

# **Modelado y Síntesis de Sistema Electrónicos Digitales**

**GIC**

**Universidad de Alcalá de Henares**

**Memoria**

Pedro Barquín Ayuso

# Indice

• Cnt_Epp		
Información.....Pg	3-4	
Circuitos Digitales.....Pg	5-6	
Código Programa.....Pg	7-9	
Código TestBench.....Pg	10-14	
Simulación Funcional.....Pg	15-16	
Simulación Temporales.....Pg	16-17	
Recursos Utilizados.....Pg	18	
• Top_System		
Información.....Pg	19-20	
Circuitos Digitales.....Pg	21	
Código Programa.....Pg	22-23	
Código TestBench.....Pg	24-25	
Simulación Funcional.....Pg	26	
Simulación Temporales.....Pg	26	
Recursos Utilizados.....Pg	27	
Generación y Test en placa.....Pg	28-30	
• Cnt_Dac		
Información.....Pg	31-32	
Circuitos Digitales.....Pg	33-34	
Código Programa.....Pg	35-39	
Código TestBench.....Pg	40	
Simulación Funcional.....Pg	41	
Simulación Temporales.....Pg	41	
Recursos Utilizados.....Pg	42	
Verificación con componente DAC121S101.vhdl.....Pg	43-45	
• Dpram_Mem		
Información.....Pg	46	
Circuitos Digitales.....Pg	47	
Código Programa.....Pg	48	
Código TestBench.....Pg	49-50	
Simulación Funcional.....Pg	51	
Simulación Temporales.....Pg	51	
Recursos Utilizados.....Pg	53	
• Cnt_Dpram		
Información.....Pg	54	
Circuitos Digitales.....Pg	55	
Código Programa.....Pg	56-58	
Código TestBench.....Pg	59-61	
Simulación Funcional.....Pg	62	
Simulación Temporales.....Pg	63	
Recursos Utilizados.....Pg	64	
• Gen_Dir		
Información.....Pg	65-66	
Circuitos Digitales.....Pg	67-68	
Código Programa.....Pg	69-71	
Código TestBench.....Pg	72-73	
Simulación Funcional.....Pg	74	
Simulación Temporales.....Pg	75	
Recursos Utilizados.....Pg	76	
• Gen_Funciones		
Información.....Pg	77	
Generación DCM.....Pg	78	
Asignación Archivos y Device 1.....Pg	79	
Código Programa.....Pg	80-82	
Código TestBench.....Pg	83-84	
Simulación Funcional.....Pg	85-86	
Simulación Temporales.....Pg	87	
Recursos Utilizados.....Pg	88	
Generación del archivo a descargar en placa.....Pg	89	
Descarga y Prueba en placa.....Pg	90	

## Memoria CNT EPP

En este apartado se busca modelar el contenido de cnt\_epp para ello sera necesario primero analizar las exigencias del código que se desea implementar, en este caso este modulo se encarga de interpretar las ordenes que obtiene desde el ordenador mediante el cable usb mediante el protocolo del DRIVER CY7C68013A con ello seremos capaces de comunicar el ordenador con el programa.

Las entradas y salidas de este apartado son las citadas a continuación ademas se especifica el tipo de dato que son y si son de entrada o de salida:

CLK : in std_logic;	Señal de Reloj
RST : in std_logic;	Señal de reset
ASTRB : in std_logic;	Activa a nivel bajo indica la transferencia de Dirección
DSTRB : in std_logic;	Indica transferencia de datos
DATA : inout std_logic_vector(7 downto 0);	Bus donde por donde se transmite la información
PWRITE : in std_logic;	Se encuentra escribiendo
PWAIT : out std_logic;	Señal de espera
DATO_RD : in std_logic_vector(7 downto 0);	Dato a ser leído en ciclo de lectura
CE_RD : out std_logic;	Habilita la lectura de Dato_Rd
DIR : out std_logic_vector(7 downto 0);	Valor de la dirección de un ciclo de acceso
DIR_VLD : out std_logic;	Validación de la dirección
DATO : out std_logic_vector(7 downto 0);	Valor correspondiente a lo pasado en un ciclo escritura
DATO_VLD : out std_logic;	Validación del dato

Señales auxiliares utilizadas en el programa para poder realizar las interconexiones entre las distintas partes del los componentes:

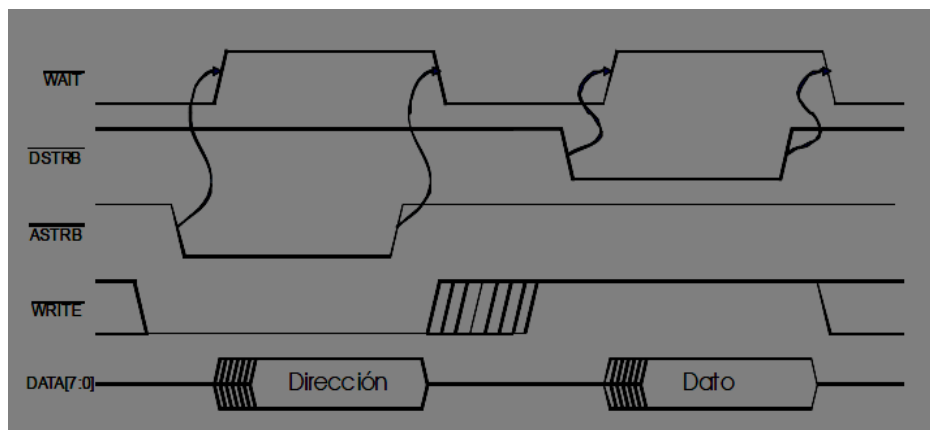
signal DFAS : std_logic ;	Detector de flanco para ASTRB
signal DFDS : std_logic ;	Detector de flanco para DSTRB
signal ASBD : std_logic ;	Salida ASTRB de biestable D
signal DSBD : std_logic ;	Salida DSTRB de biestable D

Este apartado se compone de dos ciclos o divisiones las cuales son Ciclo de lectura y ciclo de escritura las cuales se describen a continuación.

- Ciclo de lectura:

Se realiza cuando se quiere leer el contenido de una posición concreta del periférico/placa para ello este apartado se divide en dos partes:

1. Escritura de la dirección: El ciclo de escritura de la dirección se inicia cuando **PWRITE** pasa a nivel bajo, continuando con un nivel bajo en **ASTRB**, a lo que el periférico responde con un nivel alto en **PWAIT**. Durante el nivel bajo de **ASTRB** se proporciona la dirección del periférico a la que se quiere acceder. El ciclo de escritura de la dirección finaliza cuando **ASTRB** pasa a nivel alto (detección flanco de subida), instante que puede utilizar el periférico para leer el valor de la dirección.
2. Lectura de la dirección: En este apartado se ha de leer el dato que se ha pasado en el ciclo anterior para ello el dato debe estar estable antes de que se produzca el flanco de subida de **DSTRB** momento en el cual realizamos la operación de transferencia. El periférico utilizar

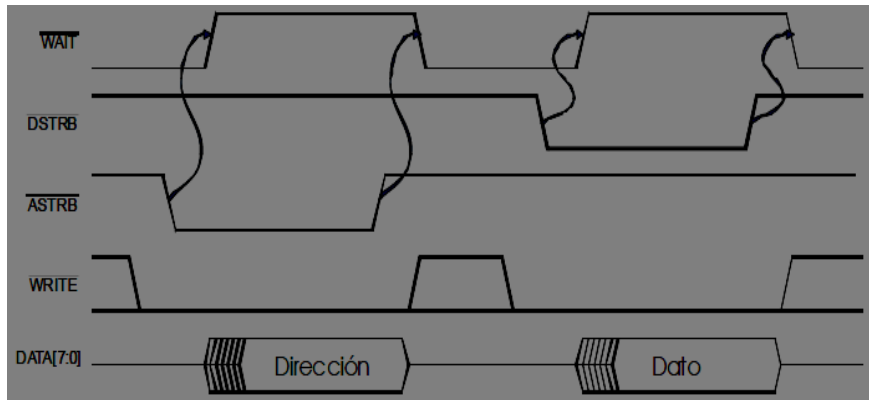


el flanco de bajada de esta señal para poner el dato en el puerto **DATA**.

- Ciclo de Escritura:

Un ciclo de escritura es aquel que realiza el PC para escribir un dato en una posición del periférico. En este caso, se realizan dos ciclos de escritura:

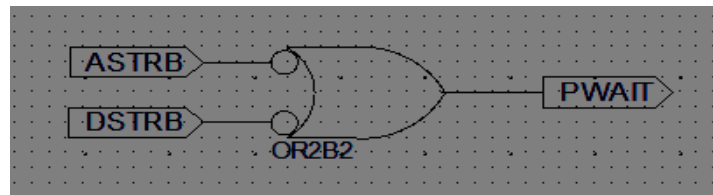
1. Escritura dirección: Este apartado es igual al del escritura del apartado anterior por lo que la explicación es la misma.
2. Escritura del dato: Este apartado es similar al de lectura pero en este caso se mantiene la señal **PWRITE** a bajo nivel para que quede reflejado que estamos escribiendo a la vez que el PC pone el dato a escribir en el periférico en **DATA**. El periférico puede utilizar el flanco de subida de **DSTRB** para leer el dato.



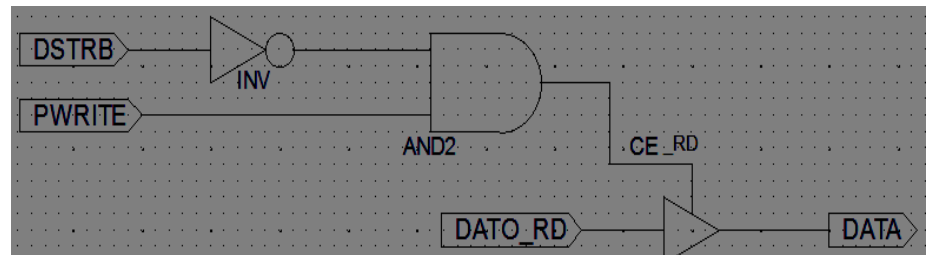
Con esta información, lo primero que hemos de realizar son los esquemas que realizan las acciones necesarias para poder realizar las funciones especificadas arriba, por ello es de vital importancia comprender correctamente la funcionalidad de el sistema.

### Circuitos digitales necesarios

**Presente en:** Ciclo de lectura y escritura  
**Función:** Una vez que ASTRB o DSTRB pasan a nivel bajo pone a nivel alto la señal PWAIT y viceversa.



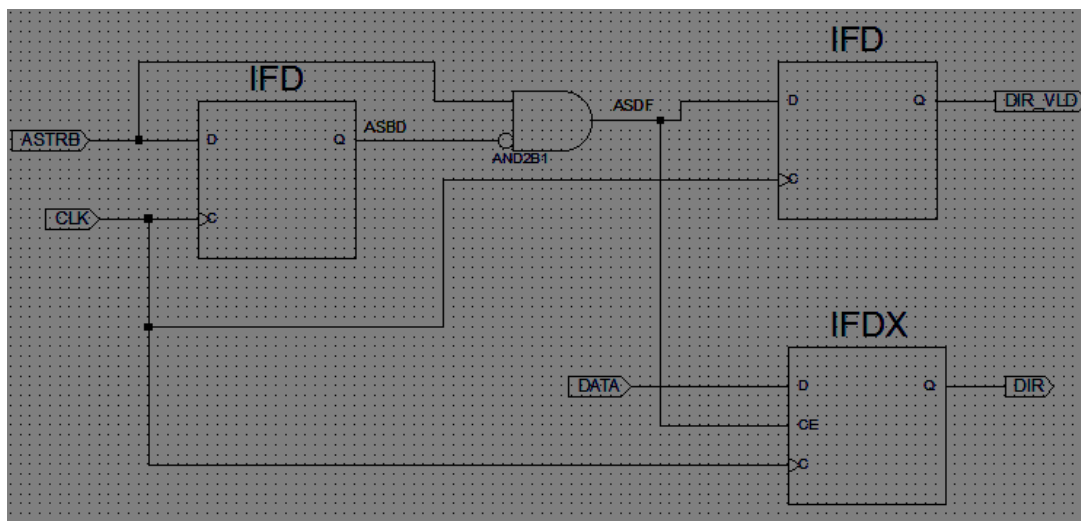
**Presente en:** Ciclo de lectura  
**Función:** Su función es una vez DSTRB esta a nivel bajo y PWRITE esta activo, Se cumple la condición por la cual se pasa la señal CE\_RD a alto nivel la cual permite mediante un triestate la transferencia de la información que hay en DATO\_RD a DATA mientras que esta señal se mantenga una vez finalizada DATA pasa a alta impedancia.



**Presente en:** Ciclo de Lectura y Escritura

**Función:** Su función es la detectar el el flanco de subida de la señal ASTRB lo cual se consigue gracias al biestable de y a la puerta and, ese pulso que se genera, durante el próximo ciclo de reloj pasara a DIR\_VLD (aviso de que hay información disponible) durante solo un pulso, ademas servirá para permitir la transferencia de DATA a DIR activando el CE del biestable d.

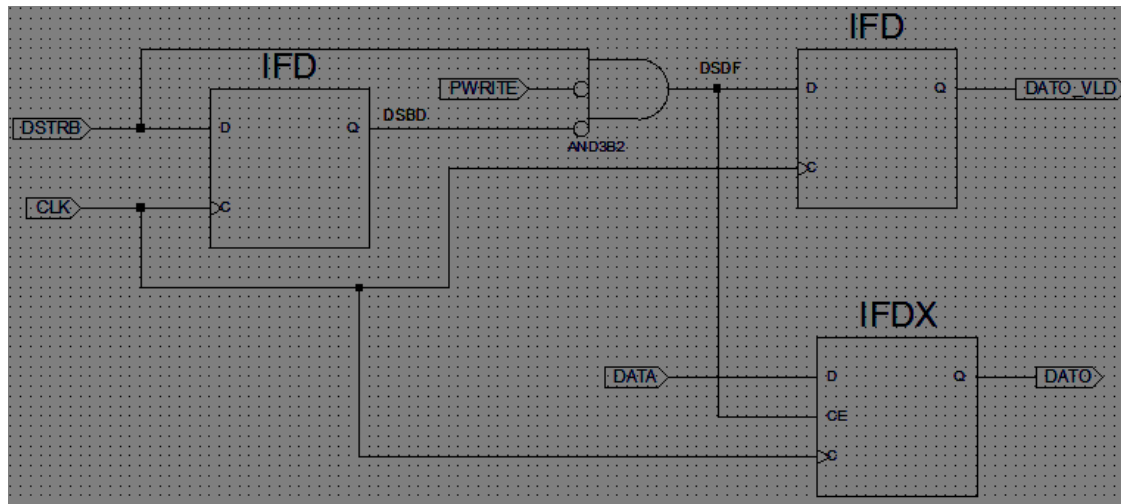
**Otros:** Son necesarias 2 señales auxiliares para su uso estas señales son ASBD (AStb Biestable D) y ASDF (AStb Detector Flanco)



**Presente en:** Ciclo de Escritura

**Función:** Su función es la detectar el el flanco de subida de la señal DSTRB lo cual se consigue gracias al biestable de y a la puerta and ademas cuenta con la señal PWRITE para evitar su activación en otro posibles casos, ese pulso que se genera, durante el próximo ciclo de reloj pasara a DATO\_VLD (aviso de que hay información disponible) durante solo un pulso, ademas servirá para permitir la transferencia de DATA a DATO activando el CE del biestable d.

**Otros:** Son necesarias 2 señales auxiliares para su uso estas señales son DSBD (**D**Strb **B**iestable **D**) y DSDF (**D**Strb **D**etector **F**lanco)



Una vez realizados y repasados los circuitos digitales, podemos pasarlo a VHDL modelando dichos circuitos mediante el código correspondiente, para ello realizamos los mismos pasos que hacemos siempre para crear el archivo y añadirlo al programa. Una vez realizados los paso y mediante el programa EMCS procedemos a realizar la conversión a código de dichos circuitos, el código mostrado a continuación es la implementación del apartado cnt\_epp.

### Código cnt\_epp

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity cnt_epp is
  port (
    CLK    : in  std_logic;
    RST    : in  std_logic; --Señal de rest a 0 en reposo y 1 activa
    ASTRB  : in  std_logic; --Señal a 1 en reposo y 0 activa
    DSTRB  : in  std_logic; --Señal a 1 en reposo y 0 activa
    DATA  : inout std_logic_vector(7 downto 0);
    PWRITE  : in  std_logic; --Señal a 1 en reposo y 0 activa
    PWAIT  : out  std_logic;
    DATO_RD : in  std_logic_vector(7 downto 0);
    CE_RD   : out  std_logic;
    DIR     : out  std_logic_vector (7 downto 0);
    DIR_VLD : out  std_logic;
    DATO    : out  std_logic_vector (7 downto 0);
    DATO_VLD : out  std_logic);
end ;

architecture rtl of cnt_epp is

  --Salidas aux para los detectores de flanco
  signal DFAS : std_logic ; --Señal aux para detector de flanco ASTRB
  signal DFDS : std_logic ; --Señal aux para detector de flanco DSTRB
  signal ASBD : std_logic ; --Señal aux de la salida del biestable d de ASTRB
  signal DSBD : std_logic ; --Señal aux de la salida del biestable d de DSTRB

begin

  -- Pone a 1 o 0 PWAIT en función de las entradas para mandar esperar en cualquiera de los casos
  PWAIT <= '1' when (ASTRB='0' or DSTRB='0') else '0';

  -- purpose: Transfiere el contenido de DATO_RD a DATA
  process (DSTRB,PWRITE, DATO_RD) is
  begin -- process
    if DSTRB='0' and PWRITE='1' then
      CE_RD <= '1';          --Pone a 1 la señal la cual permite la transferencia del dato_Rd
      DATA <= DATO_RD;      --Copia el contenido de DATP_RD a DATA
    else
      CE_RD <= '0';          --Si no se cumple no permite la copia del dato
      DATA <= (others => 'Z'); --Se implementa alta impedancia
    end if;
  end process;

  -- purpose: Detector de flanco de subida de la señal ASTRB parte 1/3
  process (CLK, RST) is
  begin -- process
    if RST = '1' then        -- asynchronous reset (active high)
      ASBD <= '1';           --Señal que ponemos a 1 en caso de rest ya que si la ponemos a 0
    produce un estado no deseado
  end if;
end process;
```

```

    elsif CLK'event and CLK = '1' then -- rising clock edge
        ASBD <= ASTRB;      -- Se pasa la información cada flanco de reloj a la salida del biestable
    end if;
end process;

-- Detector flanco, cuando ocurre el flanco y se complete la condición pone a 1 la señal de
detección
DFAS <= '1' when (ASTRB='1' and ASBD='0') else '0';

-- purpose: ASTRB parte 2/3 encargada de pasar el contenido de la detección del flanco a dir_vld
process (CLK, RST) is
begin -- process
    if RST = '1' then      -- asynchronous reset (active high)
        DIR_VLD <= '0';
    elsif CLK'event and CLK = '1' then -- rising clock edge
        DIR_VLD <= DFAS;   --pasa el contenido de DFAS a DIR_VLD
    end if;
end process;

-- purpose: ASTRB parte 3/3 encargada de pasar y almacenar el contenido de data en dir solo
cuando se cumpla la condición de detección del flanco
process (CLK, RST) is
begin -- process
    if RST = '1' then      -- asynchronous reset (active high)
        DIR <= (others => '0'); --Pone a 0 el vector cuando se hace reset
    elsif CLK'event and CLK = '1' then -- rising clock edge
        if DFAS='1' then
            DIR <= DATA;
        end if;
    end if;
end process;

-- purpose: Detector de flanco de subida de la señal DSTRB parte 1/3
process (CLK, RST) is
begin -- process
    if RST = '1' then      -- asynchronous reset (active high)
        DSBD <= '1';      --Señal que ponemos a 1 en caso de rest ya que si la ponemos a 0 produce un
estado no deseado
    elsif CLK'event and CLK = '1' then -- rising clock edge
        DSBD <= DSTRB;     -- Se pasa la información cada flanco de reloj a la salida del biestable
    end if;
end process;

-- detector de flanco, cuando ocurre el flanco y se complete la condición (en este caso es necesario
que pwrite este para poder diferenciar entre lectura y escritura) pone a 1 la señal de detección
DFDS <= '1' when (DSTRB='1' and DSBD='0' and PWRITE='0') else '0';

-- purpose: DSTRB parte 2/3 encargada de pasar el contenido de la detección del flanco a datp_vld
process (CLK, RST) is
begin -- process
    if RST = '1' then      -- asynchronous reset (active high)
        DATO_VLD <= '0';
    elsif CLK'event and CLK = '1' then -- rising clock edge

```



```

    DATO_VLD <= DFDS; --copia el contenido de la salida del detector de flanco a DATO_VLD
end if;
end process;

```

-- purpose: DSTRB parte 3/3 encargada de pasar y almacenar el contenido de data en dato solo cuando se cumpla la condición de detección del flanco

```

process (CLK, RST) is
begin -- process
    if RST = '1' then          -- asynchronous reset (active high)
        DATO <= (others => '0'); --Pone a 0 el vector cuando se hace reset
    elsif CLK'event and CLK = '1' then -- rising clock edge
        if DFDS='1' then --Si DFDS esta a 1 actual como CE para que solo se almacene en ese caso
            DATO <= DATA;      --Pasa el contenido de data a dato
        end if;
    end if;
end if;
end process;

end rtl;

```

\*\*Se adjuntan todos los archivos \*.VHDL para su posible observación en caso de no ser legible el código

Una vez realizado el apartado de la implementación de los circuitos a código se realiza la síntesis para comprobar que todo el contenido es correcto.

## TestBench

Después de ello realizamos el TestBench para verificar su correcto funcionamiento para ello creamos el archivo cnt\_epp\_tb en el cual implementamos toda la lógica necesaria para poder simular el comportamiento del componente, en este caso se nos suministra el archivo epp\_device que es el que simulara los datos mandados como si se tratara del ordenador, por lo que solo es necesario adjuntar lo dentro del TestBench para poder realizar simulación.

El código TestBench con todos los datos sería el citado continuación:

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

-----

entity cnt_epp_tb is

end entity cnt_epp_tb;

-----

architecture cnt_epp of cnt_epp_tb is
--delcaracion de los componentes utilizados para la simulación
component cnt_epp is
port (
    CLK      : in std_logic;
    RST      : in std_logic;
    ASTRB    : in std_logic;
    DSTRB    : in std_logic;
    DATA    : inout std_logic_vector(7 downto 0);
    PWRITE   : in std_logic;
    PWAIT    : out std_logic;
    DATO_RD  : in std_logic_vector(7 downto 0);
    CE_RD    : out std_logic;
    DIR      : out std_logic_vector (7 downto 0);
    DIR_VLD  : out std_logic;
    DATO     : out std_logic_vector (7 downto 0);
    DATO_VLD : out std_logic);
end component;

component epp_device is
port (
    DATA : inout std_logic_vector(7 downto 0);
    PWRITE : out std_logic;
    DSTRB  : out std_logic;
    ASTRB  : out std_logic;
    PWAIT  : in std_logic);
end component;

-- señales usadas para la simulación
signal CLK_i   : std_logic := '0';
signal RST_i   : std_logic := '1';
signal ASTRB_i : std_logic := '1';
```

```

signal DSTRB_i : std_logic := '1';
signal DATA_io : std_logic_vector(7 downto 0);
signal PWRITE_i : std_logic := '1';
signal PWAIT_o : std_logic;
signal DATO_RD_i: std_logic_vector(7 downto 0) := (others => '0');
signal CE_RD_o : std_logic;
signal DIR_o : std_logic_vector(7 downto 0) := (others => '0');
signal DIR_VLD_o: std_logic;
signal DATO_o : std_logic_vector (7 downto 0);
signal DATO_VLD_o: std_logic;

begin -- architecture cnt_epp

-- instanciación del componente cnt_epp con las señales
DUT: entity work.cnt_epp
port map (
    CLK    => CLK_i,
    RST    => RST_i,
    ASTRB  => ASTRB_i,
    DSTRB  => DSTRB_i,
    DATA  => DATA_io,
    PWRITE => PWRITE_i,
    PWAIT  => PWAIT_o,
    DATO_RD => DATO_RD_i,
    CE_RD  => CE_RD_o,
    DIR    => DIR_o,
    DIR_VLD => DIR_VLD_o,
    DATO    => DATO_o,
    DATO_VLD => DATO_VLD_o);

-- instanciación del componente epp_device con las señales
epp: entity work.epp_device
port map (
    DATA => DATA_io,
    PWRITE => PWRITE_i,
    DSTRB => DSTRB_i,
    ASTRB => ASTRB_i,
    PWAIT => PWAIT_o);

-- instanciación de las señales de rest reloj con sus frecuencias y usos
CLK_i <= not CLK_i after 5 ns;
RST_i <= '1', '0' after 25 ns;
--instanciación de la señal  DATO_RD_i para que cumpla unos parámetros en el tiempo
DATO_RD_i <= x"33" after 6250 ns, x"00" after 7250 ns;
end architecture cnt_epp;

```

**Código epp\_device con 3 ciclos escritura y 1 lectura**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity epp_device is
  port (
    DATA : inout std_logic_vector(7 downto 0);
    PWRITE : out  std_logic;
    DSTRB  : out  std_logic;
    ASTRB  : out  std_logic;
    PWAIT  : in   std_logic);

end epp_device;
architecture sim of epp_device is
  constant T_clk_epp      : time           := 100 ns; -- Internal clock period.
  signal clk_epp          : std_logic      := '0'; -- Internal clock signal.
  signal read_value       : std_logic_vector(7 downto 0) := (others => '0');
  constant dir_freq       : std_logic_vector(7 downto 0) := x"F0";
  constant dir_dpram1     : std_logic_vector(7 downto 0) := x"A1";
  constant dir_dpram2     : std_logic_vector(7 downto 0) := x"A2";
  constant EPP_cycle_length : natural      := 10;
begin
  -- internal clock signal generation.

  clk_epp <= not(clk_epp) after T_clk_epp/2;

  process
    procedure epp_cycle (address : in  std_logic_vector(7 downto 0);
                        data_io  : inout std_logic_vector(7 downto 0);
                        r_w      : in  character) is

    begin
      wait until clk_epp = '1';
      PWRITE <= '0';
      wait until clk_epp = '1';
      ASTRB  <= '0';
      data   <= address;
      wait for T_clk_epp*EPP_cycle_length;
      ASTRB  <= '1';
      wait until clk_epp = '1';
      data   <= (others => 'Z');
      PWRITE <= '1';
      wait until clk_epp = '1';
      wait for T_clk_epp*EPP_cycle_length;

      -----
      if r_w = 'w' then          -- write cicle
        PWRITE <= '0';
```

```

    data <= data_io;
end if;
-----
wait until clk_epp = '1';
DSTRB <= '0';
wait for T_clk_epp*EPP_cicle_length;
if r_w = 'r' then
    data_io := data;
end if;

DSTRB <= '1';
wait until clk_epp = '1';
data <= (others => 'Z');
PWRITE <= '1';
wait until clk_epp = '1';

end procedure;

file arch_in : text;
variable bf : line;
variable dato : std_logic_vector(7 downto 0);
variable dir : std_logic_vector(7 downto 0);
begin
--inicialización
data <= (others => 'Z');
PWRITE <= '1';
DSTRB <= '1';
ASTRB <= '1';
dir := (others => '0');
wait for 130 ns;
DIR := dir_dpram1;
DATO := X"34";
epp_cicle (address => dir,
           data_io => dato,
           r_w => 'w');

DIR := x"12";
epp_cicle (address => dir,
           data_io => dato,
           r_w => 'r');

read_value <= dato;

wait for 130 ns;
DIR := dir_freq;
DATO := X"22";
epp_cicle (address => dir,
           data_io => dato,
           r_w => 'w');

wait for 130 ns;

DIR := dir_dpram2;

```

```
DATO := X"11";  
epp_cicle (address => dir,  
           data_io => dato,  
           r_w    => 'w');  
  
wait for 1 us;  
  
report "FIN CICLO R/W" severity failure;  
  
end process;  
end sim;
```

## Simulación Funcional

Diagrama de simulación con las señales del programa:

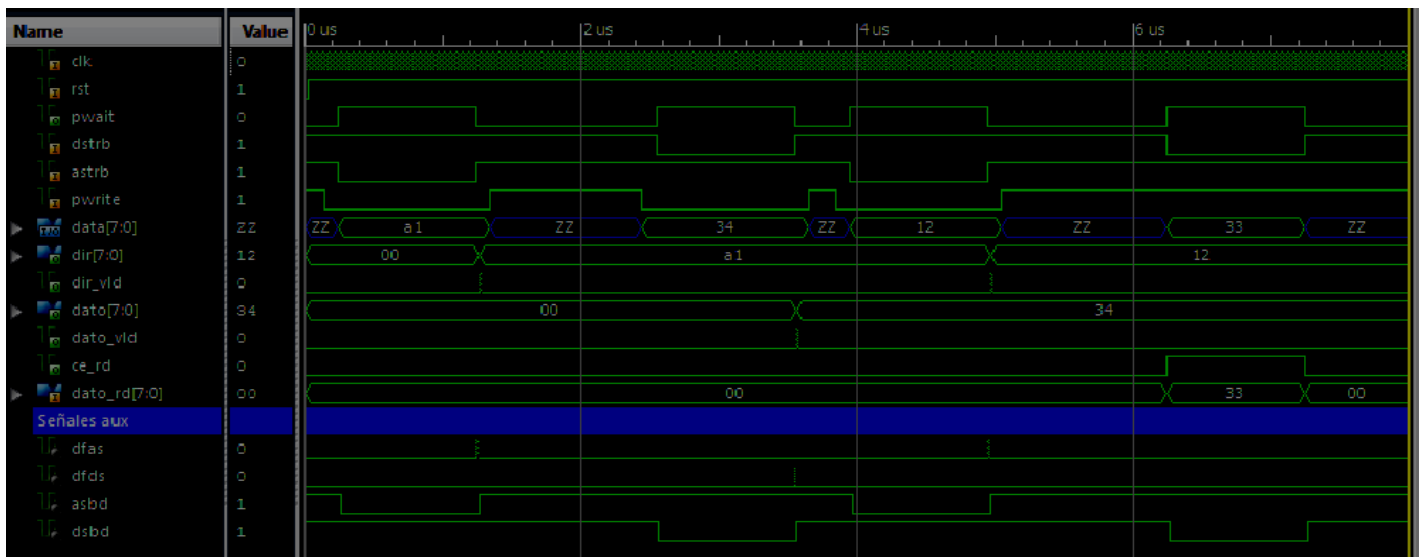
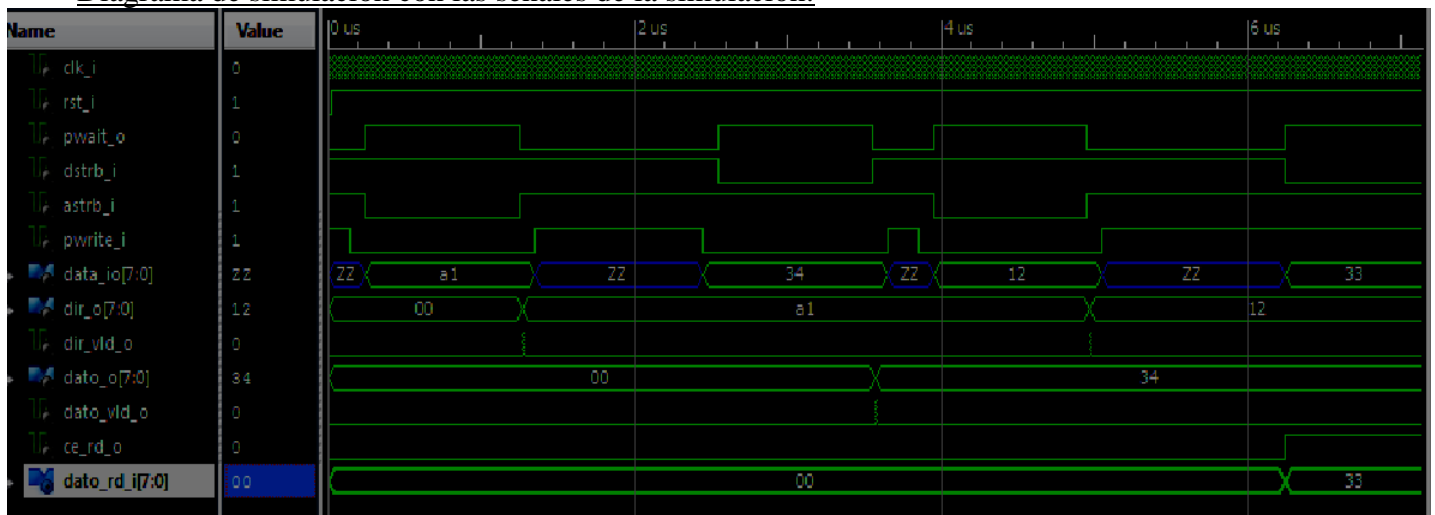


Diagrama de simulación con las señales de la simulación:



Una vez analizados los datos obtenidos de las gráficas y comparados con los de la practica si los resultados son satisfactorios podemos continuar con el siguiente paso.

Aumento del numero de ciclos de escritura para ello hemos de modificar el código suministrado por el profesor añadiendo dos ciclos mas de escritura con nuevos datos para que se muestren en la ejecución del programa, la información añadida al código se muestra a continuación:

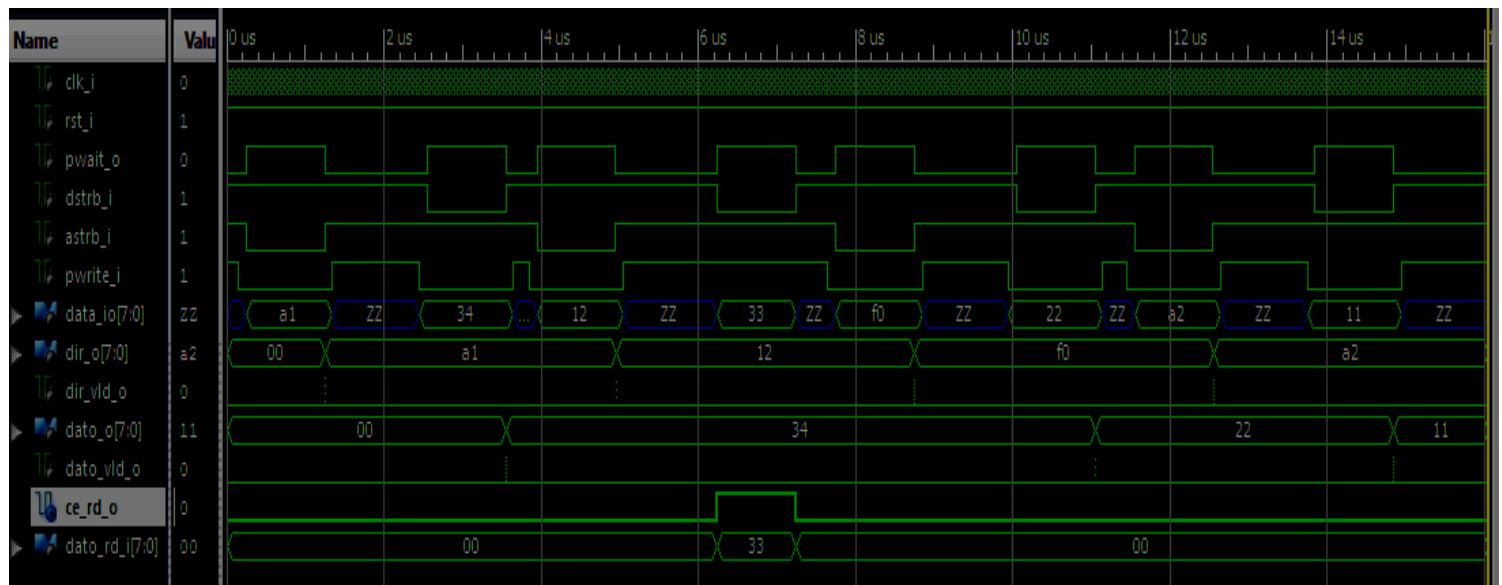
```
constant dir_freq      : std_logic_vector(7 downto 0) := x"F0";
constant dir_dpram2    : std_logic_vector(7 downto 0) := x"A2";
```

