UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

MANUAL PARA PROGRAMAR FPGA USANDO LENGUAJE VHDL, HARDWARE Y SOFTWARE DE ALTERA

MATERIAL DIÁCTICO

Que para obtener el título de

Ingeniero Mecatrónico

P R E S E N T A

Gerardo Martínez Urbina

ASESOR DE MATERIAL DIDÁCTICO

Dr. Octavio Días Hernández

Ciudad Universitaria, Cd. Mx., 2023

AGRADECIMIENTOS

Este trabajo es el resultado de la conclusión de la etapa estudiantil vivida desde la etapa de la educación preescolar y este camino siempre hubo personas quienes fueron importantes para lograr todas esas metas. En primer lugar, agradezco a la madre y al padre celestial a quienes siempre encomendé mis logros me dirigen a ser una persona buena. Los siguientes son mi madre y padre quienes siempre me dieron su apoyo y amor incondicional en las buenas y en las malas lo cual me permitió convertirme en un profesional. También agradezco a mi hermana y a mi hermano quienes son mis compañeros de vida y me permiten compartir mis logos para que ellos sigan ese ejemplo. Los abuelos son personas muy sabias a quienes siempre recurría para pedir ayuda y gracias a ellos pude tomar las mejores decisiones. Agradezco a todas las demás personas y seres que forman parte de esta gran familia que con sus valores y su apoyo incondicional forjaron a esta gran persona.

Igualmente se agradece a la comunidad Facultad de Ingeniería de la Universidad Nacional Autónoma de México donde los docentes y estudiantes quienes formaron parte de los equipos de trabajo donde yo tuve la oportunidad de participar durante los cinco años de duración del plan de estudios de la carrera de Ingeniería Mecatrónica. En especial, agradezco al Dr. Octavio Díaz Hernández quien supervisó la elaboración de este trabajo, así como el apoyo personal brindado.

Se agradece a la UNAM-DGAPA que a través del proyecto PAPIME con clave PE102022 y titulado “Diseño de prácticas de laboratorio para las asignaturas del área de electrónica en la Licenciatura de Órtesis y Prótesis”, fue realizado este trabajo.

PREFACIO

El presente manual de prácticas es un conjunto de herramientas para el diseño de circuitos, programación y aplicaciones de los dispositivos FPGA. El propósito es que el lector conozca una herramienta adicional para el desarrollo de sistemas digitales y que puedan resolver problemas ingenieriles o de la vida cotidiana.

El alcance principal de este escrito es que los alumnos aprendan a manejar los lenguajes de programación empleados en estos dispositivos, así como el uso de los distintos periféricos que necesitarán los FPGA para formar un sistema más elaborado y que sea útil.

OBJETIVOS

Objetivo general

Proponer un manual para programar FPGA usando lenguaje VHDL, así como el hardware y software de ALTERA requeridos.

Objetivos específicos

- Mostrar los comandos y estructuras básicas del lenguaje VHDL
- Implementar circuitos digitales usando lenguaje descriptivo
- Manipular el software Quartus II 13.0sp1 para programar en código VHDL y por diagrama de bloques
- Implementar dispositivos periféricos conectados el FPGA para:
    - Obtener información de sensores analógicos
    - Proponer el control de actuadores
    - Desplegar información en pantallas LCD

CARRERAS AFINES

- Ingeniería Eléctrica y Electrónica
- Ingeniería Mecatrónica
- Ingeniería en Sistemas Biomédicos
- Licenciatura en Órtesis y Prótesis

Nota

La asignatura donde está contenida el tema de los FPGA es la de circuitos digitales para las carreras de Ingeniería Mecatrónica, Sistemas Biomédicos y Licenciatura en Órtesis y Prótesis. En el caso de la carrera de Ingeniería Eléctrica y Electrónica es la asignatura de diseño digital.

POR QUÉ ES APOYO A LA DOCENCIA

Este manual pretende que los docentes de las carreras que involucren la electrónica digital en sus planes de estudio puedan incluir en sus cursos el estudio e implementación de los FPGA. Se pretende que las prácticas propuestas formen parte de la evaluación de los alumnos con el objetivo de que puedan tener una herramienta más para resolver problemas de ingeniería.

JUSTIFICACIÓN

Los FPGA son un tipo de dispositivo electrónico programable donde su estructura interna posee un número determinado de bloques los cuales contienen miles de compuestas lógicas, por lo tanto, esta se puede implementar cualquier tipo de circuito digital.

Su principal ventaja es que pueden ejecutar procesos de forma paralela y usar altas frecuencias de trabajo lo que permite aumentar el rendimiento de los sistemas implementados en el FPGA.

A QUIÉN VA DIRIGIDO

Se ha mencionado que este manual tiene el propósito a aquellos alumnos de ingeniería que estudien el área de electrónica, para poder extraer el conocimiento de este documento es importante que los alumnos tengan los conocimientos que se enlistan a continuación:

- Interpretación de diagramas electrónicos
- Algebra de Boole
- Circuitos digitales combinacionales
- Circuitos digitales secuenciales (por ejemplo, máquinas de estado)

Además, se requieren conocimientos básicos de programación, en especial el entendimiento de los distintos tipos de variables como lo que serían variables enteras, arreglos, cadenas, bits y caracteres. También se requiere de estructuras lógicas de control como lo son *"if", "else" y "case".*

Es importante también que el lector debe estar dispuesto a adquirir los dispositivos necesarios para la implementación de las prácticas de este manual con la finalidad de que el conocimiento se enriquezca y sea comprensible los conceptos que en algunas ocasiones pueden ser más complejos.

QUIÉN LO VA A USAR

- Profesores que quieran incluir en sus actividades teórico-prácticas el estudio de los FPGA
- Alumnos de las carreras antes mencionadas que quieran especializarse en el área de la electrónica y la instrumentación

CONTENIDO

RESUMEN

Antecedentes

Los dispositivos lógicos programables son una herramienta importante para el diseño de circuitos digitales debido a que rigen el funcionamiento de cualquier dispositivo inteligente que sea usado actualmente. Entre ellos sen encuentran los FPGA que, por sus siglas en inglés, se definen como matrices de compuertas lógicas programables las cuales son utilizadas para el procesamiento de datos y señales lógicas con propósitos específicos.

Los FPGA, como su nombre lo indica, contienen una matriz de miles de bloques, los cuales contienen un número definido de compuertas lógicas, donde se manipulan las interconexiones para dar forma al circuito lógico que se está diseñando. Por esta razón, la programación de estos dispositivos es descriptiva, es decir, el diseñador tiene que introducir en el código el comportamiento del circuito en respuesta a las señales de entrada. Los lenguajes de programación más usados para programar los FPGA son VHDL y Verilog.

Objetivo

Este manual permitirá a los docentes y estudiantes de carreras relacionadas con la electrónica digital implementar los FPGA para resolver problemas de ingeniería.

Metodología

Se propusieron once prácticas divididas en dos secciones: la primera sección (de la práctica 1 a la 5) contempla la implementación de circuitos electrónicos digitales básicos para el entendimiento del lenguaje VHDL. La siguiente sección (de la práctica 6 a la 11) implementa dispositivos periféricos conectados al FPGA que permiten accionar motores, leer señales digitales y analógicas con el objeto de que el lector sea capaz de elaborar un circuito digital con FPGA.

Resultados

Cada una de las prácticas contiene una solución propuesta por el autor de este trabajo.

Impacto

Se pretende que los alumnos que estudien este manual sean capaces de un FPGA programable que será una alternativa al uso de los microcontroladores y de esta manera puedan aprovechar sus ventajas.

CAPÍTULO 1
DISPOSITIVOS LÓGICOS PROGRAMABLES

1.1 DISPOSITIVO LÓGICO PROGRAMABLE

Un dispositivo electrónico programable (PLD, por sus siglas en inglés) es un dispositivo electrónico con el que se puede implementar el diseño lógico para crear un circuito digital y se pueden reconfigurar cuantas veces sea necesario. “El PLD proporciona a los programadores la flexibilidad para implementar diferentes diseños complejos para diversas aplicaciones de circuitos” (S. Parab, S. Gad, & Naik, 2018, pág. 2). Por lo tanto, se pueden implementar distintos tipos de circuitos desde memorias PROM, procesadores, incluso pueden emular el funcionamiento de un microcontrolador. Los PLD se dividen en dos grandes ramas: la **primera familia** consiste en dispositivos lógicos que cumplen una función específica y por lo tanto solo son programados una sola vez por el vendedor; la **segunda familia** abarca a los dispositivos que son programados por el usuario y pueden programarse las veces que sea necesario.

Los PLD, que pertenezcan a la segunda familia antes mencionada, son clasificados en dos grandes ramas, la primera de ellas consiste en los SPLD (dispositivos lógicos programables simples) y los HDPLD (dispositivos lógicos programables de alta densidad), en la Figura 1.1 se puede apreciar un diagrama esquemático de cada una de las familias.

.

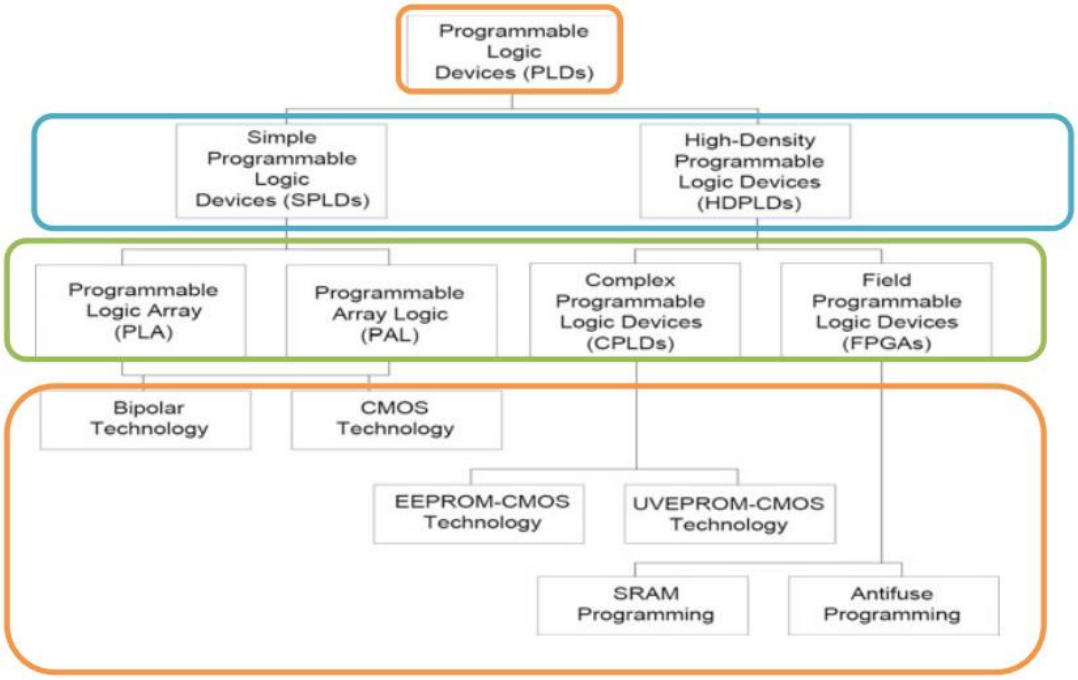


Figura 1.1 Familia de los PLD (S. Parab, S. Gad, & Naik, 2018, pág. 5).

1.1.1 DISPOSITIVOS LÓGICOS PROGRAMABLES SIMPLES

Los dispositivos lógicos programables simples (SPLD, por sus siglas en inglés) son dispositivos que poseen una matriz de compuertas lógicas, las cuales ascienden a un número de varios de cientos, y son capaces de guardar la configuración que les ha proporcionado el diseñador. Se caracterizan por su rápida respuesta debido al bajo número de conexiones que posee internamente y por lo tanto consumen poca energía. De esta rama del SPLD se derivan del PLA (arreglos lógicos programables) que utilizan tecnología bipolar mientras que los PAL (matriz lógica programable) utilizan tecnología CMOS (complementary metal-oxide-semiconductor).

1.1.2 DISPOSITIVOS LÓGICOS PROGRAMABLES DE ALTA DENSIDAD

Como su nombre lo indica, los HDPLD son dispositivos que tienen una alta densidad de arreglos de compuertas lógicas las cuales son agrupadas en bloques permitiendo que exista mayor flexibilidad, comparado con los SPLD, y capacidad de realizar conexiones entre los

bloques lógicos los cuales contienen compuertas AND y OR. Esta variante permite al diseñador implementar diseños lógicos complejos debido a su flexibilidad.

Los **CPLD** (dispositivos lógicos programables complejos) tienen un funcionamiento similar a una memoria EPROM (Erasable Programmable Read-Only Memory) o EEPROM (Electric Erasable Programmable Read-Only Memory) debido a que no pierden su estructura lógica programada al interrumpirse la alimentación, por lo tanto, se considera que la lógica de estos dispositivos es no volátil. Se caracterizan también por ser pequeños y baratos debido a que el número de bloques lógicos que posee es considerablemente menor a un FPGA (Field Programable Gate Array, en inglés).

Los **FPGA,** que traducido al español significa matriz de compuestas lógicas programables, tienen un funcionamiento similar a las memorias que almacenan datos de manera temporal, tales como las RAM (Random Access Memory) y SRAM (Static Random Access Memory) lo que permite que puede programarse un sinnúmero de veces, aunque si es interrumpida la alimentación se pierde el circuito lógico programado. Al igual que los PLD, poseen un número elevado bloques lógicos lo que permite a los FPGA sintetizar circuitos más complejos, por lo tanto, su tamaño es mayor y su costo es considerablemente elevado.

1.2 FPGA RECOMENDADO PARA ESTE MANUAL

Los FPGA, son dispositivos que no son fáciles de adquirir ya que su precio puede ser elevado, por lo que se recomienda adquirir las siguientes placas de desarrollo que son fáciles de encontrar y tienen un precio accesible. Este manual es compatible con los FPGA fabricados por ALTERA, por lo que se recomienda usar cualquiera de las siguientes tarjetas que se proponen.

1.2.1 CYCLONE II EP2C5T144

Figura 1.2 FPGA Cyclone II de ALTERA

En la Figura 1.2 se muestra una fotografía de la tarjeta de desarrollo que se puede adquirir en las tiendas de electrónica. Dicha tarjeta posee 89 pines que pueden ser utilizados como entradas y salidas, una RAM de varios bloques de 4 Kbits (total: 119.898 bits), una EEPROM EPCS4 de 4 Mbit (sólo programable a través del puerto AS).

Se sabe que los voltajes de alimentación normalizados más comunes son 3.3V y 5V, pero la mayoría de los dispositivos periféricos como convertidores analógicos a digital (ADC) y pantallas de cristal líquido (LCD) requieren por defecto una alimentación de 5V. Si se está trabajando las prácticas con la tarjeta de desarrollo Cyclone IV no se tendrá ningún problema ya que posee reguladores y pines para ambos voltajes.

En cambio, la tarjeta Cyclone II (en la cual están basados los diagramas y conexiones de este manual) no posee reguladores de 5 V y, por lo tanto, no contiene pines para dicho voltaje de alimentación por defecto. Este problema se puede resolver de la siguiente forma:

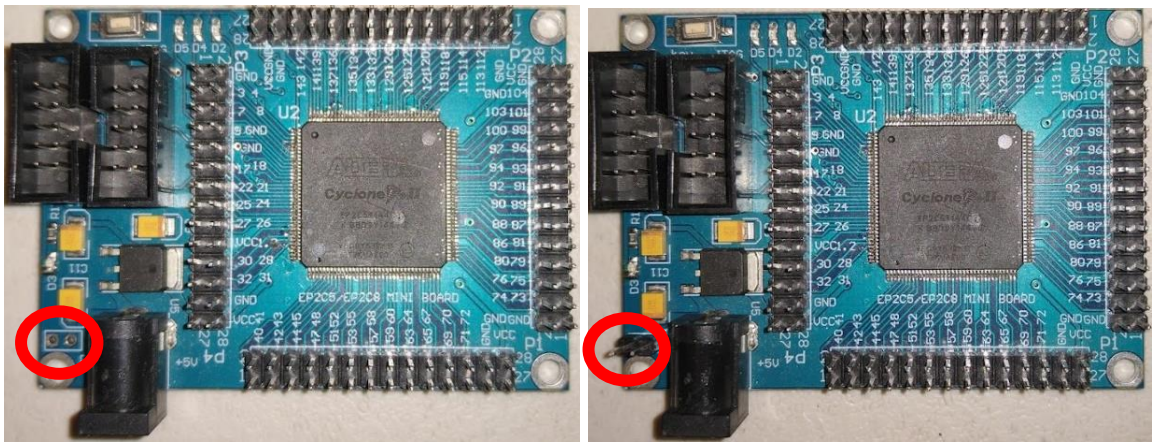


Figura 1.3 Ubicación de los pines de alimentación de 5 V en la Cyclone II

En la Figura 1.3 se aprecia que la placa de desarrollo tiene dos pequeños agujeros donde puede soldarse un par de terminales conocidos como headers (ya sean macho o hembra). Por lo que se debe soldar ese par de headers con la finalidad de que se requiera usar del voltaje suministrado al FPGA no se tenga que usar fuentes externas de voltaje.

El pin del lado izquierdo transmite la corriente eléctrica con un voltaje igual al de la alimentación, por lo tanto, se recomienda alimentar el FPGA con una fuente de 5 V para alimentar otros circuitos que requieran ese voltaje de alimentación. En cuanto a la corriente de la fuente usada debe ser capaz de suministrar más de 1 A ya que el modelo Cyclone II consume una corriente de alrededor de 600 mA, por lo tanto, es normal que presente un aumento de temperatura el dispositivo. Este detalle no se presenta en el modelo Cyclone IV ya que al ser un modelo más reciente consume menos energía lo que permite que sea alimentado con una PC mediante un puerto USB o cualquier eliminador de voltaje menor a 1 A.

1.2.2 CYCLONE IV EP4CE6E22C8N

Figura 1.4 Tarjeta de desarrollo Cyclone IV de ALTERA.

La Figura 1.4 muestra una fotografía de una segunda tarjeta de desarrollo que puede usar para este manual. Es ligeramente más costosa que el modelo anterior, posee 88 pines que pueden ser configurados como entrada y salida por lo que la diferencia es notoria con respecto al modelo anterior, debido a que este FPGA tiene cuatro botones, un dip-switch de cuatro interruptores, cuatro despliegues de siete segmentos, cuatro leds y un puerto de 16 pines para colocar una pantalla de cristal líquido. Adicionalmente tiene soldada una memoria EEPROM tipo M25P16 de 16 Mb programable por el puerto AS, una memoria SDRAM de 64 Mb y una memoria EEPROM de comunicación por I2C modelo AT24C08.

En el apéndice A se pueden consultar los diagramas de conexiones de ambas tarjetas de desarrollo y, en el caso de la tarjeta con Cyclone IV, se tiene también la configuración se sus pines para cada uno de los módulos que tiene integrado.

CAPÍTULO 2
INTRODUCCIÓN A LENGUAJE VHDL

Los FPGA son capaces de emular circuitos lógicos complejo a través de sus matices de compuertas lógicas que se interconectan entre ellas, para ello utilizan el lenguaje de programación descriptivo que permita establecer el comportamiento del circuito lógico a implementar.

VHDL es uno de los tantos lenguajes de programación cuyo funcionamiento consiste en utilizar un lenguaje de descripción de hardware el cual describe el funcionamiento y la estructura de un circuito lógico que será posteriormente programado en el FPGA.

> "VHDL permite la síntesis de circuitos, así como la simulación de circuitos. El primero es la traducción de un código fuente a una estructura de hardware que implementa la funcionalidad deseada, mientras que el segundo es un procedimiento de prueba para asegurar que tal de hecho, la funcionalidad se logra mediante el circuito sintetizado." (A. Pedroni, 2004, pág. 3)

Anteriormente se mencionó que el VHDL se utiliza para describir el funcionamiento de circuitos digitales por lo que inevitablemente se trabaja con señales digitales; las cuales solo pueden existir dos posibles valores: 1 (alto) y 0 (bajo).

VHDL permite también ejecutar operaciones aritméticas (ejem. sumar, restar, dividir y multiplicar), es capaz de sintetizar números enteros y con punto flotante usando los archivos de biblioteca adecuadas.

Debido a su naturaleza descriptiva, es capaz de ejecutar estructuras lógicas usadas en otros lenguajes de programación, tales como *if/else y case* así como estructuras cíclicas como *for* y *while*. También tiene la capacidad de declarar funciones y ser llamadas en cualquier parte del código.

2.1 VALORES NUMÉRICOS

Números binarios

Los números binarios son la base para el diseño digital y por lo tanto son la manera en que se pueden trabajar y sintetizar las señales digitales en VHDL. En este lenguaje de programación pueden existir números binarios de un solo bit y se representan de esta manera.

```
a <= '0'; -- a = 0
```

En el ejemplo anterior la variable "a" almacena el número 0 en un número de un solo bit. La asignación de valores en VHDL se realiza con el símbolo "<=" y, de manera similar al lenguaje C, cada línea de código se debe finalizar con ";". Por otro lado, para realizar comentarios en VHDL se usa doble guion "--" y seguido del comentario, estos se usarán para escribir el valor de las variables en esta sección.

En la línea mostrada anteriormente, se usaron dos apóstrofos para escribir un número de un solo bit el cual solo puede tener dos posibles valores (0 y 1), aunque se puede usar la siguiente notación:

```
a <= b"1" -- a = 1;
```

Para representar números binarios de dos o más bits, la variable que contenga el valor debe declararse con anterioridad como un **vector de bits**. Puede usarse la notación anterior y solamente se agregan más elementos dentro de dos comillas dobles.

```
b <= b"1001"; -- b = 9
c <= b"11110001"; -- c = 241
```

La variable "b" está declarada como un número binario de 4 bits cuyo valor corresponde a 9, mientras que "c" es un número de 8 bits y representa al número 241, por lo tanto, sus valores máximos son 15 y 255 respectivamente.

La notación usada anteriormente incluye una b antes de escribir el número binario dentro de las comillas dobles, pero se pueden suprimir y el código funciona de igual forma.

```
b <= "1001";
c <= "11110001";
```

Números hexadecimales

La letra “b” en la notación anterior indica que se trata de un número binario y de esa manera VHDL identifica que se está usando números de base 2. Si “b” es reemplazada por la letra “x”, representaría a los números base 16 o mejor conocidos como números hexadecimales. En el siguiente ejemplo se asigna a la variable “d” el valor de 205 representado en número hexadecimal.

```
d <= x"CD";
```

Números octales

Los números base 8 o también conocidos como octales se representan reemplazando la letra “h” por la letra “o” antes de escribir las comillas dobles. En el siguiente ejemplo, la variable e se asigna el valor 205 usando un número octal.

```
e <= o"315";
```

Las variables d y e han implícitamente son números de 8 bits por lo que su rango de valores es entre 0 y 255 sin importar que se usen binarios, octales o hexadecimales. La declaración de vectores de bit se explicará a detalle en la sección 2.4.

Números decimales

Los números base 10, mejor conocidos como decimales, son sintetizables en VHDL. En las siguientes líneas de código del siguiente ejemplo se muestra la asignación un número natural a la variable “f”, para la variable “g” se asigna un entero negativo y por último a “h” se asigna un número real con punto flotante.

```
f <= 143;
g <= -17;
h <= -3.14159;
```

2.2 CARACTERES Y CADENAS

Los caracteres pueden usarse también en VHDL, pero como en cualquier lenguaje, las variables deben declararse para albergar caracteres o cadenas. En el siguiente ejemplo “i” se ha declarado para almacenar caracteres y se le asigna el carácter “+”. Para cualquier carácter, se debe incluir ente dos apóstrofos:

```
i <= '+';
```

En el siguiente ejemplo, para la misma variable “i”, se hace uso de una función donde el argumento es el número que corresponde al carácter “+” en el código ASCII.

```
i <= character'val(24);
```

De una manera similar, las cadenas se escriben entre dos comillas dobles. En los siguientes ejemplos, la variable “j” se ha declarado una cadena de 6 caracteres, por lo tanto, la cadena asignada debe tener la misma extensión. De igual manera la variable “k” se ha declarada una cadena de 2 caracteres. La declaración de cadenas se explicará a detalle en la sección 2.4.8.

```
j <= "Hellow";
k <= ":)";
```

Dos o más cadenas pueden concatenarse para crear una sola cadena, para ello se usa “&” entre cada cadena. En el siguiente ejemplo “l” se ha declarado una cadena de 13 caracteres y en la asignación del su valor, se concatenan tres cadenas y un carácter.

```
l <= "Hello" & " " & "world" & ":)";
```

En la concatenación de cadenas, se puede usar otras variables que almacenen tanto cadenas como caracteres. En el siguiente ejemplo, la asignación de la cadena a la variable “l” se realiza con las variables “j” y “k” de los ejemplos anteriores y el resultado es exactamente el mismo.

```
l <= j & " " & "world" & k;
```

2.3 ESTRUCTURA DE CÓDIGO

2.3.1 ARCHIVOS DE BIBLIOTECA

Esta sección del código, como en otros lenguajes de programación, consiste en la declaración de los archivos de bibliotecas que habilitan el uso de tipos de datos, operadores y funciones. En VHDL se usa por defecto el archivo de biblioteca **ieee** el cual contiene paquetes de código que se declaran de manera individual para usar su contenido en el código principal. Existen diversos paquetes para diferentes propósitos por lo que se considera para este manual usar tres paquetes que cubren lo necesario para el funcionamiento de las actividades propuestas en las secciones subsecuentes.

Paquete std_logic_1164

Contiene el tipo de variable *STD_LOGIC* el cual permite declarar números de un solo bit. También incluye la variante *STD_LOGIC_VECTOR* que son arreglos o vectores de bits que permiten representar números de dos o más bits. El paquete define los valores que representan los posibles estados lógicos: “1” (alto) y “0” (bajo). El paquete permite sintetizar otros estados lógicos, que no se usarán en este manual, como “u” (no asignado), “z” (alta impedancia) y “-” (no importa el valor).

Paquete numeric_std

El paquete permite manipular números con signo (UNSIGNED), que se representan como arreglos de bits similar a los STD_LOGIC_VECTOR, y con signo (SIGNED) que, con base en el método de complemento a 2, agrega el bit significativo al arreglo para definir si el número es positivo o negativo. Además, permite manipular números naturales y números enteros (con y sin signo). Este paquete también contiene a los operadores aritméticos (suma, resta, multiplicación y división) para realizar operaciones con números enteros, además se pueden escribir en notación científica y ejecutar funciones que puedan realizar la conversión los *STD_LOGIG_VECTOR* a números enteros y viceversa.

Paquete math_real

Este paquete trabaja de manera similar a **numeric_std** con la diferencia que permite manipular números con punto flotante, incluye constantes matemáticas tales como el número Pi y la constante de Euler. El paquete contiene las operaciones matemáticas básicas (suma, resta multiplicación y división) para operar con números flotantes y añadiendo operaciones matemáticas avanzadas como potencias, raíces y funciones trigonométricas.

Declaración de los archivos de bibliotecas y los paquetes

Para declarar un archivo de biblioteca primero debe escribirse en las primeras líneas de código siguiendo la siguiente sintaxis:

```
library [library_name]
```

Mientras que para declarar los paquetes de los archivos de biblioteca se usa la siguiente estructura:

```
use [library_name].[package_name].all;
```

En el siguiente ejemplo se muestra la declaración del archivo de biblioteca **ieee** y los paquetes mencionados anteriormente los cuales se usarán en cada una de las prácticas que se propongan en las secciones subsecuentes:

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

2.3.2 ESTRUCTURA ENTITY

Esta sección del código define la estructura externa del circuito digital la cual es representada en el código. En ella se establece un nombre, así como el número de señales externas que interactúan en el circuito. En esta sección se utiliza una función llamada ***port()*** en la cual se listan todas las señales de entrada y salida del circuito.

```
entity entity_name is
port (

port_1_name: port_mode signal_type;
port_2_name: port_mode signal_type;
port_3_name: port_mode signal_type;

port_n_name: port_mode signal_type

);
end entity;
```

"*port_mode*" puede ser reemplazado por la palabra **in** que define que se trata de una entrada o también se puede emplear **out** que especifica que es una salida. Por otro lado, en lugar de "*signal type*" pueden usarse las variables *STD_LOGIC* para señales de un solo bit o también *STD_LOGIC_VECTOR* para señales de dos o más bits. En el siguiente ejemplo se ilustra la declaración de una entidad.

```
entity example is
port (

port_A: in std_logic;
port_B: in std_logic_vector (3 downto 0);
port_C: out std_logic_vector (6 downto 0);
port_D: out std_logic

);
end entity;
```

Adicionalmente, existe la función **generic()** que puede ser usada de manera opcional la cual declara valores paramétricos que pueden ser usados en cualquier parte del código. Para usar la función se escribe:

```
generic (

constant_1_name: constant_1_type := constant_1_value;
constant_2_name: constant_2_type := constant_2_value;
constant_3_name: constant_3_type := constant_3_value;

constant_n_name: constant_n_type := constant_n_value
);
```

Esta función se escribe antes de declarar la función **port()**. Usando el ejemplo anterior, se declara la función **generic()** para agregar parámetros:

```
entity example is

generic (

integer_constant: integer := 10;
simple_bit: std_logic :='0';
binary_number: std_logic_vector(2 downto 0) := "101"

);

port (

port_A: in std_logic;
port_B: in std_logic_vector (3 downto 0);
port_C: out std_logic_vector (6 downto 0);
port_D: out std_logic

);
end entity;
```

2.3.3 ESTRUCTURA ARCHITECTURE

ARCHITECTURE es la sección del código que describe el funcionamiento del circuito lógico. Su estructura es la siguiente:

```
architecture architecture_name of entity_name is
 --architecture_declarative_part
begin
--architecture_statements_part
end architecture;
```

La sección declarativa (situada entre **architecture** y **begin**) se puede escribir de manera opcional, en ella se declaran constantes, señales, variables internas y funciones que son utilizadas en el cuerpo de la arquitectura (“architecture_declarative”). Este describe el funcionamiento y la estructura del circuito lógico. En el siguiente ejemplo se ilustra la escritura de una arquitectura para una compuerta XOR:

```
architecture main of example is
begin

 c <= a xor b;

end architecture;
```

A continuación, se ilustra otro ejemplo que contiene declaraciones de constantes y señales internas que son usadas en la estructura de la arquitectura:

```
architecture behavioral of BLINKING is

 constant  count_limit :  integer   := 12499999;
 signal  count  :  integer range 0 to 50e6 := 0;
 signal  led_state :  std_logic;

begin
 blink: process (CLK)
  begin
  if rising_edge (CLK) then
   if (count = count_limit) then
    led_state <= not led_state;
    led  <= led_state;
    count<=0;
   else
    count<= count+1;
   end if;
  end if;
 end process;
end architecture;
```

2.3.4 UNIDADES SECUENCIALES

La estructura de ARCHITECTURE se caracteriza por contener a las unidades secuenciales, o conocidos también como PROCESS, que son la estructura principal para el funcionamiento del código en VHDL, ya que en ellos se escribe una sección, o completamente, el código que describe al circuito digital. Cada proceso representa una unidad secuencial que está sujeto al cambio de una o varias señales lógicas y cuando sucede el cambio por lo menos en una señal, se ejecuta el código contenido. Su estructura es la siguiente:

```
PROCESS_1_NAME: process (sensivity_1) is
begin
-- process statments
end process;

PROCESS_2_NAME: process (sensivity_2) is
begin
-- process statments
end process;
```

Dentro de las unidades secuenciales se ejecutan las estructuras lógicas tales como ciclos (*for* y *while*), *if/else*, *case/when*, que serán presentadas en la sección 2.7, y funciones elaboradas por el programador. Además, se pueden tener tantos procesos como sean necesarios y la gran ventaja es que cada proceso dentro del FPGA se ejecutará de manera paralela lo que permite obtener mejor velocidad y rendimiento. En el siguiente código se ejemplifica dos unidades secuenciales que se ejecutan cada vez que existe un cambio en las señales CLK y RESET.

```
PROCESS_1 : process (CLK) is
 begin
  if(CLK = '1') then
   a <= '1';
   b <= '0';
  else
   a <= '0';
   b <= '1';
  end if;
end process;

PROCESS_2 : process (CLK, RESET) is
 begin
  if(CLK = '0'and RESET = '1') then
   c <= '0';
   d <= '0';
  else
   c <= '1';
   d <= '1';
  end if;
end process;
```

2.4 TIPOS DE DATOS

Existen diversos tipos de datos en VHDL donde cada uno cumple una función específica que permiten al diseñador crear el circuito lógico de la manera más eficiente. Para usarlos basta con declarar los archivos de bibliotecas y los paquetes que permitan usarlos, y después declararlos dentro de ARCHITECTURE usando la siguiente estructura:

```
[variable_type] [data_name] : [data_type] := [data_value];
```

Si no se requiere definir un valor al momento de declarar el dato, simplemente se omite la sección que asigna el valor y se cierra la línea de código después de asignar el tipo de dato:

```
[variable_type] [data_name] : [data_type];
```

2.4.1 STD_LOGIC

Este tipo de dato permite usar un número binario de un solo bit el cual puede adquirir el valor 1 y 0. Para declararlo, se usa la sintaxis:

```
[variable_type] [name] : std_logic := '[bit_value]';
```

Ejemplo:

```
signal my_bit : std_logic := '0';
```

2.4.2 STD_LOGIC_VECTOR

Este trabaja de manera similar al anterior, con la diferencia que alberga un vector de bits y es similar a los arreglos en otros lenguajes de programación. Puede representar buses de datos de más de dos bits, así como números binarios. La sintaxis para declararlos es la siguiente:

```
[variable_type] [name] : std_logic_vector([size - 1] downto 0):=
"[binary_value]";
```

En el ejemplo, la variable “my_bus” contiene un valor STD_LOGIC_VECTOR de que representa un bus de 8 bits:

```
signal my_bus : std_logic_vector (7 downto 0) := "10101111";
```

A continuación, se muestra cómo acceder a un bit en específico de la variable "my_bus" usada anteriormente y se asigna a "my_bit" declarada como *STD_LOGIC*. Se quiere acceder al bit situado en la posición 3 (contando de derecha a izquierda contando desde 0) y el valor obtenido es 1.

```
my_bit <= my_bus(3);
```

En ocasiones se requiere obtener un conjunto de bits de un *STD_LOGIC_VECTOR*. En el siguiente ejemplo "my_bus_2" se ha declarado como un *STD_LOGIC_VECTOR* de 4 bits y el valor asignado corresponde a los bits de las posiciones 7, 6, 5 y 4 de la variable "my_bus" y el resultado es el número binario "1010".

```
my_bus_2 <= my_bus(7 downto 4);
```

2.4.3 SIGNED / UNSIGNED

Los tipos de dato con UNSIGNED trabajan de manera similar a los STD_LOGIC_VECTOR debido a que representan números binarios con más de dos bits mientras que los tipos SIGNED representan números binarios con signo donde, de acuerdo con el método de complemento a 2, el bit más representativo indica el signo del número. En la Tabla 2.1 se muestra la diferencia entra ambos tipos de dato:

Tabla 2.1 Diferencia entre SIGNED y UNSIGNED.

Número binario	SIGNED	USIGNED
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

La declaración de este tipo de datos usa la sintaxis:

```
[variable_type] [name] : signed([size-1] downto 0) :=
"[binary_number]";
[variable_type] [name] : unsigned([size-1] downto 0) :=
"[binary_number]";
```

La declaración de estos tipos de datos es semejante a los *STD_LOGIC_VECTOR* debido a que también almacenan vectores de bits. En el siguiente ejemploen las variables "num_signed" y "num_unsigned" se declaran números con y sin signo de 4 bits respectivamente:

```
signal num_signed : signed(3 downto 0) := "1011" ;
signal num_unsigned : unsigned(3 downto 0) := "1011";
```

La variable "num_signed" almacena el número binario de 1011 que, de acuerdo con el método de complemento a 2, equivale al valor numérico -5 mientras que para la variable "num_unsigned" representa el número 11.

2.4.4 NATURAL

Este tipo de dato almacena y permite operar los números naturales. En este tipo de dato existe un rango de operación el cual se encuentra desde 1 hasta $2^{31} - 1$.

```
[variable_type] [name] : natural := [value];
```

Por ejemplo:

```
signal natural_name : natural := 32;
```

2.4.5 INTEGER

Este tipo de dato permite trabajar con el conjunto de los números enteros.

```
[variable_type] [name] : integer := [value];
```

Ejemplo:

```
signal int_num : integer := -150;
```

2.4.6 REAL

Al usar este tipo de dato es posible manipular el conjunto de los números reales. VHDL utiliza 15 dígitos decimales de precisión.

```
[variable_type] [name] : real := [integer part . float part];
```

Ejemplo:

```
signal real_num : real := -100.0;
```

2.4.7 CHARACTER

Permite usar los valores alfanuméricos de la tabla del código ASCII.

```
[variable_type] [name] : character := '[character]';
```

Ejemplo:

```
signal real_num : character := '$';
```

2.4.8 STRING

Son tipos de datos que almacenan arreglos de caracteres. En VHDL, las cadenas se declaran con un número finito de caracteres y cuando se realiza la asignación de un valor, debe coincidir con el tamaño de la cadena asignado.

```
[variable_type] [name]: string(1 to [string_lenght]):= "[string]";
```

La asignación de la longitud de la cadena se declara de manera similar a los *STD_LOGIC_VECTOR*, aunque éste ultimo los bits se cuentan de derecha a izquierda iniciando desde 0, mientras que las cadenas los caracteres se cuentan de derecha a izquierda iniciando desde 1:

```
signal my_string: string(1 to 11):= "Hello world";
```

En este otro ejemplo se muestra cómo acceder a un carácter o una sección de la cadena.

```
my_char <= my_string(2);
string_2 <= my_string(4 to 8);
```

Donde el valor de “my_char” es el carácter “e”, siguiendo el conteo mencionado anteriormente. Mientras que para “string_2” el valor es una sección de la cadena “my_string” cuyo contenido es la cadena “lo w”.

2.5 TIPOS DE VARIABLES

2.5.1 CONSTANT

Un tipo de variable CONSTANT almacena un valor de cualquier tipo de dato, no se permite el cambio del valor de la variable por lo que siempre será el mismo en cualquier parte del código. Es indispensable que exista un valor por defecto asignado.

```
constant my_constant_1 : std_logic := '1';
constant my_constant_2 : std_logic_vector(3 downto 0):= "1101";
constant my_constant_3 : integer := 100;
```

2.5.2 SIGNAL

Este tipo de dato tiene el propósito de representar las señales físicas internas del circuito que se está programando en el FPGA. Representan las conexiones internas entre las señales de entrada y salida de un circuito digital, aunque también pueden realizar conexiones entre otras señales internas. En las siguientes líneas se muestran algunos tipos de datos declarados como señales internas:

```
signal my_signal_1 : std_logic := '0';
signal my_signal_2 : std_logic_vector(3 downto 0);
signal my_signal_3 : integer;
```

2.5.3 VARIABLE

Son similares a los SIGNAL debido a que se declaran de la misma forma:

```
variable my_variable_1 : std_logic := '0';
variable my_variable_2 : std_logic_vector(3 downto 0);
variable my_variable_3 : integer;
```

Su funcionamiento es casi idéntico con la gran diferencia de que SIGNAL se declara al inicio de la ARCHITECTURE lo que da el rol de una variable global, mientras que VARIABLE se declara dentro de la unidad secuencial en la cual se requiere su uso por lo tanto se considera una variable local. En el siguiente código se muestra un ejemplo de la utilización de ambos tipos de datos.

```
 1 architecture main of example is
 2  signal sig_int : integer;
 3 begin
 4  SEQUENTIAL: process (clock) is
 5   variable var_int : integer;
 6  begin
 7   sig_int <= 0;
 8   var_int := 0;
 9   if(rising_edge(clock)) then
10    sig_int <= sig_int + 1;
11    var_int := var_int + 1;
12    if(sig_int = 10) then ...
13    elsif (var_int = 10) then ...
14    end if;
15   end if;
16  end process;
17 end architecture;
```

2.5.4 TYPE

Los tipos de variable TYPE definen a un conjunto específico de valores y variables que son establecidos por el diseñador. Dichos valores son semejantes a una cadena, debido a que cada uno representa una etiqueta que puede dar nombre a un estado del circuito secuencial y por esa razón son usados para crear máquinas de estado.

En el siguiente código se ejemplifica la declaración de un TYPE que define a un tipo de dato nombrado arbitrariamente como *states* y el cual define un conjunto de diez posibles valores. El siguiente ejemplo se muestra la declaración del conjunto:

```
type states is
 (num_0,num_1,num_2,num_3,num_4,num_5,num_6,num_7,num_8,num_9);
```

En el siguiente código, las variables *Q_bus* y *D_bus* son señales definidas dentro del conjunto *states* donde la primera se inicializa en *num_0* mientras que la segunda no se define en algún valor:

```
signal Q_bus : states :=  num_0;
signal D_bus : states;
```

En las variables Q_bus y D_bus no se pueden guardar valores numéricos, *STD_LOGIC,* o cadenas, solamente los valores los valores que se declararon en el conjunto: *num_0*, *num_1*, *num_2*, *num_3*, *num_4*, *num_5*, *num_6*, *num_7*, *num_8*, y *num_9*.

2.6 OPERADORES

En VHDL se pueden encontrar diversos operadores que realizan operaciones con los tipos de datos mencionados en la sección anterior.

2.6.1 OPERADORES ARITMÉTICOS CON NÚMEROS ENTEROS.

Estos operadores permiten realizar operaciones matemáticas con los números enteros cuyo resultado son números enteros. Para hacer uso de ellos, se tiene que haber declarado antes el paquete *numeric_std*. En los siguientes ejemplos se muestra la forma en cómo se escriben los operadores matemáticos en VHDL y su resultado mostrado en el comentario.

Suma

```
a <= 10 + 6; -- a = 16
```

Resta

```
b <= 6 - 18; -- b = -12
```

Multiplicación

```
c <= -5 * 9; -- c = -45
```

División

```
d <= 5 / 2; -- d = 2
e <= 100 / 3; -- e = 33
```

Para este caso, la división solo puede regresar números enteros, por lo que, si matemáticamente el resultado es un número con punto flotante, el operador truncará el resultado.

Módulo

```
f <= 123 mod 10; -- f = 3;
```

Potencia

```
g <= 4 ** 5; -- g = 4^5 = 1024
```

Notación científica

```
h <= 14e3; -- d = 14*10^3 = 14000
```

2.6.2 OPERADORES ARITMÉTICOS DE NÚMEROS CON PUNTO FLOTANTE

Funcionan de igual forma que los mencionados en el apartado anterior, con la diferencia que el resultado de estos operadores también es un número con punto flotante y pueden operar operaciones matemáticas complejas como las funciones trigonométricas, exponentes y logaritmos. Es importante mencionar que, si el número es entero, pero se quiere operar con números con punto flotante, se debe agregar el punto seguido de un cero (ej. Si es el numero entero 2, en números con punto flotante se escribe 2.0).

Suma

```
a <= 4.5 + 10.0; -- a = 14.5
```

Resta

```
b <= 2.78 - 41.688; -- b = -38.908
```

Multiplicación

```
c <= 100.0 * 0.333; -- c = 33.3
```

División

```
d <= 5.0 / 151.3; -- d = 0.0330469
```

Exponente

```
e <= 5.6 ** 2.1; -- e = 37.2558826
```

Raíz cuadrada

```
f <= sqrt(0.56); -- f = 0.74833
```

Raíz cúbica

```
g <= cbrt(27.0); -- g = 3.0
```

Notación científica

```
h <=1.5153e-4; -- h = 0.00015153
```

Exponencial base e

```
i <= exp(3.25); -- i = 25.79034
```

Logaritmo natural

```
j <= log(12.69) -- j = 2.5408
```

Logaritmo base 10

```
k <= log10(156.4) -- k = 2.194237
```

Seno

```
l <= sin(2.36) -- l = 0.041178
```

Coseno

```
m <= cos(-2.36) -- m = 0.9991518
```

Tangente

```
n <= tan(0.423) -- n = 0.007383
```

Truncar

```
o <= trunc(15.7953) -- o = 15.0
```

Aproximar

```
p <= round(1.4896) -- p = 1.0
q <= round(-3.713) -- q = 4.0
```

2.6.3 OPERADORES LÓGICOS

Comparativos

Realizan las comparaciones matemáticas conocidas, pueden tener argumentos tanto números enteros como números flotantes y el resultado es un booleano que puede ser true (verdadero) o false (falso). En la Tabla 2.2 se muestra el símbolo asignado a cada operador de comparación.

Tabla 2.2 Operadores comparativos.

A = B	A igual a B
A /= B	A diferente de B
A < B	A menor que B
A > B	A mayor que B
A <= B	A menor o igual a B
A >= B	A mayor o igual a B

En el siguiente ejemplo se ilustra que la variable “condition” ha sido declarado variable booleana que solo guarda los valores ture o false.

```
condition <= 12 < -4; -- false
condition <= 0.1561 >= 0 --true
```

En la primera línea se obtiene que la condición es falsa ya que 12 no es mayor a -4, mientras que en la segunda línea la condición es verdadera porque 0.1561 es mayor a cero.

Lógicos

Los operadores lógicos son muy similares a los operadores comparativos del apartado anterior. Existe una diferencia entre ambos, los operadores comparativos trabajan con valores numéricos y devuelven un dato de tipo booleano, mientras que los operadores lógicos trabajan con el tipo de dato *STD_LOGIC*. Por este motivo, los operadores lógicos se consideran equivalentes a las compuertas lógicas usadas en los circuitos digitales.

Tabla 2.3 Compuertas lógicas en VHDL.

not A	Compuerta NOT
A and B	Compuerta AND
A or B	Compuerta OR
A xor B	Compuerta XOR
A nand B	Compuerta NAND
A nor B	Compuerta NOR
A xnor B	Compuerta XNOR

En la Tabla 2.3 se muestra a cada operador lógico en VHDL y su compuerta lógica equivalente. En el siguiente ejemplo la variable "C" se define como el resultado de una función booleana representada por compuertas lógicas.

```
C <= not ((A and B) xor (A or B));
```

Si A = 1 y B = 0, se obtiene que C = 1.

2.7 ESTRUCTURAS LÓGICAS Y CICLOS

Las estructuras lógicas y los siclos que se presentan en esta sección funcionan de la misma manera que en otros lenguajes de programación. Permiten ejecutar bloques de código los cuales están sometidos a condiciones. En la Tabla 2.4 se muestran las estructuras lógicas y cíclicas sintetizables en VHDL.

Tabla 2.4 Estructuras lógicas y ciclos representados en VHDL.

If / else	For

`if ([condition]) then`
`  [code]`
`else`
`  [code]`
`end if;`
	`For [counter] in [min] to [max]`
	`loop`
	`  [Code for cycle];`
	`end loop;`

If / elsif / else

```
if ([condition_1]) then
 [code];
elsif ([condition_2]) then
 [code];
elsif ([condition_N]) then
 [code];
else
 [code];
end if;
```

While

```
while ([condition]) loop
 [Code for cycle];
end loop;
```

Case

```
case ([variable/signal]) is
 when ([case 1]) =>
  [code for case 1];
 when ([case 2]) =>
  [code for case 1];
 when ([case N]) =>
  [code for case N];
 when others =>
  [code for a case that is not
in the previous cases];
end case;
```

Cada una de las estructuras lógicas presentadas en la Tabla 2.4 solamente funcionan si están contenidas dentro de una unidad secuencial *process* presentada en la sección 2.3.3. VHDL no puede sintetizarlas si no cumplen con este requerimiento y si se encuentran fuera de la unidad secuencial, habrá errores de compilación.

CAPÍTULO 3
PROGRAMA QUARTUS II 13.0

Este capítulo es una breve introducción a Altera Quartus II el cual es un programa de diseño lógico. El software es capaz de trabajar con lenguajes de programación como VHDL y Verilog los cuales son usados para describir el funcionamiento de un circuito secuencial que será implementado en un FPGA o CPLD. En las siguientes secciones se explicará el proceso de instalación de la versión gratuita y la creación de proyectos.

3.1 INSTALACIÓN

Para instalar este software se accede a la página de FPGA Software Download Center (intel.com) y en la lista que se muestra se busca "Intel Quartus II web edition Design Software Version 13.0sp1". La Figura 3.1 muestra una sección de la lista que la página de descargas, y entre todas las versiones se debe asegurar de descargar la versión web porque no requiere comprar licencia.

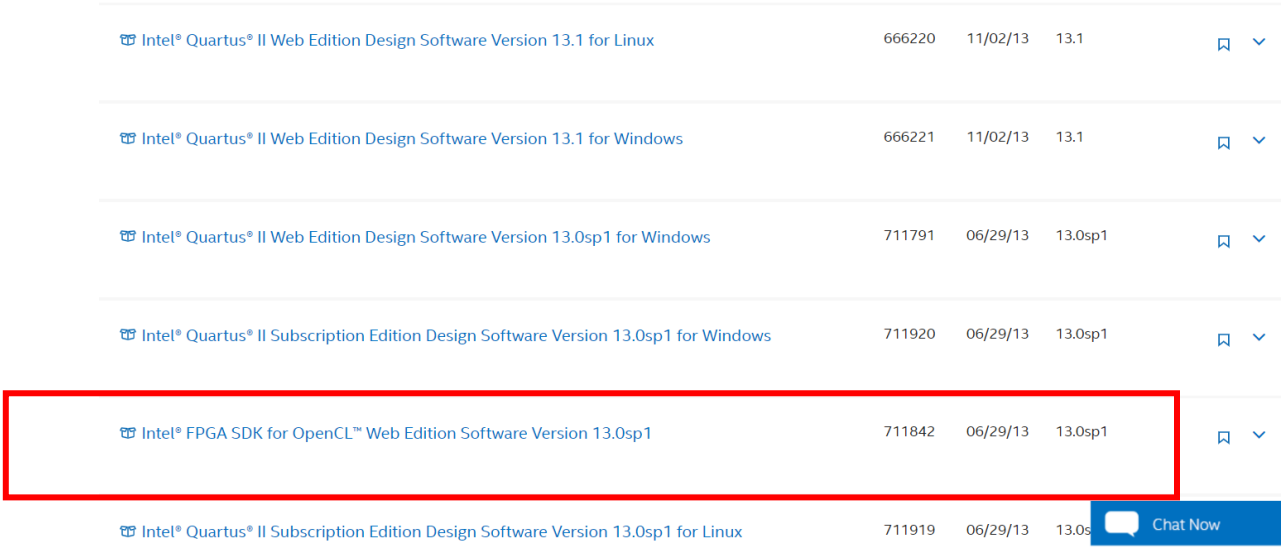


Figura 3.1 Centro de descargas de FPGA de Intel.

Al hacer clic en la opción mencionada en la lista, se dirigirá a una segunda página, donde existen cuatro opciones para descargar el software. Se selecciona la opción “Multiple Download” la cual contiene el software con lo requerido para este manual.

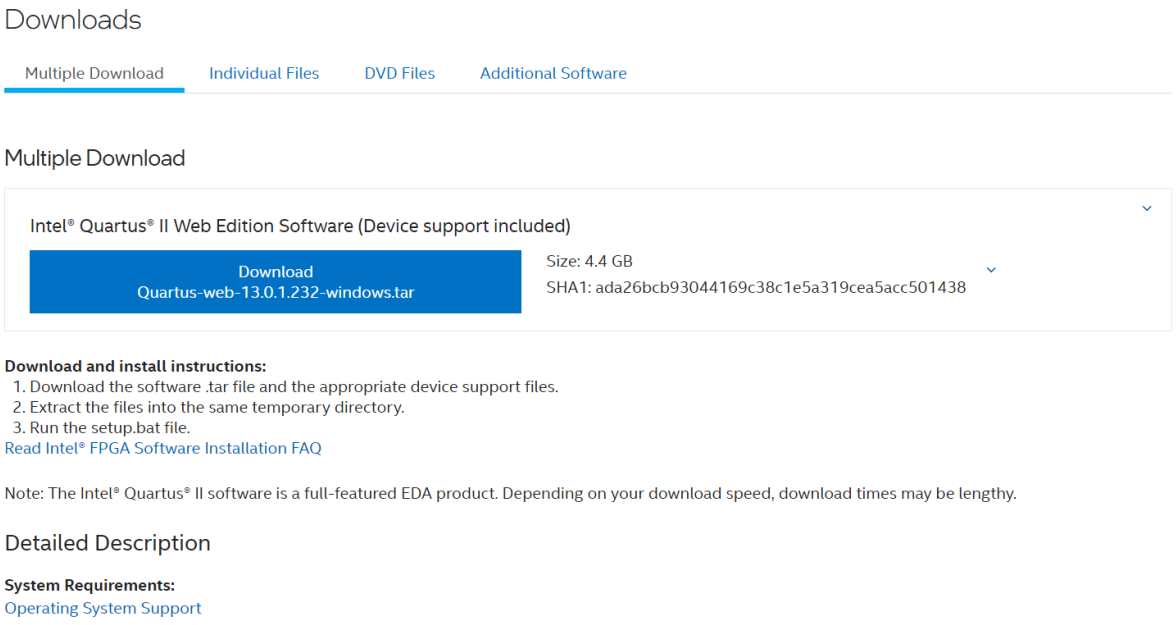


Figura 3.2 Enlace de descarga de Quartus II 13.0sp1

Por último, después de hacer clic en el botón de descarga, la página de Intel abrirá a una tercera página donde se debe aceptar un acuerdo de licencia, mostrado en la Figura 3.3, pero al ser una versión web no requiere de un registro. Al hacer clic en aceptar, la descarga comenzará y puede tardar unos cuantos minutos debido a su peso de 4.4 GB.

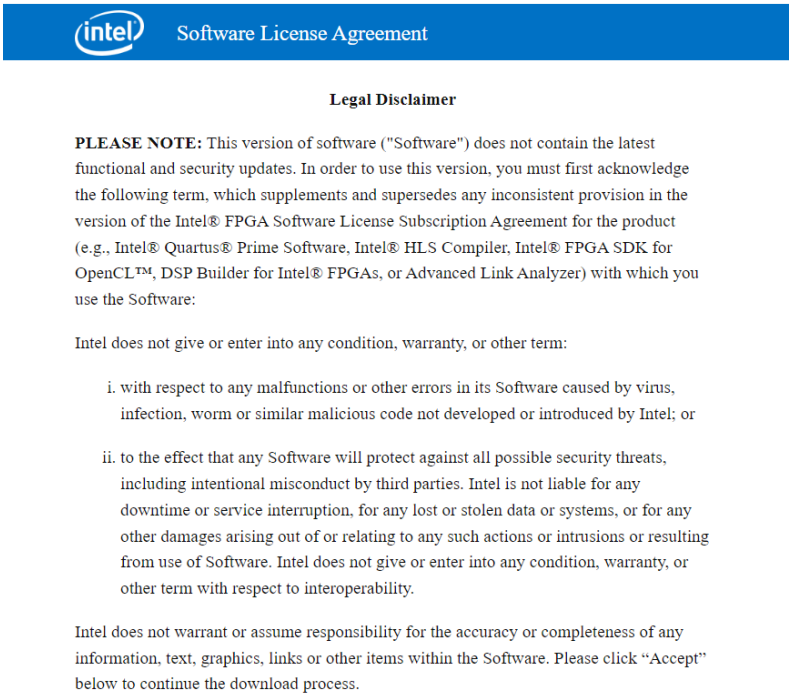


Figura 3.3 Acuerdo de licencia.

