DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

ORGANIZACIÓN DE COMPUTADORES

Laboratorio 3: Acercándose al Hardware: Programación en Lenguaje Ensamblador

Profesor: Viktor Tapia.
Alumno: Rodrigo Pereira Yánez

15 de junio de 2025

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Índice

1. Introducción.

Este informe detalla una experiencia práctica en el lenguaje ensamblador MIPS, el cual representa un paso esencial para comprender la interacción directa con la arquitectura de un procesador. El principal desafío de este trabajo radicó en aplicar de forma práctica los conceptos de MIPS para desarrollar programas complejos.

Para abordar este reto, se utilizó el simulador MARS (MIPS Assembler and Runtime Simulator) como herramienta principal. MARS permitió la escritura, el ensamblaje y la depuración de una variedad de programas en MIPS, traduciendo construcciones comunes de C como ciclos, estructuras de control, arreglos, funciones y llamadas al sistema para la entrada y salida de datos.

Los objetivos clave de este trabajo fueron:

- Dominar el uso de MARS para el desarrollo de programas MIPS.
- Implementar programas MIPS utilizando instrucciones aritméticas, de salto y de memoria.
- Profundizar en el uso de subrutinas en MIPS, prestando especial atención al manejo del stack.
- Realizar llamadas al sistema (syscall) para la interacción con el entorno.

A lo largo de este informe, se presentará el marco teórico relevante, se detallará el desarrollo de las soluciones implementadas para cada sección del laboratorio, se mostrarán los resultados obtenidos y, finalmente, se expondrán las conclusiones derivadas de esta valiosa experiencia práctica.

2. Marco Teórico.

2.1 Lenguaje Ensamblador y Arquitectura MIPS.

El **lenguaje ensamblador** es un lenguaje de programación de bajo nivel que representa las instrucciones básicas que un procesador puede ejecutar. Cada instrucción en ensamblador tiene una correspondencia directa con una instrucción en código máquina. Programar en ensamblador permite un control preciso sobre el hardware, aunque es más complejo y menos portable que los lenguajes de alto nivel.

La **arquitectura MIPS** (Microprocessor without Interlocked Pipeline Stages) es una arquitectura de conjunto de instrucciones reducido (RISC) ampliamente utilizada en entornos académicos debido a su relativa simplicidad y claridad. Se caracteriza por tener un conjunto de instrucciones regular y un gran número de registros de propósito general.

La arquitectura MIPS se basa en 4 principios de diseño fundamentales:

1. La simplicidad favorece la regularidad.
2. Hacer el caso común más rápido.
3. Lo pequeño es rápido.
4. Buen diseño demanda buenos compromisos.

2.2 Registros en MIPS.

La arquitectura MIPS dispone de un conjunto de 32 registros de propósito general, los cuales se clasifican y utilizan convencionalmente de la siguiente manera:

Tabla 2.1: Listado de registros y su descripción.

Registro	**Descripción/Uso Convencional**
`$zero`	Valor constante 0 (no se puede modificar)
`$at`	Registro temporal
`$v0 - $v1`	Retorno de valores de una función/subrutina. `$v0` también se usa para el código de `syscall`.
`$a0 - $a3`	Argumentos de funciones/subrutinas.
`$t0 - $t7`	Registros temporales.
`$s0 - $s7`	Registros guardados.
`$t8 - $t9`	Registros temporales adicionales
`$k0 - $k1`	Reservados para el kernel del sistema operativo.
`$gp`	Puntero global (apunta al área de datos estáticos).
`$sp`	Puntero de pila (Stack Pointer).
`$fp`	Puntero de marco (Frame Pointer).
`$ra`	Dirección de retorno de una función/subrutina.

2.3 Instrucciones MIPS.

El conjunto de instrucciones de MIPS se organiza en diversas categorías, incluyendo:

- **Carga (Load)**: Transfieren datos desde una ubicación en la memoria principal hacia los registros del procesador para su procesamiento.
- **Almacenamiento (Store)**: Mueven datos desde los registros del procesador de vuelta a una ubicación específica en la memoria principal.

