IIE-PCI

Una plataforma de desarrollo para el bus PCI

Proyecto de fin de carrera

Sebastián Fernández
Ciro Mondueri

Docente

Juan Pablo Oliver

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería
Montevideo, Uruguay

Diciembre 2003

1. Tabla de Contenido

1.1. Indice

1. Tabla de Contenido.....
1.1. Indice.....5
2. Introducción.....
2.1. Resumen.....9
2.2. Abstract.....13
2.3. Créditos y Agradecimientos.....15
2.4. Organización de la documentación.....16
3. Motivación.....
3.1. Motivación del proyecto.....19
3.2. Motivación para el desarrollo de la placa IIE-PCI.....21
4. Objetivos.....
4.1. Objetivos generales del proyecto.....25
4.2. Descripción detallada de los objetivos.....27
5. Antecedentes y estado del arte.....
5.1. Antecedentes.....31
5.2. Estado del arte al comenzar el proyecto.....32
6. Placa de desarrollo IIE-PCI.....
6.1. Organización del capítulo.....39
6.2. Características de la placa.....40
6.3. Descripción general.....41
6.4. Descripción funcional.....43
6.5. Diseño.....49
6.6. Fabricación.....62
6.7. Montaje.....66
6.8. Costo de la placa.....70
6.9. Pruebas.....71
6.10. Conclusiones.....72
7. Core PCI PCITWBM.....
7.1. Introducción.....79
7.2. Características del core PCI.....80
7.3. Descripción general.....81
7.4. Uso del core PCITWBM.....84
7.5. Interfaz PCI.....88
7.6. Interfaz Wishbone.....104
7.7. Arquitectura y funcionamiento.....108

7.8. Herramientas..... 124
7.9. Conclusiones.....126
8. Software.....
8.1. Organización del capítulo.....129
8.2. Características del software controlador (driver).....130
8.3. Driver RW_BAR..... 131
8.4. Herramientas de prueba.....145
8.5. Recursos PCI del sistema operativo Linux..... 148
8.6. Conclusiones.....152
9. Aplicaciones de prueba.....
9.1. WB_RAM.....155
9.2. WB_SDRAM.....159
9.3. WB_DAC..... 163
10. Referencias, Bibliografia y Glosario.....
10.1. Referencias..... 169
10.2. Bibliografía.....171
10.3. Glosario..... 172
11. Apéndices.....
11.1. Documentos..... 177

2. Introducción

2.1. Resumen

Este proyecto consistió en el desarrollo de una placa PCI basada en lógica programable para ser utilizada como plataforma de desarrollo.
Se desarrollaron además herramientas que facilitan el diseño de aplicaciones, entre las más importantes se encuentran un core PCI sintetizable y un driver para Linux que permite comunicarse con la placa.

El bus PCI es el estándar actual para la conexión de tarjetas en un PC. Permite conectar placas que le agregan funcionalidades, por ejemplo placas de sonido, capturadoras de video, placas de red, co-procesadores matemáticos, etc. Estas tarjetas suelen llamarse placas PCI.

El diseño de prototipos de placas PCI, ya sea para su posterior uso comercial o para investigación, se ve acelerado si se comienza el desarrollo a partir de un placa de propósito general, ya existente, que pueda ser rápidamente adaptada según las necesidades del diseño.

Haciendo énfasis en la adaptabilidad de la placa a la aplicación a desarrollar, la electrónica que la conforme debe ser fácilmente modificable. Hoy día, este objetivo se logra utilizando circuitos integrados de lógica reconfigurable, como ser los FPGA.
Un FPGA puede verse como una caja llena de flip-flops, compuertas lógicas y memorias RAM, que pueden ser interconectados de una cierta manera, a gusto del usuario, al configurarlo.

Partiendo de una placa PCI cuyo corazón sea un FPGA, el diseño de un prototipo consistirá básicamente en realizar, en algún lenguaje de descripción de circuitos, un diseño que configure el FPGA para que se comporte según los requerimientos establecidos. Como un FPGA puede reconfigurarse, en caso de detectar un error en el funcionamiento, basta corregir el error y volver a configurar el FPGA.

El diseño a programar en el FPGA debe implementar la funcionalidad que se desea agregar al PC y debe comunicarse utilizando el bus PCI.
Dada la complejidad del bus PCI, este diseño podría separarse en dos bloques, uno que resuelva la comunicación con el bus y otro que implemente la funcionalidad.
El bloque que resuelve la comunicación es llamado core PCI y se encarga de encapsular los detalles del estándar (manejo de errores, decodificación de direcciones,

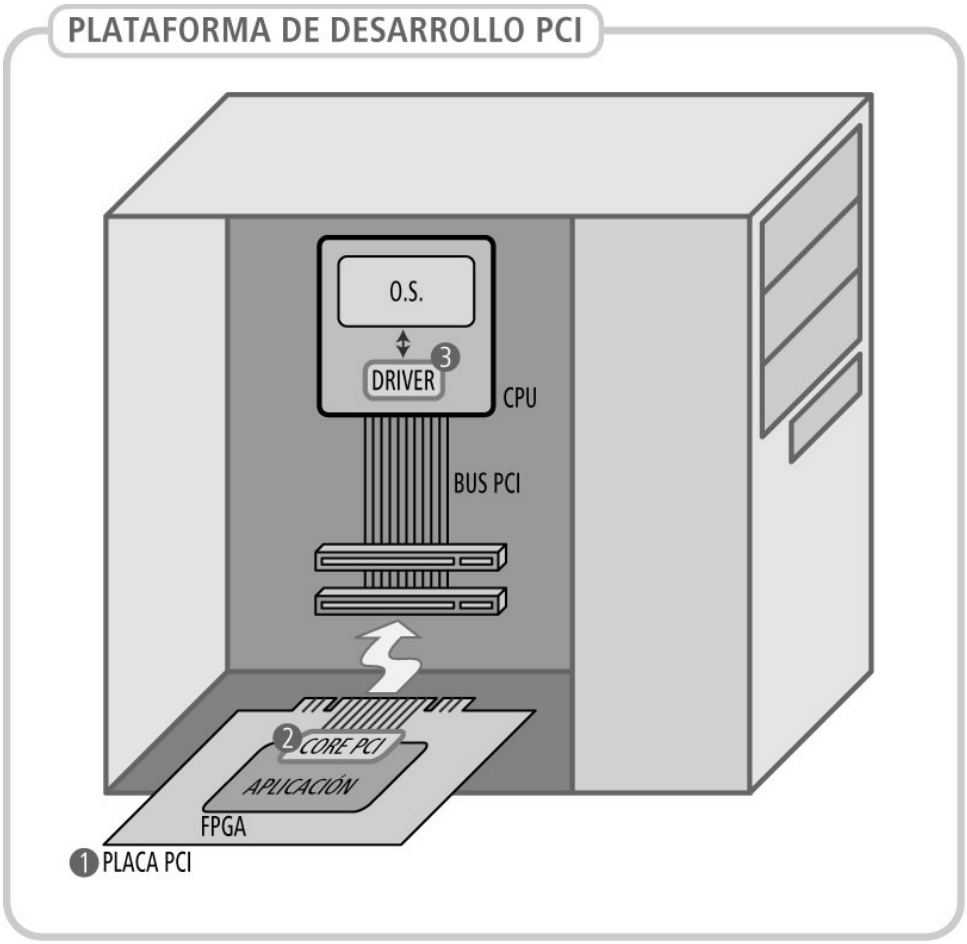
señalizaciones, etc.).

Inicialmente el proyecto planteaba el desarrollo de un core PCI de libre distribución y un software controlador genérico (driver) que permite al sistema operativo acceder a los recursos provistos por la placa PCI.

Las primeras versiones del core PCI fueron probadas en la placa PCI con lógica reconfigurable con la que cuenta el Instituto de Ingeniería Eléctrica (IIE). Estas primeras pruebas no cumplieron con los requerimientos de performance necesarios debido a que los FPGAs de la placa son lentos.
Por esta razón se incluyó como parte de los objetivos del proyecto, el diseño y fabricación de una placa PCI de bajo costo con lógica reconfigurable.

El resultado final del proyecto es una plataforma de desarrollo para diseños hardware que utilizan el bus PCI.

La plataforma esta compuesta por:
1. una placa de propósito general con un conector para bus PCI y lógica reconfigurable
2. un diseño modular (core PCI) que encapsula los detalles de funcionamiento del bus PCI, para ser utilizado por la aplicación que se configura en el FPGA.
3. software controlador (driver) que permite que desde programas ejecutándose en un PC se pueda acceder a la placa.



Características de la placa:
- FPGA ACEX1K100 de ALTERA
- conectores para agregarle placas de expansión
- 128 Mbit (4Mx32) de memoria on board
- PLL para regenerar y multiplicar señal de reloj proveniente de un cristal o reloj PCI.

Características del core PCI:
- target PCI
- interfaz Wishbone
- descripto en lenguaje VHDL

Características del driver:
- licencia de libre distribución
- compatible con Kernel Linux 2.4.x
- mapea la placa PCI como dispositivos de caracteres

- diseño modular, permite ser cargado en tiempo de ejecución no es necesario compilarlo con el kernel.

2.2. Abstract

This project involves the development of a PCI device card based in programmable logic, to be used as a development platform. In addition, a toolkit for the development of applications, including a PCI core and a Linux device driver to communicate with the PCI device card were also part of this project.

The PCI bus the current standard for the interconnection of PCI device cards installed in a PC. It defines an interface for connecting devices that add a certain function, for example, a sound, video capture, network access, mathematical co-processors, etc.

Prototyping a PCI device card for a commercial purpose or for research is greatly simplified by using an already designed and tested general purpose development card that can be easily adapted according to the requirements of the project.

As the PCI card must be easily adapted to the desired application, the electronic components that conform it must be easily modified. Nowadays, this is achieved by using circuits based in reprogrammable logic, as the FPGAs.
A FPGA by itself can be regarded as a box full of flip-flops, logic gates and RAM, that can be interconnected in a certain way, according to the preferences of the user, by a simple programming method.

The design that is programmed into the FPGA must implement the new function being added to the PC, and must also be capable of communicating with the PC through the PCI bus.

Given the complexity of the PCI bus, the design can be split in two modules, one in charge of communicating through the PCI bus, and the other responsible for providing the new function.
The module that takes care of the communication is usually called PCI core, and simplifies the use of the PCI bus by hiding the specific details (error checking, address decoding, signaling, etc.).

Initially the project consisted of developing a PCI core and a generic device driver for it, both under an open source kind of license,

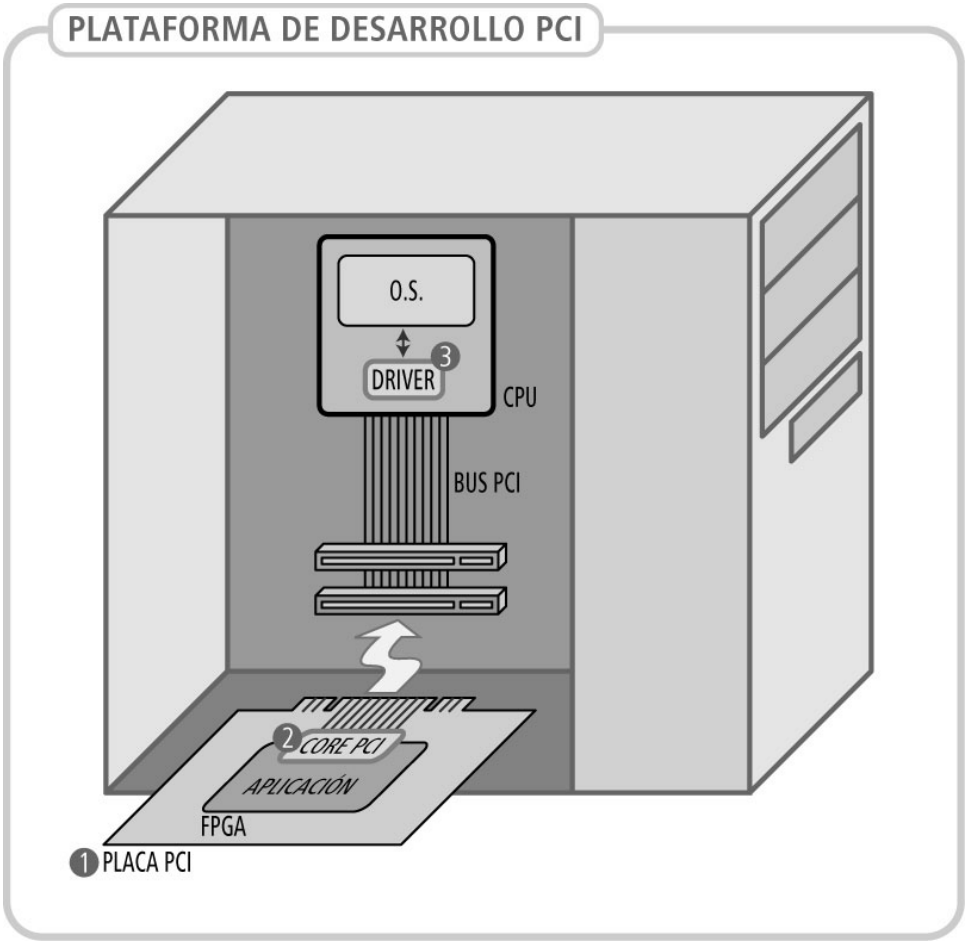
The first versions of the PCI core were tested with an existing configurable board

belonging to the IIE (Instituto de Ingeniería Eléctrica). The first tests showed the constraints in performance imposed by the FPGA chips used in those boards. Because of this, the development and manufacturing of a low cost PCI card based on FPGA devices, became part of the objectives of the project.

The final result is a hardware development platform for devices using the PCI bus.

The platform contains:

1. a general purpose board based on reprogrammable logic, that can be connected to the PCI bus.
2. a core PCI modular design that hides the inner-workings of the PCI bus from the user, to be used by the application configured in the FPGA.
3. a software driver to easily access the functions programmed in the board.



2.3. Créditos y Agradecimientos

2.3.1. Créditos

Terminar este proyecto habría sido más difícil sin el aporte de algunas empresas y personas.

- La estructura del espacio de configuración del core PCI esta basada en el diseño que nos envió Pablo Aguayo.
- Los FPGAs de ALTERA fueron donados por la compañía y Guillermo Jaquenod, el representante para Sudamérica, nos facilitó la tarea de solicitar la donación.
- Las memorias SDRAM fueron donadas por Micron, sus fabricantes.
- El FPGA y la memoria de la primer placa fueron soldados por el personal de la empresa CCC.
- El módulo controlador de memoria SDRAM fue diseñado por Jimena Saporiti y Agustin Villavedra.
- El cambio de la memoria y FPGA de la primer placa y el armado de la segunda placa fue posible gracias al soldador de punta fina prestado por Etienne Delacroix y el microscopio bifocal prestado por la Clínica Oftalmológica Mondueri - Ruiz.

2.3.2. Agradecimientos

Nuestras familias, Mariana Robano, los Tatos, Etienne Delacroix, Julio Pérez, Juan Pablo Oliver, Fiorella Haim, Javier Rodríguez, Virginia Marchesano, Pablo Rolando, Pedro Arzuaga, Fernando Silveira, Conrado Rossi, Guillermo Jaquenod, Leonardo Steinfeld, Santiago Castillo, Alvaro "Cheche" Rovira, Allison Martínez, Sebastián Filippini, Martín Guimaranes, Alvaro Tuzman, Jimmy Baikoviciuis, Carlos Pechiar, Mariana Borges, TWiki y TWiki2pdf.

GRACIAS!

2.4. Organización de la documentación

La documentación está organizada en los siguientes capítulos y apéndices:

- **Introducción**
  - Resumen del proyecto, agradecimientos y organización de la documentación.
- **Motivación**
  - Justificación del proyecto
- **Objetivos**
  - Objetivos generales y detallados de cada una de las partes del proyecto (placa, core PCI y driver)
- **Antecedentes y estado del arte**
  - Otros Cores PCI, placas de desarrollo PCI y drivers disponibles.
- **Placa de desarrollo IIE-PCI**
  - Documentación, detalles de la implementación y conclusiones
- **Core PCI PCITWBM**
  - Documentación, detalles de la implementación y conclusiones
- **Software**
  - Documentación, detalles de la implementación y conclusiones
- **Aplicaciones de prueba**
  - Aplicaciones de prueba de la plataforma
- **Apéndices**
  - **Manuales de usuario**
    - Guia de uso de placa, driver, core.
  - **Pinout del chip FPGA**
  - **Esquemáticos de la placa**
  - **Lista de materiales de la placa**
    - Costos y detalle de los componentes utilizados en la placa
  - **Especificación Wishbone**
    - Resumen de la especificación Wishbone
- **Referencias, Bibliografia y Glosario**

3. Motivación

3.1. Motivación del proyecto

La utilización de un PC para desarrollar e implementar aplicaciones hardware es una alternativa interesante, debido a, su gran difusión y bajo costo, el tener resuelta la comunicación con el mundo exterior (con personas y máquinas), la existencia de infinidad de herramientas de desarrollo y prueba.
El PC, siendo un sistema de propósito general, puede ser ajustado a las necesidades del usuario agregando placas con la electrónica necesaria, acompañadas de software que permita controlarlas. Un posible ejemplo es el procesamiento de imágenes de video: basta colocar una tarjeta adquisidora de video y configurar su software controlador para hacer que una aplicación de procesamiento de imágenes, ejecutándose en el PC, pueda procesar las imágenes adquiridas con la placa.

Hace algunos años el bus ISA era el estándar para desarrollar placas que se conectaran a un PC, lo cual era relativamente sencillo.
A mediados de los 90, las exigencias crecientes en cuanto a velocidad de transferencias de datos superaron la capacidad del bus ISA. Esto motivó que surgieran varias tecnologías de transición que derivaron en la aparición de un nuevo estándar: el bus PCI [10.1.I].
El bus PCI presenta varias ventajas con respecto al bus ISA, como ser su independencia de la arquitectura del procesador, su tasa de transferencia más elevada y la implementación de control de errores en las transferencias.
Es interesante entonces poder hacer uso del bus PCI de un PC para desarrollar aplicaciones de todo tipo.

Si a esto agregamos que una misma placa pueda reconfigurarse y servir para resolver diferentes problemas, la alternativa se vuelve aún más atractiva.
Cuando mencionamos la posibilidad de reconfiguración de una placa nos estamos refiriendo a utilizar circuitos integrados cuya lógica sea configurable por el usuario (FPGAs, PLDs, etc.)
Pero, como su nombre lo recuerda, la lógica configurable debe ser configurada.
Un FPGA sin configurar es como una caja llena de flip-flops, compuertas lógicas y memorias RAM. Es necesario interconectarlos de una cierta manera, siguiendo un cierto diseño, para que realicen una función útil.
Por lo tanto, para que una placa con un FPGA permita agregar ciertas funcionalidades a un PC, se debe especificar, en algún lenguaje de descripción de circuitos, un diseño

que implemente estas funcionalidades y, a la vez, haga uso del bus PCI para comunicarse.

Dada la complejidad del bus PCI, este diseño podría separarse en dos bloques, uno que resuelva la comunicación con el bus PCI y otro que implemente la funcionalidad.
El bloque que resuelve la comunicación es llamado core PCI y se encarga de encapsular los detalles del estándar (manejo de errores, decodificación de direcciones, señalizaciones, etc.).
Existen cores PCI de varios fabricantes pero, al ser todos comerciales, su utilización requiere de la compra de una licencia. El IIE cuenta con la licencia de un core PCI donada por la compañía ALTERA, cuyo costo es de U$S 8,995.
Al momento de comenzar el proyecto no se encontró ningún core PCI de libre distribución. Fue esta una de las motivaciones principales para realizar el proyecto.

En resumen:
El bus PCI está disponible en todas las PCs hoy en día.
Por otro lado, las placas de lógica reconfigurable son muy útiles para el desarrollo de prototipos, ya sea para enseñanza o investigación, o para la producción en pequeñas cantidades o de corta vida.
Resulta entonces más que razonable la combinación PC y placa con lógica reconfigurable, y su nexo es el bus PCI.
Para poder desarrollar una aplicación que, en forma sencilla, haga uso de los recursos del bus, debe de utilizarse algún bloque que encapsule los detalles.
Este proyecto plantea desarrollar este bloque y difundirlo bajo una licencia de libre distribución.

3.2. Motivación para el desarrollo de la placa IIE-PCI

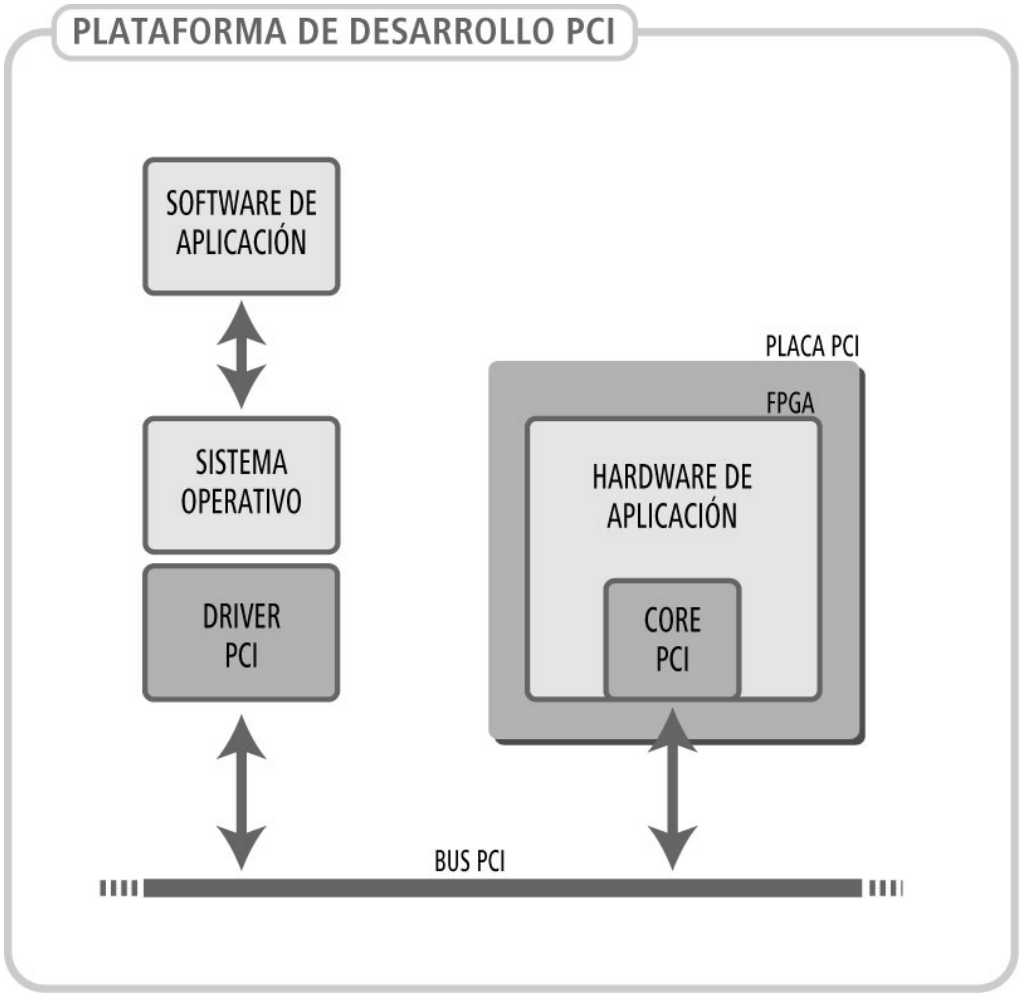
El proyecto original consistía en diseñar, en un lenguaje de descripción de hardware, un core PCI y verificar su correcto funcionamiento en las placas ARC-PCI existentes en el IIE. Acompañando dicho core, se desarrollaría un software controlador (driver) para el sistema operativo Linux. Los códigos fuente del core y driver serían publicados bajo alguna licencia de libre distribución, del tipo GPL (GNU / GPL) y su equivalente para cores IP (OHGPL / www.opencores.org).

Se comenzó por el diseño del core PCI para configurar el FPGA de la placa ARC-PCI y al completar un diseño que no tenia aún todas las funcionalidades implementadas, se comprobó que al sintetizarlo para los FPGAs de la placa ARC-PCI no cumplía con los requerimientos de performance necesarios.

El problema es que el FPGA de la ARC-PCI (EPF1K50RC240-3) es un chip lento para el tipo de diseño a implementar. Incluso el core PCI desarrollado por ALTERA con una aplicación simple debe ser sintetizado poniendo especial cuidado en las opciones de síntesis elegidas, dado que de otra forma no se llega a la performance requerida.
Ante este problema se estudiaron dos posibles soluciones, comprar una nueva placa de desarrollo o diseñar una.
Al momento de tomar la decisión no se encontraron placas de bajo costo, por ello se optó por realizar y fabricar un diseño propio.

Otras motivaciones fueron: recabar experiencia práctica en la fabricación de placas PCI, diseño de placas multicapa para frecuencias del orden de decenas de MHz, y uso de componentes de soldadura superficial.

Un esquema de la plataforma completa se muestra a continuación:


PLATAFORMA DE DESARROLLO PCI
SOFTWARE DE APLICACIÓN
SISTEMA OPERATIVO
DRIVER PCI
PLACA PCI
FPGA
HARDWARE DE APLICACIÓN
CORE PCI
BUS PCI

4. Objetivos

4.1. Objetivos generales del proyecto

El objetivo general del proyecto es contar con una plataforma propia y fácil de usar para desarrollar aplicaciones sobre bus PCI.

Dentro de dicho contexto, podemos encontrar tres grandes objetivos:
- diseño de un core PCI sintetizable
- diseño de una placa PCI de propósito general
- desarrollo de un software controlador de dispositivos genérico (driver).

Este proyecto plantea el diseño de un core PCI que se sintetizará en un FPGA.
Un core es un diseño especificado utilizando un lenguaje de descripción de hardware (AHDL, VHDL, Verilog), la especificación implica definir la interfaz que tendrá dicho diseño y cual será su comportamiento. Como decíamos, se está creando su especificación funcional en un lenguaje de descripción de hardware dado, no se está implementando físicamente el circuito. Posteriormente esta descripción puede ser sintetizada (llevada a un circuito eléctrico) en la tecnología deseada.
En nuestro caso al sintetizar el core PCI se crea un archivo para configurar el FPGA y hacer que éste se comporte como un dispositivo PCI.

Se plantea el diseño y fabricación de una placa PCI de propósito general, teniendo como objetivo principal su bajo costo.
Esta placa debe caracterizarse por ser muy flexible, permitir incorporar circuitos adicionales, proveer varios voltajes de alimentación, contar con memoria on-board y leds y llaves de propósito general.
Estas características la harán muy versátil y útil en etapas de prototipado de diseños o investigación académica. El instituto cuenta con varias placas de propósito general, pero para desarrollos basados en el bus PCI cuenta solo con 2 placas (modelo ARC-PCI de ALTERA).
En esta misma placa se probará el core PCI diseñado como parte del proyecto.

Un software controlador de dispositivos (driver) esconde la complejidad y los detalles de cómo funciona su correspondiente dispositivo. Permite acceder a los recursos brindados por el dispositivo utilizando interfaces bien definidas por el sistema operativo. De esta forma una diversidad de dispositivos pueden ser accedidos por el usuario utilizando los mismos mecanismos.

Se plantea como objetivo diseñar un driver genérico para dispositivos PCI. Estará desarrollado con la misma filosofía que la placa de propósito general, es decir, su forma de uso debe ser clara y debe poder ser fácilmente adaptable a varios tipos de dispositivos PCI.

4.2. Descripción detallada de los objetivos

4.2.1. Objetivos del core PCI

Se propone diseñar el core PCI utilizando el lenguaje de descripción hardware VHDL. Dicho lenguaje es un estándar de la IEEE (versión actual: VHDL-93 IEEE Std.1076-1993). El utilizar un lenguaje estandarizado (frente a la posibilidad de utilizar uno propiedad de alguna empresa fabricante de FPGA, cómo ser el AHDL de Altera) es parte del concepto de facilitar el uso y difusión del core. Hay una gran variedad de herramientas para sintetizar código fuente en lenguaje VHDL. Existen múltiples sitios en Internet que facilitan la difusión de cores IP, y VHDL es el lenguaje común en la mayoría de ellos.

El bus PCI se basa en transacciones punto a punto entre dispositivos. El dispositivo que comienza la transacción se llama Master y el que acepta la transacción se llama Target. Se propone diseñar un core PCI únicamente con la funcionalidad de Target, ya que desarrollar ambos modos de funcionamiento sería demasiado ambicioso. La única funcionalidad que se pierde es la de poder empezar transacciones desde la placa, ya sea hacia otro dispositivo PCI o hacia la memoria del PC (similar a DMA).

El core PCI debe comunicarse utilizando el bus PCI y proveer al resto de las aplicaciones sintetizadas en el FPGA una interfaz de comunicación sencilla. Para esto se utilizará la especificación Wishbone, que define un bus pensado para comunicar cores IP que se encuentran en un mismo integrado. Debido a esto, el core PCI se comportará entonces, como un puente entre el bus PCI externo y el bus Wishbone interno. Detalles de la especificación pueden encontrarse en los apéndices y en www.opencores.org

4.2.2. Objetivos de la placa

El objetivo principal es el de construir una placa (IIE-PCI) de bajo costo (menos de U$S250) que permita probar diseños que hagan uso del bus PCI.

El bus PCI más difundido es de ancho de palabra de 32 bits, por lo que se dotará a la placa de un conector compatible con dicho bus. El diseñarla para que sea compatible con buses de 64 bits encarecería mucho el diseño (FPGA con más patas, más superficie de impreso y layout más complicado).

Para superar la performance de la placa ARC-PCI se utilizarán FPGAs más rápidos y de

mayor capacidad.
Para que la placa sea lo suficientemente versátil deberá contar con memoria on-board y conectores de expansión que le permitan conectarle placas desarrolladas con algún fin específico.

Se busca también adquirir experiencia en el diseño y fabricación de placas para diseños digitales que utilizan frecuencias de reloj mayores o iguales a 33MHz.
Dado que algunos de los componentes están disponibles sólo en encapsulados para montaje superficial, se determinarán cuales son las mayores dificultades que surgen de usar dichos encapsulados.

4.2.3. Objetivos del controlador de dispositivo

En particular el software controlador permitirá que el sistema operativo y las aplicaciones accedan a los diseños implementados en la placa IIE-PCI.

Dado que se está planteando diseñar una plataforma de desarrollo de bajo costo con un core PCI de libre distribución y uso, se opta por desarrollar el driver para un sistema operativo de libre distribución y sin costos de licencias. Se selecciona entonces Linux (distribución Red Hat Linux) debido a su gran disponibilidad y a la experiencia por parte de los integrantes del grupo en su utilización.

Ya que la placa fue concebida como hardware de propósito general, el driver también debe ser desarrollado con los mismos principios. El código debe ser sencillo, comprensible y fácilmente modificable por los eventuales usuarios con poco conocimiento del funcionamiento interno de las funcionalidades provistas por la placa.

5. Antecedentes y estado del arte

5.1. Antecedentes

En el IIE, en más de una ocasión, se han utilizado PCs con placas adicionales para implementar un diseño hardware.
Uno de los primeros proyectos con estas características fue un prototipo de un registrador de perturbaciones de la red de transmisión eléctrica. El mismo constaba de una placa que adquiría muestras de la red eléctrica y eran procesadas en un PC para detectar varios tipos de posibles desviaciones respecto del comportamiento nominal. En caso de detección positiva de desviaciones, las señales se almacenaban en medio magnético para su posterior procesamiento.[10.1.K]

El bus ISA fue utilizado en proyectos realizados en el IIE, por ejemplo para diseñar una tarjeta adquisidora de Video [10.1.F] o implementar una red neuronal en una placa con FPGAs para utilizarla como co-procesador [10.1.H].
Para desarrollos con bus ISA, el instituto cuenta con una placa ISA con lógica reconfigurable (RIPP10 de Altera) [10.1.J] que ha sido utilizado en proyectos de aceleración de algoritmos por Hardware.
El último proyecto que utilizó el bus ISA, como interfaz de comunicación con el PC, fue el de "Implementación de algoritmos de tratamiento de imágenes en lógica reconfigurable" [10.1.C]. Consistió en implementar en la placa RIPP10 algoritmos de procesamiento de imágenes y se compararon los tiempos de procesamiento con los de funciones implementadas en software. Los resultados fueron muy satisfactorios, pero se detectó que una de las limitantes para lograr mayor performance era la transferencia de datos a través del bus ISA, ya que las tasas de transferencia que se logran son muy bajas para la cantidad de datos que requiere el procesamiento de imágenes.

Posteriormente el IIE recibió la donación de dos placas PCI con lógica reconfigurable (ARC-PCI de Altera). Con estas placas se realizó el proyecto de fin de carrera "Neuro FPGA" [10.1.G] en el cual se implementaban redes Neuronales en Hardware y esta siendo utilizada en un proyecto de maestría sobre aceleración de algoritmos por hardware.

5.2. Estado del arte al comenzar el proyecto

5.2.1. Plataformas de desarrollo para bus PCI

Es posible comprar plataformas completas de desarrollo para bus PCI, es decir paquetes que incluyan una placa con lógica reconfigurables, un core PCI y un driver de ejemplo. Al momento de comenzar el proyecto, los altos costos de estos paquetes los tornan una alternativa inviable para nuestro medio y el de muchas universidades o empresas de la región.
El alto costo de estas plataformas se debe a que utilizan FPGAs topes de línea, cores PCI comerciales y software de desarrollo de drivers comerciales.
No se encontraron plataformas de desarrollo con prestaciones intermedias a un costo accesible.

A continuación se listan las características y precios de algunas plataformas disponibles en el mercado (Diciembre 2003):

- APEX PCI Development Kit (ALTERA)
  - costo: $3,495 USD
  - ALTERA FPGA APEX EP20K1000CF672
  - PCI 32-bit o 64-bit
  - licencia de core PCI de ALTERA por 60 días
  - conector para DIMM de SDRAM de 32-Mbyte
  - conector para placas de expansión
  - versión de evaluación por 30 días del software WinDriver de Jungo, que permite desarrollar drivers para windows
  - http://www.altera.com/products/devkits/altera/kit-apex_dev_kit.html
- PCI Development Kit, Stratix Edition (ALTERA)
  - costo: $1,995 USD
  - ALTERA FPGA Stratix EP1S25F1020C5
  - PCI 32-bit o 64-bit
  - licencia de core PCI de ALTERA por 60 días
  - 256-MByte PC333 DDR SDRAM (SODIMM)
  - http://www.altera.com/products/devkits/altera/kit-pci_stx.html
- PCISYS100 (PLD applications)
  - costo: $1,790 USD
  - ALTERA FPGA EP1K100FC4841
  - PCI 32bit o 64bit
  - licencia de core PCI desarrollado por PLD Applications para ser utilizado únicamente

en la placa.
  - indican que incluyen drivers para dispositivos PCI, pero no indican para que Sistema Operativo.
  - http://www.hitechdisti.com/plda/pcisys.htm
- DS-KIT-PCI32S-200
  - costo: $1,995 USD
  - XILINX FPGA XC2S200-6FG456C
  - 32-bit PCI
  - memoria SDRAM 8Mbyte (2Mx32)
  - licencia para core PCI de XILINX
  - http://legacy.memec.com/devkits/americas.shtml

Una lista más extensa de placas PCI que utilizan FPGAs puede encontrase en http://www.fpga-faq.com/FPGA_Boards.shtml

Al momento de escribir esta documentación han aparecido plataformas que intentan llenar el vacío existente.
- PCI ProtoBoard
  - costo: $395 USD
  - fecha de revisión 1.2: 8/9/03
  - FPGA EP1K30QC208-2
  - PCI 32 bit
  - no incluye memoria on-board
  - interfaz con bus PCI custom implementada en otro integrado.
  - driver para windows y linux
  - http://www.techniprise.com/pci_protoboard.htm

5.2.2. Placas PCI

Una alternativa a la compra de una plataforma de desarrollo completa es la compra de una placa PCI con lógica reconfigurable y desarrollar los drivers y el core PCI.

El IIE cuenta con 2 placas PCI con lógica reconfigurable, permiten agregarle DIMMs de memoria SRAM y cuenta con 3 FPGAs, lo que la hace una plataforma sumamente flexible y potente. El problema es que está fuera de producción, y si bien los FPGAs tiene una capacidad adecuada para muchas aplicaciones, la performance que se logra no es la adecuada para diseños que utilizan el bus PCI.