Una vez terminada la descarga, se debe descomprimir el archivo *.rar* y posteriormente ejecutar el instalador "QuartusSetupWeb-13.0.1.232.exe". La Figura 3.4 muestra la interfaz del instalador. Habrá que seguir los pasos para concluir la instalación del software

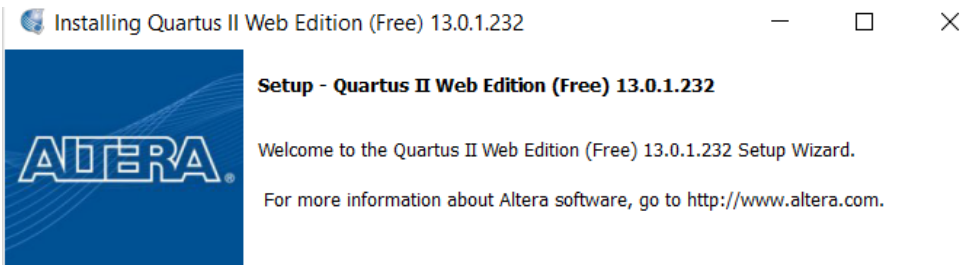


Figura 3.4 Sección de la ventana del instalador de Quartus II 13.0sp1.

3.2 CREACIÓN DE PROYECTOS

Cuando se haya terminado la instalación, el siguiente paso será ejecutar el programa haciendo clic en el acceso directo que se coloque en el escritorio con el logo de Quartus II.

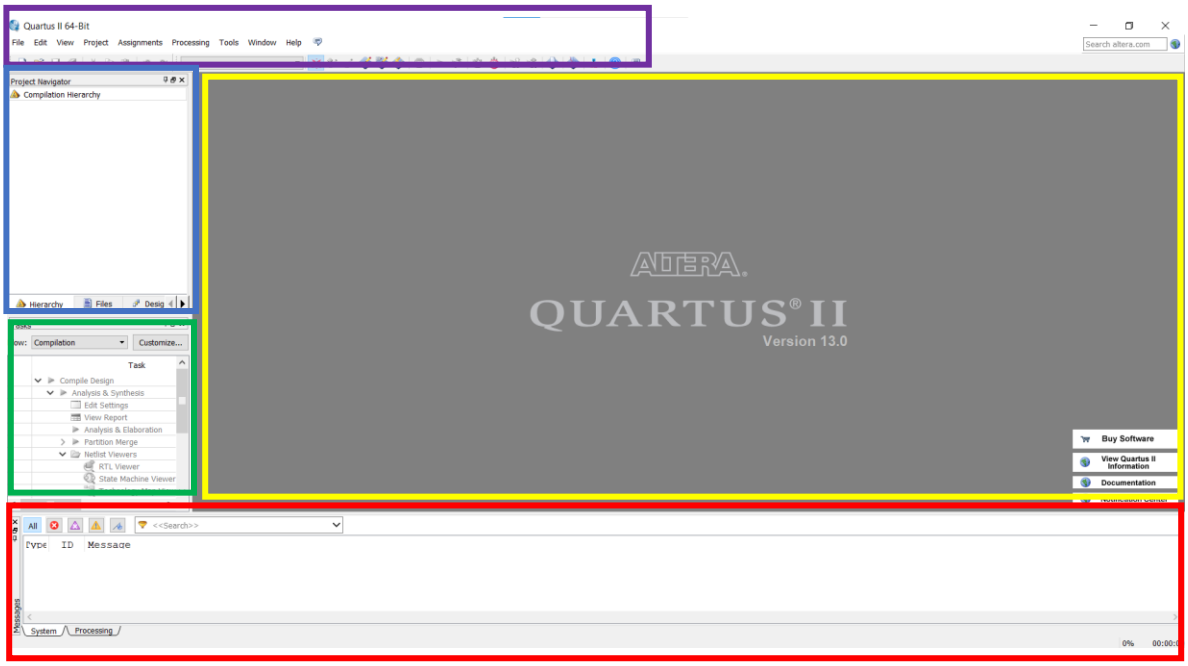


Figura 3.5 Página principal de Quartus II.

En la Figura 3.5 se aprecia la vista de la página principal del software sonde encontramos 5 secciones. En la barra de herramientas (morado) se encuentran las opciones de crear, guardar y abrir archivos. Navegador del proyecto (azul) se puede visualizar los archivos que conforman el proyecto y sus jerarquías. En el espacio de trabajo (amarillo) se situarán las pestañas que permiten escribir y editar los archivos del código fuente. En la lista de procesos (verde) se puede visualizar el estado en que se encuentra el proyecto referido a la compilación, en esta sección de la ventana es donde se muestra si el proyecto requiere ser compilado o si el proceso ya ha finalizado. Finalmente, la barra de mensajes (rojo) permite visualizar advertencias y errores que surjan durante la compilación del proyecto.

Las diferentes partes de la ventana del software resaltadas en la Figura 3.5 permitirán crear proyectos y, por lo tanto, generar código para programar un FPGA. Se recomienda seguir los siguientes pasos:

El primer paso será buscar la pestaña “File” en la barra de herramientas, que además se muestra en la Figura 3.6, después seleccionar busque la opción “New Proyect Wizard” y se abrirá una nueva ventana.

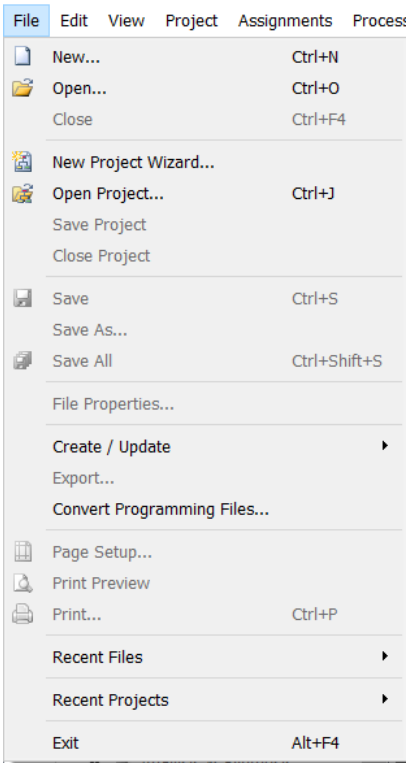


Figura 3.6 Barra de herramientas

El segundo paso será crear una carpeta, ya sea en el escritorio o en cualquier parte de la computadora. Después se debe seleccionar dicha carpeta y posteriormente asignar un nombre al proyecto. Este proceso se hará en la ventana que muestra la Figura 3.7. Una buena práctica es que el proyecto y la carpeta tengan el mimo nombre.

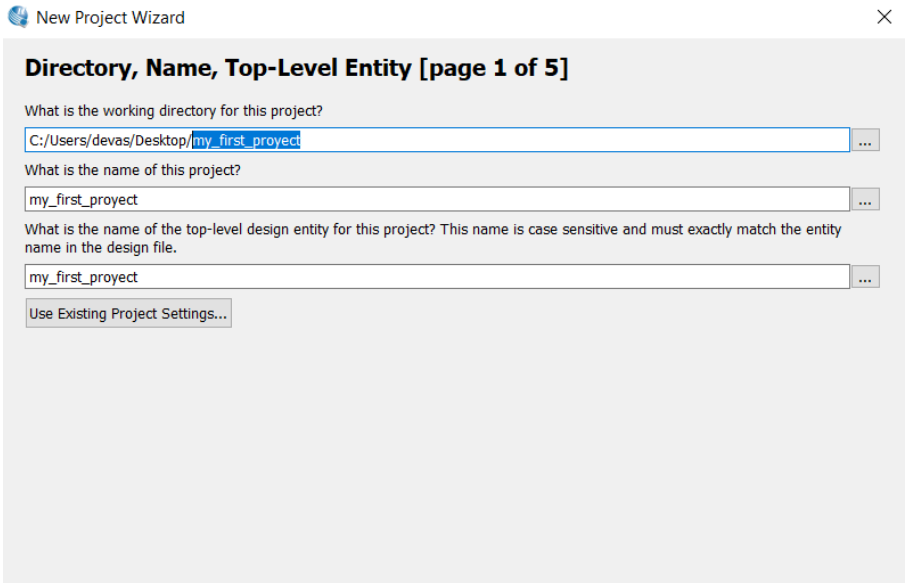


Figura 3.7 Ventana de creación del proyecto.

El tercer paso será seleccionar el FPGA para programar, la Figura 3.8 muestra la ventana con la lista de los FPGA Cyclone, este manual contempla solo la familia Cyclone II y Cyclone IV. En el apartado “Family” se puede seleccionar cualquiera de las dos familias antes mencionadas.

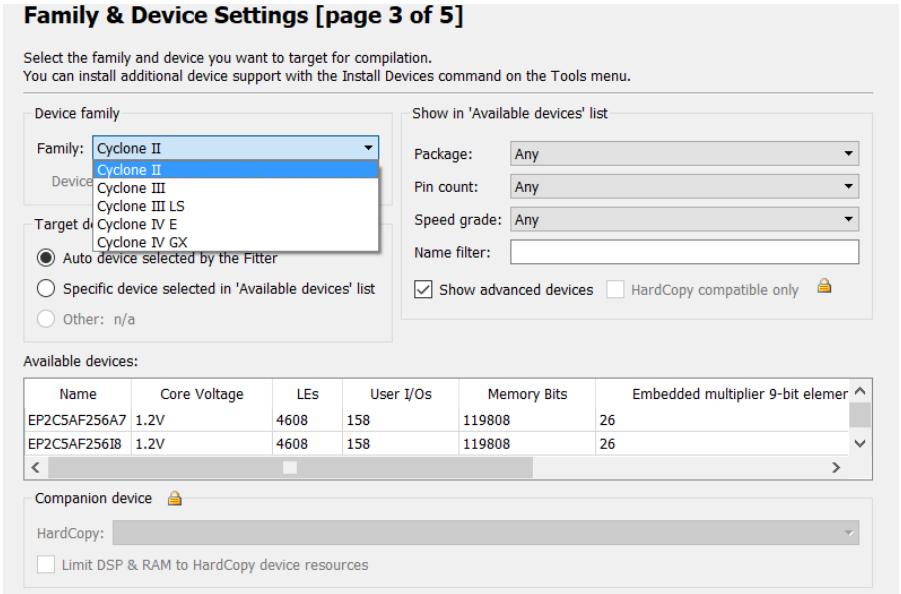


Figura 3.8 Selección de la familia del FPGA

En la lista inferior de la Figura 3.9, se muestran los modelos de cada familia de FPGA. En la lista inferior se selecciona el modelo del FPGA. Para este manual solo puede seleccionar la **EP2C5T144 (**Cyclone II**)** o la **EP4CE6E22C8N** (Cyclone IV).

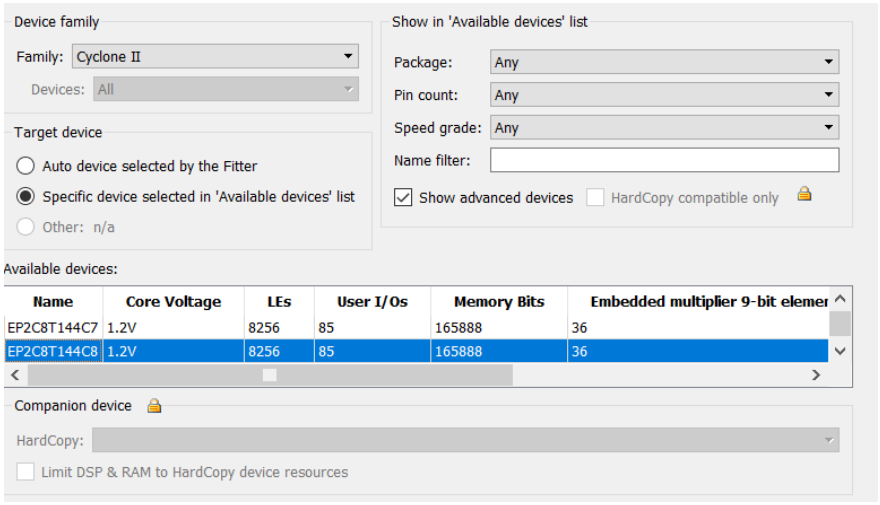


Figura 3.9 Selección del modelo de FPGA.

Se debe hacer clic en el botón “Next” hasta cerrar la ventana y finalmente pueda visualizar el botón “Finish” y por último hacer clic en él. Cuando termine el proceso se mostrará la ventana de la Figura 3.10.

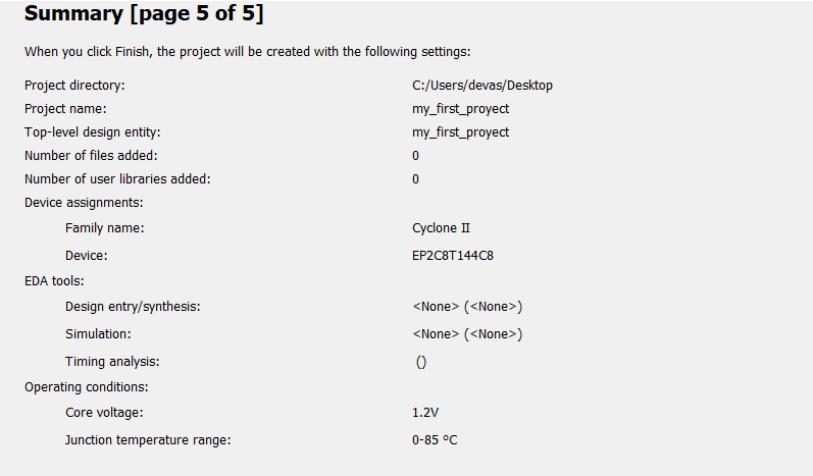


Figura 3.10 Paso final de la creación del proyecto

Finalmente, se observará en el árbol de operaciones que el proyecto se ha creado, aunque en primera instancia se encontrará vacío, como se observa en la Figura 3.11. En la sección siguiente se mostrará el proceso de creación de los archivos que contienen el código para programar el FPGA.

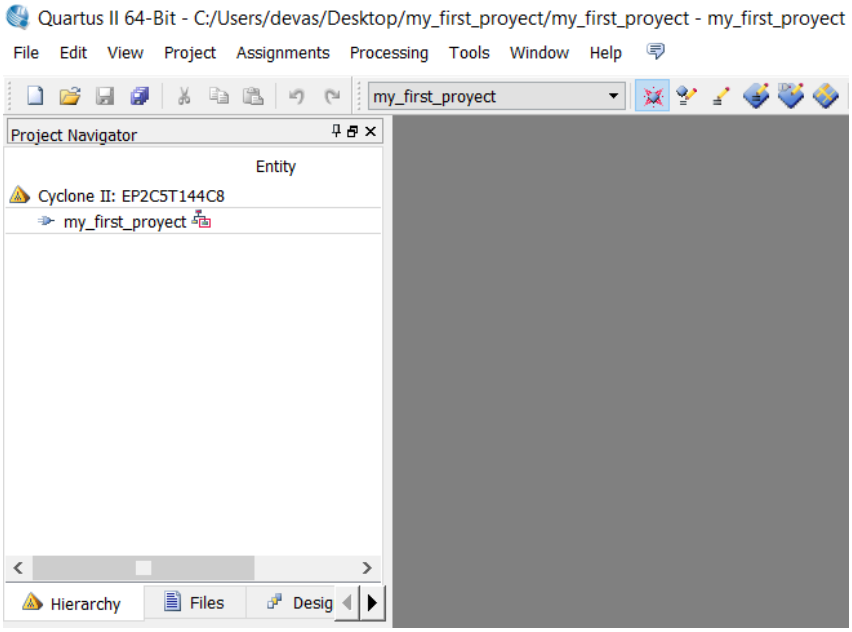


Figura 3.11 Proyecto vacío.

3.3 CREACIÓN DE ARCHIVOS VHDL

Ahora que el proyecto está creado, se procede a generar el archivo *.vhd* el cual permitirá escribir el código fuente en lenguaje VHDL. Este archivo debe tener el mismo nombre que el proyecto debido a que Quartus II rige a los archivos por jerarquías y requiere de un archivo principal que haga funcionar el proyecto, aunque puede crear tantos archivos *.vhd* como sean necesarios para simplificar el código que representen el circuito digital. Para generar el archivo, se debe buscar y seleccionar la opción “new” en la pestaña “File” como lo muestra la Figura 3.12.

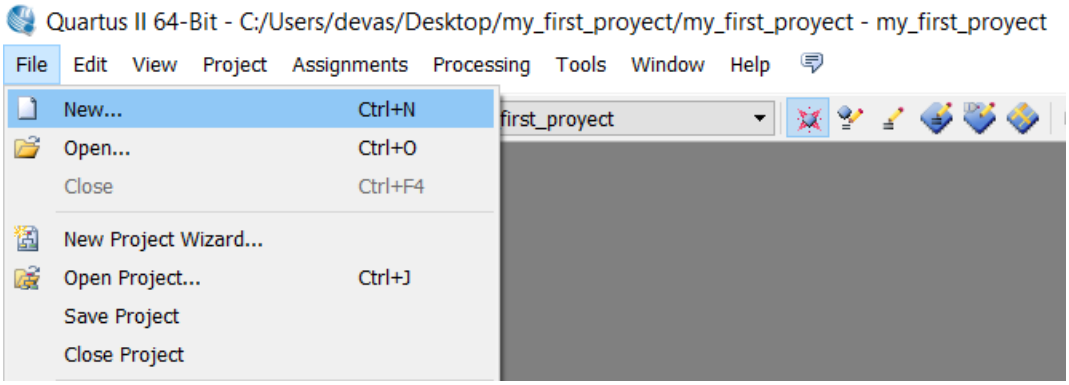


Figura 3.12 Creación de un nuevo archivo en el proyecto.

Al hacer clic en la opción indicada se mostrará la ventana ilustrada en la Figura 3.13, se trata de una ventana emergente la cual contiene una lista de los tipos de archivos que puede crear en Quartus II, la opción que se requiere es “VHDL File”.

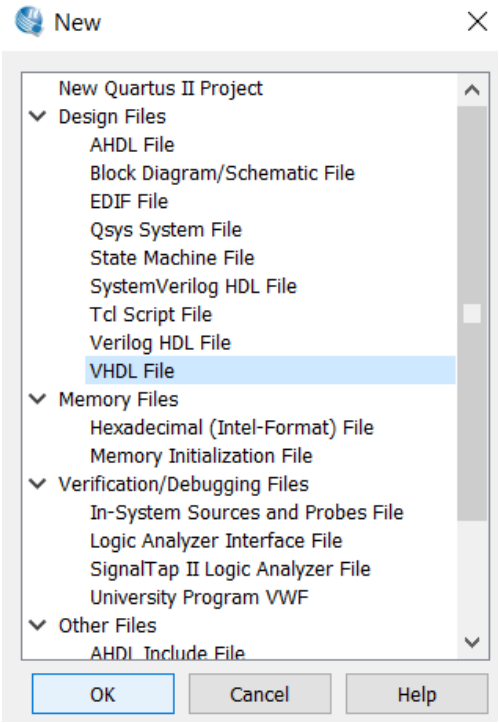


Figura 3.13 Creación de archivos .vhd para programar en VHDL.

Una vez creado el archivo *.vhd*, la pantalla principal permitirá visualizar el contenido del archivo que obviamente se encuentra vacío tal como lo muestra la Figura 3.14.

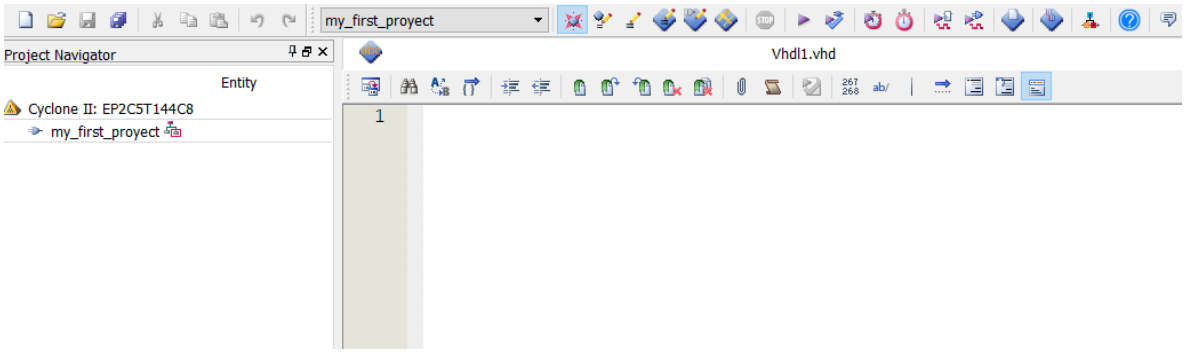


Figura 3.14 Archivo .vhd vacío.

Después, se hace clic en la opción “guardar como”, situada dentro de la pestaña “Files”, y para guardar se tendrá que asignar él mismo nombre del proyecto al archivo. Por último, ee abrirá una ventana emergente, como lo muestra la Figura 3.15, donde se indica la dirección de la carpeta creada en la sección anterior. Finalmente se hace clic en la opción guardar.

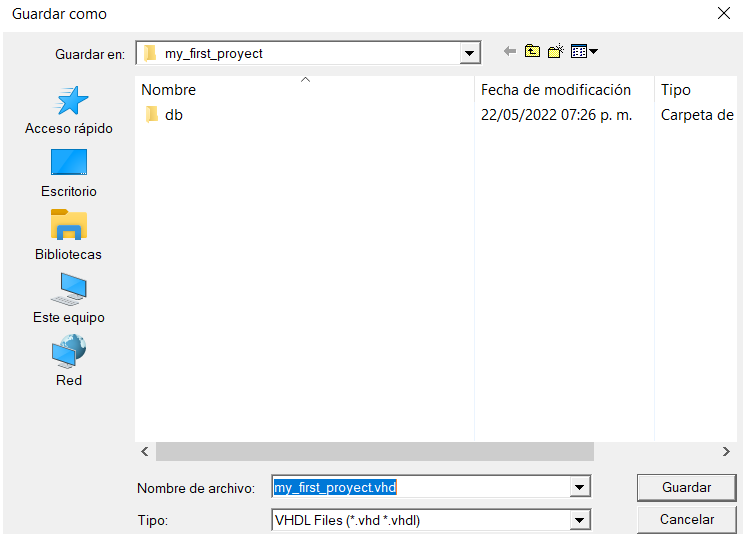


Figura 3.15 Guardado del archivo.

3.4 DIAGRAMAS DE BLOQUES

El software de Quartus II tiene la opción de programar los FPGA mediante el uso de diagramas de bloques, donde cada bloque trae consigo una parte del código en VHDL del proyecto. Esta herramienta permite programar de una manera más sencilla, ya que el usuario puede tener un mejor control de la estructura del proyecto para programar el FPGA y las líneas de código utilizadas se reducen de manera significativa. Esta técnica es recomendable cuando se requiere utilizar más de un archivo *.vhd*

3.4.1 CREACIÓN DE PROYECTOS CON DIAGRAMA DE BLOQUES

El primer paso para crear un proyecto usando los diagramas de bloques es crear el espacio de trabajo donde estarán alojados todos los bloques, conexiones, entradas y salidas. Para ello, se crea el proyecto siguiendo los pasos de la sección 3.2 y, una vez terminado, en la sección "Files" se busca la opción "New" y al abrirse la nueva ventana se selecciona Block Diagram / Schematic File como lo muestra la Figura 3.16.

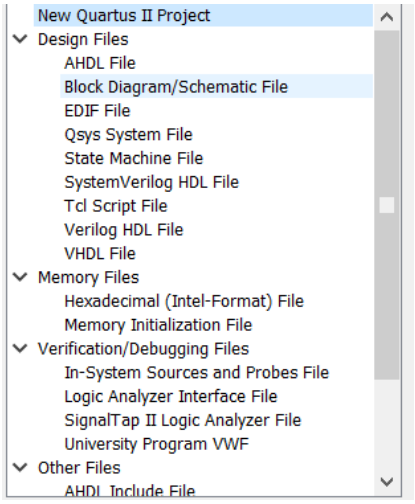


Figura 3.16 Opción para crear un diagrama de bloques.

Se creará un archivo con terminación *.bsd* que el cual contiene el diagrama de bloques del proyecto, inicialmente no tendrá un nombre asignado por lo que aparecerá por defecto

como “block1.bdf”. Regresando a la ventana principal, el archivo se visualizar como lo muestra la Figura 3.17 donde evidentemente estará vacío.

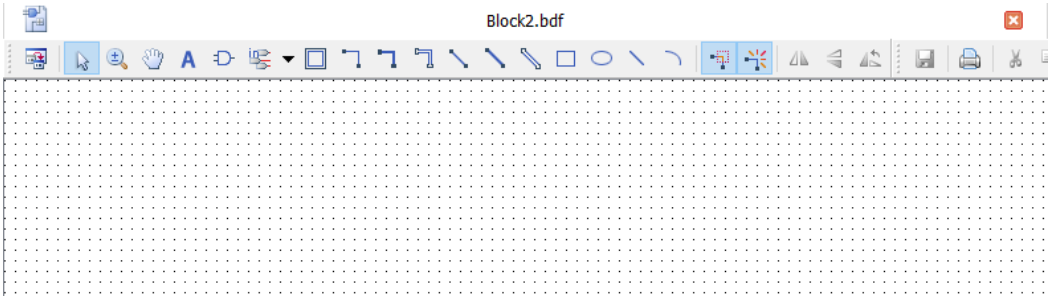


Figura 3.17 Archivo “.bdf” vacío.

El segundo paso será asignar un nombre al archivo. De la misma manera que los archivos *.vhd*, en el software de Quartus II un archivo *.bdf* puede convertirse en la entidad principal del proyecto y en él se pueden añadir un sin número de bloques y código VHDL. Para ello, en la pestaña “File” se selecciona la opción “Save as” y por defecto se añadirá el nombre que fue asignado al proyecto. Automáticamente el archivo *.bdf* se guarda en la carpeta donde se aloja el proyecto como lo muestra la Figura 3.18 y finalmente se hace clic en el botón guardar.

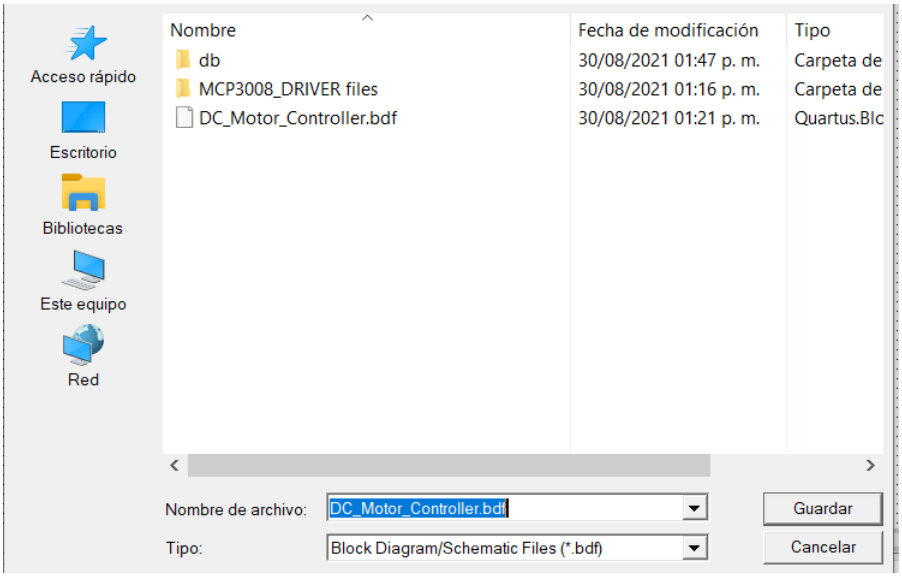


Figura 3.18 Guardado de diagrama de bloques.

Una vez terminado este proceso, el espacio de trabajo estará listo para crear el diagrama de bloques que representará el funcionamiento del FPGA.

3.4.2 CREACIÓN DE BLOQUES USANDO CÓDIGO VHDL

En la sección anterior se creó un espacio de trabajo dentro de un archivo *.bdf* que contiene el diagrama de bloques que se usarán para programar el FPGA. Pero para armar dicho diagrama se debe crear cada uno de los bloques que lo compone. En Quartus II los archivos *.bsf* almacenan de manera gráfica los bloques que se usan para elaborar el diagrama y siempre están acompañados de un archivo *.vhd* que contiene el código que describe el funcionamiento del bloque así como sus entradas y salidas. Para crear un bloque basado en su código en VHDL se usa la siguiente serie de pasos:

Ubicando la carpeta donde se alojar el proyecto, el primer paso será guardar los archivos *.vhd* que describen el funcionamiento de los diagramas de bloques. La Figura 3.19 muestra una carpeta llamada “Ejemplo” que contiene tres archivos VHDL donde cada uno generará un bloque distinto.

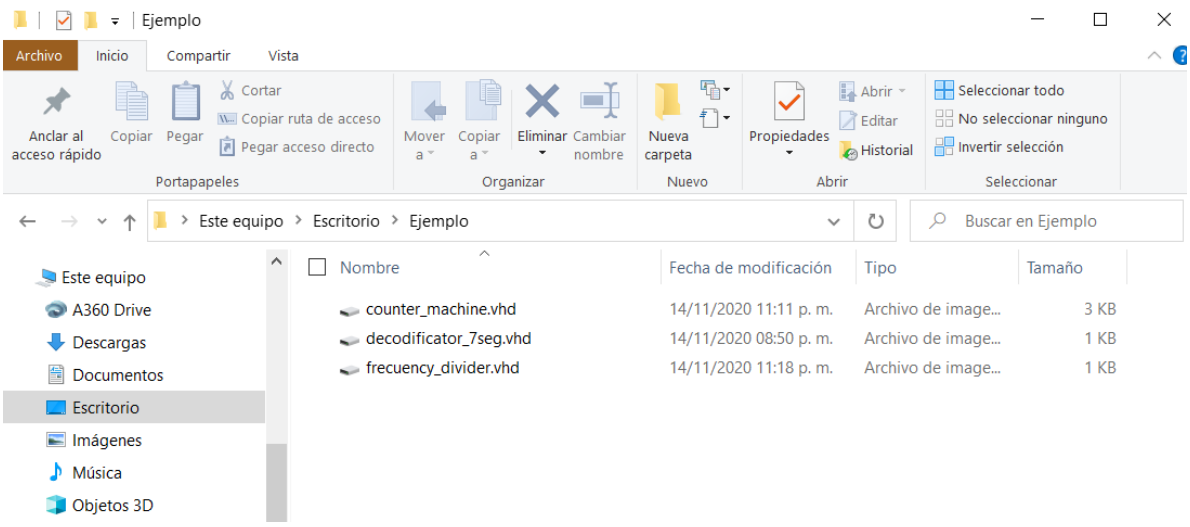


Figura 3.19 Archivos VHDL que serán convertidos a bloques.

El segundo paso es crear un proyecto configurado para diagrama de bloques como se explicó en la sección anterior.

El tercer paso es incluir el archivo (o los archivos) *.vhd* que se encuentran dentro de la carpeta creada en paso anterior. Para incluirlos en el proyecto se debe hacer clic en la pestaña “Files” situada debajo del árbol de operaciones, y después en la parte superior del cuadro se hace clic derecho sobre “Files” y se selecciona la opción “Add/Remove files in proyect …” tal como lo muestra la Figura 3.20.

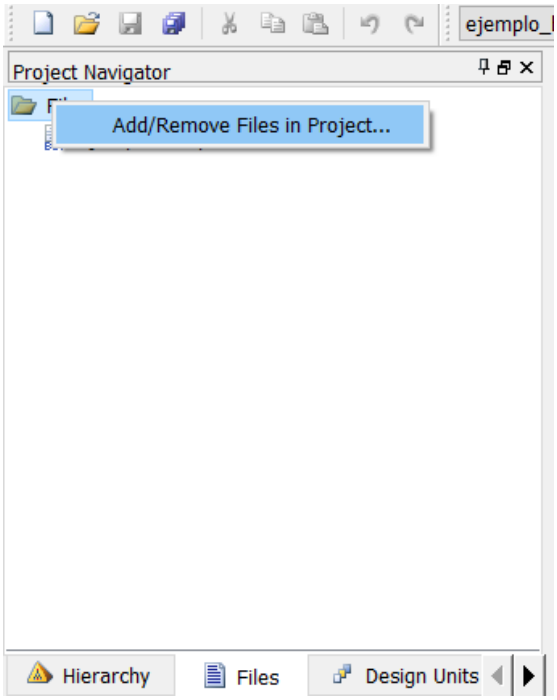


Figura 3.20 Navegador del proyecto vacío.

Después se desplegará una nueva ventana como lo ilustra la Figura 3.22 cuyo contenido principal es una lista que contiene los archivos incluidos en el proyecto. Al inicio la lista estará vacía ya que no existe ningún archivo que esté asociado al proyecto. Para que los archivos VHDL estén dentro del proyecto, se tendrá que hacer clic en el botón “…” ubicado arriba de la lista y se desplegará una ventana emergente como lo muestra la Figura 3.21 que muestra la carpeta donde se encuentra el proyecto y los archivos que se incluyeron al inicio de este proceso. Se selecciona todos los archivos *.vhd* y luego se hace clic en abrir.

Por último, cuando se haya cerrado la ventana de la Figura 3.21, en la lista de la Figura 3.22 se incluirán a los archivos en el proyecto. Finalmente, se debe hacer clic en “Add all”, luego en “Apply” y por último en “Ok”.

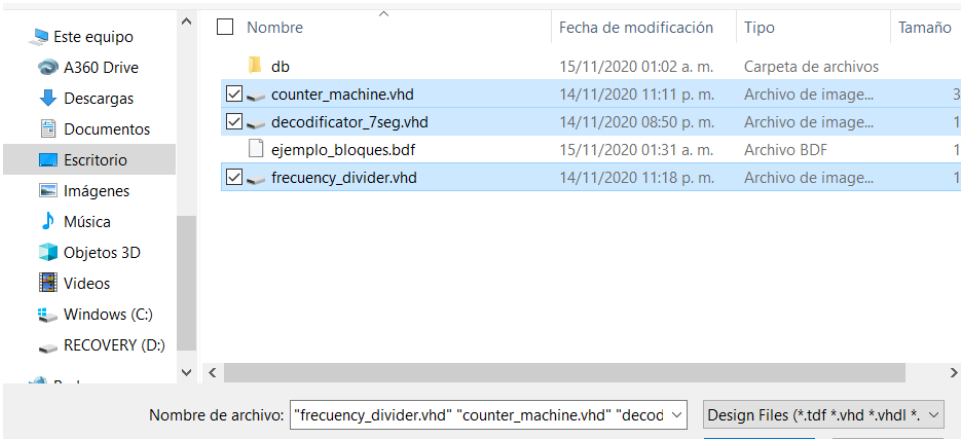


Figura 3.21 Selección de los archivos VHDL.

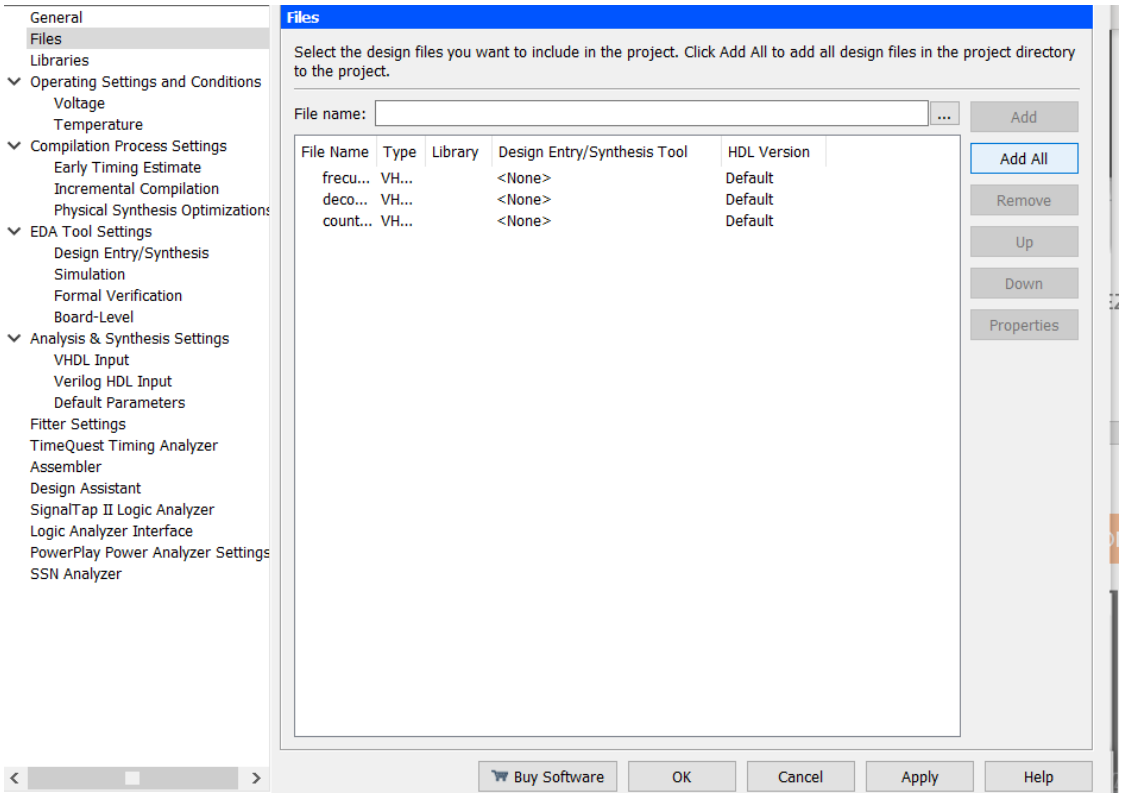


Figura 3.22 Selección de los archivos VHD

Ahora se podrá apreciar una lista donde se observa los archivos que conforman el proyecto, como se muestra en la Figura 3.23 que corresponde al navegador de archivos de Quartus II.

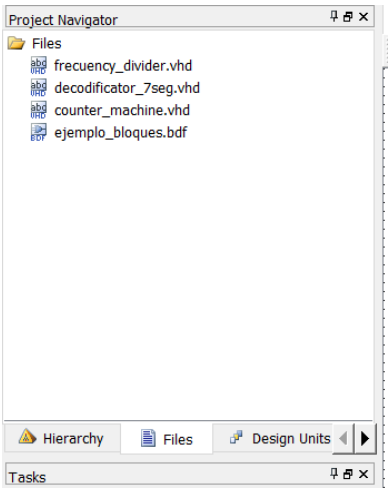


Figura 3.23 Navegador del proyecto con los archivos VHDL incluidos en el proyecto.

El cuarto paso será abrir uno de los archivos dentro del software y después seleccionar la pestaña “Files”, después se hace clic en la sección “Create” y se selecciona la opción “Create Symbol Files for Current Files”. Repetir el proceso para cada uno de los archivos.

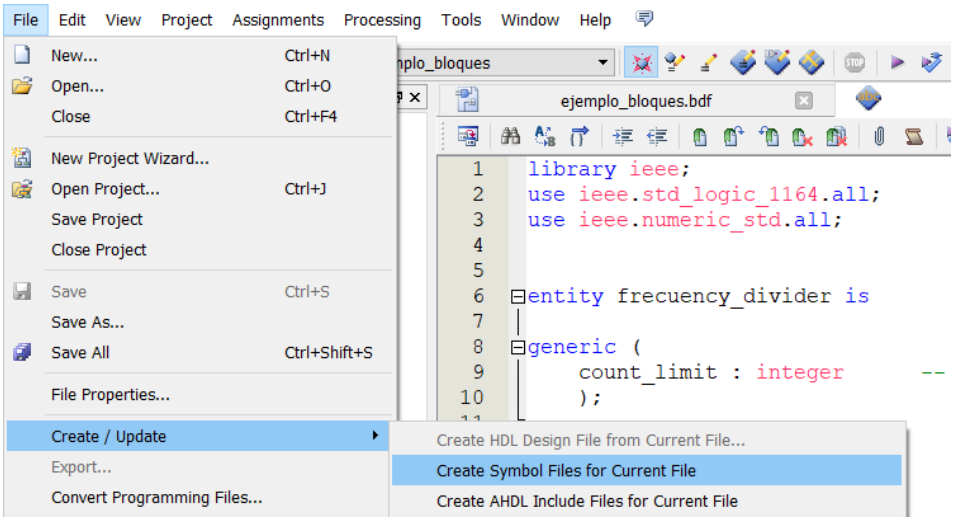


Figura 3.24 Selección de la opción para crear archivos de bloques .bsf.

Cuando se hace clic en “Create Symbol Files for Current Files” se desplegará una ventana que informa cuando el archivo se ha creado exitosamente y la barra de mensajes del compilador mostrará mensajes de éxito, tal como se observa en la Figura 3.25. Si existe un error en los códigos de cada uno de los archivos se indicará con mensajes y se deberá corregir para que el archivo del diagrama de bloques pueda crearse. Se repite este paso para cada uno de los archivos *.vhd.*

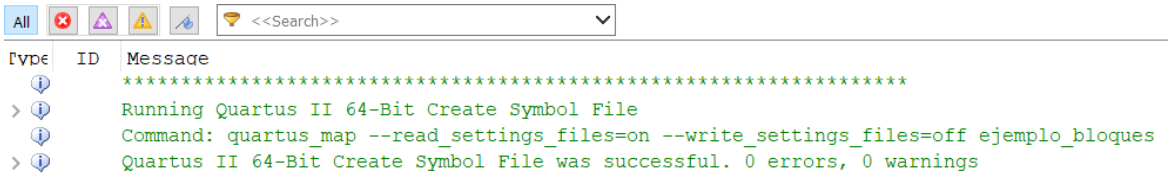


Figura 3.25 Barra de mensajes con compilación exitosa.

Para el quinto paso, una vez creado el diagrama de bloques para cada archivo y ubicando la barra de herramientas de la Figura 3.26, se selecciona la opción “Symbol Tool” situado en la barra superior de la ventana.

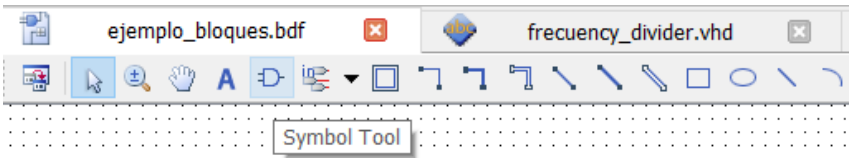


Figura 3.26 Selección de "Symbol Tool" para agregar bloques al proyecto.

Se desplegará una ventana correspondiente a la carpeta “Proyect” mostrada en la Figura 3.27 y allí se sitúan enlistados cada uno de los diagramas de bloques que corresponden a los códigos de los archivos *.vhd* creados en los pasos anteriores. Se muestran también las entradas y las salidas de cada bloque que corresponden a las entradas y salidas del código escrito en ese archivo.

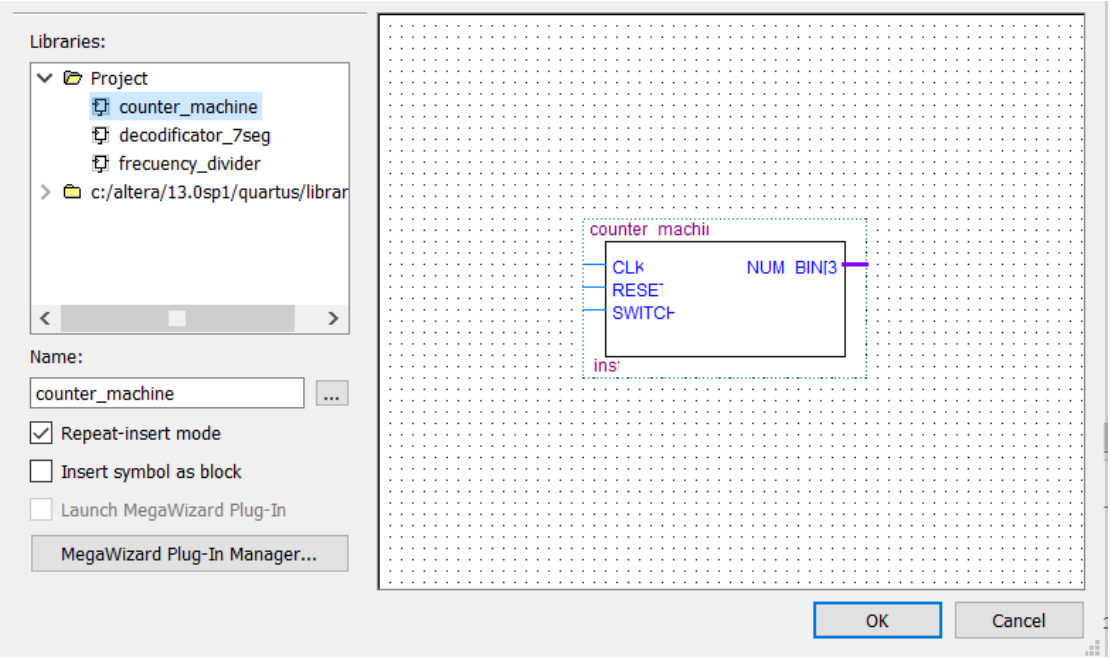


Figura 3.27 Vista previa de los bloques creados a partír de código VHDL.

Al hacer clic en “Ok” el cursor aparecerá el bloque seleccionado y se podrá situar en cualquier parte de la ventana del diagrama de bloques. La Figura 3.28 muestra el bloque con una transparencia debido a que el cursor indica que ese bloque esta seleccionado, cuando se haga clic en cualquier parte del diagrama, el bloque pasará a tener un color sólido y significa que el bloque ya formará parte del diagrama.

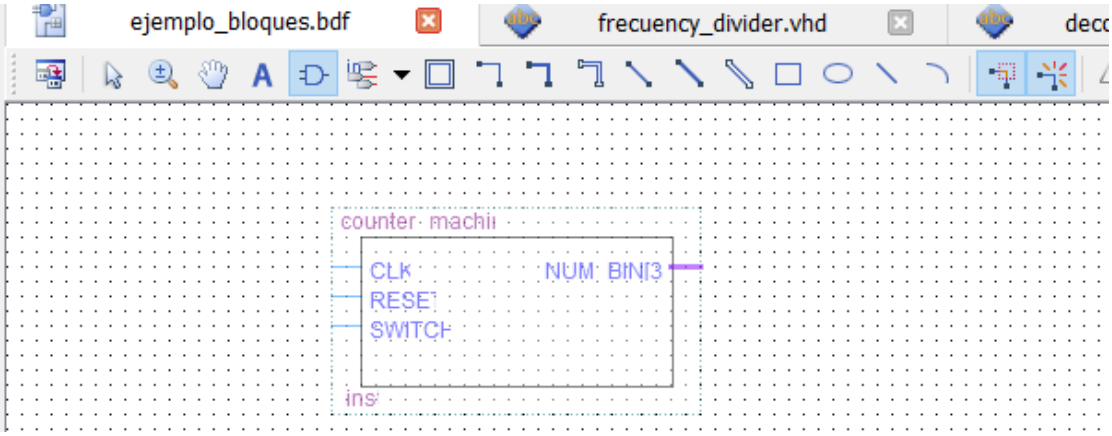


Figura 3.28 Bloque creado a partír de código VHDL.

3.4.3 CONSTRUCCIÓN DEL DIAGRAMA DE BLOQUES

Existen dos tipos de uniones entre bloques los cuales corresponden con: la unión por señal digital simple representada por a un *std_logic* en código VHDL; la unión por vector de bits, que es un conjunto de señales lógicas (ya sea de entrada o salida) del tipo de dato *std_logic_vector*. Ambos se distinguen principalmente porque la línea de la señal digital simple es más delgada y la unión del vector de bits tiene una línea más gruesa. En la Figura 3.29 se marcan resaltan las herramientas para unir los buses de datos entre los bloques del diagrama.

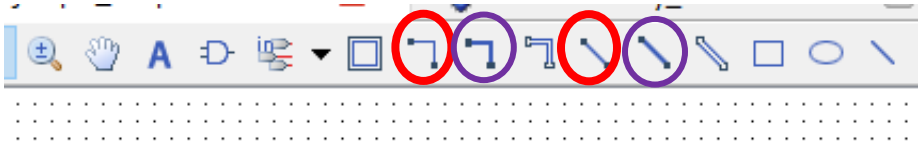


Figura 3.29 Opciones para unir los bloques. Para señales de solo bit (rojo) y vectores de bits (morado).

Con esta información, los bloques pueden interconectarse del proyecto, respetando que las señales simples se unen con otras señales simples y, de la misma manera, con los vectores de bits. En la Figura 3.30 ejemplifica la unión por un bit simple entre "frecuency_divider" y "counter_machine", mientras que la unión de éste último con "decodificator_7seg" es un vector de 4 bits.

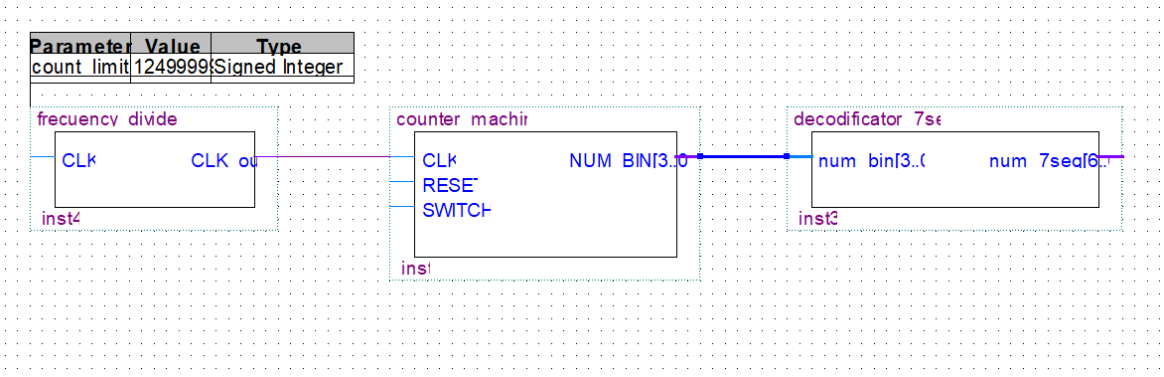


Figura 3.30 Unión de los buses de datos entra cada bloque.

Al igual que en código VHDL, se deben asignar las entradas y las salidas que corresponden al proyecto en general.

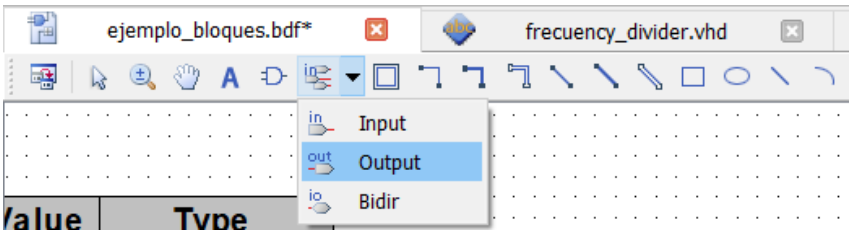


Figura 3.31 Opción para añadir entradas y salidas al proyecto.

La opción *"Pin Tool"*, mostrada en la Figura 3.31, permite al proyecto tener los pines de entrada y salidas necesarios y posteriormente asigne un nombre a cada uno de ellos. Para ello, se hace doble clic sobre el pin de entrada (o de salida) y se desplegará una ventana donde el siguiente paso será asignar un nombre a la entrada o salida tal como lo muestra la Figura 3.32.

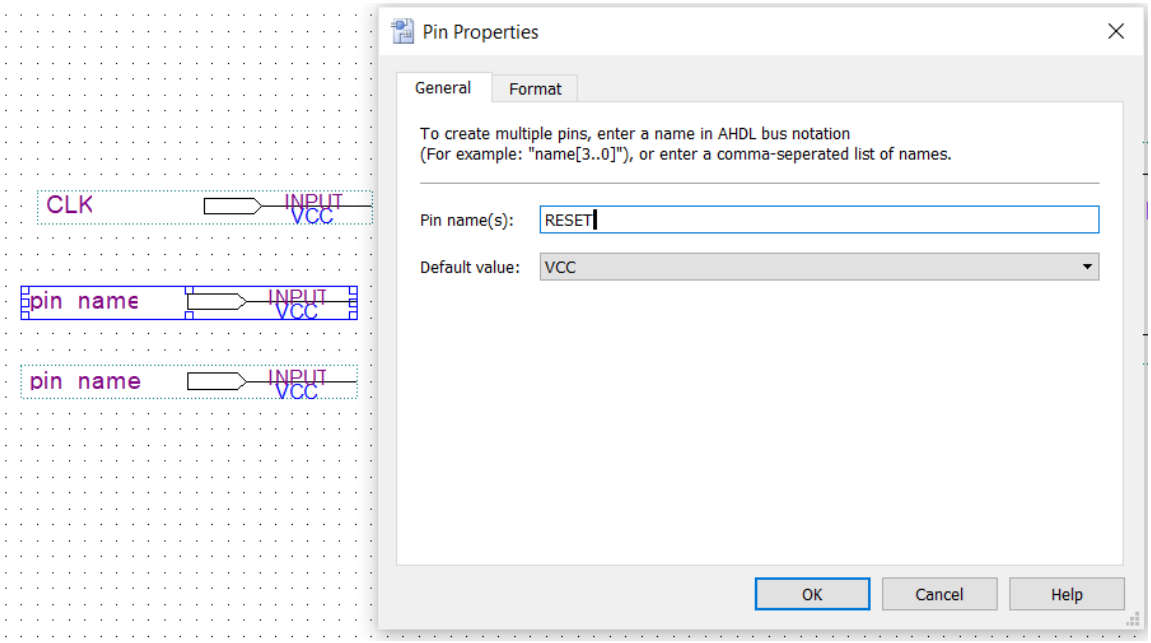


Figura 3.32 Asignación de nombre a un pin.

En el caso de que los pines correspondan a un arreglo de bits como se ha mencionado anteriormente se requerirá contemplar el número de elementos que contiene y escribir su nombre de la siguiente forma:

(nombre)[(no. De elementos -1) .. 0]

Siguiendo la estructura presentada anteriormente, en la ventana de la Figura 3.33 se escribe la estructura presentada para una salida de 7 bits.

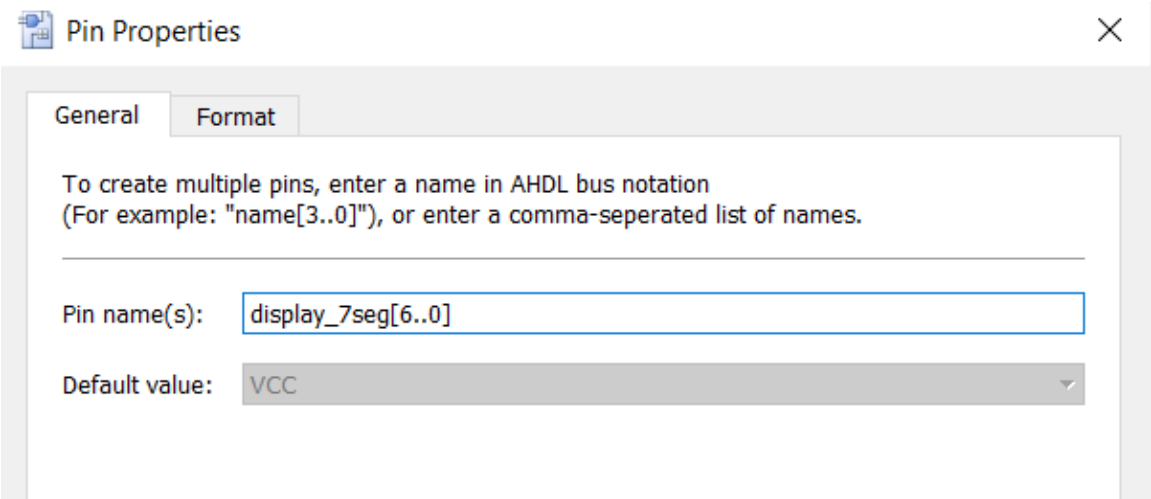


Figura 3.33 Asignación de un nombre a una señal de 7 bits.

