Instituto Politécnico Nacional

Escuela Superior de Cómputo

Arquitectura de Computadoras

“Implementación del Monociclo”

Alumno:

Malagón Baeza Alan Adrian

Profesor:
Alemán Arce Miguel Ángel

Grupo: 5CV1

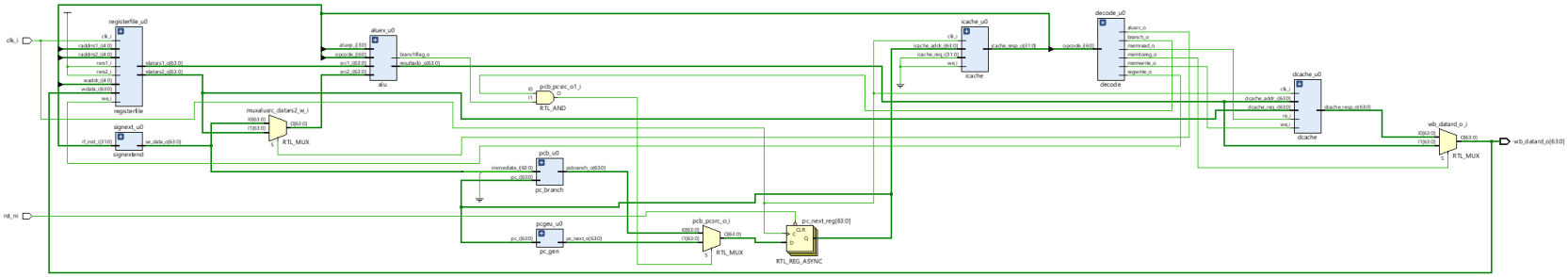
Introducción

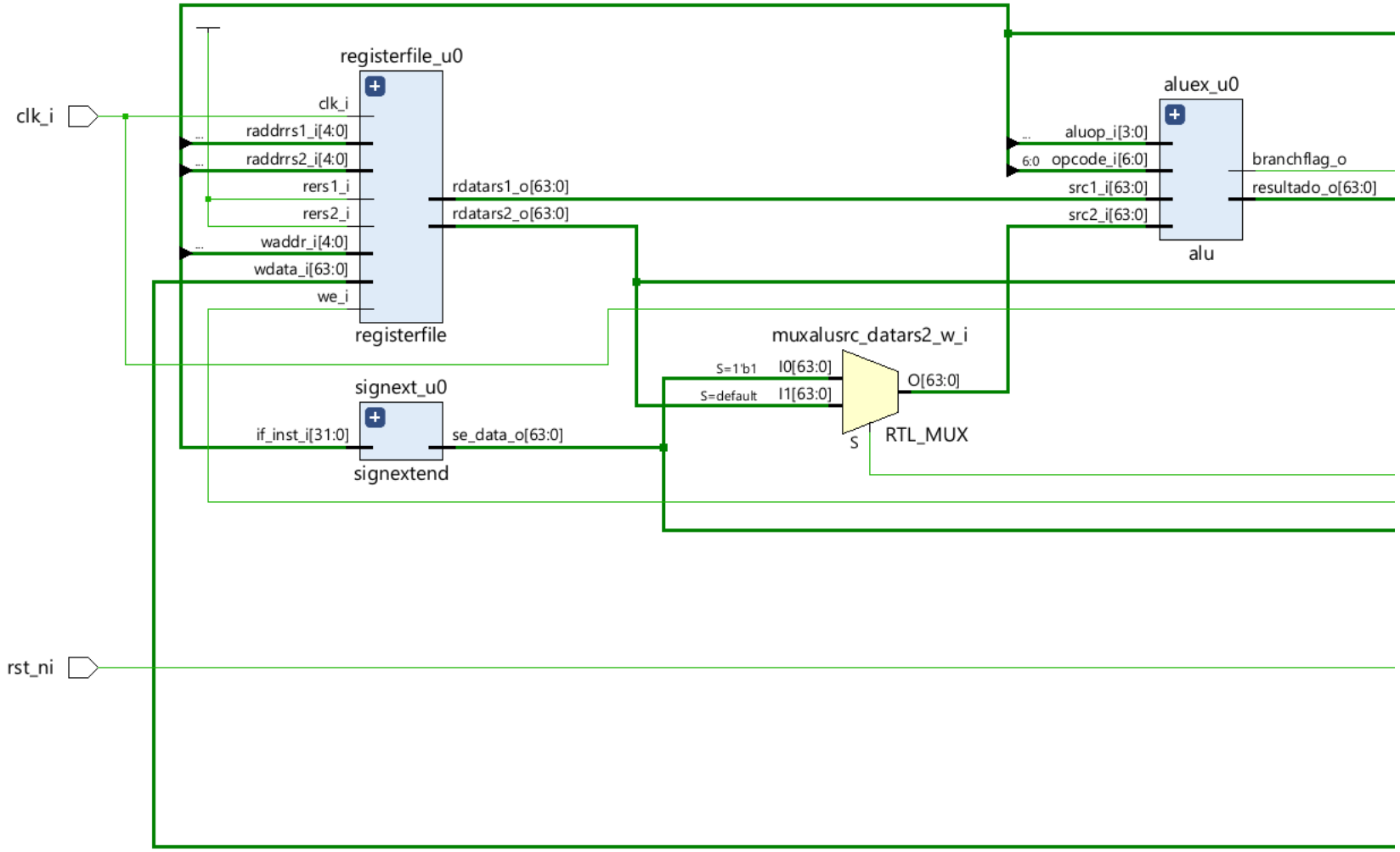
Objetivo: Implementar el procesador Monociclo siguiendo los siguientes pasos:

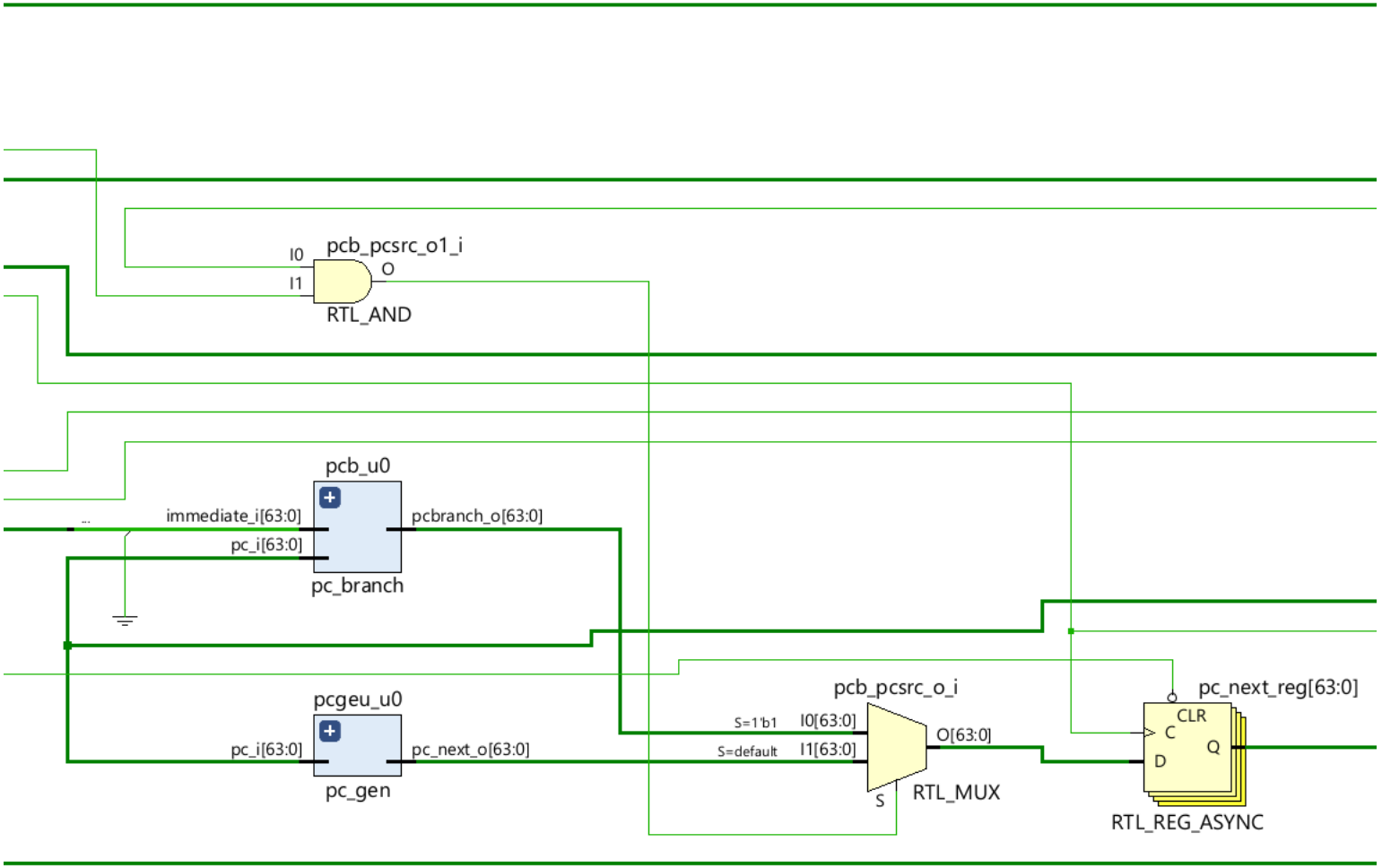
1. Localizar el archivo monociclo.qar
2. Analizar el código.
3. Compilar.
4. Obtener el RTL
5. Descargar el Archivo código de burbuja.asm incluido en la misma carpeta de material de clase.
6. Ensamblar en RARS.
7. Correr el programa y analizar su funcionamiento.
8. Generar el archivo .HEX posteriormente guardar el archivo en la carpeta de la memoria de instrucciones, icahe, y simular.
9. Examinar el correcto funcionamiento del programa en el monociclo.
10. Cambiar los números que se ordenan en el programa que ordena mediante el código de la burbuja.
11. Repetir los pasos del 6 al 9.
12. Escribir un programa en ensamblador en RARS revisando las instrucciones que fueron implementadas.
13. Repetir los pasos del 6 al 9.

