Universidad Tecnológica de Panamá

Facultad de Ingeniería Eléctrica

Laboratorio de Microprocesadores

Laboratorio #6

Fernando Guiraud
8-945-692

Profesor Elias Mendoza

Grupo: 4EE141

Semestre I 2022

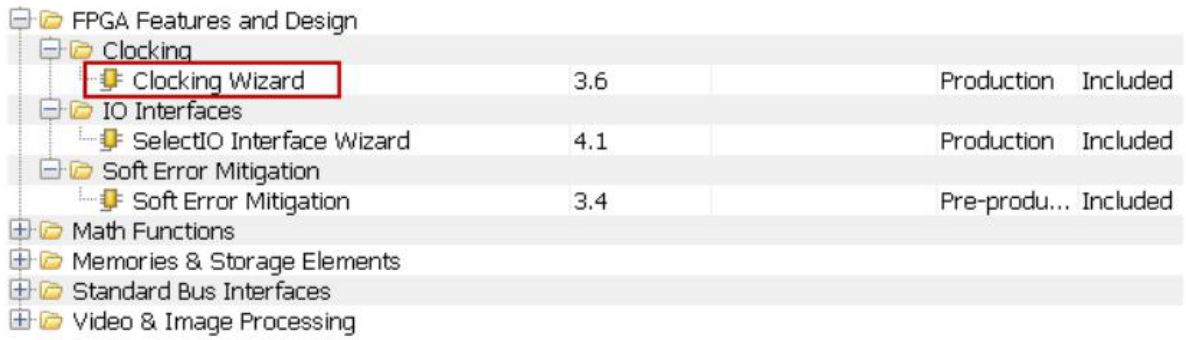
I. INTRODUCCIÓN

Los FPGA de Xilinx de hoy contienen muchos más recursos que los básicos, LUT, CLB, IOB y enrutamiento. Los FPGA ahora se están utilizando para implementar circuitos digitales mucho más complejos en comparación con la lógica de cola cuando se inventaron. Algunos recursos arquitectónicos complejos, como la sincronización, deben configurarse e instanciarse en lugar de inferirse. También hay circuitos complejos de uso común, como el decodificador ReedSolomon y las herramientas para que un diseñador no tenga que "reinventar la rueda" y desarrollar las funciones básicas por su cuenta. Este laboratorio presenta el asistente de arquitectura y las herramientas del generador CORE disponibles a través del catálogo de IP. Consulte el tutorial de PlanAhead sobre cómo utilizar la herramienta PlanAhead para crear proyectos y verificar circuitos digitales.

- **Asistente arquitectónico:**

Algunos recursos arquitectónicos especializados y avanzados se pueden utilizar de manera eficiente cuando se configuran y se crean instancias correctamente en lugar de inferirlos. Dependiendo de la familia de FPGA que se esté utilizando, la cantidad y los tipos de dichos recursos varían. En la familia Spartan-6LX, el asistente de arquitectura admite recursos de temporización, SelectIO y mitigación de errores suaves (SEM). En la familia Spartan-6LX, hay disponible un recurso de arquitectura adicional, GTP Transceiver. Se accede a estos recursos bajo la capacidad del Catálogo IP de la herramienta PlanAhead.

En la placa Atlys, hay disponible una fuente de reloj de 100 MHz que está conectada al pin L15 del FPGA. Esta fuente de reloj se puede utilizar para generar una serie de relojes de diferentes frecuencias y cambios de fase. Esto se hace mediante el uso de recursos arquitectónicos llamados Digital Clock Manager (DCM) y Phase Locked Loop (PLL) de la familia FPGA Spartan-6LX. El generador de la fuente de sincronización se puede invocar haciendo doble clic en la entrada del Asistente de sincronización en la subcarpeta de sincronización de la Carpeta de características y diseño de FPGA del Catálogo de IP.

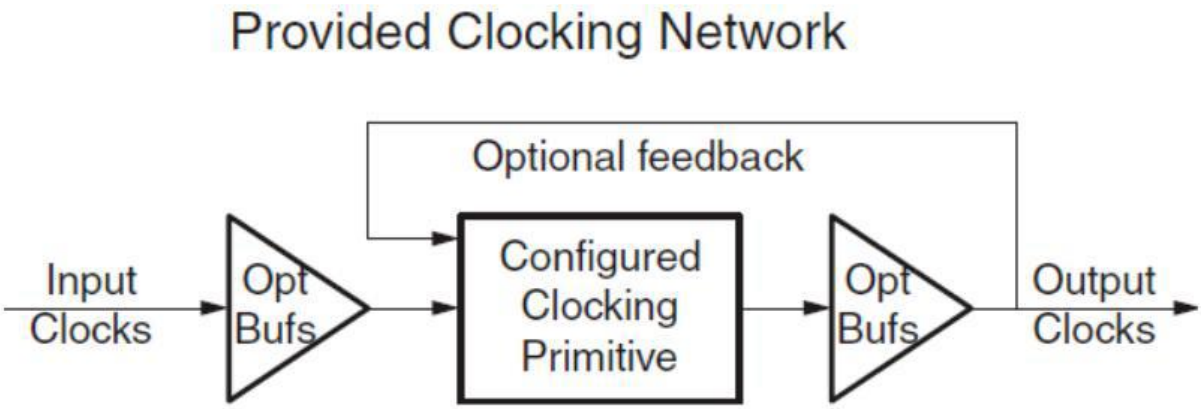


El asistente facilita la creación de envoltorios de código fuente HDL para circuitos de relojes personalizados para sus requisitos de reloj. El asistente lo guía para configurar los atributos apropiados para su primitiva de reloj, y también le permite anular cualquier parámetro calculado por el asistente. Además de proporcionar una envoltura HDL para implementar el circuito de reloj deseado, el Asistente de reloj también ofrece un resumen de parámetros de tiempo generado por las herramientas de tiempo de Xilinx para el circuito. Las características principales del asistente incluyen:

Acepta hasta dos relojes de entrada y hasta siete relojes de salida por red de reloj

- Elige automáticamente la primitiva de reloj correcta para un dispositivo seleccionado
- Configura automáticamente la primitiva de reloj basándose en las funciones de reloj seleccionadas por el usuario.
- Implementa automáticamente la configuración general que admite los cambios de fase y los requisitos del ciclo de trabajo
- Opcionalmente reforzadores de señales de reloj.