Posteriormente se unen los pines de entrada y salida con el resto de los bloques.

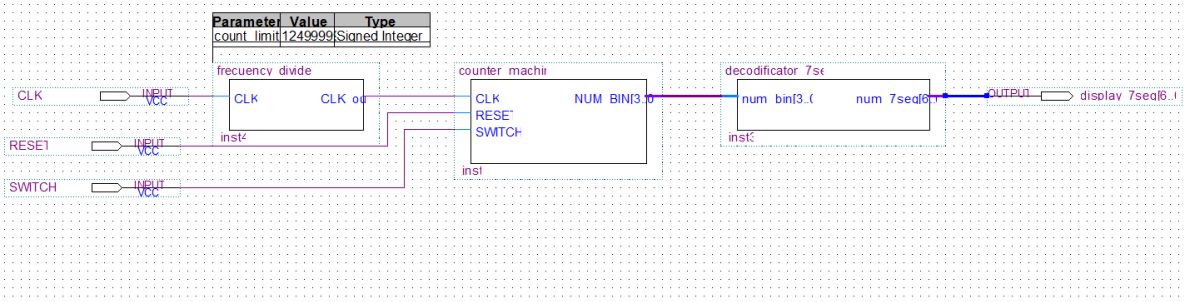


Figura 3.34 Creación y unión de los pines de entrada y salida al diagrama de bloque

De esta forma queda conformado el diagrama de bloques de todo el proyecto el cual puede compilarse sin ningún problema, y también asignar los pines físicos del FPGA a los pines de entrada y salida del diagrama.

Como se mencionó al inicio de esta sección, el objetivo de organizar el código con bloques e implementar un diagrama, es tener una mejor organización del proyecto y tener la menor cantidad de líneas de código en cada uno de los archivos *.vhd* que conforman el proyecto.

3.5 COMPILACIÓN DE PROYECTOS

Sin importar que se trate de código VHDL o diagramas de bloques explicados en las secciones pasadas, el proceso para compilar el proyecto es el mimo. La compilación genera los archivos de programación de los FPGA y se revisan los errores en código o el proyecto. Para ejecutar esta función se ubica el botón *"Start Compilation"* en la barra de herramientas como lo muestra la Figura 3.35.

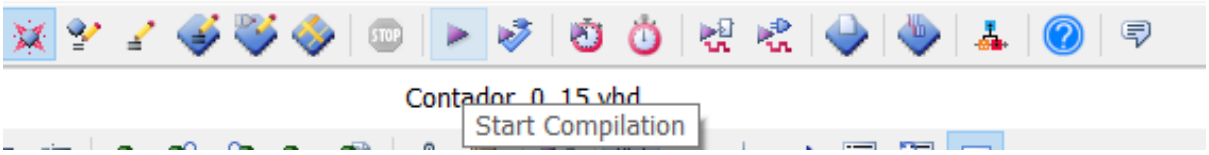


Figura 3.35 Selección de opción para compilar el proyecto.

Cuando se inicie la compilación del proyecto se podrá observar el porcentaje de avance en la lista de procesos representada en la Figura 3.36.

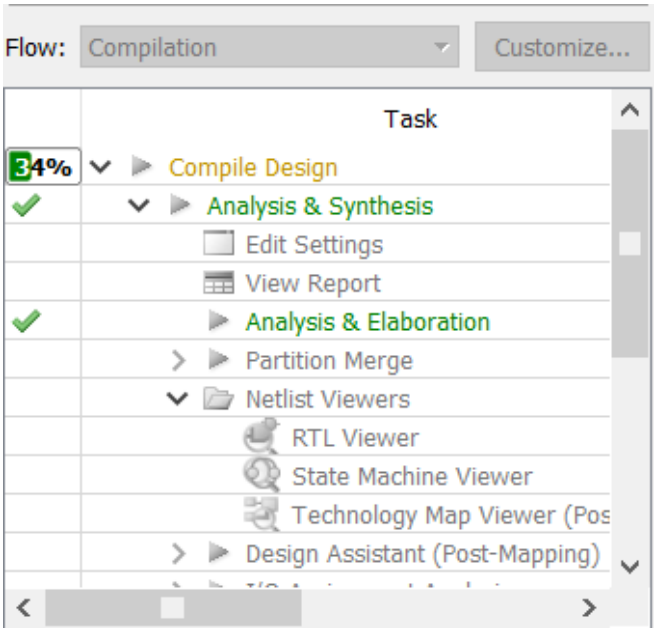


Figura 3.36 Barra de procesos indicando el porcentaje de avance de la compilación.

Una vez que se haya completado, el software indicará si la compilación fue exitosa. De lo contrario, si existen errores en el código, se pueden ubicar los mensajes de error en la barra de mensajes. En los mensajes de error se indica el posible error ya sea de sintaxis, incompatibilidad de variables, señales usadas, pero no declaradas, etc. Además del posible error, también indica la línea de código donde está ubicado. En la Figura 3.37 se ejemplifica un programa que tiene errores de sintaxis en el código donde en las líneas 16, 22 y 31 hace falta escribir el ";" al final de la línea de código.

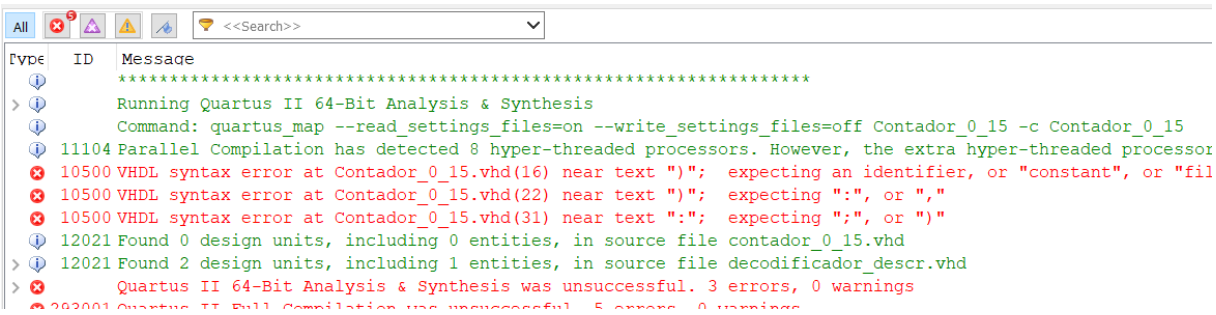


Figura 3.37 Barra de mensajes indicando errores en la compilación del proyecto.

3.6 CONFIGURACIÓN DE PINES DEL FPGA

La configuración de pines es un paso esencial para la programación de los FPGA debido a que se asocia un pin físico de este dispositivo a una variable de salida o entrada del código principal VHDL o del diagrama de bloques.

Para llevar a cabo la configuración de los pines, se selecciona la opción *"Pin Planner"* ubicado en la barra superior como lo muestra la Figura 3.38.

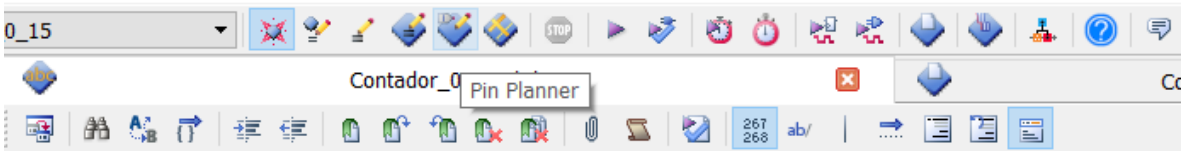


Figura 3.38 Selección de la opción "Pin planner" para configurar pines del FPGA.

Se desplegará la siguiente ventana en la cual puede observar la configuración de pines del FPGA. En el anexo A adjunta la documentación con la distribución de los pines disponibles de ambas tarjetas de desarrollo.

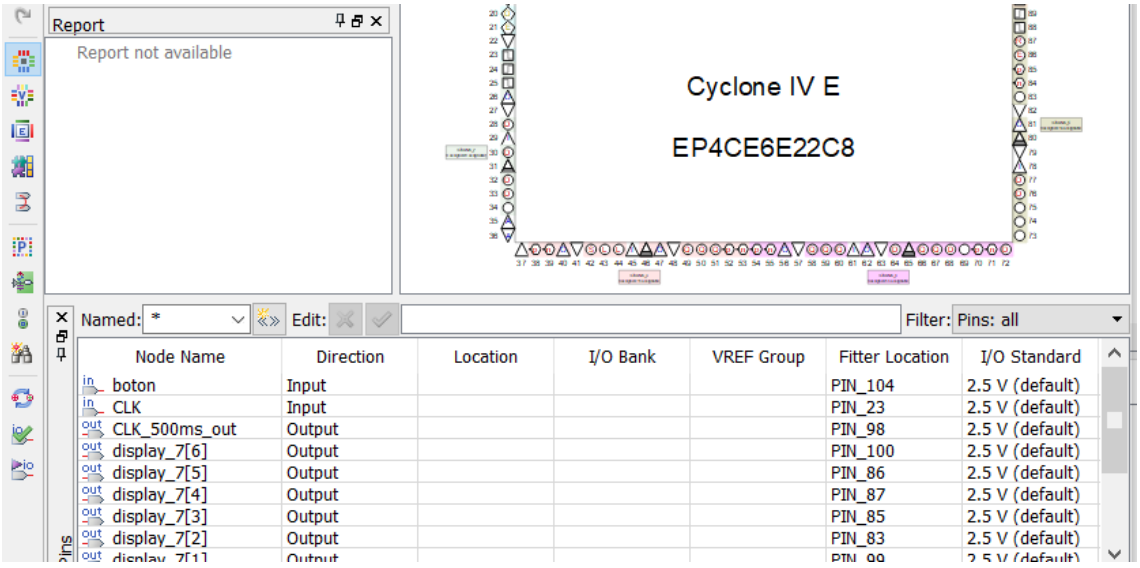
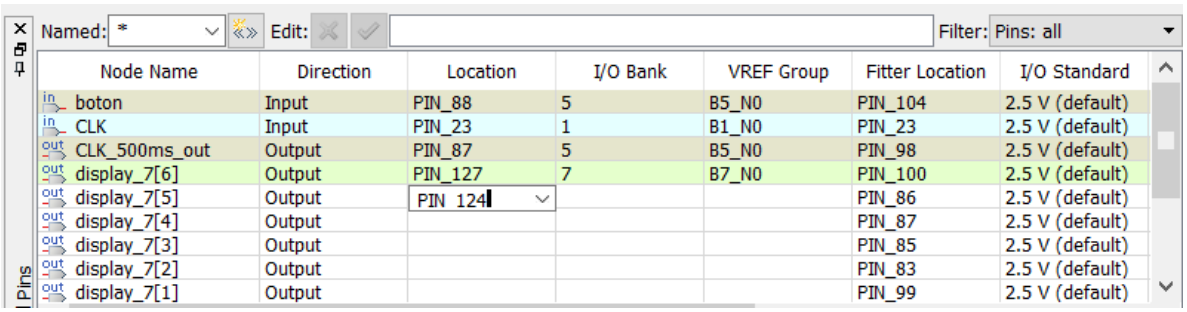


Figura 3.39 Vista previa de la configuración de pines de la Cyclone IV (arriba) y lista de las entradas y salidas del proyecto (abajo).

En Figura 3.39 muestra la organización de los pines físicos del FPGA Cyclone IV en la parte central de la imagen. Por otro lado, en la parte inferior de la ventana se encuentra una tabla que contiene cada una de las señales de entrada y salida son utilizadas en el código VHDL, o en el diagrama de bloques. Para asignar un pin a cada una de estas señales se debe dirigir el cursor en la columna *"Location"*, como lo ejemplifica la Figura 3.40, y se escribe en cada una de las celdas el número de pin que va a corresponder a la señal de la fila de la tabla. El pin se escribe de la forma: **PIN_(número del pin)** y se presiona la tecla *ENTER*.



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard
boton	Input	PIN_88	5	B5_N0	PIN_104	2.5 V (default)
CLK	Input	PIN_23	1	B1_N0	PIN_23	2.5 V (default)
CLK_500ms_out	Output	PIN_87	5	B5_N0	PIN_98	2.5 V (default)
display_7[6]	Output	PIN_127	7	B7_N0	PIN_100	2.5 V (default)
display_7[5]	Output	PIN_124			PIN_86	2.5 V (default)
display_7[4]	Output				PIN_87	2.5 V (default)
display_7[3]	Output				PIN_85	2.5 V (default)
display_7[2]	Output				PIN_83	2.5 V (default)
display_7[1]	Output				PIN_99	2.5 V (default)

Figura 3.40 Asignación de un pin físico a cada una de las entradas y salidas

Si en la programación del código VHDL se eliminan entradas o salida del código, o del diagrama de bloques, también se tendrán que eliminar de la lista de *"Pin planner"* mostrada en la Figura 3.40. Primero se tiene que compilar el código con los cambios hechos y después de la compilación, las entradas o salidas eliminadas ya no tendrán un pin físico asignado. En la lista de la Figura 3.40 se coloca el cursor sobre el nombre de la señal, hubicada en la columna "Node Name", después se hace doble clic y se presiona la tecla suprimir.

Por otro lado, si se requiere eliminar un pin físico asignado, pero sin eliminar señal de la lista, se hace un solo clic en el número de pin situado en la columna "Location" y luego se presiona la tecla suprimir.

3.7 SIMULACIÓN

Quartus II tiene una herramienta de simulación que permite simular el circuito digital diseñado en el proyecto, cuyo propósito es corroborar el correcto funcionamiento del programa, conocer su comportamiento y corregir errores de sintaxis que el compilador no es capaz de identificar.

3.7.1 CREAR ARCHIVO WAVEFORM

Para simular el proyecto que representa el circuito digital diseñado, se busqua la sección *"Files"* y después se selecciona *"new"*. En la ventana que se despliega en la pantalla, se hace clic en la opción *"University Program VWF"*.

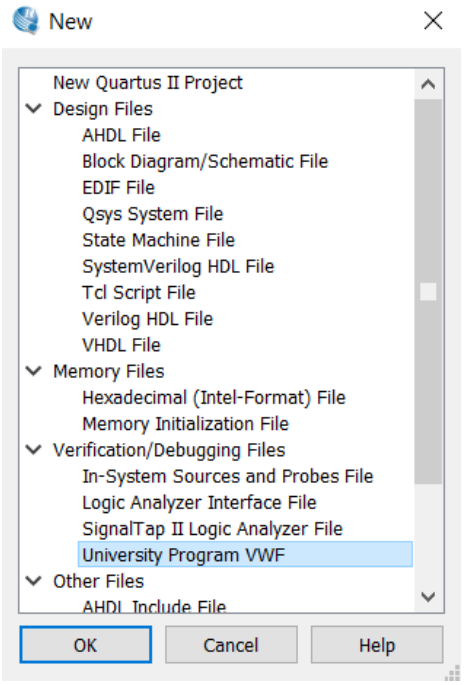


Figura 3.41 Creación del archivo de simulación.

Al mismo tiempo, se abrirá una ventana con el nombre *"Simulation Waveform Editor"*, mostrada en la Figura 3.42, que contiene una línea de tiempo la cual reportará los estados lógicos de cada señal en un punto específico del tiempo.

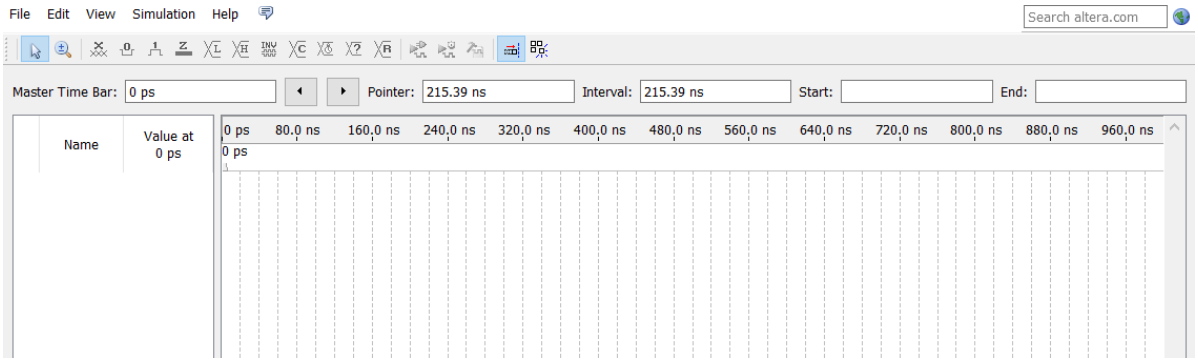


Figura 3.42 Ventana del simulador.

Antes de empezar a editar en el simulador, se guarda el nuevo archivo creado, para hacerlo se selecciona la pestaña *"Files"* y después la opción *"Save as"*. Por último, el archivo del simulador generado se debe guardar en la carpeta principal del proyecto, como lo muestra la Figura 3.43. El nombre que se asigna por defecto se puede conservar o cambiar.

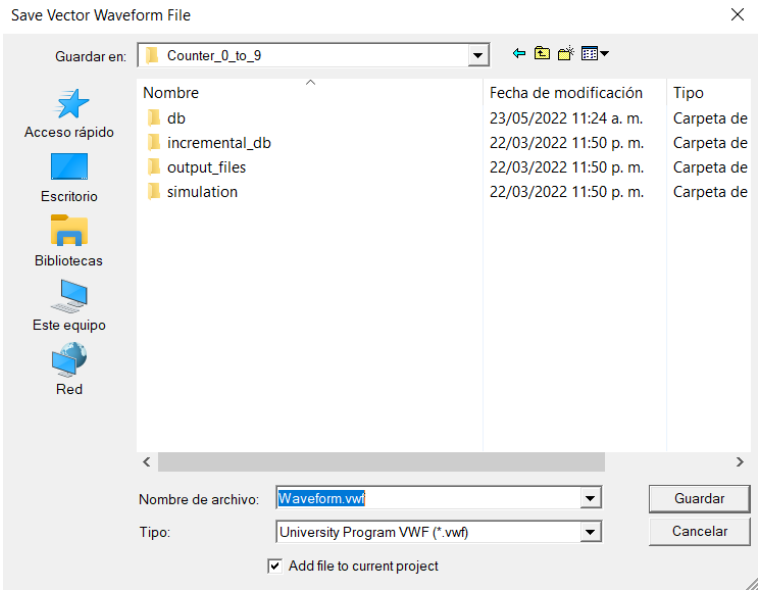


Figura 3.43 Guardado del archivo del simulador.

De regreso a la ventana principal de Quartus II, se observa en el navegador del proyecto que el nuevo archivo aparece junto al resto de los archivos de código, como lo resalta la Figura 3.44. Cuando se requiera abrir el simulador del proyecto, basta con hacer clic en el archivo.

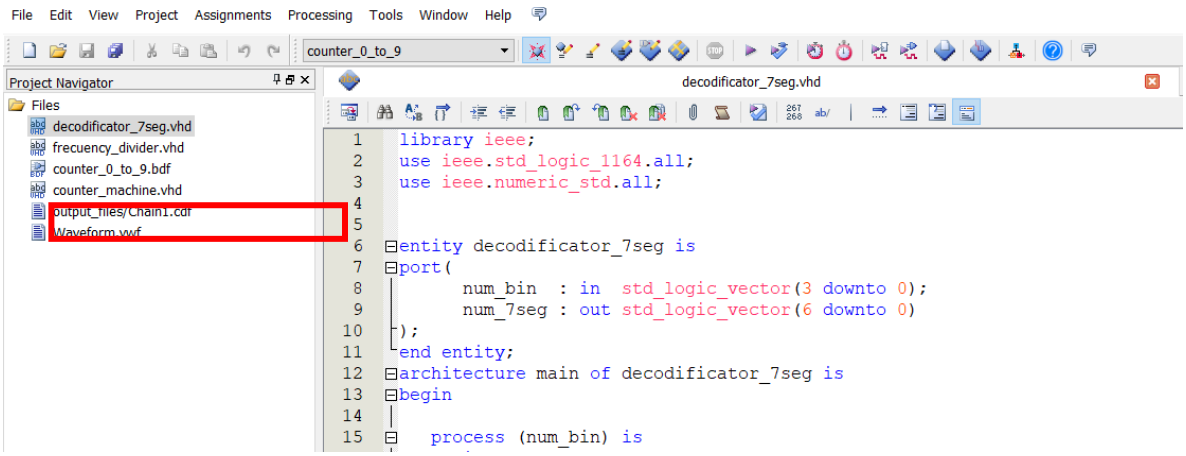


Figura 3.44 Ubicación del archivo de simulación en el navegador del proyecto.

3.7.2 ENTRADAS Y SALIDAS

Para que la simulación cumpla su propósito, es indispensable añadir todas las señales de entrada y salida que existen en el proyecto. Para ello, primero se hace clic derecho en la barra de lado derecho y luego se selecciona la opción *"Insert Node or Bus".* Se desplegará otra ventana y en ella se hace clic en el botón *"Node Finder"*, como lo muestra la Figura 3.45.

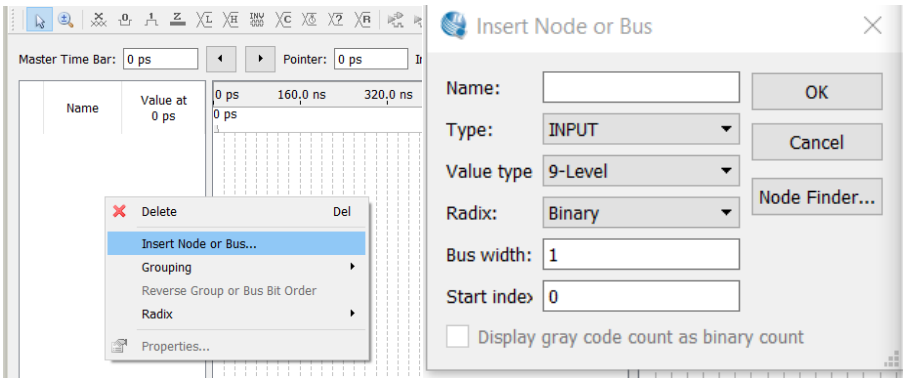


Figura 3.45 Insertar buses de entrada y salida en el simulador.

Después se abrirá otra ventana nombrada *"Node Finder"*, que se visualiza en la Figura 3.46, donde se desplegará una lista vacía del lado izquierdo y otra del lado derecho. En esas listas vacías, tal cual las presenta la Figura 3.46, se mostrarán todas las señales de entrada y salida del código VHDL. Para insertar estas señales se debe hacer clic en el botón *"List"* y se llenará la lista del lado izquierdo con todas las señales de entrada y salida del programa como lo muestra la Figura 3.37.

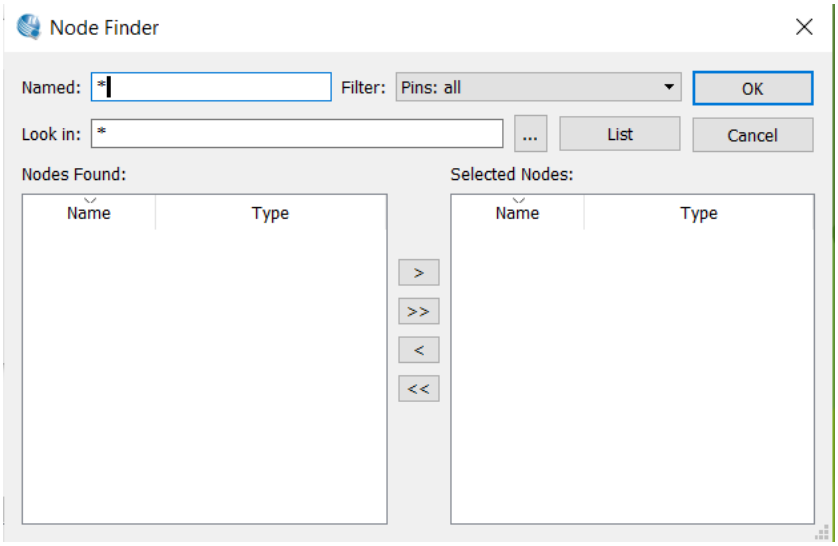


Figura 3.46 Ventana de navegación de las entradas y salidas del simulador.

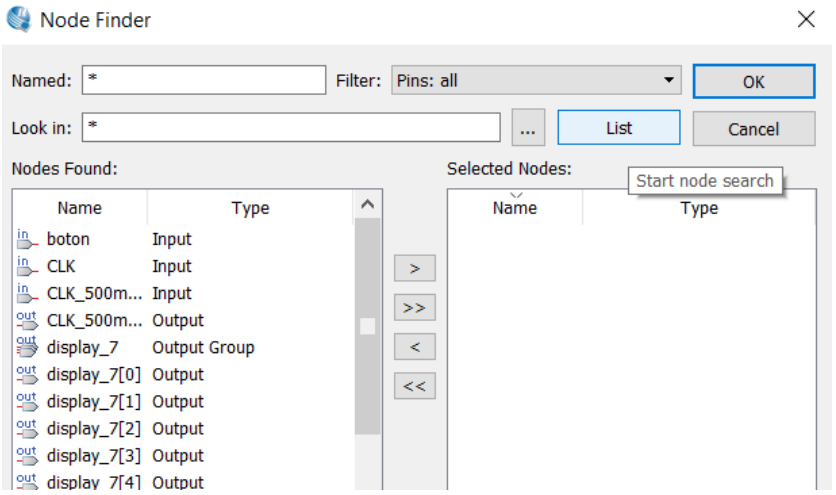


Figura 3.47 Lista de entradas y salidas del proyecto (lado izquierdo).

De acuerdo con la necesidad del diseñador, para analizar las señales se puede tomar un grupo seleccionado de señales que se consideren de mayor importancia. Para hacer uso de ellas en el simulador, se selecciona la señal o el conjunto de señales de la lista del lado izquierdo y con el botón *">"* se agregan a la lista del lado derecho que representa las entradas y salidas que se usarán el simulador.

Si se requirieren usar todas las señales del proyecto en el simulador, se hace clic en el botón *">>"* y automáticamente aparecerán en la lista del lado derecho como lo ejemplifica la Figura 3.48. Por último, se hace clic en el botón *"Ok"* y se cerrará la ventana.

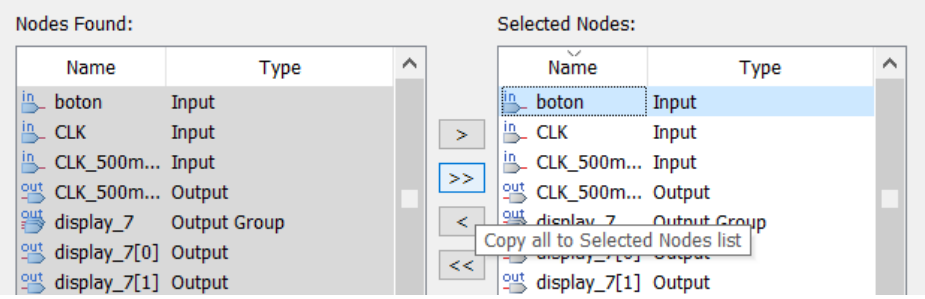


Figura 3.48 Selección de las señales que se utilizarán en el simulador.

Ahora en la ventana *"Node finder"* aparecerá *"**Multiple Items**"* en las distintas secciones de la ventana. Esto quiere decir que las entradas y las salidas están correctamente añadidas al simulador. Finalmente, en la ventana mostrada en la Figura 3.49, para finalizar este proceso se hace clic en el botón "Ok"

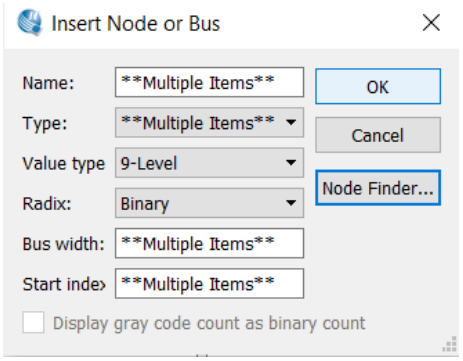


Figura 3.49 Ventana que indica la correcta asignación de las entradas y salidas al simulador.

3.7.3 CONDICIONES EN LAS ENTRADAS

Cuando las señales de entrada y salida se incluyan, como fue explicado en el apartado anterior, podrán ser visibles en el analizador lógico y estarán listas para asignar el estado lógico de las señales de entrada. En primera instancia, las entradas tendrán valores nulos y las salidas estarán en alta impedancia, como se aprecia en la Figura 3.50. Para asignar un estado lógico en las señales de entrada, se debe seleccionar la señal deseada con el cursor y después en la barra superior seleccionar la opción de estado lógico 1 (o 0 según sea el caso).

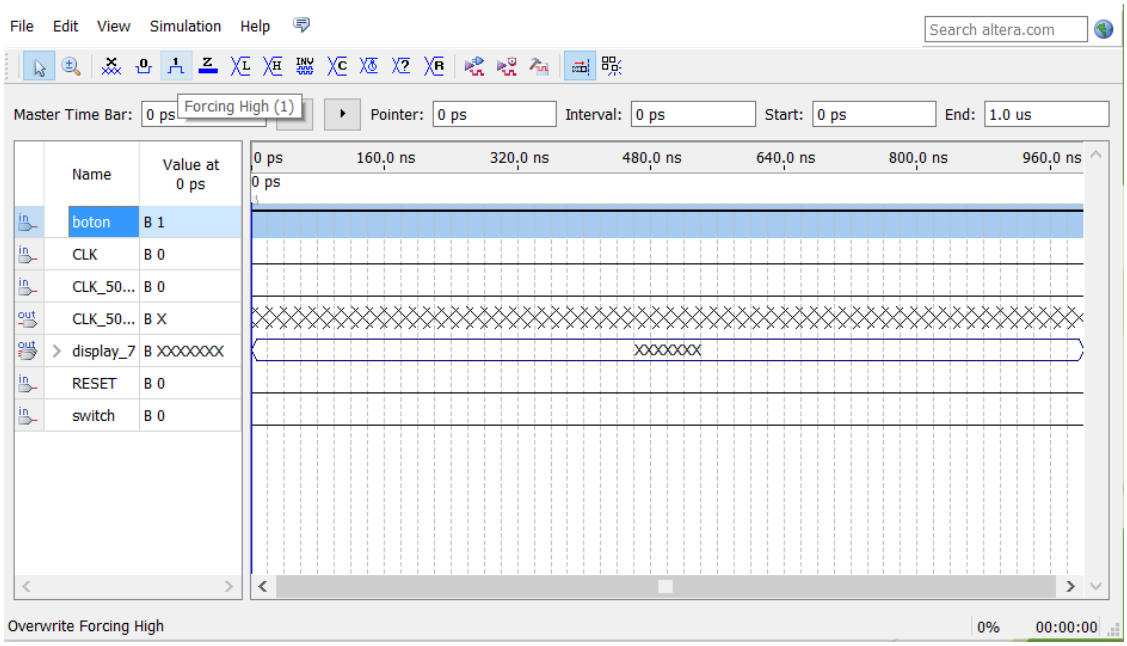


Figura 3.50 Asignación del estado lógico a una entrada.

Cuando las entradas involucran señales de reloj, se selecciona la señal y luego se hace clic en la opción *"Clock"* y se desplegará una ventana en donde se requiere introducir el periodo de la señal de reloj, también existe la opción de establecer ese periodo en términos de la frecuencia. Para realizar simulaciones de los FPGA presentados en este manual, de acuerdo con sus datos técnicos, la frecuencia de reloj de ambos dispositivos es de 50 MHz (20 ns). En la Figura 3.51 muestra la ventana donde se introducen el valor de la frecuencia para asociar la señal de onda cuadrada a una señal de entrada.

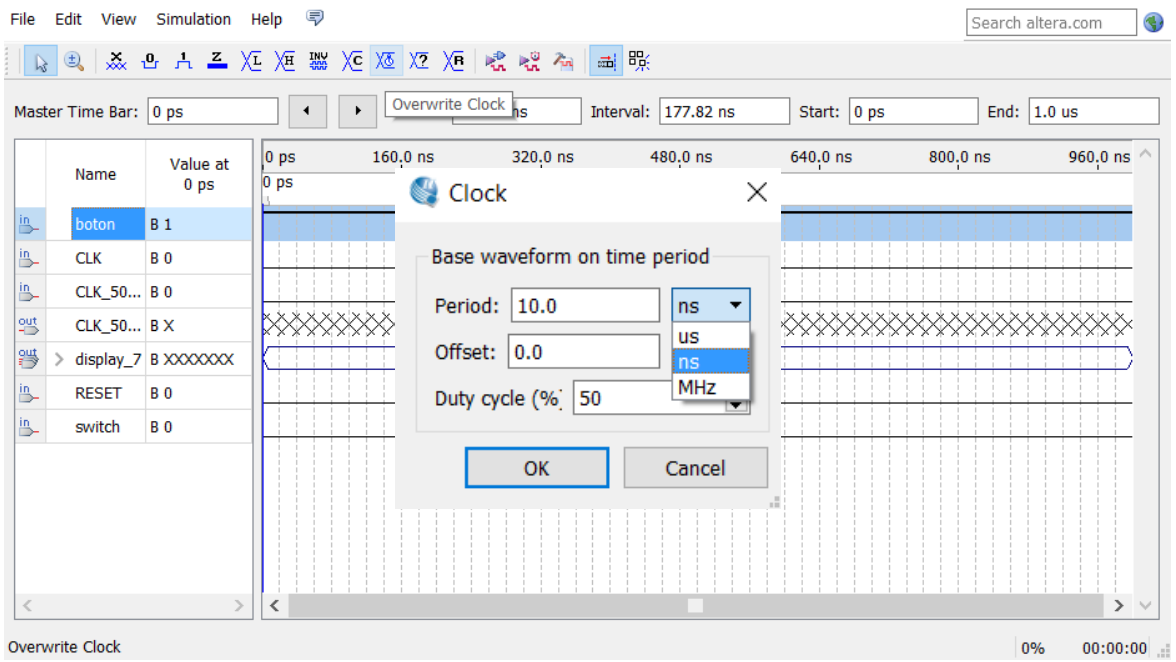


Figura 3.51 Asignación de una señal de reloj a una entrada.

3.7.4 SIMULACIÓN

Una vez establecidos los comportamientos de las entradas, se puede dar inicio con la simulación del proyecto. Para ejecutar la simulación, primero se debe configurar el simulador de Quartus II, para ello, se hace clic en la pestaña *"Simulation"* de la barra de herramientas, después se selecciona *"Options"* y cuando se despliegue la ventana de la Figura 3.52, se cambia a la opción "Quartus II Simulator".

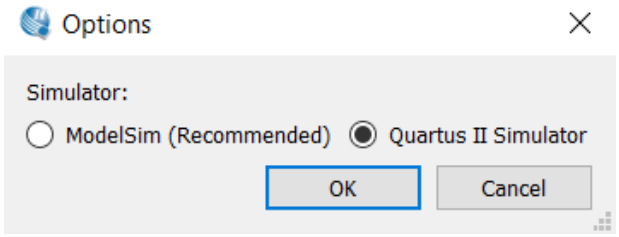


Figura 3.52 Configuración del imulador de Quartus II.

Una vez configurado el simulador, se puede ejecutar la simulación haciendo clic la opción *"Run simulation"* ubicado en la barra superior y se dará inicio al proceso de compilación que tardará unos segundos hasta que termina el proceso.

Finalmente se abrirá una nueva ventana del simulador, como lo muestra la Figura 3.53, donde se observa el trabajo del analizador lógico. Se puede observar el comportamiento de las señales de salida que dependen de los estados lógicos y comportamiento de las entradas. El rango de trabajo de este simulador es de 0 a 1 µs y presenta divisiones cada 20 ns.

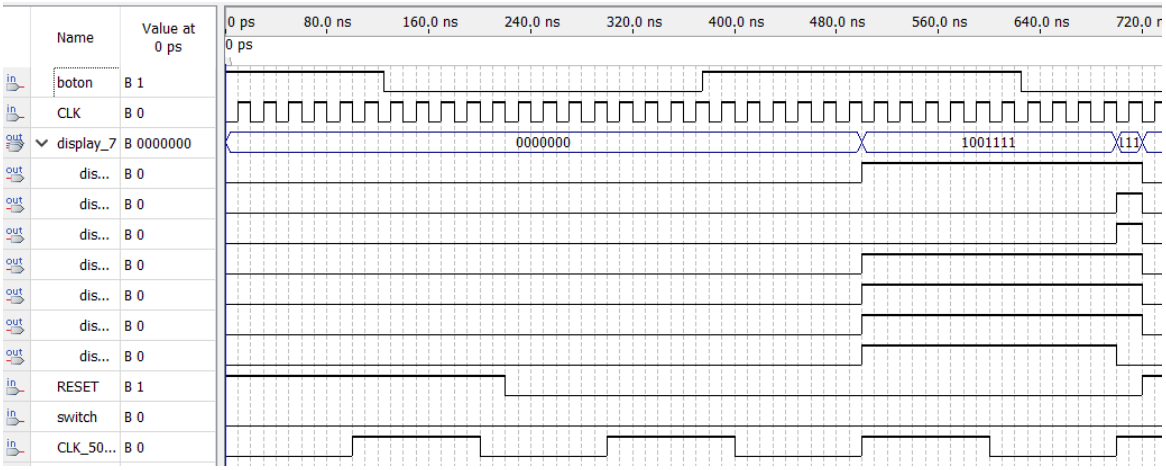


Figura 3.53 Simulador de Quartus II.

En la Figura 3.53 se aprecia que las salidas de bit simple se asemejan a una señal de reloj, porque puede verificar su comportamiento en un instante de tiempo. Las salidas que representen vectores de bits (como la señal display_7 de la Figura 3.56) el simulador es capaz de representarlos como números binarios que cambian a través del tiempo.

En la mayoría de las veces los simuladores son usados para corregir errores en el código, y al hacer estas modificaciones se vuelve a simular el código corregido. Este simulador es capaz de detectar los cambios en el código y eso sucede cuando se compila el proyecto en el software de Quartus II.

Una gran ventaja de este simulador es que al momento de compilar el proyecto no se pierden las condiciones establecidas en las señales de entrada por lo que no es necesario volverlas a asignar.

3.7.5 OBSERVACIONES DEL SIMULADOR

Este simulador solo muestra las señales de entrada y salida del circuito lógico escrito en el código VHDL, por lo tanto, no se puede analizar el comportamiento de las señales internas. Además, solo es capaz de mostrar las señales de origen lógico como los valores *STD_LOGIC* y los *STD_LOGIC_VECTOR*, por lo tanto, no es posible observar el comportamiento de otro tipo de datos en el analizador lógico

Una desventaja del simulador es que tiene un rango máximo de 0 a 100000 ns (100 µs), por lo que se recomienda usar el simulador para corregir errores de señales que tengan un periodo de tiempo menor al valor máximo de ese rango.

3.8 PROGRAMACIÓN DEL FPGA CON USB BLASTER

Los FPGA de Altera requieren de un dispositivo que establezca la comunicación entre la computadora y el FPGA. En la Figura 3.54 se tiene una fotografía del dispositivo *ALTERA USB Blaster* que establece la comunicación entre el FPGA y la computadora para transferir el código elaborado en Quartus II.

Figura 3.54 Programador de FPGA USB Blaster.

El programador se debe conectar a uno de los puertos USB de la computadora mediante un cable Mini USB Tipo B, y en el otro extremo se conecta un conjunto de cables que realizan la conexión del dispositivo al puerto JTGA del FPGA, adicionalmente también puede ser conectado al puerto AS pero requiere realizar un ajuste en el software.

3.8.1 INSTALACIÓN DEL CONTROLADOR

Al conectar el programador a la computadora no lo reconocerá al instante, por lo que se requiere instalar el controlador manualmente. El proceso es el siguiente:

1. Conectar el programador a cualquier puerto USB de la computadora.
2. Abrir el administrador de dispositivos de la computadora, mostrado en la Figura 3.55.

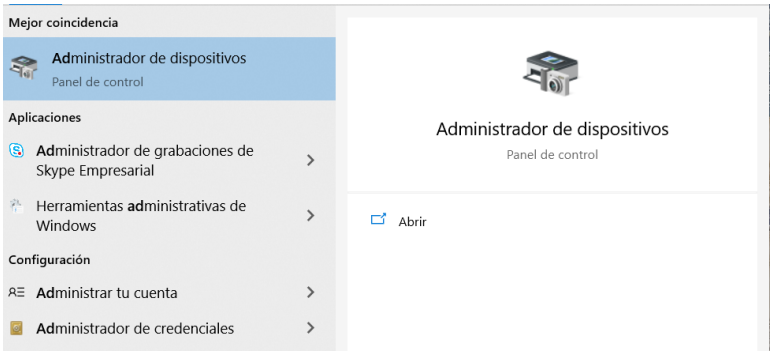


Figura 3.55 Administrador de dispositivos de Windows.

3. De la lista que se muestra en la Figura 3.56, se debe buscar en la sección “Otros dispositivos” la opción *“USB-Blaster”*, luego se hace clic derecho y se selecciona la opción “Actualizar controlador”.

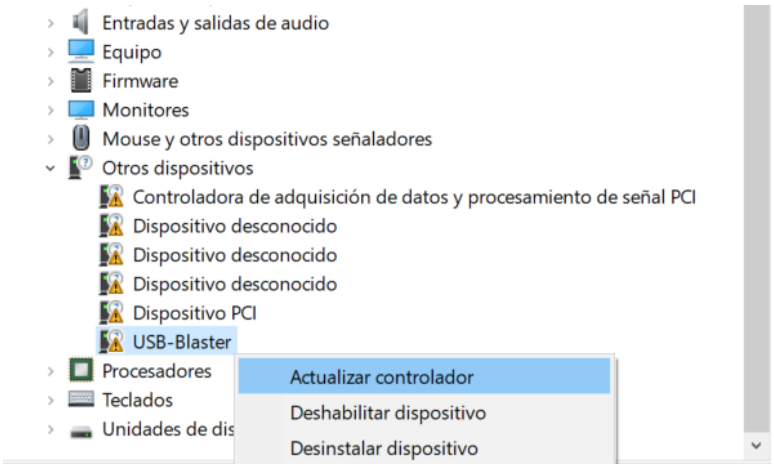


Figura 3.56 Lista de dispositivos periféricos conectados a la PC.

4. Se abrirá la siguiente ventana y luego se selecciona la segunda opción “Buscar controladores en mi equipo” mostrado en la Figura 3.57.

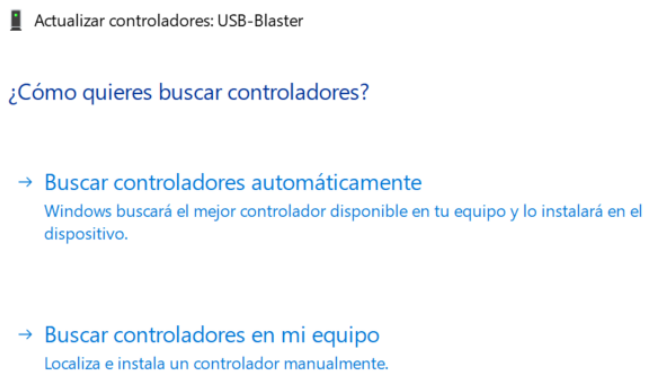


Figura 3.57 Instalador de controladores de Windows.

5. Después, de hacer clic, se abrirá el explorador de archivos y se tendrá que hacer clic en la opción “Examinar” y tendrá que buscar la carpeta donde se aloja el programa de Quartus II y *ModelSim*. Por defecto, el controlador se almacena en la dirección “C:\altera”, como lo muestra la Figura 3.58, pero si en la instalación existe otra ubicación, se verá ingresar la dirección donde se ubica la carpeta en la computadora.

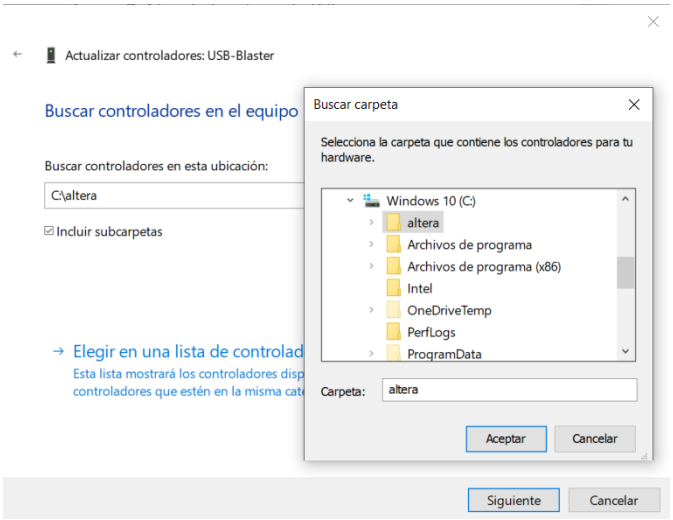


Figura 3.58 Carpeta donde se ubica el controlador para el USB Blaster.

6. Posteriormente, se abrirá la ventana de la Figura 3.59 la cual pedirá confirmar la instalación del programador. Finalmente se hace clic en “Instalar”.

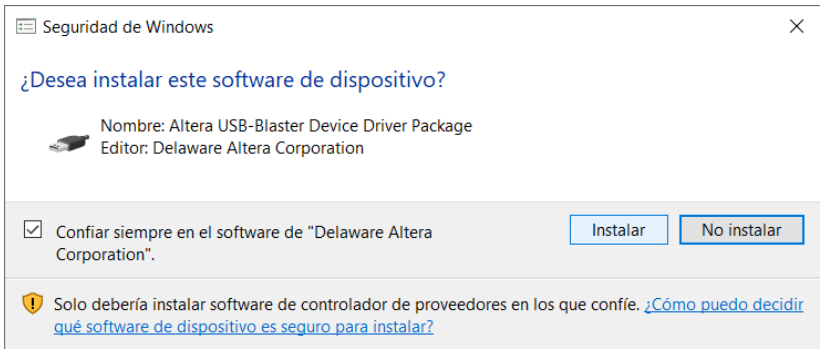


Figura 3.59 Instalación del controlador.

Después del proceso aparecerá el mensaje de la Figura 3.60 que indica que se instaló el controlador.

Figura 3.60 Proceso de instalación finalizado.

7. Para terminar el proceso de instalación, se debe abrir Quartus II y ubicar la opción “Programmer”, como lo muestra la Figura 3.61, y se hace clic sobre el icono.

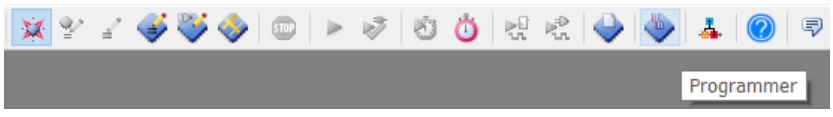


Figura 3.61 Herramienta "Programer" para usar el programador.

8. Al abrir la ventana, se deberá ubicar la opción *"Hardware Setup"* y se tendrá que hacer clic en dicho botón para abrir una segunda ventana como lo muestra la Figura 3.62. Luego, se selecciona el menú desplegable y se cambia a la opción a *"USB-Blaster [USB-0]"*. Por último, se hace clic en *"Close"* y se cierra la ventana.

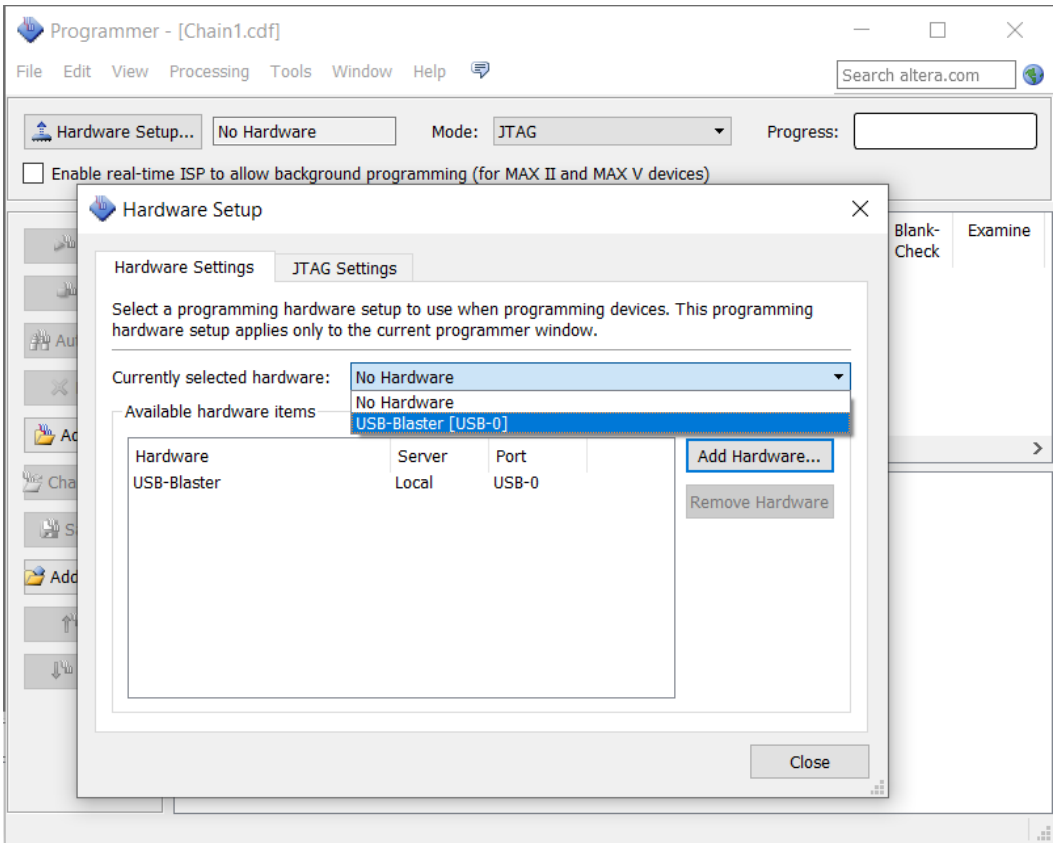


Figura 3.62 Selección del programador USB Blaster en Quartus II.

De esta manera el programador estará listo para ser usado y se podrá cargar los códigos que se usarán en los capítulos 4 y 5 de este manual. En la siguiente sección se explicará los dos modos de programación los cuales se pueden usar para programar el FPGA.

3.8.2 MODO JTGA

El puerto JTGA *(Joint Test Action Group)* utiliza la memoria volátil del FPGA para guardar el código, por lo tanto, se almacenará de manera temporal. Si el FPGA es desconectado y vuelto a conectar, el código se borra de la memoria.

Para programar el FPGA, se debe compilar el proyecto con anterioridad para que el software genere el archivo *".sof"* y después haga clic en la opción *"Programmer"* somo lo muestra la Figura 3.63.

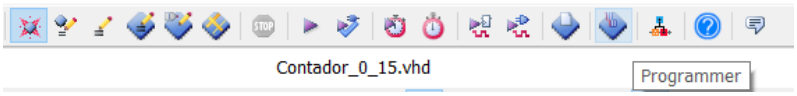


Figura 3.63 Herramienta del programador.

Se abrirá la ventana de la Figura 3.64, que corresponde a la configuración del programador, y por lo general se seleccionará el archivo *".sof"* de manera automática:

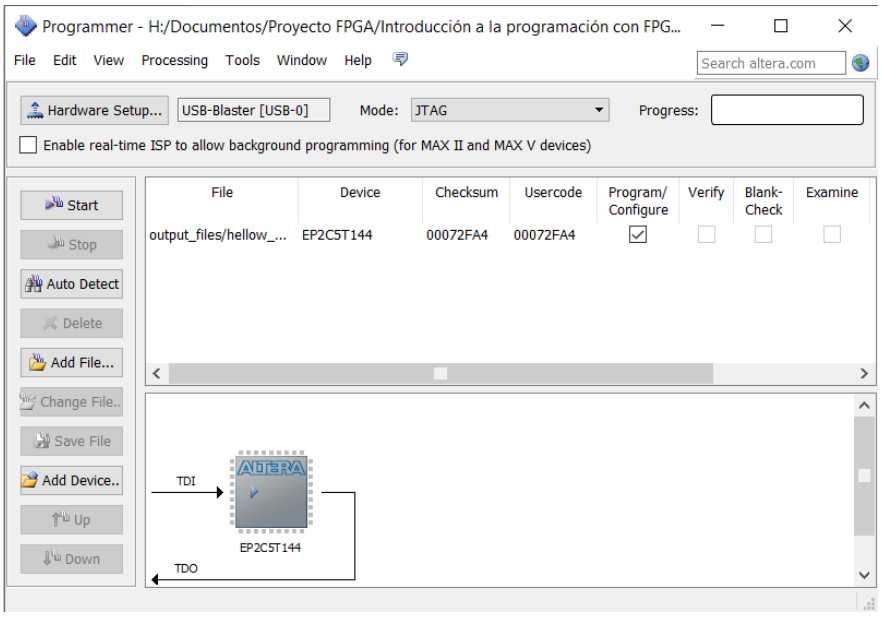


Figura 3.64 Ventana del programador con archivo cargado.

Si la ventana no aparece como en la Figura 3.64, se tendrá que buscar manualmente el archivo, para ello se hace clic en la opción *"Add File…"*, luego se abrirá el navegador de archivos y se debe buscar la carpeta *"output_files"*. Cuando abra la carpeta, como lo muestra la Figura 3.65, se selecciona el archivo *".sof"* y luego se hace clic en "Open".

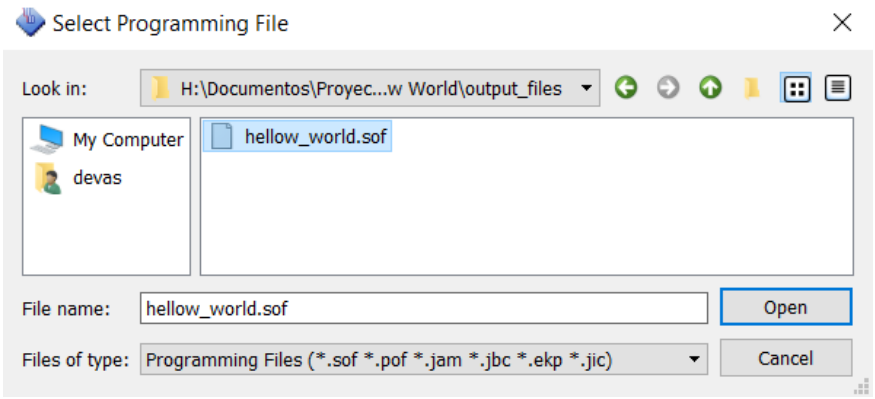


Figura 3.65 Carpeta "output_files" que contiene el archivo .sof para programar por puerto JTGA.

De esa manera el programador estará configurado y cuando se haga clic en el botón *"Start"* se iniciará el proceso de escritura en el FPGA donde la barra superior derecha mostrará el avance del proceso, como se observa en la Figura 3.67. Cuando aparezca el mensaje *"100% (successfull)"* el código estará dentro del FPGA.

