Organización y Arquitectura de las Computadoras

Práctica 06: Lenguaje Ensamblador

Bonilla Reyes Dafne
319089660

Medina Guzmán Sergio
314332428

1. Objetivo

Introducir al alumno a la programación en lenguaje ensamblador.

2. Procedimiento

Escribe cada uno de los programas de los ejercicios en lenguaje ensamblador de MIPS, cada uno en un archivo de código fuente distinto. Ensambla el código y ejecuta la simulación de cada uno.

3. Ejercicios

1. Escribe un programa que, sin usar la memoria ni la instrucción `mov`, copie el contenido de un registro a otro.

   Para este ejercicio, encontramos dos formas de hacerlo:

   - **Opción 1:** La instrucción `sll` desplaza el contenido de $t0 cero bits a la izquierda, lo que no tiene ningún efecto en el valor del registro. La segunda instrucción `srl` desplaza el contenido de $t1 cero bits a la derecha, lo que tampoco tiene ningún efecto en el valor del registro. Por lo tanto, el contenido de $t1 es igual al contenido de $t0.
   - **Opción 2:** La operación `XOR` realiza un `XOR` del contenido de $t0 con sí mismo, lo que resulta en cero y el valor de $t0 no se modifica. El resultado se almacena en $t1, por lo que $t1 ahora contiene el mismo valor que $t0.

2. Escribe un programa que calcule el máximo común divisor de dos números.

   Para este ejercicio se hizo uso del algoritmo de Euclides. Primero, se imprimen 2 mensajes para recibir 2 enteros. A continuación, se calcula el MCD usando el algoritmo y finalmente se imprime el resultado. Todo viene documentado para un mayor entendimiento de la implementación.

3. Escribe un programa que compare 2 números enteros e imprima un mensaje que nos diga si son iguales o no.

   En este ejercicio se hizo uso principalmente del control de flujo `beq`. Al igual que en el ejercicio 2, recibimos 2 enteros desde la terminal, los guardamos en 2 registros diferentes y a continuación, usando `beq` vemos si son iguales. Dependiendo del resultado, cargamos la etiqueta `equal` o saltamos a la etiqueta que carga el mensaje de `not_equal`. Se imprime el mensaje correspondiente y el programa termina.

4. Escribe un programa que calcule el cociente y el residuo de una división. No puedes utilizar la operación div, el cociente deberá guardarse en el registro $v0 y el residuo en $v1.

5. En el lenguaje de programación de tu elección, escribe un programa que calcule la serie

$$4 \times \sum_{n=0}^{m} \frac{(-1)^n}{2n+1}$$

donde m es el número de iteraciones que ejecutará el programa. **NO es necesario que mandes el código de este ejercicio**, solo manda una screenshot de tu código. Comenta que hace cada línea y que es lo que hace el código en general. Prueba poniendo $m = 10$, después $m = 1000$, $m = 10000$, etc.

6. Empleando el coprocesador y las instrucciones de punto flotante de precisión sencilla, escribe un programa (en MIPS) que calcule la serie mencionada arriba.

4. Preguntas

1. En el ejercicio 4:

   - ¿A qué valor tiende la serie?
     A π
   - ¿Cuántos dıgitos de la constante se pueden calcular con precisión sencilla?, es decir, considerando que existe precisión sencilla y precisión doble, ¿A cuantos dígitos estaría limitado nuestro resultado con precisión sencilla? Justifica tu respuesta.

     La mantisa en precisión sencilla tiene 23 bits, lo que equivale a 7 dígitos decimales de precisión. Por lo tanto, podemos esperar que el resultado calculado con precisión sencilla tenga alrededor de 7 dígitos decimales de precisión.
   - ¿Cuántas iteraciones son necesarias para calcular el mayor número de dígitos?
     Con el código escrito en Python se requieren alrededor de 30,000,000 de iteraciones para calcular 3.1415926, que es pi a una precisión de 7 dígitos (esto lo podemos ver en la captura de pantalla del código que entregamos en Python).

2. Menciona 5 llamadas a sistema (syscall) que puedes usar en MIPS. Menciona su código de instrucción y que es lo que hace.

Syscall			
Servicio	$v0	Argumentos	Resultado
Imprimir un integer	1	$a0 = integer a imprimir	
Imprimir un float	2	$f12 = float a imprimir	
Leer un double	7		$f0 contiene el double leído
Cerrar un archivo	16	$a0 = descriptor de archivo	
Imprimir integer en binario	35	$a0 = integer a imprimir	El valor mostrado es de 32 bits

   Para utilizar los servicios del sistema syscall de la tabla anterior se siguen estos pasos:

   1. Cargar el número de servicio en el registro $v0.
   2. Cargar los valores de los argumentos, si los hay, en $a0, $a1, $a2 o $f12 como se especifica.
   3. Emita la instrucción syscall.
   4. Recuperar los valores devueltos, si los hay, de los registros de resultados como se especifica.

