Objetivo

Familiarizar al alumno en el conocimiento del secuenciador básico, el cual es una parte fundamental del procesador.

Introducción

Secuenciador básico

Para el diseño de los módulos de control de una computadora se requieren máquinas de estados que sean capaces de ejecutar algoritmos más complejos. Haciendo modificaciones y agregando componentes a la variante del direccionamiento implícito se pueden crear máquinas de estados que efectúen cartas ASM con llamadas a subrutinas, estructuras DO WHILE, iteraciones tipo FOR, entre otras. Los dispositivos que son capaces de efectuar este tipo de operaciones son llamados secuenciadores.

A continuación, se muestra el diagrama de bloques de un secuenciador básico. Como puede observar en el diagrama, la dirección del estado siguiente, dada por el bus Y, puede venir de dos lugares posibles: Del registro μPC o de la entrada D.

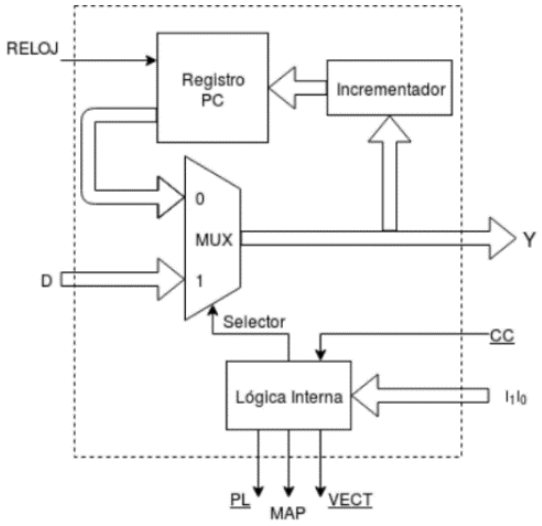

Figura 1 Diagrama de bloques interno de un secuenciador básico

El secuenciador cuenta con una lógica interna que se encarga de generar las señales que controlan al multiplexor. Dependiendo de la instrucción dada por las líneas I_1 e I_0 y de la línea CC, la lógica es capaz de seleccionar entre la salida del registro μPC o la entrada D.

La lógica interna también genera las líneas PL, MAP y VECT, las cuales seleccionan unos registros cuyas salidas están conectadas a la entrada D del secuenciador. De esta forma la dirección de salto puede venir de tres lugares distintos.

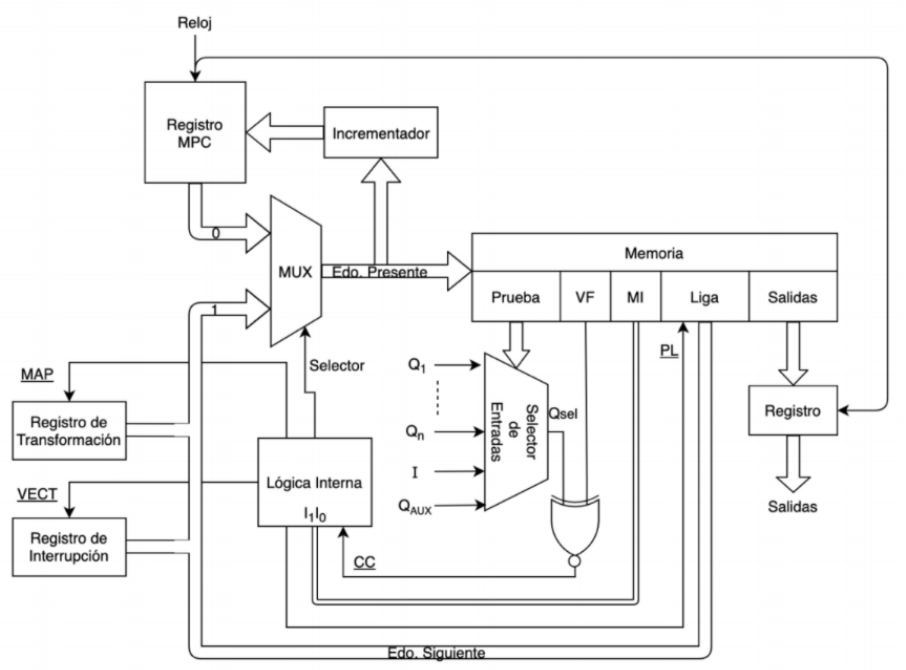

Figura 2 Diagrama del secuenciador básico conectado a una memora

A continuación, se muestran las instrucciones que el secuenciador puede ejecutar y su representación en carta ASM.

