UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tema 4
Matrius

Estructura de Computadors (EC)
2023 - 2024 Q2
Adrià Armejach (adria.armejach@upc.edu)

Les instruccions MIPS
mult, *mflo*, *mfhi*

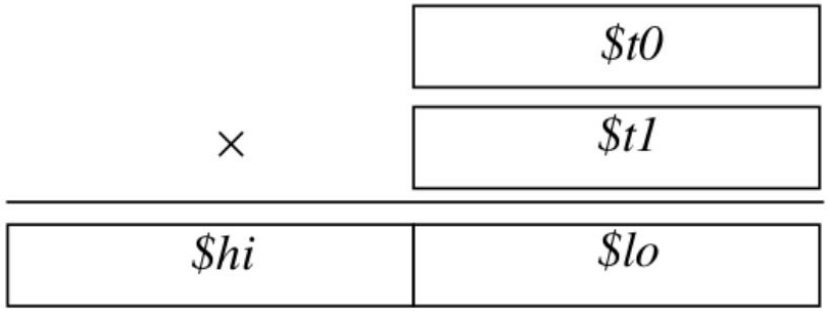
Multiplicació d'enters a MIPS

- La multiplicació de dos enters de *n* i *m* bits dóna un resultat de *n+m* bits
- La multiplicació de dos enters de *n* bits dóna un resultat de *2n* bits
- En MIPS:
  - `mult rs, rt # $hi:$lo <- rs * rt`
- `$hi` i `$lo` son dos registres especials
  - No es poden utilitzar en les instruccions estudiades fins ara
- Per moure el resultat a registres de propòsit general:
  - `mflo rd  # rd <- $lo`
  - `mfhi rd  # rd <- $hi`

Multiplicació d'enters a MIPS

- Per calcular `$t2 = $t0 * $t1`, ignorant els 32 bits de més pes del resultat:

```
mult $t0, $t1
mflo $t2
```



Constants simbòliques

Constants simbòliques

- Faciliten l'escriptura d'un programa
- En C:

```
#define N 100
```

- En MIPS:

```
.eqv N, 100
.data
…
.text
…
```

Matrius

Matrius

- Agrupació multidimensional d'elements de tipus homogeni
  - Els elements s'identifiquen per un índex en cada dimensió
  - Estudiarem matrius de dos dimensions

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

Declaració de Matrius

- En C:

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

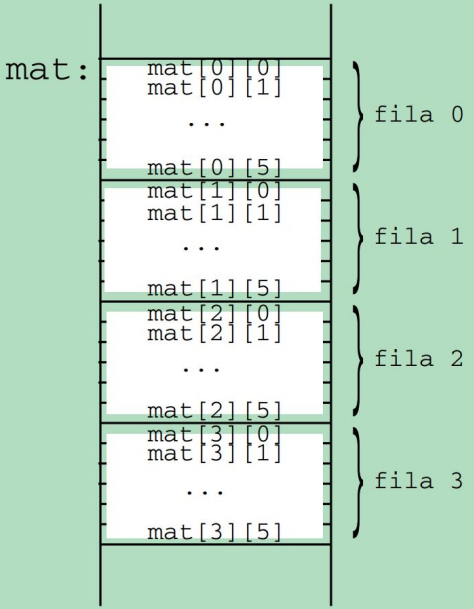
- En MIPS:

```
        .data
mat:    .space NF*NC*4
mit:    .word -1, 2, 0, 1, -12, 4
```

Emmagatzematge de matrius a memòria

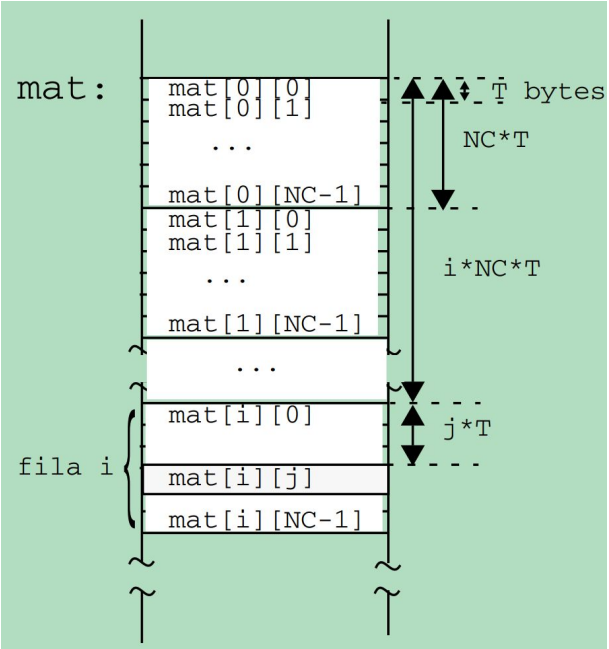
- En C les matrius s'emmagatzemen per files
  - A partir de l'adreça base `mat`
  - Primer tots els elements de la primera fila
    - `mat[0][0]`, `mat[0][1]`, `mat[0][2]`, …
  - S'han de respectar les regles d'alineament

