Matrices en Ensamblador MIPS

Bryan Manzano y Luis Villamar

I. Abstract

Las matrices son importantes en la actualidad ya que son representaciones de datos, como lo puede ser un árbol en programación, entre otras estructuras. Eso genera que conocer como son concebidas computacionalmente ayude a profundizar en como operan las memorias de almacenamiento de datos. Entonces, el documento consta de una introducción a las matrices. Además de contener un ejemplo claro de como implementar la suma matricial en MIPS, esto con la ayuda de una formas de representar arreglos en memoria.

II. Introducción

En la computación las matrices juegan un papel importante debido a las aplicaciones que otorgan dichas estructuras. Comúnmente, en dichas aplicaciones es necesario hacer operaciones entre las mismas matrices tal como suma, multiplicación por un escalar, suma de la diagonal, multiplicaciones, etc.

Por otra parte, para la mayoría de programadores todo esto se hace en lenguajes de alto nivel en donde deprendiendo del lenguaje puede ser más o menos complejo, teniendo eso en mente surgen las preguntas: ¿cómo sería implementar una matriz en lenguaje ensamblador? y ¿cómo hacer las respectivas operaciones de esas estructuras? Esas dos preguntas serán las que este documento intenta contestar mediante un enfoque práctico, en el cual se irá detallando como en código ensamblador se interactúa a nivel algorítmico con las matrices.

Para los fines de este documente se asumirá que el lector tiene conocimientos del conjunto de instrucciones MIPS y además cuenta con nociones básicas de cómo es la estructura de una matriz matemáticamente con sus operaciones respectivas.

El presente documento tiene como objetivo que el lector adquiera el conocimiento de como se representan las matrices en memoria y su respectivo tratamiento en lenguaje ensamblador, específicamente en MIPS.

En relación al contenido se iniciaría dando una breve descripción de cómo es una matriz y sus componentes. Luego, se procederá a detallar como las matrices son concebidas en memoria y como acceder a cada uno de los elementos de las mismas. Para después mostrar en código ensamblador como se realiza la operación suma de matrices. Para finalizar se emitirán observaciones generales a tomar en cuenta con respecto a la investigación en general.

III. ¿Qué es una matriz?

Una matriz es una representación de números reales o complejos ordenados en una estructura de m por n, donde m es el número de filas y n el número de columnas.

Como cualquier objeto matemático, una matriz puede ser operable dentro del espacio matemático definido mediante suma, multiplicación, etc.

Normalmente se usan las matrices de varias dimensiones para representar transformaciones lineales de vectores a espacio.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{array}$$

Figura 1. Representación de una matriz

En la figura 1 se apreciar que cada elemento que conforma la matriz se representa por la letra minúscula de la matriz y los subíndices i y j.

Un ejemplo sería a_{11} , el cual haría referencia al elemento de la primera fila y primera columna.

IV. Formas de una matriz

Las matrices al ser estructuras que contienen números ordenados en filas y columnas, poseen diferentes presentaciones.

Están las matrices fila, que solo poseen 1 fila y n columnas.

$$\begin{pmatrix} a & b & c \end{pmatrix}$$

Las matrices columna, las cuales cuentan con una columna y n filas.

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

Las matrices nulas, las cuales tienen todos sus elementos con ceros.

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Matrices diagonales, las cuales solo tienen un valor distinto de cero en su diagonal.

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$$

Matrices escalares, estas son matrices que tienen el mismo valor en toda su diagonal.

$$\begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$$

Matrices unitarias, son matrices diagonales con la particularidad que su diagonal son números uno.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matriz triangular superior, estas son matrices cuya esquina derecha superior hasta la diagonal poseen números diferentes de cero.

$$\begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix}$$

Matriz triangular inferior, son matrices opuestas a la matriz triangular superior sin ninguna otra particularidad.

$$\begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix}$$

Para para efectos de este documento nos enfocaremos en realizar operaciones matemáticas en Ensamblador usando el conjunto de instrucciones MIPS con matrices 2×2.

V. Formas de representación de una matriz en memoria

La memoria es fundamentalmente una entidad simple y lineal de espacios, gracias a esto es sencillo representar un arreglo de números o datos en la memoria, pero ¿qué pasa si se requiere usar arreglos multidimensionales como las matrices?

Una solución que surgió fue representar la estructura multidimensional como un conjunto de arreglos separados dependiendo a las necesidades, lo cual permite representar las matrices.

Para implementar matrices en lenguaje Ensamblador usando el conjunto de instrucciones MIPS se tiene dos formas principales de implementarlo. Es así que esta la forma de fila mayor y columna mayor. A continuación, se explica las dos formas de modelar dichas formas.

