PEC2: Diseño y simulación de circuitos digitales programables

Profesores responsables

Profesor responsable:

- Dr. Pere Tuset <peretuset@uoc.edu>

Profesores colaboradores:

- Dr. Francisco Vázquez <fvazquezg@uoc.edu>

Presentación

Esta PEC se focaliza en diferentes aspectos de la microelectrónica actual, desde el proceso de diseño de un chip, hasta la implementación de un diseño sobre una FPGA (Field Programmable Gate Array), pasando por las características de los diversos tipos de ASICs (Application Specific Integrated Circuits), aspectos económicos, dispositivos lógicos programables, etc. Es muy importante que se conozca el material de base facilitado en la asignatura, en concreto, la PEC contiene un conjunto de cuestiones y problemas relacionados con los contenidos de los Módulos 4 y 5.

Competencias

- Conocer las características generales y las herramientas involucradas en el proceso de diseño de un circuito integrado de aplicación específica (ASIC).
- Conocer los diferentes tipos de ASICs, con sus diferencias, ventajas e inconvenientes.
- Implementar funciones lógicas simples en dispositivos lógicos programables.
- Conocer las características de las FPGAs disponibles en el mercado y poder comparar entre ellas.
- Diseñar sobre FPGAs complejas, aplicando técnicas específicas, y verificar el correcto funcionamiento de la implementación resultante.

Objetivos

- Conocer los diferentes tipos de ASIC y sus características principales.
- Entender el proceso de diseño de un circuito integrado y las herramientas asociadas.
- Conocer las características principales de las FPGAs modernas.
- Aprender a implementar un diseño simple en VHDL, verificarlo y sintetizarlo sobre una FPGA.

Recursos

Los recursos que se recomienda utilizar para esta PEC son los siguientes:

- **Recursos básicos:**
  - **Módulo 4.** "Introducción a los circuitos integrados de aplicación específica", Jordi Riera Baburés, UOC
  - **Módulo 5**. "Dispositivos lógicos programables", Jordi Riera Baburés, UOC
  - "Simulación de sistemas digitales mediante lenguajes descriptores de hardware", Juan Antonio Martínez Carrascal, UOC
  - "VHDL Simulation: Test bench", diapositives UOC
- **Recursos complementarios:**
  - Quartus® Prime Introduction: Using VHDL Designs, Intel Corporation - FPGA University Program, June 2018
  - Using the Intel® Quartus® Prime Standard Edition Software: An Introduction (80') https://www.intel.com/content/www/us/en/programmable/support/training/course/odsw1100.html

Criterios de valoración

- Razonar la respuesta en todos los ejercicios. Las respuestas sin justificación no recibirán puntuación.
- La valoración se indica en cada uno de los subapartados.

Formato y fecha de entrega

- Hay que entregar la solución en un archivo ZIP, que contenga la solución completa de la PEC en formato PDF utilizando una de las plantillas entregadas conjuntamente con este enunciado, así como el archivo completo de los diseños de los ejercicios que requieren codificación VHDL (preferiblemente utilizando herramienta del Quartus, Project -> Archive Project).
- La solución de la PEC en PDF debe incluir todo el código en VHDL, tanto del diseño como de los bancos de pruebas utilizados en las simulaciones, así como todas las gráficas obtenidas, junto con los comentarios adecuados. Piense que si hay algún problema para reproducir los resultados de su diseño, esto será lo único que quedará para defender su trabajo.
- Se entregará a través de la aplicación de Entrega y registro de EC del apartado Evaluación de su aula.
- Para dudas y aclaraciones sobre el enunciado, diríjase al consultor responsable de su aula.
- La fecha límite de entrega es el **28 de Abril** (a las 23:59 horas).

Descripción de la actividad

Esta PEC está compuesta por diferentes tipos de actividades, en concreto:

- **Preguntas teóricas** (30%): Encontrará una serie de afirmaciones, respecto a temas que tenemos que asimilar, y se trata de que razone por qué cada una de ellas es cierta o falsa. El objetivo es asimilar conceptos mediante el razonamiento de diferentes características de los ASICs, su proceso de diseño y en especial de los dispositivos lógicos programables o FPGAs. Hay que razonar siempre cada una de las opciones de respuesta; las respuestas no razonadas se calificarán con cero puntos.
- **Problemas prácticos** (50%): Se plantean diferentes actividades prácticas relacionadas con el entorno de desarrollo de Altera para FPGAs, que deberá trabajar a partir de conocimientos del material de la asignatura y/o otras fuentes de información.
- **Cuestión de investigación** (20%): Encontrará una pregunta abierta, sin una solución cerrada y única, donde deberá hacer búsqueda de información, material o datos, con el fin de trabajar aspectos de la microelectrónica de hoy en día que van más allá de lo que hay en los apuntes.

Enunciado

Preguntas teóricas (30%)

Pregunta 1 (15%)

Determina si cada una de las siguientes afirmaciones son CIERTAS o FALSAS respecto a los ASICs y su proceso de diseño. **Hay que razonar siempre cada una de las opciones de respuesta**; las respuestas no razonadas adecuadamente se calificarán con cero puntos.

a) Los ASICs de tipo full-custom, hechos completamente a medida, son los que más se utilizan a nivel industrial porque son los más rápidos y con menos consumo.
b) A diferencia del método de diseño tradicional de un ASIC, en el proceso de diseño dirigido por prestaciones, el diseño físico sigue al diseño lógico, como si fueran dos etapas aisladas.
c) El rendimiento de un proceso de fabricación no se ve afectado por la medida del chip que se fabrica, sino por el número de defectos por unidad de área.

Solución:

a) **Falso**. A pesar de ser los que pueden obtener mejores prestaciones, menor consumo y ser los que menos área ocupan (menos coste), no se utilizan por dos motivos principales: el primero es que el tiempo de diseño es mucho más largo, alargando el tiempo de introducción en el mercado; el segundo es que se hace difícil verificar su correcto funcionamiento, ya que las celdas hechas a medida no disponen de modelos para poder

hacer simulaciones de alto nivel (y las simulaciones eléctricas de todo un ASIC son inviables).

b) **Falso**. A diferencia del método de diseño tradicional, en el proceso de diseño dirigido por prestaciones de un ASIC, los diseños lógico y físico transcurren en paralelo. Es decir, que se avanza por ambos al mismo tiempo, y considerando desde el primer momento unos estimadores de los dispositivos parásitos.

c) **Falso**. El rendimiento de un proceso de fabricación mide el número de chips correctos respecto de los fabricados. Para un mismo número de defectos en la oblea, el rendimiento será muy diferente según la medida del chip. Por ejemplo, suponemos un solo defecto en una oblea: si la oblea contiene únicamente 10 chips, el rendimiento obtenido es de 9/10 = 90%. En cambio, si el chip fuera más pequeño y cupieran 100, entonces el rendimiento sería de 99/100 = 99%.

Pregunta 2 (15%)

Determina si cada una de las siguientes afirmaciones son CIERTAS o FALSAS respecto a las FPGAs y los dispositivos lógicos programables en general. **Hay que razonar siempre cada una de las opciones de respuesta**; las respuestas no razonadas adecuadamente se calificarán con cero puntos.

a) Las FPGAs basadas en memoria RAM son ideales para el prototipado debido a su alta densidad de integración.

b) Las FPGAs basadas en memoria RAM no suelen utilizarse en aplicaciones militares, aviónica y espacio.

c) Los problemas de metaestabilidad se pueden evitar utilizando un circuito PLL que recibe un reloj externo de entrada y genera uno o varios relojes internos a la FPGA a partir de éste.

Solución:

a) **Falso**. Las FPGAs basadas en memoria RAM tienen una baja densidad de integración. Las celdas de memoria RAM ocupan un espacio significativo, a diferencia de los antifusibles, que prácticamente no influyen en el área total. Además, las conexiones entre la salida de las celdas de memoria y los puntos de configuración también requieren un área adicional.

b) **Cierto**. Las aplicaciones militares, aviónica y espacio son aplicaciones críticas respecto a la radiación ionizante. Las FPGAs basadas en memoria RAM son sensibles a la radiación. Por tanto, no suelen usarse en este tipo de aplicaciones. Las celdas de memoria son susceptibles de presentar fallos causados por la radiación, que, en el peor de los casos, pueden alterar su contenido. En este tipo de aplicaciones es preferible utilizar FPGAs basadas en antifusibles.

c) **Falso**. Para evitar problemas de metaestabilidad se tienen que utilizar elementos sincronizadores. El más simple se basa en dos biestables consecutivos. Este circuito aumenta el tiempo medio entre fallos de varios órdenes de magnitud respecto del uso de un solo biestable para capturar la señal. Otras alternativas consisten en utilizar memorias de tipo FIFO al atravesar dominios de reloj.

Instalación de Software: Quartus Prime y ModelSim

Para realizar los ejercicios prácticos de la PEC2 y la PRAC2 tenemos que instalar el software de Altera. A continuación, se detallan los pasos del proceso de descarga, instalación y configuración del software.

1) En primer lugar tenemos que crear una cuenta de usuario en el portal de Intel a través del siguiente enlace. A partir de este momento, podremos entrar en el portal de Intel utilizando nuestra cuenta de usuario (**Sign In**).
   https://www.intel.com/content/www/us/en/homepage.html
2) A continuación, vamos a descargar el software Quartus Prime Lite Edition a través del siguiente enlace.
   https://fpgasoftware.intel.com/18.1/?edition=lite
   a) Antes de iniciar la descarga, debemos configurar los siguientes elementos:
      - **Select edition**: Lite
      - **Select release**: 18.1
      - **Operating system**: Windows o Linux

   a) En la pestaña **Combined Files**, iniciar la descarga del siguiente fichero:
      i) En Windows: Quartus-lite-18.1.0.625-windows.tar
      ii) En Linux: Quartus-lite-18.1.0.625-linux.tar

   b) Una vez finalizada la descarga, ir al directorio sobre el que se ha realizado la descarga (Downloads) del fichero y descomprimirlo.

   c) Una vez descomprimido el fichero, inicia el proceso de instalación ejecutando:
      i) En Windows: **setup.bat**
      ii) En Linux: **setup.sh**

      En Windows, seleccionar el siguiente directorio para la instalación:
      C:\intelFPGA_lite\18.1

      ATENCIÓN: Se debe instalar el **Quartus Prime**, el **ModelSim-Altera Intel FPGA Starter Edition** (este no requiere licencia y es gratuito), y el **Ciclone IV device support** (es posible añadir otros dispositivos, pero no es imprescindible para la realización de los ejercicios). No se debe instalar la aplicación **ModelSim-Altera Intel FPGA Edition**, esto puede provocar problemas de licencia en las simulaciones.

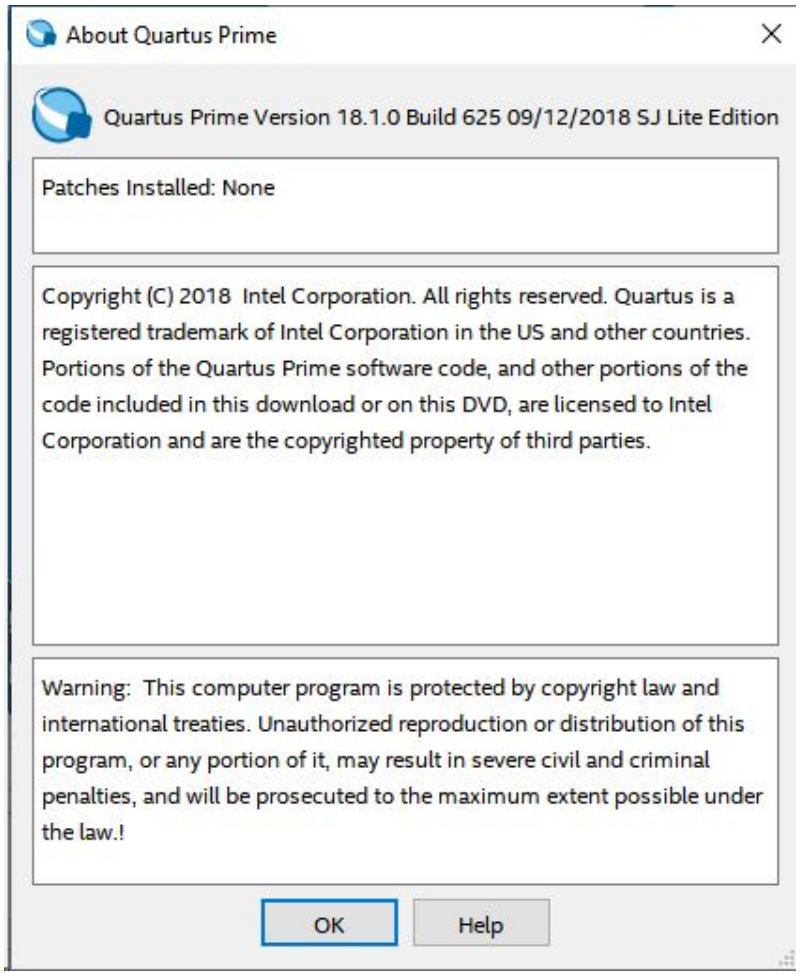
      IMPORTANTE: para evitar problemas, se recomienda no utilizar nombres de directorio con espacios en blanco ni acentuados.

   d) Una vez finalizado el proceso de instalación, ejecutar la aplicación Quartus Prime haciendo click sobre el icono del escritorio. En la aplicación Quartus Prime,

