Trabajo Práctico Integrador 1

Bartezaghi Catriel y Katzenelson Germán

***Resumen*— En este trabajo práctico integrador se procede a resolver un sistema de ecuaciones de la forma $A*x=b$ donde se debe verificar que A sea orden $n*n$ y b orden n. Luego de esta verificación se procede a su resolución implementada en Assembler.**

I. Introducción

El ejercicio en cuestión plantea la problematica de programar un código en Assembler que sea capaz de resolver sistemas de ecuaciones de orden 2 hasta 10. El límite superior de 10 se debe a que el data segment tiene una capacidad máxima de 128 espacios de memoria, en caso de ser un órden mayor a 10, excedería la capacidad de almacenamiento del segmento de datos.

El primer desafío que se nos presentó es que método utilizar para resolver el sistema de ecuaciones, tópico que se abordará en la siguiente sección.

II. Método de solución

A lo largo de la carrera hemos incurrido en numerosos métodos para la solución de un sistema de ecuaciones, muchos de ellos están atados a ciertas propiedades de la matriz por lo que decidimos alejarnos de ellos para no complejizar la resolución.

Finalmente se optó por utilizar el método de **Gauss-Jordan**, para ello, desarrollamos el algoritmo en c++, siendo este el lenguaje más utilizado en los primeros años de la carrera y con el cual nos encontramos muy familiarizados, para luego sí, proceder a la codificación en Assembler.

Consideremos un sistema de ecuaciones lineales de n incógnitas, expresado de forma matricial $Ax = b$, donde A es la matriz de coeficientes de las ecuaciones, x es un vector columna con las incógnitas y b es el vector columna con los términos independientes de las ecuaciones.

En primer lugar se procede a armar la matriz ampliada del sistema, la cual consiste en listar los coeficientes de A y a su derecha el vector columna b, de esta manera las operaciones efectuadas en los renglones se realizan tanto en la matriz como en el vector. Luego, mediante operaciones en las filas, vamos transformando la matriz ampliada en una matriz diagonal superior e inferior, esto es cuando todos los elementos que no pertenecen a la diagonal son nulos. Para realizar esto, primero dividimos cada coeficiente que se encuentra en la misma columna que un pivote por el valor de este pivote y obtendremos un valor que llamaremos ratio. Por ejemplo, el cálculo del ratio para cada elemento j de la columna i, estará dado por:

```
ratio = A[j][i] / A[i][i]
```

Después efectuamos las operaciones sobre cada elemento de la fila j, restándole el valor de la multiplicación entre, el ratio calculado y el valor del coeficiente que se encuentra en su misma columna y que corresponde a la misma fila que el pivote:

```
A[j][k] = A[j][k] - ratio * A[i][k]
```

Una vez que obtuvimos la matriz diagonal, después de aplicar el proceso en todos los elementos de la matriz, el último paso es hallar la solución. Esto se hace dividiendo cada renglón por el valor del pivote. De esta manera, en la última columna quedan listadas las soluciones del sistema de ecuaciones.

Nota: Algo importante a tener en cuenta, es que durante el proceso se debe evitar que los pivotes tomen el valor 0, ya que al momento de calcular el ratio, estaríamos haciendo una división por 0. En el cálculo manual, esto se puede solventar haciendo un intercambio de renglones. En cuanto al desarrollo de un algoritmo informático, hemos visto en otras asignaturas que esto se puede resolver mediante una estrategia de pivoteo. En nuestro código, dada la complejidad de implementación que tiene el lenguaje assembler, y las limitaciones de memoria, solo nos limitamos a comprobar en cada cambio de pivote que este no sea nulo antes de continuar con la operación.

III. Manejo de matriz

Para la declaración de la misma se decidió declarar la matriz en la directiva $.data$ utilizando $.float$, de esta forma evitaremos los problemas que acarrea la división de dos números.

Se tomó la decisión de declarar la matriz aumentada $[Ab]$, donde a lo largo del texto se la nombrará simplemente como A, de presunto orden $n, n+1$, para simplificar operaciones y hacer un desarrollo más eficiente del algoritmo, la misma es declarada como un vector dentro del programa.

