

Estructura de Computadores

Tema 4. Matrices

Multiplicación de enteros en MIPS

- La multiplicación de dos enteros de n y m bits da un resultado de $n+m$ bits

Multiplicación de enteros en MIPS

- ▶ La multiplicación de dos enteros de n y m bits da un resultado de $n+m$ bits
- ▶ La multiplicación de dos enteros de n bits da un resultado de $2n$ bits

Multiplicación de enteros en MIPS

- ▶ La multiplicación de dos enteros de n y m bits da un resultado de $n+m$ bits
- ▶ La multiplicación de dos enteros de n bits da un resultado de $2n$ bits
- ▶ En MIPS:
 - ▶ `mult rs, rt` # `$hi:$lo <- rs * rt`

Multiplicación de enteros en MIPS

- ▶ La multiplicación de dos enteros de n y m bits da un resultado de $n+m$ bits
- ▶ La multiplicación de dos enteros de n bits da un resultado de $2n$ bits
- ▶ En MIPS:
 - ▶ `mult rs, rt # $hi:$lo <- rs * rt`
- ▶ `$hi` y `$lo` son dos registros especiales
 - ▶ No se pueden utilizar en el resto de instrucciones estudiadas hasta ahora

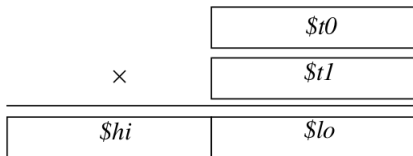
Multiplicación de enteros en MIPS

- ▶ La multiplicación de dos enteros de n y m bits da un resultado de $n+m$ bits
- ▶ La multiplicación de dos enteros de n bits da un resultado de $2n$ bits
- ▶ En MIPS:
 - ▶ `mult rs, rt` # $\$hi:\$lo \leftarrow rs * rt$
- ▶ $\$hi$ y $\$lo$ son dos registros especiales
 - ▶ No se pueden utilizar en el resto de instrucciones estudiadas hasta ahora
- ▶ Para mover el resultado a registros de propósito general:
 - ▶ `mflo rd` # $rd \leftarrow \$lo$
 - ▶ `mfhi rd` # $rd \leftarrow \$hi$

Multiplicación de enteros en MIPS

- Para calcular $\$t2 = \$t0 * \$t1$, ignorando los 32 bits de mayor peso del resultado:

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



Matrices

- ▶ Agrupación multidimensional de elementos de tipo homogéneo
- ▶ Los elementos se identifican mediante un índice en cada dimensión
- ▶ Estudiaremos matrices de dos dimensiones

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

Declaración de matrices

► En C:

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

Declaración de matrices

► En C:

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

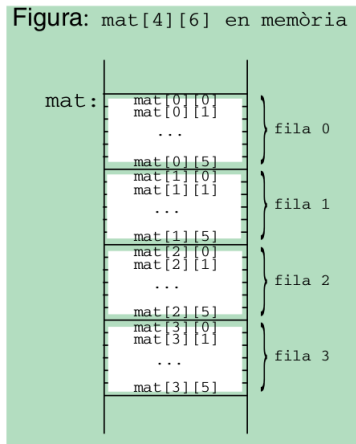
► En MIPS:

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

Almacenamiento de matrices en memoria

- En C las matrices se almacenan por filas

Figura: `mat[4][6]` en memòria



Acceso aleatorio

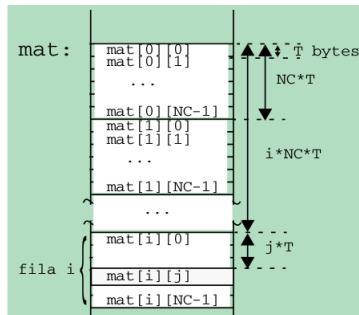
- ▶ Para acceder al elemento situado en la fila i , columna j :
 - ▶ `mat[i][j]`

Acceso aleatorio

- ▶ Para acceder al elemento situado en la fila i , columna j :
 - ▶ `mat[i][j]`
- ▶ La dirección se calcula de la siguiente forma:
 - ▶ `@mat[i][j] = mat + i*NC*T + j*T`
 - ▶ `@mat[i][j] = mat + (i*NC+j)*T`

Acceso aleatorio

- ▶ Para acceder al elemento situado en la fila i , columna j :
 - ▶ `mat[i][j]`
- ▶ La dirección se calcula de la siguiente forma:
 - ▶ `@mat[i][j] = mat + i*NC*T + j*T`
 - ▶ `@mat[i][j] = mat + (i*NC+j)*T`



Ejemplo

- ▶ Traducir a MIPS una sentencia en C, dentro de la función *func*, que accede a la matriz global *mat*. Las variables locales *i*, *j*, *k* se almacenan en los registros \$t0, \$t1, \$t2 respectivamente.

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


Ejemplo

- ▶ Traducir a MIPS una sentencia en C, dentro de la función *func*, que accede a la matriz global *mat*. Las variables locales *i*, *j*, *k* se almacenan en los registros \$t0, \$t1, \$t2 respectivamente.

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

Ejemplo

- ▶ Traducir a MIPS una sentencia en C, dentro de la función *func*, que accede a la matriz global *mat*. Las variables locales *i*, *j*, *k* se almacenan en los registros \$t0, \$t1, \$t2 respectivamente.

