UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Diseño de Prácticas de Laboratorio

para el área de Sistemas Digitales

TESIS

Que para obtener el título de

Ingeniero Eléctrico Electrónico

Presentan:

Cuevas Velázquez Arturo

Godínez Salinas Bernardo Javier

Silva Simón Allan David

Y para obtener el título de

Ingeniero Mecánico Electricista Área Eléctrica Electrónica

Presenta:

Sánchez Capdeville Ricardo

Director de tesis: M.I. Jorge Valeriano Assem

MÉXICO, CIUDAD UNIVERSITARIA 2012

ÍNDICE

Página

PRÓLOGO. ***i***

CAPÍTULO 1. INTRODUCCIÓN. **1**

CAPÍTULO 2. MARCO TEÓRICO. **5**

2.1 Descripción de Dispositivos Lógicos Programables. **5**

2.1.2 Simbología adoptada en los PLD's **5**

2.1.3 Arreglo Lógico Programable (Programmable Logic Array) o PLA **7**

2.1.4 Lógica de Arreglo Programable (Programmable Array Logic) o PAL **7**

2.1.5 Lógica de Arreglo Genérico (Generic Array Logic) o GAL **8**

2.1.6 Dispositivo Lógico Programable Complejo (Complex PLD) o CPLD **8**

2.1.7 Arreglo de Compuertas Programables en Campo (Field Programmable Gate Array) o FPGA **9**

2.1.8 Arquitectura básica de un FPGA **10**

2.1.9 Ejemplo de un CPLD de Altera de la familia MAX 7000 **11**

2.1.10 Ejemplo de un FPGA de Xilinx de la familia Spartan 3 **16**

2.2 Lenguaje de descripción de hardware. **21**

2.2.1 Características principales de los LDH's **22**

2.2.2 VHDL: Organización y Arquitectura **23**

2.2.3 Operadores **26**

2.2.4 Sentencias de descripción **27**

2.2.5 Descripción Estructural **28**

2.2.6 Descripción por comportamiento (behavioural) **29**

2.2.7 Ejemplos **29**

2.3 Herramientas de desarrollo. **34**

2.3.1 Ambientes integrados que permiten la creación de diseños para **34**

dispositivos CPLD’s y/o FPGA’s

2.3.2 Altera – Max Plus II **34**

2.3.3 Altera – Quartus II **36**

2.3.4 Xilinx-ISE (Integrated Software Environment) **38**

CAPÍTULO 3. Descarga e Instalación Paquete ISE Design Suite 13. 44

3.1 Descarga de Paquete ISE DEsign Suite 13.1 **44**

3.2 Instalación ISE Design Suite 13.1 **46**

3.3 Instalación ADEPT 2 **53**

Práctica 1: Uso del ambiente de desarrollo IDE, para el modelado. **56**

Práctica 2: Uso del ambiente de desarrollo IDE, para simulación. **65**

Práctica 3: Uso de la tarjeta de desarrollo basada en FPGA (BASYS 2). **71**

Práctica 4: Modelado, simulación e implementación de funciones booleanas. **81**

Práctica 5: Modelado e implementación de codificadores y decodificadores. **94**

Práctica 6: Modelado e implementación de mutiplexores y demultiplexores. **110**

Práctica 7: Modelado e implementación de comparadores. **129**

Práctica 8: Modelado e implementación de una ALU. **138**

Práctica 9: Modelado e implementación de Latches y FF: T, D, SR y JK. **150**

Práctica 10: Modelado e implementación de multiplexor de display. **171**

Práctica 11: Modelado e implementación de un contador decimal 0000-9999. **180**

Práctica 12: Modelado e implementación de control de un par de semáforos. **191**

CAPÍTULO 4. Resultados y conclusiones.

BIBLIOGRAFÍA

PRÓLOGO

Hoy en día, en nuestro ambiente familiar o de trabajo nos encontramos rodeados de sistemas electrónicos muy sofisticados: teléfonos celulares, computadoras personales, televisores portátiles, equipos de sonido, dispositivos de comunicaciones y estaciones de juego interactivos, entre otros; no son más que algunos ejemplos del desarrollo tecnológico que ha cambiado nuestro estilo de vida haciéndolo cada vez más confortable. Todos estos sistemas tienen algo en común: su tamaño, de dimensiones tan pequeñas que resulta increíble pensar que sean igual o más potentes que los sistemas mucho más grandes que existieron hace algunos años.

Estos avances son posibles gracias al desarrollo de la microelectrónica, la cual ha permitido la miniaturización de los componentes para obtener así mayores beneficios de los chips (circuitos integrados) y para ampliar las posibilidades de aplicación.

La evolución en el desarrollo de los circuitos integrados se ha venido perfeccionando a través de los años. Primero, se desarrollaron los circuitos de baja escala de integración (SSI o Small Scale Integration), y posteriormente los de larga escala de integración (LSI o Largue Scale Integration), para continuar con los de muy alta escala de integración (VLSI o Very Large Scale Integration) hasta llegar a los circuitos integrados de propósito específico (ASIC).

Actualmente, la gente encargada del desarrollo de nueva tecnología perfecciona el diseño de los circuitos integrados orientados a una aplicación y/o solución específica: los ASIC, logrando dispositivos muy potentes y que ocupan un mínimo de espacio. La optimización en el diseño de estos chips tienen dos tendencias en su conceptualización.

La primera tendencia es la técnica de full custom design (Diseño totalmente a la medida), la cual consiste en desarrollar un circuito para una aplicación específica mediante la integración de transistor por transistor. En su fabricación se siguen los pasos tradicionales de diseño: preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química.

La segunda tendencia en el diseño de los ASIC proviene de una innovadora propuesta, que sugiere la utilización de celdas programables preestablecidas e insertadas dentro de un circuito integrado. Con base en esta idea surgió la familia de Dispositivos Lógicos Programables (Programmable Logic Device) o PLD's, cuyo nivel de densidad de integración ha venido evolucionando a través del tiempo. Iniciaron con los Arreglos Lógicos Programables (Programmable Array Logic) o PAL's, hasta llegar al uso de los Dispositivos Lógicos Programables Complejos (Complex Programmable Logic Device) o CPLD's y los Arreglos de Compuertas Programables en Campo (Field Programmable Gate Array) o FPGA's, los cuales dada su conectividad interna sobre cada una de sus celdas han hecho posible el desarrollo de circuitos integrados de aplicación específica de una forma mucho más fácil y económica, para beneficio de los ingenieros encargados de integrar sistemas.

El contenido de este trabajo de tesis se encuentra orientado hacia este tipo de diseño, donde el objetivo principal es brindar a los estudiantes que cursen la materia de Sistemas Digitales, la oportunidad de comprender, manejar y aplicar el lenguaje de programación más poderoso para este tipo de aplicaciones: VHDL.

El lenguaje de descripción en hardware VHDL es considerado como la máxima herramienta de diseño por las industrias y universidades de todo el mundo, pues proporciona a los usuarios muchas ventajas en la planeación y diseño de los sistemas electrónicos digitales.

Este trabajo de tesis ha sido preparado especialmente para aquellos estudiantes e ingenieros que desean introducirse en el manejo de este lenguaje de programación, proporcionando una forma fácil y práctica de integrar aplicaciones digitales utilizando el lenguaje de descripción en hardware VHDL. También esperamos motivar al estudiante para que comience el desarrollo e integración de sistemas electrónicos a través de este lenguaje, con la visión y oportunidad de crecer como microempresario en el desarrollo de sistemas miniaturizados, los cuales pueden ser fácilmente comercializados, y generar así fuentes de empleo en beneficio de la sociedad.

Este trabajo de tesis es recomendable para practicar la teoría tomada en el curso de Sistemas Digitales, tanto para nivel técnico como a nivel universitario dado que para interpretar y entender las prácticas sólo requiere como antecedente un curso básico de diseño lógico que involucre el conocimiento de los temas de compuertas lógicas, minimización de funciones booleanas, circuitos combinacionales y circuitos secuenciales.

CAPÍTULO 1

INTRODUCCIÓN

Introducción

Actualmente en las áreas de control, automatización e incluso para áreas como la de tecnologías de la información, es necesario hacer uso de dispositivos como los CPLD's y los FPGA's que faciliten la implementación de determinadas soluciones por medio de un ambiente integrado.

El conocimiento sobre el manejo y funcionamiento de estas herramientas optimiza el tiempo que invierte el alumno en sus proyectos, el porcentaje de error en la implementación es casi nulo y aumenta el alcance del desarrollo de soluciones, es decir aumentará la facilidad de desarrollar proyectos más complejos.

Es por esta razón que ha surgido la necesidad de elaborar un documento que pueda proporcionar información teórica y práctica principalmente al alumno de Ingeniería del área de sistemas digitales, sobre la programación e implementación de soluciones a través de los CPLD's y FPGA's, aunque el documento podrá ser comprendido por quienes tengan conocimientos básicos de electrónica.

En la parte teórica se pretende introducir al alumno a la arquitectura, lenguajes de descripción de hardware y el uso de ambientes integrados para la programación de los CPLD's y FPGA's.

A través del manejo de un ambiente integrado se desarrollaran e implementaran las 12 prácticas que se han diseñado para este trabajo las cuales se implementarán en la tarjeta de desarrollo BASYS 2 (ver figura 1.1) que está compuesta por el FPGA Spartan 3E-100 CP132.

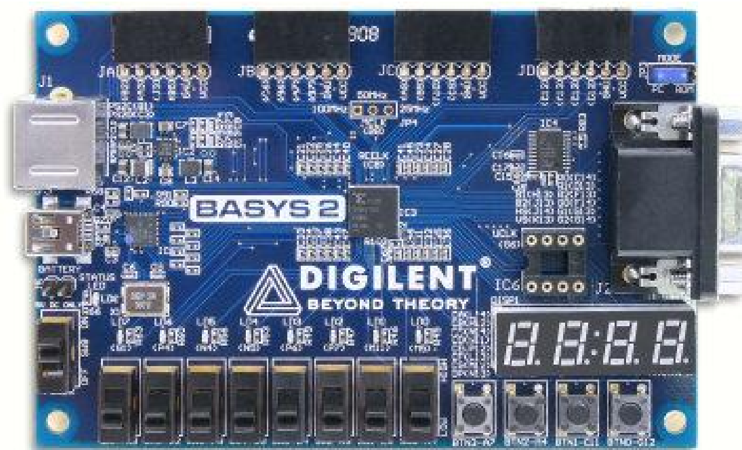


Figura 1.1 Tarjeta de desarrollo BASYS 2

Desarrollo de prácticas

- **Manejo de las herramientas**

Práctica 1: Uso del ambiente de desarrollo IDE, para el modelado.
Práctica 2: Uso del ambiente de desarrollo IDE, para simulación.
Práctica 3: Uso de la tarjeta de desarrollo basada en FPGA (BASYS 2).

- **Álgebra Booleana y compuertas lógicas**

Práctica 4: Modelado, simulación e implementación de funciones booleanas.

- **Circuitos combinacionales**

Práctica 5: Modelado e implementación de codificadores y decodificadores.
Práctica 6: Modelado e implementación de multiplexores y demultiplexores.
Práctica 7: Modelado e Implementación de comparadores.
Práctica 8: Modelado e implementación de una ALU.

- **Circuitos secuenciales**

Práctica 9: Modelado e implementación de Latches y FF: T, D, SR y JK.
Práctica 10: Modelado e implementación multiplexor de display.
Práctica 11: Modelado e implementación de un contador decimal 0000-9999.
Práctica 12: Modelado e implementación de control de un par de semáforos.

Finalmente se obtendrá un documento que se espera que el alumno del área de sistemas digitales pueda considerar como una guía y/o complemento sobre la programación e implementación de soluciones a través del manejo de un ambiente integrado en CPLD’s y FPGA’s.

CAPÍTULO 2

MARCO TÉORICO

2.1 Descripción de Dispositivos Lógicos Programables

Los **Programmable Logic Device (Dispositivo Lógico Programable) o PLD's** son circuitos integrados que ofrecen a los diseñadores en un solo chip, un arreglo de compuertas lógicas y flip-flop's (de aquí en adelante se utilizará ff's o ff), que pueden ser programados por el usuario para implementar funciones lógicas; y así, una manera más sencilla de reemplazar varios circuitos integrados estándares o de funciones fijas.

La mayoría de los PLD's están compuestos de arreglos de compuertas lógicas, uno de ellos a base de compuertas AND al que se le denomina Plano AND y el otro de compuertas OR, denominado Plano OR; éstos pueden ser programables y dependiendo del plano o los planos que lo sean, será la clasificación que reciba el PLD. Con estos recursos se implementan funciones lógicas deseadas mediante un software especial y un programador de dispositivos.

Las variables de entrada (que vienen de las terminales externas del dispositivo) tienen interconexiones hacia uno de los planos, a través de compuertas con salidas complementarias (es decir con una salida inversora y una no-inversora); y salidas de los planos, conectadas a las terminales externas del dispositivo, por elementos lógicos como pueden ser: inversores, compuertas OR y ff's; además, en algunos casos existe retroalimentación de las salidas hacia uno de los planos, para tomarlas como entradas nuevamente (aplicación utilizada frecuentemente en el caso de lógica secuencial).

La programación se lleva a cabo por medio de conexiones fusibles; de tal forma que en una compuerta OR, una entrada con conexión fusible "Fundida o Quemada" (fusible abierto) funcione como un cero lógico y una conexión intacta como el valor de la(s) variable(s) de entrada.

2.1.2 Simbología adoptada en los PLD's

Los fabricantes han sustituido el símbolo del inversor y del no-inversor en uno solo; pero con dos salidas complementadas. Han simplificado las líneas de entrada a una compuerta AND u OR, por medio de una sola línea.

Las conexiones entre compuertas se representan mediante una "X" o un punto. Las "X" se encuentran en el Plano programable y describen una conexión fusible intacta.
En el Plano fijo, un punto representa una conexión fija y que por supuesto, ya no puede cambiarse. La ausencia de estos dos símbolos en un cruce de líneas significa que no existe conexión entre ellas (ver figura 2.1.1).

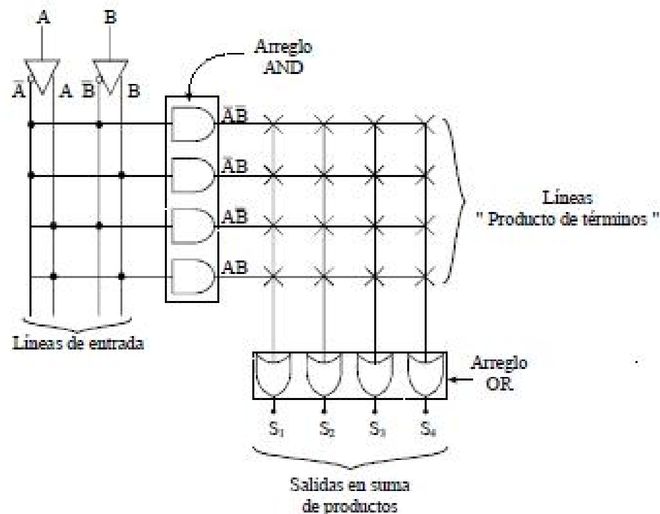


Figura 2.1.1 Conexiones entre compuertas

Los diferentes tipos de dispositivos de lógica programable que existen hoy en día pueden clasificarse por su tecnología (ver figura 2.1.2):

- Dispositivo Lógico Programable Simple **(Simple Programmable Logic Device)** o **SPLD.**
- Dispositivo Lógico Programable Complejo **(Complex Programmable Logic Device)** o **CPLD.**
- Arreglo de Compuertas Programables en Campo **(Field Programmable Gate Array)** o **FPGA.**

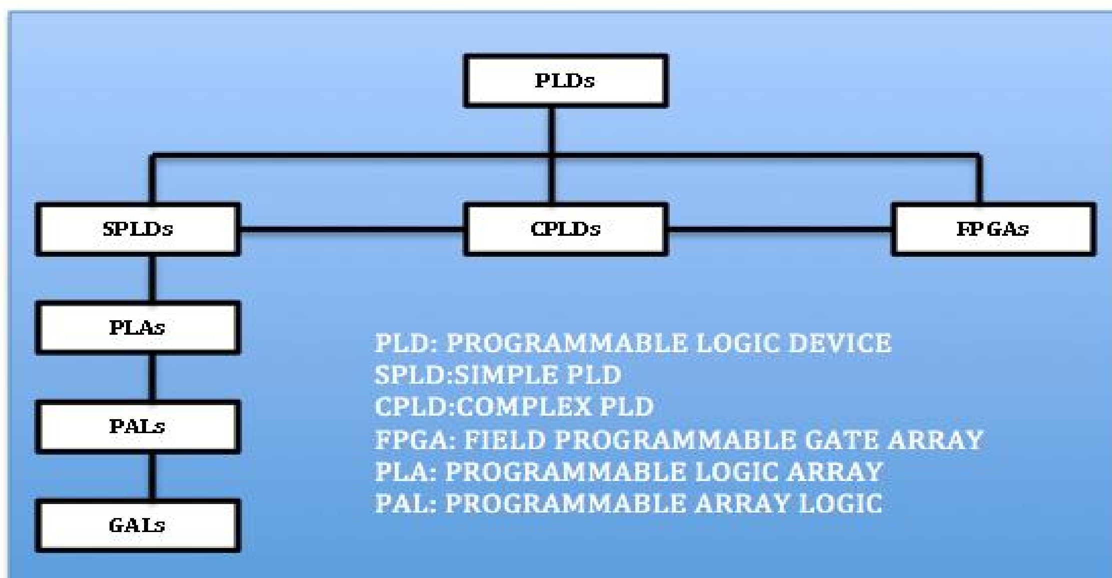


Figura 2.1.2 Clasificación PLD's

2.1.3 Arreglo Lógico Programable (Programmable Logic Array) o PLA

Los PLA's consisten de dos planos programables, AND y OR. Tanto en las entradas como en las salidas tienen compuertas NOT para obtener mayor versatilidad (ver figura 2.1.3).

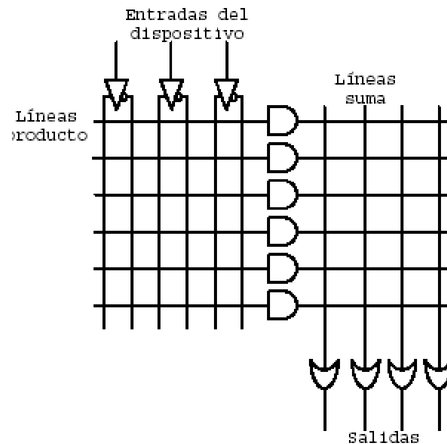


Figura 2.1.3 PLA (AND Programable - OR Programable)

Las salidas son entonces una suma de productos. Las PLA's pueden ser conectadas externamente a ff's para formar máquinas de estado.

2.1.4 Lógica de Arreglo Programable (Programmable Array Logic) o PAL

Los PAL's son una variante de las PLA's y consisten en dos planos; un plano programable AND y un plano fijo OR (ver figura 2.1.4). En las salidas también tenemos sumas de productos.
Ley de Morgan que fundamenta el correcto funcionamiento de las PAL's es:

a | b = !(!a & !b)

Al incluir inversores, se reduce el arreglo OR y se ahorra área para ubicar más lógica.

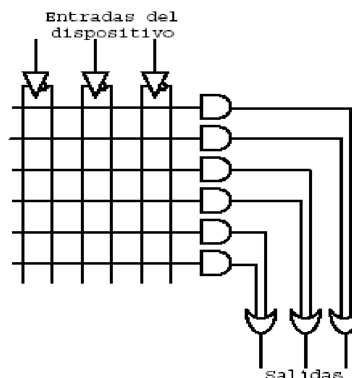


Figura 2.1.4 PAL (AND Programable - OR Fija)

2.1.5 Lógica de Arreglo Genérico (Generic Array Logic) o GAL

La GAL es un PLD E^2CMOS, la cual es básicamente un PLA pero contiene a la salida ff´s y compuertas XOR (Macrocelda) para cambiar el estado lógico de la salida, además de retroalimentar las salidas de los ff.

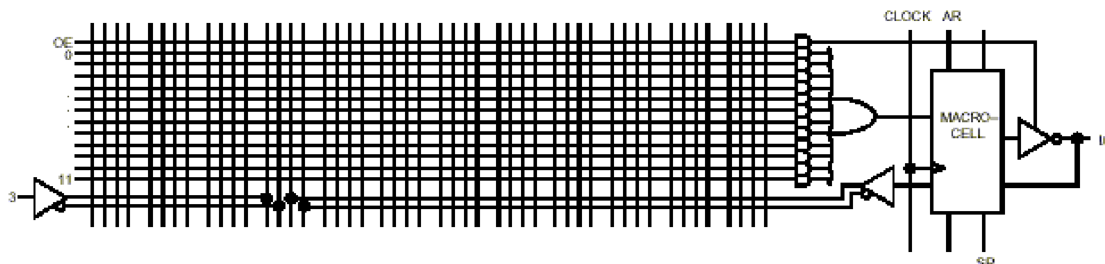


Figura 2.1.4 GAL

Limitaciones de los SPLD

- Reducida cantidad de macroceldas.
- La exigencia de optar entre la retroalimentación desde la macrocelda o desde la entrada forza que ante la necesidad de un flip-flop o de un término lógico intermedio a veces se deba perder una posible terminal de entrada/salida.
- La distribución de todas las señales por todo el chip consume mucha superficie del silicio y genera retardos capacitivos de importancia.
- En los primeros PAL, el uso de fusibles afectaba seriamente la confiabilidad del dispositivo.

2.1.6 Dispositivo Lógico Programable Complejo (Complex PLD) o CPLD

Un CPLD extiende el concepto de un PLD a un mayor nivel de integración ya que permite implementar sistemas más eficientes por que utiliza menos espacio, mejoran la confiabilidad en el circuito y reducen costos. Un CPLD se forma con múltiples bloques lógicos, cada uno similar a un PLD. Los bloques lógicos se comunican entre sí utilizando una matriz programable de interconexiones lo cual hace más eficiente el uso del silicio y conduce a un mejor desempeño
La arquitectura de un CPLD se compone de tres bloques principales (ver figura 2.1.5):

- **Bloque Lógico (Logic Block) o LB**: Un Bloque Lógico es muy similar a un PLD, cada uno de ellos poseen generalmente una matriz de compuertas AND, una matriz de compuertas OR y una configuración para la distribución de los productos en las diferentes macroceldas del bloque.
  El tamaño de Bloque lógico es una medida de la capacidad del CPLD, ya que de esto depende el tamaño de la función booleana que pueda ser implementada dentro del bloque. Los Bloques Lógicos usualmente tienen de cuatro a veinte macroceldas. La cantidad de bloques lógicos que puede poseer un CPLD depende de la familia y fabricante del dispositivo.

- **Matriz de Interconexión Programable (Programmable Interconect Matrix) o PIM**: Permite unir los pines de entrada/salida a las entradas del bloque lógico, o las salidas del bloque lógico a las entradas del bloque lógico, inclusive a las entradas del mismo bloque. La mayoría de los CPLD's usan una de dos configuraciones para esta matriz: interconexión mediante arreglo o interconexión mediante multiplexores. La primera se basa en una matriz de filas y columnas con una celda EECMOS en cada intersección. Al igual que en la GAL esta celda puede ser activada para conectar/desconectar la correspondiente fila o columna. Esta configuración permite una total interconexión entre las salidas y entradas de los bloques lógicos. En la interconexión con multiplexores, existe un multiplexor por cada entrada al bloque lógico. Las vías de interconexión

programables son conectadas a las entradas de un número fijo de multiplexores por cada bloque lógico. Las entradas de selección de estos multiplexores son programadas para permitir que sea seleccionada únicamente una vía de la matriz de interconexiones por cada multiplexor, la cual se propaga hacia el bloque lógico

- **Bloque de Entrada/Salida (Input Output Block) o IOB**: La función del Bloque de entrada/salida es permitir el paso de la señal hacia dentro o hacia el exterior del dispositivo.

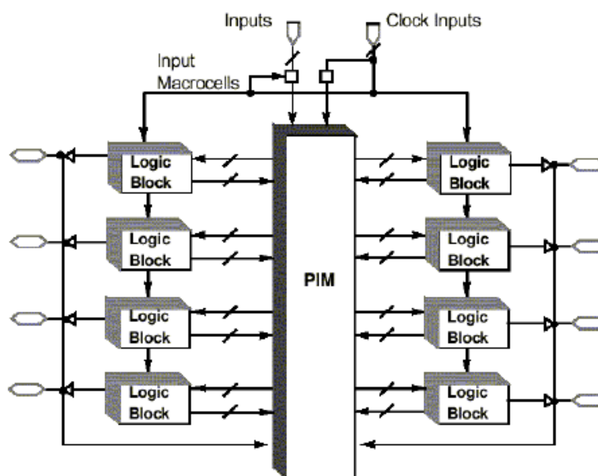


Figura 2.1.5 Arquitectura CPLD

2.1.7 Arreglo de Compuertas Programables en Campo (Field Programmable Gate Array) o FPGA

Las FPGA's (Field Programmable Gate Array) son circuitos de aplicación específica (ASIC) de alta densidad programables por el usuario en un tiempo reducido y sin la necesidad de verificación de sus componentes, se las considera como un derivado de los Gate Array, aunque es menos usado, también se les conoce como LCAs (Logic Cell Array)

Además los FPGAs presentan líneas de interconexión, agrupadas en canales verticales y horizontales. Finalmente, disponen de células de memoria de configuración (CMC, Configuration Memory Cell) distribuidas a lo largo de todo el chip, las cuales almacenan toda la información necesaria para programar los elementos programables mencionados anteriormente. Estas células de configuración suelen consistir en memoria RAM y son inicializadas en el proceso de carga del programa de configuración (ver figura 2.1.6).

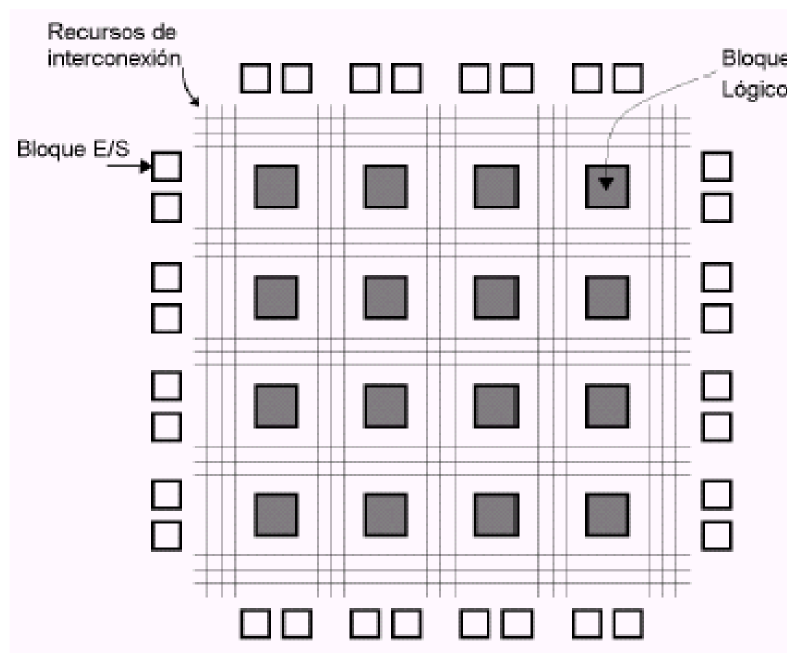


Figura 2.1.6 FPGA

2.1.8 Arquitectura básica de un FPGA

Una FPGA consta de tres tipos de elementos programables:

- Bloques lógicos configurables (CLB, Configurable Logic Block): Constituyen el núcleo de una FPGA. Cada CLB presenta una sección de lógica combinacional programable y registros de almacenamiento. Los registros de almacenamiento sirven como herramientas en la creación de lógica secuencial. La sección de lógica combinacional suele consistir en una LUT (Look Up Table), que permite implementar cualquier función booleana a partir de sus variables de entrada. Su contenido se define mediante las células de memoria (CMC). Se presentan también multiplexores, como elementos adicionales de direccionamiento de los datos del CLB, los cuales permiten variar el tipo de salidas combinacionales o registradas), facilitan caminos de realimentación, o permiten cambiar las entradas de los biestables. Se encuentran controlados también por el contenido de las células de memoria (ver figura 2.1.7)

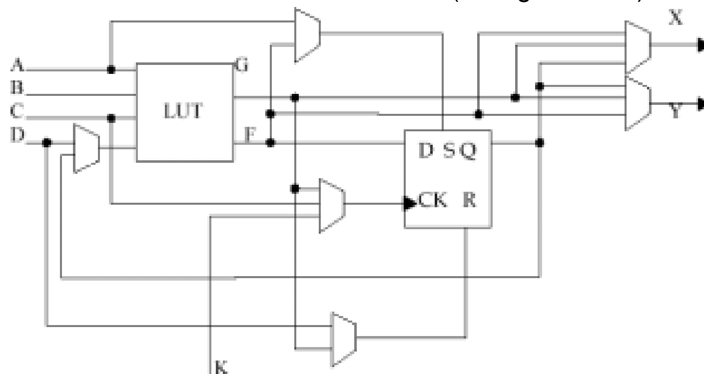


Figura 2.1.7 Bloque lógico configurable

- Matrices de interconexión (SM, Switching Matrix). Suele ser el elemento más limitante en la utilización de este tipo de dispositivos. Este bloque es el que evita que tales dispositivos no sean completos, ya que los bloques de procesado lo son. A medida de que se van sacando nuevos dispositivos, casi siempre va mejorando esta matriz, ya sea en mayor número de conexiones y/o en mejores prestaciones.

- Bloques de entrada/salida (IOB, Input/Output Blocks): La periferia de los FPGA's están constituidas por bloques de entrada/salida configurables por el usuario. Cada bloque puede ser configurado independientemente para funcionar como entrada, salida o bidireccional, admitiendo también la posibilidad de control triestado. Los IOB's pueden configurarse para trabajar con diferentes niveles lógicos (TTL, CMOS). Además, cada IOB incluye flip-flop's que pueden utilizarse para registrar tanto las entradas como las salidas (ver Figura 2.1.8).

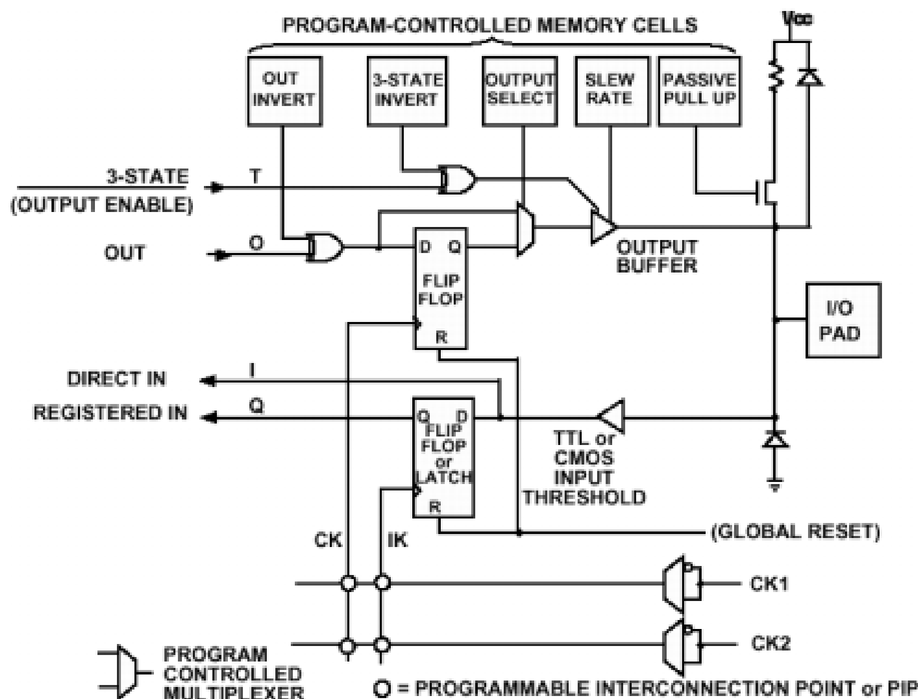


Figura 2.1.8 Estructura típica de un bloque IOB

2.1.9 Ejemplo de un CPLD de Altera de la familia MAX 7000

Descripción General

El MAX 7000 es de la familia de alta densidad. Los PLD's de alto rendimiento, se basa en la segunda generación de arquitectura de Altera MAX. Fabricados con tecnología avanzada CMOS, basada en la EEPROM de la familia MAX 7000 proporciona entre 600 y 5.000 puertas utilizables, ISP, los retrasos pin a pin son de tan sólo 5 ns, y tienen velocidades de hasta 175,4 MHz.

La arquitectura del MAX 7000 incluye los siguientes elementos:

- Logic array blocks.
- Macroceldas.
- Términos de expansión producto (compartible y paralelos).
- Matriz programable de interconexión.
- Bloques de control Entrada/ Salida.

La arquitectura del MAX 7000 incluye cuatro entradas dedicada que puede ser utilizados como insumos de uso general o de alta velocidad, un control general de señales (reloj, claro, y dos de salida permiten señales) para cada macrocelda y pin I / O.

La figura 2.1.9 muestra la arquitectura de EPM7032, EPM7064 y dispositivos EPM7096.

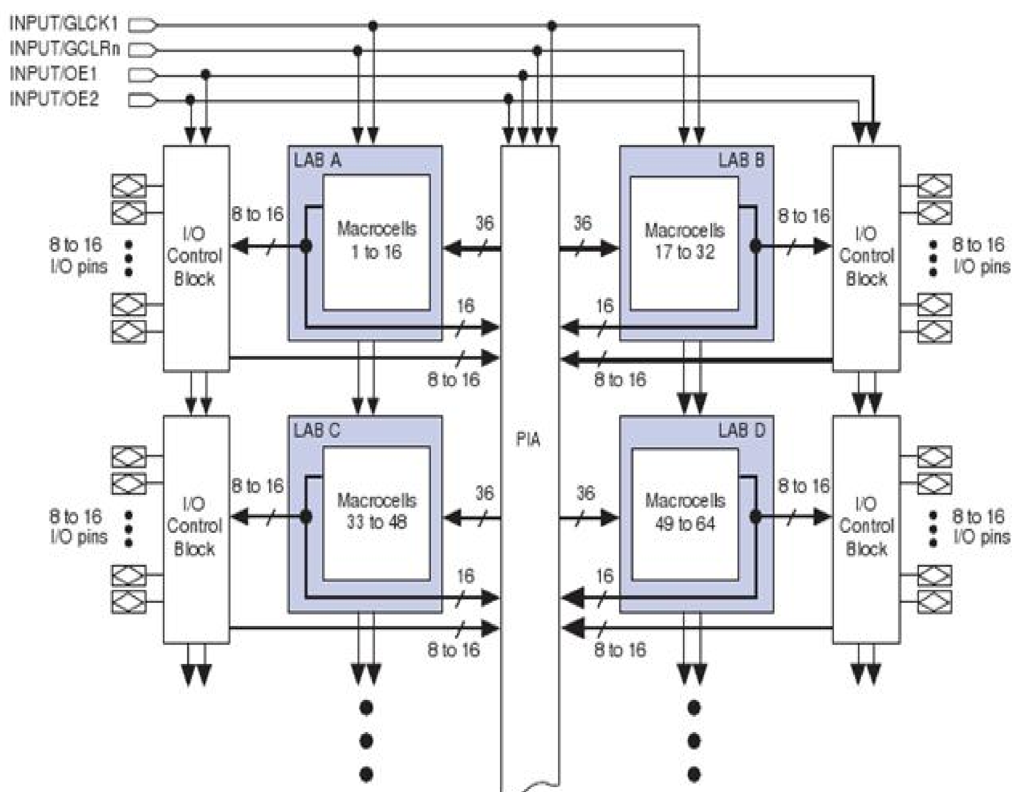


Figura 2.1.9 Diagrama a bloques del dispositivo EPM7032, EPM7064 y EPM7096

Características:

- Alto rendimiento, dispositivos programables EEPROM basado en la lógica (PLD) sobre la base de la segunda generación de la arquitectura MAX ®.
- 5.0-V para la programación del sistema (ISP) a través del built-in IEEE Std.. 1149.1 Interfaz JTAG disponible en dispositivos MAX 7000S.
- Circuitos compatibles con el estándar IEEE Std. ISP. 1532 Incluye 5.0 V en dispositivos MAX 7000 y 5.0 V-ISP-based en dispositivos MAX 7000S.
- Dispositivos con 128 o más macroceldas.
- Completa familia de EPLD con densidades que van de 600 a 5.000 puertas utilizables.
- Retrasos lógicos de 5 ns pin a pin sobre un contador de 175,4 MHz (incluida la interconexión)
- Dispositivos disponibles compatibles con PCI

Logic Array Blocks

La arquitectura del MAX 7000 se basa en la vinculación de alto rendimiento entre bloques de matriz lógica (LAB's). Los LAB's son conjuntos de 16 macroceldas. Varios LAB's están unidos entre sí a través

del programmable interconnect array (PIA), un bús global que se sustenta de todas las entradas dedicadas, pines I / O, y macroceldas

Cada LAB es alimentado por las siguientes señales:

- 36 señales del PIA que se utilizan para las entradas generales.
- Controles Globales que se utilizan para las funciones de registro secundarias.
- Rutas directas de entrada de los pines I / O, para los registros que se utilizan en los tiempos de preparación rápida (Dispositivos MAX 7000E y MAX 7000S).

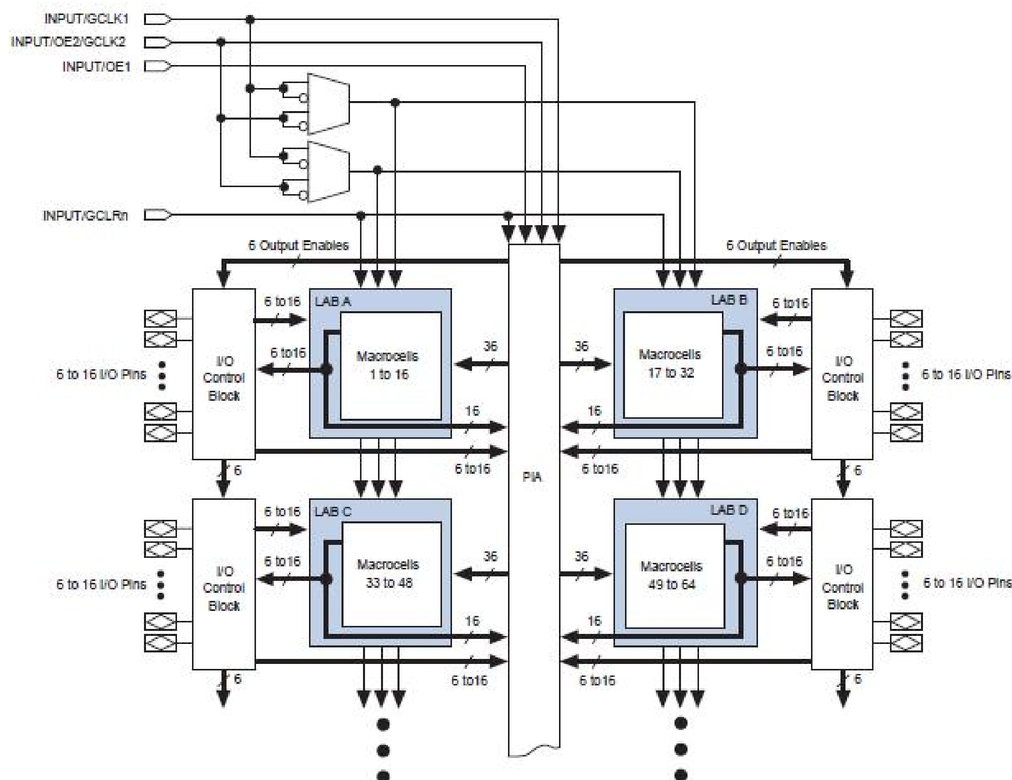


Figura 2.1.10 Diagrama a bloques del dispositivo MAX 7000E & MAX7000S

Macroceldas

Las macroceldas del MAX 7000 pueden configurarse de forma individual, ya sea para lógica secuencial o combinatoria operación. La macrocelda consiste de tres bloques funcionales: la matriz de la lógica, el producto plazo seleccione matriz, y el registro programable. La macrocelda de EPM7032, EPM7064 y dispositivos EPM7096 se muestra en la figura 2.1.11.

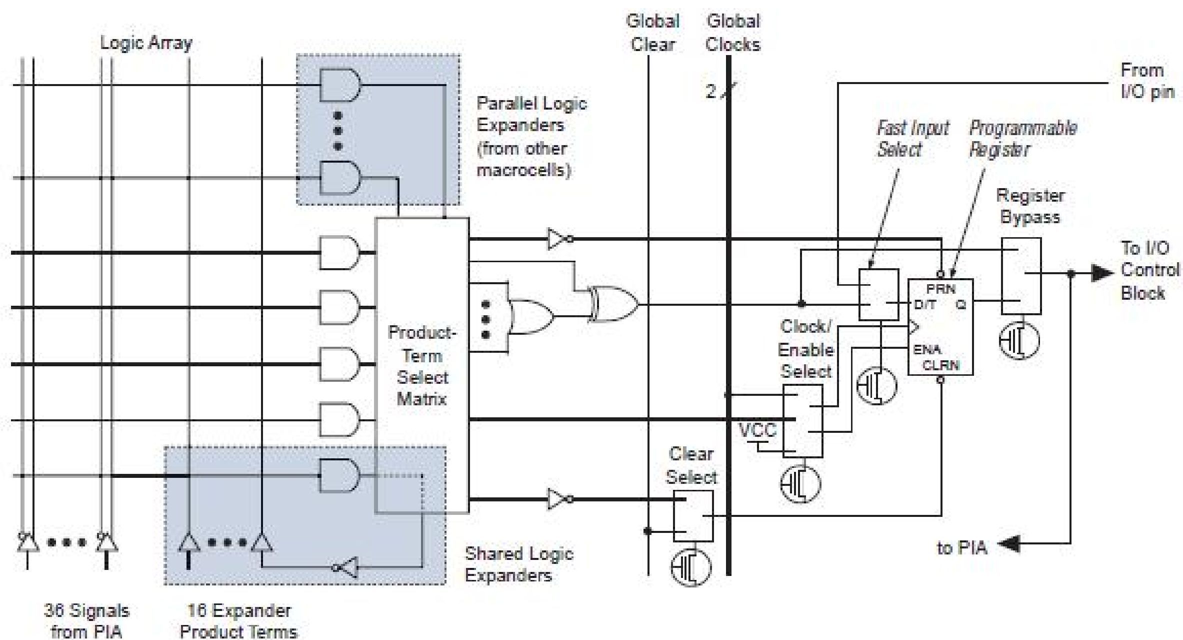


Figura 2.1.11 Macrocelda del dispositivo EPM7032, EPM7064 & EPM7096

Shareable Expanders

Aunque la mayoría de las funciones lógicas se pueden implementar con los cinco productos disponibles en cada macrocelda, una lógica más compleja de funciones requiere condiciones adicionales del producto. Otro macroceldas pueden ser utilizadas para suministrar los recursos necesarios a la lógica, sin embargo, la arquitectura MAX 7000 también permite a ambos compartir términos de expansión de producto ("expansores") que proporcionan otros términos del producto directamente a cualquier macroceldas en el mismo LAB. Estos expansores aseguran que la lógica se sintetiza con la menor cantidad los recursos posibles para obtener la lógica de mayor velocidad posible.

Cada expansor puede ser utilizado y compartido por macroceldas, cualquiera o todos del LAB's pueden crear funciones complejas. Un pequeño retraso (t_{SEXP}) ocurre cuando los expansores se utilizan.

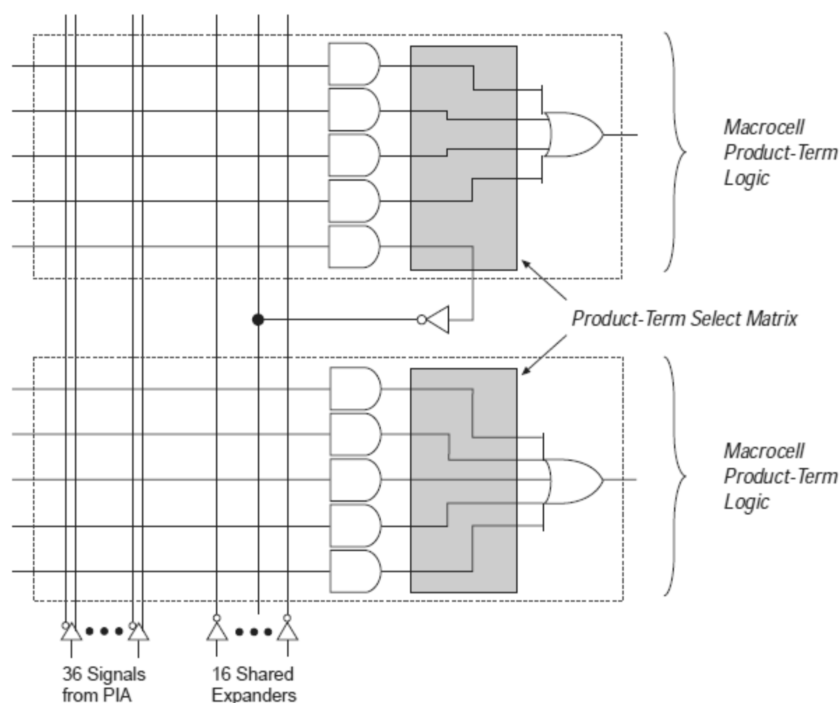


Figura 2.1.12 **Shareable Expanders**

Bloques de Control de Entrada/Salida (I/O)

El bloque de control del I/O permite que cada pin I/O pueda configurarse de forma individual como entrada, salida, o bidireccional. Todos los pines I/O tienen un búfer tri-estado que es controlado individualmente por uno que permite señales al ser directamente conectado con la tierra o VCC. La figura 2.1.13 muestra el bloque de control de I/O para la familia del MAX 7000. El bloque de control I/O del EPM7032, EPM7064 y EPM7096 permite conducir por medio de dos salidas dedicadas, pines activados (OE1 y OE2). Cuando el control de amortiguación de los tres estados se conecta a tierra, la salida es tri-state (alta impedancia) y el pin I/O se puede utilizar como entrada. Cuando el control de amortiguación de los tres estados se conecta a VCC, la salida es deshabilitada.

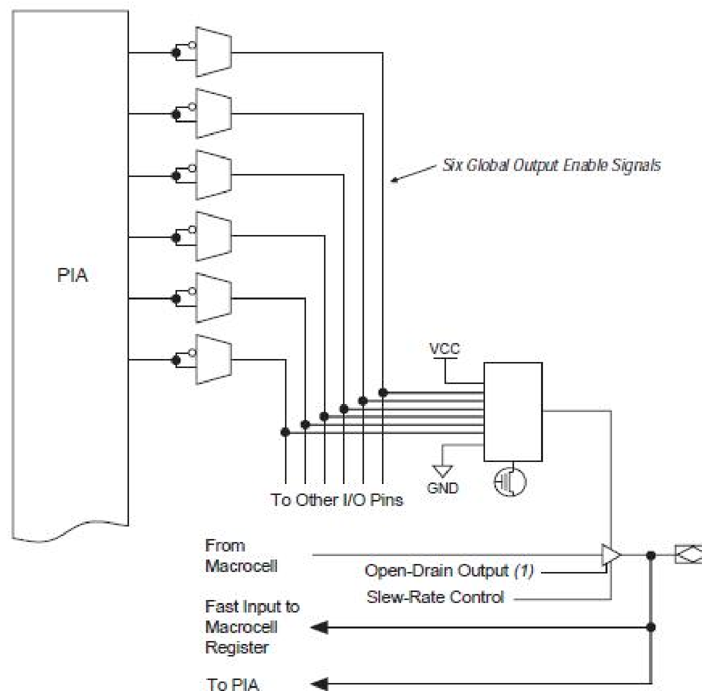


Figura 2.1.13 Bloque de Control I/O para los dispositivos
EPM7032, EPM7064 & EPM7096

2.1.10 Ejemplo de un FPGA de Xilinx de la familia Spartan 3

Arquitectura del FPGA Spartan 3-E

Como se puede observar en la figura 2.1.14, los dispositivos FPGA están conformados principalmente por Bloques Lógicos Configurables (CLB: Configurable Logic Blocks) y por Bloques de Entrada-Salida (IOB: Input/Output Blocks). De manera más precisa, los CLBs son las unidades que ejecutan las operaciones combinacionales, aritméticas y de memoria necesarias para la implementación de la aplicación descrita en el lenguaje de configuración del FPGA. Por su parte, los IOB's son módulos de interconexión entre los pines del circuito integrado FPGA y la lógica interna del dispositivo.

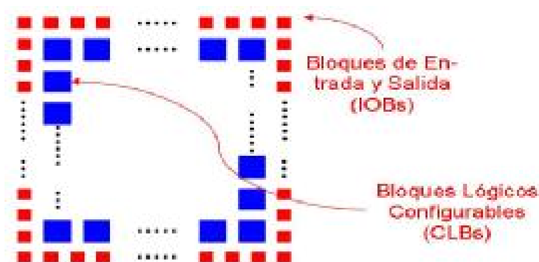


Figura 2.1.14 Estructura básica de un FPGA

Además de estos elementos fundamentales, los FPGA suelen tener otros elementos que dependerán del fabricante y del modelo del integrado. Estos recursos van desde memorias, multiplicadores, hasta arreglos lógicos más complejos como lo son los administradores de reloj. Todos estos recursos añadidos permiten realizar diversas tareas en los distintos diseños basados en FPGA, y al estar dedicados para funciones específicas, permiten implementar aplicaciones con alta eficiencia. Esto finalmente redunda en un mejor aprovechamiento de los recursos de uso global, especialmente en la minimización de la cantidad de CLB's utilizados.

Distribución de la arquitectura del FPGA Spartan 3E

Los elementos antes nombrados se encuentran distribuidos de la manera que se muestra en la figura 2.1.16 para el FPGA Spartan 3E producido por la compañía Xilinx, Inc.

Como se puede observar, los recursos de uso global (CLB's) ocupan el área más extensa del circuito integrado. Sin embargo, algunos recursos dedicados para operaciones específicas también se encuentran presentes. De manera precisa, los recursos adicionales son: los administradores digitales de reloj (DCM's: Digital Clock Managers), las memorias RAM y los multiplicadores dedicados de 18x18 bits con signo.

Para el caso particular del FPGA Spartan 3E, los CLB's se encuentran constituidos por 4 paquetes o divisiones denominados SLICE's (ver figura 2.1.15)

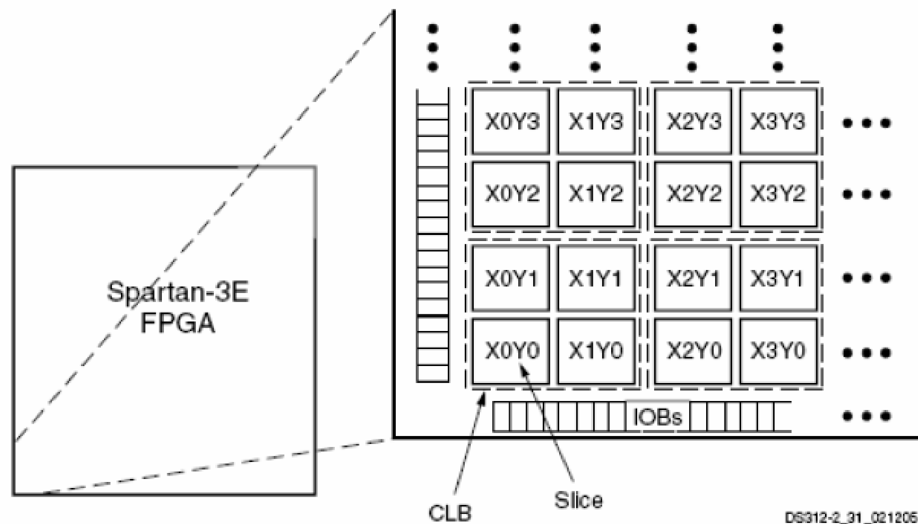


Figura 2.1.15 CLB's constituidos por SLICE's

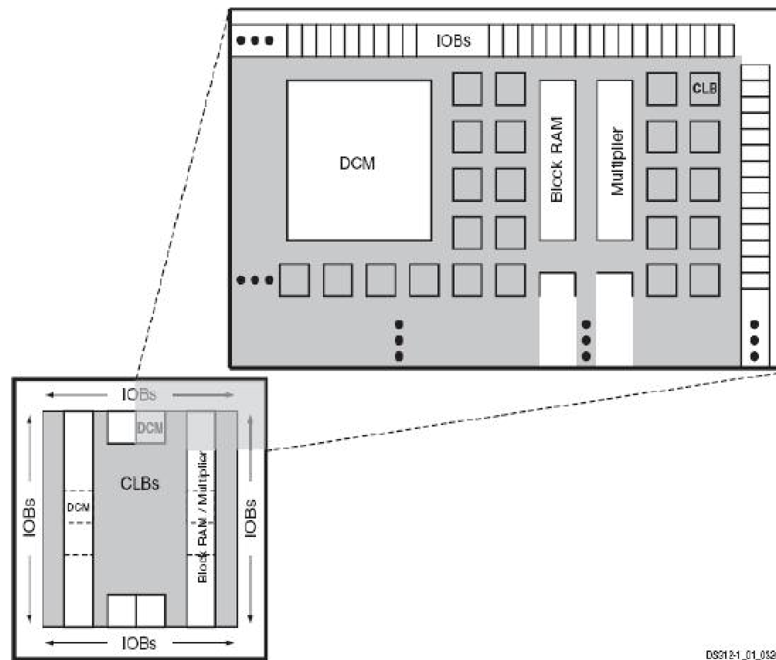
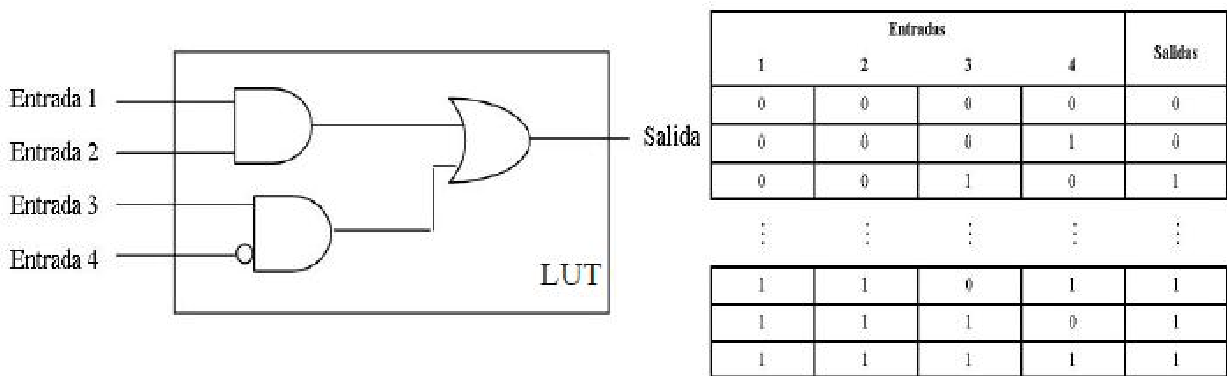


Figura 2.1.16 Arquitectura del FPGA Spartan 3 E

Cada SLICE contiene 2 tablas de verdad o LUT's (del inglés Look Up Table) que permiten la implementación de cualquier función lógica de cuatro entradas y una salida, por lo que las LUT's no tienen limitaciones en cuanto a la complejidad de la función a implementar y su retardo será constante independiente de la función combinacional que realice. Por otro lado, habrá limitación si la función excede las 4 entradas, figura 2.1.17.



Entradas				Salidas
1	2	3	4	
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
⋮	⋮	⋮	⋮	⋮
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figura 2.1.17 Función lógica representada en una LUT

Debido a la existencia de la restricción en el número de entradas de los LUT's, los SLICE's también poseen una lógica aritmética y de acarreo en las salidas de los LUT's, que se activan en aquellos casos en los cuales las funciones lógicas posean más de cuatro entradas. Además de esto, la lógica de acarreo también se encuentra interconectada entre LUT's de varios SLICE's, por lo que se pueden configurar funciones lógicas con una cantidad de entradas aún mayor. Finalmente, cada SLICE posee un registro por cada salida. Estos registros, permiten almacenar el resultado de toda la operación lógica durante un

tiempo equivalente a un ciclo de reloj. A continuación, se presenta un esquema de la distribución interna de cada SLICE.

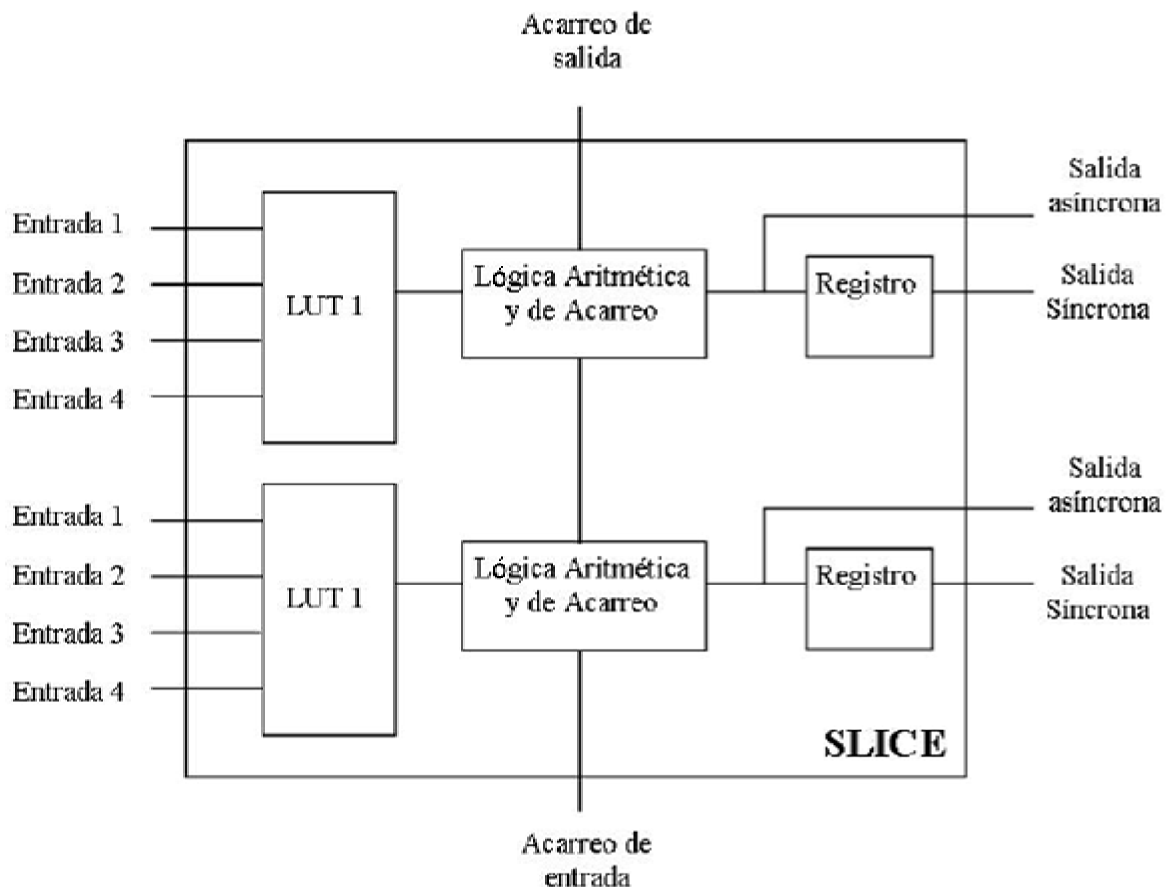
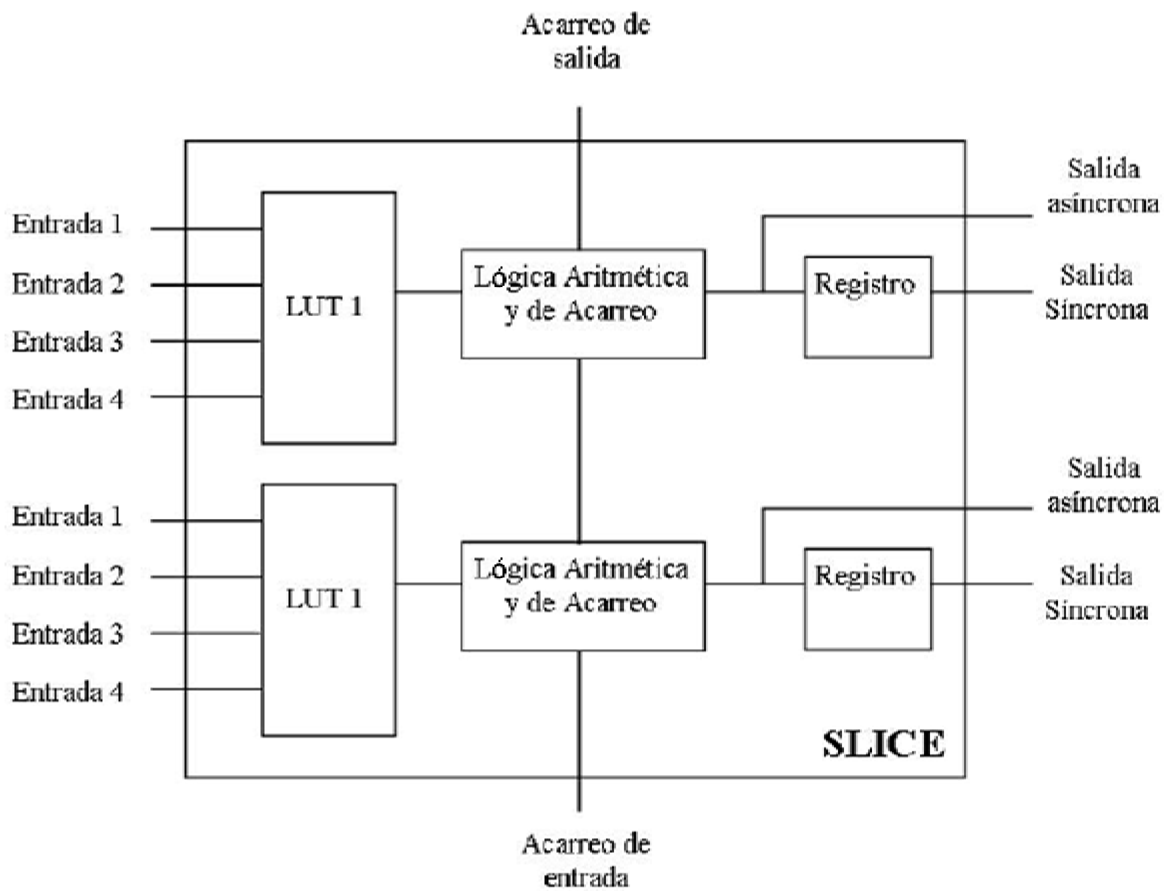


Figura 2.1.17 Distribución interna de cada SLICE

Características de los IOB’s en el FPGA Spartan 3-E

Los bloques de entradas-salidas (IOB’s) son las interfaces entre los pines de los dispositivos FPGA y el circuito lógico configurable interno. Específicamente, en el FPGA Spartan 3E los IOB’s pueden ser configurados de manera unidireccional o bidireccional según los requerimientos del diseño. Además, los IOB’s tienen la ventaja de estar diseñados para trabajar con diversos estándares digitales de tensión diferencial (LVDS, BLVDS, LDT, LVPECL) y de tensión referenciada (LVTTL, LVCMOS, PCI-X, PCI, GTL, GTLP).


Acarreo de salida
Entrada 1
Entrada 2
Entrada 3
Entrada 4
LUT 1
Lógica Aritmética y de Acarreo
Registro
Salida asíncrona
Salida Síncrona
Entrada 1
Entrada 2
Entrada 3
Entrada 4
LUT 1
Lógica Aritmética y de Acarreo
Registro
Salida asíncrona
Salida Síncrona
SLICE
Acarreo de entrada

2.2 Lenguaje de descripción de hardware

Los lenguajes de descripción de hardware (HDL's, Hardware Description Languages), son utilizados para describir la arquitectura y comportamiento de un sistema electrónico.

La descripción de hardware, en sus inicios, fue mediante esquemas, que son representaciones gráficas de los circuitos. Posteriormente se utilizaron instrucciones simples, que permitían la descripción total de un circuito, a este proceso se le denominó Netlist. Básicamente, el Netlist consistía en dar una lista de los componentes y sus interconexiones, mas no describe cómo funciona el circuito, por lo que no es un lenguaje de alto nivel. El Netlist es la primera forma que apareció de describir un circuito mediante un lenguaje; este método resultó ser complejo para diseños con un gran número de componentes.

A partir de esto, surgió el interés por describir circuitos utilizando directamente un lenguaje de alto nivel, en lugar de simples esquemas o instrucciones Netlist. Mientras las herramientas de diseño se volvían más sofisticadas, era más viable la posibilidad de desarrollar circuitos digitales, mediante dispositivos programables. Una opción fue la descripción de circuitos con el uso de un lenguaje de alto nivel de abstracción (un nivel de abstracción define los detalles que se deben considerar al describir un diseño, cuanto menos sean los detalles en una descripción, el nivel de abstracción es más alto), que permitiera no sólo la simulación, sino también la síntesis de circuitos.

Posteriormente, la interrelación creciente entre software y hardware originó la necesidad de lenguajes para hardware que soporten el diseño de sistemas electrónicos completos, que puedan ser interpretados tanto por máquinas como por personas.

El propósito inicial de los HDL's, de plantear modelos, describirlos en un lenguaje, para luego ser utilizados como herramientas de simulación, esto fue acarreando otro enfoque y nuevos propósitos surgieron, especialmente el de síntesis.

En muchos casos, la síntesis no es clara o no tiene correspondencia con el hardware, éste ha sido uno de los principales antecedentes, para incluir mejoras y estar en constante perfeccionamiento de gran parte de los lenguajes de descripción de hardware.

Cuanto más alto es el nivel de abstracción, en forma más simple se puede especificar un diseño para generar lógica compleja. Sin embargo, es útil el conocimiento de las estructuras de hardware a utilizar ya que a veces la síntesis se realiza de modo tal que el diseñador pierde la visión de cómo el compilador implementa lo especificado, con el riesgo de un uso quizás no muy eficiente del hardware que puede hasta comprometer la performance de velocidad, consumo de potencia, etc.

Los HDL's poseen herramientas de síntesis lógica y de alto nivel; entendiéndose por síntesis al proceso, que a partir de la descripción del comportamiento de un circuito se genera automáticamente otro a nivel estructural, capaz de ser acomodado en el dispositivo de destino; es decir, la descripción del circuito (en cualquier modelo de descripción) se convierte en un circuito de compuertas lógicas optimizado para finalmente ser implementado físicamente en una tecnología escogida (PLD's, CPLD's, FPGA's, celdas estándar, etc.), similar a un lenguaje para desarrollo de software, que requiere ser ensamblado a código de máquina.

Además, los HDL's utilizan técnicas de partición y jerarquización para simplificar sistemas complejos; es decir, los HDL's poseen elementos para soportar el manejo de recursos, la experimentación y el manejo del diseño. Así mismo, los HDL's permiten afinar detalles del comportamiento de circuitos mediante su descripción (por ejemplo la descripción de retardos).

Los modelos de hardware usando HDL's pueden ser estructurales, de comportamiento o una mezcla de estos dos. A nivel estructural se describe la interconexión y jerarquía entre componentes. A nivel de comportamiento de hardware se describe la respuesta entrada/salida de un componente. El comportamiento de un sistema puede modelarse a distintos niveles de abstracción o detalle: algoritmos y

comportamiento general, nivel de transferencia de registros, nivel de compuertas, etc. El tipo de modelo más usado para síntesis es el denominado RTL (Register Transfer Level), o de nivel de transferencia de registros. Existen herramientas que permiten sintetizar circuitos a partir de modelos de abstracción más elevados, pero en general lo hacen llevando el diseño a un nivel de descripción como RTL antes de sintetizarlo.

2.2.1 Características principales de los LDH's.

AHDL:

- Tiene la característica que le permiten ser eficiente para desarrollos de baja a media complejidad.
- Es un paso intermedio entre los lenguajes de bajo a alto nivel de abstracción o complejidad.
- Utiliza el modelo "behavioural" (de comportamiento) para describir la lógica que se desea implementar.

