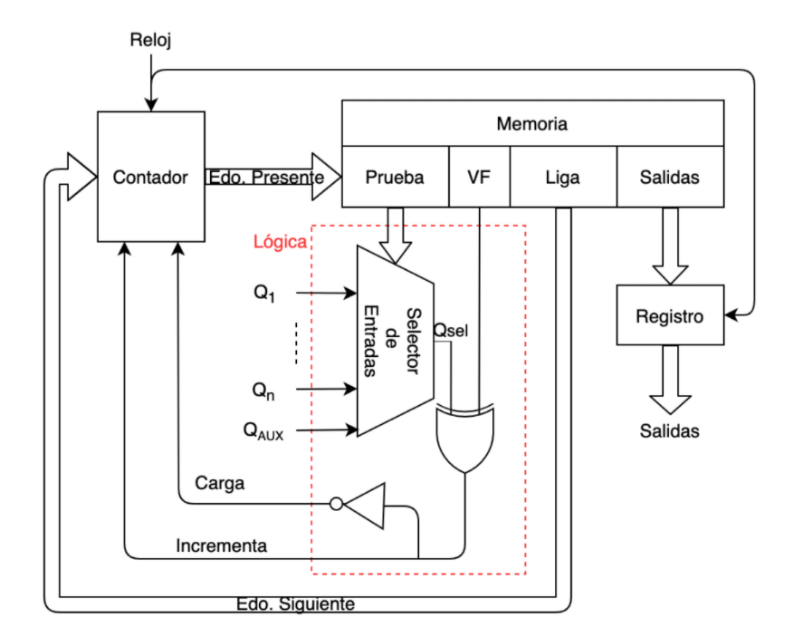
# **Objetivo**

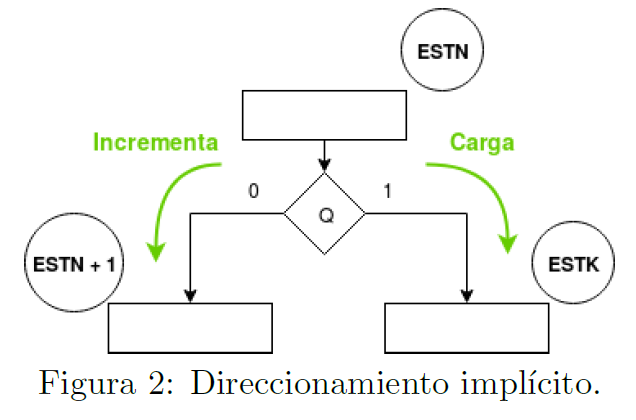
Familiarizar al alumno en el conocimiento de construcción de máquinas de estados usando direccionamiento de memorias con el método de direccionamiento implícito.

# **Introducción**

Direccionamiento Implícito

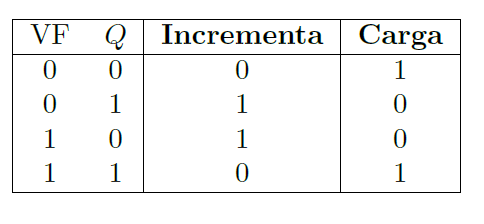
****Este tipo de direccionamiento utiliza solamente un campo de liga. Una variable de entrada seleccionada por el campo de prueba, y VF, son las que deciden si se utiliza la dirección de liga (se carga el valor de liga en el contador) o no (se incrementa el contador en una unidad).

*Figura 1 Diagrama del direccionamiento implícito*

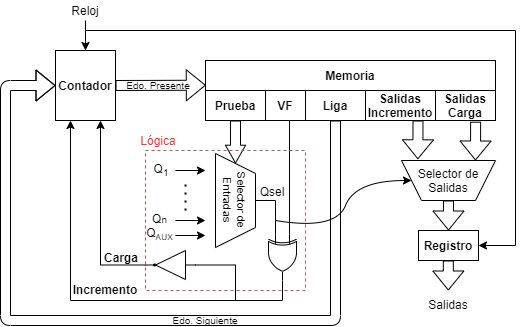
El campo VF (Verdadero-Falso) sirve para indicarle a la lógica cuánto debe valer la variable de entrada, para así cargar en el contador el valor de la liga y hacer el salto.

Supongamos que nos encontramos en el estado ESTN y la variable a censar es Q, si Q es igual a cero entonces el estado siguiente es la representación del estado presente más uno, por lo tanto, es necesario activar la señal de incremento. Si Q es igual a uno entonces el estado siguiente es el estado ESTK y su representación binaria, contenida en el campo de liga, es cargada en el contador. Para este ejemplo el campo VF es igual a uno ya que cuando Q es igual a uno se requiere hacer una carga en el contador.

