CAPÍTULO 1

Introducción.

Este capítulo dará una visión general del proyecto “APLICACIÓN DE LA TRANSFERENCIA DMA AL DESARROLLO DE UNA PLATAFORMA RECONFIGURABLE PARA PROCESAMIENTO DE IMÁGENES”. Este proyecto esta definido como la continuación e integración de varios proyectos, destacando el proyecto “Diseño de una plataforma para prototipos de tarjeta PCI” de Javier Rincón [Rincón 01].

La introducción se separa en tres partes, la primera de ellas trata de adentrar al lector en lo que va a ser el proyecto desarrollados mostrando los objetivos que se tomaron inicialmente y dando una idea general de su contenido. Posteriormente se verá una breve explicación de los pasos tomados para el desarrollo del sistema y de la plataforma que lo contiene. Finalmente se mostrará el contenido y la estructura del proyecto con un breve resumen de cada capitulo del que consta el proyecto para que el lector se sitúe en los diferentes pasos que va a encontrar a lo largo de este libro.

1.1 OBJETIVOS

La tarjeta PCI Proto-Lab/PLX es una plataforma para realizar prototipos de periféricos conectados al PC a través del Bus PCI y aprovechar las altas prestaciones que este bus presenta.

Este proyecto parte como continuación de otro, el proyecto [Rincón 01], que a proporcionado a este proyecto una comunicación física entre las diferentes placas de las consta la plataforma y un inicio de interfaz para transferencias maestro/esclavo.

Lo que se pretende con este proyecto es disponer de una plataforma para procesamiento de imágenes y visión artificial gracias a las altas tasas de transferencias del DMA y del bus PCI. Esta plataforma, al ser reconfigurable, puede realizar cualquier procesado a las imágenes, así como conectar otros periféricos a la misma. Para ello se dispone de tarjeta XS-40 con una FPGA XC-4010XL de Xilinx que será la que contendrá las diferentes aplicaciones hardware de tratamiento de la imagen.

Se pretende que esta plataforma pueda ser utilizada para la realización de cualquier aplicación sin que para futuros trabajos se tenga que estudiar de nuevo toda la comunicación entre las tarjetas ni la realización de nuevos programas que controlen el protocolo para realizar transferencias. Por lo que se necesita una plataforma que de manera transparente realice todas estas tareas y que pueda ser utilizada en cualquier proyecto sin necesidad de conocer su funcionamiento. En esto consten los objetivos principales de este proyecto

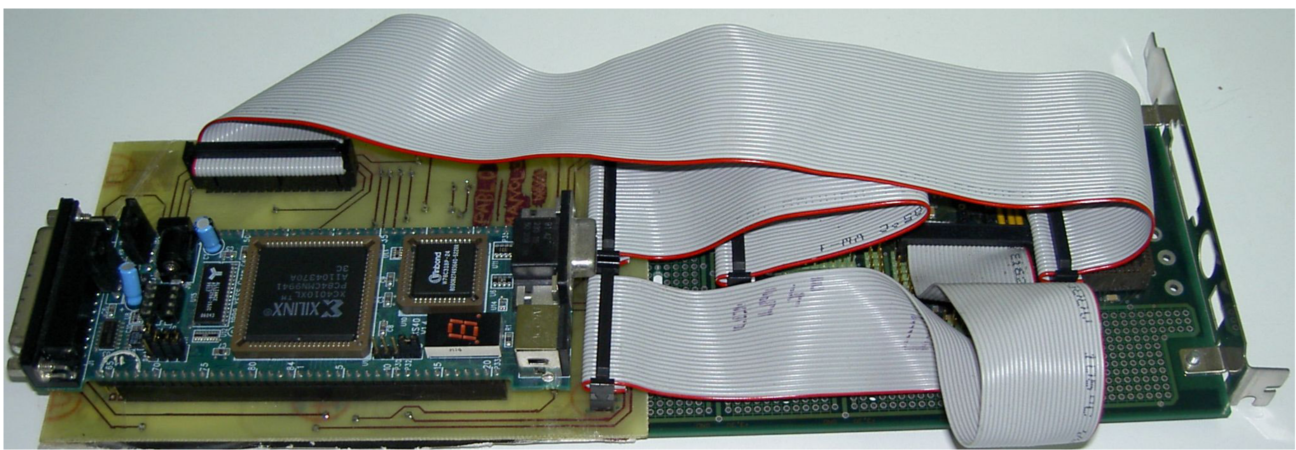


Figura 1.1: Hardware del sistema montado

La aplicación de procesado para las imágenes que se ha realizado en este proyecto es una simple binarización de la misma, pero la plataforma quedará preparada para realizar cualquier tipo de procesado que precise de altas velocidades. Además se ha pretendido que este proyecto sea fácilmente ampliable con nuevas tarjetas o nuevos modos de programación para transferencias más complejas.

Se han planteado los siguientes objetivos para poder disponer de dicha plataforma:

- Desarrollo de una API para la programación de aplicaciones que funcionen con la tarjeta Plx. Ésta API debe ser de muy sencilla utilización y entendimiento, rápida, fiable y fácilmente ampliable para futuros trabajos.

- Desarrollo de una interfaz de comunicación entre la tarjeta Plx y una FPGA genérica en VHDL que pueda realizar transferencias tanto en maestro/esclavo como en DMA indistintamente. La interfaz debe tener la capacidad de funcionar con diferentes tarjetas y diferentes anchos bus local para futuras ampliaciones del hardware. Esta interfaz se debe diseñar en VHDL para cumplir con los estándares actuales.

- Estudio detallado del driver de comunicaciones asociado al controlador PLX9054.

Para conseguir este objetivo deberemos de seguir las siguientes fases:

- Estudio de la especificación PCI rev.2.1. para comprender su protocolo de funcionamiento, así como los modos de funcionamiento del controlador PLX 9054 insertado en la tarjeta de prototipo.

- Estudio del proyecto final de carrera “Diseño de una plataforma para prototipos de tarjeta PCI” de Javier Rincón.

- Estudio del funcionamiento de la tarjeta de prototipos, que comunicará el Bus PCI, con la tarjeta que realizará la función esclava, a través del Bus PCI

- Estudio del modo de transferencia DMA y su utilización en la tarjeta PCI Proto-Lab/PLX.

- Aplicar las nuevas interfaces y modos de transferencia a la aplicación de binarización del proyecto [Rincón 01].

Se pretende que este proyecto sea el portal para nuevos proyectos más complejos siendo posible cambiar el software y el hardware, del mismo, fácilmente.

1.2 DESARROLLO DEL PROYECTO.

Este apartado pretende describir brevemente, el proceso de ejecución del proyecto para poder tener una visión global antes de emprender una lectura del resto del mismo.

En la figura 1.1, se muestra el esquema general de la plataforma de prototipado rápido, donde aparecen los tres principales elementos que integran nuestra aplicación, estos son, el Bus PCI contenido en el ordenador, la tarjeta de prototipo de Bus PCI y la tarjeta XS40.

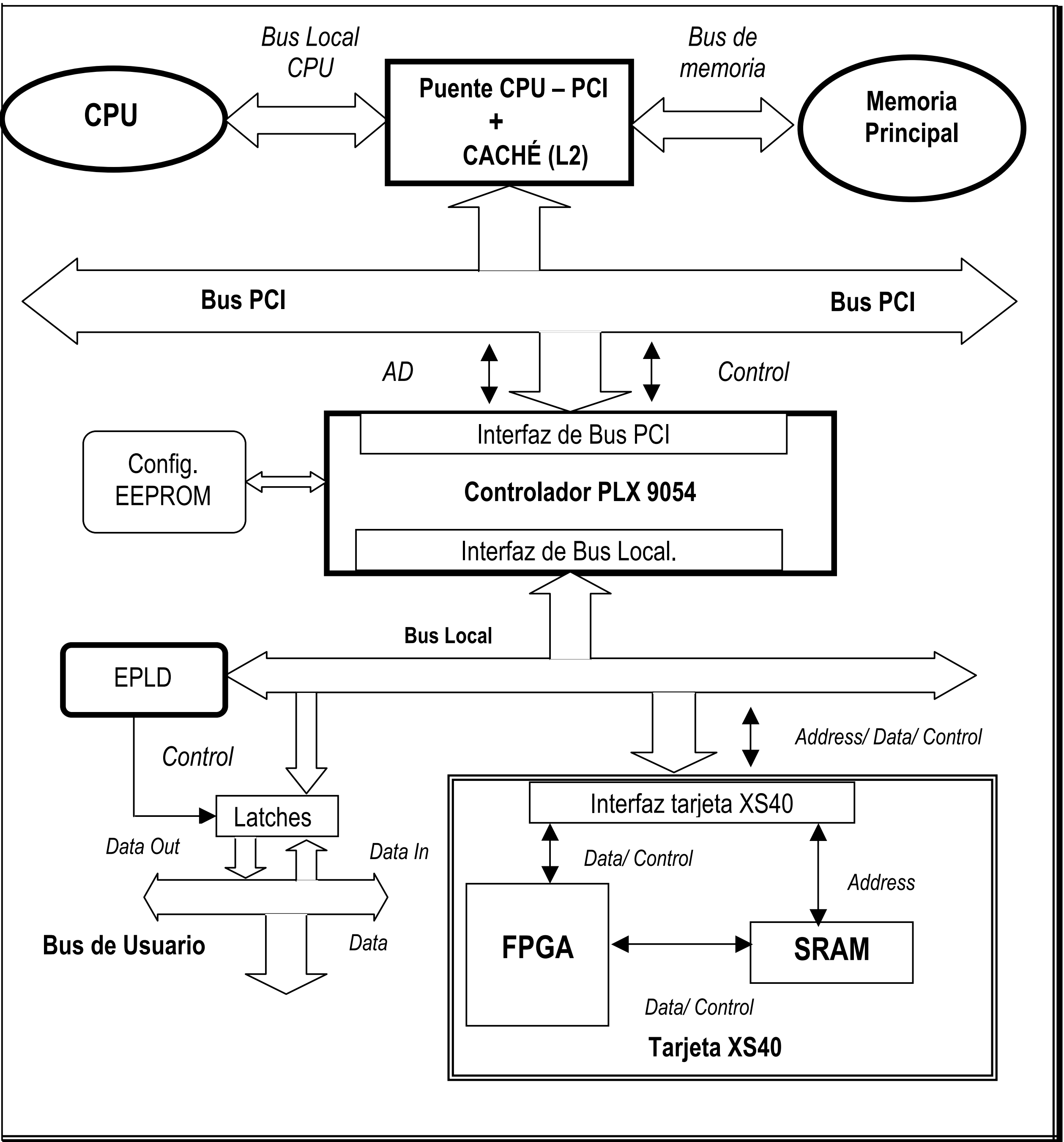


Figura 1.1 Esquema general de la plataforma de prototipado rápido.

La primera fase del proyecto contendrá la descripción del soporte físico (hardware). Esta fase estará dividida en tres partes bien diferenciadas. La primera describe el funcionamiento del Bus PCI y del modo de transferencia DMA. Una segunda parte se dedicará al estudio y descripción de las prestaciones de la tarjeta que interactúa con este bus, dedicándole especial interés al dispositivo PLX9054 que realiza el protocolo de comunicación. Terminando con una descripción del funcionamiento de la tarjeta XS40, que conectaremos a la tarjeta de prototipo para realizar la aplicación de demostración de funcionamiento.

La segunda parte del proyecto se dedicará a la descripción del soporte lógico (software) necesario para realizar la interacción de todos los elementos anteriores. Esta descripción esta compuesta por dos partes. Una de ellas comprende la explicación de los programas realizados tanto en el PC como en la FPGA. La otra, es una muestra de varios programas de prueba y sus resultados así como los resultados de la aplicación final.

1.3 ESTRUCTURA DEL PROYECTO

Cumpliendo con los propósitos enumerados anteriormente, el presente proyecto se encuentra dividido en nueve capítulos y varios anexos, los cuales quedan estructurados como sigue:

Capitulo 1: Introducción.
Este capítulo servirá para dar una visión de conjunto del presente proyecto, mostrando los objetivos perseguidos con el mismo y una breve historia del Bus PCI.

Capitulo 2: El Bus PCI.
Describe le funcionamiento del Bus PCI, presentando los antecedentes a este bus, sus características, las señales que maneja y como se transmiten y las operaciones principales que éste puede realizar. Este capítulo concluye con la organización del espacio de configuración que dará paso a la descripción del controlador de bus, que será objeto de estudio de los siguientes capítulos.

Capitulo 3: DMA.
En él se describe el modo de transferencia DMA, el protocolo de funcionamiento del mismo así como ventajas e inconvenientes, tasas de transferencia, y comparativa con otros modos de transferencia.

Capitulo 4: La tarjeta PCI Proto-Lab/PLX.
Donde se describirá el funcionamiento de la tarjeta de prototipo de Bus PCI, PCI Proto-Lab/PLX, analizando sus principales componentes, de manera que el lector vea la funcionalidad que esta presenta.

Capitulo 5: El controlador PLX. Arquitectura de control.
Este capítulo trata sobre el controlador de Bus PCI, corazón de la tarjeta de prototipo del capitulo anterior. En este capitulo se intentará mostrar al lector los modos de operación y transferencia que posee. También se muestra las diferentes configuraciones que puede contener y como cambiarlas a través de los registros. Se dedicará especial atención a la explicación de la configuración que se encuentra definida por defecto en la misma.

Capitulo 6: La tarjeta XS40 de XESS.
Este capítulo, concluirá con la descripción de los soportes hardware, uso de la aplicación. En éste se estudia la tarjeta que conectaremos a la tarjeta de prototipo, exponiendo su funcionamiento, su estructura y sus métodos de programación, de manera que describa las posibilidades y limitaciones que ésta presenta.

Capitulo 7: Sistema de procesamiento de imágenes basado en bus PCI.
Capitulo donde se presentaran los diseños lógicos realizados para controlar el funcionamiento de los sistemas hardware que componen la aplicación. Los del PC, para realizar la manipulación y transferencia de las imágenes y los realizados para la FPGA, que actuará como esclava de las transferencias y realizará un procesamiento de la imagen enviada.

En este capitulo se encuentra una explicación detallada de las interfaces diseñadas y la explicación de su funcionamiento.

También se expondrá, las características principales a tener en cuenta en el diseño de tarjetas para alta frecuencia y recomendaciones interesantes con el propósito de que sirvan a un futuro diseñador.

Capitulo 8: Pruebas y resultados.
Expondremos la descripción de las principales pruebas realizadas, para llegar a la aplicación final del proyecto, y un estudio de tiempo realizado.

Anexo A Manuales de usuario.
Compuesto por los manuales de usuario de la aplicación realizada para el proyecto, y del manual para usar la aplicación PLXMon que acompaña al kit de desarrollo PLC SDK.

Anexo B Instalación de la tarjeta PCI Proto-Lab/PLX.
Se guiará al lector por los pasos necesarios para instalar la tarjeta de prototipo PCI Proto-Lab/PLX.

Anexo C Códigos en Visual C++, VHDL y Esquemas.
Código de la API diseñada y código fuente del programa realizado para el PC en el entorno de programación Visual C++. También incluye el código VHDL de la interfaz de comunicación para la FPGA y la versión anterior basada en esquemáticos. También incluye el programa implementado en la CPLD por el fabricante de la tarjeta.

Anexo D Figuras y tablas más importantes
Tablas y esquemas más importantes del hardware para facilitar su utilización en futuros proyectos.

CAPÍTULO 2

El Bus PCI.

El bus PCI[1], fue diseñado por Intel en 1992 con el objetivo de superar las limitaciones de velocidad y anchura de datos impuestas por el bus ISA. Se trata de un bus multiplataforma con 32 bits de anchura, capaz de funcionar a una frecuencia máxima de 33 MHz, lo que proporciona una velocidad de transferencia teórica de 132 Mbytes por segundo.

Apareció de un modo un tanto discreto, cuando sus competidores más directos ya se habían asentado firmemente en el mercado. Pero poco a poco ha ido ganando terreno, hasta el punto de que la gran mayoría de los PC's de altas prestaciones que hoy se venden la incluyen en su placa base.

La llegada de Windows al mundo de los PC compatibles trajo consigo la aparición de aplicaciones más atractivas y sencillas de manejar, en las que ya no era necesario tener amplios conocimientos de informática, sino que bastaba con una cierta destreza al utilizar el ratón.

Pero lo que constituía su mayor atractivo, su entorno gráfico, se convirtió también en su mayor lastre. Los ordenadores tradicionales, limitados por el bus ISA de 8 MHz y 16 bits (con un ancho de banda máximo de 5 Mbytes por segundo), eran incapaces de mover de modo satisfactorio las enormes cantidades de datos requeridas por Windows y las aplicaciones nacidas para este entorno.

---

[1] Peripheral Component Interconnect.

El aumento en prestaciones y funcionalidad ha permitido, así mismo, la aparición de software mucho más complejo y sofisticado: cálculo intensivo, gráficos complejos, proceso transaccional en línea, redes locales o vídeo en tiempo real, aplicaciones, todas ellas, que demandan el proceso y traslado a gran velocidad de grandes volúmenes de datos entre la CPU y los periféricos.

Por ejemplo, una pantalla de 1.280 por 1.024 en la que se usen 24 bits por *píxel* (lo que se conoce como *True Color*) requiere la nada despreciable cantidad de 4 Mbytes de datos. Ni siquiera las arquitecturas EISA o MCA/2 (con anchos de banda de 33 y 40 Mbytes por segundo, respectivamente) son capaces de gestionar adecuadamente estas cifras.

El bus PCI se encuentra conectado a la CPU y a la memoria RAM a través de un controlador que admite, debido a la velocidad de funcionamiento y a las características eléctricas del propio bus, un máximo de cuatro dispositivos PCI. Este límite se puede aumentar mediante la incorporación de dispositivos especiales denominados puentes PCI-PCI[1], que actúan como controladores de buses PCI subordinados. De esta manera, un sistema PCI puede tener hasta un máximo de 255 buses PCI organizados jerárquicamente. El bus principal, o bus 0, se suele denominar bus raíz([2]).

Los actuales microprocesadores de 32 bits, funcionando a velocidades iguales o superiores a los 33 MHz, se ven así frenados porque los discos duros, las tarjetas de vídeo y los otros periféricos envían o reciben datos a través de un camino estrecho, lento e ineficaz. De ahí que los fabricantes hayan optado por el desarrollo de nuevas soluciones basadas en tecnologías de Bus Local.

2.1 ANTECEDENTES.

En 1985, la compañía IBM sacó el primer ordenador AT. El sistema de bus utilizado en aquel entonces era el bus ISA. Sin embargo, poco después de su introducción en el sistema, quedó rápidamente en evidencia que el bus ISA no sería por mucho tiempo la representación del estado hacia donde se dirigirían los desarrollos técnicos.
Estaba claro que los procesadores 286 se comportaban de forma lenta debido a este bus, por ejemplo, cuando trabajaban con tarjetas gráficas o con tecnologías de redes de alta velocidad. Esta gran pérdida en las prestaciones era debida a la compatibilidad hacia atrás que se debía mantener con los viejos sistemas de 8 bits. De forma más precisa, la baja velocidad del bus de reloj de 6 MHz, que posteriormente fue incrementada hasta 8 MHz, tampoco ayudaba a solucionar la problemática.

El resultado fue una velocidad teórica de transferencia de datos de 8 MB/s con un ancho de bus de 16 bits. Sin embargo, en la práctica real, la velocidad máxima de transferencia de datos que podía alcanzarse estaba entre 4 y 6 MBytes/ seg.

2.1.1 EL BUS MCA.

Este problema fue el incentivo para que IBM desarrollara un nuevo sistema de bus, adecuado para funcionar con 32 bits. En 1987, se introdujo en el mercado la Arquitectura de Micro Canal MCA[3]. Este sistema tenía un bus de datos y de direcciones de 32 bits con capacidad de "multimaestro", y prometía una velocidad de transferencia de datos de hasta 16 MB/s. Por desgracia, los nuevos

[1] PCI-PCI Bridge.
[2] root bus.
[3] del inglés Micro Channel Architecture.

ordenadores no disponían ranuras de conexión de cualquier tipo del antiguo modelo ISA, y como consecuencia este bus no tuvo éxito comercial.

2.1.2 El bus EISA.

La experiencia con el fallido bus MCA jugó un papel importante en el desarrollo del bus Extended Industry Standard Architecture, EISA([1]). El bus EISA trabaja con un reloj de 8 MHz y con la tecnología de 32 bits. Con un ancho de bus de 32 bits, la velocidad de transferencia de datos resultante es de 16 MB/s en el modo estándar y de 32 MB/s en el modo “ráfaga”. Posteriormente se añadieron otros dos modos de “ráfaga” (el EMB – 66 y el EMB – 133), los cuales permitirían hasta velocidades de transferencia de datos de hasta 133 MB/s. Sin embargo, este nivel de prestaciones tenía su precio, por lo que tan sólo fue utilizado en sistemas de servidores caros.

2.1.3 EL BUS VESA.

A principios de 1993 el consorcio VESA, integrado por más de 120 compañías, ofrecía una especificación que daba lugar a un estándar de facto, en el que los fabricantes de periféricos podían apoyarse de cara al diseño de nuevas tarjetas. Visualmente, un conector tipo VLB es una especie de híbrido, ya que se ha partido de un conector estándar ISA y se ha añadido otra parte de 16 bits basada en un conector MCA.

Aunque puede parecer extraña, dicha decisión tiene su lógica, ya que uno de los objetivos de la especificación VLB consistía en aprovechar al máximo los componentes ya existentes. De este modo, la diferencia de precio entre placas base con y sin Bus Local no resulta demasiado elevada.

Sin embargo, el Bus Local VESA presenta algunas limitaciones de tipo técnico. Por ejemplo, no existen placas base con más de tres ranuras de bus local, ya que la CPU es incapaz de manejar más de tres cargas externas salvo que funcione a una velocidad muy baja. De hecho, para velocidades superiores a los 33 MHz resulta incluso cuestionable la eficacia de la tercera ranura. Para mitigar un tanto el problema, los fabricantes recurren a instalar *buffers*, unos componentes electrónicos que permiten «amortiguar» la carga de la CPU.

Aunque el ancho de banda del VLB es de 132 Mbytes por segundo, dicho bus no soporta las técnicas de aceleración de gráficos, como la escritura en modo ráfaga. Y es que un buen diseño de Bus Local debe ofrecer mucho más que un elevado ancho de banda, permitiendo que las funciones periféricas obtengan todo el rendimiento posible de la potencia de proceso disponible.

Las otras críticas a la especificación VESA son su falta de previsión de futuro (no se pensó en las posibilidades del *Plug & Play*) y su escasa definición. En efecto, se trata de un estándar algo incompleto, puesto que tan sólo define la forma de conexión, no la implementación en la placa base (detalle éste que se ha dejado al libre albedrío de cada diseñador en concreto). Así que, al menos en teoría, no es posible descartar que se pueda producir alguna incompatibilidad.

2.1.4 EL BUS VLB.

Varios fabricantes de equipos periféricos, que pertenecían a un mercado nervioso y en constante movimiento, buscaron un sistema de bus que fuera capaz de proporcionar unas altas prestaciones.

[1] Arquitectura Estándar Extendida de la Industria.

Después de unos pocos intentos fallidos, el resultado fue el bus VLB[2]. Éste también era un bus con 32 bits de ancho y una velocidad de reloj comprendida entre 25 y 60 MHz. Así, aparecieron en el mercado varias versiones de este bus, denominadas VLB-1.0 8(con velocidad de reloj comprendida entre 25 y 40 MHz), VLB-2.0 (con velocidad de reloj comprendida entre 25 y 50 MHz) y VLB-64 bits(con velocidad de reloj comprendida entre 25 y 60 MHz). El principal problema de esta tecnología de bus era su inconformidad con varias especificaciones. La especificación VLB-1.0 sólo permitía utilizar dos ranuras para periféricos, a una velocidad máxima de reloj de CPU de 40 MHz. Sin embargo, algunos fabricantes construyeron sistemas que soportaban hasta tres ranuras de expansión y funcionaban a una velocidad de reloj de 50 MHz. Esto produjo problemas significativos con la estabilidad de sistemas individuales.

2.1.5 EL BUS PCI.

El diseño inicial del bus PCI[1], fue realizado por Intel en 1991. Las principales razones para el desarrollo de este nuevo bus fueron:

- Conseguir una velocidad de transferencia de datos mucho más elevada que la obtenida con el bus ISA de 16 bits.
- Lograr una mejor compatibilidad electromagnética (EMC) que la obtenida con los sistemas anteriores.
- Asegurar una continuidad futura con las siguientes generaciones de procesadores.

En 1992 el diseño fue completado y el primer bus PCI pudo ver la luz. En el curso de los posteriores desarrollos de este sistema de bus, varias especificaciones salieron a la luz(versiones 1.0, 2.0, 2.1 y 2.2, esta última la versión actual). El resultado de los requerimientos demandados definieron en 1991 un sistema de bus de 32 bits que transfería datos y direcciones en un modo multiplexado en el tiempo y que puede ejecutar ciclos de "ráfagas" de longitud variable. Intel consiguió ganar la aceptación total del mercado para un completo, realista y bien estructurado juego de definiciones de todas las señales importantes del bus, así como de las bases para el funcionamiento del bus PCI. Un objetivo en el desarrollo de este sistema de bus abierto, desprotegido y jerárquico, fue evitar la repetición de errores que se habían cometido en los sistemas de bus anteriores.

2.2 BUS PCI VS. BUS ISA.

El estándar PnP para el bus ISA, publicado inicialmente en mayo de 1994 y modificado en diciembre del mismo año, define las características y el comportamiento de las tarjetas PnP para el bus ISA. De todas las arquitecturas de bus, ISA es la más complicada de tratar, puesto que las tarjetas PnP han de convivir con tarjetas antiguas (denominadas *legacy ISA cards*) que hacen uso arbitrario de los recursos del sistema.

Esto significa, por ejemplo, que no es posible especificar de forma fija ningún puerto de lectura para una tarjeta PnP ISA, ya que el mismo puerto puede ser usado por una ISA antigua y causar una colisión en el bus de datos durante una lectura. Otro problema es que las ranuras ISA son indistinguibles unas de otras (salvo la famosa ranura 8 del XT, ya desaparecido), con lo cual se ha de establecer un protocolo especial de «selección de tarjeta» para comunicarse con una tarjeta PnP ISA específica.

---

[2] VESA Local Bus, es decir, el bus local de VESA.

[1] Peripheral Component Interfaz, es decir, Interfaz para componentes Periféricos.

Para entender el protocolo de configuración de las tarjetas que se describe a continuación, es muy importante recordar que las tarjetas están conectadas en paralelo y son equivalentes. Esto significa que si leemos de un puerto común, todas las tarjetas tratarán de responder simultáneamente, por lo que, a menos que los datos que transmitan por el bus sean iguales, obtendremos basura.

En principio, cuando se recibe una señal RSTDRV (reset), las tarjetas ISA PnP pasan a su estado inicial, denominado *Wait for Key*, o en espera de clave. Su lógica PnP se encuentra desactivada y se comportan como tarjetas ISA ordinarias con unos parámetros de fábrica en cuanto a direcciones de E/S, canales DMA, IRQs, etc. se refiere.

Tipo de Bus	Frecuencia de reloj (MHz)	Ciclos de reloj por transfer.	Tamaño de dato (bits)	Máx.Veloc. transfer. (Mbytes/s)	Ventajas	Inconvenientes
ISA	8	2	16	8	Bajo coste, compatibilidad, uso extendido	Lento
Microcanal	10	1	32	40	Más rápido que ISA	Incompatibilidad, coste, difícil de encontrar
PCI	33	1	32	132	Rápido y potente, Plug & Play	Incompatible con sistemas anteriores, puede resultar caro.

Tabla 2.1 Características de los principales buses de expansión.

A diferencia de los dispositivos ISA, las tarjetas PCI pueden decodificar un espectro completo de direcciones de entrada/ salida de 32 bits, desde la 00000000h hasta la FFFFFFFFh. Esto permite al sistema operativo asignar direcciones de E/S que no tengan conflicto con ninguna tarjeta ISA. Adicionalmente, muchas placas base PCI proporcionan un controlador con dirección base de E/S programable. Si dicha dirección base es, por ejemplo C000h, el controlador del bus PCI no realizará ninguna operación de E/S por debajo de esta dirección.

Otra diferencia más con el bus ISA es que los dispositivos PCI no pueden utilizar canales DMA. Por eso, en algunos sistemas existen puentes PCI-ISA que permiten la existencia de un bus ISA detrás del bus PCI principal. Los dispositivos ISA sobre este bus pueden, por supuesto, utilizar canales DMA, pero dichas peticiones son «rutadas» a través del puente PCI-ISA, al margen del bus PCI.
De hecho, es imposible para la BIOS o para cualquier otro software distinguir entre un bus ISA situado detrás de un puente PCI-ISA y un bus ISA conectado directamente al microprocesador a través de un controlador de bus estándar.

Los dispositivos PCI pueden hacer uso de un máximo de cuatro líneas de interrupción, llamadas INTA#, INTB#, INTC# e INTD#. Puesto que el bus PCI es independiente de la plataforma, no se puede asumir ninguna correspondencia entre estas líneas y las líneas IRQ presentes en un entorno PC. Los diseñadores de las placas base son los responsables de proporcionar el «rutado» de cuatro líneas IRQ hacia el bus PCI. En algunos casos (como veremos más adelante), dicha correspondencia es programable, es decir, el sistema operativo puede solicitar que la línea INTA#

del *slot*[1] PCI 3 sea conectada con la IRQ11, por ejemplo. No obstante, muchos de los sistemas PCI antiguos son incapaces de proporcionar o alterar la información del «rutado» de las líneas de interrupción.

Este último punto suele dar lugar a confusiones, ya que algunos autores consideran que los dispositivos PCI no son dispositivos PnP, debido a la incapacidad de modificar la configuración de interrupciones. Esto no es correcto. En la descripción de los dispositivos ISA PnP, existen descriptores de recursos que indican recursos fijos no modificables. Esto significa que una tarjeta ISA PnP puede, paradójicamente, no tener ningún recurso configurable ni las direcciones de E/S, ni las líneas de interrupción, ni las ventanas de memoria y, no obstante, ser totalmente compatible según el estándar PnP para ISA.

El punto clave a recordar para evitar estas confusiones es que el elemento principal de la arquitectura PnP es no tanto la configurabilidad de las tarjetas, que sin duda es importante, sino la posibilidad por parte del S.O. de recabar información sobre los requisitos de todas y cada una de las tarjetas instaladas en el sistema.

Finalmente, un dispositivo PCI dispone de un conjunto de registros base de 32 bits, con los cuales solicitar direcciones de memoria o de E/S. El dispositivo puede solicitar direcciones de memoria de 32 o 64 bits de anchura.

2.3 CARACTERÍSTICAS DEL BUS PCI.

El Bus PCI está formado por tres módulos diferentes:

- La Unidad de Camino de Datos.
- La Interfaz de Expansión de Bus.
- El “Host-Bridge” (o “puente receptor”), con un controlador con memoria caché DRAM.

La Unidad de Camino de Datos se usa para configurar un enlace de 32 bits hacia los componentes individuales que existen en el sistema. Se pueden conectar otros sistemas de bus a través del Interfaz de Expansión de Bus (como por ejemplo, ISA o AGP).

La Interfaz de Expansión de Bus y los puentes del sistema adicionales, proporcionan la compatibilidad hacia atrás con los sistemas que integran el bus ISA, por ejemplo, o una nueva tecnología como puede ser AMR.

El “Host Bridge” es el componente central de un sistema con Bus PCI. Éste componente puede emplearse para crear un enlace entre el Bus PCI y la CPU del sistema. Además, también convierte los ciclos de reloj PCI en ciclos CPU, y viceversa. También consigue que el procesador del Bus PCI sea independiente, en contraste con otros sistemas de bus, ya que todo lo que se necesita para conectar el bus PCI a un procesador particular del tipo Intel, Alpha, AMD, etc., es utilizar diferentes “Host Bridge”.
Esta característica hace que los sistemas con Bus PCI sean relativamente independientes de las futuras generaciones de procesadores, al mismo tiempo que lo hacen ampliable.

[1] Ranura donde introducimos la tarjeta.

La figura 2.1 muestra un esquema de la arquitectura que presenta varios dispositivos PCI conectados a un PC.

En esta arquitectura, un elemento muy importante es el puente CPU–PCI, que incorpora la caché de nivel dos (L2) para que el procesador se comunique a través de ella con los dispositivos conectados al Bus PCI y con la memoria principal.

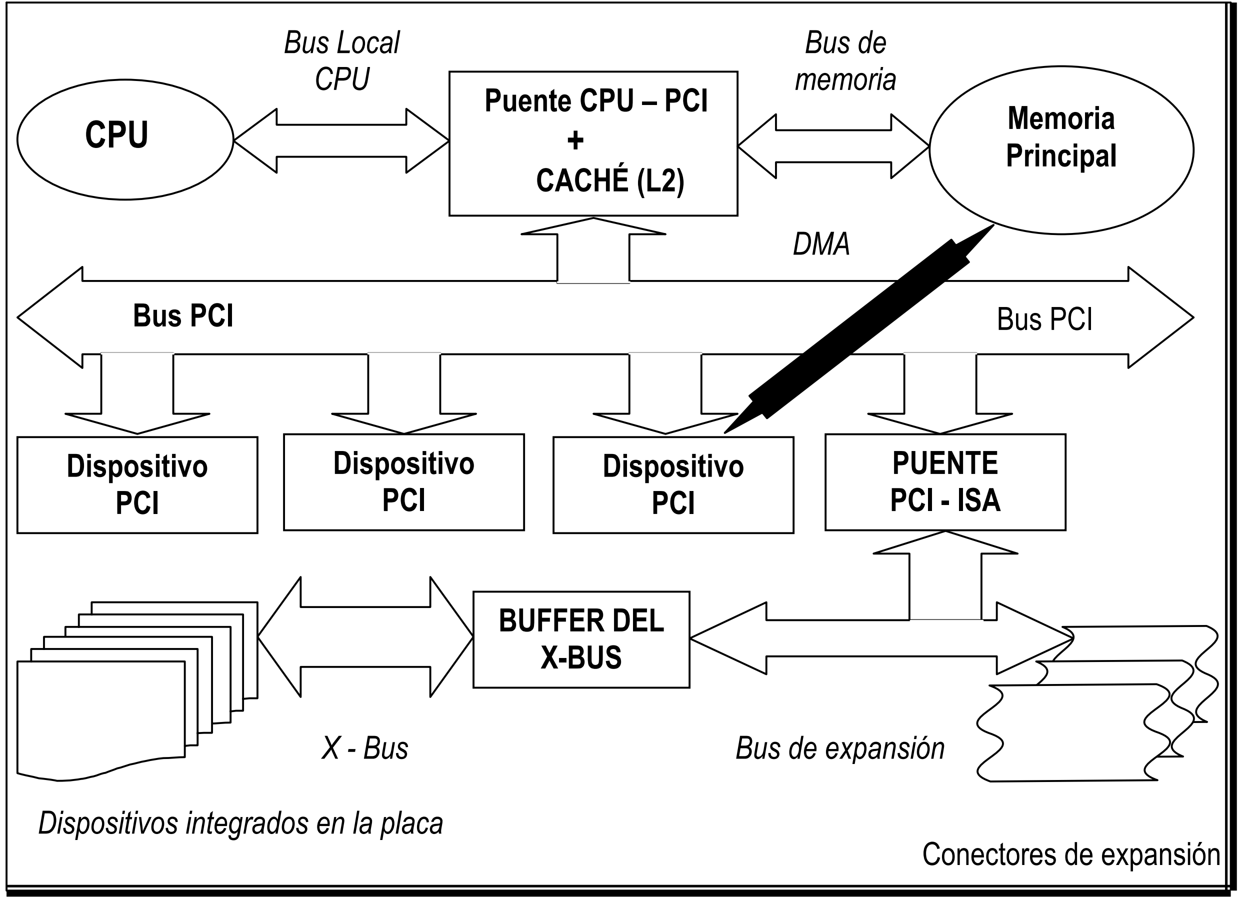


Figura 2.1 Arquitectura de un PC con bus PCI.

Será en esta memoria caché integrada en el puente CPU- PCI, donde se reserve la memoria que necesita cada dispositivo conectado al Bus PCI para su funcionamiento. Este puente CPU-PCI puede transferir datos a la misma velocidad que el bus de datos de la CPU. De esta manera, el puente CPÜ-PCI[1] no introduce ningún retardo ni ralentiza la transición de datos, dependiendo el rendimiento del sistema únicamente de las velocidades del bus de memoria y del bus de expansión. El bus PCI, mediante el puente CPU-PCI optimiza el funcionamiento de los dos elementos más importantes del ordenador: el microprocesador y los dispositivos de E/S, intentando evitar que tengan que esperar para efectuar sus transacciones. Además, este elemento dota de gran modularidad al Bus PCI y garantiza la posibilidad de actualización de sus elementos conforme avanza el desarrollo de las placas madre.

[1] Host Bridge

Como se explicará posteriormente, para que un dispositivo de Bus PCI realice una transferencia, primero debe tomar el control del bus. Para ello activa la señal REQ# indicando que solicita el control del bus. Después de cierto tiempo, el dispositivo sabrá si le ha sido concedido el control del bus consultando la señal GNT#. Mientras la señal GNT# este activada, el dispositivo tendrá el control del bus.
El elemento que se encarga de recoger las peticiones REQ# de los distintos dispositivos y de ceder el bus a uno de ellos en cada momento, es el árbitro PCI, que se encuentra integrado en el puente CPU-PCI. El árbitro PCI debe cumplir unas reglas de operación rigurosamente, pero la implementación final de la forma de elección del dispositivo que controlará el Bus PCI de este elemento, depende exclusivamente del fabricante.

Para explicar el funcionamiento del árbitro PCI, se dedicará un apartado posterior.

2.4. ESPECIFICACIONES DEL BUS PCI.

La especificación del Bus PCI permite un total de diez dispositivos en este bus. Como el "Host Bridge" se comporta como un dispositivo PCI para el bus PCI, tan sólo hay, por lo tanto, provisión para nueve dispositivos. Dichos dispositivos pueden usarse para componentes montados sobre la placa base (SCSI, EIDE, LAN y demás) o para ranuras PCI donde se conectarán las tarjetas PCI.
Además de todo esto, debemos tener en mente que cada ranura PCI exige dos dispositivos, por lo que tan sólo podremos controlar cuatro ranuras de expansión en un Bus PCI. Sin embargo, el número de ranuras puede incrementarse conectando un segundo sistema de bus PCI en la Interfaz de Expansión de Bus.

En total, se pueden ensartar hasta un total de 256 buses conjuntamente, en donde los 255 primeros pueden ser buses PCI y el bus final permitido pueden ser buses ISA, EISA o VL, o incluso un bus MCA. De esta manera, es posible implementar grandes sistemas (figura 2.2)

Otra opción para las ranuras de expansión puede verse en las actuales placas base. En estas placas base se ha conseguido controlar hasta un total de seis ranuras, compartiendo líneas de interrupción(IRQ).

En este tipo de esquema, las ranuras 5 y 6, por ejemplo, comparten una línea de interrupción IRQ. Sin embargo, si se utilizan más de cuatro tarjetas PCI, las prestaciones empeoran. Además, no es posible garantizar, en principio, que todas las combinaciones posibles de tarjetas PCI funcionarán correctamente sin ningún problema.

Se puede conectar una amplia variedad de tarjetas PCI a través de las ranuras. Así, podemos distinguir los siguientes tipos de tarjetas:

- Tarjetas PCI de función única.
- Tarjetas PCI multifunción.
- Tarjetas PCI multidispositivo.

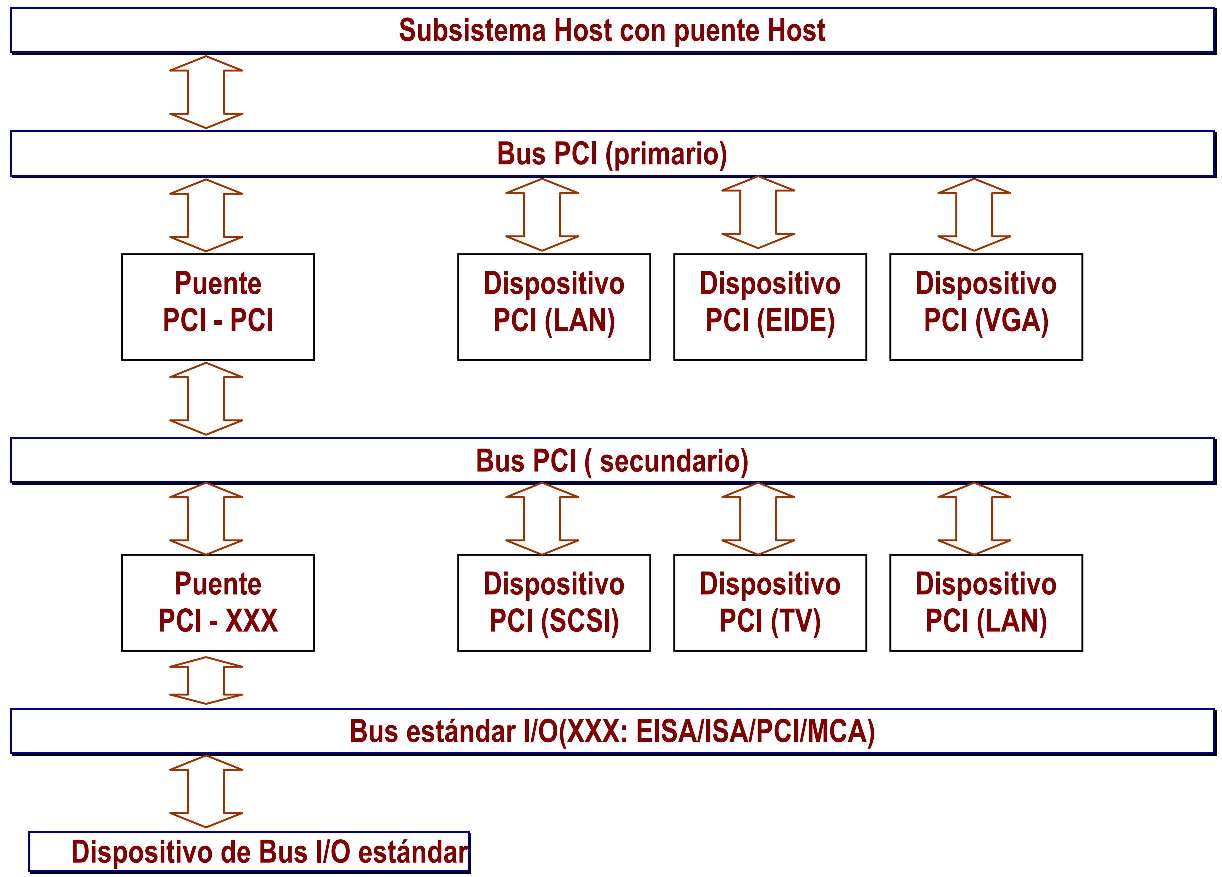


Figura 2.2: Estructura jerárquica de un sistema con bus PCI.

Las tarjetas de función única pueden ser, por ejemplo, tarjetas gráficas o tarjetas controladoras SCSI. Un dispositivo PCI puede tener hasta un total de 8 funciones de E/S diferentes, lo cuál nos llevará a las tarjetas PCI multifunción. Una tarjeta PCI multidispositivo puede estar formada de varios tipos de tarjetas PCI de las mencionadas anteriormente a través de un puente PCI a PCI. Esto significa que las tarjetas PCI multidispositivo representan un sistema completo de bus PCI.

2.5 SEÑALES.

2.5.1 DEFINICIÓN DE SEÑALES.

Conceptos:

- **Initiator o Master:** Controlador que es capaz de iniciar una transferencia.
- **Target o Slave:** Controlador que no es capaz de iniciar una transferencia.
- **Data Phase:** En una transferencia, es el periodo en el cuál se transmiten datos entre el maestro y el esclavo. Los bytes activos en una fase de datos son determinados por C/BE#. La duración de una fase de datos vendrá determinada por el maestro o por el esclavo.

- **Address Phase:** en una transferencia, es el periodo en el cuál se pone la dirección y el comando en el bus AD y C/BE respectivamente. Su duración es de un ciclo de reloj.

Se requieren un mínimo de 47 señales para un esclavo y 49 para un maestro, todas estas señales se muestran en la figura 2.3 de la página siguiente.

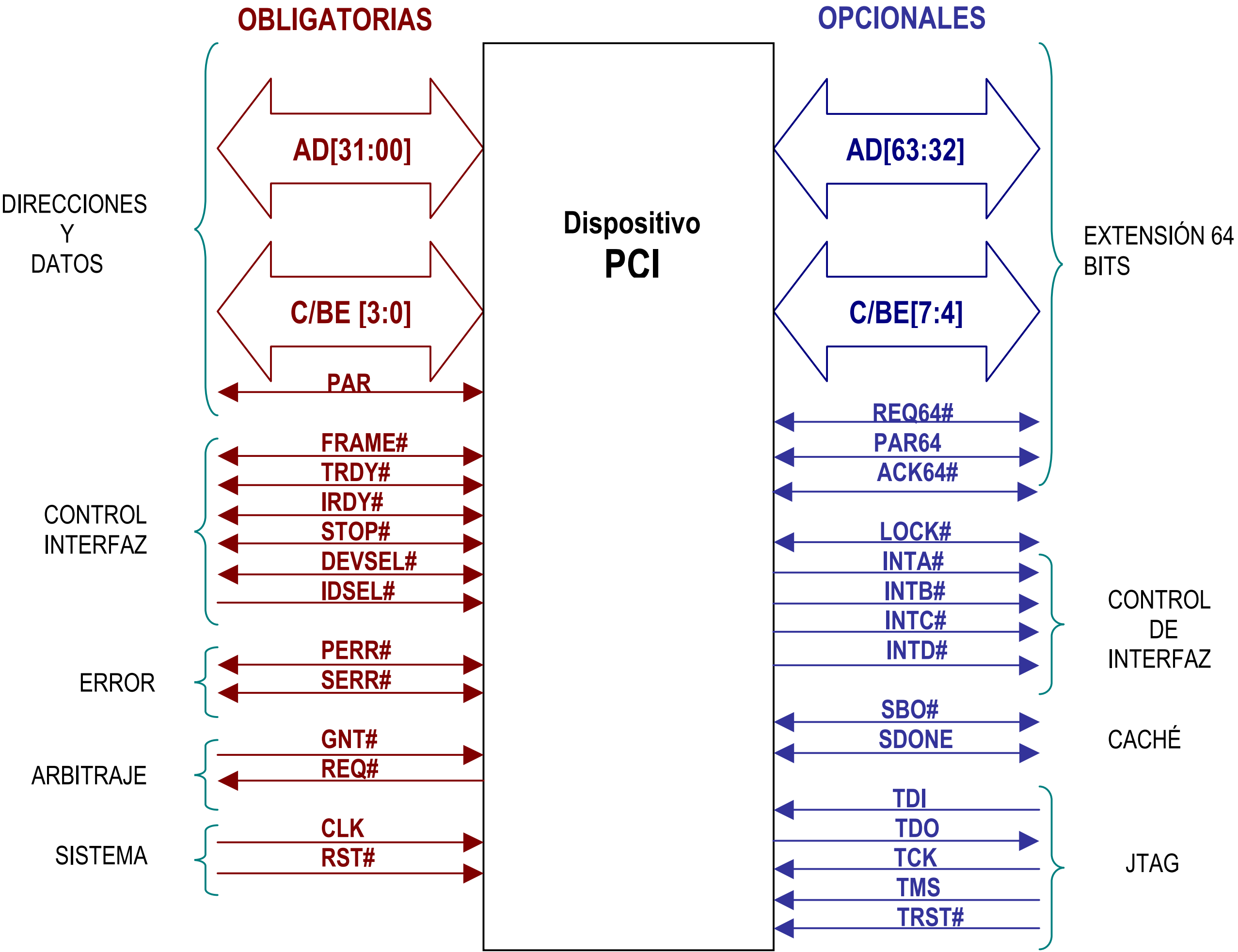


Figura 2.3 Señales agrupadas por funciones.

2.5.2 TIPOS DE SEÑALES.

- **in:** entrada estándar.
- **out :** salida tótem-pole.
- **t/s:** entrada/ salida tri-estado bidireccionales.
- **s/t/s:** tri- estado mantenido. El controlador que activa una señal de este tipo ha de ponerla a nivel alto como mínimo un ciclo de reloj antes de dejarlo en alta impedancia. Otro controlador no puede activarla como mínimo hasta un ciclo de reloj después del estado de alta impedancia. Es necesario una resistencia de pull- up.

- **o/d: drenador abierto.** Es necesario pull-up.

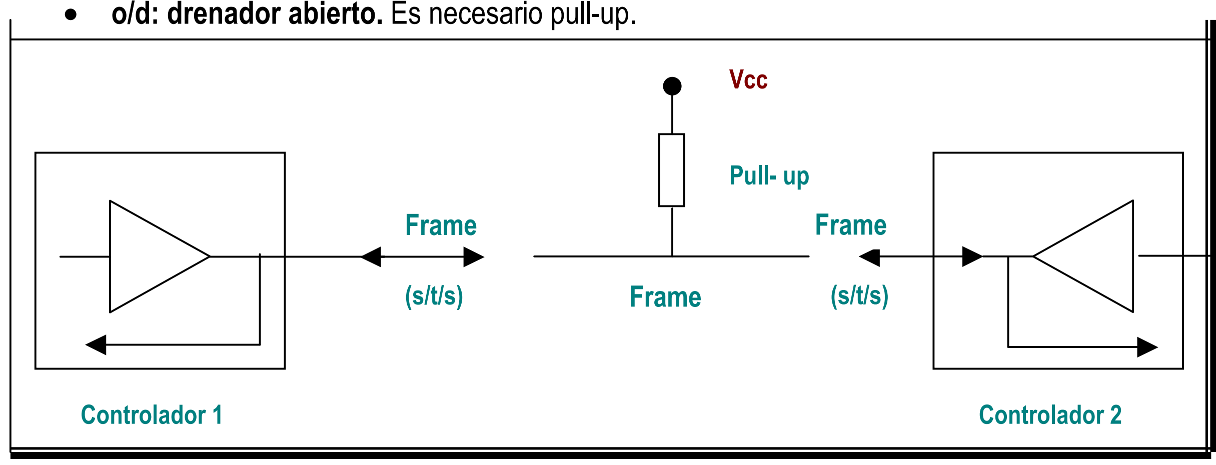


Figura 2.4 Señal del Bus PCI.

2.5.3 SEÑALES POR FUNCIÓN.

2.5.3.1 Señales del sistema.

CLK	**Clock**	Reloj del Bus PCI. Todas las señales se muestran en flanco descendente de esta señal, menos **RST#**, **INTA#**, **INTB#**, **INTC#** y **INTD#**.	in
RST#	**Reset**	Inicializa todos los registros, secuencias y señales.	in

Tabla 2.2: Señales del sistema.

2.5.3.2 Señales de direcciones y datos.

AD[31:0]	**Address & Data**	Bus multiplexado de direcciones y datos en el que se produce la transición del Bus PCI: Consiste en una fase de dirección seguida de una o más fases de datos.	t/s
C/BE[3:0]	**Bus Command and Bytes Enables**	Todos ellos multiplexados en el mismo pin. Indica de la operación que se va a realizar en el bus Durante una fase de dirección, define la orden del bus(actúa como comando). Durante una fase de datos, se usa como bytes enables del bus de datos y direcciones.	t/s
PAR	**Parity**	Paridad de AD y C/BE. Para una fase de dirección **PAR será** válida un ciclo de reloj después de ésta. Para una fase de datos, **PAR será** válida un ciclo de reloj después de haber comenzado.	t/s

Tabla 2.3: Señales de direcciones y datos.

2.5.3.3 Señales de control.

FRAME#	**Cycle Frame**	Activada por el Maestro actual para indicar el comienzo y duración de un acceso. **FRAME#** se activa en el inicio de la transición y mientras se mantiene, la transición continúa.	s /t/s
IRDY#	**Initiator Ready**	Activada por el maestro, en un ciclo de escritura indica que hay un dato válido en el bus **AD**, en un ciclo de lectura, indica que el maestro esta preparado para recibir un dato. Puede ser usado junto con la señal **TRDY#**.	s /t/s
TRDY#	**Target Ready**	Activada por el esclavo. En una lectura indica que hay un dato válido en el bus AD. En una escritura que esta preparado para recibir un dato.	s /t/s
STOP#	**Stop**	Lo activa el esclavo para indicarle al maestro que la transición actual se acaba.	s /t/s
IDSEL	**Initialization Device Select**	Es usado como un Chip Select durante las transacciones de escritura y lectura de configuración.	In(4)
DEVSEL#	**Device Select**	La activa el esclavo indicando al Maestro, que el dispositivo activado, tiene decodificada su dirección de acceso.	s /t/s(2)

Tabla 2.4: Señales de control.

2.5.3.4 Señales de arbitraje.

REQ#	**Request**	La activa el maestro para pedir el arbitrio del bus.	t/s
GNT#	**Grant**	Activada por el árbitro, par dar el acceso al maestro.	t/s

Tabla 2.5: Señales de arbitraje.

2.5.3.5 Señales de estado.

PERR#	**Parity Error**	Informa de un error de la paridad de datos durante todas las transacciones PCI, excepto durante los ciclos especiales. Se activa dos ciclos de reloj después de recibir los datos si hubo error. El tiempo mínimo de dirección es de un ciclo de reloj.	s/ t/

Tabla 2.6: Señales de estado.

[4] entrada estándar.
[2] tri-estado mantenido

2.6 OPERACIONES EN EL BUS PCI.

2.6.1 RELOJ DEL BUS PCI.

Todas las acciones en el Bus PCI están sincronizadas por la señal CLK PCI. La frecuencia de la señal CLK está comprendida entre 0 y 33 MHz.

La revisión 1.0 de la especificación, define que todos los dispositivos deben soportar operaciones desde 16 a 33 MHz, mientras se recomienda soporte de operaciones por debajo de 0 MHz.
La revisión 2x, con x = 1 o 2, indica que todos los dispositivos PCI deben soportar operaciones PCI dentro del rango de 0 a 33 MHz. En un bus con reloj funcionando a 33 MHz o menor, la frecuencia de reloj puede ser cambiada en cualquier momento y también puede detenerse.
La revisión 2.1 también define la operación del bus PCI a velocidades de hasta 66 MHz.