La funcionalidad del núcleo generado puede verse como:



Supongamos que queremos generar un reloj de 5 MHz en fase con un reloj de entrada de 100 MHz. Siga los pasos a continuación para lograr eso:

Haga doble clic en el asistente de reloj. Cuando se abra el asistente, notará que hay seis páginas configurables (pasos):

La primera página tiene parámetros relacionados con el reloj de entrada y las funciones de reloj. Desmarque la página de cambio de fase, se dan el valor de frecuencia de entrada y el rango. Dado que la fuente del reloj real es de 100 MHz, mantendremos el valor predeterminado.

Haga clic en Next para ver los parámetros relacionados con los relojes de salida y las frecuencias deseadas. Cambie la Frecuencia de salida solicitada a 1.000 MHz y observe que la frecuencia real que se muestra es 3.125 MHz, ya que es la frecuencia de reloj más lenta que se puede generar utilizando los recursos de reloj. Cambie a 5.000 MHz por ahora.

Haga clic en Next ara ver la página de configuración de Entradas/Salidas opcionales. Muestra RESET y LOCKED como el puerto seleccionado. Puede desmarcar/verificar cualquiera de los puertos y observar los cambios del diagrama de bloques en el lado izquierdo del asistente. Mantendremos el valor predeterminado, ya que nuestros circuitos tendrán entrada RESET y también nos gusta ver cuándo el reloj está estable.

Haga clic en Next para ver la página 4. Se mostrará el uso del recurso de reloj DCM_SP y el valor de los parámetros calculados. Mantener todo por defecto.

Haga clic en Next para ver la página 5 que muestra el resumen del reloj y la denominación de puertos. Mantener todo por defecto.

Haga clic en Next y aparecerá la página 6 mostrando los archivos que se generarán. Los archivos importantes son .veo (archivo de plantilla de creación de instancias), .v (el archivo de origen) y .ucf (archivo de restricciones del núcleo). Haga clic en Generate y el archivo central (.xci) se generará y agregará al proyecto.

De vuelta en la GUI de PlanAhead, para los núcleos no arquitectónicos, los archivos mencionados anteriormente se generan automáticamente. Sin embargo, para las fuentes arquitectónicas, los archivos deben ser explícitamente generados. Seleccione la entrada .xci en la pestaña Orígenes, haga clic con el botón derecho y seleccione la opción Generate Output Products… Aparecerá un formulario que muestra lo que se genera y lo que se puede generar como se muestra a continuación.

Si se selecciona la opción del menú desplegable Generar para la plantilla de creación de instancias, después de hacer clic en Aceptar, el menú generará los archivos .VHO y relacionados. Preste atención a la ubicación del producto de salida en la parte inferior del menú. Estos archivos son accesibles a través de la pestaña Fuentes de IP. El archivo .VHO es la plantilla de creación de instancias VHDL para la IP generada. Aquí hay un ejemplo del contenido del archivo .VHO:

```
-- The following code must appear in the VHDL architecture header:
component clk_wiz_v3_6_0
port
 (-- Clock in ports
  CLK_IN1           : in     std_logic;
  -- Clock out ports
  CLK_OUT1          : out    std_logic;
  -- Status and control signals
  RESET             : in     std_logic;
  LOCKED            : out    std_logic
 );
end component;

…

-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.
your_instance_name : clk_wiz_v3_6_0
  port map
   (-- Clock in ports
    CLK_IN1 => CLK_IN1,
    -- Clock out ports
    CLK_OUT1 => CLK_OUT1,
    -- Status and control signals
    RESET  => RESET,
    LOCKED => LOCKED);
```

II. OBJETIVO

- Usar el asistente arquitectónico para configurar el recurso de sincronización.
- Use la herramienta CORE Generator para configurar y usar contadores y memorias

III. MATERIAL Y EQUIPO

- Xilin ISE, 32-bit Project Navigator.

IV. DESARROLLO

1. Diseñar un generador de pulsos de un segundo. Utilice el asistente de reloj para generar un reloj de 5 MHz, dividiéndolo más por un divisor de reloj (escrito en modelos de comportamiento) para generar una señal de un segundo período. Los pasos para usar el Asistente de cronometraje descrito anteriormente (y la plantilla de creación de instancias resultante) se pueden usar para este ejercicio. Utilice la fuente de reloj de a bordo de 100 MHz, el botón BTNU para restablecer el circuito, SW0 como habilitación, LED0 para emitir la señal generada de un segundo y LED7 para emitir la señal de bloqueo del DCM. Ir a través del flujo de diseño, generar el flujo de bits y descargarlo en la placa Atlys. Verificar la funcionalidad. Asegúrese de especificar el idioma del proyecto a VHDL para generar los archivos de salida apropiados.
   Dado que no hay pantallas de 7 segmentos en la placa Atlys hay que conectar 7 segmentos externos que usan cátodos comunes y una pantalla particular se ilumina al afirmar el pin de ánodo correspondiente, se requiere un circuito de escaneo para mostrar información (dígitos) en más de una pantalla. Este circuito debe dirigir las señales de ánodo y los patrones de cátodo correspondientes de cada dígito en una sucesión continua y repetida, a una velocidad de actualización más rápida de lo que el ojo humano puede detectar. Para que cada uno de los dígitos aparezca brillante e iluminado continuamente, todos los dígitos deseados deben manejarse una vez cada 1 a 16 ms, para una frecuencia de actualización de 1 KHz a 60 Hz. Si la frecuencia de actualización o "refresco" se reduce a alrededor de 45 Hz, la mayoría de la gente comenzará a ver el parpadeo de la pantalla.

