Universidad Autónoma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Proyecto Terminal de Ingeniería en Computación

Sistema para la Programación de un FPGA Mediante Interfaz Gráfica de Usuario en Linux

Proyecto que presenta:
Usiel Alvarado Villafranca

para obtener el titulo de:
Ingeniero en Computación

Asesor de Proyecto:
M. en C. Oscar Alvarado Nava

México, D.F. **Septiembre de 2010**

Resumen

El estudio de las nuevas tecnologías es indispensable hoy en día en la mayoría de las carreras profesionales del área de la ingeniería, pues algunas de ellas tienen un amplio rango de aplicaciones para la industria y para la investigación, abren nuevos caminos para el desarrollo de nuevos conocimientos y tecnologías, la utilidad de dichos estudios, no sólo se verá reflejada en los conocimientos adquiridos por los estudiantes, sino en el constante desarrollo de sus capacidades y experiencias que el manejo de las herramientas novedosas les han dejado, y así, les permiten desenvolverse mejor en el área a la que se dediquen . Por esta razón, para la ingeniería en computación se ha vuelto una parte importante el conocimiento de las tecnologías de dispositivos lógicos programables. En ellos se pueden definir circuitos con funcionalidades muy variadas, es decir hardware que, dependiendo de su complejidad, se puede llegar a crear, desde la implementación de un simple circuito lógico, hasta todo un sistema operativo funcionando en un sólo chip. En el presente reporte se presenta el desarrollo de un sistema con interfaz gráfica para el sistema operativo Linux que se encarga de programar y permitir el uso de uno de estos dispositivos, con el objetivo de facilitar al usuario algunas tareas que, de otro modo le resultarían costosas en tiempo y estudio.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivaciones

Hoy en día, gracias al uso de los dispositivos lógicos programables, las posibilidades para la ingeniería en computación y otras ramas similares, son muy amplias, pues, existen algoritmos de complejidad tan alta que, implementados en un software, se convierten en una tarea altamente costosa en tiempo, y que por esta razón, no resulta conveniente realizar. En cambio, un algoritmo implementado en un hardware, resulta mucho más eficiente, pues reduce dramáticamente dicho tiempo de procesamiento dada la manera en la que los datos son manejados.

Existe un amplio rango de dispositivos lógicos programables, dependiendo de su capacidad, su complejidad y el uso que se le da, desde las memorias programables utilizadas en aparatos electrónicos de uso común, hasta complejos dispositivos utilizados en conjunto con las computadoras más avanzadas de los más importantes laboratorios y centros de investigación. El uso de estas tecnologías está presentes en casi todo aparato que podamos imaginar, tanto implementadas como utilizadas en la elaboración de prototipos, dada su capacidad de reconfiguración.

El FPGA[1], es una de estas tecnologías y, dado que es un tipo de dispositivo de alto nivel en el rango de complejidad de los dispositivos lógicos programables, su utilidad es muy grande en el campo de la ingeniería, en la realización de prototipos y pruebas. Sin embargo no es una tarea tan sencilla en algunos de estos dispositivos, pues no siempre se cuenta con una herramienta con licencia libre de uso para llevar a cabo su programación y uso de una manera sencilla, dado su enorme rango de posibilidades y el uso al que están destinados, pero que sin embargo, vale la pena utilizar. Tal es el caso de la tarjeta *ADM-XP*, la cual, tiene montado un FPGA Virtex II Pro, esta requiere de una API de programación proporcionada por el fabricante. El uso de estas funciones implican la codificación de un programa, además del trabajo realizado en el diseño del hardware, lo cual, da lugar a una tarea costosa en tiempo para aquellos que aún están iniciando en el diseño e implementación de hardware en FPGA.

[1] Arreglo de compuertas programables en campo *(Field programable gate array).*

Por dicho motivo se requiere de una herramienta que ayude a facilitar algunas tareas en en el proceso de implementación de hardware en la tarjeta *ADM-XP*, la cual es una de las diferentes tarjetas con las que se cuenta en el área de Sistemas Digitales de la Universidad Autónoma Metropolitana unidad Azcapotzalco, y que resulta útil para la realización de prácticas y nuevos proyectos.

1.2. Objetivos

Este reporte presenta como principal objetivo, la programación de la tarjeta *ADM-XP* mediante un sistema que incluye una interfaz gráfica, y pretende ayudar a obtener los conocimientos básicos necesarios para el diseño e implementación de hardware y el desarrollo de algunas funcionalidades específicas en dicha tarjeta, permitiendo al usuario abstraerse en primera instancia de detalles sobre el funcionamiento de la comunicación entre el FPGA y la computadora (el *host*) y contribuyendo al aprendizaje de las tecnologías de dispositivos lógicos programables y la creación de nuevos proyectos de ingeniería.

Para alcanzar dicho objetivo, fue necesario alcanzar los siguientes objetivos previos:

- **Análisis de la comunicación con el FPGA**. Identificar las bases en las que se realiza la comunicación entre el FPGA y el *host* al que se encuentra conectada. Para cumplir con este objetivo fue necesario estudiar el protocolo que utiliza el FPGA para comunicarse con el *host* mediante el puerto PCI, se reconocieron diversas formas de comunicación y se eligió un para llevar a cabo el proyecto.

- **Creación de un proyecto para la *ADM-XP***. Fue necesario entender la manera en la que se debe implementar un proyecto para generar el archivo de configuración para programar el FPGA, esto se hizo mediante la herramienta *Xilinx ISE Design Suite*, la cual, permite describir, simular e implementar un diseño en cuestión para gran cantidad de familias de dispositivos FPGA. Gracias a esta herramienta se puede generar un archivo de programación, con el cual es implementado un diseño en el FPGA para obtener la funcionalidad deseada, sin embargo, en el caso de la *ADM-XP* no es posible aplicar de forma directa. Por lo que la tarea de crear el archivo de programación (*bitstream*), requiere del conocimiento del protocolo de comunicación antes mencionado.

- **Programación de las funciones de configuración y operación del FPGA.** La transmisión del *bitstream* al FPGA se puede realizar mediante la aplicación de un API de programación en lenguaje C, proporcionado por la compañía *Aplha-Data*, fabricante de la *ADM-XP*. En este proyecto fue necesario reconocer las funciones que se encargan de realizar esta operación y las destinadas a la operación y configuración del FPGA, además de aquellas que sirven para obtener datos importantes sobre el FPGA. Con dichas funciones

se codificó una aplicación para consola usando lenguaje C++ el cual es una ampliación de C y proporciona una forma más cómoda de programar.

- **Diseño de una interfaz gráfica de usuario en QT.** La biblioteca QT, cuenta con amplias posibilidades para desarrollar interfaces gráficas de usuario, es una buena opción para realizar aplicaciones veloces y además portables, pues utiliza el lenguaje de programación C++, por lo que fue estudiada para desarrollar la interfaz gráfica que permita al usuario realizar las operaciones básicas necesarias para programar y utilizar el FPGA.

De esta manera, se obtuvo una herramienta que permite al usuario implementar un diseño directamente en el FPGA sin pasar por el proceso de realizar otro programa tan sólo para configurar la tarjeta y utilizarla, ahorrando tiempo importante si el único objetivo era el de comprobar el funcionamiento de un diseño en específico.

Capítulo 2

FPGA y tarjetas de desarrollo

2.1. Conceptos sobre FPGA

El FPGA es un dispositivo lógico programable, compuesto por un orden de cientos de miles hasta millones de bloques lógicos configurables (CLB)[1], conformados por compuertas lógicas, los cuales, mediante conectores reconfigurables entre ellos pueden ser interconectados para obtener una funcionalidad determinada, prácticamente cualquier función lógica que se desee. Como uno de los más versátiles componentes de la tecnología VLSI[2], permite implementar complejos dispositivos electrónicos diseñados mediante lenguajes de descripción de hardware como VHDL[3] o Verilog [2], además muchos de estos también contienen dentro de sí, diversos elementos como unidades de memoria que van desde simples flip-flops hasta bloques de memoria dinÃ¡mica más complejos. También pueden contener multiplicadores, comparadores e incluso procesadores para formar sistemas programables en Chip [3][4].

Sus aplicaciones pueden alcanzar ámbitos como la criptografía, la bio-informática, la radioastronomía, el procesamiento digital de señales, reconocimiento de voz y un creciente rango en otras áreas en las que el cómputo requiere de un procesamiento eficiente. Originalmente, eran competidores de los llamados dispositivos lógicos programables complejos (CPLD), Conforme su tamaño y capacidades fueron aumentando, comenzaron a tomar funciones cada vez más grandes hasta el grado de que algunos de ellos actualmente cuentan con uno o más procesadores empotrados, permitiendo con ello la creación de sistemas completos dentro de un sólo chip (System on Chip, SoC), y permiten, entre otras cosas, la aceleración de aplicaciones por medio del trabajo en conjunto de un microprocesador y hardware dedicado programado [5].

El FPGA con el que se trabajó en este proyecto es un XC2VP100-FF1704 de la familia Virtex II Pro[6], de la compañía *Xilinx*, el número 100 en el modelo del FPGA

[1] *Configurable logic block.*

[2] Integración de escala muy grande (*Very large scale integrated*).

[3] Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad (*VHSIC hardware description language*).

indica que tiene cerca de 100,000 celdas lógicas, que son la parte más pequeña que conforma a un FPGA, un CLB está formado por cuatro de estas celdas. Como se dijo anteriormente, la interconexión de estos elementos generarán la funcionalidad que se desee. Como se puede observar, esta cantidad de componentes proporcionan una infinidad de posibilidades, sin embargo, el Virtex II Pro, es actualmente un dispositivo de capacidades limitadas a comparación de un XC6VLX760 de la familia Virtex 6 que sobrepasa las 700,000 celdas lógicas. A pesar de ello, considerando que es un dispositivo creado a principios de esta década, es ya una opción para programar incluso sistemas en un sólo chip.

Además de la cantidad de celdas lógicas, las siguientes características son algunas de las principales:

- Dos procesadores PowerPC 405 (PPC405) a 300Mhz con 16kB para datos y 16kB para instrucciones en su memoria cache.
- 8 Mb de bloques RAM de doble vía
- 20 Entradas-Salidas de alta velocidad (RocketIO), con una velocidad máxima de operación de 3.125 Gbps .
- 444 multiplicadores de 18 x 18 bits
- 12 relojes para control de bloques
- 1040 puertos de entrada-salida de usuario

Estas características proporcionan un gran número de posibilidades para el desarrollo de aplicaciones y, puesto que existen hoy en día herramientas de software que ayudan a la programación, compilación, síntesis, simulación y depuración tanto de hardware como de software, se obtiene una alta flexibilidad de desarrollo, permitiendo a los usuarios centrarse en el diseño y tomando la responsabilidad de dicho diseño para obtener el máximo provecho de los recursos.

2.2. Tarjetas de desarrollo

A un FPGA, normalmente se le puede encontrar incrustado en diversas tarjetas de desarrollo que incluyen componentes extra (memorias, puertos, controles) que le ayudarán a extender sus posibilidades para obtener un hardware tan complejo como dicha tarjeta y el FPGA lo permitan. Dichos componentes conforman una plataforma de trabajo que, debido a su arquitectura interna y método de programación, representan una solución muy confiable para la síntesis de hardware y creación de nuevos prototipos y diseños.

Dada la variedad de tarjetas en las que se puede encontrar a un FPGA, se han desarrollado diferentes herramientas y métodos para su configuración y operación, dependiendo de los puertos y conexiones con los que cuenta cada una, además, el

uso de algunas de ellas se ha vuelto muy popular en universidades gracias a sus características y utilidades.

Como ejemplo de estas tarjetas, tenemos la XUP2VP, fabricada por Digilent Inc. la cual cuenta con un FPGA XC2VP30-FF896 también de la familia Virtex II Pro. Es una tarjeta con un amplio set de componentes periféricos que pueden ser usados para crear sistemas complejos y demostrar las capacidades del FPGA, tal como se muestra en la Figura 2.1, en la que podemos observar que incluso contiene algunos switches y LEDS para implementar aplicaciones sencillas hasta puertos tan avanzados como salidas de audio y video, notablemente se pueden hacer muchas aplicaciones distintas en una misma plataforma, una forma sencilla en la que se pueden implementar aplicaciones que trabajen en conjunto con una computadora es por medio del puerto serial, aunque no es un medio muy veloz, proporciona facilidad de programación y uso gracias a la ayuda de las herramientas de *Xilinx*, pues contienen los módulos necesarios para crear un sistema en un sólo chip (*Xilinx* EDK), integrándolos y generando automáticamente el archivo de configuración, el cual es necesario para programar el FPGA (*Xilinx* iMPACT). En este proceso, el usuario sólo tiene que seguir un asistente que le ayuda a preparar los detalles de configuración sin necesidad de saber la forma en que debe conectar dichos módulos, sólo centrarse en la funcionalidad que requiere. Después sólo necesita indicar el modelo de la tarjeta y el puerto por el que se realizará la transferencia del *bitstream*, en su caso, el puerto USB. Cabe mencionar que este puerto, a pesar de ser mucho más veloz que el puerto serial, está destinado únicamente para la configuración del FPGA y no puede ser utilizado para la comunicación con el *host*, sin embargo, existen muchas otras tarjetas que cumplen con esta característica, pues, como se mencionó con anterioridad, las características de una tarjeta de desarrollo dependen del uso para el cual están diseñadas[7].

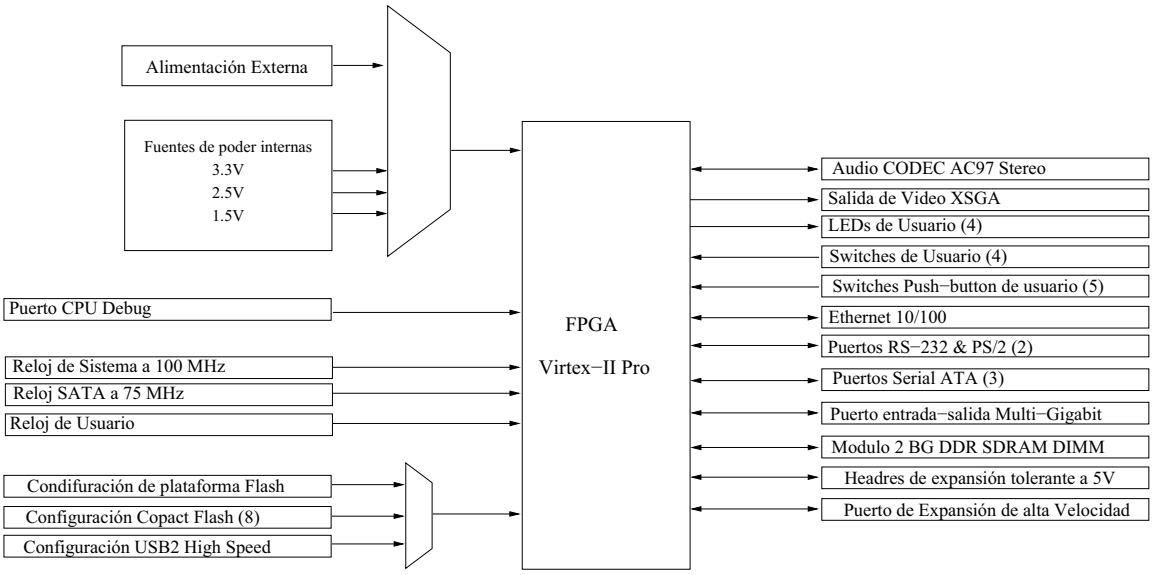

Figura 2.1: Diagrama a bloques de la tarjeta XUP2VP.

Un ejemplo de tarjeta de desarrollo que se comunica con el *host* vía USB es la tarjeta EZ1CUSB, fabricada por ALTERA, otra compañía líder en la tecnología FPGA. Es un kit de bajo costo para aplicaciones basadas en la familia Cyclone FPGA y el controlador USB FTDI FT2232C[4].

2.3. La tarjeta de desarrollo ADM-XP

Hasta ahora se ha descrito la forma en la que operan algunas tarjetas de desarrollo con el objetivo de poner en contraste a la tarjeta que es objeto de este proyecto y resaltar la importancia del mismo, a continuación se describirá a la tarjeta *ADM-XP* cuyo FPGA incrustado es el XC2VP100-FF1704.

Fabricada por *Aplha-Data*, la tarjeta ADMXRC II Pro (*ADM-XP*), es un dispositivo que contiene un FPGA Virtex II Pro, el cual está diseñado para el desarrollo de aplicaciones que requieren de un alto procesamiento y su característica más sobresaliente es que se trata de una tarjeta PMC[5], es un tipo de tarjeta que presenta físicamente un conector destinado a funcionar en el bus PCI, y mediante un puente PMC, puede ser conectada al bus PCI de una computadora. La imágen de la Figura 2.2 muestra a esta tarjeta en la que se apreceia en el centro al FPGA virtex II Pro. Es una importante diferencia con otras tarjetas de desarrollo como la XUP2VP, principalmente por que las tareas, tanto de configuración como de comunicación y operación se pueden realizar por medio de este bus, dando la ventaja de una mayor velocidad de transferencia en los datos y obteniendo como resultado una mayor eficiencia en las aplicaciones, por estas razones, el uso de este tipo de tarjetas es muy recurrente en aplicaciones de procesamiento en paralelo.

Figura 2.2: La tarjeta ADM-XP y su FPGA incrustado[8].

[4]Controlador producido por la compañía Future Technology Devices International (FTDI), especializada en tecnologías USB.

[5]*PCI Mezzanine Card*, es un estándar para dar compatibilidad con el bus PCI pero en una presentación más pequeña y robusta que las tarjetas PCI estándar, un puente PMC puede contener hasta dos tarjetas de este tipo.

En el diagrama a bloques de la Figura 2.3 podemos observar los elementos que conforman a la *ADM-XP*, principalmente se pueden apreciar los seis bancos de memoria que puede utilizar el FPGA y el puente de bus local que se encarga de la comunicación con el Bus PCI, además de que tiene todo un conjunto de pines de entrada-salida y un bus MGT[6] a los que se les puede conectar distintos periféricos y aumentar las posibilidades en los diseños.

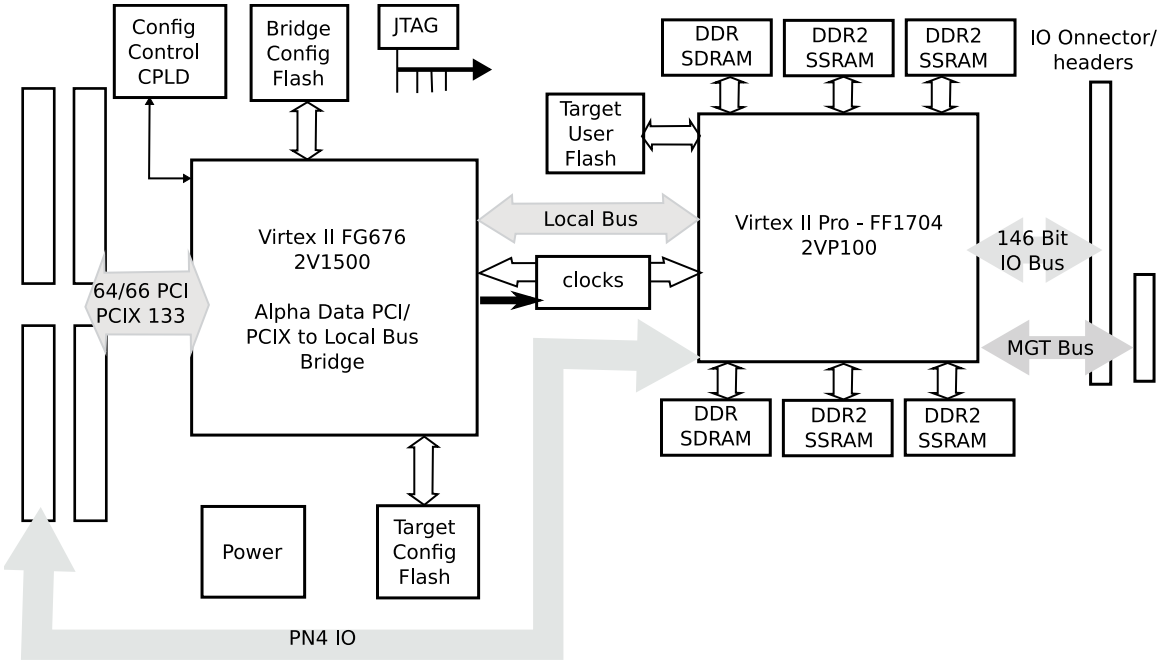

Figura 2.3: Diagrama a bloques de la tarjeta ADM-XP.

2.3.1. Especificaciones

- PCI de alto *performance* y bus local asíncrono.
- Velocidades de Bus local de hasta 80MHz.
- Cuatro bancos de 256K * 64 bits de DDR2 SSRAM
- Dos bancos de 64MB DDR SDRAM.
- Dos memorias flash de 16MB para programación de dispositivos.
- Reloj de usuario programable entre 5MHz y 200MHz.
- Adaptador de panel frontal de usuario con 146 señales de entrada-salida.

[6] *Multi-Gigabit Transceiver*, un puerto MGT incluye un dispositivo que serializa y deserializa los datos para transmitirlos a altas velocidades, arriba de 1Gbps (Giga bits por segundo)

- Soporta PCI 3.3V o PCIX a 64 bits.
- Oscilador LVPECL de 125 MHz.
- Ocho conexiones *RocketIO* MGT [9].

La Compañía Alpha-Data produce una variedad de este tipo de tarjetas dedicadas al cómputo intensivo, tratando de proporcionar productos interoperables con los estándares de la industria, tales como PMC, PCI, CompactPCI, PCI Express y VME y con los Sistemas Operativos más utilizados, siendo Linux, uno de ellos.

Lo más importante de esta tarjeta, es que no es tan sencillo programarla con otras herramientas, como las herramientas de *Xilinx* antes mencionadas. Alpha-Data proporciona sus propias herramientas para llevar a cabo esta tarea. Se trata de un driver programado en lenguaje C, que se encarga de la comunicación con el bus PCI (*ADM-XRC Device Driver*) y un kit de desarrollo de software (*ADM-XRC Software Development Kit* o solamente *SDK*) que proporciona las herramientas necesarias para la configuración y operación de aplicaciones [9]. Entre dichas herramientas se puede mencionar una API de programación en lenguaje C, códigos de ejemplo, módulos básicos para establecer la funcionalidad básica del FPGA en los distintos tipos de comunicación y su manual respectivo[10]. Se ha notado que el modelo *ADM-XP* no está soportado por las herramientas de *Xilinx*[1], por lo que no se pueden utilizar para realizar la programación del FPGA mediante iMPACT de una manera directa y simple como en el caso de la tarjeta XUP2VP, en su lugar, se debe recurrir al *SDK* de Alpha-Data, mencionado anteriormente, el cual contiene las bibliotecas necesarias para tal efecto, sin embargo las tareas referentes al FPGA, como el diseño, síntesis y generación del archivo de programación pueden ser realizadas con el programa *Xilinx ISE*. Si bien es una tarea con un cierto costo en tiempo para el programador, junto con la comprensión de las instrucciones necesarias en el diseño de hardware para llevar a cabo la comunicación con el bus PCI, el cual sigue un protocolo específico, la programación del FPGA en general es una tarea con un cierto grado de dificultad, y por la cual se necesitan bases que serán proporcionadas en el siguiente capítulo, en el que se hablará sobre el protocolo utilizado por la *ADM-XP* y cómo llega a implementarse en un diseño de hardware.

