UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA
LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS
CLAVE 6867

Profesor: M. I. José Luis Cruz Mora

Integrantes del equipo:

García Fernández Jesús Alejandro

Hernández Arrieta Carlos Alberto

Moreno Guerra Marco Antonio

Reporte Práctica 3

Fecha de entrega: 11 de septiembre del 2019

Grupo: 1

Semestre: 2020-1

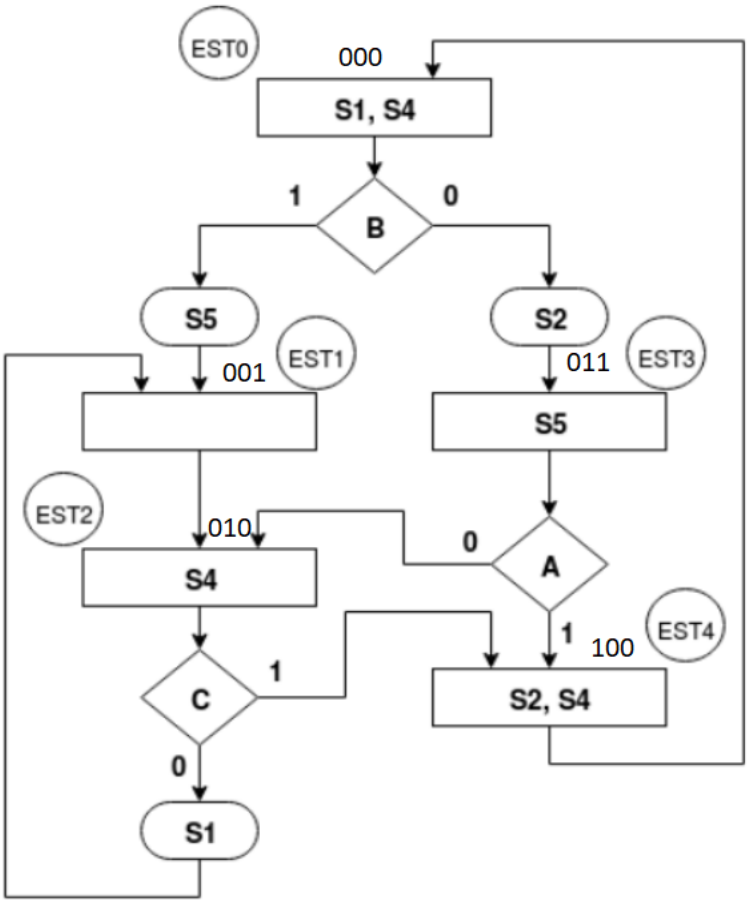
DIRECCIONAMIENTO POR TRAYECTORIA

OBJETIVO

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

DESARROLLO

A partir de la siguiente carta ASM se procedió a obtener la tabla del contenido de la memoria.

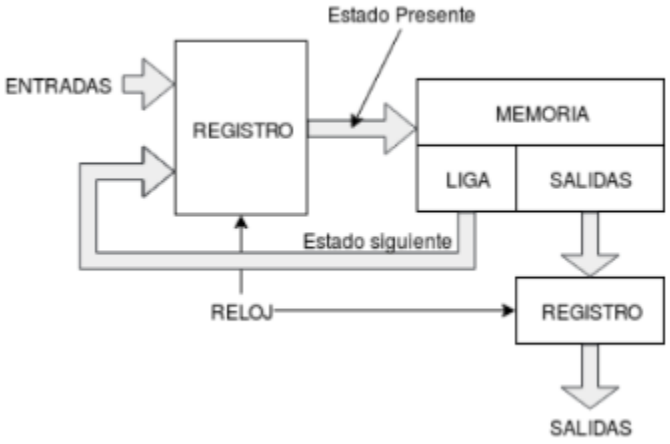


Carta ASM a resolver

Estado Presente	Entrada			Estado Siguiente			Salidas			
	A	B	C	LIGA			S1	S2	S4	S5
000	0	0	0	0	1	1	1	1	1	0
	0	0	1	0	1	1	1	1	1	0
	0	1	0	0	0	1	1	0	1	1
	0	1	1	0	0	1	1	0	1	1
	1	0	0	0	1	1	1	1	1	0
	1	0	1	0	1	1	1	1	1	0
	1	1	0	0	0	1	1	0	1	1
	1	1	1	0	0	1	1	0	1	1
001	*	*	*	0	1	0	0	0	0	0
010	*	*	0	0	0	1	1	0	1	0
			1	1	0	0	0	0	1	0
011	0	*	*	0	1	0	0	0	0	1
	1			1	0	0	0	0	0	1
100	*	*	*	0	0	0	0	1	1	0

