Universidad de Santiago de Chile

Facultad de Ingeniería

Departamento de Ingeniería Informática

Laboratorio 3: Acercándose al Hardware: Programación en Lenguaje Ensamblador

Alumno: Ignacio Villarroel E.

Asignatura: Organización de computadores

Sección Laboratorio: L – 3

Profesor: Carlos González C.

Ayudante: Ricardo Ruz

Fecha de entrega: 07/01/2022

Tabla de contenidos

1. Introducción

Al igual que muchos lenguajes de programación, Mips tiene una gran variedad en las formas de como interpretar y escribir un código, sin embargo, existen diversas técnicas que permiten la facilitación en la implementación de instrucciones, por ejemplo el uso de una recursión o la "memoización". Debido a que Mips tiene un desarrollo de instrucciones lineal, es decir, va de corrido hacia abajo, la utilización de estas técnicas tiene un desarrollo mas complejo, es por eso que se pretende estudiar y aplicar las implementaciones antes dichas. Estas se utilizarán para el desarrollo de problemáticas matemáticas.

1.1 Objetivos

Para el desarrollo óptimo de la actividad, se plantean los siguientes objetivos a realizarse

- Utilizar Mips en desarrollos aritméticos, de salto y memoria para la implementación en códigos.
- Comprender el uso de subrutinas, con el manejo de stacks.
- Utilización de syscall para llamadas necesarias.
- Implementación de algoritmos para resolver problemas matemáticos.

2. Marco teórico

En el ámbito de mips, está el uso de registros, debido a que toda acción realizada, hace referencia al uso de los datos contenidos en estos, sin embargo, existen distintas formas en estos datos se almacenan, esto por direcciones de memorias (punteros), que permiten la generación de stacks, esto permite que los datos que se van generando se apilen en un estilo de arreglo (lista), esto permite que se acceda mas rápido a ellos cuando se requiera, a este proceso de la conoce como memoización.

La recursión está ligado de manera muy cercana al uso de subrutinas, sin embargo esta en vez de ir a una etiqueta (label) distinta en el código, esta se redirige a sí mismo, creando un loop hasta que se cumpla algo que permita salir de ese estado.

El uso de syscall es muy diverso, las opciones más recurrentes van desde mostrar por pantalla un mensaje y solicitar un número, hasta finalizar el programa.

3. Desarrollo y resultados

En primer lugar se destaca que todos los códigos fueron realizados exitosamente, sin embargo la parte 1, correspondiente al desarrollo del cálculo de coseno, seno hiperbólico y a continuación se para a detallar el desarrollo de cada uno de ellos.

3.1 Implementación Recursiva -Fibonacci (parte 1)

Para el desarrollo de esta sección, se implementaron stacks, mediante los cuales eran podían ser guardados y luego obtenidos de la memoria por punteros, donde básicamente los elementos guardados en el stack se van apilando a medida que ocurren las recursiones (llamadas a subrutina "*fibonacci*"), con esta información y cada vez mas reducido las cantidades, se van sumando progresivamente, hasta igualar la cantidad del contador con el término solicitado de la sucesión. Finalmente se cierra el stack, dejándolo tal como se tomó, evitando cualquier posible error. Existen excepciones para cuando el término es cero o uno.

En el uso práctico del código se llegó a probar hasta la posición 25°, donde no cálculo, dando a entender que el programa estaba colapsado, debido a las múltiples direcciones de memoria a la que debe buscar en los stacks.

