Facultad de Informática
Universidad Complutense de Madrid

Fundamentos de Computadores II - Práctica 5
Matrices

15 de enero de 2025

1. Objetivos

En esta práctica vamos a explorar el manejo de matrices y a consolidar la programación mixta en C y ensamblador. También veremos de manera práctica las diferencias entre variables globales y variables locales, así como la manera de usar el marco de activación de las funciones como lugar en donde ubicar estas últimas.

2. Matrices

Normalmente, los elementos de una matriz se almacenan en memoria ordenados por filas, es decir, de izquierda a derecha y de arriba a abajo en direcciones consecutivas de memoria. Así, una matriz M×N se almacena como si fuera un vector de M·N componentes en donde los m primeros elementos se corresponden a la primera fila de la matriz, los m segundos a la segunda fila y así sucesivamente. Por ejemplo, la siguiente matriz 3×4 se almacena en memoria como un vector de 12 componentes.

0	1	2	3
4	5	6	7
8	9	10	11

```
m: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

El siguiente fragmento de ensamblador lee el elemento **`m[1][1]`** de la matriz.

```
la s1, m        //en s1 tenemos la dirección base
lw a0, 20(s1)   //s1 + 20 nos da la dirección efectiva
                //a0 ahora contiene un 5
```

¿Por qué se suma 20 a la dirección de comienzo de la matriz? Debemos tener en cuenta que cada elemento de la matriz ocupa 4 bytes y que esta se almacena por filas. El primer elemento, **`m[0][0]`**, está en su dirección base. El siguiente, **`m[0][1]`**, se sitúa 4 bytes más adelante. El elemento **`m[0][2]`**, 8 bytes después del **`m[0][0]`** y así sucesivamente para todos los elementos de la primera fila. La segunda fila se ubica a continuación de la primera, así el elemento **`m[1][0]`** se encuentra una fila completa (4·4 bytes) detrás de **`m[0][0]`**, mientras que **`m[1][1]`** 4 bytes detrás del primer elemento de esa segunda fila, es decir 4·4+4 = (4+1)·4 = 20 bytes más adelante del elemento **`m[0][0]`**. En general, la dirección efectiva del elemento **`m[i][j]`** de una matriz M×N se calcula como:

$$dir(m[i][j]) = dir(m) + (i*N + j)*4$$

2. Variables locales vs. variables globales

Cuando en C se declara una variable fuera del ámbito de toda función, se denomina global y puede ser leída y escrita por cualquier función del programa. Puede usarse para pasar datos entre funciones sin ningún tipo de restricción. En ensamblador, cada una de estas variables tiene asignada una dirección fija de memoria en la sección **`.data`** o **`.bss`** que es conocida y accesible mediante una etiqueta por cualquier instrucción del programa.

Código de alto nivel

```
#define N 5
int max = 0;
int v[N];

int foo(...) {
  ...
  max = v[0];
  ...
}
```

Esquema de ensamblador

```
.equ N, 5
.data
max: .word 0
.bss
v: .space N*4
.text
foo:
  ...
  la t0, v
  lw t0, 0(t0)          //lee v[0]
  la t1, max
  sw t0, 0(t1)          //escribe max
  ...
  ret
```

Sin embargo, en programación moderna, se aconseja evitar o reducir al mínimo el uso de este tipo de variables para mejorar la claridad del código, facilitar su depuración y aumentar las posibilidades de reuso. En su lugar se opta por usar variables locales y pasar datos entre funciones a través de parámetros.

En C, una variable local se declara dentro de una función y solo puede ser leída y escrita por esa misma función, no por otra. En ensamblador, estas variables se ubican en pila dentro del marco de la función. El espacio que ocupan en pila se reserva explícitamente durante la ejecución del prólogo y se libera en el epílogo. Su dirección es distinta en cada activación de la función y, por tanto, solo conocida y accesible por las instrucciones que la forman.

Código de alto nivel

```
#define N 5

int foo(...) {
  int max = 0;
  int m[N];
  ...
  max = v[0];
  ...
}
```

Esquema de ensamblador

```
.equ N, 5

.text
foo:
  add sp, sp, -(...)    //guarda registros
  ...
  sw ..., 0(sp)
  add sp, sp, -(1+N)*4  //reserva espacio
  ...
  lw t0, 4(sp)          //lee v[0]
  sw t0, 0(sp)          //escribe max
  ...
  add sp, sp, (1+N)*4   //libera espacio
  lw ..., 0(sp)         //restaura registros
  ...
  add sp, sp, (...)
  ret
```

Como puede verse, el prólogo de la función, tras guardar en pila los registros salvados que use, reserva el espacio requerido por las variables locales. En el cuerpo de la función estas variables son accedidas usando como registro base el **sp** con el desplazamiento que corresponda según la ubicación relativa de la variable en pila. En esquemas de gestión más complejos se usa el **fp** en lugar de **sp**. Por último, antes de volver a la función invocante, el epílogo de la función, primero libera el espacio ocupado por las variables locales y a continuación restaura los registros salvados.

4. Desarrollo de la práctica

Para empezar, desarrolla una función en ensamblador del RISC-V que copie una matriz cuadrada en otra según el siguiente algoritmo.

