**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

**FACULTAD DE INGENIERÍA**

**LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS**

**CLAVE 6867**

Profesor: M. I. José Luis Cruz Mora

Integrantes del equipo:

García Fernández Jesús Alejandro

Hernández Arrieta Carlos Alberto

Moreno Guerra Marco Antonio

**Reporte Práctica 3**

Fecha de entrega: 11 de septiembre del 2019

Grupo: 1

Semestre: 2020-1

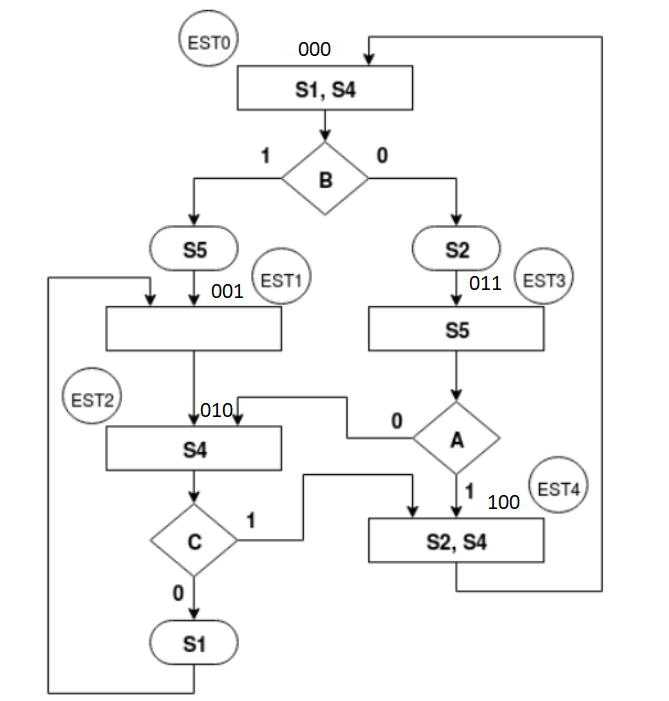
DIRECCIONAMIENTO POR TRAYECTORIA

OBJETIVO

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

DESARROLLO

A partir de la siguiente carta ASM se procedió a obtener la tabla del contenido de la memoria.

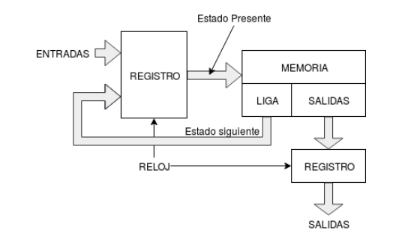


Carta ASM a resolver



Tabla del contenido de la memoria

Una vez obtenido el contenido de la memoria y tratándose del direccionamiento por trayectoria tenemos la siguiente arquitectura:



A partir de la arquitectura anterior y con ayuda de Quartus desarrollamos en VHDL cada uno de los componentes que conforman esta arquitectura de la siguiente manera:

La memoria se conforma del contenido obtenido en la tabla anterior:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity memory is

    Port( dir : in std\_logic\_vector(5 downto 0);

            data : out std\_logic\_vector(6 downto 0)

            );

end memory;

architecture Behavioral of memory is

    type mem is array(0 to 63) of std\_logic\_vector(6 downto 0);

    signal internal\_mem : mem;

    begin

        -- ESTADO 0

        internal\_mem(0) <= "011" & "1110";

        internal\_mem(1) <= "011" & "1110";

        internal\_mem(2) <= "001" & "1011";

        internal\_mem(3) <= "001" & "1011";

        internal\_mem(4) <= "011" & "1110";

        internal\_mem(5) <= "011" & "1110";

        internal\_mem(6) <= "001" & "1011";

        internal\_mem(7) <= "001" & "1011";

        -- ESTADO 1

        internal\_mem(8) <= "010" & "0000";

        internal\_mem(9) <= "010" & "0000";

        internal\_mem(10) <= "010" & "0000";

        internal\_mem(11) <= "010" & "0000";

        internal\_mem(12) <= "010" & "0000";

        internal\_mem(13) <= "010" & "0000";

        internal\_mem(14) <= "010" & "0000";

        internal\_mem(15) <= "010" & "0000";

        -- ESTADO 2

        internal\_mem(16) <= "001" & "1010";

        internal\_mem(17) <= "100" & "0010";

        internal\_mem(18) <= "001" & "1010";

        internal\_mem(19) <= "100" & "0010";

        internal\_mem(20) <= "001" & "1010";

        internal\_mem(21) <= "100" & "0010";

        internal\_mem(22) <= "001" & "1010";

        internal\_mem(23) <= "100" & "0010";

        -- ESTADO 3

        internal\_mem(24) <= "010" & "0001";

        internal\_mem(25) <= "010" & "0001";

        internal\_mem(26) <= "010" & "0001";

        internal\_mem(27) <= "010" & "0001";

        internal\_mem(28) <= "100" & "0001";

        internal\_mem(29) <= "100" & "0001";

        internal\_mem(30) <= "100" & "0001";

        internal\_mem(31) <= "100" & "0001";

            -- ESTADO 4

        internal\_mem(32) <= "000" & "0110";

        internal\_mem(33) <= "000" & "0110";

        internal\_mem(34) <= "000" & "0110";

        internal\_mem(35) <= "000" & "0110";

        internal\_mem(36) <= "000" & "0110";

        internal\_mem(37) <= "000" & "0110";

        internal\_mem(38) <= "000" & "0110";

        internal\_mem(39) <= "000" & "0110";

        process(dir)

            begin

                data <= internal\_mem(conv\_integer(unsigned(dir)));

        end process;

end Behavioral;

