ACTIVIDAD LABORATORIO NO.1
SIMULACIÓN Y OPTIMIZACIÓN DE UN PROGRAMA EN UN PROCESADOR ESCALAR SEGMENTADO

PRESENTADO POR:
ALEJANDRO DE MENDOZA TOVAR

PRESENTADO AL PROFESOR:
ING DEIVIS EDUARD RAMIREZ MARTINEZ
PROFESOR

FUNDACIÓN UNIVERSITARIA INTERNACIONAL DE LA RIOJA
FACULTAD DE INGENIERÍA – INGENIERIA INFORMÁTICA
BOGOTÁ D.C.
4 DE SEPTIEMBRE
2025

TABLA DE CONTENIDO

RESUMEN

Para el desarrollo de este laboratorio correspondiente al área de Estructura de Computadores, se emplea el análisis detallado de la ejecución de programas en lenguaje ensamblador bajo el entorno de simulación MARS (MIPS Assembler and Runtime Simulator). A partir de la implementación de tres scripts específicos (cálculo del número mayor, cálculo del número menor y generación de la serie de Fibonacci), se busca comprender de manera práctica el funcionamiento de un procesador escalar segmentado y el impacto de la programación a bajo nivel en el control del hardware.

A lo largo del proceso, se identifican y aplican los elementos fundamentales de la arquitectura MIPS, entre los que se incluyen el uso de registros, instrucciones aritméticas y de control, así como las llamadas al sistema para interacción con el usuario mediante consola. Cada uno de estos componentes representa una función clave dentro del flujo de ejecución del procesador, por lo que su correcta comprensión resulta esencial para optimizar el rendimiento del programa y garantizar resultados precisos.

Para lograr un desarrollo estructurado, primero se establecen las operaciones a implementar: comparación de múltiples números ingresados por el usuario (para determinar el mayor y el menor), y la generación de secuencias matemáticas (serie de Fibonacci). Posteriormente, se detallan las fases de ejecución de cada script, abarcando desde la carga de datos hasta la presentación de resultados en pantalla. Este proceso permite evidenciar el ciclo completo de ejecución de instrucciones en un procesador segmentado, lo cual constituye una base sólida para comprender las dinámicas de paralelismo y optimización.

Además, se resalta la importancia del uso de herramientas de simulación como MARS, que permiten visualizar en tiempo real la interacción entre registros, memoria e instrucciones, ofreciendo al estudiante un entorno didáctico para explorar y validar conceptos teóricos de arquitectura de computadores. Este enfoque facilita la detección de errores, la depuración del código y la observación del impacto de cada instrucción en el flujo general del programa.

Con este trabajo se evidencia que el estudio del lenguaje ensamblador no solo responde a un requisito académico, sino que también constituye una práctica formativa que fortalece la comprensión del hardware, el diseño de programas eficientes y la capacidad de análisis a bajo nivel. De esta manera, la programación en MIPS se concibe no como un proceso aislado, sino como parte integral de la formación en ingeniería informática, estrechamente vinculada con los valores de precisión, optimización y control técnico que rigen la disciplina.

Finalmente, se concluye que la elaboración y ejecución de estos programas en ensamblador representa una herramienta esencial para vincular los conocimientos teóricos de arquitectura de computadores con la práctica aplicada. Este ejercicio no solo permite consolidar competencias técnicas, sino que también fomenta una cultura académica orientada a la disciplina, la lógica y el rigor en el desarrollo de software de bajo nivel.

1. INTRODUCCIÓN

El presente laboratorio tiene como finalidad profundizar en la comprensión del funcionamiento de los procesadores a través del lenguaje ensamblador MIPS, empleando como entorno de trabajo el simulador MARS (MIPS Assembler and Runtime Simulator). Esta herramienta ofrece un espacio interactivo para la construcción, compilación, ejecución y depuración de programas, lo que facilita al estudiante experimentar de manera directa el ciclo de vida de las instrucciones y su impacto en los registros, la memoria y el flujo de control de un procesador escalar segmentado.

A diferencia de los lenguajes de programación de alto nivel, que priorizan la abstracción y la facilidad de uso, el lenguaje ensamblador se caracteriza por su cercanía al hardware, permitiendo un control preciso de las operaciones que ejecuta la máquina. Esta característica convierte al ensamblador en una herramienta formativa indispensable para los futuros ingenieros en sistemas y computación, ya que fomenta la capacidad de analizar, optimizar y comprender cómo las instrucciones se traducen en acciones concretas dentro del procesador.

El desarrollo de este laboratorio se centra en la implementación de tres programas con objetivos específicos:

1. Determinar el número mayor entre un conjunto de valores ingresados por el usuario.
2. Determinar el número menor entre un conjunto de valores ingresados por el usuario.
3. Generar la serie de Fibonacci hasta una cantidad indicada por el usuario e identificar la suma total de los valores obtenidos.

Cada uno de estos ejercicios busca afianzar el manejo de operaciones aritméticas, comparaciones, estructuras de control, y la interacción con el usuario mediante llamadas al sistema. Asimismo, constituyen ejemplos prácticos que permiten aplicar conceptos como el manejo de registros temporales, el uso de saltos condicionales y el control del flujo de ejecución.

Además, este laboratorio representa una oportunidad para explorar el valor pedagógico de las simulaciones, ya que el entorno MARS no solo proporciona una interfaz amigable, sino también herramientas visuales para observar en tiempo real el comportamiento de registros, memoria e instrucciones. De esta manera, el estudiante logra una visión más clara y tangible del paralelismo, segmentación y eficiencia que caracterizan a los procesadores modernos.

En suma, el trabajo aquí desarrollado no se limita a la creación de simples scripts en ensamblador, sino que constituye un ejercicio integral que vincula teoría y práctica, fomenta el razonamiento lógico y fortalece competencias clave en la formación profesional. El dominio del lenguaje ensamblador, más allá de su utilidad inmediata, representa un paso fundamental hacia la comprensión profunda de la arquitectura de computadores, sentando las bases para el diseño, la optimización y la innovación en el campo de la ingeniería informática.

