



INSTITUTO POLITÉCNICO NACIONAL

MAESTRÍA EN CIENCIAS EN SISTEMAS COMPUTACIONALES
MÓVILES

*Proyecto de la Asignatura de Arquitectura de
Dispositivos Móviles*

*Implementación en VHDL de un
Microporcesador LC3-básico*

Elaborado por:

Juan José Guzmán Cruz

Profesor:

Dr. David Ernesto Troncoso Romero

12 de junio de 2018

Contenido

1. Especificaciones	1
2. ALU	1
2.1. Descripción de la Unidad Aritmética-Lógica (ALU)	1
2.2. Simulación	2
2.3. Resumen de síntesis	7
3. Banco de registros	9
3.1. Descripción del banco de 8 registros de 16 bits	9
3.2. Simulación	13
3.3. Resumen de síntesis	17
4. Condition codes	20
4.1. Descripción del condition codes	20
4.2. Simulación	22
4.3. Resumen de síntesis	25
5. RAM	27
5.1. Descripción de la RAM	27
5.2. Simulación	28
5.3. Resumen de síntesis	30
6. LC-3	32
6.1. Descripción del procesador LC-3	32
6.2. Simulación	62
6.3. Resumen de síntesis	66
7. ANEXO	68
7.1. Componentes implementados en el procesador LC-3	68
Bibliografía	71

1. Especificaciones

1. Descripción de cada subsistema diseñado (incluir diagramas a bloques, explicar claramente cómo se espera que sea el funcionamiento de cada subsistema, etcétera).
2. Descripción de la estrategia desarrollada para simular el funcionamiento de cada subsistema (explicar claramente qué señales de prueba se introducirán a los puertos de entrada, por qué se han elegido esas señales, qué datos se introducirán en la memoria, qué comportamiento se espera observar, etcétera).
3. Descripción de la estrategia desarrollada para probar físicamente (en el kit de desarrollo) el funcionamiento de cada subsistema (explicar si hay algún sistema adicional añadido para hacer más fácil la interacción con el usuario, explicar claramente qué pines del kit se mapearán a qué puertos de entrada/salida del sistema, qué comportamiento se espera observar, etcétera. Por ejemplo, si el procesador final se va a probar con un teclado y un monitor, explicar el hardware adicional necesario para hacer interfaz entre el teclado y el procesador y entre el monitor y el procesador).
4. Pasos a seguir para diseñar el sistema completo.
5. Texto completo de todos los códigos VHDL utilizados, los archivos de asignación de pines, los archivos de inicialización de memoria (cuando aplique).
6. Imágenes generadas por el sintetizador y el simulador (máquinas de estado, diagramas RTL, formas de onda resultantes de la simulación funcional), añadiendo explicaciones correspondientes (los experimentos físicos se pueden documentar con fotografías).
7. Resultados del resumen de síntesis (cuántos recursos de hardware se utilizaron, cuál es la máxima frecuencia de operación del sistema).

2. ALU

2.1. Descripción de la Unidad Aritmética-Lógica (ALU)

Especificaciones: Se utilizan 2 señales de entrada para datos A y B de 16 bits. Una señal $ALUK$ de dos bits como entrada para el control. Una señal de salida de datos Y de 16 bits.

- A : señal de entrada de datos. 16 bits.
- B : señal de entrada de datos. 16 bits.
- $ALUK$: señal de entrada de control. 2 bits.
- Y : señal de salida de datos. 16 bits.

De acuerdo con el valor de la señal de control $ALUK$, se asigna una operación que será mapeada a la salida Y . Las operaciones se asignan de acuerdo a la tabla 1. La figura 1 muestra

el diagrama simplificado de la ALU implementada en el procesador LC-3.

Tabla 1: Tabla de verdad de la ALU del procesador LC-3.

ALUK(1)	ALUK(0)	Y
0	0	A + B
0	1	NOT A
1	0	A
1	1	A AND B

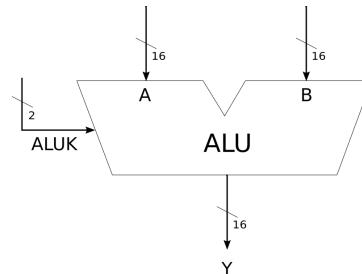


Figura 1: ALU del procesador LC-3.

El código de la implementación en VHDL se muestra a continuación.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY alu IS
  PORT(
    ALUK : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    A, B : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Y : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END alu;

ARCHITECTURE behavioral OF alu IS
  SIGNAL sum, ayb, not_a : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN
  sum <= STD_LOGIC_VECTOR(SIGNED(A) + SIGNED(B));
  ayb <= A AND B;
  not_a <= NOT A;

  Y <= sum WHEN ALUK = "00" ELSE
            not_a WHEN ALUK = "01" ELSE
            A WHEN ALUK = "10" ELSE
            ayb;

END behavioral;
  
```

2.2. Simulación

El tiempo de simulación se propuso de 100 ps en bajo y 100 ps en alto, para un ciclo completo de reloj. El tiempo total de simulación fue de 1.7 ns.

Para la simulación se propusieron 4 valores distintos para cada una de las entradas de datos A y B , por cada operación conmutada por $ALUK$. Dando un total de 16 operaciones. Las tabla 2, 3, 4 y 5 agrupan los valores de prueba utilizados en la simulación de ALU.

Tabla 2: Operaciones de suma ($ALUK = "00"$) realizadas en la simulación de la ALU implementada en el procesador LC-3.

A (hex)	B (hex)	$Y = A + B$ (hex)	$A + B = Y$ (comp2)
0003	000a	000d	$3 + 10 = 13$
0044	008f	00d3	$68 + 143 = 211$
3039	a06b	d0a4	$12345 + (-24469) = -12124$
a2d2	205b	c32d	$(-23854) + 8283 = -15571$

Tabla 3: Operaciones NOT A ($ALUK = "01"$) realizadas en la simulación de la ALU implementada en el procesador LC-3.

A (bin)	$Y = \text{NOT } A$ (bin)
0000000011111111	1111111100000000
0101010101010101	1010101010101010
0011001100110011	1100110011001100
0011101010101010	1100010101010101

Tabla 4: Operaciones $Y = A$ ($ALUK = "10"$) realizadas en la simulación de la ALU implementada en el procesador LC-3.

A (bin)	$Y = \text{NOT } A$ (bin)
0000000011111111	0000000011111111
0101010101010101	0101010101010101
0011001100110011	0011001100110011
0011101010101010	0011101010101010

Tabla 5: Operaciones $Y = A \text{ AND } B$ ($ALUK = "11"$) realizadas en la simulación de la ALU implementada en el procesador LC-3.

A (bin)	B (bin)	$Y = A \text{ AND } B$ (bin)
1111111111111111	0000000000000000	0000000000000000
000000001000100	0000000100001111	0000000000000100
001100000111001	1010000001101011	001000000101001
1010001011010010	0010000001011011	0010000001010010

El código para simular la ALU implementada en el procesador LC-3 se muestra a continuación.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY alu_testbench IS
END alu_testbench;

ARCHITECTURE behavioral OF alu_testbench IS

COMPONENT alu
PORT(
    ALUK : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    A, B : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Y : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END COMPONENT;

SIGNAL ALUK : std_logic_vector(1 DOWNTO 0);
SIGNAL A, B : std_logic_vector(15 DOWNTO 0);
SIGNAL Y : std_logic_vector(15 DOWNTO 0);

BEGIN
    -- instantiate the circuit under test
    uut : alu
    PORT MAP(ALUK => ALUK, A => A, B => B, Y => Y);

    -- test vector generator
    PROCESS
    BEGIN

        --  $Y = A+B$  (complemento a 2)
        WAIT FOR 100 ps;
        ALUK <= "00";
        A <= x"0003"; -- 3 en decimal
        B <= x"000a"; -- 10 en decimal

        WAIT FOR 100 ps;
        A <= x"0044"; -- 68 en decimal
        B <= x"008f"; -- 143 en decimal

        WAIT FOR 100 ps;
        A <= x"3039"; -- 12345 en decimal
        B <= x"a06b"; -- (-24469) en decimal

        WAIT FOR 100 ps;

    END PROCESS;

```

```

A <= x"a2d2"; -- (-23854) en decimal
B <= x"205b"; -- 8283 en decimal

-- y = not A
WAIT FOR 100 ps;
ALUK <= "01";
A <= "0000000011111111";

WAIT FOR 100 ps;
A <= "0101010101010101";

WAIT FOR 100 ps;
A <= "0011001100110011";

WAIT FOR 100 ps;
A <= "0011101010101010";

-- Y = A
WAIT FOR 100 ps;
ALUK <= "10";
A <= "0000000011111111";

WAIT FOR 100 ps;
A <= "0101010101010101";

WAIT FOR 100 ps;
A <= "0011001100110011";

WAIT FOR 100 ps;
A <= "0011101010101010";

-- Y = A and B
WAIT FOR 100 ps;
ALUK <= "11";
A <= "1111111111111111"; -- (-1) en decimal
B <= "0000000000000000";

WAIT FOR 100 ps;
A <= "000000001000100"; -- 68 en decimal
B <= "0000000100001111"; -- 143 en decimal

WAIT FOR 100 ps;
A <= "0011000000111001"; -- 12345 en decimal
B <= "1010000001101011"; -- (-24469) en decimal

WAIT FOR 100 ps;

```

```

        A <= "1010001011010010"; -- (-23854) en decimal
        B <= "0010000001011011"; -- 8283 en decimal

        -- Tiempo total de simulacion = 1.7 ns

    END PROCESS;

END behavioral;

```

La simulación de la operación SUMA se muestra en la figura 2. Para la operación NOT ver la figura 3. La operación Y=A se ve en la figura 4. Finalmente la operación AND se muestra en la figura 5.

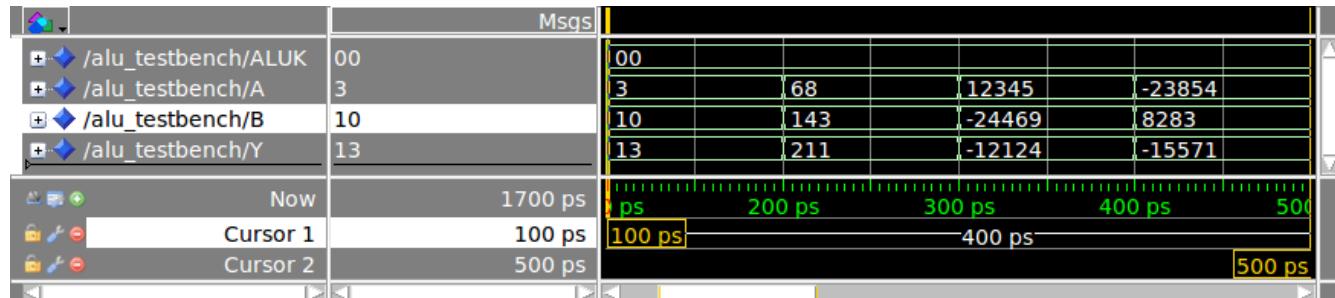


Figura 2: Simulación de la ALU. Operación Y = A + B.

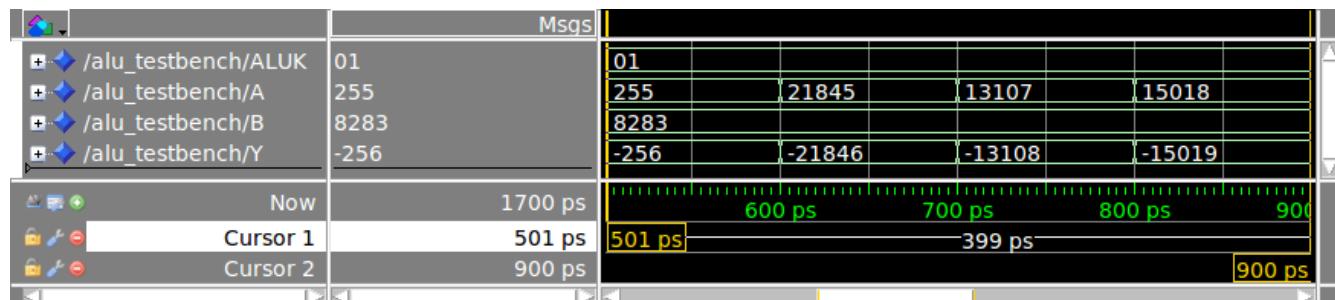


Figura 3: Simulación de la ALU. Operación Y = NOT A.

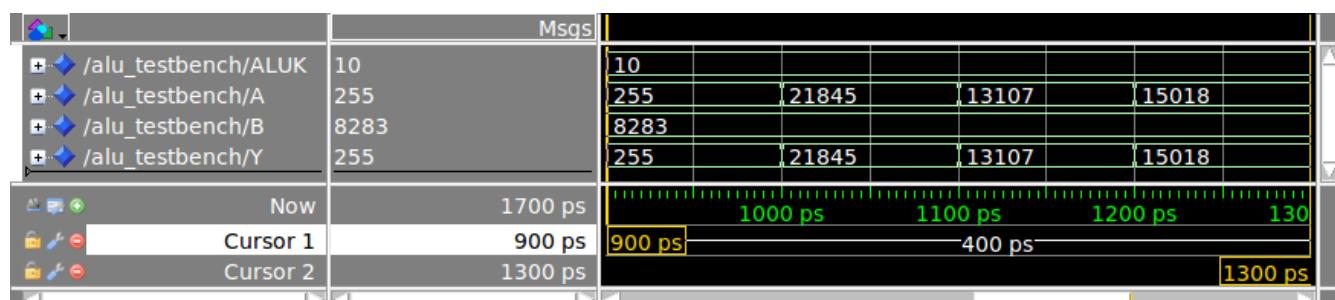


Figura 4: Simulación de la ALU. Operación Y = A.

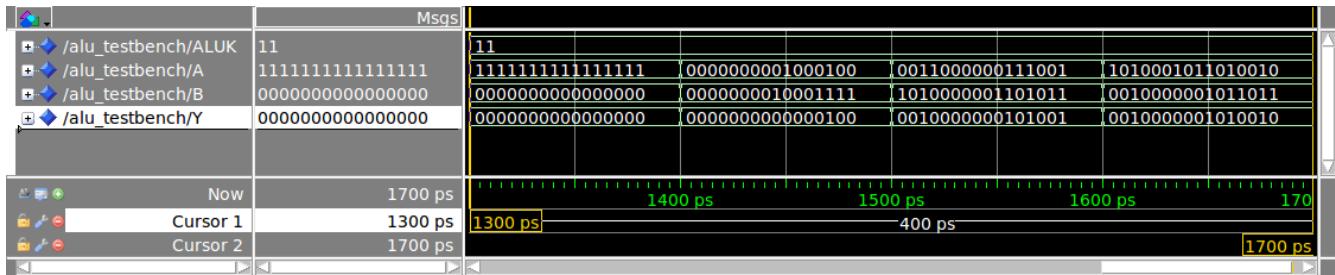


Figura 5: Simulación de la ALU. Operación $Y = A \text{ AND } B$.

2.3. Resumen de síntesis

En la figura 6 se muestra el diagrama de los componentes utilizados en la implementación de la ALU y en la figura 7 se muestra el reporte de componentes generado en Quartus Prime.

