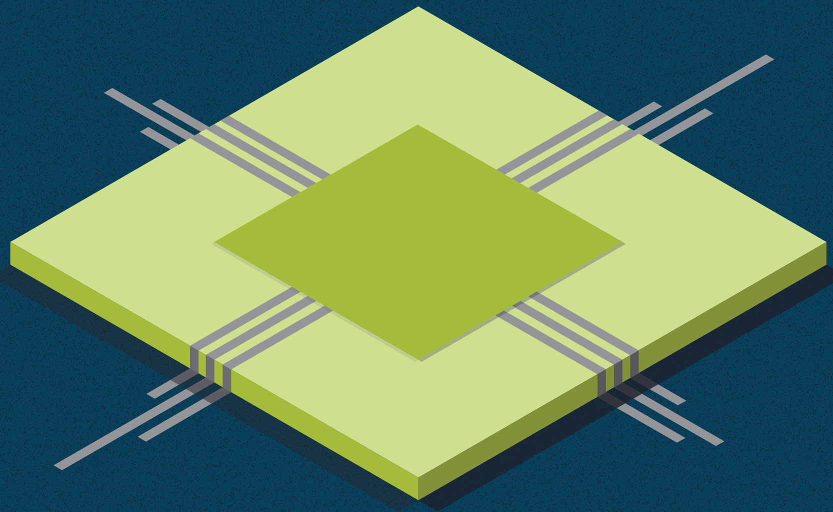
DISEÑO E IMPLEMENTACIÓN DE **CIRCUITOS DIGITALES BASADOS EN FPGA**

TATIANA MARÍN RAMÍREZ
JOSÉ CORVALÁN MARTÍNEZ



Editorial usm

Diseño e implementación de circuitos digitales basados en FPGA

Tatiana Marín y José Corvalán

© Tatiana Marín y José Corvalán

ISBN: 978-956-356-134-0

Publicado en 2025 por Editorial USM

Diseño de portada y diagramación
Rodrigo Ruiz

Corrección de estilo
Miguelángel Sánchez

Queda prohibida la reproducción parcial o total de este libro sin la autorización previa de Editorial USM y los autores.

Impreso por Fyrma

Cómo citar
Diseño e implementación de circuitos digitales basados en FPGA. (2025). Libros USM. https://doi.org/10.82140/62qq-s743 https://libros.usm.cl/utfsm/catalog/book/22

Diseño e implementación de circuitos digitales basados en FPGA © 2025 por Tatiana Marín y José Corvalán tiene licencia CC BY-NC-ND 4.0. Para ver una copia de esta licencia, visite https://creativecommons.org/licenses/by-nc-nd/4.0/

DISEÑO E IMPLEMENTACIÓN DE **CIRCUITOS DIGITALES BASADOS EN FPGA**

TATIANA MARÍN RAMÍREZ
JOSÉ CORVALÁN MARTÍNEZ

ÍNDICE

Introducción 11

Capítulo 1 – Qué es una FPGA

1.1. Introducción 12
1.2. Bloques lógicos configurables 13
1.3. Interconexiones programables 13
1.3.1. Flexibilidad 13
1.3.2. Configurabilidad 13
1.3.3. Alto nivel de integración 13
1.3.4. Optimización de recursos 14
1.4. Bloques de entrada/salida 14
1.4.1. Compatibilidad de estándares 14
1.4.2. Cantidad de pines 14
1.4.3. Características eléctricas 14
1.4.4. Recursos de interfaz específicos 14
1.4.5. Funciones de propósito especial 14
1.5. Generalidades sobre cómo se trabaja con una FPGA 14
1.5.1. Etapas 15
1.6. Ejemplo inicial: Circuitos que implementan una compuerta *and* y una *or* 15
1.6.1. Interconexión 15
1.6.2. Recursos de la tarjeta BASYS-3/FPGA para configurar los circuitos 16
1.6.3. Programa desarrollado con Verilog 16
1.6.4. Asignación de recursos de la tarjeta a las entradas y salidas de las compuertas 16
1.6.5. Circuitos resultantes 17

Capítulo 2 – Descripción de la tarjeta BASYS-3

2.1. Generalidades 18
2.2. Recursos de la tarjeta 18

Capítulo 3 – Cómo se usa la tarjeta BASYS-3

3.1. Introducción 22
3.2. Carga de Vivado en un computador 22
3.3. Carga de los drivers de las tarjetas en Vivado 25
3.4. Crear, programar y ejecutar un proyecto 25
3.4.1. Crear un proyecto 26
3.4.2. Escribir un programa en Verilog para configurar un circuito 27
3.4.3. Asignar recursos de la tarjeta BASYS-3/FPGA al circuito 28
3.4.4. Sintetizar e implementar el circuito 29
3.4.5. Generar el *bitstream* y traspasar el programa a la tarjeta BASYS-3/FPGA 30

3.5. Selección del modo de carga 31
3.5.1. Modo normal (JTAG) 31
3.5.2. Modo QSP 31
3.5.3. Modo USB (con pendrive) 34
3.6. Carpetas de Vivado 34
3.6.1. Carpeta principal del proyecto 34

Capítulo 4 – El lenguaje VERILOG

4.1. Introducción 36
4.2. Tipos de descripciones 38
4.2.1. Descripción funcional (ASSIGN) 38
4.2.2. Descripción estructural («AND», «OR», «XOR», «NOT») 39
4.2.3. Descripción procedimental («ALWAYS») 39
4.3. Arreglos [M:L] 40
4.4. Operadores 41
4.4.1. Operadores lógicos a nivel de bit... 41
4.4.2. Operadores relacionales... 42
4.4.3. Operadores lógicos... 42
4.4.4. Operadores artiméticos... 42
4.4.5. Operadores de propósitos especiales... 42
4.5. Representación de constantes 43
4.6. Sentencias condicionales 43
4.6.1. Sentencia «IF... ELSE... » 43
4.6.2. Sentencia «CASE... ENDCASE» 44
4.6.3. Sentencia «FOR» 46
4.7. Circuitos secuenciales 47
4.7.1. Detección de cantos (flancos) de reloj 47
4.7.2. Definición de constantes (parameter) 48
4.7.3. Concatenación 48
4.7.4. Reloj interno de la tarjeta BASYS-3 49
4.7.5. Asignaciones bloqueantes y no bloqueantes 50
4.7.6. *Flip flop* tipo jk 51
4.8. Aplicaciones de los *flip flops* 52
4.8.1. Registros de desplazamiento 52
4.8.2. Contadores 54
4.9. Instancias 55
4.10. Funciones 56
4.11. Tareas 58

Capítulo 5 – Funcionalidades especiales de VIVADO

5.1. Simulación ... 59
5.2. Catálogo IP ... 66
5.2.1. Proceso de integración y configuración de una IP ... 66
5.2.2. Ejemplos de bloques disponibles en el IP Catalog ... 67

Capítulo 6 – Prácticas de laboratorio

6.1. Introducción ... 68
6.2. Prácticas ... 68
6.2.1. Práctica 1: Compuertas lógicas ... 68
6.2.2. Práctica 2: Circuitos combinacionales ... 70
6.2.3. Práctica 3: Uso de entradas y salidas externas por las puertas PMOD ... 71
6.2.4. Práctica 4: Aplicación de control con circuitos combinacionales ... 74
6.2.5. Práctica 5: Multiplexor de dos canales ... 76
6.2.6. Práctica 6: Decodificador BCD a siete segmentos ... 78
6.2.7. Práctica 7: Transmisión de estados digitales con multiplexor y demultiplexor ... 79
6.2.8. Práctica 8: *Flip flops* ... 82
6.2.9. Práctica 9: Registros de desplazamiento ... 85
6.2.10. Práctica 10: Contadores ... 89

Capítulo 7 – Prácticas avanzadas

7.1. Introducción ... 93
7.2. Prácticas avanzadas ... 93
7.2.1. Práctica 11: Sumadores de 4 bit ... 93
7.2.2. Práctica 12: Uso de los *displays* con datos independientes ... 96
7.2.3. Práctica 13: Uso del reloj interno de la tarjeta ... 98
7.2.4. Práctica 14: Control de semáforos con *latch* ... 100
7.2.5. Práctica 15: Conversión análogo a digital ... 108
7.2.6. Práctica 16: Comunicación serial UART ... 122

Anexos

A.1. Uso de VIVADO ... 129
A.1.1. Creación de un proyecto BASYS-3/FPGA ... 129
A.1.2. Escribir un programa en Verilog y asignar recursos de la tarjeta ... 129
A.1.3. Sintetizar e implementar el circuito y generar el *bitstream* ... 130
A.2. Recursos de la tarjeta BASYS-3 ... 132

PRÓLOGO

Este libro nace con el propósito de proveer un texto en español que guie el complejo dominio de las tarjetas de desarrollo basadas en arreglos de compuertas programables en campo (field programmable gate arrays, FPGA) y explique cómo su uso acelera el aprendizaje de los sistemas digitales.

La programación en FPGA es una habilidad cada vez más demandada en el ámbito de la electrónica. Por ello, en este libro encontrará una introducción práctica y accesible al uso de FPGA, específicamente utilizando la tarjeta de desarrollo BASYS-3 de Digilent y el lenguaje de descripción de hardware Verilog. Concebido para quienes están dando sus primeros pasos en la programación de hardware, el libro aborda los conceptos esenciales de manera clara y directa.

A través de siete capítulos, los lectores aprenderán a diseñar y programar circuitos digitales, desde los componentes más básicos hasta realizaciones más complejas. Cada capítulo introduce un tema central, seguido de ejercicios prácticos que les permitirán aplicar de inmediato lo aprendido. Estos ejercicios han sido seleccionados cuidadosamente para que los estudiantes puedan avanzar en dificultad de manera progresiva, construyendo su confianza y entendimiento a lo largo del proceso.

Lo que distingue a este libro es su enfoque en el aprendizaje a través de la práctica. Los ejercicios están planteados de modo que cada uno aporte un valor concreto en la formación del estudiante, proporcionando una experiencia práctica que conecta directamente con situaciones reales que podrían enfrentar en sus futuras carreras. Se incluyen soluciones detalladas y explicaciones paso a paso para cada ejercicio, de modo que se entienda no solo cómo resolver un problema, sino también los fundamentos detrás de cada solución. Esta metodología facilita la comprensión profunda y promueve el desarrollo de un pensamiento crítico en el diseño digital.

En la experiencia de los autores, el uso de este texto ha permitido a los estudiantes acelerar e incrementar el aprendizaje, tanto de los fundamentos del diseño digital con FPGA como de habilidades en la programación en Verilog, un lenguaje ampliamente utilizado en la industria para configurar circuitos digitales. De este modo, los conocimientos adquiridos serán útiles tanto en el entorno académico como en el profesional, al dotar a los estudiantes de herramientas valiosas para destacar en sus prácticas y en sus futuros empleos.

Esperamos que este libro sea tanto un recurso académico, pero también un puente entre el aprendizaje y el mundo profesional. Al dominar los conceptos y habilidades que aquí se presentan, los lectores podrán llevar sus conocimientos a la práctica de manera efectiva, contribuyendo a proyectos que reflejen tanto su creatividad como su capacidad técnica.

Bienvenidos a este viaje de aprendizaje y descubrimiento en el apasionante mundo de las FPGA.

Loreto Marín Carcey
Directora del Departamento de Electrotecnia e Informática
Universidad Técnica Federico Santa María

AGRADECIMIENTOS

Queremos expresar nuestra gratitud en primer lugar a Loreto Marín por su apoyo y entusiasmo en promover la publicación de este libro, además de colaborar en su revisión, junto a nuestros colegas Guelis Montenegro y Víctor Cárdenas, quienes aportaron valiosas sugerencias y acertados comentarios que permitieron finalizar de la mejor manera este texto.

A nuestros alumnos, que probaron cada una de las experiencias prácticas y pudieron comentarlas, en especial a César Villar, Luis Roa y Ariki Atan, quienes se motivaron para ir más allá e investigar soluciones más complejas con la tarjeta BASYS 3.

Vaya a todos ustedes, y a nuestros futuros lectores, nuestro agradecimiento.

INTRODUCCIÓN

Este libro ha sido escrito con el propósito de proveer un medio de referencia a estudiantes y otras personas vinculadas a la elaboración de circuitos digitales respecto de qué son los arreglos de compuertas programables en campo (FPGA) y cómo se utilizan. Su alcance abarca los aspectos conceptuales y prácticos fundamentales que permiten al lector utilizarlos con competencia para diseñar e implementar, de manera flexible y personalizada, circuitos digitales con distintos propósitos, desde circuitos combinacionales básicos hasta circuitos secuenciales, de control y automatización de alta complejidad, así como funciones lógicas, contadores, multiplexores, registros de desplazamiento, procesadores, interfaces de comunicación, controladores de periféricos, lectura de sensores o acciones sobre actuadores.

La descripción es sucinta, simple y directa, pero con todo lo necesario para que quienes se están iniciando en el conocimiento de los sistemas digitales puedan crear y ensayar distintas soluciones utilizando, en este caso, la tarjeta BASYS-3 de Digilent, la cual contiene la FPGA Artix-7 de Xilinx, versión XC7A35TICPG236C-1L. Para esto, se incluye un conjunto de experiencias prácticas de complejidad creciente, para que el lector pueda comprender y aplicar paso a paso los conceptos y técnicas contenidas en el libro.

La idea es que estas experiencias, tipo laboratorio, guíen a los estudiantes respecto de cómo utilizar la FPGA y los recursos de la tarjeta BASYS-3 que la contiene, de manera que puedan configurar con autonomía sus propios circuitos a medida que avanzan en la lectura. Un paso ineludible para lograr este propósito es aprender el lenguaje de descripción de hardware (HDL) Verilog, el cual se requiere para programar la tarjeta BASYS-3 con el circuito deseado. Este tiene su propio apartado en el libro con todos los elementos para comprender qué significa programar hardware, a diferencia de programar software, y por qué soluciones de este tipo son tan importantes cuando se trata de operaciones en tiempo real con alta velocidad de respuesta.

Los aspectos teóricos relativos al funcionamiento de los componentes elementales de los sistemas digitales como compuertas, relojes, *flip flops*, memorias, contadores, *displays*, multiplexores y *latches* no son tratados en este libro, por lo que es necesario que el lector los estudie como requisito previo, así como los relativos a la conversión análoga a digital, bases numéricas y comunicaciones digitales.

CAPÍTULO 1

QUÉ ES UNA FPGA

1.1. Introducción

Un arreglo de compuertas programable en campo (*field programable gate array,* FPGA) es un chip semiconductor programable que contiene miles de celdas lógicas con componentes digitales como compuertas *and, or, not,* multiplexores, *flip flops* y memorias, dispuestos en bloques lógicos programables desconectados entre sí, pero disponibles para ser configurados en uno o múltiples circuitos personalizados según requiera el usuario, vía elementos de interconexión y registros programables internos (figura 1.1). La FPGA está inserta en una tarjeta electrónica del tipo sistema embebido que contiene, además, recursos como *switches*, botones, leds, puertas de entrada/salida y reloj, entre otros, según el fabricante y versión, que complementan su potencial para interconectarse vía sus bloques de entrada/salida con distintos elementos periféricos, acorde con el objetivo del sistema a desarrollar.

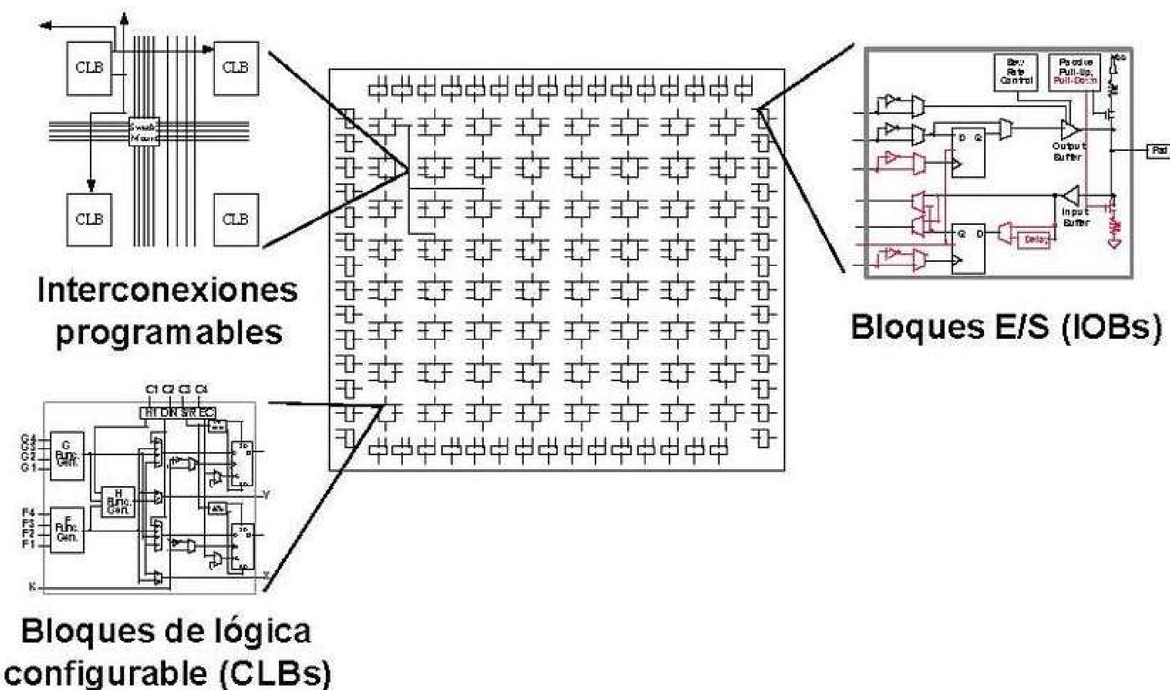


Figura 1.1
Arquitectura general de una FPGA.
Fuente: Digilent (2019).

En particular, la FPGA XC7A35TICPG236C-1L de Xilinx contiene más de 33.000 celdas lógicas. La configuración de los circuitos se realiza de manera transparente para el usuario uniendo los componentes de los bloques mediante elementos de interconexión programables, a través de un programa escrito con un lenguaje orientado a la programación de hardware (lenguaje de descripción de hardware, HDL). Para el caso de esta FPGA, se utiliza el lenguaje Verilog, el cual se estandarizó como IEEE 1364 en 1995. A diferencia de un lenguaje orientado a la programación de software como Python, C++ o similares, los cuales ejecutan secuencialmente cada instrucción mientras «corre» el programa, un lenguaje orientado a la programación de hardware se ejecuta solo una vez para configurar los elementos de hardware del circuito; es decir, para conectar internamente los distintos componentes que se requieren para dar forma a un circuito en particular, de manera similar a como se hace uniendo físicamente componentes con

cables en una *protoboard,* pero esta vez al interior de la FPGA, según las instrucciones del programa. Este programa se escribe en un computador usando una aplicación de desarrollo provista por el fabricante Xilinx, llamada Vivado, para el caso de esta FPGA, después de lo cual debe ser traspasado a la tarjeta propiamente tal. El usuario no necesita saber cuáles de los miles de componentes de la FPGA se utilizan para configurar su circuito; solo sabe que son del tipo que cumple con el programa escrito en el lenguaje Verilog, y es el sistema operativo de la tarjeta, en interacción con la FPGA, el que se encarga de «armar» el circuito bajo la lógica de dicho programa. Una vez finalizada esta tarea, el circuito funciona de manera autónoma bajo la lógica circuital configurada.

1.2. Bloques lógicos configurables

Son la unidad básica de construcción dentro de una FPGA y están diseñados para ser configurables de acuerdo con las necesidades del usuario. Los bloques están compuestos por una matriz de elementos lógicos básicos (compuertas *and, or, xor* y *not*), multiplexores, *flip flops* y *buffers,* entre otros, que pueden ser interconectados y configurados para implementar funciones lógicas específicas.

1.3. Interconexiones programables

Las interconexiones programables son las estructuras que permiten interconectar los bloques lógicos y recursos dentro de la FPGA. Son una característica esencial de las FPGA que les proporcionan flexibilidad y capacidad de adaptación para implementar una amplia variedad de circuitos digitales. Estas interconexiones permiten establecer rutas de comunicación entre los diferentes elementos, como bloques lógicos, *flip flops,* entradas/salidas y bloques de memoria. Se implementan mediante una matriz de interconexión interna que consta de una serie de cables o líneas de conexión y una red de interruptores o multiplexores. Estos cables y multiplexores pueden ser configurados mediante un proceso de programación basado en un HDL, junto a funciones de síntesis e implementación de la plataforma que soporta dicho lenguaje. A continuación, se listan las características clave de las interconexiones programables en una FPGA.

1.3.1. FLEXIBILIDAD

Las interconexiones programables ofrecen una amplia variedad de posibilidades para la conexión de los diferentes bloques y recursos dentro de la FPGA. Esto permite a los diseñadores implementar circuitos digitales con muchos propósitos distintos y reconfigurarlos tantas veces como sea necesario durante el proceso de diseño y posterior rediseño o adecuación.

1.3.2. CONFIGURABILIDAD

Las rutas de interconexión en una FPGA pueden ser programadas y reconfiguradas para adaptarse a los requisitos específicos del diseño. Esto se logra mediante la programación de los multiplexores y los interruptores en la matriz de interconexión.

1.3.3. ALTO NIVEL DE INTEGRACIÓN

Las interconexiones programables permiten variadas posibilidades de utilización de los componentes de la FPGA, lo que significa que múltiples bloques lógicos, *flip flops* y otros recursos pueden ser interconectados de manera eficiente para formar muchos circuitos complejos distintos en una sola FPGA.

1.3.4. OPTIMIZACIÓN DE RECURSOS

Los diseñadores pueden optimizar las interconexiones dentro de la FPGA para minimizar la latencia, el consumo de energía y otros parámetros de rendimiento según las necesidades del diseño.

1.4. Bloques de entrada/salida (E/S)

Los bloques de E/S son los componentes de la FPGA que le permiten la comunicación bidireccional con su entorno (sensores, actuadores, o cualquier otro dispositivo electrónico o componente externo), proporcionando flexibilidad y capacidad de adaptación para una amplia gama de diseños. A continuación, se describen algunas de las características comunes de los bloques de E/S.

1.4.1. COMPATIBILIDAD DE ESTÁNDARES

Los bloques de E/S pueden ser compatibles con una variedad de estándares eléctricos, como *low-voltage differential signaling* (LVDS), *complementary metal-oxide-semiconductor* (CMOS) o *transistor-transistor logic* (TTL), entre otros.

1.4.2. CANTIDAD DE PINES

Las FPGA ofrecen una variedad de bloques de E/S con diferentes cantidades de pines para adaptarse a distintas necesidades de diseño. Algunas ofrecen bloques de E/S dedicados para señales de alta velocidad, mientras que otros están diseñados para propósitos generales.

1.4.3. CARACTERÍSTICAS ELÉCTRICAS

Los bloques de E/S tienen características eléctricas específicas como voltajes de entrada y salida, niveles de corriente e impedancias. Estos bloques pueden ser configurados según las especificaciones del diseño.

1.4.4. RECURSOS DE INTERFAZ ESPECÍFICOS

Algunas FPGA incluyen bloques de E/S con recursos de interfaz específicos, como controladores de memoria DDR, controladores Ethernet o controladores PCI, entre otros. Estos recursos adicionales facilitan la integración de la FPGA en sistemas más complejos.

1.4.5. FUNCIONES DE PROPÓSITO ESPECIAL

Las FPGA suelen incluir bloques de E/S con funciones de propósito especial, como *digital-to-analog converters* (DAC), *analog-to-digital converters* (ADC) y *buffers* de transceptor seriales, para admitir aplicaciones específicas que requieren conversión de señales analógicas y comunicaciones digitales de alta velocidad, entre otras.

1.5. Generalidades sobre cómo se trabaja con una FPGA

En el capítulo 3 se describe en detalle cómo usar, en particular, la tarjeta BASYS-3 con la FPGA versión XC7A35TICPG236C-1L. En adelante, esta combinación se identifica como BASYS-3/FPGA o simplemente como «la tarjeta», con algunas excepciones para destacar características específicas de alguno de estos elementos, debido a que ambos funcionan de manera integrada.

Para formarse una idea preliminar, se indican en esta sección los pasos relevantes a seguir para concebir un proyecto. No es el propósito que lector los domine y aplique en propiedad con solo esta

descripción general, pero sí que le ayuden a comprender los fundamentos de cómo se trabaja con una FPGA, según se explica más adelante en el libro.

1.5.1. ETAPAS

1. Abrir Vivado. El usuario debe descargar esta plataforma desde el sitio del fabricante. Es en Vivado donde se desarrollan todas las tareas para programar e implementar un circuito.
2. Crear un proyecto y un archivo con el nombre que identifica el circuito.
3. Programar el circuito con el lenguaje Verilog.
4. Obtener un esquemático del circuito (opcional).
5. Asignar al circuito recursos específicos de la tarjeta (leds, *switches*, puertos de E/S, botones, reloj).
6. Aplicar las funciones de síntesis, implementación y *bitstream* para validar y estructurar el código final que debe ser traspasado a la tarjeta.
7. Traspasar el código resultante a la tarjeta.
8. Comenzar a operar con el circuito diseñado.

1.6. Ejemplo inicial: Circuitos que implementan una compuerta *and* y una *or*

Como una primera aproximación, se incluye este ejemplo básico y resumido de configuración de una compuerta *and* y una *or,* con el propósito de reafirmar de forma gradual la comprensión del lector respecto de lo expuesto en este capítulo. Los detalles en extenso se describen en el capítulo 3.

1.6.1. INTERCONEXIÓN

La figura 1.2 muestra el esquema de interconexión entre el computador y la BASYS-3/FPGA.

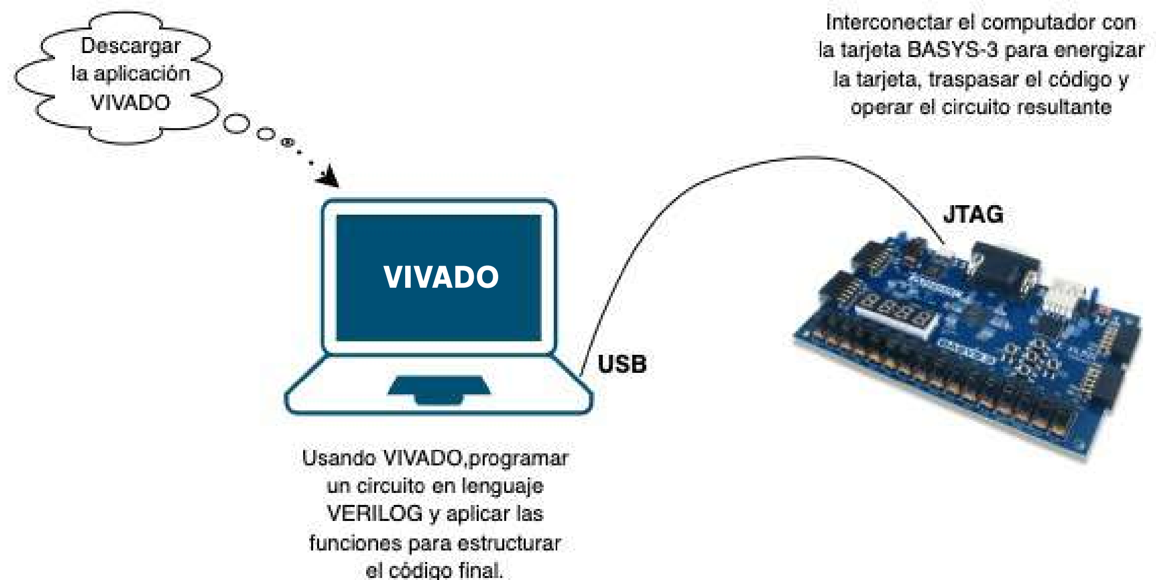


Figura 1.2
Esquema de interconexión básica.

1.6.2. RECURSOS DE LA TARJETA BASYS-3/FPGA PARA CONFIGURAR LOS CIRCUITOS

La figura 1.3 muestra las compuertas *and* y *or* de la FPGA y los *switches* y leds de la tarjeta BASYS-3 que se usan en este ejemplo inicial. Cabe destacar que estos componentes están inicialmente desconectados entre sí dentro de la tarjeta, pero disponibles para su utilización.

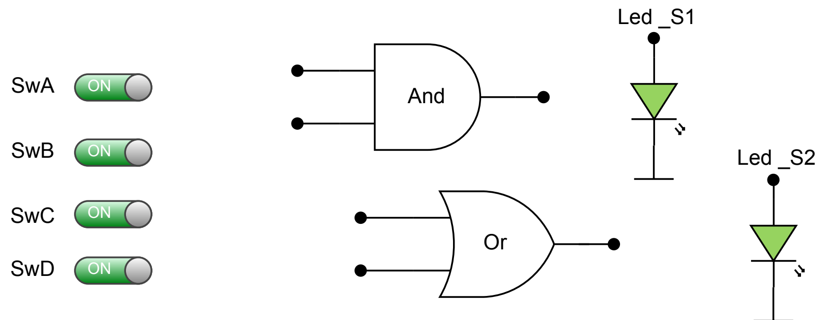


Figura 1.3
Componentes para configurar un primer circuito.

1.6.3. PROGRAMA DESARROLLADO CON VERILOG

Se entiende que el lector ha descargado e instalado Vivado en su computador y ha creado un proyecto y archivo para referenciar su circuito, según se indicó en el punto 1.5. El programa se elabora en la ventana «Sources» de Vivado (ver capítulo 3).

```
module compuertas_and_or(        // Nombre del módulo de programa.
    input SwA, SwB, SwC, SwD,    // Son las entradas a las compuertas "and" y "or".
    output Led_S1, Led_S2);      // Son las salidas de las compuertas "and" y "or".
    assign Led_S1 = SwA & SwB;   // Led_S1 es la resultante de un "and" entre SwA y SwB.
    assign Led_S2 = SwC | SwD;   // Led_S2 es la resultante de un "or" entre SwC y SwD.
endmodule
```

1.6.4. ASIGNACIÓN DE RECURSOS DE LA TARJETA A LAS ENTRADAS Y SALIDAS DE LAS COMPUERTAS

Se desarrolla en Vivado configurando el archivo «Basys-3-Master.xdc» (ver capítulo 3). Las entradas a la FPGA son 4 *switches* y las salidas 2 leds de la tarjeta BASYS-3. Los leds tienen asociadas por defecto resistencias internas que no necesitan ser asignadas explícitamente como recursos de la tarjeta.

```
## Switches
set_property -dict {PACKAGE_PIN V17    IOSTANDARD LVCMOS33 } [get_ports {SwA}]
set_property -dict {PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [get_ports {SwB}]
set_property -dict {PACKAGE_PIN W16    IOSTANDARD LVCMOS33 } [get_ports {SwC}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {SwD}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {SW4}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {SW5}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {Led_S1}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {Led_S2}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {Led2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {Led3}]
```

1.6.5. CIRCUITOS RESULTANTES

La figura 1.4 muestra los dos circuitos resultantes de la programación de la sección 1.6.3 y de la asignación de recursos de la sección 1.6.4, una vez ejecutadas las funciones de síntesis, implementación y generación de *bitstream* en Vivado, según se indica en la sección 2.4. Ambos circuitos funcionan de manera paralela e independiente con sus propios recursos.

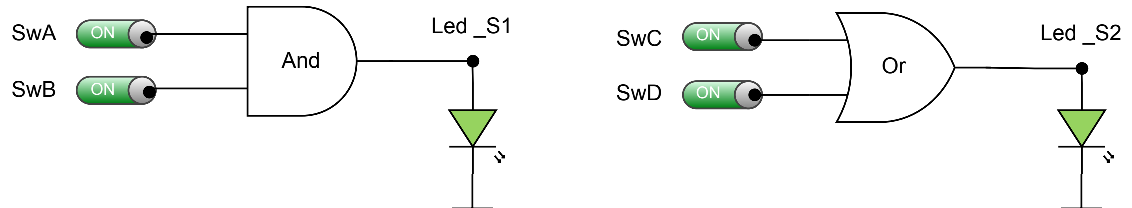


Figura 1.4
Circuitos *and* y *or* resultantes del programa del punto 1.6.3.

Es importante reiterar que en este ejemplo el programa *compuertas_and_or* configura dos circuitos distintos: una compuerta *and* y una compuerta *or*. El programa se ejecuta una sola vez, después de lo cual ambos circuitos quedan configurados en la FPGA y en adelante pueden funcionar cada uno con su propia lógica digital, pero de manera simultánea, independiente e instantánea. Es decir, ambos circuitos están presentes a la vez en la FPGA, operando cada uno con sus propias variables de entrada/salida y respondiendo instantáneamente a su propia función, sin tener que esperar que avance una secuencia de instrucciones como es el caso de una programación de software tradicional. Esto significa que, para este ejemplo, solo cuando se accionan los *switches* SwA y SwB de la tarjeta se enciende el Led_S1 (es la condición de funcionamiento de una compuerta *and*) y cuando se acciona uno o ambos *switches* SwC y SwD se enciende el Led_S2 (es la condición de funcionamiento de una compuerta *or*).

CAPÍTULO 2

DESCRIPCIÓN DE LA TARJETA BASYS-3

2.1. Generalidades

La tarjeta BASYS-3 fue desarrollada por el fabricante Diligent para el diseño e implementación de circuitos digitales, incorporando entre sus componentes la FPGA XC7A35TICPG236C-1L de Xilinx, un oscilador de cristal de 100 MHz, varios puertos multifuncionales (USB, VGA, E/S, A/D), 16 leds, 4 *displays* de 7 segmentos, 16 *switches* y 5 botones, para permitir una amplia variedad de proyectos. Una vista de la tarjeta se presenta en la figura 2.1.

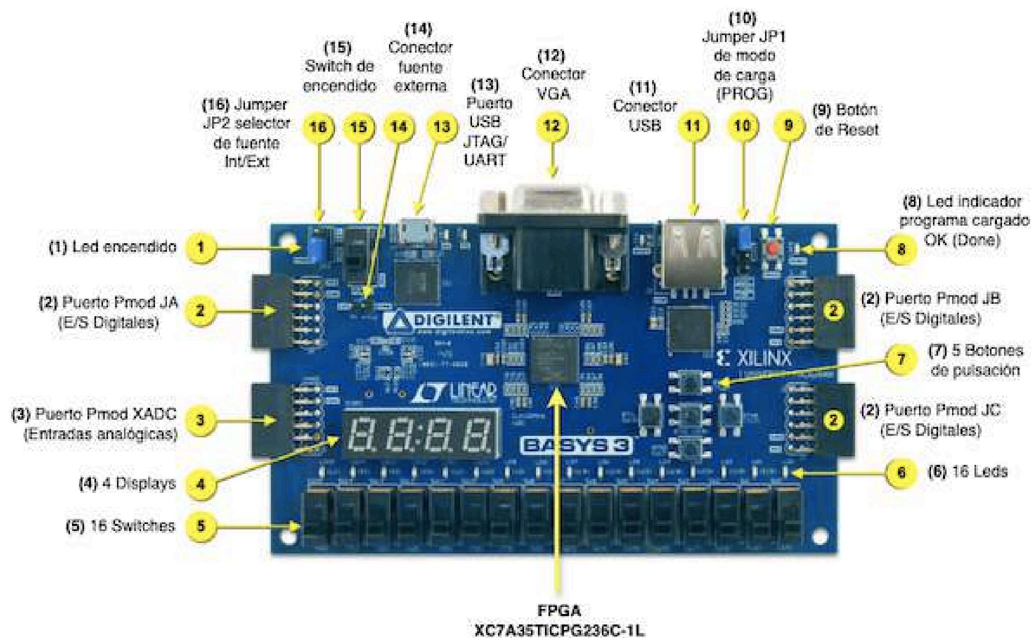


Figura 2.1
Tarjeta BASYS-3 de Digilent.
Fuente: Diligent (2019).

2.2. Recursos de la tarjeta

Los recursos para ser utilizados por los usuarios están disponibles en el archivo denominado «Basys-3-Master.xdc» de la plataforma Vivado. La lista de los recursos contenidos en dicho archivo se presenta en el Anexo A.2. En el capítulo 3 se presenta cómo utilizar Vivado y los recursos de la tarjeta BASYS-3/FPGA propiamente tal.

Siguiendo la numeración de la figura 2.1, se describen a continuación los componentes de la tarjeta.

1. LED INDICADOR DE ENCENDIDO

Cambia a color rojo cuando se acciona el *switch* de encendido de la tarjeta (ver punto 15).

2. PUERTOS PMOD JA, JB Y JC

Puertos configurables como entrada o salida digitales que interactúan con el ambiente externo a la tarjeta. Cada uno posee 8 pines de entrada o salida que trabajan con nivel alto (3,3 volts DC) o bajo (0 volt), más 2 pines de suministro de alimentación de 3,3 volts DC y 2 pines de tierra. La figura 2.2.

muestra la disposición de pines de cada puerto, los cuales se declaran en la programación asociados a variables del tipo input/output según se requiera.

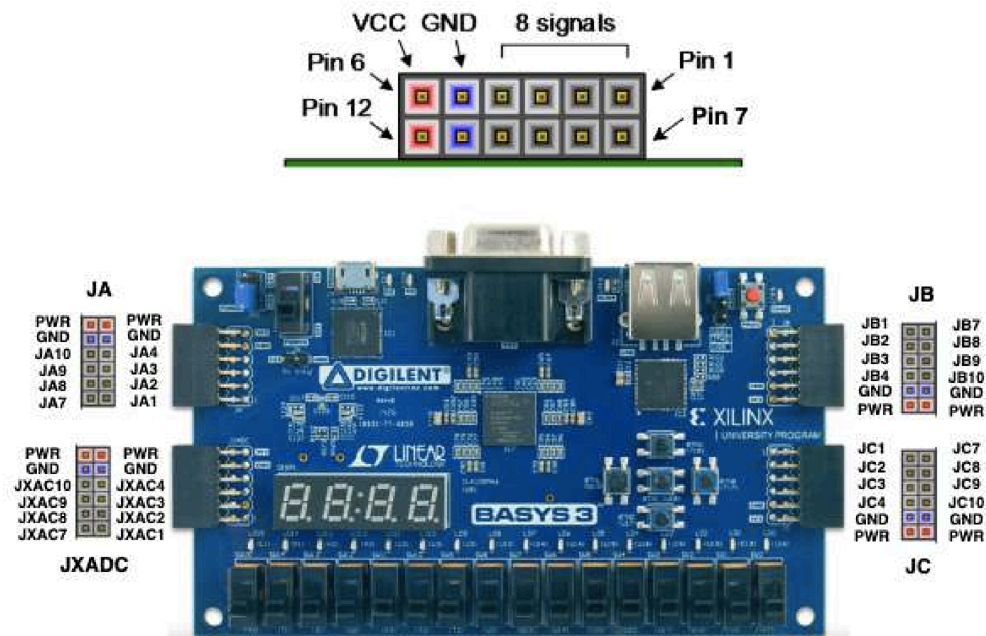


Figura 2.2
Identificación de los pines de las puertas PMod.
Fuente: Diligent (2019)

3. PUERTO PMOD XADC

Puerto de conversión a formato digital, con 8 pines que proveen 4 canales de conversión para entradas analógicas entre 0 y 1 volt desde el ambiente externo, más 2 pines para suministro de alimentación de 3,3 volts y 2 pines de tierra.

4. *DISPLAYS*

Arreglo de 4 *displays* de 7 segmentos más un punto, que se habilitan con ánodo común. Se emplean como variables tipo *output* en la programación de los circuitos.

5. *SWITCHES*

Un total de 16 *switches* dispuestos de modo que en la posición normal (hacia abajo en la figura 2.1) proveen un nivel bajo (0 volt) y al accionarlo un nivel alto (3,3 volts). Se utilizan como variables del tipo *input* en la programación de los circuitos.

6. LEDS

Un total de 16 leds que pueden ser utilizados como elementos de visualización de las salidas de los circuitos. Se utilizan asociados a variables del tipo *output* en la programación de los circuitos.

7. BOTONES DE PULSACIÓN

Son 5 botones que al pulsarlos proveen un nivel alto (3,3 volts) y un nivel bajo (0 volt) en la posición normal. Se utilizan como variables tipo *input* en la programación de los circuitos.

8. LED INDICADOR DE PROGRAMA CARGADO OK (*DONE*)

Se enciende de color verde cuando el programa se ha traspasado correctamente desde un computador a la tarjeta.

9. BOTÓN DE *RESET*

Al aplicar el *reset* se reinicia la tarjeta y se borra la configuración del circuito cuando se ha utilizado el modo de carga JTAG (ver punto 10). En este caso, para recuperarlo, se requiere volver a cargar el programa. Existe otra opción que reinicia el circuito sin necesidad de volver a cargar su configuración (ver sección 3.5).

10. *JUMPER* SELECTOR DEL MODO DE CARGA (JP1)

Este selector tiene tres posiciones de funcionamiento según el método de traspaso del programa a la tarjeta y tipo de almacenamiento empleado (ver sección 3.5):

- En la posición JTAG (posición central), el programa se recibe por el puerto JTAG (ver punto 13) y se guarda en una memoria volátil, por lo que deja de estar disponible si se suspende la energía, se aplica un *reset* o si se traspasa un nuevo programa a la tarjeta (lo sobreescribe). Por lo tanto, para recuperar un programa es necesario volver a cargarlo.

- En la posición SPI *Flash* (QSPI), el programa también se recibe por el puerto JTAG, pero esta vez se almacena en una memoria no volátil. Esto hace posible que el programa se conserve ante cortes de energía o *reset*.

- En la posición USB, el programa puede ser cargado desde un *pendrive* por el puerto USB.

11. PUERTO USB

Puerto para funciones USB estándar.

12. PUERTO VGA

Puerto para funciones de salida VGA de 12 pines. Se utilizan como variables tipo *output* en la programación de un circuito.

13. PUERTO USB TIPO MICRO-B JTAG/UART

Puerto JTAG para cargar el programa a la FPGA, funciones de comunicación UART y alimentación de la tarjeta.

14. CONECTOR DE FUENTE EXTERNA

Este es el punto de conexión de una fuente de alimentación externa (entre 4,5 y 5,5 volts DC, 1 amperio mínimo). Para utilizarlo se requiere cambiar la posición del *jumper* JP2 a EXT (ver punto 16).

15. *SWITCH* DE ENCENDIDO

Switch para aplicar energía a la tarjeta. Esta se alimenta con un voltaje de entrada de 5 volts DC a través de un cable USB tipo Micro-B (ver punto 13) o por una fuente externa alternativa (ver punto 14). Los reguladores internos transforman el voltaje de entrada a 3,3 y 1,0 volts DC, los que son utilizados por la FPGA y otros componentes. La corriente máxima que proporciona la tarjeta es de 500 mA, dependiendo de los elementos periféricos conectados. El consumo de potencia depende de la complejidad del diseño cargado en la FPGA, pero normalmente es inferior a 2,5 watts.

16. *JUMPER* SELECTOR DE FUENTE DE INT/EXT (JP2)

Permite seleccionar si la alimentación de la tarjeta es a través de la interfaz USB (posicionar el JP2 en USB) o por medio de otra fuente externa (posicionar el JP2 en EXT).

Adicionalmente, la tarjeta BASYS-3 contiene:

- 1 oscilador de cristal de 100 MHz que proporciona una fuente de reloj estable para los diseños implementados en la FPGA.
- 256 MB de memoria DDR3.
- 16 MB de memoria *flash* SPI para almacenamiento no volátil.
- Memoria EEPROM para almacenamiento de configuraciones.

CAPÍTULO 3
CÓMO SE USA LA TARJETA BASYS-3

3.1. Introducción

Para usar la tarjeta BASYS-3 se requiere cargar antes la aplicación Vivado y los *drivers* de la tarjeta en un computador. Esta carga se realiza una sola vez. Luego, utilizando Vivado, se deben ejecutar los cinco pasos generales que se indican en el punto 2 de la figura 3.1.

