



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: 6

Integrante(s): Jiménez Treviño Emilio Cristóbal

Martinez Perez Brian Erik

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

Semestre: 2026-1

Fecha de entrega: 7/11/2025

Observaciones: _____

CALIFICACIÓN: _____

Práctica 6. Secuenciador básico

Objetivo

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

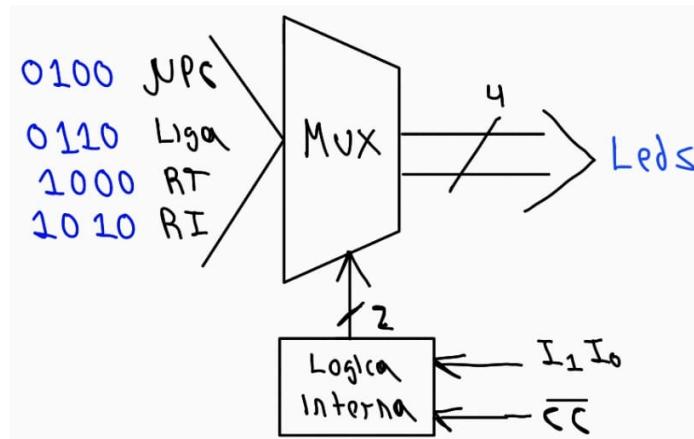
Introducción

El secuenciador básico representa el componente fundamental de la unidad de control en un procesador, responsable de generar y gestionar las direcciones de las microinstrucciones que definen el flujo de ejecución de un sistema digital. Esta práctica introduce el diseño e implementación de un secuenciador capaz de ejecutar cuatro tipos de instrucciones: continua, salto condicional, salto de transformación y saltó por interrupción.

A través del desarrollo con VHDL y herramientas Quartus, se explorará la arquitectura interna del secuenciador, que integra registros especializados (μ PC, transformación e interrupción), multiplexores para selección de direcciones y lógica de control basada en condiciones. La práctica permitirá comprender cómo este componente coordina el flujo de ejecución mediante diferentes modos de direccionamiento, manejando tanto la secuencia normal de instrucciones como desvíos por condiciones lógicas o eventos externos.

Desarrollo

Para comenzar primero se realizó la primera parte del secuenciador básico, está tomaba en cuenta los bloques de “lógica interna” y “mux”, para ello se asignaron valores arbitrarios a los registros de entrada del multiplexor, microinstrucción sería “0100”, liga “0110”, RT “1000” y por último RI “1010”.



| $I_1 I_0 \bar{CC}$ | selector | Comentarios |
|--------------------|----------|-------------------------|
| 00 * | 00 | Paso continuo |
| 01 0 | 11 | Salto Condicional |
| 01 1 | 00 | Paso continuo |
| 10 * | 01 | Salto de Transformación |
| 11 0 | 10 | Salto por interrupción |
| 11 1 | 00 | Paso Continuo |

Figura 1. parte 1 del diagrama del secuenciador “mux” y “lógica interna”.

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. 2 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.

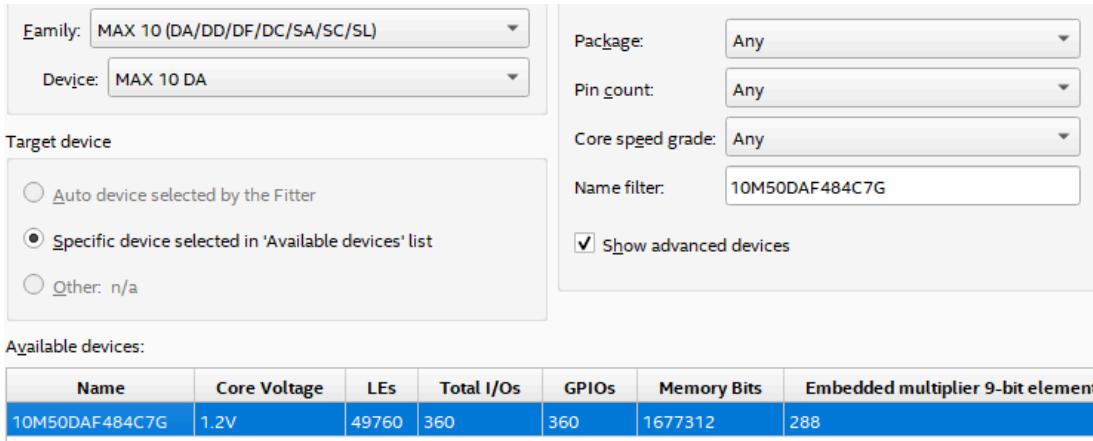


Figura 2. Device

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. 3.

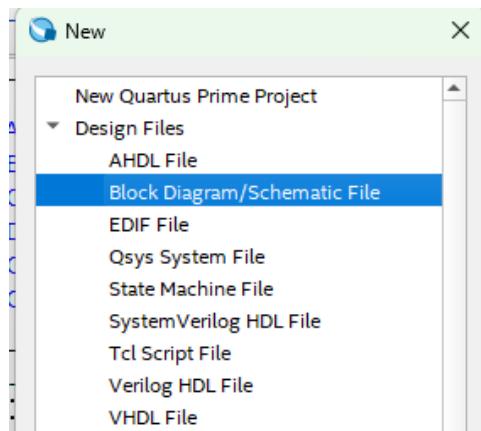


Figura 3. New File

El primer código generado fue “lógica”, el cual implementa un decodificador combinacional. Toma el código de la microinstrucción (I1I0) y una condición de prueba (cc). Genera el código de control de 2 bits (sel) que determina el tipo de salto a realizar (Paso Continuo, Salto Condicional, Salto de Transformación o Salto por Interrupción).

```

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

entity logica is
  Port (
    I0: in std_logic;
    I1: in std_logic;
    cc: in std_logic;                      -- Condición de control/prueba
    sel: out std_logic_vector (1 downto 0) -- Salida del selector de modo
  );
end logica;

architecture Behavioral of logica is
  signal I_sel : std_logic_vector(1 downto 0);
begin
  I_sel <= I1 & I0;
  process(I_sel, cc)
  begin
    sel <= "00";
    case I_sel is
      when "00" =>
        sel <= "00";
      when "01" =>
        if cc = '0' then
          sel <= "11"; -- Salto Condicional (selector=11 si cc=0)
        else -- cc = '1'
          sel <= "00"; -- Paso Continuo (selector=00 si cc=1)
        end if;
      when "10" =>
        sel <= "01";
      when "11" =>
        if cc = '0' then
          sel <= "10"; -- Salto por Interrupción (selector=10 si cc=0)
        else -- cc = '1'
          sel <= "00"; -- Paso Continuo (selector=00 si cc=1)
        end if;
      when others =>
        sel <= "00";
    end case;
  end process;
end Behavioral;

```

Figura 4. Código VHDL, para generar el bloque “lógica”.

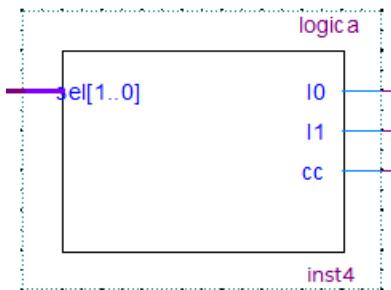


Figura 5. Bloque “lógica”.

El segundo código generado fue “mux”. Es el componente central para determinar el siguiente estado. Selecciona una de cuatro posibles direcciones de 4 bits (mpc, liga, rt, o ri) y la asigna a la salida de estado siguiente (edop). La selección se basa en el código de 2 bits (sel) generado por el módulo logica.

```

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

entity mux is
    Port ( mpc : in std_logic_vector (3 downto 0);
            liga : in std_logic_vector (3 downto 0);
            rt : in std_logic_vector (3 downto 0);
            ri : in std_logic_vector (3 downto 0);
            sel: in std_logic_vector (1 downto 0);
            edop: out std_logic_vector (3 downto 0)
        );
end mux;

architecture Behavioral of mux is
begin
    process(mpc, liga, rt, ri, sel)
    begin
        if sel = "00" then
            edop <= mpc;
        elsif sel = "01" then
            edop <= rt;
        elsif sel = "10" then
            edop <= ri;
        elsif sel = "11" then
            edop <= liga;
        end if;
    end process;
end Behavioral;

```

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

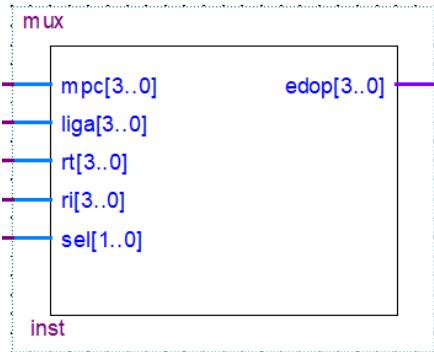


Figura 7. Bloque “mux”.