2. DESARROLLO DEL LABORATORIO

Para comprender a profundidad el uso del lenguaje ensamblador MIPS y su aplicación en la resolución de problemas computacionales, la sección de desarrollo se ha organizado en tres partes principales (correspondientes a los tres scripts solicitados en la práctica), cada una con un enfoque específico:

1. Determinación del número mayor: se implementa un programa que permite al usuario ingresar entre tres y cinco valores, almacenarlos y, mediante comparaciones sucesivas, identificar cuál de ellos corresponde al mayor.

2. Determinación del número menor: de manera análoga al primer programa, se solicita al usuario ingresar un conjunto de números (mínimo tres y máximo cinco) y, a través de estructuras condicionales, se determina cuál es el menor de todos los digitados.

3. Serie de Fibonacci: se desarrolla un programa que, a partir de la cantidad indicada por el usuario, genera la secuencia de Fibonacci e imprime cada uno de los elementos de la serie. Adicionalmente, se calcula la suma total de los términos generados, lo cual refuerza el uso de acumuladores y estructuras iterativas en ensamblador.

Cada una de estas partes está orientada a fortalecer el dominio del estudiante en aspectos como: manejo de registros, instrucciones de comparación y salto, interacción mediante llamadas al sistema, y organización lógica del código ensamblador. Asimismo, se incluyen evidencias gráficas en tres momentos distintos; antes de compilar, después de compilar y después de ejecutar, con el fin de documentar todo el proceso de construcción, depuración y validación del programa.

2.1 Herramientas utilizadas

- **MARS (MIPS Assembler and Runtime Simulator)** version 4.5.

- **Lenguaje ensamblador MIPS**.
- **GitHub** para la gestión y publicación de los scripts.

2.2 Script 1: Número Mayor

El programa solicita al usuario cuántos números desea ingresar (mínimo 3 y máximo 5). Posteriormente, los compara e imprime en pantalla el mayor valor introducido.

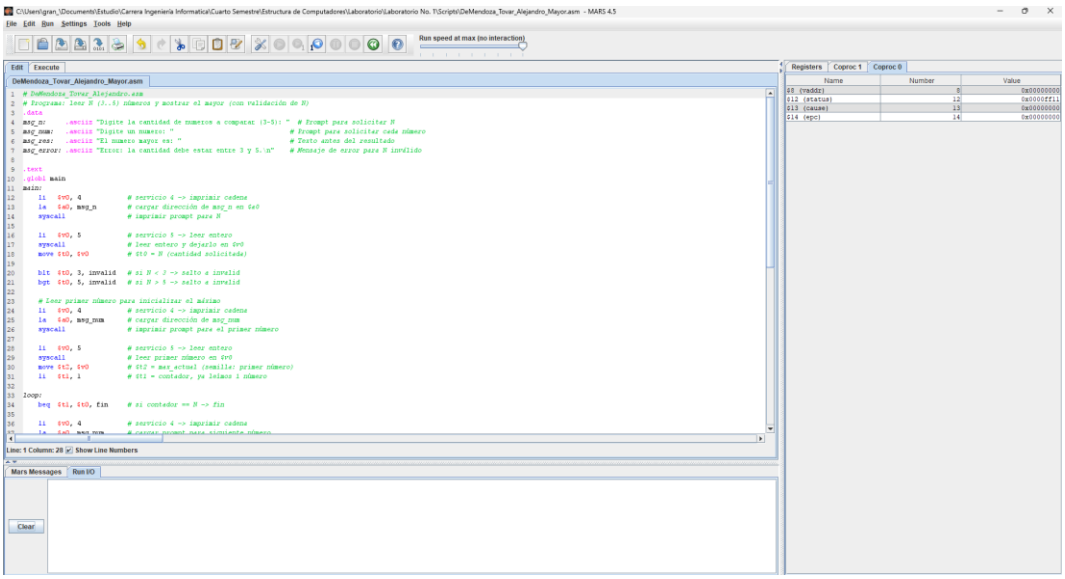
2.2.1 Capturas de pantalla y explicación del código

A continuación doy a conocer la explicación del código y las capturas de pantalla del desarrollo de este script desde antes de compilar, después de compilar y después de su ejecución para dar claridad el desarrollo del código:

a) Antes de compilar

En esta etapa se muestra el **código fuente ensamblador escrito** en el editor de MARS. El archivo DeMendozaTovarAlejandro_Mayor.asm contiene todas las instrucciones necesarias para solicitar los números al usuario, almacenarlos y determinar cuál es el mayor.

La captura evidencia que el programa está escrito correctamente en MIPS pero aún **no ha sido traducido a código máquina**. Es decir, el código se encuentra en estado de **preparación**, listo para pasar al ensamblado. A continuación la imagen respectiva:



b) Explicación del código línea por línea:

A continuación procedo a brindar la explicación de todo el código para una mayor comprensión:

Datos y texto

- .data → Indica el inicio de la sección de datos (mensajes y constantes).
- msg_n: .asciiz "Digite la cantidad de numeros a comparar (3-5): " → Mensaje que pide al usuario la cantidad de números a ingresar.
- msg_num: .asciiz "Digite un numero: " → Mensaje que se imprime antes de cada número.
- msg_res: .asciiz "El número mayor es: " → Texto previo a mostrar el resultado.
- msg_error: .asciiz "Error: la cantidad debe estar entre 3 y 5.\n" → Mensaje de error cuando N está fuera de rango.

Código principal

- .text → Comienza la sección de instrucciones ejecutables.
- .globl main → Define main como punto de entrada.
- main: → Marca el inicio del programa.

Pedir cantidad y validar

- li $v0, 4 → Servicio de imprimir cadena.
- la $a0, msg_n → Cargar dirección del mensaje msg_n.
- syscall → Mostrar el texto en consola.
- li $v0, 5 → Servicio de leer entero.
- syscall → Captura el valor de N.
- move $t0, $v0 → Guarda el número leído en $t0 (cantidad N).
- blt $t0, 3, invalid → Si N < 3, va a la etiqueta invalid.
- bgt $t0, 5, invalid → Si N > 5, también va a invalid.
- Así se valida que N esté dentro del rango [3..5].

Leer primer número (semilla del máximo)

- li $v0, 4 → Imprimir cadena.
- la $a0, msg_num → Cargar el mensaje para pedir un número.
- syscall → Mostrar mensaje.
- li $v0, 5 → Leer entero.
- syscall → Captura el primer número ingresado.
- move $t2, $v0 → Se guarda en $t2, que será el máximo actual.
- li $t1, 1 → Inicializa el contador en 1 (ya se ingresó un número).