Al comenzar el proyecto no existían placas de pequeño o mediano porte, ya que todas estaban equipadas con FPGAs de gran tamaño y electrónica adicional que permiten

desarrollar una muy variada gama de aplicaciones.
- GR-PCI-XC2V LEON
  - costo $3,450 EUR
  - XILINX FPGA Virtex-II XC2V3000-FG676-4
  - 32-bit PCI interface
  - 8 Mbyte flash prom (2M x 32)
  - 1 Mbyte static ram (256K x 32)
  - memoria SDRAM 64 Mbyte (16M x 32)
  - Ethernet PHY 10/100 Mbit transeiver
  - http://www.gaisler.com/gr-pci-xc2v.html
- ADS-XLX-V2-DEV1500
  - costo $1,000.00 USD
  - FILINX FPGA XCV1500
  - 32/64-bit PCI interface
  - memoria 128 MB DDR SDRAM DIMM
  - http://www.silica.com/eval_kits/adx-20020928.html

Luego de haber terminado la placa del proyecto, salió al mercado una placa de desarrollo fabricada por Insight Electronics a U$S250. Esta misma placa es la utilizada en la plataforma de desarrollo DS-KIT-PCI32S-200 anteriormente mencionada.
Las características de la placa (DS-KIT-2S200) son las siguientes:
- costo $250 USD
- XILINX FPGA XC2S200-6FG456C
- 32-bit PCI
- memoria SDRAM 64Mbit (2Mx32)

5.2.3. Cores PCI

5.2.3.1. Cores PCI comerciales

Existen varios cores comerciales para bus PCI de 32 bits, todos con distintos niveles de complejidad. Algunos sólo se comportan como Target PCI y otros implementan las funcionalidades de Target y Master.
ALTERA y XILINX, los mayores fabricantes de FPGAs, ofrecen cores diseñados y probados por sus grupos de desarrollo y también ofrecen cores diseñados por terceros, aprobados por ellos.

Algunos cores disponibles en el mercado:
- Diseñados por ALTERA (www.altera.com)

- Interfaz PCI, 32-bit solo Target. Costo: $4,995
- Interfaz PCI, 32-bit Master/Target. Costo: $8,995

- Certificados por ALTERA y diseñados por Eureka Technology, Inc (http://www.eurekatech.com/)
  - Interfaz PCI, 32-Bit.solo Target y Master/Target.
  - Host bridge PCI, 32-Bit. Funciona como puente entre 2 buses PCI.
- Certificados por ALTERA y diseñados por PLD Aplications (http://www.plda.com/)
  - Interfaz PCI 32-Bit & 64-Bit. Solo Target y Master/Target.
- Diseñados por XILINX (www.xilinix.com)
  - Interfaz LogiCORE PCI, 32 bit Target. Costo $4,995
  - Interfaz LogiCORE PCI, 32 bit Master/Target. Costo $8,995

Un listado más extenso puede encontrarse en el sitio web de Design and Ruse (http://www.us.design-reuse.com/sip?id=68)

5.2.3.2. Cores PCI gratuitos

Al comenzar el proyecto, se conocía la existencia de un diseño VHDL de un interfaz esclavo PCI para una aplicación de Wavelet [10.1.A]. Se realizaron contactos con el diseñador y nos envió la documentación y el código VHDL de su diseño. Luego de estudiar la interfaz concluimos que no se adecuaba a nuestros objetivos ya que el diseño era muy dependiente de la aplicación e implementaba un conjunto de funcionalidades muy reducido.

Otra fuente de cores IP es el sitio www.opencore.org, en el se publican cores terminados o diseños que están siendo implementados. Durante el transcurso del proyecto, en opencores.org se publicó la necesidad de reunir diseñadores para llevar acabo un core PCI Master/Target de libre distribución.
Actualmente hay un core PCI de libre distribución en opencores.org. La aplicación de prueba esta implementada en una placa con un FPGA de XILINX. Se están implementando diseños (Testbench) que testen exhaustivamente el core PCI.

5.2.4. Drivers PCI

La forma correcta de utilizar las funciones provistas por una placa conectada al bus PCI dependen totalmente de la aplicación. Es decir, desde el punto de vista del software de aplicación, la forma en que generalmente se accede a una tarjeta de sonido no es útil para acceder a una tarjeta de red.
Entonces, el driver debe ser desarrollado a medida, tomando en cuenta la aplicación final.
Debido a esto, solamente tiene sentido hablar de los drivers existentes como ejemplos

o ayudas para el desarrollo de uno a medida.

Las plataformas de desarrollo generalmente proveen un driver de ejemplo o un kit de desarrollo de drivers, que facilitan la tarea.

En el momento de iniciar el proyecto se encontraba disponible un driver para la placa ARC-PCI desarrollado por William Bishop [10.1.N] de la Universidad de Waterloo en Canadá. Este driver funciona bajo el sistema operativo Windows NT.

La versión 2.4 del sistema operativo Linux se encontraba en desarrollo al momento de iniciar el proyecto, y simplifica sensiblemente el desarrollo de drivers frente a versiones previas. La documentación existente, sobre el bus PCI y sobre el desarrollo de módulos de kernel, es muy completa y detallada, y se complementa con la disponibilidad del código fuente de cientos de drivers que han sido desarrollados bajo la licencia de libre distribución GPL utilizada por Linux.

6. Placa de desarrollo IIE-PCI

6.1. Organización del capítulo

El capítulo se encuentra organizado en las siguientes secciones:

Características de la placa

Resumen de las características técnicas de la placa.

Descripción general

Descripción de los bloque principales que componen la placa y su interconexión.

Descripción funcional

Breve descripción del funcionamiento de la placa, alimentación, programación, etc. Un mayor detalle puede encontrarse en el manual de usuario, en los apéndices.

Diseño

- **Selección de componentes**

Se exponen las decisiones de diseño para la elección del FPGA, la memoria, los conversores DC-DC y el PLL.

- **Decisiones de layout**

Citerios utilizados para ubicar los componentes.

- **Diseño con señales digitales de alta frecuencia**

Información sobre adaptación de impedancias, crosstalk, planos de tierra, etc.

Fabricación

Circuitos impresos de prueba, software de diseño, fabricante y características de la placa fabricada.

Montaje

Recomendaciones y equipo necesario para soldar componentes de empaque superficial.

Pruebas

Primeras pruebas de funcionamiento realizadas.

Costos

Detalle de los costos de fabricación.

Conclusiones

Conclusiones, mejoras y recomendaciones.

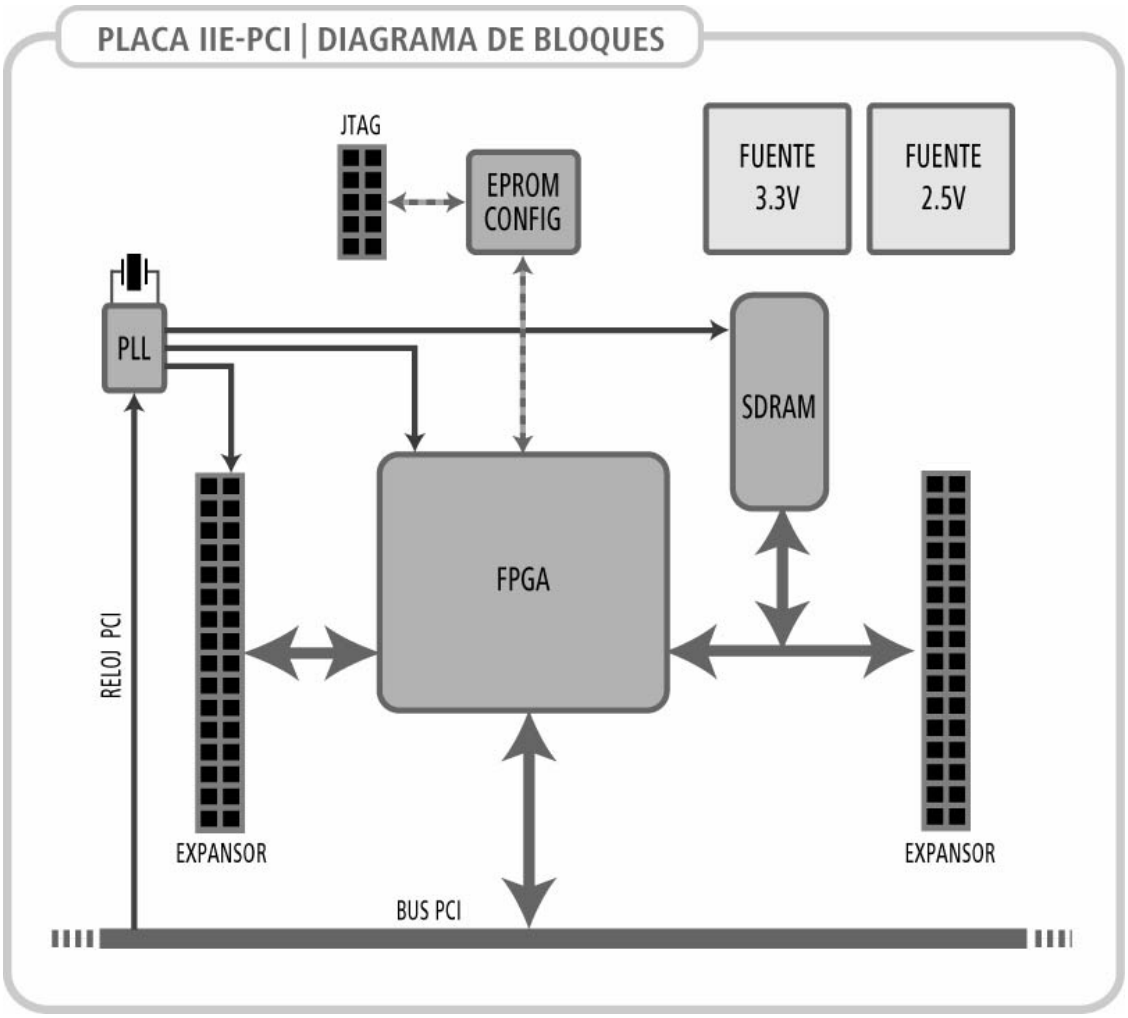
6.2. Características de la placa

Principales características de la placa:

- Compatible con bus PCI de 32 bit, 3.3V y 5V
- 128Mbit de memoria SDRAM on-board
- FPGA de la familia ACEX EP1K100PQ208
- Expansores con señales provenientes del FPGA
  - 31 señales de propósito general
  - 32 señales compartidas con el bus de datos de la SDRAM
- Conversores DC-DC para generar las fuentes de 3.3v y 2.5v utilizadas por los integrados.
- PLL que permite regenerar y multiplicar la frecuencia del reloj PCI o de un cristal en la placa.
- El FPGA puede ser programado utilizando:
  - EPROM de programación EPC2, se autoconfigura al encenderse
  - protocolo JTAG o configuración serie pasiva (PS) mediante el cable de programación ByteBlasterMV
- Cuenta con llaves y leds de propósito general
- Permite conectar una fuente de alimentación externa, para utilizar la placa sin estar conectada al bus PCI.

6.3. Descripción general

La placa IIE-PCI es una placa de propósito general de bajo costo. Su diseño está pensado para permitir desarrollar y probar diseños basados en la interfaz PCI.



El corazón de la placa es un FPGA de ALTERA, en el cual se implementa toda la lógica de la placa. Sus patas están conectadas al bus PCI, a la memoria SDRAM y a conectores de expansión.

El FPGA se desconfigura al quitarle la alimentación, por lo que la placa cuenta con una EPROM de configuración que permite que el FPGA se autoconfigure al encenderse.

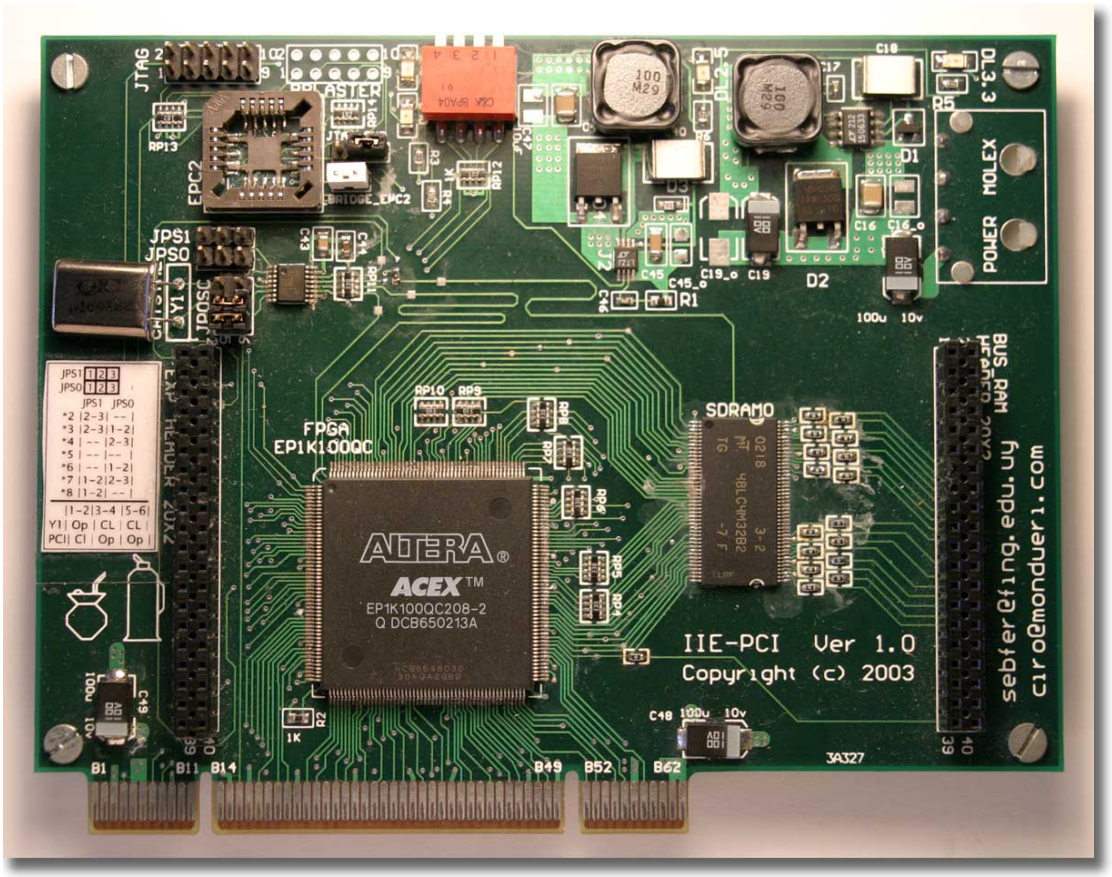
Tanto la programación de la EPROM como la configuración del FPGA se realizan

mediante el protocolo JTAG.

Cuenta con conectores del tipo header hembra que permiten conectar placas de expansión a la IIE-PCI.

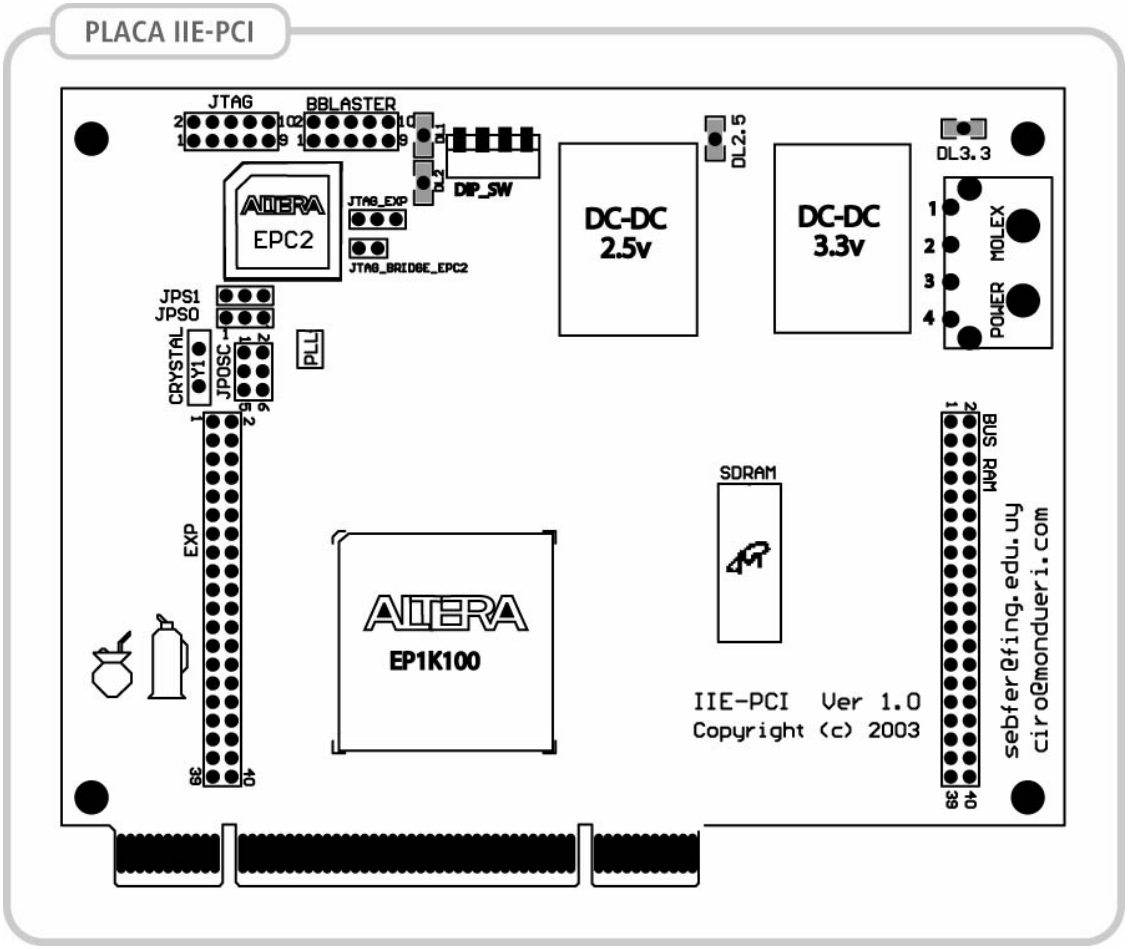
Un PLL en la placa provee señal de reloj al FPGA, a la memoria y a uno de los conectores de expansión. Estas señales de reloj pueden ser múltiplos de la frecuencia del reloj PCI o de la frecuencia de un cristal; esto se selecciona mediante jumpers en la placa.

La alimentación de los componentes proviene de dos conversores DC-DC, uno a 3.3V y el otro a 2.5V. La distribución de la alimentación (fuente y tierra) se hace utilizando las 2 capas interiores de las 4 que componen la placa.



6.4. Descripción funcional

A continuación se analizará cada componente por separado.



6.4.1. Alimentación

El plano interno de alimentación está partido en tres zonas, 3.3v, 2.5v y 5v. La zona de 5V está conectada a los contactos de 5V del bus PCI y al pin 4 del conector Molex. Este es el plano por el cual se alimenta la placa.

El plano de tierra está conectado a los contactos de GND del bus PCI y a los pines centrales del conector Molex.

La placa está diseñada para ser alimentada desde el bus PCI o utilizando una fuente de PC. Cuando la placa está conectada al bus PCI **no** es necesario proveer de alimentación

adicional a través del conector Molex.

Los expansores tienen disponibles fuentes y tierras en sus pines:
- **3.3v**: pin 4 de BUS_EXP
- **5v** : pin 5 de BUS_EXP
- **GND** : pines 2 y 3 de BUS_EXP y pines 1, 10, 11, 20, 21, 30, 31, 40 de BUS_RAM

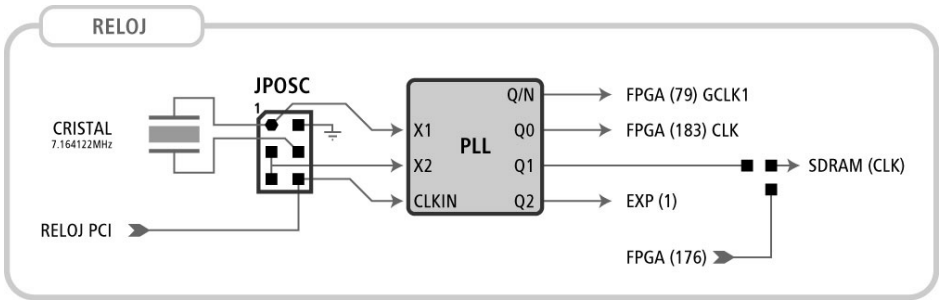
6.4.2. Señales de Reloj

Los relojes que llegan al FPGA, la SDRAM y el bus de expansión, son generados por el PLL.

EL PLL genera una señal de reloj por el pin **Q/N** y múltiplos de ésta por los pins **Q[2..0]**. La frecuencia de **Q/N** está dada por un cristal o el reloj PCI.

Las señales se conectan de la siguiente forma:
- **Q/N**: pin 79 del FPGA (GCLK1)
- **Q0** : pin 183 del FPGA (CLK)
- **Q1** : pin 68 de la SDRAM
- **Q2** : pin 1 del conector EXP

Observación: la señal de reloj que llega a la SDRAM, puede provenir de Q1 o del pin 176 del FPGA, que se comparte con la señal IO_EXP30 (pin 6 del conector EXP). Esto se selecciona soldando una resistencia de 0 ohm en una de dos posibles posiciones. Esto fue hecho para poder generar la señal de reloj desde dentro del FPGA.



El origen de las señales de reloj se elige con los jumpers **JPOSC**.

El factor de multiplicación para la frecuencia de salida se escoge mediante los Jumpers JPS1 y JPS0. Este puede ajustarse entre x2 y x8.

6.4.3. Configuración

El FPGA puede ser configurado utilizando los protocolos JTAG o modo serie pasivo (PS).

La configuración JTAG se hace utilizando el ByteBlasterMV conectado en el HEADER etiquetado *JTAG*.

La configuración en modo serie es utilizada por la memoria EPC2 o por el programador ByteBlasterMV a través del conector *BBLASTER*. En caso de seleccionar este modo, la EPC2 no debe de estar en su zócalo.

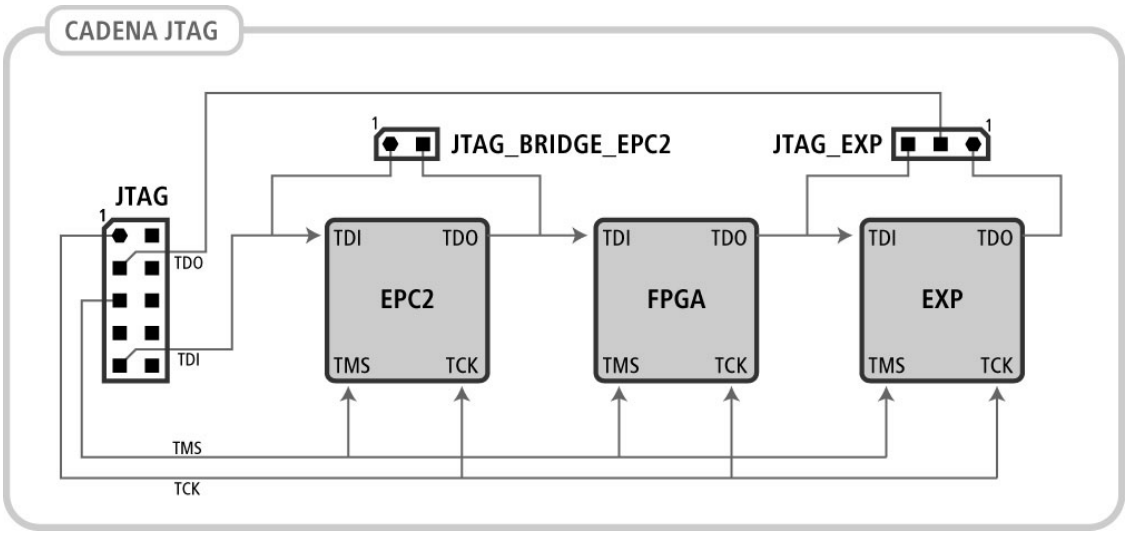
La memoria EPC2 es configurada utilizando el protocolo JTAG y se programa en la placa misma.

6.4.4. JTAG

El protocolo JTAG permite conocer el estado interno de un chip que soporte este protocolo y en el caso de los FPGAs también permite su configuración.

La placa permite encadenar la EPC2, el FPGA y cualquier dispositivo que soporte el protocolo, en el expansor. El orden de los dispositivos en la cadena JTAG es el siguiente:

1. EPC2
2. FPGA
3. EXPANSOR



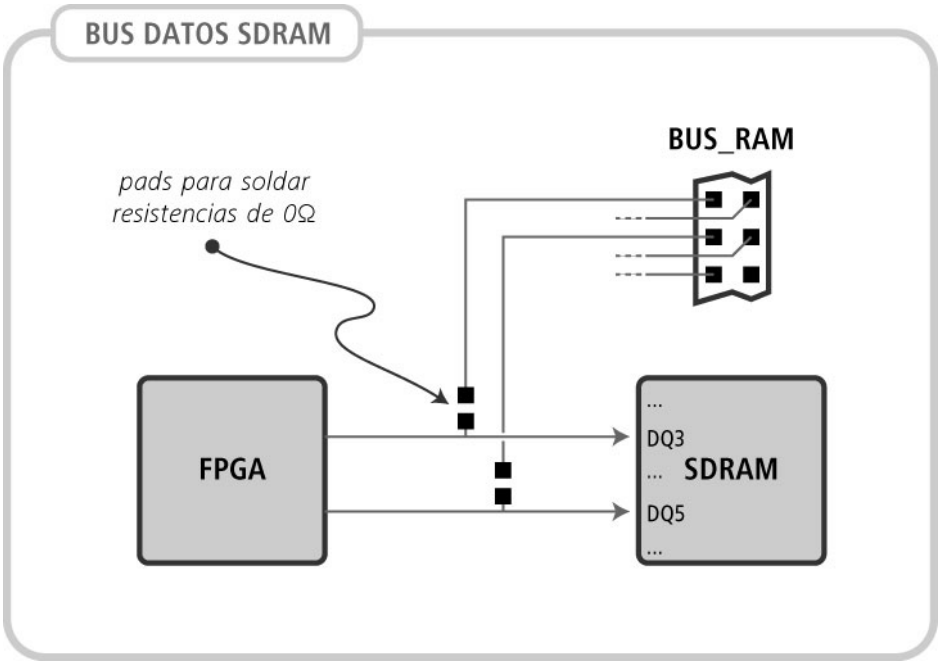
La cadena entre los pines TDI y TDO del HEADER JTAG siempre debe de estar cerrada. En caso de que la EPC2 no esté presente o que no haya un dispositivo que soporte el protocolo JTAG en el conector de expansión, se deben manipular los jumpers **JTAG_EXP** y **JTAG_BRIDGE_EPC2** para que la cadena JTAG quede cerrada de todas formas.

6.4.5. Expansores

A ambos lados del FPGA hay conectores Header hembra de 1mm, que permiten conectar a la IIE-PCI placas hijas, cuando se necesite electrónica adicional para implementar un diseño.

Al conector EXP llegan los pines libres del FPGA, reloj, fuentes de 3.3V y 5V y GND.

En el expansor del bus RAM pueden verse las señales del bus de datos de la SDRAM si se sueldan las resistencias de 0 ohm correspondientes.
Esto último permite hacer aplicaciones que lean datos de la SDRAM o utilizar estos pines sin usar la RAM. Para ello se debe de desactivar la SDRAM mediante la pata CS que está conectada al pin 175 del FPGA.



Señales en bus EXP (tabla dispuesta con la forma del expansor):

PIN_EXP	SEÑAL(PIN FPGA) -	SEÑAL(PIN FPGA)	PIN_EXP
1	EXPCLK	GND	2
3	GND	3.3v	4
5	5V	IO_EXP30(176)	6
7	IO_EXP29(197)	IO_EXP28(198)	8
9	IO_EXP27(199)	IO_EXP26(200)	10
11	IO_EXP25(202)	IO_EXP24(203)	12
13	IO_EXP23(204)	IO_EXP22(205)	14
15	IO_EXP21(206)	IO_EXP20(207)	16
17	IO_EXP19(208)	IO_EXP18(7)	18
19	IO_EXP17(8)	IO_EXP16(9)	20
21	IO_EXP15(10)	IO_EXP14(11)	22
23	IO_EXP13(12)	IO_EXP12(13)	24
25	IO_EXP11(14)	IO_EXP10(15)	26
27	IO_EXP9(16)	IO_EXP8(17)	28
29	IO_EXP7(18)	IO_EXP6(24)	30
31	IO_EXP5(25)	IO_EXP4(26)	32
33	IO_EXP3(27)	IO_EXP2(28)	34
35	IO_EXP1(29)	IO_EXP0(30)	36
37	JTAG_EXP_TDI	JTAG_EXP_TDO	38
39	FPGA_TMS	JTAG_TCK	40

Señales del bus RAM (tabla dispuesta con la forma del expansor):

PIN_RAM	SEÑAL -	SEÑAL	PIN_RAM
1	GND	DQ 0	2
3	DQ 1	DQ 2	4
5	DQ 3	DQ 4	6
7	DQ 5	DQ 6	8
9	DQ 7	GND	10
11	GND	DQ 15	12
13	DQ 14	DQ 13	14
15	DQ 12	DQ 11	16
17	DQ 10	DQ 9	18
19	DQ 8	GND	20
21	GND	DQ 31	22
23	DQ 30	DQ 29	24

PIN_RAM	SEÑAL	-	SEÑAL	PIN_RAM
25	DQ 28		DQ 27	26
27	DQ 26		DQ 25	28
29	DQ 24		GND	30
31	GND		DQ 16	32
33	DQ 17		DQ 18	34
35	DQ 19		DQ 20	36
37	DQ 21		DQ 22	38
39	DQ 23		GND	40

DQ *NN* indica que la señal de datos NN de la SDRAM está conectada al pin del expansor.

6.4.6. Llaves y leds

La placa cuenta con 2 leds y 4 llaves de propósito general.

Las 4 llaves de dos posiciones están conectadas, utilizando pull-ups, a 4 patas de entrada del FPGA.

CLOCK_LOCK: LED1 está conectado a pin CLOCK_LOCK del FPGA. En el software sintetizador se puede escoger que se encienda este led si el PLL interno del FPGA se "engancha" con la señal de reloj. En caso de no utilizarse, el pin puede ser manejado por una aplicación.

INIT_DONE: Si el FPGA se configura correctamente, se enciende LED2. También puede ser manejado por una aplicación luego de la configuración del FPGA.

6.5. Diseño

6.5.1. Selección de componentes

Con la idea de cumplir los objetivos planteados, se tomaron decisiones sobre qué componentes se utilizarían y cuál seria la arquitectura y la topología de la placa.

6.5.1.1. FPGA

De todas las familias de FPGAs de ALTERA, la que ofrecía la mayor cantidad de compuertas a menor precio era la ACEX. Esta familia de FPGAs tiene la misma estructura interna que los FLEX10K (FPGAs de la placa ARC-PCI) pero son menos costosos y más rápidos.
Se escogió utilizar el chip más grande de dicha familia, el EP1K100.

Los empaques disponibles para dicho integrado son TQFP y FBGA. Como el montaje sería realizado por los integrantes del proyecto o por alguna empresa del medio, se optó por el empaque TQFP (en el empaque TQFP, las patas están dispuestas en el perímetro del chip, mientras que en el FBGA, en vez de patas, el chip posee contactos en el cara inferior y no están accesibles directamente).
Ya que se desea tener la mayor cantidad de patas disponibles, se seleccionó el EP1K100QC208.

El modelo de integrado finalmente escogido fue el EP1K100QC208-1 de ALTERA.
El dígito final del número de parte (-1) está relacionado con los retardos de propagación del chip. Cuanto menor es éste número, menores son los retardos de propagación y por lo tanto se pueden implementar diseños que funcionen a mayor velocidad, es decir con frecuencias de reloj mayores.
Afortunadamente dicho FPGA pudo ser conseguido a través de una donación de la empresa ALTERA, ya que es el integrado más caro de la placa (U$S63.00 en enero del 2002). Se consiguieron 2 integrados EP1K100QC208-1 y 2 EP1K100QC208-2.

Se podría haber optado por un FPGA de XILINX, pero el grupo de trabajo tiene una mayor experiencia con las herramientas e integrados de la compañía ALTERA.

La siguiente tabla comparativa muestra las diferencias entre los FPGAs con los que cuenta el IIE:

placa	FPGA	Typ. gates	Max. gates	LEs	EAB	FPGA RAM	Ext. RAM
IIE-PCI	EP1K100	100,000	257,000	4.992	12	48 kbit	16 MByte
ARC-PCI	EPF10K50	50,000	116,000	2.880	10	20 kbit	12 MByte
UP1	EPF10K20	20,000	63,000	1.152	6	12 kbit	0

6.5.1.2. PLL

La norma PCI es bastante estricta en cuanto al largo de la pista de reloj. Esta debe de tener 2.5 pulgadas dentro de la placa y debe conectarse a un solo componente. Como se debe proveer de una señal de reloj para el FPGA, la memoria SDRAM y el conector de expansión, se decidió utilizar un circuito regenerador de la señal de reloj PCI con al menos 3 salidas. Esta fue una de las razones para agregar un PLL a la placa. La otra razón era poder generar, a partir de una frecuencia de reloj dada, frecuencias superiores. La frecuencia de reloj del bus PCI es de 33MHz, pero la memoria SDRAM puede funcionar hasta a 133MHz.

Por otro lado como la placa debe de poder funcionar sin estar conectada al PC, el PLL tiene que poder generar el reloj a partir de un cristal.

La elección final fue el modelo 5V925 de IDT. El mismo cumple con los siguientes requerimientos:

- se alimenta de 3.3V, genera una señal entre 0 y 3.3V pero es tolerante a señales de entrada de 5V
- puede regenerar un reloj a partir de una señal de reloj o un cristal
- tiene 4 salidas, una entrega la misma frecuencia de entrada y las 3 restantes entregan la frecuencia de entrada multiplicada entre 2 y 8 veces
- la selección del factor de multiplicación es escogida en forma sencilla (2 jumpers)

El cristal empleado es de 7.164MHz, al ser un múltiplo de la frecuencia de PALN se deja prevista la posibilidad de utilizar la placa para generar señales para televisión.

6.5.1.3. Conversores DC-DC

Los componentes de la placa funcionan con 3.3V y 2.5V. El conector PCI proporciona 3.3V y 5V. Se optó por tomar como entrada de alimentación sólo la fuente de 5V y generar las fuentes de 3.3V y la de 2.5V. De esta forma solo se necesita una fuente que entregue 5 o más volts.

La conversión de voltajes podía ser realizada mediante reguladores lineales (mas baratos, menos componentes pero menos eficientes) o conversores DC-DC (más componentes pero mucho más eficientes).

Para tomar la decisión de cuán importante es la eficiencia de la conversión se hizo la siguiente estimación de consumo de la placa.

SDRAM MT48LC4M32B2
- Utilizando planilla de cálculo proporcionada por el fabricante: 411.6 mW
- Peor caso a 3.3V: 0.125 A (estimamos 0.13 A)

FPGA
- Cálculos utilizando formulas de notas de aplicación de ALTERA.
  - Consumo Interno (de fuente de 2.5V): 0.431 A (estimamos 0.5 A)
  - Consumo Externo (de fuente de 3.3V): 0.412 A (estimamos 0.5 A)

PLL IDT5V925
- De hoja de dato: máximo 0.130 A (estimamos 0.13A)

EPC2
- De hoja de datos: máximo 0.05 A (estimamos 0.05A)

TOTALES
- 0.5A en fuente de 2.5V -> diseñamos fuente 2.5V para 1A
- 0.8A en fuente de 3.3V + 1A adicional para diseños en expansor -> diseñamos fuente 3.3V para 2A

Utilizando un conversor lineal, se disminuye el voltaje manteniendo constante la corriente. Por lo tanto deberían drenarse 3A (2A @ 3.3V y 1A @ 2.5V) de la fuente de 5V.

Utilizando conversores DC-DC, en total se necesitan 3A (2A @ 3.3V y 1A @ 2.5V) o 9.1W. Suponiendo una eficiencia típica de 0.87 para el conversor DC-DC, esto significa que la fuente de 5V debe entregar 10.4W. Por lo tanto deberían drenarse 2.1A de la fuente de 5V

Las motherboards de PC estándar están diseñadas para entregar aproximadamente 2A a 5V por conector PCI. Se optó entonces por utilizar los conversores DC-DC.

Los modelos de componentes escogidos fueron el LT1767EMS8E-2.5 para 2.5V y el LT1506CS8-3.3 para 3.3V.

6.5.1.4. Memoria

Se seleccionó memoria de tipo SDRAM ya que, por el momento, es la memoria más utilizada por la industria de PCs, y por lo tanto, su costo por MB es el más bajo.

Otra razón para seleccionar SDRAM es la existencia de controladores implementados en cores IP de libre distribución, utilizando interfaz Wishbone. Estos cores pueden ser integrados de forma relativamente sencilla en un diseño.

En una primera instancia se pensó en utilizar un DIMM de PC, pero la cantidad de señales necesarias para manejarlos, consumían casi todos las patas del FPGA (un módulo DIMM contiene 168 patas). El costo del conector DIMM también resultaba excesivo y sus dimensiones ocuparían mucho espacio de circuito impreso, aumentando el costo de fabricación de la placa. Por estas razones se optó por soldar un solo integrado de memoria SDRAM directamente sobre la placa.

