addiu   $t1, $t1, 4
addiu   $t0, $t0, 1
b       loop1
```

end1:

```
jr      $ra
```

clear2:

```
move    $t1, $a0
sll     $t2, $a1, 2
addu    $t3, $a0, $t2
```

loop2:

```
bgeu    $t1, $t3, end2
sw      $zero, 0($t1)
addiu   $t1, $t1, 4
b       loop2
```

end2:

```
jr      $ra
```

Resultat: el bucle loop3 té 1 instrucció menys!



# Avaluar la condició al final del bucle

- Optimització 3: *avaluar la condició al final del bucle*
  - Canviem el while per un do-while

```
while (condició)
{
    cos_del_bucle
}
```



```
do
{
    cos_del_bucle
}
while (condició);
```

# Avaluar la condició al final del bucle

- Optimització 3: *avaluar la condició al final del bucle*
  - Canviem el while per un do-while
  - Però el bucle pot tenir 0 iteracions!!!

```
while (condició)
{
    cos_del_bucle
}
```

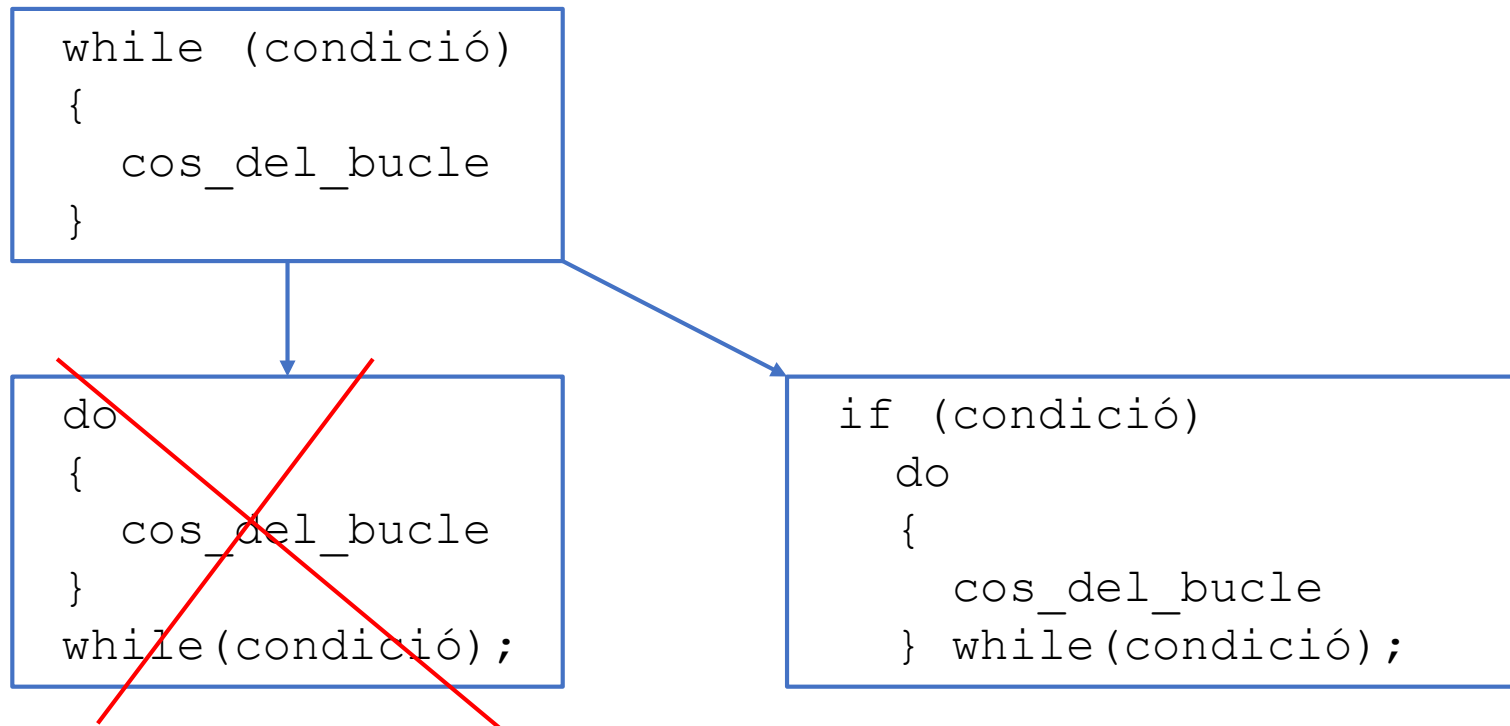
??

**Sempre executaria la 1a. iteració!!!**