Tabla del contenido de la memoria

Una vez obtenido el contenido de la memoria y tratándose del direccionamiento por trayectoria tenemos la siguiente arquitectura:



A partir de la arquitectura anterior y con ayuda de Quartus desarrollamos en VHDL cada uno de los componentes que conforman esta arquitectura de la siguiente manera:

Se tienen las Entradas a la máquina de Estados y la Liga resultante del contenido de memoria; ésta llega a un registro que le dará estabilidad y le permitirá convertirse en el estado presente. Esto se implementó de la siguiente manera:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity register2 is
    Port (  CLK : in STD_LOGIC;
            RESET : in STD_LOGIC;
            DATA_IN  : in STD_LOGIC_VECTOR(2 downto 0);
            DATA_OUT : out STD_LOGIC_VECTOR(2 downto 0));
end register2;
```

```
architecture Behavioral of register2 is
signal internal_value : std_logic_vector (2 downto 0) := B"000";
begin
    process (CLK, RESET, DATA_IN)
    begin
        if RESET = '0' then
            internal_value <= B"000";
        elsif rising_edge (CLK) then
            internal_value <= DATA_IN;
        end if;
    end process;
    process (internal_value)
    begin
        DATA_OUT <= internal_value;
    end process;
end Behavioral;
```

Como se puede observar, se le agrego una entrada Reset, ésta nos permitirá regresar al estado inicial cada que presionemos un botón.

La memoria se conforma del contenido obtenido en la tabla del contenido de memoria; en la que se determina cada uno de los casos posibles de las combinaciones de Entradas y Estados Presentes; por lo que el módulo de la memoria quedó de la siguiente manera:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity memory is
    Port( dir : in std_logic_vector(5 downto 0);
            data : out std_logic_vector(6 downto 0)
            );
end memory;

architecture Behavioral of memory is
    type mem is array(0 to 63) of std_logic_vector(6 downto 0);
    signal internal_mem : mem;
    begin
        -- ESTADO 0
        internal_mem(0) <= "011" & "1110";
        internal_mem(1) <= "011" & "1110";
```

```
internal_mem(2) <= "001" & "1011";
internal_mem(3) <= "001" & "1011";
internal_mem(4) <= "011" & "1110";
internal_mem(5) <= "011" & "1110";
internal_mem(6) <= "001" & "1011";
internal_mem(7) <= "001" & "1011";

-- ESTADO 1
internal_mem(8)  <= "010" & "0000";
internal_mem(9)  <= "010" & "0000";
internal_mem(10) <= "010" & "0000";
internal_mem(11) <= "010" & "0000";
internal_mem(12) <= "010" & "0000";
internal_mem(13) <= "010" & "0000";
internal_mem(14) <= "010" & "0000";
internal_mem(15) <= "010" & "0000";

-- ESTADO 2
internal_mem(16) <= "001" & "1010";
internal_mem(17) <= "100" & "0010";
internal_mem(18) <= "001" & "1010";
internal_mem(19) <= "100" & "0010";
internal_mem(20) <= "001" & "1010";
internal_mem(21) <= "100" & "0010";
internal_mem(22) <= "001" & "1010";
internal_mem(23) <= "100" & "0010";
-- ESTADO 3
internal_mem(24) <= "010" & "0001";
internal_mem(25) <= "010" & "0001";
internal_mem(26) <= "010" & "0001";
internal_mem(27) <= "010" & "0001";
internal_mem(28) <= "100" & "0001";
internal_mem(29) <= "100" & "0001";
internal_mem(30) <= "100" & "0001";
internal_mem(31) <= "100" & "0001";
-- ESTADO 4
internal_mem(32) <= "000" & "0110";
internal_mem(33) <= "000" & "0110";
internal_mem(34) <= "000" & "0110";
internal_mem(35) <= "000" & "0110";
internal_mem(36) <= "000" & "0110";
```

```
        internal_mem(37) <= "000" & "0110";
        internal_mem(38) <= "000" & "0110";
        internal_mem(39) <= "000" & "0110";

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

Posteriormente de esta memoria obtenemos la liga (Estado Siguiente) y las salidas. Sin embargo, estos datos son proporcionados de la salida del módulo anterior de memoria, y éstos vienen en in vector de 7 bits; por lo tanto, tendremos que separar los datos para poder procesar cada parte de forma independiente. Realizando el siguiente código que nos regresa la liga y las salidas por separado:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity divisor_datos is
    Port ( entrada : in STD_LOGIC_VECTOR (6 downto 0);
             liga : out STD_LOGIC_VECTOR (2 downto 0);
             salidas : out STD_LOGIC_VECTOR(3 downto 0));
end divisor_datos;
architecture Behavioral of divisor_datos is
begin
    process(entrada)
    begin

        liga <= entrada(6 downto 4);
        salidas <= entrada(3 downto 0);
    end process;
end Behavioral;
```