3. ¿Cuáles son los 3 tipos principales de instrucciones? Menciona que comportamiento tiene cada tipo de instrucción.

   Hay tres tipos principales de instrucciones DLX: R-Type, J-Type e I-Type. Todas las instrucciones DLX son de 32 bits y deben ser alineadas en la memoria en un límite de una palabra, sin embargo, cada una tiene una función distinta.[2]

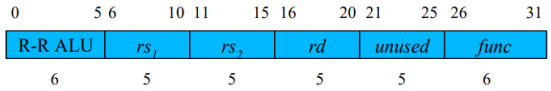

Figura 1: Formato de una instrucción del tipo R.

- R-Type: Manipula datos de uno o dos registros. Se utiliza para operaciones ALU de registro a registro, lectura y escribe desde y hacia registros especiales.
- J-Type: Dispone de las ejecuciones de saltos que no usan un operando de registro para especificar la dirección de destino de la sucursal. Incluye jump (j), jump y link (jal), trap y retorno de excepción (rfe).

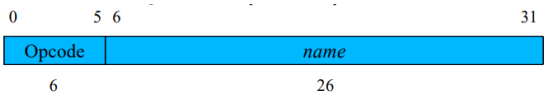

Figura 2: Formato de una instrucción del tipo J.

- I-Type: Manipula los datos proporcionados por un campo de 16 bits. Carga y almacena bytes, palabra. Se encarga de todas las operaciones ALU inmediatas y de todas las instrucciones de bifurcación condicional.

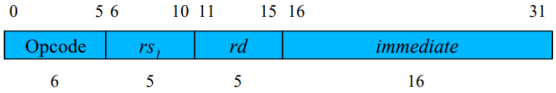

Figura 3: Formato de una instrucción del tipo I.

4. ¿Qué hacen las instrucciones de tipo FR y de tipo FI? Da algunos ejemplos de instrucciones de este tipo y menciona porque están separadas de las otras 3 principales.

   Además de la unidad de punto fijo, la arquitectura DLX también comprende una unidad de punto flotante FPU, que puede manejar números de punto flotante en precisión simple (32 bits) o precisión doble (64 bits).

   El formato FI se utiliza para cargar datos de la memoria en la FPU. Este formato también se usa para bifurcaciones condicionales en el indicador de código de condición FCC de la FPU.

   Por otro lado, el formato FR se utiliza para las restantes instrucciones FPU. Especifica un código de operación principal y uno secundario (código de operación, función), un formato de número Fmt y hasta tres registros de punto flotante (propósito general).[3]

   Además, es importante mencionar que las instrucciones de tipo FR y tipo FI están separadas del resto debido a que las operaciones de punto flotante son más complejas y requieren más recursos que las operaciones aritméticas y lógicas. Por lo tanto, separar estas instrucciones permite que el procesador maneje de manera más eficiente las operaciones de punto flotante y garantiza que se asignen los recursos adecuados a estas operaciones. Además, también simplifica el diseño del procesador y facilita la implementación de las instrucciones.

   Finalmente, los siguientes son ejemplos de este tipo de instrucciones:

   - Instrucciones de tipo FI:

Listing 1: FI-Type

```
1        addi.s $f1, $f2, 3.5
```

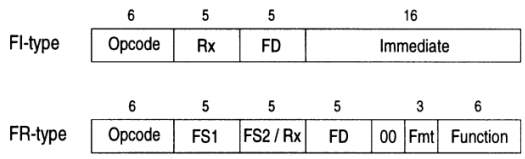

Figura 4: Formato de una instrucción del tipo FI y FR.

Suma el valor inmediato 3.5 al valor almacenado en el registro de punto flotante $f2 y guarda el resultado en el registro de punto flotante $f1.

Listing 2: FI-Type

```
1         sub.s $f1, $f2, 1.5
```

Resta el valor inmediato 1.5 al valor almacenado en el registro de punto flotante $f2 y guarda el resultado en el registro de punto flotante $f1.

- Instrucciones de tipo FR:

Listing 3: FR-Type

```
1         mul.s $f1, $f2, $f3
```

Multiplica los valores almacenados en los registros de punto flotante $f2 y $f3 y guarda el resultado en el registro de punto flotante $f1.

Listing 4: FR-Type

```
1         div.s $f1, $f2, $f3
```

Divide los valores almacenados en los registros de punto flotante $f2 y $f3 y guarda el resultado en el registro de punto flotante $f1.

5. ¿Cuáles son algunos de los desafíos de la optimización del código ensamblador para diferentes arquitecturas y plataformas de hardware?

