UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Ingenieria

División de Ingeniería Eléctrica

Reporte de Práctica 05

Construción de Máquinas de estados Usando Memorias Direccionamiento Implícito

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

04 de abril de 2022

1. Objetivo

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

2. Duración

- 2 semanas.

3. Introducción

Para esta práctica se introducen nuevos conceptos, estos son el **estado N**, el **estado N+1** y el **P**. Hablamos de un estado N a cualquier estado en el que nos encontramos actualmente (estado presente), cuando pasamos de este estado a un estado consecuente, es decir si estamos en el estado 5 y pasamos al estadp 6 se habla de un estado N+1, finalmente cuando nos encontramos en un estado N y pasamos directamente a un estado que no es el consecuente hablamos de un estado P, en otras palabras es cuando estamos en el estado 5 y el siguiente estado al que pasamos es el estado 7, es decir que directamente saltamos del estado 5 al estado 7 sin pasar por el estado 6.

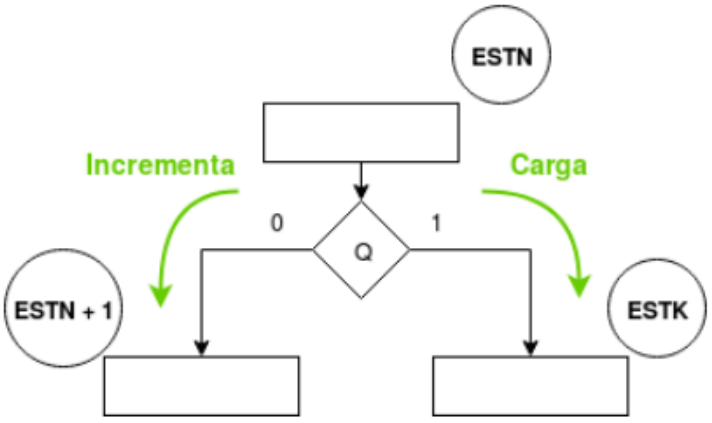

Figura 1: Representación del estado N, N+1 y P.

El direccionamiento implícito utiliza solamente un campo de liga. La variable de entrada seleccionada por el campo de prueba y VF son las que deciden si se utiliza la dirección de liga (se carga el valor de liga en el contador) o no (se incrementa el contador en una unidad).El contador en este caso nos permite obtener el estado presente, el cual se carga en la memoria ROM para asi poder obtener el contenido de memoria del estado presente y poder mostrar las salidas que le corresponden.

Un bloque importante a implementar es el de la lógica, dicho bloque se debe encargar de usar un selector de entradas para poder tomar el valor de la entrada de acuerdo con el valor de prueba del estado presente, mediante una compuerta xor se hara la operación entre el valor de la entrada y el valor de VF del estado presente, de acuerdo con la siguiente tabla que muestran los posibles resultados de la operación de la compuerta se tendra un incremento en el contador (Estado N+1) o un salto (Estado P), es decir se cargara el valor de la liga en el contador.

VF	Q	Incrementa	Carga
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

En la figura 2 se ejemplifica la misma relación mediante un diagrama lógico.

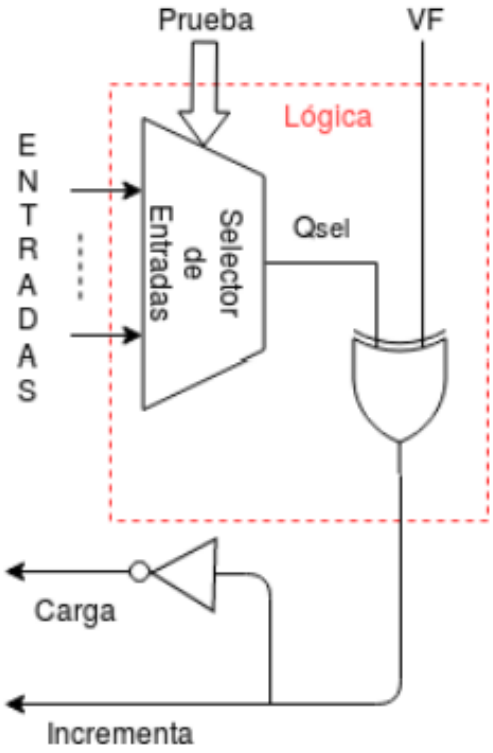

Figura 2: Diagrama de la lógica del direccionamiento implícito.

