



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA EN INGENIERÍA INFORMÁTICA

Plataforma didáctica para desarrollo de sistemas basados en FPGAs de Xilinx

Autora
Elena Cantero Molina

Directora
María Begoña del Pino Prieto



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
GRANADA, 8 DE SEPTIEMBRE DE 2020

Plataforma didáctica para desarrollo de sistemas basados en FPGAs de Xilinx

Elena Cantero Molina

Palabras clave: FPGA, Xilinx, VHDL, Vivado, Síntesis RT-Lógica

Resumen

En este trabajo de fin de grado se ha realizado un estudio de una plataforma didáctica para desarrollo de sistemas basados en FPGAs de Xilinx. Una **FPGA** es un dispositivo semiconductore basado en matrices de bloques lógicos configurables que están conectados mediante interconexiones programables. Para programarlas existen tres principales tecnologías, la tecnología Antifusible que no es reprogramable, la tecnología SRAM que son reprogramables pero con memorias son volátiles y la tecnología Flash que es reprogramable pero con memoria no volátil.

Actualmente las FPGAs pueden ser usadas para implementar sistemas en distintos ámbitos como por ejemplo Aeroespacial y defensa, Centro de procesamiento de datos, Industria, Medicina o Comunicaciones.

Dentro de los principales fabricantes de FPGAs, destaca **Xilinx** como el principal fabricante, seguido de Intel. Las características de sus dispositivos y herramientas software de desarrollo son descritas en esta memoria.

La tarjeta que se va a usar en este proyecto para el desarrollo de este proyecto es la tarjeta **ZYBO** que incluye una FPGA de la familia Zynq-7000 de Xilinx. Esta FPGA incluye un sistema de procesamiento basado en cores ARM empotrados, memoria “on-chip”, interfaces de memoria externa y lógica genérica basada en tecnología SRAM. La arquitectura de esta FPGA permite la implementación de lógica personalizada para configurar módulos hardware específicos y la ejecución de software en los procesadores empotrados, los cuales son expli-cados en el desarrollo de la memoria.

Xilinx ha sido una empresa pionera en la comercialización de herramientas de síntesis de alto nivel para describir componentes hardware a partir de descripciones C/C++ con el módulo Vivado HLS que se ha utilizando en asignaturas de perfil avanzado. En este trabajo se plantea la migración a Vivado de prácticas de laboratorio de un asignatura de carácter más básico donde se estudia el diseño de componentes a partir de herramientas de síntesis RT-lógica en las que se utilizan lenguajes de descripción hardware tipo **VHDL** o **Verilog**. El lenguaje usado en este trabajo es VHDL.

El objetivo de este trabajo es conocer la plataforma **Vivado** y realizar un flujo de diseño apoyándonos en la propia tarjeta. Vivado está preparado para la síntesis y análisis de diseños HDL. Para ello, hay que conocer las posibilidades que tiene Vivado para cada fase del flujo de diseño. Así, se ha estudiado cómo se implementan las diferentes fases del flujo de diseño con esta herramienta. A continuación, se ha realizado una descripción de los principales módulos que se van a usar en la posterior realización de casos prácticos. Estos módulos son los más relevantes para la finalidad de este proyecto, existiendo muchos más. En concreto son, un procesador didáctico, una memoria RAM, un módulo generador de reloj y un controlador VGA.

Después, como casos prácticos se detallan un computador básico, en el que se usa el bloque de memoria RAM, y la visualización en pantalla de una bola moviéndose en ella, en el que se usa el módulo generador de reloj y el controlador VGA.

Por último se realiza la conclusión final del trabajo, donde se muestra si se han conseguido los objetivos planteados y además se incluyen otras vías futuras que se pueden realizar a partir de este proyecto.

Educational platform for development of FPGA-based systems from Xilinx

Elena Cantero Molina

Keywords: FPGA, Xilinx, VHDL, Vivado, RT-Logic synthesis

Abstract

In this final degree project, a study of an educational platform for the development of systems based on Xilinx FPGAs has been carried out. An FPGA is a semiconductor device based on configurable logic blocks that are connected by programmable interconnections. To program them there are three main technologies, the Antifuse technology, that is not reprogrammable, the SRAM technology that is reprogrammable but with volatile memories and the Flash technology that is reprogrammable but with non-volatile memory.

Currently FPGAs can be used to implement systems in different areas such as Aerospace and Defense, Data Processing Center, Industry, Medicine or Communications.

Among the main FPGA manufacturers, Xilinx stands out as the main manufacturer, followed by Intel. The characteristics of its devices and development software tools are described in this report.

The board that will be used in this project for the development of this project is the ZYBO board that includes an FPGA of the Xilinx Zynq-7000 family. This FPGA includes a processing system based on embedded ARM cores, "on-chip" memory, external memory interfaces and generic logic based on SRAM technology. The architecture of this FPGA allows the implementation of custom logic to configure specific hardware modules and the execution of software in the embedded processors, which are explained in the memory development.

Xilinx has been a pioneer in the commercialization of high-level synthesis tools to describe hardware components from C / C ++ descriptions with the Vivado HLS module that has been used in advanced profile courses. In this work, the migration to Vivado of laboratory practices of a subject of a more basic nature is proposed where the design of components is studied from RT-logic synthesis tools in which hardware description languages such as VHDL or Verilog are used. The language used in this work is VHDL.

The objective of this work is to know the Vivado platform and carry out a design flow based on the board itself. Vivado is prepared for the synthesis and analysis of HDL designs. To do this, you must know the possibilities that Vivado has for each phase of the design flow. Thus, it has been studied how the different phases of the design flow are implemented with this tool. Next, a description of the main modules that will be used in the subsequent realization of practical cases has been made. These modules are the most relevant for the purpose of this project, there are many more. Specifically, they are an educational processor, a RAM memory, a clock generator module and a VGA controller.

Then, as practical cases, a basic computer is detailed, in which the RAM memory block is used, and the on-screen display of a ball moving in it, in which the clock generator module and the VGA controller are used.

Finally, the final conclusion of the work is carried out, where it is shown if the objectives have been achieved and also other future routes that can be carried out from this project are included.

Yo, **Elena Cantero Molina**, alumna de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 45744912M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Elena Cantero Molina

Granada a 08 de Septiembre de 2020.

Dña. **María Begoña del Pino Prieto**, Profesora del Área de Arquitectura y Tecnología de Computadores del Departamento Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado *Plataforma didáctica para desarrollo de sistemas basados en FPGAs de Xilinx*, ha sido realizado bajo su supervisión por **Elena Cantero Molina**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 08 de Septiembre de 2020.

La directora:

María Begoña del Pino Prieto

Agradecimientos

Me gustaría agradecer este trabajo a toda la gente que me ha hecho posible llegar hasta aquí, en especial a mis padres, porque a pesar de que no estén conmigo en estos momentos, confiaron en mí y me apoyaron incondicionalmente. También agradecere a mis titas y mi hermano por estar ahí en las malas situaciones y ayudarme a seguir hacia delante.

También me gustaría agradecer a todos los profesores de la facultad que me han ayudado a formarme, pero en especial a mi tutora María Begoña, una gran profesora y tutora que me ha dado la oportunidad de desarrollar este proyecto a su lado.

Por último, me gustaría darle las gracias a todos mis compañeros y amigos de facultad ya que ellos me han apoyado en momentos difíciles y que sin ellos no sería la persona que soy ahora.

Gracias.

Índice general

1. Motivación e introducción	1
1.1. Sistemas basados en dispositivos FPGAs	1
1.2. Niveles de síntesis automática	4
1.3. Plataformas de desarrollo	5
1.3.1. Principales fabricantes de FPGAs	6
1.3.2. Herramientas software de desarrollo	7
1.3.3. Plataformas de propósito académico	8
1.4. Estructura de la memoria	9
2. Objetivos del trabajo	11
3. Resolución del trabajo	13
3.1. Materiales	13
3.1.1. Familia Zynq-7000	13
3.1.2. Tarjeta Zybo	16
3.1.3. Vivado	18
3.2. Metodología	23
3.2.1. Diseño	23
3.2.2. Simulación	24
3.2.3. Síntesis	27
3.2.4. Implementación	27
3.2.5. Generación Bitstream	28
3.3. Módulos IP específicos	29
3.3.1. Procesador específico	31
3.3.2. Memoria RAM de un sólo puerto con entradas registradas	34
3.3.3. Módulo de generación de reloj	37
3.3.4. Controlador VGA	38
3.4. Casos prácticos	41
3.4.1. Computador básico	42
3.4.2. Visualización y movimiento de objetos en monitor VGA .	45
4. Conclusiones y vías futuras	49
Anexo	50
Bibliografia	58

Índice de figuras

1.1.	<i>Arquitectura de una FPGA [1]</i>	1
3.1.	<i>Arquitectura Zynq-7000 [38]</i>	14
3.2.	<i>ZYBO Zynq-7000 Development Board [11]</i>	17
3.3.	<i>Vivado IDE</i>	19
3.4.	<i>Entorno Principal Vivado IDE</i>	20
3.5.	<i>Flow Navigator</i>	21
3.6.	<i>Default Layout</i>	21
3.7.	<i>I/O Planning</i>	22
3.8.	<i>Floorplanning</i>	22
3.9.	<i>Timing Analysis</i>	22
3.10.	<i>Typical Xilinx FPGA Development Flow (Adaptada de [24])</i>	24
3.11.	<i>Analizador Temporal Implementación</i>	25
3.12.	<i>Hardware Manager</i>	29
3.13.	<i>IP Catalog</i>	30
3.14.	<i>Block Memory Generator</i>	34
3.15.	<i>Block Memory Generator Simulation (1)</i>	36
3.16.	<i>Block Memory Generator Simulation (2)</i>	36
3.17.	<i>Clocking Wizard</i>	37
3.18.	<i>Simulación Procesador (Parte 1)</i>	44
3.19.	<i>Simulación Procesador (Parte 2)</i>	44
3.20.	<i>Visualización Bola Cuadrada</i>	48
4.1.	<i>Netlist RT</i>	52
4.2.	<i>Simulación Funcional Contador 4 Bits</i>	53
4.3.	<i>Netlist Resultante Synthesis</i>	53
4.4.	<i>Simulación Funcional Post-Synthesis</i>	54
4.5.	<i>Resumen Temporal Synthesis</i>	54
4.6.	<i>Simulación Temporal Post-Synthesis</i>	55
4.7.	<i>Resumen Temporal Implementation</i>	56
4.8.	<i>Simulación Funcional Post-Implementation</i>	56
4.9.	<i>Simulación Temporal Post-Implementation</i>	56

Capítulo 1

Motivación e introducción

En este capítulo se explica la arquitectura de una FPGA en comparación con otros circuitos integrados digitales. Además se mencionan los campos de aplicación más importantes. Aquí también, se detallan los distintos niveles de síntesis automática (síntesis funcional, RT-lógica y física). Por último se hace mención a varios fabricantes de FPGAs con sus respectivos dispositivos y a las herramientas software de desarrollo, dando más importancia a las de Xilinx y algunas plataformas de propósito académico del fabricante Xilinx.

1.1. Sistemas basados en dispositivos FPGAs

Las FPGAs (*Field Programmable Gate Arrays*), son dispositivos semiconductores basados en matrices de bloques lógicos configurables (**CLB**) que están conectados mediante interconexiones programables (Figura 1.1) [31].

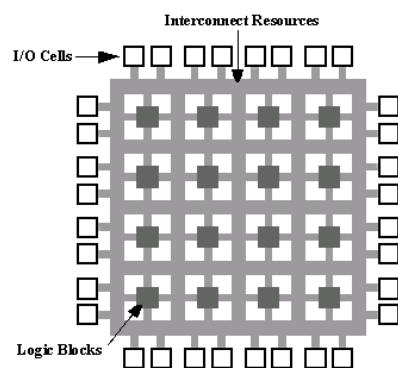


Figura 1.1: Arquitectura de una FPGA [1]

Los bloques lógicos configurables en las FPGAs basadas en celdas SRAM incluyen LUTs (*tablas de consulta*), flip-flops y multiplexores de entrada y salida.

Una LUT almacena una lista de salidas lógicas para cualquier combinación de entradas.

Estas FPGAs pueden ser reprogramadas para algún trabajo específico o para cambiar los requisitos de funcionalidad después de su fabricación. Algunas pueden ser programadas una sola vez mientras que otras pueden ser reprogramadas una y otra vez. A estos dispositivos que son programados una única vez son referidos como **OTP** (*one-time programmable*).

Field Programmable, se refiere al hecho de que su programación se hace “*en el campo*” a diferencia de otros dispositivos que su funcionalidad está programada por el fabricante [23].

Las tres principales tecnologías usadas para programar una FPGA son: **Antifusible**, **SRAM**, **Flash**

La tecnología **Antifusible** es una tecnología no reprogramable, por lo tanto son *OTP*. Son dispositivos que permiten establecer conexiones entre distintas capas de metal. No es volátil y además la conexión entre los bloques lógicos tiene un retardo muy reducido, por lo que el rendimiento es alto. Sin embargo, las FPGAs antifusibles necesitan un proceso de fabricación específico y por lo tanto es caro.

La tecnología **SRAM** es una tecnología reprogramable con una reprogramación muy rápida. Sin embargo, las FPGAs basadas en esta tecnología son volátiles, lo que significa que los datos de configuración del dispositivo se perderán cuando se desconecte la energía.

La tecnología **Flash** es reprogramable como la **SRAM** y además es no volátil como la **Antifusible**. Utiliza menos potencia que la **Antifusible** con un proceso de fabricación basado en transistores de puerta flotante.

Hay muchos tipos diferentes de circuitos integrados digitales, entre los que destacamos **PLDs** (*Programmable Logic Devices*), **ASICs** (*Application-Specific Integrated circuits*), **ASSPs** (*Application-Specific Standard Parts*) y **FPGAs**.

Los **PLDs** son dispositivos con una arquitectura interna predeterminada por el fabricante. También son configurables “*in situ*” con tecnologías basadas en fusibles, antifusibles y EPROM o EEPROM. En comparación a las **FPGAs**, contiene un número limitado de puertas lógicas y las funciones que se suelen implementar son más pequeñas y simples.

Por otro lado los **ASICs** y los **ASSPs** contienen cientos de millones de puertas lógicas y se usan para crear grandes y complejas funciones. Ambos están basados en los mismos procesos de diseño y tecnologías y pueden ser usados por millones de usuarios y compañías. La única diferencia es que un **ASIC** está diseñado y fabricado para una aplicación en concreto, mientras que un **ASSP** lo está para un dominio de aplicaciones.

Así, las **FPGAs** se encuentran entre los **PLDs** y los **ASICs** porque su funcionalidad puede ser diseñada en el campo como los **PLDs**, pero pueden contener millones de puertas lógicas y ser usadas para implementar funciones complejas que previamente sólo podían ser realizadas usando **ASICs**.

El coste de un diseño de **FPGA** es mucho menor que el de uno de un **ASIC**. Al mismo tiempo, los cambios de diseño implementados son más fáciles

en **FPGAs** y el tiempo de comercialización es más rápido [22].

Las FPGAs **SoC**(*System-on-chip*), que es un circuito integrado que tiene todos los componentes necesarios para un sistema electrónico. Además de la lógica genérica configurable incluyen empotrados en hardware uno o varios procesadores, memoria DDR, interfaces para buses estándar, interfaces de comunicaciones, e incluso unidades de procesamiento gráfico. Tienen una gran capacidad de procesamiento para adaptarse a diferentes aplicaciones. Una FPGA SoC de bajo costo y consumo se puede utilizar en aplicaciones como tarjetas de procesamiento de vídeo. Sin embargo, hay otros FPGA SoCs que se utilizan en aplicaciones de alto rendimiento en comunicaciones o computación de alto rendimiento.

A mediados del año 1980 llegaron las FPGAs, que eran usadas para implementar lógicas simples, máquinas de estados con una complejidad media y tareas de procesamiento de datos. A principios de los 90s, el mercado en el que se vendían se extendió al área de las telecomunicaciones debido a que el tamaño y sofisticación de las mismas empezaron a crecer. A finales de los 90s, el uso de las FPGAs en aplicaciones de consumo e industriales tuvo un enorme crecimiento.

Las FPGAs a menudo son utilizadas para crear prototipos de diseños ASIC o para tener un plataforma hardware donde verificar la implementación física de nuevos algoritmos [23].

Actualmente se pueden encontrar FPGAs de alto rendimiento con millones de puertas. Algunos de estos dispositivos tienen núcleos de microprocesador integrados, dispositivos de entrada-salida de alta velocidad y similares. El resultado es que actualmente las FPGAs pueden ser usadas para implementar sistemas en distintos ámbitos como por ejemplo:

- **Aeroespacial y defensa**
- **Emulación y Prototipado**
- **Audio**
- **Automoción**
- **Broadcast**
- **Electrónica de consumo**
- **Centro de procesamiento de datos**
- **Computación de alto rendimiento**
- **Industria**
- **Medicina**
- **Comunicaciones**
- **Inteligencia Artificial**
- **Procesamiento de imágenes**
- **Seguridad**

1.2. Niveles de síntesis automática

La metodología es un concepto abstracto referido a un conjunto de procesos que relacionan entre sí los niveles de complejidad y abstracción (*funcional o de comportamiento, arquitectural o transferencia de registros, lógico o de puertas y físico*), por los que pasa el diseño de un circuito [27].

Los niveles de abstracción son:

- **Funcional o de comportamiento:** Se indica el comportamiento del circuito como una relación funcional entre las entradas y salidas, sin tener en cuenta la implementación.
- **Arquitectural o de transferencia de registros:** Se realiza una partición de bloques funcionales y se planifican las operaciones que se van a realizar en cada ciclo de reloj.
- **Lógico o de puertas:** Componentes expresados como ecuaciones lógicas o puertas y elementos de una biblioteca genérica o específica de una tecnología.
- **Físico**

En general, establecer una metodología o flujo de diseño consiste en definir las distintas etapas por los que pasarán los distintos niveles de abstracción y fijar cómo pasar de unos niveles de abstracción a otros usando procesos manuales o automáticos de:

- **Síntesis:** Pasar descripciones de un nivel de abstracción a otro con mayor detalle (por ejemplo, de una descripción RT a puertas, o de puertas a física).
- **Análisis:** Extraer información de un descripción para verificar prestaciones o para validar restricciones en un nivel de abstracción superior.
- **Verificación / Simulación:** Validar las descripciones de cada etapa.

Para la realización del flujo de diseño, se puede seguir una evolución “*Top-Down*” empezando por la idea de la implementación o se puede seguir una evolución “*Bottom-Up*” comenzando por el nivel físico hasta llegar al funcional.

El diseño “*Top-Down*” consiste en partir de una idea con un alto nivel de abstracción y partiendo de ella, implementarla y si es necesario incrementar el nivel de detalle. Con este diseño se consigue una mayor adecuación a las especificaciones y requisitos.

El diseño “*Bottom-Up*” consiste en la descripción del circuito con componentes que se pueden agrupar en módulos hasta llegar al sistema completo que se desea. Este diseño está condicionado por los componentes del diseño que están disponibles.