Figura: `mat[4][6]` en memòria



Accés aleatori

- Per accedir a un element situat a la fila `i`, columna `j`:
  - `mat[i][j]`
- L'adreça es calcula:
  - `@mat[i][j] = mat + i*NC*T + j*T`
  - `@mat[i][j] = mat + (i*NC+j) * T`



Exemple 1

- Traduir a MIPS la sentència en C de la funció `func`
  - Accedeix a la matriu global `mat`
  - Les variables locals `i, j, k` s'emmagatzemen als registres `$t0, $t1, $t2` respectivament

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

Exemple 2 - columna és constant

- Traduir a MIPS la sentència en C de la funció `func`
  - Accedeix a la matriu global `mat`
  - Les variables locals `i, j, k` s'emmagatzemen als registres `$t0, $t1, $t2` respectivament

```
int mat[NF][NC];
void func() {
     int i, j, k;
     …
     k = mat[i][5];
}
```

Exemple 3 - fila és constant

- Traduir a MIPS la sentència en C de la funció `func`
  - Accedeix a la matriu global `mat`
  - Les variables locals `i, j, k` s'emmagatzemen als registres `$t0, $t1, $t2` respectivament

```
int mat[NF][NC];
void func() {
    int i, j, k;
    …
    k = mat[3][j];
}
```

```
la      $t3, mat + 3*NC*4
sll     $t4, $t1, 2                 # $t4 = j*4
addu    $t3, $t3, $t4               # $t3 = mat + 3*NC*4 + j*4
lw      $t2, 0($t3)                 # k = mat[3][j]
```

Exemple 4 - fila i columna són constants

- Traduir a MIPS la sentència en C de la funció `func`
  - Accedeix a la matriu global `mat`
  - Les variables locals `i, j, k` s'emmagatzemen als registres `$t0, $t1, $t2` respectivament

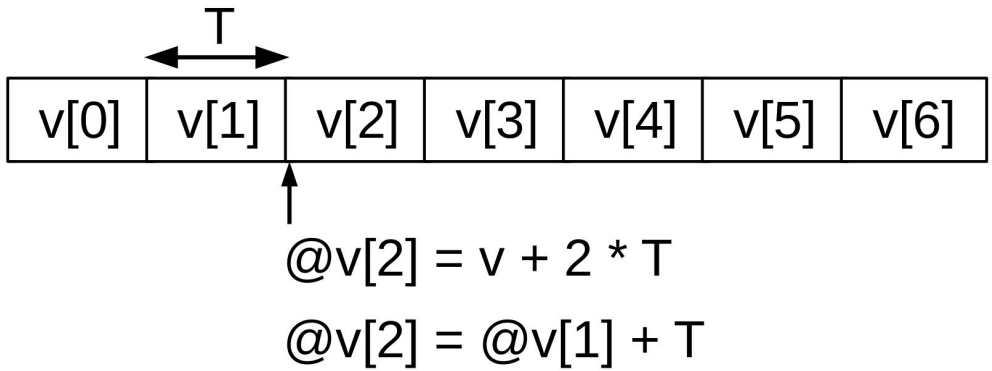
```
int mat[NF][NC];
void func() {
    int i, j, k;
    …
    k = mat[3][5];
}
```

```
la      $t3, mat + 3*NC*4 + 5*4
lw      $t2, 0($t3)                 # k = mat[3][5]
```

Accés seqüencial

Accés seqüencial a un vector o matriu

- Optimització per bucles que recorren els elements d'un vector o matriu
  - Condició: La distancia en bytes entre les adreces de dos elements consecutius del recorregut sigui constant - stride