El campo prueba selecciona la entrada a sensar que junto a VF activan la se˜nal de carga o de incrementa por medio de una compuerta XOR. Es necesario tomar precauciones al hacer la asignaci´on binaria de los estados, por que se debe asegurar que por cada entrada sensada existan dos estados siguientes: uno debe tener el valor del estado presente m´as uno y el otro puede tomar cualquier otro valor.
Finalmente en la figura 3 podemos ver el diagrama general de la arquitectura del direccionamiento Implícito con los nuevos componentes a implementar.

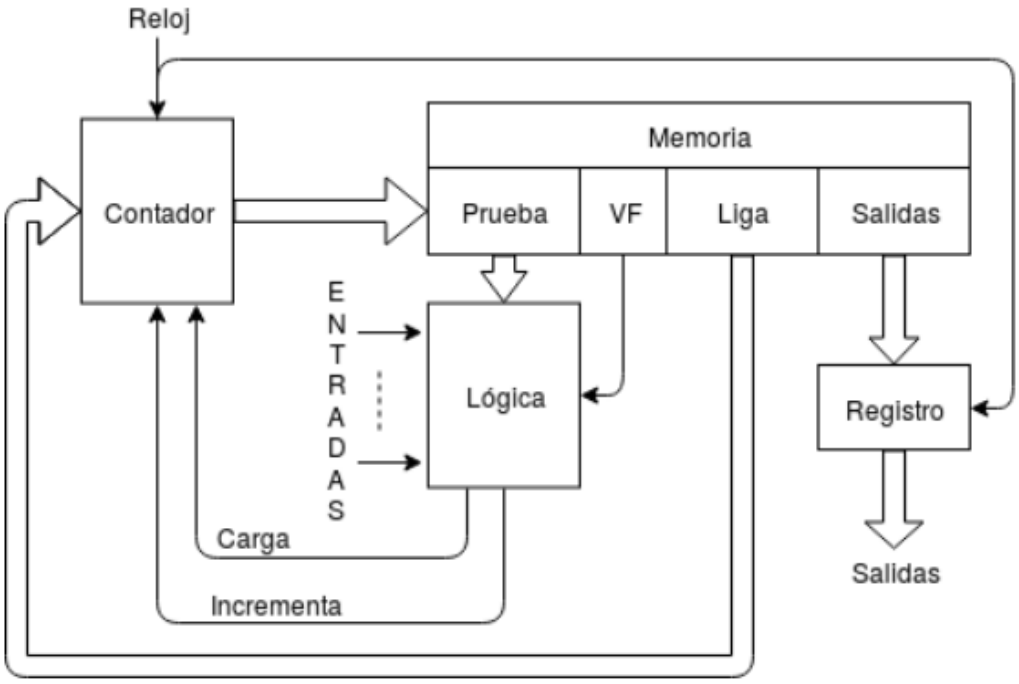

Figura 3: Diagrama del Direccionamiento Implícito.

Cálculo del tamaño de la memoria ROM:

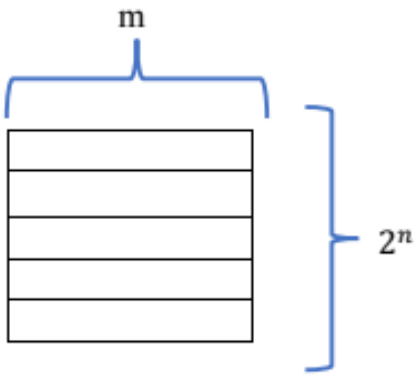

Figura 4: Memoria ROM.

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

$$m = \text{bits de prueba} + \text{VF} + \text{bits de liga} + \text{bits de salidas}$$

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

Características del direccionamiento Implícito.