Los lenguajes de descripción de hardware (*HDL*) son lenguajes de alto nivel, similares a los de programación (C, PASCAL, ADA, ...), con una sintaxis y semántica definidas para facilitar el modelado y descripción de circuitos

electrónicos, desde las celdas de base de un ASIC hasta sistemas completos, pudiéndose realizar estas descripciones a distintos niveles de abstracción, precisión y estilos de modelado. Los HDL modelan el comportamiento de un componente de cara a su simulación, aunque también se utilizan para describir el diseño de un circuito para su implementación a través de etapas de síntesis validadas vía simulación.

Los primeros lenguajes que fueron estándar IEEE son **VDHDL** y **Verilog**. Estos lenguajes se han ido desarrollando y consolidando durante bastante tiempo con el fin de dar soporte a las distintas etapas y niveles de abstracción del proceso de diseño. El lenguaje **VHDL** aparece con el fin de disponer una herramienta estándar e independiente para la especificación (modelado y/o descripción) y documentación de los sistemas electrónicos a lo largo de todo su ciclo de vida. Tiene una sintaxis más compleja y estricta, que reduce la posibilidad de errores. El lenguaje **Verilog** nace como un lenguaje de modelado ligado a un entorno de simulación, llegando a convertirse en un estándar “de facto” a nivel industrial. Tiene una sintaxis más simple, parecida a C y múltiples versiones [27].

Una de las características principales de un lenguaje de descripción hardware es que a partir de una descripción se puede generar un circuito físico. La síntesis es el paso de un nivel de descripción a uno de nivel inferior.

La síntesis física consiste en la ubicación, es decir, decidir dónde colocar todos elementos lógicos, y en el enrutamiento, en el que se decide cómo se interconectan los elementos en la FPGA.

La síntesis RT-lógica es un proceso en el se crea un diseño RTL (*Register Transfer Level*), que es una abstracción del diseño en el que se modela el circuito digital, y luego esa representación RTL es convertida a un conjunto de registros y ecuaciones booleanas equivalentes.

La principal diferencia entre síntesis RT y síntesis de alto nivel es que la primera parte de una descripción en la que de forma explícita se especifican las operaciones que deben realizarse en cada ciclo de reloj, mientras que la planificación de operaciones en ciclos de reloj se realiza de forma automática en la segunda.

La síntesis de alto nivel une hardware y software de manera que los diseñadores hardware pueden trabajar con un alto nivel de abstracción y los desarrolladores software pueden acelerar partes computacionalmente complejas de sus algoritmos en una FPGA.

El uso de una metodología de diseño de síntesis de alto nivel permite desarrollar algoritmos semejantes a los que se emplea en programación software, validar el correcto funcionamiento de un diseño de forma más rápida que con lenguajes de descripción hardware tradicionales y crear implementaciones hardware de alto rendimiento.

1.3. Plataformas de desarrollo

En este apartado se exponen los principales fabricantes de FPGAs, destacando a Xilinx, herramientas software de desarrollo de diversos fabricantes y

plataformas de propósito académico.

1.3.1. Principales fabricantes de FPGAs

Actualmente hay muchas empresas que fabrican FPGAs, pero las más importantes son *Xilinx*, *Intel*, *Microchip Technology* y *Lattice Semiconductor* [17].

Microchip Technology adquirió *Atmel* y *Microsemi*. Este fabricante proporciona dispositivos que abarcan el rango medio y alto, ofreciendo tres familias principales de FPGAs [20]:

1. **FPGA IGLOO2** - son FPGAS de gama media y baja, con tecnología flash dedicadas a funciones de propósito general.
2. **FPGA SoC SmartFusion2** - están basados en los dispositivos IGLOO2 incluyendo un procesador ARM-Cortex.
3. **FPGA SoC PolarFire** - son dispositivos de alto rendimiento, basados en tecnología flash, de costo optimizado, implementados en tecnología de procesos de 28nm.

Una de las características diferenciales de este fabricante son las FPGAs que son tolerantes a altas radiaciones y están dirigidos a entornos como el espacio. Destacan *RTG4* y *RT ProASIC3* basadas en tecnología flash, *RTAX* y *RTSX-SU* basadas en tecnología antifusible.

Lattice Semiconductor proporciona también dispositivos de rango bajo y medio y ofrece cuatro principales familias de FPGAs [18]:

1. **iCE** - son dispositivos basados en tecnología SRAM con memoria no volátil (NVM).
2. **CrossLink y CrossLinkPlus** - CrossLink está basado en tecnología SRAM mientras que CrossLinkPlus en tecnología flash, con una memoria no volátil (NVM). Están diseñadas preferentemente para aplicaciones de vídeo.
3. **MachXO** - estos dispositivos están basados en tecnología SRAM, con memoria flash. Diseñados especialmente para aplicaciones seguridad y control.
4. **ECP** - son dispositivos basados en SRAM y destinados a aplicaciones de uso general.

El principal competidor de *Xilinx* es *Intel*, que adquirió *Altera*, comercializa FPGAs basadas en tecnología SRAM de gama alta (*Agilex*, *Stratix*), gama media (*Arria*) y gama baja (*Cyclone*). Algunas de estas familias incluyen FPGAs SoCs que integran arquitecturas multicore basadas en procesadores ARM (Agilex y Stratix 10 con ARM Cortex-A53; Arria 10, Arria V y Cyclone V con ARM Cortex-A9).

Xilinx tiene bastante variedad de FPGAs en cuanto a coste y rendimiento. Actualmente, la serie *Virtex*, la serie *Zynq-7000 SoC*, la serie *Zynq UltraScale+ MPSoC* y la serie *Zynq UltraScale+ RFSoC* ocupan el mercado de gama alta, la serie *Kintex* y la serie *Zynq UltraScale+ MPSoC* de gama media y la serie *Artix* de gama baja junto con la *Spartan* que ha sido retirada del mercado.

La serie *Virtex* se basa en **CLBs**, donde cada uno de ellos equivale a varias puertas *ASIC*. Cada CLB se compone de sectores que son distintos para cada uno de los miembros de esta familia. Además de lógica configurable, se incluyen módulos “hard-core” para la memoria, bloques **DSP** (Procesador de señales digitales), controladores PCI Express, entre otros. En esta familia encontramos: *Virtex-II*, *Virtex-4*, *Virtex-5*, *Virtex-6*, *Virtex-7*, *Virtex-7 (3D)* y *Virtex UltraScale*.

La serie *Artix* se basa en la arquitectura unificada de la serie *Virtex*. Esta serie está diseñada para aplicaciones con rendimiento de bajo consumo.

La serie *Kintex* se caracteriza por consumir menos energía que la serie *Virtex*, incluyendo alto rendimiento.

1.3.2. Herramientas software de desarrollo

Dependiendo de la síntesis que queramos realizar podemos encontrar distintos software:

- **Herramientas de síntesis RT-lógica:**

- *Synplify Pro*, *Synplify Premier* (*Synopsys*) [26]
- *Precision RTL Plus*, *LeonardoSpectrum* (*Mentor Graphics*) [12]
- *Quartus* (*Altera*) [14]
- *Vivado* (*Xilinx*) [33]

- **Herramientas de síntesis de alto nivel:**

- *Vitis Unified Software Platform* (*Xilinx*) [32]
- *Synphony C Compiler* (*Synopsys*)
- *Impulse coDeveloper* (*Impulse C*)
- *Vivado High Level Synthesis* (*Xilinx*)
- *SDSoc* (*Xilinx*)
- *SDAccel* (*Xilinx*)

En cuanto a la síntesis de alto nivel, Altera apostó inicialmente por basarla en descripciones OpenCL multiplataforma, actualmente con la herramienta Intel FPGA SDK for OpenCL Software Technology [15]. Más recientemente, saca al mercado una herramienta basada en descripciones ANSI C++, denominada Intel HLS compiler [16].

Dentro de los fabricantes de FPGAs, Xilinx ha liderado el desarrollo de herramientas de síntesis de alto nivel basadas en descripciones C/C++ (*Vivado HLS* fué comercializado en 2013) y de entornos de desarrollo orientados

a la aceleración de aplicaciones C/C++/OpenCL buscando facilitar su uso a ingenieros software, sin necesidad de tener un perfil hardware más específico (SDSoC, SDAccel y actualmente Vitis).

La última herramienta comercializada por Xilinx, *Vitis* [32] es un entorno de desarrollo de aplicaciones que sustituye a las herramientas *SDSoc* y *SDAccel* que permite utilizar tanto FPGAs en tarjetas aceleradoras on premise y en la nube, como FPGAs con procesadores empotrados. Incorpora una herramienta de síntesis de alto nivel (**Vitis HLS**) que pretende reducir las diferencias entre escribir funciones para su ejecución software o para su implementación hardware. Y se dispone incluso de bibliotecas con funciones prediseñadas para diferentes dominios de aplicación (inteligencia artificial, visión, etc.).

1.3.3. Plataformas de propósito académico

Las plataformas de desarrollo con propósito académico que comercializa Xilinx son [36] [37]:

- **7-series** - *Basys3, Nexys4/DDR*
- **Zynq** - *ZedBoard, ZYBO, PYNQ-Z1, PYNQ-Z2*
- **Spartan** - *Atlys, Nexys3*
- **Virtex** - *Genesys, XUPV5*

Algunas de estas FPGAs están introducidas en las tarjetas vendidas por la empresa **Digilent**, que diseña, fabrica y distribuye herramientas de diseño electrónico, haciendo más accesible esta tecnología a estudiantes. Se asocia principalmente con Xilinx para crear plataformas usadas para el aprendizaje. Entre todas las tarjetas que ofrece esta empresa encontramos las tarjetas *Basys 3*, que es una placa de desarrollo exclusiva para Vivado y sobre todo para estudiantes y principiantes en la tecnología FPGA [6], y *Arty A7*, especial para gente que ha estado más en contacto con esta tecnología [3], con arquitectura FPGA Xilinx Artix-7, *PYNQ-Z1*, está diseñada para usarse con PYNQ, un nuevo marco de código abierto que permite a los programadores explotar las capacidades de Xilinx Zynq APSoC sin tener que diseñar circuitos lógicos programables. En cambio, el APSoC se programa usando Python y el código se desarrolla y prueba directamente en el PYNQ-Z1 [8], *Zybo Z7* (Ver apartado 1.1.2) y *ZedBoard*, placa de desarrollo de bajo costo con el que se puede crear un diseño basado en Linux, Android, Windows u otro sistema operativo [10], con arquitectura Zynq-7000 SoC, *Basys 2*, es una plataforma de diseño e implementación de circuitos para que los estudiantes estudien la construcción de circuitos digitales reales [5], con una FPGA Spartan-3E, *Arty S7*, es el miembro más reciente de la familia Arty para fabricantes y aficionados y ofrece mejor tamaño y rendimiento [4], con una Spartan-7, *Anvyl*, es una plataforma de desarrollo de circuitos digitales completa y lista como plataforma de entrenamiento [2], con una Spartan-6 y *Genesys 2*, es una plataforma de desarrollo de circuitos digitales avanzada, de alto rendimiento [7], con una Kintex-7.

1.4. Estructura de la memoria

Al comienzo de este proyecto se realiza un resumen donde se describe el contenido de este proyecto de una forma resumida además de mostrar las palabras clave. Y a continuación se encuentra el mismo resumen y palabras clave, pero en inglés.

Una vez realizada la motivación e introducción en el Capítulo 1 de la memoria que contiene el concepto del que partimos, el concepto de FPGA, explicando su arquitectura y haciendo comparaciones con otros circuitos integrados digitales y se mencionan los campos de aplicación más importantes. Además, en este capítulo se detallan los distintos niveles de síntesis automática (síntesis funcional, RT-lógica y física). Por último se hace mención a varios fabricantes de FPGAs con sus respectivos dispositivos, a las herramientas software de desarrollo dando más importancia a las de Xilinx y algunas plataformas de propósito académico del fabricante Xilinx.

En el capítulo 2 se muestran los objetivos del trabajo planteados conseguir durante la realización de este TFG.

Mejor haz referencia al apartado de Materiales (en general) y muy importante al de Metodología. Date cuenta de que la mayor parte del tiempo la has invertido en aprenderla.

Como resolución del trabajo se encuentra el capítulo 3 que está dividido en 4 subapartados, el de Materiales en el que se describe las características de la Familia Zynq-7000, así como las de la tarjeta usada (ZYBO) y la descripción de la plataforma utilizada para poder realizar proyectos. También está el subapartado de la Metodología, donde se explican las distintas etapas del flujo de diseño y en los otros dos subapartados se explican los módulos hardware que formarán parte de la plataforma de prácticas, un procesador, una memoria RAM, un módulo de generación de reloj y un controlador VGA, y se realizan un par de casos prácticos haciendo uso de los módulos explicados en las secciones anteriores.

El capítulo 4 contiene las conclusiones y vías futuras del proyecto y del aprendizaje realizado.

A continuación se presenta un anexo que describe de forma práctica el flujo de diseño de sistemas mediante síntesis RT como toma de contacto de la herramienta Vivado y la tarjeta Zybo.

Por último está la bibliografía que contiene la información relevante de los libros, artículos y páginas visitadas para la realización del proyecto.

Capítulo 2

Objetivos del trabajo

El principal objetivo de este TFG es el conocimiento de una plataforma de carácter didáctico orientada a FPGAs de la familia Zynq de Xilinx para la realización de diferentes ejercicios prácticos útiles en el aprendizaje en asignaturas relacionadas con el diseño de sistemas basados en dispositivos hardware reconfigurables.

La empresa Xilinx es una empresa líder en el desarrollo de herramientas no sólo a nivel de síntesis RT-lógica, sino también de síntesis de alto nivel y particularmente para co-diseño HW/SW de sistemas empotrados en FPGAs basado en aplicaciones y descripciones tipo C/C++. Sus herramientas se han utilizado en asignaturas de perfil más avanzado y resulta conveniente la migración a este tipo de plataformas de las prácticas que se realizan en otras asignaturas de perfil más básico, con intención de favorecer al estudio con una misma plataforma basada en la herramienta Vivado y una tarjeta de desarrollo basada en FPGAs Zynq.

Se toma como referencia la asignatura "Desarrollo de hardware digital" de la especialidad de Ingeniería de Computadores que pertenece al grado de Informática, cuyos contenidos se corresponden fundamentalmente con el aprendizaje de la metodología de diseño de sistemas basados en FPGAs con herramientas de síntesis automática y verificación a partir de descripciones VHDL en el nivel RT, para el análisis y diseño de módulos hardware específicos, tales como procesadores específicos, memorias, y módulos de interfaz y comunicaciones.

Y por último se establecen siguientes objetivos:

1. Estudio y descripción de cómo se realiza en Vivado la metodología propia de flujo de diseño con FPGAs a partir de descripciones RT en VHDL.
2. Descripción módulos de especial interés en la plataforma para la realización de prácticas (procesador, memoria, interfaz VGA, generación de reloj).
3. Realización de dos casos prácticos planteados en la asignatura para comprobar la utilidad y el correcto funcionamiento de los módulos que en este momento integran la plataforma junto con la tarjeta Zybo y la herramienta Vivado.

Capítulo 3

Resolución del trabajo

Este capítulo está dividido en cuatro subapartados, en los que se describen las características de la Familia Zynq-7000, así como su arquitectura, la tarjeta usada (ZYBO) junto con sus características. Además se realiza la descripción de la plataforma Vivado y se explican las distintas etapas del flujo de diseño y los módulos hardware que se van a usar, un procesador, una memoria RAM, un módulo de generación de reloj y un controlador VGA. Por último se realizan un par de casos prácticos haciendo uso de los módulos explicados en las secciones anteriores.

3.1. Materiales

En esta sección se describen los principales materiales usados en este proyecto dividiéndola en 3 apartados, la familia Zynq-7000 junto a sus características, la tarjeta Zybo y sus especificaciones y la herramienta Vivado.

3.1.1. Familia Zynq-7000

La familia **Zynq-7000** se basa en la arquitectura SoC. Estos productos integran un sistema de procesamiento basado en *ARM Cortex-A9* de un sólo núcleo o dual-core con muchas funciones y lógica programable de 28nm. También incluye memoria “on-chip”, interfaces de memoria externa e interfaces de conectividad periféricas [38].

Esta familia ofrece la flexibilidad y escalabilidad de un FPGA, al mismo tiempo que proporciona rendimiento, potencia y facilidad de uso típicamente asociados con ASIC y ASSP. Su gama de dispositivos permite crear aplicaciones de alto rendimiento. Si bien cada dispositivo de la familia Zynq-7000 contiene el mismo sistema de procesamiento, los recursos de lógica programable y E/S varían entre los dispositivos.

La arquitectura Zynq-7000 (Ver Figura 3.1) permite la implementación de lógica personalizada para configurar módulos hardware específicos en el PL (“Programmable Logic”) y software personalizado en el PS (“Processing Sys-

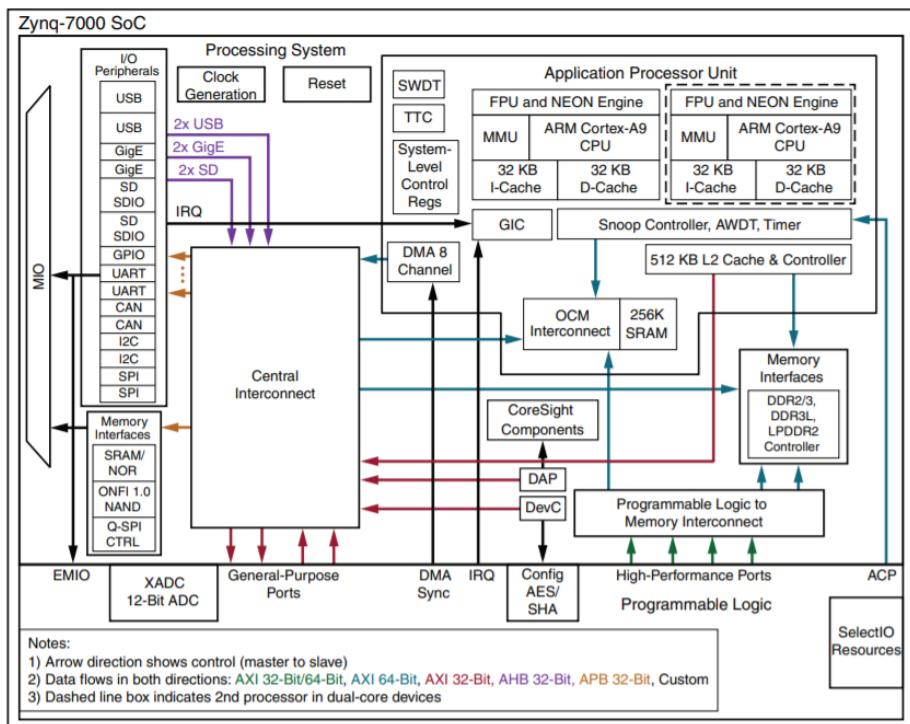


Figura 3.1: Arquitectura Zynq-7000 [38]

tem”). La integración del PS con el PL permite niveles de rendimiento que las soluciones de dos chips (por ejemplo, un ASSP con un FPGA) no pueden igualar debido a su ancho de banda de E/S, latencia y presupuestos de energía limitados.

Xilinx ofrece una gran cantidad de módulos IP (“Intellectual Property”) para la familia Zynq-7000. Los controladores de dispositivos “Stand-alone” y Linux están disponibles para los periféricos en el PS y el PL. El entorno de desarrollo de Vivado Design Suite permite un rápido desarrollo de productos para ingenieros de software, hardware y sistemas. La adopción del PS basado en ARM conlleva una amplia gama de herramientas de terceros y proveedores de módulos IP.