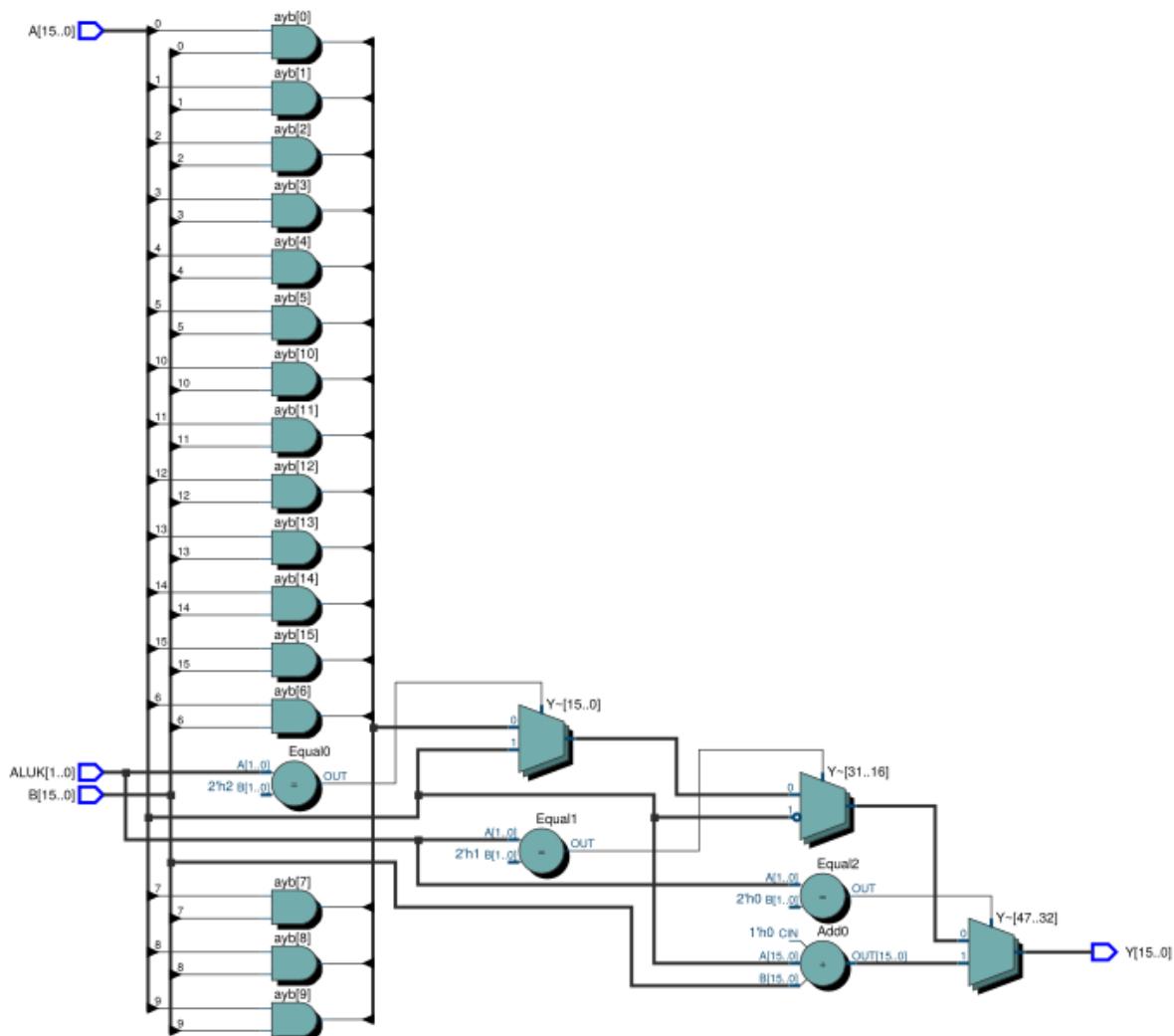


Figura 6: Esquema RTL de la ALU generado por Quartus Prime.

Partition Statistics report for alu

Sun Jun 10 16:59:13 2018

Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition

; Fitter Partition Statistics		
; Statistic	; Top	; hard_block:auto_generated_inst
; Difficulty Clustering Region	; Low	; Low
Total logic elements	; 48 / 114480 (< 1 %)	; 0 / 114480 (0 %)
-- Combinational with no register	; 48	; 0
-- Register only	; 0	; 0
-- Combinational with a register	; 0	; 0
Logic element usage by number of LUT inputs		
-- 4 input functions	; 16	; 0
-- 3 input functions	; 31	; 0
-- <=2 input functions	; 1	; 0
-- Register only	; 0	; 0
Logic elements by mode		
-- normal mode	; 33	; 0
-- arithmetic mode	; 15	; 0
Total registers	; 0	; 0
-- Dedicated logic registers	; 0 / 114480 (0 %)	; 0 / 114480 (0 %)
-- I/O registers	; 0	; 0
Total LABs: partially or completely used	; 3 / 7155 (< 1 %)	; 0 / 7155 (0 %)
Virtual pins	; 0	; 0
I/O pins	; 50	; 0
Embedded Multiplier 9-bit elements	; 0 / 532 (0 %)	; 0 / 532 (0 %)
Total memory bits	; 0	; 0
Total RAM block bits	; 0	; 0
Connections		
-- Input Connections	; 0	; 0
-- Registered Input Connections	; 0	; 0
-- Output Connections	; 0	; 0
-- Registered Output Connections	; 0	; 0
Internal Connections		
-- Total Connections	; 225	; 5
-- Registered Connections	; 0	; 0
External Connections		
-- Top	; 0	; 0
-- hard_block:auto_generated_inst	; 0	; 0
Partition Interface		
-- Input Ports	; 34	; 0
-- Output Ports	; 16	; 0
-- Bidir Ports	; 0	; 0
Registered Ports		
-- Registered Input Ports	; 0	; 0
-- Registered Output Ports	; 0	; 0
Port Connectivity		
-- Input Ports driven by GND	; 0	; 0
-- Output Ports driven by GND	; 0	; 0
-- Input Ports driven by VCC	; 0	; 0
-- Output Ports driven by VCC	; 0	; 0
-- Input Ports with no Source	; 0	; 0
-- Output Ports with no Source	; 0	; 0
-- Input Ports with no Fanout	; 0	; 0
-- Output Ports with no Fanout	; 0	; 0

Figura 7: Reporte generado en Quartus Prime de los componentes utilizados en la implementación de la ALU.

3. Banco de registros

3.1. Descripción del banco de 8 registros de 16 bits

Especificaciones: Se implemento un banco con 8 registros de 16 bits cada uno. La carga de datos a un registro no es simultánea, solo se puede agregar un valor en un ciclo de reloj, pero se pueden tener dos valores de salida. El control de carga a los registros se hace por medio de un decoder con dos señales de entrada y una señal de salida. Las salidas, que pueden ser dos simultáneas, se controlan con 2 multiplexores. Cada registro de 16 bits cuenta con sus señales de reloj, reset, enable, valor de entrada y valor de salida. Todas estas señales se listan a continuación.

Para cada registro de 16 bits.

- clk : señal de entrada de reloj. 1 bit.
- reset : señal de entrada para inicializar los registros. 1 bit.
- dec_out(x) : señal interna para habilitar la carga en los registros. 1 bit.
- data : señal portadora del valor de entrada. 16 bits.
- rx_reg : señal interna que mantiene el valor de salida. 16 bits.

Para el control de carga de registros

- ld_reg : señal de entrada para habilitar el uso del banco de registros, tanto para escritura como para lectura. 1 bit.
- dr : señal que controla a que registro se le cargara el dato de entrada. 3 bits.

Para los multiplexores que controlan la salida

- rx_reg : señal interna que mantiene el valor de salida en un registro dado y es valor de entrada a cada uno de los multiplexores. Todas las señales de salida de los registros son señales de entrada de los dos multiplexores. 16 bits.
- sr1, sr2 : señal que controla de que registro se obtendrá el valor de salida. Se tienen dos controladores ya que se pueden obtener dos salidas simultáneas. 3 bits.
- sr1out, sr2out : señales por donde se obtienen los valores de salida. 16 bits.

La forma de habilitar la escritura en un registro se hace de acuerdo con la tabla 6.

Tabla 6: Habilitación de escritura en los registros utilizando la señal dec_out[7:0].

dec_out[7:0]	habilitar el registro
10000000	7
01000000	6
00100000	5
00010000	4
00001000	3
00000100	2
00000010	1
00000001	0

En la figura 8 se representan el esquema general del banco de registros con sus entradas y salidas.

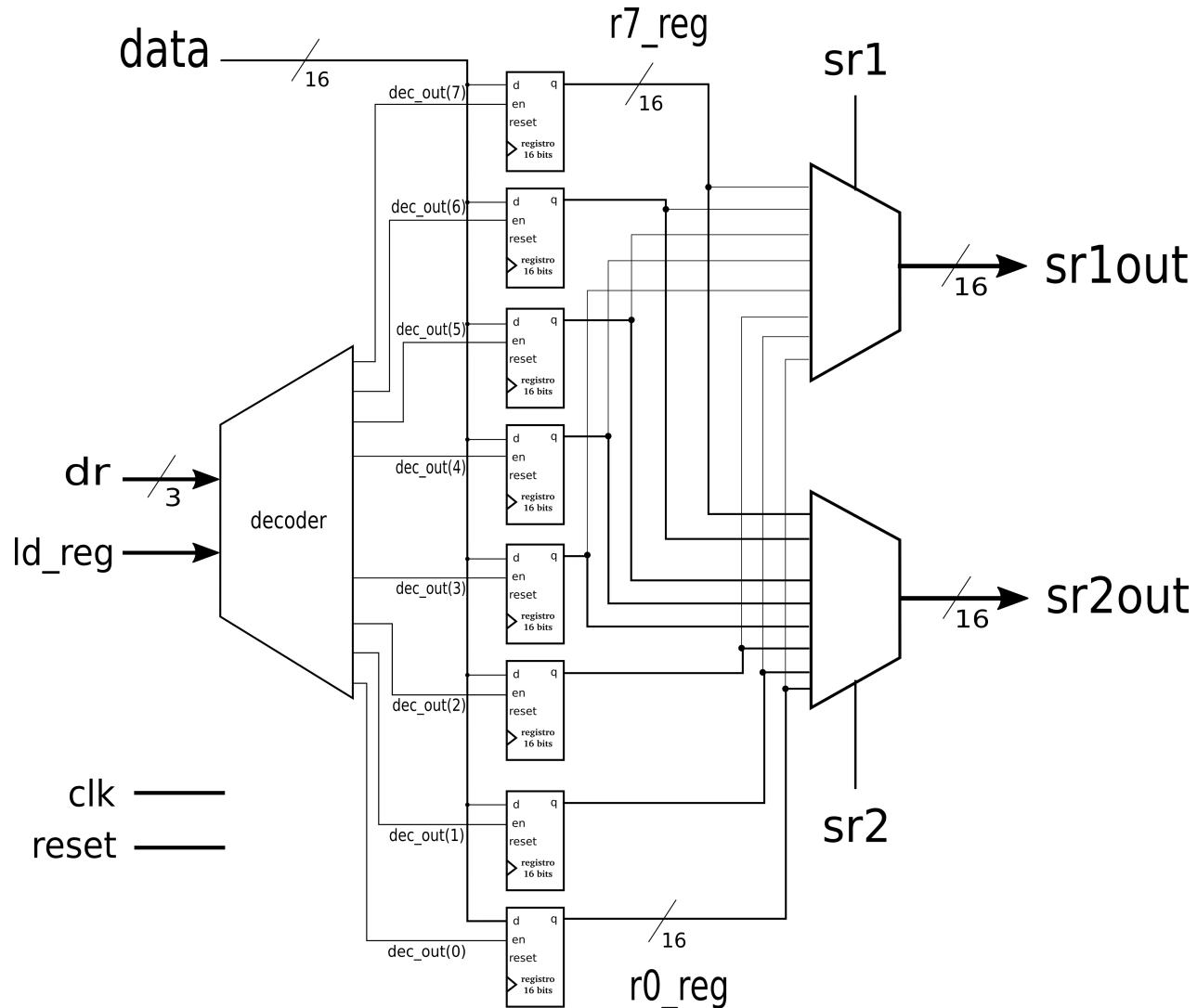


Figura 8: Diagrama simplificado del banco de registros implementado en el procesador LC-3.

El código de la implementación en VHDL se muestra a continuación.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY banco_8reg16bits IS
    PORT(
        clk, reset, ld_reg : IN STD_LOGIC;
        data : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        dr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        sr1, sr2 : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        sr1out, sr2out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END banco_8reg16bits;

ARCHITECTURE behavioral OF banco_8reg16bits IS

    SIGNAL dec_out : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL r0_next, r0_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r1_next, r1_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r2_next, r2_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r3_next, r3_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r4_next, r4_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r5_next, r5_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r6_next, r6_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL r7_next, r7_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN

    -- register
    PROCESS(clk, reset)
        BEGIN
            IF (reset='1') THEN
                r0_reg <= (OTHERS =>'0');
                r1_reg <= (OTHERS =>'0');
                r2_reg <= (OTHERS =>'0');
                r3_reg <= (OTHERS =>'0');
                r4_reg <= (OTHERS =>'0');
                r5_reg <= (OTHERS =>'0');
                r6_reg <= (OTHERS =>'0');
                r7_reg <= (OTHERS =>'0');
            ELSIF (clk'EVENT and clk='1') THEN
                r0_reg <= r0_next;
                r1_reg <= r1_next;
                r2_reg <= r2_next;
                r3_reg <= r3_next;
                r4_reg <= r4_next;
            END IF;
        END PROCESS;
    end;
```

```

        r5_reg <= r5_next;
        r6_reg <= r6_next;
        r7_reg <= r7_next;
    END IF;
END PROCESS;

-- next-state logic
r0_next <= data WHEN dec_out(0) = '1' ELSE r0_reg;
r1_next <= data WHEN dec_out(1) = '1' ELSE r1_reg;
r2_next <= data WHEN dec_out(2) = '1' ELSE r2_reg;
r3_next <= data WHEN dec_out(3) = '1' ELSE r3_reg;
r4_next <= data WHEN dec_out(4) = '1' ELSE r4_reg;
r5_next <= data WHEN dec_out(5) = '1' ELSE r5_reg;
r6_next <= data WHEN dec_out(6) = '1' ELSE r6_reg;
r7_next <= data WHEN dec_out(7) = '1' ELSE r7_reg;

-- decoder
WITH ld_reg & dr SELECT
    dec_out <= x"80" WHEN '1' & "111",
                                            x"40" WHEN '1' & "110",
                                            x"20" WHEN '1' & "101",
                                            x"10" WHEN '1' & "100",
                                            x"08" WHEN '1' & "011",
                                            x"04" WHEN '1' & "010",
                                            x"02" WHEN '1' & "001",
                                            x"01" WHEN '1' & "000",
                                            x"00" WHEN OTHERS;

-- MUX1
WITH sr1 SELECT
    sr1out <= r7_reg WHEN "111",
                                            r6_reg WHEN "110",
                                            r5_reg WHEN "101",
                                            r4_reg WHEN "100",
                                            r3_reg WHEN "011",
                                            r2_reg WHEN "010",
                                            r1_reg WHEN "001",
                                            r0_reg WHEN OTHERS;

-- MUX2
WITH sr2 SELECT
    sr2out <= r7_reg WHEN "111",
                                            r6_reg WHEN "110",
                                            r5_reg WHEN "101",
                                            r4_reg WHEN "100",
                                            r3_reg WHEN "011",
                                            r2_reg WHEN "010",
                                            r1_reg WHEN "001",

```

```

        r0_reg WHEN OTHERS;

