Arquitectura del Computador
Práctica de Laboratorio 1

Angel Navas y Jose Natera
Departamento de Computación
Universidad de Carabobo
Barbula, Carabobo

***Abstract*—Familiarizar al estudiante con la sintaxis del lenguaje ensamblador *MIPS32* y su ejecución en el entorno MARS, resolviendo problemas clásicos de programación.**

***Index Terms*—Computer Architecture, Assembly Language, Recursion, Stack management, Work Environment**

I. Informe

A. ¿Cómo se implementa la recursividad en MIPS32?

En *MIPS32*, la recursividad no constituye una estructura de control nativa, sino que se implementa utilizando instrucciones de bifurcación con enlace y la gestión explícita del Contexto de Ejecución.

- Instrucción `jal` : Actúa como el mecanismo de transferencia de control. Al ejecutar `jal Paridad`, el hardware almacena el Program Counter (PC) + 4 en el registro de enlace `$ra` de forma atómica antes de desviar el flujo hacia la etiqueta de destino
- Condición de Parada : Para evitar un bucle infinito, el código utiliza la instrucción `bne $a0, 0, else`. Si el número llega a cero, la función deja de llamarse a sí misma y comienza a retornar (`jr $ra`).
- Alternancia de Resultados: La estructura recursiva se emplea para realizar el recorrido descendente del árbol de llamadas hasta el caso base. Durante la fase de retorno, se aplica la operación lógica XOR inmediata `xori $v0, $v0, 1` la cual procesa la inversión del bit de paridad de forma sucesiva en cada marco de pila restaurado

B. ¿Qué papel cumple la pila `$sp`?

Debido a que la arquitectura MIPS32 dispone de un único registro de dirección de retorno `$ra`, cada ejecución de la instrucción `$jal`, sobrescribe de forma destructiva el valor previo. En consecuencia, el uso del Stack es necesario para mantener la integridad de la cadena de llamadas

- Persistencia del Contexto de Ejecución: Antes de realizar una llamada recursiva , se debe implementar un Prólogo de Función mediante la macro `save_in_pila` (`$ra`). Esto guarda la dirección de retorno actual hacia la memoria, evitando la pérdida del punto de retorno hacia el llamante.
- Control del Flujo de Retorno: Sin la pila, el programa entraría en un ciclo infinito en la instrucción `jr $ra`, ya que `$ra` siempre apuntaría a la misma línea dentro de la función Paridad, provocando la pérdida del puntero de retorno hacia la función main.

C. ¿Qué riesgos de desbordamiento existen? ¿Cómo mitigarlos?

a) *Riesgos y peligros:* El riesgo principal en esta implementación reside en la saturación de la memoria destinado a la pila (`$sp`). Debido a la naturaleza de la recursión , el consumo de recursos es de complejidad espacial.

- Consumo de Memoria por Llamada: Cada vez que el programa ejecuta `jal Paridad`, la macro `save_in_pila()` resta 4 bytes al puntero de pila para almacenar el registro `$ra`. El uso de la pila es directamente proporcional al valor de n. Si un usuario ingresa un número suficientemente grande (por ejemplo, $n = 1\,050\,000$), el puntero `$sp` seguirá bajando hasta colisionar con el segmento de datos o exceder los límites de memoria asignados por el simulador. Por lo tanto, el programa terminará abruptamente con un error de segmentación o una excepción de "address out of range".
- Limitaciones de Registro (Integer Overflow): Aunque el registro de propósito general `$v0` soporta una capacidad de hasta $2^{31} - 1$. la arquitectura colapsará por desbordamiento de pila órdenes de magnitud antes de alcanzar el límite superior del registro.

b) *Estrategias de Mitigación:* Para transformar este código en una solución profesional, se proponen las siguientes técnicas:

- Usar el enfoque Iterativo

Sustituir la estructura recursiva por un bucle en ensamblador, eliminando la dependencia del registro `$ra` reduciendo la complejidad espacial a $O(1)$.

- Optimización por Operadores de Bits

En lugar de restar 1 recursivamente, se puede determinar la paridad analizando el Bit Menos Significativo. Si el primer bit es 1, el número es impar; si es 0, es par. Esto reduce la complejidad de $O(n)$ a $O(1)$:

```
andi $v0, $a0, 1 # Si $v0 es 0, es par.
Si es 1, impar
```

.

D. ¿Qué diferencias encontraste entre una implementación iterativa y una recursiva en cuanto al uso de memoria y registros? Uso de la Memoria (Segmento de Pila).