Capítulo 3

Introducción al Bus Local

3.1. Introducción

La creación de un diseño para este tipo de tarjeta, ciertamente no es una tarea trivial, puesto que no es algo que hagan automáticamente herramientas como EDK o iMPACT de *Xilinx* u otra compañía, como en el caso de la XUP2VP, que ya ha sido mencionada en el capítulo 2. Al tratarse de un bus diferente, y no estar soportada la tarjeta por dichas herramientas, hay muchos detalles en un diseño que tienen que ser tratados de manera manual. En este capítulo se establecerán algunos conceptos útiles para la comprensión del modo en el que, el FPGA se comunica con el puente de bus local para transferir datos, y realizar sobre ellos el procesamiento que el usuario desee, esto implica, conocer las señales que se emiten, el protocolo que las maneja, dispositivos que ayudan a realizar esta comunicación, los archivos que se necesita incluir en un proyecto y la manera en la que los datos puedan ser manejados.

3.2. El Puente de Bus Local y sus señales

Como se mencionó en el capítulo 2, la *ADM-XP* contiene un puente de bus local que se encarga de recibir las señales del bus PCI, las administra y las transmite al FPGA usando su propio protocolo de comunicación, esto es así para que, al crear un diseño en el FPGA, estas señales sean manejadas de acuerdo con alguno de los protocolos que se manejan con el puente de bus y no con el protocolo PCI, de modo que sea el puente, el que se encargue de dicha comunicación. Entonces, las señales del bus local es un conjunto de señales que se encargan de la transmisión y recepción de datos y obedecen a un sencillo protocolo.

El objetivo principal de estas señales es que se lleven a cabo transferencias de datos de una manera correcta, el FPGA no podría hacer nada con el *host* si no transmite y recibe datos de éste último, por lo tanto necesita realizar una serie de pasos en los que la información requerida, sea transferida de manera fiable.

La Figura 3.1 muestra las señales de control entre el puente de bus local y el FPGA, también se puede observar cuales señales son controladas por el puente de bus local, cuáles son controladas por el FPGA y cuáles por ambos. Cabe señalar que en este diagrama se muestra la forma habitual en la que se manejan las señales para fines de este proyecto, pues nos estamos enfocando en el modo de transferencia directa al esclavo, sin embargo para los otros protocolos, la dirección de las señales puede ser diferente.

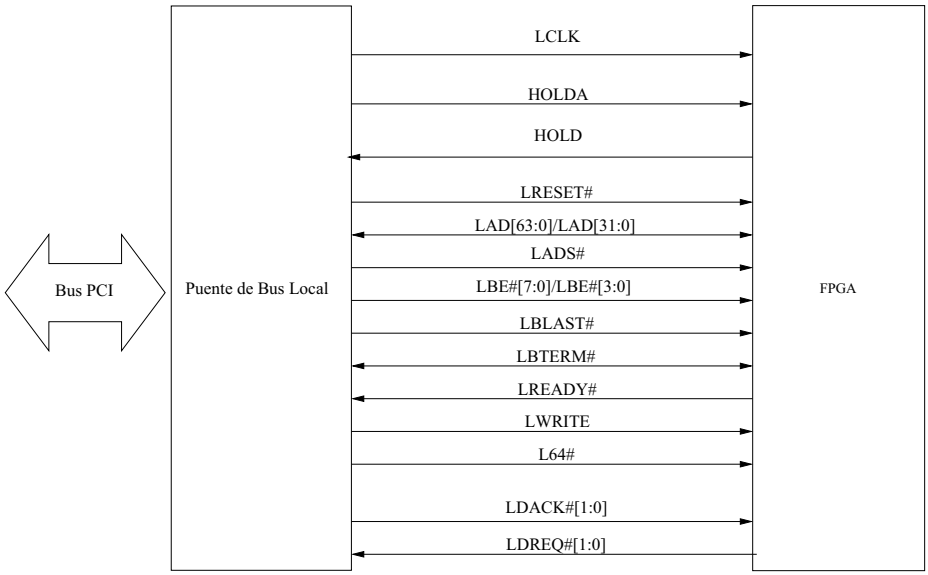

Figura 3.1: Señales que se transmiten entre el puente de bus local y el FPGA.

Cada una de estas señales es manejada ya sea por el puente o por el FPGA, dependiendo del protocolo que se esté manejando[10]. Las más importantes son descritas a continuación.

- **LAD (*Local Address Data*)**. Es un bus de hasta 64 bits, es el principal elemento del bus local, pues se encarga de manejar al bus de datos, a través del cual, se transportan datos de 32 o 64 bits, asimismo se encarga de manejar el bus de direcciones en el que se manejan direcciones en las que se leerán o se escribirán los datos, por esta razón, se dice que la tarjeta *ADM-XP* maneja un bus multiplexado. Las direcciones manejadas se pueden leer en LAD[23:2], los bits de LAD[1:0] valen cero todo el tiempo, y la dirección transferida en LAD[23:2] es un número de 22 bits.

- **LBE# (*Local Byte Enables*)**. También es un bus para manejar la dirección, sólo que se encarga de indicar cuáles bytes son los que se leerán de la dirección que se está manejando. De esta forma, LBE# permite direccionar bytes individuales. Por ejemplo, si se transmite una palabra y LBE#[3:0] contiene el dato "0001", significa que sólo se podrá leer el byte menos significativo de la palabra transferida. Si una transferencia que se está llevando a cabo es de

32 bits, entonces sólo LBE#[3:0] llevará datos. Si la transferencia es de 64 bits, entonces LBE#[7:0] llevará datos.

- **LADS# (*Local Address Strobe*)**. Sirve para marcar el inicio de una Transferencia. Es manejada por el puente.
- **L64# (*Local Bus 64 Bits*)**. Es una señal que indica si se están manejando 64 bits o 32 bits en el bus.
- **LRESET# (*Local Bus Reset*)**. Activada asíncronamente por el puente de bus local para hacer que todos los agentes de bus local regresen a un estado conocido. En este caso, si el diseño implementado y los diseños de los que pudiera estar conformado pueden recibir esta señal en caso de contar también con una terminal reset.
- **LBLAST# (*Local Bus Last*)**. Manejada por el puente. Sirve para indicar que la palabra actual es la palabra final en la transferencia. Cuando LRADY# es activada con LBLAST#, la transferencia actual termina. LBLAST# es para válida cada ciclo de una ráfaga.
- **LBTERM# (*Local Bus Terminate*)**. La envía el FPGA para terminar una ráfaga en curso inmediatamente, la palabra de datos en el bus LAD es transferida, y la ráfaga actual termina, sin hacer caso de LREADY# o LBLAST#.
- **LREADY# (*local Ready*)**. Controlado por el FPGA para indicar que la palabra de datos actual en el bus LAD ha sido transferida, si LBLAST# es también activada, la ráfaga actual termina.
- **LWRITE (*Local Write*)**. Manejado por el puente de bus local, indica si la ráfaga actual es una lectura o una escritura, si es activada, entonces el ciclo es una lectura (el puente pasa los datos a LAD)
- **LDACK# (*DMA acknowledge*)**. Manejado por el puente de PCI a bus local. Un bit de LDACK# es activado al mismo tiempo que LADS# para indicar que la ráfaga actual es una ráfaga DMA en modo demanda. Esta continúa activada hasta el final de la ráfaga. Cada bit de LDACK# corresponde a un canal DMA en el puente de PCI a Bus Local. A lo más un canal DMA puede estar realizando una ráfaga en el bus local en cualquier momento; por lo tanto, a lo más un bit de LDACK# puede ser activado en cualquier momento.
- **LDREQ# (*DMA request*).** Manejado por el FPGA. Sirve para terminar una transferencia de tipo DMA. Cada uno de los bits de LDREQ# corresponde a un canal DMA en el puente PCI a bus local. Una ráfaga puede ser terminada por la desactivación del bit correspondiente de LDREQ#. Esto es conocido como "pausar el modo demanda de DMA", y causará que el puente puente de bus local active LBLAST# tan rápido como sea posible.

- **LCLK (*Local Bus Clock*)**. Es el reloj de bus local, y no es controlada por el puente de bus local, sino por los recursos centrales, otras señales de bus local, con la excepción de LRESET#, son síncronas a LCLK. La frecuencia de LCLK normalmente se encuentra bajo el control de una aplicación que se ejecuta en el *host*.

- **HOLD y HOLDA**. Estas dos señales no son señales del bus, sino que se utilizan para que diferentes dispositivos hagan uso de él. Son controladas por un árbitro de bus, el cual decide a qué dispositivo otorgarle la posesión del bus local cuando tiene que hacer una transferencia de datos, ya sea el puente de bus local o el FPGA. Cada agente de bus (componente que hace uso del bus) puede tener este par de señales y hacer uso de ellas, dependiendo de cuál de ellos sea el maestro y cuál el esclavo. Basta decir que el FPGA recibe la señal HOLDA, con la que se le indica que puede hacer uso del bus, debido a que se trata de transferencias de tipo directas al esclavo, y no existen más que un maestro, que es el puente de bus local y un esclavo que es el FPGA, por lo tanto, no se necesita de la señal HOLD, pues el FPGA no tiene que solicitar el bus, solo esperar que se le permita utilizarlo. En el caso del FPGA, a estas señales se les identifica como **FHOLD** y **FHOLDA**.

3.3. El protocolo de comunicación con el bus PCI

El puente de bus local maneja varios protocolos de comunicación con el FPGA para ofrecer muchas posibilidades en el diseño de hardware. A continuación se mencionan los tres diferentes tipos de transferencias:

- **Transferencias directas al esclavo (Direct-Slave Transfer)**. Es el método básico de transferencia de datos desde y hacia el FPGA. En este se identifica al puente de bus local como maestro y al FPGA como esclavo. Existen dos formas en las que los datos se pueden transferir:

  - Lectura/escritura de una sola palabra (cadena de bits).
  - Lectura/escritura de una ráfaga (varias palabras en una sola transferencia).

- **Transferencias por DMA**. Las transferencias DMA sirven para grandes cantidades de datos entre dos dispositivos sin hacer ningún otro tipo de procesamiento, por lo que este tipo de transferencia, resulta útil al trabajar con datos de volúmenes grandes. Este tipo de transferencia tiene tres modos distintos:

  - Modo demanda. Este modo consiste en que un motor DMA transfiere datos "en demanda del FPGA". Por ejemplo, en un diseño, el cual contiene una FIFO cuyos datos son leídos vía el bus local, el FPGA puede solicitar que el motor DMA transfiera algunos datos sólo cuando la FIFO no esté vacía.

  - Modo LEOT. El modo LEOT ofrece una manera en la que el FPGA puede terminar una transferencia DMA antes de que el número programado de bytes de datos haya sido transferido.
  - Dirección Constante. Durante toda la transmisión DMA, el bus local presentado en LAD es mantenido constante. Esto es útil para acceder un registro que es actualmente la cabeza o la cola de una memoria FIFO que es mapeada en una dirección del bus local en particular, en lugar del incremento automáticamente de la dirección del bus local, éste permanece constante , durante una ráfaga para luego pasar a otra ráfaga y así pasa a la siguiente dirección. Este modo se usa en conjunto con los otros dos modos, ya sea **modo de demanda** o modo **LEOT**.

- **Transferencias directas al maestro (Direct-Master Transfer)**. Este tipo de transferencia es similar al Direct-Slave, pero el FPGA toma el papel de maestro y el puente de bus el de esclavo.

En este proyecto se utilizó la transferencia directa al esclavo haciendo lecturas/escrituras de una sola palabra.

Básicamente, el protocolo funciona en tres pasos:

1. La señal LADS# es activada y se transmite por LAD la dirección en la que se transmitirán los datos.

2. LAD contiene ahora los datos a transferir, por lo que el FPGA toma el control de este bus para leerlo, por su parte, el puente maneja a LBE para indicar cuáles bits de la palabra transferida serán los que se deberán leer.

3. Si LREADY es activada por el FPGA y LBLAST# ha sido activada por el puente de bus, termina la transferencia.

En el diagrama de señales de la Figura 3.2 ilustra esta secuencia. Es importante señalar nuevamente que LBLAST# y LBTERM# son manejadas por el FPGA, obsérvese que LBTERM# no es activada, ya que ésto depende de si el diseño del usuario utiliza ésta señal. En el caso de LBLAST#, al tratarse de una sola palabra, puede ser activada inmediatamente después de LADS#. Después de que LAD lleva la dirección del bus, ocurre un ciclo de reloj antes de que LAD lleve la palabra de datos.

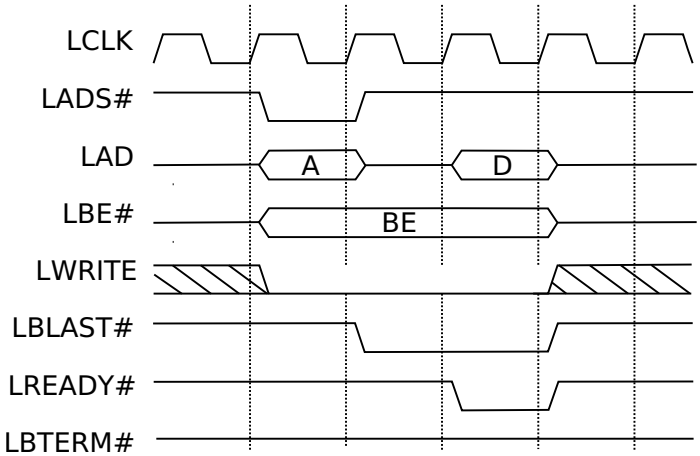

Figura 3.2: Diagrama de señales para transferencias directas al esclavo.

3.4. La máquina de estados plxdssm

Se ha hablado ya sobre el protocolo de comunicación, sobre cómo el puente de bus local emite sus señales al FPGA y éste responde tomando el control del bus e indicando que la palabra ha sido transferida, sin embargo, el FPGA es un dispositivo que, antes de su programación no puede hacer absolutamente nada, no se podría hacer un diseño y pensar que manejará, sin que nadie se lo indique las señales del puente de bus local. Por eso, se necesita describir un componente que se encargue de manejar las señales e indicar el momento en el que se deben hacer las transferencias de los datos. Por esta razón, el *SDK* viene ya con varios módulos que se encargan de dicha tarea, de acuerdo al protocolo que se maneje para llevar a cabo transferencias, ya sea directas al esclavo o por DMA, o también para trabajar con las memorias incrustadas en la tarjeta.

La máquina de estados plxdssm (direct slave state machine) ayuda a realizar la comunicación con el bus local durante las transferencias directas al esclavo. Consiste específicamente en cuatro estados, los cuales cambian de acuerdo con las señales recibidas tanto del puente de bus local como de la aplicación de usuario[1]:

1. IDLE. Indica que no se está realizando ninguna acción con el bus local.

2. DECODE. Indica el inicio de de una transferencia de datos.

3. WAIT. Se pasa a este estado si la aplicación de usuario requiere que ocurran algunos ciclos de espera antes de una transferencia.

4. XFER. En este estado se realiza la lectura o escritura de los datos.

[1]En el contexto de este capítulo, aplicación de usuario se refiere al diseño en VHDL dentro del cual se encuentra el componente plxdssm.

La máquina de estados plxdssm, como componente en VHDL, forma parte de un paquete creado para la comunicación con el bus local (localbus package) y proporciona un mecanismo de control para una interfaz de bus local dentro del diseño en un FPGA.

La Figura 3.3 muestra cuáles de sus pines se pueden conectar a la aplicación de usuario y cuáles a las señales del bus local. Este componente se debe incluir dentro del diseño en VHDL del usuario para que realice reciba las señales del puente de bus, mientras el hardware diseñado se encarga de sus propias funciones.

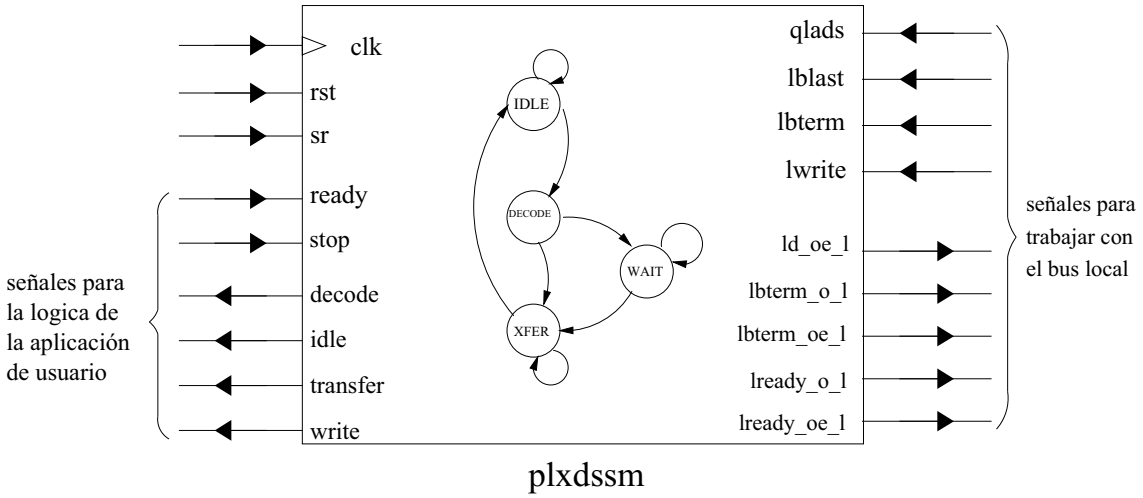

Figura 3.3: La máquina de estados plxdssm vista como componente.

El componente plxdssm se encarga de realizar las funciones más básicas de acuerdo con el protocolo de comunicación; sin embargo, sigue siendo responsabilidad del programador manejar las señales de salida de la máquina de estados y, en función de ellas, realizar la transferencia de los datos. Nótese que por esta razón las señales LBE y LAD que se encargan de direccionar y transferir los datos no son manejadas por el plxdssm sino por el programador.

A continuación se da una breve explicación de las señales recibidas y emitidas por las terminales del componente plxdssm, estas están clasificadas de acuerdo a su función de entrada o salida y para qué propósito son activadas.

3.4.1. Señales generales

- **clk**. Es la señal de reloj recibida para el funcionamiento del FPGA, la cual, también es recibida por el puente de bus local, de esta manera, ambos agentes pueden enviar y recibir señales activadas y desactivadas en cada ciclo de reloj

- **sr y rst**. Son los puertos de reset síncrono y asíncrono reespectivamente. Recuérdese que el síncrono es aquel que necesita de un pulso de reloj para que ocurra; por el contrario, el reset asíncrono puede ocurrir en cualquier instante. Ambas señales pueden ser derivadas de la señal **LRESET#,** y en la aplica-

ción de usuario puede ser usada cualquiera de las dos, dejando a una de ellas conectada con un cero lógico.

3.4.2. Señales de entrada provenientes de la aplicación de usuario

Estas señales son controladas por el diseño del usuario con el objetivo de hacer al módulo plxdssm cambiar de estado de manera independiente a las demás señales. En un diseño como el que se hizo en este proyecto, estas señales no fueron requeridas.

- **ready**. Esta entrada sirve para informar al módulo plxdssm que la aplicación de usuario está lista para transferir datos. Activar **ready** causa que **lready_o_l** sea activada en el siguiente ciclo, asumiendo que la transferencia directa a esclavo esté en progreso. En la aplicación de usuario se puede dejar conectada a un uno lógico, pensando que, a menos que se desee tener estados de espera, en todo momento se querrá que la aplicación esté lista para una trasferencia.

- **stop**. Esta entrada informa al módulo plxdssm que la aplicación del usuario desea terminar la transferencia en curso. Asumiendo que la transferencia directa de esclavo está en progreso, activar **stop** puede o no causar que **lbterm_o_l** sea activada en el siguiente ciclo, dependiendo si **ready** ha sido activada. Si se desea que el componente plxdssm no utilice esta señal, se debe dejar puesta en uno lógico.

3.4.3. Señales de salida para la aplicación de usuario

Estas conexiones de la máquina de estados proporcionan señales para que la aplicación de usuario determine qué acciones tomar en caso de requerirlo. Son las siguientes:

- **transfer**. Esta señal indica a la aplicación de usuario que en el ciclo de reloj actual se están transfiriendo datos.

- **write**. Indica que la transferencia del puente de bus local al FPGA es una escritura si tiene un nivel alto, o una lectura si tiene un nivel bajo.

- **decode**. Esta salida se activa en el siguiente ciclo después de **qlads**, del mismo modo que una dirección dada por el puente de bus local es válida también hasta un ciclo después de que se activó **qlads,** Puesto que primero es guardada en un registro , por lo que decode sirve para indicar que la dirección puede ser leída. En un diseño básico esta señal no es necesaria, debido a que sólo se necesita leer el registro en el que se guardó la dirección, sin esperar a que esté lista, por lo que se puede dejar desconectada.

- **Idle**. Esta señal indica que la máquina de estado está actualmente sin ser utilizada. Idle nunca es activada al mismo tiempo que **decode** o **transfer.** Esta

señal tampoco es necesaria, por lo que se puede dejar desconectada, ciertamente no se necesita saber si la máquina de estados está o no manejando un ciclo del bus local.

3.4.4. Señales de salida para el trabajo con el bus

Estas señales de salida sirven para indicar el inicio, proceso y término de una transferencia entre el FPGA y el bus local.

- **ld_oe_l**. Es una señal activa en nivel bajo para habilitar los pines de **LAD**.
- **lbterm_o_l**. Esta salida maneja la señal LBTERM#. Puesto que esta señal es de entrada y salida (inout), se debe utilizar un habilitador para tener un control sobre su activación y asegurar una señal salida. Para este fin, la máquina de estados tiene la señal **lbterm_oe_l**.
- **lready_o_l**. Esta salida maneja la señal LREADY#. Puesto que esta señal es de entrada y salida (inout), se debe utilizar un habilitador para tener un control sobre su activación y asegurar una señal de salida. Para este fin, la máquina de estados tiene la señal **lready_oe_l**

3.4.5. Señales de entrada provenientes del puente de bus local

Las señales que provienen del puente de bus local se conectan con las respectivas terminales del componente plxdssm, algunas necesitan alguna modificación antes de llegar a la máquina de estados dependiendo su naturaleza.