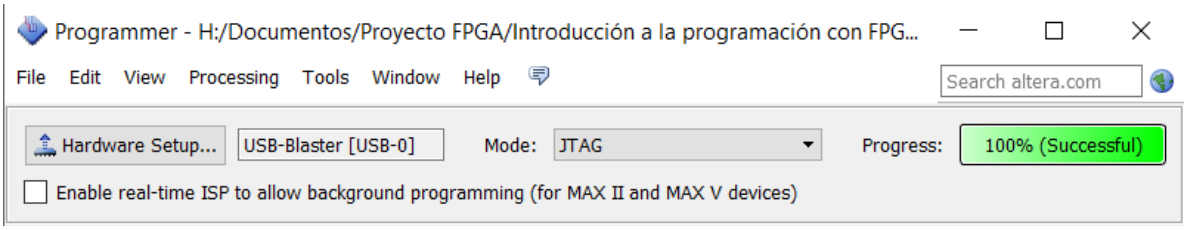


Figura 3.66 Proceso de programación por puerto JTGA finalizado.

3.8.3 MODO AS

El puerto *Active Serial* (AS) realiza la conexión a una memoria EEPROM ubicada dentro de la placa de desarrollo la cual guardará los datos del programa y permanecerán dentro del FPGA sin importar que la energía sea interrumpida.

En la sección anterior se utilizó un archivo *".sof"* para realizar la programación por el puerto JTGA. Para el puerto AS se requiere de un archivo *".pof"*. En algunas ocasiones Quartus II no genera dicho archivo automáticamente por lo que se debe configurar manualmente, para ello se deben realizar los siguientes pasos:

1. En el menú, se debe localizar la pestaña *"Assigments"* y después la opción *"Device"* como lo muestra la Figura 3.67.

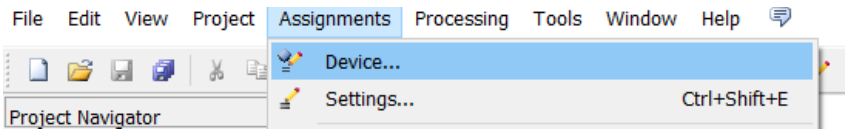


Figura 3.67 Pestaña "Assigments".

2. Se abrirá la ventana de la Figura 3.68 y se seleccionará la opción *"Device and Pin Options…"*.

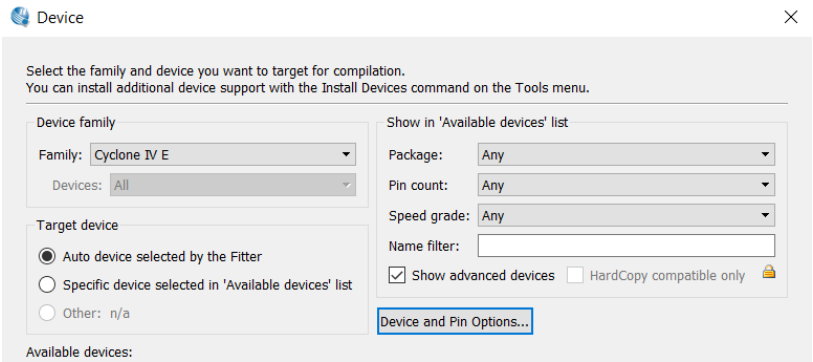


Figura 3.68 Ventana "Device".

3. Después, se desplegará la ventana de la Figura 3.69 donde en el menú izquierdo se tendrá que hacer clic en la opción *"Configuration"* y en la sección *"Configuration device"* el software tiene por defecto la opción configurada en *"Auto"*.

   La opción *"Use configuration device"* tendrá que ser habilitada y posteriormente se debe hacer clic en el menú para desplegar una lista valores que corresponden a los modelos de EEPROM que pueden incluir los FPGA en su tarjeta de desarrollo. La tarjeta Cyclone II tiene una memoria tipo EPCS4 por lo que se selecciona la opción correspondiente. Por otro lado, si se está usando la tarjeta Cyclone IV, se selecciona el modelo de EEPROM EPCS16.

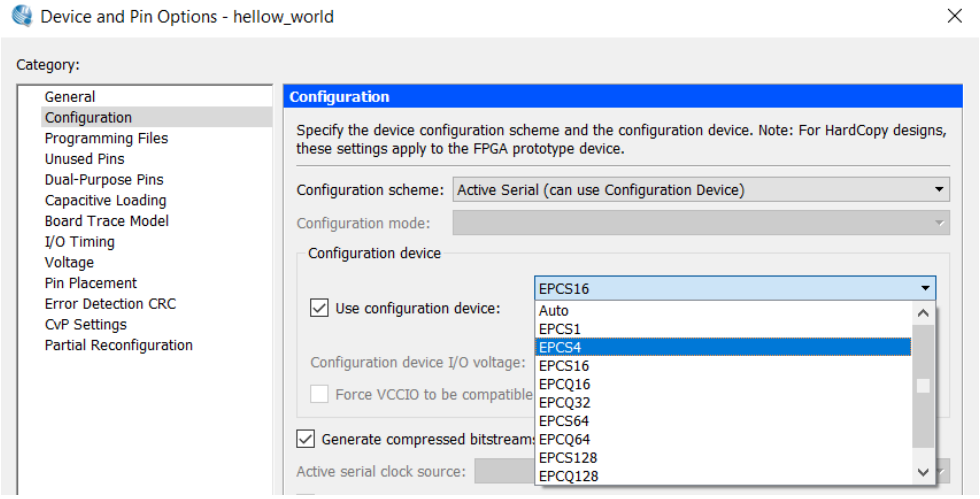


Figura 3.69 Selección del modelo de EEPROM soldado en la tarjeta del FPGA.

4. Finalmente, se hace clic en el botón aceptar y de esta manera el software de Quartus II tendrá la configuración para crear el archivo *".pof"* el cual tendrá que ser cargado en la ventana de configuraciones del programador como se mostró en la sección anterior.

Para realizar la programación por el puerto AS, se tiene que abrir la herramienta *"Programer"* y cuando hay abierto la ventana, se cambia la opción *"JTGA"* por *"Active Serial Programing"* en el menú desplegable de *"Mode"* como lo muestra la Figura 3.70.

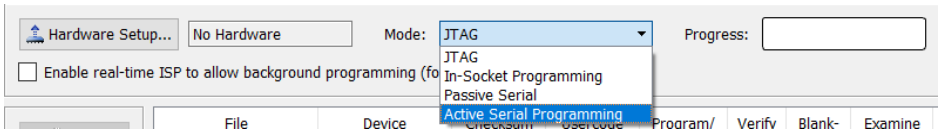


Figura 3.70 Configuración de la programación del puerto Active Serial en la ventana del programador.

Al hacer el cambio en "Mode" se desplegará una ventana emergente con un aviso y solo se hace clic en aceptar para cerrarla. La ventana de la configuración del programador se mostrará vacía, para cargar el archivo *".pof"* por lo que se tendrá que hacer clic en la opción *"Add files"* y luego buscar en la carpeta *"output_files"* en el explorador de archivos, como lo muestra la Figura 3.71, para finalmente seleccionar dicho archivo. Se hace clic en *"Open"* para guardar los cambios y después regresar a la ventana del programador.

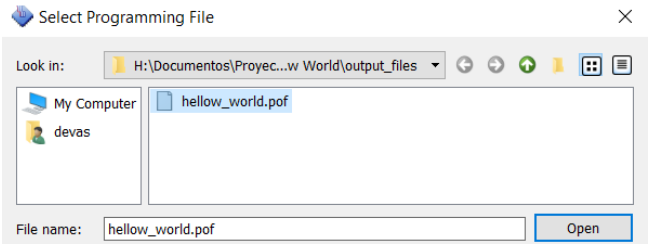


Figura 3.71 Selección del archivo .pof en la carpeta "output_files".

Después, se hace clic en cuadro *"Program/Configure"* para seleccionar el archivo que se cargará en el FPGA. Antes de cargar en archivo al FPGA se debe revisar que el programador *USB-Blaster* esté conectado en el puerto AS de la placa de desarrollo. Realizada la conexión, se hace clic en *"Start"* y la barra de carga indicará el proceso de carga y si no existe ningún error indicará que la programación del FPGA ha sido exitosa como lo muestra la Figura 3.72.

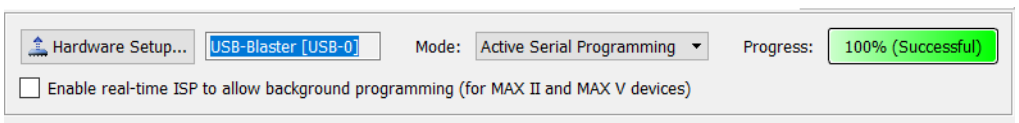


Figura 3.72. Proceso de programación por puerto AS completada.

CAPÍTULO 4
CIRCUITOS BÁSICOS CON FPGA

4.1 ENCENDIDO DE UN LED MEDIANTE UN BOTÓN

Este primer circuito tiene el propósito hacer un primer acercamiento a la programación con lenguaje VHDL y al mismo tiempo se creará un proyecto funcional con el programa Quartus II para llevar el código al FPGA. Se utilizará la teoría de las secciones 2.3, 2.4 y 2.5 donde usará una señal de entrada tipo *STD_LOGIC* la cual se asigna su valor a una señal de salida *STD_LOGIC*.

Implementando este comportamiento a un circuito físico, la señal de entrada será un botón pulsador, cuando esté sin presionar el estado lógico será 0 y cuando sea presionado será el estado lógico 1. El comportamiento esperado en la señal de salida será exactamente igual, cuando el botón esté sin presionar, el led estará apagado (estado lógico 0), y si el botón esta presionado, el led estará encendido (estado lógico 1).

Para una mejor explicación del párrafo anterior, la Figura 4.1 muestra un diagrama de flujo que representa el comportamiento del circuito lógico donde las variables serán usadas en el código son *LED* y *BUTTON*.

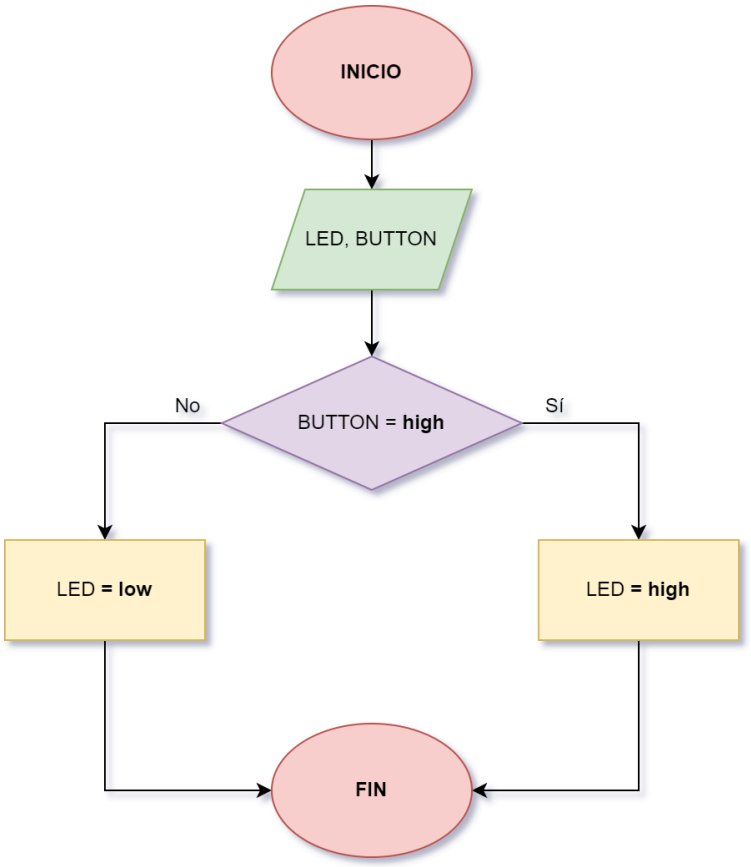


Figura 4.1 Diagrama de flujo para encender un led usando un botón pulsador.

Por otro lado, el diagrama de la Figura 4.2 representa la estructura del circuito digital de esta práctica, donde se tiene una entrada y una salida.

Figura 4.2 Diagrama de bloques que representa la estructura del circuito de la práctica 1.

Para este circuito digital, se crea un nuevo proyecto y se asigna el nombre *LED_ON_FPGA* siguiendo la metodología de la sección 3.3. Después se crea un archivo de VHDL con nombre *LED_ON_FPGA.vhd* el cual será el archivo principal y debido a que llevará el mismo nombre del proyecto.

En el archivo VHDL se empleará el código B.1.1 del apéndice, el cual tiene como entrada la señal *BUTTON* que se asocia al botón pulsador con resistencia pull-down. Por otro lado, la salida es representada por la variable *LED* y, como su nombre lo indica, se asocia al led en el circuito físico. En la Tabla 4.1 se detalla el número de pin físico asociado a ambos componentes y en la Tabla 4.2 se lista los dispositivos necesarios para esta práctica.

Se observa en la línea 17 del código empleado en esta práctica que se asigna directamente el valor de *BUTTON* a la señal *LED* y, por lo tanto, si la entrada es 1, la salida también es 1.

Para esta práctica, también se debe emplear el código B.1.2 que es muy similar al anterior, pero con la diferencia que en esa línea de código se usa una compuerta NOT la cual invertirá el comportamiento del circuito, es decir, si el botón está en 0, el led estará en 1 y viceversa.

Finalmente, para comprobar el funcionamiento del código, se debe construir el circuito del diagrama esquemático de la Figura 4.3 y debe ser parecido físicamente al diagrama de conexiones mostrado en la Figura 4.4.

Tabla 4.1 Lista de dispositivos para la práctica 1.

Dispositivo	**Valor**	**Cantidad**	**Símbolo en el diagrama**
Resistencia	10 kΩ	1	R1
Resistencia	220 Ω	1	R2
Condensador cerámico	220 nF	1	C1
Led	--	1	LED1
Botón pulsador	--	1	KEY1

Tabla 4.2 Pines del FPGA asociados a los componentes del circuito de led accionado por botón pulsador.

Entradas			**Salidas**		
Variable	VHDL	PIN FPGA	VARIABLE	En VHDL	PIN FPGA
BUTTON	BUTTON	120	LED	LED	112

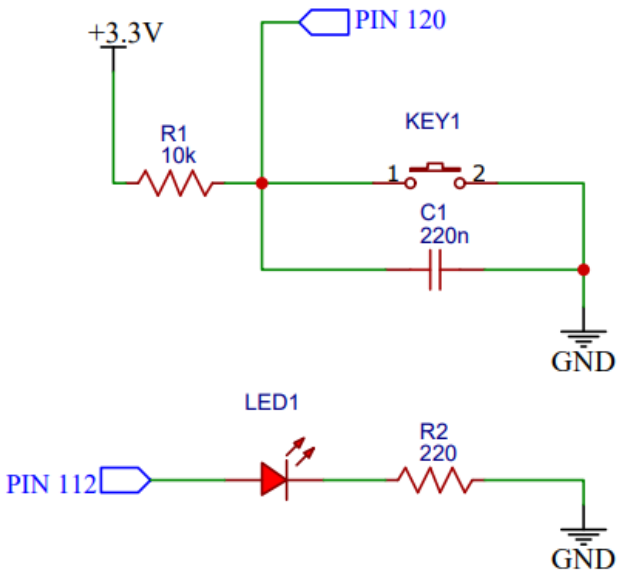


Figura 4.3 Diagrama esquemático del circuito de la práctica 1.

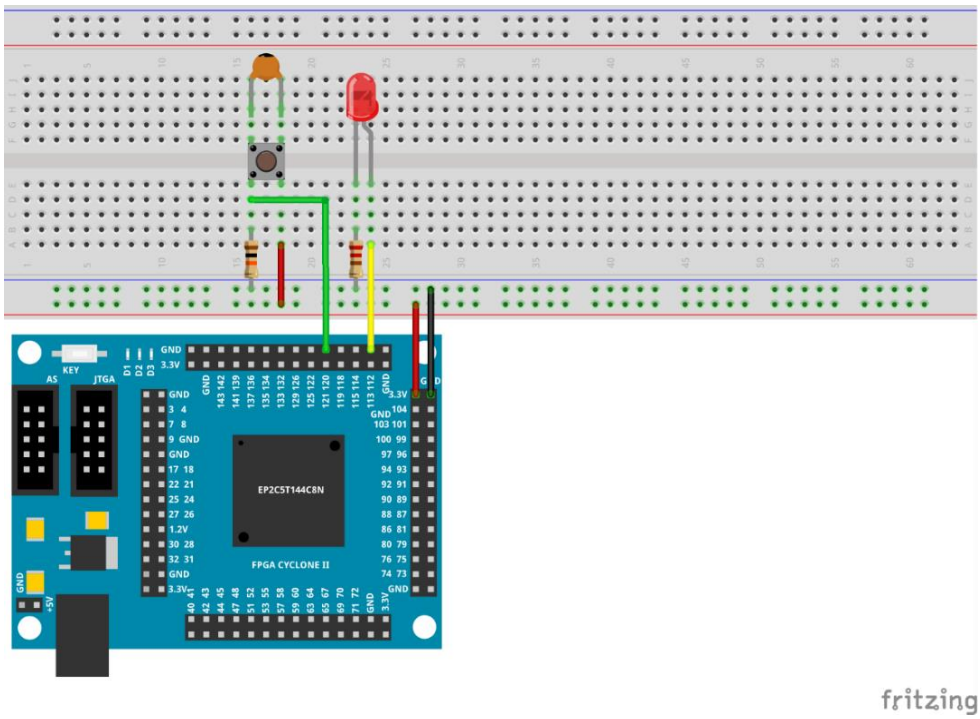


Figura 4.4 Ilustración del circuito físico de la práctica 1 implementado en una placa de pruebas.

4.2 PARPADEO DE UN LED

El siguiente circuito tendrá el propósito de hacer parpadear un led con un periodo deseado haciendo uso de una unidad secuencia *process* explicada en la sección 2.3.3 y las estructuras lógicas *if/else* de la sección 2.7. Además, se emplearán tipos de variable *signal* y *constant* explicados en las secciones 2.5.1 y 2.5.2 respectivamente.

Para llevar esto a acabo, se empleará una técnica propuesta que consiste en realizar un gasto de instrucciones del código en VHDL. Los FPGA presentados en este manual, cada uno tiene una frecuencia de trabajo de 50 MHz, por lo tanto, el periodo de tiempo que tarda cada instrucción en ejecutarse dentro de una unidad secuencial es de 20 ns. Si un led parpadea con este valor de frecuencia de reloj, el ojo humano no sería capaz de observar el comportamiento.

El gasto de instrucciones del código permitirá reducir el periodo de tiempo de la onda cuadrada del FPGA de 20 ns a 500 ms, que es el valor deseado. En otras palabras, la onda cuadrada resultante tendrá una duración de 250 ms entre el cambio de 0 a 1 y otros 250 ms entre el cambio de 1 a 0 y este ciclo será observable a través de un led que se encuentre conectado a la señal resultante en el FPGA.

El gasto de instrucciones estará regido por un contador, y a su vez se establece un valor límite de gasto de instrucciones para generar el cambio de 0 a 1 y viceversa. Para calcular el límite del contador, primero se debe establecer un valor deseado para la frecuencia, o el periodo, y después usar cualquiera de las siguientes expresiones matemáticas:

$$i_{contador} = 25{,}000\, T_{ms} - 1$$

$$i_{contador} = 25\, T_{\mu s} - 1$$

$$i_{contador} = \frac{50}{2 f_{MHz}} - 1$$

Donde $i_{contador}$ es el valor límite del contador dentro de la unidad secuencial, $T_{\mu s}$ es el periodo de tiempo deseado de la señal de reloj en µs, T_{ms} es el periodo de tiempo deseado de la señal de reloj en ms y f_{MHz} es la frecuencia deseada de la señal de reloj en MHz.

Las tres expresiones solo son válidas para los FPGA que tienen una frecuencia de reloj de 50 MHz, que son los usados en este manual (Cyclone II y Cyclone IV). La primera expresión está en términos del periodo de la señal deseada en ms.

Por ejemplo, si se requiere de un periodo de 250 ms, se sustituye 250 en T_{ms} y el resultado es un valor límite del contador de 6,249,999**.** La segunda expresión funciona de manera similar con la diferencia que el periodo deseado es en µs.

Por otro lado, la tercera expresión calcula el mismo valor límite usando como argumento la frecuencia en MHz. Por ejemplo, si se busca obtener una frecuencia de 3 MHz, el valor límite del contador es de 7.333 pero habrá que redondear este valor al valor entero más cercano, por lo tanto, se tiene un valor límite de 7 y la frecuencia aproximada será de 3.33 MHz. El diagrama de bloques de la Figura 4.5 representa las entradas y las salidas del circuito lógico que se programará en esta práctica.

Figura 4.5 Diagrama de bloques para circuito de parpadeo de LED.

La Figura 4.6 presenta un diagrama de flujo que refleja el comportamiento deseado del código que se empleará en VHDL, el cual tendrá dos variables importantes. La primera es la variable *led* que representará el estado lógico del led físico y que se tendrá que inicializar en el estado lógico 0. La segunda variable es el contador, que se tendrá que inicializar en el valor numérico 0 porque *count* realizará un incremento cada vez que se ejecute la unidad secuencial en el FPGA. Una estructura *if* evaluará cuándo el contador llega al valor límite de 12,499,999 (para obtener una señal de 500 ms). Si el contador llega a ese valor, la variable *led* invertirá su estado lógico (pasará desde 0 a 1) y el contador se reiniciará, de lo contrario *led* continuará en su estado lógico inicial y el contador continuará aumentando su valor.

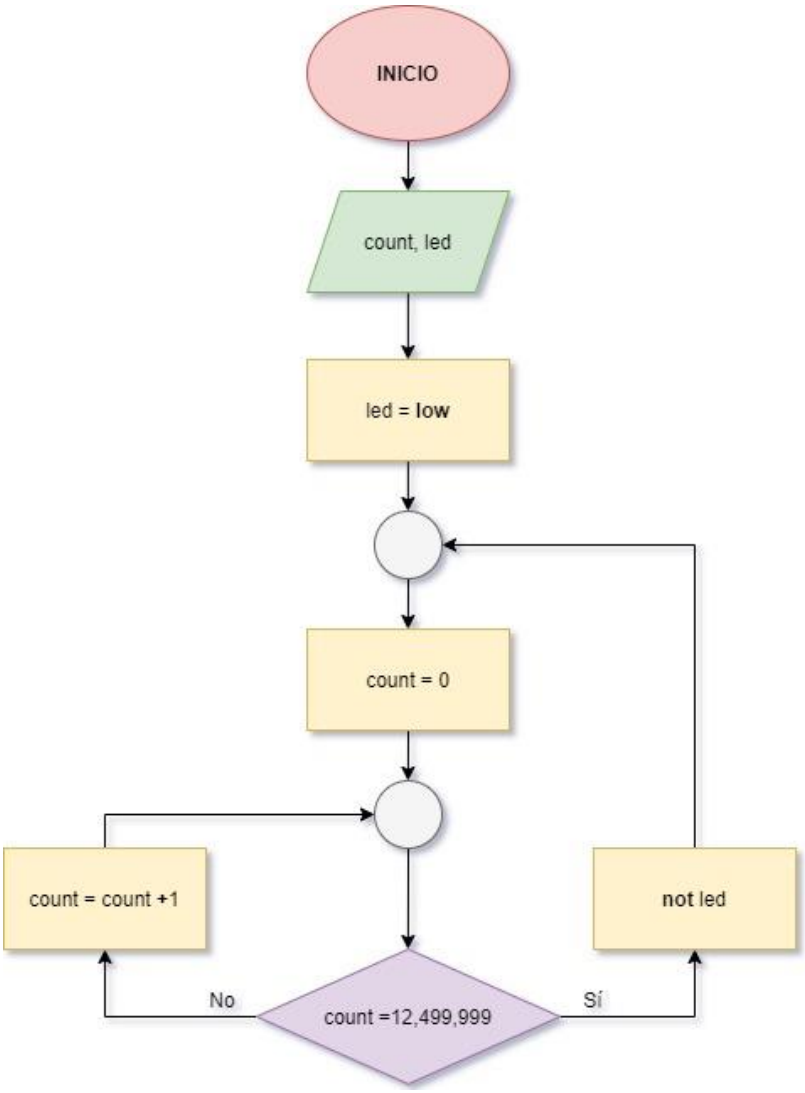


Figura 4.6 Diagrama de flujo de parpadeo de led con periodo de 500 ms.

Para empezar a programar el circuito, primero se crea un nuevo proyecto con nombre de *LED_BLINKING* y después el archivo VHDL con el mismo nombre.

Después, se deberá configurar el archivo para tener una entrada, asociada a la frecuencia de reloj del FPGA, y una salida, asociada al led que se encenderá y apagará en un periodo de tiempo de 500 ms, ambas variables se asocian a los componentes del circuito de acuerdo con la información de la Tabla 4.4.

El código que se empleará para esta práctica será el del apartado B.2 del apéndice, el valor del contador es igual a 12,499,999. El contador está empleado en una unidad secuencial, ubicada desde la línea 21 hasta la 32, nombrada *BLINK*, la cual ejecuta su contenido cuando

existe un cambio en la señal de reloj, representada por la entrada *CLK*. El contenido de esta unidad secuencial son dos estructuras lógicas *if/else* anidadas, donde el primer *if*, la de mayor jerarquía, evalúa la existencia de un flanco ascendente (un cambio de 0 a 1) con ayuda de la función *rising_edge()*, y si la condición de este primer if es verdadera, se ejecutará el segundo *if/else* contenido, el cual evalúa si el incremento de la señal *count* ha llegado a 12,499,999.

La variable *count* está restringida a tener valores solamente entre 0 y 50,000,000 para evitar que se desborde y la constante *count_limit* guarda el valor de 12,499,999. Por otra parte, la señal *led_state* es una señal interna que guarda el valor del estado lógico del led, el cual refleja su valor en la salida *LED.*

Finalmente, para la construcción del circuito se emplearán los dispositivos de la Tabla 4.3, mientras que las conexiones consisten en un led conectado al pin 112, de acuerdo con la Tabla 4.4. Para comprobar el funcionamiento del código, se debe construir el circuito del diagrama esquemático de la Figura 4.7 y físicamente debe ser similar al diagrama de conexiones mostrado en la Figura 4.8.

Tabla 4.3 Lista de dispositivos para la práctica 2.

Dispositivo	**Valor**	**Cantidad**	**Símbolo en el diagrama**
Resistencia	220 Ω	1	R1
Led	--	1	LED1

Tabla 4.4 Pines del FPGA asociados a los componentes del circuito de led parpadeando.

Entradas			**Salidas**		
Variable	En VHDL	Pin FPGA	Variable	En VHDL	Pin FPGA
CLK	CLK	17	LED	LED	112

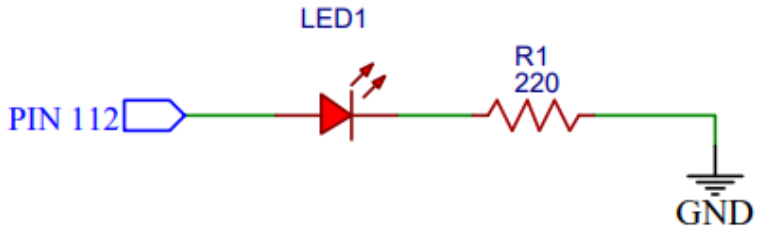


Figura 4.7 Diagrama esquemático del circuito de la práctica 1.

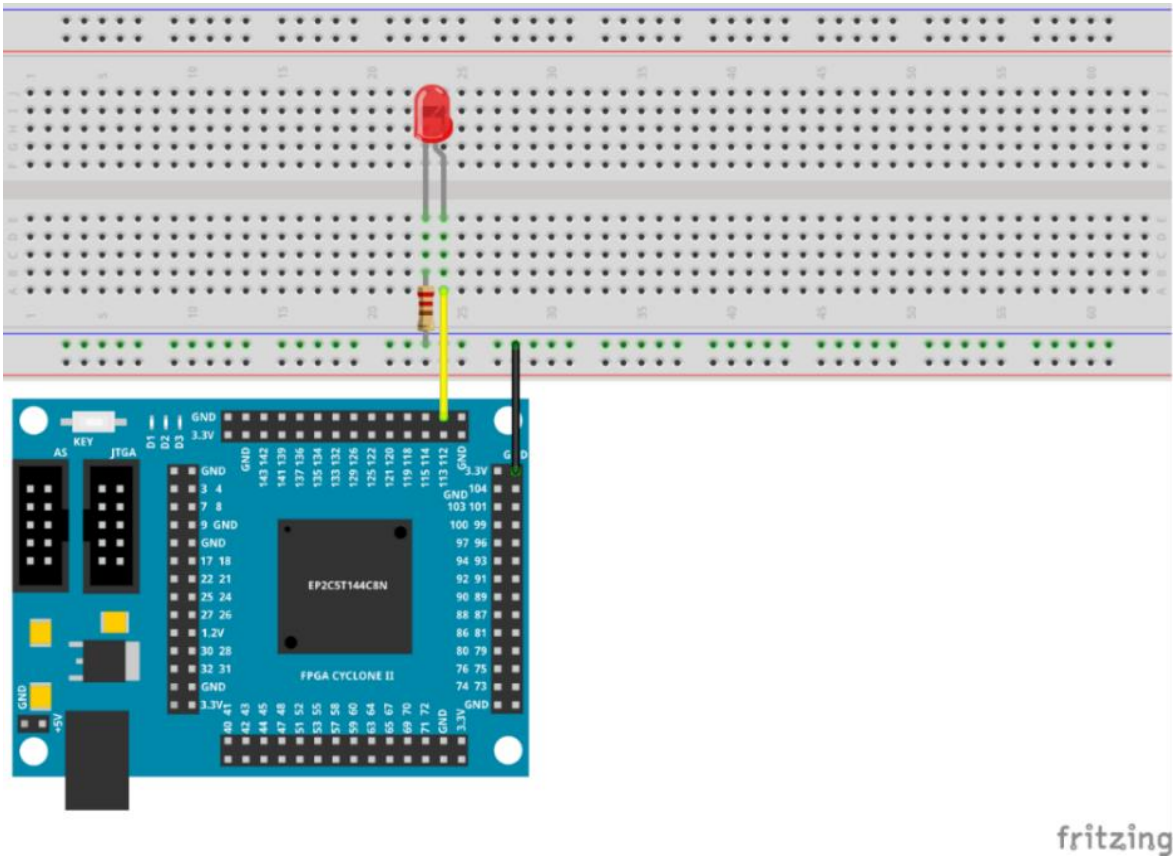


Figura 4.8 Diagrama de conexones del circuito físico de la práctica 2 implementado en una placa de pruebas.

4.3 PROGRAMACIÓN DE UNA FUNCIÓN BOOLEANA

Debido a que el FPGA es un arreglo de miles de compuertas lógicas, se pueden programar circuitos lógicos que representen funciones booleanas. Esta práctica tiene el propósito de implementar un circuito lógico usando las compuertas lógicas mostradas en la sección 2.6.2 y por primera vez se emplearán los tipos de variable *STD_LOGIC_VECTOR* para asociar entradas y salidas del código al circuito físico. Se implementará una función booleana que haga la multiplicación de dos números de dos bits cada uno, y el resultado será un número de cuatro bits. La Figura 4.9 muestra el diagrama de bloques que representa al circuito de esta práctica.

Figura 4.9 Diagrama de bloques del circuito multiplicador de 2 a 4 bits.

La Tabla 4.5 representa la tabla de verdad del circuito multiplicador de 2 a 4 bits mencionado anteriormente y a partir de ella se obtendrá la función booleana equivalente.

Los argumentos de la operación son dos números binarios con una extensión de 2 bits cada uno y que son representados por las variables A y B, por lo tanto, cada número representa en valores decimales un rango desde 0 hasta 3. El resultado de esta operación se representa con la variable C que será representado por un número binario de 4 dígitos y, al mismo tiempo, representa cada uno de los valores posibles resultados de la multiplicación de A por B, por lo tanto, si el número máximo de ambas variables es 3, el valor máximo de C debería ser 9. En la Tabla 4.5 se representa a detalle las combinaciones de A y B, con su respectivo resultado en la columna de la variable C.

Tabla 4.5 Tabla de verdad de un circuito multiplicador de 2 a 4 bits.

A		**B**		**C**			
A_1	A_0	B_1	B_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0
0	0	0	**1**	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	1	0	0	0	0
0	**1**	0	0	0	0	0	0
0	**1**	0	**1**	0	0	0	**1**
0	**1**	**1**	0	0	0	**1**	0
0	**1**	**1**	**1**	0	0	**1**	**1**
1	0	0	0	0	0	0	0
1	0	0	**1**	0	0	**1**	0
1	0	**1**	0	0	**1**	0	0
1	0	**1**	**1**	0	**1**	**1**	0
1	**1**	0	0	0	0	0	0
1	**1**	0	1	0	0	1	1
1	**1**	**1**	0	0	**1**	**1**	0
1	**1**	**1**	**1**	**1**	0	0	**1**

Los subíndices de las variables representan cada uno de los bits de A, B y C su representación como número binario. El subíndice 0 es el bit menos significado mientras que el subíndice 1 es el bit más significativo en ambos números. Mientras que C es el número de 4 bits donde C_3 es el bit más significativo. Resolviendo la tabla de verdad se obtienen las siguientes expresiones booleanas:

$$C_3 = A_1 \cdot B_1 \cdot A_0 \cdot B_0$$

$$C_2 = A_1 \cdot B_1 \cdot (\overline{A_0} + \overline{B_0})$$

$$C_1 = (A_1 \cdot B_0) \otimes (A_0 \cdot B_1)$$

$$C_0 = A_0 \cdot B_0$$

Donde el símbolo $+$ representa la compuerta OR, $\cdot$ es la compuerta AND y $\otimes$ es la compuerta XOR. Los paréntesis indican la jerarquía de cómo se deben ejecutar las operaciones.

Para empezar a programar las funciones booleanas, primero se crea un nuevo proyecto bajo el nombre *BOOLEAN_FUNCTION* y el archivo VHDL debe llevar el mismo nombre.

De acuerdo con la Figura 4.9, físicamente el FPGA estará conectado a un dip-switch de cuatro interruptores donde dos interruptores estarán asociados a los bits de A y los otros dos a los bits de B. De esta manera, cuando un interruptor este cerrando el circuito representará el número 1 binario en los bits de las variables A y B. Por otro lado, los bits de C estarán asociados a cuatro leds para representar el número binario resultante de la multiplicación representada por las funciones booleanas obtenidas de la Tabla 4.5.

En el archivo VHDL se escribirá el código A.3 del apéndice, donde A y B se declaran como *STD_LOGIC_VECTOR* de dos elementos, mientras que C será del mismo tipo, pero con cuatro elementos. Desde la línea 17 hasta la 20 se escriben cada una de las funciones booleanas mostradas en la página anterior donde se hace uso de los operadores lógicos, vistos en la sección 2.6.3, representando a cada una de las compuertas lógicas que conforma la función booleana. Los bits de estas señales estarán asociados a un pin del FPGA como lo indica la Tabla 4.7.

De acuerdo con la Figura 4.10, cada uno de los interruptores del dip-switch requieren tener conectado una resistencia pull-down para obtener voltaje y tierra que representen 1 y 0 respectivamente. El circuito construido debe ser igual o similar al presentado en la Figura 4.11. La lista de dispositivos para esta práctica se puede consultar en la Tabla 4.6.

Tabla 4.6 Lista de dispositivos para la práctica 3.

Dispositivo	Valor	Cantidad	Símbolo en el diagrama
Resistencia	220 Ω	4	R5, R6, R7, R8
Resistencia	1 kΩ	4	R1, R2, R3, R4
Dip-Switch	4 interruptores	1	SW1
LED	--	4	LED1, LED2, LED3, LED4

Tabla 4.7 Pines del FPGA asociados a los componentes del circuito del multiplicador de 2 a 4 bits.

Entradas			**Salidas**		
Variable	En VHDL	Pin FPGA	Variable	En VHDL	Pin FPGA
A_0	A[0]	96	C_0	C[6]	81
A_1	A[1]	99	C_1	C[5]	87
B_0	B[0]	101	C_2	C[4]	92
B_1	B[1]	104	C_3	C[3]	93

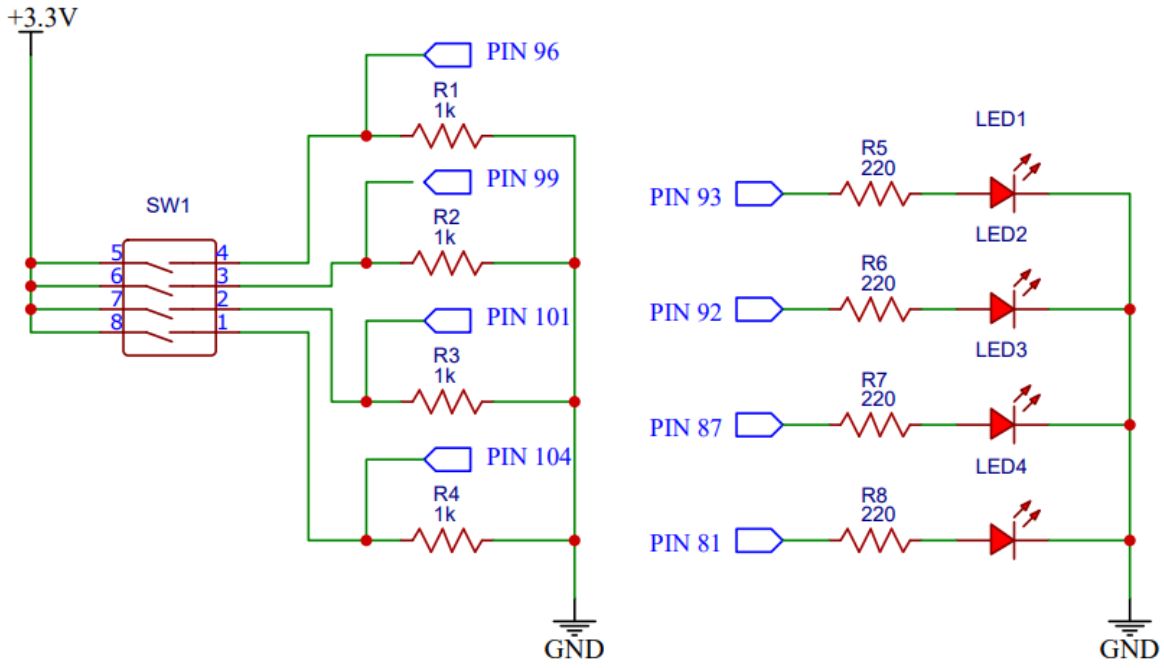


Figura 4.10 Diagrama esquemático para circuito multiplicador de 2 a 4 bits.

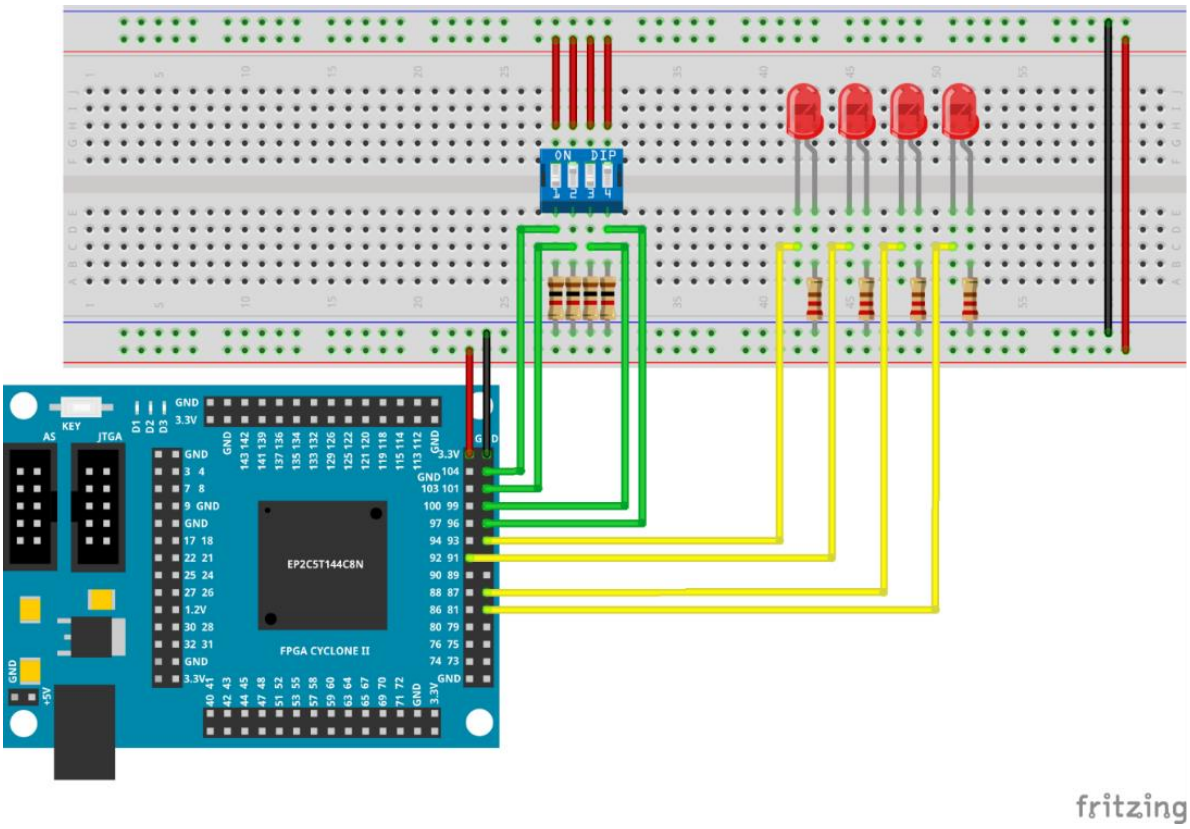


Figura 4.11 Diagrama de conexiones del circuito físico de la práctica 3 implementado en una placa de pruebas.

4.4 DECODIFICADOR DE 4 BITS A 7 SEGMENTOS

Esta práctica tiene el propósito de elaborar un decodificador para mostrar los números del 0 a 9 en un despliegue de 7 segmentos. En este decodificador, se usará un número binario de 4 bits que representa los números decimales de 0 a 9, el cual estará asociado a un dip-switch de 4 interruptores, y de esta manera se pueda manipular físicamente el decodificador a través los interruptores para escribir el número. El diagrama de bloques de la Figura 4.12 representa la estructura base de este circuito.

Figura 4.12 Diagrama de bloques de un decodificador de 4 bits a 7 segmentos.

Antes de implementar el código en VHDL, se tiene que elaborar la Tabla de verdad del decodificador para organizar la información. De igual forma que en la práctica anterior, la variable S será un número de 4 bits donde S_3 es el bit más significativo y es el número binario de 4 bits que representa los números decimales en código BCD. Por otro lado, están las letras desde A hasta G que encienden cada led de los segmentos del despliegue de para mostrar el número decimal de acuerdo con la Figura 4.13.

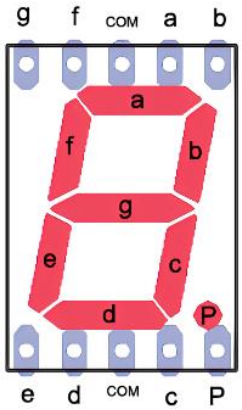


Figura 4.13 Despliegue de 7 segmentos cátodo común.

Esta información se organiza en la Tabla 4.8 donde la primera columna es el número decimal, las columnas de S representa ese número en binario y el resto de las columnas son cada uno de los segmentos del despliegue.

Tabla 4.8 Tabla de verdad de un decodificador de 4 bits a 7 segmentos.

Entradas					**Salidas**						
#	S_3	S_2	S_1	S_0	A	B	C	D	E	F	G
0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0
1	0	0	0	**1**	0	**1**	**1**	0	0	0	0
2	0	0	**1**	0	**1**	**1**	0	**1**	**1**	0	**1**
3	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0	0	**1**
4	0	**1**	0	0	0	**1**	**1**	0	0	**1**	**1**
5	0	**1**	0	**1**	**1**	0	**1**	**1**	0	**1**	**1**
6	0	**1**	**1**	0	**1**	0	**1**	**1**	**1**	**1**	**1**
7	0	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
8	**1**	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**
9	**1**	0	0	**1**	**1**	**1**	**1**	**1**	0	**1**	**1**

El primer paso para elaborar esta práctica será crear un nuevo proyecto con el nombrado *DISPLAY_7_SEGMENTS_DECODER* y el archivo VHDL principal se guarda con el mismo nombre.

El código tendrá una señal de entrada *S* asociada a un dip-switch de cuatro interruptores y será declarada como *STD_LOGIC_VECTOR* de 4 bits, mientras que los segmentos del despliegue estarán asociados a la señal de salida *7_SEG* que es declarada como un *STD_LOGIC*_VECTOR de 7 bits.

El código B.4 del apéndice será implementado en este circuito, entre las líneas 20 y 43 se usa la estructura lógica *case/when*, presentada en la sección 2.7, donde el argumento del *case* es la entrada *S* y cada uno de los *when* representa un caso para cada valor de *S*. Cada caso de esta estructura lógica asigna un valor a la salida *7_SEG* siguiendo la lógica de la Tabla 4.8.

La unidad secuencial que contiene a esta estructura lógica tiene una sensibilidad al cambio de la señal *S*, es decir, que cada vez que ocurra un cambio en cualquiera de los bits de *S* se ejecutará el *case/when* para mostrar el número en el despliegue. La entrada *S* y la salida *7_SEG* estarán asociados a los pines físicos del FPGA como lo indica la información de la Tabla 4.10 y se conectan con el resto de los componentes del circuito.

Se usarán los dispositivos listados en la Tabla 4.9, cada uno de los interruptores del dip-switch tienen que conectarse a una resistencia pull-down de acuerdo con el diagrama esquemático de la Figura 4.14 y los segmentos del despliegue deberán tener conectada una resistencia para evitar fundir los leds del dispositivo. El circuito deberá estar organizado de manera similar al diagrama de conexiones de la Figura 4.15.

Tabla 4.9 Lista de dispositivos para la práctica 4.

Dispositivo	**Valor**	**Cantidad**	**Símbolo en el diagrama**
Resistencia	220 Ω	7	R5, R6, R7, R8, R9, R10, R11
Resistencia	1 kΩ	4	R1, R2, R3, R4
Dip-Switch	4 interruptores	1	SW1
Despliegue 7 segmentos	Cátodo Común	4	SEG1

Tabla 4.10 Pines del FPGA asociados a los componentes del decodificador de 4 bits a 7 segmentos.

Entradas			**Salidas**		
Variable	**VHDL**	**PIN FPGA**	**PIN Display**	**En VHDL**	**PIN FPGA**
S_0	S[0]	96	**A**	D7SEG[6]	74
S_1	S[1]	99	**B**	D7SEG[5]	75
S_2	S[2]	101	**C**	D7SEG[4]	79
S_3	S[3]	104	**D**	D7SEG[3]	81
---	---	---	**E**	D7SEG[2]	87
---	---	---	**F**	D7SEG[1]	92
---	---	---	**G**	D7SEG[0]	93

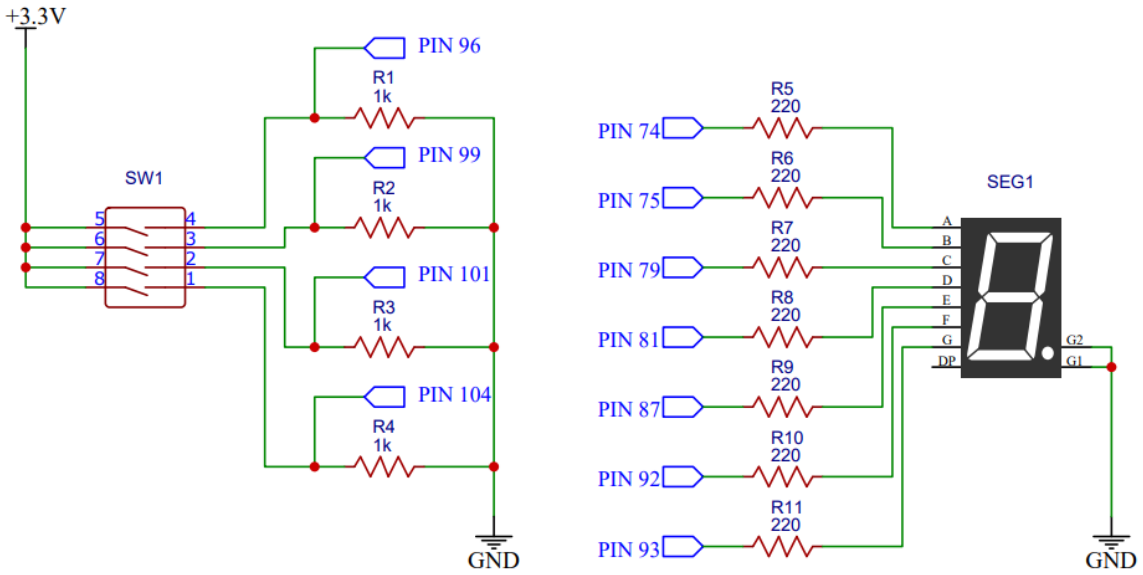


Figura 4.14 Diagrama esquemático del circuito decodificador de 4 bits a 7 segmentos.

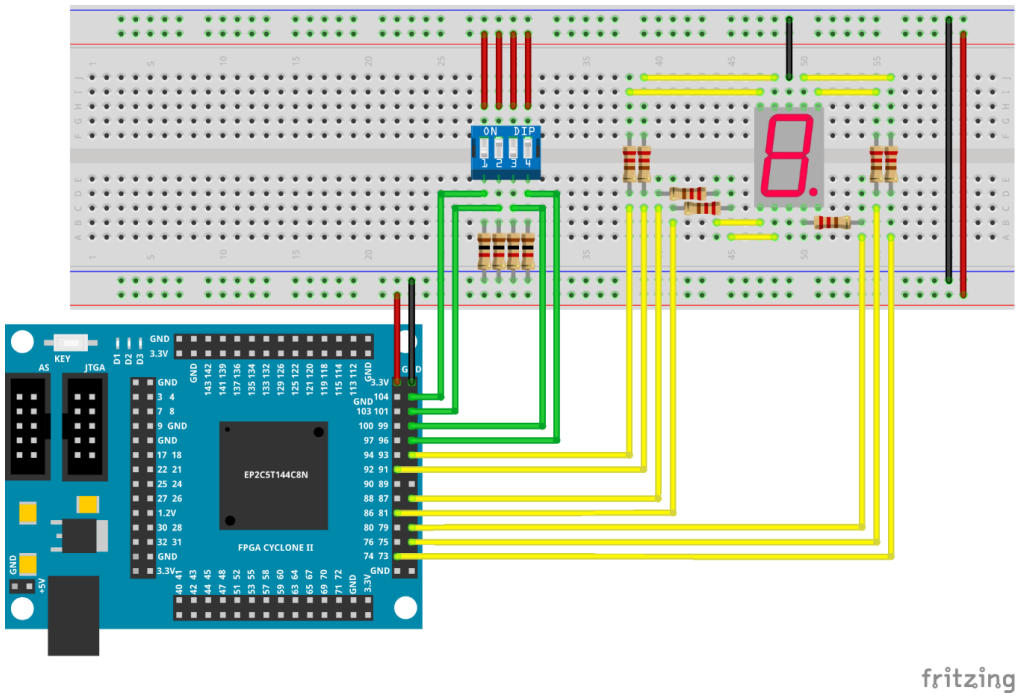


Figura 4.15 Diagrama esquemático del circuito físico de la práctica 4 implementado en una placa de pruebas.

4.5 CONTADOR ASCENDENTE Y DESCENDENTE

Esta práctica tiene el propósito de programar un contador desde 0 hasta 9 y que sea capaz de realizar el conteo de los números de forma ascendente y descendente. Un circuito lógico que permitirá implementar este contador es precisamente una máquina de estados que, en otras palabras, es un circuito secuencial conformado por diferentes estados a través del tiempo.

En la práctica anterior se implementó la estructura lógica *case/when* para hacer funcionar un decodificador de 4 bits a 7 segmentos. En este caso, se utilizará para hacer funcionar la máquina de estados que regirá el contador junto con el tipo de variable *type*, visto en la sección 2.5.4, donde cada etiqueta del conjunto representa en el código un estado del circuito secuencial y, al mismo tiempo, se crearán dos variables del mismo conjunto para representar el estado presente y el estado siguiente.

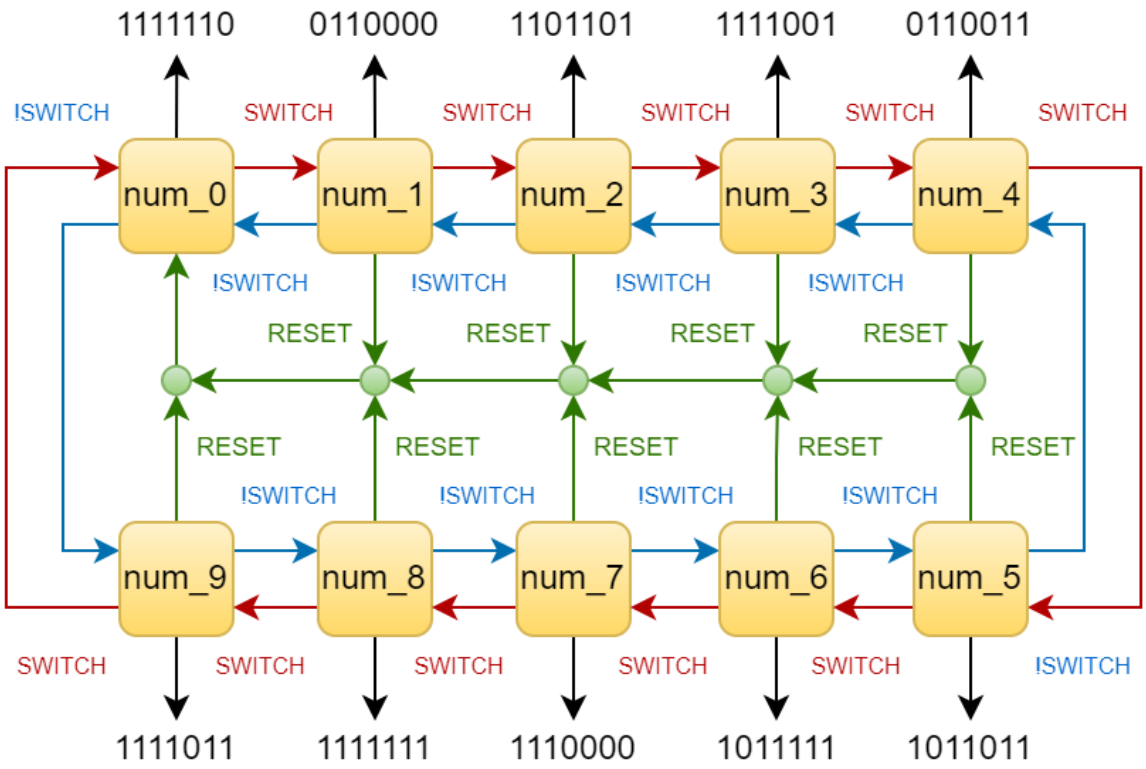


Figura 4.16 Diagrama de estados para contador reversible de 0 a 9.

La máquina de estados de este circuito seguirá el comportamiento del diagrama de estados que presenta la Figura 4.16. El circuito tiene 10 estados, donde en cada uno se imprime el número del contador en un despliegue de 7 segmentos. Cada estado tiene una salida que corresponde a un número binario de 7 bits sin significado matemático, donde cada bit está asociado a un segmento del despliegue, si el bit es 1, se encenderá el led del segmento, de lo contrario si el bit es 0, el segmento se apaga. Este comportamiento está documentado en la Tabla 4.8 de la sección anterior. También existe una señal, nombrada *SWITCH*, que controla el sentido del contador, si esta señal es 1, el contador avanza desde 0 hasta 9 y, de forma contraria, si la señal es 0, el contador avanza desde 9 hasta 0. Por último, la señal *RESET* si adquiere el valor 0, se reiniciará el contador llevando el circuito al estado *num_0* sin importar en donde se encuentre el contador.

