Apunts del Tema 4
Matrius

Joan Manuel Parcerisa

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

Licencia Creative Commons

Tema 4. Matrius

1. Les instruccions MIPS *mult*, *mflo*, *mfhi*[21]

En general, la multiplicació de dos nombres enters de n i m bits dóna un resultat de $n+m$ bits (excepte per al cas n=1, ja que no produeix carry). En particular, la multiplicació de dos nombres de n bits produeix un nombre de $2n$ bits. En MIPS, la instrucció:

```
mult rs, rt          # $hi:$lo <- rs * rt
```

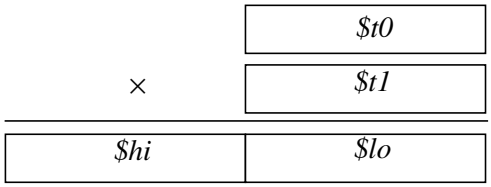
multiplica els dos registres font *rs* i *rt* (de 32 bits) i deixa el resultat (de 64 bits) en la parella de registres implícits *$hi* (32 bits de més pes) i *$lo* (32 bits de menys pes).

Els registres *$hi* i *$lo* són especials i no poden ser utilitzats en cap de les instruccions estudiades fins ara. Per copiar el seu valor a registres de propòsit general, es fan servir les instruccions especials:

```
mflo rd              # rd <- $lo
mfhi rd              # rd <- $hi
```

Per exemple, si volem calcular *$t2 = $t0 * $t1* (ignorant els 32 bits de més pes) escriurem:

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



Nou

2. Matrius

Una matriu és una agrupació multidimensional d'elements de tipus homogenis, els quals s'identifiquen per un índex en cada dimensió. Estudiarem les matrius de 2 dimensions, però els conceptes són extrapolables a qualsevol nombre de dimensions:

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

21. La multiplicació entera s'explicarà en detall al Tema 5. Aquí sols s'expliquen nocions bàsiques per poder utilitzar-les en el càlcul d'adreces d'elements de les matrius.

2.1 Declaració i emmagatzematge

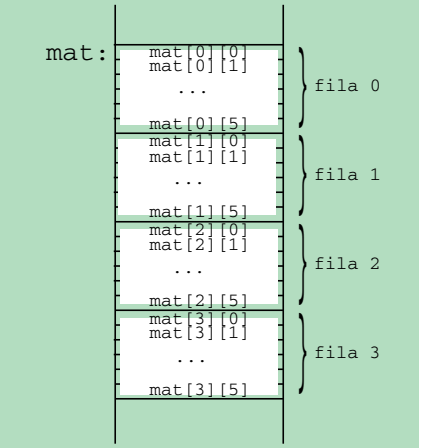
En C, declarem la variable global *mat* com una matriu de NF files i NC columnes (NF i NC són constants) d'elements *int,* així :

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

En C, els elements es guarden a memòria en posicions consecutives a partir de l'adreça *mat* "per files", és a dir, primer tots els de la primera fila, després els de la segona fila, etc. Aquest és un conveni arbitrari (en Fortran, les matrius es guarden "per columnes"). Les matrius han de respectar les regles d'alineació dels elements (*mat* es guardarà doncs en una adreça múltiple de 4). En MIPS serà:

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

Figura: `mat[4][6]` en memòria



2.2 Accés a un element qualsevol (aleatori)

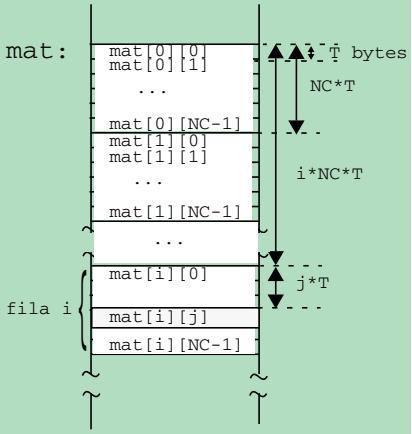
Sigui *mat* una matriu d'elements de mida *T*. Per accedir a l'element situat a la fila *i*, columna *j*, ho expressem així:

```
mat[i][j]
```

