

Informe Práctica de Laboratorio 1

José Ángel Masías
Edael Duque

Naguanagua, 24 de enero de 2026

República Bolivariana De Venezuela
Ministerio Del Poder Popular Para la Educación
Universidad De Carabobo
Facultad Experimental De Ciencias Y Tecnología
Departamento De Computación

Profesor:
José Canache

Estudiantes:
José Ángel Masías (C.I. 30.281.906)
Edael Duque (C.I. 30.800.814)

Naguanagua, 24 de enero de 2026

INFORME

1. ¿CÓMO SE IMPLEMENTA LA RECURSIVIDAD EN MIPS32? ¿QUÉ PAPEL CUMPLE LA PILA (\$SP)?

Cuando implementamos la técnica de la recursividad en Mips32 debemos tomar en cuenta como programador, diferencias con un lenguaje de alto nivel, donde una misma función se vuelve a llamar así misma, el lenguaje ensamblador requiere un proceso diferente. El programador debe simular el comportamiento de la recursividad, pero se puede observar una problemática, un número limitado de registros para guardar la información por cada recursión, a favor tenemos una medida que nos ayudará a solucionar este problema, la pila o también llamada Stack. En ella podemos guardar temporalmente los datos que necesitemos mientras se desarrolla y se retorna la función recursiva, sin tener la limitante de los registros. Para poder hacer uso de este método se necesita el registro \$sp, llamado Stack Pointer. Es un registro que funciona como puntero, marca en dónde termina la pila en memoria, la pila va a crecer hacia direcciones de memoria menores, se pueden reservar espacios para los registros que se necesiten, cada espacio requiere 4 bytes, se resta al puntero para conseguir el espacio para los registros y aquí es donde entra el comando de la pila (\$sp), debe tener un coeficiente que le permite determinar al computador a que parte de la pila vamos a hacer uso. Para obtener el espacio al que apunta \$sp usamos: 0(\$sp), y si se quiere obtener espacios superiores están 4, 8, 12, 16, etc., múltiplos de 4. Para el caso actual, la paridad recursiva funciona para reservar dos registros, \$ra que guarda la dirección de retorno y \$a0 que guarda el valor n; en cada llamada los valores cambian y estos son almacenados en los diferentes espacios que se van reservando en la pila. En el algoritmo en cuestión tenemos la línea (*addi \$sp, \$sp, -8*) que es una suma inmediata para garantizar dos espacios en la pila, y como ya determinamos se resta -8 al puntero para que habilite los espacios. Seguidamente tenemos las líneas (*addi \$sp, \$sp, 8*) que se encarga de regresar al punto de inicio al apuntador a su posición inicial en cada recursión, para que al final de la recursión se devolvió al apuntador los mismos espacios que tuvo que ceder, esto permite que no se acumulen espacios basura en la pila, y se evita un desbordamiento.

2. ¿QUÉ RIESGOS DE DESBORDAMIENTO EXISTEN? ¿CÓMO MITIGARLOS?

Mitigar los riesgos de los desbordamientos en MIPS32 es una prioridad y una tarea crítica a la hora de la gestión de la memoria manual. En contraste con los lenguajes modernos, que estos poseen recolectores de basura o límites automatizados, en lenguaje ensamblador tú eres el encargado del espacio de direcciones. A continuación, describiremos algunas técnicas que pueden evitar o repeler este problema, abordaremos tres enfoques:

- **Una buena gestión del marco de pila:** Simetría en las instrucciones. Este apartado ya se asomó en la respuesta anterior, pero se explicará con más detalle. Cada resta al puntero \$sp en el prólogo de la recursión, ejemplo (*addi \$sp, \$sp, -un número*), siempre deberá tener una suma idéntica que devuelva todos los valores en el epílogo (*addi \$sp, \$sp, un número*), con esto se garantiza que la pila borre datos que ya no serán usados y que se acumulen como datos basura. Puntos de Salida Únicos. Para tener control sobre la limpieza de la pila, se debe garantizar la simetría de las instrucciones usando el comando de salto *jr* con el apuntador

\$ra para retornar efectivamente. Minimización de Registro. Solo usar los registros necesarios para guardar en la pila para ahorrar los espacios de la misma.

- **Guardado y límites de la pila:** Se pueden establecer defensas en formato de código que nos ayuden a evitar el desbordamiento, antes de que se corrompan los datos estáticos, por ejemplo, que los registros puedan tener un número máximo de llamadas, con el apoyo de un contador monitorear este límite y cuando este sea rebasado, forzar el programa a generar un error, evitando el desbordamiento. Otra aplicación que se puede utilizar es comparar el apuntador \$sp con su límite en el segmentado de la pila, si el apuntador se acerca mucho al techo de datos que se fuerce terminar con las operaciones.
- **Fórmulas anti desbordamiento:** existen fórmulas para calcular si un algoritmo recursivo puede generar un desbordamiento, de ser el caso, debe replantearse hacer ese algoritmo con un enfoque iterativo.