Paso Contiguo (C) [00]

En la instrucción continúa la dirección del estado siguiente la proporciona el registro μPC.

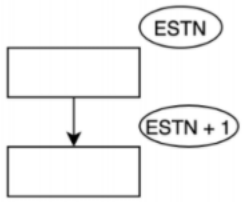

Figura 3 Carta ASM de la microinstrucción Paso Contiguo.

Salto Condicional (SCO) [01]

En esta instrucción se revisa el valor de la línea CC, si es igual a uno, la dirección del estado siguiente la proporciona el registro μPC; si es igual a cero, la dirección del estado siguiente, contenida en el registro seleccionado por PL, ingresa a través de la entrada D.

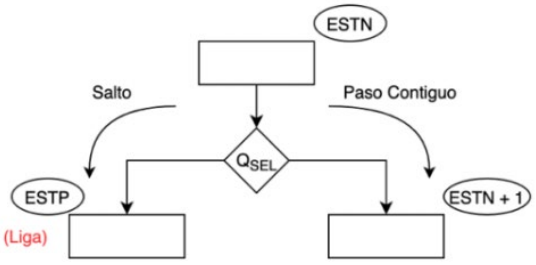

Figura 4 Carta ASM de la microinstrucción Salto Condicional.

Salto de Transformación (ST) [10]

La dirección del estado siguiente se obtiene del registro seleccionado por la línea de MAP. Este registro también está conectado a la entrada D. Aquí se introduce una nueva notación de carta ASM: un rombo con varias bifurcaciones. La bifurcación que se elija dependerá del contenido del registro seleccionado por MAP.

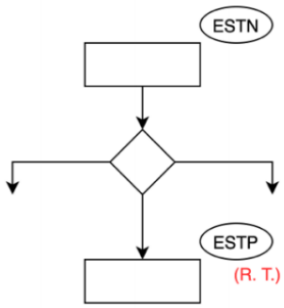

Figura 5 Carta ASM de la microinstrucción Salto de Transformación.

Salto de Interrupción (SCI) [11]

En esta instrucción se revisa el valor de CC, si es igual a uno, la dirección del estado siguiente proviene del registro μPC; si es igual a cero, la dirección del estado siguiente, contenida en el registro seleccionado por VECT, ingresa a través de la entrada D.

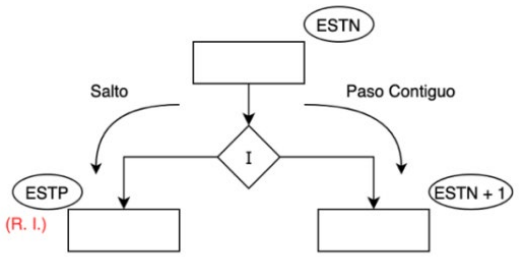

Figura 6 Carta ASM de la microinstrucción Salto de Interrupción

La lógica interna del secuenciador se construye a partir de la siguiente tabla poniendo énfasis en Y la cual nos indica de que bloque se tomara el estado siguiente.

Entradas			**Salidas**				
I1	I0	$\overline{CC}$	Selector	$\overline{PL}$	$\overline{MAP}$	$\overline{VECT}$	Y
0	0	0	0	1	1	1	μPC
0	0	1	0	1	1	1	μPC
0	1	0	1	0	1	1	Entrada D
0	1	1	0	0	1	1	μPC
1	0	0	1	1	0	1	Entrada D
1	0	1	1	1	0	1	Entrada D
1	1	0	1	1	1	0	Entrada D
1	1	1	0	1	1	0	μPC

Desarrollo

Como primer punto, analizamos la carta ASM que se muestra a continuación, otorgando los valores binarios para cada uno de los estados.

