



*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, 30 DE AGOSTO DE 2020



# Índice general

<b>1. Resumen y palabras clave</b>	<b>1</b>
<b>2. Resumen extendido y palabras clave en inglés</b>	<b>3</b>
<b>3. Motivación e introducción</b>	<b>5</b>
3.1. Sistemas basados en dispositivos FPGAs . . . . .	5
3.2. Niveles de síntesis automática . . . . .	7
3.3. Plataformas de desarrollo . . . . .	9
3.3.1. Principales fabricantes de FPGAs . . . . .	9
3.3.2. Herramientas software de desarrollo . . . . .	9
3.3.3. Plataformas de propósito académico . . . . .	9
3.4. Estructura de la memoria . . . . .	10
<b>4. Objetivos del trabajo</b>	<b>11</b>
<b>5. Resolución del trabajo</b>	<b>13</b>
5.1. Materiales . . . . .	13
5.1.1. Familia Zynq-7000 . . . . .	13
5.1.2. Tarjeta Zybo . . . . .	13
5.1.3. Vivado . . . . .	16
5.2. Metodología . . . . .	21
5.2.1. Diseño . . . . .	22
5.2.2. Simulación . . . . .	22
5.2.3. Síntesis . . . . .	24
5.2.4. Implementación . . . . .	24
5.2.5. Generación Bitstream . . . . .	25
5.3. Desarrollo de módulos hardware específicos . . . . .	26
5.3.1. Procesador específico . . . . .	27
5.3.2. Memoria RAM de un sólo puerto con entradas registradas	27
5.3.3. Módulo de generación de reloj . . . . .	29
5.3.4. Controlador VGA . . . . .	29
5.4. Casos prácticos . . . . .	29
5.4.1. Computador básico . . . . .	30
5.4.2. Visualización y movimiento de objetos en monitor VGA .	30
<b>6. Conclusiones y vías futuras</b>	<b>45</b>



# Índice de figuras

3.1. <i>Arquitectura de una FPGA</i> . . . . .	5
5.1. ZYBO Zynq-7000 Development Board . . . . .	14
5.2. Arquitectura Zynq AP SoC . . . . .	15
5.3. Vivado IDE . . . . .	17
5.4. Entorno Principal Vivado IDE . . . . .	18
5.5. <i>Flow Navigator</i> . . . . .	19
5.6. <i>Default Layout</i> . . . . .	20
5.7. <i>I/O Planning</i> . . . . .	20
5.8. <i>Clock Planning</i> . . . . .	20
5.9. <i>Floorplanning</i> . . . . .	21
5.10. <i>Timing Analysis</i> . . . . .	21
5.11. <i>Typical Xilinx FPGA Development Flow</i> . . . . .	22
5.12. <i>Hardware Manager</i> . . . . .	26
5.13. <i>IP Catalog</i> . . . . .	27
5.14. <i>Block Memory Generator</i> . . . . .	28
5.15. <i>Simulación Contador 4 Bits</i> . . . . .	31
5.16. <i>Netlist Resultante Synthesis</i> . . . . .	31
5.17. <i>Simulación Memoria RAM</i> . . . . .	34
5.18. <i>Simulación Procesador</i> . . . . .	38



## **Capítulo 1**

# **Resumen y palabras clave**



## **Capítulo 2**

# **Resumen extendido y palabras clave en inglés**



## Capítulo 3

# Motivación e introducción

### 3.1. 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 [12].

Los bloques lógicos configurables en las FPGAs basadas en celdas SRAM incluyen LUTs (*tablas de consulta*), flip-flops y multiplexores de entrada y salida (Figura 3.1). Una LUT almacena una lista de salidas lógicas para cualquier combinación de entradas.

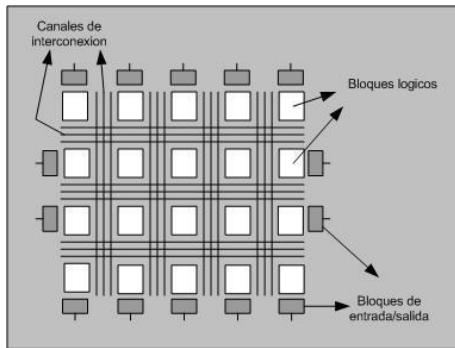


Figura 3.1: Arquitectura de una FPGA

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 [5].

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 como la **SRAM**.

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, creados para ser configurados por ingenieros en el campo para realizar diferentes funciones. 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 [4].

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. Un SoC de bajo costo y consumo se puede utilizar en aplicaciones como tarjetas de procesamiento de vídeo. Sin embargo, hay otros 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 [5].

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 casi cualquier cosa 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**

### 3.2. 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 [8].

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 acciones que se vayan a hacer.
- **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 coger 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.

#### PONER LO DE LOS LENGUAJES VHDL Y VERILOG

---

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 con respecto a la implementación ya que consume tiempo de desarrollo, validar el correcto funcionamiento de un diseño de forma más rápida que con lenguajes de descripción hardware tradicionales o crear implementaciones hardware de alto rendimiento

### 3.3. Plataformas de desarrollo

#### 3.3.1. Principales fabricantes de FPGAs

#### 3.3.2. Herramientas software de desarrollo

#### 3.3.3. Plataformas de propósito académico

Actualmente hay muchas empresas que fabrican FPGAs, pero en el top 5 se pueden encontrar *Xilinx*, *Altera*, *Lattice Semiconductor*, *Microsemi* (*antiguo Actel*) y *QuickLogic*. Tanto Xilinx como Altera ocupan un 89 % del mercado, siendo Xilinx el líder desde hace muchos años. Xilinx tiene bastante variedad de FPGAs en cuanto a coste y rendimiento. Actualmente, la serie *Virtex* y la serie *Zynq-7000* de SoC ocupan el mercado de gama alta, la serie *Kintex* de gama media y la serie *Artix* de gama baja junto con la *Spartan* que ha sido retirada del mercado.

La serie *Virtex* integra lógica **FIFO** y **ECC**, bloques **Ethernet MAC**, bloques **DSP** (*Procesador de señales digitales*), controladores **PCI-Express**. Además incluye hardware embebido con una función fija para funciones que se usan comúnmente como multiplicadores o memoria.

La serie *Kintex* se caracteriza por consumir menos energía que la serie anterior, incluyendo alto rendimiento y elementos necesarios para aplicaciones que tengan mucho volumen.

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

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

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

- *Synplify Pro, Synplify Premier (Synopsis)*
- *Precision RTL Plus, LeonardoSpectrum (Mentor Graphics)*
- *Quartus (Altera)*
- *Vivado (Xilinx)*

■ **Herramientas de síntesis de alto nivel:**

- *Synphony C Compiler (Synphony)*
- *Impulse coDeveloper (Impulse C)*
- *Vivado High Level Synthesis (Xilinx)*
- *SDSoc (Xilinx)*
- *SDAccel (Xilinx)*
- *Intel SDK for OpenCL (Intel Altera)*
- *Intel HLS Compiler (Intel Altera)*

La última herramienta comercializada por Xilinx, *Vitis* 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.).

Las plataformas de desarrollo con propósito académico que comercializa Xilinx son [15]:

- **7-series** - *Spartan-7, Artix-7, Kintex-7, Virtex-7*
- **Zynq** - *ZYBO, ZYBO Z7, ZedBoard*
- **Spartan** - *Spartan-6, Spartan-3E*
- **Virtex** - *Virtex-6, Virtex-5, Virtex-4, Virtex-2P*

### 3.4. Estructura de la memoria

## **Capítulo 4**

# **Objetivos del trabajo**



## Capítulo 5

# Resolución del trabajo

### 5.1. Materiales

#### 5.1.1. Familia Zynq-7000

La familia **Zynq-7000** integra un sistema completo con un procesador *ARM Cortex-A9 MPCore* con una lógica genérica que permite la configuración de módulos hardware específicos. Esta familia de SoCs está diseñada para llevar a cabo aplicaciones de compleja dificultad como la video-vigilancia o sistemas inalámbricos.

