Facultad de Ingeniería

Lab. Organización y Arquitectura de Computadoras

Práctica No 5
Construcción de Máquinas de estados Usando Memorias Direccionamiento Implícito

Alumnos

- Monsalvo Bolaños Melissa Monserrat
- Romero Andrade Cristian

Grupo: 01

Profesor
Ing. Adrian Ulises Mercado Martinez

Semestre
2022–1

Fecha de Entrega
21 de octubre de 2021

Practica 5

Monsalvo Bolaños Melissa Monserrat y Romero Andrade Cristian

Índice

1. Introducción

Este tipo de direccionamiento utiliza solamente un campo de liga. Una 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 campo VF (Verdadero-Falso) sirve para indicarle a la lógica cuánto debe valer la variable de entrada, para así cargar en el contador el valor de la liga y hacer el salto. El campo de

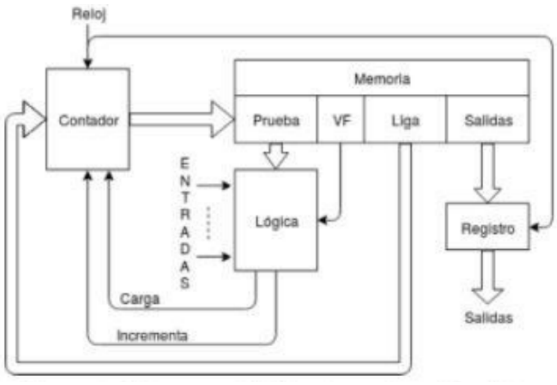

Figura 1: Diagrama de direccionamiento implícito

verdadero-falso (VF) tiene la utilidad de indicar el calo de la variable de entrada con el objetivo

Tabla 1: Relación VF y Q

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

de cargar el contador el valor de la liga y realizar el salto. En la tabla 1 muestra la relación de

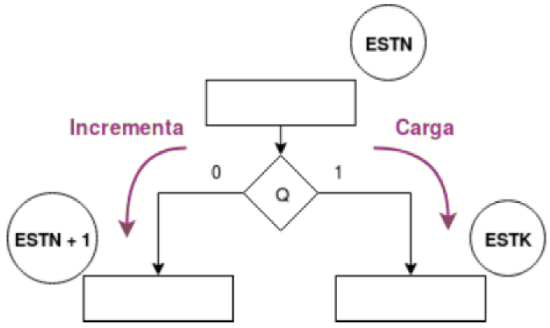

Figura 2: Direccionamiento Implicito.

VF y la variable de entrada con cadenas de incremento y carga No obstante el diseño original no esta diseño para aceptar salidas condicionales, es por ello que se propone las siguiente forma.

2. Objetivo

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

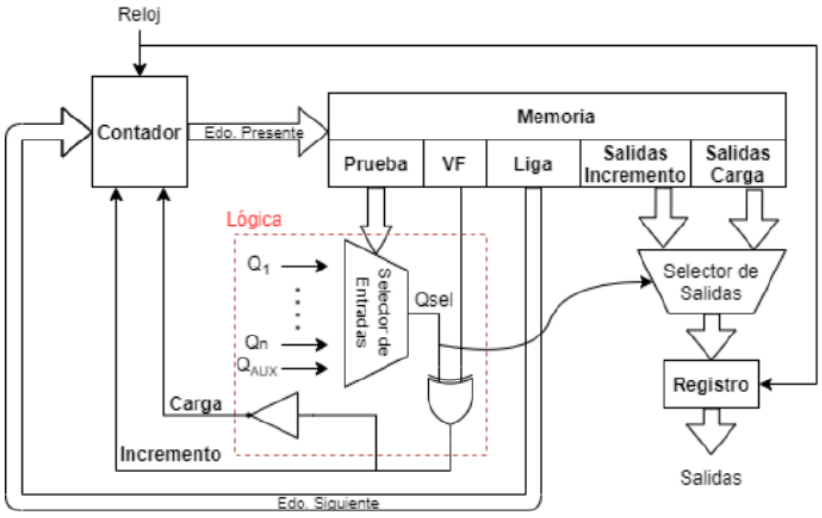

Figura 3: Diagrama Modificado

3. Desarrollo

Partimos del análisis de la carta Asm Asignamos a cada entrada y estado un valor binario

