UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Ingenieria

División de Ingeniería Eléctrica

Reporte de Práctica 04

Construción de Máquinas de estados Usando Memorias Direccionamiento Entrada - Estado

Grupo 05

Grupo de Teoría 02

Semestre 2022-2

LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

Profesor: Ing. Julio Cesar Cruz Estrada

Integrantes:
Vivanco Quintanar, Diego Armando

18 de marzo de 2022

1. Objetivo

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

2. Duración

- 1 semana.

3. Introducción

El direccionamiento entrada-estado se restringe a cartas ASM con una sola entrada por estado. Una nueva porción de la palabra de memoria contiene una representación binaria de la entrada a probar en cada estado, esta parte es llamada prueba. Con esta representación binaria un selector de entrada elige una de las variables de entrada

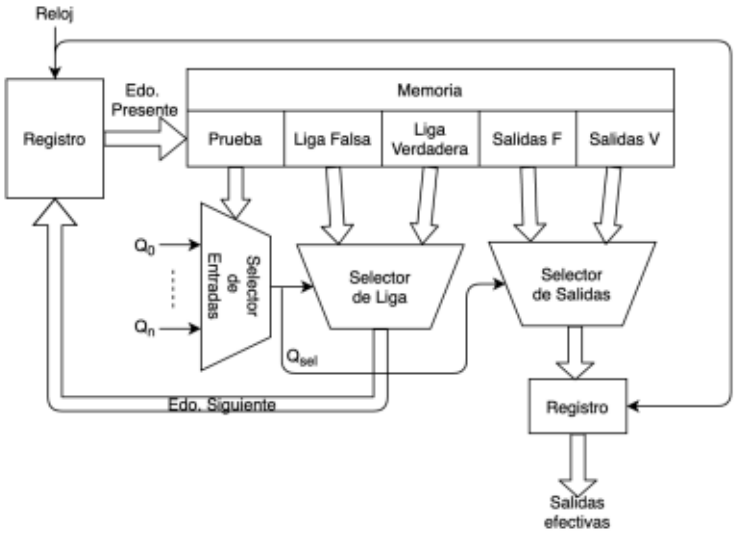

Figura 1: Arquitectura del Direccionamiento Entrada-Estado modificada para soportar salidas condicionales.

Cálculo del tamaño de la memoria ROM:

$$n = \text{bits de los estados}$$

$$m = \text{bits de prueba} + \text{bits de liga falsa} + \text{bits de liga verdadera} + \text{bits de salidas falsas} + \text{bits de salidas verdaderas}$$

$$\text{Tamaño de memoria} = 2^n * m$$

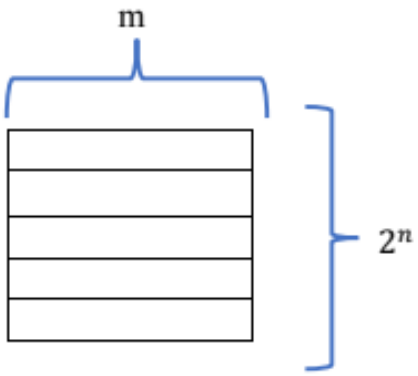

Figura 2: Memoria ROM.

Características del direccionamiento Entrada - Estado.

1. No soporta cualquier carta ASM.
2. Soporta salidas condicionales.
3. Se puede evaluar más de una condición en un estado.
4. Se desperdicia mucha memoria.
5. Elementos de Hardware.
    a Memoria ROM.
    b Registro.

4. Desarrollo

4.1. Actividad 1

Dada la carta ASM de la figura 3, encuentre el contenido de memoria utilizando el direccionamiento entrada-estado. Recuerde que antes de construir la tabla se debe asignar a cada estado de la carta ASM una representación binaria. Así mismo, no olvide asignar una representación binaria a las entradas

