Instituto Tecnológico y de Estudios

Superiores de Occidente – ITESO

ITESO

Universidad Jesuita
de Guadalajara

Materia: Organización y Arquitectura de Computadoras

Maestro: JuanPablo Ibarra Esparza

Tarea: Practica 1

Fecha: 29/03/2024

Autor(es): Contreras Barragan, Jaime Antonio - 744197

Diaz Veloz, José Daniel - 717550

Índice:

¿Qué son las Torres de Hanoi?:

Son un problema matemático y de lógica que se plantea como una serie de discos de diferentes tamaños que están dispuestos en una torre en un orden decreciente de tamos, es decir de abajo hacia arriba, con el objetivo de que la torre de origen a la torre destino, utilizando una auxiliar como temporal.

En el caso de informática se utiliza este tipo de problema para entender la recursividad. Consiste en 3 principios o reglas simples:

1. Solo se puede mover un disco a la vez
2. Nunca puedes poner un disco más grande sobre uno pequeño
3. Solo puede mover el disco superior de una torre.

Ejemplo de Torre de Hanoi en C:

```
#// Función para mover los discos de una torre a otra
#void torresHanoi(int n, int torreOrigen[], int torreAuxiliar[], int torreDestino[]) {
#    if (n == 1) {
#        // Mueve el disco de la torre origen a la torre destino
#        int disco = torreOrigen[0];
#        printf("Mueve el disco %d de la torre %d a la torre %d\n", disco, torreOrigen - torreOrigen + 1, torreDestino - torreDestino + 1);
#        // Elimina el disco de la torre origen
#        torreOrigen[0] = 0;
#        // Añade el disco a la torre destino
#        torreDestino[0] = disco;
#    } else {
#        // Mueve n-1 discos de la torre origen a la torre auxiliar usando la torre destino
#        torresHanoi(n - 1, torreOrigen, torreDestino, torreAuxiliar);
#        // Mueve el disco restante de la torre origen a la torre destino
#        torresHanoi(1, torreOrigen, torreAuxiliar, torreDestino);
#        // Mueve n-1 discos de la torre auxiliar a la torre destino usando la torre origen
#        torresHanoi(n - 1, torreAuxiliar, torreOrigen, torreDestino);
#    }
#}
#int main() {
#    // Número de discos
#    int n = 3;

#    // Definición de las torres como arreglos de enteros
#    int torreA[] = {3, 2, 1};
```

int torreB[] = {0, 0, 0};

int torreC[] = {0, 0, 0};

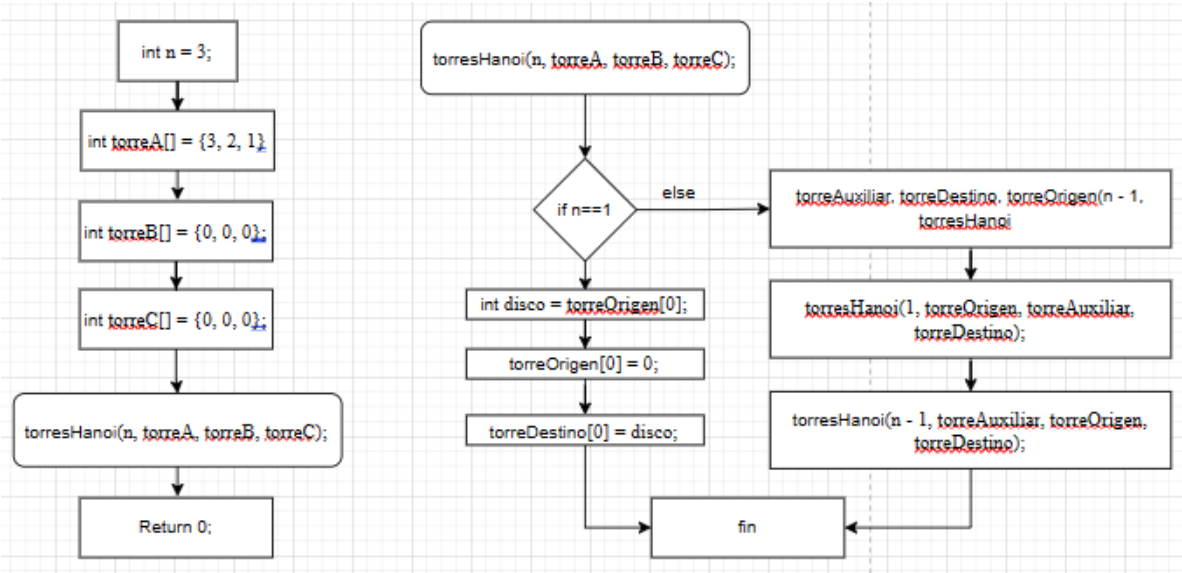
// Llama a la función de Torres de Hanoi