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 5.3) fue desarrollado para FPGAs con más capacidad y permite compilaciones de descripciones basadas en *C* gracias a la funcionalidad de síntesis de alto nivel.

#### 5.1.2. Tarjeta Zybo

Una FPGA de la familia Zynq 7000 es incluida en tarjeta **ZYBO** (*ZYBO Board*). Es una plataforma de desarrollo de circuito digital, y está construida alrededor del miembro más pequeño de la familia Zynq-7000, el **Z-7010**. 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 Zynq 7010 Ap SoC ofrece las siguientes características (Figura 5.1) [1]:

- 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,  $I^2C$

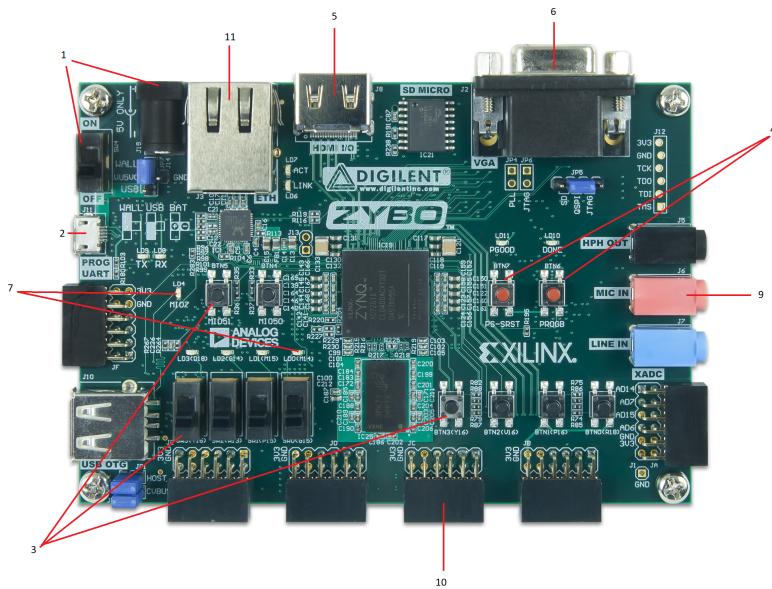


Figura 5.1: ZYBO Zynq-7000 Development Board

- Lógica Reprogramable equivalente a Artix-7 FPGA
- ZYNQ XC7Z010-1CLG400C
- Puerto HDMI
- Puerto VGA de 16 bits por pixel
- EEPROM externo
- Códec de audio con salida de auricular y micrófono
- GPIO: 6 botones, 4 interruptores, 5 LEDs
- 6 conectores Pmod

La arquitectura Zynq AP SoC está dividida en dos partes (Figura 5.2), el sistema de procesamiento (*PS*) y la lógica programable (*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*. Además, la *PL* no tiene la misma configuración hardware como las FPGAs 7-series y tiene que ser configurada por el procesador o por el puerto JTAG.

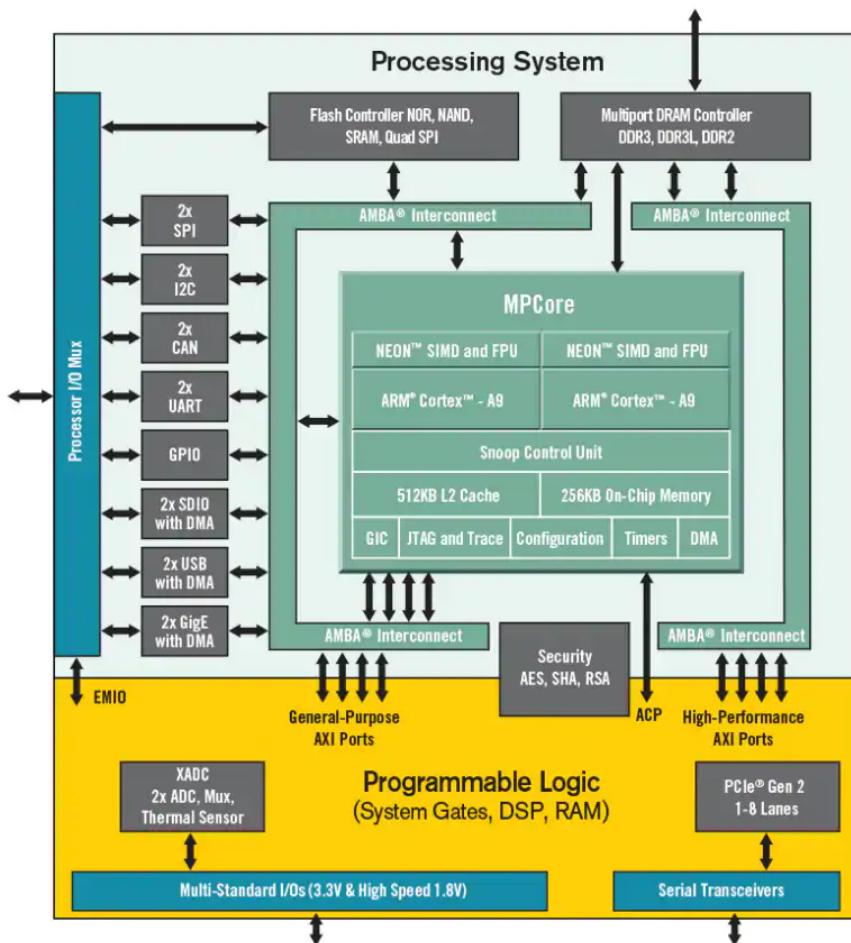


Figura 5.2: Arquitectura Zynq AP SoC

El PS consiste en un conjunto de componentes como la *APU* (Unidad de Procesamiento de Aplicaciones) que incluye dos procesadores Cortex-A9, *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.

Los elementos que forman la tarjeta Zybo son (Figura 5.1):

1. Interruptor y Conector de alimentación
2. Botones e interruptores
3. Botones de reset
4. Puerto HDMI

- 5. Puerto VGA
- 6. LEDs
- 7. Conectores de audio
- 8. Conectores Pmod
- 9. Conector Ethernet

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 vía 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 recibe señal a través de un conector VGA. Éste se suele usar para conectar dispositivos. Está compuesto de 15 pines y cada uno tiene su propia función, entre las cuales está la de transferir los colores rojo, azul y verde, la sincronización horizontal y la sincronización vertical.

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”.

### 5.1.3. Vivado

Zybo es compatible con *Vivado Design Suite* de Xilinx 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 2016.2. Para trabajar con él, se puede hacer tanto trabajando con la TCL o directamente con la GUI de Vivado IDE [14].

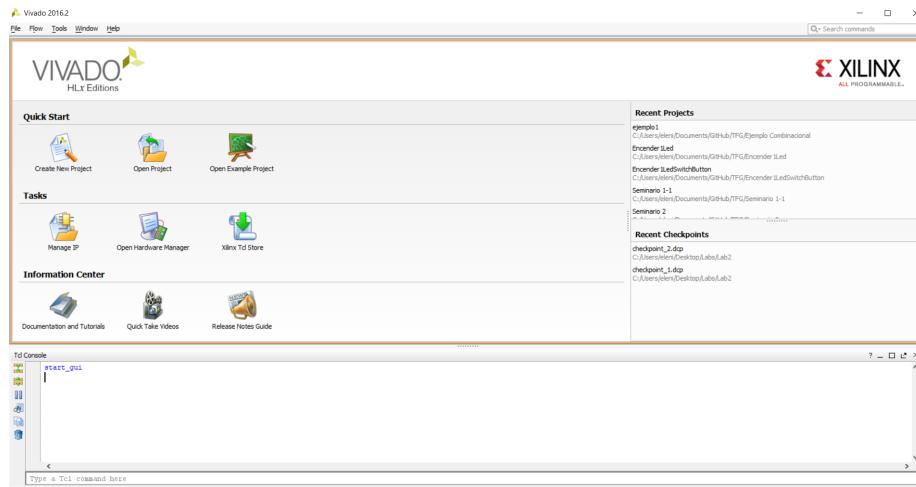


Figura 5.3: Vivado IDE

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 IP, personalizar IP y generar productos de salida. **Open Hardware Manager** nos permite conectar la tarjeta y descargar un programa en el dispositivo FPGA. **Xilinx TCL Store** es un repositorio de código TCL.