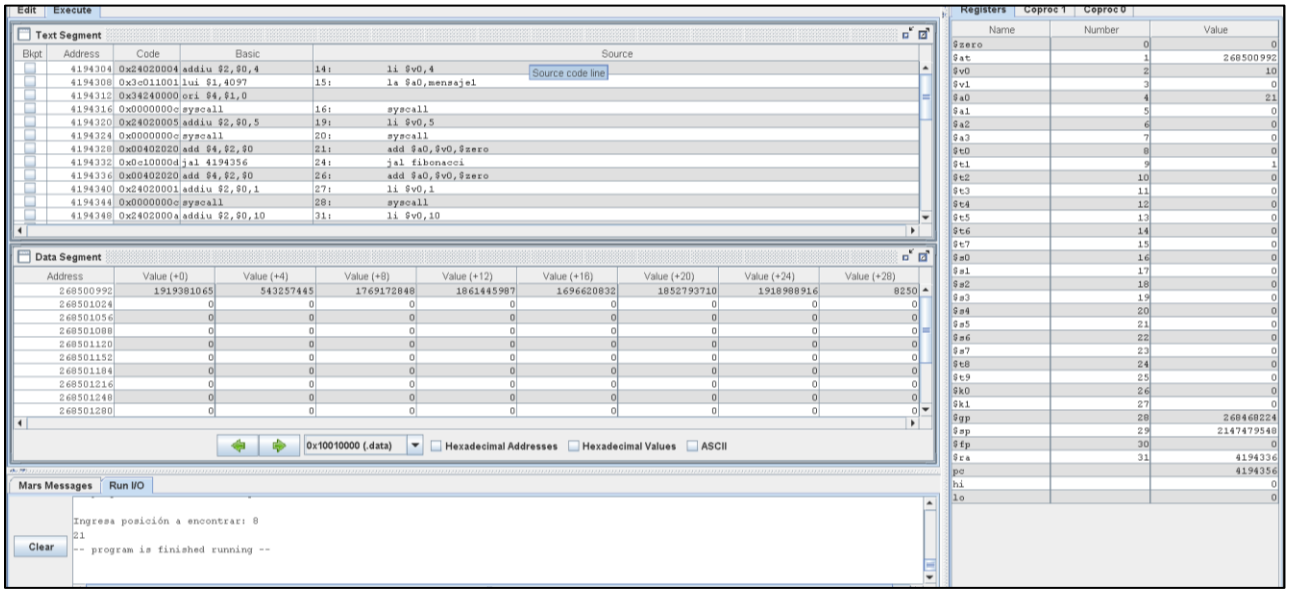

Figura 1. Cálculo de 8° término por recursión

3.2 Implementación Memoizacion -Fibonacci (parte 2)

La utilización de bufer fue efectiva y a su vez la implementación de por medio la memoización hizo mas óptima la ejecución de la obtención del enésimo término con respecto a la implementación recursiva. El enésimo término a encontrar estará ubicado en la parte de ".data" y a su vez se debe generar un espacio (arreglo) en la memoria para la resolución de la búsqueda del término, esto permite la generación de un algoritmo mas dinámico, luego con la generación de contadores y una subrutina llamada "memoria" que será encargada de registrar los números hasta que se complete el contador. A partir de lo anterior, se genera el proceso de memoización que a partir de del arreglo creado se comienzan a trabajar la formación de términos hasta que el contador llegue hasta el límite deseado. Se hicieron diferencias en el cálculo del Fibonacci respecto a la parte 1, sin embargo, se mantuvo la base en la recursión. Finalmente se comprueba si efectivamente el contador llegó a su limite, terminando el código y saliendo exitosamente de este.