- **lbterm**. Recibe la señal de entrada **LBTERM#**.
- **lwrite**. Es la señal **LWRITE**.
- **lblast**. Es una versión de la señal de bus local **LBLAST#** en nivel activo alto.
- **qlads**. Esta señal debe ser una versión activa en nivel alto de la señal de bus local LADS#. Típicamente obtenida de una función combinatoria de **!LADS#**, **!LA[23]** y **!FHOLDA**[2] . Lo que hace es que la máquina de estados plxdssm ignore los ciclos de bus local para los cuales el FPGA no está siendo utilizado. Se utiliza **LA[23]** para monitorizar cambios en la dirección del bus, y **FHOLDA** como indicación de que se puede hacer uso del bus local. La instrucción en VHDL para establecer a qlads queda de la siguiente manera:

  qlads <= !LADS# and !LA[23] and !FHOLDA

Hasta ahora se ha dado la información que se requiere para poder implementa un diseño tomando en cuenta al protocolo de comunicación y los recursos requeridos, en el capítulo 4 se hablará de cómo se llevó esto a un ejemplo real.

[2]El símbolo ! indica la negación de la señal

Capítulo 4

Creación de un diseño en VHDL para la ADM-XP

4.1. Introducción

Con la información obtenida del manual del *SDK*, se pudo diseñar un hardware muy básico que recibiera diez datos y los volviera a escribir en el bus local. Para ello, se utilizó como base el diseño `simple`, que es un ejemplo contenido en el *SDK*, que sirve para mostrar la implementación de registros. En este capítulo se muestra la estructura de un diseño básico que utiliza registros para recibir datos, y cómo los datos contenidos en registros pueden ser leídos por el bus, además se describen los archivos que se deben implementar para este fin, en un proyecto creado con la herramienta *Xilinx ISE*. La importancia de este capítulo está en que, para desarrollar el sistema de este proyecto fue necesario, en primera instancia, estudiar la estructura de un proyecto de este tipo para saber implementar una aplicación.

4.2. Codificación de un diseño básico

La estructura básica de un diseño de usuario debe ser realizada conforme al protocolo de comunicación y al nivel activo de las señales del bus, a continuación se describen las diferentes partes que debe llevar un diseño que funciona mediante transferencias directas al esclavo. En base a este diseño, sólo se necesita implementar dentro de él, un componente cuyas entradas y salidas estén conectadas a alguno de los registros de esta arquitectura.

4.2.1. Conexión de las entradas

Las entradas deben estar puestas como activas en nivel alto. Esto es que el valor de una entrada se considere activada si está en un uno lógico. Para esto, algunas de ellas

son negadas y conectadas con señales internas[1] para dejarlas con esa característica, y en otros casos, sólo para mantener sus valores y poder hacer operaciones lógicas en función del reloj.

4.2.2. Manejo de LADS#

Se trata de generar una versión adecuada de **LADS#**, la cual se activa cuando el FPGA es direccionado Y el FPGA no funciona como maestro de bus local, esto se logra tomando el valor negado de **LA[23]** y la señal de bus FHOLDA ayuda a determinar si el FPGA funciona como esclavo.

4.2.3. Almacenar la dirección del bus local

Se hace durante el pulso de **LADS#**, con el fin de leer la dirección, y en base a ello, guardar el dato que se reciba en un registro específico.

4.2.4. Manejo de LBTERM#

Cuando **LBTERM#** funciona como señal de salida, debe ser manejada sólo cuando el FPGA es direccionado, esto implica que se debe enviar la salida **lbterm_o_l** sólo cuando **lbterm_oe_l** esté activado; de otro modo estará como alta impedancia, por que el control lógico de la tarjeta podría también manejarlo.

4.2.5. Manejo de LREADY#

Cuando **LREADY#** funciona como señal de salida, debe ser manejada sólo cuando el FPGA es direccionado, esto implica que se debe enviar la salida **lready_o_l** sólo cuando **lready_oe_l** esté activado; de otro modo estará como alta impedancia, por que el control lógico de la tarjeta podría también manejarlo.

4.2.6. Manejar el bus de datos durante una lectura

La salida **ld_oe_l** de la máquina de estados se encarga de indicar cuando el diseño esta listo para enviar los datos, es por esto que la aplicación de usuario, en su lógica debe monitorizar esta salida. Mediante una señal interna **ld_o** conectada con **LAD**, se envían los datos durante la lectura sólo cuando ld_oe_l esté activada, en otro caso, la salida para **LAD** debe estar puesta en alta impedancia.

4.2.7. Actualizar los registros

Mediante un proceso (process), hacer que el diseño determine si el ciclo actual es una escritura, dependiendo de la dirección que llega en el bus de direcciónes, los

[1] En el contexto de este capítulo, entiéndase "señales internas" como las señales que se manejan dentro de la descripción del circuito, y que se declaran con la palabra reservada **signal**.

datos se deben almacenar en un registro asociado con esa dirección, esto se logra, en función de las señales **transfer** y **write** de la máquina de estados que indican que se está realizando una operación de escritura. En esta parte lo más importante es el cómo se leen los datos y la dirección del bus.
Para empezar, recordemos que tanto el bus de datos como el bus de direcciones se manejan en la misma señal **LAD**. Por lo tanto, se definen dos señales internas de la arquitectura llamadas **la_q,** que conecta con **LAD** cuando se maneja una dirección, y **ld_i**, que contendrá los datos de entrada. Una vez leída la dirección de **la_q**, se leen los bits de **LBE#** para determinar cuáles bytes de ld_i son legibles y se procede a almacenar el contenido de **ld_i** en un registro, para este proyecto, se declararon 10 registros: **reg0**, **reg1**,..,**reg9**, los cuales almacenan los datos que sean enviados a las direcciones 0 a 9, que son los valores que tomará en cuenta para **la_q**.

4.2.8. Generar la señal ld_o

Como se ha dicho una señal interna lleva los datos que se transferirán cuando se trate de una lectura del bus local. Al igual que en la escritura, se recibe una dirección almacenada en **la_q**, y dependiendo de su valor, **ld_o** es conectado con el registro que se desee mostrar. Resulta más sencilla la operación de lectura, pues sólo se necesita habilitar a LAD que ya se encuentra conectada con **ld_o** (manejar el bus de datos durante una lectura).

4.3. Conexión del diseño con el FPGA (archivo UCF)

Los archivos de especificaciones de usuario (*User Constraints File*, UCF) se encargan de hacer la relación entre los pines físicos del FPGA y los pines del hardware diseñado por el usuario, por lo que son una parte crucial en el diseño de hardware para el FPGA. El *SDK* viene con varios archivos UCF que, dependiendo de las funciones que se necesitan, indican a cuáles pines del FPGA se deben conectar las señales de entrada-salida de un diseño. Estos archivos contienen todos los pines del FPGA, los que sirven para las señales del bus local, los de los datos, las memorias y los puertos de comunicación con periféricos que se conecten a la tarjeta de desarrollo.

El archivo admxp.ucf, contiene los principales pines, que son los que manejan las señales del bus que se han explicado anteriormente. Para la creación del diseño planteado en este capítulo, se tomó como base a este archivo y también se tomaron algunos fragmentos de código de los demás archivos de ejemplo, pues manejan algunos aspectos de temporización de las señales para manejar los retardos que se puedan generar.

4.4. Uso de Xilinx ISE (Diseño Básico)

El siguiente paso en la creación del diseño, es integrar las partes descritas anteriormente en un sólo proyecto para generar el archivo de programación, esto se puede realizar con la herramienta *Xilinx ISE Design Suite* que, como se había mencionado en el capítulo 1, permite describir, simular e implementar el diseño en cuestión para gran cantidad de familias de dispositivos FPGA. En la Figura 4.1, se observan los bloques de este sistema, donde se observa que el código del diseño pasa por un proceso de síntesis, implementación y programación, además de poderse simular el diseño antes de ser programado en el dispositivo.

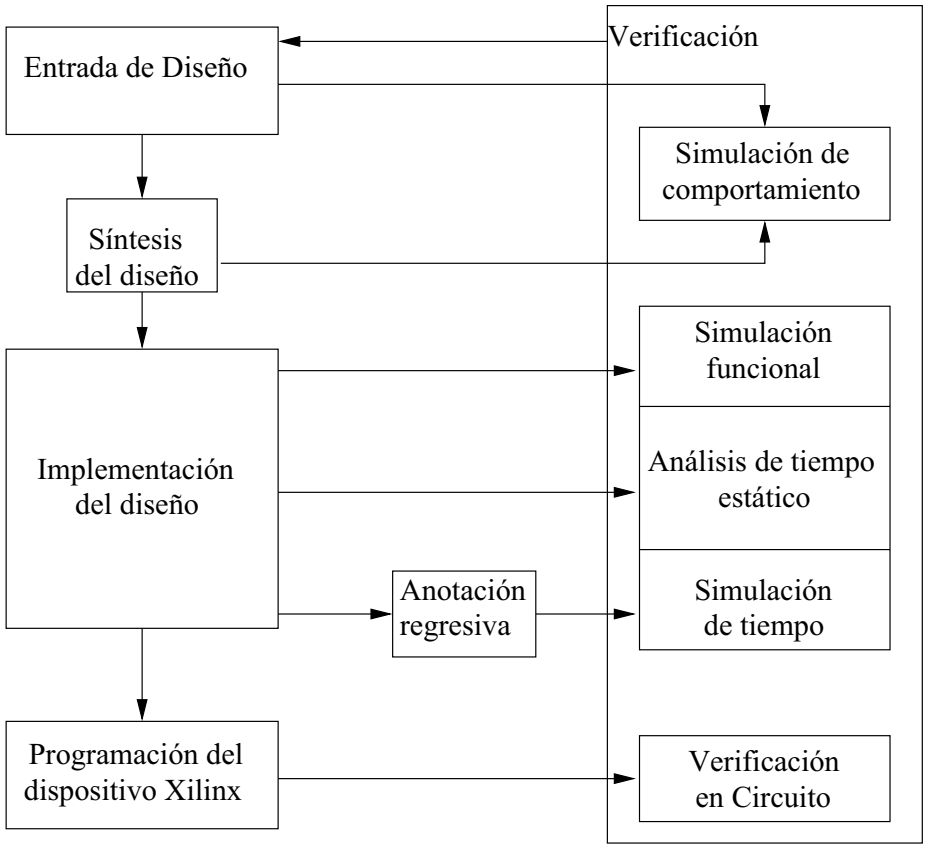

Figura 4.1: Bloques del sistema *Xilinx Ise Design Tools* [1].

Aunque la programación de la tarjeta no se puede hacer directamente con las herramientas de *Xilinx*, la tarea de generar un *bitstream* para el FPGA sigue siendo posible, ya que la versión 8.2i de *Xilinx ISE*, soporta el modelo de FPGA contenido en la tarjeta de desarrollo utilizada. Existen dos formas de llevar a cabo este procedimiento, la primera opción es utilizar un conjunto de comandos que *Xilinx* proporciona, el cual es conocido como *Xilinx ISE Tools*; la segunda opción es utilizar la herramienta gráfica conocida como Porject navigator. La segunda opción fue la elegida, pues el uso de los comandos requiere más tiempo en cuando al adiestra-

miento en su uso. El Project Navigator se encarga de ejecutar los procesos de *Xilinx ISE Tools* con las opciones y argumentos necesarios para, a partir de un proyecto de diseño, realizar las tareas de síntesis, implementación y generación del archivo de programación, por lo que, al hacerlo de forma automática, su uso resulta más fácil y rápido.

Para generar un archivo de programación, que es lo que se desea obtener para continuar con el desarrollo del proyecto, sólo se necesitaron los siguientes tres pasos:

1. Crear un proyecto nuevo.
2. Agregar los siguientes archivos:
   - **plxdssm.vhd**. Es el diseño de la máquina de estados utilizada en el diseño básico, este archivo viene en el *SDK*.
   - **localbus_pkg.vhd**. Contiene la definición de la máquina de estados, también es proporcionado por el *SDK*.
   - **design.ucf**. Es un archivo creado a partir del archivo **admxp.ucf** proporcionado por el *SDK*
   - **design.vhd**. Es el diseño básico creado a partir del ejemplo **simplexpl.vhd** proporcionado por el *SDK*.
3. Ejecutar los procesos de síntesis, implementación y generación del archivo de programación.

Obteniendo el archivo de programación, el siguiente paso es crear un a aplicación que sea capaz de programar el FPGA y también de enviar y recibir datos del dispositivo programado. Esto se verá en el siguiente capítulo.

Capítulo 5

Programación y prueba del comando en Linux

5.1. Introducción

El comando admxpprg está diseñado para interactuar con la tarjeta *ADM-XP*, realizando algunas tareas básicas que permiten al programador ahorrar tiempo en la programación de una aplicación para configurar el FPGA con un *bitstream* y para observar las salidas que el hardware que ha diseñado produce, además de que muestra algunos datos sobre el FPGA y los módulos de memoria que la tarjeta contiene. A continuación se hablará sobre la estructura del programa y las funciones que realiza.

5.2. Comportamiento del comando admxpprg

El programa se hizo pensando en el comportamiento típico de una aplicación en linux, que consiste en un comando, el cual, recibe distintas opciones desde la línea de comandos y, dependiendo de dichas opciones y los argumentos que involucren, el programa realizará sus distintas funciones. De esta misma manera, el comando admxpprg, puede realizar las siguientes opciones:

- **Mostrar el uso del comando**. Como todo comando en Linux, debe mostrar el uso del comando, por lo que, mediante la opción --**help**, el programa motstrará el uso del comando, es decir, muesra las opciones disponibles y una breve descripción de para qué son utilizadas.
- **Mostrar la información de la tarjeta**. La API de programación del *SDK* ofrece algunas funciones que sirven para obtener cierta información sobre la tarjeta de desarrollo y el FPGA. Esta opción del programa reúne dichas funciones para mostrar información sobre la versión de las herramientas, las direcciones de bus en las que funciona el dispositivo y los bancos de memoria que contiene

la tarjeta. Esta funcionalidad del programa es ejecutada mediante la opción **-i**, o con la opción larga --**info**, y no requiere de ningún argumento.

- **Programar la tarjeta *adm-xp***. Esta operación se logra utilizando la opción **-c<*bitestream*>**, o la opción larga **--configure=<*bitstream*>** agregando además, la ruta del archivo *bitstream*, el cual es un archivo de extensión **.bit** que, anteriormente ha sido generado mediante las herramientas de *Xilinx*. El programa **admxpprg** hace que se ejecuten las funciones necesarias para que el *bitstream* sea analizado y descargado en el *FPGA,* de manera que sea programado con el diseño de hardware. Lo que muestra este comando es un mensaje de éxito o un mensaje de error usando la salida estándar.

- **Ejecutar la aplicación**. El programa **admxpprg** recibe por la entrada estándar **diez** valores hexadecimales para enviarlos al FPGA, que se encarga de hacer el procesamiento correspondiente a la arquitectura del diseño programado en él. El único requisito para que los valores sean enviados al FPGA, es que sean números hexadecimales, es decir que, aunque el FPGA no usa todas las entradas, los diez valores deben ser introducidos, aunque sean cero o cualquier otro valor. Esta opción es reconocida por el programa si se escribe **-x** ó **--execute** como argumento del programa.

- **Mostrar los registros del diseño implementado**. Esta funcionalidad del programa se encarga de leer los registros del FPGA y mostrar sus valores. Se puede ejecutar escribiendo la opción **-o** o también --**output**.

- **Configurar el reloj de la tarjeta *adm-xp***. En este programa se consideró la posibilidad de poder asignar una frecuencia al reloj de la tarjeta, ya que algunas veces se querrá que los pulsos de reloj a los que funciona alguna aplicación en el FPGA tengan una mayor o menor duración. Puesto que también hay una función para realizar esta operación, se ha implementado esta opción en el comando, a la cual, se tiene acceso escribiendo **-k<valor>** o --**clock=<valor>** como argumento para el programa, en donde **valor** es la frecuencia en MHz.

La forma en la que el mecanismo que reconoce a las opciones fue programado, permite que el programa reciba como argumentos varias opciones y en cualquier orden, pues en el código se incluyó la biblioteca **getopt.h**, cuyas funciones permiten reconocer cuando los argumentos de un programa son opciones y también cuando una opción debe contener parámetros. La opción **-c** por ejemplo, indica que el programa reciba también la ruta del archivo de programación, en caso de no ser así, el programa termina indicando que falta el archivo en cuestión.

5.3. Diseño del programa

El programa fue escrito en lenguaje C++ , a pesar de ello, las funciones de la API que proporciona el *SDK* y la biblioteca **getopt.h** que fueron escritas para lenguaje

C, pueden ser utilizadas, ya que C++ es sólo una extensión de C. Por estar escrito el código fuente en lenguaje C++, se aprovechó el paradigma orientado a objetos, el cual es una característica de C++ sobre C, y que ayuda a crear programas más entendibles en cuanto a la estructura de sus funciones.

El diagrama de clases de la Figura 5.1 muestra la organización del programa, se puede observar que existe una clase Tarjeta la cual contiene los métodos para realizar las acciones sobre el FPGA. Otra Clase llamada Parser, la cual esta pensada para llevar a cabo la tarea de analizar las opciones introducidas como argumentos del programa y mantener el estado de dichas opciones en el tanscurso de su ejecución, esto lo hace almacenando el estado de unas banderas que indican si la opción correspondiente está activada o no. Por su parte la clase principal Admxpprg, contiene los métodos uso() para mostrar la ayuda sobre el comando, obtenerEntradas(), para recibir los datos para el FPGA; y el método checkData(), que analiza los datos introducidos para que sean enviados correctamente a la clase Tarjeta.

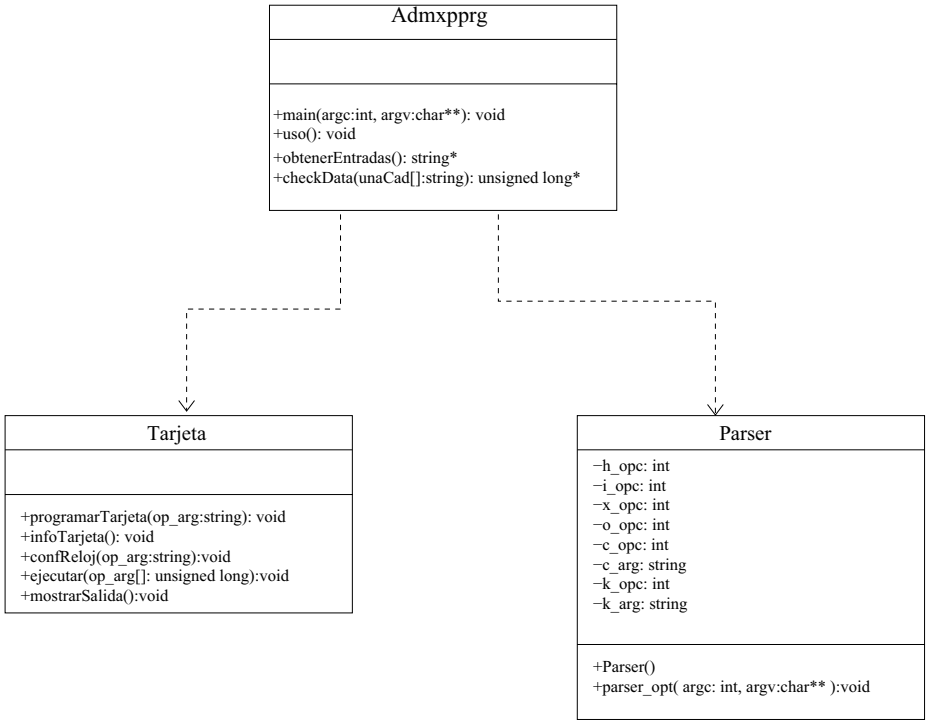

Figura 5.1: Diagrama de clases del programa admxpprg.

5.4. Funcionamiento interno del programa

El diagrama de flujo de la Figura 5.2 permite observar el flujo principal del programa, en donde, a excepción de mostrar la ayuda del comando y analizar las opciones, cada proceso consiste en un método ejecutado por una instancia de la clase Tarjeta.

Las tareas realizadas por el programa **admxpprg** sobre la tarjeta son descritas a mas detalle a continuación.

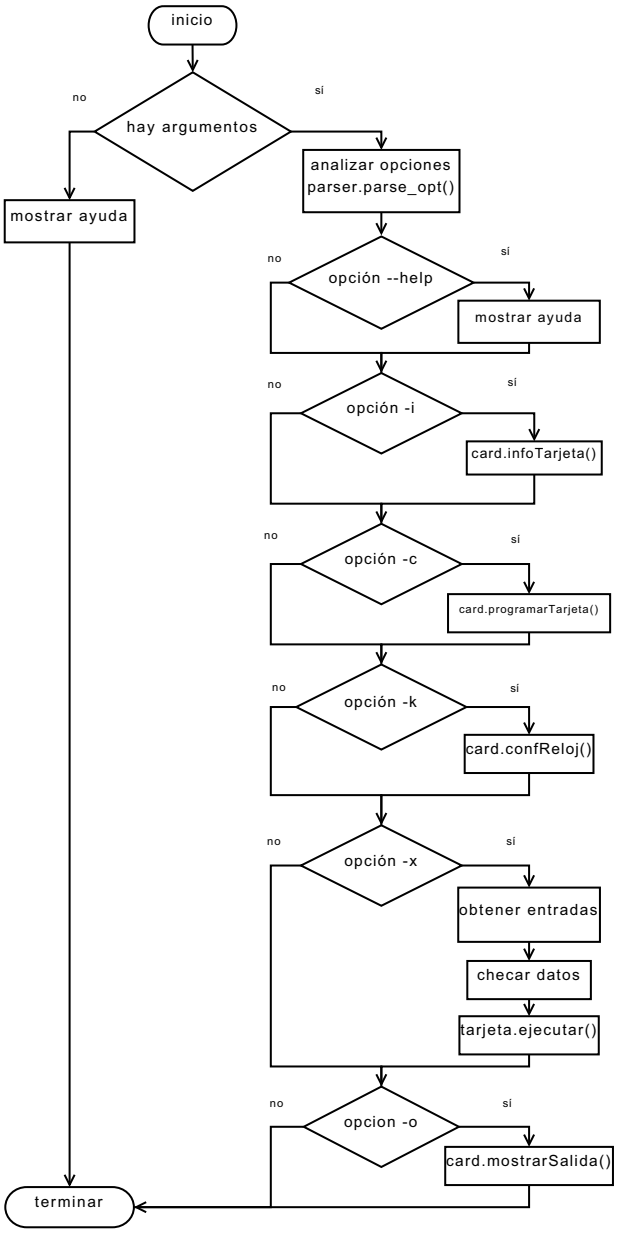

Figura 5.2: Flujo principal del programa.

5.4.1. Las Funciones de la API de programación

Como se dijo antes, la funcionalidad del programa admxpprg depende del uso de las funciones proporcionadas por Alpha-Data. En este capítulo no se pretende dar los detalles sobre dichas funciones, sino dar una idea general sobre a forma en la que trabajan y mostrar las partes principales de las funciones del programa admxpprg.

Para cada función del programa admxpprg que trabaje con la tarjeta de desarrollo, se realizan siempre dos operaciones: abrir y cerrar la tarjeta. Abrir la tarjeta significa, en térinos del código, que se obtendrá un valor de tipo **ADMXRC2_HANDLE** con el que se identificará a la tarjeta. Toda función de la API usará este valor para actuar sobre la tarjeta. En términos de su funcionamiento, Lo que sucede es que, por medio del `textitdriver`[1], el cual se encarga de realizar la comunicación con la tarjeta, se intenta obtener una respuesta satisfactoria que indique que la tarjeta está disponible, y que se puede hacer uso de ella. Cabe mencionar que por esta razón dos programas no pueden utilizar la tarjeta al mismo tiempo.

