	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio

Profesor: ING. JULIO CESAR CRUZ ESTRADA

Asignatura: LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

Grupo: 07

No de Práctica: 3

Integrante(s): Jiménez Treviño Emilio Cristóbal

Martínez Pérez Brian Erik

No. de Equipo de cómputo empleado: *s/n*

Semestre: *2026-1*

Fecha de entrega: *19/09/2025*

Observaciones:

CALIFICACIÓN: _____

Práctica 3. Construcción de Máquinas de estados Usando Memorias Direccionamiento por Trayectoria

Objetivo

Familiarizar al alumno en el conocimiento de construcción de máquinas de estados usando direccionamiento de memorias con el método de direccionamiento por trayectoria.

Introducción

En esta práctica se abordará la construcción de máquinas de estados usando memorias direccionamiento por trayectoria. Este enfoque permite organizar el diseño en etapas bien definidas: la elaboración de la carta ASM, la obtención de la tabla de verdad y la codificación de los distintos bloques en VHDL, como el registro, concatenador, memoria ROM, divisor de datos y divisor de frecuencia. Se detallarán los pasos para la creación de un proyecto en Quartus y la asignación de pines para su posterior simulación y carga en un dispositivo FPGA que en nuestro caso fue la DE10-LITE debido a su capacidad y accesibilidad

Desarrollo

Para comenzar primero se propuso una carta ASM con 5 estados, 3 entradas y 4 salidas. las restricciones eran, tener al menos un estado con más de una condición, al menos dos salidas en lógica negada, al menos un estado con salidas condicionales.

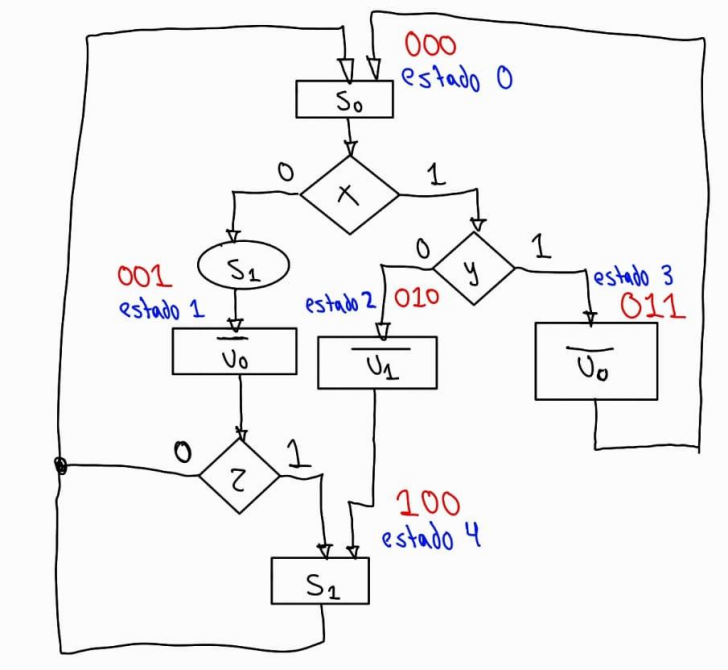


Figura 1. Carta ASM propuesta.

Después de obtener la carta ASM, obtenemos la tabla de verdad, donde debemos de tomar en cuenta que utilizamos el método de direccionamiento por trayectoria. debemos tener en la cabecera el estado presente, las entradas, estado siguiente y las salidas.

	edop	x	y	z	edos	s1	s0	~u1	~u0
EST0	000	0	*	*	001	1	1	1	1
	000	1	0	*	010	0	1	1	1
	000	1	1	*	011	0	1	1	1
EST1	001	*	*	0	000	0	0	1	0
	001	*	*	1	100	0	0	1	0
EST2	010	*	*	*	100	1	0	1	1
EST3	011	*	*	*	000	0	0	1	0
EST4	100	*	*	*	000	1	0	1	1

Figura 2. Tabla de verdad, obtenida por el método de direccionamiento por trayectoria.

Cuando ya tenemos la tabla de verdad, podemos implementar el direccionamiento por trayectoria utilizando el software de desarrollo Quartus y escribir el contenido de memoria obtenido.

Lo primero que debemos hacer es crear un proyecto en Quartus, seleccionamos la pestaña FILE -> New Project Wizard. En el asistente de creación de nuevos proyectos, en la Fig. 3 podemos observar seleccionado la familia del dispositivo en nuestro caso la MAX 10 y dentro de esta vendría a hacer la 10M50DAF484C7G dentro de esta familia.