Se escogió una memoria de ancho de palabra 32 bits para facilitar el diseño de las aplicaciones que hagan uso del bus PCI (este tiene ancho de palabra 32 bits).

La memoria SDRAM escogida fue la MT48LC4M32B2 de la empresa MICRON. Tiene una capacidad de 4M*32, dando un total de 128 Mbits.

Afortunadamente, la memoria SDRAM fue donada por la empresa MICRON.

6.5.2. Decisiones de layout

Una vez que estuvieron los componentes principales seleccionados, se pasó a estudiar como se realizaría su interconexión.

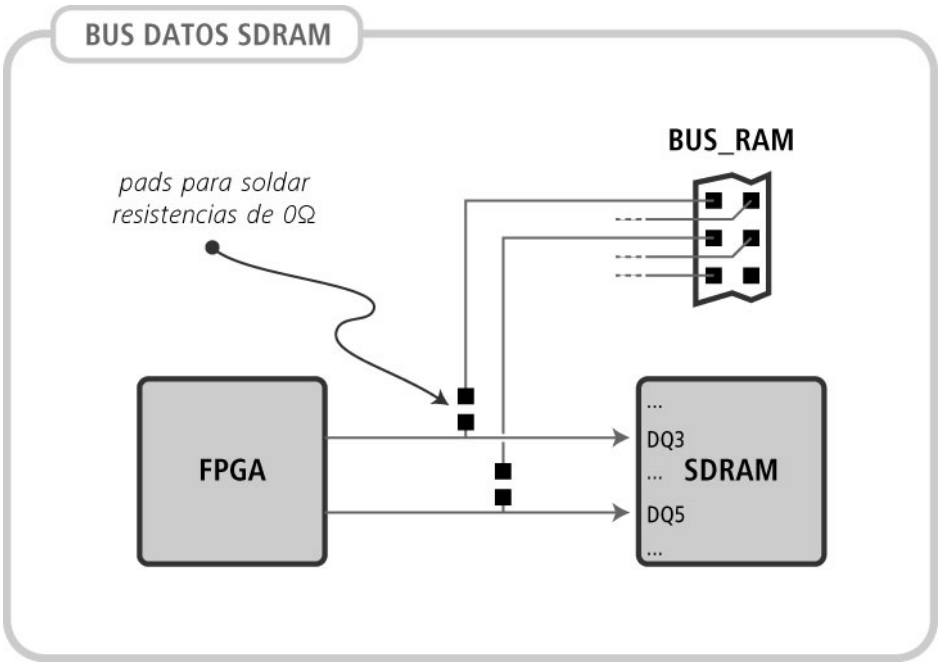
6.5.2.1. Distribución de pines del FPGA

Luego de restar las patas utilizadas para configuración, JTAG y alimentación, quedan disponibles 147 de las 208 patas del FPGA.
Estas fueron distribuidas de la siguiente forma:

- manejar 2 leds,
- recibir 2 señales de reloj (entradas especiales del FPGA para señales de reloj),
- interactuar con la SDRAM (56 patas),
- interactuar con el bus PCI (53 patas),

◦ conectar 4 DIP SWITCHES (patas únicamente de entrada del FPGA) y
◦ conectar el bus de expansión (30 patas).

Las 32 señales del bus de datos de la RAM pueden ser conectadas a uno de los expansores hembra. Las pistas que van desde la memoria hasta el expansor están cortadas. En el corte se dejó previsto un footprint para resistencias. En caso de querer conectar el bus de datos al expansor basta con soldar resistencias de 0 ohms en estos lugares.



Esto fue hecho para no dejar pistas libres sin terminación que pudieran producir ondas reflejadas e impedir la transferencia de datos a alta velocidad entre el FPGA y la memoria.

6.5.2.2. Posición de expansores

Las tiras de conexión de expansores fueron colocadas de modo de servir de conexión eléctrica y a la vez de soporte físico. Si en la placa de expansión que se diseña se colocan los componentes del lado interior (contra la placa IEE-PCI), el ancho total apenas excede el ancho de una ranura PCI.

6.5.2.3. Posición de conectores de programación y dip switches.

Los conectores de programación fueron colocados sobre el borde superior de la placa, a modo de simplificar la conexión del cable de programación cuando la placa está

conectada en el bus PCI. Lo mismo fue tomado en cuenta para las ubicar las 4 llaves de dos posiciones (dip switches).

6.5.2.4. Posición de los conversores DC-DC.

Para minimizar las interferencias producidas por la conmutación de los conversores, éstos fueron ubicados sobre el borde opuesto al conector PCI. Esto también facilita la disipación de calor por convección.

6.5.2.5. Posición del FPGA

La posición del FPGA fue elegida de modo de minimizar la distancia de las pistas de conexión con el bus PCI. El FPGA fue centrado respecto al largo del conector PCI.
La orientación fue seleccionada para que la mayor cantidad de patas de I/O estuviesen lo más cerca del conector PCI y las patas de programación lo más cerca del EPC2 y los conectores de programación. Debido a esto sólo quedaba una orientación posible.

6.5.2.6. Posición de capacitores de desacople

Los capacitores de desacople fueron colocados lo más cerca posible de las patas de alimentación de los integrados, como recomiendan los fabricantes. Para lograr este objetivo, muchos capacitores fueron colocados en el lado opuesto al de componentes.

6.5.3. Diseño con señales digitales de alta frecuencia

Para el diseño de la placa nos basamos mayormente en el libro "BLACK MAGIC HI-SPEED DIGITAL DESIGN" [10.1.E] y en la nota de aplicación Nº75 "High-speed board design" de ALTERA.[10.1.B]

Dada la cantidad de señales a encaminar y que muchas de ellas son de alta velocidad, se optó por un diseño de 4 capas, las exteriores se utilizan para señal y las interiores para tener un plano de tierra y potencia. Con esto se consigue tener un plano de tierra muy cerca del plano de señal, reduciendo la impedancia característica de las pistas de alta frecuencia y minimizando la inductancia mutua entre pistas. Se intentó encaminar la mayor parte de señales por la cara de componentes dado que el plano de tierra está más próximo a este.

6.5.3.1. Lineas de transmisión.

Un circuito compuesto por dos dispositivos interconectados mediante una una pista sobre un plano de tierra puede ser estudiado como un circuito de parámetros distribuidos o de parámetros concentrados.

En los circuitos de parámetros concentrados, puede considerarse que el voltaje en toda

la pista es uniforme. Mientras que en los circuitos de parámetros distribuidos, la pista debe ser tratada como una línea de transmisión, que presenta sobretiros, oscilaciones y atenuaciones.

El comportamiento del circuito está determinado por las siguientes relaciones:
◦ distribuido : Tr > 4*Tpd
◦ concentrado : Tr < 4*Tpd
Siendo *Tr* el tiempo de levantamiento de la señal y *Tpd* el tiempo de propagación.

El tiempo de propagación depende de la constante dieléctrica (Er) del material entre la pista y el plano de tierra, y el largo de la pista (l).

$$T_{pd} = l/V_p = l\sqrt{E_r}/C$$

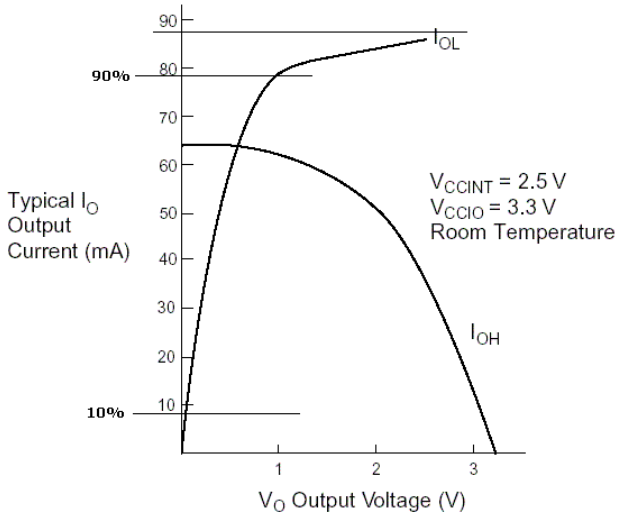
Con

$$C_0 = 3 \times 10^8 m/s$$

Tiempo de levantamiento
El tiempo de levantamiento de la señal está dado por las características del dispositivo que genera la señal y el tipo de carga al que esta conectado.

La carga del bus PCI y la memoria SDRAM pueden aproximarse a capacitores de 40pf.

El siguiente gráfico muestra la corriente de salida entregada por los drivers del FPGA ACEX (Iol) en función del voltaje de salida (Vol).



Basándonos en los cálculos propuestos en la nota de aplicación N°75 de Altera, el tiempo de levantamiento será aproximadamente: 1.1ns

Parámetros distribuidos o concentrados

El largo de pista (*l*) límite a partir del cual el comportamiento del circuito debe considerarse de parámetros distribuidos se puede calcular como:

$$l > T_r C_o/(4\sqrt{E_r}) = 3.9cm$$ /p>

Se tomó en cuenta este valor al momento de ubicar los componentes y encaminar las pistas, tratando, siempre que fuese posible, no superarlo.

6.5.3.2. Crosstalk

El crosstalk entre las pistas es causado por corrientes inducidas entre ellas. Se percibe como pulsos en lineas adyacentes a aquellas lineas en las que se produce un cambio de voltaje.

Este efecto depende de la inductancia mutua entre pistas (Lm), del tiempo de levantamiento de las señales (Tr) y la impedancia característica del circuito que genera el cambio (Ra). Estos términos se relacionan de la siguiente forma:

$$Crosstalk = L_m/(R_a T_r)$$

Por lo general al momento de diseñar el impreso el único parámetro que podemos controlar es Lm, dado que Ra y Tr estarán fijados por los componentes seleccionados. En algunos FPGA de Altera, el valor de Tr puede ser aumentado al momento de programarlo, mediante una opción de síntesis.

La inductancia mutua puede ser de tipo sélfico o capacitivo, en circuitos digitales el primer tipo es el que más influye en el crosstalk.

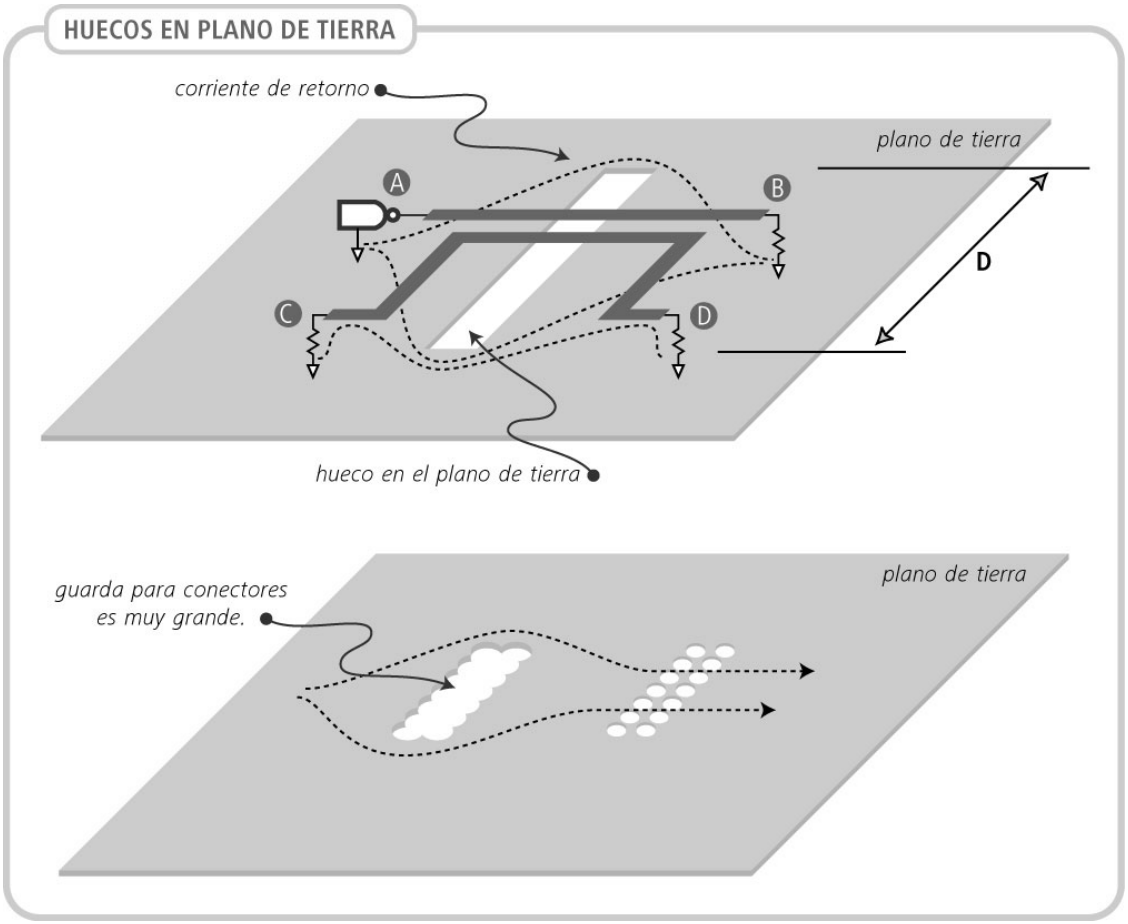
La inductancia mutua entre lineas depende de la distancia entre estas y la distancia al plano de tierra.

$$L_m = L \times \frac{1}{1+(s/h)^2}$$

Donde *L* es la inductancia de una de las pistas, *s* la distancia entre las pistas y *h* la distancia al plano de tierra.

Por lo cual los caminos para disminuir el crosstalk son:
- disminuir la distancia al plano de tierra para disminuir *h*
- aumentar la distancia entre pistas para aumentar *s*
- mantener el valor de *L* tan bajo como sea posible.

Se debe evitar dejar aberturas o ventanas en el plano de tierra, debajo de las pistas de señal, ya que esto hace que el camino de retorno de las corrientes sea más largo aumentando la *L* de la pista.



La corriente de retorno del driver en A no puede fluir directamente debajo de la pista A-B ya que hay un hueco en el plano de tierra. Por lo tanto la única opción posible es que fluya alrededor del hueco. Este camino forma un lazo aumentando así la inductancia del camino A-B.
La corriente desviada también se solapa con el lazo de corriente formado por pista C-D y su camino de retorno de corriente. Este solapamiento produce una gran inductancia mutua entre las pistas A-B y C-D.
La inductancia de la traza A-B es aproximadamente:

$$L = 5D\ln(D/W)$$

Donde *L* es la inductancia, *D* es el largo del hueco en sentido perpendicular a la pista, y *W* es el ancho de la pista.

Se tuvo especial cuidado en que no quedaran demasiadas vías juntas que atravesaran el plano de tierra de manera de no originar largas aberturas en dicho plano.

Dado que la *L* también depende del largo de la pista, se intentó minimizar esta distancia siempre que fuese posible.

6.5.3.3. Terminación de lineas y adaptación de impedancias

Diferencias entre la impedancia de un emisor y un receptor hacen que se produzcan reflexiones a lo largo de la línea de transmisión que los une. Estas reflexiones son perjudiciales pues hacen que la línea demore más tiempo en alcanzar el valor de reposos y los cambios de voltaje debido a las continuas reflexiones entre el driver y la carga inducen corrientes en pistas adyacentes, favoreciendo el efecto de crosstalk.

Para minimizar estos efectos se optó por utilizar resistencias en serie a la salida de las patas que manejan lineas de alta velocidad. El propósito de estas es:
- adaptar la impedancia del driver a la de la línea
- atenuar las reflexiones
- aumentar el tiempo de subida de las señales

Dado que la función de esta resistencia es adaptar la impedancia del driver a la de la pista, solo es necesario colocarla en la salida del driver. Por esto se empleó únicamente en líneas manejadas en un solo sentido, por ejemplo pistas de reloj y bus de control de memoria. No se colocaron resistencias en las líneas que comunican el FPGA con el bus PCI o en el bus de datos de la memoria por ser bidireccionales.

El valor de la resistencia serie debe ser elegido para que la suma de la impedancia del driver más la de la resistencia sea igual a la impedancia de la pista.

Cálculo de impedancia de las pistas

De la bibliografía consultada obtenemos que la impedancia de las pistas puede aproximase por:

$$Z_0 = \frac{87}{\sqrt{E_r + 1.41} \times \ln(5.98 \times \frac{h}{0.8w+t})}$$

Siendo:
- Er: constante dieléctrica del material

- h: distancia entre la pista y el plano de tierra (en pulgadas)
- t: espesor de la pista (en pulgadas)
- w: ancho de la pista (en pulgadas)

En el libro BLACK MAGIC HI-SPEED DIGITAL DESIGN se realizan los cálculos con los siguientes valores genéricos:
- Er: 4.6
- h: .005 pulgadas
- t: .002 pulgadas
- w: .008 pulgadas

Zo=45 ohm.

La nota de aplicación Nº75 de Altera recomienda utilizar resistencias en serie de 33 ohm, cuyo valor es consistente con los cálculos realizados anteriormente.
Por esta razones se adquirieron resistencias de 33 ohm.

El proceso de fabricación escogido hace que los parámetros sean los siguientes:
- Er: 4.6
- h: .012 pulgadas
- t: .002 pulgadas
- w: .008 pulgadas

Para estos valores, la impedancia es de Zo = 76 ohm

En este caso, el valor de las resistencias de ajuste de impedancia debería ser de cerca de 60 ohm, pero ya habían sido adquiridas de 33 ohm

Esta diferencia de valores puede hacer que el ajuste de impedancias no este siendo tan eficiente, pero dado que se está en el entorno de los valores a partir de los cuales las pistas deben ser modeladas como lineas de transmisión, es de esperar que no surjan inconvenientes.
En la práctica, no hemos tenido ningún tipo de problema.

6.5.3.4. Retardos de propagación en pistas de reloj.

La señal de reloj debe ser distribuida desde el PLL hasta el FPGA, la memoria y el conector de expansión.

En un diseño digital, se debe de asegurar que los flancos de la señal de reloj lleguen al

mismo tiempo a todos los componentes para preservar el sincronismo.
Para esto debe asegurarse que todas las pistas de reloj tengan longitudes similares.

En nuestro diseño se encaminó primero la pista más larga y finalmente se agregaron bucles en el resto de las pistas de forma de mantener igual su longitud.

6.5.3.5. Distribución de potencia a integrados

Para disminuir el ruido de baja frecuencia causado por la fuente de poder debe filtrarse la alimentación en los puntos de entrada a la placa y en las patas de alimentación de los dispositivos.
Para eso Altera recomienda utilizar en los puntos de entrada capacitores electrolíticos de 100 uF. En los casos en que se usan reguladores de voltaje o conversores dc-dc, estos capacitores deben colocarse lo más cerca posible de la salida.
Estos capacitores no solo filtran el ruido de baja frecuencia de la fuentes sino que proveen la corriente necesaria cuando varias salidas cambian simultáneamente en el circuito.

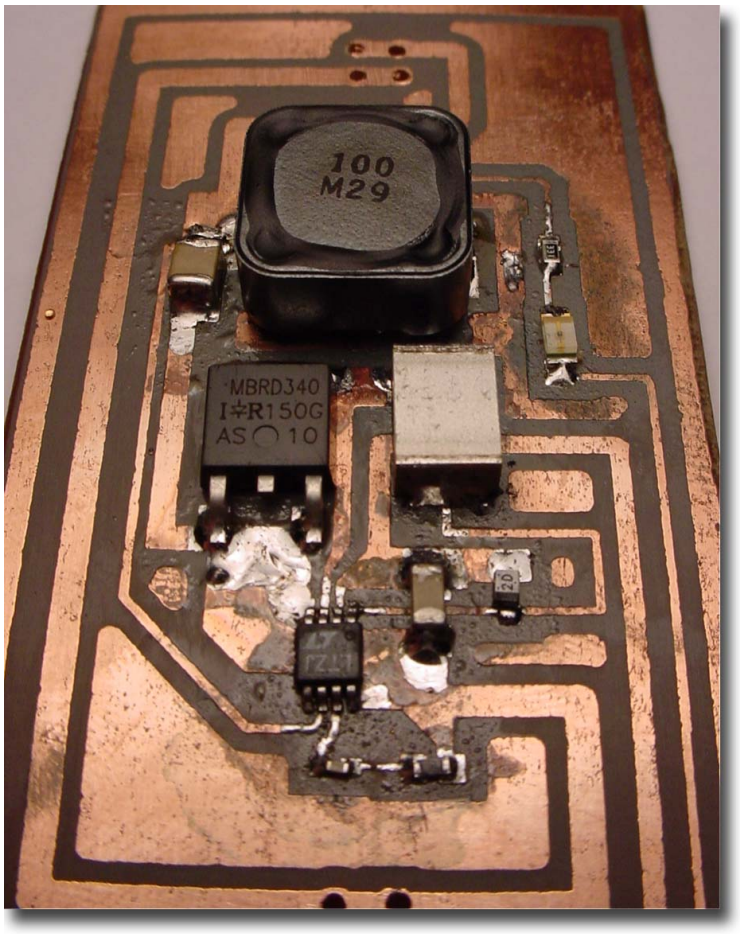
Los integrados producen ruido de alta frecuencia en el plano de alimentación. Para filtrar este ruido generalmente se utilizan capacitores de desacople colocados lo más cerca posible de las patas de alimentación y tierra del dispositivo.
El uso de planos de alimentación y tierra paralelos y separados por un dieléctrico también funciona como capacitor de desacople, reduciendo el ruido de alta frecuencia.

Dado que un plano de alimentación cubre un área relativamente grande, su resistencia es baja, permitiendo la distribución uniforme de la alimentación a todos los componentes de la placa.

6.6. Fabricación

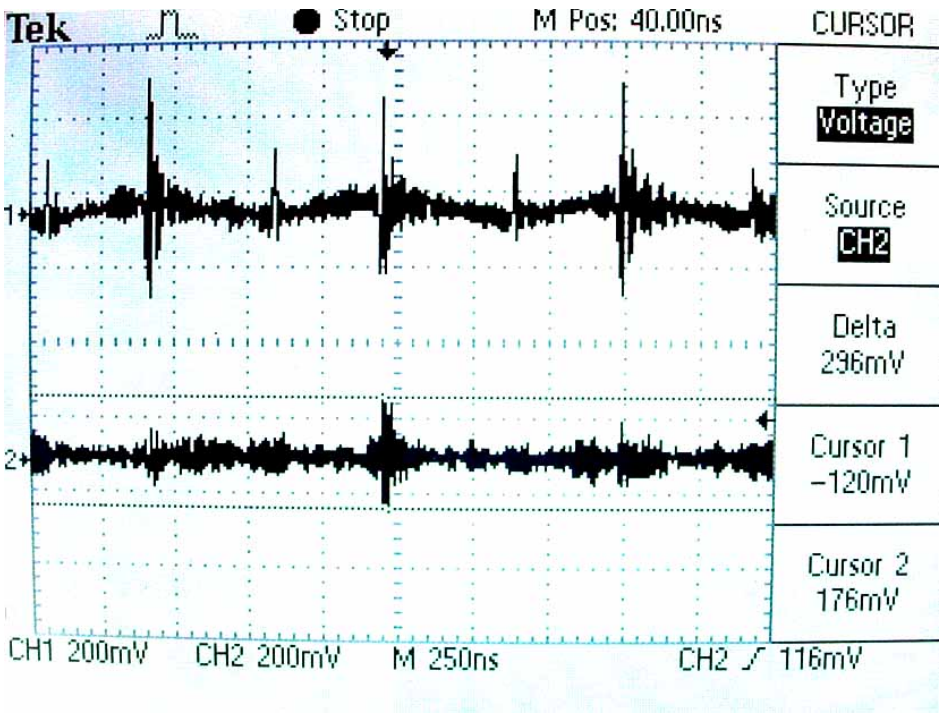
6.6.1. Fabricación de impresos de prueba de conversores dc-dc y PLL

Para estar seguros de que los diseños de los conversores dc-dc eran correctos, se realizaron circuitos impresos de prueba de una capa. La disposición de los componentes y la forma de las pistas era lo más parecida posible a la que se utilizaría en la placa IIE-PCI.



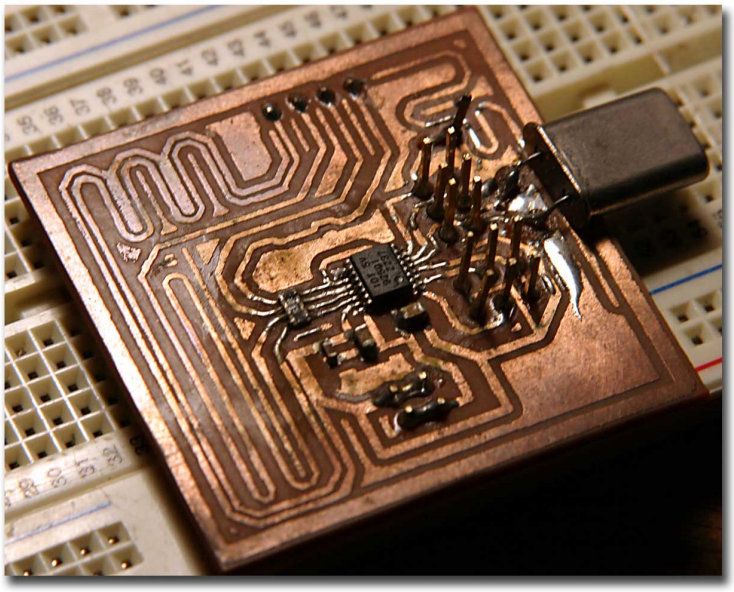
Esta placa permitió hacer pruebas de la estabilidad de los conversores frente a cargas cercanas al valor máximo para el que fueron diseñados. Se observó que al colocar 2 capacitores en paralelo en la salida, el ripple en la fuente disminuía debido a que bajaba la resistencia serie equivalente de los capacitores. Por este motivo se dejó

previsto un par de pads extra en cada salida de los conversores para soldar otro capacitor en caso de ser necesario.



El gráfico superior muestra la salida del regulador con un capacitor de 10uF en la salida. El inferior con un capacitor de 10uF en paralelo con uno de 12uF. Todos los capacitores son cerámicos y de baja resistencia serie equivalente.

También se fabricó un circuito de prueba para el PLL, de forma de estar seguros de su correcto funcionamiento con el cristal seleccionado.



La experimentación necesaria para realizar estas placas de pruebas caseras fue transmitida y aportó ideas que han sido provechosas más allá del grupo de proyecto. El proceso se documentó en: http://iie.fing.edu.uy/~sebfer/pcb/index.htm.

6.6.2. Herramientas

6.6.2.1. Software de layout.

La placa fue diseñada utilizando la versión de prueba del software Protel (www.protel.com).

Se probaron otros programas de diseño y se eligió éste debido a que era relativamente sencillo de utilizar y era suficientemente potente como para nuestro diseño.

El Protel es capaz de generar archivos gerber a partir de los diseños realizados. Los archivos gerber son utilizados ampliamente por la industria de fabricación de circuitos impresos.

El enrutamiento automático fue descartado luego de varios intentos fallidos. Se probaron enrutamientos automáticos completos y parciales (con ciertos grupos de señales enrutadas previamente en forma manual), pero ninguno fue satisfactorio, ya que utilizaban una excesiva cantidad de vías o no lograban una disposición óptima para las pistas.

Finalmente se realizó manualmente el enrutamiento de todas las señales.

6.6.2.2. Software de visualización de archivos gerber

Para verificar los diseños se utilizó el visor de archivos gerber View Mate, de la empresa Pentalogix (www.pentalogix.com).

6.6.3. Fabricación del impreso

Se fabricaron dos placas en PCBExpress (www.pcbexpress.com) a un costo de U$S150 por placa.

Características de la fabricación:

- 4 capas
- agujeros metalizados
- máscara antisoldante LPI (Liquid Photo Imageable) con la que se logran líneas de una precisión de 8 a 5 milésimas de pulgada (mil).
- máscara silkscreen (leyendas en lado de componentes)
- ancho mínimo de pistas: .007 (7 mil).
- espesor de dieléctrico (FR4):
  - entre capas 1-2 y 3-4: .012"
  - entre capas 2-3: .028"
- constantes dieléctricas del FR4
  - entre capas 1-2 y 3-4: 4.6
  - entre capas 2-3: 4.7
- espesor de cobre:
  - capas 1 y 4: .002"-.0025"
  - capas 2 y 3: .0012"-.0013"

Los archivos necesarios para la fabricación son los siguientes:

- TOP Silkscreen: iiepci_top_silkscreen.GTO
- TOP solder mask: iiepci_top_solder.GTS
- TOP Signal Layer: iiepci_top_layer.GTL
- TOP Internal Plane: iiepci_top_internal_layer.GP1
- BOTTOM Internal Layer: iiepci_bottom_internal_layer.GP2
- BOTTOM Signal Layer: iiepci_bottom_layer.GBL
- BOTTOM Silkscreen: iiepci_bottom_silkscreen.GBO
- BOTTOM Solder Mask: iiepci_bottom_solder.GBS
- NC Drill: iiepci_NC_drill.TXT

6.7. Montaje

Se montaron los conversores DC-DC y se verificó que el voltaje obtenido era el deseado.

Para montar la primera placa, por falta de experiencia el la soldadura de integrados de distancia entre patas de 1.60mm, recurrimos a CCC del Uruguay S.A. Gentilmente, la gente de CCC realizó la soldadura del FPGA y de la memoria SDRAM.

El resto de los componentes superficiales (capacitores, pll, zócalo EPC2, resistencias, etc.) fueron soldados por los integrantes del proyecto. Para esto, sólo fue necesario un soldador JBC de 25W de punta fina (1.5 mm), una pinza fina y flux líquido.

El montaje de la segunda placa y el cambio del FPGA y la memoria de la primera placa fueron realizadas enteramente por los integrantes del proyecto, utilizando los siguientes elementos:

- microscopio bifocal, de 20 aumentos y 10mm de diámetro de campo visual
- soldador de punta fina (0.4mm) de temperatura regulable
- pinzas chatas
- tira de cobre trenzado para quitar estaño
- flux líquido
- paciencia

Aunque se utilizó un microscopio de 20 aumentos (con zoom), se comprobó que 12 aumentos son suficientes. Es imprescindible que sea bifocal y que tenga suficiente campo. La iluminación coaxial (mismo eje visual) tampoco es necesaria, es más, el brillo que produce en las patas metálicas hace el trabajo más dificultoso.

6.7.1. La aventura de la soldadura superficial en casa

Y se mandaron a la guerra con un tenedor. En realidad era más que un tenedor.

El soldador de punta fina (0.4 mm) lo usaron para tocar las patas de las cucarachas negras, de a una sola a la vez. Se los prestó gentilmente Etienne, y tenía una perilla para poder elegir qué era lo que querían derretir.

Pero las patitas eran tan finitas, o los gurises estaban tan chicatos ya, que para poder ver algo no tuvieron más remedio que pedirle la lupa al Dr.Mondueri. La

lupita era enorme, de microcirugía, decía. Parece que con 15 aumentos ya daba, pero eso si, tiene que ser para los dos ojos, sino no se sabe a que altura uno viene arrimándose con el soldador y puede hacer cualquier desastre.

Por ahí habían escuchado que el flux era bárbaro para que el pegote saliera bien prolijo. Así que se compraron algo que se parecía a un sylvapen bien grueso llenito de flux, y no dudaron ni un segundo: antes de poner la cucaracha, vamos a pasarle esta agüita sobre los pads.

Apoyaron las cucarachas sobre la placa, y como era de esperar, no se quedaban quietas. Iban de un lado a otro y nunca querían dejar las patas quietas sobre los pads. Entonces surgió la idea de meterle un dedo encima, pero despacito, ya que no les queríamos quebrar ninguna pata, pobrecitas. Para la cucaracha grande, estaba bien, pero para la chiquita, era imposible. Entonces el Dr. les prestó una pinza chata. Con la pinza se aguantaba la cucaracha por el lomo, y con la otra arrimaba el soldador para pegarle un par de patitas de cada esquina y ver si iba quedando derechita en su lugar, inquieta.

Que descubrimiento el flux! Al arrimar el soldador, tocando sobre el pad, el estaño que ya venía en la placa se fundía y gracias al flux, corría y se le subía por la patita de la cucaracha, dejando un menisco de estaño muy bonito.

En muy pocas patitas, faltaba un poco de estaño, pero la punta del soldador era muy finita y el estaño tiene algo que se llama tensión superficial, entonces no había forma sencilla de cargar una nadita en la punta del soldado, era o nada o mucho. Pero con un poco de maña se lograba hacer alguna pelota de estaño y dejar tremendo cortocircuito. Con una malla de cobre para quitar estaño, se dejaba todo limpito de nuevo, pero había una pata que no quería. Terca la pata, no quería que el estaño se derritiera. Intrigados, le preguntaron porqué, y la cucaracha les enseño que esa era una pata de tierra, y que se habían olvidado del "thermal relief". Pero claro, se habían olvidado, y el pad estaba todo pegado al plano de tierra, y el soldador no le hacía ni cosquillas.

Ahí fue que sacaron del bolsillo el soldador de siempre, ese que compraron en primero de facultad, el de punta medianita (2mm). Al arrimarlo era del ancho de 3 patas de la cucaracha mínimo, pero con cuidadito pudieron limpiar el enchastre.
Pero les faltaba una herramienta, y la fueron a buscar, que estaba bien guardadita.
Se hicieron de la paciencia que les estaba faltando, una bien grande.

Y así, despacito, fueron pegando patita por patita, hasta que quedaron todas bien pegadas.

Te vas a mover, si, seguro.

6.7.2. Recomendaciones para soldadura superficial casera

Antes que nada es interesante afirmar que es perfectamente posible soldar en casa componentes superficiales, incluyendo integrados de alta densidad de patas, como ser el FPGA o la SDRAM de la placa.

Las herramientas necesarias son:
- lupa o microscopio bifocal de 10 a 15 aumentos
- soldador de punta fina (0.4 mm)
- soldador de punta mediana (1 o 2 mm)
- flux líquido
- tira de cobre trenzado para quitar estaño
- pinzas chatas
- paciencia

El microscopio o la lupa permiten ver la soldadura que se está realizando. Las patas y los pads son demasiado pequeños para trabajar con ellos a simple vista.

Se debe empezar por soldar los circuitos más complicados, en nuestro caso los integrados de FPGA y SDRAM. De esa forma se logra trabajar con una placa sin componentes que puedan llegar a molestar al acercar el soldador.

Pre-estañar los pads de los integrados no es necesariamente bueno, ya que el estaño se acumula y no deja que todas las patas apoyen a la vez. Basta utilizar flux que haga correr el estaño que ya está en el pad y el que se agrega con la punta del soldado.

Las patas que conectan a planos de tierra y alimentación, si no tienen thermal reliefs, son muy difíciles de soldar, y más complicado aún es limpiar el estaño que se acumule en ellas, ya que entre el calor disipado por el plano y la cinta de cobre, el soldador no es capaz de derretir el estaño.

Después de soldar los circuitos integrados, es conveniente empezar a colocar el resto de los componentes desde adentro hacia afuera, para no molestarse al soldar, y dejar

los componentes más altos para el final.

El soldador fino es solamente útil para las patas bien finas de los integrados. Para el resto es más útil el soldador mediano, ya que entrega el calor necesario por los pads y componentes de más tamaño. Hasta los bancos de resistencias son más cómodos de soldar con el soldador mediano.

El flux es indispensable, casi tanto como el estaño. Es lo que permite que las soldaduras pequeñas queden bien terminadas.

6.8. Costo de la placa

El costo de la placa se detalla a continuación:

Item	Costo (en dólares americanos)
FPGA	70
SDRAM	7.8
EPC2	33.5
otros componentes	69.671
circuito impreso	150
total	330.971

El costo supera los U$S 250 esperados. Hay que tener en cuenta que este es el costo de un prototipo, y simplemente haciendo una tirada de 10 impresos, el costo unitario sería de U$S 48, logrando así un costo final de U$S 230.
También hay que tener en cuenta que el FPGA utilizado es el más rápido de la gama, y por ende el más caro.

6.9. Pruebas

Para probar el funcionamiento de FPGA y la EPC2 en la placa, se realizó un diseño sencillo que manipula los leds de acuerdo a la posición de los dip switches y las señales de reloj provenientes del PLL.

Para probar el funcionamiento de la memoria SDRAM, se realizó un diseño que escribe y lee datos del la memoria, comparando lo leído con lo escrito. Para esto se utilizó un core IP controlador SDRAM disponible en opencores.org.

Para probar el funcionamiento del FPGA con el bus PCI se utilizó el core PCI de Altera y una aplicación que consistía en una memoria RAM implementada dentro del mismo FPGA. Se verificó que la placa era reconocida por el sistema y se podían realizar lecturas y escrituras a la RAM dentro del FPGA.