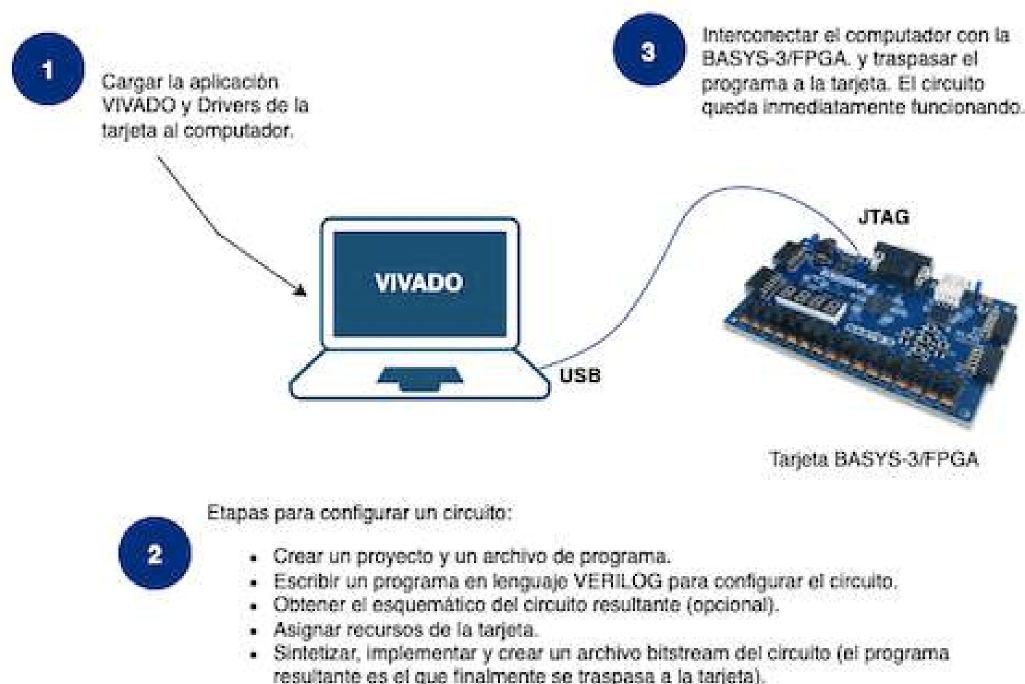


Figura 3.1
Resumen general de cómo usar la tarjeta BASYS-3/FPGA.

Es importante destacar que para cargar Vivado, los *drivers* y ejecutar los pasos hasta la síntesis, implementación y creación del *bitstream,* no se requiere tener físicamente disponible la tarjeta. De hecho, es un requisito trabajar estos puntos y depurar los errores que son normales en cualquier codificación de un programa y en la etapa de asignación de recursos, antes de conectar la tarjeta. Incluso, es posible simular el funcionamiento del circuito para verificar que cumple con lo requerido sin contar con la tarjeta, según se presenta en el capítulo 5, sección 5.1. Los pasos para crear y ejecutar un proyecto, así como el cumplimiento de la sintaxis en la programación de las instrucciones con Verilog que se describen en el capítulo 4, deben llevarse a cabo rigurosamente. Vivado no permite traspasar a la tarjeta un programa con error.

3.2. Carga de Vivado en un computador

Primero, se requiere crear una cuenta Xilinx y luego descargar e instalar Vivado. Para esto, siga el enlace https://www.xilinx.com/support/download.html, ingrese a la página principal del fabricante AMD y siga el procedimiento que se indica:

1. Haga clic sobre los recuadros en rojo de las páginas que van desplegándose sucesivamente en las láminas de la figura 3.2. Esto permite seleccionar el «Software Vivado 2018.3», ir a «Vivado

Design Suite-HLx editions-2018.3 Full Product Installation» y seleccionar «Vivado HLx 2018.3 All OS installer Single-File Download (TAR/GZIP – 18.97 GB)». Aunque existen versiones más actualizadas, se elige la versión 2018.3 por su compatibilidad con las capacidades intermedias de los computadores normalmente disponibles. Estos deben contar, al menos, con un procesador con arquitectura x86-64 de doble núcleo a 2 GHz, un mínimo de 4 GB de memoria RAM y un mínimo de 20 GB de espacio libre en disco duro para la instalación básica. Para esta versión existe compatibilidad con los sistemas operativos Windows 7 y Windows 10, además en el caso de Linux con CentOS 6.6 a 7.3, RHEL 6.6 a 7.3, SUSE Linux Enterprise 12.2, Ubuntu 16.04 LTS. Dependiendo de las características del computador, el proceso de carga de Vivado puede durar varios minutos.

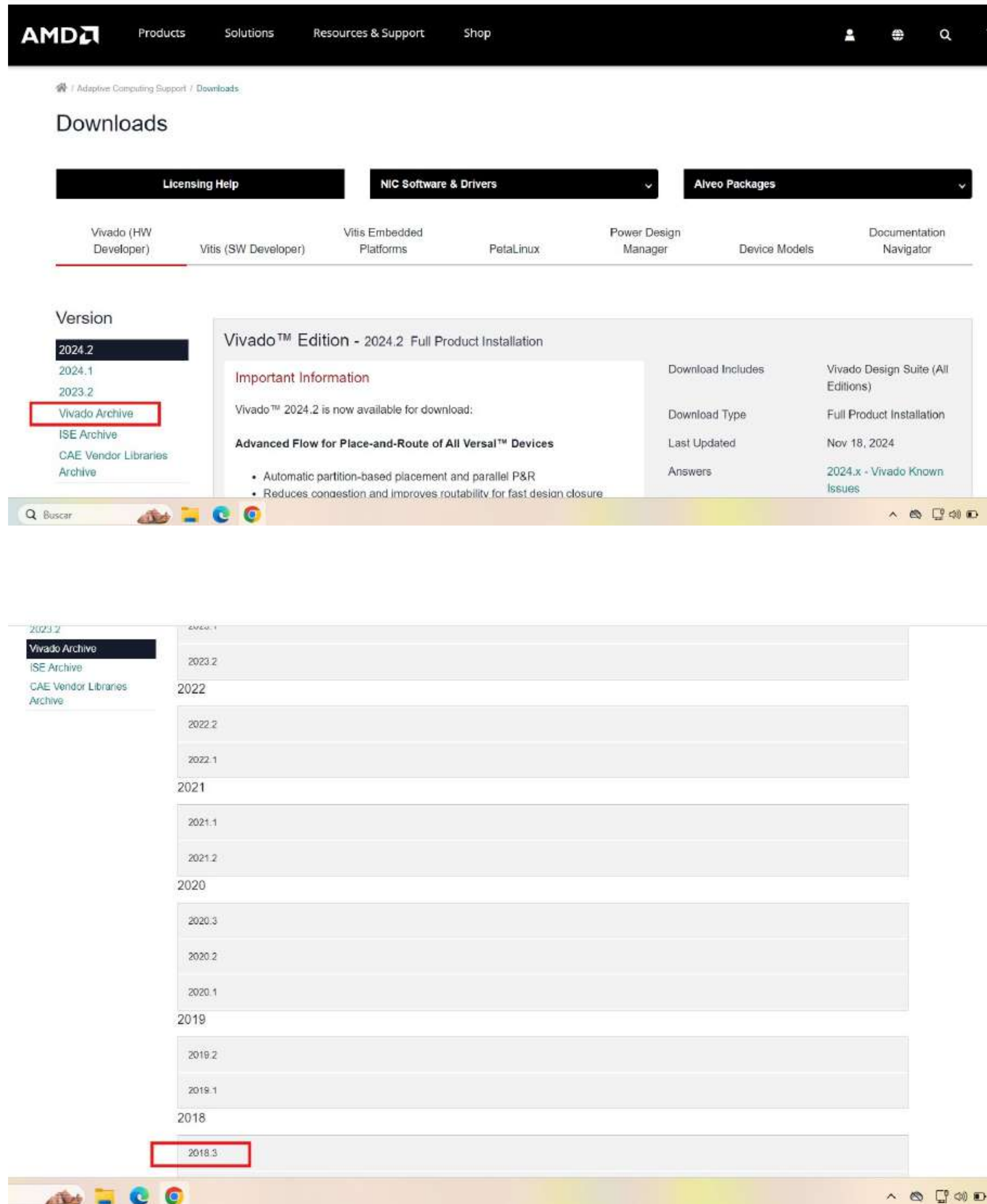
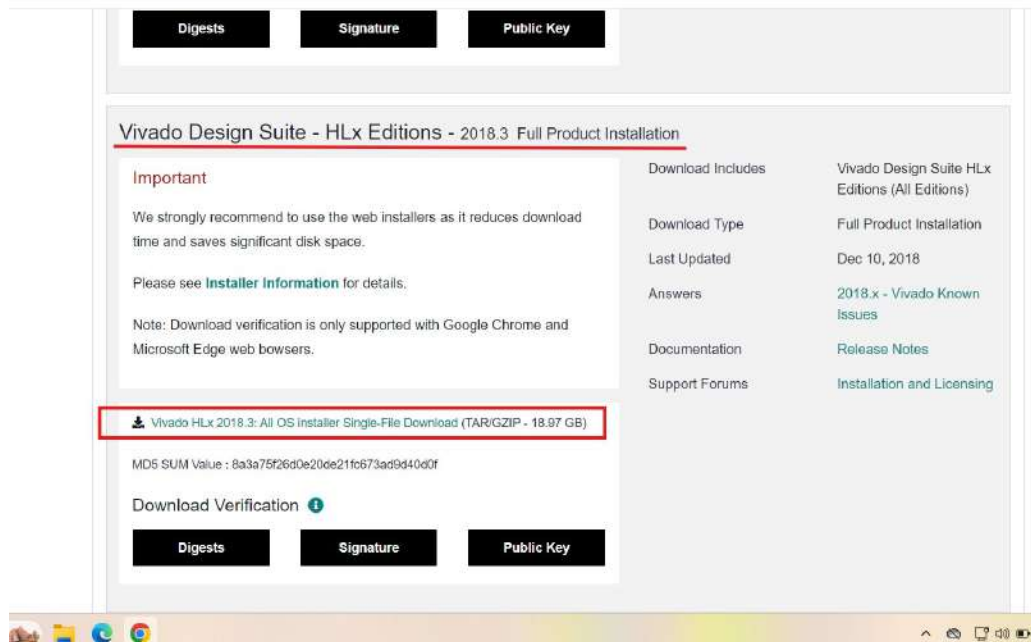
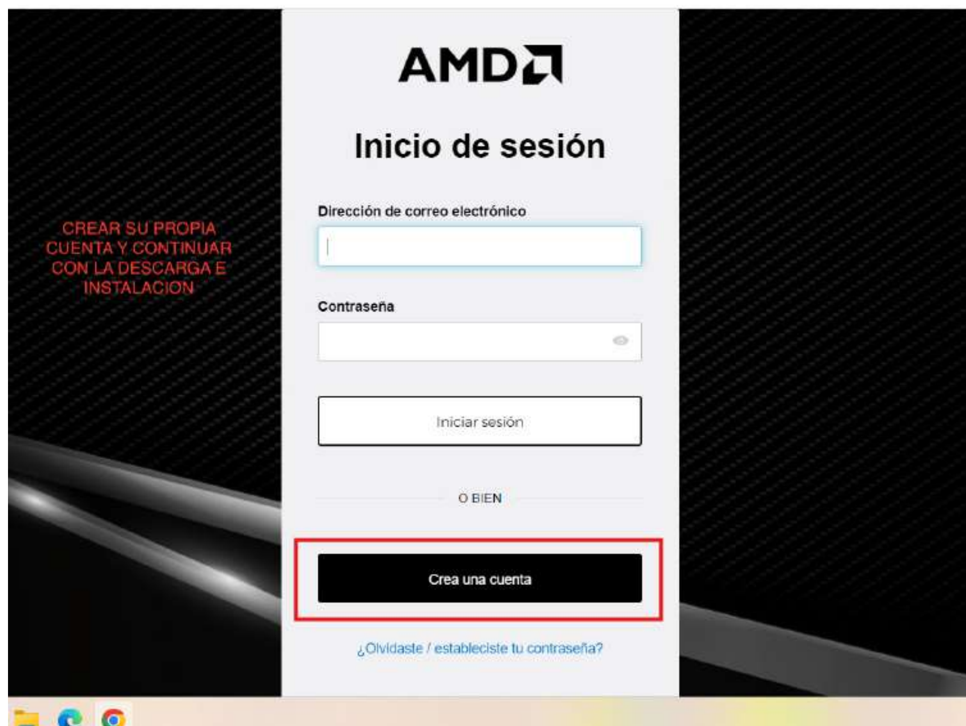


Figura 3.2
Procedimiento de carga de Vivado.





2. Se abre una ventana donde se solicita crear una cuenta de usuario en Xilinx, para lo cual debe ingresar a «Crear una cuenta». Esto es condición para continuar con el proceso de descarga.

3. Una vez completado el registro, debe iniciar la descarga e instalar el programa.

4. Finalmente, reinicie el computador y abra el archivo ejecutable «Vivado 2018.3» desde el escritorio.

3.3. Carga de los drivers de las tarjetas en Vivado

Una vez finalizada la instalación de Vivado, se deben cargar los *drivers* con las distintas versiones de tarjetas del fabricante Digilent, accediendo al enlace: https://github.com/Digilent/digilent-xdc/.

1. Seleccione en la lista de archivos de la izquierda la opción «Basys-3-Master.xcd». Se desplegarán todos los recursos de la tarjeta que se indican en el Anexo A.2.

2. Proceda a la descarga.

3. Los recursos quedan disponibles para los usuarios en una carpeta de Vivado llamado «digilent-xdc-master». Es importante tomar nota de la localización de la carpeta en la que queda almacenado este archivo (path), pues se deberá acceder a este más adelante.

4. Posteriormente, cuando se crea un proyecto y se selecciona la versión específica de la tarjeta a utilizar, los recursos de dicha versión quedan automáticamente disponibles para los usuarios en un archivo llamado «Basys-3-Master.xdc» residente en el directorio «digilent-xdc-master», según se explica en el desarrollo de la sección 3.4.

Con esto, ya puede iniciar el uso de Vivado. Es decir, crear, programar y ejecutar sus proyectos (circuitos).

3.4. Crear, programar y ejecutar un proyecto

Una vez instalado el software Vivado y el archivo de recursos «Basys-3-Master.xdc», se debe proseguir con las siguientes etapas generales para configurar un circuito, las cuales dan lugar a los 28 pasos que se indican desde la sección 3.4.1 a la 3.4.5:

- Crear un proyecto FPGA.

- Escribir un programa en Verilog para configurar un circuito.

- Obtener el esquemático del circuito resultante (opcional).

- Asignar recursos de la tarjeta BASYS-3 al circuito.

- Aplicar las funciones de síntesis, implementación y generación del *bitstream* al circuito. Estas generan el programa final que debe ser traspasado a la tarjeta.

- Traspasar el programa resultante del *biststream* a la tarjeta BASYS-3/FPGA.

En la experiencia con usuarios de Vivado se ha podido comprobar que esta sucesión de 28 pasos se logra dominar, memorizar y aplicar sin inconvenientes con un breve periodo de práctica. Por ejemplo, exceptuando los pasos relativos a diseñar, programar y asignar recursos a un circuito, un usuario que ya ha creado sus dos o tres primeros proyectos, normalmente ejecuta estos pasos en menos de noventa segundos. Más bien, el tiempo depende de las características del computador que de las acciones sobre Vivado. No obstante, dado que este procedimiento es clave para diseñar e implementar un circuito, se recomienda tenerlo a mano para consultas en las primeras prácticas (se sugiere imprimir el resumen incluido en el Anexo A.1).

3.4.1. CREAR UN PROYECTO

Consiste en crear un proyecto y asignarle un nombre, crear un archivo que identifica al programa principal, seleccionar la tarjeta y la versión de la FPGA a utilizar. Para esto se deben realizar los siguientes pasos:

1. Abra Vivado. Se despliega la ventana de la figura 3.3.

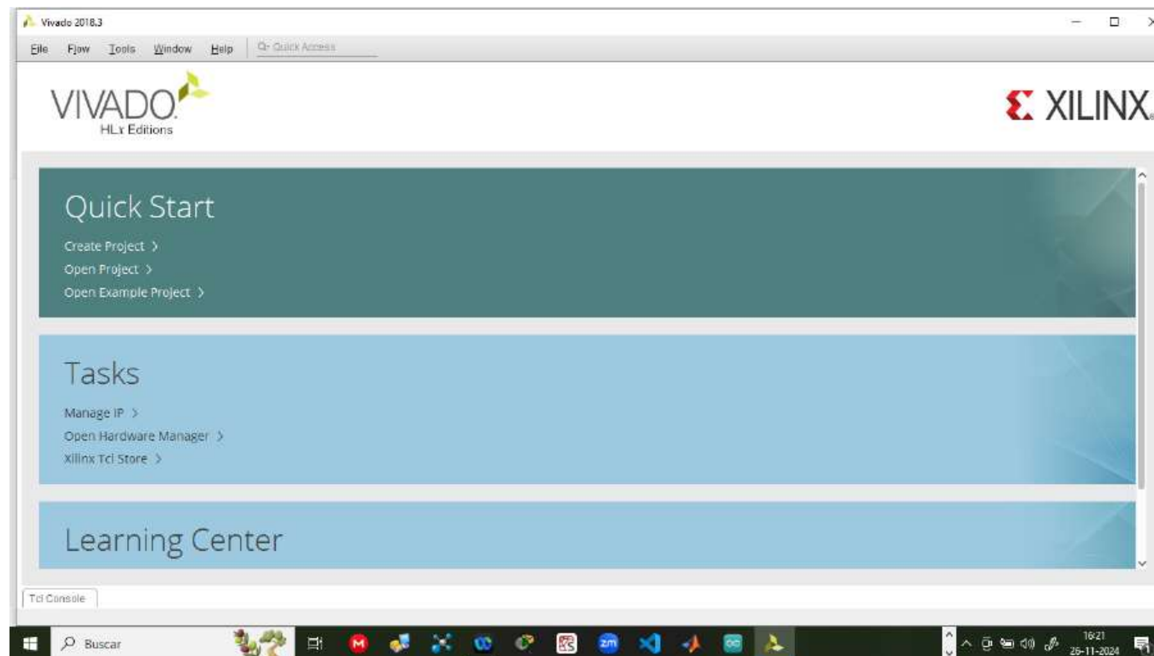


Figura 3.3
Ventana de inicio de Vivado.

A continuación, aplique las funciones de Vivado para ejecutar las acciones señaladas desde los puntos 2 al 28. En cada paso se hace referencia a los elementos que van apareciendo de manera sucesiva en cada una de las ventanas que se activan.

2. En «Quick Start», inicie la creación de un proyecto con «Create Project» y aplique «Next».

3. En la ventana «Project Name», defina el nombre del proyecto y su localización en «Project Location» (para este ejemplo lo llamamos «Compuerta_01»). Verifique que esté marcada la opción «Create Project Subdirectory» y aplique «Next». El nombre del proyecto no debe tener espacios, acentuación u otros caracteres especiales. Como separador, se puede usar guion bajo de la forma *xx_yy_zz*. Es importante tomar nota del *path* donde queda guardada la carpeta que archiva el proyecto, ya que se deberá acceder a esta más adelante.

4. Revise que en la ventana «Project Type» esté marcada la opción «RTL Project» y aplique «Next».

5. En la ventana «Add Sources», aplique «Create File». En la nueva ventana que se abre indique, en el campo «File Name», el nombre del programa principal a generar (para este ejemplo el archivo se denomina «compuerta_nand»). El nombre no debe tener espacios, acentos ni caracteres especiales, excepto el guion bajo. Luego, aplique «OK».

6. En la ventana «Add Sources» que se abre, marque la nueva línea que tiene el nombre del archivo recién definido y aplique «Next».

7. En la nueva ventana «Add Constraints» que se abre, marque «Add Files» y busque en el campo Recent Directories la localización del directorio «digilent-xdc-master», (ver sección 3.3).

8. Marque dos veces sobre el directorio «digilent-xdc-master» para desplegar todos los tipos de

tarjetas con las que trabaja Vivado.

9. Seleccione, en este caso, el archivo «Basys-3-Master.xdc», y aplique «OK».

10. Verifique que en la ventana «Add Constraints» que se abre esté seleccionado «Copy Constraints Files Into Project». Esto permite recuperar el archivo «Basys-3-Master.xdc» con los recursos asignados cuando se requiera volver a abrir el proyecto en el futuro. Aplicar «Next».

11. En la ventana «Default Part» que se abre, use el buscador «Search» para localizar y seleccionar la versión de la FPGA con que se trabaja (está rotulado en el chip FPGA y en la caja de almacenamiento de la tarjeta). En este caso es «XC7A35TICPG236-1L». Aplique «Next».

12. Aparece una ventana con el resumen del proyecto. Aplique «Finish» y espere que se cree el proyecto y aparezca la ventana «Define Module». En el campo «Module Name» aparece el nombre del programa especificado anteriormente en el punto 5. Aplique «OK» y luego «Yes». Espere a que se despliegue en la parte central superior la ventana «Sources», con la misma identificación del programa principal, pero con extensión .v, en la carpeta «Desing Sources». En este caso, «compuerta_nand.v», como se muestra en la figura 3.4. Esta es la ventana de trabajo de Vivado para configurar el circuito bajo diseño. Nótese que en el área de la izquierda están las funciones de Vivado que se referencian en los pasos que siguen, como también las áreas de trabajo para escribir el programa, asignar recursos de la tarjeta y las otras funciones con que Vivado apoya los diseños de circuitos.

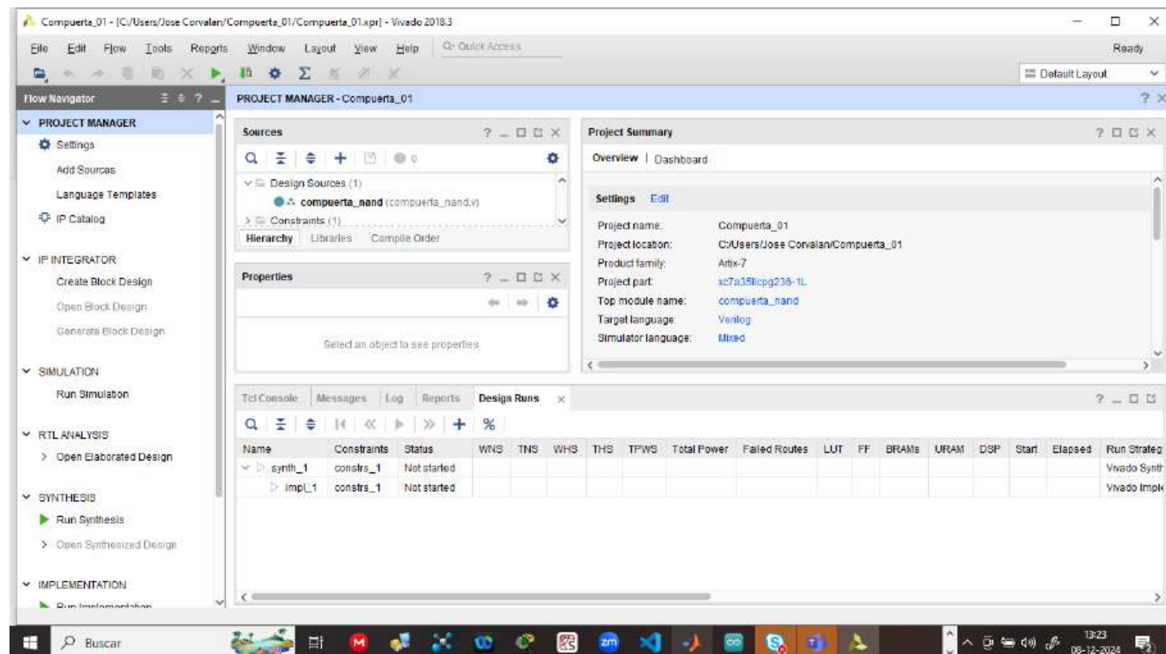


Figura 3.4
Ventana de trabajo de Vivado.

3.4.2. ESCRIBIR UN PROGRAMA EN VERILOG PARA CONFIGURAR UN CIRCUITO

En esta etapa se programa la serie de instrucciones que permiten configurar un circuito cualquiera bajo las reglas y sintaxis del lenguaje Verilog (ver las instrucciones de Verilog en el capítulo 4). Como ejemplo, se programa aquí una compuerta nand.

13. Marque dos veces sobre el nombre del programa con extensión .v en el área «Sources». Se abre una ventana a la derecha con la estructura inicial del programa a desarrollar, con el nombre y los primeros elementos de programación de Verilog. Puede agrandar el área de la ventana para tener mayor facilidad de trabajo.

14. Escriba las siguientes instrucciones en lenguaje Verilog para configurar una compuerta *nand* (ver el capítulo 4 para mayor comprensión de la estructura, instrucciones y sintaxis del lenguaje Verilog):

```
module compuerta_nand(    // Nombre del módulo de programa.
    input A, B,           // A y B son las entradas a una compuerta and (switches).
    output C );           // C es la salida de la compuerta nand (aplicada a un led).

    assign C = ~(A & B); // C es la negación de la and entre A y B (nand).

endmodule
```

Notar que un cuadrado en la parte superior izquierda se pone en «verde» cuando la sintaxis es correcta y en «rojo» en caso contrario, además de subrayar en «rojo» la línea de instrucción errónea (puede ser la línea subrayada, la inmediatamente superior o el propio bloque de programa). Tener presente que entre los errores más comunes están omitir las «comas» o «puntos y comas» al final de cada línea, poner «coma» cuando debe ser «punto y coma», o viceversa, repetir u omitir los «paréntesis», entre otros. Adicionalmente, considerar que si la línea de instrucción está con un «fondo azul» significa que existe una inconsistencia en la lógica de programación (por ejemplo, asignar un valor a una variable no declarada previamente), o si está con «fondo amarillo» significa una alerta de una posible inconsistencia asociada a alguna variable.

15. Como un paso opcional, Vivado permite ver de manera gráfica el circuito resultante para verificación visual del usuario. En la sección «RTL ANALISYS» del área de la izquierda de la ventana, marque «Open Elaborated Design», aplique «Save» si se le solicita y «OK» en la ventana «Elaborate Design» que se abre. Espere a que se abra el esquemático en la ventana «Schematic» y verifique la consistencia del circuito (si se despliega la ventana «Critical Messages», solo presione «OK»). Para el ejemplo del paso 14, el esquemático resultante es el que se muestra en la figura 3.5. En los casos en que la resolución no sea suficiente, puede hacer *zoom* utilizando las funcionalidades «+/−» de la parte superior de la ventana «Schematic». También, tenga presente que cada vez que se modifique un programa y se desee observar el nuevo esquemático, se deben salvar los cambios haciendo clic en el ícono «Guardar» de la ventana «Sources» y actualizar el esquemático aplicando «Reload» en la línea amarilla superior que se abre.

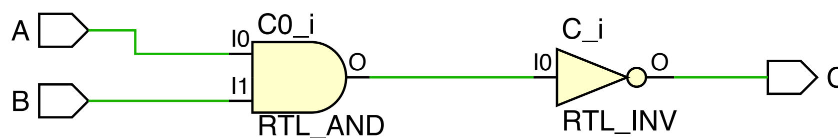


Figura 3.5 Esquemático resultante del circuito *nand*.

3.4.3. ASIGNAR RECURSOS DE LA TARJETA BASYS-3/FPGA AL CIRCUITO

Consiste en asignar componentes de la tarjeta al circuito bajo diseño. En este ejemplo, se requieren dos *switches* para las entradas A y B y un led para la salida C. Esto se logra accediendo al archivo «Basys-3-Master.xdc», que contiene todos los recursos de la BASYS-3, y luego desmarcando las líneas de los componentes requeridos y renombrándolos al final de la línea:

16. Haga doble clic en la carpeta «Constraints» de la ventana «Sources».

17. Haga doble clic en la identificación del archivo en uso («Basys-3-Master.xdc», en este caso). A la derecha se despliega una ventana con el conjunto de recursos que tiene esta tarjeta, como muestra la figura 3.6. Puede agrandar el área de la ventana para tener mejor visualización.

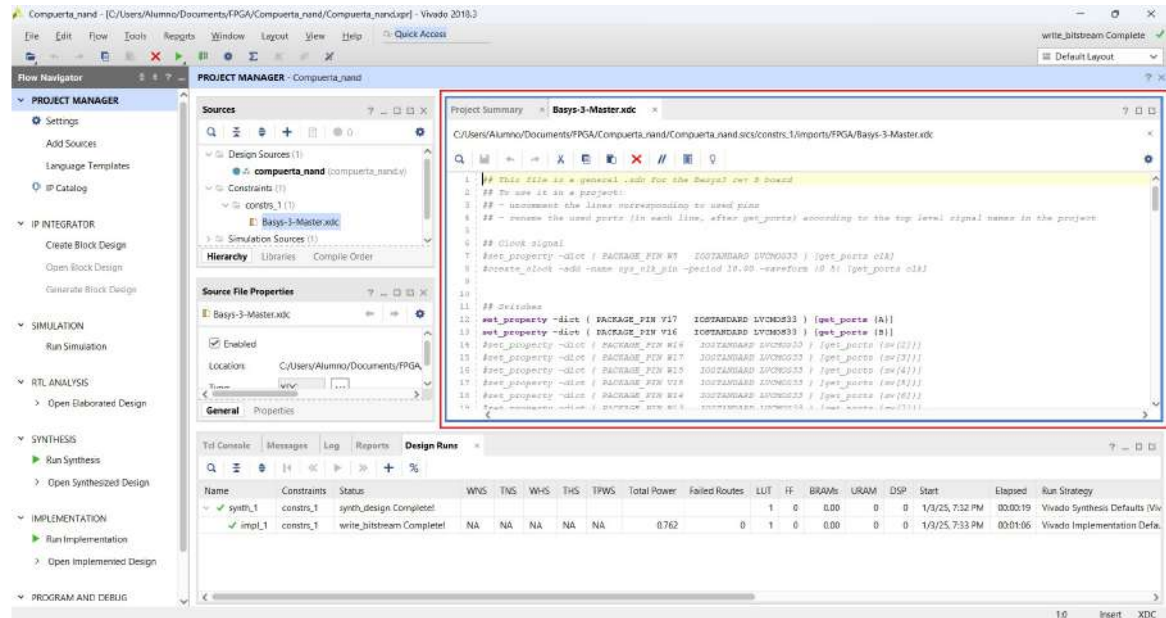


Figura 3.6
Ventana de trabajo para asignar recursos de la tarjeta al circuito.

18. Quite el símbolo «#» al comienzo de las líneas asociadas a los recursos que requiere el circuito y cambie la identificación al final de la línea (el nombre que está en los paréntesis de llaves) por los nombres especificados para cada entrada y salida del programa del paso 14. Note que al quitar el símbolo «#» la línea cambia de color; también, que las entradas A y B han sido asignadas en este ejemplo a los *switches* rotulados como V17 y V16 de la tarjeta y la salida C al led rotulado como U16 (comprobar viendo las identificaciones en la propia tarjeta). Puede escoger los recursos que estime, pero respetando el tipo y cantidad de variables del programa. En este caso deben ser 2 *switches* (los *switches* se usan como entradas y, por lo tanto, están asociados a variables del tipo *input*) y 1 led (los leds se usan asociados a variables de salida del tipo *output*).

```
## Switches (entradas)
set_property -dict {PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {A}]
set_property -dict {PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {B}]
#set_property -dict {PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {sw2}]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw3}]

## LEDs (salidas)
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {C}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]
```

No debe modificar la estructura de las líneas, solo quitar el símbolo # al comienzo de la línea y reemplazar el nombre de la variable al final. Cualquier otra alteración produce un error en las etapas siguientes. Por ejemplo, un error común es quitarle los paréntesis de llaves a las variables o el paréntesis cuadrado al final de la línea. Por otra parte, si se requiere agregar un comentario a la línea, debe hacerse después del símbolo «;» y luego con «#».

3.4.4. SINTETIZAR E IMPLEMENTAR EL CIRCUITO

Aunque no es lo mismo, las actividades que se detallan en adelante se pueden asimilar a la compilación en una programación de software tradicional. En esta etapa, Vivado procesa todos los elementos necesarios para verificar la correcta conformación estructural del programa, antes de generar el código que finalmente se traspasa a la tarjeta. Opcionalmente, si el lector ya ha ganado dominio en

la programación, puede pasar directamente a la sección 3.4.5, dado que la generación del *bitstream* obliga a una ejecución integrada con la síntesis e implementación, si estas no han sido realizadas previamente o si están desactualizadas.

19. En la sección «SYNTHESIS» de la ventana izquierda, marque «Run Synthesis», «Save» si se solicita y «OK» en la ventana «Launch Runs» que se abre. Espere a que la síntesis se complete (ver arriba a la derecha el ícono que indica que la síntesis está en progreso). Cuando concluye, se abre una ventana que indica «Synthesis Successfully Completed»(ver «View Messages» y «Messages» para analizar los errores).

20. En la sección «IMPLEMENTATION» de la ventana izquierda (o en la misma ventana anterior que señala el término de la síntesis) marque «Run Implementation» y «OK» en la ventana «Launch Runs» que se abre. Espere que la implementación se complete (ver arriba a la derecha el ícono que indica que la implementación está en progreso). Cuando concluye, se abre una ventana que indica «Implementation Successfully Completed» (ver «View Messages» y «Messages» para analizar los errores). Opcionalmente, marque «Open Implemented Design» y «OK» para ver la estructura interna que configura la FPGA al implementar el circuito.

3.4.5. GENERAR EL *BITSTREAM* Y TRASPASAR EL PROGRAMA A LA TARJETA BASYS-3/FPGA

Una vez realizada la síntesis e implementación, Vivado transforma el programa a un formato con extensión .bit por medio de un proceso conocido como *bitstream*. Este archivo es el que finalmente se traspasa a la tarjeta. Tenga en consideración que cada vez que se modifique el código del programa (punto 14) se deben actualizar la síntesis, implementación y *bitstream* (se deben rehacer) y que, al finalizar la carga, el circuito queda inmediatamente funcionando como fue diseñado.

21. Dentro de la sección «PROGRAM AND DEBUG» de la ventana izquierda marcar «Generate Bitstream» y «OK» en la ventana «Launch Runs» que se abre. Espere a que la generación del *bitstream* se complete (ver arriba a la derecha el ícono que indica que el *bitstream* está en progreso). Cuando concluye, se abre una ventana que indica «Bitstream Generation Successfully Completed» (ver «View Messages» y «Messages» para analizar los errores).

22. Verifique que el *jumper* selector de modo de carga de la tarjeta está en la posición central JTAG (componente 10 de la figura 2.1). Este modo de carga no conserva la programación cuando se apaga la tarjeta o cuando se aplica un *reset*, lo que implica que en estos casos se debe cargar nuevamente el archivo *bitstream*. En en la sección 3.5 se indica un modo de carga que conserva la programación.

23. Conecte el computador a la tarjeta en la puerta USB/JTAG/UART (componente 13 de la figura 2.1) y active el *switch* de encendido (componente 15 de la figura 2.1). Observe que el led de encendido cambia a rojo y se escuchan los tonos audibles de reconocimiento de conexión del computador.

24. Dentro de la sección «PROGRAM AND DEBUG», aplique «Open Hardware Manager», luego «Open Target» en la barra verde superior y finalmente «Auto Connect».

25. Aplique «Program Device» en la barra verde superior y busque el directorio en donde se encuentra localizado el proyecto (use los «tres puntos» del campo «*Bitsream file*»; el *path* aparece en la línea superior de la ventana del trasfondo). Haga clic en el nombre asignado al proyecto en el paso 3 de la sección 3.4.1.

26. Seleccione nombre del *proyecto.runs* e *impl_1*. Con esta acción, aparece el archivo *xx.bit* a descargar a la tarjeta.

27. Haga doble clic sobre el nombre del *archivo.bit*. Con esta acción se autocompleta el *path* del archivo en el campo «Bitstream File».

28. Aplique «Program», espere a que se traspase el programa y verifique que se ilumina en verde el led «Done» de la tarjeta (componente 8 de la figura 2.1). Esto indica que la carga fue realizada correctamente y que el programa ya está funcionando.

Ahora puede accionar los *switches* correspondientes a las entradas A y B para comprobar el funcionamiento de la compuerta *nand*. De acuerdo con su tabla de verdad, solo cuando ambos *switches* estén en «1» (deslizados hacia el centro de la tarjeta) la salida C pasa al nivel «bajo», apagando el led C (figura 3.10).

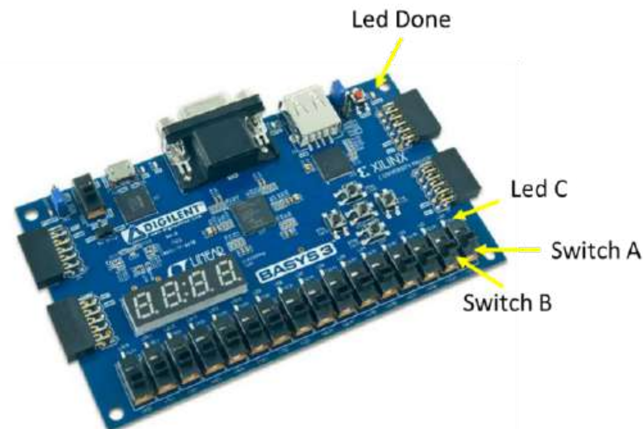


Figura 3.10
Ubicación de los recursos utilizados en el ejemplo.

3.5. Selección del modo de carga

El modo de establecer la carga del programa en la tarjeta dice relación con la opción de que el circuito permanezca o no configurado una vez que se desconecte y aplique nuevamente la energía o se ejecute un *reset*, además del puerto mediante el cual se traspasa el programa a la tarjeta. Esto se controla con la posición del *jumper* JP1, que es el selector del modo de carga indicado con el número 10 en la figura 2.1. Existen tres modos de carga: modo normal (JTAG), modo QSPI y modo USB.

3.5.1. MODO NORMAL (JTAG)

Se configura con el *jumper* JP1 en la posición JTAG. La conexión a la computadora se realiza por el puerto USB JTAG/UART (componente 13 de la figura 2.1). El procedimiento descrito en la sección 3.4.5 se corresponde con este modo de carga. En este caso, el circuito configurado deja de funcionar si se desconecta y reconecta el cable USB del computador (es decir, si se quita y se vuelve aplicar energía a la tarjeta) o si se ejecuta un *reset*. De ser este el caso, se debe volver a aplicar el paso 25 de la sección 3.4.5 para recargar el programa (aplicar «Program Device» y «Program»).

3.5.2. MODO QSPI

Este modo de carga permite mantener en funcionamiento el circuito al restablecer la energía de la tarjeta o después de un *reset*. Para aplicar este modo, ejecute los siguientes pasos:

1. Ubique el *jumper* de JP1 en la posición QSPI.

2. En Vivado, seleccione «Tools» en la parte superior de la ventana principal, luego «Setting» y «Bitstream». Marque «Bin File» para generar un *stream* del tipo *xx.bin*.

3. Genere el *bitstream* y cargue el programa.

4. Se abre la pestaña «Hardware». En caso de que haya un archivo guardado, aparece debajo de «XADC». Haga clic derecho y aplique «Remove Configuration Memory Device» (figura 3.11). De lo contrario, en el archivo xc7a35t_, haga clic derecho y aplique «Add Configuration Memory Device (figura 3.12).

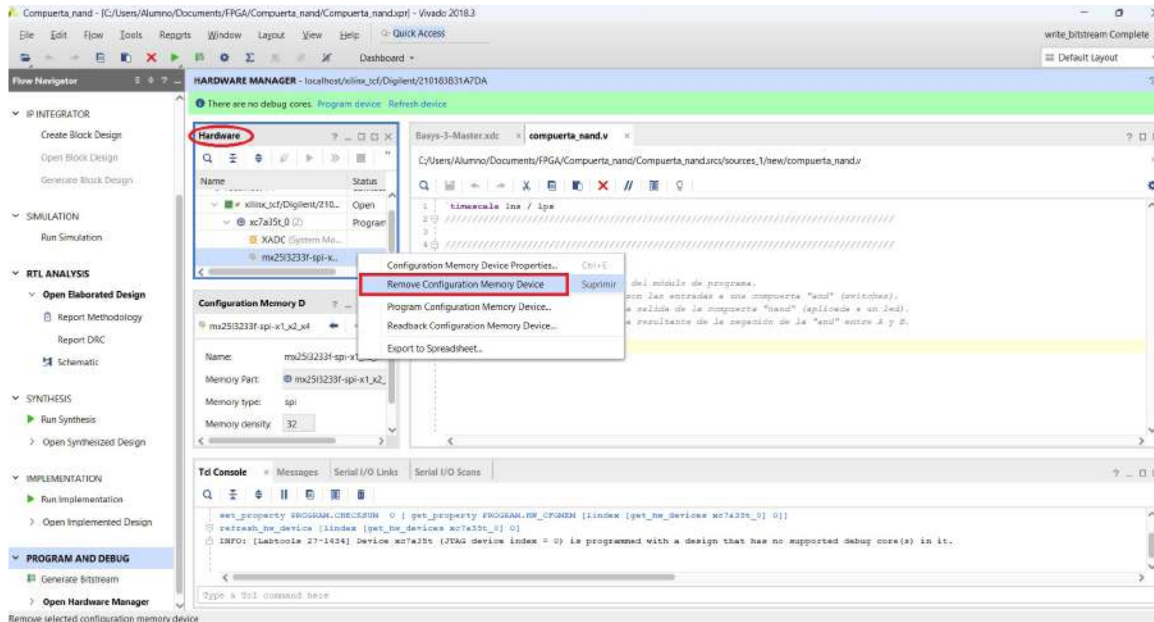


Figura 3.11
Lámina 1 del procedimiento de almacenamiento permanente.

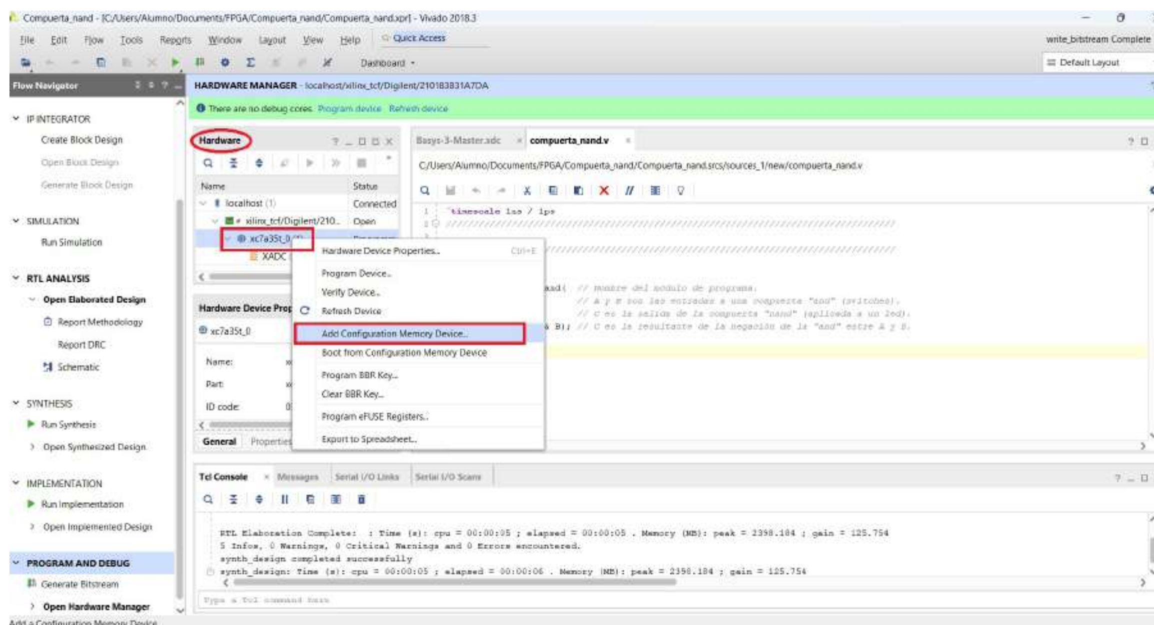


Figura 3.12
Lámina 2 del procedimiento de almacenamiento permanente.

5. Se despliega la ventana «Add Configuration Memory Device». Seleccione «Macronix», «32», «mx25l3233f-spi-x1_x2_x4», según muestra la figura 3.13.