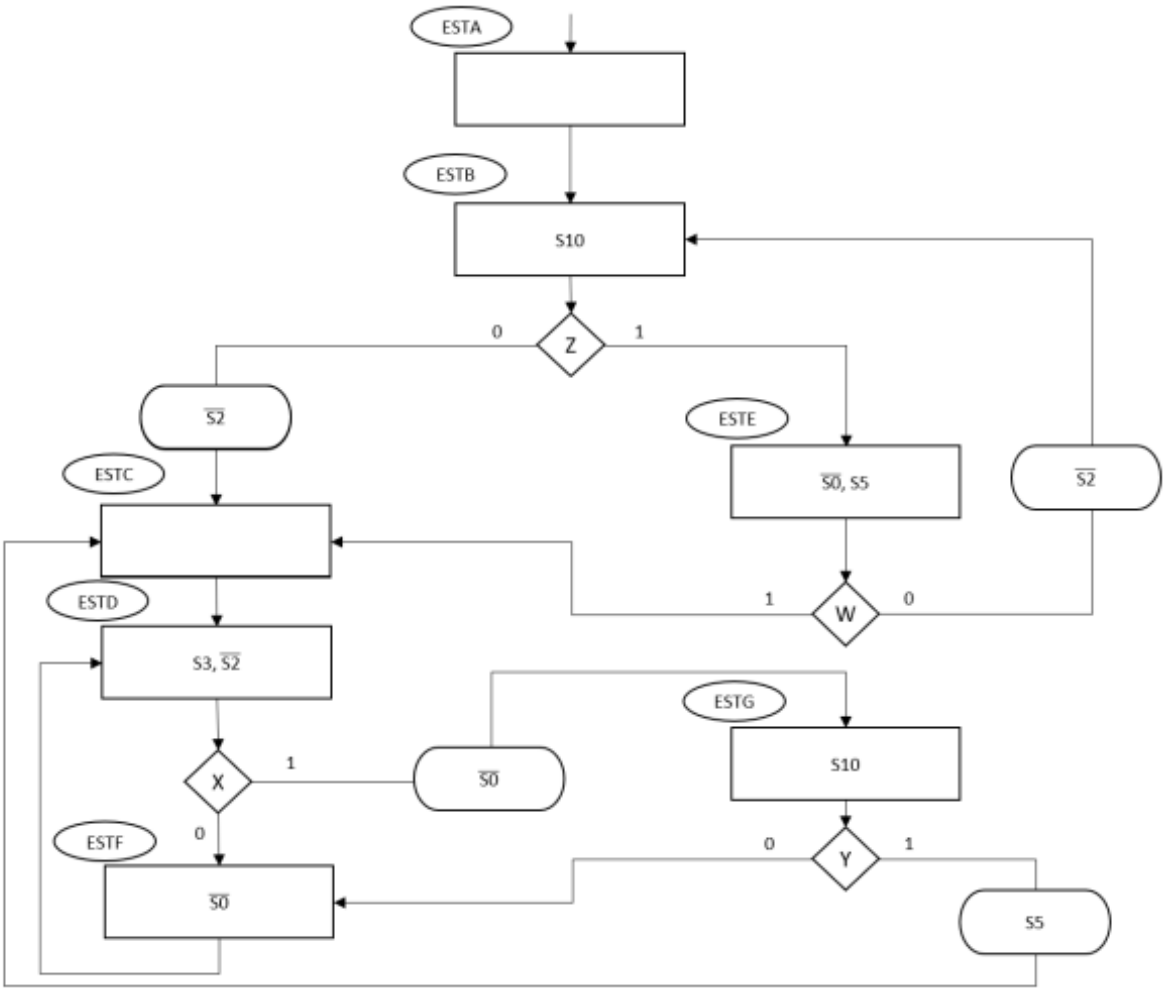

Figura 3: Carta ASM.

4.1.1. Cálculo del tamaño de la memoria ROM

Como se estan usando 7 estados, se requieren de 2^3 estados, es decir que neceistamos 3 bits para representar 8 estados, de los cuales solo 7 serán ocupados.
Considerando que tenemos 4 entradas, se ucupan 2 bits de prueba, los bits de liga son del mismo tamañao que los bits que se requieren para la asignación binaria de los estados (3 bits), como tenemos 5 salidas en la carta ASM se requieren 5 bits, como empleamos la arquitectura del direccionamiento Entrada-Estado modificado, se requieren tanto una liga falsa como una verdadera y se requieren de dos salidas, una verdadera y una falsa con lo que la carta ASM puede soportar saldias condicionales.

$$n=4$$

En este caso el número de bits empleados para los estados es el mismo para la liga y considerando que son 4 salidas las que se ocupan, se requieren de 4 bits más.

$$m=2+3+3+5+5=18$$

Finalmente:

$$\text{Tamaño de memoria} = 2^3 * 18 \text{ bits.}$$

4.1.2. Asignación binaria de los estados

Literal	Estado	Valor Binario
A	0	000
B	1	001
C	2	010
D	3	011
E	4	100
F	5	101
G	6	110
*	7	111

4.1.3. Valores de Prueba

Entrada	Prueba
W	00
X	01
Y	10
Z	11

4.1.4. Construir la tabla con el contenido de la memoria

A continuación se muestra la tabla con la dirección y el contenido de la memeria con el direccionamiento Entrada-Estado.