torresHanoi(n, torreA, torreB, torreC);

return 0;

#}

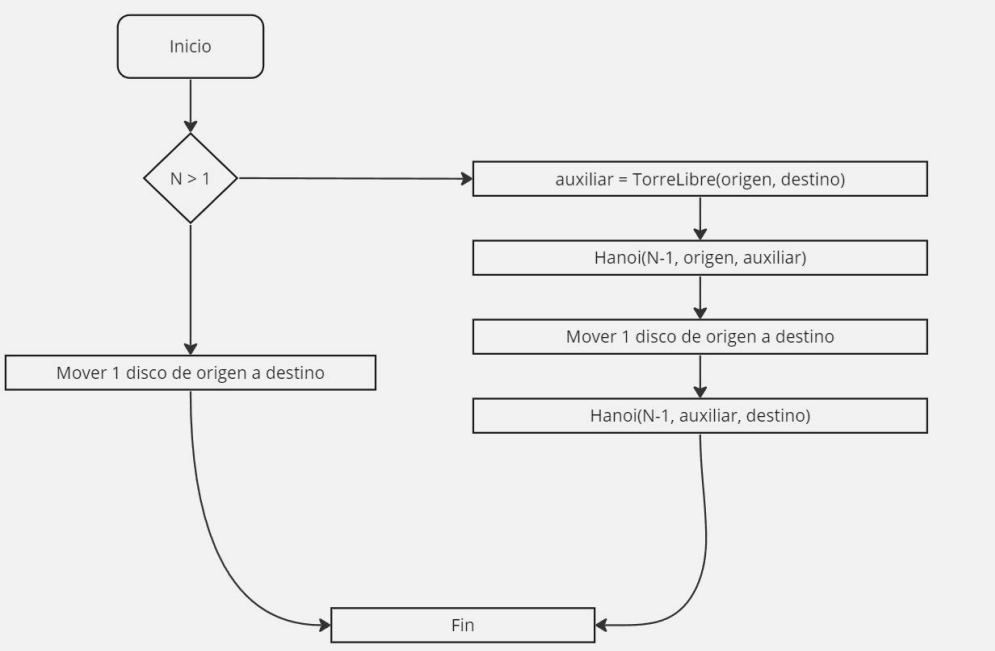
Diagrama de flujo (basado en C):



Pseudocódigo:

Hanoi (N, origen, destino)
Si N >1
Auxiliar ← TorreLibre(origen, destino)
Hanoi (N-1, origen,auxiliar)
Print “mover 1 disco de origen a destino
Hanoi (N-1, auxiliar,destino)
En caso contrario
Print “mover 1 disco de origen a destino”

Diagrama de flujo (basado en pseudocódigo):



Decisiones sobre el algoritmo:

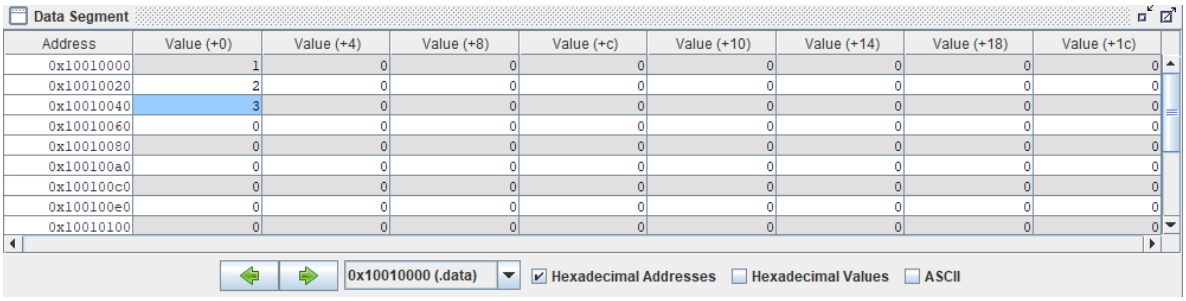
La práctica de Torres de Hanoi nos brindó la oportunidad de plantear diferentes estrategias e implementación en RISC-V, las decisiones fueron bastante diferentes, al principio de la práctica se optó por hacer una pila con un switch y otras funciones que hicieran el movimiento; Esta pila serviría para almacenar los discos durante la ejecución del algoritmo. La función principal se basaba en la recursividad, moviendo los discos de la torre origen a la de destino mediante la torre auxiliar. El switch funciona como un intermediario que evaluaba si se cumple o existen discos la función y las otras funciones se encargan del movimiento.

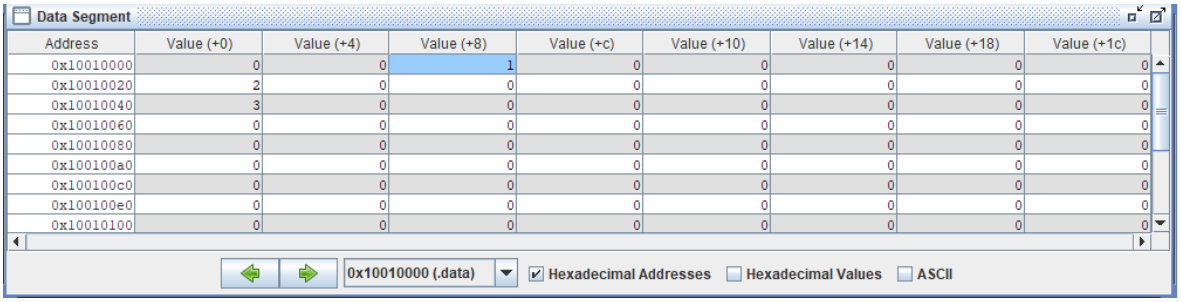
Sin embargo, la implementación inicial no logró cumplir con los requisitos del algoritmo. El principal problema radica en el manejo del movimiento de los discos entre las torres. La lógica del switch (posible problema) o el manejo del SP (Stack Pointer) no era lo suficientemente flexible para adaptarse a las diferentes configuraciones y casos posibles.

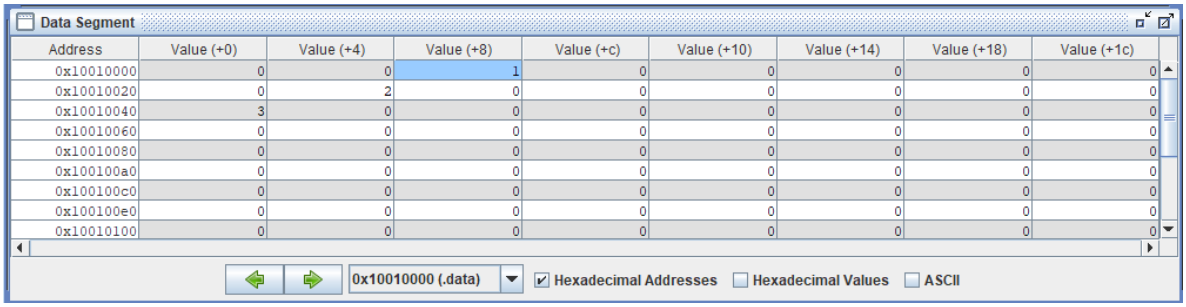
A pesar de los desafíos encontramos en la implementación inicial, la práctica de las torres de Hanói se completó de manera que funcionara más o menos correcta, estamos conscientes que no fue el resultado que se esperaba o que se tenía que lograr en la práctica, pero funciona de manera parcial, no se logró del todo arreglar las dificultades presentadas, pero después de un gran análisis profundo de las deficiencias de la implementación inicial en especial sobre el manejo y uso del “sp” y “ra”, se rediseño el algoritmo con el fin de optimizar el manejo del movimiento de discos, se mejoró la lógica y manejo del “sp” para tener mejor flexibilidad.

