SISTEMAS ELÉCTRÓNICOS DIGITALES

GUIÓN DE PRÁCTICAS DE LABORATORIO

PRÁCTICA 2

DECODIFICADOR BCD A 7 SEGMENTOS

Versión 2021_v1

Escuela Técnica Superior de Ingeniería y Diseño Industrial
Departamento de Electricidad, Electrónica, Automática y Física Aplicada.

OBJETIVOS DE LA PRÁCTICA

Con la realización de esta práctica se persiguen los siguientes objetivos:

- Familiarizarse con la creación de programas de pruebas o testbenches.
- Familiarizase con el uso de restricciones (asignación de patillas).
- Familiarizarse con el uso de los recursos de la placa de prácticas: displays de 7 segmentos y botones
- Empezar a diseñar sistemas compuestos por varios bloques.

1. INTRODUCCIÓN.

Normalmente, el objetivo último de nuestro diseño será convertir nuestro dispositivo lógico programable en un **componente a medida**. La entidad de nivel superior (llamada normalmente "**top**") se corresponde con este componente y define la interfaz de nuestro diseño con el exterior. Lo habitual es que consista en una descripción estructural en la que se describe la interconexión entre los componentes de nivel jerárquico inferior de nuestro diseño.

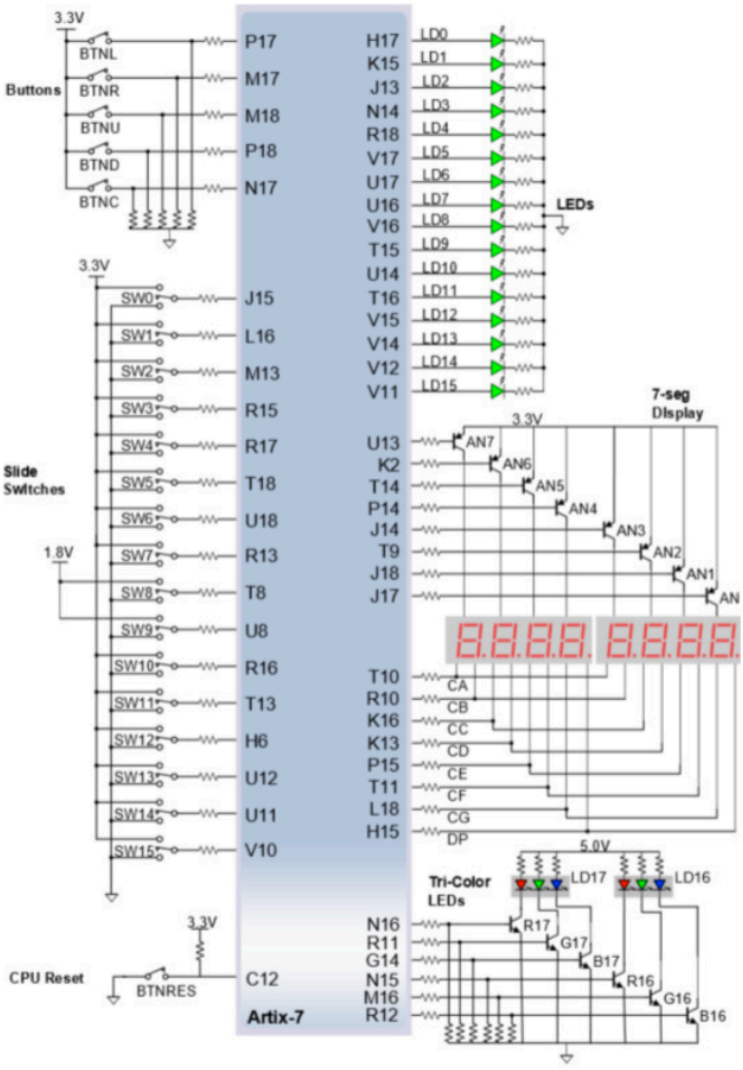

Figura 1: Diagrama de conexiones de la Nexys 4 DDR.