Es necesario tomar precauciones al hacer la asignación binaria de los estados, porque se debe asegurar que por cada entrada censada existan dos estados siguientes: uno debe tener el valor del estado presente mas uno y el otro puede tomar cualquier otro valor.

La siguiente tabla muestra la relación de VF y la variable de entrada con las líneas de Incrementa y Carga.

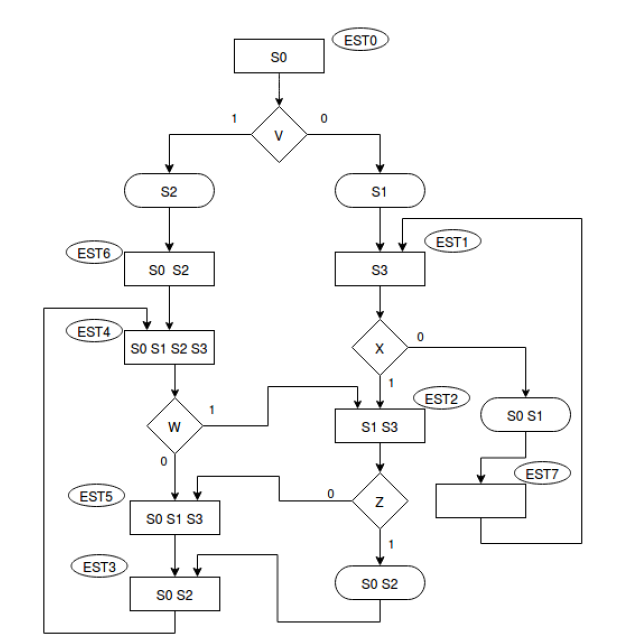
*Tabla 1 Tabla de la relación VF y Q para la Carga y el Incremento*

Sin embargo, tal y como está diseñado este tipo de direccionamiento, no se admiten salidas condicionales, es por esto que debe modificarse de la siguiente forma:

*Figura 3 Diagrama modificado del direccionamiento Implícito*

Des esta forma podremos implementar las salidas condicionales que nos ofrecen cierto tipo de cartas ASM sin ningún problema, tal y como se verá en el desarrollo de esta práctica.

# **Desarrollo**

Como primer punto, analizamos la carta ASM que se muestra a continuación, otorgando los valores binarios para cada uno de los estados.

*Figura 4 Carta ASM*

*Figura 1.0 Carta ASM.*

Analizando primeramente la carta ASM, asegurándonos de que cumple con la regla descrita en la introducción “N,N+1,P”, realizamos la tabla de verdad correspondiente al comportamiento que esta tiene. Asignando inicialmente los valores binarios para cada uno de los estados y de los valores de prueba para identificarlos de manera correcta





*Figura 4.1 Valores binarios de los estados y las entradas.*

Con lo cual obtuvimos la siguiente tabla de verdad

*Figura 4.2 Tabla de verdad de la carta ASM.*

Con los valores obtenidos, implementamos la memoria en Quartus con la programación VHDL obteniendo el siguiente código:

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.std\_logic\_arith.ALL;

USE ieee.std\_logic\_unsigned.ALL;

ENTITY rom IS

   PORT (

       addr : IN std\_logic\_vector(2 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 rom;

ARCHITECTURE behavioral OF rom IS

   TYPE mem\_rom IS ARRAY(0 TO 7) OF std\_logic\_vector(14 DOWNTO 0);

   SIGNAL data\_out : mem\_rom;

   SIGNAL data : std\_logic\_vector(14 DOWNTO 0);

BEGIN

       data\_out(0) <= "000111011001010";

       data\_out(1) <= "010011100011101";

       data\_out(2) <= "011010111110101";

       data\_out(3) <= "100010010101010";

       data\_out(4) <= "001101011111111";

       data\_out(5) <= "100001111011101";

       data\_out(6) <= "100010010101010";

       data\_out(7) <= "100000100000000";

   PROCESS (addr) BEGIN

       data <= data\_out(conv\_integer(unsigned(addr)));

       prueba <= data(14 downto 12);

       vf <= data(11);

       liga <= data(10 downto 8);

       salida\_incremento <=data(7 downto 4);

       salida\_carga <=data(3 downto 0);

   END PROCESS;

END behavioral;

*Figura 4.3 Código rom.vhd*

Posteriormente, implementamos la lógica mencionada en la introducción donde se obtendrá como resultado si se está pasando a un estado contiguo o se está haciendo un salto. En nuestro caso la XOR (Incremento) y la XNOR (Carga) la implementamos de manera directa en el código sin utilizar un bloque aparte, de esta forma se produjo el siguiente código:

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);

   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 (vf, 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 <= 'Z';

       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;

*Figura 4.4 Código logica.vhd*

Seguido de lo anterior implementamos el contador, el cual nos ayuda a seleccionar el estado siguiente dependiendo sí es un paso contiguo o un salto.