Por otra parte, se necesita que para el módulo del Registro inicial, que provee el Estado Presente se proporcione la Liga (resultante de la memoria) con las Entradas de la máquina. Por ello se requiere un módulo que concatene las variables de Entradas y la Liga. Este módulo se describe con el siguiente código en VHDL

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

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity concatenador_datos is
    Port ( entradas : in STD_LOGIC_VECTOR (2 downto 0); -- Entradas
           liga : in STD_LOGIC_VECTOR (2 downto 0); -- Liga
           salida : out STD_LOGIC_VECTOR(5 downto 0));
end concatenador_datos;
architecture Behavioral of concatenador_datos is
begin
    process(entradas, liga)
    begin
        salida <= liga & entradas; -- Liga & Entradas
    end process;
end Behavioral;
```

El funcionamiento de la arquitectura como tal está completa hasta aquí, pero con el objetivo de poder probar su correcto funcionamiento en la FPGA DE-10 Lite, se configuraron dos módulos más: el primero de ellos, mismo que fue utilizado en la práctica pasada, permite sustituir los flancos automáticos del reloj de la tarjeta por flancos de subida cada que presionemos un botón, lo cual nos permitirá apreciar de una manera mucho más sencilla cómo se va avanzando a través de los estados y poder manipular con mayor facilidad las entradas sin que salte a otro estado por un Clock automático y de periodo corto. Este módulo es *sensa_boton*:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity sensa_boton is
    Port (BOTON : in STD_LOGIC;
        CLK : in STD_LOGIC;
        RELOJ : out STD_LOGIC;
        EPRESENTE: buffer STD_LOGIC);
end sensa_boton;
architecture Behavioral of sensa_boton is
    signal ESIGUIENTE: STD_LOGIC;
        begin
        process (ESIGUIENTE, BOTON)
            begin

            if rising_edge (CLK) then
                case ESIGUIENTE is
```

```
                    when '0' =>
                        RELOJ <= '0';
                        if BOTON ='0' then
                            ESIGUIENTE <= '0';
                        else
                            ESIGUIENTE <= '1';
                        end if;
                    when '1' =>
                        if BOTON ='1' then
                            ESIGUIENTE <= '1';
                            RELOJ <= '0';
                        else
                            ESIGUIENTE <= '0';
                            RELOJ <= '1';
                        end if;
                    when others => null;
                end case;
            end if;
            EPRESENTE <= ESIGUIENTE;
end process;
end Behavioral;
```