   A continuación, se enumeran algunos de los principales desafíos en la optimización del código ensamblador:

   1. *Variaciones de las arquitecturas:* Cada arquitectura de procesador puede tener características específicas que afectan el rendimiento y la eficiencia del código ensamblador. Por lo tanto, la optimización del código debe adaptarse a cada arquitectura y aprovechar las características específicas de cada procesador.
   2. *Cambios en la arquitectura del procesador:* Las arquitecturas de procesador están en constante evolución y mejora. Esto significa que la optimización del código ensamblador debe actualizarse regularmente para aprovechar al máximo las nuevas características y mejoras en las arquitecturas de procesador.[4]
   3. *Dificultades de depuración:* El código ensamblador puede ser difícil de depurar, especialmente en plataformas de hardware en las que no se dispone de herramientas de depuración avanzadas. Por lo tanto, la optimización del código debe equilibrar el rendimiento y la eficiencia con la facilidad de depuración.
   4. *Variaciones de la configuración del sistema:* Las diferentes configuraciones del sistema, como la cantidad de memoria RAM disponible, la velocidad del bus del sistema, la frecuencia del reloj del procesador, entre otras, pueden afectar significativamente el rendimiento del código ensamblador.

Por lo tanto, la optimización del código debe tener en cuenta estas variaciones y ajustarse en consecuencia.

5. *Diferentes tipos de memoria:* Las arquitecturas y plataformas de hardware pueden variar en los tipos y velocidades de memoria que utilizan. Por lo tanto, la optimización del código debe tener en cuenta estas diferencias y ajustarse en consecuencia para aprovechar al máximo la memoria disponible.[5]

6. ¿Existe alguna diferencia en escribir programas en lenguaje ensamblador comparado con escribir programas en lenguajes de alto nivel?

   Existen varias diferencias, las principales para nosotros son mencionadas a continuación:

   1. *Portabilidad:* Los lenguajes de alto nivel son más portátiles que el lenguaje ensamblador, ya que los programas escritos en lenguajes de alto nivel se pueden compilar para ejecutarse en diferentes plataformas de hardware sin necesidad de cambiar el código fuente. En cambio, el código ensamblador es específico de una arquitectura de procesador y no es fácilmente portable a otras arquitecturas.
   2. *Velocidad de desarrollo:* Escribir programas en lenguajes de alto nivel suele ser más rápido que escribir programas en lenguaje ensamblador, ya que los lenguajes de alto nivel proporcionan estructuras y funciones predefinidas que permiten a los programadores escribir código rápidamente. En contraste, el lenguaje ensamblador requiere un mayor nivel de detalle y conocimiento de la arquitectura del procesador, lo que puede hacer que el desarrollo de programas sea más lento.
   3. *Nivel de abstracción:* El lenguaje ensamblador es un lenguaje de bajo nivel, lo que significa que proporciona un mayor control sobre el hardware subyacente, pero requiere un mayor nivel de detalle y conocimiento sobre la arquitectura del procesador. En contraste, los lenguajes de alto nivel proporcionan un mayor nivel de abstracción, lo que hace que sea más fácil y rápido escribir programas sin tener que preocuparse por detalles de bajo nivel.
   4. *Capacidad de expresión:* Los lenguajes de alto nivel ofrecen una mayor capacidad de expresión y abstracción, lo que permite a los programadores escribir código más conciso y fácil de leer. Por otro lado, el lenguaje ensamblador puede resultar más difícil de leer y escribir debido a su complejidad sintáctica.

7. ¿Cuáles son algunas de las ventajas y desventajas de usar el lenguaje ensamblador en comparación con los lenguajes de programación de nivel superior? ¿Cómo impactan estas ventajas/desventajas en el proceso de desarrollo, el rendimiento y la capacidad de mantenimiento del software?

   Ventajas:
   - Control preciso del hardware: El lenguaje ensamblador permite controlar directamente los recursos de hardware de la computadora, lo que lo hace ideal para aplicaciones que requieren de un control preciso de los mismos.
   - Eficiencia en el rendimiento: El código ensamblador es altamente eficiente en cuanto al uso de recursos de hardware, lo que se traduce en un mejor rendimiento de la aplicación en comparación con las aplicaciones escritas en lenguajes de programación de nivel superior.
   - Flexibilidad: Al ser un lenguaje de bajo nivel, el ensamblador es altamente flexible y se adapta fácilmente a diferentes arquitecturas de hardware y sistemas operativos.
   Desventajas:
   - Dificultad de aprendizaje: El lenguaje ensamblador es muy complejo y difícil de aprender, lo que limita su adopción y hace que sea necesario contar con programadores altamente capacitados y especializados.
   - Tiempo de desarrollo: Debido a su complejidad, el desarrollo de aplicaciones en ensamblador es un proceso lento y tedioso, lo que puede retrasar los plazos de entrega del software.
   - Mantenimiento: La programación en ensamblador requiere de un mayor esfuerzo de mantenimiento, ya que es necesario tener un conocimiento profundo del hardware y hacer ajustes en el código para adaptarse a diferentes plataformas y versiones de sistema operativo.

Referencias

[1] SYSCALL Functions Available in MARS

[2] The DLX Instruction Set Architecture

[3] DLX Instruction Set Architecture

[4] Ways to Optimize Assembly Code for Speed

[5] Finding the Limits of Hardware Optimization

[6] Difference Between Assembly Language and High-Level Language