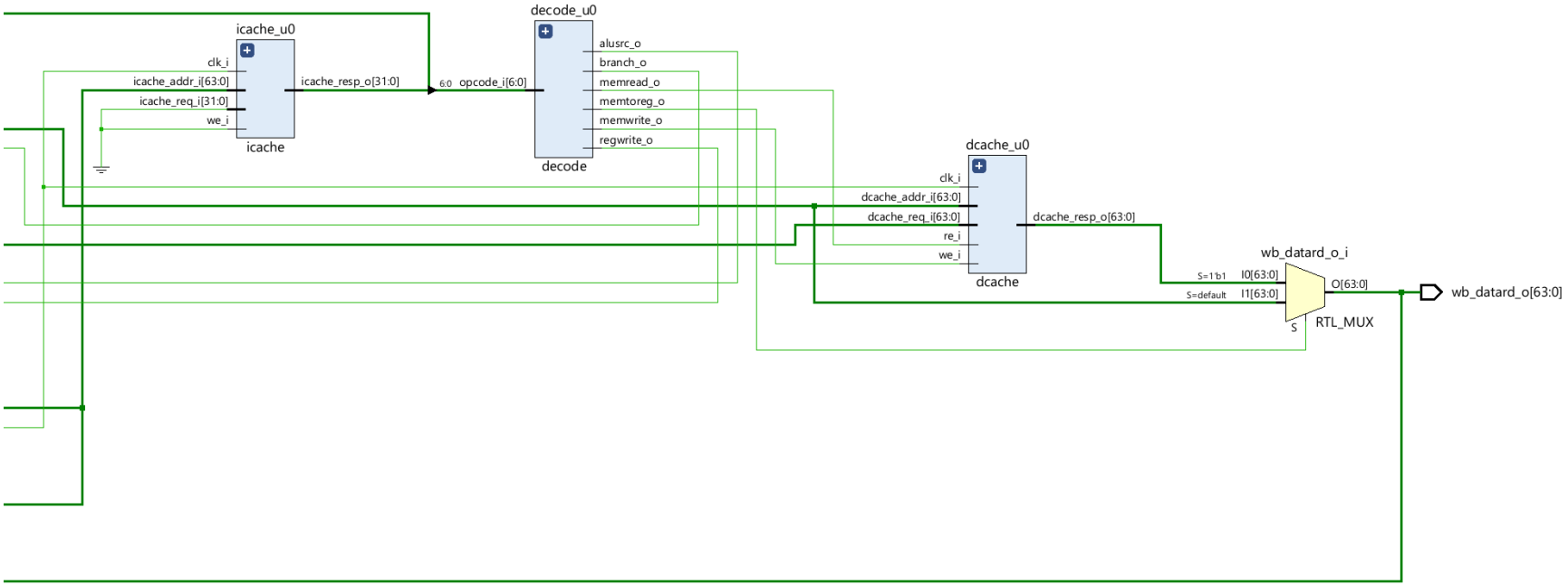
Desarrollo

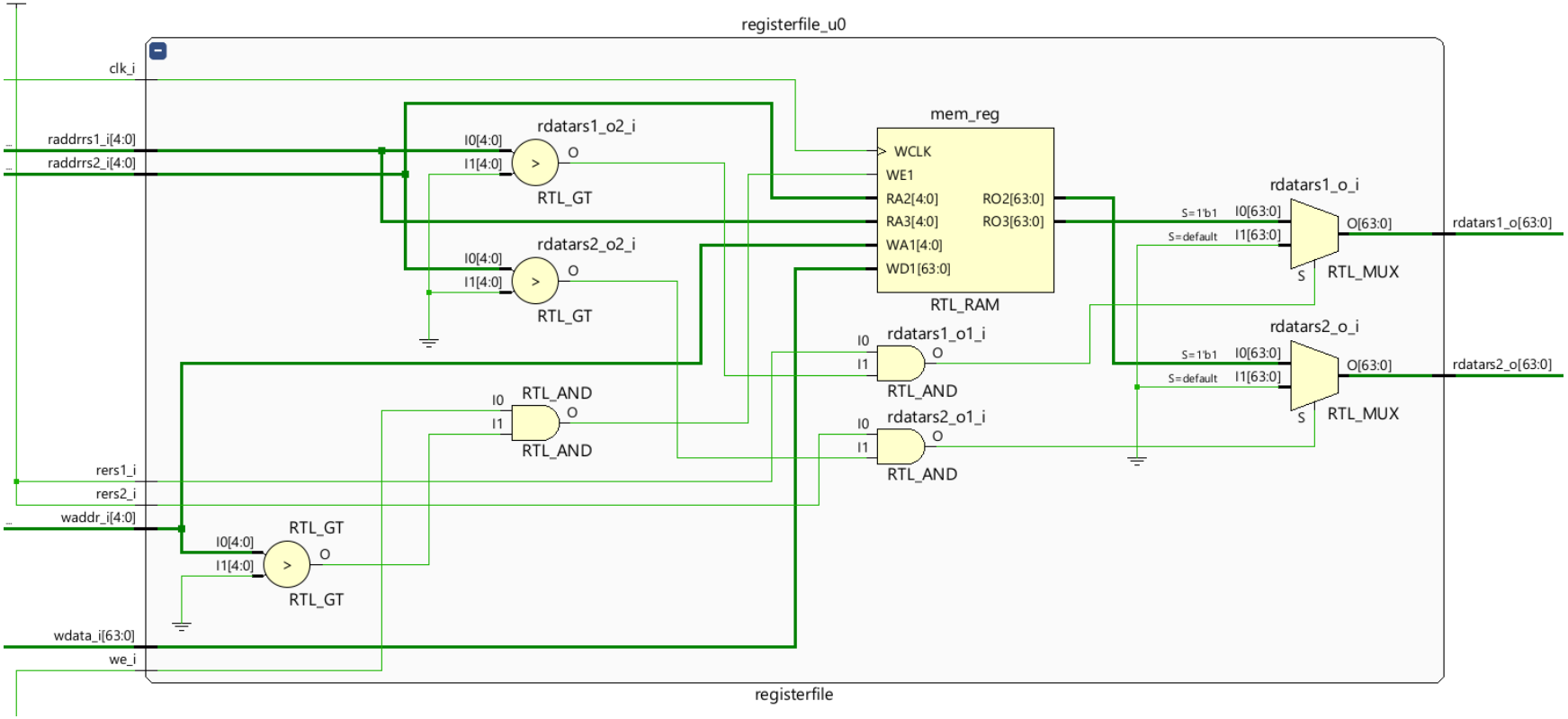
- **Descripción RTL obtenida mediante Vivado 2022.2**