Da acceso a múltiples scripts para resolver problemas y mejorar la productividad.

La última sección es **Information 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 5.4) 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*

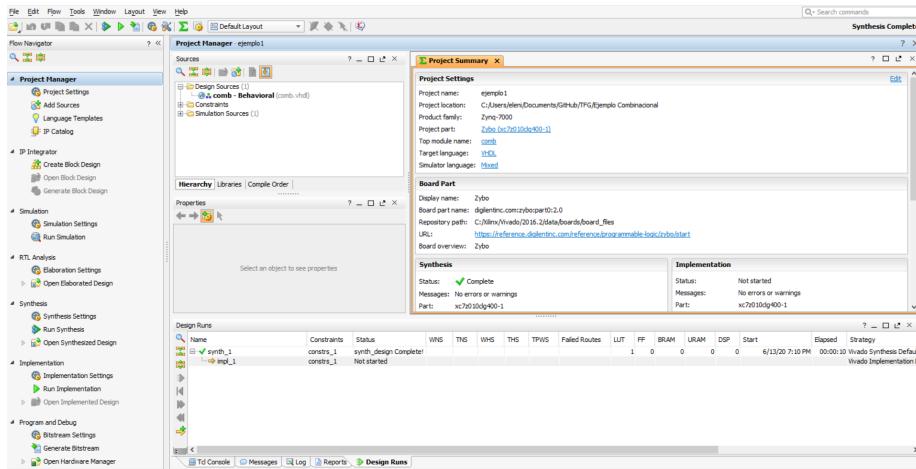


Figura 5.4: Entorno Principal Vivado IDE

*Flow Navigator* (Figura 5.5) 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:** Cambio de ajustes de simulación o simular un diseño activo.
- **RTL Analysis:** Abrir un diseño elaborado o generar un diseño de diagrama de circuitos RTL.
- **Synthesis:** Cambio de ajustes de síntesis, sintetizar un diseño activo o abrir el diseño sintetizado.
- **Implementation:** Cambio de ajuste de implementación, implementar un diseño activo o abrir el diseño implementado.
- **Program and Debug:** Cambio de ajustes del bitstream, generar un archivo bitstream o abrir una ventana para conectar la tarjeta FPGA y programarla.

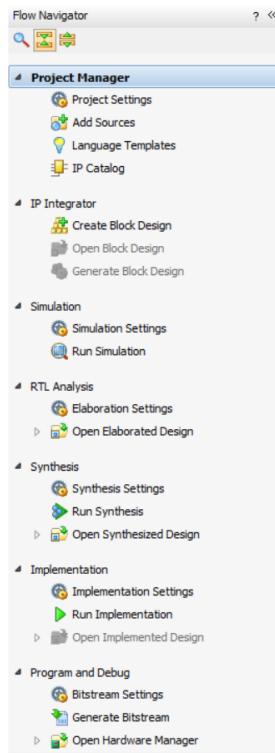


Figura 5.5: *Flow Navigator*

*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 5.6).
- **I/O Planning:** Definición de restricciones de ubicación I/O y colocación de puertos (Figura 5.7).

- **Clock Planning:** Planificación y colocación de los recursos del reloj del diseño (Figura 5.8).
- **Floorplanning:** Gestionar particiones y tareas jerárquicas (Figura 5.9).
- **Timing Analysis:** Ejecutar informes de tiempo y analizarlo (Figura 5.10).



Figura 5.6: Default Layout

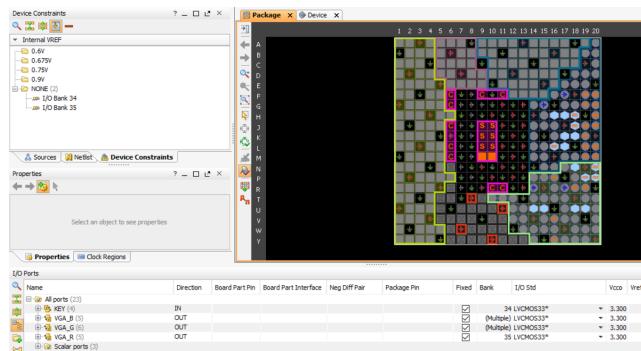


Figura 5.7: I/O Planning

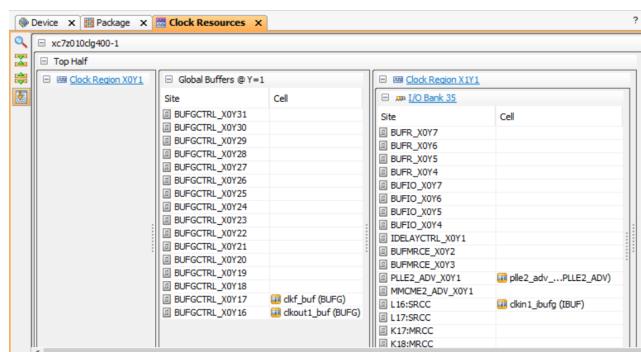
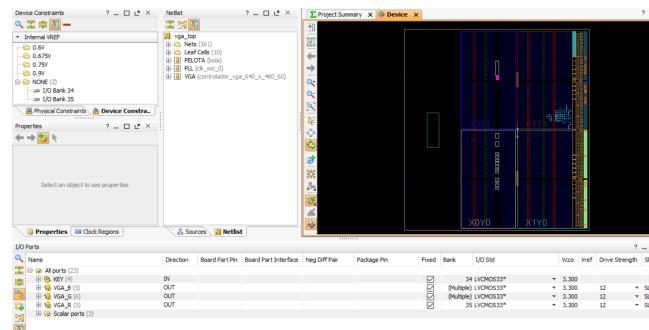
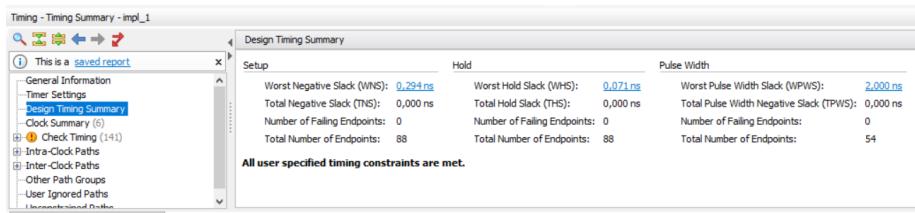


Figura 5.8: Clock Planning

Figura 5.9: *Floorplanning*Figura 5.10: *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*.

## 5.2. Metodología

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 son [6]:

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

### 5.2.1. Diseño

Vivado soporta múltiples lenguajes de descripción hardware, entre ellos VHDL y Verilog, e incluso se pueden usar ficheros con ambos lenguajes. El siguiente diagrama (Ver Figura 5.11) muestra un flujo de diseño típico con las herramientas de Xilinx.

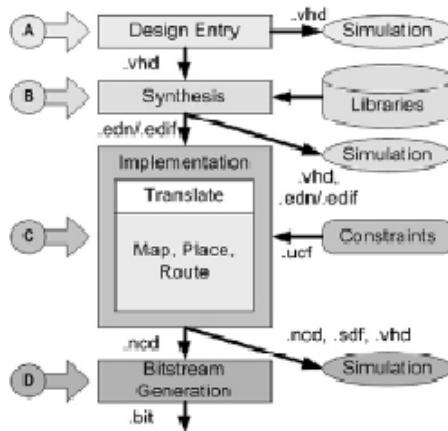


Figura 5.11: Typical Xilinx FPGA Development Flow

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.