END behavioral;

```

3.2. Simulación

El tiempo de simulación se propuso de 50 ps en bajo y 50 ps en alto, para un ciclo completo de reloj. El tiempo total de simulación fue de 1.1 ns.

En la simulación se probó tanto la escritura como la lectura en todos los registros. Los valores de prueba se muestran en la tabla 7. La manera de interpretar la tabla, tomando el primer renglón de datos como ejemplo es: 1) $en = 1$, se puede escribir en los registros 2) $dr = 111$ y $data = x"0001"$, escribir el valor 1 (en decimal) en el registro 7, 3) leer el valor del registro 7 y ponerlo en la salida $sr1$ y 4) leer el valor del registro 7 y ponerlo en la salida $sr2$. El resumen de las pruebas implementadas se muestran en la tabla 7.

Tabla 7: Datos para simulación de escritura y lectura en el banco de 8 registros de 16 bits.

ld_reg	dr (bin)	data (hex)	sr1 (bin)	sr2(bin)
1	111	0001	111	111
1	110	3039	111	111
1	101	a06b	111	111
1	100	205b	110	101
1	011	00ff	100	111
1	010	55555	100	101
1	001	a2d2	001	111
1	000	0003	001	011
0	011	3333	011	000
0	000	0003	011	000

El código para simular el banco de 8 registros de 16 bits implementado en el procesador LC-3 se muestra a continuación.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY banco_8reg16bits_testbench IS
END banco_8reg16bits_testbench;

ARCHITECTURE behavioral OF banco_8reg16bits_testbench IS

COMPONENT banco_8reg16bits IS
PORT(

```

```

clk, reset, ld_reg : IN STD_LOGIC;
data : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
dr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
sr1, sr2 : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
sr1out, sr2out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END COMPONENT;

SIGNAL clk, reset, wr_en : STD_LOGIC;
SIGNAL data : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL dr : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL sr1, sr2 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL sr1out, sr2out : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN
-- instantiate the circuit under test
uut : banco_8reg16bits PORT MAP(clk, reset, wr_en, data, dr,

-- test vector generator
PROCESS
BEGIN

reset <= '1';

-- storage a value in addr7
WAIT FOR 50 ps;
clk <= '0';
reset <= '0';
wr_en <= '1'; -- enable for write
dr <= "111"; -- enable addr7
data <= x"0001"; -- 1 en decimal
sr1 <= "111"; -- put value from addr7 on sr1out
sr2 <= "111"; -- put value from addr7 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- storage a value in addr6
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "110"; -- enable addr6
data <= x"3039"; -- 12345 en decimal
sr1 <= "111"; -- put value from addr7 on sr1out
sr2 <= "111"; -- put value from addr7 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

```

```

-- storage a value in addr5
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "101"; -- enable addr5
data <= x"a06b"; -- (-24469) en decimal
sr1 <= "111"; -- put value from addr1 on sr1out
sr2 <= "111"; -- put value from addr0 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- storage a value in addr4
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "100"; -- enable addr4
data <= x"205b"; -- enable 8283 en decimal
sr1 <= "110"; -- put value from addr6 on sr1out
sr2 <= "101"; -- put value from addr5 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- storage a value in addr3
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "011"; -- enable addr3
data <= x"00ff"; -- 255 en decimal
sr1 <= "100"; -- put value from addr4 on sr1out
sr2 <= "111"; -- put value from addr7 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1'; -- time = 500ps --

-- storage a value in addr2
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "010"; -- enable addr2
data <= x"5555"; -- 21845 en decimal
sr1 <= "100"; -- put value from addr4 on sr1out
sr2 <= "101"; -- put value from addr5 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- storage a value in addr1
WAIT FOR 50 ps;

```

```

clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "001"; -- enable addr1
data <= x"a2d2"; -- (-23854) en decimal
sr1 <= "001"; -- put value from addr1 on sr1out
sr2 <= "111"; -- put value from addr7 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- storage a value in addr0
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '1'; -- enable for write
dr <= "000"; -- enable addr0
data <= x"0003"; -- 3 en decimal
sr1 <= "001"; -- put value from addr1 on sr1out
sr2 <= "011"; -- put value from addr3 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- reset all
WAIT FOR 20 ps; reset <= '1';
-- storage a value in addr3
WAIT FOR 30 ps;
clk <= '0';
reset <= '0';
wr_en <= '0'; -- enable for write
dr <= "011"; -- enable addr3
data <= x"3333"; -- 13107 en decimal
sr1 <= "011"; -- put value from addr3 on sr1out
sr2 <= "000"; -- put value from addr0 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1'; -- time = 1 ns --

-- storage a value in addr0
WAIT FOR 50 ps;
clk <= '0';
wr_en <= '0'; -- unenable for write
dr <= "000"; -- enable addr0
data <= x"0003"; -- 3 en decimal
sr1 <= "011"; -- put value from addr3 on sr1out
sr2 <= "000"; -- put value from addr0 on sr2out
-- execute the transference
WAIT FOR 50 ps; clk <= '1';

-- Simulation total time = 1.1 ns

```

```

END PROCESS;

END behavioral;

```

La figura 9 muestra el resultado de la simulación realizada con modelsim del banco de 8 registros de 16 bits implementado en el procesador LC-3.

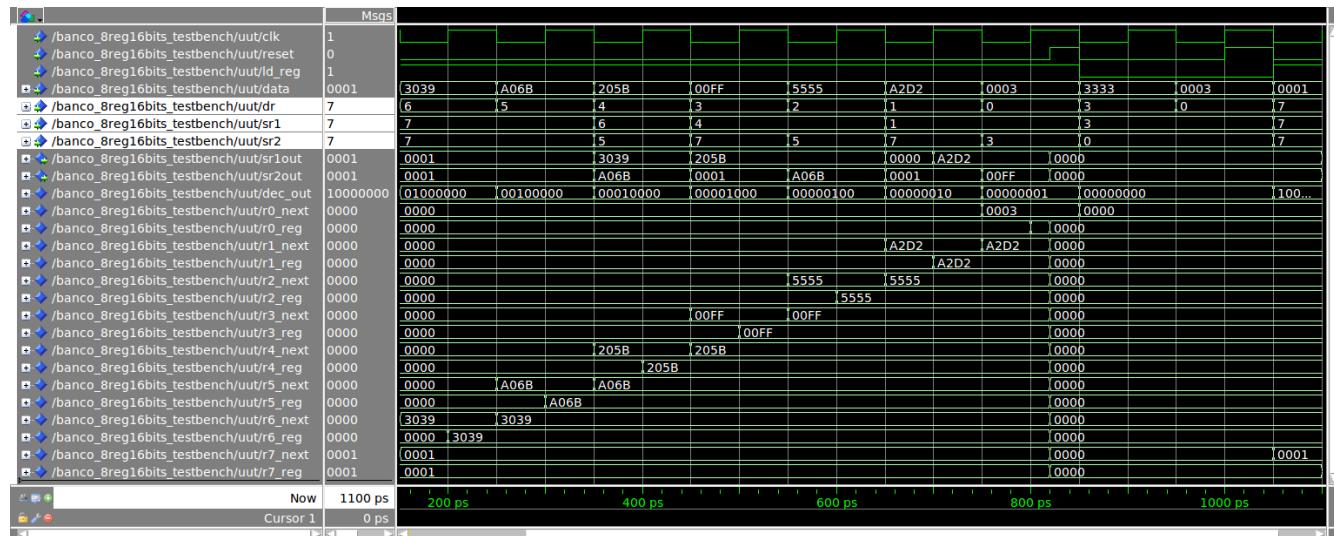


Figura 9: Simulación del banco de registros implementado en el procesador LC-3.

3.3. Resumen de síntesis

En la figura 10 se muestra el diagrama de los componentes utilizados en la implementación del banco de 8 registros de 16 bits y en la figura 11 se muestra el reporte de componentes generado en Quartus Prime.

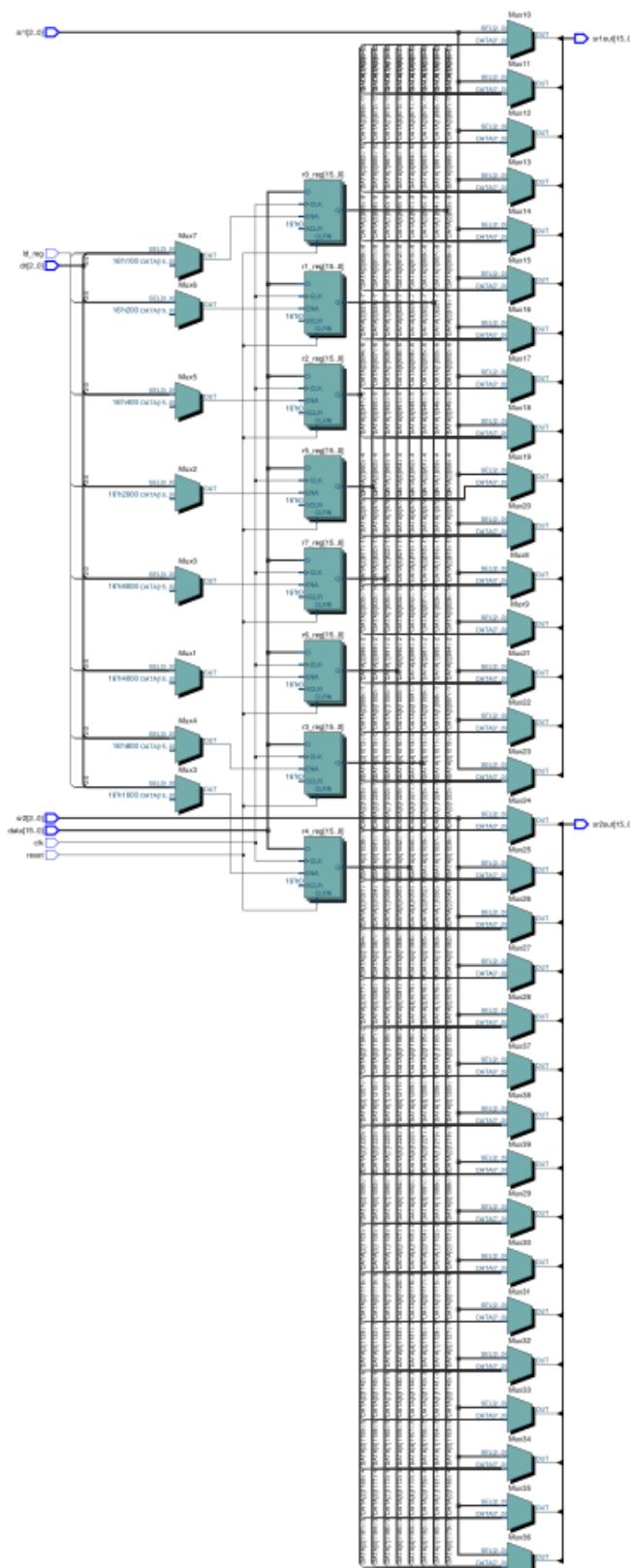


Figura 10: Esquema RTL del banco de 8 registros de 16 bits generado por Quartus Prime.

```

Partition Statistics report for banco_8reg16bits
Mon Jun 11 00:50:45 2018
Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition
+-----+
; Fitter Partition Statistics
+-----+
; Statistic ; Top ; hard_block:auto_generated_inst ;
+-----+
; Difficulty Clustering Region ; Low ; Low
+-----+
; Total logic elements ; 168 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;-- Combinational with no register ; 40 ; 0
;-- Register only ; 0 ; 0
;-- Combinational with a register ; 128 ; 0
+-----+
; Logic element usage by number of LUT inputs ;
;-- 4 input functions ; 136 ; 0
;-- 3 input functions ; 32 ; 0
;-- <=2 input functions ; 0 ; 0
;-- Register only ; 0 ; 0
+-----+
; Logic elements by mode ;
;-- normal mode ; 168 ; 0
;-- arithmetic mode ; 0 ; 0
+-----+
; Total registers ; 128 ; 0
;-- Dedicated logic registers ; 128 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;-- I/O registers ; 0 ; 0
+-----+
; Total LABs: partially or completely used ; 12 / 7155 ( < 1 % ) ; 0 / 7155 ( 0 % )
+-----+
; Virtual pins ; 0 ; 0
; I/O pins ; 60 ; 0
; Embedded Multiplier 9-bit elements ; 0 / 532 ( 0 % ) ; 0 / 532 ( 0 % )
; Total memory bits ; 0 ; 0
; Total RAM block bits ; 0 ; 0
; Clock control block ; 2 / 24 ( 8 % ) ; 0 / 24 ( 0 % )
+-----+
; Connections ;
;-- Input Connections ; 0 ; 0
;-- Registered Input Connections ; 0 ; 0
;-- Output Connections ; 0 ; 0
;-- Registered Output Connections ; 0 ; 0
+-----+
; Internal Connections ;
;-- Total Connections ; 1246 ; 5
;-- Registered Connections ; 256 ; 0
+-----+
; External Connections ;
;-- Top ; 0 ; 0
;-- hard_block:auto_generated_inst ; 0 ; 0
+-----+
; Partition Interface ;
;-- Input Ports ; 28 ; 0
;-- Output Ports ; 32 ; 0
;-- Bidir Ports ; 0 ; 0
+-----+
; Registered Ports ;
;-- Registered Input Ports ; 0 ; 0
;-- Registered Output Ports ; 0 ; 0
+-----+
; Port Connectivity ;
;-- Input Ports driven by GND ; 0 ; 0
;-- Output Ports driven by GND ; 0 ; 0
;-- Input Ports driven by VCC ; 0 ; 0
;-- Output Ports driven by VCC ; 0 ; 0
;-- Input Ports with no Source ; 0 ; 0
;-- Output Ports with no Source ; 0 ; 0
;-- Input Ports with no Fanout ; 0 ; 0
;-- Output Ports with no Fanout ; 0 ; 0
+-----+

```

Figura 11: Reporte generado en Quartus Prime de los componentes utilizados en la implementación del banco de 8 registros de 16 bits.

4. Condition codes

4.1. Descripción del condition codes

Especificaciones: El condition codes, es una conjunto de elementos que permiten saber si un valor de entrada es negativo (N), cero (Z) o positivo (P), asignando un valor en alto (uno lógico) cuando la comparación es verdadera y valor en bajo (cero lógico), cuando es falsa. Los valores de comparación se almacenan en registros de 1 bit.

Posteriormente se comparan tres valores de entrada proporcionados por el registro IR[11:9] contra los valores obtenidos en la comparación previa. Si los valores son iguales se asigna uno lógico, sino se asigna cero lógico. El valor resultante de ambas comparaciones se denomina BEN.

Por ejemplo si el resultado en la primera comparación da como resultado $NZP = "100"$ y el valor de $IR[11 : 9] = "100"$ el valor asignado a BEN, será uno lógico, y de igual manera para todas las combinaciones posibles.

Las señales necesarias para la implementación del condition codes se describen a continuación:

- clk : señal de entrada de reloj. 1 bit.
- reset : señal de entrada para inicializar los registros. 1 bit.
- ld_cc : señal de entrada para habilitar la carga en los registros N, Z y P. 1 bit.
- data : señal portadora del valor de entrada a comparar. 16 bits.
- n, z, p, ben : señales de salida para las etapas de comparación. Todas de 1 bit.

En la figura 12 se representan el esquema general del condition codes.