El tercer código generado fue “registro_mpc”. Almacena el estado actual o la dirección de la microinstrucción (mpc).

```

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

entity registro_mpc is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           mpc : out std_logic_vector (3 downto 0)
    );
end registro_mpc;

architecture Behavioral of registro_mpc is
begin
    internal_value: std_logic_vector (3 downto 0) := B"0000";
begin
    process(reloj, reset)
    begin
        if reset = '0' then
            internal_value <= B"0000";
        elsif rising_edge(reloj) then
            internal_value <= B"0100";
        end if;
    end process;
    process(internal_value)
    begin
        mpc <= internal_value;
    end process;
end Behavioral;

```

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

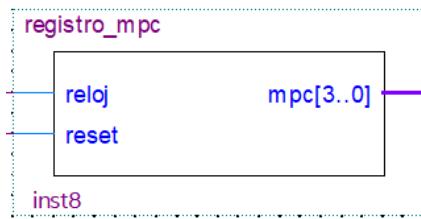


Figura 9. Bloque “registro_mpc”.

El cuarto código generado fue “registro_liga”. Almacena un valor fijo (B"0110") que representa la dirección de retorno o de salto incondicional.

```

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

entity registro_liga is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           liga : out std_logic_vector (3 downto 0)
    );
end registro_liga;

architecture Behavioral of registro_liga is
begin
    internal_value: std_logic_vector (3 downto 0) := B"0000";
begin
    process(reloj, reset)
    begin
        if reset = '0' then
            internal_value <= B"0000";
        elsif rising_edge(reloj) then
            internal_value <= B"0110";
        end if;
    end process;
    process(internal_value)
    begin
        liga <= internal_value;
    end process;
end Behavioral;

```

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



Figura 11. Bloque “registro_liga”.

El quinto código generado fue “registro_ri”. Almacena un valor fijo (B"1010") que representa una dirección de interrupción.

```

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

entity registro_ri is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           ri : out std_logic_vector (3 downto 0)
    );
end registro_ri;

architecture Behavioral of registro_ri is
-signal internal_value: std_logic_vector (3 downto 0) := B"0000";
begin
    process(reloj, reset)
begin
    if reset = '0' then
        internal_value <= B"0000";
    elsif rising_edge(reloj) then
        internal_value <= B"1010";
    end if;
end process;
    process(internal_value)
begin
    ri <= internal_value;
end process;
end Behavioral;

```

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



Figura 13. Bloque “registro_ri”.

El sexto código generado fue “registro_rt”. Almacena un valor fijo (B"1000") que puede representar una dirección de transformación o destino.

```

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

entity registro_rt is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           rt : out std_logic_vector (3 downto 0)
    );
end registro_rt;

architecture Behavioral of registro_rt is
begin
    signal internal_value: std_logic_vector (3 downto 0) := B"0000";
begin
    process(reloj, reset)
    begin
        if reset = '0' then
            internal_value <= B"0000";
        elsif rising_edge(reloj) then
            internal_value <= B"1000";
        end if;
    end process;
    process(internal_value)
    begin
        rt <= internal_value;
    end process;
end Behavioral;

```

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



Figura 15. Bloque “registro_rt”.

Para terminar la primer parte generamos todas las conexiones en el archivo esquemático, y asignamos la salida del sistema “edop” a una secuencia de leds, y las entradas de lógica “I1”, “I2” y “CC” serán los switches para controlar las salidas.

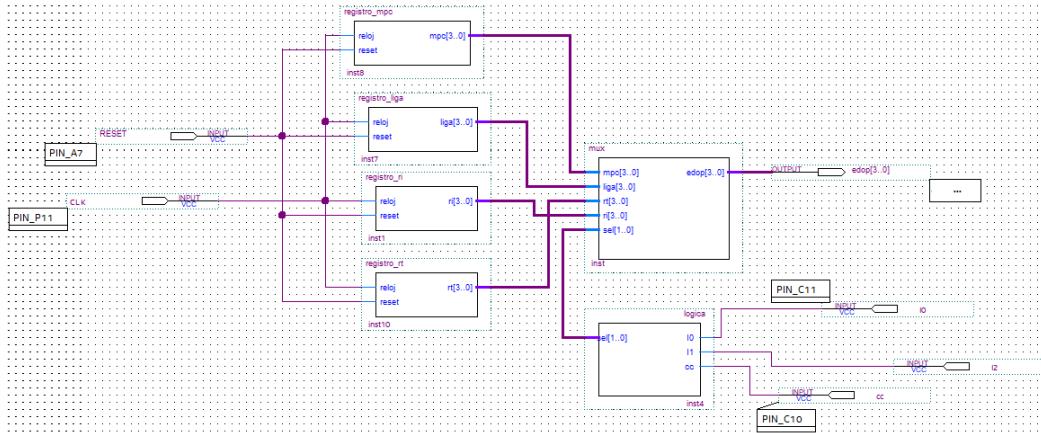


Figura 16. Diagrama de conexiones parte 1.

Para la parte 2 del secuenciador simplemente se agregó el “incrementador”, para que la parte de “mpc” pueda pasar por todos los estados declarados. para ello se toma en cuenta que el incrementador recibe el estado presente y como resultado manda el contenido de “mpc” al multiplexor, así como en el diagrama de la figura 17.

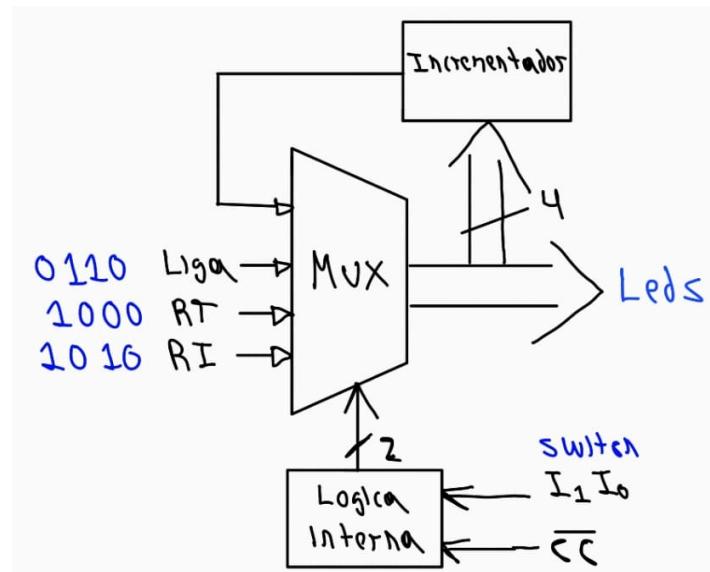


Figura 17. parte 2 del diagrama del secuenciador con “incrementador”.

El séptimo código generado fue “incrementador”. Este módulo implementa una lógica puramente combinacional cuya única función es calcular el estado siguiente secuencial.

Toma la dirección o estado actual (edop, 4 bits) y calcula la dirección siguiente (edouno) sumándole 1.

```
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 (3 downto 0);
              edouno : out std_logic_vector (3 downto 0)
            );
end incrementador;

architecture Behavioral of incrementador is
begin

    process(edop)
begin
    edouno <= edop + 1;
    end process;
end Behavioral;
```

Figura 18. Código VHDL, para generar el bloque “incrementador”.

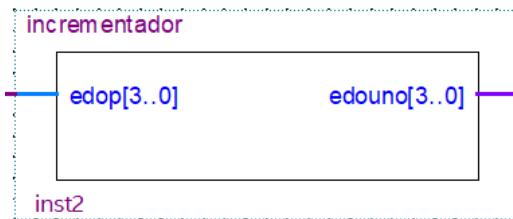


Figura 19. Bloque “incrementador”.

El código “registro_mpc”, tuvo que ser modificado para almacenar la dirección actual de la microinstrucción.

```

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

entity registro_mpc is
    Port ( reloj : in std_logic;
           reset : in std_logic;
           mpc : out std_logic_vector (3 downto 0)
    );
end registro_mpc;

architecture Behavioral of registro_mpc is
begin
    signal internal_value: std_logic_vector (3 downto 0) := B"0000";
begin
    process(reloj, reset)
    begin
        if reset = '0' then
            internal_value <= B"0000";
        elsif rising_edge(reloj) then
            internal_value <= B"0100";
        end if;
    end process;

    process(internal_value)
    begin
        mpc <= internal_value;
    end process;
end Behavioral;

```

Figura 20. Código VHDL modificado para el bloque “registro_mpc”.