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.std\_logic\_arith.ALL;

USE ieee.std\_logic\_unsigned.ALL;

ENTITY rom IS

   PORT (

       addr : IN std\_logic\_vector(2 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 rom;

ARCHITECTURE behavioral OF rom IS

   TYPE mem\_rom IS ARRAY(0 TO 7) OF std\_logic\_vector(14 DOWNTO 0);

   SIGNAL data\_out : mem\_rom;

   SIGNAL data : std\_logic\_vector(14 DOWNTO 0);

BEGIN

       data\_out(0) <= "000111011001010";

       data\_out(1) <= "010011100011101";

       data\_out(2) <= "011010111110101";

       data\_out(3) <= "100010010101010";

       data\_out(4) <= "001101011111111";

       data\_out(5) <= "100001111011101";

       data\_out(6) <= "100010010101010";

       data\_out(7) <= "100000100000000";

   PROCESS (addr) BEGIN

       data <= data\_out(conv\_integer(unsigned(addr)));

       prueba <= data(14 downto 12);

       vf <= data(11);

       liga <= data(10 downto 8);

       salida\_incremento <=data(7 downto 4);

       salida\_carga <=data(3 downto 0);

   END PROCESS;

END behavioral;

*Figura 4.5 Código contador.vhd*

Como se mencionó anteriormente, para poder llevar a cabo la correcta selección de las salidas, dado que recibimos tanto las salidas de incremento como la de las salidas de carga, necesitamos implementar un selector de salida, dándonos el siguiente código:

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

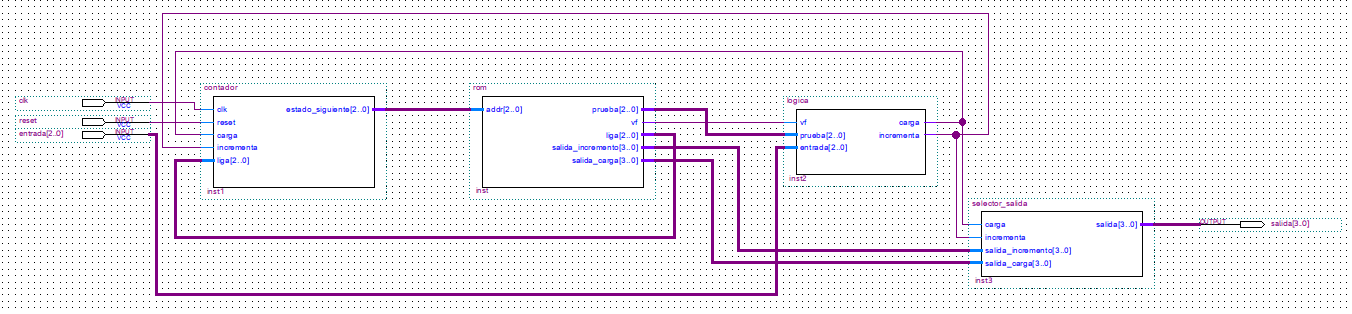
           salida <= "1000";

       END IF;

   END PROCESS;

END Behavioral;

*Figura 4.6 Código selector\_entrada.vhd*

Una vez realizado cada uno de los componentes necesarios para llevar a cabo el direccionamiento implícito, obtuvimos cada uno de sus símbolos para poder unirlos en un esquema, obteniendo el siguiente resultado:

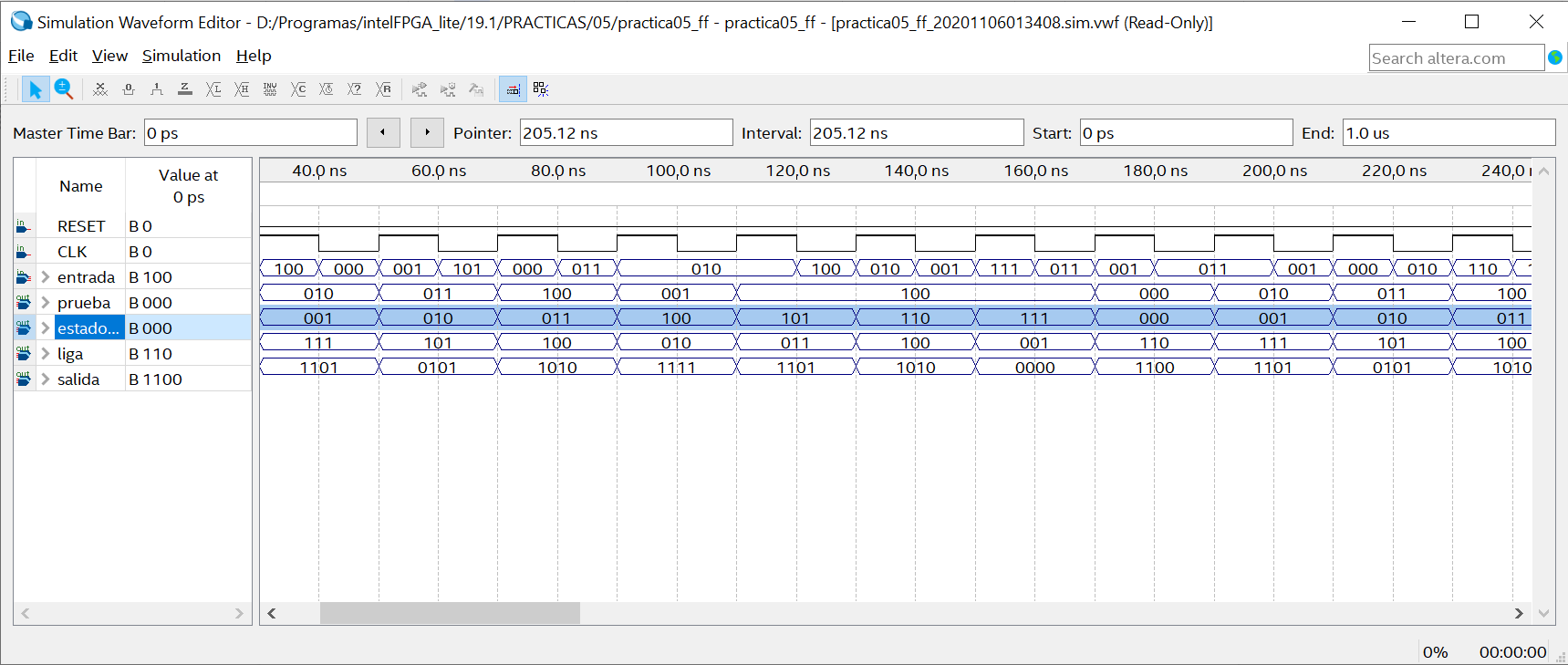
***selector\_salida***

***selector\_liga***

***contador***

***rom***

*Figura 4.7 Esquema de la máquina de estados por direccionamiento implícito*

Como podemos observar se sigue prácticamente el diagrama mostrado en la introducción que pertenece al direccionamiento implícito. Ahora bien, una vez compilado el bloque entero y asignado cada una de las entradas, así como la señal de reloj y de Reset, procedemos a simularlo

*Figura 4.8 Simulación del Esquemático*

Como podemos ver en la simulación se observan las señales de reloj (*clk*), reset, entradas, estado actual (*estado*), estado siguiente (*liga*), el valor de prueba (*prueba*) y finalmente las salidas (*salida*).

Siguiendo la carta ASM y la tabla de verdad junto con los valores de las entradas, podemos comprobar que efectivamente se lleva a cabo de manera correcta el funcionamiento de la maquina de estados por direccionamiento implícito. A diferencia de simulaciones pasadas, dado que aquí manejamos las salidas tanto de carga como de incremento utilizando un contador, no se logra observar tal cual un orden de seguimiento en cada ciclo de reloj, sin embargo, todos los valores de las salidas corresponden al estado que se está analizando. Por lo que la simulación es correcta ya que nos indica de buena manera el estado siguiente al que se debe de ir y la salida que se obtuvo con esta.

# **Fuentes de Consulta**

* Savage, J (2015) Diseño de microprocesadores. Facultad de Ingeniería. Universidad Nacional Autónoma de México. 482pp,
* Jacinto, E. (2013) Implementación de máquinas de estados basadas en rom. Consultado el 05 de noviembre del 2020 de: <https://dialnet.unirioja.es/descarga/articulo/5038459.pdf>
* Chávez, N (S.F) Construcción de máquinas de estados usando memorias. Consultado el 05 de noviembre del 2020 de: <http://profesores.fi-b.unam.mx/normaelva/Direccionamientos.pdf>