Handel-C:

- Lenguaje de alto nivel basado en C ANSI para la implementación de algoritmos en hardware.
- Extensiones para el diseño de hardware incluyendo tipos de datos flexibles, procesamiento en paralelo y comunicaciones entre elementos en paralelo.
- Soporta la metodología de software de reutilización.

System C:

- System C es un lenguaje de alto nivel basado en C++ que proporciona un conjunto de constructores que permite a los diseñadores modelar sistemas.
- Los constructores le permiten capturar la estructura jerárquica y modelar actividades en paralelo.
- También proporciona un rico conjunto de tipos de datos orientados a hardware (bits, vector de bits, caracteres, enteros, punto flotante, punto fijo, vectores de enteros, etc.)
- Soporta las señales de los cuatro estados lógicos 0, 1, X, Z.
- Comportamientos concurrentes se modelan mediante procesos.
- Un proceso es un hilo independiente que se ejecuta cuando determinadas condiciones ocurren o alguna señal cambia.

Forge:

- Permite compilar código estándar Java en Verilog, compatible con herramientas de síntesis e implementación.
- Permite al diseñador trabajar en un nivel de abstracción superior al de HDL.
- El compilador utiliza técnicas de análisis y optimización para crear arquitecturas eficientes a partir de una descripción funcional en software.

Verilog:

- Verilog fue inventado originalmente como un lenguaje de simulación.
- Cadence Design Systems decidió hacer público el lenguaje en 1990.
- Permite diseñar a varios niveles de abstracción.
- En 1995 se convierte estándar IEEE 1364.

ABEL:

- Programa cualquier tipo de PLD (Versión reciente)
- Proporciona tres diferentes formatos de entrada: ecuaciones booleanas, tablas de verdad y diagramas de estados.
- Es catalogado como un lenguaje avanzado HDL

CUPL:

- Creado por AMD para desarrollo de diseños complejos.
- Presenta una total independencia del dispositivo.
- Programa cualquier tipo de PLD.

- Genera descripciones lógicas de alto nivel.

VHDL:

- Facilita la descripción de circuitos con diversos niveles de abstracción.
- Sentencias de control de flujos (if, for while). Junto con la característica anterior potencia para desarrollar algoritmos.
- Capacidad de estructurar el código (subprogramas, funciones o procedimiento), permite afrontar algoritmos complejos.
- Posibilidad de utilizar y desarrollar bibliotecas de diseño.
- Incorpora conceptos específicos para el modelado del Hardware, como concurrencia y ciclo de simulación

2.2.2 VHDL: Organización y Arquitectura

En 1980 el rápido avance de los circuitos integrados lleva al desarrollo de estándares de diseño. Así surge VHDL como un estándar para describir circuitos digitales, siendo un estándar IEEE.

VHDL: Very High Speed Integrated Circuit Hardware Description Languaje, es un lenguaje de descripción de circuitos electrónicos digitales que utiliza distintos niveles de abstracción. VHDL no es un lenguaje de programación, por ello conocer su sintaxis no implica necesariamente saber diseñar con él.

La Entidad

La entidad únicamente describe la forma externa del circuito, aquí se enumeran las entradas y las salidas del diseño. Una entidad es análoga a un símbolo esquemático de los diagramas electrónicos, el cual describe las conexiones del dispositivo hacia el resto del diseño. La entidad puede ser vista como una "caja negra"

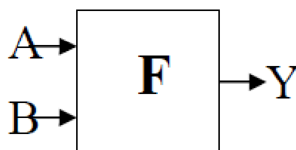


Figura 2.2.1 Entidad F

Ejemplo 1. Declaración de la entidad F únicamente con puertos de entrada y salida, ver Figura 2.2.1

```
entity F is
port (A, B: in bit;
        Y : out bit
     );
end  F;
```

Además, la entidad puede definir un valor genérico que se utilizará para declarar las propiedades y constantes del circuito, independientemente de cuál sea la arquitectura.

Ejemplo 2. Declaración de la entidad F describiendo valores genéricos (constantes) y puertos únicamente

```
entity F is
generic (cte1: tipo := valor1; cte2: tipo:= valor 2; …);
port     (entrada1, entrada2, … : in tipo;
          salida1, salida2, …: out tipo
         );
end F
```

Ports

En VHDL al declarar las entidades los puertos de entrada y salida pueden tener alguno de los siguientes modos.

In: Una señal que entra en la entidad y no sale. La señal puede ser leída pero no escrita.

Out: Una señal que sale fuera de la señal y no es usada internamente. La señal no puede ser leída dentro de la entidad.

Inout: Una señal que es bidireccional, entrada/salida de la entidad.

Buffer: Utilizado para una señal que es salida de la entidad. La señal puede ser usada dentro de la entidad, lo cual significa que la señal puede aparecer en ambos lados de la asignación.

Arquitectura

Los pares de entidades y arquitecturas se utilizan para representar la descripción completa de un diseño. Una arquitectura describe el funcionamiento de la entidad a la que hace referencia. Si una entidad la asociamos con una "caja" en la que se enumeran las interfaces de conexión hacia el exterior, entonces la arquitectura representa la estructura interna de esa caja.

Ejemplo 1.

```
architecture arch_name of entity_name is
        • declaraciones de la arquitectura
        • tipos
        • señales
        • componentes
begin
    • código de descripción
    • instrucciones concurrentes
    • ecuaciones booleanes
    process
    begin
    • código de descripción
    end process;
end arch_name;
```

Una entidad puede tener más de una arquitectura, pero cuando se compile se indica cual es la arquitectura que queremos utilizar.

PROCESS: Cuando en VHDL se escribe un process, dentro de él aparece la parte secuencial del circuito. La simulación no entra en el process hasta que no haya variado alguna de las señales o variables de su lista de sensibilidad independientemente de lo que este contenido dentro del process. Por otro lado únicamente dentro de un process pueden aparecer las sentencias de tipo if y else y nunca puede aparecer una sentencia del tipo wait.

Tipos de datos

VHDL es un lenguaje y como tal, posee sus tipos de datos y operadores. Los datos se almacenan en objetos que contienen valores de un tipo dado

- Constant. Los objetos de esta clase tienen un valor inicial que es asignado de forma previa a la simulación y que no puede ser modificado durante ésta.

  constant identificador: tipo:= valor;

- Variable. Los objetos de esta clase contienen un único valor que puede ser cambiado durante la simulación con una sentencia de asignación. Las variables generalmente se utilizan como índices, principalmente en instrucciones de bucle, o para tomar valores que permitan modelar componentes. Las variables NO representan conexiones o estados de memoria.

  variable identificador: tipo [:= valor];

- Signal. Los objetos de esta clase contienen una lista de valores que incluye el valor actual y un conjunto de valores futuros. Las señales representan elementos de memoria o conexiones y si pueden ser sintetizadas. Los puertos de una entidad son implícitamente declarados como señales en el momento de la declaración, ya que estos representan conexiones. También pueden ser declaradas en la arquitectura antes del BEGIN, lo cual nos permite realizar conexiones entre diferentes módulos.

  signal identificador: tipo;

VHDL permite utilizar tipos predefinidos, así como otros definidos por el usuario como lo son:

- **BIT** 0, 1
- **BIT_VECTOR** (rango*)
- **BOOLEAN** TRUE, FALSE
- **CHARACTER** {ascii}
- **STRING** {ascii}
- **SEVERITY_LEVEL** {WARNING, ERROR, FALURE}
- **INTEGER** rango*
- **NATURAL** rango*

- **POSITIVE** rango*
- **REAL** rango*
- **TIME**

Donde *(rango: n_min **TO** n_max; n_max **DOWNTO** n_min)

- **STD_LOGIC** Tipo predefinido en el estándar IEEE 1164. Este tipo representa una lógica multivaluada de 9 valores. Además del ‘0’ lógico y el ‘1’ lógico, posee alta impedancia ‘Z’, desconocido ‘X’ o sin inicializar ‘U’ entre otros. Igual que se permite crear un vector de bits se puede crear un vector de std_logic, STD_LOGIC_VECTOR.

Para poder utilizar el tipo std_logic hay que añadir la librería que lo soporta, por ejemplo:

Para poder utilizar el tipo std_logic: use ieee.std_logic_1164.all.

Para poder utilizar las funciones aritmética-lógicas definidas (suma, resta multiplicación): use ieee.std_logic_arith.all.

Si los vectores están en representación binaria pura: use ieee.std_logic_unsigned.all.

Los vectores están en C2: use ieee.std_logic_unsigned.all.

El tipo enumerado es un tipo de dato con un grupo de posibles valores asignados por el usuario. Los tipos enumerados se utilizan principalmente en el diseño de máquinas de estados:

type nombre **is** (valor1**,** valor2**,** …**);**

Los tipos enumerados se ordenan de acuerdo a sus valores. Los programas de síntesis automáticamente codifican binariamente los valores del tipo enumerado para que estos puedan ser sintetizados. Algunos programas lo hacen mediante una secuencia binaria ascendente, otros buscan cual es la codificación que mejor conviene para tratar de minimizar el circuito o para incrementar la velocidad del mismo una vez que la descripción ha sido sintetizada. También es posible asignar el tipo de codificación mediante directivas propias de la herramienta de síntesis.

2.2.3 Operadores

Un operador nos permite construir diferentes tipos de expresiones mediante los cuales podemos calcular datos utilizando diferentes objetos de datos con el tipo de dato que maneja dicho objeto. En VHDL existen distintos operadores de asignación con lo que se transfieren valores de un objeto de datos a otro, y operadores de asociación que relacionan un objeto de datos con otro, lo cual no existe en ningún lenguaje de programación de alto nivel.

- Lógicos: and,or, nor, xor, xnor
- Relacionales: = igual, /= distinto, < menor , <= menor o igual, > mayor, >= mayor o igual
- Misceláneos: abs valor absoluto, ** exponenciación, not negación.
- Adición: + suma, – resta, & concatenación de vectores.
- Multiplicativos: * multiplicación, / división, rem resto, mod módulo,

- Signo: +, –
- Desplazamiento (signed y unsigned): shift_right, shift_left

2.2.4 Sentencias de descripción

Las sentencias de descripción pueden ser:

- Wait, esta instrucción es utilizada en procesos que no tienen lista de sensibilidad:

```
wait on signal_list;
wait for time_expression;
wait until condition;
```

- Eventos sobre las señales (‘event) nos indican cuando ocurre un cambio en la señal

```
signal 'event
signal 'last_event
signal 'last_value
```

- If – then – else, sólo son aplicables dentro de un process

```
if condición then
  ...   sequential statements
elsif otra_condición then
  ...   sequential statements
else
...     sequential statements
end if;
```

- Case – When sólo son aplicables dentro de un process

```
case  expresión  is
when   alternativa_l => ...  seq. statements
…
when   alternativa_n => ... seq. statements
when others   =>   ...   seq. statements
end case;
```

- For – Loop sólo son aplicables dentro de un process

```
for  loop_var  in  range  loop
    ...  sequential statements
end loop;
```

- While – Loop sólo son aplicables dentro de un process

```
while  condición  loop
  ...   sequential statements
end loop;
```

- When – Else

```
Signal_name <= valor_1  when  condición1  else
               valor_2  when  condición2  else
               ...
               valor_i  when  condicióni else
               otro_valor;
```

- With – Selec – When

```
with identificador select
Signal_name <= valor_1 when valor_identificador1,
               valor_2 when valor_identificador2,
               ...
               valor_i when valor_identificadori,
               otro_valor when others;
```

2.2.5 Descripción Estructural

Esta descripción utiliza entidades descritas y compiladas previamente. Se declaran los componentes que se utilizan y después, mediante los nombres de los nodos, se realizan las conexiones entre las compuertas. Las descripciones estructurales son útiles cuando se trata de diseños jerárquicos.

Una descripción estructural:

- Describe las interconexiones entre distintos módulos.
- Estos módulos pueden a su vez tener un modelo estructural o de comportamiento.
- Normalmente ésta es una descripción a más alto nivel.

Ejemplo 1.

```
entity entity_name is
        port(…);
end  entity_name;

architecture arch_name of entity_name is
     component component_name
         port (…);
     end component;
signal declarations;
begin
     component_i: component_name port map (io_name)
end arch_name;
```

Generate: Las secuencias de generación de componentes permiten crear una o más copias de un conjunto de interconexiones, lo cual facilita el diseño de circuitos mediante descripciones estructurales.

Ejemplo 2.

```
for indice in rango generate
        -- instrucciones concurrentes
end generate;


component comp
port( x: in bit; y: out bit);
end component comp
…
signal a, b: bit_vector(0 to 7)
…
gen: for i in 0 to 7 generate
u: comp port map(a(i),b(i));
end generate gen;
```

2.2.6 Descripción por comportamiento (behavioural)

Las descripciones por comportamiento son similares a un lenguaje de programación de alto nivel por su alto nivel de abstracción. Más que especificar la estructura o la forma en que se deben conectar los componentes de un diseño, nos limitamos a describir su comportamiento. Una descripción por comportamiento consta de una serie de instrucciones, que ejecutadas modelan el comportamiento del circuito. La ventaja de este tipo de descripción es que no necesitamos enfocarnos a un nivel de compuerta para implementar un diseño.

Los módulos que se suelen describir mediante comportamiento suelen ser:

- Módulos que están al final de la jerarquía en una descripción estructural.
- Paquetes CI de una librería.
- Módulos cuyo comportamiento es complicado para poderlo describir mediante una estructura.
- Process con wait, if, else, when, case ...
- Muchas veces las funciones dependen del tiempo, VHDL resuelve este problema permitiendo la descripción del comportamiento en la forma de un programa tradicional.

2.2.7 Ejemplos

Ejemplo 1. Este ejemplo muestra un diseño de una entidad completa, así como de la arquitectura en descripción por comportamiento

```
ENTITY __entity_name IS
  GENERIC(__parameter_name : string :=  __default_value;
          __parameter_name : integer:=  __default_value);
  PORT(
          __input_name, __input_name: IN      STD_LOGIC;
          __input_vector_name          : IN STD_LOGIC_VECTOR(__high downto __low);
          __bidir_name, __bidir_name  : INOUT         STD_LOGIC;
          __output_name, __output_name       : OUT   STD_LOGIC);
END __entity_name;
```

```
ARCHITECTURE arch_name OF __entity_name IS
        SIGNAL __signal_name : STD_LOGIC;
        SIGNAL __signal_name : STD_LOGIC;
BEGIN
        -- Process Statement
        -- Concurrent Procedure Call
        -- Concurrent Signal Assignment
        -- Conditional Signal Assignment
        -- Selected Signal Assignment
        -- Component Instantiation Statement
        -- Generate Statement
END arch_name;
```

Ejemplo 2. Diseño de un Multiplexor 2X1: Para este ejemplo el objetivo es crear un sistema funcione como un multiplexor 2x1. Además se definirán varias entradas de datos que actuarán como salidas. Cuando la señal de selección este a cero no se producirá ninguna salida, es decir el valor será cero.

Entradas:

- Entradas: D0, D1
- S0: señal que indica la señal que va a ser devuelta.

Salidas:

- salida: O

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

ENTITY mux2 IS
  PORT (D0, D1, S0: in std_logic;
          O: out std_logic);
END mux2;

   ARCHITECTURE ejemplo2 OF mux2 IS
BEGIN
multiplexor: PROCESS(D0,D1,S0)
 IF (S0 = ‘1’) THEN
 O <= D1;
 ELSE
 O <= D0;
 END IF;
END  PROCESS;
END ejemplo2;
```

Ejemplo 3. Diseño de un Sumador: el objetivo es crear un sumador que dadas dos entradas de datos devuelva la suma de éstos.

Entradas:

- a: operando 1.
- b: operando 2.

Salidas:

- salida: suma de las entradas.

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

  ENTITY sum IS
   PORT (a : IN std_logic_vector(3 DOWNTO 0);
         b : IN std_logic_vector(3 DOWNTO 0);
         salida : OUT std_logic_vector(4 DOWNTO 0));
  END sum;

  ARCHITECTURE ejemplo3 OF sum IS
  BEGIN

   PROCESS (a, b) IS
   BEGIN
    salida <= a + b;
   END PROCESS;
  END ejemplo3;
```

Ejemplo 4. Diseño de una puerta triestado: el objetivo es crear una puerta que tenga una señal de operación la cual, a estado alto, habilite la salida, por lo tanto el valor de la entrada pasará a la salida. Cuando la señal de operación esté a nivel bajo la puerta no sacará una señal, es decir, estará en alta impedancia.

Entradas:

- entrada: entrada de datos.
- op: señal que indica el modo de funcionar de la puerta.

Salidas:

- salida: salida de datos.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY triestado IS
 PORT(op, entrada: IN std_logic;
    salida: OUT std_logic);
```

```
END triestado;

ARCHITECTURE ejemplo4 OF triestado IS
BEGIN
  PROCESS(entrada,op)
  BEGIN
    IF op='1' THEN
      salida <= entrada;
    ELSE
      salida <= 'Z';
    END IF;
  END PROCESS;
END ejemplo4;
```

Ejemplo 5. Contador: El objetivo es crear un contador con dos salidas, las cuales corresponderán a un contador rápido y a otro lento. Además, se dispondrá de cuatro señale de entrada, donde habrá dos reseteadores (uno para cada contador) y dos enables, que permitirán el paro e inicio de los contadores. También se incluirá el reset y el reloj del sistema.

Entradas:

- rst: Reset del sistema.
- clk: Reloj del sistema.
- reset1: Reset del contador rápido.
- reset2: Reset del contador lento.
- enable1: Activación-Desactivación del contador rápido.
- enable2: Activación-Desactivación del contador lento.

Salidas:

- count1: Salida del contador rápido.
- count2: Salida del contador lento.

Los enables (enable1 y enable2) no actúan como enables puros, realmente cuando se produce un evento en dicha entrada pasará a activarse o desactivarse el contador correspondiente.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY count IS
  PORT (clk : IN std_logic;
        rst : IN std_logic;
        enable1 : IN std_logic;
        enable2 : IN std_logic;
        reset1 : IN std_logic;
        reset2 : IN std_logic;
        count1 : OUT std_logic_vector(7 DOWNTO 0);
        count2 : OUT std_logic_vector(31 DOWNTO 0));
END count;
```

```
ARCHITECTURE ejemplo5 OF sum IS
 SIGNAL cnt1 : std_logic_vector(7 DOWNTO 0) := (others => '0');
 SIGNAL cnt2 : std_logic_vector(31 DOWNTO 0) := (others => '0');
 SIGNAL en1 : boolean := false;
 SIGNAL en2 : boolean := false;
BEGIN

 pSeq : PROCESS (clk, rst) IS
 BEGIN
  IF rst = '1' THEN
   cnt1 <= (others => '0');
   cnt2 <= (others => '0');
  ELSIF clk='1' AND clk'event THEN
   IF en1 THEN
    cnt1 <= cnt1 + 1;
   END IF;

   IF en2 THEN
    cnt2 <= cnt2 + 1;
   END IF;
  END IF;
 END PROCESS;

 pCom : PROCESS (enable1, enable2) IS
 BEGIN
  IF enable1 = '1' THEN
   en1 <= NOT en1;
  END IF;

  IF enable2 = '1' THEN
   en2 <= NOT en2;
  END IF;
 END PROCESS;

 count1 <= cnt1;
 count2 <= cnt2;

END ejemplo5;
```

2.3 Herramientas de Desarrollo

La facilidad de desarrollo de sistemas sobre PLDs depende en gran medida del software y hardware que permite crear y probar los diseños de una manera simple y eficiente. Al conjunto de estas aplicaciones se los denomina herramientas de desarrollo. A continuación se describen algunas características de estas herramientas para los distintos pasos del ciclo de diseño.

- Descripción del Diseño: en la creación de fuentes se usan diferentes herramientas para facilitar el diseño.
- Algunas herramientas comunes: modelos de comportamiento, diferentes niveles, interfaces gráficas, etc.
- Generación o traducción: se lleva todos los tipos de fuentes, tales como representaciones gráficas de máquinas de estado o diagramas esquemáticos a un lenguaje de descripción de hardware. Los más comunes son VHDL o Verilog.
- Simulación: la simulación de un sistema descrito de modelos en HDL merece algunos comentarios. Los lenguajes de descripción de hardware modelan o describen, mediante instrucciones secuenciales, bloques de hardware que funcionarán de manera concurrente, es decir, al mismo tiempo. Las señales de los bloques pueden afectarse mutuamente. Simular esto tiene su complejidad, y dentro de los estándares que definen los lenguajes de descripción de hardware VHDL y Verilog se especifica cómo se deben simular los procesos concurrentes.

2.3.1 Ambientes integrados que permiten la creación de diseños para dispositivos CPLD's y/o FPGA's.

Las herramientas software que se utilizan para asistir en el diseño se dividen en dos categorías la primera, CAD: **computer aided design** (diseño asistido por computador) y la segunda, CAE: **computer aided engineering** (asistencia para ingeniería utilizando un computador), para el diseño de sistemas digitales las herramientas están compuestas por las dos clases de software. Entre las herramientas disponibles para la simulación de un modelo hardware, se tienen: ISE Foundation, Leonardo Spectrum, Cadence HDL, Maxplus, Quartus, entre otros. Cada una de estas herramientas permite la descripción, síntesis, simulación y programación de los dispositivos lógicos programables. Actualmente Xilinx y Altera son los dos grandes principales desarrolladores de CPLD's y FPGA's, por lo que solo se dará una descripción general de estos dos.

2.3.2 Altera - Max Plus II.

El software Max Plus II está orientado al diseño de dispositivos lógicos programables de Altera, pero dadas sus características también se puede emplear como paquete de diseño y simulación electrónica digital.
La entrada de diseño se puede hacer de modo gráfico o mediante un LDH, que permite la utilización de ecuaciones booleanas, tablas de verdad, definición de máquinas de estado, etc.
Realiza la simulación de circuitos de gran tamaño presentando las gráficas de simulación, la estimación de tiempos de retardo, la partición en circuitos integrados para su programación, etc.

Características generales.

- Ambiente que integra herramientas de diferentes fabricantes como Mentor Graphics, SynaptiCAD, Synplicity, y herramientas propias en un paquete de desarrollo FPGA.
- Manejador de proyectos que facilita el proceso de desarrollo y administra las herramientas de
- diferentes fabricantes.
- Soporta las familias de dispositivos ACEX®, FLEX® y MAX® de Altera cuyas densidades se encuentran en el rango de 600 a 250,000 compuertas.
- Ambiente que soporta EDIF, VHDL, y Verilog.

El entorno de desarrollo integrado MAX+PLUS II para dispositivos lógicos programables de Altera incluye un completo flujo de diseño (figura 2.3.1) y un entorno gráfico de trabajo intuitivo.

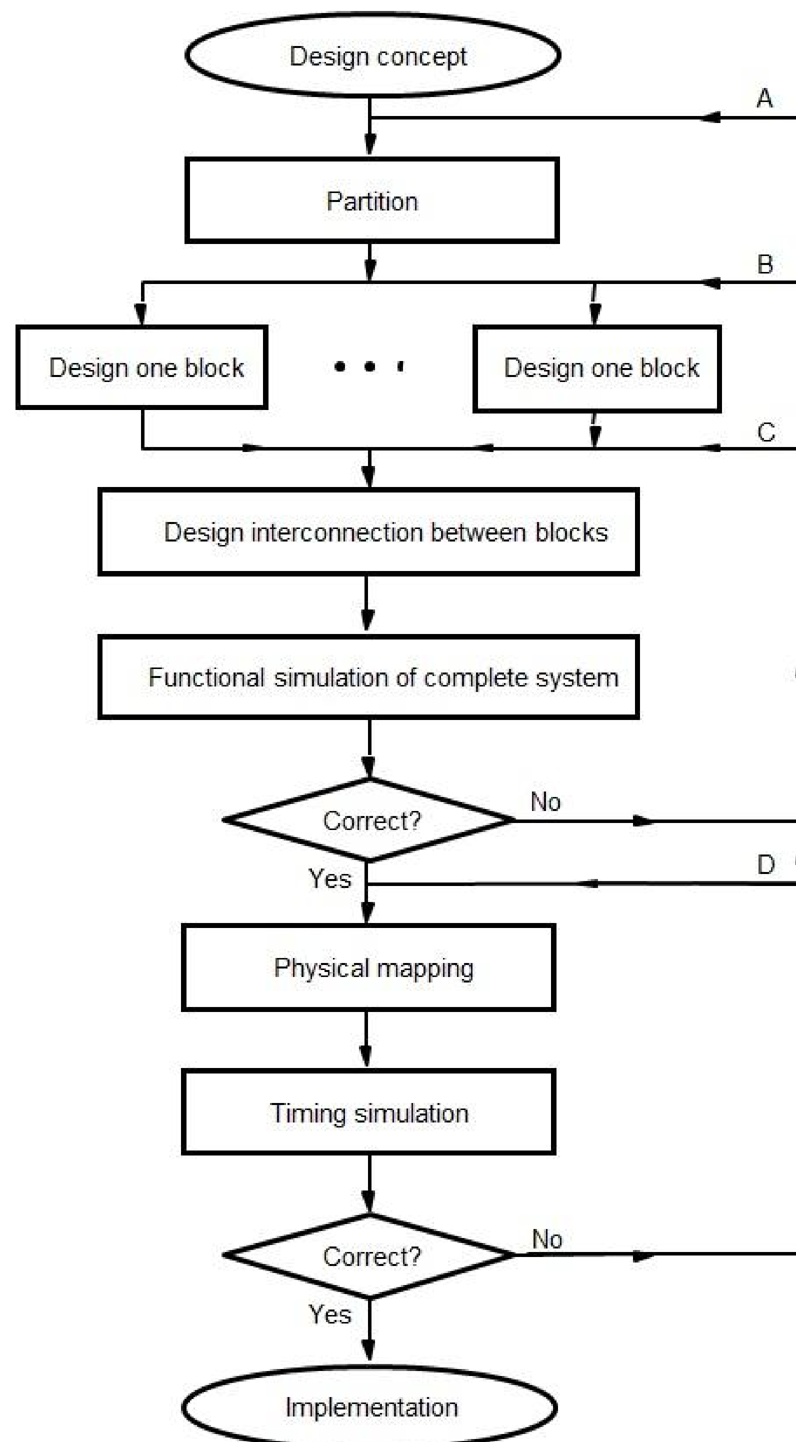


Figura 2.3.1 Flujo de diseño

MAX+PLUS II está dividido en cuatro fases claramente diferenciadas: entrada del diseño, compilación, verificación (simulación) y programación del dispositivo programable.

Entrada de diseño

Constituida por la introducción del diseño (parte lógica) y por la asignación y configuración del dispositivo (parte física). La introducción del diseño se puede realizar mediante diversos formatos:

- Esquemático: mediante un editor gráfico integrado que se complementa con una serie de librerías con las primitivas y las funciones estándar más usuales (primitive symbols, LPM functions, MegaCore functions).
- Lenguaje de descripción hardware: además de permitirse la entrada de diseño mediante el uso del lenguaje de programación hardware de Altera (AHDL), también es posible el uso de los lenguajes estándar Verilog y VHDL**.**
- Forma de onda: mediante un editor gráfico integrado de formas de onda que permite crear archivos de entrada de diseño así como vectores de entrada para la simulación.
- Estándar EDIF: es posible incorporar partes del diseño realizadas con otras herramientas de desarrollo gracias al soporte del formato estándar EDIF, soportado por compañías como Cadence, Mentor Graphics y Viewlogic.

La asignación y configuración del dispositivo permite seleccionar cualquiera de los dispositivos programables de Altera y hacer la asignación de pines de nuestro diseño sobre el dispositivo seleccionado.

Compilación

Cuando la entrada de diseño se ha completado, es necesario compilar el diseño. En el proceso de compilación se realiza en primer lugar la síntesis lógica del diseño optimizándola para la arquitectura del dispositivo programable utilizado. Tras la síntesis lógica, se realiza el particionado del diseño en uno o varios dispositivos programables (si el diseño completo no cabe en un solo dispositivo programable). En último lugar se generan los archivos de análisis temporal y de simulación del diseño, así como los archivos de programación del dispositivo programable.

Programación del dispositivo

Tras la verificación del diseño en la cual se ha comprobado su correcto funcionamiento es necesario transferir el diseño al dispositivo programable. El proceso se realiza mediante el programador de dispositivos programables integrado, el cual permite programar, verificar y borrar los dispositivos programables utilizados.

2.3.3 Altera - Quartus II

El software Quartus II proporciona una completa multiplataforma en torno de diseño, es compatible con la mayoría de los dispositivos que maneja el software MAX PLUS II, pero no es compatible con los dispositivos obsoletos o paquetes.

El sistema de desarrollo Quartus II es una plataforma de herramientas para el diseño de circuitos digitales sobre dispositivos FPGA y CPLD. Quartus II provee aplicaciones para la entrada de diseño, síntesis lógica, simulación lógica, ubicación y conexionado, análisis temporal, administración de potencia y programación de dispositivos, junto con una variedad de utilitarios y aplicaciones adicionales para el diseño lógico programable.

El software Quartus II, ahora cuenta con ventajas únicas en la metodología de flujo de diseño (figura 2.3.2) de FPGAs, diseño de sistemas, metodología de tiempo - cierre, tecnología de verificación en sistema y el apoyo de EDA a terceros.

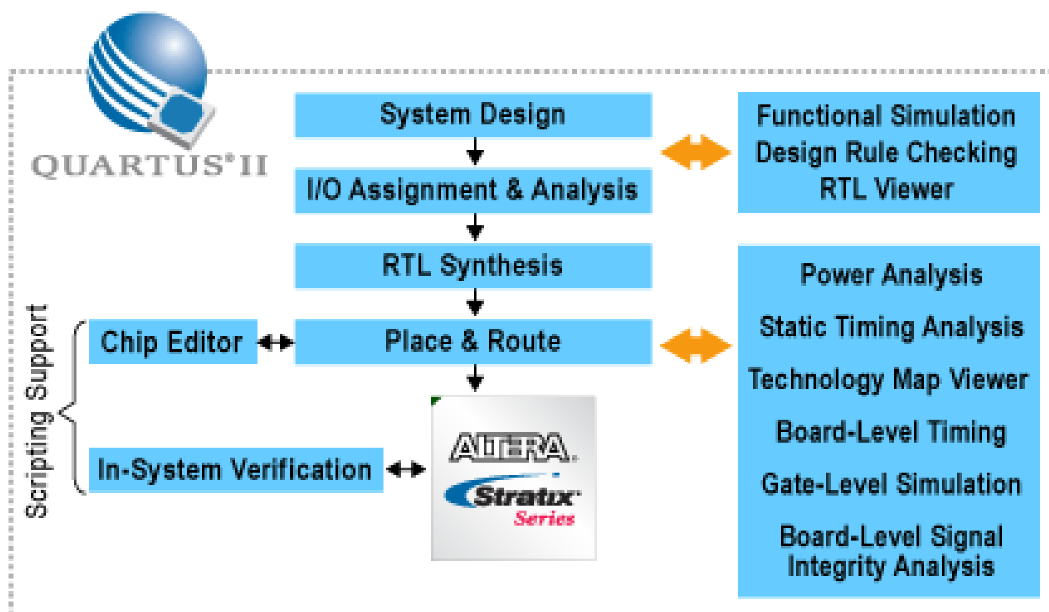


Figura 2.3.2 Flujo de Diseño

Diseño de entrada y tecnología de síntesis

Software de diseño Quartus II ofrece la más alta productividad, rendimiento y ofrece numerosas características de diseño para acelerar el proceso.

- Varios métodos de diseño de entrada.
- Secuencias de comandos de apoyo.
- Compilación incremental.
- SOPC Builder para el sistema de nivel de diseño.
- MegaWizard Plug-In Manager para integrar rápida y fácilmente una amplia cartera de propiedad intelectual (IP) de los núcleos.
- Análisis de asignación de pines y gestión de pines para I/O
- Flujo básico de compilación

Verificación y solución a nivel de placa.

Integración con todas las herramientas de verificación de los principales fabricantes de partes y metodologías, el software Quartus II establece lo siguiente:

- TimeQuest: analizador de tiempo
- Análisis integrado de alimentación con el análisis del poder y optimización de la tecnología PowerPlay.
- SignalTap II analizador lógico incorporado (con el apoyo de la función de compilación incremental para acelerar el ciclo de verificación)
- RTL Visor / Visor de mapas Tecnología
- Verificación de soporte a terceros

Herramientas de optimización

Quartus II ofrece la máxima productividad y rendimiento para FPGA's, CPLD's y ASIC, ofrece numerosas funciones de optimización para mejorar el proceso de diseño:

- Optimización de la compilación incremental.
- Optimización de la síntesis de física para aumentar el rendimiento máximo de los diseños.
- Diseño automático de optimización de secuencia de comandos.
- Optimización de asesor.

Más fácil de usar software de diseño CPLD

Quartus II Edición de subscripción y el Quartus II Edición Web gratuita, dan completo soporte al diseño de CDPL con el software MAX II con un entorno de diseño completo y fácil de usar que puede tomar proyectos de diseño de CPLD de principio a fin (Figura 2.3.3). Quartus II Edición de subscripción y el software Quartus II Edición Web también se integran a la perfección con todas las síntesis de los principales fabricantes de partes y herramientas de simulación.

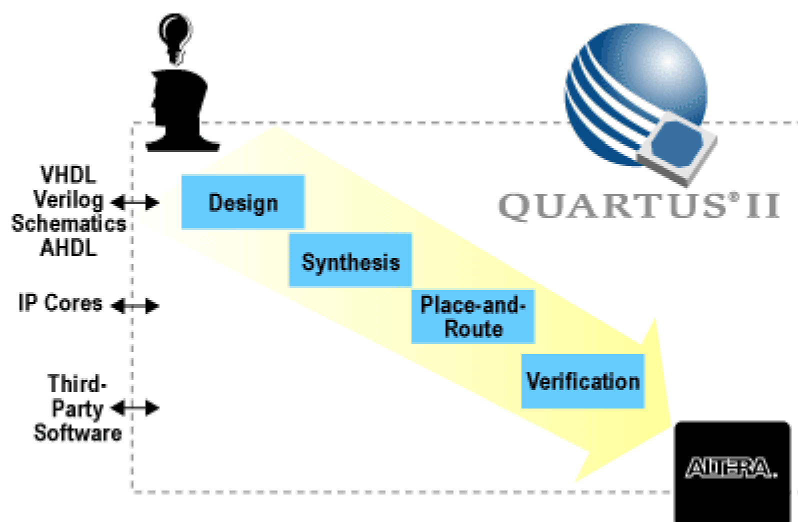


Figura 2.3.3 Flujo de diseño CPLD

Quartus II Ventajas para los usuarios de MAX + Plus II

El software de Quartus II es de más alto rendimiento y más fácil de usar software para diseños CPLD. Con la opción que permite seleccionar un ambiente como el de MAX + PLUS II, los usuarios pueden obtener todos los beneficios de las características avanzadas y un rendimiento del software Quartus II sin tener que aprender una nueva interfaz.

2.3.4 Xilinx-ISE (Integrated Software Environment)

El entorno de diseño Xilinx-ISE consiste en una herramienta que permite realizar un diseño completo basado en lógica programable (tanto CPLD como FPGA), es decir, incluye todas las etapas necesarias como son:

- La entrada de diseño, bien a través de captura esquemática, lenguajes de descripción hardware como ABEL, VHDL o Verilog, o representación gráfica de diagramas de estado (*StateCAD/State Bencher*).
- Herramientas de verificación para la obtención de una simulación del sistema, tanto a nivel funcional como de estimación de retardos. La herramienta empleada se denomina ModelsimXE. Por otro lado, también se facilita la generación de bancos de prueba para la verificación mediante la herramienta *HDL Bencher.*
- Herramientas de implementación donde se permite la especificación de restricciones o indicaciones para realizar una implementación óptima sobre el dispositivo lógico programable

especificado. Esta herramienta incluye tres etapas principales en el diseño: *Translate, Map, Place & Route.*

- Herramientas de programación, para permitir descargar el diseño sobre el dispositivo físico, ya sea en una placa de evaluación o bien en la placa definitiva mediante la programación in-situ (en sistema) a través de la programación JTAG. De este modo, es posible probar y depurar el sistema sobre el hardware de forma rápida y flexible, permitiendo tantos cambios como sean necesarios.

En la figura 2.3.4 se muestra de forma global los procesos que se llevan a cabo durante el diseño de sistemas basados en lógica programable.
Este entorno permite combinar las diferentes técnicas de diseño para facilitar la labor de descripción del diseño. Además, se permite la inclusión de restricciones para optimizar el proceso de implementación y adaptarlo a las necesidades del diseño, como ejemplo, inclusión de restricciones temporales para determinadas señales, restricciones de ubicación de la lógica en una determinada zona del arreglo programable, o bien inclusión de opciones de particularización en elementos hardware como asignación de patillas, líneas de reloj específicas, etc.

Por otro lado, el conocimiento de este tipo de entornos permite ser capaces de emplearlo en multitud de diseños, ya que independientemente de la complejidad del diseño, si éste está destinado a un dispositivo programable, ya sea CPLD o FPGA, será posible realizarlo mediante el mismo software, por lo que una vez dominado, el proceso de diseño de nuevos sistemas resulta mucho más rápido.

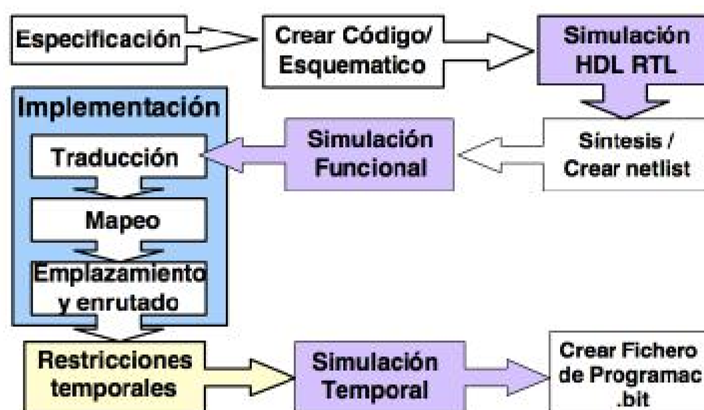


Figura 2.3.4 Diagrama de evolución de los procesos involucrados en el diseño de sistemas basados en lógica programable.

El entorno de desarrollo ISE de Xilinx posee un aspecto similar al de los entornos de programación actuales como puede ser Visual Basic o Visual C, es decir, posee diversas ventanas para visualización de tareas específicas sobre cada una de ellas. En este caso existen cuatro tipos de ventanas (figura 2.3.5):

- Ventana de ficheros fuente. En esta ventana se muestran los ficheros fuente utilizados en el diseño y las dependencias entre ellos. También es aquí donde se elige el tipo de dispositivo donde se desea implementar el diseño. Esta ventana posee diversas solapas para visualizar diferentes tipos de información relativa a las fuentes de diseño empleadas.

- Ventana de Procesos. Esta ventana muestra todos los procesos necesarios para la ejecución de cada etapa de diseño. La lista de procesos se modifica dinámicamente dependiendo del tipo de fuente seleccionado en la ventana de ficheros fuente.

- Ventanas de edición. Al hacer doble clic sobre un fichero fuente de la ventana de ficheros fuente se abre una ventana de edición para modificar el fichero (en caso de lenguaje HDL), o bien se ejecuta el programa que permite editar el diseño (en caso de diseños esquemáticos o máquinas de estado).

- Ventana de información, situada en la parte inferior. Muestra mensajes de error, aviso o información emitidos por la ejecución de los programas de compilación, implementación, etc.

Tanto en la ventana de procesos como en la de ficheros fuente es posible modificar las opciones de cada elemento a través del botón derecho del ratón, o bien a través de los menús del entorno de diseño; estos menús se modifican dependiendo del tipo de selección realizada en las ventanas de ficheros fuente y de procesos.

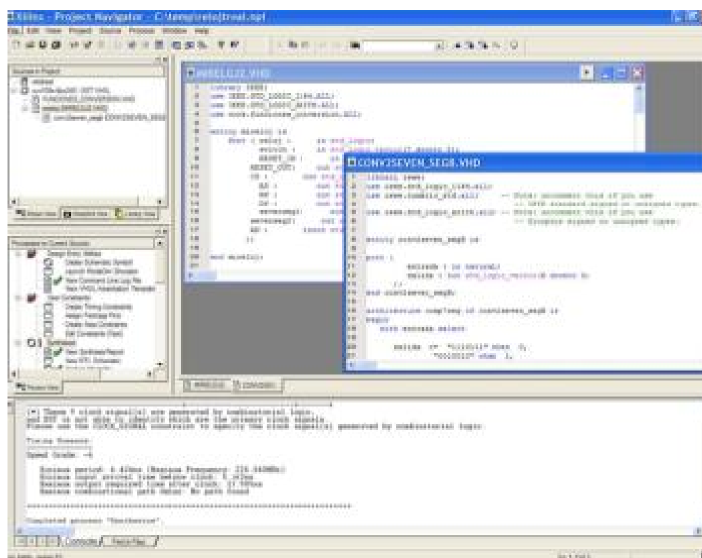


Figura 2.3.5 Pantalla principal del entorno Xilinx ISE.

En concreto, la ventana de procesos incorpora todas las opciones necesarias para realizar todos los pasos de implementación de sistemas en lógica programable, incluyendo la edición y verificación.

Desarrollo de sistemas con la plataforma ISE Desing Suite

La primera de las etapas en el flujo de diseño es la “Descripción”. Esta etapa consiste en realizar los componentes que conformarán el sistema. Para esto existen dos variantes: hardware y software. La figura 2.3.6 muestra el diagrama de flujo de diseño en Xilinx ISE, y la figura 2.3.7 muestra las diversas partes en que se divide la ventana de procesos dependiendo de la tarea a realizar.

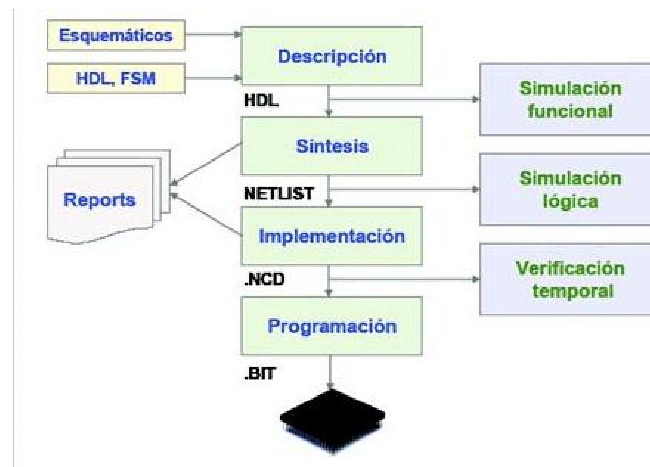


Figura 2.3.6 Proceso de diseño en Xilinx ISE

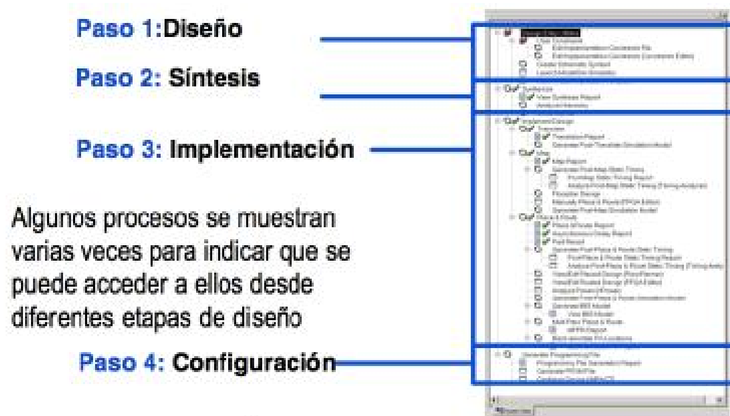


Figura 2.3.7 División de tareas dentro de la ventana de procesos.

Los componentes hardware se configuran en una interfaz para diseños esquemáticos. O sea, existe un número de módulos como contadores, multiplicadores, multiplexores, etc., que se pueden configurar e interconectar en esta interfaz. Además existen las mega-funciones, que no son más que los módulos mencionados pero sobredimensionados en sus entradas-salidas y otras características.

Los componentes software son programados en un editor para lenguajes HDL o Verilog. Mediante estos editores se realizan módulos hardware pero describiendo su estructura y funcionalidad a través de código software. Esta variante puede resultar más cómoda para los desarrolladores en tanto resulta menos engorroso implementar un algoritmo software que un circuito hardware; y el tiempo de desarrollo se acorta significativamente. No obstante, las variantes de diseños esquemáticos resultan más eficientes en cuanto al uso de recursos y exactitud en la temporización, por ejemplo.

Como parte de la etapa de descripción existen además un editor de máquinas de estado y un generador de módulos IP. Los módulos IP son componentes hardware diseñados por el desarrollador, aunque algunos los provee el fabricante. Pueden ser diseñados en un editor de esquemáticos o uno para lenguajes de descripción de hardware. Estos módulos pueden reutilizarse en otros diseños y esta es su ventaja fundamental. Existen módulos IP que constituyen descripciones software de chips microcontroladores como "Orégano", o microprocesadores como es el caso de MicroBlaze, el cual se

detallará en un apartado posterior. Esto significa que dentro de un FPGA se pueden implementar uno o más microcontroladores o microprocesadores, los cuales ejecutarán con sus respectivos CPUs una aplicación software programada en Ensamblador o C++ fundamentalmente.

Existen varios tipos de módulos IP, también llamados mega-funciones:

- Soft CORE: describe mediante código software la estructura y funcionalidad de un componente hardware. Son independientes de la tecnología sobre la cual serán implementados, por lo que pueden ser utilizados en cualquier dispositivo que disponga de recursos suficientes.
- Firm CORE: presentan una descripción estructural típicamente en HDL. Están diseñados para una arquitectura específica, aunque pueden ser implementados en otras arquitecturas.
- Hard CORE: descripción física suministrada mediante ficheros de *layout*. Son dependientes de la tecnología y no pueden ser implementados sobre otras arquitecturas porque su diseño está basado y comprometido con la utilización de los recursos específicos de un dispositivo.

Una segunda etapa en el flujo de diseño con ISE es la “Síntesis”. Además de los componentes logrados se deben definir las restricciones del diseño como dato necesario para este proceso. Durante la síntesis se detectan los errores de sintaxis, se analiza la jerarquía entre los componentes, se identifican e implementan las macros dentro del diseño, se lleva a cabo una optimización en cuanto a tiempo, se mapean los componentes del sistema en las LUTs y se realiza una replicación de los registros.
Como resultado del proceso anterior se genera un fichero de trazas y un fichero *netlist* que sirve como entrada para el proceso de implementación.

Durante la etapa de implementación se combinan varios ficheros de descripción de componentes en un solo fichero *netlist* , mediante el proceso *Ngdbuild* de ISE. También se hace corresponder cada símbolo lógico resultado del *netlist* con componentes físicos del FPGA. El proceso *Place and Route* ubica los componentes en el FPGA y los interconecta. Esto puede modificarse posteriormente por el desarrollador pero no es recomendable, pues este proceso se lleva a cabo optimizando los recursos de espacio y teniendo en cuenta las rutas más factibles para la interconexión de los elementos.

Justo antes de programar el FPGA se puede verificar el diseño mediante herramientas de simulación como ModelSim y realizar análisis de tiempo con el Xilinx Time Analyzer. Estas herramientas brindan una fiabilidad mayor aún para garantizar que el diseño se encuentre libre de errores.
Por último se programa el FPGA. En el caso de los kits de desarrollo como el de Spartan 3, es posible programar directamente el FPGA o se puede programar la memoria PROM de configuración de la placa. La PROM trae de fábrica una aplicación que inicializa el FPGA haciendo un chequeo de los componentes del kit y mostrando un conteo en las lámparas de siete segmentos

De manera resumida, el proceso de diseño resulta sencillo y se realiza en tres pasos, el primer paso consiste en añadir los ficheros fuente, en el segundo paso se selecciona el fichero de más alto nivel que se quiere implementar, y finalmente se hace doble clic sobre el último proceso al que se desea llegar, de este modo se ejecutarán todos los procesos intermedios necesarios para llegar al proceso seleccionado en último lugar (ver figura 2.3.8).

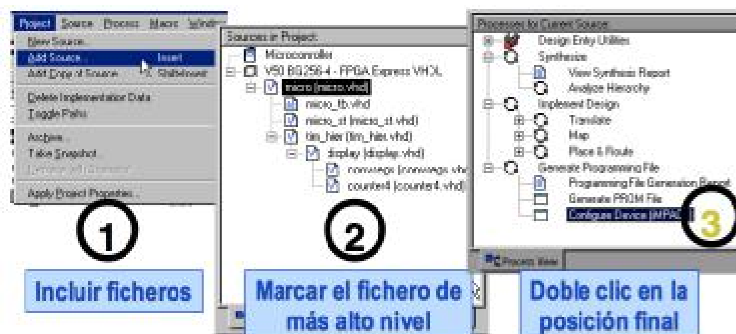


Figura 2.3.8 Proceso simplificado para el desarrollo de un diseño en Xilinx ISE.

Chips compatibles con ISE.

ISE suite de diseño del dispositivo de apoyo		
	ISE herramienta Webpack (Gratis)	**ISE Design Suite (De pago)**
Virtex FPGA	Virtex-4 LX: XC4VLX15, XC4VLX25 SX: XC4VSX25 FX: XC4VFX12 Virtex-5 LX: XC5VLX30, XC5VLX50 LXT: XC5VLX20T - XC5VLX50T FXT: XC5VFX30T Virtex-6 XCLX75T	Virtex-4 LX: Todos SX: Todos FX: Todas las Virtex-5 LX: Todos LXT: Todos SXT: Todos FXT: Todos Virtex-6 Todos los
Spartan FPGA	Spartan-3 XC3S50 - XC3S1500 Spartan-3A Todos los Spartan-3AN Todos los Spartan-3A DSP XC3SD1800A Spartan-3E Todos los Spartan-6 XC6SLX4 - XC6SLX75T XA (Xilinx Automotriz) Spartan-6 Todos los	Spartan-3 Todos los Spartan-3A Todos los Spartan-3AN Todos los Spartan-3 DSP Todos los Spartan-3E Todos los Spartan-6 Todos los XA (Xilinx Automotriz)
CoolRunner PLA CoolRunner-II CPLD CoolRunner-II CPLD	Todos los	
Serie XC9500 CPLD	Todos (excepto 9500XV familia)	

CAPÍTULO 3

DISEÑO DE LAS PRÁCTICAS

Descarga e Instalación Paquete ISE Design Suite 13

Esta tesis explica como instalar ISE Design Suite 13, la cual incluye el software (ISE) medio de software integrado, el software ChipScope Pro, Herramientas agregadas (EDK y SDK), Software Generador de Sistema para DSP y Herramientas de diseño PlanAhead.

3.1 Descarga de Paquete ISE Design Suite 13.1

Para comenzar, abra una página del navegador de internet e introduzca la siguiente dirección:

http://www.xilinx.com/support/download/index.htm

- Permita los complementos (pop-ups) desde entitlenow.com
- Establecer la configuración de seguridad para permitir elementos seguros y no seguros para ser mostrados en la misma página
- Permita que el manejador de descargas Akamai corra procesos Java.

Para descargar el software ISE, haga lo siguiente:

1.- Asegúrese de que está seleccionada la pestaña de Herramientas de Diseño (Design Tools) en la página web.

2.- Bajo el título de la versión, haga clic en la versión de la herramienta que usted esté interesado en descargar. Para este caso será la versión 13.1.

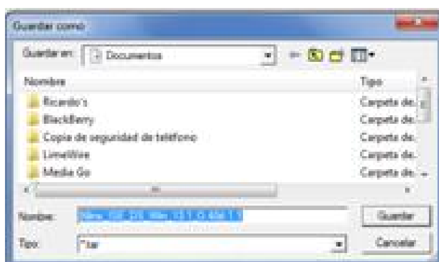
3.- Haga clic en el vínculo para el instalador que desee descargar. Para este caso será All Plataforms (TAR/GZ – 4.93 GB) de la Instalación del Producto Completo ISE Design Suite 13.1.

4.- Introduzca su User ID y su Password para ingresar a su cuenta Xilinx.
Nota: Si no tiene una cuenta de Xilinx, deberá crear una para descargar los productos.
Aparecerá una pantalla de verificación de la dirección.

About the Download Manager

The Download Manager helps ensure that your Xilinx product download is successful. If your internet connection is temporarily lost or your computer is restarted, the Download Manager is able to continue the download process where it left off.

You are being presented with a browser security warning regarding the installation of this Download Manager. By accepting the Akamai Technologies Inc. security certificate, the Download Manager will install and begin to download the requested file.



5.- Una vez que la dirección actual es correcta, dar clic en Siguiente (Next).

6.- El manejador de descargas Akamai iniciará en su navegador el proceso de descarga hasta ser completado.

3.2 Instalación ISE Design Suite 13.1

Esta sección explica el proceso de instalación para todas las plataformas para ISE Design Suite 13.

1.- Asegúrese de que tiene los privilegios apropiados en el sistema que se instalará el software. Algunos componentes, como los controladores de cable de programación de dispositivos, requieren permisos de administrador.

2.- Cierre todos los programas antes de comenzar la instalación.

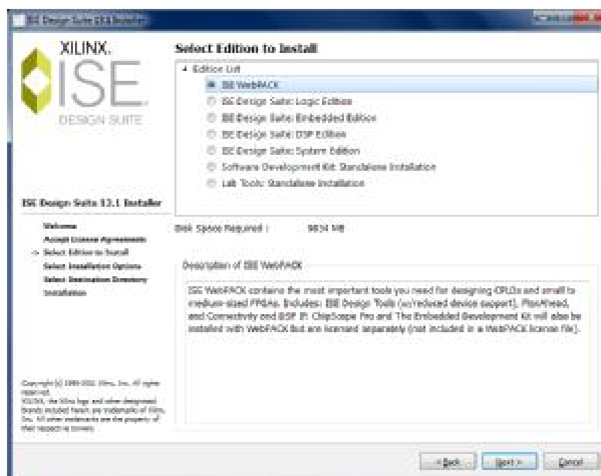
3.- Después de descargar y descomprimir el archivo Xilinx_ISE_PDS_13.1, corra el programa xsetup.exe

Biblioteca Documentos
Xilinx_ISE_DS_13.1_O.40d.1.1

Nombre	Tipo	Tamaño	Fecha de modifica...
Microsoft.VC90.CRT	Carpeta de...		12/04/2011 04:51 ...
Microsoft.VC90.MFC	Carpeta de...		12/04/2011 04:51 ...
bin	Carpeta de...		12/04/2011 04:51 ...
common	Carpeta de...		12/04/2011 04:51 ...
data	Carpeta de...		12/04/2011 04:51 ...
edk	Carpeta de...		12/04/2011 04:51 ...
idata	Carpeta de...		12/04/2011 04:55 ...
ise	Carpeta de...		12/04/2011 04:55 ...
labtools	Carpeta de...		12/04/2011 04:55 ...
lib	Carpeta de...		12/04/2011 04:55 ...
msg	Carpeta de...		12/04/2011 04:56 ...
planahead	Carpeta de...		12/04/2011 04:56 ...
sdk	Carpeta de...		12/04/2011 04:56 ...
sysgen	Carpeta de...		12/04/2011 04:56 ...
webpack	Carpeta de...		12/04/2011 04:56 ...
autorun	Informaci...	1 KB	03/02/2011 04:27 ...
xinfo	Archivo	1 KB	03/02/2011 04:27 ...
xsetup	Archivo	1 KB	03/02/2011 04:27 ...
xinfo	Aplicación	14 KB	03/02/2011 04:28 ...
xsetup	Aplicación	14 KB	03/02/2011 04:28 ...
fileset	Document...	1 KB	03/02/2011 04:29 ...

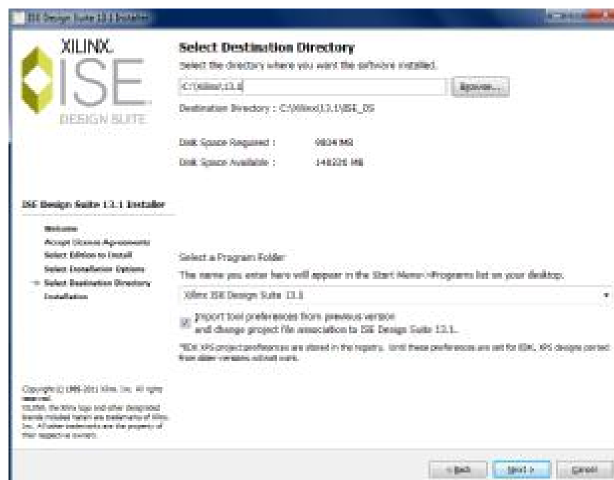
.4- Después de la pantalla de bienvenida hay dos pantallas para aceptar los acuerdos de la licencia. Necesario dar clic aceptando los términos y dar clic en siguiente (Next) en cada pantalla.

5.- Seleccionar el producto Xilinx a instalar. Para este caso seleccionar ISE WebPACK y dar clic en siguiente (Next).

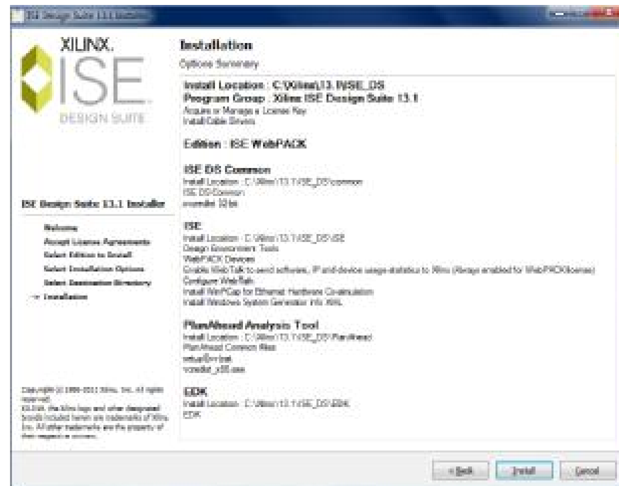


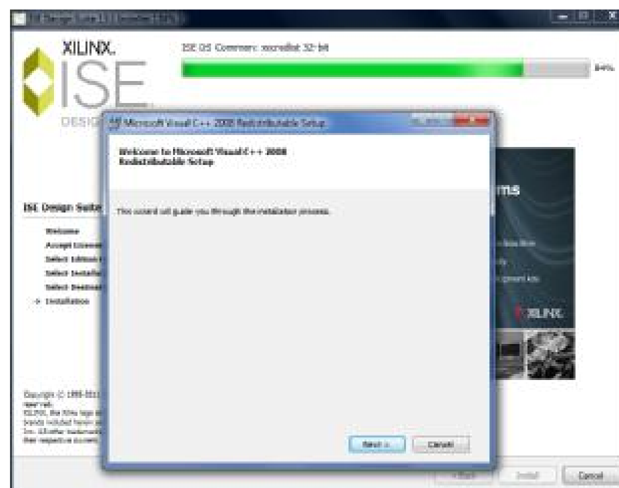
6.- Seleccionar las opciones de instalación

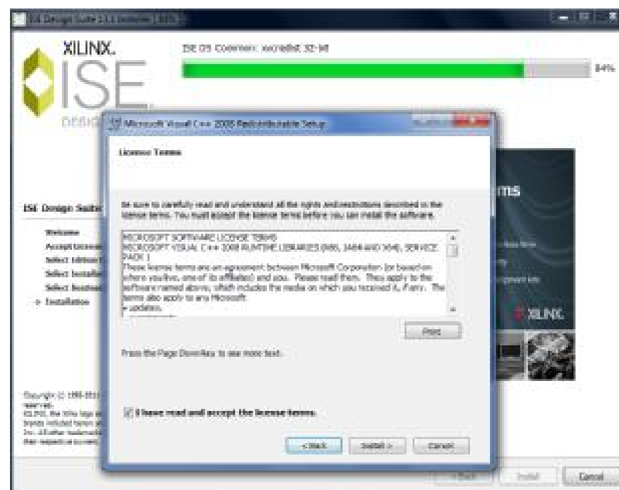
7.- Seleccionar Directorio de destino

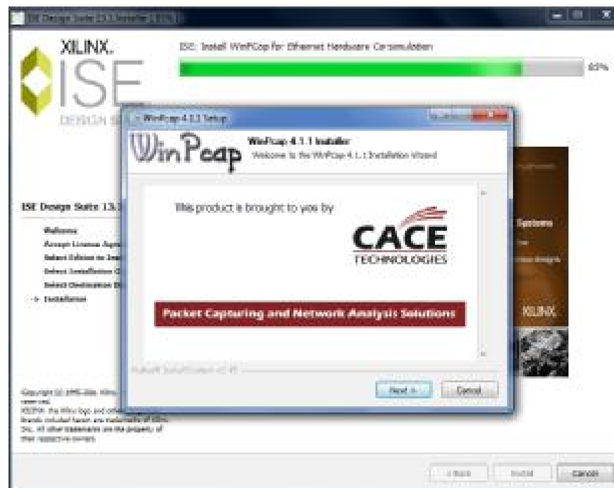


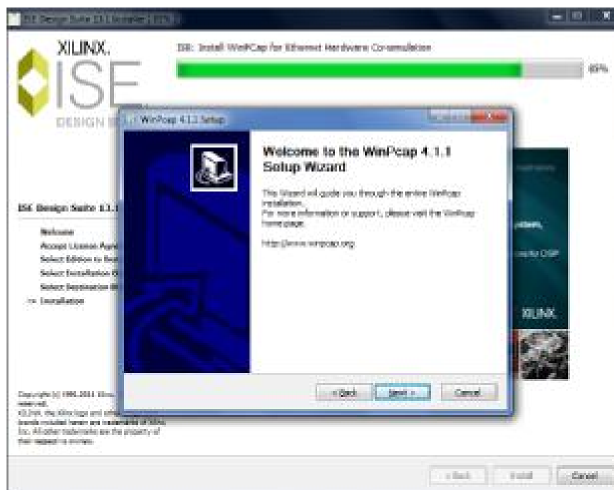
8.- Comenzar la instalación

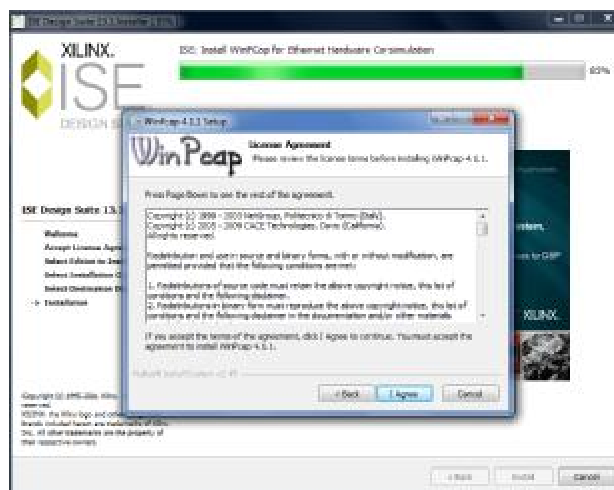


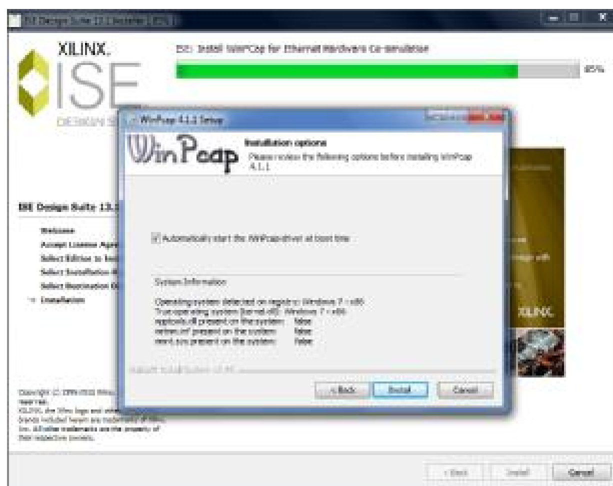


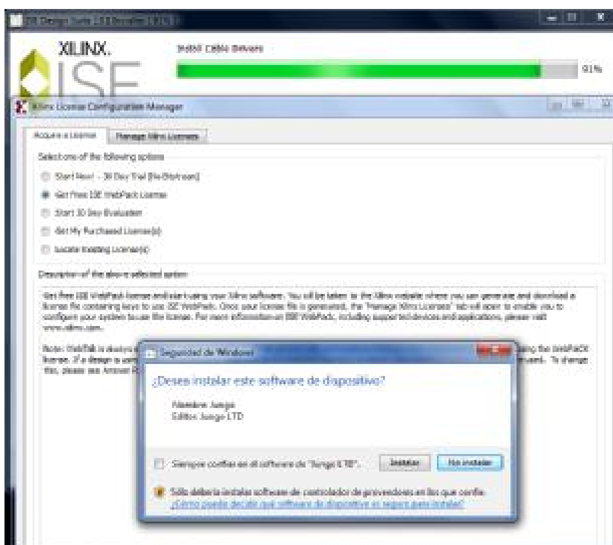


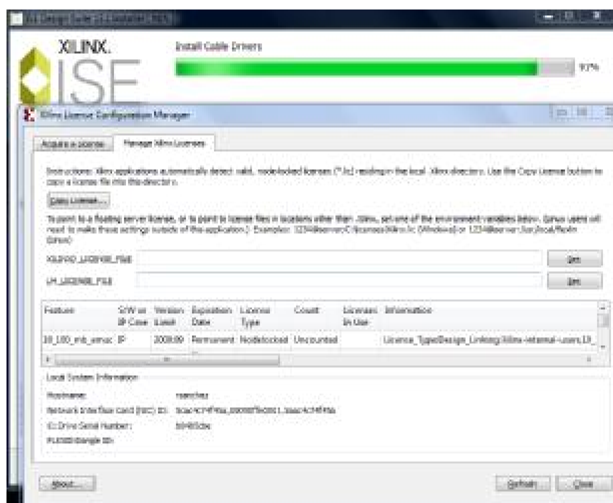

XILINX.
ISE
WinPcap
This product is brought to you by
CACE
TECHNOLOGIES
Packet Capturing and Network Analysis Solutions

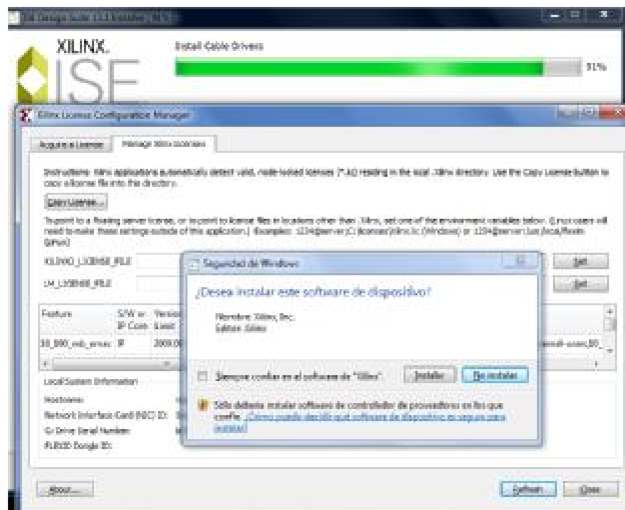

XILINX.
ISE
Welcome to the WinPcap 4.1.1
Setup Wizard
XILINX.


XILINX.
ISE
WinPcap
XILINX.

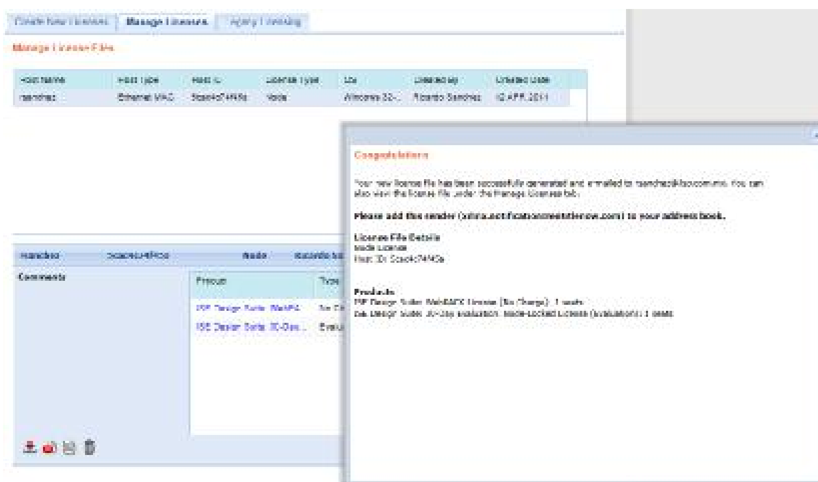

XILINX.
ISE
WinPcap
Installation options


XILINX.
ISE
91%
Seguridad de Windows
¿Desea instalar este software de dispositivo?