La inclusión de un procesador de aplicaciones permite el soporte del sistema operativo de alto nivel, por ejemplo, Linux. El PS y el PL están en dominios de energía separados, lo que permite apagar el PL para administrar la energía. Los procesadores de la PS siempre se inician primero, permitiendo un enfoque centrado en el software para la configuración de PL. La configuración de PL se gestiona mediante software que se ejecuta en la CPU.

La PS está dividida en cuatro bloques principales:

- **APU (Unidad de Procesamiento de Aplicaciones)** - Incluye un procesador dual-core o single-core ARM Cortex-A9 MPCores, canales DMA, interruptores y relojes, entre otros.

- **Interfaces de Memoria** - Incluye módulos de controladores de memoria dinámica (DDR3) e interfaces de memoria estática (por ejemplo, interfaz flash NAND).
- **IOP (Periféricos E/S)** - Esta unidad contiene los periféricos de comunicación de datos (USB, Ethernet MAC, por ejemplo).
- **Interconexiones** - Los tres anteriores están todos conectados entre sí y al PL a través de una interconexión ARM AMBA AXI de varias capas.

El PL incluye:

- **CLB** - Incluye LUT que se puede configurar como una LUT de 6 entradas con una salida, o como dos de 5 entradas con salidas independientes. Cada salida LUT se puede registrar en un flip-flop. Estos CLBs están formados por dos segmentos, que están constituidos por cuatro de estos LUT y 8 flip-flops, así como los multiplexores y la lógica aritmética.
- **Bloque RAM** - Incluye “dual-port”, lógica FIFO programable y circuito integrado opcional de corrección de errores.
- **Bloques DSP (“Digital Signal Processing”)** - Cada segmento de DSP consiste en un multiplicador de complemento a dos y un acumulador de 48 bits. Incluye un pre-sumador adicional. Este pre-sumador mejora el rendimiento. El DSP también incluye un detector de patrones que se puede utilizar para redondeo.
- **Bloques E/S Programables**
- **Módulo de configuración PL**

La arquitectura Zynq AP SoC (XC7Z010-1CLG400C) incluida en la tarjeta Zybo está dividida también en dos partes (Figura 3.1), el (*PS*) y la (*PL*). La PL usada es parecida a la de la FPGA *Xilinx 7-series Artix*, excepto porque contiene buses y puertos dedicados que hacen que esté acoplado fuertemente al PS. Contiene 28K de celdas lógicas programables, 17.600 LUTs, 35.200 Flip-Flops, 2.1Mb de bloque RAM y 80 bloques DSP. Además, cada dispositivo de la familia Zynq-7000 tiene hasta 8 “clock management tiles” (CMTs) que son un conjunto de MMCM y PLL. En concreto, esta arquitectura tiene 2 MMCM y 2 PLL.

El PS consiste en un conjunto de componentes como la *APU* (Unidad de Procesamiento de Aplicaciones) que incluye dos procesadores ARM Cortex-A9 MPCore, *AMBA* (Arquitectura de Bus de Microcontrolador Avanzada), Controlador de Memoria *DDR3*, y varios controladores periféricos con las entradas y salidas multiplexadas a 54 pines dedicados (*MIO*). Los controladores periféricos están conectados al procesador mediante la interconexión *AMBA* y la PL está conectada de la misma manera.

3.1.2. Tarjeta Zybo

La FPGA de la familia Zynq 7000 descrita en la sección anterior es incluida en tarjeta **ZYBO** (*ZYbo BOard*) [11]. Es una plataforma de desarrollo de circuito digital, y está construida alrededor del miembro más pequeño de la familia Zynq-7000, el **Z-7010**, que se basa en la arquitectura **AP SoC** (*Xilinx All Programmable System-on-Chip*), que integra un procesador de doble núcleo ARM Cortex-A9 con lógica *Xilinx 7-series FPGA*.

La tarjeta Zybo ofrece las siguientes características (Figura 3.2):

- ZYNQ XC7Z010-1CLG400C
- Puerto HDMI de doble función (fuente/receptor)
- Puerto VGA de 16 bits por pixel
- Ethernet PHY trimodo (1Gbit/100Mbit/10Mbit)
- MicroSD
- OTG USB 2.0 PHY
- EEPROM externo
- Códec de audio con salida de auricular y micrófono
- 128Mb Serial Flash con interfaz QSPI
- Programación JTAG “on-board” y convertidor UART a USB
- GPIO: 6 botones, 4 interruptores, 5 LEDs
- 6 conectores Pmod (1 procesador dedicado, 1 dual analógico / digital, 3 diferenciales de alta velocidad, 1 lógico dedicado)
- Procesador dual-core Cortex-A9 de 650Mhz
- Controlador de memoria DDR3 con 8 canales DMA
- Controladores periféricos de alto ancho de banda: 1G Ethernet, USB 2.0, SDIO
- Controladores periféricos de bajo ancho de banda: SPI, UART, CAN, I^2C
- Lógica Reprogramable equivalente a Artix-7 FPGA

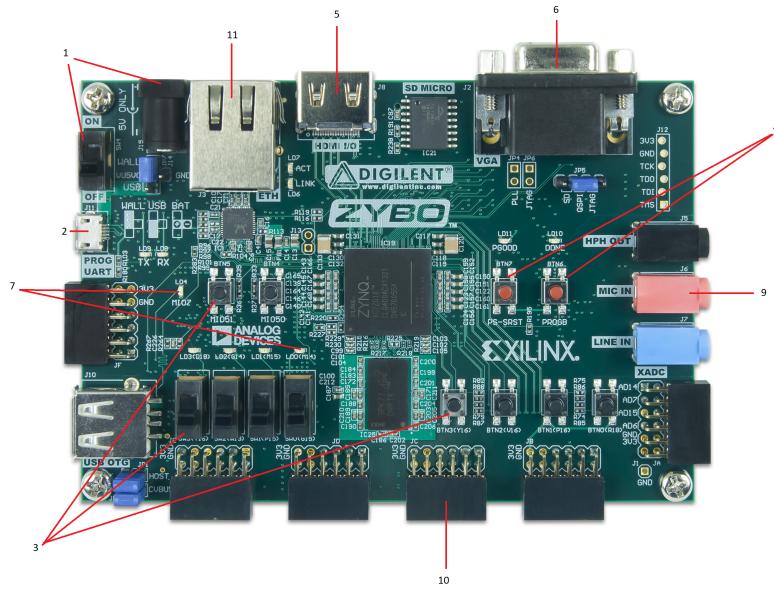


Figura 3.2: *ZYBO Zynq-7000 Development Board [11]*

La tarjeta Zybo incluye cuatro interruptores, cuatro botones y cuatro LEDs individuales a la PL. Además hay 2 botones y un LED conectados directamente al PS a través de pines MIO. Adicionalmente hay un LED de encendido de la tarjeta, otros dos para el estado del puerto USB y un último LED para el estado de la programación de la FPGA.

Uno de los **botones de reset** reestablece la PL, que permanecerá sin configurar hasta ser reprogramado de nuevo. El otro reinicia el dispositivo sin afectar al entorno de depuración.

Nos encontramos con tres **Conectores de audio**, dos entradas para micrófono y línea estéreo y una salida para auriculares.

El **Puerto VGA** (*Video Graphics Array*) es una interfaz que envía una señal desde la FPGA al conector VGA. Éste se suele usar para conectar dispositivos. La placa ZYBO utiliza 18 pines lógicos programables para crear un puerto de salida VGA analógico. Esto se traduce en una profundidad de color de 16 bits (5 bits para el rojo, 6 para el verde y 5 para el azul) y dos señales de sincronización estándar. La conversión de digital a analógico se realiza utilizando una escalera de resistencia R-2R simple. La escalera trabaja en conjunto con la resistencia de terminación de 75 ohmios de la pantalla VGA para crear señales VGA rojas, azules y verdes de 32 y 64 niveles de señal analógica.

En la sección 5.3 se realizará una descripción de este componente para más tarde usarlo en un caso práctico de la sección 5.4.

El **Puerto HDMI** es un conector de entrada y salida con el que se puede transmitir vídeo compatible con HDMI o DVI.

Los **Conectores Pmod** (*Módulos Periféricos*) son unos conectores con

estándares de módulos periféricos, para ampliar la capacidad de la lógica programable. Se comunican mediante 6,8 ó 12 pines para transportar señales de control digital. En nuestro caso, son 12 pines (2x6). Hay 6 conectores Pmod con distinto comportamiento en esta tarjeta y cada uno pertenece a una de las cuatro categorías, “standard”, “MIO connected”, “XADC” y “high-speed”. Por ejemplo, la conexión PS2 para conectar teclados y ratones, podría ser un buen ejemplo para realizar algún ejercicio práctico con propósito académico.

3.1.3. Vivado

El software de Xilinx **ISE** no estaba preparado para soportar la complejidad y capacidad de un diseño de una FPGA con un procesador ARM. *Vivado Design Suite* (Figura 3.3) fue desarrollado para FPGAs con más capacidad y permite compilaciones de descripciones basadas en *C* gracias a la funcionalidad de síntesis de alto nivel.

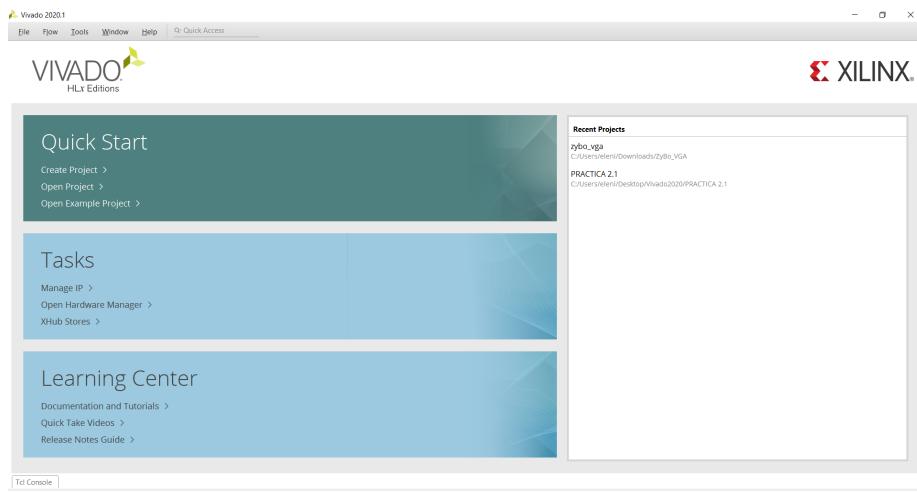
Zybo es compatible con *Vivado Design Suite* así como con el conjunto de herramientas ISE/EDK. Estas herramientas combinan el diseño lógico FPGA con el desarrollo software de ARM. Se pueden utilizar para diseñar sistemas de cualquier complejidad, desde un sistema operativo completo hasta un programa simple que controla algunos LEDs.

Vivado Design Suite es un entorno de diseño integrado (**IDE**) de Xilinx para la síntesis y análisis de diseños HDL. Vivado incluye su propio simulador lógico y además está la posibilidad de usar otros simuladores como *ModelSim*, *Mentor Questa*, *Cadence IES* o *Synopsys VCS*. Además incluye síntesis a alto nivel con una herramienta que convierte código C a lógica programable.

Está formado por 4 componentes:

- **Vivado High-Level Synthesis** - Permite usar programas en *C*, *C++* y *SystemC* en dispositivos Xilinx sin necesidad de crear un RTL manualmente. Aumenta la productividad del desarrollador y admite clases, plantillas, funciones y sobrecarga de operadores.
- **Vivado Simulator** - Es un simulador controlado por eventos de lenguaje de descripción hardware (**HLD**) que admite simulación de comportamiento y tiempos. Además admite scripts *TCL* (*Tool Command Language*) en lenguaje mixto, es decir, admite lenguajes como *Verilog*, *SystemVerilog* y *VHDL*.
- **Vivado IP Integrator** - Permite integrar y configurar IP (“*Intellectual Property*”) desde la biblioteca propia de Xilinx.
- **Vivado TCL Store** - Es un sistema de comandos para desarrollar complementos para Vivado además de agregar y modificar las capacidades de Vivado. Todas las funciones de Vivado se pueden controlar con los scripts *TCL*.

En concreto, la versión que se ha utilizado es con Vivado 2020.1. Para trabajar con él, se puede hacer tanto trabajando con la *TCL* o directamente con la *GUI* de Vivado IDE [35].

Figura 3.3: *Vivado IDE*

La sección **Quick Start** nos proporciona fácil acceso a la creación de un nuevo proyecto, abrir proyectos existentes o abrir proyectos de ejemplo ofrecidos por Xilinx. Además, en la sección **Recent Projects** se pueden abrir proyectos usados recientemente.

En la sección **Tasks** encontramos el acceso a **Manage IP** que nos permite ver el catálogo de módulos IP, personalizar módulos IP y generar productos de salida. **Open Hardware Manager** nos permite conectar la tarjeta y descargar un programa en el dispositivo FPGA. **XHub Store** sirva para instalar placas, aplicaciones Tcl (de código abierto) y diseños de ejemplo desarrollados por Xilinx y otros proveedores.

La última sección es **Learning Centre** donde se encuentra el acceso directo a la documentación, tutoriales y videos sobre lo que se puede hacer con esta herramienta.

Los componentes principales del entorno principal de Vivado (ver Figura 3.4 son:

1. *Menu Bar*
2. *Main Toolbar*
3. *Flow Navigator*
4. *Layout Selector*
5. *Data Windows Area*
6. *Workspace*
7. *Menu Command Search Field*
8. *Project Status Bar*
9. *Results Windows Area*

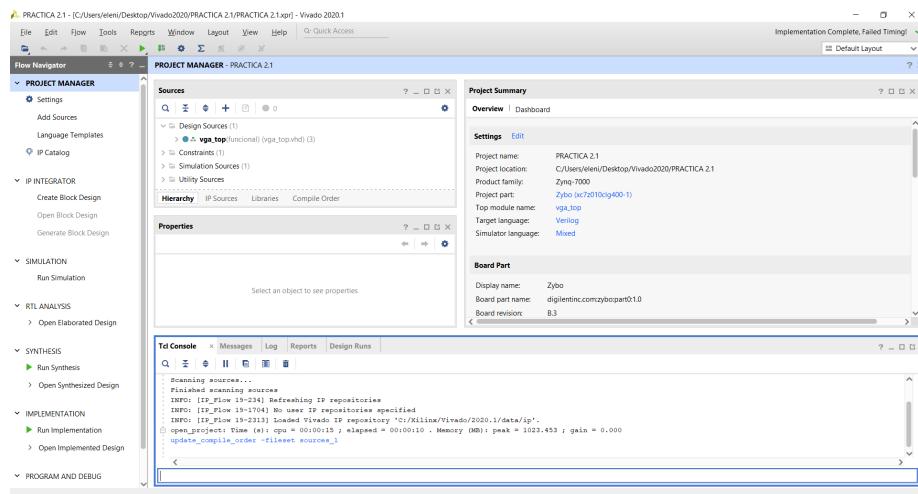
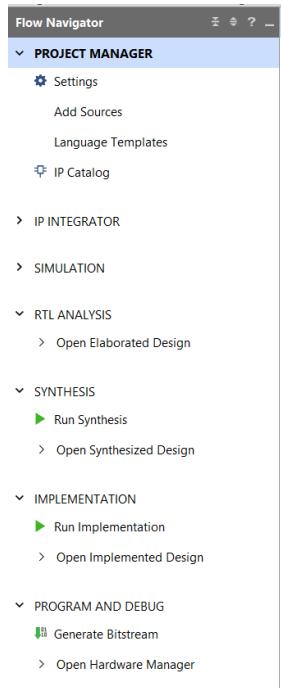


Figura 3.4: Entorno Principal Vivado IDE

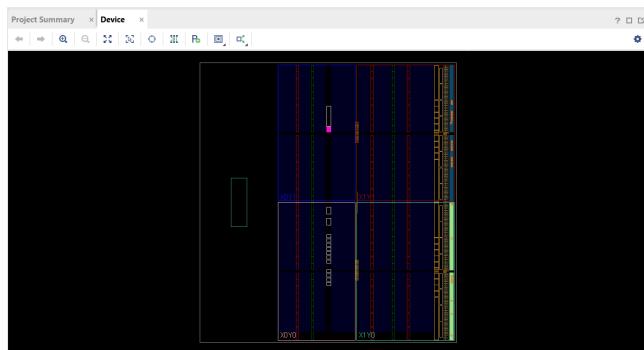
Flow Navigator (Figura 3.5) permite acceder a comandos y herramientas que van desde abrir diseños a crear un archivo bitstream. Las diferentes secciones permiten hacer lo siguiente:

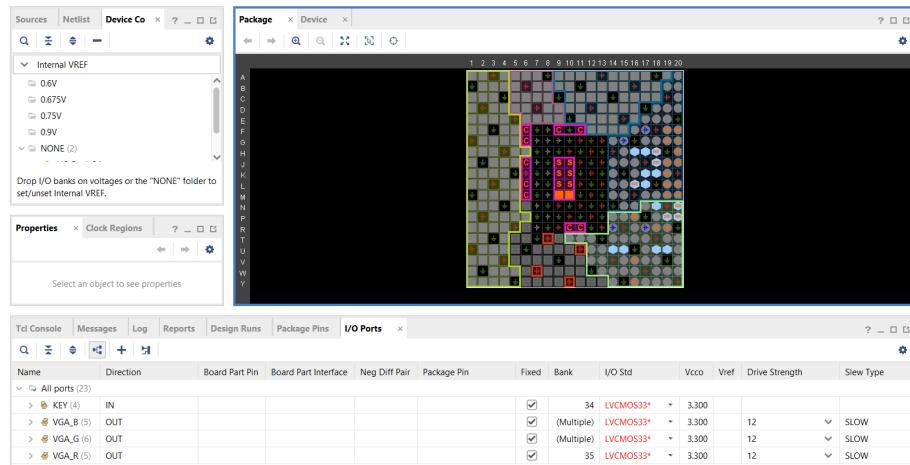
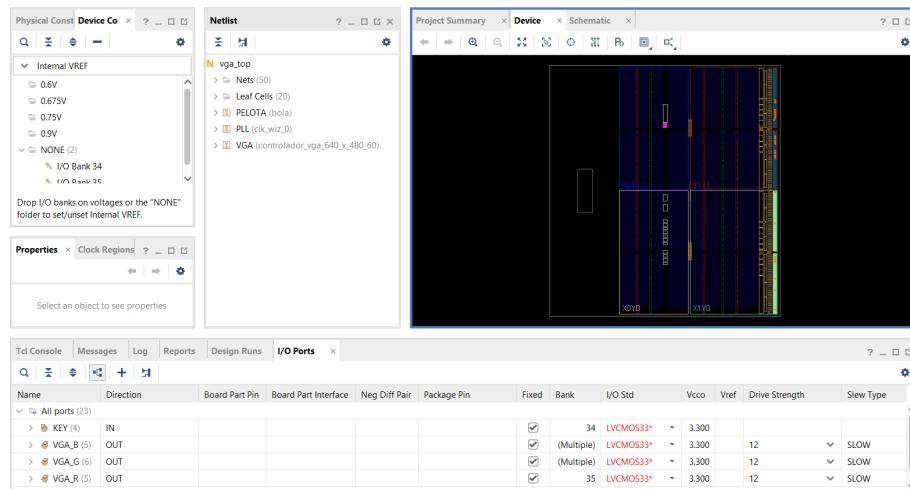
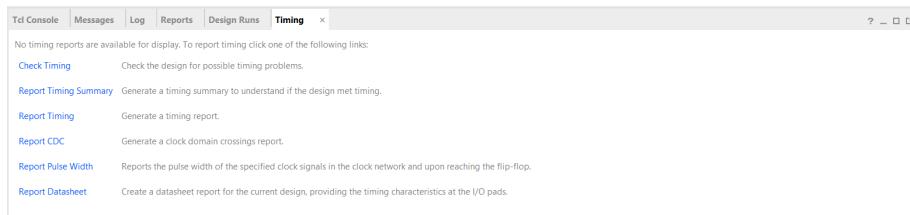
- **Project Manager:** Cambio de ajustes generales, añadir o crear archivos o abrir el Catálogo de IPs
- **IP Integrator:** Crear, abrir o generar un bloque de diseño.
- **Simulation:** Ejecutar una simulación
- **RTL Analysis:** Abrir un diseño elaborado o generar un diseño de diagrama de circuitos RTL.
- **Synthesis:** Ejecutar la síntesis o abrir el diseño sintetizado, donde se encuentran informes sobre el mismo que realiza Vivado.
- **Implementation:** Ejecutar la implementación, implementar un diseño activo o abrir el diseño implementado, donde se encuentran informes sobre el mismo que realiza Vivado.
- **Program and Debug:** Generar un archivo bitstream o abrir una ventana para conectar la tarjeta FPGA y programarla.