   Después de realizar el procedimiento de generación del clock con el clocking wizzard, se escribió el siguiente código:

   Codigo generado por el clocking wizzard

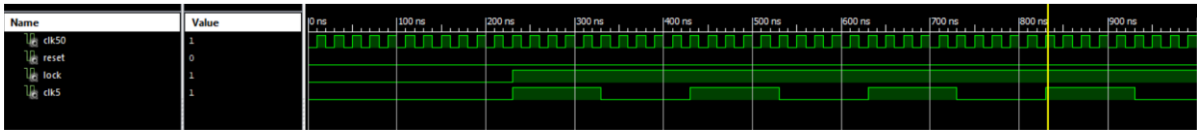
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity R50 is
Port ( clk50,reset : in STD_LOGIC;
lock,clk5 : out STD_LOGIC);
end R50;

architecture arch_rel of R50 is

component RB
port
(-- Clock in ports
CLK_IN1 : in std_logic;
-- Clock out ports
CLK_OUT1 : out std_logic;
-- Status and control signals
RESET : in std_logic;
LOCKED : out std_logic);
end component;
```

```
begin
---copia de reloj
RA:RB
port map
(-- Clock in ports
CLK_IN1 => clk50,
-- Clock out ports
CLK_OUT1 => clk5,
-- Status and control signals
RESET => reset,
LOCKED => lock);
end arch_rel
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
entity R50test is
end R50test;
architecture behavior of R50test is

Component R50
Port(clk50 : IN std_logic;
     reset : IN std_logic;
     lock : OUT std_logic;
     clk5 : OUT std_logic);
end component;

signal clk50 : std_logic := '0';
signal reset : std_logic := '0';

signal lock : std_logic;
signal clk5 : std_logic;

constant clk50p : time := 20 ns;
begin
uut: R50
port map(clk50 => clk50,
          reset => reset,
          lock => lock,
          clk5 => clk5);
clk500pr :process
begin
clk50 <= '0';
wait for clk50p/2;
clk50 <= '1';
wait for clk50p/2;
end process;
END;
```

Simulación



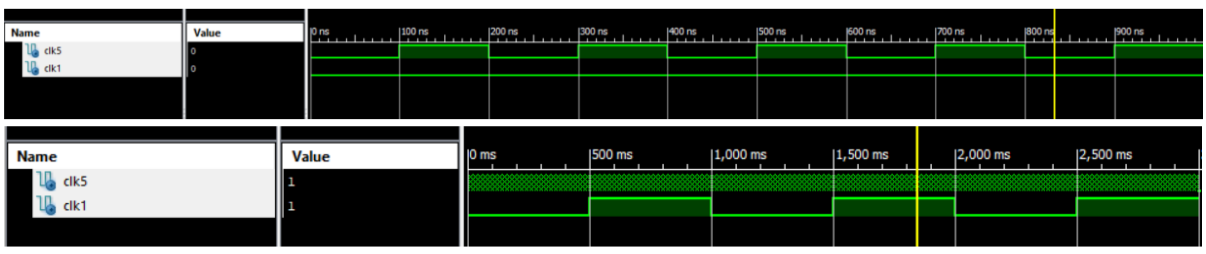
Segundo divisor de frecuencia de 5 MHz a 1 MHz

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity Div5_1 is
Port ( clk5,reset: in STD_LOGIC;
clk1 : out STD_LOGIC);
end Div5_1;

architecture arch of Div5_1 is
constant max :INTEGER := 2500000;
signal clk_state: STD_LOGIC :='0';
signal count: INTEGER range 0 to max;
begin

process (clk5)
begin
if reset='1' then
clk_state <= '0';
count <= 0;
elsif (clk5' event and clk5='1')then
if count <= max then
count <= count +1;
clk_state <= clk_state;
else
clk_state <= not clk_state;
count <= 0;
end if;
end if;
clk1 <= clk_state;
end process;
end arch;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Div5_1_test IS
END Div5_1_test;

ARCHITECTURE behavior OF Div5_1_test IS

signal clk5 : std_logic := '0';

signal clk1 : std_logic;
BEGIN

clk5 <= not clk5 after 100 ns; --periodo de 200ns
conec: entity work.Div5_1(arch) port map (clk5, clk1);
END;
```

Simulaciones:



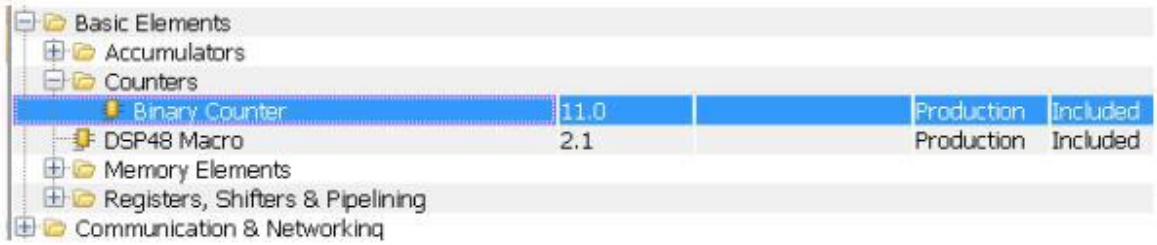
2. Implemente un convertidor de binario a BCD para mostrar las entradas binarias de 4 bits convertidas a valores BCD en dos pantallas de 7 segmentos (en lugar de una de 7 segmentos y un LED). Utilice la fuente de reloj de 100 MHz para generar un reloj de 5 MHz y el circuito divisor de reloj apropiado para controlar las dos pantallas de 7 segmentos con una frecuencia de actualización de aproximadamente 500 Hz. Genere el flujo de bits y descárguelo en la placa Nexys3 para verificar la funcionalidad.