Com que la matriu s'emmagatzema per files, l'adreça d'aquest element es calcularà així (vegi's la Figura):

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

Figura: offset de `mat[i][j]`



EXEMPLE1: Volem traduir una sentència en C, dins la funció *func*, que accedeix a la matriu global *mat*:

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

En MIPS, suposant que *i*, *j*, *k* ocupen $t0, $t1, $t2, la traducció de la sentència serà:

```
     la     $t3, mat
     li     $t4, NC
     mult   $t4, $t0                # ... = i*NC
     mflo   $t4                     # $t4 = ...
     addu   $t4, $t4, $t1           # $t4 = i*NC + j
     sll    $t4, $t4, 2             # $t4 = (i*NC + j)*4
     addu   $t3, $t3, $t4           # $t3 = mat + (i*NC + j)*4
     lw     $t2, 0($t3)             # k = mat[i][j]
```

EXEMPLE2: En C, amb les mateixes declaracions, si la columna és constant:

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

En MIPS serà:

```
     la     $t3, mat + 5*4
     li     $t4, NC*4
     mult   $t4, $t0                  # ... = i*NC*4
     mflo   $t4                       # $t4 = ...
     addu   $t3, $t3, $t4             # $t3 = mat + i*NC*4 + 5*4
     lw     $t2, 0($t3)               # k = mat[i][5]
```

EXEMPLE3: En C, amb les mateixes declaracions, si la fila és constant:

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

En MIPS serà:

```
     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]
```

EXEMPLE4: En C, amb les mateixes declaracions, si fila i columna són constants:

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

En MIPS serà:

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

2.14

3. Accés seqüencial a un vector

Quan un bucle recorre els elements d'un vector o matriu sovint apareix la possibilitat d'aplicar una optimització anomenada "accés seqüencial". La condició perquè es pugui aplicar aquesta optimització és que la distància en bytes entre les adreces de dos elements consecutius del recorregut sigui constant, la qual anomenem *stride*.

EXEMPLE5: Suposem en C el següent recorregut dels elements del vector array:

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

En MIPS, una primera versió no optimitzada del bucle seria:

```
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:
```

Observem en aquest exemple que l'*stride* és de 4 bytes i és constant. L'optimització consisteix a calcular l'adreça de cada element sumant l'stride a l'adreça de l'element anterior del recorregut. Aquesta operació sol ser més simple que calcular-la cada vegada a partir de l'adreça inicial i l'índex. Sigui el recorregut format per la seqüència d'elements

$$n_0, n_1, n_2, \ldots \quad n_i, n_{i+1}, \ldots$$

Si les adreces de qualsevols dos elements consecutius estan separades per una distància constant ($@n_{i+1}$ - $@n_i$ = *stride)*, llavors podem calcular inductivament cada adreça a partir de l'anterior

$$@n_{i+1} = @n_i + stride$$

excepte la primera: $@n_0$

3.1 Passos a fer

Sigui el bucle de l'exemple 5 anterior, el qual admet accés seqüencial. Llavors,

a) Calculem l'adreça del primer element a recórrer, i inicialitzem un punter p (registre $t1) amb aquesta adreça:

```
En C:
       p = array;          // p = @array[0] = array + 0*4
En MIPS:
       move $t1, $a0       # inicialitzem punter $t1 = @array[0]
```

b) Calculem l'stride (en aquest exemple és òbviament 4), restant les adreces de dos elements consecutius del recorregut. Llavors accedirem a memòria sempre desreferenciant el punter, i al final de cada iteració li sumarem l'stride, perquè apunti al següent element

```
stride = @array[i+1] - @array[i]
       = (array + (i+1)*4) - (array + i*4)
       = 4
En C:
       *p = 0;             // desreferenciem p
       p++;                // fem que p apunti "al següent element"
                           // sumant-li 1 (aritmètica de punters!)
En MIPS:
       sw     $zero, 0($t1) # accés a memòria usant el punter $t1
       addiu  $t1, $t1, 4   # $t1 = $t1 + stride
```

3.2 Resultat

EXEMPLE6: El codi MIPS de l'exemple 5 optimitzat queda de la següent manera:

```
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:
```

Què hi hem guanyat? El bucle *loop2* té una instrucció menys que *loop1*.

