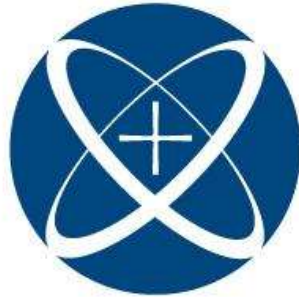


INSTITUTO TECNOLÓGICO DE ESTUDIOS SUPERIORES DE OCCIDENTE

ARQUITECTURA COMPUTACIONAL



ITESO, Universidad
Jesuita de Guadalajara

NOMBRE DEL DOCENTE

JUANPABLO IBARRA ESPARZA

INTEGRANTES:

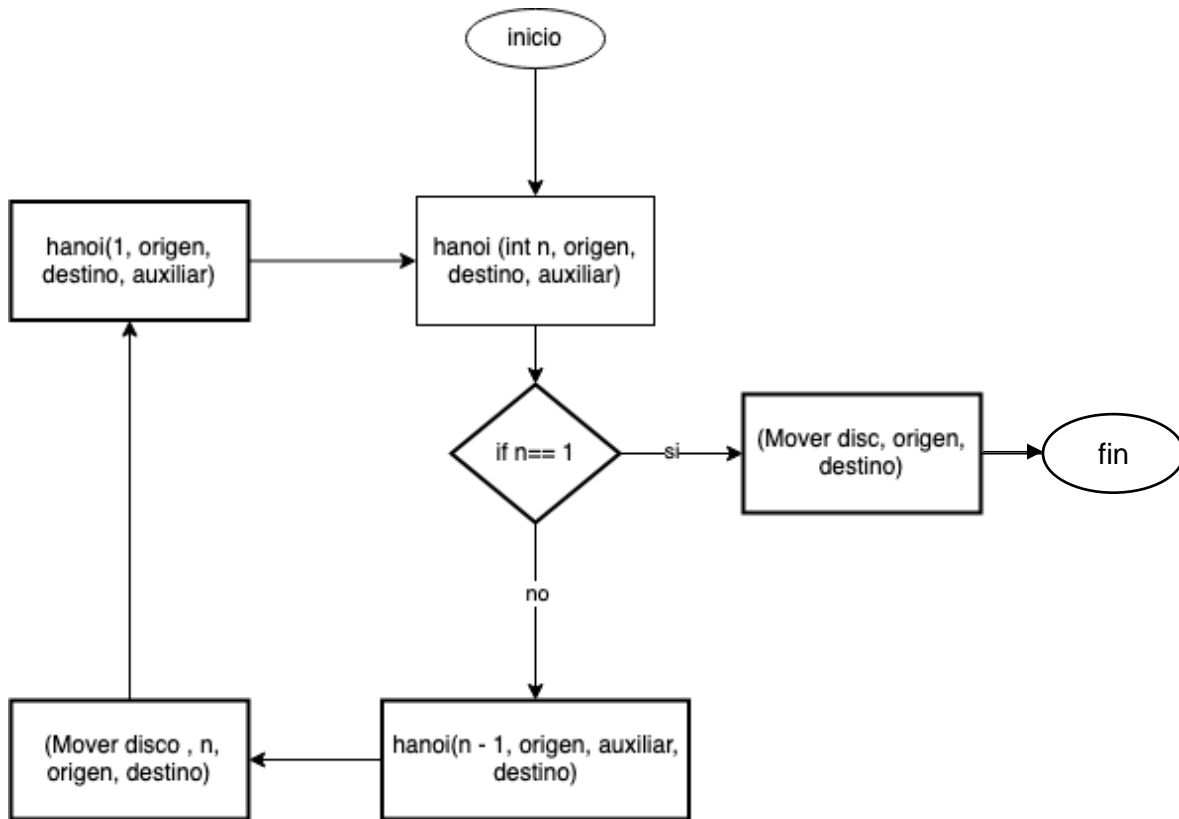
Paulina Valadez Fernández 728412

Perla Marina García García 728149

Práctica 1

25 de octubre de 2023

1. Diagrama de flujo del programa implementado realizado en Visio o programa equivalente.



2. Las decisiones que se tomaron al diseñar su programa se pueden tomar como referencia el diagrama de flujo del programa.