    - Sistema generador de CORE

    El sistema Generador CORE, disponible en el Catálogo de IP de la herramienta PlanAhead, le permite configurar y generar varios núcleos funcionales. En el catálogo de IP, los núcleos se agrupan según la funcionalidad que varía desde núcleos básicos simples como un sumador hasta núcleos bastante complejos como el procesador MicroBlaze. También cubre núcleos de diversas áreas de aplicación que van desde Automotor hasta Procesamiento de Video e Imagen.

    El proceso de configuración y generación de los núcleos es similar al Asistente de arquitectura. Los núcleos utilizarán diversos recursos, como LUT, CLB, DSP48, BRAM, etc., según sea necesario. Veamos cómo configurar y generar un núcleo de contador.

    La generación de la base del contador binario se puede iniciar haciendo doble clic en la entrada Contador binario en la subcarpeta Contadores ubicada en la rama Elementos básicos del catálogo de IP.

    Cuando se invoca, verá solo una página de la configuración. Los parámetros de configuración del núcleo incluyen:

    Implementar utilizando: Fabric o DSP48
    Ancho de salida
    Valor de incremento
    Se puede cargar, restringir recuento, modo de recuento (arriba, abajo, UPDPWN), borrado sincrónico, habilitación de reloj y varios otros ajustes.

    El diseñador puede seleccionar la funcionalidad deseada y hacer clic en el botón Generar. Tenga en cuenta que, a diferencia del asistente de arquitectura, que requiere un paso explícito de generación de plantillas de creación de instancias, los núcleos no arquitectónicos generan automáticamente el archivo de plantillas de instanciación. Los archivos de salida y simulación sintetizados aún deben generarse explícitamente.

Codigo del convertidor binario a BCD

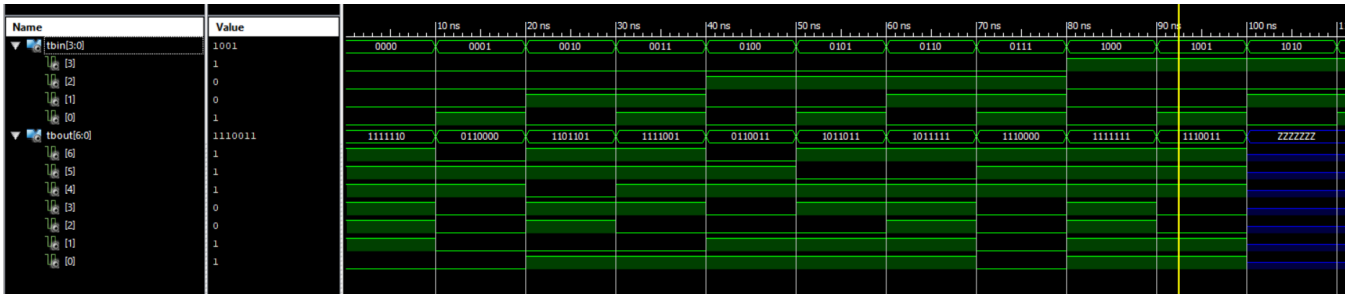
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BCD_7seg is
Port ( x3,x2,x1,x0 : in STD_LOGIC;
a,b,c,d,e,f,g : out STD_LOGIC);
end BCD_7seg;

architecture Behavioral of BCD_7seg is
begin
process (x3,x2,x1,x0)
variable VectOut: std_logic_vector (6 downto 0);
variable VectIn: std_logic_vector (3 downto 0);

begin
VectIn(3) := x3;
VectIn(2) := x2;
VectIn(1) := x1;
VectIn(0) := x0;
if VectIn = "0000" then VectOut := "1111110"; -- 0
elsif VectIn = "0001" then VectOut := "0110000"; -- 1
elsif VectIn = "0010" then VectOut := "1101101"; -- 2
elsif VectIn = "0011" then VectOut := "1111001"; -- 3
elsif VectIn = "0100" then VectOut := "0110011"; -- 4
elsif VectIn = "0101" then VectOut := "1011011"; -- 5
elsif VectIn = "0110" then VectOut := "1011111"; -- 6
elsif VectIn = "0111" then VectOut := "1110000"; -- 7
elsif VectIn = "1000" then VectOut := "1111111"; -- 8
elsif VectIn = "1001" then VectOut := "1110011"; -- 9
else VectOut := (others =>'Z');
end if;
a <= VectOut(6);
b <= VectOut(5);
c <= VectOut(4);
d <= VectOut(3);
e <= VectOut(2);
f <= VectOut(1);
g <= VectOut(0);
end process;
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY BCD_test IS
END BCD_test;

ARCHITECTURE behavior OF BCD_test IS

COMPONENT BCD_7seg
PORT(
x3 : IN std_logic;
x2 : IN std_logic;
x1 : IN std_logic;
x0 : IN std_logic;
a : OUT std_logic;
b : OUT std_logic;
c : OUT std_logic;
d : OUT std_logic;
e : OUT std_logic;
f : OUT std_logic;
g : OUT std_logic
);
END COMPONENT;
```

```
signal tbin: std_logic_vector (3 downto 0);
signal tbout: std_logic_vector (6 downto 0);
BEGIN