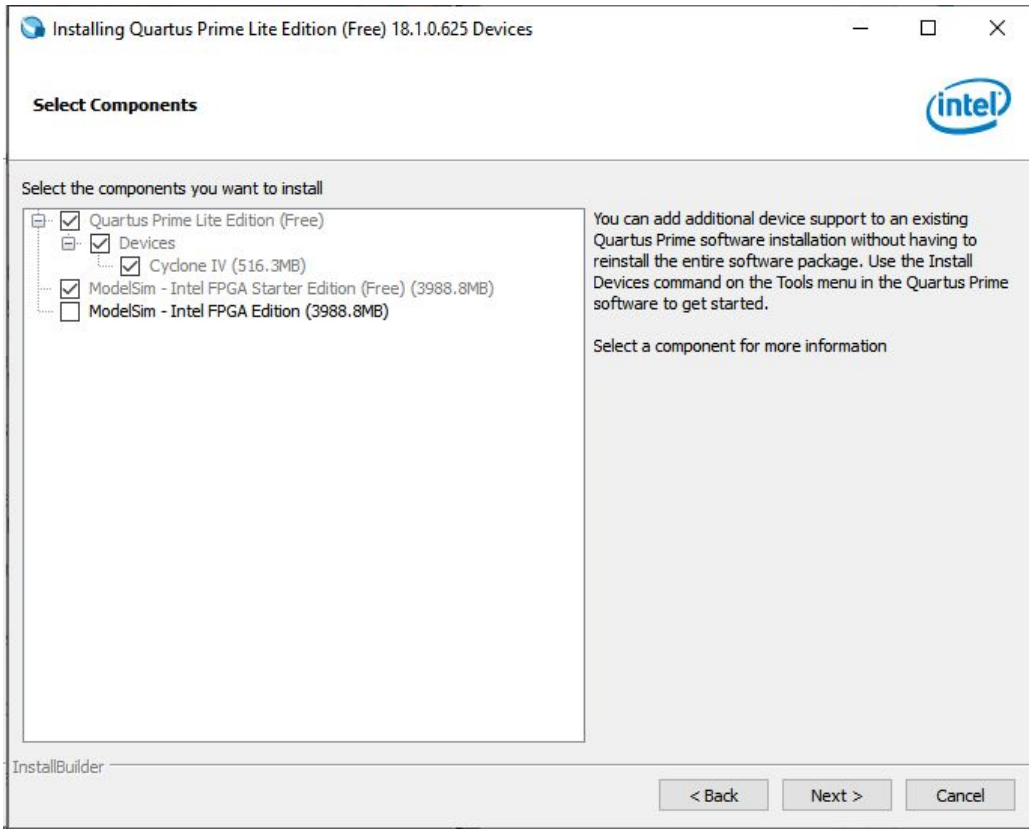
configurar el path del ejecutable del ModelSim-Altera de la forma siguiente. A través del menú *Tools* > *Options*, aparecerá la ventana de configuración. Seleccionar *General* > *EDA Tool Options*, y en el apartado de ModelSim-Altera, introducir el path del ejecutable:

C:\intelFPGA_lite\18.1\modelsim_ase\win32aloem

3) Ejecutar el **Quartus Prime**, ir a *Help* → *About Quartus Prime* y comprobar la versión instalada.



4) Ejecutar el **Device Installer** para ver los dispositivos y componentes que hemos descargado e instalado. Si utilizas Windows, puedes ejecutar el Device Installer a través del grupo de aplicaciones "Intel FPGA" que encontrarás en el menú de inicio de Windows. Se abrirá una aplicación donde tenemos que seleccionar el directorio en el que hemos descargado nuestros dispositivos. A continuación se abrirá una ventana donde se pueden seleccionar nuevos componentes, y nos muestra también los que ya tenemos. En la siguiente captura de pantalla se puede comprobar que tenemos instalados el *«Modelsim-Intel FPGA Starter Edition»* y las FPGAs de la familia *«Cyclone IV»*.



Problemas prácticos (50%)

Problema 1 (10%)

La modulación por ancho de pulsos, en inglés, *Pulse Width Modulation* (PWM), es una técnica de modulación típicamente utilizada para controlar el movimiento de servo-motores, ajustar el nivel de iluminación de lámparas LED, etc. La señal de salida de un modulador de tipo PWM es una señal digital con una frecuencia fija y un Duty Cycle (ciclo de trabajo) variable definido como

Duty-Cycle= t_on / (t_on + t_off) = t_on / t_pwm,

donde "t_on" es el tiempo en que la señal está a nivel lógico alto "1", "t_off" es el tiempo en que la señal está a nivel lógico bajo "0", y "t_pwm" es el periodo de la señal. Los intervalos "t_on" y "t_off" son variables y el período "t_pwm" es constante. La información de control de la señal PWM está contenida en el valor del Duty Cycle, de manera que cuanto mayor es el valor del Duty Cycle, mayor será "t_on", y más alta será la velocidad de rotación del motor o más alto será el nivel de iluminación.

El siguiente listado corresponde al código VHDL de un modulador PWM sencillo:

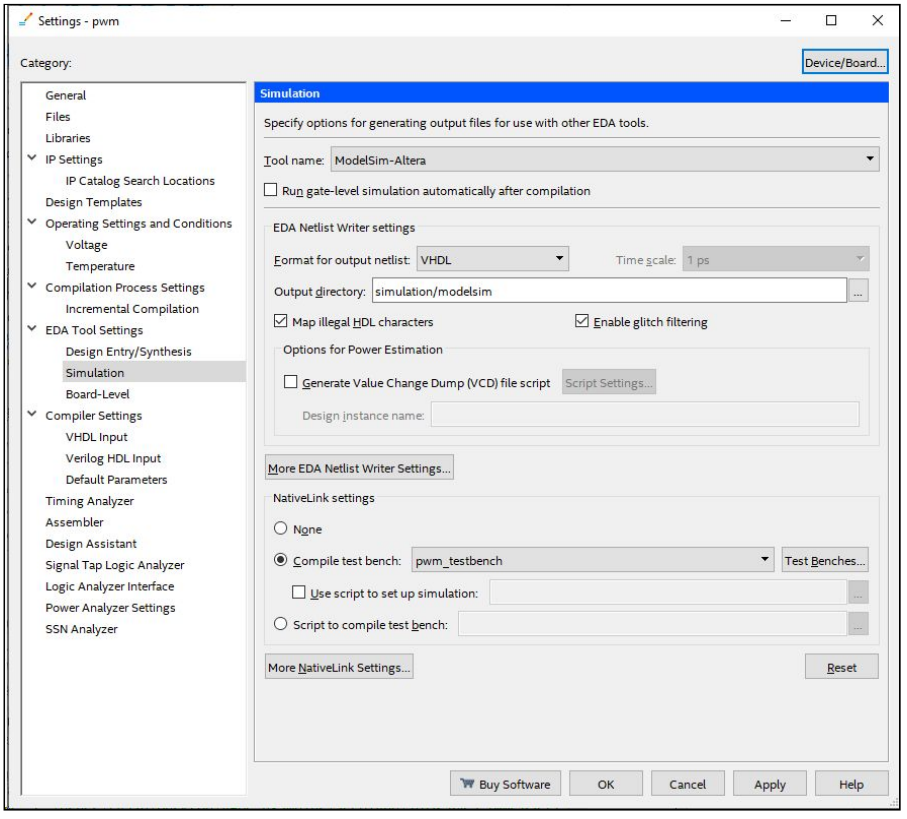
```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity pwm is
6      generic (
7         TON_W    : natural := 6;    -- Number of bits of PWM counter
8         CNT_MAX  : natural := 31    -- Maximum value of PWM counter
9         );
10     port (
11        clk      : in std_logic;    -- Clock signal
12        rst      : in std_logic;    -- Reset signal
13        t_on     : in std_logic_vector(TON_W-1 downto 0);-- Time ON
14        pwm_out  : out std_logic    -- PWM output signal
15     );
16  end entity pwm;
17
18  architecture rtl of pwm is
19     signal cnt : unsigned(TON_W-1 downto 0);
20
21  begin
22
23     cnt_pr : process(clk, rst)
24     begin
25        if (rst = '1') then
26           cnt <= (others => '0');
27           pwm_out <= '0';
28        elsif (rising_edge(clk)) then
29           if (cnt = CNT_MAX) then
30              cnt <= (others => '0');
31           else
32              cnt <= cnt + 1;
33           end if;
34           if (cnt < unsigned(t_on)) then
35              pwm_out <= '1';
36           else
37              pwm_out <= '0';
38           end if;
39        end if;
40     end process cnt_pr;
41
42  end rtl;
43
```

Y este sería un posible banco de pruebas para el diseño anterior:

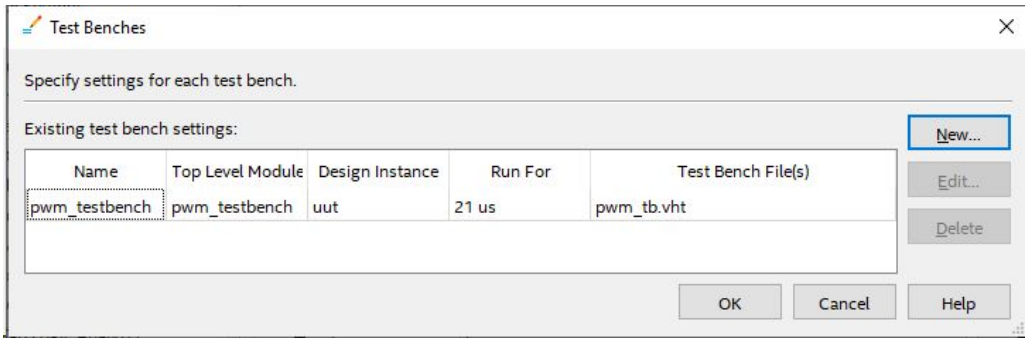
```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity pwm_testbench IS
6   end pwm_testbench;
7
8   architecture behavior of pwm_testbench is
9
10      -- Unit under test.
11      component pwm
12          port(
13              clk      : in  std_logic;
14              rst      : in  std_logic;
15              t_on     : in std_logic_vector(5 downto 0);
16              pwm_out  : out std_logic
17          );
18      end component;
19
20      signal clk : std_logic := '0';
21      signal reset: std_logic := '0';
22      signal t_on : unsigned (5 downto 0) := (others => '0');
23      signal pwm_out : std_logic;
24
25      -- Clock definition
26      constant clk_period : time := 10 ns;
27
28  begin
29      -- Instance of the unit under test.
30      uut: pwm PORT MAP (
31          clk => clk,
32          rst => reset,
33          t_on => std_logic_vector(t_on),
34          pwm_out  => pwm_out
35      );
36
37      -- Definition of the clock process.
38      clk_process :process
39      begin
40          clk <= '0';
41          wait for clk_period/2;
42          clk <= '1';
43          wait for clk_period/2;
44      end process;
45
46      -- Stimuli process.
47      stimuli: process
48      begin
49          reset <= '1';
50          wait for 50 ns;
51          reset <= '0';
52
53          for k in 0 to 31 loop
54              wait for clk_period * 32;
55              t_on <= t_on + 1;
56          end loop;
57
58          for k in 0 to 31 loop
59              wait for clk_period * 32;
60              t_on <= t_on - 1;
61          end loop;
62
63          wait;
64      end process;
65  end;
66
```

Junto con el enunciado de la PEC, se os ha proporcionado un archivo llamado "2019_PAC2_pwm_Part1.qar" que corresponde al proyecto de diseño y simulación del código del PWM mediante el banco de pruebas. Con él se os pide hacer lo siguiente:

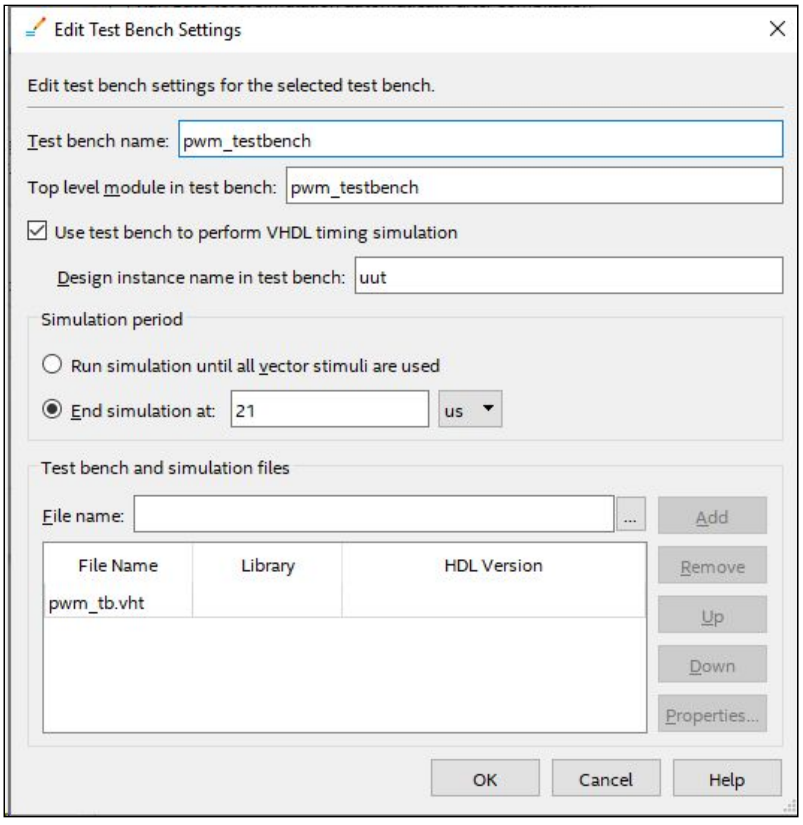
1) Debéis recuperar el proyecto a partir del archivo proporcionado (en Quartus Prime, ir a *Project -> Restore Archived Project*).
2) Describe detalladamente el funcionamiento del proceso “cnt_pr” incluido en el código VHDL del modulador PWM sencillo.
3) Compilar el diseño sobre una FPGA de Altera de la familia Cyclone IV E ejecutando *Processing → Start Compilation*. **Se deben mostrar los resultados de ocupación y explicar brevemente los elementos lógicos, número de pines, etc.**
4) Responde a las siguientes preguntas sobre el código VHDL del banco de pruebas del modulador PWM: ¿cuál es la función realizada por el proceso “clk_process”? ¿qué funciones realiza el proceso “stimuli”?
5) El proyecto incluye un banco de pruebas preparado para comprobar el funcionamiento del diseño mediante simulación con *Modelsim-Altera Starter Edition*. Podemos comprobar la configuración de la simulación desde la ventana *Tasks* en la izquierda, seleccionando en el desplegable *RTL Simulation* y entonces la tarea *RTL Simulation → Edit Settings.* Debería quedar algo similar a:



Y seleccionando el botón *Test Benches…*



A continuación, seleccionar “pwm_testbench” y clickar en *Edit* para editar la configuración del banco de pruebas para comprobar la configuración. Debería obtenerse lo siguiente.