- **Aritméticas**: Ejecutan operaciones matemáticas básicas como la suma, resta, multiplicación y división, almacenando los resultados en registros del procesador.
- **Lógicas**: Realizan operaciones lógicas bit a bit, como AND, OR, XOR y NOR, entre operandos y guardan el resultado en un registro.
- **Comparaciones**: Comparan los valores de dos registros y, basándose en el resultado, pueden establecer ciertos indicadores o utilizarse para decisiones en instrucciones de salto condicional.
- **Movimiento entre registros**: Permiten copiar el valor contenido en un registro a otro.
- **Desplazamiento (Shift)**: Realizan desplazamientos de bits a la izquierda o a la derecha dentro de un registro, lo que puede utilizarse para multiplicaciones o divisiones rápidas por potencias de dos.
- **Salto y Bifurcaciones (Jump and Branch)**: Alteran el flujo secuencial de la ejecución del programa. Los saltos incondicionales transfieren el control a una nueva dirección de memoria sin condiciones previas. Los saltos condicionales transfieren el control solo si se cumple una determinada condición (resultado de una comparación entre registros).

2.4 Segmentos de Memorias y Directivas de Ensamblador.

Un programa en ensamblador MIPS típicamente se organiza en segmentos:

- **Segmento de Datos** (`.data`): Se utiliza para declarar variables estáticas y constantes que el programa utilizará. Por ejemplo, se pueden definir cadenas de texto (con la directiva `.asciiz`) o arreglos de números (con la directiva `.word`).
- **Segmento de Texto** (`.text`): Contiene las instrucciones ejecutables del programa. Es aquí donde se escribe la lógica principal del código.

Las directivas como `.data`, `.text`, `.word`, `.asciiz`, `.space` (para reservar espacio en memoria) y `.globl` (para declarar etiquetas globales) son instrucciones para el lenguaje ensamblador que ayudan a organizar el código y los datos.

2.5 Subrutinas (Funciones).

Las subrutinas son bloques de código que realizan una tarea específica y pueden ser llamados desde diferentes partes del programa. En MIPS, la gestión de subrutinas implica:

- **Llamada**: La instrucción `jal <nombre_etiqueta>` (jump and link) salta a la subrutina y guarda la dirección de la siguiente instrucción (dirección de retorno) en el registro `$ra`.
- **Paso de Argumentos**: Generalmente, se utilizan los registros `$a0 - $a3` para pasar argumentos.
- **Retorno de Valores**: Los registros `$v0 - $v1` se usan para devolver resultados.

- **Retorno de la Subrutina**: La instrucción `jr $ra` (jump register) se utiliza para regresar al punto desde donde se llamó la subrutina, utilizando la dirección almacenada en `$ra`.

2.6 Llamas al sistema (syscall).

Las llamadas al sistema son el mecanismo mediante el cual un programa en MIPS interactúa con el sistema operativo o el simulador (MARS) para realizar operaciones de entrada/salida (E/S) o para terminar la ejecución. Para realizar una `syscall`:

- Se carga el código del servicio deseado en el registro `$v0`. Por ejemplo:
  - Código 1: Imprimir un entero
  - Código 4: Imprimir una cadena
  - Código 5: Leer un entero
  - Código 10: Terminar el programa.
- Si el servicio requiere argumentos, estos se cargan en los registros `$a0 - $a3` según corresponda.
- Se ejecuta la instrucción `syscall`. El simulador o el SO se encarga de realizar la operación solicitada.

2.7 Simulador MARS.

El simulador MARS (MIPS Assembler and Runtime Simulator) es un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) específicamente diseñado para facilitar la educación y el aprendizaje del lenguaje ensamblador MIPS. Desarrollado por los doctores Pete Sanderson y Kenneth Vollmar, MARS ofrece un entorno donde los estudiantes pueden experimentar directamente con la programación en MIPS sin la necesidad de hardware físico. Permite visualizar el funcionamiento interno de un procesador MIPS simulado, comprender el flujo de ejecución de los programas de manera detallada (paso a paso) y facilita significativamente la identificación y corrección de errores a través de sus herramientas de depuración.