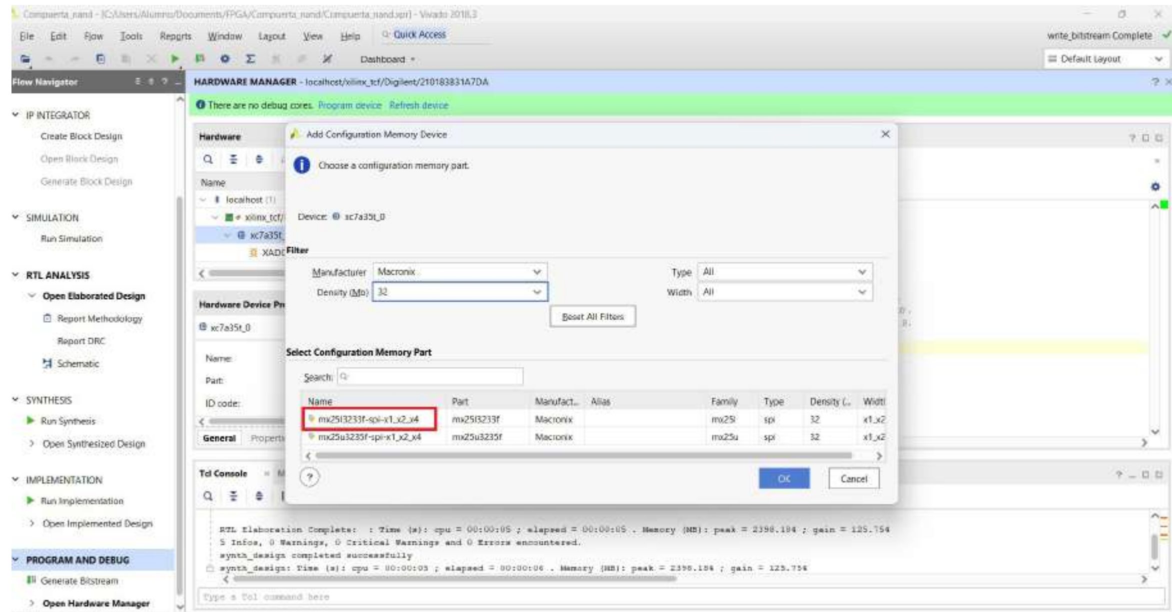


Figura 3.13
Lámina 3 del procedimiento de almacenamiento permanente.

6. En «Program Configuration Memory Device», seleccione el archivo nombre.bin y verifique que estén con marca «Erase», «Program» y «Verify». Luego aplicar «Apply» y finalmente «OK» (figura 3.14).

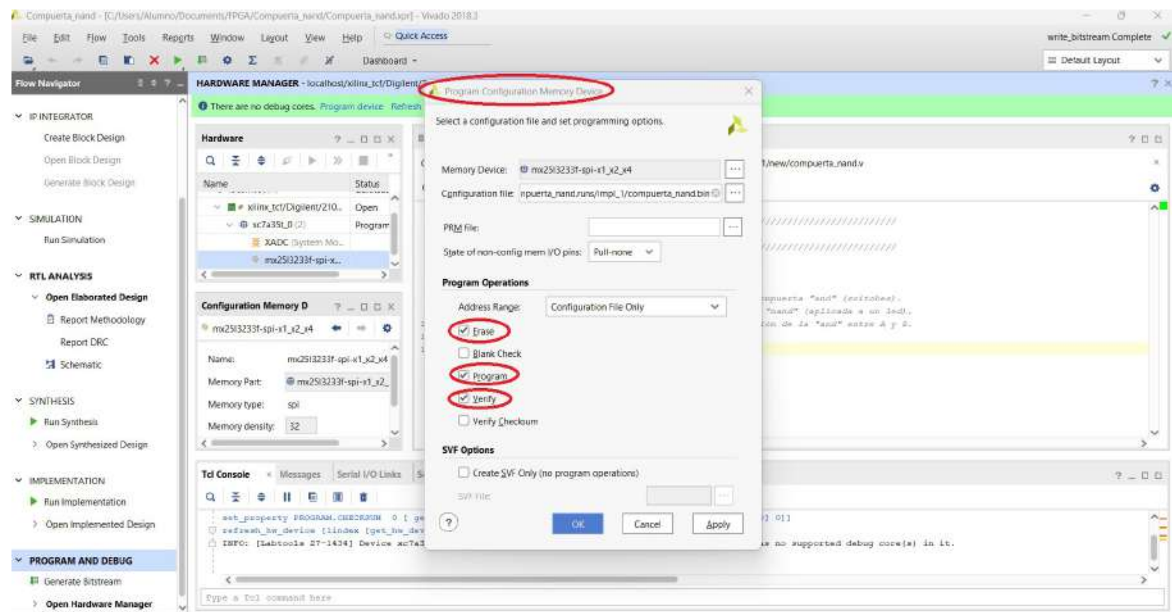


Figura 3.14
Lámina 4 del procedimiento de almacenamiento permanente.

7. Cargue el programa en la tarjeta y aplique el botón de «Reset» (componente 9 de la figura 2.1). Espere unos segundos hasta que se ilumine el led «Done» (componente 8 de la figura 2.1).

3.5.3. MODO USB (CON PENDRIVE)

Se configura con el *jumper* JP1 en la posición USB (componente 10 de la figura 2.1). Este permite el traspaso del programa con extensión *.bit* directamente desde un *pendrive* a la tarjeta. Para esto, ejecute los siguientes pasos:

1. Grabe el archivo de interés con extensión *.bit* en el directorio raíz del *pendrive*. En el directorio raíz del *pendrive* puede haber distintos tipos de archivos, pero solo uno con esta extensión. Este archivo se encuentra en el *path* dado por la carpeta con el nombre del proyecto, subcarpetas *.runs* e *.impl_1* y finalmente el archivo con extensión *.bit* (ver sección 3.6).

2. Conecte el *pendrive* a la tarjeta en el puerto USB.

3. Coloque el *jumper* JP1 en la posición USB.

4. Encienda el *switch* de alimentación y presione el botón de *reset*. Espere unos segundos hasta que se ilumine el led «Done» (componente 8 de la figura 2.1).

3.6. Carpetas de Vivado

Cuando se crea un proyecto en Vivado se genera una carpeta principal y automáticamente varias subcarpetas (cinco o más) con una extensión particular, junto al archivo con extensión *.xpr*. Lo anterior, con el fin de organizar y almacenar los distintos tipos de archivos de diseño, configuración, síntesis, implementación y simulación. Esto permite un acceso rápido al resultado y facilita el proceso de diseño. Vivado organiza estos archivos en carpetas y directorios según el flujo de diseño y las configuraciones del proyecto. El archivo *.xpr* es el principal para acceder al proyecto. Los archivos *.bit* o *.bin* están contenidos en una subcarpeta y son los que contienen el programa final a traspasar a la FPGA.

3.6.1. CARPETA PRINCIPAL DEL PROYECTO

Carpeta raíz del proyecto donde se almacena el principal archivo del proyecto con extensión *.xpr* y todas las subcarpetas que Vivado genera.

3.6.1.1. NOMBRE_DEL_PROYECTO.CACHE

Subcarpeta de caché donde se almacenan datos temporales y sentencias intermedias de la síntesis, implementación y generación de *bitstream*. Esta subcarpeta puede incluir subcarpetas como:

- *ip:* Caché de los núcleos IP utilizados en el proyecto.

- *xilinx.com:* Caché de componentes o IP que provienen de la biblioteca de Xilinx.

3.6.1.2. NOMBRE_DEL_PROYECTO.HW

Esta subcarpeta contiene configuraciones de hardware y archivos asociados al hardware de la FPGA, especialmente si el proyecto está configurado para trabajar con un entorno de diseño específico de hardware o una placa de desarrollo de Xilinx.

3.6.1.3. NOMBRE_DEL_PROYECTO.IP_USERS_FILES:

Subcarpeta donde se guardan los archivos de configuración específicos de los núcleos IP personalizados del Catálogo IP (ver sección 5.3) que se utilizan en el proyecto. Incluye archivos *.xci* y configuraciones XML para cada IP instanciado.

Si un proyecto no utiliza archivos del Catálogo de IP, de igual manera se crea esta subcarpeta, pero sin archivos en su interior.

3.6.1.4. NOMBRE_DEL_PROYECTO.RUNS

Esta subcarpeta almacena los archivos generados durante el proceso de síntesis e implementación:

- synth_1: Contiene los resultados de la síntesis, como *netlists* y *checkpoints* de síntesis (*.dcp*).
- impl_1: Contiene los archivos de implementación, *checkpoints* (*.dcp*) y otros archivos generados durante la implementación.

3.6.1.5. NOMBRE_DEL_PROYECTO.SIM

Contiene archivos de simulación generados durante el proceso de simulación. Estos archivos incluyen:

- Simulación HDL: Archivos de simulación para ModelSim o XSIM.
- Archivos *waveform* (*.wdb, .wvf*): Archivos de formas de onda y resultados de simulación que pueden visualizarse en el simulador integrado.
- Subcarpeta de salida (/xsim): Subcarpeta donde se almacenan los archivos de simulación temporal y registros de simulación.

3.6.1.6. NOMBRE_DEL_PROYECTO.SRCS

Esta subcarpeta contiene todos los archivos fuente del proyecto:

- HDL (*.v, .sv, .vhd*): Archivos de código fuente en Verilog, SystemVerilog o VHDL.
- *Constraints* (*.xdc*): Archivos de restricciones donde se define el mapeo de pines y restricciones de temporización asociados a los recursos asignados de la tarjeta.
- IP *repositories*: Almacena configuraciones de núcleos IP personalizados.
- Diseños de bloques (BD): Si se utiliza IP integrator, los diseños de bloques (*block designs*) se guardan aquí en formato *.bd*.

CAPÍTULO 4
EL LENGUAJE VERILOG

4.1. Introducción

Verilog es un lenguaje descriptivo de hardware (HDL) que permite programar circuitos digitales de diversa complejidad, para luego ser implementados en dispositivos programables, en este caso, en una FPGA.

Se sugiere al lector estudiar este capítulo con la aplicación Vivado ya instalada en su computador y ensayar de inmediato la programación de los ejemplos que se exponen, estudiando detenidamente los comentarios explicativos que acompañan las instrucciones, aunque no disponga físicamente de la tarjeta. A este respecto, según se indicó en el capítulo anterior, con Vivado se puede crear un proyecto, escribir un programa, ver el diagrama esquemático resultante, asignarle recursos de la tarjeta, sintetizarlo, implementarlo y generar el archivo *bitstream* que se traspasa finalmente a la tarjeta, sin la obligación de que la tarjeta esté conectada al computador. Más aún, se puede simular el comportamiento del circuito prescindiendo de la tarjeta, según se describe en el capítulo 5. Obviamente, la tarjeta es indispensable al final del proceso descrito en la sección 3.4, cuando la programación y todas las operaciones anteriores se hayan realizado sin error y sea necesario operar físicamente con el circuito.

Verilog posee la capacidad de describir la estructura física y el comportamiento de diferentes tipos de circuitos digitales, como registros, multiplexores, contadores y controladores entre otros. Se organiza en bloques, que son unidades de código HDL. El bloque de programación principal es el módulo «module», el cual finaliza con «endmodule». La figura 4.1. muestra la estructura básica de un programa Verilog, con tres áreas para distintos propósitos.

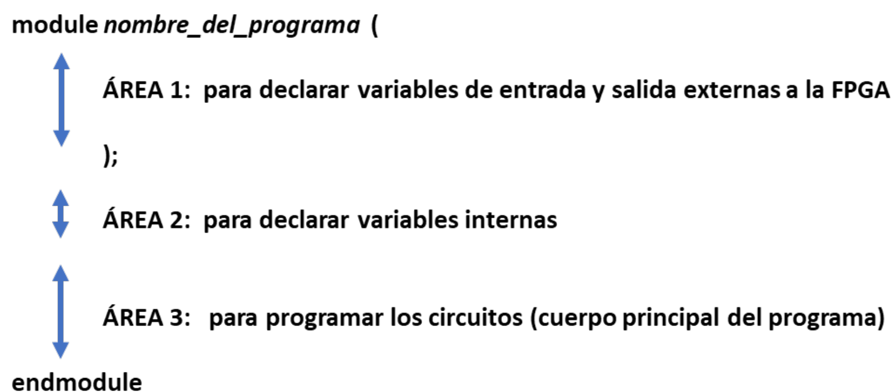


Figura 4.1
Estructura general de un programa en Verilog.

Para comprender de mejor manera esta estructura, se plantea como primer ejemplo la programación del circuito combinacional de la figura 4.2.

Ejemplo 1. Programa para configurar un circuito combinacional con dos compuertas «or», tres variables de entrada y una de salida.

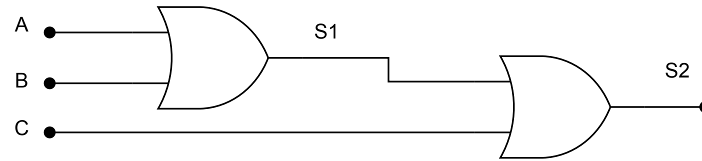


Figura 4.2 Circuito combinacional del ejemplo 1.

```
        module circuito_combinacional_01(   // Nombre del módulo de programa.
Área 1:     input A, B, C,                  // Declara A, B y C como entradas (switches).
            output S2 );                    // Declara S2 como salida (led).

Área 2:     wire S1;                        // Declara S1 como un cable interno.

Área 3:     assign S1 = A | B;              // Asigna a S1 el resultado de la función A or B.
            assign S2 = S1 | C;             // Asigna a S2 el resultado de la función S1 or C.

        endmodule
```

Primeras observaciones y reglas de sintaxis:

- El nombre no puede incluir espacios, acentos (usar guion bajo «_» para conectar nombres).
- Todo bloque debe comenzar con «module» y finalizar con «endmodule» (deben escribirse en minúsculas).
- Los nombres, variables e instrucciones son sensibles a mayúsculas y minúsculas. Por ejemplo, para Verilog, «S1» no es lo mismo que «s1».
- La lista de entradas y salidas deben separarse por comas «,» y debe finalizar con un cierre de paréntesis seguido de punto y coma «);».
- Las instrucciones deben finalizar con punto y coma «;».
- El símbolo «//» se utiliza para agregar comentarios. No es ejecutable.
- Entre símbolos puede o no haber espacios (por ejemplo, «S2 = C | S1» es lo mismo que «S2=-C|S1»).
- Si una salida solo es utilizada internamente, debe ser declarada como «wire» (cable interno) en el área para declarar señales y variables internas. Esta área no tiene ningún delimitador con el área para programar los circuitos.
- Las variables que se declaran como entrada o salida, es decir como «input» y «output», requieren obligatoriamente recursos de la tarjeta.
- A diferencia de las anteriores, una variable «wire» no requiere recursos de la tarjeta. Solo es un cable de conexión interno en la FPGA que el programa genera de manera transparente para el usuario.

4.2. Tipos de descripciones

Para describir un proyecto con Verilog existen distintos procedimientos que se clasifican en descripción funcional, estructural y procedimental. Para explicar estos conceptos se utiliza como ejemplo la programación de las funciones lógicas *or* y *and* en el circuito combinacional que se ilustra en la figura 4.3.

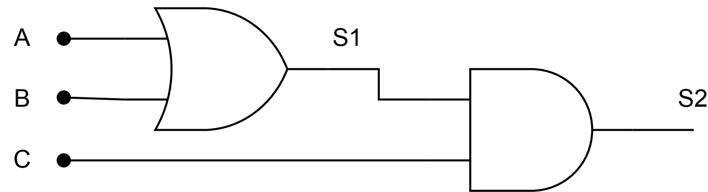


Figura 4.3 Circuito combinacional del ejemplo 2.

4.2.1. DESCRIPCIÓN FUNCIONAL (ASSIGN)

Utiliza la sentencia «assign» para asignar las salidas de manera concurrente y operadores lógicos a nivel de bit como and («&») y or («|»), entre otros.

Ejemplo 2. Solución con descripción funcional para el circuito combinacional de la figura 4.3, que realiza una operación *or* entre A y B con salida en S1 y una operación *and* entre S1 y C con salida en S2 (programar de acuerdo con la sección 3.4.2, paso 14). Nótese el uso de la instrucción «assign» para este tipo de descripción.

```
module ejemplo_2(          // Solución con descripción funcional.
    input A, B, C,         // Declara A, B y C como entradas externas (switches).
    output S2 );           // Declara S2 como salida externa (led).

    wire S1;               // Declara S1 como un cable interno (no ocupa recursos).

    assign S1= A | B;      // Configura una compuerta or entre A y B con salida S1.
    assign S2= S1 & C;     // Configura el circuito combinacional con salida en S2.

endmodule
```

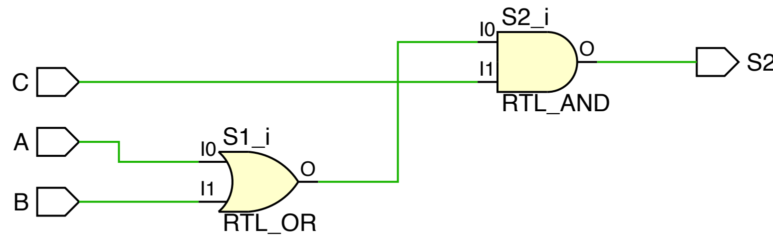
Una posible asignación de recursos es la que se indica a continuación (ver sección 3.4.3, paso 18). Nótese que S1 no tiene recursos asignados, pues es solo un cable interno (*wire*).

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {A}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {B}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {C}]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw[3]}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {S2}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
```

El diagrama esquemático resultante que entrega Vivado es el que se indica en la figura 4.4 (obtener de acuerdo con la sección 3.4.2, paso 15).

Figura 4.4
Esquemático del ejemplo 2.



4.2.2. DESCRIPCIÓN ESTRUCTURAL («AND», «OR», «XOR», «NOT»)

Utiliza módulos predefinidos en Verilog como «and», «nand», «or», «nor», «xor», «xnor» y «not».

Ejemplo 3. Solución con descripción estructural para el mismo circuito combinacional del ejemplo 2.

```
module ejemplo_3(          // Solución con descripción estructural.
    input A, B, C,         // Declara A, B y C como entradas externas (switches).
    output S2 );           // Declara S2 como salida externa (led).

    wire S1;               // Declara S1 como un cable interno (no requiere recursos).

    or (S1, A, B);         // Configura una compuerta or entre A y B con salida en S1.
    and (S2, S1, C);       // Configura el circuito combinacional con salida en S2.

endmodule
```

En los ejemplos 2 y 3 se obtiene el mismo resultado en la salida S2 (puede obtener el esquemático del circuito resultante para comprobarlo). Básicamente, la única diferencia es que en el primer caso Verilog utiliza la instrucción «assign» y operadores lógicos, mientras que en el segundo utiliza los módulos predefinidos «or» y «and» para ejecutar las operaciones lógicas.

4.2.3. DESCRIPCIÓN PROCEDIMENTAL («ALWAYS»)

Es utilizada para describir estructuras de control, facilitando con ellas la creación de funciones complejas. Está basada en la definición de un bloque «always» que va inserto en uno del tipo «module», como el de los ejemplos anteriores. Un bloque «always» se ejecuta solo si cambia de estado (nivel) alguna de las señales de una lista de sensibilidad. Sintácticamente se expresa como:

```
always@ (x, y, z)        // lista de sensibilidad separada por comas.
```

o también:

```
always@ (*)              // si se requiere incluir todas las entradas en la lista.
```

La lista de sensibilidad son las entradas para las cuales se desea que opere el «always» si se produce algún cambio en alguna de ellas (son sólo las entradas).

Consideraciones para el uso de «always»:

- Todas las asignaciones ejecutadas dentro de una instrucción «always» deben ser sobre una señal de salida tipo «reg».

- Una variable del tipo «reg» modela el almacenamiento de datos. Es decir, son variables con memoria.
- Cuando un bloque «always» contiene varias instrucciones debe usarse «begin» al comienzo y «end» al final.
- Dentro de un bloque «always» no se utiliza la instrucción «assign». La asignación a la variable se puede realizar solo con una igualdad, de manera directa, como se muestra en el ejemplo 4.

Ejemplo 4. Solución con descripción procedimental para el mismo circuito combinacional del ejemplo 2.

```
module ejemplo_4(          // Solución con descripción procedimental.
    input A, B, C,         // Declara A, B y C como entradas externas (switches).
    output reg S2 );       // Declara S2 como salida externa, tipo registro, para ser usada
                           // dentro del bloque always (led).
    reg S1;                // Declara S1 como un registro interno, para ser usado dentro del
                           // bloque always (no requiere recursos).
    always@(A, B, C)       // El always opera sólo cuando cambia alguna de las entradas A, B ó C.
    begin                  // Se requiere un begin cuando un always tiene más de una instrucción.
        S1= A | B;         // Configura una compuerta or con entradas A y B, con salida S1.
        S2= S1 & C;        // Configura el circuito combinacional con salida en S2.
    end                    // Cierra el procedimiento always (está asociado al begin).

endmodule
```

Obtenga el esquemático correspondiente y compruebe que los ejemplos 2, 3 y 4 producen el mismo circuito combinacional.

4.3. Arreglos [M:L]

Los arreglos son estructuras de datos que permiten almacenar múltiples valores en una sola variable. Es decir, aquellas variables con más de un dígito. La sintaxis en Verilog es:

```
[ M : L ]        // donde M es el dígito más significativo (MSB) y L el menos
                 // significativo (LSB).
```

Por ejemplo, en:

```
input [ 3 : 0 ] B
```

la variable B tiene 4 bits (B3, B2, B1, B0), que se procesan individualmente como «B[3]», «B[2]», «B[1]» y «B[0]», donde «B[3]» es el MSB y «B[0]» el LSB.

Ejemplo 5. Aplicación con arreglos para obtener el complemento 1 de un número de 4 bits.

- Si A es 1010, el programa devuelve C1 como 0101.
- Si por ejemplo desea acceder al tercer dígito de C1, debe especificarlo como C1[2], que en este ejemplo es «1».

```
module ejemplo_5(          // Ejemplo con arreglos.
    input [3:0] A,         // Declara A como arreglo de entrada de 4 bits (4 switches).
    output [3:0] C1 );     // Declara C1 como arreglo de salida de 4 bits (4 leds).

    assign C1= ~A;         // Asigna a C1 el arreglo A negado (el complemento a 1 de A).

endmodule
```

Se presenta a continuación una posible asignación de recursos para el ejemplo 5. El lector puede escoger la que desee, pero respetando el tipo (los *switches* son *input* y los leds *output*) y cantidad de variables (4 *switches* y 4 leds en este ejemplo), según se indicó en la sección 3.4.3, paso 18.

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {A[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {A[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {A[2]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {A[3]}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {sw[4]}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {sw[5]}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {C1[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {C1[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {C1[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {C1[3]}]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
```

La figura 4.5 muestra el esquemático del circuito resultante elaborado por Vivado. Observe que Vivado identifica de manera explícita los arreglos definidos en el programa.

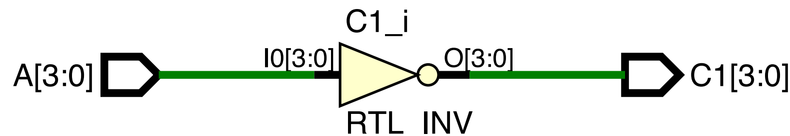


Figura 4.5
Esquemático del ejemplo 5.

4.4. Operadores

Existe una gran variedad de operadores para distintos propósitos. En esta sección se presentan los más utilizados, los cuales son requeridos en los ejemplos que siguen.

4.4.1. OPERADORES LÓGICOS A NIVEL DE BIT

Estos operadores resuelven operaciones lógicas bit a bit. Por ejemplo, si A es «1011» y B es «1101», entonces la operación «A & B» entrega como resultado «1001».

Las sintaxis de los operadores a nivel de bit son las siguientes:

```
And      «&»  : A & B;           // operación A and B.
Or       «|»  : A | B;           // operación A or B.
Not      «~»  : ~B;              // operación negado de B.
Or exc   «^»  : A ^ B;           // operación A or exclusivo B.
Nand     «~&»: ~( A & B );       // operación nand entre A y B.
Nor      «~|» : ~( A | B );      // operación nor entre A y B.
Xnor «~^» : ~( A ^ B );          // operación xnor entre A y B.
```

4.4.2. OPERADORES RELACIONALES

Son operadores que permiten comparar valores y determinar si una relación entre ellos es verdadera o falsa. Por lo tanto, el resultado que producen siempre solo es verdadero (1) o falso (0). El significado y la sintaxis en Verilog son los siguientes:

```
a > b;    // ¿es a mayor que b?.
a < b;    // ¿es a menor que b?.
a <= b;   // ¿es a menor o igual que b?.
a >= b;   // ¿es a mayor o igual que b?.
a == b;   // devuelve «1» si a es igual que b.
a != b;   // devuelve «1» si a es distinto de b.
```

4.4.3. OPERADORES LÓGICOS

Estos operadores difieren con los de nivel de bit (ver sección 4.4.1), ya que solo arrojan como resultado un «1» ó «0».

```
a && b;     // devuelve «1» si a y b son verdaderos.
a || b;     // devuelve «1» si a o b es verdadero.
!a;         // devuelve «1» si a es falso ó «0» si a es verdadero.
```

4.4.4. OPERADORES ARITMÉTICOS

Permiten realizar operaciones del tipo aritméticas. Por ejemplo, si A es «1011» y B es «1101», entonces «A + B» entrega como resultado «11000».

```
Multiplicación   «*»    c = a * b;    // operación a multiplicado por b.
División         «/»    c = a / b;    // operación a dividido en b.
Suma             «+»    c = a + b;    // operación suma de a y b.
Resta            «-»    c = a - b;    // operación resta de a y b.
```

4.4.5. OPERADOR DE PROPÓSITOS ESPECIALES

Este operador es del tipo condicional, ya que devuelve distintos resultados dependiendo del valor de una condición (es una forma especial de «if»):

```
«?»: D = a ? b : c;
```

En este caso, si «a» es «1», entonces D = b; es decir, D toma el valor de «b». A su vez, si «a» es «0», entonces D = c; es decir, D toma el valor de «c».

Ejemplo 6. Programa que obtiene un «1» ó «0» en la salida de 2 actuadores, según el valor de la entrada, usando el operador «?».

```
module ejemplo_6(                     // Ejemplo con operador de propósito especial "?".
    input A,                          // Entrada A de 1 bit (1 switch).
    output Actuador_1,                // Salida Actuador_1 (1 led).
    output Actuador_2 );              // Salida Actuador_2 (1 led).

    assign Actuador_1= (A==1) ? 0:1; // Si A es 1, Actuador_1 es 0. Si A es 0, Actuador_1 es 1.
    assign Actuador_2= (A==1) ? 1:0; // Si A es 1, Actuador_2 es 1. Si A es 0, Actuador_2 es 0.

endmodule
```

4.5. Representación de constantes

Cuando se requiere asignar un valor directo a una variable, este puede ser representado en formato decimal, binario, hexadecimal u octal (entre los más utilizados). La sintaxis en código Verilog es como se indica (<tamaño> es la cantidad mínima de bits que debe tener un registro para contener el <valor>):

```
<tamaño> ' <base> <valor>
```

Como ejemplo, la representación del número 47 en distintas bases, con su correspondiente tamaño de bits:

```
Decimal         x = 6'd47;
Binario         x = 8'b00101111;
Hexadecimal     x = 8'h2F;
Octal           x = 6'o57;
```

Ejemplo 7. Programa que obtiene las salidas en distintas bases.

```
module ejemplo_7(                              // Salidas en base binaria y hexadecimal.
    input A,                                   // Entrada A de 1 bit (1 switch).
    output [8:0] Actuador_1,                   // Salida Actuador_1 de 8 bits.
    output [4:0] Actuador_2 );                 // Salida Actuador_2 de 4 bits.

    assign Actuador_1 = (A==1) ? 8'b11110000 : 8'b00001111;
                                               // Si A es 1, entonces Actuador_1 es 11110000.
                                               // Si A es 0, entonces Actuador_1 es 00001111.

    assign Actuador_2 = (A==1) ? 4'hF : 4'h0;  // Si A es 1, entonces Actuador_2 es 1111.
                                               // Si A es 0, entonces Actuador_2 es 0000.
endmodule
```

4.6. Sentencias condicionales

Estas sentencias establecen una condición para la ejecución de una instrucción o de un bloque del programa. Las sentencias condicionales siempre deben ser contenidas en procedimientos «always» y pueden usarse todos los operadores lógicos y relacionales para especificar la condición.

4.6.1. SENTENCIA «IF... ELSE... »

Esta sentencia responde al diagrama de flujo de la figura 4.6. Si C es verdadero ejecuta el bloque 1, de lo contrario ejecuta el bloque 2.

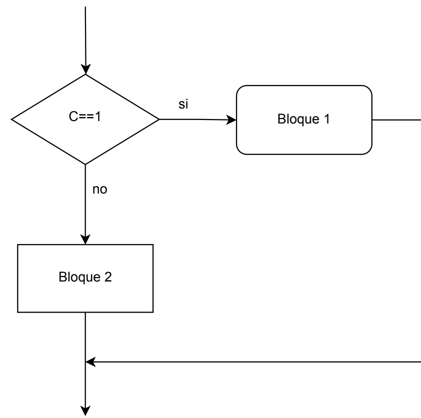


Figura 4.6
Diagrama de flujo de la sentencia «if... else...».

Para demostrar cómo opera esta sentencia, se diseña un programa que resuelve el funcionamiento de un multiplexor de dos entradas como el de la figura 4.7, en donde si M = 0, entonces F = A, y si M = 1, entonces F = B.

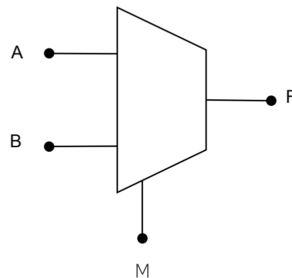


Figura 4.7
Esquema de un multiplexor 2 a 1.

Ejemplo 8. Programa de un multiplexor de dos entradas y una salida.

```
module ejemplo_8(         // Multiplexor de 2 entradas y 1 salida.
    input A, B,           // Entradas A y B al multeplexor (2 switches).
    input M,              // Entrada de control M, de 1 bit (1 switch).
    output reg F );       // Salida F, tipo registro, de 1 bit (led).

    always@ (*)           // Lista de sensibilidad que incluye todas las entradas.
        if( M == 1)       // Si M es 1, entonces...
            F = B;        // F toma el valor de B.
        else              // Si no...
            F = A;        // F toma el valor de A.

endmodule
```

4.6.2. SENTENCIA «CASE... ENDCASE»

Esta sentencia responde al diagrama de flujo que muestra la figura 4.8. Es un caso especial de un «if múltiple» en el que, dependiendo del valor que tome la variable «Val», es el caso («case») que se ejecuta, el cual debe contener una o más instrucciones. Si la variable «Val» no coincide con ningún valor de los especificados en los distintos «case», se ejecuta el *default*. Una vez finalizado el «case», se sigue la secuencia normal del programa.

Si un «case» contiene más de una sentencia, se debe incluir «begin» y «end», como en los bloques «always».

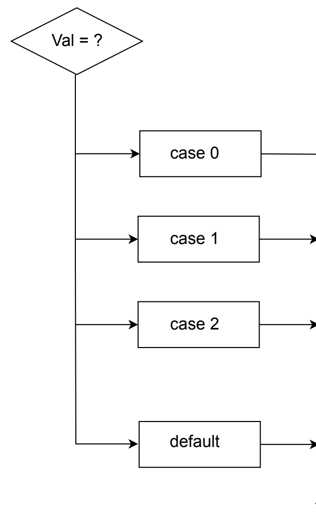


Figura 4.8
Diagrama de flujo de la sentencia «case... endcase».

Para demostrar el funcionamiento de esta sentencia, el ejemplo 9 resuelve la operación de un decodificador de 2 a 4, como muestra la figura 4.9. Es decir, según la combinación en la entrada «X» se genera un «1» en una de las salidas «Y» y «0» en las restantes.

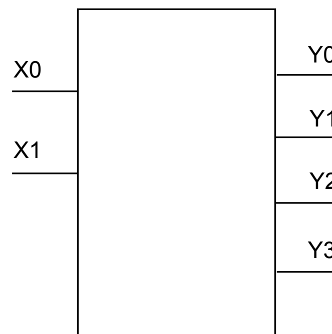


Figura 4.9.
Decodificador 2 a 4.

Ejemplo 9. Demostración del funcionamiento de la sentencia «case» con un decodificador.

```
module ejemplo_9(               // Decodificador 2 a 4.
    input [1:0] X,              // Entrada X de 2 bits (puede representar 0,1,2,3).
    output reg [3:0] Y );       // Salida Y de 4 bits.

    always@ (*)                 // La lista de sensibilidad incluye todas las entradas.
        case (X)                // En el caso de que X sea...
            0: Y = 4'b0001;     // 0, entonces el bit0 de la salida Y toma el valor 1.
            1: Y = 4'b0010;     // 1, entonces el bit1 de la salida Y toma el valor 1.
            2: Y = 4'b0100;     // 2, entonces el bit2 de la salida Y toma el valor 1.
            3: Y = 4'b1000;     // 3, entonces el bit3 de la salida Y toma el valor 1.
        endcase

endmodule
```

4.6.3. SENTENCIA «FOR»

Cuando se requiere programar estructuras repetitivas, para simplificar el código es conveniente utilizar un bloque «for». La figura 4.10. muestra el diagrama de flujo de esta sentencia y funciona como sigue:

- «expresión 1» es una variable de control y contiene un valor inicial.
- «condición» define si «expresión 1» es menor o igual al valor final.
- Si se cumple la «condición», se ejecuta repetitivamente «sentencia(s)» y «expresión 2» hasta alcanzar el valor final.
- Cuando ya no se cumple la «condición», el programa sigue su secuencia normal (camino de la salida «NO»).

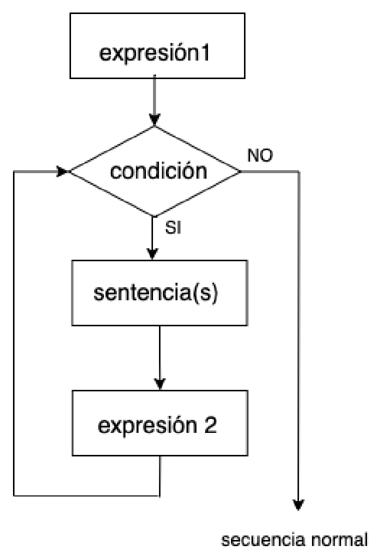


Figura 4.10. Diagrama de flujo de la sentencia «for».

Sintaxis:

```
for (expresión 1; condición; expresión 2)

sentencia;
```

Ejemplo:

```
Integer k;                      // «k» se define como variable entera,
                                // para usar como control del ciclo «for».

for ( k=0 ; k <= n ; k = k+1 ); // se establece un loop desde «0» hasta «n»,
                                // con incremento en 1 de la variable «k» en cada iteración.

assign valor = valor + 1;       // la variable «valor» almacena su propio valor más 1
                                // en cada iteración, hasta k = n, que establece el fin del «for».
```

Ejemplo 10. Este programa resuelve la operación de un decodificador 2 a 4 con *enable* (habilitador), haciendo uso de la sentencia «for». La iteración permite recorrer todas las combinaciones posibles con dos entradas (4 combinaciones).

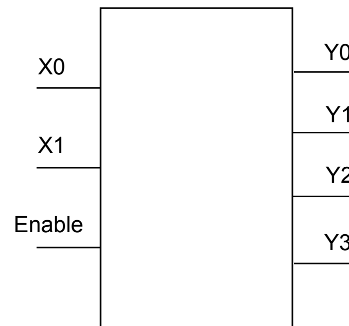


Figura 4.11. Decodificador 2 a 4 con habilitador.

```
module ejemplo_10(                  // Decodificador 2 a 4 con enable.
    input [1:0] X,                  // Entrada X de 2 bits (puede representar 0,1,2,3).
    input Enable,                   // Para habilitar el funcionamiento del decodificador.
    output reg [3:0] Y );           // Salida Y, tipo registro, de 4 bits.

    integer k;                      // Declara k como entero, para controlar la iteración
                                    // del for.

    always@ (X, Enable)
        for (k=0; k<=3; k=k+1)      // En cada iteración k incrementa en 1, desde 0 a 3.
            if( X==k && Enable==1 ) // Si X y k son iguales y Enable es 1, entonces...
                Y[k] = 1;           // Y[k] es 1.
            else                    // Si no (si X es distinto de k o Enable es 0)...
                Y[k] = 0;           // Y[k] es 0.

endmodule
```

4.7. Circuitos secuenciales

En el lenguaje Verilog existen comandos y operadores que son propios de los circuitos secuenciales. Esto es, circuitos que modifican su estado (salidas) dependiendo de la entrada y de una señal de reloj.

4.7.1. DETECCIÓN DE CANTOS (FLANCOS) DE RELOJ

Los *flip flops* son circuitos secuenciales que determinan sus salidas de acuerdo con una tabla específica según su tipo y la aplicación de un reloj (*clock*). En Verilog, la lista de sensibilidad de la sentencia «always» incluye la detección de cantos de reloj que pueden ser de subida «posedge» o de bajada «negedge».

Sintaxis:

```
always @ (posedge clock)      // las acciones en el bloque always se ejecutan cada vez
                              // que el canto del reloj pasa de un nivel bajo a alto.

always @ (negedge clock)      // el bloque se ejecuta con el canto de bajada del reloj.
```

4.7.2. DEFINICIÓN DE CONSTANTES (PARAMETER)

Para la definición de máquinas de estado es útil aplicar el comando «parameter», que asigna un valor directamente a la variable. Por ejemplo:

```
parameter digito_a = 4'b 0011,    // declara los valores para digito_a y el
digito_b =4'b 1001;               // digito_b.

reg salida [ 3:0 ];               // declara «salida» como un registro de 4 bits.

salida = digito_b;                // «salida» toma el valor binario «1001».
```

4.7.3. CONCATENACIÓN

Este operador es utilizado para agrupar variables, constantes y hasta expresiones en un arreglo. La concatenación se hace de izquierda a derecha, en paréntesis de llaves «{}».

Sintaxis:

```
variable = { x, y, z }      // «variable» queda conformada por los valores
                            // concatenados de x, y, z.
```

Ejemplos de las distintas expresiones de concatenación:

```
variable = { a, b[3:0], c }  // si a y c son de 8 bits, variable es de 20 bits.

variable = { 3 {z} }         // variable es equivalente a {z,z,z}.

variable = { b, 3 {c,d} }    // es equivalente a {b,c,d,c,d,c,d}.

variable = { c, 4'b0000 }// al valor constante c se agregan 4 ceros a la derecha.

variable = { c, a | b }      // al valor constante de c se agrega el resultado de a or b.
```

Es importante verificar que el tamaño total de la concatenación coincida con el tamaño de la variable a la que se asigna el resultado. La concatenación también es útil para combinar señales en módulos o para construir patrones específicos en los registros.

Ejemplo de generación de números aleatorios (concatenación de una variable de 4 bits con una expresión *or* exclusivo):

```
reg val = 32'hb7a4;          // valor inicial cualquiera de val, de 32 bits.

always @(posedge clk ) begin

val <= { val[30:0] , val[31] ^ val[21] ^ val[1] ^ val[0] };

end
```

La variable «val» es de 32 bits. A los 31 bits más significativos se agrega a la derecha el resultado de la operación *or* exclusivo entre los bits 31, 21, 1 y 0, respectivamente. Esto provoca que «val» resulte en un número aleatorio. También, nótese que por cada canto de reloj se provoca un desplazamiento hacia la izquierda del contenido de la variable «val» (efecto registro de desplazamiento).

Generación de un registro de desplazamiento a la derecha:

```
initial reg_desp =10'b1000000000;        // <<initial>> se usa para asignar un valor inicial.

always @ (posedge clk)

reg_desp <= { 1'b0, reg_desp [9:1] };    // registro de 10 bits.
```

Con cada canto positivo del reloj (ver sección 4.7.1) la palabra inicial «reg_desp» de 10 bits (D9... D0) se desplaza una posición a la derecha y se completa con un cero en el bit más significativo.

Ejemplo 11. Este programa trata las variables a, b y c como una concatenación. La variable «alarma» se activa solo cuando a, b y c son «1».

```
module ejemplo_11(              // Operación con variables concatenadas.
    input a, b, c,
    output reg alarma );

    always@ (*)
        case ( {a,b,c} )        // El case actúa según la concatenación abc.
            3'b111: alarma = 1; // Si a, b y c son 1, entonces alarma toma el valor 1.
            default alarma = 0; // En caso contrario, por defecto, alarma toma el valor 0.
        endcase

endmodule
```

4.7.4. RELOJ INTERNO DE LA TARJETA BASYS-3

Esta tarjeta cuenta un reloj interno de frecuencia 100 MHz para uso de los dispositivos secuenciales. Sin embargo, algunos desarrollos requieren relojes de frecuencias menores, por ejemplo, cuando se asignan recursos para visualizar el funcionamiento de leds que se encienden y apagan regularmente, o cuando se analiza el comportamiento de un contador o registro de desplazamiento. En estos casos, es necesario disminuir la frecuencia de la tarjeta a una que se pueda percibir con claridad; se sugiere que sea inferior a 20 Hz.

Para los ejemplos que se desarrollan en este texto se utiliza un reloj de 1 Hz, lo que permite verificar sin problema la operación de un *display* o un led, entre otros. El bloque que muestra la programación de un divisor de frecuencia se muestra en el ejemplo 12 (la frecuencia de 1 Hz se obtiene dividiendo la frecuencia de 100 MHz del reloj de la tarjeta por el parámetro 100.000.000).

Ejemplo 12. Programa para dividir la frecuencia del reloj interno de la BASYS-3.

```
module divisor_frecuencia(                  // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                    // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );                // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;           // Asigna el valor de 100M a una constante Seg1
                                            // para producir 1 seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);        // Guarda en Nbits el número de bits necesarios
                                            // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0;  // Define el registro contador_divisor para
                                            // utilizarlo como contador y lo inicializa en 0.

    always@(posedge reloj_in)               // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
        if (contador_divisor1 == Seg1-1)    // Si el contador_divisor llega al valor máximo, lo
            contador_divisor1 <= 0;         // deja en 0 (para iniciar nuevo conteo).
        else
            contador_divisor1 <= contador_divisor1 + 1; // Si el contador_divisor no ha llegado
                                                        // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];      // Asigna al reloj_out el valor más
                                                        // significativo del registro
endmodule                                               // contador_divisor (frec. de 1Hz).
```

Una vez codificado el programa «divisor_frecuencia», se puede utilizar en cualquier etapa de los circuitos diseñados llamándolo con la instanciación:

```
divisor_frecuencia(                 // Llamado al módulo divisor_frecuencia.
    .clk_in(clk),                   // Traspasa el reloj interno de 100 MHz a la variable clk_in.
    .clk_out(new_clk_1Hz) );        // clk_out retorna con el valor de 1 Hz y lo traspasa a
                                    // new_clk_1Hz.
```

Para mayor comprensión, se sugiere al lector analizar los ejemplos 17 y 18 de este capítulo o las prácticas 8, 9 y 10 del capítulo 6.

4.7.5. ASIGNACIONES BLOQUEANTES Y NO BLOQUEANTES

Se identifican dentro de un bloque «always» por su sintaxis:

- Asignación bloqueante «=»: Significa que una salida cambia inmediatamente, como ocurre en los circuitos combinacionales cuando se modifica una entrada.

- Asignación no bloqueante «<=»: En este caso se determinan los valores de todas las asignaciones «<=», sin importar la ubicación dentro del bloque «always», y luego se asignan sus valores simultáneamente. Esta asignación modela el funcionamiento de un *flip flop*.

Ejemplo 13. El *flip flop* tipo D de la figura 4.12 cambia su estado si cambia su entrada D, pero en el momento que ocurre un canto de subida del reloj (el «always» se procesa solo al momento que ocurre cada canto de subida del reloj).

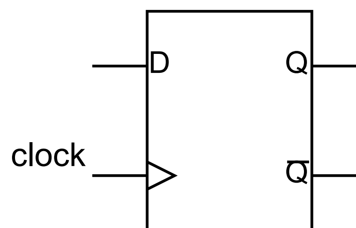


Figura 4.12.
Flip flop tipo D.

```
module ejemplo_13(              // Flip Flop tipo D.
    input D, clock,             // D es la entrada al FF y clock un reloj interno o externo.
    output reg Q );             // Q es la salida del FF.

    always@ (posedge clock)     // El always se activa solo con el canto de subida del clock.
        Q <= D;                 // Q toma erl valor D (no bloqueante) en cada pulso de subida
                                // del clock.
endmodule
```

Ejemplo 14. *Flip flop* tipo D de la figura 4.13, con *reset*, aplicando «asignación bloqueante». En este caso, el *reset* se ejecuta de forma inmediata, independiente del reloj.