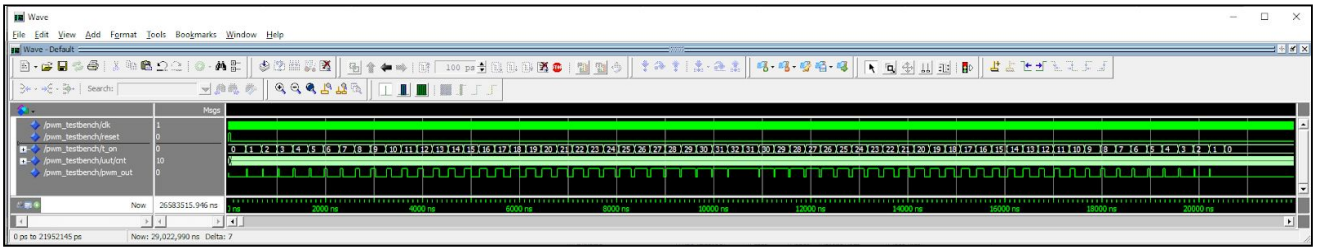
6) Cerramos todas las ventanas y lanzamos el ModelSim (clickar con botón derecho en *RTL Simulation* → clickar en *Start*). Automáticamente, se realizará la simulación y se abrirá una ventana Wave con las señales del PWM y del testbench: ‘clk’, ‘reset’, ‘t_on’ y ‘pwm_out’. A continuación, añadiremos la señal 'cnt' en ventana Wave para verificar el correcto funcionamiento del PWM. Para ello, en la ventana “sim - Default”, seleccionar “uut” para que aparezcan todas las señales internas del PWM en la ventana “Objects”. En la ventana “Objects”, seleccionar la señal “cnt”, clickar con botón derecho y seleccionar “Add Wave”.

Finalmente, lanzar de nuevo la simulación ejecutando los siguientes comandos en la consola (ventana “Transcript”) de ModelSim:

> restart

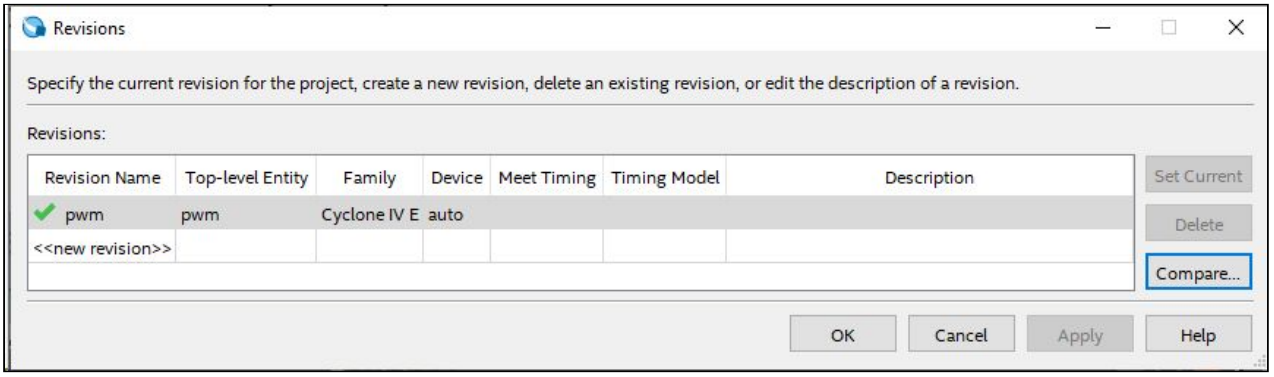
> run 21 us

Se debe explicar el resultado de la simulación.

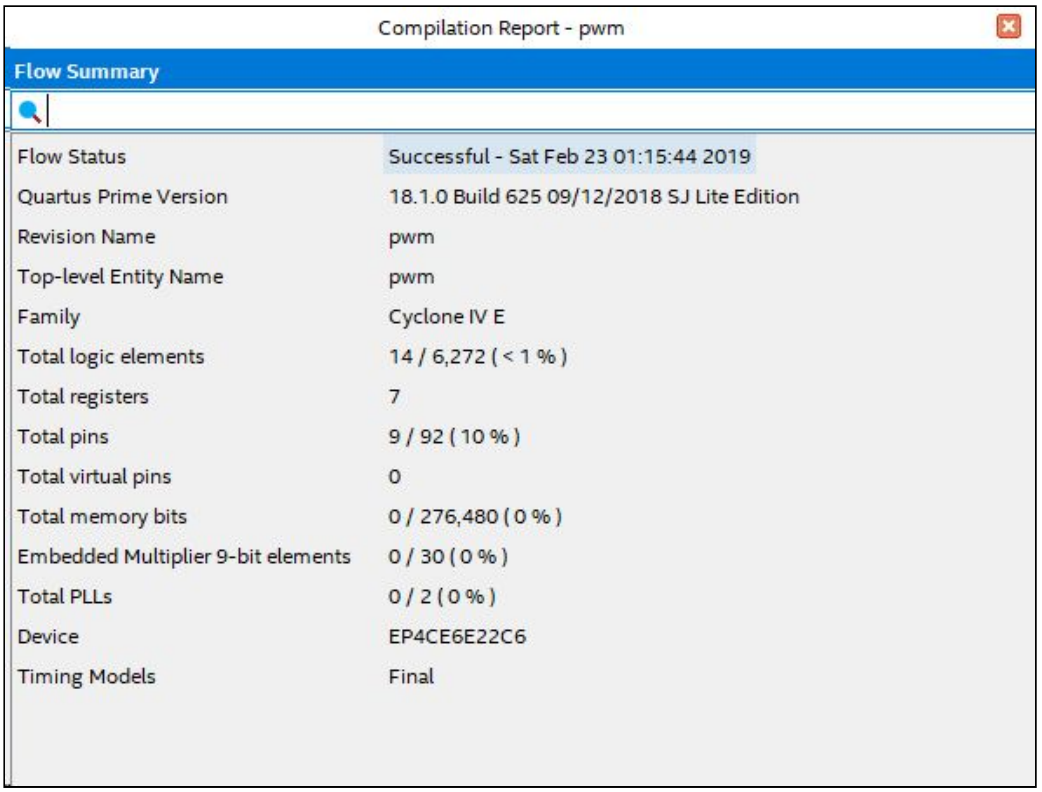


Solución:

1) Recuperamos el proyecto siguiendo las instrucciones. Podemos ver que se ha cargado correctamente con *Project* → *Revisions*:



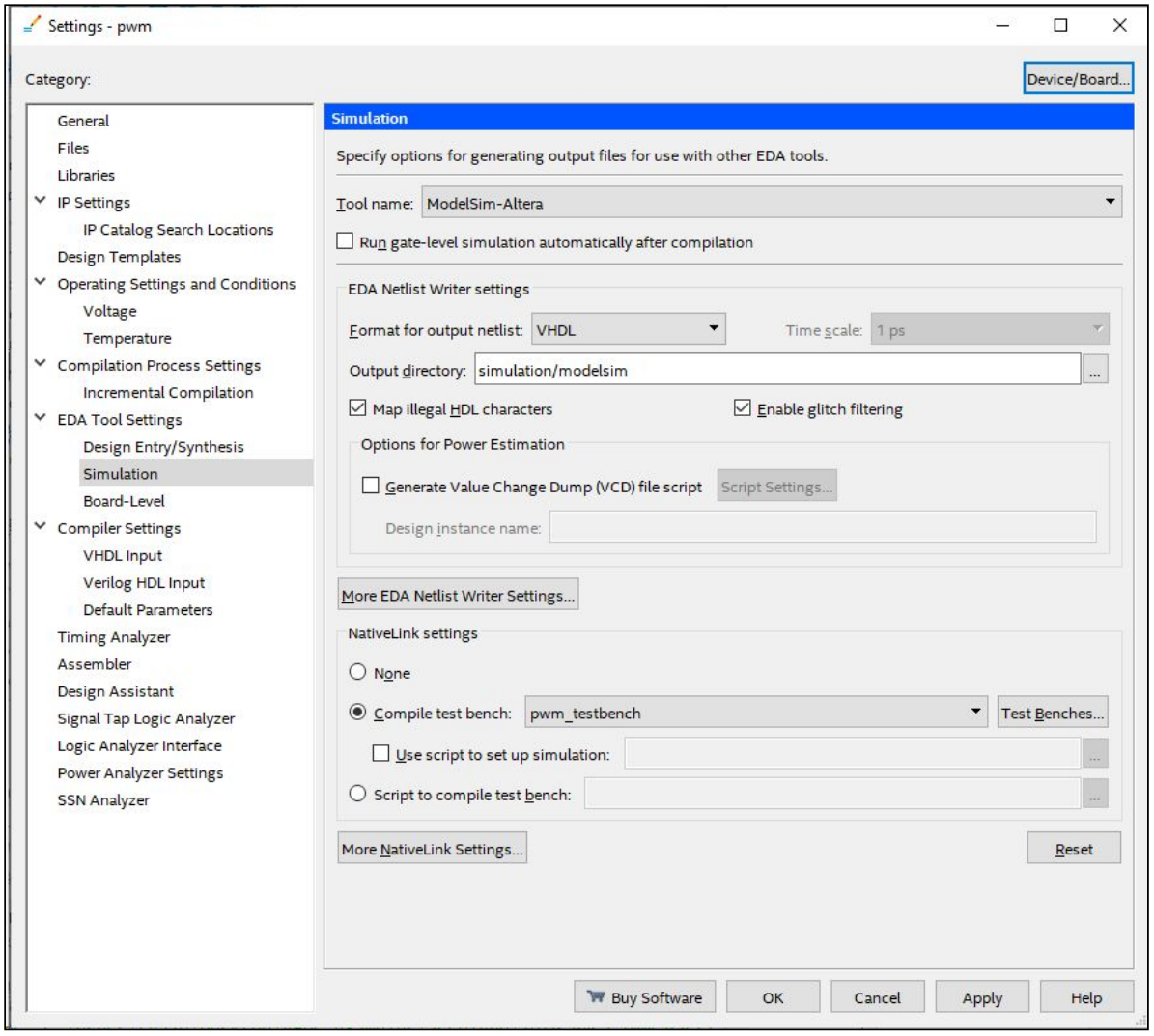
2) El funcionamiento del proceso “cnt_pr” incluido en el modulador PWM se describe a continuación. El proceso “cnt_pr” tiene dos señales de salida (“cnt”, “pwm_out”) y dos señales de entrada (“clk”, “rst”). Si la señal de entrada “rst” se activa a “1”, entonces las salidas “cnt” y “pwm_out” se resetean a 0 (valor en decimal) y “0” lógico, respectivamente. Si la entrada “rst” está a nivel lógico “0” y se produce un flanco de subida en la señal de clock “clk”, entonces ocurre lo siguiente:
   - Si “cnt” ha llegado al valor CNT_MAX, entonces “cnt” se resetea a 0 (valor en decimal). Si el valor de “cnt” es inferior a CNT_MAX, entonces “cnt” se incrementa en 1.
   - Si el valor de “cnt” es inferior a “t_on”, entonces a “pwm_out” se le asigna “1”.Si el valor de “cnt” es mayor o igual a “t_on”, entonces a “pwm_out” se le asigna “0”.
3) A continuación, compilamos el diseño del PWM con *Processing* → *Start Compilation*. El resultado será similar a:



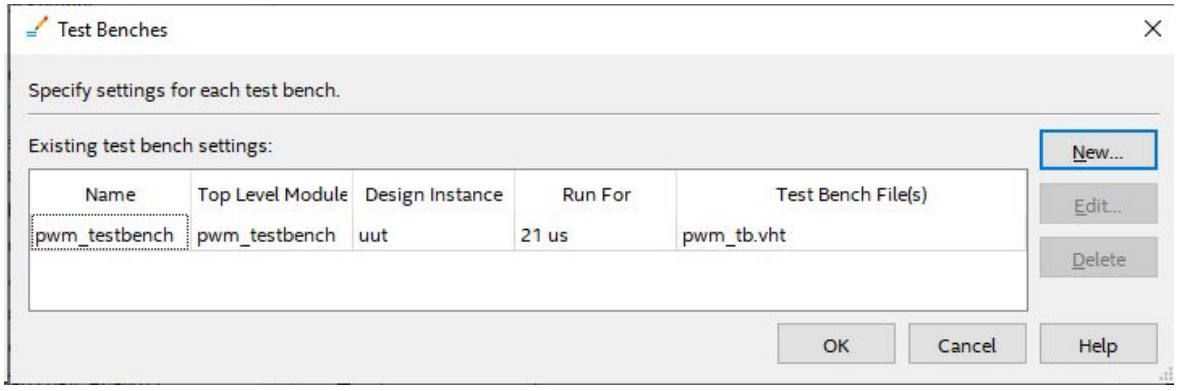
Como se puede comprobar, el diseño necesita 7 registros, además de la correspondiente lógica combinacional (14 elementos) y se ha implementado sobre un dispositivo de la familia Cyclone IV E. Se necesitan 9 pines.