Bucle para leer y comparar

- loop: → Inicio del ciclo.
- beq $t1, $t0, fin → Si el contador ($t1) llegó a la cantidad N, termina el bucle.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg_num → Cargar mensaje para pedir número.
- syscall → Mostrar en pantalla.
- li $v0, 5 → Leer entero.
- syscall → Captura el número ingresado.
- move $t3, $v0 → Se almacena en $t3.

Comparación y actualización

- ble $t3, $t2, skip → Si el número ingresado es menor o igual al máximo actual, no cambia nada y salta a skip.
- move $t2, $t3 → Si el número ingresado es mayor, actualiza $t2.

Incremento y repetición

- skip: → Continuación del ciclo.
- addi $t1, $t1, 1 → Aumenta el contador en 1.
- j loop → Regresa al inicio del bucle.

Mostrar resultado

- fin: → Marca el final del proceso.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg_res → Cargar mensaje de resultado.
- syscall → Mostrar en consola.
- li $v0, 1 → Servicio de imprimir entero.

- move $a0, $t2 → Pasar el valor máximo a $a0.
- syscall → Imprime el número mayor.
- li $v0, 10 → Servicio para terminar programa.
- syscall → Finaliza la ejecución.

Caso de error

- invalid: → Etiqueta de manejo de error.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg_error → Cargar mensaje de error.
- syscall → Mostrar mensaje de error.
- li $v0, 10 → Terminar programa.
- syscall → Fin de ejecución.

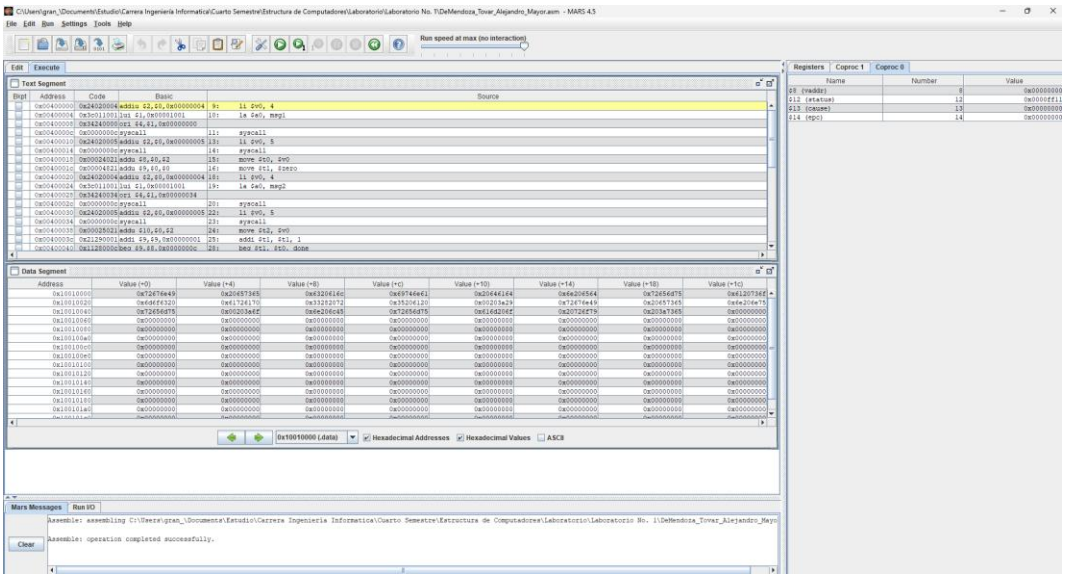
Resumen de registros (Mayor)

- $t0 → Cantidad de números a comparar (N).
- $t1 → Contador de números leídos.
- $t2 → Valor máximo encontrado hasta el momento.
- $t3 → Número leído en cada iteración.

c) Después de compilar

En esta etapa, el archivo DeMendozaTovarAlejandro_Mayor.asm ya ha sido procesado mediante el botón Assemble de MARS. Esto significa que el código fuente en ensamblador ha sido traducido a instrucciones en lenguaje máquina que el simulador puede interpretar.

En la interfaz se observa la ventana de instrucciones cargadas en memoria con su representación en hexadecimal y binario, así como los registros listos para recibir datos durante la ejecución. Esta evidencia confirma que el programa no presenta errores de sintaxis y que está preparado para ejecutarse correctamente. A continuación la imagen respectiva:



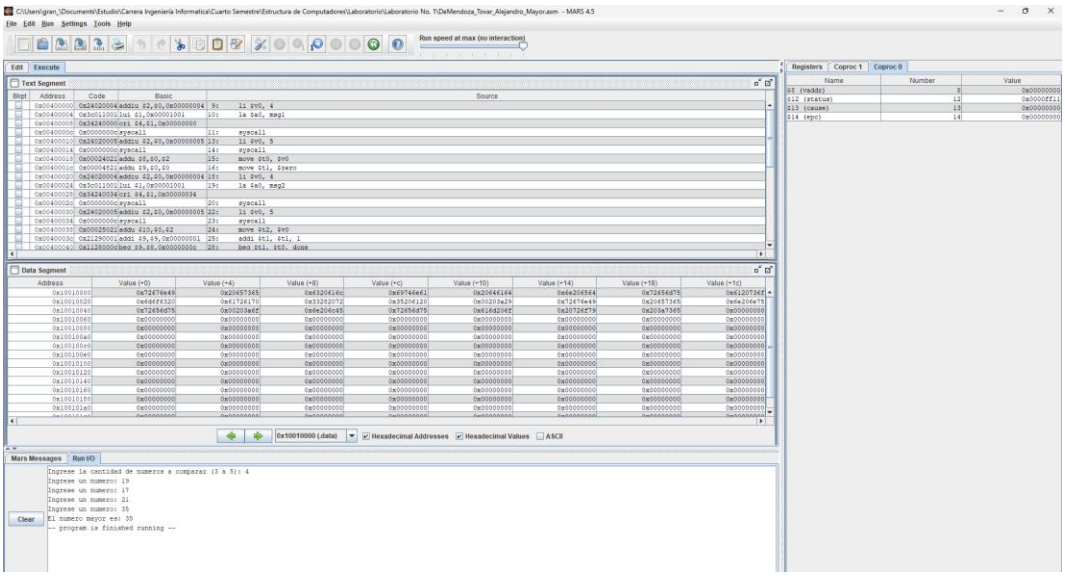
d) Después de ejecutar

En la ejecución del programa correspondiente a la búsqueda del número mayor, procedí a ingresar los valores **19, 17, 21 y 35**. El algoritmo fue comparando cada número secuencialmente hasta determinar cuál era el mayor.