Primero recordar lo que son las torres de Hanoi y cómo se realizaban los pasos, después se tomó en cuenta encontrar un programa en C y entender el funcionamiento de que realizaba dicho programa, el siguiente código fue nuestra base:

```
#include <stdio.h>
#include <unistd.h>

void hanoi(int n, char origen, char destino, char auxiliar) {
    if (n == 1) {
        printf("Mover disco 1 desde %c a %c\n", origen, destino);
        return;
    }
}
```

```

    }
    hanoi(n - 1, origen, auxiliar, destino);

    printf("Mover disco %d desde %c a %c\n", n, origen, destino);
    hanoi(1, origen, destino, auxiliar);
    hanoi(n - 1, auxiliar, destino, origen);
}

int main() {
    int num_discos;
    printf("Ingresa el número de discos: ");
    scanf("%d", &num_discos);
    printf("\nLos pasos a seguir son:\n");
    hanoi(num_discos, 'A', 'C', 'B');
    printf("Esperando 20 segundos antes de salir...\n");
    sleep(60);
    return 0;
}

```

También identificamos cuales eran los pasos en cada llamada recursiva:

1.- Mover parte superior a NO Destino

1.1 Ajustar apuntadores (Reducir Origen y Aumentar Destino)

2.- Mover pieza actual a Destino

3.- Mover parte superior a Destino

3. Una simulación en el RARS para 3 discos (La simulación son impresiones de pantalla de data segment del RARS).

The screenshot displays the RARS simulator interface. The 'Text Segment' window shows the assembly code for the Hanoi Tower program. The 'Data Segment' window shows the memory layout, with the 'Data' segment starting at address 0x1000000. The 'Registers' window on the right shows the current state of the registers, including the 'PC' (Program Counter) at 0x1000000.

Register	Name	Number	Value
PC	PC	0	0x1000000
R0	R0	1	0x0000000
R1	R1	2	0x0000000
R2	R2	3	0x0000000
R3	R3	4	0x0000000
R4	R4	5	0x0000000
R5	R5	6	0x0000000
R6	R6	7	0x0000000
R7	R7	8	0x0000000
R8	R8	9	0x0000000
R9	R9	10	0x0000000
R10	R10	11	0x0000000
R11	R11	12	0x0000000
R12	R12	13	0x0000000
R13	R13	14	0x0000000
R14	R14	15	0x0000000
R15	R15	16	0x0000000
R16	R16	17	0x0000000
R17	R17	18	0x0000000
R18	R18	19	0x0000000
R19	R19	20	0x0000000
R20	R20	21	0x0000000
R21	R21	22	0x0000000
R22	R22	23	0x0000000
R23	R23	24	0x0000000
R24	R24	25	0x0000000
R25	R25	26	0x0000000
R26	R26	27	0x0000000
R27	R27	28	0x0000000
R28	R28	29	0x0000000
R29	R29	30	0x0000000
R30	R30	31	0x0000000