Todas las funciones que se mencionarán a continuación regresan un valor de tipo **ADMXRC2_STATUS**, que sirve para indicar el resultado de la llamada de la función. Dado que es un tipo enumerado, contiene una lista de palabras que tendrán un valor consecutivo, de esta forma, una función de la API, que retorne un error o éxito, no lo hará con un valor entero en específico, sino con una palabra definida en la enumeración. Esto es con el objetivo de que los errores puedan ser manejados con mayor facilidad. Normalmente se espera que el valor de retorno sea siempre **ADMXRC2_SUCCESS**. Por ejemplo, hay ocasiones en las que el `textitdriver` de la tarjeta no esta activado, entonces, la función **ADMXRC2_OpenCard** regresará el valor **ADMXRC2_CARD_NOT_FOUND**

5.4.2. Programación del FPGA por un *bitstream*

Para programar el FPGA se necesita hacer uso de algunas funciones de la API del DSK, en especial, existe una función que se encarga de realizar la programación a partir de un archivo de extensión **.bit**, siempre y cuando sea un archivo válido, el cual, como ya hemos visto se puede generar con Xilinx ISE. La función es **ADMXRC2_ConfigureFromFile**, para que funcione, se le debe dar como parámetros la ruta del bitstream y un índice que le indica qué tarjeta se está utilizando, pues el `textitdriver` puede controlar a más de una tarjeta. Obsérvese la Figura 5.3, en la que semuestra el diagrama de flujo para programar el FPGA.

[1]El textitdriver de la *ADM-XP* también es proporcionado por Alpha-Data, y debe ser previamente instalado y ejecutado antes que cualquier programa que utilice la API del SDK.

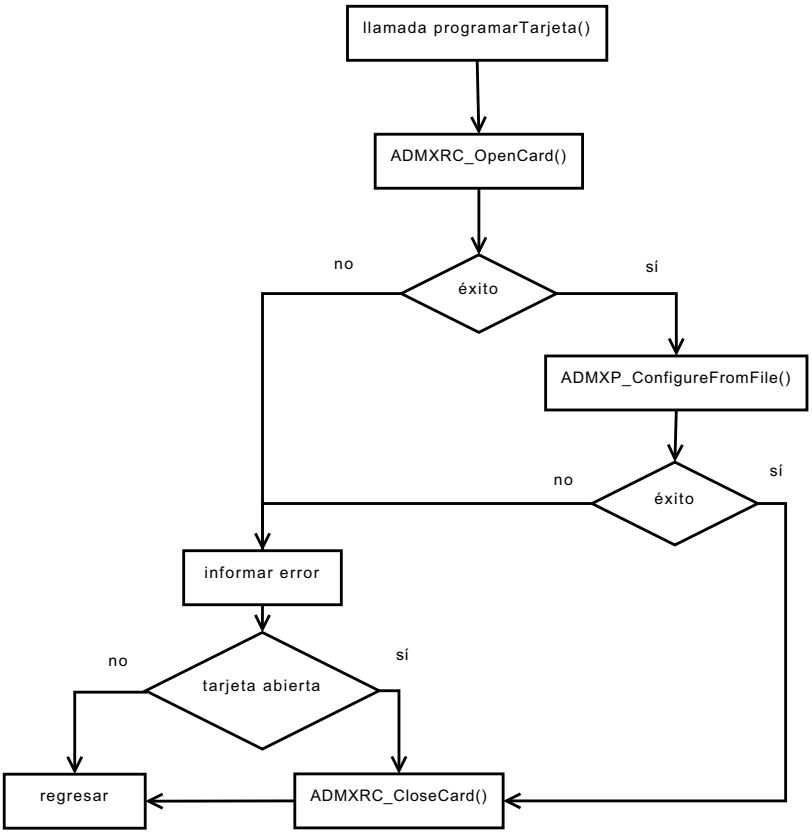

Figura 5.3: Diagrama de flujo de la función para programar la tarjeta.

5.4.3. Obtención de la información del FPGA

Para obtener la información mediante la opción **-i** del comando, se recurrió a algunas de las funciones de la API del *SDK*. Aunque no se obtienen demasiados datos relevantes, se piensa que la implementación de esta opción para el comando proporciona una idea de la forma en la que se debería implementar en un futuro la posibilidad de extraer información más útil para el usuario. El diagrama de flujo de la Figura 5.4 muestra el funcionamiento de esta opción. Las funciones utilizadas que se listan a continuación extraen toda la información sobre la tarjeta, el espacio de direcciones en el que trabaja, y los bancos de memoria con los que cuenta.

- **ADMXRC2_GetCardInfo**. Esta función ayuda a obtener información general sobre la tarjeta. Un parámetro de esta función que es una estructura de tipo ADMXRC2_CARD_INFO almacena datos como el tipo de tarjeta, número de serie, el número de generadores de reloj, el número de canales DMA, de bancos de RAM, y de espacios de bus local en los que puede escribir datos. Además de ser la información básica, se puede saber las direcciones en las que se pueden escribir datos, puesto que el espacio de bus local es aquel en el que se escriben datos que serán enviados directamente al FPGA, conocer estas direcciones es útil para comprender el cómo se escribe y lee un dato del FPGA, haciendo uso

de direcciones tanto físicas como virtuales.

- **ADMXRC2_GetVersionInfo**. Esta función regresa información sobre la bilioteca de la API y el `textitdriver`. En la API Existe una estructura llamada ADMXRC2_VERSION_INFO la cual es pasada como argumento por referencia y en la cual se guarda la información que se está solicitando. La utilidad de obtener esta información es que, el éxito en el funcionamiento de las herramientas de las que se dispone muchas veces está determinado por la versión que se está utilizando, el saber la versión de una herramienta ayudará a tomar mejores deciciones para solucionar un problema.

- **ADMXRC2_GetSpaceInfo**. Sirve para obtener información sobre un espacio de direcciones de bus local, por medio de una estructura llamada ADMXRC2_SPACE_INFO. La *ADM-XP* contiene dos regiones en las que hay un espacio de direcciones, una es la del FPGA, y otra sirve para el control de la tarjeta. Al asignarse los valores a la estructura, se obtiene la siguiente información:

  - La dirección PCI en la que comienza la región del espacio de direcciones
  - La dirección en la que comienza el espacio de direcciones de la tarjeta y su tamaño en bytes, en la aplicación se usa esta información para representar el rango de direcciones en las que se puede escribir en el FPGA.
  - La dirección en la que comienza el espacio de direcciones virtual y su tamaño, este es, el espacio de direcciones que utiliza el programa, accediendo a ellas usando apuntadores. Escribiendo y leyendo datos en dichos apuntadores es como el programa lee y escribe sobre los registros que se programan en el FPGA.

Esta información resulta evidentemente útil para saber los detalles sobre las direcciones que la aplicación utiliza para enviar y recibir datos del FPGA.

- **ADMXRC2_GetBankInfo**. Esta función se encarga de obtener la información sobre un banco de memoria, almacenándola en una estructura llamada ADMXRC2_BANK_INFO. El punto más importante es que, utilizando un índice como argumento de esta función, se puede acceder a cada banco para saber sus detalles. Como resultado, se obtiene el número de direcciones (palabras direccionables) y el tamaño de estas en bits, y por multiplicación de estos dos valores el tamaño total del banco. Muchas ocasiones se necesitará trabajar con estos bancos, sin embargo, si bien es cierto que la cantidad con la que se cuenta es limitada, tendrán que crearse aplicaciones que hagan un buen uso de estos recursos sin el peligro de agotarlos y provocar una disminución en la eficiencia, por lo que saber la capacidad de dichos bancos es muy importante para planear un buen uso de ellos.

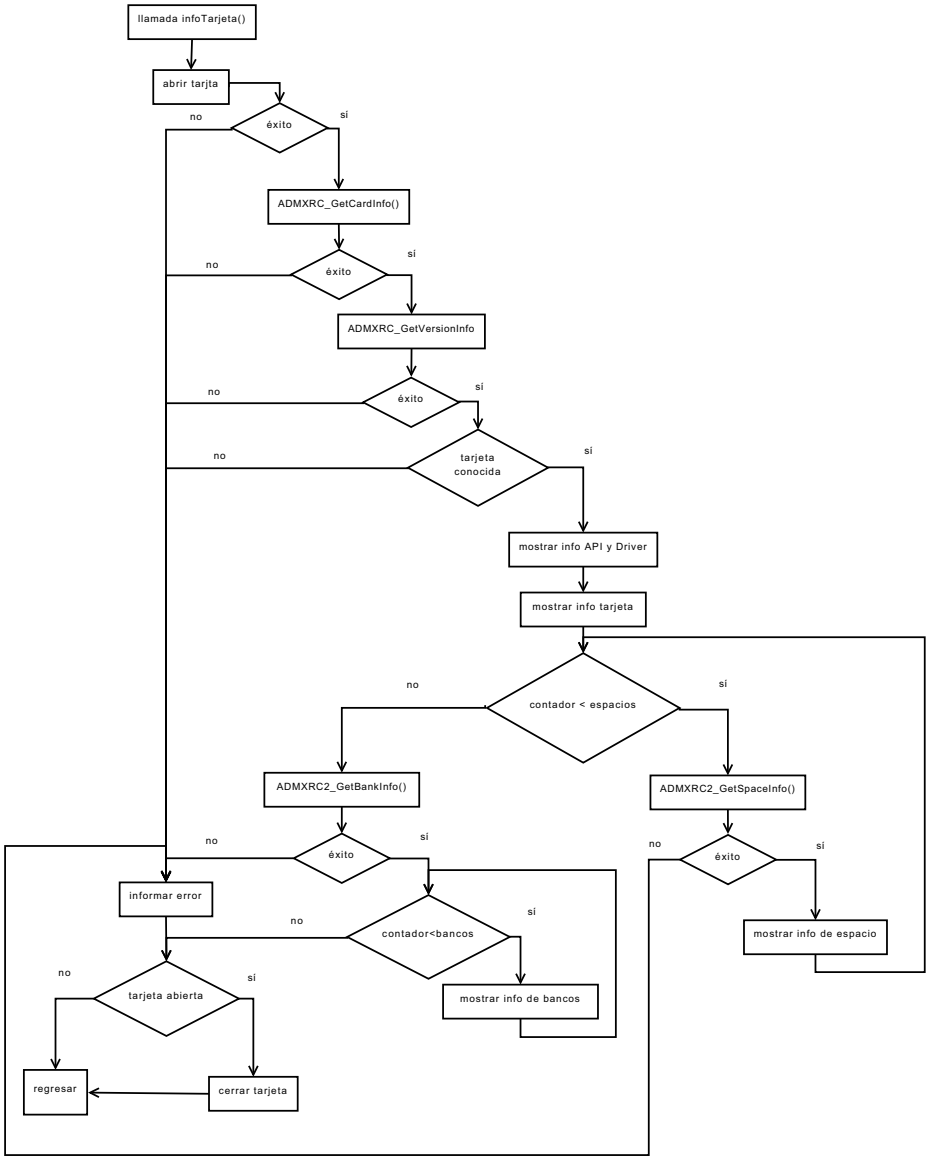

Figura 5.4: Diagrama de flujo de la función para obtener la información de la tarjeta.

5.4.4. Ejecución de la aplicación programada en el FPGA

Una vez programado el FPGA, sólo hay que enviarle los datos para que haga con ellos el procesamiento necesario. Lo primero que debe de hacer es obtener el espacio virtual de direcciones mediante la función **ADMXRC2_GetSpaceInfo** que ya ha sido descrita en este capítulo y que sirve para obtener el espacio de direcciones en el que el programa escribirá y leerá los datos que desea transferir al FPGA. Dado que, mediante esta función se obtiene un apuntador al inicio de dicho espacio de direcciones, en el cual, el programa puede leer y escribir libremente, cada una de sus direcciones está asociada con el espacio de la tarjeta, por lo que, si definimos un apuntador llamado **fpgaSpace**, que contenga la dirección de inicio del espacio virtual, cada dato puede ser manejado direccionando al apuntador como si fuera un arreglo, así, `textitfpgaSpace[0]` sería la dirección 0 del FPGA, `textitfpgaSpace[1]` la dirección 1, etc.

El diagrama de flujo de la opción ejecutar del programa se puede observar en la Figura 5.5. Esta opción, que en el programa designamos como **-x**, sólo consiste en enviar los datos al FPGA y terminar.

Es importante, durante la ejecución de la aplicación observar que, aunque el programa ha enviado los datos y terminado, el FPGA, como dispositivo físico, sigue funcionando independientemente de si un programa está o no accediendo a él. Por lo que los datos que ha manejado y almacenado en registros, continúan ahí. Por esta razón, el programa ha sido diseñado para mostrar las salidas que el FPGA ha producido, en la opción **-o**, con el propósito de demostrar la manera en la que funciona el FPGA y que, mientras este siga conectado y encendido, el hardware programado seguirá funcionando, una vez que se apague, el FPGA perderá la información y la aplicación se borrará. La Figura 5.6 muestra el digrama de flujo de los método mostrarSalida().

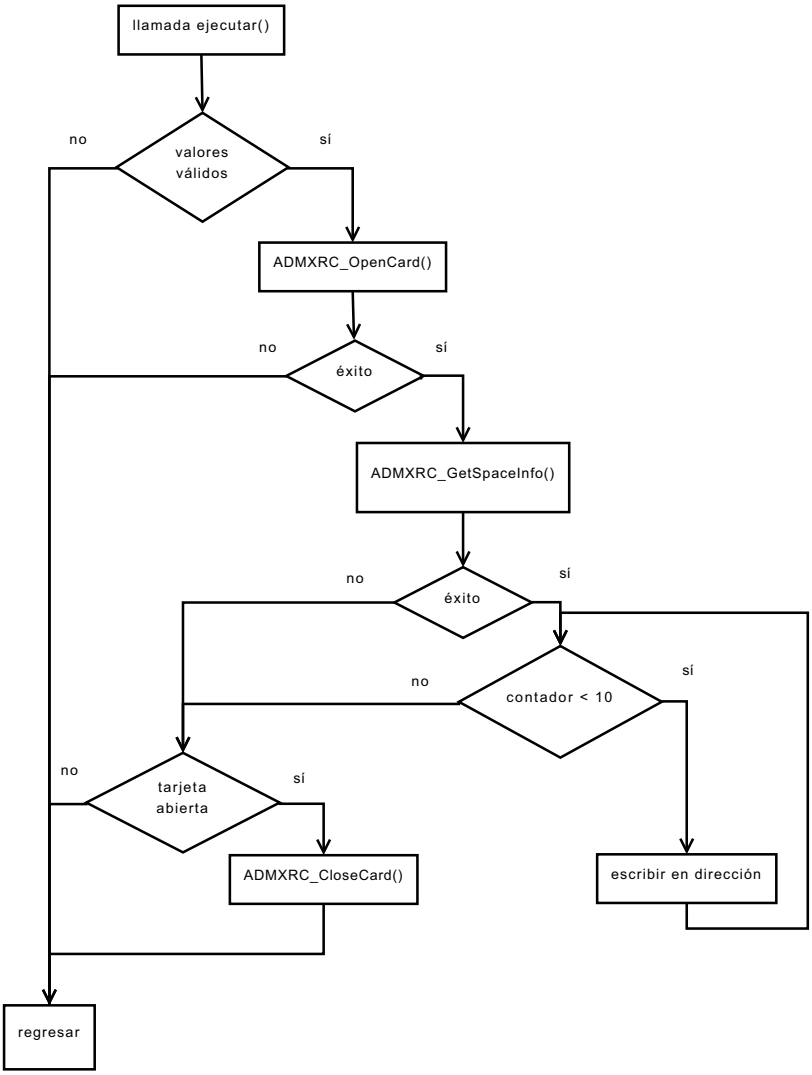

Figura 5.5: Diagrama de flujo de la opción ejecutar.

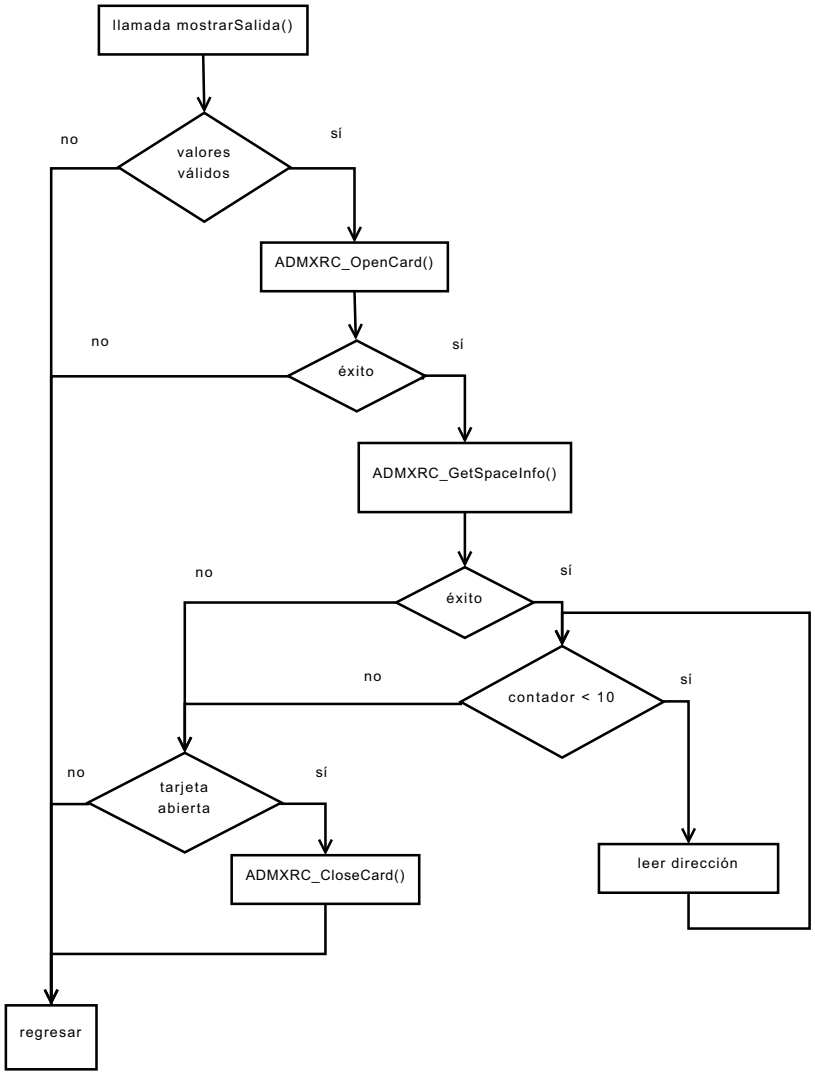

Figura 5.6: Diagrama de flujo de la opción para mostrar salidas.

5.4.5. Configuración del reloj

Como se dijo anteriormente, existe la posibilidad de configurar el generador de reloj de la tarjeta para hacer que la aplicación programada en ella trabaje a una frecuencia específica. La configuración del reloj se realiza mediante la función **ADMXRC2_SetClockRate**, el cual recibe un valor de tipo double para indicar una frecuencia determinada en Hertz, y para el caso de la *ADM-XP*, debe ser entre 6 y 80MHz. Una vez configurado el generador de la tarjeta, la frecuencia se mantendrá así hasta que sea configurada nuevamente, es decir, no importa si se programa el FPGA o se realiza alguna otra acción, el valor de la frecuencia no cambiará. La Figura 5.7, representa el comportamiento de esta función.

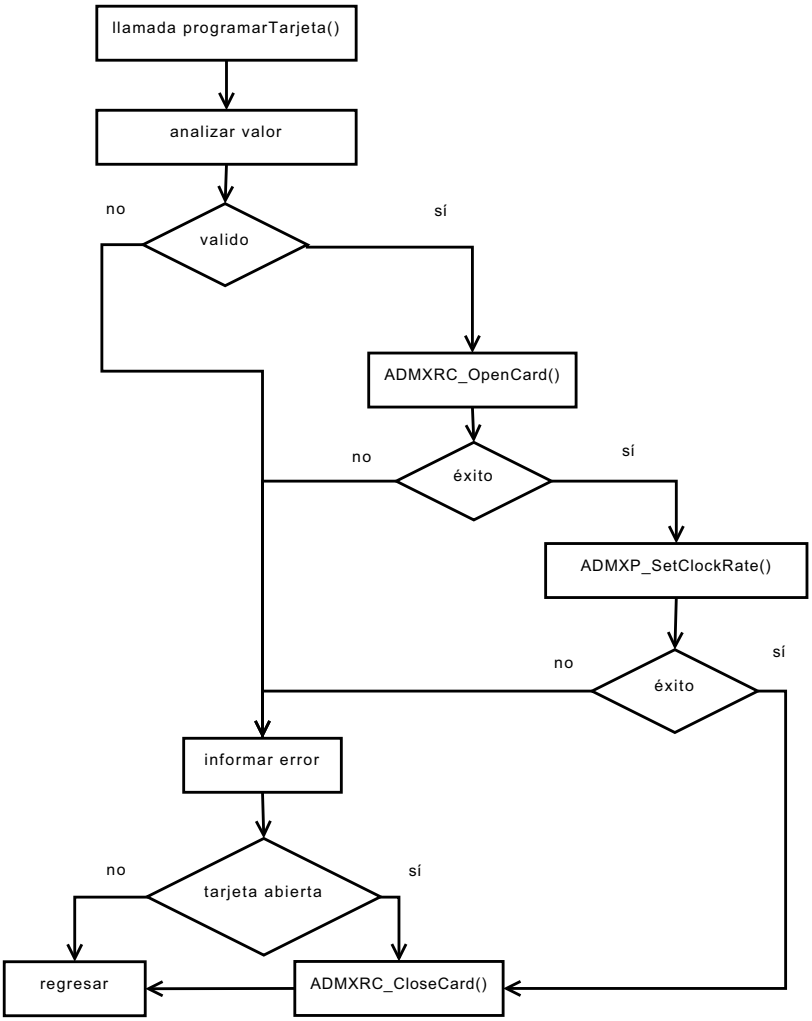

Figura 5.7: Diagrama de flujo de la función para configurar el reloj.

5.5. Un ejemplo sencillo de aplicación: Sumador de 32 bits

Después de establecer las bases para el diseño e implementación del hardware en la tarjeta, podemos dar un ejemplo sencillo de aplicación, lo importante es mostrar cuáles son los pasos importantes para llevar a cabo esta tarea, por lo que no resulta importante el diseño sino la manera en que se debe desarrollar.

Este es un ejemplo sencillo de aplicación, se trata de un diseño que realiza la suma de dos números hexadecimales, siendo recibidos en dos registros distintos. Posteriormente, el resultado de dicha operación es puesto en un registro que será leído y mostrado en la salida del programa.