### 5.2.2. Simulación

Cuando ya tenemos el diseño, hay que comprobar si el correcto funcionamiento del circuito. Para ello, se usa la simulación, que puede realizarse después del diseño, de la síntesis o de la implementación. A veces, la simulación se olvida por lo que lleva a largos tiempos de verificación y depuración.

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 5.11):

- 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 rightarrow 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 rightarrow 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. Para usar esta simulación Vivado tiene la siguiente opción *Flow Navigator → Simulation → Run Simulation rightarrow Run Post-Implementation Simulation*.

### 5.2.3. 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 faltan 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 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 afecta a la interpretación e 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*”).

### 5.2.4. 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 5.11):

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 de **Síntesis**.

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 5.7). 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.

### 5.2.5. Generación Bitstream

El último paso es convertir el diseño que se ha generado en los pasos anteriores a un formato 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 habilitada. 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 5.12)

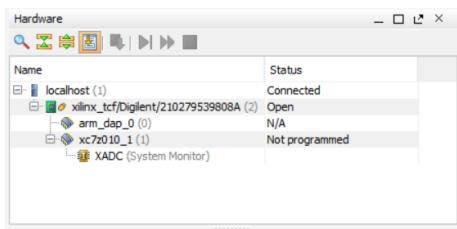


Figura 5.12: *Hardware Manager*

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*.

### 5.3. Desarrollo de módulos hardware específicos

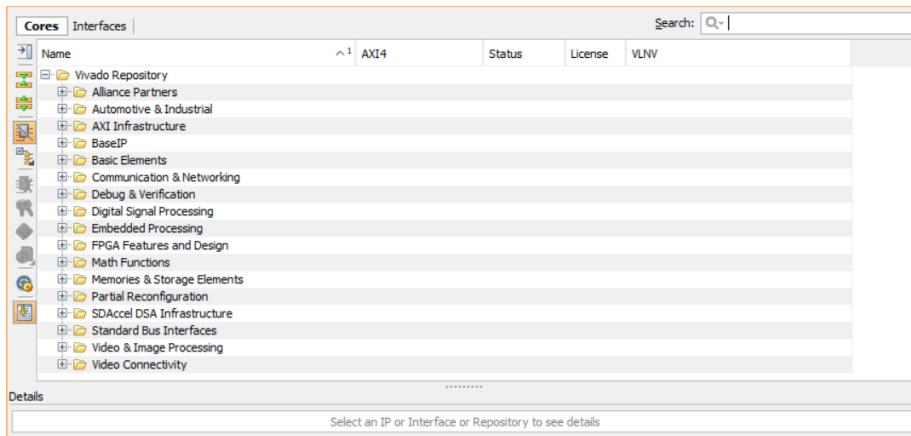
En esta sección se van describir módulos de especial interés en la plataforma para la realización de prácticas, en concreto un procesador específico, un módulo de memoria RAM de un sólo puerto con entradas registradas, un módulo de interfaz para sincronización VGA y un módulo de generación de reloj.

Vivado ofrece un catálogo IP (“*Intellectual Property*”) 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++*), IP de terceros o diseños empaquetados como IP usando alguna herramienta de empaquetado de IP Vivado [13].

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

- Utilizar el “*Manage IP*” para personalizar IP y generar salidas, incluyendo el **DCP** (Control de Diseño Sintetizado) para guardar la configuración para su uso en otras versiones.
- Utilizar IP en los modos *Proyecto* o *No Proyecto* referenciando el archivo **XCI** (“*Xilinx core instance*”) creado, que es un método recomendado 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.

En este TFG se trabaja con la tercera opción. Accediendo al catálogo IP desde *Flow Navigator* → *Project Manager* y vemos toda la selección de IP disponibles (Ver Figura 5.13)

Figura 5.13: *IP Catalog*

El procesador es una adaptación del procesador propuesto en el Capítulo 9 de [2], 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.

### 5.3.1. Procesador específico

### 5.3.2. Memoria RAM de un sólo puerto con entradas registradas

Para poder tener 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 5.17).

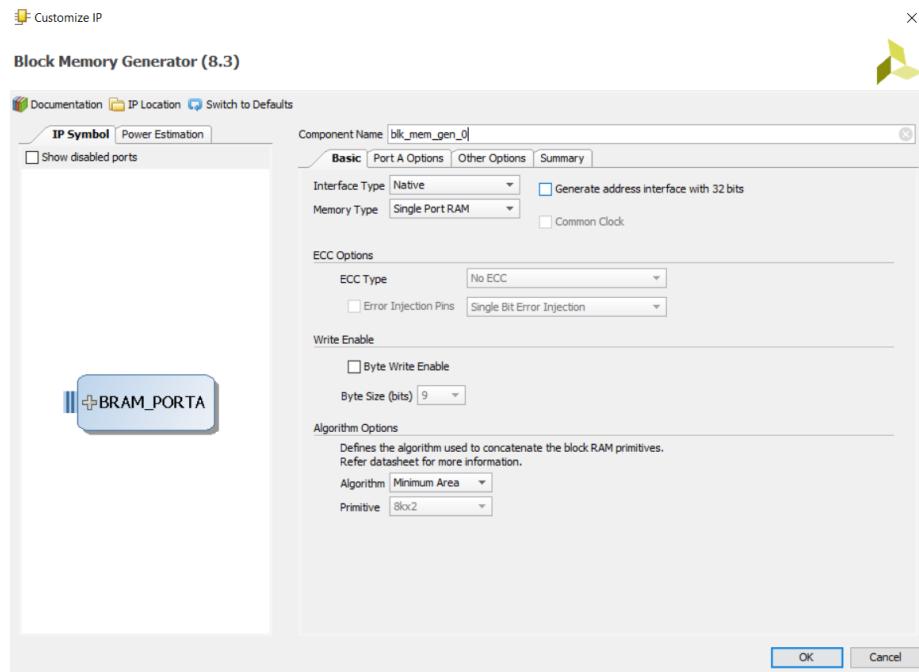


Figura 5.14: *Block Memory Generator*

Este asistente construye una memoria que utiliza memorias optimizadas para área y rendimiento usando recursos de bloques RAM integrados en FPGAs de Xilinx [10].

Este módulo tiene dos puertos independientes que acceden a un espacio de memoria compartida, teniendo ambos interfaz de lectura y escritura. En algunas arquitecturas FPGA de la serie *UltraScale*, cada una de las cuatro interfaces se pueden configurar con un ancho de datos diferente. En caso de no usar las cuatro, se puede seleccionar una configuración de memoria simplificada, por ejemplo, una memoria con un único puerto o una memoria con doble puerto, para reducir la utilización de recursos de la FPGA.

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.

Otra parte importante de este módulo es la selección del algoritmo usado para unir las primitivas del bloque RAM:

- **Algoritmo de área mínima:** La memoria se genera usando el mínimo número de primitivas de bloque RAM. 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 primitivas de bloque RAM durante la operación de lectura o escritura.
- **Algoritmo de primitiva fija:** La memoria se genera usando sólo un tipo de primitiva de bloque RAM.

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.

Está 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 desactivado, 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 inicializarla opcionalmente, 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.

### 5.3.3. Módulo de generación de reloj

(Tendrías que describir el componente IP configurable y cómo se particulariza para generar la señal de frecuencia deseada; similar a lo que se has hecho para el módulo de memoria)

### 5.3.4. Controlador VGA

(configurable?)

## 5.4. 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.

### 5.4.1. Computador básico

### 5.4.2. Visualización y movimiento de objetos en monitor VGA

#### Contador 4 bits

Para la primera toma de contacto se ha realizado este simple ejemplo con el que se pretende conocer los elementos de la tarjeta **Zybo** además de conocer las diferentes fases de flujo de diseño con Vivado.

Se ha comenzado creando un proyecto en Vivado con el siguiente código en *VHDL*. Para la verificación de su funcionalidad se hizo una simulación de comportamiento (Ver Figura 5.15).

```

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;
```

Para poder comprobar que la simulación funciona correctamente se necesita un testbench o bien añadir las señales manualmente durante la simulación.