```
wait for 130 ns;
  DIR  := dir_freq;
  DATO := X"22";
  epp_cicle (address => dir,
             data_io => dato,
             r_w     => 'w');
```

```
wait for 130 ns;
  DIR  := dir_dpram2;
  DATO := X"11";
```

```
epp_cicle (address => dir,
           data_io => dato,
           r_w    => 'w');
```

Con eso añadimos la nueva información para poder transferir e iniciamos dos ciclos de escritura resultando el gráfico adjunto

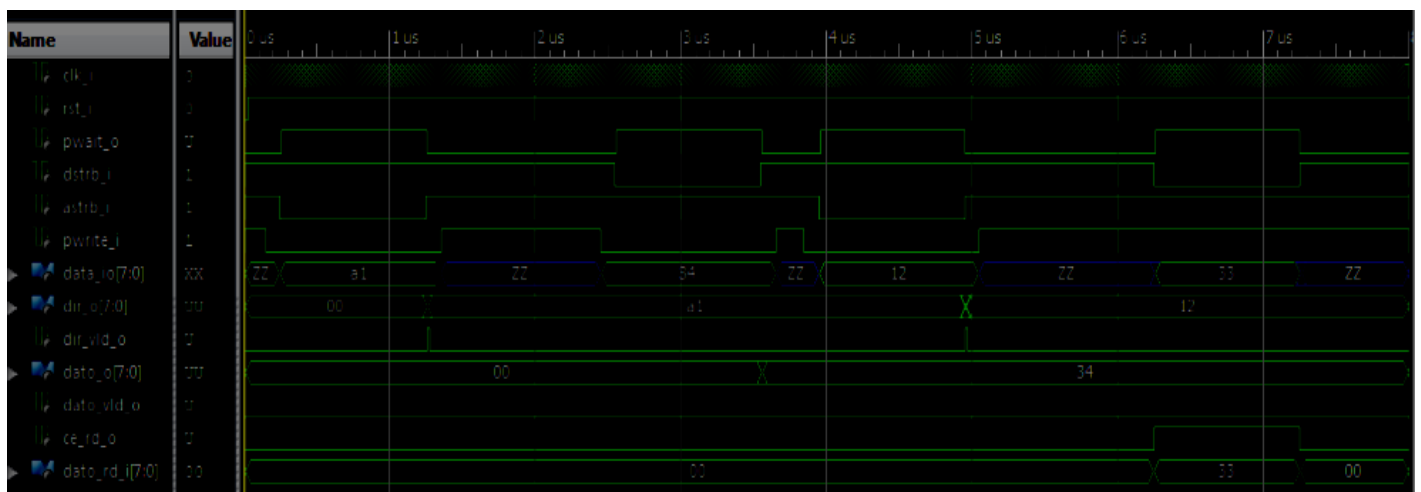


### Simulación Temporal

Después de verificar que los ciclos son correctos pasamos a realizar la simulación temporal para ello seguimos los paso explicados en clase:

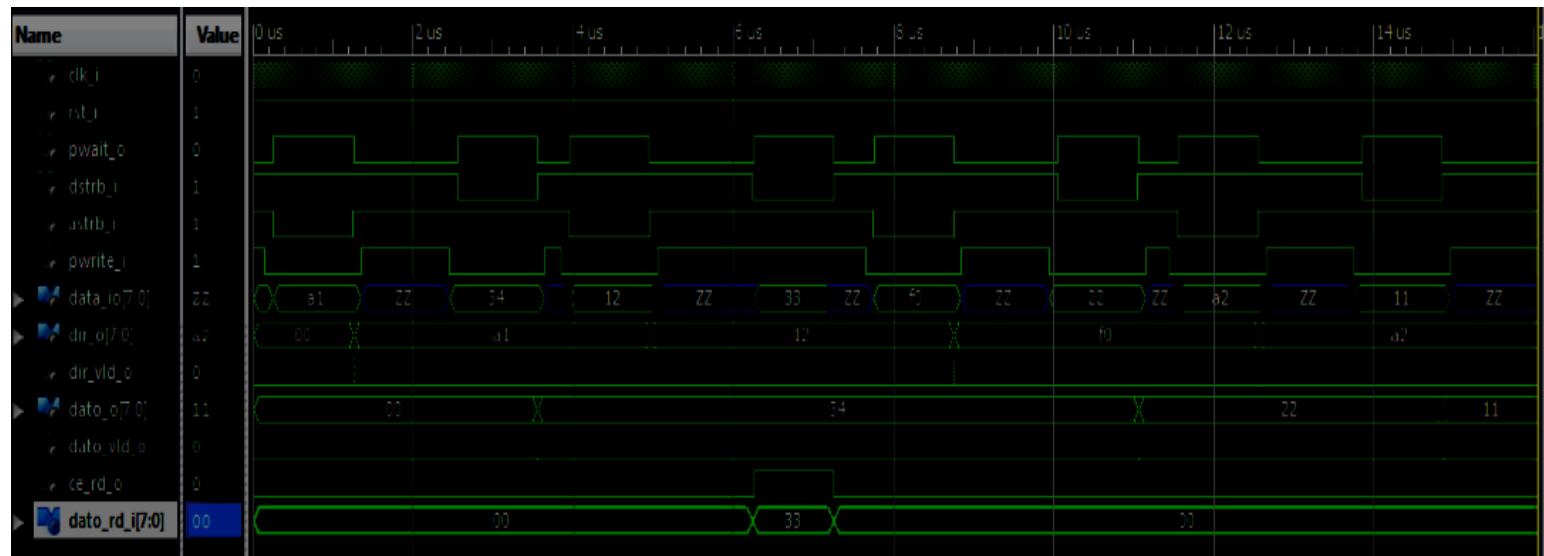
- Realizamos en el apartado de implementación el Post&Place and Route
- Cambiamos a la ventana de simulación
- Elegimos Post-Rute en el desplegable
- Chequeamos la síntesis del código
- Realizamos la simulación temporal en la cual se tienen en cuenta los retardos de las puertas lógicas y del resto de los componentes

### Simulación Temporal Ciclo de Escritura y Lectura





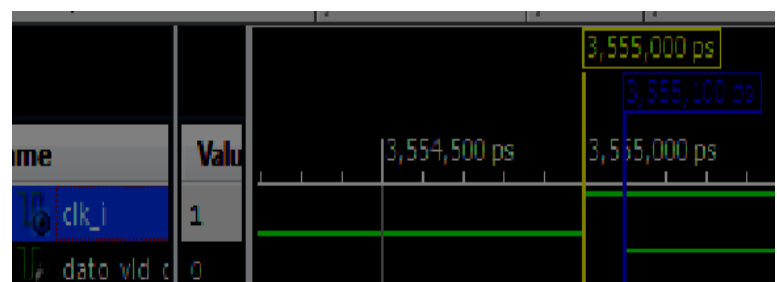
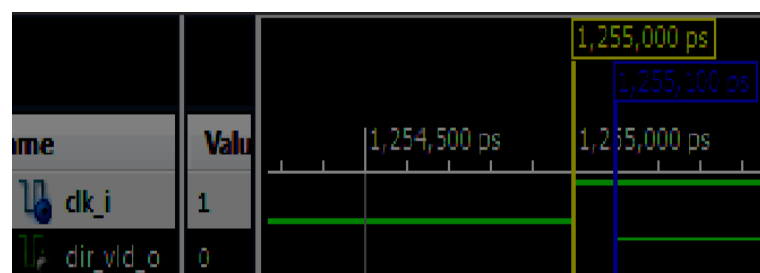
## Simulación Temporal 3 Ciclos de Escritura y Lectura



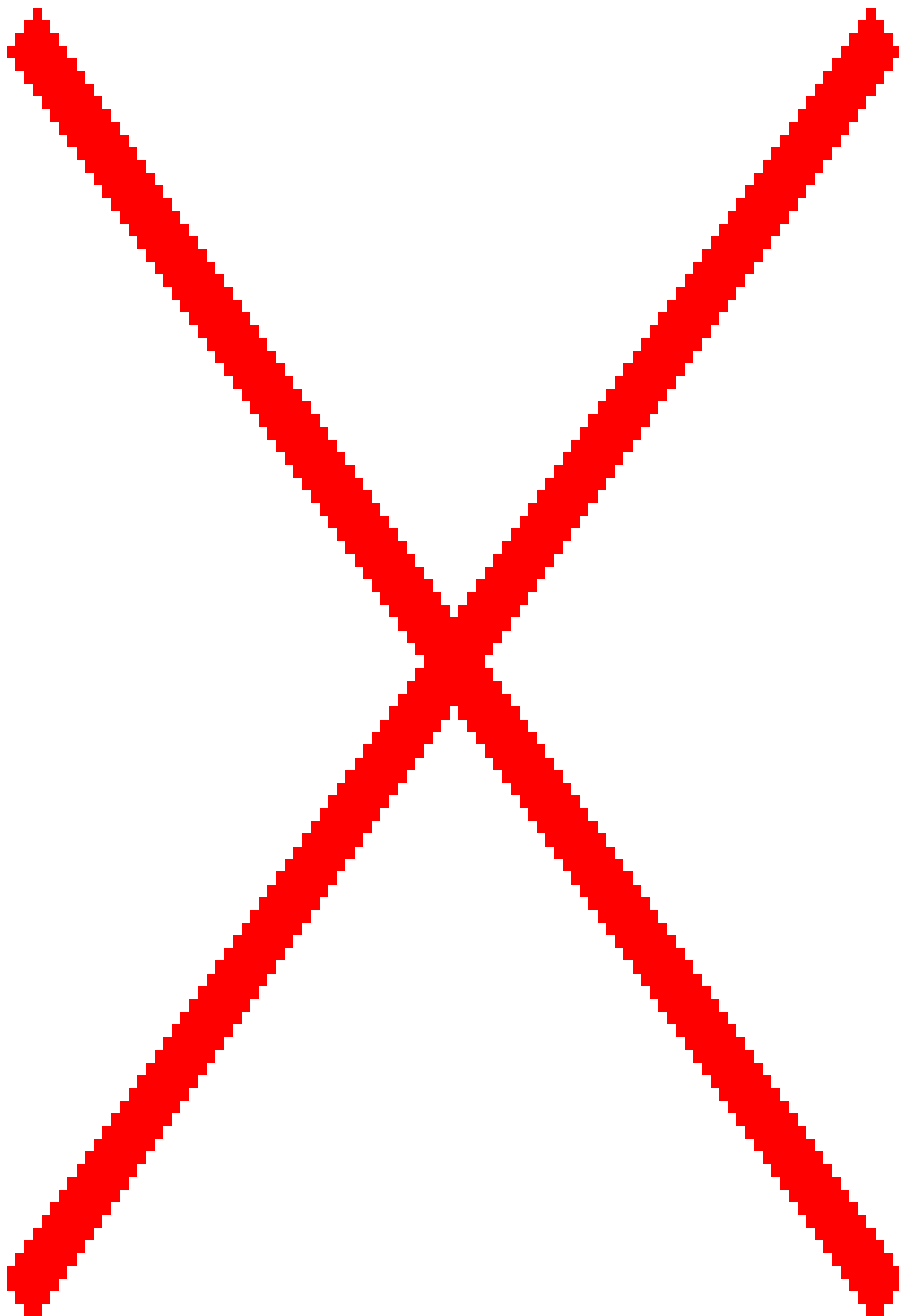
Una vez Obtenidos las simulaciones temporales del circuito hemos de comprobar que el sistema funciona correctamente teniendo en cuenta los retardos que no se contaban en la simulación temporal, si cumple con estos requisitos el modulo esta terminado a falta de añadirlo al grupo y de testearlo en la placa.

Retardo entre el flanco activo de la señal de reloj y la activación de los puertos DIR\_VLD y DATO\_VLD, este retardo ha de tenerse en cuenta ya que es una señal cuya duración es muy pequeña y si los retardos en el circuito son grandes es posible que no llegue a mostrarse nunca.

Retardo Dir_VLD	100ps
Retardo DATO_VLD	100ps



*Recursos Utilizados*



## Memoria TOP SYSTEM

En este apartado se crea el modulo top\_system en el cual esta incluido el modulo cnt\_epp, con este modulo se pretende simular completamente la funcionalidad de dicho componente y una vez verificado se volcara a la placa para su comprobación final.

Las entradas y salidas de este apartado son las citadas a continuación ademas se especifica el tipo de dato que son y si son de entrada o de salida:

```
CLK : in std_logic;
RST : in std_logic;
ASTRB : in std_logic;
DSTRB : in std_logic;
DATA : inout std_logic_vector(7 downto 0);
PWRITE : in std_logic;
PWAIT : out std_logic;
SWITCHES_I : in std_logic_vector(7 downto 0);
PSH_BUTTON: in std_logic;
LEDS_O : out std_logic_vector (7 downto 0));
```

Señales auxiliares utilizadas en el programa para poder realizar las interconexiones entre las distintas partes del los componentes:

```
signal DIR ,DIR_REG, DATO, DATO_REG, DATO_RD : std_logic_vector (7 downto 0);
signal DIR_VLD, DATO_VLD, CE_RD : std_logic;
```

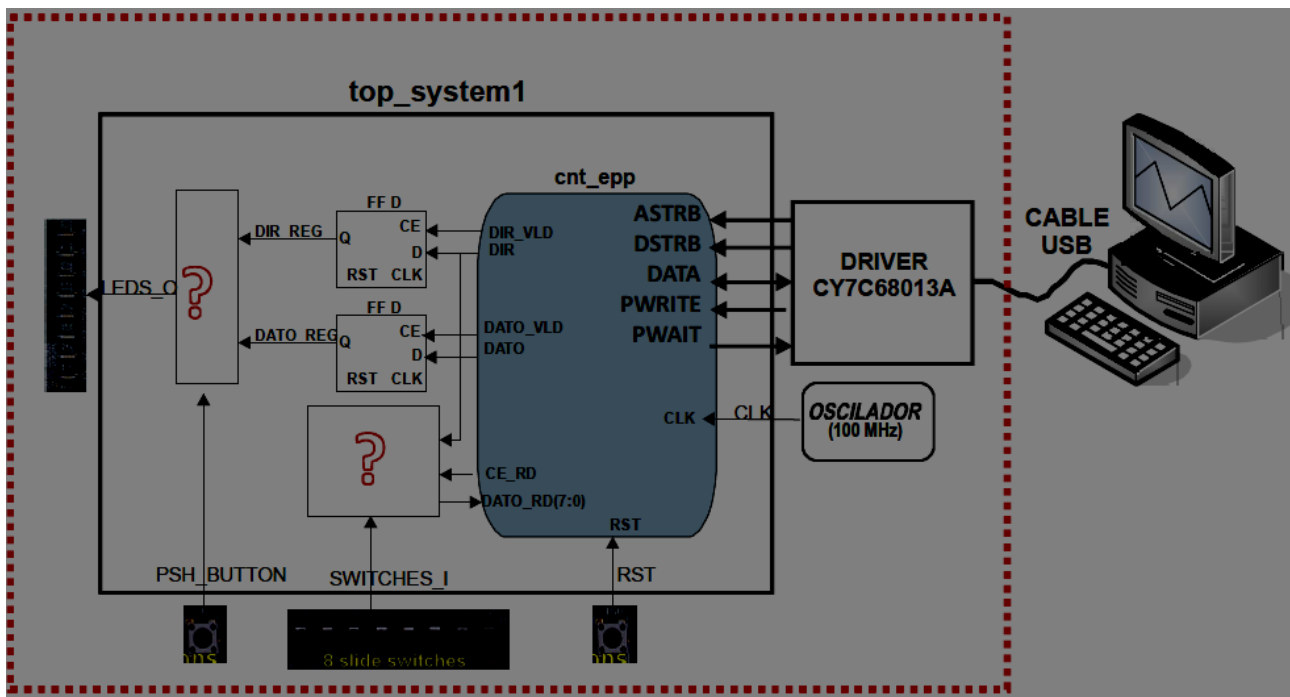
Se ha de especificar el componente que hemos utilizado dentro del sistema para poder usarlo de no ser así el sistema no funcionara.

```
DUT : entity work.cnt_epp
port map (
    CLK    => CLK,
    RST    => RST,
    ASTRB  => ASTRB,
    DSTRB  => DSTRB,
    DATA  => DATA,
    PWRITE => PWRITE,
    PWAIT  => PWAIT,
    DATO_RD => DATO_RD,
    CE_RD  => CE_RD,
    DIR    => DIR,
    DIR_VLD => DIR_VLD,
    DATO   => DATO,
    DATO_VLD => DATO_VLD);
```

Para su implementación dentro del archivo adjunto entregado por el profesor se ha de realizar el circuito descrito en la imagen adjunta a continuación.

Los dos circuitos con ? Corresponden a lo descrito en el siguiente apartado:

- El bloque controlado con la señal *PSH\_BUTTON* permite seleccionar que información se visualiza en los LEDS de forma que con un nivel bajo se selecciona la dirección registrada y con un alto el dato registrado.
- El bloque al que tiene conectado los 8 switches de la placa (puerto *SWITCHES\_I*) deberá llevarlos a la entrada *DATO\_RD* del componente *cnt\_epp* cuando se realice un ciclo de lectura en la posición 32HEX.

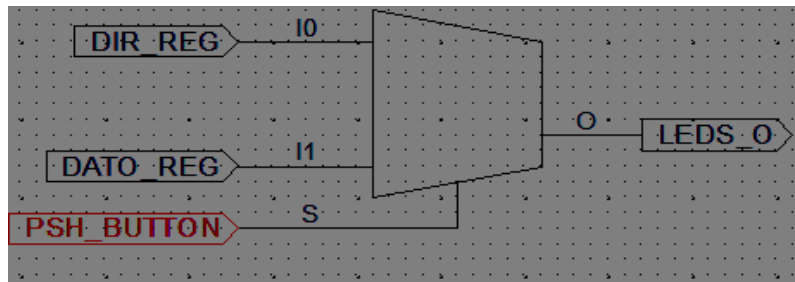


Tal y como se ha explicado arriba los dos bloques han de ser desarrollados para poder capturar la información a través de los swithces y poder mostrarla a través de los leds, pudiendo comprobar si realiza bien su finalidad para la cual se ha implementado dicho apartado. Su funcionamiento esta mas detallado en la descripción de los bloques que lo forman.

## Circuitos Digitales con explicación

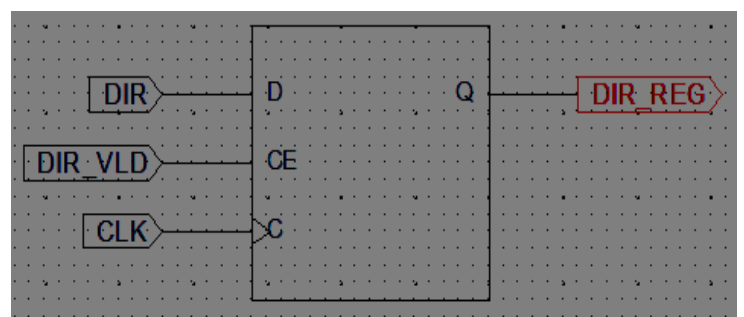
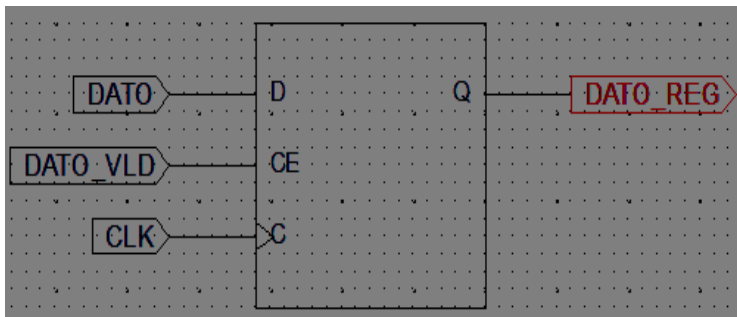
**Presente en:** top\_system

**Función:** Su funcionamiento es el de elegir una de las dos entradas y pasarla a los leds para ello se utiliza el botón psh\_button en cual selecciona a nivel bajo la dirección y a nivel alto el dato



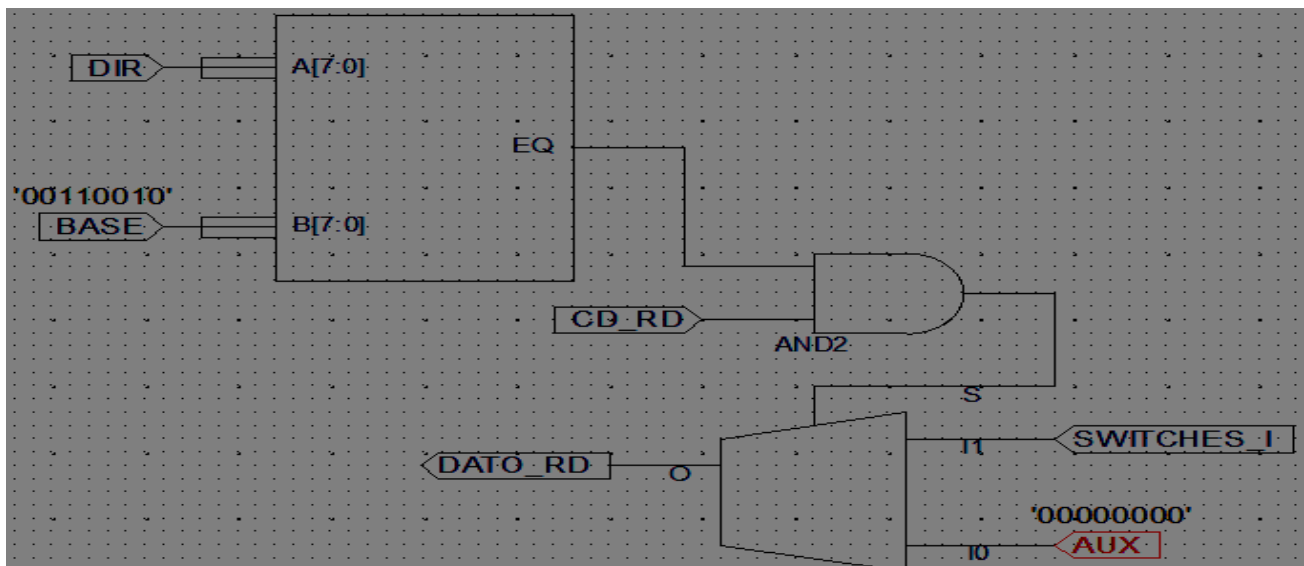
**Presente en:** top\_system

**Función:** Su función es la de almacenar en su salida el contenido de la entrada ya sea DATO o DIR, estos se consiguen cuando se realiza un pulso de reloj y CE esta activo para cada uno de ellos (DIR\_VLD o DATO\_VLD), al cumplirse esta condición se almacena en la salida el contenido de ese momento en la entrada permaneciendo ahí hasta la próxima vez que se cumpla la condición.



**Presente en:** top\_system

**Función:** La finalidad de este circuito es la de transferir el contenido de SWITCHES\_I a DATO\_RD cuando se cumpla la condición y durante cierto periodo de tiempo, para ello primero ha de cumplir que en DIR se encuentre el valor x"32" para que pueda continuar, de ser así si la señal CD\_RD esta activa entonces permite el paso del contenido de SWITCHES\_I para que sea copiado en DATO\_RD, si no se cumplen las condiciones anteriores se transmite la señal aux que contiene "00000000" como información.



## Top System Code

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

entity top_system1 is
port(
  CLK      : in  std_logic;
  RST      : in  std_logic;
  ASTRB    : in  std_logic;
  DSTRB    : in  std_logic;
  DATA    : inout std_logic_vector(7 downto 0);
  PWRITE   : in  std_logic;
  PWAIT    : out std_logic;
  SWITCHES_I : in  std_logic_vector(7 downto 0);
  PSH_BUTTON : in  std_logic;
  LEDS_O    : out std_logic_vector (7 downto 0));
end top_system1;

architecture rtl of top_system1 is
  signal DIR ,DIR_REG, DATO, DATO_REG, DATO_RD : std_logic_vector (7 downto 0);
  signal DIR_VLD, DATO_VLD, CE_RD : std_logic;
begin -- rtl

  DUT : entity work.cnt_epp
  port map (
    CLK    => CLK,
    RST    => RST,
    ASTRB   => ASTRB,
    DSTRB   => DSTRB,
    DATA   => DATA,
    PWRITE  => PWRITE,
    PWAIT   => PWAIT,
    DATO_RD => DATO_RD,
    CE_RD   => CE_RD,
    DIR     => DIR,
    DIR_VLD => DIR_VLD,
    DATO    => DATO,
    DATO_VLD => DATO_VLD);

  --DIR_REG biestable que almacena la información de la dir cada DIR_VLD
  process (CLK, RST)
  begin
    if RST='1' then
      DIR_REG <= (others => '0');
    elsif (CLK'event and CLK='1') then
      if DIR_VLD = '1' then
        DIR_REG <= DIR;
      end if;
    end if;
  end process;
```

```

--DATO_REG biestable que almacena la información de la dir cada DATO_REG
process (CLK, RST)
begin
  if RST='1' then
    DATO_REG <= (others => '0');
  elsif (CLK'event and CLK='1') then
    if DATO_VLD = '1' then
      DATO_REG <= DATO;
    end if;
  end if;
end process;

--Multiplexor que en función de el estado de PSH_BUTTON elige DATO_REG o DIR_REG
LEDS_O <= DATO_REG WHEN PSH_BUTTON ='1' ELSE DIR_REG;

-- Transfiere el contenido de SWITCHES_I a DATO_RD cuando es el ciclo de lectura x32
process (DIR, CE_RD, SWITCHES_I) is
begin
  if DIR=x"32" and CE_RD='1' then
    DATO_RD <= SWITCHES_I;
  else
    DATO_RD <= (others => '0');
  end if;
end process;

end rtl;

```

## TestBench

Una vez creado el código en base a los circuitos digitales que queremos implementar, pasamos a la creación del testbench de este apartado, para verificar su correcto funcionamiento para ello creamos el archivo top\_system\_tb en el cual implementamos toda la lógica necesaria para poder simular el comportamiento del componente, hemos de incluir el archivo epp\_device para poder probar correctamente su funcionamiento, el archivo cnt\_epp no es necesario declararlo ya que esta incluido en el propio top\_system.