Exemple - no optimitzat

```
void clear1(int array[], int nelem)
{
      int i;
      for (i=0; i<nelem; i+=1)
            array[i] = 0;
}
```

```
clear1:
      move   $t0, $zero                  # i=0
loop1:
      bge    $t0, $a1, end1              # salta si i<nelem és fals
      sll    $t1, $t0, 2                 # i*4
      addu   $t2, $a0, $t1               # @array[i] = @array[0] + i*4
      sw     $zero, 0($t2)               # array[i] = 0
      addiu  $t0, $t0, 1                 # i = i + 1
      b      loop1
end1:
```

Passos per aplicar accés seqüencial

1. Calcular l'adreça del primer element a recorrer i inicialitzar un punter `p` (`$t1`) amb aquesta adreça:
   - En C: `p = array;`
   - En MIPS: `move $t1, $a0`

2. Calcular l'stride, restant les adreces de dos elements consecutius del recorregut
   - `stride = @array[i+1] - @array[i]`
   - `= (array + (i+1)*4) - (array + i*4)`
   - `= 4`

Passos per aplicar accés seqüencial

3. Accedirem a memòria desreferenciant el punter, i al final de la iteració li sumarem l'stride
   - En C:
     - `*p = 0;` `// desreferenciem p`
     - `p++;` `// fem que p apunti al "seguent element"`
   - En MIPS:
     - `sw $zero, 0($t1)` `# accés a memòria usant el punter $t1`
     - `addiu $t1, $t1, 4` `# $t1 = $t1 + stride`

Amb accés seqüencial

```
void clear1(int array[], int nelem)
{
      int i;
      for (i=0; i<nelem; i+=1)
            array[i] = 0;
}
```

```
clear2:
      move   $t1, $a0            # inicialitzem punter $t1 = @array[0]
      move   $t0, $zero          # i=0
loop2:
      bge    $t0, $a1, end2      # salta si i<nelem és fals
      sw     $zero, 0($t1)       # accés a memòria usant el punter $t1
      addiu  $t1, $t1, 4         # $t1 = $t1 + stride
      addiu  $t0, $t0, 1         # i = i + 1
      b      loop2
end2:
```

Optimització: Eliminació de la variable d'inducció

```
void clear1(int array[], int nelem)
{
     int i;
     for (i=0; i<nelem; i+=1)
          array[i] = 0;
}
```

```
clear3:
     move   $t1, $a0            # inicialitzem punter $t1 = @array[0]
     sll    $t2, $a1, 2         # nelem*4
     addu   $t3, $a0, $t2       # $t3 = @array[nelem]
loop3:
     bgeu   $t1, $t3, end3      # salta si $t1 < @array[nelem] és fals
     sw     $zero, 0($t1)       # accés a memòria usant el punter
     addiu  $t1, $t1, 4         # $t1 = $t1 + stride
     b      loop3
end3:
```

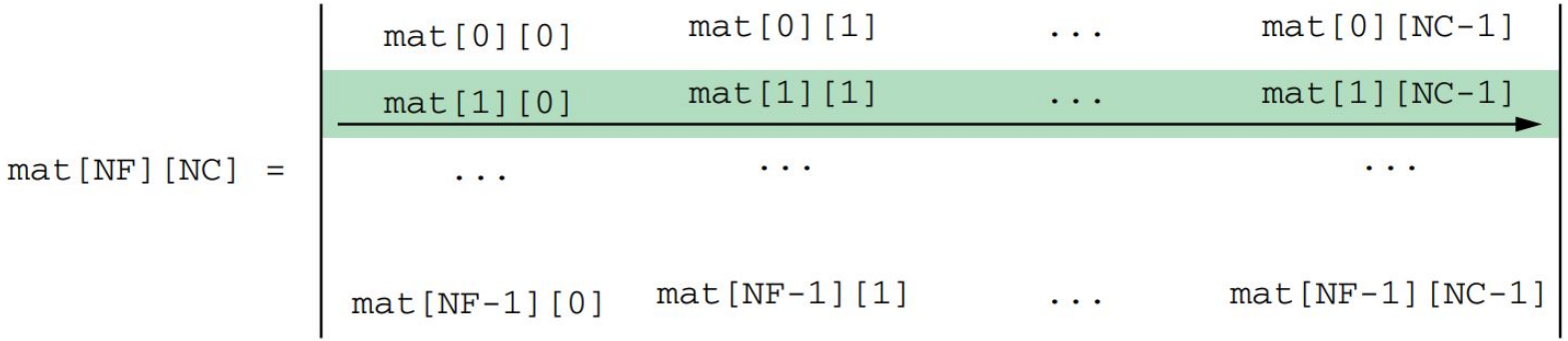
Optimització: Avaluació de la condició al final del bucle

```
void clear1(int array[], int nelem)
{
      int i;
      for (i=0; i<nelem; i+=1)
            array[i] = 0;
}
```

```
clear4:
      move   $t1, $a0            # inicialitzem punter $t1 = @array[0]
      sll    $t2, $a1, 2         # nelem*4
      addu   $t3, $a0, $t2       # $t3 = @array[nelem]
      bgeu   $t1, $t3, end4      # salta si $t1 < @array[nelem] és fals
loop4:
      sw     $zero, 0($t1)       # accés a memòria usant el punter
      addiu  $t1, $t1, 4         # $t1 = $t1 + stride
      bltu   $t1, $t3, loop4     # salta si $t1 < @array[nelem] és cert
end4:
```

