

Facultad de Ingeniería



Lab. Organización y Arquitectura de Computadoras

Práctica No 5 Construcción de Máquinas de estados Usando Memorias Direcccionamiento 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	Objetivo	2
2	Introducción	2
3	Desarrollo	4
3.1	Códigos	5
3.2	Diagrama	9
3.3	Simulación	10
4	Conclusiones	11
	Referencias	11

1. Objetivo

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

2. 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.

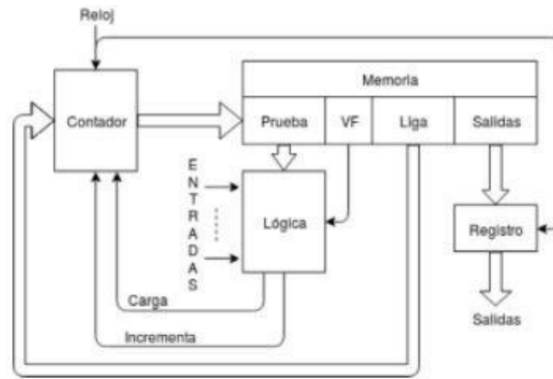


Figura 1: Diagrama de direccionamiento implícito

El campo de verdadero-falso (VF) tiene la utilidad de indicar el caso de la variable de entrada con el objetivo de cargar el contador el valor de la liga y realizar el salto.

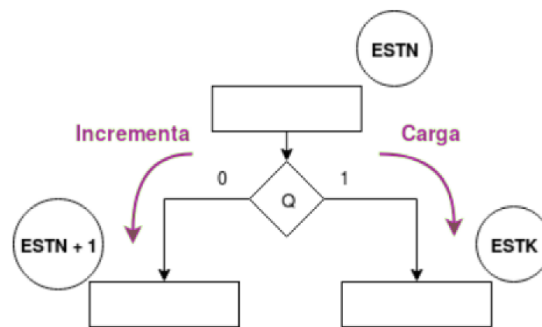


Figura 2: Direccionamiento Implícito.

En la tabla 1 muestra la relación de VF y la variable de entrada con cadenas de incremento y carga

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

No obstante el diseño original no esta diseñado para aceptar salidas condicionales, es por ello que se propone las siguiente forma.