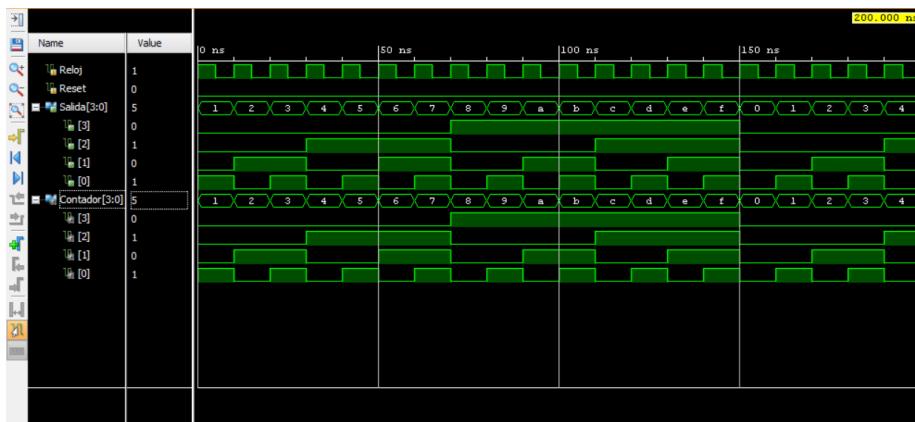


Figura 5.15: Simulación Contador 4 Bits

Si las salidas simuladas son correctas, se realiza la *Synthesis*, donde podemos ver netlist resultante (Ver Figura 5.16).

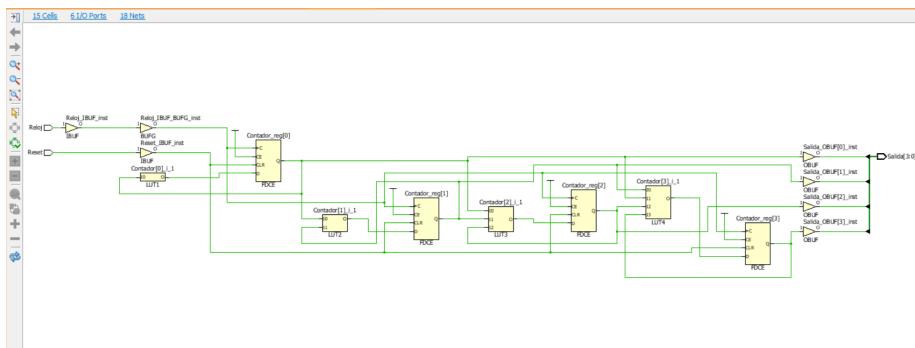


Figura 5.16: Netlist Resultante Synthesis

Si la compilación ha salido bien, ejecutamos la *Implementation*, y si no tenemos ningún error añadimos 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 y los LEDs.

Cuando esté todo hecho, se crea el 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 como el que se muestra a continuación.

```

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 Div_Frec is
7     Port(    Velocidad      : IN std_logic_vector(0 to 2);
8             Reloj          : IN  std_logic;
9             Salida         : OUT std_logic);

```

```

10    End Div_Frec;
11    -----
12    Architecture funcional of Div_Frec IS
13        Signal Contador : std_logic_vector(25 downto 0); --valores 0 a 67108863;
14        signal Salida_int : std_logic;
15        begin
16            process (Reloj)
17            begin
18                if reloj'event AND Reloj = '1' then
19                    Contador <= Contador + 1;
20                    Salida <= Salida_int;
21                end if;
22            end process;
23
24            Salida_int <= contador(25) WHEN velocidad = "000" ELSE
25                contador(24) WHEN velocidad = "001" ELSE
26                contador(23) WHEN velocidad = "010" ELSE
27                contador(22) WHEN velocidad = "011" ELSE
28                contador(21) WHEN velocidad = "100" ELSE
29                contador(20) WHEN velocidad = "101" ELSE
30                contador(19) WHEN velocidad = "110" ELSE
31                contador(18) WHEN velocidad = "111" ELSE '0';
32        end funcional;

```

Pero no basta sólo con añadir este archivo, necesitamos otro archivo que une estos dos anteriores.

```

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;

```

Ahora se vuelve a realizar lo mismo que antes 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.

### Procesador

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

Para ello primero tenemos que crear la 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, tenemos que simularlo para comprobar cómo funciona y si se ha configurado bien (Figura 5.17). Se puede observar que la salida “douta” muestra los primeros valores del archivo de inicialización siendo “addra” la dirección donde se encuentra cada uno de ellos.