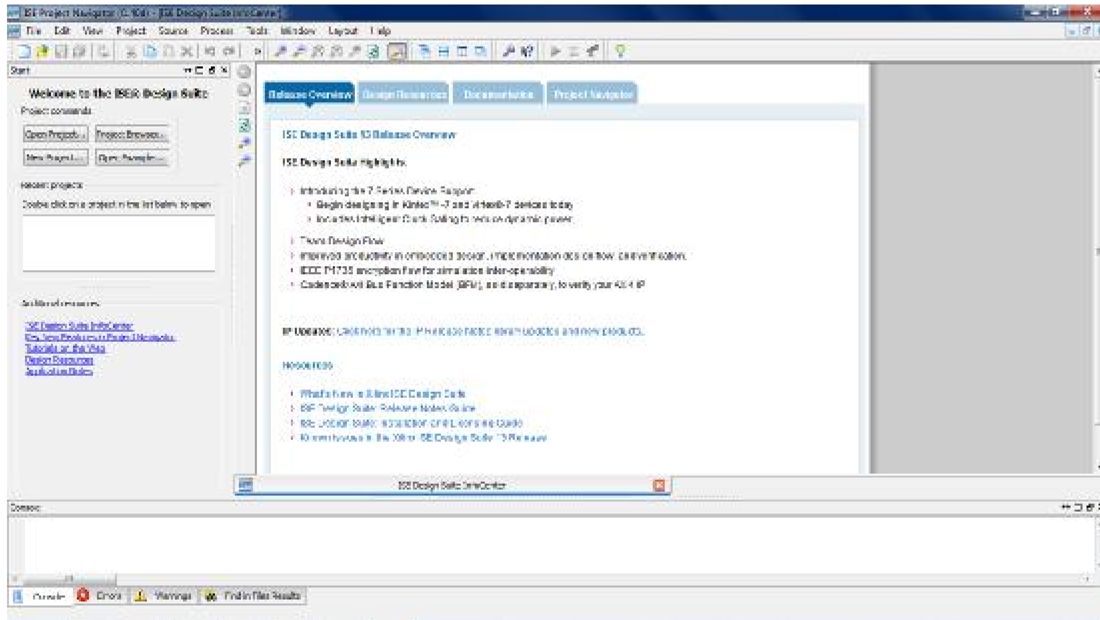

XILINX.
ISE
Install Cable Drivers



9.- Generar una licencia gratuita y agregarla al software ISE Design Suite.



10.- Ejecutar el programa IDE Design Suite y verificar que sea cargada la aplicación.



3.3 Instalación ADEPT 2

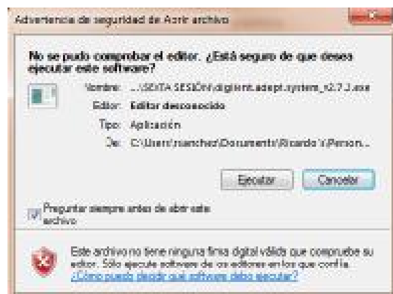
Esta sección explica el proceso de instalación de la utilería ADEPT 2.

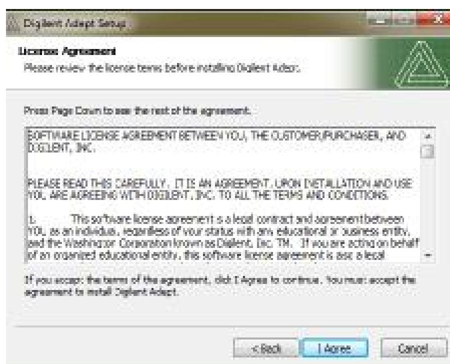
1.- Asegúrese de que tiene los privilegios apropiados en el sistema que se instalará el software.

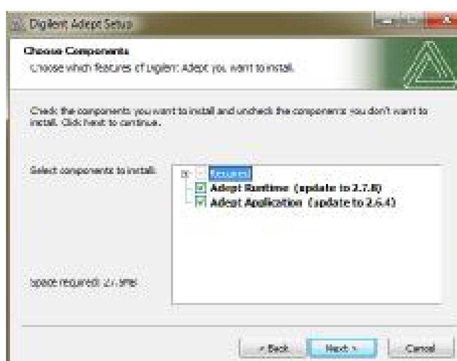
2.- Cierre todos los programas antes de comenzar la instalación y desde el explorador de Internet ingrese la siguiente dirección:

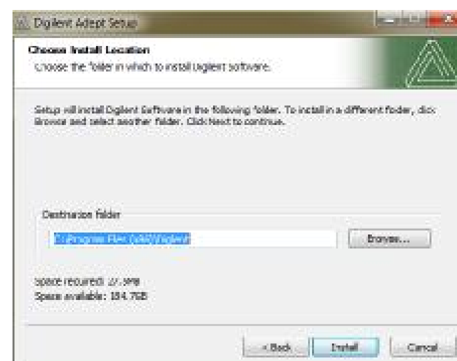
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,828&Prod=ADEPT2

3.- Después de descargar y descomprimir el archivo Adept 2.7.2 System, ejecute el programa digilent.adept.system_v2.7.2









Una vez completada la instalación, la tarjeta Basys 2 será reconocida al momento de ser conectada al equipo de cómputo.

Práctica 1
Uso del ambiente de desarrollo IDE, para el modelado

Objetivos.

Conocer el ambiente de desarrollo ISE creando un proyecto VHDL que tenga la función de una compuerta AND con entradas A, B y salida S.
Compilar el proyecto y verificar que no tenga errores de síntesis y diseño.

Desarrollo.

El primer paso es ejecutar el programa ISE Project Navigator, que se encuentra en nuestro escritorio, aparecerá la pantalla que se muestra en la figura 3.1.1

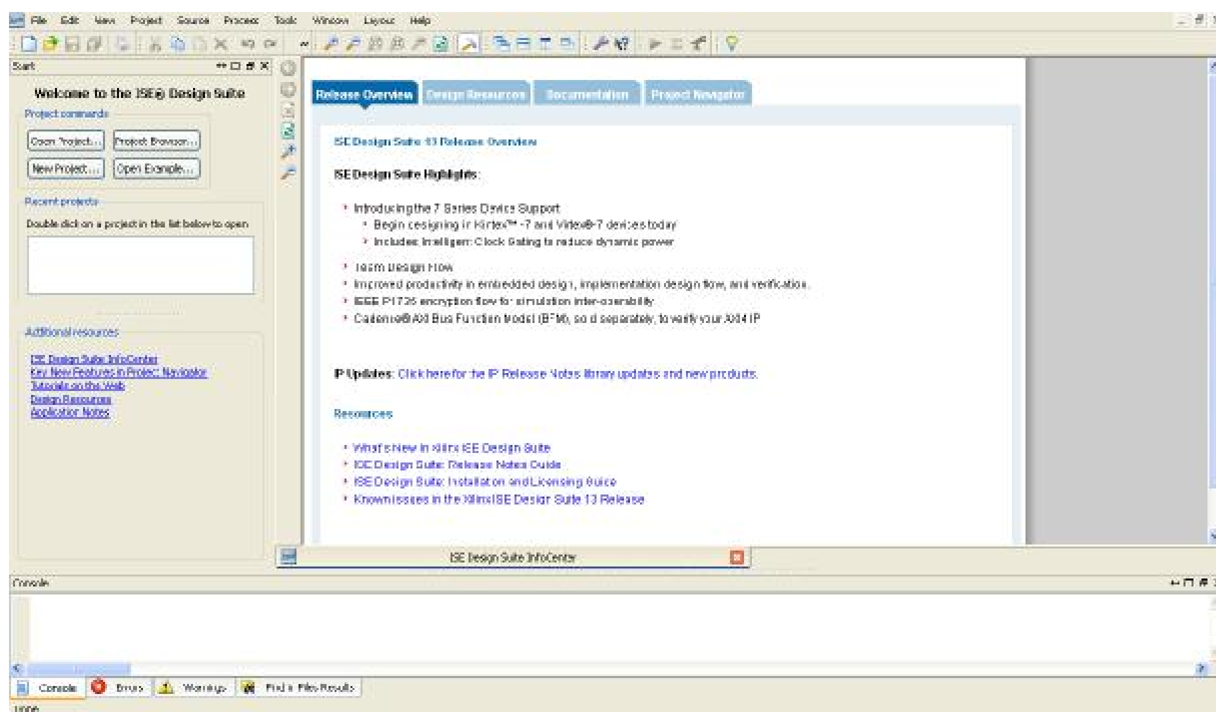


Figura 3.1.1 Ventana principal ISE Proyec Navigator.

Para crear un proyecto ir a barra de herramienta y seleccionar File -> New Project y Aceptar (ver figura 3.1.2

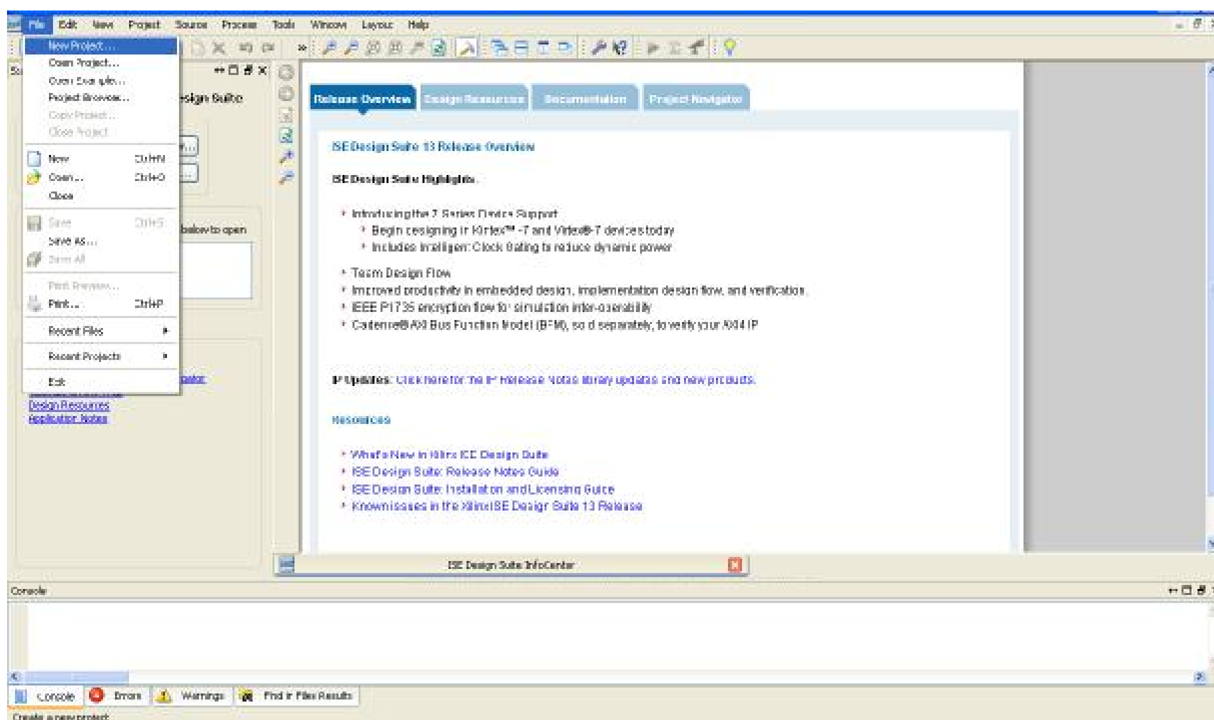


Figura 3.1.2 Creación de un nuevo proyecto.

En la siguiente pantalla nombraremos a nuestro proyecto en la opción de Name "PRÁCTICA1", escogeremos la ubicación en donde se guardará nuestro proyecto en Location, verificaremos que en la barra "Top-level source type" este la opción HDL como se muestra en la figura 3.1.3, posteriormente daremos aceptar en la opción Next.

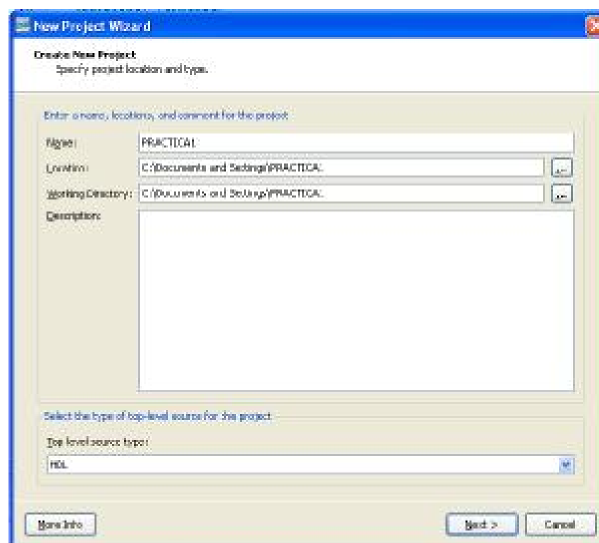


Figura 3.1.3 Nombre y ubciación de nuestro proyecto.

En la siguiente pantalla que aparece tenemos que llenar los datos de la FPGA que se utilizará. Nosotros utilizaremos los siguientes datos del FPGA de la tarjeta BASYS 2, son los siguientes:

Familia: Spartan3E
Dispositivo: XC3S100E
Empaquetado: CP132
Lenguaje: VHDL

Los ajustes deberan quedar como aparece en la figura 3.1.4, para finalizar daremos click en el icono “Next”.

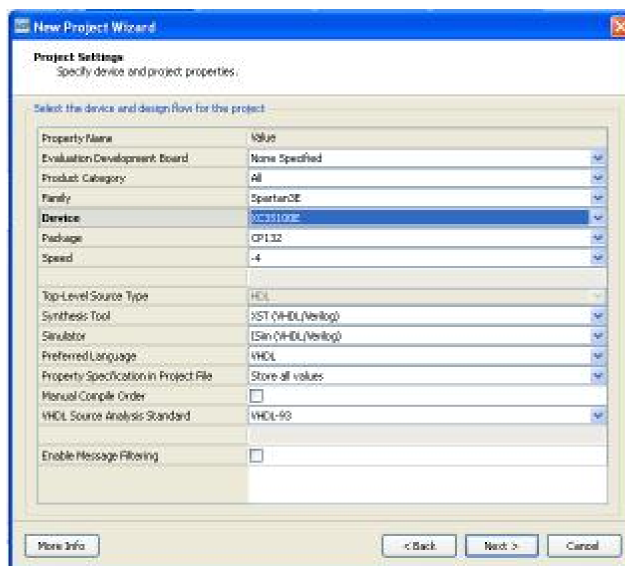


Figura 3.1.4 Ajuste del proyecto en base a la tarjeta BASYS 2.

La pantalla de la figura 3.1.5 sólo nos proporciona el resumen de las opciones que se eligieron en el paso anterior, daremos click en la opción “Finish”.

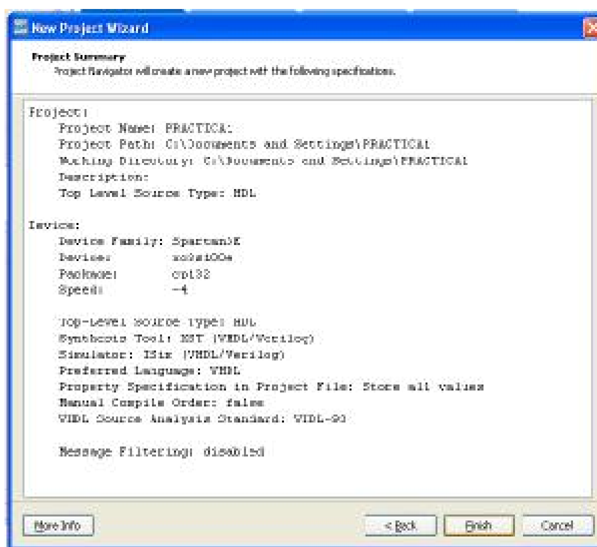


Figura 3.1.5 Resumen del proyecto.

Nuestro proyecto aparecerá en el cuadro izquierdo en el cuadro de “Design” con el nombre que elegimos, para este caso podemos observar nuestro proyecto “PRÁCTICA1” (ver figura 3.1.6).

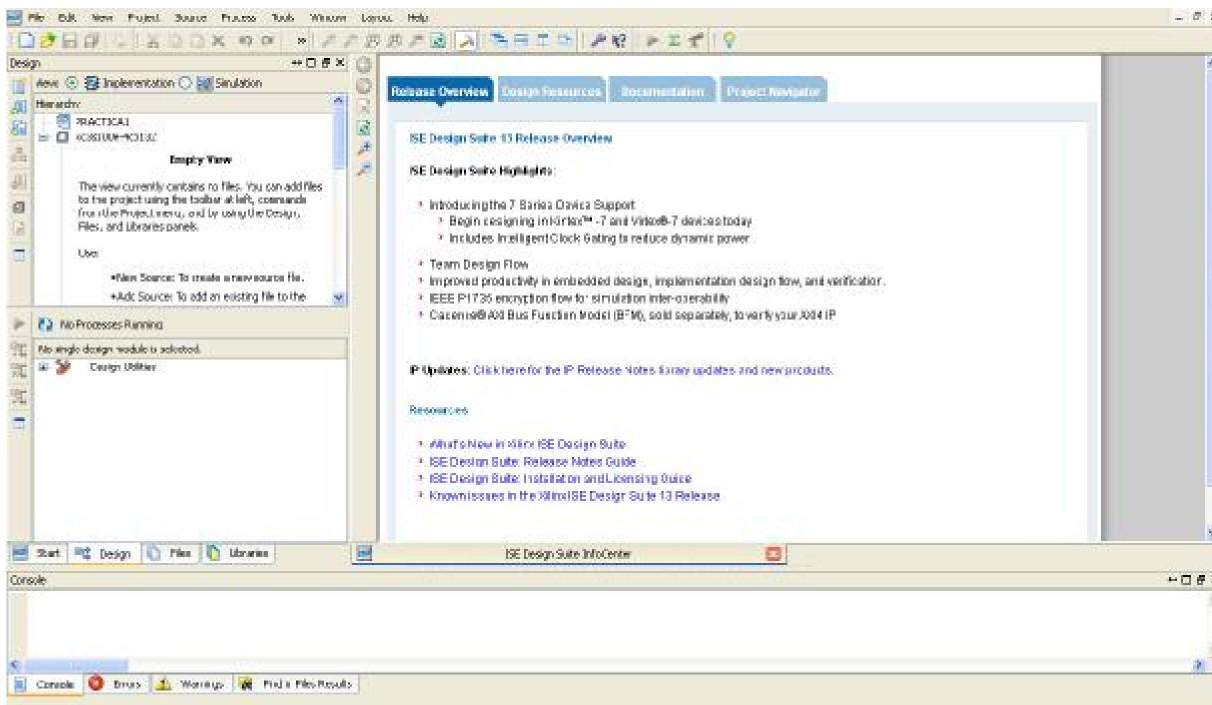


Figura 3.1.6 Ventana principal con el proyecto creado “PRÁCTICA1”.

Ahora crearemos un archivo VHDL el cual estará ligado a nuestro Proyecto, para esto nos posicionaremos en el Dispositivo XC3S100, daremos click derecho y seleccionamos "New Source", o podemos ir a barra de herramientas Project -> New Source (ver figura 3.1.7).

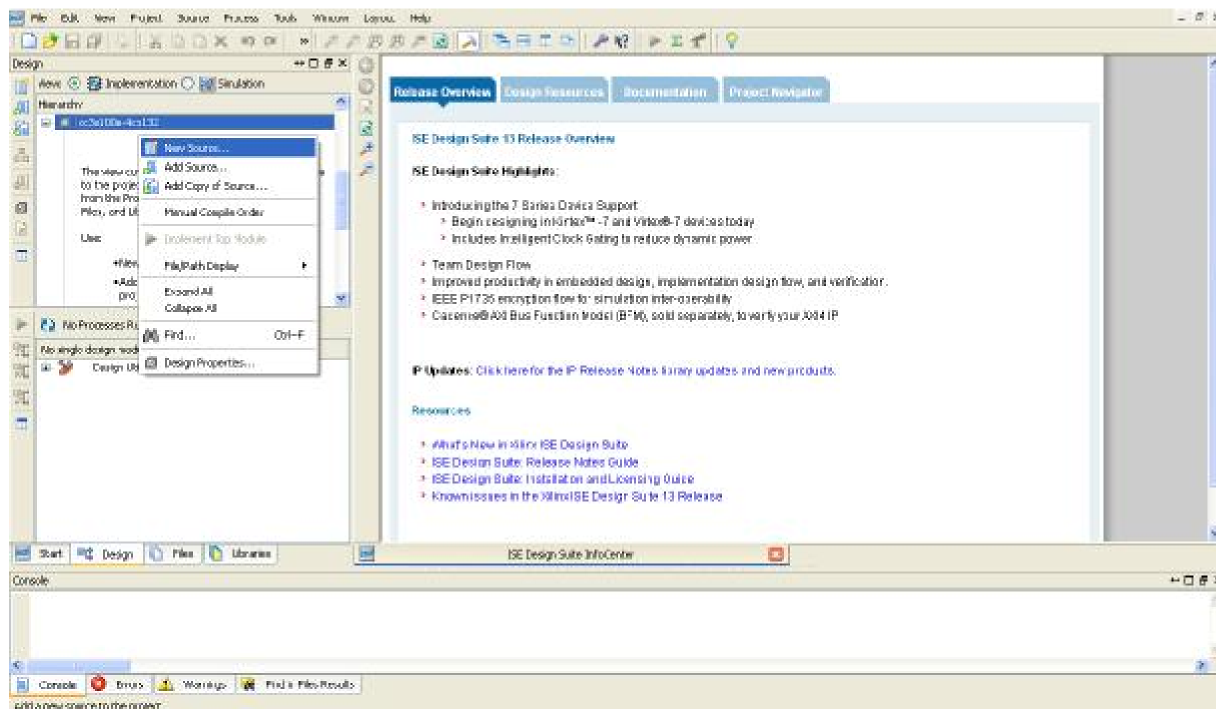


Figura 3.1.7 Crear una nueva fuente.

La figura 3.1.8 muestra los diferentes tipos de fuentes, para nuestro proyecto seleccionaremos VHDL Module con el nombre COMPUERTA, después marcaremos la casilla Add to Project y daremos aceptar en la opción Next.

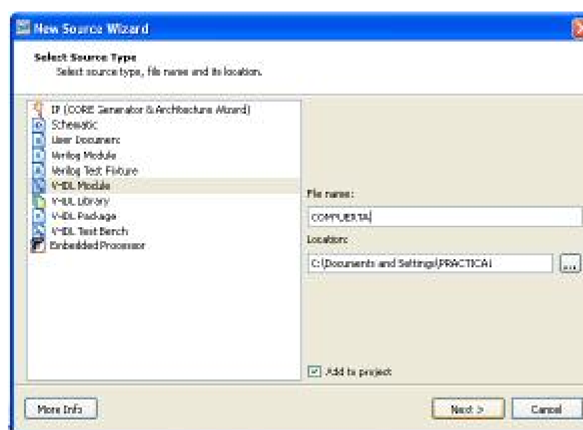


Figura 3.1.8 Selección de una fuente.

Ahora tenemos que definir los puertos de entradas y salidas de nuestra fuente, para COMPUERTA se necesitaran dos entradas A y B, y una salida S. Como las entradas y la salida son de un solo bit, no se habilita la opción de Bus, posteriormente daremos click en la opción Next (ver figura 3.1.9).

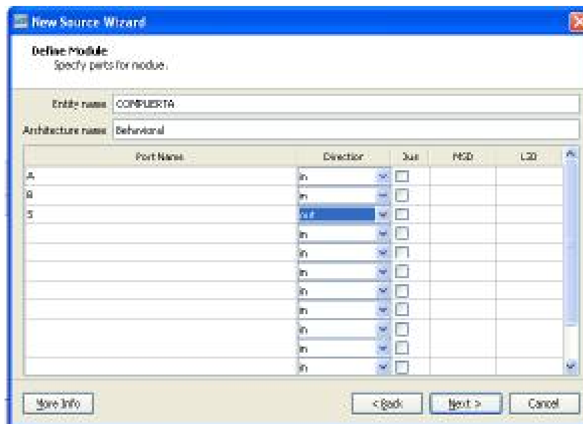


Figura 3.1.9 Definición de puertos.

La siguiente pantalla solo arroja el resumen de nuestro archivo VHDL que hemos creado en el paso anterior, daremos click en la opción Finish (ver figura 3.1.10).

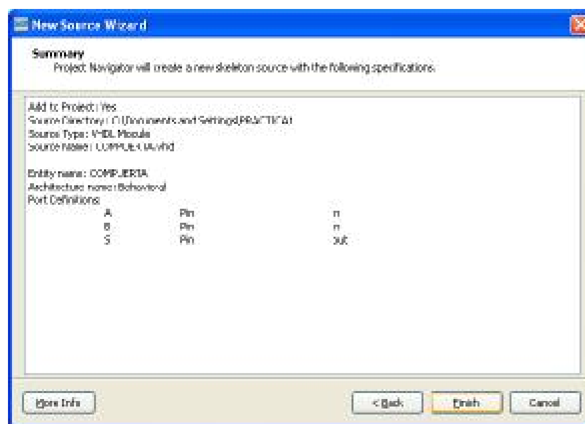


Figura 3.1.10 Resumen de la fuente.

Podemos ver que a nuestro proyecto “PRÁCTICA1” se le ha asociado un archivo VHDL llamado COMPUERTA y de lado derecho de la pantalla se genera un código que contiene las librerías principales, la entidad y las palabras reservadas para la arquitectura (ver figura 3.1.11).

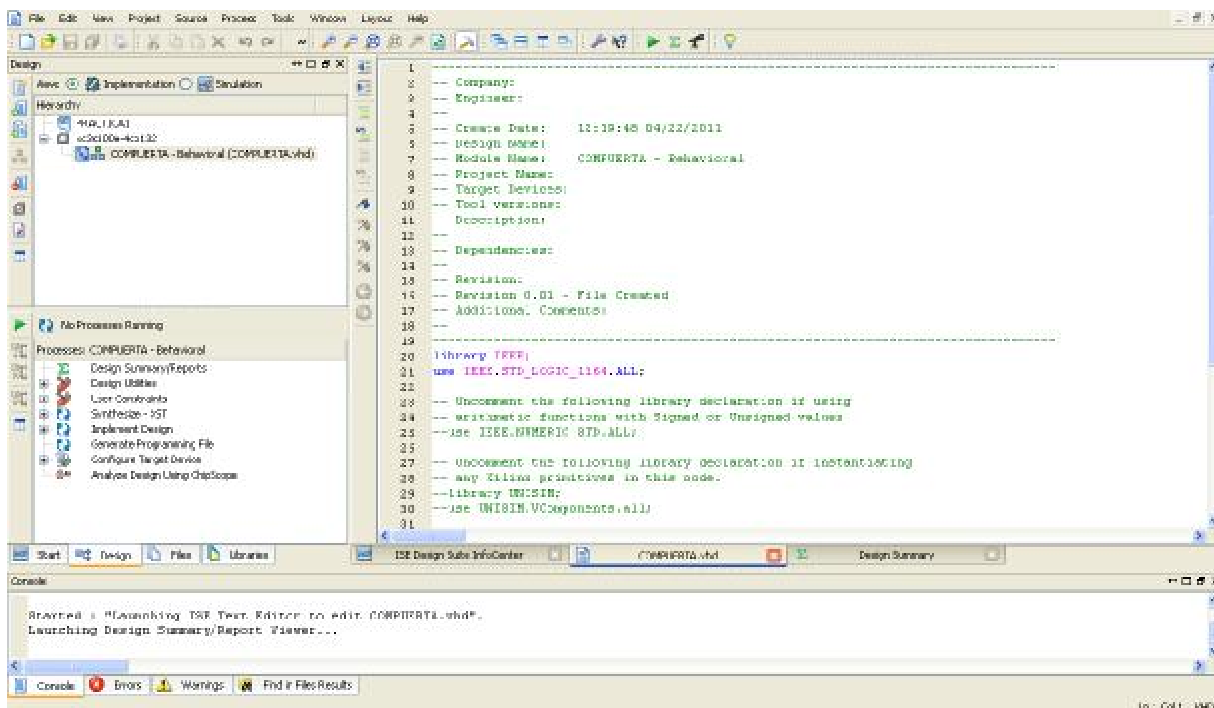


Figura 3.1.11 Asociacion de la fuente al proyecto.

Agregamos nuestro código para el diseño de una compuerta AND entre begin (después de architecture behavioral) y end Behavioral y guardamos los datos que hemos modificado (ver figura 3.1.12). El código para una and con dos entradas A y B y salida b es: S <= A and B;

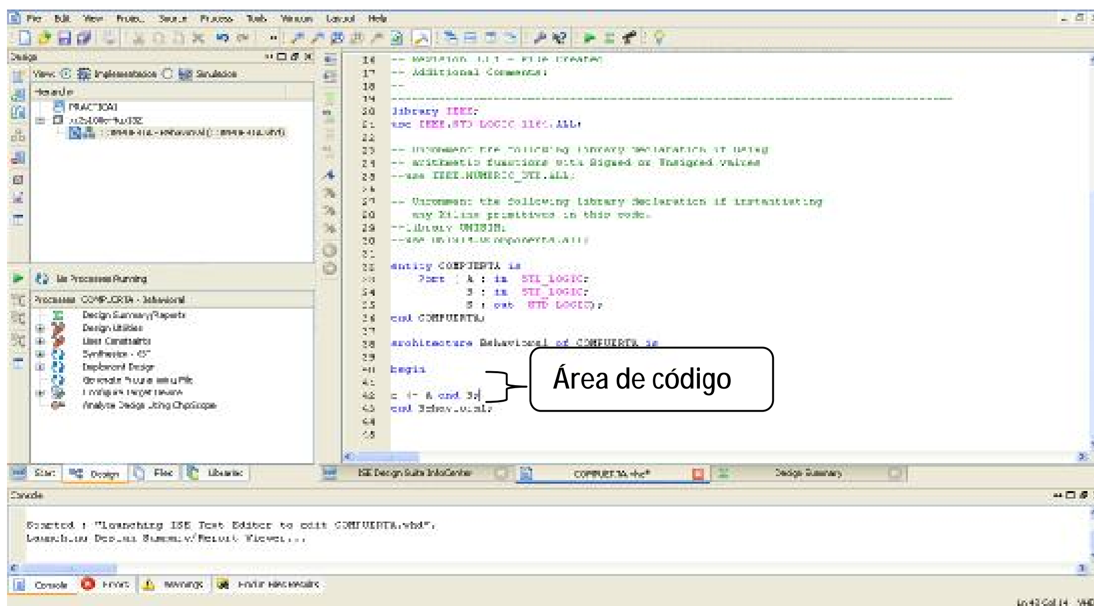


Figura 3.1.12 Implementación del código

Finalmente para compilar nuestro trabajo y revisar que no tenga errores daremos click en el icono de Implement Top Module, esta herramienta correrá los procesos Synthesize-XST e Implement Design (ver figura 3.1.13).

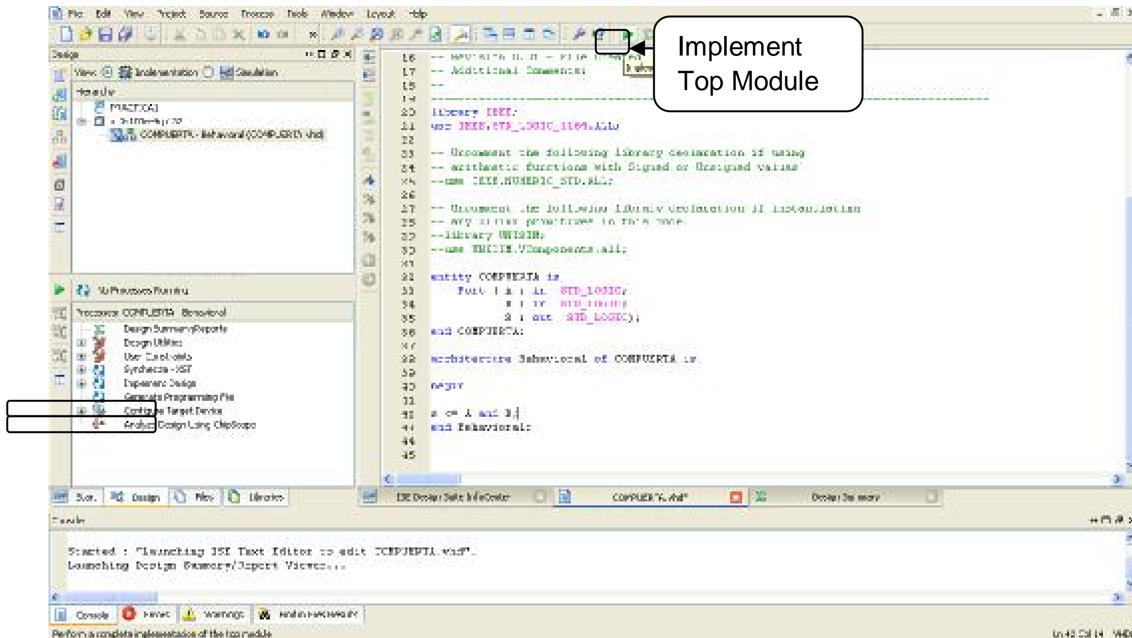


Figura 3.1.13 Compilacion del proyecto con Implement Top module.

Si nuestro proyecto no contiene errores veremos Synthesize- XST e Implement Design successfully con un círculo verde a su lado y en la ventana de Consola se desplegará que el proyecto fue completado satisfactoriamente como se muestra en la siguiente pantalla de la figura 3.1.14.

.

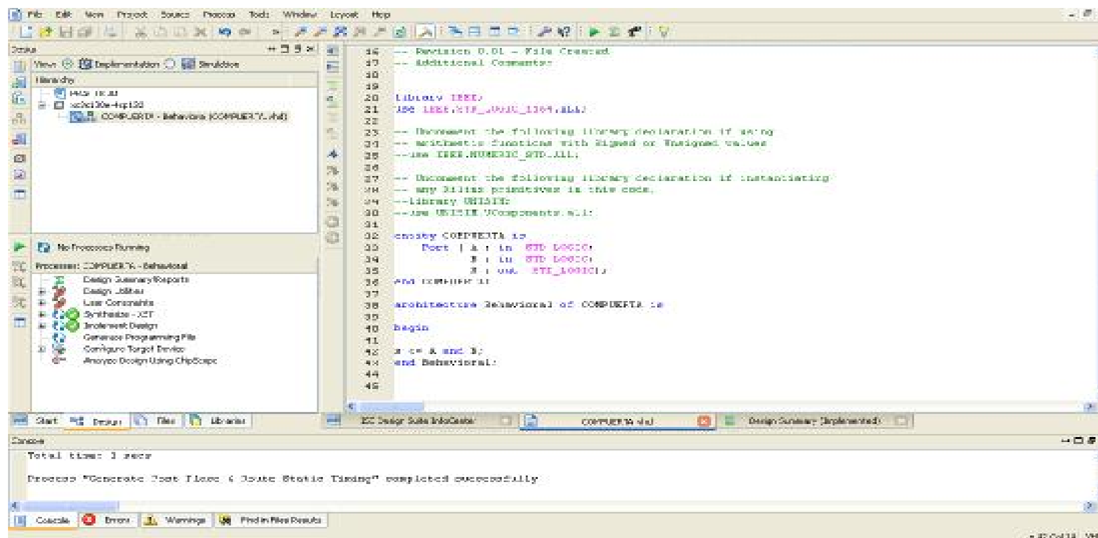


Figura 3.1.14 Compilación e Implementación exitosa.

Si existiera algún error los verificaríamos en la misma ventana de Consola, como ejemplo hemos quitado el símbolo ";" en nuestra descripción y al compilar nuevamente se genera un error en la línea 42 (ver figura 3.1.15).

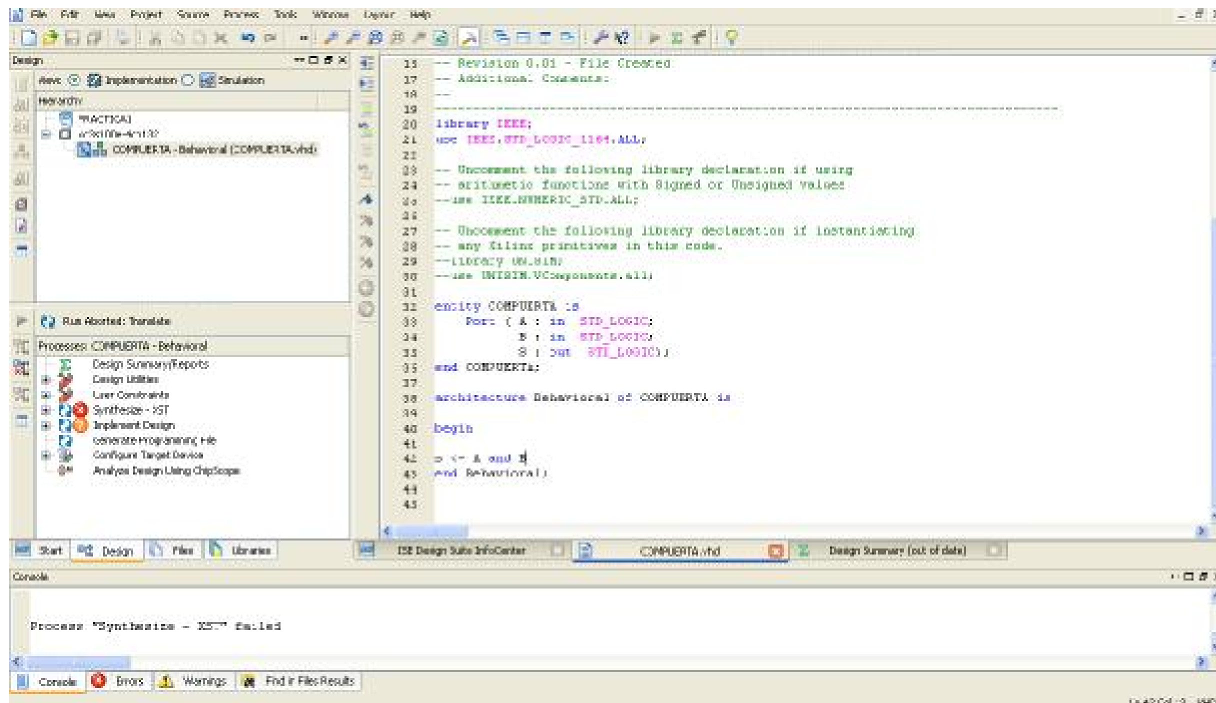


Figura 3.1.15 Compilación fallida

Es importante que nuestro proyecto no contenga errores ya que si existiera alguno no sería posible simularlo o bajarlo a nuestra FPGA.

Conclusiones.

Se familiarizó con el ambiente de desarrollo ISE, generando el proyecto nombrado "PRACTICA1". Al proyecto se le asocio una fuente VHDL con entradas A, B y salida S. La función de la fuente se basó en la ecuación lógica de una compuerta AND expresándola en código VHDL. Por último se verificó errores de compilación y diseño.

En general el ambiente tiene una interfaz amigable, la generación de proyectos es fácil y nos proporciona herramientas para la verificación del mismo.

Práctica 2
Uso del ambiente de desarrollo IDE, para simulación

Objetivo.

Simular un proyecto VHDL utilizando la herramienta ISim del software de Xilinx ISE Project Navigator V13.1.
Corroborar la simulación con la teoría de una compuerta AND.

Desarrollo.

Abriremos el proyecto nombrado “PRACTICA1” para simularla con la herramienta ISim. El proyecto no deberá tener errores de compilación y diseño, esto lo comprobamos corriendo el top module (ver figura 3.1.13) de la Práctica 1.

Si el análisis del diseño y código son correctos nos mostrará en la consola que el proceso se completó exitosamente además de marcar con una paloma los módulos Shintethize – XST y Implement Design ver figura 3.1.14 de la Práctica 1.

En la parte superior izquierda encontraremos la ventana de diseño daremos click en la parte de simulación para que nos cambie a la ventana de fuentes a simular como se muestra en la figura 3.2.1.

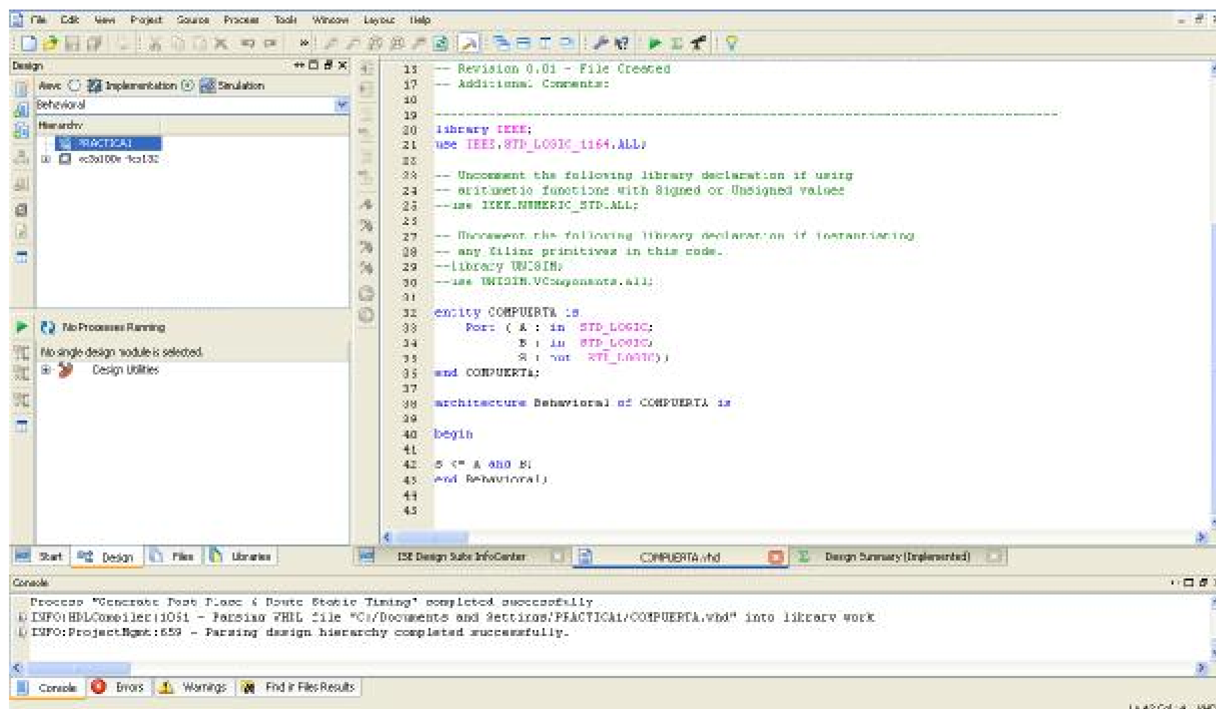


Figura 3.2.1 Ventana de fuentes a simular

En esta misma ventana seleccionamos la fuente que simularemos, en este caso sólo es el módulo VHDL COMPUERTA que anteriormente habíamos asociado a nuestro proyecto, que es el que se muestra en la figura 3.2.2.

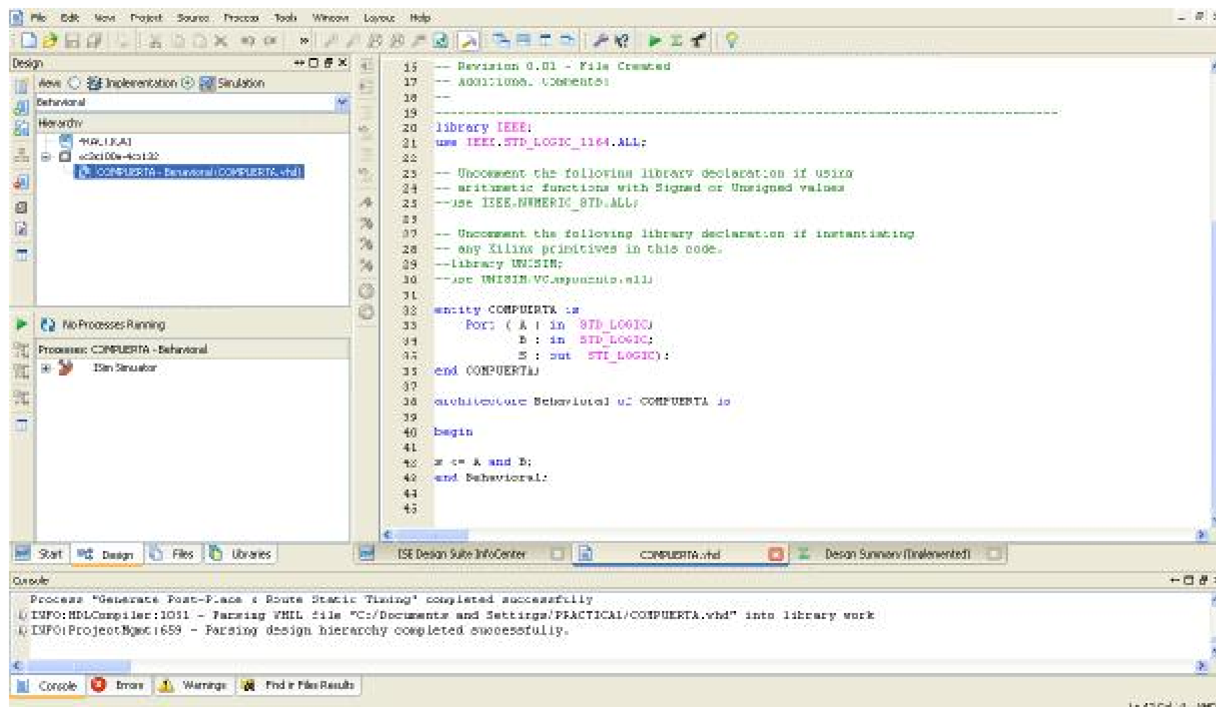


Figura 3.2.2 Selección de fuente a simular.

Daremos click derecho en Simulate Behavioral Model ->Run, como se muestra en la figura 3.2.3, para que se ejecute el ISim, que es el simulador que trae integrado ISE.

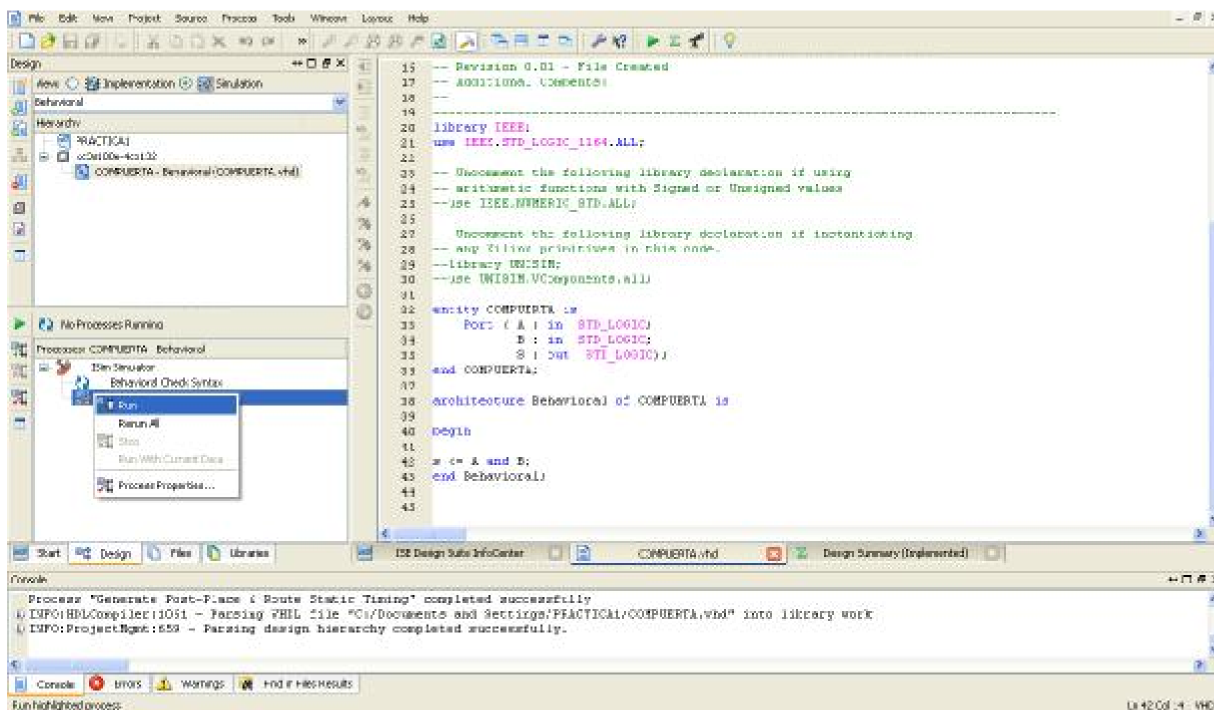


Figura 3.2.3 Apertura del Simulador.

Después de cargar los datos necesarios, se abrirá el ISim como se muestra en la figura 3.2.4, en esta ventana observaremos los puertos que declaramos en el módulo VHDL, así como sus propiedades.

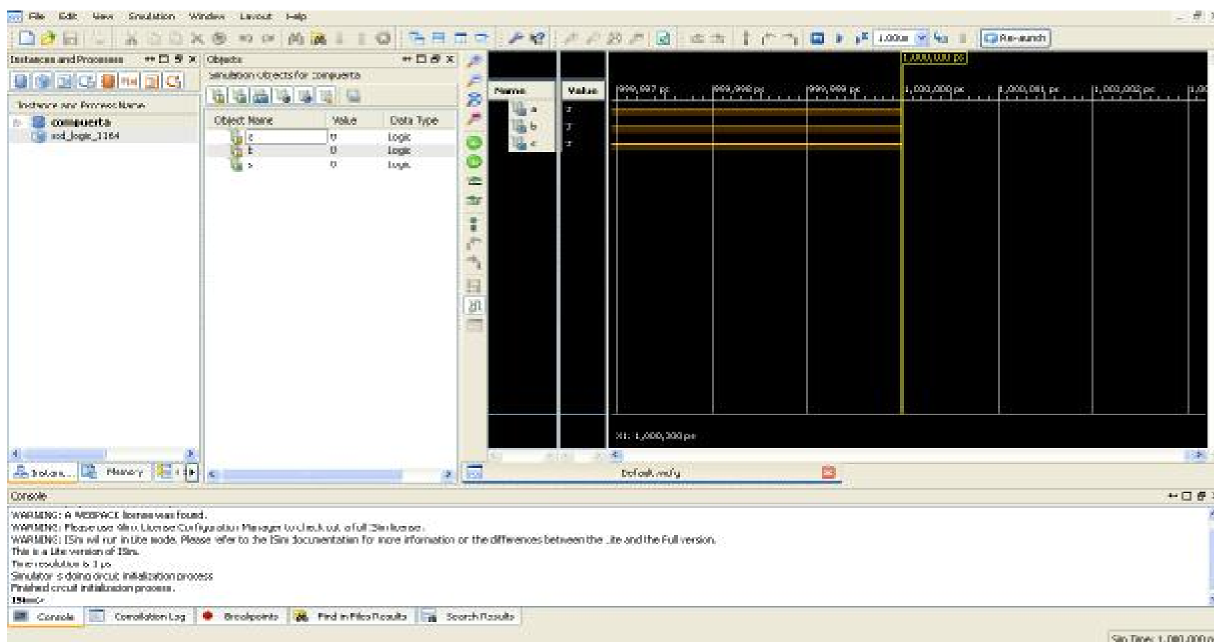
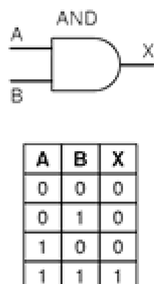


Figura 3.2.4 Simulador ISim.

Para comprobar que efectivamente nuestro proyecto funciona, tendremos que simular con los datos de la tabla de verdad de la compuerta AND como se muestra en la figura 3.2.5, para lograr esto utilizaremos dos señales de reloj desfasadas con un periodo de 40 ns, ciclo de trabajo del 50%.



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Figura 3.2.5 Tabla de verdad compuerta AND.

Las características de las señales se ingresan en el ISim como se muestra en la figura 3.2.6, dando click derecho sobre cualquiera de las señales de entrada y seleccionando Force Clock.

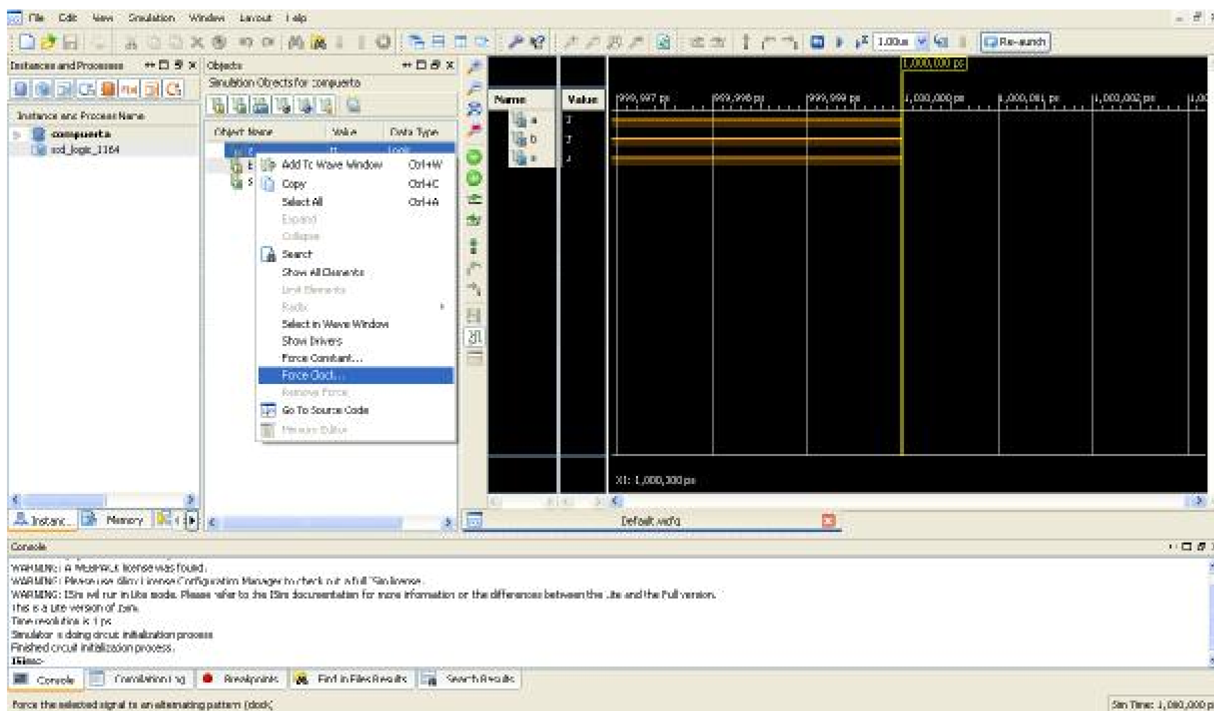


Figura 3.2.6 Configuración de señales de entrada

Después procederemos a configurar las características de cada señal como se muestra en la figura 3.2.7 y 3.2.8, las dos señales tienen periodo de 40 ns y un cilclo de trabajo del 50 %, la segunda señal iniciará 10 ns después para poder apreciar los valores de salida en la última señal, y el análisis terminará en 300ns.

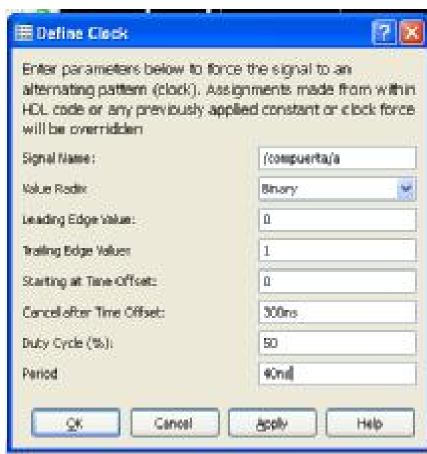


Figura 3.2.7 Entrada a

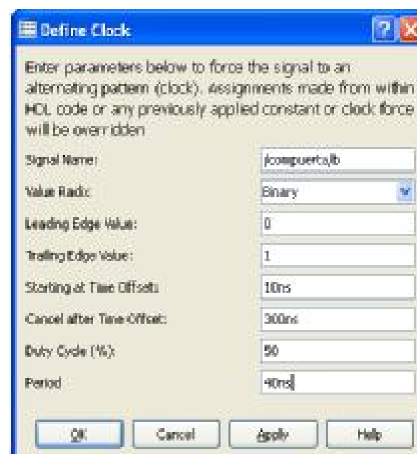


Figura 3.2.8 Entrada b

Después de aplicar los valores a las señales daremos click en Run All (ver figura 3.2.9).

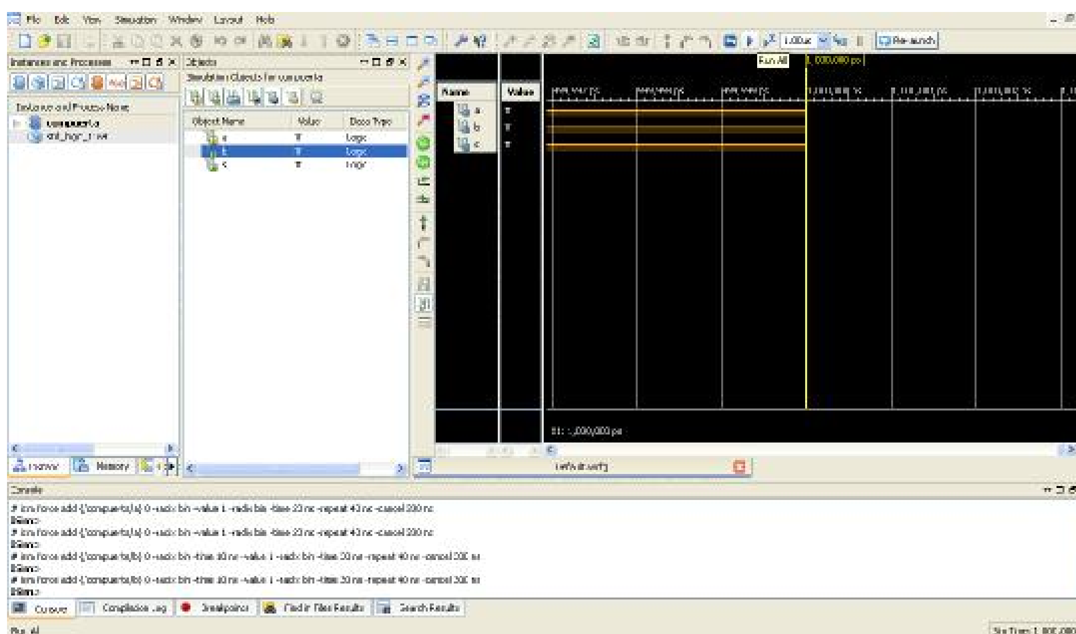


Figura 3.2.10 Inicio de Simulación

Y finalmente nos muestra la simulación de las dos señales de entrada y la de salida, donde podemos apreciar que en la señal de salida se cumple la tabla de verdad de la compuerta AND (ver figura 3.2.10).

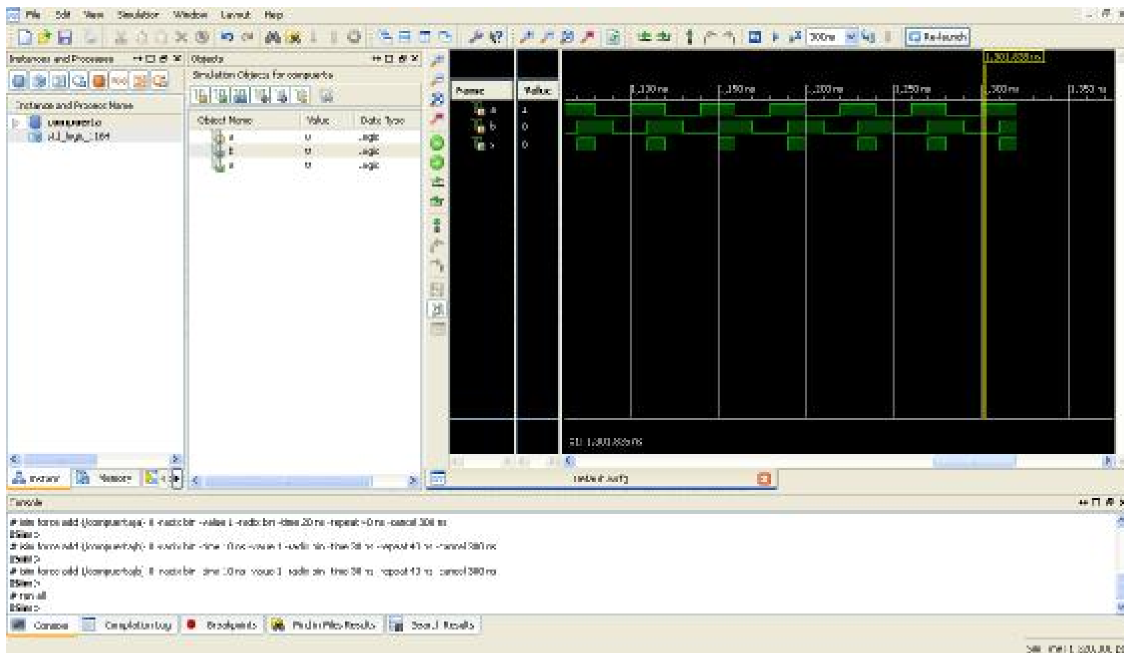


Figura 3.2.10 Resultado de la simulación.

Se puede observar en la ventana simulación que unicamente la salida S se encuentre en alto cuando las entradas A y B tambien se encuentran en alto. Se observa el comportamiento de la compuerta AND.

Conclusiones.

Se simuló con éxito la práctica llamada “PRÁCTICA 1”, la cual tuvo como función la implementación de una compuerta AND con entradas A, B y salida S.
Para simular cualquier proyecto primero se debe verificar que no haya errores en compilación y diseño, es requisito de ISim.
Las entradas se pueden configurar en la venta ISim con la opción “force clock”. También se puede configurar el tiempo de simulación y línea del tiempo para observar mejor los resultados.

Práctica 3
Uso de la tarjeta de desarrollo basada en FPGA (BASYS 2)

Objetivos.

Asignación de pines del proyecto “PRÁCTICA1” con el sub-programa “PlanAhead” de la ISE Design Suite de Xilinx.
Se generara un archivo de programa con extensión .BIT en ISE, el cual se descargará a la tarjeta BASYS 2 usando el programa “Adept”.

Desarrollo.

Se abrirá el proyecto nombrado “PRÁCTICA1” en la ventana de implementación (ver figura 3.3.1).

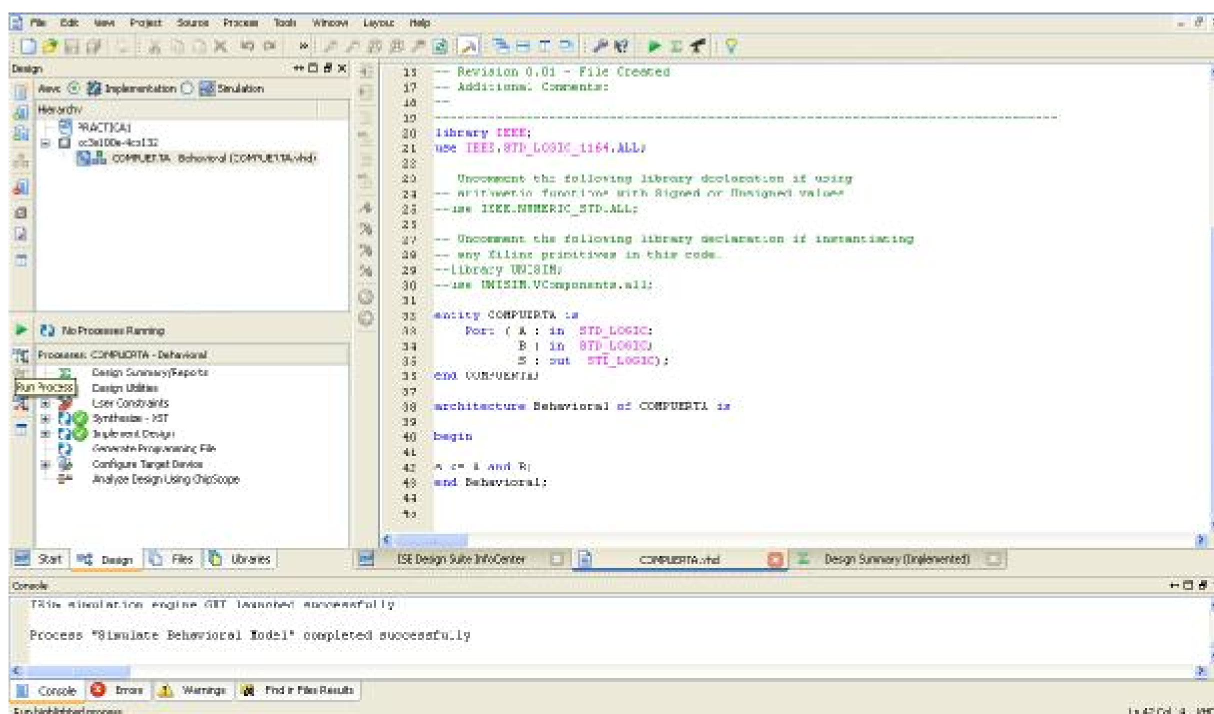


Figura 3.3.1 Proyecto PRÁCTICA1 ventana de implementación

Se hará la asignación de pines de entrada y salida, con la opción “User Constraints”.
Para lograrlo desplegaremos el submenú de “User Constraints” y daremos doble click en “I/O Planning (PlanAhead)- Pre- Synthesis” (ver figura 3.3.2).

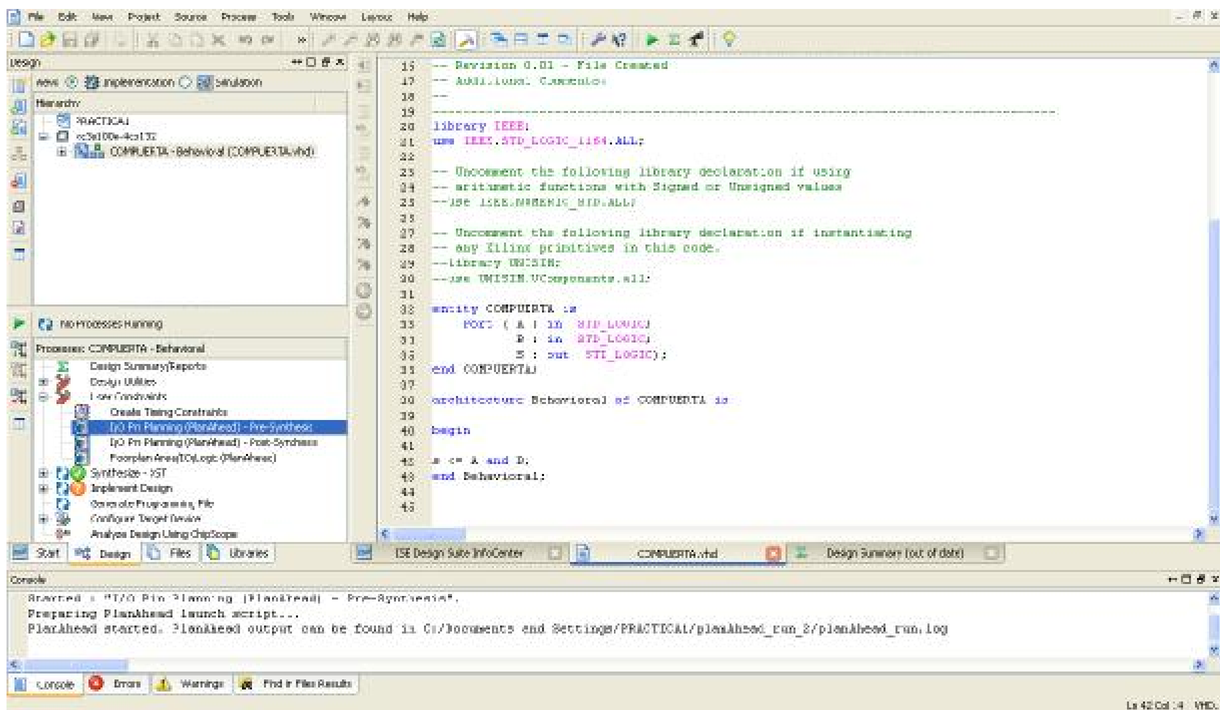


Figura 3.3.2 Asignación de pines

Se abrirá el programa PlanAhead, donde podremos seleccionar los pines de entrada y salida (ver figura 3.3.3).

