



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 4: Matrices

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



Índice

- ❑ Instrucciones para multiplicar enteros en MIPS
- ❑ Matrices
- ❑ Acceso Secuencial a un Vector
- ❑ Acceso Secuencial a una Matriz

Instrucciones para Multiplicar Enteros en MIPS

- Para representar la multiplicación de 2 enteros de n y m bits necesitamos n+m bits.
- Caso de 2 números x e y de n bits en ca2
 - Rango x, y: $(-2^{n-1} \dots 2^{n-1}-1)$
 - Valor máximo $x*y: (-2^{n-1}) * (-2^{n-1}) = 2^{2n-2}$

n	Mínimo	Máximo	Min x*y	Max x*y	
8	-128	127	-16.256	16.384	Necesitamos 16 bits
16	-32.768	32.767	-1.073.709.056	1.073.741.824	Necesitamos 32 bits

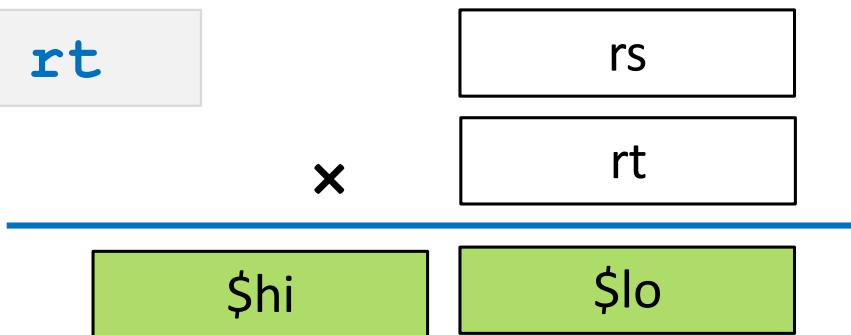
- En MIPS trabajamos con números de 32 bits
 - $32 \times 32 \text{ bits} \rightarrow 64 \text{ bits}$
 - ✓ ¿Dónde dejamos el resultado?

Para calcular la dirección de $M[i][j]$ es necesario hacer una multiplicación.

Instrucciones para Multiplicar Enteros en MIPS

- Instrucción en MIPS para multiplicar 2 números enteros de 32 bits

```
mult rs, rt # $hi:$lo = rs * rt
```



- El resultado se guarda en los registros **\$hi** y **\$lo**

- Son **registros especiales**. No se pueden usar en las instrucciones que hemos visto.
 - Para acceder a estos registros necesitamos instrucciones específicas.

```
mflo rd      # rd = $lo  
mfhi rd      # rd = $hi
```

Instrucciones para Multiplicar Enteros en MIPS

- Si tenemos el siguiente código en C

```
int a, b, d;  
...  
d = a * b;
```

- El resultado se guarda en d (32 bits) . ¿Qué hacemos con el resto de bits?
 - Truncaremos los 64 bits del producto, nos quedamos con los 32 bits bajos.
 - ✓ Possible overflow
 - ✓ En C, se ignora el overflow del producto
- En MIPS (si a, b, d están en \$t0, \$t1, \$t2)

```
mult $t0,$t1    # $hi:$lo = $t0*$t1  
mflo $t2        # $t2 = $lo
```

Matrices

Una **matriz** es una agrupación multidimensional de elementos del mismo tipo, identificados en cada dimensión por un índice.

- Los elementos se identifican por índice en cada dimensión
 - Si una dimensión tiene N elementos
 - ✓ Se indexan de 0 a $N-1$
- Sólo estudiaremos matrices de dos dimensiones
 - Hablaremos de filas y columnas

Ejemplo de Matriz

```
int M[6][8];
```

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]	M[0][6]	M[0][7]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]	M[1][6]	M[1][7]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]	M[2][6]	M[2][7]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]	M[3][6]	M[3][7]
M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]	M[4][6]	M[4][7]
M[5][0]	M[5][1]	M[5][2]	M[5][3]	M[5][4]	M[5][5]	M[5][6]	M[5][7]

Columna 2

Fila 3

Declaración de Matrices

- En C, las dimensiones de las matrices han de ser constantes o literales

```
int Mat[nFil][nCol];  
int MM[2][3]={{-1,2,3},{0,-4,7}};
```

- Si son variables globales, en MIPS quedaría así:

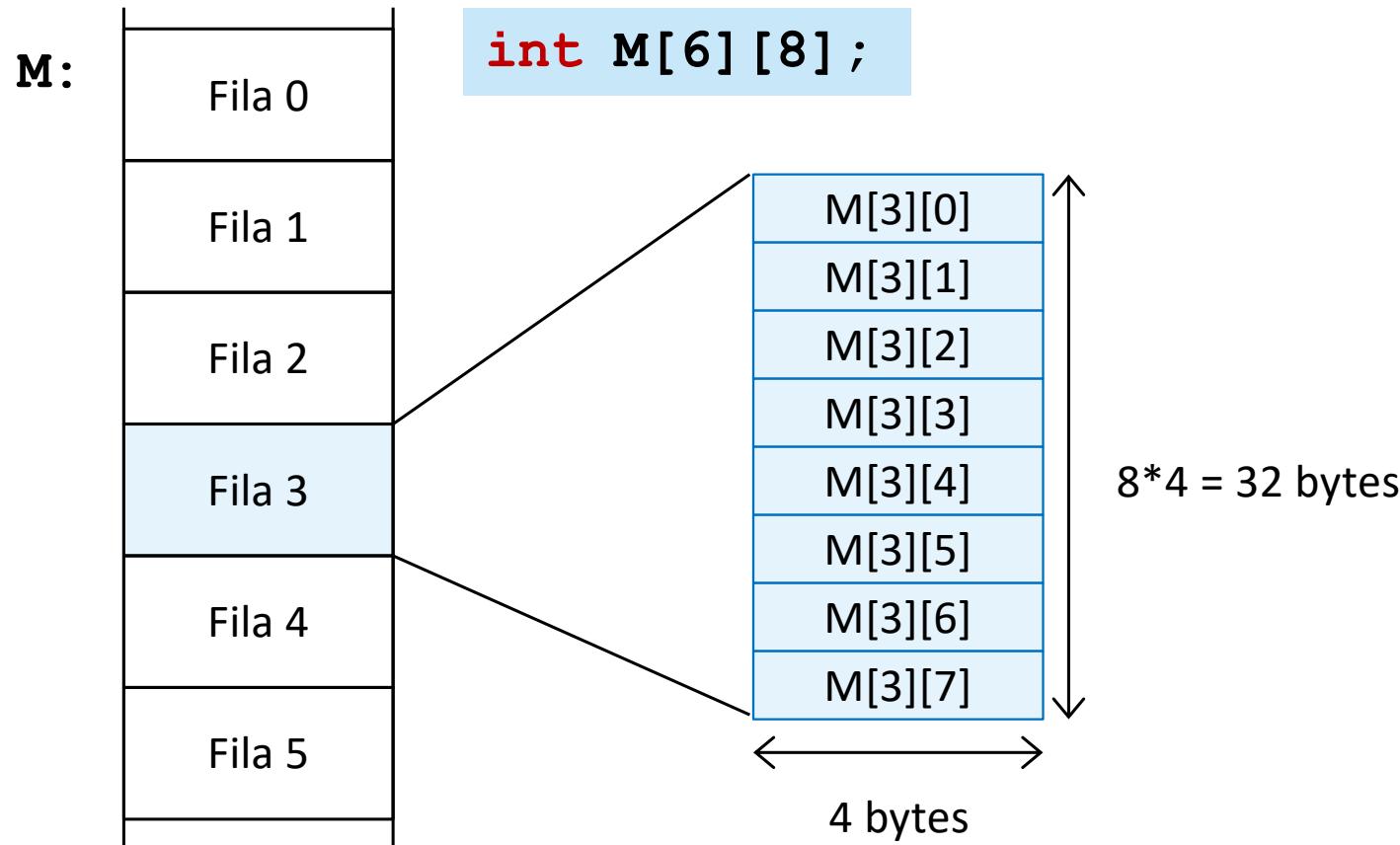
```
.data  
.align 2  
Mat:.space nFil*nCol*4  
MM: .word -1, 2, 3, 0, -4, 7
```

```
MM: .word -1, 2, 3  
.word 0, -4, 7
```

¿“más fácil de ver”?

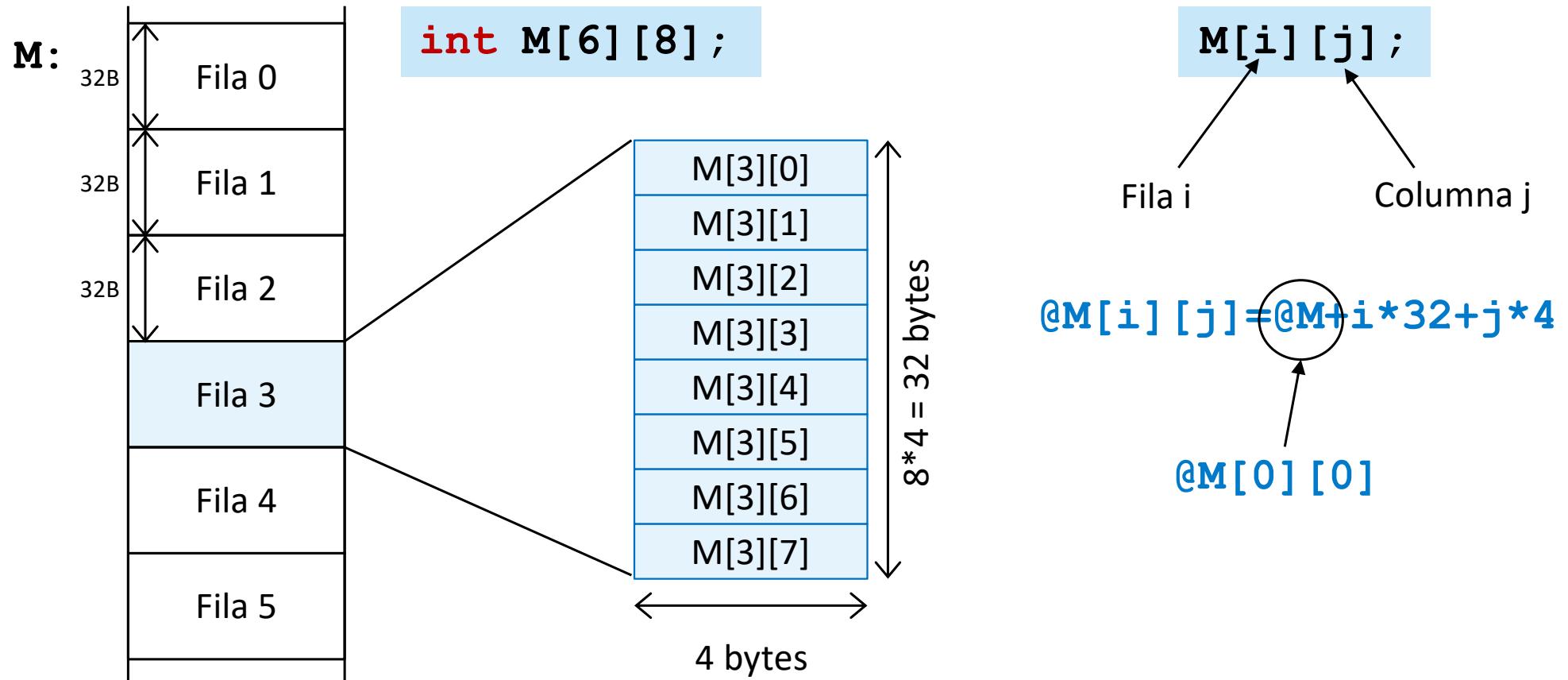
Cómo se Almacenan las Matrices en Memoria