1. No soporta cualquier carta ASM.
    a No soporta salidas condicionales.
    b No se puede evaluar más de una condición en un estado, las entradas deben ser mutuamente excluyentes.
2. Se debe cumplir la regla N, N+1, P para todos los estados de la carta ASM.
3. Ahorra mucha memoria.
4. Elementos de Hardware.
    a Memoria ROM.
    b Registro.
    c MUX
    d Contador o Incrementador.
    e XOR

4. Desarrollo

4.1. Actividad 1

Dada la carta ASM de la figura , encuentre el contenido de memoria utilizando el direccionamiento implícito. 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 y la variable auxiliar.

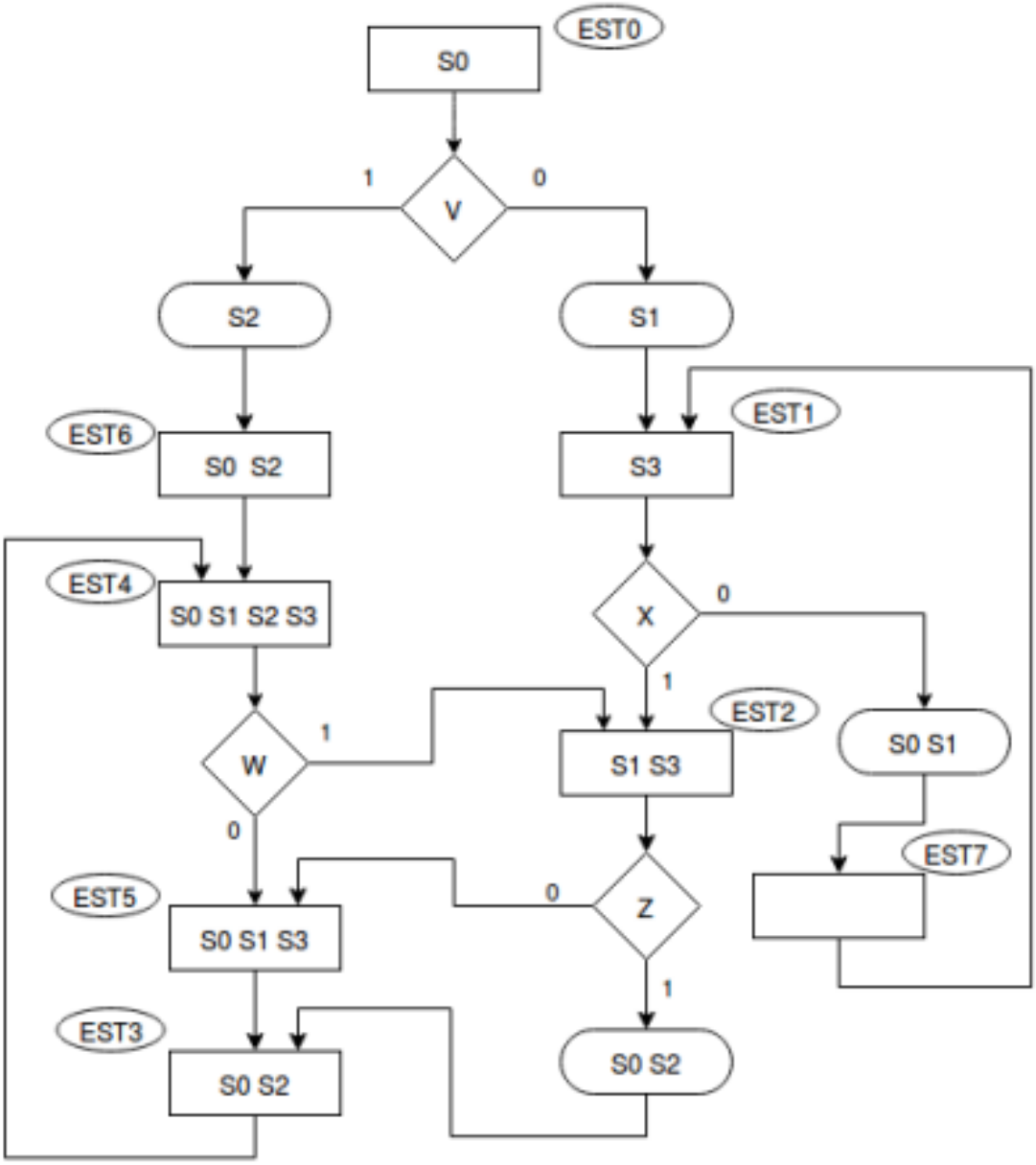

Figura 5: Carta ASM.