2.6.2 COMANDOS.

Los comandos indican al esclavo el tipo de transición que el maestro necesita. Se codifica en el bus **C/BE#** durante una fase de dirección. Son opcionales excepto los de configuración.

C/BE[3:0]	Comando	Descripción
0000	Interrupción Ack.	Direccionamiento de lectura al sistema controlado por interrupciones.
0001	Ciclo especial[1].	Mensaje por broadcast[2].
0010	Lectura I/O.	Leer datos de un µC mapeado en el espacio I/O.
0011	Escritura I/O.	Escribir datos en I/O.
0100	Reservado.	-
0101	Reservado.	-
0110	Lectura de Memoria.	Leer datos de un µC mapeado en el espacio de memoria.
0111	Escritura en Memoria.	Escribir datos en memoria.
1000	Reservado.	-
1001	Reservado.	-
1010	Lectura de configuración.	Leer datos de un µC mapeado en el espacio I/O.
1011	Escritura en configuración.	Escribir datos en configuración.
1100	Múltiple lectura de memoria.	Semejante a la lectura de memoria, pero se pueden leer múltiples líneas caché [3].
1101	Ciclo de dirección dual.	Para transferir direcciones de 64 bits.
1110	Línea de lectura de memoria.	Semejante a la lectura de memoria, además, indica que 3 o más datos se pueden leer de la línea caché.
1111	Escritura de memoria e invalidar.	Semejante a la escritura de memoria, pero garantiza una transferencia mínima de toda la capacidad de la línea caché.

Tabla 2.7: Códigos de comando.

[1] Ciclo especial: Sirve para enviar un mensaje simple con el mecanismo de broadcast (envió a todos) al PCI.
[2] Emisión.
[3] Parte de la memoria dinámica principal, se guarda en la memoria caché para conseguir accesos más rápidos. La parte más pequeña que se puede guardar se llama línea caché.

2.6.3 FUNDAMENTOS DEL PROTOCOLO PCI.

El mecanismo básico de transferencia de un bus PCI es burst(4), que está compuesto por una fase de dirección, seguida de una o varias de datos. Soporta transferencias burst en el espacio I/O y de memoria.

Cada señal tiene su tiempo de setup y de hold respecto al flanco descendente de la señal de reloj que se ha de respetar.

Todas las transferencias se controlan por 3 señales: **FRAME#**, **IRDY#** y **TRDY#**,

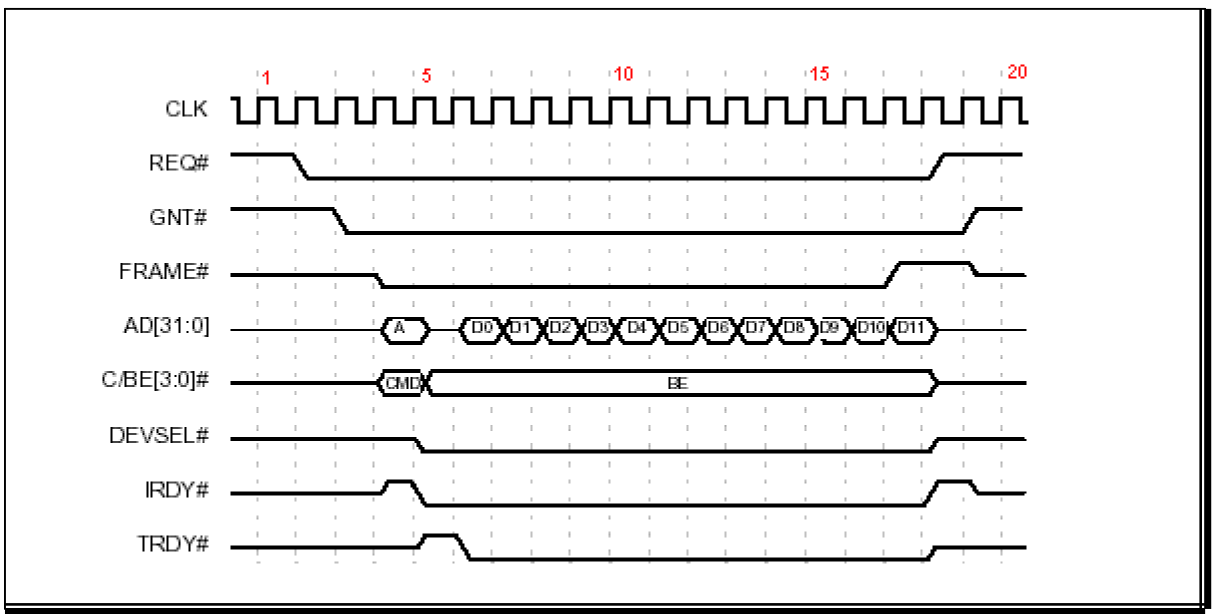


Figura 2.5 Transferencia del Bus PCI.

El controlador comienza en estado de espera: FRAME#, IRDY# inactivas.
Fase de dirección: en el 1er ciclo de reloj que se activa FRAME#. La dirección y el comando se transmite en el mismo ciclo de reloj.
Fase de datos: Comienza en el siguiente ciclo de reloj; IRDY# y TRDY# están activas (si alguna se desactiva durante el periodo de una fase de datos, se introduce una espera)

- Los datos son válidos en una transición de escritura cuando **IRDY#** está activa.
- En una transición de lectura cuando **TRDY#** está activa.

El controlador puede retrasar la activación de **xRDT#** cuando no está preparado sin sobrepasar las especificaciones. Los datos sólo se transfieren cuando **IRDY#** y **TRDY#** están activas conjuntamente.

[4] ráfaga

Cuando el maestro activa **IRDY#,** no puede cambiarse **IRDY#** ni **FRAME#** hasta que termine la fase de datos. Cuando el esclavo activa **TRDY#** o **STOP#**, no puede cambiar; **DEVSEL#, TRDY#** ni **STOP#** hasta que se acabe la fase de datos**.**

Cada vez que el maestro intenta acabar uno o más datos de una transferencia, desactiva FRAME# (IRDY# activa para indicar que está preparado). Después el esclavo indica que está a punto de acabar la transferencia activando TRDY#. Inmediatamente el controlador retorna al estado de espera con FRAME# y IRDY# desactivadas.

2.6.3.1 Arbitración.

Este elemento es imprescindible, ya que pueden existir varios dispositivos conectados al Bus PCI del ordenador. Si, por ejemplo, uno de ellos (una tarjeta de adquisición de datos) tuviese que enviar una gran cantidad de datos a la memoria principal, monopolizaría el Bus PCI durante bastante tiempo, impidiendo que la tarjeta de vídeo (entre otros posibles dispositivos) recibiese la información que necesita de la CPU, por lo que aparecería una situación de mal funcionamiento en el sistema.

Debido a esto, existe este elemento, que impide que haya dispositivos que transfieran datos durante mucho tiempo, permitiendo de esta manera que todos los dispositivos vayan enviando datos durante intervalos determinados de tiempo.

Una ventaja que posee el Bus PCI, es que la negociación entre el árbitro PCI y un dispositivo iniciador se puede realizar mientras otro dispositivo tiene el control del bus y esta en el transcurso de una transacción. A esto se le llama arbitraje oculto(1), que evita tener que perder un tiempo innecesario en la adjudicación del bus cuando éste estuviese libre. Así, en cuanto un dispositivo cede el control del bus, rápidamente otro dispositivo toma el control, consiguiendo que el Bus PCI esté en uso en todo momento.

El algoritmo usado por un árbitro de Bus PCI decide entre quien de los maestros que piden el bus la ganará.

Árbitro.

En un instante de tiempo dado, uno o más dispositivos maestros de bus pueden requerir el uso del Bus PCI para realizar transferencias de datos con otro dispositivo PCI. Cada maestro que lo pide, activa su salida REQ# para informar al árbitro de bus de su petición pendiente para el uso del bus.

En la figura 2.6 de la página siguiente, se muestra la relación de varios maestros PCI con el árbitro de bus.

En este ejemplo, hay 5 posibles maestros conectados al árbitro de Bus PCI. Cada maestro está conectado al árbitro mediante diferentes señales REQ# y GNT#. Aunque se muestra el árbitro como un componente diferente, normalmente se encuentra integrado en el puente CPU-PCI.

(1) hidden arbitration

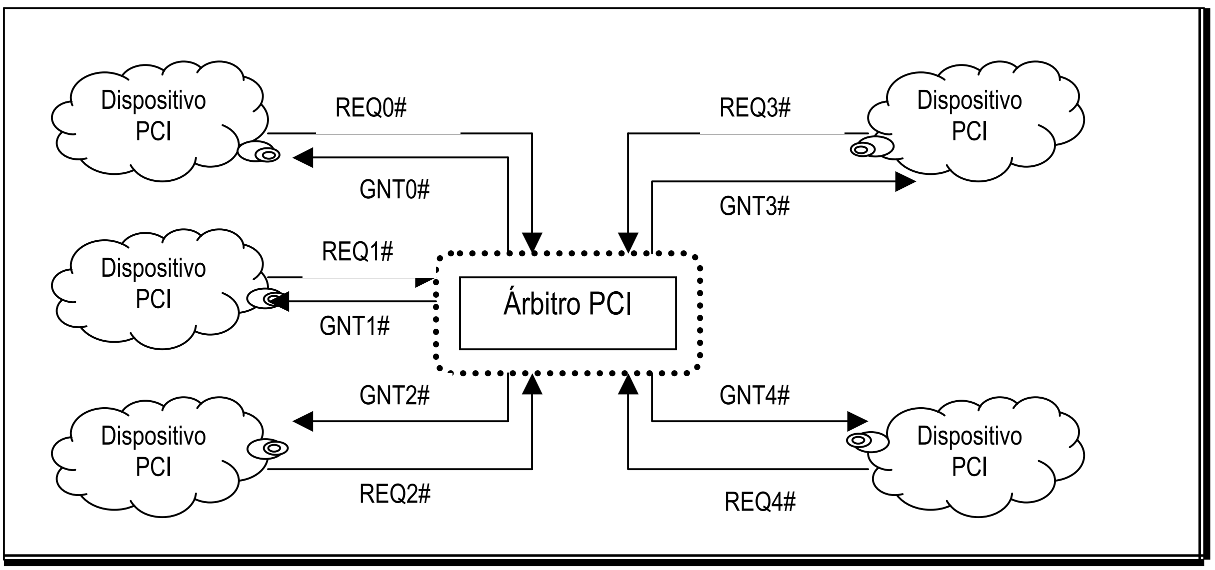


Figura 2.6 Arbitración de varios maestros de bus.

Algoritmo de arbitración.

El árbitro puede utilizar cualquier esquema, como mixto, rotacional o una mezcla de los dos. Idealmente el árbitro de Bus debería ser programado por el sistema, en este caso, el software de configuración puede determinar la prioridad para ser asignada a cada miembro de la comunidad de buses maestros, leyendo desde el registro de máxima latencia (Max_Lat), asociada con cada bus maestro.

El diseñador del bus maestro escribe este registro para indicar, en incrementos de 250 ns, como de rápido el maestro requerirá el acceso al bus para realizar una adecuada prioridad.

Para ceder el Bus PCI a un bus maestro, el árbitro activa su respectiva señal GNT#. Este cede el bus al maestro para una transacción (que puede estar constituida de una o varias fases de datos)

Si el maestro genera una petición y se le cede el bus, y tras 16 señales de reloj PCI, después de que el bus ponga idle, no inicia la transacción (activo FRAME#), el árbitro puede asumir que el maestro esta funcionando mal.

2.6.3.2 Direccionamiento.

Se permite el acceso a 3 espacios de direcciones:

1. **Memoria:** Acceso de hasta 4GB (2^{32}) o (2^{64}) posiciones si se usan 64 bits.
2. **E/S:** Acceso de hasta 4GB (2^{32}) o (2^{64}) posiciones si se usan 64 bits.
3. **Configuración:** Está dividido en un especio de configuración por controlador PCI formado por 64 palabras dobles:

- 7 primeras: Espacio de cabecera de configuración.
- Resto: Registros de configuración específicos del dispositivo.

La información del bus AD varía según el espacio de direcciones seleccionado.

Direccionamiento de memoria

La dirección de comienzo usada durante cualquier forma de transacción a memoria, es una dirección Dword en AD[31:2] durante la fase de dirección.

Durante la secuencia de dirección durante el direccionamiento de memoria en modo ráfaga, la memoria maestra comienza la dirección en un contador de direcciones y la usa para la primera fase de datos. Hasta que se complete esta fase y asuma que no se trata de una transferencia simple de datos, la memoria debe actualizar su contador de direcciones en el punto siguiente Dword para ser transferido.

En un acceso a memoria, la memoria Target debe chequear el estado de los bits de dirección 0 y 1, AD[1:0], para determinar la política que usa cuando actualice su contador de direcciones al final de cada fase de datos. En la siguiente tabla se muestra el significado de estos bits.

AD1	AD0	Secuencia de direccionamiento
0	0	**Lineal o secuencial.** Direccionamiento secuencial durante la ráfaga.
0	1	**Reservada.** Indica el direccionamiento del modo Toggle Intel. Cuando se detecte, la memoria Target debe desconectar con una transferencia de datos durante la 1ª fase de datos o una desconexión sin transferencia de datos durante la 2ª fase de datos.
1	0	**Modo Cache line Wrap.**
1	1	**Reservado.** Cuando se detecte, la memoria Target indica una señal de desconexión con una transferencia de datos durante la 1ª fase de datos o una desconexión sin transferencia de datos durante la 2ª fase de datos.

Tabla 2.8 Líneas bajas del direccionamiento de memoria

Direccionamiento PCI I/O.

Para asegurar un correcto funcionamiento del dispositivo I/O, los puentes nunca deben usarse en accesos secuenciales I/O en una fase singular de datos o en fases de datos múltiples.

Cada transición individual I/O generada por el procesador debe ser realizada en el Bus PCI como aparece en el bus Host. Esta regla incluye a ambos espacios I/O y espacios de memoria mapeada I/O.
Los puentes no están permitidos para realizar uniones de bytes en sus buffer de escritura de memoria posterior cuando se está escribiendo a una memoria de no pre-escritura.

Durante una transición I/O, el comienzo de dirección I/O colocado en el bus AD durante la fase de dirección tiene el siguiente formato.
AD[32:2] identifica la palabra Dword del espacio I/O.
AD[1:0] identifica el byte menos significativo dentro de la palabra Dword con que el Initiator desea realizar una transferencia, por ejemplo 00b es el byte 0, 01b es el byte 1, etc.

Al final de la fase de dirección, todos los I/O Target mantienen el comienzo de dirección, el comando de lectura o escritura I/O y el comienzo de la dirección.

2.6.3.3 Accionamiento del bus.

Un ciclo turnaround(1), permite evitar colisiones cuando un controlador deja de activar la señal y otro comienza a activarla(necesario para las señales que se activan para más de un controlador). Las señales **IRDY#**, **DEVSEL#**, **TRDY#** y **STOP#** usan la fase de dirección para hacerlo. **FRAME#**, **C/BE#** y **AD** usan el estado de espera entre transiciones para hacer este ciclo. La señal **PERR#** ha de hacerlo en el cuarto ciclo de reloj después de la última fase de datos.

2.6.4 TEMPORIZACIÓN DEL BUS PCI.

El Bus PCI es un bus síncrono, que envía y recibe todas las transferencias de datos con referencia a la señal de reloj del sistema (CLK). La interrelación mútua entre las señales individuales se pueden ilustrar utilizando el comando “Read” como un ejemplo, véase el diagrama de tiempos mostrado en la figura 2.7. Los pasos individuales en el procesamiento de un comando “Read” son los siguientes:

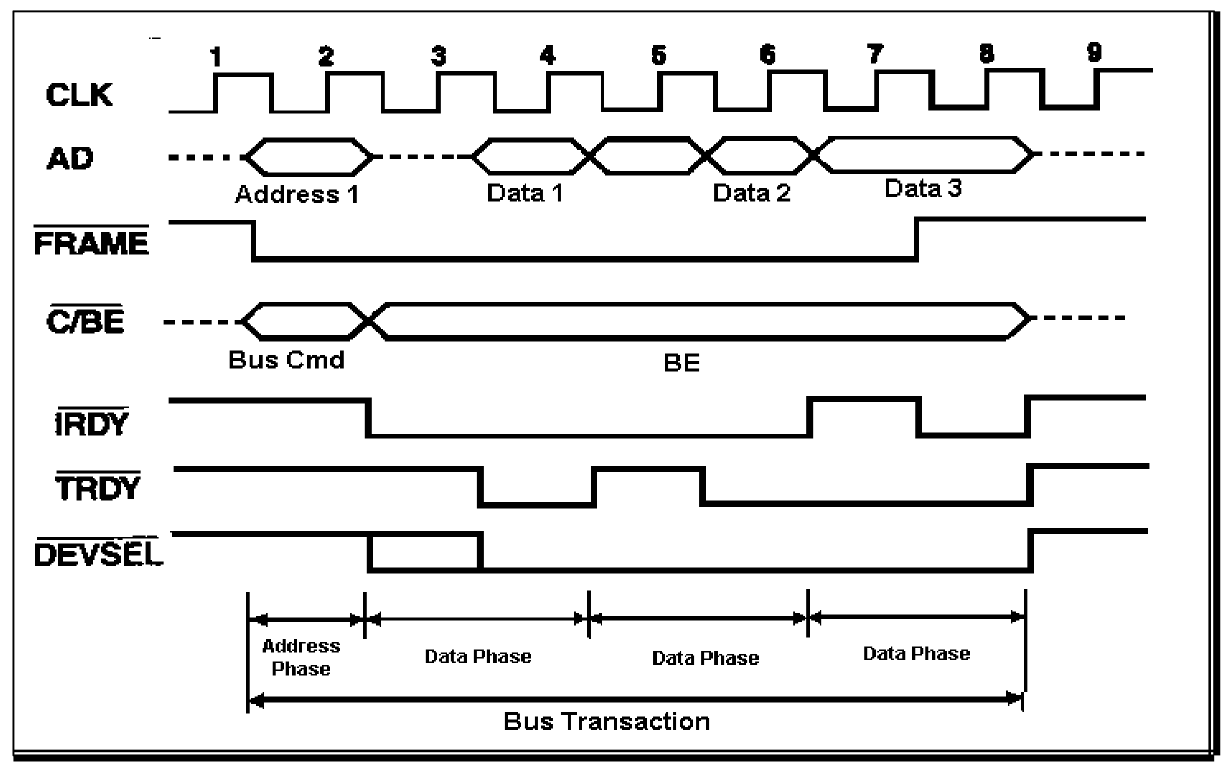


Figura 2.7. Diagrama de tiempos.

Ciclo 1:
El bus PCI se encuentra en el estado de reposo.

Ciclo 2:
Se activa la señal FRAME # (TRAMA #), por medio del dispositivo maestro para indicar el inicio de una transición.
El registro AD almacena una dirección.
C/BE# almacena un comando del bus, indicando que operación se está realizando

[1] vuelta.

Las señales IRDY #, TRDY # y DEVSEL # están en un ciclo de cambio, ya que se está produciendo un intercambio de controlador.

Ciclo 3:
El registro AD realiza un cambio de estado desde el dispositivo maestro hasta el dispositivo objetivo, debido al control de fase. C/BE# está controlado por la señal "Byte Enable" (Byte Habilitado).
La señal IRDY # está activa debido a que el dispositivo maestro está listo para leer el dato. La señal TRDY# se mantiene inactiva debido a que el dispositivo objetivo no ha sido encontrado todavía.

Ciclo 4:
El registro AD almacena el primer dato. La señal TRDY# está activa debido a que el primer dato legible está disponible en el bus. La señal DEVSEL# está activa ya que el dispositivo objetivo ha reconocido la dirección que se envió en el Ciclo 2.

Ciclo 5:
El registro AD aún almacena el primer dato de la fase de dato 1.
La señal TRDY# está inactiva debido a que el dispositivo objetivo (fuente de datos) quiere insertar una pausa.

Ciclo 6:
El registro AD almacena el dato de la fase de dato 2.La señal TRDY# indica que el dato puede ser leído.

Ciclo 7:
El registro AD obtiene el dato de la fase de dato 3. La señal IRDY# está desactivada por el dispositivo maestro, ya que desea disponer de un estado de espera.

Ciclo 8:
La señal FRAME#, AD y C/BE# inician una serie de cambios de estado de modo que los controladores puedan intercambiarse datos.
Las señales IRDY#, TRDY# y DEVSEL# están inactivas, ya que no se está produciendo ninguna transacción de datos.
El bus está en el estado de reposo, ya que las señales FRAME# =1 e IRDY# =1.

2.6.5 VELOCIDAD DEL BUS PCI.

Gracia a su independencia de la velocidad del procesador, el bus PCI puede ofrecer nuevas posibilidades para ampliaciones tales como multimedia, servicio de redes, instrumentación y muchas otras. Esto es debido principalmente a sus altas velocidades de transferencia de datos (especialmente la versión PCI Bus 64 bits 2.1), lo que permite que los datos puedan procesarse de forma más rápida entre tarjetas PCI y el procesador, en ambas direcciones. Si utilizamos una vieja tarjeta de sonido ISA en combinación con una tarjeta aceleradora de gráficos 3D PCI, podríamos estar disminuyendo la velocidad del proceso de nuestro sistema alrededor de un 10% o un 20 % bajo ciertas condiciones, si las prestaciones disponibles de la placa PCI están siendo explotadas totalmente.

La configuración individual de la BIOS de un ordenador también es importante con respecto a las prestaciones estables del sistema, ya que dicha configuración tiene una influencia considerable en la estabilidad del sistema. Gracias al ancho del bus de datos y a las altas velocidades de reloj de las

placas base modernas, el bus PCI trabaja a velocidades mucho más altas que las del bus ISA estándar, por ejemplo.

Por otro lado, para una transferencia de datos libre de errores, es necesario realizar ciertos retardos en el Bus PCI, ya que la placa base no siempre puede alcanzar los límites de trabajo de las tarjetas insertadas en las distintas ranuras. Las siguientes opciones ajustan la longitud de retardo en el bus PCI para una transacción entre la ranura PCI especificada y la CPU. El valor depende, entre otras cosas de la unidad maestra PCI que se está utilizando.

Una de las configuraciones más importantes es el valor del tiempo del Temporizador de Latencia PCI, el cuál debe estar configurado normalmente a 32 ciclos de reloj. Esta opción determina la cantidad de tiempo que se permite a la tarjeta PCI reservar el Bus PCI como bus maestro, cuando otra tarjeta PCI está también solicitando el acceso al bus. Si la configuración para la liberación del Temporizador de Latencia PCI es demasiado corta o demasiado larga, es posible para la CPU acceder a los circuitos de modo más rápido que lo permitido por el procesamiento en el decodificador. Aunque esto puede aumentar de forma generosa la velocidad, tanto del ordenador como de las tarjetas (especialmente con un valor de 0), a menudo tiene consecuencias que sólo pueden mostrar después de un amplio periodo de funcionamiento (tales como sobrecalentamiento, seguido por fallos en los circuitos). Ambos problemas, direcciones de E/S erróneas, dispositivo de datos incorrecto y problemas de acceso a los circuitos, son ejemplos de los tipos de errores que pueden suceder si la configuración de la BIOS no es correcta (demasiado rápido para los circuitos instalados en las ranuras de bus).

La segunda característica más importante en la BIOS es la configuración de Retardo de Transacción PCI(del inglés PCI Delay Transaction), la cuál debe estar configurada como “Enable” . Todo esto se hace para especificar que los circuitos utilizados cumplan al menos con la especificación PCI 2.1. Los protocolos internos del bus y los paquetes de datos especiales serán transferidos de acuerdo a la especificación, algo que se realiza para incrementar las prestaciones entre la unidad de proceso y las placas PCI colocadas en las ranuras. Por supuesto, también hay una gran cantidad de factores que entran en juego en lo que respecta a las transacciones del bus PCI.

2.6.6 LATENCIA.

Latencia del esclavo:
Número de ciclos de reloj que el esclavo espera antes de activar **TRDY#**. Todos los esclavos necesitan completar la fase de datos inicial antes de 16 ciclos de reloj tras la activación de **FRAME#.**

Latencia de los datos del maestro:
Número de ciclos que el maestro necesita para activar **IRDY#**, que indica que está preparado para transferir datos. Todos los maestros activan esta señal dentro de 8 ciclos de reloj después de activar **FRAME#.**

Latencia del árbitro:
Número de ciclos desde el momento que el maestro activa **REQ#** hasta que el bus llega al estado de espera y el maestro recibe la señal **GNT#** activa.

El timer de latencia del maestro es un temporizador programable por cada maestro a través de un registro de configuración. El temporizador empieza a contar una vez que el maestro activa **FRAME#**.

2.6.7 TRANSICIONES EN EL BUS PCI.

Como ejemplo de éstas, mostraremos la operación de lectura:

1. Comienza cuando **FRAME#** se activa (figura 2.8, 4º ciclo de reloj).
   - Dirección → **AD[31:0]**.
   - Comando → **C/BE#[3:0]**.
2. Fase de datos (ciclo 6º)
   - Bytes activos en la fase de datos → **C/BE#** su buffer de salida puede estar activo desde el 1er ciclo de la fase de datos hasta el final de la transición. En una lectura burst típicamente todos los bytes están activos.
   - La primera fase de datos requiere turnaround (forzada por el esclavo con **TRDY#**).
3. Cuando se activa **DEVSEL#** el esclavo activa el bus **AD**(no esta flotante) después del turnaround hasta el final de la transición. **TRDY#** no se activa hasta **DEVSEL#**.
4. El maestro avisa (en el ciclo 17) que la próxima fase de datos será la última desactivando **FRAME#** (si **IRDY#** está activa, es decir, el maestro está preparado).
5. La fase de datos acaba cuando se transfieren todos los datos y las señales **TRDY#** y **IRDY#** se desactivan en el mismo flanco de reloj.
6. Cuando **TRDY#** o **IRDY#** se desactivan se inserta un estado de espera y el dato no se transfiere.

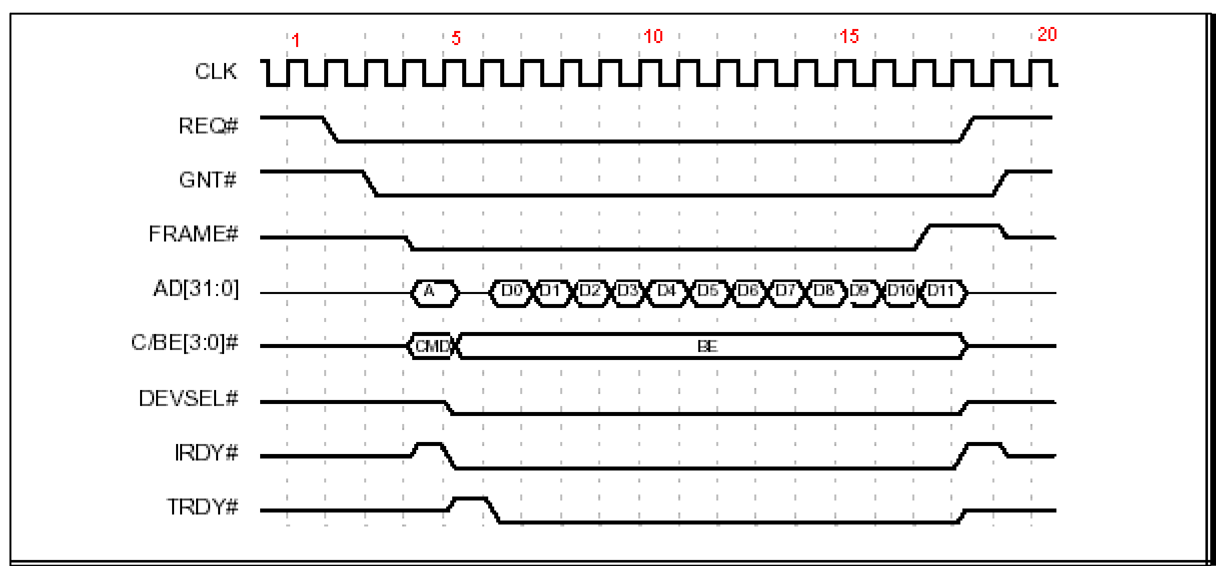


Figura 2.8 Ciclo de transición en el Bus PCI.

1. Comienza cuando **FRAME#** se activa (Figura 2.6, **4**º ciclo de reloj)

2. No se requiere turnaround después de la fase de dirección, porque el maestro suministra la dirección y el dato. La fase de datos es igual que antes.

2.6.8 TERMINACIONES EN EL BUS PCI.

Terminación iniciada por el maestro.

Activando la señal **FRAME#** y desactivando **IRDY#**. El final de la fase de datos, es cuando se desactivan **TRDY#** y **IRDY#**.

- **Transición completa**: Acaba la transición normalmente.
- **Timeout**: Cuando el maestro ve la señal **GNT#** activa y el tiempo de latencia expira porque el esclavo induce un tiempo en el acceso o porque la operación es muy larga.
  Permite al maestro terminar una transición cuando el esclavo no responde (el maestro aborta) después de 5 ciclos de reloj si **DEVSEL#** no se activa.

Terminación iniciada por el esclavo.

En condiciones normales el esclavo suministra los datos para el maestro. Pero cuando no puede responder a la solicitud, el esclavo puede activar **STOP#** para iniciar una terminación de transición.

Entre los tipos de terminaciones, encontramos:

- **Retry**: Terminación demandada antes de transferir datos porque el esclavo está ocupado y desactiva temporalmente la transición. Se inicia activando **STOP#** y desactivando **TRDY#** en el inicio de la fase de datos.
- **Disconnect**: Terminación demandada durante o después de que los datos sean transferidos porque el esclavo es incapaz de responder a tiempo. La transición se desconecta temporalmente, y los datos pueden ser transferidos o no. Lo inicia activando a la vez **STOP#** y **TRDY#**.
- **Target- abort**: Terminación anormal porque el esclavo detecta un error fatal o que nunca será capaz de cumplir lo que se le demanda. Una vez activada **DEVSEL#** se puede provocar cuando se desactivan y activan a la vez **DEVSEL#** y **STOP#**, respectivamente.

2.6.9 SEÑALES DE PROTOCOLO PARA EL ÁRBITRO.

Un controlador demanda el bus activando la señal **REQ#**, cuando realmente lo necesite. Un controlador nunca puede usar **REQ#** para aparcarse. Si se ha implementado bus parking, el árbitro designa el bus al poseedor por defecto. Cuando el árbitro determina que un controlador puede usar el bus activa la señal **GNT#** del controlador.

Un maestro puede comenzar una transición cuando **GNT#** está activa y el bus en espera, independientemente de **REQ#**. Una vez **GNT#** está activa, se puede desactivar mediante:
Si **GNT#** se desactiva y **FRAME#** se activa en el mismo flanco de reloj, la transición es válida.
Se puede desactivar coincidiendo con la activación de otra **GNT#**, si el bus no está en estado de espera. En caso contrario, es necesario un ciclo de reloj entre la desactivación y la activación de **GNT#**.
Mientras **FRAME#** está activa, **GNT#** se puede desactivar por orden de un servicio de alta prioridad de un maestro.

2.6.10 OTRAS OPERACIONES SOBRE EL BUS PCI.

Selección del controlador.

La señal **DEVSEL#** activada por el esclavo indica que éste puede responder a la transición. **DEVSEL#** se puede activar:

- 1 ciclo de reloj después de la fase de dirección → decodificación rápida.
- 2 ciclos de reloj después de la fase de dirección → descodificación media.
- Ciclos de reloj después de la fase de dirección → decodificación lenta. Si el sistema no está pensado para este tipo de decodificación ignorará **DEVSEL#**, is est, abortará.

Cada esclavo indica el tiempo de selección en el registro de configuración de estado.
Si el esclavo recibe un comando de configuración, **IDSEL#** está activa y **AD[1:0]** es “00”, esto activa **DEVSEL#** para responder a la transición de configuración.

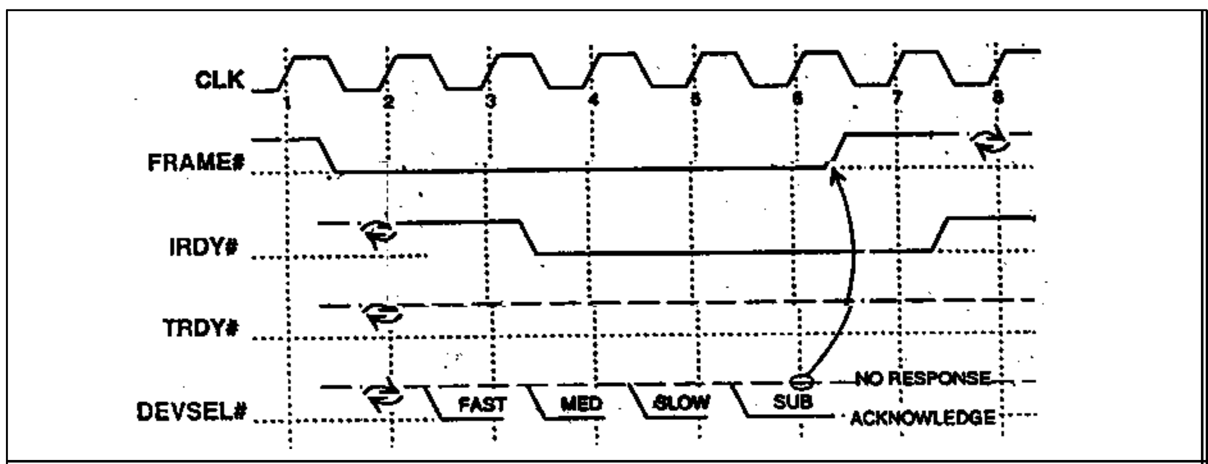


Figura 2.9 Selección del controlador.

Ciclo de configuración.

Cuando se pone en marcha el sistema, un software de configuración, el PCI bus enumerator, busca los controladores que existen. Para facilitar esto, todos los controladores PCI, tienen unos registros de configuración definidos por la especificación del bus. Dependiendo de la funcionalidad, pueden tener otros registros de configuración.

El software de configuración accede a los registros del controlador para determinar su presencia y el tipo. Después, accede al registro de dirección base, que determina cuantos bloques de memoria o E/S necesita. Y el software carga en el registro de la dirección base donde comienza a descodificar. De esta forma el controlador responde en el margen de direcciones asignado.

La figura 2.8, muestra un ciclo de configuración de un comando de lectura. Un controlador responde a un comando de configuración cuando:

- IDSEL se activa.
- AD[1:0] son “00”.
- AD[7:2] contiene el valor para acceder a uno de los 64 registros DWORD.
- C/BE contiene el comando en la fase de dirección y los bytes activos en la de datos.

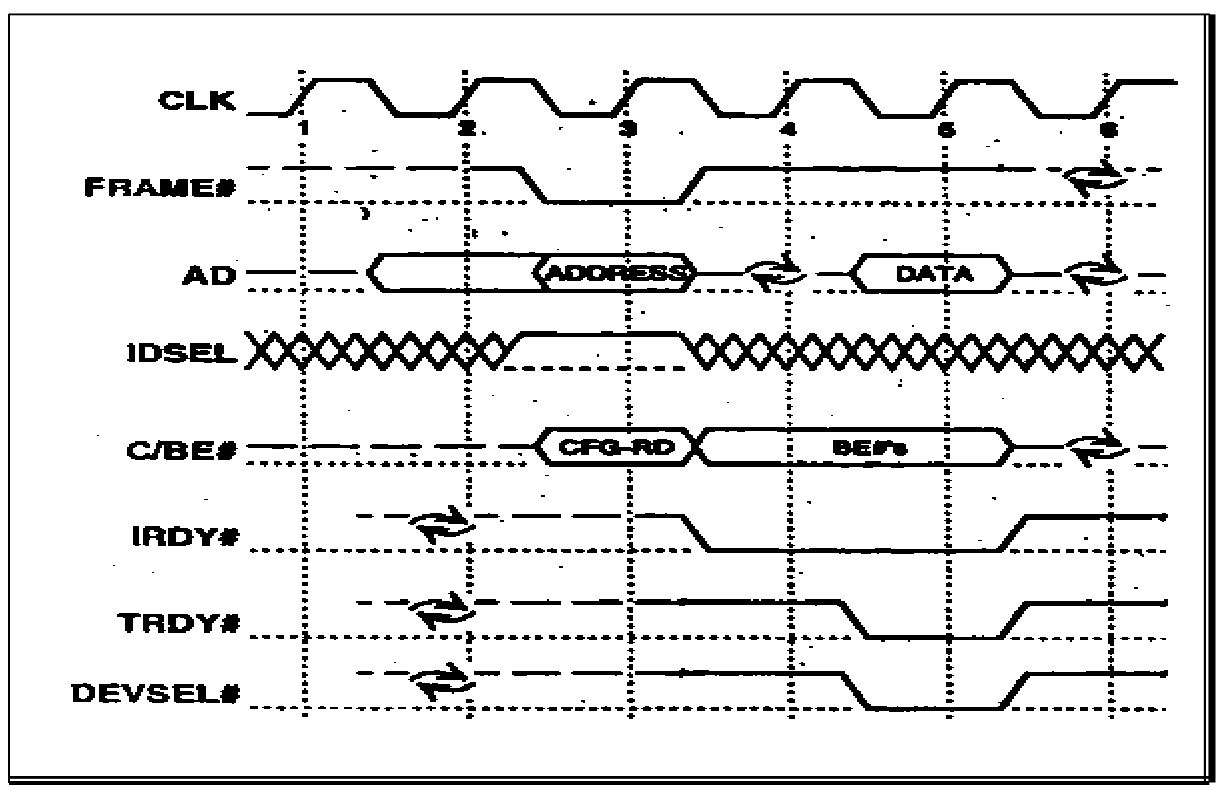


Figura 2.10 Ciclo de configuración.

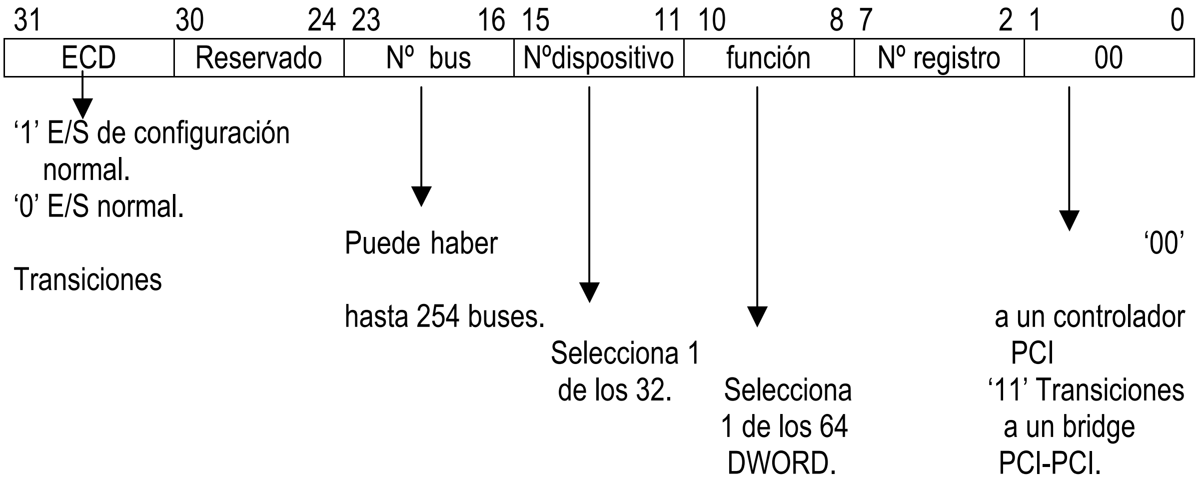
Mecanismos de configuración.

Hay dos métodos que usan acceso a E/S iniciadas por el procesador para indicar al host/ PCI bridge que se ha de ejecutar un acceso de configuración.

1. Mecanismo de configuración #2: Se define sólo para mantener compatibilidades con las primeras versiones PCI (revisión 1.0)

2. Mecanismo de configuración #1: 2 puertos E/S de 32 bits (direcciones 0CF8h y 0CFCh):

a. Puerto de direcciones de configuración: 0CF8h-0CFBh.



b. Puerto de datos de configuración 0CFCh –0CFFh.

Tras escribir en el puerto de direcciones de configuración, hay que realizar una lectura o escritura de este puerto.

Existen 2 tipos de conversión:

b.1 Tipo 0. Cuando el controlador direccionado está en el bus PCI conectado al bridge. El bridge descodifica el número del controlador y activa la señal **IDSEL** adecuada. El resto de los bits, del 0 al 10 (de las direcciones de configuración) son una dirección y se copian a **AD.**

31	30 24	23 16	15 11	10 8	7 2	1 0
	Reservado	Nº bus	Nºdispositivo	función	Nº registro	00
1 de 21 líneas de IDSEL o reservadas				función	Nº registro	00

b.2 Tipo 1. Cuando el controlador está en otro bus PCI detrás de bridge. El bridge hace una copia del contenido del registro de configuración de dirección al bus AD, durante la fase de dirección.

31	30 24	23 16	15 11	10 8	7 2	1 0
	Reservado	Nº bus	Nºdispositivo	función	Nº registro	00
0000 0000				función	Nº registro	00

2.6.11 ERRORES.

El PCI incorpora detección de error en cada transición de paridad. Para asegurar que los datos se transfieren bien, se calcula la paridad de:

Paridad ➔3 bytes de datos AD + bits del bus C/BE.

Se calcula de la misma manera para todas las transiciones y su generación no es opcional:
Será '1' si el número de '1' de **AD** y **CBE** es impar o '0' si es par.

El controlador que activa **AD** es el que activa **PAR.**

- El maestro activa la señal PAR en la fase de dirección (ciclos 3 y 7).
- El maestro activa la señal PAR en la fase de datos en una transición de escritura (ciclo 8).
- El esclavo activa la señal PAR en la fase de datos de una transición de lectura (ciclo 5).

Un controlador activa la señal **PERR#** cuando detecta un error de paridad, dos ciclos de reloj después de haberse transferido el dato. Cuando un maestro detecta un error de paridad activa **PERR#** en una transición de lectura o mira el estado de éste en una transición de escritura.
Para informar al sistema del error, se pone a '1' el bit 8 del registro status.

2.6.12 MÉTODO DE TRANSMISIÓN DE LAS SEÑALES

En el Bus PCI, el driver debe conseguir que por la línea del bus se transmitan 1.5 voltios, en vez de los 3 voltios de los buses anteriores. Las líneas del bus pueden considerarse líneas de transmisión (trabajan a frecuencias superiores a 30 MHz) con una impedancia característica de alrededor de 50Ω. De esta forma, cuando la señal de tensión incidente atraviese el bus, los dispositivos no registrarán ni un nivel alto, ni un nivel bajo, ya que 1.5 V es un valor intermedio. Cuando la señal de tensión llegue al final de la línea, se encontrará con un cambio de impedancia muy brusco, y se reflejará, superponiéndose en su recorrido de vuelta por el bus a la señal de 1.5 V incidente, dando como resultado la señal de tensión de 3 V necesaria para que los dispositivos registren un valor lógico alto (tecnología CMOS). Cuando la señal reflejada llegue de vuelta al driver, será absorbida por su resistencia interna, evitando que se refleje otra vez por el bus.

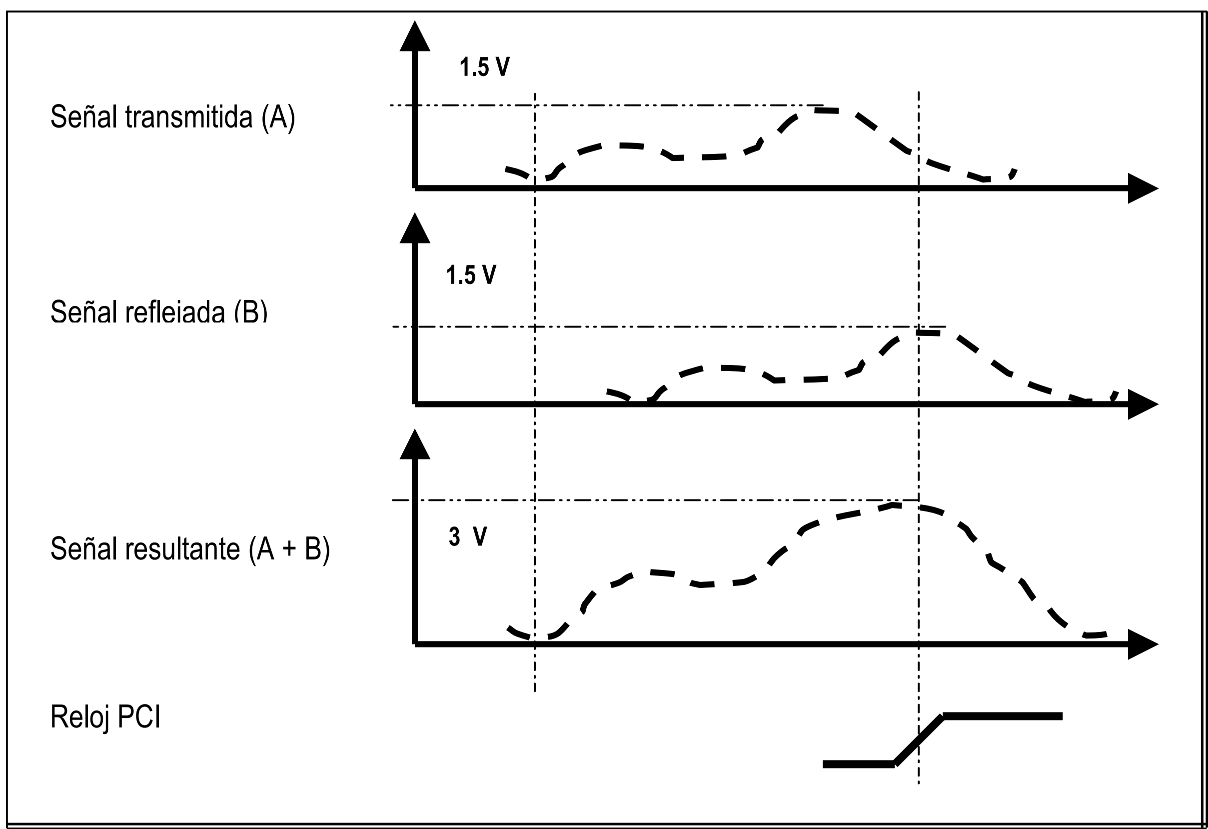


Figura 2.11 Tecnología CMOS de transmisión por onda reflejada.

Con este método, denominado transmisión por onda reflejada se ahorra en el consumo de corriente (casi hasta la mitad respecto a la transmisión por onda incidente que utilizaban los buses anteriores), en el consumo de potencia y, por tanto, en el tamaño final del driver.

Este método también tiene algunos inconvenientes. Ahora las señales que mande un dispositivo tendrán que recorrer todo el bus y reflejarse antes de poder ser muestreadas para obtener su valor. En un Bus PCI a 33 MHz, el tiempo de propagación es de 10-12 ns. Esto es un gran problema para implementar en la placa madre un bus PCI con una frecuencia por encima de 33 MHz, ya que siempre es preciso contar con este tiempo de propagación. También limita el número de dispositivos PCI por cada bus, ya que el bus no puede ser muy largo. Por esta razón suele haber en las placas madre cuatro o cinco ranuras de Bus PCI.

2.7 ORGANIZACIÓN DEL ESPACIO DE CONFIGURACIÓN.

Cada una de las unidades funcionales PCI dispone de 64 registros Dword reservados para sus registros de configuración. El formato de los 16 primeros (Configuración header región) está definido en la especificación PCI y existen 2 tipos:

- ❖ Tipo 1: Definido para el bridge PCI-PCI.
- ❖ Tipo 0: Definido para el resto de los controladores.

Los restantes registros son específicos de cada controlados.

<table>
<tr><td>00h</td><td colspan="2">ID del dispositivo</td><td colspan="2">ID del proveedor</td></tr>
<tr><td>04h</td><td colspan="2">Status</td><td colspan="2">Comando</td></tr>
<tr><td>08h</td><td colspan="3">Código de clase</td><td>ID de Revisión</td></tr>
<tr><td>0Ch</td><td>BIST</td><td>Tipo de cabecera</td><td>Timer de Latencia</td><td>Tamaño de línea caché</td></tr>
<tr><td>10h</td><td colspan="4" rowspan="6">Registros de direcciones base</td></tr>
<tr><td>14h</td></tr>
<tr><td>18h</td></tr>
<tr><td>1Ch</td></tr>
<tr><td>20h</td></tr>
<tr><td>24h</td></tr>
<tr><td>28h</td><td colspan="4">Puntero CIS</td></tr>
<tr><td>2Ch</td><td colspan="2">ID subsistema</td><td colspan="2">ID proveedor subsistema</td></tr>
<tr><td>30h</td><td colspan="4">Dirección base de expansión ROM</td></tr>
<tr><td>34h</td><td colspan="4">Reservado</td></tr>
<tr><td>38h</td><td colspan="4">Reservado</td></tr>
<tr><td>3Ch</td><td>Max_Lat</td><td>Min_Gnt</td><td>Pin Interrupción</td><td>Línea Interrupción</td></tr>
</table>

- ❖ **ID del vendedor:** Registro de 16 bits de lectura que indica al fabricante.
- ❖ **ID del dispositivo:** 16 bits de lectura que identifica al tipo de controlador.
- ❖ **Comando:** 16 bits de lectura/ escritura que proporcionan el control de diferentes accesos y respuestas del controlador.
  - **Habilitar acceso E/S** (bit0): El controlador responde a accesos de E/S cuando esta a '1'. Por defecto '0'.
  - **Habilitar acceso a memoria** (bit1): El controlador responde a accesos de memoria cuando está a '1'. Por defecto '0'.
  - **Habilitar maestro** (bit2): El controlador responde como maestro si vale'1'.

- **Reconocimiento ciclo especial** (bit 3): puede hacer transiciones de ciclo especial si es ‘1’. En caso contrario los ignora.
- **Habilitar Invalidar y Escritura en memoria** (bit4): EL maestro puede usar el comando Invalidar y Escritura en memoria si es ‘1’.
- **Habilitar Palette Snoop VGA** (bit 5): Accesos de escritura a los registros de la paleta. Controladores compatibles VGA y de gráficos tiene que estar a ‘1’.
- **Respuesta Error de paridad**(bit6): Tiene en cuenta los errores de paridad si es ‘1’.
- **Habilitar Ciclo de espera** (bit 7)
- **Habilitar Error del Sistema** (bit 8): a ‘1’ si el controlador puede responder a errores del sistema.
- **Habilitar Fast Back- to- Back** (bit 9): Si el controlador puede hacer transiciones Fast Back- to- Back debe estar a ‘1’.
- **El resto de los bits están reservados.**

❖ **Status:** 16 bits de lectura/ escritura que controlan el resultado de las operaciones del bus.

- **Capacidad 66 MHz** (bit 5): ‘1’ para 66 MHz, ‘0’ a 33MHZ. Es de sólo lectura.
- **Soporte UDF** (bit 6): Si puede soportar User Definable Features. Predefinido por el fabricante. Es de sólo lectura.
- **Capacidad Fast Back- to- Back** (bit 7): Si un esclavo es capaz de recibir transiciones Fast Back- to- Back cuando no son del mismo maestro. Sólo lectura.
- **Informar de Paridad de datos** (bit 8): Sólo cuando el controlador trabaja como maestro. Se pone a ‘1’ cuando se cumple:

    - La señal **PERR#** está activa.
    - El bit de Respuesta de Error de Paridad está activado.

- **Estado de los tiempos de selección de dispositivo** (bit 9:10): Se definen los tiempos de respuesta de la señal **DEVSEL#.**

    - “00” → rápido.
    - “01” → medio.
    - “10” → lento.
    - “11” → reservado.

- **Aborto del objetivo señalado** (bit 11): Cuando un esclavo aborta la transferencia se pone a ‘1’.
- **Aborto del objetivo recibido** (bit 12): Cuando un maestro detecta que el esclavo ha abortado.
- **Aborto del maestro recibido** (bit 13): Cuando un maestro aborta la transferencia.
- **Error del Sistema señalado** (bit 14): Cuando un controlador genera un error de sistema (**SERR#** activa) se pone a ‘1’.
- **Error de paridad Detectado** (bit15): Cuando un controlador detecta un error de paridad.

❖ **Registro ID de Revisión:** 8 bits de lectura que indican el número de revisión.

❖ **Registro de Código de Clase:** 24 bits de lectura divididos en 3 registros de 8 bits:

- **Clase base:** Indica la función básica del controlador.
- **Sub-clase:** Especifica una subclase más específica.
- **Interfaz de programación:** Interfaz de programación específica.

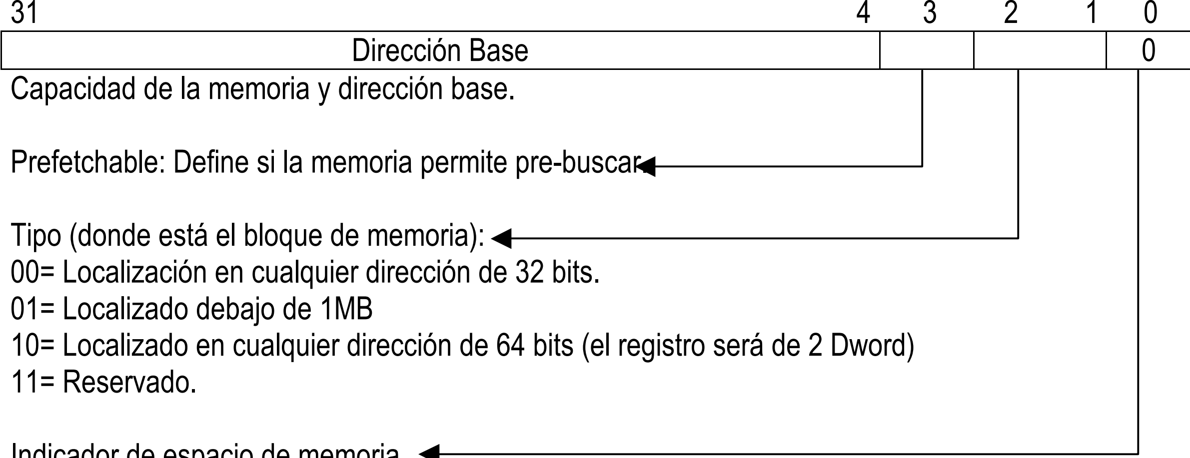
❖ **Registro Tipo Cabecera:** 8 bits de lectura divididos en dos partes:
- **Bit 7:** Indican si se trata de un controlador capaz de realizar más de una función.
- **Bits(6:0):** Indican el tipo de Cabecera de Configuración.