3.3 Optimització: eliminació de la variable d'inducció

Encara podem millorar la solució, eliminant el comptador: per controlar el bucle comprovarem si el punter apunta més enllà del darrer element a recórrer. Alerta! perquè cal usar una comparació de naturals, ja que comparem adreces (*bgeu* en lloc de *bge*).

EXEMPLE7: Optimitzem el codi MIPS de *loop2*, eliminant el comptador:

```
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:
```

Què hi hem guanyat? Observem que *loop3* té dues instruccions menys que *loop1*.

3.4 Optimització: avaluació de la condició al final del bucle

Encara podem millorar la solució, eliminant un dels salts: fent l'avaluació de la condició al final (com un do-while). No obstant, cal assegurar-se que el codi manté la semàntica d'un while en el cas de zero iteracions. Si existeix aquesta possibilitat, cal afegir una comprovació inicial abans d'entrar en el bucle (instrucció *bgeu* en el següent exemple):

EXEMPLE8: Optimitzem el codi MIPS avaluant la condició al final:

```
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:
```

Què hi hem guanyat? Observem que *loop4* té tres instruccions menys que *loop1*.

3.5 Expressar l'accés seqüencial en alt nivell

El llenguatge C també permet escriure en alt nivell l'accés seqüencial usant punters. El següent codi és la traducció literal de l'exemple 7, en C:

```
void clear3(int *array, int nelem)
{
     int *p;
     for (p=array; p < &array[nelem]; p=p+1)
           *p = 0;
}
```

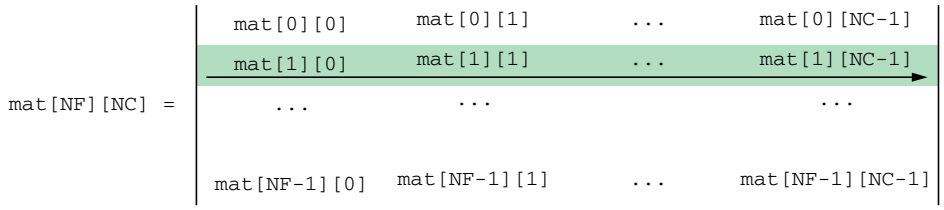
Però no és indispensable usar punters en C per fer que el compilador generi el codi més optimitzat. Si el compilador de C és prou bo, ha de ser capaç d'optimitzar d'igual manera el codi de *clear1* i el codi de *clear3*, generant el mateix resultat.

2.14

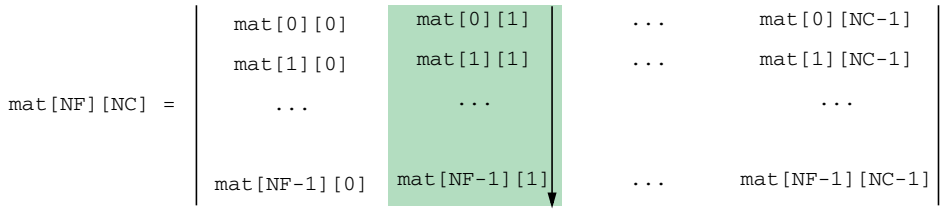
4. Recorreguts d'una matriu amb accés seqüencial

També podem aplicar l'accés seqüencial a molts recorreguts d'una matriu:

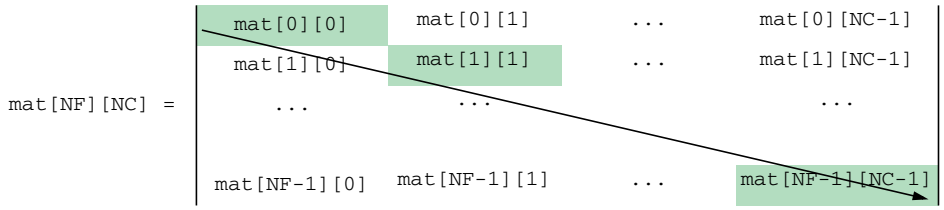
a) Una fila: stride = mida d'un element = T



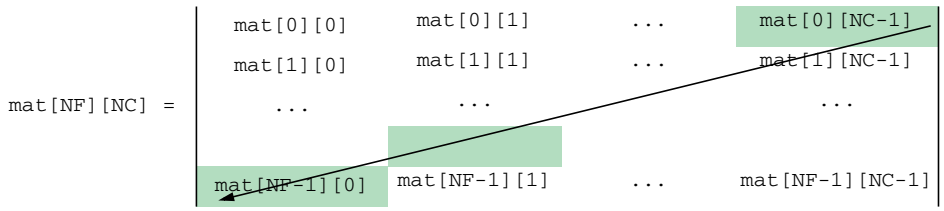
b) Una columna: stride = mida d'una fila = NC*T



c) La diagonal principal: Stride = mida de fila + mida d'elem. = (NC+1)*T



d) La diagonal secundària: stride = mida de fila - mida d'elem. = (NC-1)*T



4.1 Exemple de revisió

EXEMPLE8: La següent funció en C recorre una columna d'una matriu. Traduir-la a MIPS usant accés aleatori:

```
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;
}
```

L'adreça de `mat[i][col]` és:

```
@mat[i][col] = mat + i*NC*2 + col*2
```

El codi en MIPS és:

```
sumacolumna:
     move   $v0, $zero          # suma = 0  (en $v0, serà el resultat)
     move   $t0, $zero          # i = 0
     li     $t1, NC*2           # $t1 = NC*2  (invariant)
     sll    $t2, $a1, 1
     addu   $t2, $a0, $t2       # $t2 = mat + col*2  (invariant)
for:
     bge    $t0, $a2, fi        # salta si i<nfiles és fals
     mult   $t0, $t1            # i*NC*2
     mflo   $t3
     addu   $t3, $t2, $t3       # $t3 = mat + col*2 + i*NC*2

     lh     $t3, 0($t3)         # llegim l'element de mat
     addu   $v0, $v0, $t3       # el sumem
     addiu  $t0, $t0, 1         # i++
     b      for
fi:
     jr     $ra
```

EXEMPLE9: Traduir a MIPS l'anterior funció usant accés seqüencial:

a) L'adreça inicial del punter serà: `@mat[0][col] = mat + col*2`

b) L'stride serà: `@mat[i+1][col] - @mat[i][col] =`

`= mat + (i+1)*NC*2 + col*2 -(mat + i*NC*2 + col*2) = NC*2`

El codi en MIPS queda així:

```
sumacolumna:
     sll    $t1, $a1, 1         # col*2
     addu   $t1, $t1, $a0       # $t1 = mat + col*2
     move   $v0, $zero          # suma = 0  (en $v0, serà el resultat)
     move   $t0, $zero          # i = 0
for:
     bge    $t0, $a2, fi        # salta si i<nfiles és fals
     lh     $t2, 0($t1)         # accés a memòria usant el punter
     addiu  $t1, $t1, NC*2      # sumem l'stride a $t0
     addu   $v0, $v0, $t2       # sumem l'element
     addiu  $t0, $t0, 1         # i++
     b      for
fi:
     jr     $ra
```

L'accés seqüencial es pot expressar també en C així:

```
short sumacolumna (short mat[][NC], int col, int nfiles) {
     int i;
     short *p;
     short suma = 0;

     p = &mat[0][col];
     for (i=0; i<nfiles; i++) {
            suma = suma + *p;
            p = p + NC;
     }
     return suma;
}
```