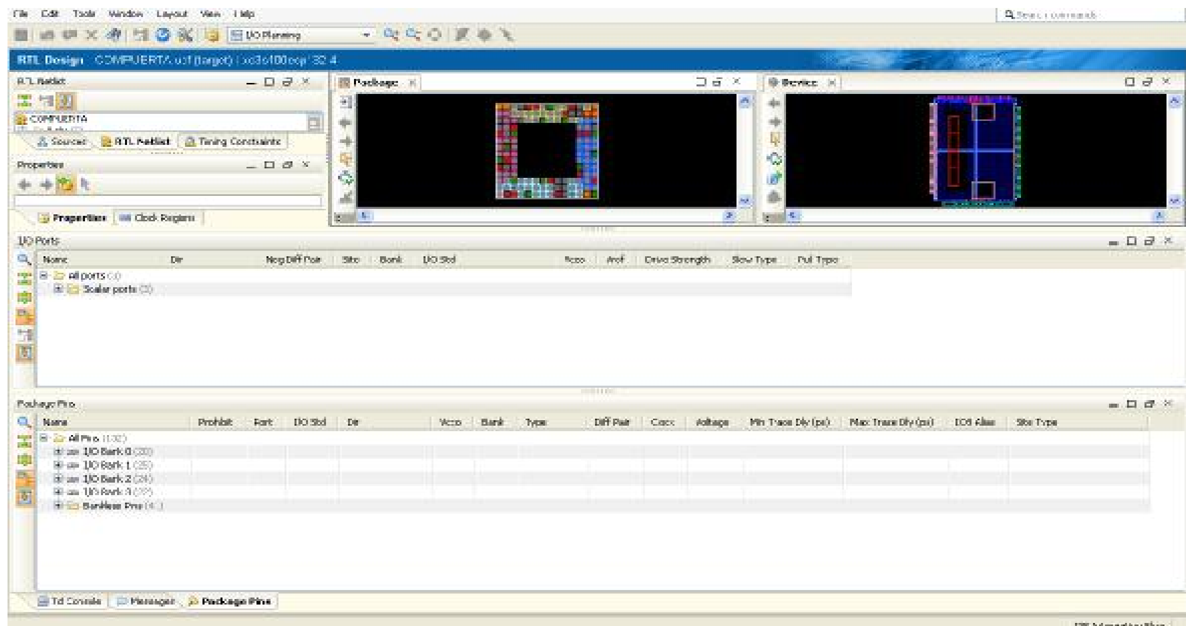


Figura 3.3.3 PlanAhead

Para elegir los pines de entrada y salida, se expandirá el menú “Scalar ports” (ver figura 3.3.4).

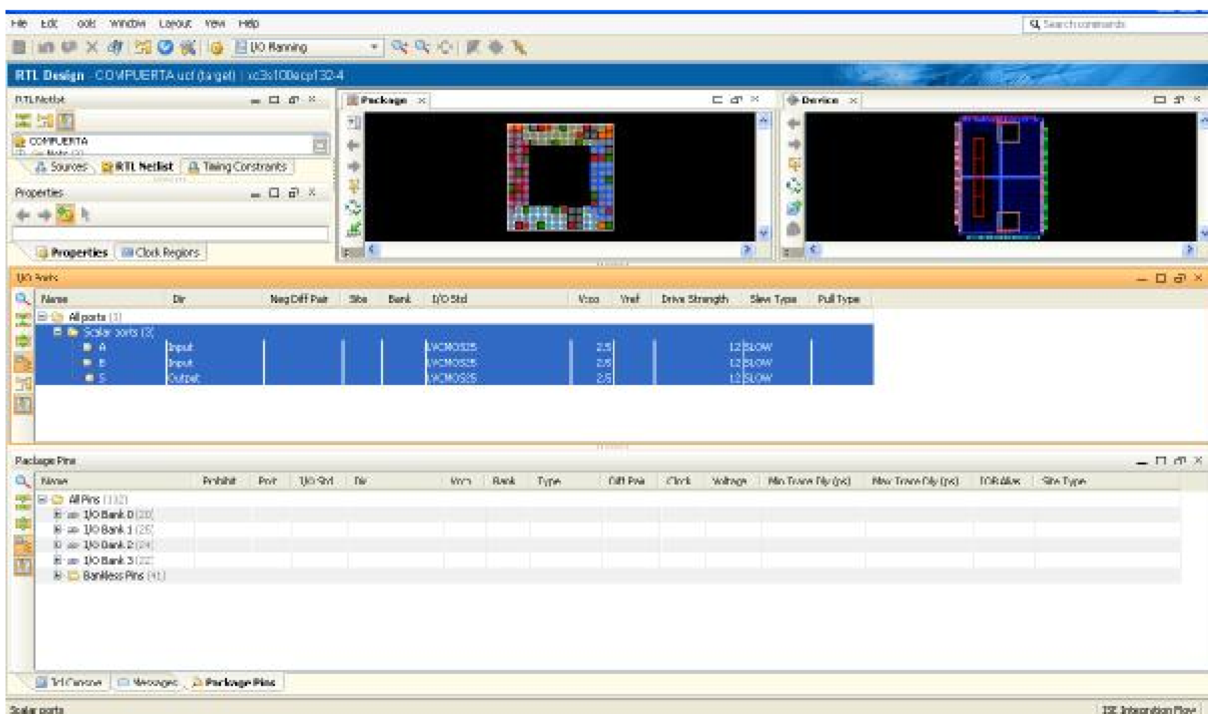


Figura 3.3.4 Puertos de entra y salida.

Para elección de entradas y salida nos basaremos en el diagrama de nuestra tarjeta (ver figura 3.3.5).

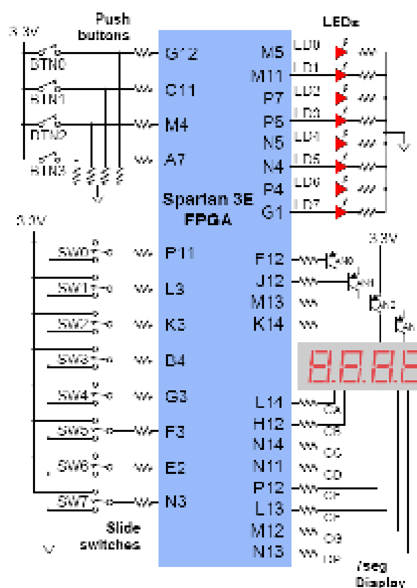


Figura 3.3.5 Diagrama de entradas y salidas BASYS 2

Para nuestro proyecto elegiremos como entrada los SW0 y SW1, y para salida el LD0. Vemos en nuestro diagrama que estas entradas y salidas están mapeadas con pines al FPGA. Entonces nuestros pines quedarán de la siguiente manera:

Entrada A = P11
Entrada B = L3
Salida S = M5.

Mapeamos estos pines en la ventada de I/O de "PlanAhead" en la columna SITE, para cada entrada y salida (ver figura 3.3.6).

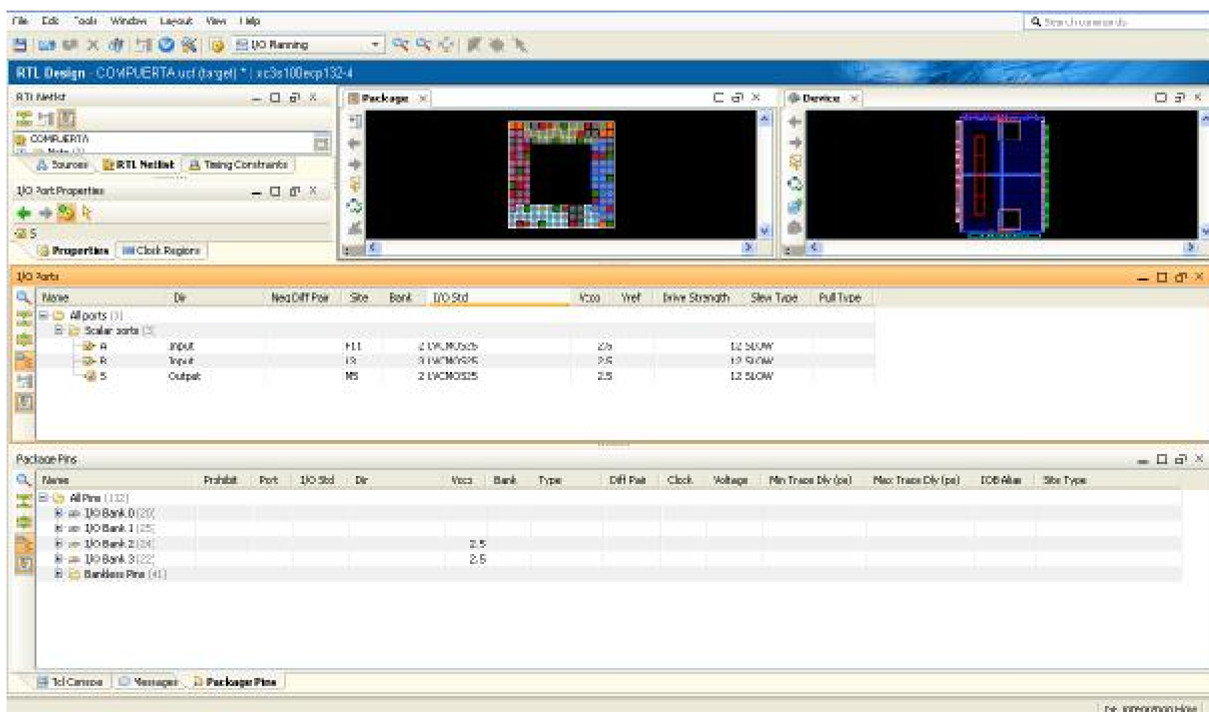


Figura 3.3.6 Mapeo de pines

Ahora tendremos que guardas los cambios dando click en "File" y después en "Save Design" (ver figura 3.3.7).

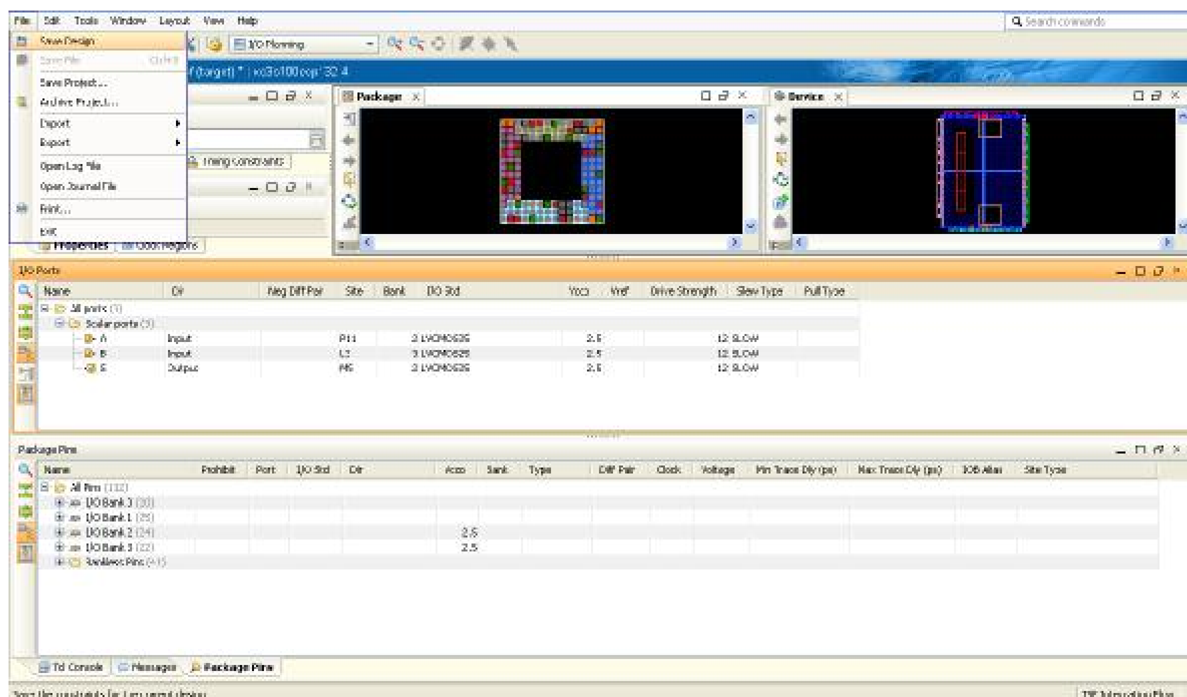


Figura 3.3.7 Guardar diseño.

Después de guardar el diseño de entradas y salidas, volveremos a “ISE Project Navigator”, donde utilizaremos “Top module” para verificar que nuestro proyecto no tenga errores de diseño (ver figura 3.3.8).

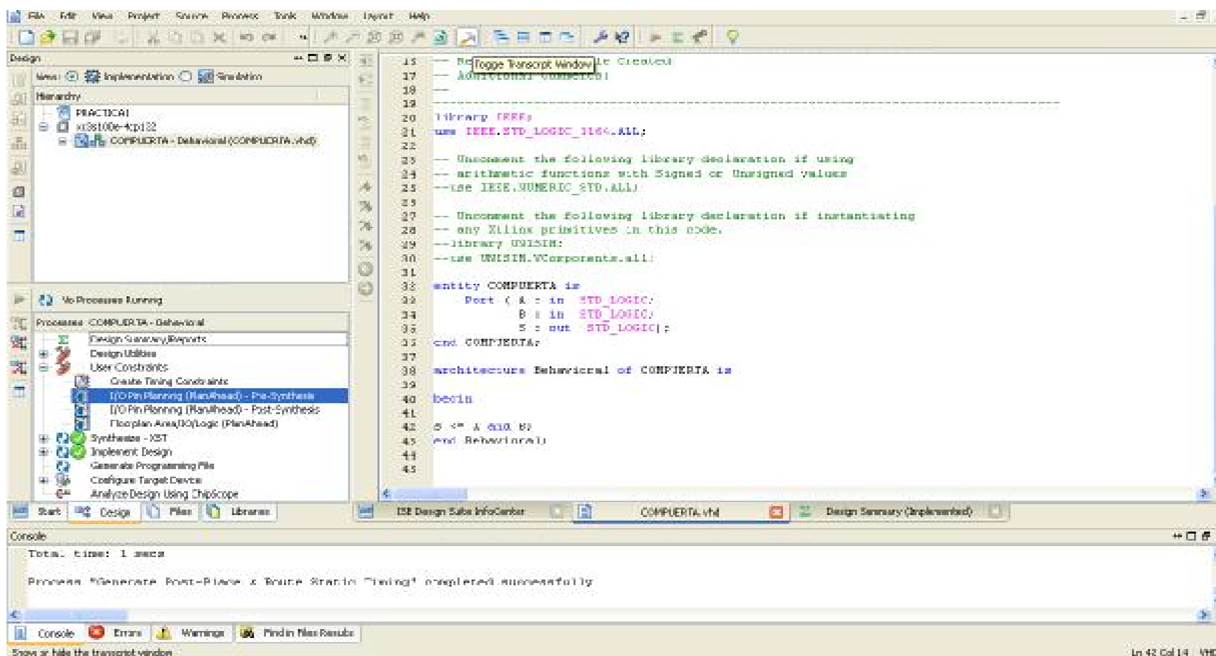


Figura 3.3.8 Verificación del diseño

Después de una implementación exitosa y antes de generar el programa .bit, debemos dar click derecho en “Generate Programming File” y después en “Process Properties” (ver figura 3.3.9).

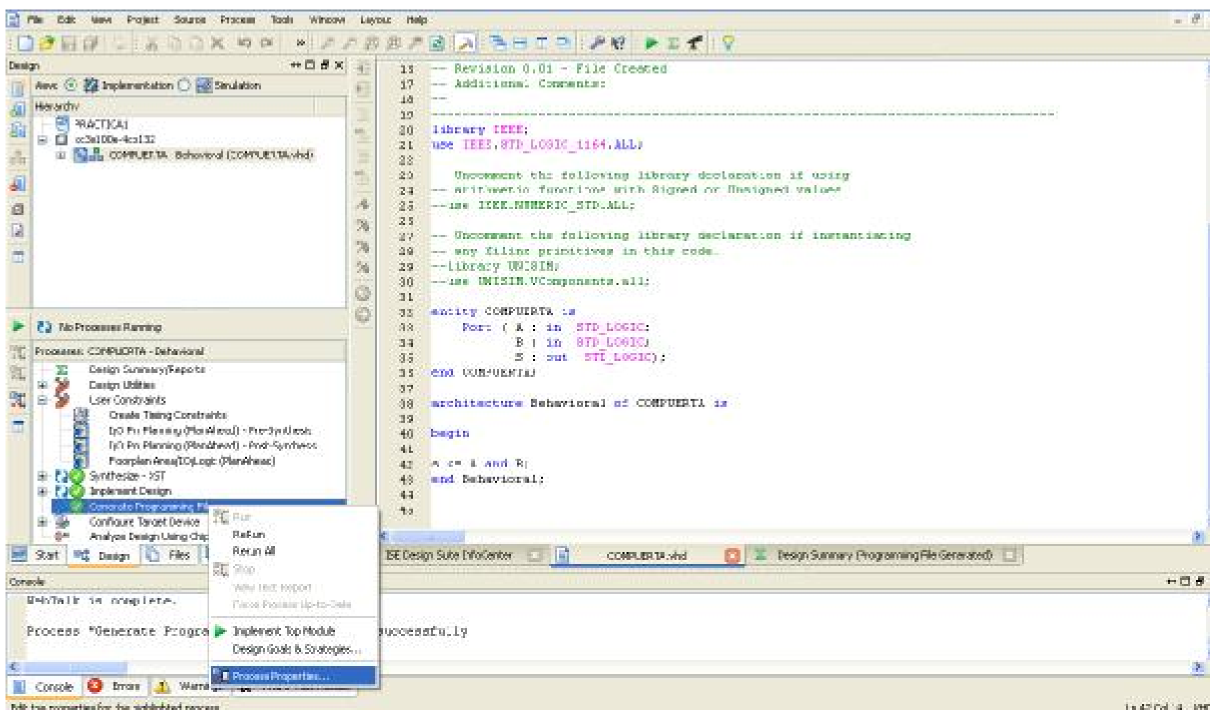


Figura 3.3.9 Propiedades del proceso

En la pantalla seleccionamos “Startup Options” y cambiaremos la opción “FPGA Star-Up Clock” de “CCLK” a “JTAG clock” y aplicaremos el cambio (ver figura 3.3.10). Esta configuración nos permitirá bajar el archivo a nuestra tarjeta Basys”.

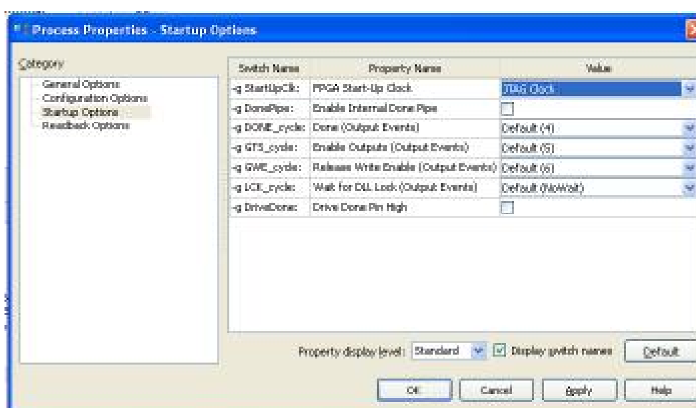


Figura 3.3.10 StartUp Options

Generaremos el programa .BIT, dando doble click en “Generate Programming File” (ver figura 3.3.11). Si el archivo no hay errores el archivo .BIT se llamara como la fuente, en este caso compuert.bit, y se almacenará en la carpeta del proyecto.

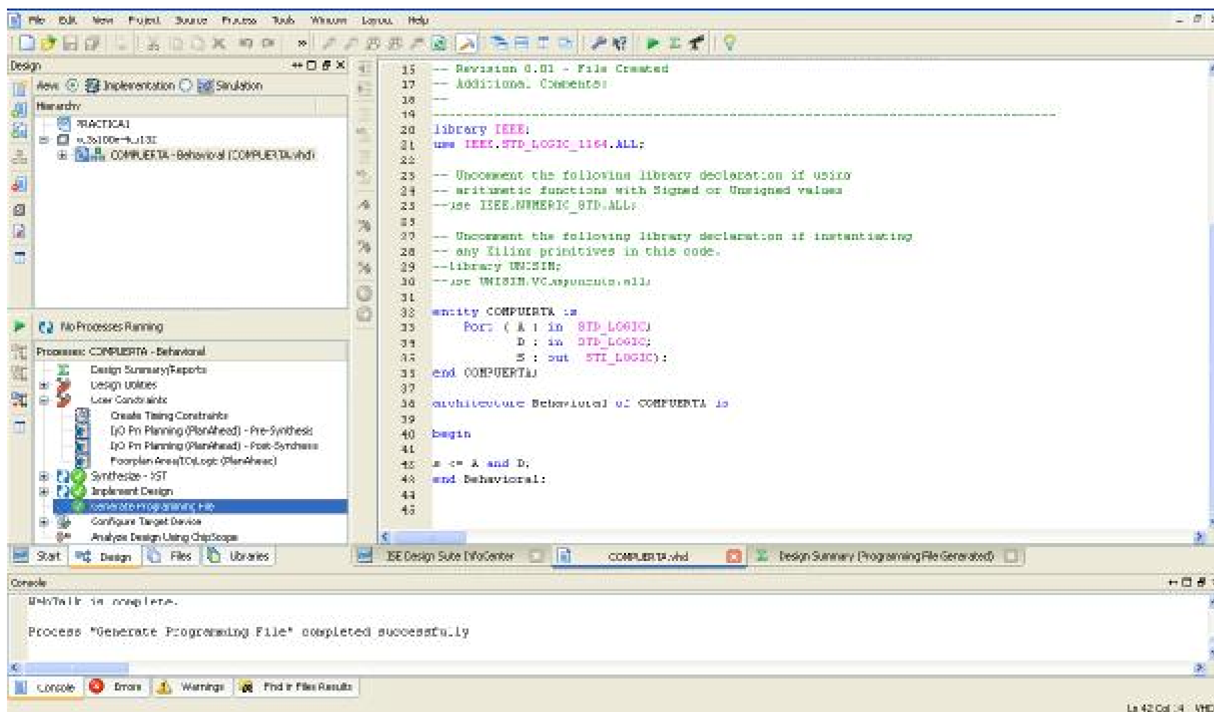


Figura 3.3.11 Generando programa con extensión .bit

Para descargar el programa compuerta.bit, nuestra tarjeta BASYS 2 deberá estar conectada por USB a nuestra PC y con swicth de power en la posición de ON (ver figura 3.3.12).

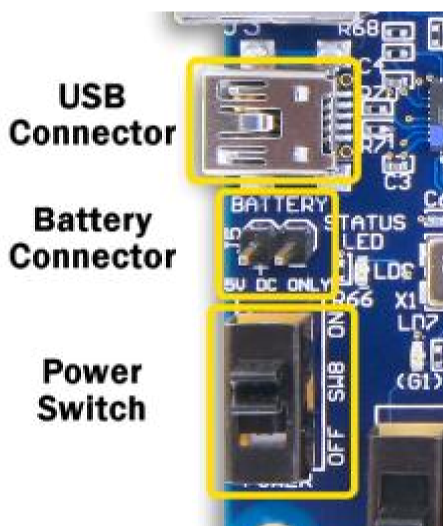


Figura 3.3.12 Conexión de la tarjeta BASYS 2

Ahora debemos abrir el programa “Adept” para descargar el programa compuerta.bit a la tarjeta. El programa “Adept” reconocerá el FPGA XC3S100E y la tarjeta BASYS 2 siempre (ver figura 3.3.13).

Figura 3.3.13 Adept, detección automática

Para descargar el programa compuerta.bit, debemos dar click en “Browse” y buscar el archivo en la carpeta del proyecto “PRÁCTICA1” (ver figura 3.3.14).

Figura 3.3.14 Selección del programa compuerta.bit

Ahora sólo falta dar click en “Program” para descargar nuestro programa a la tarjeta (ver figura 3.3.15).

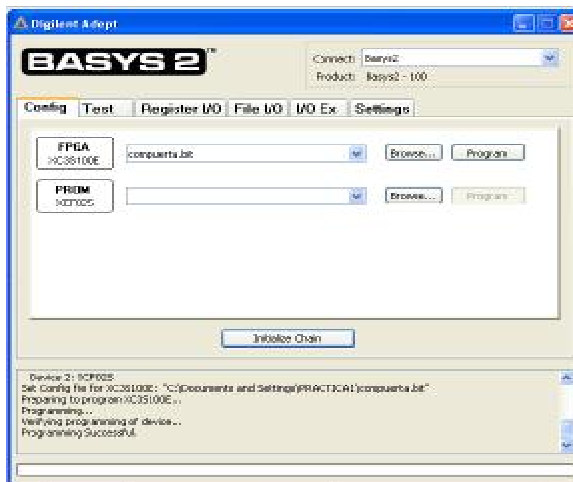


Figura 3.3.15 Descarga del programa

Por último realizaremos pruebas con la tarjeta, sólo cuando los SW0 y SW1 estén en alto, al mismo tiempo deberá encender el LD0, figuras 3.3.16, 3.3.17, 3.3.18 y 3.3.19.

Figura 3.3.16 SW0: bajo SW1: bajo LD0: bajo

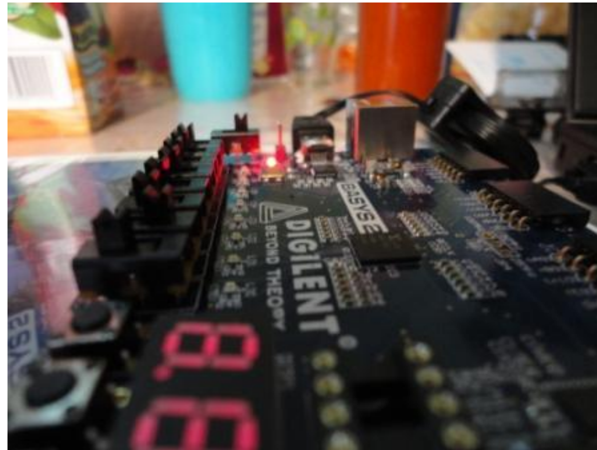


Figura 3.3.17 SW0: alto SW1: bajo LD0: bajo

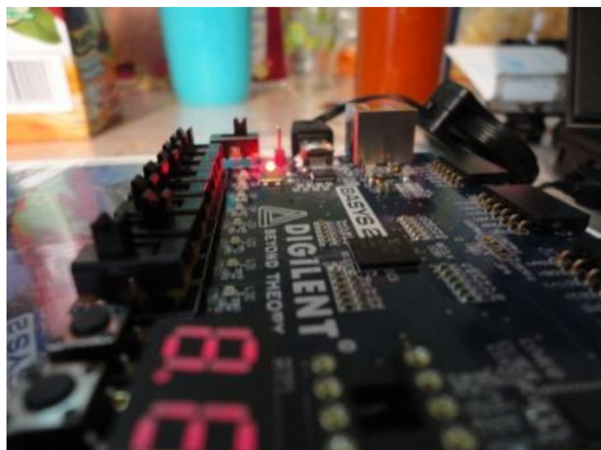


Figura 3.3.18 SW0: bajo SW1: alto LD0: bajo

Figura 3.3.19 SW0: alto SW1: alto LD0: alto

Conclusiones.

Se implementó el proyecto nombrado “PRÁCTICA1” en la tarjeta BASYS 2. Se observó el comportamiento de una compuerta AND teniendo como entradas dos switch y como salida un led.
La implementación se realizó de forma rápida y eficaz sin la necesidad de hacer conexiones entre dispositivos.
La tarjeta Basys 2 cuenta con varios dispositivos para probar nuestros diseños; led, push-botton, switch, display de siete segmentos, etc.

Práctica 4

Modelado, simulación e implementación de funciones booleanas

Objetivo

A partir de una función S de cuatro variables se generará una tabla de verdad de la cual se obtendrán funciones de salida que deberán de ser reducidas para programarlas en VHDL e implementarlas en la tarjeta de trabajo BASYS 2.

Desarrollo

Dada la función S obtendremos la tabla que se muestra en la figura 3.4.1

$$S = 35(W + Z) + 25Z(X + Y) + 10$$

W	X	Y	Z	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	1	0	1	0
0	0	0	1	0	0	1	0	1	1	0	1
0	0	1	0	0	0	0	0	1	0	1	0
0	0	1	1	0	1	0	0	0	1	1	0
0	1	0	0	0	0	0	0	1	0	1	0
0	1	0	1	0	1	0	0	0	1	1	0
0	1	1	0	0	0	0	0	1	0	1	0
0	1	1	1	0	1	0	1	1	1	1	1
1	0	0	0	0	0	1	0	1	1	0	1
1	0	0	1	0	1	0	1	0	0	0	0
1	0	1	0	0	0	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1	0	0	1
1	1	0	0	0	0	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1	0	0	1
1	1	1	0	0	0	1	0	1	1	0	1
1	1	1	1	1	0	0	0	0	0	1	0

Figura 3.4.1 Tabla de la función S

Como se puede apreciar la función de salida S_7 al colocar los minterminos en un mapa de karnaugh la función no se puede simplificar más, por lo que la función simplificada será la misma, tal como se muestra en la figura 3.4.2.

$$S_7 = WXYZ$$

WX \ YZ	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	0	0	0	0

$$S_7 = WXYZ$$

Figura 3.4.2 Simplificación de S_7

Para la función de salida S_6 al colocar lo mintérminos se observa que éstos pueden ser agrupados para simplificar la función, tal como se muestra en la figura 3.4.3.

$$S_6 = \overline{W}\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XYZ + W\overline{X}\overline{Y}Z + W\overline{X}YZ + WX\overline{Y}Z$$

WX \ YZ	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	0	1	0	0
10	0	1	1	0

$$S_6 = \overline{W}YZ + W\overline{X}Z + X\overline{Y}Z$$

Figura 3.4.3 Simplificación de S_6

De la misma manera lo hacemos para S_5 y al agrupar los mintérminos la función se simplifica como se muestra en la figura 3.4.4.

$$S_5 = \overline{W}\overline{X}\overline{Y}Z + W\overline{X}\overline{Y}\overline{Z} + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\overline{Z} + WX\overline{Y}Z + WXY\overline{Z}$$

WX \ YZ	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	1	1	0	1
10	1	0	1	1

$$S_5 = \overline{WXY}Z + WX\overline{Y} + W\overline{X}Y + W\overline{Z}$$

Figura 3.4.4 Simplificación de S_5

Para el caso de S_4 se puede apreciar que al colocar los mintérminos estos no se pueden agrupar, por lo que la función simplificada será la misma, como se muestra en la figura 3.4.5.

$S_4 = \bar{W}XYZ + W\bar{X}\bar{Y}Z$

WX \ YZ	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	0	0
10	0	1	0	0

$$S_4 = \overline{W}XYZ + W\overline{XY}Z$$

Figura 3.4.5 Simplificación S_4

Haciendo lo mismo para S_3 al colocar los mintérminos en el mapa se puede observar como la agrupación de los mintérminos reduce notablemente la función, tal como se aprecia en la figura 3.4.6.

$S_3 = \bar{W}\bar{X}\bar{Y}\bar{Z} + \bar{W}\bar{X}\bar{Y}Z + \bar{W}\bar{X}Y\bar{Z} + \bar{W}XY\bar{Z} + \bar{W}XYZ + W\bar{X}\bar{Y}\bar{Z} + \bar{W}XY\bar{Z} + W\bar{X}YZ + WX\bar{Y}\bar{Z} + WX\bar{Y}Z$

$+WXY\bar{Z}$

WX \ YZ	00	01	11	10
00	1	1	0	1
01	1	0	1	1
11	1	1	0	1
10	1	0	1	1

$$S_3 = \overline{WXY} + \overline{W}XY + W\overline{X}Y + WX\overline{Y} + \overline{Z}$$

Figura 3.4.6 Simplificación de S_3

Para S_2 se puede apreciar que la agrupación de mintérminos es más amplia, lo que simplifica significativamente la función, como se muestra en la figura 3.4.7.

$S_2 = \bar{W}\bar{X}\bar{Y}Z + \bar{W}\bar{X}YZ + \bar{W}X\bar{Y}Z + \bar{W}XYZ + W\bar{X}\bar{Y}\bar{Z} + W\bar{X}Y\bar{Z} + WX\bar{Y}\bar{Z} + WXY\bar{Z}$

WX \ YZ	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	0	0	1
10	1	0	0	1

$\boldsymbol{S_2 = \bar{W}Z + W\bar{Z}}$

Figura 3.4.7 Simplificación de S_2

Finalmente podemos observar la simplificación correspondiente a S_1 y S_0, en las figuras 3.4.8 y 3.4.9 respectivamente.

$S_1 = \bar{W}\bar{X}\bar{Y}\bar{Z} + \bar{W}\bar{X}Y\bar{Z} + \bar{W}\bar{X}YZ + \bar{W}X\bar{Y}\bar{Z} + \bar{W}X\bar{Y}Z + \bar{W}XY\bar{Z} + \bar{W}XYZ + WXYZ$

WX \ YZ	00	01	11	10
00	1	0	1	1
01	1	1	1	1
11	0	0	1	0
10	0	0	0	0

$\boldsymbol{S_1 = \bar{W}X + \bar{W}Y + \bar{W}\bar{Z} + XYZ}$

Figura 3.4.8 Simplificación de S_1

$S_0 = \overline{W}\overline{X}\overline{Y}Z + \overline{W}XYZ + W\overline{X}\overline{Y}\overline{Z} + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\overline{Z} + WX\overline{Y}Z + WXY\overline{Z}$

WX \ YZ	00	01	11	10
00	0	1	0	0
01	0	0	1	0
11	1	1	0	1
10	1	0	1	1

$$\boldsymbol{S_0 = \overline{WXY}Z + \overline{W}XYZ + W\overline{X}Y + WX\overline{Y} + W\overline{Z}}$$

Figura 3.4.9 Simplificación de S_0

Ahora que ya se han simplificado las funciones de salida se tendrá que determinar el código en VHDL de cada función, haciendo uso de los operadores necesarios y así obtener el código.

Operadores:

- and, or, nand, nor, xor
- <= asignación de valores a señales

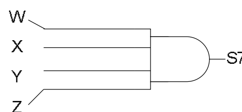
Ejemplo:

$s = wx + \bar{z}$

s <= (w and x) or not(z);

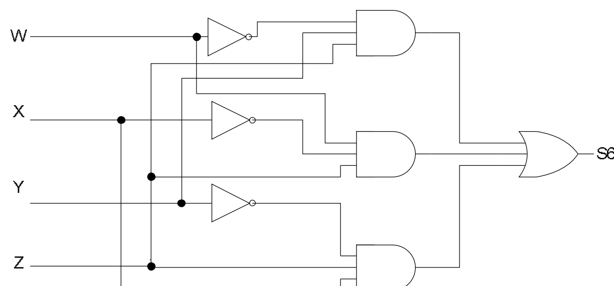
Entonces las funciones quedaran representadas de la siguiente manera:

$\boldsymbol{S_7 = WXYZ}$



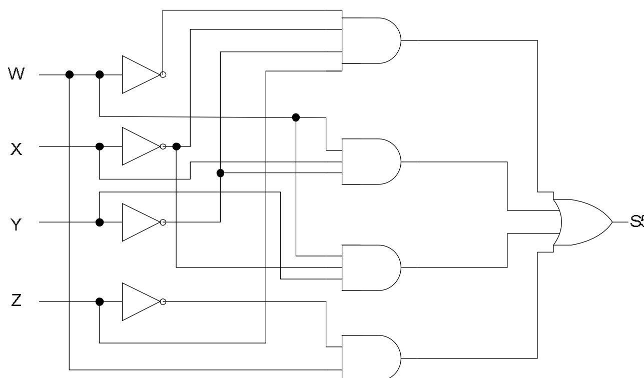
S7 <= W AND X AND Y AND Z;

$S_6 = \overline{W}YZ + W\overline{X}Z + X\overline{Y}Z$



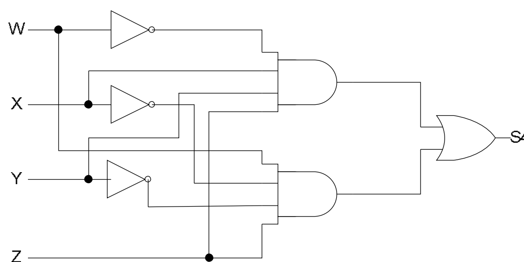
S6 <= (NOT(W) AND Y AND Z) OR (W AND NOT(X) AND Z) OR (X AND NOT(Y) AND Z);

$S_5 = \overline{W}\overline{X}\overline{Y}Z + WX\overline{Y} + W\overline{X}Y + W\overline{Z}$



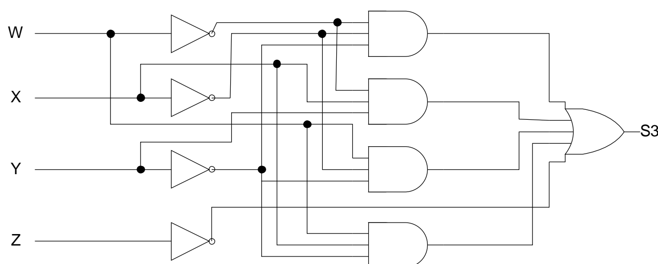
S5 <= (NOT(W) AND NOT(X) AND NOT(Y) AND Z) OR (W AND X AND NOT(Y))OR (W AND NOT(X) AND Y) OR (W AND NOT(Z));

$S_4 = \overline{W}XYZ + W\overline{X}\overline{Y}Z$



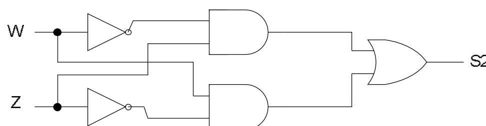
S4 <= (NOT(W) AND X AND Y AND Z) OR (W AND NOT(X) AND NOT(Y) AND Z);

$$S_3 = \overline{WXY} + \overline{W}XY + W\overline{X}Y + WX\overline{Y} + \overline{Z}$$



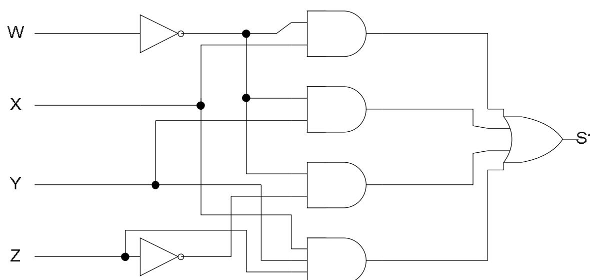
S3 <= (NOT(W) AND NOT(X) AND NOT(Y)) OR (NOT(W) AND X AND Y) OR (W AND NOT(X) AND Y) OR (W AND X AND NOT(Y)) OR NOT(Z);

$$S_2 = \overline{W}Z + W\overline{Z}$$



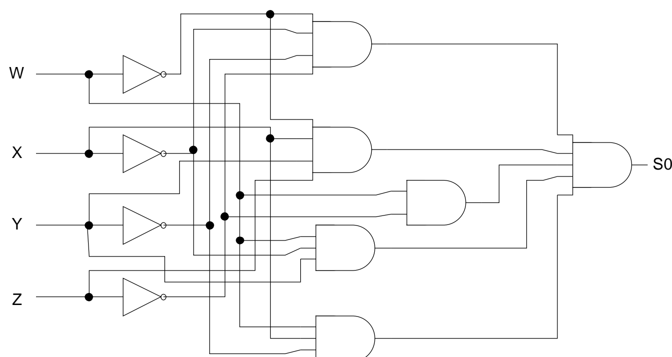
S2 <= (NOT(W) AND Z) OR (W AND NOT(Z));

$$S_1 = \overline{W}X + \overline{W}Y + \overline{WZ} + XYZ$$



S1 <= (NOT(W) AND X) OR (NOT(W) AND Y) OR (NOT(W) AND NOT(Z)) OR (X AND Y AND Z);

$$S_0 = \overline{WXY}Z + \overline{W}XYZ + W\overline{X}Y + WX\overline{Y} + W\overline{Z}$$



S0 <= (NOT(W) AND NOT(X) AND NOT(Y) AND Z) OR (NOT(W) AND X AND Y AND Z) OR (W AND NOT(X) AND Y) OR (W AND X AND NOT(Y)) OR (W AND NOT(Z));

Ahora ya podemos crear nuestro proyecto (ver práctica 1), el cual se nombrara como Practica4 en donde declararemos los puertos de entrada y salida como se muestra en la figura 3.4.10.

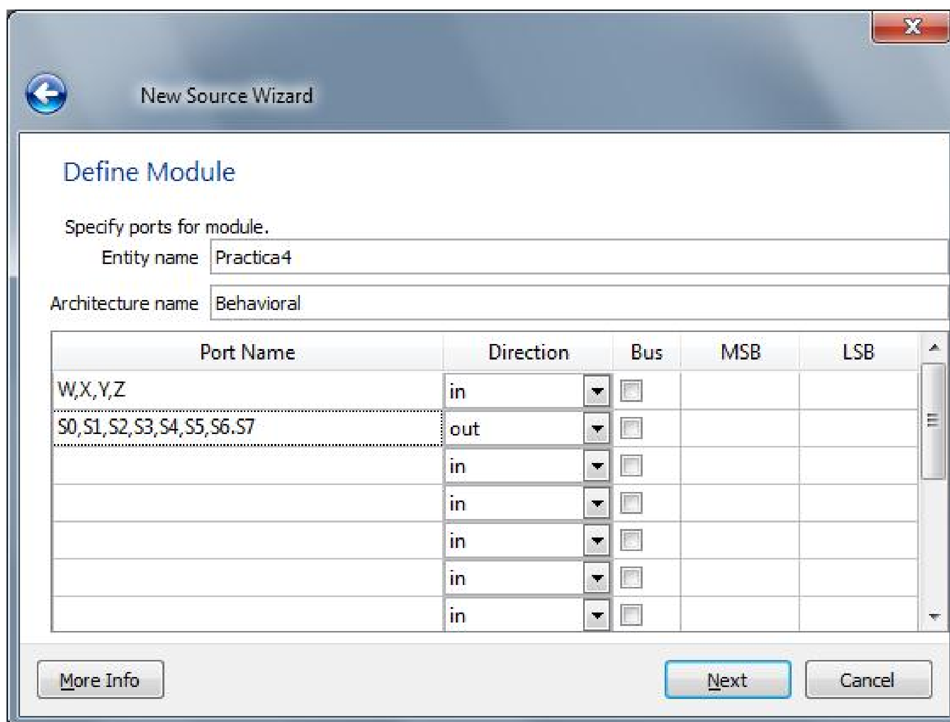


Figura 3.4.10 Declaración de puertos

Ya creado el proyecto introduciremos el código VHDL que generamos para nuestras funciones de salida, como se muestra en la figura 3.4.11, y compilamos nuestro proyecto para verificar que no hay errores (ver práctica 1).

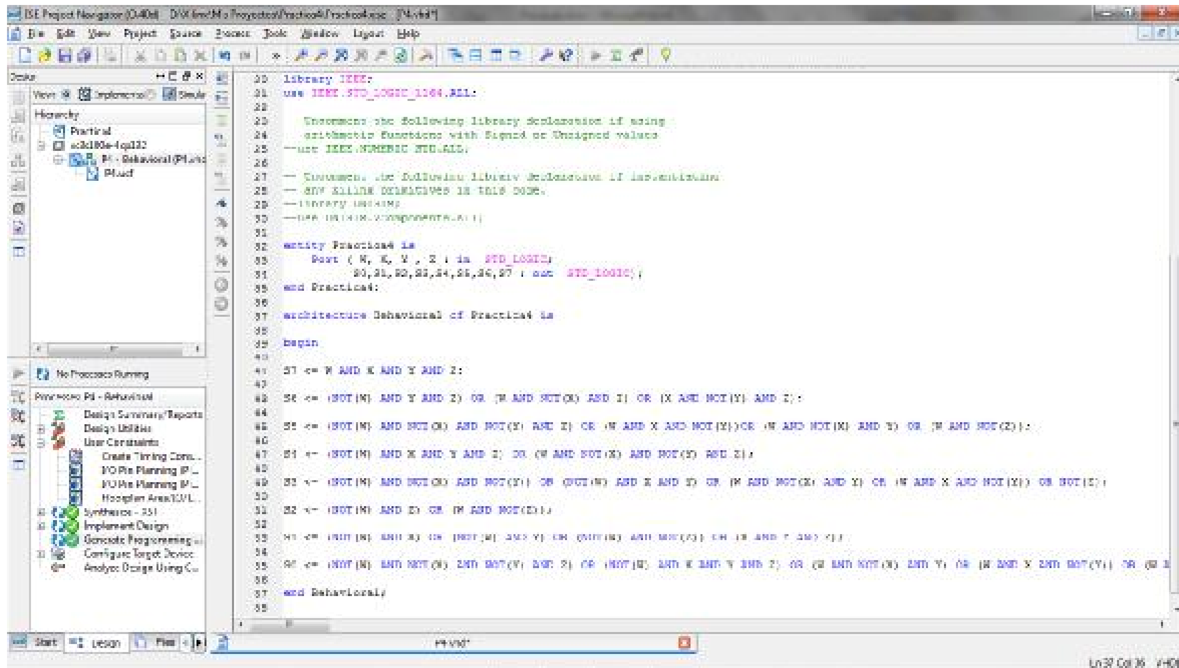


Figura 3.4.11 Compilación correcta del diseño

Ahora se simulará el proyecto (ver práctica 2) para revisar que cumple con la tabla de la figura 3.4.1 que generamos al inicio de la práctica, necesitaremos configurar cuatro señales de reloj de entrada (W,X,Y y Z) de las mismas características y con un retraso de 5ns entre ellas después de la primera señal como se muestra en la figura 3.4.12; esto es para tratar de comprobar el mayor número de combinaciones posibles como se muestra en la figura 3.4.13

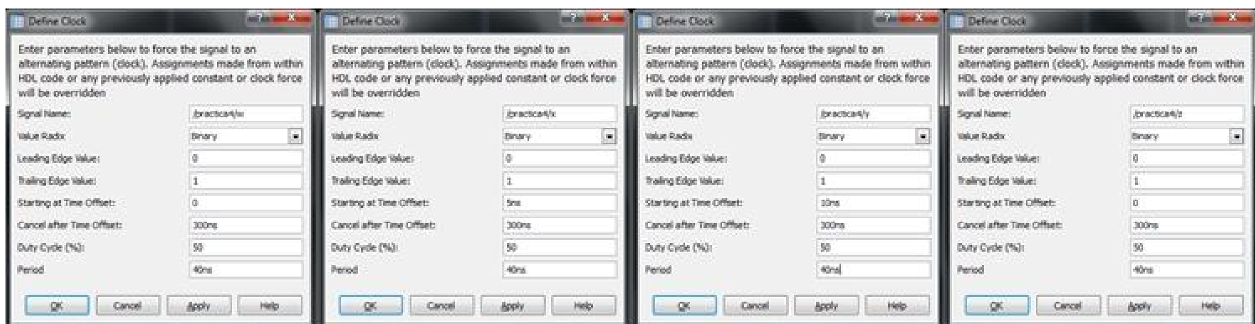


Figura 3.4.12 Configuración de señales de entrada

En la Figura 3.4.13 se puede observar como para el valor 1 1 1 0 se obtiene la salida que muestra la tabla 3.4.1 para este valor que es 0 0 1 0 1 1 0 1, de la misma forma se puede verificar para cualquier otro valor con la línea de posicionamiento.

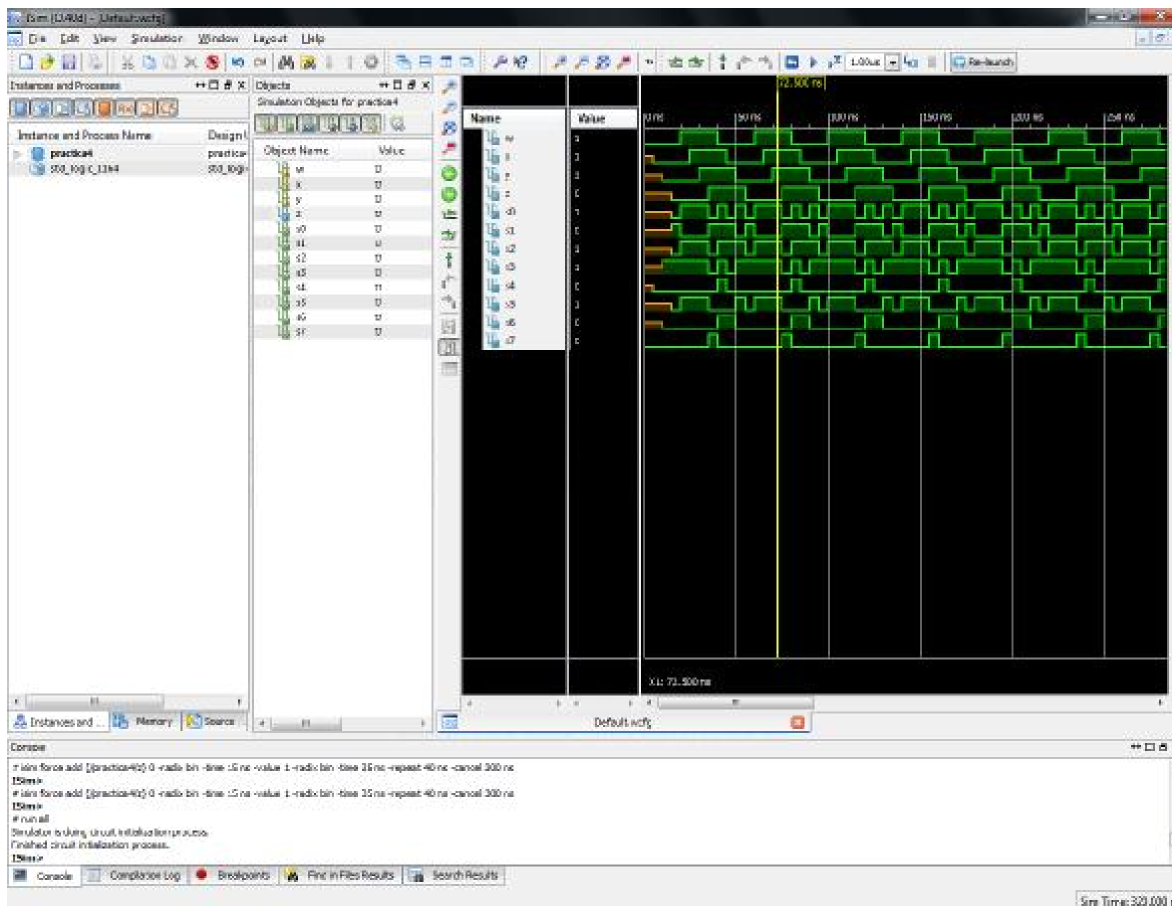


Figura 3.4.13 Simulación correcta

La parte final de la práctica será generar el archivo .bit para la tarjeta BASYS2 a través del programa PlanAhead (ver práctica 3) y la asignación de pines quedará como se muestra en la figura 3.4.14.

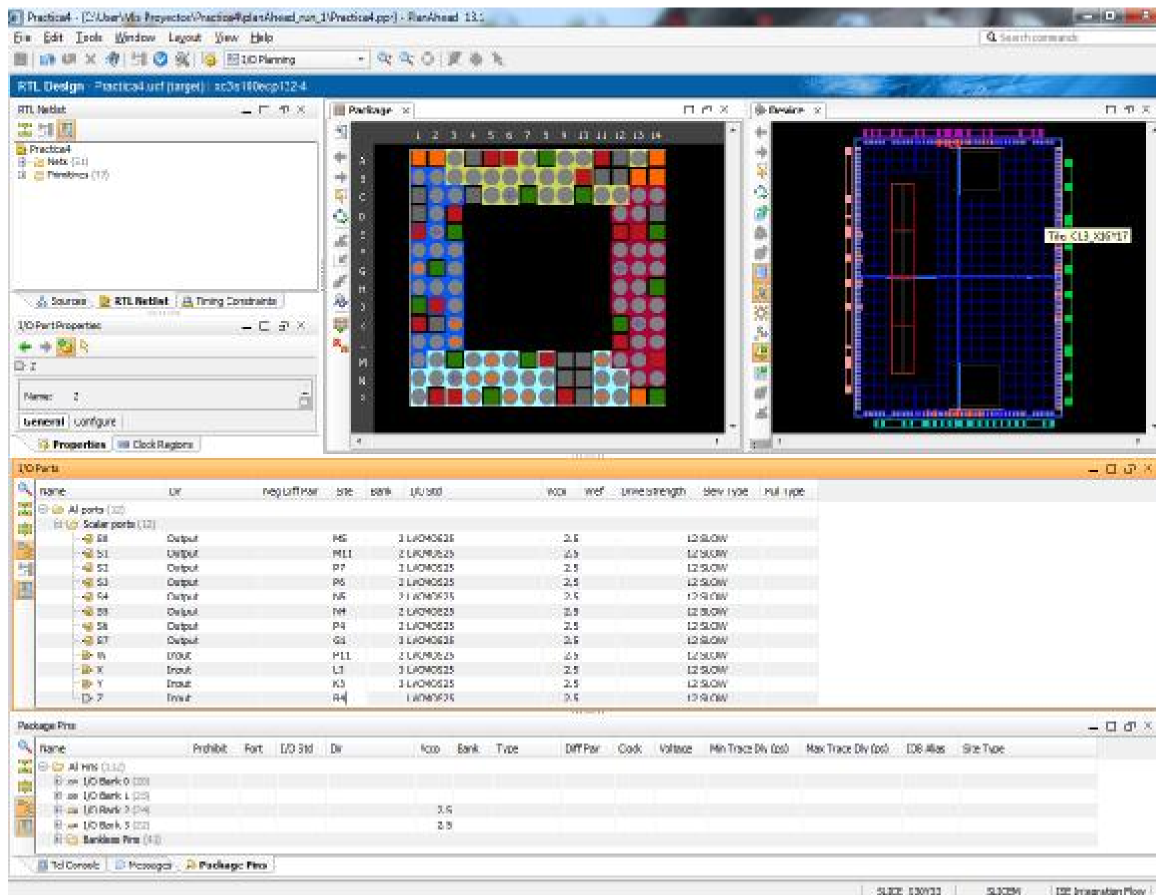


Figura 3.4.14 Asignación de pines

Una vez que se ha guardado la asignación de pines, verificamos que no haya errores de diseño con el "Implement Top Module" como se muestra en la figura 3.4.15, y generamos nuestro archivo .bit para descargarlo a la tarjeta BASYS2 (ver práctica 3).

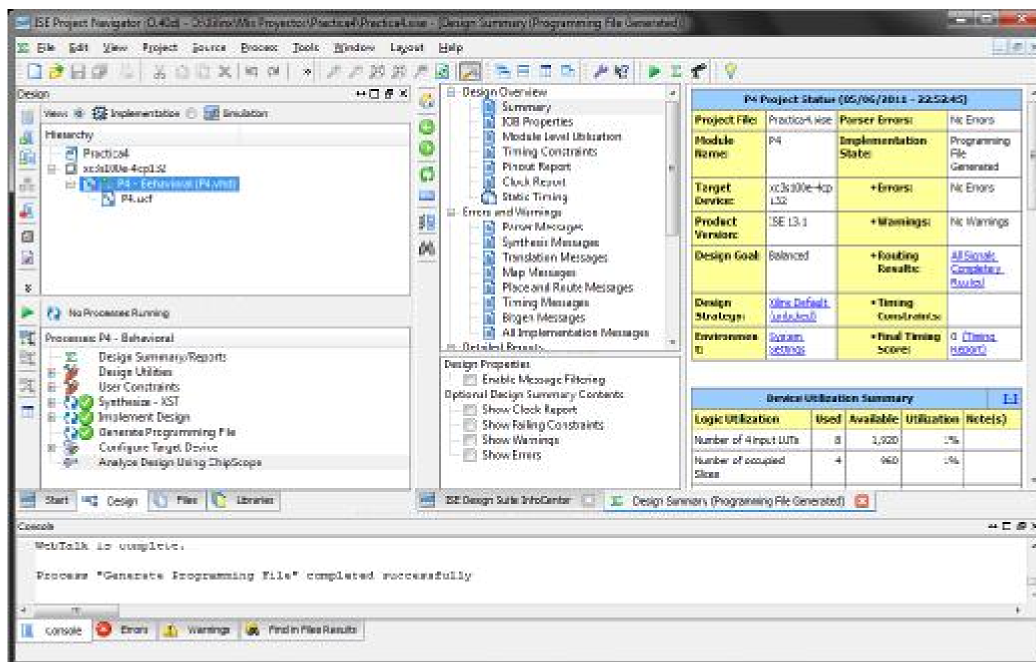


Figura 3.4.15 Generación exitosa del archivo .bit

Finalmente se puede observar en las siguientes imágenes el funcionamiento del proyecto creado en Xilinx ISE para la tarjeta BASYS 2, para los valores: 0000 (figura 3.4.16), 0100 (figura 3.4.17) y 1111 (figura 3.4.18).

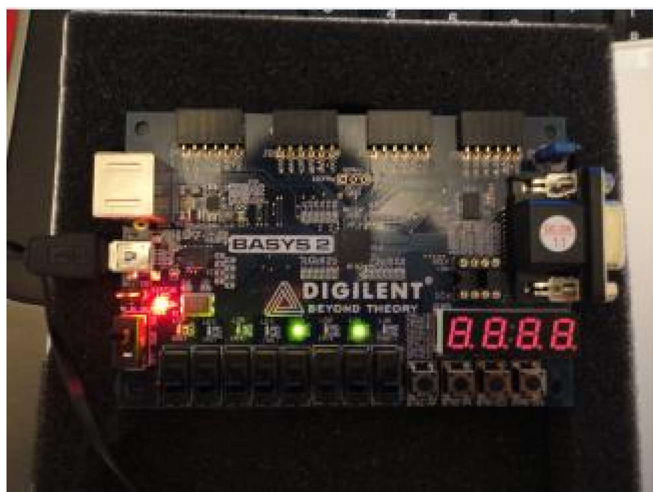


Figura 3.4.16 Valor de la función S en 0000

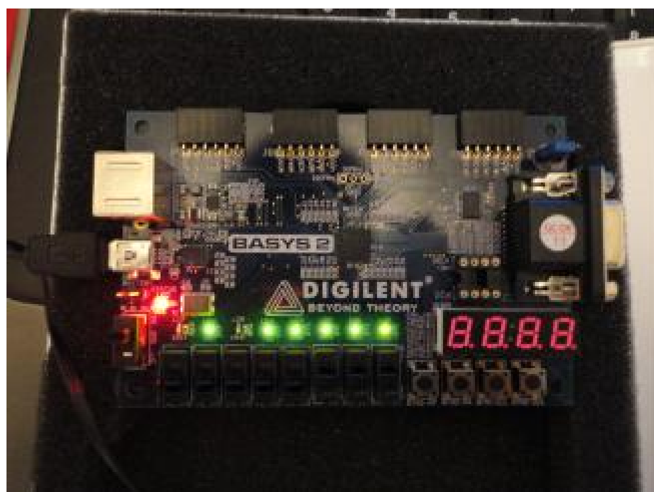


Figura 3.4.17 Valor de la función S en 0100

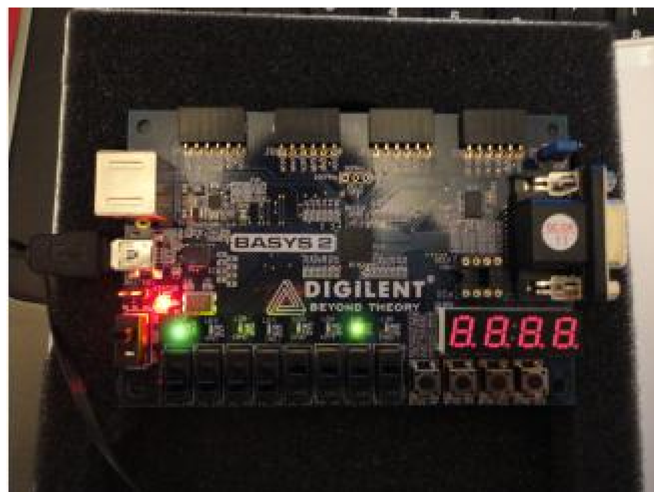


Figura 3.4.18 Valor de la función S en 1111

Conclusión

A través de una función booleana S se implementaron y modelaron sus funciones de salida por medio del ambiente de desarrollo Xilinx ISE, se generó el programa .bit para la tarjeta BASYS 2, del cual se pudo comprobar su funcionamiento correcto como se mostraba en las simulaciones realizadas durante la práctica.

Práctica 5

Modelado e implementación de Codificadores y Decodificadores

Objetivo

Modelar e implementar un codificador 8 a 3 y decodificador BCD a 7 segmentos ánodo común utilizando la tarjeta de trabajo BASYS 2 y el software de Xilinx.

Desarrollo

Parte 1: Codificador 8 a 3.

Para el desarrollo del codificador es necesario establecer una prioridad para los casos en los que sean varias las entradas activas en un momento determinado, ver figura 3.5.1

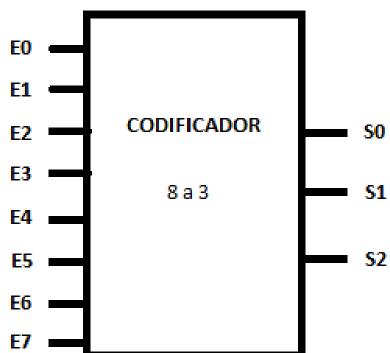


Figura 3.5.1 Codificador 8 a 3

En la tabla de la figura 3.5.2 podemos observar el comportamiento de nuestro codificador para las entradas y los valores que tomaran a la salida, cabe mencionar que se declarará una salida tipo bit a la que nombraremos “error” para el caso de una entrada que sea incorrecta

E0	E1	E2	E3	E4	E5	E6	E7	S0	S1	S2	error
1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	1	1	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	1	0	1	0
0	0	0	0	0	0	1	0	1	1	0	0
0	0	0	0	0	0	0	1	1	1	1	0
X	X	X	X	X	X	X	X	X	X	X	1

Figura 3.5.2 Tabla del codificador 8 a 3

Generamos el proyecto en ISE Project Navigator el cual lleva por nombre "Practica5_codificador", agregamos una nueva fuente de tipo VHDL Module, la nombramos codificador8a3 tal y como se muestra en la Figura 3.5 .3. y corroboramos que este habilitada la opcion Add to project.

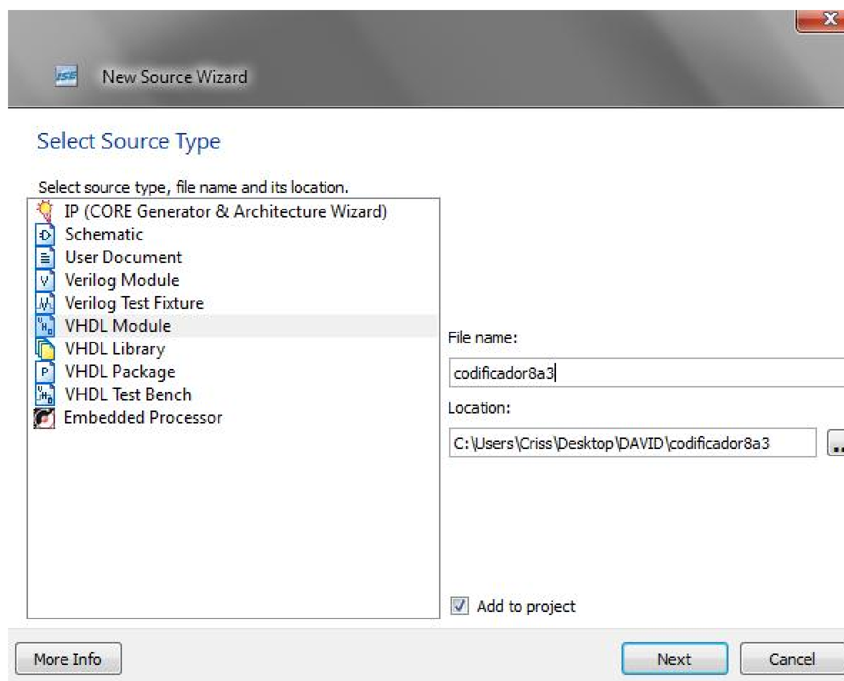


Figura 3.5.3 Fuente VHDL

A las entradas E0,E1,E2,E3,E4,E5,E6,E7 las describiremos en nuestro proyecto con el nombre "iEntrada", las salidas S0,S1,S2 las describiremos con el nombre "Salida" (ver figuira 3.5.4).

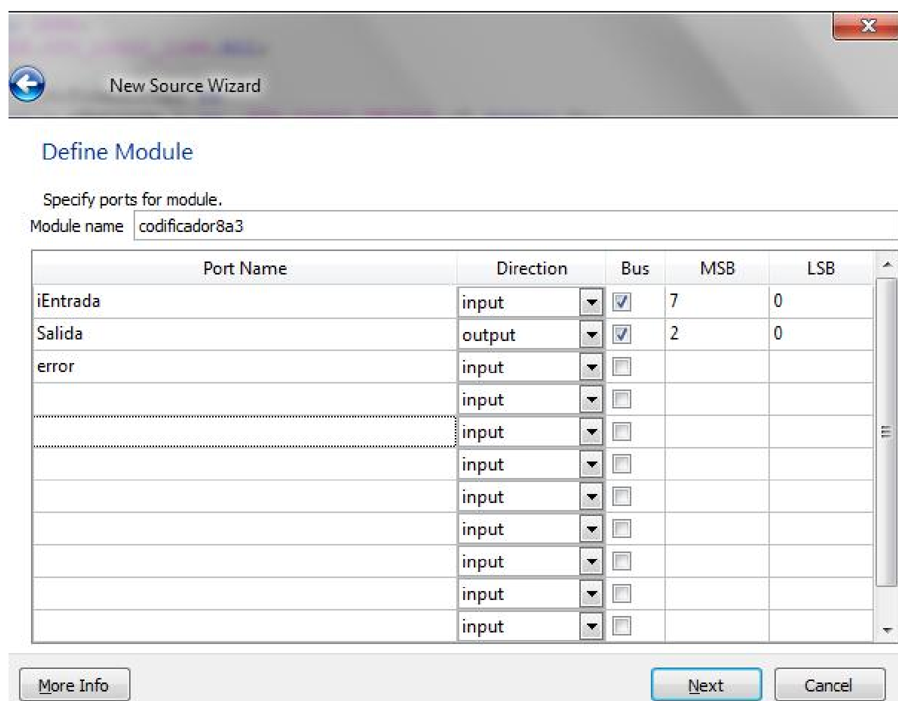


Figura 3.5.4 Declaracion de puertos Entrada/Salida

A continuacion se describe el codigo en VHDL del codificador 8 a 3:

```
--Inicia Codigo

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity codificador8a3 is

   Port ( iEntrada : in  STD_LOGIC_VECTOR (7 downto 0);

              error : out STD_LOGIC;

              Salida : out  STD_LOGIC_VECTOR (2 downto 0));

end codificador8a3;

architecture Behavioral of codificador8a3 is

begin

Codif: PROCESS (iEntrada)

begin

   error <= '0'; ! Se limpia el bit de error
```

```
  case iEntrada is
        when "00000001" => Salida <= "000";
        when "00000010" => Salida <= "001";
        when "00000100" => Salida <= "010";
        when "00001000" => Salida <= "011";
         when "00010000" => Salida <= "100";
         when "00100000" => Salida <= "101";
         when "01000000" => Salida <= "110";
         when "10000000" => Salida <= "111";
   when others => error  <= '1';
  end case;
end process;
end Behavioral;
--Finaliza Codigo
```

En la Figura 3.5.5 se muestra el codigo anteriormente descrito compilado sin errores con la herramienta Xilinx.