❖ **Temporización de Latencia:** 8 bits obligatorios para los maestros y que son capaces de realizar una transferencia burst de más de dos fases de datos. Define los tiempos en ciclos de reloj durante los cuales el maestro puede retener el bus. Una vez comenzada una transferencia en el bus, se decrementa el valor de este registro cada flanco descendente de reloj. La transferencia puede continuar hasta que termina o el valor del temporizador de latencia llega a 0, entonces otro contador maestro puede apropiarse del bus.

❖ **Registro BIST:** 8 bits para controladores que implementan el *test Built- In Self*.
- **Bit (3:0)** Indica si el test se hizo con éxito.
- **Bit (5:4)** Reservados.
- **Bit (6)** Controlan el BIST.
- **Bit (7)** Si vale '1' el cotrolador no soporta BIST.

❖ **Registro de Direcciones Base:** Permiten cambiar los espacios de memoria que se asignan a cada controlador PCI: Hay 6 registros y pueden ser de dos tipos:

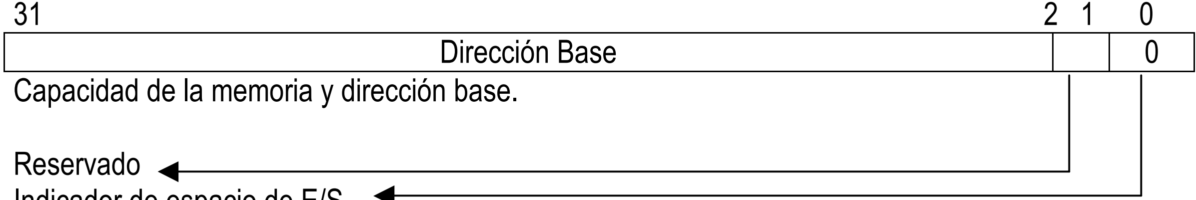
1. Registro Base de Memoria.



Para determinar la capacidad del bloque de memoria y la dirección base se han de seguir los siguientes pasos en el software de configuración:
Escribir '1' en todos los bits del registro.
Leer el valor retornado. Si es todo 0, el registro no está implementado. Comprobar que el bit 0 es '0' y que se trata de una solicitud de memoria.
Comenzar a buscar a partir del bit 4 en sentido ascendente el primer bit que valga '1'. Éste determina la capacidad del bloque demandado. Por ejemplo, si es el bit 5 el controlador demanda 32 KBytes.

Escribir la dirección base en el registro.

2. Registro Base de E/S.



Para terminar la capacidad del bloque de E/S y la dirección base se han de seguir los siguientes pasos en el software de configuración:

- Escribir '1' en todos los bits del registro.
- Leer el valor retornado. Si es todo 0, el registro no está implementado. Comprobar que el bit o
  es '1' y que se trata de una solicitud de E/S.
- Comenzar a buscar a partir del bit2 en sentido ascendente el primer bit que valga '1'. Éste determina la capacidad del bloque demandado. Por ejemplo, si es el bit 8, el controlador demanda 256 bytes.
- Escribir la dirección base en el registro.

❖ **Registro Pin de Interrupción**: 8 bits necesario solamente si el controlador genera peticiones de
interrupción. Hay 4 posibles líneas de solicitud de interrupción: **INTA#**, **INTB#**, **INTC#**, **INTD#.**
Estos registros (de lectura) pueden definirse:

- 00h: El controlador no genera interrupciones.
- 01h: El controlador usa INTA#.
- 02h: EL controlador usa INTA#.
- 03h: El controlador usa INTB#.
- 04h: El controlador usa INTC#.
- 05h: El controlador usa INTD#.

❖ Registro de Línea de Interrupción: 8 bits que identifican que línea de solicitud de interrupción se usa. En un sistema basado en PC, los valores 00h hasta 0Fh corresponden con las líneas IRQ0 a IRQ15. Los restantes valores están reservados, excepto FFh que indica que no hay conexión.

CAPÍTULO 3

Acceso Directo a Memoria (DMA).

A la hora de realizar transferencias con los periféricos tenemos varias opciones, mediante E/S programada (o transferencias maestro/esclavo), E/S por interrupciones, o transferencias DMA.

La E/S por interrupciones, aunque es más eficaz que la E/S controlada por programa, requiere también la intervención activa de la CPU para transferir datos entre memoria y un periférico. En ambos casos, cualquier transferencia de datos debe recorrer un camino que pasa a través de la CPU. Estas dos formas de E/S sufren de dos desventajas inherentes:

1) la transferencia de datos está limitada por la velocidad con que la CPU puede comprobar y atender un periférico.

2) la CPU está obligada gestionar la transferencia de E/S; hay que ejecutar una serie instrucciones durante cada operación de E/S.

De hecho existe un cierto compromiso entre estas dos desventajas. Para ver esto, considérese la transferencia y un bloque de datos. Si se utiliza una E/S controlada por programa, la CPU está dedicada exclusivamente a la tarea de E/S y puede mover los datos con una relativa velocidad a costa de no hacer nada más. Por el contrario, si se emplea E/S por interrupción, se libera en cierta medida a la CPU a expensas de disminuir la velocidad de transferencia de E/S ya que el programa de servicios de la interrupción puede contener muchas instrucciones ajenas a la transferencia del dato en sí. Ambos métodos tienen pues un efecto adverso sobre la actividad de la CPU, sobre la velocidad de transferencia de entrada salida.

Cuando se mueven grandes cantidades de datos se necesita una técnica más eficaz, esta técnica es el acceso directo a memoria (DMA). El acceso directo a memoria, como su nombre no implica, es una interfaz que proporcionar transferencia E/S de datos directamente a y desde la unidad de memoria y un periférico, a trabes de un controlador de DMA ubicado en el periférico, liberando a la CPU del peso de las transferencias

3.1 EL CONTROLADOR DMA.

En el acceso directo a memoria (direct memory access) el controlador del periférico se comunica directamente con la memoria principal del computador, llevando todo el peso de la transferencia, sin intervención de la UCP. Por tanto, la transferencia elemental se realiza sin que se ejecute una instrucción en la UCP, siendo el controlador del periférico quien se encarga de generar las señales de control del correspondiente ciclo de memoria.

Como se verá, el DMA exige que el computador envíe una seria informaciones al controlador del periférico, para que éste sepa lo que tiene que hacer. Además al finalizar la operación este debe interrumpir a la UCP para avisarle de ello. Esto hace que este tipo transferencia sólo tenga sentido cuando se pida o se envíe al periférico un bloque de datos. En este sentido, el DMA incluye los dos tipo transferencia, el de bloque y la elemental. Se hablará, por tanto, de operación de DMA, para referirse a la transferencia de un bloque, y transferencia por DMA, para referirse al intercambio de una palabra.

Para poder hacer transferencias de acceso directo a memoria, el controlador del periférico deberá conocer las siguientes informaciones:

- Dirección de memoria principal, para poder generar, durante el acceso memoria, las necesarias señales de dirección.
- Tipo de operación: lectura o escritura.
- Número de datos a transferir.
- Dirección en el periférico.

Estas informaciones deberán estar en poder del mencionado controlador para que se pueda realizar las transferencias deseadas. Será misión de la fase inicialización de la operación de DMA el hacer llegar estas informaciones al controlador.

El acceso directo memoria da lugar a conflictos de acceso, puesto que varios clientes (UCP y controladores) puede generar ciclos de acceso simultáneos a la memoria, que es un recurso único. Es necesario disponer de un mecanismo de prioridad que resuelva estos conflictos.
Existen dos formas básicas de realizar el acceso directo memoria que se estudiarán a continuación:

- Por memoria multipuerta.
- Por robo de ciclo.

3.1.1 MEMORIA MULTIPUERTA.

El acceso directo memoria, el caso de que esta tenga varias puertas, es conceptualmente muy sencillo. La UCP se conectará a una de estas puertas, reservando las demás para la conexión de los controladores de los periférico con acceso directo. La figura 3.1 presenta esta solución.

Es evidente, que la memoria multipuerta permite que los periféricos dispongan de la memoria principal ninguna intervención de la UCP, que accede a este dispositivo de forma independiente, ya que los buses las señales de control de cada puerta son totalmente independientes.

Una memoria multipuerta suele estar dividida en varios bloques de memoria, que permiten accesos en paralelo. De esta forma, se consigue que las peticiones de acceso, que se reciben por cada puerta, se puedan tratar en paralelo, siempre que no direccionen a un mismo bloque de memoria.

El mayor inconveniente de la memoria multipuerta es su coste, puesto que, además de la lógica propia de cada puerta, debe tener un dispositivo de gestión de prioridades para resolver las colisiones, esto es, las peticiones simultáneas a un mismo bloque, concediendo un acceso y retardando las demás. Las colisiones suponen un retardo, sin embargo, empleando métodos adecuados de asignación de direcciones, se pueden reducir al mínimo las interferencias entre los accesos a memoria de la UCP y los periféricos, por lo que se consiguen grandes velocidades de operación.

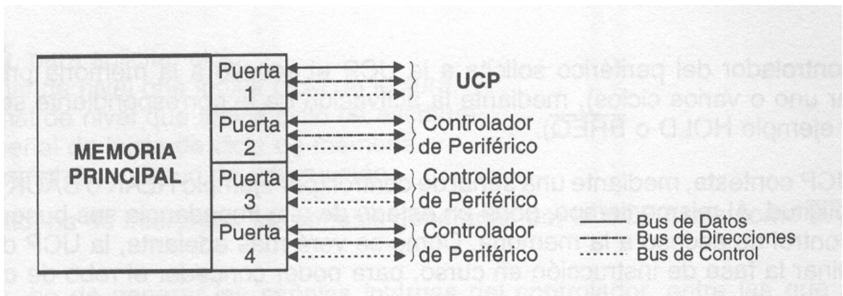


Figura 3.1 Acceso Directo a memoria multipuerta

Las señales que deben generar el controlador del periférico, para comunicarse con la memoria, son las siguientes:

- Dirección en memoria principal.
- Dato, si es un escritura.
- Inicio de ciclo de memoria.
- Lectura o escritura.

Las señales con las que contestará memoria son las siguientes:

- Dato, si es una lectura.
- Fin de ciclo de memoria.

Dado el coste de cada una de las puertas, se suele conectar en cada una de ellas varios dispositivos periféricos, de forma que el tráfico total que soporten se acerque a su máximo. El inconveniente de esta alternativa es que se pueden producir conflictos de acceso a una puerta, lo que obliga a tener un mecanismo adicional de prioridad en aquellas puertas que soporten acceso múltiple. Obsérvese, finalmente, que es necesaria una señal de sincronización entre la memoria y el controlador del periférico, puesto que la duración del ciclo de memoria es variable, dependiendo de la existencia o no de colisión. La señal de fin de ciclo cumple la misión de sincronización, indicando controlador del dato está disponible o que ha terminado la escritura.

3.1.2 ROBO DE CICLO.

El robo de ciclo es una forma más económica que hacer acceso directo memoria. En este caso, la memoria tiene una sola puerta, que deben ser compartida por la UCP y por los periféricos.

La transferencia exige que la UCP y el controlador del periférico se pongan de acuerdo para tener el control de los buses y señales de control de la única puerta de la memoria. En general, el control pertenece a la UCP, que lo cederá cuando el periférico lo solicite, perdiendo (o dejándose robar) un ciclo o fase.

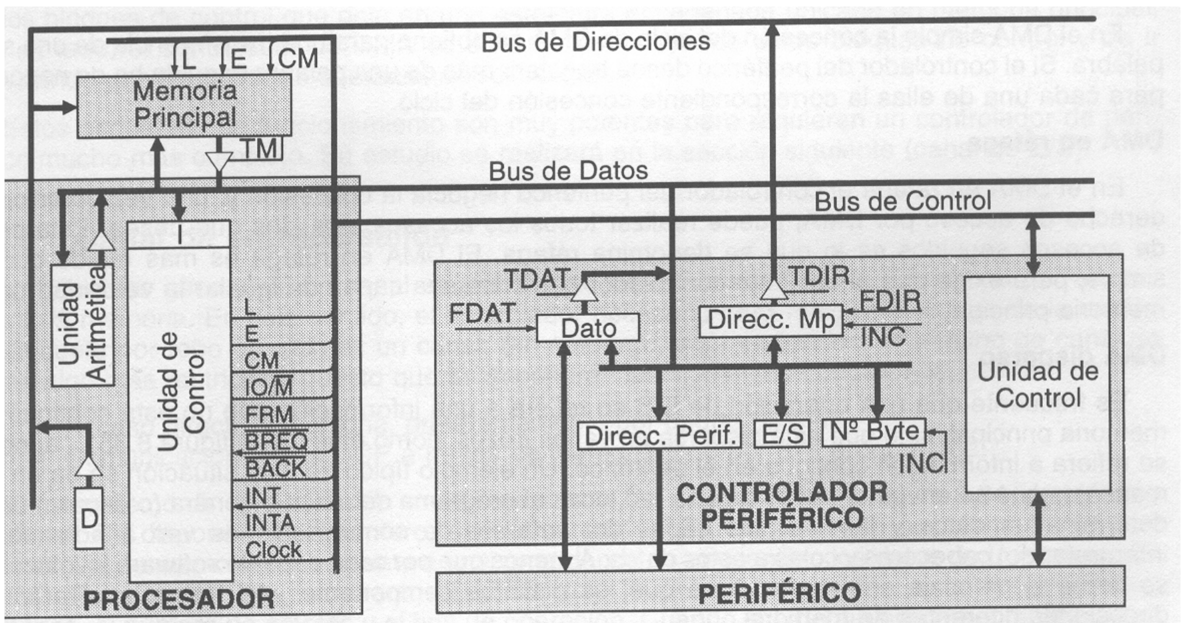


Figura 3.2: Acceso Directo a memoria por robo de ciclo

Los pasos seguidos son las siguientes:

- El controlador del periférico solicita a la UCP el acceso la memoria principal (solicita robar uno o varios ciclo), mediante la activación del correspondiente señal de control (por ejemplo HOLD).

- La UCP contesta, mediante una señal de control (por ejemplo HLACK o GNT), concediendo la solicitud. Al mismo tiempo, por en estado de alta impedancia sus buses y las señales de control de acceso la memoria. Como se verá más adelante, la UCP debe esperar a terminar la fase de instrucción en curso, para poder conceder el robo de ciclo solicitado.

- El controlador del periférico realiza los accesos a memoria principal, activando las señales de control y generando las direcciones correspondientes.

- Cuando el controlador del periférico ha terminado, devuelve a la UCP el control de los buses y demás señales, desactivando las señal de petición de ciclo (esto es, desactivando HOLD).

- Finalmente, la UCP desactiva BACK y recupera el uso de los buses.

Está claro que, aunque la transferencia elemental la realiza en su totalidad el controlador del periférico, hay una interferencia con la urgente, debería despegar, mientras éste hace su trabajo. La ejecución de las instrucciones sufrir una pequeña parada, cada vez que se solicite un robo de ciclo.

Generalmente del periférico que solicita el robo de ciclo será más lento que la UCP, por lo que va intercalando, esporádicamente, robos de ciclo para hacer transferencias individuales. Sin embargo, puede darse el caso de que, una vez obtenido el ciclo, el controlador del periférico realicé múltiples transferencia antes de desactivar la señal de petición. Un ejemplo de esta situación lo constituyen los controladores de vídeo que hacen robos de ciclo múltiples del tipo "ráfaga" (*burst*). En el resto del texto se consideran robo de ciclo simples, pero todo ellas fácilmente extrapolable al caso de las ráfagas.

La UCP sólo puede aceptar un robo de ciclo, esto es, pasar al estado de HOLD, cuando se encuentra al final de una fase. El controlador del periférico debe esperar a que la UCP termine la fase en curso para aceptarse en mitad de instrucción. Finalizando el robo de ciclo, la UCP sigue con la fase siguiente. El cronograma resultante se obtienen sin más que cortar, por la fase correspondiente, el cronograma normal de instrucción y separar ambas partes el tiempo que tarde el periférico en acceder a memoria.

Para estudiar el robo de ciclo, se utilizara el esquema de la figura 3.2, que muestra la conexión necesaria para hacer este tipo de acceso. Cabe destacar que el controlador del periférico tiene una serie de registros para almacenar la información de la transferencia, esto es, la dirección de memoria, la dirección en el periférico, el numero de octetos que quedan por transferir y el tipo de operación. El controlador debe ser capaz de ir incrementando el registro de dirección de memoria y de decrementar el registro de cuenta de octetos, para actualizar estas informaciones cada vez q realiza un acceso.
Los controladores de DMA son complejos, por las diversas funciones que han de realizar y por los señales de control que han de generar. En el ejemplo de la figura 3.2, ha de enviar a la UCP las siguientes señales:

- BREQ, para solicitar ciclo.
- L, señal de nivel que indica ciclo de lectura.
- E, señal de nivel que indica ciclo de escritura.
- CM, señal de inicio de ciclo de memoria.
- INT, señal de solicitud de interrupción.

Por otro lado, ha de interpretar las señales generadas por la UCP, tales como BACK, INTA, L, E, I0/M, etc.

Finalmente, ha de generar las señales internas del controlador, entre las que destacan las siguientes:

- TDAT, señal de nivel que abre el registro de datos del controlador al bus de datos.
- TDIR, señal de nivel que abre el registro de direcciones del controlador al bus de direcciones.
- FDAT, señal de flanco que carga, en el registro de datos del controlador, el contenido del bus correspondiente (para las operaciones de lectura).
- INC, señal que incrementa el registro de direcciones del controlador y decrementa la cuenta de octetos.

El controlador del periférico genera la señal BREQ, solicitando un robo de ciclo, a lo que contesta el computador con la señal de aceptación BACK. La señal BACK es utilizada por el controlador para activar TDIR, que pone en el bus de direcciones la dirección correspondiente a la posición de memoria deseada. Seguidamente, activa las señales necesarias para realizar el acceso a memoria. Finalizado éste, el controlador de periférico desactiva BREQ, con lo que el computador desactiva BACK y sigue con el ciclo siguiente.

3.2 ARRANQUE DEL ACCESO DIRECTO A MEMORIA.

Tal y como se ha indicado repetidamente, el controlador de acceso directo a memoria debe conocer las direcciones origen y destino, el sentido de la transferencia y el número de octetos a transmitir. Es misión de la UCP hacerle llegar estas informaciones, que denominaremos **bloque de control**.

La técnica más sencilla para hacer llegar el bloque de control al controlador del periférico consistente en dotar al mismo de una o más puertas de E/S programada. Basta con ejecutar un programa que contenga tantas instrucciones de salida como palabras tenga el bloque de control, con las direcciones asignadas a las puertas del controlador.

Una vez recibido el bloque de control, el controlador realiza, de forma independiente, los necesarios ciclos de acceso directo a memoria. Finalizada la operación de E/S, el controlador del periférico envía una señal de interrupción a la UCP, para avisar que dicha operación terminó y que queda disponible.

3.3 TIPOS DE DMA.

En esta sección se analizan las modalidades de DMA más corrientes.

3.3.1 DMA SIMPLE.

En el DMA simple la concesión del ciclo de DMA se obtiene para una transferencia de una sola palabra. Si el controlador del periférico desea transferir más de una palabra seguida ha de negociar para cada una de ellas la correspondiente concesión del ciclo.

3.3.2. DMA EN RÁFAGA

En el DMA en ráfaga el controlador del periférico negocia la concesión y, una vez obtenido el derecho de acceso por DMA, puede realizar todos los accesos seguidos que desee. Esta serie de accesos seguidos es lo que se denomina ráfaga. El DMA en ráfaga es más rápido que el simple, pero exige que el conjunto controlador-periférico sea capaz de igualar la velocidad de la memoria principal.

3.3.3 DMA DISPERSO

Es frecuente que una operación de E/S se refiera a una información que no está contigua en memoria principal, sino que está dispersa en varios trozos, aunque se refiera a información contigua en el periférico. Un ejemplo típico de esta situación se da en los mensajes que se envían a través de una red local: el programa de usuario genera (o espera) unos determinados datos y las distintas capas del *software* de comunicaciones van añadiendo (o interpretando) cabeceras y colas a estos datos. A menos que por cada capa de software involucrado se copie el mensaje entero, resultará que los distintos componentes del mensaje estarán en direcciones diferentes de memoria.

Según se ha visto, las operaciones de acceso directo permiten hacer la transferencia de un bloque de datos siempre que esté en direcciones contiguas tanto en memoria principal como en el periférico. Sin embargo, se puede sofisticar el controlador de DMA para que contemple operaciones que afecte a varios bloques de la memoria principal. Esta solución es adecuada cuando el número de estos bloques es conocido y es fijo, como es el caso de las comunicaciones antes mencionado.

El bloque de control para este tipo de DMA debe contener las direcciones de memoria principal, y los tamaños de todos los bloques que forman la operación.

Se utilizará el término de **recopilar** *(gather)* para expresar una escritura desde varios bloques de memoria a un periférico. Mientras que se utilizará el término de **esparcir** *(scatter)* para indicar la operación inversa, esto es, una lectura desde un periférico a varios bloques de memoria principal.

3.3.4 DMA GOBERNADO POR MEMORIA

Según se ha visto anteriormente, la UCP ha de preparar la operación de DMA generando el bloque de control que ha de enviar al controlador. Una forma de realizar este envío es escribiendo directamente el bloque de control en los registros del controlador mediante instrucciones de E/S. Otra forma, más sofisticada, consiste en dejar el bloque de control en una zona predeterminada de memoria principal. El controlador la leerá directamente de esta zona, por lo que el gobierno del mismo se realiza por memoria y no por E/S programada.

El esquema puede completarse añadiendo **encadenamiento.** En este caso la UCP genera varios bloques de control que deja en una estructura encadenada ubicada en memoria principal. Por su lado, el controlador de DMA se encarga de ir leyendo estos bloques de control y de ir ejecutando una tras otra las operaciones solicitadas.

Estos esquemas de funcionamiento son muy potentes pero requieren un controlador de periférico mucho más complejo. Su estudio se realizará en la sección siguiente (canal de E/S).

3.4 CONFIGURACIONES DEL DMA

El *DMA,* se puede configurar de diferentes formas. En las Figuras 3.3 a 3.5 se muestran algunas posibilidades. En el primer caso (ver Figura 3.3) todos los módulos comparten el mismo bus del sistema. El controlador de *DMA,* que actúa como un sustituto de la *CPU,* utiliza *E/S* controlada por programa para intercambiar datos entre la memoria y un periférico a través del controlador de *DMA.*

Esta configuración, aunque puede ser muy económica, es claramente poco eficaz, ya que cada transferencia de una palabra consume dos ciclos del bus igual que con la *E/S* controlada por programa.

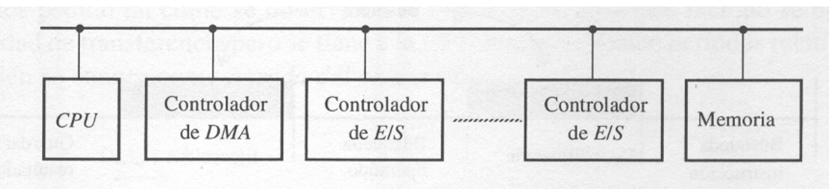


Figura 3.3: Bus único con DMA conectado

Se puede reducir sustancialmente el número de ciclos de bus necesitados integrando las funciones de *DMA* y *E/S.* Tal como se muestra en la Figura 3.4, esto significa que hay un camino entre el controlador de *DMA* y uno o más controladores de *E/S* que no incluyen al bus del sistema. La lógica del *DMA* puede ser una parte de un controlador de *E/S* o puede ser un módulo independiente que controla a uno o más controladores de *E/S.*

El concepto anterior se puede generalizar si se utiliza un bus de *E/S* para conectar los controladores de *E/S* al controlador de *DMA* (ver Figura 3.5). Este método reduce a una el número de interfaces de *E/S* en el controlador de *DMA* y proporciona una configuración fácilmente ampliable.

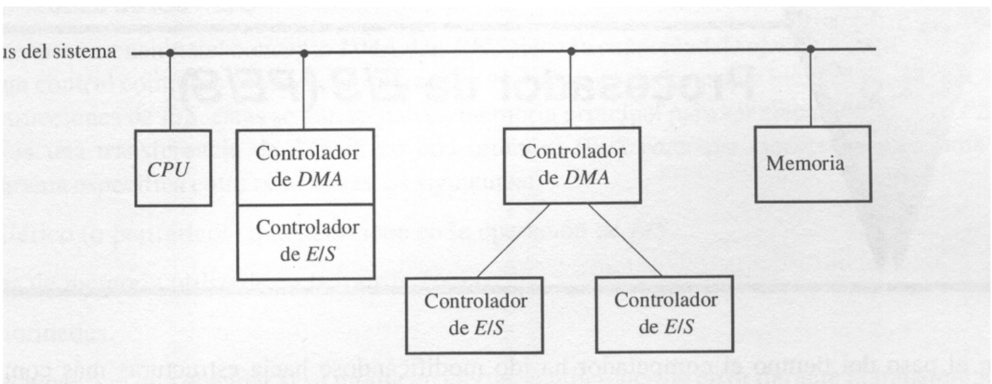


Figura 3.4: Bus único con DMA integrado

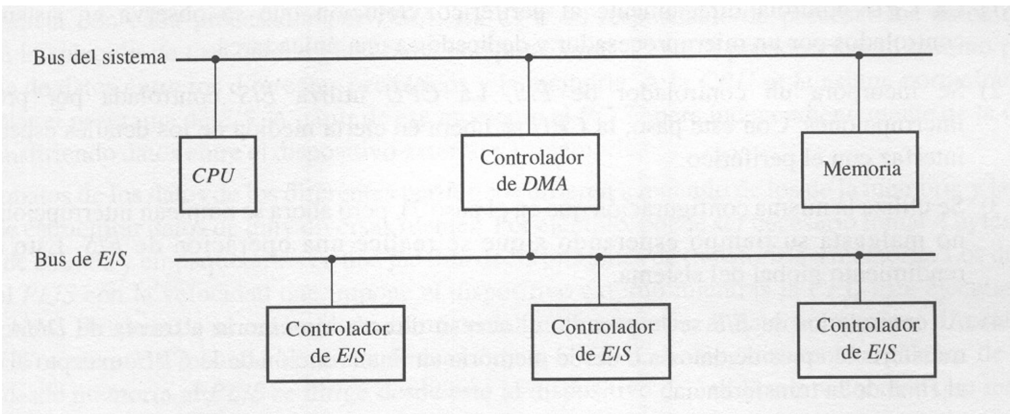


Figura 3.5: Controlador de DMA conectado al bus de E/S

En todos los casos el bus del sistema, que el controlador de *DMA* comparte con la *CPU* y la memoria, lo utiliza únicamente para intercambiar datos con la memoria. Sin embargo el flujo de datos entre el *DMA* y los controladores de *E/S* tienen lugar fuera del bus del sistema sólo en las dos últimas situaciones.

3.5 CANAL DE ENTRADA/SALIDA

En términos generales, un canal es una unidad de E/S automática que funciona por acceso directo a memoria.

El mecanismo de comunicación suele organizarse de la forma siguiente: Se dispone de un puntero, que llamaremos PCANAL, ubicado posiblemente en una posición fija de memoria, que apunta a una zona donde se encuentran los datos de una operación de E/S. Estos datos, que llamaremos DATES, son: la dirección en memoria, la dirección en el periférico, el número de octetos y el tipo de operación. Cuando la UCP ha preparado estos datos, envía, mediante una única instrucción de E/S, una señal al canal, para indicarle que tiene que realizar una operación.

Seguidamente, el canal accede directamente a memoria y, a través de PCANAL, obtiene los datos DATES de la operación a realizar. Finalizada ésta, el canal inserta en la correspondiente zona DATES el balance de la operación, e interrumpe para indicar que ha terminado. Este balance informa del desarrollo de la operación, indicando si se ha realizado correctamente o si se ha producido algún error.

Dado que un computador suele disponer de más de un canal, habrá que hablar de una tabla de punteros PCANAL, o de una tabla de datos DATES.

Obsérvese que el mecanismo de comunicación propuesto, si bien reduce la duración de los programas de iniciación y finalización de la operación de E/S, no los elimina, por lo que la reducción de tiempo obtenida no es demasiado importante. Veamos seguidamente como se puede mejorar esta situación, encadenando las operaciones de E/S.

3.6 ENCADENAMIENTO DE OPERACIONES DE *E/S*.

El encadenamiento de las operaciones de E/S se puede realizar sin más que añadir, al final de la zona DATES, una nueva zona DATES o una indicación de fin. El canal, al terminar la operación correspondiente a DATES1, introduce el balance de la operación en DATES 1 y lee DATES2, para seguir trabajando. El proceso se repite hasta que el canal encuentra una indicación de fin.

Cada DATES se compone de las siguientes informaciones:

- Orden. Tipo de operación a realizar tal como leer, escribir, rebobinar, grabar marca de inicio o fin, etc.
- Dirección en memoria. Es la dirección absoluta de la zona de memoria principal que se emplea en la transferencia.
- Número de octetos. Indica el tamaño de la zona a transferir, generalmente expresada en octetos, aunque puede hacerse en palabras.
- Dirección en el periférico. Identificación del sector o del registro a transferir. Dado que un canal puede tener conectado más de un periférico, se debe incluir la identificación del dispositivo.
- Bit de interrupción. Si este bit está activado, el canal deberá interrumpir al finalizar la operación correspondiente.
- Bit de encadenamiento. Si este bit está activo, el canal deberá tomar la siguiente orden DATES y proseguir trabajando. En caso contrario deberá parar, puesto que este bit representa la orden de fin.
- Balance de la operación. Esta información la coloca el canal al finalizar la operación de E/S correspondiente, para indicar ala UCP que la operación ha terminado, así como las posibles incidencias.

Al conjunto de DATES se le suele llamar programa canal, especialmente en aquellos casos en que la variedad de operaciones es grande, permitiendo incluso bifurcaciones condicionales en programa canal. Obsérvese que la UCP va generando dinámicamente el programa canal, a medida que van surgiendo las necesidades de E/S. Por tanto, debe garantizarse que no existan conflictos de accesos a estas informaciones, por ejemplo, que la UCP añada un nuevo DATES y modifique el bit de encadenamiento del anterior, cuando ya ha sido leído por el canal, que, por tanto, finalizaría sin tener en cuenta el nuevo DATES añadido. Para ello la UCP se basará en las interrupciones y en las informaciones de balance, con los que puede conocer la operación que está tratando el controlador.

3.7 ESTRUCTURA DE LOS CANALES

Cada canal suele estructurarse mediante un bus común, al que se conectan diversos periféricos. Esta estructura se basa en consideraciones de coste y de frecuencia de uso. Como la circuitería e un canal es compleja y cara, deberá ser compartida por varios periféricos. Además, puesto que los periféricos no suelen estar operando continuamente, pueden compartir un canal.

Sin embargo, esta estructura presenta el inconveniente de que pueden existir, simultáneamente, canales parados y canales con lista de espera, lo que redunda en una pobre eficacia del sistema e E/S. Para evitar esta situación, algunos fabricantes han introducido el concepto de canal flotante, que no es más que un canal que se puede asignar a cualquiera de los periféricos. En este caso, las peticiones de operaciones de E/S se asignan, no a un canal determinado, sino al conjunto de ellos, que se van repartiendo el trabajo. Sin embargo, los canales flotantes exigen que todos los periféricos estén conectados a todos los canales, lo que requiere más circuitería, por lo que es más caro.

Dependiendo del tipo de periféricos para el que esté diseñado un canal, se puede hablar de canal simple o de canal multiplexor.

Un **canal simple** o **selector** está diseñado para periféricos de alta velocidad de transferencia, tales como discos y cintas. Solamente permite una operación al tiempo y tiene una elevada velocidad de transferencia.

Un **canal multiplexor** está diseñado para periféricos lentos, tales como impresoras, lectoras le documentos y terminales. Este tipo de canal permite realizar simultáneamente varias operaciones de E/S, puesto que se divide en varios subcanales, que trabajan de forma cíclica, atendiendo cada uno a un periférico.

3.8 PRESTACIONES DE LA E/S

En esta sección se analizan las prestaciones de la E/S desde dos puntos de vista. Primeramente se compara la velocidad de transferencia que se puede obtener con mecanismos de DMA frente a las operaciones programadas. Seguidamente se estudia, mediante un ejemplo, el tiempo de UCP utilizado en una operación de E/S según sea el mecanismo empleado.

3.8.1 VELOCIDAD DEL DMA FRENTE A LA E/S PROGRAMADA

Según se ha visto anteriormente las operaciones de E/S transmiten los datos entre el controlador del periférico y la memoria del computador mediante dos técnicas básicas: DMA o un bucle de instrucciones de E/S.

En esta sección se analizará la máxima velocidad de transferencia que se puede alcanzar mediante estos dos mecanismos. Para realizar este análisis se supondrá que el controlador del periférico sobre el que se hace la transferencia es muy rápido (p.e. es un controlador de comunicaciones con *buffer),* por lo que es capaz de recibir la información a la máxima velocidad que el computador sea capaz de suministrarla.

Se considerarán tres casos. DMA simple, DMA en ráfaga y E/S programada.

DMA simple

La operación de DMA exige lo siguiente:

- La inicialización del controlador, lo que puede exigir unas 10 instrucciones de máquina, de las cuales varias son de E/S.

- Por cada palabra transferida hay una fase de petición de bus más una fase de devolución.
- Al final de la operación de DMA el controlador interrumpe. El tratamiento de esta interrupción exige varias decenas de instrucciones de máquina.

DMA en ráfaga

Este caso es similar al anterior, pero la fase de petición del bus solamente se hace al comienzo de la operación y la devolución al final de la misma.

E/S programada

Se ha supuesto que el controlador de periférico es muy rápido, por lo que no se incluido en el bucle la típica comprobación de que está preparado, comprobación que se hace leyendo el estado. Por lo tanto, el bucle de envío se podría reducir a lo siguiente:

```
BUCLE: LD    .1,[.3++]  ;Carga el dato en el registro .1
       OUT   .1,[.4++]  ;Envía el dato al controlador
       DEC   .2                    ;Decrementa el contador de caracteres
       BNZ   $BUCLE  ;Repite si .2 1 0
```

Esto es, aparte de la preparación que debe incluir la carga inicial de los registros .2, .3 y .4, se necesita ejecutar 4 instrucciones por palabra enviada, de las cuales una es de E/S. La figura 3.3 muestra el tiempo que se tarda en la operación de escritura en función del tamaño de la información a enviar. Para el cálculo de la esta figura se han supuesto los siguientes tiempos: preparación de la operación de DMA: 450 ns, tratamiento de la interrupción final: 500 ns, lectura de memoria principal: 60 ns y concesión más devolución de bus: 30 ns. Para el bucle de E/S programada se ha supuesto que las instrucciones de E/S tardan 100 ns, mientras que las normales tardan 20 ns (gracias al uso de memoria caché).

Evidentemente, para otros supuestos de funcionamiento se obtienen otros tiempos distintos de los de la mencionada figura, sin embargo, cabe destacar dos hechos generales. Primero: para tamaños de unas pocas palabras es más rápido emplear E/S programada que DMA. Segundo: el DMA por ráfagas es más rápido que el simple; ahora bien, el controlador del periférico ha de ser suficientemente rápido para aceptar la cadencia que aquél impone.

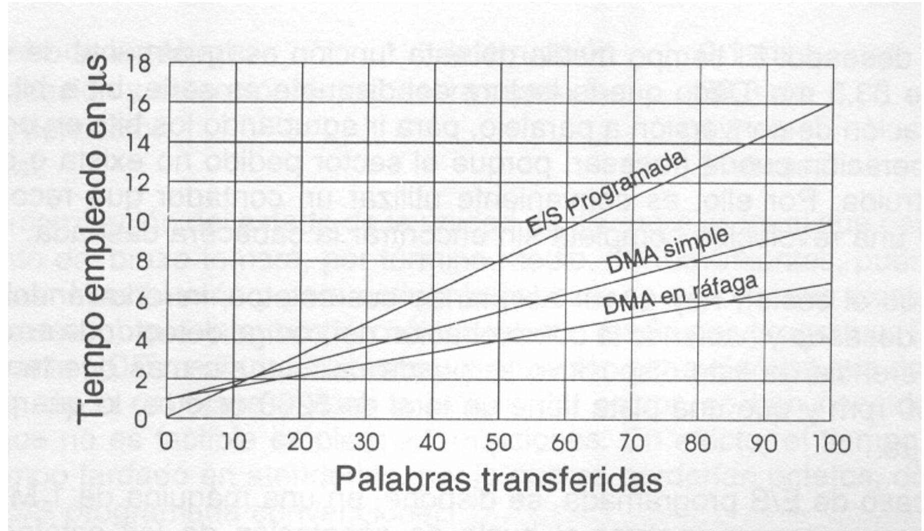


Figura 3.6: Velocidad de transferencia en función del tamaño

3.8.2 TIEMPO DE UCP EMPLEADO EN UNA DE OPERACIÓN DE LECTURA

Como ejemplo de las distintas formas de abordar una operación de E/S, y así poder evaluar las distintas alternativas, se ha seleccionado una unidad de disquetes de 8". Se han considerado varias alternativas de diseño. Aunque en la práctica no se empleen, por el reducido coste alcanzado por los controladores de disquete y de DMA, sin embargo, las alternativas planteadas siguen siendo válidas para otros dispositivos, y su estudio comparativo es de gran utilidad.

Se propone realizar una operación de lectura de un sector de 128 octetos de un disquete de 8 pulgadas. Las fases que se requieren en algunos de estos dispositivos son las siguientes:

- Primero, hay que comprobar el estado de la unidad de disquete. En concreto, hay que comprobar los siguientes aspectos:
  -Si está encendido
  -Si tiene disquete insertado
  -Si está desocupado
  Esta fase requiere leer la palabra de estado del periférico y analizarla, para detectar posibles problemas.
- Seguidamente, hay que enviar una orden para posicionar el brazo en la pista que contenga el sector deseado. Para ello, hay que conocer la posición actual del brazo y enviar tantos pulsos como pistas se tenga que avanzar o retroceder. En caso de no conocer la posición del brazo, hay que llevarlo hasta la pista cero. Esto se hace mediante un bucle que envía pulsos de avance y que comprueba la señal de estado de pista cero. Una vez detectada esta posición, se avanza el brazo hasta la pista deseada. La detección de pista cero se realiza mediante un fotodiodo que se activa cuando el brazo alcanza la posición final. Se tomará un tiempo medio de acceso a la pista deseada de 100 ms.
- Posicionado el brazo, hay que activar el electroimán de la cabeza lectora, para que se apoye sobre el disquete. Se supondrá un tiempo de 30 ms para esta operación.
- Activada la cabeza, hay que conseguir el sincronismo con el flujo de señales leídas. Esta función se realiza al leer la primera cabecera de sector. Seguidamente, hay que ir reconociendo los octetos leídos, detectando las cabeceras de los sectores, hasta encontrar el sector deseado. El tiempo medio de esta función es igualmente de media revolución, eso es de 83,3 ms. Dado que la lectura del disquete es serie, bit a bit, hay que realizar una operación de conversión a paralelo, para ir agrupando los bits en octetos. Obsérvese que la operación puede fracasar, porque el sector pedido no exista o su cabecera haya sido destruida. Por ello, es conveniente utilizar un contador que reconozca que se ha realizado una revolución completa sin encontrar la cabecera deseada.
- Encontrado el sector, hay que ir aceptando sus octetos, introduciéndolos en la zona de memoria deseada y haciendo la comprobación del código detector de errores. La velocidad de transferencia de este dispositivo se puede calcular sin más que tener en cuenta que gira a 360 rpm y que una pista tiene un total de 5208 octetos, lo que produce un octeto cada 32 μs.

Para el caso de E/S programada, se dispone, en una máquina de 1 MIPS, del orden de 32 instrucciones para realizar el bucle de aceptación de los octetos, lo que permite perfectamente ir comprobando su paridad y almacenándolos en memoria. Incluso se podría ir calculando sobre la

marcha el código polinomial de detección de error del sector, procedimiento que ya no sería posible para un disquete de doble densidad de grabación, puesto que sólo se dispondría del orden de 16 instrucciones por octeto. Ello obligaría, en el caso de emplear E/S programada, a realizar esta comprobación posteriormente.

Una vez vistas las que podrían ser las fases de lectura de un sector, se pasará a analizar las distintas alternativas de distribución del trabajo entre el controlador del disquete y la UCP. Nótese que las fases de toda operación de E/S dependen del periférico, por lo que la secuencia anterior sólo debe tomarse como un ejemplo y no como caso general, ni siquiera para las unidades de disquete de 8".

***E/S* programada**

Dada la velocidad de la unidad de disquete, cualquier computador actual es capaz de realizar todas las funciones por E/S programada. En este caso, los tiempos empleados por la UCP serían los siguientes:

t1: Comprobación del estado:	0,1 ms
t2: Posicionamiento del brazo:	100 ms
t3: Activación de la cabeza:	30 ms
t4: Sincronización y detección de cabecera deseada:	83,3 ms
t5: Lectura del sector:	4,2 ms
Total:	217,6 ms

Para determinar t1, se ha supuesto que la comprobación del estado requiere, como mucho, ejecutar un pequeño programa de unas 100 instrucciones de máquina. Para los tiempos t2 a t5 se han empleado sus tiempos medios. Finalmente, para t6 se ha tomado el tiempo de leer los 131 octetos del sector, lo que multiplicado por 32 da aproximadamente 4,2 ms. El total de la operación es, por tanto, como media del orden de 218 ms, estando la UCP ocupada durante todo este tiempo.

El tiempo medio de UCP empleado por octeto es de 217,6/128 = 1,7 ms.

Mediante interrupciones

Se considerará, en este caso, que el funcionamiento es por interrupciones, pero que la UCP realiza todas las funciones arriba mencionadas. Habrá que calcular el número de interrupciones que ha de atender el procesador, así como el tiempo de tratamiento de cada una de ellas. Se supondrá un tiempo medio de la rutina de interrupción de 0,1 ms, obteniéndose los tiempos totales siguientes:

t1: Comprobación del estado:	0,1 ms
t2: Posicionamiento del brazo:	39 interrupciones 3,9 ms
t3*:* Activación de la cabeza:	1 interrupción 0,1 ms
t4*:* Sincronización y detección de cabecera deseada:	83,3 ms
t5: Lectura del sector:	4,2 ms
Total	91,6 ms

La fase de comprobación del estado de la unidad de disquete es igual que para el caso anterior. El posicionamiento del brazo tomará, por término medio, 39 movimientos, puesto que existen 77 pistas. Para realizar esta función y para activar la cabeza, se supone que existe un temporizador externo (o reloj externo) que interrumpe con un retardo fijado por programa. En caso contrario, debería temporizar la UCP mediante un bucle y la situación sería idéntica a la de E/S programada. Para la detección de la cabecera y para la lectura del sector, se han considerado los tiempos totales, puesto que no es factible emplear interrupciones. En efecto, el tiempo entre éstas sería menor que el tiempo tardado en atenderlas, por lo que se perderían octetos, debiendo realizarse estas fases por E/S programada convencional.

El ahorro de tiempo de UCP es apreciable, obteniéndose un gasto de 0,7 ms por octeto.

Interrupciones con controlador inteligente

La disponibilidad de controladores integrados a bajo coste hace que prácticamente todas las unidades comerciales empleen este tipo de dispositivo, pudiéndose hacer la transferencia de octetos por E/S programada, como se contempla en este apartado, o por DMA, como se contempla en el siguiente.

Se considerará, en este caso, que la unidad de disquete está dotada de un controlador inteligente, que es capaz de realizar las funciones de posicionar el brazo, de activar la cabeza y de buscar el sector deseado. La UCP solamente deberá lanzar la operación de lectura, enviando al controlador, mediante E/S programada, la identificación del sector, su tamaño y la orden de lectura. Todo ello puede hacerse con un sólo programa que, primero, compruebe el estado de la unidad de disquete y del controlador, y que, a continuación, envíe la información necesaria para éste. Seguidamente, el controlador posiciona el brazo, activa la cabeza y detecta el inicio de pista, pasando, acto seguido, a la lectura de las cabeceras, hasta que encuentra el sector deseado. En ese momento interrumpe ala UCP que, a partir de entonces, se encarga del resto de la operación de lectura, de forma idéntica a como hiciera en los dos casos anteriores.

Los tiempos empleados serían los siguientes:

t1: Comprobación del estado:	0, 15 ms
t2: Posicionamiento del brazo:	0 ms
t3: Activación de la cabeza:	0 ms
t4: Sincronización y detección de cabecera deseada	0 ms
t5: Lectura del sector:	4,2 ms
Total	4,5 ms

Se aprecia una importante reducción en el tiempo total pe UCP requerido, con una media de 33,6 µs por octeto.

Optimizando la rutina de tratamiento de la interrupción del controlador, daría tiempo, incluso, para comprobar el estado del controlador y conocer si la búsqueda del sector ha sido satisfactoria antes de que llegue el primer octeto.

Robo de ciclo

El empleo de un controlador inteligente y de robo de ciclo corresponde a la solución que se emplea realmente en la práctica. En este caso, el controlador de la unidad de disquete, además de realizar las funciones anteriores, es capaz de enviar el sector deseado, empleando una serie de accesos directos a memoria por robo de ciclo. Por tanto, el tiempo t6 queda reducido a los 128 robos de ciclo, por lo que, asignando un tiempo de 300 ns por robo de ciclo, queda t6 = 38.4 µs. Nótese que si la longitud de palabra de la memoria fuera de p octetos este tiempo se podría dividir por p, sin más que dotar al controlador de la lógica necesaria para poder agrupar cada p octetos en una palabra y hacer la transferencia elemental al nivel de palabra y no de octeto.

Finalizada la transferencia del sector, el controlador de la unidad de disquete interrumpe ala UCP, para indicar que la operación ha terminado e informar de los posibles errores o problemas encontrados. Nótese que en las soluciones anteriores quien detectaba los errores era directamente el programa ejecutado, sin embargo, en este caso toda la operación la hace el controlador, por lo que es quien detecta los errores, lo que ha de comunicar al computador. Suponiendo que el tratamiento de esta interrupción es de 100 instrucciones, quedan los tiempos de ocupación de UCP siguientes:

t1: Comprobación del estado:	150 µs
t2: Posicionamiento del brazo:	0 µs
t3: Activación de la cabeza:	0 µs
t4: Sincronización y detección de cabecera deseada:	0 µs
t5: Lectura del sector:	38,4µs
t6: Tratamiento final:	100 µs
Total:	288,4 µs

El tiempo total de UCP es fundamentalmente el empleado en iniciar y terminar la operación de E/S, más un pequeño tiempo perdido por los robos de ciclo. La media obtenida es, en este caso, de 2,25 µs por octeto.

La figura 3.7 muestra gráficamente estas cuatro alternativas. Aunque es imposible hacer una representación a escala, puesto que se deben representar tiempos de cientos de milisegundos y de unos pocos microsegundos, la figura 3.7 da una buena idea de como se reparte la ocupación de la UCP. Obviamente, las técnicas de canal de E/S permiten reducir aún más el tiempo de UCP.

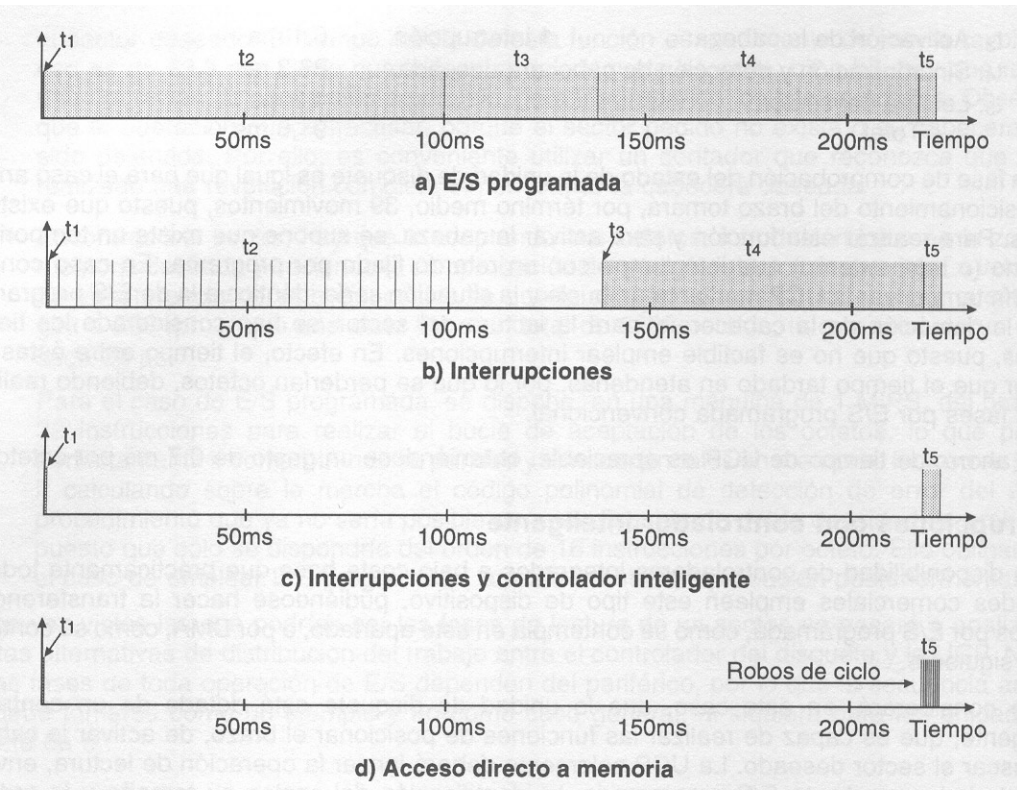


Figura 3.7: Asignación de tiempo de CPU según la organización de E/S

CAPÍTULO 4

La tarjeta PCI Proto-Lab/Plx.

El desarrollo de una placa de expansión basada en el Bus PCI es relativamente complejo y, normalmente, necesita de una gran inversión de tiempo y dinero antes de conseguir resultados aceptables. La tarjeta prototipo para Bus PCI presentada en este capítulo ofrece una solución mejor. Su descodificador PCI preconfigurado y su bus de datos de 32 bits de ancho, junto con una serie de puertos direccionados, nos permiten centrarnos en el desarrollo y en las pruebas del circuito prototipo sin necesidad de entrar en detalle de cómo configurar el Bus PCI estándar.

Desde que aparecieron los PC´s compatibles IBM en el mercado, cada vez que se ha producido un avance importante en el desarrollo de los mismos, se ha introducido un nuevo bus de sistema. El Bus PCI, que representa el estado actual de la situación, no esta limitado a varias generaciones de PC´s, gracias a su alto nivel de prestaciones. El precio que se paga por ello es su estructura, relativamente compleja.

En el capítulo presente, se describirá el funcionamiento y las características de la tarjeta de prototipo PCI Proto- Lab/PLX de la compañía HK Meßsysteme GmbH.

PCI-Proto LAB/PLX es una eficiente ayuda para desarrollo de aplicaciones para ordenadores personales y otros sistemas que se equipen con un Bus PCI.

PCI-Proto LAB/PLX trabaja con el PCI9054 de PLX Technology, Inc., que controla todos los modos típicos de operación PCI y cumple con la revisión de la especificación V2.2 del Bus PCI.

4.1 VISTA RÁPIDA A LOS COMPONENTES DE LA TARJETA PCI PROTO-LAB / PLX.

Esta tarjeta contiene los siguientes elementos:

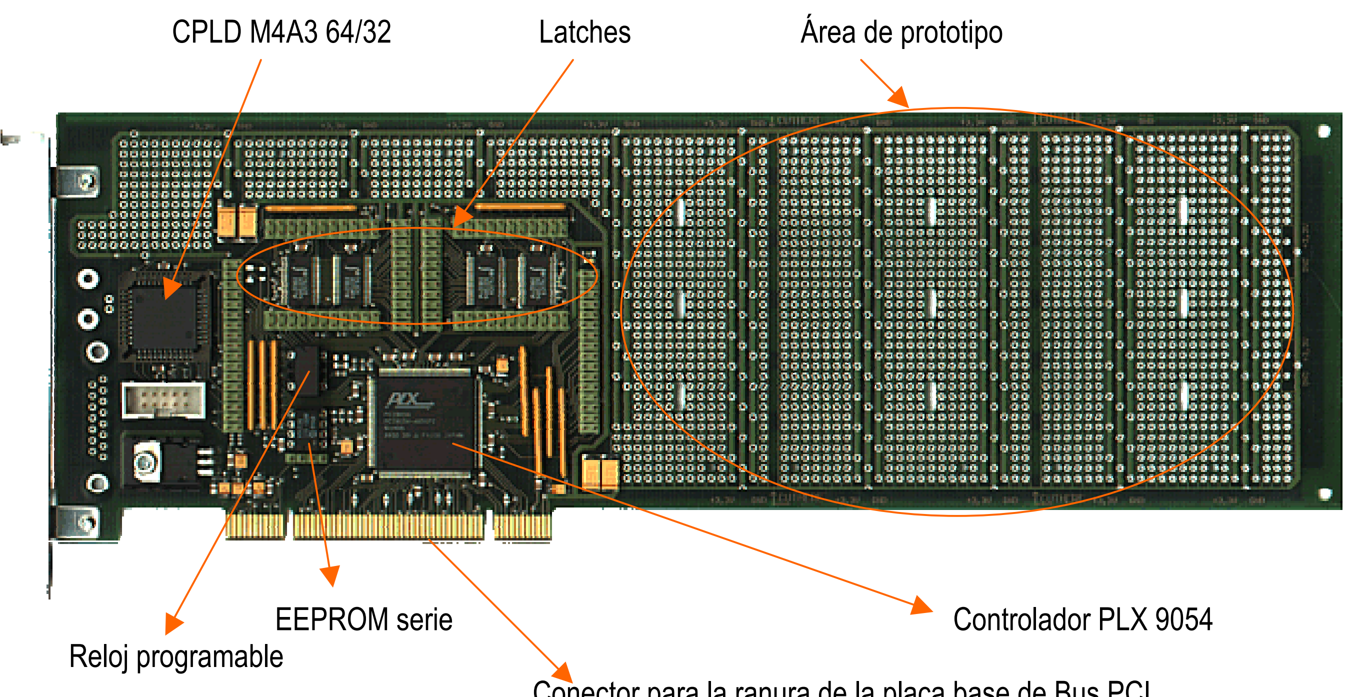
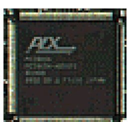


Figura 3.1: Tarjeta prototipo PCI Proto_Lab/PLX.