En una de las pruebas, la placa se conectó corrida en el bus PCI lo que produjo daños irreparables en el FPGA y supuestamente en la memoria RAM.

6.10. Conclusiones

El objetivo de bajo costo fue alcanzado, ya que de realizar un tiraje de 10 placas el costo sería de U$S230.

Fue posible diseñar, fabricar y probar una placa PCI de propósito general, que funcionó en el primer intento. El diseño es satisfactorio y cumplió con los objetivos propuestos.

Realizando las pequeñas correcciones mencionadas más abajo, es una placa que puede producirse para ser utilizada en diseños genéricos que hagan uso del bus PCI.
Sin crear grandes expectativas, creemos que es un producto final que puede posicionarse favorablemente como placa de desarrollo. Todos los componentes utilizados son fácilmente conseguibles, se utilizaron FPGAs y memorias baratas y la fabricación del circuitos impresos puede realizarse a costos razonables. El montaje de las partes tampoco requiere equipamientos demasiado sofisticados.

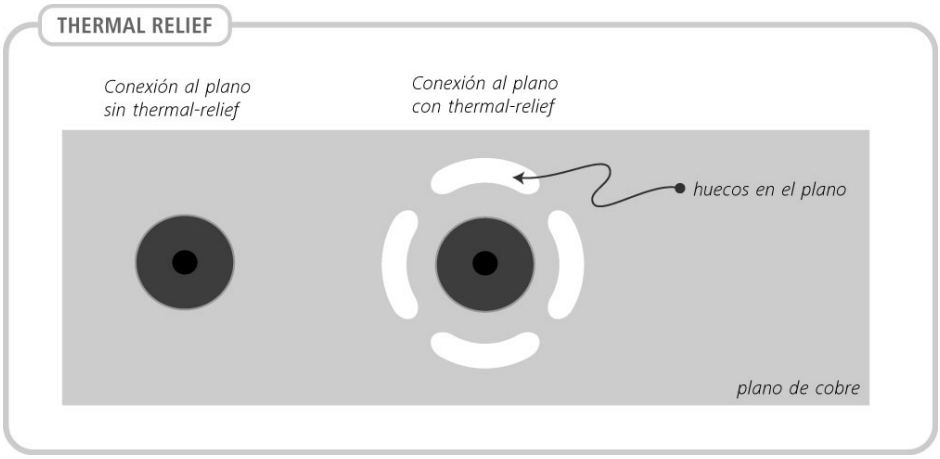
6.10.1. Mejoras y recomendaciones

Afortunadamente no hubo ningún error de gravedad tal que impidiera utilizar el diseño original de las placas. Sin embargo, mencionamos a continuación posibles mejoras o correcciones a tener en cuenta.

6.10.1.1. Contactos a planos de alimentación y tierra

Realizar soldaduras en las patas de alimentación y tierra de los componentes resultó dificultoso debido a que el gran tamaño de los planos de tierra y alimentación disipa el calor y no permite llegar a la temperatura de fusión del estaño. Para esto se debieron haber realizado las conexiones de los pads de soldadura a los planos a través de "thermal reliefs".

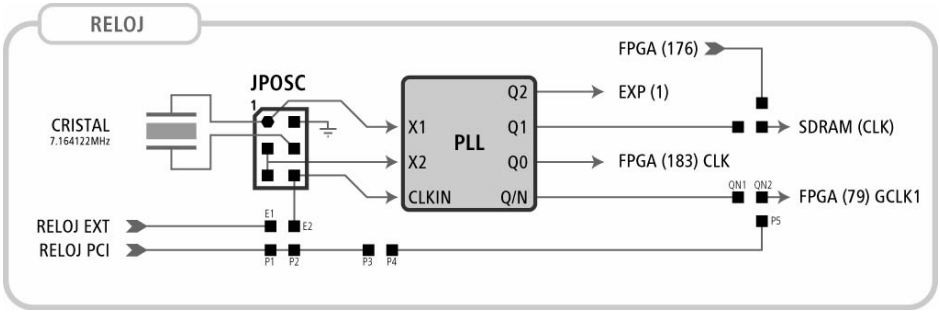
La idea es disminuir el área de contacto con el plano en cuestión. Esto se logra haciendo huecos entre el pad o vía y el plano, cómo se indica en el siguiente diagrama:



6.10.1.2. Señales de reloj

Se proponen una serie de modificaciones que permitirían utilizar una combinación más variada de fuentes de reloj, incluyendo una entrada externa de reloj para que el funcionamiento stand-alone no este limitado a frecuencias del cristal y sus múltiplos.

El diagrama modificado sería el siguiente:



Las combinaciones de soldaduras de resistencias de 0 ohm entre pads de acuerdo a cómo se desean encaminar las señales de reloj se muestran en la siguiente tabla:

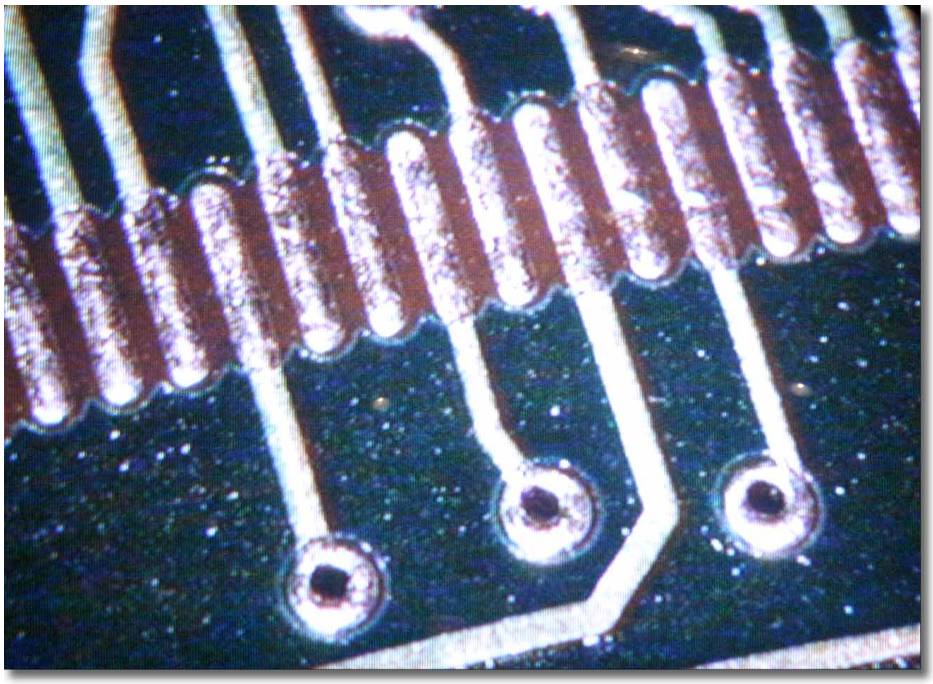
FPGA GCLK1	FPGA CLK, SDRAM y EXP	Cortocircuitar
PCI	PCI x N	P1-E2, QN1-QN2
PCI	cristal x N	P3-P4, P5-QN2
PCI	reloj externo x N	P3-P4, P5-QN2, E1-E2
cristal	cristal x N	QN1-QN2
reloj externo	reloj externo x N	E1-E2, QN1-QN2
reloj externo	cristal x N	E1-P1, P3-P4, P5-QN2

6.10.1.3. Jumpers cerca de zócalo EPC2

Los jumpers de PLL fueron colocados muy cerca del zócalo de la EPC2, lo que dificulta su extracción con la pinza para encapsulados PLCC.

6.10.1.4. Modificar máscara antisoldante

Al momento de generar la máscara antisoldante se debe verificar que esté presente entre las patas de los integrados. Esto facilita el trabajo de soldadura ya que dificulta la formación de gotas de estaño entre las patas. En la placa IIE-PCI, no hay máscara antisoldante entre las patas del FPGA y de la memoria SDRAM.



6.10.1.5. Omisiones en silkscreen

Las indicaciones sobre posiciones de los jumpers no se incluyeron en el archivo de silkscreen.
Tampoco se incluyó la polaridad de los LEDs.

6.10.1.6. Compra de componentes previa a realizar el layout de la placa.

Es recomendable adquirir todos los componentes antes de realizar el layout de la placa, ya que a veces no es sencillo conseguir los componentes deseados. Al elegir un reemplazo, no siembre posee el mismo footprint o encapsulado, lo cual indefectiblemente modifica el layout del circuito impreso.

Es extremadamente útil poseer los componentes ya que al colocarlos todos sobre una impresión en papel del diseño de la placa, pueden indicar eventuales problemas que no son visibles en un simple dibujo. También permite corroborar que los footprints de los componentes coinciden con el diseño realizado.

7. Core PCI PCITWBM

7.1. Introducción

El core PCI PCITWBM es una implementación de la interfaz PCI para ser utilizada por diseños que hagan uso del estándar Wishbone.
Su funcionamiento se puede interpretar como el de un puente entre los dos buses, el PCI [10.1.I] y el Wishbone [10.1.M], es decir envía y recibe datos de un bus a otro.

7.1.1. Organización del capítulo

El capítulo se encuentra organizado en las siguientes secciones:

Características del Core PCI
Características técnicas del core.

Descripción general
Breve descripción del core, diagrama de señales manejadas y ejemplo de uso.

Uso del core PCITWBM
Interfaz, parámetros y jerarquía de archivos VHDL

Interfaz PCI
Introducción a PCI, descripción de las señales PCI del core y espacio de configuración PCI.

Interfaz Wishbone
Introducción a Wishbone, descripción de las señales Wishbone del core.

Arquitectura y funcionamiento
Diagramas de bloques, funcionamiento del puente PCI-Wishbone, máquinas de estado.

Herramientas
Herramientas utilizadas y opciones de compilación.

Conclusiones
Conclusiones, mejoras y recomendaciones.

7.2. Características del core PCI

Principales características del core PCI:

- funcionamiento PCI Target con las siguientes características:
  - 32 bit de ancho de palabra
  - detección de errores de paridad
  - hasta 6 registros de dirección de base (BARs) con tamaño y tipo ajustable en el momento de síntesis.
- soporte de la mayoría de los comandos PCI, incluyendo:
  - lectura y escritura de configuración
  - lectura y escritura de memoria
  - lectura y escritura de I/O
- soporte de transferencias en modo burst
- funcionamiento comprobado utilizando FPGA ACEX EP1K100 de Altera en buses PCI de 33MHz
- desarrollado en lenguaje VHDL
- interfaz de aplicación Wishbone compatible
- la aplicación y el bus PCI pueden utilizar diferentes relojes.
- no se implementó el manejo de interrupciones

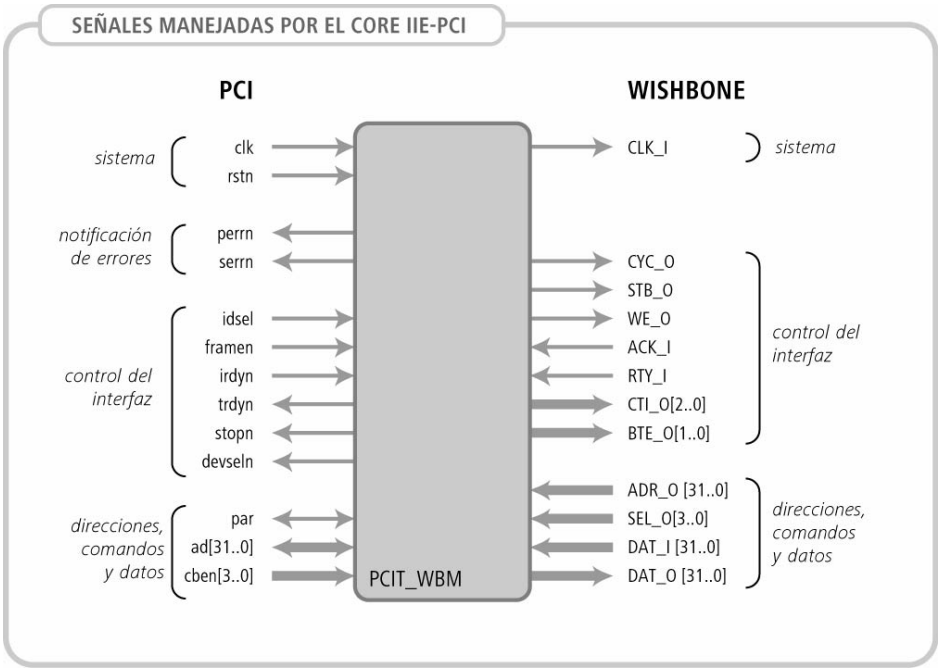
7.3. Descripción general

El core PCI es una implementación del interfaz PCI para ser utilizada por diseños que hagan uso del estándar Wishbone.
Su funcionamiento se puede interpretar como el de un puente entre los dos buses, el PCI y el Wishbone, es decir envía y recibe datos de un bus a otro.

La ventaja de pasar de un tipo de bus a otro radica en que la especificación Wishbone esta pensada para interconectar diseños dentro de un mismo integrado. Las interfaces y los ciclos de transferencia de datos son iguales para todos los cores IP sin importar su función (controlador de memoria, interfaz PCI, registros, etc.) y no es necesario conocer el funcionamiento del bus PCI para hacer un diseño. Esto además de facilitar la tarea, permite la reutilización. Si ya existe un controlador de memorias con interfaz Wishbone, basta con diseñar la interconexión entre dicho controlador y el core IIE-PCI.

Fue concebido para funcionar como un dispositivo PCI target. Esto significa que el core, por si mismo, no es capaz de iniciar una transferencia en el bus PCI, sino que requiere de un dispositivo PCI master que sea el responsable de iniciar la transacción.
Esto no es una limitante ya que en la mayoría de las aplicaciones el core es utilizado como Target, siendo el CPU el que inicia las transacciones. Las lecturas y escrituras se realizan cuando la aplicación software lo requiere.

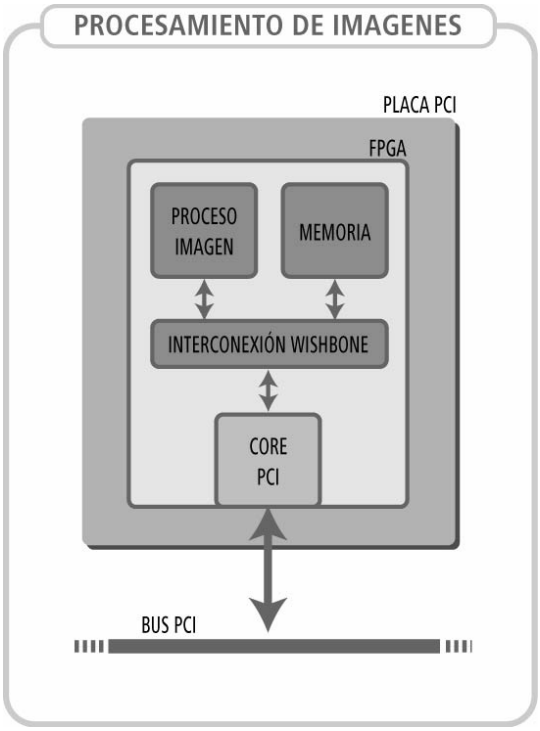
Visto del lado de la aplicación que lo utiliza, el core PCI desarrolla el rol de master Wishbone, es decir, el es capaz de iniciar una transferencia, seleccionando previamente la dirección destino de ésta.



7.3.1. Ejemplo de uso

Un posible ejemplo de uso de una placa PCI con memoria puede ser la aplicación de filtros a una imagen. Para llevar a cabo esto se debe transferir la imagen a una memoria en la placa, aplicar los filtros y luego leer la imagen procesada.

Existen en internet cores IP controladores de memoria con interfaz Wishbone, el core PCI también es compatible con la especificación, por lo que solo se debe diseñar una aplicación compatible con Wishbone que lea una imagen de una memoria, la procese y vuelva a escribirla. No es necesario conocer como funciona la memoria ni como se accede a ella, basta con conocer los ciclos de lectura y escritura Wishbone.


PROCESAMIENTO DE IMAGENES
PLACA PCI
FPGA
PROCESO IMAGEN
MEMORIA
INTERCONEXIÓN WISHBONE
CORE PCI
BUS PCI

7.4. Uso del core PCITWBM

7.4.1. Interfaz y parámetros del core PCITWBM

El core PCI PCITWBM está escrito en VHDL. Su diseño está divido en varios bloques, el de más alta jerarquía es *pcitwbm_top*.

Declaración del componente pcitwbm_top

```
COMPONENT pcitwbm_top
    GENERIC (vendor_id: unsigned:= X"1172";
             device_id: unsigned := X"ABBA";
             subsystem_id: unsigned := X"10E9";
             subsystem_vid: unsigned := X"10E9";
             NUMBER_OF_BARS : integer := 3;
             BAR_0_SIZE : integer := 8192;
             BAR_0_LOW_NIBBLE: integer := 0;
             BAR_1_SIZE : integer := 8192;
             BAR_1_LOW_NIBBLE: integer := 0;
             BAR_2_SIZE : integer := 8192;
             BAR_2_LOW_NIBBLE: integer := 0;
             BAR_3_SIZE : integer := 65536;
             BAR_3_LOW_NIBBLE: integer := 0;
             BAR_4_SIZE : integer := 65536;
             BAR_4_LOW_NIBBLE: integer := 0;
             BAR_5_SIZE : integer := 65536;
             BAR_5_LOW_NIBBLE: integer := 0;
             FIFO_NUMWORDS: integer:= 14;
             LAT_TIMER_INITIAL_VALUE : integer := 7;);
    PORT(
      -- PCI
      rstn     : IN  STD_LOGIC;
      clk      : IN  STD_LOGIC;
      irdyn    : IN  STD_LOGIC;
      idsel    : IN  STD_LOGIC;
      framen   : IN  STD_LOGIC;
      cbe      : IN  STD_LOGIC_VECTOR(3 downto 0);
      devseln  : OUT STD_LOGIC;
      stopn    : OUT STD_LOGIC;
      trdyn    : OUT STD_LOGIC;
      serrn    : OUT STD_LOGIC;
      perrn    : OUT STD_LOGIC;
      ad       : INOUT  STD_LOGIC_VECTOR(31 downto 0);
      par      : INOUT  STD_LOGIC;
      -- WB
      CLK_I    : IN  STD_LOGIC;
      DAT_I    : IN  STD_LOGIC_VECTOR(31 downto 0);
```

```
      DAT_O    : OUT STD_LOGIC_VECTOR(31 downto 0);
      ACK_I    : IN  STD_LOGIC;
      ADR_O    : OUT STD_LOGIC_VECTOR(31 downto 0);
      CYC_O    : OUT STD_LOGIC;
      RTY_I    : IN  STD_LOGIC;
      SEL_O    : OUT STD_LOGIC_VECTOR(3 downto 0);
      STB_O    : OUT STD_LOGIC;
      WE_O     : OUT STD_LOGIC;
      CTI_O    : OUT STD_LOGIC_VECTOR(2 downto 0);
      BTE_O    : OUT STD_LOGIC_VECTOR(1 downto 0)
      );
END COMPONENT;
```

7.4.1.1. Parámetros

VENDOR_ID

Tipo: unsigned
Valor por defecto: X"1172"
Descripción: indica fabricante del dispositivo

DEVICE_ID

Tipo: unsigned
Valor: X"ABBA"
Descripción: identifica tipo o modelo del dispositivo

SUBSYSTEM_ID y SUBSYSTEM_VID

Tipo: unsigned
Valor por defecto: X"10E9"
Descripción: identifica la aplicación implementada

NUMBER_OF_BARS

Tipo: integer
Valor por defecto : 1
Descripción: número de BARs utilizados. Valores entre 1 y 6

BAR_i_SIZE

Tipo: integer
Valor por defecto : 8192 o 65536
Descripción: tamaño del rango del i-ésimo BAR. El valor debe ser una potencia de 2.

BAR_i_LOW_NIBBLE

Tipo: integer
Valor por defecto : 0

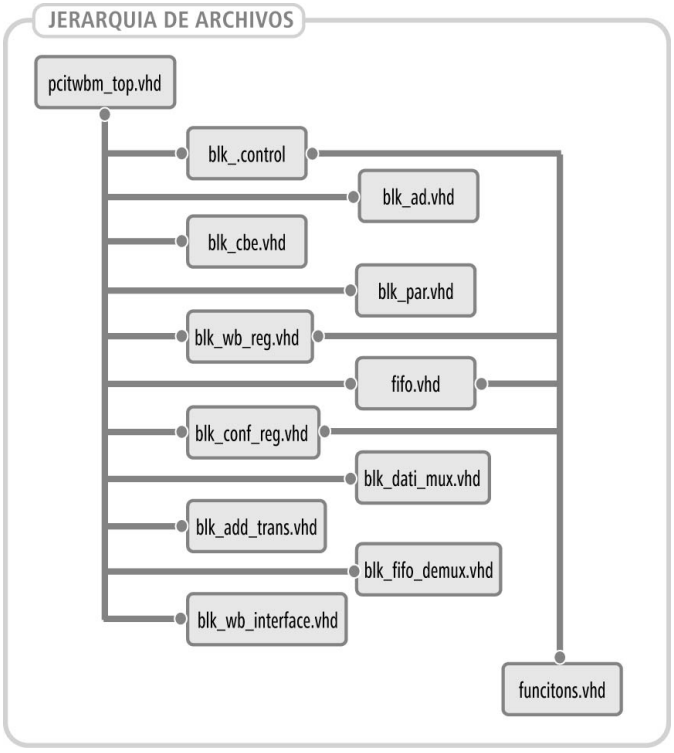
Descripción: identifica el tipo de memoria asignado al i-ésimo BAR

FIFO_NUMWORDS
Tipo: integer
Valor por defecto : 14
Descripción: profundidad de los FIFO entre la interfaz Wishbone y PCI

LAT_TIMER_INITIAL_VALUE
Tipo: integer
Valor por defecto : 7
Descripción: en caso de que el fifo de lectura se llene o el de lectura este vacío, el core espera LAT_TIMER_INITIAL_VALUE períodos de reloj e inicia un ciclo de desconexión.

7.4.2. Jerarquía de los archivos VHDL

El siguiente esquema muestra la jerarquía de los archivos .vhd que componen el core PCI.



El orden de síntesis debe de ser del de menor al de mayor jerarquía.

7.4.3. Recursos utilizados

A modo de comparación, se sintetizó un core con:

- 3 BARs
- tamaño BAR 0 (bits): 8096
- tamaño BAR 1 (bits): 1048576
- tamaño BAR 2 (bits): 4194304
- profundidad de FIFO: 7

Para el pasaje de VHDL a EDIF se utilizó SynplifyPRO 7.0.1 y para la síntesis final Max+PlusII 10.2.

Lógica	Optimización	No.BAR	Bits(%)	LCs	Max. reloj PCI	Max. reloj WB
EP1K100-1	área	3	1216(2%)	1137(22%)	50.76MHz	50.76MHz
	velocidad	3	1216(2%)	1276(25%)	56.81MHz	62.89MHz
EP1K100-2	área	3	1216(2%)	1137(22%)	38.91MHz	37.73MHz
	velocidad	5	1216(2%)	1276(25%)	42.73MHz	46.29MHz

Los valores deben ser tomados como una referencia para comparar la implementación del core en cada FPGA, ya que el tamaño y velocidad final dependen fuertemente de la aplicación con la cual se sinteticen.

7.5. Interfaz PCI

7.5.1. Introducción al interfaz PCI

7.5.1.1. Historia y características principales

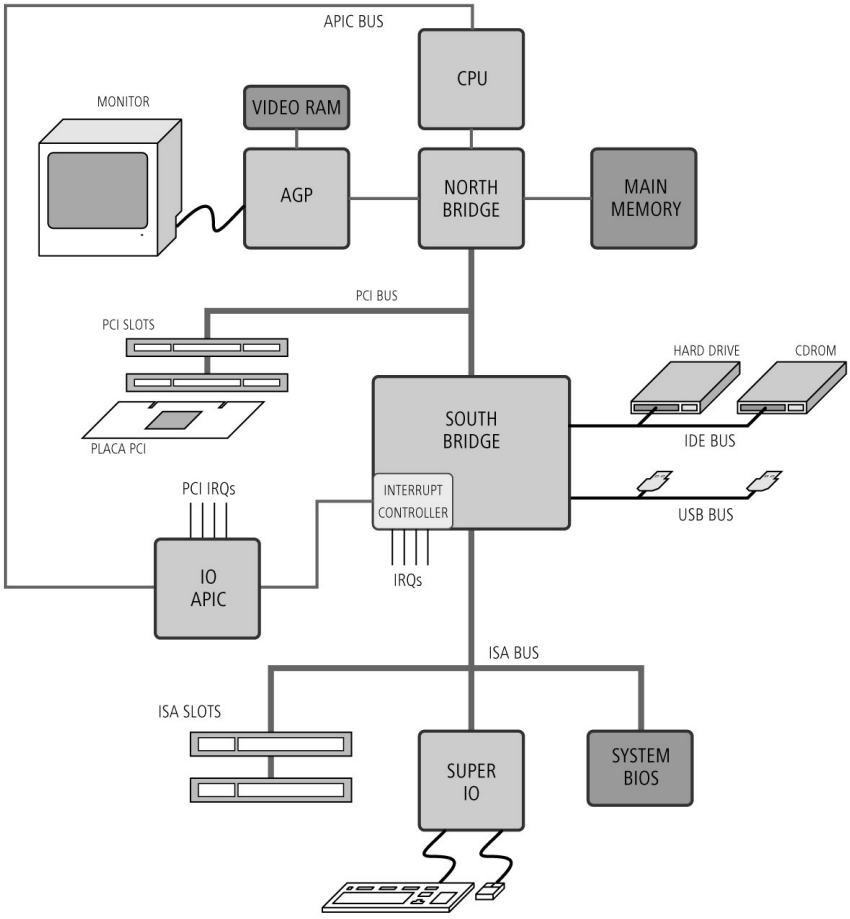
El estándar PCI fue definido por la empresa Intel para asegurarse de que el mercado no se vería inundado con varias arquitecturas de bus local peculiares y específicas para un cierto procesador. La primera revisión del estándar data de 1992. La última revisión (versión 2.2) se hizo disponible en 1999.

La sigla PCI significa *Peripherial Component Interconnect* (Interconexión de componentes periféricos). El bus PCI puede ser poblado con dispositivos, o componentes, que requieran acceso rápido entre ellos y a la memoria del sistema, y puedan ser accedidos por el procesador a velocidades que se aproximen a la de su bus local.

Resulta importante mencionar que todas las lecturas y escrituras sobre el bus PCI pueden ser realizadas como transferencias en ráfaga (burst). Al dispositivo target se le indica la dirección de comienzo y el tipo de transacción a realizar al comienzo de esta, luego se mandan datos uno tras otro, pudiendo llegarse a transferir hasta un dato por ciclo de reloj. La duración de la ráfaga la determina el master indicando si la transferencia actual es la última o no.

La siguiente figura muestra la arquitectura básica de un sistema basado en el bus PCI.

EL SISTEMA PCI



Las principales características del bus PCI son las siguientes:

- independencia de la arquitectura del procesador
- soporta hasta cerca de 80 funciones en el bus; 10 dispositivos por bus y 8 funciones por dispositivo (una función puede ser considerada como un dispositivo lógico)
- soporta hasta 256 buses
- bajo consumo
- bursts realizables en todas las lecturas y escrituras (le permite alcanzar 132 MBytes/seg con un bus de 32 bits).
- velocidades de bus de 33MHz y 66MHz
- bus de 32 o 64 bits
- tiempo de acceso bajo (60ns a 33Mhz)
- soporte de bus master, que permite transferencias entre dispositivos o entre dispositivos y memoria.

- el arbitraje del bus puede ser realizado mientras otro master esta realizando una transferencia
- baja cantidad de pines necesarios (mínimo 47 pines para un target funcional).
- chequeo de integridad de transferencias utilizando paridad de direcciones, comando y datos
- espacios de memoria, de entrada/salida y de configuración separados
- configuración automática utilizando registros de configuración específicos
- transparencia al momento de escribir software, ya que los drivers utilizan el mismo tipo de acceso a todos los dispositivos PCI.

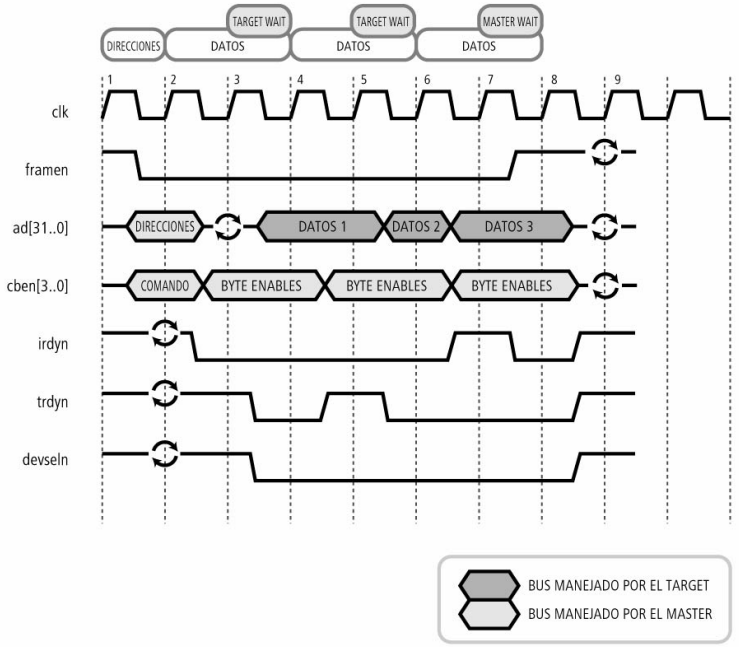
7.5.1.2. Ciclos PCI

Existe dos participantes en cada transacción sobre el bus PCI: el master y el target.
El master tiene la capacidad de iniciar una transacción. El target es accedido por el master para realizar la transacción.
Durante la documentación se usará en forma indistinta el termino ciclo o transacción PCI.

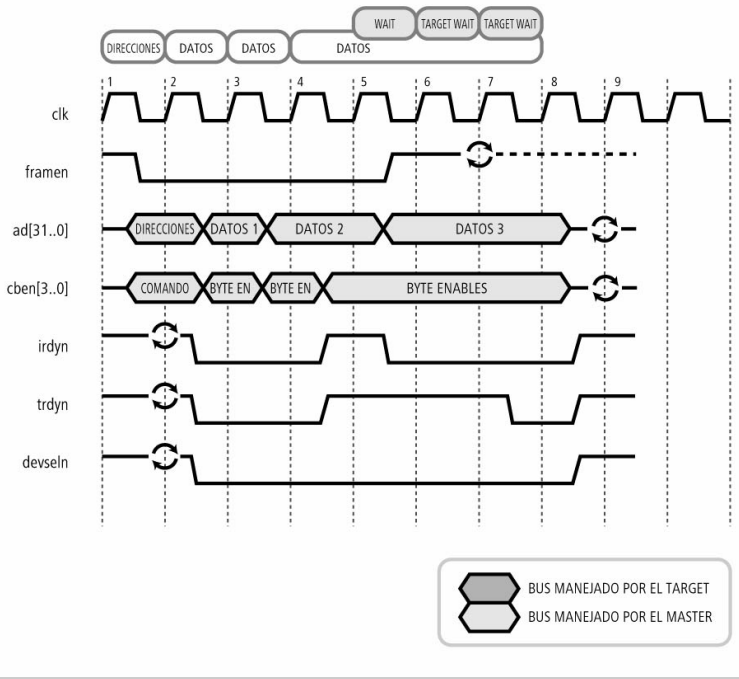
El bus PCI es un bus compartido, por lo que el master debe solicitar su uso a un arbitro antes de iniciar un ciclo. Una vez que se le concede el uso podemos identificar las siguientes fases en un un ciclo:

- fase de direcciones
- solicitud de transacción
- fase o fases de datos
- fin de la transacción.

CICLO DE LECTURA SOBRE EL BUS PCI



CICLO DE ESCRITURA SOBRE EL BUS PCI



Fase de direcciones

Cada ciclo PCI se inicia con una fase de direcciones de duración de un período de reloj. Durante esta fase, el dispositivo master activa la señal *framen* para indicar el comienzo de un ciclo, coloca un comando en el bus *cben* (Command / Byte Enable) que identifica el tipo y coloca la dirección a la que se realiza la transacción.
Es responsabilidad de los dispositivos target registrar la dirección y el comando, ya que esta estará presente únicamente durante ese período de reloj.

Solicitud de transacción

Lo dispositivo target deben decodificar la dirección y el comando registrados y determinar si se encuentra dentro del rango de direcciones que le fueron asignadas y si es un comando soportado.
Cuando un dispositivo target determina que él está siendo accedido en una transacción, la solicita activando la señal *devseln*. Si ningún dispositivo la solicita, el master aborta la transacción luego de un cierto tiempo predeterminado.

Es importante notar que el master sólo indica la dirección de comienzo, y lo hace únicamente en la fase de direcciones. Al terminar la fase de direcciones, el bus se convierte en el bus de datos por toda la duración del ciclo. Es responsabilidad del dispositivo target el registrar e incrementar la dirección para apuntar al siguiente grupo de ubicaciones accedidas en las subsiguientes transferencias de datos.

Fase o fases de datos

La fase de datos de una transacción es el período durante el cual un grupo de datos es transferido entre el master y el target.
Cada fase de datos dura al menos un período de reloj PCI. Tanto el dispositivo master como el target pueden regular la velocidad de transferencia de datos, indicando si están listos para completar una fase de datos, o dicha fase debe ser extendida por más períodos de reloj. El bus pci cuenta con un par de señales (*trdyn* y *irdyn*) para este propósito.
El ancho del bus de datos es de 4 bytes, el dispositivo master puede indicar cuales de estos bytes son válidos en cada fase activando las señales del bus *cben(3..0)*. Esto permite realizar transferencias de menos de 4 bytes.

Duración del ciclo

El master no le indica al target el número de bytes a transferir. En cambio, en cada fase de datos le indica si ese es el último dato a transferir. La señal *framen* es activada al comienzo de la fase de direcciones y se mantiene hasta que el master esta listo para

completar la última fase de datos. Cuando el target detecta la señal *irdyn* activa y la señal *framen* desactiva en una fase de datos, sabe que esta es la última de la transacción. El master mantiene las señales en este estado hasta que el target active la señal *trdyn*, indicando que se llevo a cabo la última transferencia.

Fin de la transacción y retorno a estado inactivo

Cuando la última transferencia de datos se realizó con éxito, el master devuelve el bus pci a su estado inactivo desactivando la señal *irdyn*.
Si la posesión del bus ha sido otorgada por el árbitro a otro dispositivo maestro, este detecta que el bus ha retornado al estado inactivo al desactivarse las señales *framen* e *irdyn*.

Transferencias en ráfagas

Una transferencia en ráfaga es aquella constituida de una fase de dirección única, seguida de dos o más fases de datos. El bus master debe solicitar el control del bus únicamente antes de empezar la transacción.
La dirección de comienzo y el tipo de transacción a realizar son colocados en el bus durante la fase de dirección. Todos los dispositivos del bus deben registrar la dirección y el tipo de transacción y decodificarlos para determinar cual de ellos es el dispositivo target. El dispositivo target registra la dirección de comienzo en un contador de direcciones y es responsable de incrementar la dirección en cada fase de datos.
Un dispositivo puede ser diseñado para no soportar transferencias en ráfagas.

Las transferencias en ráfagas están compuestas de la siguiente forma:

- fase de direcciones única, en la que se transfieren la dirección y el tipo de transacción.
- múltiples fases de datos, en las que se transfieren 32 o 64 bits cada a vez.

Suponiendo que ni el master ni el target insertan tiempos de espera en cada fase de datos, la transferencia máxima es de 132 MBytes/seg para buses de 33 MHz y 32 bits. Para uno de 64 bits y 66 MHz, la transferencia máxima posible es de 528 MBytes.

7.5.1.3. Registros BAR

El PC debe configurarse automáticamente de forma que cada espacio de memoria y entrada/salida de los dispositivos PCI utilice un rango de direcciones distinto. Para poder realizar esta tarea, el sistema debe ser capaz de detectar cuanta espacio necesita el dispositivo para cada rango de memoria o entrada/salida. Para esta tarea se utilizan los BAR.

Los BAR (base address register o registros de dirección base), que forman parte del espacio de configuración, son utilizados para implementar los decodificadores del dispositivo. Cada registro ocupa 32 bits y hay 6 disponibles.

7.5.1.4. Reloj PCI

Todas las acciones realizadas sobre el bus PCI están sincronizadas con la señal de reloj *clk*. La frecuencia de la señal *clk* puede tomar cualquier valor entre 0 y 33MHz. Esto permite disminuir el consumo de energía cuando se desee, y también hacer debugging paso a paso. El core PCI puede funcionar en todo el rango de frecuencias requerido.