**En el archivo epp\_device se ha forzado la dir x"32" para verificar una de sus funciones no se añade el código ya que es el mismo que en el apartado cnt\_epp**

El código TestBench con todos los datos seria el citado a continuación:

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

-----

entity top_system1_tb is

end entity top_system1_tb;

-----

architecture top_system1 of top_system1_tb is

    -- component ports
    signal CLK      : std_logic := '0';
    signal RST      : std_logic := '1';
    signal ASTRB    : std_logic := '1';
    signal DSTRB    : std_logic := '1';
    signal DATA    : std_logic_vector(7 downto 0);
    signal PWRITE   : std_logic := '1';
    signal PWAIT    : std_logic;
    signal SWITCHES_I : std_logic_vector(7 downto 0) := x"30";
    signal PSH_BUTTON : std_logic := '0';
    signal LEDS_O    : std_logic_vector (7 downto 0);

begin -- architecture top_system1

    -- component instanciación
    DUT: entity work.top_system1
    port map (
        CLK      => CLK,
        RST      => RST,
        ASTRB    => ASTRB,
        DSTRB    => DSTRB,
        DATA    => DATA,
        PWRITE   => PWRITE,
        PWAIT    => PWAIT,
        SWITCHES_I => SWITCHES_I,
        PSH_BUTTON => PSH_BUTTON,
```



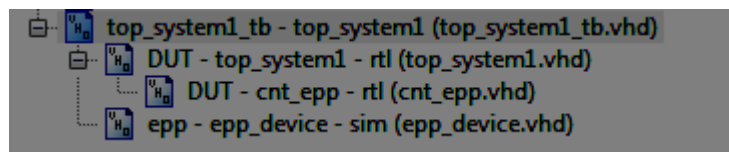
```
LEDS_O => LEDS_O);
```

```
epp: entity work.epp_device  
port map (  
    DATA => DATA,  
    PWRITE => PWRITE,  
    DSTRB => DSTRB,  
    ASTRB => ASTRB,  
    PWAIT => PWAIT);
```

```
PSH_BUTTON <= not PSH_BUTTON after 1000 ns;  
-- clock generation  
CLK <= not CLK after 5 ns;  
RST <= '1', '0' after 25 ns;
```

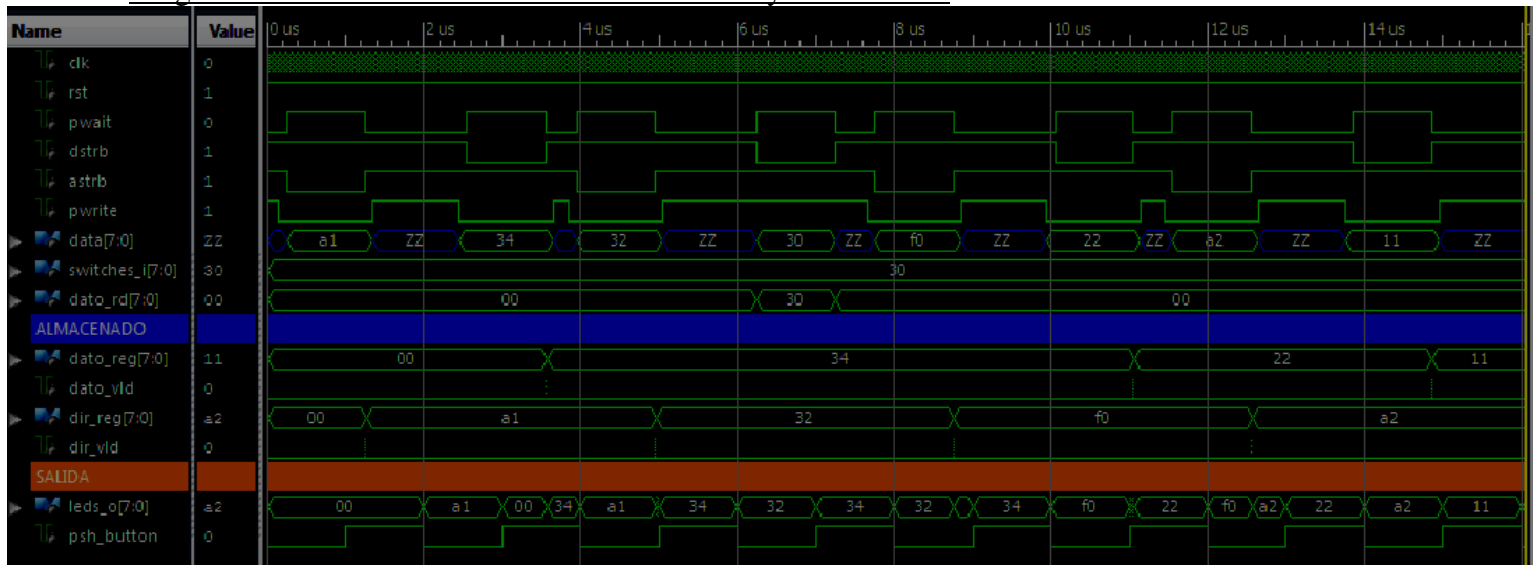
```
end architecture top_system1;
```

En el programa hemos de obtener algo parecido a lo mostrado en la imagen para que su funcionamiento sea el correcto.



## Simulación Funcional

Diagrama de simulación con 3 ciclos de escritura y 1 de lectura:



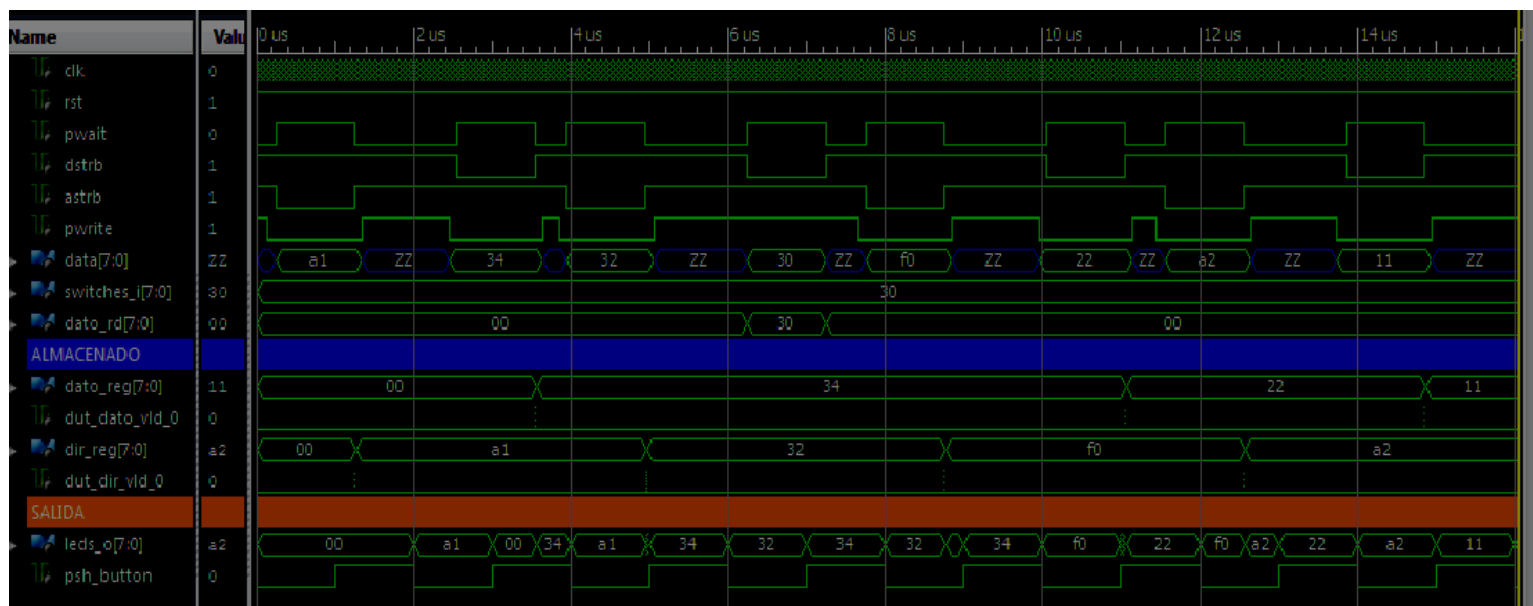
Una vez analizados los datos obtenidos de las gráficas y comparados con los de la practica si los resultados son satisfactorios podemos continuar con el siguiente paso.

## Simulación Temporal

Después de verificar que los ciclos son correctos pasamos a realizar la simulación temporal para ello seguimos los paso explicados en clase:

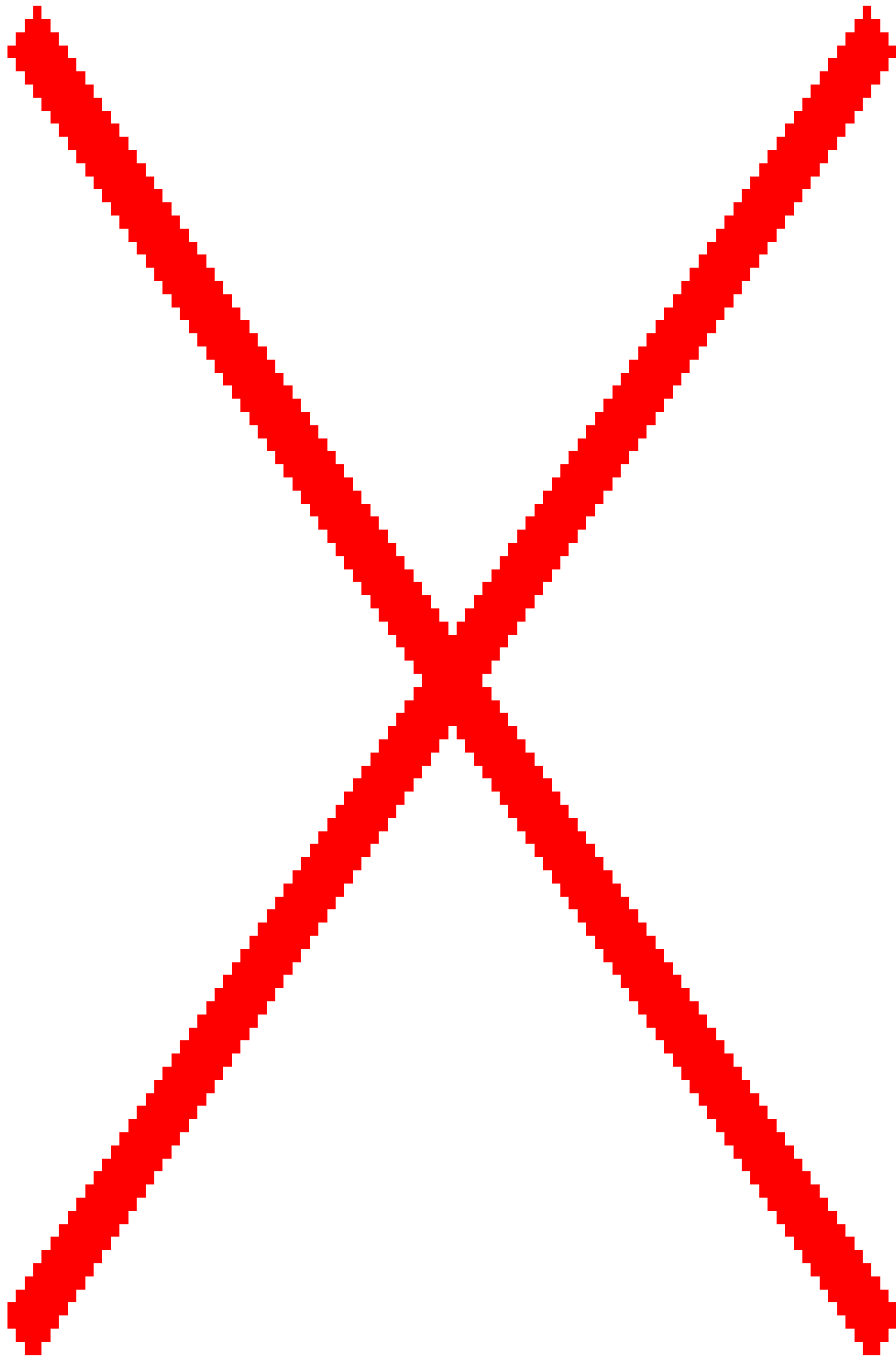
- Realizamos en el apartado de implementación el Post&Place and Route
- Cambiamos a la ventana de simulación
- Elegimos Post-Rute en el desplegable
- Chequeamos la síntesis del código
- Realizamos la simulación temporal en la cual se tienen en cuenta los retardos de las puertas lógicas y del resto de los componentes

Diagrama de simulación temporal con 3 ciclos de escritura y 1 de lectura:



Una vez analizados los datos obtenidos de las gráficas y comparados con los de la practica si los resultados son satisfactorios podemos continuar con el siguiente paso.

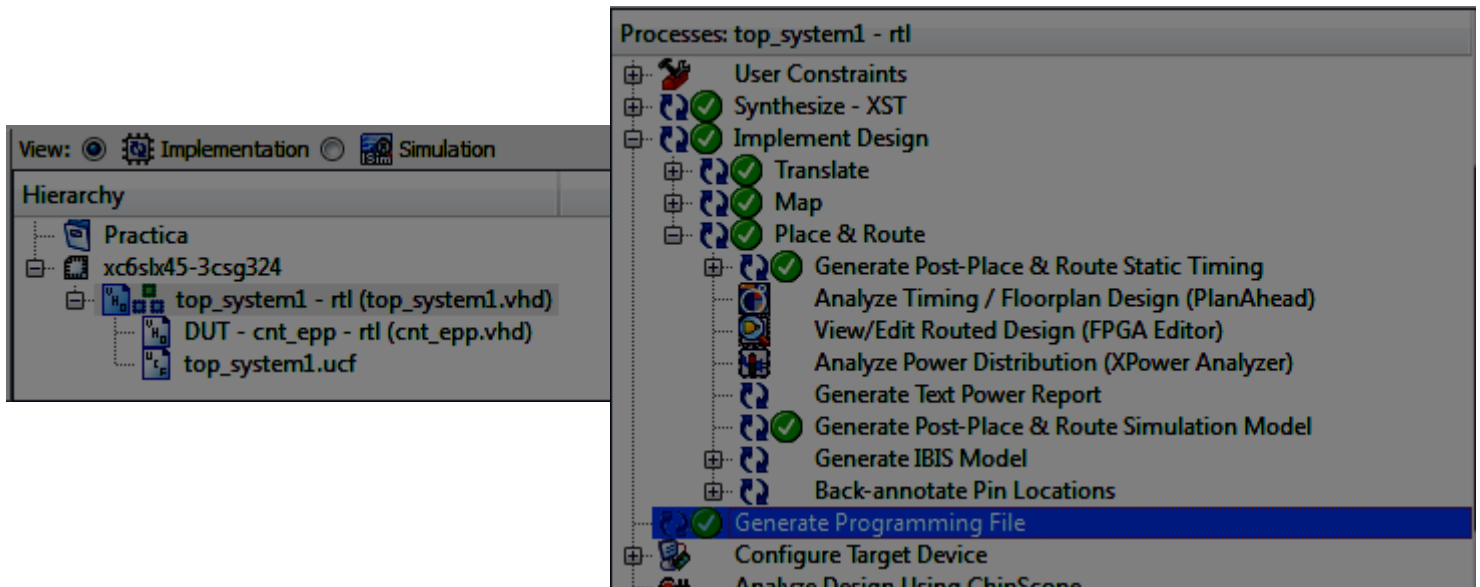
*Recursos Utilizados*



### Generación del archivo a descargar en placa

Una vez comprobado que la simulación temporal es correcta pasamos a añadir al programa el archivo top\_system1.uucf en el cual se especifican cuales son los puertos que se usan y a que salida o entrada corresponden en la placa. Quedando igual que en la imagen.

Después generamos el archivo.bit el cual sera el descargado en la placa para su posterior testeo final



### Top\_system.ucf Code

```
NET "CLK" LOC = "L15";
```

```
# onBoard USB controller
```

```
NET "ASTRB" LOC = "B9";  
NET "DSTRB" LOC = "A9";  
NET "PWRITE" LOC = "C15";  
NET "PWAIT" LOC = "F13";  
NET "DATA<0>" LOC = "A2";  
NET "DATA<1>" LOC = "D6";  
NET "DATA<2>" LOC = "C6";  
NET "DATA<3>" LOC = "B3";  
NET "DATA<4>" LOC = "A3";  
NET "DATA<5>" LOC = "B4";  
NET "DATA<6>" LOC = "A4";  
NET "DATA<7>" LOC = "C5";
```

```
# onBoard Pushbuttons
```

```
NET "PSH_BUTTON" LOC = "F5";  
NET "RST" LOC = "N4";
```

```
# onBoard LEDS
```

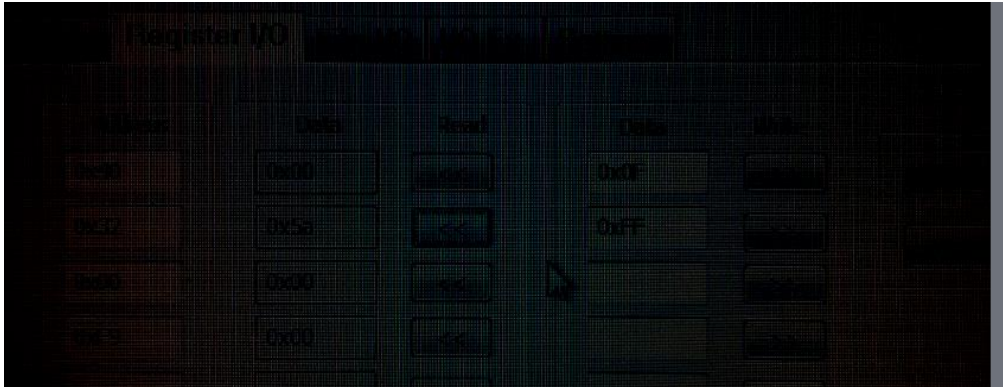
```
NET "LEDS_O<0>" LOC = "U18";  
NET "LEDS_O<1>" LOC = "M14";  
NET "LEDS_O<2>" LOC = "N14";  
NET "LEDS_O<3>" LOC = "L14";  
NET "LEDS_O<4>" LOC = "M13";  
NET "LEDS_O<5>" LOC = "D4";  
NET "LEDS_O<6>" LOC = "P16";  
NET "LEDS_O<7>" LOC = "N12";
```

```
# onBoard SWITCHES
```

```
NET "SWITCHES_I<0>" LOC = "A10";  
NET "SWITCHES_I<1>" LOC = "D14";  
NET "SWITCHES_I<2>" LOC = "C14";  
NET "SWITCHES_I<3>" LOC = "P15";  
NET "SWITCHES_I<4>" LOC = "P12";  
NET "SWITCHES_I<5>" LOC = "R5";  
NET "SWITCHES_I<6>" LOC = "T5";  
NET "SWITCHES_I<7>" LOC = "E4";
```

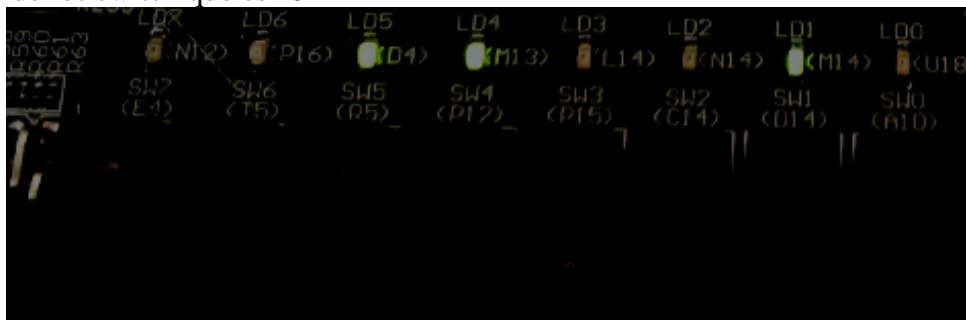
### *Descarga y test en placa Top system*

Para este apartado una vez generado el archivo .bit hemos de ir al ordenador del profesor en el cual se encuentra instalado el programa para volcar los datos a la placa para su posterior prueba.



En el programa cargamos en la parte de la izquierda las direcciones de donde queremos leer la información que en este caso la información se encuentra almacenada en los switch de la placa, pero solo puede ser leída si se elige la dirección x32 el resto no leen nada

En el caso de la imagen superior cuando leemos la dirección x32 mostrada en los leds nos muestra el contenido de los switch que es x5A



Y en la parte de la derecha podemos insertar una dirección de escritura la cual se mostrara en los leds cuando se pulse el botón que se a definido para alternar la información entre la dirección de lectura y la de escritura en este caso la dirección introducida es xF0.

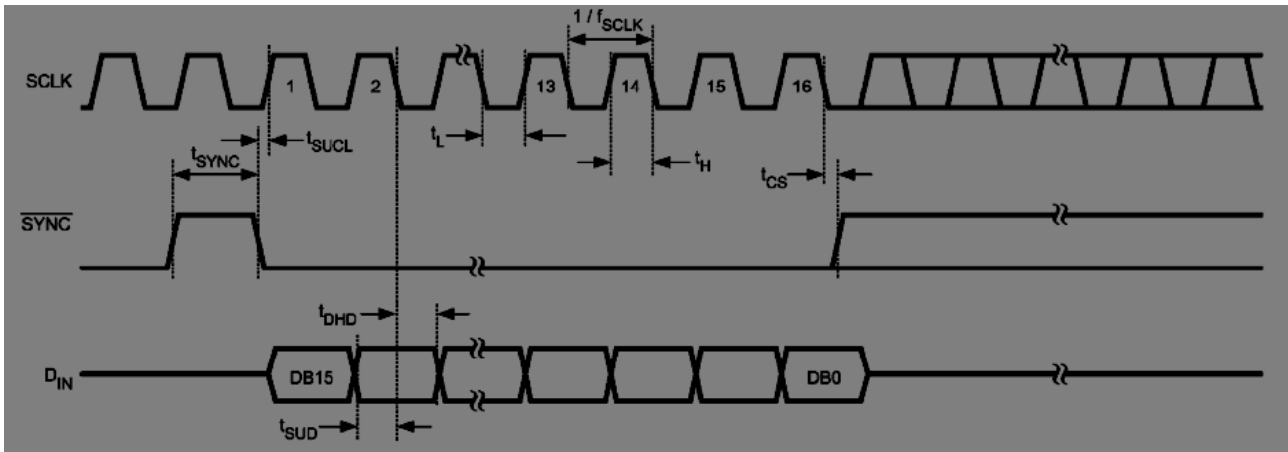


## Memoria CNT DAC

La entidad, es la encargada de proporcionar las señales de control de los dos DAC121S101 de la placa PmodDA1, los cuales van a proporcionar las tensiones de salida Vo1 y Vo2 a partir de los datos almacenados en las memorias dual port.

El DAC121S101 es un convertidor digital analógico de 12 bits con una interface serie que acepta diferentes protocolos . Para ello utiliza tres líneas de control: ***Din*** para introducir los bits del dato a convertir y ***SCLK*** y ***SYNC*** que sincronizan dicha transferencia.

Para una correcta operación, la transferencia de datos debe cumplir el cronograma de la imagen adjunta, nótese como en la entrada ***DIN*** se proporciona un nuevo valor coincidiendo con el flanco de subida de ***SCLK***.



Las entradas y salidas de este apartado son las citadas a continuación además se especifica el tipo de dato que son y si son de entrada o de salida:

```
CLK : in std_logic;
RST : in std_logic;
DATO1 : in std_logic_vector(7 downto 0);
DATO2 : in std_logic_vector(7 downto 0);
DATO_OK : in std_logic;
SYNC : out std_logic;
SCLK : out std_logic;
D1 : out std_logic;
D2 : out std_logic;
```

Señales auxiliares utilizadas en el programa para poder realizar las interconexiones entre las distintas partes de los componentes:

```
signal D1BD : std_logic_vector(7 downto 0); -- señal aux salida biestable d de dato 1
signal D2BD : std_logic_vector(7 downto 0); -- señal aux salida biestable d de dato 2
signal SCDATA : std_logic_vector(3 downto 0); --salida contador para multiplexor
signal CEC : std_logic; -- señal de CE "activación" para contador
signal Q0 : std_logic; -- Salida contador binario de 2 bits
signal Q1 : std_logic; -- Salida contador binario de 2 bits
signal FinTX : std_logic; --Final de la cuenta del contador "0000"
signal Stado_Rep : std_logic; --señal Para indicar estado de reposo y resetear el contador
signal RE_CB : std_logic; --Señal para resetear el contador binario
```

## Declaración de la maquina de estados

```
type MEF is (REP, TX, R1, R2);
signal std_act, prox_std : MEF;
```

En este apartado tal y como se ha dicho al principio se trata de enviar a las salidas la información almacenada dentro de la memorias pero para ello han de enviarse en formato serie y han de cumplir una restricciones de tiempo para que la recepción sea la correcta.

Se va a establecer para *SCLK* una frecuencia constante, fijando sus tiempos de nivel bajo ( $t_L$ ) y alto ( $t_H$ ) a 20 ns, valor mínimo que se puede conseguir, con una señal *CLK* de frecuencia 100 Mhz mediante un contador binario la que nos dividiría la frecuencia que para 100 Mhz nos deja 5ns en 2 veces quedando un tiempo de 20ns lo necesario para la practica. Con esto logramos que la transferencia de la información cumpla la restricciones y que se transfiera de forma constante obteniendo asi un numero de 66 ciclos necesarios para transmitir la información.

También se han de respetar el orden de los datos ya que cada uno de ellos tiene un cometido dentro de lo que es el ejercicio a realizar.

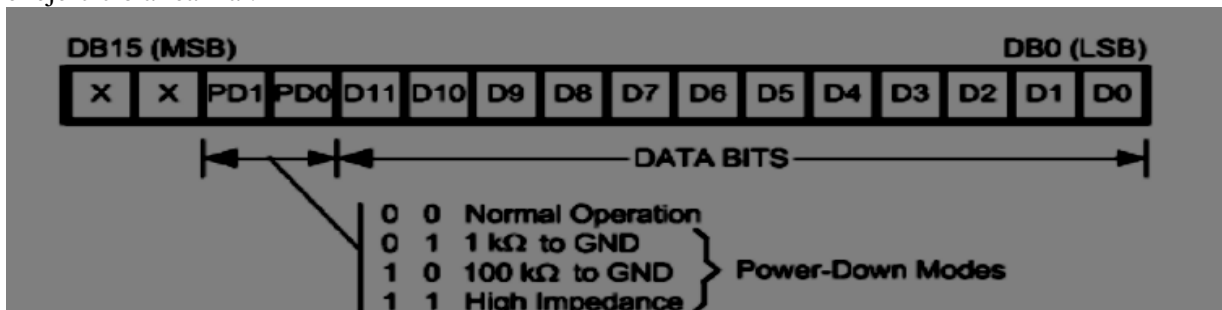


Tabla con información importante sobre los tiempos que se han de respetar para su correcto funcionamiento

Symbol	Parameter	Conductions		Typical	Limits	Units (Limits)
f <sub>SCLK</sub>	SCLK Frequency				30	MHz (max)
t <sub>s</sub>	Output Voltage Settling Time (Note 10)	400h to C00h code change, R <sub>L</sub> = 2kΩ	C <sub>L</sub> ≤ 200 pF	8	10	μs (max)
			C <sub>L</sub> = 500 pF	12		μs
		00Fh to FF0h code change, R <sub>L</sub> = 2kΩ	C <sub>L</sub> ≤ 200 pF	8		μs
			C <sub>L</sub> = 500 pF	12		μs
SR	Output Slew Rate			1		V/μs
	Glitch Impulse	Code change from 800h to 7FFh		12		nV-sec
	Digital Feedthrough			0.5		nV-sec
t <sub>WU</sub>	Wake-Up Time	V <sub>A</sub> = 5V		6		μs
		V <sub>A</sub> = 3V		39		μs
1/f <sub>SCLK</sub>	SCLK Cycle Time				33	ns (min)
t <sub>H</sub>	SCLK High time			5	13	ns (min)
t <sub>L</sub>	SCLK Low Time			5	13	ns (min)
t <sub>SUCL</sub>	Set-up Time SYNC to SCLK Rising Edge			-15	0	ns (min)
t <sub>SUD</sub>	Data Set-Up Time			2.5	5	ns (min)
t <sub>DHD</sub>	Data Hold Time			2.5	4.5	ns (min)
t <sub>CS</sub>	SCLK fall to rise of SYNC	V <sub>A</sub> = 5V		0	3	ns (min)
		V <sub>A</sub> = 3V		-2	1	ns (min)
t <sub>SYNC</sub>	SYNC High Time	2.7 ≤ V <sub>A</sub> ≤ 3.6		9	20	ns (min)
		3.6 ≤ V <sub>A</sub> ≤ 5.5		5	10	ns (min)

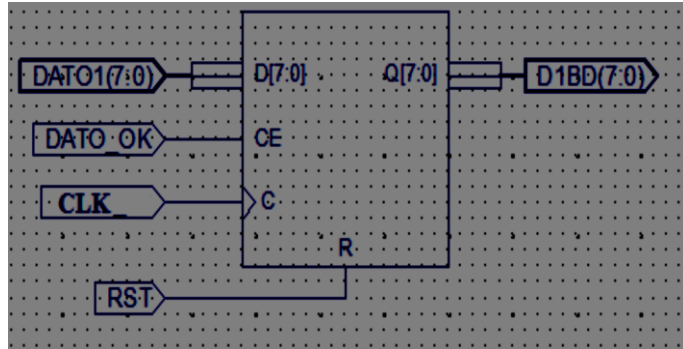
Los esquemas digitales presentados a continuación son los ideales para poder cumplir con las restricciones que presenta el ejercicio tanto temporales como de información, para ello con la información que disponemos se a optado por esta solución al problema presentado pudiendo haber mas soluciones posibles.



## CIRCUITOS DIGITALES

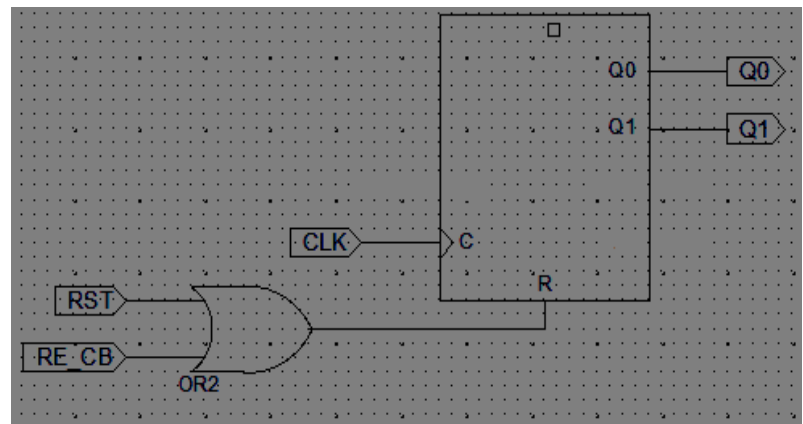
**Nombre:** biestable para almacenar dato 1 y dato 2

**Descripción:** Este circuito se encarga de pasar y almacenar el contenido que hay en DATO1/2 para su posterior utilización, su funcionamiento es el básico de el biestable d, y el dato solo se puede almacenar cuando la señal DATO\_OK esta activa a nivel alto lo que indica que hay un dato disponible. Este esquema se encuentra en las dos DATO1 y DATO2



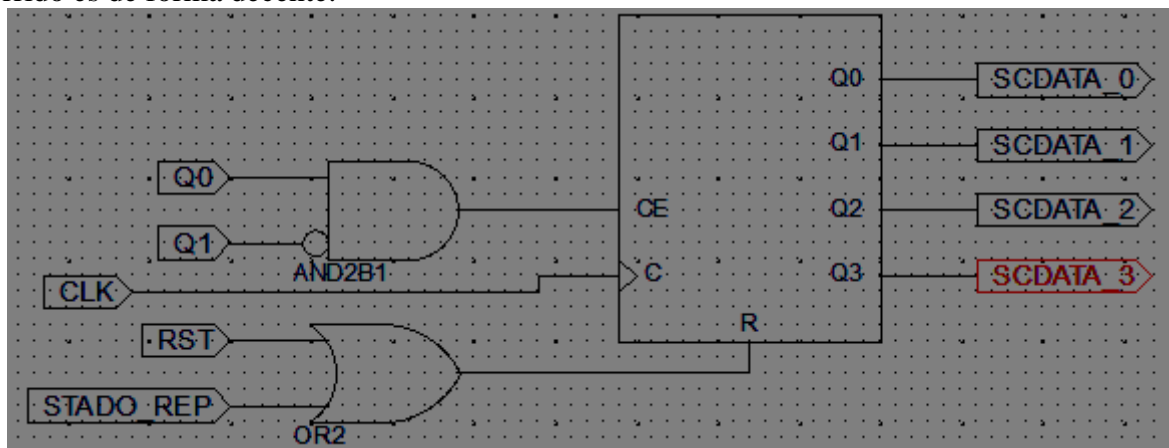
**Nombre:** Contador Binario

**Descripción:** Con este contador binario lo que buscamos es poder dividir la señal de salida del reloj de la placa pudiendo alcanzar las restricciones que se imponen en el circuito, dicho contador solo funcionara cuando la señal RST o RE\_CB no estén reseteando su cuenta situación que se cumple cuando hemos de empezar a transmitir los datos, las salidas sirven para habilitar la cuenta del contador de el multiplexor de data1/2 y ademas Q1 también actúa como SCLK.



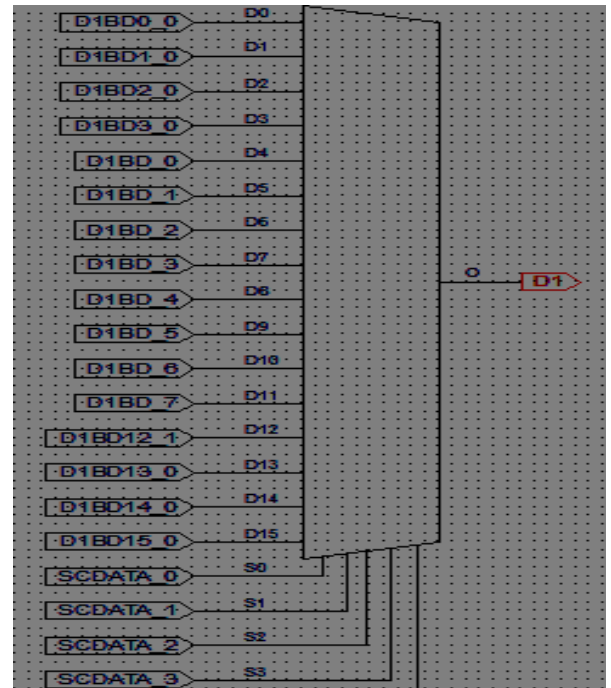
**Nombre:** Contador multiplexores Data1 y Data 2

**Descripción:** Este circuito es el contador de 4 bits el cual se encarga de mandar un dato u otro del multiplexor, la señal de reset esta gobernada por el RST de la placa y también por el estado de reposo de la maquina de estado impidiendo así contar antes de que se incumplan las condiciones de reset, para poder aumentar la cuenta se ha de cumplir que las salidas del contador binario se encuentren a 1 y 0 respectivamente, esto para que solo se permita contar en determinadas situaciones especificadas en la practica, en cuanto a las salidas serán las encargadas de elegir las salidas del multiplexor de Dato1/2 ya que este esquema es valido para los dos multiplexores. Su recorrido es de forma decente.



**Nombre:**Multiplexor Data1 y Data2

**Descripción:**El circuito descrito a continuación se basa en transformar la información que le llega en formato paralelo a formato serie gracias a los otros componentes pasa la información de forma ordenada y cumpliendo las restricciones de tiempos, A la salida de el biestable D se le han de concatenar las cadenas "0000" al principio y "0000" ya que para lo que se pide en la practica es necesario que contenga dicha información, mediante la salida del contador se han de seleccionar los bits que se han de transmitir a la salida D1/D2



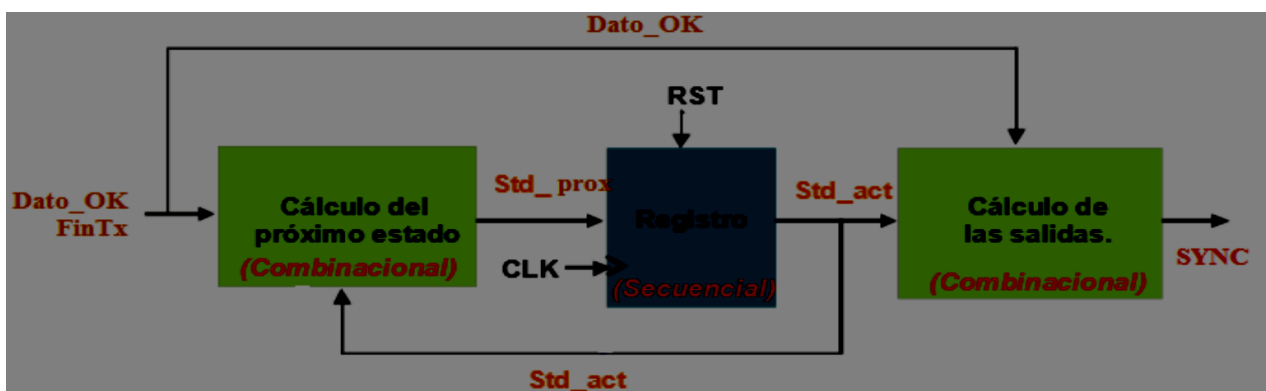
**Nombre:**Maquina de estados Mealy

**Descripción:**En la maquina de estados que utilizamos para el circuito cumple las características de las maquinas explicadas en clase.

Apartado 1: Calcula el próximo estado a realizar empezando en el estado de reposo hasta que dato\_ok pase a nivel alto, después de llegar a TX no cambia hasta que no termine la cuenta para después cambiar directamente mediante dos pasos intermedios (necesarios para cumplir las restricciones de tiempo) hasta llegar a reposo otra vez.

Apartado 2: Solo se encarga de moverse al apartado que le toca en cada pulso de reloj.

Apartado 3: En función del apartado en el que se encuentre sera necesario que se mande una señal u otra en este caso cuando nos encontremos en reposo o en tx pero dato\_ok no este a cero aun entonces SYNC es 1 para el resto de casos es 0 por ello es una maquina de mealy.



### Código Cnt\_dac

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

```
entity cnt_dac is
port (
  CLK   : in std_logic;
```



```
  RST   : in std_logic;
  DATO1 : in std_logic_vector(7 downto 0);
  DATO2 : in std_logic_vector(7 downto 0);
  DATO_OK : in std_logic;
  SYNC   : out std_logic;
  SCLK    : out std_logic;
  D1      : out std_logic;
  D2      : out std_logic);
end cnt_dac;
```

architecture RTL of cnt\_dac is