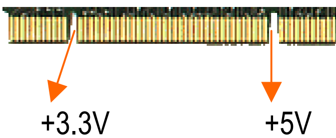
PCI-Proto LAB/PLX es una tarjeta de Bus PCI fabricada en cuatro capas, que puede ser cortada en el lugar marcado para transformarla en una tarjeta corta. Nosotros usamos el área de prototipo para sujetar la FPGA conectada al bus local.

Controlador PLX 9054
Corazón de la tarjeta de desarrollo que realizará la conversión de las señales desde el Bus PCI al Bus Local y viceversa, dependiendo del modo de transferencia elegido. (especificación PCI).



Conector para la ranura del Bus PCI.
PCI-Proto LAB/PLX se ha diseñado como tarjeta universal para soportar operaciones de Bus PCI de 3.3V o 5V.

El voltaje de funcionamiento para todos los circuitos electrónicos es 3,3V, aunque PCI-Proto LAB/PLX también puede trabajar en ranuras viejas del PCI con 5V de alimentación, poniendo o quitando los puentes soldados para utilizarlo en modo de 5V o de 3.3V como se puede observar en la ranura del PCI.

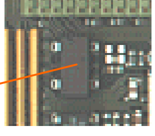


Dispositivo lógico programable CPLD M4A3 64/32.
Este dispositivo servirá para simular un hardware conectado a nuestro Bus Local. Este posee un ejemplo cargado que pretende escribir y leer datos de 32 bits en y desde el Bus de Usuario.

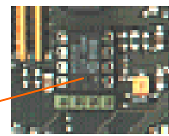
Latches.
La tarjeta posee 4 latches de 16 bits cada uno, 2 de entrada y 2 de salida, que están conectados entre la CPLD y el Bus de Usuario para realizar las simulaciones cargadas en la CPLD.

latches de salida. latches de entrada.

Oscilador programable.
Este permitirá cambiar el reloj de Bus Local, independientemente del Bus PCI. Este reloj es programable de 0 a 50 MHz.



EEPROM
Instalada en la tarjeta para almacenar la configuración del controlador PLX, de forma que cada vez que apliquemos tensión a la tarjeta. Esta configuración será cargada automáticamente en el 9054.



La tarjeta tiene los taladros realizados para equiparla con un conector de 15 contactos sub-D y un conector BNC.

Tipo	Tarjeta Universal PCI 32bit (compatible con sistemas de 3.3V y 5V)
Interfaz de Bus	Basado en la Especificación Revisión 2.2 del Bus Local.
Controlador PCI	PLX PCI9054, 33Mhz
Tamaño	12.28 pulgadas x 4.17 pulgadas 6.85 pulgadas x 4.17 pulgadas
Área de la tarjeta Bread	25.1 pulgadas^2
Alimentación	+3.3 V, +5 V, ±12 V, ajustables mediante las jumpers soldados.

Tabla 4.1: Características principales de la tarjeta PCI Proto–Lab/ PLX.

4.2 HARDWARE.

Como introducción, al hardware de la placa se presenta el diagrama de bloques de la figura 3.2. Funcionalmente, el hardware se puede analizar en tres partes:

- Controlador de PCI,
- EEPROM serie,
- Aplicación de ejemplo con 32 bit latches y EPLD.

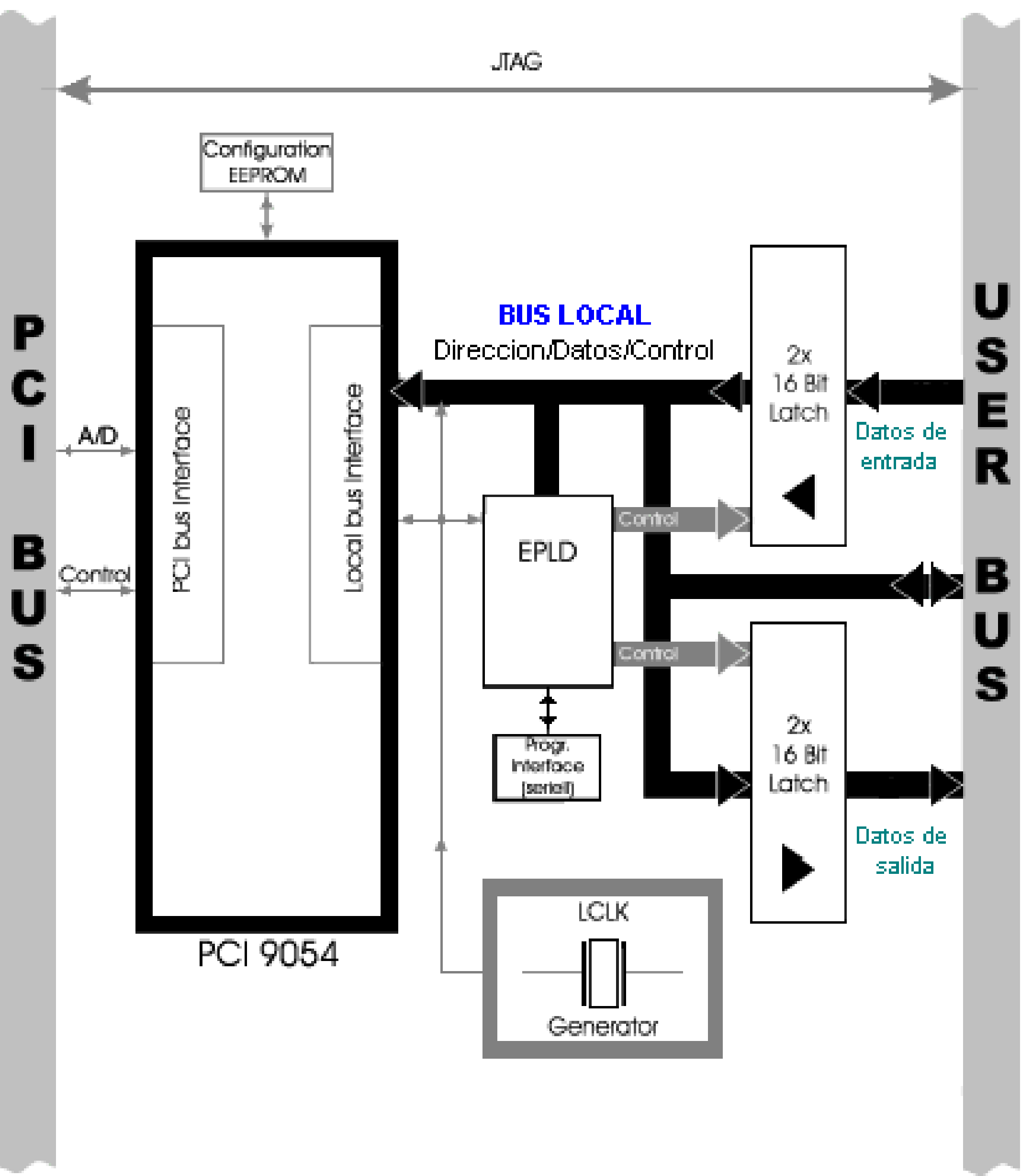


Figura 4.2: Diagrama de bloques de la tarjeta de prototipo.

El funcionamiento en la tarjeta PCI Proto LAB/PLX es el siguiente:

El controlador se comunica por un lado mediante la interfaz de Bus PCI intercambiando datos y direcciones multiplexados y señales de control. Por el otro lado, y mediante el interfaz de Bus Local, se comunica con este intercambiando datos, direcciones y señales de control. La configuración del controlador se realizará por medio de la EEPROM serie cada vez que se inicie la tarjeta.
La tarjeta posee los pines de acceso a las señales de datos, direcciones y control del Bus Local antes citados.

También podemos observar la presencia de una CPLD(1) conectada al Bus Local por un lado, y por el otro al Bus de Usuario mediante unos latches, que los divide en datos de entrada y salida, como muestra la figura 4.2. La función de esta CPLD junto con los latches de 32 bits de entrada y 32 bits de salida, es poder realizar la simulación de un sistema hardware que conectemos al área de prototipo a través del Bus de Usuario, añadido por el fabricante para ello.

Por tanto, tenemos la presencia de dos buses con los que poder trabajar. Por un lado el Bus de Usuario, que nos ayudará a realizar la simulación de un hardware implementado en la CPLD. Y por otro lado el Bus Local en donde conectaremos el diseño hardware que realicemos, para ello extraeremos la CPLD, deshabilitando el Bus de Usuario.

La tarjeta también posee un reloj programable desde 0 a 50 MHz para sincronizar las operaciones del Bus Local, independientemente del reloj del Bus PCI.

Mediante los pines JTAG podremos testar ambos buses, el PCI y el Local.

Como se puede observar en la figura 4.1 la tarjeta dispone de un área de prototipo donde podremos realizar el sistema hardware necesario para llevar a cabo nuestra aplicación. Por ser esta la primera aplicación realizada para nuestra tarjeta, y no querer estropear esta área, para futuras modificaciones, montaremos la tarjeta XS40 encima de esta área, mediante unas tuercas.

4.2.1 CONTROLADOR PCI.

El controlador conectado a nuestra tarjeta, PCI 9054 de tecnología PLX, es el corazón de la tarjeta del prototipo y forma el puente entre el bus PCI y los circuitos de usuario. Tiene dos interfaces para este propósito:

- el interfaz de Bus PCI.
- el interfaz de Bus Local.

La tarjeta se inicializa de tal forma que el PC reserva un espacio de memoria que será utilizado para las operaciones sobre el Bus. Existen 4 regiones de direccionamiento de entrada/ salida y de memoria:

(1) Dispositivo lógico programable complejo

Región	Acceso	Descripción
Región 0 PCI.	Memoria mapeada.	Usada por el acceso del Bus PCI para los registros Locales, de tiempo de ejecución y DMA.
Región 1 PCI.	I/O mapeada.	Usada por el acceso del Bus PCI para los registros Locales, de tiempo de ejecución y DMA.
Región 2 PCI.	I/O mapeada.	Rango de direcciones de 16 bytes, correspondientes con el espacio 0 de dirección local.
Región 3 PCI.	Memoria mapeada.	Rango de direcciones de 16 bytes, correspondientes con el espacio 1 de dirección local.

Tabla 4.2: Posibles accesos a las regiones de dirección.

El rango del Bus PCI de ambos espacios locales (0 y 1) se inicializan a 32 bits.

Es posible tener acceso a los latches instalados en la tarjeta con la aplicación de ejemplo, esto se describirá en un apartado posterior, con comandos de lectura o escritura de 32, 16 y 8 bits, accediendo a los latches, con comandos de I/O, usando la región 2 PCI o con comandos de memoria, usando la región 3 PCI.

Para lectura/ escritura de datos de 32 bits se utilizará la dirección offset 0, para lectura/ escritura de datos de 16 bits la dirección offset 0 o 2 y para lectura/ escritura de datos de 8 bits puede usarse la direcciones 0, 1, 2 o 3.

PCI-Proto LAB/PLX tiene su propio número de identificación del vendedor, de identificación del dispositivo, identificación secundaria del vendedor e identificación del subsistema. La identificación del vendedor (10B5h) y la identificación del dispositivo (9054h) es dada por el PCI-SIG(1) a PLX Technology para diferenciar entre los distintos componentes que salen al mercado y no debe ser cambiada. La identificación secundaria del dispositivo (9054h) fue editada por PLX Technology para el tipo usado de controlador PCI entre los que posee la compañía y tampoco se debe cambiar. La identificación el subsistema (2263h) fue escrita específicamente para el producto PCI-Proto LAB/PLX.

La figura 4.3 muestra la arquitectura interna del controlador, la cuál se explicará con detalle en el capítulo dedicado al controlador.

4.2.1.1 Interfaz de Bus PCI.

El interfaz de Bus PCI sirve para unir el controlador al bus PCI. En PCI-Proto LAB/PLX, el interfaz de bus PCI esta totalmente conectado por el conector PCB, por lo que no deberemos de realizar ningún esfuerzo en la conexión con este Bus.

[1] PCI-SIG: Special Interesting Group del PCI.

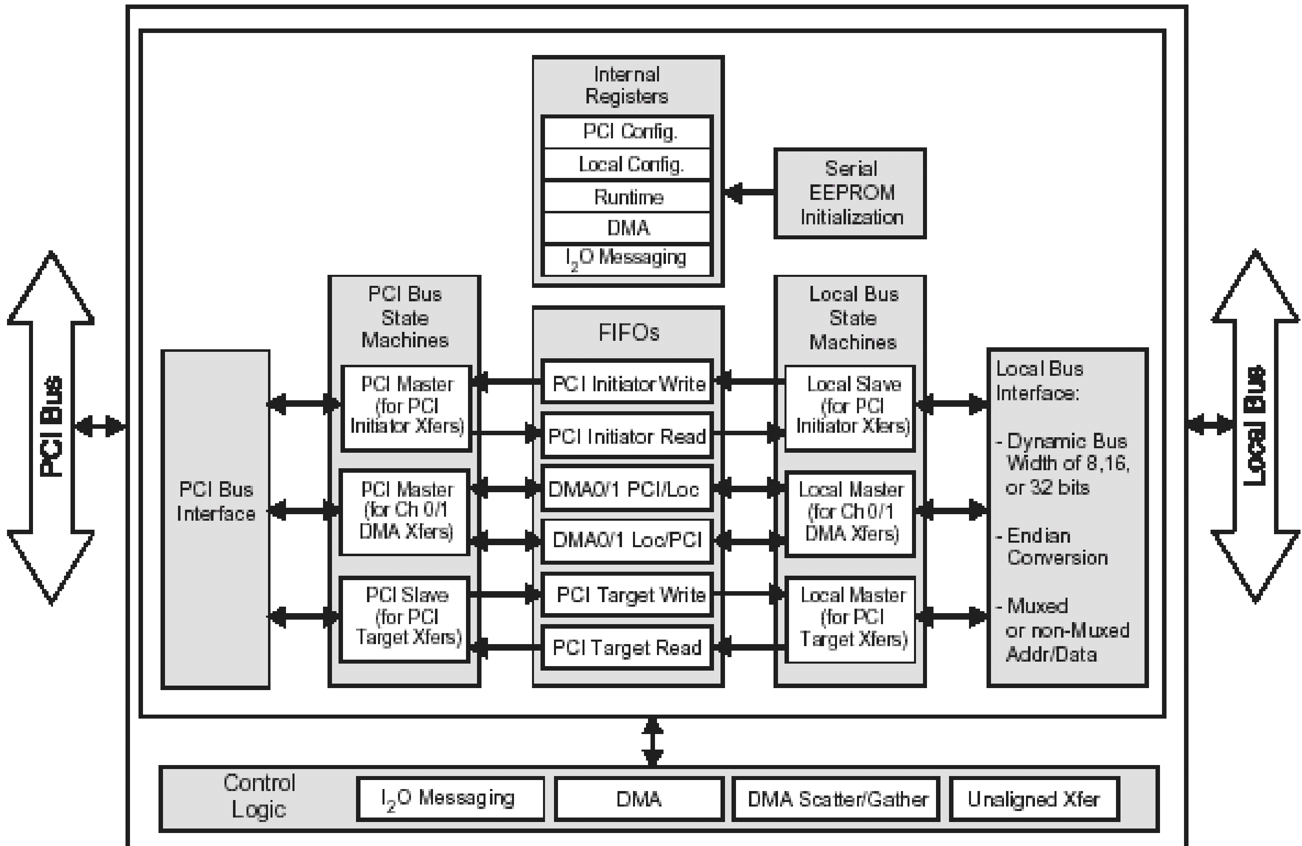


Figura 4.3 Diagrama de bloques interno del controlador 9054.

4.2.1.2 Interfaz de Bus Local.

El interfaz de Bus local es al que deberemos prestar mayor atención, debido a que este es donde se conectarán las aplicaciones. Permite la operación de periféricos hardware con datos de 8, 16 o 32 bits. El interfaz de Bus Local utiliza hasta 32 bits de datos locales y hasta 32 bits de direccionamientos locales latcheados.

El control de sistema específico de usuario, maneja el intercambio de datos con eficacia, a través del interfaz de Bus Local con la ayuda de las clásicas señales LHOLD, HOLDA, WAIT#, READY# o LW/R#, etc.

Los periféricos que se conecten con ella pueden consistir desde un sistema microprocesador o, en el caso más simple, en un latch de datos. En nuestro caso conectaremos un sistema formado por una FPGA y una memoria SRAM.

La arquitectura del controlador PCI también utiliza la integración de memorias, como se explicó en el apartado 3.2.1. Existen dos espacios de direccionamiento local, siendo posible direccionar hasta 4 Gbytes de memoria para cada espacio uno.

PLX9054 tiene un conjunto de registros para salvar datos de inicialización, para efectuar las configuraciones, activar y desactivar los modos de funcionamiento y para intercambiar datos. Es posible tener acceso a estos registros tanto desde el Bus PCI como del Bus Local. Esto significa que las funciones del controlador PCI se pueden utilizar desde ambos lados. En nuestro caso, debido a que la tarjeta que contiene la FPGA que conectamos al bus Local tiene un ancho de bus de tan sólo 8 bits, no podremos acceder a estos registros desde el mismo, ya que sólo se puede acceder a los registros de la tarjeta con palabras completas de 32 bits. Pero podemos seguir accediendo a los mismos desde el bus PCI mediante las funciones de las que se dispone.

También posee dos DMAs independientes, cuyo inicio de dirección y contador de transferencia se pueden ajustar, mediante los registros de transferencia de datos sin usar la CPU del ordenador principal (modo Maestro de transferencia).

Para la extensión de la BIOS del controlador se puede conectar memorias ROM con la interfaz paralela al PCI9054.

Se pueden generar interrupciones tanto desde el Bus Local como desde el Bus PCI.

En el capítulo 4, se describirá con más detalle el controlador PLX 9054.

4.2.2 EEPROM SERIE.

PCI-Proto LAB/PLX utiliza un EEPROM serie con una tamaño de 2 Kbits que está instalada en la misma tarjeta.

La EEPROM serie se usa en la fase de inicialización y contiene los datos de configuración que inicializan el controlador PCI, especialmente para la aplicación de PCI-Proto LAB/PLX. Es posible leer, corregir y escribir el contenido de la memoria EEPROM con la ayuda del programa PLXMon 200x del kit de desarrollo PLX Host SDK.

Es posible variar muchos de los registros que escribe la EEPROM una vez que estos son cargados. Incluso si esta no se encuentra, se puede configurar aceptablemente la tarjeta. En el capitulo dedicado al controlador se explicara más detenidamente como configurar los registros.

Offset EEPROM	Descripción	Bits de registro afectados.
0h	ID del dispositivo	PCIIDR[31:16]
2h	ID del vendedor	PCIIDR[15:0]
4h	Código de clase	PCICCR[23:8]
6h	Código/ Revisión de clase	PCICCR[7:0 / PCIREV[7:0]]
8h	Máxima latencia/ mínima concesión	PCIMLR[7:0] / PCIMGR[7:0]
Ah	interrumpir pin/ interrumpir línea de ruteo	PCIIPR[7:0] / PCIILR[7:0]
Ch	MSW de Mailbox 0 (definido por el usuario)	MBOX0[31:16]
Eh	LSW de Mailbox 0 (definido por el usuario)	MBOX0[15:0]
10h	MSW de Mailbox 1 (definido por el usuario)	MBOX1[31:16]
12h	LSW de Mailbox 1 (definido por el usuario)	MBOX1[15:0]
14h	MSW de rango para el espacio 0 de dirección PCI a Local	LAS0RR[31:16]
16h	LSW de rango para el espacio 0 de dirección PCI a Local	LAS0RR[15:0]
18h	MSW de dirección base local(Remapeo) para el espacio 0 de dirección PCI a Local Mailbox 0	LAS0BA[31:16]
1Ah	LSW de dirección base local(Remapeo) para el espacio 0 de dirección PCI a Local Mailbox 0	LAS0BA[15:0]
1Ch	MSW del registro de arbitración del modo DMA.	MARBR[31:16]
1Eh	LSW del registro de arbitración del modo DMA	MARBR[15:0]
20h	MSW de la dirección de escritura protegida de la EEPROM	PROT_AREA[15:0]
22h	LSW de la dirección de escritura protegida de la EEPROM	LMISC[7:0] / BIGEND[7:0]
24h	MSW de Rango de expansión ROM para PCI a Local.	EROMRR[31:16]
26h	LSW de Rango de expansión ROM para PCI a Local.	EROMRR[15:0]
28h	MSW de dirección base local (Remapeo) para la de expansión ROM para PCI a Local.	EROMBA[31:16]
2Ah	LSW de dirección base local (Remapeo) para la de expansión ROM para PCI a Local.	ERONBA[15:0]
2Ch	MSW de los descriptores de región de bus para PCI a Local.	LBRD0[31:16]
2Eh	LSW de los descriptores de región de bus para PCI a Local.	LBRD0[15:0]
30h	MSW de rango para PCI Initiator a Local.	DMRR[31:16]
32h	LSW de rango para PCI Initiator a Local.	DMRR[15:0]
34h	MSW de dirección base local para PCI Initiator a memoria PCI.	DMLBAM[31:16]
36h	LSW de dirección base local para PCI Initiator a Local.	DMLBAM[15:0]
38h	LSW de dirección base local para PCI Initiator a configuración I/O PCI.	DMLBAI[31:16]
3Ah	MSW de dirección base local para PCI Initiator a configuración I/O PCI.	DMLBAI[15:0]
3Ch	MSW de dirección base PCI para PCI Initiator a PCI.	DMPBAM[31:16]
3Eh	MSW de dirección base PCI para PCI Initiator a PCI.	DMPBAM[15:0]
40h	MSW de los registros de dirección de configuración para PCI Initiator a configuración I/O PCI.	DMCFGA[31:16]
42h	LSW de los registros de dirección de configuración para PCI Initiator a configuración I/O PCI.	DMCFGA[15:0]

Tabla 4.3 Registros que son cargados desde la EEPROM serie.

4.2.3 SOPORTE JTAG.

El diseñador de un dispositivo PCI puede implementar opcionalmente el test IEEE 1149.1 para permitir el testeo en el circuito del dispositivo. Estos pines deben operar al mismo voltaje que las señales del Bus PCI.

PCI-Proto LAB/PLX dispone de los pines con los que controlar estas líneas, también llamadas "JTAG fija ".

Pin	Nombre	Descripción
01	TDI	Test de datos de entrada. Usado junto a TCK para intercambiar datos e información dentro del puerto de acceso de test (TAP) en una cadena de bits serie.
02	TDO	Test de datos de salida. Usado junto a TCK para intercambiar datos e información fuera del puerto de acceso de test (TAP) en una cadena de bits serie.
03	TCK	Test de reloj. Usado para la información del estado de reloj y datos dentro y fuera del dispositivo
04	TMS	Test de selección de modo. Para controlar el estado del controlador del puerto de acceso de test.

Tabla 4.4: Pines JTAG accesibles en la tarjeta de prototipo.

Adicionalmente, aunque no sea accesible desde los pines JTAG de la tarjeta, existe el pin TRST#. Test de reset. Usado para forzar al puerto de acceso de test a un estado de inicialización.

4.2.4 CPLD Y LATCHES.

PCI-Proto LAB/PLX esta equipada con latches para almacenaje de datos de entrada-salida de 32 bits, desde el Bus de Usuario, de manera que podamos realizar la simulación de un hardware conectado a nuestra tarjeta antes de conectarlo a nuestro Bus Local, ver figura 4.2

Este dispositivo puede ser programado directamente en el circuito con un simple cable de conexión por el puerto paralelo del ordenador. Los textos originales de la CPLD(1) para el control de estos latches, se encuentran en el anexo.

La aplicación de ejemplo cargada en la CPLD, tiene como propósito simular cómo varios requisitos hardware (accesos a I/O y a memoria, activación de señales en alta y baja) conectados al Bus de Usuario, pueden ser ajustados. El ejemplo de la aplicación permite escribir y leer datos largas (32 bits de datos) sin requerir un hardware adicional.

Para este propósito, se ha implementado en la CPLD un decodificador simple de la señal de control que proporciona señales de control para escritura y lectura de datos, estos se dividirán entre datos de entrada y datos de salida del Bus de Usuario.

(1) Electronic programmable Device.

Este dispositivo, tiene espacio reservado para las aplicaciones de usuario, es decir, mediante esta EPLD podremos realizar la simulación de un periférico hardware sencillo, para probar nuestra tarjeta.

Para la aplicación del ejemplo, las direcciones offset que se utilizan son las siguientes:

Para acceso de datos de 32bits		
Offset : 0h	Bits de datos: 0 - 31	LBE: 0, 1, 2, 3
Para acceso de datos de 16 bits		
Offset : 0h Offset : 2h	Bits de datos: 0 - 15 Bits de datos: 16 - 31	LBE: 0, 1 LBE: 2, 3
Para acceso de datos de 8 bits		
Offset : 0h Offset : 1h Offset : 2h Offset : 3h	Bits de datos: 0 - 7 Bits de datos: 8 - 15 Bits de datos: 16 - 23 Bits de datos: 24 - 31	LBE: 0 LBE: 1 LBE: 2 LBE: 3

Tabla 4.5 Offset de direcciones para la aplicación de ejemplo.

El cronograma y diagrama de estados de la aplicación de ejemplo se muestra en la figura 4.3 y 4.4 respectivamente:

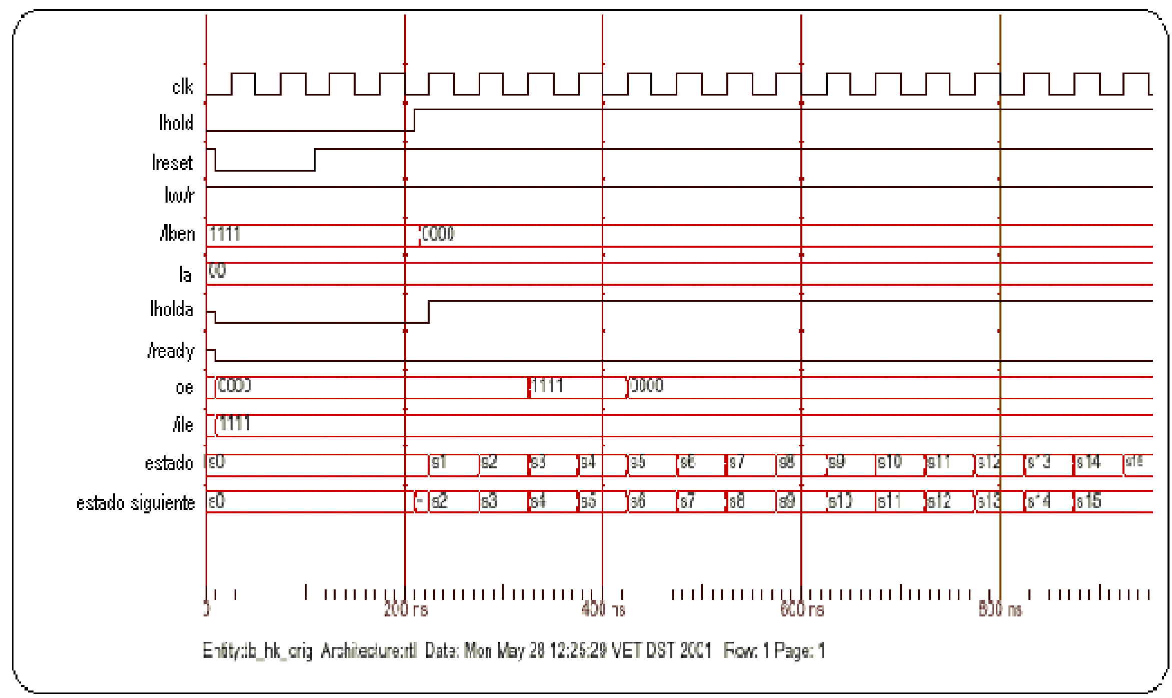


Figura 4.4: Cronograma de la aplicación de ejemplo.

El funcionamiento de la aplicación es el siguiente:

Tras realizar un reset de la tarjeta, el maestro pide el Bus con LHOLD, el cual no será cedido hasta que se active la señal LHOLDA. Como podemos observar, la señal LW/R indica que se trata de una

escritura. OE, output habilitado, indica el estado de los latches, que son activos cuanto estos están a '1', del mismo modo ILE# indica la habilitación de los latches de entrada que estarán activos cuando sean '0', en este caso como se trata de una escritura, permanecerán a '1', es decir desactivados. Las ultimas dos líneas indican el estado actual y el estado siguiente de los latches.

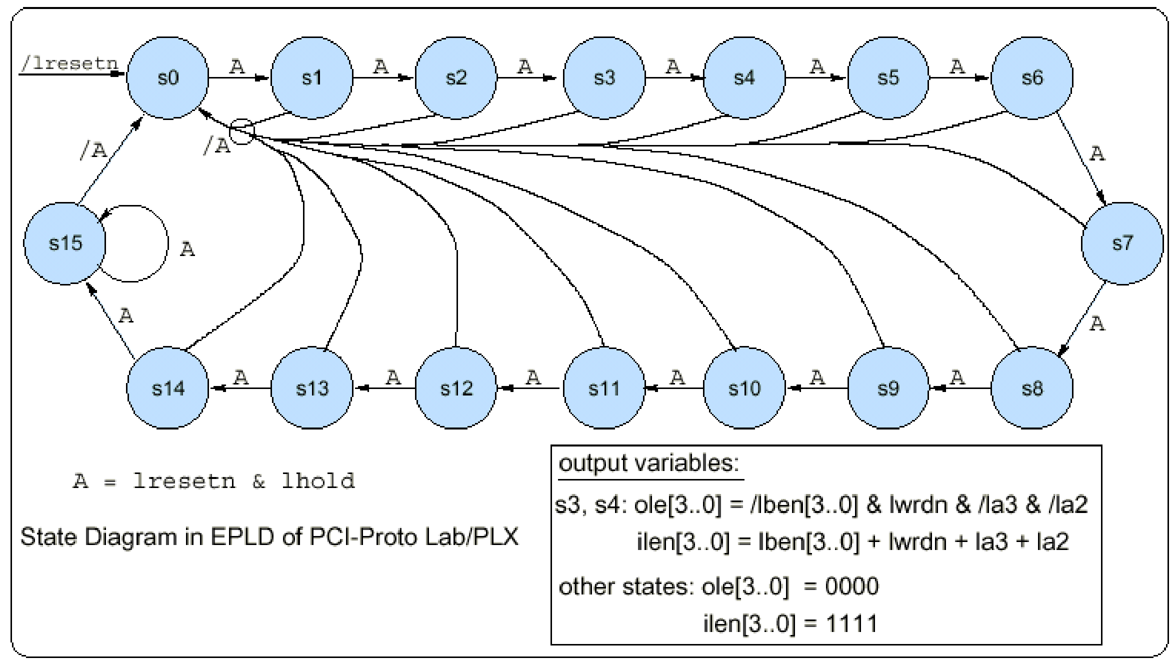


Figura 4.5 Diagrama de estados de la aplicación de ejemplo.

La EPLD instalada en la tarjeta de prototipo, M4A3-64/32 de la compañía Lattice semiconductor, trabaja con tensión de alimentación de +3.3 V y sus características se muestran en la tabla 4.6:

Parameter Symbol	Parameter Description	Test Conditions		Min	Typ	Max	Unit
V_{OH}	Output HIGH Voltage	V_{CC} = Min, $V_{IN} = V_{IH}$ or V_{IL}	I_{OH} = –100 µA	V_{CC} – 0.2			V
			I_{OH} = –3.2 mA	2.4			V
V_{OL}	Output LOW Voltage	V_{CC} = Min, $V_{IN} = V_{IH}$ or V_{IL} (Note 1)	I_{OL} = 100 µA			0.2	V
			I_{OL} = 24 mA			0.5	V
V_{IH}	Input HIGH Voltage	Guaranteed Input Logical HIGH Voltage for all Inputs		2.0		5.5	V
V_{IL}	Input LOW Voltage	Guaranteed Input Logical LOW Voltage for all Inputs		–0.3		0.8	V
I_{IH}	Input HIGH Leakage Current	V_{IN} = 3.6 V, V_{CC} = Max (Note 2)				5	µA
I_{IL}	Input LOW Leakage Current	V_{IN} = 0 V, V_{CC} = Max (Note 2)				–5	µA
I_{OZH}	Off-State Output Leakage Current HIGH	V_{OUT} = 3.6 V, V_{CC} = Max, $V_{IN} = V_{IH}$ or V_{IL} (Note 2)				5	µA
I_{OZL}	Off-State Output Leakage Current LOW	V_{OUT} = 0 V, V_{CC} = Max, $V_{IN} = V_{IH}$ or V_{IL} (Note 2)				–5	µA
I_{SC}	Output Short-Circuit Current	V_{OUT} = 0.5 V, V_{CC} = Max (Note 3)		–15		–160	mA

Tabla 4.6: Características de la EPLD M4A-64/32 de Lattice.

En la siguiente tabla se presentan las señales de la EPLD, las cuales se pueden utilizar para manejar los latches y el control del Bus de usuario.

Nº Pin.	Nombre	Descripción	función
1	GND		-
2	LHOLD	Señal de petición del Bus local.	Entrada
3	BTERM#	Burst terminado.	Entrada (inactiva)
4	no usado		I/O 2
5	BLAST#	Bus Last, última transferencia de ciclo de bus.	Salida (inactiva)
6	LW/R#	Escritura/ lectura.	Entrada
7	LA2	dirección Local 2.	Entrada
8	ADS#	Address Strobe; dirección válida en el bus.	Entrada (inactiva)
9	LA3	dirección Local 3.	Entrada
10	TDI	JTAG (Test de datos de entrada).	TDI
11	no usado	-	CLK0 I/O
12	GND	-	-
13	TCLK	JTAG (Reloj de test).	TCLK
14	IOE3#	Salida del latch de entrada habilitada 3.	Salida
15	IOE2#	Salida del latch de entrada habilitada 2.	Salida
16	IOE1#	Salida del latch de entrada habilitada 1.	Salida
17	IOE0#	Salida del latch de entrada habilitada 0.	Salida
18	OLE0	Escritura del latch de salida habilitada 0.	Salida
19	no usado	-	-
20	OLE1	Escritura del latch de salida habilitada 1.	Salida
21	BREQO	Petición de ciclo modo ráfaga.	Entrada (inactiva)
22	+3.3 V	-	-
23	GND	-	-
24	OLE2	Escritura del latch de salida habilitada 2.	Salida
25	BREQI	Petición de ciclo modo ráfaga.	Salida (inactiva)
26	no usado	-	-
27	U/DAC/L#	Conocimiento de modo DMA.	Entrada (inactiva)
28	OLE3	Escritura del latch de salida habilitada 3.	Salida
29	LBE3#	Enable 3 del Bus Local.	Entrada
30	U/DRE/L#	Petición del modo DMA.	Salida (inactiva)
31	LBE2#	Enable 2 del Bus Local.	Entrada
32	TDS	JTAG (Test de modo de salida).	-
33	LCLK	Reloj local.	-
34	GND	-	-
35	TDO	JTAG(Test de dato de salida).	-
36	LBE1#	Enable 1 del Bus Local.	Entrada
37	LBE0#	Enable 0 del Bus Local.	Entrada
38	WAIT#	Espera.	Salida (inactiva)
39	LRESET#	Reset local.	Entrada
40	no usado	-	-
41	no usado	-	-
42	READY#	Señal de dato válido en el bus.	Salida
43	LHOLDA	Cesión del Bus Local.	Salida
44	+3.3 V	-	-

Tabla 4.7: Pines del dispositivo EPLD.

4.2.5 OSCILADOR PROGRAMABLE.

El oscilador EPSON 2-0513B sirve para dotar el reloj del Bus Local de la tarjeta. Este puede ser programado desde 0 a 50 MHz. Debido a que nuestra tarjeta de aplicación XS40 también posee un oscilador programable, deshabilitaremos este reloj, extrayéndolo de la tarjeta, y su salida de reloj, mostrada en la figura 4.6 la conectaremos con el pin 13 de entrada de reloj de la FPGA. De esta forma, el reloj de Bus Local lo podemos modificar a través del programa GXSCLK explicado en el capítulo 6.

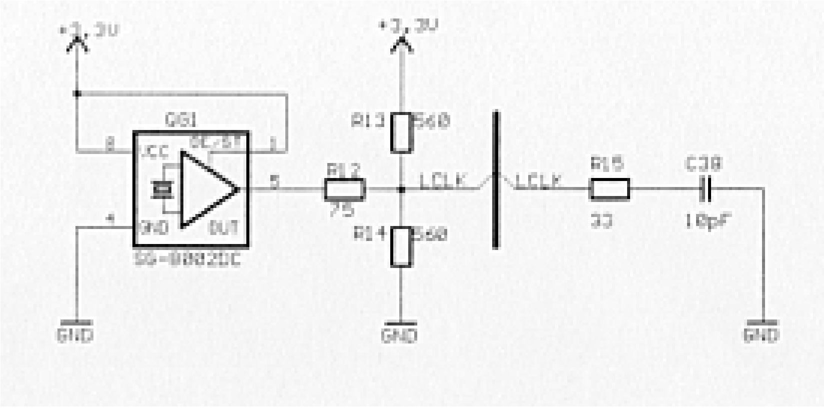


Figura 4.6 Esquema del oscilador programable de la tarjeta de prototipo.

Para ello simplemente deberemos de conectar el pin 13 de la tarjeta XS40, con el pin 5 del zócalo del oscilador, mostrado en la figura.

4.2.6 CONECTOR DE BUS PCI .

El conector de Bus PCI, compuesto por 116 líneas, por donde se transmiten las señales desde el PC a nuestra tarjeta de desarrollo.
Como se puede observar este tiene dos hendiduras realizadas, las cuales sirven para alimentar la tarjeta a +3.3V o +5V, simplemente deberemos de cambiarlos jumpers 5 y del 9 al 20, ver tabla 4.7

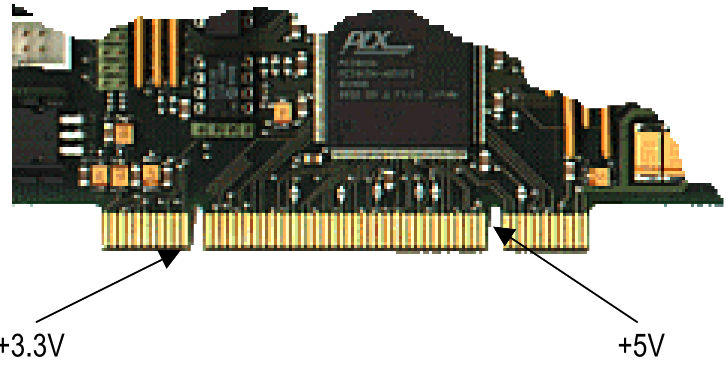
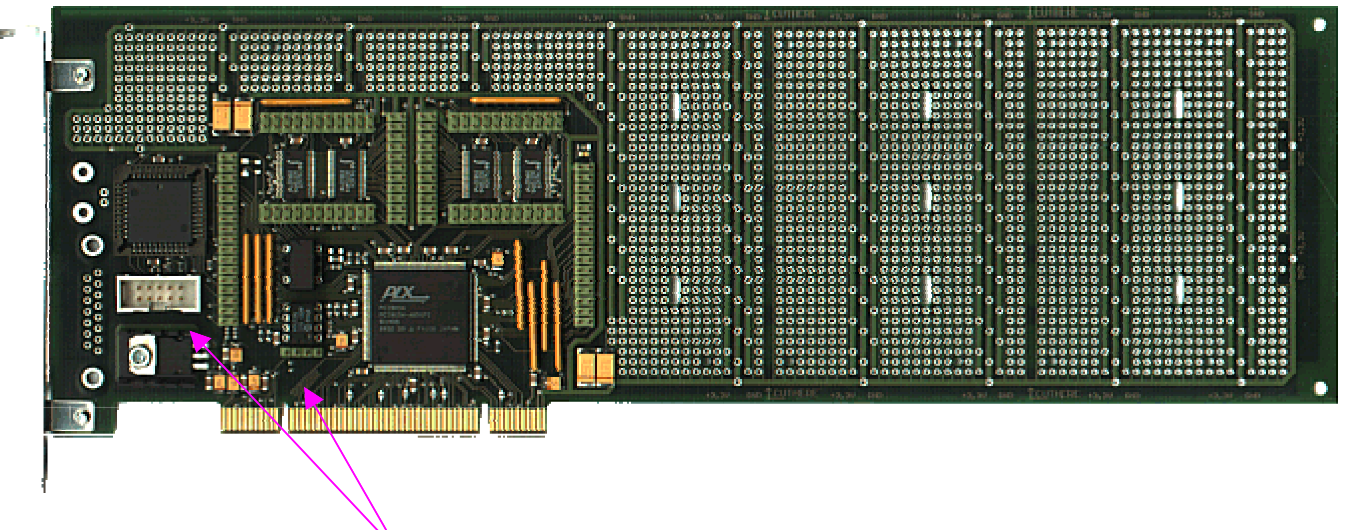


Figura 4.7 Conector de Bus PCI.

4.3 CONFIGURACIÓN DE LAS TARJETAS

La misión de la CPLD de la tarjeta es transmitir datos al Bus de Usuario a través de los latches instalados en la misma. por tanto, para nuestra aplicación final, la desinstalamos, consiguiendo que no interfieran los datos aleatorios que se encuentran en este bus de Usuario.

Del mismo modo, sustraeremos el oscilador programable colocado en la tarjeta como muestra la figura 6.2 para configurar nuestro reloj de Bus Local, con el reloj programable instalado en la tarjeta XS40, conectando ambas salidas de reloj, como se detalla en un apartado posterior.



Componentes que debemos desinstalar
para realizar nuestra aplicación

Figura 4.8: Sustracción del reloj y de la CPLD instalada en la tarjeta prototipo PCI

Por otro lado, en la tarjeta XS40 deberemos dejar los jumpers instalados por defecto, salvo cuando queramos configurar la frecuencia de reloj de ésta, donde tendremos que seguir los pasos indicados por el programa de configuración de reloj, cambiando el jumper J12.
En el caso de querer dejar la tarjeta configurada, y no querer estar configurándola cada vez que queramos usarla, tendremos que instalar una EEPROM serie en el zócalo previsto para ello y cambiar los jumpers:

- J5 (On), J10 (On) y J11(Off) Mientras se configura la EEPROM
- J4 (Off) Cuando queramos hacer uso de la configuración cargada en la EEPROM, una vez queramos dejar nuestra tarjeta configurada para una aplicación concreta
- J12 (2-3) set cuando queramos cambiar la configuración de reloj.

4.4 SOFTWARE.

PLX Host-SDK es un paquete software que acompaña al kit de desarrollo, con el fin de entender y controlar el funcionamiento de los procesos en la tarjeta de prototipo PCI-Proto LAB/PLX.
Host-SDK esta compuesto por una biblioteca API, drivers para los sistemas operativos Windows2000/NT/98, programa 'PLXMon 200x', para controlar la tarjeta y manuales en pdf.

El programa PLXMon 200x' permite leer y escribir datos de tamaño byte, Dword y Lword en la I/O o en la memoria. También podemos acceder con este programa a la aplicación de ejemplo implementada en la CPLD. Para ello, existen dos formas de acceder a los latches de datos, mediante:

- Comandos de la entrada-salida, usando la región 2 de dirección PCI, correspondiente con el espacio de direccionamiento local 0.

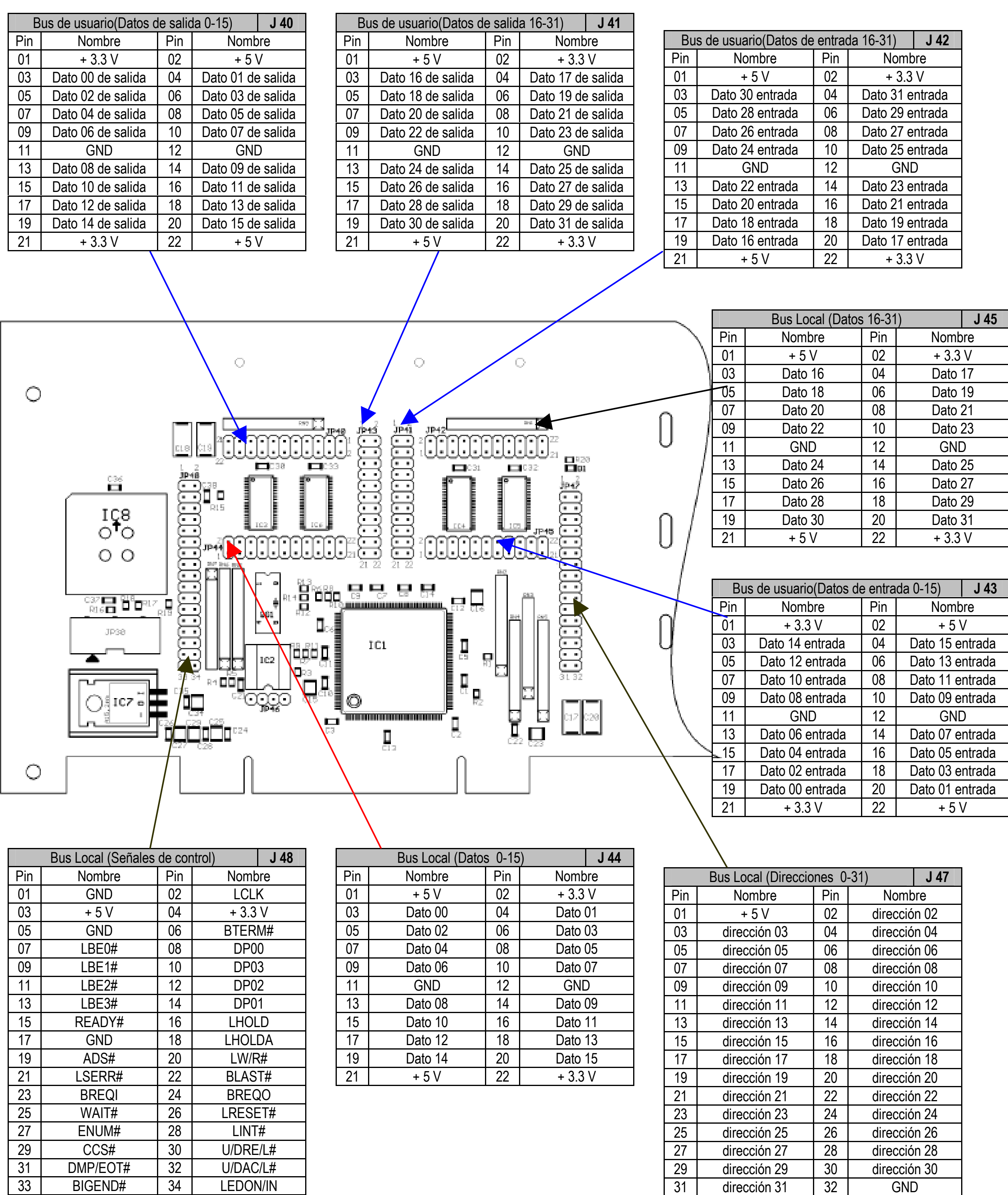
- Comandos de memoria, usando la región 3 de la dirección PCI, correspondiente con el espacio de direccionamiento local 1.

Con el programa 'PLXMon 200x' también podemos leer y escribir en la configuración en la EEPROM serie, por lo que existe la posibilidad que el usuario configure una aplicación especifica y la pruebe con este software.

Mediante los drivers y las bibliotecas API, podemos desarrollar un software con el que controlar las operaciones que se realicen en la tarjeta, a través del Bus PCI. Se ha diseñado una API simplificada de la que se distribuye con el pack de desarrollo. Esta API es más sencilla de utilizar que la original y necesita de menos conociemientos. En el capitulo Programación Software se explica detalladamente esta API, las funciones que la componen y el funcionamiento de las mismas.

Existen completos documentos en formato pdf donde podemos encontrar toda la documentación necesaria, para hacer uso de estos paquetes de software.

4.5 PINES DE LA TARJETA PCI PROTO- LAB/PLX



Bus de usuario(Datos de salida 0-15)			J 40
Pin	Nombre	Pin	Nombre
01	+ 3.3 V	02	+ 5 V
03	Dato 00 de salida	04	Dato 01 de salida
05	Dato 02 de salida	06	Dato 03 de salida
07	Dato 04 de salida	08	Dato 05 de salida
09	Dato 06 de salida	10	Dato 07 de salida
11	GND	12	GND
13	Dato 08 de salida	14	Dato 09 de salida
15	Dato 10 de salida	16	Dato 11 de salida
17	Dato 12 de salida	18	Dato 13 de salida
19	Dato 14 de salida	20	Dato 15 de salida
21	+ 3.3 V	22	+ 5 V

Bus de usuario(Datos de salida 16-31)			J 41
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 16 de salida	04	Dato 17 de salida
05	Dato 18 de salida	06	Dato 19 de salida
07	Dato 20 de salida	08	Dato 21 de salida
09	Dato 22 de salida	10	Dato 23 de salida
11	GND	12	GND
13	Dato 24 de salida	14	Dato 25 de salida
15	Dato 26 de salida	16	Dato 27 de salida
17	Dato 28 de salida	18	Dato 29 de salida
19	Dato 30 de salida	20	Dato 31 de salida
21	+ 5 V	22	+ 3.3 V

Bus de usuario(Datos de entrada 16-31)			J 42
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 30 entrada	04	Dato 31 entrada
05	Dato 28 entrada	06	Dato 29 entrada
07	Dato 26 entrada	08	Dato 27 entrada
09	Dato 24 entrada	10	Dato 25 entrada
11	GND	12	GND
13	Dato 22 entrada	14	Dato 23 entrada
15	Dato 20 entrada	16	Dato 21 entrada
17	Dato 18 entrada	18	Dato 19 entrada
19	Dato 16 entrada	20	Dato 17 entrada
21	+ 5 V	22	+ 3.3 V

Bus Local (Datos 16-31)			J 45
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 16	04	Dato 17
05	Dato 18	06	Dato 19
07	Dato 20	08	Dato 21
09	Dato 22	10	Dato 23
11	GND	12	GND
13	Dato 24	14	Dato 25
15	Dato 26	16	Dato 27
17	Dato 28	18	Dato 29
19	Dato 30	20	Dato 31
21	+ 5 V	22	+ 3.3 V

Bus de usuario(Datos de entrada 0-15)			J 43
Pin	Nombre	Pin	Nombre
01	+ 3.3 V	02	+ 5 V
03	Dato 14 entrada	04	Dato 15 entrada
05	Dato 12 entrada	06	Dato 13 entrada
07	Dato 10 entrada	08	Dato 11 entrada
09	Dato 08 entrada	10	Dato 09 entrada
11	GND	12	GND
13	Dato 06 entrada	14	Dato 07 entrada
15	Dato 04 entrada	16	Dato 05 entrada
17	Dato 02 entrada	18	Dato 03 entrada
19	Dato 00 entrada	20	Dato 01 entrada
21	+ 3.3 V	22	+ 5 V

Bus Local (Señales de control)			J 48
Pin	Nombre	Pin	Nombre
01	GND	02	LCLK
03	+ 5 V	04	+ 3.3 V
05	GND	06	BTERM#
07	LBE0#	08	DP00
09	LBE1#	10	DP03
11	LBE2#	12	DP02
13	LBE3#	14	DP01
15	READY#	16	LHOLD
17	GND	18	LHOLDA
19	ADS#	20	LW/R#
21	LSERR#	22	BLAST#
23	BREQI	24	BREQO
25	WAIT#	26	LRESET#
27	ENUM#	28	LINT#
29	CCS#	30	U/DRE/L#
31	DMP/EOT#	32	U/DAC/L#
33	BIGEND#	34	LEDON/IN

Bus Local (Datos 0-15)			J 44
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 00	04	Dato 01
05	Dato 02	06	Dato 03
07	Dato 04	08	Dato 05
09	Dato 06	10	Dato 07
11	GND	12	GND
13	Dato 08	14	Dato 09
15	Dato 10	16	Dato 11
17	Dato 12	18	Dato 13
19	Dato 14	20	Dato 15
21	+ 5 V	22	+ 3.3 V

Bus Local (Direcciones 0-31)			J 47
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	dirección 02
03	dirección 03	04	dirección 04
05	dirección 05	06	dirección 06
07	dirección 07	08	dirección 08
09	dirección 09	10	dirección 10
11	dirección 11	12	dirección 12
13	dirección 13	14	dirección 14
15	dirección 15	16	dirección 16
17	dirección 17	18	dirección 18
19	dirección 19	20	dirección 20
21	dirección 21	22	dirección 22
23	dirección 23	24	dirección 24
25	dirección 25	26	dirección 26
27	dirección 27	28	dirección 28
29	dirección 29	30	dirección 30
31	dirección 31	32	GND

Figura 4.9 Pines de la tarjeta de prototipo.