Para terminar la parte 2 del secuenciador, se procedió a integrar el incrementador en el diagrama esquemático, para ello se tuvo en cuenta que el incrementador recibe el estado presente y envío su resultado al registro de “mpc”.

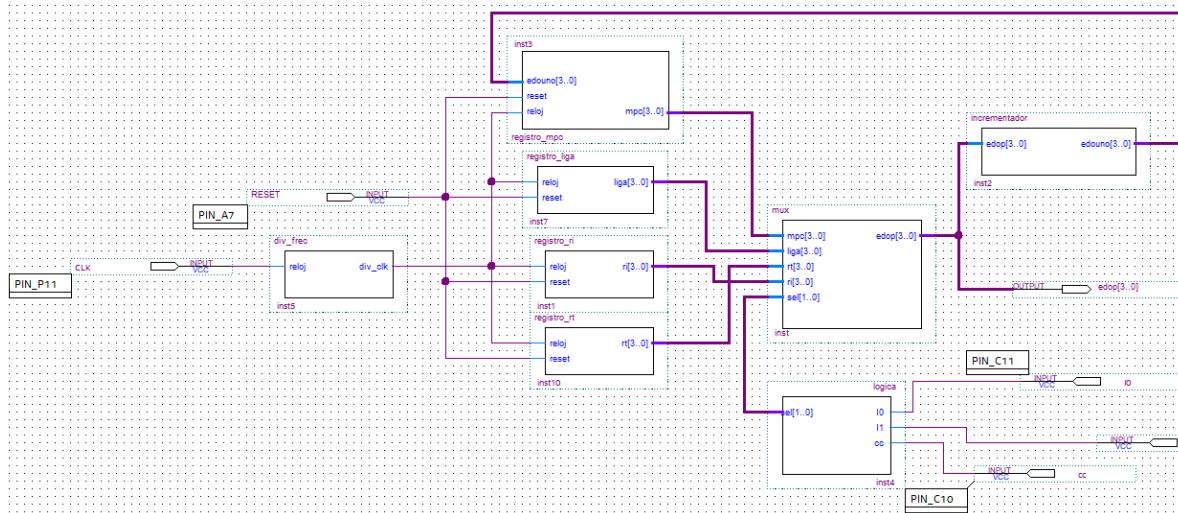


Figura 21. Diagrama de conexiones parte 2.

Para la última parte del secuenciador se solicitó crear una carta ASM con las siguientes restricciones, Estados: 15 Entradas: 2 Salidas: 4 Al menos 2 saltos de transformación Al menos 2 saltos de interrupción Al menos 2 salidas en lógica negada, además de generar la tabla de verdad para poder llenar la memoria ROM del secuenciador.

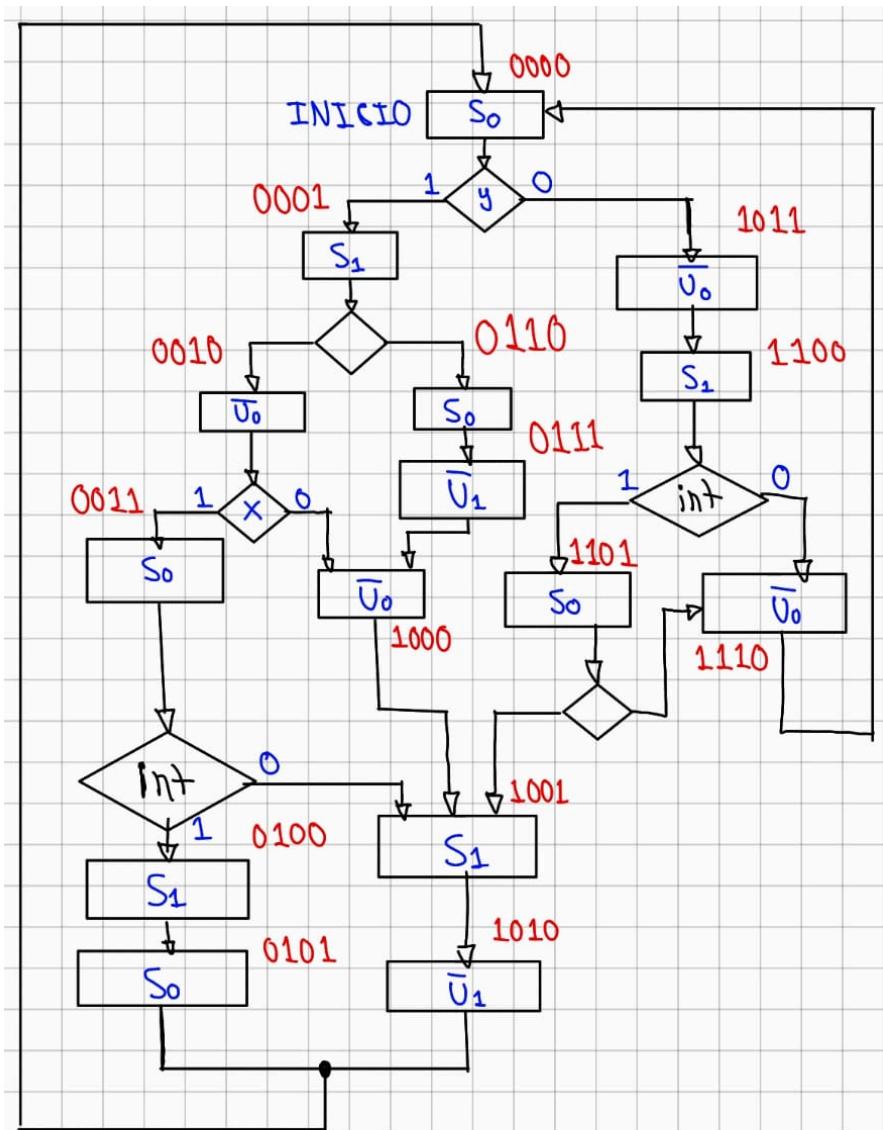


Figura 22. carta ASM creada.

| E.presente | | | | prueba | | Liga | | | | instrucción | | Salidas | | | | |
|------------|----|----|----|--------|----|------|----|----|----|-------------|----|---------|----|----|----|----|
| P3 | P2 | P1 | P0 | E1 | E0 | VF | L3 | L2 | L1 | L0 | I1 | I0 | S1 | S0 | U1 | U0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | * | * | * | * | * | * | * | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | * | * | * | * | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | * | * | * | * | * | * | * | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | * | * | * | * | * | * | * | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | * | * | * | * | * | * | * | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | * | * | * | * | * | * | * | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | * | * | * | * | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | * | * | * | * | * | * | * | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | * | * | * | * | * | * | * | 0 | 0 | 0 | 0 | 1 | 1 |

Figura 23. tabla de verdad de la carta ASM creada.

El octavo código generado fue “rom”. Este módulo implementa la Memoria de Microprograma (ROM). Almacena todas las instrucciones de control del sistema. Almacena 16 palabras de control (direcciones 0 a 15), donde cada palabra es un vector de 13 bits (12 downto 0). Cuando se le proporciona una dirección de entrada de 4 bits (dirección), devuelve el vector de control de 13 bits (data) almacenado en esa posición.

```

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 (3 downto 0);
            data : out std_logic_vector (12 downto 0)
        );
end rom;

architecture Behavioral of rom is

type mem is array (0 to 15) of std_logic_vector (12 downto 0);
signal internal_mem: mem;

begin
    -- prueba & VF & Ligas & MI & Salidas

    internal_mem(0) <= "10" & "0" & "1011" & "01" & "0111";
    internal_mem(1) <= "00" & "0" & "0000" & "10" & "1011";
    internal_mem(2) <= "01" & "0" & "1000" & "01" & "0010";
    internal_mem(3) <= "11" & "0" & "0000" & "11" & "0111";
    internal_mem(4) <= "00" & "0" & "0000" & "00" & "1011";
    internal_mem(5) <= "00" & "0" & "0000" & "00" & "0111";
    internal_mem(6) <= "00" & "0" & "0000" & "00" & "0111";
    internal_mem(7) <= "00" & "0" & "0000" & "00" & "0001";
    internal_mem(8) <= "00" & "0" & "0000" & "00" & "0010";
    internal_mem(9) <= "00" & "0" & "0000" & "00" & "1011";
    internal_mem(10) <= "00" & "0" & "0000" & "00" & "0001";
    internal_mem(11) <= "00" & "0" & "0000" & "00" & "0010";
    internal_mem(12) <= "11" & "0" & "0000" & "11" & "1011";
    internal_mem(13) <= "00" & "0" & "0001" & "10" & "0111";
    internal_mem(14) <= "00" & "0" & "0000" & "00" & "0010";
    internal_mem(15) <= "00" & "0" & "0000" & "00" & "0011";
process(direccion)
begin
    data <= internal_mem(conv_integer(unsigned(direccion)));
end process;

```

Figura 24. Código VHDL, para generar el bloque “rom”.