Es a esta entidad a la que se le aplican las restricciones de asignación de señales a patillas concretas. Como ya se ha visto en la práctica 1, para realizar la asignación de restricciones en la placa Nexys 4 DDR se recomienda descargar en fichero Master XDC de la URL:
https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start

En esta práctica vamos a crear una entidad top compuesta de varios componentes, y se va a trabajar con elementos indispensables en los trabajos: displays de 7 segmentos y botones, es decir, entradas y salidas.

Para mayor comodidad a la hora de asignar patillas y periféricos, se incluye un diagrama de conexiones de éstos en la placa de desarrollo Nexys 4 DDR, el cual se puede ver en la Fig. 1.

2. REALIZACIÓN DE LA PRÁCTICA

En esta práctica vamos a utilizar **cuatro displays de 7 segmentos** de la placa para visualizar un **número binario** introducido a través de los **interruptores deslizantes**.

Se empezará por controlar un solo dígito y se irá progresando hasta controlar los cuatro disponibles.

2.1. Decodificador BCD a 7 segmentos

El principal componente de este diseño será un **decodificador de BCD a siete segmentos**. Un decodificador es un circuito combinacional que realiza la operación inversa a la de un codificador de datos, por eso también se puede definir un decodificador como circuito que convierte un código binario concreto en una forma sin codificar.

Los dígitos de la tarjeta son de tipo LED. Cada segmento es un LED que tiene un terminal accesible y el otro común para todos los segmentos: existen versiones de cátodo común y de ánodo común. En nuestro caso se trata de un ánodo común, por lo que para que luzca un segmento debemos aplicar al terminal no común un '0'. La placa tiene tantos dígitos que, para reducir el total de líneas, todos ellos comparten las mismas señales de control de los segmentos; esto es, todos los terminales no comunes de los segmentos A están unidos, y así hasta el segmento G. **La forma de mostrar números distintos en cada dígito consiste en irlos encendiendo en secuencia haciendo que las líneas de control de los segmentos (ANx) reflejen el número correspondiente al dígito activo en ese momento**. Si está secuencia se ejecuta a suficiente velocidad el ojo resulta engañado y percibe todos los dígitos iluminados a la vez. Para iluminar un dígito concreto en esta tarjeta es necesario aplicar un '0' a la línea de control pertinente.

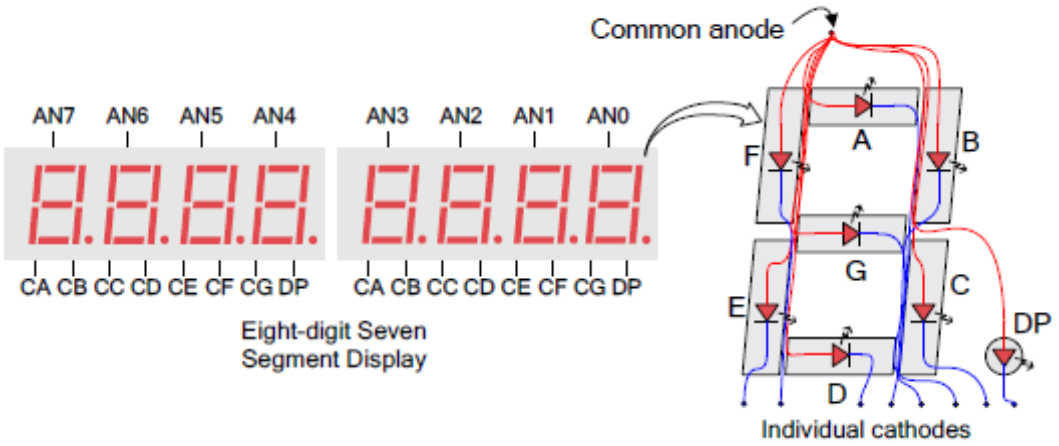

Figura 2: Visualizador de 8 dígitos y líneas correspondientes de la FPGA.