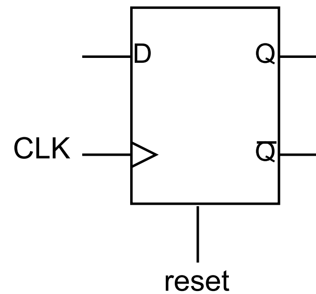


Figura 4.13.
Flip flop tipo D con reset.

```
module ejemplo_14(                        // Flip Flop tipo D con reset y asignación bloqueante.
    input D, clk, reset,
    output reg Q );

    always@ (posedge clk, posedge reset)  // El always se activa con el canto de subida del clk
        if (reset)                        // o reset.
            Q = 0;                        // Reset asincrónico, con asignación bloqueante.
        else
            Q <= D;                       // Asignación de Q no bloqueante.

endmodule
```

Ejemplo 15. *Flip flop* tipo D con *reset* (el mismo del ejemplo anterior), aplicando «asignación no bloqueante». En este caso, el *reset* se ejecuta una vez que ocurre el canto de subida del reloj.

```
module ejemplo_15(              // Flip Flop tipo D con reset y asignación no bloqueante.
    input D, clk, reset,
    output reg Q );

    always@ (posedge clk)       // La señal reset no se encuentra en la lista de sensibilidad.
        if (reset)              // El reset actúa en sincronía con el clock (no bloqueante).
            Q <= 0;
        else
            Q <= D;

endmodule
```

4.7.6. *FLIP FLOP* TIPO JK

La figura 4.14 muestra un esquema del *flip flop* JK. Este cuenta con dos entradas, dos salidas y un reloj. Adicionalmente, si se indica, puede contar con borrado (reset) que pone la salida en «0» y preseteado (*preset*) que pone la salida en «1».

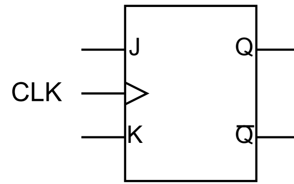


Figura 4.14.
Flip flop tipo JK.

Ejemplo 16. Implementación de un *flip flop* tipo JK utilizando la sentencia «case».

```
module ejemplo_16(                  // Flip Flop JK, utilizando instrucción case.
    input clk,                      // Reloj de 100 MHz de la tarjeta.
    input J, K,                     // Entradas J y K al FF.
    output reg Q );                 // Salida Q del FF.

    always@ (negedge clk)           // El FF trabaja con el canto de bajada del reloj.
        case ({ J,K })              // Variables JK concatenadas.
            2'b11: Q <= ~Q;         // Si J y K son 1, Q cambia al valor negado de Q.
            2'b01: Q <= 0;          // Si J es 0 y K es 1, Q toma el valor 0.
            2'b10: Q <= 1;          // Si J es 1 y K es 0, Q toma el valor 1.
            2'b00: Q <= Q;          // Si J y K son 0, Q mantiene su valor.
        endcase

endmodule
```

4.8. Aplicaciones de los *flip flops*

Los *flip flops* tienen su principal uso como registros de desplazamiento y como contadores. Para los registros de desplazamiento se usan *flip flop* tipo D y para los contadores *flip flop* tipo T.

4.8.1. REGISTROS DE DESPLAZAMIENTO

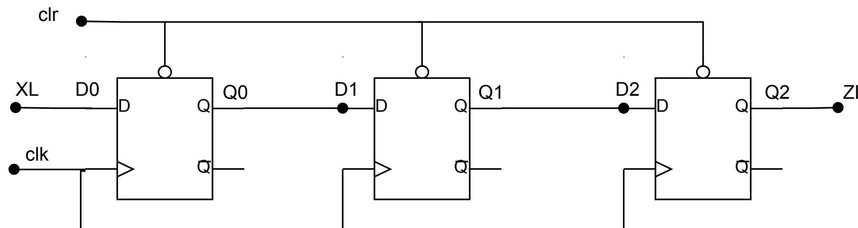


Figura 4.15.
Registro desplazamiento con reset (clr).

El ejemplo 17 muestra una solución en lenguaje Verilog para un registro de desplazamiento del tipo entrada serie y salida serie. El esquema en bloques de la figura 4.15 muestra la entrada XL por D0 y salida ZL por Q2.

Ejemplo 17. Implementación de un registro de desplazamiento con 3 *flip flops* conectados en serie.

```
module ejemplo_17(                      // Registro de desplazamiento serie-serie.
    input XL,
    input wire clk,
    input clear,
    output reg ZL,
    output reg Q0, Q1, Q2 );

    wire new_clk_1Hz;

    divisor_frecuencia (                // Llamada a la instancia divisor_frecuencia
        .reloj_in(clk),                 // que produce un reloj de 1Hz, a partir del
        .reloj_out(new_clk_1Hz) );      // reloj de 100 MHz de la tarjeta.

    always@ (posedge new_clk_1Hz)       // Bloque que produce el desplazamiento al ritmo
    begin                               // de los cantos de subida del reloj de 1 Hz.
        if (clear) Q0 = 1'b0;
        else
            case (XL)
                1'b0: Q0 <= 0;          // Si XL es 0, Q0 es 0.
                1'b1: Q0 <= 1;          // Si XL es 1, Q0 es 1.
            endcase

        if (clear) Q1 = 1'b0;
        else
            case (Q0)
                1'b0: Q1 <= 0;          // Si Q0 es 0, Q1 es 0.
                1'b1: Q1 <= 1;          // Si Q0 es 1, Q1 es 1.
            endcase

        if (clear) Q2 = 1'b0;
        else

           case (Q1)
               1'b0: Q2 <= 0;      // Si Q1 es 0, Q2 es 0.
               1'b1: Q2 <= 1;      // Si Q1 es 1, Q2 es 1.
           endcase

        ZL = Q2;
    end

endmodule

module divisor_frecuencia(            // Divide la frecuencia del reloj de 100 MHz a 1 Hz.
    input wire clk_in,                // Reloj interno de 100 MHz.
    output wire clk_out );

    parameter M = 100_100_100;        // Asigna un valor de 100M a una constante M.
    localparam N = $clog2(M);         // Guarda en N el número de bits necesarios para
                                      // almacenar M.
    reg [N-1:0] divcounter = 0;       // Define el registro divcounter para utilizar como
                                      // contador y lo inicia en 0.
    always@(posedge clk_in)           // Cuenta de 0 a M-1, en cada subida del reloj int.
        if (divcounter == M-1)        // Si el divcounter llega al valor máximo,
            divcounter <= 0;          // lo deja en 0 (para nuevo conteo desde cero).
        else
           divcounter <= divcounter +1; // Si no ha llegado al valor máximo, incrementa en 1.

    assign clk_out = divcounter[N-1];  // Asigna al reloj de salida el valor más
                                      // significativo del registro divcounter.
endmodule
```

4.8.2. CONTADORES

Son circuitos secuenciales que permiten llevar la cuenta de un proceso, incrementando de uno en uno una variable. El esquema de la figura 4.16. muestra 3 flip flops JK conectados como tipo T, donde la salida QA es el dígito LSB y la salida QC el MSB.

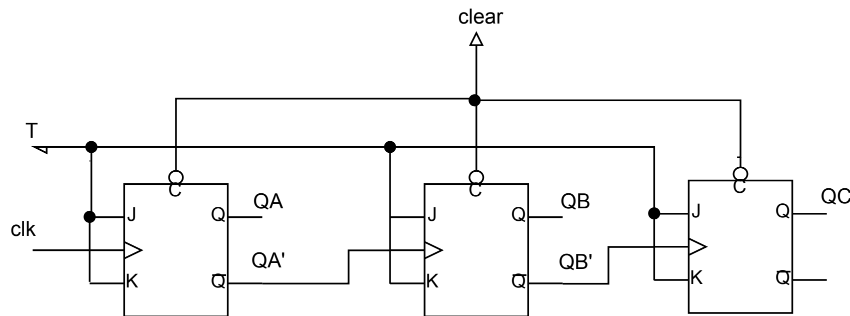


Figura 4.16.
Contador binario de 3 bit con *reset* (*clear*).

Ejemplo 18. La figura 4.16 muestra un contador binario de 3 bits diseñado con *flip flops* tipo T que disponen de *clear*. En el programa se incluye el módulo divisor de frecuencia para poder visualizar la cuenta en los recursos asignados a las salidas (leds). Si T es «1», el contador lleva la cuenta desde 000 hasta 111 (0 a 7 decimal); si T es «0» detiene la cuenta y, al reactivar, continúa desde el valor que tenía cuando se detuvo. Ante un *clear*, el contador se reinicia en «000».

```
module ejemplo_18(                          // Contador binario de 3 bits.
    input T,
    input clk, clear,                       // clk es el reloj de 100MHz de la tarjeta.
    output reg QA, QB, QC );

    wire reloj_1Hz ;

    divisor_frecuencia                      // Llama a la instancia divisor_frecuencia.
        DIV_1(
            .reloj_in(clk),                 // Traspasa el clk al reloj_in.
            .reloj_out(reloj_1Hz) );        // Traspasa el reloj_out retornado del divisor
                                            // a la variable reloj_1Hz.

    always@ (posedge reloj_1Hz, posedge clear)  // Construye el 1er flip flop T.
        begin
            if (clear)  QA = 1'b0;              // Efectúa el reset asincrónico del 1er FF.
            else
                case (T)
                    1'b0: QA <= QA;
                    1'b1: QA <= ~QA;
                endcase
        end
     always@ (posedge ~QA, posedge clear)       // Construye el 2do flip flop T.
        begin
            if (clear)  QB = 1'b0;
            else
                case (T)
                    1'b0: QB <= QB;
                    1'b1: QB <= ~QB;
                endcase
        end
```

```
    always@ (posedge ~QB, posedge clear)       // Construye el 3er flip flop T.
       begin
           if (clear)  QC = 1'b0;
           else
               case (T)
                   1'b0: QC <= QC;
                   1'b1: QC <= ~QC;
               endcase
       end
endmodule

module divisor_frecuencia(                 // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                   // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );               // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;          // Asigna el valor de 100M a una constante Seg1
                                           // para producir 1 seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);       // Guarda en Nbits el número de bits necesarios
                                           // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0; // Define el registro contador_divisor para
                                           // utilizarlo como contador y lo inicializa en 0.

    always@(posedge reloj_in)              // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
        if (contador_divisor1 == Seg1-1)   // Si el contador_divisor llega al valor máximo, lo
            contador_divisor1 <= 0;        // deja en 0 (para iniciar nuevo conteo).
        else
            contador_divisor1 <= contador_divisor1 + 1; // Si el contador_divisor no ha llegado
                                                        // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];      // Asigna al reloj_out el valor más
                                                        // significativo del registro
endmodule                                               // contador_divisor (frec. de 1Hz).
```

4.9. Instancias

Verilog contempla descripciones jerárquicas de redes complejas que permiten generar un programa compuesto de un módulo principal y uno o más módulos adicionales, en lugar de un único gran módulo. Estos módulos complementarios se «instancian» (se invocan) desde el programa principal para conformar el sistema completo. Las instancias permiten reutilizar módulos y simplificar los diseños.

Descripción general de la estructura modular:

```
module programa_principal (// Declaración del nombre
                            // de la función y la cantidad.
        // Declaración de las variables );
        // Declaración de variables internas.
        // Cuerpo del programa principal
        // Llamado a la instancia: instancia_1 (var de entrada, var de salida).
        // Resto del programa principal.
endmodule

module instancia_1 (

        // Declaración de las variables );
        // Declaración de variables internas.
        // Cuerpo del módulo de instanciación
endmodule
```

Ejemplo 19. El siguiente ejemplo corresponde a la instanciación de una unidad que realiza 3 operaciones lógicas con variables de 4 bits (and, or, or exclusivo), dependiendo de la posición de 2 *switches* selectores (como una referencia adicional, ver práctica 11 del capítulo 7 para un sumador de 4 bits).

```
module operaciones_logicas(            // Programa operaciones lógicas con instanciaciones.
    input [3:0] In1, In2, In3, In4,
    input [1:0] Sel1, Sel2,
    output [3:0] Res1, Res2 );

 // Dos instanciaciones lógicas para computar según el valor del selector (Sel).
 // (Dos llamados al módulo "logica" con distintos datos en cada instanciación).
    logica ( .A(In1), .B(In2), .Sel(Sel1), .Res(Res1) );
    logica ( .A(In3), .B(In4), .Sel(Sel2), .Res(Res2) );

endmodule

module logica(                         // Módulo de instanciación.
    input [3:0] A, B,
    input [1:0] Sel,                   // Selector de la instanciación.
    output reg [3:0] Res );

    always@ (*)
        case (Sel)
            2'b00: Res = A & B;        // Operación and.
            2'b01: Res = A | B;        // Operación or.
            2'b10: Res = A ^ B;        // Operación or exclusivo.
            default: Res = 4'B0000;
        endcase
endmodule
```

4.10. Funciones

En Verilog, las funciones son bloques de código reutilizables que permiten realizar operaciones específicas. Son llamadas desde un programa principal y devuelven un solo valor. Se aplican para simplificar un código, al no tener que reescribirlo varias veces en un mismo programa. Un usuario puede escribir las funciones que desee. Algunas funciones propias de Verilog son:

```
$time       // Entrega el tiempo actual de simulación.

$display    // Imprime por pantalla un mensaje que puede incluir variables.

$monitor    // Imprime un mensaje en pantalla siempre que cambie una de las variables.

$write      // Similar a «$display», pero no incluye retorno de carro.

$finish     // Indica el final de una simulación.

$random     // Retorna un valor aleatorio entero de 32 bits.
```

Las funciones se definen en el módulo en el que se utilizan, aunque también es posible definirlas en archivos separados y utilizar la directiva de compilación «`include» para incluir la función en el módulo principal (es el tilde invertido seguido de la palabra *include*). El código de la función se escribe entre las palabras «function» y «endfunction», y las reglas de diseño son:

- No puede incluir retardos de tiempo como «#delay» y bloques «always@» (posedge, negedge). Por lo mismo, se usan solo con lógica combinacional.
- Solo debe tener un retorno de función.
- Puede tener varias entradas, pero una sola salida.
- El orden de declaración dentro de la función define como se utilizan las variables al ser llamadas.
- Puede incluir llamado a otras funciones.

Sintaxis: Una función, además de tener en cuenta las restricciones propias, debe cumplir con la siguiente estructura del bloque:

```
function [ ancho : 0 ] nombre_de_funcion; // Declaración del nombre de la función
                                          // y la cantidad de bits de salida.
    // Declaración de los argumentos de entrada.
    // Declaración de variables internas de tipo reg (si es necesario).
    // Cuerpo de la función
    begin
            nombre_funcion = valor;       // La función debe asignar un valor
                                          // a su nombre
    end
endfunction
```

Ejemplo 20. Este ejemplo muestra en detalle cómo opera una función para la suma aritmética de dos variables de 4 bits. El resultado se almacena en un registro de 5 bits (para contener un posible acarreo).

```
module funciones_01(                   // Programa principal que incluye una función.
    input [3:0] A, B, C, D,            // 16 switches.
    output [4:0] Suma1, Suma2 );       // 10 leds.

    assign Suma1 = suma_4bits(A, B);   // Las variables Suma1 y Suma 2 toman el valor de
    assign Suma2 = suma_4bits(C, D);   // retorno de la función suma_4bits.

    function [4:0] suma_4bits;         // Declaración de la función suma_4bits que retorna
        input [3:0] X;                 // un un datos de 5 bits (son 5 bits para contener
        input [3:0] Y;                 // un posible acarreo).
        begin
            suma_4bits = X + Y;        // El valor de retorno de la función es la suma
        end                            // aritmética de X e Y.
    endfunction

endmodule
```

4.11. Tareas

Las tareas, al igual que las funciones, son bloques de código reutilizables que permiten reducir el código del diseño, pero en este caso, no retornan ningún valor; esperan la ocurrencia de eventos específicos y, de acuerdo con ellos, realizan un proceso y modifican variables. Se definen en el módulo en el que se utilizan o bien en un archivo externo, al inicio del programa con la palabra « ` include» (tilde invertido o acento grave más la palabra include). El código se explicita entre los comandos «task» y «endtask». Las reglas de diseño son:

- Puede incluir retardos de tiempo como «#delay», posedge, negedge. Por lo mismo, las tareas modelan lógica combinacional y secuencial. Esto las hace útiles para secuencias de prueba, simulaciones o bloques que necesitan esperar eventos específicos.
- Puede tener varias entradas y salidas.
- El orden de declaración dentro de la tarea define cómo se utilizan las variables al ser llamadas.
- Pueden incluir llamados a otras funciones o tareas.

Sintaxis:

```
task <nombre>;
    <argumentos>        // Variables de entrada o salida.
    <declaraciones>     // Variables internas si se requiere.
    <funcionalidad>     // Cuerpo de la tarea.
endtask
```

Ejemplo 21. En este ejemplo se muestra cómo hacer un llamado a una tarea generada en el mismo

módulo. En este proyecto existen dos llamados a la tarea para calcular el complemento a 1 de una variable y se ejecutan siempre que ocurra un cambio en las entradas.

```
module tareas_01( a, b, c, d );      // Programa principal que incluye una tarea.
    input [3:0] a, c;
    output reg [3:0] b, d;

    always@ (a)
        complemento1 (a, b);         // Primer llamado a la tarea.

    always@ (c)
        complemento1 (c, d);         // Segundo llamado a la tarea.

    task complemento1;               // Declaración de la tarea complemento1.
        input [3:0] entrada;         // Los argumentos de la llamada quedan
        output [3:0] salida;         // asociados a entrada y salida.
        begin
            salida = - entrada;
        end
    endtask

endmodule
```

CAPÍTULO 5
FUNCIONALIDADES ESPECIALES DE VIVADO

5.1. Simulación

Vivado, al igual que otras plataformas de desarrollo, otorga la posibilidad de simular el funcionamiento de los circuitos diseñados antes de ser implementados en la tarjeta BASYS-3/FPGA. Para esto se crea un programa especial de simulación que integra tres módulos:

- El módulo principal, que corresponde al programa de diseño del circuito cuyo funcionamiento se desea probar (simular). El lector debe estar previamente familiarizado con la programación e implementación de un circuito cualquiera.

- Un módulo de test, que contiene las sentencias de simulación y los valores posibles de las entradas al circuito con las que se desea simular su comportamiento.

- Un módulo de unificación de los módulos de diseño y test, denominado «banco de prueba», que instancia los dos módulos anteriores.

La programación de tipo jerárquica de Vivado permite que coexistan en un único programa de simulación el módulo de diseño con el módulo de test, lo cual hace posible que la síntesis del módulo de diseño utilice los valores de las variables de simulación contenidas en el módulo de test. Para especificar el módulo de test se debe tener en cuenta lo siguiente:

- Las variables de entrada y salida del programa principal pasan a ser las variables de salida y entrada, respectivamente, en el módulo de test.

- Comenzar con la palabra «initial» y terminar con la palabra «$finish».

- Toda simulación requiere establecer un retardo de tiempo. Este se logra con el comando «#». Por ejemplo, «#10» indica un retardo de 10 unidades de tiempo de simulación.

- Los valores de las variables de entrada y salida se visualizan en la «Tcl Console» de Vivado. Para esto, se utiliza el comando «Monitor», el cual imprime los valores en la Tcl Console, en un formato tipo tabla de verdad.

- El formato de la impresión de salida puede ser «%b» para un formato binario, «%d» para decimal y «%h» para hexadecimal.

Para ejemplificar el uso de la simulación, se utiliza el circuito de la figura 5.1, correspondiente a la práctica 2 del capítulo 6. La tabla 5.1. contiene los resultados de las salidas S1 y S2. Ejecute los siguientes pasos para implementar la simulación:

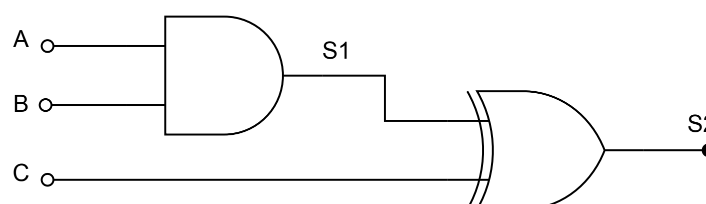


Figura 5.1. Circuito combinacional para simulación.

A	B	C	S_1	S_2
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

Tabla 5.1.
Tabla de verdad del circuito combinacional

Paso 1. Escriba el programa principal del circuito en Verilog. Verifique la sintaxis y ejecute la síntesis e implementación para asegurar la consistencia de su programa.

```
module combinacional(                          // Módulo principal.
     input A,B,C,
     output S1,S2 );
     assign S1=A&B;
     assign S2=S1^C;
endmodule
```

Paso 2. Adicione al programa principal los módulos de test con las variables de simulación y el banco de prueba que integra ambos módulos.

```
module combinacional(                      // Módulo principal.
    input A,B,C,
    output S1,S2 );
    assign S1=A&B;
    assign S2=S1^C;
endmodule

module testcombinacional(S1,S2,A,B,C);     // Módulo de test.
    input S1,S2;
    output A,B,C;
    reg A,B,C;
    initial
        begin          // El Monitor (Tcl Console) muestra la salida en formato binario.
                       // Las comas indican espacios adicionales.
        $monitor("t", $time,,"A=%b, B=%b, C=%b, S1=%b, S2=%b", A, B, C, S1, S2);
        // Se generan todas las combinaciones de entrada.
            A=0; B=0; C=0;
        #10 A=0; B=0; C=1;    // La simulacion se efectua cada 10 unidades de tiempo.
        #10 A=0; B=1; C=0;
        #10 A=0; B=1; C=1;
        #10 A=1; B=0; C=0;
        #10 A=1; B=0; C=1;
        #10 A=1; B=1; C=0;
        #10 A=1; B=1; C=1;
        #10 $finish;
        end
endmodule
module BancoDePrueba;    // Módulo que unifica los 2 anteriores.
    wire A, B, C, S1, S2;
    combinacional comb(A,B,C,S1,S2);
    testcombinacional test(S1,S2,A,B,C);
endmodule
```

Paso 3. Verifique la correcta sintaxis de la codificación del programa y la creación de los 3 módulos en la ventana «Sources» que se destacan en la figura 5.2.

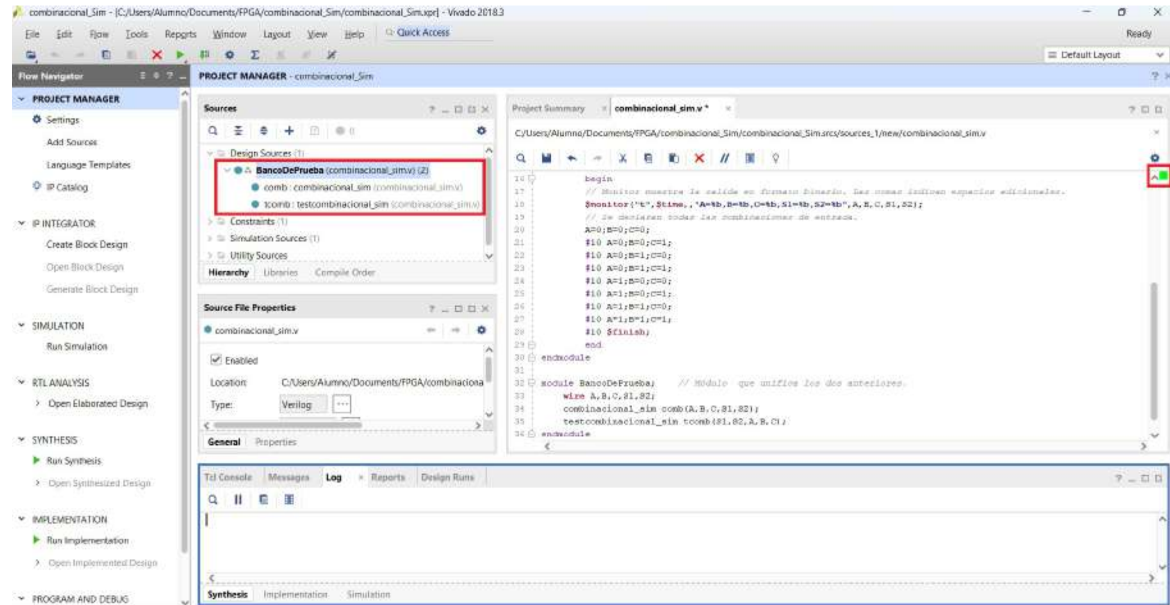


Figura 5.2. Lámina 1 del procedimiento de simulación.

Paso 4. En la sección «SIMULATION» de Vivado, haga clic en «Run Simulation» y «Run Behavioral Simulation». Verifique las marcas circulares (en rojo) que se agregan al código cuando finaliza la simulación (figuras 5.3 y 5.4).

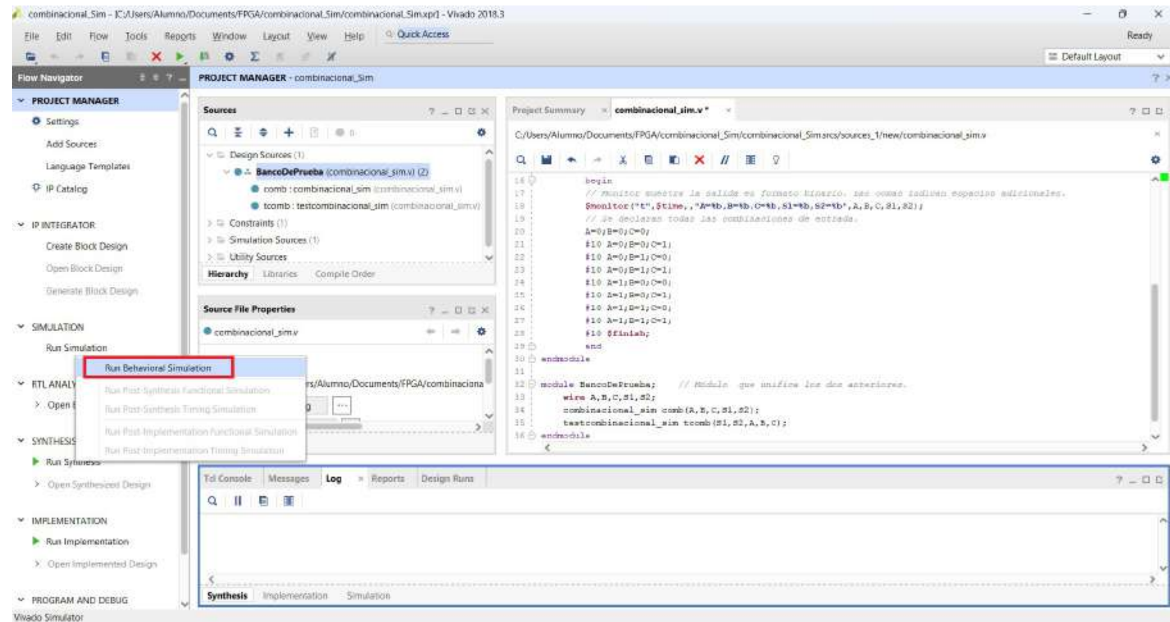


Figura 5.3. Lámina 2 del procedimiento de simulación.

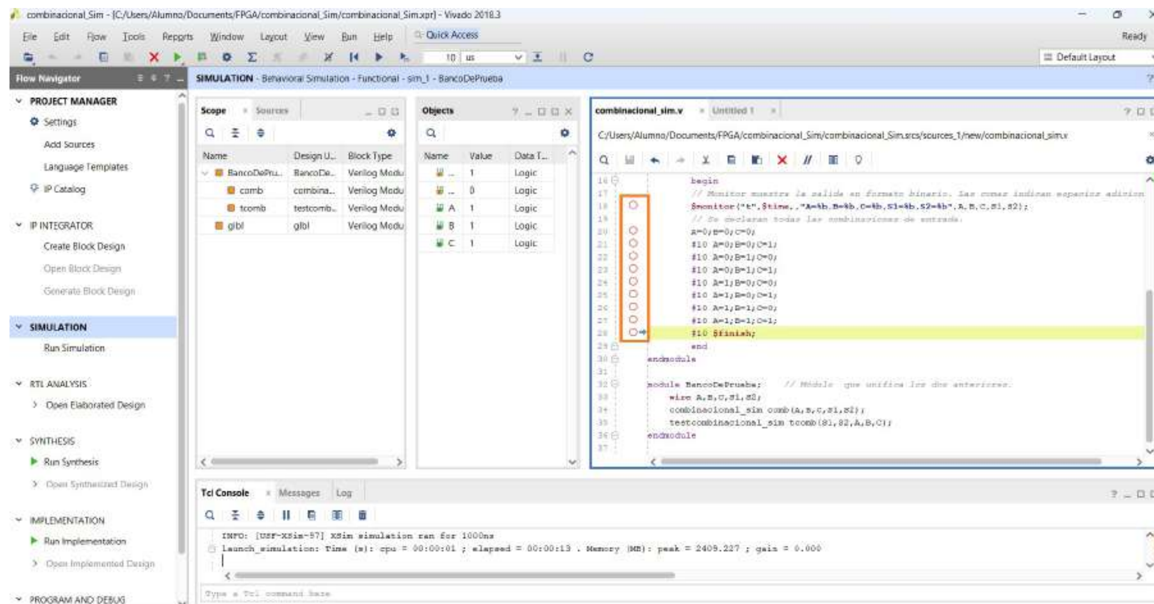


Figura 5.4. Lámina 3 del procedimiento de simulación.

Paso 5. Haga clic en la pestaña «Untitled», que contiene las opciones de simulación, y luego amplie su área de visualización (figuras 5.5 a 5.7).

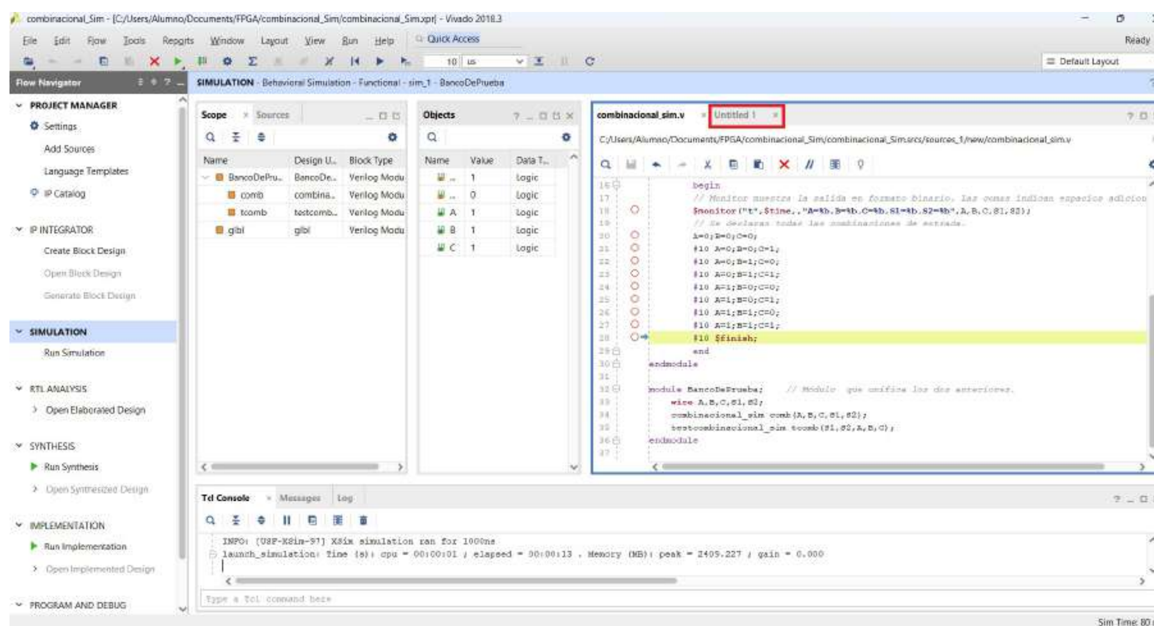


Figura 5.5. Lámina 4 del procedimiento de simulación.

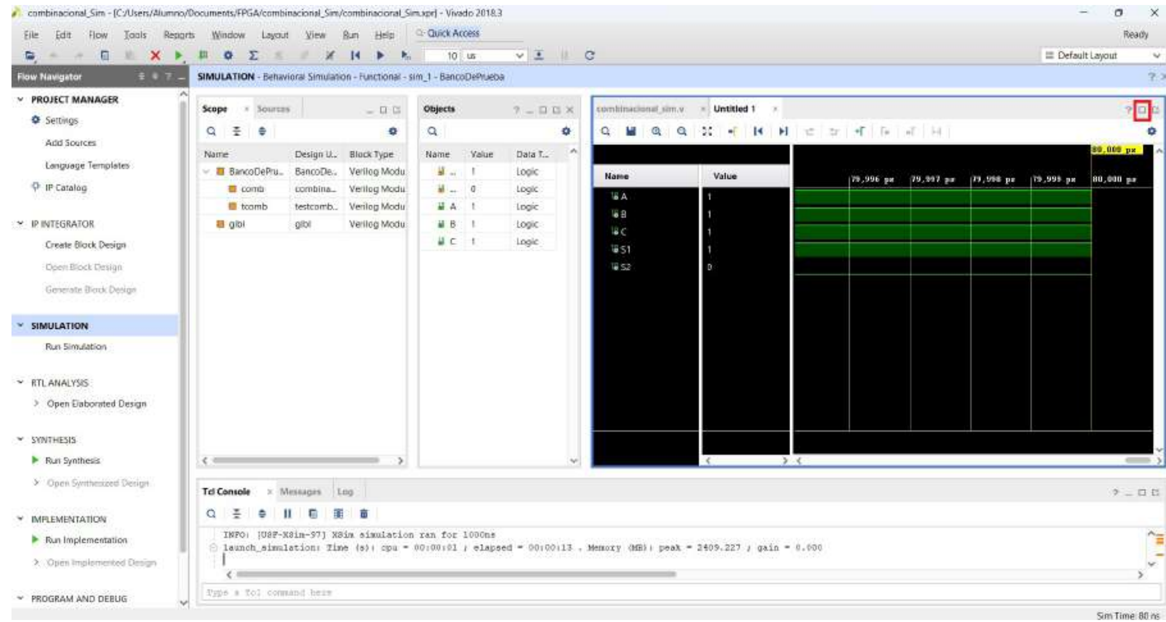


Figura 5.6.
Lámina 5 del procedimiento de simulación.

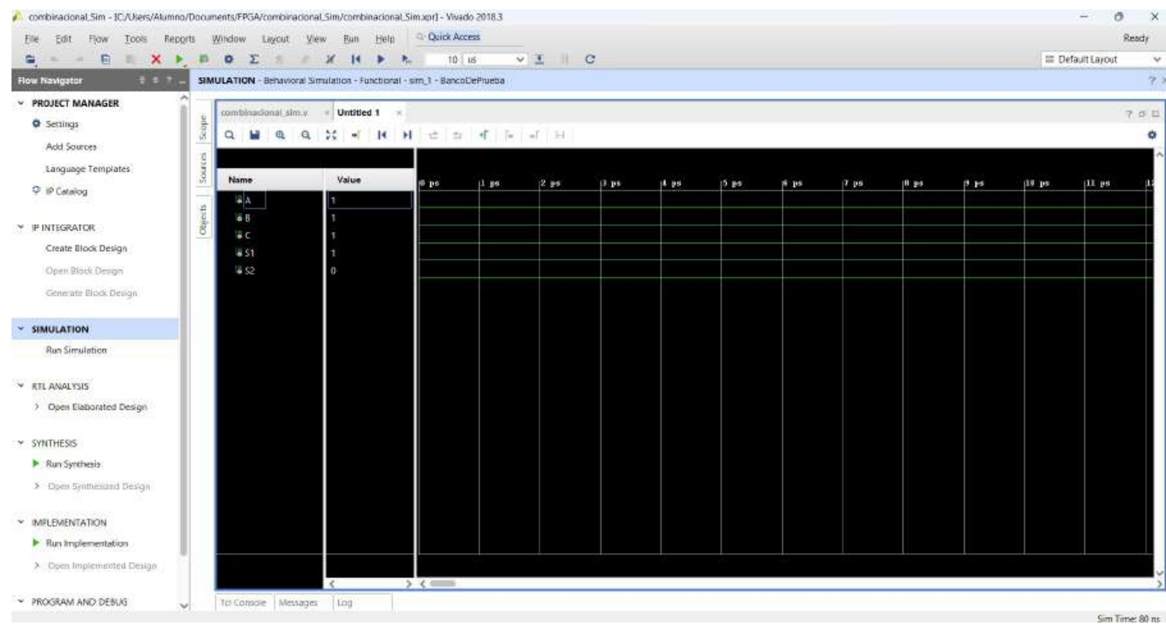


Figura 5.7.
Lámina 6 del procedimiento de simulación.

Paso 6. Haga clic en el ícono que permite la visualización de los cambios de los valores de las variables de entrada y salida en el tiempo. Nótese que las columnas «Name» y «Value» contienen los valores binarios de las entradas y salidas, los cuales van cambiando según el retardo de 10 unidades de tiempo definidas en el módulo de test (figura 5.8).

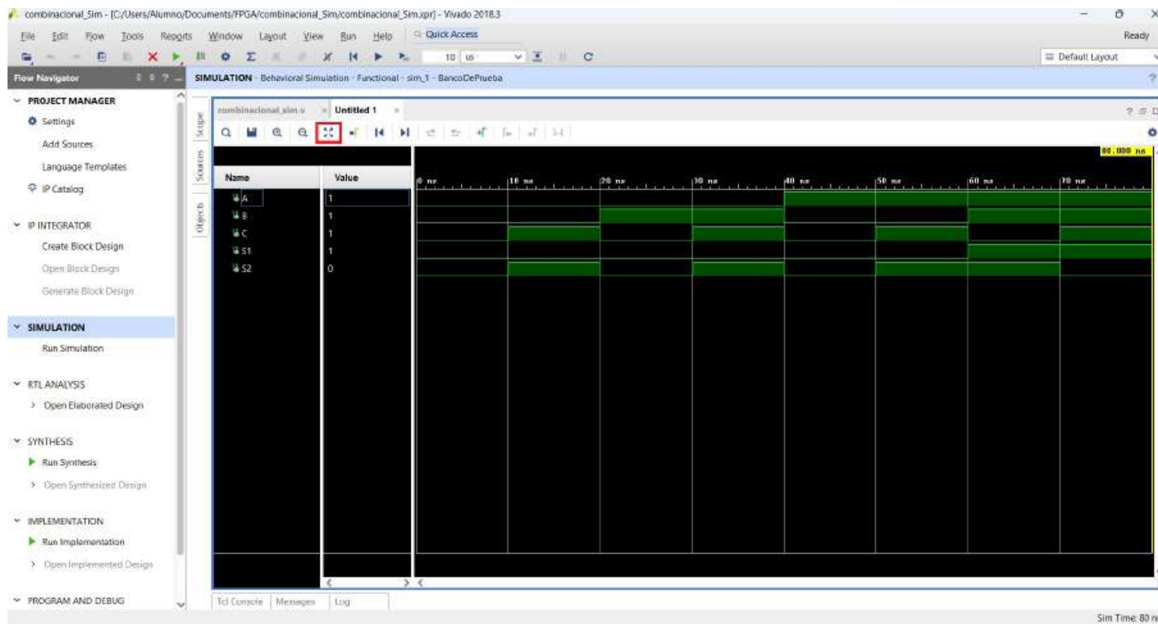


Figura 5.8.
Lámina 7 del procedimiento de simulación.

Paso 7. Posicione el cursor sobre el recuadro en amarillo y arrástrelo hasta la unidad de tiempo que desee. Observe los cambios en las columnas «Name» y «Value» (figuras 5.9 y 5.10).

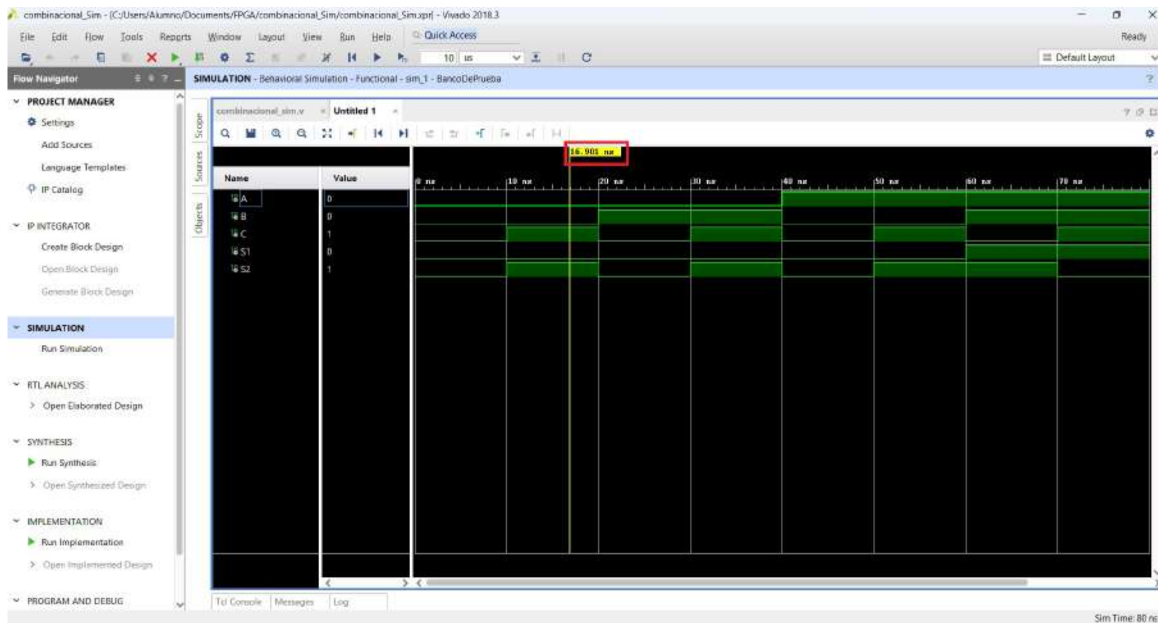


Figura 5.9.
Lamina 8 del procedimiento de simulación.

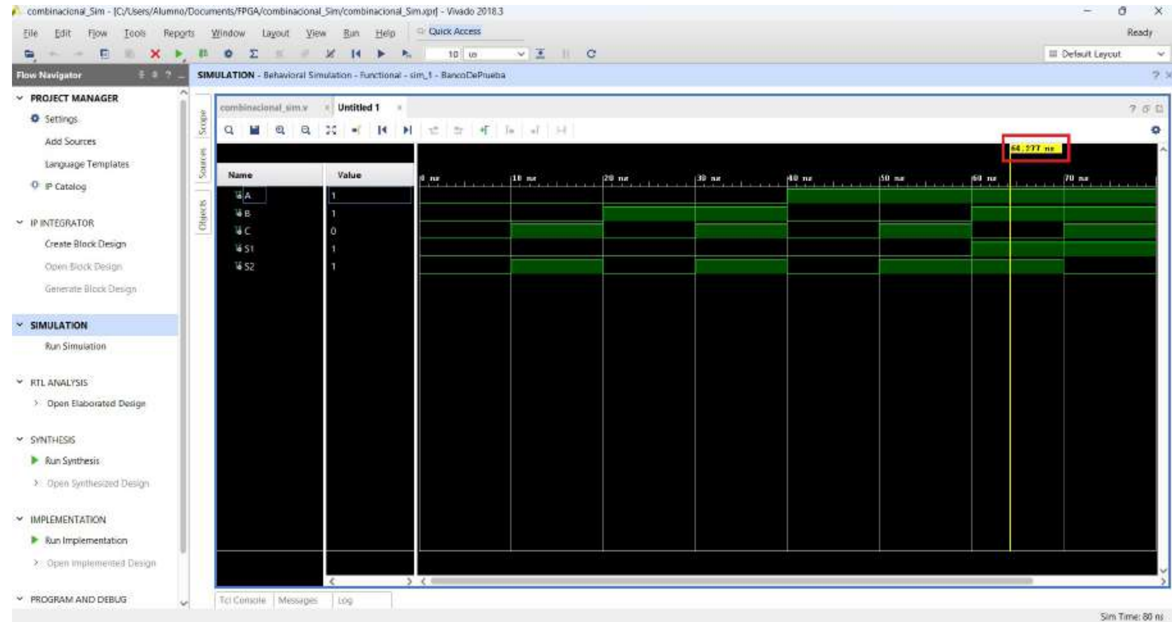


Figura 5.10.
Lámina 9 del procedimiento de simulación.

Paso 8. En la pestaña «Tcl Console» puede verificar las salidas resultantes del circuito para cada una de las combinaciones de las entradas (figura 5.11). Esta debe ser congruente con la tabla de verdad (tabla 5.1).