Estado Presente			Prueba		Liga Falsa			Liga Verdadera			Salidas Falsas					Salidas Verdaderas				
P2	P1	P0	K1	K0	F2	F1	F0	V2	V1	V0	S10	S5	S3	S2	S0	S10	S5	S3	S2	S0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	1	1
0	0	1	1	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	1
0	1	0	0	0	0	1	1	0	1	1	0	0	0	1	1	0	0	0	1	1
0	1	1	0	1	1	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0
1	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1	0
1	0	1	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	0	1	0
1	1	0	1	0	1	0	1	0	1	0	1	0	0	1	1	1	1	0	1	1
1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1

4.2. Actividad 2

Una vez que haya obtenido el contenido de memoria, implemente el direccionamiento de entrada-estado utilizando el software de desarrollo Quartus y utilize el contenido de memoria obtenido.

Los bloques creados para la implementación del direccionamiento Entrada-Estado en VHDL son los siguientes:

- Divisor de Frecuencia
- Memoria ROM
- Registro
- Multiplexor de Entradas
- Multiplexor de Salidas
- Multiplexor de Ligas
- Divisor de Datos

En la figura 4 podemos ver el esquematico de nuestra arquitectura implementada en Quartus, se observa a detalle la interconexón de nuestros bloques enlistados anteriormente.

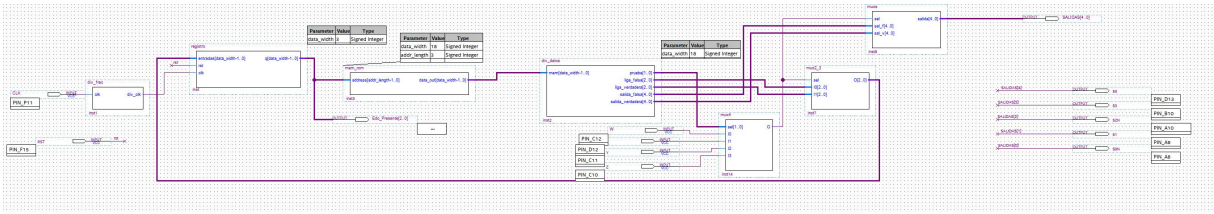

Figura 4: Esquematico de la Arquitectura del Direccionamiento Entrada-Estado modificado.

4.2.1. Divisor de Frecuencia

Dado que las condiciones nos han permitido regresar a la modalidad de clases presenciales ya no solo debemos simular el funcionamiento de nuestros programas si no que ahora podemos implementarlas de manera fisica a traves de tarjetas, en mi caso utilizo la tarjeta DE10lite. Para que nuestros programas se ejecuten correctamente en la tarjeta debemos usar un divisor de frecuencia, que utilice como entrada la frecuencia que otorga la tarjeta y transforme esa frecuencia en aproximadamente un lapso de un 1 segundo para ver el cambio en los estados al recorrer la carta ASM.

```
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(clk: in std_logic;
       div_clk: out std_logic);
end div_frec;
architecture Behavioral of div_frec is
begin
  process (clk)
    variable cuenta: std_logic_vector(27 downto 0):=x"0000000";
    begin
    if rising_edge(clk) then
      if cuenta = x"3FFFFFF" then
        cuenta := x"0000000";
      else
        cuenta := cuenta+1;
      end if;
    end if;
    div_clk <= cuenta(25);
  end process;
end Behavioral;
```

Listing 1: Implementación de un divisor de frecuencia en VHDL.

