	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: 5

Integrante(s): Jiménez Treviño Emilio Cristóbal

Martínez Pérez Brian Erik

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

Semestre: *2026-1*

Fecha de entrega: *17/10/2025*

Observaciones:

CALIFICACIÓN: _____

Práctica 5. Construcción de Máquinas de estados Usando Memorias Direccionamiento Implícito

Objetivo

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

Introducción

En esta práctica, exploramos la construcción de una máquina de estados con direccionamiento Entrada - Estado. A lo largo del documento, se detalla el proceso que inicia con la propuesta de una carta ASM, la cual servirá como base para la elaboración de la tabla de verdad, considerando el direccionamiento implícito. Posteriormente, se abordará la implementación práctica de este sistema mediante el software Quartus, donde se configura el dispositivo MAX 10 y se creará el proyecto. Se describirá la interconexión de bloques lógicos clave, como el contador, la memoria ROM, el registro y la lógica de control, cada uno con funciones específicas para el correcto funcionamiento de la máquina de estados. Finalmente, se presentarán los códigos VHDL utilizados para generar cada bloque, la asignación de pines en la FPGA DE-10 Lite y una serie de simulaciones que muestran el comportamiento de la maquina de estados.

Desarrollo

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

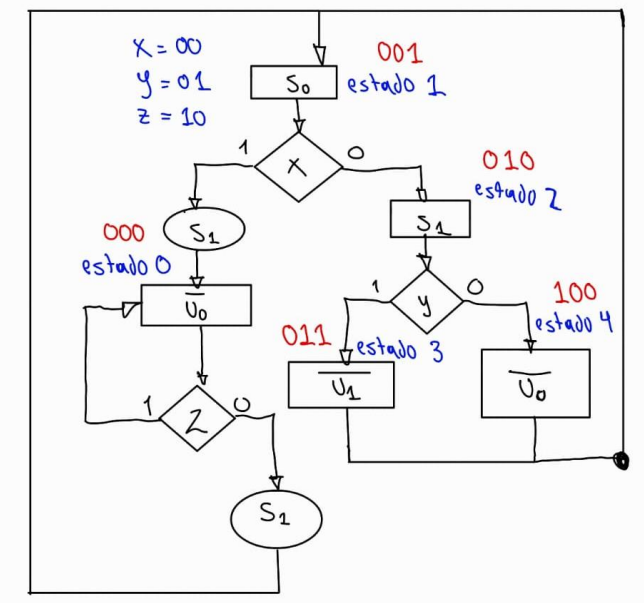


Figura 1. Carta ASM propuesta.

Después de obtener la carta ASM, obtenemos la tabla de verdad, donde debemos de tomar en cuenta que utilizamos el método de direccionamiento implícito. Debemos tener en la cabecera el estado presente, prueba, liga, valor falso, salidas falsa y salidas verdaderas.

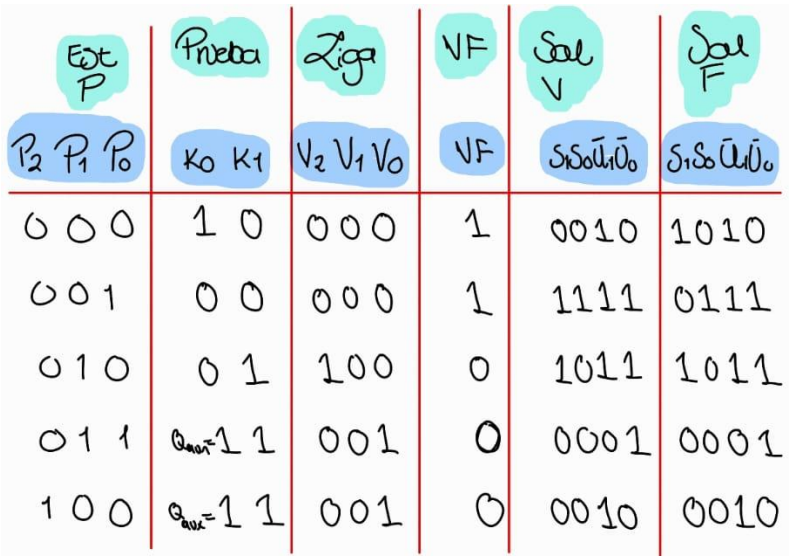


Figura 2. Tabla de verdad, obtenida por el método de direccionamiento implícito.

Cuando ya tenemos la tabla de verdad, podemos implementar el direccionamiento entrada-estado 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.