```
  signal D1BD    : std_logic_vector(7 downto 0); -- señal aux salida biestable d de dato 1
  signal D2BD    : std_logic_vector(7 downto 0); -- señal aux salida biestable d de dato 2
  signal SCDATA  : std_logic_vector(3 downto 0); --salida contador para multiplexor
  signal CEC     : std_logic; -- señal de CE "activación" para contador
  signal Q0      : std_logic;    -- Salida contador binario de 2 bits
  signal Q1      : std_logic;    -- Salida contador binario de 2 bits
  signal FinTX   : std_logic;
  signal Stado_Rep : std_logic;
  signal RE_CB   : std_logic;
```

```
--MAQUINA ESTADO
type MEF is (REP, TX, R1, R2);
signal std_act, prox_std : MEF;
```

```
begin -- RTL
```

```
--Biestable data1
process (CLK, RST) is
begin
  if RST = '1' then
    D1BD <= (others => '0');
```

```

elsif CLK'event and CLK = '1' then
  if DATO_OK = '1' then
    D1BD(0) <= DATO1(0);
    D1BD(1) <= DATO1(1);
    D1BD(2) <= DATO1(2);
    D1BD(3) <= DATO1(3);
    D1BD(4) <= DATO1(4);
    D1BD(5) <= DATO1(5);
    D1BD(6) <= DATO1(6);
    D1BD(7) <= DATO1(7);
  end if;
end if;
end process;

```

--Biestable data2

process (CLK, RST) is

begin

if RST = '1' then

  D2BD <= (others => '0');

elsif CLK'event and CLK = '1' then

  if DATO\_OK = '1' then

    D2BD(0) <= DATO2(0);

    D2BD(1) <= DATO2(1);

    D2BD(2) <= DATO2(2);

    D2BD(3) <= DATO2(3);

    D2BD(4) <= DATO2(4);

    D2BD(5) <= DATO2(5);

    D2BD(6) <= DATO2(6);

    D2BD(7) <= DATO2(7);

  end if;

end if;

end process;

--multiplexor de DATA1

process (D1BD, SCDATA) is

begin

  case SCDATA is

    when "0000" => D1 <= '0';

    when "0001" => D1 <= '0';

    when "0010" => D1 <= '0';

    when "0011" => D1 <= '0';

    when "0100" => D1 <= D1BD(0);

    when "0101" => D1 <= D1BD(1);

    when "0110" => D1 <= D1BD(2);

    when "0111" => D1 <= D1BD(3);

    when "1000" => D1 <= D1BD(4);

    when "1001" => D1 <= D1BD(5);

    when "1010" => D1 <= D1BD(6);

    when "1011" => D1 <= D1BD(7);

    when "1100" => D1 <= '0';

    when "1101" => D1 <= '0';

    when "1110" => D1 <= '0';

    when others => D1 <= '0';

```

end case;
end process;

--multiplexor de DATA2
process (D2BD, SCDATA) is
begin
  case SCDATA is
    when "0000" => D2 <= '0';
    when "0001" => D2 <= '0';
    when "0010" => D2 <= '0';
    when "0011" => D2 <= '0';
    when "0100" => D2 <= D2BD(0);
    when "0101" => D2 <= D2BD(1);
    when "0110" => D2 <= D2BD(2);
    when "0111" => D2 <= D2BD(3);
    when "1000" => D2 <= D2BD(4);
    when "1001" => D2 <= D2BD(5);
    when "1010" => D2 <= D2BD(6);
    when "1011" => D2 <= D2BD(7);
    when "1100" => D2 <= '0';
    when "1101" => D2 <= '0';
    when "1110" => D2 <= '0';
    when others => D2 <= '0';
  end case;
end process;

```

```

--salida que controla el CE del contador de los multiplexores
CEC <= '1' when Q0 = '1' and Q1 = '0' else '0';

```

```

SCLK <= Q1;

```

```

--contador para transferir los datos del multiplexor 1 y 2

```

```

process (CLK, RST, Stado_Rep) is
begin
  if RST = '1' then
    SCDATA <= (others => '0');    --pone a 0 pero seria 1 preguntar
  elsif Stado_Rep = '0' then
    SCDATA <= (others => '0');    --pone a 0 pero seria 1 preguntar
  elsif CLK'event and CLK = '1' then
    if CEC = '1' then
      if SCDATA = x"0" then
        SCDATA <= (others => '1');
      else
        SCDATA <= std_logic_vector(unsigned(SCDATA)-1);
      end if;
    end if;
  end if;
end process;

```

```

--Contador binario para dividir la frecuencia del reloj a la mitad de la mitad
process (CLK, RST, RE_CB) is

```

```

    variable cnt : std_logic_vector(1 downto 0);
begin
    if RST = '1' then
        cnt := (others => '0');
        Q0 <= '0';
        Q1 <= '0';
    elsif RE_CB = '0' then
        cnt := (others => '0');
        Q0 <= '0';
        Q1 <= '0';
    elsif CLK'event and CLK = '1' then
        if cnt = "11" then
            cnt := (others => '0');
            Q0 <= cnt(0);
            Q1 <= cnt(1);
        else
            cnt := std_logic_vector(unsigned(cnt)+1);
            Q0 <= cnt(0);
            Q1 <= cnt(1);
        end if;
    end if;
end process;

```

--Parte 1 de la maquina de estados finitos

process (DATO\_OK, FinTX, std\_act) is

```

begin
    case std_act is
        when REP =>
            if DATO_OK = '1' then
                prox_std <= TX;
            else
                prox_std <= REP;
            end if;
        when Tx =>
            if FinTX = '1' then
                prox_std <= R1;
            else
                prox_std <= TX;
            end if;
        when R1 => prox_std <= R2;
        when R2 => prox_std <= REP;
    end case;
end process;

```

--Parte 2 de la maquina de estados finitos

process (CLK, RST) is

```

begin -- process
    if RST = '1' then          -- asynchronous reset (active low)
        std_act <= REP;
    elsif CLK'event and CLK = '1' then -- rising clock edge
        std_act <= prox_std;
    end if;
end process;

```

```

--Parte 3 de la maquina de estados finitos
process (std_act, DATO_OK) is
begin -- process
  case std_act is
    when REP => SYNC <= '1';
    when TX =>
      if DATO_OK = '0' then
        SYNC <= '0';
      else
        SYNC <= '1';
      end if;
    when R1 => SYNC <= '0';
    when R2 => SYNC <= '0';
  end case;
end process;

--FinTX comprueba si a llegado al final el contador
FinTX <= '1' when SCDATA = "0000" and std_act = TX and Q1 = '1' else '0';

--Señal extra para el reset del contador
Stado_Rep <= '0' when std_act = REP else '1';

--Reset contador binario
RE_CB <= '0' when std_act = REP or DATO_OK = '1' else '1';

end RTL;

```

### **TestBench**

Después de ello realizamos el TestBench para verificar su correcto funcionamiento para ello creamos el archivo cnt\_dac\_tb en el cual implementamos toda la lógica necesaria para poder simular el comportamiento del componente.

El código TestBench con todos los datos sería el citado a continuación:

```
library ieee;
use ieee.std_logic_1164.all;
-----
entity cnt_dac_tb is

end entity cnt_dac_tb;
-----

architecture for_dac_tb of cnt_dac_tb is

    -- component ports
    signal CLK    : std_logic := '1';
    signal RST    : std_logic;
    signal DATO1  : std_logic_vector(7 downto 0);
    signal DATO2  : std_logic_vector(7 downto 0);
    signal DATO_OK : std_logic;
    signal SYNC   : std_logic;
    signal SCLK   : std_logic;
    signal D1     : std_logic;
    signal D2     : std_logic;

begin -- architecture for_dac_tb

    -- component instantiation
    DUT : entity work.cnt_dac
    port map (
        CLK    => CLK,
        RST    => RST,
        DATO1  => DATO1,
        DATO2  => DATO2,
        DATO_OK => DATO_OK,
        SYNC   => SYNC,
        SCLK   => SCLK,
```



```
D1    => D1,
D2    => D2);
```

```
-- clock generation
```

```
Clk <= not Clk after 5 ns;
```

```
rst <= '1', '0' after 50 ns;
```

```
DATO1 <= x"00", x"A5" after 100 ns;
```

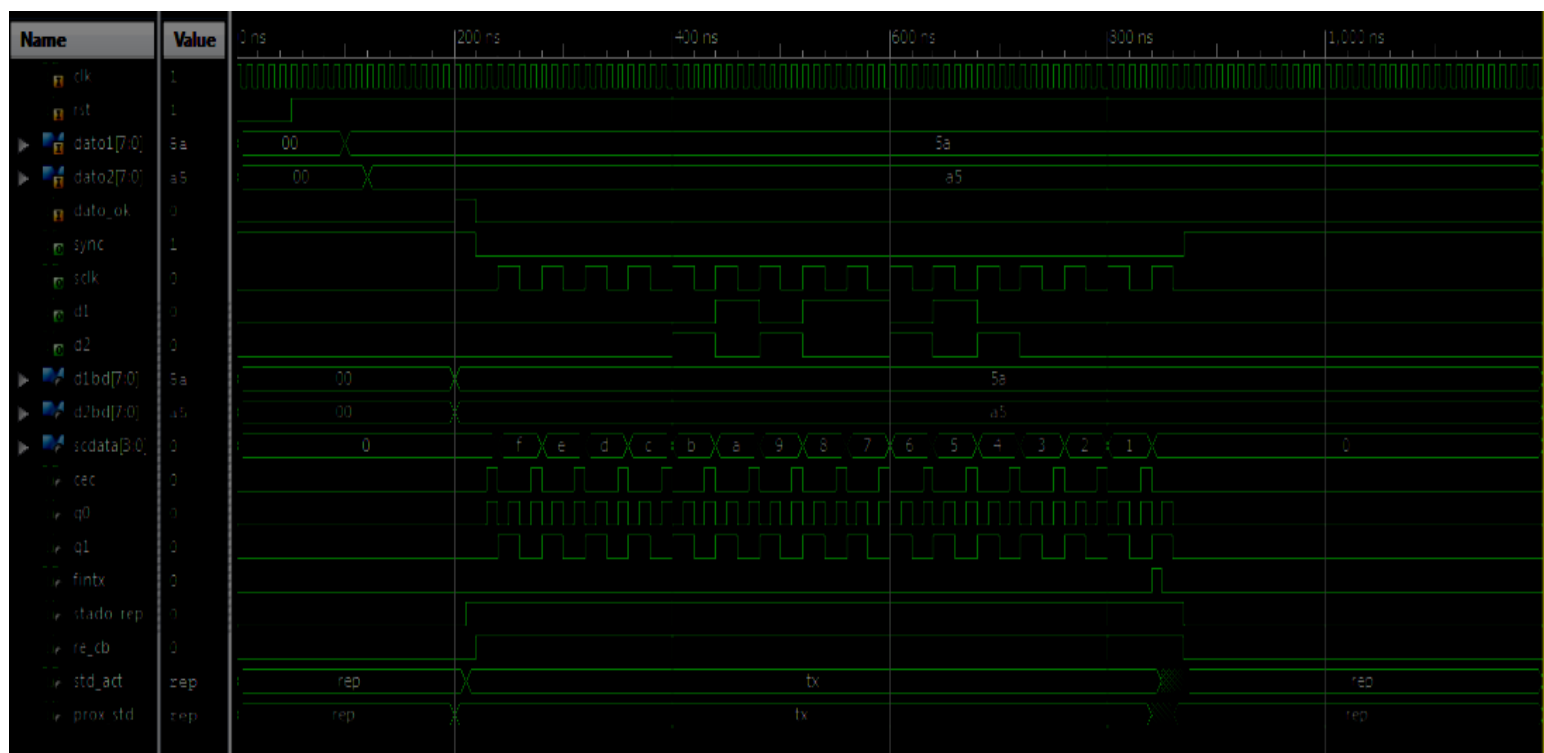
```
DATO2 <= x"00", x"5A" after 120 ns;
```

```
DATO_OK <= '0', '1' after 200 ns, '0' after 220 ns;
```

```
end architecture for_dac_tb;
```

### Simulación Funcional

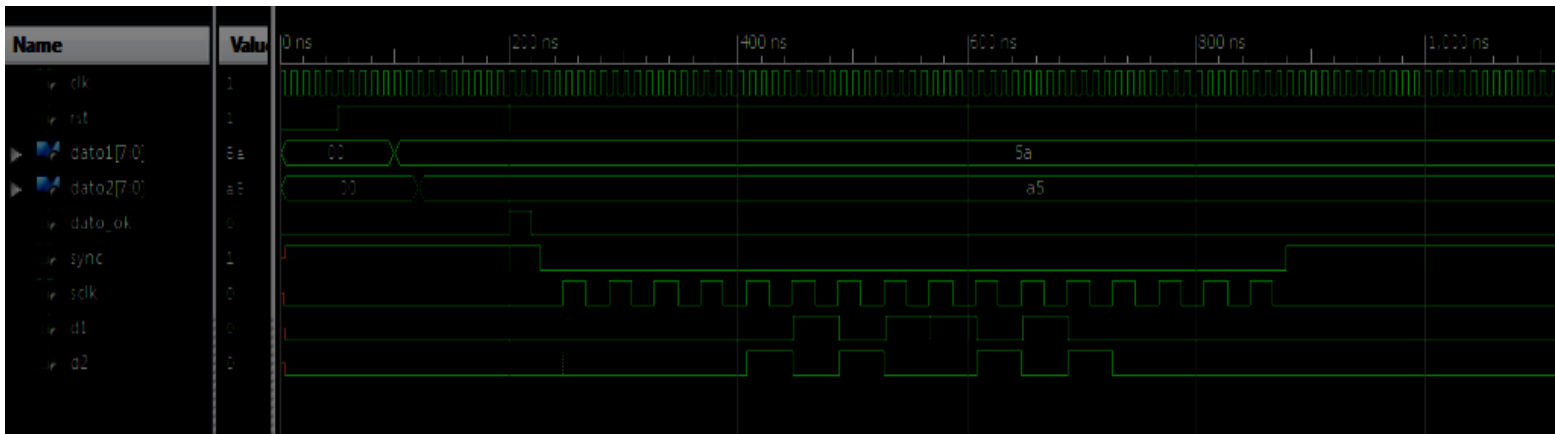
Diagrama funcional del circuito implantado y testeado con el código de pruebas.



Una vez analizados los datos obtenidos de las gráficas y comparados con los de la practica si los resultados son satisfactorios podemos continuar con el siguiente paso.

### Simulación Funcional

Diagrama temporal del circuito implantado y testeado con el código de pruebas.



Una vez Obtenidos las simulaciones temporales del circuito hemos de comprobar que el sistema funciona correctamente teniendo en cuenta los retardos que no se contaban en la simulación temporal, si cumple con estos requisitos el modulo esta terminado.

*Recursos Utilizados*

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	8	54,576	1,00%
Number used as Flip Flops	8		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	15	27,288	1,00%
Number used as logic	15	27,288	1,00%
Number using O6 output only	12		
Number using O5 output only	0		
Number using O5 and O6	3		
Number used as ROM	0		
Number used as Memory	0	6,408	0,00%
Number of occupied Slices	7	6,822	1,00%
Number of MUXCYs used	0	13,644	0,00%
Number of LUT Flip Flop pairs used	15		
Number with an unused Flip Flop	9	15	60,00%
Number with an unused LUT	0	15	0,00%
Number of fully used LUT-FF pairs	6	15	40,00%
Number of unique control sets	3		
Number of slice register sites lost to control set restrictions	16	54,576	1,00%
Number of bonded IOBs	23	218	10,00%
IOB Flip Flops	16		
Number of RAMB16BWERs	0	116	0,00%
Number of RAMB8BWERs	0	232	0,00%
Number of BUFIO2/BUFIO2_2CLKs	0	32	0,00%
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0,00%
Number of BUFG/BUFGMUXs	1	16	6,00%
Number used as BUFGs	1		
Number used as BUFGMUX	0		
Number of DCM/DCM_CLKGENs	0	8	0,00%
Number of ILOGIC2/ISERDES2s	16	376	4,00%
Number used as ILOGIC2s	16		
Number used as ISERDES2s	0		
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	376	0,00%
Number of OLOGIC2/OSERDES2s	0	376	0,00%
Number of BSCANs	0	4	0,00%
Number of BUFHs	0	256	0,00%
Number of BUFPLLs	0	8	0,00%
Number of BUFPLL_MCBs	0	4	0,00%
Number of DSP48A1s	0	58	0,00%
Number of ICAPs	0	1	0,00%
Number of MCBs	0	2	0,00%
Number of PCILOGICSEs	0	2	0,00%
Number of PLL_ADVs	0	4	0,00%
Number of PMVs	0	1	0,00%
Number of STARTUPs	0	1	0,00%
Number of SUSPEND_SYNCs	0	1	0,00%
Average Fanout of Non-Clock Nets	2,06		

### **Verificación de Funcionamiento mediante el Componente DAC121S101.vvvhdl**

Una vez completados los pasos anteriores se ha de verificar que el funcionamiento es el correcto y para ello se ha de probar el funcionamiento de cnt\_dac con DAC121S101.vhdl teniendo en cuenta

que este complemento ha de estar por duplicado, una vez incorporado en el programa se han de repetir los pasos para verificar que tanto la funcional como la temporal funcionan correctamente.

Los Dac se basan en la recepción de una señal SYNC SCLK Y D1/2 las cuales si cumplen los requisitos de dicho componente respetando los tiempos y como han de ser dichas señales, entonces el Dac se encargará de pasar el dato que le llega en formato serie en un dato compendiado entre 0 y 3,6.

Para ello hemos de incorporar en el archivo testbench los componentes de la forma descrita a continuación

T1 : entity work.DAC121S101

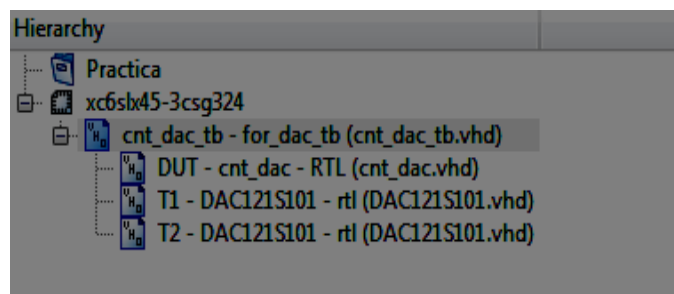
```
port map (  
  SYNC => SYNC,  
  SCLK => SCLK,  
  DIN => D2);
```

T2 : entity work.DAC121S101

```
port map (  
  SYNC => SYNC,  
  SCLK => SCLK,  
  DIN => D1);
```

La señal de salida no se incluye ya que no queremos sacarla ahora mismo si no solo comprobar si funciona correctamente por lo que no es necesario incluirla.

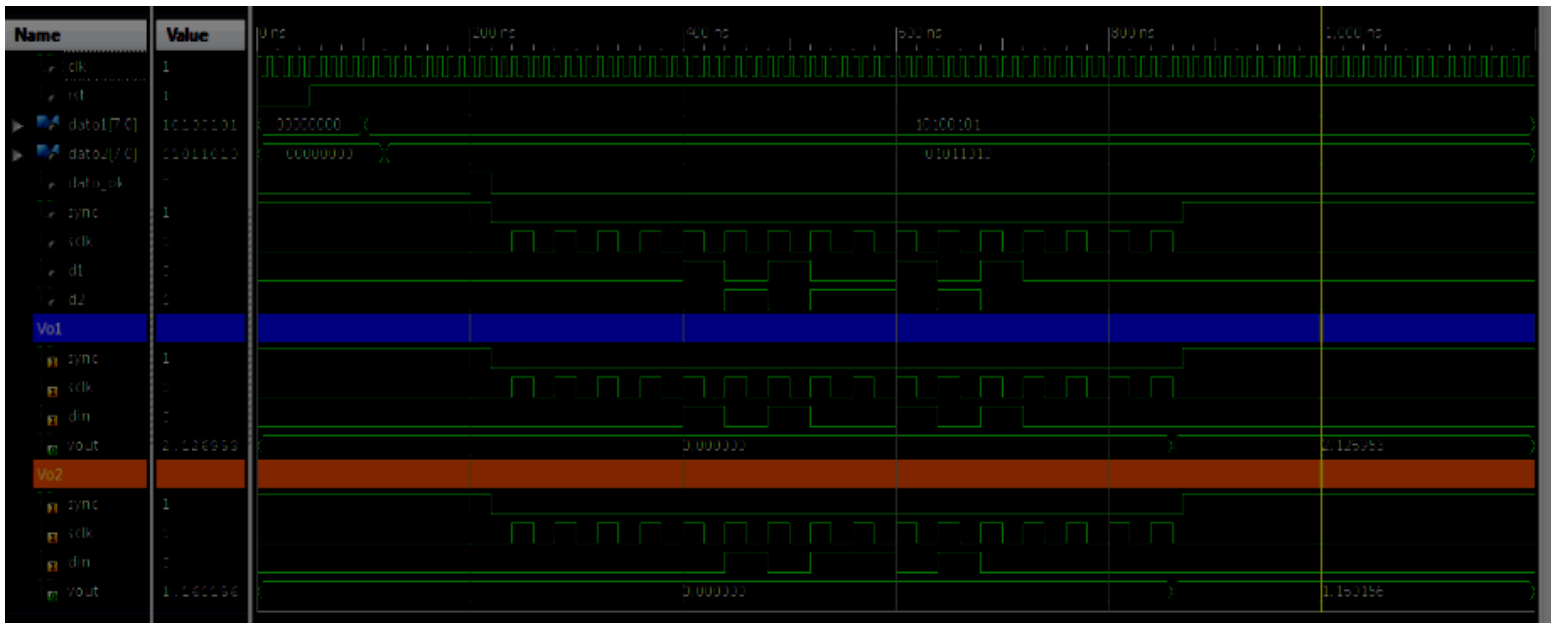
Una vez realizado dicho paso el esquema de simulación tendrá que quedar de la siguiente manera.



Después procedemos a realizar la simulación temporal y funcional con los nuevos componentes para así poder verificar que su funcionamiento es el correcto, en caso de que las restricciones de tiempo o de funcionamiento no se cumplan el nuevo componente muestra por pantalla un mensaje avisando del error.

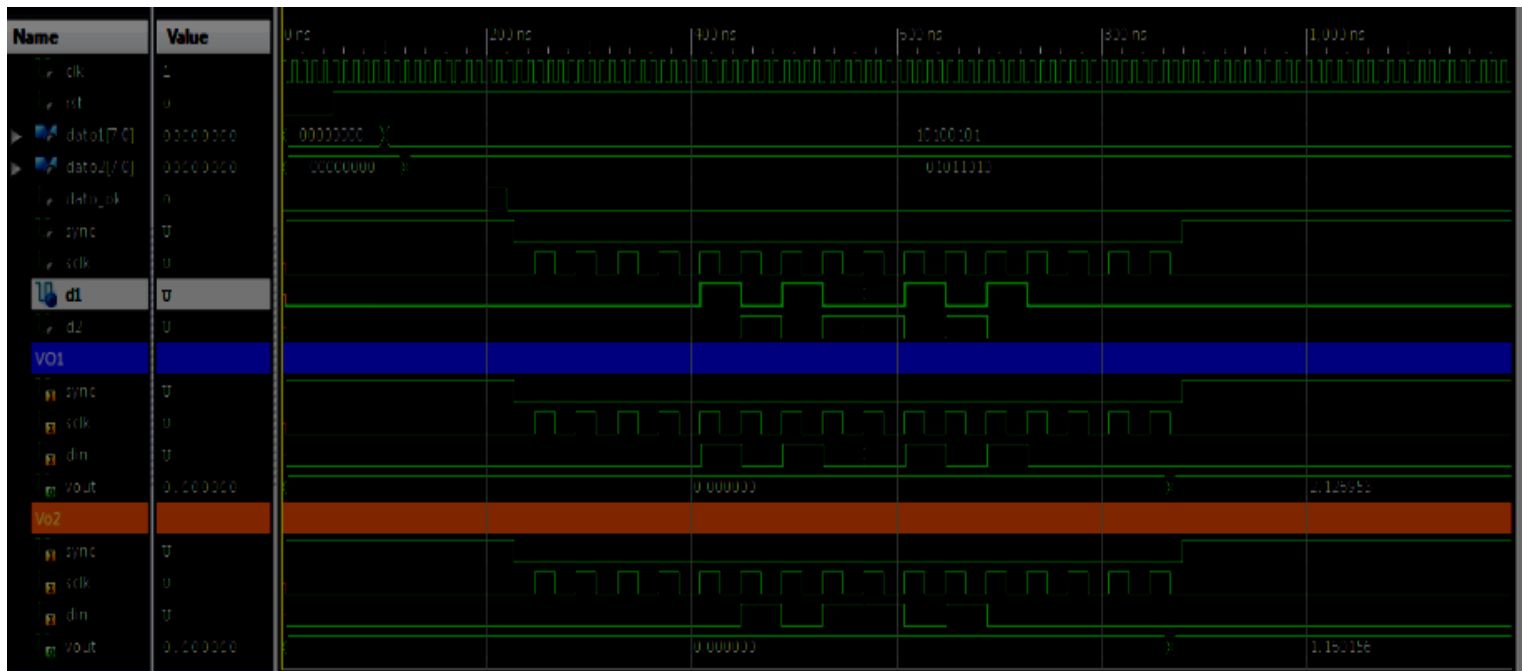
### **Simulación Funcional**

Con los resultados de las simulaciones tanto funcionales como temporales podemos verificar si el circuito cumple con las restricciones impuesta y de ser este el caso se podría decir que el apartado está completo.



### Simulación Temporal

En esta simulación se tienen en cuenta los retardos por lo que es mas propensa al fallo y en ese caso sera necesario modificar el circuito.



### Resultados de las simulación Temporal

Running: fuse.exe -relaunch -intstyle "ise" -incremental -lib "secureip" -o  
 "C:/Users/pedro/GIC/PracticaLibre/Practica/cnt\_dac\_tb\_isim\_par.exe" -prj  
 "C:/Users/pedro/GIC/PracticaLibre/Practica/cnt\_dac\_tb\_par.prj" "work.cnt\_dac\_tb"

```

ISim P.20131013 (signature 0x7708f090)
Number of CPUs detected in this system: 2
Turning on mult-threading, number of parallel sub-compilation jobs: 4
Determining compilation order of HDL files
Parsing VHDL file "C:/Users/pedro/GIC/PracticaLibre/Practica/./Souce/DAC121S101.vvvhd" into
library work
Parsing VHDL file "C:/Users/pedro/GIC/PracticaLibre/Practica/netgen/par/cnt_dac_timesim.vhd"
into library work
Parsing VHDL file "C:/Users/pedro/GIC/PracticaLibre/Practica/./Souce/cnt_dac_tb.vhd" into
library work
Starting static elaboration
Completed static elaboration
Compiling package standard
Compiling package std_logic_1164
Compiling package textio
Compiling package vital_timing
Compiling package vcomponents
Compiling package vital_primitives
Compiling package vpackage
Compiling package numeric_std
Compiling architecture x_ckbuf_v of entity X_CKBUF [\X_CKBUF(true,true,"UNPLACED",(0...)]
Compiling architecture x_obuf_v of entity X_OBUF [\X_OBUF(true,true,"DONT_CARE",12...)]
Compiling architecture x_buf_v of entity X_BUF [\X_BUF(true,true,"UNPLACED",(0,0...)]
Compiling architecture x_inv_v of entity X_INV [\X_INV(true,true,"PAD222",(0,0),...)]
Compiling architecture x_ff_v of entity X_FF [\X_FF(true,true,true,"UNPLACED",...)]
Compiling architecture x_mux2_v of entity X_MUX2 [\X_MUX2(true,true,"UNPLACED",(0,...)]
Compiling architecture x_zero_v of entity X_ZERO [\X_ZERO("UNPLACED")(1,8)]
Compiling architecture x_lut6_v of entity X_LUT6 [\X_LUT6(true,true,"UNPLACED",(0,...)]
Compiling architecture x_lut5_v of entity X_LUT5 [\X_LUT5(true,true,"UNPLACED",(0,...)]
Compiling architecture x_one_v of entity X_ONE [\X_ONE("UNPLACED")(1,8)]
Compiling architecture x_roc_v of entity X_ROC [\X_ROC("UNPLACED",100000,"*")(1,...)]
Compiling architecture x_toc_v of entity X_TOC [\X_TOC("UNPLACED",0,"*")(1,8,1,1...)]
Compiling architecture structure of entity cnt_dac [cnt_dac_default]
Compiling architecture rtl of entity DAC121S101 [dac121s101_default]
Compiling architecture for_dac_tb of entity cnt_dac_tb
Time Resolution for simulation is 1ps.
Waiting for 15 sub-compilation(s) to finish...
Compiled 211 VHDL Units
Built simulation executable C:/Users/pedro/GIC/PracticaLibre/Practica/cnt_dac_tb_isim_par.exe
Fuse Memory Usage: 79716 KB
Fuse CPU Usage: 2729 ms
===== Fuse: succeeded =====

```

Una vez comprobado que el circuito funciona y que se cumplen las restricciones tanto temporales como de diseño podemos dar por concluido este apartado.

## Memoria MEMORIA DUAL PORT

La memoria dual por será la encargada de almacenar un periodo de cada una de las señales a generar. Esta memoria tendrá una organización de 256x8 bits, teniendo un puerto de sólo escritura y otro de sólo lectura. Su finalidad tal y como se ha dicho es almacenar en las direcciones que recibe el contenido que es enviado desde los otros apartados del programa pudiendo posteriormente ser leído dicho dato.

Las entradas y salidas de este apartado son las citadas a continuación además se especifica el tipo de dato que son y si son de entrada o de salida:

```
DIN : in std_logic_vector(7 downto 0);
ADDR_IN : in std_logic_vector(7 downto 0);
WE : in std_logic;
CLK : in std_logic;
RST : in std_logic;
ADDR_OUT : in std_logic_vector(7 downto 0);
DOUT : out std_logic_vector(7 downto 0);
```

La finalidad de los pines citados es:

**DIN.** Bus de datos correspondiente al puerto de solo escritura.

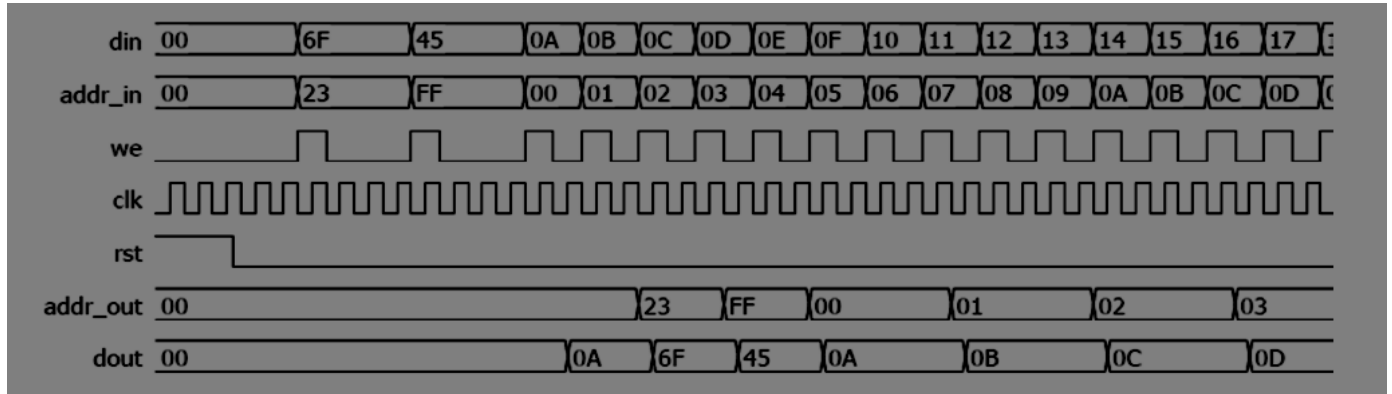
**ADDR\_IN.** Bus de dirección del puerto de sólo escritura.

**WE.** Señal de validación de la operación de escritura. Es activa a nivel alto.

**DOUT.** Bus de datos correspondientes al puerto de solo lectura.

**ADDR\_OUT.** Bus de dirección del puerto de sólo lectura.

El esquema de funcionamiento de la memoria es suministrado por el profesor por lo que solo es necesario comprender su funcionamiento y como ha de utilizarse correctamente y obteniendo finalmente un resultado parecido al descrito en la imagen.

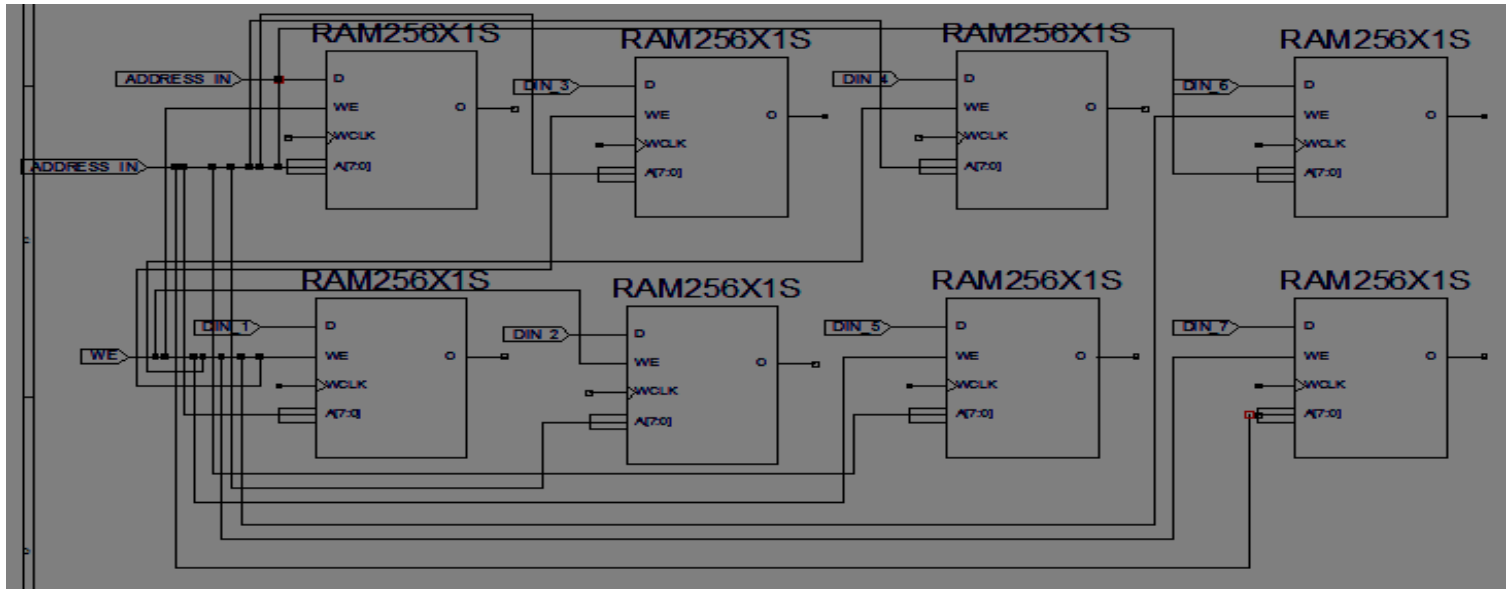


Este código a sido suministrado por el profesor por lo que solo se ha de entender si funcionamiento y verificar si funciona correctamente en los casos posibles.

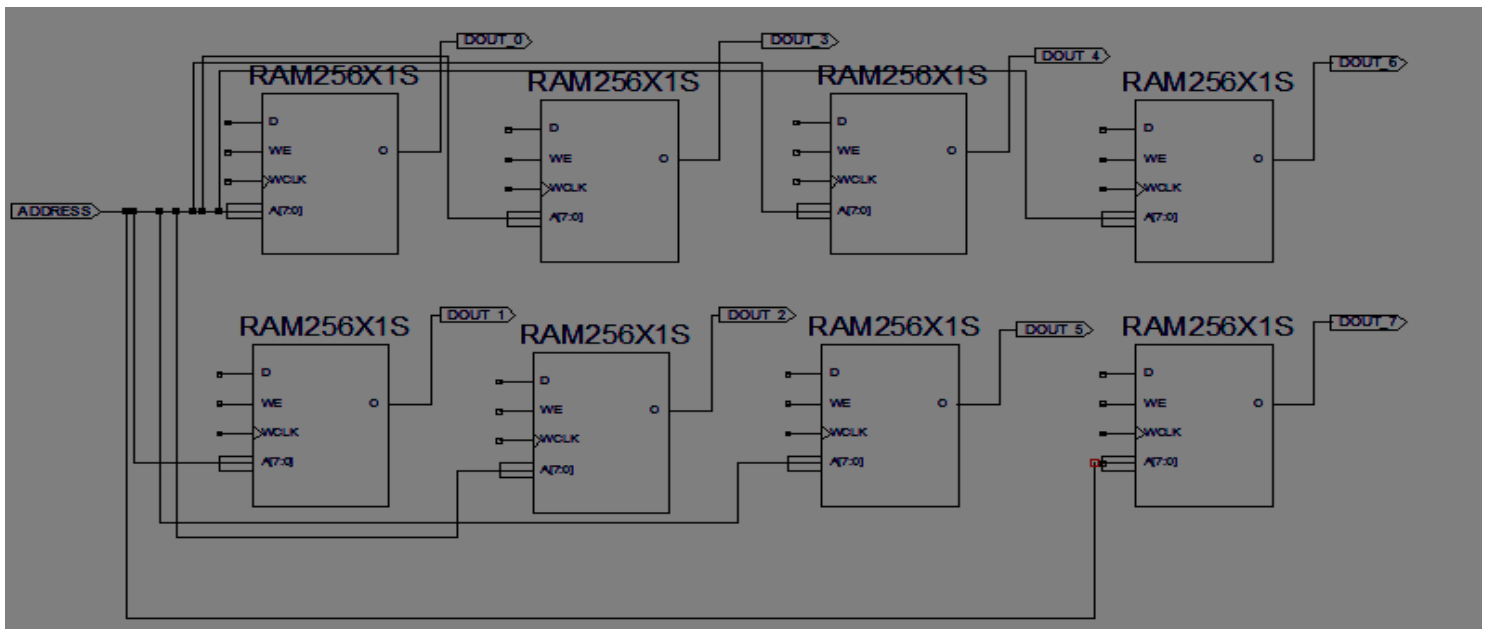


## Circuitos digitales y funcionamiento

**Escritura:** Este circuito seria algo aproximado a lo mostrado en la imagen adjunta a continuación, su finalidad es que una vez que la dirección donde se quiere almacenar la información y que la información esta preparada se produce la señal de WE la cual permite guardar el contenido de DIN en la dirección que queremos.



**Lectura:** De la misma manera que en la escritura en este acaso cuando queremos ver el contenido de la memoria solo hemos de insertar la dirección donde queremos hacer la lectura y se volcara el contenido de esta en la salida.



Tanto en la escritura como en la lectura las acciones se realizan durante CLK'event and CLK = '1' pero no se ha implementado para simplificar el dibujo.

### Código Memoria dual

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

entity dpram_mem is
  port (
    DIN    : in  std_logic_vector(7 downto 0);
    ADDR_IN : in  std_logic_vector(7 downto 0);
    WE      : in  std_logic;
    CLK     : in  std_logic;
    RST     : in  std_logic;
    ADDR_OUT : in  std_logic_vector(7 downto 0);
    DOUT    : out std_logic_vector(7 downto 0));
end entity;

architecture rtl of dpram_mem is
  type mem_type is array (255 downto 0) of std_logic_vector(7 downto 0);
  signal mem : mem_type;
  attribute ram_style : string;
  attribute ram_style of mem : signal is "block";

begin

  process (CLK) is
  begin
    if CLK'event and CLK = '1' then
      if WE='1' then
        mem(to_integer(unsigned(ADDR_IN))) <= DIN;
      end if;
    end if;
  end process;

  process (CLK, RST) is
  begin
    if RST = '1' then
      DOUT <= (others => '0');
    elsif CLK'event and CLK = '1' then
      DOUT <= mem(to_integer(unsigned(ADDR_OUT)));
    end if;
  end process;

end rtl;
```

### Código TestBench Memoria\_dual

Con este código se busca probar que todos los casos posibles tanto de lectura como de escritura funcionan correctamente.

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

-----

entity dpram_mem_tb is

end entity dpram_mem_tb;

-----

architecture dpram_mem of dpram_mem_tb is

    -- component ports
    signal DIN      : std_logic_vector(7 downto 0) :=(others =>'0');
    signal ADDR_IN  : std_logic_vector(7 downto 0) :=(others =>'0');
    signal WE       : std_logic:= '0';
    signal CLK      : std_logic:= '1' ;
    signal RST      : std_logic;
    signal ADDR_OUT : std_logic_vector(7 downto 0) :=(others =>'0');
    signal DOUT     : std_logic_vector(7 downto 0);

begin -- architecture dpram_mem

    -- component instantiation
    DUT: entity work.dpram_mem
    port map (
        DIN      => DIN,
        ADDR_IN  => ADDR_IN,
        WE       => WE,
        CLK      => CLK,
        RST      => RST,
        ADDR_OUT => ADDR_OUT,
        DOUT     => DOUT);

    -- clock generation
    CLK <= not CLK after 5 ns;
    RST <= '1', '0' after 50 ns;

    process is
    begin -- process escritura donde se realizan las inserciones de los datos
        wait for 100 ns;
        for i in 0 to 255 loop
            DIN <= std_logic_vector(to_unsigned(35+i, 8));
            ADDR_IN <= std_logic_vector(to_unsigned(i, 8));
```

```

    WE <= '1';
    wait for 10 ns;
    WE <= '0';
    wait for 10 ns;
end loop; -- i
end process;

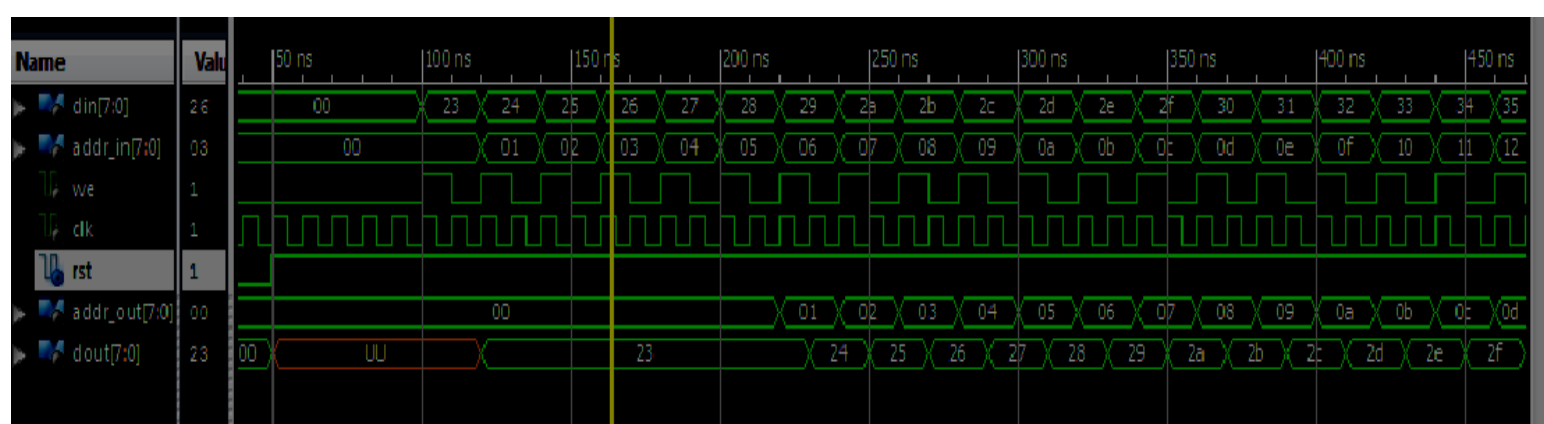
process is
begin -- process lectura de donde se leen los datos ya escritos
    wait for 200 ns;
    for o in 0 to 255 loop
        ADDR_OUT <= std_logic_vector(to_unsigned(o, 8));
        wait for 20 ns;
    end loop; -- o
end process;

end architecture dpram_mem;

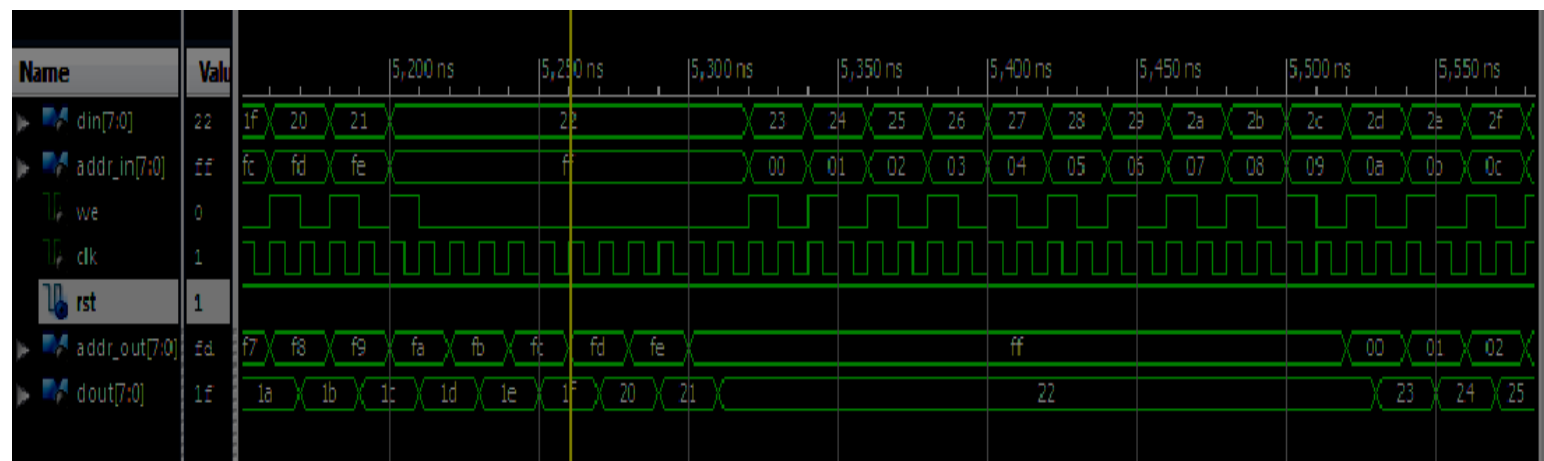
```

## Simulación Funcional

En el diagrama funcional del circuito se busca implementar todos los posibles casos que hay para así verificar el correcto funcionamiento de circuito y así poder testearlo con el código de pruebas y verificar que se cumplen los requisitos.



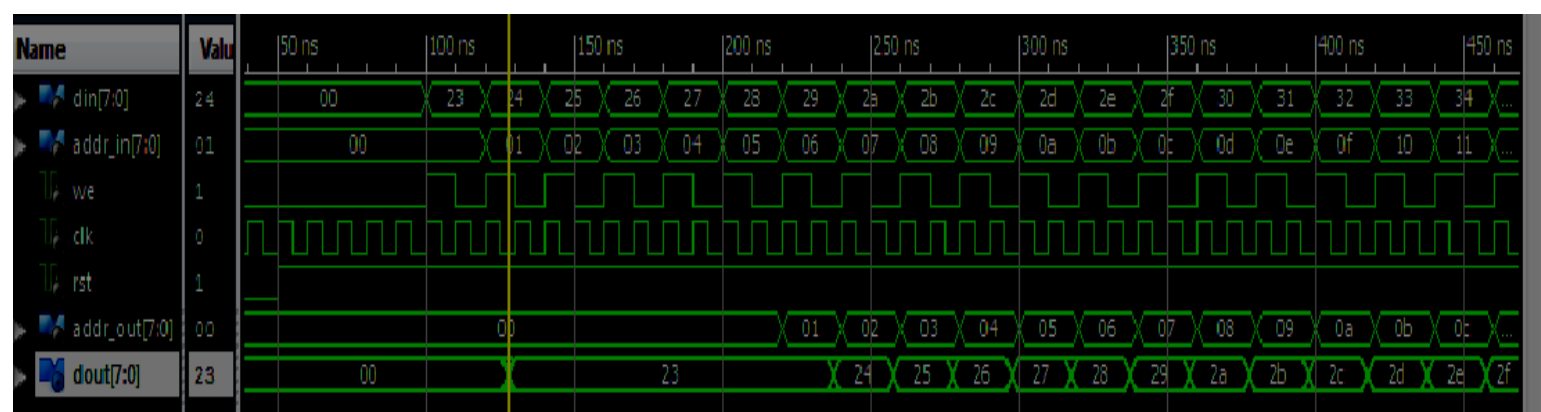
Por el gran tamaño de los casos a comprobar solo se muestra una captura genérica con todo en conjunto, En el siguiente diagrama funcional se muestra como cambia al superar el limite de las 255 posiciones.



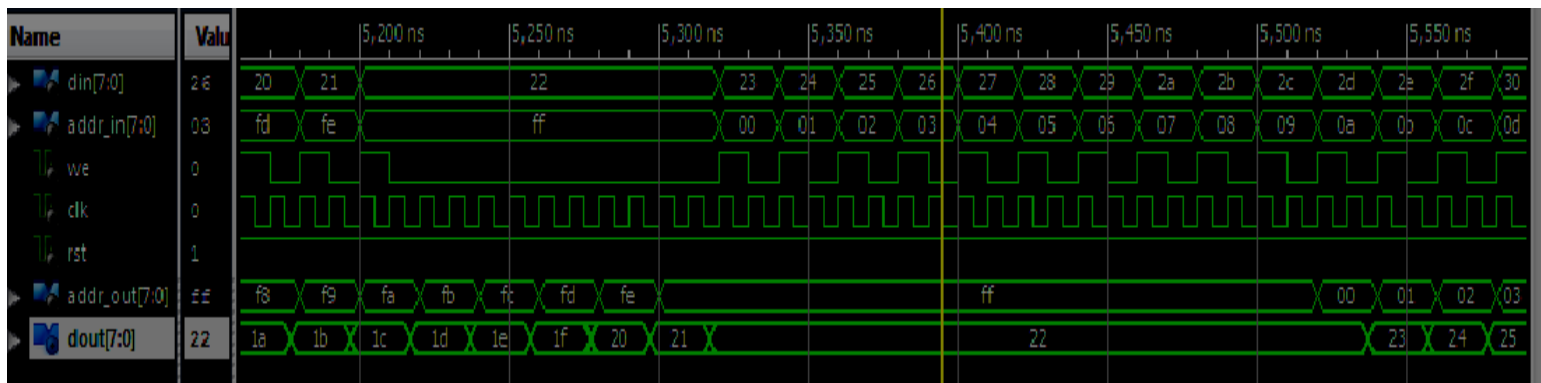
Una vez verificado este apartado podemos realizar el post-place&route para comprobar que el circuito que hemos diseñado cumple con las características en el apartado temporal.

## Simulación Temporal

En el diagrama temporal del circuito se busca implementar todos los posibles casos que hay para así verificar el correcto funcionamiento de circuito y así poder testearlo con el código de pruebas y verificar que se cumplen los requisitos este esquema es mas propenso al fallo ya que se tienen en cuenta los retardos de las puerta lo cual puede hacer que el circuito que en el funcional iba perfectamente necesite cambios para poder llegar a funcionar bien.

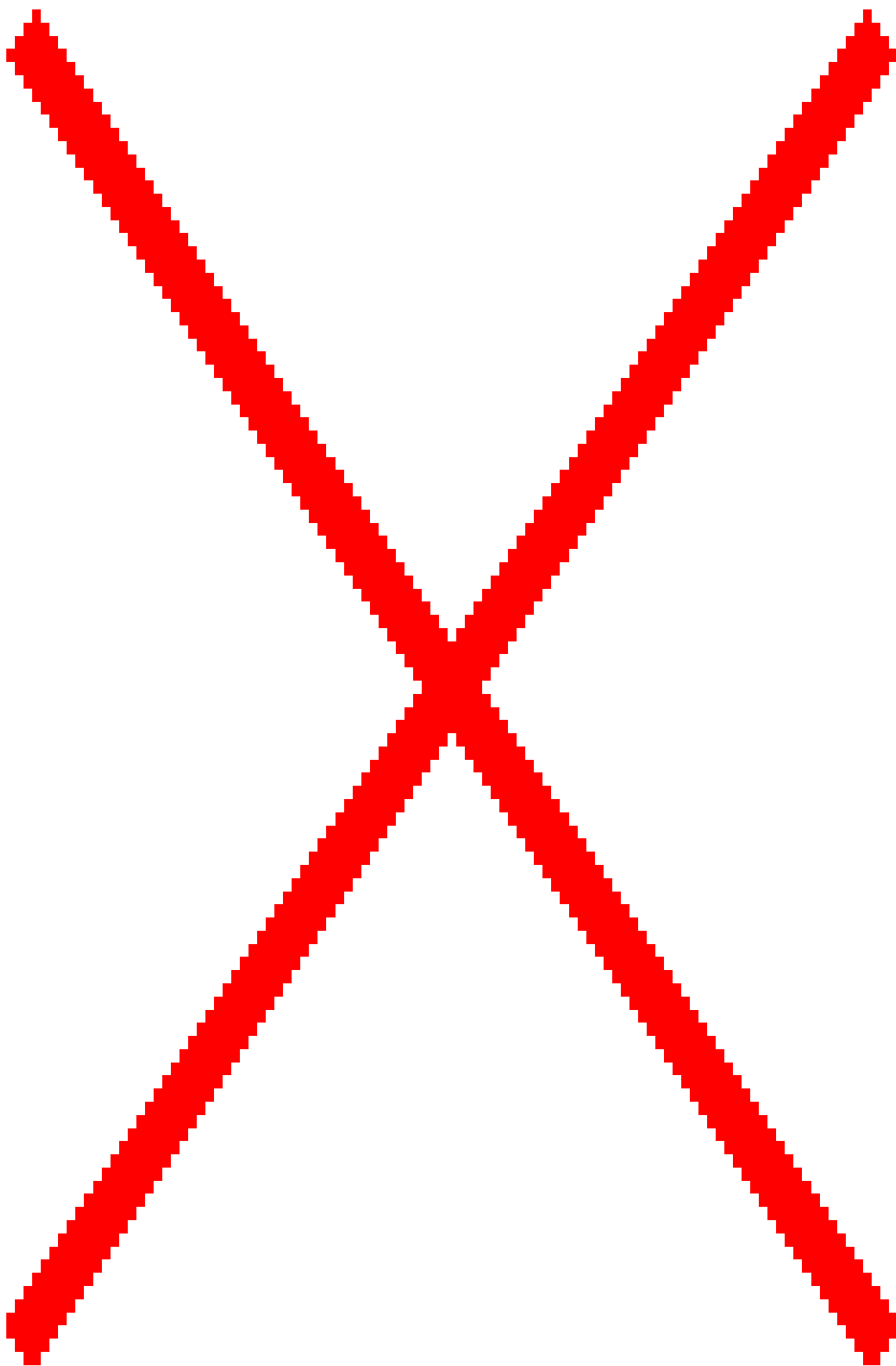


Por el gran tamaño de los casos a comprobar solo se muestra una captura genérica con todo en conjunto, En el siguiente diagrama funcional se muestra como cambia al superar el limite de las 255 posiciones.



Una vez comprobado que todos los casos se resuelven correctamente podemos dar por concluido este apartado del trabajo.

### Recursos utilizados



## Memoria CNT DPRAM

En este apartado se busca transmitir la información que se recibe de los módulos anteriores a las memoria duales para poder almacenar dicha información, para poder llevar a cabo esta misión se han de cumplir ciertas restricciones ya que de no ser así el almacenamiento de la memoria no sería el adecuado.

- Cada dato a almacenar en la memoria dual port se proporciona con un ciclo de escritura en la dirección correspondiente.
- En la memoria dual port 1 se escribirán los datos cuya dirección es A1<sub>HEX</sub> y en la dual port 2 los correspondientes a la dirección A2<sub>HEX</sub>.
- A medida que se van enviando datos con la misma dirección, se irán almacenando en posiciones consecutivas.
- Una vez almacenado un dato en la última posición de memoria (FF<sub>HEX</sub>), si se envía datos manteniendo la misma dirección, estos se irán almacenando desde la primera posición (00<sub>HEX</sub>) y en posiciones consecutivas.
- Cuando se realiza un ciclo de escritura en una dirección diferente al anterior ciclo se entiende que el dato se debe almacenar en la posición 0 de la memoria que corresponde con dicha dirección. Siempre y cuando esta dirección se corresponda con una de las dos utilizadas en este diseño: A1<sub>HEX</sub> y A2<sub>HEX</sub>.
- Las memorias pueden ser escritas parcialmente

Las entradas y salidas de este apartado son las citadas a continuación además se especifica el tipo de dato que son y si son de entrada o de salida:

```
CLK : in std_logic;  
RST : in std_logic;  
DIR : in std_logic_vector(7 downto 0);  
DIR_VLD : in std_logic;  
DATO : in std_logic_vector(7 downto 0);  
DATO_VLD : in std_logic;  
ADDRESS : out std_logic_vector(7 downto 0);  
DATA : out std_logic_vector(7 downto 0);  
WE_DP1 : out std_logic;  
WE_DP2 : out std_logic;
```

Señales auxiliares utilizadas en el programa para poder realizar las interconexiones entre las distintas partes de los componentes:

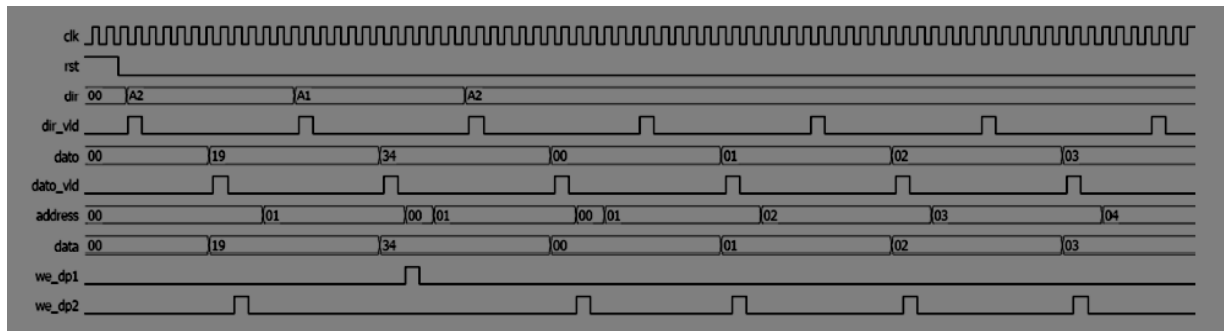
```
constant dir_dpram1 : std_logic_vector(7 downto 0) := x"A1";  
constant dir_dpram2 : std_logic_vector(7 downto 0) := x"A2";  
signal dir_ant : std_logic_vector(7 downto 0);  
signal CEB : std_logic;  
signal REC : std_logic;  
signal CEC : std_logic;
```

Declaración de la máquina de estados:

```
type MEF is (REP, ESP, ESW, ESD, RES, ESC);  
signal std_act, prox_std : MEF;
```

Con esta información podemos empezar a realizar el montaje del módulo cnt\_dpram el cual ha de dar un resultado parecido al de la gráfica mostrada a continuación.

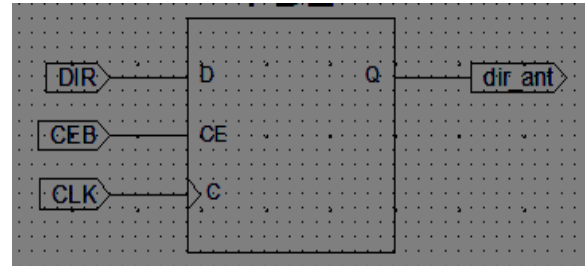




## CIRCUITOS DIGITALES

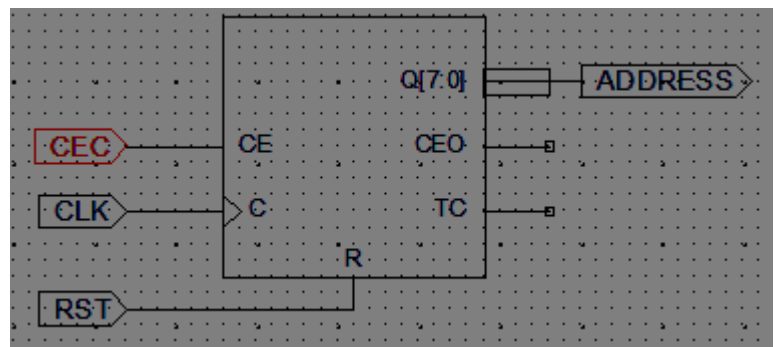
**Nombre:** Biestable tipo d

**Finalidad:** La función de este componente es la de almacenar la dirección que se a utilizado en el paso anterior de forma que en caso de ser la misma se realice un paso o si por el contrario la dirección es distinta entonces se realiza otro paso distinto. Se habilita mediante una señal de la maquina de estados



**Nombre:** Contador de 8 bits

**Finalidad:** Este contador se usa para pasar la dirección donde se debe escribir el dato que se envía para ello se han de mantener ciertos requisitos, descritos al principio de la practica, algunos de ellos son que en caso de que la dirección anterior no sea la misma que la actual se ha de resetear la cuenta o en caso de llegar al limite de 255 se ha de empezar de 0. Se habilita mediante uno de los estados de la maquina de estado.



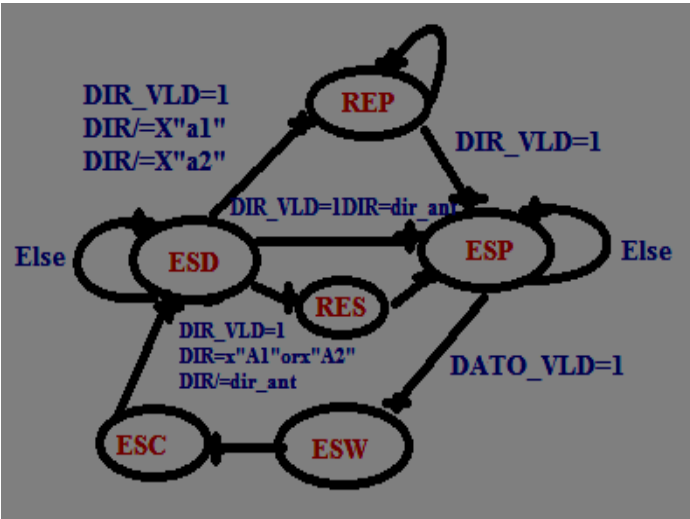
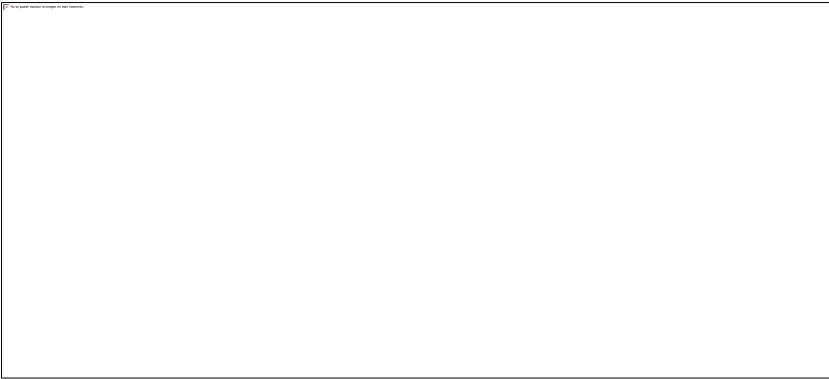
**Nombre:** Maquina de estados

**Finalidad:** Se utiliza para poder modelar ciertas restricciones o comportamiento de forma mas sencilla y efectiva.

Apartado 1: En este paso se calcula cual sera el siguiente paso a realizar en función de las entradas actuales y del estado en el que se encuentra ahora mismo, ademas en este caso ciertos pasos han de cumplir la restricción de que las direcciones actuales han de ser x"A1" o x"A2".

Apartado 2: En este paso se pasa de un apartado a otro en función del pulso de reloj, la base de este apartado es un biestable d que cambia de posición en función del CLK y la entrada.

Apartado 3: Aquí se habilitan las salidas en función del estado y de una ciertas condiciones, CEC se habilita solo cuando estamos en el estado de ESC, CEB se habilita en el estado de ESP, REC se habilita tanto en el estado de RES como en el de REP y por ultimo WE\_DP1/2 cuando estamos en el estado de escritura y se cumple que la direcciones x"A1" o x"A2".



**Otros:** DATA contiene el contenido de DATO ya que no es necesaria ninguna restricci3n ya que aunque el dato no este en la direcci3n correcta no se va a copiar a no ser de que la direcci3n sea valida.

### C3digo Cnt\_drpam

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

entity cnt_drpam is
port (
  CLK    : in std_logic;
  RST    : in std_logic;
  DIR    : in std_logic_vector (7 downto 0);
  DIR_VLD : in std_logic;
  DATO    : in std_logic_vector (7 downto 0);
  DATO_VLD : in std_logic;
  ADDRESS : out std_logic_vector(7 downto 0);
  DATA   : out std_logic_vector(7 downto 0);
  WE_DP1  : out std_logic;
  WE_DP2  : out std_logic);
end cnt_drpam;

```

```

architecture RTL of cnt_drpam is
  constant dir_drpam1 : std_logic_vector(7 downto 0) := x"A1";
  constant dir_drpam2 : std_logic_vector(7 downto 0) := x"A2";

  signal dir_ant : std_logic_vector(7 downto 0); -- direcci3n anterior para saber si continua en la
  misma
  signal CEB    : std_logic; -- se1al de enable biestable
  signal REC    : std_logic; -- reset contador cuando no se han cumplido las condiciones
  signal CEC    : std_logic; -- clock enable contador

```

```

--Maquina de estados REP-reposo ESP-espera ESW-estado escribiendo ESD-esperando direcci3n
--RES-reset ESC-estado contador
type MEF is (REP, ESP, ESW, ESD, RES, ESC);
signal std_act, prox_std : MEF;

```

```

begin -- RTL

--biestable para guardar la dir anterior
process (CLK, RST) is
begin -- process
  if RST = '1' then          -- asynchronous reset (active low)
    dir_ant <= (others => '0');
  elsif CLK'event and CLK = '1' then -- rising clock edge
    if CEB = '1' then        --FALTA
      dir_ant <= DIR;
    end if;
  end if;
end process;

--contador de 8 bits para las 255 direcciones
process (CLK, RST, REC) is
  variable cnt : std_logic_vector(7 downto 0);
begin -- process
  if RST = '1' then          -- asynchronous reset (active low)
    ADDRESS <= (others => '0');
    cnt := x"00";
  elsif REC='0' then
    ADDRESS <= (others => '0');
    cnt := x"00";
  elsif CLK'event and CLK = '1' then -- rising clock edge
    if CEC = '1' then
      if cnt = x"FF" then
        cnt := x"00";
        ADDRESS <= std_logic_vector(unsigned(cnt));
      else
        cnt := std_logic_vector(unsigned(cnt)+1);
        ADDRESS <= std_logic_vector(unsigned(cnt));
      end if;
    end if;
  end if;
end process;

--transferimos el contenido de dato a data sin miedo ya que si no es una
--dirección valida no se va a copiar aunque este ahí
DATA <= DATO;

--Parte 1 Maquina de estados finitos
process (DIR_VLD, DATO_VLD, std_act, DIR, dir_ant) is
begin
  case std_act is
    when REP =>
      if DIR_VLD = '1' and (DIR = dir_dpram1 or DIR = dir_dpram2) then
        prox_std <= ESP;
      else
        prox_std <= REP;
      end if;
    when ESP =>

```

```

    if DATO_VLD = '1' then
        prox_std <= ESW;
    else
        prox_std <= ESP;
    end if;
    when ESW => prox_std <= ESC;
    when ESC => prox_std <= ESD;
    when ESD =>
        if DIR_VLD = '1' and DIR = dir_ant then
            prox_std <= ESP;
        elsif DIR_VLD = '1' and DIR /= dir_ant then
            prox_std <= RES;
        elsif DIR_VLD = '1' and DIR /= dir_dpram1 and DIR /= dir_dpram2 then
            prox_std <= REP;
        else
            prox_std <= ESD;
        end if;
        when RES => prox_std <= ESP;
    end case;
end process;

--parte 2 maquina de estados
process (CLK, RST) is
begin -- process
    if RST = '1' then          -- asynchronous reset (active low)
        std_act <= REP;
    elsif CLK'event and CLK = '1' then -- rising clock edge
        std_act <= prox_std;
    end if;
end process;

--parte "3" maquina de estados
CEC <= '1' when std_act=ESC else '0';
CEB <= '1' when std_act=ESP else '0';
REC <= '0' when (std_act=RES or std_act=REP) else '1';
WE_DP1 <= '1' when std_act = ESW and DIR = dir_dpram1 else '0';
WE_DP2 <= '1' when std_act = ESW and DIR = dir_dpram2 else '0';

end RTL;

```

### **TestBench Genérico**

Después de ello realizamos el TestBench para verificar su correcto funcionamiento para ello creamos el archivo cnt\_drpam\_tb en el cual implementamos toda la lógica necesaria para poder simular el comportamiento del componente y así poder comprobar si realiza todas las acciones de forma correcta.