uut: BCD_7seg PORT MAP (
x3 => tbin(3),
x2 => tbin(2),
x1 => tbin(1),
x0 => tbin(0),
a => tbout(6),
b => tbout(5),
c => tbout(4),
d => tbout(3),
e => tbout(2),
f => tbout(1),
g => tbout(0)
);
stim_proc: process
begin
tbin <= "0000"; wait for 10 ns;
tbin <= "0001"; wait for 10 ns;
tbin <= "0010"; wait for 10 ns;
tbin <= "0011"; wait for 10 ns;
tbin <= "0100"; wait for 10 ns;
tbin <= "0101"; wait for 10 ns;
tbin <= "0110"; wait for 10 ns;
tbin <= "0111"; wait for 10 ns;
tbin <= "1000"; wait for 10 ns;
tbin <= "1001"; wait for 10 ns;
tbin <= "1010"; wait for 10 ns;
tbin <= "1011"; wait for 10 ns;
tbin <= "1100"; wait for 10 ns;
tbin <= "1101"; wait for 10 ns;
tbin <= "1110"; wait for 10 ns;
tbin <= "1111"; wait for 10 ns;
wait;
end process;
END;
```

Simulación:



3. Use el sistema CORE Generator para generar un simple núcleo de contador de 4 bits que cuenta de 0 a 9 (Sugerencia: use el Threshold output al configurar el núcleo del contador). Instáncielo dos veces para crear un contador BCD de dos dígitos que cuente cada segundo. Use el Asistente de arquitectura para generar un reloj de 5 MHz y luego use el modelado de comportamiento para generar una señal precisa de 1 Hz para impulsar los contadores. Muestra el resultado en las dos pantallas de 7 segmentos. La entrada de diseño será una fuente de reloj de 100 MHz, una señal de reinicio con el botón BTNU y una señal de habilitación con SW0. Verifique la funcionalidad de diseño en hardware utilizando la placa Atlys.

Codigo del contador de 0 a 9

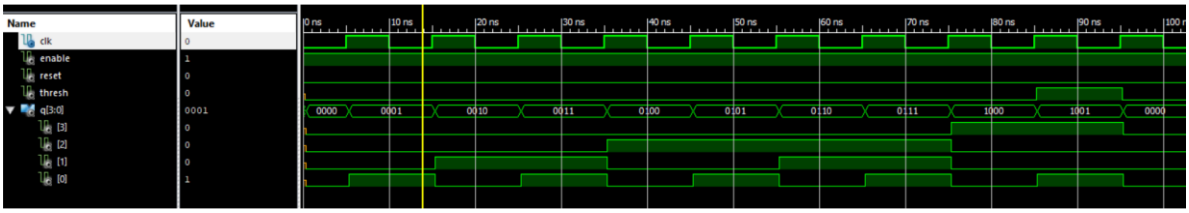
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Contador0_9 is
Port ( clk, enable, reset : in STD_LOGIC;
thresh : out STD_LOGIC;
q : out STD_LOGIC_VECTOR (3 downto 0));
end Contador0_9;

architecture Behavioral of Contador0_9 is

COMPONENT Counter0_9
PORT (
clk : IN STD_LOGIC;
ce : IN STD_LOGIC;
sclr : IN STD_LOGIC;
thresh0 : OUT STD_LOGIC;
q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;
begin
ContadorA: Counter0_9
PORT MAP (
clk => clk,
ce => enable,
sclr => reset,
thresh0 => thresh,
q => q
);
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Contador0_9_test IS
END Contador0_9_test;

ARCHITECTURE behavior OF Contador0_9_test IS

COMPONENT Contador0_9
PORT(
clk : IN std_logic;
enable : IN std_logic;
reset : IN std_logic;
thresh : OUT std_logic;
q : OUT std_logic_vector(3 downto 0)
);
END COMPONENT;

signal clk : std_logic := '0';
signal enable : std_logic := '1';
signal reset : std_logic := '0';

signal thresh : std_logic;
signal q : std_logic_vector(3 downto 0);
```

```
constant clk_period : time := 10 ns;
BEGIN

uut: Contador0_9 PORT MAP (
clk => clk,
enable => enable,
reset => reset,
thresh => thresh,
q => q
);

clk_process :process
begin
clk <= '0';
wait for clk_period/2;
clk <= '1';
wait for clk_period/2;
end process;
END;
```

Simulación



Código del divisor de frecuencia de 5 MHz a 500 Hz

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Divisor5_500 is
Port ( clk5, reset: in STD_LOGIC;
clk500 : out STD_LOGIC);
end Divisor5_500;

architecture arch of Divisor5_500 is
constant max :INTEGER := 5000;
signal clk_state: STD_LOGIC :='0';
signal count: INTEGER range 0 to max;
begin

process (clk5)
begin
if reset='1' then
clk_state <= '0';
count <= 0;
elsif (clk5' event and clk5='1')then
if count <= max then
count <= count +1;
clk_state <= clk_state;
else
clk_state <= not clk_state;
count <= 0;
end if;
end if;
clk500 <= clk_state;
end process;
end arch;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Divisor_5_500_test IS
END Divisor_5_500_test;

ARCHITECTURE behavior OF Divisor_5_500_test IS

signal clk5 : std_logic := '0';
signal reset : std_logic := '0';

signal clk500 : std_logic;
BEGIN

clk5 <= not clk5 after 100 ns;
conec: entity work.Divisor5_500(arch) port map (clk5,reset, clk500);
process
begin
reset<='0'; wait for 3 ms;
reset<='1'; wait for 2 ms;
reset<='0'; wait;
end process;
END;
```

Simulación:

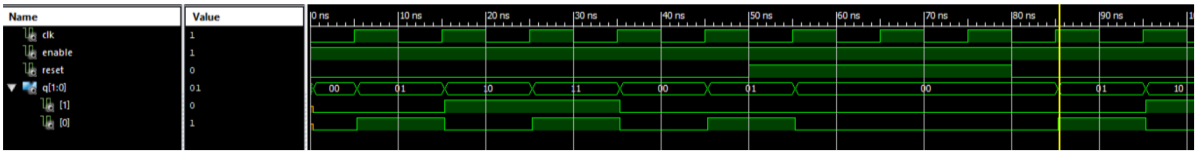
Código del contador de 1 bit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Contador2 is
Port ( clk,enable, reset : in STD_LOGIC;
q : out STD_LOGIC_VECTOR (1 downto 0));
end Contador2;

architecture Behavioral of Contador2 is

COMPONENT Cont2
PORT (
clk : IN STD_LOGIC;
ce : IN STD_LOGIC;
sclr : IN STD_LOGIC;
q : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT;
begin

ContadorB : Cont2
PORT MAP (
clk => clk,
ce => enable,
sclr => reset,
q => q
);
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Contador2_test IS
END Contador2_test;

ARCHITECTURE behavior OF Contador2_test IS

COMPONENT Contador2
PORT(
clk : IN std_logic;
enable : IN std_logic;
reset : IN std_logic;
q : OUT std_logic_vector(1 downto 0)
);
END COMPONENT;

signal clk : std_logic := '0';
signal enable : std_logic := '1';
signal reset : std_logic := '0';

signal q : std_logic_vector(1 downto 0);
constant clk_period : time := 10 ns;
BEGIN
uut: Contador2 PORT MAP (
clk => clk,
enable => enable,
reset => reset,
q => q
);
clk_process :process
begin
clk <= '0';
wait for clk_period/2;
clk <= '1';
wait for clk_period/2;
end process;
stim_proc: process
begin
reset<='0'; wait for 5*clk_period;
reset<='1'; wait for 3*clk_period;
reset<='0'; wait;
end process;
END;
```