V-A. Fila Mayor

Esta forma de implementar se basa en la idea de que todas las filas, se guardan secuencialmente en memoria. Es decir, la primera fila se guarda ordenadamente en memoria y la siguiente fila se escoge el primer elemento de ella y se la dispone en el siguiente espacio en la memoria, siguiendo este ciclo hasta que toda la información de la matriz este en memoria.

Para poder recuperar valores de una matriz 2×2 se considera la siguiente ecuación:

$$\begin{aligned} \text{direccionMemoria} = \text{direccionBase} + \\ (\text{indiceFila} \times \text{numeroColumnas} + \\ \text{indiceColumna}) \times \text{tamañoDato} \end{aligned} \quad (1)$$

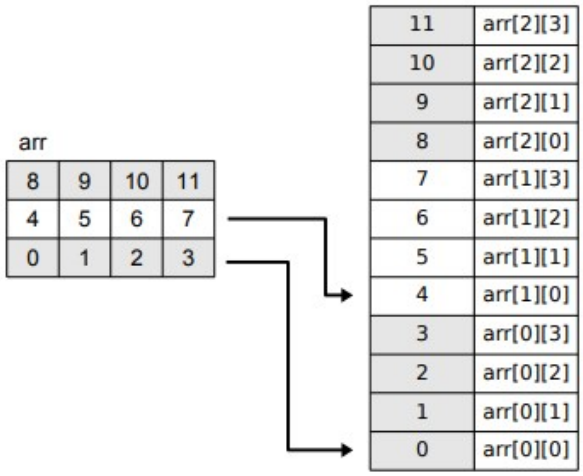

Figura 2. Fila Mayor

Donde direccionBase es la dirección inicial de la matriz, por tanto, en el ejemplo de arriba seria 0 en $arr[0][0]$. Por otra indiceFila es el índice de la fila que se desea acceder, en el caso del acceder al 0 seria índice 0 y el de 8 índice 2. Luego numeroColumnas es la cantidad total de columnas que contiene la matriz e indiceColumna es el índice de la columna del elemento que se quiere acceder, para el caso del elemento 0 seria índice 0 y para el elemento 11 seria índice 2.

Finalmente, tamañoDato es el tamaño en bytes que ocupa el dato que estamos guardando y es el cual permite ir escalando de 4 bits en 4 bits desde la dirección base del arreglo.

V-B. Columna Mayor

Esta forma de implementar se fundamenta en que las columnas son guardadas de forma sucesiva. En otras palabras, la primera columna se guarda secuencialmente una tras otra. Permitiendo tener acceso a todos los elementos de la matriz orientados por columna.

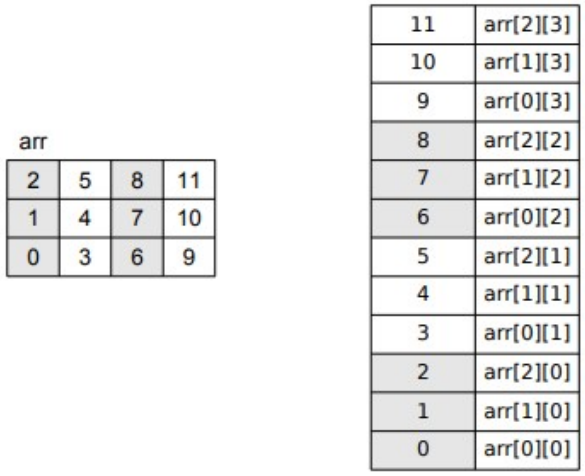

Figura 3. Columna Mayor

Al igual que la anterior manera de modelar la matriz, también se puede recuperar los datos mediante el uso de la ecuación.

$$\begin{aligned} \text{direccionMemoria} = \text{direccionBase} + \\ (\text{indiceColumna} \times \text{numeroFilas} + \\ \text{indiceFila}) \times \text{tamañoDato} \end{aligned} \quad (2)$$

Como podemos observar es parecida a la anterior ecuación que representa el formato fila mayor.

Donde direccionBase es la dirección inicial de la matriz. Siguiendo la ecuación tenemos a indiceColumna, el cual es el índice de la columna a la que se desea acceder, siendo 0 para la primera columna y $n-1$ para la última. Por otra parte, numeroFilas es el número total de filas de nuestra matriz e indiceFila representa el índice de la fila a la se accederá, siendo 0 la primera fila y m-1 la última fila Finalmente, tamañoDato es el tamaño en bytes que ocupa el dato que estamos guardando y es el cual permite ir escalando de 4 bits en 4 bits desde la dirección base del arreglo.