7.5.2. Comandos del bus PCI

Durante la fase de direcciones de un ciclo, el bus *cben[3..0]* es utilizado para indicar el tipo de transacción a realizar. La siguiente tabla muestra un resumen de los tipos de transacciones, y cuales ciclos son soportados. En caso de que se le solicite al core un tipo de transacción no soportado, este la ignorará.

cben[3..0]	Tipo de ciclo	Acción
0000	Atención de interrupción	Ignorado
0001	Ciclo especial	Ignorado
0010	Lectura E/S	Aceptado
0011	Escritura E/S	Aceptado
0100	Reserved	Ignorado
0101	Reserved	Ignorado
0110	Lectura de memoria	Aceptado
0111	Escritura de memoria	Aceptado
1000	Reserved	Ignorado
1001	Reserved	Ignorado
1010	Lectura de configuración	Aceptado
1011	Escritura de configuración	Aceptado
1100	Lectura múltiple de memoria	Ignorado
1101	Ciclo de dirección doble	Ignorado
1110	Lectura de linea de memoria	Ignorado
1111	Lectura e invalidación	Ignorado

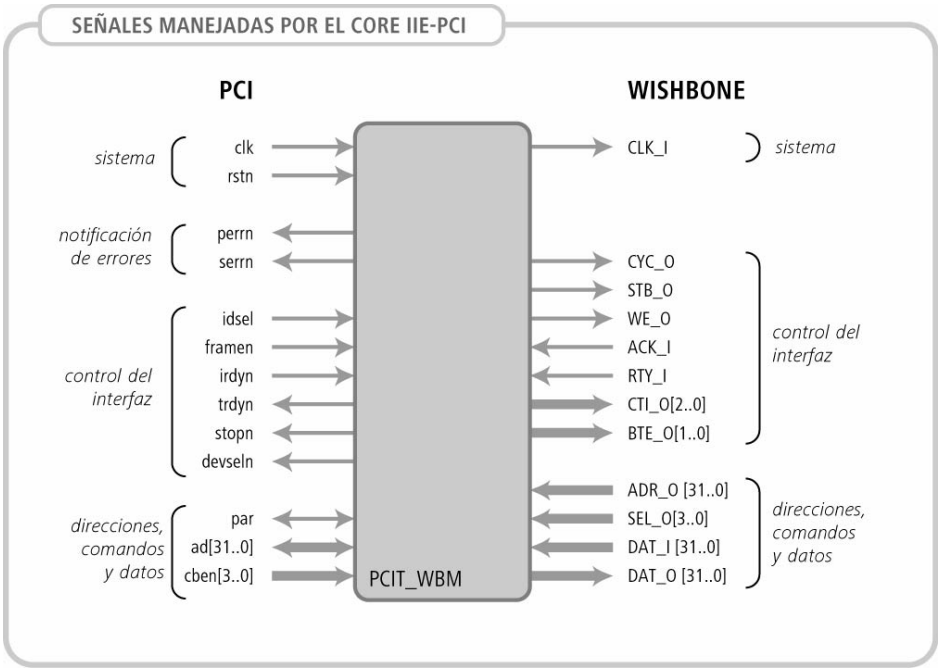
7.5.3. Señales del bus PCI

Los señales del bus PCI pueden ser clasificadas según su tipo:

- **Entrada**

- **Salida**
- **Bidireccional**
- **STS** (tri-state sostenido): señales manejadas por un dispositivo a la vez. El dispositivo, antes de soltar la señal, debe mantenerla alta por un período de reloj. Otro dispositivo que desee manejar la señal debe esperar al menos un ciclo de reloj luego de que haya sido liberada por el dispositivo anterior.
- **Colector abierto**: señales que funcionan como un OR cableado entre los dispositivos que la manejan. Poseen un pull-up débil que las mantiene altas cuando no son manejadas por los dispositivos.

El estándar PCI define algunas señales como obligatorias y otras opcionales, a continuación se enumeran las señales del bus PCI son utilizadas por el core PCI.
Las señales activas en nivel bajo están terminadas con la letra *n*.



clk
Tipo: Entrada
Descripción: La entrada clk es el reloj del interfaz PCI. Excepto rstn (reset), todas las señales son síncronas, sus niveles son válidos solo durante el flanco de subida de reloj.

rstn
Tipo: Entrada
Nivel Activo: Bajo

Descripción: Reset. Es la señal de reset para el interfaz PCI y es asíncrona respecto al reloj. Cuando está activa, las señales de salida del bus PCI deben estar en tercer estado y las señales de colector abierto deben estar flotantes.

ad[31..0]
Tipo: Tri-estado
Descripción: Bus de direcciones/datos multiplexado en tiempo. Cada transacción consiste de una fase de dirección seguida de una o más fases de datos. Una transferencia de datos es llevada a cabo cuando *irdyn* y *trdyn* están ambas activas.

cben[3..0]
Tipo: Tri-estado
Nivel Activo: Bajo
Descripción: Command/byte enable. Este bus está multiplexado en tiempo. Durante la fase de direcciones este bus indica el comando PCI deseado definiendo el tipo de transacción a realizar; durante la fase de datos, este bus indica que byte o bytes en el bus *ad* son válidos.

par
Tipo: Tri-estado
Descripción: Paridad. Es el resultado de calcular la paridad de los bits bus *ad* y del bus *cben*. La paridad de los datos transferidos en una fase de datos es presentada en el flanco de reloj siguiente.

idsel
Tipo: Entrada
Nivel Activo: Alto
Descripción: La señal *idsel* le indica al dispositivo cuando se está realizado un ciclo de acceso a sus registros de configuración.

framen
Tipo: STS
Nivel Activo: Bajo
Descripción: Frame. La señal *framen* es manejada por el dispositivo master del bus en dicho instante, e indica el comienzo y la duración de una operación en el bus. Cuando *framen* está activa, la dirección y el comando están presentes en los buses *ad* y *cben*. La señal *framen* es mantenida activa durante la transferencia de datos y se desactiva para indicar el fin de un ciclo.

irdyn
Tipo: STS
Nivel Activo: Bajo
Descripción: Initiator ready. La señal *irdyn* es manejada por el dispositivo master del bus en dicho instante, e indica que éste puede completar la transacción de datos que se está realizando. En una transacción de escritura, *irdyn* indica que el bus *ad* tiene datos válidos. En una transacción de lectura, *irdny* indica que el dispositivo maestro está listo para aceptar los datos presentes en el bus *ad*.

devseln
Tipo: STS
Nivel Activo: Bajo
Descripción: Device select. El dispositivo target activa *devseln* cuando ha decodificado su dirección y solicita la transacción.

trdyn
Tipo: STS
Nivel Activo: Bajo
Descripción: Target Ready. El dispositivo target activa *trdyn* para indicar que puede completar la transferencia de datos que se está realizando. En una operación de lectura, *trdyn* indica que el target está colocando datos válidos en el bus *ad*. En una operación de escritura, *trdyn* indica que el dispositivo target está listo para aceptar datos.

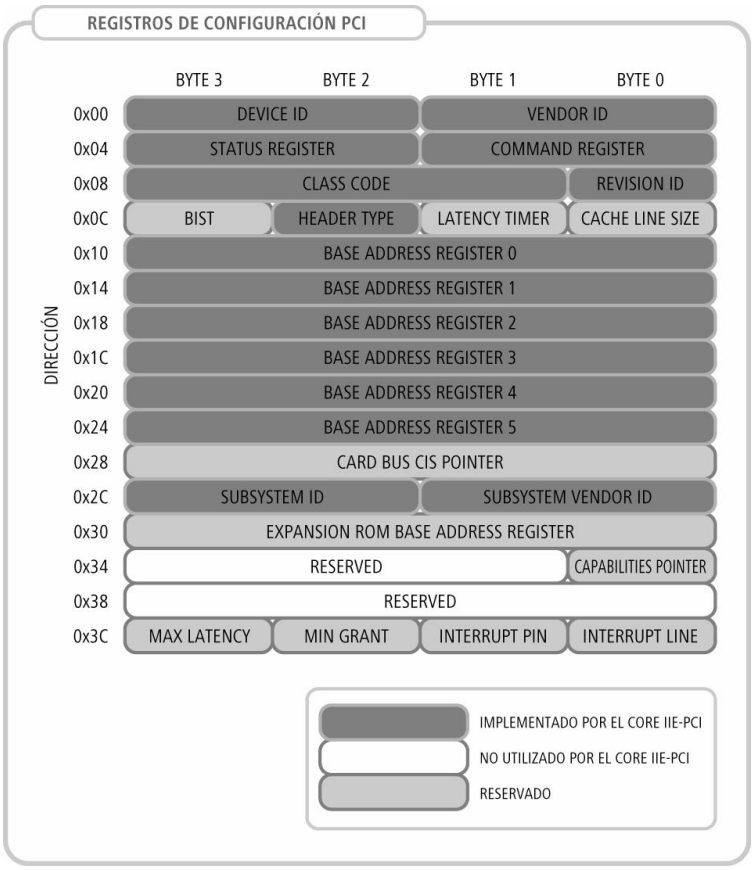
stopn
Tipo: STS
Nivel Activo: Bajo
Descripción: Stop. El dispositivo target activa *stopn* para indicar al dispositivo master que debe terminar la transacción en curso. La señal *stopn* se usa en conjunto con *trdyn* y *devseln* para indicar el tipo de terminación de transacción iniciada por el target.

perrn
Tipo: STS
Nivel Activo: Bajo
Descripción: Parity Error. La señal *perrn* indica que hubo un error de paridad en los datos. La señal *perrn* es activada un ciclo de reloj después de la señal *par*, o lo que es lo mismo dos ciclos de reloj luego de haber ocurrido un error de paridad en el bus.

serrn
Tipo: Colector abierto
Nivel Activo: Bajo
Descripción: System Error. La señal *serrn* indica un error del sistema y error de paridad en la dirección. Los dispositivos pci deben activar la señal *serrn* si detectan un error de paridad durante una fase de transferencia de direcciones.

7.5.4. Registros de configuración PCI

Cada dispositivo lógico del bus PCI debe incluir un bloque de 64 palabras dobles (un double word contiene 4 bytes o 32 bits) para almacenar los valores de configuración. El formato de las 16 primeras palabras dobles está definido en la especificación PCI y se muestra a continuación:



Los registros ocupados por las primeras 16 palabras dobles son utilizados para identificar el dispositivo, controlar las funciones del bus PCI, y proveer de información de estado.

La siguiente tabla resume las configuraciones soportadas en los registros de configuración, implementados por el core PCI:

Offset (hexa)	Rango	Tamaño (bytes)	Tipo	Nombre Interno	Nombre
00	00-01	2	L	vendor_id	Vendor ID
02	02-03	2	L	device_id	Device ID
04	04-05	2	L/E P	command	Command
06	06-07	2	L/E P	status	Status
08	08-08	1	L	rev_id	Revision ID
09	09-0B	3	L	classcode	Class code
0E	0E-0E	1	L	header_type	Class code
10	10-13	4	L/E	bar0	Base address register cero
14	14-17	4	L/E	bar1	Base address register uno
18	18-1B	4	L/E	bar2	Base address register dos
1C	1C-1F	4	L/E	bar3	Base address register tres
20	20-23	4	L/E	bar4	Base address register cuatro
24	24-27	4	L/E	bar5	Base address register cinco
2C	2C-2D	2	L	subsystem_vid	Subsystem vendor ID
2E	2E-2F	2	L	subsystem_id	Subsystem ID
3C	3C-3C	1	L/E	int_line	Interrupt line
3D	3D-3D	1	L	int_pin	Interrupt pin

Referencia:

Tipo:

- L : Lectura. En funcionamiento, solo puede ser leído desde el bus PCI. Para cambiar su valor hay que volver a sintetizar el core PCI modificando los parámetros asignados al instanciar el core PCI.
- L/E : Lectura y Escritura. EL registro puede ser leído y escrito en funcionamiento normal, a través del bus PCI.

- L/E P : Lectura y Escritura parcial. Todos los bits del registro son legibles, pero sólo algunos pueden ser modificados desde el bus PCI. Todos pueden ser modificados en el momento de instanciar el core PCI.

7.5.4.1. Registros requeridos por la especificación PCI

Registro Vendor ID

Es un registro de 16 bits que identifica el fabricante del dispositivo. Este valor se configura al sintetizar y su acceso es solo de lectura. El valor de este registro es asignado por una autoridad central (PCI SIG). El valor utilizado en el core PCI es el correspondiente a Altera (0x1172), es un parámetro del core.

Registro Device ID

Es un registro de 16 bits, asignado por el fabricante para determinar el tipo o modelo de dispositivo. El valor utilizado en el core PCI es 0xABBA, es un parámetro del core.

Registros Subsystem Vendor ID y Subsytem ID.

Este par de registros de 16 bit permite diferenciar entre dos tarjetas de usos distintos que han sido diseñadas alrededor de una misma lógica PCI. El valor de Subsystem Vendor ID es asignado por el PCI SIG, mientras que el Subsystem ID es asignado por el fabricante de la placa.

Registro Revision ID

Es un registro de 8 bits, de lectura, que identifica el número de revisión del dispositivo. Su valor es asignado por el fabricante. Puede se cambiado al sintetizar.

Registro Class Code

La función de este registro de 24 bits es Identificar la función básica del dispositivo (controlador de red, dispositivo de almacenamiento, etc.). Está divido en tres partes, el byte superior indica la clase, el medio la subclase y el inferior el interfaz de programación. El valor utilizado en el core PCI (0x0B4000) indica que es de clase "Processors" y sub-clase "Co-processor". Es de esa forma que es desplegado por los chequeos realizados por el BIOS al encender el PC y por las utilidades que dan información sobre el estado del bus PCI (*lspci* en Linux).

Registro Command

Es un registro de 16 bits, de escritura y lectura, que provee control básico sobre las posibilidades del dispositivo para responder o realizar accesos al bus PCI. Los bits 0 a 9 tiene asignadas funciones en la especificación PCI 2.2 10.1.I . El resto de los bits están reservados para uso futuro.

Solo los siguientes bits pueden ser modificados:

Bit	Nombre	Función
0	espacio de E/S	habilita los accesos a E/S
1	espacio de memoria	habilita los accesos a memoria
6	perrn enable	habilita el reporte de errores de paridad mediante la señal *perrn*
8	serrn enable	habilita el reporte de errores de paridad mediante la señal *serrn*

Los restantes bits corresponden a funcionalidades de PCI Master y ciclos no soportados.

Registro Status

Este registro de 16 bits mantiene el estado del dispositivo PCI.
El registro puede ser leído normalmente, pero la escritura se maneja de forma diferente. En una escritura, el bit que está activo se desactiva mediante una escritura de un 1. No es posible activar un bit por software.
Este método fue elegido para simplificar el trabajo del programador. Luego de leer el Status y hacerse cargo de los errores correspondientes, el programador desactiva los bits escribiendo el valor que había sido leído en primera instancia. De esta forma, si algún otro bit cambió durante el proceso, este es preservado.

El significado de los bits del registro de estado implementados por el core PCI es el siguiente:

Bit	Nombre	Condición de activación
11	target abort señalado	dispositivo abortó una transacción
14	system error señalado	dispositivo activó la señal *serrn*
15	error de paridad	error de paridad detectado

Registro Header Type

El bit más alto de este registro indica si el dispositivo es multifunción. El resto de los bits indican el tipo de cabezal de configuración que se está especificando. Todas las descripciones realizadas hasta ahora corresponden al cabezal de tipo 0, y es el usado por la mayoría de los dispositivos PCI. Existen otros tipos definidos para puentes PCI-PCI, etc.

7.5.4.2. Registros opcionales implementados

Registro Interrupt Pin

Este registro es requerido si el dispositivo es capaz de de generar pedidos de interrupción.
Determina cual de los cuatro pines de pedido de interrupción PCI está conectado al dispositivo. En el estado actual del core PCI no se utilizan interrupciones, por lo tanto su valor está asignado a 0.

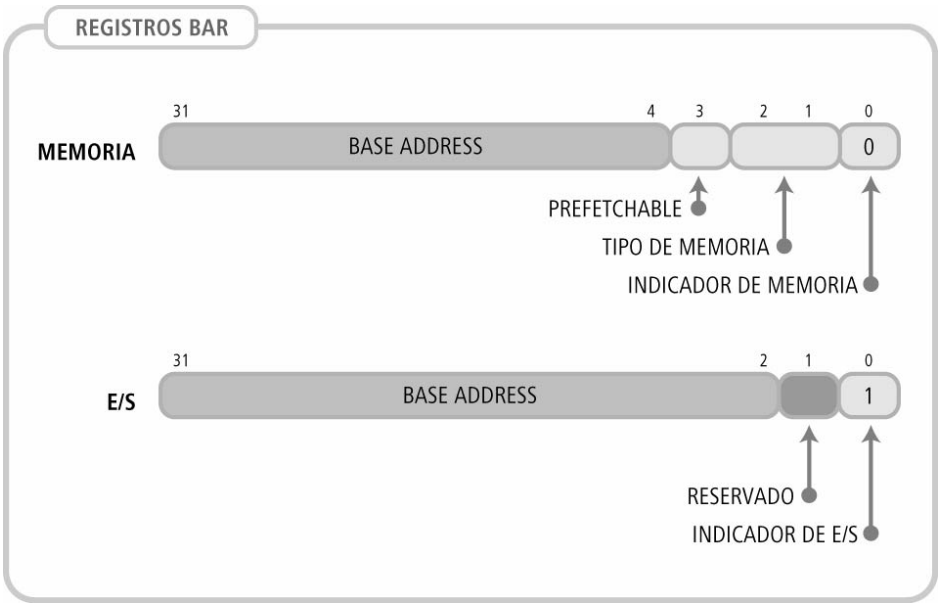
Registro Interrupt Line

Este registro de lectura y escritura es requerido si el dispositivo es capaz de de generar pedidos de interrupción. Se utiliza para identificar a que entrada del controlador de interrupciones está conectado el pin indicado por el registro Interrupt Pin.

Registros BAR (Base Address Register)

Al menos un registro de este tipo es requerido si el dispositivo memoria y entrada/salida. Estos registros son utilizados para determinar los requerimientos de tamaño y cantidad de rangos de direcciones de memoria y entrada/salida deben ser asignados al dispostivo.

El bit más bajo del registro es solo de lectura y determina si el rango de direcciones corresponde a memoria (0) o entrada/salida (1). El valor de los restantes bits del registro depende del bit 0 y se muestra en el siguiente diagrama:



En el momento de inicialización, el sistema escribe unos en cada registro y luego los lee para obtener la información de cantidad y tamaño solicitados por el dispositivo. Si el valor leído es 0, esto significa que el BAR no está implementado. Si el valor no es 0, la posición del primer 1 contando a partir del bit menos significativo del campo Base Address (ver diagrama anterior) determina el tamaño de memoria o entrada/salida. Luego, el sistema escribe en el BAR la dirección de comienzo del rango asignado.

Ejemplo:
El sistema escribe 0xFFFFFFFF en el BAR0 y lee el valor 0xFFF00000. Como el valor es distinto de 0, significa que el BAR está implementado. La información obtenida es la siguiente:

- bit 0 = 0 -> memoria
- bit [2:1] = 00 -> memoria ubicada debajo de los 4GB de direcciones
- bit 3 = 0 -> memoria "prefetchable" (puede ser leída de antemano para aumentar performance).
- bit 20 es el primer 1 encontrado en el campo Base Address. El peso de este bit indica que corresponde a una memoria de 1MB.

7.6. Interfaz Wishbone

Es un requerimiento para el uso del core PCITWBM tener un conocimiento básico del interfaz Wishbone, pero no es necesario conocer los detalles de funcionamiento del interfaz PCI.

7.6.1. Introducción al interfaz Wishbone

La interfaz Wishbone es una especificación que define un método de interconexión de diseños digitales dentro de un circuito integrado.

La arquitectura Wishbone resuelve un problema básico en el diseño de circuitos integrados, que es cómo conectar circuitos entre sí de una manera simple, flexible y transportable. Los circuitos a los que nos referimos proveen alguna funcionalidad específica, y generalmente son distribuidos como cores IP (Intellectual Property Cores, o núcleos de propiedad intelectual), que pueden ser adquiridos o desarrollados.
Los IP cores son, entonces, los bloques funcionales que forman parte del sistema. Generalmente son desarrollados independientemente, y el hacerlos trabajar juntos resulta problemático. El interfaz Wishbone estandariza los interfaces utilizados por IP cores, lo que simplifica su interconexión para la creación de sistemas a medida sobre un chip (SoC).

La especificación Wishbone pertenece al dominio público, puede ser libremente copiada y distribuida, y utilizada para el diseño y producción de componentes de circuitos integrados sin necesidad de pagar ningún tipo de licencia o royalties.

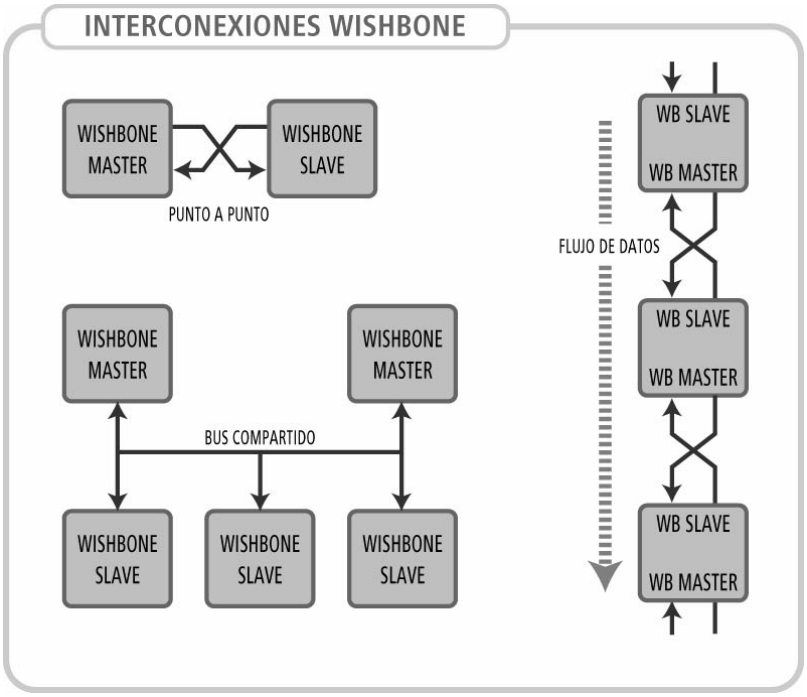
En el apéndice "Interfaz Wishbone" se hace una descripción más detallada del funcionamiento del bus Wishbone.

7.6.1.1. Interconexión maestro/esclavo

Wishbone utiliza una arquitectura maestro/esclavo, lo que significa que aquellos módulos con interfaz maestro inician las transacciones de datos realizadas con los módulos esclavos.

El esquema de interconexión de módulos puede adaptarse de acuerdo a la función que estos cumplen y el uso final que se les de. Puede tenerse una conexión única entre 2 módulos, conexiones tipo bus donde todos los master pueden iniciar transacciones con todos los esclavos, encadenar varios módulos maestro/esclavo, etc.

Algunos ejemplos de conexiones:



Wishbone permite que el integrador del sistema ajuste esta interconexión según las necesidades de intercomunicación entre los módulos que desea interconectar. Esto es posible ya que la especificación esta pensada para ser utilizada dentro de circuitos integrados donde la lógica reconfigurable y los chips de circuitos integrados tienen caminos de interconexión que pueden ser ajustados o diseñados según las necesidades.

En el apéndice "Interfaz Wishbone" se puede encontrar una descripción más detallada.

7.6.2. Señales del interfaz Wishbone

A continuación se describen las señales del interfaz Wishbone utilizadas por el core PCI. Todas las señales son activas por nivel alto y la terminación *_I* y *_O* indica si son entradas o salidas al core respectivamente.

CLK_I
Tipo: Entrada
Descripción: Reloj. La señal de reloj coordina todas las actividades para la lógica dentro de un dispositivo Wishbone. Todas las señales de salida Wishbone son

registradas en el flanco de subida de *CLK_I*. Todas las entradas Wishbone deben estar estables antes del flanco de subida de *CLK_I*.

DAT_I[31..0]
Tipo: Entrada
Descripción: El bus *DAT_I* es la entrada de datos. Su ancho de palabra es de 32 bits.

DAT_O[31..0]
Tipo: Salida
Descripción: El bus *DAT_O* es la salida datos. Su ancho de palabra es de 32 bits.

ACK_I
Tipo: Entrada
Descripción: Acknowledge Input. La señal *ACK_I* le indica al dispositivo master que se ha realizado una transferencia en forma exitosa.

ADR_O[31..0]
Tipo: Salida
Descripción: El bus *ADR_O* es manejado por el master y e indica la dirección de los datos que deben leerse o escribirse.

CYC_O
Tipo: Salida
Descripción: Cycle Output. La señal *CYC_O* indica que un ciclo válido está en progreso. La señal se mantiene activa por la duración de todo el ciclo. Por ejemplo, en una transferencia en bloque, la señal *CYC_O* se activa en la primer transferencia y se mantiene activa hasta la última.

RTY_I
Tipo: Entrada
Descripción: Retry Input. Indica que el interfaz no esta listo para aceptar o enviar datos, y que el ciclo debe ser reintentado más tarde. En este caso, el core PCI interrumpe el ciclo Wishbone y vuelve a comenzar un nuevo ciclo para terminar la transferencia de datos.

SEL_O[3..0]
Tipo: Salida
Descripción: Select Output. El bus *SEL_O* indica qué bytes del bus *DAT_O* contienen datos válidos, o en que bytes del bus *DAT_I* se esperan datos válidos. Estas señales se

corresponden con las señales del bus PCI *cben*.

STB_O

Tipo: Salida

Descripción: Strobe Output. La señal *STB_O* indica un la presencia de un dato válido en *DAT_O* en un ciclo de escritura o que esta listo para recibir datos en un ciclo de lectura. El dispositivo esclavo Wishbone debe responder con alguna de las señales *ACK_I* o *RTY_I* frente a cada activación de la señal *STB_O*.

WE_O

Tipo: Salida

Descripción: Write Enable Output. La señal *WE_O* indica si el ciclo de bus corresponde a una lectura o a una escritura. La señal se activa durante los ciclos de escritura y se desactiva en los ciclos de lectura.

CTI_O[2..0]

Tipo: Salida

Nivel Activo: Alto

Descripción: Cycle Type Identifier. La señal CTI_O provee información adicional sobre el ciclo que se está realizando. Es parte de la especificación de Wishbone avanzado. El master le envía esta información al esclavo y este debe usarla para preparar la respuesta a dar en el ciclo siguiente.

La siguiente tabla muestra los posibles tipos de ciclos:

CTI_O[2..0]	Descripción
000	Ciclo clásico
001	Ciclo burst de dirección constante
010	Ciclo burst con incremento de direcciones
111	Fin del ciclo burst

BTE_O[1..0]

Tipo: Salida

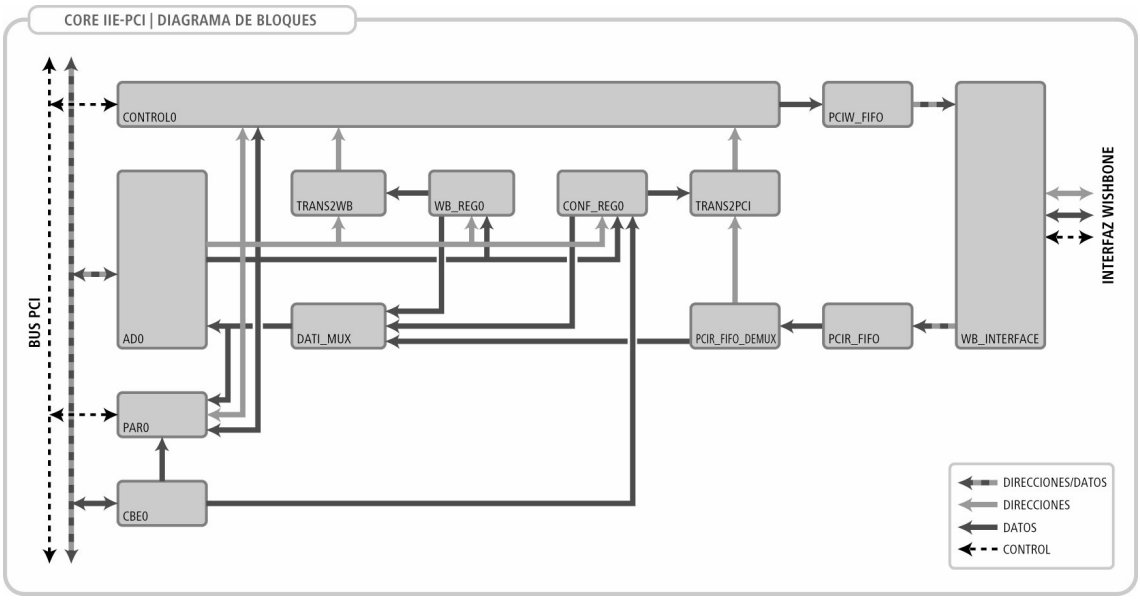
Descripción: Burst Type Extension. La señal *BTE_O* provee información adicional sobre el ciclo que se está realizando. Esta información es relevante únicamente para ciclos burst con incremento de direcciones.

En la implementación del core PCI se incluye, ya que es mandatoria en la especificación Wishbone si se desea utilizar CTI_O, pero su valor es siempre 00. Esto significa que si se utilizan ciclos burst con incremento de direcciones, su incremento es lineal.

7.7. Arquitectura y funcionamiento

7.7.1. Arquitectura

El diseño se dividió en varios bloque para facilitar el diseño y prueba.



CONTROL0

Instancia de entidad blk_control.

Bloque principal de control. Realiza el control general de todo el funcionamiento del core y también contiene multiplexores para enviar direcciones y datos hacia el interfaz Wishbone.

AD0

Instancia de entidad blk_ad.

Bloque de interfaz con las señales AD del bus PCI. Se encarga de leer y escribir las señales de direcciones y datos del bus PCI.

PAR0

Instancia de entidad blk_par.

Bloque de paridad. Realiza el cálculo de paridad, ya sea de los datos leídos, así como de los enviados por el bus PCI.

CBE0

Instancia de entidad blk_cbe.

Bloque de comando o de habilitación de bytes. Registra los comandos y las señales de

habilitación de bytes del bus PCI y las mantiene disponibles para su uso posterior.

CONF_REG0

Instancia de entidad blk_conf_reg.

Bloque de registro de configuración PCI. Almacena los registros de configuración PCI del dispositivo.

WB_REG0

Instancia de entidad blk_wb_reg.

Bloque de registro de mapeo de direcciones a espacio Wishbone.

TRANS2PCI

Instancia de entidad blk_add_trans.

Bloque de transformación a espacio PCI. Realiza el mapeo de direcciones Wishbone a direcciones PCI.

TRANS2WB

Instancia de entidad blk_add_trans.

Bloque de transformación a espacio Wishbone. Realiza el mapeo de direcciones PCI a direcciones Wishbone.

DATI_MUX

Instancia de entidad blk_dati_mux.

Bloque multiplexor de datos. Realiza la multiplexión de datos a escribir en el bus PCI. Permite leer datos de cualquiera de los registros de configuración (CONF_REG0 y WB_REG0) y los datos provenientes del bus Wishbone.

PCIW_FIFO

Instancia de entidad fifo.

Bloque de buffer FIFO para escritura hacia Wishbone. Permite que el reloj utilizado por la aplicación a conectar a través del interfaz Wishbone utilice una frecuencia de reloj diferente a la utilizada por el bus PCI.

PCIR_FIFO

Instancia de entidad fifo.

Bloque de buffer FIFO para lectura desde Wishbone. Permite que el reloj utilizado por la aplicación a conectar a través del interfaz Wishbone utilice una frecuencia de reloj diferente a la utilizada por el bus PCI.

PCIR_FIFO_DEMUX

Instancia de entidad blk_fifo_demux.

Bloque demultiplexor del FIFO de lectura desde Wishbone. Realiza la tarea de separar las direcciones de los datos leídos que se envían a través del fifo.

WB_INTERFACE

Instancia de entidad blk_wb_interface.

Bloque de interfaz con Wishbone. Implementa el protocolo de comunicación Wishbone.

7.7.2. Puente PCI-Wishbone

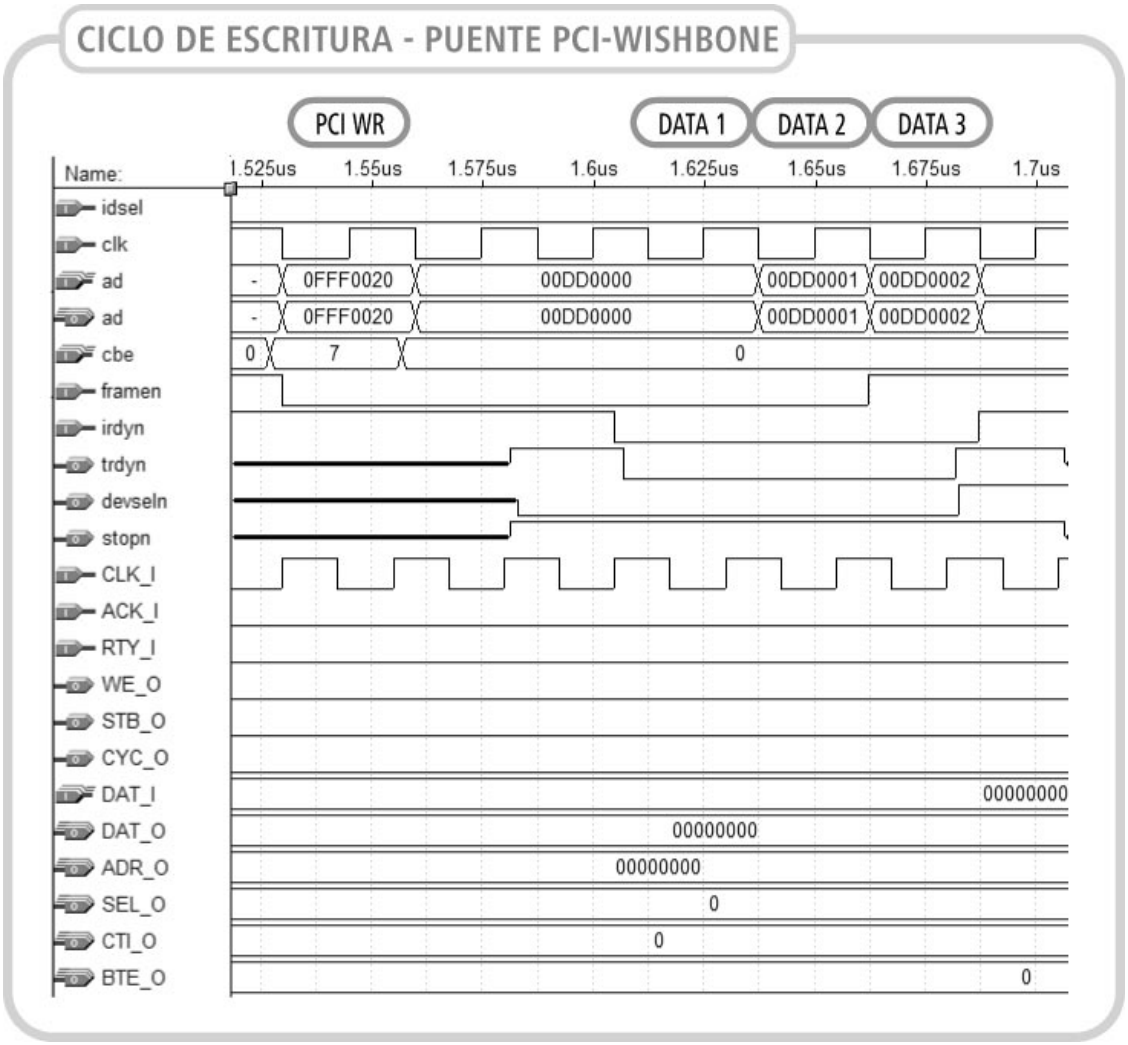
La mayoría de las interfaces PCI existentes resuelven parte de la comunicación con el bus PCI, la interfaz de aplicación tiene señales que deben ser controladas para responder a ciclos en el bus PCI. Esto hace que el diseñador de la aplicación deba conocer el protocolo PCI para implementar exitosamente un diseño.

Como solución a esto se optó por diseñar un modulo que provee una interfaz de aplicación más sencilla (Wishbone), cuyo comportamiento no dependa mayormente de lo que sucede en el bus PCI, sino solo si ocurre una escritura o una lectura desde el bus PCI.
Esto se logra con un módulo puente entre el bus PCI y la aplicación. El modulo se encarga de manejar las señales del bus PCI para recibir los datos o enviar datos y por otro lado maneja las señales de la interfaz wishbone para enviar los datos recibidos o solicitar los datos pedidos desde el bus PCI.