```
do
{
    cos_del_bucle
}
while (condició);
```

# Avaluar la condició al final del bucle

- Optimització 3: *avaluar la condició al final del bucle*
  - Canviem el while per un do-while
  - Però el bucle pot tenir 0 iteracions!!!
  - Abans d'executar la primera iteració cal comprovar la condició del bucle



# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll      $t2, $a1, 2
addu     $t3, $a0, $t2
```

loop2:

```
bgeu     $t1, $t3, end2
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
b         loop2
```

end2:

```
jr       $ra
```

clear3:

```
move     $t1, $a0
sll       $t2, $a1, 2
addu      $t3, $a0, $t2
```

L'inici és igual

# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll      $t2, $a1, 2
addu     $t3, $a0, $t2
```

loop2:

```
bgeu     $t1, $t3, end2
```

```
sw       $zero, 0($t1)
addiu    $t1, $t1, 4
```

```
b        loop2
```

end2:

```
jr       $ra
```

clear3:

```
move     $t1, $a0
sll       $t2, $a1, 2
addu      $t3, $a0, $t2
```

loop3:

```
sw       $zero, 0($t1)
addiu    $t1, $t1, 4
```

El cos del bucle queda igual

# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll     $t2, $a1, 2
addu    $t3, $a0, $t2
```

loop2:

```
bgeu    $t1, $t3, end2
```

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
b       loop2
```

end2:

```
jr      $ra
```

clear3:

```
move    $t1, $a0
```

```
sll     $t2, $a1, 2
```

```
addu    $t3, $a0, $t2
```

loop3:

```
sw      $zero, 0($t1)
```

```
addiu   $t1, $t1, 4
```

```
bltu   $t1, $t3, loop3
```

end3:

La condició s'avalua al final (**bltu** en lloc de **bgeu**)

# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll      $t2, $a1, 2
addu     $t3, $a0, $t2
```

loop2:

```
bgeu     $t1, $t3, end2
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
b         loop2
```

end2:

```
jr       $ra
```

clear3:

```
move     $t1, $a0
sll       $t2, $a1, 2
addu      $t3, $a0, $t2
```

```
bgeu     $t1, $t3, end3
```

loop3:

```
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
bltu      $t1, $t3, loop3
```

end3:

```
jr       $ra
```

Però no oblidem comprovar la condició  
també abans d'entrar al bucle!

# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll      $t2, $a1, 2
addu     $t3, $a0, $t2
```

loop2:

```
bgeu     $t1, $t3, end2
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
b         loop2
```

end2:

```
jr       $ra
```

clear3:

```
move     $t1, $a0
sll       $t2, $a1, 2
addu      $t3, $a0, $t2
bgeu      $t1, $t3, end3
```

loop3:

```
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
bltu      $t1, $t3, loop3
```

end3:

```
jr       $ra
```

Resultat: el bucle loop4 té 1 instrucció menys!



# Exemple: avaluar la condició al final del bucle

- Optimització 3: avaluar la condició al final del bucle

clear2:

```
move    $t1, $a0
sll      $t2, $a1, 2
addu     $t3, $a0, $t2
```

loop2:

```
bgeu     $t1, $t3, end2
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
b         loop2
```

end2:

```
jr       $ra
```

clear3:

```
move     $t1, $a0
sll       $t2, $a1, 2
addu      $t3, $a0, $t2
```

<b>b</b>	<b>test</b>
----------	-------------

loop3:

```
sw        $zero, 0($t1)
addiu     $t1, $t1, 4
test:    bltu     $t1, $t3, loop3
```

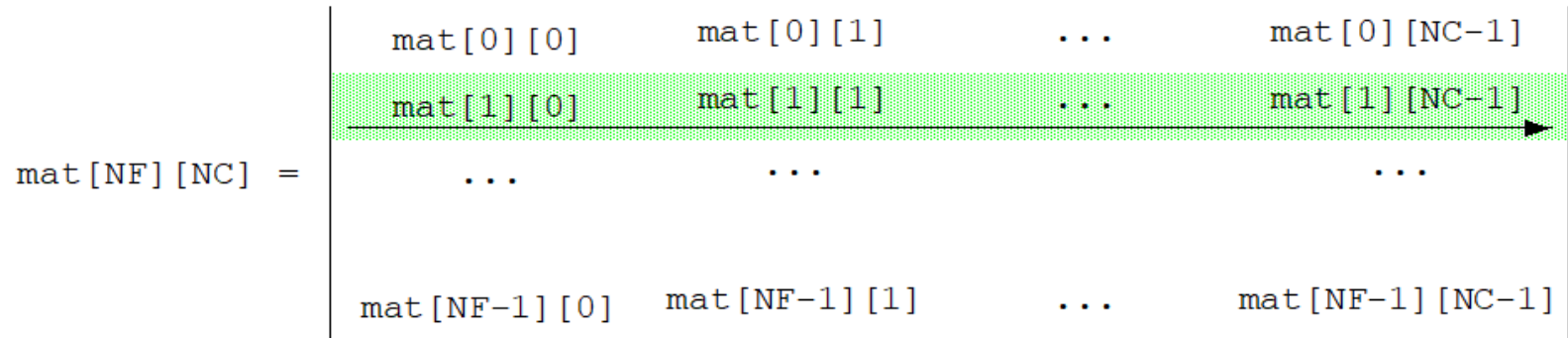
end3:

```
jr       $ra
```

Si l'avaluació de la condició comporta moltes instruccions, podem substituir-la per un simple salt
--

# Accés seqüencial a una matriu

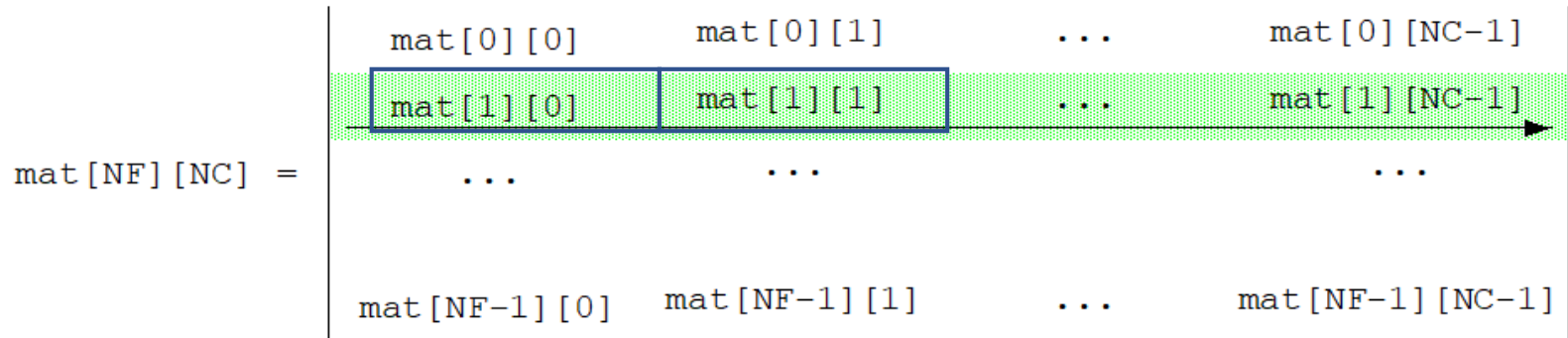
- Recorregut d'una *fila*



- Quin és el *stride*?

# Accés seqüencial a una matriu

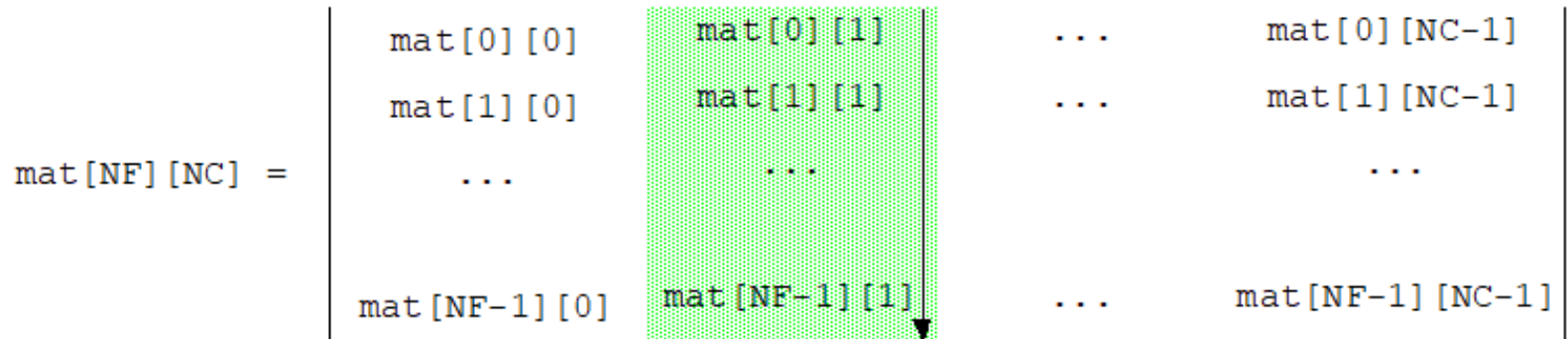
- Recorregut d'una *fila*



- Quin és el *stride*?
  - Els elements de la fila estan consecutius en memòria
  - *stride* = mida d'un element  
=  $T$

# Accés seqüencial a una matriu

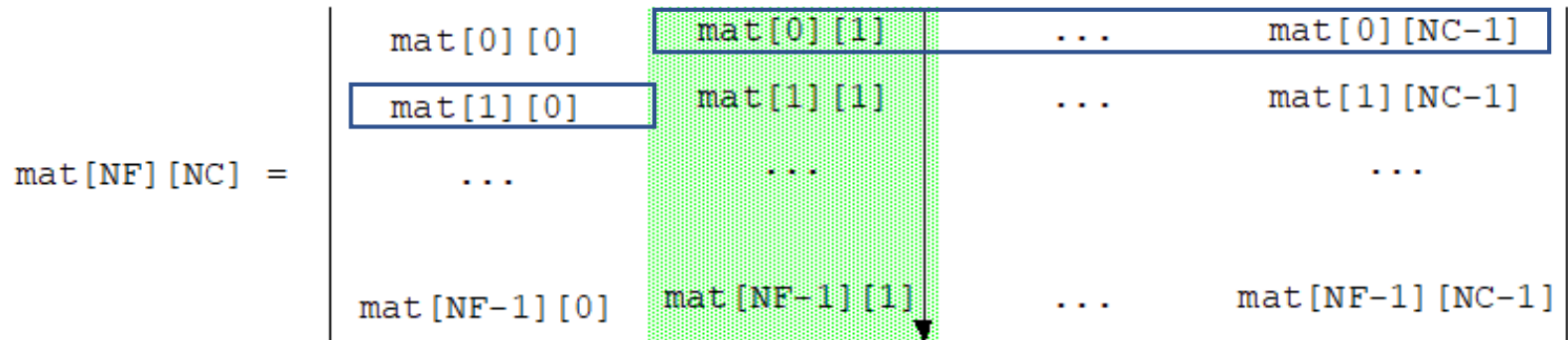
- Recorregut d'una *columna*



- Quin és el *stride*?