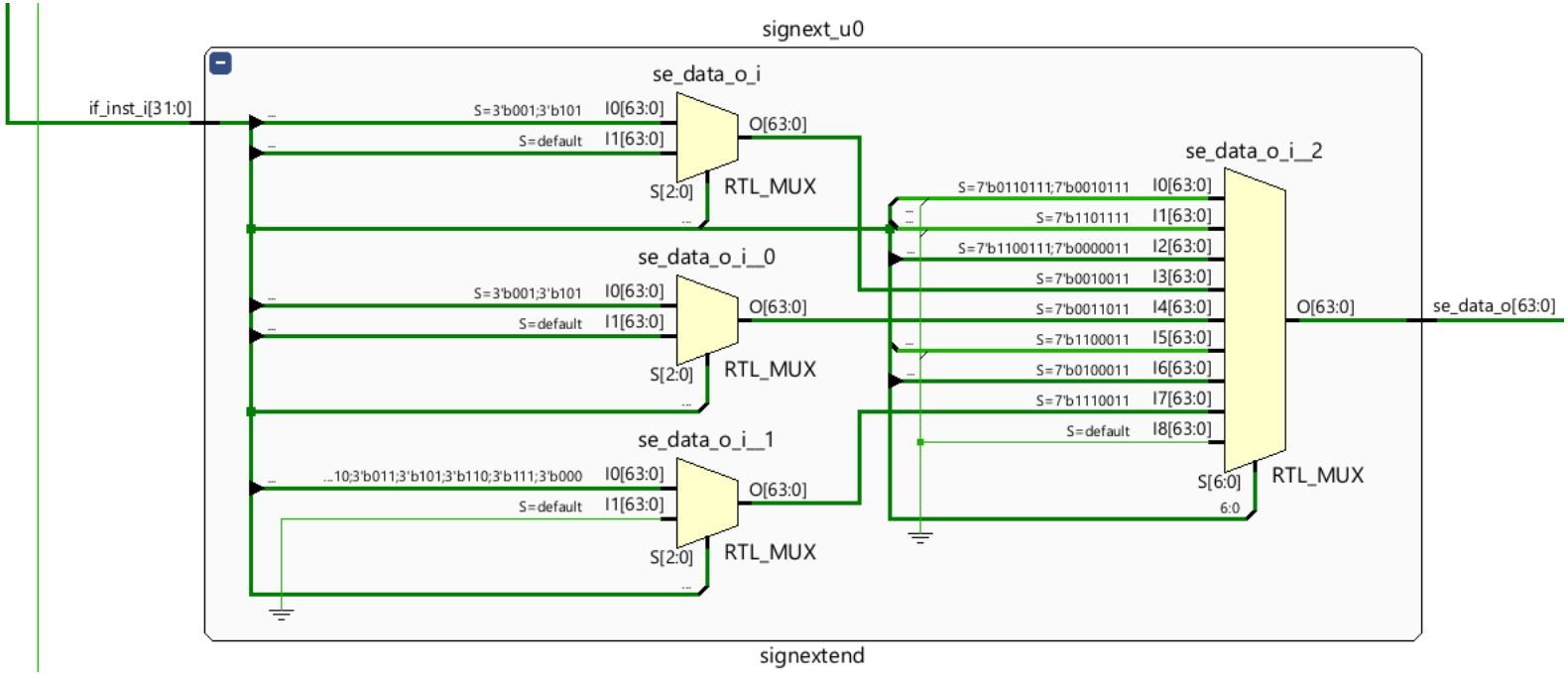


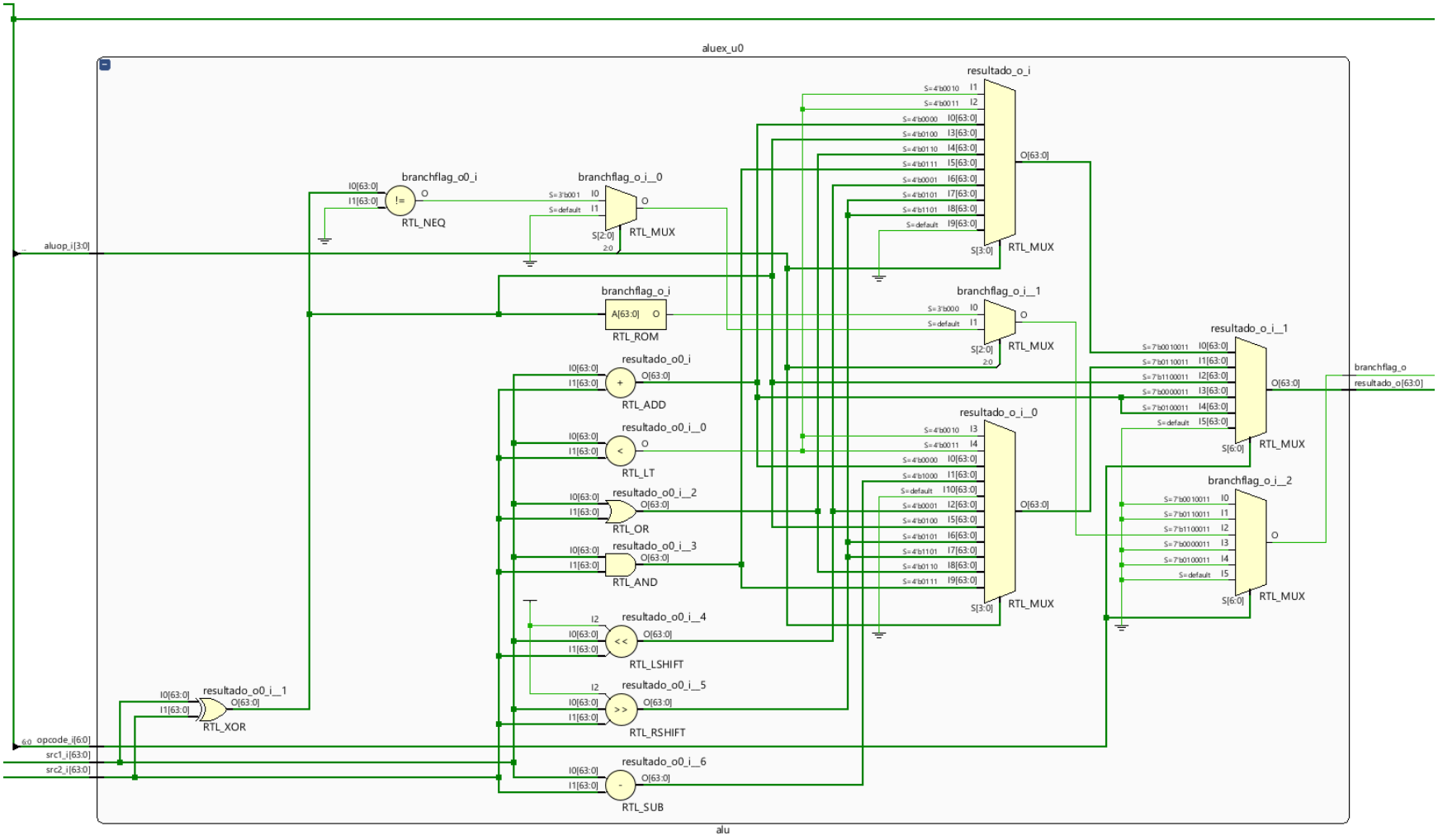

registerfile_u0
clk_i
raddrrs1_i[4:0]
raddrrs2_i[4:0]
rers1_i
rers2_i
waddr_i[4:0]
wdata_i[63:0]
we_i
rdatars1_o[63:0]
rdatars2_o[63:0]
registerfile
aluex_u0
aluop_i[3:0]
opcode_i[6:0]
src1_i[63:0]
src2_i[63:0]
branchflag_o
resultado_o[63:0]
alu
muxalusrc_datars2_w_i
S=1'b1
S=default
I0[63:0]
I1[63:0]
O[63:0]
S
RTL_MUX
signext_u0
if_inst_i[31:0]
se_data_o[63:0]
signextend
clk_i
rst_ni

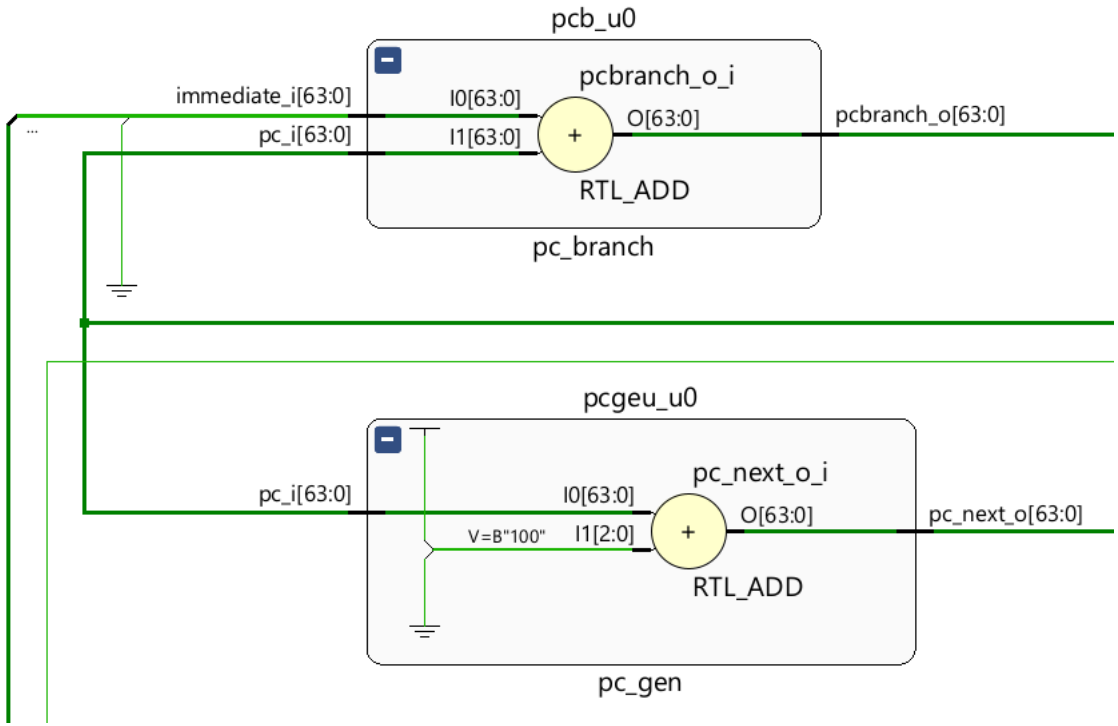

I0
I1
pcb_pcsrc_o1_i
O
RTL_AND
pcb_u0
immediate_i[63:0]
pc_i[63:0]
pcbranch_o[63:0]
pc_branch
pcgeu_u0
pc_i[63:0]
pc_next_o[63:0]
pc_gen
pcb_pcsrc_o_i
S=1'b1
S=default
I0[63:0]
I1[63:0]
O[63:0]
S
RTL_MUX
pc_next_reg[63:0]
CLR
C
D
Q
RTL_REG_ASYNC

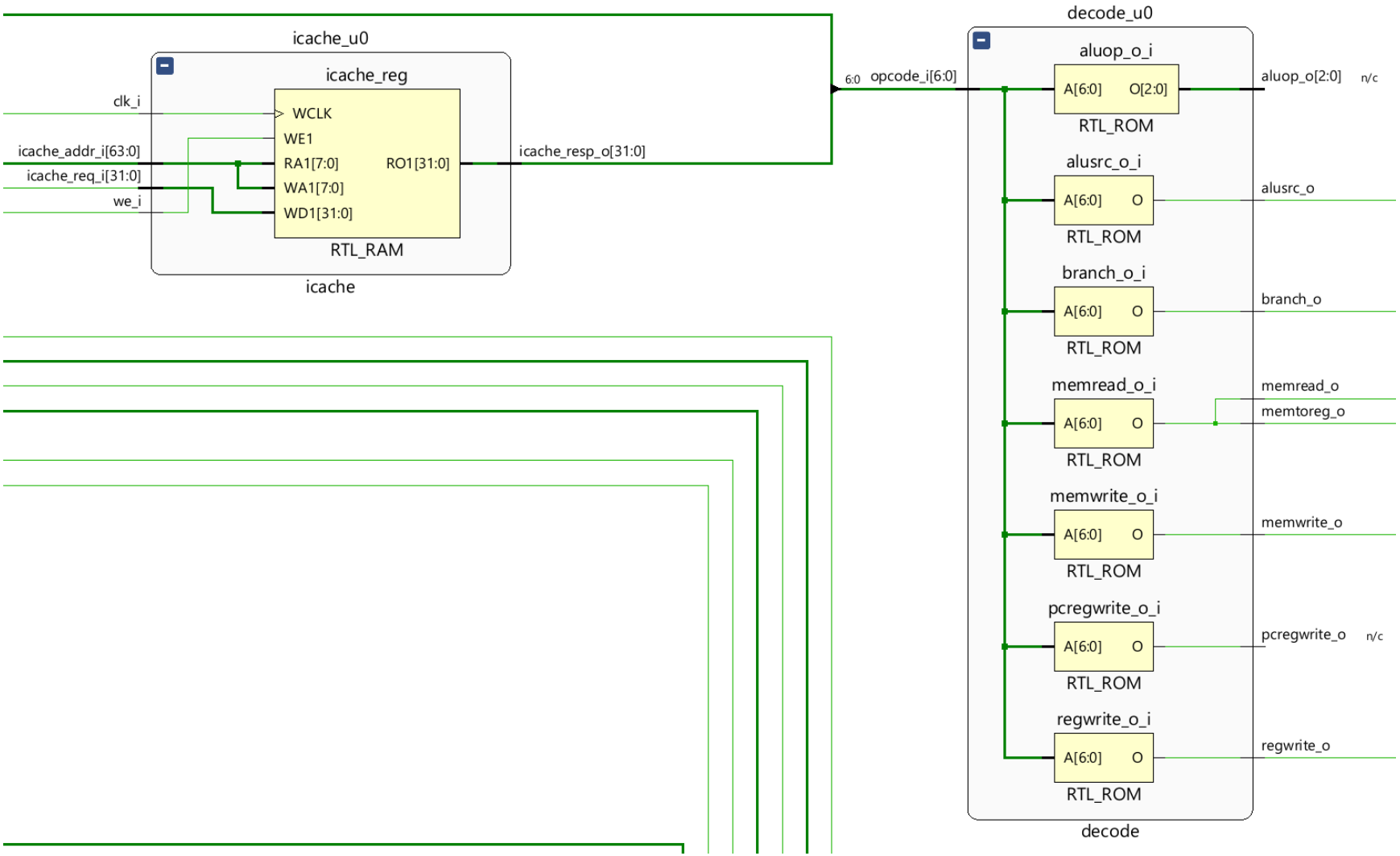

icache_u0
clk_i
icache_addr_i[63:0]
icache_req_i[31:0]
we_i
icache_resp_o[31:0]
icache
6:0
opcode_i[6:0]
decode_u0
alusrc_o
branch_o
memread_o
memtoreg_o
memwrite_o
regwrite_o
decode
dcache_u0
clk_i
dcache_addr_i[63:0]
dcache_req_i[63:0]
re_i
we_i
dcache_resp_o[63:0]
dcache
wb_datard_o_i
S=1'b1
I0[63:0]
S=default
I1[63:0]
O[63:0]
S
RTL_MUX
wb_datard_o[63:0]