Para elaborar el circuito de esta práctica, se utilizará la programación por diagrama de bloques para hacer más entendible la información. Se usarán tres bloques para elaborar el circuito: el primer bloque estará asociado al código B.5.2 que implementa la máquina de estados de la Figura 4.16; el segundo estará asociado código B.5.1 que corresponde a un divisor de frecuencia que es similar al código B.4 usado en la práctica 2 para reducir la frecuencia del FPGA de 50 MHz a un periodo de tiempo de 500 ms, por lo tanto el cambio entre un número y otro pueda ser visible; y el tercer bloque estará asociado al código B.2 del decodificador usado en la practica 4, porque la máquina de estados escribirá un número desde 0 hasta 9 representado en un número binario de 4 bits, pero este número se debe mostrar en el despliegue de siete segmentos.

Los tres códigos mencionados en el párrafo anterior se tienen que transformar a bloques para construir el diagrama en Quartus II, para ello la sección 3.4.2 explica el proceso de construcción de bloques a partir de código VHDL para programar el FPGA a partir de un diagrama de bloques y en la sección 3.4.3 se explica el proceso de construcción. El diagrama de la Figura 4.17 detalla la estructura que debe tener el diagrama de bloques que se implementa en Quartus II.

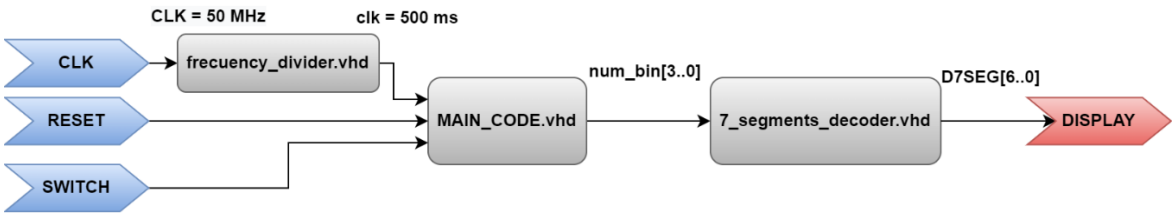


Figura 4.17 Diagrama de bloques para circuito contador decimal ascendente y descendente.

El primer paso para implementar este circuito digital, es crear un proyecto de diagrama de bloques siguiendo la metodología de la sección 3.4.1, después nombrar el proyecto y el archivo principal será el diagrama de bloques *.bdf* con el nombre *COUNTER_0_TO_9*. El código B.2 se usará nuevamente para crear el bloque correspondiente siguiendo los pasos de la sección 3.4.2. De igual manera, se crean los dos bloques restantes a partir de los códigos B.5.1 y B.5.2 del apéndice.

De acuerdo con la Figura 4.17, las entradas serán las señales *CLK*, *RESET* y *SWITCH*, que estarán asociadas a la señal de reloj del FPGA, a un botón pulsador y a uno de los interruptores del dip-switch respectivamente de acuerdo con la información de la Tabla 4.12. La salida *D7SEG* estará asociada al despliegue de siete segmentos como se implementó en la sección anterior.

Cuando se haya generado el proyecto y los bloques a partir del código VHDL, tal como lo indica la metodología de la sección 3.4.2, se deberán conectar siguiendo la estructura de los diagramas de bloques de las figuras 4.17 y 4.18, para ello se debe seguir los pasos indicados en la sección 3.4.3 y también se deben incluir las señales de entrada y salidas correspondientes.

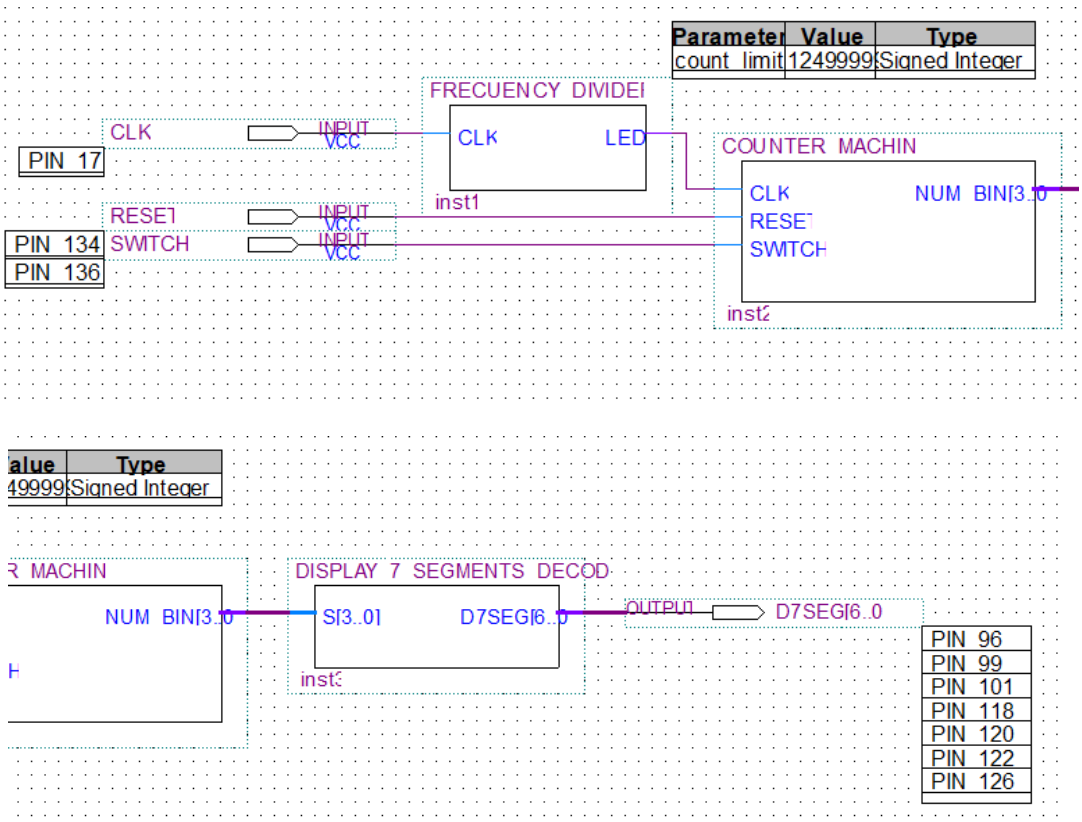


Figura 4.18 Diagrama de bloques elaborado en Quartus II.

El código B.5.1 que corresponde al bloque *FRECUENCY_DIVIDER* que es similar al código B.4 usado en la sección 4.4, con la diferencia que entre las líneas 7 y 9 se implementó la función *generic()*. En la sección 2.3.2 se establece que la función permite declarar variables paramétricas y en esta práctica se declara la variable *count_limit* que corresponde al límite del contador del divisor de frecuencia y de esta manera se puede modificar la frecuencia con la que trabaja la máquina de estados del bloque *COUNTER_MACHINE* desde el diagrama de bloques de Quartus II, sin tener que alterar el código del bloque.

En las líneas 16 y 17 del código B.5.1 se declara el conjunto *states* que contiene las etiquetas con las cuales se identificarán los estados de la unidad secuencial, desde *num_0* hasta *num_9* de acuerdo con el diagrama de estados de la Figura 4.16. En las líneas 18 y 19 se declaran dos variables del conjunto *states* nombradas *Q_bus* (estado presente), y *D_bus* (estado siguiente).

La primera unidad secuencial, ubicada entre las líneas 25 y 34, corresponde a un flip-flop tipo D que asigna el valor del estado futuro *D_bus* al estado presente *Q_bus* cada vez que existe un flanco ascendente en la señal *CLK*. Esta unidad secuencial será sensible a la señal de reloj de 500 ms obtenida del bloque *FRECUENCY_DIVIDER*, por lo tanto, se ejecutará el código del flip-flop cada vez que ocurra ese cambio y esto permitirá que la máquina de estados recorra todos los estados del circuito con ese valor de periodo. Los valores de *D_bus* y *Q_bus* dependerán de las condiciones de la segunda unidad secuencial que se ejecuta de manera paralela. La variable *Q_bus* se inicializa en *state_0,* que corresponde al punto de partida de la máquina de estados.

La segunda unidad secuencial nombrado *Machine,* situada entre las líneas 37 y 134, contiene la máquina de estados que controla el programa, el *case/when* recibe como argumento a la variable *Q_bus* y de acuerdo con el valor de la señal *SWITCH* recorrerá los estados *num_0, num_1, num_2, num_3, num_4, num_5, num_6, num_7, num_8, y num_9* ya sea de forma ascendente o descendente. Cada uno de los estados del circuito tiene una señal de salida nombrada *NUM_BIN,* que representa el valor en un número binario de 4 bits que al mismo tiempo es un número decimal dentro del rango desde 0 hasta 9. Esta señal establece la conexión con el bloque *7_SEGMENTS_DECODER* para realizar la conversión del número binario a un despliegue de 7 segmentos.

Código 4.1 Sección del código B.5.2 que corresponde estado "num_2" del circuito.

```
58    when num_2 =>
59
60      NUM_BIN <= "0010";
61      if(SWITCH = '0') then
62       D_bus <= num_3;
63      else
64       D_bus <= num_1;
65      end if;
66    end case;
```

Para explicar a detalle el funcionamiento de la máquina de estados, el código 4.1 muestra el caso cuando *Q_bus* se encuentra en el estado *num_2*, donde se muestra el número 2 decimal en el despliegue a través de la señal *NUM_BIN* escrito en número binario de 4 bits. Siguiendo la estructura de la Figura 4.16, el estado siguiente *D_bus* depende de una estructura *if/else* que determina si el valor de la entrada *SWITCH* es 0, se asigna el valor *num_3* a la variable *D_bus*, por lo tanto, el estado siguiente es mostrar el número 3 en el despliegue. De lo contrario, si *SWITCH* es 1, el valor *state_1* se asigna a la variable *D_bus* y la máquina de estados se regresará a mostrar el número 1.

Para construir el circuito, se usarán los dispositivos listados en la Tabla 4.12, los interruptores del dip-switch se conectan a una resistencia pull-down de acuerdo con el diagrama esquemático de la Figura 4.19 y en los segmentos del despliegue se conecta una resistencia para evitar fundir los leds del dispositivo. El circuito deberá estar organizado de manera similar al diagrama de conexiones de la Figura 4.20.

Tabla 4.11 Lista de dispositivos para la práctica 5.

Dispositivo	Valor	Cantidad	Símbolo en el diagrama
Resistencia	10 kΩ	4	R1
Resistencia	1 kΩ	4	R2
Resistencia	220 Ω	7	R3, R4, R5, R6, R7, R8, R9
Condensador cerámico	220 nF	1	C1
Dip-Switch	4 interruptores	1	SW1
Botón pulsador	--	1	KEY1
Display 7 segmentos	Cátodo Común	4	SEG1

Tabla 4.12 Pines del FPGA asociados a los componentes del contador decimal ascendente-descendente.

Entradas			**Salidas**		
Variable	VHDL	PIN FPGA	PIN Display	En VHDL	PIN FPGA
CLK	CLK	17	A	D7SEG[6]	74
RESET	RESET	104	B	D7SEG[5]	75
SWITCH	SWITCH	101	C	D7SEG[4]	79
---	----	---	D	D7SEG[3]	81
---	---	---	E	D7SEG[2]	87
---	---	---	F	D7SEG[1]	92
			G	D7SEG[0]	93

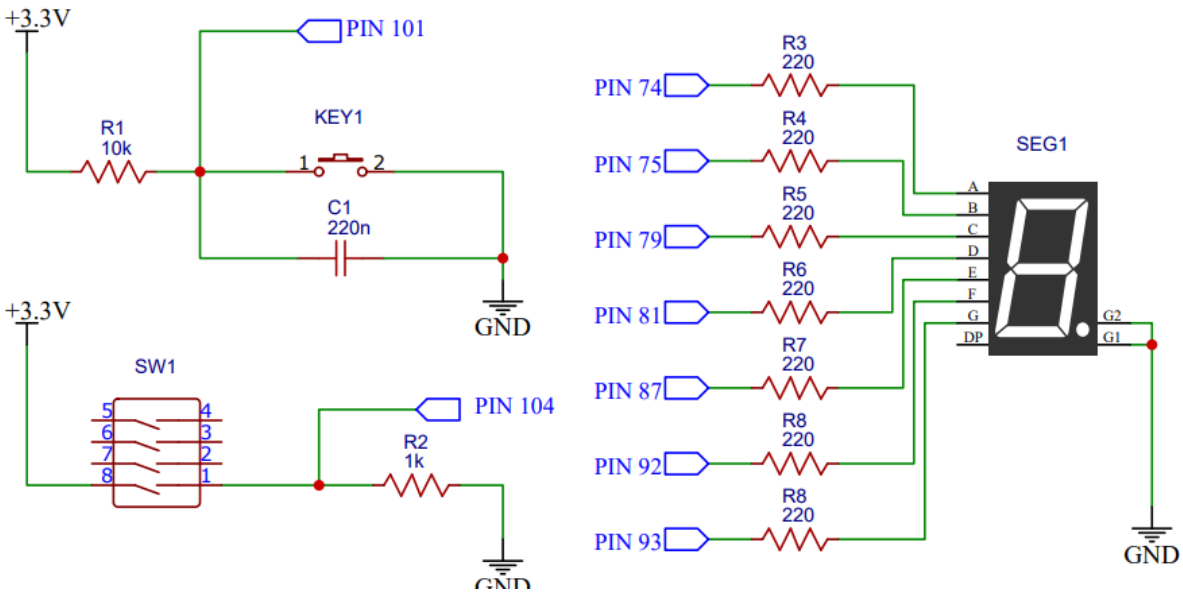


Figura 4.19 Diagrama esquemático para contador reversible de 0 a 9.

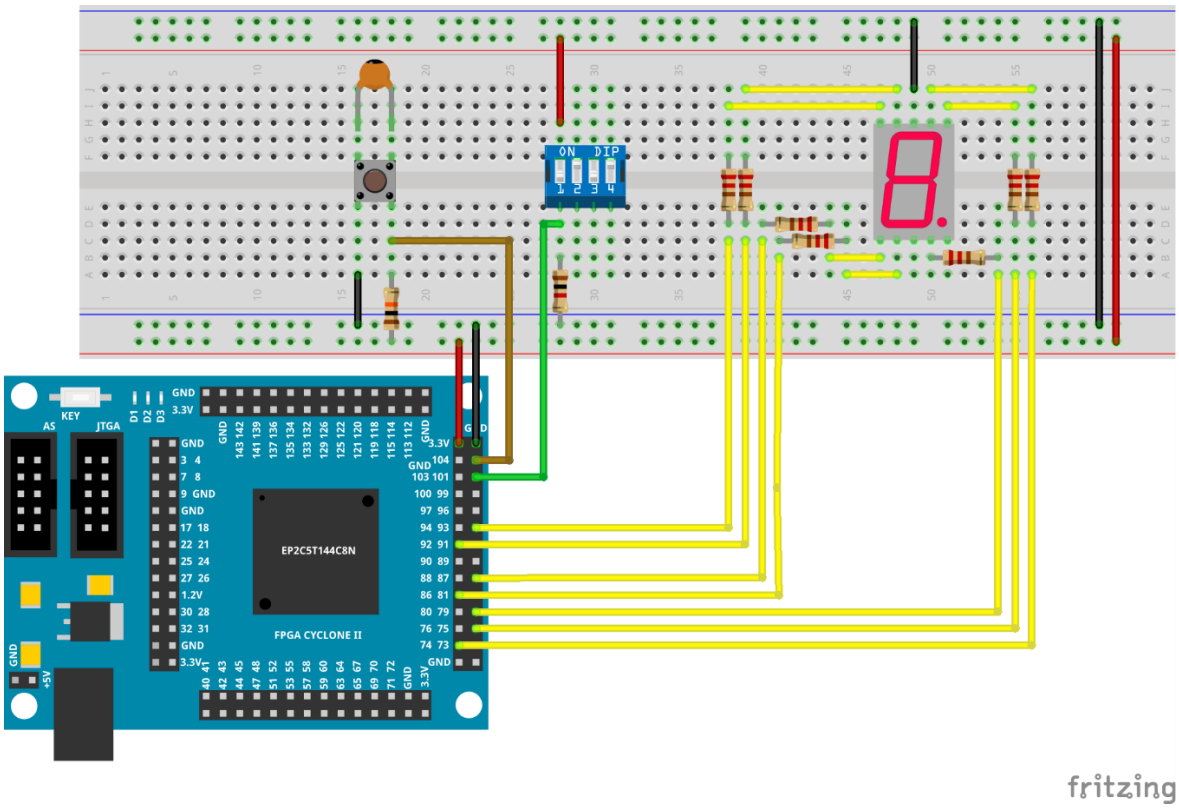


Figura 4.20 Diagrama de conexiones del circuito físico de la práctica 5 implementado en una placa de pruebas

.

CAPÍTULO 5
HARDWARE CON FPGA

Las prácticas anteriores consistieron en emplear circuitos combinacionales y secuenciales, además se implementó la programación con código VHDL y diagrama de bloques. Por su parte, las prácticas de este capítulo usarán al FPGA como un controlador para establecer comunicación con dispositivos periféricos como un motor, una pantalla de cristal líquido o un convertidor analógico a digital. Cada uno de los códigos de los proyectos presentados a continuación se pueden consultar en el apéndice B. También se usarán los archivos de biblioteca del apéndice C, que contiene bloques con sus respectivos códigos VHDL, que permitan establecer la comunicación entre los dispositivos antes mencionados y el FPGA y, de esta manera se simplificará la elaboración de cada práctica.

5.1 CONTROLADOR DE UN MOTOR PASO A PASO

Esta práctica tiene el objetivo de usar al FPGA para controlar el sentido y la velocidad de giro de un motor paso a paso. Para realizar este controlador, es evidente que se recurra a un circuito secuencial donde cada estado representa el paso del motor y la polarización de las bobinas, mientras que el estado siguiente indicará en que el sentido gira el motor. Un divisor

de frecuencia será usado para controlar la velocidad de trancisión de un estado a otro y, por lo tanto, la velocidad de giro del motor paso a paso.

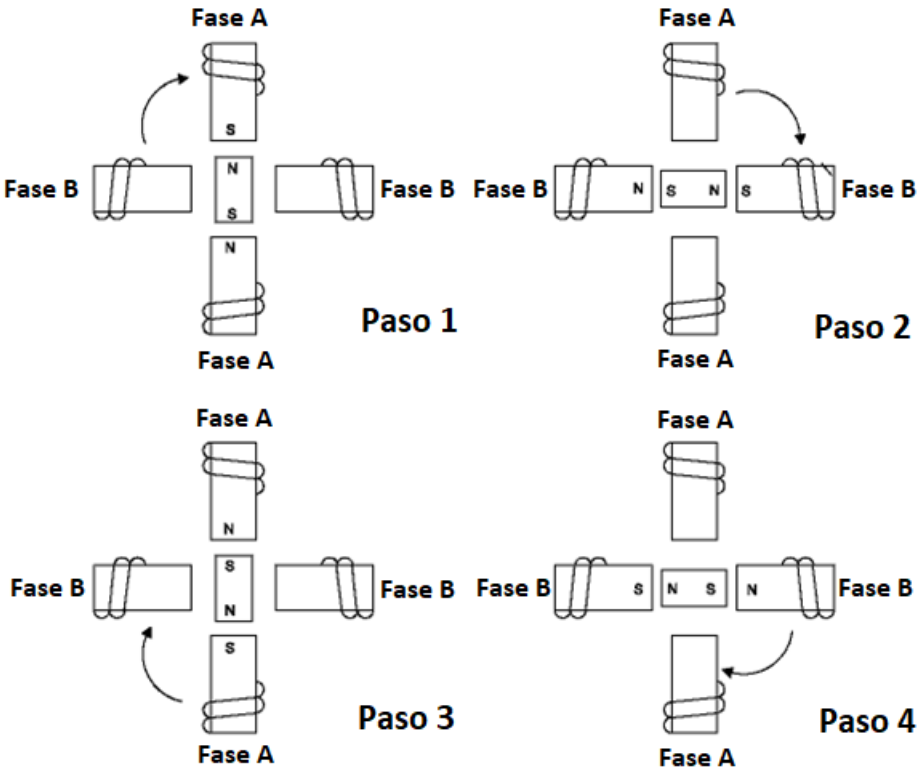


Figura 5.1 Fases para girar el rotor del motor paso a paso.

Los esquemas de la Figura 5.1 muestran el funcionamiento de un motor paso a paso de dos fases. Cada fase está conformada por un sistema de dos bobinas complementarias, es decir, si una de las dos bobinas se polariza en norte, la bobina complementaria se polariza en sur y viceversa.

Para simplificar la comprensión del funcionamiento, se puede considerar que un motor paso a paso bipolar tiene cuatro bobinas, de acuerdo con la Figura 5.1, las fases (o polos) se les asignan las letras A y B mientras que las bobinas restantes reciben el nombre de $\bar{A}$ y $\bar{B}$. Las bobinas del estator se polarizan de tal forma que el campo magnético del imán en el rotor se alinee con el campo magnético del estator generado por las bobinas. Para hacer

girar al motor se debe crear una secuencia de tal manera que las bobinas puedan mantener un giro constante en el rotor.

Para el diseño de la máquina de estados se contemplan ocho estados deben ser "enumerados" de 0 a 7, cuatro establecen el giro en sentido horario, mientras que los cuatro restantes establecen el giro en sentido antihorario. La Figura 5.2 presenta de manera grafica la máquina de estados que se implementará en el FPGA para controlar el motor paso a paso.

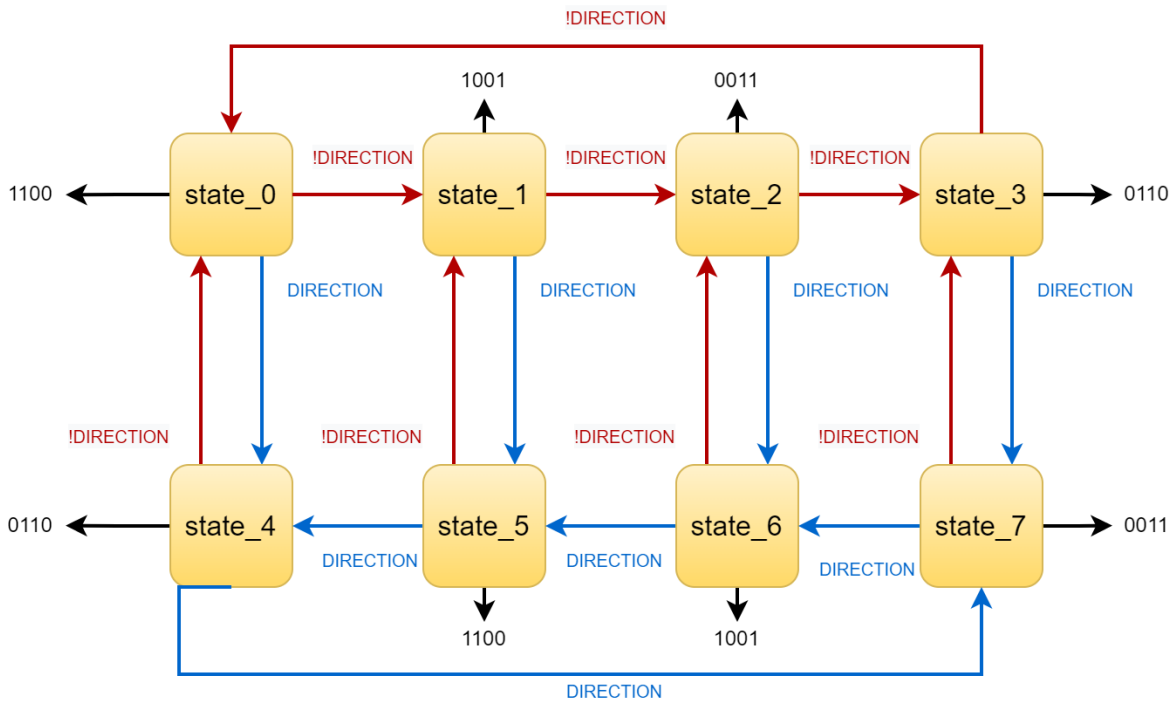


Figura 5.2 Máquina de estados para el controlador de motor paso a paso.

Cada estado del circuito tiene una señal de salida que polariza las bobinas del motor paso a paso la cual se representa como un número binario de 4 bits sin significado matemático. Los primeros dos bits, contando desde el bit más significativo al menos significativo, representan las bobinas A y B mientras que los dos últimos son las bobinas complementarias $\bar{A}$ y $\bar{B}$. Cuando un bit esté en 1, la bobina se polariza en sur, mientras que si está en 0 se polariza en norte.

La señal *DIRECTION* establece el sentido de giro y se puede invertir en cualquier instante del tiempo, sin importar el estado donde se encuentre el circuito secuencial

La señal *DIRECTION* establece el sentido de giro y éste puede cambiar en cualquier momento del tiempo sin importar el estado donde se encuentre el circuito secuencial.

La Tabla 5.1 contiene la información de la Figura 5.2 organizada de tal manera que se pueda interpretar como una tabla de verdad de un circuito secuencial, donde existe el estado presente y el estado futuro correspondiente.

Tabla 5.1 Tabla de verdad para circuito controlador de motor paso a paso.

	Estado presente				**Estado siguiente**			
DIRECTION	***A***	***B***	$\overline{A}$	$\overline{B}$	***A***	***B***	$\overline{A}$	$\overline{B}$
0	**1**	**1**	0	0	**1**	0	0	**1**
0	**1**	0	0	**1**	0	0	**1**	**1**
0	0	0	**1**	**1**	0	**1**	**1**	0
0	0	**1**	**1**	0	**1**	**1**	0	0
1	**1**	**1**	0	0	0	**1**	**1**	0
1	0	**1**	**1**	0	0	0	**1**	**1**
1	**1**	**1**	0	0	**1**	0	0	**1**
1	**1**	0	0	**1**	**1**	**1**	0	0

Para comenzar con el diseño del circuito lógico de esta práctica, el diagrama de la Figura 5.3 que corresponde al diagrama de bloques que describe la estructura del circuito, se tienen tres entradas: *CLK* que es la señal de reloj de 50 MHz; *DITECTION* que está asociada a un interruptor; y *SPEED* que es el valor de la velocidad de giro representada por un numero binario de 8 bits para obtener un rango de velocidad desde 0 hasta 255.

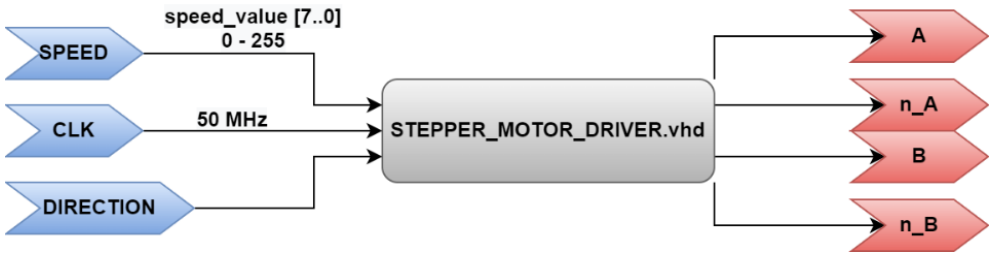


Figura 5.3 Diagrama de bloques del controlador de motor paso a paso.

El primer paso es crear un proyecto y un archivo de diagrama de bloques, como lo explica la sección 3.4.1, y nombrarlo como *STEEPER_MOTOR_DRIVER*.

Para elaborar el bloque para el controlador del motor paso a paso se usa el código B.6 del apéndice y después se crea el bloque usando la metodología de la sección 3.4.2. La Figura 5.4 muestra el bloque resultante de este proceso, donde las entradas y salidas son las señales indicadas en la Figura 5.3.

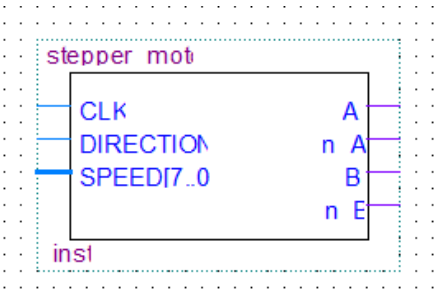


Figura 5.4 Bloque para el código del motor a paso a paso.

El código B.6 se divide en tres unidades secuenciales. La primera consiste en un divisor de frecuencia que reduce los 50 MHz de la señal de reloj a un valor de frecuencia variable que permite controlar la velocidad de la máquina de estados y, en consecuencia, la velocidad del motor.

Código 5.1 Función programada para mapear la velocidad del motor paso a paso.

```
19  function MAP_MOTOR( VEL : integer )
20    return integer is
21     variable counter : integer ;
22    begin
23     if(VEL > 0 and VEL < 44) then
24          counter := VEL*(-5813) + 499999;
25     elsif(VEL > 43 and VEL < 86) then
26          counter := VEL*(-2976) + 377975;
27     elsif(VEL > 85 and VEL < 171) then
28          counter := VEL*(-735) + 187499;
29     elsif(VEL > 172 and VEL < 256) then
30          counter := VEL*(-245) + 104199;
31         end if;
32    return counter;
33   end function;
```

En la sección 4.2 de la práctica 2 se elaboró un divisor de frecuencia usando un contador que almacena el número de instrucciones ejecutadas cada 20 ns, dicho contador tiene un límite establecido en 12,499,999 y al llegar a este valor se cambia el estado de la señal de salida de 0 a 1 (y viceversa), de esta manera se obtiene señal de reloj de 2 Hz. En esta práctica, el valor límite del contador será variable y la frecuencia obtenida modificará la velocidad con la que se ejecuta el circuito secuencial y, por lo tanto, se modifica la velocidad de giro del motor.

El código 5.1 corresponde a una sección, situado entre las líneas 19 y 33, del código B.6, allí se programó una función en VHDL para realizar el proceso de mapeo donde, en base a datos experimentales, donde el rango de la variable independiente es desde 1 hasta 255, porque se representa el valor de una señal PWM de 8 bits, mientras que el rango de la variable dependiente es desde 500,000 que disminuye hasta llegar a 41,724, y que representa al valor límite del contador del divisor de frecuencia. El comportamiento es inverso debido a que entre menor sea el valor del límite del contador, la frecuencia obtenida aumenta, por lo tanto, la velocidad del circuito secuencial, y por ende la velocidad del motor, aumenta. El valor mínimo y máximo de este divisor de frecuencia es desde 50 Hz (20 ms) hasta 600 Hz (1.66 ms) respectivamente. Estos datos experimentales solo aplican para un motor paso a paso bipolar 28BYJ-48.

La segunda unidad secuencial del código corresponde a un flip-flop que controla el avance del estado presente y el estado siguiente, representados por las variables *Q_bus* y *D_bus* respectivamente y declaradas como tipo de variable *states*. El flip-flop se implementa de la misma manera que en la práctica de la sección 4.5, pero excluyendo el reinicio.

La tercera unidad secuencial es la máquina de estados de la Figura 5.2 implementada en una estructura lógica *case/when*. Cada estado del circuito se representa por una etiqueta del conjunto *states,* que proviene del tipo de variable *type* explicado en la sección 2.5.4.

La entrada *SPEED* del bloque de la Figura 5.4 representa el valor de la velocidad del motor, como se mencionó anteriormente, pero no está asociada a un componente físico del circuito. Esta entrada se asocia con un bloque tipo *lpm_constant* en el cual se asigna un valor numérico y que lo representa con un número. Para generar este valor constante, se busca la opción

Symbol tool para insertar los bloques del espacio de trabajo haciendo clic en el icono indicado en la Figura 5.5 que corresponde a la barra de herramientas.

Figura 5.5 Herramienta para insertar bloques.

En la ventana de la Figura 5.6 muestra una sección de la ventana *Symbol tool* la cual es un directorio con dos carpetas, la primera corresponde al del proyecto que se está programado, mientras que la segunda carpeta contiene los archivos de biblioteca que almacenan los diferentes bloques predefinidos por Quartus II. El bloque que se busca para este proyecto es *lpm_constant* el cual se accede dentro de la carpeta *megafunctions* y dentro la subcarpeta *gates*.

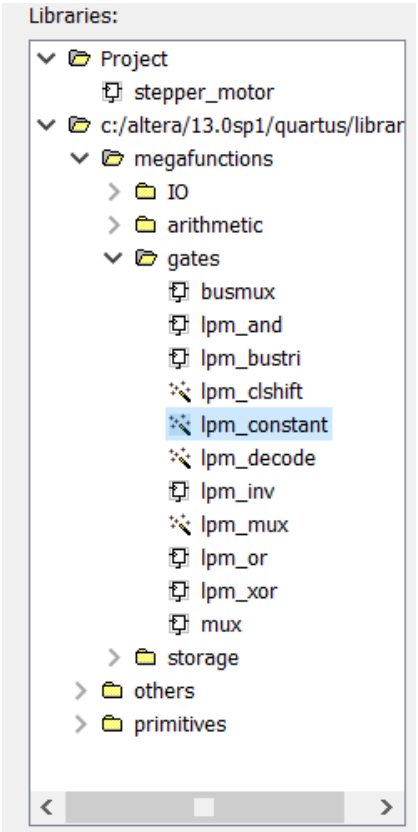


Figura 5.6 Bloque lpm_constant.

Se selecciona la opción *lpm_constant*, después se desplegará una nueva ventana la cual permitirá generar el bloque para el valor constante. Antes, se tiene que asignar un nombre al bloque como *speed_val* como lo indica la Figura 5.7.

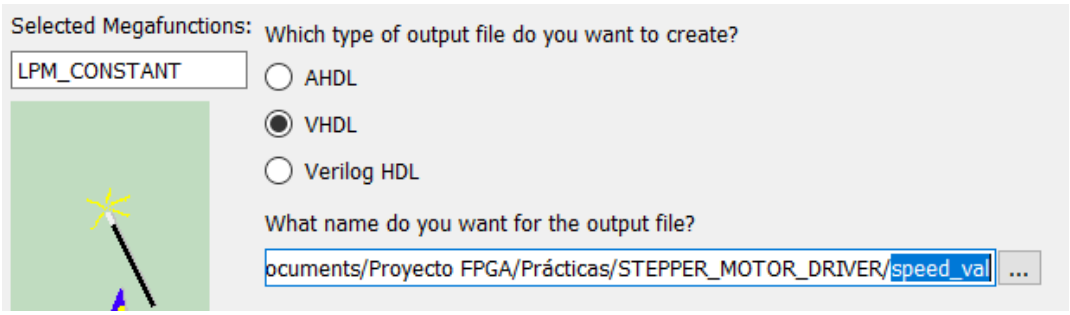


Figura 5.7 Configuración del bloque "speed_val".

Después, se hace clic en el botón *Aceptar* y la ventana abre otra sección, como lo muestra la Figura 5.8, donde se asigna el valor constante representado por un número binario. Se ajusta ese número binario con una extensión de 8 bits para tener un rango de valores decimales desde 0 hasta 255. El valor de la constante se fija en 128 para obtener una velocidad media en el motor.

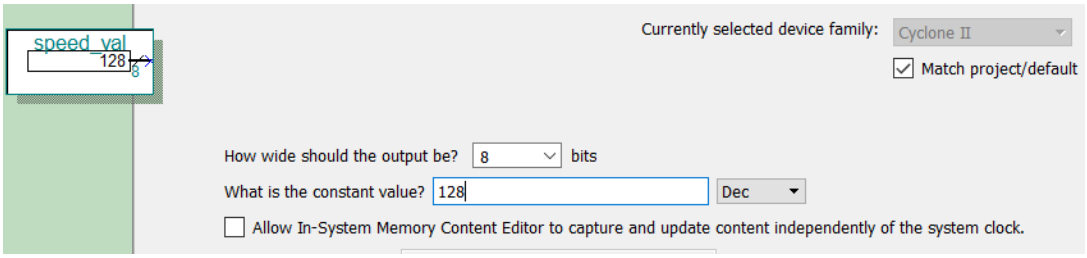


Figura 5.8 Diagrama de bloques para motor paso a paso.

Cuando se haya generado el bloque *speed_val* se realiza la conexión con el bloque *stepper_motor* con la entrada *SPEED* como se representa en la Figura 5.9.

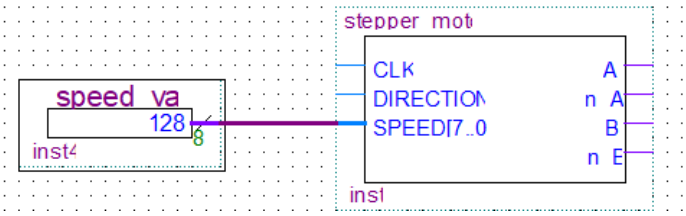


Figura 5.9 Conexión entre "speed_val" y "stepper_motor".

El siguiente paso es crear las entradas para las señales *CLK, RESET y SWITCH*, y de igual manera las salidas *A, n_A, B y n_B.* La Figura 5.10 muestra el diagrama de bloques resultante de estas conexiones. Con ayuda de la herramienta *"Pin Planner"* se asignan las salidas del diagrama de bloques a los pines físicos del FPGA de acuerdo con la información presentada en la Tabla 5.4. La sección 3.6 explica detalladamente el proceso de asignación de pines.

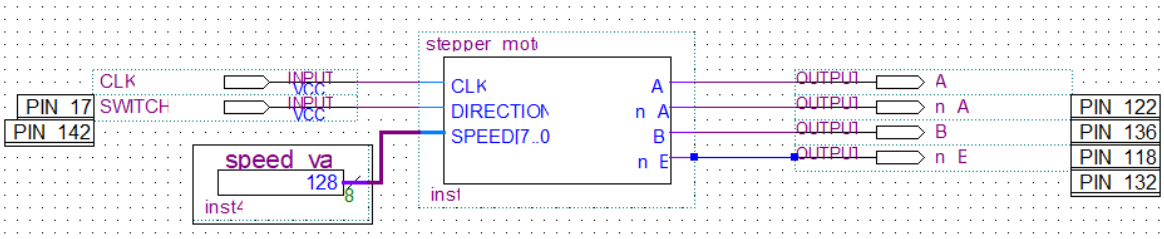


Figura 5.10 Diagrama de bloques para el controlador de motor paso a paso.

Cuando se cargue el código en el FPGA, el motor deberá girar en un sentido, si el interruptor (asociado la entrada DIRECTION) se acciona en 1, se deberá invertir el sentido de giro. Si se cambia el valor de la contante *speed_val* por el número 64, se notará que el motor gira más lento y, contrariamente, si se aumenta el valor a 190 el motor girará más rápido. Es importante señalar que la velocidad máxima está representada por 255 mientras que 1 es la velocidad mínima y 0 representa cuando el motor no gira.

En la construcción del circuito físico, se requiere de una fuente de alimentación de 9 V adicional a la que alimenta al FPGA para alimentar al motor. Recordando que el FPGA opera con valores TTL de 3.3 V, se recurre a un circuito de potencia para que las señales de los pines accionen el motor. La Tabla 5.3 lista los dispositivos necesarios para esta práctica y los diagramas de las figuras 5.11 y 5.12 muestran las conexiones necesarias.

Tabla 5.2 Lista de dispositivos para la práctica 6.

Dispositivo	**Valor**	**Cantidad**	**Símbolo en el diagrama**
Resistencia	1 kΩ	4	R1
Dip-Switch	4 interruptores	1	SW1
Puente H	L293D	1	U1
Motor paso a paso	28BYJ-48	1	M1
Batería	9 V	1	vcc

Tabla 5.3 Pines del FPGA asociados a los componentes del controlador de motor paso a paso.

Entradas			**Salidas**		
Variable	VHDL	PIN FPGA	Motor	VHDL	PIN FPGA
CLK	CLK	17	5 (azul)	A	122
SWITCH	SWITCH	142	3 (amarillo)	n_A	136
SPEED	speed_val	--	4 (rosa)	B	118
--	--	--	2 (naranja)	n_B	132

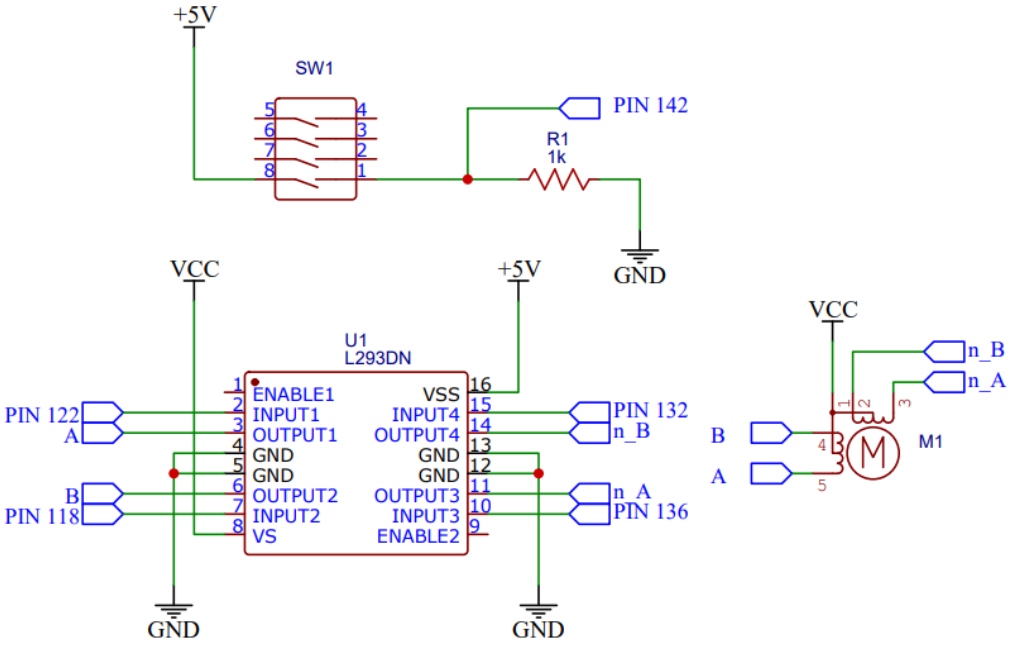


Figura 5.11 Diagrama esquemático del controlador para motor paso a paso.

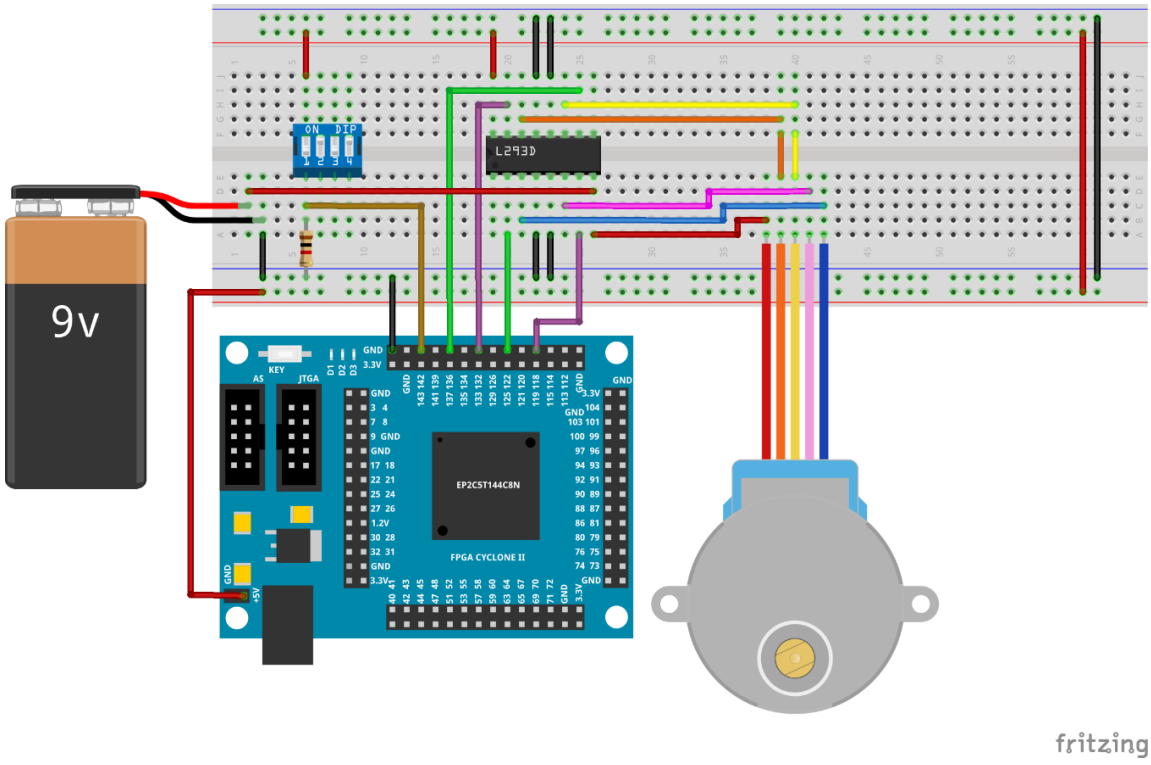


Figura 5.12 Diagrama de conexiones del circuito físico de la práctica 6 implementado en una placa de pruebas.

5.2 GENERADOR DE UNA SEÑAL PWM

En la práctica anterior se utilizó el FPGA como controlador de un motor paso a paso, con el cual se logró variar la velocidad y el sentido de giro de este. En esta práctica se tiene un propósito similar, se usará un motor de corriente directa para controlar su velocidad. El circuito empleado en el FPGA será un generador de modulación por ancho de pulso (PWM) para modificar la velocidad del actuador.

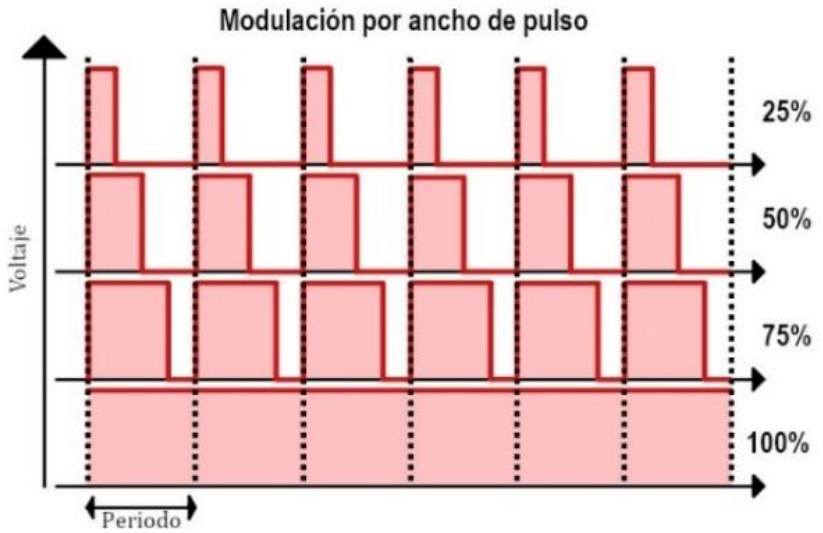


Figura 5.13 Gráfica comparativa de la variación del ancho de pulso.

La señal PWM se caracteriza por ser una onda cuadrada que, a diferencia de las señales de reloj, el ciclo de trabajo es variable mientras que el voltaje de la señal es constante, tal y como lo muestra la gráfica de la Figura 5.13. Cuando la señal PWM se aplica para alimentar un motor, esta característica permite modificar la velocidad del actuador sin modificar el voltaje de alimentación.

Las entradas del circuito que se implementará en el FPGA, se dividen en dos grupos: el primero son las señales físicas como se han empleado en las prácticas interiores; el segundo grupo son variables paramétricas que no están asociadas a los componentes físicos del circuito.

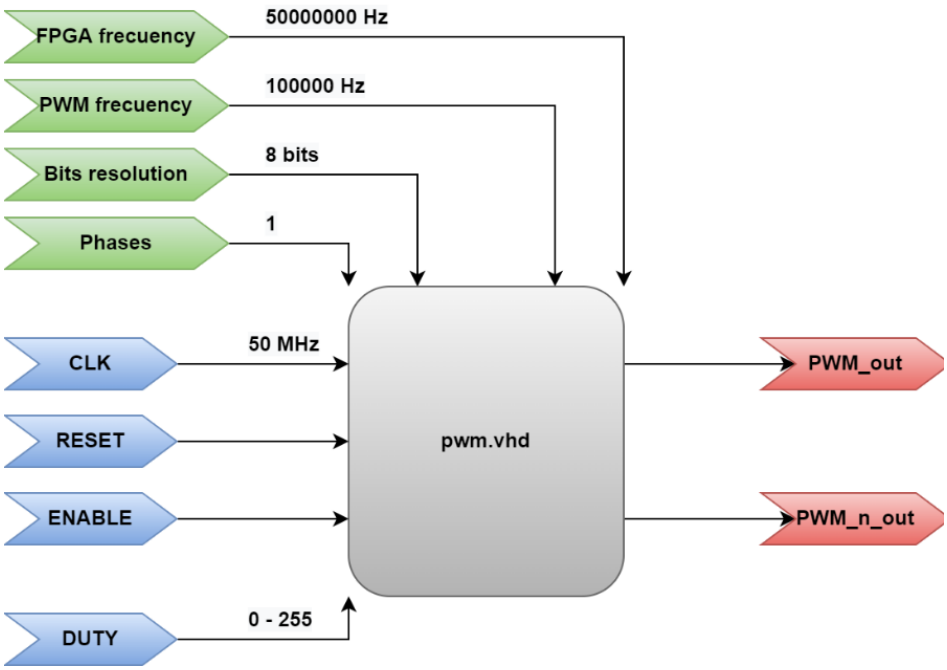


Figura 5.14 Diagrama de bloques para el generador de PWM.

La estructura del circuito tendrá ocho entradas y dos salidas. La Figura 5.14 muestra en color azul las entradas que representan señales físicas, mientras las que se enmarcan en color verde corresponden a las que están asociadas a variables paramétricas dentro del mismo proyecto y que no tienen un significado físico. Por su parte, las salidas son señales de un silo bit que representan a la señal PWM y la misma salida invertida.

El primer paso para elaborar el circuito de esta práctica es crear el proyecto y el archivo del diagrama de bloques con el nombre *PWM_MOTOR*. Después, se construye el bloque que describe el circuito. El bloque debe contener el código citado en la sección C.2 del apéndice y existen dos maneras para crearlo e incluirlo en el proyecto.

La primera sería generar el bloque como lo indica la sección 3.4.1 usando el código de la sección C.2 con el nombre *pwm*, que es el mismo que se asigna a la entidad dentro del código VHDL.

La segunda manera es acceder a la carpeta virtual del apéndice D y usar la dirección *FPGA "Handbook/libraries/PWM Generator/"* donde se encontrará el archivo VHDL y el archivo del bloque, los cuales se descargan e incluyen en la carpeta del proyecto en Quartus II. Se deben descargar y agregar a la carpeta del proyecto como lo muestra la Figura 5.15.

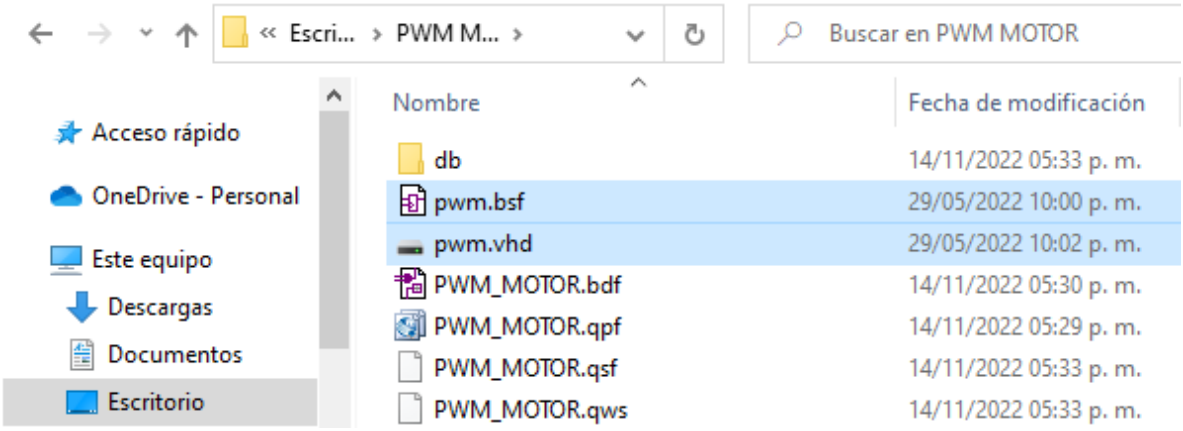


Figura 5.15 Archivos descargados de la carpeta virtual.

Después, en el navegador del proyecto, que es la pestaña del lado derecho del software de Quartus II nombrada como *"Proyect Navigator"*, se hace clic en la pestaña *"Files"* (situada en la parte inferior) y después se hace clic derecho en el icono *"Files"* y se selecciona la opción *"Add/Remove Files in Proyect ..."* como lo indica la Figura 5.16

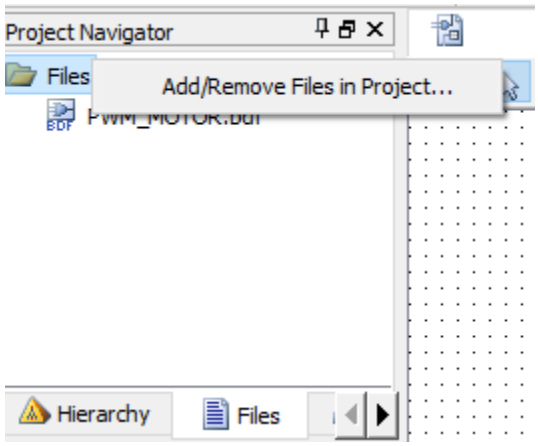


Figura 5.16 Navegador del proyecto.

Se abrirá la ventana de la Figura 5.17 donde se tiene que hacer clic en el botón *"..."* y después de se abrirá el navegador de archivos de Windows donde se debe seleccionar solamente el archivo VHDL con terminación *.vhd* y se hace clic en el botón *"Abrir"*. Luego, aparecerá el nombre del archivo, como lo muestra la Figura 5.17, en el campo *"File name"*, después se hace clic en el botón *"Add"* donde aparecerá en nombre del archivo en la lista de abajo, y finalmente se hace clic en el botón *"Ok"*.

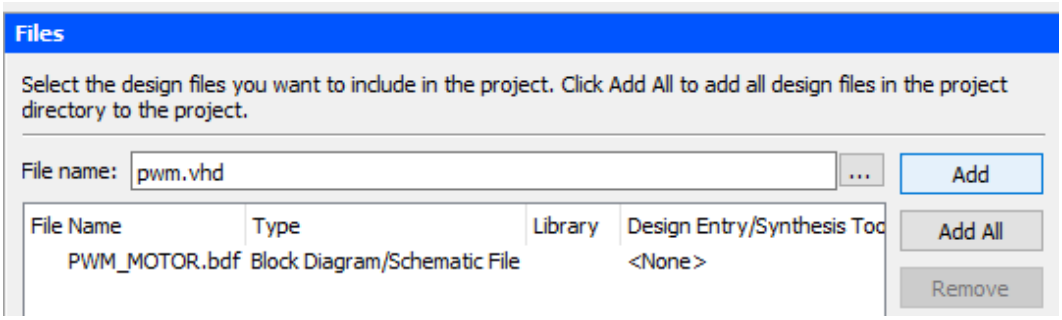


Figura 5.17 Ventana para añadir un archivo con al proyecto.