Tabla 2: Valores binarios deestados

Estado	Código binario		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tabla 3: Valores binarios de variables

v	0	0	0
w	0	0	1
x	0	1	0
z	0	1	1
aux	1	0	0

Posteriormente rellenamos la tabla de verdad conforme a la carta ASM

Tabla 4: Tabla de verdad de la carta ASM

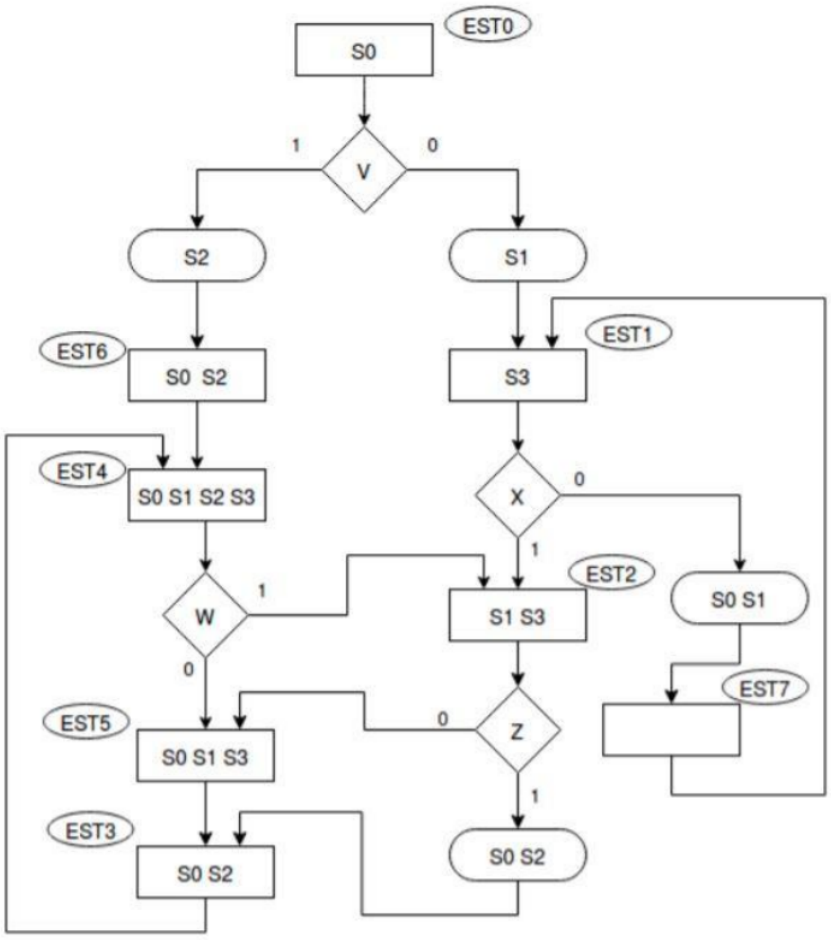

Figura 4: Carta ASM

Dirección de memoria			Contenido de la memoria															
Estado Presente			Prueba			VF	Liga			Salidas falsas				Salidas verdaderas				
										S3	S2	S1	S0	S3	S2	S1	S0	
0	0	0	0	0	0		1	1	0	0	0	1	1	0	1	0	1	
0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0	0	
0	1	0	0	1	1	0	1	0	1	1	0	1	0	1	1	1	1	
0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	1	0	1	
1	0	0	0	0	1	0	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	

Una vez obtenido el comportamiento de la carta ASM, proseguimos a programar los componentes de la Figura 1 en Quartus

Memoria Se retomó el código de prácticas anteriores y se llenó el contenido de la memoria con el contenido de la tabla de verdad obtenida previamente, tiene como entrada la dirección de memoria a la que se desea acceder y como salida el contenido de dicha localidad.