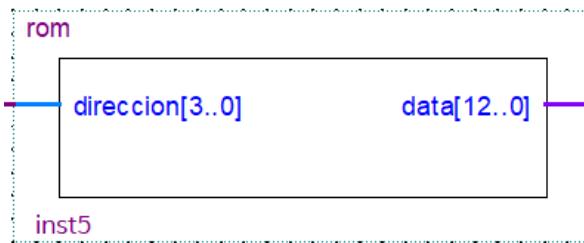


Figura 25. Bloque “rom”.

El noveno código generado fue “div_datos”. Su función es dividir la palabra de control de 13 bits (data) en sus campos lógicos funcionales. prueba (2 bits), VF (1 bit), liga (4 bits), MI (2 bits), salidas (4 bits)

```

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 (12 downto 0);
            prueba : out std_logic_vector (1 downto 0);
            VF: out std_logic;
            liga: out std_logic_vector (3 downto 0);
            MI : out std_logic_vector (1 downto 0);
            salidas: out std_logic_vector (3 downto 0)
        );
end div_datos;

architecture Behavioral of div_datos is
begin
    process(data)
    begin
        prueba <= data(12 downto 11);
        VF <= data(10);
        liga <= data(9 downto 6);
        MI <= data(5 downto 4);
        salidas <= data(3 downto 0);
    end process;
end Behavioral;

```

Figura 26. Código VHDL, para generar el bloque “div_datos”.

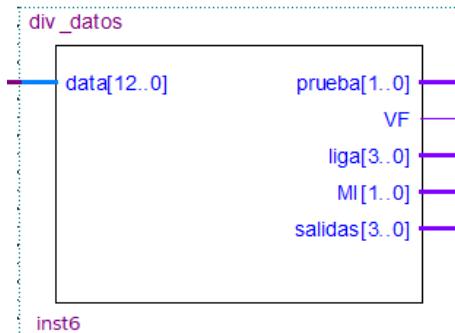


Figura 27. Bloque “div_datos”.

El décimo código generado fue “mux_prueba”. selecciona la variable lógica que se utilizará para el salto condicional. Utiliza el campo prueba (2 bits) para seleccionar una de las cuatro entradas de condición disponibles (qaux, x, y, o int) y la asigna a la salida qsel (la condición lógica seleccionada).

```

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;
            int: 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, int, qaux)
    begin
        if prueba = "00" then
            qsel <= qaux;
        elsif prueba = "01" then
            qsel <= x;
        elsif prueba = "10" then
            qsel <= y;
        elsif prueba = "11" then
            qsel <= int;
        end if;
    end process;
end Behavioral;

```

Figura 28. Código VHDL, para generar el bloque “mux_prueba”.

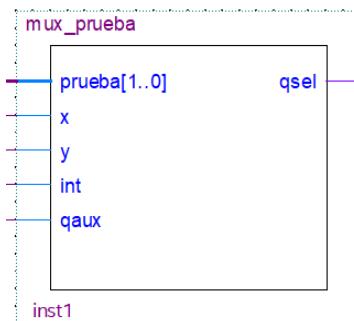


Figura 27. Bloque “mux_prueba”.

Al ya tener terminado por completo el secuenciador básico de 4 instrucciones, el contenido de algunos bloques de código tuvo que ser modificado para que tuviera el comportamiento de la carta ASM creada. los códigos modificados fueron los siguientes: “lógica”, “registro_ri” y “registro_rt”.

```

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

entity logica is
    Port (
        I0: in std_logic;
        I1: in std_logic;
        cc: in std_logic;
        sel: out std_logic_vector (1 downto 0)
    );
end logica;

architecture Behavioral of logica is
    signal I_sel : std_logic_vector(1 downto 0);
begin
    I_sel <= I1 & I0;
    process(I_sel, cc)
    begin
        sel <= "00";
        case I_sel is
            when "00" =>
                sel <= "00";
            when "01" =>
                if cc = '0' then
                    sel <= "11";
                else
                    sel <= "00";
                end if;
            when "10" =>
                sel <= "01";
            when "11" =>
                if cc = '0' then
                    sel <= "10";
                else
                    sel <= "00";
                end if;
            when others =>
                sel <= "00";
        end case;
    end process;
end Behavioral;

```

Figura 28. Código VHDL modificado para el bloque “lógica”.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity registro_ri is
    Port( CLK : in STD_LOGIC;
          RESET : in STD_LOGIC;
          ENA : in STD_LOGIC;
          DATA_IN : in STD_LOGIC_VECTOR(3 downto 0); --Salto
          DATA_OUT : out STD_LOGIC_VECTOR(3 downto 0)--Edo Sig
        );
end registro_ri;

architecture Behavioral of registro_ri is
    signal valor_interno : std_logic_vector(3 downto 0) := B"0000";
    constant alta_impedancia : std_logic_vector(3 downto 0) := "ZZZZ";
    constant zero : std_logic_vector(3 downto 0) := B"0000";
begin
    process (CLK, RESET, DATA_IN)
    begin
        if RESET = '0' then
            valor_interno <= zero;
        elsif rising_edge (CLK) then
            valor_interno <= DATA_IN;
        end if;
    end process;

    process(valor_interno, ENA)
    begin
        if ENA = '1' then
            DATA_OUT <= alta_impedancia;
        else
            DATA_OUT <= valor_interno;
        end if;
    end process;
end Behavioral;

```

Figura 29. Código VHDL modificado para el bloque “registro_ri”.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity registro_rt is
    Port( CLK : in STD_LOGIC;
          RESET : in STD_LOGIC;
          ENA : in STD_LOGIC;
          DATA_IN : in STD_LOGIC_VECTOR(3 downto 0); --Salto
          DATA_OUT : out STD_LOGIC_VECTOR(3 downto 0)--Edo Sig
        );
end registro_rt;

architecture Behavioral of registro_rt is
    signal valor_interno : std_logic_vector(3 downto 0) := B"0000";
    constant alta_impedancia : std_logic_vector(3 downto 0) := "ZZZZ";
    constant zero : std_logic_vector(3 downto 0) := B"0000";
begin
    process (CLK, RESET, DATA_IN)
    begin
        if RESET = '0' then
            valor_interno <= zero;
        elsif rising_edge (CLK) then
            valor_interno <= DATA_IN;
        end if;
    end process;

    process(valor_interno, ENA)
    begin
        if ENA = '1' then
            DATA_OUT <= alta_impedancia;
        else
            DATA_OUT <= valor_interno;
        end if;
    end process;
end Behavioral;

```

Figura 30. Código VHDL modificado para el bloque “registro_rt”.

Al final el diagrama esquemático quedó de la siguiente forma.

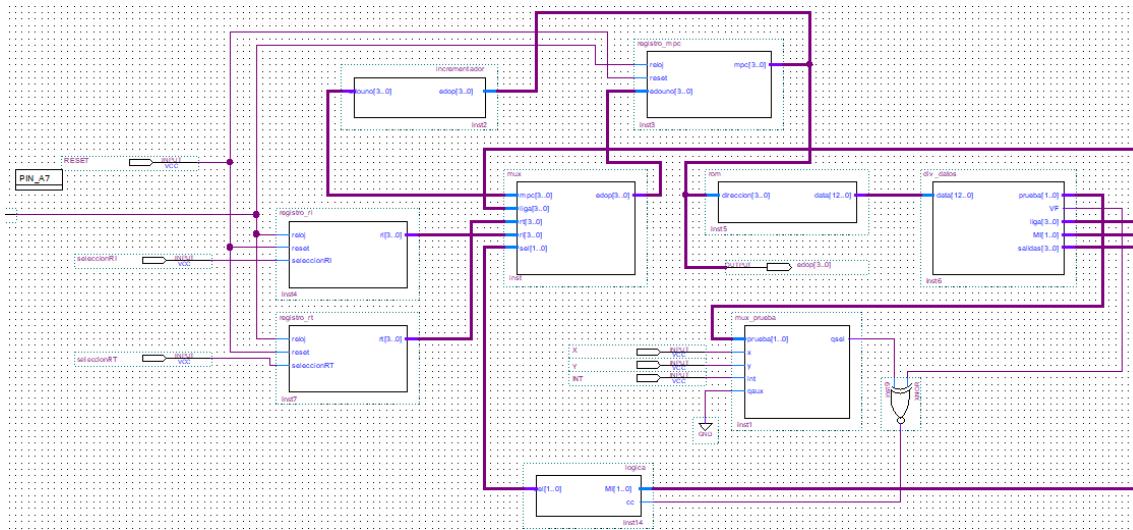


Figura 31. Diagrama de conexiones para el secuenciador de 4 instrucciones.