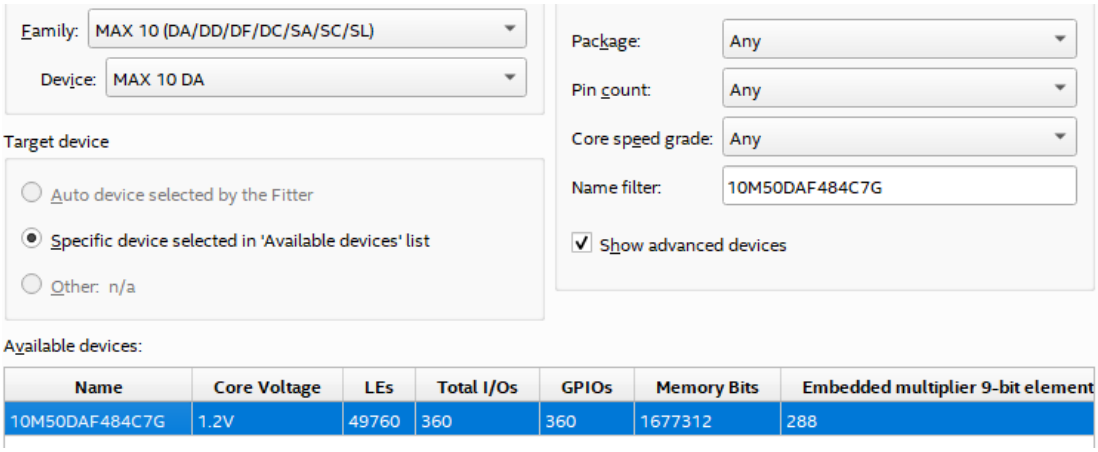


Figura 3. Device

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.

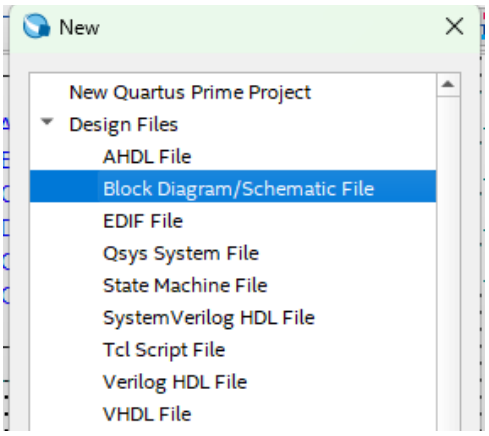


Figura 4. New File

Para implementar el circuito de direccionamiento implícito, debemos tener en cuenta que tenemos un bloque llamado “contador”, el cual tiene como entradas el estado siguiente, incrementa, carga, el reloj y reset. El bloque de memoria almacena, prueba, valor falso, liga, salidas verdaderas y salidas falsas. Saliendo de la memoria está la prueba y valor falso, los cuales entran en el bloque “lógica”, el cual recibe ambos, además de las entradas. El bloque genera carga e incrementa, ambos datos se dirigen al bloque de contador. Por último se tiene el bloque de “registro” el cual determina la salida que se mostrará en el estado presente.

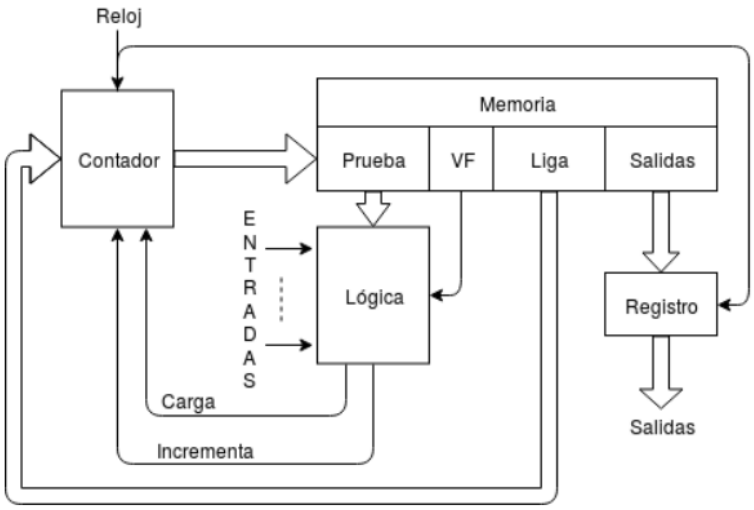


Figura 5. Diagrama circuito direccionamiento implícito.

El primer código generado fue “registró”, su función es almacenar el código del estado actual (edop, 3 bits) a partir del estado siguiente (edos). Es un circuito secuencial que se actualiza en el flanco de subida del reloj y puede ser reiniciado asíncronamente.