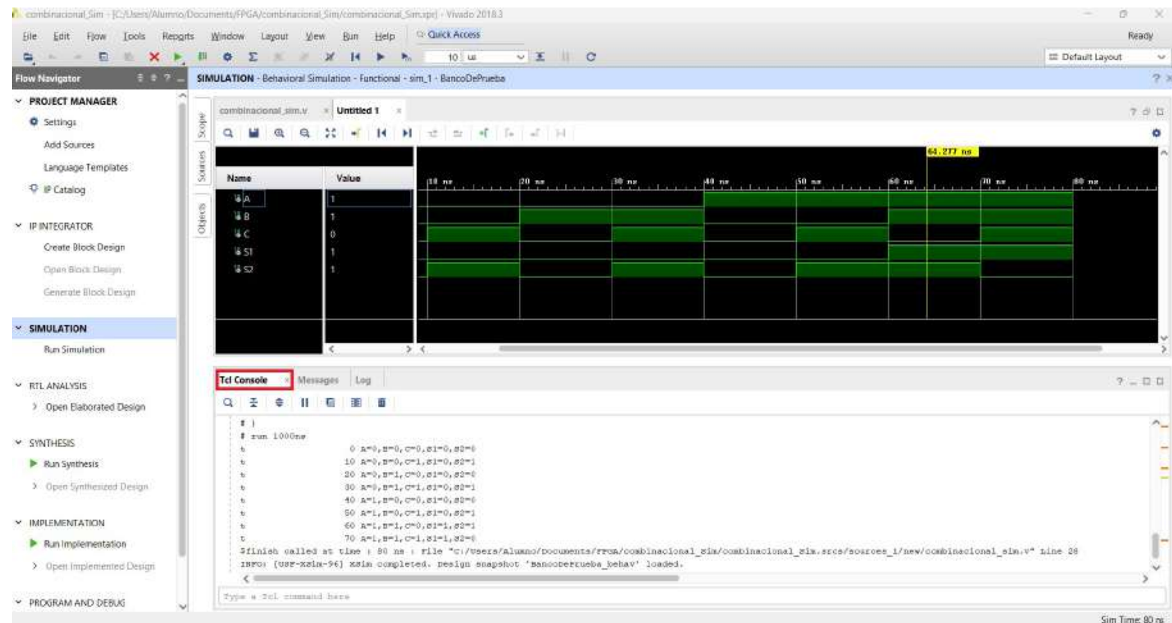


Figura 5.11.
Lámina 10 del procedimiento de simulación.

5.2. Catálogo IP

El Catálogo IP (intelectual property catalog, IP Catalog) de Vivado es una biblioteca que contiene una colección de módulos de programa de propiedad intelectual (IP) que están disponibles para ser utilizados por los usuarios en el desarrollo de sistemas con dispositivos FPGA. Estos bloques de programas incluyen múltiples funcionalidades, como controladores de interfaces, procesamiento de señales, módulos de comunicación, conversores de señales análogo a digital y procesador de reloj, entre otros.

El acceso al catálogo es a través de la funcionalidad «IP Catalog» que se encuentra en el menú de la izquierda de la ventana de desarrollo de Vivado. Esta funcionalidad permite:

- Buscar IP específicas por categorías, según los requerimientos del proyecto.
- Personalizar las IP, ya que muchas son configurables. Esto significa que sus parámetros se pueden ajustar según las necesidades.
- Integrar en un diseño propio el código HDL disponible.
- Facilitar el flujo de trabajo con las IP y favorecer su integración con otros diseños en el entorno de Vivado.

5.2.1. PROCESO DE INTEGRACIÓN Y CONFIGURACIÓN DE UNA IP

Un ejemplo de uso del IP Catalog se muestra en el capítulo 7 de prácticas avanzadas. En específico, la práctica 15 de conversión análogo a digital, la cual utiliza los módulos de programa «xadc» y «clocking» del catálogo. Se sugiere al lector seguir el procedimiento indicado en esa práctica para conocer cómo se parametriza cada bloque y cómo se integra al programa principal. En resumen, cuando se requiera ingresar una entrada analógica a la BASYS-3 para convertirla a un formato digital y continuar con su procesamiento en ese formato (por ejemplo, una señal proveniente de un sensor de temperatura), se utilizan los dos bloques del IP Catalog indicados: «xadc» y «clocking». El procedimiento general para integrarlos es el siguiente:

- Seleccione IP Catalog.
- En la ventana a la derecha, busque «xadc».
- Renombre el módulo (por ejemplo, «xadc1»).
- Se genera el archivo y se despliega la ventana de parametrización.
- Configure según se requiera. Nótese que en este caso no se utilizan, por ejemplo, alarmas.
- Una vez configurado, aplique «OK». Se puede observar la generación de los archivos en la ventana «Sources».
- Escriba el programa principal y la instancia en la IP correspondiente «xadc1».

Además, en este ejemplo se requiere integrar de la misma forma un segundo bloque de programa correspondiente al reloj «clocking» para establecer el tiempo de conversión, lo que determina la frecuencia de muestreo o, de manera equivalente, las muestras por segundo (sample per seg, SPS), como se desarrolla en la práctica 15 del capítulo 7.

5.2.2. EJEMPLOS DE BLOQUES DISPONIBLES EN EL IP CATALOG

Algunos de los bloques IP más comunes incluyen:

- Axi Interconnect: Facilita la comunicación entre varios maestros y esclavos en un sistema basado en AXI.

- Fifo Generator: Permite crear colas FIFO para el almacenamiento temporal de datos.

- Xadc: Permite la conversión análoga a digital.

- Dsp48e: Es un bloque de procesamiento digital que se utiliza para ejecutar operaciones matemáticas complejas.

- Block Memory Generator: Genera bloques de memoria para almacenamiento de datos.

- Clocking: Permite obtener distintas frecuencias de reloj.

- Uart: Protocolo de comunicación serie asíncrono para interfaces de comunicación.

- Gpio: Bloques de entrada/salida generales, para la interconexión con periféricos.

CAPÍTULO 6
PRÁCTICAS DE LABORATORIO

6.1. Introducción

En este capítulo se presenta un conjunto de prácticas de complejidad creciente que permiten profundizar gradualmente en los temas tratados en este texto. Cuando es necesario, se hace referencia a las secciones del libro que se relacionan con el propósito de cada práctica, a fin de facilitar su desarrollo.

6.2. Prácticas

Las diez prácticas que a continuación se desarrollan incluyen el código Verilog, el diagrama esquemático y la asignación de recursos a través del archivo «Basys-3-Master.xdc».

6.2.1. PRÁCTICA 1: COMPUERTAS LÓGICAS

Consiste en diseñar e implementar las compuertas lógicas *and, or, not, nand, nor, xor* y *xnor* en un mismo programa, pero cada una constituyendo un circuito independiente que funciona con sus propias entradas y salidas. La programación del circuito requiere que haya descargado Vivado y desarrollado previamente los puntos 1 al 12 de la sección 3.4.1.

1. Elabore las tablas de verdad de las compuertas *and, or, not, nand, nor, xor y xnor* para 2 entradas.

2. Estudie los ejemplos de la sección 4.4.1. que utilizan los operadores lógicos a nivel de bit.

3. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module Practica_01(                        // Implementación de compuertas lógicas

    input A,B,C,D,E,F,G,H,I,J,K,L,M,       // Entradas a las compuertas (switches)
    output S1,S2,S3,S4,S5,S6,S7);          // Salidas de las compuertas (leds).

    assign S1 = ~A;                        // Compuerta "not".
    assign S2 = B & C;                     // Compuerta "and"
    assign S3 = D | E;                     // Compuerta "or"
    assign S4 = ~(F & G);                  // Compuerta "nand"
    assign S5 = ~(H | I);                  // Compuerta "nor"
    assign S6 = J ^ K;                     // Compuerta "xor"
    assign S7 = ~(L ^ M);                  // Compuerta "xnor"

endmodule
```

4. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.1).

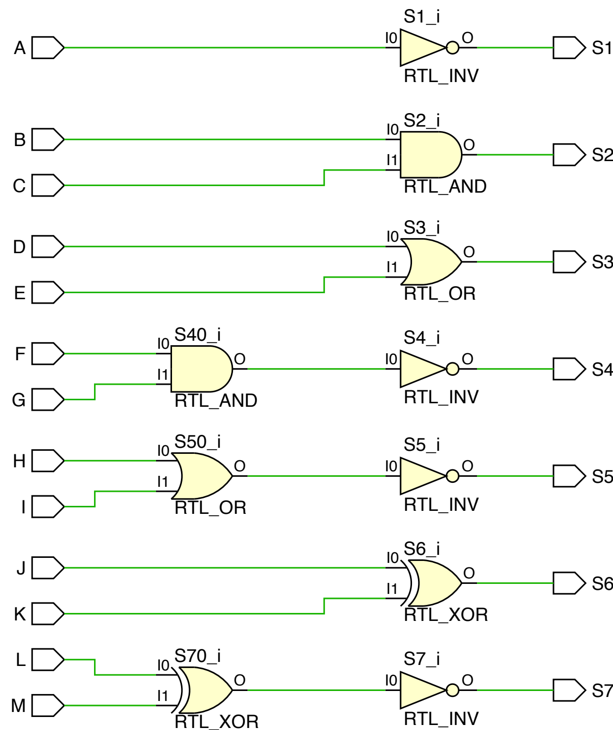


Figura 6.1. Esquemático de la práctica 1.

5. Ejecute los pasos 16 a 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {A}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {B}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {C}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {D}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {E}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {F}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {G}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {H}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {I}]
set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {J}]
set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {K}]
set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {L}]
set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {M}]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {sw13}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {S1}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {S2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {S3}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led5}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {S4}]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led7}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {S5}]
#set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led9}]
set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {S6}]
#set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {led11
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {S7}]
```

6. Ejecute los pasos 19 a 28 de las secciones 3.4.4 y 3.4.5, para sintetizar, implementar, generar el *bitstream* y traspasar el programa a la tarjeta.

7. Opere los *switches* asignados en el punto 5 y observe los leds correspondientes. Compare el funcionamiento con las tablas de verdad solicitadas en el punto 1.

6.2.2. PRÁCTICA 2: CIRCUITOS COMBINACIONALES

Consiste en diseñar e implementar los tres circuitos combinacionales que se muestran en la figura 6.2 en un solo programa. Estos funcionan de manera independiente con sus propias entradas y salidas. La programación requiere que haya descargado Vivado y desarrollado previamente los pasos 1 al 12 de la sección 3.4.1.

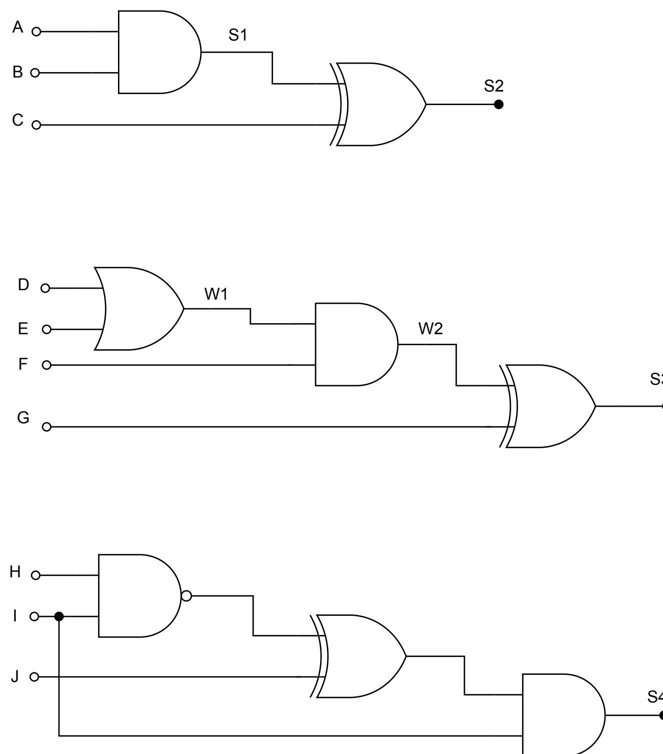


Figura 6.2. Circuitos combinacionales de la práctica 2.

1. Elabore las tablas de verdad de los circuitos correspondientes. Esto permite comprobar fácilmente la operación de los circuitos una vez que estén funcionando en la tarjeta.

2. Estudie los ejemplos 2, 3 y 4 de la sección 4.2 que utilizan las instrucciones «*input*», «*output*», «*assign*» y los operadores lógicos a nivel de bit de la sección 4.4.1.

3. Realice el programa «Práctica_2» en Verilog (ver paso 14 de la sección 3.4.2). Observe que los circuitos se realizan de tres formas distintas: S1 es una salida explícita en el primer circuito y, por lo tanto, requiere un recurso de la tarjeta, de igual modo que S2. En el segundo circuito, las salidas de las compuertas intermedias se conectan vía un cable interno (variables del tipo wire) a las compuertas siguientes, por lo que no requieren recursos de la tarjeta; solo S3 requiere recursos. En el tercer circuito, con la programación de una sola función se obtiene la realización del circuito en su totalidad. Estas opciones están disponibles en Verilog para que el diseñador las utilice como prefiera.

```
module Practica_02(                    // Configuración de 3 circuitos combinacionales.

    input A,B,C,D,E,F,G,H,I,J,         // Entradas a las compuertas (switches).
    output S1,S2,S3,S4);               // Salidas de las compuertas (leds).

    wire W1, W2;                       // Salidas que no requieren recursos de la Basys-3
                                       // (son sólo cables).
    assign S1 = A & B;                 // Primer circuito de la figura 6.2.
    assign S2 = S1 ^ C;

    assign W1 = D | E;                 // Segundo circuito de la figura 6.2.
    assign W2 = W1 & F;
    assign S3 = W2 ^ G;

    assign S4 = (~(H & I) ^ J) & I; // Tercer circuito de la figura 6.2.

endmodule
```

4. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.3).

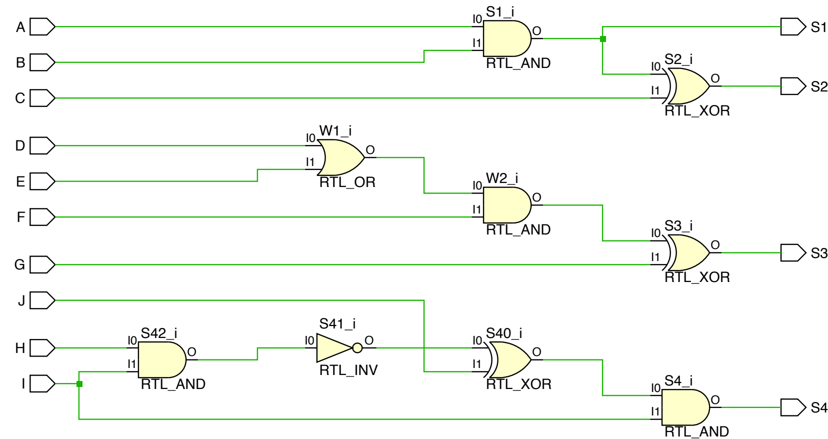


Figura 6.3. Esquemático de la práctica 2.

5. Ejecute los pasos 16 al 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {A}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {B}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {C}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {D}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {E}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {F}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {G}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {H}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {I}]
set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {J}]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {sw10}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {S1}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {S2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {S3}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led5}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {S4}]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led7}]
```

6. Ejecute los pasos 19 al 28 de las secciones 3.4.4 y 3.4.5, para sintetizar, implementar, generar el *bitstream* y traspasar el programa a la tarjeta.

7. Opere los *switches* asignados en el punto 5 y observe cómo se encienden los leds correspondientes. Compare el funcionamiento con las tablas de verdad solicitadas en el punto 1.

6.2.3. PRÁCTICA 3: USO DE ENTRADAS Y SALIDAS EXTERNAS POR LAS PUERTAS PMOD

Esta práctica tiene como propósito experimentar el uso de las puertas Pmod para interactuar con el medio exterior, ingresando y produciendo señales digitales a ese medio (ver sección 2.1). El circuito a implementar se muestra en la figura 6.4. Las acciones de entrada se realizan con un pulsador externo conectado a la entrada JC3 y un pulsador propio de la tarjeta. La salida externa es por la puerta JA3 aplicada sobre un led externo (LedR) y la salida interna a un led propio de la tarjeta. Las tierras se conectan a los puertos GND de la tarjeta.

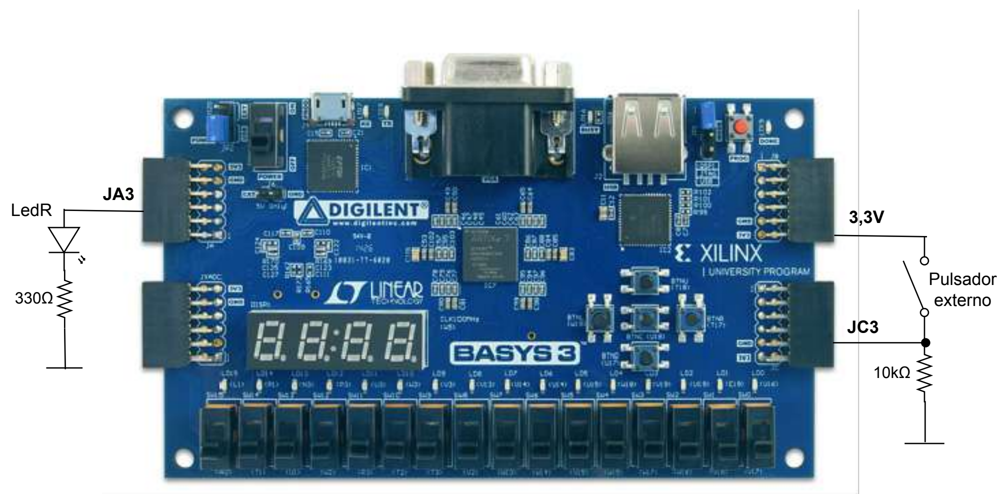


Figura 6.4.
Uso de las puertas Pmod.

1. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module Practica_03(                    // Entradas y salidas externas usando puertas Pmod.

    input Puls_Int, Puls_Ext,          // Entradas vía pulsadores interno y externo.
    output Led_Int, LedR );            // Salidas a leds interno y externo (LedR).

    assign Led_Int = Puls_Int;         // El pulsador interno enciende el led interno.
    assign LedR = Puls_Ext;            // El pulsador externo enciende el LedR (externo).

endmodule
```

2. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.5).

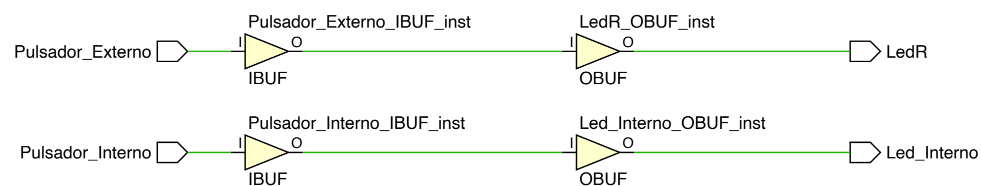


Figura 6.5. Esquemático de la práctica 3.

3. Ejecute los pasos 16 al 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {Led_Int}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]

##Buttons
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {Puls_Int}]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]

##Pmod Header JA
#set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {JA1}]
#set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports {JA2}]
set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {LedR}]
#set_property -dict { PACKAGE_PIN G2   IOSTANDARD LVCMOS33 } [get_ports {JA4}]

##Pmod Header JC
#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports {JC1}]
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {JC2}]
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {Puls_Ext}]
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports {JC4}]
```

4. Ejecute los pasos 19 al 28 de las secciones 3.4.4 y 3.4.5, para sintetizar, implementar, generar el *bitstream* y traspasar el programa a la tarjeta (arme antes el circuito de la figura 6.4).

5. Opere los pulsadores y compruebe el funcionamiento de los leds respectivos.

6.2.4. PRÁCTICA 4: APLICACIÓN DE CONTROL CON CIRCUITOS COMBINACIONALES

Esta experiencia resume las prácticas vistas hasta ahora con la implementación de una aplicación de control de la operación de la máquina expendedora de café que se muestra en la figura 6.6 (la máquina ha sido simplificada para este ejemplo).

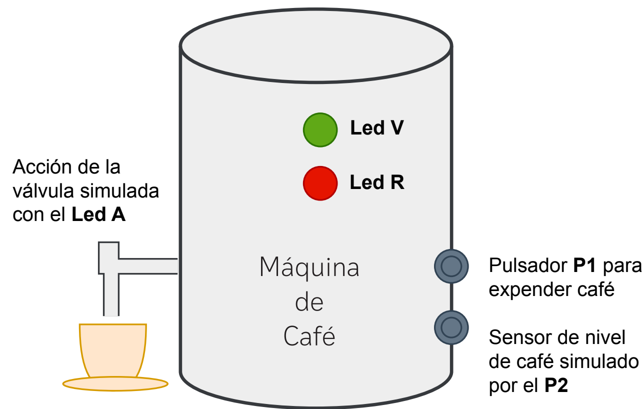


Figura 6.6. Máquina expendedora de café.

El sistema de control debe interactuar con los elementos de la máquina cumpliendo las siguientes especificaciones:

- Una válvula electromagnética simulada con el led A se activa para proveer café cuando se acciona un pulsador P1 (cuando pasa de «0» a «1»).
- La máquina cuenta con un sensor de nivel de café, simulado con el pulsador P2, que está en «0» cuando el nivel es suficiente y en «1» cuando el nivel es insuficiente.
- Un indicador luminoso led V está encendido con un «1» cuando la máquina contiene suficiente café en su interior y está en condiciones de operar correctamente. Este es un requisito para que el pulsador P1 pueda activar la válvula de café.
- Un indicador luminoso led R se enciende con un «1» cuando el nivel de café es insuficiente. Esta es una indicación de alarma que, además de encender el led R, debe apagar el led V e impedir que el pulsador P1 accione la válvula (en esta condición, aunque se accione el pulsador P1, el led A no debe encenderse).

1. Analise si la tabla de verdad (tabla 6.1) cumple con las especificaciones del sistema de control.

Tabla de verdad 6.1. Sistema de control de una máquina de café.

Entradas		Salidas		
Pulsador para solicitar café (P1)	Sensor de Nivel de Café (P2)	Indicador Máquina OK (LedV)	Válvula abre o cierra para extraer Café (LedA)	Indicador Máquina con insuficiente Café (LedR)
0	0	1	0	0
0	1	0	0	1
1	0	1	1	0
1	1	0	0	1

2. Determine un circuito combinacional que cumpla con la tabla 6.1. Es decir, aquel que realice las funciones booleanas para cada led de salida (se puede evaluar una mayor simplificación entre ellas usando mapas de Karnaugh):

Led V = P1’ * P2’ + P1 * P2’
Led A = P1 * P2’
Led R = P1’ * P2 + P1 * P2

3. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module control_maquina_cafe(            // Sistema de control de una máquina de café.
    input P1, P2,
    output LedV, LedA, LedR );

    assign LedV = ~P1&~P2 | P1&~P2;
    assign LedA =  P1&~P2;
    assign LedR = ~P1&P2 | P1&P2;

endmodule
```

4. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.7).

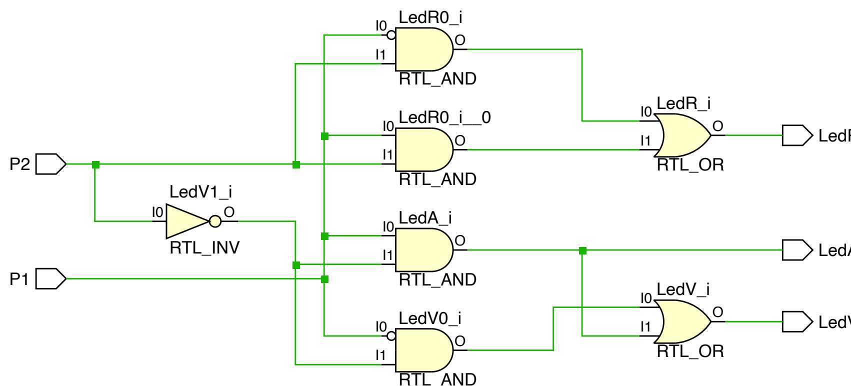


Figura 6.7. Esquemático de la práctica 4.

5. Para la realización del sistema de control, implemente el circuito de la figura 6.8. Obtenga los voltajes de 3,3 volts DC y las tierras de la misma tarjeta.

Figura 6.8. Montaje del circuito de control.

6. Ejecute los pasos 16 a 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
##Pmod Header JA
set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {LedA}]; #JA1
set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports {LedV}]; #JA2
set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {LedR}]; #JA3
#set_property -dict { PACKAGE_PIN G2   IOSTANDARD LVCMOS33 } [get_ports {JA4}]

##Pmod Header JB
set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports {P2}]; #JB1
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {JB2}]

##Pmod Header JC
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports {P1}]; #JC1
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {JC2}]
```

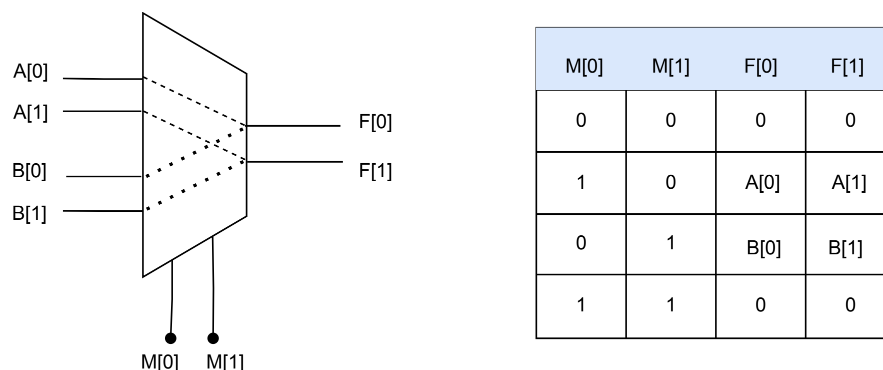
7. Ejecute los pasos 19 a 28 de las secciones 3.4.4 y 3.4.5 para sintetizar, implementar, generar el *bitstream* y traspasar el programa a la tarjeta (arme antes el circuito de la figura 6.8 considerando la distribución de pines de la tarjeta BASYS-3 que se muestra en el capítulo 2, figura 2.2).

8. Opere los pulsadores y compruebe el funcionamiento de los leds de acuerdo con la tabla 6.1.

6.2.5. PRÁCTICA 5: MULTIPLEXOR DE DOS CANALES

Consiste en diseñar e implementar un multiplexor de dos canales y dos entradas cada uno, como el que muestra la figura 6.9. Este funciona de la siguiente manera:

- Las salidas F[0] y F[1] son iguales a las entradas A[0] y A[1] o B[0] y B[1], según el valor de la variable de control M que indica la tabla de la figura 6.9.
- Así, por ejemplo, si M[0] = 1 y M[1] = 0, entonces F[0] = A[0] y F[1] = A[1].



M[0]	M[1]	F[0]	F[1]
0	0	0	0
1	0	A[0]	A[1]
0	1	B[0]	B[1]
1	1	0	0

Figura 6.9. Multiplexor de 2 canales.

1. Estudie el bloque procedimental «always» de la sección 4.2.3, los «arreglos» de la sección 4.3 y las sentencias «case» y «endcase» de la sección 4.6.2.

2. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module mux_2pines(           // Configuración de un multiplexor de buses de 2 pines.
    input [1:0] A, B,        // A y B son las entradas al multiplexor, como buses de 2 pines.
    input [1:0] M,           // M es el selector del multiplexor.
    output reg [1:0] F );    // F es un bus de salida de 2 pines. Se declara como registro
                             // (reg) debido a que se va a usar dentro de un always.
    always @(A, B, M)        // El always se requiere cada vez que se usa la sentencia case.
        case(M)              // La instrucción case puede entenderse como un "if múltiple".
        2'b00: F = 2'b00;    // Si M es 00, entonces F toma el valor 00 binario.
        2'b01: F = B;        // Si M es 01, entonces F toma el valor del bus B.
        2'b10: F = A;        // Si M es 10, entonces F toma el valor del bus A.
        2'b11: F = 2'b00;    // Si M es 11, entonces F toma también el valor 00 binario.
        endcase

endmodule
```

3. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.10).

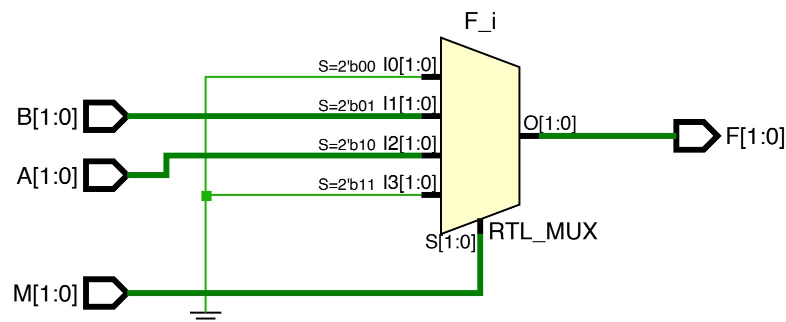


Figura 6.10. Esquemático del multiplexor de 2 canales.

4. Ejecute los pasos 16 al 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación (observe que las variables A, B, F y M se escriben como arreglos):

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {B[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {B[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {A[0]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {A[1]}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {sw4}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {sw5}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {M[0]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {M[1]}]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {sw8}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw9}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {F[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {F[1]}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
```

5. Sintetice, implemente, genere el *bitstream*, cargue el programa, opere los *switches* y compruebe el funcionamiento de los leds respectivos de acuerdo con la tabla de la figura 6.9.

6.2.6. PRÁCTICA 6: DECODIFICADOR BCD A SIETE SEGMENTOS

El propósito de esta práctica es utilizar uno de los *displays* de 7 segmentos de la tarjeta para representar la decodificación BCD de un código binario producido por 4 *switches* (decodificador BCD a 7 segmentos). Esto implica que, por ejemplo, si con los *switches* se genera un «0101» en binario, el display debe mostrar un «5». Si se genera un «1000», el *display* debe mostrar un «8», y así cualquier combinación entre 0 y 9.

1. Estudie el bloque procedimental «always» de la sección 4.2.3, los «arreglos» de la sección 4.3 y el operador de propósitos especiales «?:» de la sección 4.4.5.

2. Tenga presente que el *display* de la tarjeta opera con +Vcc en modalidad ánodo común (en esta condición todos los segmentos están apagados), por lo que para encender un segmento debe aplicarse un «0» en el cátodo de dicho segmento, según la relación que indica la figura 6.11. Además, considere que existen 4 *displays* en la tarjeta y que cada uno se habilita con un «0» (ver la variable «display» del programa del punto 3).

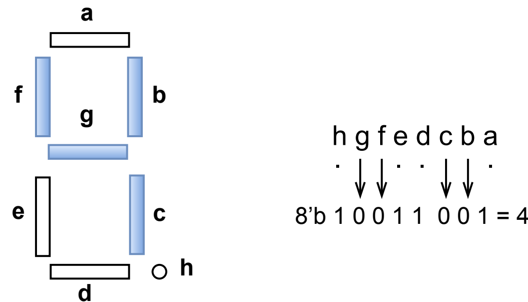


Figura 6.11. Segmentos en un display de la BASYS-3.

3. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module bcd_7_segmentos(      // Decodificador BCD a 7 segmentos.

  input [3:0] ent,           // Entrada tipo bus de 4 pines (4 bits ).
  output [7:0] seg,          // Salida a 8 segmentos de un display (7 más el punto).
  output [3:0] dis );        // Selector del display( se selecciona con 4 bits).

  assign seg= (ent == 4'b0000) ? 8'b11000000: // Si "ent" es 0000, entonces "seg" es 11000000.
              (ent == 4'b0001) ? 8'b11111001: // Si "ent" es 0001, "seg" es 10000110 (el n°1).
              (ent == 4'b0010) ? 8'b10100100: // Si "ent" es 0010, "seg" es 11011011 (el n°2).
              (ent == 4'b0011) ? 8'b10110000: // Si "ent" es 0011, "seg" es 10110000 (el n°3).
              (ent == 4'b0100) ? 8'b10011001: // N°4.
              (ent == 4'b0101) ? 8'b10010010: // N°5.
              (ent == 4'b0110) ? 8'b10000011: // N°6.
              (ent == 4'b0111) ? 8'b11111000: // N°7.
              (ent == 4'b1000) ? 8'b10000000: // N°8.
              (ent == 4'b1001) ? 8'b10011000: // N°9.
              8'b10000110;    // Si no se cumple ninguna de las anteriores, seg es "E" (error).

  assign dis = 4'b1110;       // El número se muestra en el primer display de la derecha.

endmodule
```

4. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.12). Utilice el zoom «+/−» para ver detalles del circuito.

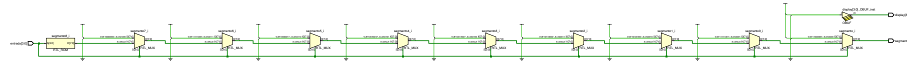


Figura 6.12. Esquemático del circuito BCD a 7 segmentos.

5. Ejecute los pasos 16 a 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación (observe que las variables «entrada», «segmento» y «display» se escriben como arreglos):

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {ent[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {ent[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {ent[2]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {ent[3]}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {sw4}]


##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {seg[0]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {seg[1]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {seg[2]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {seg[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {seg[4]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {seg[5]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {seg[6]}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {seg[7]}]

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {dis[0]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {dis[1]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {dis[2]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {dis[3]}]
```

6. Sintetice, implemente, genere el *bitstream*, cargue el programa, opere los *switches* y compruebe la correcta generación de los dígitos 0 a 9. Si es mayor a 9, debe aparecer «error» (la letra «E»).

6.2.7. PRÁCTICA 7: TRANSMISIÓN DE ESTADOS DIGITALES CON MULTIPLEXOR Y DEMULTIPLEXOR

Esta práctica tiene por objetivo implementar un sistema de enrutamiento de estados digitales básicos («1» o «0») con un multiplexor y demultiplexor dispuesto como se muestra en la figura 6.13. Este sistema debe funcionar de la siguiente manera:

- Las entradas «In_Mux» al multiplexor se seleccionan mediante el «Select_Mux», de modo que solo la entrada seleccionada se transfiera a la salida «Out_Mux». Observe que el multiplexor tiene 8 entradas, por lo que el «Select_Mux» tiene 3 bits de control.

- Por otra parte, de manera inversa, la entrada «In_Demux» al demultiplexor se transfiere a una de las 8 salidas «Out_Demux» (se usan 8 leds), según la información de control del «Select_Demux», que también es de 3 bits. Nótese que la salida «Out_Mux» se reinserta a la tarjeta como «In_Demux».

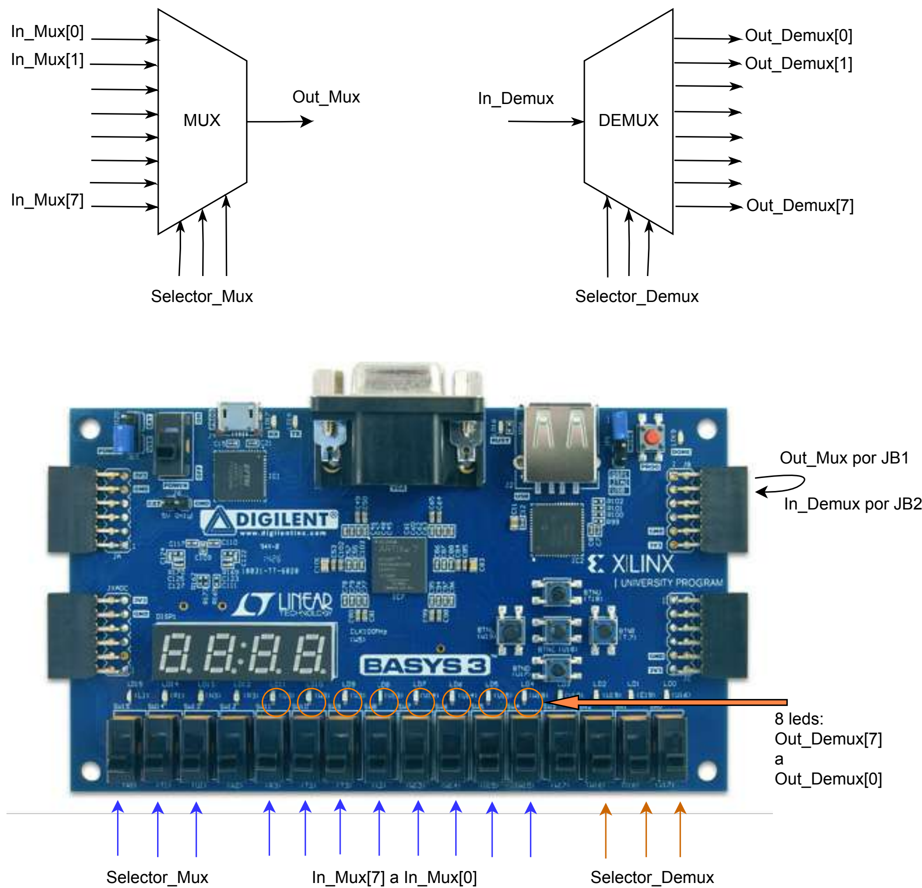


Figura 6.13.
Circuitos multiplexor y demultiplexor con la BASYS-3/FPGA.

1. Estudie el bloque procedimental «always» de la sección 4.2.3, los «arreglos» de la sección 4.3 y la sentencia «case» de la sección 4.6.2.

2. Realice el siguiente programa en Verilog (ver paso 14 de la sección 3.4.2):

```
module mux_and_demux(                       // Circuitos multiplexor y demultiplexor.
    input [7:0] In_Mux,                     // Entradas al Mux (8 switches).
    input [2:0] Select_Mux,                 // Selector del Mux (control vía 2 switches).
    output reg Out_Mux,                     // Salida del Mux (por JB1).
    input In_Demux,                         // Entrada al Demux (por JB2).
    input [2:0] Select_Demux,               // Selector del Demux (control vía 2 switches).
    output reg [7:0] Out_Demux );           // Salida del Demux (8 leds).

    always@ (Select_Mux, In_Mux)             // Circuito multiplexor.
        case(Select_Mux)
            3'b000: Out_Mux = In_Mux[0];
            3'b001: Out_Mux = In_Mux[1];
            3'b010: Out_Mux = In_Mux[2];
            3'b011: Out_Mux = In_Mux[3];
            3'b100: Out_Mux = In_Mux[4];
            3'b101: Out_Mux = In_Mux[5];
            3'b110: Out_Mux = In_Mux[6];
            3'b111: Out_Mux = In_Mux[7];
        endcase
```

```
    always@ (Select_Demux, In_Demux)      // Circuito demultiplexor.
        case(Select_Demux)
            3'b000: Out_Demux[0] = In_Demux;
            3'b001: Out_Demux[1] = In_Demux;
            3'b010: Out_Demux[2] = In_Demux;
            3'b011: Out_Demux[3] = In_Demux;
            3'b100: Out_Demux[4] = In_Demux;
            3'b101: Out_Demux[5] = In_Demux;
            3'b110: Out_Demux[6] = In_Demux;
            3'b111: Out_Demux[7] = In_Demux;
        endcase
endmodule
```

3. Ejecute el paso 15 de la sección 3.4.2 para obtener el esquemático del circuito diseñado (figura 6.14).

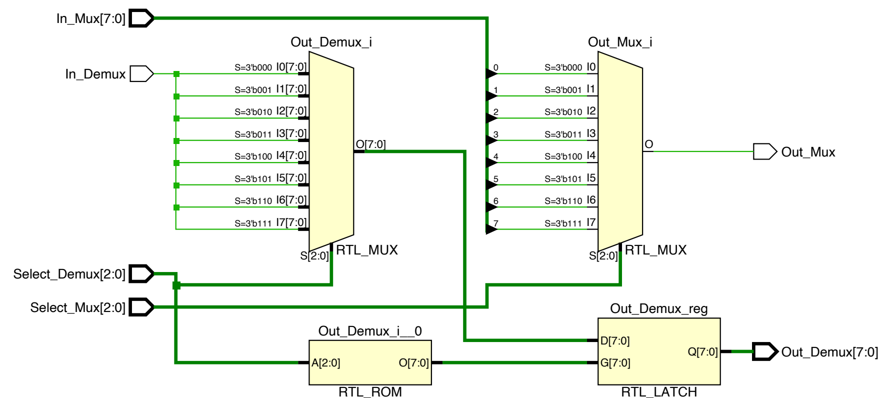


Figura 6.14. Esquemático de los circuitos multiplexor y demultiplexor.

4. Ejecute los pasos 16 al 18 de la sección 3.4.3 para asignar los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {Select_Demux[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {Select_Demux[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {Select_Demux[2]}]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw3}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {In_Mux[0]}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {In_Mux[1]}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {In_Mux[2]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {In_Mux[3]}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {In_Mux[4]}]
set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {In_Mux[5]}]
set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {In_Mux[6]}]
set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {In_Mux[7]}]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {sw12}]
set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {Select_Mux[0]}]
set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {Select_Mux[1]}]
set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {Select_Mux[2]}]
```

```
## LEDs
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led0}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[0]}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[1]}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[2]}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[3]}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[4]}]
set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[5]}]
set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[6]}]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {Out_Demux[7]}]
#set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {led12}]

##Pmod Header JB
set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports {Out_Mux}];  #JB1
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {In_Demux}]; #JB2
#set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports {JB3}]
```

5. Sintetice, implemente, genere el *bitstream*, cargue el programa, opere los *switches* y compruebe el correcto funcionamiento tanto del circuito multiplexor como del demultiplexor.

6.2.8. PRÁCTICA 8: *FLIP FLOPS*

Esta práctica tiene el propósito de comprender la implementación y el funcionamiento de los circuitos secuenciales básicos. Es decir, los *flip flops* tipo D, T y J-K. Los diagramas de conexión y sus respectivas tablas de verdad se muestran en las figuras 6.15, 6.16 y 6.17. La implementación de los circuitos requiere del módulo de instanciación «divisor_frecuencia» para proveer las señales de reloj (clk) (ver la práctica 13 del capítulo 7).

Clk	D	Q	Q’
↑	0	0	1
	1	1	0

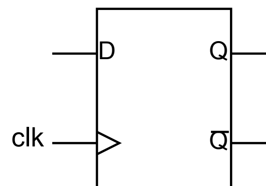


Figura 6.15. Diagrama y tabla de verdad del *flip flop* tipo D.

Clk	T	Q	Q’
↑	0	Q_{n-1}	Q'_{n-1}
	1	Q'_{n-1}	Q_{n-1}

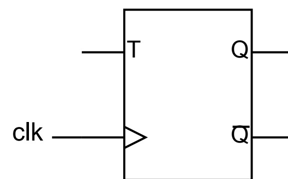


Figura 6.16. Diagrama y tabla de verdad del *flip flop* tipo T.

preset	clear	clk	J	K	Q	Q’
0	0	↑	0	0	Q_{n-1}	Q'_{n-1}
0	0	↑	0	1	0	1
0	0	↑	1	0	1	0
0	0		1	1	Q'_{n-1}	Q_{n-1}
1	0	X	X	X	1	0
0	1	X	X	X	0	1

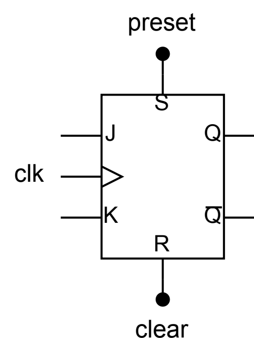


Figura 6.17. Diagrama y tabla de verdad del *flip flop* tipo J-K.