4) En el código VHDL del banco de pruebas del modulador PWM, la función realizada por el proceso “clk_process” consiste en generar una señal de clock (“clk”) con un periodo de 10 ns definido en la constante “clk_period”. Periódicamente durante la duración de toda la simulación, la señal “clk” se fija a “0” durante un semiperiodo de 5 ns, y se fija a “1” durante el siguiente semiperiodo de 5 ns. El proceso “stimuli” genera los niveles requeridos en las señales “reset” y “t_on” para verificar el funcionamiento del PWM. En primer lugar, fija la señal “reset” a nivel lógico “1” durante 50 ns para resetear el sistema. Una vez desactivada la señal de “reset”, el proceso “stimuli” incrementa el valor de “t_on” de 1 en 1 desde 0 a 32. Cada valor de “t_on” se mantiene durante 32 ciclos de reloj “clk”. Nótese que el periodo de la señal de salida del PWM (“pwm_out”) dura 32 ciclos de reloj. Finalmente, el proceso “stimuli” decrementa el valor de “t_on” de 1 en 1 desde 32 a 0, manteniendo cada valor de “t_on” durante 32 ciclos de reloj. Esta secuencia en la generación de estímulos en las señales de entrada al PWM permiten verificar su funcionamiento para todos los valores posibles de “t_on”.
5) A continuación, comprobamos que el banco de pruebas ya está preparado. Siguiendo los pasos indicados en la documentación para editar un nuevo banco de pruebas, podemos comprobar la configuración de lo que tenemos incluido en el proyecto desde la ventana

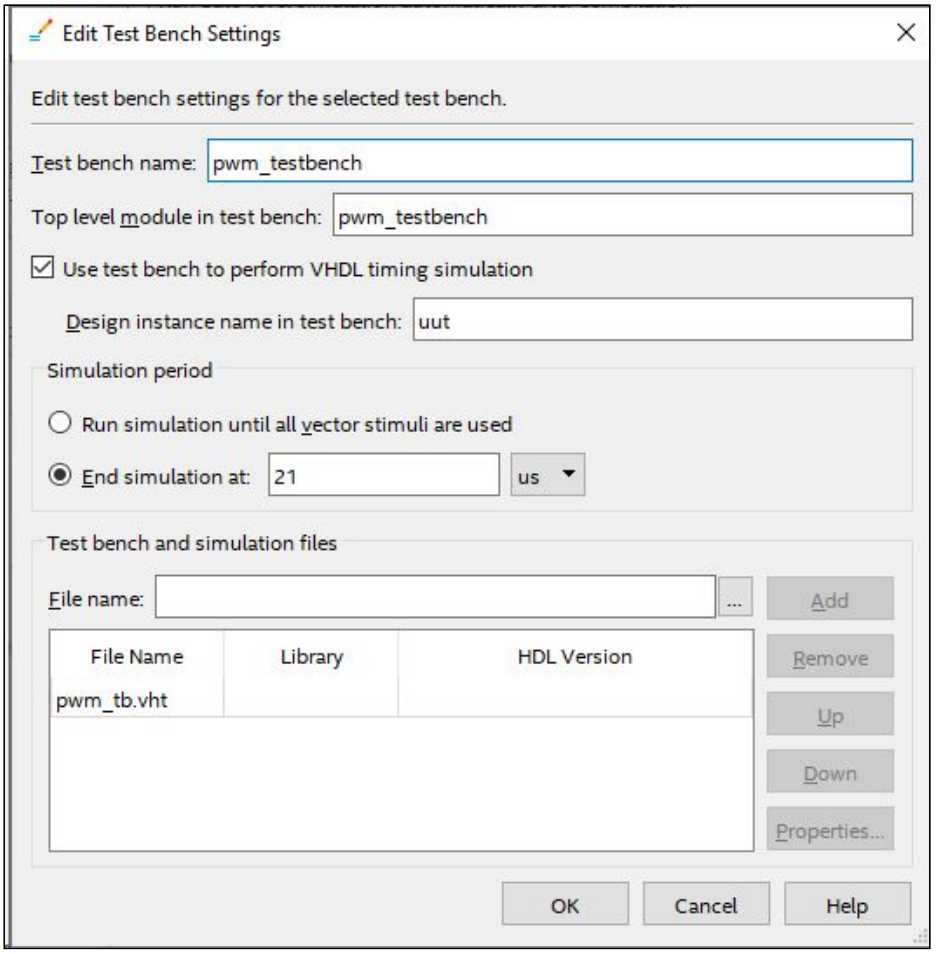
Tasks en la izquierda, seleccionando en el desplegable *Gate Level Simulation* y entonces la tarea *Gate-level Simulation → Edit Settings.* Debería quedar algo similar a:



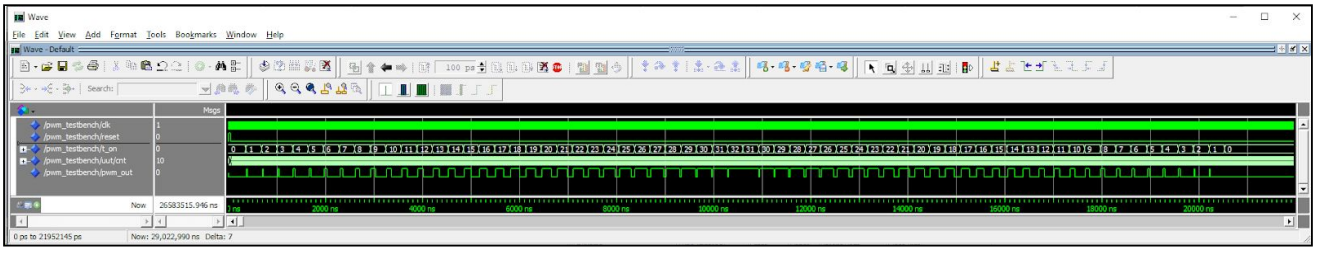
Y seleccionando el botón *Test Benches…*



A continuación editamos la configuración del banco de pruebas para comprobar la configuración:



6) Finalmente, cerramos todas las ventanas y simulamos (con *Gate Level Simulation → Start*) para obtener algo similar a las formas de onda de la figura siguiente.



Como se puede observar, el valor del tiempo de ON (t_on) se va incrementando en una unidad (desde 0 a 32) en cada ciclo (o período) de la señal PWM de salida (pwm_out), y luego se va decrementando en una unidad (desde 32 a 0) en cada ciclo de la señal PWM. El

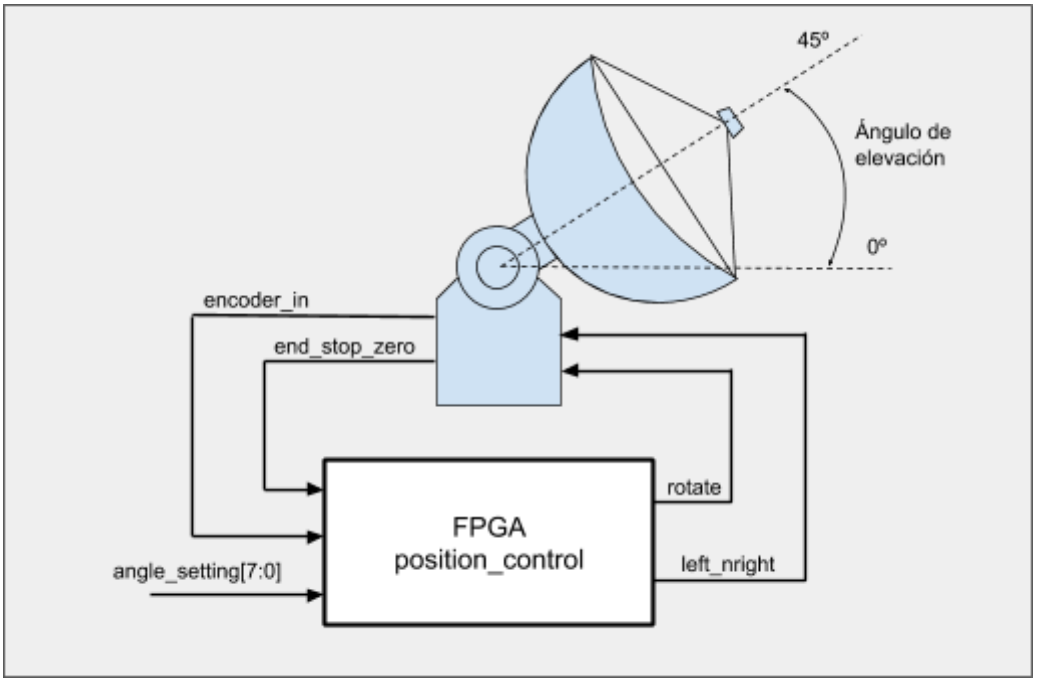
periodo de la señal pwm_out es fijo y el tiempo a nivel lógico "1" varía en cada ciclo de la señal según sea el valor de entrada t_on.

En la siguiente figura se muestra el detalle de la simulación para un ciclo de la señal PWM con t_on = 16, que equivale a un Duty Cycle del 50% ya que en cada ciclo o periodo de la señal PWM hay un total de 32 ciclos de reloj. Se puede ver cómo el contador 'cnt' es el elemento que controla el período de la señal PWM, incrementando en un unidad en cada flanco de subida de la señal de reloj, y contando de 0 a 31 de manera periódica.

Problema 2 (20%)

Los satélites LEO (Low Earth Orbit) orbitan alrededor de la tierra a una altura de entre 200 y 2000 km, tienen una velocidad relativa respecto a un punto fijo de la superficie terrestre, y pueden realizar una vuelta completa a la tierra en 90 minutos aproximadamente. En los sistemas de comunicaciones por satélite LEO se realiza un seguimiento de los satélites desde estaciones terrestres. Cada estación terrestre está equipada con una antena parabólica que puede ajustar el haz principal de su diagrama de radiación para "apuntar" y "seguir" a un satélite mientras atraviesa la zona de cobertura de la estación. La antena tiene dos motores que permiten ajustar el ángulo de elevación (0-180º) y el ángulo de azimut (0-360º) de la antena, respectivamente.

El objetivo de este problema consiste en el diseño de un sencillo sistema de control del ángulo de elevación de la antena de una estación terrestre. El sistema de control está formado por un **motor**, un **encoder incremental** para medir el desplazamiento angular, dos **finales de carrera** mecánicos para evitar que el motor gire fuera del rango entre 0º y 180º, un **detector de ángulo 0º**, y un **dispositivo de control de posición** basado en una FPGA. La siguiente figura muestra un diagrama de bloques del sistema con las señales de entrada/salida del dispositivo de control de posición: "encoder_in", "end_stop_zero", "angle_setting[7:0]", "rotate" y "left_nright".



El **motor** permanece parado cuando la señal "rotate" está a nivel lógico "0". El motor gira hacia la izquierda cuando la señal "rotate" está a nivel lógico "1" y la señal "left_nright" está a "1". El motor gira hacia la derecha cuando la señal "rotate" está a nivel lógico "1" y la señal "left_nright" está a "0". Cuando el motor gira hacia la derecha (en sentido horario), el ángulo de elevación disminuye hasta llegar al **final de carrera de 0º**. Si el motor llega al final de carrera de 0º, el **detector de ángulo 0º**

activa a nivel lógico “1” una señal denominada “end_stop_zero”. Cuando el motor gira hacia la izquierda (en sentido anti-horario), el ángulo de elevación aumenta hasta llegar al **final de carrera de 180º**.

El **encoder incremental** está formado por un disco codificado, que se desplaza unido al eje de la antena, y un cabezal detector que permanece fijo. El disco codificado está dividido en sectores y el cabezal produce un cambio en una señal digital de salida cada vez que el disco se desplaza un ángulo fijo (de 1º) entre dos sectores. La salida del encoder incremental es una señal periódica cuadrada, denominada “encoder_in”, cuya frecuencia depende de la velocidad de rotación del motor. Consideramos que la velocidad de rotación del motor es constante, por tanto, el periodo de la señal “encoder_in” es constante e igual al **tiempo que tarda el motor en realizar un desplazamiento angular de 1º**. Cuando el motor está parado, la señal “encoder_in” permanece fija a nivel lógico “0”.