La diferencia entre ambas implementaciones radica en la gestión del Stack Segment y la eficiencia en el ciclo de

instrucción del procesador: Implementación Recursiva: El código actual presenta un uso de memoria de clase $O(n)$. Cada llamada a Paridad reserva 4 bytes para almacenar el registro `$ra`. Si se ingresa el número 1 000, la ocupación en la memoria RAM asciende a 4000 bytes, incrementando linealmente el riesgo de Stack Overflow por colisión con otros segmentos de memoria. Implementación Iterativa: El uso de memoria es de clase $O(1)$. Un bucle no requiere guardar direcciones de retorno de forma sucesiva, por lo que el puntero de pila permanece estático. El consumo de *RAM* es cero, independientemente de qué tan grande sea el número ingresado. Acceso a Memoria: La implementación recursiva es intrínsecamente más lenta debido a las instrucciones `$sw` y `$lw`. El acceso a la memoria *RAM* es considerablemente más costoso en ciclos de reloj que el acceso a registros. Sobrecarga de Control: Cada instrucción de salto propia introduce una ruptura en el flujo secuencial de instrucciones. Dependiendo de la implementación de la unidad de control, esto puede generar stalls ("estancamiento") en el pipeline del procesador, mientras que el bucle iterativo mantiene un flujo más predecible.

E. ¿Qué diferencias encontraste entre los ejemplos académicos del libro y un ejercicio completo y operativo en MIPS32?

Principalmente, el enfoque, los ejercicios del libro estaban orientados a introducir al estudiante a la naturaleza de un lenguaje ensamblador como lo es *MIPS32*, dando a conocer sus instrucciones básicas y enfoncandose en el como utilizarlas, mientras que un ejercicio completo y operativo puede buscar, basandose en este conocimiento adquirido de los ejercicios del libro, exigir una tarea que requiere de un buen dominio y entendimiento de estos conceptos, una analogia para entender esta diferencia es ver a los ejercicios del libro como los ejemplos de un manual de instrucciones de alguna una nueva herramienta o instrumento, mientras que un ejercicio completo puede ser toda una tarea que para ser resuelta, requiere un buen uso de esos instrumentos.

F. Tutorial de Ejecución Paso a Paso en MARS

a) *Paso 1 - Ensamblaje y Expansión de Directivas:* Antes de la ejecución, el código fuente debe someterse a un proceso de ensamblaje para ser traducido a lenguaje máquina.

1) Cargar el archivo: Se importa el archivo *.asm* en el entorno de desarrollo integrado (IDE) de MARS.
2) Proceso de Ensamblaje: Se presiona el ícono de la llave inglesa y el martillo para iniciar (F3) .
   - Resultado: Si la sintaxis es correcta, MARS cambiara del modo Edit al modo Execute.
   - En la ventana Text Segment, se puede ver cómo las pseudo-instrucciones y macros se expanden en instrucciones atómicas de la arquitectura *MIPS32*

b) *Paso 2 - Configuración del Entorno de Simulación:*
En la pestaña Execute, se deben localizar tres áreas:

- Registers: A la derecha, para observar los cambios en `$v0`, `$a0`, `$ra` y `$sp`.
- Data Segment: En la parte inferior, para ver cómo se guardan las direcciones en la pila.
- Run Speed: Se recomienda configurar la frecuencia de reloj simulada a 2-5 Instrucciones por segundo para realizar una trazabilidad efectiva de la recursión.

c) *Paso 3 - Ejecución Inicial y Entrada de Datos:*

1) Ejecución paso a paso: Se utiliza el icono con el número 1 o la tecla F7.
2) Llamada al sistema (Syscall 5): El programa se detendrá en la consola de la parte inferior. El usuario debe ingresar un número y presionar Enter. El valor aparecerá en el registro `$v0`.

d) *Paso 4 - Llamada a la función:*
Este es el punto crítico de la ejecución. Al procesar la instrucción `jal Paridad`:

1) Salto y Enlace: Al presionar Step, se observará que el registro `$ra` cambia a la dirección de la instrucción siguiente al `jal`.

Aquí hay un cambio importante entre la versión iterativa y la versión recursiva

- Versión Iterativa

1) Caso base: Si el número es negativo o igual a cero, retorna -1 (error) o 1 (0 es par) respectivamente
2) Entrada al ciclo for:
   - El registro `$t0` hace de iterador desde $i = 0$ hasta $i = n - 1$ aumentando de 1 en 1
   - El registro `$v0` (el valor de retorno) va alternando entre 0 y 1 en cada iteración

- Versión Recursiva

1) Crecimiento de la Pila: Al entrar en la macro `save_in_pila`:
   - El registro `$sp` (Stack Pointer) disminuye en 4
   - En el Data Segment, en la dirección indicada por `$sp`, aparecerá el valor guardado de `$ra`.
2) Llamadas anidadas: Si se ingresó el número 3, este proceso se repetirá 3 veces. Se verá cómo la pila se llena de direcciones de retorno idénticas.

e) *Paso 5 - La Salida:*

- En la versión iterativa, se termina con el ciclo *for*:

Cuando `$t0` llega a $i = n - 1$, pasa al directamente al retorno usando `jr $ra` para regresar al donde fue llamado (el *caller*), al no haber manipulado los valores de entrada ni haber hecho llamadas a otras funciones o procedimientos, no maneja la pila recursiva.