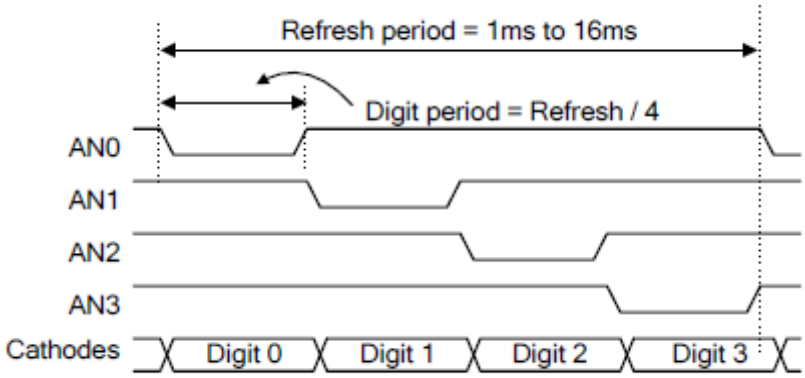

Figura 3: Secuencia de encendido de los dígitos.

Teniendo esto en cuenta podemos obtener fácilmente la tabla de verdad de este decodificador:

A	B	C	D	a	b	C	d	e	f	g
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0

Figura 4: Tabla de verdad decodificador BCD a 7 segmentos.

Ahora estamos en condiciones de crear nuestro decodificador:

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

ENTITY decoder IS
	PORT (
		code : IN  std_logic_vector(3 DOWNTO 0);
		led  : OUT std_logic_vector(6 DOWNTO 0)
	);
END ENTITY decoder;

ARCHITECTURE dataflow OF decoder IS
BEGIN
	WITH code SELECT
		led <=  "0000001" WHEN "0000",
			"1001111" WHEN "0001",
			"0010010" WHEN "0010",
			"0000110" WHEN "0011",
			"1001100" WHEN "0100",
			"0100100" WHEN "0101",
			"0100000" WHEN "0110",
			"0001111" WHEN "0111",
			"0000000" WHEN "1000",
			"0000100" WHEN "1001",
			"1111110" WHEN others;
END ARCHITECTURE dataflow;
```

Y, a continuación, creamos el *tesbench*:

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

ENTITY decoder_tb IS
END decoder_tb;

ARCHITECTURE BEHAVIORAL OF decoder_tb IS
	COMPONENT decoder
		PORT(
			code : IN  std_logic_vector(3 DOWNTO 0);
			led  : OUT std_logic_vector(6 DOWNTO 0)
		);
	END COMPONENT;

	SIGNAL code : std_logic_vector(3 DOWNTO 0);
	SIGNAL led  : std_logic_vector(6 DOWNTO 0);

	TYPE vtest is record
		code : std_logic_vector(3 DOWNTO 0);
		led  : std_logic_vector(6 DOWNTO 0);
	END RECORD;

	TYPE vtest_vector IS ARRAY (natural RANGE <>) OF vtest;

	CONSTANT test: vtest_vector := (
		(code => "0000", led => "0000001"),
		(code => "0001", led => "1001111"),
		(code => "0010", led => "0010010"),
		(code => "0011", led => "0000110"),
		(code => "0100", led => "1001100"),
		(code => "0101", led => "0100100"),
```

```
                (code => "0110", led => "0100000"),
                (code => "0111", led => "0001111"),
                (code => "1000", led => "0000000"),
                (code => "1001", led => "0000100"),
                (code => "1010", led => "1111110"),
                (code => "1011", led => "1111110"),
                (code => "1100", led => "1111110"),
                (code => "1101", led => "1111110"),
                (code => "1110", led => "1111110"),
                (code => "1111", led => "1111110")
        );

BEGIN
        uut: decoder PORT MAP(
                code => code,
                led  => led
        );

        tb: PROCESS
        BEGIN
                FOR i IN 0 TO test'HIGH LOOP
                        code <= test(i).code;
                        WAIT FOR 20 ns;
                        ASSERT led = test(i).led
                                REPORT "Salida incorrecta."
                                SEVERITY FAILURE;
                END LOOP;

                ASSERT false
                        REPORT "Simulacin finalizada. Test superado."
                        SEVERITY FAILURE;
        END PROCESS;
END BEHAVIORAL;
```

Tarea 1 (para casa)

- Ejecutar el *tesbench* del decodificador en el simulador y comprobar que no hay errores y que el comportamiento del módulo corresponde con el esperado.
- Si todo funciona correctamente, en la consola no debería salir ningún mensaje del tipo “Salida incorrecta”, y sí un mensaje “Simulación finalizada. Test superado.”
- Analice el código del testbench y pregunte lo que no entienda.