1. Realice el siguiente programa en Verilog. Nótese que el módulo «divisor_frecuencia» es el que provee un reloj de frecuencia de 1 Hz solo para visualizar de mejor forma la operación de los *flip flops*. Esto significa que en una aplicación normal puede usar directamente el reloj de 100 MHz de la tarjeta.

```
module flip_flops_D_T_JK(          // Programa que implementa los flip flops D, T y J-K.

    input wire clk,                // Corresponde al reloj interno de 100MHz de la tarjeta.
    input D, T, J, K,              // Entradas a los 3 flip flops (switches).
    input preset, clear,           // Define el preset y clear del FF J-K (botones).
    output reg Qd, Qt, Qjk,        // Salidas Q de los 3 FF (leds).
    output QdNeg, QtNeg, QjkNeg,   // Salidas negadas de los 3 FF (leds).
    output Led_1Hz );              // Led que se ilumina cada 1 seg, con el canto de subida
                                   // del reloj.
    wire reloj_1Hz;                // Variable interna que recibe el reloj de 1Hz desde el
                                   // módulo divisor_fecuencia.
    divisor_frecuencia(            // Instanciación al módulo divisor_frecuencia.
            .reloj_in(clk),        // Traspasa el clk de 100MHz a la variable clk_in.
            .reloj_out(reloj_1Hz)); // Recibe en la variable clk_out el reloj de 1Hz desde
                                   // el módulo divisor_frecuencia y lo traspasa a clk_1Hz.
    assign QdNeg = ~Qd;            // Asigna a QdNeg el valor de Qd negado.
    assign QtNeg = ~Qt;            // Asigna a QtNeg el valor de Qt negado.
    assign QjkNeg = ~Qjk;          // Asigna a QjkNeg el valor de Qjk negado.

    assign Led_1Hz = reloj_1Hz;    // Para visualizar la pulsación del reloj cada 1 seg.
    always@ (posedge reloj_1Hz)    // Implementa el flip flop D (cambia de estado con
                                   // cada canto de subida del reloj).
        case( D )                  // Implementa la operación del FF según el valor de D.
            1'b0: Qd <= 0;         // Si D es 0, Qd toma el valor 0.
            1'b1: Qd <= 1;         // Si D es 1, Qd toma el valor 1.
        endcase

    always@ (posedge reloj_1Hz)    // Implementa el flip flop T.
        case( T )                  // Implementa la operación del FF según el valor de T.
            1'b0: Qt <= Qt;        // Si T es 0, Qt adopta el valor anterior.
            1'b1: Qt <= QtNeg;     // Si T es 1, Qt adopta el valor anterior complementado.
        endcase

    always@ (posedge reloj_1Hz)            // Implementa el flip flop JK.
        if (preset)
            Qjk = 1'b1;
        else
            if (clear)
                Qjk = 1'b0;
            else
            case( {J, K} )                 // Implementa la operación del FF según J y K.
                 2'b00: Qjk <= Qjk;        // Si JK son 00, Q conserva su valor (no cambia).
                 2'b01: Qjk <= 0;          // Si JK son 01, Q se "limpia" y va a 0.
                 2'b10: Qjk <= 1;          // Si JK son 10, Q toma el valor 1.
                 2'b11: Qjk <= QjkNeg;     // Si JK son 11, Q toma el valor anterior
            endcase                        // complementado.
endmodule

module divisor_frecuencia(                 // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                   // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );               // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;          // Asigna el valor de 100M a una constante Seg1
                                           // para producir 1 seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);       // Guarda en Nbits el número de bits necesarios
                                           // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0; // Define el registro contador_divisor para
                                           // utilizarlo como contador y lo inicializa en 0.
```

```
    always@(posedge reloj_in)              // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
        if (contador_divisor1 == Seg1-1)   // Si el contador_divisor llega al valor máximo, lo
            contador_divisor1 <= 0;        // deja en 0 (para iniciar nuevo conteo).
        else
            contador_divisor1 <= contador_divisor1 + 1; // Si el contador_divisor no ha llegado
                                                        // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];      // Asigna al reloj_out el valor más
                                                        // significativo del registro
endmodule                                               // contador_divisor (frec. de 1Hz).
```

2. Obtenga el esquemático del circuito resultante (figura 6.18).

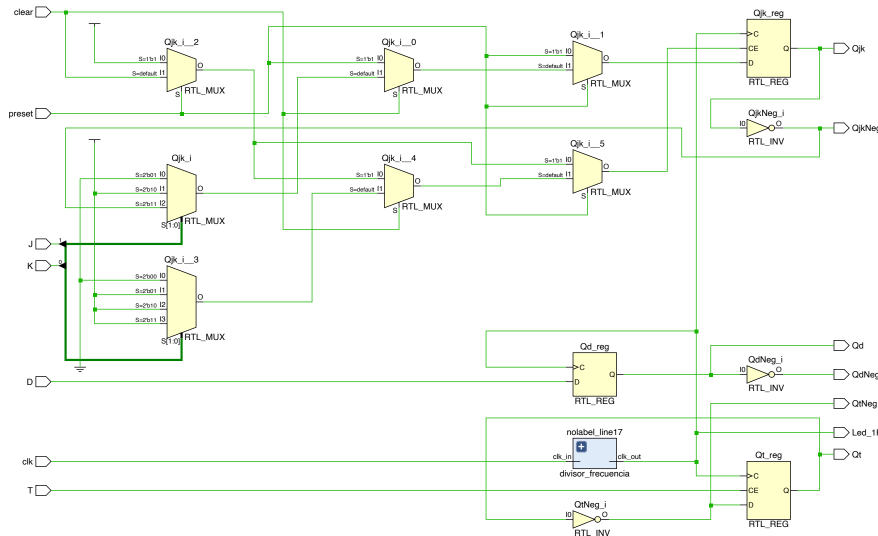


Figura 6.18. Esquemático del circuito flip flops tipo D, T y J-K.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

## Switches (entradas)
set_property -dict {PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {K}]
set_property -dict {PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {J}]
#set_property -dict {PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {sw2}]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw3}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {T}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {sw5}]
#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {sw6}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {D}]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {sw8}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw9}]

## LEDs (salidas)
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {QjkNeg}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {Qjk}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {QtNeg}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {Qt}]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led6}]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led7}]
```

```
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {QdNeg}]
set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {Qd}]
#set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {led10}]
#set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {led11}]
#set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {led12}]
#set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {led13}]
#set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {led14}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {Led_1Hz}]

##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {btnC}]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {bntU}]
set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports {clear}]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {preset}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {btnD}]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe el correcto funcionamiento de cada uno de los *flip flops*.

6.2.9. PRÁCTICA 9: REGISTROS DE DESPLAZAMIENTO

Los registros de desplazamiento se realizan con un conjunto de *flip flops* en aplicaciones de sistemas digitales que requieren almacenar y transferir datos. La capacidad de almacenar de un registro lo convierte en un dispositivo de memoria que tiene tantos bits como el número de *flip flops* que se utilicen en su implementación. Tanto el almacenamiento como la transferencia se pueden realizar en paralelo o en serie. Esta práctica tiene el propósito de comprender la implementación y el funcionamiento de registros de carga en paralelo y desplazamiento en serie a la derecha, como se muestra en la figura 6.19, bajo las siguientes condiciones de operación:

- El reloj (clk) es provisto por un módulo «divisor_frecuencia» ralentizado a 1 Hz para observar de mejor forma los cambios de estado de los *flip flops* y el desplazamiento propiamente tal (ver la práctica 13 del capítulo 7).

- El registro de desplazamiento se realiza con 4 *flip flops* tipo D (ver práctica 8) para recibir un conjunto de 4 bits en paralelo (A, B, C y D).

- Una lógica circuital complementaria se adiciona para establecer que con un «1» en la señal de «control» se apliquen (se carguen) los 4 bits a cada una de las entradas de los *flip flops*, y con un «0» comience el desplazamiento hacia la derecha.

- El desplazamiento a la derecha «desaparece» después de alcanzar el último *flip flop* (FFD). Por lo tanto, el lector puede, por ejemplo, variar el programa para producir una salida por una puerta Pmod después del FFD y reinyectarla por otra puerta en lugar del bit A (al FFA). Esto permite observar el desplazamiento de manera circular e indefinida. También puede ensayar la interconexión de la tarjeta con otro desarrollador para extender la trayectoria del desplazamiento.

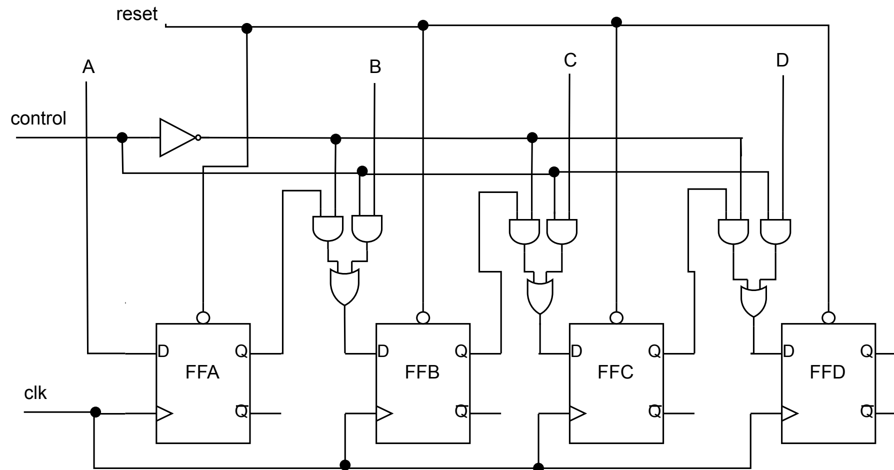


Figura 6.19. Registro de desplazamiento de 4 bits.

1. Realice el siguiente programa en Verilog. Nótese que el módulo «divisor_frecuencia» provee un reloj ralentizado de frecuencia de 1 Hz para visualizar de mejor forma la operación del registro:

```
module registro_desplazamiento(     // Registro de desplazamiento de 4 bits.
    input control,                  // Switch para activar la carga/desplazamiento de los 4 FF.
    input A, B, C, D,               // Switches que conforman la data que se carga en los FF.
    input wire clk,                 // clk es el reloj interno de 100 MHz de la tarjeta.
    input reset,                    // Botón para producir el reseteo del desplazamiento.
    output reg Da, Db, Dc, Dd,      // Leds para visualizar las entradas a los FF.
    output reg Qa, Qb, Qc, Qd,      // Leds correspondientes a las salidas de los FF.
    output Led_1Hz );               // Led para observar que el circuito funciona al ritmo del
                                    // reloj de 1Hz.
    wire reloj_1Hz;                 // Variable a la que se asigna el reloj de 1Hz.

    divisor_frecuencia(             // Instanciación (llamado) al módulo divisor_frecuencia.
            .reloj_in(clk),         // Traspasa el reloj interno de 100 MHz al reloj_in.
            .reloj_out(reloj_1Hz)); // El reloj_out retorna con el valor de 1 Hz y lo
                                    // traspasa al reloj de 1Hz.
    assign Led_1Hz = reloj_1Hz;

    always @ ( posedge reloj_1Hz, posedge reset ) // Bloque de carga y desplazamiento (4 bits).
        if (reset) begin                          // Si se activa el reset, pone las salidas
            Qa = 1'b0;                            // de los FF en 0.
            Qb = 1'b0;
            Qc = 1'b0;
            Qd = 1'b0;
        end

       else begin                   // Con "control" en 1 carga las entradas A,B,C,D
                                    // y con "control" en 0 activa el desplazamiento.
           Da <= A;                              // Entrada al FFA (según su fn transferencia).
           Db <= ~control & Qa | control & B;    // Entrada al FFB (según su fn transferencia).
           Dc <= ~control & Qb | control & C;    // Entrada al FFC (según su fn transferencia).
           Dd <= ~control & Qc | control & D;    // Entrada al FFD (según su fn transferencia).

           case(Da)                 // FFA
               1'b0: Qa <= 0;
               1'b1: Qa <= 1;
           endcase

           case(Db)                 // FFB
               1'b0: Qb <= 0;
               1'b1: Qb <= 1;
           endcase
```

```
            case(Dc)                    // FFC
                1'b0: Qc <= 0;
                1'b1: Qc <= 1;
            endcase

            case(Dd)                    // FFD
                1'b0: Qd <= 0;
                1'b1: Qd <= 1;
            endcase
        end
endmodule

module divisor_frecuencia(                  // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                    // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );                // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;           // Asigna el valor de 100M a una constante Seg1
                                            // para producir 1 seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);        // Guarda en Nbits el número de bits necesarios
                                            // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0;  // Define el registro contador_divisor para
                                            // utilizarlo como contador y lo inicializa en 0.

    always@(posedge reloj_in)               // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
        if (contador_divisor1 == Seg1-1)    // Si el contador_divisor llega al valor máximo, lo
            contador_divisor1 <= 0;         // deja en 0 (para iniciar nuevo conteo).
        else
            contador_divisor1 <= contador_divisor1 + 1; // Si el contador_divisor no ha llegado
                                                         // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];       // Asigna al reloj_out el valor más
                                                         // significativo del registro
endmodule                                                // contador_divisor (frec. de 1Hz).
```

2. Obtenga el esquemático del circuito resultante de la figura 6.20 (aplique zoom «+/−» para ver detalles).

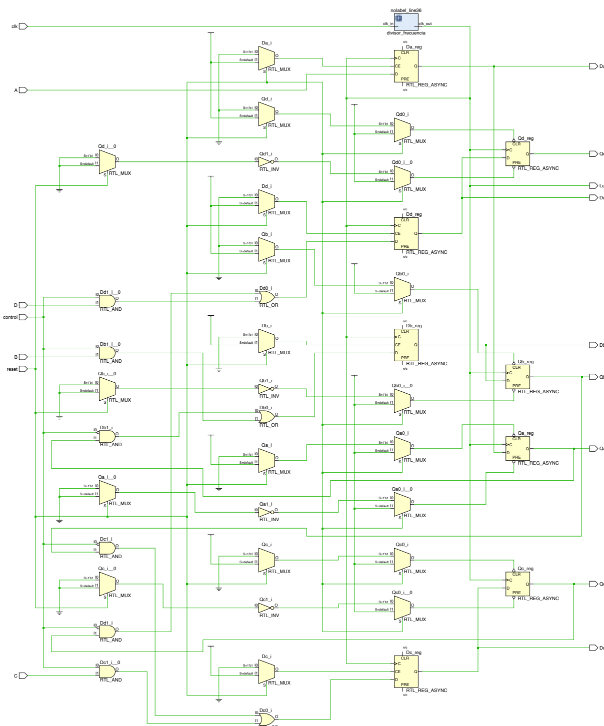


Figura 6.20. Esquemático del circuito de registro de desplazamiento.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]


## Switches
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {sw0}]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {sw1}]
#set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {sw2}]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw3}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {sw4}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {sw5}]
#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {sw6}]
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {sw7}]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {sw8}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw9}]
set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {D}]
set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {C}]
set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {B}]
set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {A}]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {sw14}]
set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {control}]

## LEDs
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led0}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led3}]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led4}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {Qd}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {Qc}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {Qb}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {Qa}]
#set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led9}]
set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {Dd}]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {Dc}]
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {Db}]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {Da}]
#set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {led14}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {Led_1Hz}]


##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {reset}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe el correcto desplazamiento de los 4 bits (A, B, C y D).

6.2.10. PRÁCTICA 10: CONTADORES

Esta práctica tiene el propósito de comprender la implementación y el funcionamiento de los contadores asincrónicos. En este ejemplo, un conjunto de 4 *flip flops* del tipo «T» se interconectan para estructurar un contador binario (visualización hexadecimal), de manera tal que solo el *flip flop* correspondiente al dígito LSB (primer *flip flop*) recibe el reloj (clk_1Hz) y los siguientes reciben como reloj la salida negada del flip flop anterior (figura 6.21). Para visualizar apropiadamente la operación, el circuito utiliza el módulo de instanciación «divisor frecuencia» que provee un reloj de 1 Hz (ver la práctica 13 del capítulo 7).

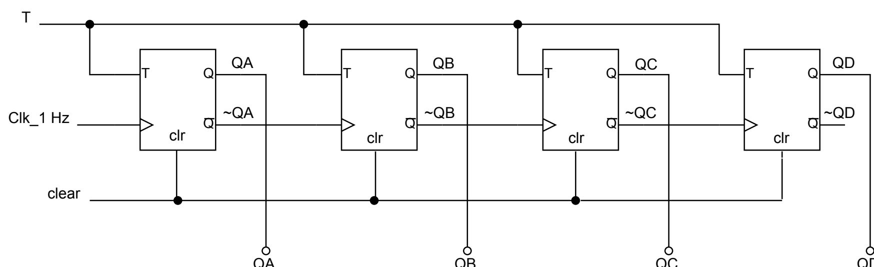


Figura 6.21.
Contador asincrónico de 4 bits.

1. Realice el siguiente programa en Verilog. Nótese que el módulo «divisor_frecuencia» es el que provee el reloj de frecuencia de 1 Hz solo para visualizar de mejor forma la operación de los *flip flops*. Esto significa que en una aplicación normal se puede usar directamente el reloj de 100 MHz de la tarjeta.

```
module contador_asincronico_4ff(          // Contador asincrónico con 4 flip flops tipo T.
    input T,                              // Entrada común a los 4 flip flops.
    input clk, clear,                     // clk es el reloj de 100MHz de la tarjeta.
    output reg [7:0] seg,                 // Para visualizar la cuenta en un display.
    output reg QA, QB, QC, QD,            // Para visualizar la cuenta en 4 leds.
    output [3:0] dis );                   // Para seleccionar un display de la tarjeta.

    wire reloj_1Hz ;

    divisor_frecuencia(                   // Instanciación (llamado) al módulo divisor_frecuencia.
            .reloj_in(clk),               // Traspasa el reloj interno de 100 MHz a reloj_in.
            .reloj_out(reloj_1Hz) );      // El reloj_out retorna con el valor de 1 Hz y lo
                                          // traspasa al reloj de 1Hz.

    always@ (posedge reloj_1Hz, posedge clear)  // Construye el 1er flip flop.
    begin
        if (clear)  QA = 1'b0;                  // Efectua el borrado de la salida del 1er FF.
        else
            case (T)
            1'b0: QA <= QA;
            1'b1: QA <= ~QA;
            endcase
        end

     always@ (posedge ~QA, posedge clear)        // Construye el 2do flip flop.
     begin
        if (clear)  QB = 1'b0;                  // Efectua el borrado de la salida del 2do FF.
        else
            case (T)
                1'b0: QB <= QB;
                1'b1: QB <= ~QB;
            endcase
```

```
    end

    always@ (posedge ~QB, posedge clear)       // Construye el 3er flip flop.
    begin
       if (clear)  QC = 1'b0;                  // Efectua el borrado de la salida del 3er FF.
       else
           case (T)
               1'b0: QC <= QC;
               1'b1: QC <= ~QC;
           endcase
       end

    always@ (posedge ~QC, posedge clear)       // Construye el 4to flip flop.
    begin
       if (clear)  QD = 1'b0;                  // Efectua el borrado de la salida del 4to FF.
       else
           case (T)
               1'b0: QD <= QD;
               1'b1: QD <= ~QD;
           endcase
    end

    always@( { QD, QC, QB, QA} )        // Muestra en el display el valor hex de la cuenta.
          case({QD, QC, QB, QA})        // (Si estuvieran negadas la cuenta sería decreciente.
               4'b0001:seg=8'b11111001;
               4'b0010:seg=8'b10100100;
               4'b0011:seg=8'b10110000;
               4'b0100:seg=8'b10011001;
               4'b0101:seg=8'b10010010;
               4'b0110:seg=8'b10000010;
               4'b0111:seg=8'b11111000;
               4'b1000:seq=8'b10000000;
               4'b1001:seg=8'b10010000;
               4'b1010:seg=8'b10001000;
               4'b1011:seg=8'b10000011;
               4'b1100:seg=8'b11000110;
               4'b1101:seg=8'b10100001;
               4'b1110:seg=8'b10000110;
               4'b1111:seg=8'b10001110;
               default:seg=8'b11000000;
          endcase
          assign dis =4'b1110;                 // Selecciona uno de los 4 displays.
endmodule

module divisor_frecuencia(                  // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                    // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );                // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;           // Asigna el valor de 100M a una constante Seg1
                                            // para producir 1 seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);        // Guarda en Nbits el número de bits necesarios
                                            // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0;  // Define el registro contador_divisor para
                                            // utilizarlo como contador y lo inicializa en 0.

    always@(posedge reloj_in)               // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
        if (contador_divisor1 == Seg1-1)    // Si el contador_divisor llega al valor máximo, lo
            contador_divisor1 <= 0;         // deja en 0 (para iniciar nuevo conteo).
        else
            contador_divisor1 <= contador_divisor1 + 1; // Si el contador_divisor no ha llegado
                                                        // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];      // Asigna al reloj_out el valor más
                                                        // significativo del registro
endmodule                                               // contador_divisor (frec. de 1Hz).
```

2. Obtenga el esquemático del circuito resultante de la figura 6.22 (aplique zoom «+/−» para ver detalles).

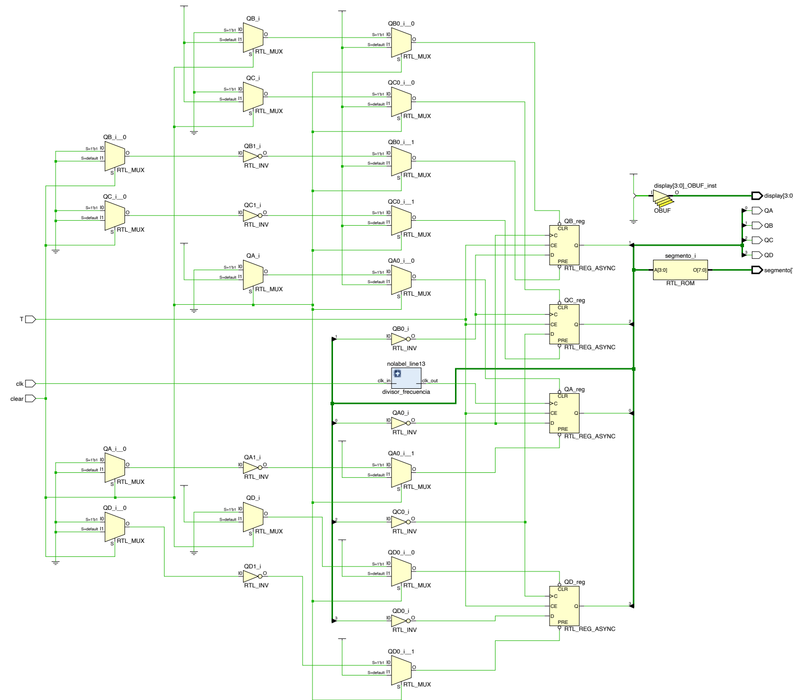


Figura 6.22. Esquemático del circuito contador.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {T}]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {sw1}]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {QA}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {QB}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {QC}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {QD}]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led4}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led5}]
```

```
##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {seg[0]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {seg[1]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {seg[2]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {seg[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {seg[4]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {seg[5]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {seg[6]}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {seg[7]}]

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {dis[0]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {dis[1]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {dis[2]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {dis[3]}]


##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {clear}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe la correcta secuencia de conteo hexadecimal en el *display* y los leds.

CAPÍTULO 7
PRÁCTICAS AVANZADAS

7.1. Introducción

Este capítulo contiene un conjunto de prácticas complementarias de mayor complejidad en cuanto a la configuración de los circuitos y el uso de los recursos de la tarjeta BASYS-3/FPGA. Por lo tanto, se sugiere al lector abordarlas cuando haya experimentado satisfactoriamente las propuestas del capítulo anterior y tenga dominio de la programación con el lenguaje Verilog. Esto, porque si bien es cierto que algunas prácticas pueden considerarse como una extensión simple de las anteriores, otras abordan la utilización de interfaces no tratadas hasta ahora, así como módulos especiales de Vivado utilizados para facilitar la configuración de los circuitos que requieren el conocimiento previo de Verilog.

7.2. Prácticas avanzadas

Para facilitar la implementación por parte del lector, los programas de estas prácticas avanzadas se presentan en un formato que puede ser copiado y traspasado directamente a Vivado.

7.2.1. PRÁCTICA 11: SUMADOR DE 4 BITS

Esta práctica define el funcionamiento de un sumador de 4 bits. Para tal efecto, se utiliza la descripción jerárquica del lenguaje Verilog (ver sección 4.9), que permite definir una instanciación sumadora de un bit (figura 7.1), la que luego es invocada cuatro veces para lograr el sumador de 4 bit (figura 7.2). Las entradas se establecen con *switches* y se visualizan en los leds internos de la tarjeta. Del mismo modo, la operación resultante se visualiza en los leds internos, tanto el resultado de las sumas binarias como el *carry* de salida. A partir de este ejemplo, el lector puede modificar el programa para producir sumadores con tantos bits como desee, además de considerar la visualización del resultado en los *displays* ocupando los conocimientos previos.

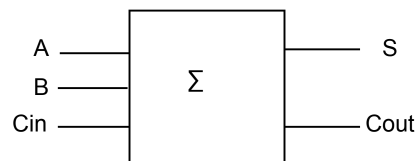


Figura 7.1.
Sumador de 1 bit.

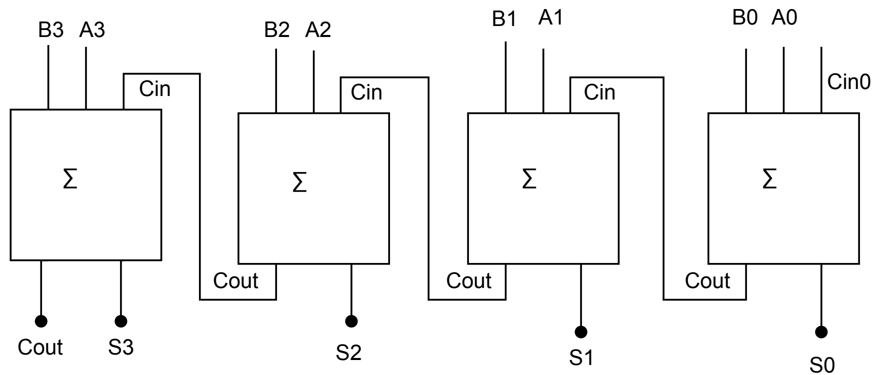


Figura 7.2.
Sumador de 4 bits.

1. Realice el siguiente programa en Verilog. Tenga presente que existen dos módulos: el primero es el módulo principal, que invoca al módulo de instanciación en cuatro oportunidades para lograr la suma completa, y el segundo un módulo de instanciación que suma 1 bit de las entradas A [x] y B[y].

```
module sumador_4_bits(                     // Módulo principal del sumador de 4 bits.
  input [3:0] A, B,                        // Entradas A y B de 4 bits.
  input Cin0,                              // Carry de entrada (es un switch puesto en 0).
  output [3:0] led_A,                      // Led indicador de las entradas A[x], de 4 bits.
  output [3:0] led_B,                      // Led indicador de las entradas B[y], de 4 bits.
  output [3:0] S,                          // Resultado de la suma de 4 bits.
  output Cout4 );                          // Carry de la suma final de 4 bits.

  wire Cout1, Cout2, Cout3;                // Conexiones de los carries de salida internos.

  assign led_A = A;                        // Asigna a led_A[x] los valores de A[x],
                                           // para visualizar las entradas A[x].
  assign led_B = B;                        // Asigna a led_B[y] los valores de B[y],
                                           // para visualizar las entradas B[y].

 //  Instanciaciones (llamadas) sucesivas al módulo que suma de a 1 bit
// (4 instanciaciones), para lograr la suma de 4 bits.
   sumador_1_bit f0 ( .A(A[0]),  .B(B[0]),  .Cin(Cin0),   .S(S[0]),   .Cout(Cout1) );    // LSB.
   sumador_1_bit f1 (  .A(A[1]),  .B(B[1]),  .Cin(Cout1),  .S(S[1]),   .Cout(Cout2) );
   sumador_1_bit f2 ( .A(A[2]),  .B(B[2]),  .Cin(Cout2),  .S(S[2]),   .Cout(Cout3) );
   sumador_1_bit f3 ( .A(A[3]),  .B(B[3]),  .Cin(Cout3),  .S(S[3]),   .Cout(Cout4) );    // MSB.
endmodule

module sumador_1_bit(                      // Módulo de instanciación que suma de 1 bit.
   input A, B,                             // Entradas de los sumandos A y B de 1 bit.
   input Cin,                              // Carry de entrada al sumador de 1 bit.
   output S,                               // Salida que contiene el resultado de la suma de 1 bit de A y B.
   output Cout );                          // Carry de salida del sumador de 1 bit.
   assign S = A ^ B ^ Cin;                 // Sumador implementado con or entre los sumandos y el carry.
   assign Cout = A&B | A&Cin | B&Cin ;     // Lógica de obtención del carry interno.
endmodule
```

2. Obtenga el esquemático del circuito resultante (figura 7.3).

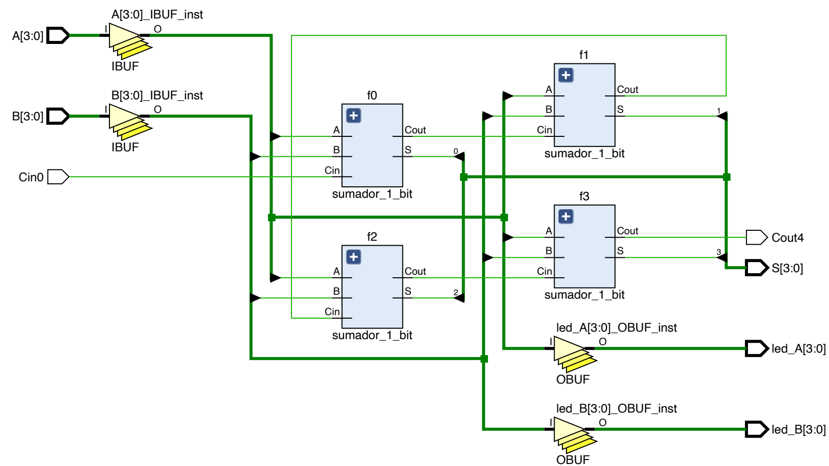


Figura 7.3. Esquemático del circuito sumador de 4 bits.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Switches (entradas)
set_property -dict {PACKAGE_PIN V17    IOSTANDARD LVCMOS33 } [get_ports {A[0]}]
set_property -dict {PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [get_ports {A[1]}]
set_property -dict {PACKAGE_PIN W16    IOSTANDARD LVCMOS33 } [get_ports {A[2]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {A[3]}]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {sw4}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {B[0]}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {B[1]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {B[2]}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {B[3]}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw9}]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {sw10}]
#set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {sw11}]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {sw12}]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {sw13}]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {sw14}]
set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {Cin0}]

## LEDs (salidas)
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led_A[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led_A[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led_A[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led_A[3]}]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led4}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led_B[0]}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led_B[1]}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led_B[2]}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {led_B[3]}]
#set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led9}]
#set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {led10}]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {S[0]}]
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {S[1]}]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {S[2]}]
set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {S[3]}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {Cout4}]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe el correcto funcionamiento del circuito.

7.2.2. PRÁCTICA 12: USO DE LOS DISPLAYS CON DATOS INDEPENDIENTES

En la práctica 6 del capítulo anterior se observó que la conversión BCD a 7 segmentos se muestra en un solo *display* (cualquiera de ellos), o con una modificación en la variable «display», el mismo dato en todos los *displays* simultáneamente. En esta práctica se independiza el uso de cada *display* recurriendo a la multiplexación de los datos que se traspasan a cada uno de ellos y a la persistencia de la visión humana. En efecto, el dato de cada *display* se va aplicando uno a la vez en el *display* que le corresponde, lo que a una frecuencia de actualización razonablemente alta produce el efecto visual de que en todo momento están encendidos cada uno con sus respectivos datos. Esto es necesario debido a que los recursos del archivo Basys-3-Master.xdc ponen a disposición de los usuarios los mismos 8 segmentos para los 4 *displays* de la tarjeta. En consecuencia, para usar los *displays* en forma independiente, se multiplexa el uso de dichos segmentos. A modo de ejemplo, en esta práctica se escribe la palabra «HOLA» con cada uno de los caracteres en un *display* distinto.

1. Realice el siguiente programa en Verilog.

```
module display_4_mux(                              // Activación independiente de los 4 displays.
    input clock,                                   // Reloj interno de 100 MHz.
    input reset,                                   // Reset del contador y displays.
    output reg [7:0] segmento,
    output reg [3:0] display );

    reg [19:0] contador;                           // Contador de retardo para activación del display.
    wire [1:0] contador_activa_display;            // Contador que activa el display.

    always@ (posedge clock, posedge reset)         // Bloque que incrementa la
                                                   // variable contador.
    begin
        if( reset == 1 )                           // Reinicia el contador en cero.
            contador <= 0;
        else
            contador <= contador + 1;
    end

    assign contador_activa_display = contador[19:18];   // El contador que activa los
                                                        // displays usa los 2 digitos MSB del registro.

    always@ (*) begin                              // Muestra datos independientes en cada display.
        if( reset == 1 ) begin                     // Si aplica «reset», los displays se apagan.
        display = 4'b1111;
        segmento = 8'b11000000;
        end
        else
            case( contador_activa_display )
                2'b00: begin
                display = 4'b0111;                 // Activa display 4.
```

```
            segmento = 8'b10001001;          // Enciende letra H.
            end
            2'b01: begin
            display = 4'b1011;               // Activa display 3.
            segmento = 8'b11000000;          // Enciende letra O.
            end
            2'b10: begin
            display = 4'b1101;               // Activa display 2.
            segmento = 8'b11000111;          // Enciende letra L.
            end
            2'b11: begin
            display = 4'b1110;               // Activa display 1.
            segmento = 8'b10001000;          // Enciende letra A.
            end
        endcase
    end
endmodule
```

2. Obtenga el esquemático del circuito (figura 7.4).

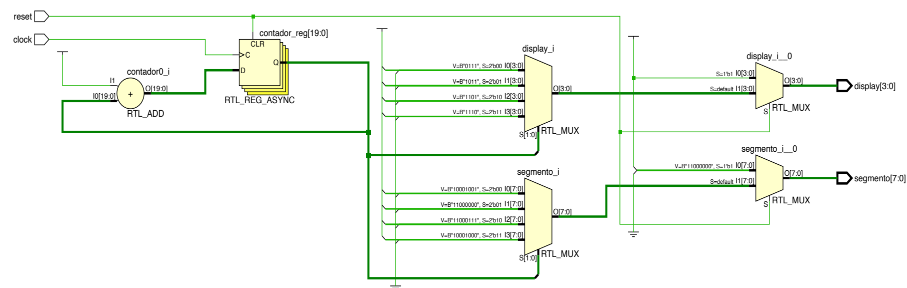


Figura 7.4.
Esquemático de la práctica 12.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clock}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {segmento[0]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {segmento[1]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {segmento[2]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {segmento[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {segmento[4]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {segmento[5]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {segmento[6]}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {segmento[7]}]
```

```
set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {display[0]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {display[1]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {display[2]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {display[3]}]


##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {btnC}]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {bntU}]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports {btnL}]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {reset}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {btnD}]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe el correcto funcionamiento del circuito.

7.2.3. PRÁCTICA 13: USO DEL RELOJ INTERNO DE LA TARJETA

En esta práctica se generan dos señales de reloj de distintas frecuencias a partir del reloj interno de 100 MHz de la tarjeta. Las pulsaciones resultantes se pueden visualizar mediante el parpadeo de dos leds: uno cada 1 segundo y el otro cada 4 segundos. A partir de estos ejemplos el lector puede modificar el programa para producir las señales de reloj que desee, tanto en cantidad como en frecuencia, y disponerlas para el funcionamiento de otros componentes como *flip flops*, registros de desplazamiento o contadores, entre otros, ya sea configurados en la propia tarjeta o interconectados como componentes periféricos externos, a los cuales se les proporcionan señales de reloj vía salidas Pmod.

Una particularidad de este programa es que la obtención de las señales de reloj de 1 y 4 segundos se efectúa ralentizando el reloj de 100 MHz de la tarjeta mediante un módulo de instanciación denominado «divisor_frecuencia», el cual es llamado desde el programa principal, retornando las dos señales indicadas anteriormente.

1. Realice el siguiente programa en Verilog.

```
module parpadeo_relojes(            // Circuito que enciende un led cada 1seg y otro cada 4seg.
  input clk,                        // clk es el reloj interno de 100MHz de la tarjeta.
  output led_1Hz,                   // Led que se enciende cada 1 segundo.
  output led_4Hz );                 // Led que se enciende cada 4 segundos.

  wire clk_1Hz, clk_4Hz;            // Variables internas que reciben el reloj de 1Hz y 4Hz
                                    // desde el divisor_fecuencia.

  divisor_frecuencia (              // Instanciación al módulo divisor_frecuencia.
    .clk_in(clk),                   // Traspasa el clk de 100MHz a la variable clk_in.
    .clk_out1(clk_1Hz),             // Recibe en clk_out1 el reloj de 1Hz desde el módulo
                                    // divisor_frecuencia y lo traspasa a la variable clk_1Hz.
    .clk_out2(clk_4Hz) );           // Recibe en clk_out2 el reloj de 4Hz desde el módulo
                                    // divisor_frecuencia y lo traspasa a la variable clk_4Hz.
```

```
    assign led_1Hz = clk_1Hz;                          // Encendido del led a una frecuencia de 1 Hz.
    assign led_4Hz = clk_4Hz;                          // Encendido del led a una frecuencia de 4 Hz.

endmodule

module divisor_frecuencia(                             // Divide la frecuencia del reloj de 100MHz a 1Hz y 4Hz.
    input wire clk_in,                                 // Variable que recibe reloj interno de 100MHz.
    output wire clk_out1,                              // Reloj de salida de 1Hz.
    output wire clk_out2 );                            // Reloj de salida de 4Hz.

    parameter Seg1 = 100_000_000;                      // Asigna el valor de 100M a una constante
                                                       // Seg1 para producir 1seg (100M/100MHz).
    localparam Nbits1 = $clog2(Seg1);                  // Guarda en Nbits1 el número de bits
                                                       // necesarios para almacenar Seg1.
    reg [Nbits1-1:0] contador_divisor1 = 0;            // Define el reg contador_divisor1 para
                                                       // usarlo como contador y lo inicia en 0.

    parameter Seg2 = 400_000_000;                      // Asigna el valor de 400M a una constante
                                                       // Seg2 para producir 4seg (400M/100MHz).
    localparam Nbits2 = $clog2(Seg2);                  // Guarda en Nbits2 el número de bits
                                                       // necesarios para almacenar Seg2.
    reg [Nbits2-1:0] contador_divisor2 = 0;            // Define el reg contador_divisor2 para
                                                       // usarlo como contador y lo inicia en 0.

    always@(posedge clk_in)                            // Realiza el bloque always con cada
                                                       // subida del reloj clk_in.
    begin
        if (contador_divisor1 == Seg1-1)               // Si el contador_divisor1 llega al máximo,
            contador_divisor1 <= 0;                     // lo deja en 0 para comenzar un nuevo conteo.
        else
        contador_divisor1 <= contador_divisor1 + 1;     // Si no ha llegado al valor máximo lo
                                                       // incrementa en 1.

        if (contador_divisor2 == Seg2-1)               // Si el contador_divisor2 llega al máximo,
            contador_divisor2 <= 0;                    // lo deja en 0 para comenzar un nuevo conteo.
        else
        contador_divisor2 <= contador_divisor2 + 1;     // Si no ha llegado al valor máximo lo incrementa en 1.
    end

    assign clk_out1 = contador_divisor1[Nbits1-1];     // Asigna al clk_out1 y clk_out2 el bit
    assign clk_out2 = contador_divisor2[Nbits2-1];     // más significativo de los registros
                                                       // contadores divisores.

endmodule
```

2. Obtenga el esquemático del circuito (figura 7.5). Puede ampliar para obtener más detalles.

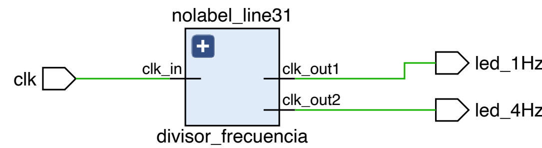


Figura 7.5. Esquemático de la práctica 13.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación (para proveer señales de reloj a dispositivos periféricos puede reemplazar las salidas leds por salidas Pmod):

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]

## LEDs (salidas)
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led_1Hz}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led_4Hz}]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe el correcto funcionamiento del circuito.

7.2.4. PRÁCTICA 14: CONTROL DE SEMÁFOROS CON LATCH

Esta práctica implementa un sistema de control de semáforos de una intersección de calles. Consta de la secuenciación de los semáforos verde, amarillo y rojo en un sentido (semáforo 1), y lo propio en el sentido transversal (semáforo 2). La temporalidad de la secuencia de encendido de los semáforos está elaborada de modo que no se produzcan colisiones. Es decir, la luz verde y amarilla de un semáforo debe coincidir con la luz roja del otro.

Los tiempos asignados a cada semáforo se obtienen de un contador conformado por cinco *flip flops* tipo T que proporcionan una cuenta de 32 segundos para esta práctica. El conteo se visualiza en un *display* en código hexadecimal. Este es el ciclo completo de 32 segundos del verde, amarillo y rojo obtenido a partir de la ralentización del reloj de la tarjeta, por lo que se puede modificar como se desee. La figura 7.6 indica la distribución de tiempos.

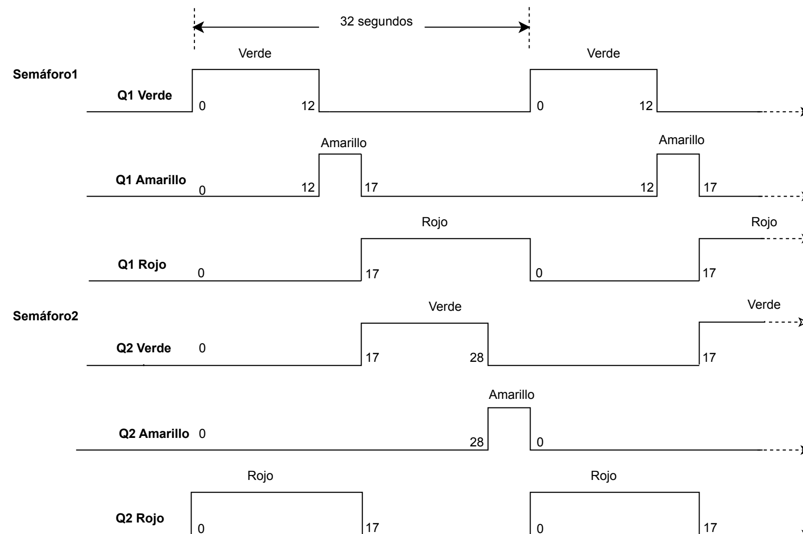


Figura 7.6. Diagramas de tiempo del semáforo.

Cada uno de los tiempos necesarios para la operación de los semáforos que se indican en la figura 7.6 (0, 12, 17 y 28 segundos) son detectados con compuertas *nand* desde el contador y aplicados convenientemente a un arreglo de 6 *latchs*, 1 para cada color de ambos semáforos (flip flops tipo S-R). Estos activan un «1» en la salida «Q» deseada cuando la entrada «S» pasa a «0» y se conserva en este estado hasta que la entrada «R» pasa a «0». Precisamente, la necesidad de generar y conservar 6 temporizaciones distintas para las 6 luces de los 2 semáforos hace conveniente esta solución con *latch* (el lector puede optimizar la solución reusando algunas de estas temporizaciones). En síntesis, tal como se muestra de manera resumida en la figura 7.7, el circuito se configura con 5 FF tipo T para establecer una cuenta de 0 a 31 segundos y 6 *latches*, uno para cada color de los 2 semáforos, en donde la salida «Q» de cada uno de ellos es la que activa el color del semáforo correspondiente, al ritmo del contador y de la detección de los tiempos asignados para cada color.

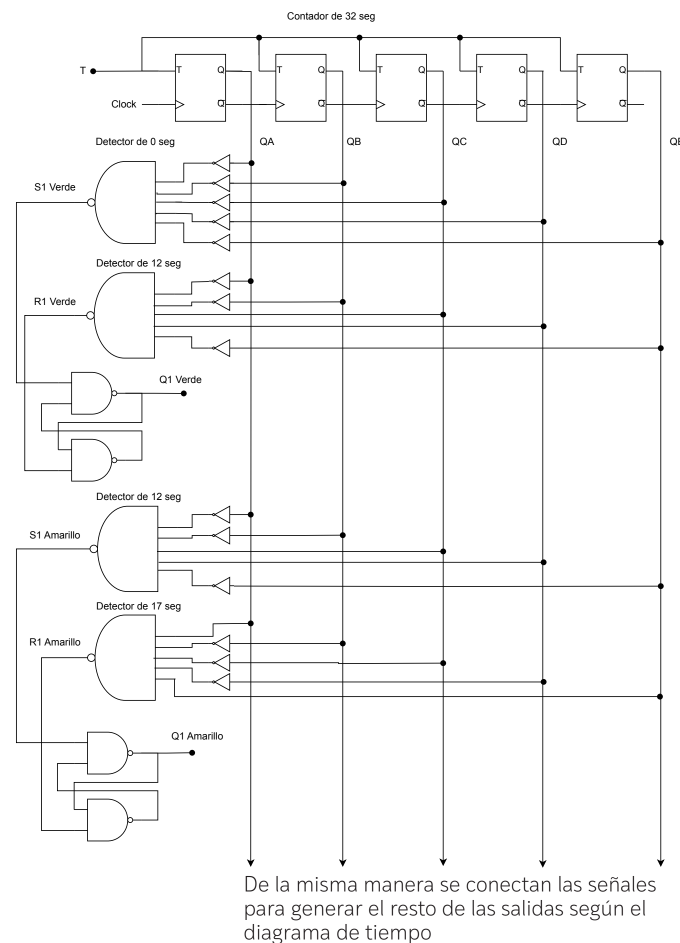


Figura 7.7. Diagrama general del sistema de control de 2 semáforos.