Figura 3.5: *Flow Navigator*

Layout Selector proporciona el diseño de ventanas predefinidas para facilitar el proceso de diseño. Entre las opciones tenemos:

- **Default Layout:** Muestra el diseño con el mínimo número de ventanas, con un resumen global del diseño (Figura 3.6).
- **I/O Planning:** Definición de restricciones de ubicación I/O y colocación de puertos (Figura 3.7).
- **Floorplanning:** Gestionar particiones y tareas jerárquicas (Figura 3.8).
- **Timing Analysis:** Ejecutar informes de tiempo y analizarlo (Figura 3.9).

Figura 3.6: *Default Layout*

Figura 3.7: *I/O Planning*Figura 3.8: *Floorplanning*Figura 3.9: *Timing Analysis*

Project Status Bar da información sobre el estado actual del diseño activo. *Data Windows Area* muestra información sobre los archivos que forman el

diseño. *Workspace* muestra ventanas como el editor de textos o el diseño del diagrama de circuitos, entre otros. Y *Results Windows Area* presenta los resultados de los comandos ejecutados. Además se muestran distintas ventanas, como *Tcl Console*, *Messages*, *Log*, *Reports* y *Design Runs*.

3.2. Metodología

Para el desarrollo de este trabajo se ha seguido una planificación y secuencia temporal concreta. Primero se ha revisado el estado actual del mercado de FPGAs, centrándonos en los dispositivos y herramientas software del fabricante Xilinx (apartado 1.3). Después, se han estudiado las características de la tarjeta y se han realizado algunas pruebas de configuración como encender y apagar los leds de la tarjeta. A continuación, se ha estudiado el flujo de diseño con Vivado para síntesis RT-lógica a partir de descripciones VHDL, en concreto, a partir de la documentación aportada por el fabricante (Xilinx), se ha estudiado detalladamente las diferentes funcionalidades y se han reproducido experimentalmente con un ejemplo sencillo. Con esto, hemos seguido el flujo de diseño para generar y validar módulos IP que han sido útiles para la realización de prácticas en asignaturas relacionadas con este tema. Por último se han utilizado estos módulos en dos casos prácticos para validarlos experimentalmente.

Dentro del flujo de diseño, lo que más ha llevado tiempo ha sido el tema de la simulación y los dos tipos de simulación funcional y temporal. Además me costó crear bien los testbenches para poder realizar las simulaciones. La principal dificultad a la hora de realizar el trabajo ha sido tener que entender la metodología y el flujo de diseño a partir de descripciones RT.

En esta sección se trata de estudiar y describir cómo se realiza con Vivado la metodología propia de flujo de diseño con FPGAs a partir de descripciones RT en VHDL.

Las etapas del flujo de diseño en FPGAs se explican en los siguientes apartados y son [24]:

1. **Diseño**
2. **Síntesis**
3. **Simulación**
4. **Implementación**
5. **Generación Bitstream**

Este flujo de diseño que ahora se va describir, se ha reproducido en base a un ejemplo sencillo (contador de 4 bits) descrito en el Anexo de este proyecto.

3.2.1. Diseño

Vivado soporta múltiples lenguajes de descripción hardware, entre ellos VHDL y Verilog, e incluso se pueden usar ficheros con ambos lenguajes. El siguiente dia-

grama (Ver Figura 3.10) muestra un flujo de diseño típico con las herramientas de Xilinx.

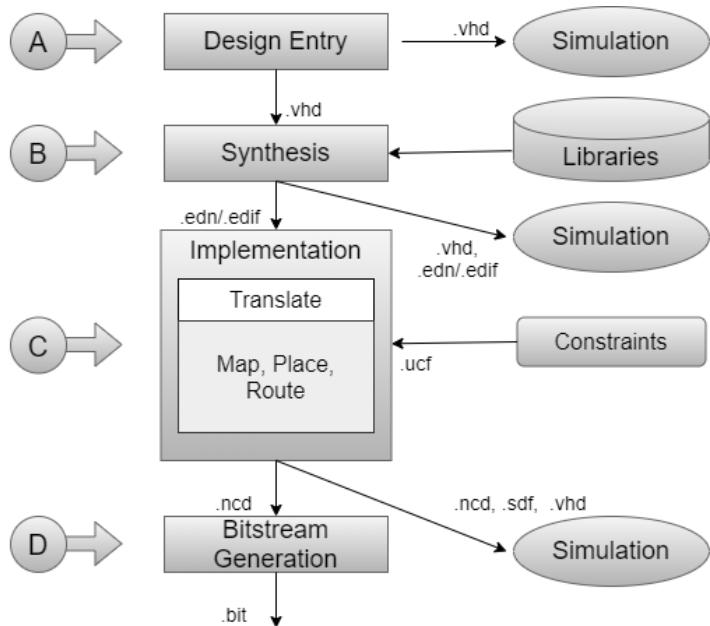


Figura 3.10: *Typical Xilinx FPGA Development Flow (Adaptada de [24])*

Para comenzar con todo este proceso, primero tenemos que crear un proyecto. En Vivado los pasos a seguir son *Create New Project* → *Project Name* → *Project Type* → *Default Part*. En la parte *Project Type* se pueden agregar ficheros en VHDL y en la parte *Default Part* se establece la tarjeta que se va a configurar, en nuestro caso, la tarjeta ZYBO.

Además de agregar ficheros, Xilinx nos da la opción de crear el diseño sin tener que escribir o agregar ficheros con un generador automático (*Core Generator*), del que se hablará más adelante.

Cuando se crea el proyecto, podemos agregar nuevos ficheros con la opción *Flow Navigator* → *Project Manager* → *Add Sources*.

En caso de querer cambiar la tarjeta que hemos elegido anteriormente sólo tenemos que cambiarlo en la sección *Project Manager* en el apartado *Project Settings*, además del lenguaje usado, la librería y el nombre del módulo principal.

3.2.2. Simulación

Cuando ya tenemos el diseño, hay que comprobar el correcto funcionamiento del circuito. Para ello, se usa la simulación, que puede realizarse después del diseño, de la síntesis o de la implementación. Hay dos tipos de simulación, la funcional y la temporal. La simulación funcional se realiza después del diseño, para comprobar que se ejecuta correctamente. Y la simulación temporal puede

realizarse después de la síntesis lógica sobre la netlist con elementos de biblioteca de las celdas propias de la FPGA elegida para el proyecto, y/o después de la implementación una vez que ya se pueden estimar los retardos asociados tanto a bloques lógicos como a interconexiones. Así, por ejemplo, en la simulación se tienen en cuenta los retardos de propagación de los componentes y de los caminos (buses) por los que se propagan las señales digitales. La simulación temporal requiere que se haga un proceso previo de *Place and Route* (Ver apartado 5.2.4), con el fin de que el simulador conozca los parámetros temporales de los componentes que debe utilizar en la simulación del sistema.

Hay otras herramientas útiles en la verificación como el analizador temporal (*Flow Navigator → Project Manager → Synthesis/Implementation → Report Timing Summary*) (Figura 3.11) para poder realizar la simulación temporal adecuadamente (por ejemplo, hay que conocer la frecuencia máxima para describir la forma de onda de la señal de reloj) o el módulo de visualización de la netlist RT (*Flow Navigator → Project Manager → RTL Analysis → Schematic*) para identificar antes de simular funcionalmente los elementos de memoria que dejan ver la arquitectura descrita.

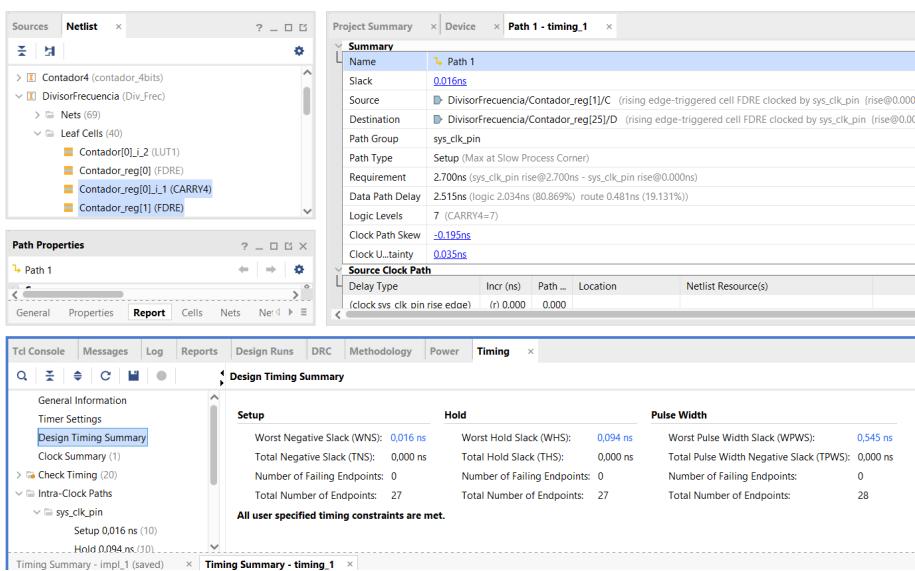


Figura 3.11: Analizador Temporal Implementación

La simulación requiere de un diseño, ya sea un código o una netlist o una implementación completa y la definición de estímulos correspondientes a las señales de entrada del circuito como un banco de pruebas (“*Testbench*”). Un banco de pruebas es uno o más módulos que conectan el diseño, la Unidad Bajo Prueba(*UUT*), con estímulos generados desde un archivo para controlar las de la *UUT* y poder estudiar sus salidas.

Vivado integra su propia herramienta de simulación, aunque existe la posibilidad de usar otras herramientas como *ModelSim*, *Questa Advance*, *Incisive Enterprise Simulator* o *Verilog Compiler Simulator*.

Estas herramientas además de hacer uso de un banco de pruebas nos da la posibilidad de manejar las señales de entrada que queremos aplicar al diseño y se muestra de manera gráfica las salidas proporcionadas por el circuito además de unas listas con los componentes de los que se quiera conocer el estado.

Hay cuatro ubicaciones principales donde ejecutar la simulación (Ver Figura 3.10):

- A. **“Behavioral Simulation”** - Se realiza antes de la síntesis.
- B. **“Netlist Simulation”** - Se realiza después de la síntesis.
- C. **“Post-Map simulation”**- Simulación posterior al mapeado tecnológico.
- D. **“Post-Implementation”** - Simulación después de la implementación.

A. Esta simulación se conoce como simulación de comportamiento o funcional. Como sólo se dispone del código y no se sabe cómo se va a implementar el diseño, esta simulación se usa para comprobar la función de los módulos realizados. Esta es la simulación que se va a usar durante la realización de los casos prácticos posteriores. Para ejecutar esta simulación Vivado nos ofrece la opción *Flow Navigator → Simulation → Run Simulation → Run Behavioral Simulation*.

B. Esta simulación ejecuta la versión del diseño con la lista de conexiones que contiene información sobre los recursos pero no la ubicación final, por lo que no hay información de enrutamiento. La simulación post-síntesis variará en el tiempo de la pre-síntesis, pero el comportamiento debe ser el mismo. Por lo tanto, esta simulación nos sirve para verificar la generación de las conexiones. En Vivado podemos acceder con *Flow Navigator → Simulation → Run Simulation → Run Post-Synthesis Simulation*.

C. La simulación posterior al mapeado tecnológico no se suele ejecutar y por lo tanto no tenemos opción en Vivado. Pero en ella se conocen conocimientos sobre estrategia de implementación y a veces la ubicación física para generar una simulación de tiempo más detallada.

D. Esta simulación es la más precisa ya que se conocen todos los retardos. Si el diseño simula correctamente, pasa el análisis de tiempo y el banco de pruebas es válido, entonces la FPGA debería funcionar correctamente. Es la simulación que requiere más tiempo y recursos tanto en tiempo de procesador como en la redacción de un banco de pruebas de calidad. En algunas fases de determinados proyectos esta simulación no se realiza, sino que se realizan pruebas directamente configurando la FPGA de la tarjeta. Por ejemplo, en el caso del módulo de sincronización de la VGA explicado en el apartado 5.3.4 (sería mucho más difícil verificar la correcta funcionalidad del módulo observando las formas de onda de las señales de sincronización horizontal y vertical que probando si se visualiza correctamente la imagen en el monitor). Para usar esta simulación Vivado tiene la siguiente opción *Flow Navigator → Simulation → Run Simulation → Run Post-Implementation Simulation*.

3.2.3. Síntesis

El proceso de síntesis consiste en convertir los archivos VHDL en una netlist, que es una lista de elementos lógicos y otra de conexiones que describe cómo se conectan los elementos. Una netlist puede ser independiente de la plataforma.

Todos los errores generados por las herramientas de síntesis se deben resolver antes de pasar al siguiente paso, y además generan bastante advertencias que no pueden ignorarse porque pueden ocasionar errores más adelante.

Muchos entornos de desarrollo proporcionan herramientas adicionales, como visores de netlist. Vivado lo integra y tiene un apartado especial, *Synthesis*, dentro de *Flow Navigator*. Además de configurar los ajustes de síntesis y hacer ejecutarla, encontramos una serie de opciones que se pueden usar después de haberla ejecutado. Entre ellas encontramos *Schematic* que nos muestra el esquemático de nuestro diseño, *Constraints Wizard* que es un asistente para crear restricciones de tiempo con la metodología de diseño de Xilinx además de analizar las restricciones de tiempo que puedan faltar en el diseño y hacer recomendaciones. *Edit Timing Constraints* nos ayuda a modificar las restricciones creadas anteriormente. Y por último hay una serie de informes que nos indican el estado del diseño, entre ellos, está el informe resumido de tiempos, el informe de interacción de relojes, el informe sobre la utilización de elementos o el informe sobre la estimación de energía de la netlist de la síntesis.

Los factores más importantes a la hora de convertir el código a un circuito equivalente son la descripción del circuito, los recursos disponibles (de la arquitectura de la FPGA) y las directivas de síntesis elegidas. A la hora de la descripción no sólo se establece cómo va a funcionar el circuito sino que también se describe cómo se hace. Los recursos afectan a la implementación de las funciones en recursos lógicos. Y las directivas son los parámetros configurables que se tienen en cuenta cuando se implementan las funciones.

El flujo del proceso de síntesis para por la comprobación del diseño, la optimización y el mapeado de la tecnología.

La netlist creada aquí pueden tener distintos formatos, como EDIF / EDF / SEDIF / EDN. Vivado usa el formato EDIF (“*Electronic Design Interchange Format*”).

3.2.4. Implementación

También conocida como “*Place and Route*” consiste en convertir una o más netlist en un patrón específico de FPGA, es decir, transforma la netlist obtenida durante el proceso de síntesis y la mapea en la arquitectura de la FPGA. Este proceso se divide en los siguientes subpasos (Figura 3.10):

1. “**Translation**”
2. “**Mapping**”
3. “**Place-and-Route**”

1. Translate. El trabajo del traductor es recopilar las netlists en una sola gran netlist y verificar que se cumplan las restricciones. Si algún módulo no ha sido detectado en el proceso de síntesis, en esta etapa se marcará.

2. Mapping. Se encarga de comparar los recursos especificados en la netlist creada en el anterior paso con los recursos de la FPGA objetivo. Si no están especificados todos los recursos o hay alguno que es incorrecto, se mostrará un error y se detiene la implementación.

3. Place and Route. Es un proceso iterativo que intenta colocar (“place”) los recursos, luego enruta (“route”) las señales entre los recursos cumpliendo las limitaciones de tiempo. Esta etapa usa el **UCF** (“User Constraints File”) que especifica el tiempo máximo que puede durar la transmisión de una señal de un recurso a otro.

Vivado tiene integrado las opciones para la realización de este proceso. En concreto, hay un apartado llamado *Implementation* dentro de *Flow Navigator*. En esta parte encontramos ajustes de implementación, la ejecución de la misma y una serie de opciones que se habilitan cuando se realiza todo este proceso sin errores. Estas opciones son las mismas que las explicadas anteriormente en el apartado 5.2.3 (**Síntesis**), la ejecución de la implementación, *Schematic*, *Constraints Wizard*, *Edit Timing Constraints* y una serie de informes que nos indican el estado del diseño, entre ellos, está el informe resumido de tiempos, el informe de interacción de relojes, el informe sobre la utilización de elementos o el informe sobre la estimación de energía de la netlist de la implementación.

Cuando se quiere realizar la ejecución de la implementación, si no se ha realizado previamente la síntesis, Vivado te muestra una advertencia de que no se ha realizado antes y después hace la ejecución de la síntesis y posteriormente la de la implementación como queríamos.

Una cosa importante a tener en cuenta es que si a la hora de realizar esta compilación, los pines que vayamos a usar no han sido asignados, nos saldrá con errores y no podremos continuar. Para que no nos suceda esto, después de la síntesis se pueden asignar los pines en el *Layout Selector* cambiando *Default Layout* por *I/O Planning* (Ver figura 3.7). Otra manera sería *Window → I/O Ports*, pero sólo se vería la ventana donde se asignan los pines manualmente y no gráficamente como de la otra manera.

3.2.5. Generación Bitstream

El último paso es convertir el diseño que se ha generado en los pasos anteriores a un formato que permita configurar la FPGA asignando el valor apropiado a las celdas SRAM de configuración del dispositivo. El fichero bitstream puede ser generado para cargarlo directamente en la FPGA a través de la conexión USB o en un a memoria no volátil como una *PROM* (“Programmable Read-Only Memory”).