En la entidad del código podemos ver que se usan dos señales, la primera es la entrada llamda clk, la cual tomara la frecuencia de la tarjeta, dicha señal es de un bit, la señal de salida llamada div_clk, tomara el valor de la frecuencia obtenida del dividor y que será suministrada a nuestra arquitectura. Posteriormente tenemos un proceso, en el que nos apoyamos de una variable llamada cuenta, la cual nos permite hacer un conteo de los ciclos de la señal de la tarjeta, con el fin de ir reduciendo el ciclo de trabajo y obtener la frecuencia que deseamos, esto se logra con un if, en donde valdiamos que la cuenta ha llegado al valor Hexadecimal X'3FFFFFF', de ser el caso el conteo se reinicia a 0. En caso contrario, cuenta se incrementa en uno, al finalizar este proceso el valor de cuenta se pasa a la señal de salida div_clk.

4.2.2. Memoria ROM

A continuacion se muestra el código que nos ayuda a crear una memoria ROM, en la definicion de la entidad, declaramos dos varibales como genericas, siendo estas las que nos indican los bits que se necesitan tanto las direcciones de memoria como el tamaño del contenido de la memoria. Finalmente declaramos las entradas y las salidas del bloque de la memoria, siendo las entradas los bits que nos apuntan a las direcciones de la memoria (address) y como salida tenemos los bits de las salidas (data out).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mem_rom is
  generic(data_width : natural :=18;  --Contenido de la memoria
        addr_length : natural :=3 --Direccion de memoria
        );
  port(
      --clk: in std_logic; --Reloj
      address: in std_logic_vector(addr_length-1 downto 0); --Edo. Presente
      data_out: out std_logic_vector(data_width-1 downto 0)  --Edo.
         Siguiente
     );
end mem_rom;

architecture arch_rom of mem_rom is
  constant mem_size : natural := 2**addr_length;
  type mem_type is array(mem_size-1 downto 0) of std_logic_vector(
     data_width-1 downto 0);
  constant mem : mem_type :=
    (
      -- Contenido de la memoria
      0 => b"000010010001100011", 1 =>b"110101001000110011",
      2 => b"000110110001100011", 3 =>b"011011100010100100",
      4 => b"000010100100001010", 5 =>b"000110100001000010",
      6 => b"101010101001111011", 7 =>b"000000000001100011"
    );

begin
  process (address)
  begin
      data_out <= mem(to_integer (unsigned(address)));
  end process;
end architecture arch_rom;
```

Listing 2: Implementación de una memoria ROM con las direcciones y contenido de memoria del direccionamiento Entrada-Estado.

Posteriormente podemos ver la estrutura de la arquitectura de la memoria en donde se calcula el tamano de la memoria, para definir una variable de tipo array a la que le asociamos el vector del contenido de la memoria, finalmente asociamos lo anterior a una variable de tipo constante para cargar el contenido de la memoria obtenido en la actividad anterior.
En seguida, colocamos las direcciones de la memoria que apuntan al contenido de la memoria obtenido en la activiad anterior. Una vez que hemos definido el contenido de la memoria, procedemos a crear el proce- so de direccionamiento, para ello utilizamos el reloj, por cada flanco de subida, estare- mos recorriendo las direcciones de memoria mediante el casteo de la entrada (address).

4.2.3. Registro

En el siguiente bloque de codigo podemos ver la implementaci on de un registro en VHDL, como primer punto colocamos las bibliotecas basicas, posteriormente declaramos una variable generica llamada data width la cual definira el numero de bits para la salida, posteriormente definimos las entradas y las salidas del registro. Tenemos en las entradas el reset, el reloj (clk), entradas(bits de la liga) y como salida tenemos lo que es el estado presente.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity registro is
  generic(
    data_width : natural :=3
  );
  port(
    q : out std_logic_vector(data_width-1 downto 0);
    entradas: in std_logic_vector(data_width-1 downto 0);
    rst: in std_logic;
    clk: in std_logic
  );
end entity registro;

architecture arch_reg of registro is
signal estado : std_logic_vector(data_width-1 downto 0) := std_logic_vector
   (to_unsigned(0,entradas'length));
begin
  process (clk, rst)
  begin
    if rst = '1' then
      estado <= std_logic_vector(to_unsigned(0,q'length));
    elsif rising_edge(clk) then
      estado <= entradas;
    end if;
  end process;
  process (estado)
  begin
    q <= estado;
  end process;

end architecture arch_reg;
```

Listing 3: Implementación de un registro en VHDL.