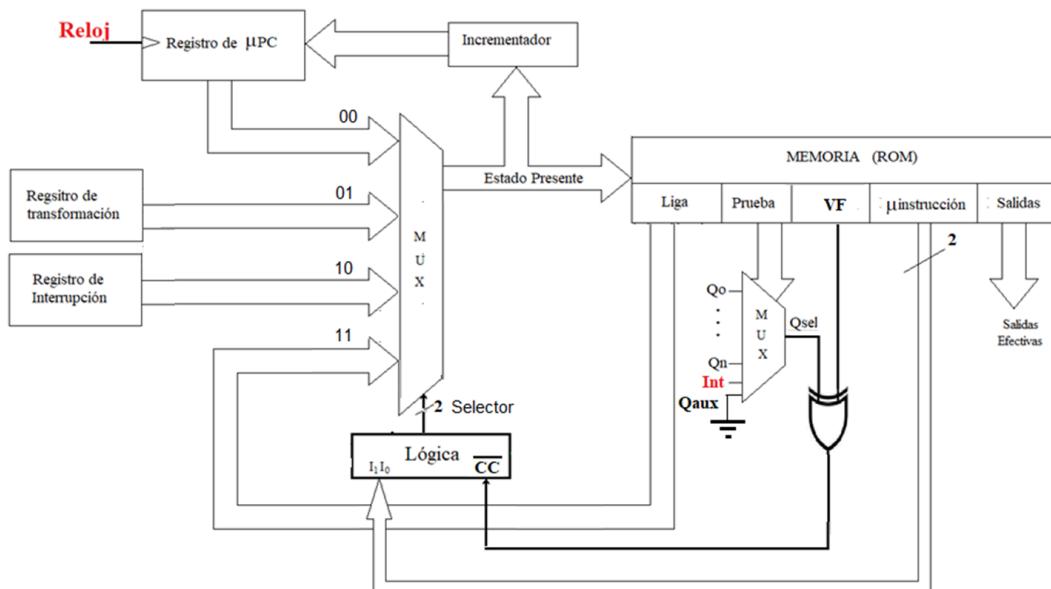


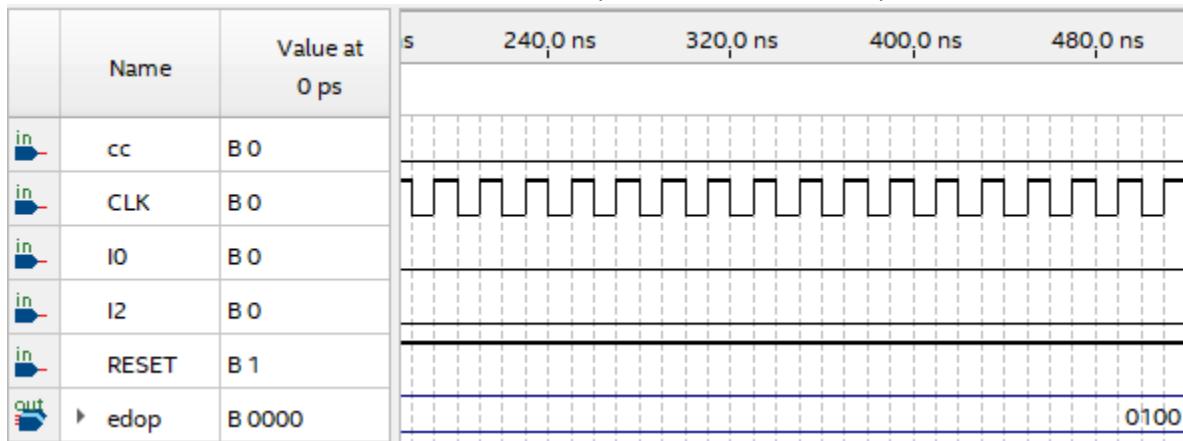
Figura 31. Diagrama del secuenciador básico de 4 instrucciones.

Simulaciones

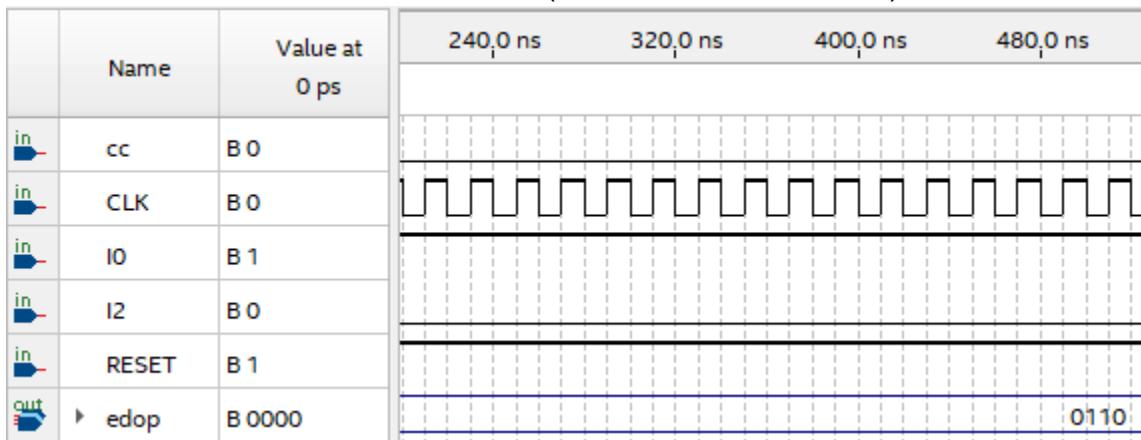
| $I_1 I_0 \overline{CC}$ | selector | Comentarios |
|-------------------------|----------|-------------------------|
| 00 * | 00 | Paso continuo |
| 01 0 | 11 | Salto Condicional |
| 01 1 | 00 | Paso continuo |
| 10 * | 01 | Salto de Transformación |
| 11 0 | 10 | Salto por interrupción |
| 11 1 | 00 | Paso Continuo |

Parte 1

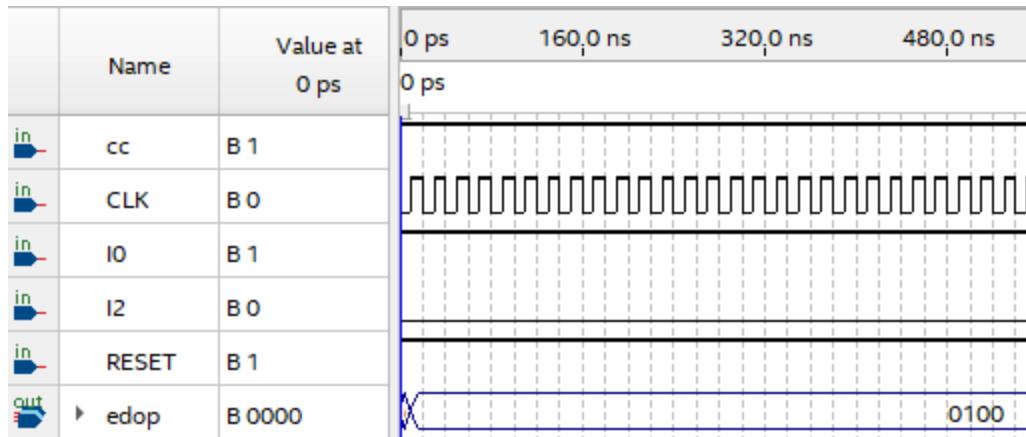
$I_1 = 0, I_0 = 0, CC = * - (\text{PASO CONTINUO})$



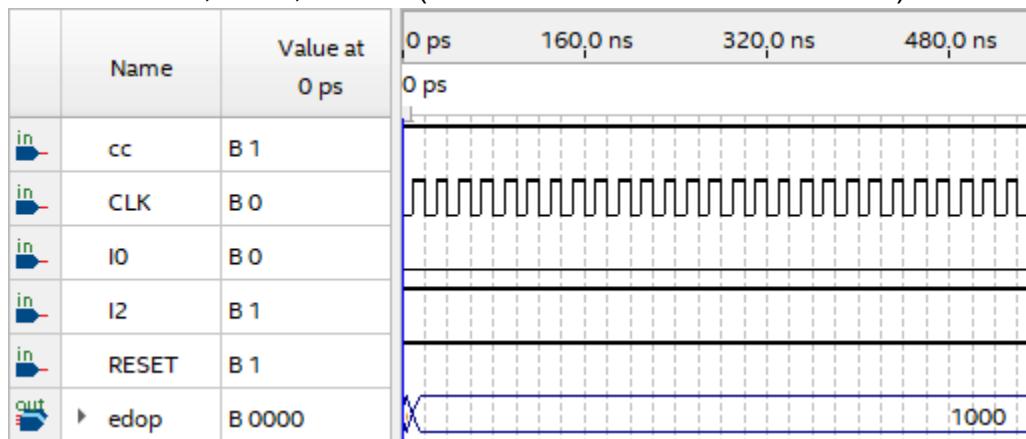
$I_1 = 0, I_0 = 1, CC = 0 - (\text{SALTO CONDICIONAL})$