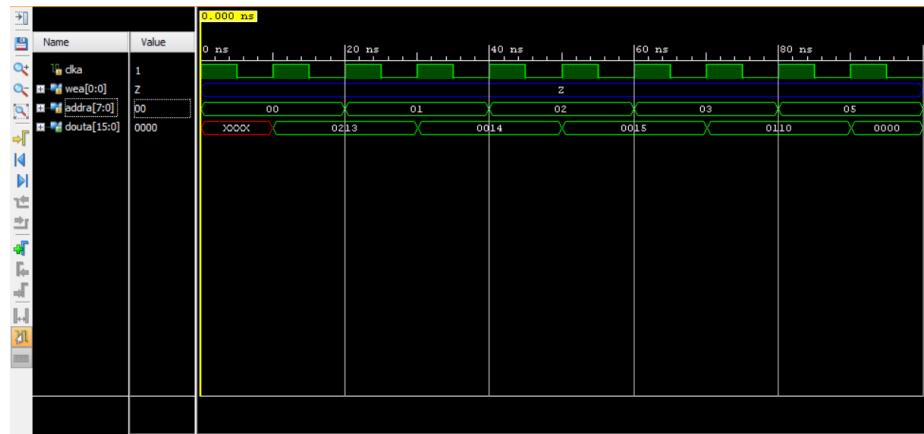


Figura 5.17: Simulación Memoria RAM

Como ya lo tenemos creado, incluimos los ficheros donde se encuentra la descripción del procesador, el fichero principal donde se crean los componentes del procesador y de la memoria y además las restricciones de la tarjeta.

```

1 library iee;
2 use iee.std_logic_1164.all;
3 use iee.std_logic_arith.all;
4 use iee.std_logic_signed.all;
5
6 entity procesador is
7     port( clock : in std_logic;
8           reset : in std_logic;
9           AC_out : out std_logic_vector(15 downto 0);
10          IR_out : out std_logic_vector(15 downto 0);
11          PC_out : out std_logic_vector(7 downto 0);
12          MEMq : in std_logic_vector(15 downto 0);
13          MEMdata: out std_logic_vector(15 downto 0);
14          MEMwe : out std_logic;
15          MEMadr : out std_logic_vector(7 downto 0);
16          IO_input : in std_logic_vector(7 downto 0);
17          IO_output : out std_logic_vector(7 downto 0)
18      );
19 end procesador;
20
21 architecture Behavioral of procesador is
22
23     TYPE STATE_TYPE IS ( reset_pc, fetch1, fetch0, decode, add2, add1, add0,
24                           load1, load0,
25                           store0, store1, jump, sub0,
26                           sub1, nand0, nand1,
27                           jneg, jpos, jzero, shr0,
28                           sh10, in1, out1);
29
30     SIGNAL state: STATE_TYPE;
31     SIGNAL IR, AC, RT: STD_LOGIC_VECTOR(15 DOWNTO 0 );
32     SIGNAL PC : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
33
34     BEGIN
35
36         -- Asignaciones a puertos de salida
37         --
38         AC_out <= AC;
39         IR_out <= IR;
40         PC_out <= PC;
41
42         FSMD: PROCESS ( CLOCK, RESET, state, PC, AC, IR )
43
44             begin
45                 if (RESET = '1') then
46                     state <= reset_pc;
47                 else
48                     if (CLOCK'event and CLOCK = '1') then
49                         if (state = fetch1) then
50                             if (IR = "0000000000000000") then
51                                 state <= fetch0;
52                             else
53                                 state <= decode;
54                             end if;
55                         elsif (state = fetch0) then
56                             if (IR = "0000000000000000") then
57                                 state <= add2;
58                             else
59                                 state <= add1;
60                             end if;
61                         elsif (state = add2) then
62                             if (IR = "0000000000000000") then
63                                 state <= add1;
64                             else
65                                 state <= add0;
66                             end if;
67                         elsif (state = add1) then
68                             if (IR = "0000000000000000") then
69                                 state <= add0;
70                             else
71                                 state <= load1;
72                             end if;
73                         elsif (state = add0) then
74                             if (IR = "0000000000000000") then
75                                 state <= load0;
76                             else
77                                 state <= store0;
78                             end if;
79                         elsif (state = load1) then
80                             if (IR = "0000000000000000") then
81                                 state <= store1;
82                             else
83                                 state <= jump;
84                             end if;
85                         elsif (state = store0) then
86                             if (IR = "0000000000000000") then
87                                 state <= jump;
88                             else
89                                 state <= sub0;
90                             end if;
91                         elsif (state = store1) then
92                             if (IR = "0000000000000000") then
93                                 state <= jump;
94                             else
95                                 state <= sub1;
96                             end if;
97                         elsif (state = jump) then
98                             if (IR = "0000000000000000") then
99                                 state <= nand0;
100                            else
101                                state <= nand1;
102                            end if;
103                        elsif (state = sub0) then
104                            if (IR = "0000000000000000") then
105                                state <= nand0;
106                            else
107                                state <= nand1;
108                            end if;
109                        elsif (state = sub1) then
110                            if (IR = "0000000000000000") then
111                                state <= nand0;
112                            else
113                                state <= nand1;
114                            end if;
115                        elsif (state = nand0) then
116                            if (IR = "0000000000000000") then
117                                state <= nand1;
118                            else
119                                state <= jneg;
120                            end if;
121                        elsif (state = nand1) then
122                            if (IR = "0000000000000000") then
123                                state <= jpos;
124                            else
125                                state <= jzero;
126                            end if;
127                        elsif (state = jneg) then
128                            if (IR = "0000000000000000") then
129                                state <= shr0;
130                            else
131                                state <= sh10;
132                            end if;
133                        elsif (state = jpos) then
134                            if (IR = "0000000000000000") then
135                                state <= shr0;
136                            else
137                                state <= in1;
138                            end if;
139                        elsif (state = jzero) then
140                            if (IR = "0000000000000000") then
141                                state <= shr0;
142                            else
143                                state <= out1;
144                            end if;
145                         end if;
146                     end if;
147                 end if;
148             end process;
149
150         end architecture;
151 
```

```

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

```

```

113           MEMwe <= '0';
114           MEMdata <= (others => '-');
115       end case;
116   END PROCESS;
118
119 end Behavioral;

```

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 [2]. 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	$PC \leftarrow address$

```

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

```

```

48      IR_out : out std_logic_vector(15 downto 0);
49      PC_out : out std_logic_vector(7 downto 0);
50      MEMq : in std_logic_vector(15 downto 0);
51      MEMdata: out std_logic_vector(15 downto 0);
52      MEMwe : out std_logic;
53      MEMadr : out std_logic_vector(7 downto 0);
54      IO_input : in std_logic_vector(7 downto 0);
55      IO_output : out std_logic_vector(7 downto 0)
56  );
57 end component;
58
59 begin
60
61 -- Instancia denominada MEM del IP core ram_256_x_16
62
63   MEM: blk_mem_gen_0        PORT MAP (
64     addra => MEMadr,
65     clka => reloj,
66     dina => MEMdata,
67     wea => MEMwe,
68     douta => MEMq
69 );
70
71 -- Instancia denominada PROC de la version inicial del procesador
72 --
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
96 end PROCESS;
97
98 end rtl;

```

Una vez añadidos estos ficheros, realizamos la simulación del conjunto (Figura 5.18). 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. Despues, 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. 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.

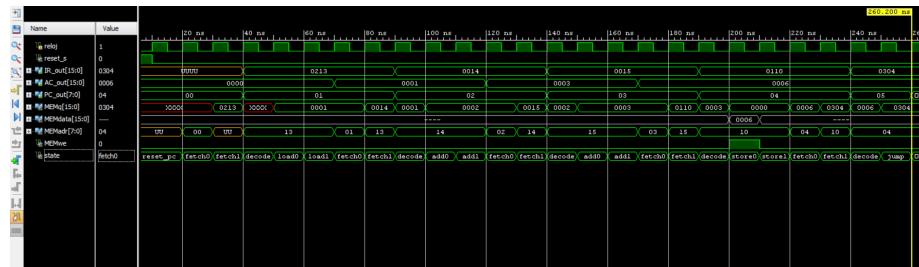


Figura 5.18: Simulación Procesador

### Controlador VGA

En esta práctica se pretende realizar una implementación de un juego interactivo 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 que se mueve en vertical rebotando con los bordes superior e inferior de la pantalla. Además hay que usar un módulo IP que genera un reloj de 25MHz a partir del reloj que proporciona 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 controlador_vga_640_x_480_60 IS
7 PORT(
8     clock_25          : IN      STD_LOGIC;
9     r,g,b            : IN      STD_LOGIC;
10    vga_r             : OUT     STD_LOGIC;
11    vga_g             : OUT     STD_LOGIC;
12    vga_b             : OUT     STD_LOGIC;
13    vga_blank_N       : OUT     STD_LOGIC;
14    vga_hs            : OUT     STD_LOGIC;
15    vga_vs            : OUT     STD_LOGIC;
16    vga_clk           : OUT     STD_LOGIC;
17    pixel_y           : OUT    STD_LOGIC_VECTOR(9 DOWNTO
18                                0);
18    pixel_x           : OUT    STD_LOGIC_VECTOR(9
19                                DOWNTO 0)
20 );
21
22 END controlador_vga_640_x_480_60;
23
24 ARCHITECTURE rtl OF controlador_vga_640_x_480_60 IS
25
26 -- Especificacione temporales VGA 640 x 480 pixels (60 Hz), 25M pixels/s
27 -- Sincronizacion horizontal (en numero de pixels/linea)
28   CONSTANT h_a: integer := 96; -- Retorno horizontal
29   CONSTANT h_b: integer := 48; -- "Back porch" horizontal (Margen
30                                izquierdo)
31   CONSTANT h_c: integer := 640; -- Area de visualizacion horizontal
32   CONSTANT h_d: integer := 16; -- "Front porch" horizontal (Margen
33                                derecho)
34   CONSTANT h_total : integer := h_a + h_b + h_c + h_d;
35
36 -- Sincronizacion Vertical (en numero de lineas/pantalla)
37   CONSTANT v_a: integer := 2; -- Retorno vertical
38   CONSTANT v_b: integer := 33; -- "Back porch" vertical
39   CONSTANT v_c: integer := 480; -- Area de visualizacion vertical
40   CONSTANT v_d: integer := 10; -- "Front porch" vertical
41   CONSTANT v_total : integer := v_a + v_b + v_c + v_d;

```