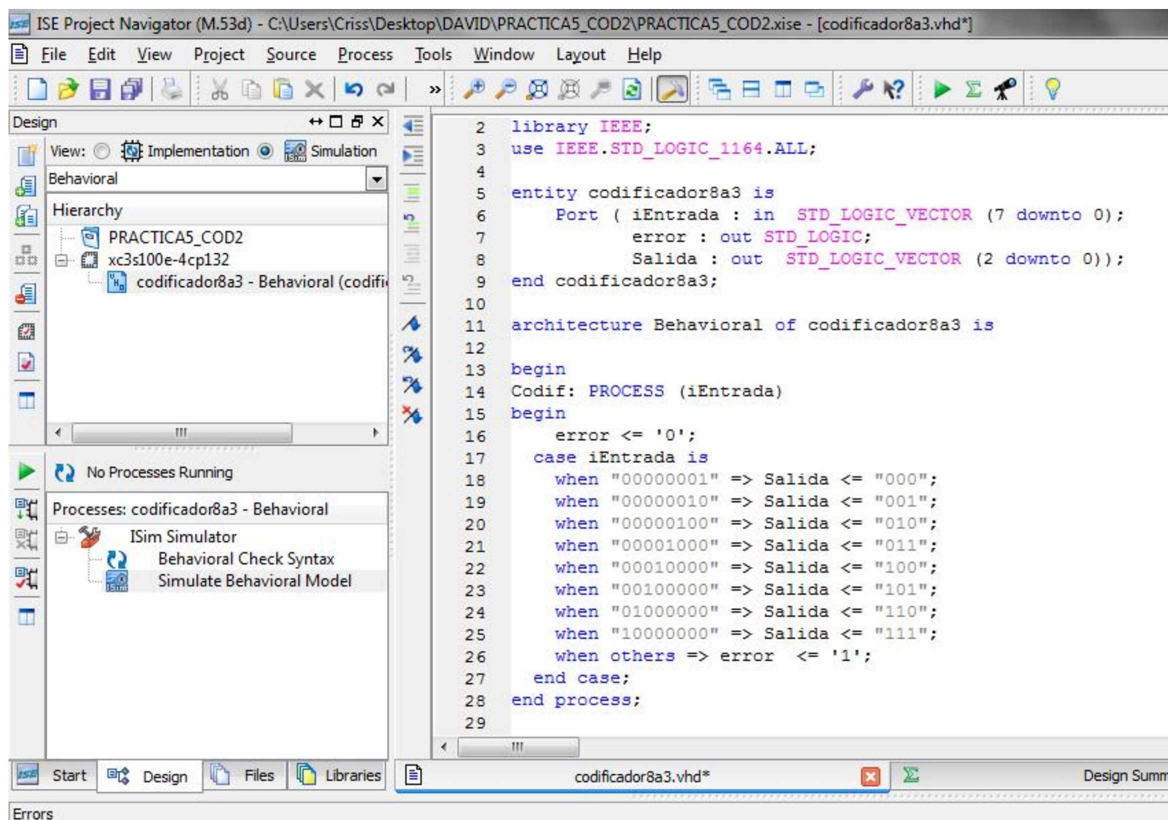


Figura 3.5.5 Proyecto Codidicador 8 a 3

Para verificar el comportamiento de nuestra descripción utlizaremos la herramienta ISim de Xilinx para simular nuestro diseño en VHDL.

Para asignar valores constantes a nuestro puerto iEntrada damos click derecho sobre la variable del mismo nombre y posteriormente en el menú que se despliega damos aceptar en la opción “Forcé Constant” (ver figura 3.5.6).

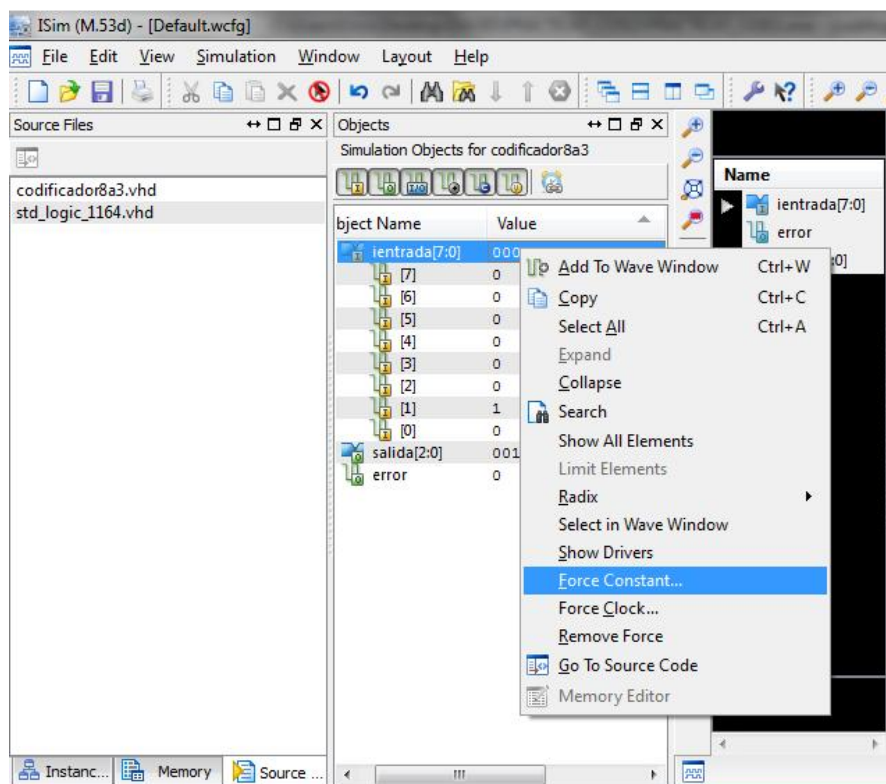


Figura 3.5.6 Selector de Constante

Para colocar los valores a la entrada podemos ponerlos en hexadecimal, binario, decimal, etc. Nosotros escogeremos binario y asignaremos el numero "00000010" en force to value, posteriormente damos aceptar en aplicar y aceptamos los cambios como se muestra en la figura 3.5.7 . Para correr la simulacion seleccionamos "F5" ó damos click en el icono Run all.

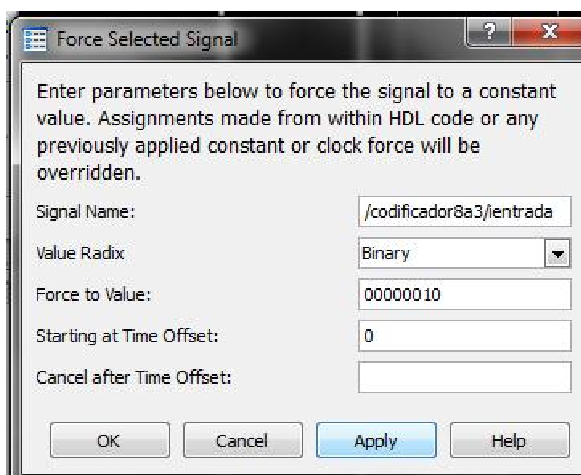


Figura 3.5.7 Selector de valores

Podemos observar en la figura 3.5.8 que para el valor en binario de entrada "00000010" tenemos como salida el valor "001" el cual es correcto de acuerdo a nuestra tabla descrita en la figura 3.5.2.

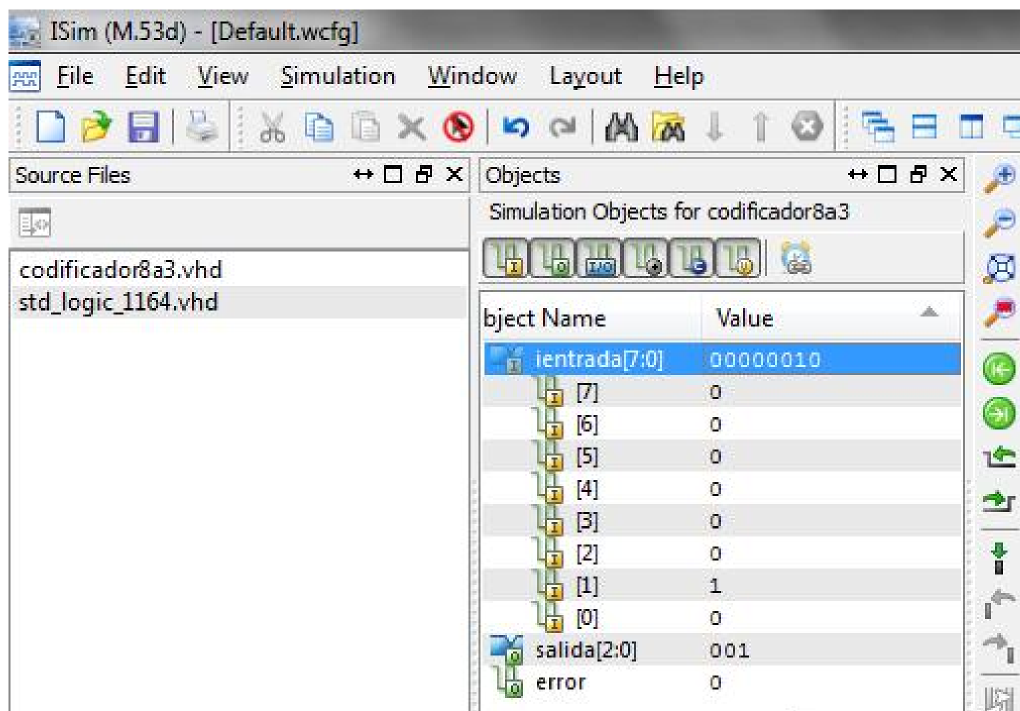


Figura 3.5.8 Simulacion

Para bajar nuestro proyecto a la tarjeta de pruebas BASYS 2, utilizaremos la herramienta PlanAHead Pre-Synthesis de Xilinc en donde asignaremos los puertos físicos de la tarjeta a las entradas y salidas declaradas en nuestra descripción.

En la Figura 3.5.9 se observa la asignación de los pines que hemos adoptado. Para las entradas utilizamos del SW0 al SW7, para la salida del LD0 al LD2 y para el bit de error asignamos el LD4.

I/O Ports

Name	Dir	Neg Diff Pair	Site	Bank
All ports (12)				
Salida (3)	Output			
Salida[0]	Output		M5	
Salida[1]	Output		M11	
Salida[2]	Output		P7	
iEntrada (8)	Input			
iEntrada[0]	Input		P11	
iEntrada[1]	Input		L3	
iEntrada[2]	Input		K3	
iEntrada[3]	Input		B4	
iEntrada[4]	Input		G3	
iEntrada[5]	Input		F3	
iEntrada[6]	Input		E2	
iEntrada[7]	Input		N3	
Scalar ports (1)				
error	Output		N5	

Figura 3.5.9 Asignación de pines

Para poder conectar la tarjeta de pruebas BASYS 2 a nuestra PC regresamos a la pantalla principal de nuestro proyecto, damos click derecho en “Generate Programming File” y en la opción “Process Properties” verificamos que se encuentre JTAG Clock como valor seleccionado en las propiedades del FPGA Start-Up Clock.

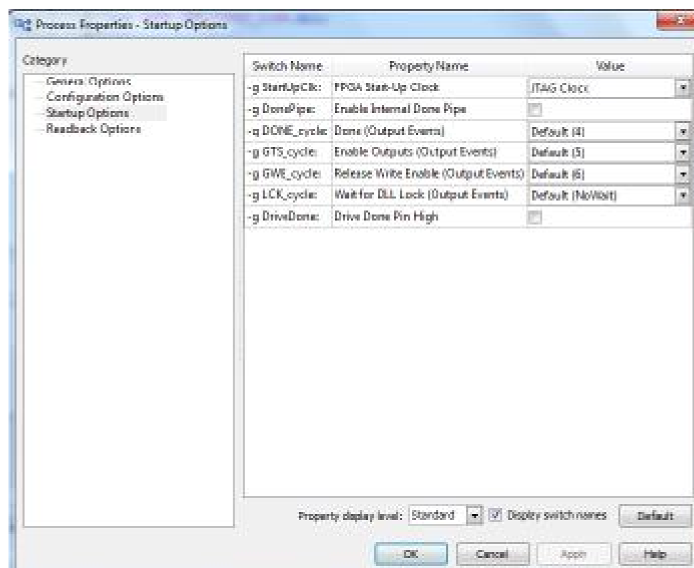


Figura 3.5.10 Configuración de reloj del FPGA

Con esto ya podemos generar nuestro archivo .bit y bajarlo a nuestra tarjeta de prueba. En la figura 3.5.11 se muestra una imagen con una entrada activada por los switches de la tarjeta.

Caso a:

iEntrada: SW0=0 , SW1=0, SW2=0, SW3=0, SW4=0, SW5=0, SW6=0, SW7=0.
Revisando la salida se observa que correctamente se enciende el led configurado para error
Salida: LD0=X, LD1=X, LD2=X.
Error: LD4=1

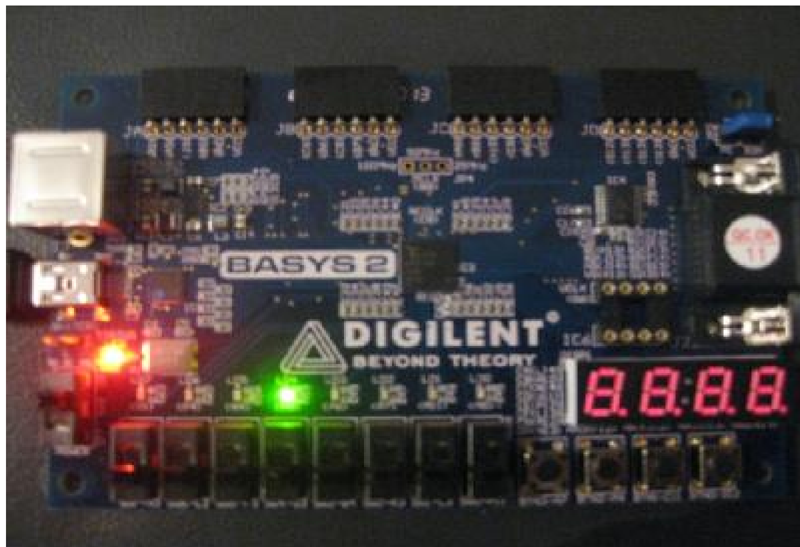


Figura 3.5.11 Caso a

En la Figura 3.5.12 se muestra ahora el comportamiento de nuestro codificador 8 a 3 con los valores de iEntrada del caso b.

Caso b:
iEntrada: SW0=0 , SW1=0, SW2=0, SW3=0, SW4=0, SW5=0, SW6=0, SW7=1.
Salida: LD0=1, LD1=1, LD2=1.
Error: LD4=0

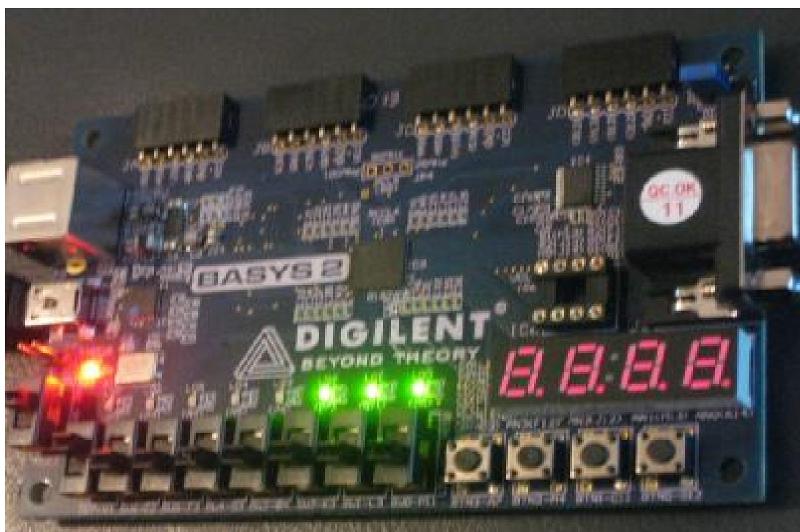


Figura 3.5.12 Caso b

Parte 2: Decodificador BCD a 7 segmentos.

Se desarrollara una descrpción que decodifique un número BCD a 7 segmentos (ver figura 3.5.13).

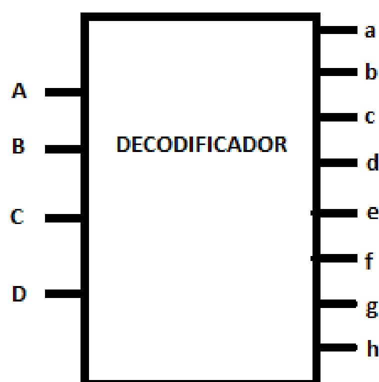


Figura 3.5.13 Decodificador BCD a 7 segmentos

En la tabla de la figura 3.5.14 se observa el comportamiento del decodificador que se implementara en la tarjeta BASYS 2, cabe mencionar que los displays de 7 segmentos de ánodo común encienden con “0” lógico.

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

Figura 3.5.14. Tabla Decodificador BCD a 7 segmentos

Generamos el proyecto de nombre PRACTICA5_DECODIFICADOR al cual asociaremos el archivo VHDL Module de nombre “decodificadorBCD”, la entrada BCD será del tipo std_logic_vector de 3 a 0 y como puerto de salida la variable “Segmento” del tipo std_logic_vector de 6 a 0 (ver figura 3.5.15).

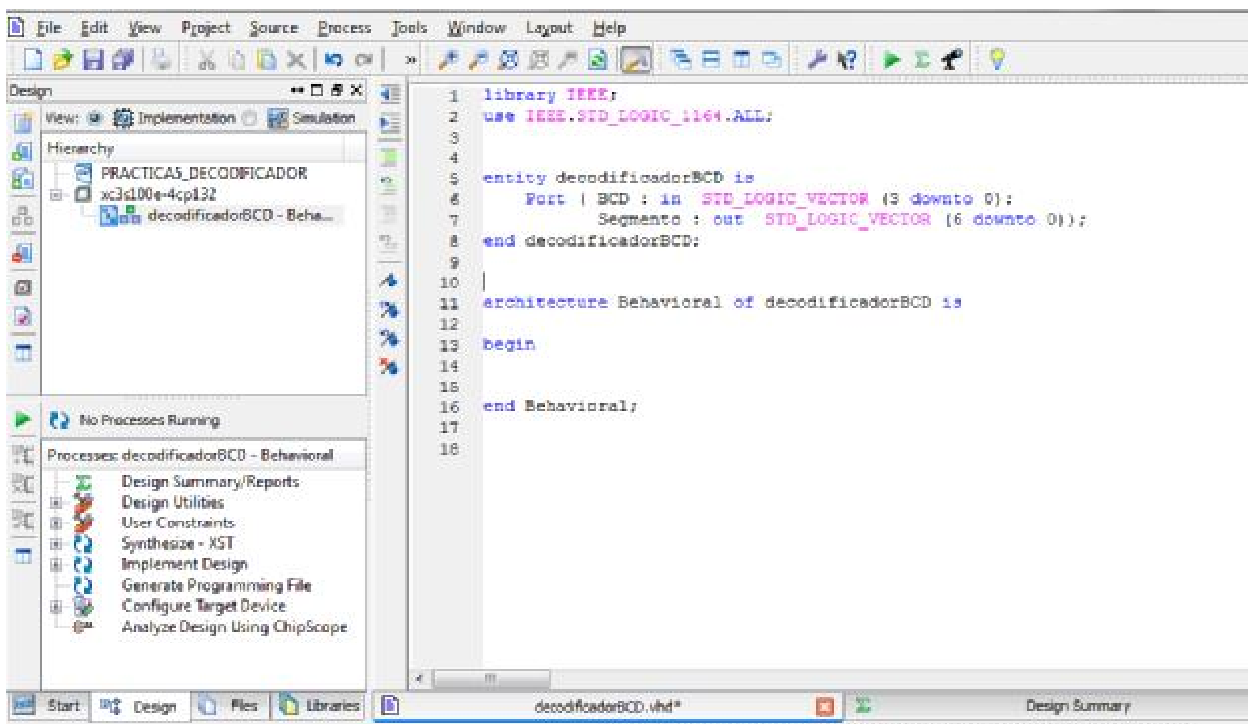


Figura 3.5.15 Creación del Proyecto

A continuación se describe el código del decodificador de BCD a 7 segmentos ánodo común.

---Inicia Código

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity decodificadorBCD is
    Port (     BCD : in  STD_LOGIC_VECTOR (3 downto 0);
               ENABLE : out  STD_LOGIC_VECTOR (3 downto 0);
               Segmento : out  STD_LOGIC_VECTOR (6 downto 0));
end decodificadorBCD;
architecture Behavioral of decodificadorBCD is
begin
ENABLE <= "1110" ; --habilitamos el display AN3(DISP1)
Decod: PROCESS (BCD)
```

```
BEGIN
        CASE BCD IS
-- Orden de las salidas (segmentos) “abcdefg”
                WHEN "0000" => Segmento <= "0000001";
                WHEN "0001" => Segmento <= "1001111";
                WHEN "0010" => Segmento <= "0010010";
                WHEN "0011" => Segmento <= "0000110";
                WHEN "0100" => Segmento <= "1001100";
                WHEN "0101" => Segmento <= "0100100";
                WHEN "0110" => Segmento <= "0100000";
                WHEN "0111" => Segmento <= "0001110";
                WHEN "1000" => Segmento <= "0000000";
                WHEN "1001" => Segmento <= "0000100";
                WHEN OTHERS => Segmento <= "1111111";
        END CASE;
END PROCESS Decod;
end Behavioral;
---Termina Código
```

Compilamos la descripción en VHDL y verificamos que la descripción no contenga errores, si fuera el caso depurar hasta que en la consola aparezca Process "Check Syntax" completed successfully (ver figura 3.5.16)

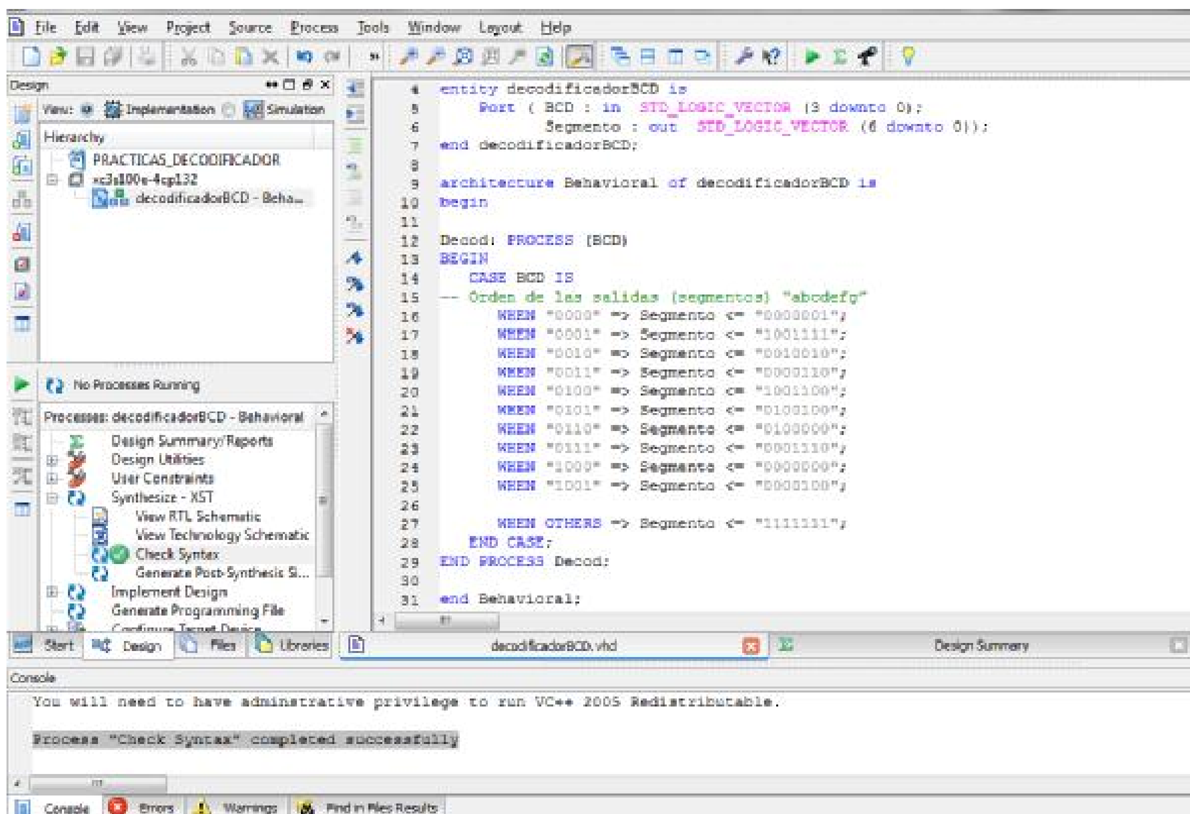


Figura 3.5.16 Proyecto PRACTICA5_DECODIFICADOR compilado

Para verificar que nuestra descripción funcione correctamente utilizaremos la herramienta ISim de Xilinx, en la figura 3.5.17 se observa que para el valor de entrada BCD “1001” tenemos en la salida Segmento el valor “0000100” lo cual es correcto.

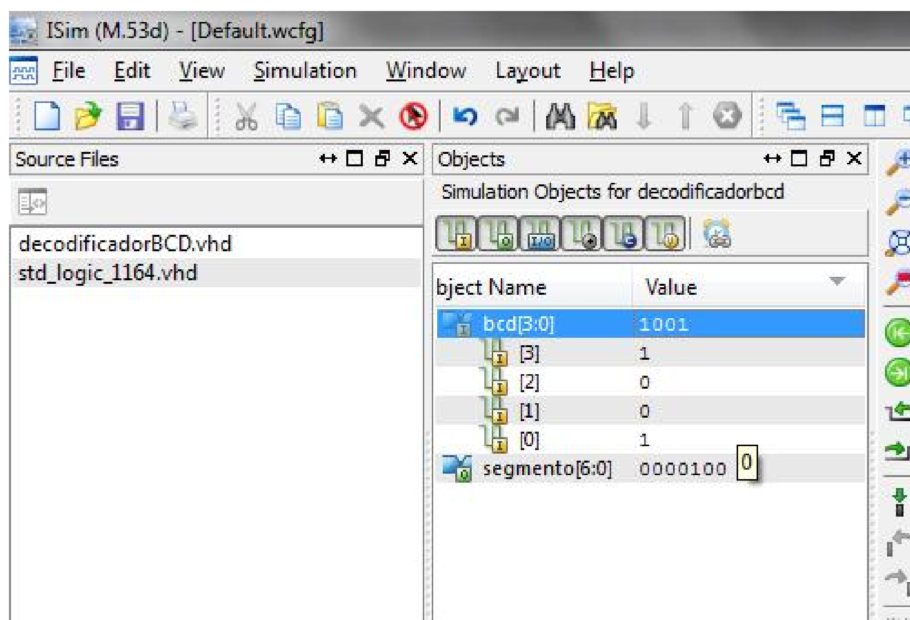


Figura 3.5.17 Herramienta ISim

Con la herramienta PlanAHead Pre-Synthesis asignamos los puertos físicos de la tarjeta BASYS 2 a las entradas y salidas, una vez realizada dicha asignación debemos guardar los cambios, para este caso se observa en la figura 3.5.18 la distribución de los pines que hemos adoptado, para la entrada BCD hemos utilizados los switches del SW3 al SW0, para la salida ENABLE hemos habilitado únicamente el Display 1 AN3, por ultimo para los la asignación de los segmentos el bit menos significativo pertenece al led g y el bit más significativo al del led a.

I/O Ports

Name	Dir	Neg Diff Pair	Site	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
All ports (15)									
BCD (4)	Input				LVCMOS25	2.5		12	SLOW
BCD[0]	Input		P11	2	LVCMOS25	2.5		12	SLOW
BCD[1]	Input		L3	3	LVCMOS25	2.5		12	SLOW
BCD[2]	Input		K3	3	LVCMOS25	2.5		12	SLOW
BCD[3]	Input		B4	0	LVCMOS25	2.5		12	SLOW
ENABLE (4)	Output			1	LVCMOS25	2.5		12	SLOW
ENABLE[0]	Output		K14	1	LVCMOS25	2.5		12	SLOW
ENABLE[1]	Output		M13	1	LVCMOS25	2.5		12	SLOW
ENABLE[2]	Output		J12	1	LVCMOS25	2.5		12	SLOW
ENABLE[3]	Output		F12	1	LVCMOS25	2.5		12	SLOW
Segments (7)	Output				LVCMOS25	2.5		12	SLOW
Segmento[0]	Output		M12	1	LVCMOS25	2.5		12	SLOW
Segmento[1]	Output		L13	1	LVCMOS25	2.5		12	SLOW
Segmento[2]	Output		P12	2	LVCMOS25	2.5		12	SLOW
Segmento[3]	Output		N11	2	LVCMOS25	2.5		12	SLOW
Segmento[4]	Output		N14	1	LVCMOS25	2.5		12	SLOW
Segmento[5]	Output		H12	1	LVCMOS25	2.5		12	SLOW
Segmento[6]	Output		L14	1	LVCMOS25	2.5		12	SLOW
Scalar ports (0)									

Figura 3.5.18 Asignación de pines

Posteriormente generamos el archivo .bit dando doble click en "Generate Programming File" y el archivo .bit será guardado en la carpeta donde se encuentra nuestro proyecto.

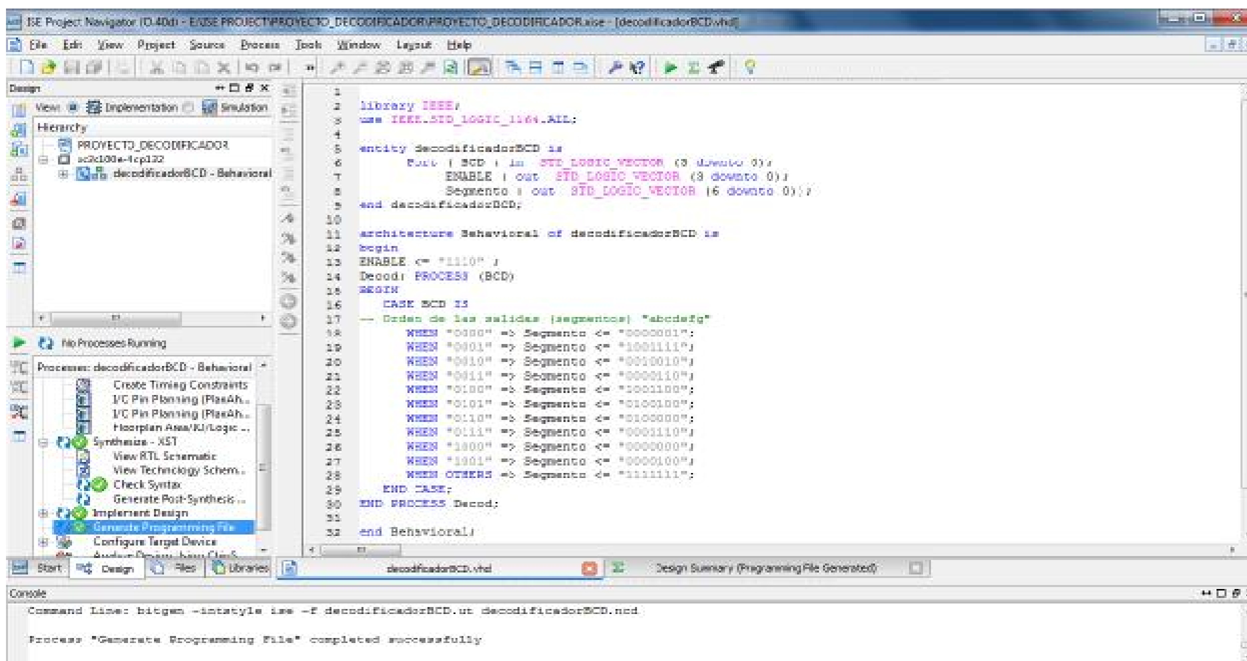


Figura 3.5.19 Creación del archivo .bit

Ya con el archivo .bit generado solo queda bajarlo a la tarjeta para probar su funcionamiento. En las figuras 3.5.20 y 3.5.21 podemos observar y verificar que el comportamiento de nuestro decodificador BCD a 7 segmentos ánodo común es correcto.

Bcd: SW0=1 , SW1=0 , SW2=0 , SW3=0.
Enable: DISP1=0 , DISP2=1 , DISP3=1 , DISP4=1.
Segmento: a=1, b=0 ,c=0 ,d=1 ,e=1 ,f=1,g=1 .

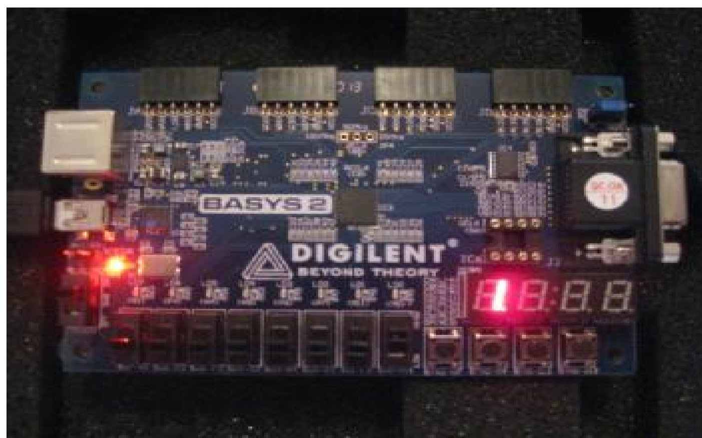


Figura 3.5.20 Representación de numero "1".

Bcd: SW0=0 , SW1=0 , SW2=1 , SW3=0.
Enable: DISP1=0 , DISP2=1 , DISP3=1 , DISP4=1.
Segmento: a=1, b=0 ,c=0 ,d=1 ,e=1 ,f=0,g=0.

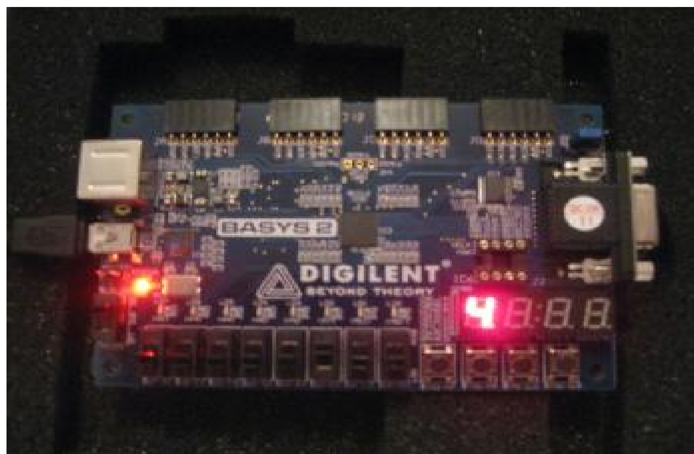


Figura 3.5.21 Representación de numero “4”.

Conclusión

Se modela y se implementa el codificador 8 a 3 y el decodificador de BCD a 7 segmentos ánodo común utilizando VHDL y se comprueba exitosamente su funcionamiento físico utilizando la tarjeta de trabajo BASYS 2. Se observa la facilidad de utilizar la descripción en VHDL para el desarrollo de codificadores y decodificadores.

Práctica 6

Modelado e implementación de Multiplexores y Demultiplexores

Objetivo

Se implementará y simulará el diseño de un multiplexor de 4X1, a través de 2 métodos; el primero mediante la ecuación lógica del multiplexor 4X1 y el segundo por un proceso secuencial de lógica combinatoria. También se implementara un demultiplexor 1X4 con un proceso secuencial de lógica combinatoria.

Desarrollo

Un multiplexor es un circuito combinacional que selecciona una de n líneas de entrada y transmite su información binaria a la salida. La selección de la entrada es controlada por un conjunto de líneas de selección. La relación de líneas de entrada y líneas de selección está dada por la expresión 2n, donde n corresponde al número de líneas de selección y 2n al número de líneas de entrada.

En esta práctica desarrollaremos un multiplexor de 4 entradas. El multiplexor de 4 entradas es un multiplexor de 4 líneas a 1 línea. La figura 3.6.1 muestra el diagrama de bloques del multiplexor. Las entradas son I0, I1, I2 e I3 y la selección viene dada por las entradas S0 y S1. El valor de la salida Y depende de los valores lógicos presentes en las entradas de datos y la selección.

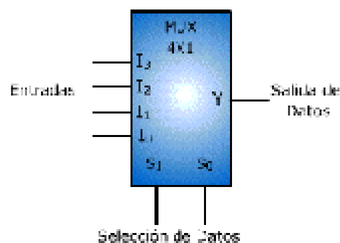


Figura 3.6.1 Diagrama multiplexor 4X1

La tabla de verdad se muestra en la figura 3.6.2. Por ejemplo, sí I0=1, I1=1, I2=0, I3=1 y S1=1, S0=0 entonces Y=I2=0.

Entrada de Selección de datos		Entrada Seleccionada
S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figura 3.6.2 Tabla de verdad de un multiplexor 4X1

El problema consiste en definir un conjunto de expresiones para construir el circuito lógico. La ecuación en cada fila, se obtiene a partir del dato de entrada y la entrada de selección de datos:

La salida es Y= I0, sí S1=0 y S0=0.

$$Y_0 = I0\overline{S1}\,\overline{S0}$$

La salida es Y= I1, sí S1=0 y S0=1.

$$Y_1 = I1\overline{S1}S0$$

La salida es Y= I2, sí S1=1 y S0=0.

$$Y_2 = I2s1\overline{S0}$$

La salida es Y= I3, sí S1=1 y S0=1.

$$Y_3 = I3S1S0$$

Sumando lógicamente las ecuaciones anteriores:

Y = I0·S1'·S0' + I1·S1'·S0 + I2·S1·S0' + I3·S1·S0

El circuito asociado a la ecuación se muestra en la figura 3.6.3.

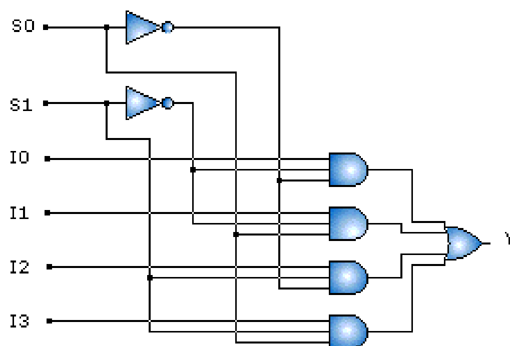


Figura 3.6.3 Circuito lógio multiplexor 4x1

El código VHDL de la ecuacion lógica del multiplexor es:

S0 <= (I0 AND NOT(S1) AND NOT(S0)) OR (I1 AND NOT(S1) AND S0) OR (I2 AND S1 AND NOT(S0)) OR (I3 AND S1 AND S0);

Ahora crearemos un proyecto llamado PRACTICA6, y le añadiremos una fuente llamada MUX1 con las siguientes entradas y salidas, ver figura 3.6.4.

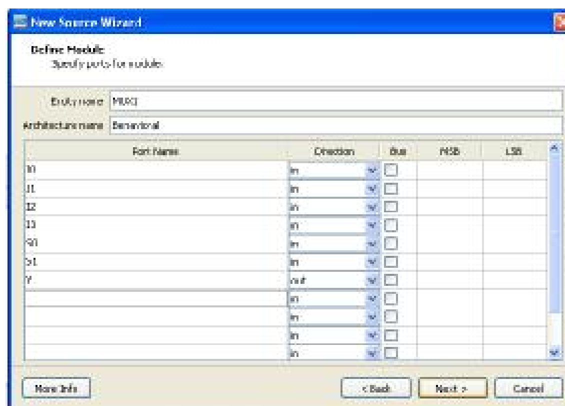


Figura 3.6.4 Entrada y salidas MUX1

Ya creado el proyecto introduciremos el código VHDL que generamos para función lógica del multiplexor, como se muestra en la figura 3.6.5, y compilaremos nuestro proyecto para verificar que no tenga errores (ver Práctica 1).

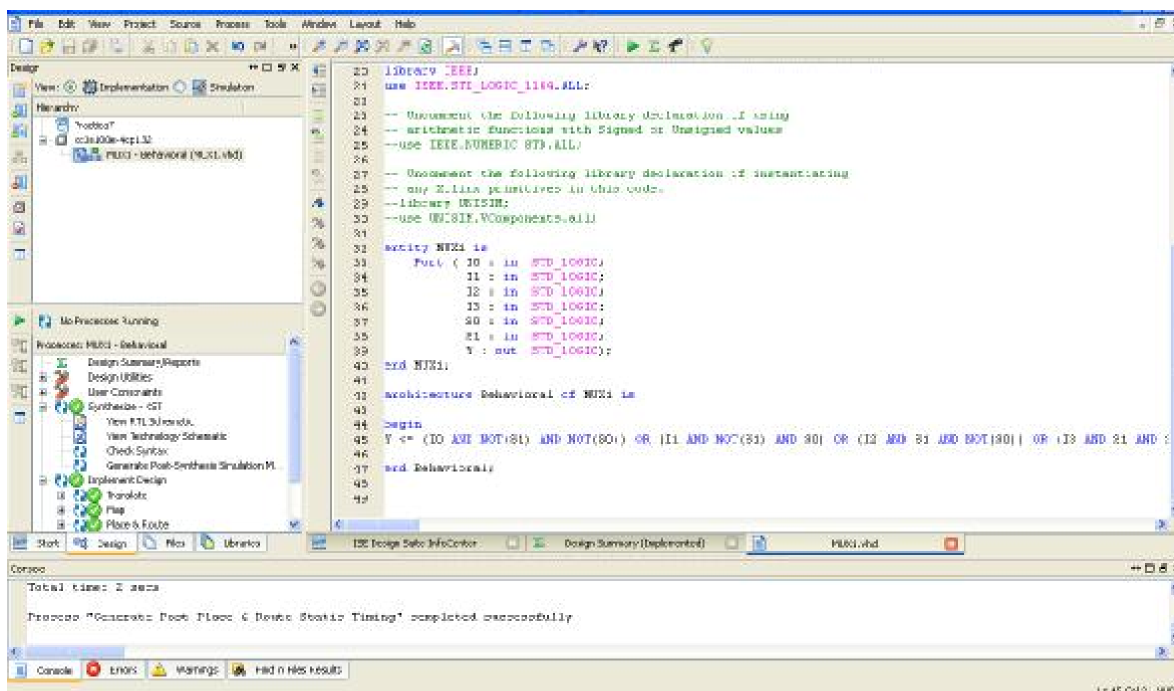


Figura 3.6.5 Compilación correcta del multiplexor

Ahora se simulara el multiplexor (ver practica 2) para verificar que cumple con la tabla de verdad de la figura 3.6.2. Se necesitara configurar seis señales de reloj, cuatro de entrada (I0, I1, I2 e I3) y dos de selección (S0 y S1). Para las entradas tendremos los siguientes valores;

Las configuraciones de reloj se muestran en la figura 3.6.6.

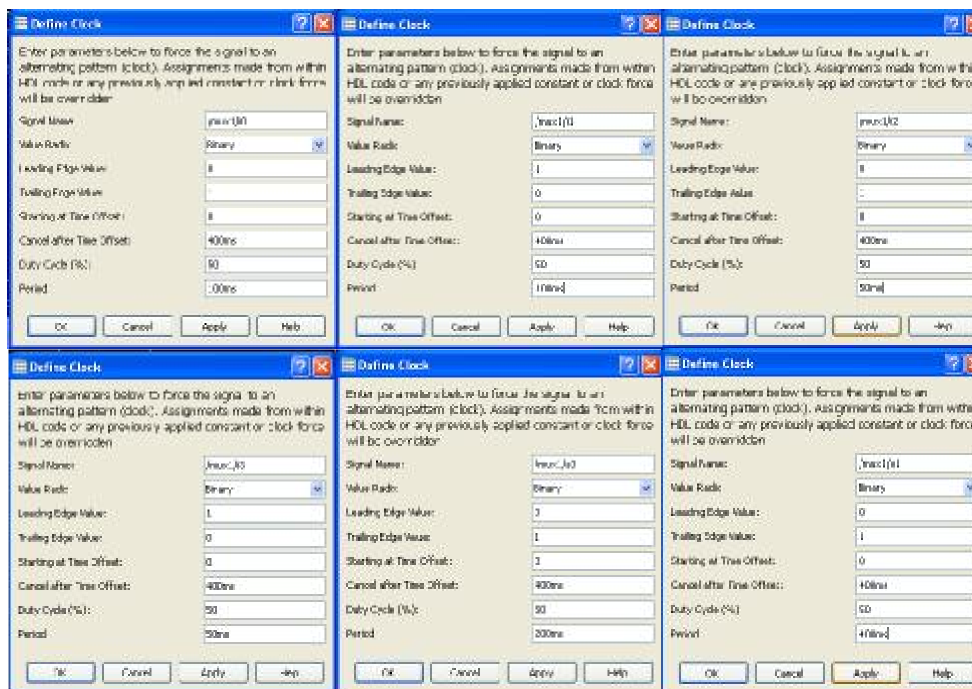


Figura 3.6.6 Configuraciones de las entradas y salidas.

En la figura 3.6.7 se puede observar lo siguiente:

Cuando S0=0 y S1=0, Y=I0

Cuando S0=0 y S1=1, Y=I1

Cuando S0=1 y S1=0, Y=I2

Cuando S0=1 y S1=1, Y=I3

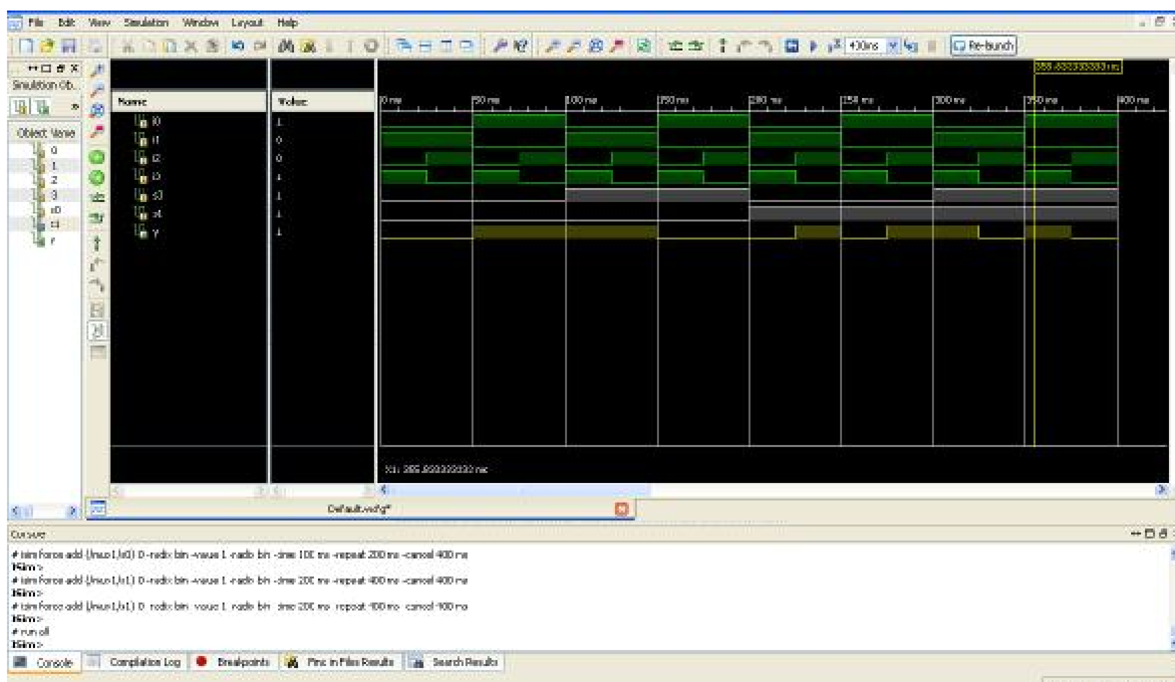


Figura 3.6.7 Simulación multiplexor.

Ahora se diseñará el multiplexor por medio de un process. Un process es una parte de código en la cual las instrucciones se ejecutan en secuencia (siguiendo el orden de escritura), tienen las siguientes características:

- Una misma arquitectura puede contener varios PROCESS.
- Todos los process se ejecutan en paralelo. (un PROCESS es equivalente a una instrucción concurrente compleja).
- El juego de instrucciones utilizables dentro de un process es diferente del juego de instrucciones concurrentes.
- El orden de escritura de las instrucciones afecta los resultados de simulación y de síntesis.
- En simulación, un process se dice dormido hasta que las señales susceptibles de activarlo cambian de estado (lógica combinatoria, lógica sincrónica y lógica sincrónica con inicialización asincrónica).

Estructura general de un process:

[etiqueta:] process (lista_de_sensibilidad)

-- parte declaratoria

-- declaración eventual de variables ...

begin -- parte operatoria

-- instrucciones secuenciales (if, case, loop...)

end process;

De la tabla de verdad del multiplexor tendremos el siguiente código.

```
process(S,I0,I1,I2,I3)

begin

case S is

  when "00" => Y <= I0;

  when "01" => Y <= I1;

  when "10" => Y <= I2;

  when others => Y <= I3;

end case;

end process;
```

También debemos cambiar la entidad para S.
S: in STD_LOGIC_VECTOR(1 downto 0)
Sustituimos el código en ISE y verificamos síntesis y diseño, figura 3.6.8.

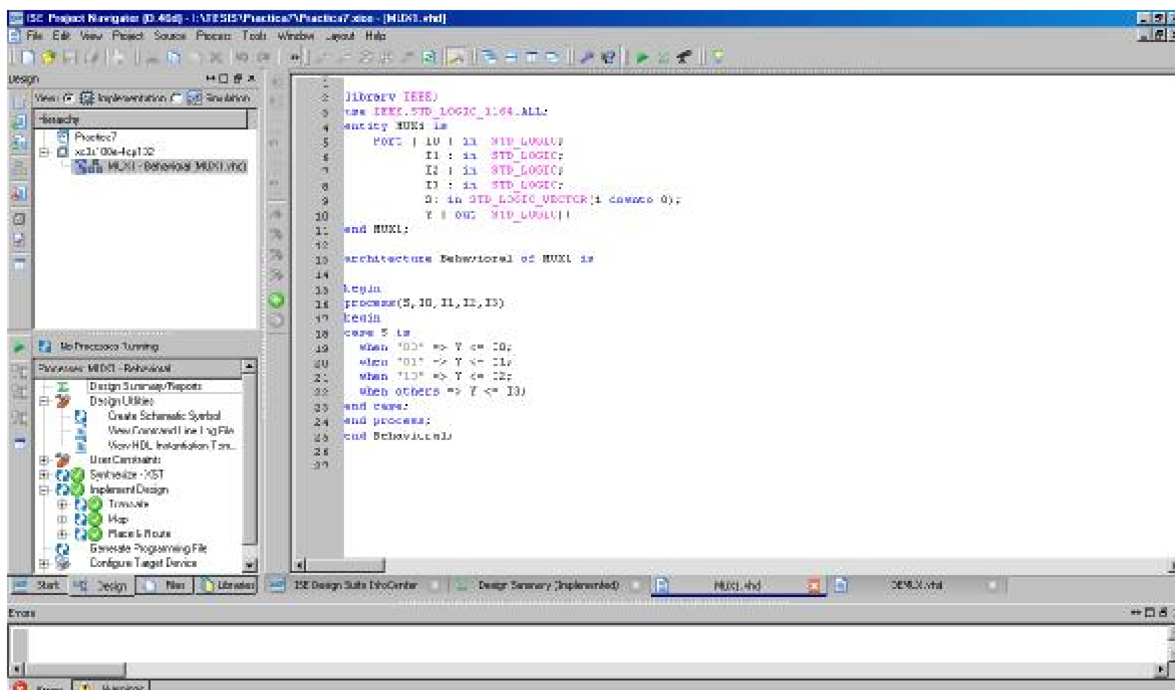


Figura 3.6.8 Verificación de síntesis y diseño con Process

La simulación en ISim se realizará en 2 partes, la primera para s=00 y s=01 y la segunda para s=10 y s=11. Las entradas I0, I1, I2 e I3, tendrán la misma configuración para las 2 etapas, figura 3.6.9.

Figura 3.6.9 Configuraciones de entradas I0,I1,I2 e I3

En la figura 3.6.10 podemos ver la configuración de S para la primera etapa.

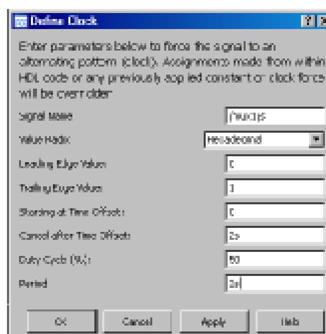


Figura 3.6.10 Configuracion de entrada s

La simulación de la primera etapa se puede apreciar en la figura 3.6.11, donde solo las señales de entrada I0 e I1 han sido seleccionadas.

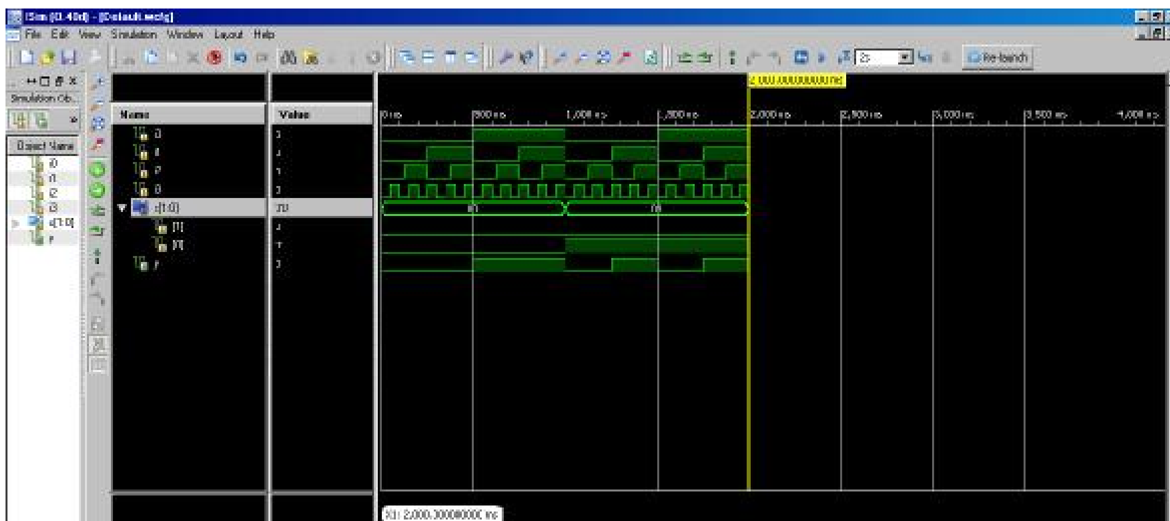


Figura 3.6.11 Primera etapa simulación

En la figura 3.6.12 podemos ver la configuración de S para la segunda etapa.

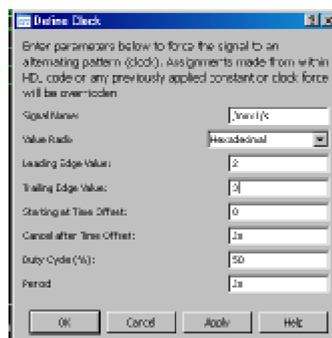


Figura 3.6.12 Configuración de entrada S

La simulación de la primera etapa se puede apreciar en la figura 3.6.13, donde solo las señales de entrada I2 y I3 han sido seleccionadas.

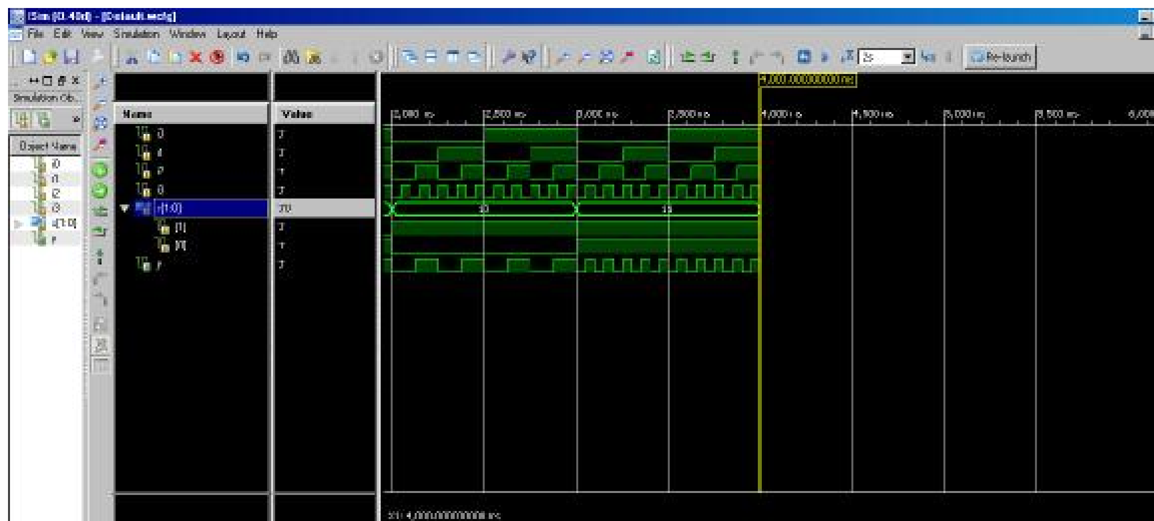


Figura 3.6.13 Segunda etapa simulación

El código que dejaremos para la implementación en la tarjeta será el del process.

Ahora diseñaremos un demultiplexor. Un demultiplexor es un circuito combinacional que recibe información en una sola línea y la transmite a una de 2n líneas posibles de salida. La selección de una línea de salida específica se controla por medio de los valores de los bits de n líneas de selección. La operación es contraria al multiplexor. La figura 3.6.14 muestra el diagrama de bloques del demultiplexor.

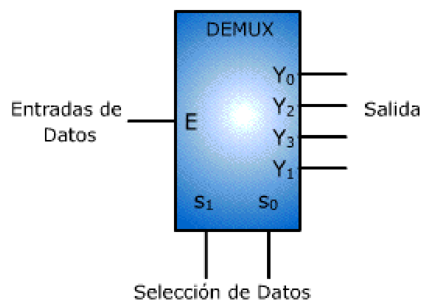


Figura 3.6.14 Diagrama demultiplexor 1X4

La figura 3.6.15 muestra el circuito logico de un demultiplexor de 1 a 4 líneas. Las líneas de selección de datos activan una compuerta cada vez y los datos de la entrada pueden pasar por la compuerta hasta la salida de datos determinada. La entrada de datos se encuentra en común a todas las AND.

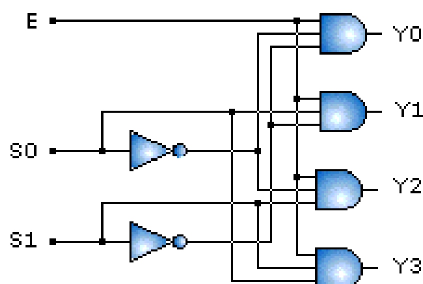


Figura 3.6.15 Circuito logico demultiplexor 1X4

La tabla de verdad se muestra en la figura 3.6.16

E	S0	S1	Y0	Y1	Y2	Y3
E	0	0	E	0	0	0
E	0	1	0	E	0	0
E	1	0	0	0	E	0
E	1	1	0	0	0	E

Figura 3.6.16 Tabla de verdad demultiplexor 1X4

El demultiplexor se implementará con instrucciones secuenciales utilizando un process. De la tabla de verdad del demultiplexor podemos observar que para diferentes combinaciones de selección tenemos diferentes salidas, por ejemplo:

Cuando S0=0 y S1=0, Y0=E

Cuando S0=0 y S1=1, Y1=E

Cuando S0=1 y S1=0, Y2=E

Cuando S0=1 y S1=1, Y3=E

Entonces utilizaremos el siguiente código VHDL para nuestro process será:

```
process(E,S)
begin
case S is
  when "00" => Y0 <= E; Y1 <= '0'; Y2 <= '0'; Y3 <='0';
  when "01" => Y1 <= E; Y0 <= '0'; Y2 <= '0'; Y3 <='0';
  when "10" => Y2 <= E; Y0 <= '0'; Y1 <= '0'; Y3 <='0';
  when others => Y3 <= E;  Y0 <= '0'; Y1 <= '0'; Y2 <='0';
end case;
end process;
```

Es hora de añadir una nueva fuente llamada DEMUX a nuestro proyecto, con las entradas y salidas de la figura 3.6.17.

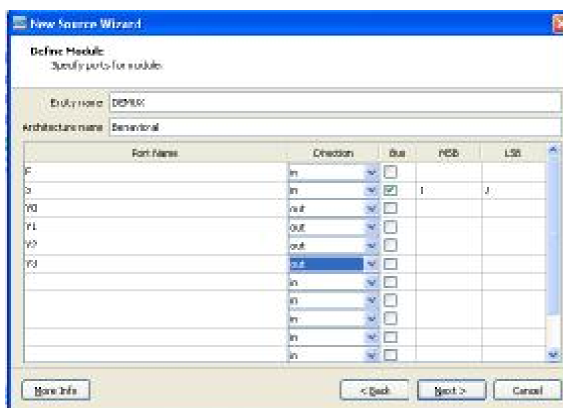


Figura 3.6.17 Entradas y salidas DEMUX

Ya creada la nueva fuente introduciremos el código VHDL del process del demultiplexor, como se muestra en la figura 3.6.18 y compilamos nuestro proyecto para verificar que no haya errores (ver práctica 1).

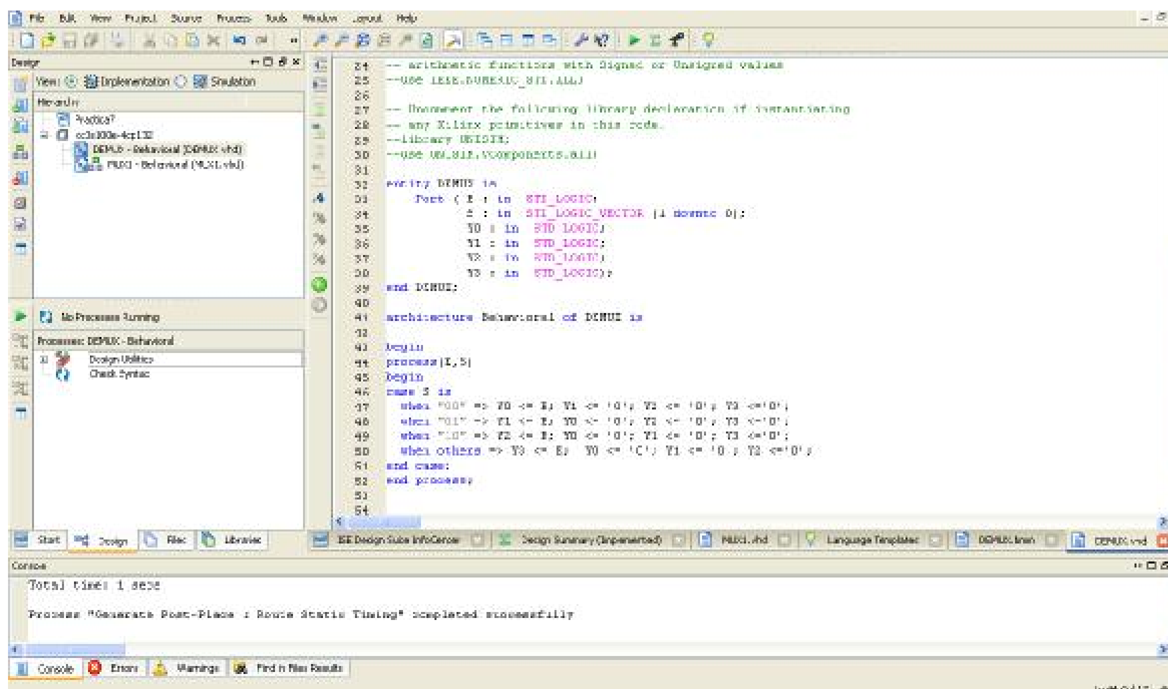
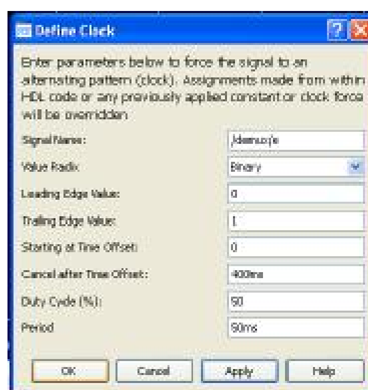


Figura 3.6.18 Compilación correcta del demultiplexor

Ahora se simulará el demultiplexor para revisar que cumple con la tabla de verdad de la figura 3.6.10. La simulación se correrá en 4 etapas, en cada etapa el bus tendrá un diferente valor constante y una misma señal de entrada de reloj.

Nota: Esta vez se simulará en 4, para apreciar las salidas.

La configuración de la entrada E se muestra en la figura 3.6.19, se tiene una señal con periodo de 100ms y ciclo duty del 50%, esta señal no se modificara en las etapas de simulación.



3.6.19 Configuracion entrada E

Para la primera etapa tenemos la configuración del bus con una constante hexdecimal 0, figura 3.6.20.

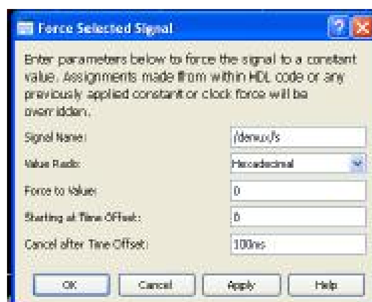


Figura 3.6.20 S=0

Para simular la primera etapa, solo simularemos 100ms, figura 3.6.21.

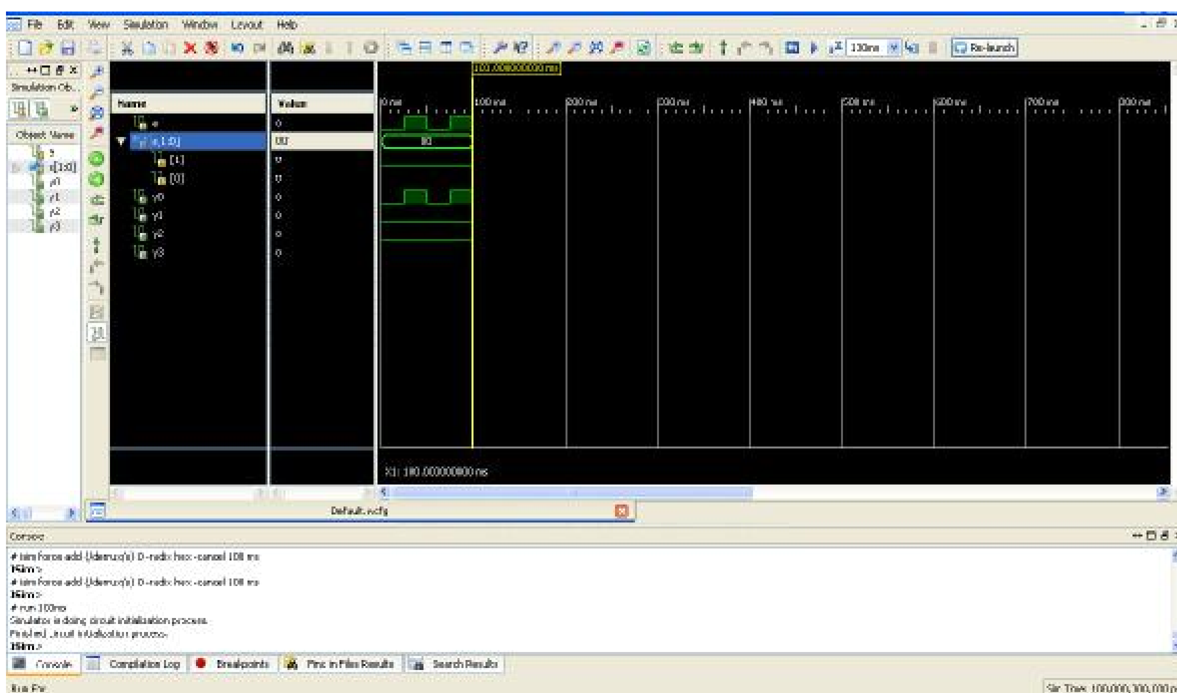


Figura 3.6.21 Primera etapa simulación

Para la segunda etapa tenemos la configuración del bus con una constante hexdecimal 1 (ver figura 3.6.22).

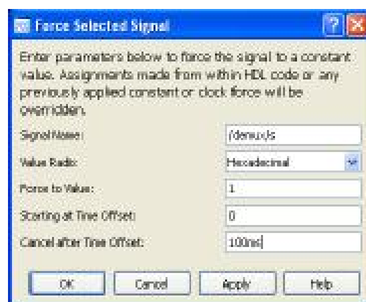


Figura 3.6.22 S=1

Para simular la segunda etapa, simularemos los siguientes 100ms (ver figura 3.6.23).

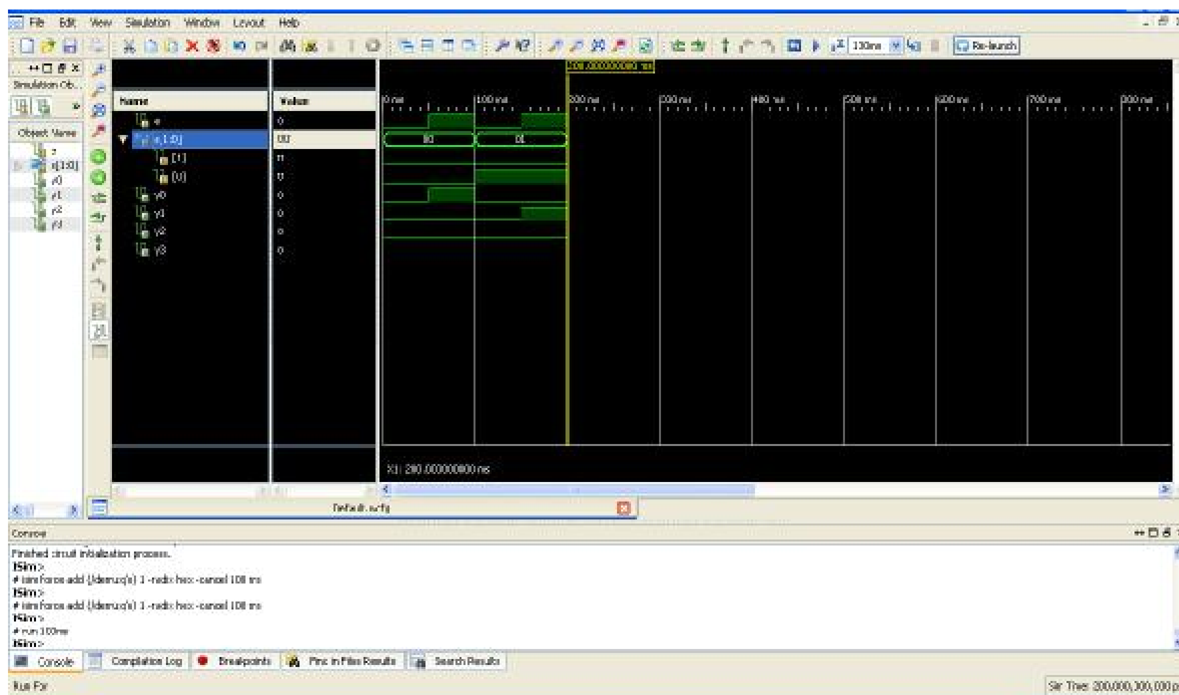


Figura 3.6.23 Segunda etapa simulación

Para la tercera etapa tenemos la configuración del bus con una constante hexdecimal 2, figura 3.6.24.

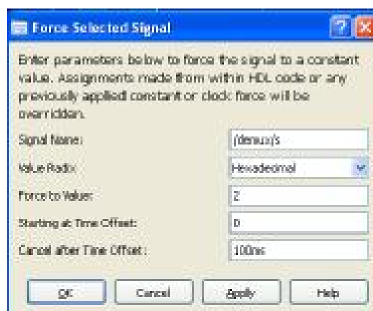


Figura 3.6.24 S=2

Para simular la tercera etapa, simularemos los siguientes 100ms (ver figura 3.6.25).

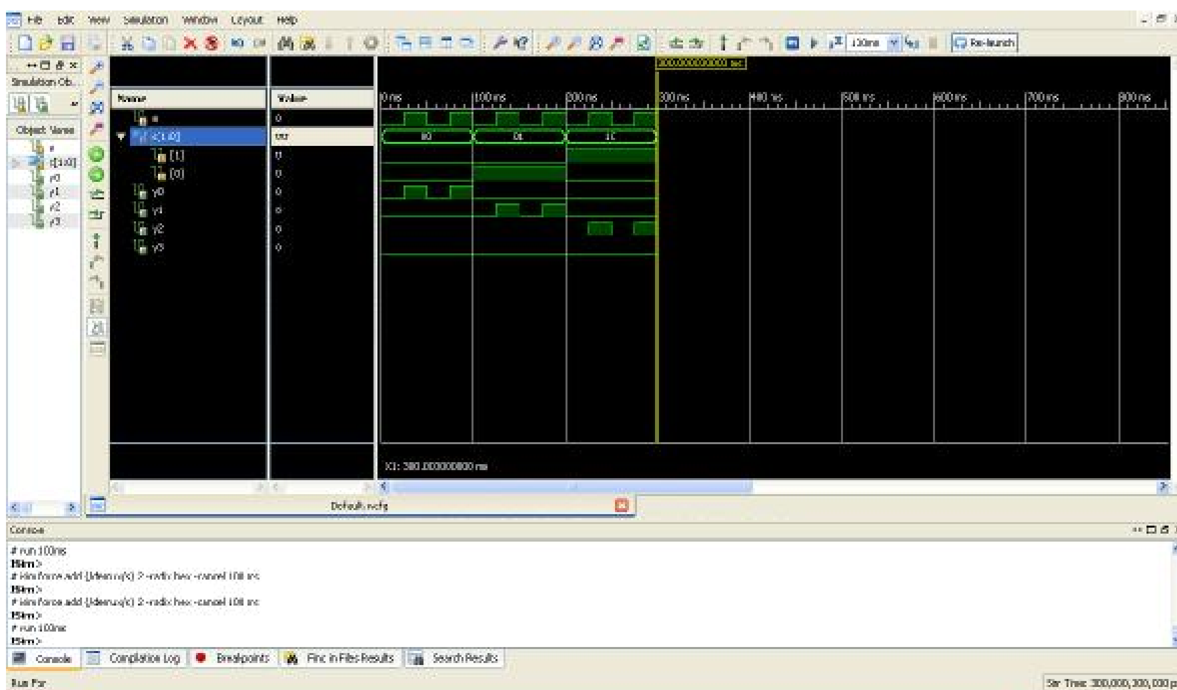


Figura 3.6.25 Tercera etapa simulación

Para la cuarta etapa tenemos la configuración del bus con una constante hexdecimal 3 (ver figura 3.6.26).

Figura 3.6.26 S=3

Para simular la cuarta y última etapa, simularemos los siguientes 100ms (ver figura 3.6.27).

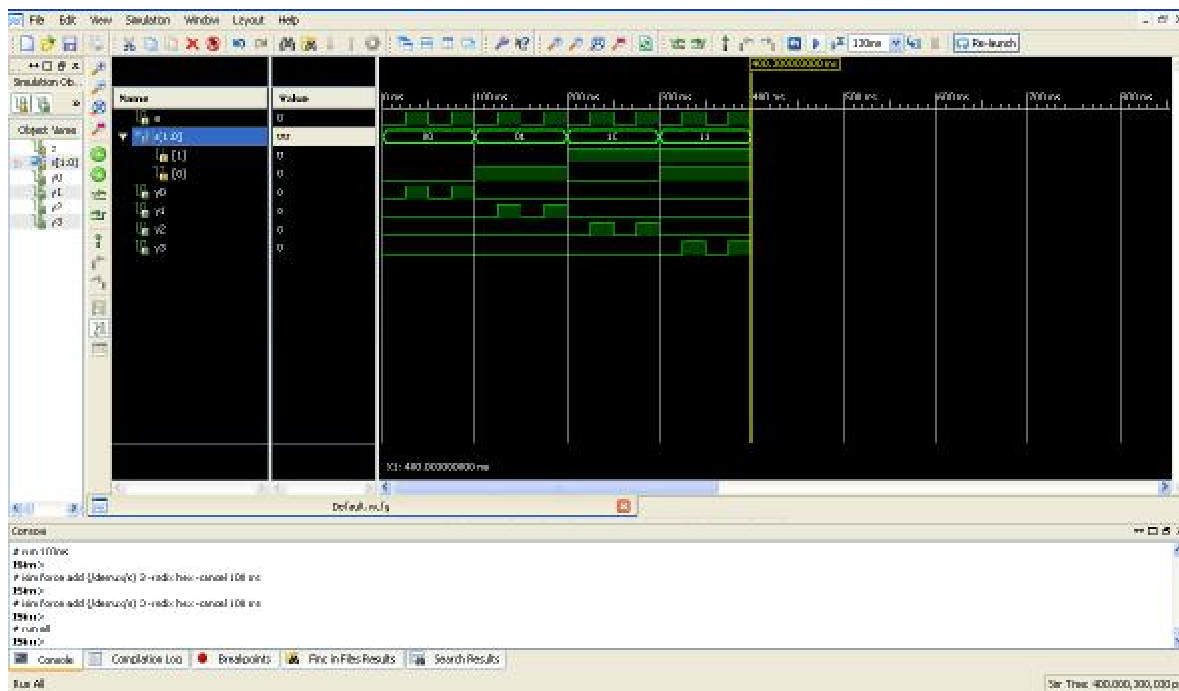


Figura 3.6.27 Cuarta etapa simulación

Para la asignación de pines usaremos la herramienta PlanAHead como lo hicimos en la Práctica 1. Primero asignaremos los pines para el multiplexor dando click en I/O Pin Planning (PlanAhead)–Pre–Synthesis, debe estar seleccionado el MUX1 en las fuentes VHDL, ver figura 3.6.28.

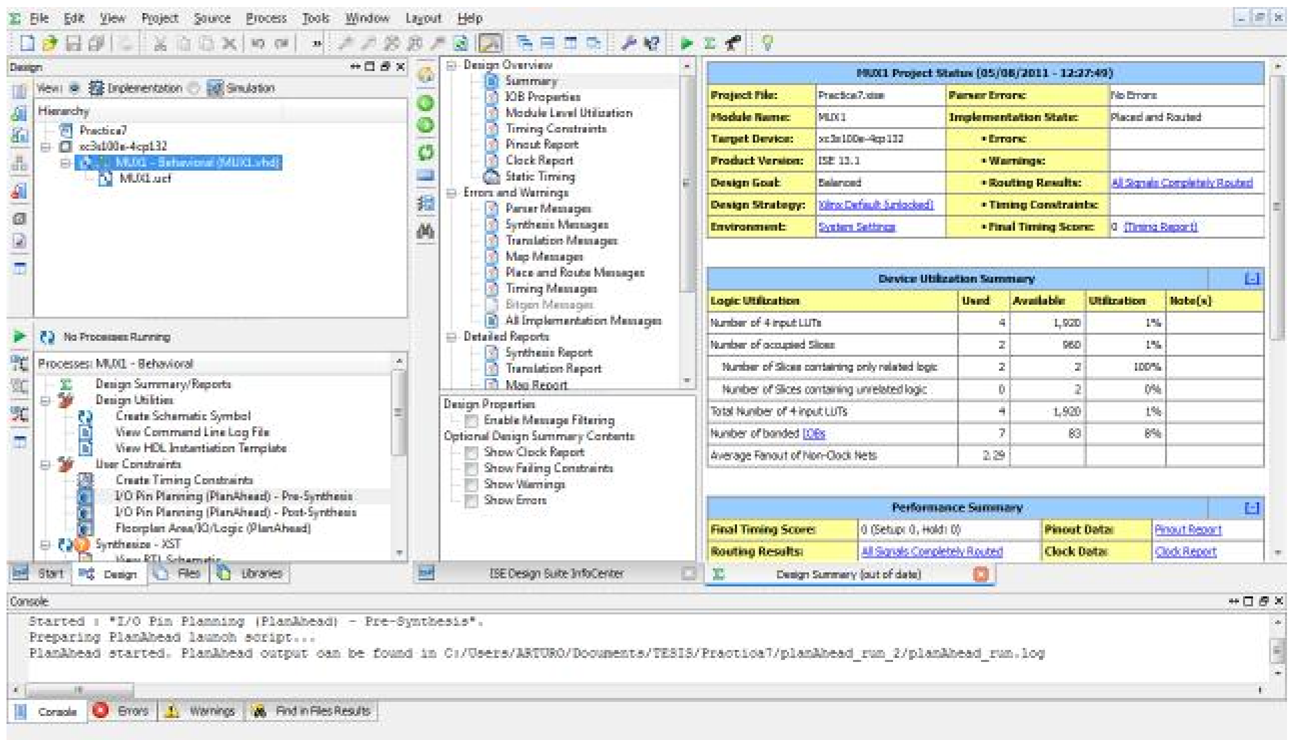


Figura 3.6.28 Pin Planning (PlanAhead)-Presynthesis MUX1

Ya abierta la aplicación PlanAhead asignaremos los pines de acuerdo a la figura 3.6.29. Donde las entradas serán los SW 0 al SW 3, las entradas de selección serán los la salida SW 4 al SW 5 y la salida será el LD 0.

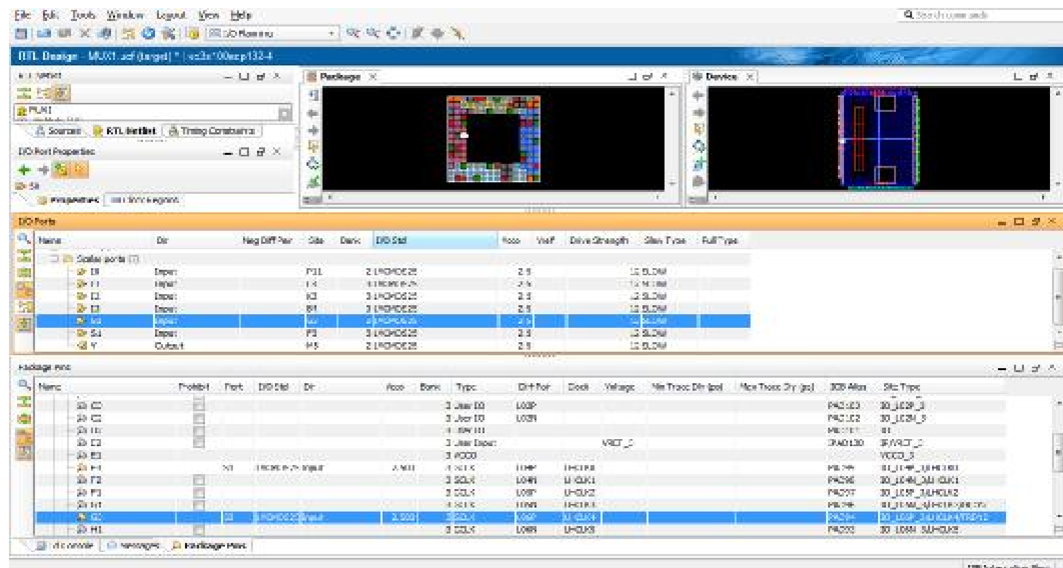


Figura 3.6.29 Mapeo de pines I/O MUX1

Cerraremos la aplicación PlanAhead, con esto se creará un archivo .ucf adjunto a la fuente MUX1.VHDL. Regresamos a nuestro proyecto en ISE Proyect, daremos click derecho en “Generate Programming File” y en Process Properties verificamos que se encuentre JTAG Clock como valor seleccionado, figura 3.6.30.

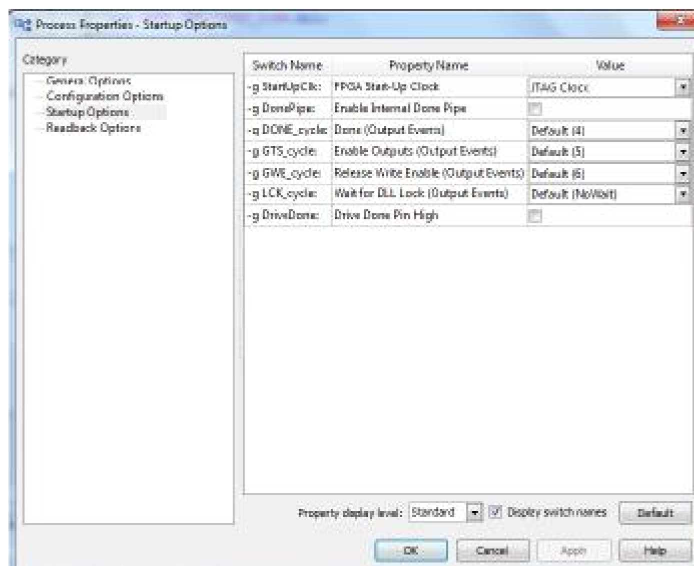
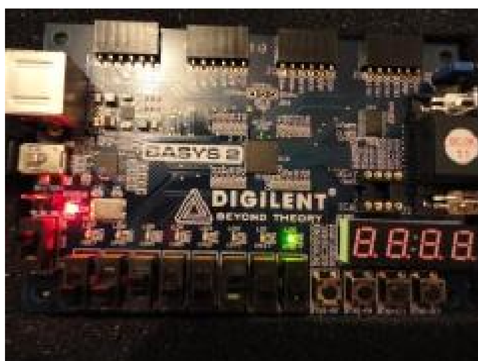


Figura 3.6.30 FPGA Start-Up Clok

Con esto ya podemos generar nuestro archivo .bit y bajarlo a nuestra tarjeta, ver práctica 1. En las Figuras 3.6.31 , 3.6.32, 3.6.33 y 3.6.34 podemos observar y verificar que el comportamiento de nuestro multiplexor.

Figura 3.6.31 S0=0, S1=0,Y=I0

Figura 3.6.32 S0=1, S1=0, Y=I1



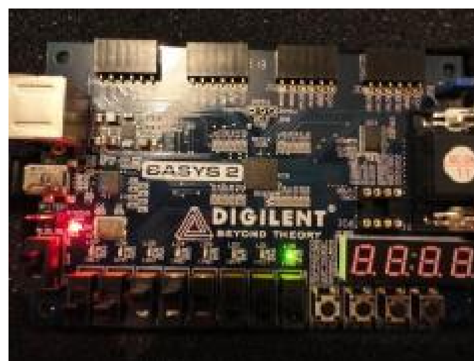


Figura 3.6.33 S0=1, S1=0,Y=I2 **Figura 3.6.34 S0=1, S1=1, Y=I3**

Para el demultiplexor haremos lo mismo, seleccionaremos la fuente DEMUX.VHDL y asignaremos sus pines I/O como se muestra en la figura 3.6.35. Con esta asignación tendremos que los SW0 y el SW1 serán los selectores, el SW2 será la entrada E y los LD0 al LD2 serán las salidas.

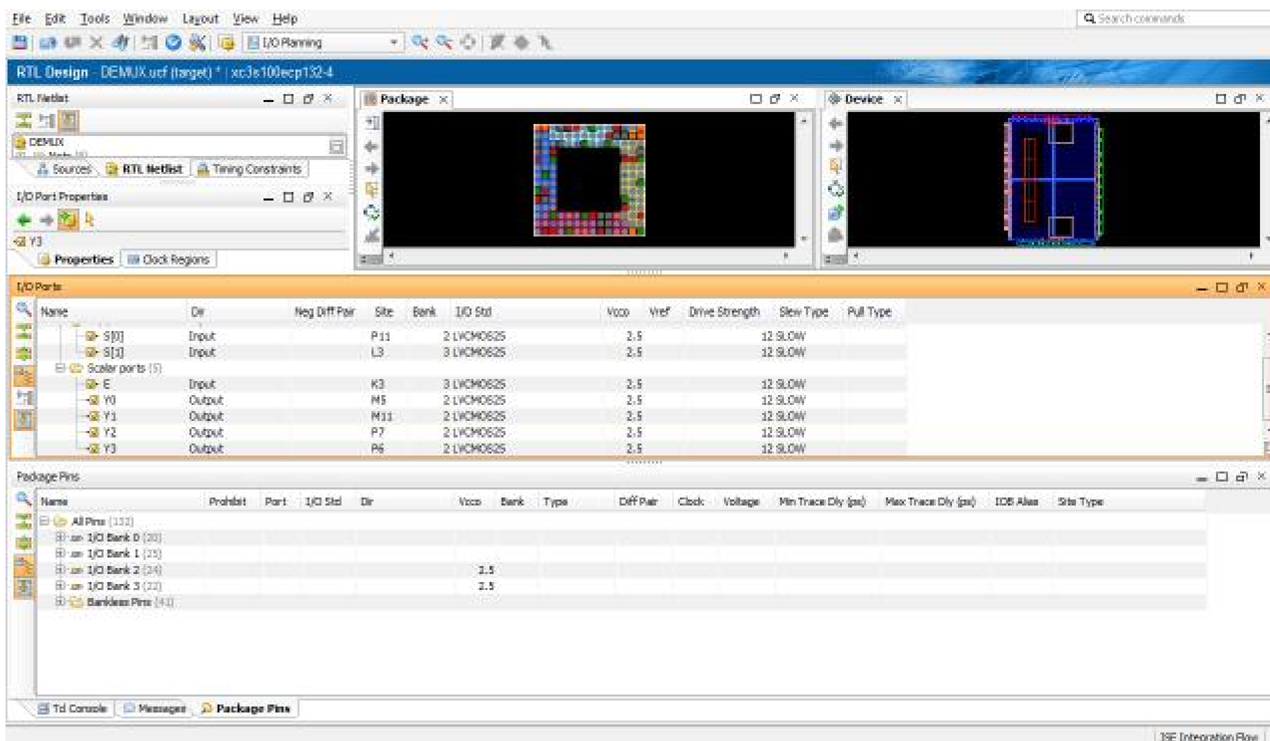


Figura 3.6.35 Asignación de pines I/O DEMUX

Ahora solo falta generar el programa .bit y bajarlo a la tarjeta. En las figuras 3.6.36, 3.6.37, 3.6.38 y 3.6.39, se muestran el comportamiento de nuestro demultiplexor.

Figura 3.6.36 S0=0, S1=0, Y0=E

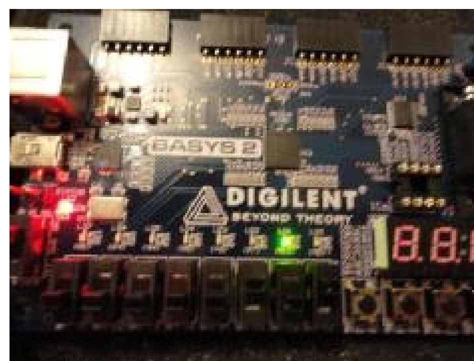


Figura 3.6.37 S0=1, S1=0, Y1=E

Figura 3.6.38 S0=0, S1=1, Y2=E

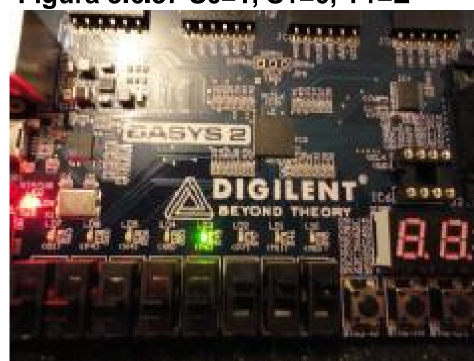


Figura 3.6.39 S0=1, S1=1, Y3=E

Conclusión

Se modeló e implementó un multiplexor y demultiplexor en lenguaje VHDL. El multiplexor se implementó con la ecuación lógica y con un process, mientras que el demultiplexor solo con process. Se observa claramente que la implementación con process es más rápida, ya que no se necesita llegar a una ecuación lógica, el código se puede obtener fácilmente de la tabla de verdad.

Práctica 7

Modelado e Implementación de Comparadores

Objetivo:

Implementar un comparador de dos vectores de 4 bits y enviar el resultado a un display en cuál se podrá apreciar cuál de los dos vectores es mayor, menor o si son iguales.

Desarrollo:

La comparación de la igualdad de dos palabras binarias es una operación comúnmente utilizada en sistemas de cómputo e interfaces de dispositivos. A un circuito que compara dos palabras binarias e indica si son iguales se le conoce como comparador (ver figura 3.7.1). Algunos comparadores también indican una relación aritmética (mayor, menor o igual) entre las palabras. Estos dispositivos se denominan comparadores de magnitud.

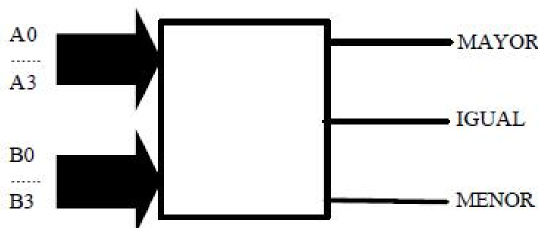


Figura 3.7.1 Comparador de 4 bits

E.datos	E.datos	E.datos	E.datos	E.expa	E.expa	E.expa	salidas	salidas	salidas
A3,B3	**A2,B2**	**A1,B1**	**A0,B0**	<	>	=	**A>B**	**A<B**	**A=B**
A3>B3	X	X	X	X	X	X	H	L	L
A3<B3	X	X	X	X	X	X	L	H	L
A3=B3	A2>B2	X	X	X	X	X	H	L	L
A3=B3	A2<B2	X	X	X	X	X	L	H	L
A3=B3	A2=B2	A1>B1	X	X	X	X	H	L	L
A3=B3	A2=B2	A1<B1	X	X	X	X	L	H	L
A3=B3	A2=B2	A1=B1	A0>B0	X	X	X	H	L	L
A3=B3	A2=B2	A1=B1	A0<B0	X	X	X	L	H	L
A3=B3	A2=B2	A1=B1	A0=B0	H	L	L	H	L	L
A3=B3	A2=B2	A1=B1	A0=B0	L	H	L	L	H	L
A3=B3	A2=B2	A1=B1	A0=B0	X	X	H	L	L	H
A3=B3	A2=B2	A1=B1	A0=B0	H	H	H	L	L	L
A3=B3	A2=B2	A1=B1	A0=B0	L	L	L	H	H	L

Figura 3.7.2 Tabla de verdad comparador de 4 bits.

Comenzaremos con la creación del proyecto en el ambiente de desarrollo Xilinx ISE, el cual nombraremos como “comparador4b”, y el módulo VHDL se llamara “compara”, se tendrán que crear las siguientes variables A, C, ENABLE y RESULTADO (ver figura 3.7.3).

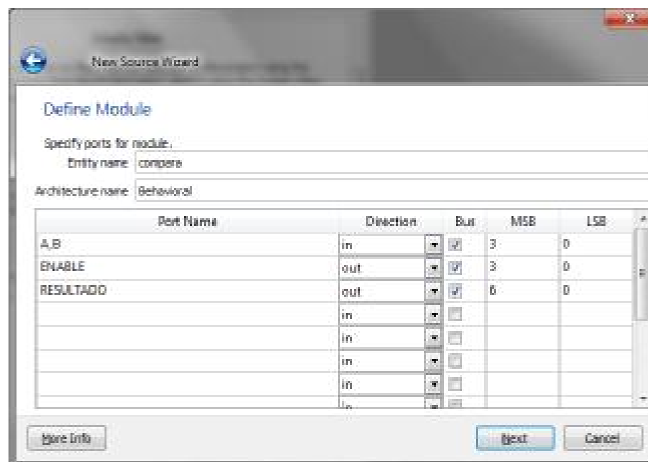


Figura 3.7.3 Puertos de entrada/salida

- Donde **A y C** son los vectores de entrada (se ha escogido “A” y “C” por la comodidad de ser representados en el Display).
- **ENABLE**, nos servirá para habilitar el display (ver practica 5)
- **RESULTADO**, enviará el resultado de comparación al display.

Hecho lo anterior comenzaremos a diseñar el código en VHDL.

--Inicia código

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity compara is

   Port ( A,C: in  STD_LOGIC_VECTOR (3 downto 0);

                          ENABLE : out  STD_LOGIC_VECTOR (3 downto 0);

          RESULTADO : out  STD_LOGIC_VECTOR (6 downto 0));

end compara;

architecture Behavioral of compara is

begin

ENABLE<="1110"; --Habilita display 1

            process(A,C)
```

```
        begin
                if A=C then
                RESULTADO<="1110110"; -- cuando son iguales enviará el signo = al display
                elsif A>C then
                RESULTADO<="0001000";-- cuando A>C enviará A al display
                else
                RESULTADO<="0110001";-- caso contrario enviará C al display
                end if;
    end process;
end Behavioral;
```

--Fin de código

Compilamos y verificamos que el código no tenga errores (ver figura 3.7.4).

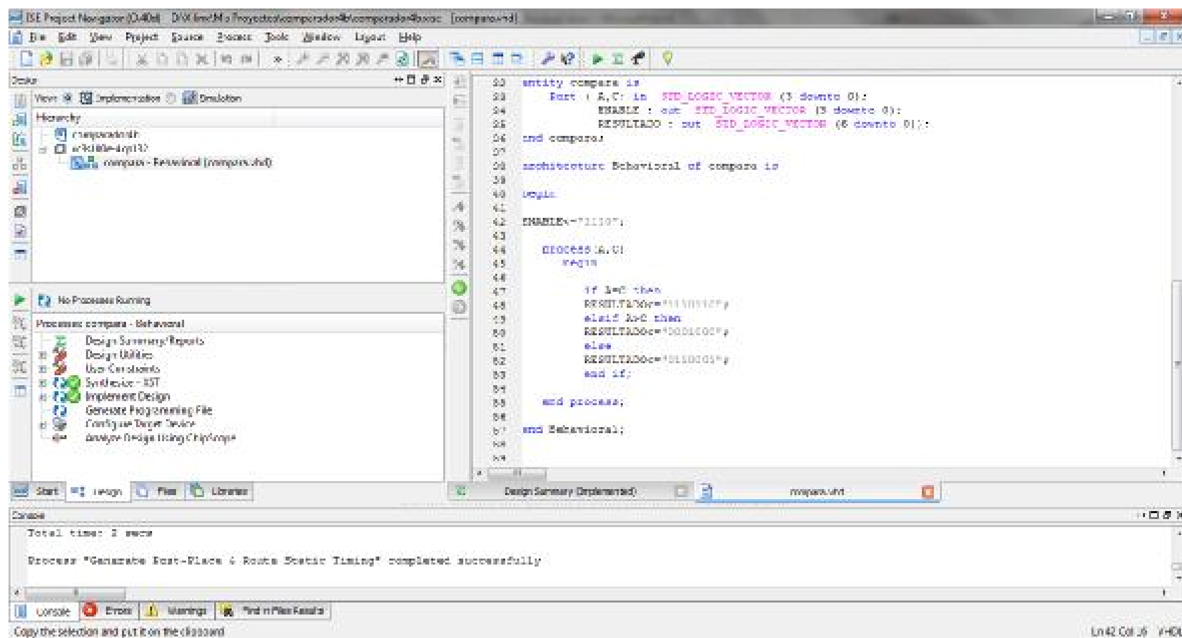


Figura 3.7.4 Compilación exitosa

Una vez verificado el código se procederá a realizar la simulación del proyecto, en el cual se asignaran algunos valores constantes para verificar su funcionalidad, empezaremos con un valor A>C de tal modo que la asignación de valores quedara como se muestra en la figura 3.7.5, y obtendremos la simulación de la imagen 3.7.6.

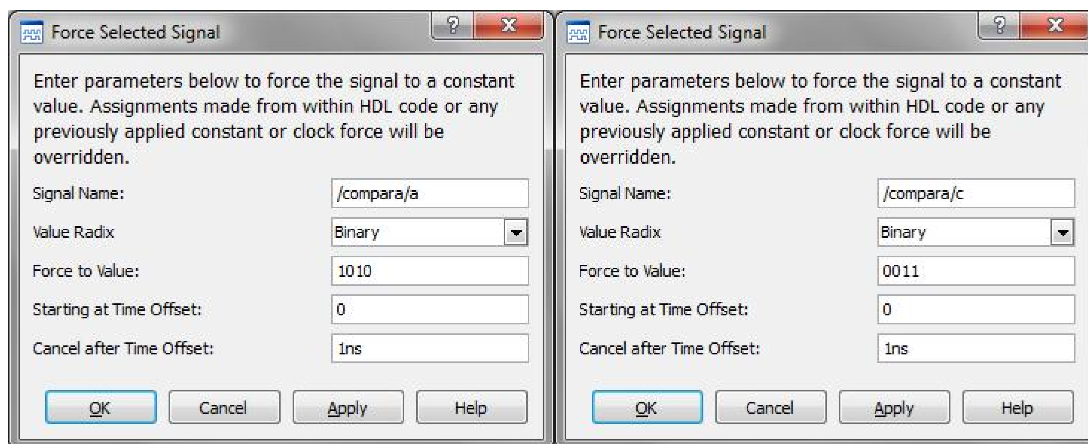


Figura 3.7.5 Caso A>C

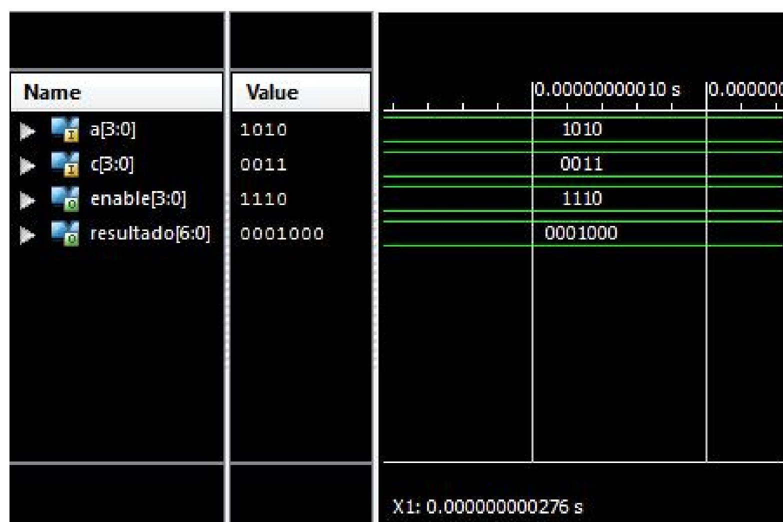


Figura 3.7.6 Simulación caso A>C

Evidentemente no podemos visualizar el resultado del display, pero tomando el valor de la variable “RESULTADO” se puede apreciar que activa los segmentos “a,b,c,e,f y g” que en conjunto forman la letra “A” en el display, lo que verifica que A>C, la variable “ENABLE” siempre mostrara el valor “1110” ya que indica que el display se ha habilitado como ánodo común.

Para el caso A<C asignaremos los valores como se muestra en la figura 3.7.7, y obtendremos la simulación de la figura 3.7.8.

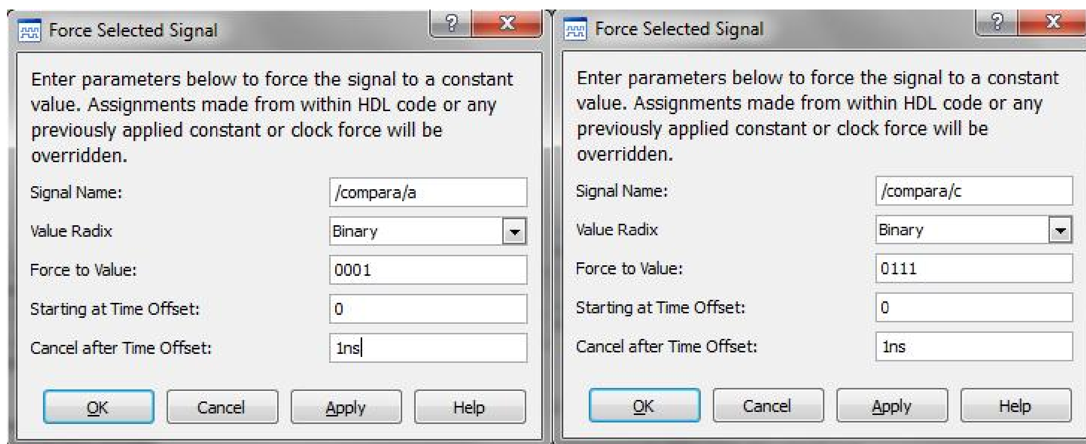


Figura 3.7.7 Caso A<C

Figura 3.7.8 Simulación A<C

Para este caso podemos apreciar que la variable “RESULTADO” se han activado los segmentos “a,d,e y f” del display, que en conjunto forman la letra “C”, lo que verifica que A<C.

Y para el último caso en que A=C asignaremos los valores como se muestra en la figura 3.7.9 y obtendremos la simulación de la figura 3.7.10.

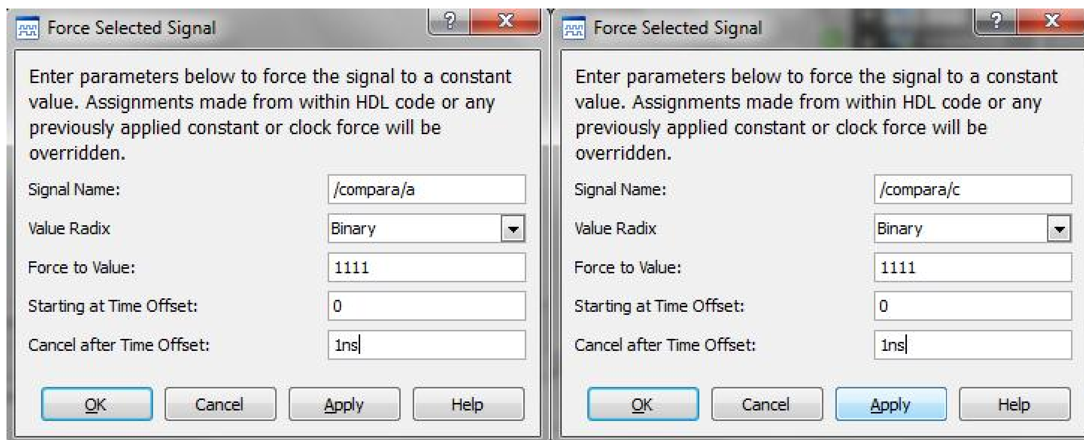


Figura 3.7.9 Caso A=C

Figura 3.7.10 Simulación A=C

En este último caso podemos apreciar que en la variable "RESULTADO" se activan los segmentos "d y g" del display, que forman el signo "=" lo que verifica que el vector A=C.

Como ya se ha verificado que la simulación arroja los valores esperados, se procederá a asignar los puertos correspondientes a las variables de entrada y salida para la tarjeta BASYS 2, los cuales quedaran como se muestra en la figura 3.7.11.

Figura 3.7.11 Asignación de pines

Después de haber hecho la asignación de pines volvemos a compilar nuestro proyecto para verificar que no se han producido errores tal y como se muestra en la figura 3.7.12 y finalmente generamos nuestro archivo .bit para bajarlo a la tarjeta BASYS 2.

Figura 3.7.12 Compilación exitosa y archivo .bit generado

Finalmente descargamos el archivo .bit a la tarjeta BASYS 2 y comprobamos para los tres casos que se simularon (3.7.6, 3.7.8 y 3.8.10), observaremos que todos los casos se cumplen como se muestra en las figuras 3.7.13, 3.7.14 y 3.7.15.

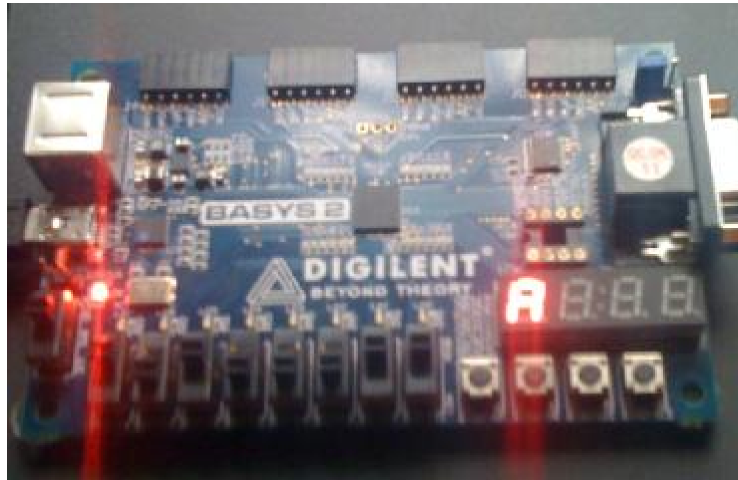


Figura 3.7.13 Implementación en tarjeta BASYS 2 en A>C

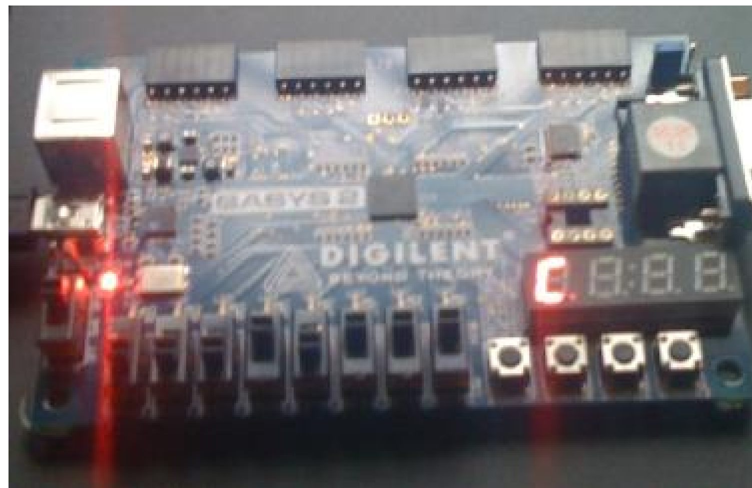


Figura 3.7.14 Implementación en tarjeta BASYS 2 en C>A

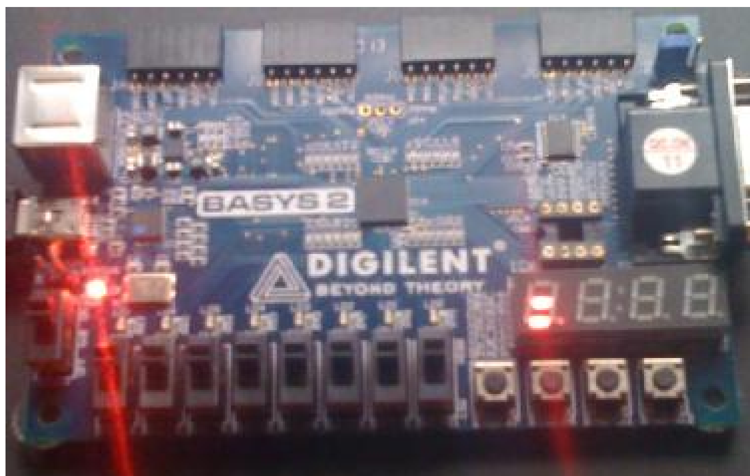


Figura 3.7.15 Implementación en tarjeta BASYS 2 en A=C

Conclusión

Se ha diseñado con éxito un código en VHDL que compara dos vectores de 4 bits y que muestra en un display el que es mayor, menor o si son iguales y se implementó el programa generado por dicho código en la tarjeta BASYS 2 el cual funcionó correctamente tal como se había previsto en las simulaciones.

Práctica 8

Modelado e implementación de una ALU

Objetivo:

Desarrollar e implementar en VHDL un diseño que permita realizar diferentes operaciones aritméticas y lógicas características de un ALU (Unidad Lógica Aritmética) de 4 bits con selector también de 4 bits.

Desarrollo:

El diseño contará con las siguientes operaciones:

- A + B
- A – B
- B – A
- A AND B
- A OR B
- A XOR B
- NOT A
- NOT B
- A NAND B
- A * 2
- B * 2
- A / 2
- B /2
- A ++
- B ++
- A XNOR B

Para realizar la implementación de nuestro diseño propondremos la entidad como se muestra en la figura 3.8.1. El diseño contará con dos entradas de 4 bits “A” y “B”, un selector de 4 bits “OPCION”, una salida de 4 bits “RESULTADO” y un bit más de salida para desplegar el “ACARREO”

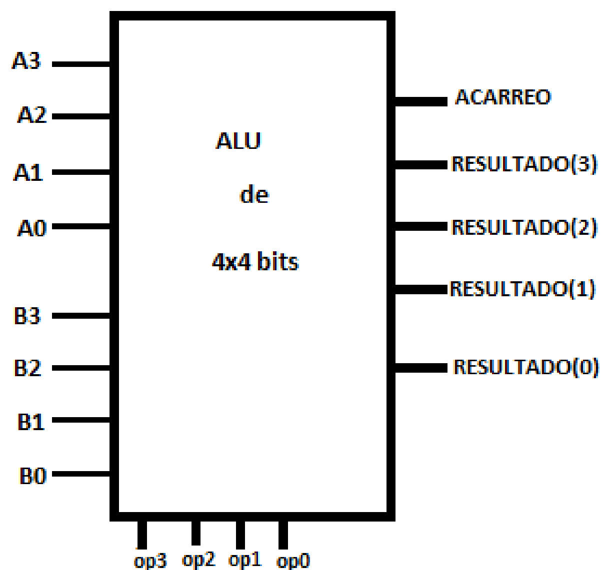


Figura 3.8.1 ALU de 4x4 bits

Para llevar a cabo las operaciones propondremos el orden que se muestra en la tabla 3.8.2. En donde op3, op2, op1, op0 será un vector de 4 bits para seleccionar la instrucción que se realizará, nombraremos a este vector OPCION en nuestro diseño.

op3	op2	op1	op0	INSTRUCCIÓN
0	0	0	0	A+B
0	0	0	1	A-B
0	0	1	0	B-A
0	0	1	1	A AND B
0	1	0	0	A OR B
0	1	0	1	A XOR B
0	1	1	0	NOT A
0	1	1	1	NOT B
1	0	0	0	A NAND B
1	0	0	1	A * 2
1	0	1	0	B * 2
1	0	1	1	A / 2
1	1	0	0	B / 2
1	1	0	1	A ++
1	1	1	0	B ++
1	1	1	1	A XNOR B

Figura 3.8.2 Tabla de operaciones

Las entradas se declaran como números sin signo para poder realizar las operaciones aritméticas propuestas. Al comienzo de la arquitectura se declaran las señalas A1, B1, R, MULTI de 5 bits las cuales nos ayudarán a calcular de una manera más rápida y sencilla el acarreo para las operaciones que lo generan.

A continuación podemos ver el diseño final de nuestra ALU de 4 bits

```
--Inicia Codigo

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use ieee.numeric_std.all;

entity operadores is

    Port ( A,B,OPCION : in  UNSIGNED  (3 downto 0);

         ACARREO: out STD_ULOGIC;

        RESULTADO  : buffer  UNSIGNED (3 downto 0));

end    operadores;

architecture Behavioral of operadores is

  SIGNAL A1: UNSIGNED (4 downto 0);

  SIGNAL B1: UNSIGNED (4 downto 0);

  SIGNAL R: UNSIGNED (4 downto 0);

begin

  WITH OPCION SELECT

  RESULTADO <= (A + B)    when "0000",

           (A - B)    when "0001",

           (B - A) when "0010",

           (A AND B)  when "0011",

          (A OR B)   when "0100",

          (A XOR B)  when "0101",

          (NOT A)    when "0110",

          (NOT B)    when "0111",
```

```
(A NAND B) when "1000",
(A SLL 1)  when "1001",
(B SLL 1)  when "1010",
(A SRL 1)  when "1011",
(B SRL 1)  when "1100",
(A + "0001")  when "1101",
(B + "0001")  when "1110",
(A XNOR B)    when "1111",
"0000"       when others;
 A1(4) <= '0';
 A1(3 downto 0)<= A;
B1(4) <= '0';
B1(3 downto 0)<= B;
R <= (A1 + B1)      when  (OPCION="0000") ELSE
(A1 - B1)     when  (OPCION="0001") ELSE
(B1 - A1)     when  (OPCION="0010") ELSE
(A1 SLL 1)     when  (OPCION="1001") ELSE
(B1 SLL 1)     when  (OPCION="1010") ELSE
(A1 + "00001") when  (OPCION="1101") ELSE
(B1 + "00001") when  (OPCION="1110") ELSE
 "00000"  ;
ACARREO <= R(4) WHEN (OPCION="0000") OR (OPCION="0001")
OR (OPCION="0010") OR (OPCION="1001")
OR (OPCION="1010") OR (OPCION="1101")
OR (OPCION="1110") ELSE '0' ;
end Behavioral;
--Termina Código
```

Ahora utilizaremos la herramienta ISE Project para copiar el diseño de nuestra ALU, el proyecto llevará el nombre PRACTICA8ALU (ver figura 3.8.3).

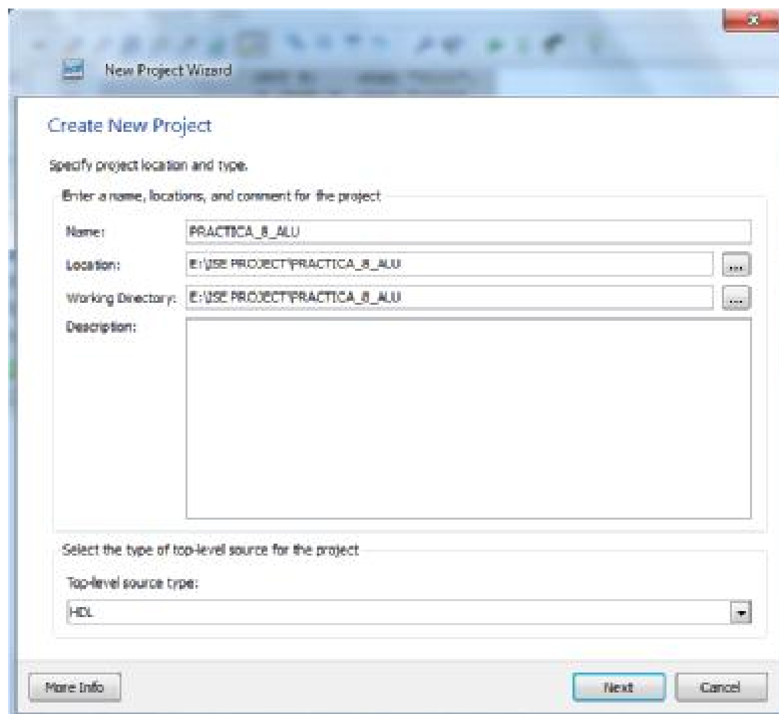


Figura 3.8.3 Creación de Proyecto

Ahora seleccionamos el tipo de fuente VHDL Module, la entidad la nombraremos operadores (ver figura 3.8.4). Las entradas y salidas no las definiremos en la definición de modulo.

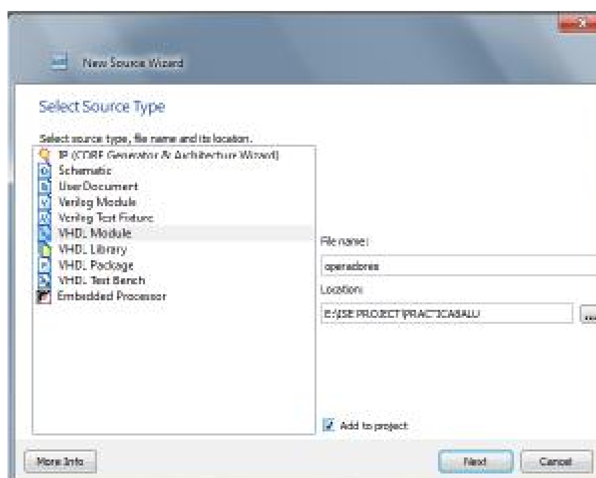


Figura 3.8.4 Nombre de la Entidad

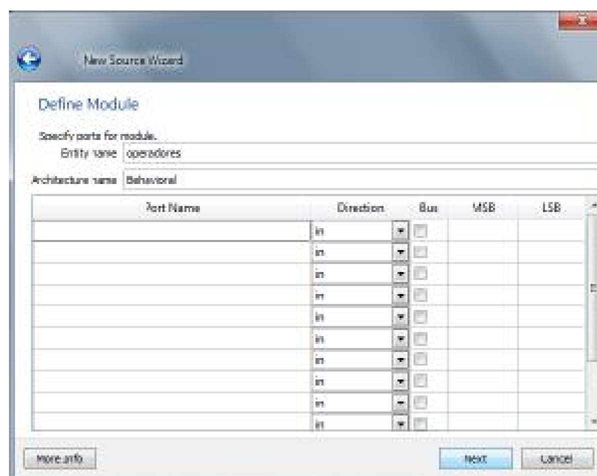


Figura 3.8.5 Nombre de los puertos

Ahora copiamos el diseño que se mostró anteriormente remplazando el código que genera ISE Project y guardamos los cambios realizados (ver figura 3.8.6).

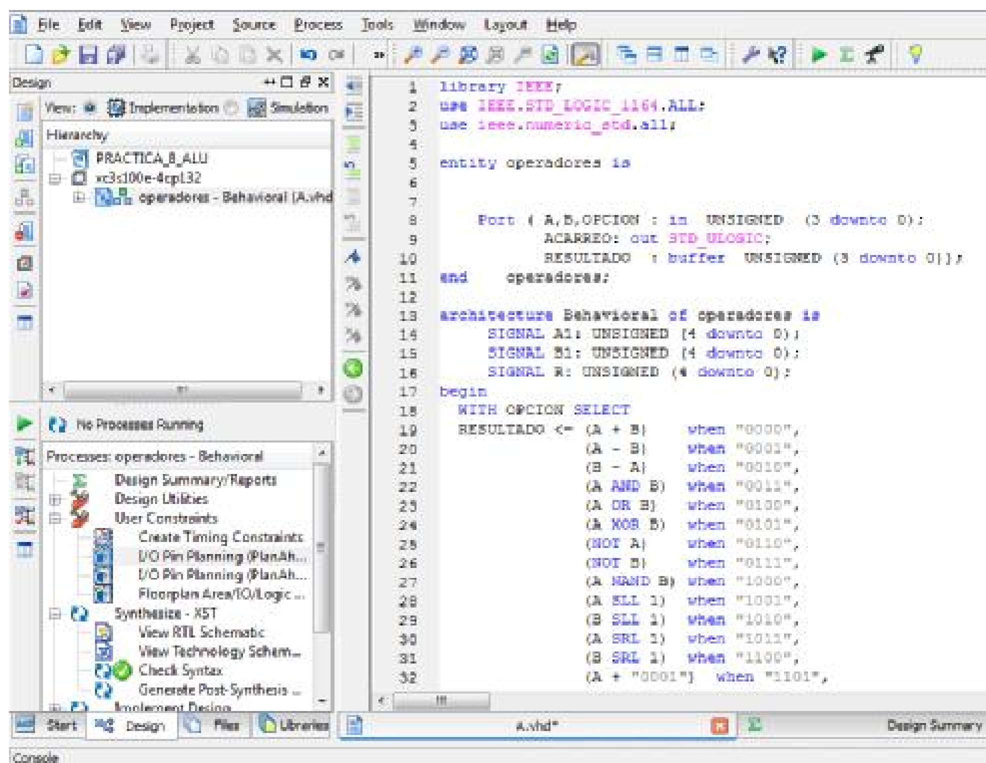


Figura 3.8.6 Proyecto ALU en ISE

Para comprobar el funcionamiento utilizaremos la herramienta ISim, por lo que previamente debemos de revisar que no existan errores de sintaxis en nuestro diseño. Probamos con las operaciones "A + B", "A NAN B" y "B / 2" cuando OPCION tome los valores 0000,1000, 1100 respectivamente y los valores de la entradas serán A = "1110" y B= "1100".

Para el caso OPCION=0000 y las entradas A = “1110” y B= “1100” , las salidas serán RESULTADO=1010 ACARREO=1. (ver figura 3.8.7)

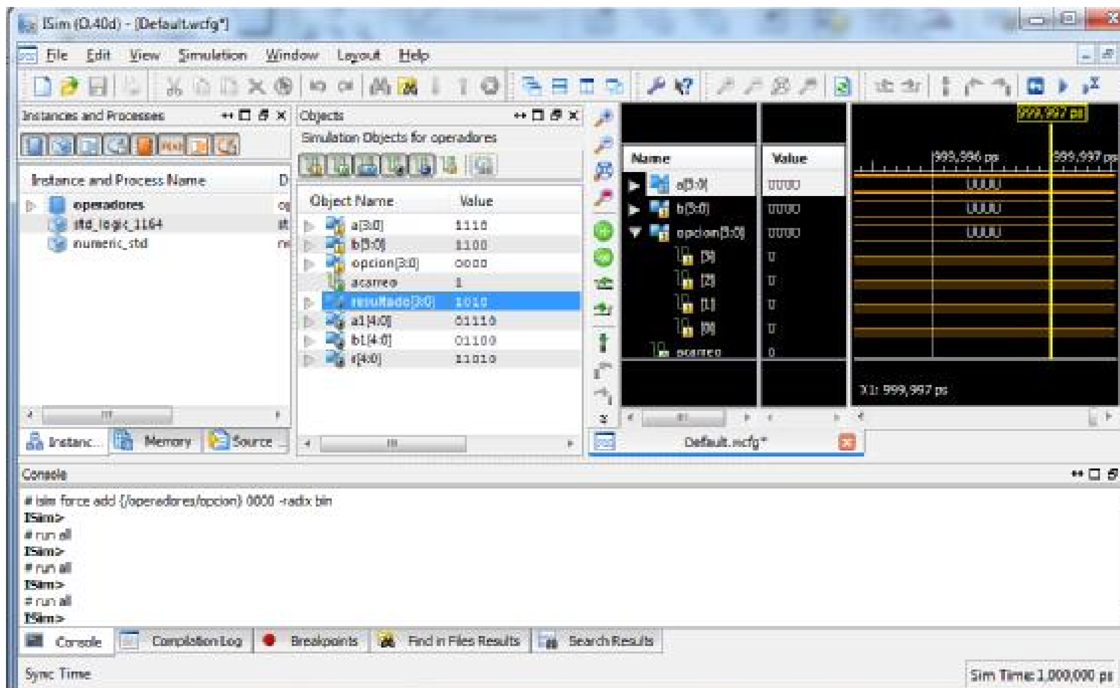


Figura 3.8.7 Simulación para OPCION=0000, A=1110 Y B=1100

Para el caso OPCION=1000 y las entradas A = “1110” y B= “1100”, las salidas serán RESULTADO=0011 ACARREO=0. Ver Figura 3.8.8

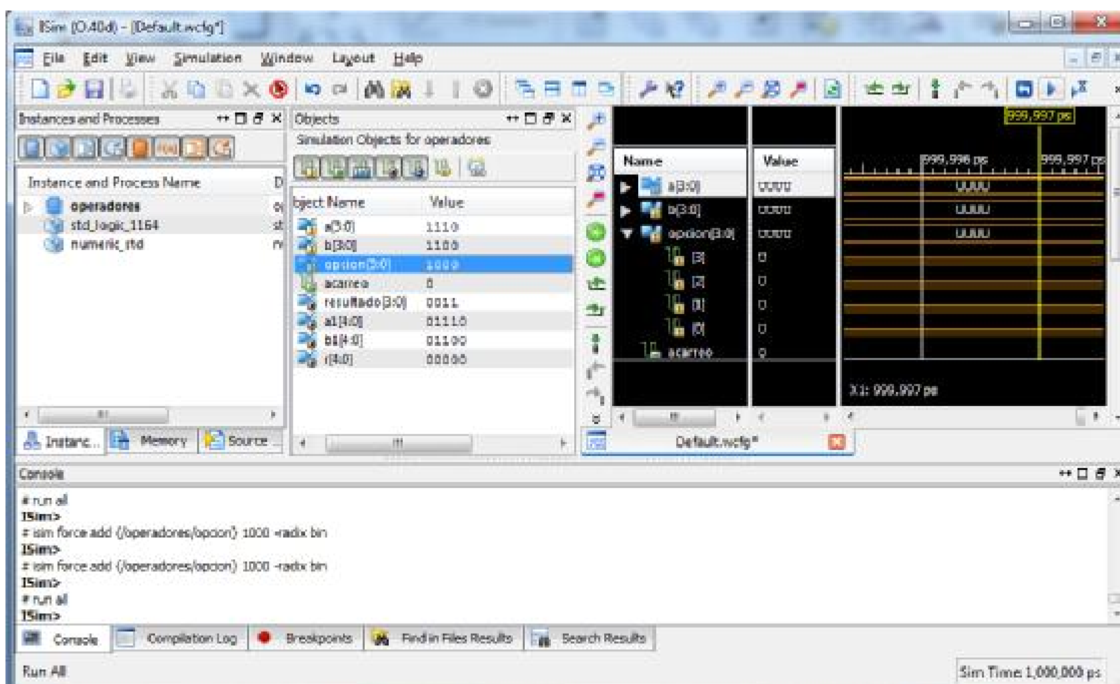


Figura 3.8.8 Simulación para OPCION=1000, A=1110 Y B=1100

Para el caso OPCION=1100 y las entradas A = “1110” y B= “1100”, las salidas serán RESULTADO=0110 ACARREO=0. Ver Figura 3.8.8

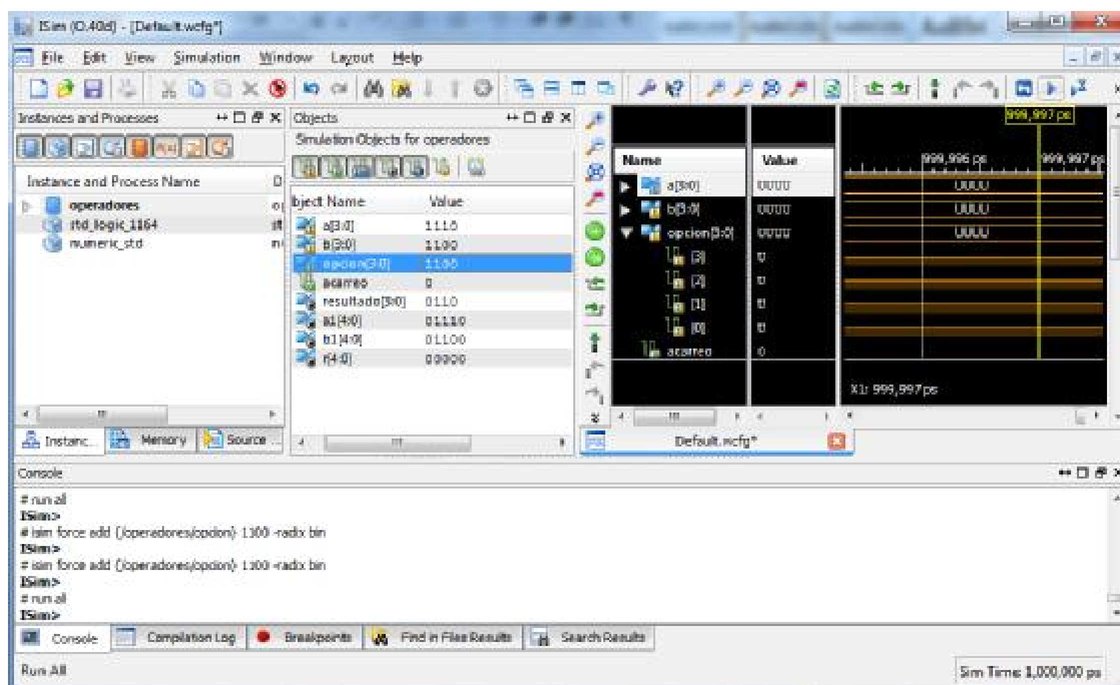


Figura 3.8.9 Simulación para OPCION=1100, A=1110 Y B=1100

Ahora bajaremos nuestro diseño a la tarjeta de prueba BASYS2, para ello asignaremos los puertos de la siguiente manera, ver Figura 3.8.10.

A(3)=N3

A(2)=E2

A(1)=F3

A(0)=G3

B(3)=B4

B(2)=K3

B(1)=L3

B(0)=P11

ACARREO=N5

RESULTADO(3)=P6

RESULTADO(2)=P7

RESULTADO(1)=M11

RESULTADO(0)=M5

OPCION(3)=A7

OPCION(2)=M4

OPCION(1)=C11

OPCION(0)=G12

PRACTICA_8_ALU - [E:\ISE PROJECT\PRACTICA_8_ALU\planAhead_run_2\PRACTICA_8_ALU.ppr] - PlanAhead 13.1

File Edit Tools Window Layout View Help

I/O Planning

RTL Design - operadores.ucf (target) * | xc3s100ecp132-4

I/O Ports

Name	Dir	Neg Diff Pair	Site	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
All ports (17)									
A (4)	Input				LVCMOS25	2.5		12	SLOW
A[0]	Input		G3	3	LVCMOS25	2.5		12	SLOW
A[1]	Input		F3	3	LVCMOS25	2.5		12	SLOW
A[2]	Input		E2	3	LVCMOS25	2.5		12	SLOW
A[3]	Input		N3	2	LVCMOS25	2.5		12	SLOW
B (4)	Input				LVCMOS25	2.5		12	SLOW
B[0]	Input		P11	2	LVCMOS25	2.5		12	SLOW
B[1]	Input		L3	3	LVCMOS25	2.5		12	SLOW
B[2]	Input		K3	3	LVCMOS25	2.5		12	SLOW
B[3]	Input		B4	0	LVCMOS25	2.5		12	SLOW
OPCION (4)	Input				LVCMOS25	2.5		12	SLOW
OPCION[0]	Input		G12	1	LVCMOS25	2.5		12	SLOW
OPCION[1]	Input		C11	0	LVCMOS25	2.5		12	SLOW
OPCION[2]	Input		M4	2	LVCMOS25	2.5		12	SLOW
OPCION[3]	Input		A7	0	LVCMOS25	2.5		12	SLOW
RESULTADO (4)	Output			2	LVCMOS25	2.5		12	SLOW
RESULTADO[0]	Output		M5	2	LVCMOS25	2.5		12	SLOW
RESULTADO[1]	Output		M11	2	LVCMOS25	2.5		12	SLOW
RESULTADO[2]	Output		P7	2	LVCMOS25	2.5		12	SLOW
RESULTADO[3]	Output		P6	2	LVCMOS25	2.5		12	SLOW
Scalar ports (1)									
ACARREO	Output		N5	2	LVCMOS25	2.5		12	SLOW

Figura 3.8.10 Asignación pines.

Una vez realizado las asignaciones de los pines guardamos los cambios y generamos el archivo .bit para poder bajar el diseño a nuestra tarjeta.

Probamos nuestro diseño con los mismos valores de las simulaciones, para OPCION=0000 y las entradas A = “1110”, B= “1100” las salidas deben ser RESULTADO=1010 ACARREO=1, ver Figura 3.8.11.

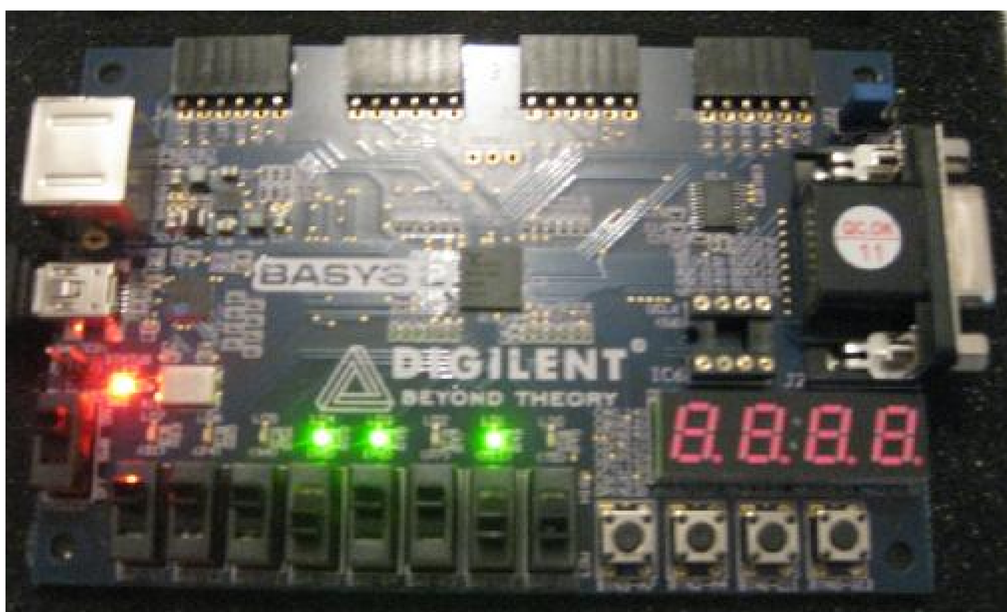


Figura 3.8.11 OPCION=0000, RESULTADO=1010, ACARREO=1.