Por último, agregamos un módulo que nos permite visualizar en los LEDs de la FPGA dos diferentes casos dependiendo la posición de un switch; en el primer caso podemos ver el Estado Presente con las Salidas y en el segundo caso el Estado Siguiente y las Salidas. Para ello, se requiere que este módulo *switchSalidas*, que se encargará de determinar qué bits de información se mandarán a la interfaz de salida, se requieren los siguientes datos de entrada:

- Contenido de la memoria (7 bits).
- Estado Presente.
- Botón Switch.
- Las salidas

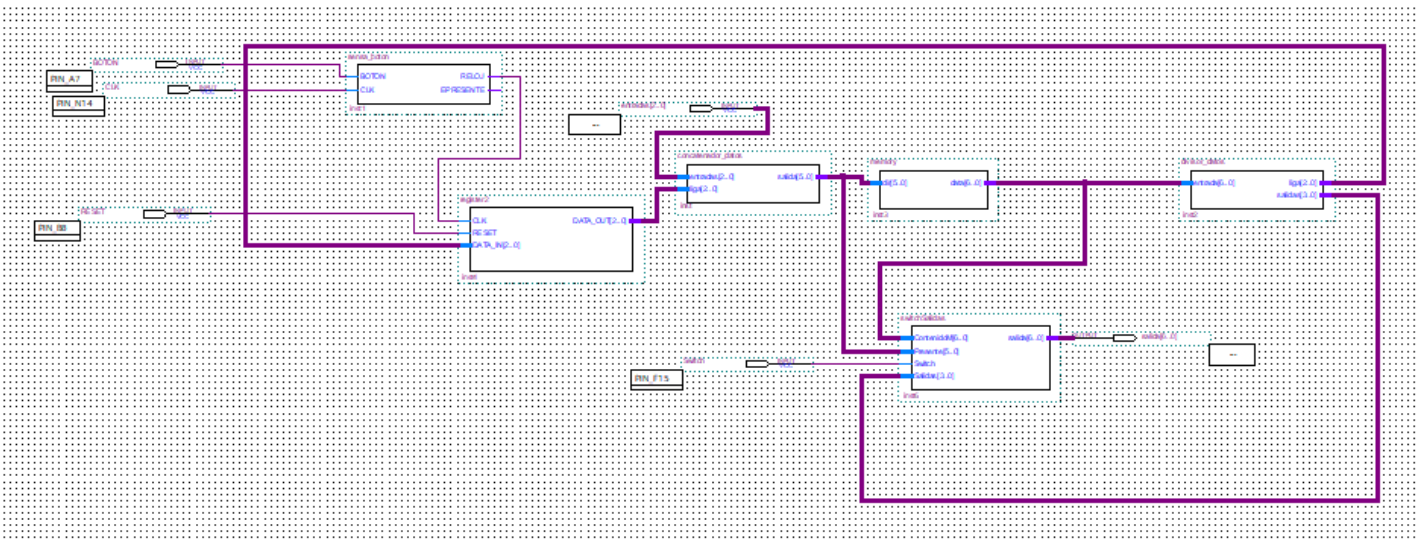
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity switchSalidas is
    Port ( ContenidoM : in STD_LOGIC_VECTOR (6 downto 0); -- Entradas
           Presente : in STD_LOGIC_VECTOR (5 downto 0); -- Liga
```