La Figura 5.18 muestra al archivo añadido en el navegador del proyecto, que significa que ya forma parte de éste y se podrá disponer del bloque al cual está asociado el código VHDL que contiene el archivo.

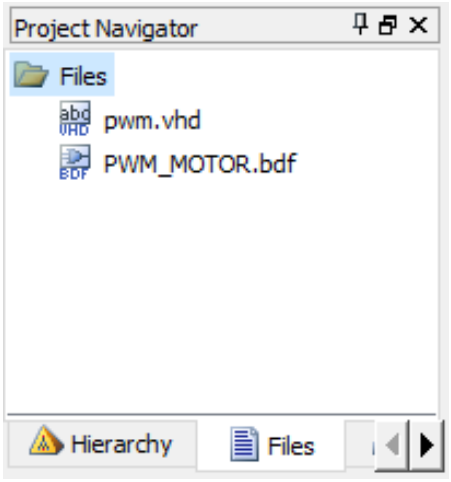


Figura 5.18 Archivo "pwm.vhd" incluido en el proyecto "PWM_MOTOR".

De esta manera se incluyó en el proyecto un bloque del archivo de biblioteca de la sección C.2 del apéndice, este método será recurrente en las prácticas subsecuentes debido a que el número de bloques involucrados en el circuito será mayor y para evitar repetir varias veces el proceso de la sección 3.4.1.

Para incluir el bloque en el proyecto, el siguiente paso es hacer clic en el siguiente icono *"Symbol tool"* ubicado en la barra de herramientas como lo muestra la Figura 5.20:

Figura 5.19 Herramienta "Symbol tool" en la barra de herramientas.

Se abrirá la ventana de la herramienta, donde aparecerá el directorio que contiene la carpeta del proyecto, se debe hacer clic en *"Proyect"* para desplegar el contenido y buscar el bloque *pwm* insertado anteriormente, como lo muestra la Figura 5.20.

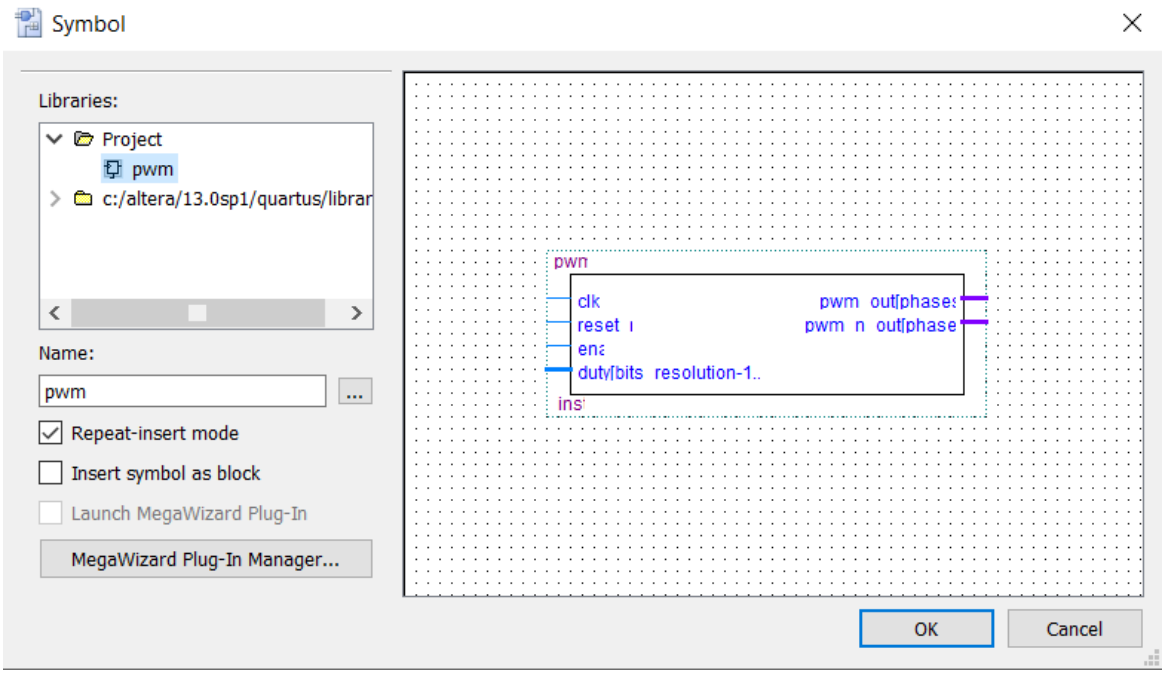


Figura 5.20 Vista previa del bloque generador de PWM.

Se hace clic sobre el icono del bloque en el navegador de la ventana mostrada en la Figura 5.21 y después se hace clic en el botón *"OK"*. Después la ventana se cerrará y el cursor tomará la forma del bloque seleccionado y al hacer clic en cualquier parte del diagrama, se generará el bloque seleccionado y estará listo para realizar las conexiones siguientes. La Figura 5.21 muestra al bloque insertado en el diagrama de Quatus II.

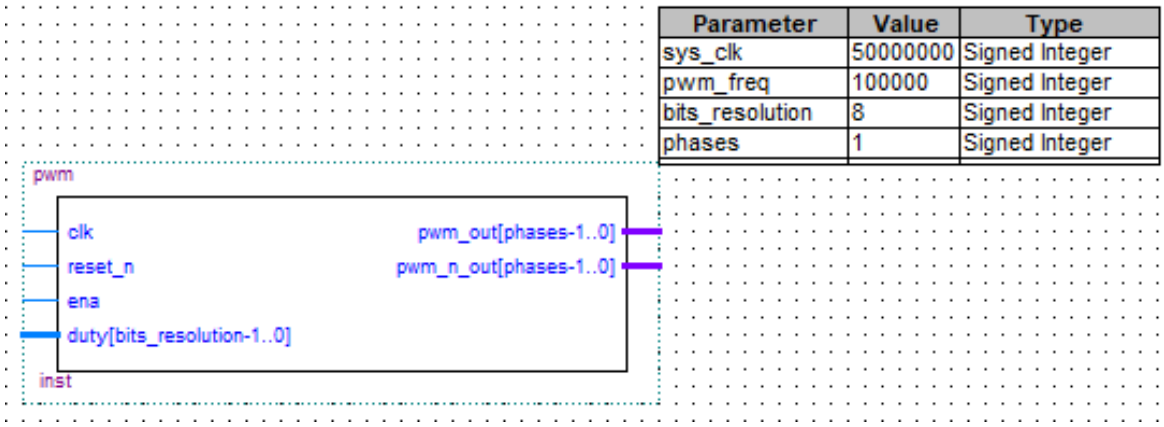


Figura 5.21 Bloque "pwm".

Se deberá crear un código citado en el apéndice B1 que permita generar señales PWM donde se puede establecer la frecuencia del PWM, la fase y resolución en bits, la cual estará establecida en 8 bits.

La estructura del bloque de la Figura 5.12 contiene cuatro entradas: la señal *CLK* representa a la señal de 50 MHz; la señal *RESET* es la que establece el reinicio del circuito; la señal *ENABLE* es el que habilita el funcionamiento de este; y la señal *DUTY* establece el valor de la señal PWM que se definirá con un número binario de 8 bits y, por lo tanto, tendrá un rango de valores desde 0 hasta 255. Por otro lado, las dos salidas del bloque son: *pwn_out* que es la señal PWM; y *pwn_n_out* que es la señal invertida.

Código 5.2 Sección del código del bloque "pwm".

```
40 entity pwm is
41   generic(
42       sys_clk         : integer := 50_000_000;
43               --system clock frequency in hz
44       pwm_freq        : integer := 100_000;
45               --pwm switching frequency in hz
46       bits_resolution : integer := 8;
47               --bits of resolution setting the duty cycle
48       phases          : integer := 1);
49               --number of output pwms and phases
```

En la sección 2.3.2 se explica la estructura de la función *generic()* que permite declarar variables paramétricas. La tabla que aparece en la Figura 5.21 muestra a las variables paramétricas declaradas en el código VHDL del bloque *pwm*, y el código 5.2 muestra las líneas donde se declaran las cuatro variables paramétricas: *sys_clk* corresponde al valor de la frecuencia del FPGA en Hz, *pwm_frec* es la frecuencia de la señal PWM en Hz, *bits_resolution* es la resolución en bits de la señal y *phases* es el número de salidas de señal de PWM. Las cuatro variables paramétricas se corresponden en el mismo orden con las entradas paramétricas planteadas en la Figura 5.14, enmarcadas en color verde. La ventaja de usar variables paramétricas en la programación con diagrama de bloques es que, si se requiere de cambiar los valores de dichas variables se puede hacer directamente desde el diagrama de bloques haciendo doble clic en las celdas de la tabla que aparezca cerca del bloque y escribir el valor deseado, sin tener que modificar el código VHDL del bloque.

La señal *CLK* será la única que tenga asociada un pin físico en el FPGA, por su parte las variables *RESET*, *ENABLE*, y *DUTY* estarán asociadas a otros bloques dentro del mismo diagrama, se pueden considerar como señales “virtuales”.

La señal *ENABLE* es la que habilita el funcionamiento del circuito, por lo tanto, se requiere que esta entrada siempre esté en estado 1 para que el circuito pueda funcionar. Se puede incluir el bloque *lpm_constant*, usado en la sección anterior, para crear una constante de un bit en estado 1, pero también existe el bloque *vcc* dentro de las carpetas de la herramienta *“Block Diagram”* de la Figura 5.22 el cual es equivalente a una señal de alimentación, para ello se accede a la carpeta “*primitives”* y dentro esta la subcarpeta *“other”* la cual contiene el bloque *vcc*, como lo muestra la Figura 5.22.

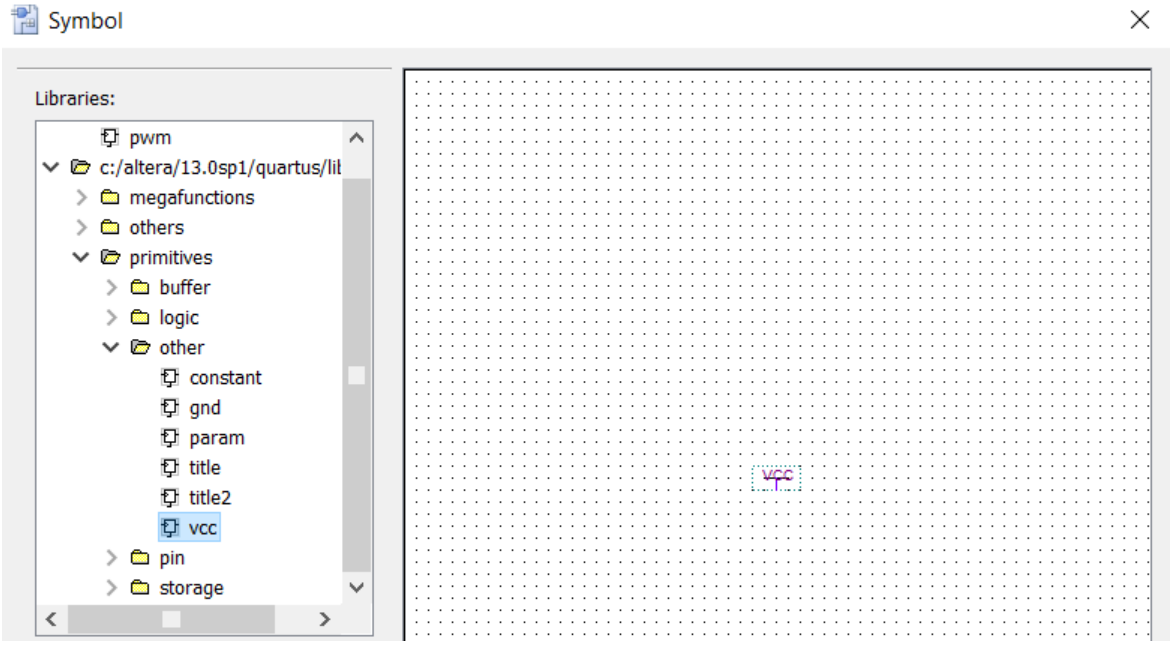


Figura 5.22 Ubicación del bloque vcc.

Después se conecta el bloque *vcc* a la entrada del bloque *enable* siguiendo la conexión que muestra la Figura 5.23.

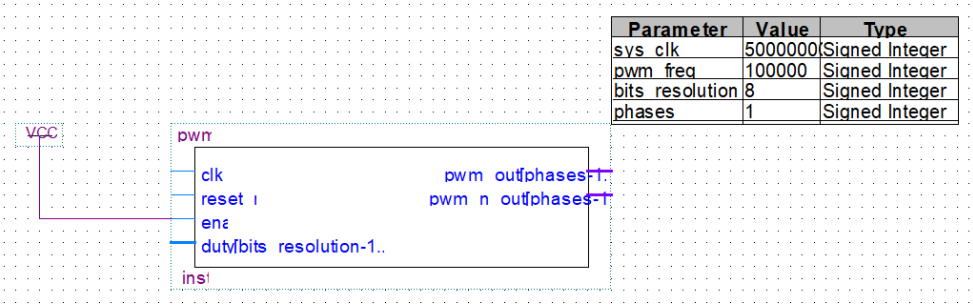


Figura 5.23 Unión entre la fuente vcc y el bloque PWM_GENERATOR

El siguiente paso es generar una constante numérica la cual indicará al generador de PWM el porcentaje del ciclo de trabajo expresado en un numero de 8 bits. Para ello, se requiere de crear el bloque *lpm_constant* que, de la misma manera que el bloque *vcc*, se accede a la carpeta "*megafunctions"* y dentro esta la subcarpeta *"gates"* la cual contiene el bloque *lpm_constant*, como lo Figura 5.24.

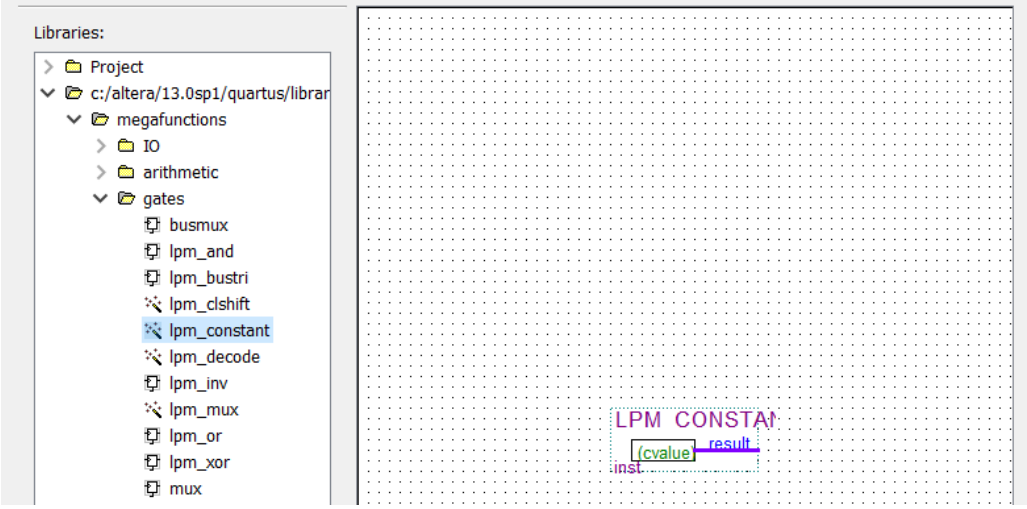


Figura 5.24 Ubicación del bloque lpm_constant

Se abrirá la ventana que muestra la Figura 5.25, que es el programa *"MegaWizard Plug-in Manager*", donde se asigna como nombre *duty_cycle* al bloque y después se hace clic en el botón *"Next >"*.

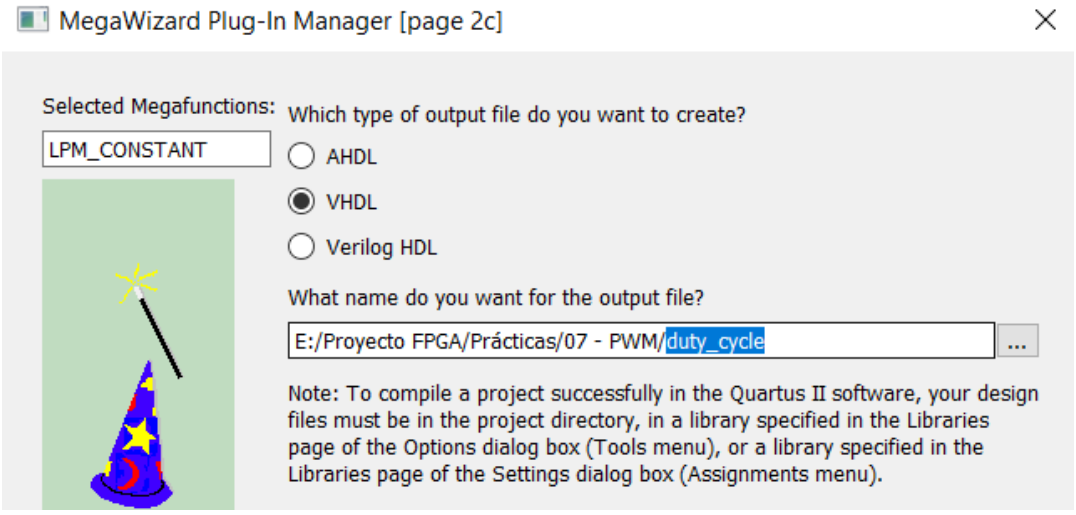


Figura 5.25 Ventana de MegaWizard Plug-In Manager.

Una buena práctica es que todos los bloques generados de esta forma tengan su nombre en minúsculas, mientras que para que se distingan de los bloques generados por código VHDL, sus nombres se escriben en mayúsculas. De esta manera al crecer el número de bloques en un proyecto, es más fácil identificar los archivos principales del proyecto.

Al hacer clic en *Siguiente* el software abrirá una ventana donde se establece el valor de la contante, que será un número de 8 bits y se asigna un valor entre 0 y 255. Para esta primera prueba se fija en un valor de 128, como lo muestra la Figura 5.26.

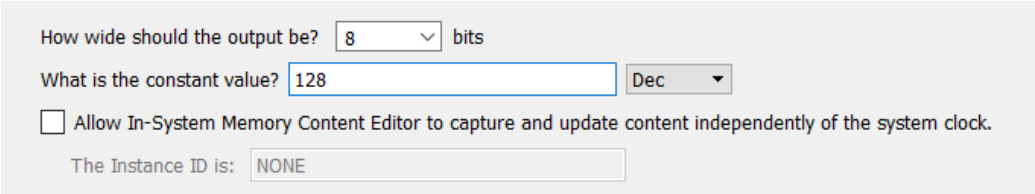


Figura 5.26 Asignación del valor 128 al bloque lpm_constant

Cuando se genere el bloque, se realiza la conexión al bloque del generador uniéndolo como se indica en la Figura 5.27.

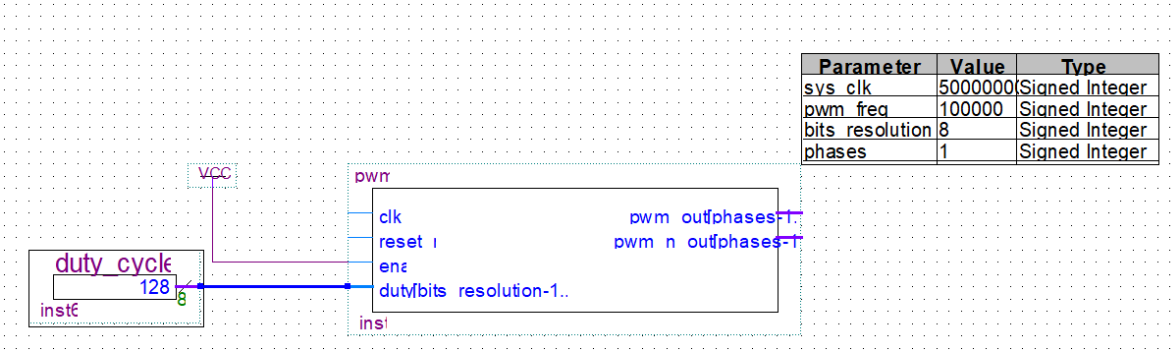


Figura 5.27 Conexión del bloque "lpm_constant" y del bloque principal.

Finalmente se agregan los pines de entrada CLK, RESET, y la señal de salida PWM_OUTPUT, tal como se explica en la sección3.4.3, y usando la herramienta "Pin Planner" se asignan los pines físicos del FPGA de acuerdo con la información de la Tabla 5.6. La salida PWM_n_OUTPUT no tendrá una conexión con un pin del FPGA y, por lo tanto, no tendrá una asignación a un pin físico. El software puede compilar al programa sin

problemas, aunque no exista esa conexión. Finalmente, el diagrama elaborado en Quartus II debe ser similar al que presenta la Figura 5.28.

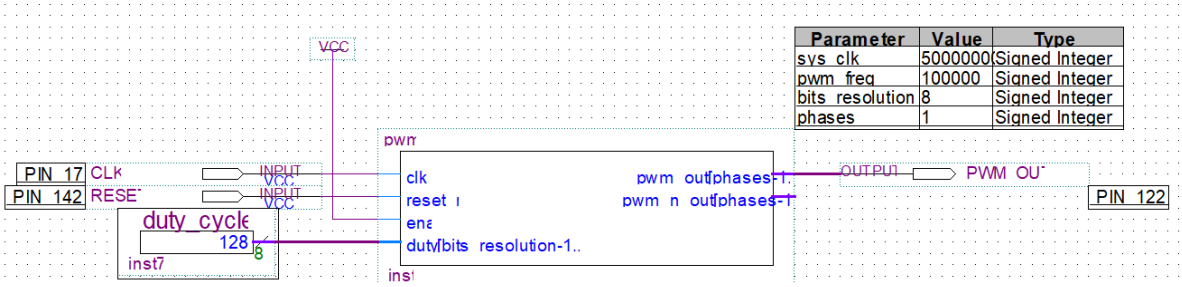


Figura 5.28 Adición de las entradas y salidas del proyecto.

Cuando se cargue el programa al FPGA, el motor debe empezar a girar con una velocidad media, porque el valor se estableció en 128 de acuerdo con la Figura 5.28. Después, si se hacen pruebas con los valores 32, 64, 72, 90, 128, 160 y 225, la velocidad debe modificarse de manera proporcional. De acuerdo con estos datos experimentales, con valores menores a 64, la señal PWM no tiene la suficiente potencia para mover el rotor.

Para la construcción de este circuito, de igual que en la práctica anterior, se usa una alimentación adicional a la que mantiene en funcionamiento el FPGA. En este caso, el valor de esa alimentación se propone que sea de 9 V y, por lo tanto, el voltaje nominal del motor debe ser igual a este valor. También se emplea un circuito de potencia con un puente H para que el FPGA pueda ser capaz de accionar el motor de CD, aunque también puede emplearse un sistema de potencia con transistores bipolares de juntura. Los diagramas de las figuras 5.29 y 5.30 indican las conexiones que se emplean para construir el circuito y la Tabla 5.5 los componentes electrónicos necesarios para su elaboración.

Tabla 5.4 Lista de dispositivos para la práctica 7.

Dispositivo	Valor	Cantidad	Símbolos asociados
Resistencia	1 kΩ	1	R1
Condensador cerámico	100 nF	1	C2
Condensador cerámico	220 nF	1	C1
Push-BUTTON	--	1	KEY1
Puente H	L293D	1	U1
Motor de CD	9 V	1	M1
Batería	9 V	1	vcc

Tabla 5.5 Pines del FPGA asociados a los componentes del generador de señal PWM.

Entradas			**Salidas**		
Variable	**VHDL**	**PIN FPGA**	**Señal**	**VHDL**	**PIN FPGA**
CLK	CLK	17	PWM	PWM_OUTPUT	122
RESET	SWITCH	142	!PWM	PWM_n_OUTPUT	--
ENABLE	ENABLE	vcc	--	--	--
DUTY	pwm_duty	--	--	--	--

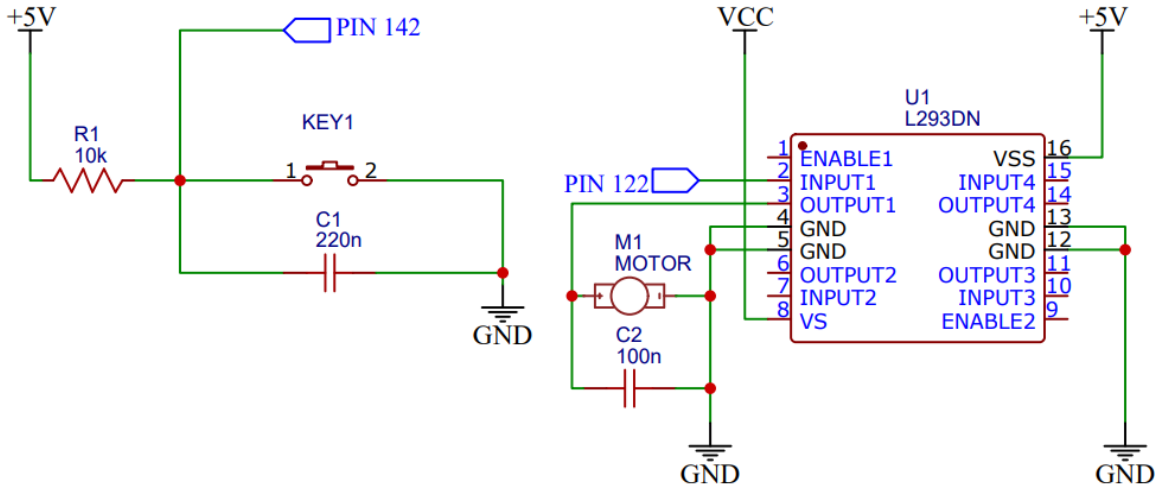


Figura 5.29 Diagrama esquemático para generador de PWM.

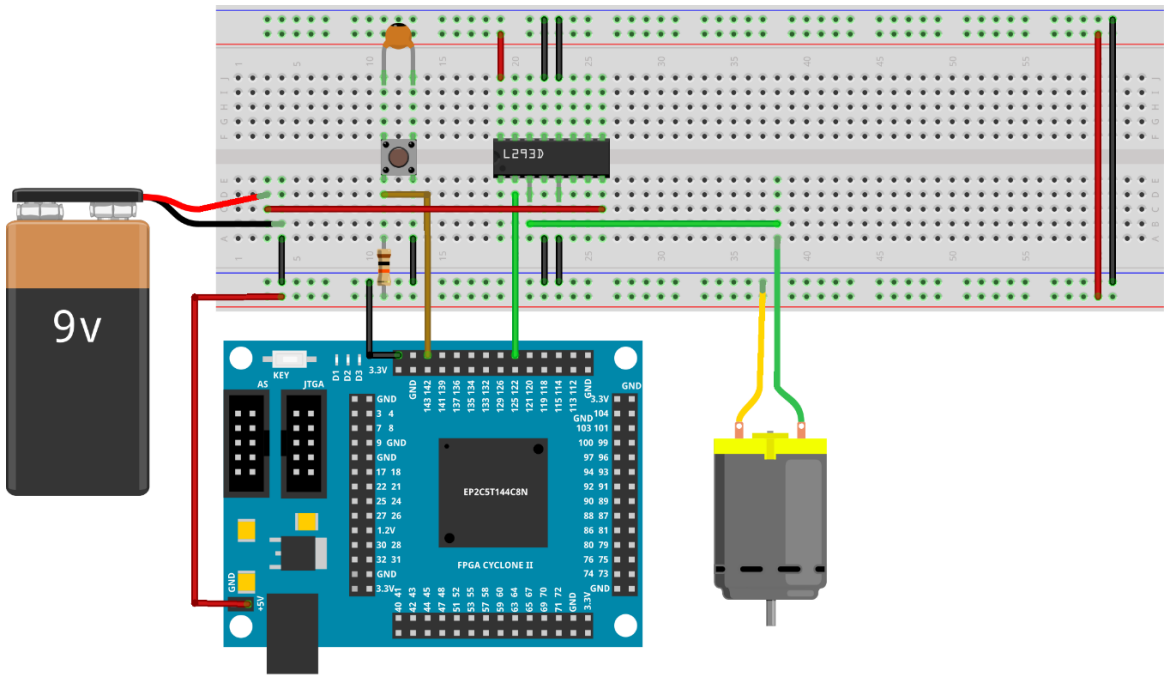


Figura 5.30 Diagrama de conexiones del circuito físico de la práctica 7 implementado en una placa de pruebas.

5.3 EMPLEO DE UNA PANTALLA LCD 16 X 2

En las prácticas anteriores se han diseñado controladores para dos tipos diferentes de motores, mientras que en esta práctica se tiene como objetivo elaborar un controlador para una pantalla de cristal líquido, o LCD por sus siglas en inglés, para imprimir un mensaje. La pantalla que se usará en esta práctica, así como en las secciones siguientes, será de 32 caracteres distribuidos en 2 renglones de 16 caracteres cada uno.

Es importante resaltar que, si se está trabajando con el modelo Cyclone II, solo tiene alimentación de 3.3 V, así que es necesaria la modificación en la tarjeta de desarrollo mencionada en la sección 1.2.1 y después alimentar el FPGA con 5 V. Por otro lado, en la tarjeta Cyclone IV no se requiere de alimentación o modificación adicional, solamente se coloca la pantalla de cristal líquido en el puerto dedicado y ubicar los pines correspondientes con la información que se presenta en la Tabla A.2 del apéndice.

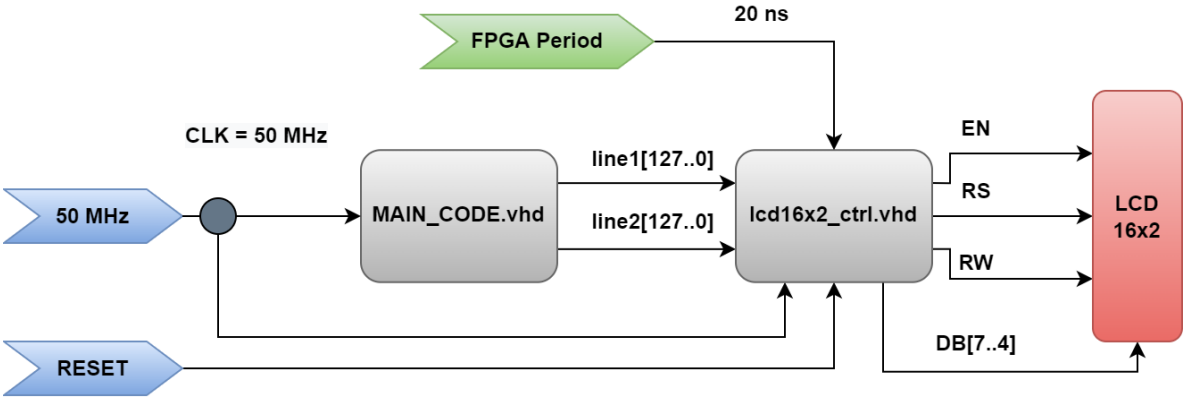


Figura 5.31 Diagrama de bloques para el manejo de un LCD implementado en FPGA.

En esta práctica se empleará un bloque principal, nombrado *MAIN_CODE* donde se escribirán las líneas de código necesarias para imprimir el mensaje en la LCD, mientras que un segundo bloque, nombrado *lcd16x2_ctrl*, que procesará el mensaje escrito en el bloque principal del proyecto y para escribirlo en la pantalla, tal y como lo propone el diagrama de la Figura 5.31.

Para simplificar el código del bloque principal *MAIN_CODE*, se hará uso del archivo de biblioteca *intrument* que se puede descargar de la carpeta virtual del apéndice C y se accede con la dirección *FPGA Handbook/Libraries/* y en la carpeta se encuentra el archivo VHDL del archivo de biblioteca que se empleará en el código del bloque. Este archivo de biblioteca contiene una función nombrada *to_std_logic_vector()*, la cual convierte una cadena de caracteres en un vector de bits. El resultado de esta función se asigna a las salidas de *MAIN_CODE* las cuales, de acuerdo con la Figura 5.31, son *line1* y *line2* las cuales representan los renglones superior e inferior de la pantalla LCD respectivamente.

Para hacer uso del archivo de biblioteca *instrument*, primero se crea el proyecto y el archivo del diagrama de bloques con el nombre *FPGA_LCD*. Después se descarga el archivo *instrument.vhd* dentro de la carpeta virtual del apéndice C con la dirección *FPGA Handbook/ Libraries/* y se incluye el archivo dentro de la carpeta donde se aloja el proyecto, tal como lo muestra la Figura 5.32.

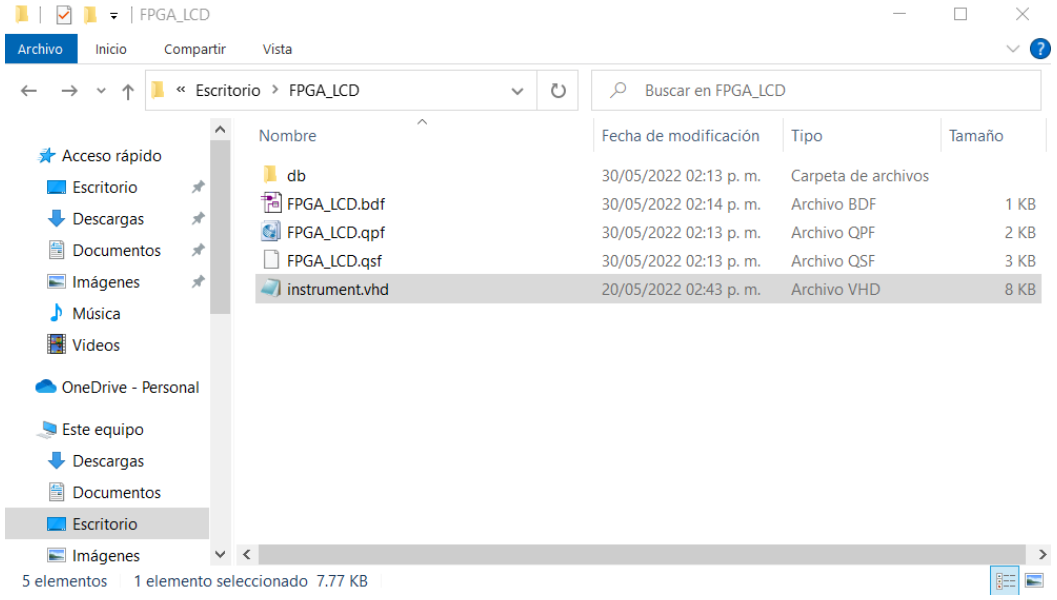


Figura 5.32 Archivo de biblioteca "instrument" incluida en la carpeta del proyecto.

El siguiente paso es incluir el archivo de biblioteca *instrument.*vhd en el código de *MAIN_CODE*, en el navegador del proyecto, se selecciona la pestaña *"Files"*, después se hace clic izquierdo sobre el icono *"Files"* en la parte superior de la lista, nombrado de igual manera como *"Files"*, y se selecciona *"Add/Remove files in proyect"* con lo que se abrirá la ventana mostrada en la Figura 5.33.

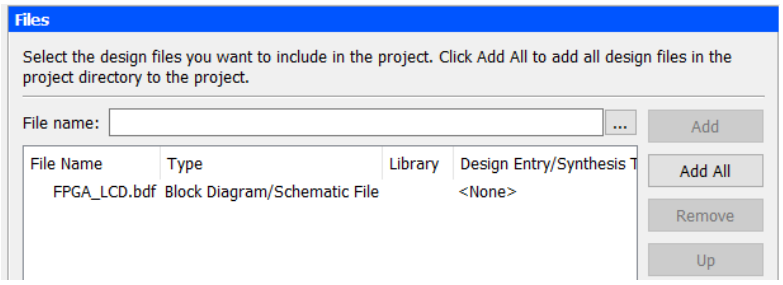


Figura 5.33 Ventana "Settings"

En la sección *"File name"* se selecciona el botón *"..."* y se abre una ventana del navegador de archivos de Windows, después el archivo *instrument.vhd* se selecciona y se

hace clic en el botón *"Abrir"*. Regresando en la ventana de la Figura 5.33, se hace clic en el botón *"Add"* y el archivo de biblioteca aparecerá en la lista de la parte central, y finalmente se selecciona el botón *"Ok"*. La Figura 5.34 muestra el navegador del proyecto con el archivo *instrument.vhd* dentro de la lista junto al archivo del diagrama de bloques

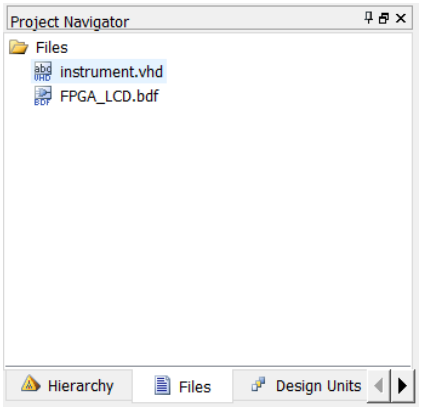


Figura 5.34 archivo de biblioteca "instrument.vhd" dentro del navegador del proyecto.

Después, se debe generar el archivo *MAIN_CODE.vhd* con el código B.6 del apéndice y se tendrá que generar el bloque correspondiente usando la metodología de la sección 3.4.2. El código 5.3 muestra las primeras seis líneas del código B.6 donde, específicamente en la línea 1 y 5, se declara un segundo archivo de biblioteca el cual tiene el nombre de *work*. Este archivo permite usar las funciones almacenadas dentro de un archivo VHDL programado como archivo de biblioteca y que se encuentra dentro de la carpeta del proyecto. En este caso, *intrument.vhd* tiene una serie de funciones que permitan facilitar el manejo de la pantalla LCD.

Código 5.3 Declaración de la biblioteca "instrument" en el código de MAIN_CODE.vhd

```
1       library ieee;
2       use ieee.std_logic_1164.all;
3       use ieee.numeric_std.all;
4
5       library work;
6       use work.instrument.all;
```

En las líneas que se muestran en el código 5.4, las salidas declaradas en el código B.6 como *LINE1_lcd* y *LINE2_lcd* son de tipo *STD_LOGIC_VECTOR*, las cuales que tienen una extensión de 128 bits. El controlador de la pantalla que se emplea en este circuito, citado en apartado C.4, transforma los caracteres de las dos líneas de la pantalla en números binarios de 8 bits los cuales almacenan el valor numérico que corresponden al código ASCII de cada carácter. La función *to_std_logic_vector()* realiza la conversión de una cadena de 16 caracteres, donde cada línea es un *STD_LOGIC_VECTOR* con la longitud mencionada anteriormente. En las líneas 25 y 26 se emplea dicha función como lo muestra el código 5.4.

Código 5.4 Código 5.4 Código básico para escribir en una LCD de 16x2.

```
25        LINE1_lcd <= to_std_logic_vector("HELLOW WORLD :) ");
26        LINE2_lcd <= to_std_logic_vector("A FPGA is used  ");
```

Recordando lo visto en la práctica 6 de la sección 5.2, el bloque *lcd16x2_ctrl* y su respectivo código VHDL también se deben descargar de la carpeta virtual del apéndice D donde se accede a través de la dirección *Handbook/libraries/LCD 16x2* y se incluye la carpeta, con los archivos de su contenido, en el proyecto como lo muestra la Figura 5.35.

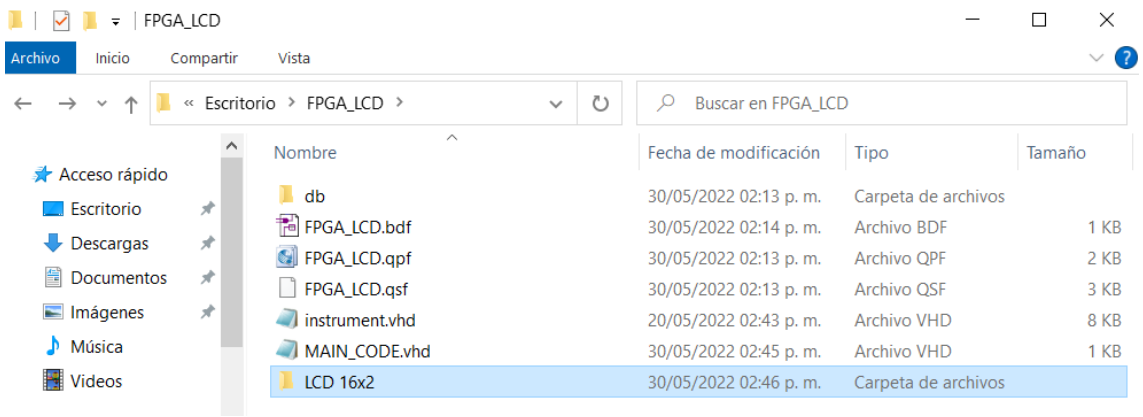


Figura 5.35 Carpeta "LCD 16x2" insertada en la carpeta del proyecto.

De la misma forma como se incluyó el archivo de biblioteca *instrument*, en el navegador del proyecto se accede a la pestaña *"Files"*, luego se hace clic izquierdo sobre el icono superior de la lista y se selecciona la opción *"Add/Remove files in proyect"* y se abrirá la ventana de la Figura 5.36 que lista los archivos del proyecto.

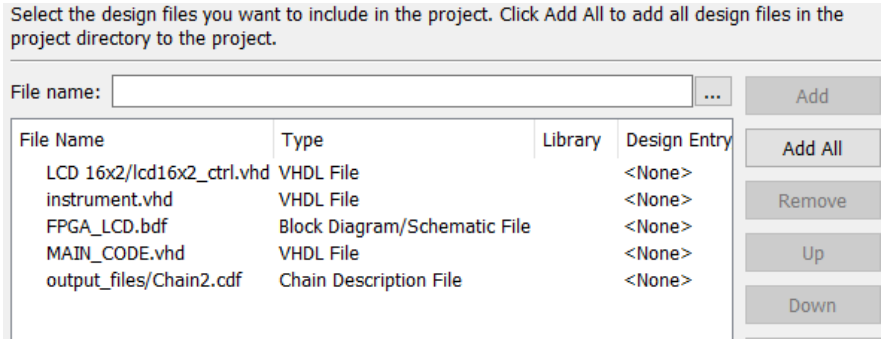


Figura 5.36 Sección "Settings" de la ventana "Files".

En la sección *"File name"* se selecciona el botón *"..."* y se abre una ventana del navegador de archivos de Windows, se entra a la carpeta *"LCD 16x2"* descargada anteriormente y se abre el archivo *lcs16x2_ctrl.vhd* como lo indica la Figura 5.37.

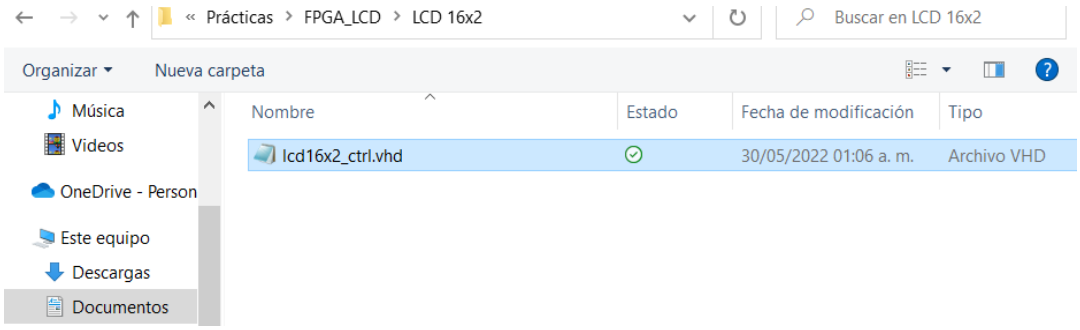


Figura 5.37 Selección del archivo "lcd16x2_ctrl.vhd"

Después de hacer clic en *"Abrir"*, se cierra el navegador de Windows y se hace clic en el botón *"Add"* y el archivo se incluirá en la lista que muestra la Figura 5.38.

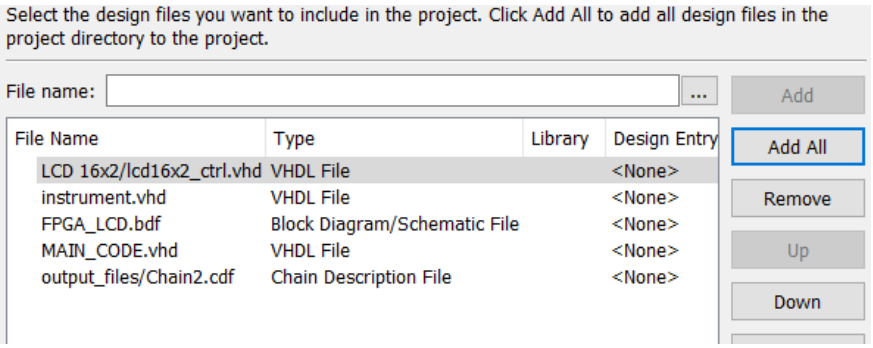


Figura 5.38 Archivo "lcd16x2_ctrl.vhd" añadido a los archivos del proyecto.

Posteriormente, se hace clic en el botón *"Ok"* y el archivo del controlador del LCD aparecerá en la jerarquía del navegador del proyecto. Dentro del diagrama de bloques del proyecto, se entra a la herramienta *"Sybol Tool"* y se incluyen en el diagrama los bloques *MAIN_CODE* y *lcd16x2_ctrl*.

También se incluyen las entradas y salidas siguiendo el método de la sección 3.4.3, y los bloques se conectan de acuerdo con lo que establece la Figura 5.39. Finalmente se asignan los pines físicos a las entradas y salidas de acuerdo con la Tabla 5.6 mostrada más adelante.

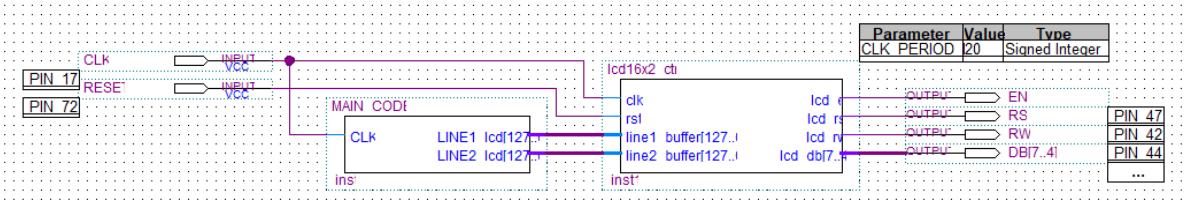


Figura 5.39 Diagrama de bloques en Quartus II para escribir en un LCD.

Finalmente, se realizan las conexiones de acuerdo con la información de las figuras 5.40 y 5.41, mientras que la lista de componentes para construir el circuito se encuentra en la Tabla 5.7. Cuando se haya construido el circuito en la placa de pruebas, se deberá cargar el código al FPGA y visualizará los mensajes que muestra la Figura 5.40 que corresponden a las cadenas dentro de la función *to_std_logic_vector()* en las líneas 25 y 26 del código B.6.

El mensaje se puede modificar para continuar con las pruebas de funcionamiento, se puede aplicar la concatenación de cadenas, mostrada en la sección 2.4.8 y la conversión de

código ASCII a carácter mencionada en la sección 2.4.7, siempre y cuando el tamaño de la cadena que se escriba siempre contengan los 16 caracteres de cada línea, de lo contrario habrá errores al momento de compilar el proyecto.

Figura 5.40 Imprimiendo el mensaje programado.

Tabla 5.6 Lista de dispositivos para la práctica 8.

Dispositivo	Valor	Cantidad	Símbolos asociados
Resistencia	10 kΩ	1	R1
Resistencia	220 Ω	1	R2
Trimpod	1 kΩ	1	RP1
LCD	16x2	1	LCD1
Botón pulsador	--	1	KEY1
Condensador Cerámico	220 nF	1	C1

Tabla 5.7 Pines del FPGA asociados a los componentes del controlador de pantalla LCD.

Entradas			**Salidas**		
Variable	VHDL	PIN FPGA	LCD	VHDL	PIN FPGA
CLK	CLK	17	EN	EN	47
RESET	RESET	72	RS	RS	42
			RW	RW	44
			DB[7]	DB[7]	71
			DB[6]	DB[6]	69
			DB[5]	DB[5]	65
			DB[4]	DB[4]	63

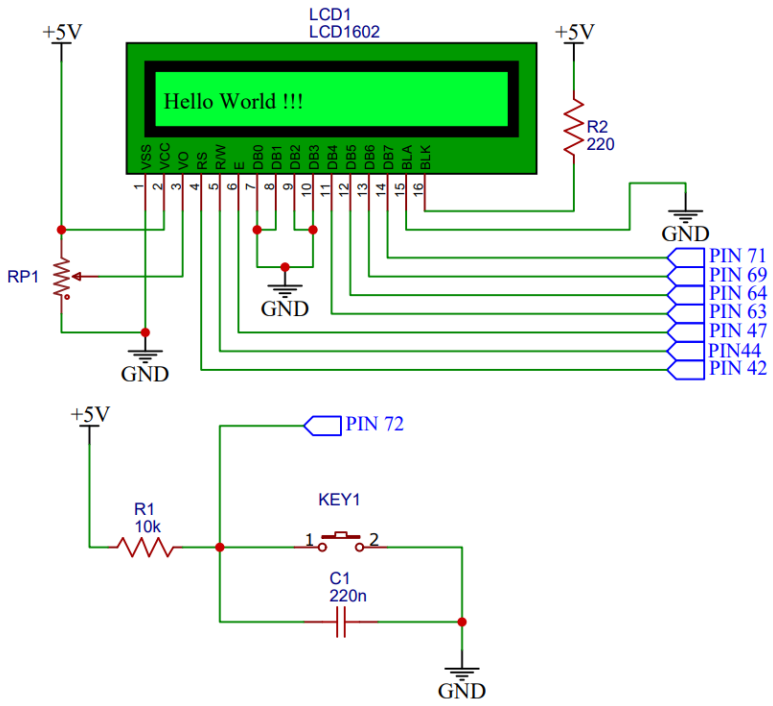


Figura 5.41 Diagrama esquemático para controlador de LCD.

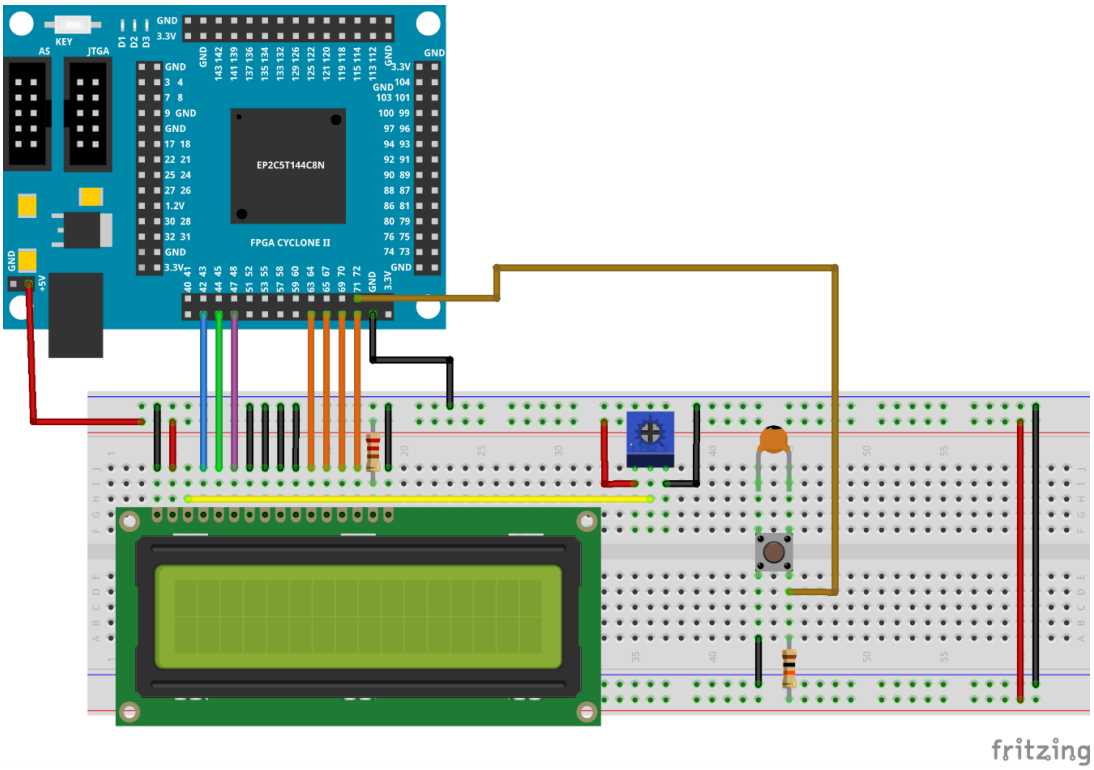


Figura 5.42 Diagrama de conexiones del circuito físico de la práctica 8 implementado en una placa de pruebas.

5.4 CONVERTIDOR ANALÓGICO A DIGITAL

La práctica de esta sección tiene el propósito de programar FPGA para leer una señal de voltaje y desplegar el valor de la medición en una pantalla de cristal líquido. El dispositivo periférico que se implementará para obtener la medición es convertidor analógico a digital, que es un dispositivo que permite la digitalización de señales analógicas. Esta digitalización permite procesar y de obtener información de variables físicas que pueda utilizarse en otros procesos dentro del sistema.

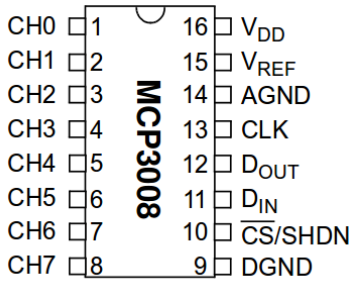


Figura 5.43 Configuración de pines del circuito MCP3008.

El circuito MCP3008, cuya función está mostrada en la Figura 5.43, es un convertidor analógico digital de 8 canales analógicos. Tiene una resolución de 10 bits, por ejemplo, si el voltaje de referencia es de 5 V, cada bit obtenido de la señal representa 4.88 mV. Por otro lado, el formato de comunicación de este dispositivo es por comunicación de interfaz de periférico serial, o SPI por sus siglas en inglés.

El circuito integrado contiene 16 pines, desde el pin 1 hasta el pin 8, corresponden a los 8 canales de lectura analógica, desde el pin 10 hasta el 13, corresponden a la interfaz de comunicación SPI y los pines restantes corresponden a la alimentación del dispositivo. En la Tabla 5.8 se tiene la información explicada a detalle de la función de cada pin del circuito.

Tabla 5.8 Simbología y descripción de pines del MCP3008 mostrado en la Figura 5.43.

PIN	Nombre	Descripción	PIN	Nombre	Descripción
1	CH0	Entrada Analógica	**9**	DGND	Voltaje de tierra del circuito
2	CH1	Entrada Analógica	**10**	$\overline{CS}$/SHDN	Habilita el funcionamiento del circuito
3	CH2	Entrada Analógica	**11**	D_{IN}	Lectura de la información de maestro a esclavo
4	CH3	Entrada Analógica	**12**	D_{OUT}	Escritura de la información de esclavo a maestro
5	CH4	Entrada Analógica	**13**	CLK	Señal de reloj
6	CH5	Entrada Analógica	**14**	AGND	Voltaje de tierra de la señal analógica
7	CH6	Entrada Analógica	**15**	V_{REF}	Voltaje de referencia de la señal analógica
8	CH7	Entrada Analógica	**16**	V_{DD}	Voltaje de alimentación del circuito

De la misma forma que en la práctica anterior, el circuito tendrá un bloque nombrado *MAIN_CODE* en el cual se escribe el código principal y el mensaje que se imprimirá en la pantalla LCD. Las salidas de este bloque se reciben en que el controlador que interpreta la información y se escribe en la pantalla. De manera paralela, se emplea un bloque controlador como se implementó en la práctica anterior, para el circuito MCP3008 que se encarga de convertir la señal de voltaje analógico a una señal digital.

