Torre de Hanói

Nombres:

Santiago Josué García Morales 750912

Bryan Alejandro Villalpando Castillo 751772

Materia: Organización y Arquitectura De Computadoras

Índice

Pág.

Introducción

El problema de las Torres de Hanói es un clásico en la computación que representa un excelente caso de estudio para aplicar técnicas de recursividad. Su simplicidad en las reglas contrasta con la complejidad computacional que implica su resolución para un número creciente de discos. En este proyecto, se implementó una solución recursiva del problema utilizando lenguaje ensamblador RISC-V, ejecutada y simulada en el entorno RARS (RISC-V Assembler and Runtime Simulator).

Objetivo

El objetivo principal es demostrar que estructuras lógicas como la recursión pueden ser implementadas de forma eficiente incluso en un lenguaje de bajo nivel como el ensamblador. Para ello, se desarrolló un programa que permite resolver el problema para N discos, donde N es configurable mediante una constante inicializada en un registro. Además de la resolución algorítmica, se analiza el comportamiento de la pila en tiempo de ejecución, especialmente al observar la ejecución con 3 discos mediante capturas del segmento de datos. También se estudia el rendimiento en términos de conteo de instrucciones (IC), clasificando los tipos de instrucciones utilizadas y cómo estas escalan conforme aumenta el número de discos.

Código de Torre de Hanói

```
.text
main:
    addi s0, zero, 3       # N = 3 Numero de discos


    lui  s1, 0x10010        # Dirección base de la torre A
    addi s1, s1, 0          # Torre A: 0x10010000
    addi s2, s1, 4          # Torre B: 0x10010004
    addi s3, s2, 4          # Torre C: 0x10010008

    slli t0, s0, 5          # Calcula el desplazamiento de memoria
    add  s2, s2, t0         # Ajusta la direccion de la Torre B
    add  s3, s3, t0         # Ajusta la direccion de la Torre C

    addi t2, zero, 0x00     # Inicializacion de variables auxiliares
    addi s10, zero, 0       # Contador de movimientos
    addi s11, zero, 0x20    # Espaciado entre discos en memoria
    addi t1, s0, -1         # n = N - 1 su uso es para las iteraciones
    addi t0, zero, 0        # Contador
    addi t5, zero, 0        # Desplazamiento

# Este es un bucle que inicializa la Torre A
for:
    beq s0, t0, torre_hanoi   # Si t0 == s0, inicia la recursion de Torre de Hani
    addi t1, t0, 1            # Incrementa el indice del disco
    add  t2, zero, s1         # Direccion base de la Torre A
    add  t4, t2, t5           # Calcula la direccion especifica del disco
    sw   t1, 0(t4)            # Guarda el disco en memoria
    addi t5, t5, 0x20         # Avanza en la memoria
    addi t0, t0, 1            # Incrementa el contador
    j    for                  # Se repite el bucle

# Esto realiza la llamada a la función recursiva de la Torre de Hanoi
jal ra, torre_hanoi
j exit

# Funcion recursiva de Torre de Hanoi
torre_hanoi:
    addi t0, zero, 1
```

```
    # Condicion en la que si hay solo un disco,se movera directamente
if:
    bne  s0, t0, else
    sw   zero, 0x0(s1)      # Remueve el disco de la torre de origen
    add  s1, s1, s11        # Ajusta puntero de torre A
    sub  s3, s3, s11        # Ajusta puntero de torre C
    sw   s0, 0(s3)          # Mueve el disco a la torre destino
    addi s10, s10, 1        # Incrementa el contador de movimientos
    jalr ra                 # Realiza el retorno

# En esto es la parte recursiva
else:
    addi sp, sp, -4
    sw   ra, 0x0(sp)        # Guarda el valor de "ra" en la pila
    addi sp, sp, -4
    sw   s0, 0x0(sp)        # Guarda s0 en la pila
    addi s0, s0, -1         # Reduce el numero de discos

 # Intercambia torres para la primera llamada recursiva
    add  t1, s2, zero       # Guarda torre intermedia temporalmente
    add  s2, s3, zero       # Torre intermedia toma el valor de torre destino
    add  s3, t1, zero       # Torre destino toma el valor original de torre intermedia
    jal  ra, torre_hanoi    # Llamada recursiva

    # Restaura el orden original de las torres despues de la primera llamada
    add t1, s2, zero
    add s2, s3, zero
    add s3, t1, zero

    # Recupera los valores de pila para continuar
    lw s0, 0x0(sp)
    addi sp, sp, 4
    lw ra, 0x0(sp)
    addi sp, sp, 4

    # Mueve el disco mas grande directamente desde torre origen a torre destino
    sw zero, 0x0(s1)        # Elimina el disco de la torre de origen
    add s1, s1, s11         # Ajusta puntero de torre origen
    sub s3, s3, s11         # Ajusta puntero de torre destino
    sw s0, 0x0(s3)          # Mueve el disco
    addi s10, s10, 1        # Incrementa el contador de movimientos

    # Segunda llamada recursiva: mueve los discos restantes al destino final
    addi sp, sp, -4
    sw ra, 0x0(sp)          # Guarda ra
    addi sp, sp, -4
    sw s0, 0x0(sp)          # Guarda s0
    addi s0, s0, -1         # Decrementa numero de discos
```

```
    # Intercambia torres para preparar segunda llamada recursiva
    add t1, s1, zero
    add s1, s2, zero
    add s2, t1, zero

    jal ra, torre_hanoi     # Llamada recursiva

    # Restaura el orden original de las torres despues de la segunda llamada
    add t1, s1, zero
    add s1, s2, zero
    add s2, t1, zero

    # Recupera registros
    lw s0, 0x0(sp)
    addi sp, sp, 4
    lw ra, 0x0(sp)
    addi sp, sp, 4

    jalr ra                 # Retorna

# Fin del programa
exit:
    j exit
```