Código 1: `memoria.vhd`

```
        library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity memoria is
  generic(
    data_width : natural := 15;
    addr_length : natural := 3
    );
  port(
    clk: in std_logic;
    address: in std_logic_vector(2 downto 0);
    data_out: out std_logic_vector(14 downto 0);
    mon_address: out std_logic_vector(2 downto 0)
    );
end memoria;
architecture behavioral of memoria is
  constant mem_size: natural := 15;
  type mem_type is array (mem_size-1 downto 0) of std_logic_vector(data_width-1 downto 0);
  constant mem : mem_type :=
    (
      0 => b"000011000110101",
      1 => b"010011110111000",
      2 => b"011010110101111",
      3 => b"100110001010101",
      4 => b"001001011111111",
      5 => b"100001110111011",
      6 => b"100010001010101",
      7 => b"100000100000000",
      others=>b"111111111111111"
      );
begin
  rom : process(clk)
```

```
  begin
    if rising_edge(clk) then
      mon_address <= address;
      data_out <= mem(to_integer(unsigned(address)));
    end if;
  end process rom;
end architecture behavioral;
```

Separador Este bloque separa la salida en memoria en los campos que necesitamos. `separador.vhd`

```
library ieee;
use ieee.std_logic_1164.all;

entity separador is port (
  data_in : in std_logic_vector (14 downto 0);
  prueba: out std_logic_vector(2 downto 0);
  vf : out std_logic;
  liga : out std_logic_vector(2 downto 0);
  salida_incremento : OUT std_logic_vector(3 DOWNTO 0);
  salida_carga : OUT std_logic_vector(3 DOWNTO 0)
  );
end separador;
architecture behavioral of separador is
begin
  process (data_in)
  begin
    vf <= data_in(11);
    liga <= data_in(10 downto 8);
    salida_incremento <= data_in(7 downto 4);
    salida_carga <= data_in(3 downto 0);
  end process;
end behavioral;
```

Contador Es el que se encarga de proporcionar el estado siguiente cada vez que nos encontremos con un paso continuo

Código 2: `contador.vhd`

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

entity contador is
  port(
    clk: in std_logic;
    rst: in std_logic;
    load: in std_logic;
    salida_count: out std_logic_vector(2 downto 0);
    liga: in std_logic_vector(2 downto 0) -- el que no es continuo
    );
end entity;

architecture rtl of contador is
```

```
  signal cuenta : integer range 0 to 7;


begin
  process(clk,rst)
  begin
    if (rst='1') then
      cuenta <= 0;
    elsif(rising_edge(clk)) then
      if load = '1' then
        cuenta<= to_integer(unsigned(liga));
      else
        if(cuenta=7) then
          cuenta <= 0;
        else
          cuenta <= cuenta + 1;
        end if;
      end if;
    end if;
  end process;

  salida_count <= std_logic_vector(to_unsigned(cuenta, 3));

end architecture;
```

Lógica Aquí se decide qué tipo de acción se debe realizar dependiendo del valor de la entrada y la prueba. Ya sea un paso continuo o un salto condicional.

Código 3: `logica.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity logica is port (
  vf : in std_logic;
  prueba : in std_logic_vector(2 downto 0);
  entrada : in std_logic_vector(2 downto 0);
  mon_prueba: out std_logic_vector(2 downto 0);
  carga : out std_logic;
  incrementa : out std_logic
  );
end logica;

architecture behavioral of logica is
  signal qset : std_logic;
  signal xor_res : std_logic;
begin
  process (prueba)
  begin
    case(entrada) is
      when "000" =>
        if entrada = prueba then
          qset <= '1';
        else
          qset <= '0';
        end if;
      when "001" =>
```

```
        if entrada = prueba then
          qset <= '1';
        else
          qset <= '0';
        end if;
      when "010" =>
        if entrada = prueba then
          qset <= '1';
        else
          qset <= '0';
        end if;
      when "011" =>
        if entrada = prueba then
          qset <= '1';
        else
          qset <= '0';
        end if;
      when "100" =>
        if entrada = prueba then
          qset <= '1';
        else
          qset <= '0';
        end if;
      when others =>
        qset <= '0';
    end case;
  end process;

  process (qset)
  begin
    if qset = '1' xor vf='1' then
      xor_res <= '1';
    else
      xor_res <= '0';
    end if;
  end process;

  process(xor_res)
  begin
    carga <= not xor_res;
    incrementa <= xor_res;
  end process;
end behavioral;
```

Selector de salida Dependiendo si se trate de un paso continuo o un salto condicional, este bloque utilizará la salida para enviar el estado siguiente.