# Accés seqüencial a una matriu

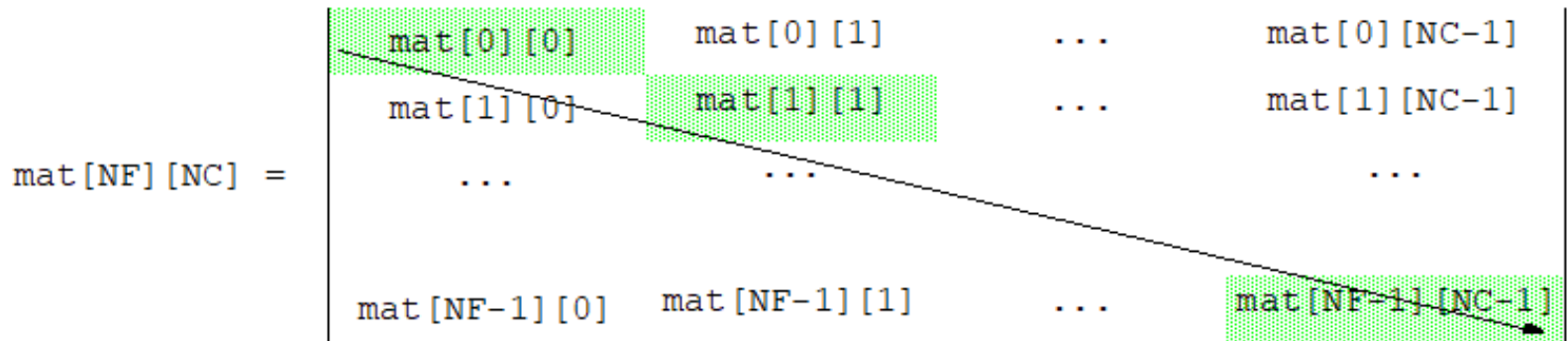
- Recorregut d'una *columna*



- Quin és el *stride*?
  - Els elements de la columna estan separats per una fila completa d'elements
  - *stride* = mida d'una fila  
=  $\text{NC} * \text{T}$

# Accés seqüencial a una matriu

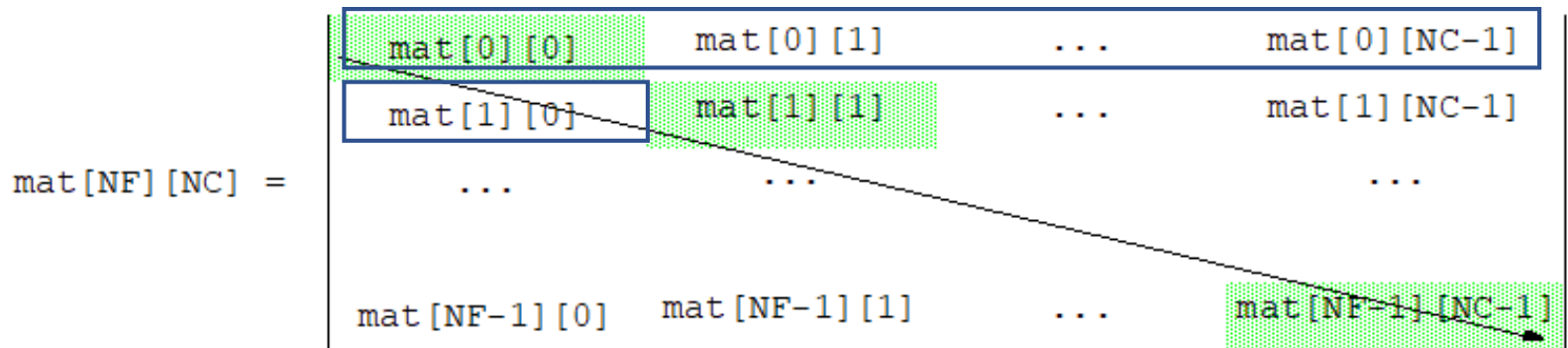
- Recorregut de la *diagonal principal*



- Quin és el *stride*?

# Accés seqüencial a una matriu

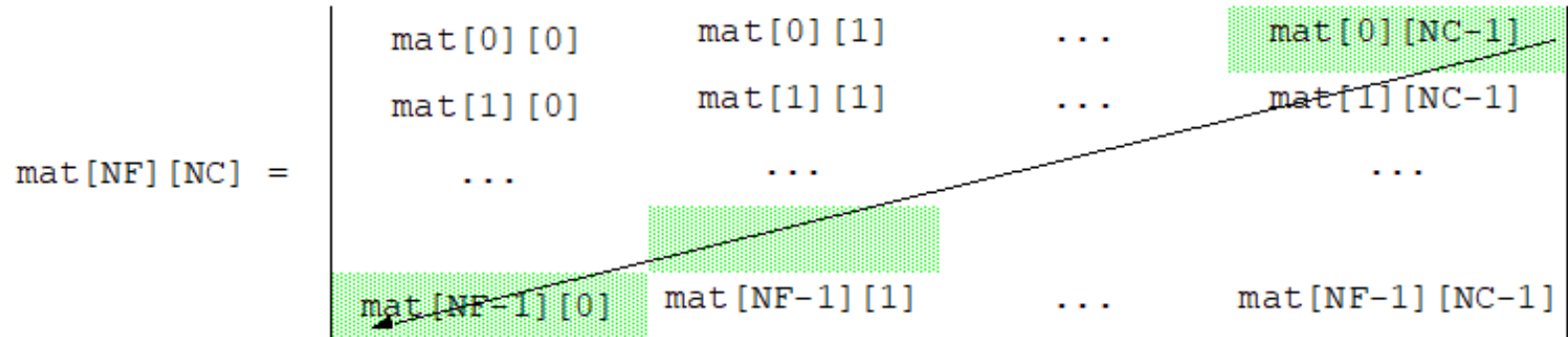
- Recorregut de la *diagonal principal*



- Quin és el *stride*?