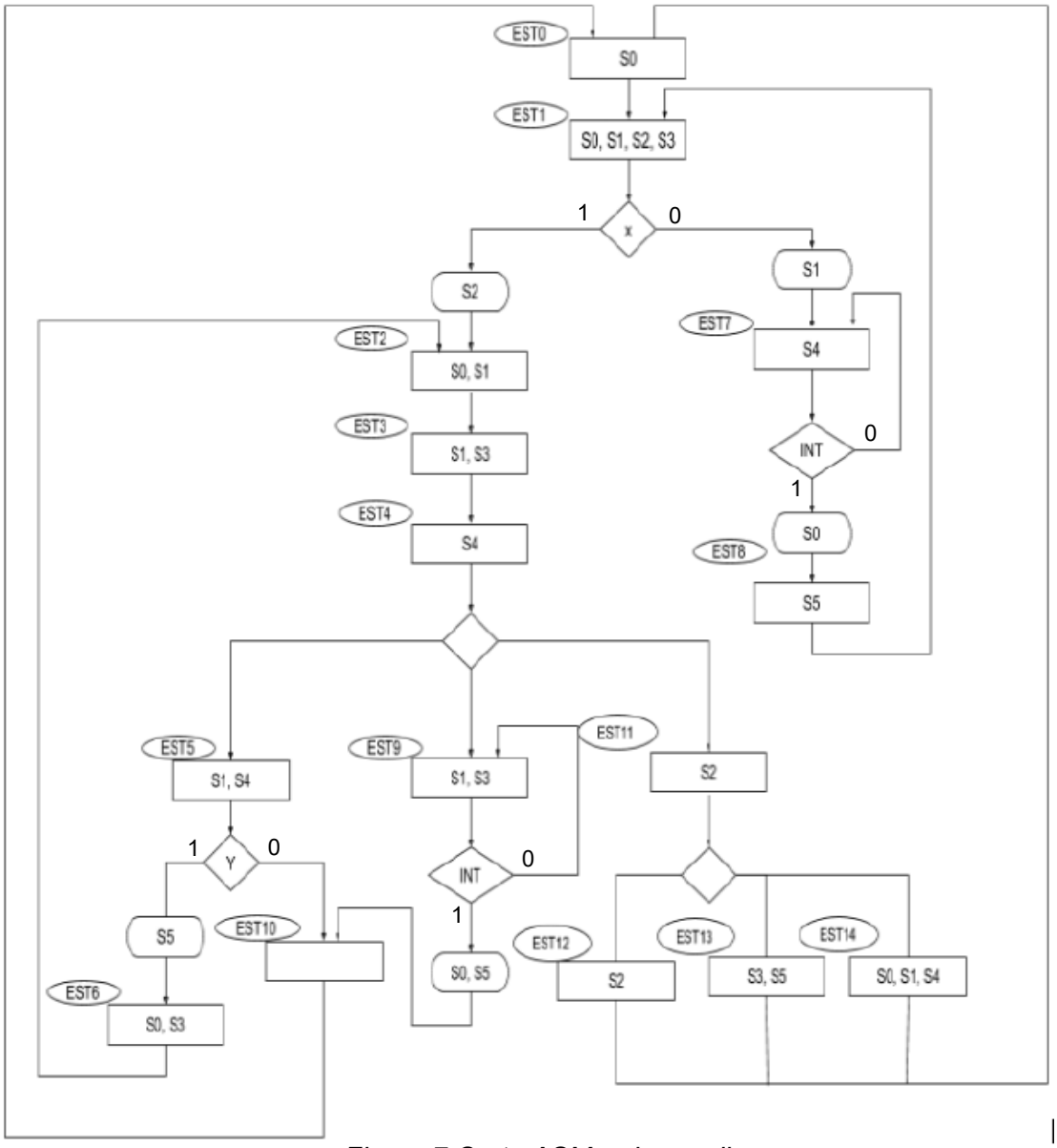

Figura 7 Carta ASM a desarrollar

Analizando primeramente la carta ASM, asegurándonos de que cumple con la regla “N, N+1, P”, realizamos la tabla de verdad correspondiente al comportamiento que esta tiene. Asignando inicialmente los valores binarios para cada uno de los estados y de los valores de prueba para identificarlos de manera correcta

Estados	
EST0	0000
EST1	0001
EST2	0010
EST3	0011
EST4	0100
EST5	0101
EST6	0110

Estados	
EST7	0111
EST8	1000
EST9	1001
EST10	1010
EST11	1011
EST12	1100
EST13	1101
EST14	1110

Prueba	
X	00
Y	01
INT	10
AUX	11

MI	
PC	00
SC	01
ST	10
SI	11

PC – Paso Contiguo
SC – Salto Condicional
ST – Salto de Transformación
SI – Salto de Interrupción

Figura 7.1 Valores binarios de los estados y las entradas.

Con lo cual obtuvimos la siguiente tabla de verdad

Direccion				Contenido de la Memoria																				
Edo. Presente				Prueba		VF	MI		Liga				Salidas Incremento						Salidas Carga					
P3	P2	P1	P0	K1	K0		I1	I0	L3	L2	L1	L0	S5	S4	S3	S2	S1	S0	S5	S4	S3	S2	S1	S0
0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1
0	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1
0	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	0	1	0
0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	1	0	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	0	1	0
0	1	1	0	1	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1
0	1	1	1	1	0	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1	0	0	0	0