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

entity registro is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           edos : in std_logic_vector (2 downto 0);
           edop : out std_logic_vector (2 downto 0)
    );
end registro;


architecture Behavioral of registro is

signal internal_value: std_logic_vector (2 downto 0) := B"000";
begin
    process(reloj, reset, edos)
    begin

        if reset = '0' then
            internal_value <= B"000";
        elsif rising_edge(reloj) then
            internal_value <= edos;
        end if;
    end process;

    process(internal_value)
    begin
        edop <= internal_value;
    end process;


end Behavioral;
```

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

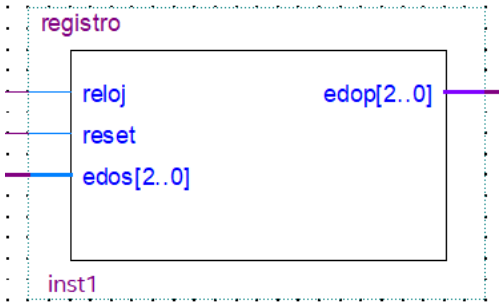


Figura 7. Bloque “registro”.

El segundo código generado fue “mux_control”, su función es seleccionar la entrada al Registro de Estado (edos), eligiendo entre la dirección incrementada (edouno) o una dirección de salto cargada (edok), determinado por la señal de control carga..

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

entity mux_control is
    Port (   carga: in std_logic;
             edouno : in std_logic_vector (2 downto 0);
             edok : in std_logic_vector (2 downto 0);
             edos : out std_logic_vector (2 downto 0)
    );
end mux_control;

architecture Behavioral of mux_control is
begin

    process(carga, edouno, edok)
    begin

    if carga = '0' then
          edos <= edouno;
    elsif carga = '1' then
          edos <= edok;
    end if;

    end process;

end Behavioral;
```

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

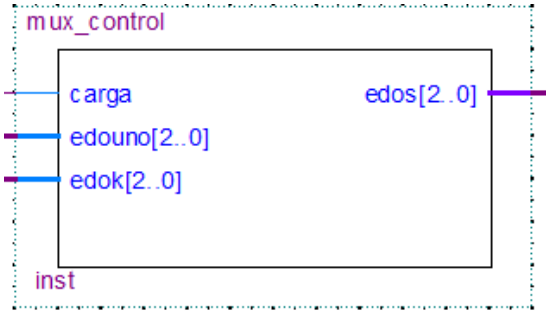


Figura 9. Bloque “mux_control”.

El tercer código generado fue “incrementador”, su función es generar el estado siguiente, calculando la dirección actual (edop) más 1 (edouno). Esta es la dirección predeterminada para el paso continuo de la microprogramación.

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

entity incrementador is
    Port (   edop: in std_logic_vector (2 downto 0);
             edouno : out std_logic_vector (2 downto 0)
    );
end incrementador;

architecture Behavioral of incrementador is
begin

    process(edop)
    begin

        edouno <= edop + 1;

    end process;

end Behavioral;
```

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

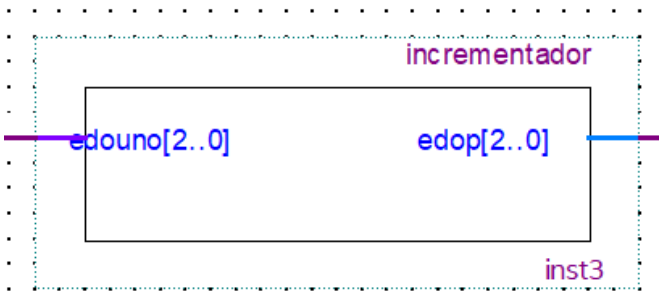


Figura 11. Bloque “incrementador”.

Para esta práctica se utilizaron 2 archivos esquemáticos, en esta primera parte vamos a conectar los bloques generados “registro”, “mux_control” y “incrementador”. El diagrama de conexiones se muestra en la imagen 12. Este diagrama esquemático se convertirá en otro bloque llamado “contador”, el cual será el que tendrá el control de los estados de nuestra carta ASM.

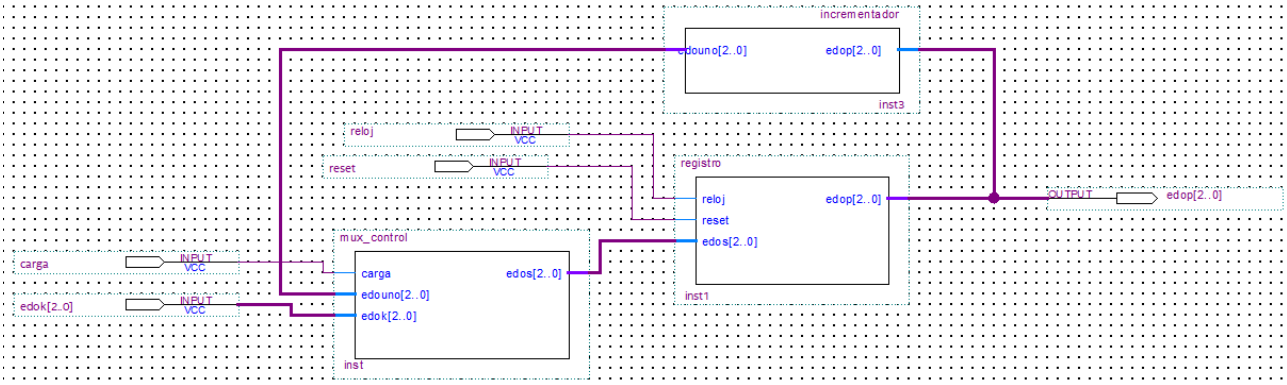


Figura 12. Diagrama de conexiones para el bloque “contador”

El cuarto código generado fue “rom”, su función es almacenar las microinstrucciones (palabras de control de 14 bits) en 8 direcciones. La dirección actual (dirección), produce un conjunto de señales de control (data) que definen el comportamiento de la carta ASM para el ciclo actual.