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

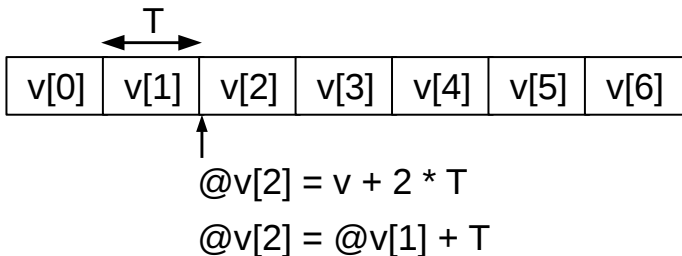
Ejemplo

- ▶ Traducir a MIPS una sentencia en C, dentro de la función *func*, que accede a la matriz global *mat*. Las variables locales *i*, *j*, *k* se almacenan en los registros \$t0, \$t1, \$t2 respectivamente.

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

Acceso secuencial a un vector

- ▶ Optimización para bucles que recorren los elementos de un vector o matriz
- ▶ La distancia en bytes (**stride**) entre las direcciones de dos elementos consecutivos debe ser constante



Ejemplo

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

Ejemplo

```
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:
```

Pasos para aplicar acceso secuencial

1. Calcular la dirección del primer elemento a recorrer, e inicializar un puntero `p` (`$t0`) con esta dirección:
 - ▶ En C: `p = array;`
 - ▶ En MIPS: `move $t1, $a0`

Pasos para aplicar acceso secuencial

1. Calcular la dirección del primer elemento a recorrer, e inicializar un puntero p ($\$t0$) con esta dirección:
 - ▶ En C: $p = \text{array};$
 - ▶ En MIPS: `move $t1, $a0`
2. Calcular el stride, restando las direcciones de dos elementos consecutivos del recorrido:
 - ▶ $\text{stride} = \text{@array}[i+1] - \text{@array}[i]$
 - ▶ $\text{stride} = (\text{array} + (i+1)*4) - (\text{array} + i*4)$
 - ▶ $\text{stride} = 4$

Pasos para aplicar acceso secuencial

1. Calcular la dirección del primer elemento a recorrer, e inicializar un puntero p ($\$t0$) con esta dirección:
 - ▶ En C: $p = \text{array};$
 - ▶ En MIPS: `move $t1, $a0`
2. Calcular el stride, restando las direcciones de dos elementos consecutivos del recorrido:
 - ▶ $\text{stride} = \text{@array}[i+1] - \text{@array}[i]$
 - ▶ $\text{stride} = (\text{array} + (i+1)*4) - (\text{array} + i*4)$
 - ▶ $\text{stride} = 4$
3. La dirección de un elemento del recorrido se calcula sumando el stride a la dirección del anterior elemento:
 - ▶ `addiu $t1, $t1, 4`

Acceso secuencial

```
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:
```

Eliminación de la variable de inducción

```
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:
```

Evaluación de la condición 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:
```

Acceso secuencial a matrices

► Recorrido de una fila

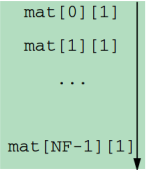
$\text{mat}[\text{NF}][\text{NC}] =$

$\text{mat}[0][0]$	$\text{mat}[0][1]$...	$\text{mat}[0][\text{NC}-1]$
$\text{mat}[1][0]$	$\text{mat}[1][1]$...	$\text{mat}[1][\text{NC}-1]$
...
$\text{mat}[\text{NF}-1][0]$	$\text{mat}[\text{NF}-1][1]$...	$\text{mat}[\text{NF}-1][\text{NC}-1]$

► ¿Stride?

Acceso secuencial a matrices

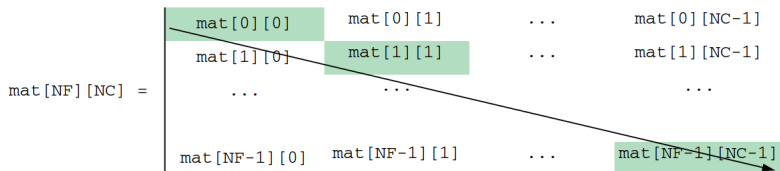
► Recorrido de una columna

$$\text{mat}[\text{NF}][\text{NC}] = \left| \begin{array}{ccc} \text{mat}[0][0] & \text{mat}[0][1] & \dots & \text{mat}[0][\text{NC}-1] \\ \text{mat}[1][0] & \text{mat}[1][1] & \dots & \text{mat}[1][\text{NC}-1] \\ \dots & \dots & & \dots \\ \text{mat}[\text{NF}-1][0] & \text{mat}[\text{NF}-1][1] & \dots & \text{mat}[\text{NF}-1][\text{NC}-1] \end{array} \right|$$


► ¿Stride?

Acceso secuencial a matrices

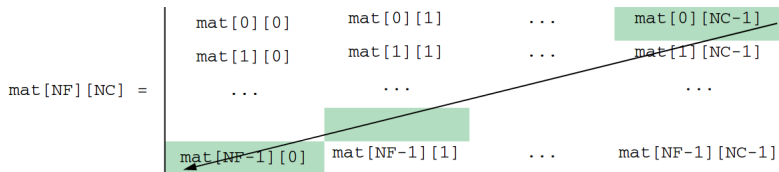
► Recorrido de la diagonal



► ¿Stride?

Acceso secuencial a matrices

► Recorrido de la diagonal secundaria



► ¿Stride?

Ejemplo

- ▶ Traduce a MIPS la siguiente función en C.

```
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;
}
```