Edit Execute						Registers Floating Point Control and Status		
Text Segment						Name	Number	Value
Byte	Address	Code	Source		Base	PC	0	0x00000000
	0x00401014	0x00412493 (14)	lw r2, 4(r0)(PC)		lw r12, 4(r0)	TA	1	0x00400014
	0x00401018	0x00412493 (17)	lw r3, 8(r0)(PC+3)		lw r13, 8(r0)	PD	2	0x7FFFFF00
	0x0040101C	0x00412793 (18)	lw r4, 12(r0)(PC+6)		lw r14, 12(r0)	DE	3	0x10000000
	0x00401020	0x00412793 (19)	lw r5, 16(r0)(PC+9)		lw r15, 16(r0)	DP	4	0x00000000
	0x00401024	0x01410113 (20)	addi r0, r0, +20 #regress a la d...		addi r2, r2, 20	TO	5	0x00000000
	0x00401028	0x00000094 (21)	jalr r0, #regress a la liame		jalr r1, r1, 0	TI	6	0x00000000
	0x0040102C	0x00414a33 (24) if:	lw r2, 0(r0) # cargo r1 valor de...		lw r7, 0(r13)	TD	7	0x00000000
	0x00401030	0x00777a03 (25)	sw r2, 0(r0) # destino = origen		sw r7, 0(r15)	DO	8	0x00000000
	0x00401034	0x00414a33 (26)	sw zero, 0(r0) # se carga un 0 en...		sw r0, 0(r13)	DI	9	0x10100000
	0x00401038	0x00000094 (27)	jalr r0		jalr r1, r1, 0	PO	10	0x00000000
	0x0040103C					PI	11	0x00000000
						P2	12	0x00000000
						P3	13	0x10100000
						P4	14	0x10000000
						P5	15	0x10100100
						P6	16	0x00000000
						P7	17	0x00000000
						P8	18	0x00000000
						P9	19	0x00000000
						PA	20	0x00000000
						PI	21	0x00000000
						PE	22	0x00000000
						PT	23	0x00000000

Text Segment						Name	Number	Value
Byte	Address	Code	Source		Base	PC	0	0x00000000
	0x00401014	0x00412493 (14)	lw r2, 4(r0)(PC)		lw r12, 4(r0)	TA	1	0x00400014
	0x00401018	0x00412493 (17)	lw r3, 8(r0)(PC+3)		lw r13, 8(r0)	PD	2	0x7FFFFF00
	0x0040101C	0x00412793 (18)	lw r4, 12(r0)(PC+6)		lw r14, 12(r0)	DE	3	0x10000000
	0x00401020	0x00412793 (19)	lw r5, 16(r0)(PC+9)		lw r15, 16(r0)	DP	4	0x00000000
	0x00401024	0x01410113 (20)	addi r0, r0, +20 #regress a la d...		addi r2, r2, 20	TO	5	0x00000000
	0x00401028	0x00000094 (21)	jalr r0, #regress a la liame		jalr r1, r1, 0	TI	6	0x00000000
	0x0040102C	0x00414a33 (24) if:	lw r2, 0(r0) # cargo r1 valor de...		lw r7, 0(r13)	TD	7	0x00000000
	0x00401030	0x00777a03 (25)	sw r2, 0(r0) # destino = origen		sw r7, 0(r15)	DO	8	0x00000000
	0x00401034	0x00414a33 (26)	sw zero, 0(r0) # se carga un 0 en...		sw r0, 0(r13)	DI	9	0x10100000
	0x00401038	0x00000094 (27)	jalr r0		jalr r1, r1, 0	PO	10	0x00000000
	0x0040103C					PI	11	0x00000000
						P2	12	0x00000000
						P3	13	0x10100000
						P4	14	0x10000000
						P5	15	0x10100100
						P6	16	0x00000000
						P7	17	0x00000000
						P8	18	0x00000000
						P9	19	0x00000000
						PA	20	0x00000000
						PI	21	0x00000000
						PE	22	0x00000000
						PT	23	0x00000000