El resultado final mostrado por consola fue:

- **“El número mayor es: 35”**

Esto confirma que el programa cumplió correctamente con la lógica de comparación implementada, mostrando el valor más alto entre los números ingresados por el usuario. A continuación la imagen respectiva:



2.3 Script 2: Número Menor

En este caso el programa solicita al usuario cuántos números desea ingresar (mínimo 3 y máximo 5). Posteriormente, los compara e imprime en pantalla el menor valor introducido.

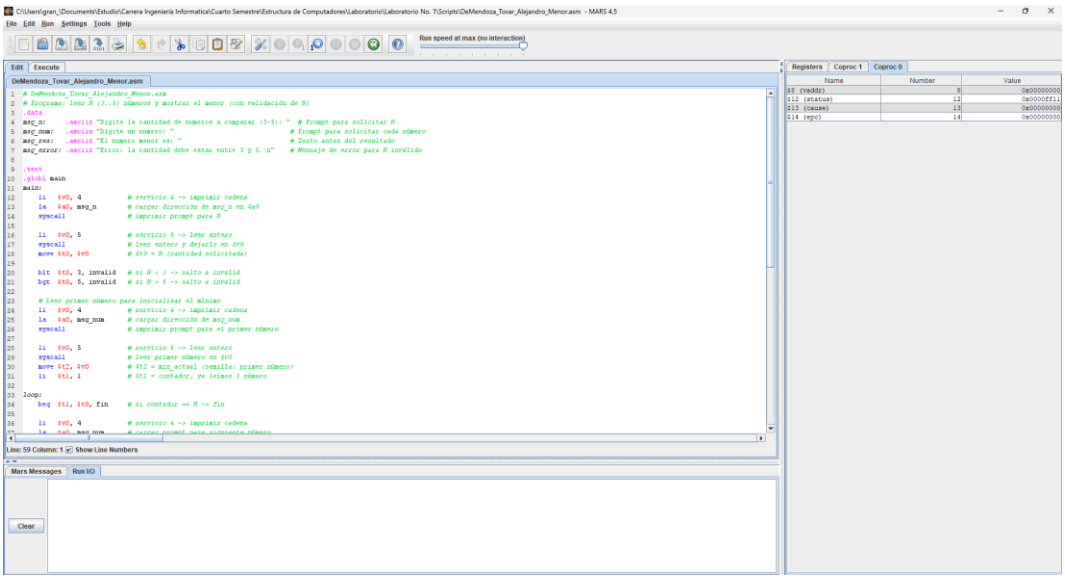
2.3.1 Capturas de pantalla y explicación del código

A continuación doy a conocer la explicación del código y las capturas de pantalla del desarrollo de este script desde antes de compilar, después de compilar y después de su ejecución para dar claridad el desarrollo del código:

a) Antes de compilar

Aquí se presenta en el editor el archivo DeMendozaTovarAlejandro_Menor.asm. Se observa el conjunto de instrucciones que permiten recibir varios números por entrada y compararlos para identificar el menor valor.

La captura refleja que el programa todavía está en su fase inicial de edición, sin haber pasado por el proceso de Assemble. En este punto se pueden verificar detalles como la estructura del programa, las secciones .data y .text, y la presencia de comentarios en cada línea para explicar la lógica. A continuación la imagen respectiva:



b) Explicación del código línea por línea:

A continuación procedo a brindar la explicación de todo el código para una mayor comprensión:

Datos y texto

- .data → Indica el inicio de la sección de datos (constantes y mensajes).
- msg_n: .asciiz "Digite la cantidad de numeros a comparar (3-5): " → Cadena para pedir la cantidad de números que se van a ingresar.
- msg_num: .asciiz "Digite un numero: " → Cadena para solicitar cada número individual.
- msg_res: .asciiz "El número menor es: " → Cadena que se muestra antes de imprimir el resultado.
- msg_error: .asciiz "Error: la cantidad debe estar entre 3 y 5.\n" → Mensaje de error que aparece si el valor de N no está en el rango permitido.

Código

- .text → Comienza la sección de instrucciones ejecutables.
- .globl main → Define la etiqueta main como el punto de entrada del programa.
- main: → Marca el inicio del programa.

Pedir cantidad y validar

- li $v0, 4 → Servicio para imprimir cadena.
- la $a0, msg_n → Cargar dirección de msg_n.
- syscall → Imprime el mensaje para pedir N.
- li $v0, 5 → Servicio para leer un entero.
- syscall → Espera la entrada del usuario.
- move $t0, $v0 → Se guarda el valor leído en $t0 (cantidad N).
- blt $t0, 3, invalid → Si N es menor a 3, salta a la rutina invalid.
- bgt $t0, 5, invalid → Si N es mayor a 5, también salta a invalid.
- De esta manera se valida que N esté dentro del rango [3..5].

Leer primer número (semilla del mínimo)

- li $v0, 4 → Imprimir cadena.
- la $a0, msg_num → Cargar el prompt para pedir número.
- syscall → Muestra el mensaje.
- li $v0, 5 → Leer un entero.
- syscall → Captura el número del usuario.

- move $t2, $v0 → Se guarda en $t2, que será el valor mínimo actual.
- li $t1, 1 → Se inicializa el contador en 1 (ya tenemos un número leído).

Bucle para leer y comparar

- loop: → Marca el inicio del ciclo.
- beq $t1, $t0, fin → Si ya se leyeron N números, salta a la rutina de finalización.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg_num → Cargar el mensaje para pedir otro número.
- syscall → Mostrar en pantalla.
- li $v0, 5 → Leer un entero.
- syscall → Captura el número ingresado.
- move $t3, $v0 → Se almacena en $t3.

Comparación y actualización

- bge $t3, $t2, skip → Si el número ingresado es mayor o igual al mínimo actual, no cambia nada y salta a skip.
- move $t2, $t3 → Si el número ingresado es menor, se actualiza $t2 con este nuevo valor.