3. Desarrollo de la Solución.

La resolución de los problemas se centró en la implementación de subrutinas sin para resolver las operaciones matemáticas como multiplicación, división y factorial, sin utilizar instrucciones de específicas para estas operaciones, todo escrito en código MIPS y con pruebas en el simulador MARS.

3.1 Parte 1: Subrutina para multiplicación y división de números.

3.1.1 Multiplicación de dos enteros (program1.asm).

Se implementó una subrutina en MIPS para la multiplicación de dos enteros sin usar instrucciones nativas de multiplicación, división o desplazamiento. La técnica empleada fue la **suma repetida**.

En la rutina principal (`main`), los operandos (9 y 7) se inicializaron «*en duro*» en los registros `$s0` y `$s1`. Se utilizaron subrutinas auxiliares `imprimir_integer`, `imprimir_string` para mostrar los valores y los símbolos de la operación en la consola.

La subrutina `multiplicación` recibió los operandos en `$a1` y `$a2`. Su lógica se basó en un bucle for: `$t0` actuó como contador, y `$t1` como acumulador del resultado. El primer múltiplo `$a1` se sumó repetidamente a `$t1` mientras `$t0` fuera menor que el segundo múltiplo `$a2`. Al finalizar, el resultado se movió a `$v0` y se retornó a `main`, donde fue impreso.

3.1.2 Cálculo de factorial (program2.asm).

A partir de la subrutina de multiplicación implementada previamente, se desarrolló un programa para calcular el factorial de un número entero, incorporando una validación para números no positivos. El programa principal `main` inicializa el número a calcular (en este caso, 6) en el registro `$s0`.

Antes de iniciar el cálculo factorial, se realiza una validación del número de entrada mediante la subrutina `validacion_cero_o_positivo`. Si el número es 0, se imprime 1 (ya que 0!=1) y el programa termina. Si el número es negativo, se muestra un mensaje de error «*Error! No se puede calcular el factorial de un número negativo.*» y el programa también finaliza. Si el número es positivo, la ejecución continúa con el cálculo del factorial.

La lógica del factorial se maneja con un bucle `while`. Se utilizan `$t2` para el valor actual del factorial `(value)`, y `$t3` como contador decremental `(count)`, inicializado en `value - 1`. El bucle (`while`) persiste mientras el contador `$t3` sea mayor o igual a 1. Dentro de cada iteración:

- El valor actual `$t2` y el contador `$t3` se pasan como argumentos `$a1`, `$a2` a la subrutina `multiplicación`.
- El resultado de la multiplicación se recupera de `$v0` y se actualiza en `$t2` `(value)`.
- El contador `$t3` se decrementa en uno.

Una vez que el bucle concluye, el resultado final del factorial, almacenado en `$t2`, se imprime utilizando `imprimir_integer`. Las subrutinas `imprimir_integer` e `imprimir_string` se reutilizan para una presentación clara en la consola, y el programa finaliza con una llamada al sistema de salida.

3.1.3 División de dos enteros (program3.asm).

Se implementó una subrutina en MIPS para la división de dos enteros, atendiendo a divisiones no exactas y calculando el cociente hasta dos decimales de precisión, sin utilizar instrucciones nativas de división ni desplazamiento. La técnica empleada fue la **resta repetida**.

En la rutina principal `main`, el dividendo (17) y el divisor (8) se inicializan «*en duro*» en los registros `$s0` y `$s1` respectivamente. Se utilizan las subrutinas `imprimir_integer` e `imprimir_string` para mostrar los operandos y los símbolos de la operación en la consola.

Antes de la división, se incorpora una subrutina `validacion_divisor` para asegurar que el divisor no sea cero. Si el divisor es 0, se imprime un mensaje de error «*Error! No se puede dividir por cero.*» y el programa termina.

La lógica principal de la división reside en la subrutina `división`, que recibe el dividendo en `$a1` y el divisor en `$a2`.