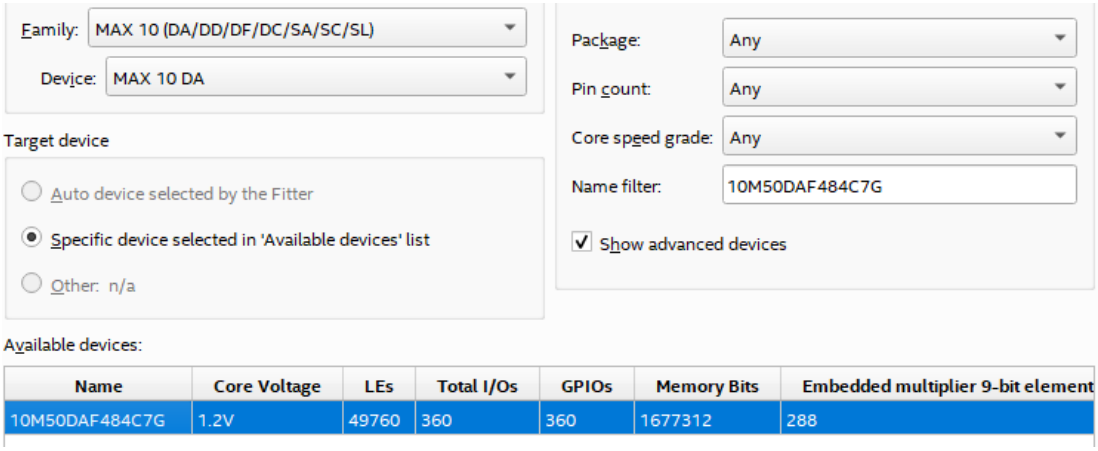


Figura 3. Device

Una vez creado nuestro proyecto, creamos un archivo de tipo .bdf dando click en la opción de Block Diagram/Schematic File como se muestra en la Fig. 4.

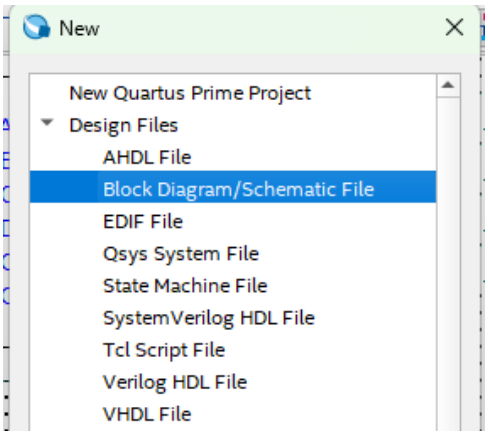


Figura 4. New File

Para implementar el circuito de direccionamiento por trayectoria, debemos tener en cuenta que tenemos un bloque llamado “registro”, el cual tiene como entradas la liga “estado siguiente”, el reloj, y las entradas de nuestra carta ASM. y otro bloque donde contenemos la memoria con las combinaciones de la tabla de verdad, la liga y las salidas de cada estado.

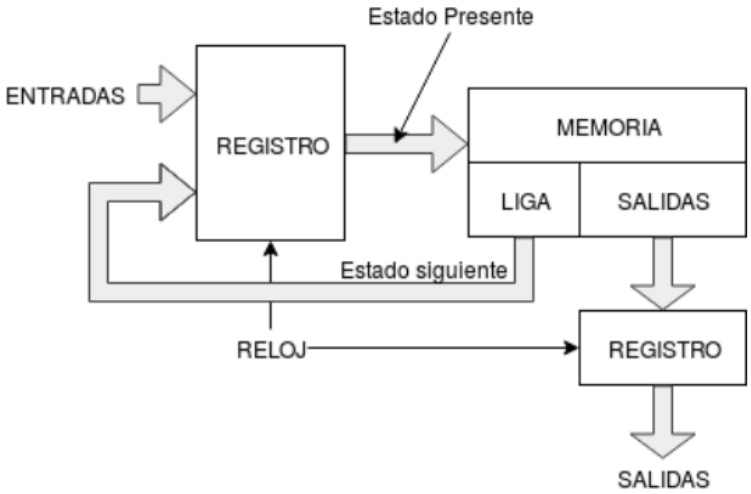


Figura 5. Diagrama circuito direccionamiento por trayectoria.

El primer código generado fue “registró”, Su función es almacenar el valor de una señal de entrada (edos) y mantenerlo en una señal de salida (edop). El registro se actualiza en cada flanco de subida de la señal de reloj (reloj), a menos que la señal de reinicio (reset) esté activa, en cuyo caso se inicializa a "000".

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity registro is
7     Port ( reloj : in std_logic;
8            reset : in std_logic;
9            edos : in std_logic_vector (2 downto 0);
10           edop : out std_logic_vector (2 downto 0)
11    );
12 end registro;
13
15 architecture Behavioral of registro is
16
17 signal internal_value: std_logic_vector (2 downto 0) := B"000";
18 begin
19    process(reloj, reset, edos)
20    begin
21
22       if reset = '0' then
23          internal_value <= B"000";
24       elsif rising_edge(reloj) then
25          internal_value <= edos;
26       end if;
27    end process;
28
29    process(internal_value)
30    begin
31       edop <= internal_value;
32    end process;
33
34
35 end Behavioral;
```

Figura 6. Código VHDL, para generar el bloque “registro”.

Figura 7. Bloque “registro”.

El segundo código fue “concatenador”, Su única función es unir dos vectores de bits de entrada (edop y entradas) para formar un único vector de salida (salida). En este caso, une dos vectores de 3 bits cada uno para crear una salida de 6 bits.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_ARITH.ALL;
4   use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6   entity concatenador is
7       Port ( entradas : in std_logic_vector (2 downto 0);
8              edop : in std_logic_vector (2 downto 0);
9              salida : out std_logic_vector (5 downto 0)
10      );
11  end concatenador;
12
13  architecture Behavioral of concatenador is
14  begin
15
16      process(edop, entradas)
17      begin
18
19          salida <= edop & entradas;
20
21
22      end process;
23
24  end Behavioral;
```

Figura 8. Código VHDL, para generar el bloque “concatenador”.