Incremento y repetición

- skip: → Continuación del bucle.
- addi $t1, $t1, 1 → Incrementa el contador en 1.
- j loop → Regresa al inicio del ciclo.

Mostrar resultado

- fin: → Marca la parte final del programa.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg_res → Cargar el mensaje de resultado.
- syscall → Mostrar en pantalla.
- li $v0, 1 → Servicio para imprimir entero.
- move $a0, $t2 → Pasar el valor mínimo al registro $a0.
- syscall → Imprimir el número menor.
- li $v0, 10 → Servicio para terminar el programa.
- syscall → Fin de ejecución.

Caso de error

- invalid: → Etiqueta para manejar entradas inválidas.
- li $v0, 4 → Servicio para imprimir cadena.
- la $a0, msg_error → Cargar mensaje de error.
- syscall → Mostrar el error en pantalla.
- li $v0, 10 → Servicio para terminar programa.
- syscall → Fin de ejecución.

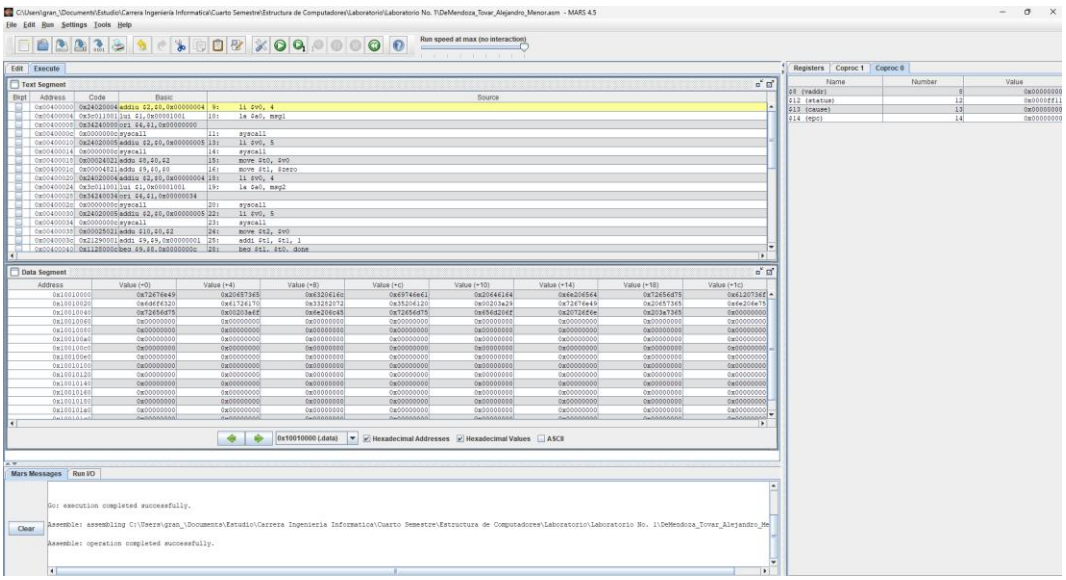
Resumen de registros (Menor)

- $t0 → Cantidad de números a comparar (N).
- $t1 → Contador de números leídos.
- $t2 → Valor mínimo encontrado hasta el momento.
- $t3 → Número leído en cada iteración.

c) Después de compilar

Tras ensamblar el archivo DeMendozaTovarAlejandro_Menor.asm, se puede apreciar que el código fuente fue convertido exitosamente a código máquina sin mostrar errores en consola. En la zona inferior del simulador aparecen las instrucciones traducidas, cada una asociada a una dirección de memoria específica.

Este resultado asegura que el programa es válido desde el punto de vista sintáctico y que el siguiente paso será ejecutarlo para comprobar su lógica funcional. La captura en este punto demuestra el éxito del proceso de compilación antes de proceder con la ejecución. A continuación la imagen respectiva:



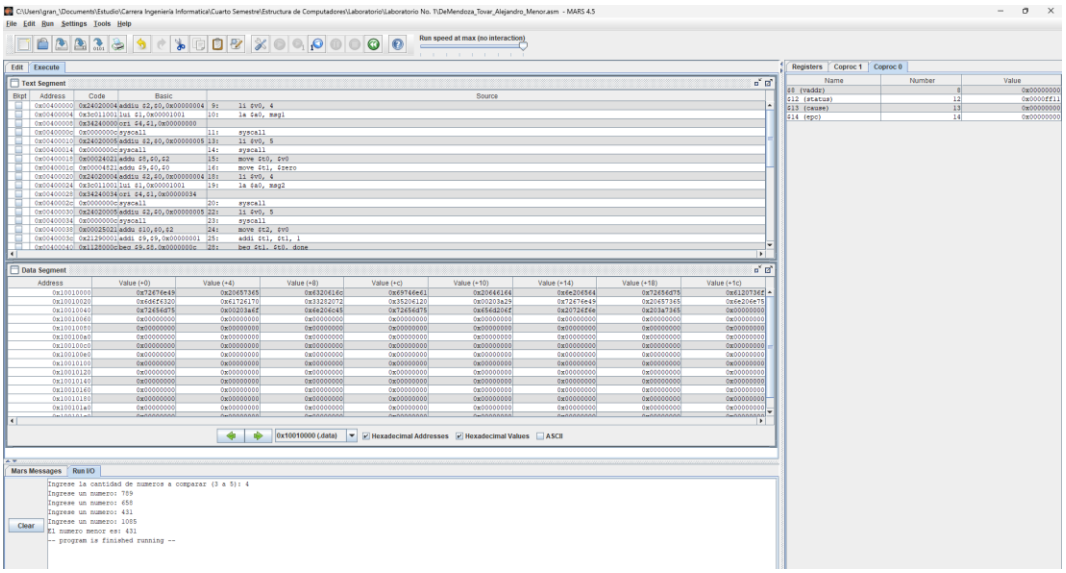
d) Después de ejecutar

En la ejecución del programa correspondiente a la búsqueda del número menor, procedí a ingresar los valores **785, 658, 431 y 1085**. El programa fue verificando cada uno de los valores y actualizando el menor registrado en memoria.

El resultado final mostrado en la consola fue:

- **“El número menor es: 431”**

De esta manera, se valida que el código realiza la comparación adecuada para identificar el valor más pequeño entre los números digitados por el usuario. A continuación la imagen respectiva:



2.4 Script 3: Serie Fibonacci

Ahora para este caso el programa solicita al usuario un número *n* y genera en consola los primeros *n* términos de la serie Fibonacci, además de mostrar la suma total de todos los valores generados.