```

41      SIGNAL hs, vs : STD_LOGIC;
42      SIGNAL video_on : STD_LOGIC;
43      SIGNAL cont_hs, cont_vs :STD_LOGIC_VECTOR(9 DOWNTO 0);
44
45 BEGIN
46
47 -- La senial vga_clk que va al DAC coincide con el reloj de 25MHz.
48 vga_clk <= clock_25;
49
50 PROCESS
51 BEGIN
52   WAIT UNTIL(clock_25'EVENT) AND (clock_25='1');
53
54   -- Se generan las seniales de sincronizacion horizontal y vertical
55   -- a partir de los contadores cont_hs y cont_vs
56
57   -- El contador cont_hs cuenta los pixels/fila
58   -- La senial de sincronizacion horizontal (hs) vale cero durante el
59   -- retorno horizontal
60
61   --
62   IF (cont_hs = h_total - 1) THEN
63     cont_hs <= "0000000000";
64   ELSE
65     cont_hs <= cont_hs + 1;
66   END IF;
67
68   IF (cont_hs <= h_c+h_d+h_a-1) AND (cont_hs >= h_c+h_d) THEN
69     hs <= '0';
70   ELSE
71     hs <= '1';
72   END IF;
73
74   -- El contador cont_vs cuenta las filas/pantalla
75   -- La senial de sincronizacion vertical (vs) vale cero durante el retorno
76   -- vertical
77
78   IF (cont_vs = v_total - 1) AND (cont_hs >= h_total - h_a - h_d) THEN
79     cont_vs <= "0000000000";
80   ELSIF (cont_hs = h_total - h_a - h_d) THEN
81     cont_vs <= cont_vs + 1;
82   END IF;
83
84   IF (cont_vs <= v_c+v_d+v_a-1) AND (cont_vs >= v_c+v_d) THEN
85     vs <= '0';
86   ELSE
87     vs <= '1';
88   END IF;
89
90   -- Se generan seniales para informar a la salida de la coordenada de
91   -- pixel a visualizar
92   IF (cont_hs <= 639) THEN pixel_x <= cont_hs;    END IF;
93   IF (cont_vs <= 479) THEN pixel_y <= cont_vs; end if;
94
95   -- La senial video_on esta en alta cuando se transmite informacion de
96   -- video
97   -- (para valores de cont_vs entre 0 y 479, y valores de cont_hs entre 0
98   -- y 639)
99
100  if (cont_hs <= 639) and (cont_vs <= 479) then video_on <= '1'; else
101    video_on <= '0'; end if;
102
103  -- Se registran todas las seniales de video para eliminar retardos que
104  -- puedan emborrifar la imagen
105  vga_r <= r AND video_on;
106  vga_g <= g AND video_on;
107  vga_b <= b AND video_on;
108  vga_hs      <= hs;
109  vga_vs      <= vs;
110  vga_blank_n <= video_on;
111
112 END PROCESS;

```

```
108 |     END rtl;
```

Este código del controlador VGA 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.

La lógica interna de la mayoría de las placas usan un reloj de 25MHz generado por un PLL, módulo IP que se generará desde el catálogo IP. 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 [2].

Tras conocer el controlador VGA, hay que crear el módulo PLL. Para ello sólo tenemos que entrar en el catálogo IP de Vivado (*FPGA Features and Design → Clocking → Clocking Wizard*) y configurándolo de manera que se seleccione la opción PLL y en la salida del reloj poner la frecuencia deseada, en este caso, 25MHz.

*Clocking Wizard* 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**) o un bucle de bloqueo de fase (**PLL**). También ayuda a verificar la frecuencia de reloj generada por la salida en la simulación [11].

Por último se agrega la descripción de la bola, en la que además de la bola, se especifican las palas y una red para la finalización del juego de un modo más sencillo.

```

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 LIBRARY lpm;
6 USE lpm.lpm_components.ALL;
7
8 ENTITY bola IS
9 PORT(
10     Red,Green,Blue : OUT std_logic;
11     vs : IN std_logic;
12     pixel_Y, pixel_X : IN std_logic_vector(9 downto 0);
13     up1, down1, up2, down2 : IN std_logic
14 );
15 END bola;
16
17 architecture funcional of bola is
18     -- Señales para el tamaño de la bola y su desplazamiento
19     SIGNAL Bola_on : STD_LOGIC;      -- "Dibujar" bola
20     SIGNAL Desplaza_Bola_Y: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
21         Desplazamiento en pixeles de la bola en y
22     SIGNAL Desplaza_Bola_X: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
23         Desplazamiento en pixeles de la bola en x
24     SIGNAL Bola_Y : STD_LOGIC_VECTOR(9 DOWNTO 0);  -- "Eje" y de la bola
25     SIGNAL Bola_X : STD_LOGIC_VECTOR(9 DOWNTO 0);  -- "Eje" x de la bola
26
27     -- Señales para el tamaño de la pala izquierda y su desplazamiento
28     -- vertical
29     SIGNAL Pala_izq_on : STD_LOGIC; -- "Dibujar" pala izquierda
30     SIGNAL Desplaza_Pala_izq_Y: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
31         Desplazamiento en pixeles de la pala izquierda en y
32     SIGNAL Desplaza_Pala_izq_X: STD_LOGIC_VECTOR(9 DOWNTO 0);    --
33         Desplazamiento en pixeles de la pala izquierda en x

```

```

29      SIGNAL Pala_izq_Y : STD_LOGIC_VECTOR(9 DOWNTO 0);          -- "Eje" y de
30          la pala izquierda
31
32      -- Señales para el tamaño de la pala derecha y su desplazamiento
33          vertical
34      SIGNAL Pala_der_on : STD_LOGIC; -- "Dibujar" pala derecha
35      SIGNAL Desplaza_Pala_der_Y: STD_LOGIC_VECTOR(9 DOWNTO 0);      --
36          Desplazamiento en pixeles de la pala derecha en y
37      SIGNAL Desplaza_Pala_der_X: STD_LOGIC_VECTOR(9 DOWNTO 0);      --
38          Desplazamiento en pixeles de la pala derecha en x
39      SIGNAL Pala_der_Y : STD_LOGIC_VECTOR(9 DOWNTO 0);          -- "Eje" y de
40          la pala derecha
41
42      -- Señales para la red del campo
43      SIGNAL Red_on : STD_LOGIC;
44      SIGNAL Red_y : STD_LOGIC_VECTOR(9 DOWNTO 0);
45      CONSTANT Red_x : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(320,10);
46      CONSTANT Size_red_y : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(479,10);
47      CONSTANT Size_red_x : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(1,10);
48
49      CONSTANT Size_X : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(4,10);    -- Tamaño del objeto en el eje x
50      CONSTANT Size_Y : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(40,10);   -- Tamaño del objeto en el eje y
51      CONSTANT Pala_izq_X : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(40,10);   -- "Eje" x de la pala derecha
52          , constante
53      CONSTANT Pala_der_X : STD_LOGIC_VECTOR(9 DOWNTO 0) :=           CONV_STD_LOGIC_VECTOR(600,10);  -- "Eje" x de la pala
54          izquierda, constante
55
56      BEGIN
57
58          -- Pelota gris, pala izquierda roja y pala derecha azul
59          Red <= Bola_on OR Pala_der_on;
60          Green        <= Bola_on OR Red_on;
61          Blue         <= Bola_on OR Pala_izq_on OR Red_on;
62
63          Dibujar_Bola: Process (Bola_Y, pixel_X, pixel_Y)
64          BEGIN
65              -- Chequear coordenadas X e Y para identificar el área de la bola
66              -- Poner Bola_on a '1' para visualizar la bola
67              IF (Bola_X <= pixel_X + Size_X) AND
68                  (Bola_X + Size_X >= pixel_X) AND
69                  (Bola_Y <= pixel_Y + Size_Y) AND
70                  (Bola_Y + Size_Y >= pixel_Y ) THEN
71
72                  Bola_on <= '1';
73              ELSE
74                  Bola_on <= '0';
75              END IF;
76
77          END process Dibujar_Bola;
78
79          Dibujar_Pala_Izq: Process (Pala_izq_Y, pixel_X, pixel_Y)
80          BEGIN
81              -- Chequear coordenadas X e Y para identificar el área de la pala
82              -- Poner Pala_on a '1' para visualizar la pala
83              IF (Pala_izq_X <= pixel_X + Size_X) AND
84                  (Pala_izq_X + Size_X >= pixel_X) AND
85                  (Pala_izq_Y <= pixel_Y + Size_Y) AND
86                  (Pala_izq_Y + Size_Y >= pixel_Y ) THEN
87
88                  Pala_izq_on <= '1';
89              ELSE
90                  Pala_izq_on <= '0';
91              END IF;
92          END process Dibujar_Pala_Izq;
93
94          Dibujar_Pala_Der: Process (Pala_der_Y, pixel_X, pixel_Y)
95          BEGIN

```