- En C, las matrices se almacenan por FILAS en posiciones consecutivas de memoria



Acceso aleatorio a un elemento

- En C, las matrices se almacenan por FILAS en posiciones consecutivas de memoria



Acceso aleatorio a un elemento

- En general, en C, si tenemos una matriz M de elementos de tipo XXX

XXX M[nFil] [nCol];

- Para acceder al elemento de la fila i, columna j

M[i] [j]

- Hay que calcular su dirección de la siguiente forma:

@M[i] [j] = @inicioM + (i*nCol + j)*tam

siendo **tam**, el tamaño en bytes de los elementos de tipo **XXX**

Ejemplo de traducción de un Acceso Aleatorio

- Supongamos que **Mat** es global y que **i** , **j** , **k** están en **\$t0** , **\$t1** , **\$t2**

```
int Mat[nFil][nCol];
void f(){
    int i, j, k;
    ...
    k = Mat[i][j];
}
```

- Hay que calcular la siguiente dirección:

$$@Mat[i][j] = @inicioMat + (i*nCol + j)*4$$

Ejemplo de traducción de un Acceso Aleatorio

- Supongamos que **Mat** es global y que **i**, **j**, **k** están en **\$t0**, **\$t1**, **\$t2**

```
k = Mat[i][j];
```

```
# @Mat[i][j] = @inicioMat + (i*nCol + j)*4

la $t3, Mat          # t3 ← @Mat[0][0]
li $t4, nCol
mult $t0,$t4          # i*nCol
mflo $t4              # t4 ← i*nCol
addu $t4,$t4,$t1      # t4 ← i*nCol + j
sll $t4,$t4,2          # t4 ← (i*nCol + j)*4
addu $t4,$t4,$t3      # t4 ← @Mat[i][j]
lw $t2,0($t4)          # t2 ← Mat[i][j]
```

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que **Mat** es global y que **i** , **j** , **k** están en **\$t0** , **\$t1** , **\$t2**

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

```
# @Mat[i][5] = @inicioMat + (i*nCol + 5)*4
# @Mat[i][5] = @inicioMat + i*nCol*4 + 5*4

la $t3, Mat+20      # t3 ← @inicioMat+20
li $t4, nCol*4
mult $t0,$t4          # i*nCol*4
mflo $t4              # t4 ← i*nCol*4
addu $t4,$t4,$t3      # t4 ← @Mat[i][5]
lw $t2,0($t4)          # t2 ← Mat[i][5]
```

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que **Mat** es global y que **i**, **j**, **k** están en **\$t0**, **\$t1**, **\$t2**

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

```
# @Mat[i][5] = @inicioMat + (i*nCol + 5)*4
# @Mat[i][5] = @inicioMat + i*nCol*4 + 5*4

la $t3, Mat          # t3 ← @Mat[0][0]
li $t4, nCol*4
mult $t0,$t4          # i*nCol*4
mflo $t4              # t4 ← i*nCol*4
addu $t4,$t4,$t3      # t4 ← @Mat[i][0]
lw $t2,20($t4)        # t2 ← Mat[i][5]
```

Solución
equivalente

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que **Mat** es global y que **i** , **j** , **k** están en **\$t0** , **\$t1** , **\$t2**

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

```
# @Mat[3][j] = @inicioMat + (3*nCol + j)*4
# @Mat[3][j] = @inicioMat + 3*nCol*4 + j*4

la $t3, Mat+nCol*12    # t3 ← @Mat+nCol*3*4
sll $t4,$t1,2            # t4 ← j*4
addu $t4,$t4,$t3         # t4 ← @Mat[3][j]
lw $t2,0($t4)            # t2 ← Mat[3][j]
```

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que **Mat** es global y que **i** , **j** , **k** están en **\$t0** , **\$t1** , **\$t2**