5.5.1. Descripción del hardware

Primero se debe diseñar un componente nuevo, tal y como se hace normalmente, esta tarea puede ser llevada a cabo con cualquier editor de texto plano, debido a que sólo se requiere describir el hardware usado el lenguaje VHDL. El siguiente código muestra dicho diseño:

```
1
2       entity sum32 is
3               Port (  e1 : in  STD_LOGIC_VECTOR (31 downto 0);
4                       e2 : in  STD_LOGIC_VECTOR (31 downto 0);
5                       sal : out  STD_LOGIC_VECTOR (31 downto 0));
6       end sum32;
7       architecture Behavioral of sum32 is
8               begin
9                       sal = e1 + e2;
10      end Behavioral;
```

Como se puede observar, el diseño sólo consiste en dos datos de entrada de 32 bits y la salida recibe la suma de dichos valores. Esto es posible gracias a la biblioteca STD_LOGIC, que cuenta con operadores aritméticos para trabajar de manera sencilla con los datos de este tipo.

Una vez creado un nuevo proyecto en ISE, se agrega el nuevo componente al proyecto y se debe conectar dentro del diseño básico, que se desarrolló en el capítulo 4 (y cuyos archivos también han sido agregados). La conexión del nuevo componente dentro del diseño básico (archivo design.vhd) se logra añadiendo los siguientes fragmentos de código en el lugar correspondiente.

1. Declarar el componente. Este fragmento de código se escribe antes de la instrucción begin de la arquitectura del hardware.

```
1         component sum32 is
2       Port ( e1 : in  STD_LOGIC_VECTOR (31 downto 0);
3              e2 : in  STD_LOGIC_VECTOR (31 downto 0);
4              sal : out  STD_LOGIC_VECTOR (31 downto 0)
5              );
6         end component sum32;
```

2. Declarar las señales que conectarán con las entradas y salidas del diseño. La declaración de las señales también se escribe antes de la instrucción begin dentro de la arquitectura

```
1       --agregue los registros que necesite
2       signal e_e1: std_logic_vector(31 downto 0);
3       signal e_e2: std_logic_vector(31 downto 0);
4       signal s_sal: std_logic_vector(31 downto 0);
```

3. Conectar los registros que contienen los datos de entrada con las señales que conectan a su vez con las entradas del nuevo componente. Esto puede ir en cualquier parte dentro de la descripción del hardware.

```
1       e_e1 <= reg0;
2       e_e2 <= reg1;
```

4. Conectar las salidas del componente con los registros que contendrán dichos datos. Esto se hace dentro del proceso **genera_ld_o**. Recuérdese que los bits la_q[5:0] indican la dirección a la que se conecta el registro de saida; en este caso

es sólo una salida, la señal **s_sal** cuyo dato sale por la dirección 2, esta señal se escribe en lugar del registro que originalmente estaba conetado, el cual era reg2.

```
1         elsif la_q(5)= '0' and la_q(4)= '0' and la_q(3)= '1' and la_q(2)='0' then
2                 ld_o <= s_sal;
```

5. Hacer el mapeo de puertos del diseño que se ha agregado como componente. Consiste en conectar los pines del nuevo componente con las señales agregadas en la parte 2.

```
1         design1: sum32
2                 Port map (
3                           e1 => e_e1,
4                           e2 => e_e2,
5                           sal=> s_sal
6                         );
```

Los pasos anteriores se pueden hacer con cualquier componente que no contenga entradas y salidas que, juntas superen a los diez registros de los que dispone el diseño genérico de este proyecto, sin embargo, dado que cada registro es de 32 bits; si las salidas no son más que de unos cuántos o incluso sólo un bit, un sólo registro puede contener varias salidas juntas; y cuando se muestra la salida de la aplicación en el programa, el usuario podrá interpretar los datos observando cada byte de dicho registro, ya que los resultados están en formato hexadecimal.

Realizadas las anexiones anteriores, con *Xilinx ISE* se genera el archivo *bitstream* y se ejecuta el comando para programar la tarjeta, introducir los valores y observar las salidas.

5.5.2. Prueba con el comando en Linux

Tal como se hizo anteriormente, se ejecutó el comando admxpprg con las opciones correspondientes (-c<elArchivo> para programar y -xo para ejecutar y ver la salida), la Figura 5.8 muestra el resultado de este procedimiento. Como se puede observar, se introducen diez valores, de los cuales, los dos primeros son los que se desea sumar, los demás datos pueden ser ceros o cualquier valor, ya que el diseño programado no los toma en cuenta en este ejemplo, una vez introducidos los 10 valores, el programa muestra todos los registros, entre los cuales, se puede observar que el tercer registro contiene el resultado de la suma, los demás registros contienen los mismos datos que fueron introducidos, pues durante la conexión del componente con el diseño básico, estos no fueron modificados.

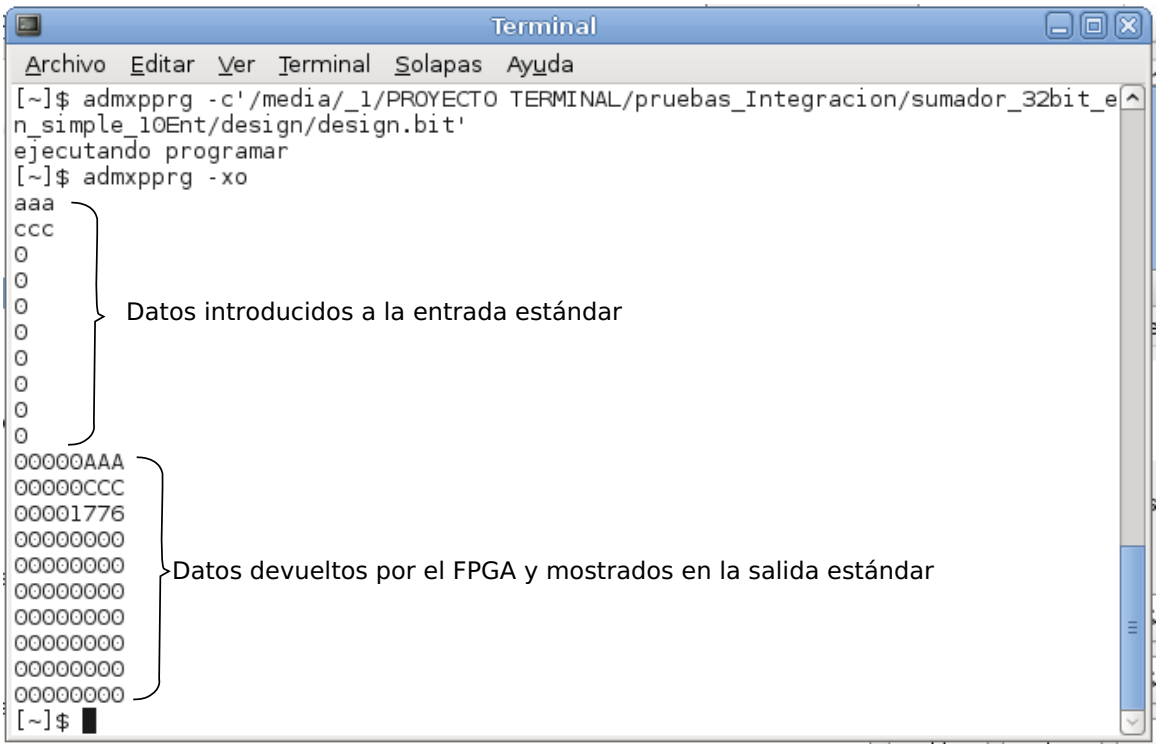

Figura 5.8: Uso del sumador mediante el comando admxpprg.

Ciertamente existen muchas maneras de diseñar un circuito sumador, por ejemplo, en lugar del tipo STD_LOGIC_VECTOR, se puede utilizar el tipo BIT, y hacer la suma bit a bit, de manera combinacional o serial, esta opción de implementar un sumador tiene la ventaja de que se pueden obtener obtener las banderas que indican el estado del resultado, es decir si el resultado fue cero, si hubo un desbordamiento, si el resultado es negativo, en otras palabras, la forma en la que opera una unidad aritmética lógica, sólo se necesita tener en cuenta que el tipo de dato de las entradas y salidas del componente deben ser transformadas usando las funciones **to_bit**, **to_stdulogic**, **to_bitvector** y **to_stdlogicvector** , que se encuentran en la biblioteca **std_logic_1164**.

Capítulo 6

Programación de la Interfaz Gráfica de usuario

6.1. Introducción

El programa **admxpprgui** es una interfaz gráfica que ayuda a acceder a la funcionalidad del programa admxpprg de manera más sencilla, pues para hacer uso de la aplicación programada en el FPGA, sólo se requiera hacer click en algunos botones, sin la necesidad de escribir varios comandos en una terminal, esto resulta útil si se quiere ejecutar varias veces seguidas la aplicación. Por ejemplo, como ya se describió en el capítulo anterior, para ejecutar la opción de introducir valores al FPGA, estos tiene que ser escritos en la entrada estándar de la terminal, tarea que se tendría que llevar a cabo cada vez que se ejecute el programa, por lo que poner los datos en una interfaz, la cual conserve esos datos y sólo haya que oprimir un botón cada vez que se quiera transferirlos al FPGA resulta mucho más sencillo. En este capítulo se hablará sobre la programación de la interfaz gráfica.

6.2. Características generales de la interfaz gráfica

Una vez que se hizo el programa admxpprg y se observó su correcto funcionamiento se diseño el uso de este comando por medio de una interfaz gráfica, es decir, que cuando el usuario manipule los controles de la interfaz, esta se encargue de ejecutar el comando con los argumentos correspondientes, introducir los valores y mostrar resultados al usuario, esto proporciona una manera fácil y rápida de realizar el trabajo con el FPGA.

La ventaja de crear la interfaz gráfica de manera independiente al comando, es que puede diseñarse toda la lógica de la aplicación sin tener que pensar en otras cuestiones, como paso siguiente, sólo enfocar esfuerzos en el diseño de la interfaz. De esta manera, el trabajo se realiza de una forma más ordenada, y el código y estructura de cada aplicación es más entendible. Con respecto a los cambios, si se

quiere realizar cambios en la Interfaz, sólo serían cuestiones de presentación de la información obtenida por el comando, sin que esto afecte de manera directa en su funcionamiento; si se quiere realizar cambios en el programa que lleva la lógica de la aplicación, puede ocurrir que le sean agregadas nuevas funcionalidades, cosa que no impacta tampoco en el funcionamiento de la interfaz, pues el resultado es que la interfaz no tenga posibilidad de acceder a dichas nuevas opciones en el comando, pero sí seguir dando al usuario el uso de las opciones originales. Siendo de esta forma, sólo se necesita extender las características para la interfaz gráfica de modo que pueda hacer uso de las nuevas opciones en el comando.

La interfaz gráfica fue programada utilizando la biblioteca Qt, hoy día es ampliamente utilizada en muchas aplicaciones, no sólo en aplicaciones para Linux, sino también para otros sistemas operativos, por estar construida en lenguaje C++, lo que le da la característica de ser multiplataforma. Esto da a la interfaz gráfica de este proyecto la posibilidad de ser portable a otros sistemas operativos sin mucha complicación. Además, el hecho de estar desarrollada bajo C++, implica que toda su funcionalidad se encuentra bajo el paradigma orientado a objetos, lo cual también es una ventaja para una fácil programación.

El diseño de una interfaz gráfica aceptable, requiere de muchos conocimientos sobre la biblioteca, es decir tipos de datos, clases y métodos que se necesitan para dar a la interfaz un comportamiento y presentación aceptables. Sin embargo, gracias al uso de entornos integrados de desarrollo (*Integrated Development Environment, IDE*), hoy en día el diseño de interfaces gráficas se ha convertido en una tarea más sencilla, puesto que el IDE se encarga de los detalles de programación de la presentación, y el programador de los detalles del funcionamiento de la interfaz. Para realizar la interfaz se utilizó Qt Creator, la cual es una herramienta desarrollada por la compañía *Trolltech* para el desarrollo de aplicaciones con las bibliotecas Qt.

Dado que el desarrollo de interfaces gráficas mediante un IDE se convierte en una tarea sencilla, esta tarea no tuvo gran complicación. Sin embargo, sí se tuvo que hacer una investigación en cómo lograr que la interfaz hiciera uso del comando **admxpprg** o de cualquier proceso en general, enviando y recibiendo datos de él de una manera transparente, por lo que se tuvo que pensar en varios métodos y funciones para la comunicación entre procesos como opción para llevar a cabo este objetivo. Al final, la comunicación entre la interfaz y el programa se logró haciendo uso de la clase **QProcess**, la cual es una clase de la librería Qt diseñada específicamente para este fin. Gracias a esta clase, se pueden aprovechar el método **start**(), al que se le indica el proceso que se quiere ejecutar y los argumentos que se deseen, así como los metodos **write()** y **readAllStandardOutput()** para escribir y leer de la entrada y salida estándar respectivamente[11].

6.3. Diseño de la Interfaz

En cuanto al diseño de la interfaz, se pensó en que fuera una herramienta muy sencilla e intuitiva de usar, por lo que, en una sola ventana, se integró el acceso a

todas las opciones que proporciona el pograma **admxpprg**, y de las cuales se hablará a continuación.

6.3.1. Área principal

El diseño contiene principalmente dos elementos, uno de tipo QTabWidget y un QTextEdit. QTabWidget es una clase que permite mostrar varias áreas de trabajo en un mismo lugar separadas a modo de pestañas, por lo que, cada pestaña se diseño para realiza tres funciones principales: programar el FPGA, ejecutar la aplicación del FPGA, y obtener la información de la tarjeta.

El segundo elemento principal que contiene la tarjeta es un área de texto, en la que se puede observar el comando ejecutado por la interfaz con sus respectivos argumentos. Es similar a una terminal en la que se puede ver la ejecución del programa principal, y los mensajes de éxito o error qué este genere durante su ejecución. Muchos programas en la actualidad funcionan de manera similar, pues también ejecutan procesos externos y sus respectivos mensajes a la salida estándar son mostrados en un área dedicada para este fin. Eclipse, Qt Creator o Xilinx ISE, entre otros, son un ejemplo de este tipo de aplicaciones, razón que motivó a dar a esta aplicación un diseño similar para dar al usuario la sensación de familiaridad con el programa.

6.3.2. La Pestaña programar

En esta pestaña se puede ejecutar el comando con las opciones de programar la tarjeta y configurar el reloj (-c y -k). La Figura 6.1 muestra la interfaz funcionando al manejar esta pestaña, cabe mencionar que la opción de configurar el reloj es opcional. Al oprimir el botón programar se ejecuta el comando con la opción **-c** y la ruta del bitstream, si fue introducido un valor para el reloj, este también será configurado, sin embargo, si no se introduce un archivo para programar la tarjeta, la interfaz no realizará la ejecución del programa.

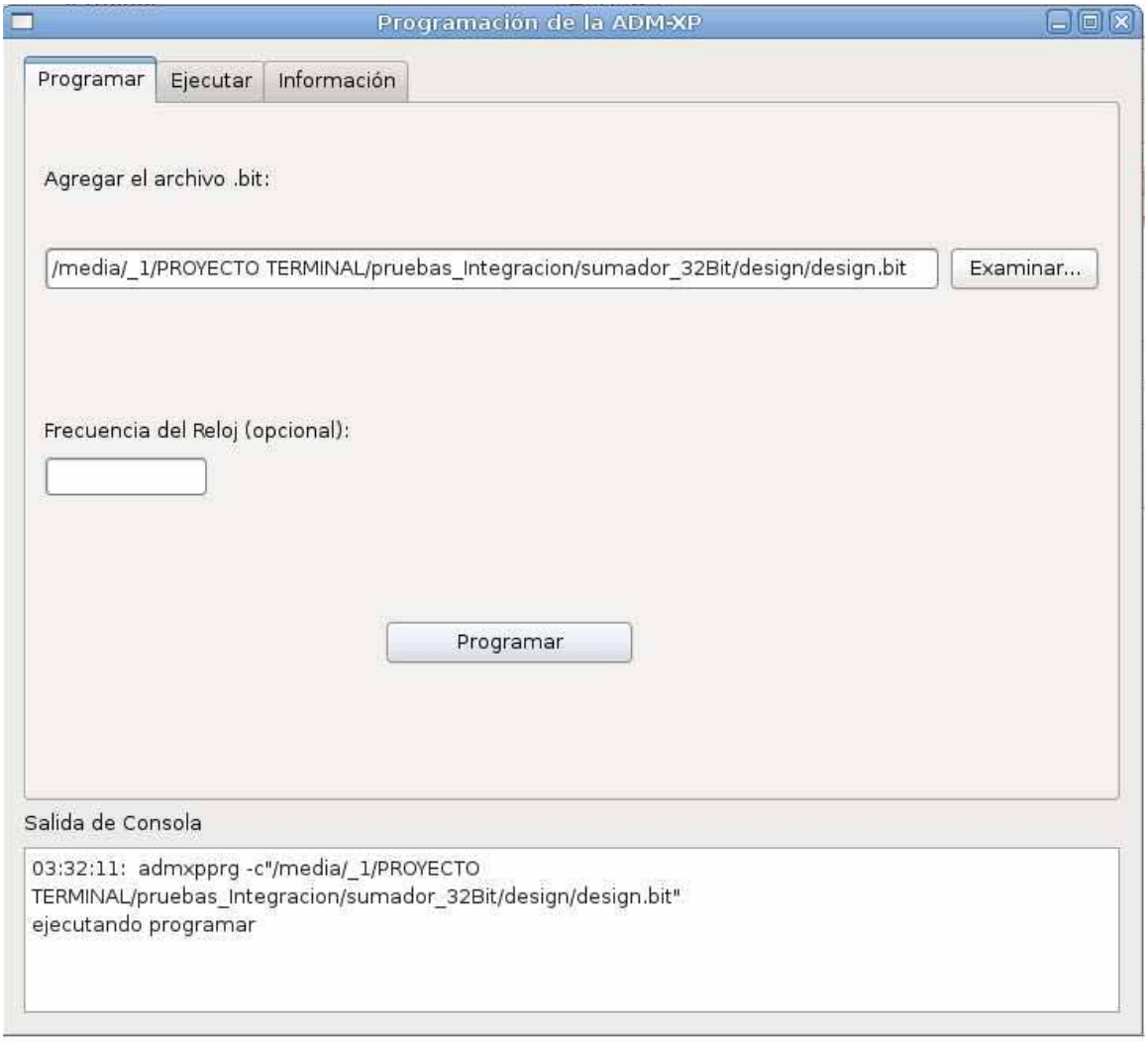

Figura 6.1: Pestaña de la interfaz para programar la tarjeta.

6.3.3. La pestaña ejecutar

En esta pestaña se puede ejecutar el comando con las opciones de ejecutar y mostrar salidas (**-x** y **-o**). Al introducir los diez valores de entrada en los cuadros de texto, se puede oprimir el botón **Ejecutar**, con esto, mediante el método **write** heredado por **QProcess** de la clase **QIODevice**, la interfaz envía a la entrada estándar del comando, los diez valores tal y como si los hubiera introducido el usuario por la terminal. El Área de texto que se incluye en esta pestaña, sirve para mostrar las salidas del FPGA, esto ocurre cuando se oprime el botón **Ver salida**. Dado que no todos los registros que se mostrarán son salidas del hardware programado, le será más útil al usuario ver solamente aquellos registros que sí lo sean, por lo que cada registro cuenta con una casilla que sirve para indicar que este sea mostrado al oprimir el botón para ves las salidas, de esta forma, los registros que no sean marcados, no

se mostrarán en el área de texto. La Figura 6.2 muestra cómo se ve esta pestaña durante su ejecución, también se puede observar que la aplicación programada es la del sumador de 32 bits, desarrollada en el capítulo 5, las entradas fueron introducidas en los registros 0 y 1; mientras que la salida es mostrada en el registro 2.

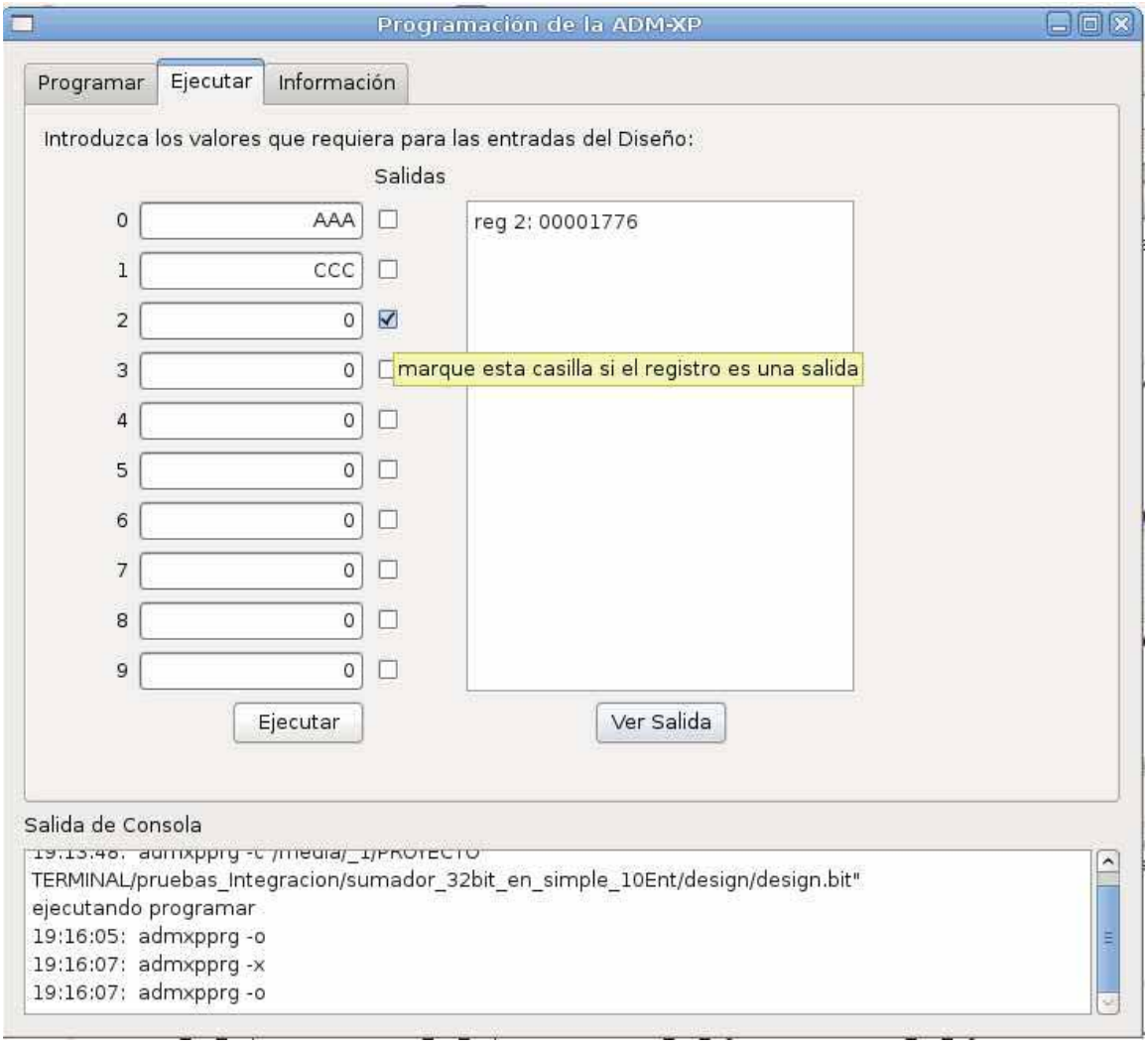

Figura 6.2: Pestaña de la Interfaz para ejecutar la aplicación del FPGA.