1	0	0	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	1	0	1	0
1	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
1	1	0	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0
1	1	1	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1
1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 7.2 Tabla de verdad de la carta ASM.

Con los valores obtenidos, implementamos la memoria en Quartus con la programación VHDL obteniendo el siguiente código:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY rom IS
   PORT (
       addr : IN std_logic_vector(3 DOWNTO 0);
       prueba : OUT std_logic_vector(1 DOWNTO 0);
       vf : OUT std_logic;
       mi : OUT std_logic_vector(1 DOWNTO 0);
       liga : OUT std_logic_vector(3 DOWNTO 0);
       salida_incremento : OUT std_logic_vector(5 DOWNTO 0);
       salida_carga : OUT std_logic_vector(5 DOWNTO 0)

   );
END rom;

ARCHITECTURE behavioral OF rom IS
   TYPE mem_rom IS ARRAY(0 TO 14) OF std_logic_vector(20 DOWNTO 0);
   SIGNAL data_out : mem_rom;
   SIGNAL data : std_logic_vector(20 DOWNTO 0);
BEGIN
   data_out(0) <= "11000000100000100001";
   data_out(1) <= "00001011100111001111";
   data_out(2) <= "11000001100001100011";
   data_out(3) <= "11000010000101000101O";
   data_out(4) <= "11010000001000001000";
   data_out(5) <= "01001101011001001001O";
   data_out(6) <= "11001001000100100100";
   data_out(7) <= "10011011101000101000";
   data_out(8) <= "11001000110000010000";
   data_out(9) <= "10011100110101100101O";
```