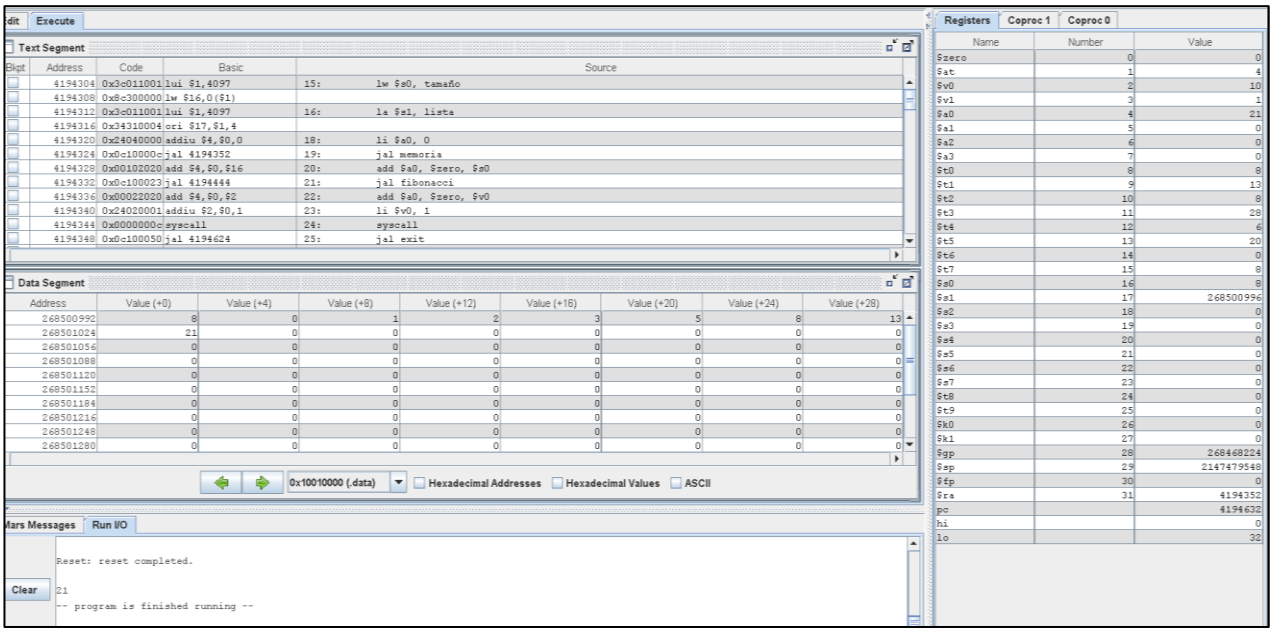

Figura 2. Cálculo de 8° término por memoizacion

3.3 Rendimiento

A la hora de ver manualmente hasta que término pueden encontrar los códigos, se ve que el límite de cálculo para cada algoritmo es:

Fibonacci recursivo	Hasta el término n° 25.
Fibonacci por memoizacion	No tiene límite.

Viendo esto, se nota que la diferencia a la hora de encontrar los números en la memoria no es tan eficiente, por lo que la creación de un arreglo emula a lo que pasa dentro de una caché que existe una cohesión entre los términos que se van creando por lo que al momento de buscar en la memoria, la caché hace todo el trabajo optimizando el proceso. Otra ventaja es la implementación de stacks para la retención para la búsqueda por direcciones de memoria.

4. Conclusiones

Los objetivos se cumplieron en su totalidad puesto que la parte 1 se completó efectivamente, creando un método recursivo para el cálculo del enésimo término, también la parte 2 que efectua lo mismo que la parte 1 pero con un algoritmo creado en base a memorización sin embargo en la creación de los códigos de las parte 2, , se dominó el uso de subrutinas como de stacks para retener información temporalmente por direcciones de memoria, así como la creación de una lista para optimización en la búsqueda de datos.

Se corroboró con plenitud las diferencias entre el uso de recursión y el uso de la memoización, viendo la cantidad de término que a los que pueden calcular, se ve que el código donde se implementó la memoización puede calcular con mayor rapidez, esto se debe a que los elementos al estar en un arreglo dinámico, su acceso es más sencillo a la hora de utilización de datos, por lo que el computador no debe recorrer largas distancias entre direcciones de memoria, por otro lado, la parte recursiva utiliza stacks por punteros, esto hace que los datos están distribuidos en distintos puntos de la memoria, por lo que este debe recorrer mayores cantidades de memoria para obtener los términos guardados, debido a la realización de esto, no se pudo comprobar.

Pese a que las realización del código fue de gran éxito, eso no implica a que las implementaciones fueran las mejores, puesto que al estar recién conociendo el lenguaje ensamblador, alguno de los códigos puede tener creaciones innecesarias, que solo crean demoras dentro de este mismo haciendo menos eficiente el código creado.

5. Referencia

- Castillo Catalán, M., & Claver Iborra, J. M. (2004). *Prácticas guiadas para el Ensamblador del MIPS R2000* (1.ª ed., Vol. 1). https://www3.uji.es/~ochera/curso_2004_2005/ig09/PractR2000_antiguo.pdf
- *Recursión de MIPS: cómo calcular el resultado final de la pila: ensamblaje, recursión, mips*. (2018). living-sun. https://living-sun.com/es/assembly/85314-mips-recursion-how-to-compute-end-result-from-stack-assembly-recursion-mips.html