6.3.4. La pestaña de información

Al ser seleccionada, esta pestaña muestra la información de la tarjeta, ejecutando el comando con la opción **-i**, una vez ejecutado sólo redirecciona la salida estándar del programa al área de texto, que es el único elemento que conforma a esta pestaña. El comando ejecutado se muestra en el área de texto de la consola. Esta parte de la interfaz en ejecución se puede observar en la Figura 6.3.

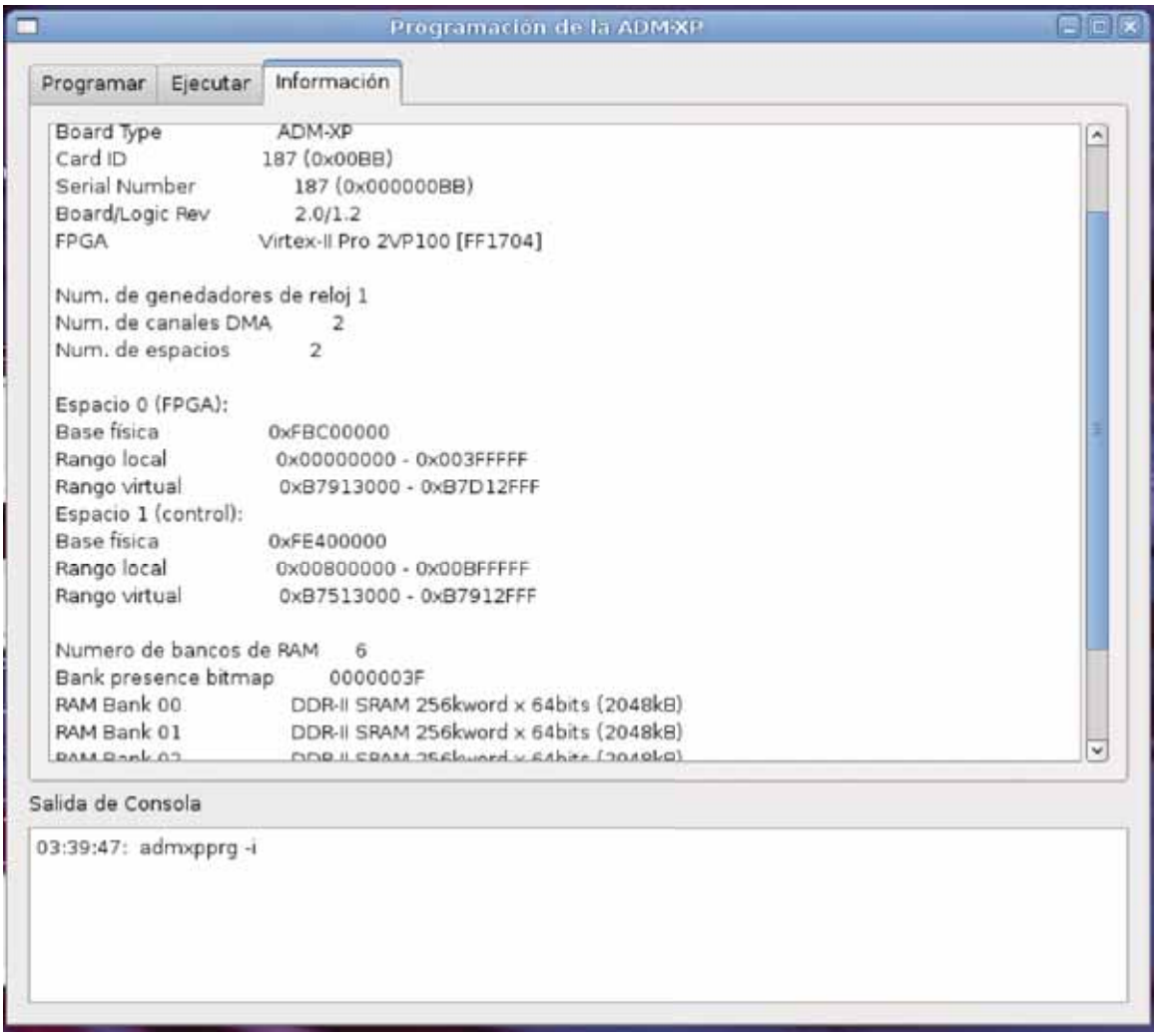

Figura 6.3: Pestaña de la Interfaz para mostrar la información de la tarjeta.

Capítulo 7

Conclusiones y trabajo Futuro

7.1. Conclusiones

Si bien es cierto que las funciones del API de programación que vienen en el *SDK* son fáciles de entender y utilizar, no lo es tan sencillo para alguien que no está familiarizado con el concepto de estructuras en lenguaje C, pues el manejo de estas junto con los apuntadores es un tema difícil de aprender para la mayoría de los estudiantes de programación, es ahí donde se ha justificado el que este proyecto requiera de un ingeniero en computación, además, integrar estas funciones en un programa también requieren de una persona con estas habilidades, sobre todo cuando se trata de un sistema operativo como Linux, que hoy en día ofrece una vasta plataforma en la que existen cientos de herramientas útiles para la ingeniería, e incluso, para otras ramas del conocimiento. Sin embargo, también es cierto que comprender el protocolo de comunicación entre el puente de bus local y el FPGA, y la manera en la que este es aplicado en un diseño de hardware para la tarjeta de desarrollo es una tarea que resulta difícil para un estudiante de computación, el cual se encuentra poco familiarizado con cuestiones que son más propias de la electrónica. Sin embargo, al observar las grandes posibilidades que este tipo de tecnologías ofrecen, se hace más evidente la necesidad de aumentar los esfuerzos por aumentar los conocimientos sobre el tema sin importar la especialidad que se tenga.

7.2. Trabajo Futuro

En cuanto a la continuación de este proyecto, pueden realizarse muchas ampliaciones al comando admxpprg, es decir, se le pueden agregar otras opciones, que permitan manejar distintos elementos de la tarjeta de desarrollo, como los módulos de RAM, o los puertos incluidos en ella, o también se podría agregar alguna función que permita utilizar los procesadores incrustados en en FPGA, además de incluir otros tipos de transferencias de datos, como el uso de DMA, o poder transferir ráfagas de datos en lugar de sólo escribir un dato a la vez en un sólo registro en el FPGA, esto implicaría

también rediseñar la descripción de hardware para el FPGA, de modo que permita distintas opciones, esto se puede lograr agregando direcciones de bus en las cuales realice las distintas funciones que se deseen y para que el usuario pueda agregar sus propios módulos dentro de este diseño. Asimismo, la interfaz gráfica puede ser ampliada, mediante QtCreator y de manera sencilla, para utilizar las nuevas opciones implementadas en el comando. Por otro lado, este proyecto podría ser migrado al sistema operativo Windows con el objeto de que la aplicación sea no exclusiva de un sistema operativo. Puesto todas las herramientas utilizadas para este proyecto son portables, el único detalle, sería observar las características del código, propias de Linux, que requieran ser adecuadas.

En general, el uso de un FPGA proporciona muchas posibilidades para proyectos de ingeniería. El estudio que se realizó, con respecto al desarrollo de un diseño en VHDL para el PGA, fue una tarea compleja, pues requirió de estudiar e interpretar la extensa información contenida en varios manuales para explicar muchas cosas que vienen en este reporte, por lo tanto, se espera proporcionar, para futuras referencias y nuevos proyectos, además de la información que proporciona el manual del fabricante, una base documental en la cual se puedan basar los próximos usuarios de la tarjeta de desarrollo *ADM-XP* sin tener que recurrir a dichos manuales, es decir, una guía rápida, la cual pueda indicar los primeros pasos en su utilización y así, obtener con ello, un importante ahorro en tiempo de estudio y pruebas. Se considera que esto le da un importante valor al desarrollo de este proyecto, pues antes de él, además de la proporcionada por el fabricante, En este centro de estudios no se tenía ninguna información interpretada y simplificada sobre el uso de esta tarjeta. Además, toda la información contenida en el ADM-XRC *SDK*, se pudo observar que está diseñada para personas que ya poseen una cierta cantidad de conocimientos sobre el tema. En muchos centros de estudios no se cuenta con el tiempo suficiente para estudiar un manual o varios de ellos y en seguida estar desarrollando aplicaciones, pues la duración de una materia o un proyecto en ocasiones es muy reducida, como en el caso de la Universidad Autónoma Metropolitana, unidad Azcapotzalco, la cual maneja sus planes de estudio en trimestres, y aún más será el tiempo que tomará cuando la información viene en otro idioma. Es por eso, que este proyecto se considera de gran utilidad para aquellas personas que deseen desarrollar nuevos proyectos en el amplio rango de aplicación que tiene la tecnología FPGA. Como ejemplos podemos mencionar la codificación Hardware-Software (CHS), el cómputo intensivo y los sistemas embebidos [3].

Apéndice A

Detalles Sobre las herramientas utilizadas

A.1. Introducción

En este apéndice se da una breve descripción sobre el uso de las herramientas utilizadas para llevar a cabo este proyecto, así como problemas que se encontraron y las soluciones que se encontraron. Es muy importante señalar las versiones de cada aplicación pues el uso de otras versiones de cualquiera de las herramientas aquí mencionadas no garantiza el funcionamiento de algún aspecto del proyecto realizado.

A.2. Herramientas utilizadas

Para desarrollar este proyecto se utilizaron los siguientes recursos:

- Arquitectura del procesador: 32 bits de intel
- Sistema Operativo: Debian 5
- Aplicaciones disponibles en la página de Alpha-Data: `ftp://ftp.alphadata.co.uk/pub/admxrc/linux/old/`
  - Driver de la tarjeta: ADM-XRC Device Driver versión 2.11.0
  - ADM-XRC *SDK* versión 2.7.1
- Proporcionado por el área de Sistemas digitales de la Universidad Autónoma Metropolitana, unidad Azcapotzalco.
  - Tarjeta de desarrollo: *ADM-XP*
  - Xilinx Ise,versión 8.2i

- Aplicaciones disponibles en los repositorios de Debian
  - g++ 4.3.2
  - qt 4.4.3
  - qmake 2.01a
  - automake 1.10.1
  - autoconf 2.61
- Qt Creator versión 1.2.1 (con qt 4.5), disponible en: `http://qt.nokia.com/downloads/downloads`

A.3. Problemas con las herramientas utilizadas y prevención de errores

A.3.1. Las versiones que se recomienda utilizar

La herramienta utilizada para generar el archivo de configuración fué Xilinx ISE versión 8.2i. Durante el desarrollo de este proyecto terminal se trabajó también con la versión 9.2i, sin embargo, la aplicación EDK, que se usa para crear sistemas en un sólo chip, no soportan tarjetas como la *ADM-XP* que son hoy en día obsoletas, por lo tanto, la versión 8.2i es la que se utilizó, pues se piensa que en un futuro podrían hacerse proyectos pensados en este tipo de aplicaciones.

El *SDK* viene con *scripts* para crear automáticamente los proyectos de los diseños de ejemplo para Xilinx ISE. Durante el desarrollo de este proyecto, como parte de la investigación se probó crear dichos proyectos. Es importante remarcar que los *scripts* de la versión 2.7.1 del *SDK* están diseñados para crear proyectos para ISE 8.2i, por lo tanto, estas versiones se consideraron las más adecuadas.

A.3.2. Generación automática de proyectos para Xilinx ISE

La creación de proyectos de ISE, ayuda al proceso de aprendizaje en la creación de un diseño para la tarjeta de desarrollo, el *SDK* viene con unos *scripts* que se encargan de generar los proyectos de manera automática para los diseños de ejemplo, por lo que a continuación, se presentan algunos detalles sobre este tema.

Un problema con la creación de los proyectos para los diseños de muestra, fue que necesitaban el archivo **projnav.tcl**, que por error Alpha-Data omitió en el *SDK*. Dicho problema se solucionó contactando al soporte técnico, quienes enviaron el archivo correcto, si se desea obtener este archivo se recomienda ponerse en contacto el autor de este proyecto (bleusciel@gmail.com) o contactar de igual manera al soporte técnico de la compañía Alpha Data.

Cuando se desea usar los *scripts* para generar proyectos para ISE de los diseños de muestra, en el directorio de cada diseño, hay un archivo llamado **mkproj.tcl**,

que contiene las instrucciones para generar los proyectos para cada modelo de tarjeta soportada por el *SDK*, por lo que se recomienda eliminar las líneas de código para todo modelo de tarjeta, a excepción de la *ADM-XP*, de esta manera el *script* no tardará demasiado en ejecutarse y no generará archivos innecesarios. El archivo (en el caso del diseño de muestra, llamado "simple") quedaría de la manera siguiente:

```
1 source ../../../projnav.tcl
2 set _cwd [pwd]
3 set _VHDMod $::projNav::VHDMod
4 set _VHDPkg $::projNav::VHDPkg
5 set _design "simple"
6 set _entity "simple-mixed"
7 set _projects [ list \
8         [ list \
9                 virtex2p 2vp100 ff1704 5 xp [ list \
10                        [ list "../../../../common/localbus/localbus_pkg.vhd" $_VHDPkg ] \
11                        [ list "../../../../common/localbus/plxdssm.vhd" $_VHDMod ] \
12                        [ list "../../../simple-xpl.vhd" $_VHDMod ] \
13                        [ list "../../../simple-xp.ucf" "simple" ] \
14                ] \
15        ] \
16 ]
17 ::projNav::makeProjects $_cwd $_design $_entity $_projects 1
```

A.3.3. Variables de ambiente

Es muy importante en Debian la variable de ambiente ADMXRC_SDK para utilizar las aplicaciones de muestra del *SDK* y los *scripts*. También el Makefile del programa utiliza esta variable de ambiente, pues en el directorio de instalación del *SDK* se encuentra la biblioteca admxrc2.h donde se definen las funciones de la API. Para agregar la variable de ambiente, se debe escribir en el archivo /etc/profile la siguiente línea:

export ADMXRC_SDK4=<la ruta en la que se instaló el *SDK*>

Las variables de ambiente se pueden actualizar escribiendo el comando

$source /etc/profile

También se debe agregar la variable XILINX_TOOLS=<ubicación_de_Xilinx>/bin/lin para usar los *scripts* que generan proyectos para ISE[10].

A.3.4. Paquetes que deben ser instalados en Debian para usar Xilinx ISE

Para poder ejecutar Xilinx ISE versión 8.2i, se debe instalar el paquete **libstdc++5**, disponible en los repositorios de Debian 5.

Al instalar Qt Creator en Debian se necesitó instalar los paquetes que se listan a continuación, pues de otro modo no se podía ejecutar.

- libfreetype6-dev
- libglib2.0-dev

- libsm-dev
- libxrender-dev
- libfontconfig1-dev
- libxext-dev

A.4. Simulación usando ISE versión 8.2i

Xilinx ISE tiene la posibilidad de simular los diseños en VHDL, si bien no resultaría fácil simular un diseño para la *ADM-XP* debido a todas las señales de entrada y salida que maneja, un componente diseñado como se hizo en el capítulo 5, sí puede ser simulado mediante Xilinx ISE. Sin embargo, durante la realización de este proyecto terminal, se encontró con un error al querer hacer una simulación, el cual se muestra a continuación:

ERROR: Simulator: 222 - Generated C++ compilation was unsuccessful.

Este error se debe a que Xilinx ISE trae por defecto una versión propia de g++, la cual genera este error. La forma de solucionarlo es la siguiente:

1. Ir al direcorio en el que se encuentra instalado Xilinx ISE entrar al subdirectorio gnu/gcc/3.2.3/lin/lib/gcc-lib/i686-pc-linux-gnu/3.2.3
2. En el archivo de nombre “specs”, reemplazar todas las ocurrencias de “-lc” por “-lcxil”
3. Guardar una copia de “/usr/lib/libc.so” como “gnu/gcc/3.2.3/lin/lib/gcc-lib/i686-pc-linux-gnu/3.2.3/libcxil.so” en eldirectorio raíz de Xilinx
4. Editar la última línea para remover la instrucción AS_NEEDED y el correspondiente par de paréntesis para dejar esta línea como se muestra a continuacion: “GROUP (/lib/libc.so.6 /usr/lib/libc_nonshared.a /lib/ld-linux-x86-64.so.2)” [12]

Apéndice B

Guía de instalación y uso del sistema

B.1. Introducción

Con los siguientes pasos descritos, el usuario final obtiene dos comandos ejecutables tal y como normalmente se tienen en Linux, es decir, que no se tiene que especificar la ruta completa del programa, sino que este está instalado en el directorio **/usr/bin** de modo que sólo tiene que escribir el nombre del comando para que se ejecute.

B.2. Instalación previa del Driver de la tarjeta

Primero se debe instalar el driver de la tarjeta, de acuerdo con las instrucciones que vienen con él, sin embargo, es importante señalar que el driver, se comporta como un servicio al que se tiene que iniciar de forma manual. En Debian, este paso se realiza con la siguiente instrucción en la línea de comandos:

$/etc/init.d/admxrc2 start

Se usa la misma instrucción para detenerlo, pero con el argumento **stop**.

Estas indicaciones también vienen en el manual del driver, sin embargo, dado que es un paso fundamental para el funcionamiento del sistema, se ha incluido también en esta guía.

B.3. Generación e instalación del comando admxp-prg

Para dar al comando **admxp** una mayor portabilidad, se utilizó **automake** y **autoconf** para generar el Makefile y así compilarlo, generar el archivo ejecutable e

instalarlo, así como incluir el manual correspondiente. Los pasos a seguir fueron los siguientes:

1. Crear fuentes y el archivo Makefile.am, este lleva los nombres de los fuentes
   - Se deben poner archivos .h y .c(pp), de entrada parece que no importa el orden de ninguno
   - Poner también el archivo del manual
2. Ejecutar **autoscan**, que genera el archivo configure.scan
   - genera **autoscan.log**, **configure.scan**
3. **configure.scan** se renombra como **configure.ac**
4. Ejecutar **autoheader**, genera un archivo llamado **config.h.in**, también una carpeta **autom4te.cache**
5. Se le agrega la macro AM_INIT_AUTOMAKE al archivo **configure.ac**, justo después de AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
6. Ejecutar **aclocal**, que sirve para agregar la macro del paso 5
   - genera el archivo **aclocal.m4**
7. Ejecutando **automake** --add-missing --copy se copian los siguientes archivos:
   a) AUTHORS
   b) INSTALL
   c) README
   d) NEWS
   e) COPYING

   Y genera los siguientes archivos:
   a) ChangeLog
   b) depcomp
   c) install-sh
   d) Makefile.in
   e) missing
8. Ejecutar **autoconf**

- sólo genera **configure**

Dado que los pasos anteriores fueron llevados a cabo, el usuario final sólo necesita ejecutar lo siguiente:
$./configure
$make
$sudo make install

Una vez instalado, el usuario puede ver la ayuda del comando escribiendo el comando con la opción **--help** o con el siguiente comando:
$man admxpprg

B.4. Generación e instalación de la Interfaz Gráfica

Por su parte, a pesar de desarrollar la interfaz gráfica, también se generó el archivo Makefile para su compilación mediante el comando **qmake**, los pasos realizados fueron:

en el directorio del proyecto, dado que ya existe el archivo de extensión .pro, solamente se requiere ejecutar el comando:

$qmake <elProyecto>.pro

Con esto se genera el archivo Makefile, al cual se le puede agregar la siguientes líneas para que se pueda instalar:
INSTALL:
cp <ruta del archivo>/admxpprgui /usr/bin

El usuario final, debe realizar sólo las siguientes instrucciones**:**
$make
$sudo make install

Apéndice C

Guía para la implementación de un diseño

La siguiente Guía, muestra más detalles sobre los pasos necesarios para llevar a cabo la creación de un diseño y asimismo muestra en forma resumida el proceso desde la creación de un diseño hasta su implementación y prueba.

El procedimiento está probado en Debian 5 (lennny) y el éxito en cada uno de los pasos siguientes no se garantiza en otras distribuciones.

C.1. Creación del diseño con Xilinx ISE 8.2i

1. Para arrancar Xilinx se deben escribir en una terminal las siguientes instrucciones:

   $source <ruta de Xilinx ISE>/settings.sh

   $<ruta de Xilinx ISE>/bin/lin/ise

2. Crear un proyecto. Hacer click en File -> New Proyect. Nombrar el proyecto y observar bien dónde se van a guardar los archivos y dar click en **next**.

3. Se debe tener cuidado de seleccionar bien las características del FPGA, la Figura C.1 muestra exactamente cómo cómo deben, ser, se puede dar click en **next** en las siguientes ventanas hasta terminar.

4. Agregue los archivos base, mencionados en el capiítulo 4,

   - **plxdssm.vhd**. Se localiza en:
     `<ruta del SDK>/admxrc_sdk-2.7.1/fpga/vhdl/common/localbus`
   - **localbus_pkg.vhd**. Se localiza en:
     `<ruta del SDK>/admxrc_sdk-2.7.1/fpga/vhdl/common/localbus`

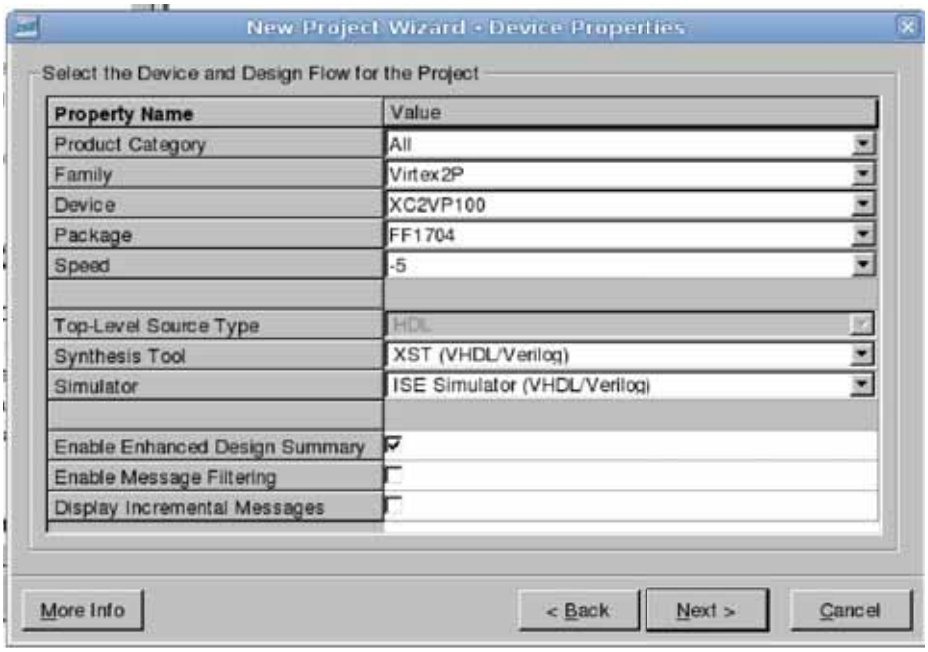

Figura C.1: Selección de las caracter'isticas del FPGA.