4.1.1. Cálculo del tamaño de la memoria ROM

Como se estan usando 8 estados, se requieren de 2^3 estados, es decir que neceistamos 3 bits para representar 8 estados, de los cuales se ocupan todos.
Considerando que tenemos 4 entradas, se ucupan 4 bits, por lo que:

$$n=4$$

Considerando que tenemos 4 entradas+ Q_{AUX}, se ocupan 3 bits de prueba, 1 bit para VF, 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 4 salidas en la carta ASM se

requieren 4 bits, como adaptaremos la arquitectura del direccionamiento implícito para que soporte salidas condicionales usaremos tanto una salida falsa como una salida verdadera, por lo que para estas requerimos un total de 8 bits.

$$m=3+1+3+4+4=15$$

Finalmente:

$$\text{Tamaño de memoria} = 2^4 * 15bits$$

4.1.2. Asignación binaria de los estados

Estado	Valor Binario
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

4.1.3. Valores de Prueba

Entrada	Prueba
V	000
W	001
X	010
Z	011
Qaux	100

4.1.4. Tabla de verdad de la lógica empleando una XNOR

Más adelante explicaremos el uso de esta tabla.

VF	Q	XNOR	Acción
0	0	1	Salto
0	1	0	P. Contiguo
1	0	0	P. Contiguo
1	1	1	Salto

4.1.5. Construir la tabla con el contenido de la memoria

A continuación se muestran las tablas con la dirección y el contenido de la memeria con el direccionamiento por trayectoria.

Direccion de memoria			Contenido de la memoria													
Estado Presente			Prueba			VF	Liga			Salidas Verdaderas				Salidas Falsas			
P2	P1	P0	K2	k1	k0		L2	L1	L0	S3	S2	S1	S0	S3	S2	S1	S0
0	0	0	0	0	0	1	1	1	0	0	1	0	1	0	0	1	1
0	0	1	0	1	0	0	1	1	1	1	0	0	0	1	0	1	1
0	1	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	0
0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	1	0	1
1	0	0	0	0	1	1	0	1	0	1	1	1	1	1	1	1	1
1	0	1	1	0	0	0	0	1	1	1	0	1	1	1	0	1	1
1	1	0	1	0	0	0	1	0	0	0	1	0	1	0	1	0	1
1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0

4.2. Actividad 2

Una vez que haya obtenido el contenido de memoria, implemente el direccionamiento impl´ıcito utilizando el software de desarrollo Quartus y escriba el contenido de memoria obtenido.
Los bloques creados para la implementación del direccionamiento Implícito en VHDL son los siguientes:

- Divisor de Frecuencia
- Memoria ROM
- Registro
- Multiplexor de Entradas
- Multiplexor de Salidas
- Multiplexor de direcciones
- Divisor de Datos
- Divisor de Salidas
- Incrementador
- Lógica

Al momento de hacer la implementación tuve un problema al realizar el bloque del contador, respecto a la lógica no tuve problemas pero al momento de conectarlo con el contador no otorgaban el estado presente adecuado, para solucionar esto, tuve que implementar el direccionamiento implícito usando el diagrama con incrementador que se muestra en la figura 6.