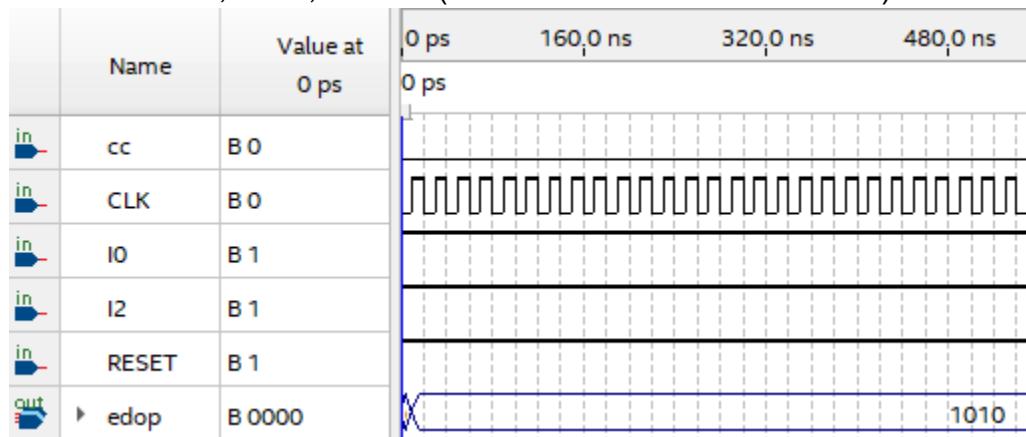
$I_1 = 0, I_0 = 1, CC = 1 - (\text{PASO CONTINUO})$



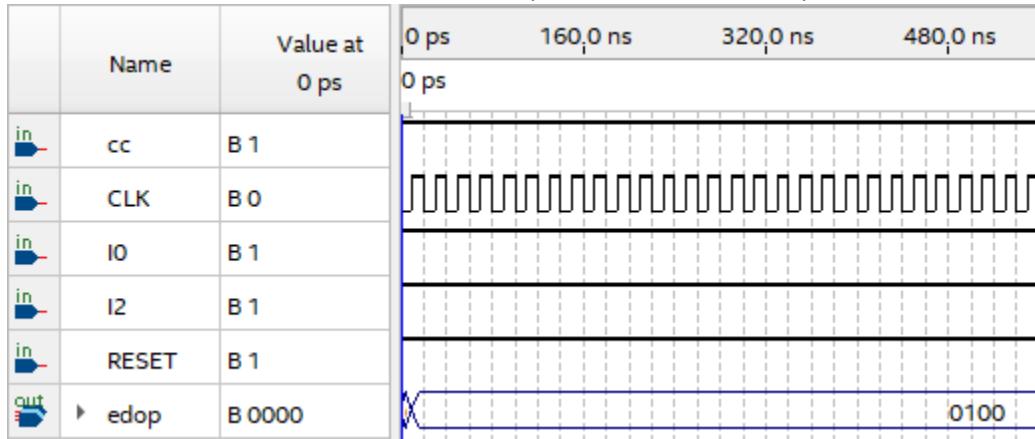
$I1 = 1, IO = 0, CC = 1$ - (SALTO DE TRANSFORMACIÓN)



$I1 = 1, IO = 1, CC = 0$ - (SALTÓ POR INTERRUPCIÓN)



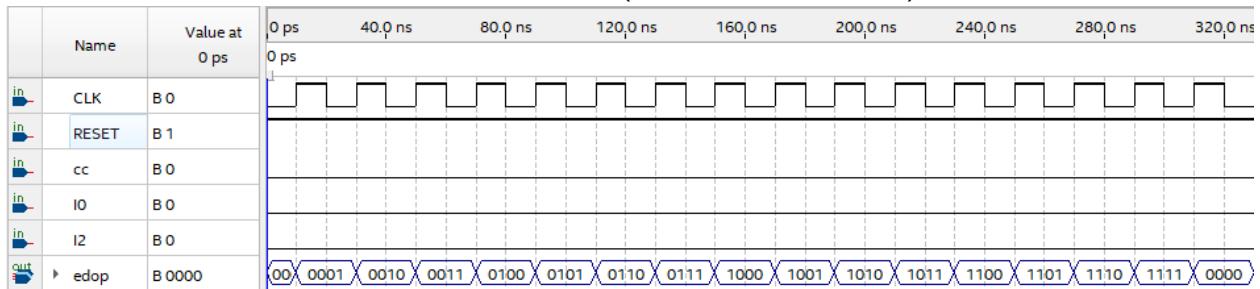
I1= 1, I0= 1, CC=1 - (PASO CONTINUO)



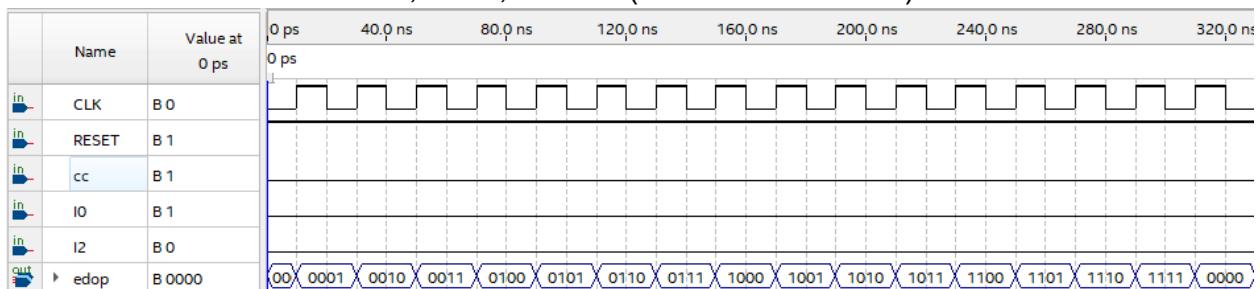
Parte 2

Verificamos para los casos de “PASO CONTINUO”, ya que las demás se mantienen

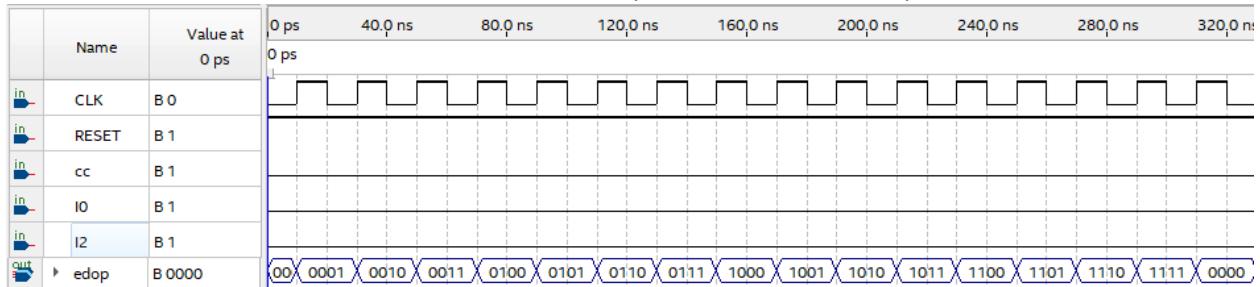
I1= 0, I0= 0, CC= * - (PASO CONTINUO)



I1= 0, I0= 1, CC=1 - (PASO CONTINUO)