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

```
# @Mat[3][j] = @inicioMat + (3*nCol + j)*4  
# @Mat[3][j] = @inicioMat + 3*nCol*4 + j*4
```

```
la $t3, Mat          # t3 ← @Mat[0][0]  
sll $t4,$t1,2        # t4 ← j*4  
addu $t4,$t4,$t3      # t4 ← @Mat[0][j]  
lw $t2,nCol*12($t4)   # t2 ← Mat[3][j]
```

Solución
equivalente

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que **Mat** es global y que **i**, **j**, **k** están en **\$t0**, **\$t1**, **\$t2**

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

```
# @Mat[i][5] = @inicioMat + (3*nCol + 5)*4  
  
la $t4, Mat+(3*nCol+5)*4    # t4 ← @Mat[3][5]  
lw $t2,0($t4)                 # t2 ← Mat[3][5]
```

```
# @Mat[i][5] = @inicioMat + (3*nCol + 5)*4  
  
la $t4, Mat                  # t4 ← @Mat[0][0]  
lw $t2,(3*nCol+5)*4($t4)    # t2 ← Mat[3][5]
```

Solución
equivalente

Optimización: Acceso Secuencial

- Situación muy habitual

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

```
clear: move $t0,$zero  
for:   bge $t0,$a1,end  
       sll $t1,$t0,2  
       addu $t2,$a0,$t1  
       sw $zero,0($t2)  
       addiu $t0,$t0,1  
       b for  
end:   jr $ra
```

- Estamos recorriendo TODOS los elementos del vector.
- No es necesario calcular la dirección de **v[i]** en cada iteración

Instrucciones Ejecutadas:
 $6 \times N + 3$

```
@v[i] = @inicioV + i*4
```

Optimización: Acceso Secuencial

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

- Estamos recorriendo TODOS los elementos del vector.
- No es necesario calcular la dirección del **v[i]** en cada iteración

```
@v[i] = @inicioV + i*4
```

- Si tenemos la dirección de **v[i]** es muy fácil calcular la dirección de **v[i+1]**

```
@v[i+1] = @v[i] + stride
```

- El **stride** depende de tipo de los elementos del vector, si son **int** el **stride** es 4.

Optimización: Acceso Secuencial

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

```
clear: move $t0,$zero  
for:   bge $t0,$a1,end  
       sll $t1,$t0,2  
       addu $t2,$a0,$t1  
       sw $zero,0($t2)  
       addiu $t0,$t0,1  
       b for  
end:   jr $ra
```

Instrucciones Ejecutadas:
 $5 \times N + 4$

```
CL1: move $t0,$zero  
      move $t1,$a0          # t1 ← @v[0]  
for:  bge $t0,$a1,end  
      sw $zero,0($t1)      # v[i] ← 0  
      addiu $t0,$t0,1  
      addiu $t1,$t1,4      # t1 ← @v[i+1]  
      b for  
end:  jr $ra
```

Optimización: Eliminar la variable de inducción

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

- ¿Cómo controlamos el bucle? Con la variable de inducción **i**.
 - El bucle itera entre **i=0** e **i=N** (cuando llega a **N**, el bucle acaba)
- ¿Podemos controlar el bucle con el puntero (**@v[i]**)?
 - **SI**, el puntero itera entre **@v[0]** y **@v[N-1]**
- ¿Podemos prescindir de la variable **i**?
 - **SI**, a condición de que no se utilice para otras cosas

OPTIMIZACIÓN: Eliminar la Variable de Inducción

Optimización: Eliminar la variable de inducción

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

- ¿Cómo lo hacemos?

1. Calcularemos la dirección del puntero después del último acceso “@v[N]”

$$\$t3 = @v[N] = @inicioV + N*4$$

2. En vez de comparar “i < N” compararemos el puntero ($\$t1$) con $\$t3$

$\$t1 < \$t3$

iAtención, MUY IMPORTANTE! Estamos comparando direcciones, en vez de usar `bge` hemos de usar `bgeu`

Optimización: Eliminar la variable de inducción

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

```
CL1: move $t0,$zero  
      move $t1,$a0  
for: bge $t0,$a1,end  
      sw $zero,0($t1)  
      addiu $t0,$t0,1  
      addiu $t1,$t1,4  
      b for  
end: jr $ra
```

Instrucciones Ejecutadas:
 $4 \times N + 5$

```
CL2: move $t1,$a0      # t1 ← @v[0]  
      sll $t2,$a1,2    # t2 ← N*4  
      addu $t3,$a0,$t2  # t3 ← @v[N]  
for: bgeu $t1,$t3,end  # compara punteros  
      sw $zero,0($t1)   # v[i] ← 0  
      addiu $t1,$t1,4    # t1 ← @v[i+1]  
      b for  
end: jr $ra
```

Optimización: Evaluar la condición al final del bucle

- Cambiaremos el while por un do-while

```
while (cond) {  
    CUERPO;  
}
```



```
do {  
    CUERPO;  
} while (cond)
```

- Pero, el bucle puede que tenga 0 iteraciones, está traducción puede ser incorrecta

```
while (cond) {  
    CUERPO;  
}
```



```
if (cond)  
    do {  
        CUERPO;  
    } while (cond)
```

- Antes de ejecutar la primera iteración hay que comprobar la condición del bucle

Optimización: Evaluar la condición al final del bucle

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

```
CL2: move $t1,$a0  
      sll $t2,$a1,2  
      addu $t3,$a0,$t2  
for: bgeu $t1,$t3,end  
      sw $zero,0($t1)  
      addiu $t1,$t1,4  
      b for  
end: jr $ra
```