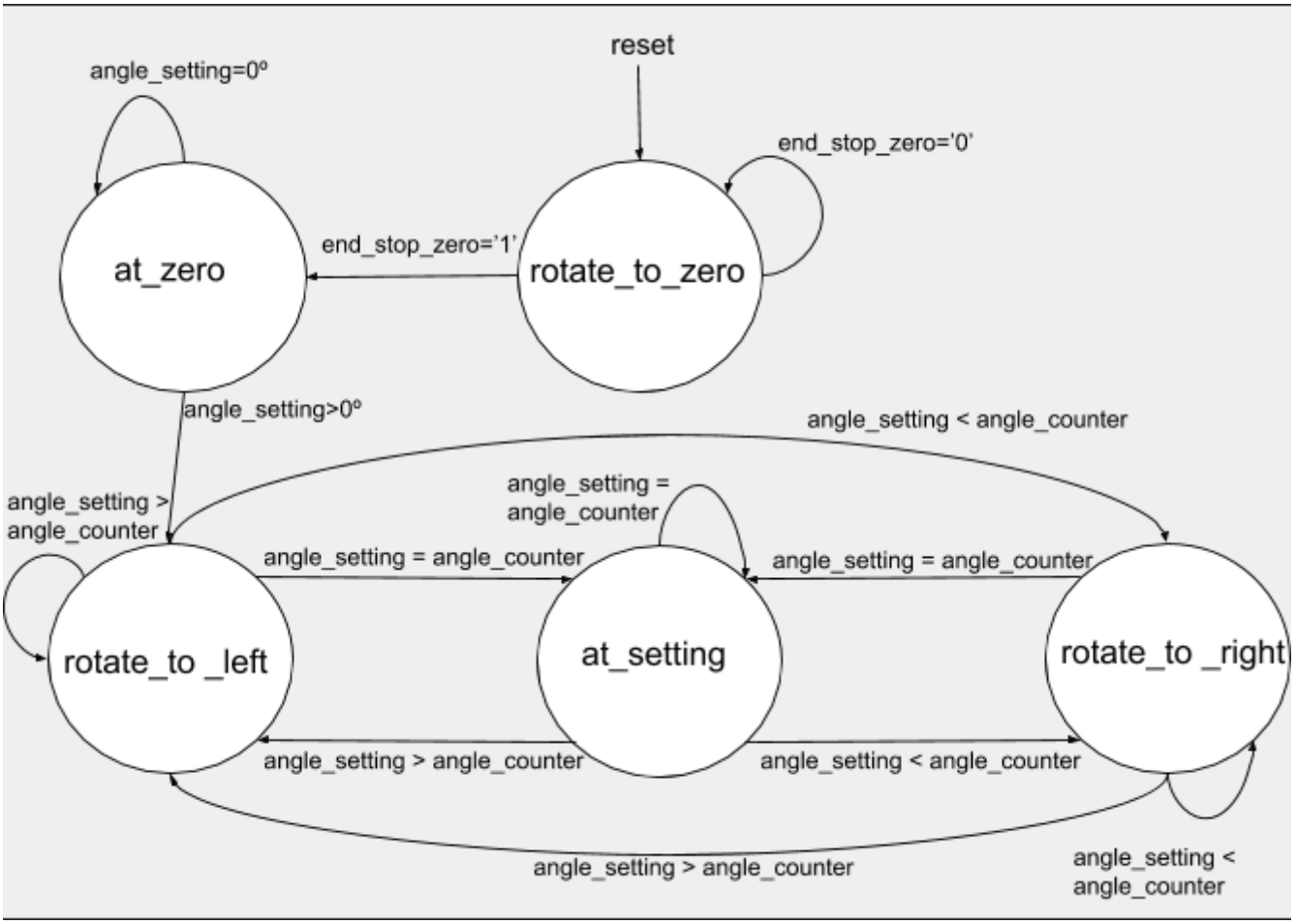
El **dispositivo de control de posición** basado en una FPGA se encarga de ajustar el ángulo de elevación al valor introducido en la entrada digital “angle_setting[7:0]” de 8 bits, que puede tomar valores entre 0º y 180º. El dispositivo de control de posición incluye un contador up/down binario (de 8 bits) cuya salida es el valor del ángulo de elevación actual, denominado “angle_counter[7:0]”. Cada vez que se introduce un nuevo valor en “angle_setting[7:0]”, el dispositivo de control de posición dará la orden de giro del motor en el sentido adecuado, mediante sus señales de salida “rotate” y “left_nright”, según sea el ángulo de elevación actual de la antena. Se pueden dar los siguientes casos:

- Si el valor de “angle_setting[7:0]” **es mayor que** “angle_counter[7:0]”, entonces **el motor debe girar hacia la izquierda** y la señal “angle_counter[7:0]” se incrementa en 1º en cada flanco de subida de la señal “encoder_in” procedente del encoder.
- Si el valor de “angle_setting[7:0]” **es menor que** “angle_counter[7:0]”, entonces **el motor debe girar hacia la derecha** y la señal “angle_counter[7:0]” se decrementa en 1º en cada flanco de subida de la señal “encoder_in” procedente del encoder.
- Si el valor de “angle_setting[7:0]” **es igual que** “angle_counter[7:0]”, entonces **el motor debe permanecer parado**.

Cuando se realiza un reset del dispositivo de control de posición o se pone en marcha, el valor del ángulo de elevación actual “angle_counter[7:0]” conmuta a 0º. Al utilizarse un encoder incremental, y no un encoder absoluto, cada vez que se da esta situación debe realizarse una **maniobra de inicialización del sistema** que consiste en girar el motor hacia la derecha hasta que el motor llega al final de carrera mecánico y se activa la salida “end_stop_zero” del detector de ángulo 0º.

El funcionamiento descrito del dispositivo de control de posición se representa en el siguiente diagrama de estados. Se trata de una **máquina de Moore**, que es un tipo de máquina de estados en la que el valor de las salidas sólo depende del estado actual, y no dependen del valor de las entradas. En las **máquinas de Mealy** las salidas dependen del estado actual y de las entradas.

Como puede observarse en el diagrama de estados, en cada flecha de transición se indica la condición necesaria para conmutar de un estado a otro o para permanecer en el mismo estado.



En cada estado, las salidas de la máquina de estados deben hacer lo siguiente:

- En el estado “rotate_to_zero”:
  - El motor gira a la derecha (“rotate”=“1” y “left_nright”=“0”).
  - La señal “angle_counter[7:0]” se resetea a 0º.
- En el estado “at_zero”:
  - El motor está parado (“rotate”=“0” y “left_nright”=“0”).
  - La señal “angle_counter[7:0]” se mantiene a 0º.
- En el estado “rotate_to_left”:
  - El motor gira a la izquierda (“rotate”=“1” y “left_nright”=“1”).
  - La señal “angle_counter[7:0]” se incrementa en 1º en cada flanco de subida de “encoder_in”.
- En el estado “rotate_to_right”:
  - El motor gira a la derecha (“rotate”=“1” y “left_nright”=“0”).
  - La señal “angle_counter[7:0]” se decrementa en 1º en cada flanco de “encoder_in”.

- En el estado “at_setting”:
  - El motor está parado (“rotate”=“0” y “left_nright”=“0”).
  - La señal “angle_counter[7:0]” se mantiene constante.

El siguiente listado (dividido en 3 partes) corresponde al código VHDL del dispositivo de control de posición basado en FPGA. El proceso “rising_edge_detector_pr” detecta los flancos de subida en la señal “encoder_in”, generando un pulso de 1 ciclo de reloj cada vez que se produce un flanco de subida. El proceso “fsm_pr” describe las condiciones para cambiar de un estado a otro y el proceso “fsm_outputs” describe la asignación de valores a las salidas de la máquina. El proceso “counter_pr” describe el contador up/down binario del ángulo de elevación.

Como se puede comprobar, la descripción VHDL de los procesos “fsm_pr”, “fsm_outputs” y “counter_pr” está incompleta en los espacios donde hay comentarios que indican “Completar aquí”. En este problema se pide completar el diseño VHDL y verificar el funcionamiento mediante simulación.

Listado VHDL del fichero "position_control.vhd" (parte 1 de 3)

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.numeric_std.all;
 4
 5  entity position_control is
 6  port (
 7      clk               : in  std_logic;      -- Clock signal
 8      reset             : in  std_logic;      -- Reset signal
 9      encoder_in        : in  std_logic;      -- Encoder signal
10      end_stop_zero     : in  std_logic;      -- End Stop 0º signal
11      angle_setting     : in  std_logic_vector(7 downto 0); -- Angular position setting
12      rotate            : out std_logic;      -- Rotate motor
13      left_nright       : out std_logic       -- 1 = left, 0 = right
14      );
15  end position_control;
16
17  architecture rtl of position_control is
18
19      type state_type is (rotate_to_zero, at_zero, rotate_to_left, rotate_to_right, at_setting);
20
21      signal state            : state_type;
22
23      signal angle_counter : unsigned(7 downto 0); -- Counter of current angle
24      signal counter_en       : std_logic;         -- Enable counter
25      signal counter_up_ndown : std_logic;         -- 1 = count up, 0 = count down
26      signal counter_rst      : std_logic;         -- Reset counter
27
28      signal in_rise          : std_logic;         -- Rising edge of signal "encoder_in"
29      signal in_delayed       : std_logic;         -- Signal "encoder_in" delayed 1 clock
30
31  begin
32
33  -- Detector of rising edges of signal "encoder_in"
34  rising_edge_detector_pr : process(clk, reset)
35  begin
36      if(reset ='1') then
37          in_rise  <= '0';
38          in_delayed  <= '0';
39      elsif(rising_edge(clk)) then
40          in_rise  <= not in_delayed and encoder_in;
41          in_delayed  <= encoder_in;
42      end if;
43  end process rising_edge_detector_pr;
44
```

Listado VHDL del fichero "position_control.vhd" (parte 2 de 3)

```
44
45  -- Finite states machine: conditions to switch between states
46  fsm_pr: process (clk, reset)
47  begin
48     if (reset ='1') then
49        --------------------
50        -- Completar aquí
51        --------------------
52     elsif (clk'event and clk = '1') then
53        case state is
54           when rotate_to_zero =>
55              --------------------
56              -- Completar aquí
57              --------------------
58           when at_zero =>
59              --------------------
60              -- Completar aquí
61              --------------------
62           when rotate_to_left =>
63              --------------------
64              -- Completar aquí
65              --------------------
66           when rotate_to_right =>
67              --------------------
68              -- Completar aquí
69              --------------------
70           when at_setting =>
71              --------------------
72              -- Completar aquí
73              --------------------
74           when others =>
75              state <= rotate_to_zero;
76
77        end case;
78     end if;
79  end process fsm_pr;
80
81  -- Finite states machine: outputs in each state
82  fsm_outputs: process (state)
83  begin
84     case state is
85        when rotate_to_zero =>
86              --------------------
87              -- Completar aquí
88              --------------------
89        when at_zero =>
90              --------------------
91              -- Completar aquí
92              --------------------
93        when rotate_to_left =>
94              --------------------
95              -- Completar aquí
96              --------------------
97        when rotate_to_right =>
98              --------------------
99              -- Completar aquí
100             --------------------
101       when at_setting =>
102             --------------------
103             -- Completar aquí
104             --------------------
105       when others =>          rotate <= '0';
106                               left_nright <= '0';
107                               counter_en <= '0';
108                               counter_up_ndown <= '0';
109                               counter_rst <= '0';
110    end case;
111 end process fsm_outputs;
112
```

Listado VHDL del fichero "position_control.vhd" (parte 3 de 3)

```
113   -- Up/down binary counter of current position (1 count step = 1º)
114   counter_pr : process(clk, reset)
115   begin
116      if (reset = '1') then
117         angle_counter <= (others => '0');
118      elsif (clk'event and clk = '1') then
119         if (counter_rst = '1') then
120            --------------------
121            -- Completar aquí
122            --------------------
123         elsif (counter_en = '1' and in_rise = '1') then
124            if (counter_up_ndown = '0') then
125               --------------------
126               -- Completar aquí
127               --------------------
128            else
129               --------------------
130               -- Completar aquí
131               --------------------
132            end if;
133         end if;
134      end if;
135   end process counter_pr;
136
137   end rtl;
```

Junto con el enunciado de la PEC, se os ha proporcionado un archivo llamado "2019_2_PAC2_PositionControl.qar" que corresponde al proyecto a partir del cual se debe completar el diseño del código del "position_control.vhd". Con él se pide hacer lo siguiente:

1) Recuperar el proyecto a partir del archivo proporcionado (en Quartus Prime, ir a *Project -> Restore Archived Project*).
2) Completar el fichero "position_control.vhd" con el código que falta en la descripción de la máquina de estados en el proceso “fsm_pr”. Las **condiciones para cambiar de un estado a otro** se detallan en el diagrama de estados presentado antes.
3) Completar el fichero "position_control.vhd" con el código que falta en la descripción de **las salidas de la máquina de estados** en el proceso “fsm_outputs”. Deben asignarse los valores adecuados a las salidas (rotate, left_nright, counter_en, counter_up_ndown, counter_rst) en cada uno de los estados.
4) Completar el fichero "position_control.vhd" con el código que falta en la descripción del **contador up/down binario** en el proceso “counter_pr”. Deben asignarse los valores adecuados a la salida del contador (angle_counter) para cada una de las condiciones.
5) Compilar el diseño sobre una FPGA de Altera de la familia Cyclone IV E ejecutando *Processing → Start Compilation*. Se deben mostrar los resultados de ocupación y explicar brevemente los elementos lógicos, número de pines, etc.
6) El proyecto incluye un banco de pruebas (fichero “position_control_tb.vht”) que debe completarse para verificar el funcionamiento del diseño del "position_control.vhd" mediante simulación RTL con *Modelsim-Altera Starter Edition*. **Se pide añadir el código que falta en el fichero “position_control_tb.vht”** para verificar los **4 test cases** siguientes:
    a) Maniobra de inicialización a 0º después del reset.
    b) Ajuste del ángulo de elevación a 90º.
    c) Ajuste del ángulo de elevación a 180º.
    d) Ajuste del ángulo de elevación a 45º.

7) **Lanzar la simulación con ModelSim** (en Tasks, seleccionar *RTL Simulation* → clickar con botón derecho en *RTL Simulation* → clickar en *Start*). Si es necesario añadir más señales a la ventana Wave, en la ventana "sim - Default" debe seleccionarse "uut" para que aparezcan todas las señales internas del "position_control" en la ventana "Objects". En la ventana "Objects", seleccionar todas las señales deseadas, clickar con botón derecho y seleccionar "Add Wave". Finalmente, lanzar de nuevo la simulación ejecutando los siguientes comandos en la consola (ventana "Transcript") de ModelSim:

    > restart
    > run 40 us

ATENCIÓN: Se debe explicar el resultado de la simulación de cada test case.

Solución:

1) Recuperamos el proyecto siguiendo las instrucciones.

2) En este apartado se pide completar el código VHDL del proceso “fsm_pr” con las condiciones necesarias para cambiar de un estado a otro. El código del proceso "fsm_pr" podría quedar como el que se muestra en el siguiente listado. Tal como se puede observar, las condiciones para cambiar de estado coincides con las que se detallan en el enunciado.

Listado del código VHDL del proceso "fsm_pr":