  - Els elements de la diagonal estan separats per una fila completa i un element més
  - *stride* = mida d'una fila i un element  
=  $(NC + 1) * T$

# Accés seqüencial a una matriu

- Recorregut de la *diagonal secundària*

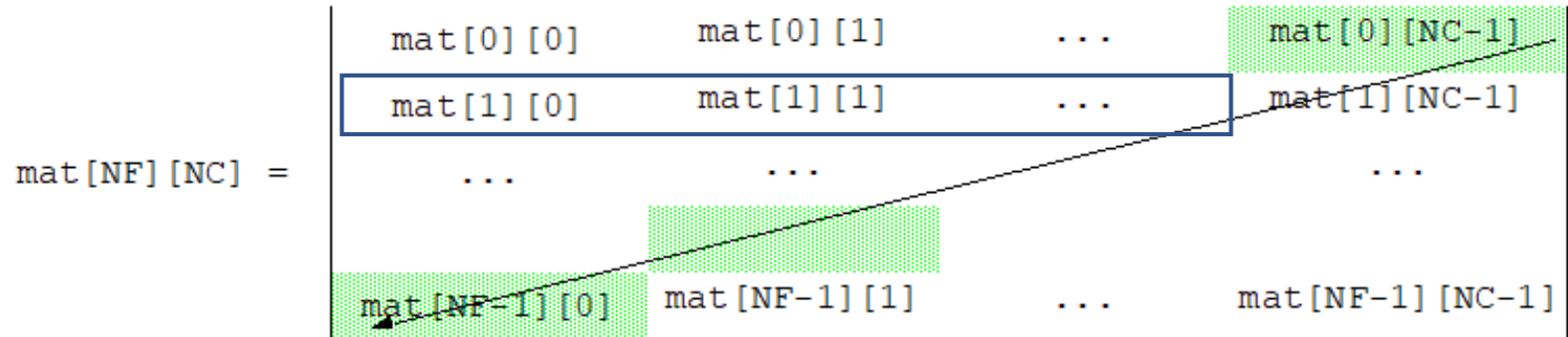


- Quin és el *stride*?



# Accés seqüencial a una matriu

- Recorregut de la *diagonal secundària*



- Quin és el *stride*?
  - Els elements de la diagonal estan separats per una fila completa menys un element
  - *stride* = mida d'una fila menys un element  
=  $(NC - 1) * T$

# Exemple de revisió

- Traduir a MIPS la funció *exemple* (recorre la columna *col*)