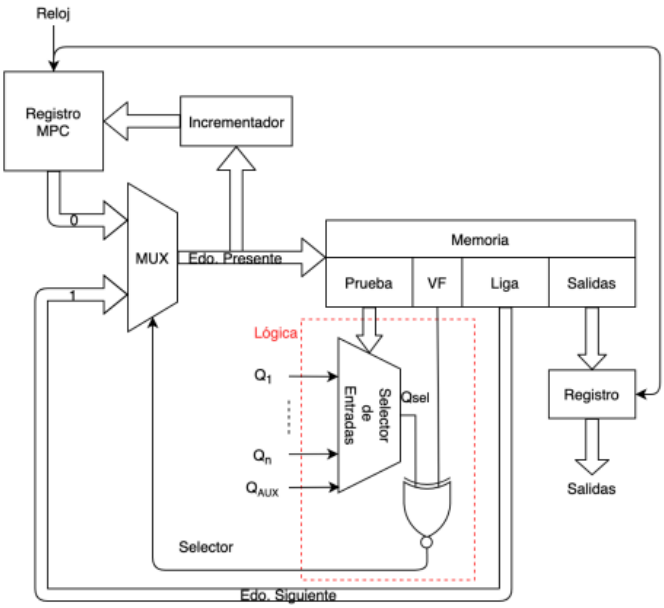

Figura 6: Direccionamiento Implícito (Diagrama con Incrementador).

Con este diagrama el contador es reemplazado por un incrementador, el cual solo incrementa el estado N, además se implementa un multiplexor para seleccionar si se ocupa la liga para hacer un salto condicional o un paso contiguo (estado N+1). Para determinar si se hace un salto o un paso contiguo usamos la lógica empleando una compuerta xnor, cuyo diagrama se muestra en la figura 7.

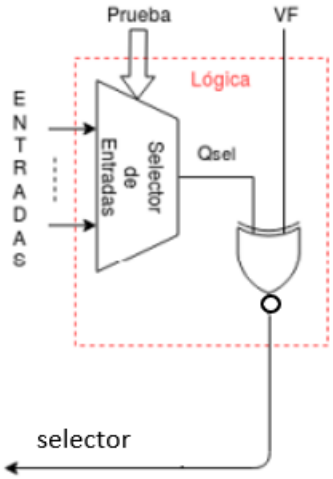

Figura 7: Lógica con compuerta XNOR.

Esta es la razón por la que colocamos la tercer tabla en el punto anterior, cuando la operación de la compuerta XNOR entre el valor de la entrada y el valor de VF es 1, se hace un salto, es decir cargamos en la memoria ROM la liga del estado presente, cuando el resultado de la operación es 0 entonces el estado presente entra en el incrementador y como salida carga el estado N+1 en la memoria ROM.

4.2.1. Divisor de Frecuencia

En la entidad del codigo podemos ver que se usan dos señales, la primera es la entrada llamada clk, la cual tomara la frecuencia de la tarjeta, dicha señal es de un bit, la señnal 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 senal de salida div_ clk.

```
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.

4.2.2. Memoria ROM

A continuacion se muestra el código que nos ayuda a crear una memoria ROM, en la definición 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 :=15;  --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"000111001010011", 1 =>b"010011110001011",
      2 => b"011010111111010", 3 =>b"100110001010101",
      4 => b"001101011111111", 5 =>b"100001110111011",
      6 => b"100010001010101", 7 =>b"100000100000000"

    );

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

Listing 2: Implementación de una Memoria ROM en VHDL.

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, estaremos 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, entradas)
  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.

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 (V,W,X 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 = '000' 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 mux_ent is
```

```
  Port(sel: in  STD_LOGIC_VECTOR(2 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 mux_ent;

architecture Behavioral of mux_ent is
begin
  process (sel, I0, I1,I2,I3)
  begin
    if sel = "000" then
      O <= I0;
    elsif sel = "001" then
      O <= I1;
    elsif sel = "010" then
      O <= I2;
    elsif sel = "011" then
      O <= I3;
    elsif sel = "100" then  -- Auxiliar
      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 4 bits, una que corresponde a la salida falsa y otra a la salida verdadera y finalmente una señal de 4 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(3 downto 0); --Salida Falsa
       sal_v: in std_logic_vector(3 downto 0); --Salida Verdadera
       salida: out std_logic_vector(3 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 direcciones

Este es uno de los nuevos bloques que se implementan para esta práctica, su función es la de seleccionar la dirección que tomara la memoria ROM, bien puede ser el valor de la liga del estado presente o el valor del estado N+1 del estado presente. el selector de este multiplexor es otorgado por la salida 'INC' del bloque que implementa la lógica del direccionamiento implícito, si dicha saldia se encuentra activada, el multiplexor permitirá cargar el valor del estado N+1 hacia la memoria ROM, en caso de que se encuentre desactivada se cargara el valor de la liga en la memoria ROM para hacer un salto.

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


entity mux_dir is
  Port( sel: in std_logic;
      paso_c: in std_logic_vector(2 downto 0); --Paso contiguo
      liga: in std_logic_vector(2 downto 0); --Salto
      edo_pres: out std_logic_vector(2 downto 0) -- Edo. Presente
      );
end mux_dir;

architecture arch_mdir of mux_dir is
begin
  process (sel,paso_c, liga)
  begin
    if sel = '0' then
      edo_pres <= paso_c; --Estado N+1
    else
      edo_pres <= liga;   --Estado P
    end if;
  end process;
end arch_mdir;
```