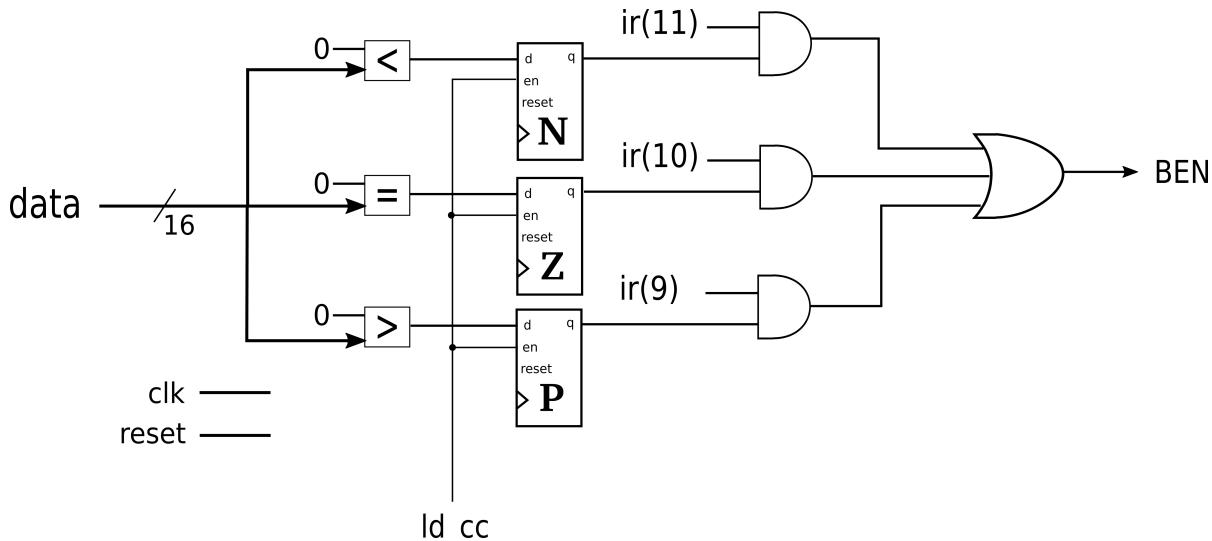


Figura 12: Diagrama simplificado del condition codes implementado en el procesador LC-3.

El código de la implementación en VHDL se muestra a continuación.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cond_cod IS
    PORT(
        clk, reset, ld_cc : IN std_logic;
        data : IN std_logic_vector(15 DOWNTO 0);
        ir : IN std_logic_vector(11 DOWNTO 9);
        n, z, p, ben : OUT std_logic
    );
END cond_cod;

ARCHITECTURE behavioral OF cond_cod IS

    SIGNAL n_next, n_reg, n_value : std_logic;
    SIGNAL z_next, z_reg, p_value : std_logic;
    SIGNAL p_next, p_reg, z_value : std_logic;

BEGIN

    PROCESS(clk, reset)
    BEGIN
        IF reset='1' THEN
            n_reg <= '0';
            z_reg <= '0';
            p_reg <= '0';
        ELSIF(clk'EVENT AND clk='1') THEN
            n_reg <= n_next;
            z_reg <= z_next;
            p_reg <= p_next;
        END IF;
    END PROCESS;

    PROCESS(data)
    BEGIN
        IF(data(15) = '1') THEN
            n_value <= '1';
            z_value <= '0';
            p_value <= '0';
        ELSIF(data=x"0000") THEN
            n_value <= '0';
            z_value <= '1';
            p_value <= '0';
        ELSE
            n_value <= '0';
        END IF;
    END PROCESS;

```

```

        z_value <= '0';
        p_value <= '1';
    END IF;
END PROCESS;

-- NEXT-STATE LOGIC
n_next <= n_value when ld_cc='1' else n_reg;
z_next <= z_value when ld_cc='1' else z_reg;
p_next <= p_value when ld_cc='1' else p_reg;

-- OUTPUT LOGIC
n <= n_reg;
z <= z_reg;
p <= p_reg;

-- ben
ben <= (ir(11) and n_reg) or (ir(10) and z_reg) or (ir(9) and p_reg);

END behavioral;

```

4.2. Simulación

El tiempo de simulación se propuso de 50 ps en bajo y 50 ps en alto, para un ciclo completo de reloj. El tiempo total de simulación fue de 900 ps.

En la simulación se probó las comparaciones resultantes con tres valores de entrada, uno negativo, otro igual a cero y uno positivo. Para cada valor se hicieron tres comparaciones con respecto al IR[11:9]. La tabla 8 resume los valores propuestos para la simulación.

Tabla 8: Datos para la simulación del condition codes.

data	IR[11:9]	IR[11:9]	IR[11:9]
0000 (hex)	100	010	001
9a6b (hex), -26005 (comp2)	100	010	001
35a5 (hex), 13733 (comp2)	100	010	001

El código para simular el condition codes implementado en el procesador LC-3 se muestra a continuación.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cond_cod_testbench IS
END cond_cod_testbench;

```

```

ARCHITECTURE behavioral OF cond_cod_testbench IS

COMPONENT cond_cod IS
PORT(
    clk, reset, ld_cc : IN std_logic;
    data : IN std_logic_vector(15 DOWNTO 0);
    ir : IN std_logic_vector(11 DOWNTO 9);
    n, z, p, ben : OUT std_logic
);
END COMPONENT;

SIGNAL clk, reset, ld_cc : STD_LOGIC;
SIGNAL data : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ir : std_logic_vector(11 DOWNTO 9);
SIGNAL n, z, p : STD_LOGIC;

BEGIN

uut : cond_cod PORT MAP(clk, reset, ld_cc, data, ir, n, z, p);

PROCESS
    BEGIN

-- ir[11:9] = NZP

-- initial conditions
reset <= '1';
ld_cc <= '1';

----- data EQUAL TO ZERO -----
-- ir = 100
WAIT FOR 50 ps; clk <= '0';
reset <= '0';
data <= x"0000";
ir <= "100";
WAIT FOR 50 ps; clk <= '1';

-- ir = 010
WAIT FOR 50 ps; clk <= '0';
ir <= "010";
WAIT FOR 50 ps; clk <= '1';

-- ir = 001 => BEN = 0
WAIT FOR 50 ps; clk <= '0';
ir <= "001";
WAIT FOR 50 ps; clk <= '1';

```

```

----- NEGATIVE data-----
-- ir = 100
WAIT FOR 50 ps; clk <= '0';
data <= x"9a6b"; -- (-26005) ld_reg decimal
ir <= "100";
WAIT FOR 50 ps; clk <= '1';

-- ir = 010
WAIT FOR 50 ps; clk <= '0';
ir <= "010";
WAIT FOR 50 ps; clk <= '1'; -- 500 ps

-- ir = 001
WAIT FOR 50 ps; clk <= '0';
ir <= "001";
WAIT FOR 50 ps; clk <= '1';

----- POSITIVE data-----
-- ir = 100
WAIT FOR 50 ps; clk <= '0';
data <= x"35a5"; -- 13733 ld_reg decimal
ir <= "100";
WAIT FOR 50 ps; clk <= '1';

-- ir = 010
WAIT FOR 50 ps; clk <= '0';
ir <= "010";
WAIT FOR 50 ps; clk <= '1';

-- ir = 001
WAIT FOR 50 ps; clk <= '0';
ir <= "001";
WAIT FOR 50 ps; clk <= '1';

-- simulation total time 900 ps

END PROCESS;
END behavioral;

```

La figura 13 muestra el resultado de la simulación realizada con modelsim del condition codes.

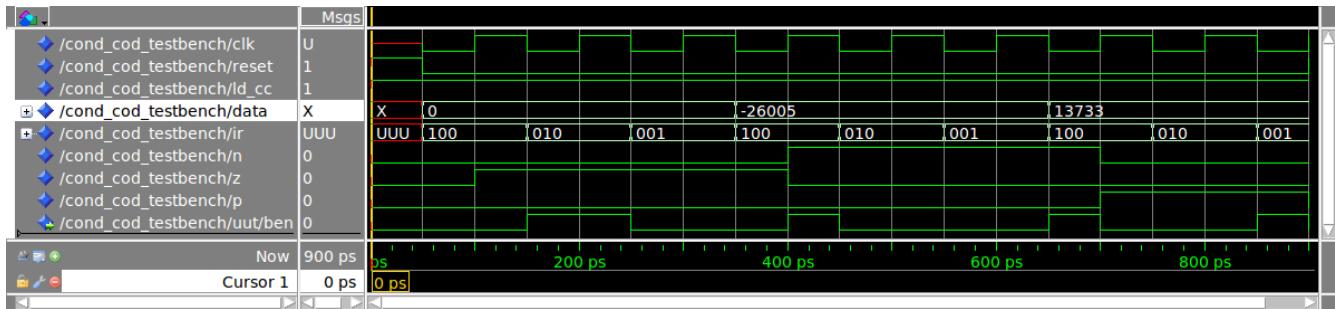


Figura 13: Simulación del condition codes implementado en el procesador LC-3.

4.3. Resumen de síntesis

En la figura 14 se muestra el diagrama de los componentes utilizados en la implementación del condition code y en la figura 15 se muestra el reporte de componentes generado en Quartus Prime.

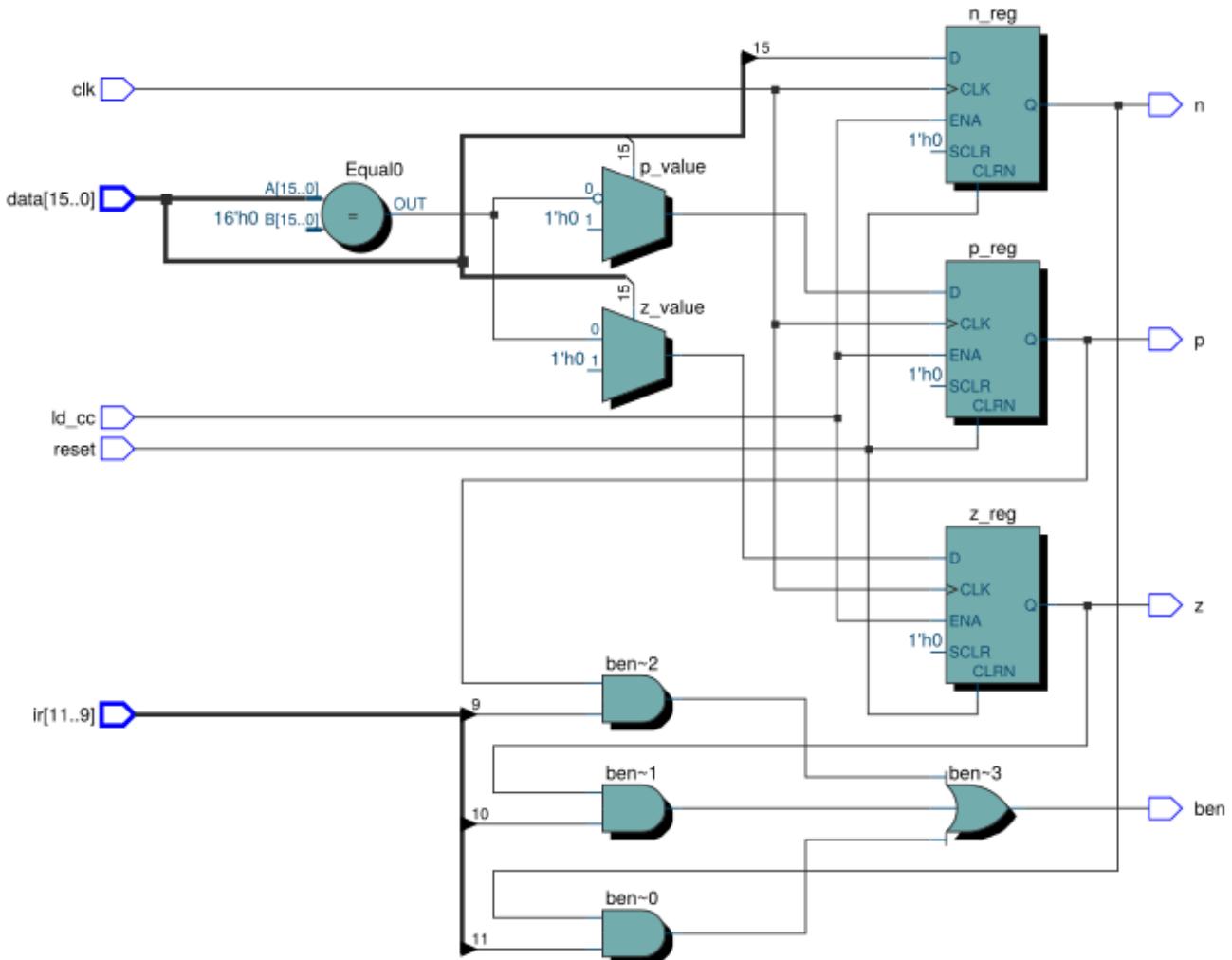


Figura 14: Esquema RTL del banco de 8 registros de 16 bits generado por Quartus Prime.

```

Partition Statistics report for cond_cod
Mon Jun 11 03:10:49 2018
Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition
+-----+
; Fitter Partition Statistics
+-----+
; Statistic ; Top ; hard_block:auto_generated_inst ;
+-----+
; Difficulty Clustering Region ; Low ; Low
+-----+
; Total logic elements ; 8 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;   -- Combinational with no register ; 5 ; 0
;   -- Register only ; 0 ; 0
;   -- Combinational with a register ; 3 ; 0
+-----+
; Logic element usage by number of LUT inputs
;   -- 4 input functions ; 6 ; 0
;   -- 3 input functions ; 1 ; 0
;   -- <=2 input functions ; 1 ; 0
;   -- Register only ; 0 ; 0
+-----+
; Logic elements by mode
;   -- normal mode ; 8 ; 0
;   -- arithmetic mode ; 0 ; 0
+-----+
; Total registers ; 3 ; 0
;   -- Dedicated logic registers ; 3 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;   -- I/O registers ; 0 ; 0
+-----+
; Total LABs: partially or completely used ; 5 / 7155 ( < 1 % ) ; 0 / 7155 ( 0 % )
+-----+
; Virtual pins ; 0 ; 0
; I/O pins ; 26 ; 0
; Embedded Multiplier 9-bit elements ; 0 / 532 ( 0 % ) ; 0 / 532 ( 0 % )
; Total memory bits ; 0 ; 0
; Total RAM block bits ; 0 ; 0
; Clock control block ; 2 / 24 ( 8 % ) ; 0 / 24 ( 0 % )
+-----+
; Connections
;   -- Input Connections ; 0 ; 0
;   -- Registered Input Connections ; 0 ; 0
;   -- Output Connections ; 0 ; 0
;   -- Registered Output Connections ; 0 ; 0
+-----+
; Internal Connections
;   -- Total Connections ; 73 ; 5
;   -- Registered Connections ; 6 ; 0
+-----+
; External Connections
;   -- Top ; 0 ; 0
;   -- hard_block:auto_generated_inst ; 0 ; 0
+-----+
; Partition Interface
;   -- Input Ports ; 22 ; 0
;   -- Output Ports ; 4 ; 0
;   -- Bidir Ports ; 0 ; 0
+-----+
; Registered Ports
;   -- Registered Input Ports ; 0 ; 0
;   -- Registered Output Ports ; 0 ; 0
+-----+
; Port Connectivity
;   -- Input Ports driven by GND ; 0 ; 0
;   -- Output Ports driven by GND ; 0 ; 0
;   -- Input Ports driven by VCC ; 0 ; 0
;   -- Output Ports driven by VCC ; 0 ; 0
;   -- Input Ports with no Source ; 0 ; 0
;   -- Output Ports with no Source ; 0 ; 0
;   -- Input Ports with no Fanout ; 0 ; 0
;   -- Output Ports with no Fanout ; 0 ; 0
+-----+

```

Figura 15: Reporte generado en Quartus Prime de los componentes utilizados en la implementación del banco de 8 registros de 16 bits.

5. RAM

5.1. Descripción de la RAM

Especificaciones: En el procesador LC-3 se utiliza una RAM (Random Access Memory) para almacenar instrucciones del sistema operativo, programas pequeños y como área de intercambio de información, almacenamiento temporal. Utiliza una señal como apuntador de la dirección que será utilizada para lectura o escritura de datos. Una señal portadora del valor a escribir y una señal para la lectura de un dato. En esencia funciona de forma similar al banco de registros implementado en este proyecto y cuenta con sus señales de reloj y enable. El tamaño de la RAM implementada es de 65536 registros de 16 bits cada uno.