Cálculo de la Parte Entera:

- Se inicializan `$t0` como `cociente_entero` (igual a cero) y `$t1` como `resto_entero` (con el valor del dividendo).
- Un bucle `for_entero` resta repetidamente el `divisor $t2` del `resto_entero $t1`, incrementando el `cociente_entero $t0` en cada resta. Este proceso continúa mientras el `resto_entero` sea mayor o igual al `divisor`.
- El `cociente_entero` se imprime utilizando `imprimir_integer`.

Cálculo de la Parte Decimal (hasta dos decimales):

- Si el `resto_entero` es cero, la división es exacta y el proceso termina.
- Si hay un resto, se imprime un punto (.) usando `imprimir_string`.
- Para obtener dos decimales de precisión, el `resto_entero $a1` se multiplica por 100 utilizando la subrutina `multiplicación` previamente definida. El resultado se almacena en `$t1 resto_decimal`.
- Se inicializa `$t0` como `cociente_decimal` (a cero).

- Un segundo bucle `for_decimal` realiza el mismo proceso de resta repetida, esta vez restando el divisor `$t2` del `resto_decimal $t1`, incrementando el `cociente_decimal $t0`.
- Finalmente, el `cociente_decimal` se imprime con `imprimir_integer`.

Gestión de Registros en Subrutinas:

Es importante destacar que la subrutina división guarda y restaura el registro de dirección de retorno `$ra` en la pila `$sp` para preservar su valor, dado que llama a otras subrutinas.

El programa termina con una llamada al sistema de salida.

4. Resultados.

Todos los programas desarrollados fueron ensamblados y ejecutados satisfactoriamente en el simulador MARS. Las pruebas realizadas fueron con valores numéricos «*en duro*» y con variaciones de distintos números, incluyendo números negativos para la pregunta 1b) y cero para la parte 1b) y 1c). Se demostró el correcto funcionamiento de cada solución.

4.1 Parte 1: Subrutinas para multiplicación y división de números.

4.1.1 Multiplicación de dos enteros (program1.asm).

El programa implementó exitosamente (Figura 4.1.1) la operación de multiplicación. Mostró por consola los valores de la multiplicación, con formato especial para representar esta operación. La salida final fue (probado con diferentes valores):

```
None
9 x 7 = 63
-- program is finished running --

7 x 9 = 63
-- program is finished running --

7 x 0 = 0
-- program is finished running --

7 x 1 = 7
```

```
-- program is finished running --
```