```
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 rom is
7       Port ( direccion : in std_logic_vector (2 downto 0);
8              data : out std_logic_vector (13 downto 0)
9       );
10  end rom;
11
12  architecture Behavioral of rom is
13
14  type mem is array (0 to 7) of std_logic_vector (13 downto 0);
15  signal internal_mem: mem;
16
17  begin
18
19      --  prueba & VF & liga & SF & SV
20
21      internal_mem(0) <= "10" & "1" & "000" & "1010" & "0010";
22      internal_mem(1) <= "00" & "1" & "000" & "0111" & "1111";
23      internal_mem(2) <= "01" & "0" & "100" & "1011" & "1011";
24      internal_mem(3) <= "11" & "0" & "001" & "0001" & "0001";
25      internal_mem(4) <= "11" & "0" & "001" & "0010" & "0010";
26      internal_mem(5) <= "11" & "0" & "001" & "0011" & "0011";
27      internal_mem(6) <= "11" & "0" & "001" & "0011" & "0011";
28      internal_mem(7) <= "11" & "0" & "001" & "0011" & "0011";
29       process(direccion)
30       begin
31         data <= internal_mem(conv_integer(unsigned(direccion)));
32
33      end process;
34
35  end Behavioral;
```

Figura 13. Código VHDL, para generar el bloque “rom”.

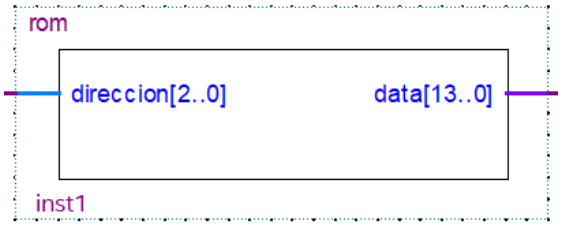


Figura 14. Bloque “rom”.

El quinto código generado fue “div_datos”, su función es dividir la palabra de control de 14 bits leída de la ROM (data) en sus campos específicos: el código de prueba (prueba), los campos para la dirección de salto (liga, VF), y los campos de salida (SF, SV).

```
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 div_datos is
7     Port ( data : in std_logic_vector (13 downto 0);
8
9               prueba : out std_logic_vector (1 downto 0);
10              VF: out std_logic;
11              liga: out std_logic_vector (2 downto 0);
12              SF : out std_logic_vector (3 downto 0);
13              SV : out std_logic_vector (3 downto 0)
14    );
15 end div_datos;
16
17
18 architecture Behavioral of div_datos is
19 begin
20
21    process(data)
22    begin
23
24          prueba <= data(13 downto 12);
25          VF <= data(11);
26          liga <= data(10 downto 8);
27          SF <= data(7 downto 4);
28          SV <= data(3 downto 0);
29
30    end process;
31
32 end Behavioral;
```

Figura 15. Código VHDL, para generar el bloque ”div_datos”.

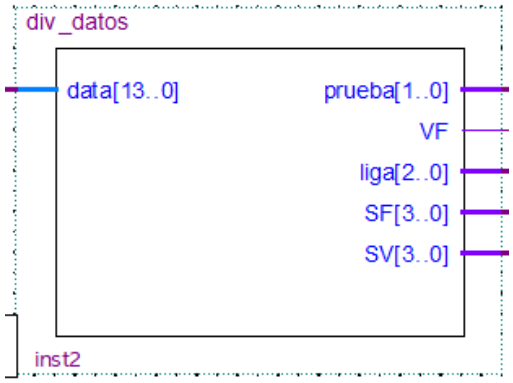


Figura 16. Bloque “div_datos”.

El sexto código generado fue “mux_prueba”, su función es.

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

entity mux_prueba is
    Port ( prueba : in std_logic_vector (1 downto 0);
           x: in std_logic;
           y: in std_logic;
           z: in std_logic;
           qaux: in std_logic;
           qsel: out std_logic
    );
end mux_prueba;

architecture Behavioral of mux_prueba is
begin

    process(prueba, x, y, z, qaux)
    begin

    if prueba = "00" then
        qsel <= x;
    elsif prueba = "01" then
        qsel <= y;
    elsif prueba = "10" then
        qsel <= z;
    elsif prueba = "11" then
        qsel <= qaux;
    end if;

    end process;

end Behavioral;
```

Figura 17. Código VHDL, para generar el bloque “mux_prueba”.