Edit Execute						Registers Floating Point Control and Status		
Text Segment						Name	Number	Value
Byte	Address	Code	Source		Base	PC	0	0x00000000
	0x00401014	0x00412493 (14)	lw r2, 4(r0)(PC)		lw r12, 4(r0)	TA	1	0x00400014
	0x00401018	0x00412493 (17)	lw r3, 8(r0)(PC+3)		lw r13, 8(r0)	PD	2	0x7FFFFF00
	0x0040101C	0x00412793 (18)	lw r4, 12(r0)(PC+6)		lw r14, 12(r0)	DE	3	0x10000000
	0x00401020	0x00412793 (19)	lw r5, 16(r0)(PC+9)		lw r15, 16(r0)	DP	4	0x00000000
	0x00401024	0x01410113 (20)	addi r0, r0, +20 #regress a la d...		addi r2, r2, 20	TO	5	0x00000000
	0x00401028	0x00000094 (21)	jalr r0, #regress a la liame		jalr r1, r1, 0	TI	6	0x00000000
	0x0040102C	0x00414a33 (24) if:	lw r2, 0(r0) # cargo r1 valor de...		lw r7, 0(r13)	TD	7	0x00000000
	0x00401030	0x00777a03 (25)	sw r2, 0(r0) # destino = origen		sw r7, 0(r15)	DO	8	0x00000000
	0x00401034	0x00414a33 (26)	sw zero, 0(r0) # se carga un 0 en...		sw r0, 0(r13)	DI	9	0x10100000
	0x00401038	0x00000094 (27)	jalr r0		jalr r1, r1, 0	PO	10	0x00000000
	0x0040103C					PI	11	0x00000000
						P2	12	0x00000000
						P3	13	0x10100000
						P4	14	0x10000000
						P5	15	0x10100100
						P6	16	0x00000000
						P7	17	0x00000000
						P8	18	0x00000000
						P9	19	0x00000000
						PA	20	0x00000000
						PI	21	0x00000000
						PE	22	0x00000000
						PT	23	0x00000000
						PO	24	0x00000000
						PI	25	0x00000000
						PI	26	0x00000000

[illegible]