El diagrama de la Figura 5.44 muestra la estructura del circuito, donde se tiene cuatro entradas (*RESET*, *CHANNEL*, *CLK*, y *MOSI*), cuatro salidas asociadas a la pantalla LCD (*EN*, *RS*, *RW*, y *DB*), tres salidas asociadas al circuito MCP3008 (*ADC_CLK*, *MISO*, *CS*) y una variable paramétrica *CLK_PERIOD_NS*.

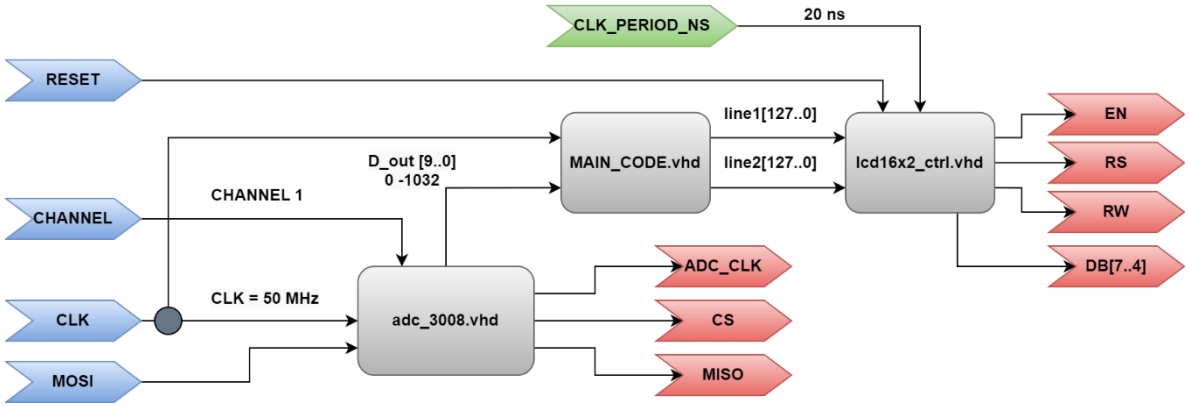


Figura 5.44 Diagrama de bloques del controlador de ADC.

El primer paso será crear el proyecto y el archivo del diagrama con el nombre *FPGA_ADC* e incluir el archivo de biblioteca *intrument.vhd* siguiendo el mismo método que en la sección anterior. Después, en la carpeta virtual de la sección D del apéndice se accede a la dirección *FPGA Handbook/Libraries/* y se descarga la carpeta *"MCP 3008 driver"*, que contiene el bloque y el código de la sección C.3, y se guarda dentro de la carpeta que contiene el proyecto. En la misma dirección de la carpeta virtual, se descarga la carpeta *LCD 16x2*, que contiene el bloque y el código de la sección C.4, y también se incluye en la carpeta del proyecto.

La Figura 5.45 muestra el navegador de Windows donde se encuentra la carpeta que aloja el proyecto con las carpetas descargadas, que corresponden a los controladores del convertidor analógico a digital y la pantalla de cristal líquido, y también el archivo de biblioteca *instrument.vhd*

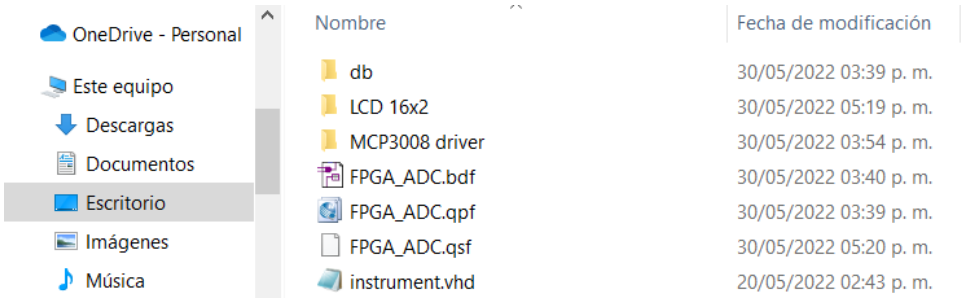


Figura 5.45 archivos y carpetas del proyecto "FPGA ADC"

Para añadir los archivos VHDL al proyecto, se repite el proceso de las practicas anteriores donde se hace clic en la pestaña *"Files",* situado en la parte inferior del navegador del proyecto de Quartus II, y se abre la ventana que administra los archivos del proyecto. Se selecciona el botón *"..."*, situado en la parte superior de la lista y cuando se abra el navegador de Windows se seleccionan los archivos *.vhd* que estén dentro de la carpeta del proyecto (que incluye los controladores del convertidor analógico a digital y la pantalla LCD) y se selecciona el botón *"Abrir"* y después se hace clic en el botón *"Add"* , donde los archivos aparecerán en la lista inferior de la ventana, y finalmente se hace clic en *"Ok"*. El resultado de este proceso se debe mostrar como lo indica la Figura 5.47.

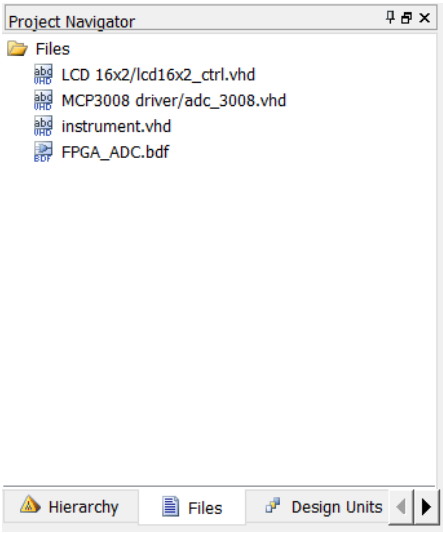


Figura 5.46 Navegador de proyectos de "FPGA ADC" con los archivos y carpetas añadidos.

El siguiente paso es crear el archivo VHDL que se nombra como *MAIN_CODE* y el código que se empleará será es el de la sección B.8 del apéndice. A simple vista este código es muy similar al código B.7 usado en la práctica de la sección anterior, pero existe una diferencia notoria porque se añade una entrada en la línea 13 llamada *ADC,* que corresponde a un número binario de 10 bits que representa la lectura del convertidor analógico a digital.

El siguiente cambio es en la línea 23, donde se declara una señal interna llamada *ADC_VAL* declarada como una variable de tipo numérica donde se restringe en un rango de valores desde 0 hasta 1023.

Código 5.5 Sección del código B.8

```
27   ADC_VAL <= to_integer(unsigned(ADC));
28   LINE1_lcd <=
29   to_std_logic_vector("BIN : " & to_string(ADC));
30   LINE2_lcd <=
31   to_std_logic_vector("NUMERIC :  " & to_string(ADC_VAL,5));
```

En la línea 27 del código 5.5, la entrada *ADC* es declarada como un *STD_LOGIC_VECTOR* de 10 bits, pero se hace una conversión de tipo de dato a *UNSIGNED* y se introduce como argumento de la función *to_integer(),* función que pertenece al paquete *numeric* del archivo de biblioteca *ieee*, que realiza la conversión del número binario sin signo a un valor tipo entero y el resultado se asigna a la señal *ADC_VAL*.

En las líneas 28 y 29, se asigna a la salida *LINE1_lcd*, que es el vector de bits que guarda la cadena de 16 caracteres de la primera línea de la pantalla, el resultado de la función *to_std_logic_vector()*, y dentro de la función se usa otra función llamada *to_string()* la cual convierte el vector de 10 bits de la entrada *ADC* a una cadena de 10 caracteres, y de esta manera en la primera línea de la pantalla se despliega el valor binario de la lectura del convertidor analógico a digital.

Por otro lado, en las líneas 30 y 31 también se usa la función *to string()*, dentro de la función *to_std_logic_vector()*, con la diferencia de que se usa otra definición. En este caso, la función convierte el valor de *ADC_VAL*, que es un numero entero, a una cadena de 5 caracteres donde el primer carácter corresponde al signo del número y los otros cuatro

despliegan el valor del ADC que se encuentra entre 0 y 1023. De esta manera, cuando se cargue el código al FPGA, se podrá ver en la segunda línea de la pantalla el valor de la medición del convertidor en número decimal.

Cuando se haya terminado de escribir el código en el bloque *MAIN_CODE*, se crea el bloque asociado a este código, que se explica en la sección 3.4.2, y se incluyen todos los bloques en el diagrama de bloques en Quartus II, tal como lo indica la Figura 5.47.

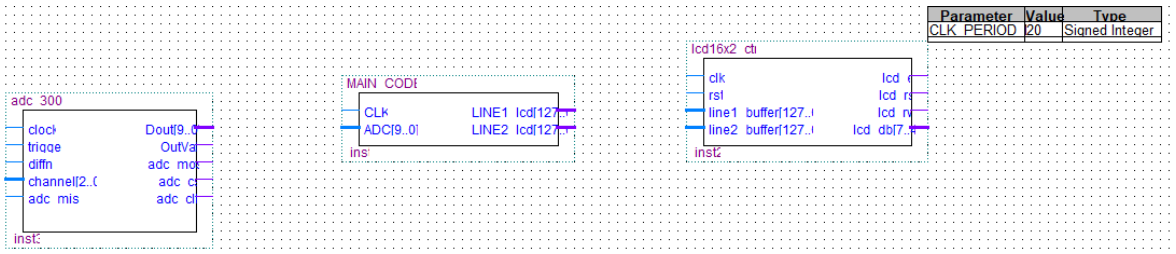


Figura 5.47 Bloque principal "MAIN_CODE" y bloques controladores del ADC y la pantalla LCD.

Antes de realizar las conexiones, se incluye un bloque *vcc* y un *lpm_constant*, tal como se realizó en la práctica de la sección anterior. Para ello, se usa la opción "*Symbol Tool"*, se busca en la carpeta *Primitives* y dentro de la subcarpeta *other* se selecciona el bloque *vcc*. Por otro lado, se ubica la carpeta *megafunctions* y dentro de la subcarpeta *gates* se selecciona el bloque *lpm_constant*.

Para el bloque *lpm_constant*, se nombra como *CHANNEL* y después se asigna el valor de 0 para indicar al controlador del circuito MCP3008 que la señal de voltaje del canal 1 debe ser leída, y el valor de *CHANNEL* será representado por un número binario de 3 bits, tal como lo indica la Figura 5.48.

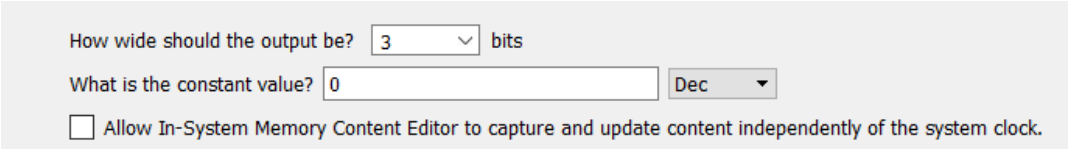


Figura 5.48 Configuración del valor constante "channel" para leer el canal 1 del convertidor analógico a digital.

Se conecta el bloque *CHANNEL* en la entrada *channel[2..0]* del bloque *mcp3008*, mientras que el bloque *vcc* se conecta en la entrada *diffn*. Se incluyen también las entradas de un solo bit *CLK*, *RESET* y *MOSI*, y las salidas para conectar el FPGA al circuito MCP 3008, las cuales son *CS*, *ADC*, *CLK* y *MISO*.

Por otro lado, para conectar el FPGA a la pantalla LCD, las salidas de un solo bit son *RS*, *EN* y *RW*, y la salida de 4 bits *DB[7..4]*.

Adicionalmente se realiza una conexión de realimentación en el bloque mcp3008 donde se une la salida *adc_clk* con la entrada *trigg*, de esta manera el convertidor analógico a digital puedan realizar mediciones todo el tiempo que esté el FPGA en funcionamiento. La salida *Dout[9..0]* del bloque *mcp3008.vhd* se conecta a la entrada *ADC* del bloque *MAIN_CODE* mientras que las salidas *LINE1_lcd* y *LINE2_lcd* se conectan a las entradas del bloque *lcd16x2_ctrl.vhd*. El diagrama de bloques resultante se muestra en la Figura 5.49.

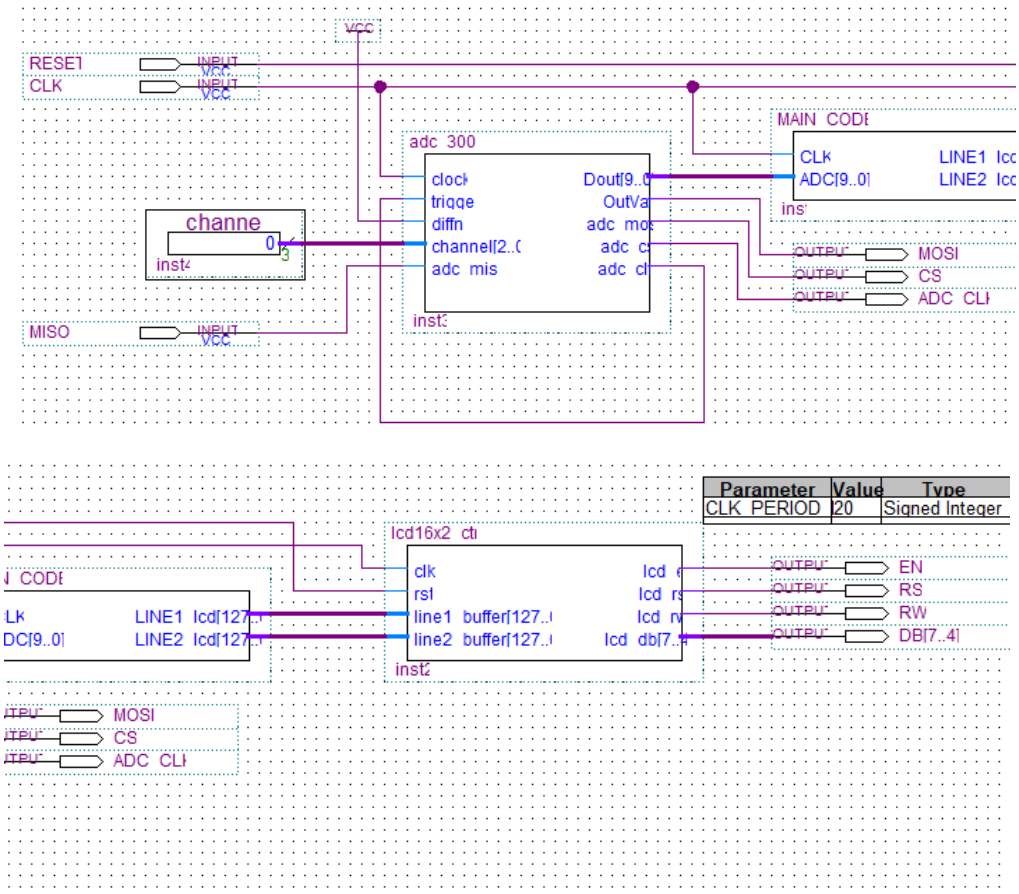


Figura 5.49 Diagrama de bloques para programa con MCP3008 y pantalla LCD.

Finalmente, las entradas y las salidas del diagrama de bloques se asocian a los componentes físicos del circuito, como lo indica la Tabla 5.10. El circuito se construye con los componentes listados en la Tabla 5.9 y las conexiones necesarias en la placa de pruebas se muestran en las figuras 5.51 y 5.52.

La pantalla LCD debe mostrar los mensajes programados en el código B.8 y en la parte derecha de ambos renglones se debe visualizar el valor del convertidor analógico a digital en número binario (arriba), y en número decimal (abajo). Con el potenciómetro incluido en las conexiones del circuito, al momento de girar la perilla se debe modificar el valor de ambos números de forma proporcional, la Figura 5.50 muestra las mediciones cuando el potenciómetro está en un valor medio.

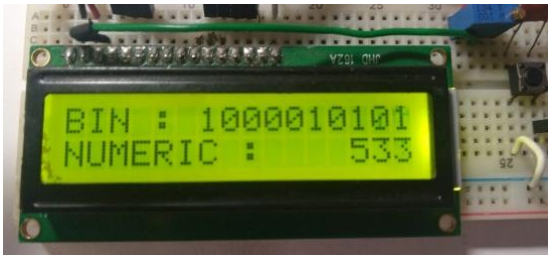


Figura 5.50 Pantalla LCD desplegando el valor del ADC en binario (arriba) y en decimal (abajo).

Tabla 5.9 Lista de dispositivos para la práctica 8.

Dispositivo	Valor	Cantidad	Símbolos asociados
Resistencia	10 kΩ	1	R1
Resistencia	220 Ω	1	R2
Condensador cerámico	220 nF	1	C1
Potenciómetro	10 kΩ	1	RP1
Trimpod	1 kΩ	1	RP2
Convertidor analógico a digital	MCP3008	1	ADC1

Tabla 5.10 Pines del FPGA asociados al circuito con convertidor analógico a digital y pantalla LCD.

Entradas			Salidas		
Variable	**VHDL**	**PIN FPGA**	**Señal**	**VHDL**	**PIN FPGA**
Reloj FPGA	CLK	17	CS	CS	86
MISO	MISO	75	Reloj ADC	ADC_CLK	74
RESET	RESET	72	MOSI	MOSI	79
			EN	EN	47
			RS	RS	42
			RW	RW	44
			DB[7]	DB[7]	71
			DB[6]	DB[6]	69
			DB[5]	DB[5]	65
			DB[4]	DB[4]	63

.

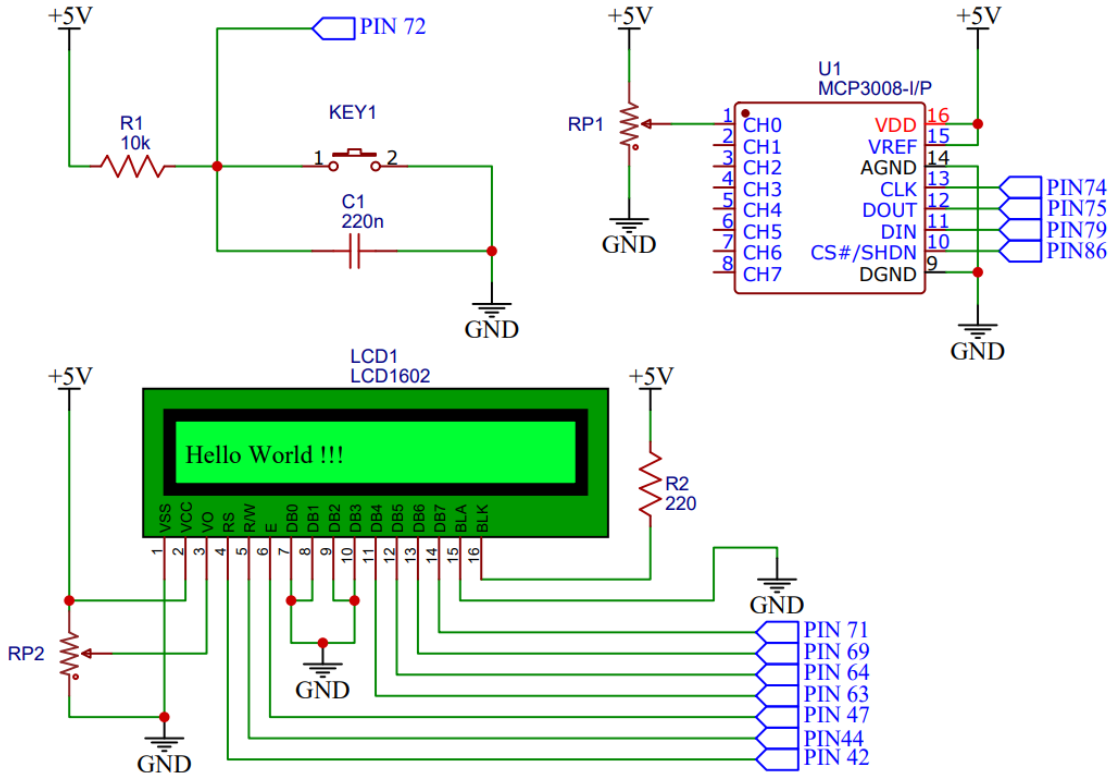


Figura 5.51 Diagrama esquemático para circuito con convertidor analógico a digital y pantalla LCD.

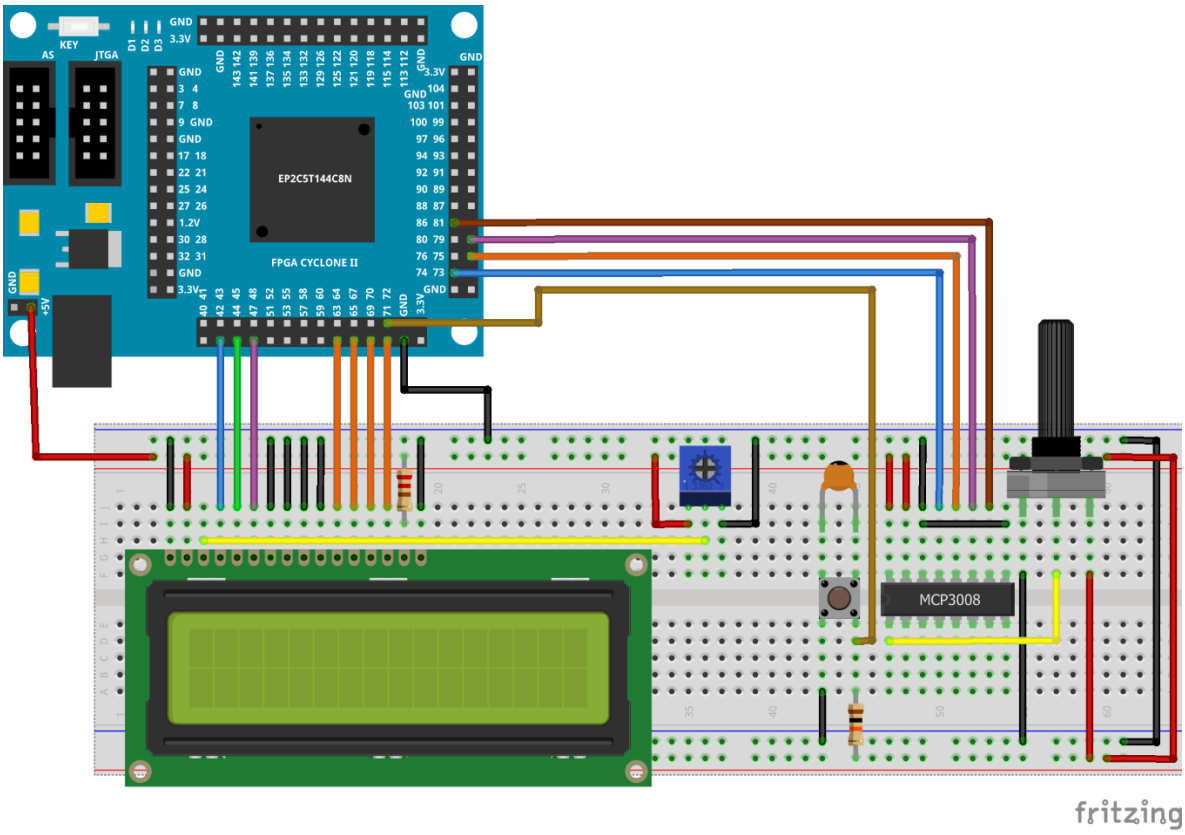


Figura 5.52 Diagrama de conexiones del circuito físico de la práctica 8 implementado en una placa de pruebas.

5.5 MEDICIÓN DE VOLTAJE

En la práctica anterior se obtuvieron mediciones de un convertidor analógico a digital y se desplegó tanto el valor en binario de 10 bits, como el valor en decimal en el intervalo desde 0 hasta 1023. Los datos desplegados representan a la magnitud física, pero para obtener el valor de voltaje se debería realizar el cociente entre el número obtenido del convertidor y el voltaje de referencia, que en este caso es de 5 V.

El propósito de un convertidor analógico a digital es de leer señales de voltaje que sean obtenidas de algún transductor que mida una magnitud física. En esta práctica se harán las modificaciones adecuadas para leer la señal de voltaje obtenida del circuito, donde éste

estará configurado de tal manera que se puedan obtener lecturas de voltaje desde 0 hasta 5 V con una resolución de 10 bits que aporta el convertidor.

Tabla 5.11 Valores de voltaje expresados en un número de 10 bit.

Valor en bits	Valor en volts
1	0.004883
32	0.1563
128	0.6250
300	1.4648
512	2.5000
850	4.1504
990	4.8340
1023	4.9951 ≈ 5

En el caso particular del circuito MCP3008, cada bit del convertidor equivale aproximadamente a 4.8 mV y en la Tabla 5.11 se muestra en la columna izquierda el valor en número decimal de la lectura del convertidor analógico a digital, mientras que en la columna derecha se muestra el valor de voltaje que representa cada número.

La estructura del circuito que se programa en el FPGA en esta práctica se utiliza la misma que fue implementada en la práctica anterior, la cual está mostrada en la Figura 5.44. Los bloques controladores *mcp_3008* y *lcd_ctrl*, obtenidos de la carpeta virtual citada en la sección D del apéndice, volverán a ser usados en la programación del diagrama de bloques del proyecto en Quartus II

Para elaborar el circuito, se crea un proyecto y el diagrama de bloques con el nombre VOLTMETER, se usará la misma metodología de la práctica de la sección anterior porque se construirá el mismo circuito. Se incluye el archivo de biblioteca *instrument.vhd*, que se descarga de la carpeta virtual de la sección C usando la dirección *Handbook/Libraries/* y se guarda en la carpeta contenedora del proyecto. De la misma manera, también se descargan y guardan los bloques *mcp_3008* y *lcd_ctrl*, que se citan en los apartados C.3 y C.4 respectivamente.

Usando la estructura de la Figura 5.44 de la sección anterior, se repite el proceso se construcción del diagrama de bloques para que el resultado final sea el que se muestra en la Figura 5.49.

La asignación de los pines del FPGA a las entradas y salidas del circuito, es exactamente igual a la que se muestra en la Tabla 5.10 debido a que no hay modificaciones en la implementación física del circuito, que es prácticamente igual a la práctica anterior.

A pesar de que la estructura del diagrama de bloques y el circuito físico sean exactamente iguales, el código implementado en el bloque *MAIN_CODE* tendrá una diferencia sustancial, porque en lugar desplegar el valor en número binario del convertidor analógico a digital, ahora será en términos del voltaje leído por el circuito en el canal 1 en donde se conecta el potenciómetro. El código B.9 será implementado en el bloque *MAIN_CODE* y se podrá observar que la diferencia se encuentra desde la línea 23 hasta la 26, así como desde la línea 30 hasta la 34.

Código 5.6 Sección del código B.9 donde se declaran las señales internas.

```
23    constant max_volt : integer := 5;
24    constant resolution : integer := 10**4;
25    signal ADC_VAL: integer range 0 to 1023;
26    signal volts: integer range 0 to max_volt*resolution;
```

El código 5.6 muestra que se declara una variable, nombrada *max_volt*, de tipo constante la cual guarda un número entero con valor de 5, el cual representa el valor máximo de referencia del convertidor analógico a digital. La variable *resolution* representa el numero 10 elevado al número de dígitos de precisión que se requiere, es decir, si son 4 dígitos de precisión, el valor de esta variable es 10^4 que es lo mismo que 10,000. En la variable *ADC_VAL* se guarda el valor en número decimal obtenido del convertidor en un rango desde 0 hasta 1023. Por último, la variable *volts* guarda los valores obtenidos de la señal *ADC_VAL* a través de un proceso de mapeo y se restringe sus valores en un rango desde 0 hasta el valor de *max_volt* multiplicado por el valor de *max_resolution*.

Código 5.7 Sección del código B.9 donde se procesa la señal de voltaje obtenida.

```
30   ADC_VAL <= to_integer(unsigned(ADC));
31   volts <= mapping(ADC_VAL,0,1023,0,max_volt*resolution);
32   LINE1_lcd <= to_std_logic_vector("Voltmeter  FPGA ");
33   LINE2_lcd <= to_std_logic_vector("Voltage :" &
34   to_string(volts,7,4));
```

En el código 5.7 muestra el valor de la entrada *ADC* que se convierte de un nímero binario de 10 bits a un número decimal y el valor obtenido se guarda en *ADC_VAL*. En la línea 31, se usa la función *mapping()* proveniente del archivo de biblioteca *instrument.vhd*, donde se mapea el valor de *ADC_VAL* en un rango desde 0 hasta 1023 en un segundo rango proporcionad desde 0 hasta el producto del valor de *max_volt* por *resolution*, en este código los valores de ambas variables son 5 y 10000 respectivamente y el rango sería desde 0 hasta 50000. Se entiende que el proceso de mapeo debería ser entre 0 y 5, pasando por los números con punto flotante, porque los números flotantes (vistos en la sección 2.4.6) son sintetizables en VHDL, pero Quartus II no puede aplicar los números flotantes en el FPGA. Esa la razón por la que el rango de medición es desde 0 hasta 5000, porque se busca una resolución de 0.0001, para lograrlo se usa el recíproco de ese número, el cual es 10000, y al multiplicarlo por el valor máximo de voltaje el resultado es 50000. De esta manera, se evitará el uso de números flotantes y solamente se trabaja con números enteros para medir la señal de voltaje. En la Tabla 5.12 se muestran los valores de algunas mediciones.

Tabla 5.12 Valor de voltaje dentro del código del bloque "MAIN_CODE".

Valor en bits	Valor en volts	Valor en el código
1	0.004883	49
32	0.1563	1563
128	0.6250	6250
300	1.4648	14648
512	2.5000	25000
850	4.1504	41504
990	4.8340	48340
1023	4.9951 ≈ 5	49951 ≈ 50000

La función *mapping()* calcula para un valor *"x"*, que se sitúa en un rango *x1 – x2*, su valor correspondiente *"y"* proporcional en un rango *y1 – y2* y aplicando la ecuación general de una recta. En la tercera columna de la Tabla 5.12 muestra el resultado del proceso de mapeo de los valores de la primera columna.

En la línea 34 se usa la función *to_string()*, con una definición diferente a la usada en el código B.8 de la practica anterior, debido a que cuenta con tres argumentos (el número entero, la longitud de la cadena y la posición del punto decimal) donde el primero es el valor obtenido de la variable *volts* obtenido de la función *mapping()*, en el segundo se indica que la cadena tendrá una longitud de 7 caracteres (uno para el signo del número, uno para el punto flotante y 5 para los dígitos) y en el tercer argumento se indica la posición del punto decimal en el número de 4 dígitos contando de derecha a izquierda (ej. 49951 se escribe como 4.9951).

Es importante mencionar que la función *mapping()* tiene una excepción, la cual restringe que el rango *y1 – y2* debe ser estrictamente mayor que el rango *x1 – x2*, si no se cumple la condición la función asigna 0 como resultado de la operación.

De esta manera, los valores de voltaje se pueden desplegar en la pantalla LCD como números con punto flotante. La imagen de la Figura 5.53 muestra el valor del voltaje cuando el potenciómetro se encuentra en un valor medio de su resistencia interna, por lo que corresponde a un valor cercano a 2.5 V.

Figura 5.53 Pantalla LCD desplegando el valor del voltaje medido en el potenciómetro.

Debido a que el circuito de esta práctica es exactamente el mismo a la práctica anterior, no hay modificaciones en los componentes ni en las conexiones. Se emplean la misma lista de dispositivos de la Tabla 5.9 y las conexiones son como lo indican las figuras 5.51 y 5.52 de la práctica de la sección anterior.

5.6 CONTROL DE VELOCIDAD DE UN MOTOR DE CD

En las prácticas de las secciones 5.4 y 5.4, se programó el FPGA para obtener y procesar una señal analógica, obtenida por un convertidor analógico a digital, y cuyo valor se desplegó en una pantalla de cristal líquido. Por lo general, cuando se lee una señal de voltaje, puede ser para obtener información y procesarla para que en el circuito se realicen otros procesos con base en la información que se obtiene de algún sensor o transductor. En muchas ocasiones, las señales de voltaje son usadas para accionar actuadores cuando se trata de sistemas de control. En esta práctica se usará el valor obtenido por el convertidor analógico a digital para controlar un motor de corriente directa usando la modulación por ancho de pulso, PWM.

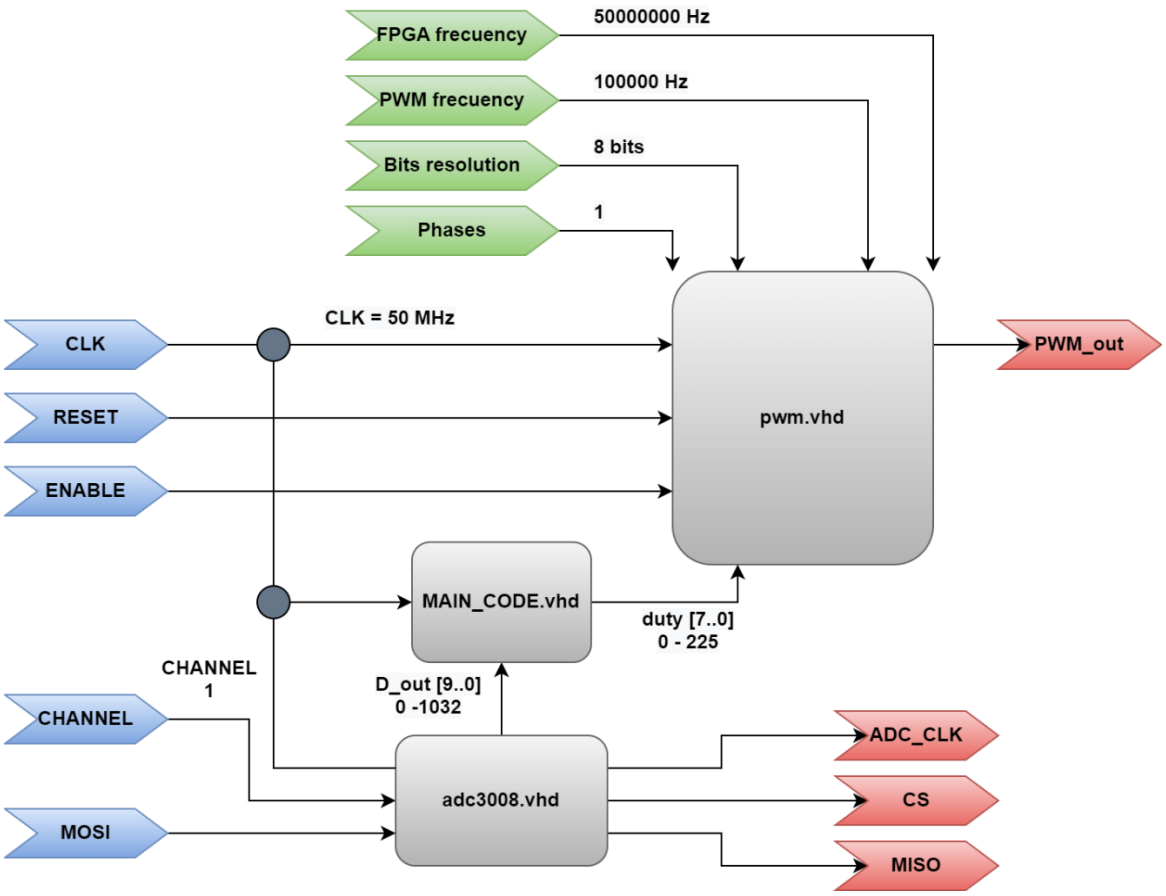


Figura 5.54 Diagrama de bloques para controlador de motor de CD.

En el diagrama de la Figura 5.54, se presenta el diagrama de bloques que describe la estructura de este circuito, nuevamente se utiliza el bloque para el generador de PWM que fue implementado en la práctica de la sección 5.2, recordando que el bloque fue configurado para una señal PWM con resolución de 8 bits y en dicha práctica se implementó un bloque *LPM_CONTSTANT* para establecer el valor de PWM en 128 representado por un número de 8 bits. En contraste, en esta práctica el FPGA será programado para generar una señal PWM variable y que será proporcional a la señal de voltaje de un potenciómetro obtenida del convertidor analógico a digital.

El primer paso será crear el proyecto y el diagrama de bloques en Quartus II, el cual se debe asignar el nombre *DC_MOTOR_CONTROLLER*. Después, se crea el archivo VHDL que, al igual que en las prácticas anteriores, se deberá nombrar como *MAIN_CODE* y se crea también el bloque asociado al archivo VHDL con la metodología de la sección 3.4.2. Por último, se descarga de la carpeta virtual, citada en la sección D del apéndice, las carpetas del bloque generador de PWM (citada en la sección C.2), la carpeta del controlador del convertidor analógico a digital MCP3008 (citado en la sección C.3), y también el archivo de biblioteca *instrument.vhd.* En la Figura 5.55 se muestra en el administrador de archivos la carpeta del proyecto con el archivo y carpetas mencionados

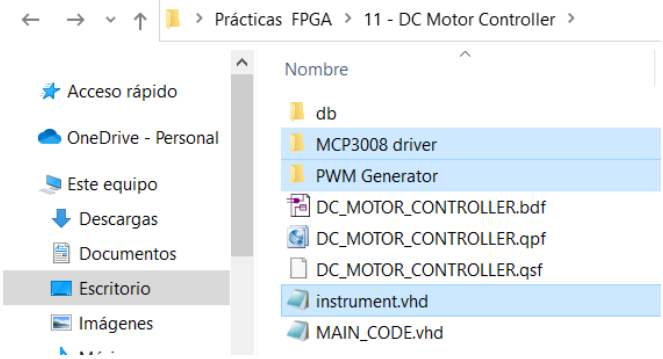


Figura 5.55 Carpetas del generador de PWM, MCP3008 y el archivo de biblioteca "instrument.vhd" dentro de la carpeta del proyecto.

Para incluir los archivos en el proyecto, se requiere hacer clic en la pestaña *"Files"* de la ventana *"Proyect navigator"*, después se hace clic derecho en el icono *"Files"*, y se

accede a la opción *"Add/Remove files"* donde se abrirá el administrador de archivos del proyecto y se deberán incluir los archivos VHDL de los bloques descargados, así como el archivo de biblioteca *instriment.vhd*; el resultado de este proceso lo muestra la Figura 5.56

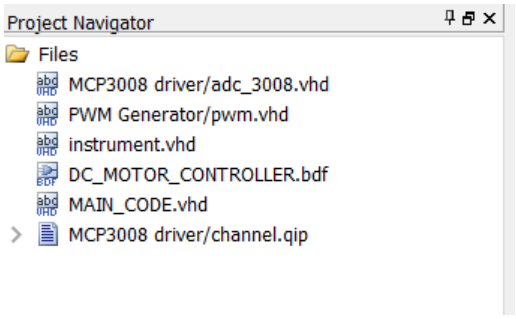


Figura 5.56 Ventana del navegador con los archivos del generador de PWM y el MCP3008.

Para el bloque *MAIN_CODE*, se requiere escribir el código B.10 el cual tiene la misma estructura que en los códigos de las secciones 5.4 y 5.4. Las entradas del código son la señal de reloj *CLK*, y el valor del convertidor analógico a digital de 10 bits, representado por la señal *ADC*, mientras que la salida del bloque es un número de 8 bits representado por la señal *PWM*.

Código 5.8 Arquitectura del código B.10

```
18 architecture behavioral of MAIN_CODE is
19
20  constant max_pwm : integer := 255;
21  constant resolution : integer := 10**4;
22  signal ADC_VAL : integer range 0 to 1023;
23  signal pwmx10000 : integer range 0 to max_pwm*resolution;
24
25  begin
26
27   ADC_VAL <= to_integer(unsigned(ADC));
28   pwmx10000 <= mapping(ADC_VAL,0,1023,0,max_pwm*resolution);
29   PWM <=
30 std_logic_vector(to_unsigned(pwmx10000/resolution,PWM'length));
31 end architecture;
```

En la práctica de la sección anterior, se implementó un proceso donde se mapearon los valores desde 0 a 1023, que corresponden a la lectura del convertidor analógico a digital, a un valor dentro del rango desde 0 hasta 50,000 que corresponde al valor de voltaje obtenido multiplicado por 10,000 que significa la precisión de la medición.

La señal de entrada, que corresponde al convertidor analógico a digital, tiene un rango de valores desde 0 hasta 1023, mientras que la señal de salida, el PWM de 8 bits, tiene un rango de valores es desde 0 hasta 255, y por lo tanto, el rango de salida es menor al rango de entrada. La función *mapping()* (contenida en el archivo de biblioteca *instrument.vhd*) usa un proceso matemático para calcular la ecuación de una recta, si el rango *y2-y1* es menor que el rango *x2-x1*, en la pendiente de la recta se tendrá un valor entre 0 y 1. Quartus II no puede sintetizar números con punto flotante en el FPGA, por lo tanto ese número se redondeará a 0 y el resultado de la función *mapping()* será 0, sin importar el número que reciba en sus argumentos. Para evitar esta situación, se usará la misma técnica de la práctica de la sección 5.5, el rango del PWM se multiplica por una potencia de 10 la cual será 10^4 y, de esta manera, la función *mapping()* trabajará con un intervalo *y2-y1* desde 0 hasta 2,550,000 y en consecuencia la pendiente de la recta será un número mayor a 1.

En la línea 23 del código 5.8, se declara una señal nombrada *pwmX10000* restringido en el rango desde 0 hasta 2,550,000, y en la línea 28 se asigna el resultado del proceso de mapeo de los valores obtenidos del convertidor (representado por la entrada de 10 bits *ADC*) situados entre 0 y 1023.

Finalmente, la salida *PWM* se declara un número de 8 bits, la cual enviará la información al bloque *pwm.vhd* para generar la señal de control de velocidad del motor, y en esta salida se asigna el valor de la señal *pwmX10000* dividida por la potencia de 10 usada (10^4 o 10,000) para realizar el proceso de mapeo, y obtener los valores de la señal PWM en el rango desde 0 hasta 255. Esta técnica permite trabajar con los números enteros para realizar operaciones que involucran números con punto flotante.

Cuando se haya escrito el código B.10 en el bloque *MAIN_CODE*, se debe construir el diagrama de bloques que se muestra en la Figura 5.57, que sigue la estructura del diagrama presentado en la Figura 5.54.

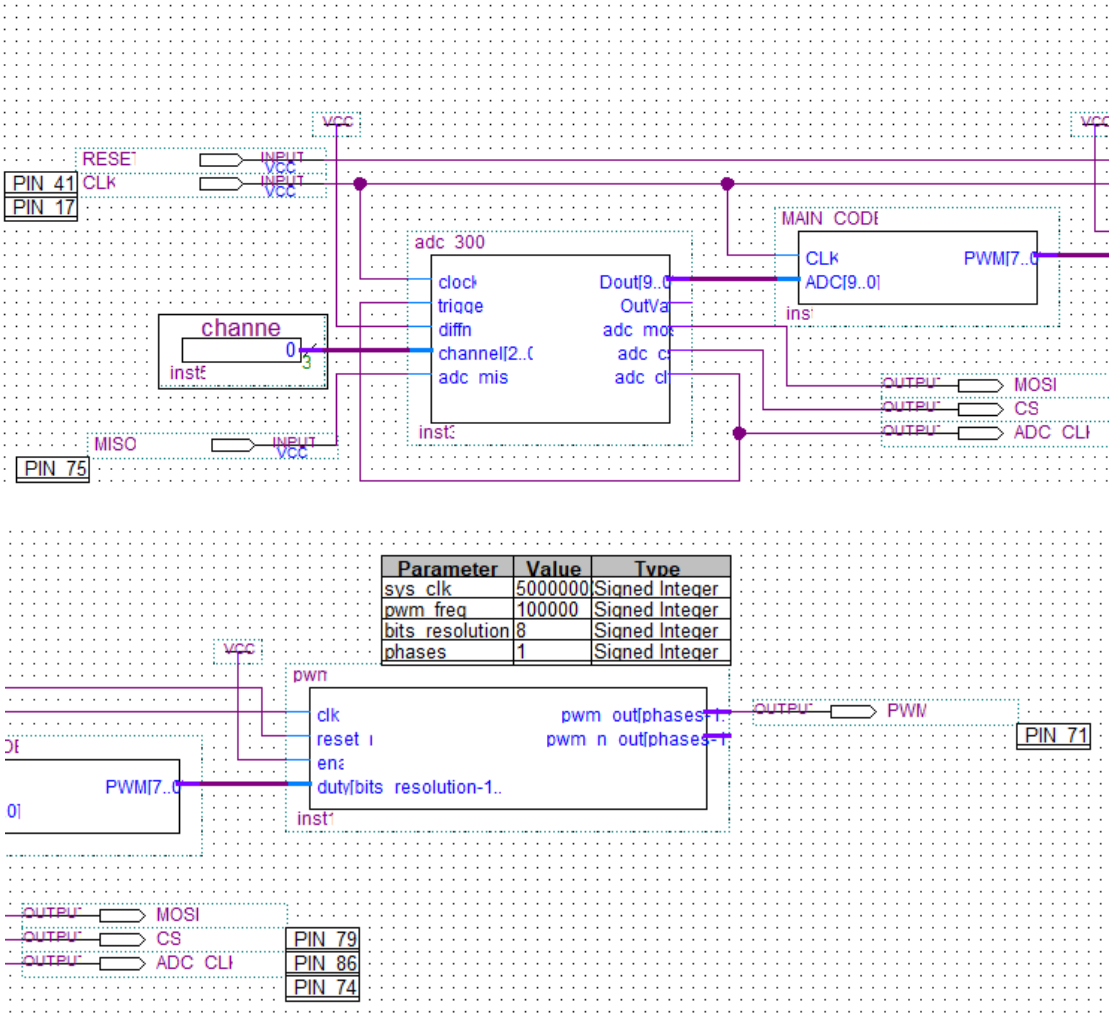


Figura 5.57 Diagrama de bloques en Quartus II para controlador de motor CD.

Con la información de la Tabla 5.13, se asocian los pines físicos del FPGA a las entradas y salidas del diagrama de bloques de la Figura 5.57. Por otro lado, para la construcción física del circuito, se usan los componentes listados en la Tabla 5.14. De la misma manera que en las prácticas de las secciones 5.1 y 5.2, se tiene que usar una alimentación adicional de 9 V y un circuito de potencia para que el pin del FPGA pueda accionar el motor. Las figuras 5.58 y 5.59 muestran las conexiones necesarias entre el FPGA y los componentes ensamblados en la placa de pruebas

Tabla 5.13 Lista de dispositivos para la práctica 11.

Dispositivo	Valor	Cantidad	Símbolos asociados
Resistencia	10 kΩ	1	R1
Condensador cerámico	220 nF	1	C1
Botón pulsador	---	1	KEY1
Motor de CD	9 V	1	M1
Potenciómetro	1 kΩ	1	RP1
Puente H	L293D	1	U1
Convertidor ADC	MCP3008	1	U2
Batería	9 V	1	vcc2

Tabla 5.14 Pines del FPGA asociados al circuito controlador de motor de corriente directa.

Entradas			**Salidas**		
Señal	VHDL	PIN FPGA	Señal	VHDL	PIN FPGA
Reloj FPGA	CLK	17	CS	CS	86
RESET	RESET	40	Reloj ADC	ADC_CLK	74
MOSI	MOSI	79	MISO	MISO	79
			PWM	PWM_out	71

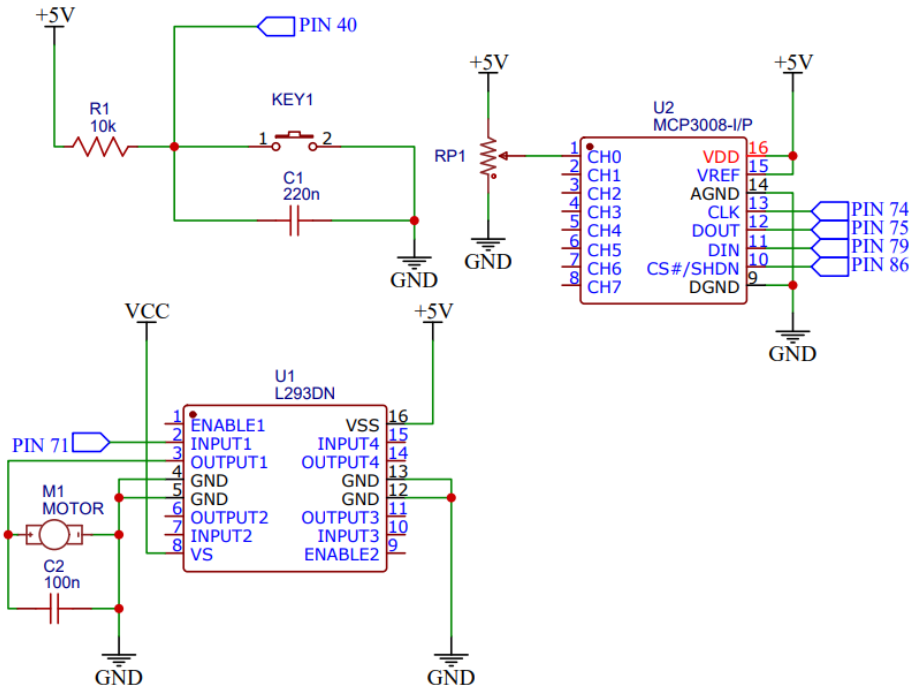


Figura 5.58 Diagrama esquemático para el circuito controlador de un motor de CD.

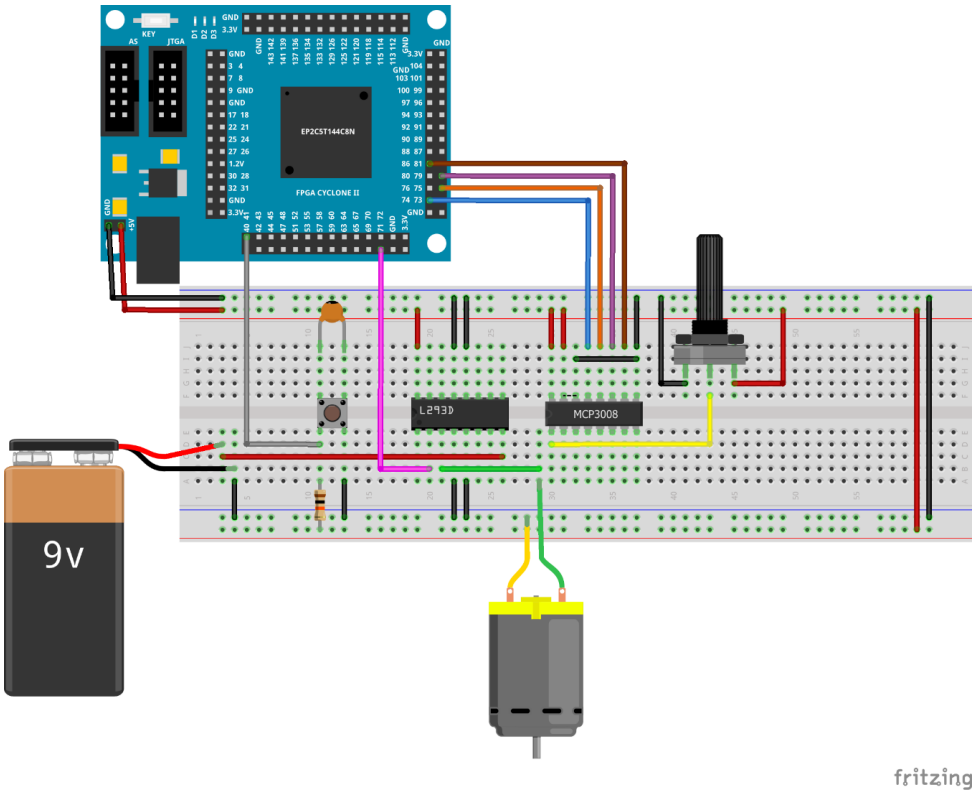


Figura 5.59 Diagrama de conexiones del circuito físico de la práctica 11 implementado en una placa de pruebas.

APÉNDICE

A ESPECIFICACIONES DE LAS PLACAS DE DESARROLLO

A.1 CYCLONE II EP2C5T144

En la Figura A.1, se pueden observar el detalle de los pines, los puertos y los dispositivos periféricos de la placa de desarrollo Cyclone II:

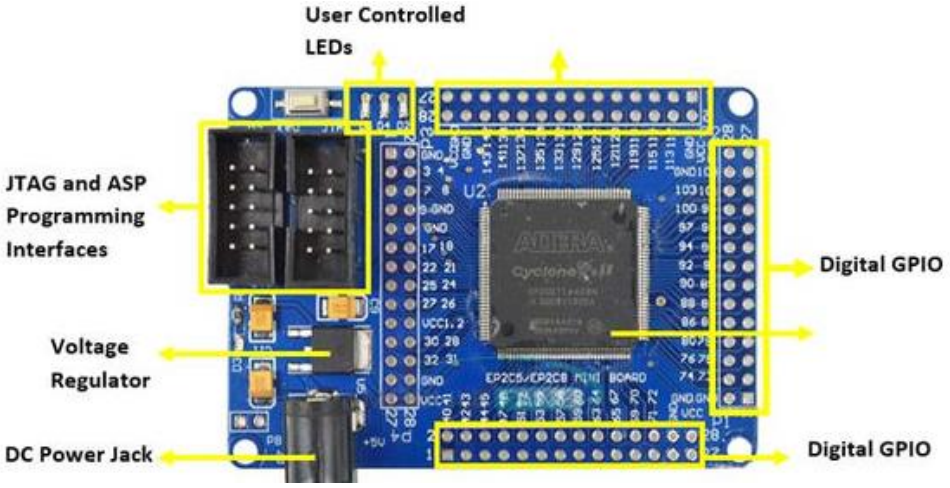


Figura A.1 Placa de desarrollo Cyclone II.

La placa de desarrollo contiene componentes electrónicos lo cuales están asociados a algunos pines del FPGA, en la Tabla A.1 se enlistan los dispositivos de la placa de desarrollo.

Tabla A.1 Distribución de pines de la tarjeta de desarrollo Cyclone II.

Dispositivo periférico	FPGA PIN
LED 1	PIN 3
LED 2	PIN 7
LED 3	PIN 9
BUTTON	PIN 144
FPGA 50 MHz CLOCK	PIN 17

A.2 CYCLONE IV EP4CE6E22C8N

La placa Cyclone IV un número mucho mayor de dispositivos periféricos que la placa de desarrollo Cyclone II, en la Figura A.2 se indica la ubicación de cada uno de los dispositivos periféricos soldados a la placa.