2.4.1 Capturas de pantalla y explicación del código

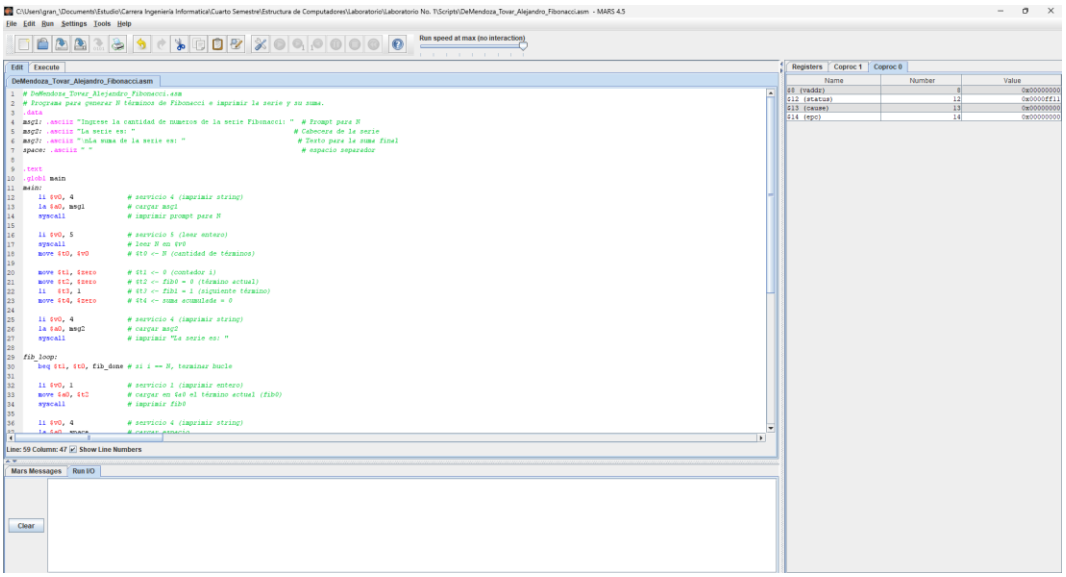
A continuación doy a conocer la explicación del código y las capturas de pantalla del desarrollo de este script desde antes de compilar, después de compilar y después de su ejecución para dar claridad el desarrollo del código:

a) Antes de compilar

En este caso, la captura corresponde al archivo DeMendozaTovarAlejandro_Fibonacci.asm abierto en el editor de MARS. El código está preparado para leer la cantidad de números de la serie que el usuario desea, generarlos en consola y calcular su suma total.

La evidencia muestra que, previo a la compilación, el código aún se encuentra en formato legible para el programador, sin ser traducido a instrucciones de máquina. Esto resalta la importancia de la etapa previa a ensamblar, ya que permite comprobar la claridad de la lógica y los comentarios incluidos en el programa.

En conclusión, en las tres capturas “antes de compilar” se busca demostrar que el estudiante escribió y organizó correctamente el código ensamblador en MARS, listo para pasar al ensamblador, pero todavía en estado de código fuente. A continuación la imagen respectiva:



b) Explicación del código línea por línea:

A continuación procedo a brindar la explicación de todo el código para una mayor comprensión:

Datos y texto

- .data → Sección de datos.
- msg1: .asciiz "Ingrese la cantidad de números de la serie Fibonacci: " → Prompt para N.
- msg2: .asciiz "La serie es: " → Encabezado de la serie.
- msg3: .asciiz "\nLa suma de la serie es: " → Texto para la suma final.
- space: .asciiz " " → Un espacio para separar números.

Código

- .text → Sección de código.
- .globl main → Punto de entrada.
- main: → Inicio.

Pedir N

- li $v0, 4 → Imprimir cadena.
- la $a0, msg1 → “Ingrese la cantidad…”.
- syscall
- li $v0, 5 → Leer entero.
- syscall
- move $t0, $v0 → $t0 = N (cantidad de términos).
- move $t1, $zero → $t1 = 0 (contador i).
- move $t2, $zero → $t2 = 0 (fib0).
- li $t3, 1 → $t3 = 1 (fib1).
- move $t4, $zero → $t4 = 0 (acumulador/suma).

Imprimir encabezado de la serie

- li $v0, 4 → Imprimir cadena.
- la $a0, msg2 → “La serie es: ”.
- syscall

Bucle de generación/impr.

- fib_loop: → Etiqueta del bucle.
- beq $t1, $t0, fib_done → Si i == N, termina.

Imprimir término actual (fib0)

- li $v0, 1 → Imprimir entero.
- move $a0, $t2 → $a0 = fib0.
- syscall → Muestra el término.

Imprimir espacio separador

- li $v0, 4 → Imprimir cadena.
- la $a0, space → Un espacio.
- syscall

Actualizar suma y avanzar la serie

- add $t4, $t4, $t2 → suma += fib0.
- add $t5, $t2, $t3 → $t5 = fib0 + fib1 (siguiente término).
- move $t2, $t3 → fib0 = fib1.
- move $t3, $t5 → fib1 = siguiente.
- addi $t1, $t1, 1 → i++ (avanza contador).
- j fib_loop → Repite.