- **archivo ucf**. Se puede usar el archivo que está en la ubicación: <`ruta del SDK`>`/admxrc_sdk-2.7.1/fpga/vhdl/simple/simple-xpl.ucf.`
- **design.vhd**. Es el diseño básico que se creó a partir del ejemplo `simplexpl.vhd` proporcionado por el *SDK*. Véase el código fuente del apendice A.

5. Editar el archivo `design.vhd` para agregar el, o los componentes que se quieran implementar. Véase como ejemplo, el archivo fuente del apéndice A.

6. Ejecutar los procesos de chequeo de sintaxis, implementación y generación de archivo de programación. La Figura C.2 muestra que éstos se encuentran en una sección específica. Dichos procesos se ejecutan dando doble click en ellos. Si al ejecutarlos no se obtuvo ningún error, se ha creado satisfactoriamente el archivo de programación.

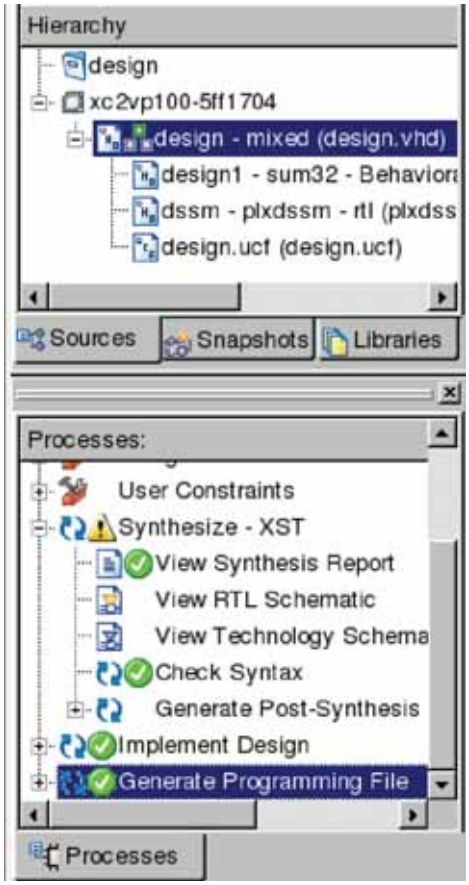

Figura C.2: Vista de los procesos que se pueden ejecutar en Xilinx ISE y de los archivos agregados al proyecto.

La sección de ISE llamada *console* muestra el resultado de la ejecución de los procesos. La Figura C.3 muestra cómo se debe ver cuando el resultado ha sido satisfactorio. Se recomienda mucho la lectura de esta consola cuando se ha producido un error o para obtener información más detallada sobre la ejecución de los procesos.

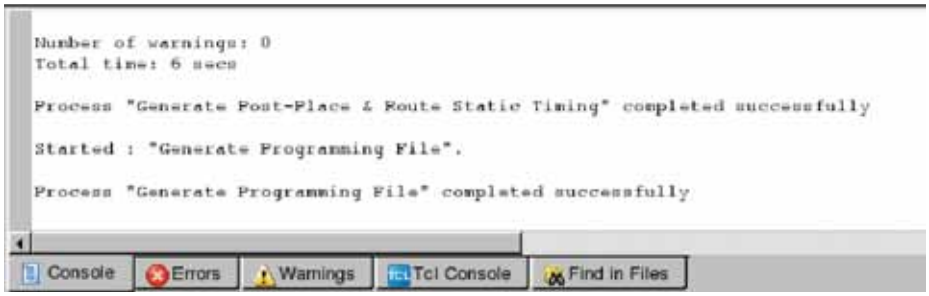

Figura C.3: Vista de la consola que muestra los avances y errores en los procesos ejecutados.

C.2. Prueba del diseño con admxpprg y admxpprgui

1. Asegurarse de tener instalado el driver de la tarjeta y ejecutarlo con la siguiente instrucción:

   ```
   $/etc/init.d/admxrc2 start
   ```

2. Asegurarse de tener instalado el programas admxpprg en la ruta /usr/bin, el programa admxpprgui puede ejecutarse desde cualquier ubicación.

3. Se puede ejecutar el programa admxpprg desde la línea de comandos como muestra la Figura C.4, donde se puede apreciar la programación y uso del sumador de 32 bits, introduciendo los 10 datos en la entrada estándar y la salida del programa, en la que, el resultado de la suma, ha sido almacenado en el tercer registro.

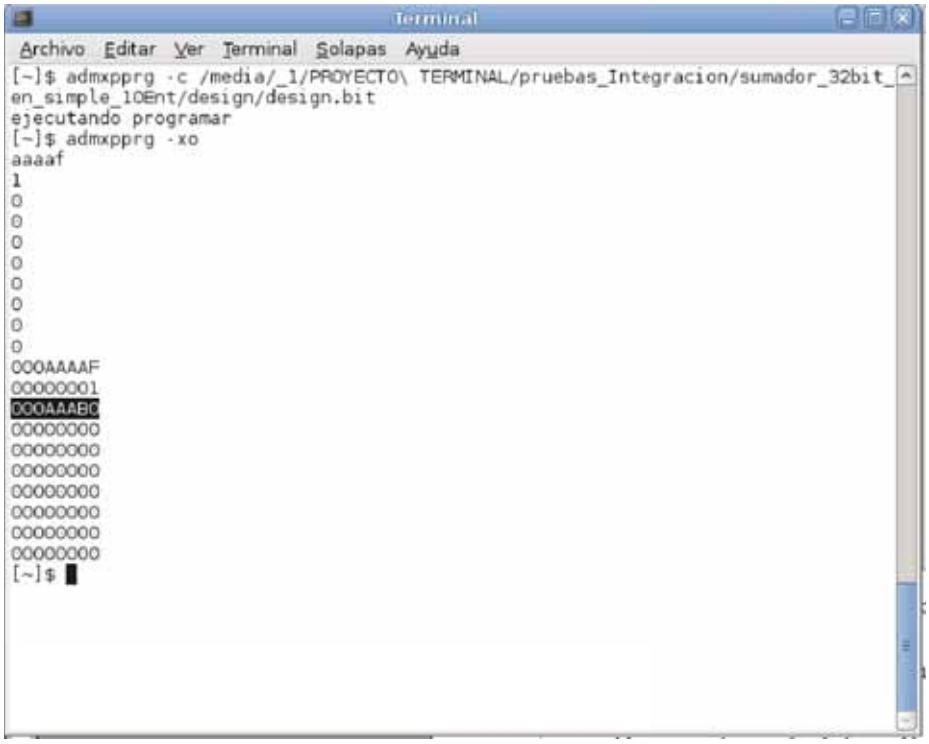

Figura C.4: Ejecución del comando admxpprg para el uso del sumador de 32 bits.

C.2.1. Uso de la interfaz gráfica

1. Iniciar el programa **admxpprgui**, si está instalado en `/usr/bin`, basta con ejecutarlo como sigue:

   ```
   $admxpprgui
   ```

2. Cargar el *bitsetram*, haciendo click en el botón examinar, como se muestra en la Figura C.5 y dar click en el botón programar. Se puede observar también el resultado en la salida de consola en la interfaz.

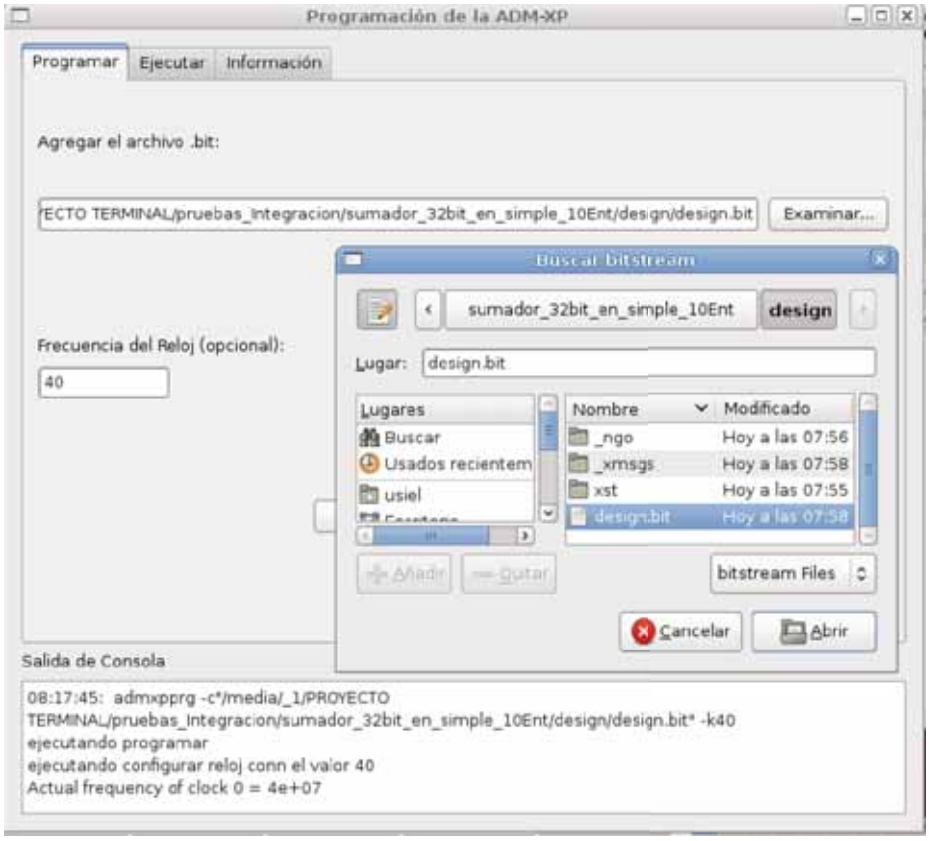

Figura C.5: selección del archivo de programación y ejecución del comando para programar el FPGA.

3. Pasar a la pestaña de ejecución, introducir los valores y ver el resultado, la Figura C.6 muestra cómo se ve la interfaz para el caso del sumador de 32 bits.

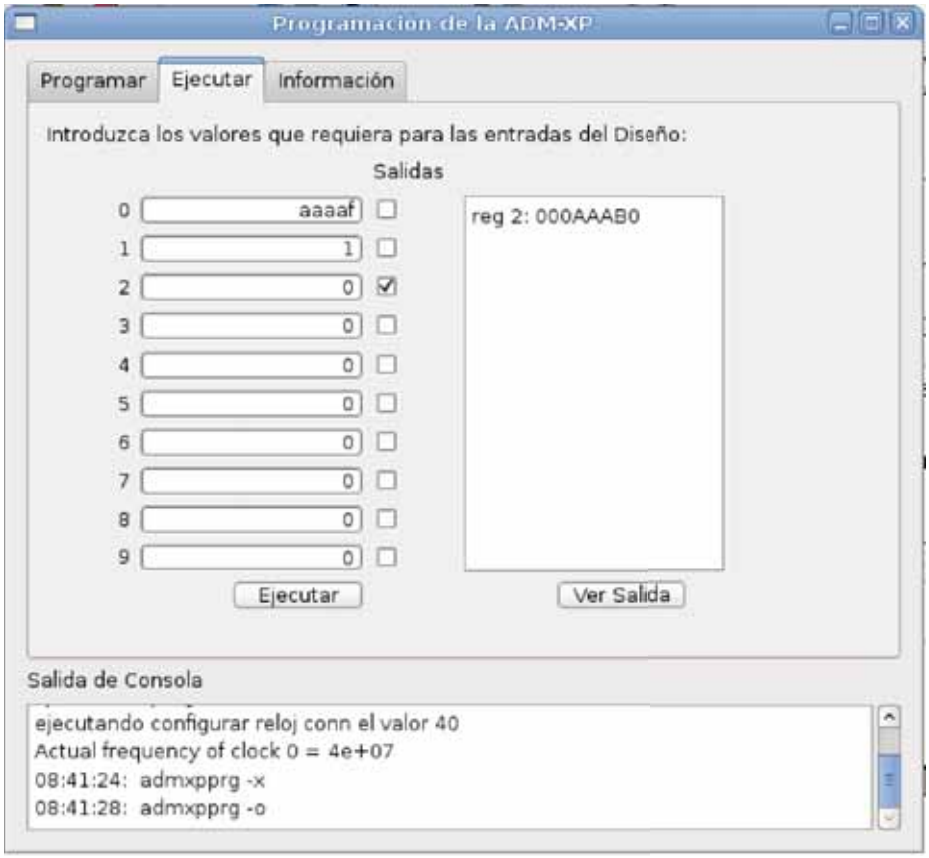

Figura C.6: Uso del FPGA mediante la interfaz gráfica.

Apéndice D

Código Fuente del diseño en VHDL

Archivo: `design.vhd`

```
1
2 ################################################################################
3 # Este Código es una modificación al diseño Base para
4 # trabajar con el comando admxpprg
5 #
6 # Implementa un sumador de 32 bits de dos entradas
7 #
8 # Septiembre 2010
9 #
10 ################################################################################
11 -- Diseño para la ADM-XP que implementa un sumador de 32 bits
12 -- las entradas se reciben en los registros 0 y 1
13 -- la salida se muestra en el registro 2
14
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18
19 library work;
20 use work.localbus.all;
21
22 entity design is
23     port(
24         lclk             : in    std_logic;
25         lreset_l         : in    std_logic;
26         lwrite           : in    std_logic;
27         lads_l           : in    std_logic;
28         lblast_l         : in    std_logic;
29         lbterm_l         : inout std_logic;
30         lad              : inout std_logic_vector(31 downto 0);
31         lready_l         : out   std_logic;
32         lbe_l            : in    std_logic_vector(3 downto 0);
33         fholda           : in    std_logic);
34 end design;
35
36 architecture mixed of design is
37
38    component sum32 is
39       Port ( e1 : in  STD_LOGIC_VECTOR (31 downto 0);
40              e2 : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
41          sal : out  STD_LOGIC_VECTOR (31 downto 0)
42    );
43   end component sum32;
44
45    signal rst : std_logic;
46    signal lblast_i : std_logic;
47    signal lads_i : std_logic;
48    signal lwrite_i : std_logic;
49    signal lready_o_l : std_logic;
50    signal lready_oe_l : std_logic;
51    signal lbterm_i : std_logic;
52    signal lbterm_o_l : std_logic;
53    signal lbterm_oe_l : std_logic;
54    signal la_i : std_logic_vector(23 downto 2);
55    signal ld_o : std_logic_vector(31 downto 0);
56    signal ld_i : std_logic_vector(31 downto 0);
57    signal ld_oe_l : std_logic;
58    signal lbe_i : std_logic_vector(3 downto 0);
59
60    signal qlads : std_logic;
61    signal ds_xfer : std_logic;
62    signal ds_decode : std_logic;
63    signal ds_write : std_logic;
64
65    signal la_q : std_logic_vector(23 downto 2);
66    signal logic0, logic1 : std_logic;
67
68--agregue los registros que necesite
69
70  signal e_e1: std_logic_vector(31 downto 0);
71  signal e_e2: std_logic_vector(31 downto 0);
72  signal s_sal: std_logic_vector(31 downto 0);
73
74    signal reg0 : std_logic_vector(31 downto 0);
75    signal reg1 : std_logic_vector(31 downto 0);
76    signal reg2 : std_logic_vector(31 downto 0);
77    signal reg3 : std_logic_vector(31 downto 0);
78    signal reg4 : std_logic_vector(31 downto 0);
79    signal reg5 : std_logic_vector(31 downto 0);
80    signal reg6 : std_logic_vector(31 downto 0);
81    signal reg7 : std_logic_vector(31 downto 0);
82    signal reg8 : std_logic_vector(31 downto 0);
83    signal reg9 : std_logic_vector(31 downto 0);
84
85
86 begin
87
88    logic0 <= '0';
89    logic1 <= '1';
90
91    --
92    -- Convertir las entradas a un nivel activo alto.
93    --
94
95    rst <= not lreset_l;
96    lblast_i <= not lblast_l;
97    lbterm_i <= not lbterm_l;
98    lads_i <= not lads_l;
99    lwrite_i <= lwrite;
100   la_i <= lad(la_i'range);
101   ld_i <= lad;
102   lbe_i <= not lbe_l;
103
104  --
105  -- Generar una versión adecuada de lads_l, el cual
106  -- se activa cuando el FPGA es direccionado Y el FPGA no funciona
```

```
107  -- como maestro de bus local
108  --
109
110    qlads <= lads_i and not la_i(23) and not fholda;
111
112    --
113    -- Mantener la dirección del bus local en el pulso de lads_l.
114    --
115
116    latch_addr : process(rst, lclk)
117    begin
118        if rst = '1' then
119            la_q <= (others => '0');
120        elsif lclk'event and lclk = '1' then
121            if lads_i = '1' then
122                la_q <= la_i;
123            end if;
124        end if;
125    end process;
126
127    --
128    -- 'lbterm_l' debe ser manejado sólo cuando el FPGA es direccionado; de otro modo
129    -- estará como alta impedancia, por que el control lógico de la tarjeta
130  -- podría tambien manejarlo.
131    --
132
133    lbterm_l <= lbterm_o_l when lbterm_oe_l = '0' else 'Z';
134
135    --
136    -- 'lready_l' debe ser manejado sólo cuando el FPGA es direccionado; de otro modo
137    -- estará como alta impedancia, por que el control lógico de la tarjeta
138  -- podría tambien manejarlo.
139    --
140
141    lready_l <= lready_o_l when lready_oe_l = '0' else 'Z';
142
143    --
144    -- Manejar el bus local de datos en una lectura.
145    --
146
147    lad <= ld_o when ld_oe_l = '0' else (others => 'Z');
148
149    --
150    -- Si el ciclo actual es una escritura, actualizar los registros
151    -- Habilite los bytes que desea utilizar
152  --
153
154    update_regs : process(lclk, rst)
155    begin
156        if rst = '1' then
157            reg0 <= (others => '0');
158            reg1 <= (others => '0');
159        elsif lclk'event and lclk = '1' then
160            if ds_xfer = '1' and ds_write = '1' then
161                if la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '0' and la_q(2) = '0' then
162                    if lbe_i(0) = '1' then reg0(7 downto 0) <= ld_i(7 downto 0); end if;
163                    if lbe_i(1) = '1' then reg0(15 downto 8) <= ld_i(15 downto 8); end if;
164                    if lbe_i(2) = '1' then reg0(23 downto 16) <= ld_i(23 downto 16); end if;
165                    if lbe_i(3) = '1' then reg0(31 downto 24) <= ld_i(31 downto 24); end if;
166
167                elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '0' and la_q(2) = '1' then
168                    if lbe_i(0) = '1' then reg1(7 downto 0) <= ld_i(7 downto 0); end if;
169                    if lbe_i(1) = '1' then reg1(15 downto 8) <= ld_i(15 downto 8); end if;
170                    if lbe_i(2) = '1' then reg1(23 downto 16) <= ld_i(23 downto 16); end if;
171                    if lbe_i(3) = '1' then reg1(31 downto 24) <= ld_i(31 downto 24); end if;
172
```

```
173                elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '1' and la_q(2) = '0' then
174               if lbe_i = "1111" then reg2 <= ld_i; end if;
175
176                elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '1' and la_q(2) = '1' then
177               if lbe_i = "1111" then reg3 <= ld_i; end if;
178
179                elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '0' and la_q(2) = '0' then
180               if lbe_i = "1111" then reg4 <= ld_i; end if;
181
182                elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '0' and la_q(2) = '1' then
183               if lbe_i = "1111" then reg5 <= ld_i; end if;
184
185                elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '1' and la_q(2) = '0' then
186               if lbe_i = "1111" then reg6 <= ld_i; end if;
187
188           elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '1' and la_q(2) = '1' then
189               if lbe_i = "1111" then reg7 <= ld_i; end if;
190
191           elsif la_q(5) = '1' and la_q(4) = '0' and la_q(3) = '0' and la_q(2) = '0' then
192               if lbe_i = "1111" then reg8 <= ld_i; end if;
193
194                elsif la_q(5) = '1' and la_q(4) = '0' and la_q(3) = '0' and la_q(2) = '1' then
195               if lbe_i = "1111" then reg9 <= ld_i; end if;
196                end if;
197            end if;
198        end if;
199    end process;
200
201
202   e_e1 <= reg0;
203   e_e2 <= reg1;
204
205
206     --
207     -- Generar la señal ld_o, lleva los datos de salida para el bus local
208     --
209
210     generate_ld_o : process(rst, lclk)
211     begin
212         if rst = '1' then
213             ld_o <= (others => '0');
214         elsif lclk'event and lclk = '1' then
215             if la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '0' and la_q(2)='0' then
216                 ld_o <= reg0;
217             elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '0' and la_q(2)='1' then
218                 ld_o <= reg1;
219         elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '1' and la_q(2)='0' then
220                 ld_o <= s_sal;
221         elsif la_q(5) = '0' and la_q(4) = '0' and la_q(3) = '1' and la_q(2)='1' then
222                 ld_o <= reg3;
223         elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '0' and la_q(2)='0' then
224                 ld_o <= reg4;
225         elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '0' and la_q(2)='1' then
226                 ld_o <= reg5;
227         elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '1' and la_q(2)='0' then
228                 ld_o <= reg6;
229         elsif la_q(5) = '0' and la_q(4) = '1' and la_q(3) = '1' and la_q(2)='1' then
230                 ld_o <= reg7;
231         elsif la_q(5) = '1' and la_q(4) = '0' and la_q(3) = '0' and la_q(2)='0' then
232                 ld_o <= reg8;
233         elsif la_q(5) = '1' and la_q(4) = '0' and la_q(3) = '0' and la_q(2)='1' then
234                 ld_o <= reg9;
235         else
236                 ld_o <= x"00000000";
237             end if;
238         end if;
```

```
239     end process;
240
241     --
242     -- Instanciar la maquina de estados de esclavo directo; monitorea el bus local
243     -- para ciclos de esclavo directo y responder apropiadamente
244     --
245
246
247
248     dssm : plxdssm
249         port map(
250             clk => lclk,
251             rst => rst,
252             sr => logic0,
253             qlads => qlads,
254             lblast => lblast_i,
255             lbterm => lbterm_i,
256             lwrite => lwrite_i,
257             eld_oe => open,
258             ld_oe_l => ld_oe_l,
259             lready_o_l  => lready_o_l,
260             lready_oe_l => lready_oe_l,
261             lbterm_o_l => lbterm_o_l,
262             lbterm_oe_l => lbterm_oe_l,
263             idle => open,
264             transfer => ds_xfer,
265             decode => ds_decode,
266             write => ds_write,
267             ready => logic1,
268             stop => logic1);
269
270   design1: sum32
271     Port map (
272         e1 => e_e1,
273         e2 => e_e2,
274         sal=> s_sal
275       );
276
277 end mixed;
```