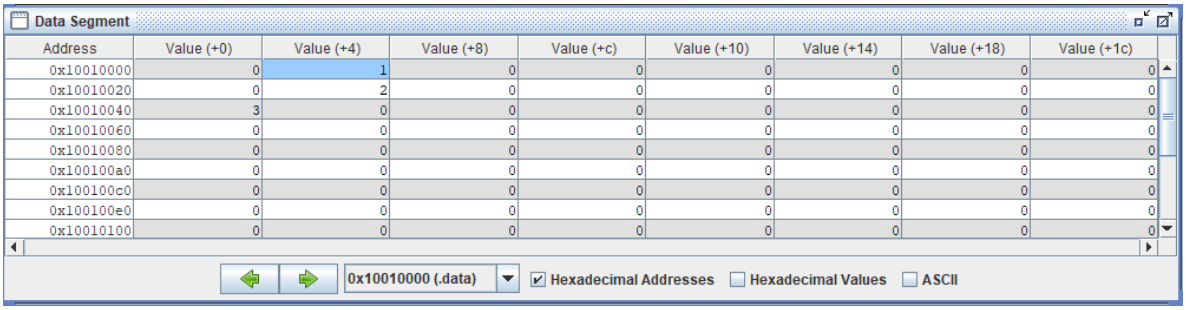
Demostración:

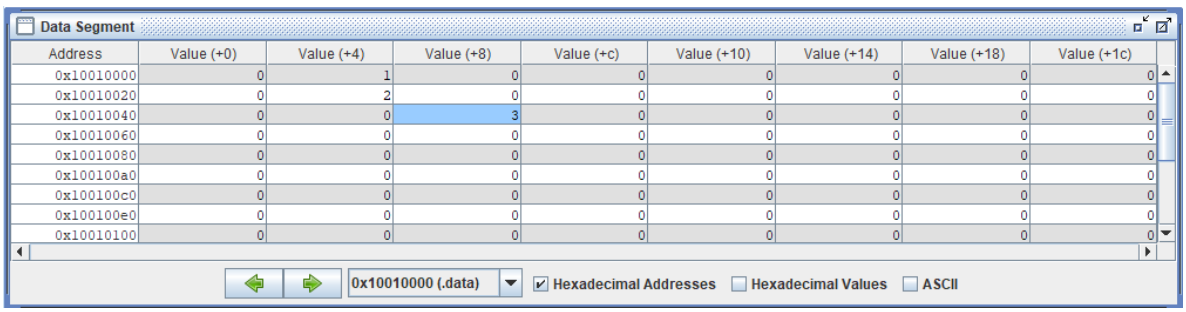
En este ejemplo se usó 3 discos para comprobar el funcionamiento/ejecución adecuado de la Torre de Hanói