Ahora probaremos para el caso OPCION=1000 y las entradas A = “1110” y B= “1100”, las salidas serán RESULTADO=0011 ACARREO=0, ver Figura 3.8.12.

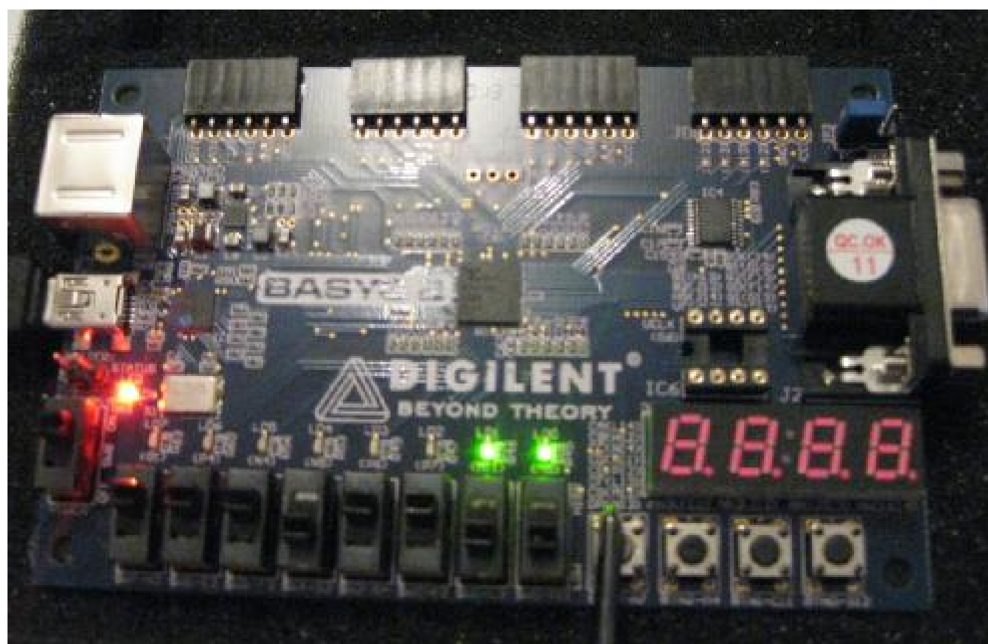


Figura 3.8.12 OPCION= 1000, RESULTADO=0011, ACARREO=0.

Por último probaremos para el caso OPCION=1100 y las entradas A = “1110” y B= “1100”, las salidas serán RESULTADO=0110 ACARREO=0, ver Figura 3.8.13.

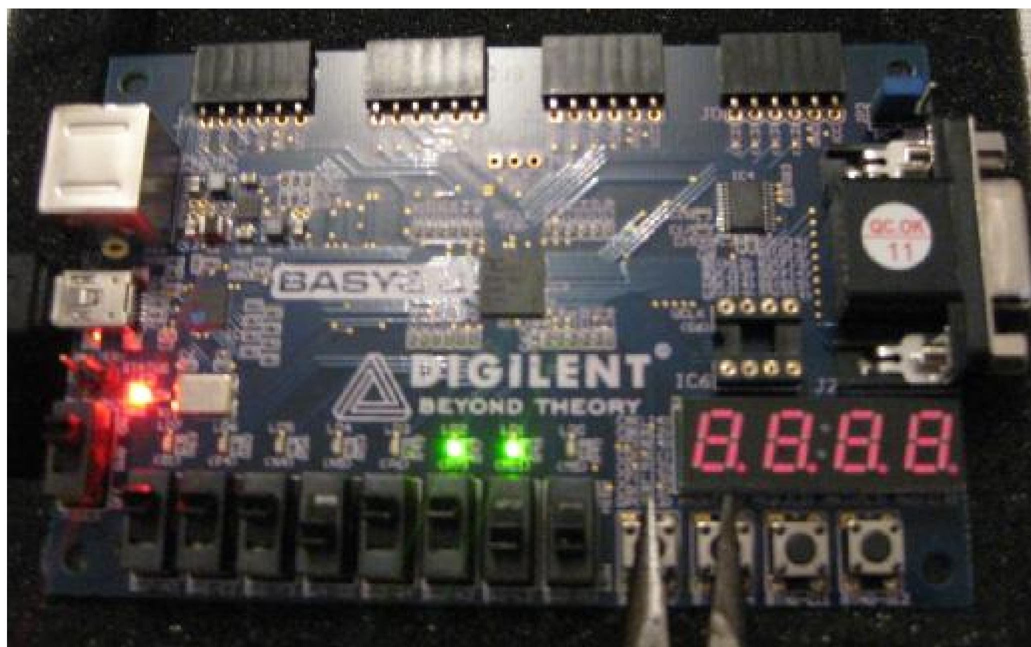


Figura 3.8.13 OPCION=1100, RESULTADO=0110, ACARREO=0.

Conclusión

Se diseña y se implementa satisfactoriamente las operaciones lógicas y aritméticas descritas en la tabla 3.8.2. Se corrobora su buen funcionamiento tanto en simulación como en la tarjeta de prueba.

Práctica 9

Modelado e implementación de Latches y FF: T, D, SR y JK

Objetivo

Conocer y diseñar las configuraciones básicas de los sistemas secuenciales como los Latches y Flip-Flop's, comprobar sus tablas de verdad así como sus formas de activación ya sea por nivel (alto o bajo) o por flanco (positivo o negativo).

Desarrollo

Un sistema secuencial está formado por un circuito combinacional y un elemento de memoria encargado de almacenar de forma temporal la historia del sistema ver Figura 3.9.1. En esencia, la salida de un sistema secuencial no sólo depende del valor presente en las entradas en un instante determinado, sino también de la historia del sistema (se dice que los secuenciales son circuitos con memoria, mientras que los combinacionales no tienen memoria).

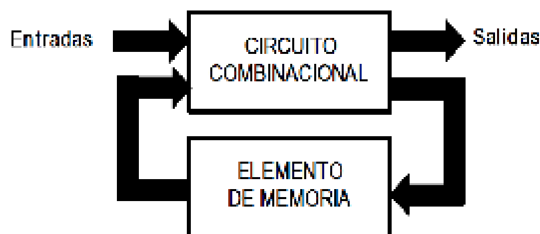


Figura 3.9.1 Sistema Secuencial

En el caso de los sistemas secuenciales, se partirá del elemento mínimo que será el biestable. Este bloque secuencial de construcción constituye el ladrillo con los que se edifican los sistemas secuenciales (equivalente a las puertas lógicas en los sistemas combinacionales).

Es posible agrupar los biestables en dos grupos:

- Asíncronos: La salida cambia de estado cuando cambian las entradas.
- Síncronos: La salida cambia de estado (en función de las entradas) de forma acompasada con una señal de reloj.

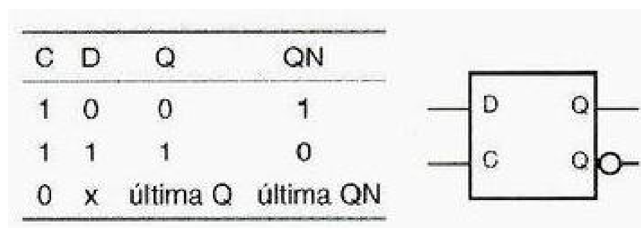
Este tipo de dispositivos síncronos a su vez se clasifican en:

- Activos por nivel (alto o bajo): La salida cambia de estado sólo cuando la señal de reloj se encuentra en el estado lógico ‘1’ (nivel alto) ó ‘0’ (nivel bajo).
- Activos por flanco (subida o bajada): La salida cambia de estado sólo cuando la señal de reloj pasa de un nivel lógico de ‘0’ a ‘1’ (flanco de subida) o de ‘1’ a ‘0’ (flanco bajada).

Esto nos lleva a la siguiente clasificación de los biestables:

- Latch: Se les llama así a los biestables asíncronos o síncrono por nivel.
- Flip-flop: Se les llama así a todos los biestables síncronos por flanco.

Latch Tipo D Síncrono habilitado por nivel alto



C	D	Q	QN
1	0	0	1
1	1	1	0
0	x	última Q	última QN


```
--Inicia Código

entity LATCHD is
   Port (C,D : in  STD_LOGIC;
      Q,Qn : inout  STD_LOGIC);
end LATCHD;

architecture Behavioral of LATCHD is

begin

PROCESS(C, D, Q, Qn)

BEGIN
   IF(C='1') THEN -- CAMBIA DE ESTADO SOLO SI 'C' ESTA EN NIVEL ALTO '1'
                  IF (D = '0') THEN
                  Q <= '0'; Qn <= '1';
                  ELSIF (D = '1') THEN
                  Q <= '1'; Qn <= '0';
                  ELSE Q <= Q; Qn <= Qn;
                  END IF;
         ELSE NULL;
   END IF;
END PROCESS;

end Behavioral;

--Fin de código
```

Ahora definiremos los valores de las entradas como se muestra en la figura 3.9.2

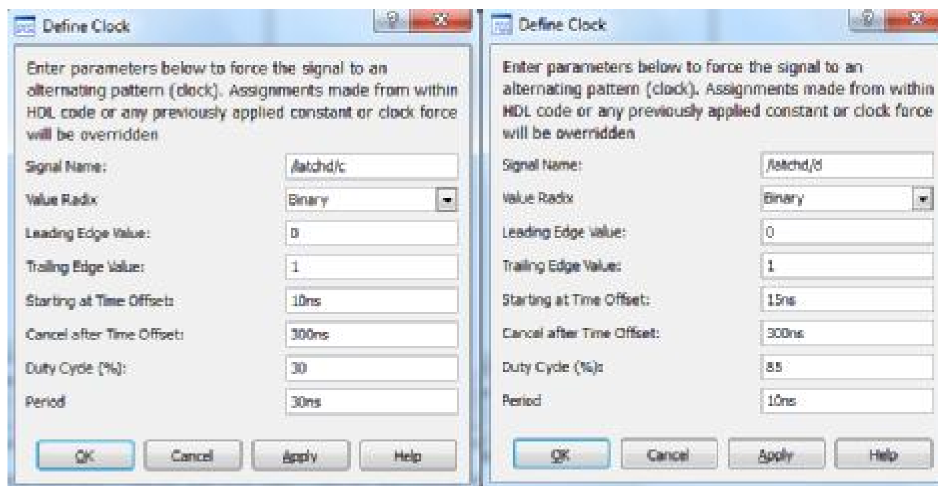


Figura 3.9.2 Asignación de valores de entrada LATCH D

Asignados estos valores de entrada obtendremos la simulación de la figura 3.9.3-A donde podemos apreciar que hasta que se inicia el pulso ‘C’ está en niel alto vemos actividad en las salidas Q y Qn, para este caso C=1, D= 0 por lo que Q=0 y Qn=1

Figura 3.9.3-A Caso C=1, D=0

En la figura 3.9.3-B podemos observar que se cumplen los valores de salida de la tabla de verdad para C=1 y D=1.

Figura 3.9.3-B Caso C = 1, D = 1

En la figura 3.9.3-C se aprecia que se cumplen los valores de salida de la tabla de verdad para C=0 y D=X.

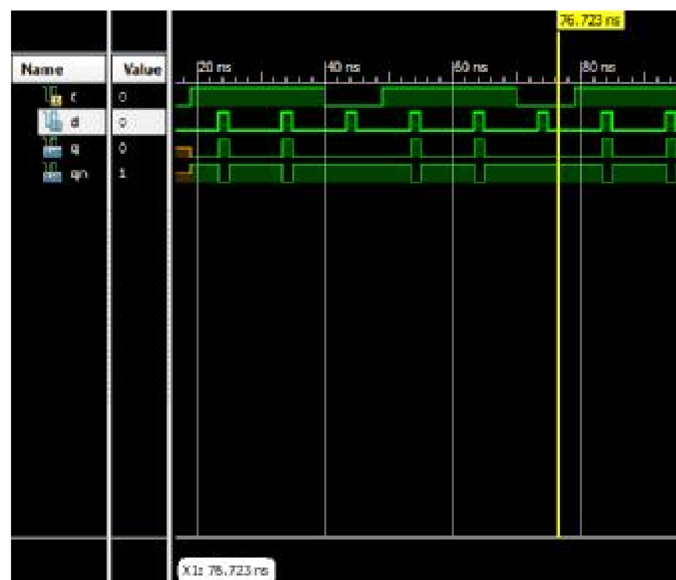


Figura 3.9.3-C Primer caso C = 0, D = X

En la figura 3.9.3-D se aprecia que se cumplen los valores de salida de la tabla de verdad para C=0 y D=X

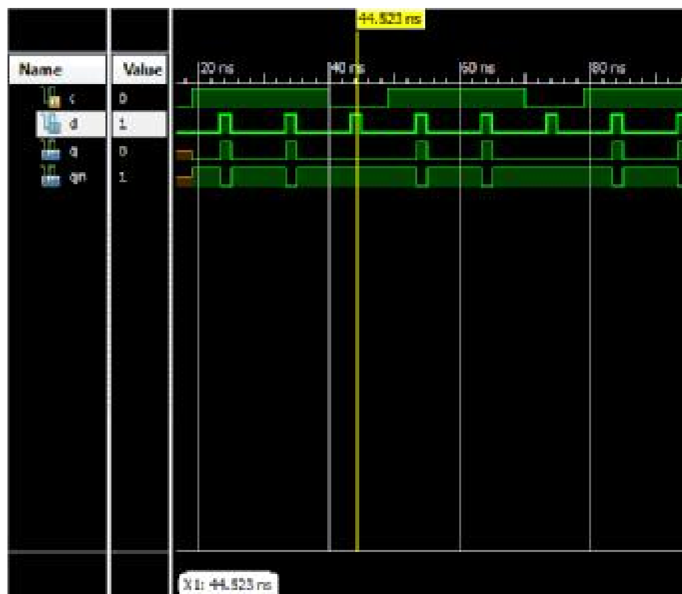


Figura 3.9.3-D Segundo caso C = 0, D = X

Latch S-R síncrono habilitado por nivel alto.

S	R	C	Q	QN
0	0	1	última Q	última QN
0	1	1	0	1
1	0	1	1	0
1	1	1	1	1
x	x	0	última Q	última QN

S
C
R
Q
Q

```
--Inicia Codigo

entity LATCHSR is
   Port (S,R,C : in  STD_LOGIC;
         Q,Qn : inout  STD_LOGIC);
end LATCHSR;

architecture Behavioral of LATCHSRH is

begin
```

```
PROCESS(S, R, C, Q, Qn)

BEGIN
        IF(C='1') THEN -- CAMBIA DE ESTADO SOLO SI 'C' ESTA EN NIVEL ALTO '1'
                IF (S = '0' and R = '0') THEN
                  Q <= Q; Qn <= Qn;
                  ELSIF (S = '0' and R = '1') THEN
                  Q <= '0'; Qn <= '1';
                  ELSIF (S = '1' and R = '0') THEN
                  Q <= '1'; Qn <= '0';
                  ELSE Q <= '-'; Qn <= '-';
                END IF;
           ELSE NULL;
        END IF;
END PROCESS;

end Behavioral;

--Termina código
```

Para la parte de la simulación asignaremos los valores de entrada como se muestra en la figura 3.9.4

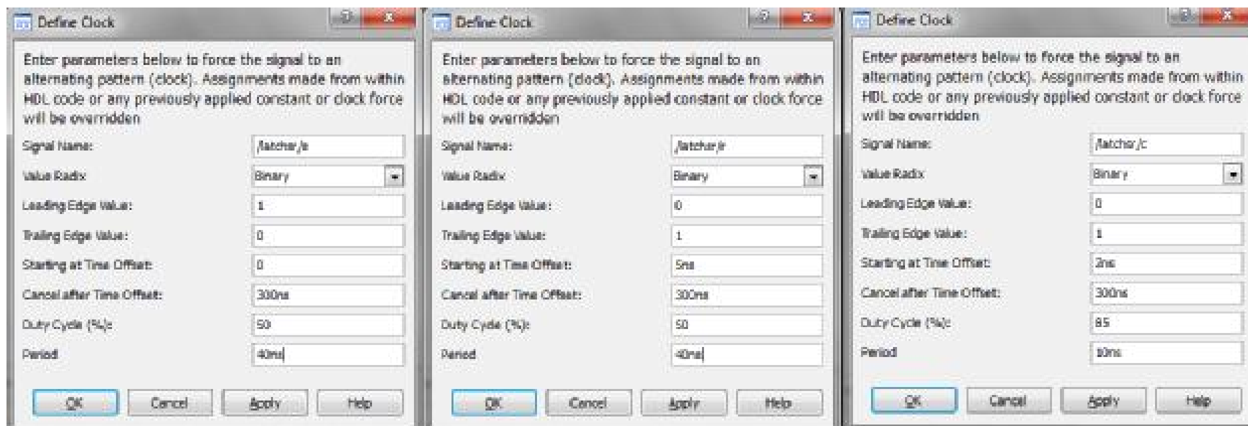


Figura 3.9.4 Asignación de valores de entrada LATCH S-R

Con la asignación de estos valores podremos apreciar que se cumple la tabla de verdad del LATCH S-R, para el Caso A (S=0 R=0 C=1) Figura 3.9.5-A, Caso B (S=0 R=1 C=1) Figura 3.9.5-B, Caso C (S=1 R=0 C=1) 3.9.5-C, Caso D(S=1 R=1 C=1) Figura 3.9.5-D y Caso E (S=X R=X C=0) Figura 3.9.5-E

Caso A (S=0 R=0 C=1)

Figura 3.9.5-A Q = ultima Q y Qn = ultima Qn

Caso B (S=0 R=1 C=1)

Figura 3.9.5-B Q = 0 y Qn = 1

Caso C (S=1 R=0 C=1)

Figura 3.9.5-C Q = 1 y Qn = 0

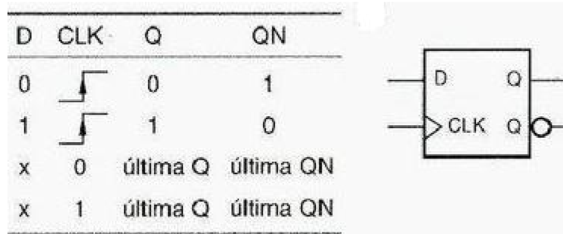
Caso D(S=1 R=1 C=1)

Figura 3.9.5-D Q = 0 y Qn = 1

Caso E (S=X R=X C=0)

Figura 3.9.5-E Q = ultima Q y Qn = ultima Qn

Flip-Flop Tipo D habilitado por flanco positivo



D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	última Q	última QN
x	1	última Q	última QN

--Inicia código

```
entity FlipFlopD is
   Port ( D,CLK : in  STD_LOGIC;
          Q,Qn : inout  STD_LOGIC);
end FlipFlopD;

architecture Behavioral of FlipFlopD is
begin


PROCESS (CLK)
   BEGIN
     IF (CLK'event and CLK = '1') THEN
       Q <= D; Qn <= not D;
     END IF;
END PROCESS;
```

end Behavioral;

--Fin de código

Para la simulación asignaremos los valores de entrada que se muestran en la figura 3.9.6.

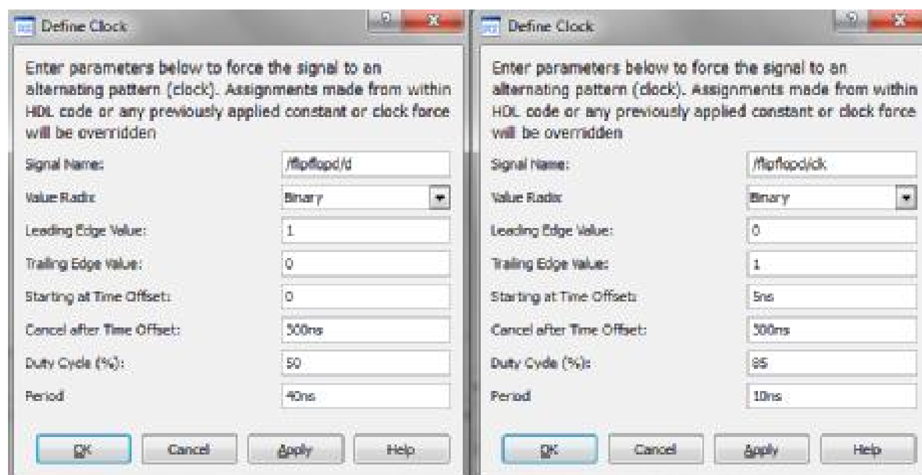


Figura 3.9.6 Asignación de valores de entrada Flip Flop tipo D

Con los valores asignados podremos a preciar en la simulación que se cumple la tabla de verdad del Flip Flop tipo D, Caso A (D=0 CLK= ↑) Figura 3.9.7-A, Caso B (D=1 CLK=↓) Figura 3.9.7-B y Caso C (D= X CLK = 0 ó 1) Figura 3.9.7-C.

Caso A (D=0 CLK= ↑)

Figura 3.9.7-A Q = 0 y Qn = 1

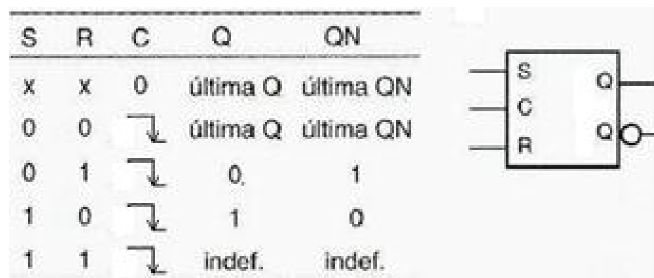
Caso B (D=1 CLK=↑)

Figura 3.9.7-B Q = 1 y Qn = 0

Caso C (D= X CLK = 0 ó 1)

Figura 3.9.7-C Q = ultima Q y Qn = ultima Qn

Flip Flop RS habilitado por flanco negativo



S	R	C	Q	QN
x	x	0	última Q	última QN
0	0	↓	última Q	última QN
0	1	↓	0	1
1	0	↓	1	0
1	1	↓	indef.	indef.


```
--Inicia código

entity FlipFlopSR is
Port ( S,R,CLK : in  STD_LOGIC;
Q,Qn : inout  STD_LOGIC);
end FlipFlopSR;

architecture Behavioral of FlipFlopSR is

begin

PROCESS (CLK) BEGIN

  IF (CLK'event and CLK = '0') THEN – Indica que cambiara de estado con el flanco negativo
     IF (S = '0' and R = '0') THEN
      Q <= Q; Qn <= Qn;
      ELSIF (S = '0' and R = '1') THEN
      Q <= '0'; Qn <= '1';
      ELSIF (S = '1' and R = '0') THEN
      Q <= '1'; Qn <= '0';
      ELSE Q <= '-'; Qn <= '-';  --- Para el caso no valido se asignamos '-' a Q y Qn
     END IF;
      ELSE NULL;
 END IF;

END PROCESS;

end Behavioral;

--Fin de código
```

Para realizar la simulación asignaremos los valores de entrada como se muestra en la figura 3.9.8

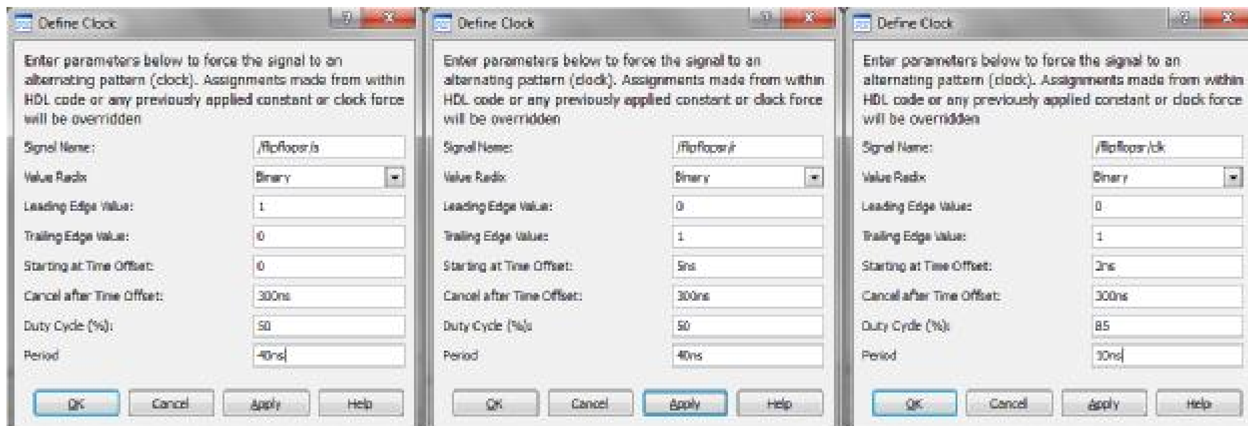


Figura 3.9.8 Asignación de valores de entrada Flip Flop tipo RS

La simulación derivada de estos datos, nos permite comprobar la tabla de verdad del Flip Flop tipo RS, que podemos observar en las figuras de los siguientes casos: Caso A (S=0 R=0 CLK=X) figura 3.9.9-A, Caso B (S=0 R=1 CLK=↓) figura 3.9.9-B, Caso C (S=1 R=0 CLK=↓) figura 3.9.9-C, Caso D (S=1 R=1 CLK= ↓) figura 3.9.9-D.

Caso A (S=0 R=0 CLK=X)

Figura 3.9.9-A Q = ultima Q y Qn = ultima Qn

Caso B (S=0 R=1 CLK= ↓)

Figura 3.9.9-B Q = 0 Q y Qn = 1

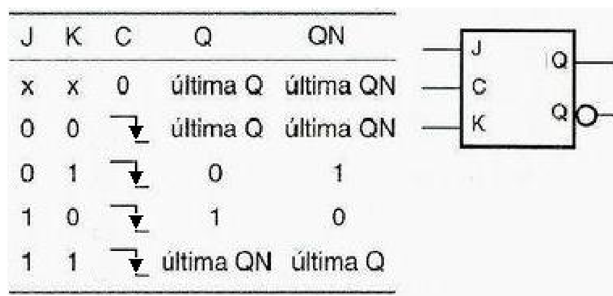
Caso C (S=1 R=0 CLK=↓)

Figura 3.9.9-C Q = 1 Q y Qn = 0

Caso D (S=1 R=1 CLK= ↓)

Figura 3.9.9-C Caso no valido

Flip Flop tipo JK habilitado por flanco negativo



J	K	C	Q	QN
x	x	0	última Q	última QN
0	0	↓	última Q	última QN
0	1	↓	0	1
1	0	↓	1	0
1	1	↓	última QN	última Q


```
--Inicia código

entity FlipFlopJK is
Port ( J,K,CLK : in  STD_LOGIC;
Q,Qn : inout  STD_LOGIC);
end FlipFlopJK;

architecture Behavioral of FlipFlopJK is

begin

   PROCESS (CLK) BEGIN

     IF (CLK'event and CLK = '0') THEN -- Cambia de estado cuando el flanco es negativo
     IF (J = '0' and K = '0') THEN
     Q <= Q; Qn <= Qn;
```

```
      ELSIF (J = '0' and K = '1') THEN
      Q <= '0'; Qn <= '1';
      ELSIF (J = '1' and K = '0') THEN
      Q <= '1'; Qn <= '0';
      ELSE Q <= Qn; Qn <= Q; --Estado de basculación
      END IF;
      ELSE NULL;
   END IF;

END PROCESS;

end Behavioral;

--Fin de código
```

Para obtener las simulaciones y poder comprobar la tabla de verdad del Flip Flop JK , asignaremos los valores de entrada de la figura 3.9.10.

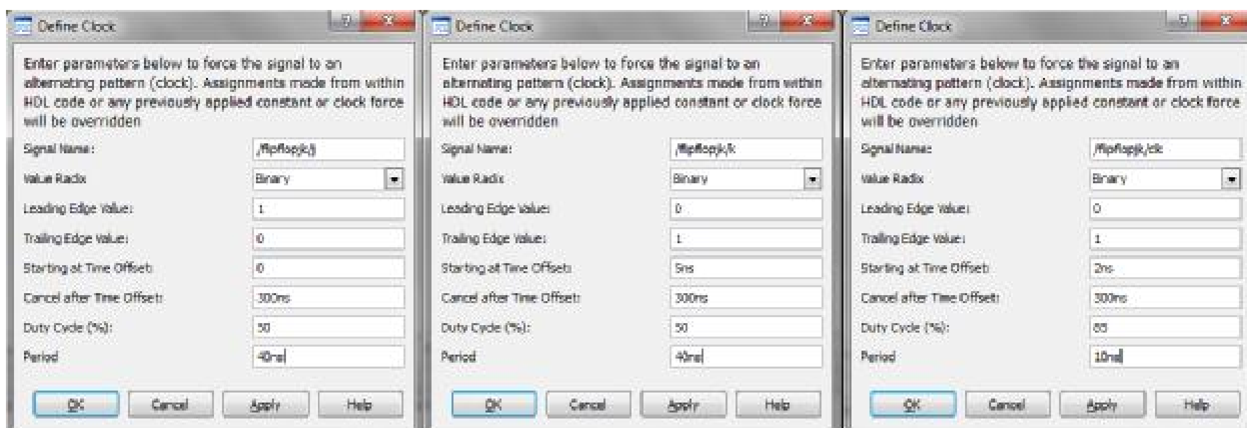


Figura 3.9.10 Asignación de valores de entrada Flip Flop tipo JK

En la simulación obtenida de estos datos de entrada podremos apreciar que se cumple la tabla de verdad del Flip Flop JK como se muestra en las figuras de los siguientes casos: Caso A (J=0 K=0 CLK= ↓) figura 3.9.10-A, Caso B (J=0 K=1 CLK=↓) figura 3.9.10-B, Caso C (J=1 K=0 CLK=↓) figura 3.9.10-C y Caso D (J=1 K=1 CLK= ↓) figura 3.9.10-D.

Caso A (J=0 K=0 CLK= ↓)

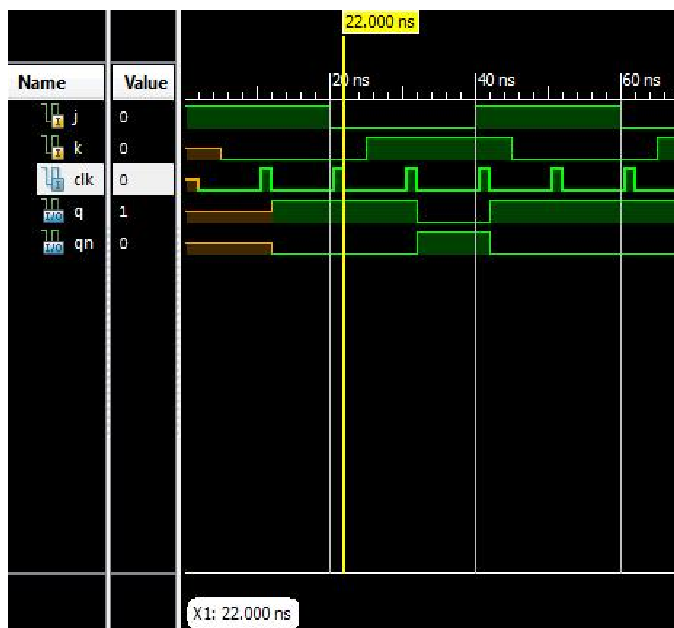


Figura 3.9.10-A Q = ultima Q y Qn = ultima Qn

Caso B (J=0 K=1 CLK=↓)

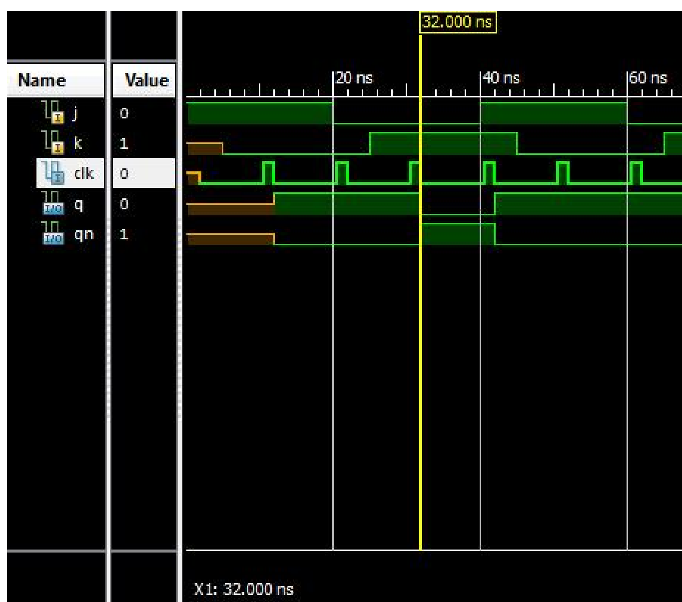


Figura 3.9.10-B Q = 0 y Qn = 1

Caso C (J=1 K=0 CLK=↓)

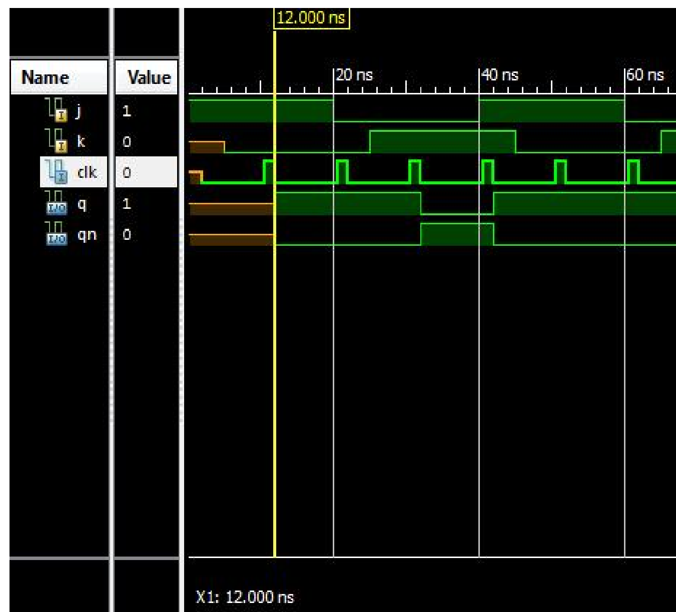


Figura 3.9.10-C Q = 1 y Qn = 0

Caso D (J=1 K=1 CLK= ↓)

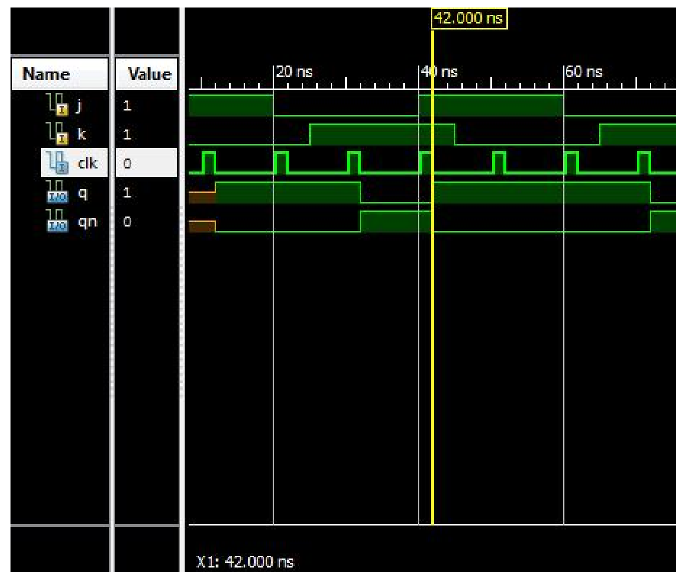
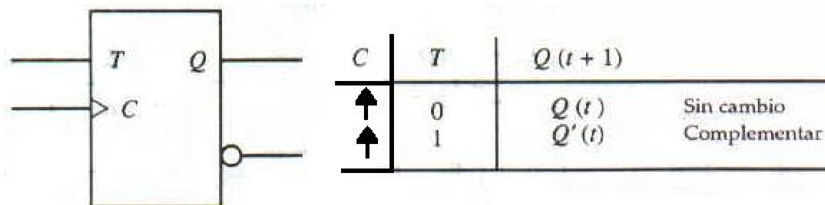


Figura 3.9.10-D Q = Qn y Qn = Q

Flip Flop Tipo T habilitado por flanco positivo



El flip flop T basa toda su operatividad en el valor lógico de su única entrada de control. Si en 'T' tenemos el nivel lógico uno, la salida 'Q' basculará, cambiando constantemente su valor según le vayan llegando los flancos de subida del reloj. Si la entrada 'T' está a nivel lógico cero, el flip flop mantiene su valor anterior a modo de una memoria.

```
--Inicia código

entity FlipFlopT is
Port ( T,CLK : in  STD_LOGIC;
Q,Qn : inout  STD_LOGIC);
end FlipFlopT;

architecture Behavioral of FlipFlopT is

begin

PROCESS (CLK) BEGIN

IF (CLK'event and CLK = '1') THEN -- Cambia de estado cuando el flanco es positivo
IF (T = '0') THEN
Q <= T; Qn <= NOT (T);
ELSIF(T= '1') THEN
Q <= NOT(Q); Qn <= Q;
END IF;
END IF;

END PROCESS;
end Behavioral;

--Fin de código
```

Para la simulación asignaremos los valores de entrada como se muestra en la Figura 3.9.11.

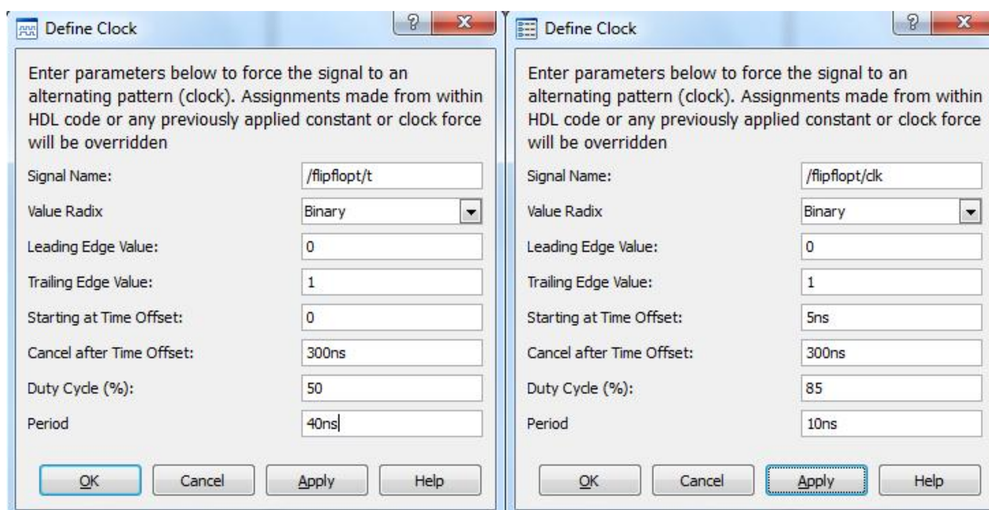


Figura 3.9.11 Asignación de valores de entrada Flip Flop tipo T

En las simulaciones obtenidas podremos apreciar que se cumple la tabla de verdad como se muestra en las figuras del Caso A (C=↑ T=0) figura 3.9.12-A y Caso B (CLK=↑ T=1) figura 3.9.12-B.

Caso A (CLK=↑ T=0)

Figura 3.9.12-A Q = se mantiene la última Q y Qn = complemento de la última Q

Caso B (C=↑ T=1)

Figura 3.9.12-B Q = complemento de la última Q y Qn = complemento de la última Qn

Conclusión

Se diseñó e implementaron los diseños básicos de Latches y Flip Flop's, así como las formas de habilitarlos por flanco positivo, negativo, nivel alto o bajo. Se comprobaron las tablas de verdad para todos los Flip Flop's y Latches a través del análisis de las simulaciones obtenidas.

También podemos concluir lo siguiente:

- **Latch:** Se les llama así a los biestables asíncronos o síncrono por nivel.
- **Flip-flop:** Se les llama así a todos los biestables síncronos por flanco.

Nota: Para el manejo de estos diseños en el ambiente se Xilinx ISE solo se recomienda hacer uso de los Flip Flop's, ya que los Latches se vuelven inestables.

Práctica 10

Modelado e implementación multiplexor de display

Objetivo

Se creara un controlador de display que permita multiplexar los 4 display de 7 segmentos que tiene la tarjeta BASYS 2. El controlador podrá ser capaz de desplegar una cifra de 4 dígitos.

Desarrollo

Nuestra tarjeta BASYS 2 tiene 4 display ánodo común. Los segmentos de A, B, C, D, E, F, G y J están conectados en bus a los 4 display, solo el enable es único para cada display. Entonces si habilitamos los 4 display se mostrara el mismo digito en todos los displays (ver figura 3.10.1). Para poder desplegar diferentes dígitos tendremos que multiplexar el enable a una frecuencia en la cual el ojo humano no detecte el encendido y apagado de los displays.

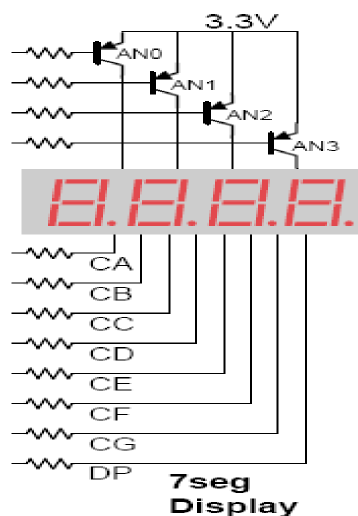


Figura 3.10.1 Displays en tarjeta Basys2

Nuestro controlador tendrá 2 funciones, la primera se encargará de decodificar los datos de binario al display de 7 segmentos mientras que la segunda multiplexara esta salida a los 4 displays de la tarjeta. Para realizar estas funciones necesitaremos los siguientes circuitos:

-CONTADOR: Genera la frecuencia del multiplexor

-MUXDISPLAY: Selecciona display y codifica salida a 7 segmentos.

El circuito de nuestro proyecto se muestra en la figura 3.10.2

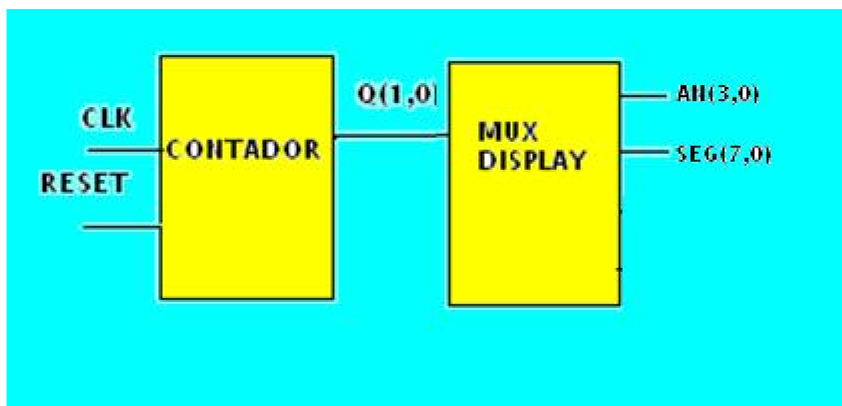


Figura 3.10.2 Circuito proyecto

Los dos circuitos los implementaremos con 2 procesos, uno que dependa de las señale CLK y otro de la señal sel.

Ahora iniciaremos la práctica creando un proyecto llamado PRACTICA10, el cual tendrá un sola fuente llamada P10, las entradas y salidas se muestran en la figura 3.10.3.

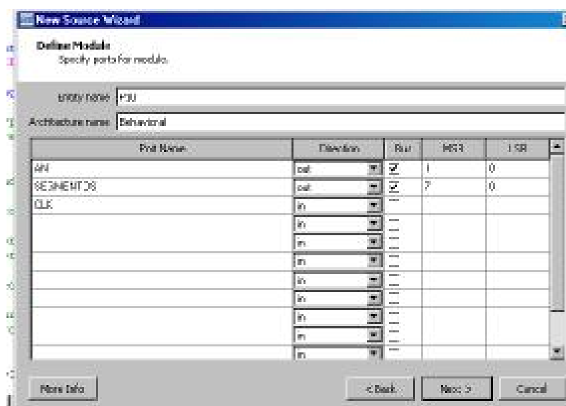


Figura 3.10.3 I/O P10

Para el contador hay que tomar en cuenta que el reloj por default de la tarjeta BASYS 2 es de 50Mhz, para nuestro diseño tendremos que modificar esta frecuencia con una señal divisora. Nuestro contador será de dos bits y cambiara su resultado con una frecuencia de 100 Hz, ver figura 3.10.4.

RELOJ(100 Hz)	sel(1,0)
INICIALMENTE	00
1	01
2	10
3	11
NUEVO CICLO	00

Figura 3.10.4 Tabla de verdad CONTADOR

Para el reloj con frecuencia de 100Hz debemos escalar la señal de reloj de 50Mhz:
50 Mhz / 500 000 = 100 Hz
500 000 =1111010000100100000

El código VHDL del contador es el siguiente:

```
SIGNAL sel: STD_LOGIC_VECTOR (1 downto 0):="00"; --contador (1,0)
SIGNAL temp1: STD_LOGIC_VECTOR (18 downto 0):="0000000000000000000"; --señal divisora
contador: process (CLK)
begin
if CLK='1' and CLK'event then
   temp1 <= temp1 + 1;
      if temp1 = "1111010000100100000" then

                              if sel < "11" then
                                          sel <= sel +1;
                                        else
              sel <= "00";
            end if;
        temp1 <="0000000000000000000";
      end if;
end if;
end process;
```

Para nuestro codificador de 7 segmentos y multiplexor de ánodos tendremos una entrada sel(1,0), la cual definirá ciertos digitos en nuestro display para cierto display encendido. En este ejemplo se desplegara la cifra 4312, los segmentos de nuestro display se muestran en la figura 3.10.5.

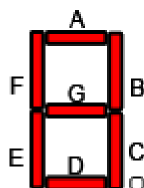


Figura 3.10.5 Segmentos en Display.

Recordemos que los display son ánodo común entonces para que estén encendidos necesitan tener un valor de 0. La tabla de verdad de nuestro circuito MUXDISPLAY está dada en la figura 3.10.6

sel(1,0)	SEGMENTOS	AN
00	10100100	1110
01	11111001	1101
10	10110000	1011
11	10011001	0111

Figura 3.10.6 Tabla de MUXDISPLAY

El código VHDL de MUXDISPLAY es:

```
MUXDISPLAY: process(sel)
begin
```

```
case sel is
        when "00" => SEGMENTOS <= "10100100"; AN <= "1110";
        when "01" => SEGMENTOS <= "11111001"; AN <= "1101";
        when "10" => SEGMENTOS <= "10110000"; AN <= "1011";
        when "11" => SEGMENTOS <= "10011001"; AN <= "0111";
when others =>
null;
end case;
end process;
```

Ahora colocaremos estos dos códigos y verificaremos si hay errores de código o diseño, ver figura 3.10.7.

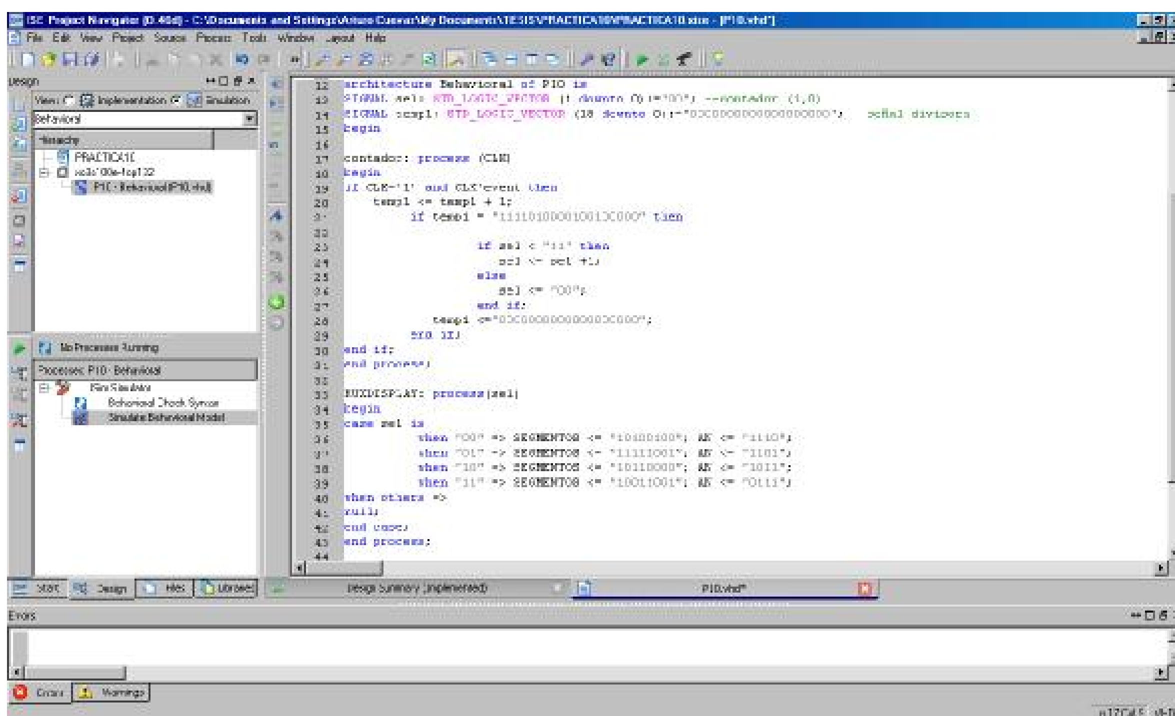


Figura 3.10.7 Verificación Código VHDL P10

Es importante agregar las librerías:
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Si no estuvieran *tendríamos* errores para el contador.

Ahora asignaremos pines con PlanAhead (ver figura 3.10.8).

I/O Ports

Name	Dir	Neg Diff Pair	Site	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type
All ports (13)										
AN (4)	Output			1	LVCMOS25	2.5		12	SLOW	
AN[0]	Output		F12	1	LVCMOS25	2.5		12	SLOW	
AN[1]	Output		J12	1	LVCMOS25	2.5		12	SLOW	
AN[2]	Output		M13	1	LVCMOS25	2.5		12	SLOW	
AN[3]	Output		K14	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS (8)	Output				LVCMOS25	2.5		12	SLOW	
SEGMENTOS[0]	Output		L14	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[1]	Output		H12	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[2]	Output		N14	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[3]	Output		N11	2	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[4]	Output		P12	2	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[5]	Output		L13	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[6]	Output		M12	1	LVCMOS25	2.5		12	SLOW	
SEGMENTOS[7]	Output		N13	1	LVCMOS25	2.5		12	SLOW	
Scalar ports (1)										
CLK	Input		B8	0	LVCMOS25	2.5		12	SLOW	

Figura 3.10.8 Asignación de pines

Para simular nuestro proyecto en ISim modificaremos el valor de la señal temp1 la cual tendrá un valor máximo de 01. El código para el reloj quedara de la siguiente forma:

```
SIGNAL sel: STD_LOGIC_VECTOR (1 downto 0):="00"; --contador (1,0)

SIGNAL temp1: STD_LOGIC_VECTOR (1downto 0):="00"; --señal divisora

begin


reloj: process (CLK)

begin

if CLK='1' and CLK'event then

   temp1 <= temp1 + 1;

      if temp1 = "01" then

            if sel < "11" then

                    sel <= sel +1;

          else

              sel <= "00";

            end if;

        temp1 <="00";

      end if;
```

end if;

end process;

Para la simulación solo forzaremos la señal de reloj con un periodo de un 1ms (ver figura 3.10.9).

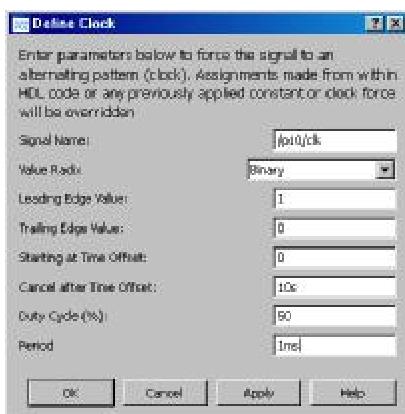


Figura 3.10.9 Reloj CLK Simulación

Después de haber corrido la simulación por 10s tendremos el siguiente resultado, figura 3.10.10.

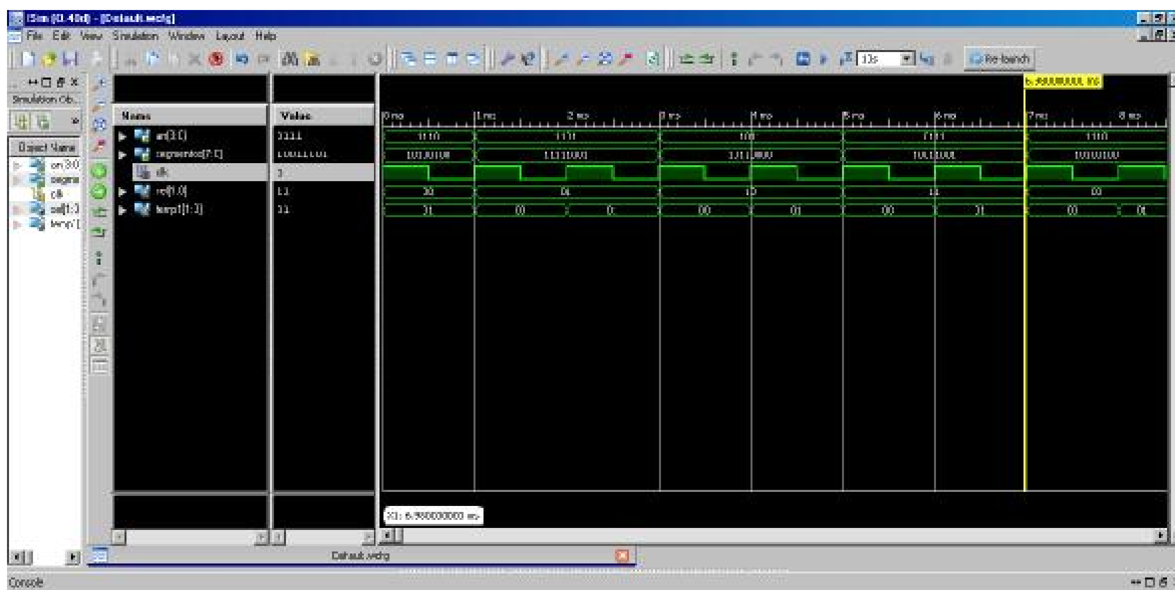


Figura 3.10.10 Simulación

Como se puede apreciar tenemos lo siguientes valores:

Para an=1110 , segmentos= 10100100

Para an=1101, segmentos= 11111001

Para an=1011, segmentos= 10110000

Para an=0111, segmentos= 10011001

Con lo cual afirma que se desplegara los digitos 4,3,1,2 en nuestra tarjeta basys2.

Despues de haber obtenido una simulación exitosa regresaremos al valor original de temp1.

```
SIGNAL sel: STD_LOGIC_VECTOR (1 downto 0):="00"; --contador (1,0)
SIGNAL temp1: STD_LOGIC_VECTOR (18 downto 0):="0000000000000000000"; --señal divisora
contador: process (CLK)
begin
if CLK='1' and CLK'event then
   temp1 <= temp1 + 1;
      if temp1 = "1111010000100100000" then

                                      if sel < "11" then
                                                   sel <= sel +1;
                                                 else
                  sel <= "00";
               end if;
          temp1 <="0000000000000000000";
        end if;
end if;
end process;
```

Verificaremos la sintesis y diseño de nuestro proyecto con la herramienta Top module, figura 3.10.11.

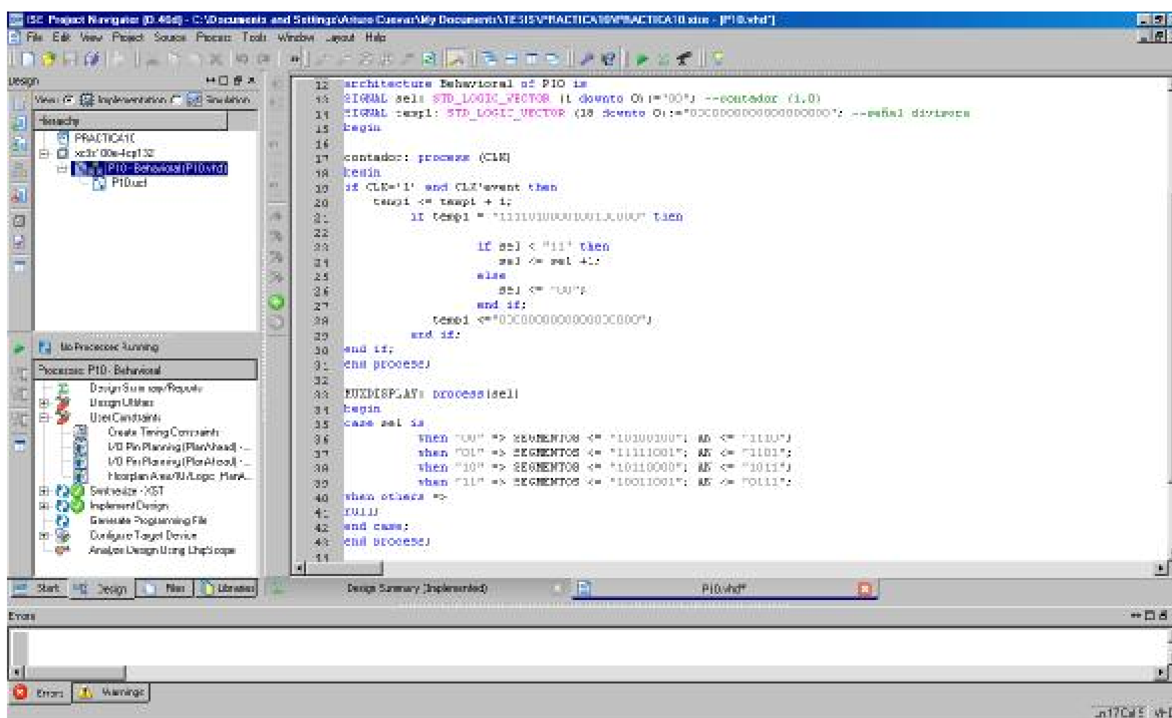


Figura 3.10.11 Verificación de la Sintesis y diseño del proyecto

Por último generamos el programa P10.bit y lo bajaremos a nuestra tarjeta (ver figura 3.10.12).

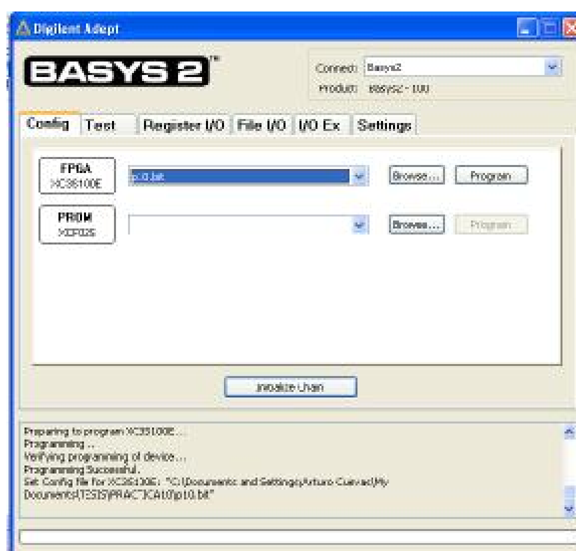


Figura 3.10.12 Programa P10.bit

Por último verificamos resultados (ver figura 3.10.13).

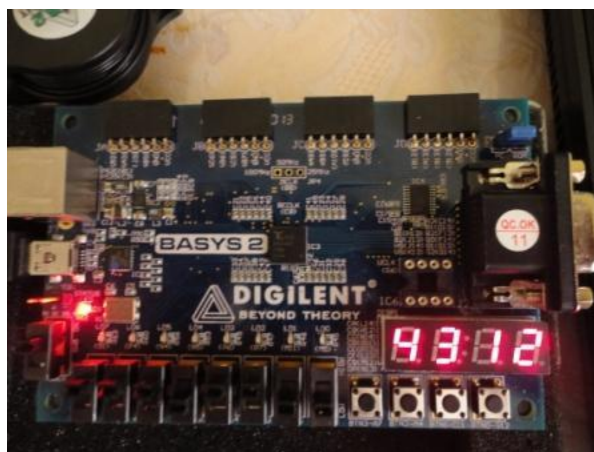


Figura 3.10.13 Resultado en tarjeta.

Conclusiones.

Se creó un proyecto que permitiera multiplexar displays de 7 segmentos, este principio es el utilizan los monitores y televisores. El principio se basa en multiplexar una salida tan rápido que el ojo humano no pueda detectar los cambios. El código es muy sencillo se basó en 2 procesos; uno que da la frecuencia de multiplexado y el otro que selecciona las salidas (multiplexor).

Práctica 11

Modelado e implementación de un contador decimal 0000-9999

Objetivo

Diseñar un contador decimal ascendente de 0000 a 9999.

Desarrollo

Los contadores son una parte fundamental de la electrónica digital, ya que la mayoría de los procesos están sincronizados por ellos. En esta práctica se desarrollará un contador decimal de cuatro dígitos. Cada dígito estará gobernado, sincronizado, por un contador y cada contador tendrá diferente frecuencia (ver figura 3.11.1).

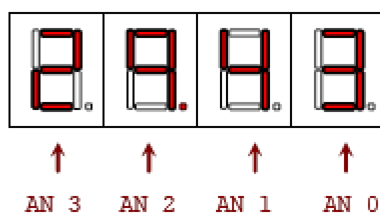


Figura 3.11.1 Dígitos Contador

Entonces se diseñarán 4 contadores a diferente frecuencia, las frecuencias se obtendrán de un reloj maestro de 50Mhz (reloj por default de la tarjeta BASYS 2). Por otro lado se necesitará otro reloj para seleccionar los displays de 7 segmentos, como se hizo en la practica 11. También se necesitara un módulo que sea el multiplexor de los ánodos y uno que sea un codificador BCD a 7 segmentos, el diagrama de nuestro contador se muestra en la figura 3.11.2.

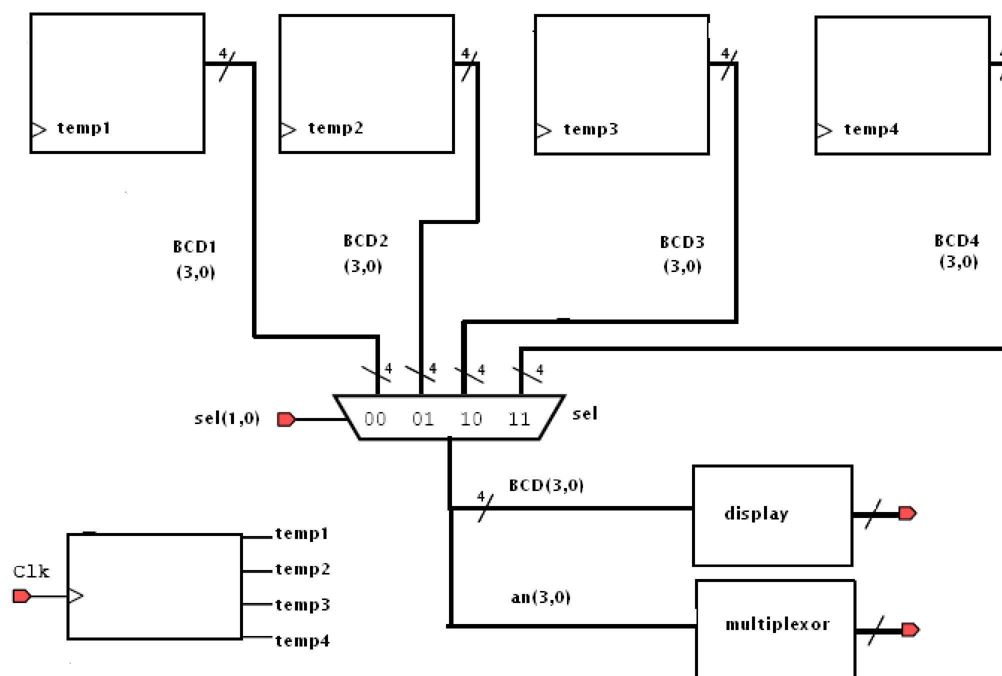


Figura 3.11.2 Diagrama Contador

El tema principal de la práctica es diseñar los relojes, todos deben estar sincronizados.

Para nuestro reloj selector de display, se necesitara una frecuencia de 60 Hz, la cual la obtenemos con una señal divisora (a partir de nuestro reloj de 50 Mhz):

50 Mhz / 50 M = 1 Hz
50 Mhz / 60 Hz = 833333.333
833333.333 = 11001011011100110101 -> señal divisora

De la misma forma obtendremos señales divisoras para los dígitos de nuestro contador, en la figura 3.11.3, se muestran estas frecuencias.

DIGITO	FRECUENCIA	SEÑAL DIVISORA (50Mhz /FRECUENCIA)	SEÑAL DIVISORA BINARIA
UNIDAD	3 Hz	16666666	111111100101000000101010
DECENA	0.3 Hz	166666660	100111101111100100000110100100
CENTENA	0.03 Hz	1666666600	1111100001011010010010011001101000
MILLAR	0.003 Hz	16666666000	10011011001110000110111000000000010000

Figura 3.11.3 Frecuencias de los Dígitos

Se puede observar que la frecuencia de los dígitos es más lenta y disminuye de acuerdo a su posición DECENA = UNIDAD*10, CENTENA =UNIDAD*100 y MILLAR = UNIDAD*1000. La frecuencia de 3Hz se eligió para apreciar los cambios en el contador.

El código VHDL de nuestros digitos quedará representador por los siguientes procesos:

```
relomux: process (CLK_50Mhz)
begin
if CLK_50Mhz='1' and CLK_50Mhz'event then
    temp <= temp + 1;
        if temp = "11001011011100110101" then
                                        if sel < "11" then
                                                    sel <= sel +1;
                                                  else
                sel <= "00";
              end if;
          temp <="00000000000000000000";
        end if;
end if;
end process;


relojunidad: process (CLK_50Mhz)
begin
if CLK_50Mhz='1' and CLK_50Mhz'event then
    temp1 <= temp1 + 1;
        if temp1 = "111111100101000000101010" then

                                        if bcd1 < "1001" then
                                                    bcd1 <= bcd1 +1;
                                                  else
                bcd1 <= "0000";
              end if;
          temp1 <="000000000000000000000000";
        end if;
end if;
end process;

relojdecena: process (CLK_50Mhz)
begin
if CLK_50Mhz='1' and CLK_50Mhz'event then
    temp2 <= temp2 + 1;
        if temp2 = "1001111011110010000110100100" then

                                        if bcd2 < "1001" then
                                                    bcd2 <= bcd2 +1;
                                                  else
                bcd2 <= "0000";
              end if;
          temp2 <="0000000000000000000000000000";
        end if;
end if;
end process;
relojcentena: process (CLK_50Mhz)
begin
if CLK_50Mhz='1' and CLK_50Mhz'event then
```

```
    temp3 <= temp3 + 1;
        if temp3 = "11000110101011101010000001101000" then
                                    if  bcd3 < "1001" then
                                              bcd3 <= bcd3 +1;
               else
                                                            bcd3 <= "0000";
                                                         end if;
                              temp3 <="00000000000000000000000000000000";
        end if;
end if;
end process;

relojmillar: process (CLK_50Mhz)
begin
if CLK_50Mhz='1' and CLK_50Mhz'event then
    temp4 <= temp4 + 1;
        if temp4 = "11111000010110100100100100000010000" then
                                    if  bcd4 < "1001" then
                                              bcd4 <= bcd4 + 1;
               else
                  bcd4 <="0000";
               end if;
                                        temp4 <="00000000000000000000000000000000000";
        end if;
end if;
end process;
```

Donde temp1, temp2, temp3 y temp4 son nuestras señales divisoras y bcd1, bcd2, bcd3 y bcd4 son nuestros dígitos.