```
short exemple(short mat[][NC], int col, int nfiles) {  
    int i;  
    short suma = 0;  
    for (i=0; i<nfiles; i++) {  
        suma = suma + mat[i][col];  
    }  
    return suma;  
}
```

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],  
              int col, int nfiles) {  
    int i;  
    short suma = 0;  
    for (i=0; i<nfiles; i++) {  
        suma = suma + mat[i][col];  
    }  
    return suma;  
}
```

- La subrutina és *uninivell*: no cal usar registres segurs

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

- La subrutina és *uninivell*: no cal usar registres segurs
- L'adreça de mat[i][col] és:  
 $@mat[i][col] = mat + i * NC * 2 + col * 2$

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

- La subrutina és *uninivell*: no cal usar registres segurs
- L'adreça de `mat[i][col]` és:  
 $@mat[i][col] = mat + i * NC * 2 + col * 2$
- $mat + col * 2$  és invariant del bucle: el calcularem fora del bucle
- $NC * 2$  també és invariant del bucle

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],  
              int col, int nfiles) {  
  
    int i;  
    short suma = 0;  
    for (i=0; i<nfiles; i++) {  
        suma = suma + mat[i][col];  
    }  
    return suma;  
}
```

exemple:

move	\$v0, \$zero	# suma=0
move	\$t0, \$zero	# i=0

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

Calculem l'invariant:

$\$t1 = NC * 2$

Calculem l'invariant:

$\$t2 = mat + col * 2$

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li       $t1, NC*2       # NC*2
sll      $t2, $a1, 1
addu     $t2, $a0, $t2    # mat+col*2
```

for:

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li      $t1, NC*2        # NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, ffor
```



# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

Calculem l'adreça de mat[i][col]:  
 $\text{mat} + i * \text{NC} * 2 + \text{col} * 2$

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li       $t1, NC*2       # NC*2
sll      $t2, $a1, 1
addu     $t2, $a0, $t2   # mat+col*2
```

for:

```
bge      $t0, $a2, ffor
mult     $t0, $t1         # i*NC*2
mflo     $t3
addu     $t3, $t2, $t3    # @
```

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

Llegim l'element de mat, i el  
sumem

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li      $t1, NC*2        # NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, ffor
mult    $t0, $t1          # i*NC*2
mflo    $t3
addu    $t3, $t2, $t3    # @
lh      $t4, 0($t3)
addu    $v0, $v0, $t4
```

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
              int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li      $t1, NC*2        # NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, fifor
mult    $t0, $t1          # i*NC*2
mflo    $t3
addu    $t3, $t2, $t3    # @
lh      $t4, 0($t3)
addu    $v0, $v0, $t4
```

```
addiu   $t0, $t0, 1
b       for
```

fifor:

# Exemple: accés aleatori

- Traducció amb accés aleatori

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
li       $t1, NC*2       # NC*2
sll      $t2, $a1, 1
addu     $t2, $a0, $t2   # mat+col*2
for:
bge      $t0, $a2, fifor
mult     $t0, $t1        # i*NC*2
mflo     $t3
addu     $t3, $t2, $t3   # @
lh       $t4, 0($t3)
addu     $v0, $v0, $t4
addiu    $t0, $t0, 1
b        for
fifor:
jr       $ra
```

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

```
short exemple(short mat[][NC],  
              int col, int nfiles) {  
  
    int i;  
    short suma = 0;  
    for (i=0; i<nfiles; i++) {  
        suma = suma + mat[i][col];  
    }  
    return suma;  
}
```

1. L'adreça inicial del punter és:

$\text{@mat}[0][\text{col}] =$   
 $= \text{mat} + 0 * \text{NC} * 2 + \text{col} * 2$   
 $= \text{mat} + \text{col} * 2$

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

```
short exemple(short mat[][NC],
               int col, int nfiles) {
    int i;
    short suma = 0;
    for (i=0; i<nfiles; i++) {
        suma = suma + mat[i][col];
    }
    return suma;
}
```

1. L'adreça inicial del punter és:

$$\begin{aligned}\text{@mat}[0][\text{col}] &= \\ &= \text{mat} + 0 * \text{NC} * 2 + \text{col} * 2 \\ &= \text{mat} + \text{col} * 2\end{aligned}$$

2. Calculem **stride** =

$$= \text{@mat}[i+1][\text{col}] - \text{@mat}[i][\text{col}]$$

$$\begin{aligned}&= \text{mat} + (i+1) * \text{NC} * 2 + \text{col} * 2 \\ &- (\text{mat} + i * \text{NC} * 2 + \text{col} * 2)\end{aligned}$$

$$= \text{NC} * 2$$

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move $v0, $zero
move $t0, $zero
li    $t1, NC*2
sll   $t2, $a1, 1
addu  $t2, $a0, $t2
```

for:

```
bge   $t0, $a2, fifor
mult  $t0, $t1
mflo  $t3
addu  $t3, $t2, $t3
lh    $t4, 0($t3)
addu  $v0, $v0, $t4
addiu $t0, $t0, 1
b     for
```

fifor:

```
jr    $ra
```

exemple:

```
move $v0, $zero    # suma=0
move $t0, $zero    # i=0
```

L'inici queda igual

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move    $v0, $zero
move    $t0, $zero
li      $t1, NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2
```

for:

```
bge     $t0, $a2, fifor
mult    $t0, $t1
mflo    $t3
addu    $t3, $t2, $t3
lh      $t4, 0($t3)
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

exemple:

```
move    $v0, $zero    # suma=0
move    $t0, $zero    # i=0
```

```
sll     $t2, $a1, 1
addu    $t3, $a0, $t2 # mat+col*2
```

for:

Inicialitzem el punter **\$t3**  
amb l'adreça de  
mat[0][col]



# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move    $v0, $zero
move    $t0, $zero
li      $t1, NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2
```

for:

```
bge     $t0, $a2, fifor
mult    $t0, $t1
mflo    $t3
addu    $t3, $t2, $t3
lh      $t4, 0($t3)
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
sll     $t2, $a1, 1
addu    $t3, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, fifor
```

La condició queda igual

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move $v0, $zero
move $t0, $zero
li $t1, NC*2
sll $t2, $a1, 1
addu $t2, $a0, $t2
```

for:

```
bge $t0, $a2, ffor
```

```
mult $t0, $t1
```

```
mflo $t3
```

```
addu $t3, $t2, $t3
```

```
lh $t4, 0($t3)
```

```
addu $v0, $v0, $t4
```

```
addiu $t0, $t0, 1
```

```
b for
```

ffor:

```
jr $ra
```

exemple:

```
move $v0, $zero # suma=0
```

```
move $t0, $zero # i=0
```

```
sll $t2, $a1, 1
```

```
addu $t3, $a0, $t2 # mat+col*2
```

for:

```
bge $t0, $a2, ffor
```

```
lh $t4, 0($t3)
```

Ja no calen càlculs d'adreça!  
Accedim a memòria usant el  
punter **\$t3**

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move $v0, $zero
move $t0, $zero
li $t1, NC*2
sll $t2, $a1, 1
addu $t2, $a0, $t2
```

for:

```
bge $t0, $a2, ffor
mult $t0, $t1
mflo $t3
addu $t3, $t2, $t3
lh $t4, 0($t3)
addu $v0, $v0, $t4
addiu $t0, $t0, 1
b for
```

ffor:

```
jr $ra
```

exemple:

```
move $v0, $zero # suma=0
move $t0, $zero # i=0
sll $t2, $a1, 1
addu $t3, $a0, $t2 # mat+col*2
```

for:

```
bge $t0, $a2, ffor
lh $t4, 0($t3)
```

```
addiu $t3, $t3, NC*2
```

Sumem el stride **NC\*2** al punter **\$t3**

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move    $v0, $zero
move    $t0, $zero
li      $t1, NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2
```

for:

```
bge     $t0, $a2, fifor
mult    $t0, $t1
mflo    $t3
addu    $t3, $t2, $t3
lh      $t4, 0($t3)
```

```
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
sll     $t2, $a1, 1
addu    $t3, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, fifor
lh      $t4, 0($t3)
addiu   $t3, $t3, NC*2
```

```
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

La resta del bucle queda igual

# Exemple: accés seqüencial

- Traducció amb accés seqüencial

exemple:

```
move    $v0, $zero
move    $t0, $zero
li      $t1, NC*2
sll     $t2, $a1, 1
addu    $t2, $a0, $t2
```

for:

```
bge     $t0, $a2, fifor
mult    $t0, $t1
mflo    $t3
addu    $t3, $t2, $t3
lh      $t4, 0($t3)
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

exemple:

```
move    $v0, $zero      # suma=0
move    $t0, $zero      # i=0
sll     $t2, $a1, 1
addu    $t3, $a0, $t2    # mat+col*2
```

for:

```
bge     $t0, $a2, fifor
lh      $t4, 0($t3)
addiu   $t3, $t3, NC*2
addu    $v0, $v0, $t4
addiu   $t0, $t0, 1
b       for
```

fifor:

```
jr      $ra
```

**Resultat: un bucle amb 2 instruccions menys i sense la ineficient multiplicació**