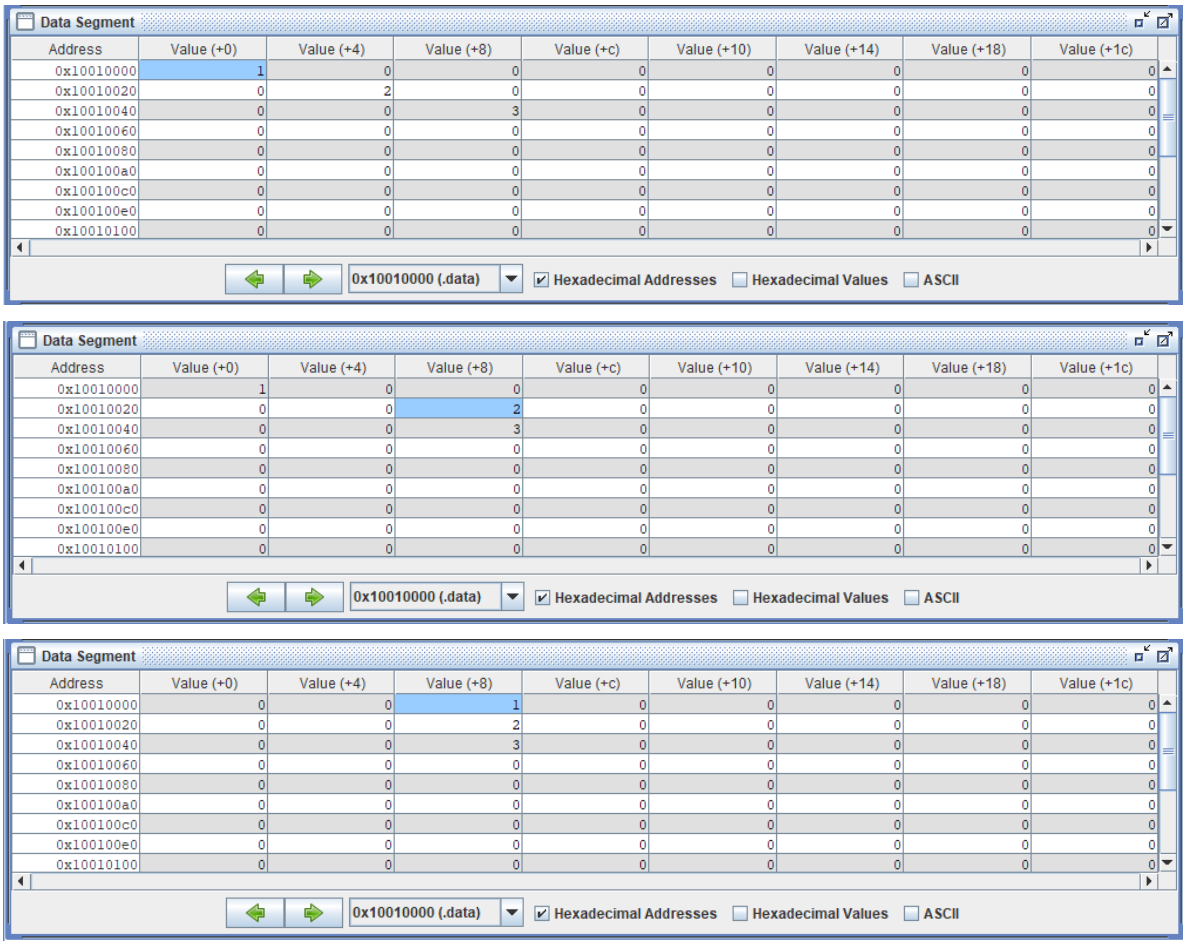












Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1	0	0	0	0	0	0	0
0x10010020	0	2	0	0	0	0	0	0
0x10010040	0	0	3	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1	0	0	0	0	0	0	0
0x10010020	0	0	2	0	0	0	0	0
0x10010040	0	0	3	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0	0	1	0	0	0	0	0
0x10010020	0	0	2	0	0	0	0	0
0x10010040	0	0	3	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Análisis:

En términos generales el algoritmo de la Torre de Hanói (stack) funciona de la siguiente manera:

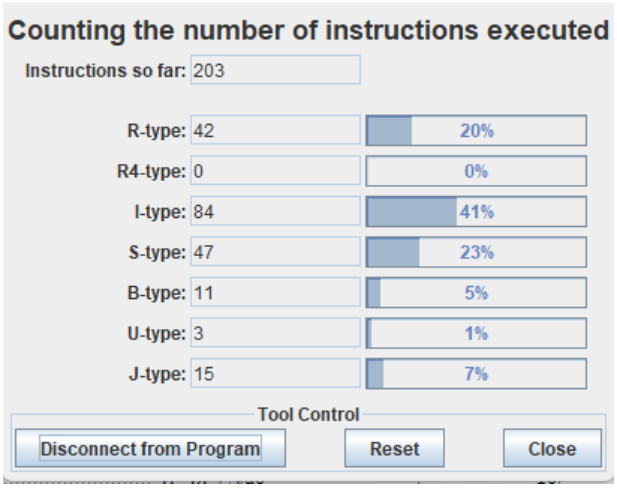
1. Revisa si el valor de s0 es igual a 1 para entrar al caso base.
2. 1. reservamos el espacio de memoria para almacenar las variables importantes (origen, auxiliar y destino, s0)
3. Se guardan los valores mediante las instrucciones “sw” antes de realizar la llamada recursiva
4. Antes de llamar a la función se modifica el valor de “n”, en este decrementando en 1, también se hace un swap de auxiliar con destino
5. Se modifica los argumentos y variables a través de la llamada recursiva de la función torres de Hanói que es igual a “n - 1”.
6. Cada vez que se llama a la función recursiva, se restauran los valores de n, origen, auxiliar, destino y “ra” mediante la instrucción “lw”.
7. s11 sirve para tener un correcto flujo en la ejecución y garantizar el retorno al punto correcto después de realizar el movimiento de discos, que se hace con una etiqueta que copia el caso base, que se hace en la llamada recursiva.

8. El offset “t5” funciona para poder acceder a las posiciones correctas de los discos y en qué torre se encuentra
9. El movimiento de los discos se realiza copiando y limpiando las posiciones correspondientes
10. Se implementan los casos u condicionales para ver si n es igual a 1, en dado caso que, si sea, mueve el disco a otra torre
11. Después se restaura el puntero del “stack” en su posición anterior.
12. Y repetimos para la segunda llamada recursiva, pero haciendo un swap entre origen y auxiliar.

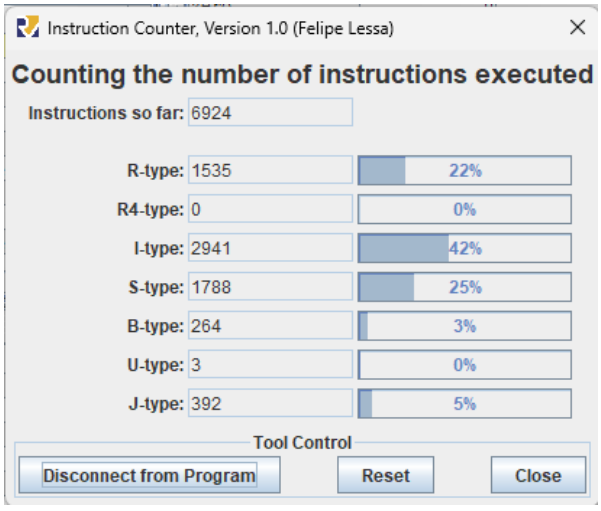
Count (IC):

El count (IC) nos permite sabre el número de instrucciones que se realizaron durante la ejecución del algoritmo, nos muestra el número de veces que se utilizaron las instrucciones R,I,S,B,U y J.

Porcentaje de instrucciones para 3 discos.



Porcentaje de instrucciones para 8 discos.