2.2. Entidad top

A continuación **vamos a crear una entidad “top”** que contenga el decodificador que hemos creado anteriormente. A la entidad top le conectaremos las señales de entrada (código o número a mostrar) y de salida (señales que van a los displays).

Usaremos el decodificador para mostrar en cuatro displays de 7 segmentos el **número binario introducido a través de los interruptores deslizantes SW3, SW2, SW1 y SW0**. Los **dígitos activos se elegirán por medio de SW15, SW14, SW13 y SW12**. Recuerde que todos los displays comparten las mismas señales de control de los segmentos, por lo que sólo hay que activar el dígito en el que se quiera mostrar el dato, y desactivar los demás.

Los interruptores ponen a '0' su línea asociada cuando están abajo (hacia el exterior de la tarjeta) y a '1' cuando están arriba (hacia el interior).

Para ello se creará un nuevo módulo VHDL con una entidad y su correspondiente arquitectura como indica la siguiente figura:

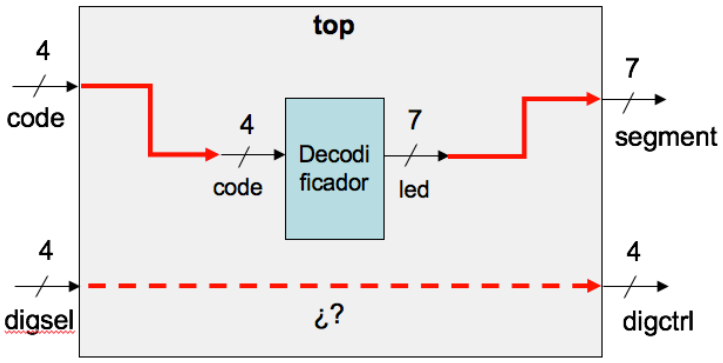

Figura 5. Esquema de la estructura "top"

La entidad tendrá los siguientes puertos:

```
ENTITY top IS
    PORT ( code    : IN  std_logic_vector(3 DOWNTO 0);
           digsel  : IN  std_logic_vector(3 DOWNTO 0);
           digctrl : OUT std_logic_vector(3 DOWNTO 0);
           segment : OUT std_logic_vector(6 DOWNTO 0)
    );
END top;
```

La arquitectura, con una descripción de tipo estructural, contendrá un único componente, el decodificador, y las señales necesarias para realizar las conexiones apropiadas entre el decodificador, los interruptores, los segmentos y los ánodos de los dígitos. **Es posible que sea necesario invertir alguna señal: lo podemos hacer con el operador de VHDL `NOT`.**

Para utilizar el decodificador como un componente de nuestra entidad top deberemos declararlo primero e instanciarlo después.

La sintaxis para la declaración es:

```
COMPONENT decoder
       PORT (
              code : IN  std_logic_vector(3 DOWNTO 0);
              led  : OUT std_logic_vector(6 DOWNTO 0)
              );
END COMPONENT;
```

Y la sintaxis de instanciación es:

```
Inst_decoder: decoder PORT MAP (
       code => ,
       led =>
);
```

Para implementar la entidad top se procede de la siguiente manera:

1. Copiar la declaración del componente en la parte declarativa (entre `ARCHITECTURE` y `BEGIN`) de la arquitectura de `top`.
2. Copiar la instanciación del componente en la parte ejecutiva (entre `BEGIN` y `END`) de la arquitectura de `top`.