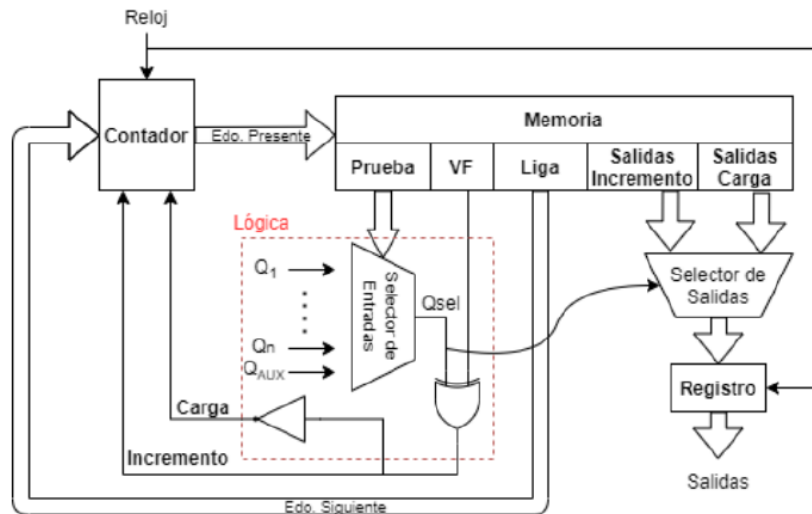


Figura 3: Diagrama Modificado

3. Desarrollo

Partimos del análisis de la carta Asm

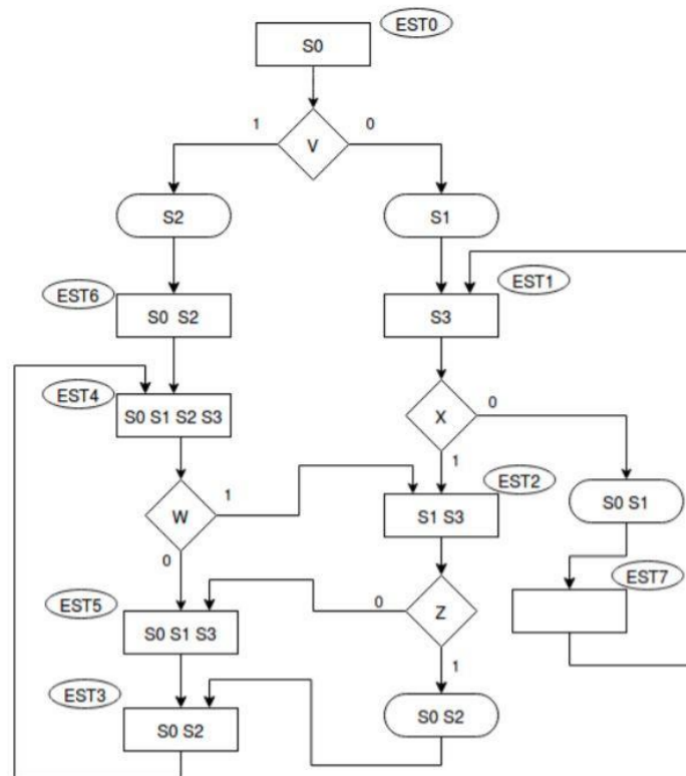


Figura 4: Carta ASM

Asignamos a cada entrada y estado un valor binario

Tabla 2: Valores binarios de estados

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

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

3.1. Códigos

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

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  entity memoria is
5      generic(
6          data_width : natural := 15;
7          addr_length : natural := 3
8      );
9  port(
10     clk: in std_logic;
11     address: in std_logic_vector(2 downto 0);
12     data_out: out std_logic_vector(14 downto 0)
13 );
14 end memoria;
15 architecture behavioral of memoria is
16     constant mem_size: natural := 15;
17     type mem_type is array (mem_size-1 downto 0) of std_logic_vector(data_width-1 downto 0);
18     constant mem : mem_type :=
19     (
20         0 => b"000111000110101",
21         1 => b"010011110111000",
22         2 => b"011010110101111",
23         3 => b"100110001010101",
24         4 => b"001101011111111",
25         5 => b"100001110111011",
26         6 => b"100010001010101",
27         7 => b"100000100000000",
28         others=>b"111111111111111"
29     );
30 begin
31     rom : process(clk)
32     begin
33         if rising_edge(clk) then
34             data_out <= mem(to_integer(unsigned(address)));
35         end if;
36     end process rom;
37 end architecture behavioral;
```

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

Código 2: separador.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity separador is port (
5      data_in : in std_logic_vector (14 downto 0);
6      prueba: out std_logic_vector(2 downto 0);
7      vf : out std_logic;
8      liga : out std_logic_vector(2 downto 0);
9      salida_falsa : OUT std_logic_vector(3 DOWNT0 0);
10     salida_verdadera : OUT std_logic_vector(3 DOWNT0 0)
11 );
12 end separador;
13 architecture behavioral of separador is
14 begin
15     process (data_in)
16     begin
```

```

17     vf <= data_in(11);
18     liga <= data_in(10 downto 8);
19     salida_falsa <= data_in(7 downto 4);
20     salida_verdadera <= data_in(3 downto 0);
21     end process;
22     end behavioral;

```

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

Código 3: contador.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity contador is
6  port(
7      clk: in std_logic;
8      rst: in std_logic;
9      load: in std_logic;
10     salida_count: out std_logic_vector(2 downto 0);
11     liga: in std_logic_vector(2 downto 0) -- el que no es continuo
12 );
13 end entity;
14
15 architecture rtl of contador is
16     signal cuenta : integer range 0 to 7;
17
18
19 begin
20     process(clk,rst)
21     begin
22         if (rst='1') then
23             cuenta <= 0;
24         elsif(falling_edge(clk)) then
25             if load = '1' then
26                 cuenta<= to_integer(unsigned(liga));
27             else
28                 if(cuenta=7) then
29                     cuenta <= 0;
30                 else
31                     cuenta <= cuenta + 1;
32                 end if;
33             end if;
34         end if;
35     end process;
36
37     salida_count <= std_logic_vector(to_unsigned(cuenta, 3));
38
39 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 4: logica.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity logica is port (
7      vf : in std_logic;
8      prueba : in std_logic_vector(2 downto 0);
9      entrada : in std_logic_vector(4 downto 0);
10     qsel: out std_logic;
11     carga : out std_logic;
12     incrementa : out std_logic
13 );
14 end logica;
15
16 architecture behavioral of logica is
17     signal qset : std_logic;
18     signal xor_res : std_logic;
19 begin
20     process (prueba)
21     begin
22         case(prueba) is
23             when "000" =>
24                 qset <= entrada(4);
25             when "001" =>
26                 qset <= entrada(3);
27             when "010" =>
28                 qset <= entrada(2);
29             when "011" =>
30                 qset <= entrada(1);
31             when others =>
32                 qset <= '0';
33         end case;
34     end process;
35
36     process (qset)
37     begin
38         xor_res <= qset xor vf;
39         qsel <= qset;
40     end process;
41
42     process(xor_res)
43     begin
44         carga <= not xor_res;
45         incrementa <= xor_res;
46     end process;
47 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 5: selector_salida.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
```