```
   data_out(10) <= "11001000000000000000";
   data_out(11) <= "11010000000100000100";
   data_out(12) <= "11001000000100000100";
   data_out(13) <= "11001000010100010100";
   data_out(14) <= "11001000010011010011";
   PROCESS (addr) BEGIN
       data <= data_out(conv_integer(unsigned(addr)));
       prueba <= data(20 DOWNTO 19);
       vf <= data(18);
       mi <= data(17 DOWNTO 16);
       liga <= data(15 DOWNTO 12);
       salida_incremento <= data(11 DOWNTO 6);
       salida_carga <= data(5 DOWNTO 0);
   END PROCESS;
END behavioral;
```

Figura 7.3 Código rom.vhd

Posteriormente implementamos la lógica del secuenciador como se describió en la introducción, teniendo como salidas a MAP, PL, VECT y el selector que nos sirve para saber si se esta haciendo un paso contiguo o un salto.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY secuenciador_logica IS
   PORT (
       mi : IN std_logic_vector(1 DOWNTO 0);
       ccn : IN std_logic;
       mapn : OUT std_logic;
       pln : OUT std_logic;
       selector: OUT std_logic;
       vectn : OUT std_logic
   );
END secuenciador_logica;

ARCHITECTURE behavioral OF secuenciador_logica IS
    BEGIN
   PROCESS (mi, ccn) BEGIN
           mapn <= '0';
           pln <= '0';
           vectn <= '0';
           CASE mi IS
               WHEN "00" =>
                   selector <= '0';
               WHEN "01" =>
                   IF ccn = '1' THEN
                       selector <= '0';
                   ELSE
                       selector <= '1';
                       pln <= '1';
                   END IF;
               WHEN "10" =>
                   selector <= '0';
```

```
                mapn <= '1';
            WHEN OTHERS =>
                IF ccn = '1' THEN
                    selector <= '0';
                ELSE
                    selector <= '1';
                    vectn <= '1';
                END IF;
        END CASE;
   END PROCESS;
END behavioral;
```

Figura 7.4 Código secuenciador_logica.vhd

Después, implementamos el selector de entrada con el cual conoceremos el valor de prueba que se esta utilizando conforme a lo descrito en la tabla de verdad.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY selector_entrada IS PORT (
   prueba : IN std_logic_vector (1 DOWNTO 0);
   X : IN std_logic;
   Y : IN std_logic;
   INT : IN std_logic;
   AUX : IN std_logic;
   valor_entrada : OUT std_logic);
END selector_entrada;
ARCHITECTURE Behavioral OF selector_entrada IS
BEGIN
   PROCESS (prueba) BEGIN
      CASE(prueba) IS
          WHEN "00" =>
          valor_entrada <= X;
          WHEN "01" =>
          valor_entrada <= Y;
          WHEN "10" =>
          valor_entrada <= INT;
          WHEN OTHERS =>
          valor_entrada <= AUX;
      END CASE;
   END PROCESS;
END Behavioral;
```

Figura 7.5 Código selector_entrada.vhd

Así mismo creamos un selector de salida y un selector de liga para poder reconocer si se está obteniendo la salida de incremento o la salida de carga.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY selector_salida IS PORT (
   ccn : IN std_logic;
   salida_incremento : IN std_logic_vector (5 DOWNTO 0);
   salida_carga : IN std_logic_vector (5 DOWNTO 0);
   salida : OUT std_logic_vector (5 DOWNTO 0));
END selector_salida;
ARCHITECTURE Behavioral OF selector_salida IS
BEGIN
   PROCESS (salida_carga, salida_incremento) BEGIN
       IF ccn = '1' THEN
           salida <= salida_carga;
       ELSE
           salida <= salida_incremento;
       END IF;
   END PROCESS;
END Behavioral;
```

Figura 7.6 Código selector_salida.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY selector_liga IS PORT (
   selector : IN std_logic;
   liga_incremento : IN std_logic_vector(3 DOWNTO 0);
   liga_carga : IN std_logic_vector(3 DOWNTO 0);
   liga_presente : OUT std_logic_vector(3 DOWNTO 0)
);
END selector_liga;
ARCHITECTURE Behavioral OF selector_liga IS
BEGIN
   PROCESS (selector) BEGIN
       IF selector = '1' THEN
           liga_presente <= liga_carga;
       ELSE
           liga_presente <= liga_incremento;
       END IF;
   END PROCESS;
END Behavioral;
```

Figura 7.7 Código selector_liga.vhd

Y como se ha mencionado a lo largo de la introducción, realizamos el incrementador, que nos ayudara a pasar a un estado contiguo.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY incremento IS PORT (
   liga_presente : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   liga_incremento : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
);
END incremento;
ARCHITECTURE Behavioral OF incremento IS
BEGIN
   PROCESS(liga_presente) BEGIN
       liga_incremento <= liga_presente + 1;
   END PROCESS;
END Behavioral;
```