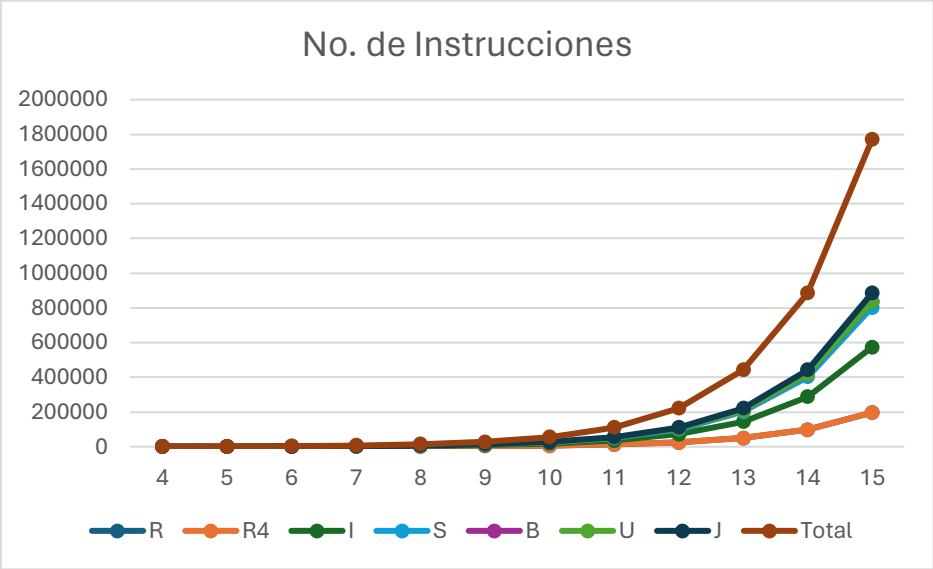


Grafica del incremento gradual de los discos 4 a 15:

En términos generales la gráfica proporcionada ilustra el aumento del número total de las instrucciones (IC) necesarias para resolver/mover el algoritmo de las Torres de Hanoi, en la primera imagen podemos observar los datos que se obtuvieron de 4 a 15 discos, donde "x" es eje que los representa y el "y" muestra el número total de las instrucciones, por ende, se puede observar un crecimiento exponencial al aumentar el número de discos.

Tipo de instrucciones/ No. Disco	4	5	6	7	8	9	10	11	12	13	14	15
R	91	188	381	766	1535	3072	6145	12290	24579	49156	98309	196614
R4	0	0	0	0	0	0	0	0	0	0	0	0
I	177	362	731	1468	2941	5886	11775	23552	47105	94210	188419	376836
S	104	217	442	891	1788	3581	7166	14335	28672	57345	114690	229379
B	20	37	70	135	264	521	1034	2059	4108	8205	16398	32783
U	3	3	3	3	3	3	3	3	3	3	3	3
J	28	53	102	199	392	777	1546	3083	6156	12301	24590	49167
Total	424	861	1730	3463	6924	13841	27670	55323	110624	221221	442410	884783

El crecimiento exponencial se debe a la función recursiva del algoritmo para resolverlo, cada paso implica mover un disco y realizar operaciones adicionales a nivel memoria que multiplica el número de instrucciones además de la complejidad que conlleva, esto se ve visualizado en la siguiente grafica.



Conclusiones:

Jaime:

Le dedicamos mucho tiempo a esta práctica. Queríamos lograr que nuestros movimientos de los discos fueran iguales a los vistos en el gif de muestra. Lamentablemente por más que lo intentamos no lo logramos, tuvimos que cambiar nuestra estructura, y aceptar que esta vez fallamos.

Sin embargo, sin duda seguimos agregando conocimiento a partir de nuestros errores a lo largo de la elaboración, aunque seguimos con dudas.

Supongo tendré que pedir ayuda para poder terminar de entender como sirven sp, ra cuando das saltos a etiquetas en funciones recursivas.

Daniel:

En general, la práctica represento un desafío por la dificultad del algoritmo para que funcione bien, personalmente trate de mil maneras para finalizarlo o que funcionase, lo implica hacerlo iterativa en vez de recursiva para ver si había alguna manera de pasar de uno a otro, lo que sorprendió es imposible.

Tuvimos que cambiar nuestra estructura hasta llegar un código que funcione medianamente bien. Para finalizar tendré que aclar mis dudas o preguntar más sobre las funciones recursivas en ensamblador y cómo manejarlas de mejor manera.