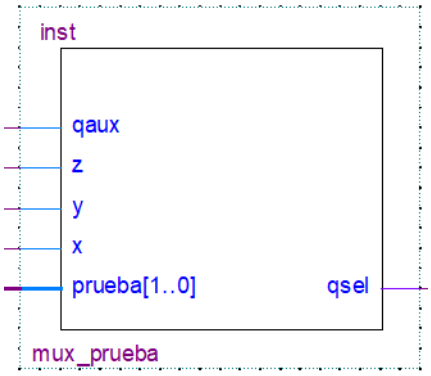


Figura 18. Bloque “mux_prueba”.

El séptimo código generado fue “mux_salida”, su función es controlar las salidas efectivas del sistema (salida, 4 bits). Selecciona entre un conjunto de salidas falsas (SF) o un conjunto de salidas verdaderas (SV) en función de la señal de prueba (qsel).

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

entity mux_salida is
    Port (    SF : in std_logic_vector (3 downto 0);
              SV : in std_logic_vector (3 downto 0);
              qsel: in std_logic;
              salida : out std_logic_vector (3 downto 0)
    );
end mux_salida;

architecture Behavioral of mux_salida is
begin

    process(qsel, SF, SV)
    begin

    if qsel = '0' then
          salida <= SF;
    elsif qsel = '1' then
          salida <= SV;
    end if;

    end process;

end Behavioral;
```

Figura 19. Código VHDL, para generar el bloque “mux_salida”.

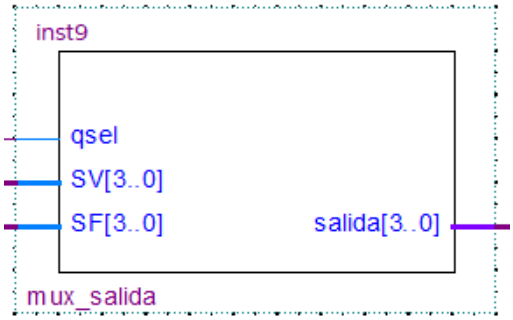


Figura 20. Bloque “mux_salida”.

El último código generado fue “div_frec”, su función es generar una señal de reloj de baja frecuencia (div_clk) a partir de un reloj de entrada más rápido (reloj), utilizando un contador interno, división de 25 millones de pulsos para nuestro caso. Este divisor de frecuencia nos ayuda a ver cómo cambia los estados a través de los leds.