El código TestBench con todos los datos seria el citado a continuación en este caso las señales se generan mediante un procedimiento a petición del profesor:

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

-----

```
entity cnt_dpram_tb is
```

```
end entity cnt_dpram_tb;
```

-----

```
architecture cnt_dpram of cnt_dpram_tb is
```

```
-- component ports
signal CLK    : std_logic := '1';
signal RST    : std_logic;
signal DIR    : std_logic_vector (7 downto 0):= (others => '0');
signal DIR_VLD : std_logic :='0';
signal DATO   : std_logic_vector (7 downto 0):= (others => '0');
signal DATO_VLD : std_logic :='0';
signal ADDRESS : std_logic_vector(7 downto 0);
signal DATA   : std_logic_vector(7 downto 0);
signal WE_DP1  : std_logic;
```

```

signal WE_DP2 : std_logic;

begin -- architecture cnt_dpram

-- component instantiation
DUT: entity work.cnt_dpram
port map (
    CLK    => CLK,
    RST    => RST,
    DIR    => DIR,
    DIR_VLD => DIR_VLD,
    DATO    => DATO,
    DATO_VLD => DATO_VLD,
    ADDRESS => ADDRESS,
    DATA   => DATA,
    WE_DP1  => WE_DP1,
    WE_DP2  => WE_DP2);

-- clock generation
CLK <= not CLK after 5 ns;
RST <= '1', '0' after 50 ns;

process is

    procedure dpram(Dvalue:in std_logic_vector(7 downto 0);
                    Dat:in integer;
                    Daux:in integer)is
    begin
        wait for 60 ns;
        DIR <= Dvalue;
        DIR_VLD <= '1';
        wait for 10 ns;
        DIR_VLD <= '0';
        wait for 40 ns;
        DATO <= std_logic_vector(to_unsigned(Dat+Daux, 8));
        DATO_VLD <= '1';
        wait for 10 ns;
        DATO_VLD <= '0';
    end procedure;

begin
    wait for 100 ns;
    for io in 0 to 5 loop
        dpram(Dvalue => x"A1",Dat => 35,Daux => io);
    end loop;
    for iu in 0 to 10 loop
        dpram(Dvalue => x"A2",Dat => 213,Daux => iu);
    end loop;
    dpram(Dvalue => x"AA",Dat => 73,Daux => 0);
    for ie in 0 to 2 loop
        dpram(Dvalue => x"A2",Dat => 11,Daux => ie);
    end loop;
end process;

```

```

end loop;
dpram(Dvalue => x"AA",Dat => 30,Daux => 1);
for ia in 0 to 2 loop
  dpram(Dvalue => x"A1",Dat => 251,Daux => ia);
wait for 60 ns;
end loop;
assert True report "FINAL TEST" severity note;
end process;

```

```

end architecture cnt_dpram;

```

### **TestBench Todas las posiciones**

En este testbench se prueban todas las posiciones y que al llegar al final de ellas se empieza otra vez en la dirección 0. El contenido general es el mismo pero el proceso cambia por lo que solo se mostrara el contenido del proceso.

#### **Con Proceso:**

```

process is
begin
  for io in 0 to 255 loop
    wait for 100 ns;
    DIR <= x"A1";
    DIR_VLD <= '1';
    wait for 10 ns;
    DIR_VLD <= '0';
    wait for 40 ns;
    DATO <= std_logic_vector(to_unsigned(io, 8));
    DATO_VLD <= '1';
    wait for 10 ns;
    DATO_VLD <= '0';
  end loop;
end process;

```

#### **Con Procedure:**

```

process is

procedure dpram(Dvalue:in std_logic_vector(7 downto 0);
               Dat:in integer;
               Daux:in integer)is
begin
  wait for 60 ns;
  DIR <= Dvalue;

```

```

DIR_VLD <= '1';
wait for 10 ns;
DIR_VLD <= '0';
wait for 40 ns;
DATO <= std_logic_vector(to_unsigned(Dat+Daux, 8));
DATO_VLD <= '1';
wait for 10 ns;
DATO_VLD <= '0';
end procedure;

```

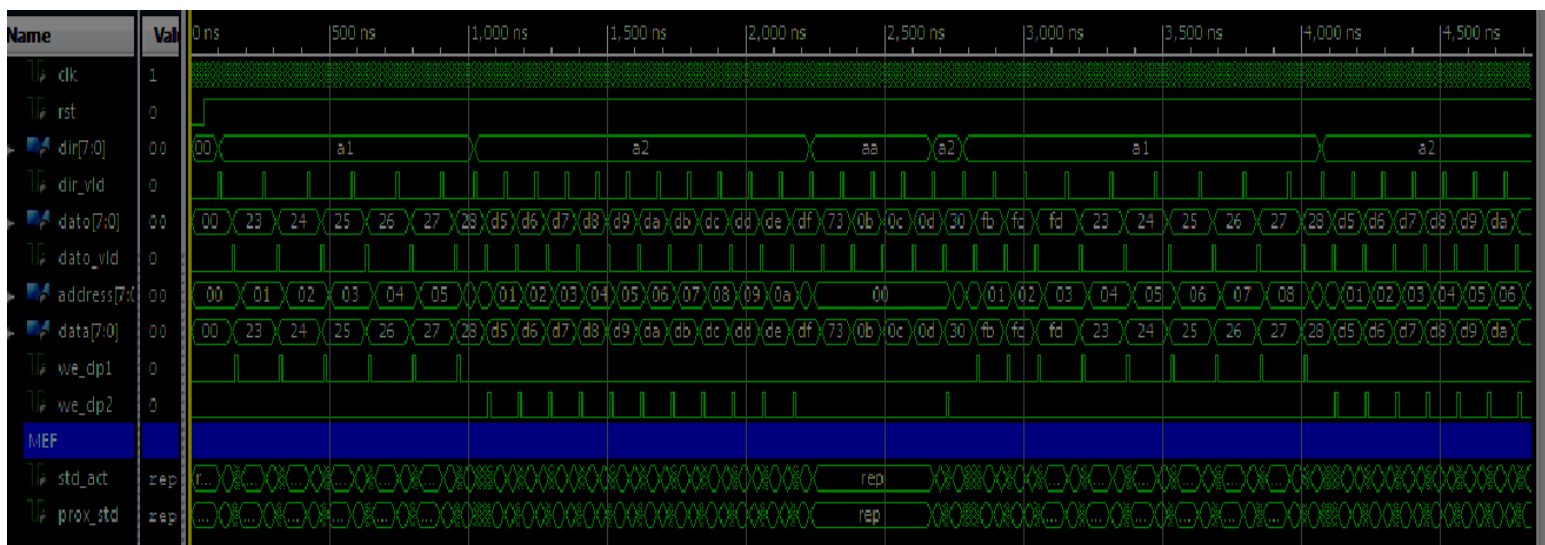
```

begin
wait for 100 ns;
for io in 0 to 255 loop
dpram(Dvalue => x"A1",Dat => 0,Daux => io);
end loop;
end process;

```

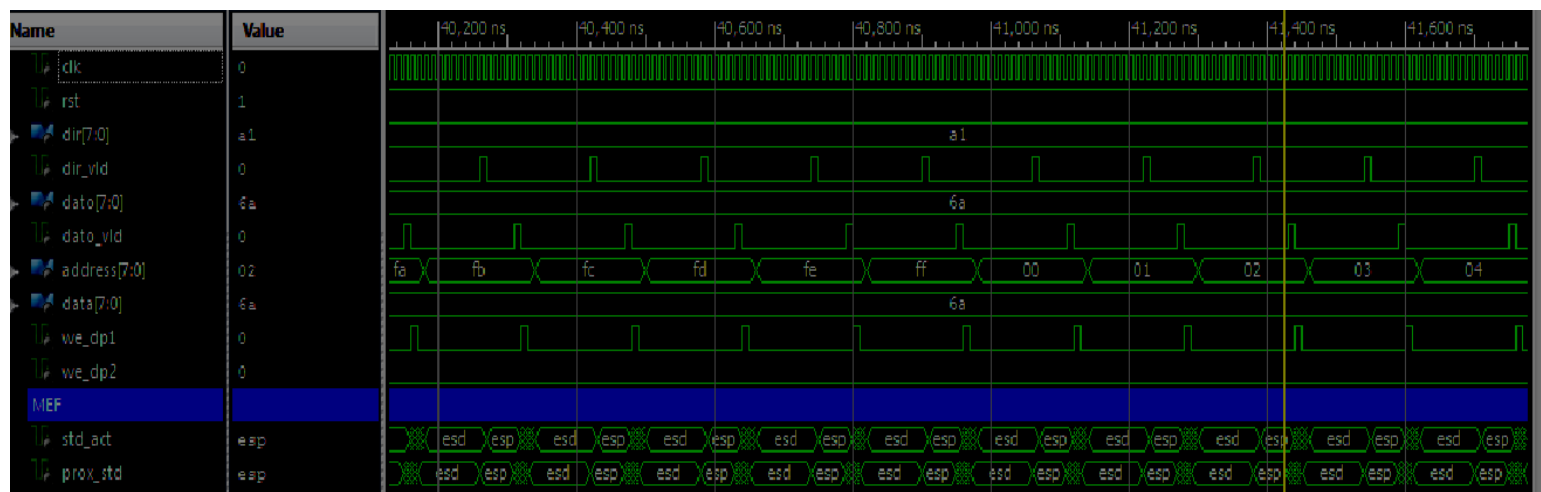
### Simulación Funcional

En el diagrama funcional del circuito se busca implementar todos los posibles casos que hay para así verificar el correcto funcionamiento de circuito y así poder testearlo con el código de pruebas y verificar que se cumplen los requisitos.



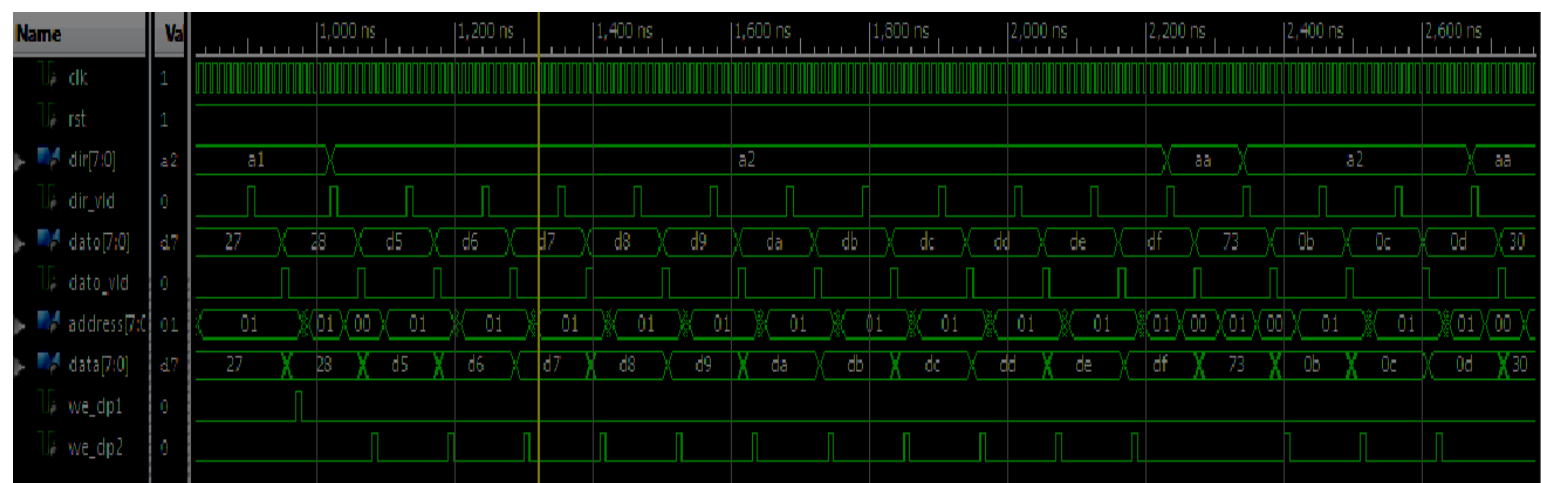
Por el gran tamaño de los casos a comprobar solo se muestra una captura genérica con todo en conjunto, En el siguiente diagrama funcional se muestra como cambia al superar el limite de las 255 posiciones.



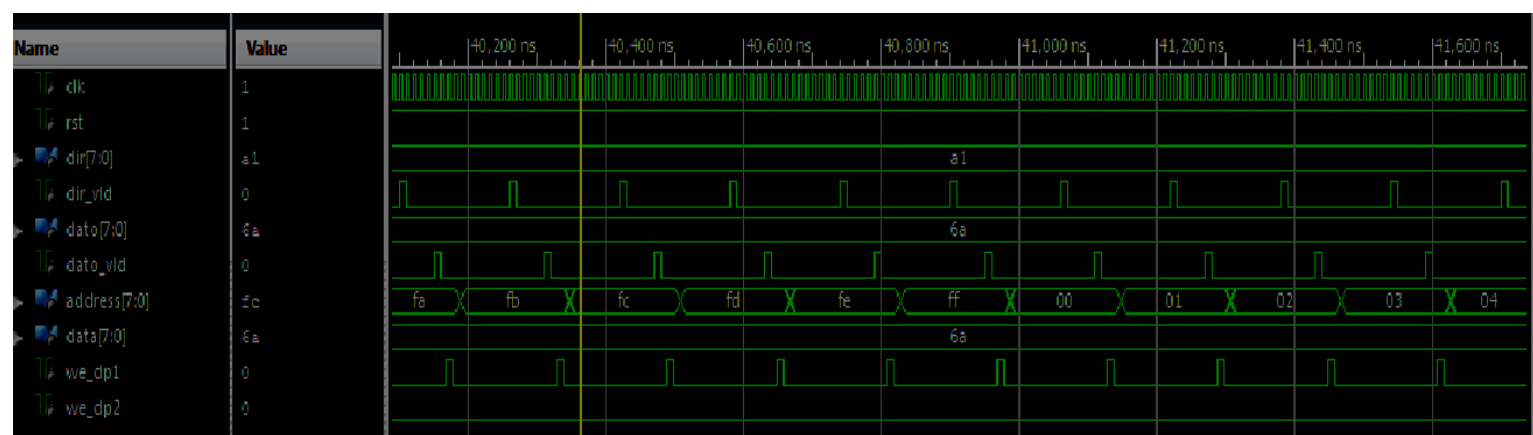


### Simulación Temporal

En el diagrama temporal del circuito se busca implementar todos los posibles casos que hay para así verificar el correcto funcionamiento de circuito y así poder testearlo con el código de pruebas y verificar que se cumplen los requisitos este esquema es mas propenso al fallo ya que se tienen en cuenta los retardos de las puerta lo cual puede hacer que el circuito que en el funcional iba perfectamente necesite cambios para poder llegar a funcionar bien.

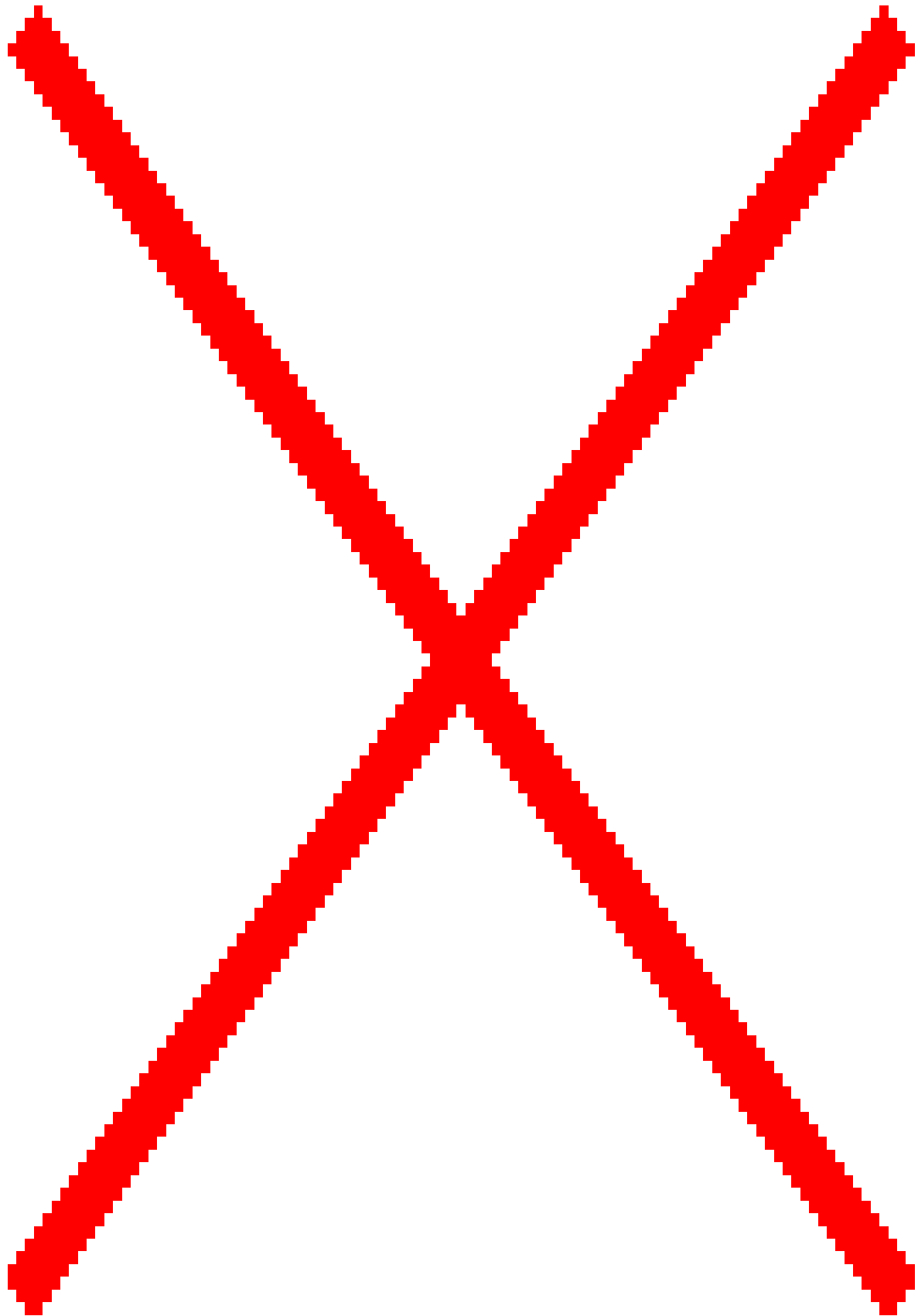


Por el gran tamaño de los casos a comprobar solo se muestra una captura genérica con todo en conjunto, En el siguiente diagrama funcional se muestra como cambia al superar el limite de las 255 posiciones.



Si cumple con estos requisitos especificados en el manual el modulo esta terminado.

### **Recursos Utilizados**



## Memoria GEN DIR

La entidad gen\_dir se encarga de generar las direcciones de los puertos de sólo lectura de las memorias dual port. El ritmo al que cambian estas direcciones es proporcional al valor del dato seleccionado desde el puerto paralelo al escribir en la dirección F0 HEX.

Las entradas y salidas de este apartado son las citadas a continuación además se especifica el tipo de dato que son y si son de entrada o de salida:

```
CLK : in std_logic;
RST : in std_logic;
DIR : in std_logic_vector (7 downto 0);
DIR_VLD : in std_logic;
DATO : in std_logic_vector (7 downto 0);
DATO_VLD : in std_logic;
ADDR_OUT : out std_logic_vector(7 downto 0);
DATO_OK : out std_logic;
```

Señales auxiliares Constantes y MEF utilizadas en el programa para poder realizar las interconexiones entre las distintas partes de los componentes:

```
--Constante
constant dir_freq      : std_logic_vector(7 downto 0) := x"F0";

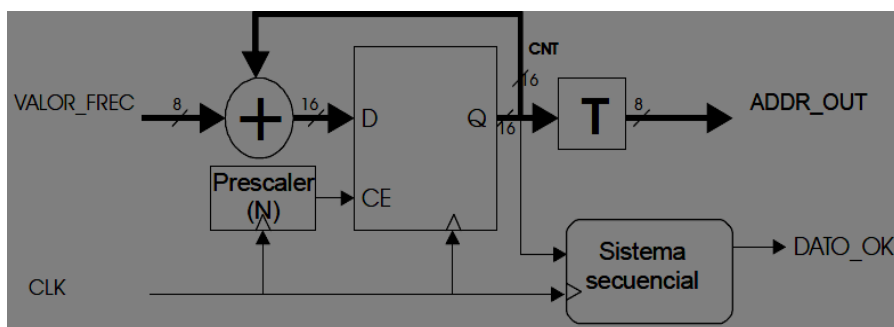
--Señales
signal Valor_freq      : std_logic_vector(7 downto 0);
signal Entrada_BD      : std_logic_vector(15 downto 0);
signal Salida_BD       : std_logic_vector(15 downto 0);
signal Salida_BD_ANT    : std_logic_vector(15 downto 0);
signal CEBD            : std_logic;

--Almacenado especial
signal DIRAUX          : std_logic_vector (7 downto 0);

--Contador
signal cnt              : unsigned (8 downto 0);

--Maquina de estados finitos
type MEF is (REP, ESP, DAOK);
signal std_act, prox_std : MEF;
```

Para modelar el funcionamiento de la entidad gen\_dir hemos de basarnos en el esquema suministrado por el profesor en el enunciado de la practica ya que ahí viene especificado como ha de comportarse para su correcto funcionamiento.



Para la realización de este apartado hemos de tener en cuenta los otros módulos que se han diseñado con anterioridad ya que su funcionamiento será el correcto siempre y cuando se adapte al conjunto y a las restricciones que nos impone el resto de los componentes.

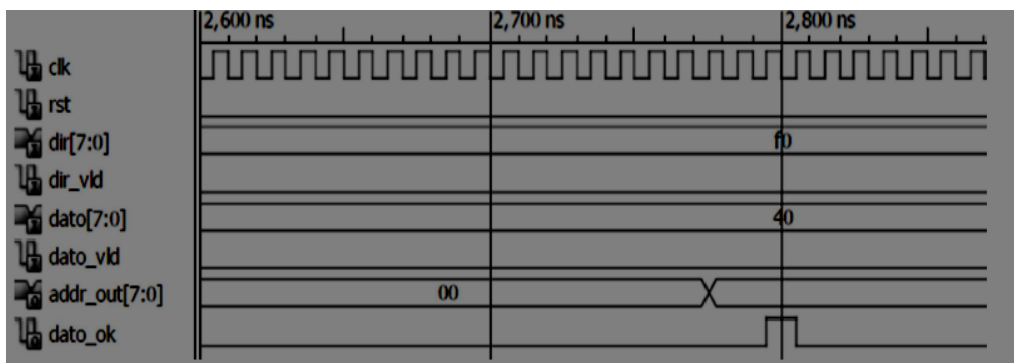
Una de las condiciones es el tiempo, el cual ha de cumplir con unas restricciones para su correcto funcionamiento, esas restricciones vienen de los apartados anteriores y su solución es:

$$f \approx \frac{VALOR\_FREC}{N * 2^{16}} * f_{CLK}$$

Donde fCLK representa la frecuencia de reloj de la señal de reloj (CLK) que en la placa que usamos es de 100MHz. El factor de división del prescaler (N) es igual al número mínimo de ciclos de reloj que se tarda en digitalizar un dato. Este valor, tal y como obtenemos en el apartado de **CNT\_DAC** ha de ser de mas de 66 ya que esta entidad necesita 64 para transmitir los 16 bits del dato y dos para llevar SYNC a nivel alto. Por lo que se le da un poco de margen y se utiliza una N=68 dejando espacio extra dejando ese tiempo para que la transferencia y carga en la memoria se haga efectiva.

El porque del uso de un biestable de 16bits al cual se le suma la entrada y la salida es porque las lecturas que se realizan en cierto intervalo de lecturas (cambia la salida del biestable) el cual como se ha dicho antes viene definido por el dato de la dirección F0Hex (Valor\_frec) , para la obtención de la dirección donde se ha de leer el dato usaremos los 8 bits de mayor peso el cual en función de el dato (Valor\_frec) cambiara a mayor o menor velocidad alterando la velocidad de la actualización, para que el incremento de los datos sea definido y controlado. El cambio de este dato también esta condicionado por el tiempo que tarda en serializarse un dato y por ello el mínimo va a ser de 68 ciclos en el mejor de los casos.

Para informar al módulo controlador de los DACS de que hay un nuevo dato a digitalizar, la entidad gen\_dir proporciona la salida DATA\_OK. Esta señal está retardada, con respecto a la dirección de la memoria dual port (ADDR\_OUT) un periodo de CLK en mi caso he optado por darle 2 ciclos de reloj en principio para asegurar la estabilidad del sistema ya que en la simulación temporal no se cumpliría el ciclo de espera completo, para compensar el tiempo de acceso de la memoria. Dejando así tiempo para evitar los posibles fallos que puede ocasionar si la señal no llega a tiempo a su destino.

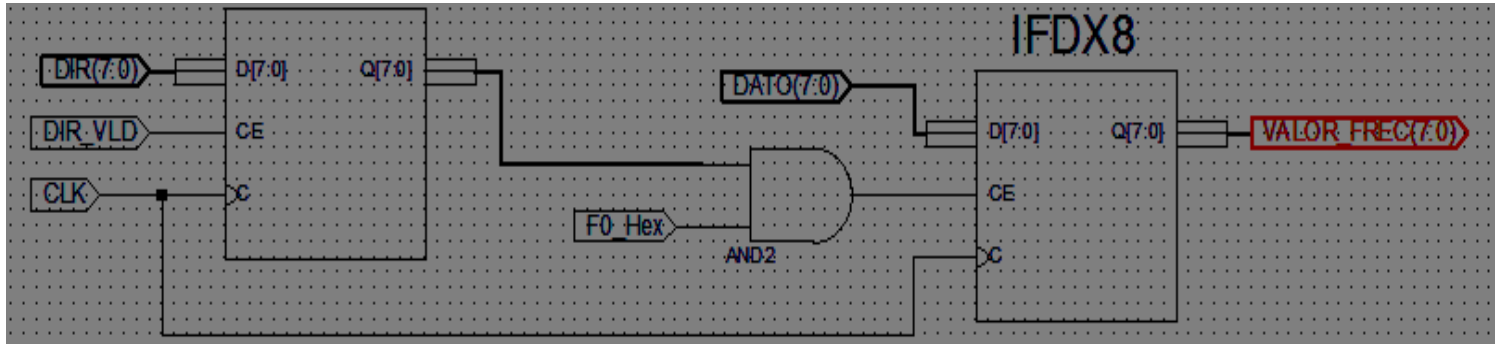


Teniendo en cuenta estas restricciones podremos modelar el funcionamiento de lo que seria el componente o apartado gen\_dir de nuestra practica.

### Circuitos digitales necesarios

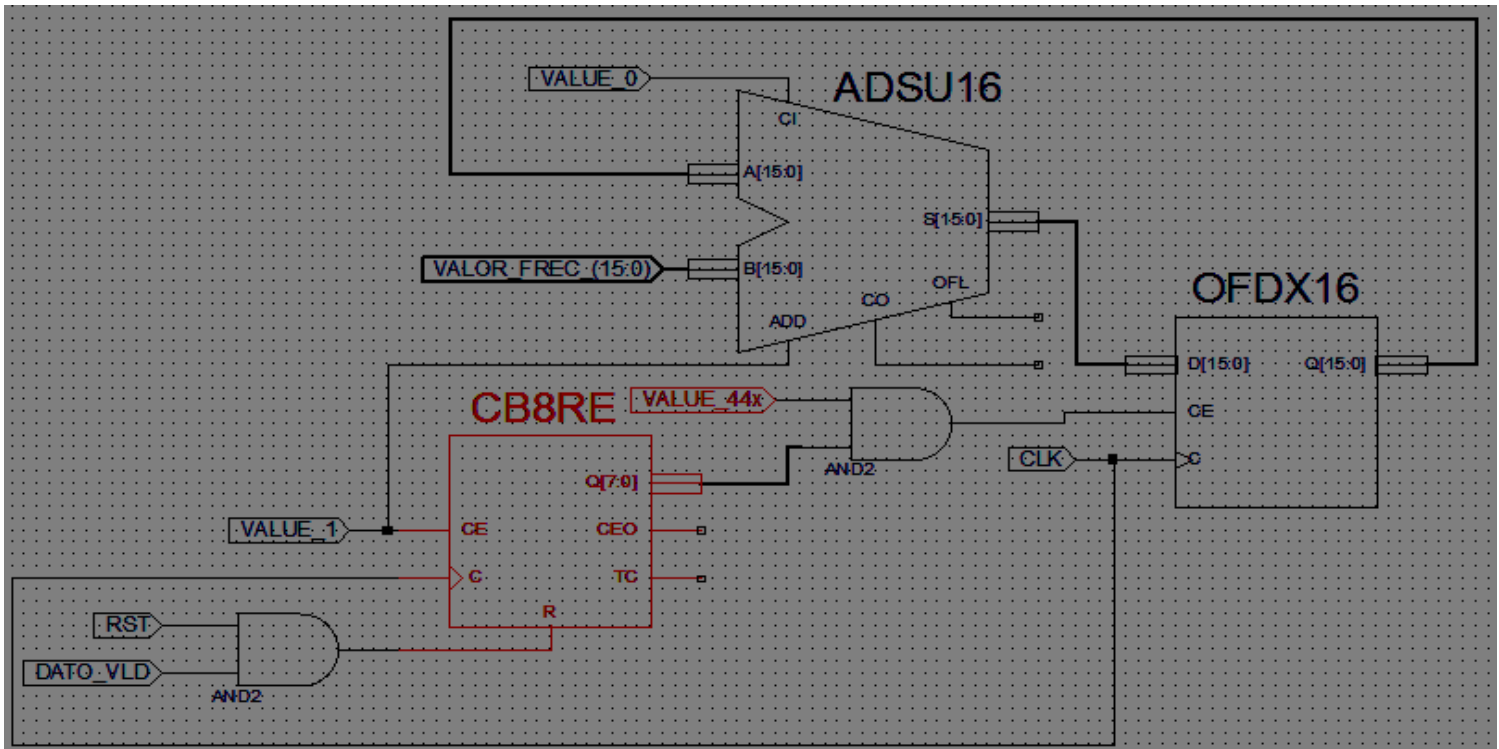
**Nombre:** Almacena Dir y Transfiere Dato

**Descripción:** Este apartado se basa en dos biestables tipo d, el primero de ellos almacena la dirección que se ha enviado cuando esta se hace efectiva mediante la señal dir\_vld, el segundo biestable es el encargado de pasar el dato que se almacena en la dirección que se ha transferido y validado antes pero nosotros solo queremos los datos de la dirección F0x y por ello solo sera posible la copia del dato cuando la dirección sea la adecuada.