En Vivado hay un apartado dentro de *Flow Navigator*, llamado *Program and Debug* donde se puede generar este fichero e incluso realizar ajustes sobre el bitstream generado. Por último está *Open Hardware Manager* que tiene tres opciones, de las cuales dos están deshabilitadas hasta que se usa la única habi-

litada. Esta es *Open Target* que nos permite conectar la tarjeta al ordenador. Para saber que no ha habido ningún error en la conexión, en Vivado nos debería de mostrar el nombre de la tarjeta (Ver Figura 3.12)

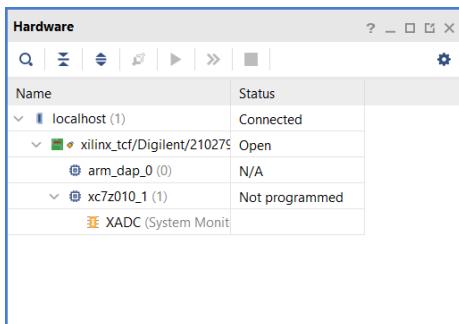


Figura 3.12: *Hardware Manager*

Una vez hecho esto se habilita la opción *Program Device* que se encarga de descargar el fichero a la FPGA. Y si todo está correcto se habilitará también la última opción para añadir alguna configuración a la memoria *Add Configuration Memory Device*.

3.3. Módulos IP específicos

En esta sección se van describir módulos de especial interés en la plataforma para la realización de prácticas, en concreto un procesador específico, un módulo de memoria RAM de un sólo puerto con entradas registradas, un módulo de interfaz para sincronización VGA y un módulo de generación de reloj.

Vivado ofrece un catálogo IP que permite agregar módulos IP a los diseños. Este catálogo contiene los módulos de Xilinx, pero se puede ampliar añadiendo módulos de “System Generator” para diseños DSP (**MATLAB \ Simulink**), diseños de Vivado HLS (algoritmos *C/C++*), módulos IP de terceros o diseños empaquetados como módulos IP usando alguna herramienta de empaquetado de módulos IP Vivado [34].

Los métodos disponibles para trabajar con módulos IP en un diseño son:

- Utilizar el “*Manage IP*” para ver el catálogo IP, personalizar módulos IP y generar salidas, incluyendo el **DCP** (Control de Diseño Sintetizado) para guardar la configuración para su uso en otras versiones. La personalización de módulos IP (XCI) y los productos de salida generados se almacenan en directorios separados ubicados fuera del proyecto Manage IP. El proyecto Manage IP gestiona las ejecuciones de diseño de módulos IP para la generación de archivos de punto de control de diseño sintetizado (DCP) y otros productos de salida.
- Utilizar módulos IP en los modos *Proyecto* o *No Proyecto* referenciando el archivo **XCI** (“Xilinx core instance”) creado, que es un método reco-

mendado para trabajar con proyectos grandes en los que participan varias personas.

- Acceder al catálogo IP desde un proyecto para personalizar y agregar un módulo IP al diseño, almacenar los archivos IP localmente en el proyecto o fuera del mismo.

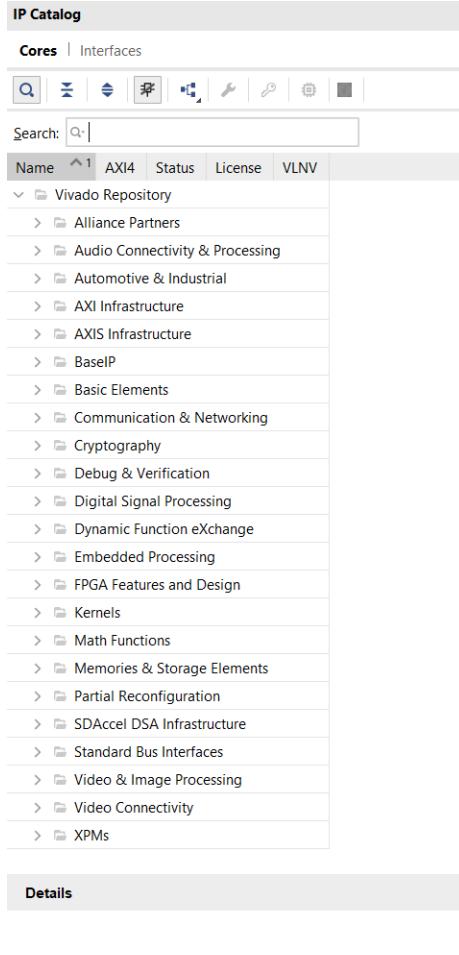


Figura 3.13: *IP Catalog*

En este TFG se trabaja con la tercera opción. Accediendo al catálogo IP desde *Flow Navigator* → *Project Manager* y vemos toda la selección de módulos IP disponibles (Ver Figura 3.13)

A continuación se van a describir los módulos que se han generado en este TFG para ser utilizados en prácticas. El procesador es una adaptación del procesador propuesto en el Capítulo 9 de [13], el módulo de sincronización está basado en el propuesto en el Capítulo 10 del mismo libro. Los otros módulos se obtienen a partir del catálogo de componentes IP de Vivado.

3.3.1. Procesador específico

Un computador digital tradicional consta de tres unidades principales, el procesador o la unidad central de procesamiento (CPU), la memoria que almacena las instrucciones y los datos del programa y el hardware de entrada/salida que se comunica con otros dispositivos.

Estas unidades están conectadas por una colección de señales digitales paralelas llamadas bus. Normalmente, las señales en el bus incluyen la dirección y los datos de la memoria, y el estado del bus. Las señales de estado del bus indican la operación actual del bus, la lectura y la escritura de la memoria o la operación de entrada / salida.

La operación principal realizada por el procesador es la ejecución de secuencias de instrucciones almacenadas en la memoria principal. La CPU o el procesador lee o recupera una instrucción de la memoria, decodifica la instrucción para determinar qué operaciones se requieren y luego ejecuta la instrucción. La unidad de control controla esta secuencia de operaciones en el procesador.

Para demostrar el funcionamiento de un computador, el siguiente código VHDL describe un procesador sencillo, que ya se venía utilizando en la asignatura Desarrollo Hardware Digital.

```
1 -- Descripcion de una procesador que ejecuta cuatro instrucciones.
2 -- Basado en ejemplo de Hamblen, J.O., Hall T.S., Furman, M.D.:
3 -- Rapid Prototyping of Digital Systems : SOPC Edition, Springer 2008.
4 -- (Capítulo 9)
5
6 LIBRARY IEEE;
7 USE IEEE.STD_LOGIC_1164.ALL;
8 USE IEEE.STD_LOGIC_ARITH.ALL;
9 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
10
11 entity procesador is
12     PORT( clock : IN STD_LOGIC;
13           reset : IN STD_LOGIC;
14           AC_out : out std_logic_vector(15 downto 0);
15           IR_out : out std_logic_vector(15 downto 0);
16           PC_out : out std_logic_vector(7 downto 0);
17           MEMq : in std_logic_vector(15 downto 0);
18           MEMdata: out std_logic_vector(15 downto 0);
19           MEMwe : out std_logic;
20           MEMadr : out std_logic_vector(7 downto 0);
21           IO_input : in std_logic_vector(7 downto 0);
22           IO_output : out std_logic_vector(7 downto 0)
23     );
24 end procesador;
25
26 architecture Behavioral of procesador is
27
28 TYPE STATE_TYPE IS ( reset_pc, fetch1, fetch0, decode, add1, add0, load1,
29                     load0,
30                     store0, store1, jump);
31 SIGNAL state: STATE_TYPE;
32 SIGNAL IR, AC, RT: STD_LOGIC_VECTOR(15 DOWNTO 0 );
33 SIGNAL PC : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
34
35 BEGIN
36
37     -- Asignaciones a puertos de salida
38     --
39     AC_out <= AC;
40     IR_out <= IR;
41     PC_out <= PC;
```

```

42
43
44 FSMD: PROCESS ( CLOCK, RESET, state, PC, AC, IR )
45
46 BEGIN
47
48 -- Asignaciones a REGISTROS en datapath y MAQUINA DE ESTADOS de la unidad de
49 -- control
50
51 --version original
52 IF reset = '1' THEN
53     state <= reset_pc;
54     ELSIF clock'EVENT AND clock = '1' THEN
55         CASE state IS
56             WHEN reset_pc =>
57                 PC      <= "00000000";
58                 AC     <= "0000000000000000";
59                 state <= fetch0;
60             WHEN fetch0 =>
61                 state <= fetch1;
62             WHEN fetch1 =>
63                 IR     <= MEMq;
64                 PC     <= PC + 1;
65                 state <= decode;
66             WHEN decode =>
67                 CASE IR( 15 DOWNTO 8 ) IS
68                     WHEN "00000000" =>
69                         state <= add0;
70                     WHEN "00000001" =>
71                         state <= store0;
72                     WHEN "00000010" =>
73                         state <= load0;
74                     WHEN "00000011" =>
75                         state <= jump;
76                     WHEN OTHERS =>
77                         state <= fetch0;
78                 END CASE;
79             WHEN add0 =>
80                 state <= add1;
81             WHEN add1 =>
82                 AC     <= AC + MEMq;
83                 state <= fetch0;
84             WHEN store0 =>
85                 state <= store1;
86             WHEN store1 =>
87                 state <= fetch0;
88             WHEN load0 =>
89                 state <= load1;
90             WHEN load1 =>
91                 AC     <= MEMq;
92                 state <= fetch0;
93             WHEN jump =>
94                 PC     <= IR( 7 DOWNTO 0 );
95                 state <= fetch0;
96             WHEN OTHERS =>
97                 state <= fetch0;
98         END CASE;
99     END IF;
100
101 -- Asignaciones a BUSES de entrada a MEMORIA (Direcciones, Datos y control de
102 -- escritura)
103
104 --version original
105 CASE state IS
106     WHEN fetch0 =>
107         MEMadr <= PC;
108         MEMwe <= '0';
109         MEMdata <= (others => '-');
110     WHEN add0 | load0 =>
111         MEMadr <= IR(7 downto 0);
112         MEMwe <= '0';
113         MEMdata <= (others => '-');
114     WHEN store0 =>
115         MEMadr <= IR(7 downto 0);

```

```

114      MEMwe <= '1';
115      MEMdata <= AC;
116      WHEN others =>
117          MEMadr <= IR(7 downto 0);
118          MEMwe <= '0';
119          MEMdata <= (others => '-');
120      end case;
121  END PROCESS;
123 end Behavioral;
124

```

Es básicamente una máquina de estados basada en VHDL que implementa el ciclo de búsqueda, decodificación y ejecución. Las primeras líneas declaran registros internos para el procesador junto con los estados necesarios para el ciclo de recuperación, decodificación y ejecución. Se necesita un estado de reinicio para inicializar el procesador. En el estado de restablecimiento, varios de los registros se restablecen a cero y se inicia una lectura de memoria de la primera instrucción. Esto obliga al procesador a comenzar a ejecutar instrucciones en la ubicación 00 en un estado predecible después de un reinicio.

El estado de búsqueda agrega uno a la PC y carga la instrucción en el registro de instrucciones (IR). Después del flanco ascendente de la señal de reloj, comienza el estado de decodificación. En la decodificación, los ocho bits inferiores del registro de instrucciones se utilizan para iniciar una operación de lectura de memoria en caso de que la instrucción necesite un operando de datos de la memoria. El estado de decodificación se decodifica la instrucción usando el valor del código de operación en los ocho bits más altos de la instrucción. Esto significa que el computador puede tener hasta 256 instrucciones diferentes, aunque solo cuatro están implementadas en el modelo básico. Después del flanco ascendente de la señal de reloj, el control se transfiere a un estado de ejecución que es específico para cada instrucción.

Algunas instrucciones pueden ejecutarse en un ciclo de reloj y algunas instrucciones pueden tardar más de un ciclo de reloj. Las instrucciones que escriben en la memoria requerirán más de un estado para ejecutarse debido a las limitaciones de tiempo de la memoria. Para garantizar que se cumplen las restricciones temporales las entradas de memoria están registradas. Esto ayuda a que la escritura se realice correctamente y por lo tanto que cuando se dé una orden a la memoria en ciclo, la operación se tiene que realizar en el siguiente ciclo.

Como se ve en la instrucción STORE, la dirección de la memoria y los datos deben ser estables antes y después de que la señal de escritura de la memoria sea alta, por lo tanto, se utilizan estados adicionales para evitar violar la configuración de la memoria y los tiempos de espera. Cuando cada instrucción termina el estado de ejecución, se inicia la búsqueda de la siguiente instrucción. Después del estado de ejecución final de cada instrucción, el control vuelve al estado de recuperación.

Este procesador puede ejecutar 4 instrucciones: **ADD**, **STORE**, **LOAD** Y **JUMP**. El repertorio inicial de instrucciones corresponde al del computador elemental presentado en el capítulo 9 de [13]. Cada instrucción ocupa 16 bits. En la versión original, los primeros 8 bits (los más significativos) corresponden a un código de operación mientras que los restantes son la dirección.

CODOP	Instrucción	Descripción
00	ADD address	$AC \leftarrow AC + M(address)$
01	STORE address	$M(address) \leftarrow AC$
02	LOAD address	$AC \leftarrow M(address)$
03	JUMP address	$PC \leftarrow address$

3.3.2. Memoria RAM de un sólo puerto con entradas registradas

En este apartado se va a diseñar un módulo de memoria RAM con un sólo puerto y cuyas entradas están registradas y que luego será conectado al procesador para realizar uno de los casos prácticos.

Para poder generar un bloque de memoria ya definido tenemos que seleccionar *IP Catalog* en el *Project Manager*. Con esto, nos saldrá una lista de módulos IP disponibles. Lo que buscamos se encuentra en *Memories and Storage Elements → RAMs and ROMs and BRAM* y su nombre es **Block Memory Generator**. Al seleccionarlo tendremos un asistente de configuración que nos ayudará a poder definir la memoria como queramos (Ver Figura 3.14).

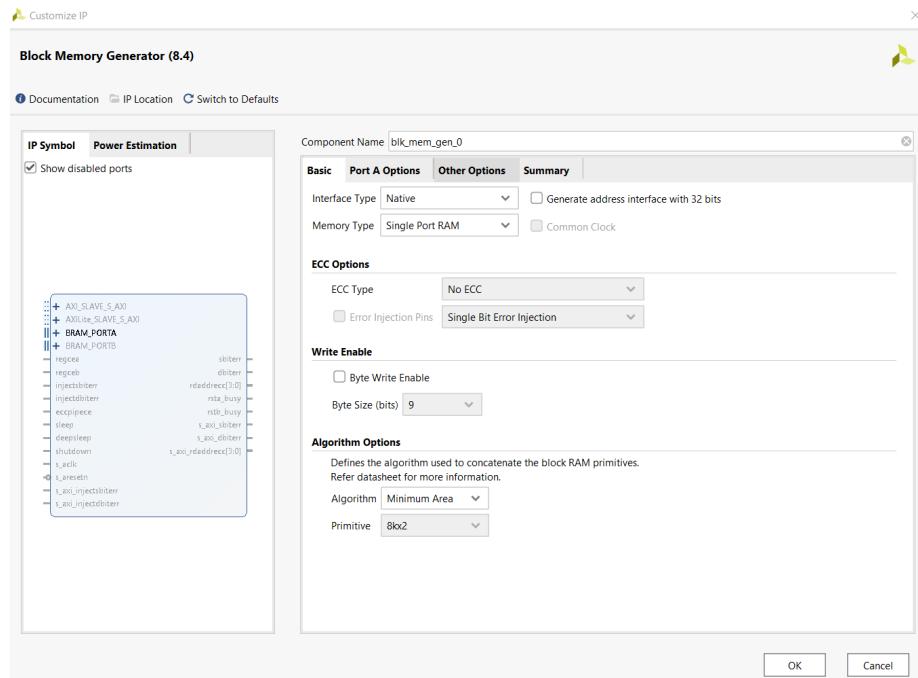


Figura 3.14: *Block Memory Generator*

Este asistente construye una memoria que utiliza memorias optimizadas para área y rendimiento usando recursos de bloques RAM integrados en FPGAs de Xilinx [29].

Este módulo de memoria, tiene dos puertos independientes que acceden a un

espacio de memoria compartida, teniendo ambos interfaz de lectura y escritura. La interfaz utilizada puede ser “*Native*” o “*AXI*”. En este domucumento nos vamos a centrar en la primera opción.

Se pueden generar cinco tipos de memorias, *Single-port RAM*, *Simple Dual-port RAM*, *True Dual-port RAM*, *Single-port ROM* y *Dual-port ROM*. Para las memorias con dos puertos, cada uno de ellos es independiente en el modo de funcionamiento, la frecuencia de reloj, registros de salida opcionales y pines se pueden seleccionar por puerto. Las entradas de la memoria están registradas para que cuando se dé una orden a la memoria en un ciclo, la operación se realice en el siguiente, y así la escritura se ejecuta correctamente.

Otra parte importante de este módulo es la selección del algoritmo usado para unir los bloques de memoria RAM embebida en la FPGA:

- **Algoritmo de área mínima:** La memoria se genera usando el mínimo número de bloques de memoria RAM, proporcionando soluciones optimizadas y reduce la multiplexación de salida. Se usan tanto datos como bits de paridad.
- **Algoritmo de baja potencia:** La memoria se genera de manera que se habilita el mínimo número de bloques de memoria RAM durante la operación de lectura o escritura. Este algoritmo no está optimizado para el área y podría usar más RAM y multiplexores que el algoritmo de área mínima.
- **Algoritmo de primitiva fija:** La memoria se genera usando sólo un tipo de bloque de memoria RAM único. El núcleo construye la memoria concatenando este tipo de memoria único en ancho y profundidad.

Se pueden generar bloques de memoria con estructuras de 1 a 4608 bits de ancho y al menos dos ubicaciones de profundidad. La profundidad máxima de la memoria está limitada sólo por el número de primitivas de bloque RAM en el dispositivo de destino.

Además admite varios modos de funcionamiento de primitivas de bloque RAM como “*WRITE FIRST*”, “*READ FIRST*” y “*NO CHANGE*” y a cada puerto se le puede asignar un propio modo de funcionamiento.

Existe la opción de habilitar escritura que proporciona soporte de escritura de bytes para anchos de memoria que son múltiplos de ocho (sin paridad) o nueve (con paridad).

También se proporcionan dos etapas opcionales de registro de salida para aumentar el rendimiento de la memoria. Estos registros se pueden elegir por separado para los dos puertos.

Hay dos pines principales que son opcionales, el pin “*enable*” que hace que se permita controlar el funcionamiento de la memoria, y si está desativado, no se realizan operaciones de lectura, escritura o reseteo en el puerto correspondiente. Si no se usa este pin, el puerto por defecto está habilitado. Y el pin “*reset*” que inicializan la salida de lectura de cada puerto a un valor programable.

La memoria tiene una opción para ser inicializada, usando un archivo **COE** o usando una opción de datos predeterminada. Un archivo **COE** puede definir

el contenido inicial de cada ubicación de memoria individual, mientras que la opción de datos predeterminada define el contenido inicial de todas las ubicaciones.

Un **COE** es un archivo de texto en el que se especifican dos parámetros:

- “**memory_initialization_radix**” - Es la base de los valores del vector de inicialización de la memoria. Los valores válidos son: 2 (binario), 10 (decimal) o 16 (hexadecimal).
- “**memory_initialization_vector**” - Define el contenido de cada elemento de la memoria.

Un ejemplo de fichero COE podría ser:

```

1 |     memory_initialization_radix = 16;
2 |     memory_initialization_vector =
3 |     12, 34, 56, 78, AB, CD, EF, 12, 34, 56, 78, 90, AA, A5, 5A, BA;
```

Este ejemplo sería para una RAM de 8 bits de ancho y 16 de profundidad. Los punto y coma marcan el final de una línea y los coeficientes se separan mediante comas, espacios o cada uno en una línea distinta.