Figura 7.8 Código incremento.vhd

Por último, implementamos una serie de registros de diferentes tamaños que nos sirve para poder manejar las salidas del secuenciador, uno para PL, MAP y VECT (registro_tri.vhd); otro para MI (registro_2.vhd); para CC (registro_1.vhd) y finalmente para poder manipular la liga de incremento (registro_4.vhd).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY registro_tri IS
   PORT (
       entrada : IN std_logic_vector(3 DOWNTO 0);
       activado : IN std_logic;
         reset: IN std_logic;
       clk : IN std_logic;
         salida : OUT std_logic_vector(3 DOWNTO 0)
   );
END registro_tri;

ARCHITECTURE behavioral OF registro_tri IS
BEGIN
   PROCESS (clk, entrada, activado) BEGIN
       IF rising_edge(clk) THEN
           IF activado = '0' or reset = '1' THEN
               salida <= "ZZZZ";
           ELSE
               salida <= entrada;
           END IF;
       END IF;
   END PROCESS;
END behavioral;
```

Figura 7.9 Código registro_tri.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY registro_1 IS
   PORT (
       entrada : IN std_logic;
       reset : IN std_logic;
       clk : IN std_logic;
       salida : OUT std_logic
   );
END registro_1;

ARCHITECTURE behavioral OF registro_1 IS
SIGNAL reg : std_logic;
BEGIN
   PROCESS (clk, entrada) BEGIN
       IF rising_edge(clk) THEN
           IF reset = '1' THEN
               reg <= 'Z';
           ELSE
               reg <= entrada;
           END IF;
       END IF;
       salida <= reg;
   END PROCESS;
END behavioral;
```

Figura 7.10 Código registro_1.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY registro_2 IS
   PORT (
       entrada : IN std_logic_vector(1 downto 0);
       reset : IN std_logic;
       clk : IN std_logic;
       salida : OUT std_logic_vector(1 downto 0)
   );
END registro_2;

ARCHITECTURE behavioral OF registro_2 IS
SIGNAL reg : std_logic_vector(1 downto 0);
BEGIN
   PROCESS (clk, entrada) BEGIN
       IF rising_edge(clk) THEN
           IF reset = '1' THEN
               reg <= "ZZ";
           ELSE
               reg <= entrada;
           END IF;
       END IF;
       salida <= reg;
   END PROCESS;
END behavioral;
```

Figura 7.11 Código registro_2.vhd

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY registro_4 IS
    PORT (
        entrada : IN std_logic_vector(3 downto 0);
        reset : IN std_logic;
        clk : IN std_logic;
        salida : OUT std_logic_vector(3 downto 0)
    );
END registro_4;

ARCHITECTURE behavioral OF registro_4 IS
SIGNAL reg : std_logic_vector(3 downto 0);
BEGIN
    PROCESS (clk, entrada) BEGIN
        IF rising_edge(clk) THEN
            IF reset = '1' THEN
                reg <= "ZZZZ";
            ELSE
                reg <= entrada;
            END IF;
        END IF;
        salida <= reg;
    END PROCESS;
END behavioral;
```

Figura 7.11 Código registro_4.vhd

Una vez realizado cada uno de los componentes necesarios para llevar a cabo cada una de las partes para el secuenciador, obtuvimos cada uno de sus símbolos para poder unirlos en un esquema, obteniendo el siguiente resultado:

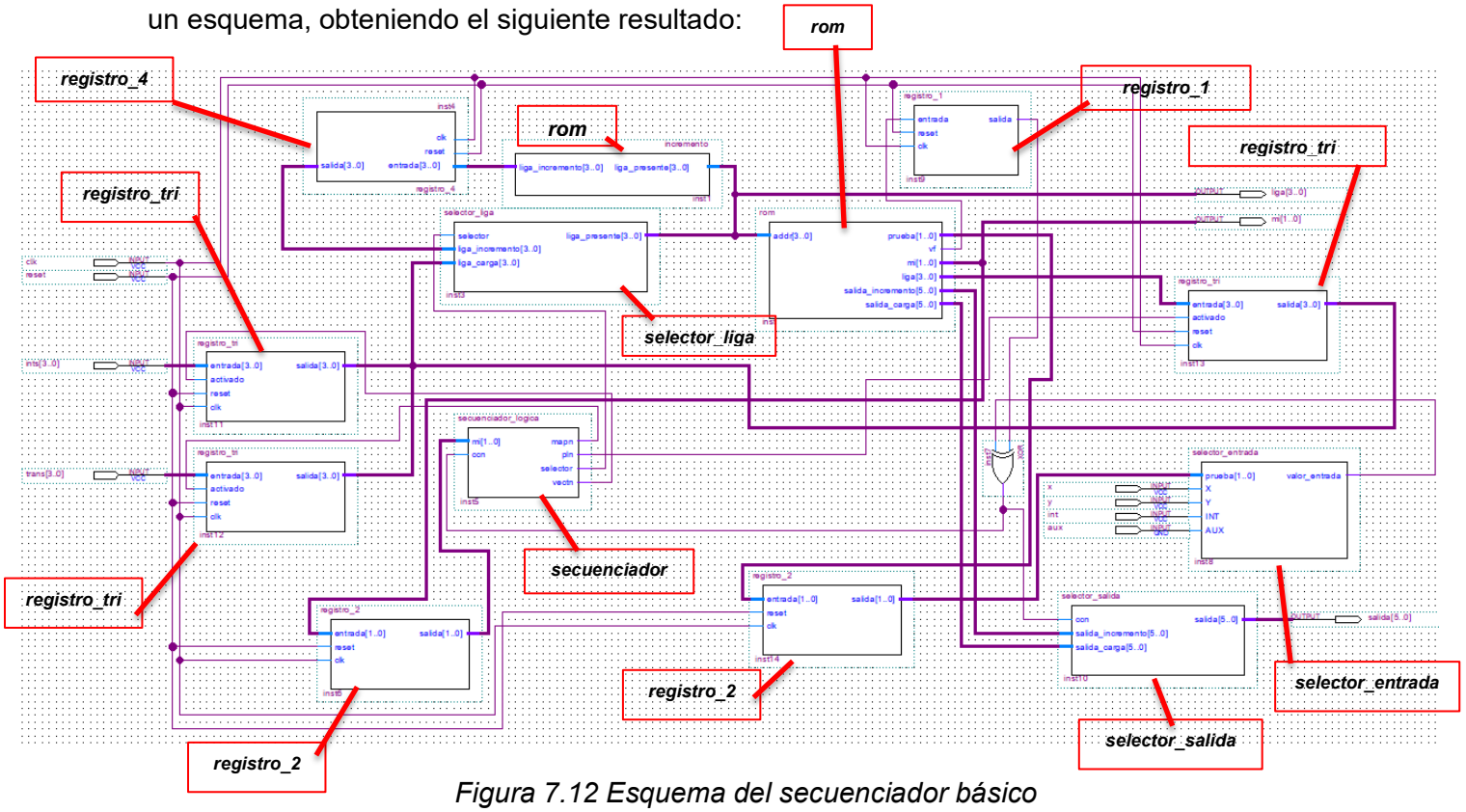

Figura 7.12 Esquema del secuenciador básico

Como podemos observar se sigue prácticamente el diagrama mostrado en la introducción que pertenece al direccionamiento implícito. Ahora bien, una vez compilado el bloque entero y asignado cada una de las entradas, así como la señal de reloj y de Reset, procedemos a simularlo