4.6 DIAGRAMA DE CONEXIONES Y JUMPERS.

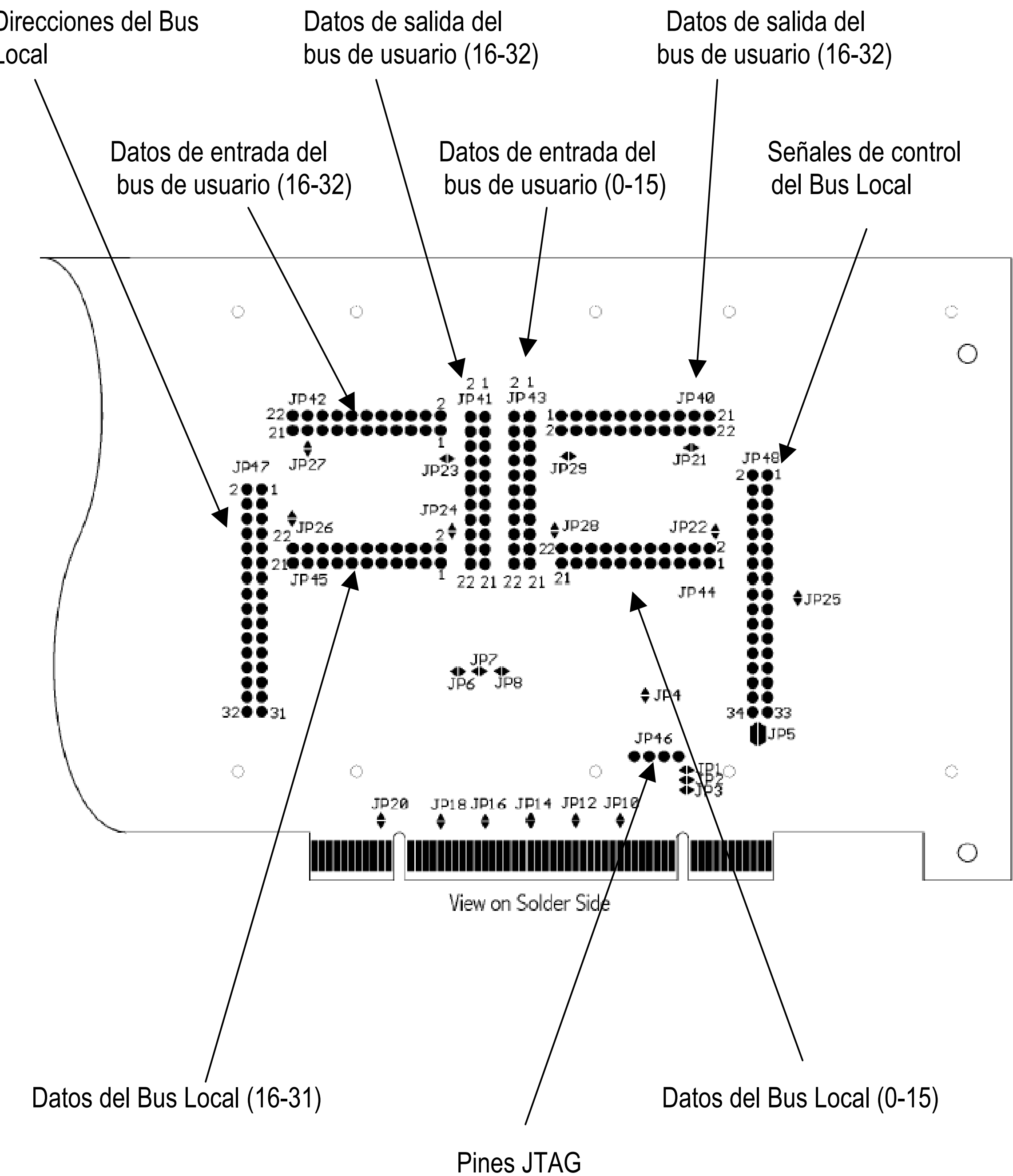


Figura 410: Diagrama de conexiones y jumpers.

En la tabla se muestra el conjunto de jumpers existente en la tarjeta de prototipo, con los cuáles podemos cambiar algunas funciones de la tarjeta de prototipo del Bus PCI PCI Proto-Lab/PLX.

Jumper	default setting	Function	
		open	closed
1	closed	motherboard JTAG chain broken	Motherboard JTAG chain closed
2	closed	Card Power Requirement Indication PRSNT1# is open (high)	Card Power Requirement Indication PRSNT1# is low
3	open	Card Power Requirement Indication PRSNT2# is open (high)	Card Power Requirement Indication PRSNT2# is low
4	open	blank or programmed serial EEPROM present on board [1]	no serial EEPROM present on board [1]
5	closed	+3.3V Main Supply active [2]	+5V Main Supply active [2]
6	open	TEST input pin (PCI9054) is low	TEST input pin (PCI9054) is high
7	open	MODE0 input pin (PCI9054) is low	MODE0 input pin (PCI9054) is high
8	open	MODE1 input pin (PCI9054) is low	MODE1 input pin (PCI9054) is high
9	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
10	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
11	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
12	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
13	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
14	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
15	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
16	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
17	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
18	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
19	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
20	open	+5V Main Supply active [2]	+3.3V Main Supply active [2]
21	closed	Output latch 'Data 0- 7' at tristate	Output latch 'Data 0- 7' active [3]
22	closed	Output latch 'Data 8- 15' at tristate	Output latch 'Data 8- 15' active [3]
23	closed	Output latch 'Data 16- 23' at tristate	Output latch 'Data 16- 23' active [3]
24	closed	Output latch 'Data 24- 31' at tristate	Output latch 'Data 24- 31' active [3]
25	open	/LHOLD and /LHOLDA are disconnected	/LHOLD and /LHOLDA are connected
26	closed	Input latch 'Data 0 - 7' locked	Input latch 'Data 0 - 7' transparent [3]
27	closed	Input latch 'Data 8 - 15' locked	Input latch 'Data 8 - 15' transparent [3]
28	closed	Input latch 'Data 16- 23' locked	Input latch 'Data 16- 23'transparent [3]
29	closed	Input latch 'Data 24- 31' locked	Input latch 'Data 24- 31'transparent [3]

Tabla 4.9: Jumpers de la tarjeta de prototipo.

4.7 ESQUEMAS.

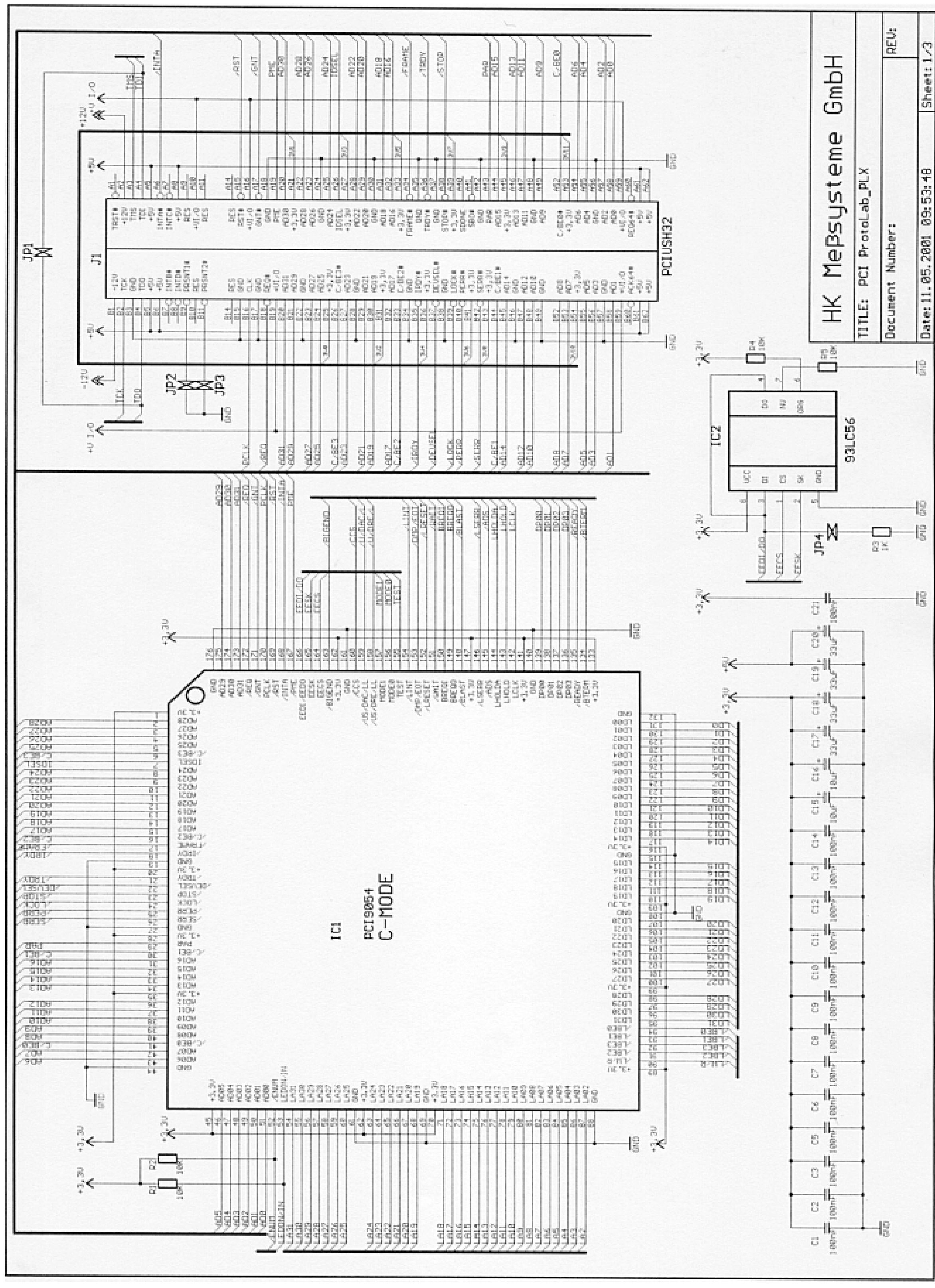


Figura 4.10 Esquema de la tarjeta PCI Proto/Lab PLX(1)

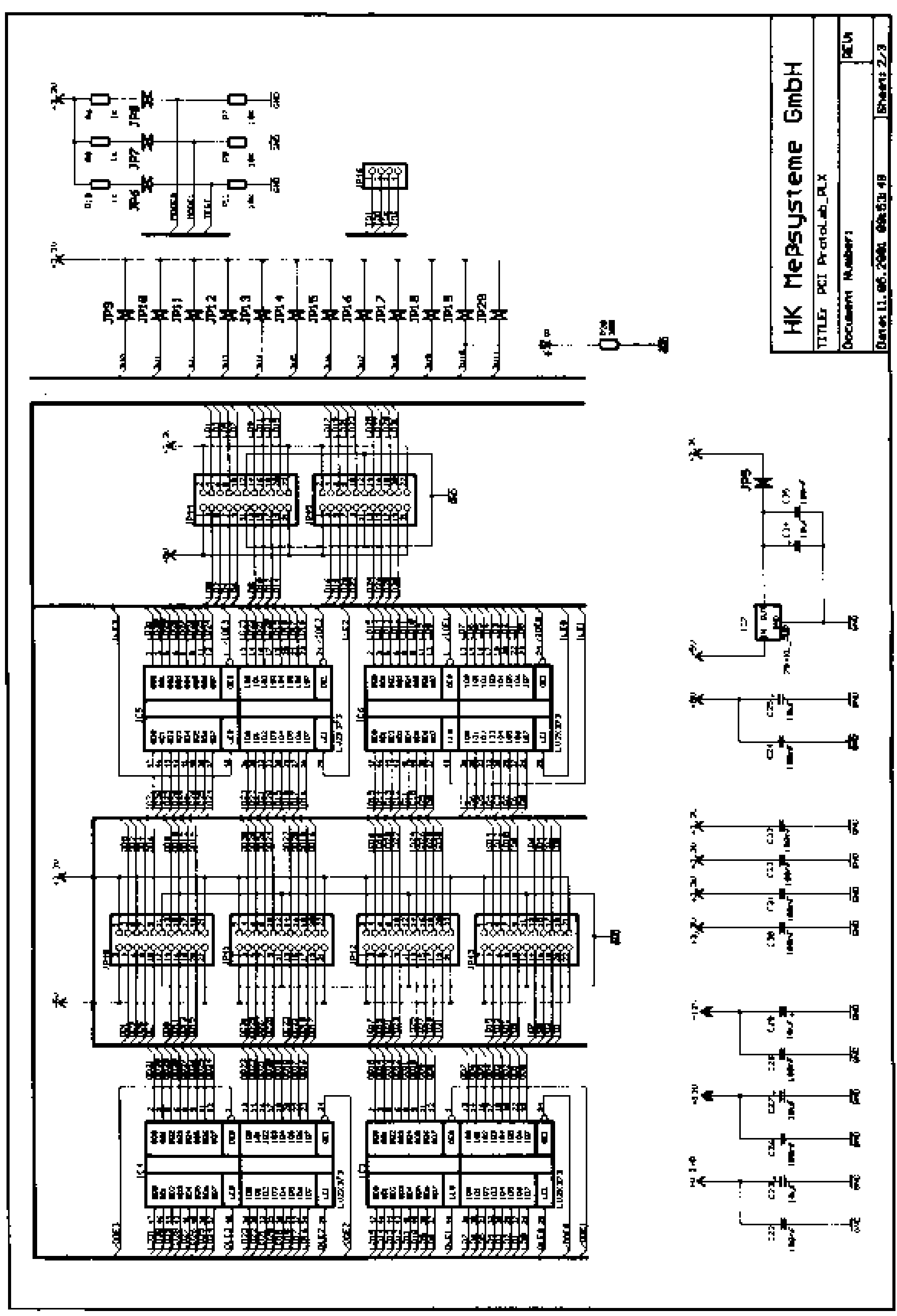


Figura 4.11 Esquema de la tarjeta PCI Proto-Lab/PLX(2)

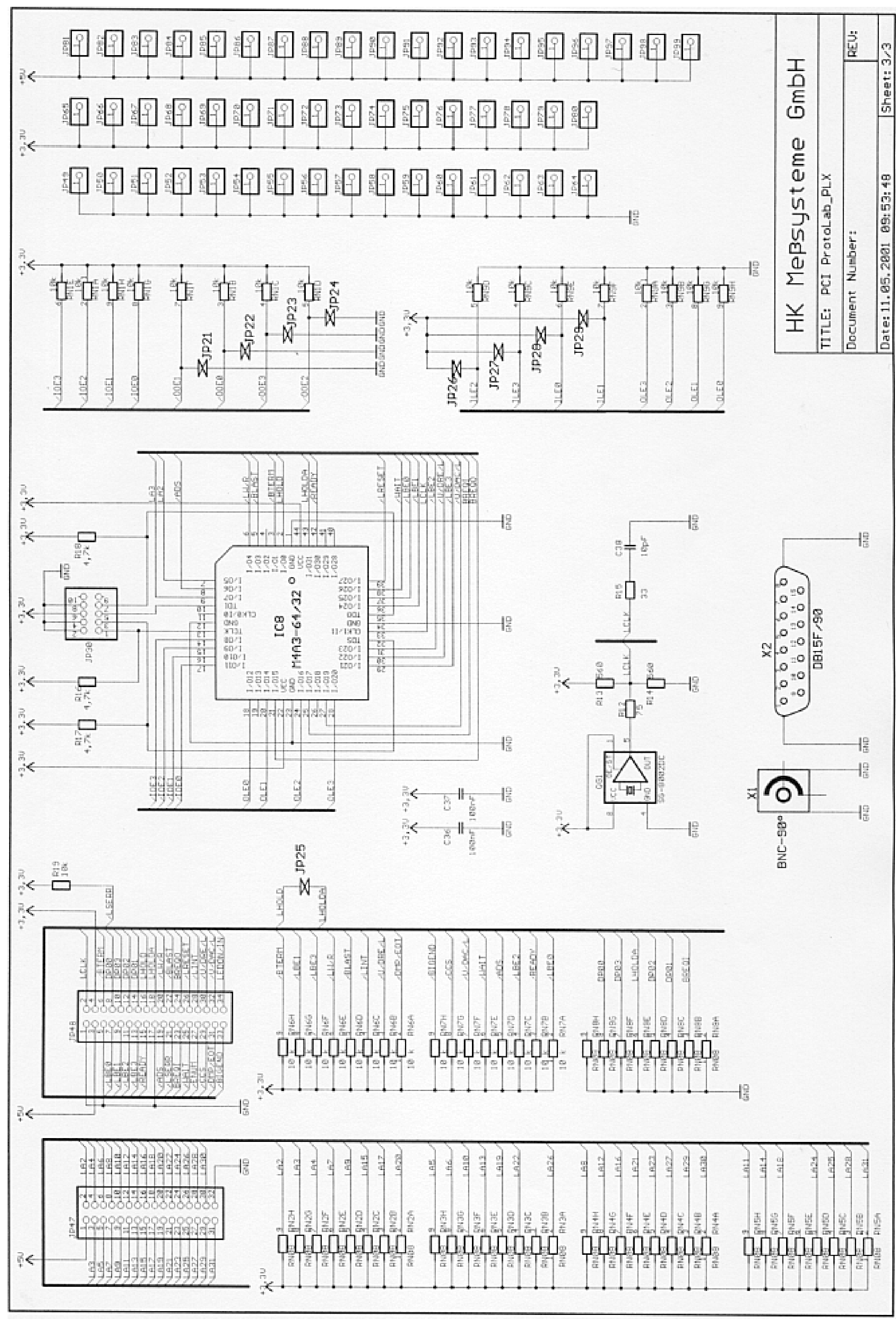


Figura 4.12 Esquema de la tarjeta PCI Proto-Lab/PLX(3).

CAPÍTULO 5

El controlador PLX. Arquitectura de control.

Como se ha dicho en el capítulo anterior, cualquier tarjeta que se conecte al Bus PCI necesita un elemento para controlar las transferencias de datos entre dicha tarjeta y el PC u otros dispositivos PCI. Para ello, la tarjeta de prototipos PCI Proto Lab/ PLX incorpora un controlador PCI 9054 del fabricante PLX. Este controlador enlaza el Bus PCI (donde estará conectada la tarjeta) con el Bus Local, presente en la propia tarjeta y donde se conecta el hardware que pongamos en la tarjeta de prototipos. Un esquema sencillo de cómo sería esta arquitectura se muestra en la figura 5.1, donde se puede observar como el controlador PLX hace de enlace entre los dos buses, con el PC en un lado y el hardware de prototipo en el otro lado.

En este capítulo se describirán las principales características y los modos de funcionamiento de nuestro controlador de Bus PCI, centrándonos finalmente en el modo de operación y transferencia con el que realizaremos nuestra aplicación.

El PCI 9054 cumple las especificaciones de la versión 2.2 del Bus PCI. Éste, posee un acelerador de I/O mediante el bus PCI, de 32 bits a 33 Mhz. Es el dispositivomásavanzado para propósito general de Bus Maestro, permitiendo transferencias en modo de ráfaga de hasta 132 MB/ segundo.

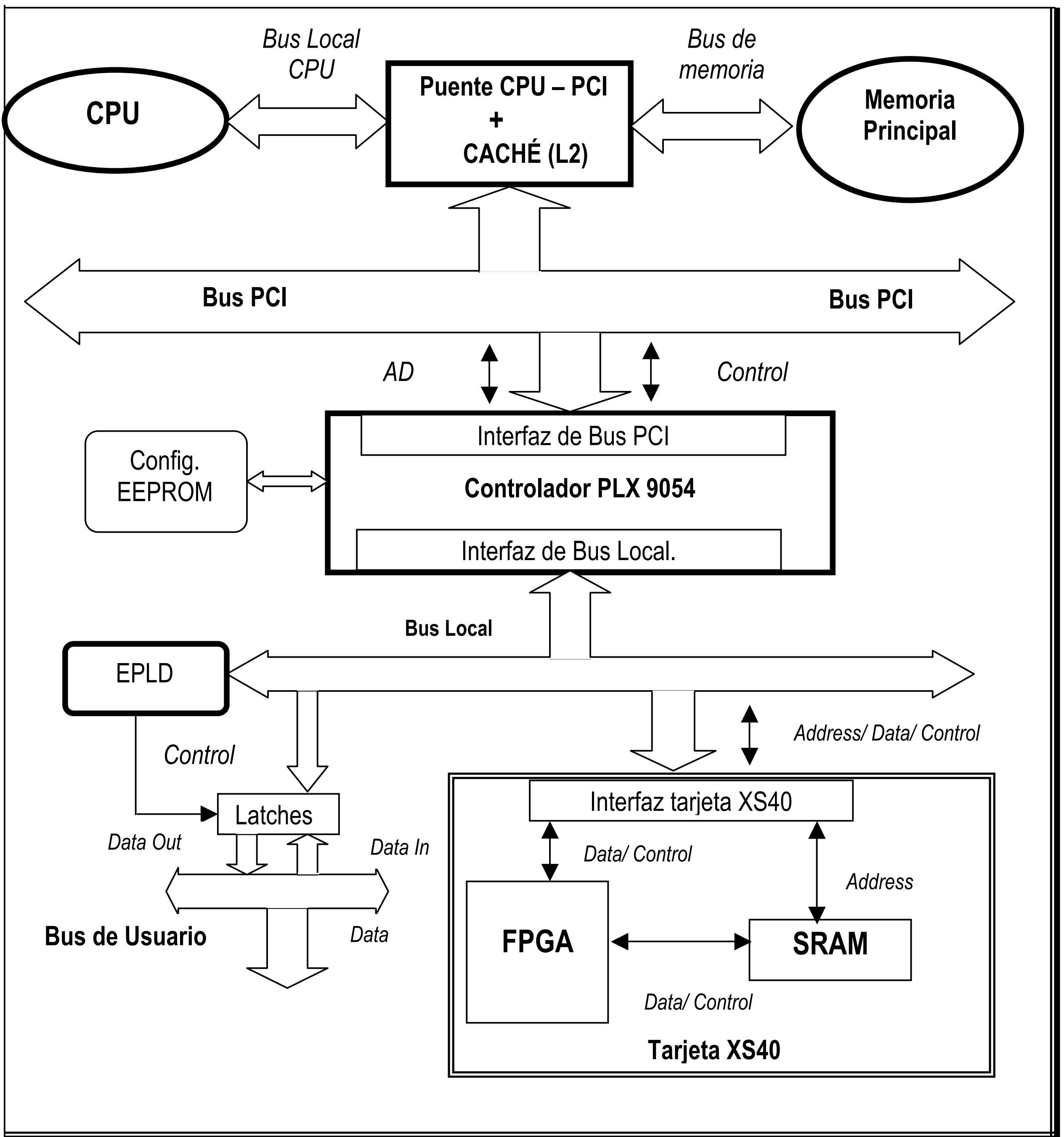


Figura 5.1 Arquitectura de la aplicación.

El PCI 9054 posee la principal tecnología de transferencia de datos, incluyendo los modos de transferencia DMA, PCI Initiator, donde actuará como maestro, y PCI Target Data-Transfer, donde actuará como esclavo, además de funciones mensajes PCI.

En el presente capítulo se referirá en minúsculas a los registros internos PCI y con mayúsculas a las señales, por ejemplo, Bterm es el registro interno del PCI 9054, y BTERM# la señal externa de transferencia burst terminada.

Como podemos observar en el diagrama de bloques del controlador 9054, este se comunica por un lado con el Bus PCI y por el otro con el Bus Local.

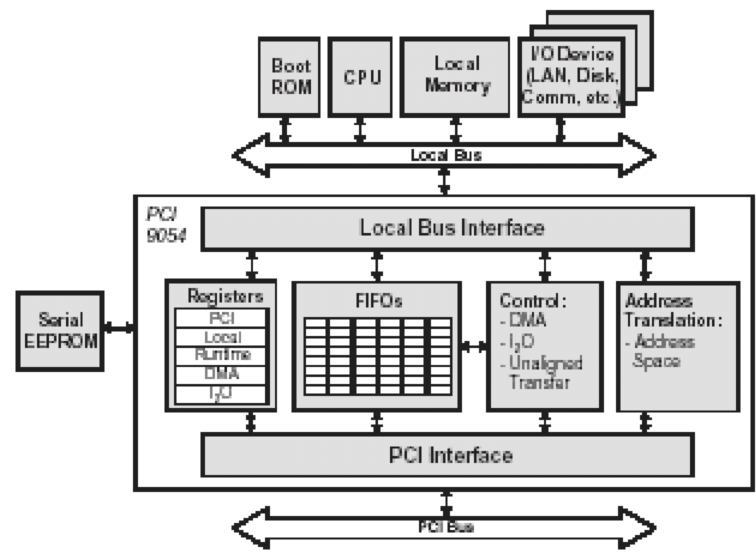


Figura 5.2 Diagrama de bloques del controlador PCI 9054

En el lado de la interfaz de Bus PCI, puede realizar transferencias PCI Initiator, para transferencias directas, donde el maestro es el sistema conectado al Bus Local[1], PCI Initiator para transferencias en modo DMA por el canal 0 o 1 y PCI Target para transferencias donde el sistema conectado al Bus Local actúe como esclavo, siendo el PC, el que actúe como maestro.

Por otro lado, en la parte del Bus Local, se puede realizar conversión Little Endian, manejar el ancho del bus a 8, 16 o 32 bits, multiplexar o no los datos y las direcciones, realizar transferencias de esclavo local, de maestro local por los canales 0 y 1 DMA y transferencias directas.

Entre ambos interfaces, se encuentra una FIFO de 32 Lword de escritura y 16 Lword de lectura, para estos tipos de transferencias.

Los registros del controlador serán cargados mediante la EEPROM serie conectada con el mismo.

[1] Cuando hablemos de sistema conectado al Bus Local, este puede ser desde un uC hasta el caso más simple de un latch de datos.

5.1 MODOS PROGRAMABLES DEL BUS DE OPERACIÓN

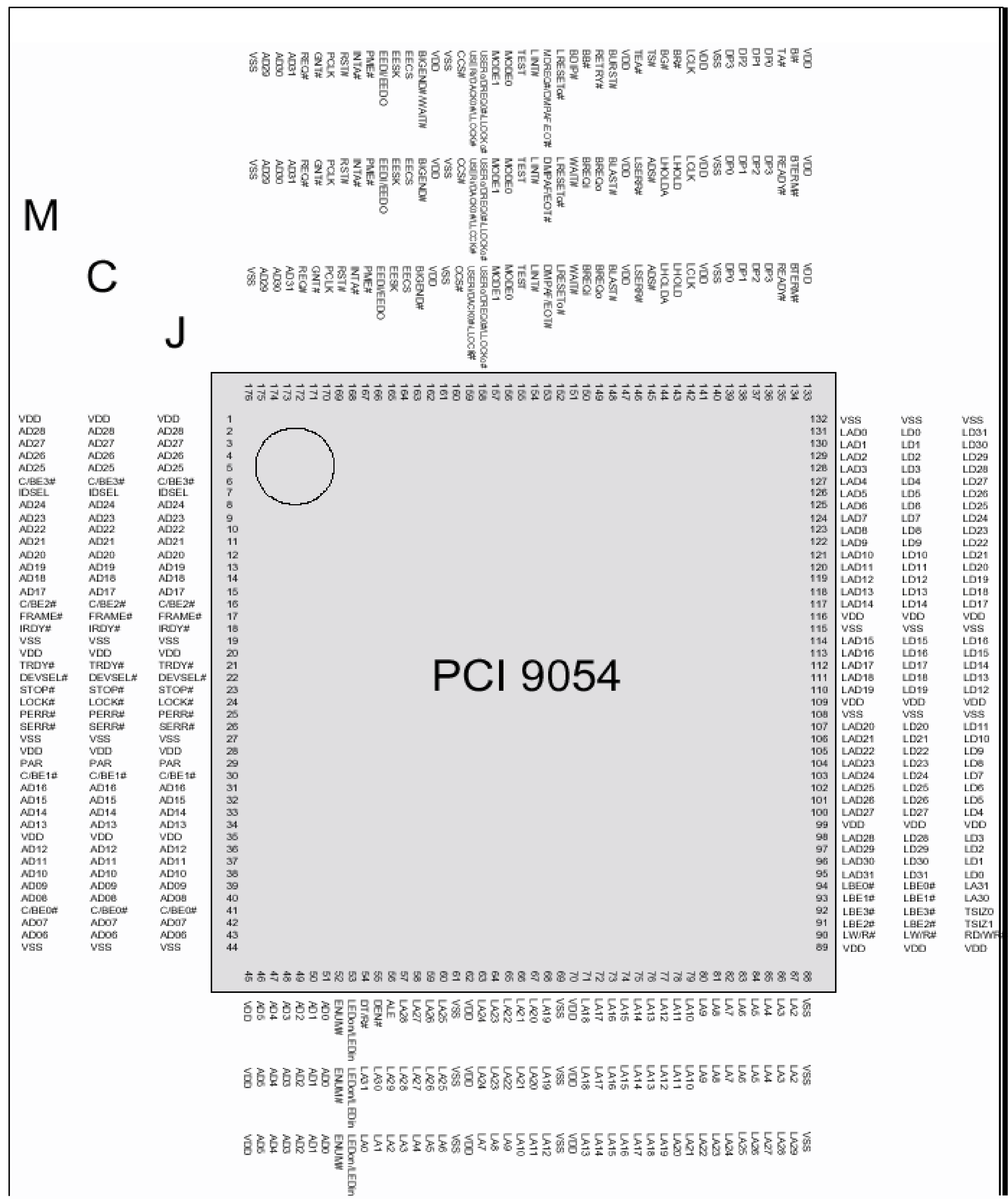


Figura 5.3 Pines del controlador PCI 9054 en sus diferentes modos

El PCI 9054, es un controlador de Bus PCI que lo conecta con un Bus Local de tres modos diferentes, modo M, modo C o modo J, el cual será seleccionado mediante los pines de modo, p157 y p156.

Por lo que con un diseño similar el PCI 9054 puede ser conectado a cualquier Bus Local, mediante la lógica de estos pines, tal y como se muestra en la Tabla 5.1.

Pin 157	Pin 156	Modo	Descripción
1	1	M	32 bits no multiplexados
1	0	Reservado	-
0	1	J	32 bits multiplexados
0	0	C	32 bits no multiplexados

Tabla 5.1: Modos del Bus Local.

En la tarjeta PCI Proto Lab/PLX, el controlador viene configurado en modo C, por lo que no necesitaremos modificar estos pines.

5.1.1 OPERACIÓN EN MODO M.

En este modo el PCI 9054 esta diseñado para comunicarse con el sistema conectado al Bus Local, pudiendo ser por ejemplo un microprocesador (Motorola MPC 850 o MPC 860), de 5 modos diferentes:

- **Acceso a los registros de configuración.**
- **Operación PCI Initiator.** Donde el sistema conectado al Bus Local actúa como maestro, y el que está conectado al Bus PCI como esclavo.
- **Operación PCI Target.** Donde el sistema conectado al Bus PCI, el PC, actúa como maestro, y el sistema conectado al Bus Local como esclavo.
- **Operación DMA.** Se realiza un acceso directo desde el controlador a la memoria conectada en el Bus Local. ❷ en la figura 5.4, del mismo modo se puede realizar un acceso DMA a la memoria principal del PC
- **Operación IDMA/ SDMA.** Se realiza un acceso directo desde el controlador 9054 al procesador conectado en el Bus Local. ❶ en la figura 5.4

En la operación DMA el PCI 9054 puede operar de forma bidireccional con el PCI 9054 como maestro para ambos buses, para dirigir transferencias de datos desde el Bus Local al Bus PCI, y desde el Bus PCI al Bus Local. El PCI 9054 tiene ilimitada capacidad de ráfaga para cualquier diseño con cualquier µprocesador MPC860.

Al mismo tiempo, para operación IDMA, el PCI 9054 transfiere datos al PCI bajo el control del protocolo IDMA.

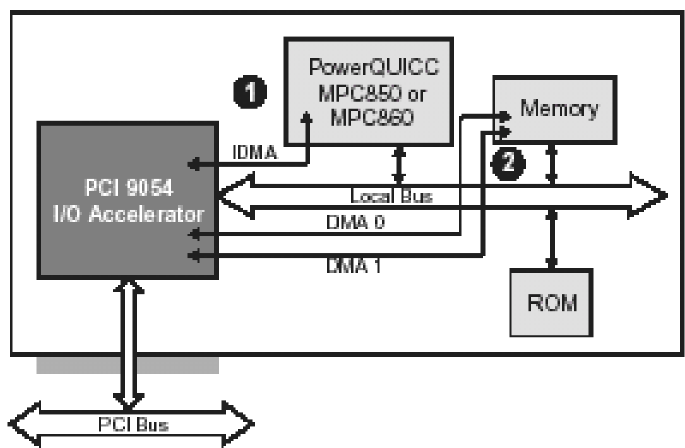


Figura 5.4 Transferencia DMA e IDMA.

5.1.2 OPERACIÓN EN MODO C Y J.

En este modo el PCI 9054 esta diseñado, para conectarse con el microprocesador Intel 960 o el IBM PPC 401. Del mismo modo podemos conectar otro muchos dispositivos. Este es el modo con el que opera el controlador en nuestra tarjeta de prototipo.

El controlador se comunica con el procesador de 4 modos diferentes:

- **Acceso a los registros de configuración.**
- **Operación PCI Initiator.** Donde el sistema conectado al Bus Local actúa como maestro, y el que está conectado al Bus PCI como esclavo.
- **Operación PCI Target.** Donde el sistema conectado al Bus PCI, el PC, actúa como maestro, y el sistema conectado al Bus Local como esclavo.
- **Operación DMA.** Se realiza un acceso directo desde el controlador a la memoria conectada en el Bus Local. como muestra la figura 5.4

En la tarjeta Proto-Lab/PLX 9054 RDK860, el controlador PCI 9054, esta configurado en modo C, por lo que a partir de este punto nos centraremos en este modo de operación.

5.2 MODO DE TRANSFERENCIA DIRECTA DE DATOS.

El PCI 9054 posee 3 modos de transferencia directa de datos, dependiendo de que sistema conectado al Bus Local o Bus PCI, actúe como maestro:

- PCI Initiator.
- PCI Target.
- DMA.

5.2.1 PCI INITIATOR (MAESTRO DIRECTO).

En el modo PCI Initiator, la CPU local (dispositivo conectado al bus local) accede a la memoria del PCI o al puerto I/O de intercambio de datos. En este modo el Bus PCI actúa como esclavo, siendo la CPU local[1] que actúe como maestro.

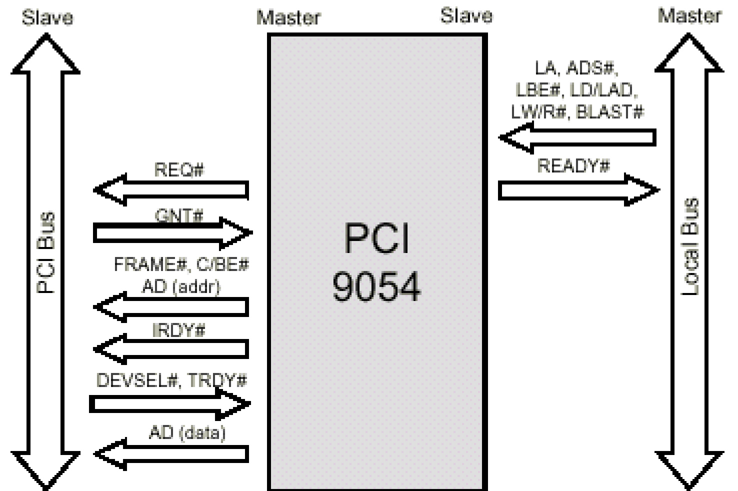


Figura 5.5: Ciclo de escritura PCI Initiator.

Los ciclos del Bus Local pueden ser ciclos singulares o de tipo burst[2], los cuales se seleccionan mediante la señal BLAST#. Si BLAST# se activa, en el comienzo de la primera fase del ciclo de datos, se realizará un ciclo de bus singular.

Por otro lado, cuando el PCI 9054 realiza un ciclo burst, BLAST# se activa al final de dicho ciclo.

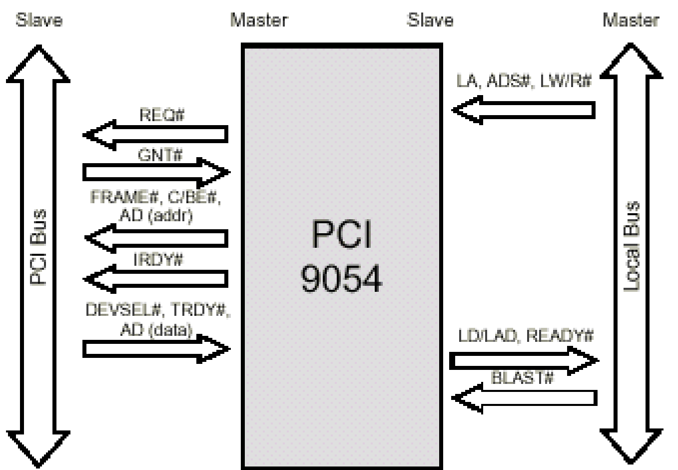


Figura 5.6: Ciclo de lectura PCI Initiator.

Las características más importantes de este modo de transferencia son las siguientes:

- Ciclos de configuración de Tipo 0 y Tipo 1[3].
- Soporta todo tipo de ciclos acceso a I/O y a Memoria PCI.
- Pretransacción de lectura.
- Puntero de longitud de la ráfaga programable.
- Control de transferencia no alineada.
- Conversion Big/Little Endian.

---

1 Dispositivo conectado al Bus Local.
2 Ráfaga.
3 Tipo 0 definido para el puente PCI- PCI; Tipo 1 definido para el resto de los controladores (nuestro caso).

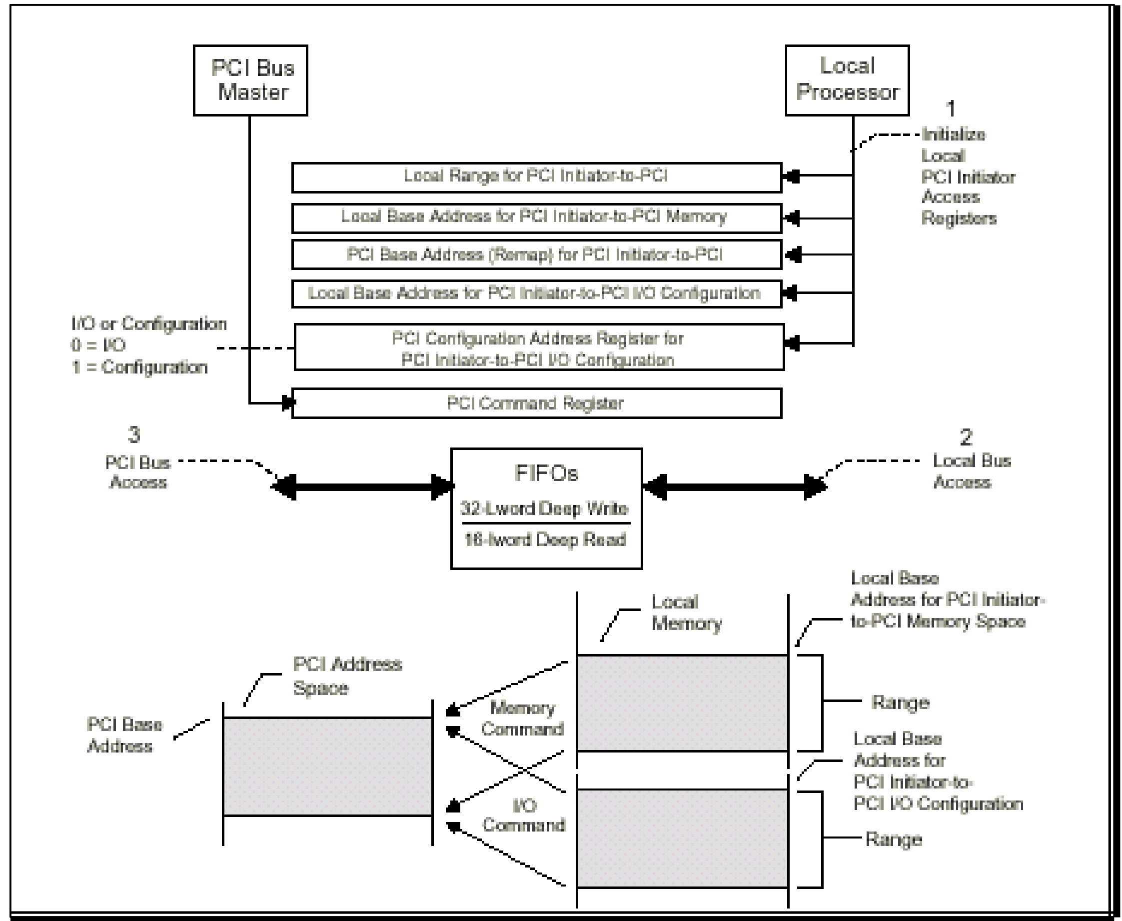


Figura 5.7: Acceso PCI Initiator al Bus PCI.

Un ciclo del modo PCI Initiator, sigue las siguientes etapas:

1. El procesador Local inicializa los accesos a los registros PCI Initiator, dejando que el bus PCI acceda a los registros de comando PCI.
2. El procesador Local accede al Bus Local, seleccionando la dirección base local y el rango de memoria.
3. Mediante la cuál y a través de la FIFO interna poder direccionar el espacio de dirección PCI base, para acceder al Bus PCI.

5.2.2 PCI TARGET (ESCLAVO DIRECTO).

En éste, el bus PCI como maestro, accede a la memoria local o al puerto I/O. En el modo PCI Target, el bus PCI actúa como maestro y la CPU local (1) actúa como esclavo.

El Bus PCI, como maestro lee desde el Bus Local y escribe en él.

Las características más importantes de este modo de transferencia son las siguientes:

- Múltiples espacios de dirección independientes.
- Control dinámico del ancho del Bus Local.
- Pre-transacción de lectura.
- Conversión Big/Little Endian.
- Control de prioridad del Bus Local.
- Temporizador de latencia PCI.

Figura 5.8: Ciclo de escritura PCI Target.

Por las características de este modo de funcionamiento, nuestra aplicación la realizaremos en este modo de funcionamiento, que explicaremos con detalle en un punto posterior.

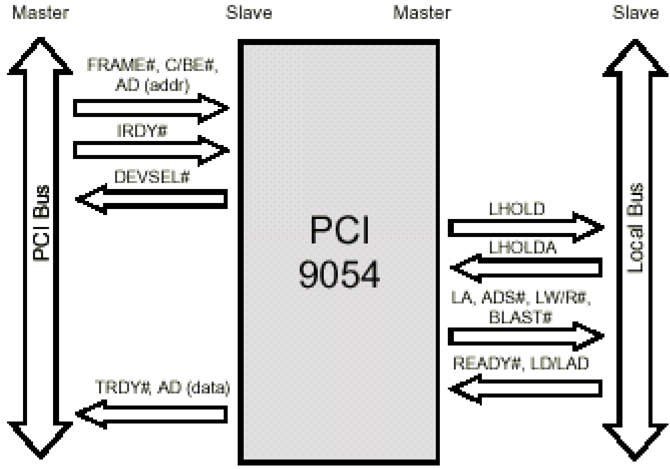


Figura 5.9: Ciclo de lectura PCI Target.

[1] Dispositivo conectado al Bus Local.

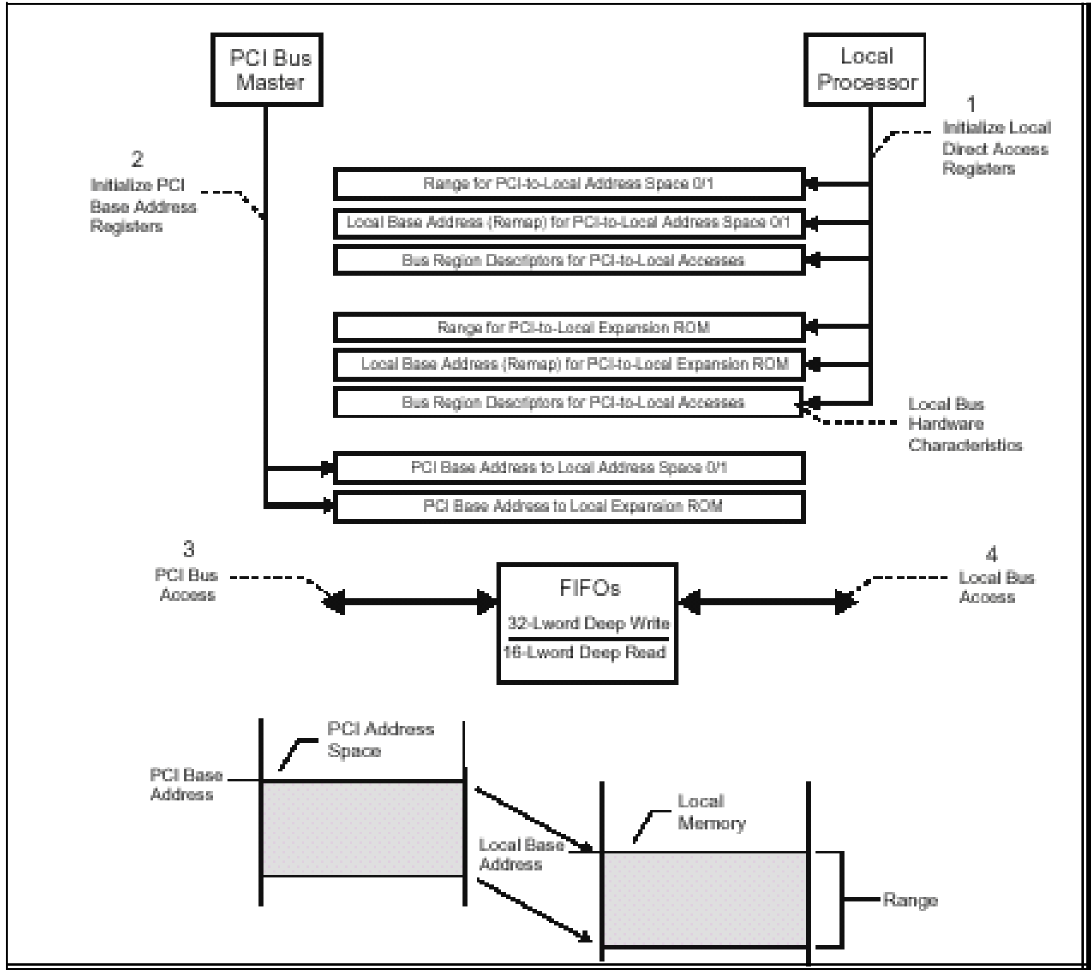


Figura 5.10: Acceso PCI Target al Bus Local.

Un ciclo del modo PCI Target, sigue las siguientes etapas:

1. El procesador Local inicializa los accesos a los registros de acceso directo.
2. Mientras que el Bus PCI, inicializa los registros de dirección base PCI.
3. El Bus PCI accede a la FIFO interna, seleccionando el espacio de dirección base.
4. Con lo que se direcciona el espacio de memoria local y su rango para acceder al Bus Local.

5.2.3 TRANSFERENCIA DMA (ACCESO DIRECTO A MEMORIA).

El controlador DMA del PCI 9054, lee o escribe en la memoria PCI a o desde la memoria local.
El PCI 9054 posee dos canales independientes de acceso directo a memoria, capaces de transferir datos desde:

- El Bus Local al Bus PCI.
- El Bus PCI al Bus Local.

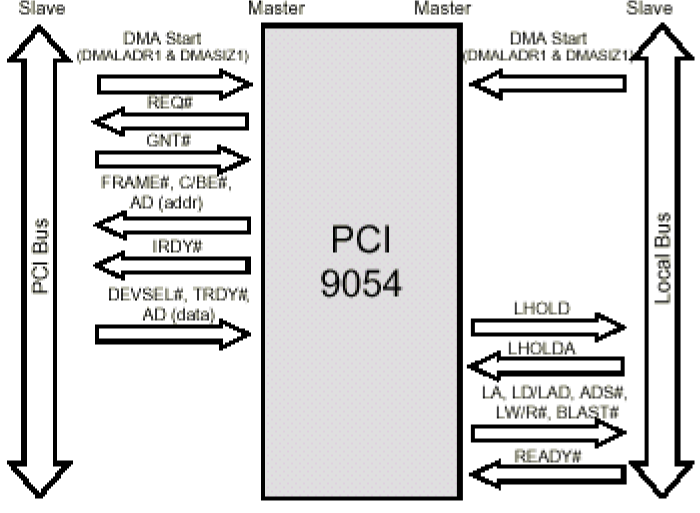


Figura 5.11: Ciclo DMA del Bus PCI al Bus Local.

Cada canal consiste en un controlador DMA y una FIFO bidireccional dedicada. Ambos canales soportan transferencia de bloques, transferencia “Scatter / Gather” [1], con o sin señal de fin de transferencia(EOT#), pero sólo el canal 0 de DMA soporta el modo de demanda DMA.

Además, ambos canales pueden ser programados para:

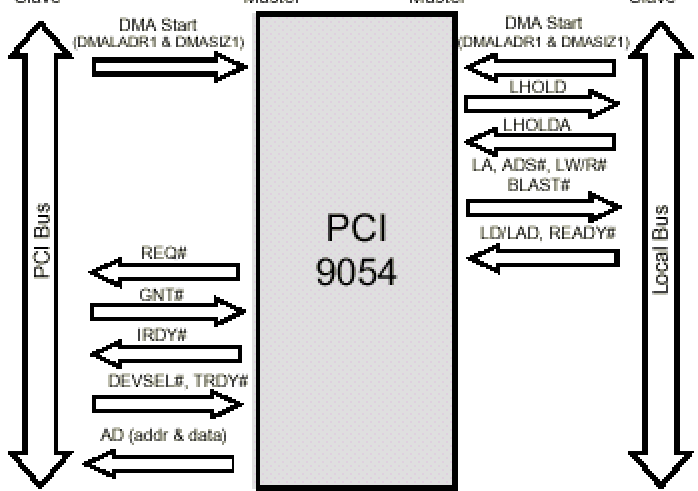


Figura 5.12: Ciclo DMA del Bus Local al Bus PCI.

- Operar con un ancho de Bus Local de 8, 16 o 32 bits.
- Uso de 0 a 15 estados internos de espera.
- Activación / desactivación de los estados internos de espera (Bus Local).
- Activación / desactivación de transferencia en modo ráfaga (Bus Local).
- Limitación de ráfaga a 4-Lwords (Bus Local).
- Mantener constante la dirección local o incrementarla.
- Realizar escritura en la memoria PCI e invalidarla o realizar un ciclo de escritura normal en la memoria PCI.
- Detener la transferencia local con o sin la señal BLAST#.

[1] Scatter: Esparcir; Gather: Recoger.

- Activar la interrupción PCI (INTA#) o la interrupción Local(LINT#), cuando la transferencia DMA se completa o cuando el contador haya llegado a su límite durante el modo de transferencia “Scatter/Gather”(1).

5.2.3.1 Ciclos DMA de dirección dual PCI

El PCI 9054 soporta ciclos de dirección dual PCI (DAC) , es decir que puede generar direcciones de 64 bits (necesario si haymásde 4-GB de memoria) sin necesidad de utilizar dos pulsos de reloj, esto sólo se puede activar cuando se están haciendo transferencias block DMA o Scatter/Gatter y para utilizarlo hay que cambiar los registros DMADAC0 y DMADAC1 o bien DMAMODE0[18] y DMAMODE1[18].

5.2.3.2 Modo Block DMA

Para este tipo de transferencia el procesador maestro fija las direcciones PCI y local el tamaño de la transferencia y la dirección de la misma, entonces afirma el bit DMA Start (DMACSR0[1], DMACSR1[1]) para indicar que se comienza la transferencia. El PCI 9054 pide el bus PCI y el local y realiza la transferencia. Una vez que esta se ha completado el PCI 9054 afirma los bits de transferencia realizada (DMACSR0[4]=1 y/o DMACSR1[4]=1).

Los registros antes mencionados son accesibles desde los buses PCI y local (Figura 5.13).

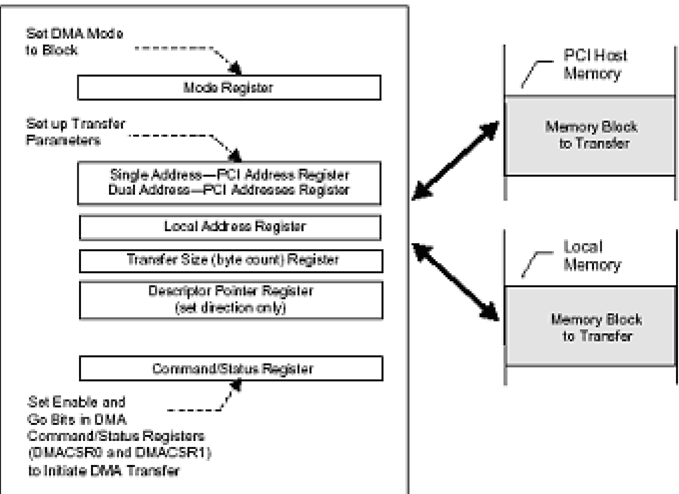


Figura 5.13: Inicialización de un bloque DMA (acceso dual o simple)

Durante una transferencia DMA el PCI 9054 actúa como maestro de los buses local y PCI para un acceso simultaneo a ambos, pero puede ceder los buses en ciertos casos aunque se este realizando una transferencia.

Cederá el bus PCI si:

- La cola esta llena
- La cola esta vacía
- Se termina la transferencia
- Se alcanza el tiempo máximo de latencia del bus
- Se activa la señal STOP#

Cederá el bus local si:

- La cola está llena
- La cola está vacía
- Se termina la transferencia
- Existiendo un tiempo máximo de latencia del bus local este se alcanza
- Se activa la señal BREQ#

5.2.3.2.1 Ciclo de acceso dual en Block DMA

El PLX 9054 también soporta DAC en el modo Block DMA. Este tipo de direccionamiento estará siempre activado a menos que los registros DMADAC0 o DMADAC1 contengan el valor 0x00000000, en cuyo caso se haría single arddess cycle (SAC).

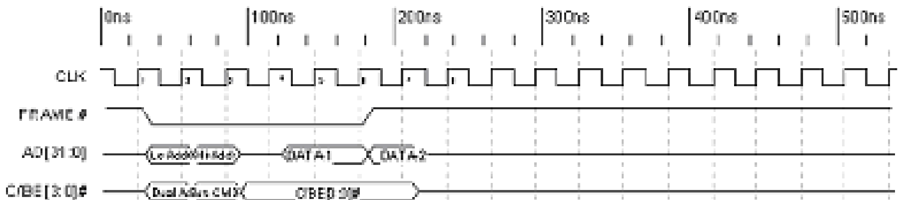


Figura 5.14: Cronograma de acceso dual

5.2.3.3 Modo DMA Scatter/Gatter

El modo de transferencia Scatter/Gatter (esparcir/recojer), como se explico en el capitulo de DMA es útil cuando los bloques de memoria que se quieren transmitir no están es espacios contiguos. Por lo que en este modo de transferencia, el procesador maestro, lo primero que hace es crear unos descriptores de los bloques a transmitir y los guarda en la memoria local o principal. Estos bloques están compuestos por las direcciones locales y PCI, tamaño de transferencia, dirección de la transferencia y dirección del siguiente bloque descriptor. Los bloques descriptores se van cargando uno tras otro en los registros de configuración de la tarjeta para definir la nueva transferencia. El ultimo bloque descriptor tiene el bit de fin de transferencia (End of Chain) activado para indicar que la transferencia debe terminar. Una vez creados los descritores, para realizar la transferencia, el bus maestro:

- Activa el modo Scatter/Gatter.
  (DMAMODE0[9]=1 y/o DMAMODE1[9]=1)

- Activa la direccion del bloque inicial en los registros del PCI 9054
  (DMADPR0 y/o DMADPR1)

- Inicia la transferencia activando los bits de control
  (DMACSR0[1:0], DMACSR1[1:0])

Una vez cargados estos parámetros el PCI 9054 lee el primer bloque e inicia la transferencia de datos, continua leyendo bloques y enviando hasta que detecte que ha llegado el final (DMADPR0[1] y/o DMADPR1[1] activados), entonces completa la transferencia del bloque actual que se supone será el ultimo y activa los bits de transferencia realizada (DMACSR0[4] y/o DMACSR1[4]), entonces el PCI 9054 activa una interrupción PCI y/o una interrupción local.