1. Realice el siguiente programa en Verilog.

```
module semaforos(                                   // Semáforos configurados con latches.
   input T,                                         // Entrada común a 5 flip flops.
   input clk,                                       // clk es el reloj interno de 100MHz de la tarjeta.
   input clear,                                     // Variable para aplicar reset.
   output reg [7:0] segmento,                       // Variable para mostrar el conteo del reloj en 1 display.
   output [3:0] display,                            // Variable para seleccionar 1 display.
   output led_1Hz,                                  // Led indicador del parpadeo de un reloj de 1Hz.
   output reg Q1_Verde, Q1_Amarilla, Q1_Roja,       // Salidas Q de los latchs del semáforo 1.
   output reg Q2_Verde, Q2_Amarilla, Q2_Roja );     // Salidas Q de los latchs del semáforo 2.
```

```
wire new_clk_1Hz;                          // Variable que recibe el reloj de 1Hz desde el divisor.

reg QA, QB, QC, QD, QE;                    // Salidas de los FF T que implementan el contador.

// Entradas S y R a los latches del semáforo 1.
reg S1_Verde, R1_Verde, S1_Amarilla, R1_Amarilla, S1_Roja, R1_Roja;

// Salidas Qnegadas de los latches del semáforo 1.
reg Qneg1_Verde, Qneg1_Amarilla, Qneg1_Roja;

// Entradas S y R a los latches del semáforo 2.
reg S2_Verde, R2_Verde, S2_Amarilla, R2_Amarilla, S2_Roja, R2_Roja;

// Salidas Qnegadas de los latches del semáforo 2.
reg Qneg2_Verde, Qneg2_Amarilla, Qneg2_Roja;

divisor_frecuencia(                        // Instanciación (llamado) al módulo divisor_frecuencia.
.reloj_in(clk),                            // Traspasa el reloj interno de 100 MHz a reloj_in.
.reloj_out(new_clk_1Hz) );                 // El reloj_out retorna con el valor de 1Hz y lo traspasa a
                                           // new_clk_1Hz.

assign led_1Hz = new_clk_1Hz;              // Asigna el reloj de 1Hz a la variable led_1Hz.

// LATCHES GENERADORES DE LA SEÑALES VERDE, AMARILLA Y ROJA DE LOS 2
// SEMÁFOROS.
always@ (*)
begin
if (clear) begin                     // Opción para resetear la operación dejando las salidas Q en 0.
   S1_Verde = 1;                     // (inicializa las salidas Q de los latches en 0).
   S1_Amarilla = 1;
   S1_Roja = 1;
   R1_Verde = 0;
   R1_Amarilla = 0;
   R1_Roja = 0;

   S2_Verde = 1;
   S2_Amarilla = 1;
   S2_Roja = 1;
   R2_Verde = 0;
   R2_Amarilla = 0;
   R2_Roja = 0; end
```

```
// SEMÁFORO 1: LUZ VERDE.
 S1_Verde = ~( ~QE & ~QD & ~QC & ~QB & ~QA );          // Detecta el 0seg del contador.
 R1_Verde = ~( ~QE & QD & QC & ~QB & ~QA );            // Detecta los 12seg del contador.

 if ( S1_Verde == 0 & R1_Verde == 1 ) begin            // Procesa la entrada 01 al latch Verde.
   Q1_Verde <= ~( 0 & Qneg1_Verde );
   Qneg1_Verde <= ~( 1 & Q1_Verde ); end

 if ( S1_Verde == 1 & R1_Verde == 0 ) begin            // Procesa la entrada 10 al latch Verde.
   Q1_Verde <= ~( 1 & Qneg1_Verde );
   Qneg1_Verde <= ~( 0 & Q1_Verde ); end

 if ( S1_Verde == 1 & R1_Verde == 1 ) begin            // Procesa la entrada 11 al latch Verde.
   Q1_Verde <= ~( 1 & Qneg1_Verde );
   Qneg1_Verde <= ~( 1 & Q1_Verde ); end

// SEMÁFORO 1: LUZ AMARILLA.
 S1_Amarilla = ~( ~QE & QD & QC & ~QB & ~QA );         // Detecta los 12seg del contador.
 R1_Amarilla = ~( QE & ~QD & ~QC & ~QB & QA );         // Detecta los 17seg del contador.

 if ( S1_Amarilla == 0 & R1_Amarilla == 1 ) begin      // Procesa la entrada 01 al latch Amar.
   Q1_Amarilla <= ~( 0 & Qneg1_Amarilla );
   Qneg1_Amarilla <= ~( 1 & Q1_Amarilla ); end

 if ( S1_Amarilla == 1 & R1_Amarilla == 0 ) begin      // Procesa la entrada 10 al latch Amar.
   Q1_Amarilla <= ~( 1 & Qneg1_Amarilla );
   Qneg1_Amarilla <= ~( 0 & Q1_Amarilla ); end

 if ( S1_Amarilla == 1 & R1_Amarilla == 1 ) begin      // Procesa la entrada 11 al latch Amar.
   Q1_Amarilla <= ~( 1 & Qneg1_Amarilla );
   Qneg1_Amarilla <= ~( 1 & Q1_Amarilla ); end

// SEMÁFORO 1: LUZ ROJA.
 S1_Roja = ~( QE & ~QD & ~QC & ~QB & QA );             // Detecta los 17seg del contador.
 R1_Roja = ~( ~QE & ~QD & ~QC & ~QB & ~QA );           // Detecta los 0seg del contador.

 if ( S1_Roja == 0 & R1_Roja == 1 ) begin              // Procesa la entrada 01 al latch Rojo.
   Q1_Roja <= ~( 0 & Qneg1_Roja );
   Qneg1_Roja <= ~( 1 & Q1_Roja ); end

 if ( S1_Roja == 1 & R1_Roja == 0 ) begin              // Procesa la entrada 10 al latch Rojo.
   Q1_Roja <= ~( 1 & Qneg1_Roja );
```

```
    Qneg1_Roja <= ~( 0 & Q1_Roja ); end

  if ( S1_Roja == 1 & R1_Roja == 1 ) begin          // Procesa la entrada 11 al latch Rojo.
    Q1_Roja <= ~( 1 & Qneg1_Roja );
    Qneg1_Roja <= ~( 1 & Q1_Roja ); end

// SEMÁFORO 2: LUZ VERDE.
  S2_Verde = ~( QE & ~QD & ~QC & ~QB & QA );        // Detecta los 17seg del contador.
  R2_Verde = ~( QE & QD & QC & ~QB & ~QA );         // Detecta los 28seg del contador.

  if ( S2_Verde == 0 & R2_Verde == 1 ) begin        // Procesa la entrada 01 al latch Verde.
    Q2_Verde <= ~( 0 & Qneg2_Verde );
    Qneg2_Verde <= ~( 1 & Q2_Verde ); end

  if ( S2_Verde == 1 & R2_Verde == 0 ) begin        // Procesa la entrada 10 al latch Verde.
    Q2_Verde <= ~( 1 & Qneg2_Verde );
    Qneg2_Verde <= ~( 0 & Q2_Verde ); end

  if ( S2_Verde == 1 & R2_Verde == 1 ) begin        // Procesa la entrada 11 al latch Verde.
    Q2_Verde <= ~( 1 & Qneg2_Verde );
    Qneg2_Verde <= ~( 1 & Q2_Verde ); end

// SEMÁFORO 2: LUZ AMARILLA.
  S2_Amarilla = ~( QE & QD & QC & ~QB & ~QA );      // Detecta los 28seg del contador.
  R2_Amarilla = ~( ~QE & ~QD & ~QC & ~QB & ~QA );   // Detecta el 0seg del contador.

  if ( S2_Amarilla == 0 & R2_Amarilla == 1 ) begin  // Procesa la entrada 01 al latch Amar.
    Q2_Amarilla <= ~( 0 & Qneg2_Amarilla );
    Qneg2_Amarilla <= ~( 1 & Q2_Amarilla ); end

  if ( S2_Amarilla == 1 & R2_Amarilla == 0 ) begin  // Procesa la entrada 10 al latch Amar.
    Q2_Amarilla <= ~( 1 & Qneg2_Amarilla );
    Qneg2_Amarilla <= ~( 0 & Q2_Amarilla ); end

  if ( S2_Amarilla == 1 & R2_Amarilla == 1 ) begin  // Procesa la entrada 11 al latch Amar.
    Q2_Amarilla <= ~( 1 & Qneg2_Amarilla );
    Qneg2_Amarilla <= ~( 1 & Q2_Amarilla ); end

// SEMÁFORO 2: LUZ ROJA.
  S2_Roja = ~( ~QE & ~QD & ~QC & ~QB & ~QA );       // Detecta el 0seg del contador.
  R2_Roja = ~( QE & ~QD & ~QC & ~QB & QA );         // Detecta los 17seg del contador.

  if ( S2_Roja == 0 & R2_Roja == 1 ) begin          // Procesa la entrada 01 al latch Rojo.
```

```
      Q2_Roja <= ~( 0 & Qneg2_Roja );
      Qneg2_Roja <= ~( 1 & Q2_Roja ); end

    if ( S2_Roja == 1 & R2_Roja == 0 ) begin          // Procesa la entrada 10 al latch Rojo.
      Q2_Roja <= ~( 1 & Qneg2_Roja );
      Qneg2_Roja <= ~( 0 & Q2_Roja ); end

    if ( S2_Roja == 1 & R2_Roja == 1 ) begin          // Procesa la entrada 11 al latch Rojo.
      Q2_Roja <= ~( 1 & Qneg2_Roja );
      Qneg2_Roja <= ~( 1 & Q2_Roja ); end

    end

  // CONTADOR EN BASE A 5 FLIP FLOPS.
    always@ (posedge led_1Hz, posedge clear)          // Configura el primer FF.
    begin
      if (clear) QA = 1'b0;                           // Efectúa reset del primer FF.
      else
        case (T)
          1'b0: QA <= QA;
          1'b1: QA <= ~QA;
        endcase
    end

    always@ (posedge ~QA, posedge clear)              // Configura el segundo FF.
    begin
      if (clear) QB = 1'b0;                           // Efectúa reset del segundo FF.
      else
        case (T)
          1'b0: QB <= QB;
          1'b1: QB <= ~QB;
        endcase
    end

    always@ (posedge ~QB, posedge clear)              // Configura el tercer FF.
    begin
      if (clear) QC = 1'b0;                           // Efectúa reset del tercer FF.
      else
        case (T)
          1'b0: QC <= QC;
          1'b1: QC <= ~QC;
        endcase
    end
```

```
  always@ (posedge ~QC, posedge clear)            // Configura el cuarto FF.
  begin
    if (clear) QD = 1'b0;                         // Efectúa reset del cuarto FF.
    else
      case (T)
        1'b0: QD <= QD;
        1'b1: QD <= ~QD;
      endcase
  end

  always@ (posedge ~QD, posedge clear)            // Configura el quinto FF.
  begin
    if (clear) QE = 1'b0;                          // Efectúa reset del quinto FF.
    else
      case (T)
        1'b0: QE <= QE;
        1'b1: QE <= ~QE;
      endcase
  end

 // MUESTRA EN UN DISPLAY LA CUENTA DEL CONTADOR (EN HEXADECIMAL).
  always@( { QD, QC, QB, QA} )                    // (ver uso de los displays en el cap. 6, práctica 6).
    case({QD, QC, QB, QA})
      4'b0001: segmento = 8'b11111001;
      4'b0010: segmento = 8'b10100100;
      4'b0011: segmento = 8'b10110000;
      4'b0100: segmento = 8'b10011001;
      4'b0101: segmento = 8'b10010010;
      4'b0110: segmento = 8'b10000010;
      4'b0111: segmento = 8'b11111000;
      4'b1000: segmento = 8'b10000000;
      4'b1001: segmento = 8'b10010000;
      4'b1010: segmento = 8'b10001000;
      4'b1011: segmento = 8'b10000011;
      4'b1100: segmento = 8'b11000110;
      4'b1101: segmento = 8'b10100001;
      4'b1110: segmento = 8'b10000110;
      4'b1111: segmento = 8'b10001110;
      default:   segmento = 8'b11000000;
    endcase
  assign display =4'b1110;                         // Selecciona el display de la derecha.

endmodule
```

```
module divisor_frecuencia(                          // Divide la frecuencia del reloj de 100MHz a 1Hz.
    input wire reloj_in,                            // Variable que recibe el reloj interno de 100MHz.
    output wire reloj_out );                        // Reloj de salida del divisor_frecuencia.

    parameter Seg1 = 100_000_000;                   // Asigna el valor de 100M a una constante Seg1
                                                    // para producir 1seg (100M/100MHz).
    localparam Nbits = $clog2(Seg1);                // Guarda en Nbits el número de bits necesarios
                                                    // para almacenar Seg1.
    reg [Nbits-1:0] contador_divisor1 = 0;          // Define el registro contador_divisor para
                                                    // utilizarlo como contador y lo inicializa en 0.

    always@(posedge reloj_in)                       // Cuenta de 0 a Seg1-1 en cada subida de reloj_in.
      if (contador_divisor1 == Seg1-1)              // Si el contador_divisor llega al valor máximo,
        contador_divisor1 <= 0;                     // lo deja en 0 (para iniciar nuevo conteo).
      else
        contador_divisor1 <= contador_divisor1 +1;  // Si el contador_divisor no ha llegado
                                                    // al valor máximo, lo incrementa en 1.
    assign reloj_out = contador_divisor1[Nbits-1];  // Asigna al reloj_out el valor más
                                                    // significativo del registro
endmodule                                           // contador_divisor (frec. de 1Hz).
```

2. Obtenga el esquemático del circuito (figura 7.8). Puede hacer zoom para ver los detalles.

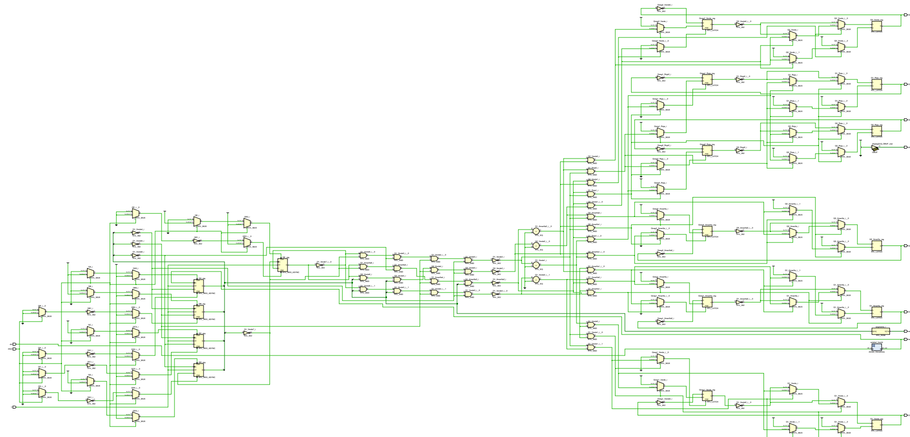


Figura 7.8. Esquemático del circuito semáforo.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
```

```
## Switches (entradas)
set_property -dict {PACKAGE_PIN V17    IOSTANDARD LVCMOS33 } [get_ports {T}]
#set_property -dict {PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [get_ports {sw1}]
#set_property -dict {PACKAGE_PIN W16    IOSTANDARD LVCMOS33 } [get_ports {sw2}]

## LEDs (salidas)
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led_1Hz}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led1}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led2}]

##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {segmento[0]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {segmento[1]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {segmento[2]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {segmento[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {segmento[4]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {segmento[5]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {segmento[6]}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {segmento[7]}]

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {display[0]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {display[1]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {display[2]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {display[3]}]

##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {btnC}]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {bntU}]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports {btnL}]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {clear}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {btnD}]

##Pmod Header JA
set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {Q1_Verde}]
set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports {Q1_Amarilla}]
set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {Q1_Roja}]
#set_property -dict { PACKAGE_PIN G2   IOSTANDARD LVCMOS33 } [get_ports {JA4}]
set_property -dict { PACKAGE_PIN H1   IOSTANDARD LVCMOS33 } [get_ports {Q2_Verde}]
set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports {Q2_Amarilla}]
set_property -dict { PACKAGE_PIN H2   IOSTANDARD LVCMOS33 } [get_ports {Q2_Roja}]
#set_property -dict { PACKAGE_PIN G3   IOSTANDARD LVCMOS33 } [get_ports {JA10}]
```

4. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe la correcta secuenciación de encendido de los semáforos.

7.2.5. PRÁCTICA 15: CONVERSIÓN ANÁLOGO A DIGITAL

El circuito que se implementa en esta práctica convierte la lectura de un voltaje analógico a su equivalente digital con 16 bits (65.536 niveles, el máximo que provee la tarjeta BASYS-3). La entrada al conversor se obtiene a partir de un arreglo divisor de voltaje de 3,3 volts como el que muestra la figura 7.9, que genera un rango de entrada entre 0 y 1 volt al conversor. La conversión digital se muestra escalada al valor decimal correspondiente en los 4 *displays* de la tarjeta, en un formato unidad-punto-décima-centésima-milésima (3 decimales). De este modo, como un mecanismo de comprobación, si se mide el voltaje con un *tester* digital en el punto J3 (pin1) del divisor que muestra la figura 7.9 (punto medio del potenciómetro), esta medición debe ser la misma que el valor mostrado en los 4

displays. Tener presente que la tarjeta posee 4 canales para conversión análogo a digital provistos por las entradas PMod XADC. En esta práctica se selecciona el canal VAUXP6 y VAUXN6 (corresponde al canal 1).

Figura 7.9. Conexión de la entrada analógica al conversor.

La utilización del conversor de la tarjeta implica necesariamente la configuración de 2 módulos específicos de Vivado. Uno de ellos es el reloj «clocking», renombrado como «clock_104» en este caso, y el otro es el conversor «xadc», renombrado como «xadc1» (para mayor información sobre estos recursos de Vivado, ver el capítulo 5, sección 5.2). Adicionalmente, se requiere programar el «source» correspondiente, que, entre otros aspectos, accede a estos módulos con «instanciaciones» (ver el capítulo 4, sección 4.9). Estos módulos contienen variables para una amplia gama de aplicaciones. En esta práctica, la parametrización está acotada a producir una conversión básica. Se detallan a continuación los pasos sucesivos a seguir para parametrizar los módulos, así como la programación propiamente tal del «source» que cumple con la especificación detallada.

1. Realice la parametrización del reloj (*clocking*). Tenga presente que antes de invocar estas funciones de Vivado, es necesario crear un proyecto de manera normal y avanzar hasta la primera ventana que se indica en esta sucesión de tareas (figura 7.10). Luego, el procedimiento consiste en continuar ejecutando sucesivamente las acciones asociadas a las láminas indicadas desde la figura 7.10 (activar IP catálogo; ver sección 5.2.1) hasta la figura 7.23.

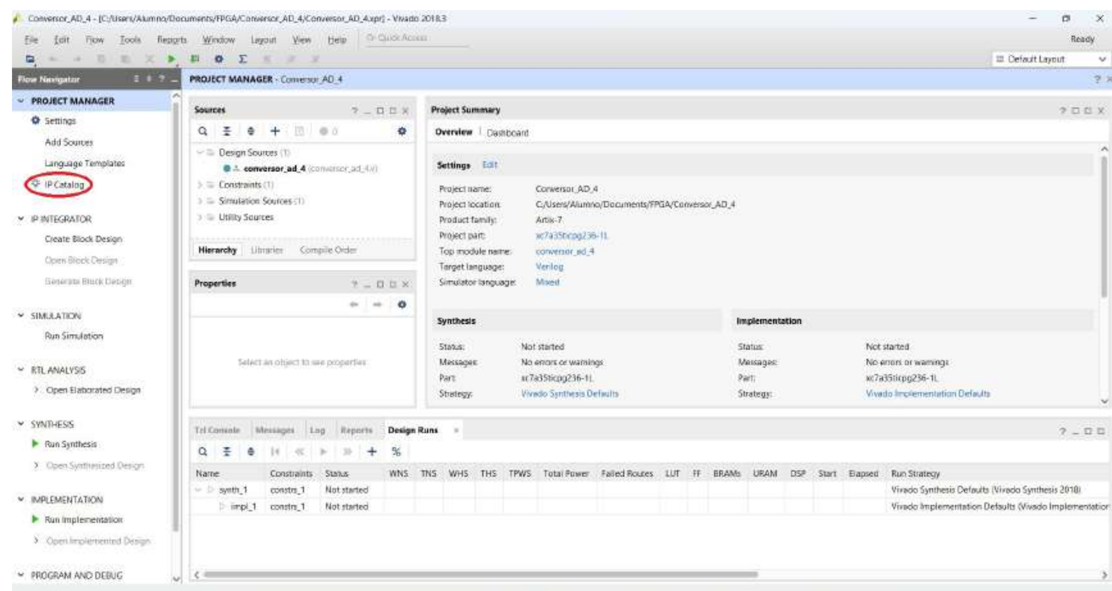


Figura 7.10. Lámina 1 del procedimiento de configuración de los bloques de conversión AD.

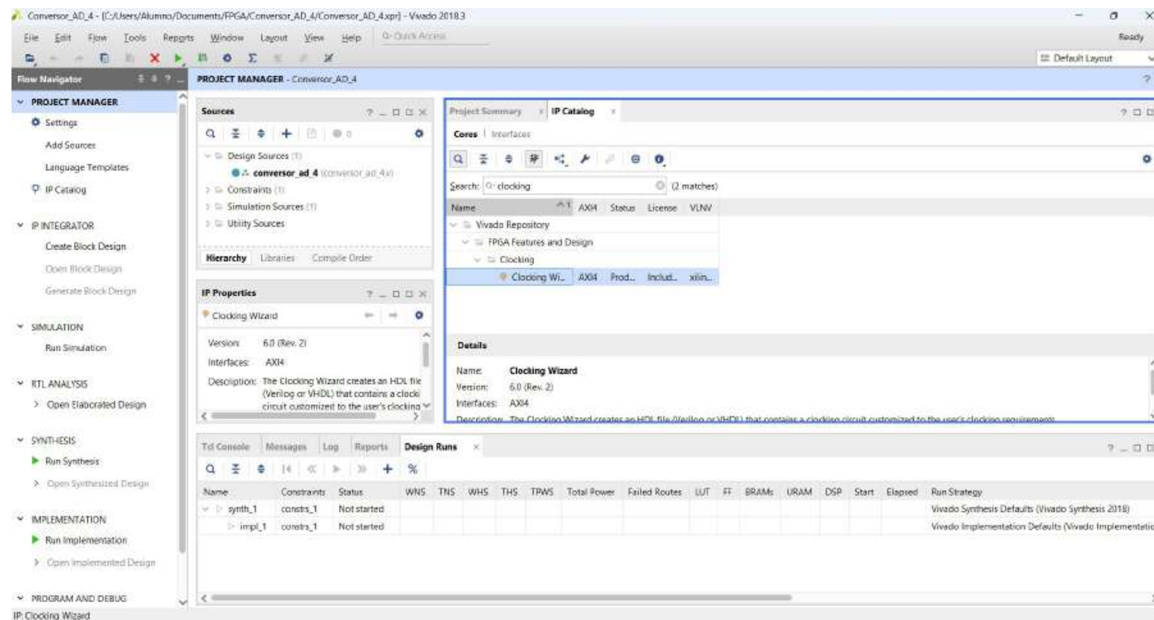


Figura 7.11. Lámina 2 del procedimiento de configuración de los bloques de conversión AD.

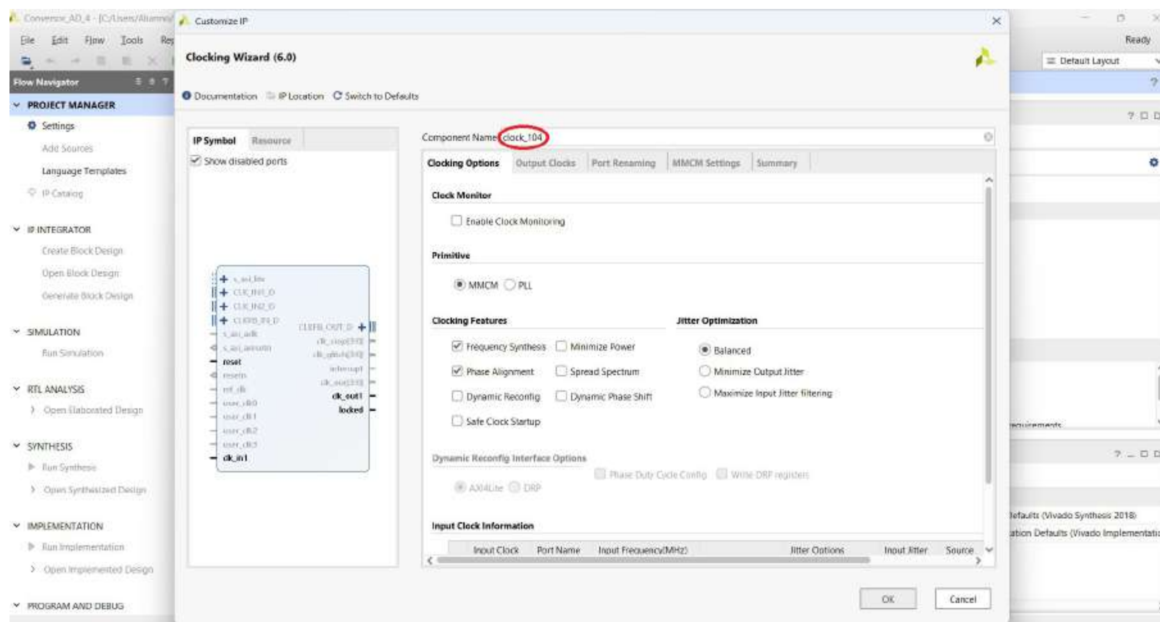


Figura 7.12. Lámina 3 del procedimiento de configuración de los bloques de conversión AD.

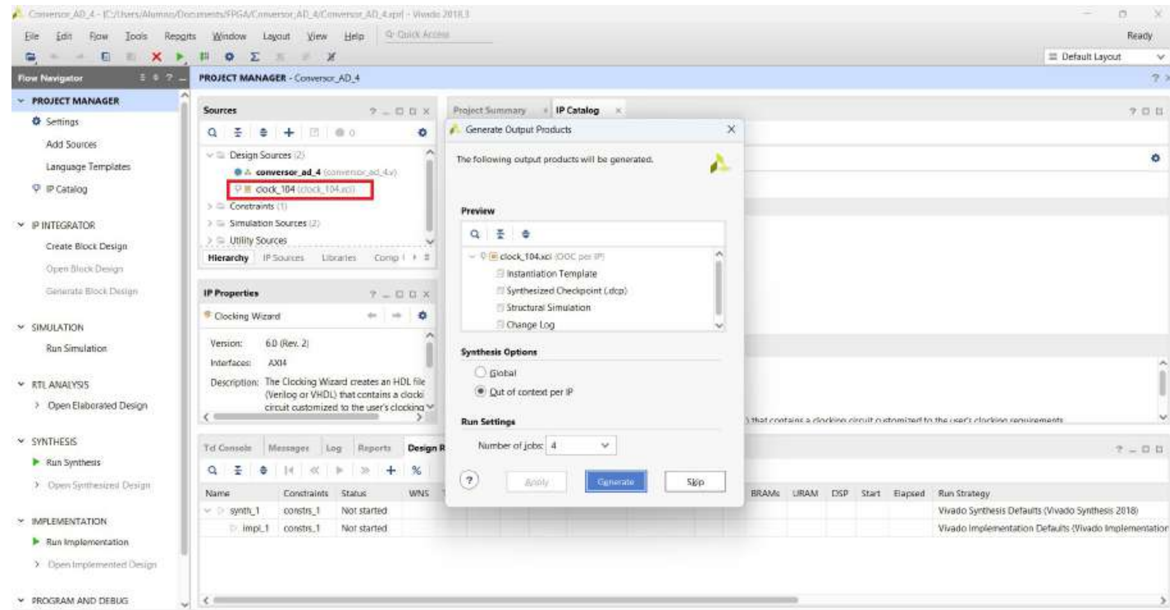


Figura 7.13. Lámina 4 del procedimiento de configuración de los bloques de conversión AD.

Nótese que ahora ya aparece indicada la instancia «clock_104» vinculada al «source» del programa, en este caso, llamado «conversor_ad_4».

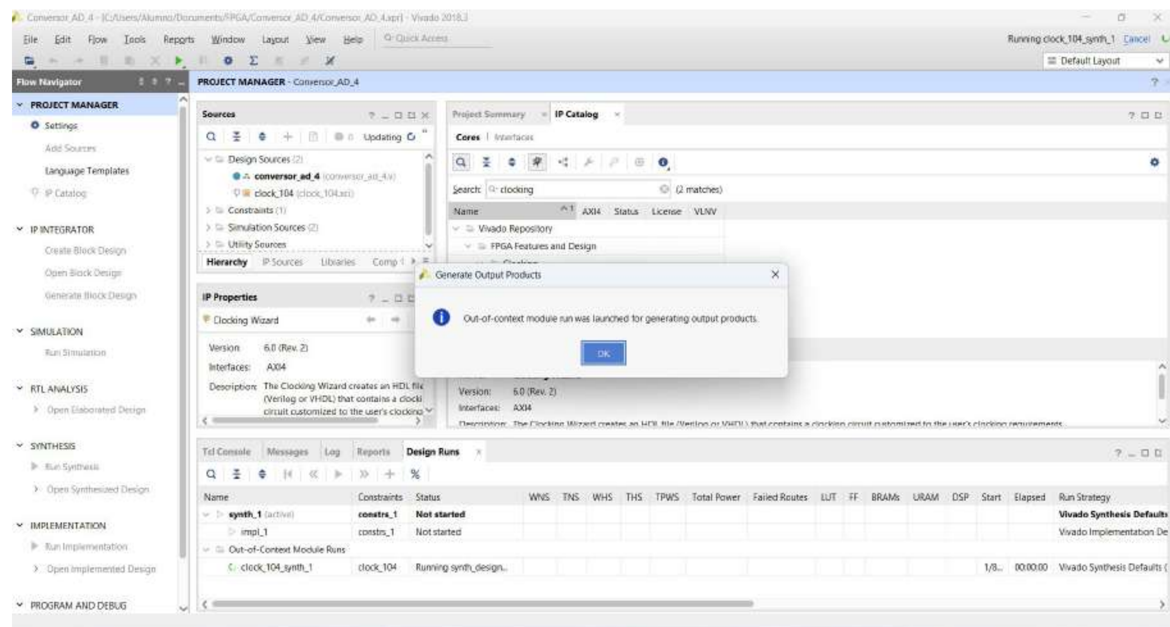


Figura 7.14. Lámina 5 del procedimiento de configuración de los bloques de conversión AD.

2. Realice la siguiente parametrización del «conversor» (xadc). Consiste en continuar la secuencia de configuración de manera similar a la indicada en el punto 1, pero esta vez referenciando al «conversor».

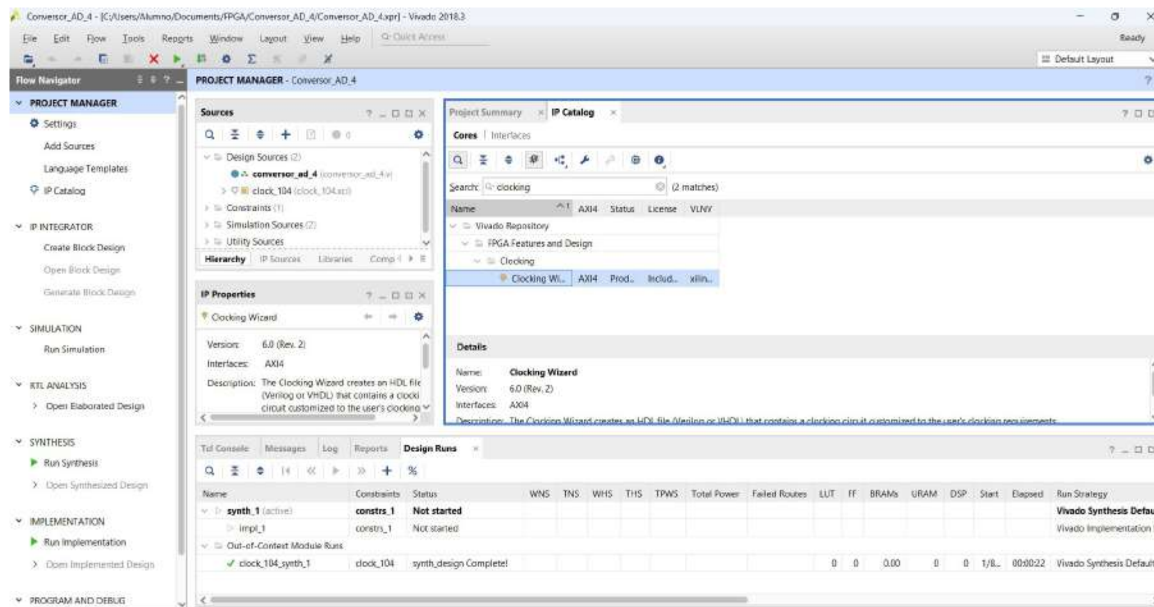


Figura 7.15. Lámina 6 del procedimiento de configuración de los bloques de conversión AD.

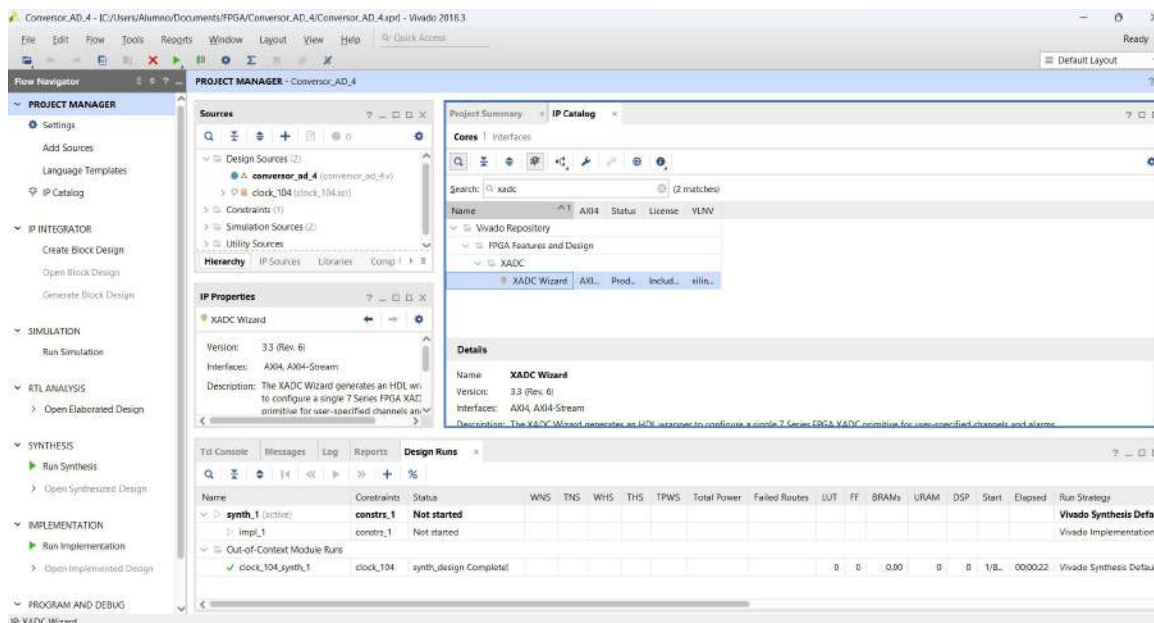


Figura 7.16. Lámina 7 del procedimiento de configuración de los bloques de conversión AD.

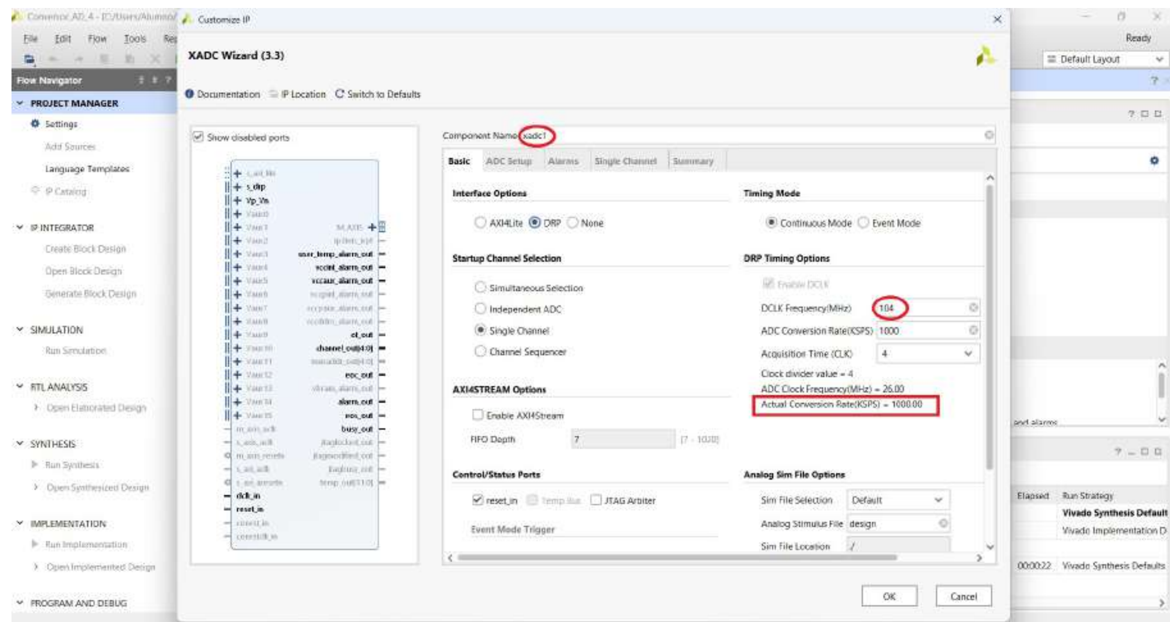


Figura 7.17. Lámina 8 del procedimiento de configuración de los bloques de conversión AD.

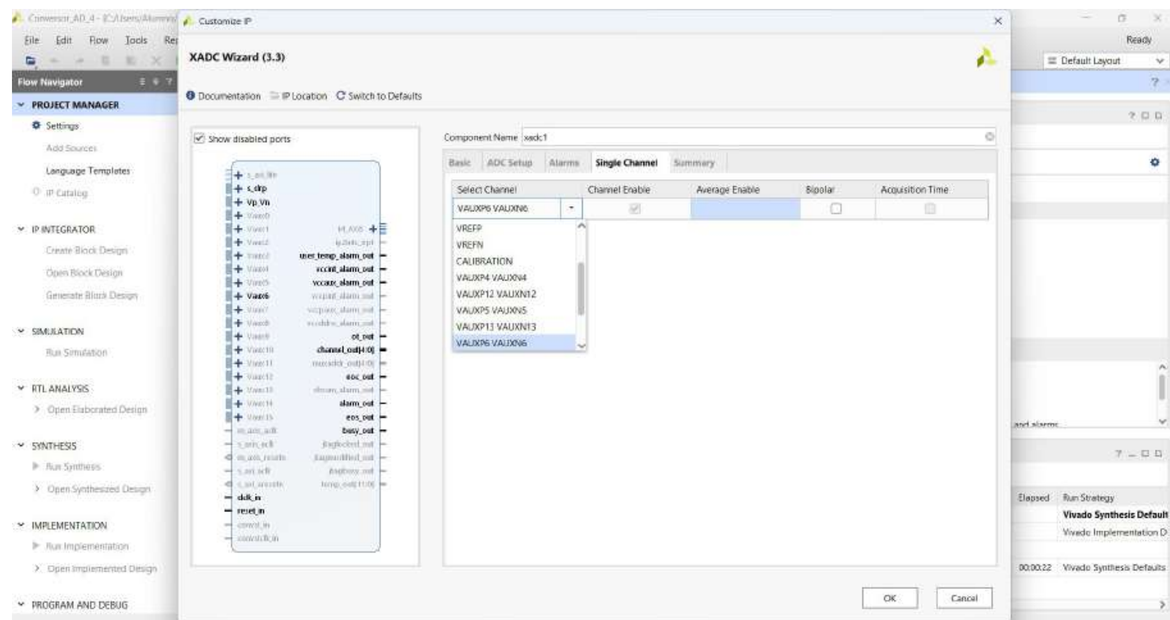


Figura 7.18. Lámina 9 del procedimiento de configuración de los bloques de conversión AD.

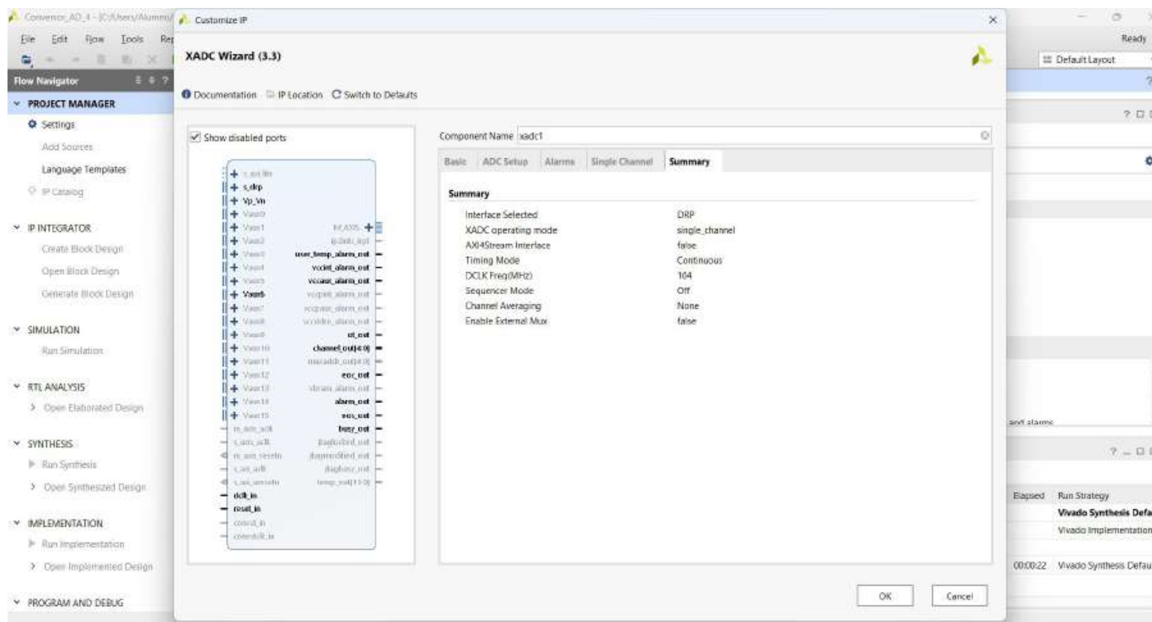


Figura 7.19. Lámina 10 del procedimiento de configuración de los bloques de conversión AD.

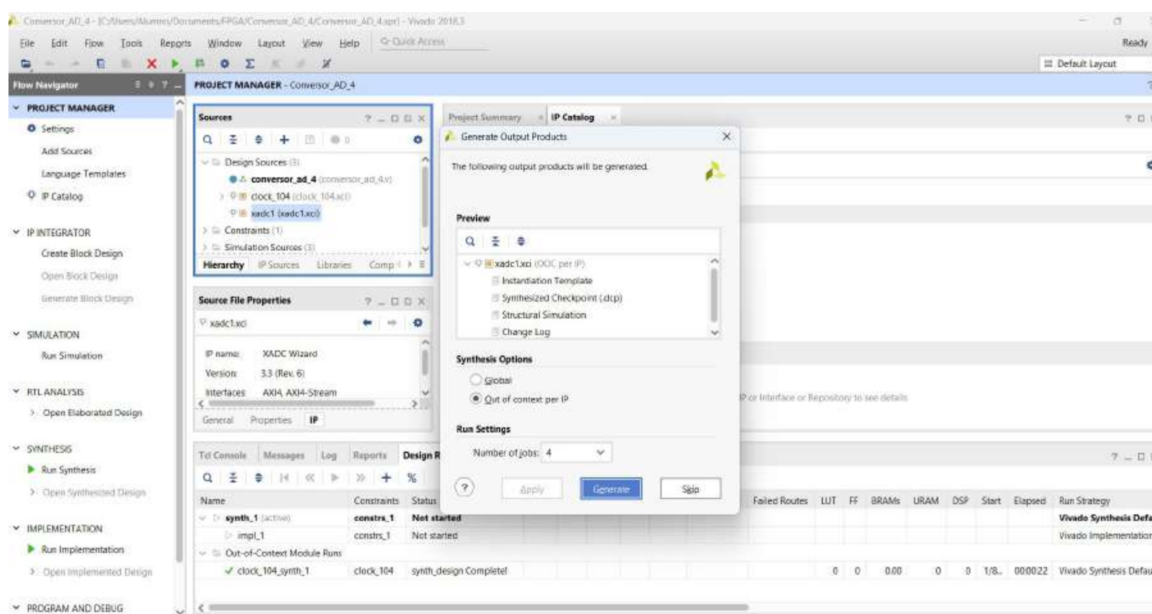


Figura 7.20. Lámina 11 del procedimiento de configuración de los bloques de conversión AD.

Nótese que ahora ya aparece indicada la instancia «xadc1» vinculada al «source» del programa llamado, en este caso, «conversor_ad_4».

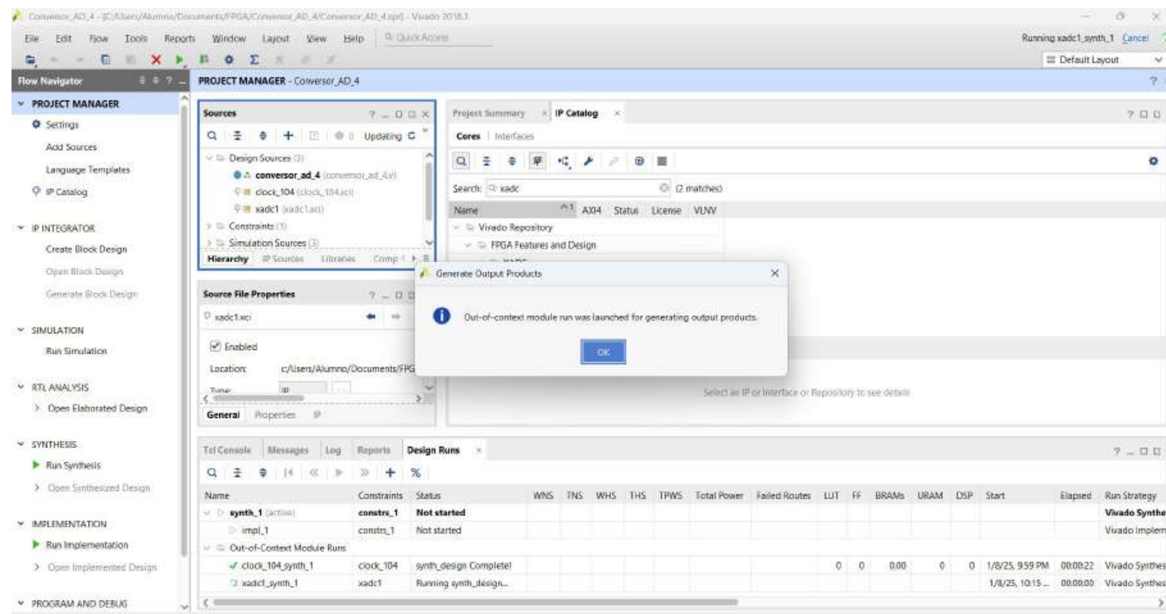


Figura 7.21.
Lámina 12 del procedimiento de configuración de los bloques de conversión AD.

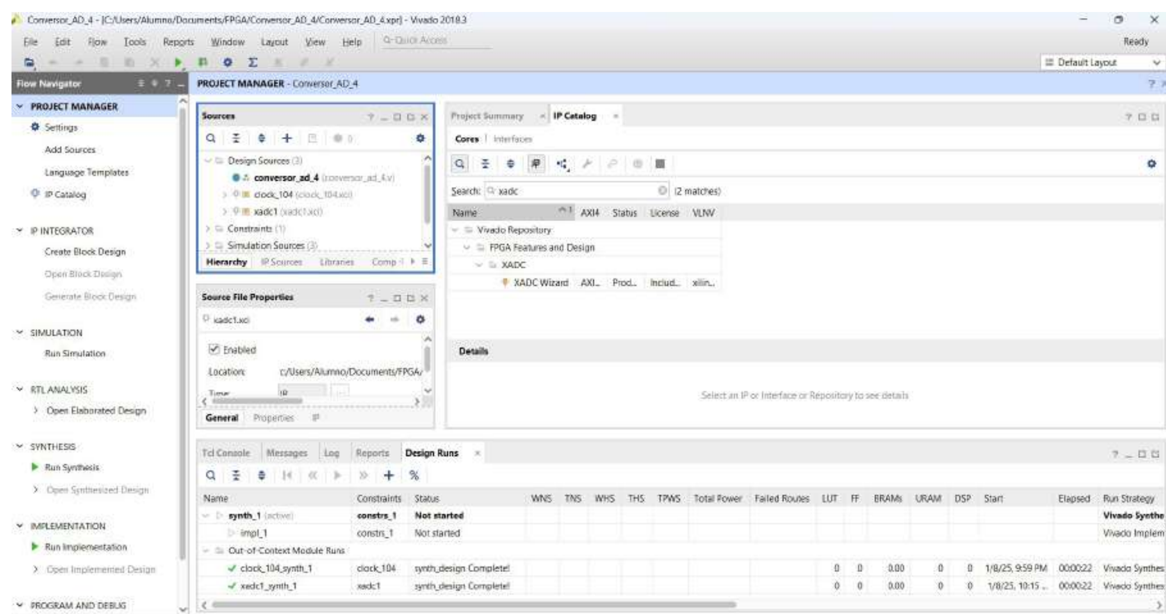


Figura 7.22.
Lámina 13 del procedimiento de configuración de los bloques de conversión AD.

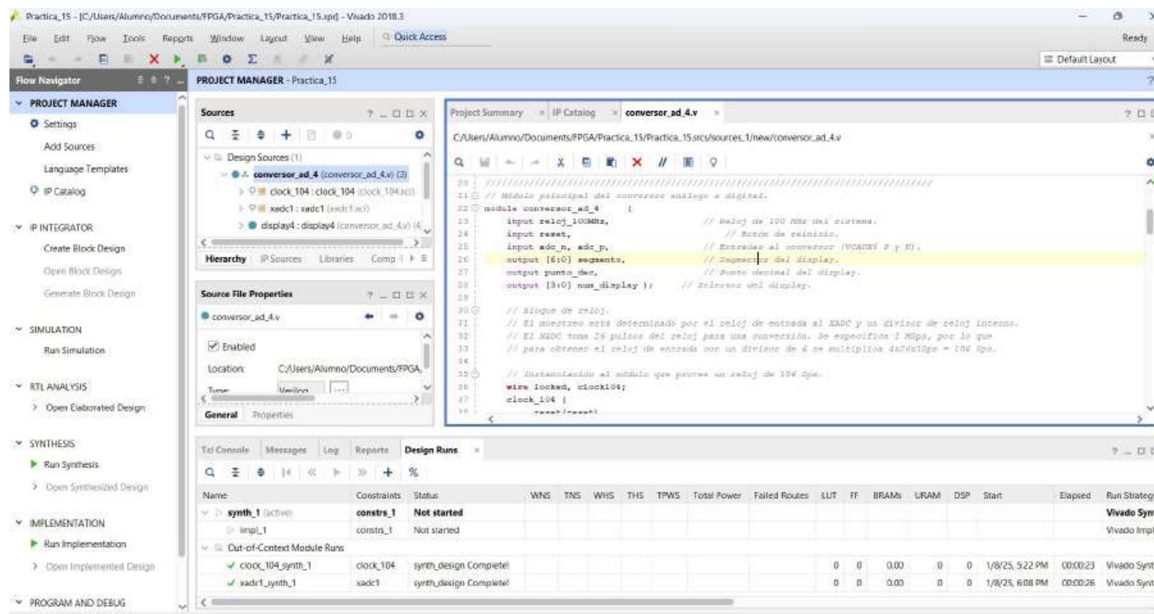


Figura 7.23. Lámina 14 del procedimiento de configuración de los bloques de conversión AD.

3. Realice el siguiente programa en Verilog. Para mayor comprensión, la figura 7.24 muestra el diagrama estructural del programa.

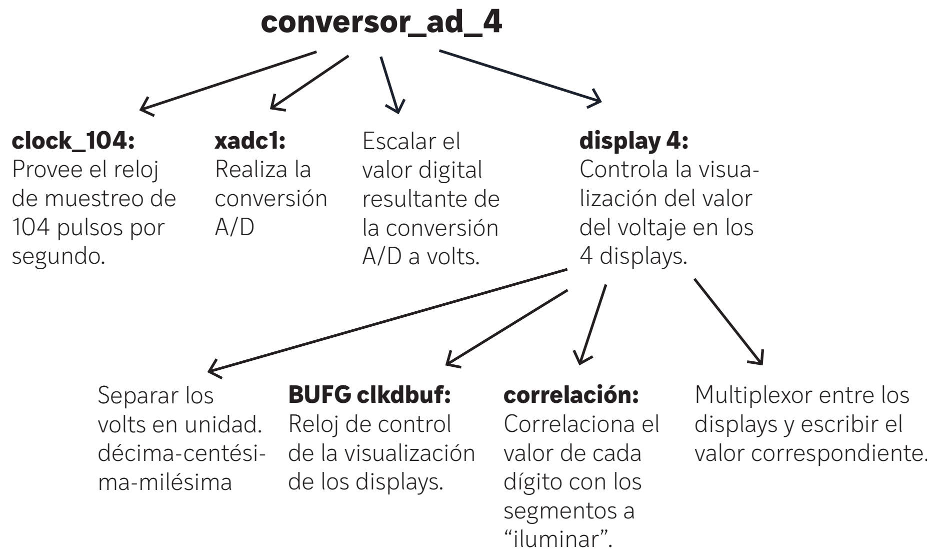


Figura 7.24. Diagrama estructural del programa de conversión AD.