Apéndice E

Código Fuente para la programación del FPGA

Archivo: `tarjeta.cpp`

```
1 # ##############################################################################
2 #Esta clase realiza todas las acciones sobre la tarjeta ADM-XP
3 #
4 #Está basada en los ejemplos simple, info, y clock del ADM-XRC SDK de Alpha Data
5 #
6 #Septiembre 2010
7 #
8 # ##############################################################################
9 #include "tarjeta.h"
10 #include "admxpprg.h"
11
12
13
14 void Tarjeta::programarTarjeta(string op_arg){
15   cout<<"ejecutando programar"<<endl;
16   int                 ret = 0;
17   ADMXRC2_HANDLE  card = ADMXRC2_HANDLE_INVALID_VALUE;
18   ADMXRC2_STATUS      status;
19   ADMXRC2_SPACE_INFO spInfo;
20   volatile uint32_t* fpgaSpace;
21
22   const char* filename=op_arg.c_str();
23
24   status = ADMXRC2_OpenCard(0, &card);
25   if (status != ADMXRC2_SUCCESS) {
26     cout<<"Falló al abrir la tarjeta: "<<
27                 ADMXRC2_GetStatusString(status)<<endl;
28     ret = -1;
29     goto done;
30   }
31
32   /* Obtener la Dirección del Espacio del FPGA*/
33   status = ADMXRC2_GetSpaceInfo(card, 0, &spInfo);
34   if (status != ADMXRC2_SUCCESS) {
35     cout<<"Fallo al Obtener la información del espacio: "<<
36                 ADMXRC2_GetStatusString(status)<<endl;
```

```
37      ret = -1;
38      goto done;
39    }
40    fpgaSpace = (volatile uint32_t*) spInfo.VirtualBase;
41
42    //Configurar el FPGA
43    status = ADMXRC2_ConfigureFromFile(card, filename);
44    if (status != ADMXRC2_SUCCESS) {
45      cout<<"Fallo al cargar el bitstream "<<filename<<": "<<
46                  ADMXRC2_GetStatusString(status)<<endl;
47      cout<<"status: "<<status<<endl;
48      ret = -1;
49      goto done;
50    }
51    done:
52    if (card != ADMXRC2_HANDLE_INVALID_VALUE) {
53      ADMXRC2_CloseCard(card);
54    }
55
56 }
57 void Tarjeta::infoTarjeta(){
58    int                    ret = 0;
59    ADMXRC2_HANDLE         card = ADMXRC2_HANDLE_INVALID_VALUE;
60    ADMXRC2_STATUS         status;
61    ADMXRC2_VERSION_INFO   versionInfo;
62    ADMXRC2_CARD_INFO      cardInfo;
63    ADMXRC2_SPACE_INFO*    spaces;
64    stringstream             fname;
65    stringstream             bname;
66    stringstream             pname;
67    uint32_t               i;
68
69
70    status = ADMXRC2_OpenCard(0, &card);
71    if (status != ADMXRC2_SUCCESS) {
72      cout<<"Falló al abrir la tarjeta: "<<
73                ADMXRC2_GetStatusString(status)<<endl;
74      ret = -1;
75      goto done;
76    }
77
78
79    status = ADMXRC2_GetCardInfo(card, &cardInfo);
80    if (status != ADMXRC2_SUCCESS) {
81      cout<<"Fallo al obtener la información de la tarjeta: "<<
82                ADMXRC2_GetStatusString(status)<<endl;
83      ret = -1;
84      goto done;
85    }
86
87    status = ADMXRC2_GetVersionInfo(card, &versionInfo);
88    if (status != ADMXRC2_SUCCESS) {
89      cout<<"Fallo al obtener información de la versión: "<<
90                ADMXRC2_GetStatusString(status)<<endl;
91      ret = -1;
92      goto done;
93    }
94
95    if (cardInfo.BoardType < 0 ||
96              cardInfo.BoardType >= ADMXRC2_BOARD_UNKNOWN) {
97      bname<<"desconocido (0x"
98              <<hex<<(unsigned long) cardInfo.BoardType<<")";
99      pname<<"package desconocido";
100   } else {
101     bname<<"ADM-XP";
102     pname<<"FF1704";
```

```
103   }
104
105   if (cardInfo.FPGAType < 0 || cardInfo.FPGAType >= ADMXRC2_FPGA_UNKNOWN) {
106     fname<<"desconocido (0x"<<hex<<(unsigned long) cardInfo.FPGAType<<")";
107   } else {
108     fname<<"Virtex-II Pro 2VP100";
109   }
110
111   cout<<"API ver              "<<(unsigned long) versionInfo.APIMajor<<"."<<
112         (unsigned long) versionInfo.APIMinor<<endl;
113   cout<<"Driver ver           "<<(unsigned long) versionInfo.DriverMajor<<"."<<
114         (unsigned long) versionInfo.DriverMinor<<endl<<endl;
115   cout<<"Board Type           "<<bname.str()<<endl;
116   cout.fill('0');
117   cout<<"Card ID              "<<(unsigned long)cardInfo.CardID << " (0x";
118   cout.width(4);
119   cout<<right<<hex<<uppercase<<(unsigned long) cardInfo.CardID<< ")" <<endl;
120
121   cout<<"Serial Number        "<<dec<<(unsigned long) cardInfo.SerialNum<<" (0x";
122   cout.width(8);
123   cout<<right<<hex<<(unsigned long) cardInfo.SerialNum<<")"<<endl;
124
125   cout<<"Board/Logic Rev  "<< ((int)cardInfo.BoardRevision >> 4) <<"."<<
126               (cardInfo.BoardRevision & 0xF)<<
127               "/"<<((int)cardInfo.LogicRevision >> 4)<<
128               "."<<(cardInfo.LogicRevision & 0xF)<<endl;
129
130   cout<<"FPGA                 "<<fname.str()<<" ["<<pname.str()<<"]"<<endl;
131   cout<<endl;
132
133   cout<<"Num. de genedadores de reloj "<<
134             (unsigned long) cardInfo.NumClock<<endl;
135   cout<<"Num. de canales DMA          "
136             <<(unsigned long) cardInfo.NumDMAChan<<endl;
137   cout<<"Num. de espacios             "
138             <<(unsigned long) cardInfo.NumSpace<<endl;
139   cout<<endl;
140
141   spaces = (ADMXRC2_SPACE_INFO*) malloc(cardInfo.NumSpace *
142                   sizeof(ADMXRC2_SPACE_INFO));
143   for (i = 0; i < cardInfo.NumSpace; i++) {
144     unsigned long lsize, vsize;
145
146     status = ADMXRC2_GetSpaceInfo(card, i, &spaces[i]);
147     if (status != ADMXRC2_SUCCESS) {
148       cout<<"ESpacio "<<i<<": unable to get information"<<endl;
149     } else {
150       cout<<"Espacio "<<i<<" ("<<(i==0?"FPGA":"control")<<"):"<<endl;
151       cout<<"Base física                  0x";
152       cout.fill('0');
153       cout.width(8);
154       cout<<right<<hex<<uppercase<<
155           (unsigned long) spaces[i].PhysicalBase<<endl;
156       lsize = spaces[i].LocalSize;
157       if (lsize) {
158         cout<<"Rango local                  0x";
159         cout.fill('0');
160         cout.width(8);
161         cout<<right<<hex<<uppercase<<
162             (unsigned long) spaces[i].LocalBase<<" - 0x";
163         cout.width(8);
164         cout<<right<<hex<<uppercase<<
165         (unsigned long) (spaces[i].LocalBase + spaces[i].LocalSize - 1)
166                                     <<endl;
167       }
168       vsize = spaces[i].VirtualSize;
```

```
169       if (vsize) {
170         cout<<"Rango virtual                0x";
171         cout.fill('0');
172         cout.width(8);
173         cout<<right<<hex<<uppercase<<
174             (unsigned long) spaces[i].VirtualBase<<" - 0x";
175         cout.width(8);
176         cout<<right<<hex<<uppercase<<
177     (unsigned long) ((unsigned long)spaces[i].VirtualBase + vsize - 1)<<endl;
178       }
179     }
180   }
181   free(spaces);
182   cout<<endl;
183
184   cout<<"Numero de bancos de RAM      "<<
185       (unsigned long) cardInfo.NumRAMBank<<endl;
186   cout<<"Bank presence bitmap         ";
187   cout.width(8);
188   cout<<right<<hex<<uppercase<<(unsigned long) cardInfo.RAMBanksFitted<<endl;
189   for (i = 0; i < cardInfo.NumRAMBank; i++) {
190     ADMXRC2_BANK_INFO bank;
191
192     cout<<"RAM Bank ";
193     cout.width(2);
194     cout<<right<<(unsigned long) i<<"                    ";
195
196     status = ADMXRC2_GetBankInfo(card, i, &bank);
197     if (status != ADMXRC2_SUCCESS) {
198       cout<<"no se puede determinar"<<endl;
199     } else {
200       if (!bank.Fitted) {
201         cout<<"No Reconocido"<<endl;
202       } else {
203         uint32_t size, byteWidth;
204         string     type;
205         int       ntype = 0;
206
207         type="";
208
209         if (bank.Type & ADMXRC2_RAM_SDRAM_DDR) {
210           if (ntype) {
211             type += "+";
212           }
213           type+= "DDR SDRAM";
214           ntype++;
215         }
216         if (bank.Type & ADMXRC2_RAM_SRAM_DDR2) {
217           if (ntype) {
218             type+="+";
219           }
220           type+= "DDR-II SRAM";
221           ntype++;
222         }
223
224         if ((bank.Width % 9) == 0) {
225           /* Take care of x9 memories (with parity bits) */
226           byteWidth = bank.Width / 9;
227         } else {
228           byteWidth = bank.Width / 8;
229         }
230         size = byteWidth * (bank.Size / 1024);
231         cout<<type<<dec<<" "<<(unsigned long) (bank.Size / 1024)<<
232             "kword x "<<(unsigned long) bank.Width<<"bits ("<<
233             (unsigned long) size<<"kB)"<<endl;
234       }
```

```
235     }
236   }
237   cout<<endl;
238
239   done:
240   if (card != ADMXRC2_HANDLE_INVALID_VALUE) {
241     ADMXRC2_CloseCard(card);
242   }
243 }
244
245 void Tarjeta::confReloj(string op_arg){
246   double num;
247   for(unsigned int i = 0; i < op_arg.length();i++){
248     if(! (isdigit((int)op_arg.at(i)) || op_arg.at(i)=='.') ){
249       cout<<"valor incorrecto para el reloj"<<endl;
250       exit(1);
251     }
252   }
253   num = strtod(op_arg.c_str(),NULL);
254
255   if(num>80||num<6){
256     cout<<"valor incorrecto para el reloj"<<endl;
257     exit(1);
258   }
259
260   cout<<"ejecutando configurar reloj conn el valor "<<num<<endl;
261
262   int                ret = 0;
263   ADMXRC2_STATUS     status;
264   ADMXRC2_HANDLE     card = ADMXRC2_HANDLE_INVALID_VALUE;
265   double             freq;
266   double             afreq;
267   unsigned long      index;
268
269   status = ADMXRC2_OpenCard(0, &card);
270   if (status != ADMXRC2_SUCCESS) {
271     cout<<"Failed to open card with ID 0: " <<
272             ADMXRC2_GetStatusString(status) <<endl;
273     ret = -1;
274     goto done;
275   }
276
277   index = 0;
278   freq = num * 1000000.0;
279
280
281   /*
282    ** Establecer la frecuencia del generador de reloj indicado
283    */
284   status = ADMXRC2_SetClockRate(card, index, freq, &afreq);
285   if (status != ADMXRC2_SUCCESS) {
286     cout<<"Failed to set clock "<<index<<
287           " to "<<freq / 1.0e6<<"MHz: "<<
288             ADMXRC2_GetStatusString(status)<<endl;
289     ret = -1;
290     goto done;
291   }
292
293   cout<<"Actual frequency of clock "<<index<<" = "<<afreq<<endl;
294
295   done:
296   if (card != ADMXRC2_HANDLE_INVALID_VALUE) {
297     ADMXRC2_CloseCard(card);
298   }
299 }
300
```

```
301 void Tarjeta::ejecutar(unsigned long op_arg[]){
302   if(op_arg!= NULL)
303     _ejecutar(op_arg);
304   else
305     cout<<"error en los valores para --execute"<<endl;
306 }
307
308 void Tarjeta::_ejecutar(unsigned long op_arg[]){
309   int  ret = 0;
310   ADMXRC2_HANDLE      card = ADMXRC2_HANDLE_INVALID_VALUE;
311   ADMXRC2_STATUS        status;
312   ADMXRC2_SPACE_INFO spInfo;
313   volatile uint32_t* fpgaSpace;
314
315   status = ADMXRC2_OpenCard(0, &card);
316   if (status != ADMXRC2_SUCCESS) {
317     cout<<"Falló al abrir la tarjeta: "<<
318             ADMXRC2_GetStatusString(status)<<endl;
319     ret = -1;
320     goto done;
321   }
322
323   /* Obtener la Dirección del Espacio del FPGA*/
324   status = ADMXRC2_GetSpaceInfo(card, 0, &spInfo);
325   if (status != ADMXRC2_SUCCESS) {
326     cout<<"Fallo al Obtener la información del espacio 0: "<<
327             ADMXRC2_GetStatusString(status)<<endl;
328     ret = -1;
329     goto done;
330   }
331   fpgaSpace = (volatile uint32_t*) spInfo.VirtualBase;
332
333   for(int i=0;i<10;i++)
334     fpgaSpace[i] = op_arg[i];
335
336   done:
337   if (card != ADMXRC2_HANDLE_INVALID_VALUE) {
338     ADMXRC2_CloseCard(card);
339   }
340
341
342 }
343
344 void Tarjeta::mostrarSalida(){
345   int ret = 0;
346   ADMXRC2_HANDLE        card = ADMXRC2_HANDLE_INVALID_VALUE;
347   ADMXRC2_STATUS        status;
348
349   volatile uint32_t* fpgaSpace;
350   ADMXRC2_SPACE_INFO spInfo;
351
352   status = ADMXRC2_OpenCard(0, &card);
353   if (status != ADMXRC2_SUCCESS) {
354     cout<<"Falló al abrir la tarjeta: "<<ADMXRC2_GetStatusString(status)<<endl;
355     ret = -1;
356     goto done;
357   }
358
359   /* Obtener la Dirección del Espacio del FPGA*/
360   status = ADMXRC2_GetSpaceInfo(card, 0, &spInfo);
361   if (status != ADMXRC2_SUCCESS) {
362     cout<<"Fallo al Obtener la información del espacio 0: "<<
363               ADMXRC2_GetStatusString(status)<<endl;
364     ret = -1;
365     goto done;
366   }
```

```
367   fpgaSpace = (volatile uint32_t*) spInfo.VirtualBase;
368
369
370   for(int i=0;i<10;i++){
371     cout.fill('0');
372     cout.width(8);
373     cout<<right<<hex<<uppercase<< (unsigned long) fpgaSpace[i]<<endl;
374   }
375
376   done:
377   if (card != ADMXRC2_HANDLE_INVALID_VALUE) {
378     ADMXRC2_CloseCard(card);
379   }
380 }
```

Apéndice F

Código Fuente para la Interfaz Gráfica

Archivo: `interfaz.cpp`

```
1
2# ##############################################################################
3#Esta clase contiene la lógica de la interfaz gráfica para el manejo
4#del comando admxpprg
5#
6#
7#Septiembre 2010
8#
9# ##############################################################################
10
11#include <QFileDialog>
12#include "interfaz.h"
13#include "ui_interfaz.h"
14#include <QProcess>
15#include <iostream>
16#include <QMessageBox>
17
18 using namespace std;
19
20 Interfaz::Interfaz(QWidget *parent) :
21     QWidget(parent),
22     ui(new Ui::Interfaz)
23 {
24     ui->setupUi(this);
25 }
26
27 Interfaz::~Interfaz()
28 {
29     delete ui;
30 }
31
32
33 void Interfaz::on_botonProgramar_clicked()
34 {
35          comando="admxpprg -c\"";
36
37   if(!examinarWarning())
```

```
38      return;
39
40          comando+=ui->PathBitLine->text()+"\"";
41
42    if(!ui->frecLine->text().isEmpty())
43      comando+=" -k"+ui->frecLine->text();
44
45          cout<<comando.toStdString().c_str()<<endl;
46
47    textSal=ui->textConsola;
48
49
50    ejecutarProceso();
51 }
52
53 void Interfaz::on_botonEjecutar_clicked()
54 {
55    QString dataIn,salto="\n";
56    QStringList data = getDatos();
57    comando = "admxpprg -x";
58
59    for(int i=0;i<data.size();i++)
60      dataIn+=data.at(i) + salto;
61
62    textSal = ui->textConsola;
63
64    ejecutarProceso();
65
66    proc->write(dataIn.toStdString().c_str());
67    proc->closeWriteChannel();
68
69 }
70
71 void Interfaz::on_botonVerSalida_clicked()
72 {
73    ui->textSalida->clear();
74    comando="admxpprg -o";
75    ui->textConsola->append(QTime::currentTime().toString()+
76                  ":  "+comando);
77
78    textSal = ui->textSalida;
79
80          if(textSal!=ui->textSalida && textSal!=ui->textInfo)
81          textSal->append(QTime::currentTime().toString()+
82                    ":  "+comando);
83          proc = new QProcess( this );
84      connect( proc, SIGNAL(readyReadStandardOutput ()),
85           this, SLOT(readFromStdoutSalidaDatos()) );
86          proc->start(comando.toStdString().c_str());
87 }
88
89 void Interfaz::on_tabWidget_currentChanged(int index)
90 {
91    if(index==2){
92        ui->textInfo->clear();
93        comando="admxpprg -i";
94
95        ui->textConsola->append(QTime::currentTime().toString()+
96                        ":  "+comando);
97
98        textSal = ui->textInfo;
99        ejecutarProceso();
100   }
101 }
102
103 void Interfaz::ejecutarProceso(){
```

```
104   if(textSal!=ui->textSalida && textSal!=ui->textInfo)
105     textSal->append(QTime::currentTime().toString()+":  "+comando);
106   proc = new QProcess( this );
107   connect( proc, SIGNAL(readyReadStandardOutput ()),
108         this, SLOT(readFromStdoutConsola()) );
109   proc->start(comando.toStdString().c_str());
110 }
111
112 void Interfaz::on_botonExamBit_clicked()
113 {
114   QString fileName;
115   fileName = QFileDialog::getOpenFileName(this,
116           tr("Buscar bitstream"),
117             QDir::homePath(),
118               "bitstream Files (*.bit)");
119   ui->PathBitLine->setText(fileName);
120 }
121
122 bool Interfaz::examinarWarning(){
123   if(ui->PathBitLine->text().isEmpty()){
124     QMessageBox::warning(this, tr("Abrir Bitstream"),
125       tr("Debe escribir la ruta de un Bitstream"),
126           QMessageBox::Cancel);
127         return false;
128   }
129   return true;
130 }
131
132 void Interfaz::readFromStdoutConsola()
133 {
134   QByteArray str,str2;
135   str = proc->readAllStandardOutput();
136
137   for(int j = 0,k=0;(j = str.indexOf('\n', j)) != -1;++j,k=j){
138     str2 = str.mid(k,j-k);
139     textSal->append(str2);
140   }
141 }
142
143 void Interfaz::readFromStdoutSalidaDatos()
144 {
145         QByteArray str,str2;
146         int *lista = listarRegSal();
147
148         str = proc->readAllStandardOutput();
149
150     for(int i=0,j = 0,k=0;(j = str.indexOf('\n', j)) != -1;++j,k=j,i++){
151             if(lista[i]){
152                 QString auxstr = "reg ";
153                 auxstr+=QString::number(i,10);
154                 auxstr+=": ";
155                 //textSal->append(auxstr);
156                 str2 = str.mid(k,j-k);
157                 textSal->append(auxstr+str2);
158             }
159         }
160 }
161
162 int* Interfaz::listarRegSal(){
163     int *lista = new int[10];
164
165     for(int i=0;i<10;i++)
166         lista[i]=0;
167
168
169
```

```
170     if(ui->check_0->isChecked())        lista[0]=1;
171     if(ui->check_1->isChecked())        lista[1]=1;
172     if(ui->check_2->isChecked())        lista[2]=1;
173     if(ui->check_3->isChecked())        lista[3]=1;
174     if(ui->check_4->isChecked())        lista[4]=1;
175     if(ui->check_5->isChecked())        lista[5]=1;
176     if(ui->check_6->isChecked())        lista[6]=1;
177     if(ui->check_7->isChecked())        lista[7]=1;
178     if(ui->check_8->isChecked())        lista[8]=1;
179     if(ui->check_9->isChecked())        lista[9]=1;
180
181
182
183     return lista;
184 }
185
186 QStringList Interfaz::getDatos(){
187   QStringList cad;
188   cad.append(ui->datoLine_0->text());
189   cad.append(ui->datoLine_1->text());
190   cad.append(ui->datoLine_2->text());
191   cad.append(ui->datoLine_3->text());
192   cad.append(ui->datoLine_4->text());
193   cad.append(ui->datoLine_5->text());
194   cad.append(ui->datoLine_6->text());
195   cad.append(ui->datoLine_7->text());
196   cad.append(ui->datoLine_8->text());
197   cad.append(ui->datoLine_9->text());
198   return cad;
199 }
```

Bibliografía

[1] Xilinx Inc. Documentación de de xilinx ise 8 design suite software. http://www.xilinx.com/support/sw_manuals/xilinx8/download/, January 2010.

[2] Kime C. R. Morris Mano M. *Fundamentos de Diseño Lógico y Computadoras*. Prentice Hall, primera edición edition, 1998.

[3] Oscar Alvaado Nava. *Implementación en FPGAs de Algoritmos de Compresión-Descompresión para Dispositivos Móviles*. cinvestav, 2007.

[4] Pablo Huerta Pellitero. Sistemas mpsoc en fpgas. http://www.escet.urjc.es/~phuerta/pdf/442.pdf, January 2010.

[5] Miguel Morales Sandoval. Introducción a los fpgas y el cómputo reconfigurable. http://ccc.inaoep.mx/~mmorales/documents/FPGAsyReconfig.pdf, January 2010.

[6] Xilinx Inc. Manual del fpga virtex ii pro. www.xilinx.com/support/documentation/user_guides/ug012.pdf, September 2010.

[7] Xilinx Inc. Xupv2p hardware reference. http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf, January 2010.

[8] Alpha Data. Adm-xp data sheet. http://www.alpha-data.com/archive/adm-xp.pdf, September 2010.

[9] Alpha-Data. Manual de la tarjeta adm-xp. http://www2.informatik.hu-berlin.de/~fwinkler/psvfpga/alpha-data/ADM-XP-Manual_0.2.pdf, January 2010.

[10] Alpha-Data. Manual del sdk de alpha data. ftp://ftp.alphadata.co.uk/pub/admxrc/linux/old/, January 2010.

[11] Nokia Corporation. Qt reference documentation. http://doc.trolltech.com/4.5/index.html, July 2010.

[12] Desconocido. Foro de alumnos de informática de la universidad alicante. http://penyainformatica.mforos.com/229121/5811146-programita-ec/, July 2010.