Posteriormente de esta memoria obtenemos la liga y las salidas, que nos darán nuestro estado siguiente y las salidas que se encenderán pero de la memoria estos datos salen concatenados, es decir, los obtenemos por medio de una misma cadena de bits de 7 bits, por lo tanto tendremos que separar los datos para poder procesar cada parte de forma independiente, realizando el siguiente código que nos regresa la liga y las salidas por separado:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity divisor\_datos is

    Port ( entrada : in STD\_LOGIC\_VECTOR (6 downto 0);

             liga : out STD\_LOGIC\_VECTOR (2 downto 0);

             salidas : out STD\_LOGIC\_VECTOR(3 downto 0));

end divisor\_datos;

architecture Behavioral of divisor\_datos is

begin

    process(entrada)

    begin

        liga <= entrada(6 downto 4);

        salidas <= entrada(3 downto 0);

    end process;

end Behavioral;

Nuestra liga ahora llega al registro que le dará estabilidad y le permitirá convertirse en el estado presente, se implementó de la siguiente manera:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity register2 is

    Port ( CLK : in STD\_LOGIC;

             RESET : in STD\_LOGIC;

             DATA\_IN : in STD\_LOGIC\_VECTOR(2 downto 0);

             DATA\_OUT : out STD\_LOGIC\_VECTOR(2 downto 0));

end register2;

architecture Behavioral of register2 is

signal internal\_value : std\_logic\_vector (2 downto 0) := B"000";

begin

    process (CLK, RESET, DATA\_IN)

    begin

        if RESET = '0' then

            internal\_value <= B"000";

        elsif rising\_edge (CLK) then

            internal\_value <= DATA\_IN;

        end if;

    end process;

    process (internal\_value)

    begin

        DATA\_OUT <= internal\_value;

    end process;

end Behavioral;

Como se puede observar, se le agrego una entrada Reset, ésta nos permitirá regresar al estado inicial cada que presionemos un botón.

Posteriormente necesitamos para ingresar a la memoria y descubrir cuál será el salto siguiente las entradas y el estado presente, esta información viene de forma independiente, por lo cual necesitamos un módulo concatenador de estos datos que nos permitirán ser procesados en la memoria:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity concatenador\_datos is

    Port ( entradas : in STD\_LOGIC\_VECTOR (2 downto 0); -- Entradas

             liga : in STD\_LOGIC\_VECTOR (2 downto 0); -- Liga

             salida : out STD\_LOGIC\_VECTOR(5 downto 0));

end concatenador\_datos;

architecture Behavioral of concatenador\_datos is

begin

    process(entradas, liga)

    begin

        salida <= liga & entradas; -- Liga & Entradas

    end process;

end Behavioral;

El funcionamiento de la arquitectura como tal esta completa hasta aquí, pero con el objetivo de poder probar su correcto funcionamiento en la fpga Max 10 se configuraron dos módulos más, el primero de ellos ya fue utilizado en la práctica pasada, el módulo nos permite sustituir los flancos automáticos del reloj de la tarjeta por flancos de subida cada que presionemos un botón, lo cual nos permitirá apreciar a nuestro ritmo como se va avanzando a través de los estados:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity sensa\_boton is

    Port (BOTON : in STD\_LOGIC;

        CLK : in STD\_LOGIC;

        RELOJ : out STD\_LOGIC;

        EPRESENTE: buffer STD\_LOGIC);

end sensa\_boton;

architecture Behavioral of sensa\_boton is

    signal ESIGUIENTE: STD\_LOGIC;

        begin

        process (ESIGUIENTE, BOTON)

            begin

            if rising\_edge (CLK) then

                case ESIGUIENTE is

                    when '0' =>

                        RELOJ <= '0';

                        if BOTON ='0' then

                            ESIGUIENTE <= '0';

                        else

                            ESIGUIENTE <= '1';

                        end if;

                    when '1' =>

                        if BOTON ='1' then

                            ESIGUIENTE <= '1';

                            RELOJ <= '0';

                        else

                            ESIGUIENTE <= '0';

                            RELOJ <= '1';

                        end if;

                    when others => null;

                end case;

            end if;

            EPRESENTE <= ESIGUIENTE;

end process;

end Behavioral;

Y por último, agregamos un módulo que nos permite visualizar en los leds de la fpga dos diferentes casos dependiendo la posición de un switch, en el primero podemos ver el estado presente con las salidas y en el segundo caso el estado siguiente y las salidas:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity switchSalidas is

    Port ( ContenidoM : in STD\_LOGIC\_VECTOR (6 downto 0); -- Entradas

             Presente : in STD\_LOGIC\_VECTOR (5 downto 0); -- Liga

             Switch : in std\_logic;

             Salidas: in STD\_LOGIC\_VECTOR (3 downto 0);

             salida : out STD\_LOGIC\_VECTOR(6 downto 0));

end switchSalidas;

architecture Behavioral of switchSalidas is

begin

    process(ContenidoM, Presente, Switch, Salidas)

    begin

        if Switch = '0' then

            salida <= ContenidoM;

        else

            salida <= Presente(5 downto 3) & Salidas;

        end if;

    end process;

end Behavioral;

CONCLUSIONES

García Fernández Jesús Alejandro: En esta práctica implementamos la primera forma de direccionamiento vistas en la clase de teoría, ahí aprendí que este modo de direccionamiento puede resolver cualquier problema que se pueda modelar por medio de una carta ASM, realizando este direccionamiento de forma práctica me enfrente a varias dudas que no me podía enfrentar en teoría, al momento de pasar el programa a la tarjeta nos dimos cuenta que automáticamente el comportamiento era irse por el camino donde la variable condicional valía 0, al principio pensamos que era un error en la tabla de direccionamiento, pero con ayuda del profesor, logramos analizar y darnos cuenta que al momento de iniciar