```
-- Finite states machine: conditions to switch between states
fsm_pr: process (clk, reset)
begin
   if (reset ='1') then
      state <= rotate_to_zero;

   elsif (clk'event and clk = '1') then
      case state is
         when rotate_to_zero =>
            if (end_stop_zero = '1') then
               state <= at_zero;
            else
               state <= rotate_to_zero;
            end if;

         when at_zero =>
            if (unsigned(angle_setting) > 0) then
               state <= rotate_to_left;
            else
               state <= at_zero;
            end if;

         when rotate_to_left =>
            if (angle_counter = unsigned(angle_setting)) then
               state <= at_setting;
            elsif (angle_counter < unsigned(angle_setting)) then
               state <= rotate_to_left;
            else
               state <= rotate_to_right;
            end if;

         when rotate_to_right =>
            if (angle_counter = unsigned(angle_setting)) then
               state <= at_setting;
            elsif (angle_counter < unsigned(angle_setting)) then
               state <= rotate_to_left;
            else
               state <= rotate_to_right;
            end if;

         when at_setting =>
            if (angle_counter = unsigned(angle_setting)) then
               state <= at_setting;
            elsif (angle_counter > unsigned(angle_setting)) then
               state <= rotate_to_right;
            else
               state <= rotate_to_left;
            end if;

         when others =>
            state <= rotate_to_zero;

      end case;
   end if;
end process fsm_pr;
```

3) En este apartado se pide completar el código VHDL del proceso “fsm_outputs” con la asignación de valores a las salidas (rotate, left_nright, counter_en, counter_up_ndown, counter_rst) en cada uno de los estados. El código del proceso "fsm_outputs" podría quedar como el que se muestra en el siguiente listado.

Listado del código VHDL del proceso "fsm_outputs":

```
101   -- Finite states machine: outputs in each state
102   fsm_outputs: process (state)
103   begin
104      case state is
105         when rotate_to_zero =>  rotate <= '1';
106                                 left_nright <= '0';
107                                 counter_en <= '0';
108                                 counter_up_ndown <= '0';
109                                 counter_rst <= '1';
110
111         when at_zero =>         rotate <= '0';
112                                 left_nright <= '0';
113                                 counter_en <= '0';
114                                 counter_up_ndown <= '0';
115                                 counter_rst <= '0';
116
117         when rotate_to_left =>  rotate <= '1';
118                                 left_nright <= '1';
119                                 counter_en <= '1';
120                                 counter_up_ndown <= '1';
121                                 counter_rst <= '0';
122
123         when rotate_to_right => rotate <= '1';
124                                 left_nright <= '0';
125                                 counter_en <= '1';
126                                 counter_up_ndown <= '0';
127                                 counter_rst <= '0';
128
129         when at_setting =>      rotate <= '0';
130                                 left_nright <= '0';
131                                 counter_en <= '0';
132                                 counter_up_ndown <= '0';
133                                 counter_rst <= '0';
134
135         when others =>          rotate <= '0';
136                                 left_nright <= '0';
137                                 counter_en <= '0';
138                                 counter_up_ndown <= '0';
139                                 counter_rst <= '0';
140      end case;
141   end process fsm_outputs;
142
```

4) En este apartado se pide completar el código VHDL del proceso “counter_pr” asignando los valores adecuados a la salida del contador (angle_counter) para cada una de las condiciones. El código podría quedar como el que se muestra en el siguiente listado.

Listado del código VHDL del proceso "counter_pr":

```
143   -- Up/down binary counter of current position (1 count step = 1º)
144   counter_pr : process(clk, reset)
145   begin
146      if (reset = '1') then
147         angle_counter <= (others => '0');
148      elsif (clk'event and clk = '1') then
149         if (counter_rst = '1') then
150            angle_counter <= (others => '0');
151         elsif (counter_en = '1' and in_rise = '1') then
152            if (counter_up_ndown = '0') then
153               angle_counter <= angle_counter - 1;
154            else
155               angle_counter <= angle_counter + 1;
156            end if;
157         end if;
158      end if;
159   end process counter_pr;
160
161   end rtl;
```

5) Compilamos y anotamos los resultados obtenidos:

Compilation Report - position_control

Flow Summary

<<Filter>>

Flow Status	Successful - Sat Mar 28 19:14:35 2020
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	position_control
Top-level Entity Name	position_control
Family	Cyclone IV E
Total logic elements	45 / 6,272 (< 1 %)
Total registers	15
Total pins	14 / 92 (15 %)
Total virtual pins	0
Total memory bits	0 / 276,480 (0 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)
Device	EP4CE6E22C6
Timing Models	Final

Ninguna sorpresa respecto el número de pines: 14 (es la suma exacta de los que tenemos definidos en la entidad). El número de registros (15 flip-flops) es también lo que toca: 8 registros para el contador, 4 para las salidas de la máquina de estados y 3 más para

codificar los estados de la máquina. Por otro lado, los elementos lógicos se elevan a 45, ya que tenemos diversos comparadores del valor de “angle_counter” respecto a “angle_setting”, 2 sumadores de 8 bits en el contador, además de la lógica de control.

6) El listado de un posible banco de pruebas para verificar los 4 test cases se muestra a continuación.

Listado del banco de pruebas “position_control_tb.vht” (1 de 2):

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

entity position_control_testbench IS
end position_control_testbench;

architecture behavior of position_control_testbench is

    -- Unit under test.
    component position_control
        port(
            clk               : in  std_logic;      -- Clock signal
            reset             : in  std_logic;      -- Reset signal
            encoder_in        : in  std_logic;      -- Encoder signal
            end_stop_zero     : in  std_logic;      -- End Stop 0º signal
            angle_setting     : in  std_logic_vector(7 downto 0); -- Angular position setting
            rotate            : out std_logic;      -- Rotate motor
            left_nright       : out std_logic       -- 1 = left, 0 = right
        );
    end component;

    signal clk                : std_logic := '0';
    signal reset              : std_logic := '0';
    signal encoder_in         : std_logic := '0';
    signal end_stop_zero      : std_logic := '0';
    signal angle_setting      : unsigned (7 downto 0) := (others => '0');

    -- Clock definition
    constant clk_period : time := 10 ns;
    constant encoder_in_period : time := 100 ns;

begin
    -- Instance of the unit under test.
    uut: position_control PORT MAP (
        clk => clk,
        reset => reset,
        encoder_in => encoder_in,
        end_stop_zero => end_stop_zero,
        angle_setting => std_logic_vector(angle_setting),
        rotate => open,
        left_nright => open
        );

    -- Definition of the clock process.
    clk_process :process
    begin
        clk <= '1';
        wait for clk_period/2;
        clk <= '0';
        wait for clk_period/2;
    end process;

    -- Definition of the encoder signal process.
    encoder_in_process :process
    begin
        encoder_in <= '1';
        wait for encoder_in_period/2;
        encoder_in <= '0';
        wait for encoder_in_period/2;
    end process;
```

Listado del banco de pruebas “position_control_tb.vht” (2 de 2):

```
62
63      -- Stimuli process.
64      stimuli: process
65      begin
66          -- Reset circuitry
67          reset <= '1';
68          wait for clk_period * 2;
69          reset <= '0';
70
71          wait for 1 ps;
72
73          -- Test case 1: initialization (rotating to position 0°)
74          wait for 10 * clk_period;
75          end_stop_zero <= '1';
76          wait for clk_period;
77          end_stop_zero <= '0';
78
79          -- Test case 2: angular position setting = 90 (rotate to right from 0° to 90°)
80          angle_setting <= to_unsigned(90, 8);
81          wait for 90 * encoder_in_period;
82
83          wait for encoder_in_period * 2;
84
85
86          -- Test case 3: angular position setting = 180 (rotate to left from 90° to 180°)
87          angle_setting <= to_unsigned(180, 8);
88          wait for 90 * encoder_in_period;
89
90          wait for encoder_in_period * 2;
91
92          -- Test case 4: angular position setting = 45 (rotate to right from 180° to 45)
93          angle_setting <= to_unsigned(45, 8);
94          wait for (180 - 45) * encoder_in_period;
95
96          wait;
97      end process;
98  end;
99
```

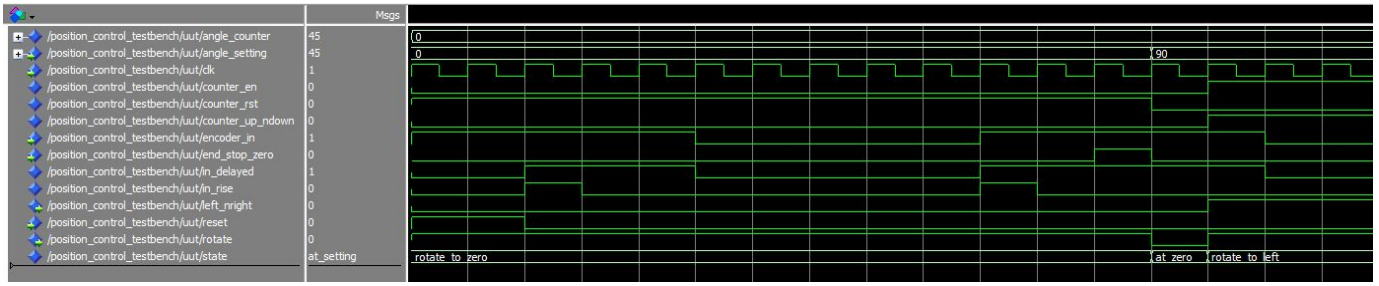
Tal como se puede observar en el banco de pruebas:

- Se declara el componente a verificar (position_control) [líneas 11-21] y la instancia [36-43] utilizando las señales internas del banco de pruebas declaradas previamente [23-27].
- Se genera la señal de reloj en el proceso "clk_process" [líneas 46-52]. La señal de reloj se inicializa en la línea 23 y utiliza un período de 10 ns definido en la constante "clk_period" [línea 30].
- Se genera la señal encoder_in en el proceso "encoder_in_process" [líneas 55-61]. La señal encoder_in se inicializa en la línea 25 y utiliza un período de 100 ns definido en la constante "encoder_in_period" [línea 31]. Por tanto, la señal encoder_in es una señal cuadrada con periodo constante.
- Se generan los estímulos de entrada necesarios para hacer la comprobación del diseño en el proceso "estimuli" [líneas 64-97] en los 4 test cases.

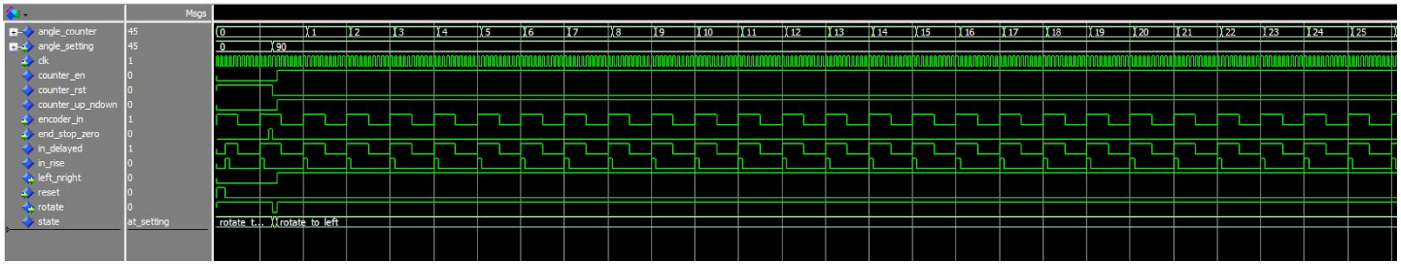
Tal como se ha comentado, el proceso "stimuli" es el que se encarga de ir generando los estímulos que se aplicarán a las entradas del módulo "position_control" que estamos verificando. En este caso, no se hace la comprobación explícita de la salida en este mismo proceso y, por tanto, será el/la diseñador/a quien tendrá que hacer la verificación visual de las formas de onda obtenidas.

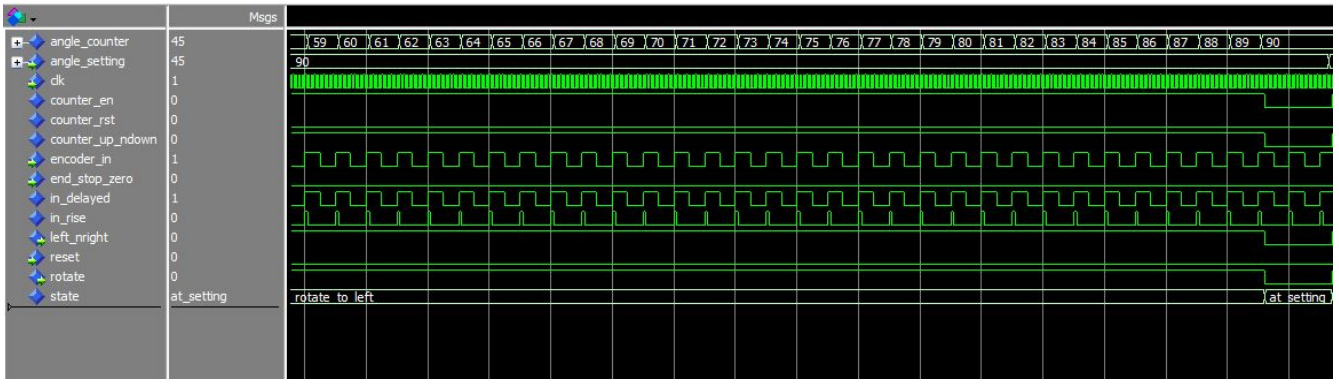
7) A continuación, comprobaremos el resultado de la simulación con RTL Simulation. Es importante destacar que el hecho de utilizar una simulación tipo RTL permite observar todas las señales internas, lo cual ayuda mucho en caso de tener que corregir errores de diseño.