Las señales necesarias para la implementación de la RAM se describen a continuación:

- clk : señal de entrada de reloj. 1 bit.
- r_w : señal de entrada para habilitar la lectura o escritura de un dato. 1 bit.
- mem_en : señal de entrada para habilitar el uso de la RAM, tanto para escritura como para lectura. 1 bit.
- ram_in : señal portadora del valor de entrada a escribir en la RAM. 16 bits.
- addr_ram : señal de entrada que funciona como apuntador. Esta señal indica el número de registro que se ha de utilizar, ya sea para escritura o lectura de un dato. Entero con un rango entre 0 y 65535.
- ram_out : señal de salida que contendrá el dato leído en la RAM. 16 bits.

En la figura 16 se representan el esquema general de la RAM.

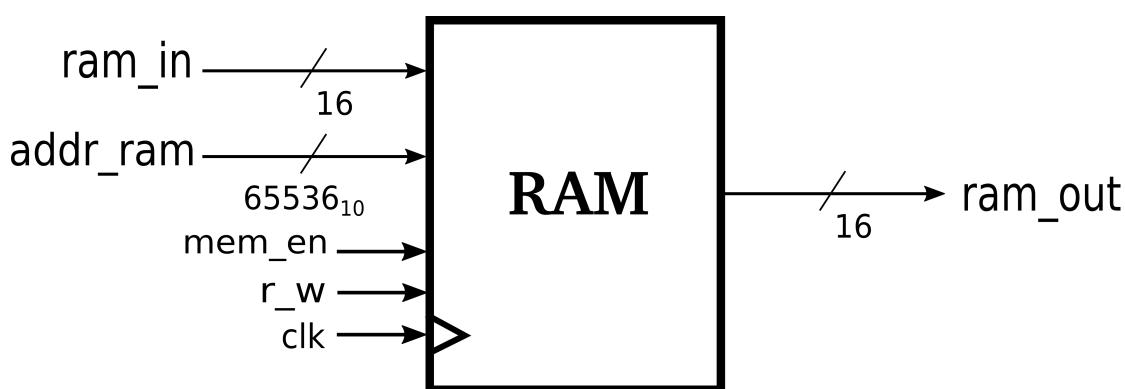


Figura 16: Diagrama simplificado de la RAM implementada en el procesador LC-3.

El código de la implementación en VHDL se muestra a continuación.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```

ENTITY ram IS
  PORT (
    clk : IN STD_LOGIC;
    r_w, mem_en : IN STD_LOGIC;
    ram_in : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    addr_ram : IN INTEGER RANGE 0 TO 65535;
    ram_out : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
  );
END ENTITY ram;

ARCHITECTURE ram_arch OF ram IS

  TYPE memory IS ARRAY (65535 DOWNTO 0) OF STD_LOGIC_VECTOR (15 DOWNTO 0);
  SIGNAL content: memory;

  -- ATTRIBUTE ram_init_file : STRING;
  -- ATTRIBUTE ram_init_file OF content:
  --   SIGNAL IS "sum.mif";

BEGIN

  PROCESS(clk, mem_en)

    BEGIN
      IF (RISING_EDGE(clk) AND mem_en = '1') THEN
        CASE r_w IS
          WHEN '1' => content(addr_ram) <= ram_in;
          WHEN OTHERS => ram_out <= content(addr_ram);
        END CASE;
      END IF;
    END PROCESS;
  END ARCHITECTURE;

```

5.2. Simulación

La RAM no se simuló como objeto independiente. Se observó su funcionamiento en la simulación final del LC-3.

El archivo para probar el funcionamiento del procesador LC-3, es el ejemplo 6.4 de [1] que implementa el algoritmo mostrado en la figura 17.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	R1 <- R1 + 15
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	R2 <- M[M[x3009]]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	BRn x3008
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- R1 - 1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	R2 <- R2 + R2
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	BRn x3008
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	BRnzp x3004
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3009	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	x3100

Figura 17: Ejemplo 6.4 de [1]. Programa LC-3 para encontrar el primer 1 en una palabra.

El ejemplo utilizado realiza los siguientes pasos, comenzando siempre la lectura en el registro $x3000$:

1. Poner cero en el registro 1.
2. Sumar 15 a el valor del registro 1.
3. Obtener el valor de la memoria ubicado en la posición que dé el valor almacenado en la posición $x3009$ de la memoria. Poner el valor obtenido en el registro 2.
4. Si encuentras un uno, lee el valor almacenado en $x3008$ de la memoria. Sino, continua.
5. Resta uno al valor en el registro 1.
6. Duplica el valor del registro 2.
7. Si encuentras un uno, lee el valor almacenado en $x3008$ de la memoria. Sino, continua.
8. Si encontraste un uno, continua. Sino lee en valor almacenado en $x3004$ de la memoria (esto significa volver al paso 5 de esta lista).
9. Finaliza el programa.

El archivo `.mem` utilizado para la simulación en modelsim contiene la siguiente información. *Nota: la información está resumida ya que se cuentan con 65536 registros, pero no todos son ocupados.*

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/ram_testbench/uut/content
// format=mti addressradix=d dataradix=h version=1.0 wordsperline=1
65535: XXXX
65534: XXXX
65533: XXXX

.
.

12557: 0000
12556: 0000
12555: 0000
12554: 0000
```

```

12553: 0004
12552: 0007
12551: 0019
12550: 11B1
12549: 1110
12548: 0110
12547: 0310
12546: 0110
12545: 2819
12544: 3107
.
.
.
```

```

12297: 3100
12296: F025
12295: OFFC
12294: 0801
12293: 1482
12292: 127F
12291: 0804
12290: A406
12289: 126F
12288: 5260
.
.
.
```

```

3: XXXX
2: XXXX
1: XXXX
0: XXXX
```

5.3. Resumen de síntesis

En la figura 18 se muestra el diagrama de los componentes utilizados en la implementación de la RAM y en la figura 19 se muestra el reporte de componentes generado en Quartus Prime.

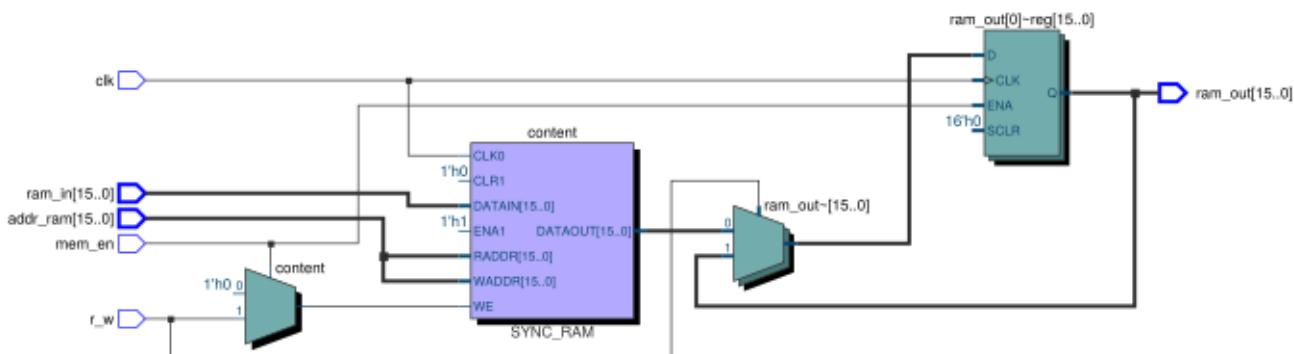


Figura 18: Esquema RTL de la RAM implementada en el procesador LC-3 generado por Quartus Prime.

```

Partition Statistics report for ram
Mon Jun 11 21:52:41 2018
Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition
+-----+
; Fitter Partition Statistics
+-----+
; Statistic ; Top ; hard_block:auto_generated_inst ;
+-----+
; Difficulty Clustering Region ; Low ; Low
+-----+
; Total logic elements ; 109 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;-- Combinational with no register ; 106 ; 0
;-- Register only ; 0 ; 0
;-- Combinational with a register ; 3 ; 0
+-----+
; Logic element usage by number of LUT inputs ;
;-- 4 input functions ; 84 ; 0
;-- 3 input functions ; 16 ; 0
;-- <=2 input functions ; 9 ; 0
;-- Register only ; 0 ; 0
+-----+
; Logic elements by mode ;
;-- normal mode ; 109 ; 0
;-- arithmetic mode ; 0 ; 0
+-----+
; Total registers ; 3 ; 0
;-- Dedicated logic registers ; 3 / 114480 ( < 1 % ) ; 0 / 114480 ( 0 % )
;-- I/O registers ; 0 ; 0
+-----+
; Total LABs: partially or completely used ; 16 / 7155 ( < 1 % ) ; 0 / 7155 ( 0 % )
+-----+
; Virtual pins ; 0 ; 0
; I/O pins ; 51 ; 0
; Embedded Multiplier 9-bit elements ; 0 / 532 ( 0 % ) ; 0 / 532 ( 0 % )
; Total memory bits ; 1048576 ; 0
; Total RAM block bits ; 1179648 ; 0
; M9K ; 128 / 432 ( 29 % ) ; 0 / 432 ( 0 % )
; Clock control block ; 1 / 24 ( 4 % ) ; 0 / 24 ( 0 % )
+-----+
; Connections ;
;-- Input Connections ; 0 ; 0
;-- Registered Input Connections ; 0 ; 0
;-- Output Connections ; 0 ; 0
;-- Registered Output Connections ; 0 ; 0
+-----+
; Internal Connections ;
;-- Total Connections ; 4575 ; 5
;-- Registered Connections ; 160 ; 0
+-----+
; External Connections ;
;-- Top ; 0 ; 0
;-- hard_block:auto_generated_inst ; 0 ; 0
+-----+
; Partition Interface ;
;-- Input Ports ; 35 ; 0
;-- Output Ports ; 16 ; 0
;-- Bidir Ports ; 0 ; 0
+-----+
; Registered Ports ;
;-- Registered Input Ports ; 0 ; 0
;-- Registered Output Ports ; 0 ; 0
+-----+
; Port Connectivity ;
;-- Input Ports driven by GND ; 0 ; 0
;-- Output Ports driven by GND ; 0 ; 0
;-- Input Ports driven by VCC ; 0 ; 0
;-- Output Ports driven by VCC ; 0 ; 0
;-- Input Ports with no Source ; 0 ; 0
;-- Output Ports with no Source ; 0 ; 0
;-- Input Ports with no Fanout ; 0 ; 0
;-- Output Ports with no Fanout ; 0 ; 0
+-----+

```

Figura 19: Reporte generado en Quartus Prime de los componentes utilizados en la implementación de la RAM.

6. LC-3

6.1. Descripción del procesador LC-3

Especificaciones: La implementación del procesador LC-3 se llevó a cabo incorporando la ALU, el banco de registros, el condition codes y la RAM. Adicionalmente se implementaron los siguientes componentes: 9 registros (tabla 9), 1 sumador (tabla 10), 11 multiplexores (tablas 11 a 20 y figura 28) y una máquina de estados (figura 21). Estos componentes se describen en el Anexo.

Las señales de entrada y salida del procesador LC-3 son las siguientes:

- clk : señal de entrada de reloj. 1 bit.
- reset : señal de entrada para inicializar los registros. 1 bit.
- data_in : señal portadora del valor de entrada que puede venir de un dispositivo de entrada (un teclado por ejemplo). 16 bits.
- data_out : señal de salida que contendrá el dato que se enviará a un dispositivo de salida (por ejemplo un monitor). 16 bits.

En la figura 20 se representa el esquema general del procesador LC-3 implementado.

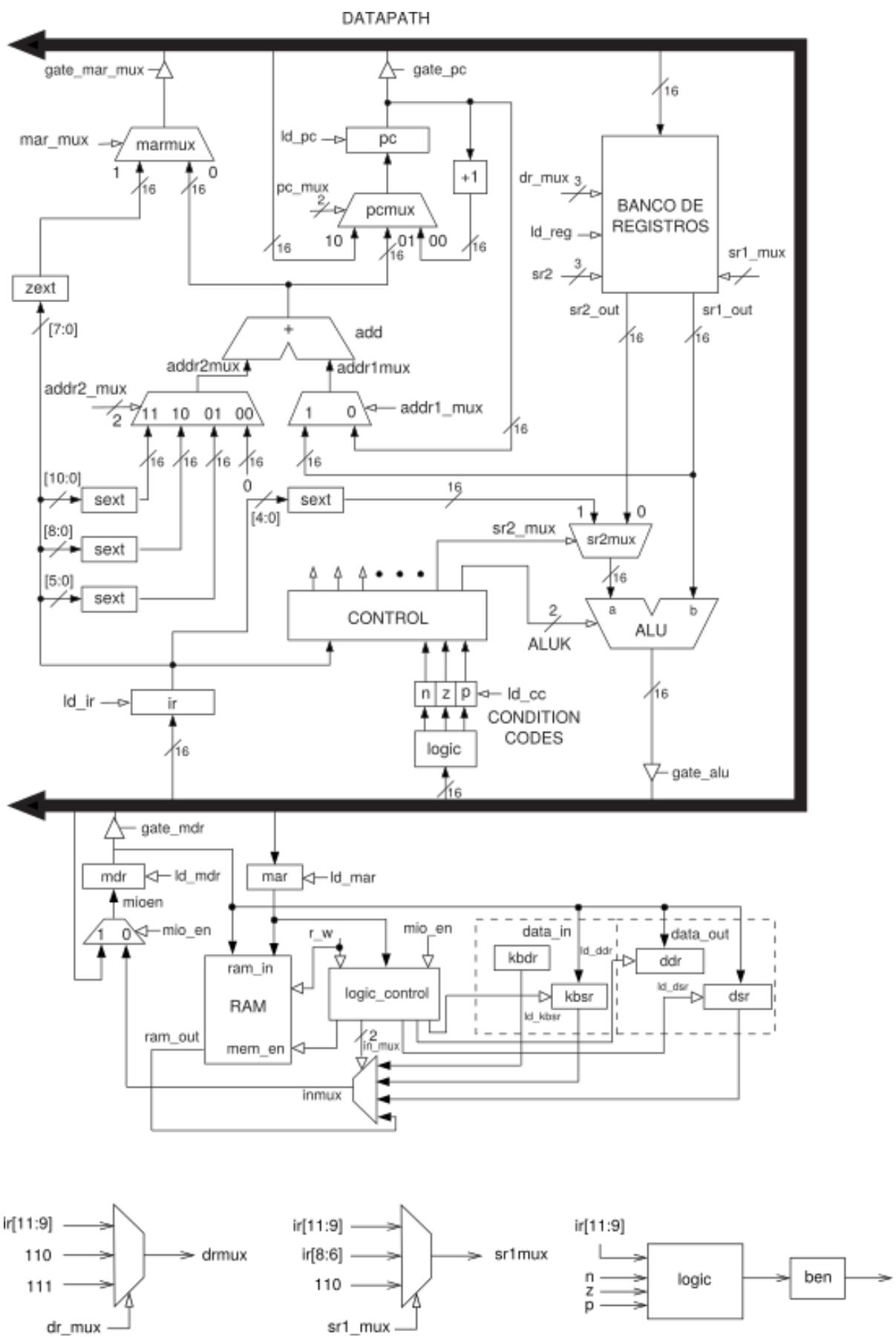


Figura 20: Diagrama simplificado del procesador LC-3 implementado.

La lógica de estados juntos con la lógica de control, se implementó siguiendo el diagrama de estados mostrado en la figura 21.