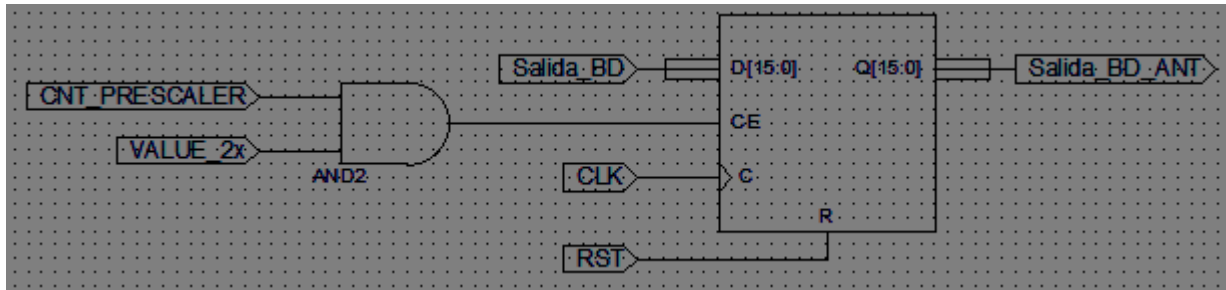
**Nombre:** Bloque principal

**Descripción:** La finalidad de este bloque es la de mandar la dirección donde ha de ser leído el dato pero con ciertas restricciones temporales para que su funcionamiento sea el correcto, para ello contamos con un prescaler el cual una vez validado el dato empezara a contar los 68 ciclos explicados al principio de este apartado, una vez llegado a la cuenta permite la copia del dato de un lado al otro del biestable d, este dato se compone de la suma de el valor\_ferc(8) que es la salida del apartado anterior y la suma de la salida del biestable(16), la utilización de este bloque viene explicada en la definición del conjunto de forma mas extensa y concreta, en la salida de este bloque solo nos quedaremos con los 8bits de mayor peso que son los que necesitamos para poder acceder a la memoria.



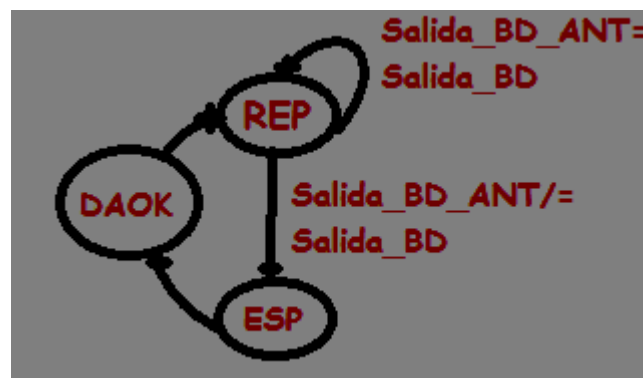
**Nombre:** Almacena dato anterior del biestable

**Descripción:** La finalidad de esta parte del circuito es la de almacenar la salida anterior del biestable principal para poder realizar la comparación de que el dato a sido cambiado pudiendo así desencadenar ciertos eventos en la maquina de estado, esta actualización del cambio del estado anterior de la salida solo se realiza cuando el contador alcanza la cuenta de 2 estado en el cual ya se ha realizado el envío del dato\_ok.



**Nombre:** MEF

**Descripción:** La función es la misma que en todas las maquinas de estado finitos en este caso el salto o cambio se produce cuando la salida del biestable principal cambia con respecto al estado anterior, en esta situación pasamos un ciclo de reloj en espera para poder cumplir las restricciones de tiempo y después mantenemos la señal del dato\_ok otro ciclo de reloj a nivel alto para que el resto de componentes que la usen tengan tiempo suficiente para aceptar el mensaje.



### *Código gen\_dir*

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

entity gen_dir is
port (
    CLK    : in std_logic;
    RST    : in std_logic;
    DIR    : in std_logic_vector (7 downto 0);
    DIR_VLD : in std_logic;
    DATO    : in std_logic_vector (7 downto 0);
    DATO_VLD : in std_logic;
    ADDR_OUT : out std_logic_vector(7 downto 0);
    DATO_OK  : out std_logic);
end gen_dir;

architecture rtl of gen_dir is
--Constante
constant dir_freq      : std_logic_vector(7 downto 0) := x"F0";
--Señales
signal Valor_freq      : std_logic_vector(7 downto 0);
signal Entrada_BD      : std_logic_vector(15 downto 0);
signal Salida_BD       : std_logic_vector(15 downto 0);
signal Salida_BD_ANT    : std_logic_vector(15 downto 0);
signal CEBD            : std_logic;
--Almacenado especial
signal DIRAUX          : std_logic_vector (7 downto 0);
--Contador
signal cnt             : unsigned (8 downto 0);
--Maquina de estados finitos
type MEF is (REP, ESP, DAOK);
signal std_act, prox_std : MEF;

begin

--Almacena la dir cada vez que sepamos que es valida SI DA LATCH PROBAR CON
--UN BIESTABLE D Y EL CE EL DIR_VLD
process (CLK, RST, DIR_VLD) is
begin -- process
    if RST = '1' then
        DIRAUX <= (others => '0');
    elsif CLK'event and CLK = '1'then
        if DIR_VLD = '1' then
            DIRAUX <= DIR;
        end if;
    end if;
end process;

-- purpose: Almacena el contenido del dato siempre que la direccion sea la correcta.
process (CLK, RST) is
```



```

begin -- process
  if RST = '1' then          -- asynchronous reset (active low)
    Valor_frec <= (others => '0');
  elsif CLK'event and CLK = '1' then -- rising clock edge
    if DIRAUX = dir_frec then
      Valor_frec <= DATO;
    end if;
  end if;
end process;

--Suma las dos partes
Entrada_BD <= std_logic_vector(unsigned(Valor_frec)+unsigned(Salida_BD));

-- purpose: Biestable D Principal con el cual aumentamos la dirección de salida de forma
constante:
process (CLK, RST) is
begin -- process
  if RST = '1' then          -- asynchronous reset (active low)
    Salida_BD <= (others => '0');
  elsif CLK'event and CLK = '1' then -- rising clock edge
    if CEBD = '1' then
      Salida_BD <= Entrada_BD;
    end if;
  end if;
end process;

--Solo seleccionamos lo que queremos para mandar la dirección de salida
ADDR_OUT <= Salida_BD(7 downto 0);

--PRESCALER Contador de 68
process (CLK, RST, DATO_VLD, DIR) is
begin -- process
  if RST = '1' then          -- asynchronous reset (active low)
    cnt <= (others => '0');
  elsif DATO_VLD = '1' or DIR/=dir_frec then
    cnt <= (others => '0');
  elsif CLK'event and CLK = '1' then -- rising clock edge
    if cnt = 68 then
      cnt <= (others => '0');
    else
      cnt <= cnt+1;
    end if;
  end if;
end process;

--Habilita la copia del dato cada 68 ciclos de reloj
CEBD <= '1' when cnt = 68 else '0';

--Parte 1 MEF
process (std_act, Salida_BD_ANT, Salida_BD) is
begin -- process
  case std_act is
    when REP =>

```

```

    if Salida_BD = Salida_BD_ANT then
        prox_std <= REP;
    else
        prox_std <= ESP;
    end if;
    when ESP => prox_std <= DAOK;
    when DAOK => prox_std <= REP;
end case;
end process;

--Parte 2 MEF
process (CLK, RST) is
begin -- process
    if RST = '1' then -- asynchronous reset (active low)
        std_act <= REP;
    elsif CLK'event and CLK = '1' then -- rising clock edge
        std_act <= prox_std;
    end if;
end process;

--Parte 3 MEF
DATO_OK <= '1' when std_act = DAOK else '0';

--Almacena la salida anterior
process (CLK, RST) is
begin -- process
    if RST = '1' then -- asynchronous reset (active low)
        Salida_BD_ANT <= (others => '0');
    elsif CLK'event and CLK = '1' then -- rising clock edge
        if cnt=2 then
            Salida_BD_ANT <= Salida_BD;
        end if;
    end if;
end process;

end rtl;

```

## TestBench

Después de ello realizamos el TestBench para verificar su correcto funcionamiento para ello creamos el archivo gen\_dir\_tb en el cual implementamos toda la lógica necesaria para poder simular el comportamiento del componente, para ello crearemos mediante un proceso la simulación de el envío de datos con ciertos tiempo los cuales deberían de ser los adecuados para así comprobar si funciona correctamente, también se prueba que no realiza ninguna acción si la dirección no es la correcta verificando así todo los casos posibles.

El código TestBench con todos los datos seria el citado continuación:

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

-----

entity gen_dir_tb is

end entity gen_dir_tb;

-----

architecture gen_dir of gen_dir_tb is

    -- component ports
    signal CLK      : std_logic          := '0';
    signal RST      : std_logic          := '1';
    signal DIR      : std_logic_vector (7 downto 0) := (others => '0');
    signal DIR_VLD  : std_logic          := '0';
    signal DATO     : std_logic_vector (7 downto 0) := (others => '0');
    signal DATO_VLD : std_logic          := '0';
    signal ADDR_OUT : std_logic_vector(7 downto 0);
    signal DATO_OK  : std_logic;

begin -- architecture gen_dir

    -- component instantiation
    DUT : entity work.gen_dir
    port map (
        CLK    => CLK,
        RST    => RST,
        DIR    => DIR,
        DIR_VLD => DIR_VLD,
        DATO   => DATO,
        DATO_VLD => DATO_VLD,
        ADDR_OUT => ADDR_OUT,
        DATO_OK => DATO_OK);

    -- clock generation
    CLK <= not CLK after 5 ns;
    RST <= '1', '0' after 30 ns;
```

process is

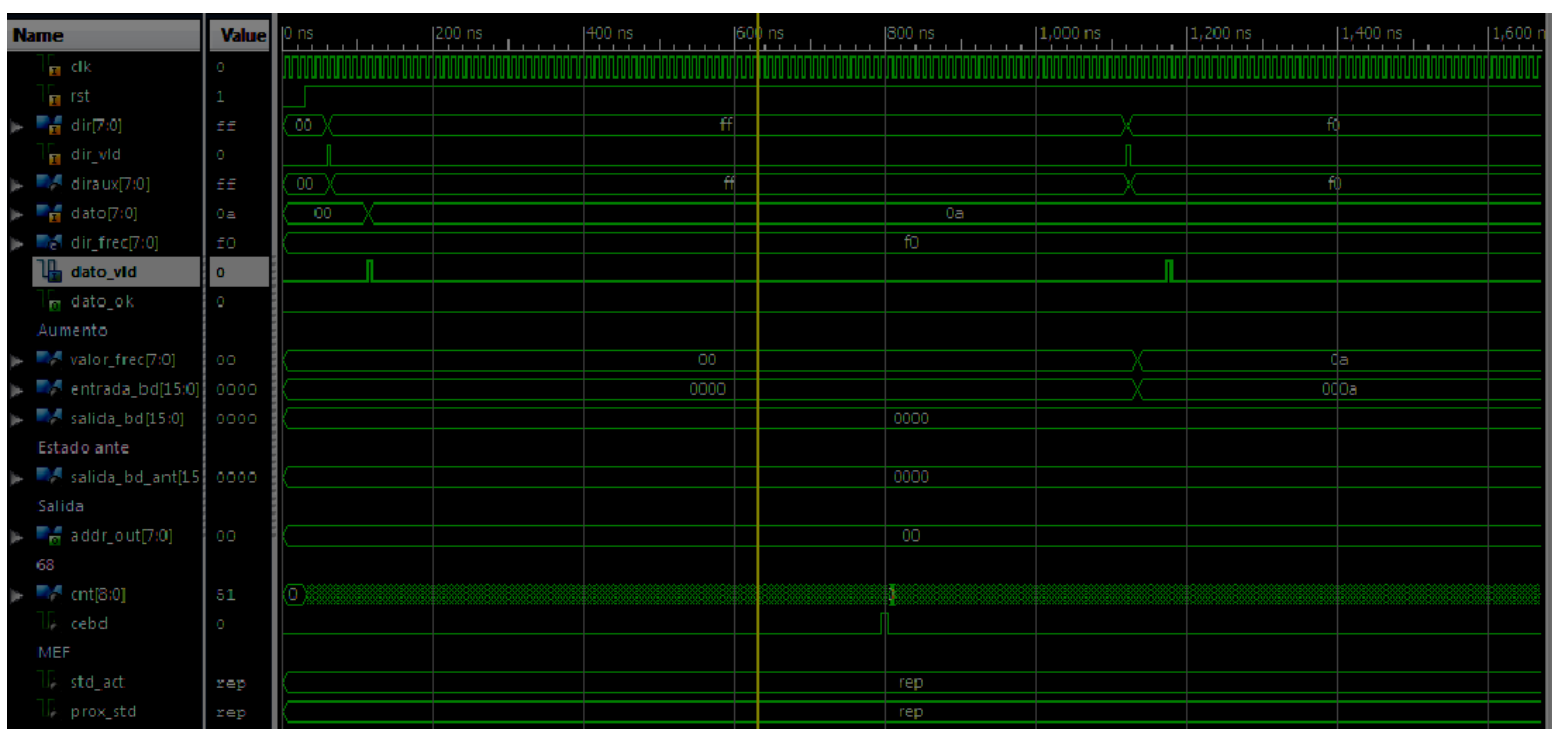
```
procedure dpram(Dvalue:in std_logic_vector(7 downto 0);
               Dat:in std_logic_vector(7 downto 0))is
begin
  DIR <= Dvalue;
  DIR_VLD <= '1';
  wait for 10 ns;
  DIR_VLD <= '0';
  wait for 50 ns;
  DATO <= Dat;
  DATO_VLD <= '1';
  wait for 10 ns;
  DATO_VLD <= '0';
end procedure;
```

```
begin
  wait for 80 ns;
  dpram(Dvalue => x"FF",Dat => x"0A");
  wait for 1000 ns;
  dpram(Dvalue => x"F0",Dat => x"0A");
  wait for 50000 ns;
  dpram(Dvalue => x"FF",Dat => x"0A");
  wait for 2000 ns;
  assert True report "FINAL TEST" severity note;
end process;
```

```
end architecture gen_dir;
```

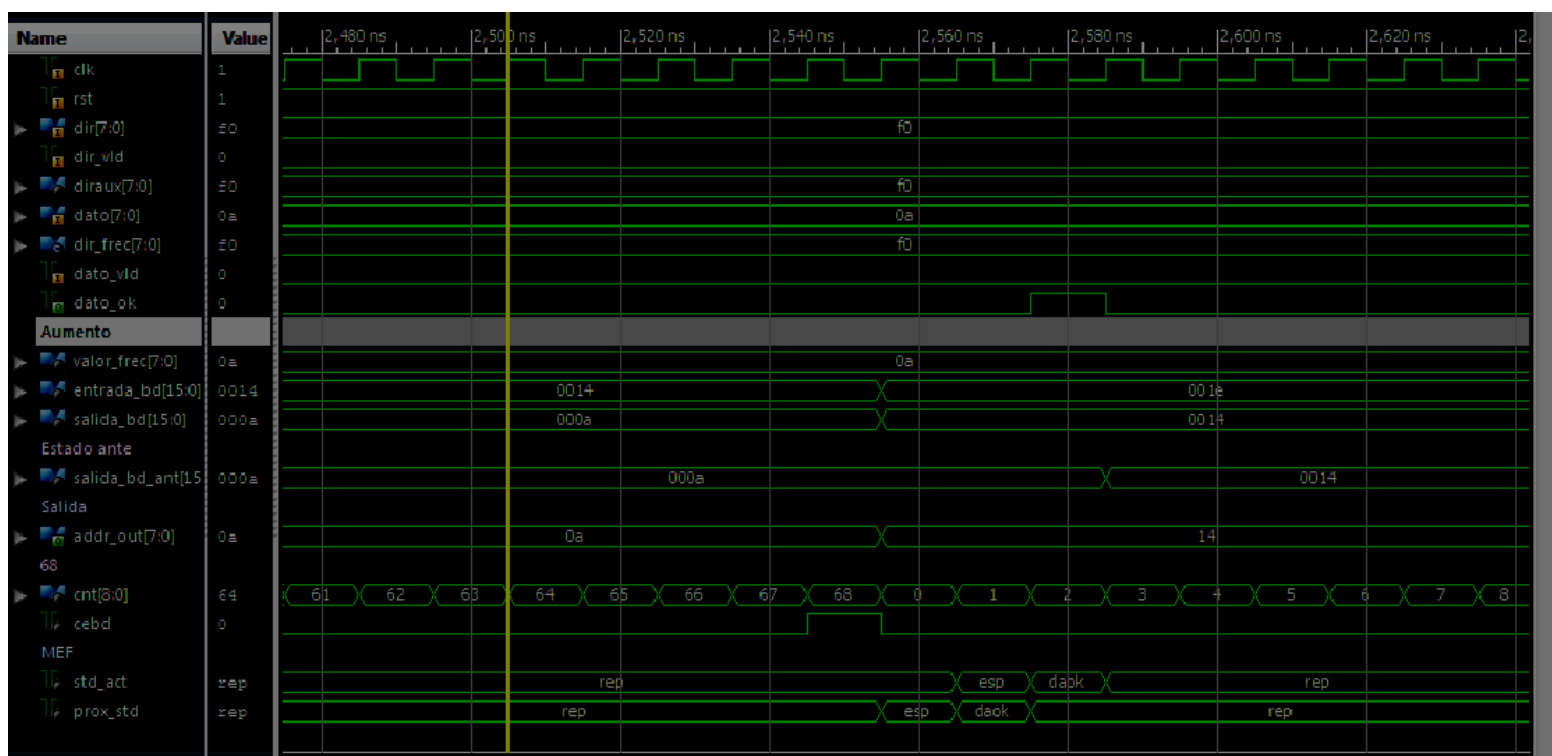
## Simulación Funcional

En la simulación funcional observamos el comportamiento del circuito mediante el estímulo de las señales generadas en el testbench.



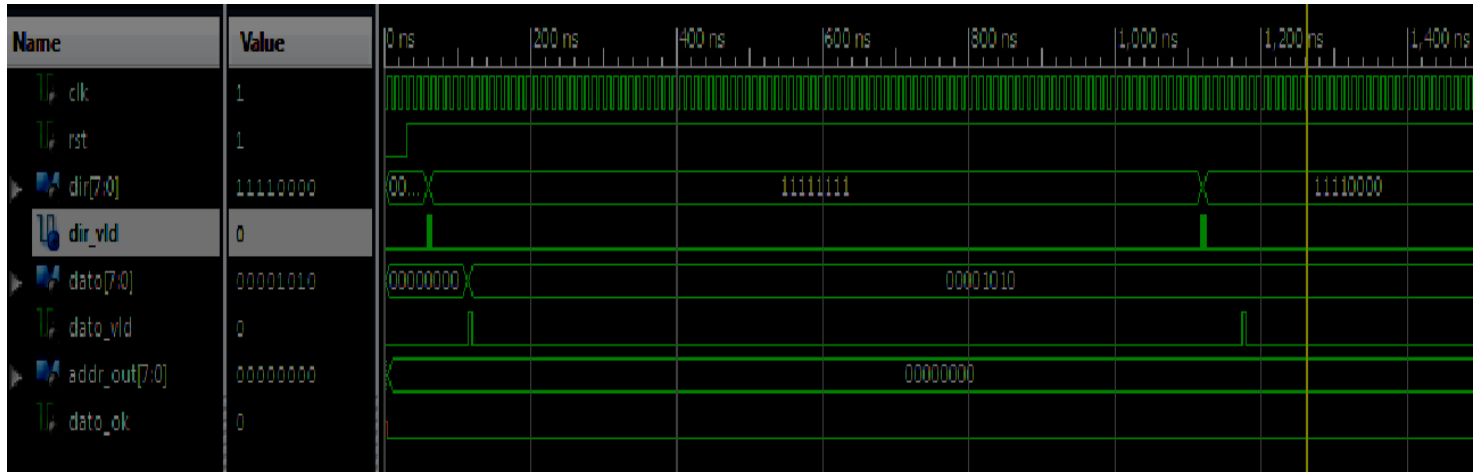
En este primer caso podemos observar como en ella situación de que la dirección no sea la adecuada entonces el sistema no hace ninguna acción.

Sin en cambio en la segunda parte cuando la dirección es correcta observamos que el sistema se comporta correctamente ademas de cumplir con las restricciones como no cambiar el valor de addres\_out hasta después de 68 ciclos y esperar 2 ciclos para aprobar la señal de dato\_ok.



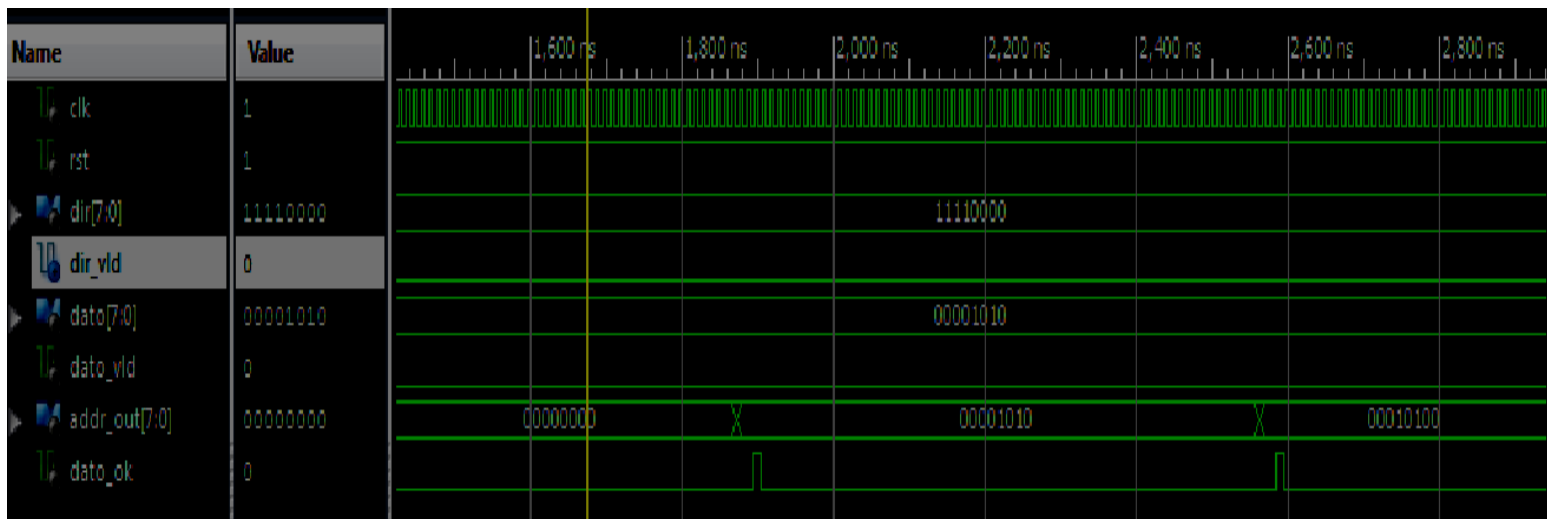
## Simulación Temporal

Una vez comprobado que la simulación funcional funciona correctamente y cumple con los requisitos deseados, en esta simulación realizaremos los pasos necesarios para hacer el place&route al circuito (pasos descritos al principio de la memoria) pudiendo así realizar dicha simulación en la cual se va a tener en cuenta como sería el circuito en la placa y los retardos de los que constaría.



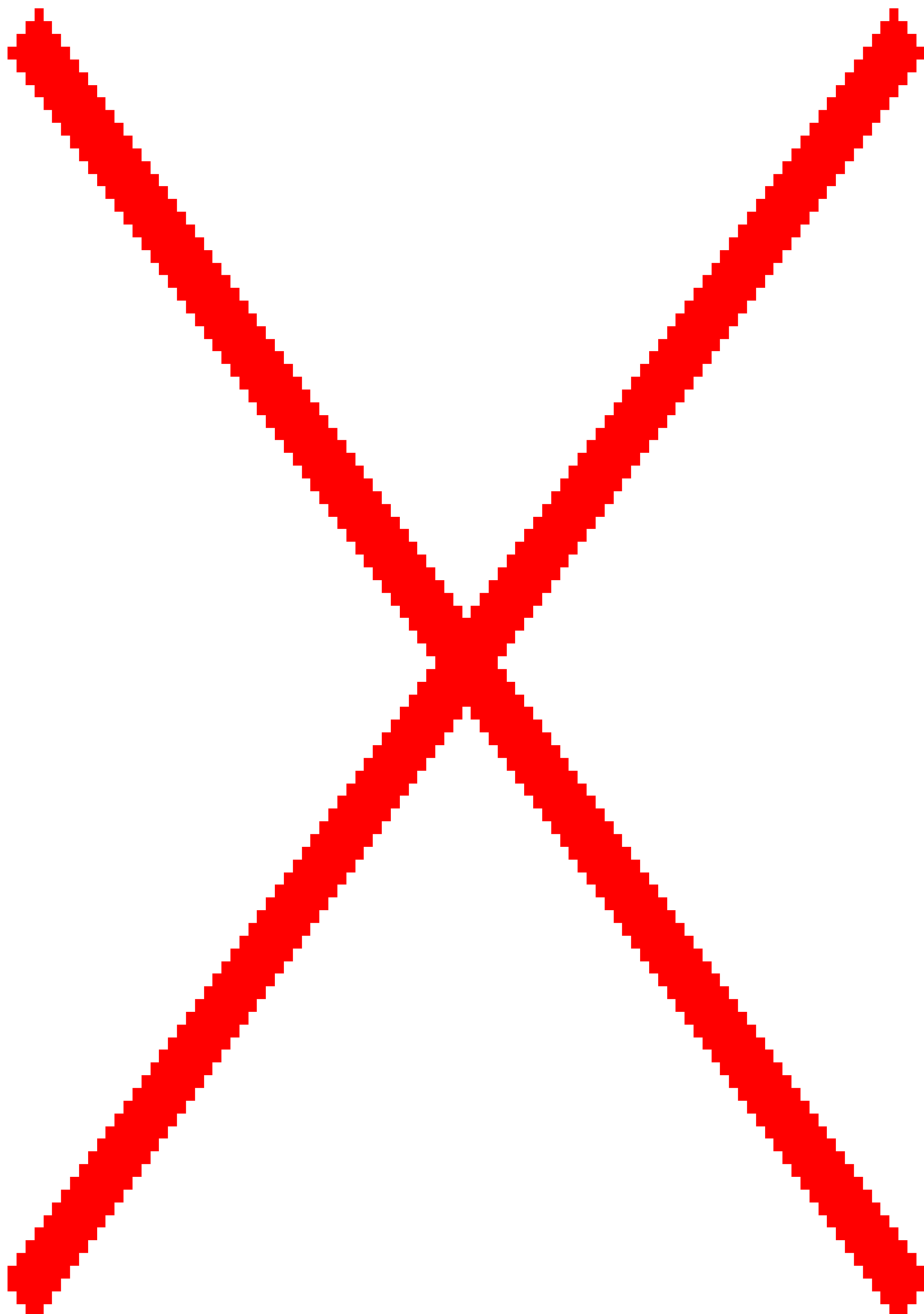
De la misma manera que en la funcional si la dirección no es la adecuada no realiza ninguna acción el componente.

En la segunda captura podemos observar (aunque no del todo bien ya que no entraría toda la simulación en la pantalla) que si la dirección es la adecuada se comporta correctamente y antes de realizar el cambio de la dirección espera los 68 ciclos predefinidos y después de ello espera dos ciclos para enviar la señal de dato\_ok.



Con esta información podemos concretar que este apartado cumple su función y esta listo para probarlo en el conjunto total.

## Recursos Utilizados



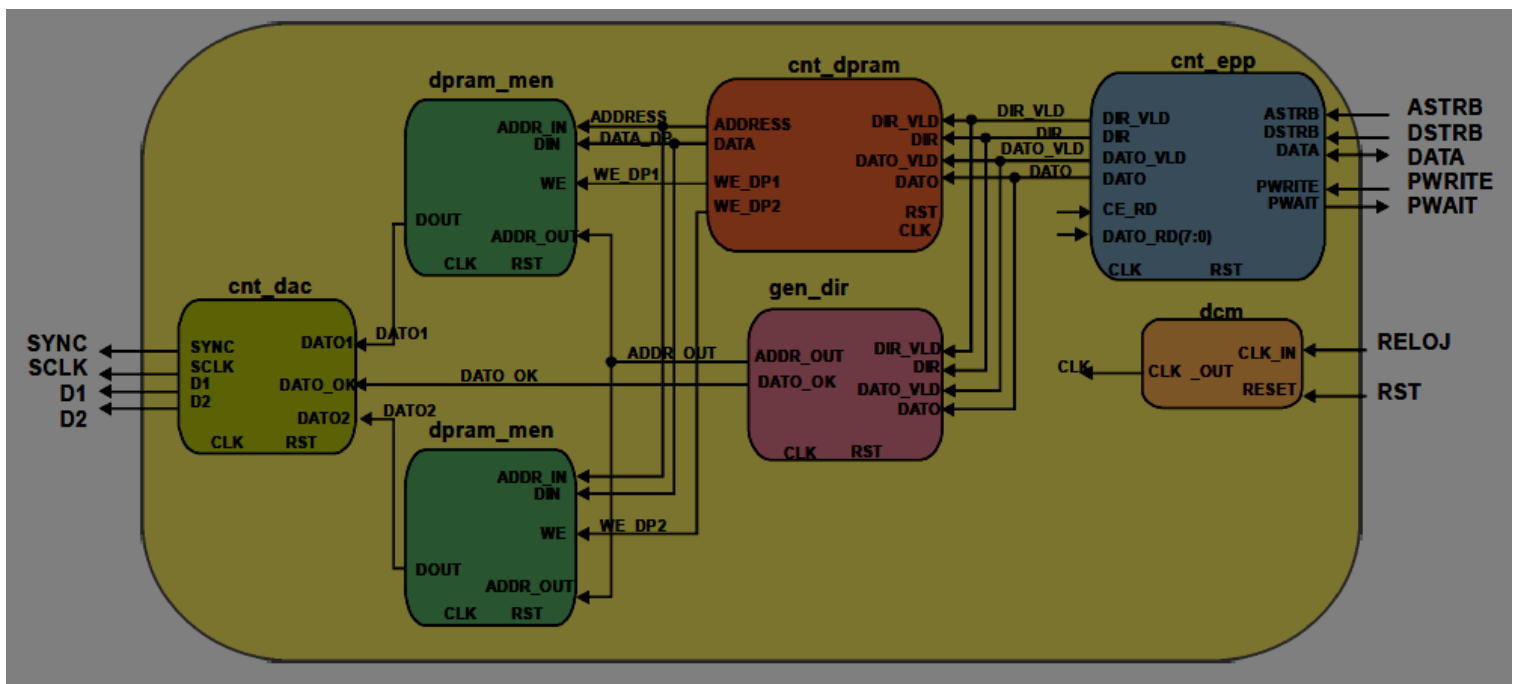
## Memoria GEN FUNCIONES

Esta entidad es la base de la practica y es donde se montan todos los componentes que se han desarrollado por separado para que el resultado final sea el deseado satisfaciendo así el objetivo de la practica.

Las entradas y salidas de este apartado son las citadas a continuación además se especifica el tipo de dato que son y si son de entrada o de salida:

```
RELOJ : in std_logic;  
RST : in std_logic;  
-- PUERTO EPP  
ASTRB : in std_logic;  
DSTRB : in std_logic;  
DATA : inout std_logic_vector (7 downto 0);  
PWRITE : in std_logic;  
PWAIT : in std_logic;  
-- DAC  
SYNC : out std_logic;  
SCLK : out std_logic;  
D1 : out std_logic;  
D2 : out std_logic;
```

Las interconexiones dentro del programa se realizaran respetando los nombre y cumpliendo los esquemas del dibujo.



El esquema viene ya implementado en el archivo gen\_funciones suministrado para la realización de la practica. Por lo que solo sera necesario añadir los distintos módulos al programa.




## Generación DCM

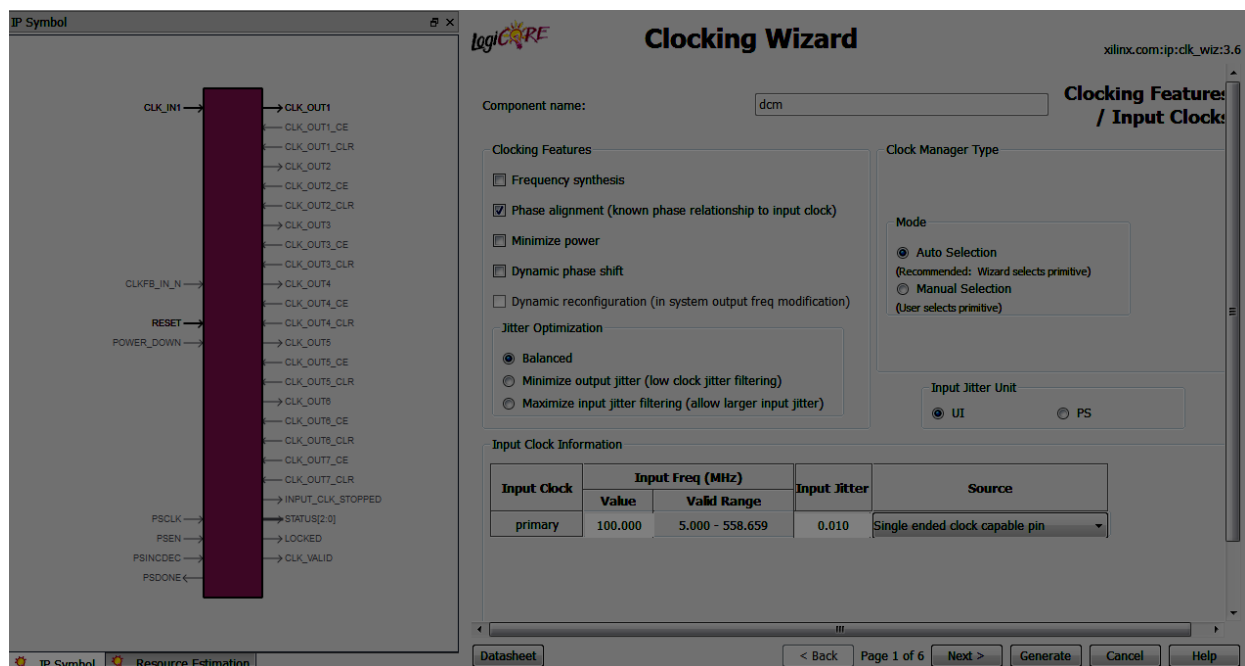
Para este apartado crearemos un modulo especial el cual se va encargar de distribuir la señal de reloj a todos los componentes internos apartir de la señal de reloj de la placa, el porque de esta medida es para tener todos los componentes con la misma señal de reloj y así evitar problemas de sincronización entre esto, por ello la mejor solución es esta, en caso de ser deseado se puede reducir la frecuencia de manera sencilla y rápida, pero no es nuestro caso.