EL PCI 9054 también puede ser programado para generar interrupciones después de leer cada uno de los bloques.

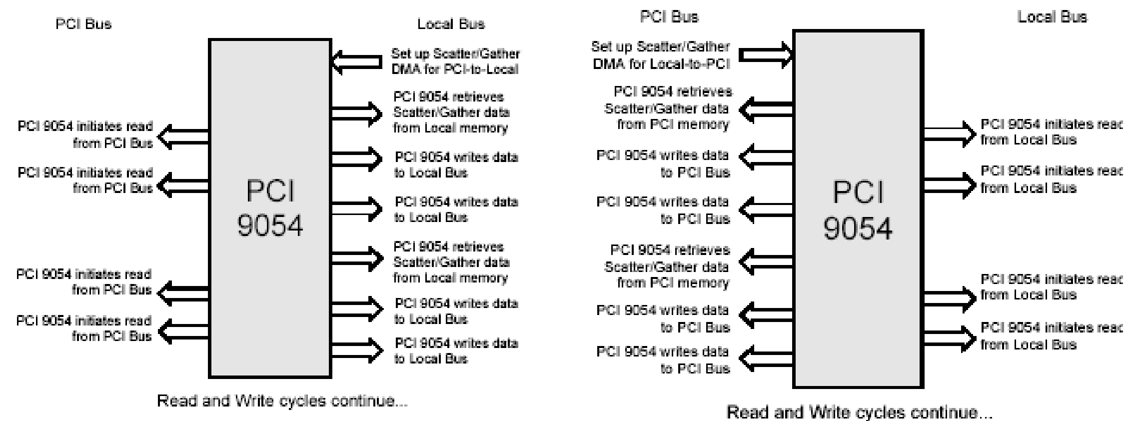


Figura 5.15: Ciclo de lectura Scatter/Gatter desde el bus PCI al bus Local

Figura 5.16: Ciclo de lectura Scatter/Gatter desde el bus Local al bus PCI

5.2.3.3.1 Acceso a memoria con dirección dual en DMA Scatter/Gatter

En este tipo de transferencia el PCI 9054 sólo puede realizar acceso dual a memoria para transferencias de datos. Los descriptores de los bloques a transmitir deben estar contenidos dentro de los 4 GB que se pueden direccionar con un acceso a memoria de este tipo (32 bits).
Existen tres maneras diferentes de realizar DAC en DMA Scatter/Gatter. Asumimos que los descriptores de los bloques están en el bus PCI.

- DMADAC0 y/o DMADAC1 contienen un valor distinto de cero y DMAMODE0[18] y/o DMAMODE1[18] contienen 0. En este caso se realiza acceso simple de memoria (SAC, single address cycle) con un bloque descriptor de 4 lwords de la memoria PCI y una transferencia DMA con DAC (dual address cycle) en el bus PCI (ver figura 5.17).

- DMADAC0 y/o DMADAC1 contienen 0x00000000 y DMAMODE0[18] y/o DMAMODE1[18] contienen 1. En este caso se realiza una SAC con un bloque descriptor de 5 lwords de la memoria PCI y una transferencia DMA con DAC en el bus PCI (ver figura 5.18).

- DMADAC0 y/o DMADAC1 contienen un valor distinto de cero y DMAMODE0[18] y/o DMAMODE1[18] contienen 1. En este caso se realiza una SAC con un bloque descriptor de 5 lwords desde la memoria PCI y una transferencia DMA con DAC en el bus PCI , además el quinto descriptor sobrescribe el valor de DMADAC0 y/o DMADAC1. (ver figura 5.18)

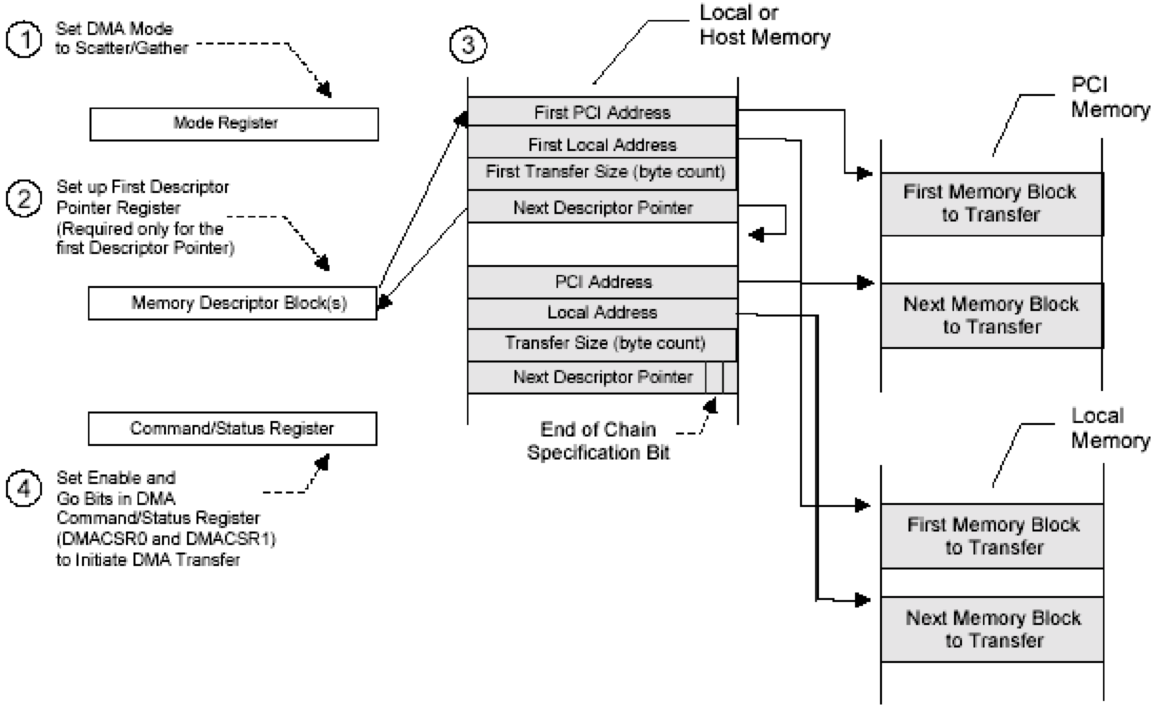


Figura 5.17: Inicialización del Descriptor para modo Scatter/Gatter con SAC/DAC en PCI

5.2.3.4 Escritura e invalidación de memoria en DMA

En este modo de operación el PCI 9054 espera hasta que haya leído el numero de Lwords especificado en el registro de tamaño de cache requerido para rellenar una línea de la caché antes de empezar el acceso al bus PCI. Esto asegura que se escriba una línea completa en cuando el bus maestro recibe el control de bus. Si el destino desconectase mientras se está haciendo la transferencia el PCI 9054 completa esa línea de caché usando modos de transferencia normales antes de volver a realizar escritura e invalidación de memoria de nuevo.

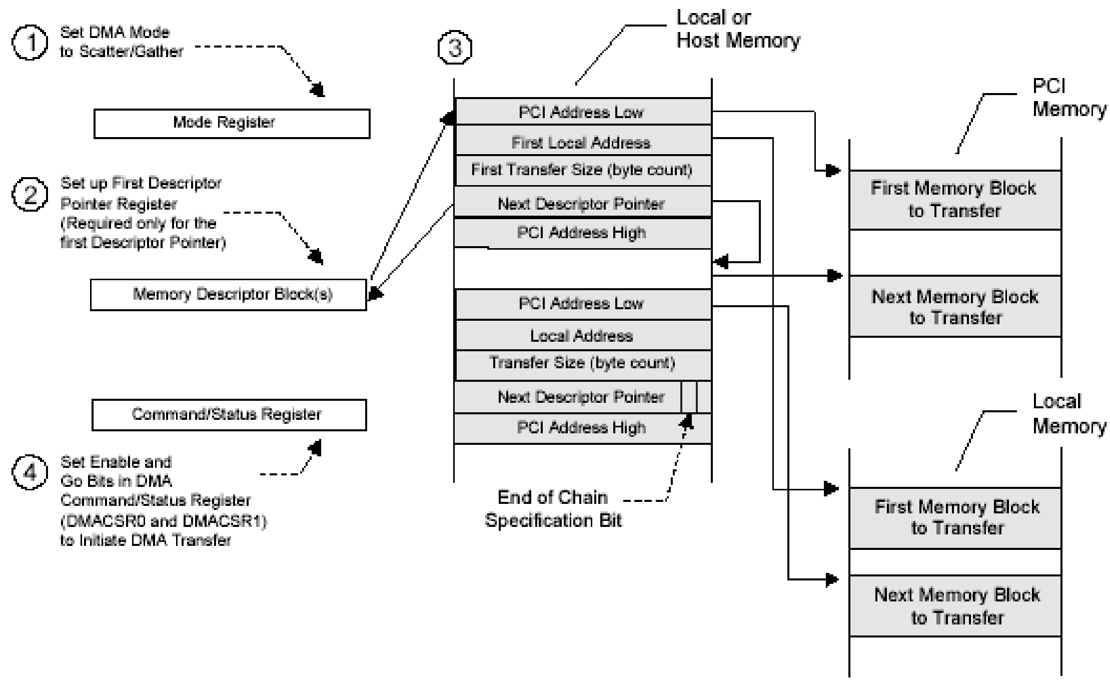


Figura 5.18: Inicialización del Descriptor para modo Scatter/Gatter con DAC en PCI

Si existe un ciclo de lectura e invalidación en proceso este continúa rafagueando si la nueva línea de la caché ya esta preparada. En caso contrario cede el control del bus hasta que la línea de cache este completa. Si el último bloque a transmitir no llega a completar una línea de caché el PCI 9054 transmite ese bloque con escrituras normales a memoria.

El PCI 9054 soporta líneas de caché de 8 o de 16 Lwords , este tamaño está especificado en PCICLSR[7:0]. Para activar este tipo de escritura hay que activar lo bits DMAMODE0[13] y/o DMAMODE1[13] y el bit de escritura e invalidación de memoria (PCICR[4]) ha de estar afirmado.

5.2.3.4.1 Cancelación de transferencia

A parte de cancelar una transferencia mediante la señal de EOT# (end of transmision), se puede cancelar la transferencia de un bloque:

- Poniendo a cero el bit de enabled del canal (DMACSR0[0]=0 y/o DMACSR1[0]=0)

- Activando el bit de cancelación del canal (DMACSR0[2]=1 y/o DMACSR1[2]=1). Después de activar el bit, pueden terminarse una o dos transferencias.

- Esperando hasta que se active el bit de transferencia finalizada. (DMACSR0[4]=1 y/o DMACSR1[4]=1)

Si se cancela antes de que una transferencia DMA esté en progreso se cancelara la siguiente que se realice.

5.2.3.5 Prioridad en DMA

La prioridad de un canal de DMA se puede especificar cambiando los bits de prioridad MARBR[20:19]. Se pueden especificar las siguientes propiedades:

- Rotacional (Van rotando para hacer las transferencias) MARBR[20:19]=00
- Prioridad en el canal DMA 1: MARBR[20:19]=01
- Prioridad en el canal DMA 0: MARBR[20:19]=10

5.2.3.6 Interrupciones DMA

Un canal DMA puede generar interrupciones en el bus PCI o en el bus de usuario cuando la transferencia esté completa o cada vez que se transfiera un bloque en el modo scatter/gatter.

Para especificar que la interrupción se producirá en el puerto PCI se activa DMAMODE0=[17] y/o DMAMODE1=[17], si la quisiéramos generar en el bus local se negarían los mismos bits. El procesador local puede leer los bits de interrupcion del canal 0 (INTCSR[21]) o del canal 1 (INTCSR[22]) para saber cual hay pendiente una interrupcion.

Se pueden leer los bits DMACSR0[14] y/o DMACSR1[14] (bits de transferencia terminada) para saber si se ha producido la interrupción por que la transferencia terminó o por que se termino de transferir un bloque descriptor.

Para eliminar una interrupción se usan los bits DMACSR0[3] y/o DMACSR1[3].

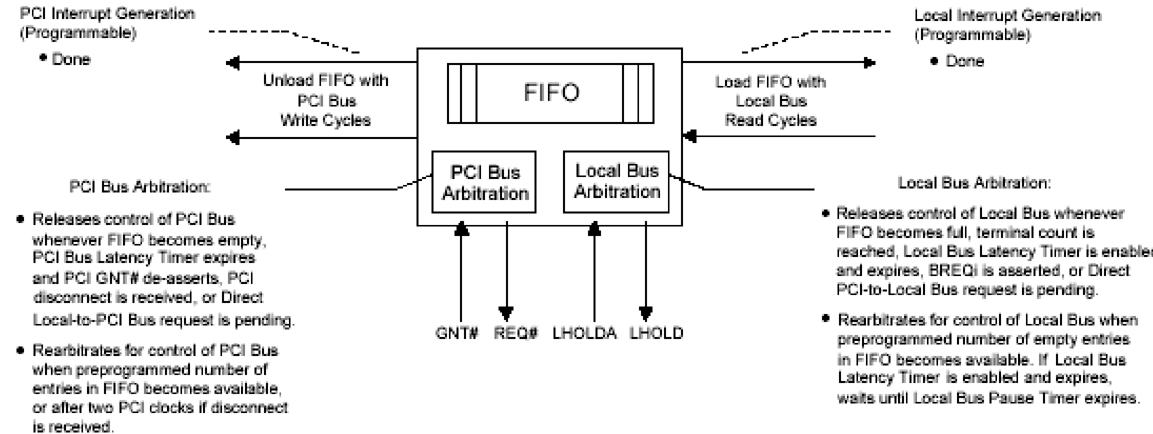


Figura 5.19: Operación de transferencia DMA del bus Local al PCI

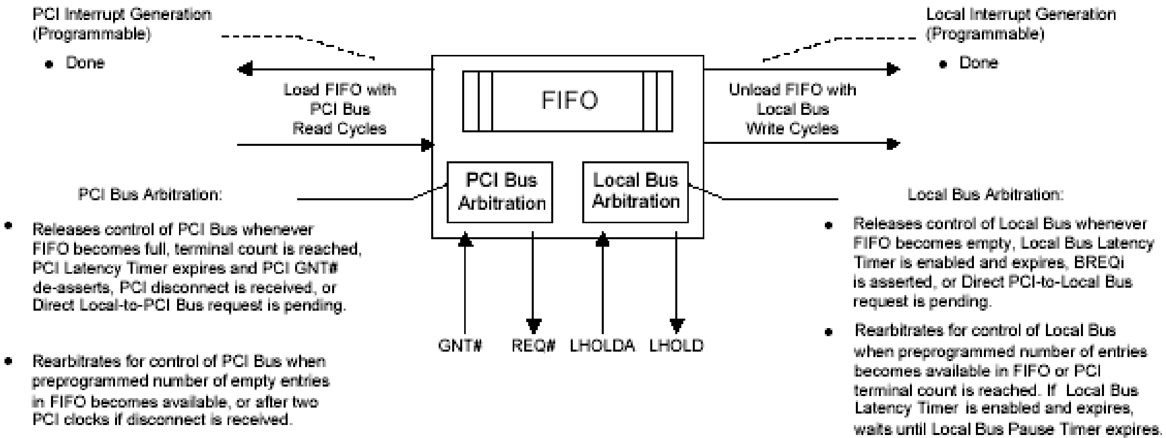


Figura 5.20: Operación de transferencia DMA del bus PCI al Local

5.2.3.8 DMA modo demanda , canal 0

El modo demanda DMA es la manera más sencilla de permitir al bus Local iniciar transferencias DMA en modo maestro. En ella, el bus local pone a 0 la señal DREQ0# para indicar al controlador PLX9054 que desea el control de bus. Este responde con la señal LHOLD para indicarle que le cede el bus y espera a recibir la confirmación LHOLDA. Una vez que el bus local tiene el control de bus puede iniciar transferencias de lectura/escritura hacia el bus PCI.

Los bits de modo de terminación rapido/lento (DMAMODE0[15] y/o DMAMODE1[15]) determina el numero de Lwords a transferir después de que la señal DREQ# se desactive.

Si la señal BLAST# es necesaria para la transferencia del último Lword (bit[15]=0) el controlador transfiere uno o dos Lwords. Si DREQ0# está desactivada durante la fase de dirección de la primera transferencia el controlador DMA completa el Lword actual. Si DREQ0# esta desactivada durante cualquier otra fase que no sea la de dirección el controlador completa el Lword actualmásuno adicional (esto permite que la señal BLAST# se active mientras se transmite el último Lword).

DREQ0# controla sólo el número de Lwords a transferir. Para un bus de 8 bits, el PCI 9054 deja el bus después de haber transmitido el último byte del Lword , para un bus de 16 bits , el PCI 9054 deja el bus después de transferir el último word del Lword.

5.2.3.9 Señal de fin de transferencia EOT#

Los bits DMAMODE0[14] y/o DMAMODE1[14] determinan el número de Lwords que se transferirán después de que el controlador DMA active EOT#, dicha señal debe estar activa sólo cuando el PCI 9054 esté controlando un bus.

Si la señal BLAST# no es necesaria para que el último Lword se transfiera (DMAMODE0[15]=1 y/o DMAMODE1[15]=1) el controlador DMA dejará el bus y terminará la transferencia después de recibir la señal READY# o el contador interno de espera se decremente hasta cero. Si el controlador DMA estuviera enviando algo que no fuera la última fase de datos del envio BLAST# no estará activa.

Si la señal BLAST# es necesaria para que el último Lword se transfiera (DMAMODE0[15]=0 y/o DMAMODE1[15]=0) el controlador DMA transfiere una o dos Lwords dependiendo del ancho del bus. Si EOT# está activa , el controlador DMA completa el Lword que esté transmitiendo y uno mas, lo que permite a BLAST# ser afirmada durante la transmisión del último Lword. Si la cola FIFO esta llena o vacía después de la fase de datos en la cual EOT# está afirmada el último Lword no es transmitido.

5.2.3.10 Arbitración DMA

EL controlador PCI 9054 deja el control del bus local (desactivando LHOLD) cuando ocurre uno de los siguientes eventos:

- El temporizador de inactividad se agota
- BREQ! Está desactivado (BREQ! Puede estar activado, desactivado o controlado por un temporizador de inactividad antes de que se deje el bus local)
- Haya pendiente un acceso de la tarjeta a PCI
- Se reciba una señal EOT# activa.

El controlador DMA deja el control del bus PCI cuando ocurre alguno de los siguientes eventos:

- Su cola está llena o vacía
- Se consume el tiempo de espera del temporizado (PCILTR[7:0])

El controlador DMA cuando desactiva su bus PCI lo hace por lo menos durante dos pulsos de reloj.

5.2.3.11 Tiempos de espera para el bus local y los temporizadores

Los tiempos de espera del bus local y los temporizadores de pausa son programables con el registro de arbitración MARBR[7:0,15:8] Si el temporizador de espera del bus local está activado y se consume MARBR[7:0] el PCI 9054 completa el Lword actual y desactiva LHOLD . Después de que esta parada programada se termine , se reactiva LHOLD y se reanuda la transferencia cuando se recibe LHOLDA. La transferencia al bus PCI continúa hasta que la cola se vacía para una transmisión PCI o se llena para una transmisión del bus PCI al local.

Una transmisión DMA puede pausarse poniendo un cero en el bit de canal activado (no recomendable durante una transferencia)

5.2.4 ACTIVACIÓN DE LAS SEÑALES EN FUNCIÓN DEL ESTADO DE LA FIFO, Y EL MODO DE TRANSFERENCIA.

Las señales que se activan en los buses, según el estado de la FIFO interna del controlador, se describen en la tabla 5.2.

Modo	Dirección	FIFO	Bus PCI	Bus Local
PCI Initiator Write	Bus Local a Bus PCI	Llena	Normal	Desactiva READY#
		Vacía	Desactiva REQ# (Desactiva Bus PCI)	Normal
PCI Initiator Read	Bus PCI a Bus Local	Llena	Desactiva REQ# O estrangula IRDY# (1)	Normal
		Vacía	Normal	Desactiva READY#
PCI Target Write	Bus PCI a Bus Local	Llena	Desconecta TRDY# (2)	Normal
		Vacía	Normal	Desactiva LHOLD, activa BLAST# (3)
PCI Target Read	Bus Local a Bus PCI	Llena	Normal	Desactiva LHOLD, activa BLAST#
		Vacía	Estrangula TRDY#	Normal
DMA	Bus Local a Bus PCI	Llena	Normal	Desactiva LHOLD, activa BLAST#
		Vacía	Desactiva REQ#	Normal
	Bus PCI a Bus Local	Llena	Desactiva REQ#	Normal
		Vacía	Normal	Desactiva LHOLD, activa BLAST#

Tabla 5.2: Respuesta a FIFO llena o vacía.

Por ejemplo, en una transferencia de escritura PCI Initiator cuando la FIFO esta llena, en el Bus PCI no se responderá con ninguna señal, mientras en el Bus Local se desactivará la señal READY#, que se encontraba activa.
Cuando la FIFO se vacía, en el Bus PCI se desactiva la señal REQ#, desactivando el bus, siendo el Bus Local el que continúe con su funcionamiento normal.

[1] Que estrangule IRDY# depende del bit de modo lectura PCI del PCI Initiator (DMPBAM[4]).
[2] Que estrangule TRDY# depende del bit de modo escritura PCI del PCI Target (LBRDO[27])
[3] Que LHOLD se desactive sobre el Bus Local PCI Target depende del bit de modo bus liberado [MARBR[2:1]).

5.3 Modo C de operación.

5.3.1 Ciclos del Bus PCI.

5.3.1.1 Código de comando PCI Target.

Los comandos indican al esclavo el tipo de transición que el maestro necesita realizar. Estos se codifican mediante el bus C/BE[3:0], listados en la tabla 5.3, durante una fase de dirección. Son opcionales excepto los de configuración.

C/BE[3:0]	Comando	Descripción
0000	Interrupción Ack.	Direccionamiento de lectura al sistema controlado por interrupciones.
0001	Ciclo especial(1).	Mensaje por broadcast(2).
0010	Lectura I/O.	Leer datos de un uC mapeado en el espacio I/O.
0011	Escritura I/O.	Escribir datos en I/O.
0100	Reservado.	-
0101	Reservado.	-
0110	Lectura de Memoria.	Leer datos de un uC mapeado en el espacio de memoria.
0111	Escritura en Memoria.	Escribir datos en memoria.
1000	Reservado.	-
1001	Reservado.	-
1010	Lectura de configuración.	Leer datos de un uC mapeado en el espacio I/O.
1011	Escritura en configuración.	Escribir datos en configuración.
1100	Múltiple lectura de memoria.	Semejante a la lectura de memoria, pero se pueden leer múltiples líneas caché (3).
1101	Ciclo de dirección dual.	Para transferir direcciones de 64 bits.
1110	Línea de lectura de memoria.	Semejante a la lectura de memoria, además, indicando que 3 o más datos se pueden leer de la línea caché.
1111	Escritura de memoria e invalidar.	Semejante a la escritura de memoria, pero garantiza una transferencia mínima de toda la capacidad de la línea caché.

Tabla 5.3: Códigos de comando.

Todos los accesos de lectura o de escritura al controlador PCI 9054 pueden ser acceso Byte (8bits), Word(16 bits) o Lword (32 bits) y todos los accesos I/O al PCI 9054 tienen un límite de tamaño Lword.

1 Ciclo especial: Sirve para enviar un mensaje simple con el mecanismo de broadcast (envió a todos) al PCI.

2 Emisión.

3 Parte de la memoria dinámica principal se guarda en la memoria caché para conseguir accesos más rápidos. La parte más pequeña que se puede guardar se llama línea caché.

5.3.1.2 Árbitro PCI.

Este elemento es imprescindible, ya que pueden existir varios dispositivos conectados al Bus PCI del ordenador. Si, por ejemplo, uno de ellos (una tarjeta de adquisición de datos) tuviese que enviar una gran cantidad de datos a la memoria principal, monopolizaría el Bus PCI durante bastante tiempo, impidiendo que la tarjeta de vídeo (entre otros posibles dispositivos) recibiese la información que necesita de la CPU, por lo que aparecería una situación de mal funcionamiento en el sistema.
Debido a esto, existe este elemento, que impide que haya dispositivos que transfieran datos durante mucho tiempo, permitiendo de esta manera que todos los dispositivos vayan enviando datos durante intervalos determinados de tiempo.

Una ventaja que posee el Bus PCI, es que la negociación entre el árbitro PCI y un dispositivo iniciador se puede realizar mientras otro dispositivo tiene el control del bus y está en el transcurso de una transacción. A esto se le llama arbitraje oculto[1], que evita tener que perder un tiempo innecesario en la adjudicación del bus cuando éste estuviese libre. Así, en cuanto un dispositivo cede el control del bus, rápidamente otro dispositivo toma el control, consiguiendo que el Bus PCI esté en uso en todo momento.

El controlador PCI 9054 actuando como árbitro del Bus PCI, activa la señal REQ# para pedir el control del bus PCI, cuando el árbitro determina que el controlador puede usar el bus, activa la señal GNT# del mismo.
El maestro puede comenzar la transición cuando GNT# esta activo y el bus se encuentra en espera, independientemente de REQ#. Una vez este activo GNT#, se puede desactivar mediante:

1. Si GNT# se desactiva y FRAME# se activa en el mismo flanco de reloj, la transición es válida.
2. Se puede desactivar coincidiendo la activación de otra GNT#, si el bus no esta en estado de espera. En caso contrario, es necesario un ciclo de reloj entre la desactivación y la activación de GNT#.
3. Mientras FRAME# este activo, GNT# se puede desactivar por orden de un servicio de alta prioridad de un maestro.

5.3.2 Ciclos de Bus Local.

5.3.2.1 Arbitración del Bus Local.

El PCI 9054 actuando como árbitro del Bus Local, activa LHOLD para pedir el bus Local, este lo posee cuando LHOLD y LHOLDA están activados. Cuando el controlador reconoce la activación de la señal BREQi durante una transferencia de ciclo de escritura PCI Target o una Transferencia DMA, libera el Bus Local tras 2 transferencias Lword, desactivando LHOLD, y dejando flotantes las salidas del Bus Local si cualquiera de las siguientes condiciones existen:

- BREQi esta activado y habilitado.
- El bloqueo esta habilitado y el temporizdor de latencia del Bus Local esta habilitado y expira (MARBR[27,7:0]).

[1] hidden arbitration

El árbitro Local puede ahora, conceder el Bus Local a otro Bus Maestro que lo pida, cuando el PCI 9054 recibe la señal de LHOLDA, se queda con el control del Bus Local y continúa con la transferencia.

5.3.2.2 PCI Target.

El PCI 9054 soporta transferencia de acceso en modo burst [1] memoria-mapeada e I/O-mapeada, y transferencia de acceso en modo singular al Bus Local desde el Bus PCI, una lectura de 16 Lword (64 bytes) en la FIFO de lectura y una escritura de 32 Lword(128 bytes) en la FIFO de escritura.
Los registros de dirección base PCI, están provistos para configurar la localización del dispositivo en la memoria PCI y el espacio I/O. Se permiten tres espacios; Espacio 0, Espacio1 y extensión ROM, como se explicó anteriormente.

Para un ciclo de lectura singular, el PCI 9054 lee un Lword o una parte del Lword en el Bus Local. El PCI 9054 se desconecta tras una transferencia de todos los accesos de I/O del Bus Local.
Para la más alta tasa de transferencia de datos, el PCI 9054 soporta escrituras posteriores y puede ser programado para prescribir durante un ciclo de lectura tipo ráfaga.
El tamaño de pretransferencia, cuando este está activo, puede ir desde 1 a 16 Lwords o hasta que el Bus PCI detenga la petición del bus.

En modo pretransferencia continuo, el PCI 9054 pretransfiere tanto espacio FIFO como se permita y la detiene cuando el Bus PCI termina la solicitud del Bus.
Si la pre-lectura esta desactivada el PCI 9054 se desconecta tras una transferencia de lectura. Además, en el modo pretransferencia, el PCI 9054 soporta el Modo Read Ahead.

Cada espacio Local puede ser programado, para operar en un ancho de 8, 16 o 32 bits del Bus Local.

El PCI 9054 tiene un generador interno de estado de espera, Ready, y una entrada externa de estado de espera, READY#, que puede estar habilitada o no, mediante los registros de configuración internos.
Con o sin estado de espera, el bus Local, independientemente del Bus PCI puede:

- Transferir en modo de ráfaga tantos datos como quiera.
- Transferir en modo de ráfaga, 4 Lwords por ciclo (modo recomendado).
- Realizar un ciclo singular continuo.

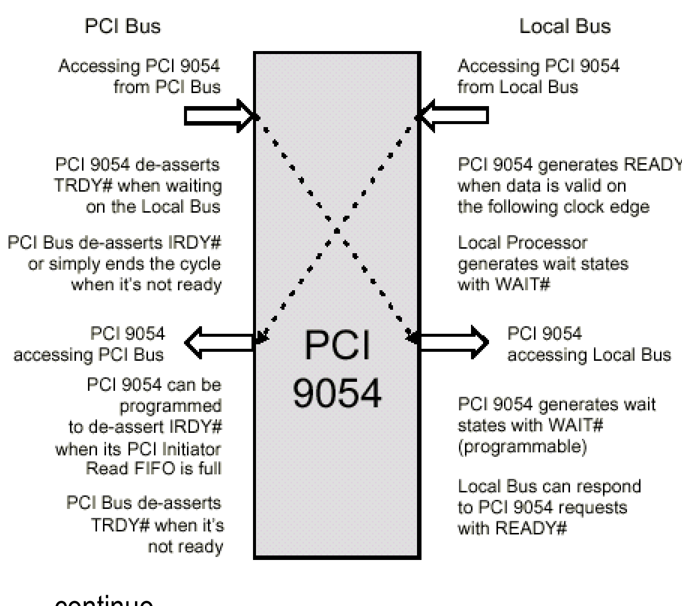


Figura 5.21: Modo Read Ahead.

[1] Ráfaga.

5.3.2.3 Control del estado Wait.

Si el modo Ready esta desactivo, la señal de entrada externa READY#, no tiene efecto en los estados de espera para un acceso local. Los estados de wait se activan internamente por el contador de estado de espera. Este contador se inicializa con el valor del registro de configuración, al comienzo de cada acceso de datos.
Si el modo Ready se activa, este no tiene efecto hasta que el contador de estados de espera llegue a cero. La señal READY# controla el número adicional de estados de espera.
La entrada BTERM# no se muestra hasta que el contador de estados de espera no llegue a cero, por lo que BTERM# hace caso omiso de la señal READY#, mientras BTERM# este habilitado y activo.

5.3.2.3.1 Bus Local.

EL PCI 9054 actúa como un Bus Maestro Local, insertando estados internos de Wait con la señal WAIT#. El procesador Local, activa estados de espera externos, retrasando la señal READY#. Los bits de estado Wait (LBRDO[21:18,5:2], y LBRDI[5:2]), pueden ser usados para programar el número de estados de espera entre la primera dirección-dato, y los siguientes dato a dato en el modo ráfaga.
La señal READY# no tiene efecto hasta que el contador de estados Wait llegue a 0, la señal READY# controla el número de estados de espera siendo desactivados en medio de la transferencia de datos.

5.3.2.3.2 Bus PCI.

El Bus PCI actuando como maestro, acelera IRDY# y actuando como esclavo, acelera TRDY# para activar los estados de espera del Bus PCI.

5.3.2.4 Modo Burst (modo ráfaga) y Continuous Burst.

En las siguientes se referirá como Bterm al registro interno del PCI 9054, y como BTERM# a la señal externa BTERM#.

5.3.2.4.1 Modo Burst y Bterm.

Modo	Burst	Bterm	Resultado
Ciclo Singular	0	0	Un ADS# por dato.
Ciclo Singular	0	1	Un ADS# por dato.
burst-4	1	0	Un ADS# cada 4 datos.
Burst Forever	1	1	Un ADS# por BTERM#.

Tabla 5.4 Burst y Bterm en el Bus Local.

En el Bus Local, las señales BLAST# y BTERM realizan lo siguiente:

- Si el bit de modo Burst esta habilitado, pero el bit de Bterm, no lo está, entonces el PCI 9054 puede transferir en modo ráfaga hasta 4 Lword de límite.

- BLAST#, se activa al comienzo de la cuarta fase de datos Lword (LA[3:2]=11b) y una nueva señal ADS# se activa en el primer Lword (LA[3:2]=00b) de la siguiente transferencia en modo ráfaga.

- Si BTERM# se habilita y activa, el PCI 9054 termina el ciclo Burst de la última fase de datos, sin generar la señal BLAST#. El PCI 9054 genera entonces una nueva transferencia ráfaga comenzando con la señal ADS#, y terminándola como un ciclo normal, con la señal BLAST#

- La entrada BTERM# es válida sólo cuando el PCI 9054 es el Maestro del Bus Local(Modos Target y DMA).

- Como entrada BTERM#, se activa por la lógica externa, lo que conlleva al PCI54 a romper el ciclo Burst.

- BTERM# se usa para indicar que un acceso a memoria sobrepasa el límite de página o requiere un nuevo ciclo de dirección

Si el bit de modo Burst esta deshabilitado, el PCI 9054 realiza lo siguiente:

- En transferencia de 32 bits, transfiere en modo ráfaga 4 Lwords.
- En transferencia de 16 bits, transfiere en modo ráfaga 2 Lwords.
- En transferencia de 8 bits, transfiere en modo ráfaga 1 Lwords.

En cada uno de los tres casos realiza 4 transacciones.

5.3.2.4.2 Modo Ráfaga- 4 Lword (modo recomendado).

Si el bit de modo Burst esta habilitado y el bit de modo Bterm esta deshabilitado, la transferencia en modo ráfaga puede empezar desde cualquier límite Lword y continuar hasta el límite de dirección de 16 bytes.
Después de que el dato que llega al limite, sea transferido, EL PCI 9054 activa un nuevo ciclo de dirección ADS#.

En el Modo Ráfaga Continua, los bits de modo Burst y Bterm se encuentran activos, y el PCI 9054 puede operarmásallá del limite de ráfaga 4-Lword.
El Modo Bterm, activa el PCI 9054 para realizar grandes ráfagas a los distintos dispositivos que puedan aceptar ráfagas demásde 4- Lwords. El PCI 9054 activa un ciclo de dirección y comienza a transferir datos en forma de ráfaga. Si un dispositivo requiere un nuevo ciclo de dirección (ADS#), se puede activar la señal de entrada BTERM#, para causar que el PCI 9054 active un nuevo ciclo de dirección.
La señal de entada BTERM#, reconoce una transferencia normal de datos y pide que un nuevo ciclo de dirección se active(ADS#). La nueva dirección será para la siguiente transferencia de datos.

Si el bit de modo Bterm, se habilita y la señal de BTERM# se activa, el PCI 9054 activa BLAST# sólo si la FIFO de lectura se llena, la FIFO de escritura se vacía, o la transferencia se completa.

5.3.2.4.3 Acceso Lword partido.

Los accesos Lword en el cuál no todos los bytes habilitados activos serán interrumpidos en accesos singulares.
Burst comienza a direccionar, con cualquier limite Lword. Si el Burst comienza a direccionar no se alinea en un limite Lword, el PCI9054 primero realiza un ciclo singular, luego comienza una ráfaga con el límite Lword si permanecen datos que no hayan terminado en totalidad una transferencia Lword.

5.3.2.5 Acceso de lectura al Bus Local.

Para todos los accesos de lectura al Bus Local en ciclo modo singular, el PCI 9054 lee sólo bytes correspondientes habilitados por la petición del PCI Initiator. Para todos los ciclos de lectura modo ráfaga, el PCI 9054 pasa todos los bytes y puede ser programado para:

- Realizar una pretransferencia.
- Realizar una transferencia en modo Read Ahead.
- Generar estados internos de espera.
- Habilitar el control externo de espera (READY#).
- Habilitar el tipo modo Ráfaga para realizar la transferencia.

5.3.2.6 Acceso de escritura al Bus Local.

Para la escritura del Bus Local, sólo los bytes especificados por el Bus PCI Maestro o por el controlador PCI 9054 DMA se escriben.

5.3.3 Big Endian/ Little Endian.

5.3.3.1 Modo Little Endian del Bus PCI.

En este modo, el Bus PCI es un bus Little Endian, es decir, la dirección es invariante y los datos son alineados en tamaño Lword por el bytemásbajo.

Byte Number	Byte Lane
0	AD[7:0]
1	AD[15:8]
2	AD[23:16]
3	AD[31:24]

Tabla 5.5: Disposición de los bytes en el modo Little Endian del Bus Local.

5.3.3.2 Modo Big /Little Endian del Bus Local.

En este, el Bus Local del PCI 9054 puede ser programado para operar en modo Big o Little Endian.

Mode	Byte Number		Byte Lane
	Big Endian	Little Endian	
C	3	0	LD[7:0]
	2	1	LD[15:8]
	1	2	LD[23:16]
	0	3	LD[31:24]

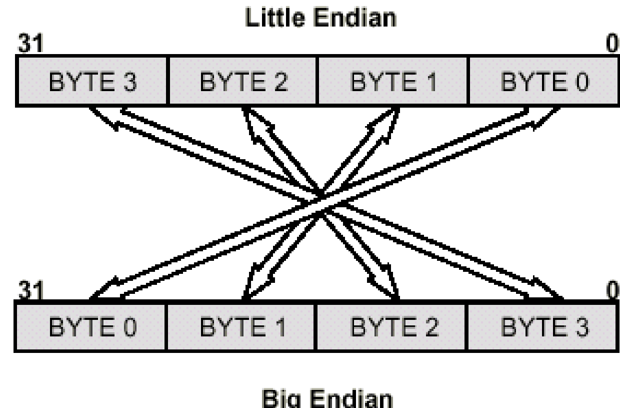
Tabla 5.6: Número de byte y disposición cruzada de bytes.

En el modo Big Endian, el PCI 9054 transpone los datos por grupos de bytes. El dato es transferido como se muestra en la figura 5.22.

Burst Order	Byte Lane
First transfer	Byte 0 appears on Local Data [31:24]
	Byte 1 appears on Local Data [23:16]
	Byte 2 appears on Local Data [15:8]
	Byte 3 appears on Local Data [7:0]

Tabla 5.7: Disposición de los bytes en la transferencia Big /Little Endian del Bus Local.

5.3.3.2.1 Modo Big Endian de 32 bits del Bus Local.



En este modo de transferencia el dato se alinea por el camino del bitmásalto, es decir, el bloque del bit 31 al 24 se coloca como bloque de 7 a 0 y así sucesivamente.

Figura 5.22: Transferencia Big/ Little Endian de 32 bits.

5.3.3.2.2 Modo Big Endian de 16 bits del Bus Local.

En la transferencia Big Endian de 16 bits, el PCI 9054 puede ser programado para usar la palabra alta o baja.
Se divide el dato de 32 bits en dos bloques, desde el 31 al 16 y desde el 15 al 0. Cada uno de estos bloques transpone sus bytes, por ejemplo, en el bloque del 15 al 0, cogemos el byte del bit 0 al 7 y lo colocaremos como dato del bit 8 al 15.

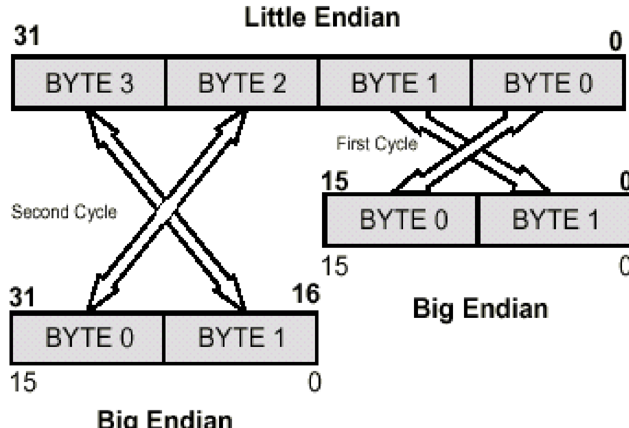


Figura 5.23: Transferencia Big/ Little Endian de 16 bits.

5.3.3.2.3 Modo Big Endian de 8 bits del Bus Local.

En la transferencia Big Endian de 8 bits, el PCI 9054 también puede ser programado para usar la palabra alta o baja.

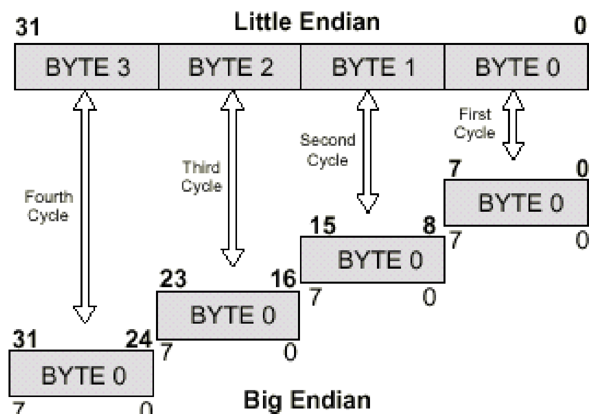


Figura 5.24: Transferencia Big/Little Endian de 8 bits.

5.3.3.3 Modo de acceso Bit/ Little Endian del Bus Local.

Para cada uno de los modos de transferencia, el bus Local del PCI 9054 puede ser independientemente programado para operar en modo Big Endian o Little Endian.

- Acceso del Bus Local a los registros de configuración del PCI 9054.

- Acceso PCI Target del PCI 9054 del CPI al espacio 0 o 1 de la dirección Local.
- Acceso PCI Target del PCI 9054 del CPI a la expansión ROM.
- Acceso del Canal 0 y 1 DMA al Bus Local.
- Acceso PCI Initiator Bus PCI.

Para accesos al Bus Local a los registros internos de configuración y accesos PCI Initiator, usa BIGENDIAN# para cambiar dinámicamente el modo Endian.

5.3.4 EEPROM serie.

La EEPROM puede ser usada para cargar la configuración de nuestro controlador, lo que será de gran ayuda por cargar información que es única para cada dispositivo, tales como el Device ID o el Vendor ID de una tarjeta de red.
Durante la inicialización de la EEPROM serie, el PCI 9054 responde al acceso PCI Target con Retry, respondiendo a un procesador local retrasando el conocimiento del ciclo (READY#).

5.3.4.1 Operación EEPROM serie.

Tras el reset, el PCI 9054 intenta leer la EEPROM serie para determinar su presencia. Un bit de comienzo se pone a 0 para indicar que la EEPROM serie está presente.
La primera Lword, se chequea entonces para verificar que esta programada.
Si la primera Lword (33 bits) es todo 1, indica que una EEPROM en blanco se encuentra presente, en cambio, si es todo 0, indica que no hay EEPROM. En ambas condiciones, el PCI 9054 rescribe los valores por defecto
La EEPROM serie puede ser escrita o leída desde el Bus PCI o el Bus Local, los bits de registro de control de EEPROM serie controlan los pines del PCI 9054 que habilitan la lectura o escritura de los bits de datos de la EEPROM serie.

5.3.4.2 Carga de los registros en la EEPROM serie.

Los registros listados en la tabla 4.8 son cargados desde la EEPROM serie tras un reset. La EEPRON serie se organiza en datos de tamaño Word (16 bit). El controlador PCI 9054 carga primero los bits más significativos (MSW[31:16]), comenzando desde el bit más significativo [31]. Tras ellos, el PCI 9054 carga los bits menos significativos (LSW[15:0]), comenzando por el bit más significativo [15].
Por lo tanto el PCI 9054 carga el Device ID, Vendor ID, el código de clase y demás registros.
Los valores de la EEPROM serie pueden ser programados usando un programador de datos I/O.
Los valores pueden ser también programados usando la función VPD o a trabes de los registros de control de la EEPROM serie (CNTRL).

El registro CNTRL permite programar un bit cada vez. Para volver a leer el valor desde la EEPROM serie, el bit CNTRL[27] o la función VPD debiera ser utilizada.
Mediante un uso total del VPD, el diseñador puede realizar lecturas y escrituras desde o a la EEPROM serie, 32 bits cada vez, estos valores deberían ser programados en el orden listado en la tabla 5.8. Los 34 registros de 16 bits listados en la tabla son guardados de forma secuencial en la EEPROM serie.

Offset EEPROM	Descripción	Bits de registro afectados.
0h	Device ID	PCIIDR[31:16]
2h	Vendor ID	PCIIDR[15:0]
4h	Código de clase	PCICCR[23:8]
6h	Código/ Revisión de clase	PCICCR[7:0 / PCIREV[7:0]]
8h	Máxima latencia/ mínima concesión	PCIMLR[7:0] / PCIMGR[7:0]
Ah	interrumpir pin/ interrumpir línea de ruteo	PCIIPR[7:0] / PCIILR[7:0]
Ch	MSW de Mailbox 0 (definido por el usuario)	MBOX0[31:16]
Eh	LSW de Mailbox 0 (definido por el usuario)	MBOX0[15:0]
10h	MSW de Mailbox 1 (definido por el usuario)	MBOX1[31:16]
12h	LSW de Mailbox 1 (definido por el usuario)	MBOX1[15:0]
14h	MSW de rango para el espacio 0 de dirección PCI a Local	LAS0RR[31:16]
16h	LSW de rango para el espacio 0 de dirección PCI a Local	LAS0RR[15:0]
18h	MSW de dirección base local(Remapeo) para el espacio 0 de dirección PCI a Local Mailbox 0	LAS0BA[31:16]
1Ah	LSW de dirección base local(Remapeo) para el espacio 0 de dirección PCI a Local Mailbox 0	LAS0BA[15:0]
1Ch	MSW del registro de arbitración del modo DMA.	MARBR[31:16]
1Eh	LSW del registro de arbitración del modo DMA	MARBR[15:0]
20h	MSW de la dirección de escritura protegida de la EEPROM	PROT_AREA[15:0]
22h	LSW de la dirección de escritura protegida de la EEPROM	LMISC[7:0] / BIGEND[7:0]
24h	MSW de Rango de expansión ROM para PCI a Local.	EROMRR[31:16]
26h	LSW de Rango de expansión ROM para PCI a Local.	EROMRR[15:0]
28h	MSW de dirección base local (Remapeo) para la de expansión ROM para PCI a Local.	EROMBA[31:16]
2Ah	LSW de dirección base local (Remapeo) para la de expansión ROM para PCI a Local.	ERONBA[15:0]
2Ch	MSW de los descriptores de región de bus para PCI a Local.	LBRD0[31:16]
2Eh	LSW de los descriptores de región de bus para PCI a Local.	LBRD0[15:0]
30h	MSW de rango para PCI Initiator a Local.	DMRR[31:16]
32h	LSW de rango para PCI Initiator a Local.	DMRR[15:0]
34h	MSW de dirección base local para PCI Initiator a memoria PCI.	DMLBAM[31:16]
36h	LSW de dirección base local para PCI Initiator a Local.	DMLBAM[15:0]
38h	LSW de dirección base local para PCI Initiator a configuración I/O PCI.	DMLBAI[31:16]
3Ah	MSW de dirección base local para PCI Initiator a configuración I/O PCI.	DMLBAI[15:0]
3Ch	MSW de dirección base PCI para PCI Initiator a PCI.	DMPBAM[31:16]
3Eh	MSW de dirección base PCI para PCI Initiator a PCI.	DMPBAM[15:0]
40h	MSW de los registros de dirección de configuración para PCI Initiator a configuración I/O PCI.	DMCFGA[31:16]
42h	LSW de los registros de dirección de configuración para PCI Initiator a configuración I/O PCI.	DMCFGA[15:0]

Tabla 5.8: Registros que son cargados desde la EEPROM serie.

5.4. APLICACIONES DEL PCI 9054.

Las aplicaciones que se pueden conseguir con el controlador de bus PCI 9054, son adaptadores para aplicaciones Telecom y de establecimiento de red.

Estas aplicaciones, incluyen comunicaciones de alta complejidad como WAN /LAN, tarjetas módem de alta velocidad, tarjetas de retransmisión de marcos, ruteadores y switches.

La flexibilidad del PCI 9054 que posee tolerancia de 3.3V y 5V, combinados con operaciones del Bus Local de hasta 50 MHz, es ideal para operaciones actuales y futuras con procesadores Power QUICC.

El PCI 9054 también provee soporte para canales IDMA de MPC850 o MPC860 para intercambio de datos entre los puertos de I/O del MPC850 o el MPC860 y el Bus PCI.

Además, el PCI 9054 hace uso también de Data Pipe Architecture technology (1), permitiéndole una capacidad para transferir en modo ráfaga de forma ilimitada, tal y como muestra la figura 5.25.

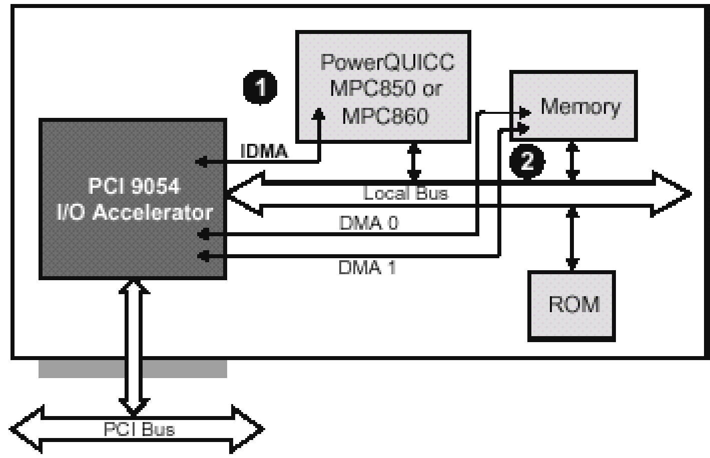


Figura 5.25: Diseño de adaptador de alto rendimiento del MPC850 o MPC860 Power QUICC.

Para operaciones IDMA PowerQUICC, PCI 9054 transfiere datos al Bus PCI bajo el control del protocolo handshake IDMA.
Al mismo tiempo, la tecnología DMA de arquitectura de tubería de datos, puede operar de forma bidireccional con el PCI 9054 como maestro en ambos buses, para administrar transferencias de datos desde el Bus Local al Bus PCI o desde el Bus PCI al Bus Local.

[1] Tecnología avanzada de arquitectura de tubería de datos.