Instrucciones Ejecutadas:
 $3 \times N + 5$

```
CL3: move $t1,$a0      # t1 ← @v[0]  
      sll $t2,$a1,2    # t2 ← N*4  
      addu $t3,$a0,$t2  # t3 ← @v[N]  
      bgeu $t1,$t3,end # misma cond  
do:  sw $zero,0($t1)   # v[i] ← 0  
      addiu $t1,$t1,4    # t1 ← @v[i+1]  
      bltu $t1,$t3,do    # compara punteros  
end: jr $ra
```

Optimizando

```
void clear(int v[], int N) {  
    for (int i=0; i<N; i++)  
        v[i] = 0;  
}
```

```
clear: move $t0,$zero  
for:   bge $t0,$a1,end  
       sll $t1,$t0,2  
       addu $t2,$a0,$t1  
       sw $zero,0($t2)  
       addiu $t0,$t0,1  
       b for  
end:   jr $ra
```

6 inst/iter
2 saltos/iter
 $6 \times N + 3$ inst

```
CL1: move $t0,$zero  
      move $t1,$a0  
for:  bge $t0,$a1,end  
      sw $zero,0($t1)  
      addiu $t0,$t0,1  
      addiu $t1,$t1,4  
      b for  
end:  jr $ra
```

5 inst/iter
2 saltos/iter
 $5 \times N + 4$ inst

```
CL2: move $t1,$a0  
      sll $t2,$a1,2  
      addu $t3,$a0,$t2  
for: bgeu $t1,$t3,end  
      sw $zero,0($t1)  
      addiu $t1,$t1,4  
      b for  
end: jr $ra
```

4 inst/iter
2 saltos/iter
 $4 \times N + 5$ inst

```
CL3: move $t1,$a0  
      sll $t2,$a1,2  
      addu $t3,$a0,$t2  
      bgeu $t1,$t3,end  
do:   sw $zero,0($t1)  
      addiu $t1,$t1,4  
      bltu $t1,$t3,do  
end:  jr $ra
```

3 inst/iter
1 salto/iter
 $3 \times N + 5$ inst

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]
M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]
M[5][0]	M[5][1]	M[5][2]	M[5][3]	M[5][4]	M[5][5]

```
for (int j=0; j<6; j++)  
    ... M[3][j] ...  
}
```

□ Recorrido secuencial de 1 fila. ¿Cual es el stride?

- En el ejemplo es 4 (tamaño de un `int`)
- En general, es T (tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

`int M[6][6];`

M[0][0] +0	M[0][1] +4	M[0][2] +8	M[0][3] +12	M[0][4] +16	M[0][5] +20
M[1][0] +24	M[1][1] +28	M[1][2] +32	M[1][3] +36	M[1][4] +40	M[1][5] +44
M[2][0] +48	M[2][1] +52	M[2][2] +56	M[2][3] +60	M[2][4] +64	M[2][5] +68
M[3][0] +72	M[3][1] +76	M[3][2] +80	M[3][3] +84	M[3][4] +88	M[3][5] +92
M[4][0] +96	M[4][1] +100	M[4][2] +104	M[4][3] +108	M[4][4] +112	M[4][5] +116
M[5][0] +120	M[5][1] +124	M[5][2] +128	M[5][3] +132	M[5][4] +136	M[5][5] +140

```
for (int j=0; j<6; j++)
    ... M[3][j] ...
}
```

stride = 4

□ Recorrido secuencial de 1 fila. ¿Cual es el stride?

- En el ejemplo es 4 (tamaño de un `int`)
- En general, es T (tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]
M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]
M[5][0]	M[5][1]	M[5][2]	M[5][3]	M[5][4]	M[5][5]

```
for (int i=0; i<6; i++)  
    ... M[i][2] ...  
}
```

□ Recorrido secuencial de 1 columna. ¿Cual es el stride?

- En el ejemplo es 6*4 (tamaño de una fila de 6 elementos de tipo **int**)
- En general, es nCol*T (T es el tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

`int M[6][6];`

M[0][0] +0	M[0][1] +4	M[0][2] +8	M[0][3] +12	M[0][4] +16	M[0][5] +20
M[1][0] +24	M[1][1] +28	M[1][2] +32	M[1][3] +36	M[1][4] +40	M[1][5] +44
M[2][0] +48	M[2][1] +52	M[2][2] +56	M[2][3] +60	M[2][4] +64	M[2][5] +68
M[3][0] +72	M[3][1] +76	M[3][2] +80	M[3][3] +84	M[3][4] +88	M[3][5] +92
M[4][0] +96	M[4][1] +100	M[4][2] +104	M[4][3] +108	M[4][4] +112	M[4][5] +116
M[5][0] +120	M[5][1] +124	M[5][2] +128	M[5][3] +132	M[5][4] +136	M[5][5] +140

```
for (int i=0; i<6; i++)
    ... M[i][2] ...
}
```

stride = 24

□ Recorrido secuencial de 1 columna. ¿Cual es el stride?

- En el ejemplo es 6*4 (tamaño de una fila de 6 elementos de tipo `int`)
- En general, es nCol*T (T es el tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]
M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]
M[5][0]	M[5][1]	M[5][2]	M[5][3]	M[5][4]	M[5][5]

```
for (int i=0; i<6; i++)
    ... M[i][i] ...
}
```

□ Recorrido secuencial de la diagonal principal. ¿Cual es el stride?