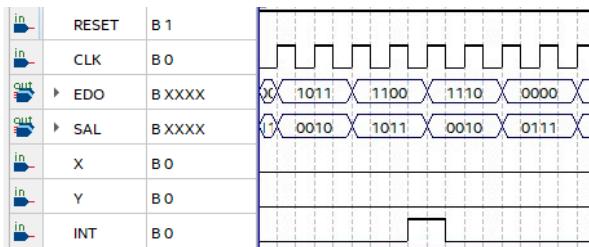
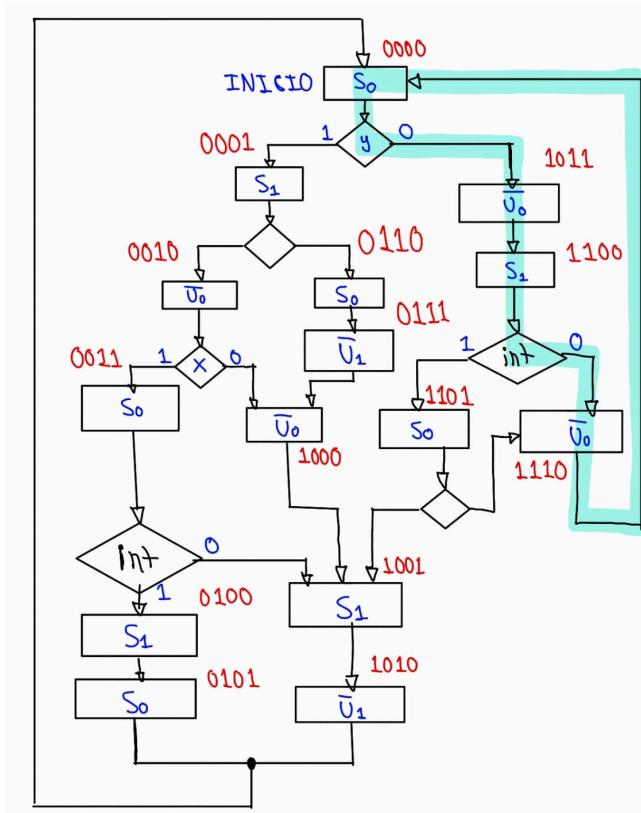


I1= 1, I0= 1, CC=1 - (PASO CONTINUO)

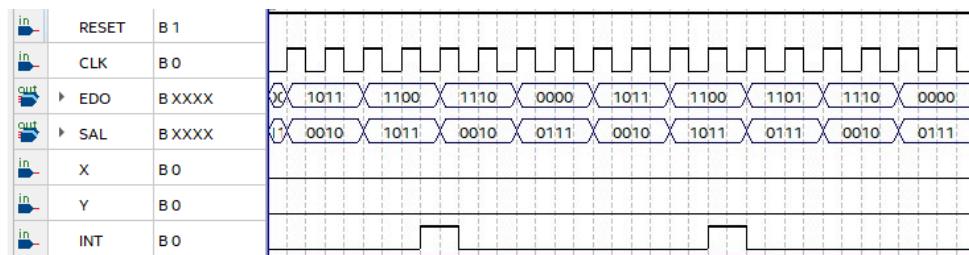
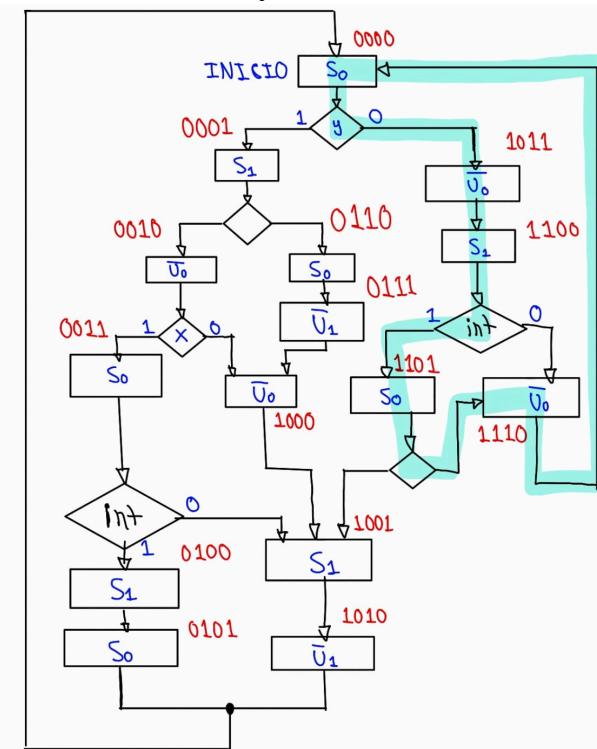


Parte 3

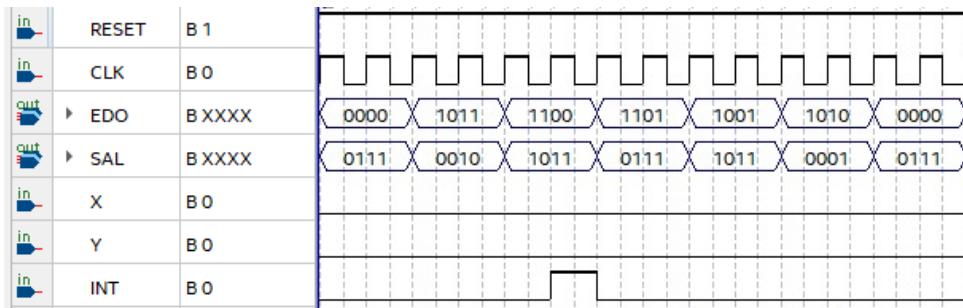
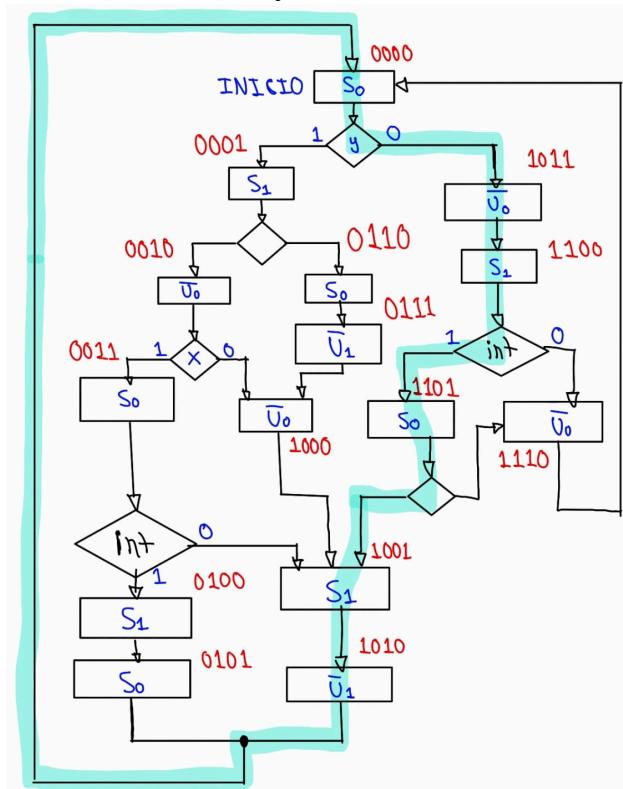
Trayectoria 1



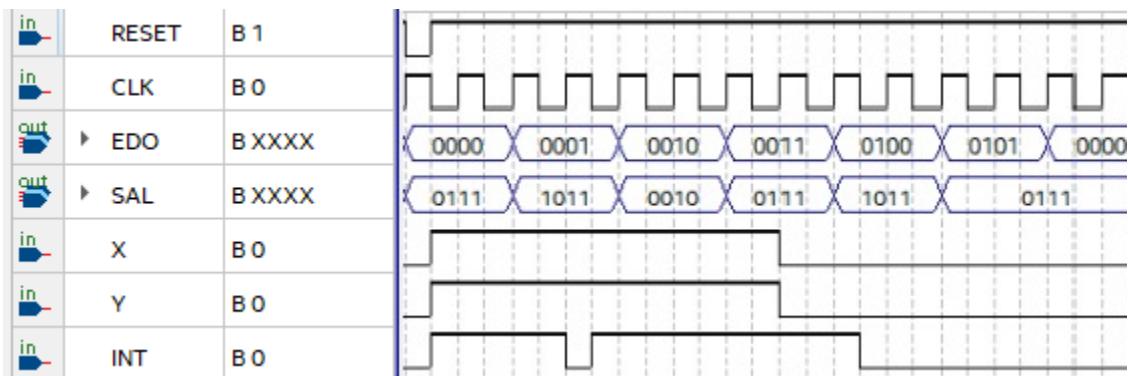
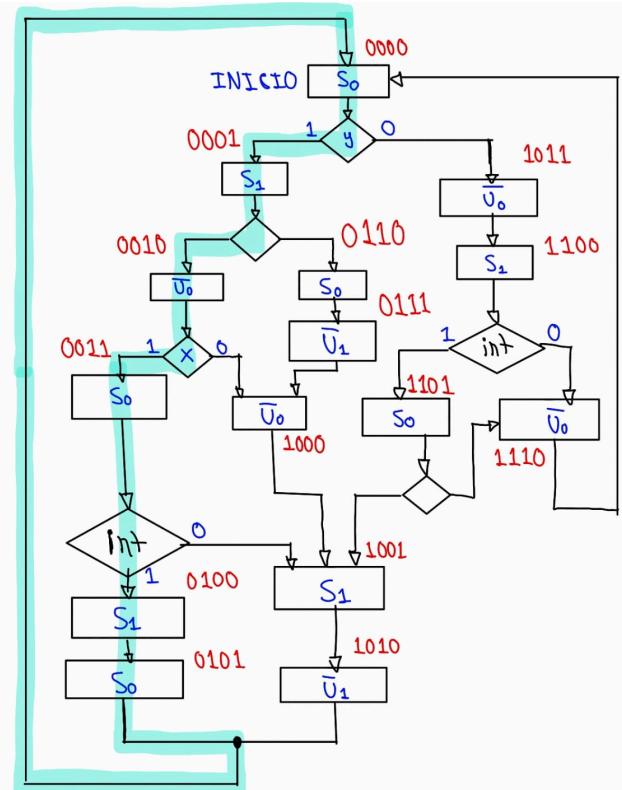
Trayectoria 2