Resultado de la simulación del test case 1 (Maniobra de inicialización a 0º después del reset): Como puede observarse en la siguiente forma de onda de la simulación, el periodo de la señal "encoder_in" es constante y de 100 ns. Además, se puede ver el funcionamiento del circuito de detección de flancos de subida en la señal "encoder_in", cuando llega un flanco "encoder_in" se produce un pulso en la señal “in_rise”. Después del reset, el estado de la máquina es “rotate_to_zero”. A continuación, 10 ciclos de reloj más tarde, se activa la señal “end_stop_zero” para indicar que el motor ha llegado al final de carrera de 0º, y cuando eso ocurre, la máquina conmuta al estado “at_zero”. En ese estado se activa a nivel “1” la señal “counter_rst” para fijar el valor de “angle_counter” a 0º.

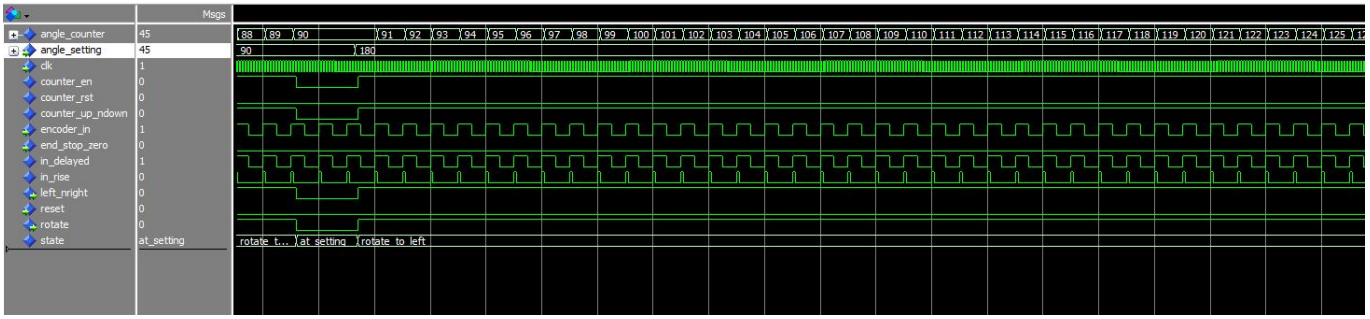


Resultado de simulación del test case 2 (Ajuste del ángulo de elevación a 90º): Una vez que se ha realizado la maniobra de inicialización a 0º, se introduce un valor de “angle_setting=90º” para que el motor gire hasta llegar al ángulo de 90º. Como el valor de “angle_setting” es mayor que el de “angle_counter=0º”, la máquina conmuta al estado “rotate_to_left”, activándose a nivel “1” la señal “left_nright” y la señal “rotate” para girar el motor hacia la izquierda. Además, se habilita el contador activando a “1” la señal “counter_en” y hacia arriba (up) activando a “1” la señal “counter_up_ndown”. En cada flanco de subida de la señal “encode_in”, se genera un pulso en “in_rise” que incrementa el valor de “angle_counter” en 1 unidad. Cuando el valor de “angle_counter” llega a 90º, la máquina pasa al estado “at_setting” y se ponen a “0” las señales “rotate” y “counter_en” para inhabilitar el movimiento del motor y el contador.

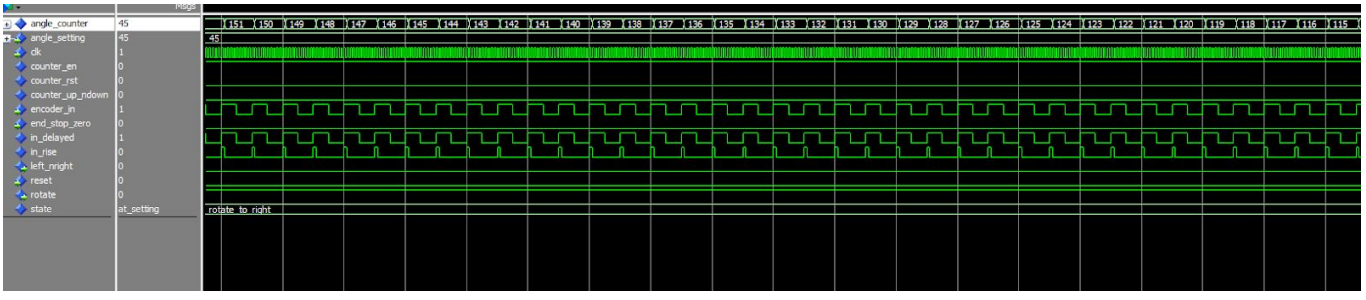




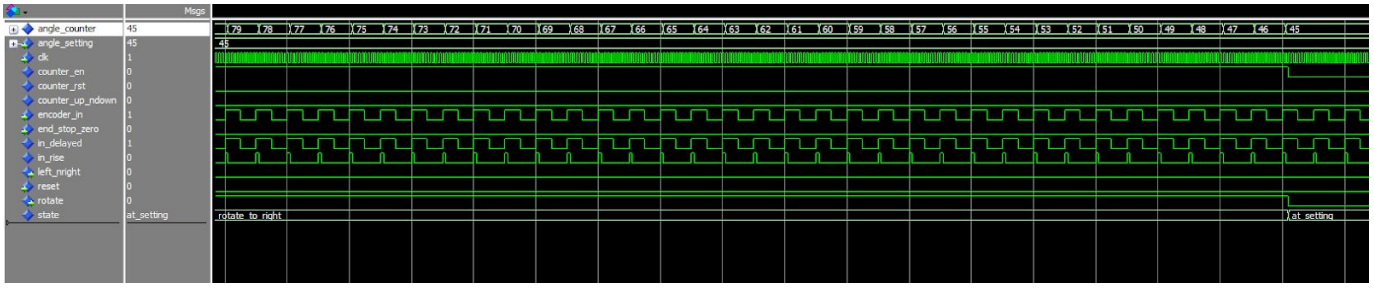
Resultado de simulación del test case 3 (Ajuste del ángulo de elevación a 180º): Una vez que motor ha llegado al ángulo de 90º, se introduce un nuevo valor de "angle_setting=180º" para que el motor gire hasta llegar al ángulo de 180º. Como el valor de "angle_setting" es mayor que el de "angle_counter=90º", la máquina conmuta de nuevo al estado "rotate_to_left", activándose a nivel "1" la señal "left_nright" y la señal "rotate" para girar el motor hacia la izquierda. Además, se habilita el contador activando a "1" la señal "counter_en" y hacia arriba (up) activando a "1" la señal "counter_up_ndown". En cada flanco de subida de la señal "encode_in", se genera un pulso en "in_rise" que incrementa el valor de "angle_counter" en 1 unidad. Cuando el valor de "angle_counter" llega a 180º, la máquina pasa al estado "at_setting" y se ponen a "0" las señales "rotate" y "counter_en" para inhabilitar el movimiento del motor y el contador.



Resultado de simulación del test case 4 (Ajuste del ángulo de elevación a 45º): Una vez que motor ha llegado al ángulo de 180º, se introduce un nuevo valor de "angle_setting=45º" para que el motor gire hasta llegar al ángulo de 45º. Como el valor de "angle_setting" es menor que el de "angle_counter=180º", la máquina conmuta al estado "rotate_to_right", activándose a nivel "0" la señal "left_nright" y a nivel "1" la señal "rotate" para girar el motor hacia la derecha. Además, se habilita el contador activando a "1" la señal "counter_en" y hacia abajo (down) poniendo "0" en la señal "counter_up_ndown". En cada flanco de subida de la señal "encode_in", se genera un pulso en "in_rise" que decrementa el valor de "angle_counter" en 1 unidad. Cuando el valor de "angle_counter" llega a 45º, la máquina pasa al estado "at_setting" y se ponen a "0" las señales "rotate" y "counter_en" para inhabilitar el movimiento del motor y el contador.




Msgs
angle_counter
angle_setting
clk
counter_en
counter_rst
counter_up_ndown
encoder_in
end_stop_zero
in_delayed
in_rise
left_nright
reset
rotate
state
rotate_to_right


Msgs
angle_counter
angle_setting
clk
counter_en
counter_rst
counter_up_ndown
encoder_in
end_stop_zero
in_delayed
in_rise
left_nright
reset
rotate
state
rotate_to_right
at_setting

Problema 3 (20%)

Una máquina está equipada con un encoder incremental para medir el desplazamiento angular y la velocidad de rotación de un motor. El encoder genera un par de señales cuadradas idénticas en contra-fase, con una frecuencia (y período) que dependen de la velocidad de rotación. En este problema se propone implementar el código VHDL de un módulo denominado "period_measure" que permita medir el período de una de las señales de salida del encoder incremental. La definición de la entidad "period_measure" se muestra a continuación:

```
entity period_measure is
port (
    clk             : in  std_logic;      -- Clock signal
    rst             : in  std_logic;      -- Reset signal
    encoder_in      : in  std_logic;      -- Encoder signal
    period_len      : out std_logic_vector(N-1 downto 0)   -- Duration of period
    );
end period_measure;
```

Concretamente, el módulo "period_measure" debe cumplir los siguientes requerimientos:

- La medida del periodo de la señal "encoder_in" debe darse en número de ciclos de clock.
- La resolución temporal de la medida del periodo debe ser de 10 ns.
- La frecuencia de la señal de salida del encoder puede ser superior a 2 kHz.
- El módulo debe detectar un flanco de subida de la señal cuadrada procedente del encoder para empezar a medir el período de la señal. Después debe detectar el siguiente flanco de subida para guardar el valor del período medido y reiniciar la medida del siguiente periodo de la señal.
- El resultado de la medida del último período medido se debe almacenar en un registro de salida.

Junto con el enunciado de la PEC, se os ha proporcionado un archivo llamado "2019_PAC2_Speed_encoder.qar" que corresponde al proyecto a partir del cual se debe completar el diseño del módulo "period_measure". Con él se debe hacer lo siguiente:

1) Recuperar el proyecto a partir del archivo proporcionado (a Quartus Prime, ir a *Project -> Restore Archived Project*).
2) Modificar el archivo "period_measure.vhd" para incluir el código que falta para cumplir los requerimientos del módulo "period_measure".
3) Compilar el diseño sobre una FPGA de Altera de la familia Cyclone IV E. Se deben mostrar y explicar los resultados de ocupación (elementos lógicos, número de pines, etc).
4) El proyecto incluye un banco de pruebas (fichero “period_measure_tb.vht”) que debe completarse para verificar el funcionamiento del diseño del "period_measure" mediante simulación RTL con *Modelsim-Altera Starter Edition*. En particular, tenéis que verificar tres test cases con un periodo de la señal del encoder diferente para cada caso. **Se pide añadir el código que falta en el fichero “period_measure_tb.vht”** y **lanzar la simulación con ModelSim** (clickar con botón derecho en *RTL Simulation* → clickar en *Start*).

ATENCIÓN: Se debe explicar detalladamente el resultado de la simulación.

Solución:

1) Recuperamos el proyecto siguiendo las instrucciones.

2) Se nos pide implementar un módulo VHDL para medir el periodo de una señal cuadrada con una frecuencia superior a 2 kHz con una resolución de 10 ns. El código del módulo "period_measure" podría quedar como el que se muestra en el listado de la siguiente página.

   Como se puede ver en el listado, para medir la duración del período de la señal "encoder_in" utilizamos un contador, incluido en el proceso "count_period_pr", que funciona a la frecuencia del reloj de entrada (señal clk) y que cuenta el número de ciclos de reloj que hay dentro del período de la señal "encoder_in". Si se quiere conseguir una resolución de 10 ns, el ciclo de reloj debe ser de 10 ns y, por tanto, la frecuencia de clock debe ser de 1 / 10ns = 100 MHz. Para obtener el número de bits del contador (llamado N en la definición de la entidad) se debe calcular el número máximo de ciclos de reloj que se pueden contar cuando el periodo de la señal de entrada alcanza su valor máximo (es decir, cuando la frecuencia de la señal es mínima, 2 kHz). El número máximo de ciclos de reloj es 100 MHz / 2 kHz = 50.000 ciclos, que se puede codificar con N = 16 bits.

   Además del contador, en el diseño del módulo "period_measure" se incluye un proceso llamado "edge_detector_pr" que se encarga de la detección de flancos de subida (transición de nivel lógico "0" a nivel lógico "1") en la señal de entrada "encoder_in". Cuando se produce un flanco de subida en la señal "encoder_in", se genera un pulso en la señal "in_rise" que dura 1 ciclo de reloj.

   Cuando se produce el primer flanco de subida en la señal "encoder_in", se habilita el contador, y éste comienza a contar ciclos de reloj hasta que se produzca un reset del sistema. Una vez habilitado el contador, su valor incrementa en 1 unidad en cada ciclo de reloj. Cuando llega un flanco de subida en la señal "encoder_in", el valor del contador se carga en la salida "period_len", la salida del contador se fija al valor 1, y el contador continúa midiendo el siguiente periodo de la señal de entrada.