```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"2FAF080" then --25M pulsos
                cuenta:= x"0000000";
            else
                cuenta:= cuenta+1;
            end if;
        end if;
        div_clk <= cuenta(25);
    end process;
end Behavioral;
```

Figura 21. Código VHDL, para generar el bloque “div_frec”.

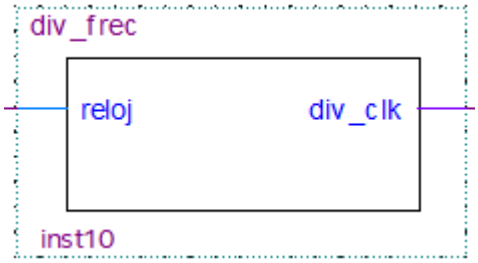


Figura 22. Bloque “div_frec”.

En el segundo archivo esquemático, tenemos todos los bloques generados a partir del código vhdl y del primer archivo esquemático, en este tenemos 5 entradas, “Reset”, “Z”, “X”, “Z” y “CLK”. Además tenemos 7 salidas que serán asignadas a los leds de la tarjeta. “salida[0]”, “salida[1]”, “salida[2]”, “salida[3]”, “edop[0]”, “edop[1]”, “edop[2]”.

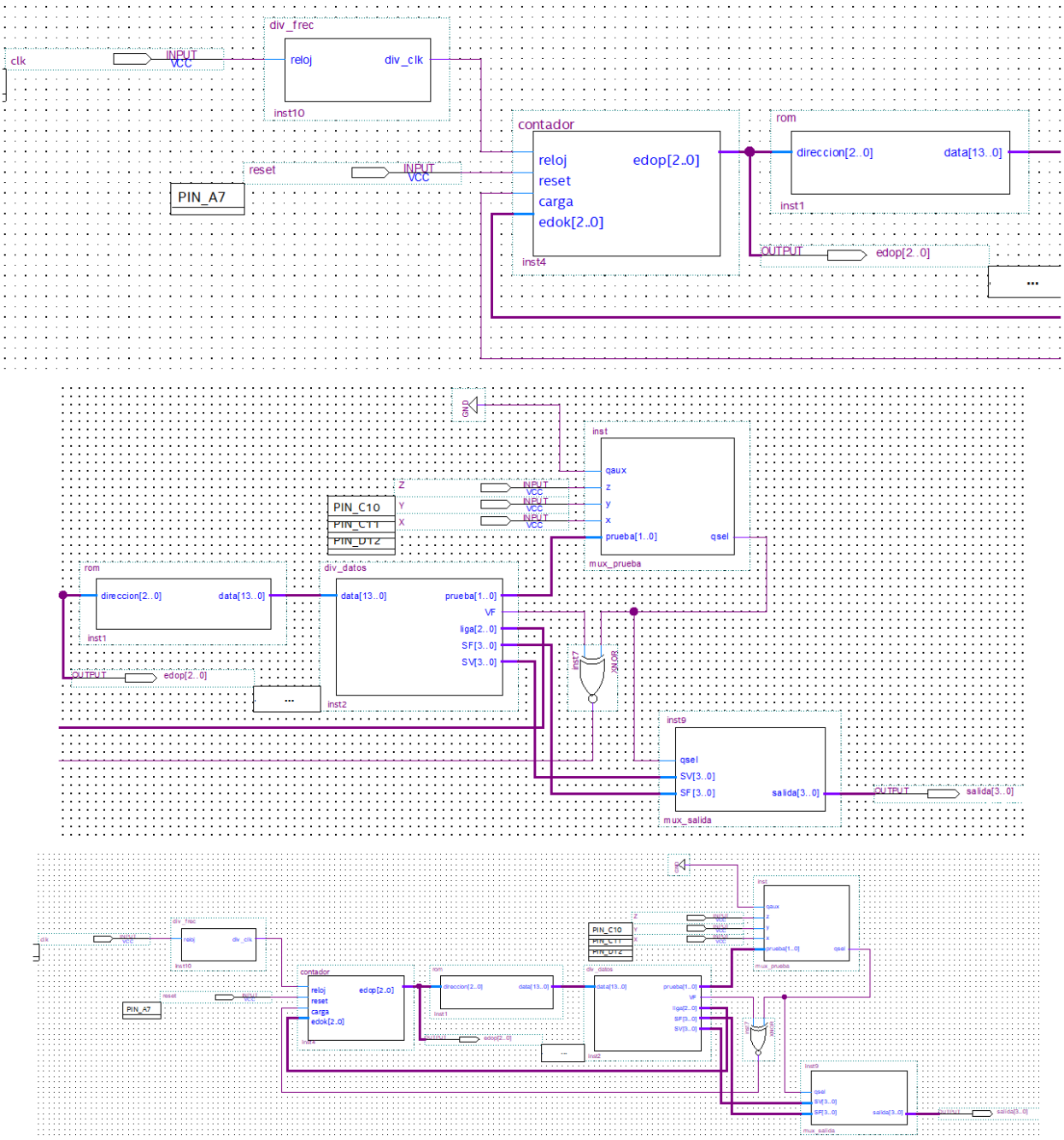


Figura 23. Diagrama de conexiones

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. 24

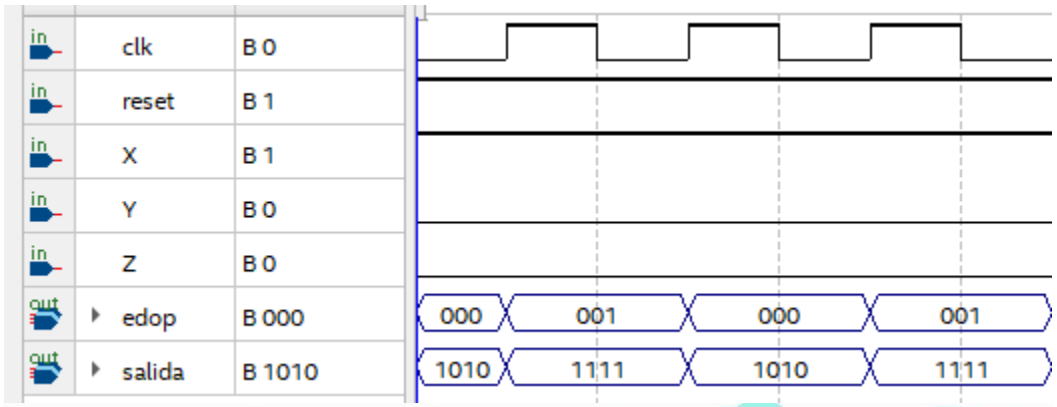
podremos ver cuales se asignaron para este caso, recordemos que deben asignarse desde el más significativo al menos, para una correcta secuencia.

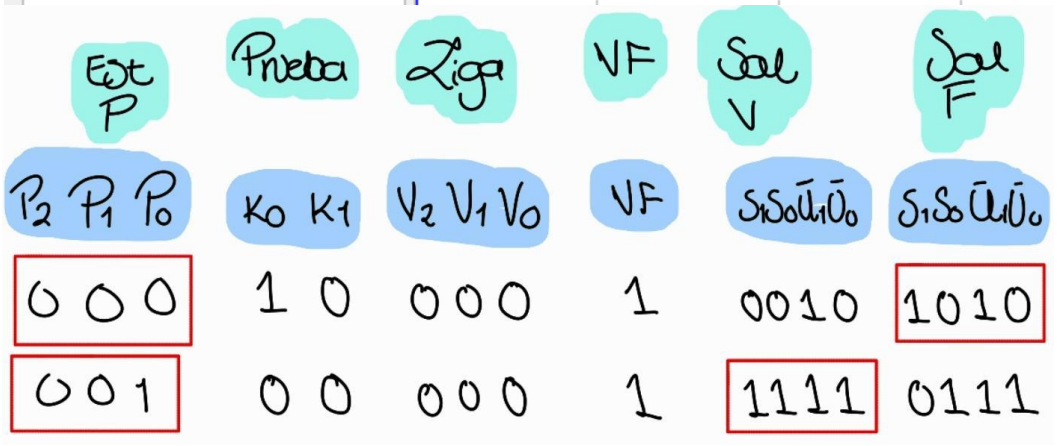
Node Name	Direction	Location
in Z	Input	PIN_C10
in Y	Input	PIN_C11
in X	Input	PIN_D12