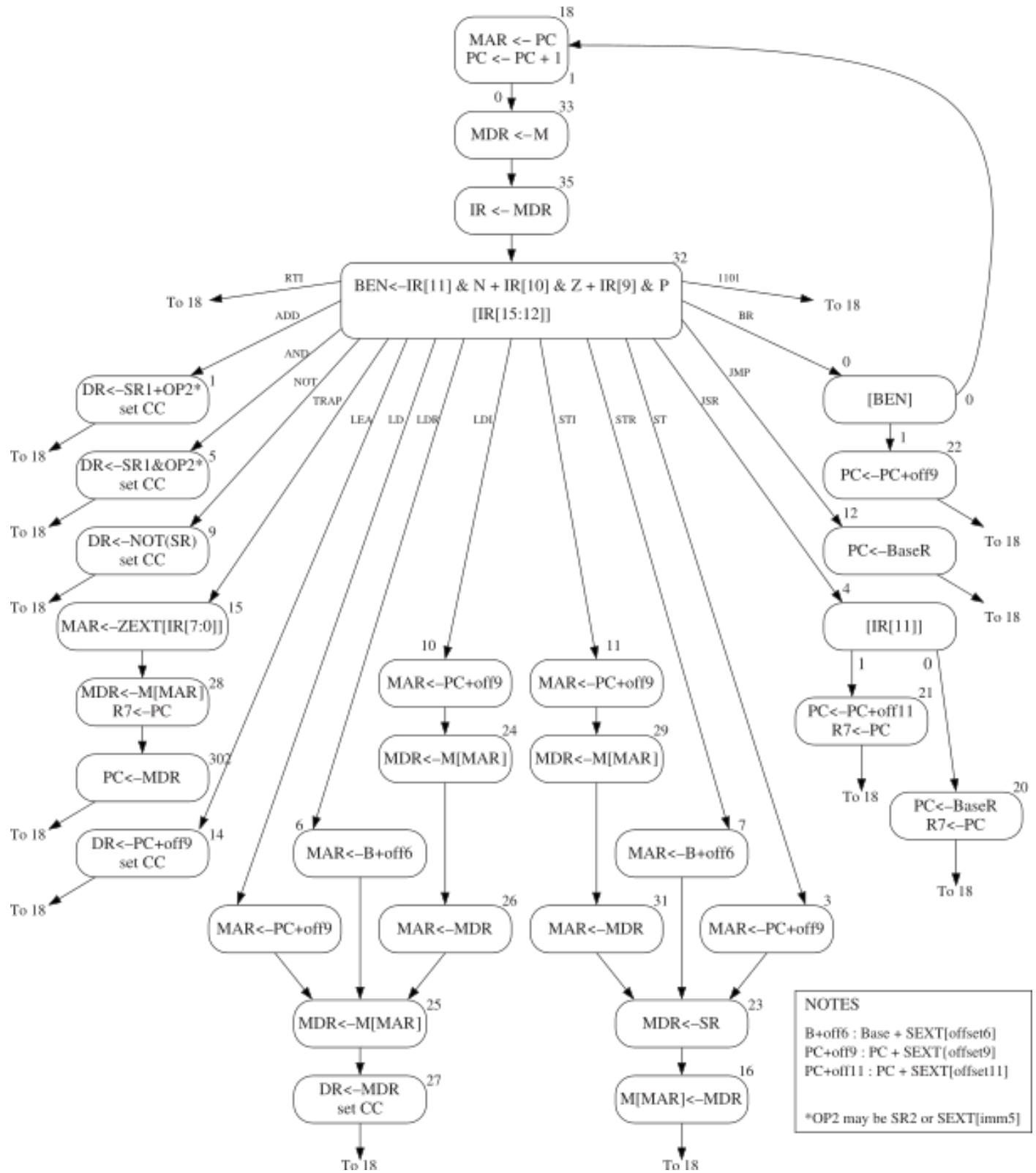


Figura 21: Diagrama de estados utilizado en la implementación del procesador LC-3.

En cada estado se especificó el estado siguiente y las habilitaciones correspondientes. El siguiente es un fragmento del código implementado que ejemplifica la lógica de estado siguiente y de control en cada estado.

```

WHEN e18 =>

    state_next <= e33;

    -- routing
    ld_mar <= '1';
    ld_mdr <= '0';
    ld_ir <= '0';
    ld_ben <= '0';
    ld_reg <= '0';
    ld_cc <= '0';
    ld_pc <= '1';

    gate_pc <= '1';
    gate_mdr <= '0';
    gate_alu <= '0';
    gate_mar_mux <= '0';

    addr1_mux <= '0';
    mar_mux <= '0';
    mio_en <= '1';
    r_w <= '0';
    sr2_mux <= '0';

    pc_mux <= "00";
    dr_mux <= "00";
    sr1_mux <= "00";
    addr2_mux <= "00";
    aluk <= "00";

```

El código completo de la implementación en VHDL se muestra a continuación.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY lc3 IS
    PORT(
        clk, reset : IN STD_LOGIC;
        data_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END lc3;

```

```

ARCHITECTURE behavioral OF lc3 IS

----- COMPONENT alu -----
COMPONENT alu IS
    PORT(
        aluk : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        a, b : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        y : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END COMPONENT;

----- COMPONENT banco_8reg16bits -----
COMPONENT banco_8reg16bits IS
    PORT(
        clk, reset, ld_reg : IN STD_LOGIC;
        data : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        dr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        sr1, sr2 : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        sr1out, sr2out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END COMPONENT;

----- COMPONENT cond_cod -----
COMPONENT cond_cod IS
    PORT(
        clk, reset, ld_cc : IN std_logic;
        data : IN std_logic_vector(15 DOWNTO 0);
        ir : IN std_logic_vector(11 DOWNTO 9);
        n, z, p, ben : OUT std_logic
    );
END COMPONENT;

COMPONENT ram IS
    PORT (
        clk : IN STD_LOGIC;
        r_w, mem_en : IN STD_LOGIC;
        ram_in : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        addr_ram : IN INTEGER RANGE 0 TO 65535;
        ram_out : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END COMPONENT;

-- STATES
TYPE state_type IS (e0, e1, e2, e3, e4, e5, e6, e7, e9, e10,
                    e11, e12,      e14, e15, e16,      e18,      e20,
                    e21, e22, e23, e24, e25, e26, e27, e28, e29, e30,
                    e31, e32, e33,      e35);

```

```

SIGNAL state_reg, state_next : state_type;

-- CONTROL REGISTER
SIGNAL mar_reg, mar_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL mdr_reg, mdr_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ir_reg, ir_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL pc_reg, pc_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ben_reg, ben_next : STD_LOGIC;
SIGNAL kbdr_reg, kbdr_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL kbsr_reg, kbsr_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ddr_reg, ddr_next : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL dsr_reg, dsr_next : STD_LOGIC_VECTOR(15 DOWNTO 0);

-- SIGNALS
SIGNAL datapath : STD_LOGIC_VECTOR(15 DOWNTO 0) := x"0000";
SIGNAL dr, sr1, sr2 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL sr1out, sr2out : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL sr2mux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL n, z, p, ben : STD_LOGIC;
SIGNAL alu_out : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL add : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL addr1mux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL addr2mux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL pcmux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL marmux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ram_in, ram_out : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL addr_ram : INTEGER RANGE 0 TO 65535;
SIGNAL mioen, inmux : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL logic_control : STD_LOGIC_VECTOR(5 DOWNTO 0);

-- 1 BIT CONTROL SIGNALS
SIGNAL ld_mar, ld_mdr, ld_ir, ld_ben, ld_reg, ld_cc, ld_pc,
       gate_pc, gate_mdr, gate_alu, gate_mar_mux,
       addr1_mux, mar_mux, r_w, mio_en, mem_en, sr2_mux,
       ld_kbsr, ld_dsr, ld_ddr : STD_LOGIC;

-- 2 BITS CONTROL SIGNALS
SIGNAL pc_mux, dr_mux, sr1_mux, addr2_mux, aluk,
       in_mux : STD_LOGIC_VECTOR(1 DOWNTO 0);

-- BEGIN ARCHITECTURE
BEGIN

-- MAKE COMPONENTS
uut_alu : alu PORT MAP(aluk, sr1out, sr2mux, alu_out);

uut_banco : banco_8reg16bits PORT MAP(clk, reset, ld_reg, datapath,

```

```

        dr, sr1, sr2,
        sr1out, sr2out);

uut_cond_cod : cond_cod PORT MAP(clk, reset, ld_cc, datapath,
        ir_reg(11 DOWNTO 9), n, z, p, ben);

uut_ram : ram PORT MAP(clk, r_w, mem_en, ram_in, addr_ram, ram_out);

-- add
add <= STD_LOGIC_VECTOR(UNSIGNED(addr1mux) + UNSIGNED(addr2mux));

-- pc mux
pcmux <= STD_LOGIC_VECTOR(UNSIGNED(pc_reg) + 1) WHEN pc_mux = "00" ELSE
        add WHEN pc_mux = "01" ELSE
        datapath;

-- dr mux
WITH dr_mux SELECT
        dr <= ir_reg(11 DOWNTO 9) WHEN "00",
                "110" WHEN "01",
                "111" WHEN OTHERS;

-- sr1 mux
WITH sr1_mux SELECT
        sr1 <= ir_reg(11 DOWNTO 9) WHEN "00",
                ir_reg(8 DOWNTO 6) WHEN "01",
                "110" WHEN OTHERS;

-- sr2
sr2 <= ir_reg(2 DOWNTO 0);

-- sr2 mux
sr2mux <= sr2out WHEN sr2_mux = '0' ELSE
        "000000000000" & ir_reg(4 DOWNTO 0) WHEN ir_reg(4) = '0' ELSE
        "111111111111" & ir_reg(4 DOWNTO 0);

-- addr1 mux
addr1mux <= pc_reg WHEN addr1_mux = '0' ELSE sr1out;

-- addr2 mux
addr2mux <= x"0000" WHEN addr2_mux = "00" ELSE
        "0000000000" & ir_reg(5 DOWNTO 0) WHEN addr2_mux="01" AND ir_reg(5)='0' ELSE
        "1111111111" & ir_reg(5 DOWNTO 0) WHEN addr2_mux="01" AND ir_reg(5)='1' ELSE
        "0000000" & ir_reg(8 DOWNTO 0) WHEN addr2_mux="10" AND ir_reg(8)='0' ELSE
        "1111111" & ir_reg(8 DOWNTO 0) WHEN addr2_mux="10" AND ir_reg(8)='1' ELSE
        "00000" & ir_reg(10 DOWNTO 0) WHEN addr2_mux="11" AND ir_reg(10)='0' ELSE
        "11111" & ir_reg(10 DOWNTO 0);

```

```

-- marmux
marmux <= add WHEN mar_mux = '0' ELSE
                  "00000000" & ir_reg(7 DOWNTO 0);

-- addr_ram
addr_ram <= TO_INTEGER(UNSIGNED(datapath));

-- mioen
mioen <= inmux WHEN mio_en = '0' ELSE datapath;

-- inmux
WITH in_mux SELECT
    inmux <= kbdr_reg WHEN "00",
                           kbsr_reg WHEN "01",
                           dsr_reg WHEN "10",
                           ram_out WHEN OTHERS;

-- CONTROL LOGIC FOR MEMORY-MAPPED I/O

logic_control <= "0XX000" WHEN mio_en = '0' ELSE
    "001000" WHEN mar_reg = x"FE00" AND mio_en = '1' AND r_w = '0' ELSE
    "011100" WHEN mar_reg = x"FE00" AND mio_en = '1' AND r_w = '1' ELSE
    "000000" WHEN mar_reg = x"FE02" AND mio_en = '1' AND r_w = '0' ELSE
    "011000" WHEN mar_reg = x"FE02" AND mio_en = '1' AND r_w = '1' ELSE
    "010000" WHEN mar_reg = x"FE04" AND mio_en = '1' AND r_w = '0' ELSE
    "011010" WHEN mar_reg = x"FE04" AND mio_en = '1' AND r_w = '1' ELSE
    "011000" WHEN mar_reg = x"FE06" AND mio_en = '1' AND r_w = '0' ELSE
    "011001" WHEN mar_reg = x"FE06" AND mio_en = '1' AND r_w = '1' ELSE
    "111000" WHEN mio_en = '1' AND r_w = '0' ELSE
    "1XX001";

mem_en <= logic_control(5);
in_mux <= logic_control(4 DOWNTO 3);
ld_kbsr <= logic_control(2);
ld_dsr <= logic_control(1);
ld_ddr <= logic_control(0);

-- CONTROL PATH : STATE REGISTER
PROCESS(clk, reset)
BEGIN
    IF reset='1' THEN
        state_reg <= e18;
    ELSIF(clk'EVENT AND clk='1') THEN
        state_reg <= state_next;
    END IF;

```

```

END PROCESS;

-- DATA PATH : DATA REGISTER
PROCESS(clk, reset)
BEGIN
    IF reset='1' THEN
        mar_reg <= (OTHERS => '0');
        mdr_reg <= (OTHERS => '0');
        ir_reg <= (OTHERS => '0');
        --pc_reg <= (OTHERS => '0');
        --pc_reg <= "00000000000010000";
        pc_reg <= x"3000";
        ben_reg <= '0';
        kbdr_reg <= (OTHERS => '0');
        kbsr_reg <= (OTHERS => '0');
        dsr_reg <= (OTHERS => '0');
        ddr_reg <= (OTHERS => '0');
    ELSIF(clk'EVENT AND clk='1') THEN
        mar_reg <= mar_next;
        mdr_reg <= mdr_next;
        ir_reg <= ir_next;
        pc_reg <= pc_next;
        ben_reg <= ben_next;
        kbdr_reg <= kbdr_next;
        kbsr_reg <= kbsr_next;
        dsr_reg <= dsr_next;
        ddr_reg <= ddr_next;
    END IF;
END PROCESS;

-- NEXT-STATE LOGIC DATA REGISTER
mar_next <= datapath WHEN ld_mar = '1' ELSE mar_reg;
mdr_next <= mioen WHEN ld_mdr = '1' ELSE mdr_reg;
ir_next <= datapath WHEN ld_ir = '1' ELSE ir_reg;
pc_next <= pcmux WHEN ld_pc = '1' ELSE pc_reg;
ben_next <= ben WHEN ld_ben = '1' ELSE ben_reg;
kbdr_next <= data_in;
kbsr_next <= mdr_reg when ld_kbsr = '1' ELSE kbsr_reg;
dsr_next <= mdr_reg when ld_dsr = '1' ELSE dsr_reg;
ddr_next <= mdr_reg when ld_ddr = '1' ELSE ddr_reg;

-- datapath GATES
datapath <= marmux WHEN gate_mar_mux='1' AND gate_pc='0'
                    AND gate_alu='0' AND gate_mdr='0' ELSE
                    pc_reg WHEN gate_mar_mux = '0' AND gate_pc = '1'
                    AND gate_alu = '0' AND gate_mdr = '0' ELSE
                    alu_out WHEN gate_mar_mux = '0' AND gate_pc = '0'

```

```

        AND gate_alu = '1' AND gate_mdr = '0' ELSE
        mdr_reg;

--CONTROL & DATA PATH : NEXT-STATE & OUTPUT LOGICc
PROCESS(state_reg, ir_reg, ben)

BEGIN
    CASE state_reg IS
        WHEN e18 =>

            state_next <= e33;