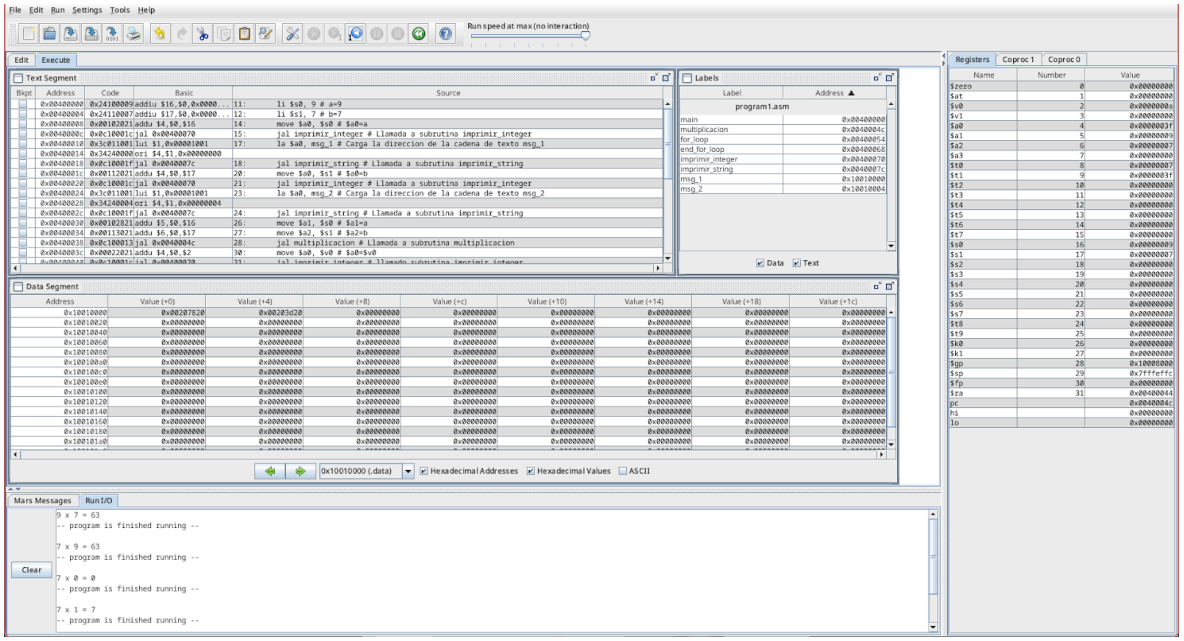

Figura 4.1.1: Execute - MARS, program1.asm.

4.1.2 Cálculo de factorial (program2.asm).

El programa implementó exitosamente (Figura 4.1.2) la operación de factorial usando el programa de la parte 1a). Mostró por consola los valores del factorial de diferentes números (incluyendo el cero y un número negativo), con formato especial para representar esta operación. La salida final fue:

```
None
0! = 1
-- program is finished running --

1! = 1
-- program is finished running --

4! = 24
-- program is finished running --

5! = 120
-- program is finished running --
```

```
6! = 720
-- program is finished running --

-6! = Error! No se puede calcular el factorial de un número negativo.
-- program is finished running --
```

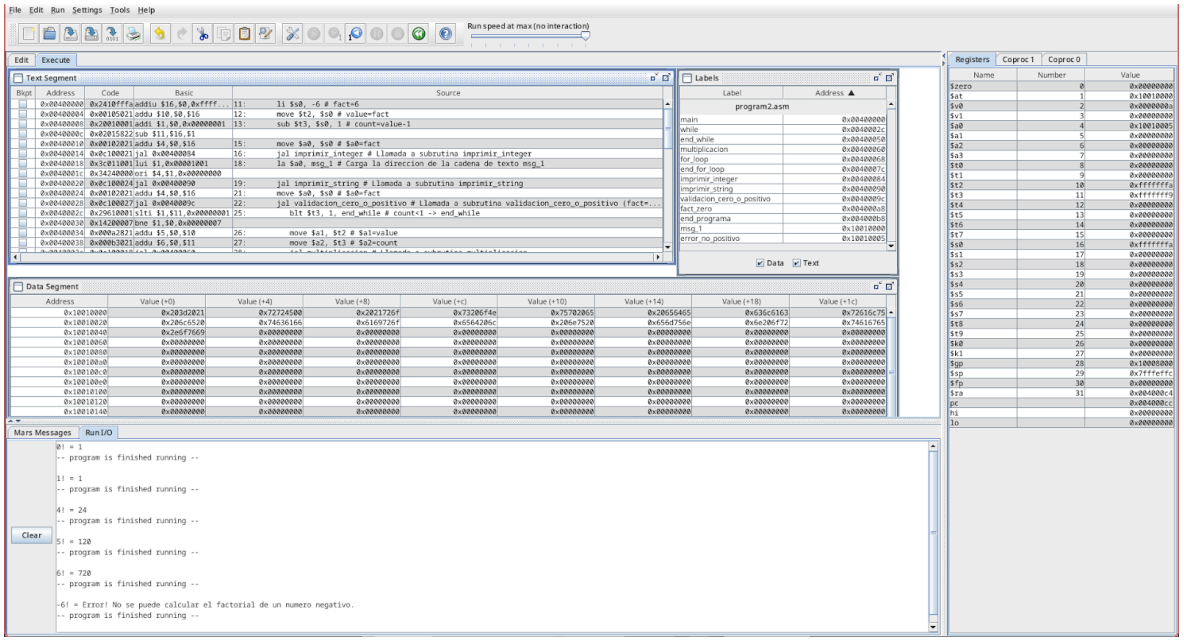

Figura 4.1.2: Execute - MARS, program2.asm.