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

4.2.7. Incrementador

Como parte de la adaptación del diagrama de Direccionamiento Implícito con Incrementador, necesitamos implementar dicho incrementador, la función de este bloque es la de incremrimentar en una unidad al estado presente (Estado N) para obtener el estado siguiente (Estado N+1), con el que podemos hacer el paso contiguo y cargar la dirección del estado siguiente en la Memoria ROM.

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

entity incrementador is
  generic(
    data_width: natural := 3
  );
  port(
    edo_n: in std_logic_vector(data_width-1 downto 0);
    est_pres: out std_logic_vector(data_width-1 downto 0)
  );
end entity incrementador;

architecture inc of incrementador is
begin
  process(edo_n)
  begin
    est_pres <=std_logic_vector(unsigned(edo_n)+1);
  end process;
end architecture;
```

Listing 7: Implementación de un selector de salidas en VHDL.

4.2.8. Lógica

Para la implementación de este bloque se creó un archivo esquemático, en donde usamos el bloque del multiplexor de entrdas y una compuerta XNOR de acuerdo con el diagrama del Direccionamiento Implícito con Incrementador, la interconexión de los elementos, las entradas y las salidas se observan en la figura 8.

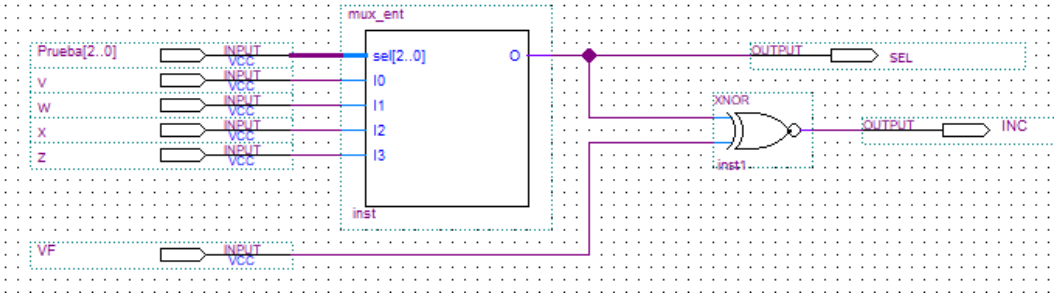

Figura 8: Esquematico de la Lógica con compuerta XNOR.

Las entradas de la lógica son los 3 bits de la prueba, 4 bits para las entradas (1 bit por entrada) y un bit del valor de VF; para las salidas tenemos un bit para el selector de las salidas (SEL) verdadera/falsa y un bit para el selector de dirección (INC) para el valor de la liga o del estado N+1.

4.2.9. Divisor de Datos

Este bloque divide los datos del contenido de la memoria, permitiendonos obtener los bits de prueba, el bit de VF, los de la liga; asi como los bits de las salidas tanto falsas como verdaderas. Se tiene una variable generica llamada 'data_width' cuyo valor es igual al número de bits del contenido de la memoria. Como entrada tenemos un vector de 15 bits (mem) que en realidad es el contenido de la memoria, como salidas tenemos un vector de 3 bits (prueba), 1 bit para VF, un vector de 3 bits que corresponde a la liga y finalmente tenemos dos vectores de 4 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 :=15
  );
  port(
    mem: in std_logic_vector(data_width-1 downto 0);
    prueba: out std_logic_vector(2 downto 0);
    vf: out std_logic;
    liga: out std_logic_vector(2 downto 0);
    salida_verdadera : out std_logic_vector(3 downto 0);
    salida_falsa: out std_logic_vector(3 downto 0)
  );
end entity div_datos;

architecture arch_div_d of div_datos is
begin
  process (mem)
  begin
    prueba <= mem(14 downto 12);
    vf <= mem(11);
    liga <= mem(10 downto 8);
    salida_verdadera <= mem(7 downto 4);
    salida_falsa <= mem(3 downto 0);
  end process;
end architecture arch_div_d;
```