5.5 REGISTROS DE CONFIGURACIÓN

Estos registros sirven para configurar todos los parámetros programables de la PCI 9054 y sus transferencias, todos llevaran un valor por defecto que se cargará desde una memoria EPROM al inicializar la tarjeta. Estos valores se pueden cambiar con facilidad mediante las funciones de la clase MiTarjeta adjunta.
Estos registros se agrupan según sus funciones y son los siguientes:

5.5.2 REGISTROS DE MAPEO DE DIRECCIONES

5.5.2.1 Configuración PCI

PCI Configuration Register Address	Local Access (Offset from Chip Select Address)	To ensure software compatibility with other versions of the PCI 9054 family and to ensure compatibility with future enhancements, write 0 to all unused bits. 31 30 24	23 16	15 8	7 0	PCI Writable	Serial EEPROM Writable
00h	00h	Device ID		Vendor ID		N	Y
04h	04h	Status		Command		Y	N
08h	08h	Class Code			Revision ID	N	Y [31:8]
0Ch	0Ch	BIST	Header Type	PCI Bus Latency Timer	Cache Line Size	Y [7:0]	N
10h	10h	PCI Base Address 0; used for Memory-Mapped Configuration Registers (PCIBAR0)				Y	N
14h	14h	PCI Base Address 1; used for I/O-Mapped Configuration Registers (PCIBAR1)				Y	N
18h	18h	PCI Base Address 2; used for Local Address Space 0 (PCIBAR2)				Y	N
1Ch	1Ch	PCI Base Address 3; used for Local Address Space 1 (PCIBAR3)				Y	N
20h	20h	PCI Base Address 4; used for Local Address Space 2 (PCIBAR4)				Y	N
24h	24h	PCI Base Address 5; used for Local Address Space 3 (PCIBAR5)				Y	N
28h	28h	Cardbus CIS Pointer *(Not supported)*				N	N
2Ch	2Ch	Subsystem ID		Subsystem Vendor ID		N	Y
30h	30h	PCI Base Address for Local Expansion ROM				Y	N
34h	34h	*Reserved*			Next_Cap Pointer	Y [7:0]	N
38h	38h	*Reserved*				N	N
3Ch	3Ch	Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	Y [7:0]	Y [15:8]
40h	180h	Power Management Capabilities		Next_Cap Pointer	Capability ID	Y [31:8]	N
44h	184h	Data	PMCSR Bridge Support Extensions	Power Management Control/Status Register		Y [15, 12:8, 1:0]	N
48h	188h	*Reserved*	Control/Status Register	Next_Cap Pointer	Capability ID	Y [23:16], Local [15:0]	Y [15:0]
4Ch	18Ch	F VPD Address		Next_Cap Pointer	Capability ID	Y [31:16], Local [15:8]	N
50h	190h	VPD Data				Y	N

Register 11-1. (PCIIDR; PCI:00h, LOC:00h) PCI Configuration ID

Bit	Description	Read	Write	Value after Reset
15:0	Vendor ID. Identifies manufacturer of device. Defaults to the PCI SIG-issued Vendor ID of PLX (10B5h) if blank or if no serial EEPROM is present.	Yes	Local/ Serial EEPROM	10B5h or 0
31:16	Device ID. Identifies particular device. Defaults to PLX part number for PCI interface chip (9054h) if blank or noserial EEPROM is present.	Yes	Local/ Serial EEPROM	9054h or 0

Register 11-2. (PCICR; PCI:04h, LOC:04h) PCI Command

Bit	Description	Read	Write	Value after Reset
0	I/O Space. Writing a 1 allows the deviceto respond to I/O space accesses. Writing a 0 disables the device from responding to I/O space accesses.	Yes	Yes	0
1	Memory Space. Writing a 1 allows the device to respond to Memory Space accesses. Writing a 0 disables the device from responding to Memory Space accesses.	Yes	Yes	0
2	Master Enable. Writing a 1 allows device to behave as a Bus Master. Writing a 0 disables device from generating Bus Master accesses.	Yes	Yes	0
3	Special Cycle. *Not supported.*	Yes	No	0
4	Memory Write and Invalidate Enable. Writing a 1 enables the Memory Write and Invalidate mode for PCI Initiator and DMA. (Refer to the DMA Mode register(s), DMAMODE0[13] and/or DMAMODE1[13].)	Yes	Yes	0
5	VGA Palette Snoop. *Not supported.*	Yes	No	0
6	Parity Error Response. Writing a 0 indicates parity error is ignored and the operation continues. Writing a 1 indicates parity checking is enabled.	Yes	Yes	0
7	Wait Cycle Control. Controls whether a device does address/data stepping. Writing a 0 indicates the device never does stepping. Writing a 1 indicates the device always does stepping. *Note: Hardcoded to 0.*	Yes	No	0
8	SERR# Enable. Writing a 1 enables SERR# driver. Writing a 0 disables SERR# driver.	Yes	Yes	0
9	Fast Back-to-Back Enable. Indicates what type of fast back-to-back transfers a Master can perform on the bus. Writing a 1 indicates fast back-to-back transfers can occur to any agent on the bus. Writing a 0 indicates fast back-to-back transfers can only occur to the same agent as in the previous cycle. *Note: Hardcoded to 0.*	Yes	No	0
15:10	*Reserved.*	Yes	No	0h

Register 11-3. (PCISR; PCI:06h, LOC:06h) PCI Status

Bit	Description	Read	Write	Value after Reset
3:0	*Reserved.*	Yes	No	0h
4	New Capability Functions Support. Writing a 1 supports New Capabilities Functions. If enabled, the first New Capability Function ID is located at PCI Configuration offset [40h]. Can be written only from the Local Bus. Read-only from the PCI Bus.	Yes	Local	1
5	*Reserved.*	Yes	No	0
6	User Definable Functions. If set to 1, this device supports user definable functions. Can be written only from the Local Bus. Read-only from the PCI Bus.	Yes	Local	0
7	Fast Back-to-Back Capable. Writing a 1 indicates an adapter can accept fast back-to-back transactions. *Note: Hardcoded to 1.*	Yes	No	1
8	Master Data Parity Error Detected. Set to 1 when three conditions are met: 1) PCI 9054 asserted PERR# or acknowledged PERR# asserted; 2) PCI 9054 was Bus Master for operation in which error occurred; 3) Parity Error Response bit is set (PCICR[6]=1). Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0
10:9	DEVSEL# Timing. Indicates timing for DEVSEL# assertion. Writing a 01 sets this bit to medium. *Note: Hardcoded to 01.*	Yes	No	01
11	Target Abort. When set to 1, indicates the PCI 9054 has signaled a Target abort. Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0
12	Received Target Abort. When set to 1, indicates the PCI 9054 has received a Target Abort signal. Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0
13	Received Master Abort. When set to 1, indicates the PCI 9054 has received a Master Abort signal. Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0
14	Signaled System Error. When set to 1, indicates the PCI 9054 has reported a system error on SERR#. Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0

15	Detected Parity Error. When set to 1, indicates the PCI 9054 has detected a PCI Bus parity error, even if parity error handling is disabled (the Parity Error Response bit in the Command register is clear). One of three conditions can cause this bit to be set: 1) PCI 9054 detected parity error during PCI Address phase; 2) PCI 9054 detected data parity error when it was the Target of a write; 3) PCI 9054 detected data parity error when performing Master Read operation. Writing a 1 clears this bit to 0.	Yes	Yes/Clr	0

Register 11-4. (PCIREV; PCI:08h, LOC:08h) PCI Revision ID

Bit	Description	Read	Write	Value after Reset
7:0	Revision ID. Silicon revision of the PCI 9054.	Yes	Local/ Serial EEPROM	Current Rev #

Register 11-5. (PCICCR; PCI:09-0Bh, LOC:09-0Bh) PCI Class Code

Bit	Description	Read	Write	Value after Reset
7:0	Register Level Programming Interface. None defined.	Yes	Local/ Serial EEPROM	0h
15:8	Subclass Code (Other Bridge Device).	Yes	Local/ Serial EEPROM	80h
23:16	Base Class Code (Bridge Device).	Yes	Local/ Serial EEPROM	06h

Register 11-6. (PCICLSR; PCI:0Ch, LOC:0Ch) PCI Cache Line Size

Bit	Description	Read	Write	Value after Reset
7:0	System Cache Line Size. Specified in units of 32-bit words (8 or 16 Lwords). If a size other than 8 or 16 is specified, the PCI 9054 performs Write transfers rather than Memory Write and Invalidate transfers.	Yes	Yes	0h

Register 11-7. (PCILTR; PCI:0Dh, LOC:0Dh) PCI Bus Latency Timer

Bit	Description	Read	Write	Value after Reset
7:0	PCI Bus Latency Timer. Specifies amount of time (in units of PCI Bus clocks) the PCI 9054, as a Bus Master, can burst data on the PCI Bus.	Yes	Yes	0h

Register 11-8. (PCIHTR; PCI:0Eh, LOC:0Eh) PCI Header Type

Bit	Description	Read	Write	Value after Reset
6:0	Configuration Layout Type. Specifies layout of bits 10h through 3Fh in configuration space. Only one encoding, 0h, is defined. All other encodings are reserved.	Yes	Local	0h
7	Header Type. Writing a 1 indicates multiple functions. Writing a 0 indicates single function.	Yes	Local	0

Register 11-9. (PCIBISTR; PCI:0Fh, LOC:0Fh) PCI Built-In Self Test (BIST)

Bit	Description	Read	Write	Value after Reset
3:0	BIST Pass/Fail. Writing 0h indicates a device passed its test. Non-0h values indicate a device failed its test. Device-specific failure codes can be encoded in a non-0h value.	Yes	Local	0h
5:4	*Reserved.*	Yes	No	00
6	PCI BIST Interrupt Enable. The PCI Bus writes 1 to enable BIST. Generates an interrupt to the Local Bus. The Local Bus resets this bit when BIST is complete. The software should fail device if BIST is not complete after two seconds. Refer to the Runtime registers for Interrupt Control/Status.	Yes	Yes	0
7	BIST Support. Returns 1 if device supports BIST. Returns 0 if device is not BIST-compatible.	Yes	Local	0

Register 11-10. (PCIBAR0; PCI:10h, LOC:10h) PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA

Bit	Description	Read	Write	Value after Reset
0	Memory Space Indicator. Writing a 0 indicates the register maps into Memory space. Writing a 1 indicates the register maps into I/O space. *Note: Hardcoded to 0.*	Yes	No	0
2:1	Register Location. Values: 00—Locate anywhere in 32-bit Memory Address space 01—Locate below 1-MB Memory Address space 10—Locate anywhere in 64-bit Memory Address space 11—Reserved *Note: Hardcoded to 00.*	Yes	No	00
3	Prefetchable. Writing a 1 indicates there are no side effects on reads. Does not affect operation of the PCI 9054. *Note: Hardcoded to 0.*	Yes	No	0
7:4	Memory Base Address. Memory base address for access to Local, Runtime, and DMA registers (requires 256 bytes). *Note: Hardcoded to 0h.*	Yes	No	0h
31:8	Memory Base Address. Memory base address for access to Local, Runtime, and DMA registers.	Yes	Yes	0h

Note: For I_2O, Inbound message frame pool must reside in address space pointed to by PCIBAR0. Message Frame Address (MFA) is defined by I_2O as offset from this base address to start of message frame.

Register 11-11. (PCIBAR1; PCI:14h, LOC:14h) PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA

Bit	Description	Read	Write	Value after Reset
0	**Memory Space Indicator.** Writing a 0 indicates the register maps into Memory space. Writing a 1 indicates the register maps into I/O space. *Note: Hardcoded to 1.*	Yes	No	1
1	*Reserved.*	Yes	No	0
7:2	**I/O Base Address.** Base Address for I/O access to Local, Runtime, and DMA registers (requires 256 bytes). *Note: Hardcoded to 0h.*	Yes	No	0h
31:8	**I/O Base Address.** Base Address for I/O access to Local, Runtime, and DMA registers. PCIBAR1 can be enabled or disabled by setting or clearing the Base Address Register 1 Enable bit (LMISC[0]).	Yes	Yes	0h

Register 11-12. (PCIBAR2; PCI:18h, LOC:18h) PCI Base Address Register for Memory Accesses to Local Address Space 0

Bit	Description	Read	Write	Value after Reset
0	**Memory Space Indicator.** Writing a 0 indicates the register maps into Memory space. Writing a 1 indicates the register maps into I/O space. (Specified in LAS0RR register.)	Yes	No	0
2:1	**Register Location (If Memory Space).** Values: 00—Locate anywhere in 32-bit Memory Address space 01—Locate below 1-MB Memory Address space 10—Locate anywhere in 64-bit Memory Address space 11—*Reserved* (Specified in LAS0RR register.) If I/O Space, bit 1 is always 0 and bit 2 is included in the base address.	Yes	Mem: No I/O: bit 1 no, bit 2 yes	00
3	**Prefetchable (If Memory Space).** Writing a 1 indicates there are no side effects on reads. Reflects value of LAS0RR[3] and provides only status to the system. Does not affect operation of the PCI 9054. The associated Bus Region Descriptor register controls prefetching functions of this address space. (Specified in LAS0RR register.) If I/O Space, bit 3 is included in the base address.	Yes	Mem: No I/O: Yes	0
31:4	**Memory Base Address.** Memory base address for access to Local Address Space 0. PCIBAR2 can be enabled or disabled by setting or clearing the Space 0 Enable bit (LAS0BA[0]).	Yes	Yes	0h

Register 11-13. (PCIBAR3; PCI:1Ch, LOC:1Ch) PCI Base Address Register for Memory Accesses to Local Address Space 1

Bit	Description	Read	Write	Value after Reset
0	**Memory Space Indicator.** Writing a 0 indicates the register maps into Memory space. Writing a 1 indicates the register maps into I/O space. (Specified in LAS1RR register.)	Yes	No	0
2:1	**Register Location.** Values: 00—Locate anywhere in 32-bit Memory Address space 01—Locate below 1-MB Memory Address space 10—Locate anywhere in 64-bit Memory Address space 11—*Reserved* (Specified in LAS1RR register.) If I/O Space, bit 1 is always 0 and bit 2 is included in the base address.	Yes	Mem: No I/O: Bit 1 No, Bit 2 Yes	00
3	**Prefetchable (If Memory Space).** Writing a 1 indicates there are no side effects on reads. Reflects value of LAS1RR[3] and only provides status to the system. Does not affect operation of the PCI 9054. The associated Bus Region Descriptor register controls prefetching functions of this address space. (Specified in LAS1RR register.) If I/O Space, bit 3 is included in base address.	Yes	Mem: No I/O: Yes	0
31:4	**Memory Base Address.** Memory base address for access to Local Address Space 1. PCIBAR3 can be enabled or disabled by setting or clearing the Space 1 Enable bit (LAS1BA[0]). If QSR[0]=1, PCIBAR3 returns 0h.	Yes	Yes	0h

Register 11-14. (PCIBAR4; PCI:20h, LOC:20h) PCI Base Address

Bit	Description	Read	Write	Value after Reset
31:0	*Reserved.*	Yes	No	0h

Register 11-15. (PCIBAR5; PCI:24h, LOC:24h) PCI Base Address

Bit	Description	Read	Write	Value after Reset
31:0	*Reserved.*	Yes	No	0h

Register 11-16. (PCICIS; PCI:28h, LOC:28h) PCI Cardbus CIS Pointer

Bit	Description	Read	Write	Value after Reset
31:0	Cardbus Information Structure Pointer for PCMCIA. *Not supported.*	Yes	No	0h

Register 11-17. (PCISVID; PCI:2Ch, LOC:2Ch) PCI Subsystem Vendor ID

Bit	Description	Read	Write	Value after Reset
15:0	Subsystem Vendor ID (unique add-in board Vendor ID).	Yes	Local/ Serial EEPROM	10B5h

Register 11-18. (PCISID; PCI:2Eh, LOC:2Eh) PCI Subsystem ID

Bit	Description	Read	Write	Value after Reset
15:0	Subsystem ID (unique add-in board Device ID).	Yes	Local/ Serial EEPROM	9054h

Register 11-19. (PCIERBAR; PCI:30h, LOC:30h) PCI Expansion ROM Base

Bit	Description	Read	Write	Value after Reset
0	Address Decode Enable. Writing a 1 indicates a device accepts accesses to the Expansion ROM address. Writing a 0 indicates a device does not accept accesses to Expansion ROM space. Should be set to 0 if there is no Expansion ROM. Works in conjunction with EROMRR[0].	Yes	Yes	0
10:1	*Reserved.*	Yes	No	0h
31:11	Expansion ROM Base Address (upper 21 bits).	Yes	Yes	0h

Register 11-20. (CAP_PTR; PCI:34h, LOC:34h) New Capability Pointer

Bit	Description	Read	Write	Value after Reset
7:0	New Capability Pointer. Offset into PCI Configuration Space for the location of the first item in the New Capabilities Linked List.	Yes	Local	40h
31:8	*Reserved.*	Yes	No	0h

Register 11-21. (PCIILR; PCI:3Ch, LOC:3Ch) PCI Interrupt Line

Bit	Description	Read	Write	Value after Reset
7:0	Interrupt Line Routing Value. Value indicates which input of the system interrupt controller(s) is connected to each interrupt line of the device.	Yes	Yes	0h

Register 11-22. (PCIIPR; PCI:3Dh, LOC:3Dh) PCI Interrupt Pin

Bit	Description	Read	Write	Value after Reset
7:0	Interrupt Pin Register. Indicates which interrupt pin the device uses. The following values are decoded (the PCI 9054 supports only INTA#): 0 = No Interrupt pin 1 = INTA# 2 = INTB# 3 = INTC# 4 = INTD#	Yes	Local/ Serial EEPROM	1h

Register 11-23. (PCIMGR; PCI:3Eh, LOC:3Eh) PCI Min_Gnt

Bit	Description	Read	Write	Value after Reset
7:0	Min_Gnt. Specifies how long a burst period device needs, assuming a clock rate of 33 MHz. Value is a multiple of 1/4 µs increments.	Yes	Local/ Serial EEPROM	0h

Register 11-24. (PCIMLR; PCI:3Fh, LOC:3Fh) PCI Max_Lat

Bit	Description	Read	Write	Value after Reset
7:0	Max_Lat. Specifies how often the device must gain access to the PCI Bus. Value is a multiple of 1/4 µs increments.	Yes	Local/ Serial EEPROM	0h

Register 11-25. (PMCAPID; PCI:40h, LOC:180h) Power Management Capability ID

Bit	Description	Read	Write	Value after Reset
7:0	Power Management Capability ID.	Yes	No	1h

Register 11-26. (PMNEXT; PCI:41h, LOC:181h) Power Management Next Capability Pointer

Bit	Description	Read	Write	Value after Reset
7:0	Next_Cap Pointer. Points to the first location of the next item in the capabilities linked list. If power management is the last item in the list, then this register should be set to 0.	Yes	Local	48h

Register 11-27. (PMC; PCI:42h, LOC:182h) Power Management Capabilities

Bit	Description	Read	Write	Value after Reset
2:0	Version. Writing a 1 indicates this function complies with *PCI Power Management Interface Specification v1.0*.	Yes	Local	001
3	PCI Clock Required for PME# Signal. When set to 1, indicates a function relies on the presence of the PCI clock for PME# operation. The PCI 9054 does not require the PCI clock for PME#, so this bit should be set to 0.	Yes	Local	0
4	Auxiliary Power Source. Because the PCI 9054 does not support PME# while in a $D3_{cold}$ state, this bit is always set to 0.	Yes	No	0
5	DSI. When set to 1, the PCI 9054 requires special initialization following a transition to a D_0 uninitialized state before a generic class device driver is able to use it.	Yes	Local	0
8:6	*Reserved.*	Yes	No	000
9	D_1_Support. When set to 1, the PCI 9054 supports the D_1 power state.	Yes	Local	0
10	D_2_Support. When set to 1, the PCI 9054 supports the D_2 power state.	Yes	Local	0
14:11	PME#_Support. Indicates power states in which the PCI 9054 may assert PME#. Value — Description XXX1 — PME# can be asserted from D_0 XX1X — PME# can be asserted from D_1 X1XX — PME# can be asserted from D_2 1XXX — PME# can be asserted from D_{3hot}	Yes	Local	0h
15	*Reserved.*	Yes	No	0

Register 11-28. (PMCSR; PCI:44h, LOC:184h) Power Management Control/Status

Bit	Description	Read	Write	Value after Reset
1:0	Power State. Determines or changes the current power state. Value — State 00 — D_0 01 — D_1 10 — D_2 11 — D_{3hot} Transition from a D_{3hot} state to a D_0 state causes a soft reset. Should only be initiated from the PCI Bus because the Local Bus interface is reset during a soft reset. In a D_{3hot} state, PCI Memory and I/O accesses are disabled, as well as PCI interrupts, and only configuration is allowed. The same is true for the D_2 state if the corresponding D_2_Support pin is set.	Yes	Yes	00
7:2	*Reserved.*	Yes	No	0h
8	PME#_En. Writing a 1 enables PME# to be asserted.	Yes	Yes	0
12:9	Data_Select. Selects which data to report through the Data register and Data_Scale bits.	Yes	Yes	0h
14:13	Data_Scale. Indicates the scaling factor to use when interpreting the value of the Data register. Value and meaning of this bit depends on the data value selected by the Data_Select bit. When the Local CPU initializes the Data_Scale values, must use the Data_Select bit to determine which Data_Scale value it is writing. For Power Consumed and Power Dissipated data, the following scale factors are used. Unit values are in watts. Value — Scale 0 — Unknown 1 — 0.1x 2 — 0.01x 3 — 0.001x	Yes	Local	00
15	PME#_Status. Indicates PME# is being driven if the PME#_En bit is set (PMCSR[8]=1). Writing a 1 from the Local Bus sets this bit; writing a 1 from the PCI Bus clears this bit to 0. Depending on the current power state, set only if the appropriate PME#_Support bit(s) is set (PMC[15:11]=1).	Yes	Local/ Set, PCI/Clr	0

Register 11-29. (PMCSR_BSE; PCI:46h, LOC:186h) PMCSR Bridge Support Extensions

Bit	Description	Read	Write	Value after Reset
7:0	*Reserved.*	Yes	No	0h

Register 11-30. (PMDATA; PCI:47h, LOC:187h) Power Management Data

Bit	Description	Read	Write	Value after Reset
7:0	Power Management Data. Provides operating data, such as power consumed or heat dissipation. Data returned is selected by the Data_Select bit(s) (PMCSR[12:9]) and scaled by the Data_Scale bit(s) (PMCSR[14:13]). Data_Select — Description 0 — D_0 Power Consumed 1 — D_1 Power Consumed 2 — D_2 Power Consumed 3 — D_3 Power Consumed 4 — D_0 Power Dissipated 5 — D_1 Power Dissipated 6 — D_2 Power Dissipated 7 — D_{3hot} Power Dissipated	Yes	Local	0h

Register 11-31. (HS_CNTL; PCI:48h, LOC:188h) Hot Swap Control

Bit	Description	Read	Write	Value after Reset
7:0	Hot Swap ID.	Yes	Local/ Serial EEPROM	06h

Register 11-32. (HS_NEXT; PCI:49h, LOC:189h) Hot Swap Next Capability Pointer

Bit	Description	Read	Write	Value after Reset
7:0	Next_Cap Pointer. Points to the first location of the next item in the capabilities linked list. If Hot Swap is the last item in the list, then this register should be set to 0.	Yes	Local/ Serial EEPROM	4Ch

Register 11-33. (HS_CSR; PCI:4Ah, LOC:18Ah) Hot Swap Control/Status

Bit	Description	Read	Write	Value after Reset
0	*Reserved.*	Yes	No	0
1	ENUM# Interrupt Clear. Writing a 0 enables the interrupt. Writing a 1 clears the interrupt.	Yes	Yes/Clr	0
2	*Reserved.*	Yes	No	0
3	LED Software On/Off Switch. Writing a 1 turns on the LED. Writing a 0 turns off the LED.	Yes	PCI	0
4	*Reserved.*	Yes	No	0
5	*Reserved.*	Yes	No	0
6	Board Removal ENUM# Status Indicator. Writing a 1 reports the ENUM# assertion for removal process.	Yes	Yes	0
7	Board Insertion ENUM# Status Indicator. Writing a 1 reports the ENUM# assertion for insertion process.	Yes	Yes	1
15:8	*Reserved.*	Yes	No	0h

Register 11-34. (PVPDCNTL; PCI:4Ch, LOC:18Ch) PCI Vital Product Data Control

Bit	Description	Read	Write	Value after Reset
7:0	VPD ID. Capability ID = 03h for VPD.	PCI	No	03h

Register 11-35. (PVPD_NEXT; PCI:4Dh, LOC:18Dh) PCI Vital Product Data Next Capability Pointer

Bit	Description	Read	Write	Value after Reset
7:0	Next_Cap Pointer. Points to first location of next item in the capabilities linked list. VPD is the last item in the capabilities linked list. This register is set to 0h.	PCI	Local	0h

Register 11-36. (PVPDAD; PCI:4Eh, LOC:18Eh) PCI Vital Product Data Address

Bit	Description	Read	Write	Value after Reset
14:0	VPD Address. Byte address of the VPD address to be accessed. Supports a 2K or 4K bit serial EEPROM.	PCI	Yes	0h
15	F. Flag used to indicate when the transfer of data between PVPDATA and the storage component is complete. Writing a 0 along with the VPD address causes a read of VPD information into PVPDATA. The hardware sets this bit to 1 when the VPD Data transfer is complete. Writing a 1 along with the VPD address causes a write of VPD information from PVPDATA into a storage component. The hardware sets this bit to 0 after the Write operation is complete.	PCI	Yes	0

Register 11-37. (PVPDATA; PCI:50h, LOC:190h) PCI VPD Data

Bit	Description	Read	Write	Value after Reset
31:0	VPD Data Register.	PCI	Yes	0h

5.5.2.2 Configuración local

<table>
<tr><th>PCI (Offset from Base Address)</th><th>Local Access (Offset from Chip Select Address</th><th colspan="4">To ensure software compatibility with other versions of the PCI 9054 family and to ensure compatibility with future enhancements, write 0 to all unused bits.
31 24 | 23 16 | 15 8 | 7 0</th><th>PCI/Local Writable</th><th>Serial EEPROM Writable</th></tr>
<tr><td>00h</td><td>80h</td><td colspan="4">Range for PCI-to-Local Address Space 0</td><td>Y</td><td>Y</td></tr>
<tr><td>04h</td><td>84h</td><td colspan="4">Local Base Address (Remap) for PCI-to-Local Address Space 0</td><td>Y</td><td>Y</td></tr>
<tr><td>08h</td><td>88h</td><td colspan="4">Mode/DMA Arbitration</td><td>Y</td><td>Y</td></tr>
<tr><td>0Ch</td><td>8Ch</td><td>Reserved</td><td>Serial EEPROM Write-Protected Address Boundary</td><td>Local Miscellaneous Control</td><td>Big/Little Endian Descriptor</td><td>Y</td><td>Y</td></tr>
<tr><td>10h</td><td>90h</td><td colspan="4">Range for PCI-to-Local Expansion ROM</td><td>Y</td><td>Y</td></tr>
<tr><td>14h</td><td>94h</td><td colspan="4">Local Base Address (Remap) for PCI-to-Local Expansion ROM and BREQo Control</td><td>Y</td><td>Y</td></tr>
<tr><td>18h</td><td>98h</td><td colspan="4">Local Bus Region Descriptors (Space 0 and Expansion ROM) for PCI-to-Local Accesses</td><td>Y</td><td>Y</td></tr>
<tr><td>1Ch</td><td>9Ch</td><td colspan="4">Range for PCI Initiator-to-PCI</td><td>Y</td><td>Y</td></tr>
<tr><td>20h</td><td>A0h</td><td colspan="4">Local Base Address for PCI Initiator-to-PCI Memory</td><td>Y</td><td>Y</td></tr>
<tr><td>24h</td><td>A4h</td><td colspan="4">Local Base Address for PCI Initiator-to-PCI I/O Configuration</td><td>Y</td><td>Y</td></tr>
<tr><td>28h</td><td>A8h</td><td colspan="4">PCI Base Address (Remap) for PCI Initiator-to-PCI</td><td>Y</td><td>Y</td></tr>
<tr><td>2Ch</td><td>ACh</td><td colspan="4">PCI Configuration Address Register for PCI Initiator-to-PCI I/O Configuration</td><td>Y</td><td>Y</td></tr>
<tr><td>F0h</td><td>170h</td><td colspan="4">Range for PCI-to-Local Address Space 1</td><td>Y</td><td>Y</td></tr>
<tr><td>F4h</td><td>174h</td><td colspan="4">Local Base Address (Remap) for PCI-to-Local Address Space 1</td><td>Y</td><td>Y</td></tr>
<tr><td>F8h</td><td>178h</td><td colspan="4">Local Bus Region Descriptor (Space 1) for PCI-to-Local Accesses</td><td>Y</td><td>Y</td></tr>
<tr><td>FCh</td><td>17Ch</td><td colspan="4">PCI Base Dual Address Cycle (Remap) for PCI Initiator-to-PCI (Upper 32 bits)</td><td>Y</td><td>N</td></tr>
</table>

Register 11-38. (LAS0RR; PCI:00h, LOC:80h) Local Address Space 0 Range Register for PCI-to-Local Bus

Bit	Description	Read	Write	Value after Reset
0	Memory Space Indicator. Writing a 0 indicates Local Address Space 0 maps into PCI Memory space. Writing a 1 indicates Local Address Space 0 maps into PCI I/O space.	Yes	Yes	0
2:1	When mapped into Memory space, encoding is as follows: 2/1 Meaning 0 0 Locate anywhere in 32-bit PCI Address space 0 1 Locate below 1 MB in PCI Address space 1 0 Locate anywhere in 64-bit PCI Address space 1 1 *Reserved* When mapped into I/O space, bit 1 must be set to 0.Bit 2 is included with bits [31:3] to indicate the decoding range.	Yes	Yes	00
3	When mapped into Memory space, writing a 1 indicates reads are prefetchable (does not affect operation of the PCI 9054, but is used for system status). When mapped into I/O space, it is included with bits [31:2] to indicate the decoding range.	Yes	Yes	0
31:4	Specifies which PCI Address bits to use for decoding a PCI access to Local Bus Space 0. Each bit corresponds to a PCI Address bit. Bit 31 corresponds to address bit 31. Write 1 to all bits that must be included in decode and 0 to all others (used in conjunction with PCIBAR2). Default is 1 MB. *Notes: Range (not Range register) must be power of 2. "Range register value" is inverse of range.* *User should limit all I/O spaces to 256 bytes per PCI v2.1 spec.*	Yes	Yes	FFF0000h

Register 11-39. (LAS0BA; PCI:04h, LOC:84h) Local Address Space 0 Local Base Address (Remap)

Bit	Description	Read	Write	Value after Reset
0	Space 0 Enable. Writing a 1 enables decoding of PCI addresses for PCI Target access to Local Bus Space 0. Writing a 0 disables decoding.	Yes	Yes	0
1	*Reserved.*	Yes	No	0
3:2	If Local Bus Space 0 is mapped into Memory space, bits are not used. When mapped into I/O space, included with bits [31:4] for remapping.	Yes	Yes	00
31:4	Remap PCI Address to Local Address Space 0 into Local Address Space. Bits in this register remap (replace) PCI Address bits used in decode as Local Address bits. *Note: Remap Address value must be a multiple of the Range (not the Range register).*	Yes	Yes	0h

Register 11-40. (MARBR; PCI:08h or ACh, LOC:88h or 12Ch) Mode/DMA Arbitration

Bit	Description	Read	Write	Value after Reset
7:0	Local Bus Latency Timer. Number of Local Bus clock cycles to occur before de-asserting HOLD and releasing the Local Bus.	Yes	Yes	0h
15:8	Local Bus Pause Timer. Number of Local Bus Clock cycles to occur before reasserting HOLD after releasing the Local Bus. The pause timer is valid only during DMA.	Yes	Yes	0h
16	Local Bus Latency Timer Enable. Writing a 1 enables the latency timer. Writing a 0 disables the latency timer.	Yes	Yes	0
17	Local Bus Pause Timer Enable. Writing a 1 enables the pause timer. Writing a 0 disables the pause timer.	Yes	Yes	0
18	Local Bus BREQ Enable. Writing a 1 enables the Local Bus BR#/BREQi. When BR#/BREQi is active, the PCI 9054 de-asserts HOLD and releases the Local Bus.	Yes	Yes	0
20:19	DMA Channel Priority. Writing a 00 indicates a rotational priority scheme. Writing a 01 indicates Channel 0 has priority. Writing a 10 indicates Channel 1 has priority. Writing an 11 indicates *reserved*.	Yes	Yes	00
21	Local Bus PCI Target Release Bus Mode. When set to 1, the PCI 9054 de-asserts HOLD and releases the Local Bus when the PCI Target Write FIFO becomes empty during a PCI Target Write or when the PCI Target Read FIFO becomes full during a PCI Target Read.	Yes	Yes	1
22	PCI Target LOCK# Enable. Writing a 1 enables PCI Target locked sequences. Writing a 0 disables PCI Target locked sequences.	Yes	Yes	0
23	PCI Request Mode. Writing a 1 causes the PCI 9054 to de-assert REQ# when it asserts FRAME during a Master cycle. Writing a 0 causes the PCI 9054 to leave REQ# asserted for the entire Bus Master cycle.	Yes	Yes	0
24	Delayed Read Mode. When set to 1, the PCI 9054 operates in Delayed Transaction mode for PCI Target reads. The PCI 9054 issues a Retry to the PCI Host and prefetches Read data.	Yes	Yes	0
25	PCI Read No Write Mode. Writing a 1 forces a Retry on writes if a read is pending. Writing a 0 allows writes to occur while a read is pending.	Yes	Yes	0
26	PCI Read with Write Flush Mode. Writing a 1 submits a request to flush a pending Read cycle if a Write cycle is detected. Writing a 0 submits a request to not effect pending reads when a Write cycle occurs (*PCI Specification v2.1* compatible).	Yes	Yes	0
27	Gate Local Bus Latency Timer with BREQi. (C and J modes only.)	Yes	Yes	0
28	PCI Read No Flush Mode. Writing a 1 submits a request to not flush the Read FIFO if the PCI Read cycle completes (Read Ahead mode). Writing a 0 submits a request to flush the Read FIFO if a PCI Read cycle completes.	Yes	Yes	0
29	When set to 0, reads from the PCI Configuration Register address 00h returns Device ID and Vendor ID. When set to 1, reads from the PCI Configuration register address 00h returns Subsystem ID and Subsystem Vendor ID.	Yes	Yes	0
30	FIFO Full Status Flag. When set to 1, the PCI Initiator Write FIFO is almost full. Reflects the value of the DMPAF pin.	Yes	No	0
31	BIGEND#/WAIT# Input/Output Select (M mode only). Writing a 1 selects the wait functionality of the signal. Writing a 0 selects Big Endian input functionality.	Yes	Yes	0

Register 11-41. (BIGEND; PCI:0Ch, LOC:8Ch) Big/Little Endian Descriptor

Bit	Description	Read	Write	Value after Reset
0	**Configuration Register Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for Local accesses to the Configuration registers. Writing a 0 specifies Little Endian ordering. Big Endian mode can be specified for Configuration register accesses by asserting BIGEND# during the Address phase of the access.	Yes	Yes	0
1	**PCI Initiator Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for PCI Initiator accesses. Writing a 0 specifies Little Endian ordering. Big Endian mode can be specified for PCI Initiator accesses by asserting BIGEND# input pin during the Address phase of the access.	Yes	Yes	0
2	**PCI Target Address Space 0 Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for PCI Target accesses to Local Address Space 0. Writing a 0 specifies Little Endian ordering.	Yes	Yes	0
3	**PCI Target Address Expansion ROM 0 Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for PCI Target accesses to Expansion ROM. Writing a 0 specifies Little Endian ordering.	Yes	Yes	0
4	**Big Endian Byte Lane Mode.** Writing a 1 specifies that in any Endian mode, use the following byte lanes for the modes listed: **M Mode** [0:15] for a 16-bit Local Bus [0:7] for an 8-bit Local Bus **C and J Modes** [31:16] for a 16-bit Local Bus [31:24] for an 8-bit Local Bus Writing a 0 specifies that in any Endian mode, use the following byte lanes for the modes listed: **M Mode** [16:31] for a 16-bit Local Bus [24:31] for an 8-bit Local Bus **C and J Modes** [15:0] for a 16-bit Local Bus [7:0] for an 8-bit Local Bus	Yes	Yes	0
5	**PCI Target Address Space 1 Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for PCI Target accesses to Local Address Space 1. Writing a 0 specifies Little Endian ordering.	Yes	Yes	0
6	**DMA Channel 1 Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for DMA Channel 1 accesses to the Local Address space. Writing a 0 specifies Little Endian ordering.	Yes	Yes	0
7	**DMA Channel 0 Big Endian Mode (Address Invariance).** Writing a 1 specifies use of Big Endian data ordering for DMA Channel 0 accesses to the Local Address space. Writing a 0 specifies Little Endian ordering.	Yes	Yes	0

Register 11-42. (LMISC; PCI:0Dh, LOC:8Dh) Local Miscellaneous Control

Bit	Description	Read	Write	Value after Reset
0	Base Address Register 1 Enable. If set to 1, the Base Address 1 Register for I/O accesses to Configuration registers is enabled. If set to 0, the Base Address 1 Register for I/O accesses to Configuration registers is disabled.	Yes	Yes	1
1	Base Address Register 1 Shift. If Base Address Register 1 Enable is low, and this bit is set to 0, then PCIBAR2 and PCIBAR3 remain at PCI Configuration addresses 18h and 1Ch. If Base Address Register 1 Enable is low, and this bit is set to 1, then PCIBAR2 (Local Address Space 0) and PCIBAR3 (Local Address Space 1) are shifted to become PCIBAR1 and PCIBAR2 at PCI Configuration addresses 14h and 18h. Set if a blank region in Base Address Register Space could not be accepted by system BIOS.	Yes	Yes	0
2	Local Init Status. Writing a 1 indicates Local Init done. Responses to PCI accesses are Retrys until this bit is set. If the PCI 9054 has a blank serial EEPROM attached, the Local processor must set the Local Init Status bit to 1.	Yes	Local/ Serial EEPROM	0
3	*Reserved.*	Yes	No	0
4	M Mode PCI Initiator Deferred Read Enable. Writing a 1 enables the PCI 9054 to operate in Delayed Transaction mode for PCI Initiator reads. The PCI 9054 issues a RETRY# to the M mode Master and prefetches Read data from the PCI Bus.	Yes	Yes	0
5	M Mode TEA# Input Interrupt Mask. When set to 1, TEA# input causes SERR# output on the PCI Bus if enabled (PCICR[8]=1) and the Signaled System Error bit is set (PCISR[14]=1). Writing 0 masks the TEA# input to create SERR#. The SERR# Status bit is set in both cases.	Yes	Yes	0
6	PCI Initiator Write FIFO Almost Full RETRY# Output Enable. When set to 1, the PCI 9054 issues a RETRY# to the MPC850 or MPC860.	Yes	Yes	0
7	*Reserved.*	Yes	No	0

Register 11-43. (PROT_AREA; PCI:0Eh, LOC:8Eh) Serial EEPROM Write-Protected Address Boundary

Bit	Description	Read	Write	Value after Reset
6:0	Serial EEPROM Starting at Lword Boundary (48 Lwords = 192 bytes) for VPD Accesses. Any serial EEPROM address below this boundary is read-only. *Note: Anything below the programmed address may contain the PCI 9054 Configuration data.*	Yes	Yes	0110000
15:7	*Reserved.*	Yes	No	0h

Register 11-44. (EROMRR; PCI:10h, LOC:90h) Expansion ROM Range

Bit	Description	Read	Write	Value after Reset
0	Address Decode Enable. Bit 0 can only be enabled from the serial EEPROM. To disable, set the PCI Expansion ROM Address Decode Enable bit to 0 (PCIERBAR[0]=0).	Yes	Serial EEPROM Only	0
10:1	*Reserved.*	Yes	No	0h
31:11	Specifies which PCI Address bits to use for decoding a PCI-to-Local Bus Expansion ROM. Each bit corresponds to a PCI Address bit. Bit 31 corresponds to address bit 31. Write 1 to all bits that must be included in decode and 0 to all others (used in conjunction with PCIERBAR). Default is 64 KB. *Note: Range (not Range register) must be power of 2. "Range register value" is inverse of range.*	Yes	Yes	FFFF00h

Register 11-45. (EROMBA; PCI:14h, LOC:94h) Expansion ROM Local Base Address (Remap) and BREQo Control

Bit	Description	Read	Write	Value after Reset
3:0	M Mode: RETRY# Signal Assertion Delay Clocks. Number of Local Bus clocks in which a PCI Target BR# request is pending and a Local PCI Initiator access is in progress and not being granted the bus BG# before asserting RETRY#. Once asserted, RETRY# remains asserted until PCI 9054 samples de-assertion of BB# by the Local Arbiter (Least Significant Bit is 8 or 64 clocks). C and J Modes: Backoff Request Delay Clocks. Number of Local Bus clocks in which a PCI Target HOLD request is pending and a Local PCI Initiator access is in progress and not being granted the bus (LHOLDA) before asserting BREQo (Backoff Request Out). BREQo remains asserted until the PCI 9054 receives LHOLDA (Least Significant Bit is 8 or 64 clocks).	Yes	Yes	0h
4	Local Bus Backoff Enable (C, J, and M Modes). Writing a 1 enables the PCI 9054 to assert BREQo/RETRY#.	Yes	Yes	0
5	Backoff Timer Resolution. Writing a 1 changes the Least Significant Bit of the Backoff Timer from 8 to 64 clocks.	Yes	Yes	0
10:6	*Reserved.*	Yes	No	0h
31:11	Remap PCI Expansion ROM Space into Local Address Space. Bits in this register remap (replace) the PCI Address bits used in decode as Local Address bits. *Note: Remap Address value must be a multiple of the Range (not the Range register).*	Yes	Yes	0h

Register 11-46. (LBRD0; PCI:18h, LOC:98h) Local Address Space 0/Expansion ROM Bus Region Descriptor

Bit	Description	Read	Write	Value after Reset
1:0	Memory Space 0 Local Bus Width. Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
5:2	Memory Space 0 Internal Wait States (data-to-data; 0-15 wait states).	Yes	Yes	0h
6	Memory Space 0 TA#/READY# Input Enable. Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
7	Memory Space 0 BTERM# Input Enable. Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or Section 4.2.5 for C and J modes.	Yes	Yes	0
8	Memory Space 0 Prefetch Disable. When mapped into Memory space, writing a 0 enables Read prefetching. Writing a 1 disables prefetching. If prefetching is disabled, the PCI 9054 disconnects after each Memory read.	Yes	Yes	0
9	Expansion ROM Space Prefetch Disable. Writing a 0 enables Read prefetching. Writing a 1 disables prefetching. If prefetching is disabled, the PCI 9054 disconnects after each Memory read.	Yes	Yes	0
10	Prefetch Counter Enable. When set to 1 and Memory prefetching is enabled, the PCI 9054 prefetches up to the number of Lwords specified in prefetch count. When set to 0, the PCI 9054 ignores the count and continues prefetching until it is terminated by the PCI Bus.	Yes	Yes	0
14:11	Prefetch Counter. Number of Lwords to prefetch during Memory Read cycles (0-15). A count of zero selects a prefetch of 16 Lwords.	Yes	Yes	0h
15	*Reserved.*	Yes	No	0
17:16	Expansion ROM Space Local Bus Width. Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
21:18	Expansion ROM Space Internal Wait States (data-to-data; 0-15 wait states).	Yes	Yes	0h
22	Expansion ROM Space TA#/READY# Input Enable. Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
23	Expansion ROM Space BTERM# Input Enable. Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or to Section 4.2.5 for C and J modes.	Yes	Yes	0

24	Memory Space 0 Burst Enable. Writing a 1 enables bursting. Writing a 0 disables bursting.	Yes	Yes	0
25	Extra Long Load from Serial EEPROM. Writing a 1 loads the Subsystem ID and Local Address Space 1 registers. Writing a 0 indicates not to load them.	Yes	Serial EEPROM Only	0
26	Expansion ROM Space Burst Enable. Writing a 1 enables bursting. Writing a 0 disables bursting.	Yes	Yes	0
27	PCI Target PCI Write Mode. Writing a 0 indicates the PCI 9054 should disconnect when the PCI Target Write FIFO is full. Writing a 1 indicates the PCI 9054 should de-assert TRDY# when the PCI Target Write FIFO is full.	Yes	Yes	0

Register 11-47. (DMRR; PCI:1Ch, LOC:9Ch) Local Range Register for PCI Initiator-to-PCI

Bit	Description	Read	Write	Value after Reset
15:0	*Reserved* (64-KB increments).	Yes	No	0h
31:16	Specifies which Local Address bits to use for decoding a Local-to-PCI Bus access. Each bit corresponds to a PCI Address bit. Bit 31 corresponds to address bit 31. Write 1 to all bits that must be included in decode and 0h to all others. *Note: Range (not Range register) must be power of 2. "Range register value" is inverse of range.*	Yes	Yes	0h

Register 11-48. (DMLBAM; PCI:20h, LOC:A0h) Local Bus Base Address Register for PCI Initiator-to-PCI Memory

Bit	Description	Read	Write	Value after Reset
15:0	*Reserved.*	Yes	No	0h
31:16	Assigns a value to bits to use for decoding Local-to-PCI Memory accesses. *Note: Local Base Address value must be a multiple of the Range (not the Range register).*	Yes	Yes	0h

Register 11-49. (DMLBAI; PCI:24h, LOC:A4h) Local Base Address Register for PCI Initiator-to-PCI I/O Configuration

Bit	Description	Read	Write	Value after Reset
15:0	*Reserved.*	Yes	No	0h
31:16	Assigns a value to bits to use for decoding Local-to-PCI I/O or Configuration accesses. *Notes: Local Base Address value must be a multiple of the Range (not the Range register).* *Refer to DMPBAM[13] for the I/O Remap Address option.*	Yes	Yes	0h

Register 11-50. (DMPBAM; PCI:28h, LOC:A8h) PCI Base Address (Remap) Register for PCI Initiator-to-PCI Memory

Bit	Description	Read	Write	Value after Reset
0	PCI Initiator Memory Access Enable. Writing a 1 enables decode of PCI Initiator Memory accesses. Writing a 0 disables decode of PCI Initiator Memory accesses.	Yes	Yes	0
1	PCI Initiator I/O Access Enable. Writing a 1 enables decode of PCI Initiator I/O accesses. Writing a 0 disables decode of PCI Initiator I/O accesses.	Yes	Yes	0
2	PCI Initiator Cache Enable. Writing a 1 causes prefetch to occur infinitely.	Yes	Yes	0
12, 3	PCI Initiator Read Prefetch Size Control. Values: 00 = PCI 9054 continues to prefetch Read data from the PCI Bus until the PCI Initiator access is finished. This may result in an additional four unneeded Lwords being prefetched from the PCI Bus. 01 = Prefetch up to four Lwords from the PCI Bus. 10 = Prefetch up to eight Lwords from the PCI Bus. 11 = Prefetch up to 16 Lwords from the PCI Bus. PCI Initiator Burst reads should not exceed programmed limit.	Yes	Yes	00
4	PCI Initiator PCI Read Mode. Writing a 0 indicates the PCI 9054 should release the PCI Bus when the Read FIFO becomes full. Writing a 1 indicates the PCI 9054 should keep the PCI Bus and de-assert IRDY# when the Read FIFO becomes full.	Yes	Yes	0
10, 8:5	Programmable Almost Full Flag. When the number of entries in the 32-word PCI Initiator Write FIFO exceeds this value, the MDREQ#/DMPAF signal is asserted high.	Yes	Yes	00000
9	Memory Write and Invalidate Mode. When set to 1, the PCI 9054 waits for 8 or 16 Lwords to be written from the Local Bus before starting a PCI access. In addition, all Memory Write and Invalidate cycles to the PCI Bus must be 8 or 16 Lword bursts.	Yes	Yes	0
11	PCI Initiator Prefetch Limit. Writing a 1 causes the PCI 9054 to not prefetch past 4-KB boundaries.	Yes	Yes	0
13	I/O Remap Select. Writing a 1 forces PCI Address bits [31:16] to all zeros. Writing a 0 uses bits [31:16] of this register as PCI Address bits [31:16].	Yes	Yes	0
15:14	PCI Initiator Write Delay. Delays PCI Bus request after PCI Initiator Burst Write cycle has started. Values: 00 = No delay; start cycle immediately 01 = Delay 4 PCI clocks 10 = Delay 8 PCI clocks 11 = Delay 16 PCI clocks	Yes	Yes	00
31:16	Remap Local-to-PCI Space into PCI Address Space. Bits in this register remap (replace) Local Address bits used in decode as the PCI Address bits. *Note: Remap Address value must be a multiple of the Range (not the Range register).*	Yes	Yes	0h

Register 11-51. (DMCFGA; PCI:2Ch, LOC:ACh) PCI Configuration Address Register for PCI Initiator-to-PCI I/O Configuration

Bit	Description	Read	Write	Value after Reset
1:0	Configuration Type (00=Type 0, 01=Type 1).	Yes	Yes	00
7:2	Register Number.	Yes	Yes	0
10:8	Function Number.	Yes	Yes	0
15:11	Device Number.	Yes	Yes	0
23:16	Bus Number.	Yes	Yes	0h
30:24	*Reserved.*	Yes	No	0h
31	Configuration Enable. Writing a 1 allows Local-to-PCI I/O accesses to be converted to a PCI Configuration cycle. Parameters in this table are used to assert the PCI Configuration address. *Note: For more information, refer to the PCI Initiator Configuration Cycle example in Section 3.4.1.7 for M mode or Section 5.4.1.6.1 for C and J modes.*	Yes	Yes	0

Register 11-53. (LAS1BA; PCI:F4h, LOC:174h) Local Address Space 1 Local Base Address (Remap)

Bit	Description	Read	Write	Value after Reset
0	Space 1 Enable. Writing a 1 enables decoding of PCI addresses for PCI Target access to Local Bus Space 1. Writing a 0 disables decoding.	Yes	Yes	0
1	*Reserved.*	Yes	No	0
3:2	Not used if Local Bus Space 1 is mapped into Memory space. Included with bits [31:4] for remapping when mapped into I/O space.	Yes	Yes	00
31:4	Remap PCI Address to Local Address Space 1 into Local Address Space. Bits in this register remap (replace) the PCI Address bits used in decode as Local Address bits. *Note: Remap Address value must be a multiple of the Range (not the Range register).*	Yes	Yes	0h

Register 11-54. (LBRD1; PCI:F8h, LOC:178h) Local Address Space 1 Bus Region Descriptor

Bit	Description	Read	Write	Value after Reset
1:0	Memory Space 1 Local Bus Width. Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
5:2	Memory Space 1 Internal Wait States (data-to-data; 0-15 wait states).	Yes	Yes	0h
6	Memory Space 1 TA#/READY# Input Enable. Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
7	Memory Space 1 BTERM# Input Enable. Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or Section 4.2.5 for C and J modes.	Yes	Yes	0
8	Memory Space 1 Burst Enable. Writing a 1 enables bursting. Writing a 0 disables bursting.	Yes	Yes	0
9	Memory Space 1 Prefetch Disable. When mapped into Memory space, writing a 0 enables Read prefetching. Writing a 1 disables prefetching. If prefetching is disabled, the PCI 9054 disconnects after each Memory read.	Yes	Yes	0
10	Read Prefetch Count Enable. When set to 1 and Memory prefetching is enabled, the PCI 9054 prefetches up to the number of Lwords specified in prefetch count. When set to 0, the PCI 9054 ignores the count and continues prefetching until it is terminated by the PCI Bus.	Yes	Yes	0
14:11	Read Prefetch Count. Number of Lwords to prefetch during Memory Read cycles (0-15).	Yes	Yes	0h
31:15	*Reserved.*	Yes	No	0h

Register 11-55. (DMDAC; PCI:FCh, LOC:17Ch) PCI Initiator PCI Dual Address Cycle

Bit	Description	Read	Write	Value after Reset
31:0	Upper 32 Bits of PCI Dual Address Cycle PCI Address during PCI Initiator Cycles. If set to 0, the PCI 9054 performs 32-bit PCI Initiator Address access.	Yes	Yes	0h

5.5.3.3 Registros de configuación DMA

Register 11-70. (DMAMODE0; PCI:80h, LOC:100h) DMA Channel 0 Mode

Bit	Description	Read	Write	Value after Reset
1:0	Local Bus Width. Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
5:2	Internal Wait States (data-to-data).	Yes	Yes	0h
6	TA#/READY# Input Enable. Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
7	BTERM# Input Enable. Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or Section 4.2.5 for C and J modes.	Yes	Yes	0
8	Local Burst Enable. Writing a 1 enables Local bursting. Writing a 0 disables Local bursting.	Yes	Yes	0
9	Scatter/Gather Mode. Writing a 1 indicates Scatter/Gather mode is enabled. For Scatter/Gather mode, DMA source address, destination address, and byte count are loaded from memory in PCI or Local Address spaces. Writing a 0 indicates Block mode is enabled.	Yes	Yes	0
10	Done Interrupt Enable. Writing a 1 enables an interrupt when done. Writing a 0 disables an interrupt when done. If DMA Clear Count mode is enabled, the interrupt does not occur until the byte count is cleared.	Yes	Yes	0
11	Local Addressing Mode. Writing a 1 holds the Local Address bus constant. Writing a 0 indicates the Local Address is incremented.	Yes	Yes	0
12	Demand Mode. Writing a 1 causes the DMA controller to operate in Demand mode, as well as to make USERo/DREQ0#/LLOCKo# an input (DREQ0#) and USERi/DACK0#/LLOCK# an output (DACK0#). In Demand mode, the DMA controller transfers data when its DREQ0# input is asserted. Asserts DACK0# to indicate the current Local Bus transfer is in response to DREQ0# input. DMA controller transfers Lwords (32 bits) of data. This may result in multiple transfers for an 8- or 16-bit bus.	Yes	Yes	0
13	Memory Write and Invalidate Mode for DMA Transfers. When set to 1, the PCI 9054 performs Memory Write and Invalidate cycles to the PCI Bus. The PCI 9054 supports Memory Write and Invalidate sizes of 8 or 16 Lwords. Size is specified in the System Cache Line Size bits (PCICLSR[7:0]). If a size other than 8 or 16 is specified, the PCI 9054 performs Write transfers rather than Memory Write and Invalidate transfers. Transfers must start and end at cache line boundaries.	Yes	Yes	0
14	DMA EOT# Enable. Writing a 1 enables the EOT# input pin. Writing a 0 disables the EOT# input pin.	Yes	Yes	0
15	Fast/Slow Terminate Mode Select. Writing a 0 sets PCI 9054 into the Slow Terminate mode. As a result in C or J modes, BLAST# is asserted on the last Data transfer to terminate DMA transfer. As a result in M mode, BDIP# is de-asserted at the nearest 16-byte boundary and stops the DMA transfer. Writing a 1 indicates that if EOT# is asserted or DREQ0# is de-asserted in Demand mode during DMA will immediately terminate the DMA transfer. In M mode, writing a 1 indicates BDIP# output is disabled. As a result, the PCI 9054 DMA transfer terminates immediately when EOT# is asserted or when DREQ0# is de-asserted in Demand mode.	Yes	Yes	0
16	DMA Clear Count Mode. Writing a 1 clears the byte count in each Scatter/Gather descriptor when the corresponding DMA transfer is complete.	Yes	Yes	0
17	DMA Channel 0 Interrupt Select. Writing a 1 routes the DMA Channel 0 interrupt to the PCI Bus interrupt. Writing a 0 routes the DMA Channel 0 interrupt to the Local Bus interrupt.	Yes	Yes	0
18	DAC Chain Load. When set to 1, enables the descriptor to load the PCI Dual Address Cycle value. Otherwise, it uses the contents of the register.	Yes	Yes	0
31:19	*Reserved.*	Yes	No	0h

Register 11-71. (DMAPADR0; PCI:84h, LOC:104h) DMA Channel 0 PCI Address

Bit	Description	Read	Write	Value after Reset
31:0	PCI Address Register. Indicates from where in PCI Memory space DMA transfers (reads or writes) start.	Yes	Yes	0h

Register 11-72. (DMALADR0; PCI:88h, LOC:108h) DMA Channel 0 Local Address

Bit	Description	Read	Write	Value after Reset
31:0	Local Address Register. Indicates from where in Local Memory space DMA transfers (reads or writes) start.	Yes	Yes	0h

Register 11-73. (DMASIZ0; PCI:8Ch, LOC:10Ch) DMA Channel 0 Transfer Size (Bytes)

Bit	Description	Read	Write	Value after Reset
22:0	DMA Transfer Size (Bytes). Indicates the number of bytes to transfer during a DMA operation.	Yes	Yes	0h
31:23	*Reserved.*	Yes	No	0h

Register 11-74. (DMADPR0; PCI:90h, LOC:110h) DMA Channel 0 Descriptor Pointer

Bit	Description	Read	Write	Value after Reset
0	Descriptor Location. Writing a 1 indicates PCI Address space. Writing a 0 indicates Local Address space.	Yes	Yes	0
1	End of Chain. Writing a 1 indicates end of chain. Writing a 0 indicates not end of chain descriptor. (Same as Block mode.)	Yes	Yes	0
2	Interrupt after Terminal Count. Writing a 1 causes an interrupt to be asserted after the terminal count for this descriptor is reached. Writing a 0 disables interrupts from being asserted.	Yes	Yes	0
3	Direction of Transfer. Writing a 1 indicates transfers from the Local Bus to the PCI Bus. Writing a 0 indicates transfers from the PCI Bus to the Local Bus.	Yes	Yes	0
31:4	Next Descriptor Address. Qword aligned (bits [3:0]=0000).	Yes	Yes	0h