Dentro de la arquitectura tenemos un proceso, el cual verifica el estado del reset, si esta activado el registro manda a su salida el estado 0 y las salidas de la carta ASM desactivadas. Cuando el reset esta desactivado y cuando se tenga un flanco de subida en la senal de reloj se le asiganara el valor de la entrada a la señal de salida q.

4.2.4. Multiplexor de Entradas

En este bloque de código se implementa un selector de entradas, las señales de entrada son las 4 señales de entrada de nuestra arquitectura (W,X,Y y Z) llamadas I0, I1, I1 e I3 respectivamente, los bits de prueba (sel) y una señal de salida llamada O (1 bit).
En el proceso tenemos unas condicionales, las cuales validan el valor que corresponde a la prueba, de acuerdo con la entrada 'sel', por ejemplo, si sel = '00' ese valor de prueba corresponde con la entra X, es decir con la entrada I0, por lo que la salida 'O' tendra el valor de la entrada I0 (X). Lo mismo aplica para las demás entradas.

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

entity mux4 is
  Port(sel: in  STD_LOGIC_VECTOR(1 downto 0);
       I0: in std_logic; --W
       I1: in std_logic; --X
       I2: in std_logic; --Y
       I3: in std_logic; --Z
       O:  out std_logic);
end mux4;

architecture Behavioral of mux4 is
begin
  process (sel, I0, I1,I2,I3)
  begin
    if sel = "00" then
      O <= I0;
    elsif sel = "01" then
      O <= I1;
    elsif sel = "10" then
      O <= I2;
    elsif sel = "11" then
      O <= I3;
    else
      O <= '0';
    end if;
  end process;
end Behavioral;
```

Listing 4: Implementación de un selector de entradas en VHDL.

4.2.5. Multiplexor de Salidas

La función de este bloque es la seleccionar el tipo de salida a tomar como salida efectiva, es decir si se tomara la salida falsa o la salida verdadera, como entrada tenemos una señal de un 1 bit, llamada 'sel', dos señales de 5 bits, una que corresponde a la salida falsa y otra a la salida verdadera y finalmente una señal de 5 bits que corresponden a las salidas efectivas. En el proceso se valida la condición de que 'sel=0', en ese caso a la señal 'salida' se le asigna el valor de la salida falsa, en caso contrario se le asigna el valor de la salida verdadera.

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


entity muxs is
  Port( sel: in std_logic;
      sal_f: in std_logic_vector(4 downto 0); --Salida Falsa
      sal_v: in std_logic_vector(4 downto 0); --Salida Verdadera
      salida: out std_logic_vector(4 downto 0) --Salidas efectivas
      ); --
end muxs;

architecture arch_muxs of muxs is
begin
  process (sel,sal_f, sal_v)
  begin
    if sel = '0' then
      salida <= sal_f;
    elsif sel = '1' then
      salida <= sal_v;
    end if;
  end process;
end arch_muxs;
```

Listing 5: Implementación de un selector de salidas en VHDL.

4.2.6. Multiplexor de Ligas

La función de este bloque es la seleccionar el tipo de liga a tomar, es decir si se tomara la liga falsa o la liga verdadera, como entrada tenemos una señal de un 1 bit, llamada 'sel', dos señales de 3 bits, una que corresponde a la liga falsa y otra a la liga verdadera y finalmente una señal de 3 bits que corresponden a la liga efectiva. En el proceso se valida la condición de que 'sel=0', en ese caso a la señal 'O' se le asigna el valor de la liga falsa, en caso contrario se le asigna el valor de la liga verdadera.

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


entity mux2_3 is
  Port( sel: in std_logic;
      I0: in std_logic_vector(2 downto 0); --Liga Falsa
      I1: in std_logic_vector(2 downto 0); --Liga Verdadera
      O: out std_logic_vector(2 downto 0)  --Estado Siguiente
      );
end mux2_3;