- En la versión recursiva, se hace "El *Unwinding*":

Cuando `$a0` llega a 0, el programa llega al `jr $ra` del caso base.

1) Retorno: El programa salta a la instrucción `addi $a0, $a0, 1` dentro de `else`.
2) Operación Lógica: Se observa cómo `$v0` cambia de 0 a 1 (o viceversa) con la instrucción `xori`.
3) Limpieza de Pila: La macro `Load_in_pila` recuperará el valor de la memoria hacia `$ra` y aumentará el `$sp`, "liberando" ese espacio.

f) *Paso 6 - Finalización:*
El flujo regresará al main. Dependiendo del valor final de `$v0` (0 o 1), el simulador mostrara el mensaje correspondiente y finalmente, la instrucción `li $v0, 10` cerrará el simulador de forma segura.

G. *Justificar la elección del enfoque (iterativo o recursivo) según eficiencia y claridad en* MIPS32

1) La Recursividad: El Enfoque Académico

Se elige principalmente con fines educativos o cuando la estructura de datos es intrínsecamente recursiva (como lo podrian ser los arboles) a pesar de sus desventajas en el rendimiento de hardware.

1) La Iteración: El Enfoque Seguro

Se elige para optimizar el uso de recursos y garantizar la estabilidad del sistema. Es más eficiente que la recursividad. Al prescindir de las operaciones de lectura/escritura en el segmento de pila, se minimiza la latencia de memoria. El consumo de memoria es constante $O(1)$.

1) La Versión Directa (*AND*): La Excelencia en Bajo Nivel

La forma más óptima de resolver la paridad en *MIPS32* no es ni recursiva ni iterativa, sino aritmética/lógica mediante la instrucción `andi $v0, $a0, 1`.

1) Complejidad Constante $O(1)$: Mientras que la recursión e iteración tardan más tiempo si el número es más grande, el andi tarda exactamente un ciclo de reloj, sin importar si el número es 1 o 2 000 000 000.
2) Aprovechamiento de la arquitectura: Los números en binario tienen una propiedad única: el bit menos significativo (LSB) determina la paridad.
   - Si termina en …0, es par.
   - Si termina en …1, es impar.
3) Ahorro energético y de hardware: No requiere saltos ni accesos a memoria. Es una sola operación lógica que ocurre directamente en la *ALU* .
4) Resumen Comparativo

Criterio	**Recursivo**	**Iterativo**	**Directo (*AND*)**
Tiempo (*CPU*)	Muy lento ($O(n)$ + accesos a la *RAM*)	Rápido ($O(n)$ en registros)	Instantáneo ($O(1)$)
Espacio (*RAM*)	Alto ($O(n)$ en la pila)	Constante ($O(1)$)	Constante ($O(1)$)
Riesgo	Alto (Posibilidad de *Stack Overflow*)	Bajo	Ninguno

H. *Análisis y Discusión de los Resultados*

- Puntos críticos: Cada instrucción sw y lw dentro de la recursión representa un acceso al bus de datos. Esto genera latencia comparado con una operación que ocurre puramente dentro de la *ALU*.
- Estado del Registro `$v0`: En el modelo recursivo, es notable cómo el valor de retorno se "propaga" mediante el desapilado. La operación xori actúa como un inversor lógico que, tras n llamadas, deja el bit de estado en la posición correcta para ser evaluado por el main.
- Flujo Optimizadamente Local : La variante iterativa elimina el flujo intenso hacia la memoria principal. Al mantener el contador y el estado de paridad dentro del Register File, el procesador opera a su máxima frecuencia sin depender de la latencia de la memoria *RAM*.

Estabilidad y Escalabilidad Al discutir los resultados, se identifican dos escenarios opuestos:

- Escenario de Éxito: Con valores reducidos (ej. n < 10,000), el programa es estable. Sin embargo, su escalabilidad está limitada por el hardware. Mientras que la version iterativa mantiene una complejidad $O(1)$. Un enfoque iterativo permite procesar números masivos sin riesgo de Stack Overflow, garantizando la integridad del sistema ante cualquier valor de entrada.
- Escenario de Fallo: A medida que n aumenta, el tiempo de ejecución crece de manera lineal ($O(n)$). El riesgo de desbordamiento de pila es la mayor debilidad del diseño actual, ya que la memoria física asignada al segmento de la pila no es infinita.

¿Es la solución óptima? Aunque los resultados son funcionalmente correctos, concluimos que el enfoque recursivo es ineficiente para este problema específico , el iterativo es la mejor solucion si no se toma en cuenta la solucion `andi $v0, $a0, 1`