```
void matrixCopy(int n, int x[n][n], int z[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            z[i][j] = x[i][j];
}
```

Seguidamente desarrolla una función en ensamblador del RISC-V que multiplique dos matrices cuadradas de la misma dimensión según el siguiente algoritmo. En esta función usa la instrucción **mul** para multiplicar valores.

```
void matrixMul(int n, int x[n][n], int y[n][n], int z[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            z[i][j] = 0;
            for (int k = 0; k < n; k++)
                z[i][j] = z[i][j] + x[i][k] * y[k][j];
        }
}
```

Por último, desarrolla una función en ensamblador del RISC-V que calcule la potencia de una matriz cuadrada según el siguiente algoritmo. El espacio para la matriz auxiliar **aux** deberá crearse en pila por encima de los registros salvados y deberá eliminarse antes de restaurarlos. No es necesario hacer uso del registro **fp**, basta con usar **sp** como registro base de **aux**.

```
void matrixPow(int n, int x[n][n], int e, int z[n][n]) {
    int aux[n][n];

    for (int j=0; j<n; j++)
       for (int k=0; k<n; k++)
           if (j==k)
               z[j][k] = 1;
           else
               z[j][k] = 0;
    for (int i = 1; i <= e; i++) {
        matrixMul(n, x, z, aux);
        matrixCopy(n, aux, z);
    }
}
```

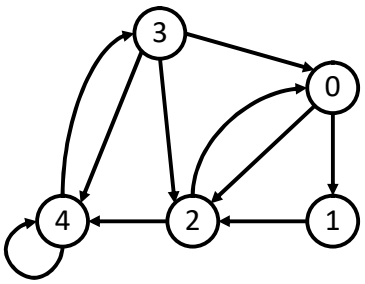
Para probar el correcto funcionamiento de las anteriores funciones usa el siguiente programa en C que, dado un grafo dirigido representado mediante su matriz de adyacencia, calcula para un par de nodos el número de caminos diferentes que los conectan atravesando un número de arcos determinado.

```
#define N      5      //número de nodos del grafo
#define STEPS  3      //número de arcos del camino
#define ORG    0      //nodo origen
#define DST    3      //nodo destino

extern void matrixPow(int n, int [n][n], int, int [n][n]);

int graph[N][N] =    //matriz de adyacencia del grafo
{
    {0, 1, 1, 0, 0},
    {0, 0, 1, 0, 0},
    {1, 0, 0, 0, 1},
    {1, 0, 1, 0, 1},
    {0, 0, 0, 1, 1}
};
int z[N][N];         //matriz resultado del algoritmo
int paths;

void main() {

    matrixPow(N, graph, STEPS, z);
    paths = z[ORG][DST];

   while(1);
}
```

En este programa, la matriz de adyacencia **`graph`** representa al siguiente grafo dirigido:

nodo destino

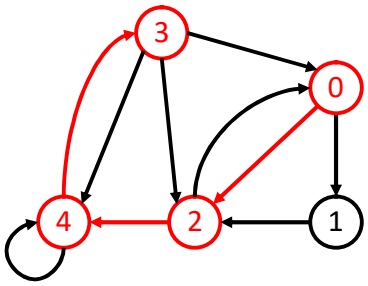
nodo origen	**0**	**1**	**2**	**3**	**4**
0	0	1	1	0	0
1	0	0	1	0	0
2	1	0	0	0	1
3	1	0	1	0	1
4	0	0	0	1	1



Su ejecución con los datos indicados hace que la variable **`paths`** tome el valor 1 ya que entre el nodo 0 y el 3 existe un único camino que atraviesa exactamente por 3 arcos. La matriz auxiliar **`z`** resultante (que, para cada par de nodos, calcula el número de caminos diferentes con 3 etapas que los conecta) es la siguiente.

nodo destino

nodo origen	**0**	**1**	**2**	**3**	**4**
0	1	1	1	**1**	2
1	0	1	1	1	1
2	2	0	2	1	3
3	2	1	3	2	4
4	2	1	2	2	4



No obstante, se recomienda que, para simplificar la depuración de las funciones en ensamblador, modifiques el programa C facilitado y pruebes individualmente cada una de ellas con distintas matrices ejemplo.