Listing 8: Implementación de un divisor de datos del contenido de una memoria ROM.

4.2.10. Divisor de Salidas

Este bloque divide las salidas obtenidas del selector de salidas, permitiendonos obtener los bits por separado de las 4 salidas. Se tiene una variable generica llamada 'data_width' cuyo valor es igual al número de bits de las salidas. Como entrada tenemos un vector de 4 bits (salidas), como salidas tenemos 4 salidas de un bit cada una, en donde se les asocian las salidas S0, S1, S2 y S3 respectivamente En el proceso se le van asignando a cada una de las señales de salida determinados bits de la señal de entrada 'salidas'.

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

entity div_salidas is
  generic(
    data_width : natural :=4
  );
  port(
    rst: in std_logic;
    salidas: in std_logic_vector(data_width-1 downto 0);
    s3: out std_logic;
    s2: out std_logic;
    s1: out std_logic;
    s0: out std_logic
  );
end entity div_salidas;

architecture arch_div_s of div_salidas is
begin
  process (salidas)
  begin
    if rst = '1' then

      s3 <= '0';
      s2 <= '0';
      s1 <= '0';
      s0 <= '0';
    else
      s3 <= salidas(3);
      s2 <= salidas(2);
      s1 <= salidas(1);
      s0 <= salidas(0);
    end if;
  end process;
end architecture arch_div_s;
```

Listing 9: Implementación de un divisor de salidas.

4.3. Actividad 3

Simule su diseño para probar su funcionamiento. Sus simulaciones deben mostrar el contenido de la memoria, así como el estado presente.

Una vez implementada la arquitectura del direccionamiento Implícito procedemos a simular el funcionamiento de la arquitectura con la carta ASM definida al inicio de la práctica.