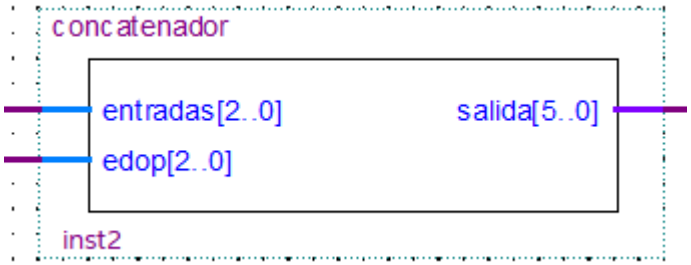


Figura 9. Bloque “concatenador”.

El tercer código fue “rom”, Almacena una serie de valores predefinidos en un arreglo interno (internal_mem). Cuando se le proporciona una dirección de 6 bits (direccion), el circuito devuelve el valor de 7 bits almacenado en esa posición de la memoria, actuando como una tabla de consulta.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rom is
    Port ( direccion : in std_logic_vector (5 downto 0);
           data : out std_logic_vector (6 downto 0)
    );
end rom;

architecture Behavioral of rom is

type mem is array (0 to 63) of std_logic_vector (6 downto 0);
signal internal_mem: mem;
    --estado 6
    internal_mem(48) <= "000" & "0011";
    internal_mem(49) <= "000" & "0011";
    internal_mem(50) <= "000" & "0011";
    internal_mem(51) <= "000" & "0011";
    internal_mem(52) <= "000" & "0011";
    internal_mem(53) <= "000" & "0011";
    internal_mem(54) <= "000" & "0011";
    internal_mem(55) <= "000" & "0011";

    --estado 7
    internal_mem(56) <= "000" & "0011";
    internal_mem(57) <= "000" & "0011";
    internal_mem(58) <= "000" & "0011";
    internal_mem(59) <= "000" & "0011";
    internal_mem(60) <= "000" & "0011";
    internal_mem(61) <= "000" & "0011";
    internal_mem(62) <= "000" & "0011";
    internal_mem(63) <= "000" & "0011";

    process(direccion)
    begin
      data <= internal_mem(conv_integer(unsigned(direccion)));
   end process;
end Behavioral;
```

```
begin
    --Estado 0
    internal_mem(0) <= "0011111";--
    internal_mem(1) <= "0011111";--
    internal_mem(2) <= "0011111";--
    internal_mem(3) <= "0011111";--

    internal_mem(4) <= "0100111";--
    internal_mem(5) <= "0100111";--

    internal_mem(6) <= "0110111";--
    internal_mem(7) <= "0110111";--

    --estado 1
    internal_mem(8) <= "0000010";--
    internal_mem(9) <= "0000010";--
    internal_mem(10) <= "0000010";-
    internal_mem(11) <= "0000010";

    internal_mem(12) <= "1000010";
    internal_mem(13) <= "1000010";
    internal_mem(14) <= "1000010";
    internal_mem(15) <= "1000010";

    --estado 2
    internal_mem(16) <= "1001011";
    internal_mem(17) <= "1001011";
    internal_mem(18) <= "1001011";
    internal_mem(19) <= "1001011";
    internal_mem(20) <= "1001011";
    internal_mem(21) <= "1001011";
    internal_mem(22) <= "1001011";
    internal_mem(23) <= "1001011";

    --estado 3
    internal_mem(24) <= "0000010";
    internal_mem(25) <= "0000010";
    internal_mem(26) <= "0000010";
    internal_mem(27) <= "0000010";
    internal_mem(28) <= "0000010";
    internal_mem(29) <= "0000010";
    internal_mem(30) <= "0000010";
    internal_mem(31) <= "0000010";
```

```
    --estado 4
    internal_mem(32) <= "0001011";
    internal_mem(33) <= "0001011";
    internal_mem(34) <= "0001011";
    internal_mem(35) <= "0001011";
    internal_mem(36) <= "0001011";
    internal_mem(37) <= "0001011";
    internal_mem(38) <= "0001011";
    internal_mem(39) <= "0001011";

    --estado 5
    internal_mem(40) <= "000" & "0011";
    internal_mem(41) <= "000" & "0011";
    internal_mem(42) <= "000" & "0011";
    internal_mem(43) <= "000" & "0011";
    internal_mem(44) <= "000" & "0011";
    internal_mem(45) <= "000" & "0011";
    internal_mem(46) <= "000" & "0011";
    internal_mem(47) <= "000" & "0011";
```

Figura 10. Código VHDL, para generar el bloque “rom”.

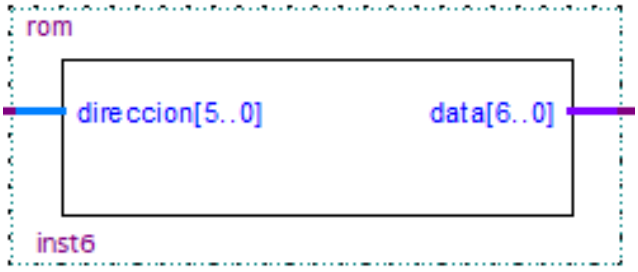


Figura 11. Bloque “rom”.

El cuarto código fue “div_datos”, Toma un vector de bits de entrada (data) de 7 bits y lo divide en dos vectores de salida más pequeños. El primer vector de salida (liga) toma los 3 bits más significativos (del 6 al 4) y el segundo vector (salidas) toma los 4 bits menos significativos (del 3 al 0).

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity div_datos is
    Port ( data : in std_logic_vector (6 downto 0);
           liga : out std_logic_vector (2 downto 0);
           salidas : out std_logic_vector (3 downto 0)
    );
end div_datos;

architecture Behavioral of div_datos is
begin

    process(data)
    begin

        liga <= data(6 downto 4);
        salidas <= data (3 downto 0);

    end process;

end Behavioral;
```

Figura 12. Código VHDL, para generar el bloque “div_datos”.