Accés seqüencial a matrius

- Recorregut d'una fila

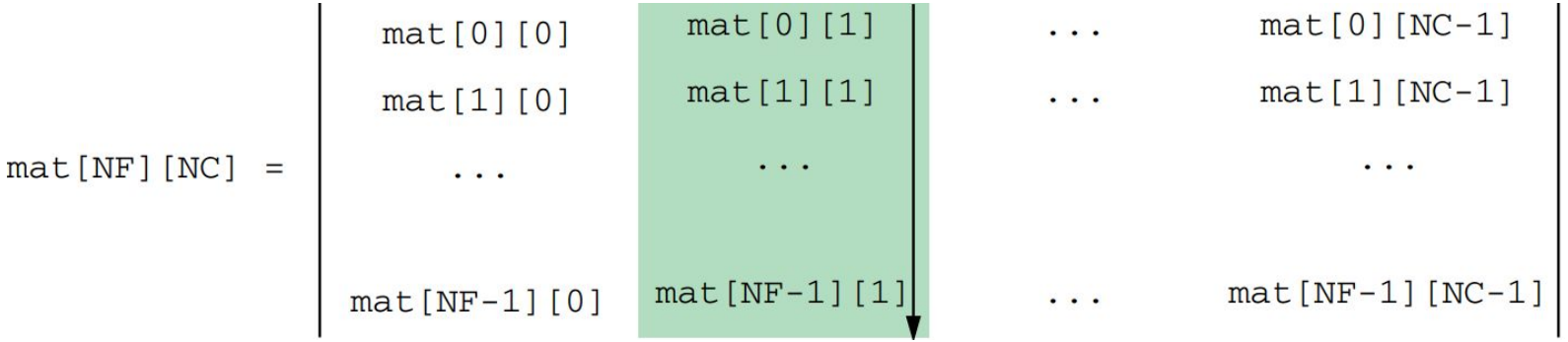


- Stride?

stride = mida d'un element = **T**

Accés seqüencial a matrius

- Recorregut d'una columna

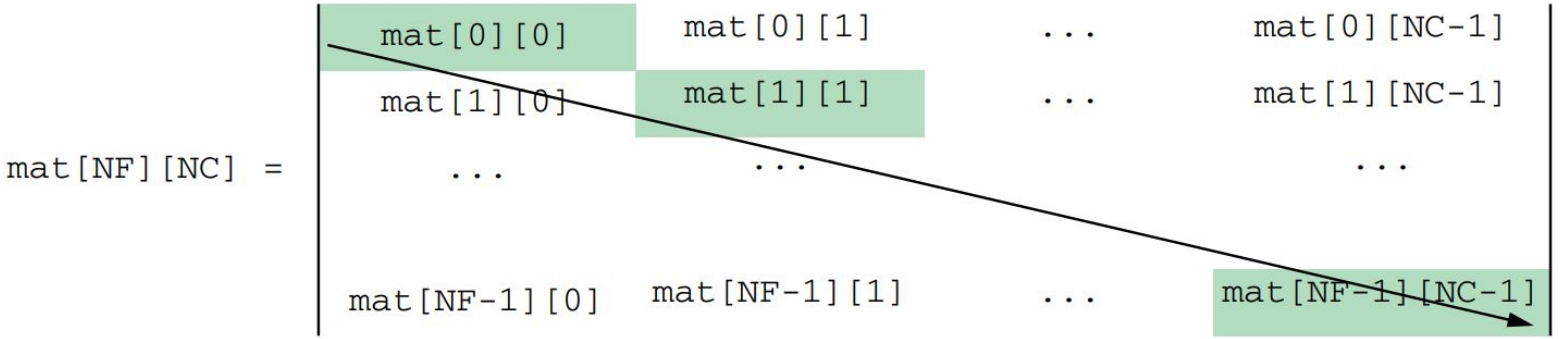


- Stride?