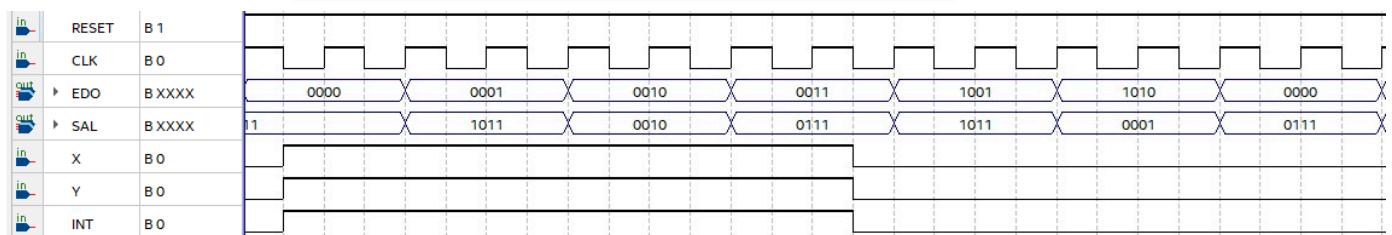
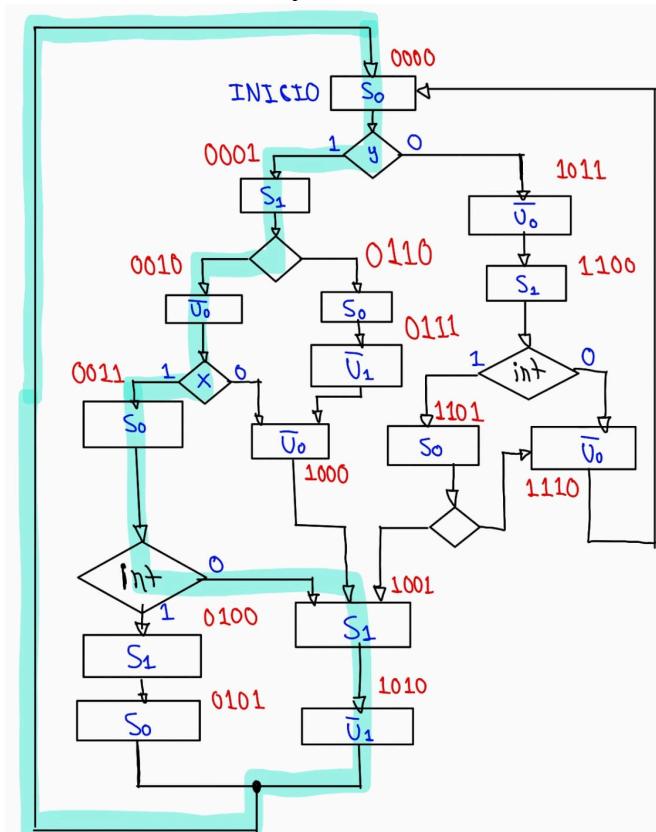
Trayectoria 3



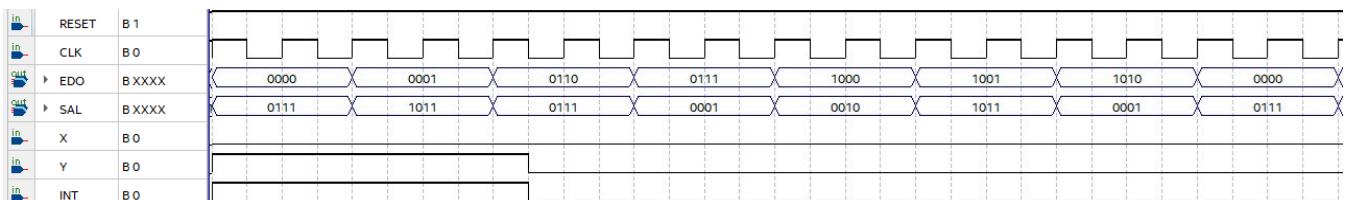
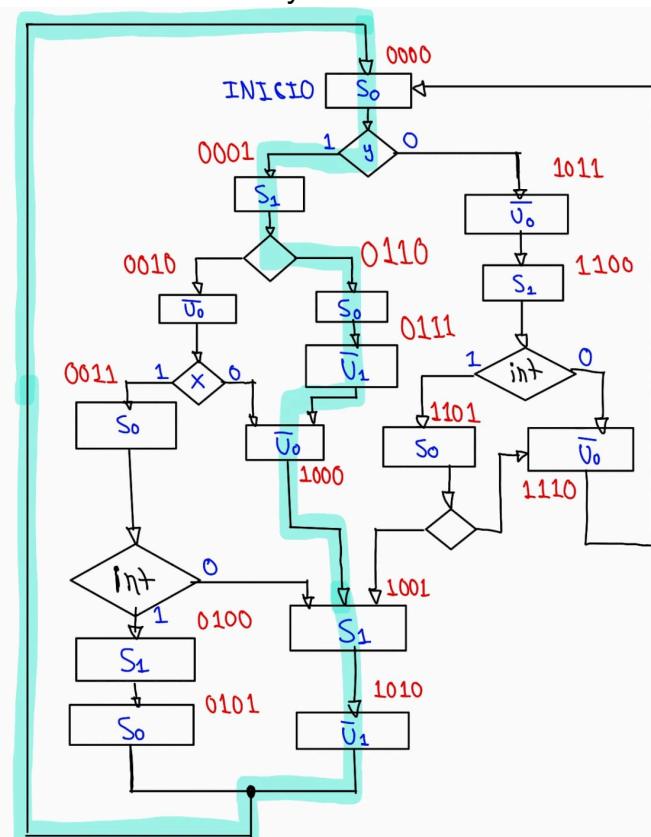
Trayectoria 4



Trayectoria 5



Trayectoria 6



Conclusiones

Jiménez Treviño Emilio Cristóbal

Durante el desarrollo de esta práctica, se logró implementar y comprender a fondo el diseño de un secuenciador básico de 4 instrucciones utilizando VHDL y el entorno Quartus. Se observó que la generación de la siguiente dirección de microinstrucción es un proceso dinámico regido por el multiplexor y la lógica interna, que decide el flujo (Paso Continuo, Salto Condicional, Salto de Transformación o Salto por Interrupción) basándose en las entradas de microinstrucción y la condición de prueba. En esta práctica se aprendió a cómo implementar cada bloque lo difícil fue la implementación del VWF que consistió en ir probando cada una de los caminos que va ir tomando nuestra carta ASM, debido a que nos costó trabajo debido a que en la entradas de reloj si no entraba cuando debía ya no hacía lo que debería, para solucionarlo fuimos observando el comportamiento y con prueba y error se logró solucionar.

Martinez Perez Brian Erik

La práctica ha sido útil para comprender la arquitectura del secuenciador básico, componente fundamental en la unidad de control de un procesador. logramos crear la integración de la lógica de siguiente estado con la memoria ROM, donde la dirección del microcódigo es generada dinámicamente. Se confirmó el mecanismo central de control: el Registro PC/Incrementador proporciona la dirección de Paso Continuo, mientras que el MUX Selector habilita el salto (a Liga, Interrupción o Transformación). La decodificación de las instrucciones (Continúa, Salto Condicional, etc.) se logró mediante la Lógica Interna, la cual dirige correctamente el flujo hacia el Registro PC. Esto demostró la capacidad del sistema para ejecutar secuencias y tomar decisiones basadas en condiciones de prueba. Finalmente, la implementación de la Carta ASM en el contenido de la ROM verificó el ciclo del diseño en VHDL.

Bibliografía

Laboratorio de Organización y Arquitectura de Computadoras. (2020, noviembre 3). *Práctica No. 6 Secuenciador básico*.