3. ¿QUÉ DIFERENCIA ENCONTRASTE ENTRE UNA IMPLEMENTACIÓN ITERATIVA Y UNA RECURSIVA EN CUANTO AL USO DE MEMORIA Y REGISTROS?

El enfoque iterativo necesita cinco registros para poder funcionar y el recursivo requiere de al menos seis registros mínimos, es cierto que se usan menos registros de la línea \$t y eso puede ser positivo, sin embargo, se necesitan registros como \$ra para guardar la dirección de retorno y el puntero de la pila \$sp también es necesario. Entonces podemos deducir que el enfoque iterativo es más eficiente en cuanto registros se refiere.

En el apartado de la memoria podemos ver que el enfoque iterativo tiene una ventaja notable, porque el número de n no es importante, ya que mantiene un uso de memoria constante, debido a que siempre usa los mismos registros como memoria, es de $O(1)$ para una función de Espacio que depende de n ($E(n)$). Para el enfoque recursivo requiere de más memoria, específicamente es de $O(n)$ para una función $E(n)$, ya que según qué tan grande sea el n , afectará de manera lineal el espacio ocupado, porque en cada recursión se realizará una llamada que guardará un espacio en la pila, así que el espacio ocupado dependerá de cuan grande sea n .

4. ¿QUÉ DIFERENCIAS ENCONTRASTE ENTRE LOS EJEMPLOS ACADÉMICOS DEL LIBRO Y UN EJERCICIO COMPLETO Y OPERATIVO EN MIPS32?

La principal diferencia del libro con respecto a los ejercicios completos radica en que los ejemplos tratados en el libro tienen un enfoque especializado en la instrucción que se aborda en cada sección del capítulo, por lo tanto, no se aborda un problema en específico en donde los ejemplos puedan ser utilizados, sino que están planteados para un entendimiento general sobre el uso de esa instrucción. Por supuesto como cada sección tiene como objetivo hablar sobre limitadas instrucciones, no se ven algoritmos completos que se usan para resolver un problema. El libro aborda los engranajes que conforman un reloj, pero no analiza el reloj como un gran engranaje. El escrito funciona para darte orientaciones generales, para ti, como programador poder guiarte y aplicar estas

reglas generales a casos específicos, problemas en particular que tengas que resolver, agregando todas las particularidades de un problema en cuestión, gustos y decisiones del programador. Porque en un problema tal vez debas requerir de las instrucciones enseñadas de varias secciones y no problemas que se resuelvan de forma segmentada como las presenta en el libro.

5. ELABORAR UN TUTORIAL DE EJECUCIÓN PASO A PASO EN MARS:

MARS, abreviatura de *MIPS Assembler and Runtime Simulator*, es un programa de entorno integrado, de ligero tamaño que está basado en Java. Su función es servir como simulador, editor de programas basados en el lenguaje ensamblador de MIPS de 32 bits. Es utilizada como herramienta de desarrollo y didáctica. A continuación, vamos a describir un ligero tutorial de varios pasos sencillos para la correcta ejecución de un código que quieras ejecutar, opción básica y fundamental para el MARS:

- **Paso 1 (Abrir el archivo):** Dirígete a la barra superior del editor y selecciona la opción "File", luego selecciona .open.° Ctrl + O, seguidamente navega por el buscador de carpetas hasta encontrar tu archivo ".asm".^{en} el lugar designado por ti. Si es tu primera vez en MARS y no tienes ningún archivo ".asm", entonces en lugar de seleccionar la opción .open", selecciona la opción "New.° Ctrl + N, y eso te permitirá desde el mismo editor, crear un archivo nuevo, navega por el buscador de archivos y elige la ruta de acceso.
- **Paso 2 (Ensamblar):** Si tu código está listo para funcionar, para correrlo usa F3 o seleccionar en la barra superior, en el segundo espacio al lado de "File", la opción Runτ entrarás en una nueva pestaña donde podrás ver en tiempo real, línea por línea, la corrida en frío del algoritmo (text segment), el monitoreo de la pila (data segment), los valores que toman los registros que son usados en el algoritmo durante su ejecución (registers) y también la salida por pantalla.
- **Paso 3 (Interfaz de corrida):** En la segunda barra superior podrás observar varias opciones, una opción para correr todo el código de una vez (run the current program), hasta que termine el código o hasta que requiera una entrada por el usuario. La segunda opción es para correr el algoritmo línea por línea (single step), otro para hacer retroceder los pasos del algoritmo línea por línea (last step). Otra opción para congelar la corrida (breakpoint) y otra opción para detener el algoritmo (stop) y reiniciarlo (reset).