```

6  entity selector_salida is port (
7      qsel : in std_logic;
8      salida_falsa : in std_logic_vector (3 downto 0);
9      salida_verdadera : in std_logic_vector (3 downto 0);
10     salida : out std_logic_vector (3 downto 0)
11 );
12 end selector_salida;
13
14 architecture behavioral of selector_salida is
15 begin
16     process (qsel)
17     begin
18         if qsel = '1' then
19             salida <= salida_verdadera;
20         else
21             salida <= salida_falsa;
22         end if;
23     end process;
24 end behavioral;

```

3.2. Diagrama

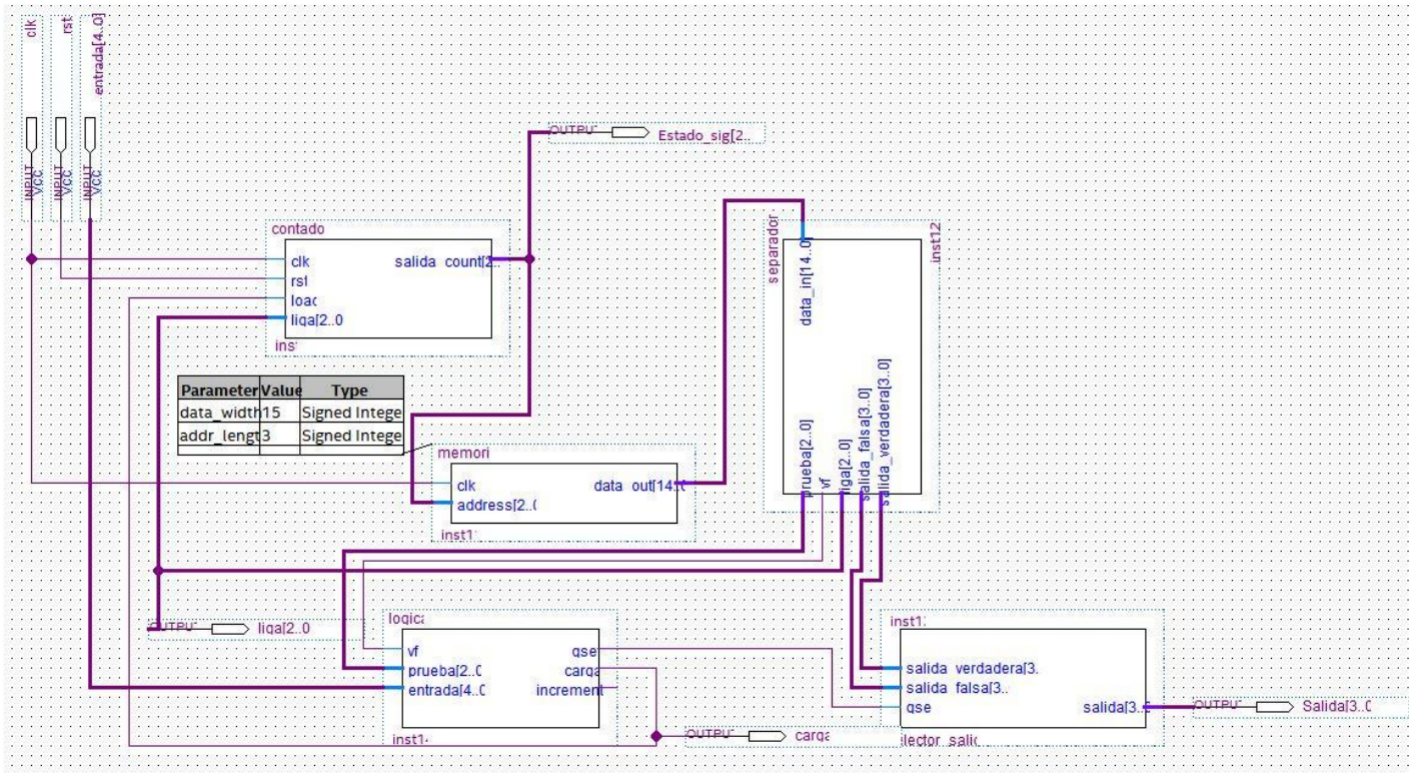


Figura 5: Diagrama

3.3. Simulación

Para este caso simularemos todas las entradas en 0

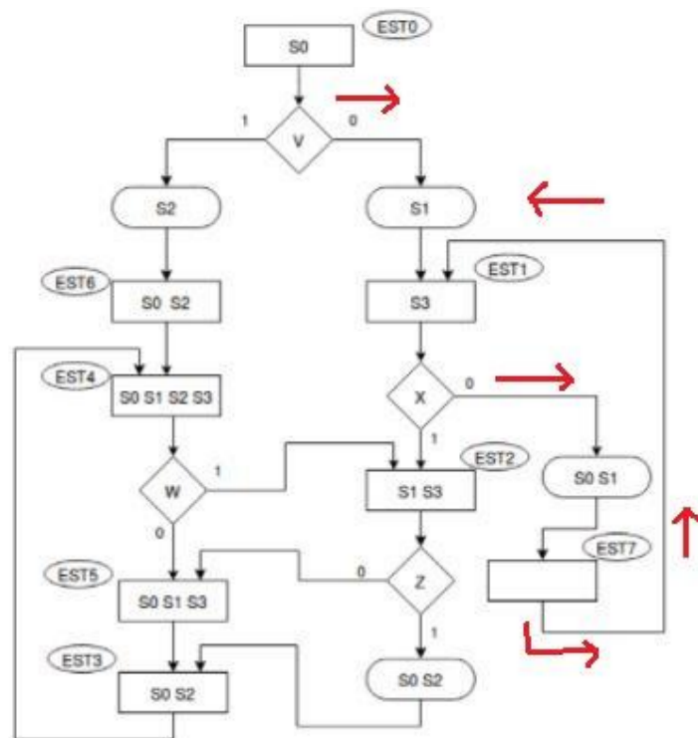


Figura 6: Simulación

1. Podemos observar que en el estado 0 se evalúa la variable V, como está es 0 se pasa al estado 1 teniendo como salidas a S0 y S1.