            -- routing
            ld_mar <= '1';
            ld_mdr <= '0';
            ld_ir <= '0';
            ld_ben <= '0';
            ld_reg <= '0';
            ld_cc <= '0';
            ld_pc <= '1';

            gate_pc <= '1';
            gate_mdr <= '0';
            gate_alu <= '0';
            gate_mar_mux <= '0';

            addr1_mux <= '0';
            mar_mux <= '0';
            mio_en <= '1';
            r_w <= '0';
            sr2_mux <= '0';

            pc_mux <= "00";
            dr_mux <= "00";
            sr1_mux <= "00";
            addr2_mux <= "00";
            aluk <= "00";

        WHEN e33 =>

            state_next <= e35;

            -- routing
            ld_mar <= '0';
            ld_mdr <= '1';
            ld_ir <= '0';
            ld_ben <= '0';

```

```

ld_reg <= '0';
ld_cc <= '0';
ld_pc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";

```

WHEN e35 =>

```
state_next <= e32;
```

```
-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '1';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
ld_pc <= '0';

gate_pc <= '0';
gate_mdr <= '1';
gate_alu <= '0';
gate_mar_mux <= '0';


```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';


```

```
pc_mux <= "00";
dr_mux <= "00";
```

```

        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e32 =>

        IF(ir_reg(15 DOWNTO 12)="0001") THEN -- ADD
            state_next <= e1;
        ELSIF(ir_reg(15 DOWNTO 12)="0101") THEN -- AND
            state_next <= e5;
        ELSIF(ir_reg(15 DOWNTO 12)="1001") THEN -- NOT
            state_next <= e9;
        ELSIF(ir_reg(15 DOWNTO 12)="1111") THEN -- TRAP
            state_next <= e15;
        ELSIF(ir_reg(15 DOWNTO 12)="1110") THEN -- LEA
            state_next <= e14;
        ELSIF(ir_reg(15 DOWNTO 12)="0010") THEN -- LD
            state_next <= e2;
        ELSIF(ir_reg(15 DOWNTO 12)="0110") THEN -- LDR
            state_next <= e6;
        ELSIF(ir_reg(15 DOWNTO 12)="1010") THEN -- LDI
            state_next <= e10;
        ELSIF(ir_reg(15 DOWNTO 12)="1011") THEN -- STI
            state_next <= e11;
        ELSIF(ir_reg(15 DOWNTO 12)="0111") THEN -- STR
            state_next <= e7;
        ELSIF(ir_reg(15 DOWNTO 12)="0011") THEN -- ST
            state_next <= e3;
        ELSIF(ir_reg(15 DOWNTO 12)="0100") THEN -- JSR
            state_next <= e4;
        ELSIF(ir_reg(15 DOWNTO 12)="1100") THEN -- JMP
            state_next <= e12;
        ELSIF(ir_reg(15 DOWNTO 12)="0000") THEN -- BR
            state_next <= e0;
        ELSE -- default RTI and others
            state_next <= e18;
        END IF;

-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '1';
ld_reg <= '0';
ld_cc <= '1';

```

```

        gate_pc <= '0';
        gate_mdr <= '0';
        gate_alu <= '0';
        gate_mar_mux <= '0';

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e1 => -- ADD

state_next <= e18;

-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '1';
ld_cc <= '1';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '1';
gate_mar_mux <= '0';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
IF(ir_reg(5) = '0') THEN
    sr2_mux <= '0';
ELSE
    sr2_mux <= '1';
END IF;

pc_mux <= "00";
dr_mux <= "00";

```

```

        sr1_mux <= "01";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e5 => -- AND

        state_next <= e18;

        -- routing
        ld_mar <= '0';
        ld_mdr <= '0';
        ld_ir <= '0';
        ld_pc <= '0';
        ld_ben <= '0';
        ld_reg <= '1';
        ld_cc <= '1';

        gate_pc <= '0';
        gate_mdr <= '0';
        gate_alu <= '1';
        gate_mar_mux <= '0';

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        IF(ir_reg(5) = '0') THEN
            sr2_mux <= '0';
        ELSE
            sr2_mux <= '1';
        END IF;

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "01";
        addr2_mux <= "00";
        aluk <= "11";

WHEN e9 => -- NOT

        state_next <= e18;

        -- routing
        ld_mar <= '0';
        ld_mdr <= '0';
        ld_ir <= '0';
        ld_pc <= '0';

```

```

ld_ben <= '0';
ld_reg <= '1';
ld_cc <= '1';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '1';
gate_mar_mux <= '0';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "01";
addr2_mux <= "00";
aluk <= "01";

```

WHEN e15 => -- *TRAP_1*

```
state_next <= e28;
```

```
-- routing
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';

addr1_mux <= '0';
mar_mux <= '1';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
```

```

        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e28 => -- TRAP_2

        state_next <= e30;

        -- routing
        ld_mar <= '0';
        ld_mdr <= '1';
        ld_ir <= '0';
        ld_pc <= '0';
        ld_ben <= '0';
        ld_reg <= '1';
        ld_cc <= '0';

        gate_pc <= '1';
        gate_mdr <= '0';
        gate_alu <= '0';
        gate_mar_mux <= '0';

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "11";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e30 => -- TRAP_3
        state_next <= e18;

        -- routing
        ld_mar <= '0';
        ld_mdr <= '0';
        ld_ir <= '0';
        ld_pc <= '1';
        ld_ben <= '0';
        ld_reg <= '0';
        ld_cc <= '0';

        gate_pc <= '0';

```

```
    gate_mdr <= '1';
    gate_alu <= '0';
    gate_mar_mux <= '0';
```

```
    addr1_mux <= '0';
    mar_mux <= '0';
    mio_en <= '0';
    r_w <= '0';
    sr2_mux <= '0';
```

```
    pc_mux <= "10";
    dr_mux <= "00";
    sr1_mux <= "00";
    addr2_mux <= "00";
    aluk <= "00";
```

WHEN e14 => -- *LEA*

```
    state_next <= e18;
```

-- *routing*

```
    ld_mar <= '0';
    ld_mdr <= '0';
    ld_ir <= '0';
    ld_pc <= '0';
    ld_ben <= '0';
    ld_reg <= '1';
    ld_cc <= '1';
```

```
    gate_pc <= '0';
    gate_mdr <= '0';
    gate_alu <= '0';
    gate_mar_mux <= '1';
```

```
    addr1_mux <= '0';
    mar_mux <= '0';
    mio_en <= '0';
    r_w <= '0';
    sr2_mux <= '0';
```

```
    pc_mux <= "00";
    dr_mux <= "00";
    sr1_mux <= "00";
    addr2_mux <= "10";
    aluk <= "00";
```

WHEN e2 => -- *LD*

```
state_next <= e25;
```

```
-- routing
```

```
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
```

```
gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';
```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';
```

```
pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "10";
aluk <= "00";
```

```
WHEN e25 => -- LD*_2
```

```
state_next <= e27;
```

```
-- routing
```

```
ld_mar <= '0';
ld_mdr <= '1';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
```

```
gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';
```

```

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e27 => -- LD*_3
    state_next <= e18;

    -- routing
    ld_mar <= '0';
    ld_mdr <= '0';
    ld_ir <= '0';
    ld_pc <= '0';
    ld_ben <= '0';
    ld_reg <= '1';
    ld_cc <= '1';

    gate_pc <= '0';
    gate_mdr <= '1';
    gate_alu <= '0';
    gate_mar_mux <= '0';

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e6 => -- LDR

    state_next <= e25;

    -- routing
    ld_mar <= '1';

```

```

ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';

addr1_mux <= '1';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "01";
addr2_mux <= "01";
aluk <= "00";

WHEN e10 => -- LDI_1
state_next <= e24;

-- routing
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

```

```

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "10";
aluk <= "00";

```

WHEN e24 => -- LDI_2

```
state_next <= e26;
```

-- *routing*

```

ld_mar <= '0';
ld_mdr <= '1';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

```

```

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

```

```

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';

```

```

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";

```

WHEN e26 => -- LDI_3

```
state_next <= e25;
```

-- *routing*

```

ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';

```

```

ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '1';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
alu_k <= "00";

WHEN e11 => -- STI_1

state_next <= e29;

-- routing
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "10";

```

```

        aluk <= "00";

WHEN e29 => -- STI_2

        state_next <= e31;

        -- routing
        ld_mar <= '0';
        ld_mdr <= '1';
        ld_ir <= '0';
        ld_pc <= '0';
        ld_ben <= '0';
        ld_reg <= '0';
        ld_cc <= '0';

        gate_pc <= '0';
        gate_mdr <= '0';
        gate_alu <= '0';
        gate_mar_mux <= '0';

        addr1_mux <= '0';
        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e31 => -- STI_3

        state_next <= e23;

        -- routing
        ld_mar <= '1';
        ld_mdr <= '0';
        ld_ir <= '0';
        ld_pc <= '0';
        ld_ben <= '0';
        ld_reg <= '0';
        ld_cc <= '0';

        gate_pc <= '0';
        gate_mdr <= '1';

```

```
gate_alu <= '0';
gate_mar_mux <= '0';
```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';
```

```
pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";
```

WHEN e23 => -- *ST*_1*

```
state_next <= e16;
```

-- *routing*

```
ld_mar <= '0';
ld_mdr <= '1';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
```

```
gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '1';
gate_mar_mux <= '0';
```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';
```

```
pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "10";
```

WHEN e16 => -- *ST*_2*

```
state_next <= e18;
```

```
-- routing
```

```
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
```

```
gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';
```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '1';
sr2_mux <= '0';
```

```
pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";
```

```
WHEN e7 => -- STR
```

```
state_next <= e23;
```

```
-- routing
```

```
ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
```

```
gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';
```

```
addr1_mux <= '1';
```

```

    mar_mux <= '0';
    mio_en <= '1';
    r_w <= '0';
    sr2_mux <= '0';

```

```

    pc_mux <= "00";
    dr_mux <= "00";
    sr1_mux <= "01";
    addr2_mux <= "01";
    aluk <= "00";

```

WHEN e3 => -- ST

```
state_next <= e23;
```

-- *routing*

```

ld_mar <= '1';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

```

```

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '1';

```

```

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '1';
r_w <= '0';
sr2_mux <= '0';

```

```

pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "10";
aluk <= "00";

```

WHEN e4 => -- ISR

```

IF(ir_reg(11)='1') THEN
    state_next <= e21;
ELSE
    state_next <= e20;

```

```
END IF;
```

```
-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
ld_pc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';
```

```
addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';
```

```
pc_mux <= "00";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";
```

```
WHEN e21 => -- ISR_a
```

```
state_next <= e18;
```

```
-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '1';
ld_ben <= '0';
ld_reg <= '1';
ld_cc <= '0';

gate_pc <= '1';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '0';
```

```
    mar_mux <= '0';
    mio_en <= '0';
    r_w <= '0';
    sr2_mux <= '0';
```

```
    pc_mux <= "01";
    dr_mux <= "11";
    sr1_mux <= "00";
    addr2_mux <= "11";
    aluk <= "00";
```

WHEN e20 => -- *ISR_b*

```
state_next <= e18;
```

-- *routing*

```
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '1';
ld_ben <= '0';
ld_reg <= '1';
ld_cc <= '0';
```

```
gate_pc <= '1';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';
```

```
addr1_mux <= '1';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';
```

```
pc_mux <= "01";
dr_mux <= "11";
sr1_mux <= "00";
addr2_mux <= "00";
aluk <= "00";
```

WHEN e12 => -- *JMP*

```
state_next <= e18;
```

-- *routing*

```
ld_mar <= '0';
```

```

ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '1';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '1';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "01";
dr_mux <= "00";
sr1_mux <= "01";
addr2_mux <= "00";
aluk <= "00";

```

WHEN e0 => -- *BR_1*

```

IF(ben = '1') THEN
    state_next <= e22;
ELSE
    state_next <= e18;
END IF;

```

```

-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';
ld_pc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '0';

```

```

        mar_mux <= '0';
        mio_en <= '0';
        r_w <= '0';
        sr2_mux <= '0';

        pc_mux <= "00";
        dr_mux <= "00";
        sr1_mux <= "00";
        addr2_mux <= "00";
        aluk <= "00";

WHEN e22 => -- BR_2

state_next <= e18;

-- routing
ld_mar <= '0';
ld_mdr <= '0';
ld_ir <= '0';
ld_pc <= '1';
ld_ben <= '0';
ld_reg <= '0';
ld_cc <= '0';

gate_pc <= '0';
gate_mdr <= '0';
gate_alu <= '0';
gate_mar_mux <= '0';

addr1_mux <= '0';
mar_mux <= '0';
mio_en <= '0';
r_w <= '0';
sr2_mux <= '0';

pc_mux <= "01";
dr_mux <= "00";
sr1_mux <= "00";
addr2_mux <= "10";
aluk <= "00";

END CASE;
END PROCESS;
END behavioral;
```

6.2. Simulación

Como se especificó en la sección correspondiente a la RAM, el programa utilizado para la simulación del procesador fue el ejemplo 6.4 de [1]. Como se puede apreciar en el diagrama de estados, el estado inicial es el estado 18, y a él se regresa nuevamente cada vez que termina algún conjunto de instrucciones. En la simulación realizada con modelsim se puede apreciar el resultado de cada instrucción del ejemplo utilizado en los estados 18. El tiempo total de simulación fue de 10 ns.

La figura 22 muestra las dos primeras instrucciones utilizadas en la simulación, 1) poner un cero en el registro 1 y 2) poner un 15 en el registro 1.



Figura 22: Primer fragmento de simulación realizada en modelsim del procesador LC3.

La figura 23 muestra 3 ns de simulación en la que se realizaron las primeras 5 instrucciones.



Figura 23: Segundo fragmento de simulación realizada en modelsim del procesador LC3.

La figura 24 muestra a partir de la 5 operación el ciclo que se estará repitiendo en la simulación (colores rosa, verde, azul y naranja).

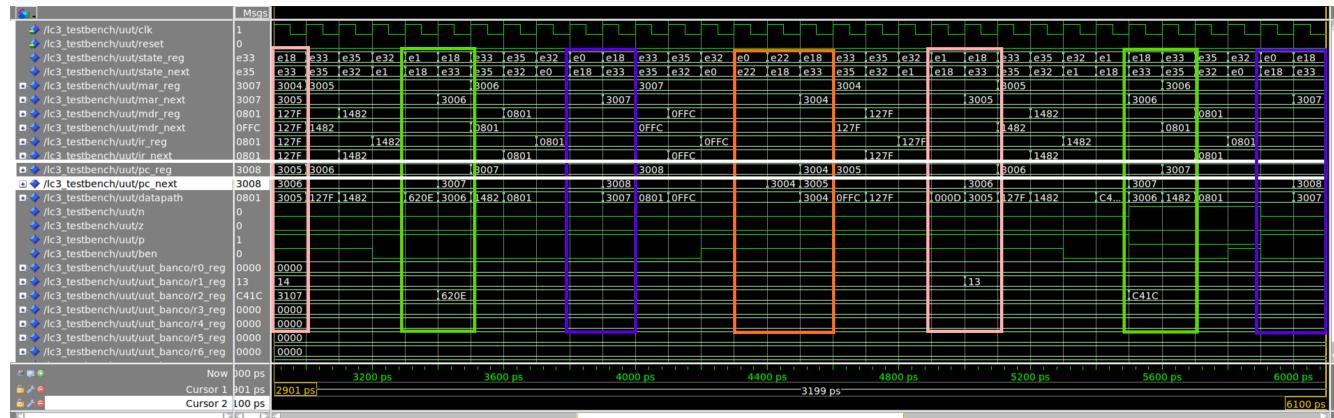


Figura 24: Tercer fragmento de simulación realizada en modelsim del procesador LC3.

Finalmente la figura 25 muestra la simulación completa de 1 ns.