Después de incluir el componente `decoder` en nuestra arquitectura, por medio de señales conectaremos los puertos de la entidad `top` con los puertos del componente `decoder` como sea necesario para obtener la funcionalidad pedida. **Complete las asignaciones de señales que faltan**.

Una vez realizadas las conexiones y corregido cualquier error sintáctico que hubiéramos podido cometer, podemos sintetizar el diseño y generar el fichero de programación. En estas condiciones la herramienta realizará una implementación optimizada en recursos (usar el mínimo número de bloques lógicos posibles, mantener la longitud de las conexiones tan corta como sea posible, etc.). Sin embargo, en este caso el circuito impreso ya se encuentra fabricado y cada uno de sus componentes conectado a una patilla concreta de la FPGA. Sólo la casualidad hará coincidir las patillas elegidas por la herramienta para cada señal con aquellas a las que realmente debería conectarse.

Para conectar cada señal a la patilla correcta debemos imponer *restricciones* a la herramienta. En concreto restricciones a la asignación de los recursos: las patillas.

En la tarjeta de prácticas, los interruptores deslizantes están conectados a las siguientes patillas de la FPGA:

SW15	SW14	SW13	SW12	SW3	SW2	SW1	SW0
V10	U11	U12	H6	R15	M13	L16	J15

Figura 6: Conexión de los interruptores a la FPGA

Las conexiones de los dígitos de siete segmentos se dan en la Figura 1.

Para asociar las señales con las patillas adecuadas de la FPGA se debe utilizar el **fichero de restricciones Master XDC (ya utilizado en la práctica 1)**, descomentando y modificando como corresponda cada línea (básicamente cambiando el nombre de la señal asociada a la patilla) en función de los recursos de la placa que se van a utilizar. Debería asignar las señales code, digsel, digctrl y segment.

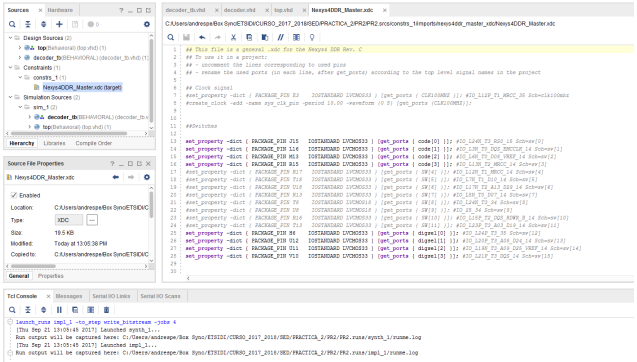

Figura 7: Fichero de restricciones

Es importante **marcar el modulo top como la entidad principal del proyecto**, en caso de que no se haya hecho automáticamente. Para hacerlo, se pulsa con el botón derecho sobre el fichero top en la ventana Sources y se pulsa “Set as Top”, tal y como se muestra en la siguiente imagen:

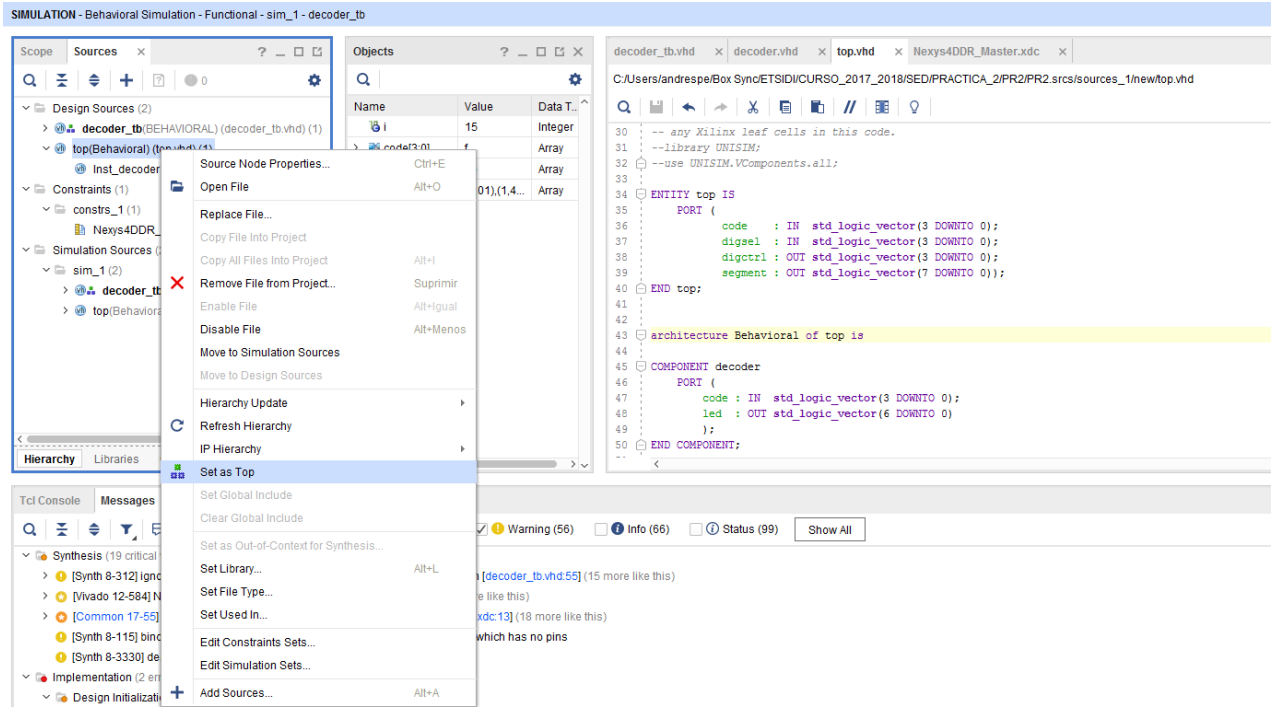

Figura 8: Asignar top como entidad principal del proyecto.

Tarea 2

- Crear la entidad top con el decodificador, generar el fichero de programación (bitstream) y programar la placa tal y como se realizó en la práctica 1.
- Comprobar que funciona como se espera.

2.3. Pulsadores

A continuación vamos a añadir un pulsador que incremente un contador (de 0 a 9) cuya salida se visualice en los displays de siete segmentos. Cada vez que se pulse el botón se deberá incrementar el contador en una unidad.

El uso de pulsadores es prácticamente imprescindible en cualquier trabajo para interaccionar con el usuario. Sin embargo, aunque parezcan sencillos de usar, su utilización no es tan directa.

Por un lado, su uso viene acompañado de un inconveniente: **el rebote o *bouncing***. Al ser accionados, los pulsadores tardan cierto tiempo en alcanzar un estado estable, es decir, que durante muy poco tiempo (del orden de los milisengundos) pueden cambiar entre el estado pulsado y no pulsado hasta que la señal se estabiliza. A pesar de ser muy poco tiempo, es el suficiente para que un sistema digital lo detecte como varias pulsaciones.

Por otro lado, toda entrada externa que se introduzca al sistema debe ser **sincronizada**, porque si no sería una fuente de **inestabilidades** (metaestabilidades). Además, el software Vivado suele dar errores o warnings si no se hace.

Por tanto el esquema que vamos a realizar es el siguiente:

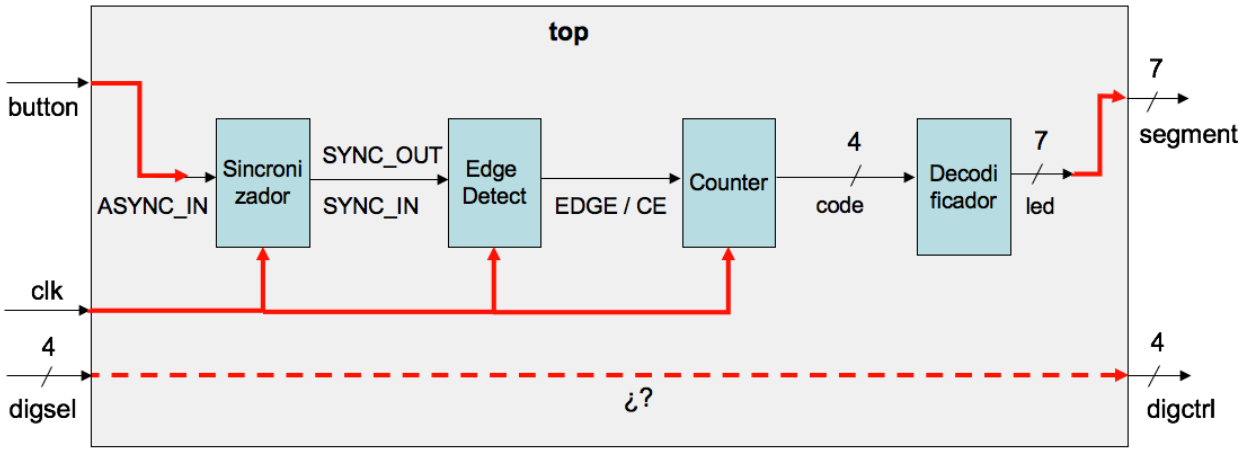

Figura 9: Esquema a realizar

En este caso no utilizaremos el módulo antirrebotes por simplificación y porque con nuestra placa el sincronizador suele ser suficiente.

2.4. Módulo de sincronización

Como se ha comentado, las señales de entrada a la FPGA pueden generar metaestabildiad. Esto ocurre cuando, por ejemplo, conectamos la entrada a un flip flop y la señal de entrada cambia en alguno de los flancos de la señal de reloj, no respetando el tiempo de setup o de hold (ver Figura 10). Esto ocurre cuando queremos detectar un flanco de la señal de entrada y usamos 'event. Además, Vivado suele detectar estos problemas y nos da un error o warning.

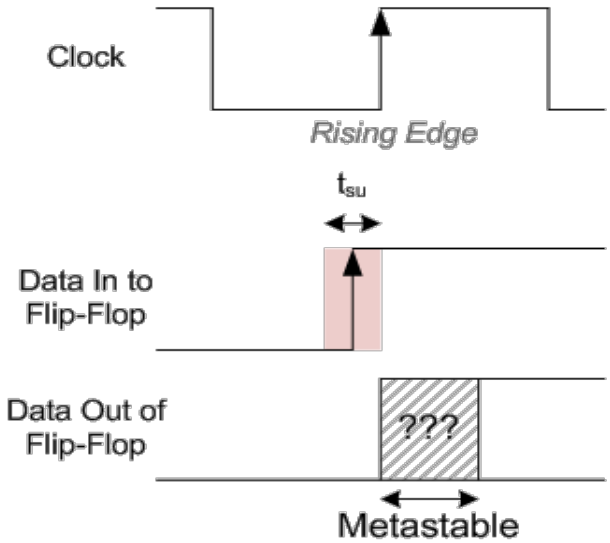

Figura 10: Ejemplo de metaestabilidad

Una de las posibles soluciones pasa por hacer la señal de entrada por varios Flip-Flops, de manera que minimizamos las probabilidades de que ocurra la metaestabilidad. Como ejemplo se tiene la Figura 11.

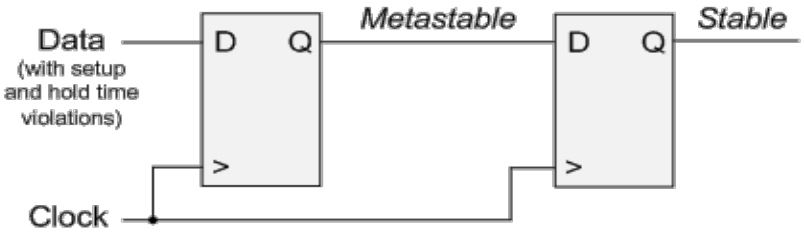

Figura 11: Ejemplo de solución a la metaestabilidad

La parte de sincronización se puede realizar con el siguiente código).