2. Luego, en el estado 1 se evalúa x, como esta esta en 0 pasa al estado 7 teniendo como salida s3, s0 y s1.
3. Por último, se realiza un salto condicional al estado 1 con con todas las salidas en bajo, a partir de aquí se repiten los pasos 2 y 3.

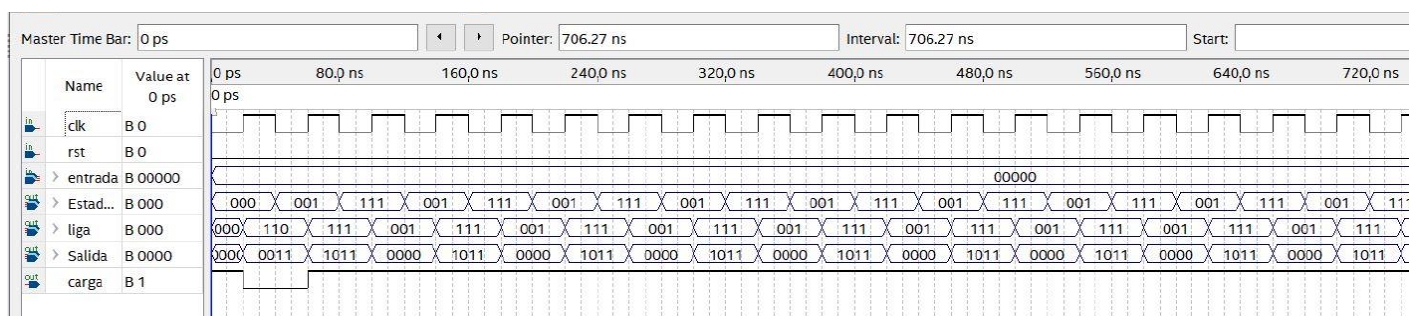


Figura 7: Simulación

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 Ingeniería.

Índice de tablas

1	Relación VF y Q	3
2	Valores binarios de estados	5
3	Valores binarios de variables	5
4	Tabla de verdad de la carta ASM	5

Índice de figuras

1	Diagrama de direccionamiento implícito	3
2	Direccionamiento Implícito.	3
3	Diagrama Modificado	4
4	Carta ASM	4
5	Diagrama	9
6	Simulación	10
7	Simulación	10

Índice de Códigos

1.	memoria.vhd	5
2.	separador.vhd	6
3.	contador.vhd	7
4.	logica.vhd	7
5.	selector_salida.vhd	8