Para comprobar la funcionalidad de este bloque de memoria, en la siguiente simulación (Figuras 3.15 3.16) se realizan operaciones de escritura y lectura, para mostrar que las entradas están registradas.

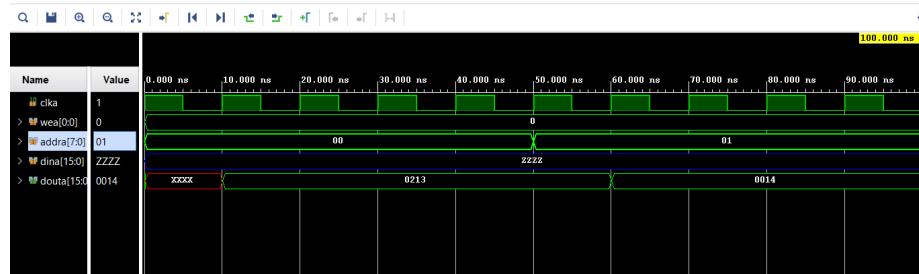


Figura 3.15: Block Memory Generator Simulation (1)

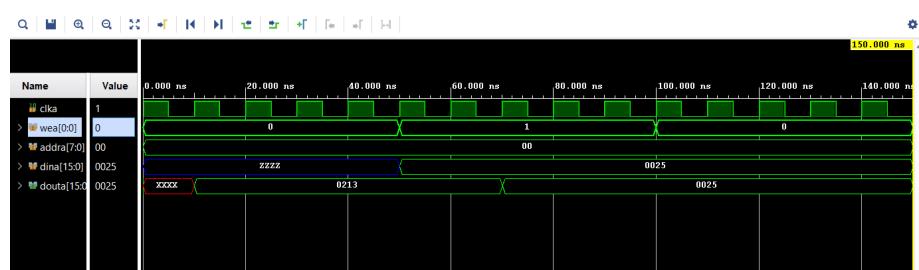


Figura 3.16: Block Memory Generator Simulation (2)

En esta simulación vemos como estando la señal de escritura sin habilitar, **wEA** con valor “0”, podemos leer lo que se encuentra en la primera dirección (“00”), en este caso “0213”, con la señal de salida **dout**. Si continuamos con el valor de esta señal y cambiamos a la dirección “01”, observamos que le corresponde el valor “0014”. Ahora que sabemos que la memoria tiene un correcto funcionamiento de lectura, ahora comprobamos si también es el caso de la escritura. Para ello, habilitamos la señal de escritura, **wEA** con valor “1”, y en la señal de entrada **dina**, escribimos cualquier valor, por ejemplo “0025” en la dirección “00”. En el momento que volvemos a deshabilitar esta señal, podemos ver cómo la dirección en la que habíamos escrito ha cambiado de valor. Al estar las entradas registradas, podemos ver cómo la operación con la memoria se realiza en el siguiente ciclo del ciclo en que se ha ordenado la operación y por lo tanto, también tiene un correcto funcionamiento de escritura.

3.3.3. Módulo de generación de reloj

El Asistente de reloj (*Clocking Wizard*) es una interfaz gráfica de usuario interactiva (GUI) que crea una red de reloj basada en necesidades específicas del diseño. Con el asistente, se pueden configurar las opciones de la primitiva de reloj elegida, y además se puede dejar que el asistente determine automáticamente la primitiva y la configuración óptimas, en función de las funciones requeridas para sus redes de reloj individuales [30].

Clocking Wizard (Ver Figura 3.17) crea un circuito de reloj para una frecuencia, fase y ciclo de trabajo del reloj de salida requeridos mediante un administrador de reloj de modo mixto (MMCM - “Mixed-Mode Clock Manager”) o un bucle de seguimiento de fase (PLL - “Phase-Locked Loop”). También ayuda a verificar la frecuencia de reloj generada por la salida en la simulación. Para acceder a este módulo IP sólo tenemos que seleccionarlo así *Catálogo IP* → *FPGA Features and Design* → *Clocking* → *Clocking Wizard*

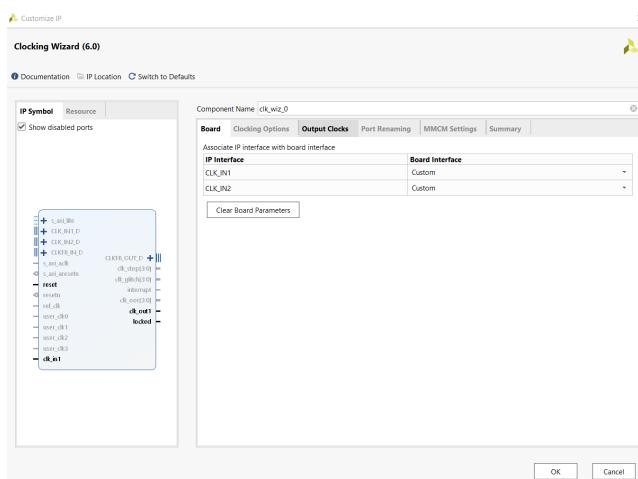


Figura 3.17: *Clocking Wizard*

La primera pestaña del IDE (*Clocking Options*) permite identificar las características requeridas de la red de reloj y configurar los relojes de entrada. En este apartado, se pueden seleccionar todas las funciones que se quieran, pero algunas de ellas consumen recursos adicionales y otras pueden generar un mayor consumo de energía.

Las opciones más importantes de esta sección son:

- **Frequency Synthesis** - permite que los relojes de salida tengan frecuencias diferentes a las del reloj de entrada activo.
- **Phase Alignment** - permite que el reloj de salida esté bloqueado en fase a una referencia, como el pin del reloj de entrada de un dispositivo.
- **Minimize Power** - permite minimizar la cantidad de energía necesaria.
- **Safe Clock Startup** - habilita un reloj estable y válido en la salida. La función de secuenciación habilita los relojes en una secuencia de acuerdo con el número ingresado en el IDE. El retraso entre dos relojes de salida habilitados en secuencia es de ocho ciclos del segundo reloj en el reloj de secuencia. Esta función es útil para un sistema en el que los módulos deben comenzar a funcionar uno tras otro.

Además encontramos la frecuencia de entrada que puede ser modificada según convenga.

La segunda pestaña *Output Clocks* permite establecer la frecuencia de salida del PLL, además de cambiar el rendimiento de ciclo (“duty cycle”) si es necesario. Y se pueden añadir más de un reloj de salida si nuestro diseño lo necesitara. La pestaña *Port Renaming* muestra información sobre los relojes de entrada y salida configurados anteriormente. La siguiente *MMCM Settings* muestra configuraciones basadas en las entradas de las pestañas anteriores. Y la última pestaña *Summary* presenta un resumen de las configuraciones hechas para el PLL.

Para comprobar el correcto funcionamiento de este módulo, se genera con las siguientes opciones. Se elige PLL y se seleccionan las opciones *Frecuency Synthesis*, *Phase Alignment* y *Minimize Power*. El reloj de entrada se deja por defecto y sólo tenemos que cambiar el valor del reloj de salida a 25MHz.

3.3.4. Controlador VGA

Para comprender cómo es posible generar una imagen de vídeo usando una placa FPGA, primero es necesario comprender los diversos componentes de una señal de vídeo. Una señal de vídeo VGA contiene 5 señales activas. Se utilizan dos señales de sincronización, horizontal y vertical, para la sincronización del vídeo y 3 señales de color que son rojo, verde y azul. A menudo se denominan colectivamente señales RGB. Al cambiar los niveles analógicos de las tres señales RGB, se producen todos los demás colores [13].

En formato VGA estándar, la pantalla contiene 640 por 480 píxeles. La señal de vídeo debe volver a dibujar la pantalla completa 60 veces por segundo para

proporcionar movimiento en la imagen y reducir el parpadeo. Este período se llama frecuencia de actualización o refresco de la pantalla. El ojo humano puede detectar parpadeos a frecuencias de actualización inferiores de 30 a 60 Hz.

El color de cada píxel está determinado por el valor de las señales RGB cuando la señal escanea cada píxel. En el modo de 640 x 480 píxeles, con una frecuencia de actualización de 60 Hz, esto es aproximadamente 40 ns por píxel. Un reloj de 25MHz tiene un período de 40ns. Una frecuencia de reloj ligeramente más alta producirá una frecuencia de actualización más alta. Este reloj es usado para controlar los contadores que generan las señales de sincronización horizontal y vertical.

Para generar las señales de sincronización horizontal y vertical, se utilizan contadores de 10 bits, cont_hs para el contador horizontal y cont_vs para el contador vertical. cont_hs y cont_vs generan una dirección de columna y fila de píxeles. Se utilizan estas señales para determinar las coordenadas x e y de la ubicación de video actual. La dirección de píxel se utiliza para generar los datos de color RGB de la imagen.

El proceso de actualización de la pantalla comienza en la esquina superior izquierda y dibuja 1 píxel a la vez de izquierda a derecha. Al final de la primera fila, la fila aumenta y la dirección de la columna se restablece a la primera columna. Cada fila se pinta hasta que se hayan mostrado todos los píxeles. Una vez que se ha pintado toda la pantalla, el proceso de actualización comienza de nuevo.

La salida de video final para las señales RGB y de sincronización en cualquier diseño debe ser directamente de una salida flip-flop. Incluso un pequeño retraso de unos pocos nanosegundos de la lógica que genera las señales de color RGB provocará una imagen de video borrosa. Dado que las señales RGB deben retrasarse un período de reloj de píxeles para eliminar cualquier posible retraso de sincronización, las señales de sincronización también deben retrasarse sincronizándolas a través de un flip-flop D. Si todas las salidas provienen directamente de una salida flip-flop, las señales de video cambiarán todas al mismo tiempo y se producirá una imagen de video más nítida.

Pero la principal diferencia es que la resolución y la frecuencia de refresco dependen de la frecuencia de reloj y de los intervalos temporales que están parametrizados para el diseño de las señales de sincronización horizontal y vertical.

```

1  -- Basado en ejemplo de Hamblen, J.O., Hall T.S., Furman, M.D.:
2  -- Rapid Prototyping of Digital Systems : SOPC Edition, Springer 2008.
3  -- (Capítulo 10)
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.all;
7  use IEEE.STD_LOGIC_ARITH.all;
8  use IEEE.STD_LOGIC_UNSIGNED.all;
9
10 entity controlador_vga_640_x_480_60 IS
11     PORT(
12         clock_25          :      IN      STD_LOGIC;
13         r,g,b    :      IN      STD_LOGIC;
14         vga_r           :      OUT     STD_LOGIC;
15         vga_g           :      OUT     STD_LOGIC;
16         vga_b           :      OUT     STD_LOGIC;
17         vga_blank_N   :      OUT     STD_LOGIC;
18

```

```

19      vga_hs          : OUT STD_LOGIC;
20      vga_vs          : OUT STD_LOGIC;
21      vga_clk         : OUT STD_LOGIC;
22      pixel_y         : OUT STD_LOGIC_VECTOR(9 DOWNTO
23                                0);
23      pixel_x         : OUT STD_LOGIC_VECTOR(9
24                                DOWNTO 0)
24 );
25
26 END controlador_vga_640_x_480_60;
27
28 ARCHITECTURE rtl OF controlador_vga_640_x_480_60 IS
29
30   -- Especificacione temporales VGA 640 x 480 pixels (60 Hz), 25M pixels/s
31   -- Sincronizacion horizontal (en numero de pixels/linea)
32   CONSTANT h_a: integer := 96; -- Retorno horizontal
33   CONSTANT h_b: integer := 48; -- "Back porch" horizontal (Margen
34   izquierdo)
35   CONSTANT h_c: integer := 640; -- Area de visualizacion horizontal
36   CONSTANT h_d: integer := 16; -- "Front porch" horizontal (Margen
37   derecho)
38   CONSTANT h_total : integer := h_a + h_b + h_c + h_d;
39
40   -- Sincronizacion Vertical (en numero de lineas/pantalla)
41   CONSTANT v_a: integer := 2; -- Retorno vertical
42   CONSTANT v_b: integer := 33; -- "Back porch" vertical
43   CONSTANT v_c: integer := 480; -- Area de visualizacion vertical
44   CONSTANT v_d: integer := 10; -- "Front porch" vertical
45   CONSTANT v_total : integer := v_a + v_b + v_c + v_d;
46
47   SIGNAL hs, vs : STD_LOGIC;
48   SIGNAL video_on : STD_LOGIC;
49   SIGNAL cont_hs, cont_vs :STD_LOGIC_VECTOR(9 DOWNTO 0);
50
51 BEGIN
52
53   -- La senial vga_clk que va al DAC coincide con el reloj de 25MHz.
54   vga_clk <= clock_25;
55
56 PROCESS
57 BEGIN
58   WAIT UNTIL(clock_25'EVENT) AND (clock_25='1');
59
60   -- Se generan las seniales de sincronizacion horizontal y vertical
61   -- a partir de los contadores cont_hs y cont_vs
62
63   -- El contador cont_hs cuenta los pixels/fila
64   -- La senial de sincronizacion horizontal (hs) vale cero durante el
65   -- retorno horizontal
66
67   IF (cont_hs = h_total - 1) THEN
68     cont_hs <= "0000000000";
69   ELSE
70     cont_hs <= cont_hs + 1;
71   END IF;
72
73   IF (cont_hs <= h_c+h_d+h_a-1) AND (cont_hs >= h_c+h_d) THEN
74     hs <= '0';
75   ELSE
76     hs <= '1';
77   END IF;
78
79   -- El contador cont_vs cuenta las filas/pantalla
80   -- La senial de sincronizacion vertical (vs) vale cero durante el retorno
81   -- vertical
82
83   IF (cont_vs = v_total - 1) AND (cont_hs >= h_total - h_a - h_d) THEN
84     cont_vs <= "0000000000";
85   ELSIF (cont_hs = h_total - h_a - h_d) THEN
86     cont_vs <= cont_vs + 1;
87   END IF;

```

```

87      IF (cont_vs <= v_c+v_d+v_a-1) AND (cont_vs >= v_c+v_d) THEN
88          vs <= '0';
89      ELSE
90          vs <= '1';
91      END IF;
92
93
94
95      -- Se generan seniales para informar a la salida de la coordenada de
96      -- pixel a visualizar
97      IF (cont_hs <= 639) THEN pixel_x <= cont_hs;      END IF;
98      IF (cont_vs <= 479) THEN pixel_y <= cont_vs; end if;
99
100     -- La senial video_on esta en alta cuando se transmite informacion de
101     -- video
102     -- (para valores de cont_vs entre 0 y 479, y valores de cont_hs entre 0
103     -- y 639)
104
105    if (cont_hs <= 639) and (cont_vs <= 479) then video_on <= '1'; else
106        video_on <= '0'; end if;
107
108    -- Se registran todas las seniales de video para eliminar retardos que
109    -- puedan emborronar la imagen
110    vga_r <= r AND video_on;
111    vga_g <= g AND video_on;
112    vga_b <= b AND video_on;
113    vga_hs      <= hs;
114    vga_vs      <= vs;
115    vga_blank_n <= video_on;
116
117 END PROCESS;
118
119 END rtl;

```

Con los valores del módulo tal como se venían utilizando en la asignatura DHD, la resolución es de 640x380 y 60Hz de frecuencia. Para conseguir una resolución mayor sólo tenemos que cambiar los valores de la siguiente parte del código:

```

1      -- Sincronizacion horizontal (en numero de pixels/linea)
2      CONSTANT h_a: integer := 128; -- Retorno horizontal
3      CONSTANT h_b: integer := 88; -- "Back porch" horizontal (Margen
4          izquierdo)
5      CONSTANT h_c: integer := 800; -- Area de visualizacion horizontal
6      CONSTANT h_d: integer := 40; -- "Front porch" horizontal (Margen
7          derecho)
8      CONSTANT h_total : integer := h_a + h_b + h_c + h_d;
9
10     -- Sincronizacion Vertical (en numero de lineas/pantalla)
11     CONSTANT v_a: integer := 4; -- Retorno vertical
12     CONSTANT v_b: integer := 23; -- "Back porch" vertical
13     CONSTANT v_c: integer := 600; -- Area de visualizacion vertical
14     CONSTANT v_d: integer := 1; -- "Front porch" vertical
15     CONSTANT v_total : integer := v_a + v_b + v_c + v_d;

```

En concreto, los valores usados son para una resolución de 800x600 píxeles con un reloj de 25MHz.

3.4. Casos prácticos

Para probar los módulos de la sección anterior, se realizan dos prácticas: un computador básico que conecta el procesador con el módulo de memoria RAM y visualización y movimiento de objetos en monitor VGA.

3.4.1. Computador básico

La finalidad de esta parte es diseñar un computador básico en VHDL, comprendiendo la descripción inicial de un procesador sencillo de 4 instrucciones, haciendo uso de una memoria RAM.