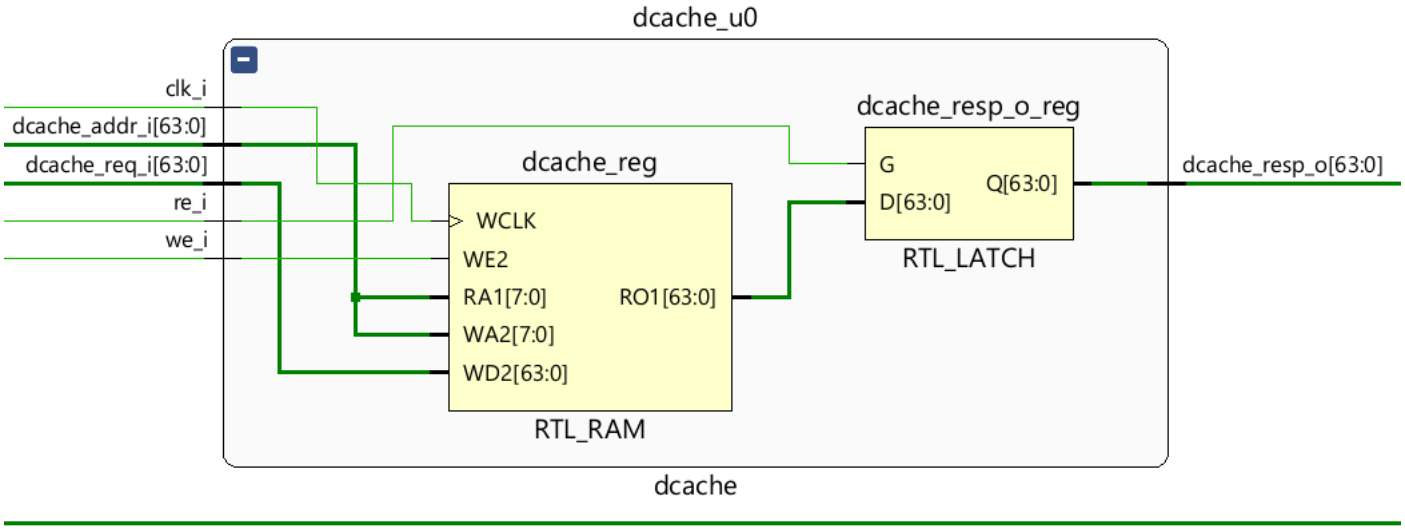

registerfile_u0
clk_i
raddrrs1_i[4:0]
raddrrs2_i[4:0]
rers1_i
rers2_i
waddr_i[4:0]
wdata_i[63:0]
we_i
rdatars1_o2_i
I0[4:0]
I1[4:0]
O
RTL_GT
rdatars2_o2_i
RTL_GT
RTL_GT
RTL_GT
RTL_AND
RTL_AND
mem_reg
WCLK
WE1
RA2[4:0]
RA3[4:0]
WA1[4:0]
WD1[63:0]
RO2[63:0]
RO3[63:0]
RTL_RAM
rdatars1_o1_i
I0
I1
RTL_AND
rdatars2_o1_i
RTL_AND
rdatars1_o_i
S=1'b1
S=default
I0[63:0]
I1[63:0]
O[63:0]
S
RTL_MUX
rdatars1_o[63:0]
rdatars2_o_i
RTL_MUX
rdatars2_o[63:0]
registerfile


signext_u0
se_data_o_i
if_inst_i[31:0]
S=3'b001;3'b101
I0[63:0]
S=default
I1[63:0]
O[63:0]
S[2:0]
RTL_MUX
se_data_o_i__0
S=3'b001;3'b101
I0[63:0]
S=default
I1[63:0]
O[63:0]
S[2:0]
RTL_MUX
se_data_o_i__1
...10;3'b011;3'b101;3'b110;3'b111;3'b000
I0[63:0]
S=default
I1[63:0]
O[63:0]
S[2:0]
RTL_MUX
se_data_o_i__2
S=7'b0110111;7'b0010111
I0[63:0]
S=7'b1101111
I1[63:0]
S=7'b1100111;7'b0000011
I2[63:0]
S=7'b0010011
I3[63:0]
S=7'b0011011
I4[63:0]
S=7'b1100011
I5[63:0]
S=7'b0100011
I6[63:0]
S=7'b1110011
I7[63:0]
S=default
I8[63:0]
S[6:0]
6:0
RTL_MUX
O[63:0]
se_data_o[63:0]
signextend


aluex_u0
alu
aluop_i[3:0]
opcode_i[6:0]
src1_i[63:0]
src2_i[63:0]
branchflag_o
resultado_o[63:0]
RTL_NEQ
RTL_MUX
RTL_ROM
RTL_ADD
RTL_LT
RTL_OR
RTL_AND
RTL_LSHIFT
RTL_RSHIFT
RTL_XOR
RTL_SUB
branchflag_o0_i
branchflag_o_i__0
branchflag_o_i
branchflag_o_i__1
branchflag_o_i__2
resultado_o_i
resultado_o_i__0
resultado_o_i__1
resultado_o0_i
resultado_o0_i__0
resultado_o0_i__2
resultado_o0_i__3
resultado_o0_i__4
resultado_o0_i__5
resultado_o0_i__6
resultado_o0_i__1


pcb_u0
pcbranch_o_i
immediate_i[63:0]
I0[63:0]
O[63:0]
pcbranch_o[63:0]
pc_i[63:0]
I1[63:0]
+
RTL_ADD
pc_branch
pcgeu_u0
pc_next_o_i
pc_i[63:0]
I0[63:0]
O[63:0]
pc_next_o[63:0]
V=B"100"
I1[2:0]
+
RTL_ADD
pc_gen


icache_u0
icache_reg
clk_i
WCLK
WE1
icache_addr_i[63:0]
RA1[7:0]
RO1[31:0]
icache_req_i[31:0]
WA1[7:0]
we_i
WD1[31:0]
RTL_RAM
icache
icache_resp_o[31:0]
6:0
opcode_i[6:0]
decode_u0
aluop_o_i
A[6:0]
O[2:0]
RTL_ROM
aluop_o[2:0]
n/c
alusrc_o_i
A[6:0]
O
alusrc_o
branch_o_i
branch_o
memread_o_i
memread_o
memtoreg_o
memwrite_o_i
memwrite_o
pcregwrite_o_i
pcregwrite_o
regwrite_o_i
regwrite_o
decode



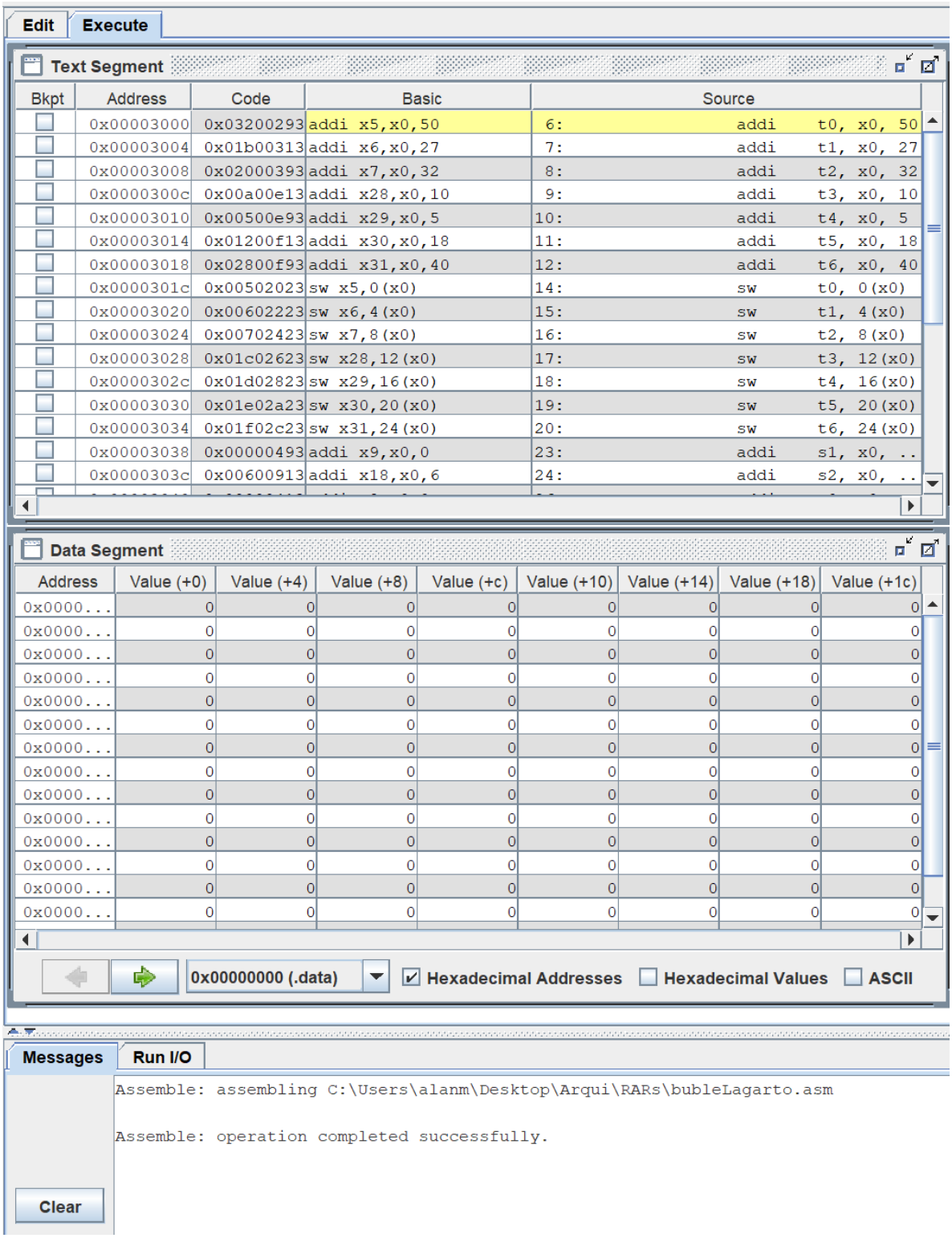
Podemos observar cómo utilizo FlipsFlops, multiplexores, operadores de suma, resta, no es igual, desplazamiento a la izquierda/derecha, menor que y mayor que, compuertas AND, OR, XOR y bloques de memoria ROM/RAM del FPGA seleccionado, y para el control del dato de salida, se implementó mediante un multiplexor.