Salida: imprimir suma y terminar

- fib_done: → Fin del bucle.
- li $v0, 4 → Imprimir cadena.
- la $a0, msg3 → “\nLa suma de la serie es: ”.
- syscall
- li $v0, 1 → Imprimir entero.
- move $a0, $t4 → $a0 = suma.
- syscall
- li $v0, 10 → Salir.
- syscall

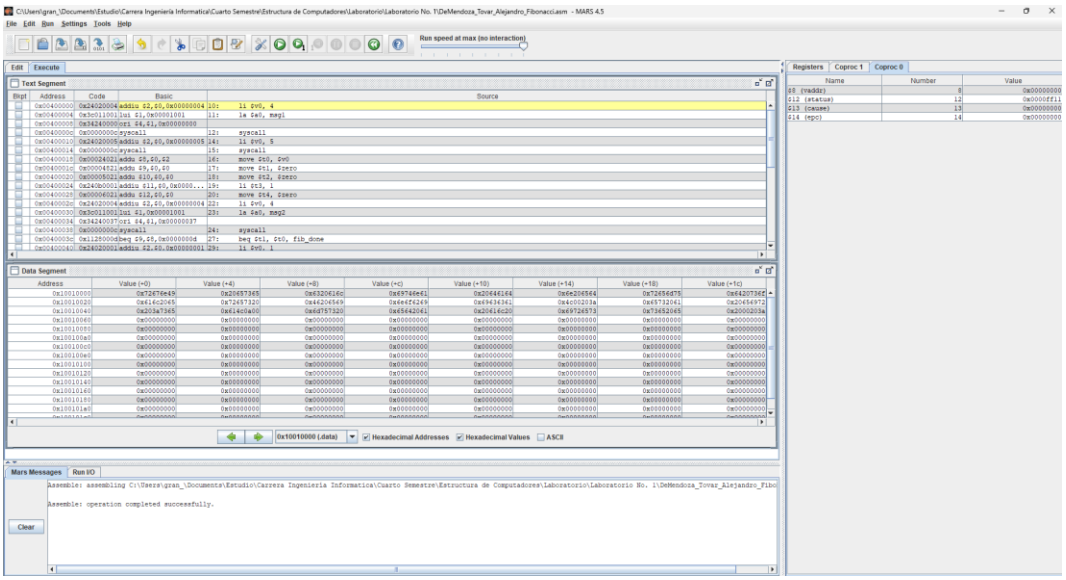
Resumen de registros (Fibonacci):

- $t0 = N (cantidad de términos)
- $t1 = i (contador)
- $t2 = fib0 (término actual impreso)
- $t3 = fib1 (siguiente término)
- $t4 = suma acumulada
- $t5 = temporal para fib0 + fib1

c) Después de compilar

En el caso del archivo DeMendozaTovarAlejandro_Fibonacci.asm, al pulsar Assemble se genera la traducción del código a instrucciones máquina. La evidencia muestra que cada instrucción del ensamblador tiene su correspondencia en código binario y hexadecimal dentro de la memoria simulada.

El hecho de que no se presenten mensajes de error indica que la estructura del programa está correcta y que puede ser ejecutada. Esta etapa representa la validación formal del código, dejando el programa en condiciones óptimas para correr y comprobar la generación de la serie Fibonacci. A continuación la imagen respectiva:



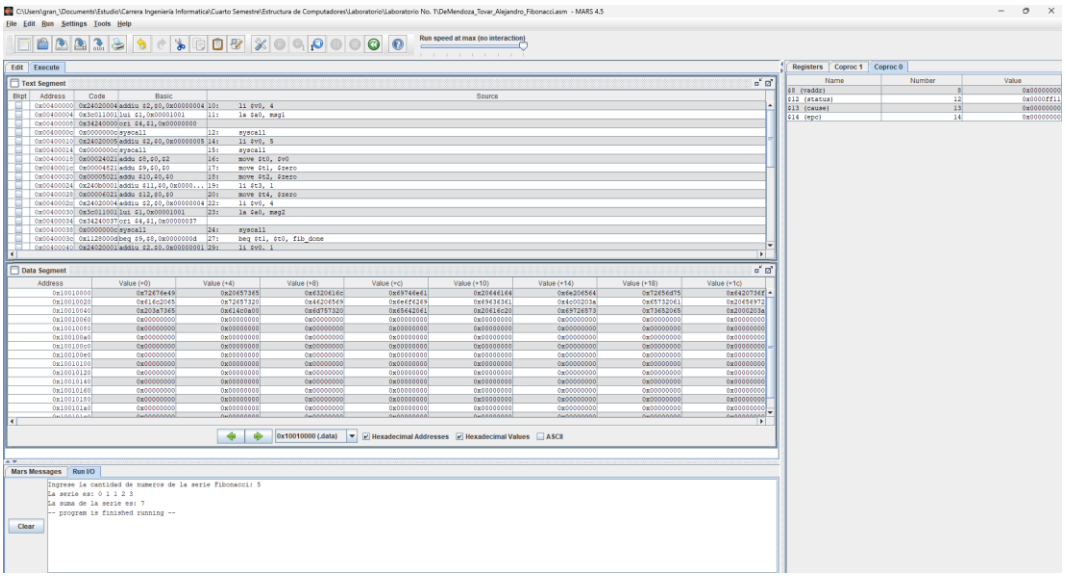
d) Después de ejecutar

En la ejecución del programa de la **Serie Fibonacci**, procedí a indicar que deseaba generar los **5 primeros números** de la secuencia. El programa imprimió la serie desde el valor inicial 0, siguiendo con los cálculos sucesivos hasta completar la cantidad indicada.

El resultado mostrado en la consola fue:

- **“La serie es: 0 1 1 2 3”**
- **“La suma de la serie es: 7”**

Con ello se comprueba que el programa no solo generó la secuencia correctamente, sino que también calculó de manera precisa la suma de los elementos de la serie solicitada. A continuación la imagen respectiva:



2.5 Enlace al repositorio de GitHub

Para iniciar este punto considero pertinente indicar además de lo indicado en la clase, que GitHub es una plataforma en línea basada en la nube que permite almacenar, gestionar y compartir proyectos de software. Funciona sobre Git, un sistema de control de versiones que permite llevar un historial detallado de los cambios realizados en los archivos de un proyecto.

Gracias a GitHub, varios desarrolladores pueden trabajar de manera colaborativa sobre el mismo código, facilitando el control de versiones, la integración de mejoras y la detección de errores.

2.5.1 Características principales de GitHub

A continuación denoto las principales características que en adición, considero clave de GitHub:

- Control de versiones: permite ver qué cambios se hicieron en cada archivo, quién los hizo y en qué momento.
- Repositorios: son espacios donde se guardan los proyectos, incluyendo código, documentación y recursos relacionados.
- Colaboración: múltiples usuarios pueden contribuir al mismo proyecto usando ramas (branches), solicitudes de cambio (pull requests) y revisiones de código.
- Accesibilidad: al estar en la nube, los proyectos se pueden consultar desde cualquier lugar.
- Integración: GitHub se conecta fácilmente con otras herramientas de desarrollo, entornos de prueba y despliegue automático.