Para ello primero utilizamos la memoria diseñada en el apartado 5.3.2 con las siguientes características, memoria RAM con interfaz tipo “*Native*”, memoria “*Single Port RAM*”, algoritmo de mínima área, ancho de escritura y lectura de 16 bits, profundidad de escritura y lectura de 256 bits, modo de operación “*Read First*”, sin registros de salida e incluyendo un archivo .coe inicial como el siguiente.

```

1 memory_initialization_radix = 16;
2
3 memory_initialization_vector =
4 0213,
5 0014,
6 0015,
7 0110,
8 0304,
9 0000,
10 0000,
11 0000,
12 0000,
13 0000,
14 0000,
15 0000,
16 0000,
17 0000,
18 0000,
19 0000,
20 0000,
21 0000,
22 0000,
23 0001,
24 0002,
25 0003,
26 0000;
```

Una vez generado el módulo IP, se escribe una descripción del computador donde se instancian como componentes, la memoria RAM y el procesador, y se añaden las restricciones de la tarjeta.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity my_scomp IS
7 port( reloj : in std_logic;
8       reset : in std_logic;
9       IR_out: out std_logic_vector(15 downto 0);
10      AC_out: out std_logic_vector(15 downto 0);
11      PC_out : out std_logic_vector(7 downto 0);
12      IO_input : in std_logic_vector(7 downto 0);
13      IO_output : out std_logic_vector(7 downto 0)
14      );
15 end my_scomp;
16
17 architecture rtl of my_scomp is
18
19     signal MEMq, MEMdata: std_logic_vector(15 DOWNTO 0 );
20     signal MEMadr : std_logic_vector( 7 DOWNTO 0 );
21     signal MEMwe : std_logic;
```

```
22      signal reset_s : std_logic;
23
24      -- Declaracion del IP core ram_256_x_16
25      --
26      --      Memoria RAM de un puerto, 256 palabras, 16 bits por palabra,
27      --      Entradas de datos, direcciones y control de memoria
28      --      REGISTRADAS,
29      --      Salida NO REGISTRADA
30      --      Fichero de inicializacion: datos.coe
31
32      component blk_mem_gen_0 is
33      port
34      (
35          addra           : IN std_logic_vector (7 DOWNTO 0);
36          clka            : IN std_logic  := '1';
37          dina            : IN std_logic_vector (15 DOWNTO 0);
38          wea             : IN std_logic ;
39          douta           : OUT std_logic_vector (15 DOWNTO 0)
40      );
41      end component;
42
43      -- Declaracion del componente con version inicial del procesador
44
45      component procesador is
46      port( clock : in std_logic;
47            reset : in std_logic;
48            AC_out : out std_logic_vector(15 downto 0);
49            IR_out : out std_logic_vector(15 downto 0);
50            PC_out : out std_logic_vector(7 downto 0);
51            MEMq : in std_logic_vector(15 downto 0);
52            MEMdata: out std_logic_vector(15 downto 0);
53            MEMwe : out std_logic;
54            MEMadr : out std_logic_vector(7 downto 0);
55            IO_input : in std_logic_vector(7 downto 0);
56            IO_output : out std_logic_vector(7 downto 0)
57      );
58      end component;
59
60      begin
61
62          -- Instancia denominada MEM del IP core ram_256_x_16
63
64          MEM: blk_mem_gen_0      PORT MAP (
65              addra => MEMadr,
66              clka => reloj,
67              dina => MEMdata,
68              wea => MEMwe,
69              douta => MEMq
70          );
71
72          -- Instancia denominada PROC de la version inicial del procesador
73
74          PROC: procesador PORT MAP (
75              clock => reloj,
76              reset => reset_s,
77              AC_out => AC_out,
78              IR_out => IR_out,
79              PC_out => PC_out,
80              MEMq => MEMq,
81              MEMdata => MEMdata,
82              MEMwe => MEMwe,
83              MEMadr => MEMadr,
84              IO_input => IO_input,
85              IO_output => IO_output
86          );
87
88          estimulos: PROCESS
89          begin
90
91              reset_s <= '1';
92              WAIT FOR 10ns;
93              reset_s <= '0';
94              WAIT;
```

```

95      end PROCESS;
96
97
98 end rtl;

```

Una vez añadidos estos ficheros, realizamos la simulación del conjunto (Figura 3.18 y 3.19). Se puede observar cómo se ejecuta la secuencia de instrucciones que hemos introducido en el fichero de inicialización. El procedimiento es el siguiente: Primero se lee lo que hay en la dirección 0, 0213, 02 indica que se va a ejecutar la instrucción LOAD, es decir, se va a leer el dato que haya en la dirección 13 y se va a asignar a la señal AC. Después, se pasa a la siguiente, 0014, 00 indica que se va a ejecutar la instrucción ADD, es decir, se va a coger lo que haya en la dirección 14, se le suma lo que se había cargado en AC y este resultado se vuelve a introducir en AC.

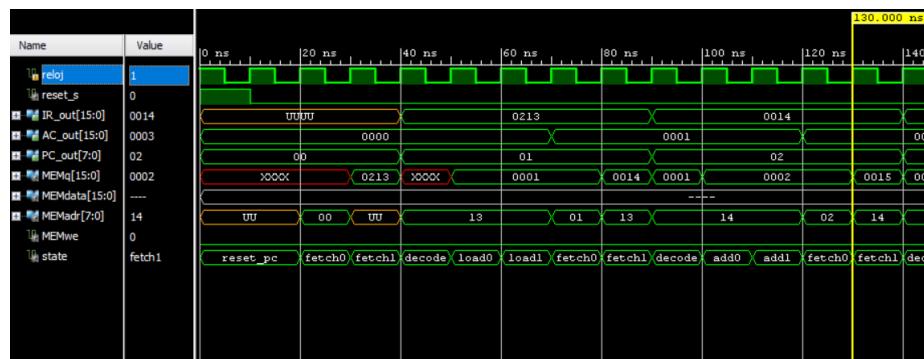


Figura 3.18: Simulación Procesador (Parte 1)

La siguiente instrucción es 0015, que hace lo mismo que la anterior. A continuación, la siguiente es 0110, 01 indica que se va a realizar un STORE, es decir, se va a cargar en la dirección 10 lo que había en AC. Por último, 0304, 03 indica que se va a hacer un JUMP, es decir, se va a saltar a la dirección 04.

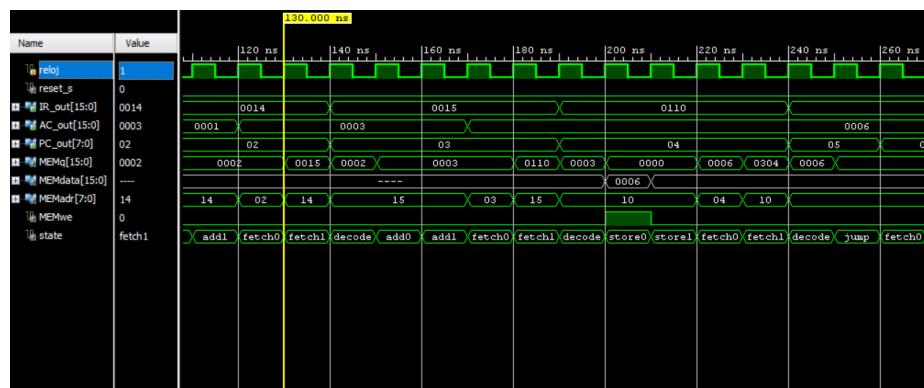


Figura 3.19: Simulación Procesador (Parte 2)

3.4.2. Visualización y movimiento de objetos en monitor VGA

En esta práctica se pretende realizar una implementación de un circuito que visualiza un objeto y lo mueve en pantalla, y que es la base para poder por ejemplo un juego tipo Pong, comprendiendo el modo de operación del módulo de interfaz VGA. Para ello disponemos de una descripción en vhdl del módulo de sincronismo VGA y un módulo que visualiza una bola cuadrada que se mueve en vertical rebotando con los bordes superior e inferior de la pantalla. Además hay que usar un módulo IP descrito en el apartado 5.3.3 que genera un reloj de 25MHz a partir del reloj que proporciona la tarjeta.

El código del controlador VGA del apartado 5.3.4 genera señales de sincronización horizontal y vertical, utilizando contadores de 10 bits. H_count hace la cuenta horizontal mientras que V_count hace la vertical. Ambos generan una dirección de píxeles en filas y columnas disponibles para otros procesos. Estas señales las usamos para determinar las coordenadas x e y de la ubicación de vídeo actual. La dirección de pixel se usa para generar los datos del color RGB de la imagen.

El módulo IP que genera el reloj de 25MHz para la resolución de 640x480 píxeles que necesitamos se especificó en el apartado 5.3.3. Los contadores se utilizan para producir señales de sincronización de vídeo. Para desactivar los datos RGB cuando no se muestran los píxeles, se genera la señal video_on [13].

Tras conocer el controlador VGA, hay que instanciar el módulo PLL y el mismo controlador con el movimiento de la bola cuya descripción se muestra a continuación:

```

1   -- Descripcion de una bola cuadrada que se mueve hacia arriba y hacia
2   -- abajo,
3   -- respetando los margenes superior e inferior de la pantalla.
4   --
5   -- Basado en ejemplo de Hamblen, J.O., Hall T.S., Furman, M.D.:
6   -- Rapid Prototyping of Digital Systems : SOPC Edition, Springer 2008.
7   -- (Capítulo 10)
8
9 LIBRARY IEEE;
10 USE IEEE.STD_LOGIC_1164.all;
11 USE IEEE.STD_LOGIC_ARITH.all;
12 USE IEEE.STD_LOGIC_UNSIGNED.all;
13
14 ENTITY bola IS
15     PORT(
16         Red,Green,Blue : OUT std_logic;
17         vs : IN std_logic;
18         pixel_Y, pixel_X : IN std_logic_vector(9 downto 0)
19     );
20 END bola;
21
22 architecture funcional of bola is
23     -- Señales para el tamaño de la bola y su desplazamiento
24     SIGNAL Bola_on : STD_LOGIC;      -- "Dibujar" bola
25     SIGNAL Desplaza_Bola_Y: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
26             Desplazamiento en píxeles de la bola en y
27     SIGNAL Desplaza_Bola_X: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
28             Desplazamiento en píxeles de la bola en x
29     SIGNAL Bola_Y : STD_LOGIC_VECTOR(9 DOWNTO 0);    -- "Eje" y de la bola
30     SIGNAL Bola_X : STD_LOGIC_VECTOR(9 DOWNTO 0);    -- "Eje" x de la bola
31
32     CONSTANT Size_X : STD_LOGIC_VECTOR(9 DOWNTO 0) :=
```

```

31           CONV_STD_LOGIC_VECTOR(4,10);    -- Tamanio del objeto en el eje x
32 BEGIN
33
34 -- Pelota gris
35 Red      <= Bola_on;
36 Green   <= Bola_on;
37 Blue    <= Bola_on;
38
39 Dibujar_Bola: Process (Bola_Y, pixel_X, pixel_Y)
40 BEGIN
41     -- Chequear coordenadas X e Y para identificar el area de la bola
42     -- Poner Bola_on a '1' para visualizar la bola
43     IF (Bola_X <= pixel_X + Size_X) AND
44         (Bola_X + Size_X >= pixel_X) AND
45         (Bola_Y <= pixel_Y + Size_X) AND
46         (Bola_Y + Size_X >= pixel_Y ) THEN
47
48         Bola_on <= '1';
49     ELSE
50         Bola_on <= '0';
51     END IF;
52
53 END process Dibujar_Bola;
54
55 Mover_Bola: PROCESS (vs)
56 BEGIN
57     -- Actualizar la posici?n de la bola en cada refresco de pantalla
58     IF vs'event and vs = '1' THEN
59         -- Detectar los bordes superior e inferior de la pantalla
60         IF Bola_Y  >= CONV_STD_LOGIC_VECTOR(479,10) - Size_X
61             THEN
62             Desplaza_Bola_Y <= CONV_STD_LOGIC_VECTOR
63                 (-2,10);
64         ELSIF Bola_Y <= Size_X  THEN
65             Desplaza_Bola_Y <= CONV_STD_LOGIC_VECTOR
66                 (2,10);
67         END IF;
68         -- Calcular la siguiente posicion de la bola
69         Bola_Y          <= Bola_Y + Desplaza_Bola_Y;
70
71         --Detectar los bordes derecho e izquierdo de la pantalla
72         IF Bola_X  >= CONV_STD_LOGIC_VECTOR(639,10) - Size_X
73             THEN
74             Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR
75                 (-2,10);
76         ELSIF Bola_X <= Size_X  THEN
77
78             Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR
79                 (2,10);
80         END IF;
81         --Calcular la siguiente posicion de la bola
82         Bola_X          <= Bola_X + Desplaza_Bola_X;
83
84     END IF;
85 END process Mover_Bola;
86
87 END funcional;

```

Las señales pixel_X y pixel_Y se usan para determinar la fila y columna actuales, respectivamente. Bola_X y Bola_Y son la dirección actual del centro de la bola. $(2 * \text{Size}_x + 1)^2$ pixeles es el tamaño de la bola cuadrada y Size_X es el tamaño de la bola en el eje x. El proceso Mover_Bola, mueve la bola unos pocos pixeles en cada sincronización vertical y comprueba si hay rebotes en la pared. Bola_on dibuja la bola en la pantalla y Desplaza_Bola_X y Desplaza_Bola_Y es el número de pixeles que la bola se tiene que mover tanto horizontal como verticalmente, dependiendo del borde en el que rebote, la bola irá en una dirección o en otra.

```

1 LIBRARY IEEE;
2 USE IEEE.STD_LOGIC_1164.all;
3 USE IEEE.STD_LOGIC_ARITH.all;
4 USE IEEE.STD_LOGIC_UNSIGNED.all;
5
6 ENTITY vga_top IS
7 PORT(
8     CLOCK : IN STD_LOGIC;
9     VGA_R : OUT STD_LOGIC_vector(4 downto 0);
10    VGA_G : OUT STD_LOGIC_vector(5 downto 0);
11    VGA_B : OUT STD_LOGIC_vector(4 downto 0);
12    VGA_HS : OUT STD_LOGIC;
13    VGA_VS : OUT STD_LOGIC;
14    KEY : IN STD_LOGIC_vector(0 to 3)
15 );
16
17 END vga_top;
18
19 ARCHITECTURE funcional OF vga_top IS
20
21     COMPONENT clk_wiz_0
22         PORT(
23             clk_in1 : IN STD_LOGIC;
24             clk_out1 : OUT STD_LOGIC );
25     END COMPONENT;
26
27     COMPONENT controlador_vga_640_x_480_60
28         PORT(
29             clock_25 : IN STD_LOGIC;
30             R,G,B : IN STD_LOGIC;
31             VGA_R, VGA_G, VGA_B : OUT STD_LOGIC;
32             vga_hs : OUT STD_LOGIC;
33             vga_vs : OUT STD_LOGIC;
34             pixel_y : OUT STD_LOGIC_VECTOR(9
35             DOWNTO 0);
36             pixel_x : OUT STD_LOGIC_VECTOR(9
37             DOWNTO 0));
38     END COMPONENT;
39
40     COMPONENT bola
41         PORT(
42             Red,Green,Blue : OUT std_logic;
43             vs : IN std_logic;
44             pixel_Y, pixel_X : IN std_logic_vector(9 downto 0)
45         );
46     END COMPONENT;
47
48 SIGNAL clock_25 : STD_LOGIC;
49 SIGNAL R_Data, G_Data, B_Data : STD_LOGIC;
50 SIGNAL pixel_x, pixel_y : STD_LOGIC_VECTOR(9 DOWNTO 0);
51 SIGNAL vs_nueva : std_LOGIC;
52 SIGNAL push_button : STD_LOGIC_VECTOR(1 DOWNTO 0);
53
54 BEGIN
55
56 --R_data <= pixel_x(0);
57 --G_data <= pixel_x(0);
58 --B_data <= pixel_x(0);
59 vga_vs <= vs_nueva;
60
61     -- PLL para generar el reloj de 25 MHz
62 PLL: clk_wiz_0 PORT MAP (
63     clk_in1 => clock,
64     clk_out1 => clock_25);
65
66     -- Controlador de la VGA
67 VGA: controlador_vga_640_x_480_60 PORT MAP (
68     clock_25 => clock_25,
69     R => R_Data,
70     g => G_Data,
71     b => B_Data,
72     vga_r => vga_r(4),
73     vga_g => vga_g(5),
74     vga_b => vga_b(4),
75     vga_hs => vga_hs,
76     vga_vs => vga_vs,
77     pixel_y => pixel_y,
78     pixel_x => pixel_x,
79     push_button => push_button,
80     vs_nueva => vs_nueva,
81     key => key);
82
83 end;

```

```
71          vga_g => vga_g(5),
72          vga_b => vga_b(4),
73          vga_hs => vga_hs,
74          vga_vs => vga_vs,
75          pixel_y => pixel_y,
76          pixel_x => pixel_x);
77
78      -- Bola
79 PELOTA: bola PORT MAP(
80     Red => R_data,
81     Green => G_data,
82     Blue => B_data,
83     vs => VS_nueva,
84     pixel_Y => pixel_y,
85     pixel_X => pixel_x
86 );
87
88 END funcional;
```

Una vez instanciados todos los componentes, sólo tenemos que ejecutar la síntesis, la implementación, asignar los pines, generar el fichero bitstream y cargar el resultado en la FPGA. Después de esto podremos ver cómo se mueve una pelota cuadrada en la pantalla, rebotando en los bordes de la pantalla (Figura 3.20).



Figura 3.20: Visualización Bola Cuadrada

Capítulo 4

Conclusiones y vías futuras

Como resultado, se han conseguido los objetivos marcados, se ha conocido y estudiado una plataforma de carácter didáctico, Vivado, a partir de la documentación facilitada por el fabricante Xilinx, para la realización de ejercicios prácticos que sean útiles para el aprendizaje del diseño de sistemas basados en FPGAs a partir de descripciones VHDL para síntesis RT.

Xilinx es la empresa líder en la fabricación de FPGAs además de en herramientas de desarrollo en los distintos niveles de síntesis automática. Además se ha analizado la plataforma Vivado junto con la tarjeta Zybo para el estudio de la metodología y flujo de diseño de sistemas basados en FPGAs a partir de descripciones VHDL para síntesis RT.

Se ha realizado un ejemplo sencillo (contador de 4 bits) con el que se ha validado el flujo de diseño y después se han realizado un par de ejemplos prácticos, un computador básico y la visualización de una bola en la pantalla, validando módulos VHDL útiles en las prácticas de laboratorio de la asignatura “Desarrollo de Hardware Digital”, como un procesador y un controlador VGA. Además para la realización de estos ejemplos se ha tenido que generar módulos de memoria y de generación de reloj a partir del catálogo de componentes IP de Vivado, los cuales se han validado y se ha comprobado su correcto funcionamiento.

Es cierto que al principio hubo algunos problemas con respecto a conocer la tarjeta usada y la plataforma de desarrollo, además de con la realización de los ejemplos prácticos. Todo esto ha tenido un buen resultado, por lo que se puede decir que se han cumplido los objetivos de este trabajo de fin de grado, resultando viable y satisfactoria la migración de las prácticas de la asignatura DHD para su realización con la herramienta Vivado y con la tarjeta de desarrollo Zybo.

Como posibles vías futuras se puede considerar la ampliación del número de módulos VHDL para integración de sistemas de interés pedagógico como módulos de interfaz PS/2 para teclado y ratón o módulos sencillos de comunicaciones como I2C o UART. Otra podría ser ampliar la plataforma con módulos descritos en C/C++ útiles en asignaturas de perfil más avanzado que emplean Vivado HLS. Incluso se podría utilizar otra tarjeta de carácter didáctico como la Zybo Z7 que es la versión actualizada de la Zybo.

La herramienta Vivado se emplea en algunas asignaturas del Máster en el que me he matriculado. Este Máster proporciona una visión general del estado del arte de las actuales y futuras tecnologías electrónicas, así como una base específica y metodológica para poder realizar labores de investigación y desarrollo en el área de los sistemas electrónicos.

Algunas de las asignaturas de este máster contienen entre otros temas la arquitectura interna de las FPGAs así como las tecnologías de fabricación más modernas y los principales fabricantes de estas. También se usa la herramienta Vivado usada en este trabajo para realizar prácticas con una placa de Digilent que incluye una FPGA Artix-7 de Xilinx.

Anexo

Como Anexo se añade un ejemplo práctico de un contador de 4 bits como primera toma de contacto. Se pretende conocer los elementos de la tarjeta **Zybo** además de conocer las diferentes fases de flujo de diseño con Vivado.

Para abordar la fase de diseño, se ha comenzado creando un proyecto en Vivado con la siguiente descripción funcional *VHDL*. Para la verificación de su funcionalidad se hizo una simulación de comportamiento (Ver Figura 4.2).

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity contador_4bits is
7     Port(    Reloj          : IN      std_logic;
8             Reset          : IN      std_logic;
9             Salida         : OUT     std_logic_vector(3 downto 0));
10 end contador_4bits;
11
12 architecture funcional of contador_4bits IS
13 signal Contador : std_logic_vector(3 downto 0):="0000"; --valores 0 a
14     255;
15 begin
16     process (Reloj, Reset)
17     begin
18         if Reset='1' then Contador <= "0000";
19             elsif reloj'event AND reloj = '1' then Contador <= Contador + 1;
20
21             end if;
22         end process;
23         Salida <= Contador;
24     end funcional;
```

Antes de la simulación, se mira la netlist resultante de la síntesis RT donde aparecen componentes genéricos independientes de la tecnología (Figura 4.1).