stride = mida d'una fila = **NC * T**

Accés seqüencial a matrius

- Recorregut de la diagonal

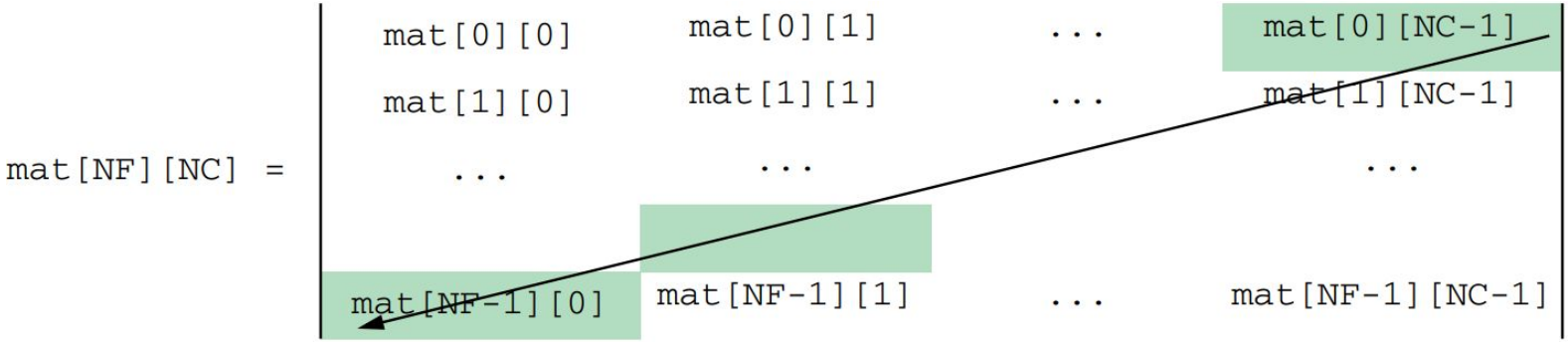


- Stride?

stride = mida d'una fila + mida d'un element = **(NC+1) * T**

Accés seqüencial a matrius

- Recorregut de la diagonal secundaria



- Stride?

stride = mida d'una fila - mida d'un element = **(NC-1) * T**

Exemple

- Tradueix a MIPS la següent funció en C

```
#define NC 100
short sumacolumna(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;
}
```

Per fer a casa....

Per fer a casa...

- Els farem la sessió de teoria del Dijous 14 de Març

Exercici 1

- Donades les següents declaracions en C, on `NF` i `NC` són constants, tradueix a MIPS la sentència marcada en verd. Suposem que `i` i `j` estan emmagatzemades en `$t0` i `$t1`, respectivament.

```
int mat[NF][NC];
int f() {
    int i, j;
    ...
    return mat[i+5][j-1];
}
```

Exercici 2

```
int mati[5][4];
void func(int veci[4]) {
  int j;
  for (j=0; j<4; j++)
    veci[j] = j;
}
```

```
void main() {
  int i;
  for (i=0; i<5; i++)
    func(&mati[i][0]);
}
```

1. Tradueix a MIPS la subrutina `func` usant accés seqüencial al vector `veci` (j està a `$t0`).
2. Tradueix a MIPS la subrutina `main` usant accés seqüencial a la matriu `mati`. Fes atenció a quins registres fas servir per guardar la variable *i* i el punter.

Exercici 4

Donada la següent funció en C:

```
short acces_aleatori(short M[][100], int i) {
        return M[i+2][i-1];
}
```

Completa els requadres del següent fragment de codi en assemblador MIPS per tal que sigui la traducció correcta de la funció anterior:

```
acces_aleatori:
        li      $t0, [          ]
        mult    $t0, [          ]
        mflo    $t0
        addu    $t0, $t0, [          ]
        lh      $v0, [          ]($t0)
        jr      $ra
```