architecture arch_mux23 of mux2_3 is
begin
  process (sel, I0, I1)
```

```
  begin
    if sel = '0' then
      O <= I0;
    elsif sel = '1' then
      O <= I1;
    end if;
  end process;
end arch_mux23;
```

Listing 6: Implementación de un selector de ligas en VHDL.

4.2.7. Divisor de Datos

Este bloque divide los datos del contenido de la memoria, permitiendonos obtener los bits de prueba, los de la liga falsa y verdadera; asi como los bits de las salidas tanto falsas como verdaderas. Se tiene una varibale generica llamada 'data_width' cuyo valor es igual al número de bits del contenido de la memoria. Como entrada tenemos un vector de 18 bits (mem) que en realidad es el contenido de la memoria, como salidas tenemos un vector de 2 bits (prueba), dos vectores de 3 bits (liga_verdadera y liga_falsa) que corresponden a las ligas y finalmente tenemos dos vectores de 5 bits (salida_falsa y salida_verdadera) que corresponden tanto a la salida falsa como a la liga verdadera respectivamente). En el proceso se le van asignando a cada una de las señales de salida determinados bits de la señal de entrada 'mem'.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity div_datos is
  generic(
    data_width : natural :=18
  );
  port(
    mem: in std_logic_vector(data_width-1 downto 0);
    prueba: out std_logic_vector(1 downto 0);
    liga_falsa: out std_logic_vector(2 downto 0);
    liga_verdadera: out std_logic_vector(2 downto 0);
    salida_falsa : out std_logic_vector(4 downto 0);
    salida_verdadera: out std_logic_vector(4 downto 0)
  );
end entity div_datos;

architecture arch_div_d of div_datos is
begin
  process (mem)
  begin
    prueba <= mem(17 downto 16);
    liga_falsa <= mem(15 downto 13);
    liga_verdadera <= mem(12 downto 10);
    salida_falsa <= mem(9 downto 5);
    salida_verdadera <= mem(4 downto 0);
  end process;
end architecture arch_div_d;
```

Listing 7: Implementación de un divisor de datos del contenido de una memoria ROM.

4.2.8. Actividad 3

Simule su diseño para probar su funcionamiento. Recuerda mostrar el estado presente, el valor de prueba, el estado presente y las salidas.
Una vez implementada la arquitectura del direccionamiento Entrada-Estado procedemos a simular el funcionamiento de la arquitectura con la carta ASM definida al inicio de la práctica.

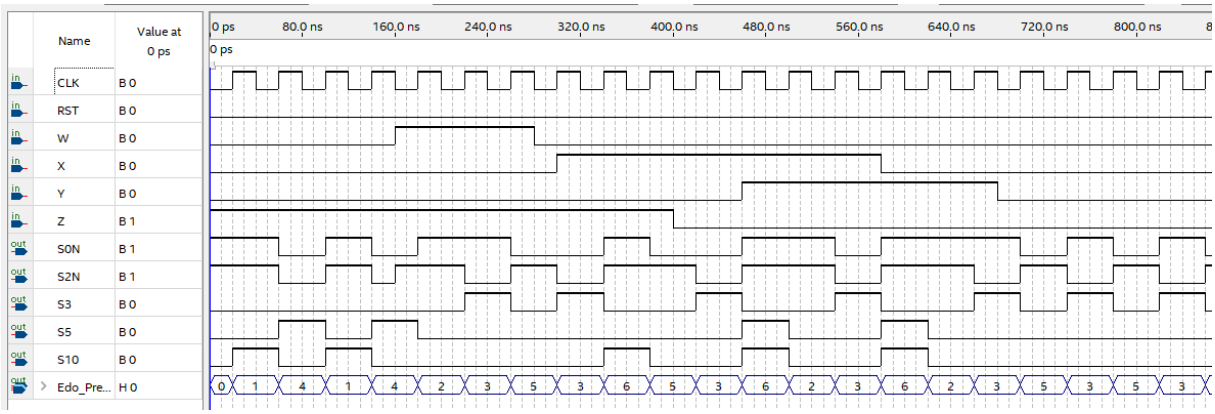

Figura 5: Salidas obtenidas al recorrer la carta ASM, cuando pasamos del estado B al estado E.