Explicación del código

```
    .text
    main:
        addi s0, zero, 3      # N = 3 Numero de discos


        lui  s1, 0x10010       # Dirección base de la torre A
        addi s1, s1, 0         # Torre A: 0x10010000
        addi s2, s1, 4         # Torre B: 0x10010004
        addi s3, s2, 4         # Torre C: 0x10010008

        slli t0, s0, 5         # Calcula el desplazamiento de memoria
        add  s2, s2, t0        # Ajusta la direccion de la Torre B
        add  s3, s3, t0        # Ajusta la direccion de la Torre C

        addi t2, zero, 0x00    # Inicializacion de variables auxiliares
        addi s10, zero, 0      # Contador de movimientos
        addi s11, zero, 0x20   # Espaciado entre discos en memoria
        addi t1, s0, -1        # n = N - 1 su uso es para las iteraciones
        addi t0, zero, 0       # Contador
        addi t5, zero, 0       # Desplazamiento
```

Este segmento de código en RISC-V prepara el entorno inicial para resolver el problema de la Torre de Hanoi con 3 discos. Primero, se define el número total de discos (N = 3) y se almacenan las direcciones base en memoria para cada una de las tres torres: la Torre A se ubica en la dirección 0x10010000, la Torre B y la Torre C se colocan a una distancia fija (96 bytes) entre ellas para evitar que se sobrepongan en memoria. Posteriormente, se inicializan varios registros auxiliares, incluyendo un contador de movimientos, el espacio que habrá entre los discos en la memoria (32 bytes por disco) y otras variables temporales que serán utilizadas para recorrer los discos, calcular desplazamientos en memoria y controlar el flujo del programa. Todo esto establece las condiciones necesarias para inicializar la Torre A con los discos y luego ejecutar la lógica recursiva que resolverá el problema de la Torre de Hanoi.

```
# Este es un bucle que inicializa la Torre A
for:
    beq s0, t0, torre_hanoi    # Si t0 == s0, inicia la recursion de Torre de Hani
    addi t1, t0, 1             # Incrementa el indice del disco
    add  t2, zero, s1          # Direccion base de la Torre A
    add  t4, t2, t5            # Calcula la direccion especifica del disco
    sw   t1, 0(t4)             # Guarda el disco en memoria
    addi t5, t5, 0x20          # Avanza en la memoria
    addi t0, t0, 1             # Incrementa el contador
    j    for                   # Se repite el bucle

# Esto realiza la llamada a la función recursiva de la Torre de Hanoi
jal ra, torre_hanoi
j exit

# Funcion recursiva de Torre de Hanoi
torre_hanoi:
    addi t0, zero, 1
```