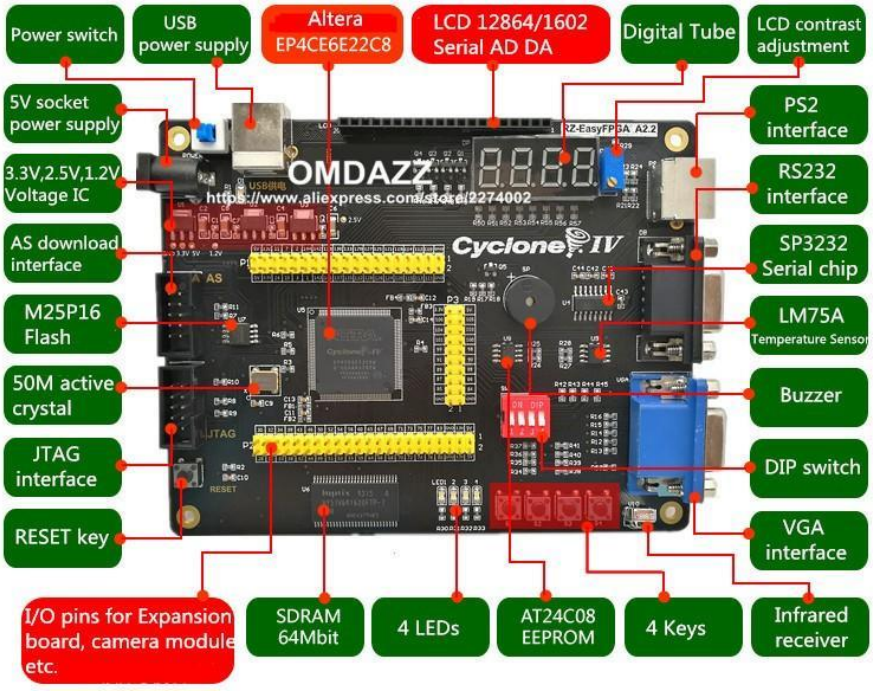


Figura A.2 Ubicación de los dispositivos periféricos de la placa de desarrollo Cyclone IV.

En la Tabla A.2 se puede consultar los pines del FPGA que corresponden a cada uno de los periféricos y puertos mostrados en la Figura A.2, y también se ubica al pin que corresponde la señal de reloj de 50 MHz.

Tabla A.2 Pines correspondientes a los dispositivos periféricos de la placa Cyclone IV.

Dispositivo periférico	**Pin del FPGA**	**Dispositivo periférico**	**Pin del FPGA**
Botón RESET	25	Despliegue de 7 segmentos	
Reloj de 50 MHz	23	Despliegue 1	133
Dip-switch		Despliegue 2	135
Interruptor 1	88	Despliegue 3	136
Interruptor 2	89	Despliegue 4	137
Interruptor 3	90	Segmento A	128
Interruptor 4	91	Segmento B	121
Zumbador		Segmento C	125
Zumbador	110	Segmento D	129
Botones pulsadores		Segmento E	132
Botón 1	88	Segmento F	126
Botón 2	89	Segmento G	124
Botón 3	90	Segmento DOT	127
Botón 4	91	**Memoria volátil SDRAM**	
Leds		DQ0	28
Led 1	87	DQ1	30
Led 2	86	DQ2	31
Led 3	85	DQ3	32
Led 4	84	DQ4	33
Comunicación UART		DQ5	34
Transmisor TX	114	DQ6	38
Reseptro RX	115	DQ7	39
		DQ8	54

Memoria EEPROM I2C		DQ9	53
SCL	112	DQ10	52
SDA	113	DQ11	51
I2C SCL	99	DQ12	50
I2C SDA	98	DQ13	49
PS2		DQ14	46
Reloj PS	119	DQ15	44
Data PS	120	A0	76
Sensor infrarrojo		A1	77
IR	100	A2	80
Salida VGA		A3	83
VGA HSYNC	101	A4	68
VGA VSYNC	103	A5	67
VGA B	104	A6	66
VGA G	105	A7	65
VGA R	106	A8	64
Puerto para pantalla LCD		A9	60
RS	141	A10	75
RW	138	A11	59
E	143	SD BS0	73
D0	142	SD BS1	74
D1	1	SD LDQM	42
D2	144	SD UDQM	55
D3	3	SD CKE	58
D4	2	SD CLK	43
D5	10	SD CS	72
D6	7	SD RAS	71
D7	11	SD CAS	70
		SD WE	69

B CÓDIGOS DE LAS PRÁCTICAS

B.1 ENCENDIDO DE UN LED MEDIANTE UN BOTÓN

Código B1.1 Encendido de un led accionado por un botón

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity LED_ON_FPGA is
 6 port (
 7  BUTTON : in  std_logic;
 8  LED : out std_logic
 9 );
10 end entity;
11
12 architecture main of LED_ON_FPGA  is
13
14 begin
15
16 LED <= BUTTON;
17
18 end architecture;
```

Código B1.2 Apagado de un led accionado por un botón

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity LED_ON_FPGA is
 6 port (
 7  BUTTON : in  std_logic;
 8  LED : out std_logic
 9 );
10 end entity;
11
12 architecture main of LED_ON_FPGA  is
13 begin
14
15  LED <= not BUTTON;
16  end architecture;
```

B.2 PARPADEO DE UN LED

Código B2 Divisor de frecuencia programado por una unidad secuencial "process"

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity LED_BLINKING is
 6 port (
 7
 8  CLK: in std_logic;
 9  LED: out std_logic);
10
11 end entity;
12
13 architecture behavioral of LED_BLINKING is
14
15  constant count_limit: integer := 12499999;
16  signal count: integer range 0 to 50e6:= 0;
17  signal led_state: std_logic := '0';
18
19 begin
20
21  BLINK: process (CLK)
22  begin
23   if rising_edge (CLK) then
24    if (count = count_limit) then
25     led_state <= not led_state;
26     led <= led_state;
27     count <= 0;
28    else
29     count<= count + 1;
30    end if;
31   end if;
32  end process;
33
34 end architecture;
```

B.3 PROGRAMACIÓN DE UNA FUNCIÓN BOOLEANA

Código B3 Programación de una función booleana de tres entradas y una salida

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity BOOLEAN_FUNCTION is
 6 port (
 7
 8  A : in std_logic_vector (1 downto 0);
 9  B : in std_logic_vector (1 downto 0);
10  C : out std_logic_vector (3 downto 0));
11
12 end entity;
13
14 architecture main of BOOLEAN_FUNCTION is
15 begin
16
17  C(0) <= (A(1) and B(1)) and (A(0) and A(0));
18  C(1) <= A(1) and (B(1) and (not A(0) or not B(0)));
19  C(2) <= (A(1) and A(0)) xor (A(0) and B(1));
20  C(3) <= A(0) and B(0);
21
22 end architecture;
```

B.4 DECODIFICADOR DE 4 BITS A 7 SEGMENTOS

Código B4 Decodificador de numero binario de 4 bits a despliegue de 7 segmentos

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity DISPLAY_7_SEGMENTS_DECODER is
 6 port(
 7
 8   S : in  std_logic_vector(3 downto 0);
 9   D7SEG : out std_logic_vector(6 downto 0)
10 );
11
```

```
12 end entity;
13
14 architecture main of DISPLAY_7_SEGMENTS_DECODER is
15 begin
16
17  process (S) is
18  begin
19
20   case(S) is
21    when x"0" =>
22     D7SEG <= "1111110";
23    when x"1" =>
24     D7SEG <= "0110000";
25    when x"2" =>
26     D7SEG <= "1101101";
27    when x"3" =>
28     D7SEG <= "1111001";
29    when x"4" =>
30     D7SEG <= "0110011";
31    when x"5" =>
32     D7SEG <= "1011011";
33    when x"6" =>
34     D7SEG <= "1011111";
35    when x"7" =>
36     D7SEG <= "1110000";
37    when x"8" =>
38     D7SEG <= "1111111";
39    when x"9" =>
40     D7SEG <= "1111011";
41    when others =>
42     D7SEG <= "0000000";
43   end case;
44  end process;
45 end architecture
```

B.5 CONTADOR ASCENDENTE Y DESCENDENTE.

Código B.5.1 Divisor de frecuencia.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity frecuency_divider is
 6
```

```
 7 generic (
 8     count_limit : integer := 12499999
 9 );
10
11 port (
12
13  CLK : in std_logic;
14  LED : out std_logic);
15
16 end entity;
17
18 architecture behavioral of frecuency_divider is
19
20  signal count : integer range 0 to 50e6   := 0;
21  signal led_state : std_logic := '0';
22
23 begin
24
25  blink: process (CLK)
26  begin
27   if rising_edge (CLK) then
28    if (count = count_limit) then
29     led_state <= not led_state;
30     led <= led_state;
31     count <= 0;
32    else
33     count<= count + 1;
34    end if;
35   end if;
36  end process;
37
38 end architecture;
```

Código B.5.2 Máquina de estados.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity COUNTER_MACHINE is
 6 port(
 7   CLK     : in std_logic;
 8   RESET   : in std_logic;
 9   SWITCH  : in std_logic;
10   NUM_BIN : out std_logic_vector(3 downto 0)
11 );
12 end entity;
13
14 architecture main of COUNTER_MACHINE is
```

```
15
16 type     states is
17 (num_0,num_1,num_2,num_3,num_4,num_5,num_6,num_7,num_8,num_9);
18 signal   Q_bus: states :=   num_1;
19 signal   D_bus: states;
20 signal   count: integer := 0;
21
22 begin
23
24  FlipFlop: process (CLK) is
25  begin
26
27   if(rising_edge(CLK)) then
28    if(RESET = '1') then
29     Q_bus <= D_bus;
30    else
31     Q_bus <= num_0;
32    end if;
33   end if;
34
35  end process;
36
37  Counter: process (CLK) is
38  begin
39   case (Q_bus) is
40
41   when num_0 =>
42     NUM_BIN <= "0000";
43     if(SWITCH = '0') then
44      D_bus <= num_1;
45     else
46      D_bus <= num_9;
47     end if;
48
49        when num_1 =>
50
51     NUM_BIN <= "0001";
52     if(SWITCH = '0') then
53      D_bus <= num_2;
54     else
55      D_bus <= num_0;
56     end if;
57
58    when num_2 =>
59
60     NUM_BIN <= "0010";
61     if(SWITCH = '0') then
62      D_bus <= num_3;
63     else
64      D_bus <= num_1;
65     end if;
```

```
 66
 67    when num_3 =>
 68
 69         NUM_BIN <= "0011";
 70     if(SWITCH = '0') then
 71      D_bus <= num_4;
 72     else
 73      D_bus <= num_2;
 74     end if;
 75
 76    when num_4 =>
 77
 78     NUM_BIN <= "0100";
 79     if(SWITCH = '0') then
 80      D_bus <= num_5;
 81     else
 82      D_bus <= num_3;
 83     end if;
 84
 85    when num_5 =>
 86
 87     NUM_BIN <= "0101";
 88         if(SWITCH = '0') then
 89      D_bus <= num_6;
 90     else
 91      D_bus <= num_4;
 92     end if;
 93
 94    when num_6 =>
 95
 96     NUM_BIN <= "0110";
 97     if(SWITCH = '0') then
 98      D_bus <= num_7;
 99     else
100      D_bus <= num_5;
101     end if;
102
103    when num_7 =>
104
105     NUM_BIN <= "0111";
106     if(SWITCH = '0') then
107      D_bus <= num_8;
108     else
109      D_bus <= num_6;
110     end if;
111
112    when num_8 =>
113
114     NUM_BIN <= "1000";
115     if(SWITCH = '0') then
116      D_bus <= num_9;
```

```
117     else
118      D_bus <= num_7;
119     end if;
120
121     when num_9 =>
122
123      NUM_BIN <= "1001";
124      if(SWITCH = '0') then
125       D_bus <= num_0;
126      else
127       D_bus <= num_8;
128      end if;
129
130     when others =>
131      D_bus <= D_bus;
132
133   end case;
134  end process;
135
136 end architecture;
```

B.6 CONTROLADOR DE UN MOTOR PASO A PASO

Código B.5.2 Máquina de estados.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity stepper_motor is
 6 port(
 7   CLK  : in std_logic;
 8   DIRECTION : in std_logic;
 9   SPEED: in std_logic_vector(7 downto 0);
10   A   :out std_logic;
11   n_A  :out std_logic;
12   B   :out std_logic;
13   n_B  :out std_logic
14 );
15 end entity;
16
17 architecture main of stepper_motor is
18
19  function MAP_MOTOR( VEL : integer )
20   return integer is
```

```
21    variable counter : integer ;
22   begin
23    if(VEL > 0 and VEL < 44) then
24         counter := VEL*(-5813) + 499999;
25        elsif(VEL > 43 and VEL < 86) then
26         counter := VEL*(-2976) + 377975;
27        elsif(VEL > 85 and VEL < 171) then
28         counter := VEL*(-735) + 187499;
29        elsif(VEL > 172 and VEL < 256) then
30         counter := VEL*(-245) + 104199;
31        end if;
32   return counter ;
33  end function ;
34
35  type  states is
36
37 (state_0,state_1,state_2,state_3,state_4,state_5,state_6,state_7);
38  signal Q_bus : states := state_0;
39  signal D_bus : states;
40  signal count : integer  := 0;
41  signal speed_value : integer := to_integer(unsigned(speed));
42  signal count_limit : integer := MAP_MOTOR(speed_value);
43  signal divider_state : std_logic :='0';
44  signal clk_motor : std_logic;
45
46 begin
47
48  Frecuency_divider:  process(CLK) is
49  begin
50   if(rising_edge (CLK)) then
51    if(count = count_limit) then
52     divider_state <= not divider_state;
53     count    <= 0;
54    else
55     count <= count +1;
56    end if;
57   end if;
58  end process;
59
60  clk_motor   <= divider_state;
61
62  FlipFlop: process (clk_motor ) is
63  begin
64   if(rising_edge(clk_motor )) then
65     Q_bus <= D_bus;
66   end if;
67  end process;
68
69  Stepper_motor_machine : process (clk_motor ) is
70  begin
71   case (Q_bus) is
```

```
 72
 73    when state_0 =>
 74
 75     A   <= '1';
 76     B  <= '1';
 77     n_A <= '0';
 78     n_B <= '0';
 79     if(DIRECTION = '0') then
 80      D_bus <= state_1;
 81     else
 82      D_bus <= state_4;
 83     end if;
 84
 85    when state_1 =>
 86
 87     A   <= '1';
 88     B  <= '0';
 89     n_A <= '0';
 90     n_B <= '1';
 91     if(DIRECTION = '0') then
 92      D_bus <= state_2;
 93     else
 94      D_bus <= state_5;
 95     end if;
 96
 97    when state_2 =>
 98
 99     A   <= '0';
100     B  <= '0';
101     n_A <= '1';
102     n_B <= '1';
103     if(DIRECTION = '0') then
104      D_bus <= state_3;
105     else
106      D_bus <= state_6;
107     end if;
108
109    when state_3 =>
110     A   <= '0';
111     B  <= '1';
112     n_A <= '1';
113     n_B <= '0';
114     if(DIRECTION = '0') then
115      D_bus <= state_0;
116     else
117      D_bus <= state_7;
118     end if;
119
120    when state_4 =>
121
122     A   <= '0';
```

```
123     B  <= '1';
124     n_A <= '1';
125     n_B <= '0';
126     if(DIRECTION = '0') then
127      D_bus <= state_0;
128     else
129      D_bus <= state_7;
130     end if;
131
132    when state_5 =>
133
134     A   <= '1';
135     B  <= '1';
136     n_A <= '0';
137     n_B <= '0';
138     if(DIRECTION = '0') then
139      D_bus <= state_1;
140     else
141      D_bus <= state_4;
142     end if;
143
144    when state_6 =>
145
146     A   <= '1';
147     B  <= '0';
148     n_A <= '0';
149     n_B <= '1';
150     if(DIRECTION = '0') then
151      D_bus <= state_2;
152     else
153      D_bus <= state_5;
154     end if;
155
156    when state_7 =>
157
158     A   <= '0';
159     B  <= '0';
160     n_A <= '1';
161     n_B <= '1';
162     if(DIRECTION = '0') then
163      D_bus <= state_3;
164     else
165      D_bus <= state_6;
166     end if;
167
168    when others =>
169     D_bus <= D_bus;
170   end case;
171  end process;
172 end architecture;
```

B.7 EMPLEDO DE UNA PANTALLA LCD 16 X 2

Código B.7 Código para escribir mensajes en pantalla LCD.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 library work;
 6 use work.instrument.all;
 7
 8 entity MAIN_CODE is
 9
10 port(
11
12  CLK: in std_logic;
13  LINE1_lcd : out  std_logic_vector(127 downto 0);--16char x
14 8bit
15  LINE2_lcd : out  std_logic_vector(127 downto 0)--16char x 8bit
16
17 );
18
19 end entity;
20
21 architecture behavioral of MAIN_CODE is
22
23
24 begin
25
26  LINE1_lcd <= to_std_logic_vector("HELLOW WORLD :) ");
27  LINE2_lcd <= to_std_logic_vector("A FPGA is used  ");
28
29 end architecture;
```

B.8 CONVERTIDOR ANALÓGICO A DIGITAL

Código B.8 Código para mostrar lecturas del convertidor analógico a digital en pantalla LCD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
```

```
 4
 5 library work;
 6 use work.instrument.all;
 7
 8 entity MAIN_CODE is
 9
10 port(
11
12  CLK: in std_logic;
13  ADC : in std_logic_vector(9 downto 0);
14  LINE1_lcd : out  std_logic_vector(127 downto 0);
15   --16char x 8bit
16  LINE2_lcd : out  std_logic_vector(127 downto 0)
17   --16char x 8bit
18 );
19 end entity;
20
21 architecture behavioral of MAIN_CODE is
22
23  signal ADC_VAL: integer range 0 to 1023;
24
25  begin
26
27   ADC_VAL <= to_integer(unsigned(ADC));
28   LINE1_lcd <=
29   to_std_logic_vector("BIN : " & to_string(ADC));
30   LINE2_lcd <=
31   to_std_logic_vector("NUMERIC :  " & to_string(ADC_VAL,5));
32 end architecture;
```

B.9 MEDICIÓN DE VOLTAJE

Código B.9 Código para mostrar lecturas del convertidor analógico a digital en pantalla LCD adaptadas para medir voltaje

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 library work;
 6 use work.instrument.all;
 7
 8 entity MAIN_CODE is
 9
```

```
port(

 CLK : in std_logic;
 ADC : in std_logic_vector(9 downto 0);
 LINE1_lcd : out  std_logic_vector(127 downto 0);
 LINE2_lcd : out  std_logic_vector(127 downto 0)

);

end entity;

architecture behavioral of MAIN_CODE is

 constant max_volt : integer := 5;
 constant resolution : integer := 10**4;
 signal ADC_VAL: integer range 0 to 1023;
 signal volts: integer range 0 to max_volt*resolution;

 begin

  ADC_VAL <= to_integer(unsigned(ADC));
  volts <= mapping(ADC_VAL,0,1023,0,max_volt*resolution);
  LINE1_lcd <= to_std_logic_vector("Voltmeter  FPGA ");
  LINE2_lcd <= to_std_logic_vector("Voltage :" &
  to_string(volts,7,4));
end architecture;
```

B.10 CONTROL DE VELOCIDAD DE UN MOTOR DE CD

Código B.10 Realiza el proceso de mapeo para controlar un motor de CD usando una señal PWM

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

library work;
use work.instrument.all;

entity MAIN_CODE is

port(

 CLK : in std_logic;
```

```
13  ADC : in std_logic_vector(9 downto 0);
14  PWM : out std_logic_vector(7 downto 0)
15 );
16 end entity;
17
18 architecture behavioral of MAIN_CODE is
19
20  constant max_pwm : integer := 255;
21  constant resolution : integer := 10**4;
22  signal ADC_VAL : integer range 0 to 1023;
23  signal pwmx10000 : integer range 0 to max_pwm*resolution;
24
25  begin
26
27   ADC_VAL <= to_integer(unsigned(ADC));
28   pwmx10000 <= mapping(ADC_VAL,0,1023,0,max_pwm*resolution);
29   PWM <=
30 std_logic_vector(to_unsigned(pwmx10000/resolution,PWM'length));
31             ture;
```

C ARCHIVOS DE BIBLIOTECA

C.1 CONTROLADOR PARA MOTOR PASO A PASO

Nombre del bloque: *stepper_motor*

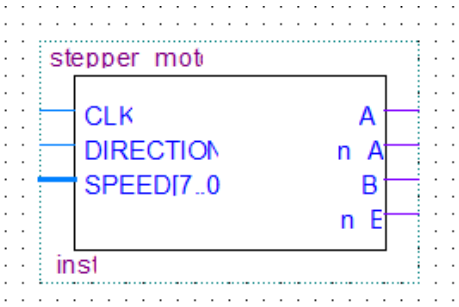


Figura B.1 Bloque del controlador de motor a pasos.

Este código corresponde a un controlador que permite controlar la velocidad de un motor a pasos, el cual recibe como entradas la señal de reloj del FPGA. De acuerdo con la Figura B.1, en el bit *DIRECTION* se establece el sentido de giro del motor y en *SPEED* se introduce un número binario de 8 bits, que representa un rango de valores desde 0 hasta 255, que representa la magnitud de velocidad del motor.

Código que define al bloque

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity stepper_motor is
 6 port(
 7   CLK  : in std_logic;
 8   DIRECTION : in std_logic;
 9   SPEED: in std_logic_vector(7 downto 0);
10   A   :out std_logic;
11   n_A  :out std_logic;
```

```
12   B   :out std_logic;
13   n_B  :out std_logic
14 );
15 end entity;
16
17 architecture main of stepper_motor is
18
19  function MAP_MOTOR( VEL : integer )
20   return integer is
21    variable counter : integer ;
22   begin
23    if(VEL > 0 and VEL < 44) then
24         counter := VEL*(-5813) + 499999;
25        elsif(VEL > 43 and VEL < 86) then
26         counter := VEL*(-2976) + 377975;
27        elsif(VEL > 85 and VEL < 171) then
28         counter := VEL*(-735) + 187499;
29        elsif(VEL > 172 and VEL < 256) then
30         counter := VEL*(-245) + 104199;
31        end if;
32   return counter ;
33  end function ;
34
35  type  states is
36
37 (state_0,state_1,state_2,state_3,state_4,state_5,state_6,state
38 _7);
39  signal Q_bus : states := state_0;
40  signal D_bus : states;
41  signal count : integer  := 0;
42  signal speed_value : integer := to_integer(unsigned(speed));
43  signal count_limit : integer := MAP_MOTOR(speed_value);
44  signal divider_state : std_logic :='0';
45  signal clk_motor : std_logic;
46
47 begin
48
49  Frecuency_divider:  process(CLK) is
50  begin
51   if(rising_edge (CLK)) then
52    if(count = count_limit) then
53     divider_state <= not divider_state;
54     count    <= 0;
55    else
56     count <= count +1;
57    end if;
58   end if;
59  end process;
60
61  clk_motor   <= divider_state;
62
```

```
 63  FlipFlop: process (clk_motor ) is
 64  begin
 65   if(rising_edge(clk_motor )) then
 66     Q_bus <= D_bus;
 67   end if;
 68  end process;
 69
 70  Stepper_motor_machine : process (clk_motor ) is
 71  begin
 72   case (Q_bus) is
 73
 74    when state_0 =>
 75
 76     A   <= '1';
 77     B  <= '1';
 78     n_A <= '0';
 79     n_B <= '0';
 80     if(DIRECTION = '0') then
 81      D_bus <= state_1;
 82     else
 83      D_bus <= state_4;
 84     end if;
 85
 86    when state_1 =>
 87
 88     A   <= '1';
 89     B  <= '0';
 90     n_A <= '0';
 91     n_B <= '1';
 92     if(DIRECTION = '0') then
 93      D_bus <= state_2;
 94     else
 95      D_bus <= state_5;
 96     end if;
 97
 98    when state_2 =>
 99
100     A   <= '0';
101     B  <= '0';
102     n_A <= '1';
103     n_B <= '1';
104     if(DIRECTION = '0') then
105      D_bus <= state_3;
106     else
107      D_bus <= state_6;
108     end if;
109
110    when state_3 =>
111     A   <= '0';
112     B  <= '1';
113     n_A <= '1';
```

```
114      n_B <= '0';
115      if(DIRECTION = '0') then
116       D_bus <= state_0;
117      else
118       D_bus <= state_7;
119      end if;
120
121     when state_4 =>
122
123      A   <= '0';
124      B  <= '1';
125      n_A <= '1';
126      n_B <= '0';
127      if(DIRECTION = '0') then
128       D_bus <= state_0;
129      else
130       D_bus <= state_7;
131      end if;
132
133     when state_5 =>
134
135      A   <= '1';
136      B  <= '1';
137      n_A <= '0';
138      n_B <= '0';
139      if(DIRECTION = '0') then
140       D_bus <= state_1;
141      else
142       D_bus <= state_4;
143      end if;
144
145     when state_6 =>
146
147      A   <= '1';
148      B  <= '0';
149      n_A <= '0';
150      n_B <= '1';
151      if(DIRECTION = '0') then
152       D_bus <= state_2;
153      else
154       D_bus <= state_5;
155      end if;
156
157     when state_7 =>
158
159      A   <= '0';
160      B  <= '0';
161      n_A <= '1';
162      n_B <= '1';
163      if(DIRECTION = '0') then
164       D_bus <= state_3;
```

```
165      else
166       D_bus <= state_6;
167      end if;
168
169     when others =>
170      D_bus <= D_bus;
171    end case;
172   end process;
173 end architecture;
```

Código de elaboración propia.

C.2 GENERADOR DE PWM

Nombre del bloque: *pwm*

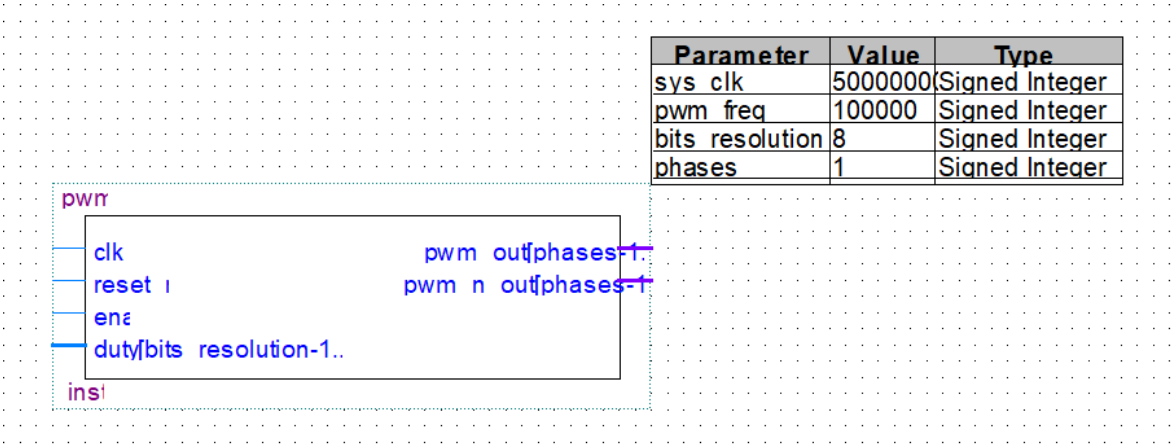


Figura B.2 Bloque generador de señal PWM.

Este bloque permite generar una señal de PWM, de acuerdo con la Figura B.2, las entradas del bloque son *clk* (la señal de reloj del FPGA), *reset* (señal de reinicio), *enable* (señal de habilitación del bloque) y *duty* (valor del PWM en número binario). Además, tiene cuatro variables paramétricas: *sys_clk* (valor de la frecuencia del FPGA en Hz), *pwm_frec* (frecuencia deseada a la que opera el PWM), *bits_resolution* (resolución del PWM en bits) y *phases* (número de salidas PWM del bloque).

Código que define al bloque

```
 1 ------------------------------------------------------------
 2 --
 3 --   FileName:         pwm.vhd
 4 --   Dependencies:     none
 5 --   Design Software:
 6 --   Quartus II 64-bit Version 12.1 Build 177
 7 --   SJ Full Version
 8 --
 9 --   HDL CODE IS PROVIDED "AS IS."  DIGI-KEY
10 --   EXPRESSLY DISCLAIMS ANY WARRANTY OF ANY KIND,
11 --   WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
12 --   LIMITED TO, THE IMPLIED WARRANTIES
13 --   OF MERCHANTABILITY, FITNESS FOR A
14 --   PARTICULAR PURPOSE, OR NON-INFRINGEMENT.
15 --   IN NO EVENT SHALL DIGI-KEY BE LIABLE FOR
16 --   ANY INCIDENTAL, SPECIAL, INDIRECT OR
17 --   CONSEQUENTIAL DAMAGES, LOST PROFITS OR
18 --   LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
19 --   PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY
20 --   OR SERVICES, ANY CLAIMS BY THIRD PARTIES
21 --   (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF),
22 --   ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER
23 --   SIMILAR COSTS.
24 --
25 --   Version History
26 --   Version 1.0 8/1/2013 Scott Larson
27 --     Initial Public Release
28 --   Version 2.0 1/9/2015 Scott Larson
29 --     Transistion between duty cycles always starts
30 --     at center of pulse to avoid
31 --     anomalies in pulse shapes
32 --
33 ------------------------------------------------------------
34
35 library ieee;
36 use ieee.std_logic_1164.all;
37 use ieee.std_logic_unsigned.all;
38
39 entity pwm is
40   generic(
41       sys_clk         : integer := 50_000_000;
42               --system clock frequency in hz
43       pwm_freq        : integer := 100_000;
44               --pwm switching frequency in hz
45       bits_resolution : integer := 8;
46               --bits of resolution setting the duty cycle
47       phases          : integer := 1);
48               --number of output pwms and phases
49   port(
```

```
      clk       : in  std_logic;
              --system clock
      reset_n   : in  std_logic;
              --asynchronous reset
      ena       : in  std_logic;
              --latches in new duty cycle
      duty      : in  std_logic_vector(bits_resolution-1
downto 0);
              --duty cycle
      pwm_out   : out std_logic_vector(phases-1 downto 0);
              --pwm outputs
      pwm_n_out : out std_logic_vector(phases-1 downto 0));
              --pwm inverse outputs
end pwm;

architecture logic of pwm is
  constant  period     :  integer := sys_clk/pwm_freq;
   --number of clocks in one pwm period
  type counters is array (0 to phases-1)
  of integer range 0 to period - 1;
   --data type for array of period counters
  signal  count        :  counters := (others => 0);
  --array of period counters
  signal   half_duty_new  :  integer
  range 0 to period/2 := 0;
   --number of clocks in 1/2 duty cycle
  type half_duties is array (0 to phases-1)
  of integer range 0 to period/2;
   --data type for array of half duty values
  signal  half_duty    :  half_duties := (others => 0);
   --array of half duty values (for each phase)
begin
  process(clk, reset_n)
  begin
    if(reset_n = '0') then
        --asynchronous reset
      count <= (others => 0);
              --clear counter
      pwm_out <= (others => '0');
              --clear pwm outputs
      pwm_n_out <= (others => '0');
              --clear pwm inverse outputs
    elsif(clk'event and clk = '1') then
        --rising system clock edge
      if(ena = '1') then
              --latch in new duty cycle
        half_duty_new <=

conv_integer(duty)*period/(2**bits_resolution)/2;
        --determine clocks in 1/2 duty cycle
      end if;
```

```
101       for i in 0 to phases-1 loop
102               --create a counter for each phase
103         if(count(0) = period - 1 - i*period/phases) then
104                 --end of period reached
105           count(i) <= 0;
106                     --reset counter
107           half_duty(i) <= half_duty_new;
108                     --set most recent duty cycle value
109         else
110                 --end of period not reached
111           count(i) <= count(i) + 1;
112                     --increment counter
113         end if;
114       end loop;
115       for i in 0 to phases-1 loop
116               --control outputs for each phase
117         if(count(i) = half_duty(i)) then
118                 --phase's falling edge reached
119           pwm_out(i) <= '0';
120                     --deassert the pwm output
121           pwm_n_out(i) <= '1';
122                     --assert the pwm inverse output
123         elsif(count(i) = period - half_duty(i)) then
124                 --phase's rising edge reached
125           pwm_out(i) <= '1';
126                     --assert the pwm output
127           pwm_n_out(i) <= '0';
128                     --deassert the pwm inverse output
129         end if;
130       end loop;
131     end if;
132   end process;
133 end logic;
```

Es código fue modificado para fines de este manual, se puede consultar el código original que fue en del sitio web de digikey.com en el siguiente enlace:

https://www.digikey.com/eewiki/pages/viewpage.action?pageId=20939345

C.3 CONVERTIDOR ANALÓGICO A DIGITAL MCP3008

Nombre del bloque: *adc_3008*

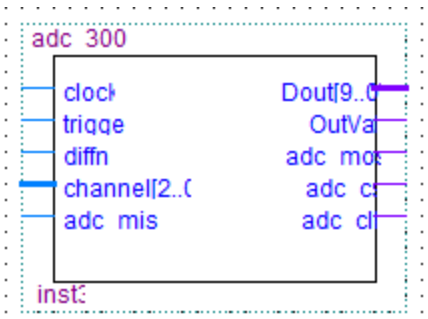


Figura B.3 Bloque de controlador para convertidor analógico a digital MCP3008.

El bloque mostrado en la Figura B.3, es una máquina de estados que realiza una comunicación serial síncrona con formato SPI que envía los registros al circuito MCP3008 en salida *adc_mosi* para realizar las mediciones del ADC. Después, la máquina espera un determinado tempo para recibir, los datos de la medición en la entrada *adc_miso* y finalmente se asigna el valor obtenido de manera secuencial en un número de 10 bits, que es asignado en la salida *Dout*. Finalmente, la salida *OutVal* es una señal de un solo bit que indica 1 cuando la medición ha terminado

La entrada nombrada *channel*, es un número entero donde se indica el canal que quiere ser leído en el circuito MCP3008, por defecto se establece el canal 1. Por otro lado, entrada *diffn* es un bit el cual si tiene un valor de 1 si las mediciones son diferenciales o 0 si son medidas directamente.

El bloque también contiene una entrada nombrada *trigger* la establece el inicio de la medición del ADC, es decir, si existe un flanco de subida (de 0 a 1) indica al circuito que tiene que accionar la máquina de estados para realizar la medición.

Código que define al bloque

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
```

```
 4
 5  entity adc_3008 is
 6
 7    generic(channel : integer := 1;
 8            diffn : std_logic:= '1');
 9    port
10      (
11        -- command input
12        clock    : in  std_logic;        -- 50MHz onboard oscillator
13        trigger  : in  std_logic;         -- assert to sample ADC
14                 -- single/differential inputs
15        -- data output
16        Dout     : out std_logic_vector(9 downto 0);  -- data from ADC
17        OutVal   : out std_logic;         -- pulsed when data sampled
18        -- ADC connection
19        adc_miso : in  std_logic;         -- ADC SPI MISO
20        adc_mosi : out std_logic;         -- ADC SPI MOSI
21        adc_cs   : out std_logic;         -- ADC SPI CHIP SELECT
22        adc_clk  : out std_logic          -- ADC SPI CLOCK
23        );
24
25  end adc_3008;
26
27   architecture behavioural of adc_3008 is
28
29    -- clock
30    signal adc_clock : std_logic := '0';
31
32    -- command
33    signal trigger_flag : std_logic                    := '0';
34    signal sgl_diff_reg : std_logic;
35    signal channel_reg  : std_logic_vector(2 downto 0) := (others => '0');
36    signal done         : std_logic                    := '0';
37    signal done_prev    : std_logic                    := '0';
38
39    -- output registers
40    signal val : std_logic                     := '0';
41    signal D   : std_logic_vector(9 downto 0) := (others => '0');
42
43    -- state control
44    signal state     : std_logic                    := '0';
45    signal spi_count : unsigned(4 downto 0)         := (others => '0');
46    signal Q         : std_logic_vector(9 downto 0) := (others => '0');
47
48  begin
49
50          --      clock divider
51          --      input clock: 50Mhz
52          --      50MHz/(3.6MHz*2) = 6.944 ~=7
53          --      adc clock: 3.57MHz
54          --      Vref = 5 V
55
56          ----------------------
57          ----------------------
58          Dout    <=      D;
59          ----------------------
60          ----------------------
```

```
 61
 62    clock_divider : process(clock)
 63    variable cnt : integer := 0;
 64    begin
 65      if rising_edge(clock) then
 66        cnt := cnt + 1;
 67        if cnt = 7 then
 68          cnt := 0;
 69          adc_clock <= not adc_clock;
 70        end if;
 71      end if;
 72    end process;
 73
 74    -- produce trigger flag
 75    trigger_cdc : process(adc_clock)
 76    begin
 77      if rising_edge(adc_clock) then
 78      if trigger = '1' and state = '0' then
 79        sgl_diff_reg <= diffn;
 80        channel_reg  <=
 81                  std_logic_vector(to_unsigned(channel-1,3));
 82        trigger_flag <= '1';
 83      elsif state = '1' then
 84        trigger_flag <= '0';
 85      end if;
 86    end if;
 87  end process;
 88
 89    adc_clk <= adc_clock;
 90    adc_cs  <= not state;
 91
 92    -- SPI state machine (falling edge)
 93    adc_sm : process(adc_clock)
 94    begin
 95      if adc_clock'event and adc_clock = '0' then
 96        if state = '0' then
 97          done <= '0';
 98          if trigger_flag = '1' then
 99            state <= '1';
100          else
101            state <= '0';
102          end if;
103        else
104          if spi_count = "10000" then
105            spi_count <= (others => '0');
106            state     <= '0';
107            done      <= '1';
108          else
109            spi_count <= spi_count + 1;
110            state     <= '1';
111          end if;
112        end if;
113      end if;
114    end process;
115
116    -- Register sample
117
```

```
118     outreg : process(adc_clock)
119     begin
120       if rising_edge(adc_clock) then
121         done_prev <= done;
122         if done_prev = '0' and done = '1' then
123           D   <= Q;
124           Val <= '1';
125         else
126           Val <= '0';
127         end if;
128       end if;
129     end process;
130
131     OutVal <= Val;
132
133     -- MISO shift register (rising edge)
134     shift_in : process(adc_clock)
135     begin
136       if adc_clock'event and adc_clock = '1' then
137         if state = '1' then
138           Q(0)          <= adc_miso;
139           Q(9 downto 1) <= Q(8 downto 0);
140         end if;
141       end if;
142     end process;
143
144     -- Decode MOSI output
145     shift_out : process(state, spi_count, sgl_diff_reg, channel_reg)
146     begin
147       if state = '1' then
148         case spi_count is
149           when "00000" => adc_mosi <= '1';  -- start bit
150           when "00001" => adc_mosi <= sgl_diff_reg;
151           when "00010" => adc_mosi <= channel_reg(2);
152           when "00011" => adc_mosi <= channel_reg(1);
153           when "00100" => adc_mosi <= channel_reg(0);
154           when others  => adc_mosi <= '0';
155         end case;
156       else
157         adc_mosi <= '0';
158       end if;
159     end process;
      end architecture;
```

El código contenido en el bloque de este bloque fue modificado para uso de este manual. Consulta el código original en la página de Digilent.com en el siguiente enlace: https://forum.digilentinc.com/topic/17410-mcp3008-interfaced-with-basys-3-using-spi/

C.4 CONTROLADOR DE UNA PANTALLA LCD 16 X 2

Nombre del bloque: *lcd16x2_ctrl*

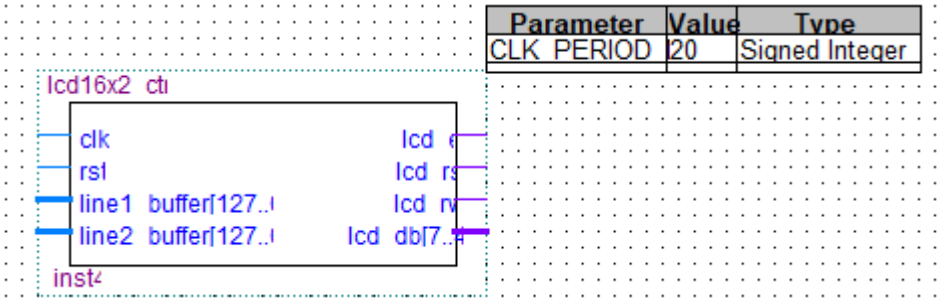


Figura B.4 Bloque del controlador para pantalla LCD 16x2.

El bloque de la Figura B.4 contiene un código donde se establece la comunicación con una pantalla de cristal líquido de 16x2 caracteres. Se usa una máquina de estados que envía los registros necesarios para inicializar y escribir el mensaje en los renglones de la pantalla.

Las entradas *clk* (la señal de reloj del FPGA) la cual depende de la variable paramétrica *CLK_PERIOD* que establece el periodo en nanosegundos de la frecuencia del FPGA, en caso de que sea diferente a 20 ns (50 MHz).

Las entradas *line1_buffer* y *line2_buffer* son dos vectores de 127 bits en los cuales se representan 16 caracteres de cada renglón de la pantalla, por lo que cada 8 bits de representa un solo carácter en su código ASCII.

Por otro lado, las salidas son los pines de la pantalla LCD que corresponden a *ENABLE, RESET*, y *READ/WRITE*. Se utiliza la interfaz de 4 bits para enviar los datos a la pantalla, por lo que solo se contemplan lo pines DB4, DB5, DB6 y DB7 en las salidas del bloque.

Código que define al bloque

```
1     ------------------------------------------------------
2     -- Title   : 16x2 LCD controller
3     -- Project  :
4     ------------------------------------------------------
5     -- File    : lcd16x2_ctrl.vhd
```

```
 6  -- Author   :  <stachelsau@T420>
 7  -- Company  :
 8  -- Created  : 2012-07-28
 9  -- Last update: 2012-11-28
10  -- Platform  :
11  -- Standard  : VHDL'93/02
12  ------------------------------------------------------------
13  -- Description: The controller initializes
14  -- the display when rst goes to '0'.
15  -- After that it writes the contend of the input signals
16  -- line1_buffer and line2_buffer continously to the display.
17  ------------------------------------------------------------
18  -- Copyright (c) 2012
19  ------------------------------------------------------------
20  -- Revisions :
21  -- Date    Version Author Description
22  -- 2012-07-28 1.0   stachelsau   Created
23  ------------------------------------------------------------
24
25  library ieee;
26  use ieee.std_logic_1164.all;
27
28  entity lcd16x2_ctrl is
29
30   generic (
31    CLK_PERIOD_NS : positive := 20
32    );  -- 50MHz
33
34   port (
35    clk : in std_logic;
36    rst : in std_logic;
37    lcd_e : out std_logic;
38    lcd_rs : out std_logic;
39    lcd_rw : out std_logic;
40    lcd_db : out std_logic_vector(7 downto 4);
41    line1_buffer : in std_logic_vector(127 downto 0):=
42    x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&
43    x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"; -- 16x8bit
44    line2_buffer : in std_logic_vector(127 downto 0):=
45    x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&
46    x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"&x"22"
47    );
48
49  end entity lcd16x2_ctrl;
50
51  architecture rtl of lcd16x2_ctrl is
52
53   constant DELAY_15_MS  : positive :=
54    15 * 10**6 / CLK_PERIOD_NS + 1;
55   constant DELAY_1640_US : positive :=
56    1640 * 10**3 / CLK_PERIOD_NS + 1;
57   constant DELAY_4100_US : positive :=
58    4100 * 10**3 / CLK_PERIOD_NS + 1;
59   constant DELAY_100_US : positive :=
60    100 * 10**3 / CLK_PERIOD_NS + 1;
61   constant DELAY_40_US  : positive :=
62    40 * 10**3 / CLK_PERIOD_NS + 1;
```

```
 63
 64      constant DELAY_NIBBLE   : positive :=
 65       10**3 / CLK_PERIOD_NS + 1;
 66      constant DELAY_LCD_E    : positive :=
 67       230 / CLK_PERIOD_NS + 1;
 68      constant DELAY_SETUP_HOLD : positive :=
 69       40 / CLK_PERIOD_NS + 1;
 70
 71      constant MAX_DELAY : positive := DELAY_15_MS;
 72
 73      -- this record describes one write operation
 74      type op_t is record
 75       rs   : std_logic;
 76       data  : std_logic_vector(7 downto 0);
 77       delay_h : integer range 0 to MAX_DELAY;
 78       delay_l : integer range 0 to MAX_DELAY;
 79      end record op_t;
 80      constant default_op   : op_t :=
 81      (rs => '1', data => X"00", delay_h => DELAY_NIBBLE,
 82       delay_l => DELAY_40_US);
 83      constant op_select_line1 : op_t :=
 84      (rs => '0', data => X"80", delay_h => DELAY_NIBBLE,
 85       delay_l => DELAY_40_US);
 86      constant op_select_line2 : op_t :=
 87      (rs => '0', data => X"C0", delay_h => DELAY_NIBBLE,
 88       delay_l => DELAY_40_US);
 89
 90      -- init + config operations:
 91      -- write 3 x 0x3 followed by 0x2
 92      -- function set command
 93      -- entry mode set command
 94      -- display on/off command
 95      -- clear display
 96      type config_ops_t is array(0 to 5) of op_t;
 97      constant config_ops : config_ops_t
 98       := (5 => (rs => '0', data => X"33",
 99           delay_h => DELAY_4100_US, delay_l => DELAY_100_US),
100         4 => (rs => '0', data => X"32",
101            delay_h => DELAY_40_US, delay_l => DELAY_40_US),
102         3 => (rs => '0', data => X"28",
103            delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
104         2 => (rs => '0', data => X"06",
105            delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
106         1 => (rs => '0', data => X"0C",
107            delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
108         0 => (rs => '0', data => X"01", delay_h =>
109             DELAY_NIBBLE, delay_l => DELAY_1640_US));
110
111     signal this_op : op_t;
112
113     type op_state_t is (IDLE,
114               WAIT_SETUP_H,
115               ENABLE_H,
116               WAIT_HOLD_H,
117               WAIT_DELAY_H,
118               WAIT_SETUP_L,
119               ENABLE_L,
```

```
120               WAIT_HOLD_L,
121               WAIT_DELAY_L,
122               DONE);
123
124     signal op_state   : op_state_t := DONE;
125     signal next_op_state : op_state_t;
126     signal cnt      : natural range 0 to MAX_DELAY;
127     signal next_cnt   : natural range 0 to MAX_DELAY;
128
129     type state_t is (RESET,
130              CONFIG,
131              SELECT_LINE1,
132              WRITE_LINE1,
133              SELECT_LINE2,
134              WRITE_LINE2);
135
136     signal state   : state_t        := RESET;
137     signal next_state : state_t;
138     signal ptr    : natural range 0 to 15 := 0;
139     signal next_ptr  : natural range 0 to 15;
140
141    begin
142
143     proc_state : process(state, op_state, ptr,
144                  line1_buffer, line2_buffer) is
145     begin
146      case state is
147       when RESET =>
148        this_op  <= default_op;
149        next_state <= CONFIG;
150        next_ptr  <= config_ops_t'high;
151
152       when CONFIG =>
153        this_op  <= config_ops(ptr);
154        next_ptr  <= ptr;
155        next_state <= CONFIG;
156        if op_state = DONE then
157         next_ptr <= ptr - 1;
158         if ptr = 0 then
159          next_state <= SELECT_LINE1;
160         end if;
161        end if;
162
163       when SELECT_LINE1 =>
164        this_op <= op_select_line1;
165        next_ptr <= 15;
166        if op_state = DONE then
167         next_state <= WRITE_LINE1;
168        else
169         next_state <= SELECT_LINE1;
170        end if;
171
172       when WRITE_LINE1 =>
173        this_op   <= default_op;
174        this_op.data <= line1_buffer(ptr*8 + 7 downto ptr*8);
175        next_ptr   <= ptr;
176        next_state  <= WRITE_LINE1;
```

```
177          if op_state = DONE then
178           next_ptr <= ptr - 1;
179           if ptr = 0 then
180            next_state <= SELECT_LINE2;
181           end if;
182          end if;
183
184         when SELECT_LINE2 =>
185          this_op <= op_select_line2;
186          next_ptr <= 15;
187          if op_state = DONE then
188           next_state <= WRITE_LINE2;
189          else
190           next_state <= SELECT_LINE2;
191          end if;
192
193         when WRITE_LINE2 =>
194          this_op   <= default_op;
195          this_op.data <= line2_buffer(ptr*8 + 7 downto ptr*8);
196          next_ptr   <= ptr;
197          next_state  <= WRITE_LINE2;
198          if op_state = DONE then
199           next_ptr <= ptr - 1;
200           if ptr = 0 then
201            next_state <= SELECT_LINE1;
202           end if;
203          end if;
204
205       end case;
206      end process proc_state;
207
208      reg_state : process(clk)
209      begin
210       if rising_edge(clk) then
211        if rst = '1' then
212         state <= RESET;
213         ptr  <= 0;
214        else
215         state <= next_state;
216         ptr  <= next_ptr;
217        end if;
218       end if;
219      end process reg_state;
220
221      -- we never read from the lcd
222      lcd_rw <= '0';
223
224      proc_op_state : process(op_state, cnt, this_op) is
225      begin
226       case op_state is
227        when IDLE =>
228         lcd_db    <= (others => '0');
229         lcd_rs    <= '0';
230         lcd_e     <= '0';
231         next_op_state <= WAIT_SETUP_H;
232         next_cnt   <= DELAY_SETUP_HOLD;
233
```

```
234         when WAIT_SETUP_H =>
235          lcd_db <= this_op.data(7 downto 4);
236          lcd_rs <= this_op.rs;
237          lcd_e <= '0';
238          if cnt = 0 then
239           next_op_state <= ENABLE_H;
240           next_cnt   <= DELAY_LCD_E;
241          else
242           next_op_state <= WAIT_SETUP_H;
243           next_cnt   <= cnt - 1;
244          end if;
245
246         when ENABLE_H =>
247          lcd_db <= this_op.data(7 downto 4);
248          lcd_rs <= this_op.rs;
249          lcd_e <= '1';
250          if cnt = 0 then
251           next_op_state <= WAIT_HOLD_H;
252           next_cnt   <= DELAY_SETUP_HOLD;
253          else
254           next_op_state <= ENABLE_H;
255           next_cnt   <= cnt - 1;
256          end if;
257
258         when WAIT_HOLD_H =>
259          lcd_db <= this_op.data(7 downto 4);
260          lcd_rs <= this_op.rs;
261          lcd_e <= '0';
262          if cnt = 0 then
263           next_op_state <= WAIT_DELAY_H;
264           next_cnt   <= this_op.delay_h;
265          else
266           next_op_state <= WAIT_HOLD_H;
267           next_cnt   <= cnt - 1;
268          end if;
269
270         when WAIT_DELAY_H =>
271          lcd_db <= (others => '0');
272          lcd_rs <= '0';
273          lcd_e <= '0';
274          if cnt = 0 then
275           next_op_state <= WAIT_SETUP_L;
276           next_cnt   <= DELAY_SETUP_HOLD;
277          else
278           next_op_state <= WAIT_DELAY_H;
279           next_cnt   <= cnt - 1;
280          end if;
281
282         when WAIT_SETUP_L =>
283          lcd_db <= this_op.data(3 downto 0);
284          lcd_rs <= this_op.rs;
285          lcd_e <= '0';
286          if cnt = 0 then
287           next_op_state <= ENABLE_L;
288           next_cnt   <= DELAY_LCD_E;
289          else
290           next_op_state <= WAIT_SETUP_L;
```

```
291           next_cnt   <= cnt - 1;
292          end if;
293
294         when ENABLE_L =>
295          lcd_db <= this_op.data(3 downto 0);
296          lcd_rs <= this_op.rs;
297          lcd_e <= '1';
298          if cnt = 0 then
299           next_op_state <= WAIT_HOLD_L;
300           next_cnt   <= DELAY_SETUP_HOLD;
301          else
302           next_op_state <= ENABLE_L;
303           next_cnt   <= cnt - 1;
304          end if;
305
306         when WAIT_HOLD_L =>
307          lcd_db <= this_op.data(3 downto 0);
308          lcd_rs <= this_op.rs;
309          lcd_e <= '0';
310          if cnt = 0 then
311           next_op_state <= WAIT_DELAY_L;
312           next_cnt   <= this_op.delay_l;
313          else
314           next_op_state <= WAIT_HOLD_L;
315           next_cnt   <= cnt - 1;
316          end if;
317
318         when WAIT_DELAY_L =>
319          lcd_db <= (others => '0');
320          lcd_rs <= '0';
321          lcd_e <= '0';
322          if cnt = 0 then
323           next_op_state <= DONE;
324           next_cnt   <= 0;
325          else
326           next_op_state <= WAIT_DELAY_L;
327           next_cnt   <= cnt - 1;
328          end if;
329
330         when DONE =>
331          lcd_db    <= (others => '0');
332          lcd_rs    <= '0';
333          lcd_e     <= '0';
334          next_op_state <= IDLE;
335          next_cnt   <= 0;
336
337       end case;
338      end process proc_op_state;
339
340      reg_op_state : process (clk) is
341      begin
342       if rising_edge(clk) then
343        if state = RESET then
344         op_state <= IDLE;
345        else
346         op_state <= next_op_state;
347         cnt   <= next_cnt;
```

```
348         end if;
349       end if;
350      end process reg_op_state;
351
352     end architecture rtl;
```

El código fue modificado para uso de este manual. El código original puede ser consultado en la página web opencores.org en el siguiente enlace: https://opencores.org/

C.5 ARCHIVO DE BIBLIOTECA “instrument.vhd”

La tabla C1 define funciones que permiten utilizar el FPGA para instrumentación, entre las funciones destaca la función *to_string()* la cual permite realizar conversiones de algunos tipos de datos a cadenas para poder ser procesados en una pantalla LCD y cuenta con varias definiciones, cabe mencionar que es de elaboración propia.

Tabla C1 Funciones de la biblioteca “Instrument.vhd” y su funcionamiento.

Función	Argumentos	Devuelve	Descripción
to_string(x)	x: Es un tipo de dato STD_LOGIC	Una cadena de un solo carácter	Convierte un tipo de dato STD_LOGIC en una cadena de un solo carácter que puede ser uno o cero.
to_string(x)	x: Es un tipo de dato STD_LOGIC_VECTOR	Una cadena de la misma longitud que x	Realiza la conversión de un dato tipo STD_LOGIC_VECTOR a una cadena de caracteres de ceros y unos.
to_string(x,n)	x: Número entero. n: Número de caracteres de la cadena.	Una cadena con n número de caracteres	Convierte un dato tipo entero x con signo a una cadena con n caracteres donde uno se reserva para el signo de dicho número.

to_string(x,n,p)	x: Número entero. n: Número de caracteres de la cadena. p: posición del punto decimal.	Una cadena con n número de caracteres	Convierte un dato tipo entero x con signo a una cadena con n caracteres donde uno se reserva para el signo y se añade un punto flotante ubicado en la posición p.
to_std_logic_vector(x)	x: Cadena de n caracteres	Un vector de 8*n bits.	Realiza la conversión de una cadena donde cada carácter se representa en su código ASCCI de 8 bits dentro de un STD_LOGIC_VECTOR.
mapping(x,x1,x2,y1,y2)	x: Variable tipo entera. x1: Límite inferior del rango x2 – x1. x2: Límite superior del rango x2 – x1. y1: Límite inferior del rango y2 – y1. y2: Límite superior del rango y2 – y1.	Un numero entero situado el rango y2 – y1.	Realiza una operación de mapeo donde un valor x perteneciente al rango x2 – x1 tendrá un valor proporcional correspondiente en el rango y2 – y1.

CARPETA VIRTUAL

En este anexo, se indica el enlace para acceder a la carpeta virtual en **mega.nz,** donde se encuentra cada uno de los proyectos de las once prácticas de este manual elaborados en Quartus II 13.0. También se encuentran los bloques, con su respectivo código VHDL, de los archivos de biblioteca de la sección C del apéndice B.

En el siguiente enlace se accede a la carpeta virtual:

https://mega.nz/folder/y7pUFLIQ#-QkMlxIXLwPlcS1VJtZw_Q

REFERENCIAS

A. Pedroni, V. (2004). *Circuit Design with VHDL.* The MIT Press.

S. Parab, J., S. Gad, R., & Naik, G. (2018). *Hands-on Experience with Altera FPGA Development Boards.* New Delhi: Springer (India).