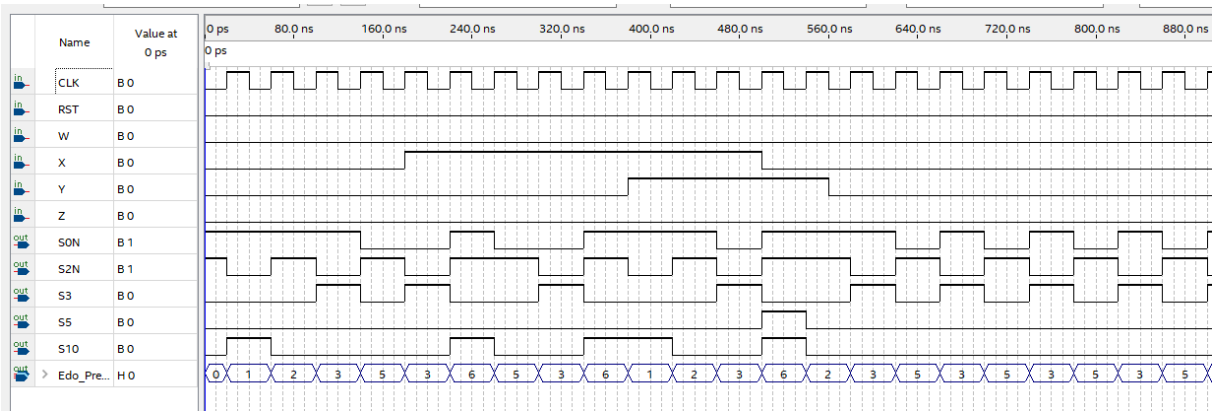

Figura 6: Salidas obtenidas al recorrer la carta ASM, cuando pasamos del estado B al estado C.

En las figura 5 y 6 podemos ver las simulaciones del recorrido de la carta ASM, el primer recorrido se puede ver en el archivo llamado **Waverform** y el segundo recorrido se puede ver en el archivo llamado **Waverform1**. Ambos archivos se anexan dentro del proyecto de la práctica. A grandes rasgos en las simulaciones se pueden ver que las salidas corresponden con el estado presente al que corresponden, de acuerdo con la tabla que obtuvimos en la actividad 1. El estado presente por comodidad y practicidad se visualiza en hexadecimal mientras que las salidas se muestran en señales con flancos de subida y bajada para ver si se encuentran activadas o no.

5. Conclusiones

El objetivo de la práctica se cumple, se entiende el funcionamiento y la lógica de un direccionamiento Entrada-Estado usando una memoria ROM, la implementación no fue complicada pero si se tuvieron que hacer ciertos ajustes como en la memoria ROM en donde la dirección solo tiene el tamaño del número de bits de la liga (sea falsa o verdadera) contrario al direccionamiento por trayectoria en donde se concatenaban los bits de la liga con el de las entrada, lo mismo ocurre con el registro, en este caso solo dejamos los bits de la liga, otro cambio importante es que hacemos que la arquitectura soporte las salidas condiciones en donde agregamos una liga y una salida adicionales, esto para poder tratar las ligas falsa y verdadera, así como una salida falsa y una verdadera, esto porque dependemos de entradas condicionales, si vale 1 entonces se usa la liga verdadera y se muestra la salida verdadera, para lograr esto se implementaron selectores de entradas, de ligas y de salidas. Para visualizar la información de mejor manera y poder manejar los datos de manera adecuada se implementó un divisor de datos, como tal no fue complicado hacer lo anterior, lo que se me complico fue entender la manera de definir el contenido de la memoria por las salidas negadas, por último, lo que puedo destacar es que con este direccionamiento optimizamos el numero de direcciones de memoria, pero a cambio se incrementa el tamaño del contenido de la memoria.

Referencias

[1] Biorobotics UNAM.(2020) *Laboratorio de Organización y Arquitectura de Computadoras Práctica No. 3 Construción de Máquinas de estados Usando Memorias Direccionamiento por Trayectoria* Recuperado el 17 de marzo de 2022, de Biorobotics UNAM en https://biorobotics.fi-p.unam.mx/wp-content/uploads/Courses/laboratorio_de_organizacion_y_arquitectura_de_computadoras/2020-2/practicas/practica3_oac.pdf

[2] Savage Carmona Jesús, Vázquez Gabriel y Chávez Rodríguez Norma Elva . (2015) *DISEÑO DE MICROPROCESADORES* Recuperado el 17 de marzo de 2022, de Biorobotics UNAM. en https://biorobotics.fi-p.unam.mx/wp-content/uploads/Courses/arquitectura_de_computadoras/material_de_apoyo/diseo_de_procesadores.pdf