Este bloque de código contiene un bucle que se encarga de inicializar la Torre A con los discos, colocándolos en la memoria uno por uno desde el más grande hasta el más pequeño. El bucle compara si el contador t0 ha alcanzado el número total de discos s0; si aún no los ha insertado todos, calcula la dirección exacta en la memoria de la Torre A donde debe colocarse el siguiente disco, lo guarda con sw y luego actualiza tanto el contador como el desplazamiento de memoria. Una vez que se han insertado todos los discos, se hace una llamada a la función recursiva que resolverá el problema de la Torre de Hanoi. Esta función empieza por comprobar si solo queda un disco por mover, en cuyo caso simplemente lo transfiere de la torre de origen a la torre destino, sin aplicar más recursión. En caso contrario, se iniciará el proceso recursivo completo para resolver el problema de manera estructurada.

```
    # Condicion en la que si hay solo un disco,se movera directamente
if:
    bne  s0, t0, else
    sw   zero, 0x0(s1)      # Remueve el disco de la torre de origen
    add  s1, s1, s11        # Ajusta puntero de torre A
    sub  s3, s3, s11        # Ajusta puntero de torre C
    sw   s0, 0(s3)          # Mueve el disco a la torre destino
    addi s10, s10, 1        # Incrementa el contador de movimientos
    jalr ra                 # Realiza el retorno

# En esto es la parte recursiva
else:
    addi sp, sp, -4
    sw   ra, 0x0(sp)        # Guarda el valor de "ra" en la pila
    addi sp, sp, -4
    sw   s0, 0x0(sp)        # Guarda s0 en la pila
    addi s0, s0, -1         # Reduce el numero de discos

 # Intercambia torres para la primera llamada recursiva
    add  t1, s2, zero       # Guarda torre intermedia temporalmente
    add  s2, s3, zero       # Torre intermedia toma el valor de torre destino
    add  s3, t1, zero       # Torre destino toma el valor original de torre intermedia
    jal  ra, torre_hanoi    # Llamada recursiva
```

En este segmento se evalúa si solo hay un disco por mover. Si esa condición se cumple, el disco se transfiere directamente desde la torre de origen a la torre destino, eliminándolo de su ubicación actual, ajustando los punteros de memoria para reflejar el movimiento, y luego guardando el disco en la nueva posición. Además, se incrementa el contador de movimientos y se retorna de la función. Si hay más de un disco, se entra a la parte recursiva del algoritmo. En esta sección se guarda el estado actual de los registros ra y s0 en la pila para no perder su información. Luego se decrementa el valor de s0, lo que representa el paso a una subinstancia del problema con un disco menos. A continuación, se realiza un intercambio de torres para preparar la primera llamada recursiva: la torre intermedia y la torre destino se intercambian para que el algoritmo pueda mover los discos superiores a la torre intermedia. Finalmente, se realiza la llamada recursiva a la función que resolverá ese subproblema.

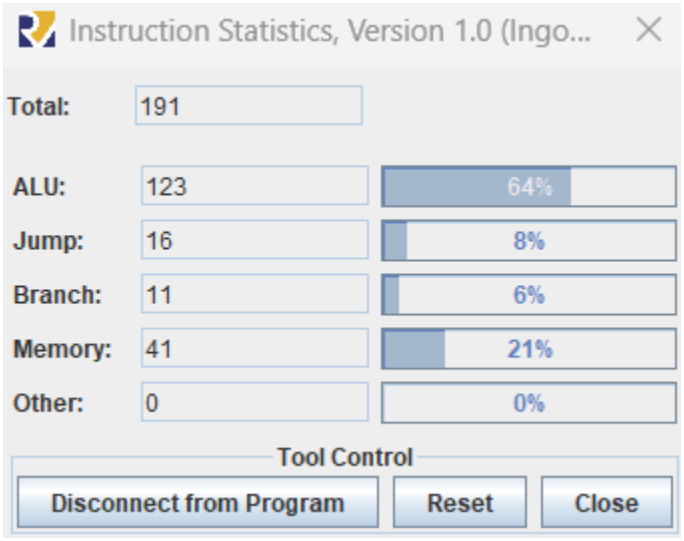
```
# Restaura el orden original de las torres despues de la primera llamada
add t1, s2, zero
add s2, s3, zero
add s3, t1, zero

# Recupera los valores de pila para continuar
lw s0, 0x0(sp)
addi sp, sp, 4
lw ra, 0x0(sp)
addi sp, sp, 4

# Mueve el disco mas grande directamente desde torre origen a torre destino
sw zero, 0x0(s1)        # Elimina el disco de la torre de origen
add s1, s1, s11         # Ajusta puntero de torre origen
sub s3, s3, s11         # Ajusta puntero de torre destino
sw s0, 0x0(s3)          # Mueve el disco
addi s10, s10, 1        # Incrementa el contador de movimientos

# Segunda llamada recursiva: mueve los discos restantes al destino final
addi sp, sp, -4
sw ra, 0x0(sp)          # Guarda ra
addi sp, sp, -4
sw s0, 0x0(sp)          # Guarda s0
addi s0, s0, -1         # Decrementa numero de discos
```