Comenzamos guardando la dirección inicial de memoria de la matriz A en el registro $a0$. Luego se procede a declarar el tamaño del dato en $t2$, en este caso al usar float será de 4, y el orden de la matriz A original (n+1).

Para acceder a una celda (i, j) de la misma, se realizan los siguientes pasos explicados a modo de ejemplo en un pseudocodigo assembler:

```
mul t1, i, n+1
```

En este paso se guarda en el registro $t1$ el producto entre la fila i que queremos acceder y n+1, de esta manera se logra realizar el salto a la fila deseada.

```
add t1, t1, j
```

Se agrega la columna j a los datos previamente calculados, en este paso tenemos numericamente cuantos pasos deberíamos hacer en el vector para llegar a nuestra celda.

```
mul t1, t1, t2
```

Se multiplica el paso numérico por el tamaño del dato almacenado, por lo ya tenemos en t1 el avance en hexadecimal que hay que realizar en la matriz.

```
add a1, t1, a0
```

Finalmente se agrega la dirección hexadecimal del comienzo de la matriz a el cálculo previo. De esta forma en $a1$ tenemos la dirección de memoria de nuestro elemento i, j de la matriz, ahora solo resta cargarlo el dato en un registro para poder operar con él:

`lw t1, (a1)`

Algo importante y que se debe tener en cuenta para operar la matriz es que tanto i, como j, deben comenzar con indice 0, es decir, si quiero acceder a la celda de la primer fila y la primer columna, tanto i como j deben ser 0

IV. Verificación de orden

Para verificar que haya concordancia en el sistema que se desea resolver, es decir que A, sea de orden $n, n+1$, la cantidad de datos almacenados debe ser de $n*n+n$. Con esto en mente, se declaró un espacio del data segment, llamado X, acontinuación de A, por lo que al cargar la dirección de X, sabremos el límite de A.

Se realiza el cálculo de $(n*n+n)*4$, el cual, nos da la diferencia que en teoría debería tener A de X en el data segment. Procedemos a sumarle la operación antes mencionada a la dirección de A, que almacenamos en $a0$, en $a2$ y hacemos un condicional, donde verificamos que, para continuar con la ejecución del algoritmo, la dirección de A y de X (guardada en $a1$) deben coincidir, de no ser así, se realiza un $jump$ a error y finaliza su ejecución.

```
verificarMatriz:
  mul a2, t6, t6 # (n * n)
  add a2, a2, t6 # (n * n) + n
  mul a2, a2, a3 # ((n * n) + n ) * 4
  add a2, a2, a0 # Sumo a la posición

  # a2 (calculo) = a1 (X)
  bne a2, a1, error
```

V. Resolución

Para explicar la resolución del enunciado decidimos enumerar las lineas de código adjuntas en la última hoja y agruparlas según su funcionamiento que se detalla a continuación:

A. Linea 1 - 7

Se realiza la declaración de la matriz en forma de vector, se reserva el espacio para la solución, y se guardan dos directivas $.asciz$, una es un espacio para mostrar los datos y la otra es un mensaje de error.

B. Linea 8 - 20

Se realizan todas las inicializaciones de registros iniciales, comenznado por n, asignando las direcciones de $.data$ en los registros e inicializando los contadores i,j,k

C. Linea 21 - 29

Verificación del orden de la matriz explicada anteriormente.

D. Linea 30 - 48

Inicialmente se verifica que el contador es menor a n, si lo es procede a verificar si el elemento $a[i][i] = 0$, si lo es, muestra un error, de lo contrario continúa con su ejecución.

E. Linea 49 - 77

Se comprueba que puede seguir iterando y también que no se encuentre en un elemento de la diagonal principal, en este caso, vuelve a iterar. Luego se hacen los cálculos para obtener el *ratio*, el cual resulta de hacer $a[j][i]/a[i][i]$

F. Linea 78 - 109

Al igual que en los ciclos anteriores, se comprueba al inicio que no excedio el limite de iteraciones, y se realiza en cálculo de $a[j][k] = a[j][k] - ratio * a[i][k]$.

Al finalizar todas estas iteraciones, lo que obtendemos finalmente es una matriz diagonal declarada dentro de lo que inicialmente era A.