VI. Implementación en MIPS

En la siguiente sección de código se procederá a explicar cómo es el proceso de implementación.

```
.data

  matrizA:  .word 2, 3
        .word 1, 6
  matrizB:  .word 2, 6
          .word 7, 3
  nroColumnas:  .word 2

  .eqv longitudDato 4

  espacio: .asciiz " "
  salto: .asciiz "\n"

.text
  principal:
    la $s0, matrizA
    la $s1, matrizB
    lw $s7, nroColumnas

    add $t0, $zero, 0
      bucle1:
        mul $t1, $t0, $s7

      add $t6, $zero, 0
        bucle2:
          addi $t2, $t1, 0
          add $t2, $t2, $t6
          mul $t2, $t2, longitudDato

          add $t3, $t2, $s0
        add $t4, $t2, $s1

        lw $s3, ($t3)
        lw $s4, ($t4)
        add $t3, $s3, $s4

          move $a0, $t3
        li $v0, 1
          syscall

        li $v0, 4
        la $a0, espacio
        syscall

          addi $t6, $t6, 1
          blt $t6, $s7, bucle2

        li $v0, 4
        la $a0, salto
        syscall

        addi $t0, $t0, 1
        blt $t0, $s7, bucle1
```

En la sección .data se tiene:

- Las directivas matrizA y matrizB están definidas de tal forma que cada .word es una fila y cada elemento representa su respectiva columna.
- La directiva nroColunmas permite obtener un número que representa el número de columnas y la directiva longitudDato el tamaño que ocupa un entero, el cual es 4.
- Las directivas espacio y salto son para cuestiones de imprimir la matriz resultante.

En la sección .text se tiene:

- Todo se dará en la declaración principal, la cual apunta a una determinada dirección de memoria como las demás declaraciones que se darán en el código.
- En $s1 y $s2 se cargará desde memoria las directivas que apuntan a las matrices, así como también $s7 cargara el número de columnas.
- La variable $t0 iniciada en 0 sirve como contador para la declaración bucle1, en la cual dentro se instancia $t1 para simular el producto índice de filas por número de columnas, esto para representar en memoria un salto de fila. Por otra parte, también se usa como contador $s6 iniciado en 0 para la declaración bucle2.
- Ya en bucle2 se copia el valor de $s1 en $t2 y se procede a sumar el resultado con el contador $s6, lo cual simula un salto de columna. Finalmente se multiplica por la constante longitudDato para dar el salto final en la memoria, la misma que es de una dimensión.
- Ya teniendo un offset que representa una posición, lo único que se hace es asignar a $t3 y $t4 la dirección de memoria de las matrices desplazadas dicho offset.
- Teniendo las direcciones de memoria listas y cargadas en los registros $s3 y $s4 se procede a guardar en $t3, la cual contendrá el elemento resultante de la matriz a la cual se desea llegar.
- Finalmente, el resto de código se encarga de seguir los bucles e imprimir y dar formato a la matriz resultante.

VII. Conclusiones

- La operación suma es el ejemplo que brinda al lector la base para hacer otras operaciones, inclusive la multiplicación que es algo más compleja debido a las operaciones necesarias entre columnas y filas.
- Las matrices en tres dimensiones seria todo un reto implementar, sin embargo, hacerlo en lenguaje ensamblador solo sería por cuestiones didácticas.
- El formato fila mayor es más intuitivo para explicar que el de columna mayor, ya que va recorriendo fila a fila los elementos de cada columna.

- Aunque solo se operó una matriz cuadrada, solo es cuestión de añadir una variable que haga el papel de contador que represente el número de filas para que uno de los bucles termine en el contador antes dicho.

[8] [9] [4] [3] [1] [5] [7] [6] [10] [2]

REFERENCIAS

[1] S. Boyd and L. Vandenbergue. *Introduction to applied linear algebra*. Cambridge university press, 2018.

[2] J. Cavanagh. *X86 AssemblyLanguageandC Fundamentals*. CRC Press, 2015.

[3] C. Hamacher and Z. Vranecic. *Computer organitation and embedded systems*. Mc Graw Hill, 2014.

[4] J. Hennessy and D. Patterson. *Computer architecture, A Quantitative Approach*. Morgan Kaufmann, 2019.

[5] Ed Jorgensen. *Assembly Language Programming using QtSpim*. BSD License, 2019.

[6] C. Kann. *Introduction to MIPS Assembly Language Programming*. Gettysburg College, 2016.

[7] Daniel Kusswurm. *Modern x86 Assembly Language Programming*. Apress Media LLC, 2014.

[8] D. Patterson and J. Hennessy. *Computer organizacion and design*. Morgan Kaufmann, 2014.

[9] X. Yang. *Engineering Mathematics with Examples and Applications*. Elsevier, 2017.

[10] D. Yurichev. *Understanding Assembly Language*. Independent, 2019.