- **Código de burbuja**

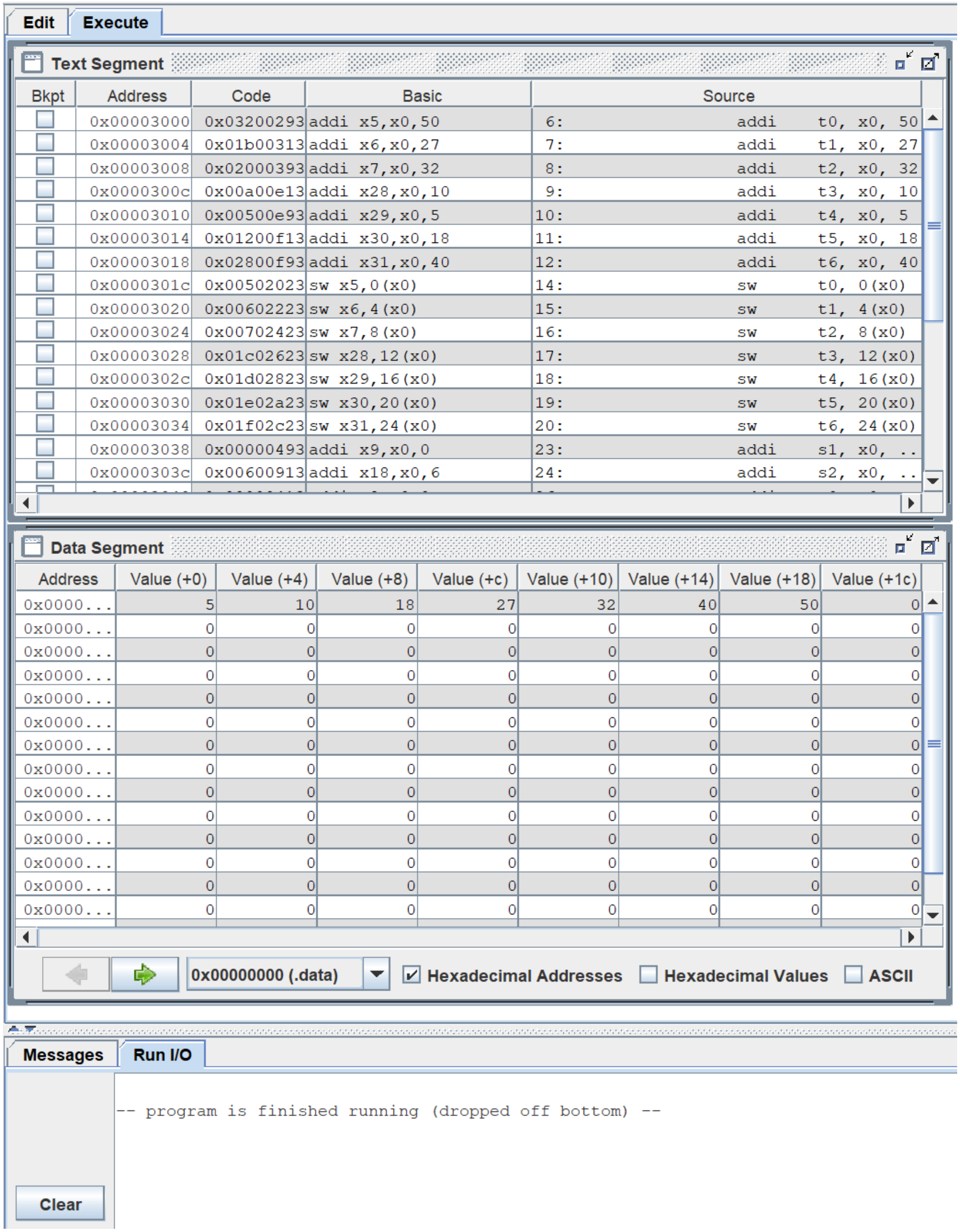
bubleLagarto.asm

```
# Alan Adrian Malagon Baeza - 5CV1
#Practica- Ordenamiento Burbuja
#Directiva de codigo
.text
#Cargar datos
                addi    t0, x0, 50
                addi    t1, x0, 27
                addi    t2, x0, 32
                addi    t3, x0, 10
                addi    t4, x0, 5
                addi    t5, x0, 18
                addi    t6, x0, 40
#cargar datos en memoria
                sw      t0, 0(x0)
                sw      t1, 4(x0)
                sw      t2, 8(x0)
                sw      t3, 12(x0)
                sw      t4, 16(x0)
                sw      t5, 20(x0)
                sw      t6, 24(x0)

#Buble externo
                addi    s1, x0, 0               #s1 = indice i
                addi    s2, x0, 6               #N = 6
for_i:
                addi    s0, x0, 0               #s0 = indice j
#for (j=0; j<N; j++)
for_j:
#t0 = j
#t1 = j + 1
                slli    t0, s0, 2               #direccion alinead j
                addi    t1, t0, 4               #Direccion alineada j+1
                lw      t2, 0(t0)               #t2 = Mem[j]
                lw      t3, 0(t1)               #t3 = Mem[j+1]
                slt     t4, t3, t2              #(t3 < t2)
                beq     t4, x0, fin_if          #salta a fin_if si t4 = 1
                sw      t3, 0(t0)
                sw      t2, 0(t1)
#fin del if
fin_if:
                addi    s0, s0, 1               #j = j + 1
                bne     s0, s2, for_j           #Si no hemos alcanzado el limite

                addi    s1, s1, 1
                bne     s1, s2, for_i
```

- **Ensamblando el programa**



- **Resultado de la simulación en RARS**



Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0
ra	1	0
sp	2	12284
gp	3	6144
tp	4	0
t0	5	20
t1	6	24
t2	7	40
s0	8	6
s1	9	6
a0	10	0
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0
s2	18	6
s3	19	0
s4	20	0
s5	21	0
s6	22	0
s7	23	0
s8	24	0
s9	25	0
s10	26	0
s11	27	0
t3	28	50
t4	29	0
t5	30	18
t6	31	40
pc		12408

El programa es una implementación del algoritmo de ordenamiento de burbuja en lenguaje ensamblador RISC-V. A continuación, se analiza su funcionamiento paso a paso:

1. Cargar datos: Se cargan los valores iniciales en los registros `t0` a `t6`.

2. Cargar datos en memoria: Los valores se almacenan en la memoria a partir de la dirección `0(x0)` en incrementos de 4 bytes (ya que son enteros).

3. Bucle externo (`for_i`): El registro `s1` se inicializa en 0 y representa el índice `i` del bucle externo. Este bucle se ejecutará `N` veces (en este caso, `N` es 6).

4. Bucle interno (`for_j`): El registro `s0` se inicializa en 0 y representa el índice `j` del bucle interno. Este bucle se ejecutará `N` veces (`N` es 6).

5. Comparación y intercambio: Se cargan los valores de `Mem[j]` y `Mem[j+1]` de la memoria en los registros `t2` y `t3`, respectivamente. Luego, se compara si `t3` es menor que `t2` utilizando la instrucción `slt` (set less than). Si la comparación es verdadera, se intercambian los valores en la memoria.

6. Fin del bucle interno: Se incrementa `s0` en 1 (para pasar al siguiente valor de `j`) y se verifica si se ha alcanzado el límite (`s2`). Si no se ha alcanzado el límite, se repite el bucle interno (`for_j`).

7. Fin del bucle externo: Se incrementa `s1` en 1 (para pasar al siguiente valor de `i`) y se verifica si se ha alcanzado el límite (`s2`). Si no se ha alcanzado el límite, se repite el bucle externo (`for_i`).

En resumen, el programa utiliza dos bucles anidados para comparar y ordenar los valores almacenados en memoria utilizando el algoritmo de ordenamiento de burbuja. El bucle externo controla el índice `i` y el bucle interno controla el índice `j`, realizando las comparaciones y los intercambios necesarios para ordenar los valores de manera ascendente.