```
            Switch: in std_logic;
            Salidas: in STD_LOGIC_VECTOR (3 downto 0);
            salida : out STD_LOGIC_VECTOR(6 downto 0));
end switchSalidas;
architecture Behavioral of switchSalidas is
begin
    process(ContenidoM, Presente, Switch, Salidas)
    begin
        if Switch = '0' then
            salida <= ContenidoM;
        else
            salida <= Presente(5 downto 3) & Salidas;
        end if;
    end process;
end Behavioral;
```

Todos los módulos anteriores descritos para solucionar la Carta ASM se desarrollaron con VHDL. Sin embargo, cada módulo está aislado del comportamiento del resto de componentes, por lo que es necesario relacionarlos entre sí para poder generar el Sistema completo.

Una manera de relacionarlos es, de igual manera, generando un archivo de Código VHDL y describir instancias de los módulos para, posteriormente, conectarlos. Mas en esta ocasión se optó por una forma más sencilla, que consiste en generar un símbolo de cada uno de los archivos VHDL.

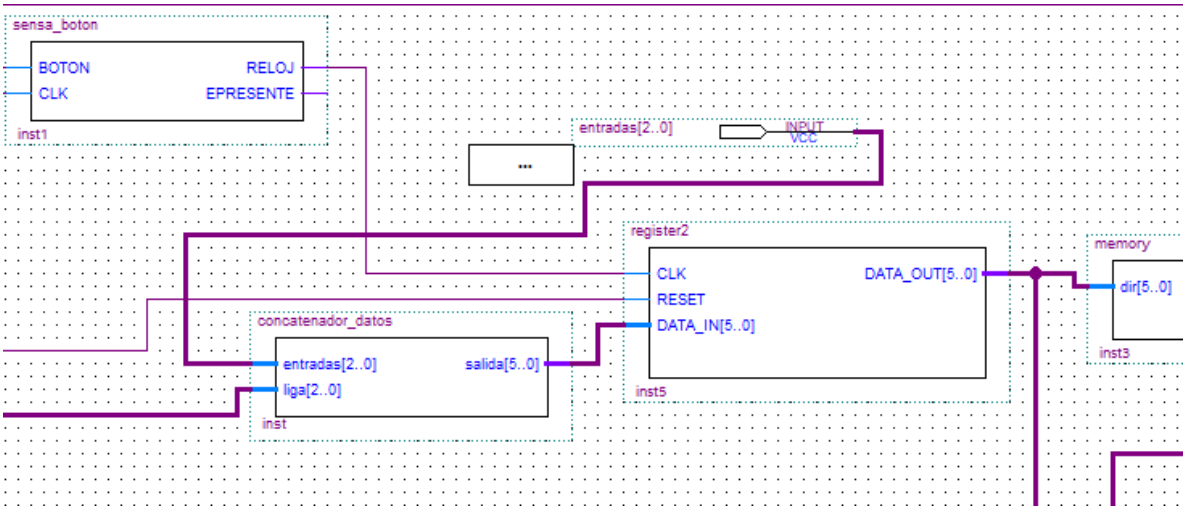
De esta manera se creó un archivo de Diagrama de Bloques y se agregaron los símbolos para conectarlos de la siguiente manera:



Para verificar que los cambios realizados en el Proyecto funcionan de manera correcta, se realice la compilación del Proyecto en Quartus, la cual fue de manera exitosa. Posteriormente, se grabó el Proyecto en la FPGA y se verificó que funcionaba de manera correcta.

Sin embargo, fue necesario realizar otro cambio, pues la máquina de Estados funcionaba de manera correcta, pero cuando se encontraba en el Estado Inicial, siempre tomaba el camino de la carta ASM cuando la variable B es cero, independientemente del valor que se le diera al PIN asignado a esa variable.

Para solucionar el error anterior, se realice un cambio en el orden de los módulos de *concatenar_datos* y *register2*, ya que inicialmente estaba así:

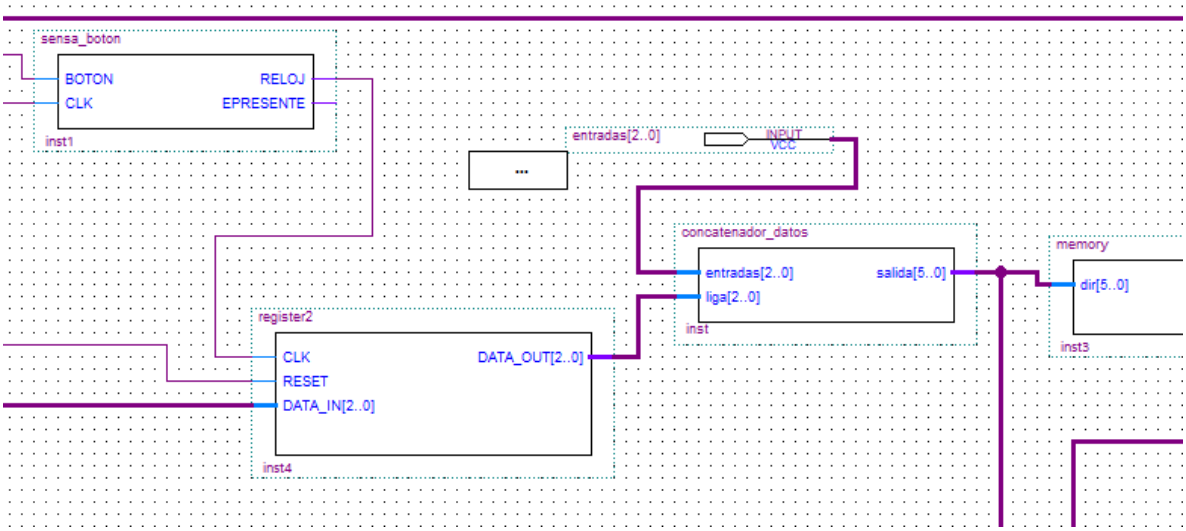


Con este orden lo que sucedía era que al establecer el RESET o estar en el Estado Inicial, se colocaba en el vector de salida de *register2* en ceros, es decir, tanto las Entradas y el Estado lo ponía en ceros, por lo que no consideraba los cambios efectuados en las Entradas.

Para solucionar lo anterior, se cambió colocó primero el módulo *register2*¸que ahora solamente recibe las entradas: CLK, RESET y un vector de 3 bits que corresponden al Estado Presente y se encargará de hacerle RESET cuando sea necesario.

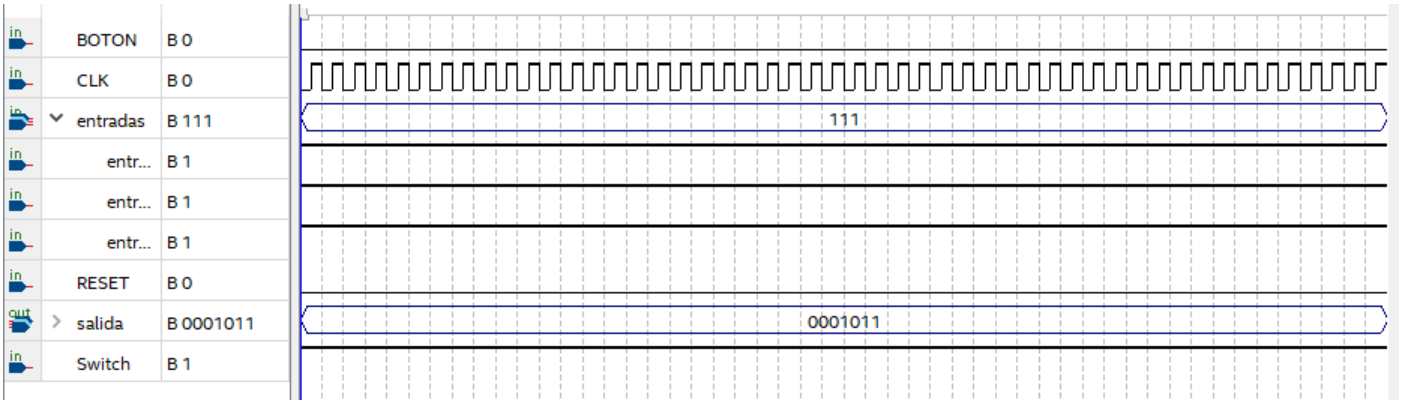