G. Linea 120 - 170

Inicialmente se reescribe la dirección del comienzo de A y se realiza un ciclo final de 0 hasta n donde cáda elemento de nuestra solución está dado por $x[i] = a[i][n+1]/a[i][i]$, finalmente luego de obtener cada $x[i]$ se procede a mostrar el elemento en el box *messages* guardando en a7 el tipo de dato a mostrar (2 para float y 4 para asciz), en $ft0$ o $a0$ el dato y finalmente un llamado $ecall$.

VI. Pruebas o rendimiento

A la hora de realizar las pruebas decidimos analizar el registro pc donde se cuentan las cantidad de operaciones que realiza el programa, primero se comenzó con matrices de orden 2 hasta llegar al orden 10 registrando cada uno de los valores en números decimales.

Orden	pc	Decimal
2	0x004001a0	416
3	0x0040019c	412
4	0x004001a0	416
5	0x0040019c	412
6	0x0040019c	412
7	0x0040019c	412
8	0x0040019c	412
9	0x0040019c	412
10	0x0040019c	412

Cuando uno observa los valores que toma pc puede notar que no existe una relación y que puede estár sucediendo un problema, al revisar paso por paso el programa, en ciertos momentos el contador realiza pasos inexplicables y que atribuimos a un problema, o del programa, o del entorno donde corre.

Por lo que se decidió cronometrar a mano limitando, redondeando los resultados, la cantidad de operaciones configurando al RARS para que corra a una velocidad de 30 por segundo para poder bosquejar una idea del rendimiento de nuestro programa, estos fueron los resultados:

Orden	Tiempo (s)	Operaciones
2	9	270
3	23	690
4	50	1500
5	93	2790
6	154	4620
7	238	7140

Por cuestiones de tiempo, el conteo manual, se realizó únicamente hasta matrices de orden 7 donde se puede observar una tendecia exponencial en la cantidad de operaciones realizadas, y en base a los siguientes gráficos predecir el comportamiento con diferentes valores.

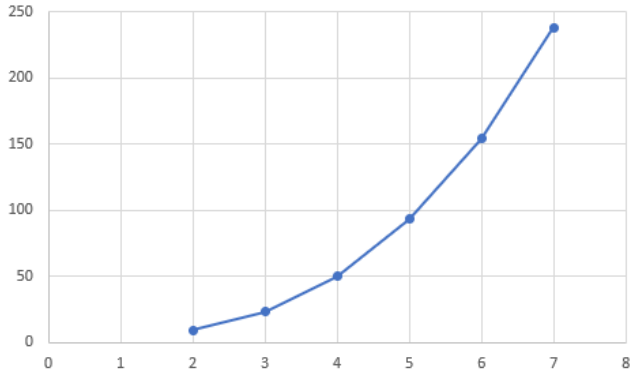

Fig. 1. Orden vs Tiempo

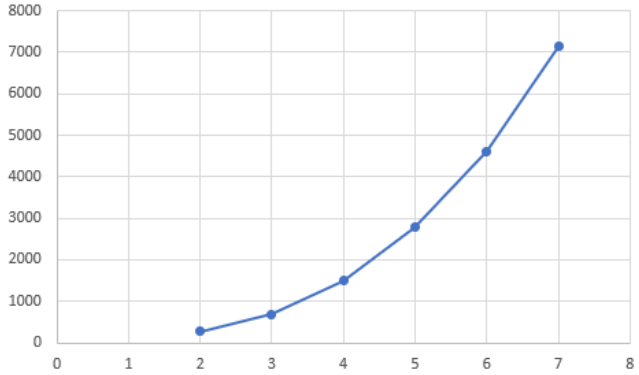

Fig. 2. Orden vs Operaciones

VII. Conclusión

El trabajo práctico nos permitió comprender con mayor profundidad la forma en que opera el procesador y los manejos de memoria en la implementación de un algoritmo, nos dio una visión de cómo se implementan las distintas estructura de control universales, como son los loop anidados, while, entre otros, en un lenguaje de bajo nivel y la complejidad de los mismos.

Algo importante, es que también tuvimos la experiencia de tener que modelar estructuras más avanzadas, como lo es una matriz, y desarrollar algoritmos complejos partiendo de las instrucciones y herramientas básicas que nos provee el lenguaje ensamblador correspondiente a la arquitectura RISCV.