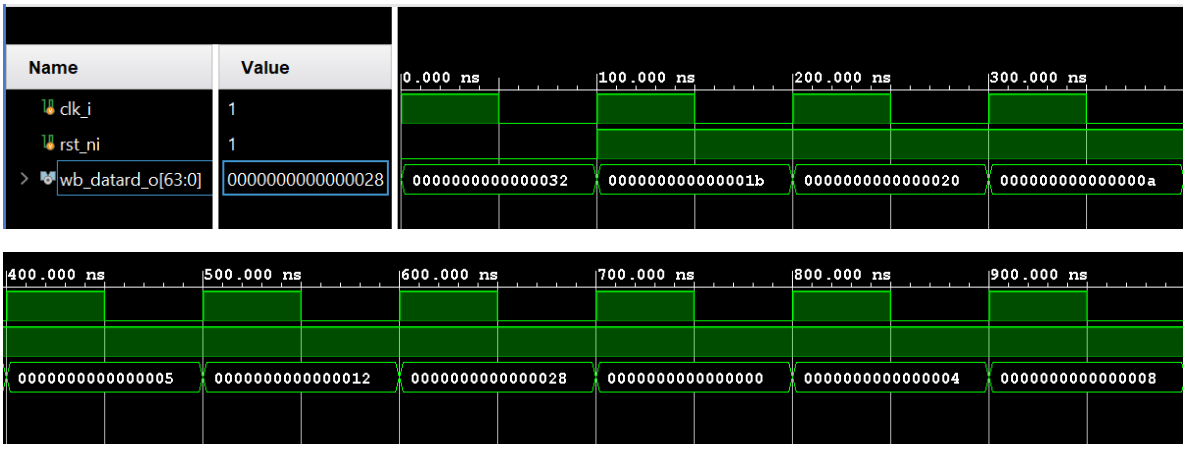
- **Generando archivo .HEX**

burbuja.mem

C:/Users/alanm/Desktop/Arqui/Vivado/monociclo/monociclo.srcs/sources_1/new/burbuja.mem

```
1  03200293
2  01b00313
3  02000393
4  00a00e13
5  00500e93
6  01200f13
7  02800f93
8  00502023
9  00602223
10 00702423
11 01c02623
12 01d02823
13 01e02a23
14 01f02c23
15 00000493
16 00600913
17 00000413
18 00241293
19 00428313
20 0002a383
21 00032e03
22 007e2eb3
23 000e8663
24 01c2a023
25 00732023
26 00140413
27 fd241ee3
28 00148493
29 fd2498e3
30
```

- **Resultado de la simulación en Vivado**



clk_i	1
rst_ni	0
FETCH - IF	
pcb_pcsrc_o	00000000000000000000000000000000000000000000000000000000000000000100
pc_next	00000000000000000000000000000000000000000000000000000000000000000000
pc_gen	00000000000000000000000000000000000000000000000000000000000000000100

CACHE DE INSTRUCCIONES - IL1		
icache_addr_i	pc_next	
we_i	0	
icache_req_i	00000000000000000000000000000000	hex mem
icache_inst_o	00000000000000000000000000000000001100100000000001010010011	3200293

DECODIFICACION/CONTROL - ID		
id_opcode_w	0010011	INSTRUCCIÓN TIPO I
id_addrd_w	00101	
id_aluop_w	0000	
id_addrs1_w	00000	
id_addrs2_w	10010	
id_alusrc_o	1	
id_regwrite_o	1	
id_memread_o	0	
id_memwrite_o	0	
id_branch_o	0	
id_memtoreg_o	0	
id_aluop_o	010	
id_pcregwrite_o	0	

BUSQUEDA DE OPERANDOS/LECTURAS DE REGISTROS - RR		
rers1_i	1	HAB DE LECTURA
raddrrs1_i	id_addrs1_w	DIR DE LECTURA
rdatars1_o	00000000000000000000000000000000000000000000000000000000000000000000	DATO DE LECTURA
rers2_i	1	HAB DE LECTURA
raddrrs2_i	id_addrs2_w	DIR DE LECTURA
rdatars2_o	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	DATO DE LECTURA
we_i	id_regwrite_o	HAB DE ESCRITURA
waddr_i	id_addrd_w	DIR DE ESCRITURA
wb_datard_o	00000000000000000000000000000000000000000000000000000000000110010	DATO DE ESCRITURA

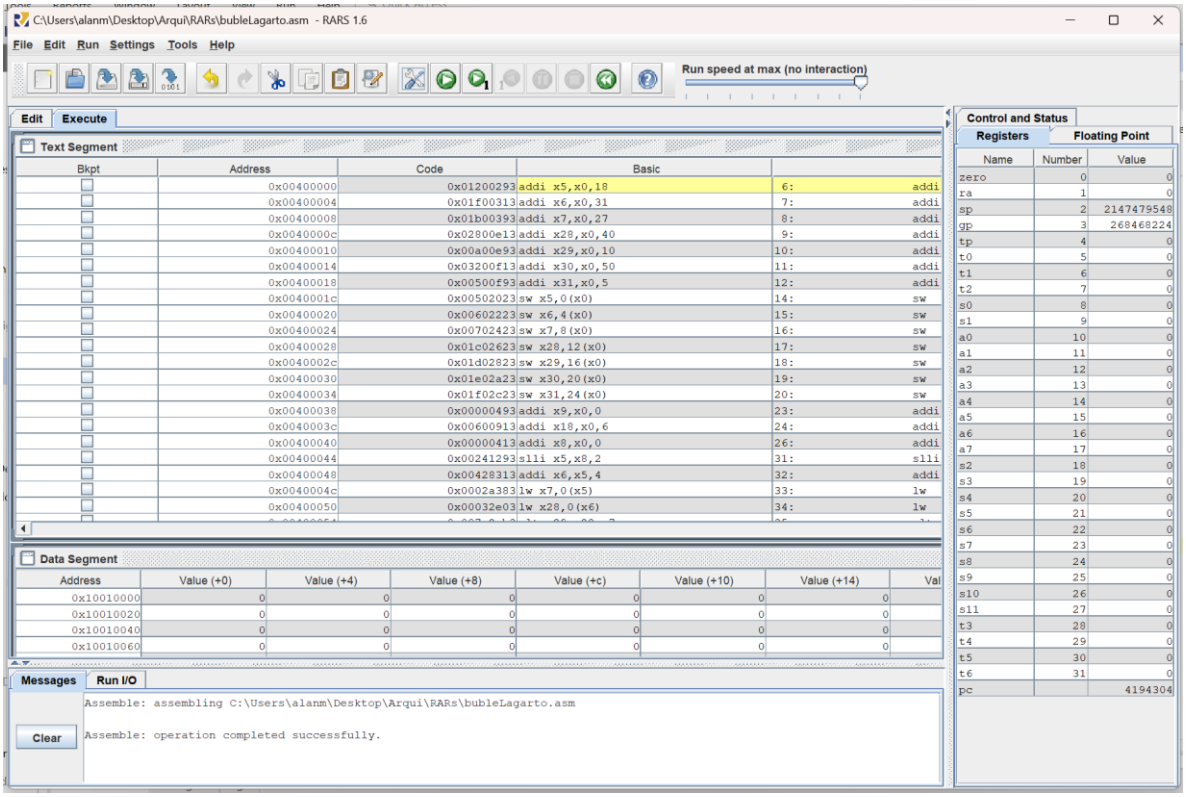
EXTENSION DE SIGNO - SE		
if_inst_i	icache_inst_o	
se_datars2_o	00000000000000000000000000000000000000000000000000000000000110010	32

- **Código de burbuja modificado**

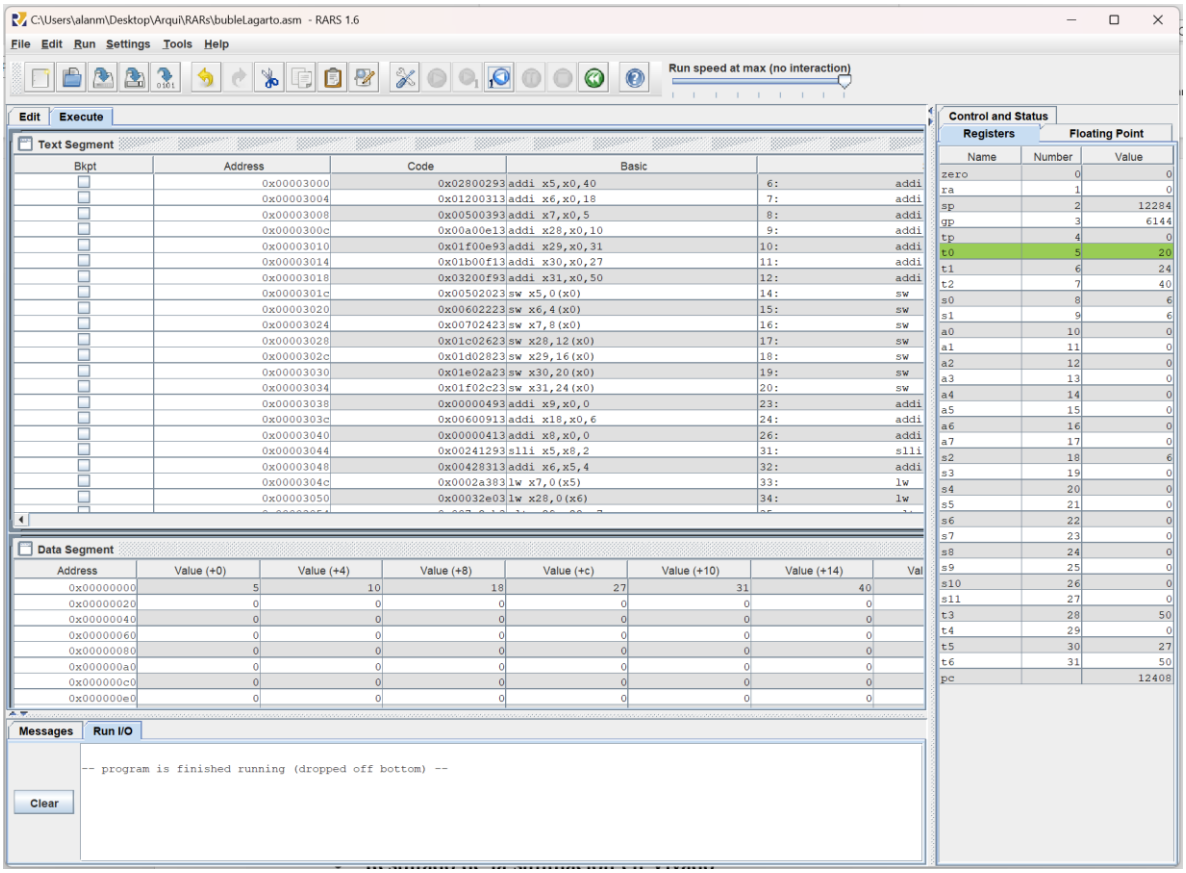
bubleLagarto.asm

```
# Alan Adrian Malagon Baeza - 5CV1
#Practica- Ordenamiento Burbuja
#Directiva de codigo
.text
#Cargar datos 50 27 31 10 5 18 40
                addi    t0, x0, 40
                addi    t1, x0, 18
                addi    t2, x0, 5
                addi    t3, x0, 10
                addi    t4, x0, 31
                addi    t5, x0, 27
                addi    t6, x0, 50
#cargar datos en memoria
                sw      t0, 0(x0)
                sw      t1, 4(x0)
                sw      t2, 8(x0)
                sw      t3, 12(x0)
                sw      t4, 16(x0)
                sw      t5, 20(x0)
                sw      t6, 24(x0)

#Buble externo
                addi    s1, x0, 0               #s1 = indice i
                addi    s2, x0, 6               #N = 6
for_i:
                addi    s0, x0, 0               #s0 = indice j
#for (j=0; j<N; j++)
for_j:
#t0 = j
#t1 = j + 1
                slli    t0, s0, 2               #direccion alinead j
                addi    t1, t0, 4               #Direccion alineada j+1
                lw      t2, 0(t0)               #t2 = Mem[j]
                lw      t3, 0(t1)               #t3 = Mem[j+1]
                slt     t4, t3, t2              #(t3 < t2)
                beq     t4, x0, fin_if          #salta a fin_if si t4 = 1
                sw      t3, 0(t0)
                sw      t2, 0(t1)
#fin del if
fin_if:
                addi    s0, s0, 1               #j = j + 1
                bne     s0, s2, for_j           #Si no hemos alcanzado el limite

                addi    s1, s1, 1
                bne     s1, s2, for_i
```