Test Segment					Registers		
Start	Address	Code	Source	Base	Name	Number	Value
0x00000000	0x00000000	0x00000000	lw \$t2, 0(\$sp)	0x00000000	\$t2	0	0x00000000
0x00000001	0x00000001	0x00000001	lw \$t3, 0(\$sp)	0x00000001	\$t3	1	0x00000001
0x00000002	0x00000002	0x00000002	lw \$t4, 0(\$sp)	0x00000002	\$t4	2	0x00000002
0x00000003	0x00000003	0x00000003	lw \$t5, 0(\$sp)	0x00000003	\$t5	3	0x00000003
0x00000004	0x00000004	0x00000004	lw \$t6, 0(\$sp)	0x00000004	\$t6	4	0x00000004
0x00000005	0x00000005	0x00000005	lw \$t7, 0(\$sp)	0x00000005	\$t7	5	0x00000005
0x00000006	0x00000006	0x00000006	lw \$t8, 0(\$sp)	0x00000006	\$t8	6	0x00000006
0x00000007	0x00000007	0x00000007	lw \$t9, 0(\$sp)	0x00000007	\$t9	7	0x00000007
0x00000008	0x00000008	0x00000008	lw \$t10, 0(\$sp)	0x00000008	\$t10	8	0x00000008
0x00000009	0x00000009	0x00000009	lw \$t11, 0(\$sp)	0x00000009	\$t11	9	0x00000009
0x0000000a	0x0000000a	0x0000000a	lw \$t12, 0(\$sp)	0x0000000a	\$t12	10	0x0000000a
0x0000000b	0x0000000b	0x0000000b	lw \$t13, 0(\$sp)	0x0000000b	\$t13	11	0x0000000b
0x0000000c	0x0000000c	0x0000000c	lw \$t14, 0(\$sp)	0x0000000c	\$t14	12	0x0000000c
0x0000000d	0x0000000d	0x0000000d	lw \$t15, 0(\$sp)	0x0000000d	\$t15	13	0x0000000d
0x0000000e	0x0000000e	0x0000000e	lw \$t16, 0(\$sp)	0x0000000e	\$t16	14	0x0000000e
0x0000000f	0x0000000f	0x0000000f	lw \$t17, 0(\$sp)	0x0000000f	\$t17	15	0x0000000f
0x00000010	0x00000010	0x00000010	lw \$t18, 0(\$sp)	0x00000010	\$t18	16	0x00000010
0x00000011	0x00000011	0x00000011	lw \$t19, 0(\$sp)	0x00000011	\$t19	17	0x00000011
0x00000012	0x00000012	0x00000012	lw \$t20, 0(\$sp)	0x00000012	\$t20	18	0x00000012
0x00000013	0x00000013	0x00000013	lw \$t21, 0(\$sp)	0x00000013	\$t21	19	0x00000013
0x00000014	0x00000014	0x00000014	lw \$t22, 0(\$sp)	0x00000014	\$t22	20	0x00000014
0x00000015	0x00000015	0x00000015	lw \$t23, 0(\$sp)	0x00000015	\$t23	21	0x00000015
0x00000016	0x00000016	0x00000016	lw \$t24, 0(\$sp)	0x00000016	\$t24	22	0x00000016
0x00000017	0x00000017	0x00000017	lw \$t25, 0(\$sp)	0x00000017	\$t25	23	0x00000017
0x00000018	0x00000018	0x00000018	lw \$t26, 0(\$sp)	0x00000018	\$t26	24	0x00000018
0x00000019	0x00000019	0x00000019	lw \$t27, 0(\$sp)	0x00000019	\$t27	25	0x00000019
0x0000001a	0x0000001a	0x0000001a	lw \$t28, 0(\$sp)	0x0000001a	\$t28	26	0x0000001a
0x0000001b	0x0000001b	0x0000001b	lw \$t29, 0(\$sp)	0x0000001b	\$t29	27	0x0000001b
0x0000001c	0x0000001c	0x0000001c	lw \$t30, 0(\$sp)	0x0000001c	\$t30	28	0x0000001c
0x0000001d	0x0000001d	0x0000001d	lw \$t31, 0(\$sp)	0x0000001d	\$t31	29	0x0000001d
0x0000001e	0x0000001e	0x0000001e	lw \$t32, 0(\$sp)	0x0000001e	\$t32	30	0x0000001e
0x0000001f	0x0000001f	0x0000001f	lw \$t33, 0(\$sp)	0x0000001f	\$t33	31	0x0000001f

4. Análisis del comportamiento del stack para el caso de 3 discos.

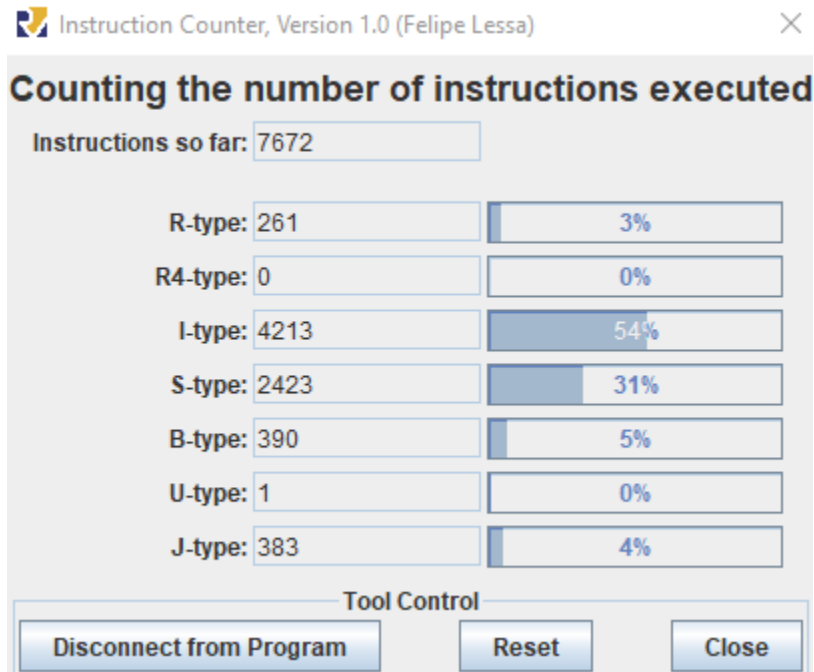
Primero modificamos la dirección de memoria en donde comenzará el stack pointer, en este caso será desde la dirección 0x7ffefe8, realizamos el push escribiendo la dirección del destino 0x10010018 ('A') en la dirección más alta para ello aplicamos un offset de 16, después en la dirección anterior escribimos la dirección del auxiliar 0x1001000c ('B') con un offset de 12, luego la dirección del origen 0x10010000 ('C') su offset es de 8, después se escribe el valor de n en ese momento que es 3, su offset es de 4, finalmente en la dirección con un offset 0 se guarda el return address 0x00400040 la cual sería la dirección a donde regresaría cuando termine la llamada. Esto lo hace cada vez que entra a una llamada recursiva sin embargo los valores cambian dependiendo del disco en el que se encuentra, ya que puede cambiar el origen, el auxiliar y el destino para cada llamada recursiva.