Simulación:



Código del divisor y controlador de los 7 segmentos

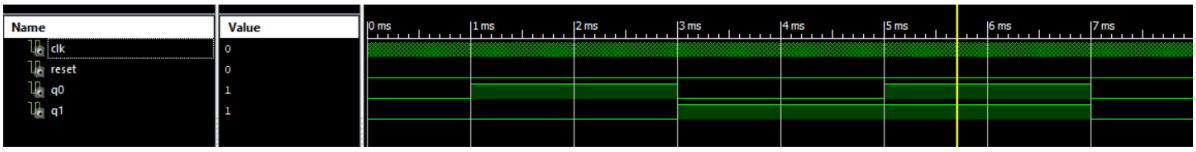
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Div_control is
Port ( clk, reset : in STD_LOGIC;
q0,q1 : out STD_LOGIC);
end Div_control;

architecture Behavioral of Div_control is

signal clk_cd: std_logic;

component Contador2 is
Port ( clk,enable, reset : in STD_LOGIC;
q : out STD_LOGIC_VECTOR (1 downto 0));
end component Contador2;

component Divisor5_500 is
Port ( clk5, reset: in STD_LOGIC;
clk500 : out STD_LOGIC);
end component Divisor5_500;

begin
C: Contador2
port map(clk=>clk_cd, enable=>'1', reset=>reset,q(0)=>q0, q(1)=>q1);
D: Divisor5_500
port map (clk5=>clk, clk500=>clk_cd, reset=>reset);
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Div_control_test IS
END Div_control_test;

ARCHITECTURE behavior OF Div_control_test IS

COMPONENT Div_control
PORT(
clk : IN std_logic;
reset : IN std_logic;
q0 : OUT std_logic;
q1 : OUT std_logic
);
END COMPONENT;

signal clk : std_logic := '0';
signal reset : std_logic := '0';

signal q0 : std_logic;
signal q1 : std_logic;
```

```
constant clk_period : time := 200 ns;
BEGIN

uut: Div_control PORT MAP (
clk => clk,
reset => reset,
q0 => q0,
q1 => q1
);

clk_process :process
begin
clk <= '0';
wait for clk_period/2;
clk <= '1';
wait for clk_period/2;
end process;
END;
```

Simulación:



Código del multiplexor

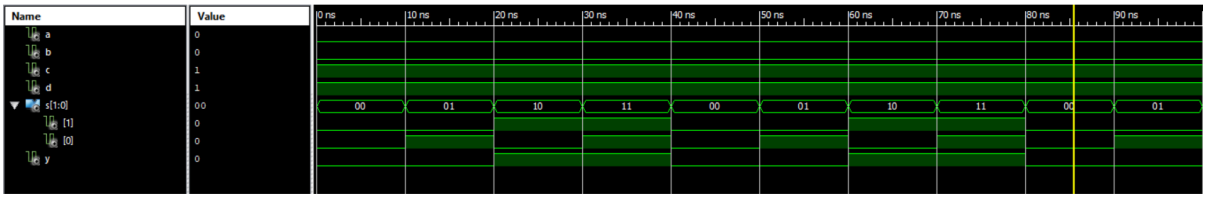
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multiplexor is
Port ( a,b,c,d : in STD_LOGIC;
s : in STD_LOGIC_VECTOR (1 downto 0);
y : out STD_LOGIC);
end Multiplexor;

architecture Behavioral of Multiplexor is
begin
y <= a when s="00" else
b when s="01" else
c when s="10" else
d;
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY multiplexor_test IS
END multiplexor_test;

ARCHITECTURE behavior OF multiplexor_test IS

COMPONENT Multiplexor
PORT(
a : IN std_logic;
b : IN std_logic;
c : IN std_logic;
d : IN std_logic;
s : IN std_logic_vector(1 downto 0);
y : OUT std_logic
);
END COMPONENT;

signal a : std_logic := '0';
signal b : std_logic := '0';
signal c : std_logic := '1';
signal d : std_logic := '1';
signal s : std_logic_vector(1 downto 0) := (others => '0');
```

```
signal y : std_logic;
BEGIN

uut: Multiplexor PORT MAP (
a => a,
b => b,
c => c,
d => d,
s => s,
y => y
);

stim_proc: process
variable i: integer :=0;
begin

for i in 0 to 3 loop
s<="00"; wait for 10 ns ;
s<="01"; wait for 10 ns ;
s<="10"; wait for 10 ns ;
s<="11"; wait for 10 ns ;
end loop;
end process;
END;
```

Simulación:



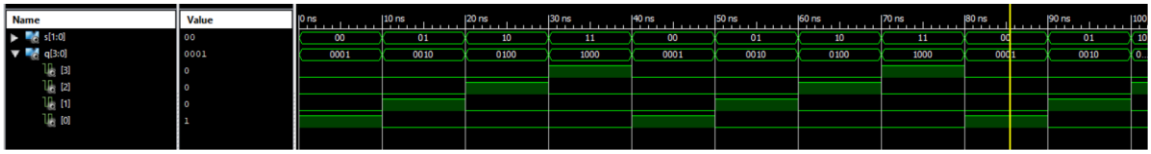
Código del demultiplexor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Demux is
Port ( s : in STD_LOGIC_VECTOR (1 downto 0);
q : out STD_LOGIC_VECTOR (3 downto 0));
end Demux;

architecture Behavioral of Demux is
begin
process(s)
begin
if (s="00") then
q <= "0001";
elsif (s="01") then
q <= "0010";
elsif (s="10") then
q <= "0100";
else q <="1000";
end if;
end process;
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Demux_test IS
END Demux_test;
ARCHITECTURE behavior OF Demux_test IS
COMPONENT Demux
PORT(
s : IN std_logic_vector(1 downto 0);
q : OUT std_logic_vector(3 downto 0)
);
END COMPONENT;
signal s : std_logic_vector(1 downto 0) := (others => '0');
signal q : std_logic_vector(3 downto 0);
BEGIN
uut: Demux PORT MAP (
s => s,
q => q
);
stim_proc: process
variable i: integer :=0;
begin
for i in 0 to 3 loop
s<="00"; wait for 10 ns ;
s<="01"; wait for 10 ns ;
s<="10"; wait for 10 ns ;
s<="11"; wait for 10 ns ;
end loop;
end process;
END;
```

Simulación:



Codigo del multiplexor completo

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux_total is
Port ( s : in STD_LOGIC_VECTOR (1 downto 0);
c0,c1 : in STD_LOGIC_VECTOR (3 downto 0);
w,x,y,z : out STD_LOGIC;
an0,an1,an2,an3 : out STD_LOGIC);
end Mux_total;
architecture Behavioral of Mux_total is
component Multiplexor is
Port ( a,b,c,d : in STD_LOGIC;
s : in STD_LOGIC_VECTOR (1 downto 0);
y : out STD_LOGIC);
end component Multiplexor;
component Demux is
Port ( s : in STD_LOGIC_VECTOR (1 downto 0);
q : out STD_LOGIC_VECTOR (3 downto 0));
end component Demux;
begin
mux0: Multiplexor
port map(a=>c0(0), b=>c1(0), c=>'0', d=>'0',s=>s, y=>z);
mux1: Multiplexor
port map(a=>c0(1), b=>c1(1), c=>'0', d=>'0',s=>s, y=>y);
mux2: Multiplexor
port map(a=>c0(2), b=>c1(2), c=>'0', d=>'0',s=>s, y=>x);
mux3: Multiplexor
port map(a=>c0(3), b=>c1(3), c=>'0', d=>'0',s=>s, y=>w);
dem: Demux
port map(s=>s, q(3)=>an3, q(2)=>an2, q(1)=>an1, q(0)=>an0);
end Behavioral;
```

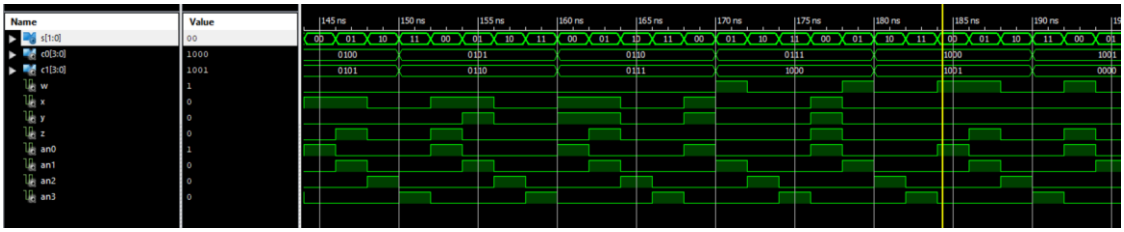
Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Mux_total_test IS
END Mux_total_test;
ARCHITECTURE behavior OF Mux_total_test IS)
COMPONENT Mux_total
PORT(
s : IN std_logic_vector(1 downto 0);
c0 : IN std_logic_vector(3 downto 0);
c1 : IN std_logic_vector(3 downto 0);
w : OUT std_logic;
x : OUT std_logic;
y : OUT std_logic;
z : OUT std_logic;
an0 : OUT std_logic;
an1 : OUT std_logic;
an2 : OUT std_logic;
an3 : OUT std_logic
);
END COMPONENT;
signal s : std_logic_vector(1 downto 0) := (others => '0');
signal c0 : std_logic_vector(3 downto 0) := (others => '0');
signal c1 : std_logic_vector(3 downto 0) := (others => '0');
signal w : std_logic;
signal x : std_logic;
signal y : std_logic;
signal z : std_logic;
signal an0 : std_logic;
signal an1 : std_logic;
signal an2 : std_logic;
signal an3 : std_logic;
```

```
BEGIN
uut: Mux_total PORT MAP (
s => s,
c0 => c0,
c1 => c1,
w => w,
x => x,
y => y,
z => z,
an0 => an0,
an1 => an1,
an2 => an2,
an3 => an3
);
signal_s: process
variable i: integer :=0;
begin
for i in 0 to 3 loop
s<="00"; wait for 2 ns ;
s<="01"; wait for 2 ns ;
s<="10"; wait for 2 ns ;
s<="11"; wait for 2 ns ;
end loop;
end process;
```