- **Ensamblando el programa**



- **Resultado de la simulación en RARS**



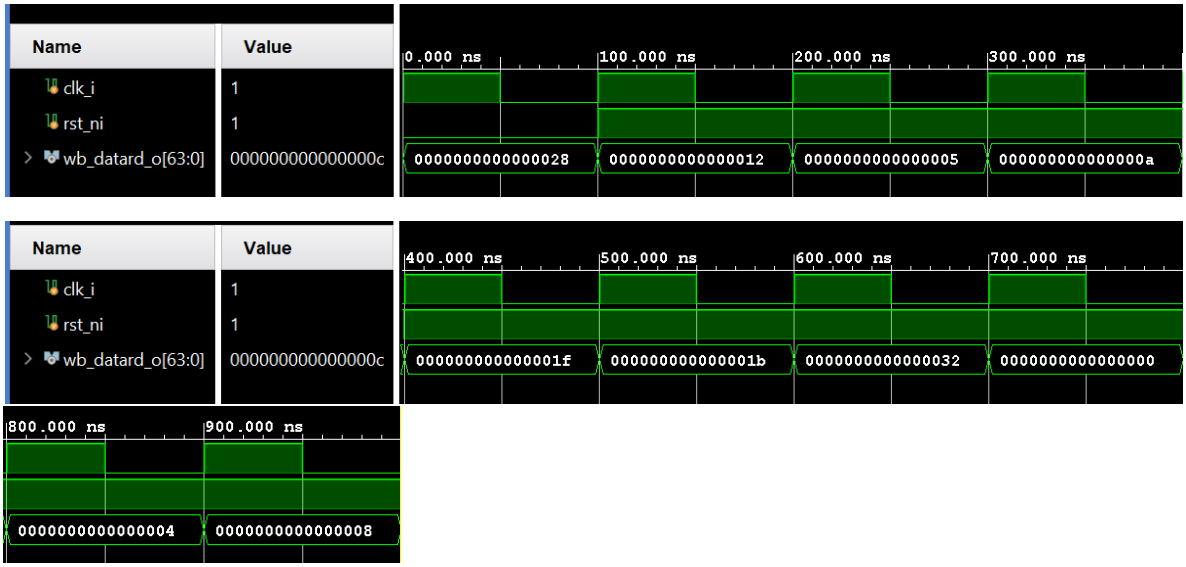
- **Generando archivo .HEX**

burbujam.mem

C:/Users/alanm/Desktop/Arqui/Vivado/monociclo/monociclo.srcs/sources_1/new/burbujam.mem

```
1  02800293
2  01200313
3  00500393
4  00a00e13
5  01f00e93
6  01b00f13
7  03200f93
8  00502023
9  00602223
10 00702423
11 01c02623
12 01d02823
13 01e02a23
14 01f02c23
15 00000493
16 00600913
17 00000413
18 00241293
19 00428313
20 0002a383
21 00032e03
22 007e2eb3
23 000e8663
24 01c2a023
25 00732023
26 00140413
27 fd241ee3
28 00148493
29 fd2498e3
```

- **Resultado de la simulación en Vivado**



- **Código propuesto: Ordenamiento por Inserción**

insercion.asm

```
# Alan Adrian Malagon Baeza - 5CV1
# Práctica - Ordenamiento por Inserción
# Directiva de código
.text
# Cargar datos 50 27 31 10 5 18 40
addi t0, x0, 40
addi t1, x0, 18
addi t2, x0, 5
addi t3, x0, 10
addi t4, x0, 31
addi t5, x0, 27
addi t6, x0, 50

# Cargar datos en memoria
sw t0, 0(x0)
sw t1, 4(x0)
sw t2, 8(x0)
sw t3, 12(x0)
sw t4, 16(x0)
sw t5, 20(x0)
sw t6, 24(x0)

# Ordenamiento por inserción
addi s1, x0, 1   # s1 = índice i
addi s2, x0, 7   # N = 7

for_i:
    addi t0, s1, 0      # t0 = i
    addi t1, t0, -1     # t1 = i - 1
    slli t0, t0, 2      # dirección alineada i
    slli t1, t1, 2      # dirección alineada i - 1
    lw t2, 0(t0)        # t2 = Mem[i]

    for_j:
        beqz t1, done       # Si hemos alcanzado el inicio del arreglo, terminamos
        lw t3, 0(t1)        # t3 = Mem[i - 1]
        slt t4, t3, t2     # (t3 < t2)
        bnez t4, shift      # Si t3 < t2, saltamos a shift

        sw t3, 0(t0)        # Mem[i] = Mem[i - 1]
        addi t0, t0, -4     # Decrementamos la dirección de i
        addi t1, t1, -4     # Decrementamos la dirección de i - 1
        j for_j             # Volvemos a comprobar el siguiente elemento

```

```
45      shift:
46          sw t2, 0(t0)        # Mem[i] = t2
47          j next_i             # Pasamos al siguiente elemento del arreglo
48
49      done:
50          addi s1, s1, 1       # Incrementamos el índice i
51          j for_i              # Volvemos a iterar para el siguiente elemento
52
53  next_i:
54
55  # Resultado ordenado almacenado en memoria
56
```

Se encuentran las instrucciones para cargar los datos en registros temporales (t0-t6). En este caso, se cargan los valores 50, 27, 31, 10, 5, 18 y 40. Estos valores pueden modificarse según sea necesario.

Después de cargar los datos en los registros temporales, se utilizan las instrucciones "sw" para almacenar los valores en la memoria. Cada valor se almacena en una ubicación de memoria específica utilizando un desplazamiento relativo a la dirección base (x0).

Una vez que los datos se han cargado y almacenado en memoria, comienza el algoritmo de ordenamiento por inserción.

El algoritmo utiliza dos bucles "for" anidados para iterar a través de los elementos del arreglo y realizar las comparaciones necesarias para ordenarlos.

El bucle externo "for_i" se encarga de iterar sobre todos los elementos del arreglo. Se utiliza una variable de índice "s1" para rastrear la posición actual del elemento que se está comparando.

Dentro del bucle externo, se inicializa la variable de índice "s0" en cero para el bucle interno "for_j". El bucle "for_j" se encarga de buscar la posición correcta para el elemento actual dentro de la porción ya ordenada del arreglo.

Dentro del bucle "for_j", se cargan los valores del elemento actual (t2) y el elemento anterior (t3) en registros temporales. Se realiza una comparación (t3 < t2) para determinar si el elemento anterior es menor al elemento actual.

Si la comparación es verdadera, se ejecuta un bloque de código donde se intercambian los valores de los elementos. Primero, se almacena el valor del elemento anterior (t3) en la posición del elemento actual (Mem[i]). Luego, se

decrementan las direcciones de memoria para comparar el siguiente par de elementos en el siguiente ciclo del bucle "for_j".

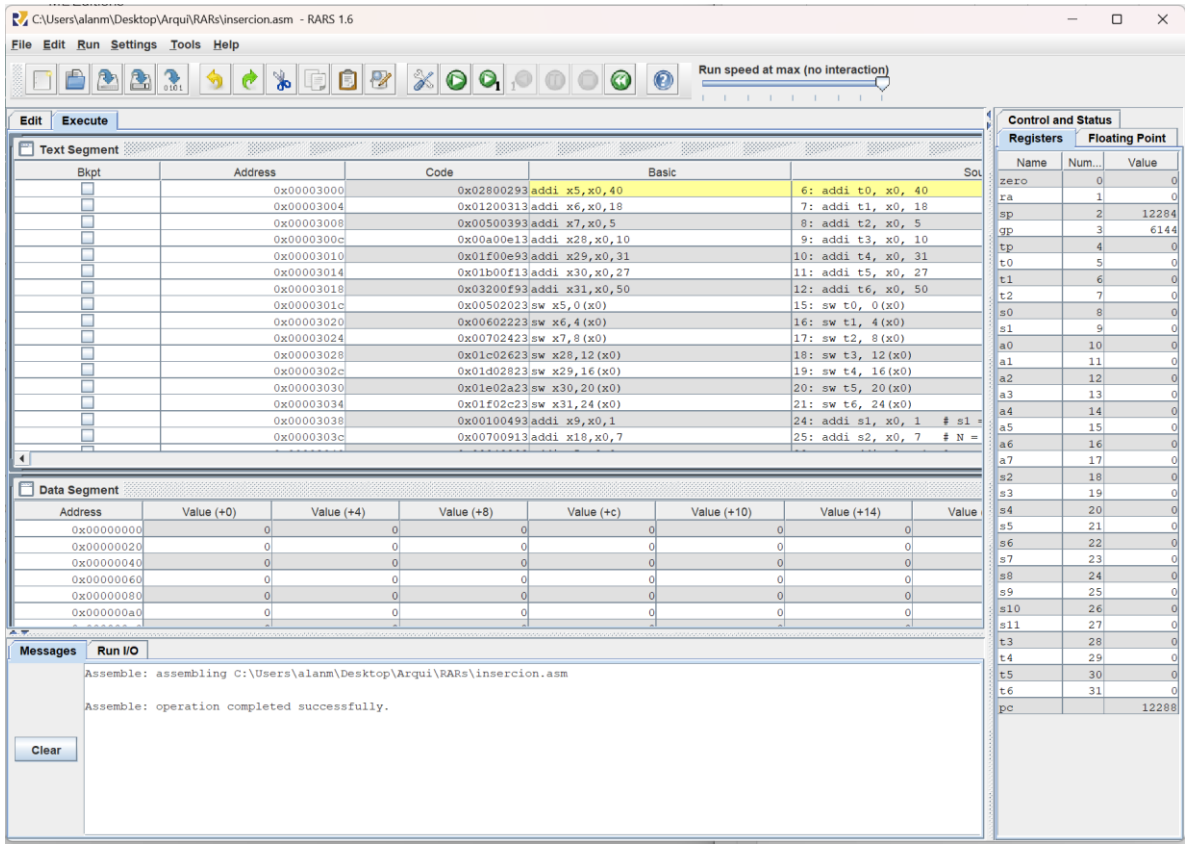
Si la comparación es falsa, el bucle "for_j" se salta el bloque de código de intercambio y pasa al siguiente par de elementos.