Código 4: `selector_salida.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity selector_salida is port (
  carga : in std_logic;
  incrementa : in std_logic;
  salida_incremento : in std_logic_vector (3 downto 0);
  salida_carga : in std_logic_vector (3 downto 0);
```

```
  salida : out std_logic_vector (3 downto 0)
  );
end selector_salida;

architecture behavioral of selector_salida is
begin
  process (carga, incrementa)
  begin
    if carga = '1' then
      salida <= salida_carga;
    elsif incrementa = '1' then
      salida <= salida_incremento;
    else
      salida <= "1000";
    end if;
  end process;
end behavioral;
```

3.1. Diagrama

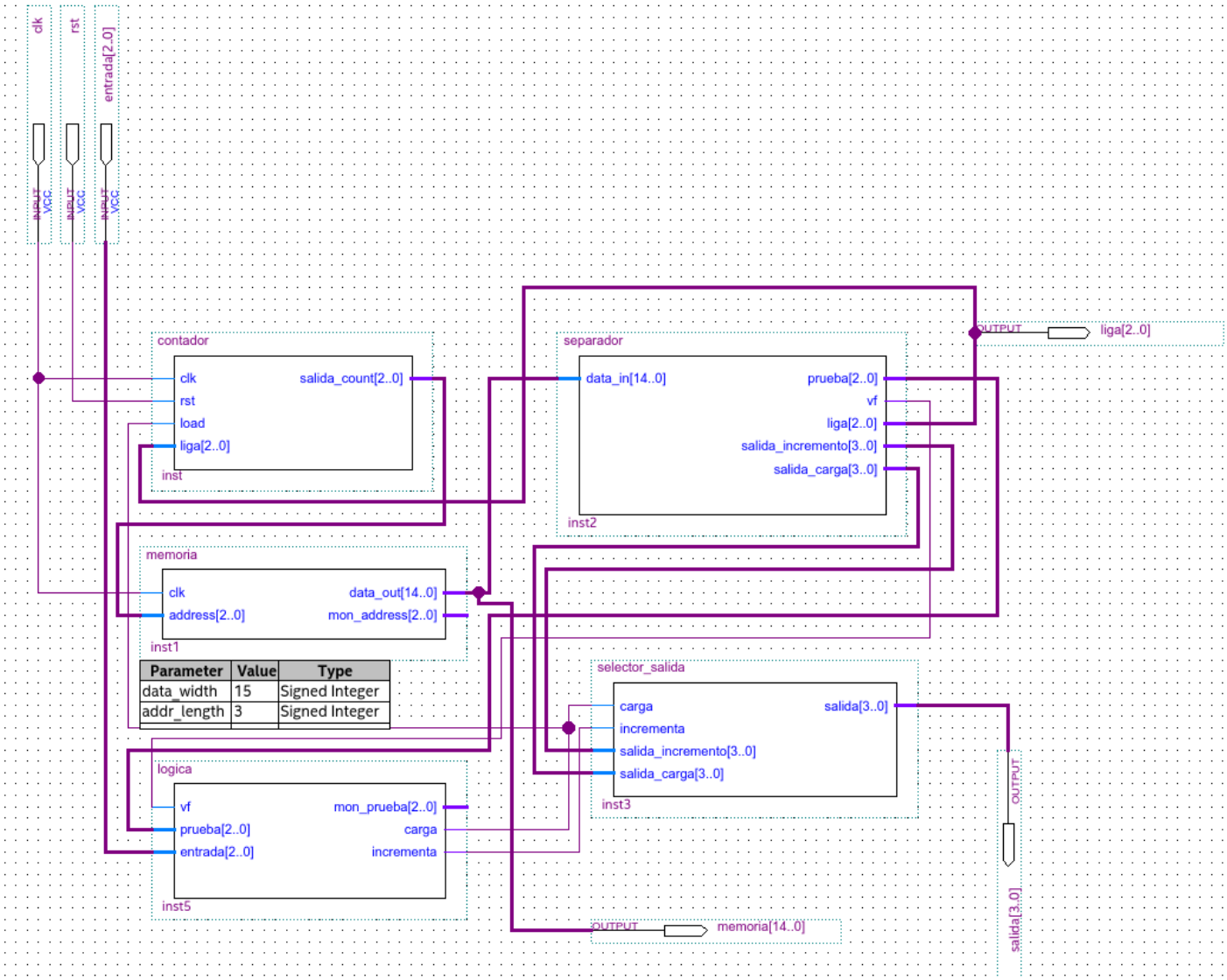

Figura 5: Diagrama

3.2. Simulación

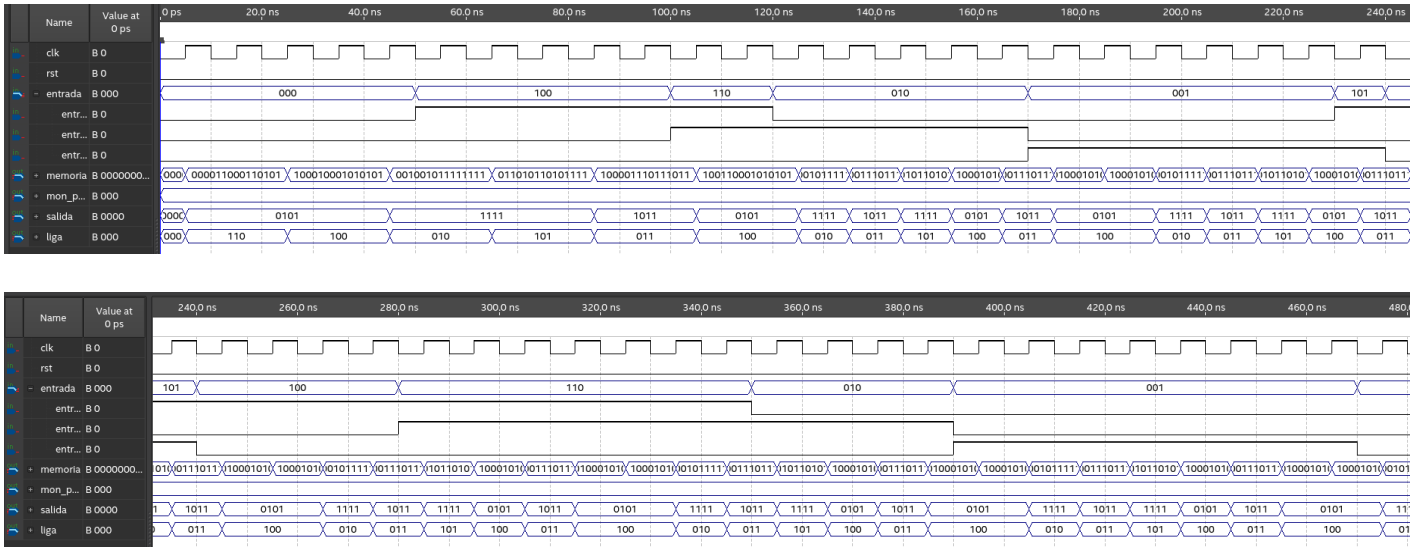

Figura 6: Simulación

4. Conclusiones

Monsalvo Bolaños Melissa Monserrat

Gracias a esta práctica pudimos implementar el desarrollo de una máquina de estado usando memorias por direccionamiento implícito, esto requirió la implementación de un contador y la modificación de la memoria para las salidas, del mismo modo utilizamos variables nuevas como vf. En comparación con prácticas pasadas está un poco más complicada.

Romero Andrade Cristian

Romero Andrade Cristian: En la práctica, el direccionamiento implícito se implementó en VHDL, donde se desarrollaron los bloques que se muestran en la Figura 1. La ventaja de este tipo de direccionamiento es que no se integra colocando ninguna dirección de forma explícita, ya que la dirección se conoce en el propio opcode los operandos a los que desea acceder o trabajar

Conclusiones en equipo

En la práctica se implementó el direccionamiento implícito en VHDL, donde se desarrollaron los bloques de la figura 1, este tipo de direccionamiento tiene el beneficio de no integrar poner ninguna dirección de forma explícita, ya que en el propio código de operación se conoce la dirección de los operandos al que se desea acceder o con el que se quiere operar.

Referencias

Chavez, N. E. (s.f.). Construcción de máquinas de estados usando memorias. http://profesores.fi-b.unam.mx/normaelva

Savage, J. & Vázquez, G. (s.f.). *Diseño de microprocesadores*. Facultad de Ingenieria.

Índice de tablas

Índice de figuras

Índice de Códigos