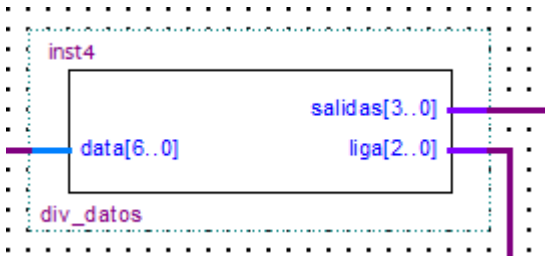


Figura 13. Bloque “div_datos”.

Por último creamos el código de “div_frec”, toma una señal de reloj de entrada (reloj) y genera una señal de salida (div_clk) con una frecuencia mucho menor. Esto nos ayuda a observar ,los efectos de los leds en la salida.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity div_frec is
    Port ( reloj : in std_logic;
    div_clk : out std_logic);
end div_frec;

architecture Behavioral of div_frec is
begin
    process (reloj)
    variable cuenta: std_logic_vector (27 downto 0):=x"0000000";
    begin

        if rising_edge (reloj) then
            if cuenta = x"17D7840" then --25M pulsos
                cuenta:= x"0000000";
            else
                cuenta:= cuenta+1;
            end if;
        end if;
        div_clk <= cuenta(24);
    end process;
end Behavioral;
```

Figura 14. Código VHDL, para generar el bloque “div_frec”.

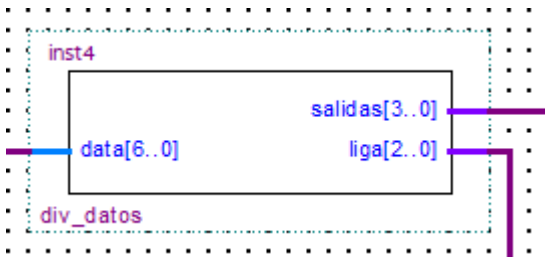


Figura 15. Bloque “div_frec”.