- En el ejemplo es $7*4$ (tamaño de una fila de 6 elementos + 1, de tipo `int`)
- En general, es $(nCol+1)*T$ (T es el tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0] +0	M[0][1] +4	M[0][2] +8	M[0][3] +12	M[0][4] +16	M[0][5] +20
M[1][0] +24	M[1][1] +28	M[1][2] +32	M[1][3] +36	M[1][4] +40	M[1][5] +44
M[2][0] +48	M[2][1] +52	M[2][2] +56	M[2][3] +60	M[2][4] +64	M[2][5] +68
M[3][0] +72	M[3][1] +76	M[3][2] +80	M[3][3] +84	M[3][4] +88	M[3][5] +92
M[4][0] +96	M[4][1] +100	M[4][2] +104	M[4][3] +108	M[4][4] +112	M[4][5] +116
M[5][0] +120	M[5][1] +124	M[5][2] +128	M[5][3] +132	M[5][4] +136	M[5][5] +140

```
for (int i=0; i<6; i++)  
    ... M[i][i] ...  
}
```

stride = 28

□ Recorrido secuencial de la diagonal principal. ¿Cuál es el stride?

- En el ejemplo es $7 * 4$ (tamaño de una fila de 6 elementos + 1, de tipo `int`)
- En general, es $(nCol+1) * T$ (T es el tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]
M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]
M[5][0]	M[5][1]	M[5][2]	M[5][3]	M[5][4]	M[5][5]

```
for (int i=0; i<6; i++)  
    ... M[i][5-i] ...  
}
```

□ Recorrido secuencial de la diagonal secundaria. ¿Cual es el stride?

- En el ejemplo es $5*4$ (tamaño de una fila de 6 elementos -1, de tipo `int`)
- En general, es $(n\text{Col}-1)*T$ (T es el tamaño de los elementos de la matriz)

Acceso secuencial a una matriz

```
int M[6][6];
```

M[0][0] +0	M[0][1] +4	M[0][2] +8	M[0][3] +12	M[0][4] +16	M[0][5] +20
M[1][0] +24	M[1][1] +28	M[1][2] +32	M[1][3] +36	M[1][4] +40	M[1][5] +44
M[2][0] +48	M[2][1] +52	M[2][2] +56	M[2][3] +60	M[2][4] +64	M[2][5] +68
M[3][0] +72	M[3][1] +76	M[3][2] +80	M[3][3] +84	M[3][4] +88	M[3][5] +92
M[4][0] +96	M[4][1] +100	M[4][2] +104	M[4][3] +108	M[4][4] +112	M[4][5] +116
M[5][0] +120	M[5][1] +124	M[5][2] +128	M[5][3] +132	M[5][4] +136	M[5][5] +140

```
for (int i=0; i<6; i++)  
    ... M[i][5-i] ...  
}
```

stride = 20

□ Recorrido secuencial de la diagonal secundaria. ¿Cuál es el stride?

- En el ejemplo es 5×4 (tamaño de una fila de 6 elementos -1, de tipo `int`)
- En general, es $(n\text{Col}-1) \times T$ (T es el tamaño de los elementos de la matriz)

Ejemplo de Revisión

- Traducir a MIPS la función sumC (recorre la columna col)

```
short sumC(short M[] [NC], int col, int NF) {  
    int i;  
    short suma = 0;  
    for (i=0; i<NF; i++)  
        suma = suma + M[i] [col];  
    return suma;  
}
```

- La subrutina es uninivel, no hace falta usar registros seguros
- La dirección de **M[i] [col]** es: **@M[i][col] = @M +i*NC*2 + col*2**
- **@M+col*2** es invariante del bucle, lo calcularemos fuera
- **NC*2** es invariante de bucle, lo calcularemos fuera

Ejemplo de Revisión. Acceso Aleatorio

- Traducir a MIPS la función sumC (recorre la columna col)

```
short sumC(short M[][][NC],  
          int col, int NF) {  
    int i;  
    short suma = 0;  
    for (i=0; i<NF; i++)  
        suma = suma + M[i][col];  
    return suma;  
}
```

$\text{@M}[i][\text{col}] = \text{@M} +$
 $+ (\text{i} * \text{NC} + \text{col}) * 2$

Instrucciones Ejecutadas: $8 \times \text{NF} + 7$

```
sumC: move $v0,$zero      # suma←0  
       move $t0,$zero      # i←0  
       li $t1,NC*2          # NC*2  
       sll $t2,$a1,1         # col*2  
       addu $t2,$a0,$t2      # @M+col*2  
for:   bge $t0,$a2,end     # si i>=NF end  
       mult $t0,$t1          # i*NC*2  
       mflo $t3  
       addu $t3,$t2,$t3      # @M[i][col]  
       lh $t4,0($t3)         # t4 ← M[i][col]  
       addu $v0,$v0,$t4      # suma+M[i][col]  
       addiu $t0,$t0,1         # i++  
b for  
end:  jr $ra
```

Ejemplo de Revisión. Acceso Secuencial

- Traducir a MIPS la función sumC (recorre la columna col)

```
short sumC(short M[] [NC], int col, int NF) {
    int i;
    short suma = 0;
    for (i=0; i<NF; i++)
        suma = suma + M[i] [col];
    return suma;
}
```