5. Incluir en el instruction count (IC) y especificar el porcentaje de instrucciones de tipo R, I y J para 8 discos.

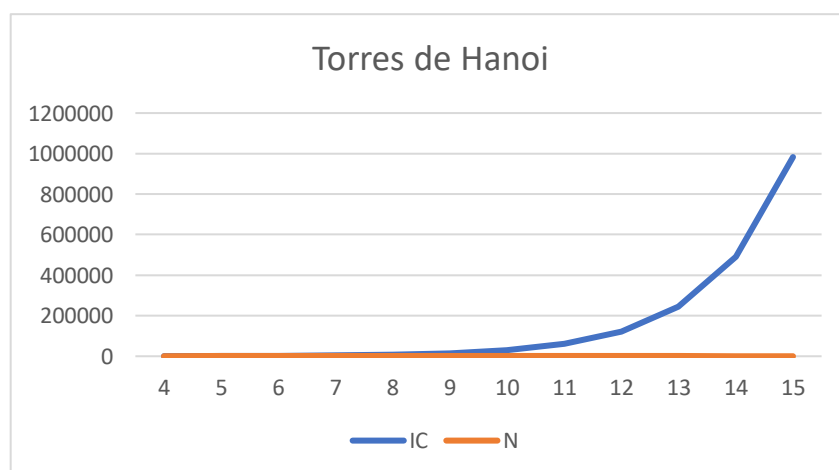
R -> 3%

I -> 54%

J ->4%



6. Una gráfica que muestre cómo se incrementa el IC para las torres de Hanoi en los casos de 4 a 15 discos.



N	IC
4	456
5	940
6	1904
7	3828
8	7672
9	15356
10	30720
11	61444
12	122888
13	245772
14	491536
15	983060

7. Conclusiones de cada integrante del equipo.

Perla: Al realizar esta práctica puede entender más el comportamiento del stack pointer para realizar las llamadas recursivas, de lo cual nunca me había preocupado ya que ensamblador necesita todas las instrucciones necesarias para escribir directamente en la memoria. Fue muy necesario comprender los pasos que realizaba las llamadas para poder traducirlas, ya que dependiendo del disco en el que se encontraba el origen, destino y auxiliar cambian para poder cumplir la regla de no poner un disco más grande encima de uno mas pequeño, y al modificar los argumentos estos deben de tener las nuevas direcciones de memoria que se habían guardado en el stack pointer.

Pau: Para esta práctica aprendí cómo se implementa el famoso jueguito de torres de hanoi utilizando llamadas recursivas y la pila que es nuestro stack. La recursión nos ayudó bastante para poder dividir el problema en subproblemas más pequeños y resolverlos de manera eficiente.