```

89      -- Chequear coordenadas X e Y para identificar el area de la pala
90      -- Poner Pala_on a '1' para visualizar la pala
91      IF (Pala_der_X <= pixel_X + Size_X) AND
92          (Pala_der_X + Size_X >= pixel_X) AND
93          (Pala_der_Y <= pixel_Y + Size_Y) AND
94          (Pala_der_Y + Size_Y >= pixel_Y ) THEN
95
96          Pala_der_on <= '1';
97      ELSE
98          Pala_der_on <= '0';
99      END IF;
100     END process Dibujar_Pala_Der;
101
102    Dibujar_Red: Process(Red_Y, pixel_X, pixel_Y)
103    BEGIN
104        IF (Red_X <= pixel_X + Size_red_X) AND
105            (Red_X + Size_red_X >= pixel_X) AND
106            (Red_Y <= pixel_Y + Size_red_Y) AND
107            (Red_Y + Size_red_Y >= pixel_Y ) THEN
108
109            Red_on <= '1';
110        ELSE
111            Red_on <= '0';
112        END IF;
113    END Process Dibujar_Red;
114
115    Mover_Bola: PROCESS (vs)
116    BEGIN
117        -- Actualizar la posicion de la bola en cada refresco de pantalla
118        IF vs'event and vs = '1' THEN
119            -- Detectar los bordes superior e inferior de la pantalla
120            IF Bola_Y >= CONV_STD_LOGIC_VECTOR(479,10) - Size_X THEN
121                Desplaza_Bola_Y <= CONV_STD_LOGIC_VECTOR(-2,10);
122            ELSIF Bola_Y <= Size_X THEN
123                Desplaza_Bola_Y <= CONV_STD_LOGIC_VECTOR(2,10);
124            END IF;
125            -- Calcular la siguiente posicion de la bola
126            Bola_Y           <= Bola_Y + Desplaza_Bola_Y;
127
128            --Detectar los bordes derecho e izquierdo de la pantalla
129            IF Bola_X >= CONV_STD_LOGIC_VECTOR(639,10) - Size_X THEN
130
131                Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR(-2,10);
132
133            ELSIF Bola_X <= Size_X THEN
134
135                Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR(2,10);
136            END IF;
137            --Calcular la siguiente posicion de la bola
138            Bola_X           <= Bola_X + Desplaza_Bola_X;
139
140            -- Rebote por la derecha, pala izquierda
141            IF (Bola_Y + Size_X + Size_Y >= Pala_izq_Y) AND
142                (Bola_Y <= Pala_izq_Y + Size_Y + Size_X) AND
143                (Bola_X <= Pala_izq_X + Size_X + Size_X) THEN
144                Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR(2,10);
145            END IF;
146
147            -- Rebote por la izquierda, pala derecha
148            IF (Bola_Y + Size_X + Size_Y >= Pala_der_Y) AND
149                (Bola_Y <= Pala_der_Y + Size_Y + Size_X) AND
150                (Bola_X + Size_X + Size_X >= Pala_der_X) THEN
151                Desplaza_Bola_X <= CONV_STD_LOGIC_VECTOR(-2,10);
152            END IF;
153
154    Mover_Pala_Izq: PROCESS (vs)
155    BEGIN
156        -- Actualizar la posicion de la bola en cada refresco de pantalla
157        IF vs'event and vs = '1' THEN
158            -- Detectar los bordes superior e inferior de la pantalla

```

```

159      IF (down1 = '0') AND (Pala_izq_Y <= CONV_STD_LOGIC_VECTOR
160          (479,10) - Size_Y) THEN
161          Desplaza_Pala_izq_Y <= CONV_STD_LOGIC_VECTOR(2,10);
162      ELSIF (up1 = '0') AND (Pala_izq_Y >= Size_Y) THEN
163          Desplaza_Pala_izq_Y <= CONV_STD_LOGIC_VECTOR(-2,10);
164      ELSE
165          Desplaza_Pala_izq_Y <= CONV_STD_LOGIC_VECTOR(0,10);
166      END IF;
167      -- Calcular la siguiente posicion de la bola
168      Pala_izq_Y <= Pala_izq_Y + Desplaza_Pala_izq_Y;
169  END IF;
170 END process Mover_Pala_Izq;
171
172 Mover_Pala_Der: PROCESS (vs)
173 BEGIN
174     -- Actualizar la posicion de la bola en cada refresco de pantalla
175     IF vs'event and vs = '1' THEN
176         -- Detectar los bordes superior e inferior de la pantalla
177         IF (down2 = '0') AND (Pala_der_Y <= CONV_STD_LOGIC_VECTOR
178             (479,10) - Size_Y) THEN
179             Desplaza_Pala_der_Y <= CONV_STD_LOGIC_VECTOR(2,10);
180         ELSIF (up2 = '0') AND (Pala_der_Y >= Size_Y) THEN
181             Desplaza_Pala_der_Y <= CONV_STD_LOGIC_VECTOR(-2,10);
182         ELSE
183             Desplaza_Pala_der_Y <= CONV_STD_LOGIC_VECTOR(0,10);
184         END IF;
185
186         -- Calcular la siguiente posicion de la bola
187         Pala_der_Y <= Pala_der_Y + Desplaza_Pala_der_Y;
188     END IF;
189 END process Mover_Pala_Der;
190
191 Mover_Red: PROCESS (vs)
192 BEGIN
193     IF vs'event and vs = '1' THEN
194         Red_Y <= CONV_STD_LOGIC_VECTOR(0,10);
195     END IF;
196 END PROCESS Mover_Red;
197
198 -- Los botones estan a la alta siempre que no esten pulsados
199
200 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. Size\_x es el tamaño de la bola cuadrada. El proceso Mover\_Bola, mueve la bola unos pocos píxeles 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 píxeles que la bola se tiene que mover tanto horizontal como verticalmente. Y para las palas y la red se ha realizado el mismo procedimiento. Las señales up1, up2, down1, down2 están conectadas a los pines que corresponden con los botones para hacer que las palas suben y bajen cuando se pulse alguno de ellos.

Una vez entendidos todos los ficheros, 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 en la pantalla el sencillo juego y además hacer que las palas se muevan para poder probarlo.



## **Capítulo 6**

# **Conclusiones y vías futuras**



# Bibliografía

- [1] Digilent. Zybo fpga board reference manual. [https://reference.digilentinc.com/\\_media/zybo:zybo\\_rm.pdf](https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf).
- [2] James O Hamblen, Tyson S Hall, and Michael D Furman. *Rapid prototyping of digital systems: SOPC edition*. Springer Science & Business Media, 2007.
- [3] 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>.
- [4] Clive Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.
- [5] Clive Maxfield. *FPGAs: instant access*. Elsevier, 2011.
- [6] Gina Smith. *FPGAs 101: Everything you need to know to get started*. Newnes, 2010.
- [7] 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.
- [8] L Terés, Y Torroja, S Olcoz, and E Villar. Vhdl lenguaje estándar de diseño electrónico. editorial mc graw hill, 1997.
- [9] Wikipedia. Xilinx. <https://en.wikipedia.org/wiki/Xilinx>.
- [10] 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](https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf).
- [11] 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](https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf).
- [12] Xilinx. Field programmable gate array (fpga). <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>.
- [13] Xilinx. Vivado design suite user guide : Designing with ip. [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2019\\_2/ug896-vivado-ip.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug896-vivado-ip.pdf).

- [14] Xilinx. Vivado design suite user guide : Using the vivado ide. [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2016\\_2/ug893-vivado-ide.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug893-vivado-ide.pdf).
- [15] Xilinx. Xup students. <https://www.xilinx.com/support/university/students.html#overview>.