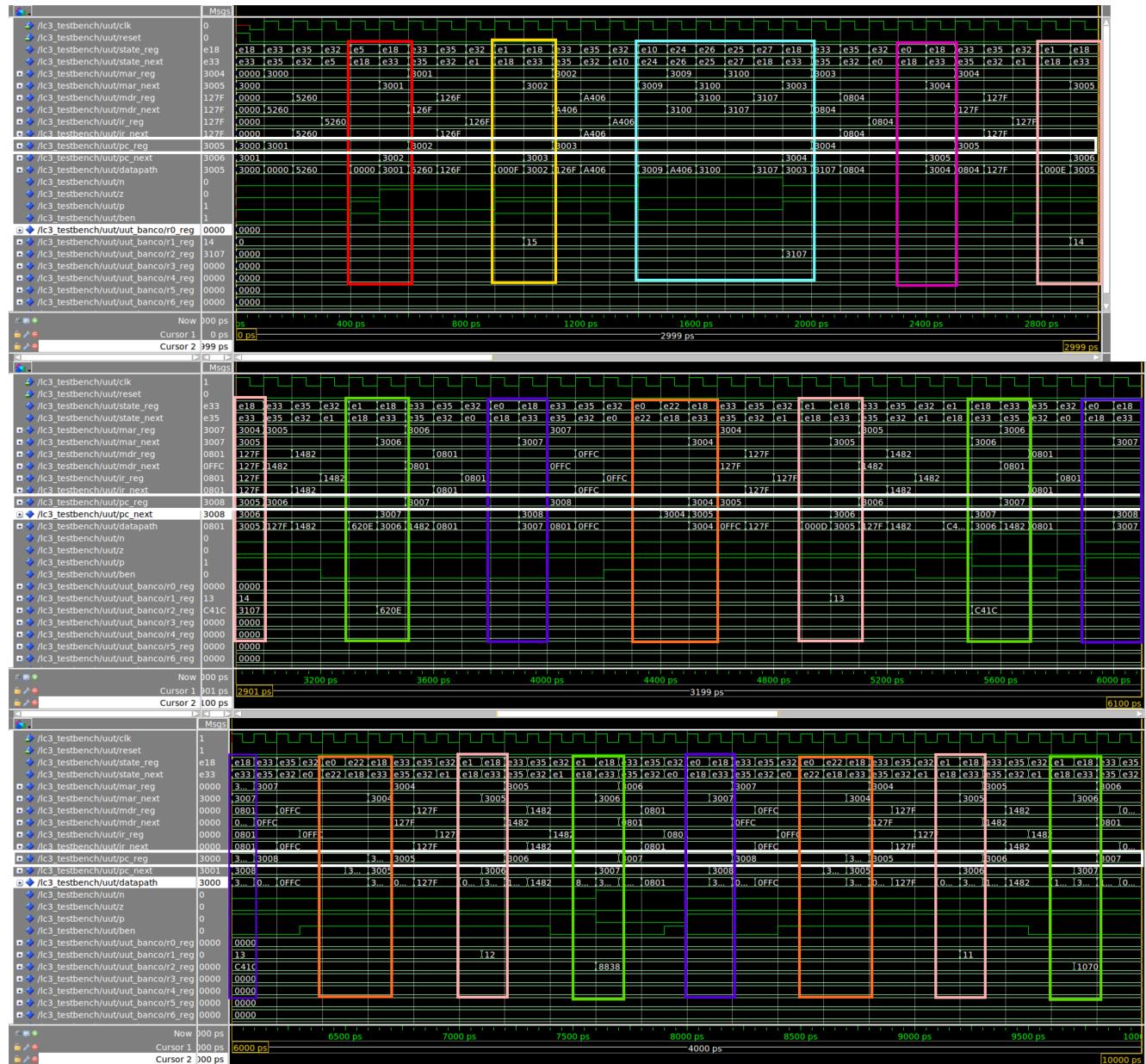


Figura 25: Simulación completa (10 ns) del procesador LC3 realizada en modelsim.

El código para simular el procesador LC-3 implementado, que se muestra a continuación, está recortado ya que es un ciclo de reloj repetido hasta completar 10 ns.

```
LIBRARY IEEE;  
USE IEEE.STD.LOGIC_1164.ALL;
```

```
ENTITY lc3_testbench IS  
END lc3 testbench;
```

ARCHITECTURE behavioral OF lc3 testbench IS

COMPONENT 1c3 TS

```

PORT(
    clk, reset : IN STD_LOGIC;
    data_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    data_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END COMPONENT;

SIGNAL clk, reset : STD_LOGIC;
SIGNAL data_in, data_out : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN
    -- instantiate the circuit under test
    uut : lc3 PORT map(clk, reset, data_in, data_out);

    -- test vector generator
PROCESS
BEGIN

    -- initial time
    reset <= '1';
    -----
    WAIT FOR 50 ps; clk <= '0';
    reset <= '0';
    WAIT FOR 50 ps; clk <= '1';

    WAIT FOR 50 ps; clk <= '0';
    WAIT FOR 50 ps; clk <= '1';

    WAIT FOR 50 ps; clk <= '0';
    WAIT FOR 50 ps; clk <= '1';

    WAIT FOR 50 ps; clk <= '0';
    WAIT FOR 50 ps; clk <= '1';

    .
    .
    .
    -----
    -- Simulation total time = 10 ns

END PROCESS;

END behavioral;

```

6.3. Resumen de síntesis

En la figura 26 se muestra el diagrama de los componentes utilizados en la implementación completa del procesador LC-3 y en la figura 27 se muestra el reporte de componentes generado en Quartus Prime.

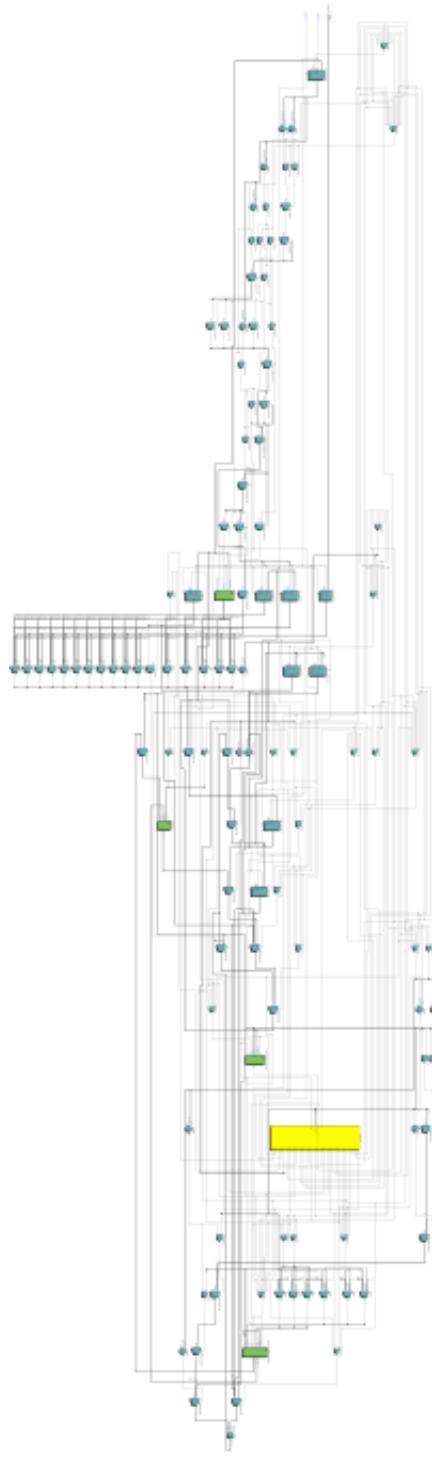


Figura 26: Esquema RTL del procesador LC-3 implementado generado por Quartus Prime.

```

Partition Statistics report for lc3
Tue Jun 12 00:36:18 2018
Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition
+-----+
; Fitter Partition Statistics
+-----+
; Statistic ; Top ; hard_block:auto_generated_inst ;
+-----+
; Difficulty Clustering Region ; Low ; Low
+-----+
; Total logic elements ; 0 / 114480 ( 0 % ) ; 0 / 114480 ( 0 % )
;-- Combinational with no register ; 0 ; 0
;-- Register only ; 0 ; 0
;-- Combinational with a register ; 0 ; 0
+-----+
; Logic element usage by number of LUT inputs ;
;-- 4 input functions ; 0 ; 0
;-- 3 input functions ; 0 ; 0
;-- <=2 input functions ; 0 ; 0
;-- Register only ; 0 ; 0
+-----+
; Logic elements by mode ;
;-- normal mode ; 0 ; 0
;-- arithmetic mode ; 0 ; 0
+-----+
; Total registers ; 0 ; 0
;-- Dedicated logic registers ; 0 / 114480 ( 0 % ) ; 0 / 114480 ( 0 % )
;-- I/O registers ; 0 ; 0
+-----+
; Total LABs: partially or completely used ; 0 / 7155 ( 0 % ) ; 0 / 7155 ( 0 % )
+-----+
; Virtual pins ; 0 ; 0
; I/O pins ; 34 ; 0
+-----+
; Embedded Multiplier 9-bit elements ; 0 / 532 ( 0 % ) ; 0 / 532 ( 0 % )
+-----+
; Total memory bits ; 0 ; 0
; Total RAM block bits ; 0 ; 0
+-----+
; Connections ;
;-- Input Connections ; 0 ; 0
;-- Registered Input Connections ; 0 ; 0
;-- Output Connections ; 0 ; 0
;-- Registered Output Connections ; 0 ; 0
+-----+
; Internal Connections ;
;-- Total Connections ; 34 ; 5
;-- Registered Connections ; 0 ; 0
+-----+
; External Connections ;
;-- Top ; 0 ; 0
;-- hard_block:auto_generated_inst ; 0 ; 0
+-----+
; Partition Interface ;
;-- Input Ports ; 18 ; 0
;-- Output Ports ; 16 ; 0
;-- Bidir Ports ; 0 ; 0
+-----+
; Registered Ports ;
;-- Registered Input Ports ; 0 ; 0
;-- Registered Output Ports ; 0 ; 0
+-----+
; Port Connectivity ;
;-- Input Ports driven by GND ; 0 ; 0
;-- Output Ports driven by GND ; 0 ; 0
;-- Input Ports driven by VCC ; 0 ; 0
;-- Output Ports driven by VCC ; 0 ; 0
;-- Input Ports with no Source ; 0 ; 0
;-- Output Ports with no Source ; 0 ; 0
;-- Input Ports with no Fanout ; 0 ; 0
;-- Output Ports with no Fanout ; 0 ; 0
+-----+

```

Figura 27: Reporte generado en Quartus Prime de los componentes utilizados en la implementación del procesador LC-3.

7. ANEXO

7.1. Componentes implementados en el procesador LC-3

Tabla 9: Registros implementados para el flujo de datos en el procesador LC-3.

Nombre	Entrada (next)	Salida (reg)	Señal de control
mar	Recibe el valor presente en el <i>datapath</i> .	Conecta el valor recibido a la RAM, haciendo una conversión a decimal para servir como apuntador.	ld_mar
mdr	Recibe el valor de salida del multiplexor <i>mioen</i> .	Conecta su valor de salida al datapath condicionado por la compuerta <i>gate_mdr</i> , a la RAM que lo utiliza como valor de entrada y al registro <i>ddr</i> .	ld_mdr
ir	Recibe el valor presente en el <i>datapath</i> .	Conecta su salida con la unidad de control.	ld_ir
pc	Recibe el valor del multiplexor <i>pcmux</i> .	Conecta su valor de salida al datapath condicionado por la compuerta <i>gate_pc</i> .	ld_pc
ben	Recibe el valor del <i>Condition Codes</i> .	Se utiliza su valor de salida como condicionante en el estado 0.	ld_ben
kbdr	Recibe el valor del dispositivo de entrada (por ejemplo un teclado).	Envía el valor de salida al multiplexor <i>inmux</i> .	ld_kbdr
kbsr	Utiliza la posición del bit más significativo para indicar si se ha recibido un valor a través de un dispositivo de entrada	Envía el valor de salida al multiplexor <i>inmux</i> .	ld_kbsr
ddr	Recibe el valor de registro <i>mdr</i> .	Envía el valor al dispositivo de salida (un monitor por ejemplo).	ld_ddr
dsr	Utiliza la posición del bit más significativo para indicar si se contiene un valor para enviar al dispositivo de salida.	Envía su valor de salida al multiplexor <i>inmux</i> .	ld_dsr

Tabla 10: Componente sumador.

Nombre	Entradas	Salida
add	addr1mux, addr2mux	addr1mux + addr2mux

Tabla 11: Tabla de verdad del componente *pcmux*

pc_mux	pcmux
00	pc + 1
01	add
10	datapath

Tabla 12: Tabla de verdad del componente *dr*

dr_mux	dr
00	ir[11:9]
01	110
1X	111

Tabla 13: Tabla de verdad del componente *sr1*

sr1_mux	sr1
00	ir[11:9]
01	ir[8:6]
1X	110

Tabla 14: Tabla de verdad del componente *sr2mux*

sr2_mux	ir(4)	sr2mux
0	X	sr2out
1	0	000000000000
1	1	111111111111

Tabla 15: Tabla de verdad del componente *addr1mux*

addr1_mux	addr1mux
0	pc
1	sr1out

Tabla 16: Tabla de verdad del componente *addr2mux*

addr2_mux	ir[5]	ir[8]	ir[10]	addr2mux
00	X	X	X	x0000
01	0	X	X	0000000000 & ir[5:0]
01	1	X	X	1111111111 & ir[5:0]
10	X	0	X	0000000 & ir[8:0]
10	X	1	X	1111111 & ir[8:0]
11	X	X	0	00000 & ir[10:0]
11	X	X	1	11111 & ir[10:0]

Tabla 17: Tabla de verdad del componente *marmux*

mar_mux	marmux
0	add
1	00000000 & ir[7:0]

Tabla 18: Tabla de verdad del componente *mioen*

mio_en	mioen
0	inmux
1	datapath

Tabla 19: Tabla de verdad del componente *inmux*

in_mux	inmux
00	kbdr
01	kbsr
10	dscr
11	ram_out

Tabla 20: Tabla de verdad para permitir el flujo de datos hacia el *datapath* por medio de las compuertas.

gate_mar_mux	gate_pc	gate_alu	gate_mdr	datapath
1	0	0	0	marmux
0	1	0	0	pc
0	0	1	0	alu
0	0	0	1	mdr

mar	mio_en	r_w	mem_en	in_mux	Id_kbsr	Id_dsr	Id_ddr
xFE00	0	R	0	X	0	0	0
xFE00	0	W	0	X	0	0	0
xFE00	1	R	0	KBSR	0	0	0
xFE00	1	W	0	X	1	0	0
xFE02	0	R	0	X	0	0	0
xFE02	0	W	0	X	0	0	0
xFE02	1	R	0	KBDR	0	0	0
xFE02	1	W	0	X	0	0	0
xFE04	0	R	0	X	0	0	0
xFE04	0	W	0	X	0	0	0
xFE04	1	R	0	DSR	0	0	0
xFE04	1	W	0	X	0	1	0
xFE06	0	R	0	X	0	0	0
xFE06	0	W	0	X	0	0	0
xFE06	1	R	0	X	0	0	0
xFE06	1	W	0	X	0	0	1
other	0	R	0	X	0	0	0
other	0	W	0	X	0	0	0
other	1	R	1	mem	0	0	0
other	1	W	1	X	0	0	0

Figura 28: Tabla de verdad utilizada para la lógica de control de direccionamiento de entrada y salida.

Bibliografía

- [1] Y. N. Patt and S. J. Patel. *Introduction to computing systems from bits and gates to C and beyond*. 2ed 2005. McGrawHill.