in reset	Input	PIN_A7
in clk	Input	PIN_P11
out salida[0]	Output	PIN_A8
out salida[1]	Output	PIN_A9
out salida[2]	Output	PIN_A10
out salida[3]	Output	PIN_B10
out edop[0]	Output	PIN_D14
out edop[1]	Output	PIN_A11
out edop[2]	Output	PIN_B11

Figura 24. Asignación de pines

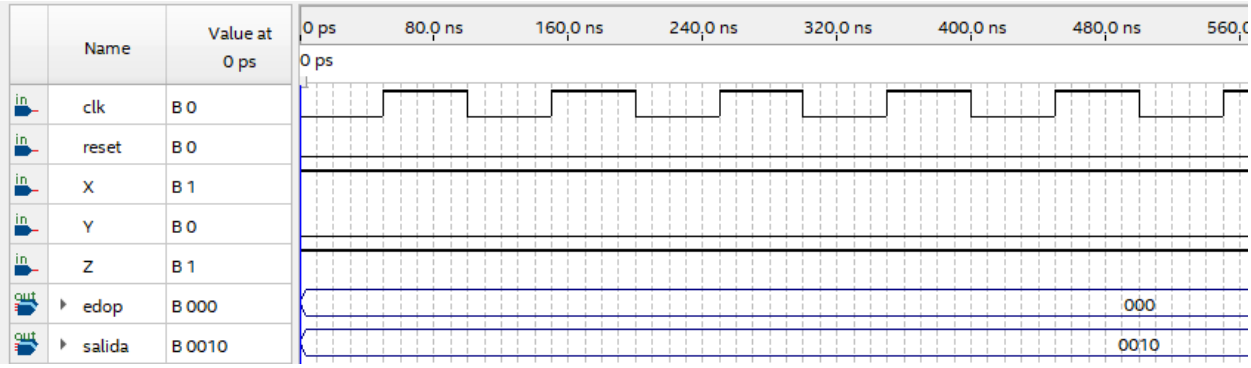
Simulación

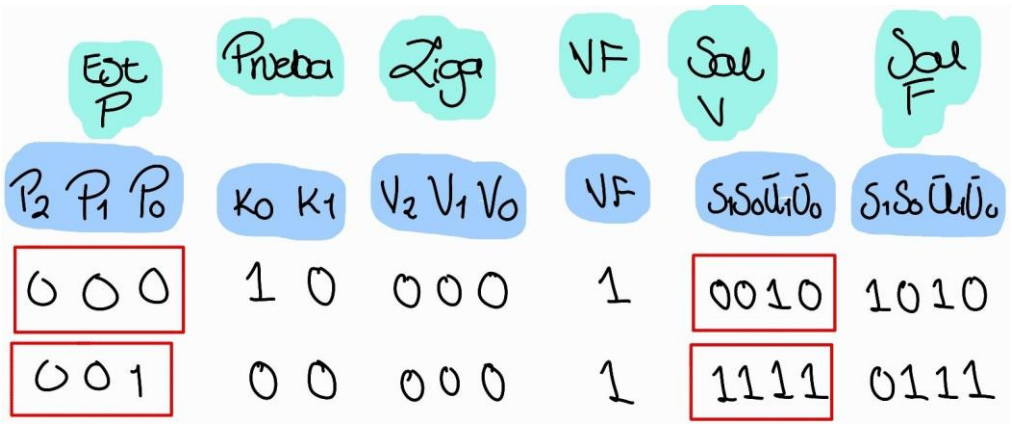
Secuencia con X=1, Z=0, estado 1 y estado 0.



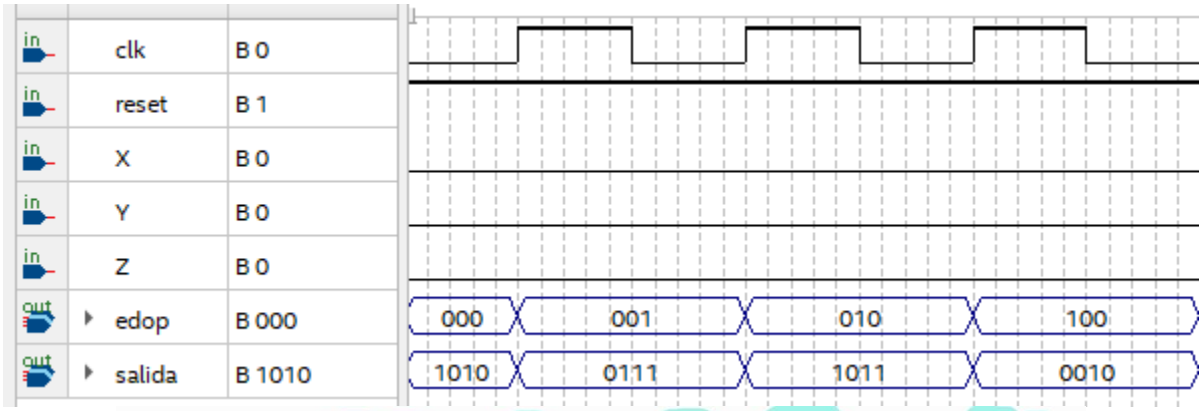


Secuencia con X=1, Z=1, estado 1 y estado 0.

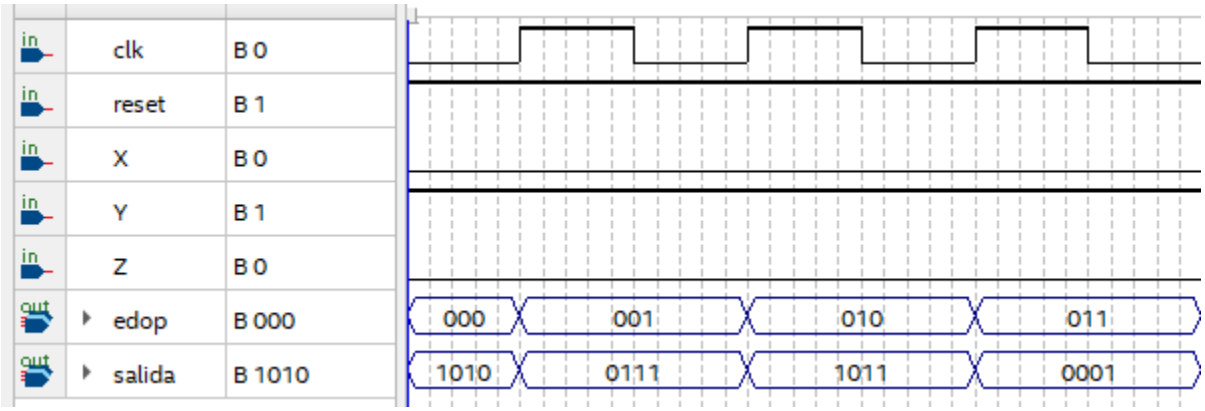




Secuencia con X=0, Y=0, estado 1, estado 2 y estado 4.



Secuencia con X=0, Y=1, estado 1, estado 2 y estado 3.



Est P	Prueba	Ziga	VF	Sal V	Sal F
$P_2\ P_1\ P_0$	$K_0\ K_1$	$V_2\ V_1\ V_0$	VF	$S_1 S_0 \bar{U}_1 \bar{U}_0$	$S_1 S_0 \bar{U}_1 U_0$
0 0 1	0 0	0 0 0	1	1 1 1 1	0 1 1 1
0 1 0	0 1	1 0 0	0	1 0 1 1	1 0 1 1
0 1 1	Qmar=1 1	0 0 1	0	0 0 0 1	0 0 0 1

Conclusiones

Jiménez Treviño Emilio Cristóbal

En esta práctica se implemento la construcción de Máquinas de estados Usando Memorias Direccionamiento Entrada - Estado en este método de direccionamiento pudimos observar el como maneja la memoria ROM logrando tener mayor eficiencia, en esta práctica tuvimos problemas al interpretar y lograr implementar este método, pudimos resolverlo investigando y visualizando que resultados tenemos para corregirlo, creo que ahora que ya tenemos un mayor conocimiento acerca del comportamiento resultante y del cómo se comporta esta máquina de estados.

Martinez Perez Brian Erik

La implementación de la Máquina de Estados Finitos mediante el método de direccionamiento implícito fue el tema central de esta práctica, demostrando ser una técnica esencial en la arquitectura de computadoras. La eficiencia de este método se valida al usar la Memoria ROM como elemento principal de control, donde se almacena el comportamiento completo de la máquina. La lógica de transición de estados se controla con la señal de “Incrementa” y la señal de “Carga” , simplificando el hardware del contador. Los módulos VHDL, desde el div_datos.vhd que segmenta la microinstrucción hasta el mux_prueba.vhd que selecciona la condición de salto, permitieron construir un sistema de direccionamiento de memoria.

Bibliografía

Laboratorio de Organización y Arquitectura de Computadoras. (2020, octubre 26). *Práctica No. 5 Construcción de Máquinas de estados Usando Memorias Direccionamiento Implıcito.*