- La primera vez que accedemos, es a **M[0][col]** la dirección es $\text{@M}+\text{col}*2$
- El stride (la distancia) entre dos accesos consecutivos es:
$$\begin{aligned} \text{@M[i+1][col]} - \text{@M[i][col]} &= \\ &= \text{@M} + (\text{i}+1) * \text{NC}*2 + \text{col}*2 - \text{@M} + \text{i} * \text{NC}*2 + \text{col}*2 = \\ &= \text{NC}*2 \end{aligned}$$

Ejemplo de Revisión. Acceso Secuencial

- Traducir a MIPS la función sumC (recorre la columna col)

```
short sumC(short M[] [NC],  
          int col, int NF) {  
    int i;  
    short suma = 0;  
    for (i=0; i<NF; i++)  
        suma = suma + M[i] [col];  
    return suma;  
}
```

```
sumC: move $v0,$zero  
      move $t0,$zero  
      li $t1,NC*2  
      sll $t2,$a1,1  
      addu $t2,$a0,$t2  
for: bge $t0,$a2,end  
      mult $t0,$t1  
      mflo $t3  
      addu $t3,$t2,$t3  
      lh $t4,0($t3)  
      addu $v0,$v0,$t4  
      addiu $t0,$t0,1  
      b for  
end: jr $ra
```

```
sumC: move $v0,$zero      # suma=0  
      move $t0,$zero      # i=0  
      sll $t2,$a1,1       # col*2  
      addu $t3,$a0,$t2    # @M[0][col]  
for: bge $t0,$a2,end     # si i>=NF end  
      lh $t4,0($t3)       # t4 = M[i][col]  
      addu $v0,$v0,$t4    # suma+M[i][col]  
      addiu $t3,$t3,NC*2 # @M[i+1][col]  
      addiu $t0,$t0,1     # i++  
      b for  
end: jr $ra
```

El bucle tiene 2 instrucciones menos y nos ahorraremos la multiplicación.
Instrucciones Ejecutadas: $6 \times NF + 6$.

Ejemplo de Revisión 2

- Traducir a MIPS la siguiente función

```
void XX(int C[M][N], int A[M][N], int B[M][N]) {
    int i,j;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

- Primero haremos la versión convencional con acceso secuencial a las matrices:
 - $\text{@C}[i][j] = \text{@C}[0][0] + (i * N + j) * 4$
 - $\text{@A}[i][j] = \text{@A}[0][0] + (i * N + j) * 4$
 - $\text{@B}[i][j] = \text{@B}[0][0] + (i * N + j) * 4$

Ejemplo de Revisión 2

```
void XX(int C[M][N], int A[M][N], int B[M][N]) {
    int i,j;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

```
# $t2 ← B[i][j]
mult $t0,$t6      # i*N
mflo $t5          # t5←i*N
addu $t5,$t5,$t1 # i*N+j
sll $t5,$t5,2     # (i*N+j)*4
addu $t2,$a2,$t5 # t2←@B[i][j]
lw $t2,0($t2)     # t2←B[i][j]
```

```
# $t3 ← A[i][j]
addu $t3,$a1,$t5 # t3←@A[i][j]
lw $t3,0($t3)     # t3←A[i][j]
```

```
# C[i][j] ← $t4
addu $t2,$a0,$t5 # t2←@A[i][j]
sw $t4,0($t2)     # A[i][j]←t4
```

```
XX:   li $t7,M
      li $t6,N
      move $t0,$zero      # i←0
fori: bge $t0,$t7,endi  # si i>=M goto endi

      move $t1,$zero      # j←0
forj: bge $t1,$t6,endj  # si j>=N goto endj

      $t2 ← B[i][j]
      $t3 ← A[i][j]
      addu $t4,$t3,$t2    # t4←A[i][j]+B[i][j]
      C[i][j] ← $t4

      addiu $t1,$t1,1      # j++
      b forj
endj:
      addiu $t0,$t0,1      # i++
      b fori
endi:
      jr $ra

@ABC[i][j] = @ABC[0][0] + (i*N + j) * 4
```

Ejemplo de Revisión 2. Acceso Secuencial

```
void XX(int C[M][N], int A[M][N], int B[M][N]) {
    int i,j;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

Si recorremos una matriz por filas, podemos acceder a ella de forma secuencial.

```
XX: li $t7,M*N
     move $t4,$a0          # t4←@C[0][0]
     move $t5,$a1          # t5←@A[0][0]
     move $t6,$a2          # t6←@B[0][0]
     move $t0,$zero         # i←0
for: bge $t0,$t7,end      # si i>=M*N goto end
     lw $t2,0($t6)          # t2 ← B[i][j]
     lw $t3,0($t5)          # t3 ← A[i][j]
     addu $t1,$t3,$t2       # t1←A[i][j]+B[i][j]
     sw $t1,0($t4)          # C[i][j] ← t1
     addiu $t0,$t0,1
     addiu $t6,$t6,4
     addiu $t5,$t5,4
     addiu $t4,$t4,4
b for
end:
jr $ra
```

Ejemplo de Revisión 2