En el archivo esquemático, tenemos todos los bloques generados a partir del código vhdl, en este tenemos 5 entradas, “Reset”, “entrada[0]”, “entrada[1]”, “entrada[2]” y “CLK”. Además tenemos 5 salidas que serán asignadas a los leds de la tarjeta. “salida[0]”, “salida[1]”, “salida[2]”, “salida[3]”, “edop[0]”, “edop[1]”, “edop[2]”.

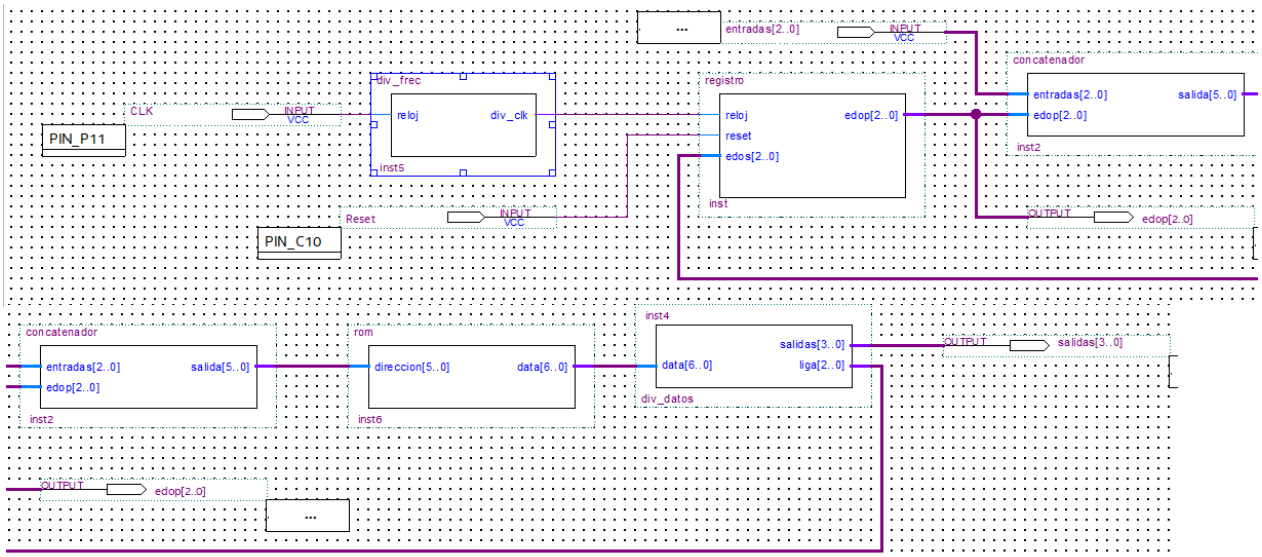


Figura 16. Diagrama de conexiones

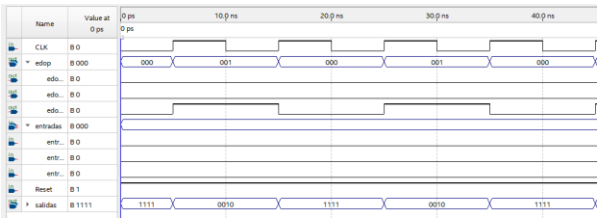
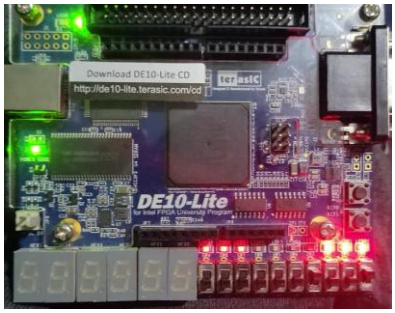
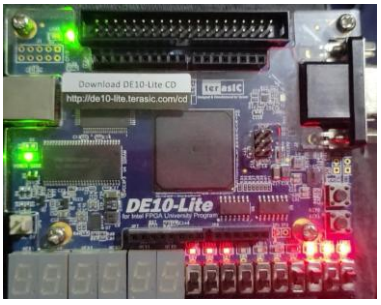
Procedemos con nuestro asignación de pines, para esto ocuparemos nuestro data sheet de nuestra FPGA DE-10 Lite, para hacer esto daremos click al icono que dice pin planner o presionando "ctrl+shift+n" los pines se colocan en el apartado de Location en la Fig. 17 podremos ver cuales se asignaron para este caso, recordemos que deben asignarse desde el más significativo al menos, para una correcta secuencia.

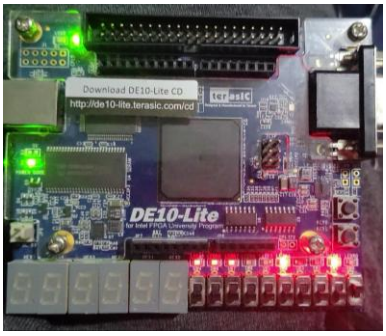
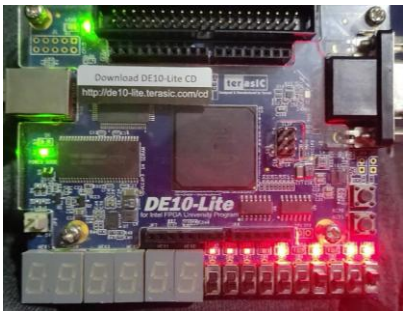
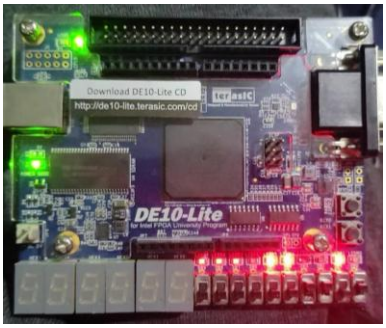
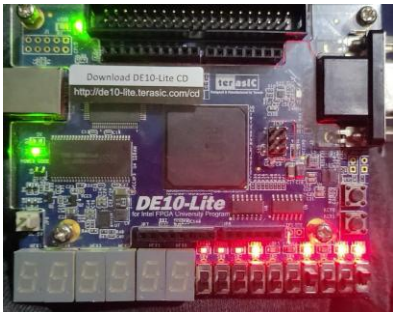
Node Name	Direction	Location
CLK	Input	PIN_P11