2.5.2 Importancia en el laboratorio académico

En el contexto de este laboratorio de Estructura de Computadores, GitHub cumple tres funciones clave:

- Almacenamiento seguro: cada script en ensamblador (.asm) se guarda en la nube, evitando pérdidas por fallos locales.
- Transparencia y presentación: los archivos quedan disponibles para que el docente pueda acceder directamente, verificando la nomenclatura y ejecución del código.
- Práctica profesional: usar GitHub acerca al estudiante a las metodologías modernas de desarrollo de software, ya que es una herramienta estándar en la industria.

2.5.3 Relación con este laboratorio

En este desarrollo procedí a crear el siguiente repositorio de nombre:

- **Laboratorio1_EstructuraComputadores**

Ahora de mi parte en la creación de este repositorio se procedió a generar una breve descripción, que en este caso es la siguiente “Repositorio con los programas en ensamblador del Laboratorio #1 de Estructura de Computadores (MIPS – MARS)”, para una mayor claridad de este proyecto en GitHub.

Luego se procedió de mi parte a cargar los tres programas desarrollados en MIPS:

- DeMendozaTovarAlejandro_MayorNúmero
- DeMendozaTovarAlejandro_Menor
- DeMendozaTovarAlejandro_Fibonacci

Ahora, el código fuente de los tres scripts desarrollados en este laboratorio se encuentra disponible en el siguiente link del repositorio:

https://github.com/AlejoTechEngineer/Laboratorio1_EstructuraComputadores.git

De esta manera puedo indicar que GitHub no solo es el medio de entrega, sino también una oportunidad para aprender buenas prácticas en la gestión de código y preparación de proyectos.

2.5.4 Inclusión de archivo README

Adicional a la subida de los tres programas en ensamblador:

- DeMendozaTovarAlejandro_Mayor.asm
- DeMendozaTovarAlejandro_Menor.asm
- DeMendozaTovarAlejandro_Fibonacci.asm)

Se creó un archivo **README.md** en el repositorio de GitHub.

Este archivo tiene como finalidad documentar el propósito del laboratorio, describir el contenido del repositorio y explicar de manera breve el procedimiento de ejecución de los scripts en el simulador **MARS (MIPS Assembler and Runtime Simulator)**.

La inclusión del README.md contribuye a la organización y presentación del repositorio, brindando a cualquier usuario una guía clara sobre los objetivos de la práctica, el uso de los programas y la estructura del proyecto.

Y con este último punto culmino el desarrollo de este laboratorio, muchas gracias, respetado profesor por sus valiosos conocimientos que van a ser de gran utilidad en mi futuro como ingeniero informático.

3 CONCLUSIONES

A lo largo del desarrollo de este laboratorio logré obtener un aprendizaje integral en torno a la programación en ensamblador y su relación con la arquitectura de computadores. En primer lugar, el uso del simulador **MARS (MIPS Assembler and Runtime Simulator)** me permitió comprender de manera práctica la relación entre el código ensamblador y el flujo de ejecución en un procesador escalar segmentado. El hecho de poder observar directamente el contenido de los registros, los cambios en memoria y el resultado inmediato de cada instrucción me ayudó a visualizar con mayor claridad cómo las operaciones básicas influyen en el comportamiento del hardware.

En segundo lugar, al implementar algoritmos como la búsqueda del número máximo, el número mínimo y la generación de la serie de Fibonacci, entendí que incluso tareas aparentemente simples requieren un control minucioso de registros, saltos condicionales y estructuras de control en bajo nivel. Esta experiencia reforzó mi lógica de programación y me permitió valorar la precisión que se debe tener al programar en un lenguaje cercano al hardware.

Asimismo, la necesidad de documentar cada línea de código y de seguir la nomenclatura propuesta me enseñó la importancia de la disciplina en la programación en ensamblador. Además, al cargar los programas en **GitHub** y crear un archivo README.md, pude aplicar buenas prácticas de documentación, organización y trazabilidad del trabajo académico. Considero que este paso no solo facilitó la revisión del laboratorio, sino que también me permitió practicar el uso de herramientas colaborativas que resultan fundamentales en el ámbito profesional.

Finalmente, reconozco que esta actividad fortaleció mis competencias técnicas y amplió mi comprensión sobre los fundamentos de la arquitectura de computadores. Comprobé que la programación en bajo nivel demanda un pensamiento estructurado y una atención al detalle mucho mayor que la que se requiere en lenguajes de alto nivel, lo cual representa una habilidad esencial para entender cómo funcionan los sistemas de cómputo modernos.

En conclusión, este laboratorio no solo me permitió poner en práctica conceptos teóricos, sino que también me impulsó a desarrollar habilidades transversales en documentación, control de versiones y simulación. Gracias a esta experiencia consolidé un aprendizaje integral en el área de estructura de computadores, que sin duda servirá como base para futuros retos académicos y profesionales.

4 BIBLIOGRAFÍA

Respetado profesor Ing. Deivis Eduard a continuación, la bibliografía implementada:

- ✓ Tema 1. Fundamentos del diseño y evolución de los computadores. Tema 2. Evaluación de prestaciones de un computador. Tema 3. Aprovechamiento de la jerarquía de memoria. Tema 4. Almacenamiento y otros aspectos de entrada/salida. Tema 5. Procesadores segmentados.
- ✓ Clases virtuales con el profesor Ing. Deivis Eduard Ramírez Martínez.
- ✓ Material de estudio del aula.
- ✓ Bryant, R., & O'Hallaron, D. (2016). Computer Systems: A Programmer's Perspective (3.ª ed.). Pearson.
- ✓ Hennessy, J. L., & Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6.ª ed.). Morgan Kaufmann.
- ✓ Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design MIPS Edition: The Hardware/Software Interface (5.ª ed.). Morgan Kaufmann.
- ✓ Mars4_5. (s.f.). MARS: MIPS Assembler and Runtime Simulator. Missouri State University. Recuperado de: http://courses.missouristate.edu/kenvollmar/mars/
- ✓ Stallings, W. (2019). Computer Organization and Architecture: Designing for Performance (11.ª ed.). Pearson.
- ✓ Tanenbaum, A. S., & Austin, T. (2012). Structured Computer Organization (6.ª ed.). Pearson.
- ✓ GitHub. (s.f.). GitHub Docs: Getting started with GitHub. Recuperado de: https://docs.github.com/