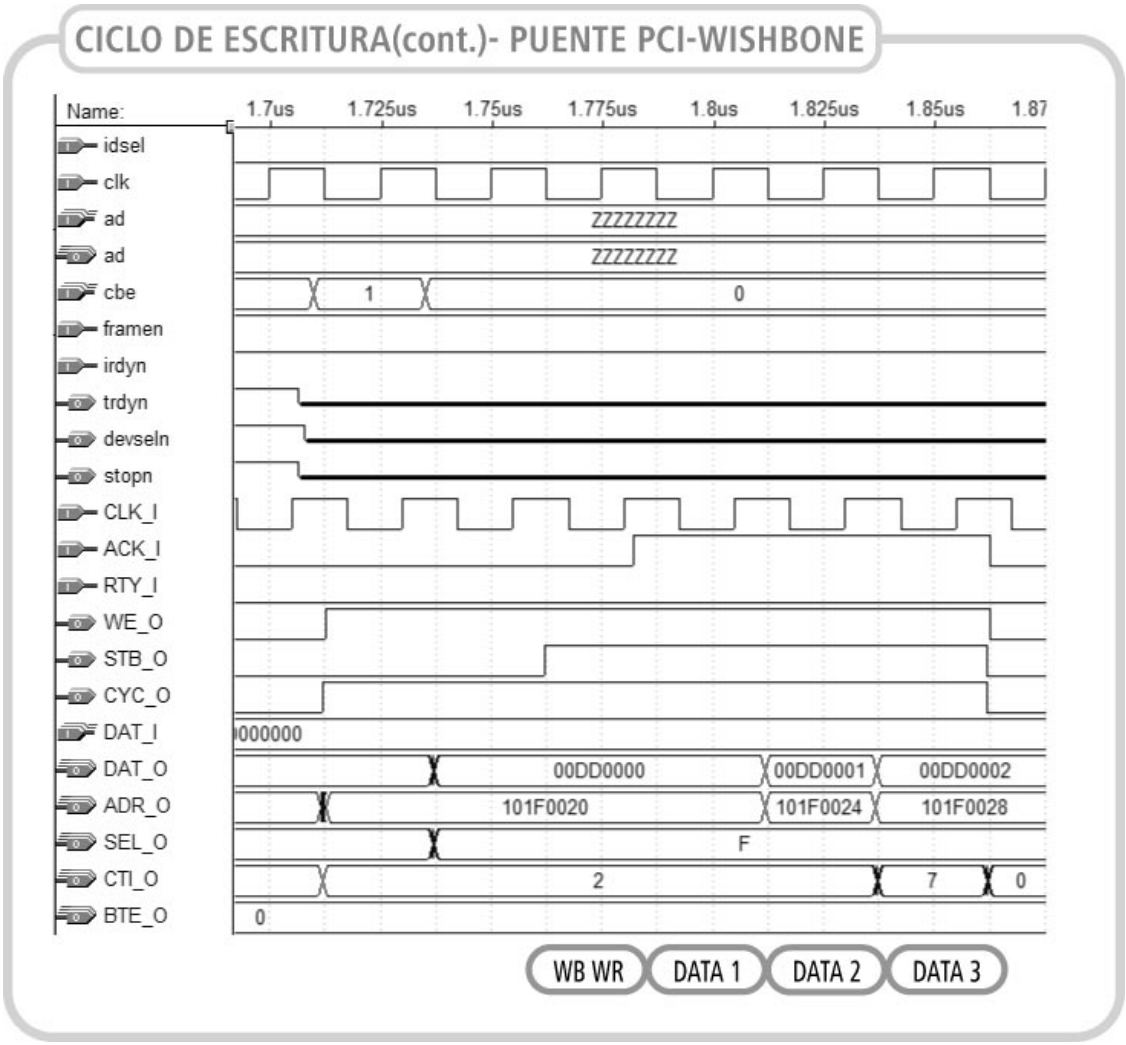
Como se decidió implementar un core PCI target, las transacciones son empezadas solo desde el bus PCI, no desde el lado Wishbone del core. Por esta razón la interfaz Wishbone es un interfaz maestro.

7.7.2.1. Funcionamiento del puente PCI-Wishbone

Cuando un master PCI realiza una escritura dentro de los rangos de direcciones asignados al core PCI, este acepta los datos y los almacena en un FIFO.

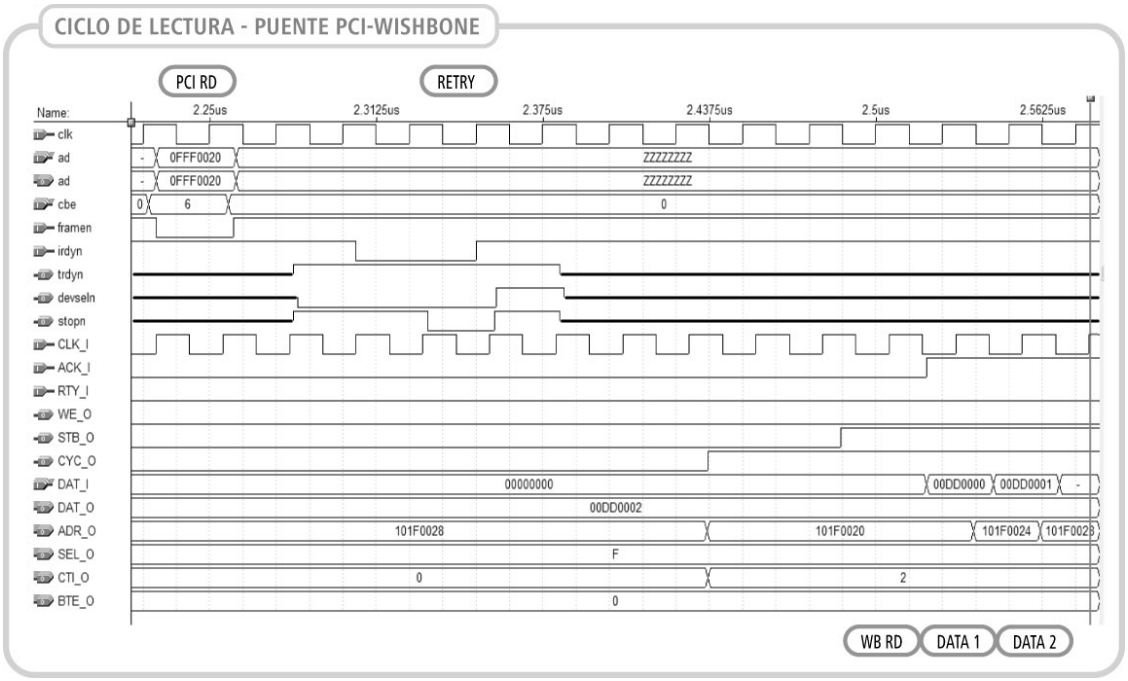


Luego al detectar que el FIFO de escritura no esta vacío, la interfaz Wishbone realiza tantos ciclos de escritura como sean necesarios hasta vaciar el FIFO.

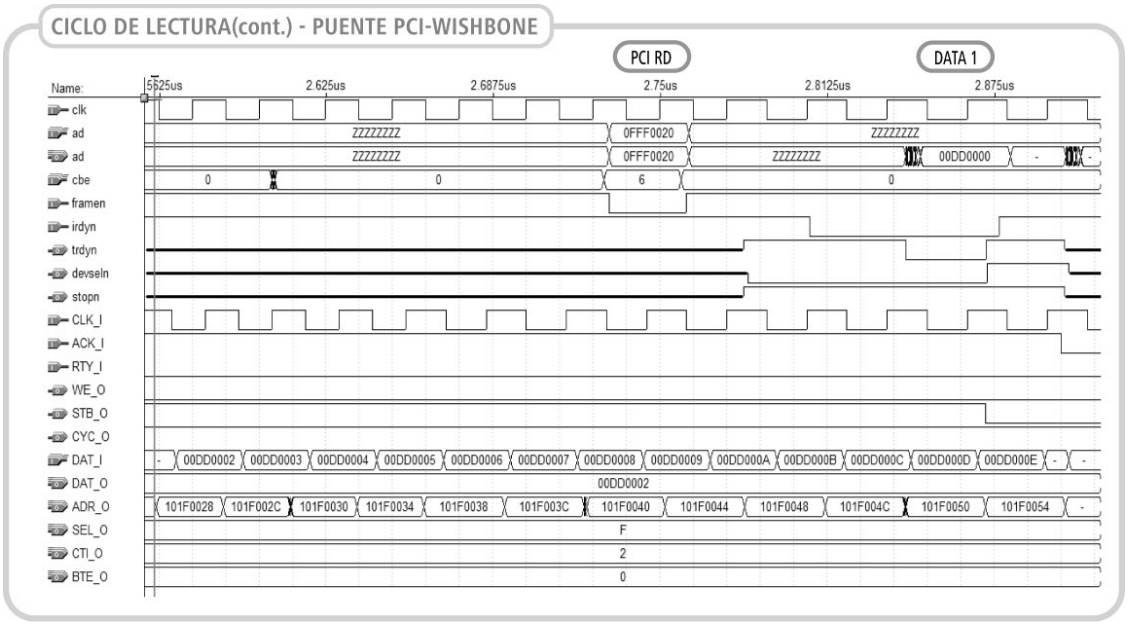


Cuando un master PCI comienza una lectura dentro de los rangos de direcciones asignados al core PCI, este registra la dirección de comienzo y responde pidiendo un *retry*, esto hace que el master PCI aborte el ciclo y vuelva a intentar nuevamente la lectura cierto tiempo después.

Mientras esto sucede la interfaz Wishbone realiza ciclos de lectura, almacenando los datos leídos en el FIFO de lectura hasta llenarlo.



Cuando el master PCI reintenta el ciclo de lectura, la interfaz PCI contesta enviando los datos almacenados en el FIFO.



7.7.2.2. Detalles de la escritura al core PCI.

En caso de querer comenzar un ciclo de escritura PCI a una dirección asignada al core PCI, si la interfaz Wishbone no ha vaciado el FIFO de escritura, el core solicita un *retry* al master PCI.
Si ya se estaba realizando un ciclo de escritura y el FIFO de escritura se llena, se insertan tiempos de wait en el bus PCI (*trdyn='1'*) para dar tiempo al master Wishbone a sacar datos del FIFO. Si transcurrido 8 períodos de reloj el FIFO aún no se ha vaciado se pide un *disconect without data* lo que provoca que el ciclo se corte. Si aún restaban datos por escribir el master PCI automáticamente comenzará otro ciclo de escritura.

Si la aplicación del lado Wishbone que recibe los datos escritos por el core esta mal implementada o no responde a las escrituras, el bus PCI podría quedar trancado pues se estarían intentando escrituras PCI continuamente y el FIFO nunca se encontraría vacío.

7.7.2.3. Detalles de la lectura al core PCI.

Cuando se realiza una lectura a una dirección asignada al core PCI, este chequea si hay datos en el FIFO de lectura y si corresponden a la dirección solicitada. En caso de coincidir las direcciones, se transfieren los datos desde el FIFO hasta que se vacíe o el ciclo sea finalizado por el master PCI.
En caso de que el FIFO de lectura se vacíe, se insertan tiempos de espera en el bus PCI (*trdyn='1'*) para dar tiempo a que el master Wishbone coloque más datos en el FIFO de lectura. Si transcurridos 8 períodos de reloj no hay nuevos datos en el FIFO de lectura se pide un *disconnect without data* lo que provoca que el ciclo se corte. Si aún restaban datos por leer el master PCI automáticamente comenzará otro ciclo de lectura.
Si lo datos almacenados en la boca del FIFO de lectura no corresponden con la dirección solicitada, se registra la nueva dirección, se vacía el fifo y se vuelve al comienzo del proceso, solicitando un *retry* al master PCI.

7.7.3. Registro de traslación de direcciones

La dirección asignada a cada BAR del core PCI es asignada automáticamente en el momento de inicio del PC, y puede variar, dependiendo del PC que se esté utilizando y de cuantos dispositivos PCI estén colocados en el bus.

Para facilitar la tarea de diseño y decodificación de direcciones dentro de la aplicación en el bus Wishbone se implementaron registros de traslación de direcciones.

La función de los mismos es lograr que un acceso PCI a una dirección dentro del rango correspondiente a un BAR determinado se refleje como un acceso siempre a la misma dirección del bus Wishbone.

Estos registros pueden leerse y escribirse a través del BAR0 del core PCI en las direcciones indicadas por la siguiente tabla:

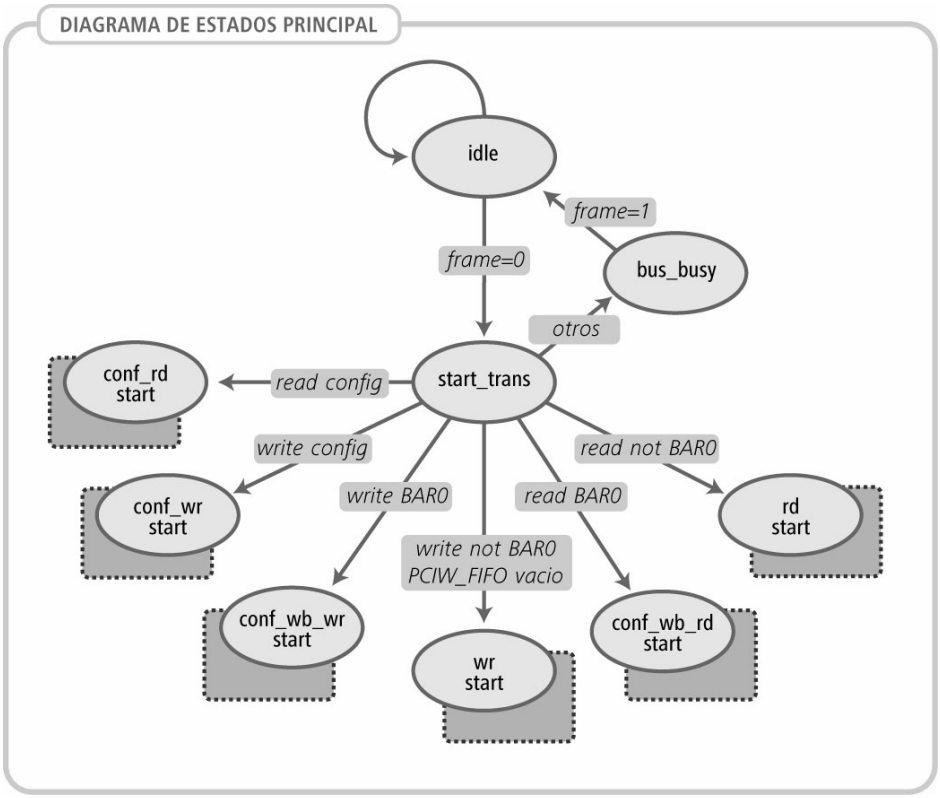
Registro de traslación	Ubicación en BAR0	Valor luego de Reset
para Bar 1	0x10	0x10000000
para Bar 2	0x14	0x20000000
para Bar 3	0x18	0x30000000
para Bar 4	0x1C	0x40000000
para Bar 5	0x20	0x50000000
para Bar 6	0x24	0x60000000

Supongamos que al momento del booteo se le asigna a BAR0 el valor 0x80000000 y a BAR1 el 0x8F000000.
Si se quiere que los accesos al rango de direcciones correspondiente a BAR1 se mapeen a direcciones Wishbone que comiencen a partir de la dirección 0xE0000000, se debe escribir ese valor en la dirección 0x80000010.
Si se realiza una escritura a la dirección PCI 0x8F001000, se estará escribiendo en la dirección Wishbone 0xE0001000.

7.7.4. Independencia de relojes PCI y Wishbone

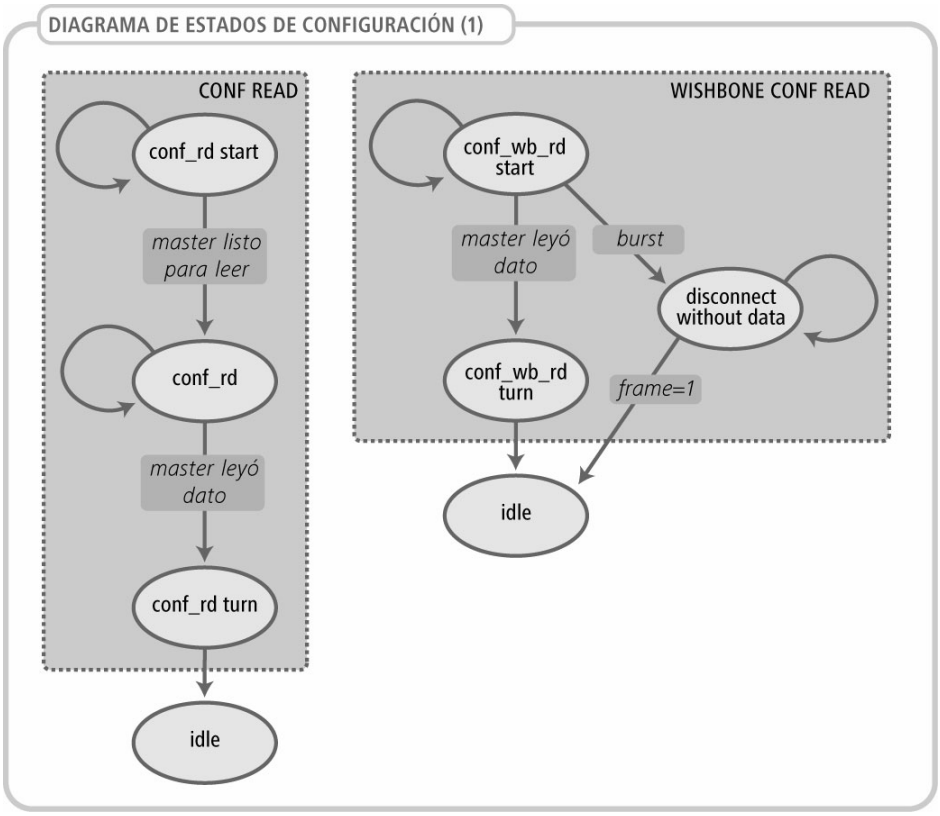
Los FIFOs entre la interfaz PCI y la interfaz Wishbone, no solo sirven como almacenamiento intermedio de los datos, sino que al contar los FIFOs con entradas de reloj independientes para la lectura y la escritura, permiten que las lógicas de las interfaces funcionen con diferentes velocidades de reloj.

7.7.5. Maquina de estado de Target PCI



El core detecta el comienzo de una transacción PCI cuando *framen* es 0 en un flanco de subida de reloj. En ese momento se latchean las direcciones y el comando que identifica el tipo de ciclo. Con esta información se pasa al estado correspondiente según se indica en el diagrama. Más abajo se muestran los ciclos de cada una de las posibilidades.

En caso de que el ciclo no sea para este dispositivo, o se intente realizar una transacción no soportada, el core pasa al estado *bus_busy* hasta que termina esa transacción.

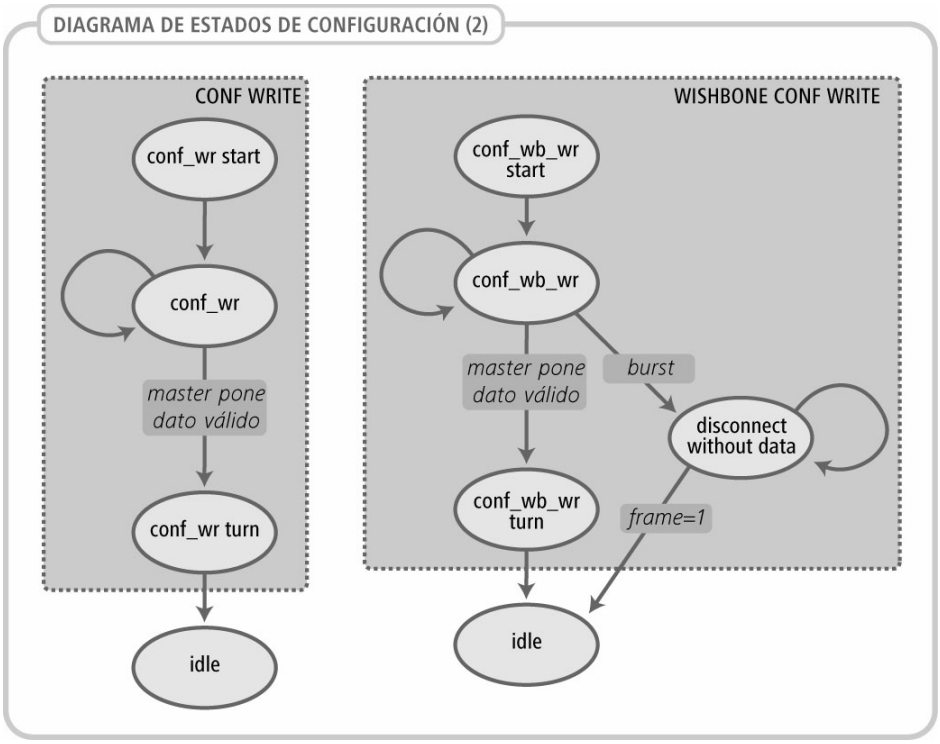


7.7.5.1. Conf Read

Al realizar un acceso a los registros de configuración se espera hasta que el master esté listo para recibir los datos, se transfieren los datos y luego se pasa a un estado de *turn arround* mediante el cual se libera el bus de acuerdo a lo especificado en el estándar.

7.7.5.2. Wishbone Conf Read

En BAR0 se encuentran mapeados los registros de traslación de direcciones PCI a Wishbone. No está permitido realizar lecturas *burst* a estos registros, por lo que en dicho caso, se realiza una desconexión sin transferencia de datos. En un acceso simple, se espera a que el master esté listo a recibir los datos, se transfieren los datos y luego se pasa a un estado de *turn arround* mediante el cual se libera el bus de acuerdo a lo especificado en el estándar.

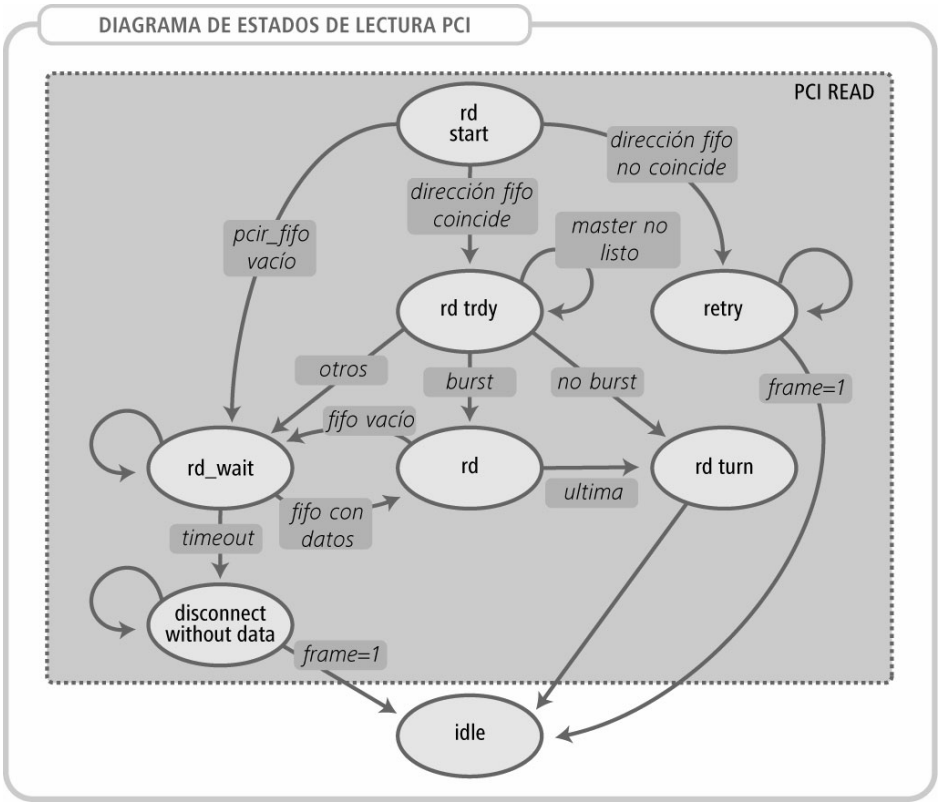


7.7.5.3. Conf Write

Al realizar una escritura a los registros de configuración, se espera a que el master ponga el dato válido en el bus, se lee y luego se pasa a un estado de *turn arround* mediante el cual se libera el bus de acuerdo a lo especificado en el estándar.

7.7.5.4. Wishbone Conf Write

De igual forma que en Wishbone Conf Read, no está permitido realizar escrituras burst a estos registros. El resto de los estados son similares a Conf Write.

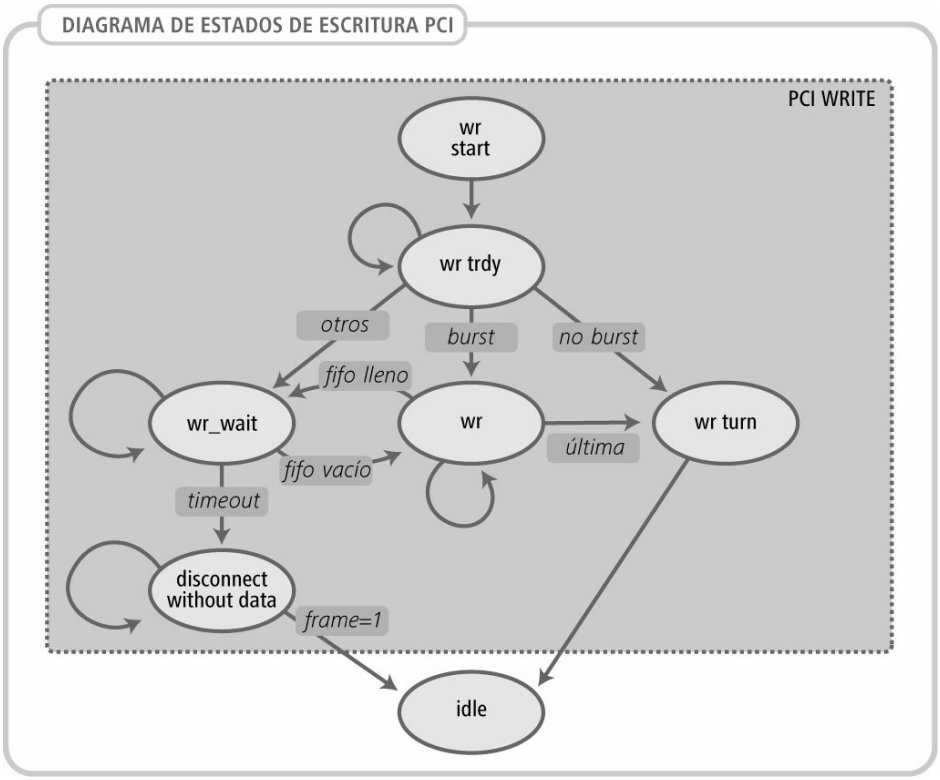


7.7.5.5. Pci Read

Si la dirección solicitada en la lectura PCI no coincide con la presente en la boca del pcir_fifo (que transfiere datos desde Wishbone a PCI), se realiza una desconexión sin transferencia de datos y el master realizará un reintento de lectura en un instante posterior.

Si la dirección solicitada en la lectura PCI coincide con la presente en la boca del pcir_fifo, se pueden empezar a realizar transferencias. Para esto se espera hasta que el master esté listo para recibir los datos y luego, se transfieren datos ya sea uno solo o en *burst*.

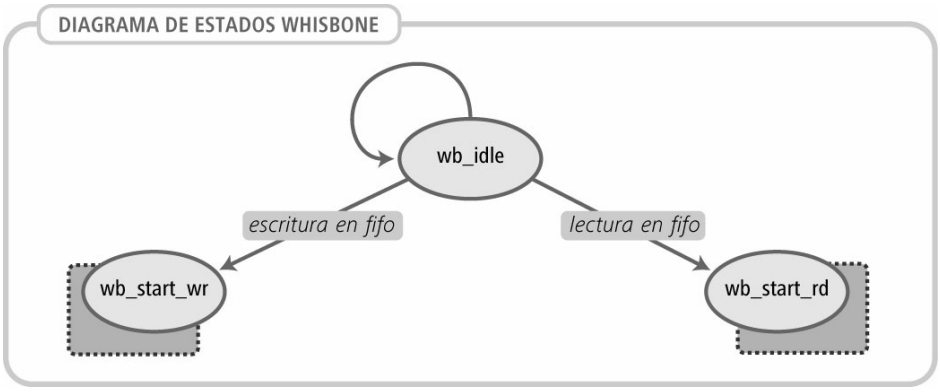
Si en algún momento de la transacción el pcir_fifo se encuentra vacío, se espera una cierta cantidad de períodos (configurable por el usuario) para dar tiempo a que Wishbone cargue datos. Si no se cargan datos durante ese tiempo, se realiza una desconexión sin transferencia de datos.



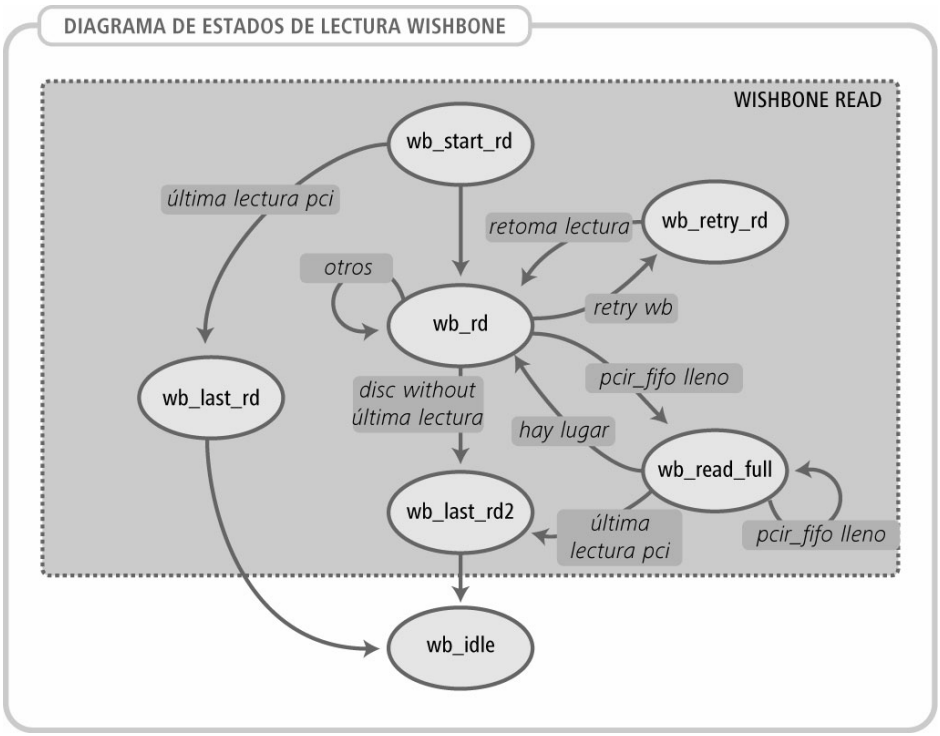
7.7.5.6. Pci Write

En una transacción de escritura, el core acepta los datos cuando el pciw_fifo está vacío. Si la aplicación Wishbone es más lenta que el bus PCI, el fifo se completa y se pasa a un estado de espera, que puede terminar por timeout en un disconnect without data (que el master reintentará nuevamente más tarde). Si la aplicación vacía el pciw_fifo antes, se continúa la transacción. En la última transferencia de datos se pasa a un estado de *turn arround* mediante el cual se libera el bus de acuerdo a lo especificado en el estándar.

7.7.6. Maquina de estado de Master Wishbone



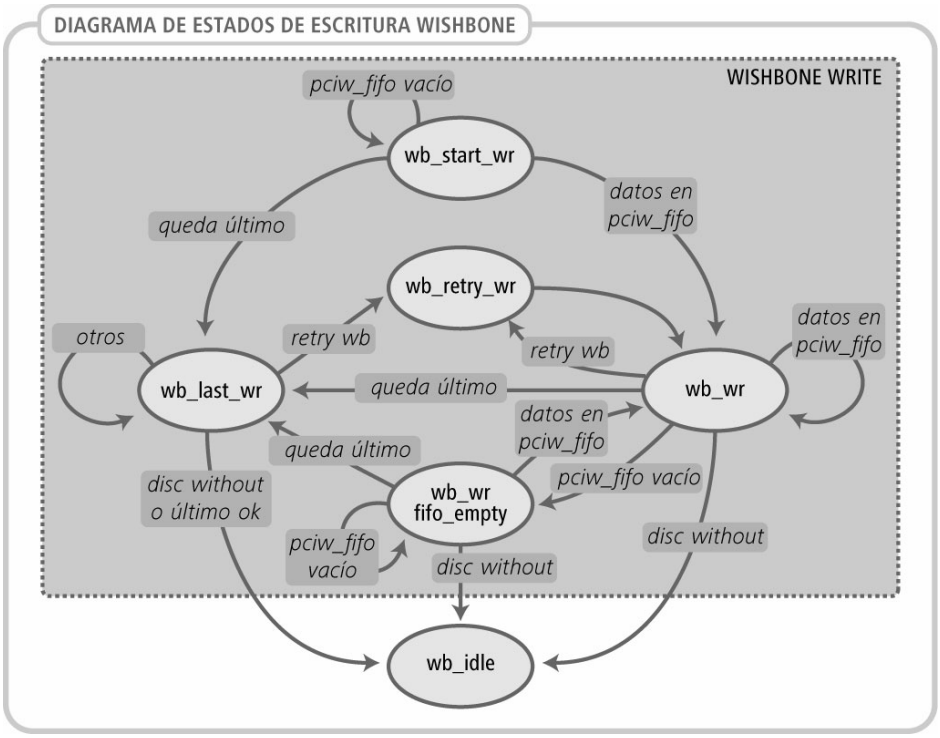
Cuando la interfaz PCI del core acepta una transacción, coloca en el pciw_fifo un comando que le indica al bloque Wishbone que operación debe realizar y la dirección de comienzo. Se puede realizar entonces una lectura o una escritura.



7.7.6.1. Wishbone Read

Se realizan ciclos de lectura por el interfaz Wishbone hasta que se realice la última lectura en el lado PCI. En caso de que se llene el pcir_fifo, se pasa a un estado de

espera hasta que desde PCI se vacíen los datos, o se termine la transacción.
En caso de que un dispositivo solicite un *retry* se corta el ciclo Wishbone y se reintenta un nuevo ciclo de lectura un ciclo de reloj después.



7.7.6.2. Wishbone Write

Se espera hasta que el pciw_fifo contenga datos, y en ese momento se pasan a realizar escrituras Wishbone. Si un dispositivo Wishbone solicita un *retry*, se vuelve a intentar un ciclo de reloj más tarde. Si el dato es el último, se escribe el dato y se vuelve al estado de espera.
Si se vacía el pciw_fifo, se pasa a un estado de espera hasta que aparezcan datos nuevamente o se salga por un disconnect without data.

7.7.7. Detalles de implementación

7.7.7.1. Codificación one-hot

Para alcanzar mayores velocidades de reloj, y simplificar la lógica de transición de estados se utilizó la técnica de "one-hot" para codificar los estados.
Esta técnica consiste en utilizar un flip-flop por estado y solo vale '1' la salida del FF del estado activo.

El software SynplifyPRO puede codificar los estado automáticamente. La declaración de los estados y la configuración de atributos para ello se muestran a continuación:

```
TYPE state_type IS
       (state1, state2, state3, state4);
SIGNAL state, nxstate : state_type;
-- ONE HOT PARA SYPLICITY
ATTRIBUTE syn_encoding : string;
ATTRIBUTE syn_encoding OF state : signal IS "onehot";
```

La asignación de variables final es:
state1: 0001
state2: 0010
state3: 0100
state4: 1000

7.7.7.2. Salidas registradas.

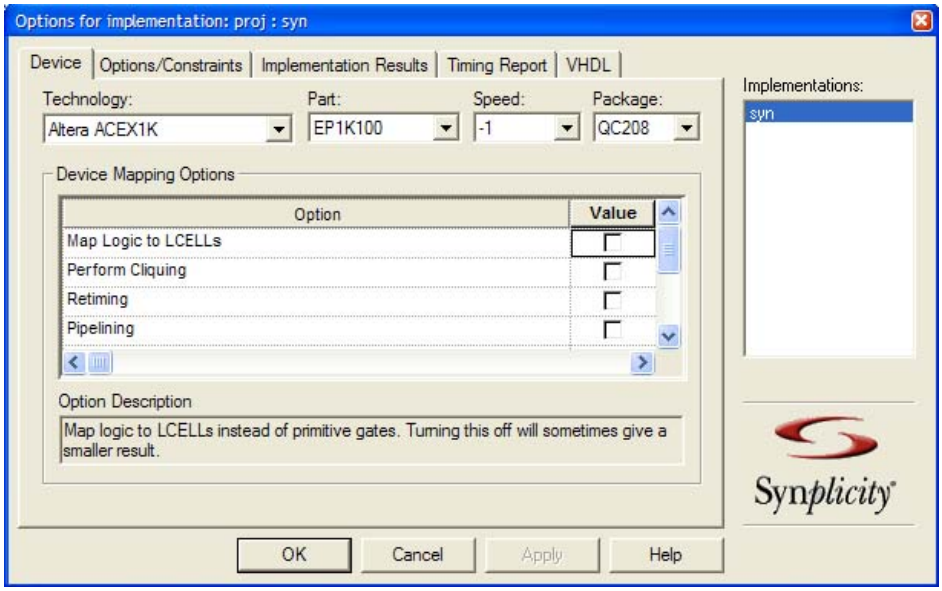
Todas las salidas del core PCITWBM están registradas, esto asegura que están libres de espurios.
Otro beneficio de esto es que evitan largas cadenas de lógica combinatoria lo que deteriora la performance del diseño.