edop[2]	Output	PIN_E14
edop[1]	Output	PIN_C13
edop[0]	Output	PIN_D13
entradas[2]	Input	PIN_C12
entradas[1]	Input	PIN_D12
entradas[0]	Input	PIN_C11
Reset	Input	PIN_C10
salidas[3]	Output	PIN_B10
salidas[2]	Output	PIN_A10
salidas[1]	Output	PIN_A9
salidas[0]	Output	PIN_A8

Figura 17. Asignación de pines

Resultados

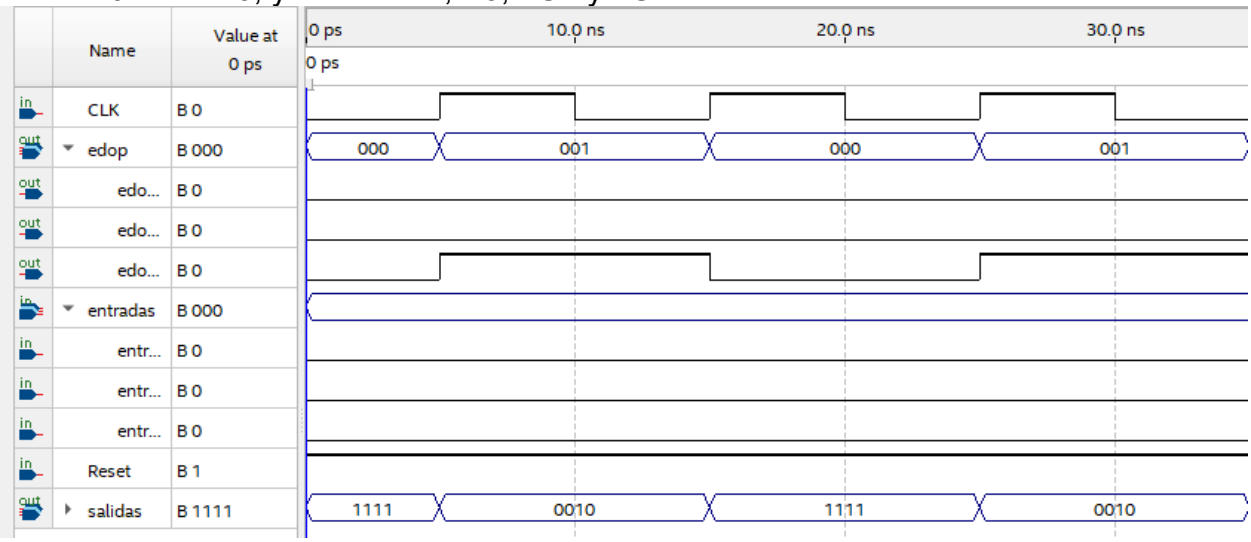
	edop	x	y	z	edos	s1	s0	~u1	~u0
EST0	000	0	*	*	001	1	1	1	1
	000	1	0	*	010	0	1	1	1
	000	1	1	*	011	0	1	1	1
EST1	001	*	*	0	000	0	0	1	0
	001	*	*	1	100	0	0	1	0
EST2	010	*	*	*	100	1	0	1	1
EST3	011	*	*	*	000	0	0	1	0
EST4	100	*	*	*	000	1	0	1	1

Estado 0 con X=0, y salidas S1, S0, ¬U1 y ¬U2.	
Estado 0 con X=1 Y=0, y salidas S0, ¬U1 y ¬U2.	
Estado 0 con X=1 Y=1, y salidas S0, ¬U1 y ¬U2.	
Estado 1 Z=0, y salidas ¬U1	

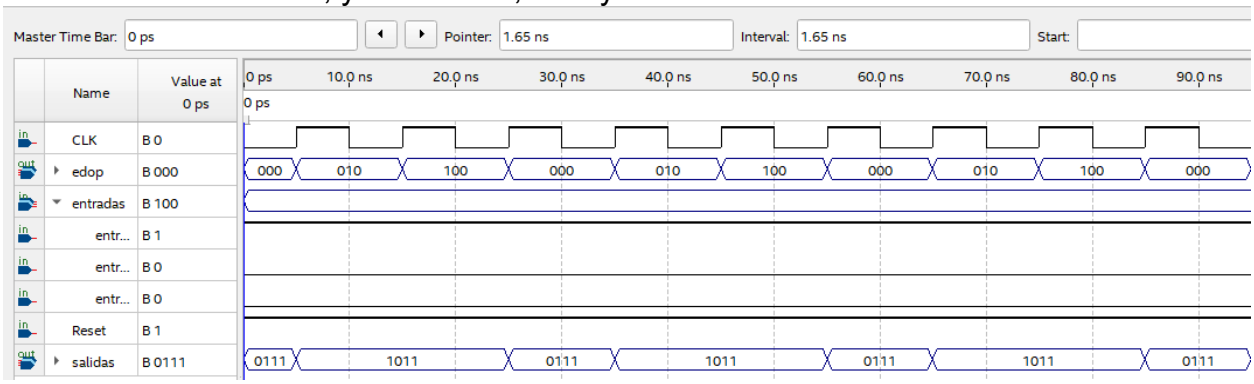
Estado 1 Z=1, y salidas ¬U1	
Estado 2, salidas S0, ¬U1 y ¬U2.	
Estado 3, salidas ¬U1	
Estado 4. salidas S0, ¬U1 y ¬U2.	

Simulación

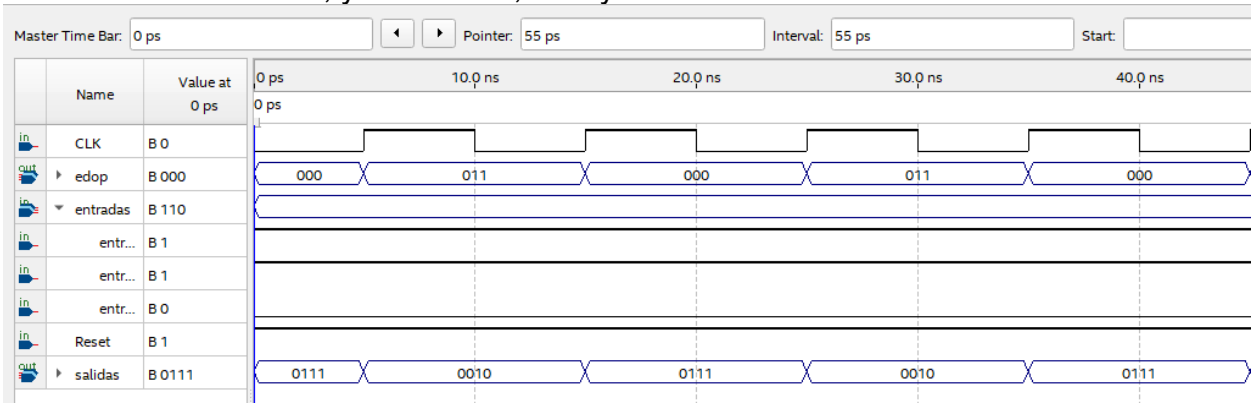
Estado 0 con X=0, y salidas S1, S0, ¬U1 y ¬U2.



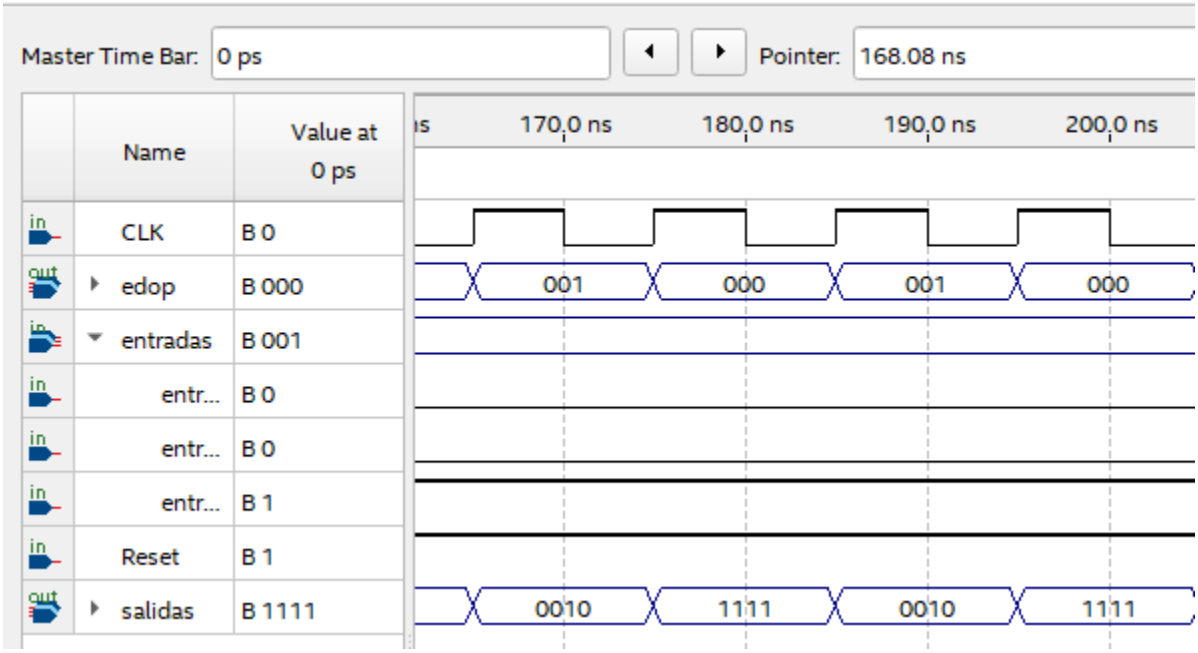