```
XX:    li $t7,M
      li $t6,N
      move $t0,$zero      # i←0
fori: bge $t0,$t7,endi   # si i>=M goto endi
      move $t1,$zero      # j←0
forj: bge $t1,$t6,endj   # si j>=N goto endj
      mult $t0,$t6
      mflo $t5
      addu $t5,$t5,$t1      # i*N
      sll $t5,$t5,2        # (i*N+j)*4
      addu $t2,$a2,$t5
      lw $t2,0($t2)         # t2←B[i][j]
      addu $t3,$a1,$t5
      lw $t3,0($t3)
      addu $t4,$t3,$t2
      addu $t2,$a0,$t5
      sw $t4,0($t2)         # A[i][j]←t4
      addiu $t1,$t1,1       # j++
      b forj
endj: addiu $t0,$t0,1     # i++
      b fori
endi: jr $ra
```

```
void XX(int C[M][N], int A[M][N], int B[M][N]){
    int i,j;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

```
XX:    li $t7,M*N
      move $t4,$a0          # t4←@C[0][0]
      move $t5,$a1          # t5←@A[0][0]
      move $t6,$a2          # t6←@B[0][0]
      move $t0,$zero         # i←0
for:   bge $t0,$t7,end    # si i>=M*N goto end
      lw $t2,0($t6)         # t2 ← B[i][j]
      lw $t3,0($t5)         # t3 ← A[i][j]
      addu $t1,$t3,$t2      # t1←A[i][j]+B[i][j]
      sw $t1,0($t4)         # C[i][j] ← t1
      addiu $t0,$t0,1        # i++
      addiu $t6,$t6,4
      addiu $t5,$t5,4
      addiu $t4,$t4,4
      b for
end:  jr $ra
```

Instrucciones Ejecutadas $\approx (14 \times N + 4) \times M$

Instrucciones Ejecutadas $\approx 10 \times N \times M$



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 4: Matrices

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



Problema 4.14

4.14 La funció fD escriu els elements de la fila i de la matriu dispersa representada pels vectors A, IA i JA (que rep com a paràmetres), en les corresponents posicions de la matriu mat (que és una variable global):

```
int mat[N][M];
void fD(int *A, int *IA, int *JA, int i) {
    int k;
    for (k=IA[i]; k<IA[i+1]; k++)
        mat[i][JA[k]] = A[k];
}
```

- a) Dels següents accisos a memòria que fa aquest bucle, ¿quins es poden traduir fent servir la tècnica d'accés seqüencial, i quins sols es poden traduir amb accés aleatori?
`IA[i], IA[i+1], JA[k], mat[i][JA[k]], A[k]`
- b) Tradueix la funció fD usant la tècnica d'accés seqüencial per als recorreguts que ho permetin, suposant que k es guarda al registre \$t0.

Problema 4.14

- ¿Qué accesos pueden hacerse de forma secuencial ?

```
int mat[N][M];
void fD(int *A, int *IA, int *JA, int i) {
    int k;
    for (k=IA[i]; k<IA[i+1]; k++)
        mat[i][JA[k]] = A[k];
}
```



```
int mat[N][M];
void fD(int *A, int *IA, int *JA, int i) {
    int k, tmp;
    for (k=IA[i]; k<IA[i+1]; k++) {
        tmp = JA[k]
        mat[i][tmp] = A[k];
    }
}
```

Podemos acceder de forma **secuencial** a:

- A[k]
- JA[k]

Hay que acceder de forma **aleatoria** a:

- IA[i]
- mat[i][tmp]

Problema 4.14

□ Traducid a MIPS

```
int mat[N][M];
void fD(int *A, int *IA, int *JA, int i) {
    int k;
    for (k=IA[i]; k<IA[i+1]; k++)
        mat[i][JA[k]] = A[k];
}
```

```
@IA[i] = @IA + i*4 = a1 + a3*4
@IA[i+1] = @IA + i*4 + 4 = a1 + a3*4 + 4
```

```
@JA[k] = @JA + k*4 = a2 + t0*4
@a[k] = @A + k*4 = a0 + t0*4
```

```
@mat[i][tmp] = @mat + i*M*4 + tmp*4
@mat[i][tmp] = @mat + a3*M*4 + tmp*4
```

```
fD: sll $t0,$a3,2      # t0 = i*4
     addu $t0,$t0,$a1 # t0 = @IA[i]
     lw $t0,0($t0)      # t0 = IA[i], t0=k
     lw $t1,4($t0)      # t1 = IA[i+1]
     sll $t2,$t0,2
     addu $t3,$t2,$a2 # t3 = @JA[k]
     addu $t2,$t2,$a0 # t2 = @A[k]
     li $t6,M*4
     mult $t6,$a3
     mflo $t6           # t6 = i*M*4
     la $t5,mat
     addu $t5,$t5,$t6 # t5 = @mat + i*M*4
for: bge $t0,$t1,end # si (k>= IA[i+1]) end
     lw $t6,0($t3)      # t6 = JA[k]
     sll $t6,$t6,2      # t6*4 (tmp*4)
     addu $t6,$t5,$t6 # t6 = @mat + i*M*4 + tmp*4
     lw $t4,0($t2)      # t4 = A[k]
     sw $t4,0($t6)      # mat[i][JA[k]] = A[k]
     addu $t2,$t2,4      # t2 = @A[k+1]
     addu $t3,$t3,4      # t3 = @JA[k+1]
     addu $t0,$t0,1      # k++
     b for
end: jr $ra
```