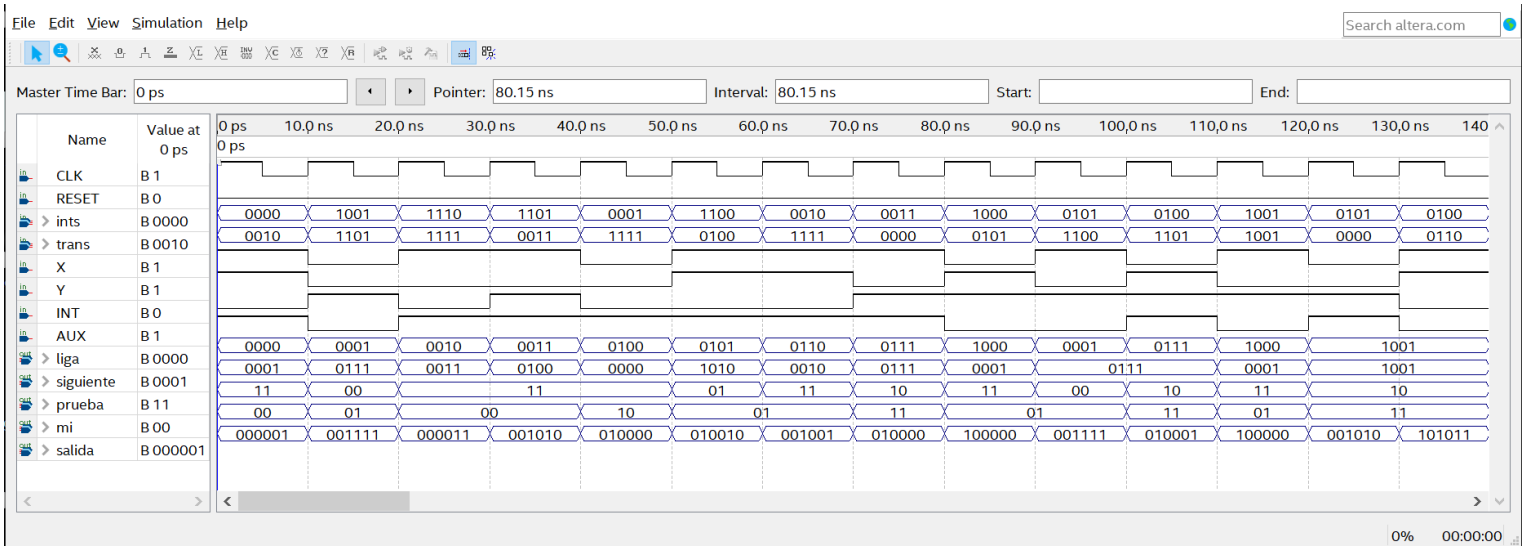

Figura 7.13 Simulación del Esquemático

Como podemos ver en la simulación se observan las señales de reloj (*clk*), reset, entradas (*X, Y, INT, AUX*), ints y trans (que nos ayudaron a activar los registros), estado actual (*liga*), estado siguiente (*siguiente*), el valor de prueba (*prueba*), el valor de la microinstrucción utilizada por el secuenciador (*mi)* y finalmente las salidas (*salida*).

Siguiendo la carta ASM y la tabla de verdad junto con los valores de las entradas y de las microinstrucciones, podemos comprobar que efectivamente se lleva a cabo de manera correcta el funcionamiento secuenciador básico. Arrojando así todos los valores de las salidas correspondientes al estado que se está analizando. Por lo que la simulación es correcta ya que nos indica de buena manera el estado siguiente al que se debe de ir y la salida que se obtuvo con esta.

Fuentes de Consulta

- Savage, J (2015) Diseño de microprocesadores. Facultad de Ingeniería. Universidad Nacional Autónoma de México. 482pp,
- Chávez, N (S.F) Construcción de máquinas de estados usando memorias. Consultado el 23 de noviembre del 2020 de: http://profesores.fi-b.unam.mx/normaelva/Direccionamientos.pdf