Estado 0 con X=1 Y=0, y salidas S0, ¬U1 y ¬U2.



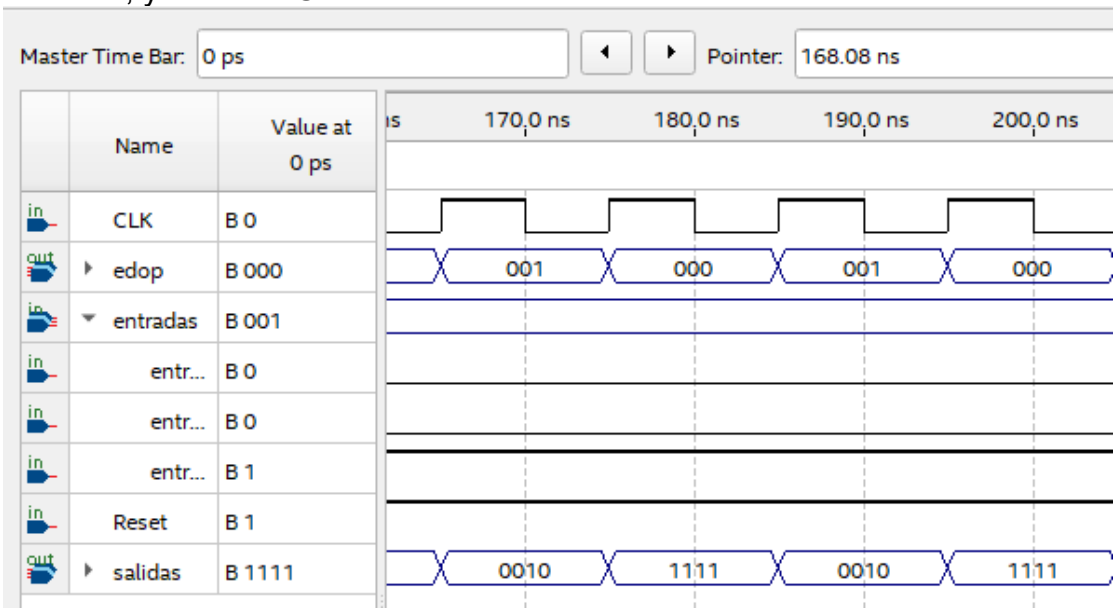
Estado 0 con X=1 Y=1, y salidas S0, ¬U1 y ¬U2.



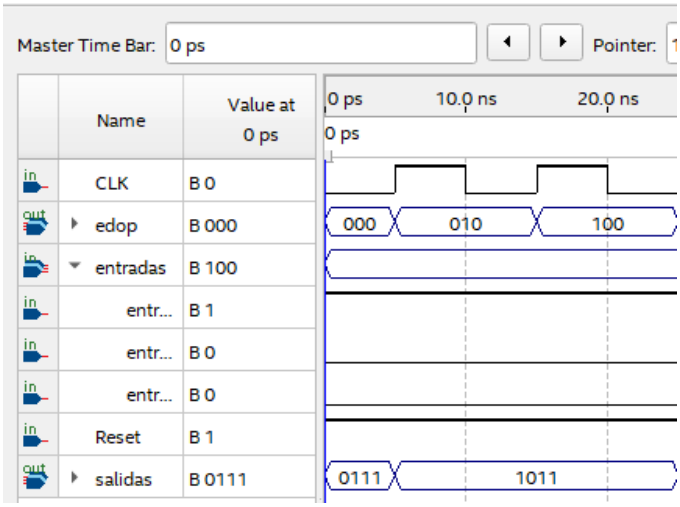
Estado 1 Z=0, y salidas ¬U1.



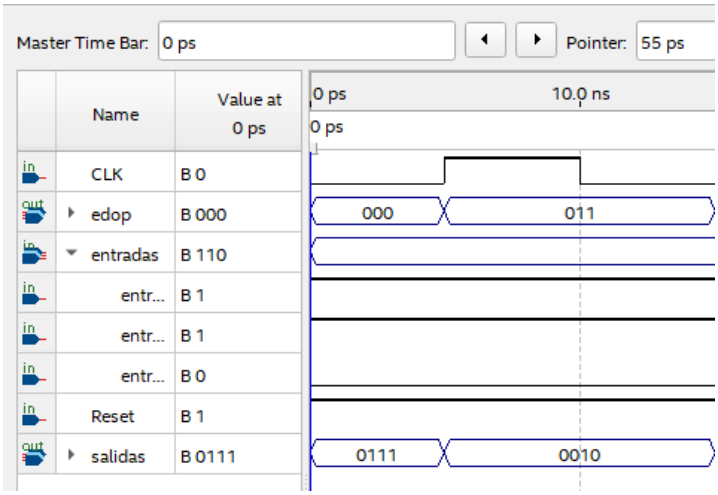
Estado 1 Z=1, y salidas ¬U1.



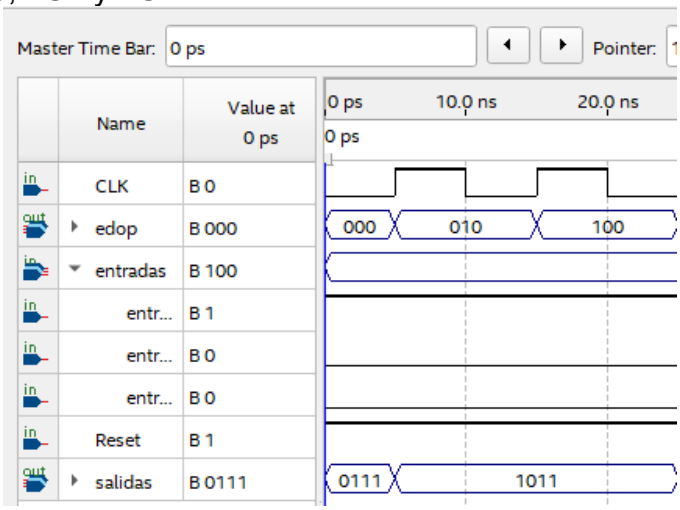
Estado 2, salidas S0, ¬U1 y ¬U2.



Estado 3, salidas ¬U1.



Estado 4. salidas S0, ¬U1 y ¬U2.



Conclusiones

Jiménez Treviño Emilio Cristóbal

En esta práctica se logró diseñar y simular una máquina de estados mediante el método de direccionamiento por trayectoria, confirmando su correcto funcionamiento en Quartus por medio de la simulación de modelsim y la FPGA DE10-Lite. Se observó que el uso de memorias facilita la implementación de lógica secuencial y se aprendió a traducir una carta ASM a bloques en VHDL y el correcto llenado de la memoria ROM que fue lo mas complicado, asi mismo las dificultades fueron la asignación de pines y la frecuencia del reloj, las cuales se solucionaron preguntando al profesor como se hacía y ya masomenos nos orientó y con eso pudimos hacer un correcto llenado de esta debido a que estabamos poniendo datos que no tenían que ver.

Martinez Perez Brian Erik

En esta práctica, hemos logrado construir y simular una máquina de estados utilizando el método de direccionamiento por trayectoria con memorias. A través del desarrollo de las actividades, se aplicaron los fundamentos teóricos para representar la carta ASM en una tabla de contenido de memoria. Este enfoque demostró ser un método sistemático y eficaz para la implementación de lógica secuencial, ya que el estado siguiente y las salidas se codifican directamente en la memoria. La fase de simulación en el software Quartus permitió validar el funcionamiento del diseño, confirmando que la máquina de estados se comporta de manera predecible y transiciona correctamente entre los estados según las entradas y las ligas de memoria.

Bibliografía

Laboratorio de Organización y Arquitectura de Computadoras. (2020, octubre 11). *Práctica No. 3 Construcción de Máquinas de estados Usando Memorias Direccionamiento por Trayectoria.*