```
signal_c0: process
variable i: integer :=0;
begin
for i in 0 to 3 loop
c0<="0000"; wait for 1 us ;
c0<="0001"; wait for 1 us ;
c0<="0010"; wait for 1 us ;
c0<="0011"; wait for 1 us ;
c0<="0100"; wait for 1 us ;
c0<="0101"; wait for 1 us ;
c0<="0110"; wait for 1 us ;
c0<="0111"; wait for 1 us ;
c0<="1000"; wait for 1 us ;
c0<="1001"; wait for 1 us ;
end loop;
end process;
signal_c1: process
variable i: integer :=0;
begin
for i in 0 to 3 loop
c1<="0001"; wait for 1 us ;
c1<="0010"; wait for 1 us ;
c1<="0011"; wait for 1 us ;
c1<="0100"; wait for 1 us ;
c1<="0101"; wait for 1 us ;
c1<="0110"; wait for 1 us ;
c1<="0111"; wait for 1 us ;
c1<="1000"; wait for 1 us ;
c1<="1001"; wait for 1 us ;
c1<="0000"; wait for 1 us ;
end loop;
end process;
END;
```

Simulación:



Codigo principal

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Sistema_completo is
Port ( clk, enable, reset: in STD_LOGIC;
an : out STD_LOGIC_VECTOR (3 downto 0);
seg : out STD_LOGIC_VECTOR (6 downto 0);
lock : out STD_LOGIC);
end Sistema_completo;
architecture Behavioral of Sistema_completo is
signal reloj5,reloj1,acarreo: std_logic;
signal mux_dec,con0,con1:std_logic_vector(3 downto 0);
signal sel:std_logic_vector(1 downto 0);

component R50 is
Port ( clk50,reset : in STD_LOGIC;
lock,clk5 : out STD_LOGIC);
end component R50;

component Div5tol is
Port ( clk5,reset: in STD_LOGIC;
clk1 : out STD_LOGIC);
end component Div5tol;
```

```
component Contador0_9 is
Port ( clk, enable, reset : in STD_LOGIC;
thresh : out STD_LOGIC;
q : out STD_LOGIC_VECTOR (3 downto 0));
end component Contador0_9;

component Div_control is
Port ( clk, reset : in STD_LOGIC;
q0,q1 : out STD_LOGIC);
end component Div_control;

component Mux_total is
Port ( s : in STD_LOGIC_VECTOR (1 downto 0);
c0,c1 : in STD_LOGIC_VECTOR (3 downto 0);
w,x,y,z : out STD_LOGIC;
an0,an1,an2,an3 : out STD_LOGIC);
end component Mux_total;

component BCD_7seg is
Port ( x3,x2,x1,x0 : in STD_LOGIC;
a,b,c,d,e,f,g : out STD_LOGIC);
end component BCD_7seg;
```

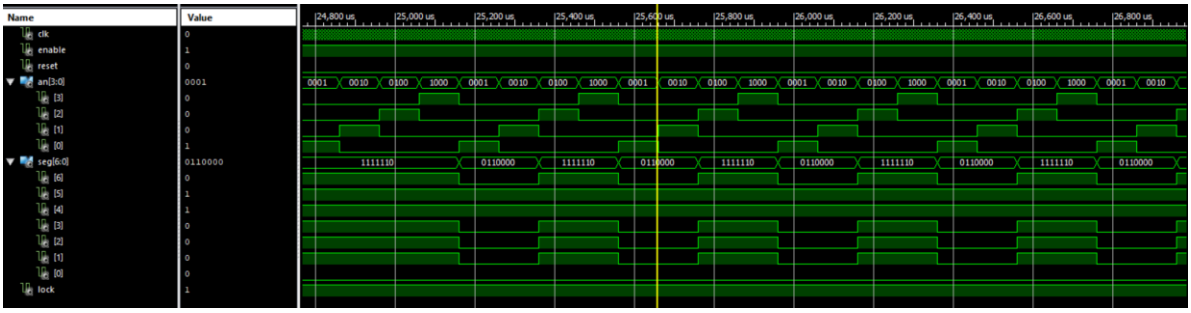
```
begin
U1: Reloj50_5
port map (clk50=>clk, reset=>reset, lock=>lock, clk5=>reloj5);
U2:Div5_1
port map (clk5=>reloj5, reset=>reset,clk1=>reloj1);
U3: Contador0_9
port map (clk=>reloj1, enable=> enable, reset=>reset,thresh=>acarreo, q=>con0);
U4: Contador0_9
port map (clk=>reloj1, enable=> enable, reset=>reset,thresh=> OPEN, q=>con1);
U5: Div_control
port map (clk=>reloj5, reset=>reset, q0=>sel(0),q1=>sel(1));
U6: Mux_total
port map (s=>sel, c0=>con0, c1=>con1, w=>mux_dec(3), x=>mux_dec(2),y=>mux_dec(1),z=>mux_dec(0), an0=>an(0),an1=>an(1),an2=>an(2),an3=>an(3));
U7: BCD_7seg
port map(x3=>mux_dec(3), x2=>mux_dec(2), x1=>mux_dec(1),x0=>mux_dec(0),a=>seg(6),b=>seg(5),c=>seg(4),d=>seg(3),e=>seg(2),f=>seg(1),g=>seg(0));
end Behavioral;
```

Test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY sis_com_test IS
END sis_com_test;
ARCHITECTURE behavior OF sis_com_test IS
COMPONENT Sistema_completo
PORT(
clk : IN std_logic;
enable : IN std_logic;
reset : IN std_logic;
an : OUT std_logic_vector(3 downto 0);
seg : OUT std_logic_vector(6 downto 0);
lock : OUT std_logic
);
END COMPONENT;
signal clk : std_logic := '0';
signal enable : std_logic := '1';
signal reset : std_logic := '0';

signal an : std_logic_vector(3 downto 0);
signal seg : std_logic_vector(6 downto 0);
signal lock : std_logic;

constant clk_period : time := 1 ns;
```

```
BEGIN
uut: Sistema_completo PORT MAP (
clk => clk,
enable => enable,
reset => reset,
an => an,
seg => seg,
lock => lock
);

clk_process :process
begin
clk <= '0';
wait for clk_period/2;
clk <= '1';
wait for clk_period/2;
end process;
END;
```

Simulación:



V. CONCLUSIÓN

El entono de clocking wizzard de Xilin nos permite modificar directamente la frecuencia interna de funcionamiento de los relojes de la tarjeta para ciertos equipos que cuentan con estas características. Esto es útil para el manejo eficiente y mássencillo de los relojes y los divisores de frecuencia para la generación de pulsos clock de un periodo especifico.

VI. REFERENCIAS

D. L. Perry, VHDL. New York: McGraw-Hill, 1991