Para nuestro multiplexor tenemos el siguiente código (multiplexor):

```
Multiplexor: process (sel,bcd1,bcd2,bcd3,bcd4)
begin
case sel is
when "00" => bcd <= bcd1; an<= "1110";
when "01" => bcd <= bcd2; an<= "1101";
when "10" => bcd <= bcd3; an<= "1011";
when "11" => bcd <= bcd4; an<= "0111";
when others => bcd <= "0000";
end case;
```

Por ultimo solo nos queda codificar valores BCD a 7 segmentos (decodificador):

```
with bcd select
display <= "11111001" when "0001", --1
"10100100" when "0010", --2
"10110000" when "0011", --3
"10011001" when "0100", --4
"10010010" when "0101", --5
"10000010" when "0110", --6
"11111000" when "0111", --7
"10000000" when "1000", --8
"10010000" when "1001", --9
```

"11000000" when others; --0

Ahora abriremos un nuevo proyecto en ISE, PRACTICA11, la cual tendrá un archivo fuente P11 con las siguientes entradas y salidas, figura 3.11.4.

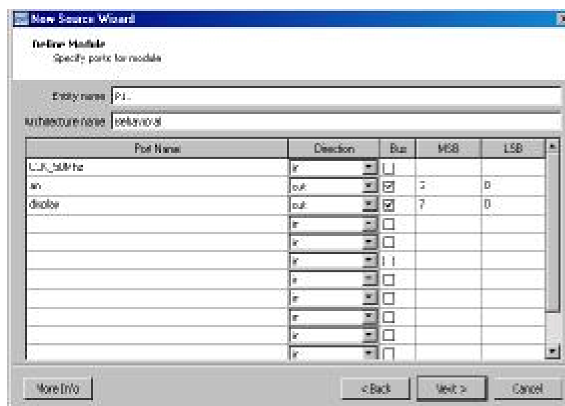


Figura 3.11.4 Definición de la fuente P11

Ahora copiamos los códigos VHDL de nuestros módulos, recordando que hay que declarar e inicializar las señales, figura 3.11.5.

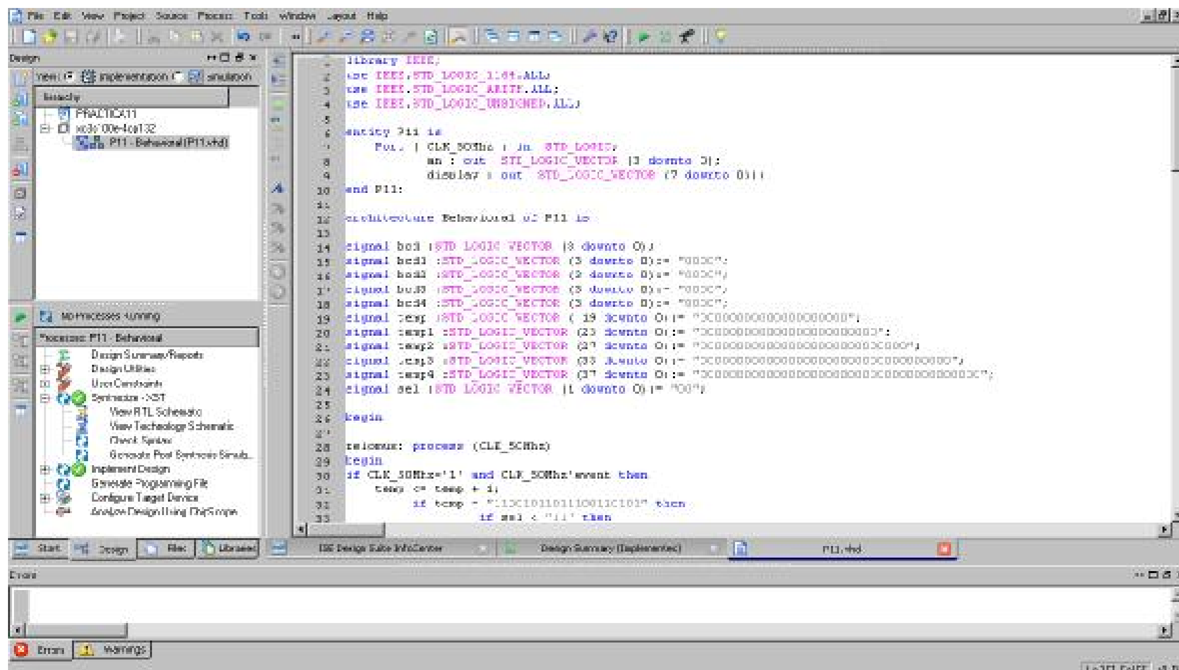


Figura 3.11.5 Código VHDL en ISE

Para simular nuestro proyecto en ISim tendremos que ajustar los contadores, de lo contrario la simulación sería demasiada larga. Cambiaremos los datos de acuerdo a la figura 3.11.6.

TAMAÑO VARIABLE	VALOR MAXIMO	VALOR DECIMAL
temp(1,0)	01	1
temp1(1,0)	01	1
temp2(3,0)	1001	19
temp3(8,0)	11000111	199
temp4(11,0)	11111001111	1999

Figura 3.11.6 Valores para simulación

Para la simulación solo forzaremos la señal de reloj con un periodo de un 1ms, figura 3.11.7.

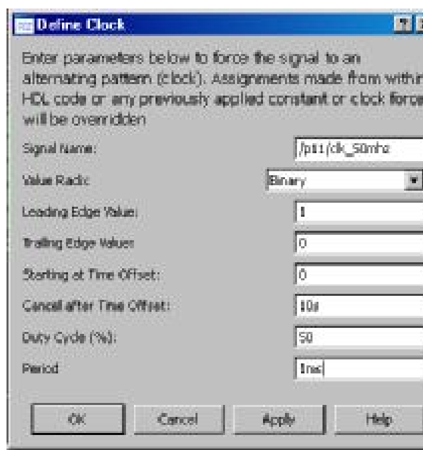


Figura 3.11.7 Reloj Clk_50mhz Simulación

También obligaremos a las señales BCD1, BCD2, BCD3 Y BCD4 que se desplieguen en decimal (ver figura 3.11.8).

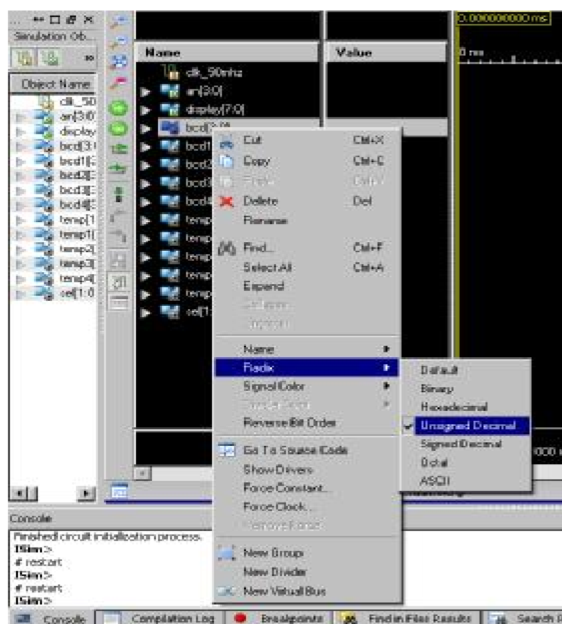


Figura 3.11.8 Señales en decimal

Después de correr la simulación debemos apreciar las señales BCD1, BCD2, BCD3 Y BCD4 en el tiempo 5.990 ms. Los dígitos unidad, decena y centena están sincronizados 0 mientras que el millar esta en 3, figura 3.11.9.

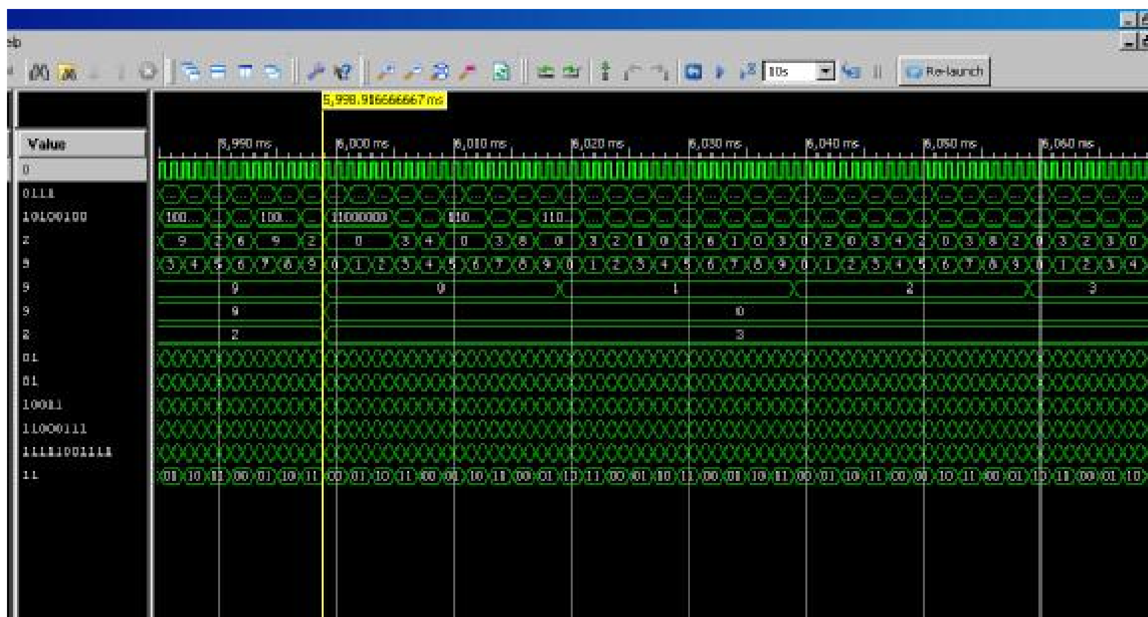
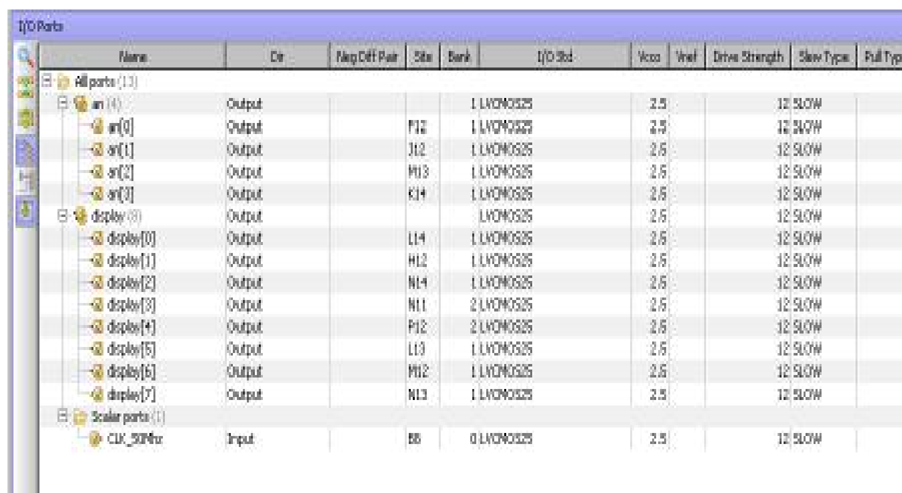


Figura 3.11.9 Reloj Simulación

Para generar el programa bit, primero haremos la asignación de pines con PlanAhed (ver figura 3.11.10).



I/O Ports

Name	Dir	Neg Diff Pair	Site	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type
All ports (13)										
an (4)	Output			1	LVCMOS25	2.5		12	SLOW	
an[0]	Output		F12	1	LVCMOS25	2.5		12	SLOW	
an[1]	Output		J12	1	LVCMOS25	2.5		12	SLOW	
an[2]	Output		M13	1	LVCMOS25	2.5		12	SLOW	
an[3]	Output		K14	1	LVCMOS25	2.5		12	SLOW	
display (8)	Output				LVCMOS25	2.5		12	SLOW	
display[0]	Output		L14	1	LVCMOS25	2.5		12	SLOW	
display[1]	Output		H12	1	LVCMOS25	2.5		12	SLOW	
display[2]	Output		N14	1	LVCMOS25	2.5		12	SLOW	
display[3]	Output		N11	2	LVCMOS25	2.5		12	SLOW	
display[4]	Output		P12	2	LVCMOS25	2.5		12	SLOW	
display[5]	Output		L13	1	LVCMOS25	2.5		12	SLOW	
display[6]	Output		M12	1	LVCMOS25	2.5		12	SLOW	
display[7]	Output		N13	1	LVCMOS25	2.5		12	SLOW	
Scalar ports (1)										
CLK_50Mhz	Input		B8	0	LVCMOS25	2.5		12	SLOW	

Figura 3.11.10 Asigandón de pines I/O PlanAhead

Antes de generar el programa bit, debemos regresar a los valores originales y correr la herramienta TopModule para verificar código y diseño (ver figura 3.11.11).

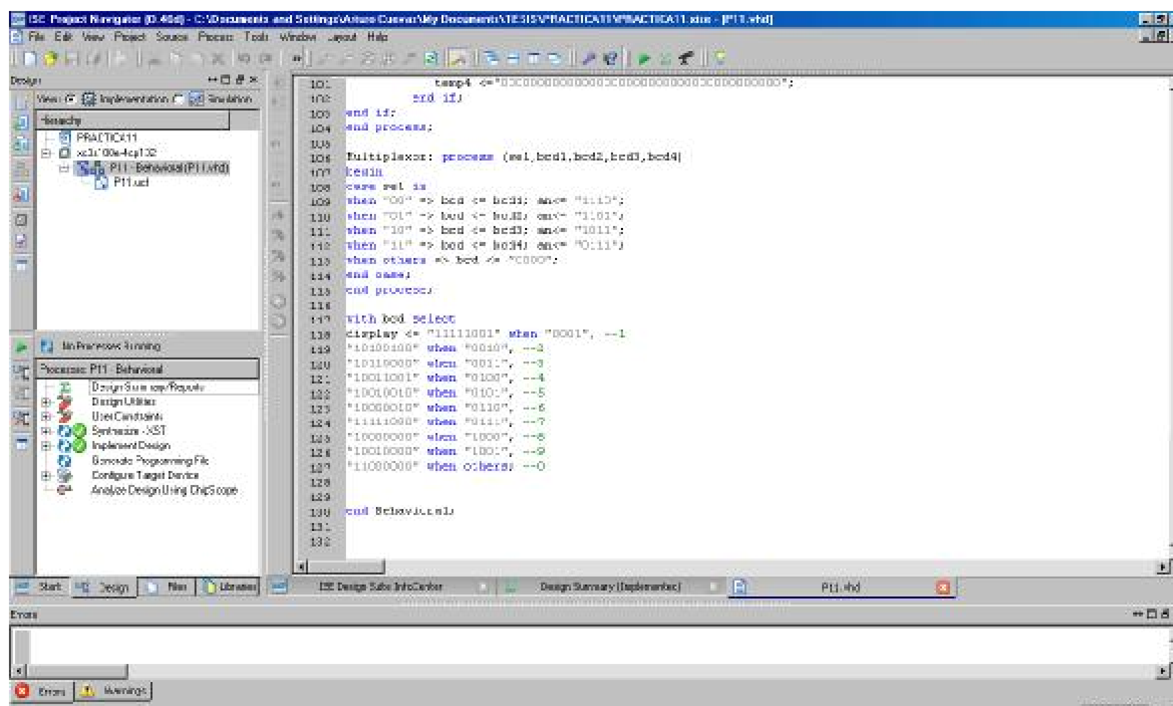


Figura 3.11.11 Verificación de Síntesis y Diseño

Después de la verificación, generamos el programa bit y lo bajamos a la tarjeta con el programa Adept (ver figura 3.11.12).

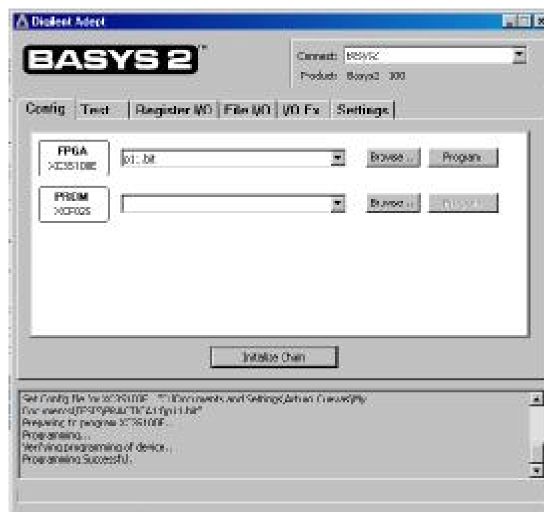


Figura 3.11.12 Programa p11.bit

Los resultados de implementación se muestran en las figuras 3.11.13, 3.11.14, 3.11.15 y 3.11.16.

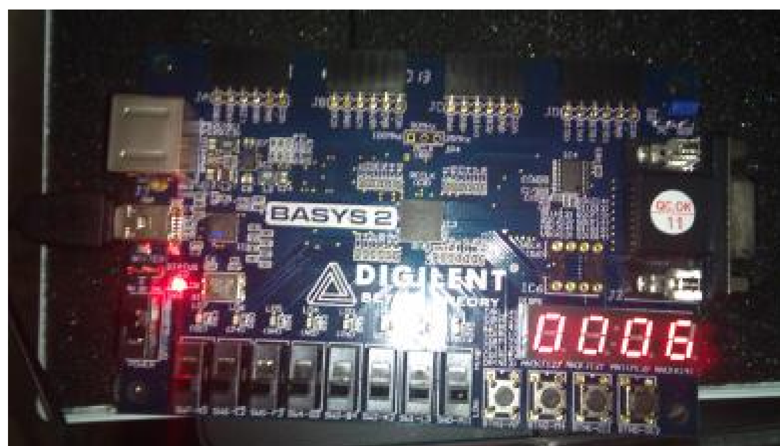


Figura 3.11.13 Contador en 0006

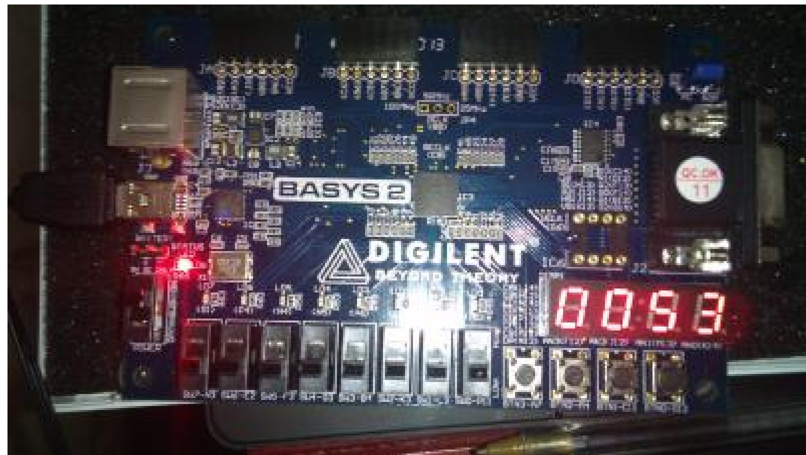


Figura 3.11.14 Contador en 0053

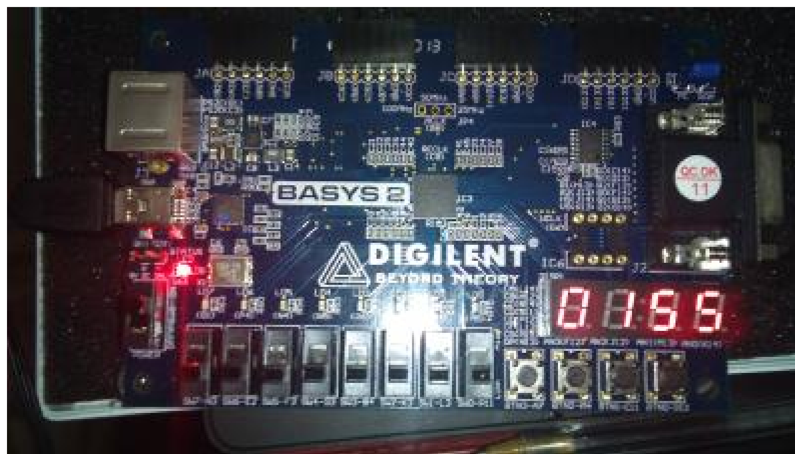


Figura 3.11.15 Contador en 0155

Figura 3.11.13 Contador en 1098

Conclusiones.

Se creó el proyecto contador en VHDL para la tarjeta BASYS 2. En este proyecto se utilizarón módulos de las prácticas anteriores como el multiplexor, decodificador y contador. Todos los módulos se implementarón a través de procesos, cada uno de estos procesos depende de un reloj principal el cual sincroniza a todos los demás.

Por otro lado, se cambiaron los valores de las señales para la simulación en ISim, donde:

DECENA = UNIDAD*10 - 1

CENTENA = UNIDAD*100 -1

MILLAR=UNIDAD*1000 - 1

Esto sucede poque en ISim los contadores comienzan con el valor 1 en lugar de cero, como lo es en la implementación real en la tarjeta BASYS 2. Con este escenario se recomienda verificar simulación contra implementación, ya que los resultados podrían variar.

Práctica 12

Modelado e implementación de control de un par de semáforos

Objetivo

Diseñar el controlador de un par de semáforos de la intersección de dos calles utilizando cartas ASM, implementarlo en lenguaje VHDL y bajarlo a la tarjeta de prueba.

Desarrollo

El diseño consistirá en controlar un par de semáforos como los que se muestran en la figura 3.12.1, el semáforo "A" comenzará en verde y el semáforo "B" en rojo, los tiempos en verde serán iguales para ambos semáforos y permanecerán por 8 segundos, la transición de verde a rojo será pasando por ámbar y el sistema incluirá un bit para poder reiniciar el sistema el cual actuará de manera asíncrona al mismo.

El semáforo A controla el tránsito de sur a norte y el semáforo B controlará el tránsito de este a oeste. Los estados del sistema serán sincronizados con el reloj interno de la tarjeta BASYS 2 el cual es de 50 MHz suministrado a través del pin B8.

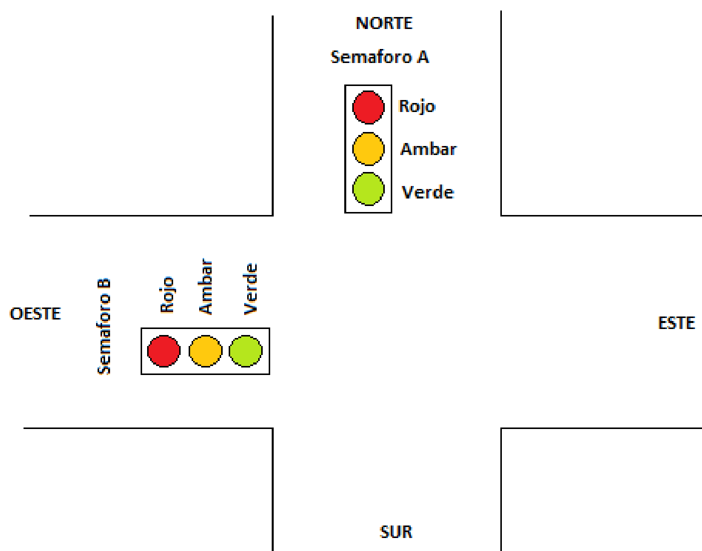


Figura 3.12.1 Diagrama de control de un par de semáforos

De la imagen anterior y de las especificaciones del problema inferimos entonces que las señales de entrada serán el bit de reinicio, y la señal de reloj, para las señales de salida representaremos a los semáforos con seis led´s de la tarjeta.

Nuestro diseño de control quedaría como el que se muestra en la figura 3.12.2

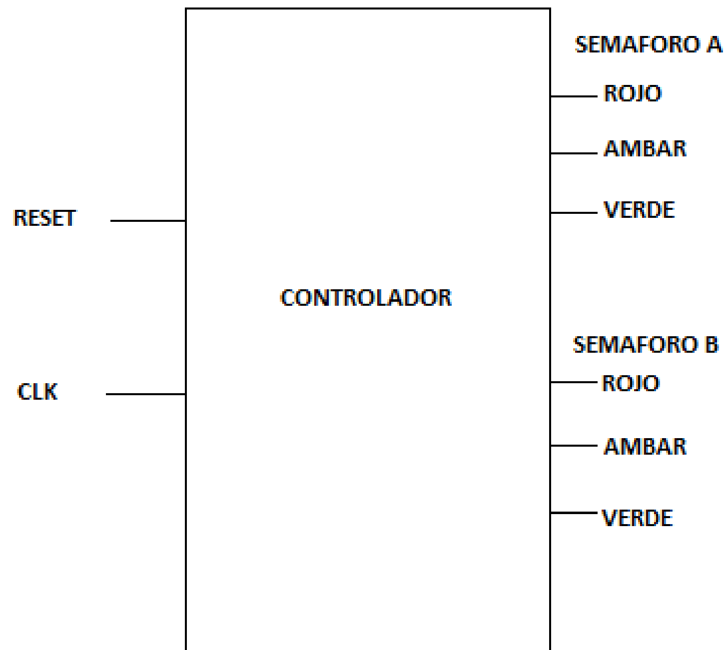


Figura 3.12.2 Señales de entrada y salida del controlador

En la figura 3.12.3 se muestra el algoritmo de control utilizando carta ASM el cual permite detallar de una manera didáctica el comportamiento de nuestro sistema:

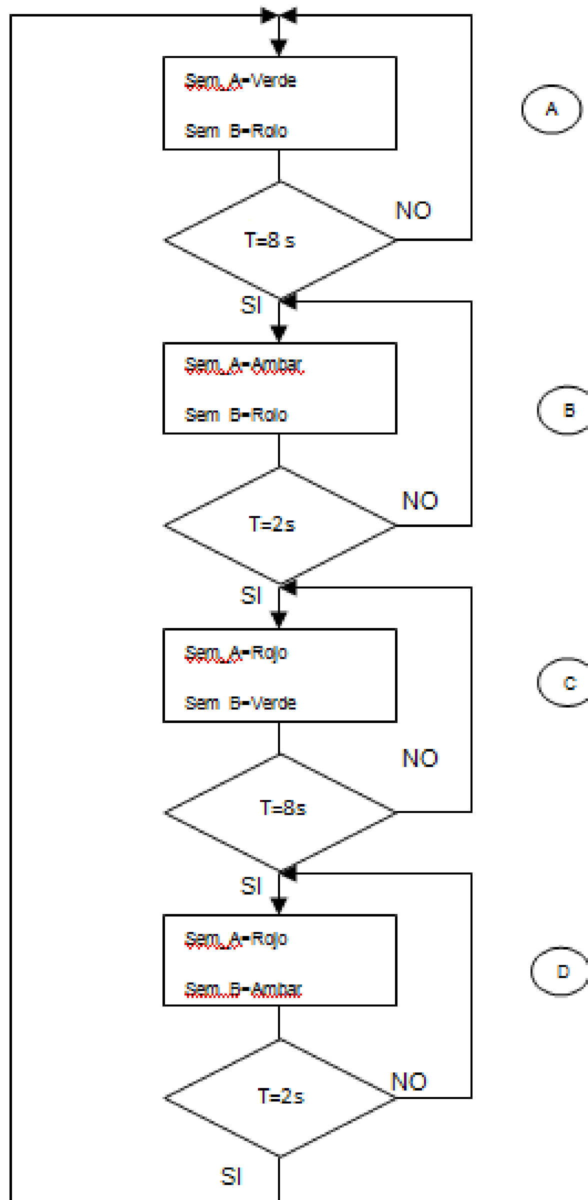


Figura 3.12.3 Carta ASM

Ahora comenzaremos con nuestro diseño del controlador en VHDL utilizando la herramienta de Xilinc ISE Desing, generamos nuestro proyecto al cual nombraremos Practica_12 tal y como se muestra en la figura 3.12.4.

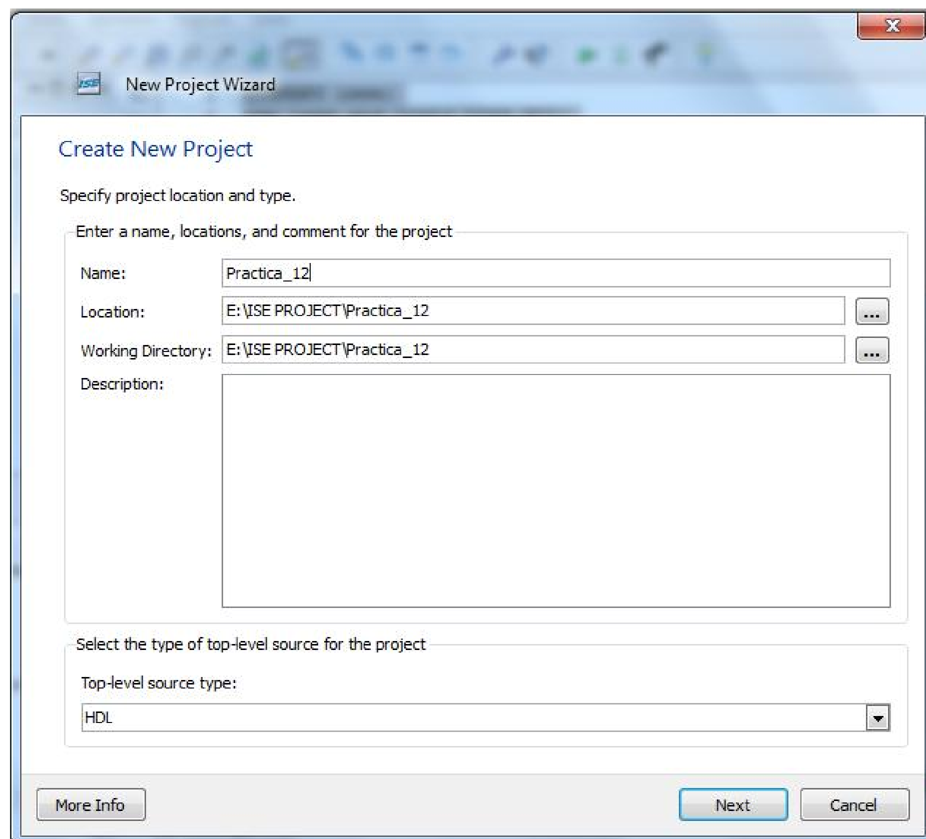


Figura 3.12.4 Creación del proyecto.

Posteriormente nos cercioramos que la configuración de nuestro proyecto se encuentre como se muestra en la figura 3.12.5.

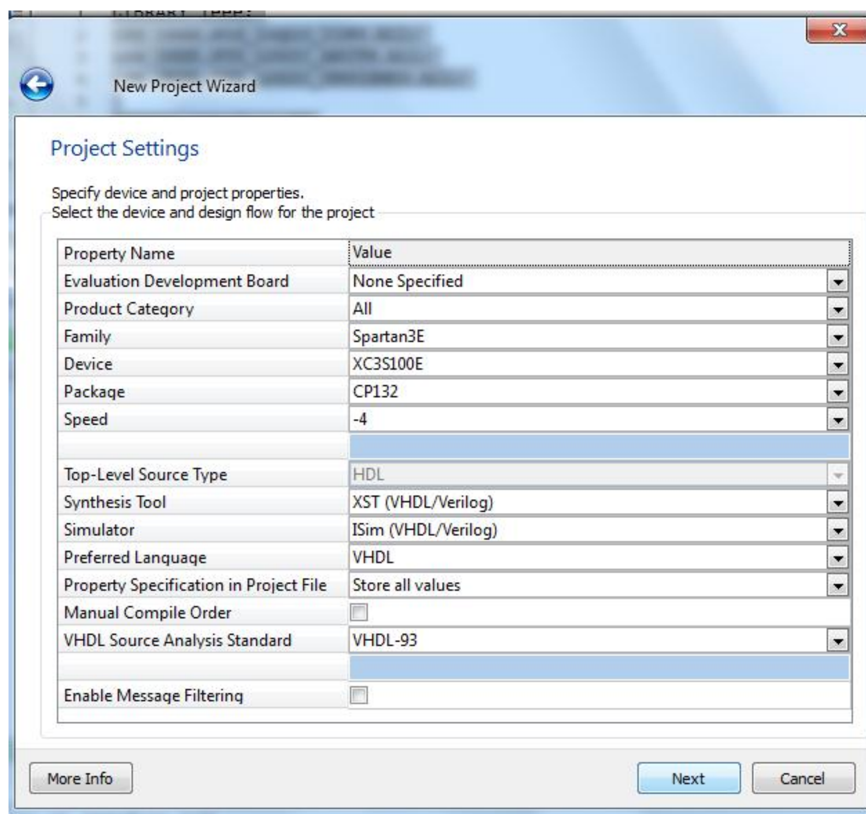


Figura 3.12.5 Configuración de proyecto

Una vez creado nuestro proyecto agregaremos una fuente de tipo “VHDL Module” y la nombraremos “semáforo” como se muestra en la figura 3.12.6

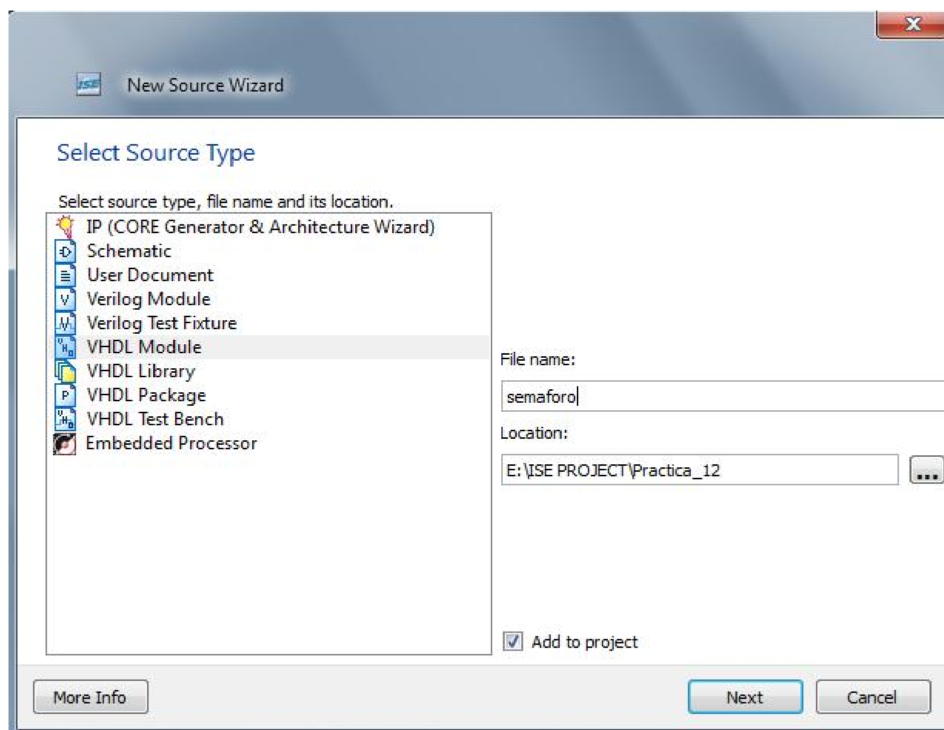


Figura 3.12.6 Configuración de proyecto

Como ya se ha mencionado, nuestro diseño contará con dos entradas de tipo std_logic las cuales nombraremos “reset” y “clk”, para referirnos a nuestro bit de reinicio y la señal de reloj respectivamente. Las salidas serán de tipo std_logic_vector a las cuales nombraremos Sem_A y Sem_B para referirnos a nuestros pares de semáforos A y B respectivamente, la arquitectura la nombraremos descripción. El diseño contara con tres procesos, denominadas máquina, salida y contador (ver figura 3.12.7).

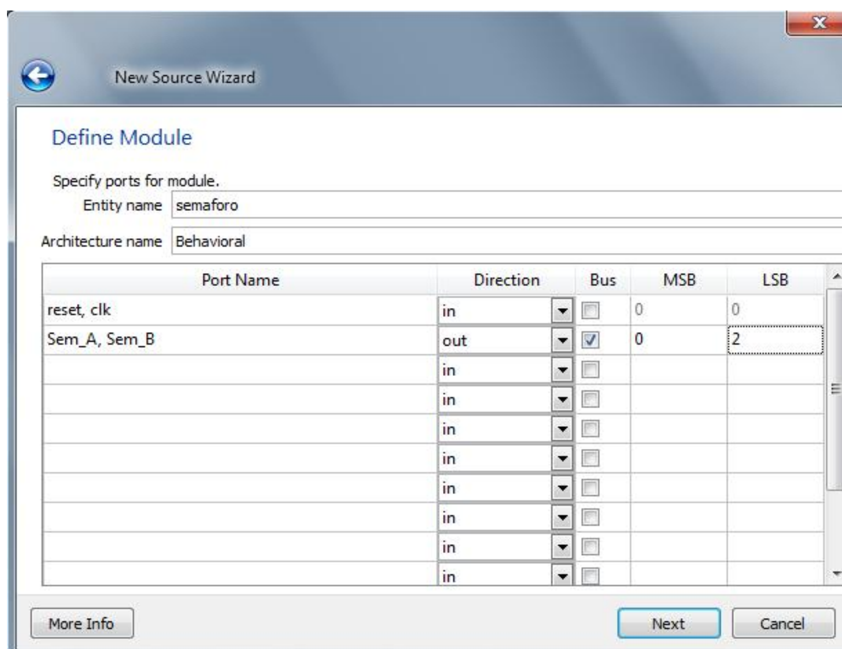


Figura 3.12.7 Definición de la fuente

A continuación se presenta la descripción en código VHDL del control de semáforos

```
--Inicia Código

LIBRARY ieee;

use ieee.std_logic_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY semaforo IS

PORT (reset,clk: IN std_logic;

Sem_A,Sem_B: OUT std_logic_vector(0 TO 2));

END semaforo;

ARCHITECTURE descripcion OF semaforo IS

TYPE estado IS (A,B,C,D);

CONSTANT verde: std_logic_vector(0 TO 2):="001";

CONSTANT amarillo: std_logic_vector(0 TO 2):="010";

CONSTANT rojo: std_logic_vector(0 TO 2):="100";
```

```
SIGNAL presente: estado:=A;
SIGNAL cuenta: STD_LOGIC_VECTOR (4 downto 0);
SIGNAL divisor: STD_LOGIC_VECTOR (25 downto 0);

BEGIN
maquina:
PROCESS(clk,reset)
BEGIN
IF reset='1' THEN
presente<=A;
ELSIF clk='1' AND clk'EVENT
THEN
CASE presente IS
WHEN A=>
If (cuenta >= "00000" and cuenta <= "01000") THEN
presente<=A;
else presente<=B;
END IF;
WHEN B=>
If (cuenta >= "01001" and cuenta <= "01010") THEN
presente<=B;
else presente<=C;
END IF;
WHEN C=>
If (cuenta >= "01011" and cuenta <= "10010") THEN
presente<=C;
else presente<=D;
```

```
END IF;
WHEN D=>
If (cuenta >= "10011" and cuenta <= "10100") THEN
presente<=D;
else presente<=A;
END IF;
END CASE;
END IF;
END PROCESS maquina;

salida:
PROCESS(presente)
BEGIN
CASE presente IS
WHEN A=>
Sem_A<=verde;
Sem_B<=rojo;
WHEN B=>
Sem_A<=amarillo;
Sem_B<=rojo;
WHEN C=>
Sem_A<=rojo;
Sem_B<=verde;

WHEN D=>
Sem_A<=rojo;
Sem_B<=amarillo;
```

```
END CASE;
END PROCESS salida;
CONTADOR:
process(CLK,reset)
begin
if clk='1' AND clk'EVENT then
if reset='1' then
divisor <= (others => '0');
cuenta <= "00000";
else
if divisor < "10111110101111000010000000" then -- si la señal divisor es menor a 50 MHz
divisor <= divisor + 1; -- señal divisora aumenta en 1
else
divisor <= (others => '0');
if   cuenta < "10100" then
cuenta <= cuenta + "00001" ;
else
cuenta <= "00000";
end if;
end if;
end if;
end if;
end process CONTADOR;
END descripcion;
--Termina Código
```

Guardamos los cambios en el proyecto, compilamos y verificamos que nuestro proyecto no genere errores (ver figura 3.12.8)

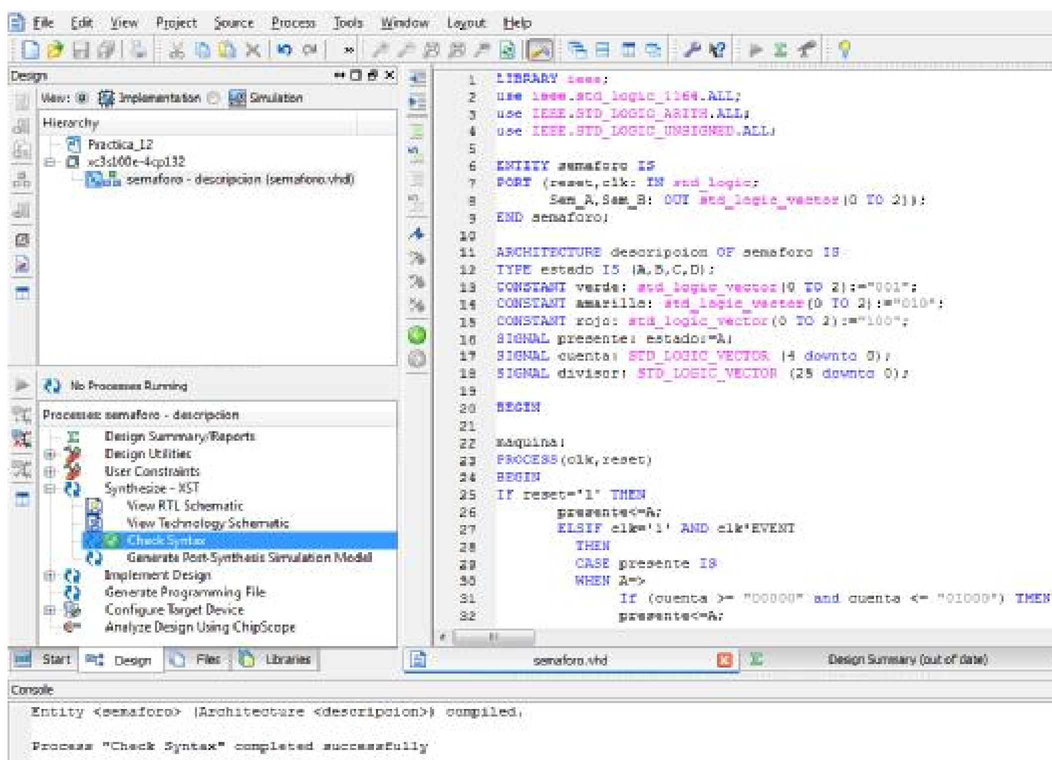


Figura 3.12.8 Revisión de sintaxis

Para simular nuestro proyecto utilizaremos la herramienta ISim Simulator, colocaremos los siguientes valores para poder visualizar el funcionamiento de nuestro diseño, al divisor asignaremos el valor constante en binario " 10111110101111000001111111" el cual en decimal equivale a 49999999 que representa un valor antes de que el contador se incremente en "1", a la entrada reset le asignaremos el valor constante en binario "0", a la señal de entrada clk definiremos un reloj con un periodo de 10 ms.

Una vez asignados los valores a las entradas daremos click en el icoco Run All y analizamos el comportamiento obtenido, en la figura 3.12.9 se observa el cambio de los estados del A al B, del B al C, del C al D, del D al A y sus salidas correspondientes.

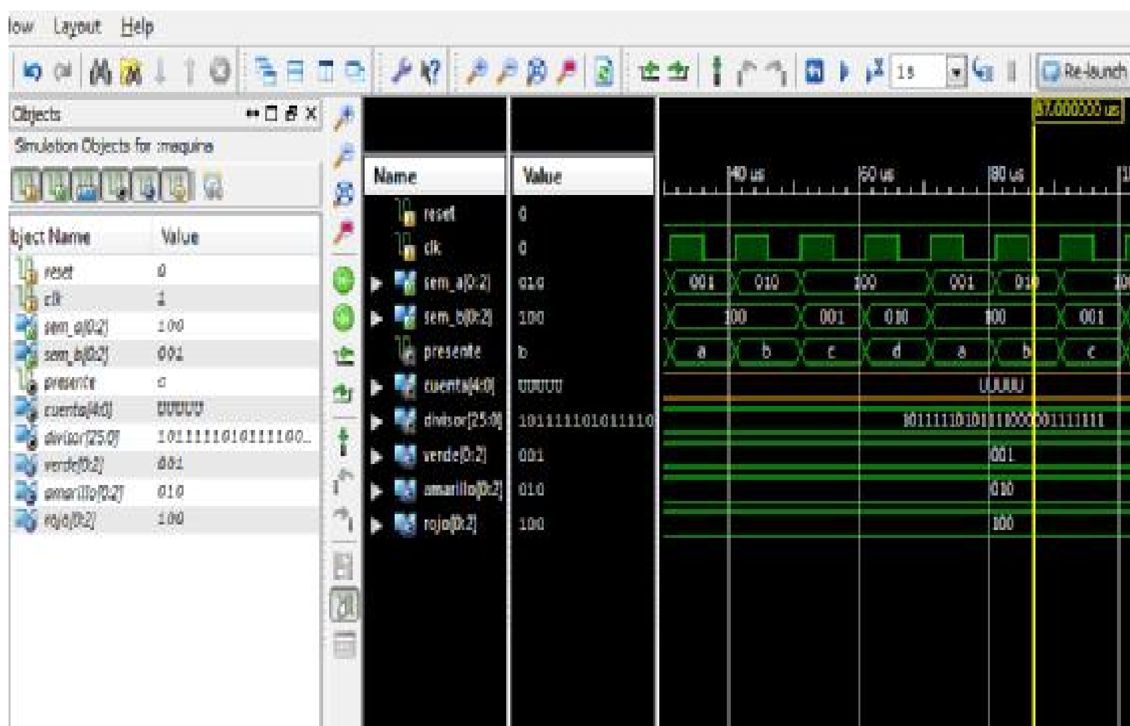


Figura 3.12.9 Simulación 1 con la herramienta ISim simulator

Ahora solo cambiaremos el valor de la entrada reset al valor de "1" para verificar que independientemente del estado en el que se presente el reinicio del sistema el estado se mantendrá en A hasta que el bit de ponga a "0" nuevamente (ver figura 3.12.10).

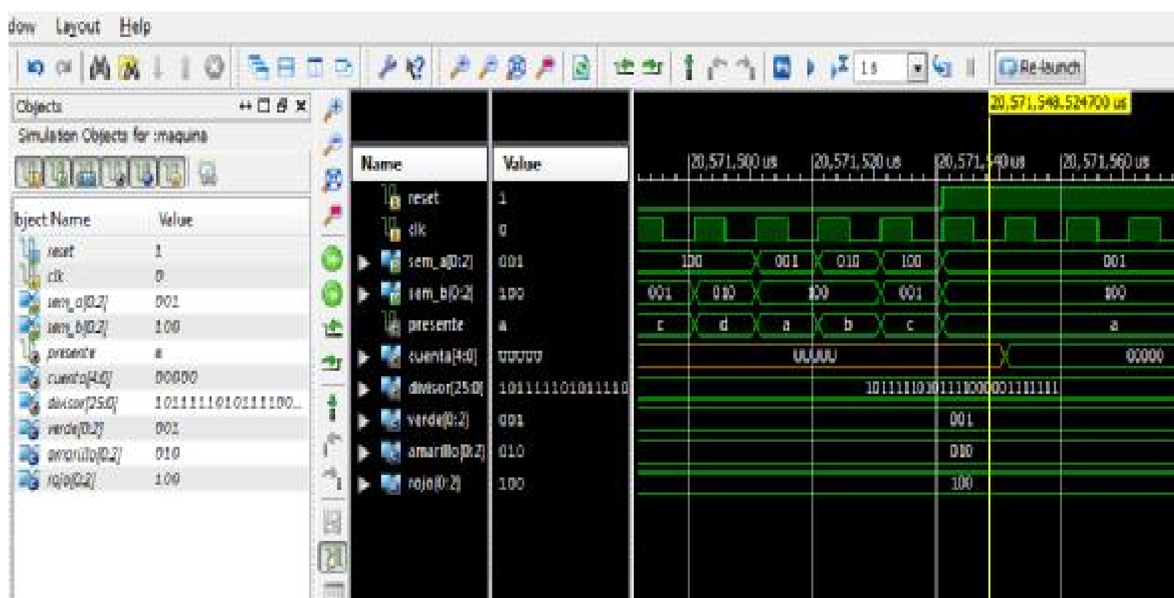


Figura 3.12.10 Simulación 2 con la herramienta ISim simulator

Una vez revisado que nuestro diseño del controlador funciona correctamente a nivel de simulación, generaremos nuestro archivo .bit para poder bajarlo a la tarjeta de pruebas BASYS 2 y probarlo físicamente.

La asignación de pines quedará como se muestra en la figura 3.12.11, una vez asignados los puertos guardamos los cambios realizados.

Entradas:

Reset= P11

Clk= B8

Salidas:

Sem_A[2]: N4

Sem_A[1]: P4

Sem_A[0]: G1

Sem_B[2]: P7

Sem_B[1]: P6

Sem_B[0]: N5

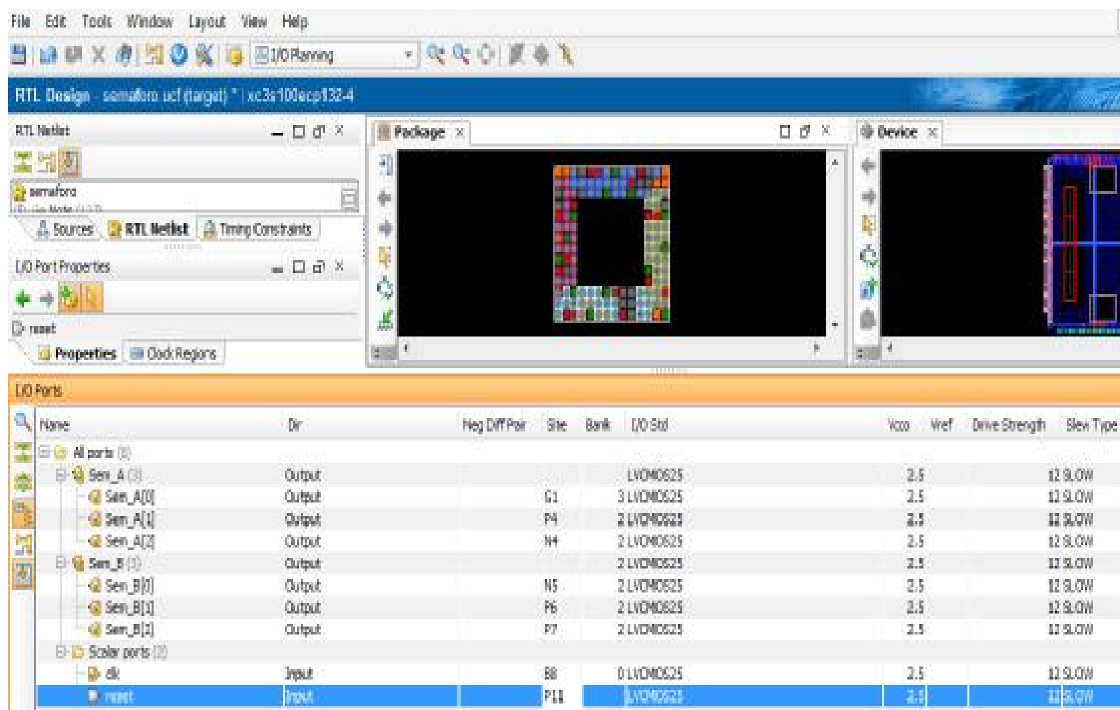


Figura 3.12.11 Asignación de puertos de entrada/salida.

Antes de generar el archivo .bit revisamos que el valor de reloj se encuentre el valor “JTAG Clock” como se muestra en la figura 3.12.12

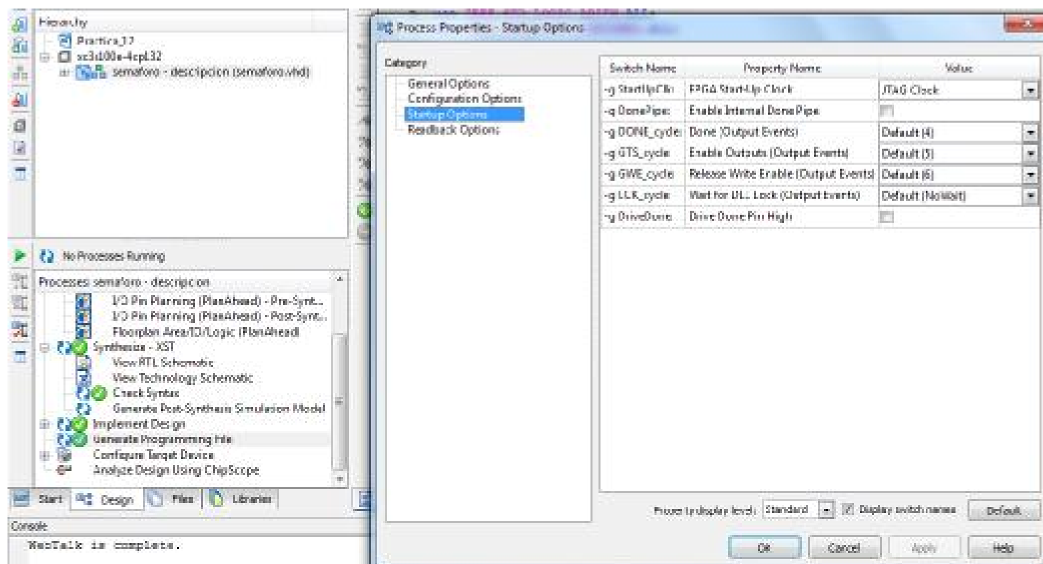


Figura 3.12.12 Propiedades del reloj del FPGA

Abrimos el programa Digilent Adept para bajar nuestro diseño a la tarjeta BASYS 2, buscamos la ubicación del archivo .bit, lo seleccionamos y damos click en la opción program como se muestra en la figura 3.12.12

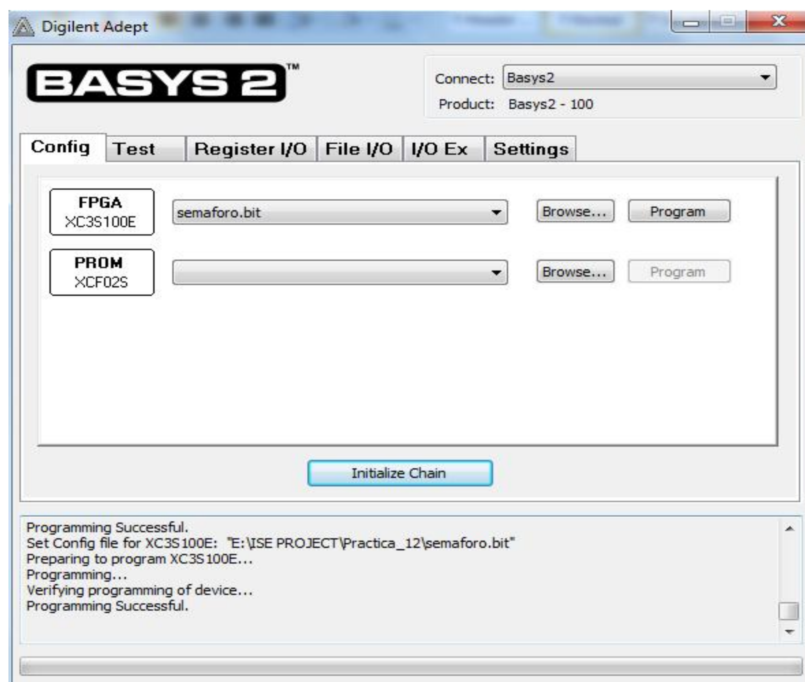


Figura 3.12.12 Programa Digilent Adept para bajar archivo .bit

En la figura 3.12.13 observamos el diseño en el estado A, el semáforo A se encuentra en verde y el semáforo B se encuentra en rojo, la entrada reset de encuentra desactivada

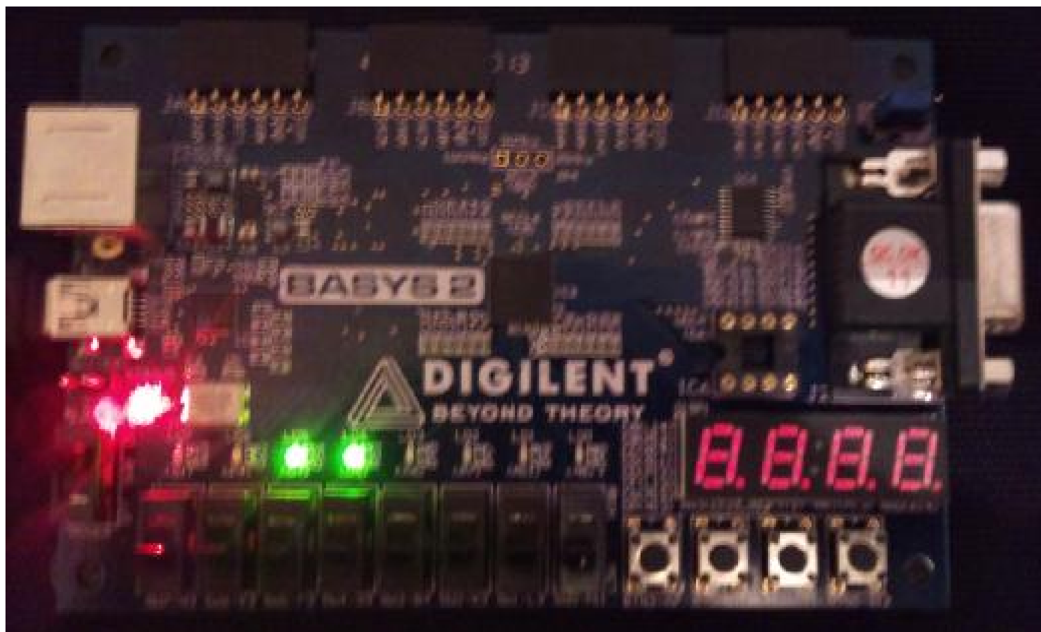


Figura 3.12.13 Tarjeta de prueba, Estado=A, Reset=0.

En la figura 3.12.14 observamos el diseño en el estado B, el semáforo A se encuentra en amarillo, el semáforo B se mantiene en rojo, la entrada reset de encuentra desactivada

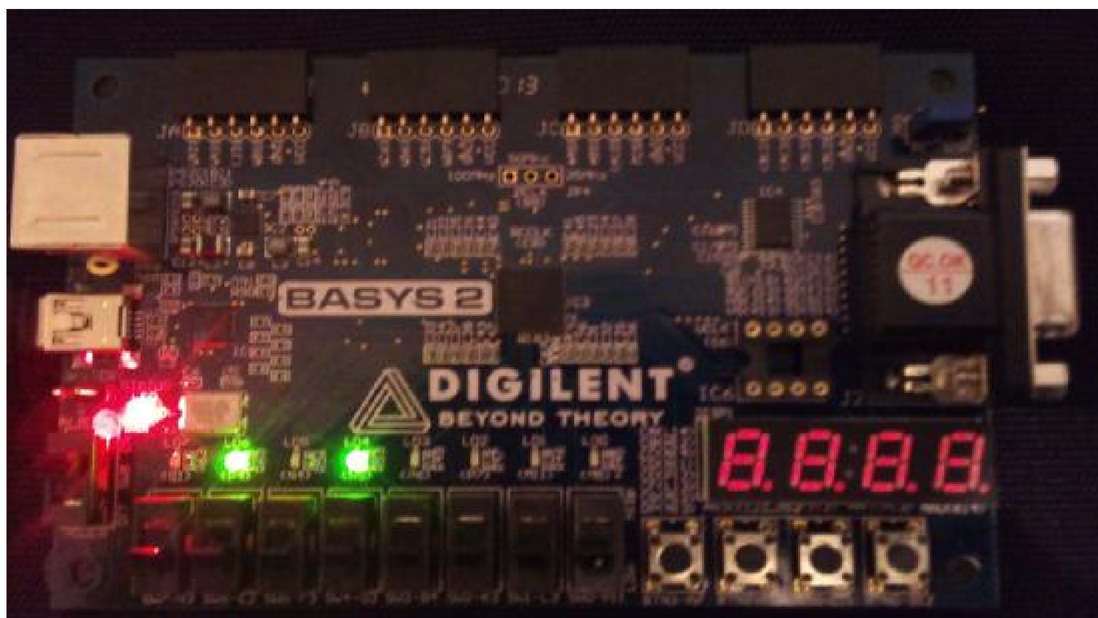


Figura 3.12.14 Tarjeta de prueba, Estado=B, Reset=0.

En la figura 3.12.15 observamos el diseño en el estado C, el semáforo A se encuentra en rojo, el semáforo B pasa a verde, la entrada reset de encuentra desactivada

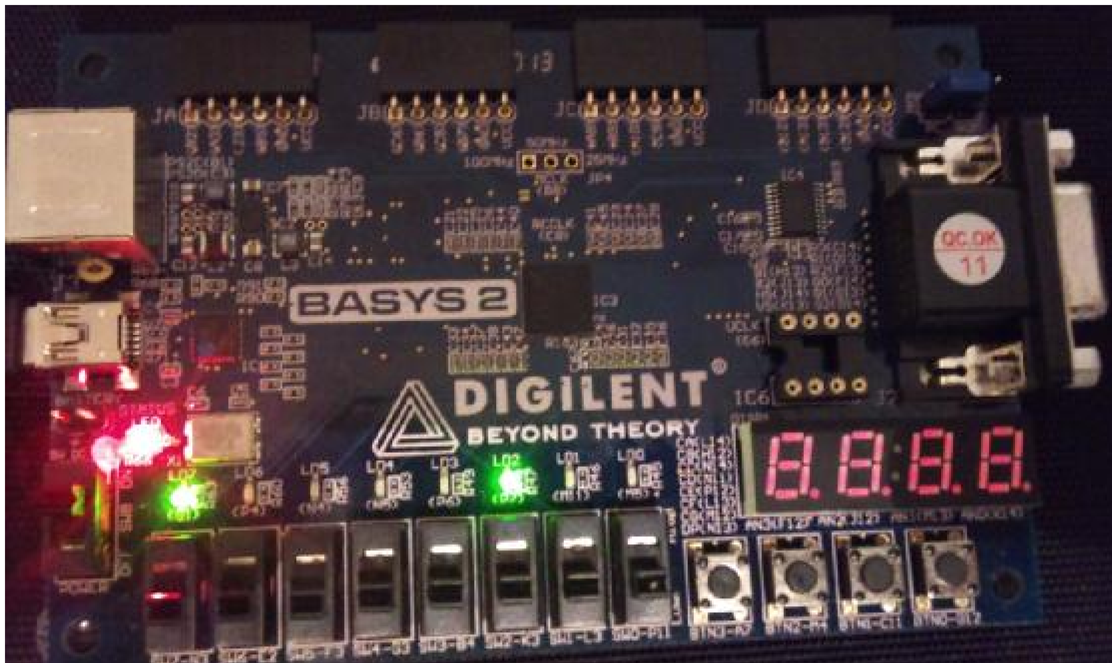


Figura 3.12.15 Tarjeta de prueba, Estado=C, Reset=0.

Conclusiones.

Se diseña exitosamente el control de un par de semáforos utilizando cartas ASM y utilizando el lenguaje de descripción VHDL. El desarrollo de cartas tiene mucho que ver con la forma y sentido común de interpretar un problema para luego visualizar los caminos para encontrar una solución óptima, por esto mismo puede existir más de una manera de resolver un problema.

CAPÍTULO 4

CONCLUSIONES Y RESULTADOS

La descripción de los PLD’s que se hizo en este trabajo nos permitió exponer como es que han ido evolucionando estos dispositivos y mostrar las principales diferencias entre ellos, lo que marcó la pauta para orientarnos al manejo de los CPLD’s y FPGA’s, que son dispositivos con una estructura más compleja y poseen la característica de ser reconfigurables ilimitadamente, lo cual facilita y optimiza la implementación de sistemas grandes.

El manejo de VHDL que se integró al trabajo, es un lenguaje de descripción de hardware que por sus características tiene ciertas ventajas sobre los otros LDH’s, que lo hacen el más usado y aceptado por las universidades del mundo. Tal como se mostró en el capítulo 2, donde se introdujeron los principios básicos para la descripción de circuitos digitales en VHDL, mismos que nos permitieron modelar desde los diseños más simples hasta algunos complejos.

El ISE Design Suite de Xilinx que se introdujo al trabajo, es el ambiente de desarrollo integrado a través del cual nos fue posible modelar, diseñar e implementar cada una de las prácticas que se propusieron en el capítulo 3. En cada práctica se fueron introduciendo nuevos conceptos que permitieron interactuar de forma más completa con el ambiente de desarrollo e ir aprendiendo a usar las distintas herramientas de las que dispone, tal como usar la herramienta ISE Project Navigator en la cual se describen los circuitos lógicos digitales a través de VHDL, la herramienta ISim que es donde se simula el funcionamiento del diseño y la herramienta PlanAhead que nos permite hacer la asignación de pines para la implementación en la tarjeta de desarrollo.

Por medio del ISE Desing Suite es posible modelar CPLD’s y FPGA’s, específicamente para este trabajo se utilizó el Spartan3E-100 CP132 de Xilinx, el cual viene integrado en la tarjeta de desarrollo BASYS 2 de DIGILENT, que fue la tarjeta de desarrollo donde se implementaron todas las prácticas que se diseñaron, lo cual nos permitió apreciar el funcionamiento correcto de nuestros diseños.

El software ISE Design Suite de Xilinx, ADEPT y la tarjeta de desarrollo BASYS 2 de DIGILENT, son los elementos que nos permitieron pasar del “plano teórico” al “plano físico” de una forma práctica y rápida que en muchas ocasiones no suele serlo. Ya que para muchos de los diseños de este trabajo, de haberse realizado con compuertas lógicas, multiplexores, led´s, resistencias, display´s, etc., habríamos requerido por lo menos de un osciloscopio, generador de funciones, fuente de CD, etc., equipos que difícilmente se tienen al alcance. Sin embargo hoy en día la mayoría de los alumnos cuentan con una computadora personal con conexión a internet, fuente desde la cual pueden descargar el software que se utilizó y adquirir alguna tarjeta de desarrollo como la que se manejó en este trabajo.
Se ha creado un trabajo donde se establecen las condiciones necesarias para que los alumnos orientados al área de sistemas digitales puedan adquirir los conocimientos para aprender a diseñar circuitos digitales a través de los dispositivos lógicos programables, principalmente en CPLD’s y/o FPGA`s.

Se ha mostrado como en el modelado, diseño e implementación de prácticas del área de sistemas digitales se puede optimizar el tiempo de trabajo que dedica el alumno cuando trabaja con un FPGA, ya

que dejará de lado el alambrado de su proyecto y eliminará la posibilidad de que haya algún elemento defectuoso (display, resistencia, compuerta, etc.) y se enfocará completamente en su diseño.
Se ha elaborado un trabajo que posee una investigación que recopila información detallada y clara sobre los CPLD's y FPGA's, cuenta con un capítulo en donde cada práctica es resuelta completamente, ésta información ha sido estructurada didácticamente lo que ayuda a su mejor comprensión, por estas razones es que consideramos que este trabajo en un futuro sea tomado en cuenta como libro de prácticas o de libro de apoyo para los alumnos del área de sistemas digitales, donde el alumno encontrará un complemento a sus estudios y podrá obtener conocimientos nuevos en la programación de CPLS's y/o FPGA's. Estos conocimientos permitirán que el alumno adquiera una habilidad para implementar soluciones en sus proyectos de ingeniería del área de sistemas digitales, lo que marca la pauta para que el alumno por cuenta propia continúe extendiendo sus conocimientos sobre el tema durante su carrera y al finalizar sus estudios contará con una considerable experiencia en el manejo de CPLD´s y/o FPGA´s que le dará la oportunidad de ser un profesionista más competitivo.

BIBLIOGRAFÍA.

WAKERLY, John F.
Digital Design principles & practices
3a edición
Upper Saddle River
Prentice Hall, 2000

UYEMURA, John P.
Diseño de Sistemas digitales Un enfoque integrado
México
Thomson, 2000

FLETCHER, William I.
An Engineering Aproach to Digital Design
New Jersey
Prentice Hall, 1990

BROWN y Vranesic.
Digital logic with VHDL Design
New York
McGraw-Hill, 2003

FLOYD THOMAS L.
Fundamentos de Sistemas Digitales
7a Edición
Prentice Hall