Una vez que el bucle "for_j" ha terminado de iterar sobre todos los elementos en la porción ya ordenada del arreglo, se almacena el valor del elemento actual (t2) en la posición correcta dentro del arreglo.

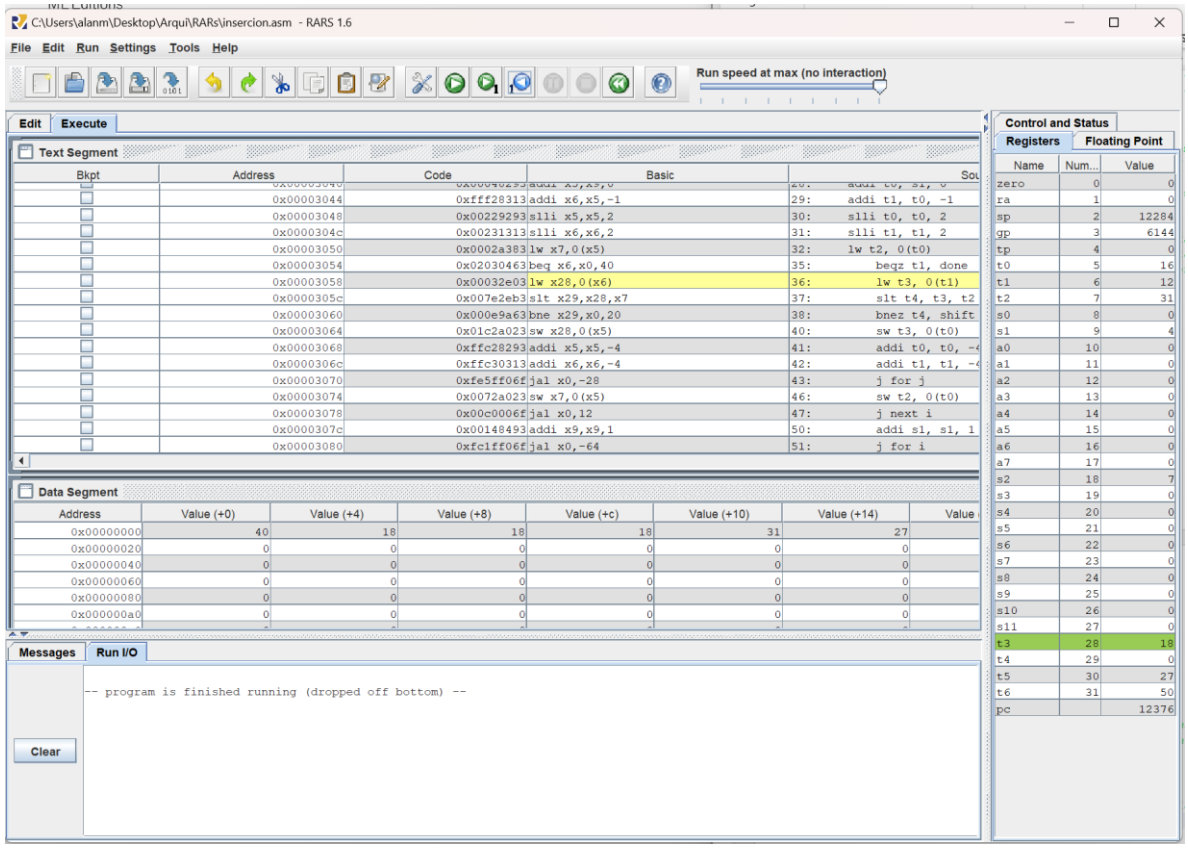
Después de eso, se incrementa el índice "s1" en el bucle externo "for_i" para pasar al siguiente elemento del arreglo y repetir el proceso.

El resultado final será un arreglo ordenado almacenado en memoria.

- **Ensamblando el programa**



- **Resultado de la simulación en RARS**



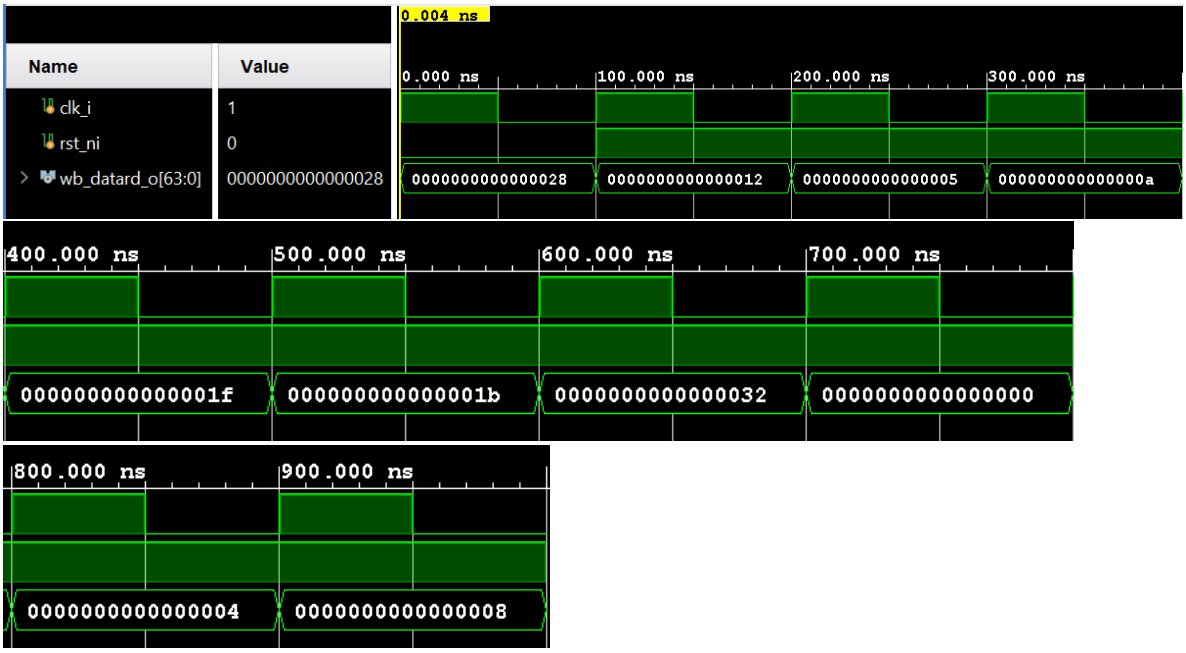
- **Generando archivo .HEX**

insercion.mem

C:/Users/alanm/Desktop/Arqui/Vivado/monociclo/monociclo.srcs/sources_1/new/insercion.mem

```
02800293
01200313
00500393
00a00e13
01f00e93
01b00f13
03200f93
00502023
00602223
00702423
01c02623
01d02823
01e02a23
01f02c23
00100493
00700913
00048293
fff28313
00229293
00231313
0002a383
02030463
00032e03
007e2eb3
000e9a63
01c2a023
ffc28293
ffc30313
fe5ff06f
0072a023
00c0006f
00148493
fc1ff06f
```

- **Resultado de la simulación en Vivado**



Conclusión

En esta actividad, se implementó y probó un procesador monociclo siguiendo una serie de pasos específicos. Durante el desarrollo del proyecto, se analizó el código fuente, se compiló y se obtuvo el RTL del procesador. Además, se ensambló un programa utilizando RARS, se ejecutó y se examinó su funcionamiento.

El proceso de implementación y prueba del procesador monociclo proporcionó una valiosa experiencia práctica en el diseño y funcionamiento de los procesadores. Se adquirieron conocimientos sobre el proceso de compilación, ensamblaje y simulación, así como sobre la interacción del procesador con la memoria de instrucciones.

Al modificar el programa para ordenar diferentes números utilizando el algoritmo de ordenamiento de burbuja, se pudo observar cómo las instrucciones del procesador afectan el resultado final y el rendimiento del programa. Esta experiencia también permitió comprender la importancia de la optimización de algoritmos y cómo las instrucciones del procesador pueden influir en ello.

En resumen, la implementación y prueba del procesador monociclo a través de los pasos propuestos proporcionó una comprensión más profunda de los conceptos teóricos relacionados con los procesadores y su funcionamiento. Esta actividad práctica permitió aplicar los conocimientos adquiridos, adquirir habilidades en el uso de herramientas de desarrollo y simulación, y obtener una visión más completa del diseño y rendimiento de los procesadores.

En conclusión, este proyecto fue una oportunidad enriquecedora para aprender y experimentar con el diseño y funcionamiento de un procesador monociclo. Los conocimientos adquiridos y las habilidades desarrolladas durante este proceso son valiosos para comprender y trabajar en el campo de la arquitectura de procesadores y la programación de bajo nivel.

Referencia

1. Brock J. LaMeres, Introduction to Logic Circuits & Logic Design with Verilog, Springer, 1st Edition, USA, 2017.
2. TheThirdOne. (n.d.). GitHub - TheThirdOne/rars: RARS -- RISC-V Assembler and Runtime Simulator. GitHub. https://github.com/TheThirdOne/rars
3. GeeksforGeeks. (2023). Insertion Sort Data Structure and Algorithm Tutorials.

   GeeksforGeeks. https://www.geeksforgeeks.org/insertion-sort/