Después de la primera llamada recursiva, se restauran las referencias originales de las torres para continuar con el proceso. Se intercambian nuevamente los punteros de las torres intermedia y destino para devolverlas a su estado previo. Luego se recuperan los valores previamente guardados en la pila, es decir, los registros s0 y ra, para mantener la integridad del estado anterior a la llamada recursiva. A continuación, se mueve el disco más grande desde la torre de origen a la torre destino, ajustando los punteros en memoria y registrando el movimiento incrementando el contador. Finalmente, se prepara la segunda llamada recursiva, que moverá los discos restantes desde la torre intermedia a la torre destino. Para esto, se guarda nuevamente el estado actual en la pila y se reduce el número de discos en uno, repitiendo así el proceso recursivo.

```
    # Intercambia torres para preparar segunda llamada recursiva
    add t1, s1, zero
    add s1, s2, zero
    add s2, t1, zero

    jal ra, torre_hanoi    # Llamada recursiva

    # Restaura el orden original de las torres despues de la segunda llamada
    add t1, s1, zero
    add s1, s2, zero
    add s2, t1, zero

    # Recupera registros
    lw s0, 0x0(sp)
    addi sp, sp, 4
    lw ra, 0x0(sp)
    addi sp, sp, 4

    jalr ra                # Retorna

# Fin del programa
exit:
    j exit
```

Se realiza un intercambio entre las torres origen e intermedia para preparar la segunda llamada recursiva de la función Torre de Hanoi. Este intercambio ajusta los punteros para que las torres se comporten como se requiere en la siguiente etapa del algoritmo. Luego, se realiza la llamada recursiva. Al finalizar esa llamada, se restauran nuevamente las referencias originales de las torres, devolviendo sus punteros a su estado anterior. Posteriormente, se recuperan los registros previamente guardados en la pila, en este caso s0 y ra, para restaurar el estado del programa. Finalmente, se retorna de la función recursiva con la instrucción jalr ra. Después, el programa entra en un bucle infinito al alcanzar la etiqueta exit, lo que indica el final de la ejecución.

Cantidad de instrucciones utilizadas para 3 discos

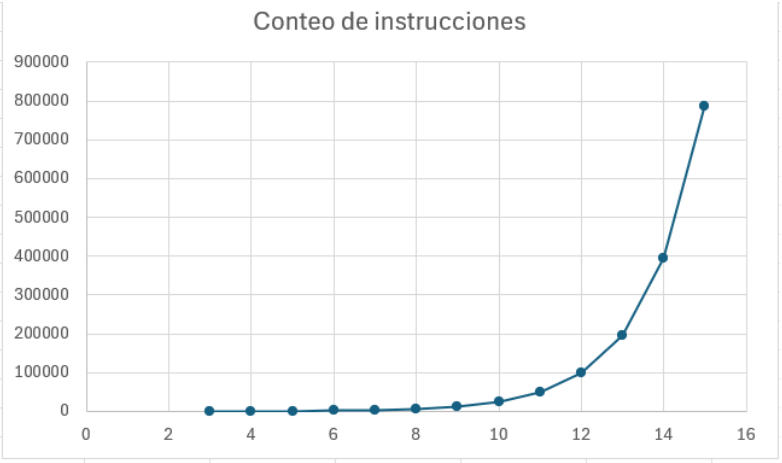


Demostración de código

https://drive.google.com/drive/folders/1M8dd1KFBlwUIoYBy_AOAEbImA9l2lStg?usp=sharing

Grafica del flujo de instrucciones

Discos	Conteo de instrucciones
3	191
4	391
5	783
6	1559
7	3103
8	6183
9	12335
10	24631
11	49215
12	98375
13	196687
14	393303
15	786527



Conclusiones

Santiago Josué García Morales

La implementación del problema de las Torres de Hanói en lenguaje ensamblador RISC-V me permitió comprender de manera profunda el funcionamiento de la recursión a bajo nivel, así como el manejo de la pila y la segmentación de datos en la arquitectura RISC. A través del desarrollo y simulación del programa en RARS, se observó que incluso algoritmos conceptualmente simples pueden implicar una considerable carga computacional, evidenciada en el crecimiento exponencial del Instruction Count a medida que se incrementa el número de discos.

Bryan Alejandro Villalpando Castillo

La implementación recursiva del algoritmo de las Torres de Hanói en lenguaje ensamblador RISC-V permitió comprender a profundidad el uso del stack para manejar llamadas recursivas, así como el control preciso del flujo del programa sin utilizar pseudo-instrucciones. La simulación del movimiento de discos en el segmento de datos y el análisis del instruction count brindaron una perspectiva más tangible sobre el impacto computacional del crecimiento exponencial del algoritmo. Esta experiencia no solo fortaleció los conocimientos técnicos, sino que también desarrolló habilidades de precisión, documentación y optimización en programación de bajo nivel.