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

entity SYNCHRNZR is
  port (
    CLK      : in  std_logic;
    ASYNC_IN : in  std_logic;
    SYNC_OUT : out std_logic
  );
end SYNCHRNZR;

architecture BEHAVIORAL of SYNCHRNZR is
  signal sreg : std_logic_vector(1 downto 0);
begin
  process (CLK)
  begin
     if rising_edge(CLK) then
       sync_out <= sreg(1);
       sreg <= sreg(0) & async_in;
     end if;
  end process;
end BEHAVIORAL;
```

Más información:
https://www.nandland.com/articles/metastability-in-an-fpga.html

Tarea 4

- Analice el comportamiento de esta entidad y describa su funcionamiento.

2.5. Detector de flanco

Cuando se pulsa el botón se genera una pulso de duración indeterminada (hasta que se suelta el botón). Esto es un inconveniente en circuitos síncronos, donde nos gusta que todo esté bien medido.

Para solucionar esto se utiliza este módulo, el cual genera un solo pulso (de duración un período de la señal de reloj) cada vez que hay un flanco de bajada en la señal de entrada (botón). De esta forma, cada vez que pulsemos el botón se generará una señal que dura un ciclo de reloj, y consecuentemene sólo habrá un flanco de subida en un período de reloj.

A continuación se presenta el código que un sincronizador:

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

entity EDGEDTCTR is
  port (
    CLK      : in  std_logic;
    SYNC_IN  : in  std_logic;
    EDGE     : out std_logic
  );
end EDGEDTCTR;

architecture BEHAVIORAL of EDGEDTCTR is
  signal sreg : std_logic_vector(2 downto 0);
begin
  process (CLK)
  begin
    if rising_edge(CLK) then
      sreg <= sreg(1 downto 0) & SYNC_IN;
    end if;
  end process;

  with sreg select
    EDGE <= '1' when "100",
            '0' when others;
end BEHAVIORAL;
```

Tarea 5

- Analice el comportamiento de esta entidad y describa su funcionamiento.

2.6. Contador

Para implementar el circuito de la Figura 9 sólo faltaría implementar el “contador”. Utilice, por ejemplo, esta declaración:

```
entity counter is
    PORT (
       clk  : in  std_logic;                      --Clock
       CE   : in  std_logic;                      --Chip Enable
       code : out std_logic_vector(3 downto 0)    --Valor de 0 a 9
    );
end counter;
```

El contador se incrementará en una unidad cada vez que haya un ‘1’ en “CE”. Para ello es importante que la señal que llega a “CE” sea un pulso de duración un periodo de reloj. Cuando “CE” valga uno y haya un flanco de reloj “clk”, “code” se incrementa en una unidad.

Nota.- Al ser un sistema síncrono, todos los cambios se deben producir según la señal de reloj. No se incrementa el contador cuando hay un flanco positivo en "CE", si no con el flanco positivo de "clk". Y para elegir cuando, se utiliza la señala "CE".

Tarea 6

- Diseñe la arquitectura del módulo contador que cuente de 0 a 9.

2.7. Implementación

Finalmente implementaremos el circuito de la Figura 9 mediante una entidad "top" que contenga los módulos vistos.

Nota.- Para la realización de varios módulos necesitará una señal de reloj. Tome la de la placa, localizando el pin correspondientes en el manual de la placa o en el fichero de restricciones.

Tarea 7

- Integre todos los módulos junto con el decodificador del apartado anterior en una entidad top y prueba el sistema.

Tarea 8: Señal de RESET

- Implemente una señal de RESET que ponga a cero el contador. Asígnela al botón de RESET de la placa. La señal RESET deberá llegar a todas las entidades que sea necesario.

Tarea 9: Memoria

- Realizar una memoria de la práctica con los ejercicios pedidos, incluyendo el código desarrollado, imagen de la simulación en caso de realizarla e impresiones personales.

--- Fin de la práctica ---