4.1.3 División de dos enteros (program3.asm).

El programa implementó exitosamente (Figura 4.1.3) la operación de división tanto para divisiones exactas (resto cero) como no exactas (con decimales) mostrando hasta 2 decimales. Se mostró por consola los valores de la división de diferentes números (incluyendo el cero), con formato especial para representar esta operación. La salida final fue:

```
None
16 / 8 = 2
-- program is finished running --

17 / 8 = 2.12
-- program is finished running --
```

```
17 / 0 = Error! No se puede dividir por cero.
-- program is finished running --
```

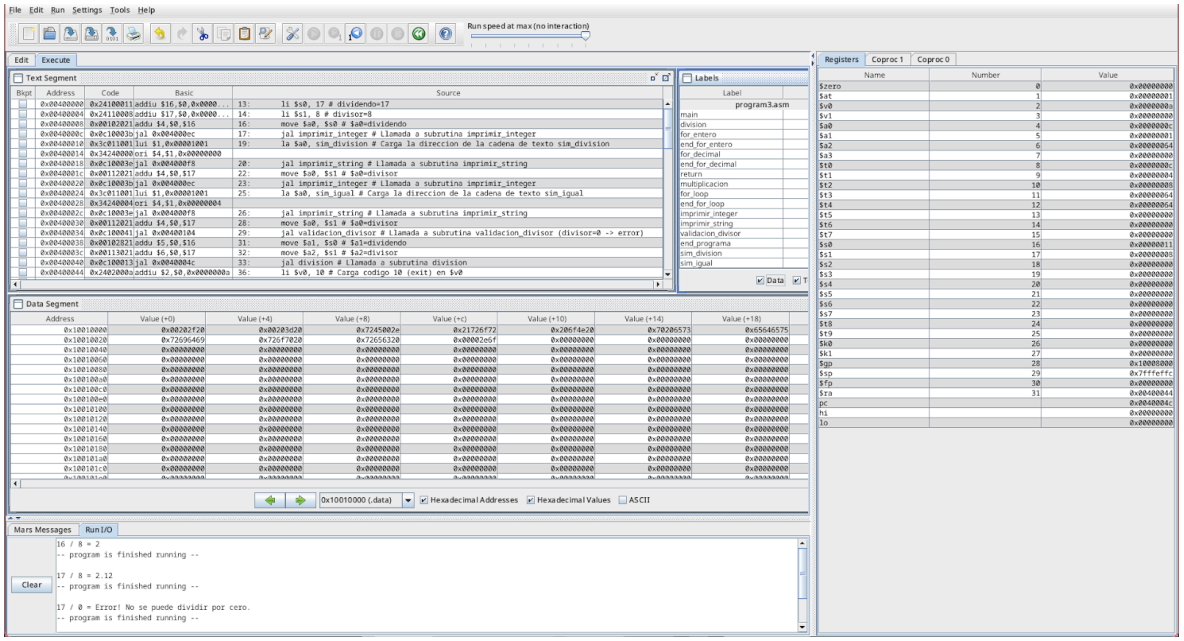

Figura 4.1.3: Execute - MARS, program3.asm.

Conclusiones.

Este laboratorio fue crucial para profundizar en la programación en lenguaje ensamblador MIPS y la interacción directa con la arquitectura del procesador, usando el simulador MARS.

El mayor desafío fue implementar multiplicación y división sin instrucciones nativas, usando suma y resta repetida, y manejando la precisión decimal y casos especiales. Esto reforzó el dominio de las subrutinas (para operaciones, impresión y validación), incluyendo el paso de argumentos, retorno de valores y la gestión del stack.

En resumen, la experiencia mejoró la comprensión de la programación de bajo nivel y la capacidad de construir algoritmos complejos con control preciso del hardware simulado, cumpliendo todos los objetivos.

Referencias.

1. MARS MIPS Simulator. (s.f.). [https://dpetersanderson.github.io/index.html].
2. Patterson, D. A., & Hennessy, J. L. (2014). Computer organization and design: The hardware/software interface. Morgan Kaufmann.