Para poder comprobar la correcta funcionalidad del contador realizaremos una simulación de comportamiento. Para ello, se necesita un testbench o bien añadir las señales manualmente durante la simulación. En el siguiente testbench se instancia el componente que queremos simular, el contador, y añadimos un estímulo para la señal del reloj y otra para la del reset.

← → Q Q X X ⊕ ⊖ + − C 5 Cells 6 I/O Ports 11 Nets

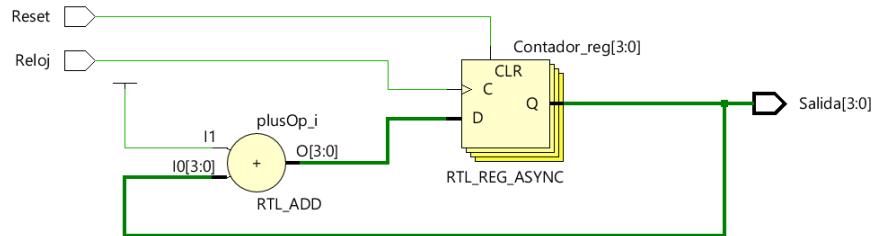


Figura 4.1: Netlist RT

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity contador_4bits_tb is
5 end contador_4bits_tb;
6
7 architecture Behavioral of contador_4bits_tb is
8     COMPONENT contador_4bits
9         PORT(
10             Reloj : in std_logic;
11             Reset : in std_logic;
12             Salida : out std_logic_vector(3 downto 0)
13         );
14     END COMPONENT;
15
16     SIGNAL reloj_s : std_logic := '0';
17     SIGNAL reset_s : std_logic;
18     SIGNAL salida_s : std_logic_vector(3 downto 0);
19
20 begin
21     DUT : contador_4bits
22     PORT MAP(
23         Reloj => reloj_s,
24         Reset => reset_s,
25         Salida => salida_s
26     );
27
28     reloj_s <= NOT reloj_s AFTER 10ns;      -- estímulo del reloj
29
30     estímulos: PROCESS
31     BEGIN
32         reset_s <= '1';
33         WAIT FOR 10ns;
34         reset_s <= '0';
35         WAIT;
36
37     END PROCESS;
38
39 end Behavioral;

```

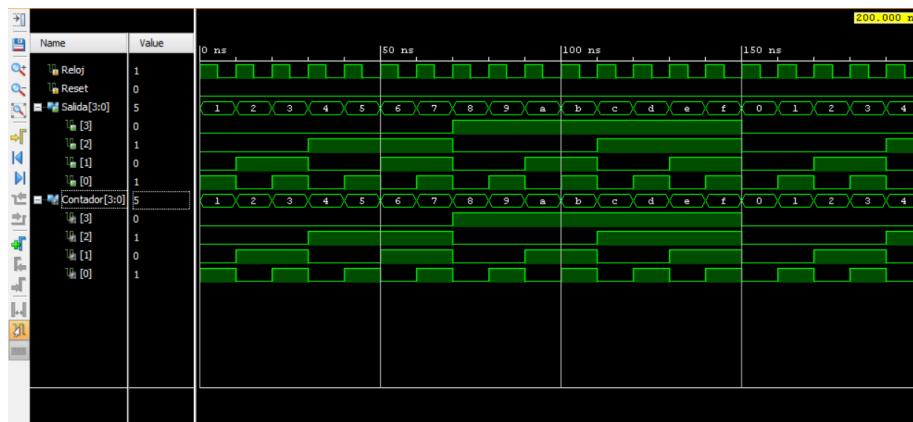


Figura 4.2: Simulación Funcional Contador 4 Bits

Si las salidas simuladas son correctas, podemos pasar a la fase de síntesis (*Synthesis*). Para ello, ejecutamos la síntesis, y si la ejecución no ha tenido errores, entonces podremos visualizar la netlist resultante con celdas propias de la biblioteca de la FPGA asignada al proyecto (la de la tarjeta Zybo) (Ver Figura 4.3).

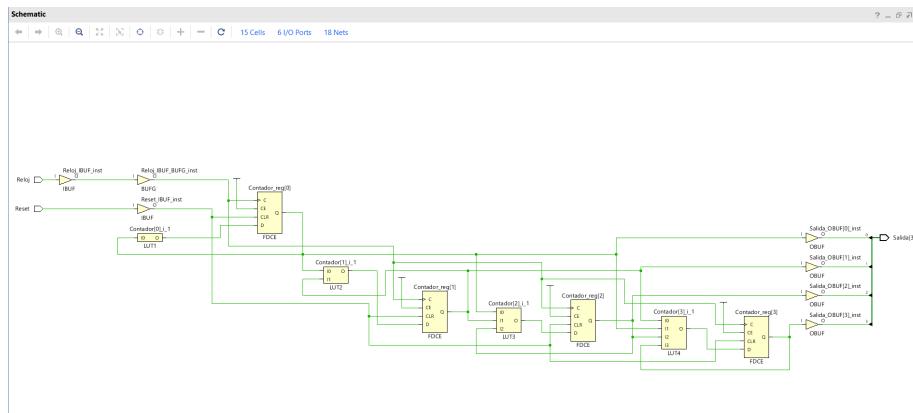


Figura 4.3: Netlist Resultante Synthesis

Para comprobar que el diseño se ejecuta correctamente, realizamos la simulación funcional post-synthesis (Ver Figura 4.4)

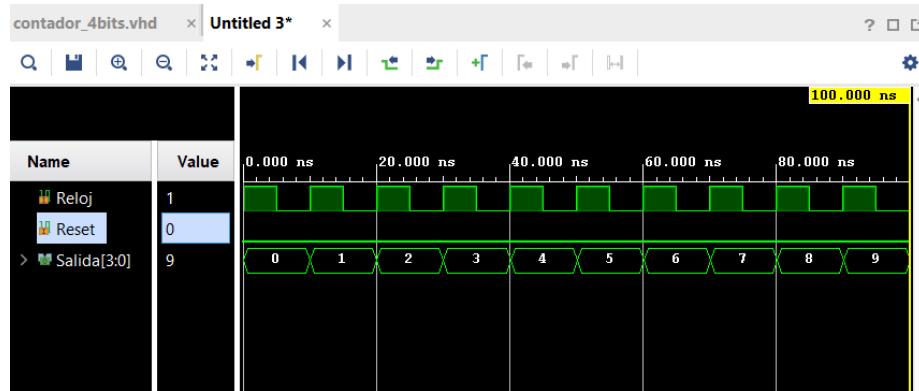


Figura 4.4: Simulación Funcional Post-Synthesis

Antes de realizar la simulación temporal, deben consultarse los resultados del análisis temporal para determinar la frecuencia máxima de funcionamiento y establecer un periodo adecuado para la señal de reloj. Al principio se usó una frecuencia de 125MHz con un reloj de 8ns, pero en el “Timing Summary” había un apartado de slacks negativos que aunque no daban error, mostraban información de que se podía aumentar la frecuencia máxima a la que trabajaba el circuito. Por lo tanto, si bajamos el período del reloj a 3,005ns tendremos una frecuencia de 332.779MHz y como consecuencia ningún slack negativo como se muestra en la figura 4.5.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0,000 ns	Worst Hold Slack (WHS): 0,131 ns	Worst Pulse Width Slack (WPWS):	0,850 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints:	0
Total Number of Endpoints: 27	Total Number of Endpoints: 27	Total Number of Endpoints:	28

All user specified timing constraints are met.

Figura 4.5: Resumen Temporal Synthesis

Después de haber ajustado el reloj a la frecuencia máxima a la que puede operar el circuito, comprobamos los retardos que tiene la netlist resultante con la simulación temporal post-synthesis (Ver Figura 4.6)



Figura 4.6: Simulación Temporal Post-Synthesis

Ahora pasamos a la fase de implementación (*Implementation*). Para ello, ejecutamos la implementación, y si la plataforma no nos muestra ningún error, sólo tenemos que añadir las restricciones de la tarjeta que estamos utilizando (*Add sources → Add or create constraints*). Además hay que asignar los pines correctamente, en este ejemplo, sólo se necesita el reloj, el reset, los botones y los LEDs.

```

1 set_property -dict { PACKAGE_PIN L16    IO_STANDARD LVCMOS33 } [get_ports {
2   clk }]; #IO_L11P_T1_SRCC_35
3 create_clock -period 3.005 -name sys_clk_pin -waveform {0.000 1.503} -add
4   [get_ports clk]
5
6 set_property -dict { PACKAGE_PIN G15    IO_STANDARD LVCMOS33 } [get_ports {
7   rst }]; #IO_L19N_T3_VREF_35
8
9 set_property -dict { PACKAGE_PIN R18    IO_STANDARD LVCMOS33 } [get_ports {
10  btn[0 }]; #IO_L20N_T3_34
11 set_property -dict { PACKAGE_PIN P16    IO_STANDARD LVCMOS33 } [get_ports {
12  btn[1 }]; #IO_L24N_T3_34
13 set_property -dict { PACKAGE_PIN V16    IO_STANDARD LVCMOS33 } [get_ports {
14  btn[2 }]; #IO_L18P_T2_34
15 set_property -dict { PACKAGE_PIN Y16    IO_STANDARD LVCMOS33 } [get_ports {
16  btn[3 }]; #IO_L7P_T1_34
17
18 set_property -dict { PACKAGE_PIN M14    IO_STANDARD LVCMOS33 } [get_ports {
19  led[0 }]; #IO_L23P_T3_35 Sch=LEDO
20 set_property -dict { PACKAGE_PIN M15    IO_STANDARD LVCMOS33 } [get_ports {
21  led[1 }]; #IO_L23N_T3_35 Sch=LED1
22 set_property -dict { PACKAGE_PIN G14    IO_STANDARD LVCMOS33 } [get_ports {
23  led[2 }]; #IO_O_35=Sch=LED2
24 set_property -dict { PACKAGE_PIN D18    IO_STANDARD LVCMOS33 } [get_ports {
25  led[3 }]; #IO_L3N_T0_DQS_AD1N_35 Sch=LED3

```

Tras realizar esta ejecución, tenemos que volver a comprobar el “Timming Summary” para asegurarnos de que no hay slacks negativos (Figura 4.7). Viendo que los slacks negativos están en positivo, podemos bajar el período del reloj usado para aumentar la frecuencia, quedando el reloj a 2.7ns y la frecuencia a 370.37MHz.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,321 ns	Worst Hold Slack (WHS): 0,094 ns	Worst Pulse Width Slack (WPWS): 0,850 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 27	Total Number of Endpoints: 27	Total Number of Endpoints: 28

All user specified timing constraints are met.

Figura 4.7: Resumen Temporal Implementation

Y ahora podemos volver a realizar la simulación funcional post-implementation y la temporal post-implementation

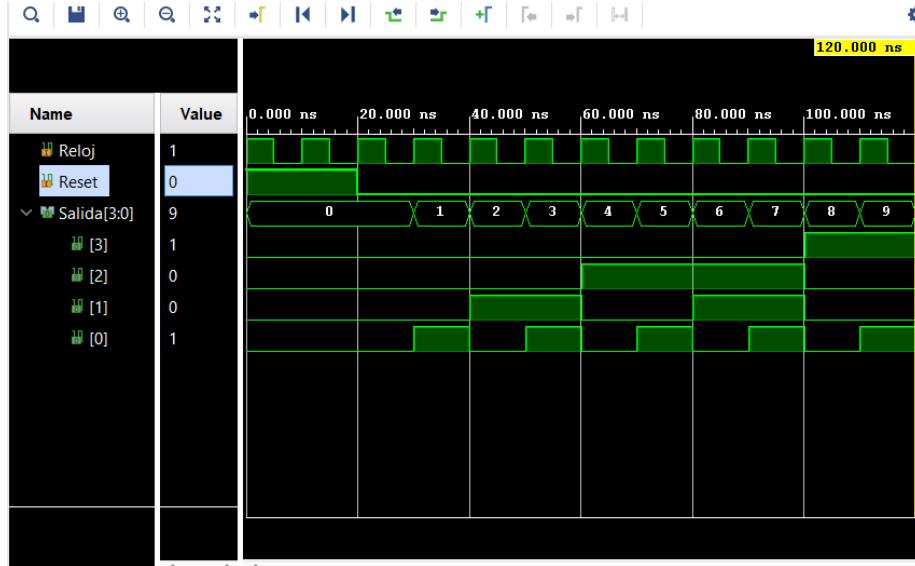


Figura 4.8: Simulación Funcional Post-Implementation



Figura 4.9: Simulación Temporal Post-Implementation

Cuando se haya realizado correctamente la ejecución de la implementación, se genera el fichero bitstream y si no hay ningún error se lleva a la FPGA. Debido a la frecuencia a la que trabaja la FPGA, la salida de los LEDs no es la

del contador, sino una luz verde fija. Para arreglar esto, tenemos que añadir un divisor de frecuencias variable cuya frecuencia de salida depende de los valores de 3 entradas.

Pero no basta sólo con añadir este archivo, necesitamos otro archivo que una estos dos anteriores. A continuación se muestra cómo se instancian el divisor de frecuencias y el contador de 4 bits.

```

1 LIBRARY IEEE;
2 USE IEEE.STD.LOGIC_1164.all;
3 USE IEEE.STD.LOGIC_ARITH.all;
4 USE IEEE.STD.LOGIC_UNSIGNED.all;
5
6 ENTITY Top IS
7   PORT(
8     btn      : IN      STD_LOGIC_VECTOR(2 downto 0);
9     clk      : IN      STD_LOGIC;
10    rst      : IN      STD_LOGIC;
11    led      : OUT     STD_LOGIC_VECTOR(0 TO 3));
12 END Top;
13
14 ARCHITECTURE estructural OF Top IS
15
16   COMPONENT Div_Frec IS
17     Port(  Velocidad      : IN      std_logic_vector(2 downto 0);
18            Reloj         : IN      std_logic;
19            Salida        : OUT     std_logic);
20   END COMPONENT;
21
22   COMPONENT contador_4bits IS
23     Port(  Reloj          : IN      std_logic;
24            Reset          : IN      std_logic;
25            Salida         : OUT     std_logic_vector(3 downto 0));
26   END COMPONENT;
27
28   SIGNAL   CKcont : std_logic;
29
30 BEGIN
31
32   DivisorFrecuencia: Div_Frec PORT MAP(
33     Velocidad => btn,
34     Reloj => clk,
35     Salida => CKcont);
36
37   Contador4: contador_4bits PORT MAP(
38     Reloj => CKcont,
39     Reset => rst,
40     Salida => led);
41
42 END estructural;
43

```

Tras haber añadido estos archivos que completan el diseño del contador, volvemos a realizar los pasos anteriormente dados, ejecución de la síntesis, de la implementación y la generación del fichero bitstream. Después lo cargamos en la FPGA y ya si se puede ver el contador reflejado en los LEDs de la tarjeta. Además con los interruptores se puede aumentar la frecuencia o disminuirla, es decir, la velocidad a la que parpadean los LEDs.

Bibliografía

- [1] MCI Capacitación. Fpga (field programmable gate array). <https://cursos.mcielectronics.cl/2019/06/18/fpga-field-programmable-gate-array/>.
- [2] Digilent. Anvyl fpga board reference manual. https://reference.digilentinc.com/_media/anvyl:anvyl_rm.pdf.
- [3] Digilent. Arty a7 reference manual. <https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual>.
- [4] Digilent. Arty s7 reference manual. <https://reference.digilentinc.com/reference/programmable-logic/arty-s7/reference-manual>.
- [5] Digilent. Basys 2 fpga board reference manual. https://reference.digilentinc.com/_media/basys2:basys2_rm.pdf.
- [6] Digilent. Basys 3 reference manual. <https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual>.
- [7] Digilent. Genesys 2 reference manual. <https://reference.digilentinc.com/reference/programmable-logic/genesys-2/reference-manual?redirect=1>.
- [8] Digilent. Pynq-z1 reference manual. <https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual>.
- [9] Digilent. System boards. <https://store.digilentinc.com/fpga-programmable-logic/system-boards/>.
- [10] Digilent. Zedboard zynq-7000 development board reference manual. <https://reference.digilentinc.com/reference/programmable-logic/zedboard/reference-manual>.
- [11] Digilent. Zybo fpga board reference manual. https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf.
- [12] Mentor Graphics. <https://www.mentor.com/>.
- [13] James O Hamblen, Tyson S Hall, and Michael D Furman. *Rapid prototyping of digital systems: SOPC edition*. Springer Science & Business Media, 2007.
- [14] Intel. <https://www.intel.com/content/www/us/en/products/programmable.html>.

- [15] Intel. Intel fpga sdk for opencl software technology. <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html?wapkw=intel%20fpga%20sdk>.
- [16] Intel. Intel high level synthesis compiler. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html?wapkw=hls%20compiler>.
- [17] Markets and Markets. Fpga market. <https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html>.
- [18] Clive Maxfield. Conceptos fundamentales de los fpga - parte 2: Presentación de los fpga de lattice semiconductor. <https://www.digikey.es/es/articles/fundamentals-of-fpgas-part-2-getting-started-with-lattice-semiconductor-fpgas>.
- [19] Clive Maxfield. Conceptos fundamentales de los fpga: ¿qué son los fpga y por qué son necesarios? <https://www.digikey.es/es/articles/fundamentals-of-fpgas-what-are-fpgas-and-why-are-they-needed>.
- [20] Clive Maxfield. Fundamentos de las fpga - parte 3: Introducción a las fpga de microchip technology. <https://www.digikey.es/es/articles/fundamentals-of-fpgas-part-3-getting-started-with-microchip-fpgas>.
- [21] Clive Maxfield. Fundamentos de los fpga - parte 4: Introducción de los fpga de xilinx. <https://www.digikey.es/es/articles/fundamentals-of-fpgas-part-4-getting-started-with-xilinx-fpgas>.
- [22] Clive Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.
- [23] Clive Maxfield. *FPGAs: instant access*. Elsevier, 2011.
- [24] Gina Smith. *FPGAs 101: Everything you need to know to get started*. Newnes, 2010.
- [25] Yang Sun, Kiarash Amiri, Guohui Wang, Bei Yin, Joseph R Cavallaro, and Tai Ly. High-level design tools for complex dsp applications. Elsevier, Waltham, MA, 2012.
- [26] Synopsys. <https://www.synopsys.com/>.
- [27] L Terés, Y Torroja, S Olcoz, and E Villar. Vhdl lenguaje estándar de diseño electrónico. editorial mc graw hill, 1997.
- [28] Wikipedia. Xilinx. <https://en.wikipedia.org/wiki/Xilinx>.
- [29] Xilinx. Block memory generator v8.4 logicore ip product guide. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.
- [30] Xilinx. Clocking wizard v6.0 logicore ip product guide. https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf.

- [31] Xilinx. Field programmable gate array (fpga). <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>.
- [32] Xilinx. Vitis unified software platform. <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html>.
- [33] Xilinx. Vivado design suite. <https://www.xilinx.com/products/design-tools/vivado.html>.
- [34] Xilinx. Vivado design suite user guide : Designing with ip. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug896-vivado-ip.pdf.
- [35] Xilinx. Vivado design suite user guide : Using the vivado ide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug893-vivado-ide.pdf.
- [36] Xilinx. Xup students. <https://www.xilinx.com/support/university/students.html#overview>.
- [37] Xilinx. Xup supported boards. <https://www.xilinx.com/support/university/boards-portfolio/xup-boards.html>.
- [38] Xilinx. Zynq-7000 soc data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.