```
// Módulo principal del conversor análogo a digital.
module conversor_ad_4 (
        input reloj_100MHz,             // Reloj de 100 MHz del sistema.
        input reset,                    // Botón de reinicio.
        input adc_n, adc_p,             // Entradas al conversor (VCAUX6 P y N).
        output [6:0] segmento,          // Segmentos del display.
        output punto_dec,               // Punto decimal del display.
        output [3:0] num_display );     // Selector del display.
```

```
// Bloque de reloj.
// El muestreo está determinado por el reloj de entrada al XADC y un divisor de reloj interno.
// El XADC toma 26 pulsos del reloj para una conversión.
// Se especifica 1 MSps, por lo que
// para obtener el reloj de entrada con un divisor de 4
// se multiplica 4x26x1Sps = 104 Sps.

// Instanciación al módulo que provee un reloj de 104 Sps.
wire locked, clock104;
clock_104 (
        .reset(reset),
        .clk_in1(reloj_100MHz),
        .locked(locked),
        .clk_out1(clock104) );

// Bloque XADC de conversión.
wire [6:0] daddr_in = 7'h16;
wire adc_ready, isbusy, adc_data_ready, eos_out, alarm;
wire [15:0] adc_data;
wire [4:0] channel_out;
// Instanciación al módulo que convierte análogo a digital.
xadc1 (
        .daddr_in(7'h16),                   // Especifica los pines vcaux6
                                            // para digitalizar.
        .dclk_in(clock104),                 // Reloj de 50MHz.
        .den_in(adc_ready),                 // Indica al adc que convierta.
        .di_in(16'h0),                      // Datos de entrada, no usados aquí.
        .dwe_in(1'b0),                      // Habilitar escritura en di_in, no usado aquí.
        .vauxp6(adc_p),                     // Entrada positiva para digitalizar.
        .vauxn6(adc_n),                     // Entrada negativa para digitalizar.
        .busy_out(isbusy),                  // Indica que el adc está ocupado
                                            // convirtiendo.
        .channel_out(channel_out[4:0]),     // canal de salida, no usado aquí.
        .do_out(adc_data),                  // Valor de la conversión adc.
        .drdy_out(adc_data_ready),          // Indica datos listos
                                            // para ser capturados.
        .eoc_out(adc_ready),                // Indica que la conversión
                                            // está completa.
        .eos_out(eos_out),                  // Indica que la secuencia de conversión está completa.
        .alarm_out(alarm),                  // Salida OR de todas las alarmas internas, no usado aquí.
        .vp_in(1'b0),                       // Par de entrada analógica diferencial, conectado a 0 si no se usa.
        .vn_in(1'b0)        );
```

```
		// Escala el valor del ADC (0-65535) a milivoltios (0-3300)
		reg [15:0] dato_escalado;
		always@ (posedge reloj_100MHz) begin
			if (reset)
				dato_escalado <= 16'd0;
			else
				if (adc_data_ready)
					dato_escalado <= (adc_data * 3300) /
					65535;	// Conversión a milivolts.
		end

		// Control de la visualización en los 4 displays.
		wire [15:0] display_actual;
		// Instanciación al módulo que muestra la conversión en los displays.
			display4 (
			.clk100(reloj_100MHz),		// Reloj de 100 MHz para la operación de los displays.
			.numero(display_actual),
			.display(num_display),
			.segmentos(segmento),
			.punto(punto_dec) );

		// Crear un reloj de ~1,5Hz para ejecutar más lenta la visualización en los leds.
		reg [26:0] contador1;
		reg [15:0] s_adc_data;
		always@ (posedge reloj_100MHz) begin
			if (reset)
				contador1 <= 0;
			else
				contador1 <= contador1 + 1;
		end

		wire reloj_1_5Hz = contador1[26];
		always@ (posedge reloj_1_5Hz)
			s_adc_data <= dato_escalado;

		assign display_actual = s_adc_data;

endmodule


// Módulo display4 para la visualización en los displays de 7 segmentos.
module display4 (
		input clk100,				// Reloj de 100 MHz.
```

```
output reg [3:0] display,          // Display a activar.
output reg [7:0] segmentos,        // Segmentos del display.
output reg punto,                  // Punto decimal.
input [15:0] numero,               // Valor a mostrar en el display.
input reset );

// Separar el número en milivoltios en 4 dígitos decimales
wire [3:0] milesima   = (numero / 1000) % 10;
wire [3:0] centesima= (numero / 100) % 10;
wire [3:0] decima     = (numero / 10) % 10;
wire [3:0] unidad     = numero % 10;

// Crear un reloj de ~760mseg a partir del reloj de 100MHz
reg [17:0] contador2 = 0;
always@ (negedge clk100)
        contador2 <= contador2 + 1;

wire digit_clock = contador2[17];
wire reloj_100MHz;

BUFG clkdbuf (
        .I(digit_clock),
                .O(reloj_100MHz));

reg [1:0] digito_x = 0;
always@ (posedge reloj_100MHz)
        digito_x <= digito_x + 1;

// Instanciación al módulo que relaciona el valor con los segmentos a iluminar.
wire [6:0] wseg0, wseg1, wseg2, wseg3;
correlacion C0 (.clk(clk100), .valor(unidad),        .seg(wseg0));
correlacion C1 (.clk(clk100), .valor(decima),        .seg(wseg1));
correlacion C2 (.clk(clk100), .valor(centesima),     .seg(wseg2));
correlacion C3 (.clk(clk100), .valor(milesima),      .seg(wseg3));

// Alternar entre los dígitos y asignar los segmentos correspondientes.
always@ (posedge reloj_100MHz) begin
        if (reset) begin
                display <= 4'b1111;          // Apagar todos los displays.
                segmentos <= 7'b1111111;     // Apagar todos los segmentos.
                punto <= 1;                  // Apagar el punto decimal.
        end
```

```
				else
				begin										// Activa 1 display y escribe el valor del
				case( digito_x )								// dígito correspondiente.
						‘h0: begin
								display <= ‘b1110;		// Activar el display 1.
								segmentos <= wseg0;		// Asignar los segmentos del dígito 1.
								punto <= 1;				 // Apagar el punto del Display 1.
								end
						‘h1: begin
								display <= ‘b1101;		 // Activar el display 2.
								segmentos <= wseg1;		// Asignar los segmentos del dígito 2.
								punto <= 1;				// Apagar el punto del Display 2.
						end
						‘h2: begin
								display <= ‘b1011;		// Activar el display 3.
								segmentos <= wseg2;		// Asignar los segmentos  del dígito 3.
								punto <= 1;				// Apagar el punto del Display 3.
						end
						‘h3: begin
								display <= ‘b0111;		// Activar el display 4.
								segmentos <= wseg3;		// Asignar los segmentos del dígito 4.
								punto <= 0;				// Activar el punto del Display 4.
						end
						endcase
				 end
		end
endmodule

module correlacion (
		input clk,
		input [3:0] valor,
		output reg [6:0] seg );

		// Definir los valores para los segmentos de 0 a 9.
		always@ (*) begin
				case (valor)
						4’d0: seg = 7’b1000000;
						4’d1: seg = 7’b1111001;
						4’d2: seg = 7’b0100100;
						4’d3: seg = 7’b0110000;
						4’d4: seg = 7’b0011001;
						4’d5: seg = 7’b0010010;
```

```
                    4'd6: seg = 7'b0000010;
                    4'd7: seg = 7'b1111000;
                    4'd8: seg = 7'b0000000;
                    4'd9: seg = 7'b0010000;
                    default: seg = 7'b1111111;      // apagado
                endcase
            end
endmodule
```

4. Obtenga el esquemático del circuito resultante (figura 7.25). Tenga en consideración que Vivado permite hacer zoom para mejorar la visualización según se desee.

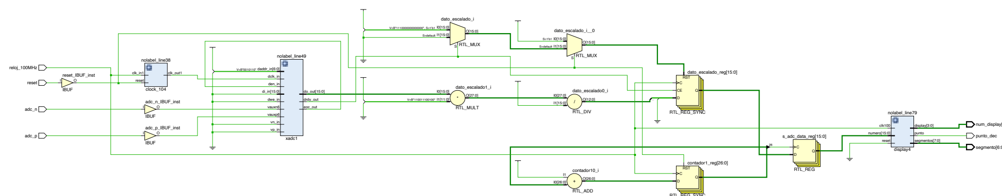


Figura 7.25. Esquemático del circuito conversor AD.

5. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación.

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {reloj_100MHz}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {segmento[0]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {segmento[1]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {segmento[2]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {segmento[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {segmento[4]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {segmento[5]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {segmento[6]}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {punto_dec}]

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {num_display[0]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {num_display[1]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {num_display[2]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {num_display[3]}]

##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {reset}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]

##Pmod Header JXADC
set_property -dict { PACKAGE_PIN J3   IOSTANDARD LVCMOS33 } [get_ports {adc_p}]
#set_property -dict { PACKAGE_PIN L3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[1]}]

set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_ports {adc_n}]
```

```
set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_ports {adc_n}]
#set_property -dict { PACKAGE_PIN M3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}]
```

6. Sintetice, implemente, genere el *bitstream*, cargue el programa y compruebe la correcta conversión.

7.2.6. PRÁCTICA 16: COMUNICACIÓN SERIAL UART

Esta práctica está concebida para reforzar el conocimiento de los fundamentos de la comunicación serial a partir de un ejercicio simplificado de generación del protocolo UART «desde cero» y transmisión de un dato desde una tarjeta a otra. Para esto, un programa de transmisión dispuesto en una de las tarjetas crea la trama de pulsos correspondientes a los «bit de inicio, 8 bits de datos y 1 bit de stop» (no se incluye bit de paridad) y lo transmite como un registro de desplazamiento a 9.600 baudios por una salida PMod de la tarjeta (JB1) hasta otra tarjeta que contiene el programa de recepción (entrada por la PMod JA1), según se ilustra en la figura 7.26. En consecuencia, se tienen dos programas: uno de transmisión residente en una tarjeta y otro de recepción en la otra.

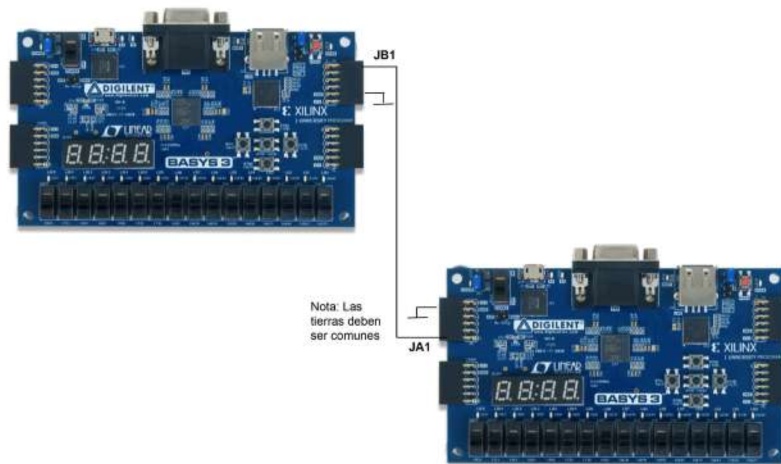


Figura 7.26. Esquema de interconexión de las tarjetas BASYS-3.

1. Realice el siguiente programa de transmisión en Verilog en la primera tarjeta.

```
// PROGRAMA DEL "TRANSMISOR" UART.
   module TX_UART (
   input wire reloj_100MHz,         // Reloj del sistema.
   input wire reset,                // Botón de reset.
   input wire carga_desplaza,       // Switch para habilitar la carga de la data o
                                    // el desplazamiento de la trama.
   input wire [7:0] dato,           // Dato de 8 bits a transmitir, generado por
                                    // 8 switches.
   output wire [9:0]led,            // Leds para visualizar la data.
   output wire salida_tx );         // Puerto por el cual se transmite serialmente
                                    // el contenido de la trama (PMod JB1).
   reg [9:0] trama_tx;              // Registro de 10 bits a desplazar (1 start bit,
                                    // 8 bits de datos y 1 stop bit).
   wire reloj_nuevo;                // Permite operar a 9600 bps.
```

```
// Llamado al divisor de frecuencia que divide la frecuencia
// original del reloj a la necesaria para transmitir a 9600 bps.
  divisor_frecuencia (
    .reloj_entrada(reloj_100MHz),
    .reloj_salida(reloj_nuevo) );

  always@ (posedge reloj_nuevo, posedge reset) begin
  if (reset)
    trama_tx <= 10'b1000000000;                // Inicializar la trama con 0 (start bit) y
                                               // con 1 (stop bit) en los extremos.
  else
    if (carga_desplaza == 1)
      trama_tx <= {1'b1, dato , 1'b0};         // Carga el dato y estructura la trama
                                               // concatenando 10 bits (start bit en 0,
                                               // los 8 bits de datos y el stop bit en 1).
    else
      trama_tx <= {1'b1, 1'b1, trama_tx[8:1]}; // Con cada pulso de reloj del always
                                               // se reestructura la trama
                                               // reposicionando los bits 1 a 9 en las
                                               // posiciones 0 a 8 y completando la
                                               // concatenación (los 10 bits) rellenando
                                               // el bit 9 con el último valor de la trama,
                                               // en este caso un 1.
  end

  assign salida_tx = (carga_desplaza | trama_tx[0]);  // Asigna a la salida tx (puerto
                                               // JA1) un 1 permanente cuando la variable
                                               // carga_desplaza está en 1 (mientras carga el dato desde
                                               // los switches) o el bit menos significativo de la trama
                                               // (carga_desplaza en 0). O sea, cuando ocurre la
                                               // transmisión serial propiamente tal.
  assign led = trama_tx;                       // Asigna la trama a los LEDs para visualización (se
                                               // encienden todos por la velocidad de transmisión).
endmodule

// INSTANCIA DIVISORA DE FRECUENCIA.
module divisor_frecuencia (
  input wire reloj_entrada,                    // Señal de reloj de entrada.
  output wire reloj_salida );                  // Señal de reloj de salida.
```

```
    parameter Seg1 = 10_417;      // 100M/10417 = 9600 bps.
    localparam T = $clog2(Seg1);
    reg [Nbits-1:0] contador_divisor1 = 0;

    always@(posedge reloj_entrada)
        if (contador_divisor1 == Seg1-1)
            contador_divisor1 <= 0;
        else
            contador_divisor1 <= contador_divisor1 + 1;
    assign reloj_salida = contador_divisor1[Nbits-1];
endmodule
```

2. Obtenga el esquemático del circuito resultante (figura 7.27). Tenga en consideración que Vivado permite hacer zoom para mejorar la visualización según se desee.

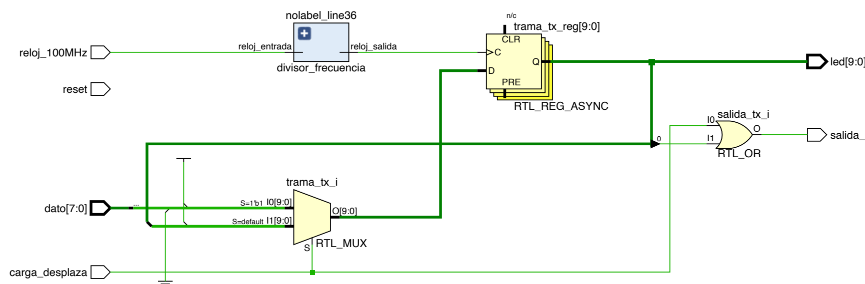


Figura 7.27. Esquemático del circuito transmisor UART.

3. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación.

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {reloj_100MHz}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]


## Switches
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {sw[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {dato[0]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {dato[1]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {dato[2]}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {dato[3]}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {dato[4]}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {dato[5]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {dato[6]}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {dato[7]}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw[9]}]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {sw[10]}]
#set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {sw[11]}]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {sw[12]}]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {sw[13]}]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {sw[14]}]
set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {carga_desplaza}
```

```
## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led[3]}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led[6]}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led[7]}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {led[8]}]
set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led[9]}]
#set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {led[10]}]
#set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {led[11]}]

##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports btnR]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {reset}]

##Pmod Header JB
set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports {salida_tx}]
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {JB[1]}]
#set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports {JB[2]}]
```

4. Realice el siguiente programa de recepción en Verilog, en la segunda tarjeta.

```
// PROGRAMA DEL "RECEPTOR" UART.
module RX_UART(
        input wire reloj_100MHz,        // Reloj del sistema.
        input wire reset,               // Variable para aplicar reset.
        input wire recepcion,           // Entrada datos seriales rx.
        output reg [7:0] dato,          // Salida de datos recibidos (8 bits).
        output reg dato_listo,          // Salida que indica que los datos fueron recibidos.
        output led_reloj        );      // Led indicador del reloj de salida del divisor.

        wire reloj_nuevo;               // Permite operar a 9600 bps.
        reg [3:0] contador_bit;         // Contador de bits recibidos.
        reg [7:0] registro_desplaza;    // Registro de desplazamiento para almacenar los 8 bits recibidos.
        reg registro_rx ;               // Registro para la sincronización de la señal de recepcion.
        reg estado;                     // Estado del receptor (0: en espera, 1: recibiendo datos)

        // Llamado al divisor de frecuencia, que divide la frecuencia
        // original del reloj a la necesaria para recibir a 9.600 baudios.
        divisor_frecuencia      (
                .reloj_entrada(reloj_100MHz),
                .reloj_salida(reloj_nuevo) );
        assign led_reloj = reloj_nuevo;
```

```
        localparam IDLE = 0;
        localparam RECEIVING = 1;

        always @(posedge reloj_nuevo or posedge reset) begin
              if (reset) begin
                    contador_bit <= 0;
                    registro_desplaza <= 8'b0;
                    registro_rx <= 1'b1;                    // RX en nivel alto (reposo)
                    estado <= IDLE;
                    dato_listo <= 0;
                    dato <= 8'b0;                           // Reiniciar la salida de datos
              end
              else begin
                    registro_rx <= recepcion;               // Sincronización de la señal recepción.
                    case (estado)
                    IDLE: begin
                    dato_listo <= 0;
                    if (registro_rx == 1'b0) begin          // Detectar el bit de inicio (nivel bajo)
                    estado <= RECEIVING;
                    contador_bit <= 0;
                    end
              end

               RECEIVING: begin
               if (contador_bit < 8) begin                  // Leer los 8 bits de datos
                 registro_desplaza <= {registro_rx, registro_desplaza[7:1]};
                 contador_bit <= contador_bit + 1;
               end
               else begin                                   // Leer el bit de parada
               if (registro_rx == 1'b1) begin               // Verificar que el bit de parada sea alto.
                 dato <= registro_desplaza;
                 dato_listo <= 1'b1;
               end
               estado <= IDLE;                              // Volver al estado de reposo.
               end
               end
               endcase
            end
        end
endmodule
```

```
// INSTANCIA DIVISORA DE FRECUENCIA
module divisor_frecuencia(
        input wire reloj_entrada,
        output wire reloj_salida);
        parameter S = 100_000_000;          // 100M/10417 = 9600 bps.
        localparam T = $clog2(S);
        reg [T-1:0] contador_divisor1 = 0;
        always@(posedge reloj_entrada)
        if (contador_divisor1 == S-1)
                contador_divisor1 <= 0;
        else
                contador_divisor1 <= contador_divisor1 + 1;
        assign reloj_salida = contador_divisor1[T-1];
endmodule
```

5. Obtenga el esquemático del circuito resultante (figura 7.28). Tenga en consideración que Vivado permite hacer zoom para mejorar la visualización según se desee.

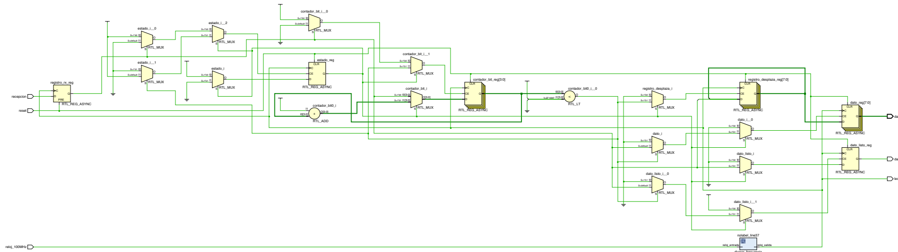


Figura 7.28. Esquemático del circuito receptor UART.

6. Asigne los recursos de la tarjeta al circuito diseñado. Una posible asignación es la que se muestra a continuación.

```
'# Clock signal
et_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {reloj_100MHz}]
'create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

'# LEDs
et_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led_reloj}]
'set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
'set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
'set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led[3]}]
'set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
'set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
et_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {dato_listo}]
'set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led[7]}]
et_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {dato[0]}]
et_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {dato[1]}]
et_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {dato[2]}]
et_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {dato[3]}]
et_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {dato[4]}]
et_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {dato[5]}]
et_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {dato[6]}]
et_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {dato[7]}]
```

```
##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports {reset}]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]

##Pmod Header JA
set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {recepcion}]
#set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports {JA[1]}]
#set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {JA[2]}]
```

7. Sintetice, implemente, genere el *bitstream* y cargue los programas en las tarjetas respectivas. Compruebe la correcta transmisión y recepción.

ANEXOS

A.1. Uso de Vivado

Una vez instalado el software Vivado y el archivo de recursos «Basys-3-Master.xdc», proceda como sigue.

A.1.1. CREACIÓN DE UN PROYECTO BASYS-3/FPGA

1. Abrir Vivado y espere que se abra la ventana de inicio «Quick Start».

2. En «Quick Start», inicie la creación de un proyecto con «Create Project» y aplicar «Next».

3. Escriba el nombre del proyecto en la ventana «Project Name» y defina su localización en «Project Location». Verifique que esté marcada la opción «Create Project Subdirectory» y aplique «Next».

4. Verifique que en la ventana «Project Type» esté marcada la opción «RTL Project» y aplique «Next».

5. En la ventana «Add Sources», aplique «Create File». En el campo «File Name» de la nueva ventana que se abre, escriba el nombre del programa principal a generar. Aplique «OK».

6. En la ventana «Add Sources», marque la nueva línea que tiene el nombre del archivo recién definido y aplique «Next».

7. En la nueva ventana «Add Constraints», marque «Add Files» y busque el directorio «digilent-xdc-master».

8. Marque dos veces sobre el directorio «digilent-xdc-master» para desplegar todos los tipos de tarjetas con las que trabaja Vivado.

9. Seleccione el archivo «Basys-3-Master.xdc» y aplique «OK».

10. Verifique que en la ventana «Add Constraints» esté seleccionado «Copy Constraints Files Into Project» y aplique «Next».

11. En la ventana «Default Part», use el buscador «Search» para localizar y seleccionar la versión XC7A35TICPG236-1L de la FPGA. Aplique «Next».

12. Aparecerá una ventana con el resumen del proyecto. Aplique «Finish» y espere a que se cree el proyecto y que aparezca la ventana «Define Module» con el nombre del programa especificado en el punto 5. Aplique «OK» y luego «Yes». Espere a que se despliegue la ventana de trabajo de Vivado.

A.1.2. ESCRIBIR UN PROGRAMA EN VERILOG Y ASIGNAR RECURSOS DE LA TARJETA

13. Marque dos veces el nombre del programa con extensión .v en el área «Sources» para iniciar la programación de un circuito. Con esto se abre una ventana a la derecha con la estructura inicial del programa a desarrollar, con el nombre y los primeros elementos de programación de Verilog.

14. Escriba las instrucciones en lenguaje Verilog para configurar el circuito deseado. Notar que un cuadrado en la parte superior izquierda se pone en «verde» cuando la sintaxis es correcta y en «rojo» en caso contrario, además de subrayar en «rojo» la línea de instrucción errónea (puede ser la línea subrayada, la inmediatamente superior o el propio bloque de programa). Tener presente que entre los errores más comunes están omitir las «comas» o «puntos y comas» al final de cada línea, poner «coma» cuando debe ser «punto y coma», o viceversa, repetir u omitir los «paréntesis», entre otros. Adicionalmente, considerar que si la línea de instrucción está con un «fondo azul» significa que existe una inconsistencia en la lógica de programación (por ejemplo, asignar un valor a una variable no declarada previamente), o si está con «fondo amarillo» significa una alerta de una posible inconsistencia asociada a alguna variable.

15. Como un paso opcional, en la sección «RTL Analisys» del área de la izquierda marque «Open Elaborated Design», aplique «Save» y «OK», y espere a que se abra la ventana «Schematic» con el esquemático del circuito. Tenga presente que cada vez que se modifique un programa y se desee observar el nuevo esquemático, se deben guardar los cambios haciendo clic en el ícono «Guardar» de la ventana «Sources» y actualizar el esquemático aplicando «Reload» en la línea amarilla superior que se abre.

16. Marque dos veces en la carpeta «Constraints» de la ventana «Sources» para iniciar la asignación de recursos de la tarjeta.

17. Marque dos veces en la identificación del archivo en uso («Basys-3-Master.xdc», en este caso). A la derecha se desplegará una ventana con el conjunto de recursos que tiene esta tarjeta.

18. Quite el símbolo «#» al comienzo de las líneas asociadas a los recursos que requiere el circuito y cambie su identificación al final de la línea por los nombres especificados en su programa (el nombre que está en los paréntesis de llaves). No debe modificar la estructura de las líneas, solo quite el símbolo «#» al comienzo de la línea y reemplace el nombre de la variable al final; cualquier otra alteración produce un error en las etapas siguientes (por ejemplo, un error común es quitarle los paréntesis de llaves a las variables o el paréntesis cuadrado al final de la línea).

A.1.3. SINTETIZAR E IMPLEMENTAR EL CIRCUITO Y GENERAR EL BITSTREAM

19. En la sección «SYNTHESIS» de la ventana izquierda, marque «Run Synthesis», «Save» si se solicita y «OK» en la ventana «Launch Runs» que se abre. Espere a que la síntesis se complete (ver arriba a la derecha el ícono que indica que la síntesis está en progreso). Cuando haya concluido, se abrirá una ventana que indica «Synthesis Successfully Completed» (ver «View Messages» y «Messages» para analizar los errores).

20. En la sección «IMPLEMENTATION» de la ventana izquierda (o en la misma ventana anterior que señala el término de la síntesis), marque «Run Implementation» y «OK» en la ventana «Launch Runs» que se abre. Espere a que la implementación se complete (ver arriba a la derecha el ícono que indica que la implementación está en progreso). Cuando haya concluido, se abrirá una ventana que indica «Implementation Successfully Completed» (ver «View Messages» y «Messages» para analizar los errores).

21. Dentro de la sección «PROGRAM AND DEBUG» de la ventana izquierda, marque «Generate Bitstream» y «OK» en la ventana «Launch Runs» que se abre. Espere a que la generación del *bitstream* se complete (ver arriba a la derecha el ícono que indica que el *bitstream* está en progreso). Cuando haya concluido, se abrirá una ventana que indica «Bitstream Generation Successfully Completed» (ver «View Messages» y «Messages» para analizar los errores).

22. Verifique que el jumper selector de modo de carga de la tarjeta esté en la posición central JTAG (componente 10 de la figura 2.1).

23. Conecte el computador a la tarjeta en la puerta USB/JTAG/UART (componente 13 de la figura 2.1) y aplique energía. Observe que el led de encendido cambie a rojo y se escuchen los tonos audibles de reconocimiento de conexión del computador.

24. Dentro de la sección «PROGRAM AND DEBUG», aplique «Open Hardware Manager», «Open Target» en la barra verde superior y luego «Auto Connect».

25. Aplique «Program Device» en la barra verde superior y busque el directorio en donde se encuentra localizado el proyecto (use los «tres puntos» del campo «*bitstream file*»; el path aparece en la línea superior de la ventana de trasfondo). Marque el archivo con el nombre asignado al proyecto en el paso 3.

26. Seleccione «nombre del proyecto.runs» e «impl_1». Con esta acción, aparecerá el archivo «nombre del programa.bit» a descargar a la tarjeta.

27. Marque «nombre del programa.bit». Con esta acción se autocompletará el path del archivo en el campo «Bitstream File».

28. Aplique «Program», espere a que se traspase el programa y verifique que se ilumina en verde el led «DONE» de la tarjeta (componente 8 de la figura 2.1). Esto indica que la carga fue realizada correctamente y que el programa ya está funcionando.

A.2. Recursos de la tarjeta BASYS-3

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level
## - signal names in the project
## Clock signal
#set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports {clk}]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}]


## Switches
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {sw[0]}]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {sw[1]}]
#set_property -dict { PACKAGE_PIN W16  IOSTANDARD LVCMOS33 } [get_ports {sw[2]}]
#set_property -dict { PACKAGE_PIN W17  IOSTANDARD LVCMOS33 } [get_ports {sw[3]}]
#set_property -dict { PACKAGE_PIN W15  IOSTANDARD LVCMOS33 } [get_ports {sw[4]}]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {sw[5]}]
#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {sw[6]}]
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {sw[7]}]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {sw[8]}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw[9]}]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {sw[10]}]
#set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {sw[11]}]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {sw[12]}]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {sw[13]}]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {sw[14]}]
#set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {sw[15]}]


## LEDs
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led[0]}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led[3]}]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led[6]}]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led[7]}]
#set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {led[8]}]
#set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led[9]}]
#set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {led[10]}]
#set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {led[11]}]
#set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {led[12]}]
#set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {led[13]}]
#set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {led[14]}]
#set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {led[15]}]


##7 Segment Display
#set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {seg[0]}]
#set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {seg[1]}]
```

```
#set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {seg[2]}]
#set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {seg[3]}]
#set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {seg[4]}]
#set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {seg[5]}]
#set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {seg[6]}]

#set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports {dp}] ;#Punto

#set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {an[0]}]
#set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {an[1]}]
#set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {an[2]}]
#set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {an[3]}]


##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports btnR]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]


##Pmod Header JA
#set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {JA[0]}] ;#JA1
#set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports {JA[1]}] ;#JA2
#set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {JA[2]}] ;#JA3
#set_property -dict { PACKAGE_PIN G2   IOSTANDARD LVCMOS33 } [get_ports {JA[3]}] ;#JA4
#set_property -dict { PACKAGE_PIN H1   IOSTANDARD LVCMOS33 } [get_ports {JA[4]}];# JA7
#set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports {JA[5]}];# JA8
#set_property -dict { PACKAGE_PIN H2   IOSTANDARD LVCMOS33 } [get_ports {JA[6]}];# JA9
#set_property -dict { PACKAGE_PIN G3   IOSTANDARD LVCMOS33 } [get_ports {JA[7]}];# JA10

##Pmod Header JB
#set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports {JB[0]}];# JB1
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {JB[1]}];# JB2
#set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports {JB[2]}];# JB3
#set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 } [get_ports {JB[3]}];# JB4
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports {JB[4]}];# JB7
#set_property -dict { PACKAGE_PIN A17   IOSTANDARD LVCMOS33 } [get_ports {JB[5]}];# JB8
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports {JB[6]}];# JB9
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports {JB[7]}];#JB10

##Pmod Header JC
#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports {JC[0]}];# JC1
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {JC[1]}];# JC2
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {JC[2]}];# JC3
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports {JC[3]}];# JC4
#set_property -dict { PACKAGE_PIN L17   IOSTANDARD LVCMOS33 } [get_ports {JC[4]}];# JC7
#set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports {JC[5]}];# JC8
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {JC[6]}];# JC9
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {JC[7]}];# JC10
```

```
##Pmod Header JXADC
#set_property -dict { PACKAGE_PIN J3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[0]}];#XA1_P
#set_property -dict { PACKAGE_PIN L3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[1]}];#XA2_P
#set_property -dict { PACKAGE_PIN M2   IOSTANDARD LVCMOS33 } [get_ports {JXADC[2]}];#XA3_P
#set_property -dict { PACKAGE_PIN N2   IOSTANDARD LVCMOS33 } [get_ports {JXADC[3]}];#XA4_P
#set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[4]}];#XA1_N
#set_property -dict { PACKAGE_PIN M3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}];#XA2_N
#set_property -dict { PACKAGE_PIN M1   IOSTANDARD LVCMOS33 } [get_ports {JXADC[6]}];#XA3_N
#set_property -dict { PACKAGE_PIN N1   IOSTANDARD LVCMOS33 } [get_ports {JXADC[7]}];#XA4_N


##VGA Connector
#set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[0]}]
#set_property -dict { PACKAGE_PIN H19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[1]}]
#set_property -dict { PACKAGE_PIN J19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[2]}]
#set_property -dict { PACKAGE_PIN N19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[3]}]
#set_property -dict { PACKAGE_PIN N18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[0]}]
#set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[1]}]
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[2]}]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[3]}]
#set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[0]}]
#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[1]}]
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[2]}]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[3]}]
#set_property -dict { PACKAGE_PIN P19   IOSTANDARD LVCMOS33 } [get_ports Hsync]
#set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports Vsync]



##USB-RS232 Interface
#set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports RsRx]
#set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVCMOS33 } [get_ports RsTx]


##USB HID (PS/2)
#set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33   PULLUP true } [get_ports PS2Clk]
#set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33   PULLUP true } [get_ports PS2Da-
ta]


##Quad SPI Flash
##Note that CCLK_0 cannot be placed in 7 series devices. You can access it using the
##STARTUPE2 primitive.
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[0]}]
#set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[1]}]
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[2]}]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[3]}]
#set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports QspiCSn]


## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

Este libro ofrece una referencia esencial para estudiantes y personas vinculadas al diseño de circuitos digitales, centrada en los arreglos de compuertas programables en campo (FPGA) y su aplicación práctica. Aborda los aspectos conceptuales y técnicos necesarios para diseñar e implementar, con flexibilidad y precisión, circuitos digitales de diversa complejidad: desde combinacionales básicos hasta secuenciales, de control y automatización, incluyendo funciones lógicas, contadores, multiplexores, registros de desplazamientos, conversores de bases, interfaces y controladores.

La descripción es directa y suficiente para que quienes se inician en los sistemas digitales puedan crear y probar soluciones utilizando la tarjeta BASYS-3 de Digilent, que integra la FPGA Artix-7 de Xilinx (versión XC7A35TICPG236C-1L). Se incluye un conjunto de experiencias prácticas de complejidad creciente que permiten comprender y aplicar paso a paso los conceptos y técnicas abordados.

Estas experiencias, tipo laboratorio, orientan al lector en el uso autónomo de la FPGA y los recursos de la BASYS-3, incorporando el aprendizaje del lenguaje de descripción de hardware (HDL) Verilog, necesario para programar la tarjeta y comprender la diferencia entre programar hardware y software, así como la relevancia de las soluciones en tiempo real.

ISBN: 978-956-356-134-0
9 789563 561340