Con esto tenemos las instrucciones básicas para poder correr un código en lenguaje ensamblador en el simulador especializado y editor de código en Mips32, MARS.

6. JUSTIFICAR LA ELECCIÓN DEL ENFOQUE (ITERATIVO O RECURSIVO) SEGÚN EFICIENCIA Y CLARIDAD EN MIPS.

El mejor enfoque es el iterativo, manifestamos los siguientes argumentos a continuación: Como primer argumento tenemos un código de MIPS32 mucho más corto, sencillo y comprensible, que abarca menos líneas que su contraparte, facilitando su entendimiento y uso. A diferencia del enfoque recursivo que contiene una mayor cantidad de líneas y un nivel de abstracción y complejidad superior, su implementación es más complicada. Como segundo argumento tenemos que la

complejidad de memoria del enfoque iterativo es $O(1)$ ya que el algoritmo usa algunos registros para almacenar enteros, sin importar qué tan grande sean. En contraparte con el enfoque recursivo, tenemos una complejidad de memoria de $O(n)$ o complejidad lineal, debido a que cada llamada reserva un espacio en la pila, entonces para cada llamada a la función de paridad se usa n por 8 bytes de memoria; agregando que con un valor de n muy alto se corre el riesgo del ya mencionado riesgo de desbordamiento por Stack Overflow. Como tercer argumento tenemos la eficiencia con respecto al tiempo, el enfoque iterativo es eficiente, porque solo ejecuta instrucciones aritméticas simples como *addi* y *sub*, y saltos simples *j* y *blez* que no requieren tanto costo en ciclos de reloj. Al contrario del enfoque iterativo, el enfoque recursivo es un algoritmo menos eficiente, debido a que existe una sobrecarga para el procesador por múltiples instrucciones *lw* y *sw* para salvar y restaurar en cada recursión; sumando al rendimiento los saltos *jal* y *jr* que son más costosos en términos de ciclo de reloj en comparación con saltos simples. Para concluir podemos decir que al menos en este caso el enfoque recursivo es un "lujo computacional" es preferible usar el enfoque iterativo de la función paridad.

7. ANÁLISIS Y DISCUSIÓN DE LOS RESULTADOS.

En esta sección del informe nos dedicaremos a analizar entradas que pueden aportar bastante en el ámbito de los resultados y desempeño del algoritmo. Vamos a generar cuatro entradas diferentes para ambos algoritmos (paridad recursiva y paridad iterativa) y veremos cómo se desenvuelven en el simulador. Estas cuatro entradas, cada una representante de un comportamiento notable en el algoritmo, descritos a continuación: un número impar, un número par, un número negativo y un número muy grande:

- **Número impar:** tras ingresar como entrada el número trece, número impar, determinamos que ambos algoritmos se desempeñan con normalidad, dando como resultado uno, confirmando que el número trece es impar.
- **Número par:** Tras ingresar como entrada el número doce, aseguramos que ambos algoritmos se comportan de forma similar, sin ningún cambio en particular, ambos algoritmos devuelven como salida cero, en efecto, para la función de paridad confirma que doce es un número par.
- **Número negativo:** Para ambos casos el algoritmo tiene unas líneas de código que funcionan como forma de validación para evitar entradas erróneas en el algoritmo que pueda derivar en resultados más erráticos, se valida la entrada en los algoritmos verificando que el número sea positivo. El algoritmo no avanzará hasta que reciba un número positivo, de forma cíclica mandará la instrucción de ingresar un número positivo hasta que la entrada sea la correcta.
- **Número muy grande:** Para el número grande, tomamos como entrada para el algoritmo el número 1234567, y podemos observar diferencias en el desempeño de la paridad recursiva y la paridad iterativa. A diferencia de las entradas anteriores en donde no hay diferencias en sus salidas y rendimiento, aquí podemos ver que el resultado fue diferente. Cuando probamos el número grande en la paridad iterativa, a pesar de tardar más que los casos anteriores, arroja uno (impar), pero cuando probamos en la paridad recursiva vemos que el programa arroja "ejecución terminada con error", podemos apreciar un error por desbordamiento de memoria, un clásico Stack Overflow, aquí se evidencia la eficiencia del enfoque iterativo porque

este a pesar del número tan grande, puede devolver un resultado válido. El desbordamiento del enfoque recursivo se da ya que un número tan grande como 1234567, producirá la misma cantidad de marcos de pila antes de poder resolver el primer cálculo, eso es demasiado en memoria RAM. En simuladores como MARS la memoria que se reserva para la pila no suele pasar de unos pocos megas, y este supera ese límite. Lo anterior genera que MARS aborte la ejecución y devuelva como salida predeterminada un error en la pantalla. Queda en evidencia empírica que el enfoque iterativo en este algoritmo es aplastantemente superior al enfoque recursivo de la paridad.