La salida de *register2* se proporciona a la entrada de *concatenar_datos*, que hará su unión junto con los 3 bits de las Entradas (que son combinacionales); de esta manera, el RESET ahora no afectará a las Entradas. Además, éstas ahora serán tomadas en cuenta desde el Estado Inicial, con lo que al efectuar un cambio en la variable B sin presionar el botón de Clock Manual de *sensa_boton,* se reflejará un cambio en los LEDs de Estado Siguiente y en las Salidas (debido a que se tienen salidas condicionales).

Dicho cambio quedó de la siguiente manera:



Finalmente, se compiló y se grabó el Proyecto en la FPGA para confirmar que el funcionamiento es el deseado y se solucionó el problema.

SIMULACIÓN



En la simulación podemos comprobar que si la entrada es 111, como nunca cambiamos la variable BOTON, no pasa al siguiente ciclo de reloj, y vemos como la salida en el estado actual es 000 y su salida son S_1,S_4,S_5, por lo que si vemos la carta ASM, comprobamos que la salida es correcta.

CONCLUSIONES

García Fernández Jesús Alejandro: En esta práctica implementamos la primera forma de direccionamiento vistas en la clase de teoría, ahí aprendí que este modo de direccionamiento puede resolver cualquier problema que se pueda modelar por medio de una carta ASM, realizando este direccionamiento de forma práctica me enfrente a varias dudas que no me podía enfrentar en teoría, al momento de pasar el programa a la tarjeta nos dimos cuenta que automáticamente el comportamiento era irse por el camino donde la variable condicional valía 0, al principio pensamos que era un error en la tabla de direccionamiento, pero con ayuda del profesor, logramos analizar y darnos cuenta que al momento de iniciar se realizaba un reset de todos los datos concatenados por lo cuál el comportamiento obtenido no era el que queríamos, llegamos al final a la conclusión de hacer el reset antes de hacer la concatenación de la liga con las entradas para que este reset afectara solamente a la liga o estado y no a las entradas y que ahora si el salto de los estados dependiera directamente de las entradas ingresadas por medio de el switch correspondiente de la tarjeta.

Moreno Guerra Marco Antonio: Con la realización de esta práctica, se reafirmó el conocimiento de las dos prácticas anteriores, es decir, el uso del software Quartus y la capacidad de implementar la solución de una Carta ASM en este software. Además, esta ocasión aprendí cómo implementar una solución de una Carta ASM pero empleando el método de direccionamiento de memoria llamado *Trayectoria*, para el cual hubo cambios significativos respecto al método tradicional, pues en éste se debe de programar el contenido de la memoria que se requiere para el funcionamiento del sistema, así como del resto de módulos para completar la arquitectura.