   Cabe destacar que el diseño asegura que la salida del contador no sobrepase nunca el valor máximo que se puede codificar con 16 bits (65.535 ciclos de reloj), lo que podría suceder cuando la frecuencia de la señal "encoder_in" es inferior a 100 MHz / 65535 = 1,525 kHz. Esta sería la frecuencia mínima de la señal "encoder_in" a la que podemos medir su periodo.

Listado del código VHDL del módulo "period_measure":

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

entity period_measure is
generic (
    N                : integer:=16        -- Number of bits of for time measurement
    );
port (
    clk              : in  std_logic;     -- Clock signal
    rst              : in  std_logic;     -- Reset signal
    encoder_in       : in  std_logic;     -- Encoder signal
    period_len       : out std_logic_vector(N-1 downto 0)   -- Duration of period
    );
end period_measure;

architecture rtl of period_measure is
    constant C_MAX_COUNT     : unsigned(N-1 downto 0):=(others=>'1');  -- Maximum value of counter
    signal counter_en        : std_logic;                              -- Enable counter of cicles in period
    signal counter           : unsigned(N-1 downto 0);                 -- Counter of cicles in period
    signal in_rise           : std_logic;                              -- Rising edge of encoder signal
    signal input_0           : std_logic;
    signal input_1           : std_logic;
    signal input_2           : std_logic;

begin

edge_detector_pr : process(clk,rst)
begin
    if(rst='1') then
        in_rise  <= '0';
        input_0  <= '0';
        input_1  <= '0';
        input_2  <= '0';
    elsif(rising_edge(clk)) then
        in_rise  <= not input_2 and input_1;
        input_0  <= encoder_in;
        input_1  <= input_0;
        input_2  <= input_1;
    end if;
end process edge_detector_pr;

count_period_pr : process(clk,rst)
begin
    if(rst='1') then
        counter_en  <= '0';
        counter     <= to_unsigned(1,N);
        period_len  <= (others=>'0');
    elsif(rising_edge(clk)) then
        if(in_rise='1') then
           counter_en  <= '1';
           counter     <= to_unsigned(1,N);
           period_len  <= std_logic_vector(counter);
        elsif (counter_en='1') then
           if(counter<C_MAX_COUNT)then
              counter  <= counter + 1;
           end if;
        end if;
    end if;
end process count_period_pr;

end rtl;
```

3) Compilamos y anotamos los resultados obtenidos:

Compilation Report - period_measure

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Feb 26 20:29:14 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	period_measure
Top-level Entity Name	period_measure
Family	Cyclone IV E
Total logic elements	53 / 6,272 (< 1 %)
Total registers	37
Total pins	19 / 92 (21 %)
Total virtual pins	0
Total memory bits	0 / 276,480 (0 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)
Device	EP4CE6E22C6
Timing Models	Final

Ninguna sorpresa respecto el número de pines: 19 (es la suma exacta de los que tenemos definidos en la entidad). El número de registros (37 flip-flops) es también lo que toca: 16 registros para el contador, 16 para la salida "period_len", y 5 más para las señales "counter_en", "in_rise", "input_0", "input_1", y "input_2". Por otro lado, los elementos lógicos se elevan a 53, ya que tenemos un comparador de 16 bits y un sumador, además de la lógica de control.

4) A continuación hacemos una simulación. El listado de un posible banco de pruebas se muestra a continuación.

Listado del banco de pruebas del módulo “period_measure” (1 de 2):

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

entity period_measure_testbench IS
end period_measure_testbench;

architecture behavior of period_measure_testbench is

    -- Unit under test.
    component period_measure
       generic (
          N                 : integer:=16        -- Number of bits of for time measurement
          );
       port(
          clk               : in  std_logic;     -- Clock signal
          rst               : in  std_logic;     -- Reset signal
          encoder_in        : in  std_logic;     -- Encoder signal
          period_len        : out std_logic_vector(N-1 downto 0)   -- Duration of period
       );
    end component;

    signal clk           : std_logic := '0';
    signal reset         : std_logic := '0';
    signal encoder_in    : std_logic := '0';
    signal period_len    : std_logic_vector(15 downto 0);

    -- Clock definition
    constant clk_period : time := 10 ns;

begin
    -- Instance of the unit under test.
    uut: period_measure PORT MAP (
       clk => clk,
       rst => reset,
       encoder_in => encoder_in,
       period_len => period_len
       );

    -- Definition of the clock process.
    clk_process :process
    begin
       clk <= '0';
       wait for clk_period/2;
       clk <= '1';
       wait for clk_period/2;
    end process;

    -- Stimuli process.
    stimuli: process
    begin
       -- Reset circuitry
       reset <= '1';
       wait for clk_period * 2;
       reset <= '0';

       -- Test case 1: encoder period = 16 clock cycles
       for k in 0 to 2 loop
          encoder_in <= '1';
          wait for clk_period * 8;
          encoder_in <= '0';
          wait for clk_period * 8;
       end loop;
```

Listado del banco de pruebas del módulo “period_measure” (2 de 2):

```
64
65          -- Test case 2: encoder period = 32 clock cycles
66          for k in 0 to 2 loop
67              encoder_in <= '1';
68              wait for clk_period * 16;
69              encoder_in <= '0';
70              wait for clk_period * 16;
71          end loop;
72
73          -- Test case 3: encoder period = 64 clock cycles
74          for k in 0 to 2 loop
75              encoder_in <= '1';
76              wait for clk_period * 32;
77              encoder_in <= '0';
78              wait for clk_period * 32;
79          end loop;
80
81          -- Test case 4: encoder period = 256 clock cycles
82          for k in 0 to 2 loop
83              encoder_in <= '1';
84              wait for clk_period * 128;
85              encoder_in <= '0';
86              wait for clk_period * 128;
87          end loop;
88
89          wait;
90      end process;
91  end;
92
```

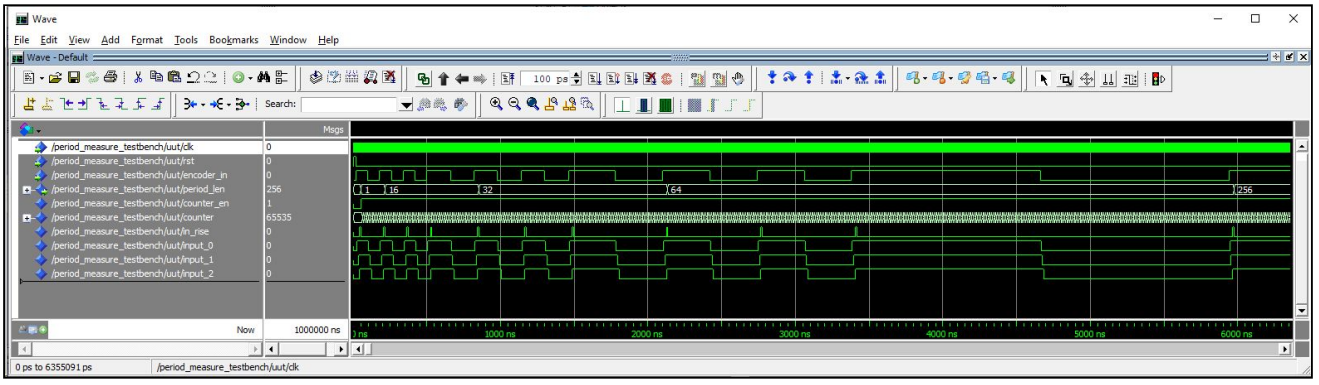
Tal como se puede observar en el banco de pruebas:

- Se declara el componente de nuestro diseño [líneas 11-21] y la instancia [33-38] utilizando las señales internas del banco de pruebas declaradas previamente [23-29].
- Se genera la señal de reloj de control en el proceso "clk_process" [líneas 41-47]. La señal de reloj se inicializa en la línea 23 y utiliza un período de 10 ns definido en la constante "clk_period" [línea 29].
- Se generan los estímulos de entrada necesarios para hacer la comprobación del diseño en el proceso "estimuli" [líneas 50-90]. En particular, se generan varios periodos de la señal "encoder_in" a 4 frecuencias diferentes: 6,25 MHz del **test case 1** (16 ciclos de reloj); 3,125 MHz del **test case 2** (32 ciclos de reloj); 1,562 MHz del **test case 3** (64 ciclos de reloj); y 390.625 kHz del **test case 4** (256 ciclos de reloj).

Tal como se ha comentado, el proceso "stimuli" es el que se encarga de ir generando los estímulos que se aplicarán a las entradas del módulo "period_measure" que estamos comprobando. En este caso, no se hace la comprobación explícita de la salida en este mismo proceso y, por tanto, será el/la diseñador/a quien tendrá que hacer la verificación visual de las formas de onda obtenidas.

A continuación, comprobaremos el resultado de la simulación con RTL Simulation. Es importante destacar que el hecho de utilizar una simulación tipo RTL permite observar todas las señales internas, lo cual ayuda mucho en caso de tener que corregir errores de diseño.

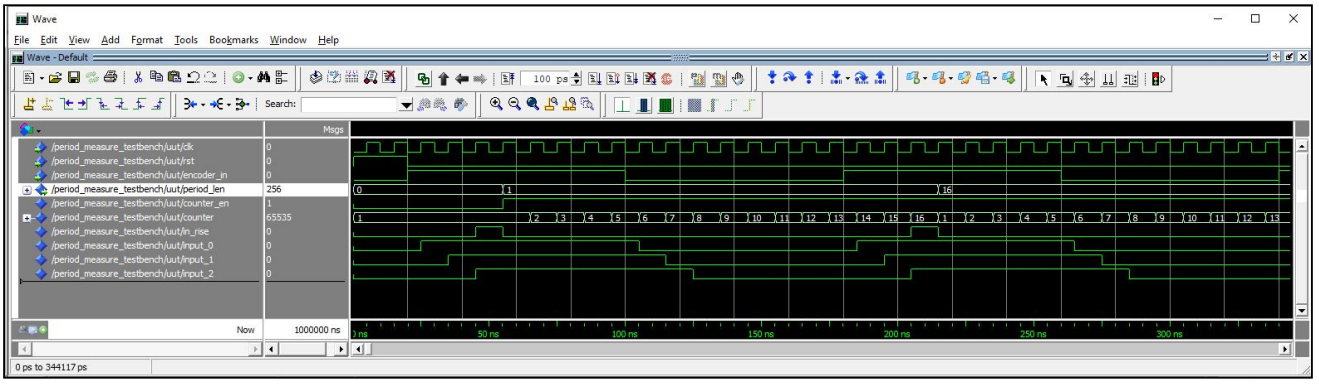
Resultado de la simulación del test cases 1, 2, 3 y 4:



Como se observa en las formas de onda de la simulación, la frecuencia de la señal "encoder_in" va variando a lo largo de la simulación. Empieza a una frecuencia elevada en el test case 1, y ésta se va reduciendo hasta llegar al test case 4. Como se puede comprobar, la medida "period_len" varía coincidiendo con los flancos de subida de la señal "encoder_in" y siempre que su frecuencia cambia. En el test case 1 se mide un "period_len" de 16 ciclos de reloj en 2 periodos consecutivos de la señal "encoder_in". En el test case 2 se mide un "period_len" de 32 ciclos de reloj en 2 periodos consecutivos de la señal "encoder_in". En el test case 3 se mide un "period_len" de 64 ciclos de reloj en 2 periodos consecutivos de la señal "encoder_in". Finalmente, en el test case 4 se mide un "period_len" de 256 ciclos de reloj en 1 periodo de la señal "encoder_in". Por tanto, podemos concluir que los resultados de la simulación son correctos.

En la siguiente figura se muestra el detalle de la simulación del test case 1, donde se puede ver el funcionamiento del circuito de detección de flancos de subida en la señal "encoder_in", la activación de la señal de habilitación "counter_en" del contador, el reset del contador a 1 cuando llega un flanco "in_rise" en la señal de entrada, y la carga de la medida en la salida "period_len".

Resultado de la simulación del test case 1:



Cuestión de investigación (20%)

Comienza la respuesta a esta pregunta en una hoja aparte.

Cuestión: Las modernas FPGAs de los grandes fabricantes (Altera, Xilinx y Microsemi) ofrecen la posibilidad de integrar un procesador en su interior (*embedded processor*), que permite mover al software del procesador parte de las funcionalidades del hardware, especialmente algunos elementos de control y gestión del sistema, sin perder prestaciones y reduciendo los tiempos de desarrollo.

Se propone realizar un pequeño ejercicio de investigación para responder a las siguientes preguntas:

- ¿Qué es un procesador de tipo *hardcore*? ¿Y un procesador de tipo *softcore*?
- ¿Cuáles son las ventajas y los inconvenientes de los procesadores de tipo *hardcore*? ¿Y las ventajas e inconvenientes de los procesadores de tipo *softcore*? Se deben considerar al menos los siguientes criterios: la frecuencia de funcionamiento, la potencia de procesado, la flexibilidad en el diseño, los periféricos disponibles, el consumo energético, el número de puertas lógicas (densidad de ocupación), etc.
- ¿Cuáles son las características básicas (arquitectura, número de bits, periféricos, etc.) de algún procesador de tipo *softcore* que suele integrarse en las FPGAs de Xilinx?

Debes incluir en la respuesta las referencias originales en las que te has basado para hacer el ejercicio de investigación.

Solución:

Seguro que ha habido muy buenas aportaciones por parte de cada uno. Mi intención es recoger esta información y poder publicar las más interesantes o diferentes para compartirlas con el resto del grupo, si nadie tiene inconveniente (en caso contrario, avisadme).