Las entradas y salidas de este apartado son las citadas a continuación ademas se especifica el tipo de dato que son y si son de entrada o de salida

```
-- Clock in ports
CLK_IN : in std_logic;
-- Clock out ports
CLK_OUT : out std_logic;
-- Status and control signals
RESET : in std_logic
```

La creación de este modulo se realiza de manera sencilla con el propio programa para ello solo hemos de seguir unos pasos necesarios y se creara automáticamente el modulo.

- Dentro del programa hemos de crear un archivo nuevo con el botón 
- Dentro del apartado de creación seleccionamos IP y le damos el nombre DCM
- Una vez dentro del gestor seleccionamos FPGA y Clocking wizar
- Cuando aparezca el asistente Clocking wizar solo hemos de seguir los pasos que nos dicta el programa con las opciones que creamos convenientes para nosotros.

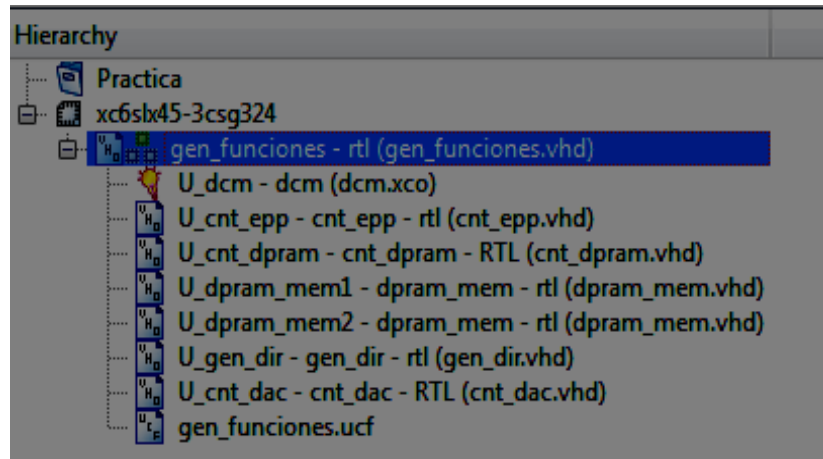


- Después de generar el DCM solo hemos de añadirlo al código del programa y ya estará listo

Con estos pasos obtendremos el archivo DCM para gestionar la señal de reloj de los componentes teniendo así ya todos los componentes.

## Agregación Archivos y Device 1

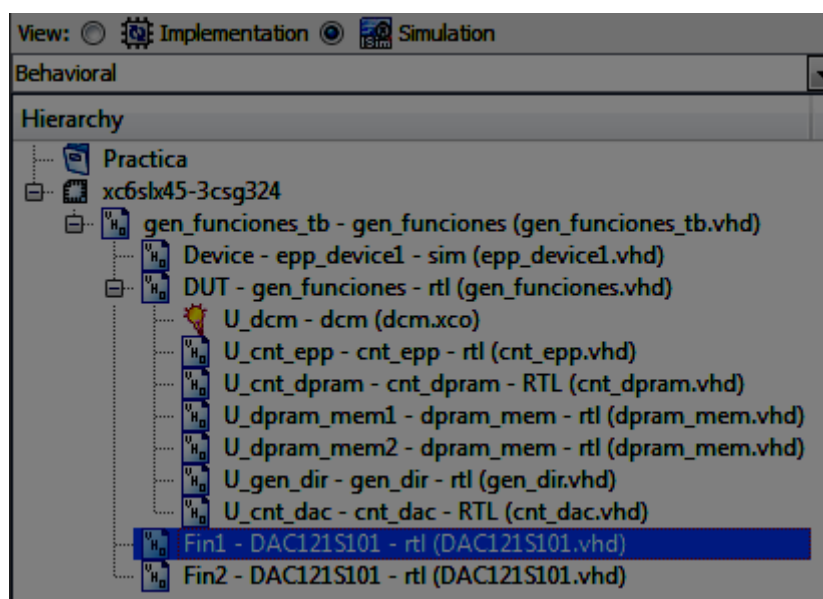
Una vez que tengamos todos los componentes solo hemos de añadirlos al programa para ello cargaremos el archivo gen\_funciones el cual al estar ya implementado nos ira solicitando todos los componentes logrando así de manera sencilla saber cuales son necesarios agregar, una vez que estén todos los archivos agregados al programa incluyendo el DCM debemos tener algo similar a lo explicado en la imagen.



También hemos de añadir el archivo .ucf el cual define los puestos o salidas que se utilizan en la placa de simulación en el laboratorio.

Una vez completado este apartado hemos de crear el correspondiente TestBench para la practica para ello se han de añadir todos los componentes de igual manera que en el otro caso pero en este caso el archivo no esta definido por lo que antes de nada se ha de crear la definición de los puestos y su utilización en este caso hemos de utilizar el componente device1 para generar datos para la pruebas y el componente DAC121S101 para las salidas. La definición de como queda el código final esta en el apartado que contiene el código del TestBench.

El resultado de este apartado ha de quedar de manera similar que en la imagen.



### Codigo Gen Funciones

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

entity gen_funciones is
  port (
    RELOJ : in std_logic;
    RST   : in std_logic;
    -- PUERTO EPP
    ASTRB : in std_logic;
    DSTRB : in std_logic;
    DATA : inout std_logic_vector(7 downto 0);
    PWRITE : in std_logic;
    PWAIT  : out std_logic;
    -- DAC
    SYNC   : out std_logic;
    SCLK   : out std_logic;
    D1     : out std_logic;
    D2     : out std_logic);
end gen_funciones;

architecture rtl of gen_funciones is

  signal CLK : std_logic;
  --SEÑALES DEL CONTROLADOR EPP--
  signal DIR   : std_logic_vector (7 downto 0);
  signal DIR_VLD : std_logic;
  signal DATO   : std_logic_vector (7 downto 0);
  signal DATO_VLD : std_logic;
  signal DATO_RD : std_logic_vector (7 downto 0);
  signal CE_RD : std_logic;
  --SEÑALES DEL CONTROLADOR DPRAM--
  signal ADDRESS : std_logic_vector(7 downto 0);
  signal DATA_DP : std_logic_vector (7 downto 0);
  signal WE_DP1 : std_logic;
  signal WE_DP2 : std_logic;
  --SEÑALES DE LAS MEMORIAS DPRAM--
  signal DATO1 : std_logic_vector (7 downto 0);
  signal DATO2 : std_logic_vector (7 downto 0);
  --SEÑALES DE GENERADOR DE _DIRECCIONES--
  signal ADDR_OUT : std_logic_vector(7 downto 0);
  signal DATO_OK : std_logic;
begin -- rtl

U_dcm :entity work.dcm
  port map
    (CLK_IN => RELOJ,
     CLK_OUT => CLK,
     RESET => RST);
```

U\_cnt\_epp:entity work.cnt\_epp

```
port map (  
    CLK    => CLK,  
    RST    => RST,  
    ASTRB  => ASTRB,  
    DSTRB  => DSTRB,  
    DATA  => DATA,  
    PWRITE => PWRITE,  
    PWAIT  => PWAIT,  
    DATO_RD => DATO_RD,  
    CE_RD  => CE_RD,  
    DIR    => DIR,  
    DIR_VLD => DIR_VLD,  
    DATO   => DATO,  
    DATO_VLD => DATO_VLD);
```

U\_cnt\_dpram:entity work.cnt\_dpram

```
port map (  
    CLK    => CLK,  
    RST    => RST,  
    DIR    => DIR,  
    DIR_VLD => DIR_VLD,  
    DATO   => DATO,  
    DATO_VLD => DATO_VLD,  
    ADDRESS => ADDRESS,  
    DATA  => DATA_DP,  
    WE_DP1 => WE_DP1,  
    WE_DP2 => WE_DP2);
```

U\_dpram\_mem1 : entity work.dpram\_mem

```
port map (  
    DIN    => DATA_DP,  
    ADDR_IN => ADDRESS,  
    WE     => WE_DP1,  
    CLK    => CLK,  
    RST    => RST,  
    ADDR_OUT => ADDR_OUT,  
    DOUT   => DATO1);
```

U\_dpram\_mem2 : entity work.dpram\_mem

```
port map (  
    DIN    => DATA_DP,  
    ADDR_IN => ADDRESS,  
    WE     => WE_DP2,  
    CLK    => CLK,  
    RST    => RST,  
    ADDR_OUT => ADDR_OUT,  
    DOUT   => DATO2);
```

U\_gen\_dir:entity work.gen\_dir

```
port map (  

```

```
CLK    => CLK,  
RST    => RST,  
DIR     => DIR,  
DIR_VLD => DIR_VLD,  
DATO    => DATO,  
DATO_VLD => DATO_VLD,  
ADDR_OUT => ADDR_OUT,  
DATO_OK => DATO_OK);
```

```
U_cnt_dac :entity work.cnt_dac
```

```
port map (  
  CLK    => CLK,  
  RST    => RST,  
  DATO1   => DATO1,  
  DATO2   => DATO2,  
  DATO_OK => DATO_OK,  
  SYNC    => SYNC,  
  SCLK    => SCLK,  
  D1      => D1,  
  D2      => D2);
```

```
end rtl;
```

## **Codigo Gen funciones TB**

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

```
-----
entity gen_dir_tb is
```

```
end entity gen_dir_tb;
-----
```

```
architecture gen_dir of gen_dir_tb is
```

```
-- component ports
signal CLK      : std_logic          := '0';
signal RST      : std_logic          := '1';
signal DIR      : std_logic_vector(7 downto 0) := (others => '0');
signal DIR_VLD  : std_logic          := '0';
signal DATO     : std_logic_vector(7 downto 0) := (others => '0');
signal DATO_VLD : std_logic          := '0';
signal ADDR_OUT : std_logic_vector(7 downto 0);
signal DATO_OK  : std_logic;
```

```
begin -- architecture gen_dir
```

```
-- component instantiation
DUT : entity work.gen_dir
port map (
    CLK    => CLK,
    RST    => RST,
    DIR    => DIR,
    DIR_VLD => DIR_VLD,
    DATO   => DATO,
    DATO_VLD => DATO_VLD,
    ADDR_OUT => ADDR_OUT,
    DATO_OK => DATO_OK);
```

```
-- clock generation
CLK <= not CLK after 5 ns;
RST <= '1', '0' after 30 ns;
```

```
process is
```

```
procedure dpram(Dvalue:in std_logic_vector(7 downto 0);
               Dat:in std_logic_vector(7 downto 0))is
begin
    DIR <= Dvalue;
    DIR_VLD <= '1';
    wait for 10 ns;
```

```

    DIR_VLD <= '0';
    wait for 50 ns;
    DATO <= Dat;
    DATO_VLD <= '1';
    wait for 10 ns;
    DATO_VLD <= '0';
end procedure;

begin
    wait for 80 ns;
    dpram(Dvalue => x"FF",Dat => x"0A");
        wait for 1000 ns;
    dpram(Dvalue => x"F0",Dat => x"0A");
        wait for 50000 ns;
    dpram(Dvalue => x"FF",Dat => x"0A");
        wait for 2000 ns;
    assert True report "FINAL TEST" severity note;
end process;

end architecture gen_dir;

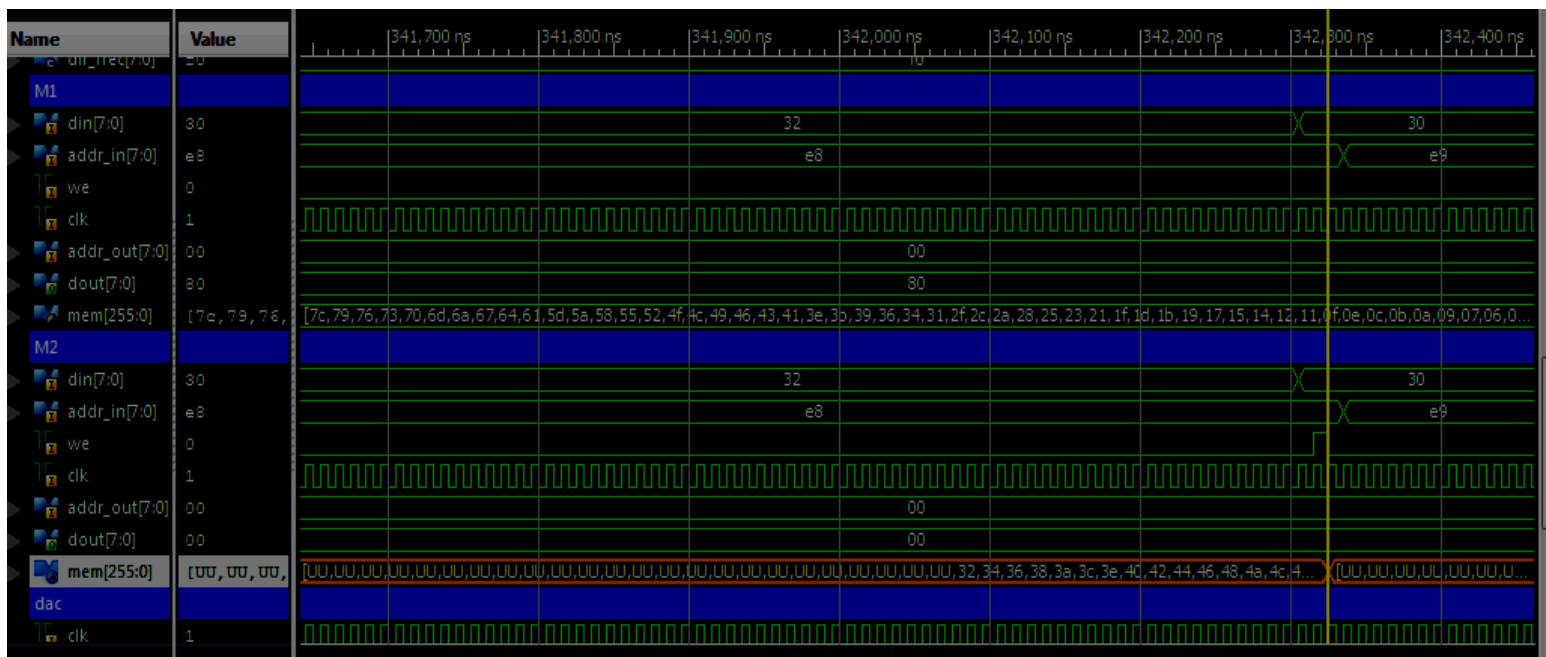
```

## Simulación Funcional

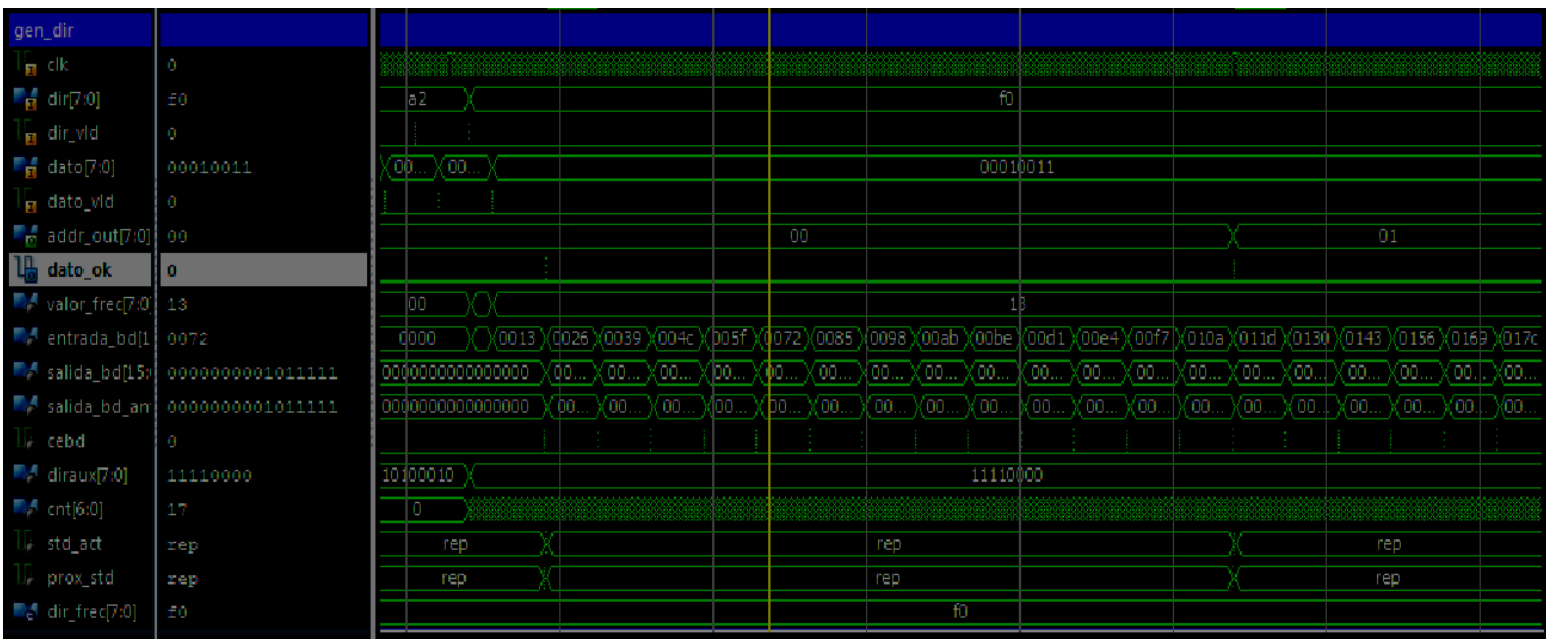
En la simulación funcional observamos el comportamiento del circuito mediante el estímulo de las señales generadas en el testbench. En este caso las señales son suministradas por el modulo Device 1 el cual obtiene los datos desde unos archivos ubicados en la carpeta del programa estos archivos son VO1.dat y VO2.dat

Como el resultado es demasiado grande para mostrar todas las señales a mostrar solo se enseñan las imágenes mas relevantes. Esta simulación tiene tres 2 apartados uno en el cual realiza la carga de los datos en la memoria y una vez transcurrido un tiempo empieza a sacar estos datos con una cierta frecuencia y a mandarlos a las salidas.

- Inserción de los datos en la memoria:

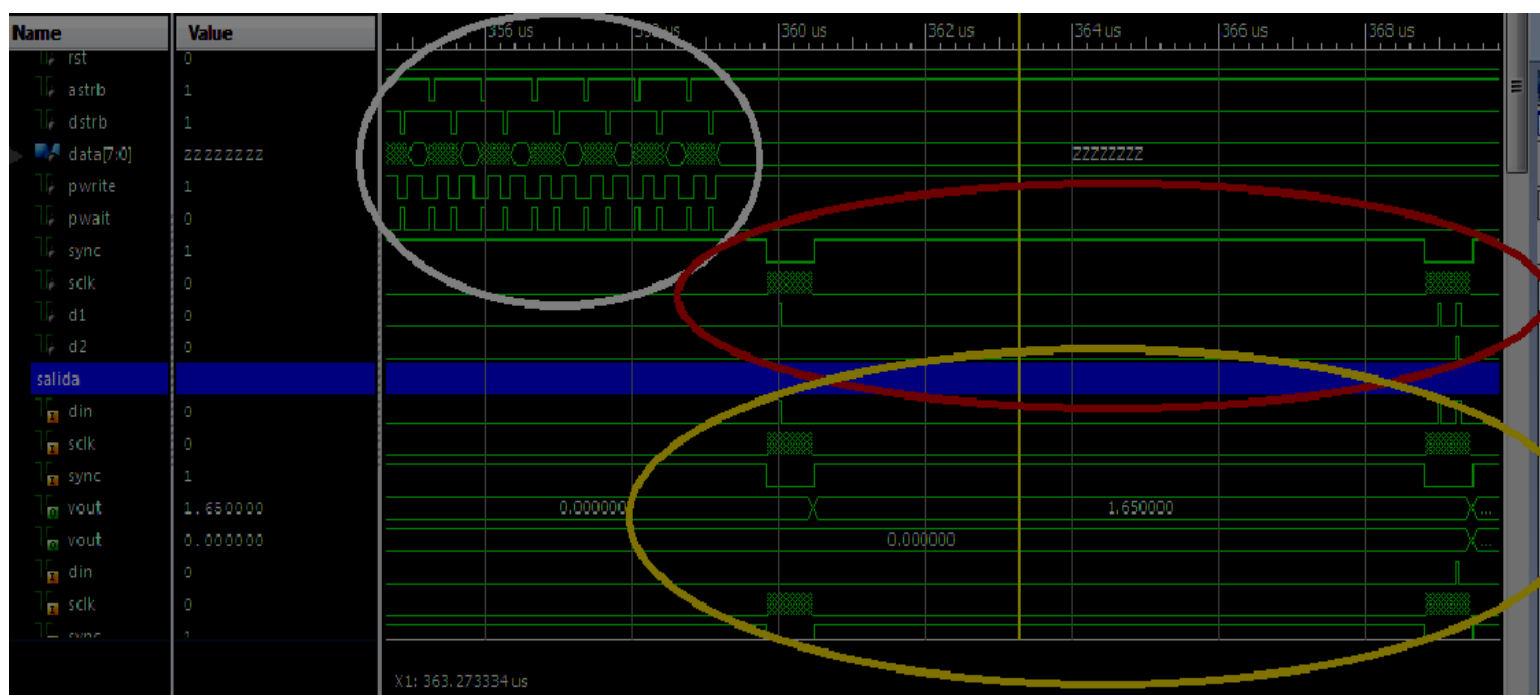


- Generación de los datos a sacar y con que frecuencia, en este caso la frecuencia inicial es 13 pero luego cambia, también se puede observar como se hace la generación del dato\_ok





- En la ultima imagen podemos observar como se digitalizan los datos obtenidos desde las memorias aparir de la dirección que es generada por gen\_dir.



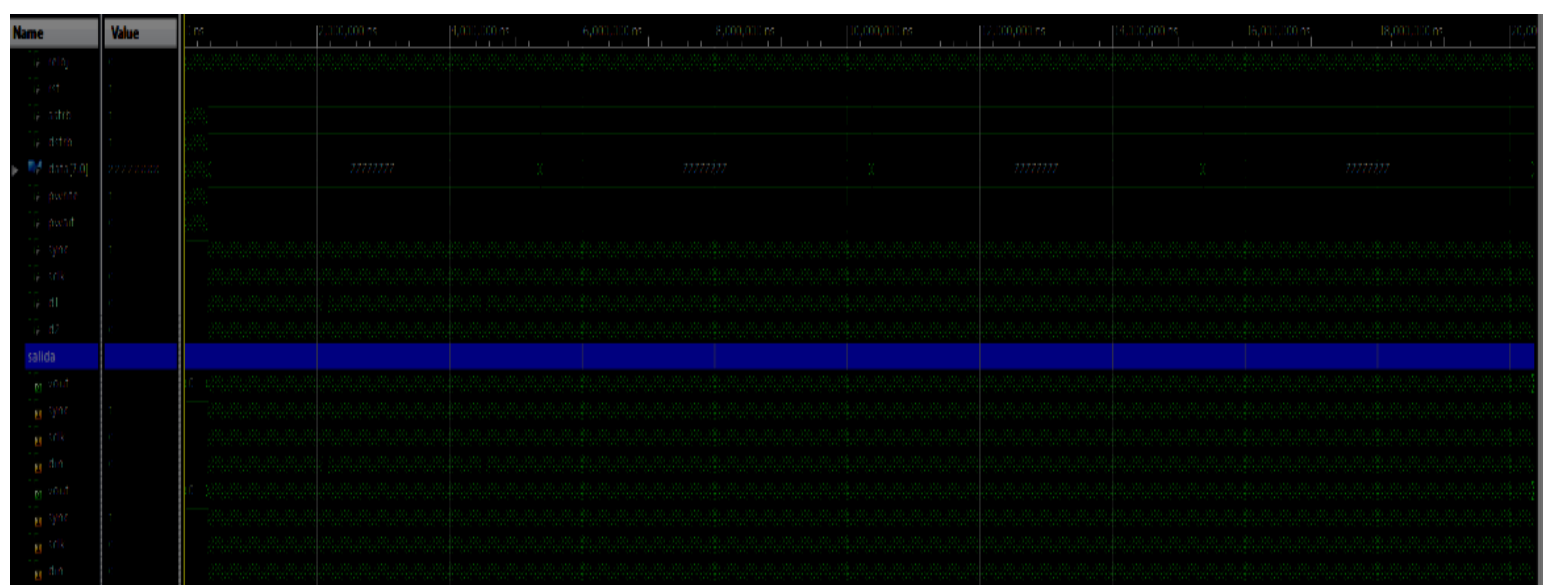
- El primer dato que obtenemos es para D1 80 y para D2 00 lo cual coincide con los datos de los archivos de los cuales se han sacado los datos para almacenarlos en la memoria. **Circulo ROJO**
- También se puede apreciar como la información es aceptada correctamente por los conversores deduciendo que las restricciones de tiempo son las correctas **Circulo Blanco**
- **Circulo Blanco** primera parte donde se realiza la inserción de los datos.

Con estos resultado podemos suponer que el funcionamiento del programa va por el camino correcto.

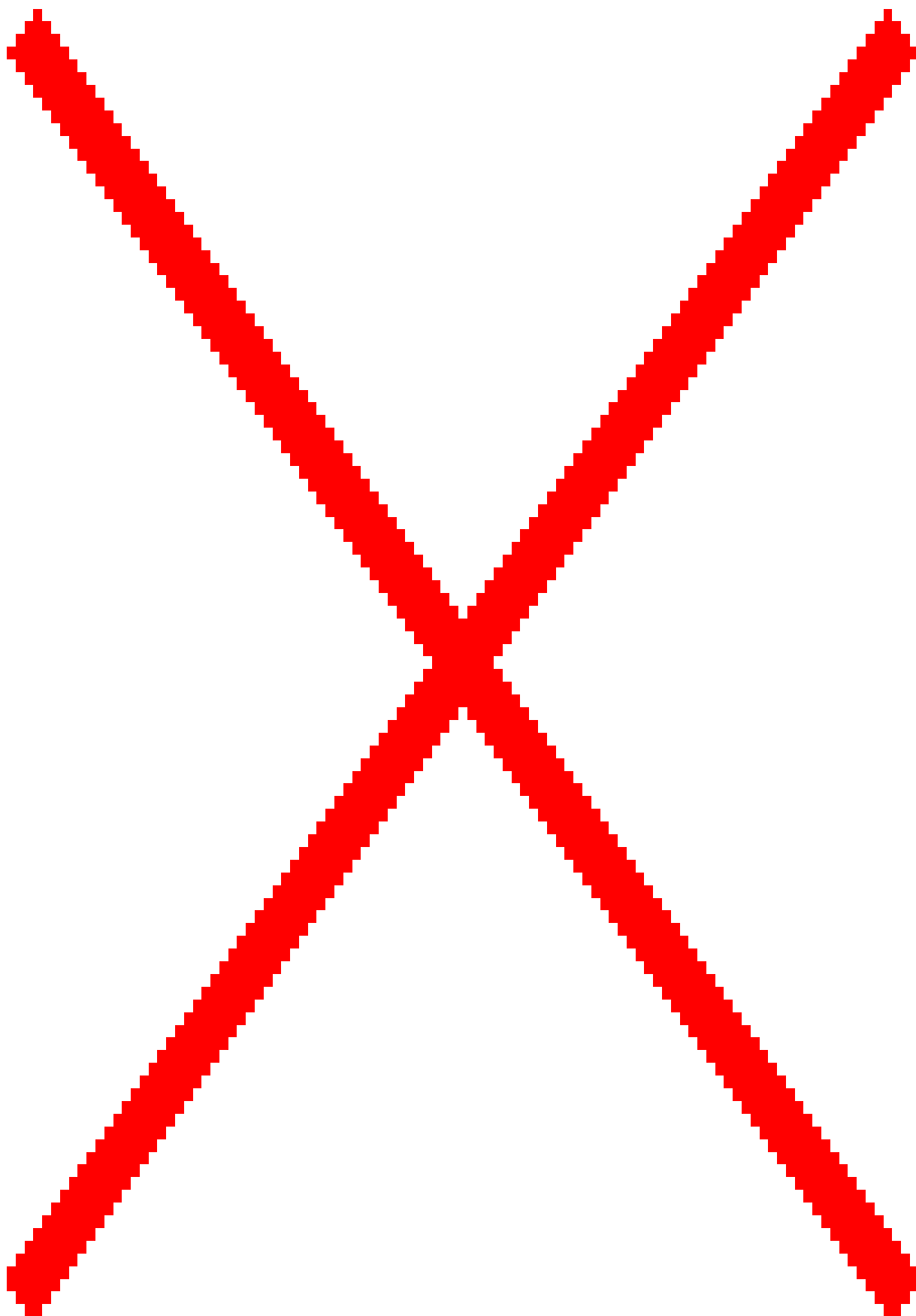
## Simulación Temporales

En el dibujo se observa como el primer dato de D1 es 80 y el de D2 cambiando en el siguiente paso a 83 y 02 es 00 y ademas también se puede observar como se digitalizan los datos en los DAC121S101 pasándolos de digital a un dato de voltaje(Amarillo).

En el dibujo se observa como el primer dato de D1 es 80 y el de D2 cambiando en el siguiente paso a 83 y 02 es 00 y ademas también se puede observar como se digitalizan los datos en los DAC121S101 pasándolos de digital a un dato de voltaje(Amarillo).



## 90



Generación del archivo a descargar en placa

Una vez comprobado que la simulación temporal es correcta pasamos a añadir al programa el archivo gen\_funciones.ucf en el cual se especifican cuales son los puertos que se usan y a que salida o entrada corresponden en la placa. Quedando igual que en la imagen.

Después generamos el archivo.bit el cual sera el descargado en la placa para su posterior testeo final

```
NET "RELOJ" LOC = "L15";
```

```
# onBoard USB controller
```

```
NET "ASTRB" LOC = "B9";
```

```
NET "DSTRB" LOC = "A9";
```

```
NET "PWRITE" LOC = "C15";
```

```
NET "PWAIT" LOC = "F13";
```

```
NET "DATA<0>" LOC = "A2";
```

```
NET "DATA<1>" LOC = "D6";
```

```
NET "DATA<2>" LOC = "C6";
```

```
NET "DATA<3>" LOC = "B3";
```

```
NET "DATA<4>" LOC = "A3";
```

```
NET "DATA<5>" LOC = "B4";
```

```
NET "DATA<6>" LOC = "A4";
```

```
NET "DATA<7>" LOC = "C5";
```

```
# onBoard Pushbuttons
```

```
NET "RST" LOC = "N4";
```

```
# DAC
```

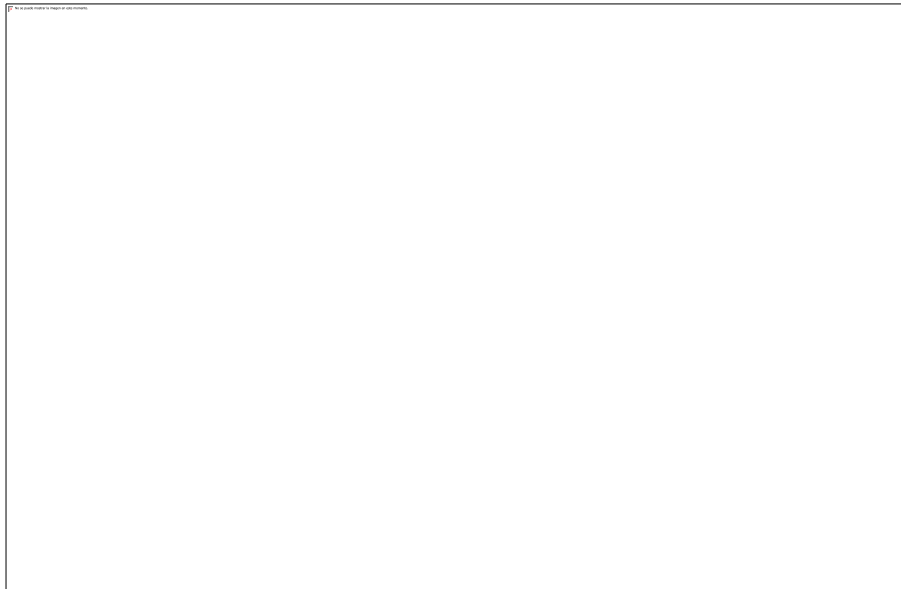
```
NET "SYNC" LOC = "T3" ;
```

```
NET "D1" LOC = "R3" ;
```

```
NET "D2" LOC = "P6" ;
```

```
NET "SCLK" LOC = "N5" ;
```

Después de generar el archivo \*.bit hemos de realizar la ultima prueba en la placa de clase, esta prueba se realizara con ayuda de un programa que suministra el profesor para matlab con el cual se irán generado datos (El valor que se da a la frecuencia en el programa mostrara sera el equivalente a Valor\_freq que se usa en gen\_dir) y mandándolos por el puerto a la placa (Rojo) con sus correspondientes señales para que la codificación sea la correcta, una vez que esos datos están almacenados la placa se encargara ella solo de mandarlos a las salidas donde se incluye un digitalizador 2 DACs (verde)el cual se encarga de convertir la información que le llega a voltaje para que pueda ser recogida por el osciloscopio donde se mostrara la gráfica correspondiente a los datos introducidos



Cuando generamos un valor\_freq con el programa de 40 entonces mediante la formula citada abajo obtenemos que la frecuencia con la que sera mostrada sera aproximada a Valor\_freq=897.575Hz

$$f \approx \frac{VALOR\_FREC}{N * 2^{16}} * f_{CLK}$$