Además, al realizar esta práctica, se puso a prueba el conocimiento teórico que tengo sobre esto, con lo que pude comprender mejor su funcionamiento, así como de pequeños detalles que tuvimos que justar pues el funcionamiento era el deseado, pero no el esperado, ya que debido a que primero se hacia la concatenación de las Entradas con el Estado Presente, el módulo de *register2* hacía un RESET de todos esos bits de Entradas y del Estado. Por ello, analizamos y propusimos, en primera instancia, el modificar la cantidad de bits que hacía el RESET, para no afectar los bits de las Entradas. Funcionó, pero se modificó el funcionamiento interno de ese registro, lo cual no fue correcto. Así que se modificó el orden, es decir, primero hacer uso de *register2* para operar únicamente con el Estado Presente, y después de concatena con las Entradas, que operan de manera combinacional.

Hernández Arrieta Carlos Alberto: Con la realización de esta práctica vemos lo visto en teoría que es el Direccionamiento por Trayectoria, este direccionamiento nos permite trabajar con cartas ASM con salidas condicionales. Vemos el llenado de la memoria usando este direccionamiento, realizamos un diagrama de bloques para emular el funcionamiento de este direccionamiento en la tarjeta. Programamos una memoria que nos permite obtener cuál será el estado siguiente, así como también la salida sabiendo las entradas y el estado actual.

El único problema durante la realización de la práctica fue que al momento de inicializar el funcionamiento dentro de la tarjeta, siempre se iba hacía 0 cuando B dadas la entradas valía 1, por lo que necesario realizar modificaciones sobre lo que originalmente teníamos, sólo consistió en poner primero el registro antes que el concatenador de datos, con ello se logra que tome las entradas al realizar la concatenación ya que teniéndolos al revés el RESET afectaba las entradas ya que todo lo ponía en cero, pero haciendo ese pequeño cambio se logró el funcionamiento esperado.

Por último, hicimos uso de los conocimientos de prácticas anteriores para poder realizar está práctica como hacer uso de sensa_boton para controlar mediante un push del botón ver el siguiente estado y sus salidas.