7.8. Herramientas

7.8.1. VHDL a EDIF

El software sintetizador Max+PlusII soporta un sub-conjunto de VHDL muy limitado y los mensajes de error no son claros, lo que hace el trabajo muy difícil.
Ante esta situación se optó por utilizar algún software de síntesis que a partir de archivos VHDL generara archivos EDIF.
Los archivos EDIF son un estándar de la industria para el pasaje de diseños entre programas.
El software utilizado fue SynlifyPRO 7.0.1 de Synplicity (http://www.synplicity.com)

Durante las pruebas se consiguieron mejores resultados si no se seleccionaba el casillero Map Logic to LCELLs



Esta opción genera un EDIF con lógica mapeada a la estructura del FPGA en el que se va a sintetizar el diseño. Al no seleccionarse, solo se genera un archivo de nodos optimizado y el Max+PlusII se encarga finalmente de mapear la lógica a los recursos del FPGA de mejor forma.

7.8.2. Sintetizador

Una ves generado el EDIF se crea un proyecto con el archivo, automáticamente el

Max+PlusII identifica que el archivo fue creado con SynlifyPRO.

Las opciones de síntesis con las que se lograron diseños capaces de utilizar mayores velocidades de reloj fueron:

- En menú: *Global Project Logic Synthesis*
  - Optimización: 10 (speed)
  - *One-Hot State Machine Encoding* seleccionado
  - En sub-menú: *Define Synthesis Style*
    - Minimización: *Partial*
    - *Carry Chain*: Manual, Max. 8
    - *Cascade Chain*: Manual, Max. 8
    - Opciones Avanzadas: Todas las opciones seleccionadas

7.8.3. Simulador

Para las simulaciones se utilizó el simulador incluido en el software Max+PlusII.

7.9. Conclusiones

El objetivo de diseñar e implementar un core PCI en lenguaje VHDL fue alcanzado.

La elección del interfaz Wishbone para la comunicación con otros cores fue acertada, ya que se comprobó que es sencilla de utilizar.

El haber implementado únicamente el funcionamiento como target PCI permite que el tamaño del core sea pequeño dejando recursos del FPGA disponibles para implementar las aplicaciones. En caso del FPGA utilizado en la placa IIE-PCI, utiliza menos de una cuarta parte de los recursos.

A diferencia de la mayoría de los cores existentes, esta implementa un puente entre PCI y Wishbone. Por lo que para hacer uso de este no es necesario conocer el protocolo de comunicación PCI, sino solo el wishbone que es un estándar de comunicaciones entre cores IP, sin importar las funcionalidades que implementen.

El punto más débil del core PCI es la falta de un test bench exhaustivo. Debido a limitaciones de tiempo no se pudo implementar.

7.9.1. Modificaciones y recomendaciones

La principal modificación recomendada es la implementación de interrupciones. En el estado actual, la única forma de saber el estado de la aplicación sintetizada en la placa es utilizando *polling*.

La escritura del código VHDL del core fue un proceso de aprendizaje. Esto se distingue al comparar las primeras líneas de código escritas con las últimas. Sería interesante escribir nuevamente algunas partes con el conocimiento adquirido de modo de poder mejorar la frecuencia máxima de funcionamiento y optimizar el área ocupada al sintetizarlo.

Otra prueba interesante sería la de compilar y probar el core PCI en otro FPGA, incluyendo la posibilidad de sintetizarlo con las herramientas de Xilinx.

8. Software

8.1. Organización del capítulo

El capítulo se encuentra organizado en las siguientes secciones:

Características del software controlador
Resumen de las características técnicas del software controlador.

Driver RW_BAR
Descripción del diseño y la implementación del driver RW_BAR. Incluye una introducción a los tipos de drivers Linux existentes, creación de módulos, transferencia de datos entre dispositivos y apliaciones de usuario, y los detalles de implementación del código del driver.

Herramientas de prueba
Se mencionan las herramientas de prueba desarrolladas específicamente para utilizar junto con el driver.

Recursos PCI del sistema operativo Linux
Enumera las herramientas provistas por el sistema operativo Linux para trabajar con dispositivos conectados en el bus PCI. También hace una breve introducción a la medición precisa del tiempo y los registros de tipo de memoria (MTRR) de los procesadores Pentium II y superiores.

Conclusiones

8.2. Características del software controlador (driver)

Las principales características del driver son las siguientes:

- licencia de libre distribución
- compatible con Kernel Linux 2.4.x
- mapea recursos de la placa PCI como dispositivos de caracteres
- simple y fácil de utilizar
- simple y fácil de adaptar
- diseño modular, permite ser cargado en tiempo de ejecución no es necesario compliarlo con el kernel.
- provee información de performance, cantidad de bytes transferidos, tasas de transferencia, etc.
- provee una generosa cantidad de información sobre valores y datos durante la ejecución, facilitando el debugging de aplicaciones.

8.3. Driver RW_BAR

La forma correcta de utilizar las funciones provistas por una placa conectada al bus PCI dependen totalmente de la apliación. Para proveer de funcionalidades avanzadas, el driver debe ser desarrollado a medida, tomando en cuenta la aplicación final.
Como el driver debe permitir desarrollar y probar aplicaciones genéricas, el mismo implementa las funciones más básicas, pero imprecindibles, para asegurar su utilidad.

Este capitulo requiere que el lector tenga ciertos conocimientos sobre la arquitectura PCI y el sistema operativo Linux.

8.3.1. Diseño

8.3.1.1. Mapeo de direcciones PCI

La aplicación sintetizada en la placa se comunica con el exterior a través de rangos de direcciones en el espacio PCI. Dichas direcciones son asignadas en el momento de arranque del sistema. La cantidad, tamaño y propiedades de dichas regiones son especificadas por el core PCI a través del comportamiento característico que poseen los BAR.

Se tomaron las siguientes decisiones:
- las direcciones del espacio de configuración del puente PCI a WISHBONE se mapearon las direcciones en espacio PCI correspondiente al BAR0.
- el resto de los BAR están libres para ser utilizados por las aplicaciones sintetizadas dentro del hardware.

Los espacios de direcciones utilizados por el hardware de la tarjeta se mapean dentro del sistema de archivos reservado para dispositivos (directorio /dev). De esta manera se puede acceder a los espacios de drirecciones de PCI como si fuesen archivos, utilizando las clásicas funciones de acceso a archivos (open, seek, read, write y close).

8.3.1.2. Dispositivo de caracteres

El sistema operativo linux hace distición entre tres tipos de dispositivos:
- char device
- block device
- network interface

Los dispositivos de caracteres ¨char¨ son aquellos que puede ser accedidos como un flujo de bytes, o cómo si fuesen un archivo.

Históricamente, los dispositivos de bloques se acceden de a múltiplos de un bloque, dónde un bloque es generalmente 1 Kbyte. En Linux, un dispositivo de bloques puede transferir cualquier cantidad de bytes y no presenta diferencias, en este sentido, con un dispositivo de caracteres. Es importante notar que para albergar un sistema de archivos es fundamental que el dispositivo sea implementado cómo un dispositivo de bloques ya que proveed de ciertos interfaces necesarios para poder montarlo y accederlo.

Los interfaces de red (network interface) están pensados para intercambiar información con otros sistemas. Están diseñados para enviar y recibir información en forma de paquetes. Al no estar orientados a la transferencia de información en forma de flujo de bytes, no pueden ser accedidos cómo si fuesen archivos y por lo tanto no pueden ser mapeados dentro del sistema de archivos, ni se pueden usar sobre ellos las clásicas funciones de archivos (open, read, write, seek y close).
Un interfaz de red puede, sin embargo, generar una interrupción para ser atendido por el kernel; algo que es imposible para los dispositivos char y los block.

La opción más acorde a las necesidades y objetivos de nuestro diseño es la de un dispositivo de caracteres. Esta elección no impide el uso de interrupciones en la placa, pero debe escribirse código dentro del controlador (driver) que atienda dichas interrupciones y realice las tareas necesarias.

8.3.1.3. Modularización

Las formas existentes de implementar un controlador (driver) de un dispositivo pueden ser:

- compilar un kernel que incluya el código necesario para manejar dicho dispositivo
- compilar un módulo que pueda ser cargado y descargado en tiempo de ejecución

El segundo método es mucho más cómodo de utilizar en una diversidad de sitaciones, en particular cuándo se está desarrollando dicho driver. Si se desean hacer modificaciones del código, solo basta con descargar el módulo, hacer los cámbios en el código fuente y previa compilación, volverlo a cargar.

También permite distribuir e instalar el driver en un PC que no posea las herramientas de desarrollo necesarias para compilar el kernel.

No hay diferencias entre el código fuente de un controlador modular y uno compilado junto con el kernel.

Los cabezales que deben incluirse para implementar un módulo son:
- *linux/module.h* : para implementar un módulo.
- *linux/init.h* : funciones *module_init* y *module_exit* para carga y descarga del módulo
- *linux/sched.h* : definiciones de la mayor parte del API del kernel usadas por el driver.

Otros cabezales utilizados por nuestro driver y de uso frecuente:
- *linux/kernel.h* : funciones de debuging (cómo ser *printk* entre otras).
- *linux/errno.h* : definiciones de codigos de error
- *linux/pci.h* : funciones de ayuda para drivers pci
- *linux/poll.h* : llamadas poll y slect usadas para hacer lecturas y escrituras no bloqueantes de archivos abiertos
- *linux/proc_fs.h* : para utilizar el sistema de archivos especial /proc
- *asm/system.h*: barreras de memoria
- *asm/uaccess.h* : contiene las funciones *copy_to_user* y *copy_from_user*
- *asm/io.h* : funciones para leer y escribir puertos i/o
- *asm/irq.h* : funciones para manejo de IRQ
- *asm/msr.h* : registros específicos de plataforma (en este caso para medición de tiempo con funcion *rdtscl*)

También es necesario darle valor a los siguientes macros, cómo se muestra a continuación:

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("sebfer@fing.edu.uy - ciro@mondueri.com")
```

Para poder pasar parámetros al módulo y hacer configuraciones diferentes al momento de cargarlo, se puede utilizar el macros MODULE_PARAM. Para documentar el parámetro, se utiliza el macro MODULE_PARAM_DESC.

```
MODULE_PARM(rw_bar_major, "i");
MODULE_PARM_DESC(rw_bar_major, "Major node to be assigned. Dynamic allocation if 0 or
unspecified");
```

8.3.1.4. Transferencia de datos con el hardware

Los módulos son ejecutados en lo que se llama *espacio de kernel*, mientras que las aplicaciones corren en el *espacio de usuario*. Este concepto es fundamental en la

teoría de sistemas operativos.

El rol de un sistema operativo es el de proveer a los programas de acceso al hardware del sistema, en forma consistente y confiable. También debe permitir que los programas operen independientemente y proveer de protecciones para evitar el acceso no autorizado a los recursos de. hardware. Esta tarea es posible si el CPU provee dichas protecciones.

La forma en que los CPU proveen esta proteccion es implementando diferentes modos de operacion, o niveles. Dichos niveles tiene diferentes roles, y algunas operaciones están deshabilitadas en los niveles más bajos. El salto de niveles debe ser realizado a través de ciertos *portales*. Todos los procesadores modernos tienen al menos dos niveles de protección. En los sistemas operativos del tipo Unix, el kernel es ejecutado en el nivel más alto, dónde todo es permitido, mientras que las aplicaciones son ejecutdas en el nivel más bajo, donde el procesador regual el acceso directo al hardware y el acceso no autorizado a ciertas regiones de memoria.

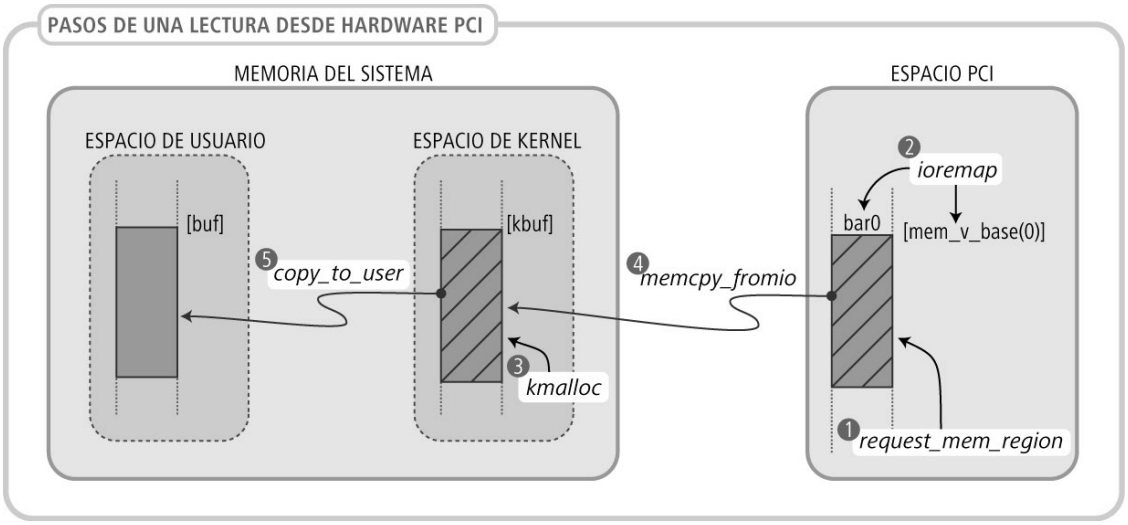
El *espacio de kernel* y el *espacio de usuario*, se refieren no sólo al modo correspondiente de operación del CPU, sino también al hecho que cada modo tiene su propio mapeo de direcciones de memoria, su propio direccionamiento de memoria.

Esta existencia de diferentes espacios de ejecución implica que la transferencia de datos entre las aplicaciones de usuario y el hardware deba ser un proceso de dos pasos.
Primero los datos deben transferirse del espacio de usuario al espacio de kernel, previa reserva de espacio suficiente de memoria de kernel por parte del módulo. Luego que los datos fueron almacenados en este buffer temporal en espacio de kernel, pueden entonces ser transferidos al hardware por parte del código del módulo, que se ejecuta en espacio de kernel.

El kernel provee funciones para hacer dichas transferencias. En el caso de una lectura desde el hardware PCI, las funciones utilizadas serían las siguientes.

- kmalloc : reserva espacio en memoria de kernel para el buffer temporal de transferencia.
- memcpy_fromio : realiza la transferencia desde el hardware hacia el buffer temporal en espacio de kernel.
- copy_to_user : copia desde buffer temporal en espacio de kernel al buffer en la aplicación de usuario.

El diagrama siguiente muestra los pasos necesarios para hacer la lectura antes mencionada:



La función memcpy_fromio trabaja sobre direcciones virtuales de memoria manejadas por el kernel, y no necesariamente con la dirección física real del dispositivo PCI. Esta direccion virtual se asigna mediante la función *ioremap*.

La función *ioremap* construye nuevas tablas de páginas de memoria, pero no reserva memoria. El valor devuelto por la función es una direccion virtual especial que puede ser usada para acceder al rango de direcciones físico especificado. Este valor obtenido no puede ser accedido directamente ya que es una dirección especial, y se deben utilizar funciones como *readb* o *memcpy_fromio*.

Pero para realizar estas transferencias, el controlador (driver) debe tener garantizado el acceso exclusivo a las regiones de memoria dónde está ubicado el dispositivo de hardware, para prevenir interferencias de otros drivers. Para esto el kernel provee tres funciones:

- *check_mem_region* : para saber si una cierta región de memoria correspondiente a un dispositivo está siendo utilizada
- *request_mem_region* : para solicitar la región de memoria correspondiente a un dispositivo
- *relase_mem_region* : para liberar la región de memoria correspondiente a un disporitivo, previamente solicitada.

8.3.2. Implementación

8.3.2.1. Estructura general de un módulo de kernel linux

Para implementar un módulo de kernel se deben implementar dos funciones, una de inicialización y otra de salida. El nombre de las funciones puede ser cualquiera y deben ser asociadas cómo tales pasando sus nombres como parámetros de las funciones module_init y module_exit.

Extracto del código RW_BAR:

```
/*****************************************************************
 * Hooks de inicializacion y salida del MODULO.
 *****************************************************************/
module_init(rw_bar_init);
module_exit(rw_bar_cleanup);
```

Las funciones de entrada y de salida deben ser marcadas con los macros __init y __exit para que sean procesadas correctamente por el compilador.

8.3.2.2. Manejo de dispositivos PCI en kernel linux

El kernel linux posee varias funcionalidades que hacen muy sencillo la codificación de un controlador PCI. Entre ellas están la búsqueda automática del dispositivo utilizando su número de fabricante y dispositivo, el registro de las funciones detección y extracción del dispositivo, etc.

Es necesario definir dos estructuras de datos, una con la información necesaria para poder identificar el dispositivo de hardware y la otra con las funciones de software a invocar en el momento de la inicialización y extracción del dispositivo. Se debe invocar el macro MODULE_DEVICE_TABLE para la estructura que contiene la información identificatoria del dispositivo.
A continuación se muestra un extracto del código con las estructuras mencionadas:

```
/*
 * dispositivos en los que se usa este driver
 */
static struct pci_device_id rw_bar_tbl[] __devinitdata = {
  { MY_DEVICE_VENDOR, MY_DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
};
MODULE_DEVICE_TABLE(pci, rw_bar_tbl);

/*
```

```
 * estructura del driver PCI
 */
static struct pci_driver rw_bar_driver = {
  name:         "rw_bar",
  probe:                rw_bar_init_one,
  remove:               rw_bar_remove_one,
  id_table:     rw_bar_tbl,
};
```

El registro de dichas estructuras se realiza desde dentro del código de inicialización del módulo de kernel (rw_bar_init en este caso) utilizando la función pci_module_init. Se comprueba el valor devuelto por dicha función para determinar si el hardware está presente o si hubo algún tipo de error. El código es el siguiente:

```
  /*
   * cargo el driver PCI.
   * pci_module_init detecta si el dispositivo especificado en
   * rw_bar_driver esta instalado en el sistema y en dicho caso
   *  ejecuta rw_bar_init_one
   */
  if ( pci_module_init (&rw_bar_driver) != 0){
    PDEBUG("rw_bar_init: pci_module_init != 0\n");
    return -ENODEV;
  }
```

La función de inicialización del dispositivo (*rw_bar_init_one*) debe ser marcada con el macro *_devinit para que sea correctamente procesada por el compilador. De igual forma, la función de extracción del dispositivo (_rw_bar_remove_one*) debe ser marcada con el macro __devexit.
Su única función importante es la de almacenar las direcciones de inicio y el tamaño de las regiones de memoria PCI correspondientes a cada BAR, en un par de variables de tipo array. Dichas variables serán accedidas luego desde las funciones de lectura y escritura de datos para identificar en qué dirección se deben acceder los datos de hardware.

A continuación, un extracto de código de la función *rw_bar_init_one* :

```
/*
 * Inicializacion del driver PCI (PROBE).
 */
static int __devinit rw_bar_init_one (struct pci_dev *pdev,
                                      const struct pci_device_id *ent)
{
```

```
  // hay 6 bars nada mas
  if (used_bars > 5) {
    used_bars = 6;
  }

  // para cada bar leer los valores
  int i;
  for (i = 0; i<used_bars; i++) {
    mem_base[i] = pci_resource_start(pdev, i);
    mem_size[i] = pci_resource_len(pdev, i);
    mem_v_base[i] = 0;

    if ( (mem_base[i] <= 0) || (mem_size[i] <= 0) ) {
      if (i == 0 ) {
        return -ENODEV;
      }
      continue;
    }
  }
  return 0;
}
```

8.3.2.3. Funciones de acceso a archivo

Para registrar las funciones que deben ser provistas por un dispositivo de caracteres, se debe:

- crear una estructura de datos con todos los punteros a las funciones
- registrar dicha estructura en el momento de inicialización del módulo (register_chrdev).

El primer parámetro de la función register_chrdev es el número mayor de dispositivo a asignar. Si es cero, el sistema asigna un número automáticamente devolviéndolo como resultado de la función. En el código de driver se contemplan las opciones de especificar como parámetro un número mayor de dispositivo, o dejar que el sistema lo seleccione automáticamente.

A continuación se muestran los dos extractos del código correspondientes:

```
/*
 * file operations
 */
static struct file_operations rw_bar_fops = {
  read: fop_read,
  write: fop_write,
  poll: fop_poll,
  open: fop_open,
```

```
  release: fop_release,
};

...

int __init rw_bar_init(void)
{
   ....
  /*
   * registro del character device (fops)
   */
  result = register_chrdev(rw_bar_major, "rw_bar", &rw_bar_fops);
  if (result < 0) {
    printk(KERN_INFO "rw_bar: can't get major number\n");
    return result;
  }
  if (rw_bar_major == 0)
    rw_bar_major = result; /* dynamic */
}
```

El trabajo real de este dispositivo es realizado por las funciones *fop_read* y *fop_write*. En ellas se hace todo el movimiento de datos entre el dispositivo hardware y el usuario. Son estas las funciones invocadas cuando se accede al dispositivo correspondiente en el sistema de archivos.
Dichas funciones se describen en la siguiente sección.

8.3.2.4. Transferencia de datos desde y hacia el hardware

En la carga del módulo, se reservan las direcciones de memoria utilizadas por cada BAR mediante la función *request_mem_region*, previo chequeo de su estado mediante la función *check_mem_region*.
Luego se hace corresponder dichas regiones de memoria del dispositivo PCI a direcciones que pueden ser accedidas desde el kernel mediante la función *ioremap*. A continuación un extracto de la función de carga del módulo dónde está contenido dicho procedimiento:

```
  /*
   * Reservo la region fisica de memoria correspondiente a cada BAR
   */
  for (i = 0; i<used_bars; i++) {
    if (check_mem_region(mem_base[i], mem_size[i])) {
      printk("drivername: memory already in use\n");
      return -EBUSY;
    }
    request_mem_region(mem_base[i], mem_size[i], "rw_bar");

    /*
```

```
     * Mapear memoria fisica dispositivo a memoria virtual
     */
    mem_v_base[i] = ioremap(mem_base[i], mem_size[i]);
  }
```

La transferencia de datos es realizada por las funciones *fop_read* y *fop_write*. Describimos unicamente la implementación de la función *fop_read* por su simetría característica.

1. reservar temporalmente memoria en espacio de kernel para hacer la transferencia (*kmalloc*)
2. hacer la transferencia desde el hardware a el espeacio temporal de memoria reservado (*memcpy_fromio*)
3. transferir desde el espacio temporal a el espacio de memoria de usuario (*copy_to_user*)
4. liberar la memoria reservada (*kfree*)

Es importante recalcar que el uso del parámetro GFP_KERNEL en la llamada a la función de reserva de memoria *kmalloc* puede hacer que el proceso sea puesto en espera por el kernel debido a la falta de memoria libre y por lo tanto su contexto debe ser reentrante. En caso de estar intentando obtener memoria desde una interrupción o timer, el parámetro adecuado es GFP_ATOMIC que jamás es puesto en espera y falla si directamente si no hay memoria libre.

A continuación se muestra la función *fop_read* simplificada (sin código de debug y estadísticas).

```
static ssize_t fop_read (struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
  int retval;
  unsigned char *kbuf, *ptr;
  void *add;
  int bar_to_read = MINOR(filp->f_dentry->d_inode->i_rdev);

  /* si quiero leer mas del tamanio disponible, achico count al
   *  tamanio max de ese bloque de memoria */
  if (count > mem_size[bar_to_read])
    /* mem_size_1 es el largo o el largo menos 1? */
    count = mem_size[bar_to_read] - 1;

  if (count < 0) return 0;

  /*
   * reserva de espacio temporal de memoria para hacer la transferencia
   */
  kbuf = kmalloc(count, GFP_KERNEL);
```

```
  if (!kbuf) return -ENOMEM;
  ptr=kbuf;
  retval=count;

  /*
   * Cambiar a nuestro espacio de direcciones virtual mapeado.
   */
  add = mem_v_base[bar_to_read] + *f_pos;

  /*
   * copia desde el dispositivo desde la direccion virtual al buffer temporal
   */
  memcpy_fromio(ptr, add, count);


  /*
   * copia desde buffer temporal al espacio de memoria de usuario
   */
  if (retval > 0)
    copy_to_user(buf, kbuf, retval);
  kfree(kbuf);
  *f_pos += retval;
  return retval;
}
```

La función *fop_read* recibe los siguientes parámetros:
- *flip: puntero a el nodo de sistema de archivos que está siendo accedido.
- *buf: puntero al espacio de memoria de usuario dónde se deben devolver los datos
- size: cantidad de bytes a leer
- *f_pos: desplazamiento en bytes a partir del cual empezar a leer

La función *fop_read* debe devolver como resultado la cantidad de bytes que fueron efectivamente leidos.

8.3.2.5. Uso de /proc para estadísticas

Para poder desplegar estadísticas de funcionamiento del hardware, se optó por la funcionalidad provista en el subdirectorio especial /proc.
Los archivos ahí presentes pueden ser accedidos cómo si fuesen archivos de caracteres permitiendo obtener información o configurar parámetros del módulo.
Se implementó el despliegue de datos estadísticos sobre la transferencia de datos desde y hacia el hardware. La información es recopilada inmediatamente antes y después de la ejecución de las funciones que hacen la transferencia desde y hacia el harware (*memcpy_toio* y *memcpy_fromio*) y almacenada en variables globales.

El código necesario para registrar un archivo dentro del subdirectorio especial /proc se invoca dentro de la carga del módulo y se muestra a continuación:

```
  /*
   * registro de entrada proc/drivers/iiepci
   */
  create_proc_read_entry("driver/iiepci",
                         0 /* default mode */,
                         NULL /*parent dir */,
                         rw_bar_read_procmem,
                         NULL /* client data */);
```

En este caso la función que realiza el trabajo de desplegar los datos se llama *rw_bar_read_procmem* . A continuación se muestra un extracto de dicha función:

```
int rw_bar_read_procmem(char *buf, char **start, off_t offset,
                        int count, int *eof, void *data)
{
  int len = 0;

  len += sprintf(buf+len, "\n--------------- IEEPCI Stats ---------------\n");
  /* memcopy */
  len += sprintf(buf+len, "-- MEMCOPY - WRITE ----------------\n");
  len += sprintf(buf+len, "total bytes    : %li\n", st_mc_w_total_bytes);
  len += sprintf(buf+len, "total tsc loops: %li\n", st_mc_w_total_tsc_loops);
  ...

  /* ultima */
  *eof = 1;
  return len;
}
```

Para almacenar los datos sobre la performance, se modifca el contenido de variables globales en los momentos inmediatamente enteriores y posteriores a la ejecución de la trasnsferencia de datos correspondientes a la lectura o escritura. A continuación se muestra un extracto del código correspondiente a la operación de escritura *fop_write* :

```
  rdtscl(st_mc_w_last_tsc_start);
  memcpy_toio(add, ptr, count);
  rdtscl(st_mc_w_last_tsc_end);
```

La primer llamada a la función *rdtscl* almacena el valor inicial del contador TSC del cpu (explicado más adelante en la sección de medición de tiempo) en una variable global.

Luego se realiza la transferencia hacia el hardware mediante la función *memcpy_toio*. Inmediatamente después se almacena el valor final del contador TSC en otra variable global.

En el momento de desplegar la información estadística, se realizan los cálculos correspndientes, ya sea para calcular el tiempo o la tasa de transferencia, utilizando un par de macros que hacen más sencillo el código (los macros son LOOPS2USECS y KBYTESPERSEC y están detallados al final de esta sección)

8.3.2.6. Compilación del driver

Para compilar el driver se deben utilizar los siguientes parámetros y símbolos:

```
CFLAGS = -D__KERNEL__ -DMODULE -Wall -O
```

El símbolo __KERNEL__ habilita muchas funciones útiles dentro de los cabezales de kernel. El símbolo MODULE debe ser incluido para todos aquellos drivers que no sean compilados dentro del kernel. El parámetro -Wall habilita el despliegue de todos los mensajes de adertencia del compilador. El parámetro -O es necesario porque muchas funciones están declaradas *inline* en los cabezales y el el compilador *gcc* no expande las funciones inline a menos que la optimización esté habilitada.

Junto con el código fuente del driver se incluye un archivo *Makefile* para facilitar el proceso de compilación.

8.3.2.7. Instalación y desinstalación del driver

Para instalar el módulo se utilza el programa *insmod* , al que se le pasan como parámetros el nombre del módulo, y los parámetros extra que este requiera. En nuestro caso es necesario especificar la cantidad de espacios de memoria PCI utilizados (cantidad de BAR).
Un ejemplo de la carga del módulo utilizando tres espacios de memoria PCI sería:

```
/sbin/insmod -f ./rw_bar.o "used_bars=3"
```

Para simplificar la tarea de determinar la cantidad de espacios de memoria PCI utilizados por el dispositivo se utiliza un script (*load_rw_bar.sh*), que hace uso de la herramienta lspci explicada más adelante. También en este script se crean los nodos en el sistema de archivos para acceder a cada espacio de memoria PCI.

Las partes más importantes del script serían:

```
#!/bin/bash

# variables utilizadas
vendor_id="1172"
device_id="abba"
module="rw_bar"
device="rw_bar"
mode="664"

# obtengo cantidad de espacios de memoria utilizados
used_bars = `lspci -v -d $vendor_id:$device_id | grep Memory | wc -l`

# instalo el módulo
/sbin/insmod -f ./$module.o "used_bars=$used_bars" $* || exit 1

# obtengo el major mode asignado al driver recién cargado
major=`cat /proc/devices | awk "\\$2==\"$module\" {print \\$1}"`

# creo los nodos en el sistema de archivos /dev
for (( I=0; $used_bars-$I; I=$I+1 )); do
   mknod /dev/${device}$I c $major $I
   chgrp $group /dev/${device}$I
   chmod $mode  /dev/${device}$I
done;
```

La descarga del driver utiliza el programa rmmod para desinstalar el módulo, y luego elimina los nodos correspondientes:

```
#!/bin/sh

# variables utilizadas
vendor_id="1172"
device_id="abba"
module="rw_bar"
device="rw_bar"

# desinstalar el módulo
/sbin/rmmod $module $* || exit 1

# eliminar todos los nodos creados
rm -f /dev/${device}?
```

8.4. Herramientas de prueba

La prueba del driver resulta bastante sencilla, ya que los espacios de memoria pueden ser accedidos tal como si fuesen archivos convencionales.

Se desarrollaron un par de scripts basados en el lenguaje Perl, uno de lectura y otro de escritura. Ambos scripts permiten manipular un byte en particular, ubicado en cualquier posicion dentro del espacio de memoria, utilizando un cierto *offset* o desplazamiento a partir del primer byte.

8.4.1. roff

El script *roff* permite leer desde un archivo una cierta cantidad de bytes especificada, a partir de un cierto desplazamiento deseado. El comando posee ayuda en linea.

Un ejemplo de utilización sería:

```
./roff.pl -C -o 2 -c 30 /dev/rw_bar1
```

Se leen 30 bytes (parámetro -c) a partir del byte 2 (parámetro -o) del archivo /dev/rw_bar1, y la salida se presenta en pantalla en forma hexadecimal y ascii en forma de columnas (parámetro -C).

El funcionamiento del programa se concentra en el siguiente extracto de código:

```
open(RW,$dev) || die "El archivo de entrada no existe\n";

# avanzo el offset deseado
if ($opt_o) {
    sysseek(RW,$offset, 1);
}

# preparo salida con hexdump si se solicita
if ($opt_C) {
    $pid = open(WRITEME, "| hexdump -C");
}

my $totalread = 0;
my $buf;

while(sysread(RW, $buf, $buffsize) and ($totalread < $count)){
    if ($opt_C) {
        print WRITEME $buf;
```

```
    } elsif ($opt_x) {
        print unpack("H*",$buf);
    } else {
        print $buf;
    }
    $totalread += length($buf);
}

if ($opt_C) {
    close(WRITEME);
}
close(RW);
```

Es importante mencionar que se utilizan las funciones *sysseek* y *sysread* de modo de evitar el uso de los buffers de entrada y salida, y poder controlar efectivamente el tamaño de bytes leídos en cada operación.

8.4.2. woff

El script *woff* permite escribir a un archivo una cierta cantidad de bytes, a partir de un cierto desplazamiento deseado. Los datos a escribir son recibidos por la entrada estándar del programa.

Un ejemplo sería el siguiente:

```
echo "testing" | ./woff.pl -o 2 /dev/rw_bar1
```

En este caso se escribe el texto "testing" al archivo /dev/rw_bar1, tomando como posición inicial de escritura el byte 2 dentro del archivo (parámetro -o). La cantidad de caracteres leída de la entrada debe ser especificada por el parámetro -c.
El uso del parámetro -x permite escribir cualquier valor deseado, representado en dos caracteres ASCII en forma hexadecimial. Por ejemplo, el siguiente ejemplo escribe el numero 255 en el primer byte del archivo /dev/rw_bar1:

```
echo ff | ./woff.pl -x -c 2 /dev/rw_bar1
```

En caso de utilizarse los parámetros -c y -x juntos, -c indica la cantidad de caracteres leídos de la entrada, que estar representados en ASCII hexadecimal, corresponden a la mitad de los que se van a escribir en el archivo de salida. El ejemplo anterior lee dos caracteres pero escribe un solo byte.

El funcionamiento del programa se resume en las siguientes líneas de código:

```
# abro el archivo de salida
open(RW,"+<$dev") || die "ERROR: el archivo de salida no existe\n";

# avanzo hasta el offset deseado
if ($opt_o) {
    sysseek(RW,$offset, 1);
}

my $totalwrite = 0;

while(read(STDIN, $buf, $buffsize) and ($totalwrite < $count) ){
    if ($opt_x) {
        # proceso la entrada como representacion ascii de hexa
        $writedata = pack("H*",$buf);
    } else {
        $writedata = $buf;
    }

    # utilizo llamada de escritura no buffereada
    $totalwrite += syswrite(RW, $writedata, length($writedata));
}

close(RW);
```

Es importante mencionar que se utilizan las funciones *sysseek* y *sysread* de modo de evitar el uso de los buffers de entrada y salida.

8.5. Recursos PCI del sistema operativo Linux

8.5.1. Disponibilidad de código fuente

El mejor recurso que posee el sistema operativo Linux para el desarrollo de controladores de hardware (drivers), es la disponibilidad de absolutamente todo su código fuente. La facilidad de poder entender a fondo cómo está implementada cada función utilizada, da otro dimensión al desarrollo de software. No es necesario hacer suposiciones ni estar adivinando el porqué de algunos comportamientos aparentemente arbitrarios. Basta con tomarse el tiempo y leer el código.

También la existencia de muchos drivers funcionales que forman parte de la distribución del sistema operativo permite aprender de dichos ejemplos.

8.5.2. Herramientas para dispositivos PCI

Linux posee abundantes herramientas para el desarrollo, testeo y debugging de hardware PCI. Mencionamos a coninuación algunas de las más utilizadas en nuestro proyecto.

8.5.2.1. /proc/pci

El archivo especial /proc/pci contiene un listado de todos los dispositivos PCI presentes en el sistema e información específica de cada uno. Entre la información desplegada se encuentra:

- número de bus en el que se encuentra el dispositivo
- tipo de dispositivo
- fabricante y modelo del dispositivo
- irq utilizada
- master o target, latencia, etc.
- información sobre direcciones de memoria utilizadas y su tipo.

Es muy util para obtener información rápidamente.

8.5.2.2. /proc/bus/pci

El sistema de archivos especial /proc/bus/pci permite obtener y modificar las configuraciones de los dispositivos PCI presentes. Su uso directo no es sencillo y es recomendable utilizar herrameintas para modificar estos valores, como ser *setpci* (mencionado más adelante).

8.5.2.3. pciutils

El paquete pciutils (que forma parte de la mayoría de las distribuciones de kernel 2.4) contiene dos aplicaciones extremadamente útiles: *lspci* y *setpci* .
Dichas aplicaciones permiten inspeccionar y configurar los dispositivos conectados al bus PCI, y realizan la mayoría de sus operaciones a través del sistema de archivos /proc/bus/pci

La aplicación *lspci* despliega información sobre todos los buses PCI presentes en el sistema, y los dispositivos que estén conectados en ellos.

La aplicación *setpci* permite obtener y modificar valores de configuración de los dispositivos PCI conectados. Esta apliación permite hacer cambios en los registros de configuración de un dispositivo en forma muy sencilla. Fue utilizada extensivamente en las primeras etapas de desarrollo del core PCI.

8.5.3. Medición del tiempo

En ciertos casos es necesario poder medir intervalos de tiempo de ejecución. En nuestro proyecto fue necesario medir intervalos de tiempo para poder obtener valores sobre la performance del sistema.

Si los intervalos a medir no son demasiado pequeños, se puede utilizar el valor de la variable *jiffies*. Este contador es incrementado cada vez que se genera una interrupción de timer. La frecuencia de interrupción de timer es dependiente de la plataforma y está defifida en por el valor de la constante *HZ* .
Hay que tomar en cuenta que el contador de *jiffies* normalmente hace overflow cada aproximadamente 16 meses de ejcución.

Si el intervalo a medir es muy pequeño o se necesita mucha precisión en los números, se pueden utilizar recursos dependientes de cada plataforma, como ser el uso de algún registro específico del procesador. La mayoría de los CPU modernos incluyn un contador de alta resolución que es incrementado en cada ciclo de reloj. Este contador permite medir intervalos en forma precisa.
En el caso de la arquitectura x86, el registro se denomina TSC (timestamp conuter), y fue introducido con la linea de procesadores Penium. Es un registro de 64 bits que cuenta los ciclos de reloj del procesador. Puede ser leído indistintamente desde el espacio de kernel y desde el espacio de usuario.

Es necesario incluir el cabezal asm/msr.h para poder accederlo con los siguientes

macros:
- *rdtsc(low,hig)* : lee el valor de 64 bits en dos variables de 32 bits.
- *rdtsc(low)* : lee la mitad inferior del registro en una variable de 32 bits.

Para dar una idea, en un procesador de 1GHz, la mitad baja de 32 bits del registro hace overflow cada 4.25 segundos.

Para obtener el tiempo en segundos al que corresponden tantos ciclos de procesador, se puede utilizar la variable current_cpu_data.loops_per_jiffy, que es el resultado de un calculo llamado bogomips que se realiza al iniciar el sistema. Esta variable indica cuantos ciclos de CPU hay en un *jiffy*, que es 1/HZ segundos (en la arquitectura x86, generalmente, un *jiffy* equivale a una centésima de segundo).

Un macro util para realizar la conversión de ciclos de cpu (loops) a segundos es el siguiente:

```
/* loops to usecs */
#define LOOPS2USECS(loops) \
   (((double)loops* ((double)1000000/((double)HZ* \
   (double)current_cpu_data.loops_per_jiffy))))
```

Un macro que facilita el cálculo de la tasa de transferencia es el siguiente:

```
#define KBYTESPERSEC(bytes, loops) \
   ((((double)(bytes)/(double)loops)* \
   (HZ*current_cpu_data.loops_per_jiffy))/((double)1024
```

8.5.4. MTRR

En la familia de procesadores Intel P6 (Pentium Pro, Pentium II, Pentium III y posteriores) exiten registros llamados MTRR (Memory Type Range Registers) que son utilizados para controlar el acceso por parte del procesador a ciertos rangos de memoria. Estos permiten que el procesador optimice las operaciones para los diferentes tipos de memoria accedidos, como ser RAM, ROM, frame buffers, dispositivos mapeados en memoria, etc.

Esto es de gran utilidad cuando se tiene un dispositivo con grandes areas de memoria para ser accedidas, ubicado en el bus PCI o en el AGP y su manejo de los accesos es desordenado. Es de particular uso para aquellos dispositivos que poseen frame buffers, en especial las tarjetas de video.

Generalmente los servidores X normalmente modifican estos registros.

Mediante la configuración de dichas areas de memoria en modo write-combining (WC) se permite que las transferencias en el bus sean agrupadas antes de ser enviadas, aumentando la performance de las escrituras hasta en 2 veces o más.
La memoria configurada como de tipo write-combining no es cacheable en los caches L1 o L2, lo cual es coherente con el uso que se le quiere dar, ya que no tiene sentido guardar informacion en un cache de datos que pertenencen a un dispositivo de entrada-salida.

Los procesadores Cyrix 6x86, AMD K6-3, AMD Athlon, también poseen registros que proporcionan un funcionamiento similar.

La configuración de dichos registros se hace modificando el archivo especial /proc/mtrr.

Un ejemplo de configuración es el siguiente:

```
echo "base=0xf8000000 size=0x400000 type=write-combining" > /proc/mtrr
```

Dicha configuración hace que el rango de memoria del espacio PCI desde la dirección 0xf8000000 hasta la 0xf8400000 trabajen en modo write-combining.

Esto se comprueba mediante el siguiente comando:

```
cat /proc/mtrr

reg00: base=0x00000000 (   0MB), size= 128MB: write-back, count=1
reg01: base=0xf8000000 (3968MB), size=   4MB: write-combining, count=1
```

Para elmininar, por ejemplo , la configuración del registro 1 se debe ejecutar el siguiente comando:

```
echo "disable=1" > /proc/mtrr
```

8.6. Conclusiones

Los objetivos planteados fueron alcanzados, ya que se logró desarrollar un driver PCI para el sistema operativo Linux.

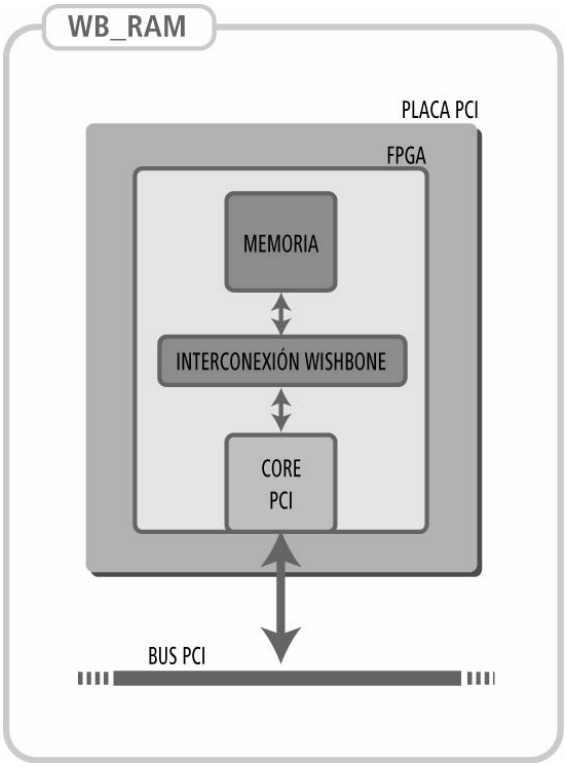
El proceso permitió descrubrir la infinidad de herramientas existentes para el desarrollo y prueba de dispositivos PCI en dicho sistema operativo, amparado por las licencias de libre distribución.

9. Aplicaciones de prueba

9.1. WB_RAM

9.1.1. Descripción general

Esta aplicación implementa una memoria dentro del FPGA. Se pueden realizar lecturas y escrituras a esta, utilizando el rango de direcciones asignado a BAR1 del core PCI.



Fue utilizada mayormente para desarrollar el driver y realizar las primeras pruebas del core PCI.

9.1.2. Descripción funcional

La aplicación usa una instancia del core PCITWBM para 2 BARs, BAR0 para los registros de traslación de direcciones y BAR1 para la memoria WB_RAM dentro del FPGA.

La memoria WB_RAM tiene ancho de palabra 32bits y una capacidad de 2048 bytes (512 DWORD).
Los accesos de lectura y escritura deben de ser de 32 bits de ancho, no permite

escritura de a byte.

La aplicación es la encargada de conectarlos utilizando sus interfaces Wishbone.

9.1.3. Archivos requeridos

El diseño requiere de:

- todos los archivos que componen el core PCITWBM.
- archivo de la memoria: *wb_ram.vhd*
- archivo de la aplicación: *iie_pci_wb_ram.vhd*

9.1.3.1. pcitwbm

Parámetros de la instancia:

```
pcitwbm_top0: pcitwbm_top
    GENERIC MAP(vendor_id    =>  X"1172",
                device_id    =>  X"ABBA",
                subsystem_id =>  X"10E9",
                subsystem_vid =>  X"10E9",
                NUMBER_OF_BARS => 2,
                BAR_0_SIZE     => 64,
                BAR_0_LOW_NIBBLE => 0,
                BAR_1_SIZE     => 2048, -- mapero wb_ram de 2048 bytes
                BAR_1_LOW_NIBBLE => 0,
                FIFO_NUMWORDS  => 14,
                LAT_TIMER_INITIAL_VALUE => 7)
```

9.1.3.2. wb_ram

El componente wb_ram tiene el siguiente interfaz:

```
 COMPONENT wb_ram_interface IS
 GENERIC (RAM_WIDTH : integer := 32;
          RAM_ADDRESS_WIDTH : integer := 7);
  PORT(
     --WB signals
     RST_I    : IN  STD_LOGIC;
     CLK_I    : IN  STD_LOGIC;
     DAT_I    : IN  STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
     DAT_O    : OUT STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
     ACK_O    : OUT STD_LOGIC;
     ADR_I    : IN  STD_LOGIC_VECTOR(RAM_ADDRESS_WIDTH-1 downto 0);
     CYC_I    : IN  STD_LOGIC;
     STB_I    : IN  STD_LOGIC;
     WE_I     : IN  STD_LOGIC;
     CTI_I    : IN STD_LOGIC_VECTOR(2 downto 0) -- Cycle type identifier
     );
  END component;
```

Internamente, wb_ram esta compuesta por una *lpm_ram_dp* y una máquina de estados para manejar el interfaz wishbone. La *lpm_ram_dp* es provista por la librería lpm.

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;
```

Parámetros utilizados:

```
wb_ram0: wb_ram_interface
GENERIC MAP (RAM_WIDTH => 32,
     RAM_ADDRESS_WIDTH => 11)
```

9.1.4. Síntesis

Utilizando el Synplify PRO 7.0.1, se creó un archivo de tipo EDIF a partir de la descripción VHDL. Luego, utilizando el MAX+Plus II se sintetizó para el FPGA de la placa IIE-PCI.

Resultados de la síntesis:

FPGA	Memoria Interna (bits)	Celdas Lógicas	fmax (MHz)
EP1K100QC208-2	17568 (35%)	1105 (22%)	35.21
EP1K100QC208-1	17568 (35%)	1105 (22%)	49.01

La asignación de pines utilizada es la incluida en el manual de usuario del core PCITWBM.

Obs: Para realizar las simulaciones de lectura y escritura, previamente deben incluirse los ciclos que inicializan el espacio de configuración del core PCITWBM, cómo lo haría el sistema al arrancar. Es necesario asignar una dirección de comienzo al BAR0 y BAR1, realizando un ciclo PCI de configuración.

9.1.5. Pruebas

Se escribió la memoria con el contenido de un archivo cualquiera y se volvió a leer su valor, comparándolo para verificar que las escrituras y lecturas se estuviesen realizando correctamente.
Para esto se utilizaron los utilitarios *woff.pl* y *roff.pl*, junto con el comando de unix *cmp* para realizar la comparación.

Se transcribe la secuencia de comandos:

```
head -c 2048 *archivo* > test.bin
./woff.pl -c 2048 /dev/rw_bar1 < test.bin
./roff.pl -c 2048 /dev/rw_bar1 > mem.bin
cmp test.bin mem.bin
```

Para 10 lecturas y escrituras, realizadas con el FPGA de velocidad -1, las estadísticas son las siguientes:

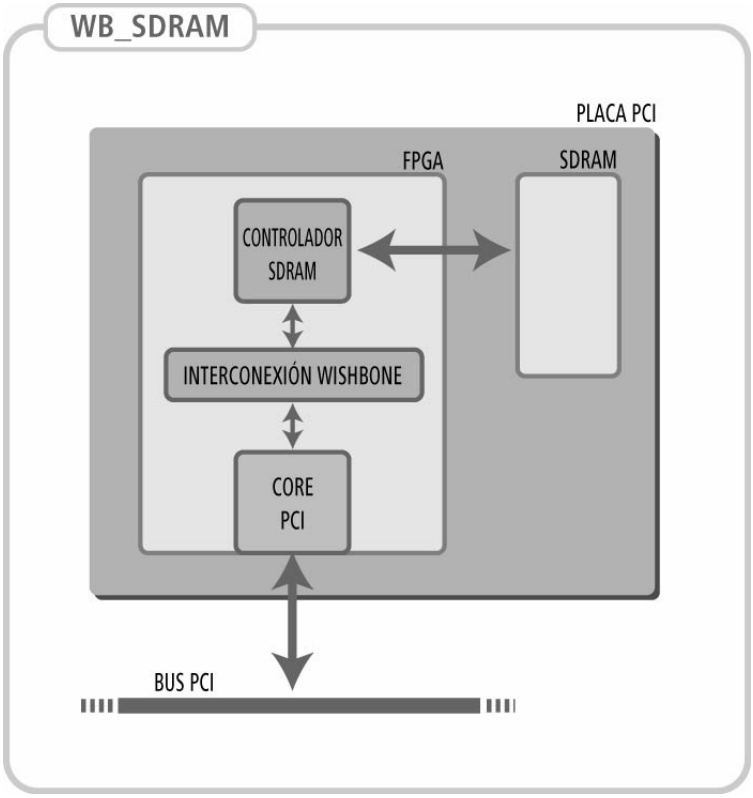
```
--------- IEEPCI Stats - HZ :  512 - loops_per_jiffy:   778240 -------

--------- W R I T E  --  M E M C O P Y -------------------------------
          |   KB/sec     |     bytes    | microseconds | tsc loops
----------+--------------+--------------+--------------+--------------
TOTAL     |         32253|         20480|           620|        247083
BAR1 TOTAL|         32253|         20480|           620|        247083
BAR2 TOTAL|             0|             0|             0|             1
BAR3 TOTAL|             0|             0|             0|             1
LAST WRITE|         32463|           512|            15|          6137
--------- R E A D  --  M E M C O P Y -------------------------------
          |   KB/sec     |     bytes    | microseconds | tsc loops
----------+--------------+--------------+--------------+--------------
TOTAL     |          6739|         25600|          3709|       1478108
BAR1 TOTAL|          6739|         25600|          3709|       1478108
BAR2 TOTAL|             0|             0|             0|             1
BAR3 TOTAL|             0|             0|             0|             1
LAST READ |          7586|           512|            65|         26260
```

9.2. WB_SDRAM

9.2.1. Descripción general

Para realizar pruebas sobre la memoria SDRAM de la placa se desarrolló esta aplicación, que esta compuesta por el core PCITWBM, un core controlador SDRAM desarrollado por Jimena Saporiti y Agustin Villavedra y las señales de interconexión necesarias.



9.2.2. Descripción funcional

La aplicación usa una instancia del core PCITWBM para 2 BARs, BAR0 para los registros de traslación de direcciones y BAR1 para la memoria SDRAM de la placa IIE-PCI.

La memoria SDRAM tiene ancho de palabra 32bits y una capacidad de 16MB y es manejada por un controlador SDRAM con interfaz Wishbone.

La aplicación es la encargada de conectarlos utilizando sus interfaces Wishbone.

9.2.3. Archivos requeridos

El diseño requiere de:

- todos los archivos que componen el core PCITWBM.
- todos los archivos que componen el controlador SDRAM.
- archivo de la aplicación: *iie_pci_sdramdl2.vhd*

9.2.3.1. pcitwbm

Parámetros de la instancia:

```
pcitwbm_top0: pcitwbm_top
GENERIC MAP(vendor_id    =>  X"1172",
               device_id      =>  X"ABBA",
               subsystem_id   =>  X"10E9",
               subsystem_vid  =>  X"10E9",
               NUMBER_OF_BARS => 2,
               BAR_0_SIZE     => 64,
               BAR_0_LOW_NIBBLE => 0,
               BAR_1_SIZE     => 16777216, -- 16MBytes
               BAR_1_LOW_NIBBLE => 0,
               FIFO_NUMWORDS  => 14,
               LAT_TIMER_INITIAL_VALUE => 7)
```

9.2.3.2. wb_sdram

El componente tiene la siguiente interfaz:

```
COMPONENT principal
   PORT(
            --Puertos de comunicación con el host según el protocolo wishbone:
      clk_i      :IN STD_LOGIC;
      rst_i      :IN STD_LOGIC;
      cyc_i      :IN STD_LOGIC;
      stb_i      :IN STD_LOGIC;
      we_i      :IN STD_LOGIC;
      ack_o      :OUT STD_LOGIC;
      dat_i      :IN STD_LOGIC_VECTOR(ancho_palabra-1 downto 0);
      sel_i       :IN STD_LOGIC_VECTOR(3 downto 0);
      dat_o      :OUT STD_LOGIC_VECTOR(ancho_palabra-1 downto 0);
      adr_i       :IN STD_LOGIC_VECTOR(num_dir-1 downto 0);
            --Puertos de comunicación con la memoria:
      dqm         :OUT STD_LOGIC_VECTOR(granos-1 downto 0);
      dq         :INOUT STD_LOGIC_VECTOR(ancho_palabra-1 downto 0);
      comandos   :OUT STD_LOGIC_VECTOR(3 downto 0);
      dir_mem      :OUT STD_LOGIC_VECTOR(num_mem-1 downto 0);
      dir_banco   :OUT STD_LOGIC_VECTOR(num_banco-1 downto 0)
      );
END COMPONENT;
```

Los parámetros de configuración son constantes dentro de la librería del componente. Más información sobre el componente puede encontrarse en: http://iie.fing.edu.uy/ense/asign/dlp/proyectos/2003/sdram/index.htm

9.2.4. Síntesis

Utilizando el Synplify PRO 7.0.1, se creó un archivo de tipo EDIF a partir de la descripción VHDL. Luego, utilizando el MAX+Plus II se sintetizó para el FPGA de la placa IIE-PCI.

Resultados de la síntesis:

FPGA	**Memoria Interna (bits)**	**Celdas Lógicas**	*fmax (MHz) *
EP1K100QC208-1	17568 (35%)	1105 (22%)	35.21

La asignación de pines utilizada es la incluida en el manual de usuario del core PCITWBM.

Obs: Para realizar las simulaciones de lectura y escritura, previamente deben incluirse los ciclos que inicializan el espacio de configuración del core PCITWBM, cómo lo haría el sistema al arrancar. Es necesario asignar una dirección de comienzo al BAR0 y BAR1, realizando un ciclo PCI de configuración.

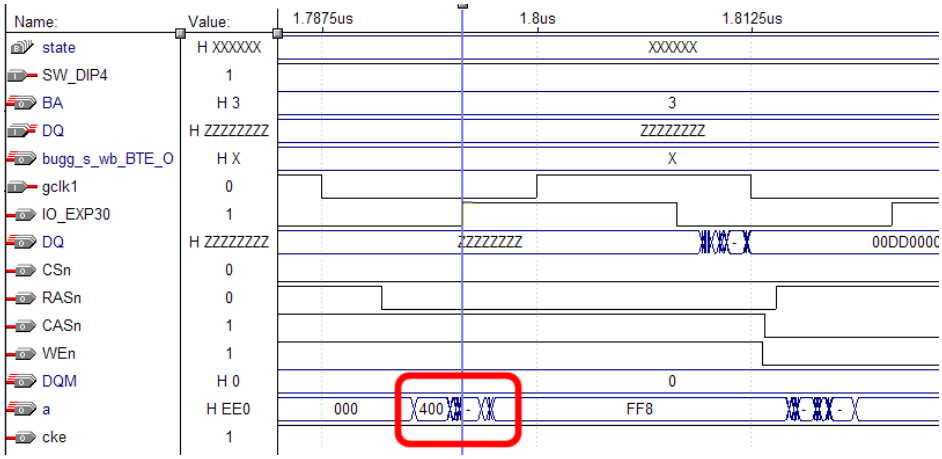
9.2.5. Pruebas

Se escribió la memoria con el contenido de un archivo cualquiera y se volvió a leer su valor, comparándolo para verificar que las escrituras y lecturas se estuviesen realizando correctamente.
Para esto se utilizaron los utilitarios *woff.pl* y *roff.pl*, junto con el comando de unix *cmp -l* para realizar la comparación.

Las pruebas se realizaron de igual forma que para la aplicación que usa la WB_RAM.
Se pudo constatar diferencias entre los archivos escritos y los datos leídos. El problema esta en las escrituras a la SDRAM, no en las lecturas, ya los valores leídos son siempre iguales. Las diferencias entre lo escrito y lo leído se presentan en forma de ráfagas.
Esto se atribuye a que el diseño del controlador SDRAM no utiliza salidas y entradas registradas, haciendo que las salidas estén conmutando continuamente.

En la siguiente simulación se puede ver que las direcciones de selección de la columna

no están estables en el flanco de reloj.



Para 10 lecturas y escrituras, realizadas con el FPGA de velocidad -1, las estadísticas son las siguientes:

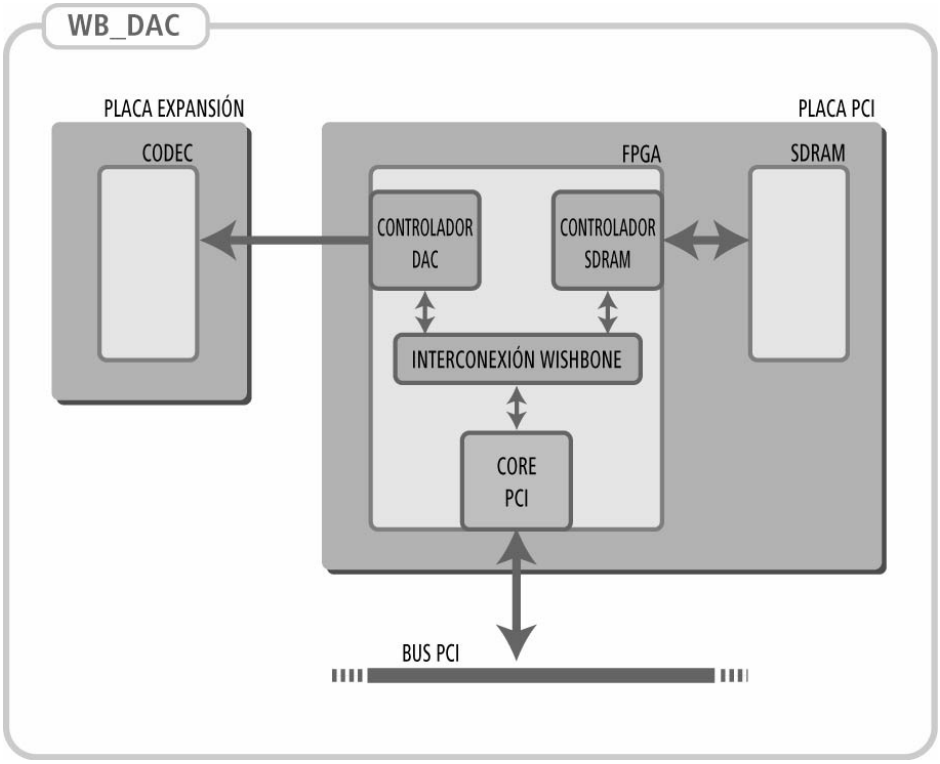
```
--------- IEEPCI Stats - HZ :  512 - loops_per_jiffy:   778240 -------

--------- W R I T E  --  M E M C O P Y -------------------------------
          |   KB/sec     |     bytes    | microseconds | tsc loops
----------+--------------+--------------+--------------+--------------
TOTAL     |         32236|         20480|           620|        247208
BAR1 TOTAL|         32236|         20480|           620|        247208
BAR2 TOTAL|             0|             0|             0|             1
BAR3 TOTAL|             0|             0|             0|             1
LAST WRITE|         32453|           512|            15|          6139
--------- R E A D  --  M E M C O P Y -------------------------------
          |   KB/sec     |     bytes    | microseconds | tsc loops
----------+--------------+--------------+--------------+--------------
TOTAL     |          5566|         25600|          4491|       1789575
BAR1 TOTAL|          5566|         25600|          4491|       1789575
BAR2 TOTAL|             0|             0|             0|             1
BAR3 TOTAL|             0|             0|             0|             1
LAST READ |          5555|           512|            90|         35862
```

9.3. WB_DAC

9.3.1. Descripción general

Como demostración para la presentación del proyecto, y como prueba de interconexión de múltiples bloques Wishbone desarrollados por diferentes personas, se realizó un reproductor de archivos de audio.



La aplicación está compuesta por:

- core PCITWBM
- controlador SDRAM (WB_SDRAM)
- interfaz wishbone para un DAC (WB_DAC)
- lógica de interconexión para arbitrar el uso de la memoria entre el core PCITWBM y el core WB_DAC.
- placa externa con un DAC

9.3.2. Descripción funcional

La aplicación usa una instancia del core PCITWBM para 3 BARs, BAR0 para los registros

de traslación de direcciones, BAR1 para los registros de control y BAR2 para la memoria SDRAM, donde se escriben las muestras de audio a ser reproducidas por el DAC.
La memoria SDRAM tiene una capacidad de 16MBytes.

La aplicación puede estar en tres posibles estados, reproduciendo, en pausa o detenida.
Esto se controla mediante los dos bits menos significativos del registro de control ubicado en la dirección 0 de BAR1.
Si bit[0]=0, la aplicación está en el estado detenida. Pueden escribirse y leerse datos a la memoria.
Si bit[1:0]=01, la aplicación reproduce el contenido de la memoria. La memoria solo puede ser accedida por WB_DAC. Para volver a utilizar la memoria desde el PCI, debe pasarse al estado detenida.
Si bit[1:0]=11, la aplicación esta en estado de pausa. Cuando se vuelve al estado de reproducción se continua desde la última dirección de memoria reproducida.

Se utiliza el codec de la placa de expansión Xsten Board V1.3 de XESS.

9.3.3. Archivos requeridos

- todos los archivos que componen el core PCITWBM.
- todos los archivos que componen el controlador SDRAM.
- todos los archivos que componen el DAC con interfaz WB.
- archivo de la aplicación: *iie_pci_sdramdl2_dac.vhd*
- registro con interfaz wishbone: *wb_register_interface.vhd*

9.3.3.1. pcitwbm

```
GENERIC MAP(vendor_id    =>  X"1172",
               device_id      =>  X"ABBE",
               subsystem_id   =>  X"10E9",
               subsystem_vid  =>  X"10E9",
               NUMBER_OF_BARS => 3,
               BAR_0_SIZE     => 64,
               BAR_0_LOW_NIBBLE => 0,
               BAR_1_SIZE     => 64, -- mapero registro
               BAR_1_LOW_NIBBLE => 0,
               BAR_2_SIZE     => 16777216, -- mapero SDRAM
               BAR_2_LOW_NIBBLE => 0,
               FIFO_NUMWORDS  => 14,
               LAT_TIMER_INITIAL_VALUE => 7)
```

9.3.3.2. wb_register

El interfaz del registro es:

```
COMPONENT wb_register_interface IS
   GENERIC (data_width: POSITIVE);
   PORT(
      --WB signals
        RST_I      : IN   STD_LOGIC;
      CLK_I      : IN   STD_LOGIC;
      DAT_I    : IN   STD_LOGIC_VECTOR(data_width-1 downto 0);
      DAT_O    : OUT   STD_LOGIC_VECTOR(data_width-1 downto 0);
      ACK_O      : OUT   STD_LOGIC;
      CYC_I      : IN   STD_LOGIC;
      STB_I      : IN   STD_LOGIC;
      --register output signals
      Q        : OUT   STD_LOGIC_VECTOR(data_width-1 downto 0)
      );
END component;
```

Se utiliza con un ancho de 32 bits, data_width=32.

9.3.3.3. wb_dac

El codec de la placa Xstend se controla con el bloque WB_DAC. Actúa como master Wishbone, leyendo datos de la memoria SDRAM y enviándolos al codec.

El interfaz del módulo es:

```
component wb_dac_interface IS
      generic
      (
         XSTEND_V1_2  : boolean  := false; -- XSTEND board model
         DAC_WIDTH: positive := 20;
         CLK_DIV_WIDTH : positive := 1
      );
      PORT(
      --WB signals
      RST_I    : IN  STD_LOGIC;
      CLK_I    : IN  STD_LOGIC;
      DAT_I    : IN  STD_LOGIC_VECTOR(DAC_WIDTH-1 downto 0);
      ACK_I    : IN  STD_LOGIC;
      CYC_O    : OUT STD_LOGIC;
      RTY_I    : IN  STD_LOGIC;
      SEL_O    : OUT STD_LOGIC_VECTOR(3 downto 0);
      STB_O    : OUT STD_LOGIC;
      WE_O     : OUT STD_LOGIC;
      CTI_O    : OUT STD_LOGIC_VECTOR(2 downto 0);
```

```
    --codec_signals
    mclk: out std_logic;
    lrck: out std_logic;
    sclk: out std_logic;
    sdout: in std_logic;
    sdin: out std_logic
    );
  END component;
```

9.3.4. Síntesis

Utilizando el Synplify PRO 7.0.1, se creó un archivo de tipo EDIF a partir de la descripción VHDL. Luego, utilizando el MAX+Plus II se sintetizó para el FPGA de la placa IIE-PCI.

Resultados de la síntesis:

FPGA	Memoria Interna (bits)	Celdas Lógicas	fmax (MHz)
EP1K100QC208-1	1216 (2%)	1772 (35%)	34.48

9.3.5. Pruebas

La frecuencia de reproducción de muestras del codec es:

$$fs = \frac{CLK\_I}{512 \times 2^{CLK\_DIV\_WID}}$$

Debido a la limitación de no poder utilizar dos fuentes de reloj simultáneas en la placa, la frecuencia fs más cercana al estándar de 44.1KHz utilizado por audio fue de 32.226KHz.Con un programa de edición de archivos WAV, se generó un archivo de audio remuestreado a esta frecuencia. Los primeros 4096 bytes de los archivos WAV forman parte del cabezal y no contienen información de audio. En caso de ser un archivo WAV de 16 bits estéreo, se alternan las muestras izquierda y derecha, que es la misma secuencia utilizada por el codec, por lo que puede ser cargado directamente en la memoria con el siguiente comando:

```
./roff.pl -o 4096 -c 16777216 archivodeaudio.wav | ./woff -c 16777216 /dev/rw_bar2
```

Para iniciar, pausar y detener la reproducción se utilizan los siguientes comandos:

```
echo -n 01 > ./woff.pl -x /dev/rw_bar1
echo -n 03 > ./woff.pl -x /dev/rw_bar1
echo -n 00 > ./woff.pl -x /dev/rw_bar1
```

10. Referencias, Bibliografia y Glosario

10.1. Referencias

10.1.A P. Aguayo, "Implementación digital de un banco de filtros gaussianos Wavelet. Diseño de un Interfaz al bus PCI Esclavo."

10.1.B Altera, "AN 75: High-Speed Board Designs", Noviembre 2001, http://www.altera.com/literature/an/an075.pdf.

10.1.C Myszne, Jorge; Calvo, Germán; Miguez, Juan Andrés. "Implementación de algoritmos de tratamiento de imágenes en lógica reconfigurable", proyecto de graduación, IIE, 1998.

10.1.D Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs", 2002, http://www.sunburst-design.com/papers/CummingsSNUG2000Boston_FSM_rev1_2.pdf

10.1.E Howard, Johnson, Martin Graham, "High-Speed digital design", Prentice-Hall, 1993.

10.1.F R. Acosta, G. Eirea, S. Louro, "LaTela Video", proyecto de graduación IIE, 1997.

10.1.G D. Ferrer, R. Gonzalez, R. Fleitas, "Neuro FPGA", proyecto de graduación, IIE, 2003.

10.1.H J. P. Oliver, A. Fonseca de Oliveira, J. Pérez, R. Canetti, "Síntesis Hardware de Redes ALN para Aplicaciones en Control", VIII RPIC99, Mar del Plata, Argentina, 23 al 25 de setiembre de 1999.

10.1.I PCI SIG, "PCI Local Bus Specification. Revision 2.2", 1998. http://www.pcisig.com.

10.1.J Placa RIPP10 Altera en "Altera Programmable Hardware Development Program", http://www.altera.com/html/programs/phd.html.

10.1.K J.H. Luján, P. Mazzara, J.P. Oliver, F. Silveira. "Registrador de Perturbaciones Para la Red Eléctrica", II Encuentro de Especialistas en Potencia Instrumentación y

Medidas, Capítulo de Potencia e Instrumentación y Medidas del IEEE Uruguay, setiembre 1991

10.1.L J. Saporiti, A. Villavedra, "Controlador de SDRAM", proyecto de fin de curso de Diseño Lógico II, 2003, http://iie.fing.edu.uy/ense/asign/dlp/proyectos/2003/sdram/index.htm.

10.1.M WISHBONE System-on-Chip Interconnection Architecture for Portable IP Cores Revision: B.3, 2002

10.1.N W.Bishop, "ARC-PCI Development Environment", 2002, http://www.pads.uwaterloo.ca/~wdbishop/arc-pci/index.html.

10.2. Bibliografía

- Alessandro Rubini, Johathan Corbet, "Linux Device Drivers - Second Edition", O'Reilly, Junio 2001.
- Intel, "Techniques for increasing PCI Performance - Application Note AP-666", Febrero 1999.
- Intel, "Intel Architecture Software Developer's Manual - Vol.3 System Programming Guide", 1997.
- Altera, "AN 116: Configuring SRAM-Based LUT Devices", http://www.altera.com/literature/an/an116.pdf.
- Peter J. Ashenden, "The Student's Guide to VHDL", Morgan Kauffman Publishers, 1998.
- Altera, Home Page, http://www.altera.com
- Opencores, Home Page, http://www.opencores.org
- Hojas de datos de los fabricantes de los componentes

10.3. Glosario

AHDL
Lenguaje de descripción de circuitos. Permite especificar formalmente un diseño. Este lenguaje fue desarrollado por ALTERA y es utilizado solo por los sitetizadores de la compañia.

ALTERA
Compañia fabricante de FPGAs, herramientas de sintesis y simulación para FPGAs, placas de desarrollo y cores IP.

ARC-PCI
Placa PCI con lógica reconfigurable fabricada por ALTERA. Permite diseñar y probar diseños para el bus PCI de un PC, su arquitectura permite reconfigurarla estando conectada al bus PCI.

BAR
Base Address Register (BAR). Registros de los dispositivos PCI que definen la dirección de comienzo, tamaño y tipo de memoria utilizada. Al prender el PC se hacen ciclos de configuración mediante los cuales se determina la cantidad de memoria requerida por cada dispositivo PCI y se le asigna una direción de comienzo.

Core o Core IP
Los diseños especificados utilizando un lenguaje de descripción de hardware (AHDL, VHDL, Verilog) se denominan cores IP, donde core hace referencia a que es un módulo que tiene una funcionalidad y una interfaz definida y puede ser integrado en otros diseños. IP hace referencia a propiedad intelectual (Intellectual Property), ya que, al especificar un diseño en un lenguaje de descripción de hardware, no se esta plasmando físicamente un diseño, sino que se esta creando su especificación funcional en un lenguaje de descripción de hardware dado. La descripción puede entonces ser sintetizada (llevarla a un circuito eléctrico) en la tecnología deseada.

DOUBLE WORD o DWORD
Dato de 32 bits de ancho o 4 bytes.

Driver
Software controlador de dispositivos que esconde la complejidad y los detalles de cómo funciona su correspondiente dispositivo. Permite acceder a los recursos

brindados por el dispositivo utilizando interfaces bien definidos por el sistema operativo.

IIE
Instituto de Ingeniería Eléctrica

IIE-PCI
Placa PCI con lógica reconfigurable diseñada y fabricada en este proyecto. Permite diseñar y probar diseños para el bus PCI de un PC.

Master PCI
El bus PCI se basa en transacciones punto a punto entre dispositivos. El dispositivo que comienza la transacción se llama Master PCI

PCI
Peripheral Component Interconnect(PCI). Especificación que define al bus PCI. Define las interconecciones y los protocolos de transferencia utilizados por placas que se conecten a al bus ubicado en la placa madre.

Testbench
Diseño hardware especificado en algún lenguaje de descripción de circuitos utilizado para testear cores IP.

Target PCI
El bus PCI se basa en transacciones punto a punto entre dispositivos. El dispositivo que acepta una transacción se llama Target PCI.

VHDL
Lenguaje estándar de descripción de circuitos. Permite especificar formalmente un diseño.

WISHBONE
Especificación que define al bus WISHBONE. Define las interconecciones y los protocolos de transferencia utilizados por cores IP de un mismo integrado.

WORD
Dato de 16 bits de ancho o 2 bytes.

XILINX

Compañia fabricante de FPGAs, herramientas de sintesis y simulación para FPGAs, placas de desarrollo y cores IP.

11. Apéndices

11.1. Documentos

11.1.1. Manual de usuario de Placa IIE-PCI

11.1.2. Manual de usuario de core PCI

11.1.3. Manual de usuario del driver PCI genérico

11.1.4. Esquemáticos de la placa

11.1.5. Lista de Materiales y costos

11.1.6. Pinout del FPGA

11.1.7. Interfaz Wishbone