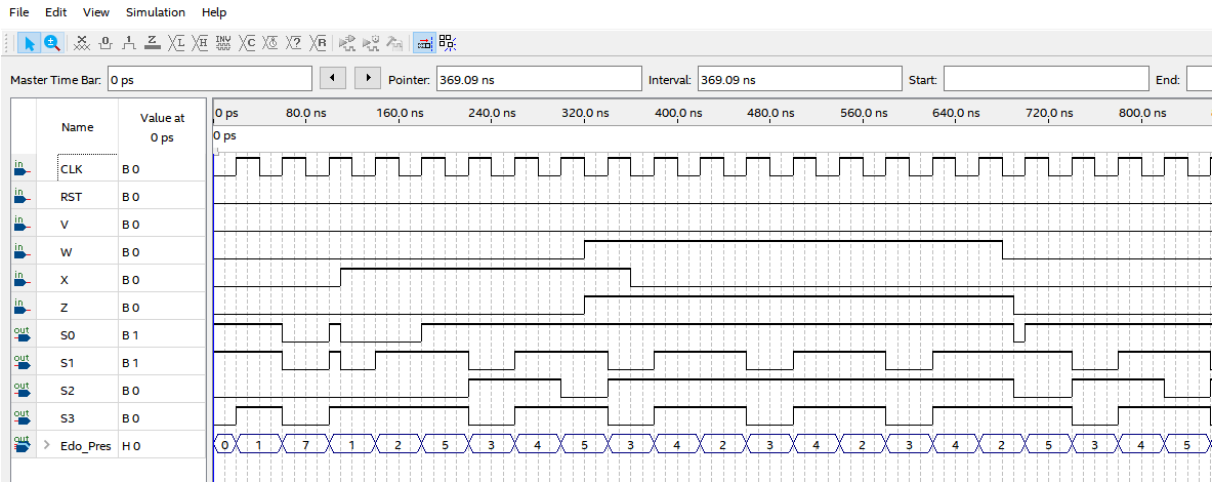

Figura 9: Salidas obtenidas al recorrer la carta ASM, cuando pasamos del estado 1 al estado 7.

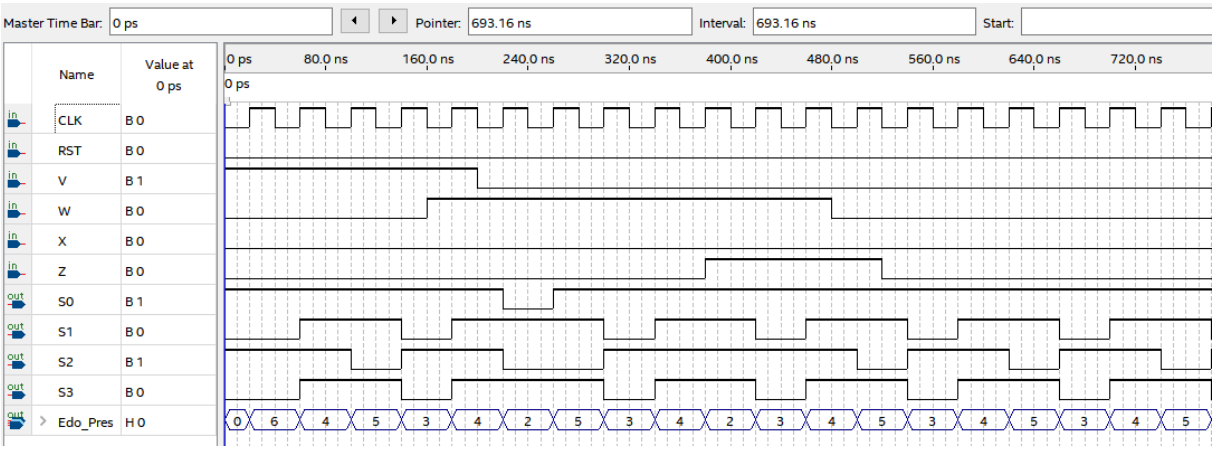

Figura 10: Salidas obtenidas al recorrer la carta ASM, cuando pasamos del estado 0 al estado 6.

En las figura 9 y 10 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, 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. Otro ajuste que se realizó fue la implementación de un selector de salidas, con el fin de que esta arquitectura soporte las salidas

condicionales al igual que con el direccionamiento Entrada-Estado de la práctica anterior.

5. Conclusiones

El objetivo de la práctica se cumple, se entiende el funcionamiento y la lógica de un direccionamiento Implícito usando una memoria ROM, la implementación no fue complicada pero si se tuvieron que hacer ciertos ajustes como lo fue el reemplazo del contador por los bloques incrementador y MUX de direcciones para poder determinar si se hacia un salto o un paso contiguo para acceder al proximo estado según el recorrido que se haga en la carta ASM. Me tarde un poco más de lo esperado con la implementación por el detalle del contador, lo nuevo que me llevo es la forma de crear un nuevo esquemático partiendo de otro esquemático y que a su vez podemos usar los bloques ya creados en Quartus para generar un nuevo esquemático. La ventajas que veo de este direccionamiento es que optimiza el tamaño del contenido de la memoria, sin embargo los bloques requeridos para su implementación aumentan.

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 03 de abril 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 03 de abril 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