Register 11-75. (DMAMODE1; PCI:94h, LOC:114h) DMA Channel 1 Mode

Bit	Description	Read	Write	Value after Reset
1:0	**Local Bus Width.** Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
5:2	**Internal Wait States** (data-to-data).	Yes	Yes	0h
6	**TA#/READY# Input Enable.** Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
7	**BTERM# Input Enable.** Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or Section 4.2.5 for C and J modes.	Yes	Yes	0
8	**Local Burst Enable.** Writing a 1 enables Local bursting. Writing a 0 disables Local bursting.	Yes	Yes	0
9	**Scatter/Gather Mode.** Writing a 1 indicates Scatter/Gather mode is enabled. For Scatter/Gather mode, the DMA source address, destination address, and byte count are loaded from memory in PCI or Local Address spaces. Writing a 0 indicates Block mode is enabled.	Yes	Yes	0
10	**Done Interrupt Enable.** Writing a 1 enables interrupt when done. Writing a 0 disables the interrupt when done. If DMA Clear Count mode is enabled, the interrupt does not occur until the byte count is cleared.	Yes	Yes	0
11	**Local Addressing Mode.** Writing a 1 holds the Local address bus constant. Writing a 0 indicates the Local address is incremented.	Yes	Yes	0
12	*Reserved.*	Yes	No	0
13	**Memory Write and Invalidate Mode for DMA Transfers.** When set to 1, the PCI 9054 performs Memory Write and Invalidate cycles to the PCI Bus. The PCI 9054 supports Memory Write and Invalidate sizes of 8 or 16 Lwords. Size is specified in the System Cache Line Size bits (PCICLSR[7:0]). If a size other than 8 or 16 is specified, the PCI 9054 performs Write transfers rather than Memory Write and Invalidate transfers. Transfers must start and end at cache line boundaries.	Yes	Yes	0
14	**DMA EOT# Enable.** Writing a 1 enables the EOT# input pin. Writing a 0 disables the EOT# output pin.	Yes	Yes	0
15	**Fast/Slow Terminate Mode Select.** Writing a 0 sets the PCI 9054 into Slow Terminate mode. As a result in C or J modes, BLAST# is asserted to terminate the DMA transfer. As a result in M mode, BDIP# is de-asserted at the nearest 16-byte boundary and stops the DMA transfer. Writing a 1 indicates that asserting EOT# during DMA will terminate the DMA transfer. In M mode, writing a 1 indicates BDIP# output is disabled. As a result, the PCI 9054 DMA transfer terminates immediately when EOT# is asserted.	Yes	Yes	0
16	**DMA Clear Count Mode.** When set to 1, the byte count in each Scatter/ Gather descriptor is cleared when the corresponding DMA transfer is complete.	Yes	Yes	0
17	**DMA Channel 1 Interrupt Select.** Writing a 1 routes the DMA Channel 1 interrupt to the PCI Bus interrupt. Writing a 0 routes the DMA Channel 1 interrupt to the Local Bus interrupt.	Yes	Yes	0
18	**DAC Chain Load.** When set to 1, enables the descriptor to load the PCI Dual Address Cycle value. Otherwise, it uses the contents of the register.	Yes	Yes	0
31:19	*Reserved.*	Yes	No	0h

Register 11-76. (DMAPADR1; PCI:98h, LOC:118h) DMA Channel 1 PCI Address

Bit	Description	Read	Write	Value after Reset
31:0	**PCI Address Register.** Indicates from where in PCI Memory space DMA transfers (reads or writes) start.	Yes	Yes	0h

Register 11-77. (DMALADR1; PCI:9Ch, LOC:11Ch) DMA Channel 1 Local Address

Bit	Description	Read	Write	Value after Reset
31:0	**Local Address Register.** Indicates from where in Local Memory space DMA transfers (reads or writes) start.	Yes	Yes	0h

Register 11-78. (DMASIZ1; PCI:A0h, LOC:120h) DMA Channel 1 Transfer Size (Bytes)

Bit	Description	Read	Write	Value after Reset
22:0	DMA Transfer Size (Bytes). Indicates the number of bytes to transfer during a DMA operation.	Yes	Yes	0h
31:23	*Reserved.*	Yes	No	0h

Register 11-79. (DMADPR1; PCI:A4h, LOC:124h) DMA Channel 1 Descriptor Pointer

Bit	Description	Read	Write	Value after Reset
0	Descriptor Location. Writing a 1 indicates PCI Address space. Writing a 0 indicates Local Address space.	Yes	Yes	0
1	End of Chain. Writing a 1 indicates end of chain. Writing a 0 indicates not end of chain descriptor. (Same as Block mode.)	Yes	Yes	0
2	Interrupt after Terminal Count. Writing a 1 causes an interrupt to be asserted after the terminal count for this descriptor is reached. Writing a 0 disables interrupts from being asserted.	Yes	Yes	0
3	Direction of Transfer. Writing a 1 indicates transfers from the Local Bus to the PCI Bus. Writing a 0 indicates transfers from the PCI Bus to the Local Bus.	Yes	Yes	0
31:4	Next Descriptor Address. Qword aligned (bits [3:0]=0000).	Yes	Yes	0h

Register 11-80. (DMACSR0; PCI:A8h, LOC:128h) DMA Channel 0 Command/Status

Bit	Description	Read	Write	Value after Reset
0	Channel 0 Enable. Writing a 1 enables channel to transfer data. Writing a 0 disables the channel from starting a DMA transfer, and if in the process of transferring data, suspends the transfer (pause).	Yes	Yes	0
1	Channel 0 Start. Writing a 1 causes the channel to start transferring data if the channel is enabled.	No	Yes/Set	0
2	Channel 0 Abort. Writing a 1 causes the channel to abort current transfer. Channel 0 Enable bit must be cleared (bit [0]=0). Sets Channel 0 Done (bit [4] = 1) when abort is complete.	No	Yes/Set	0
3	Channel 0 Clear Interrupt. Writing a 1 clears Channel 0 interrupts.	No	Yes/Clr	0
4	Channel 0 Done. Reading a 1 indicates a channel transfer is complete. Reading a 0 indicates a channel transfer is not complete.	Yes	No	1
7:5	*Reserved.*	Yes	No	000

Register 11-81. (DMACSR1; PCI:A9h, LOC:129h) DMA Channel 1 Command/Status

Bit	Description	Read	Write	Value after Reset
0	Channel 1 Enable. Writing a 1 enables channel to transfer data. Writing a 0 disables the channel from starting a DMA transfer, and if in the process of transferring data, suspends the transfer (pause).	Yes	Yes	0
1	Channel 1 Start. Writing a 1 causes channel to start transferring data if the channel is enabled.	No	Yes/Set	0
2	Channel 1 Abort. Writing a 1 causes channel to abort current transfer. Channel 1 Enable bit must be cleared (bit [0]=0). Sets Channel 1 Done (bit [4] = 1) when abort is complete.	No	Yes/Set	0
3	Channel 1 Clear Interrupt. Writing a 1 clears Channel 1 interrupts.	No	Yes/Clr	0
4	Channel 1 Done. Reading a 1 indicates a channel transfer is complete. Reading a 0 indicates a channel transfer is not complete.	Yes	No	1
7:5	*Reserved.*	Yes	No	000

Register 11-83. (DMATHR; PCI:B0h, LOC:130h) DMA Threshold

Bit	Description	Read	Write	Value after Reset
3:0	DMA Channel 0 PCI-to-Local Almost Full (C0PLAF). Number of full entries (divided by two, minus one) in the FIFO before requesting the Local Bus for writes. (C0PLAF+1) + (C0PLAE+1) should be ≤ a FIFO Depth of 32.	Yes	Yes	0h
7:4	DMA Channel 0 Local-to-PCI Almost Empty (C0LPAE). Number of empty entries (divided by two, minus one) in the FIFO before requesting the Local Bus for reads. (C0LPAF+1) + (C0LPAE+1) should be ≤ a FIFO depth of 32.	Yes	Yes	0h
11:8	DMA Channel 0 Local-to-PCI Almost Full (C0LPAF). Number of full entries (divided by two, minus one) in the FIFO before requesting the PCI Bus for writes.	Yes	Yes	0h
15:12	DMA Channel 0 PCI-to-Local Almost Empty (C0PLAE). Number of empty entries (divided by two, minus one) in the FIFO before requesting the PCI Bus for reads.	Yes	Yes	0h
19:16	DMA Channel 1 PCI-to-Local Almost Full (C1PLAF). Number of full entries, minus one, in the FIFO before requesting the Local Bus for writes. (C1PLAF+1) + (C1PLAE+1) should be ≤ a FIFO depth of 16.	Yes	Yes	0h
23:20	DMA Channel 1 Local-to-PCI Almost Empty (C1LPAE). Number of empty entries, minus one, in the FIFO before requesting the Local Bus for reads. (C1PLAF) + (C1PLAE) should be ≤ a FIFO depth of 16.	Yes	Yes	0h
27:24	DMA Channel 1 Local-to-PCI Almost Full (C1LPAF). Number of full entries, minus one, in the FIFO before requesting the PCI Bus for writes.	Yes	Yes	0h
31:28	DMA Channel 1 PCI-to-Local Almost Empty (C1PLAE). Number of empty entries, minus one, in the FIFO before requesting the PCI Bus for reads.	Yes	Yes	0h

Note: For DMA Channel 0 only, if number of entries needed is x, then the value is one less than half the number of entries (that is, x/2 - 1).

Register 11-84. (DMADAC0; PCI:B4h, LOC:134h) DMA Channel 0 PCI Dual Address Cycle Address

Bit	Description	Read	Write	Value after Reset
31:0	Upper 32 Bits of the PCI Dual Address Cycle PCI Address during DMA Channel 0 Cycles. If set to 0h, the PCI 9054 performs a 32-bit DMA Channel 0 Address access.	Yes	Yes	0h

Register 11-85. (DMADAC1; PCI:B8h, LOC:138h) DMA Channel 1 PCI Dual Address Cycle Address

Bit	Description	Read	Write	Value after Reset
31:0	Upper 32 Bits of the PCI Dual Address Cycle PCI Address during DMA Channel 1 Cycles. If set to 0h, the PCI 9054 performs a 32-bit DMA Channel 1 Address access.	Yes	Yes	0h

5.6 Cronogramas.

En este apartado se mostrarán y describirán diferentes cronogramas representando el modo de transferencia PCI Target en el modo de operación C.

Para nuestra aplicación, utilizaremos el modo de transferencia, tanto para escritura como lectura, de 4-Lword.
De esta manera, sólo conectaremos los 8 bits más bajos del Bus local, a nuestra tarjeta XS40, debido a las limitaciones que presenta de poseer una memoria con un ancho de dato de 8 bits.

El software del PC, será el encargado de transformar los datos Lword en datos de 8 bits, introduciendo ceros en los restantes bits que dejaremos 'al aire'. De esta forma dejaremos la aplicación lista parta trabajar a 32 bits, simplemente cambiando el bucle que transforma los datos de 32 bits a 8 bits.

5.6.1 Arbitración de Bus Local.

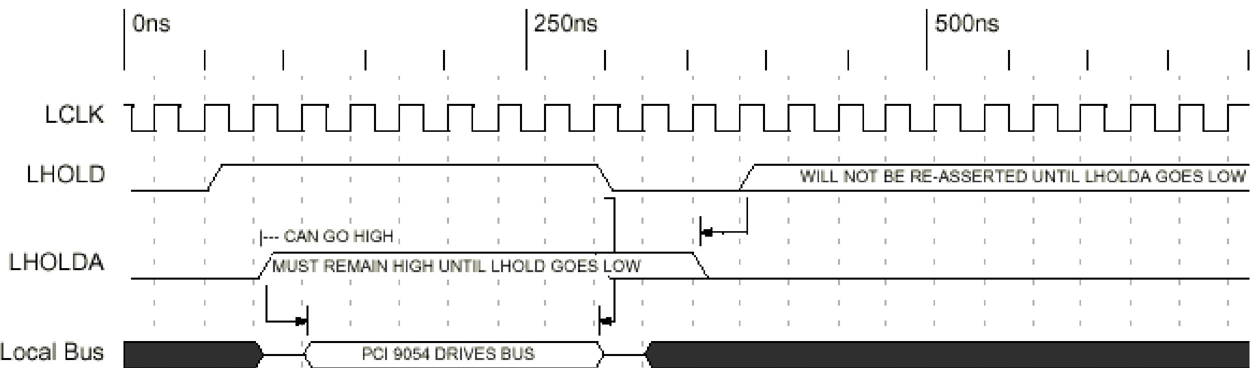


Figura 5.25: Señales de arbitración del Bus Local.

Para la petición del Bus Local el controlador PCI 9054 activará la señal LHOLD, requiriendo el bus, el cuál no será cedido hasta que el procesador local responda con la señal LHOLDA cediéndoselo. Una vez que el procesador haya terminado con el bus desactivará LHOLD, y no podrá a volver a requerirlo hasta que el procesador local no haya desactivado la señal LHOLDA.

5.6.2. Escritura/ Lectura de los registros de configuración.

5.6.2.1 Escritura de los registros de configuración PCI.

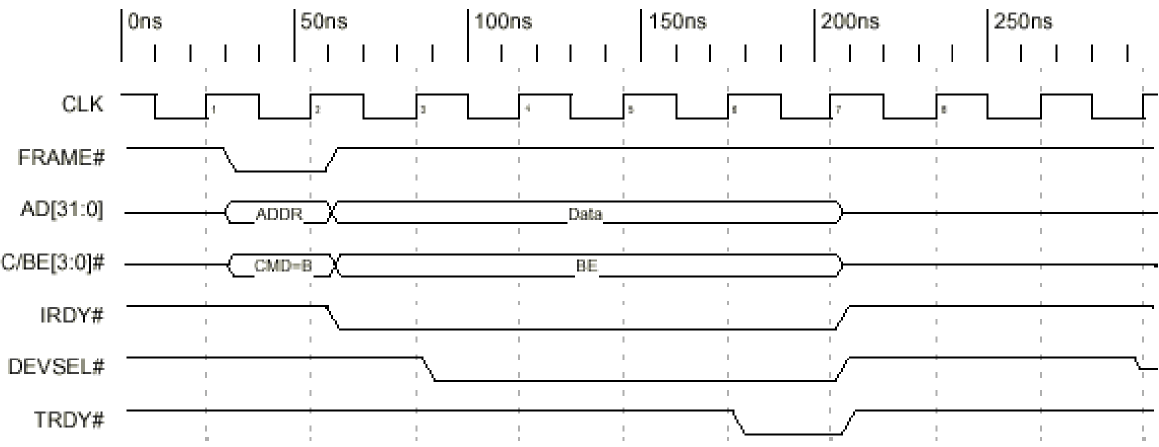


Figura 5.26: Escritura de los registros de configuración PCI.

Se activa FRAME# indicando el comienzo del acceso, a su vez por el Bus de Datos se indicará la dirección a la cual se accede y por el bus de código de comando se indicará el tipo de transición. Una vez realizado esto, se activa la señal IRDY# indicando que el dato colocado en el bus de A/D es válido. Se activa la señal DEVSEL# indicando que el dispositivo tiene descodificada su dirección de acceso. Finalmente se activa TRDY# indicando que se está preparado para recibir el dato.

5.6.2.2 Lectura de los registros de configuración PCI.

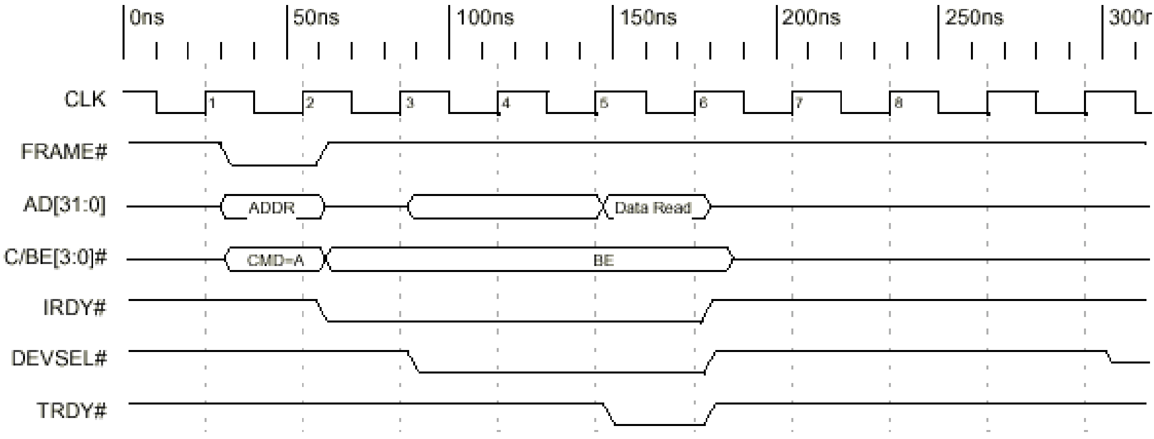


Figura 5.27: Lectura de los registros de configuración PCI.

4.6.2.3 Escritura de los registros de configuración Local.

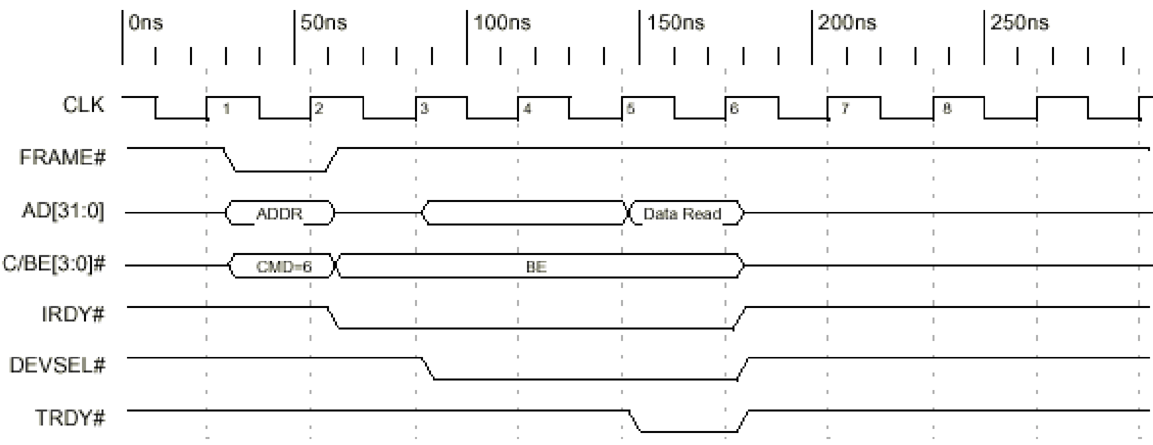


Figura 5.28: Escritura de los registros de configuración Local.

5.6.2.4 Lectura de los registros de configuración Local.

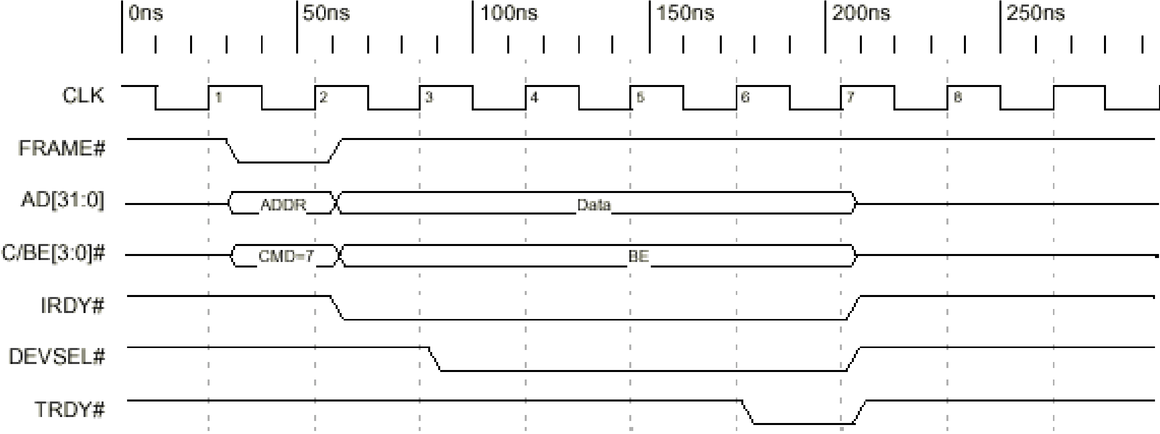
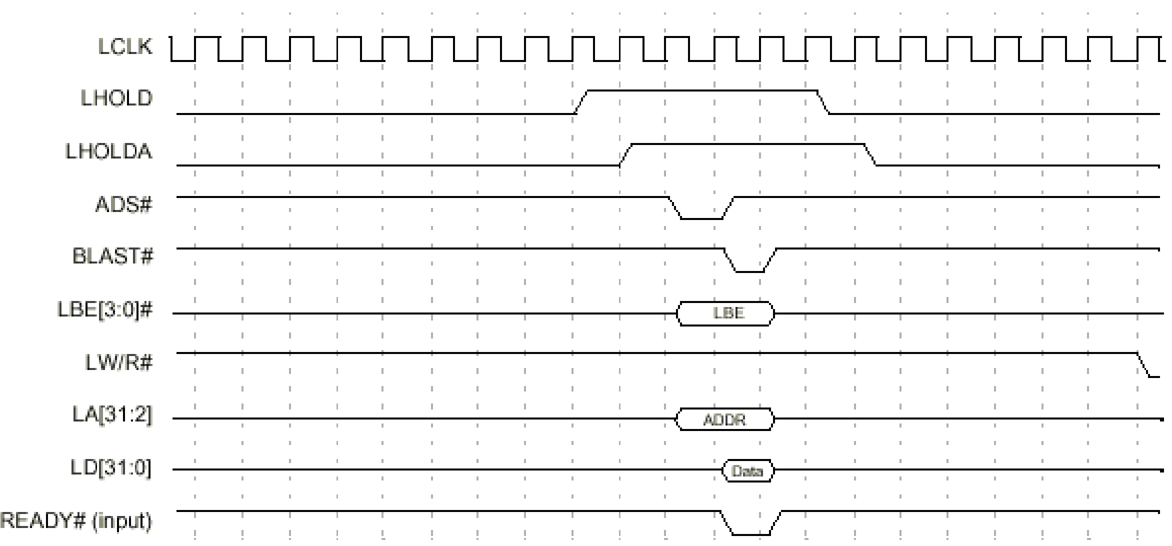


Figura 5.29: Lectura de los registros de configuración Local.

5.5.2 Ciclo de escritura Singular de 32 bits del Bus Local.



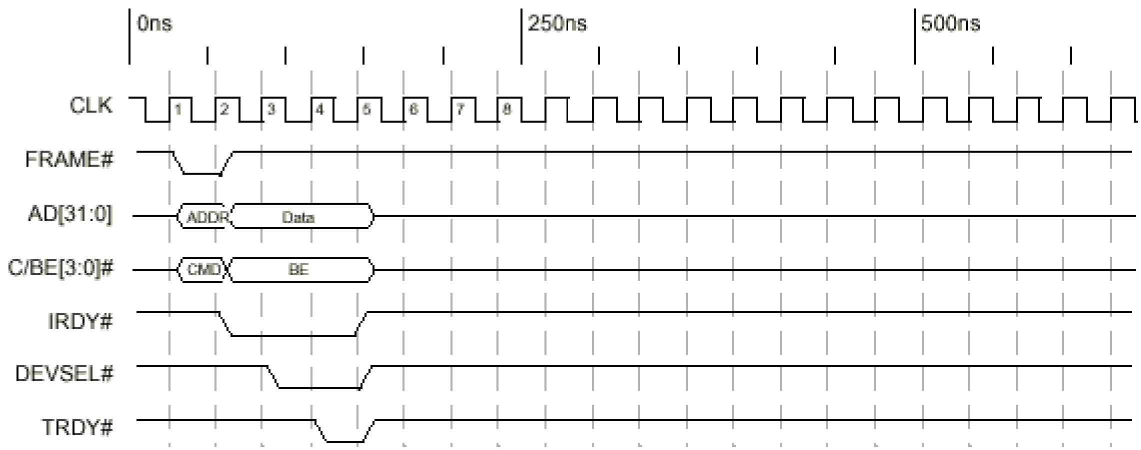


Figura 5.30: Ciclo de escritura Singular de 32 bits del Bus Local.

Una vez seleccionado el dispositivo, tras poner su dirección y datos válidos, el controlador pide el Bus Local (LHOLD). Tras recibir la cesión de este(LHOLDA), se activa la señal ADS# indicando que a partir de aquí la dirección en el Bus local será válida hasta que se active la señal de fin de bus, BLAST#.

La señal LBE# indicará que byte del dato se está transfiriendo. Se colocará la dirección y el dato que se aceptará mientras el esclavo tenga activa la señal READY#.

De igual forma, se realizará para la transferencia de datos de 16 y 8 bits, con la diferencia del bus LBE#, que indicará que tipo de byte está siendo transferido.(remítase a la sección del bus LBE#).

5.6.4 Interrupción Local activando la interrupción PCI.

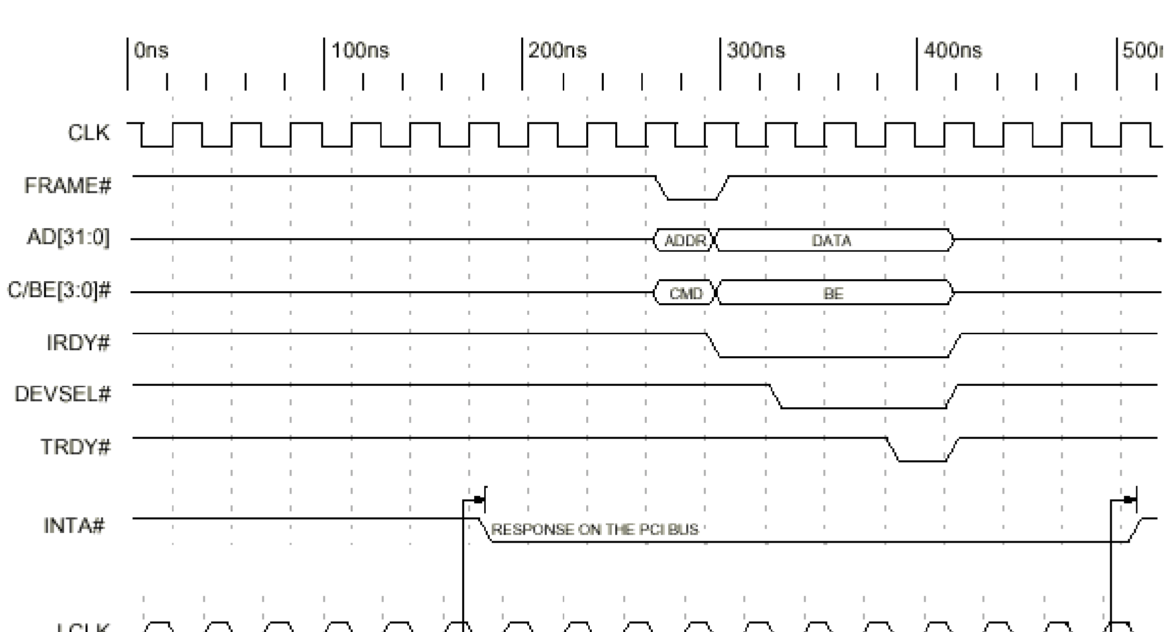


Figura 5.31: Interrupción Local activando la interrupción PCI.

Cuando una interrupción local se activa, hace que en el siguiente ciclo de reloj se active la interrupción PCI.

4.6.5 Ciclo de escritura Burst de 16 bits del Bus Local.

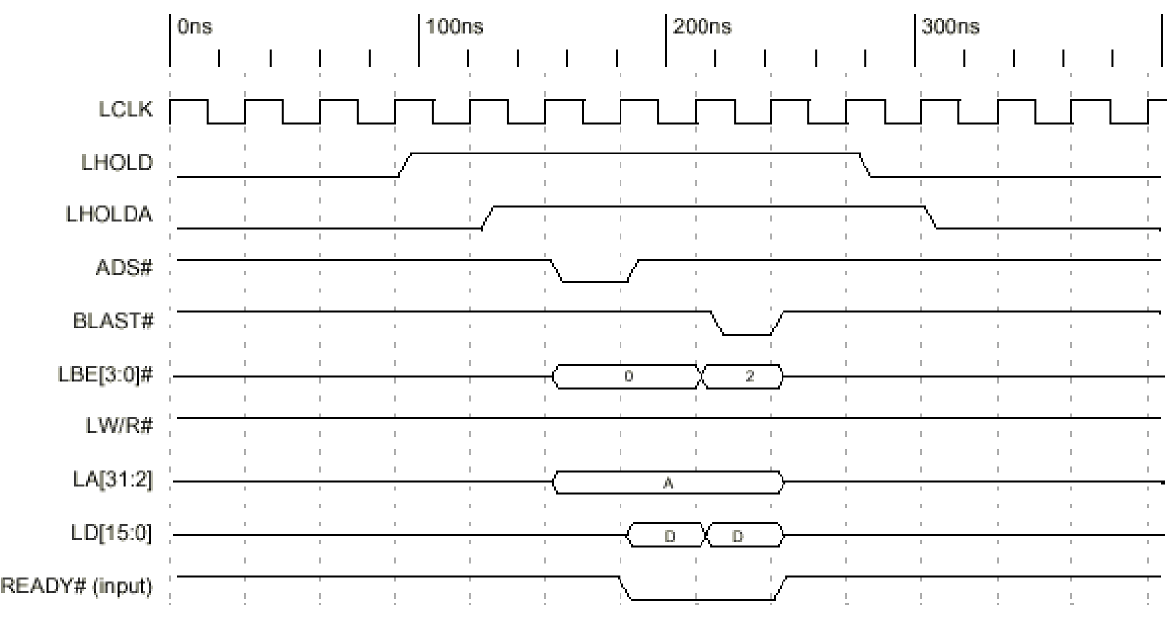


Figura 5.32: Ciclo de escritura Burst de 16 bits del Bus Local

5.6.6 Ciclo de lectura Singular de 32 bits del Bus Local.

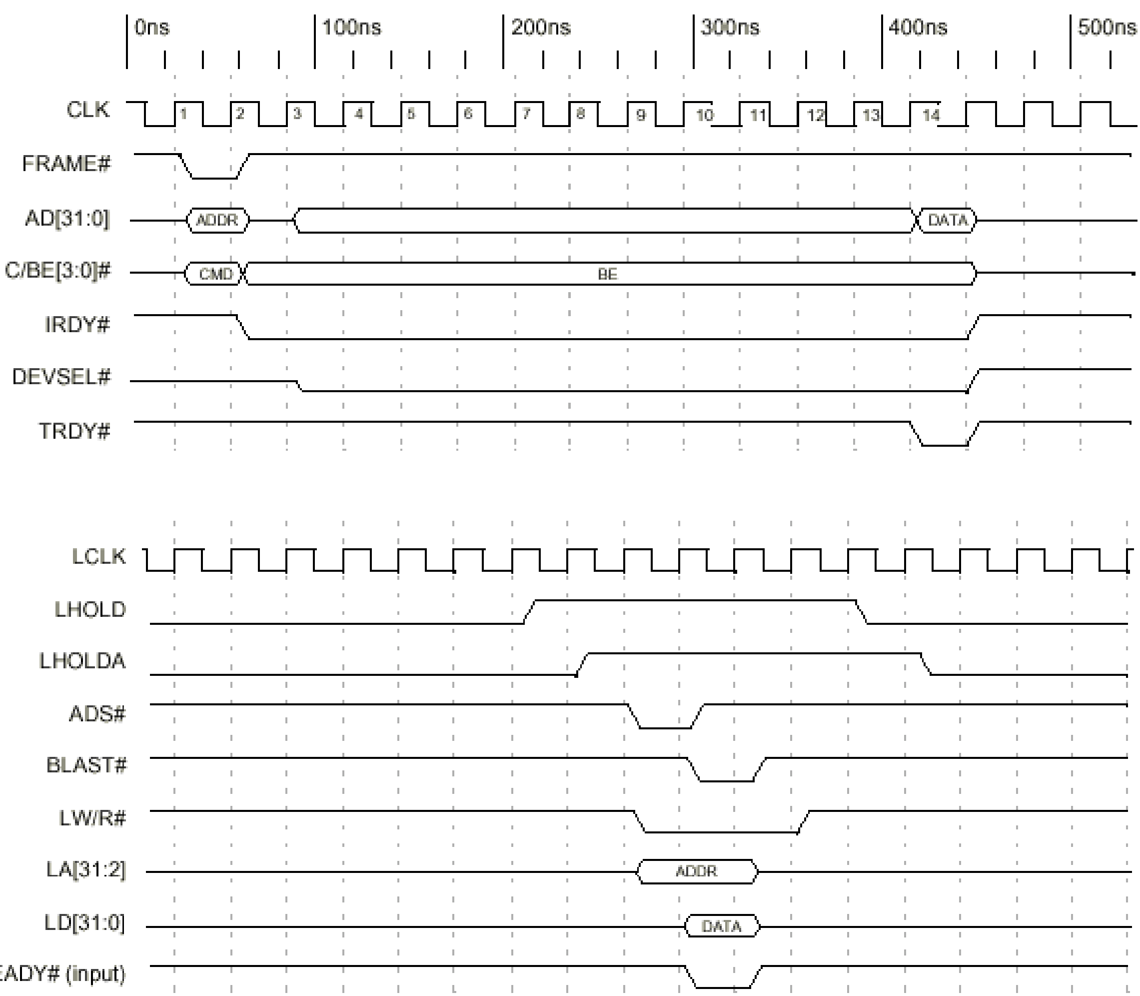


Figura 5.33:Ciclo de lectura Singular de 32 bits del Bus Local.

De igual manera que para la escritura, con la diferencia de la señal LW/R# que indicará que se está realizando una lectura.

5.6.7 Ciclo de escritura Burst de 8 bits del Bus Local.

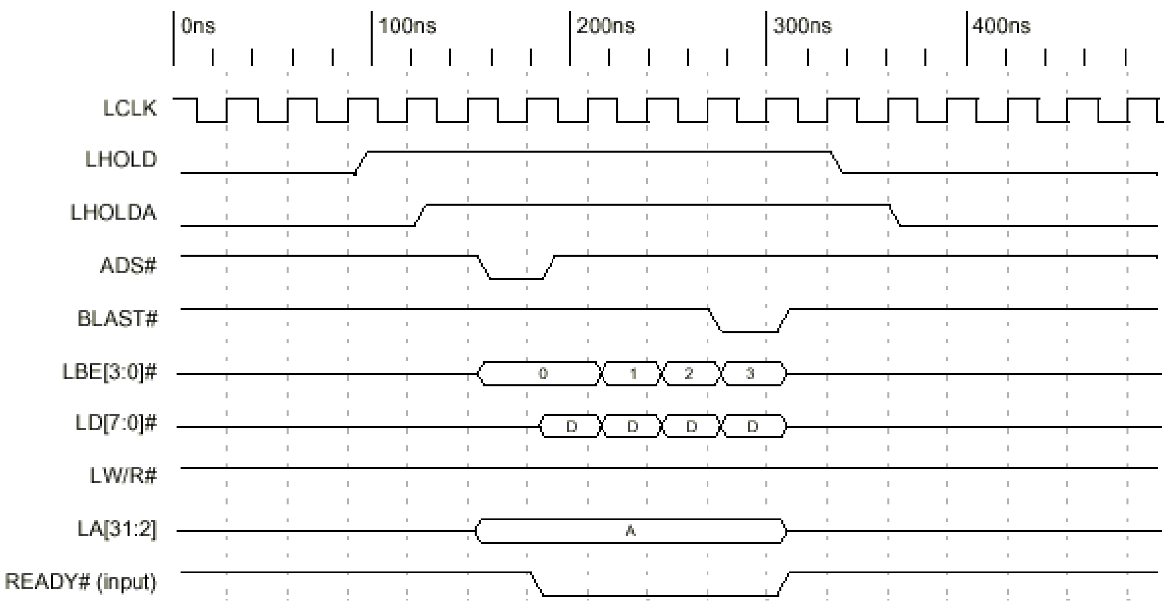
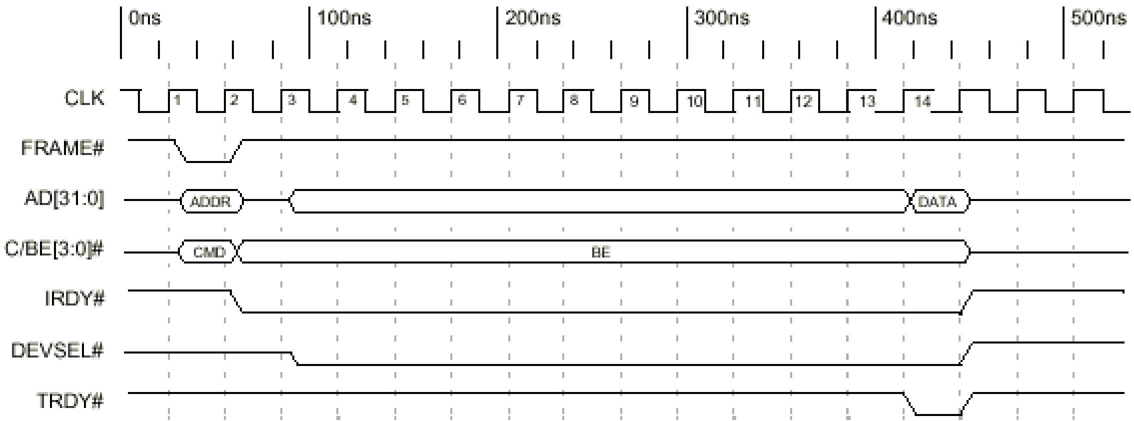


Figura5.34: Ciclo de escritura Burst de 8 bits del Bus Local.

5.6.8 Ciclo de lectura Singular de 32 bits del Bus Local, con un estado de espera usando la señal READY#.



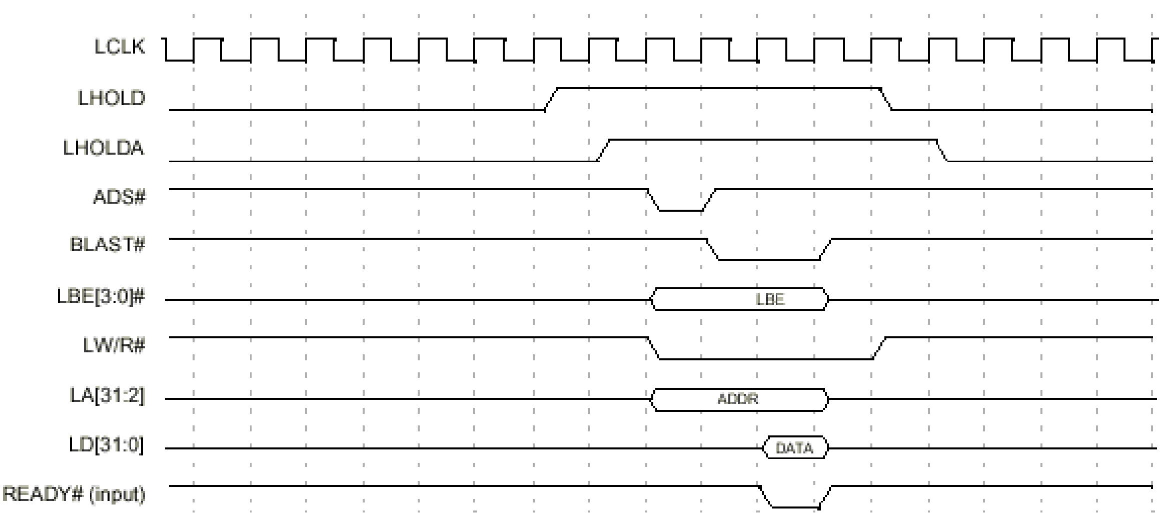


Figura 5.35: Ciclo de lectura Singular de 32 bits del Bus Local, con un estado de espera usando la señal READY#.

5.6.9 Ciclo de escritura sin ráfaga de 8 bits del Bus Local.

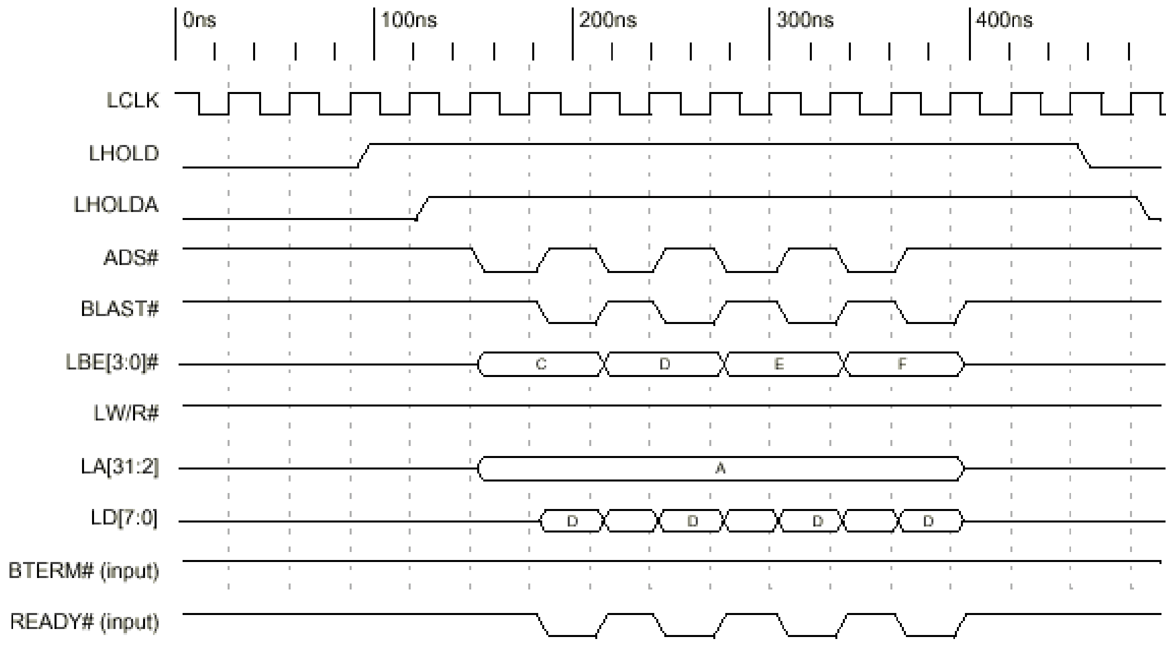


Figura 5.36: Ciclo de escritura sin ráfaga de 8 bits del Bus Local.

En este tipo de transferencia sin ráfaga, para cada dato se tendrá que activar una señal READY# entre la activación de las señales ADS# y BLAST#.
La señal LBE# nos indicará las dos líneas bajas de la dirección.

5.6.10 Ciclo de lectura Singular de 32 bits del Bus Local, con un estado de espera usando el estado interno de espera.

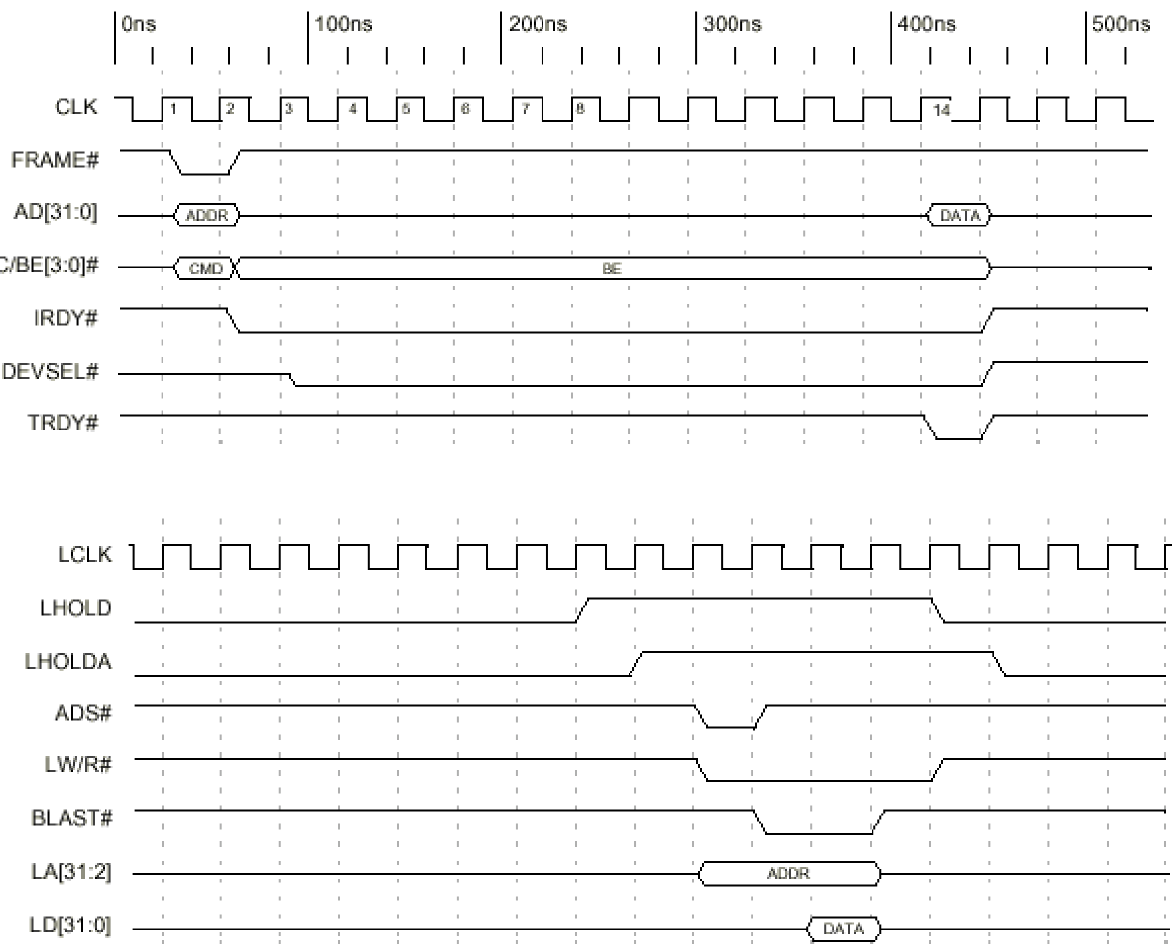
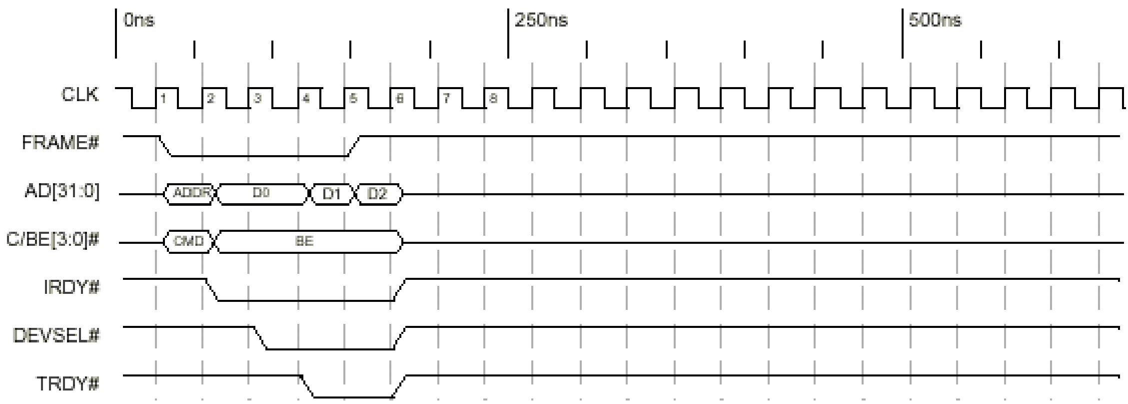


Figura 5.37: Ciclo de lectura Singular de 32 bits del Bus Local, con un estado de espera usando el estado interno de espera.

En este, no se necesita la señal READY#, generándose internamente desde el controlador, ya que se configuró como un estado de espera Wait.

5.6.11 Ciclo de escritura sin ráfaga de 32 bits del Bus Local.



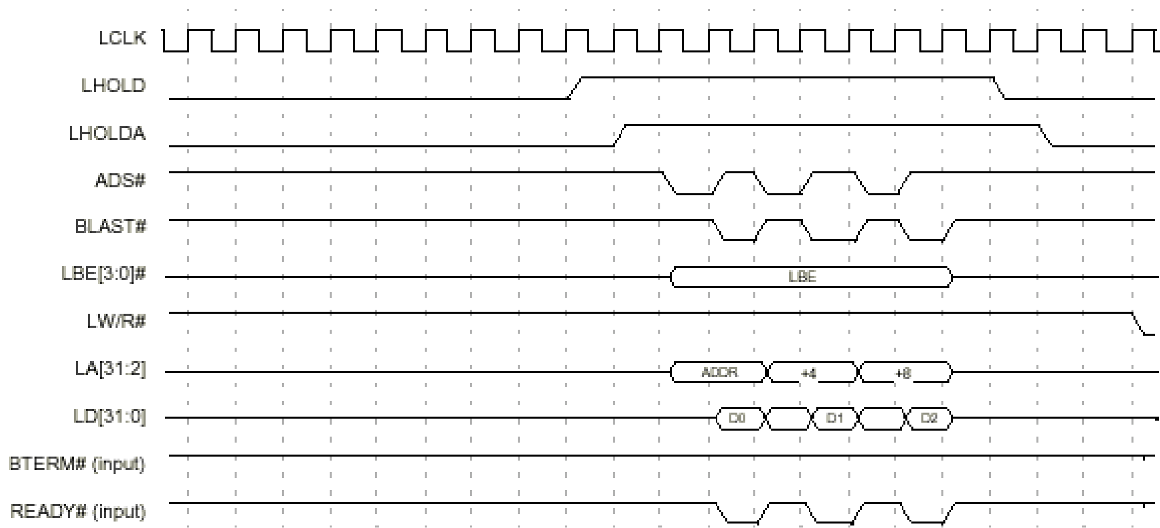
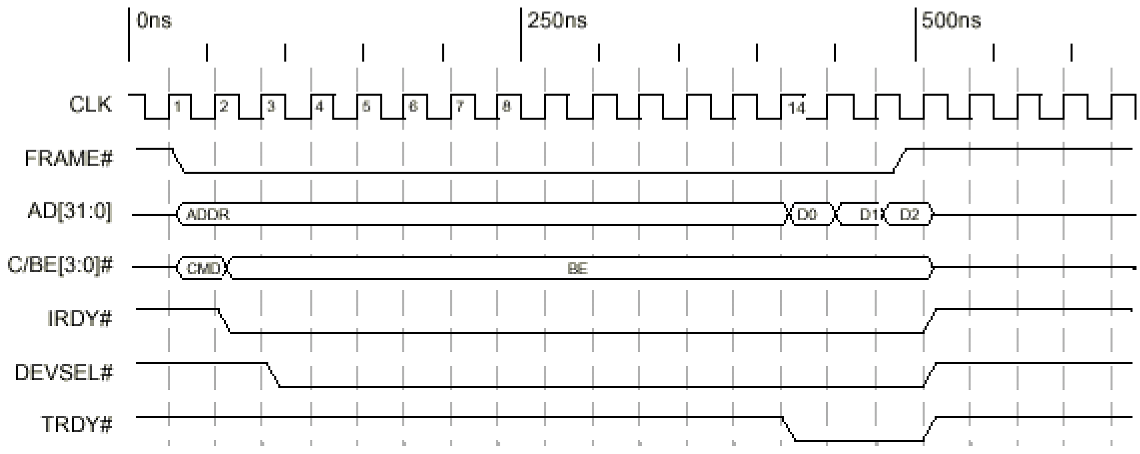


Figura 5.38: Ciclo de escritura sin ráfaga de 32 bits del Bus Local.

4.6.12 Ciclo de lectura sin ráfaga de 32bits del Bus Local.



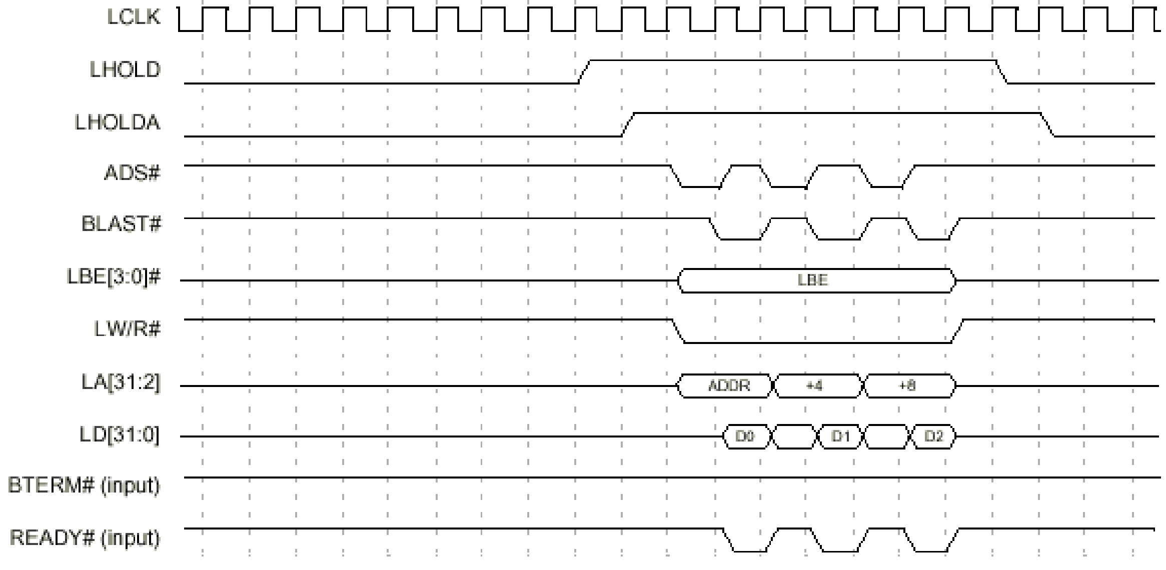


Figura 5.39: Ciclo de lectura sin ráfaga de 32bits del Bus Local.

Se transmitirá de igual manera que para la transferencia de 8 bits, con la diferencia del bus LBE# y que las direcciones se incrementarán automáticamente de 4 en 4.

Dirección siguiente = dirección actual + 4

5.6.13 Ciclo de escritura modo ráfaga de 32bits del Bus Local con el registro Bterm activo.

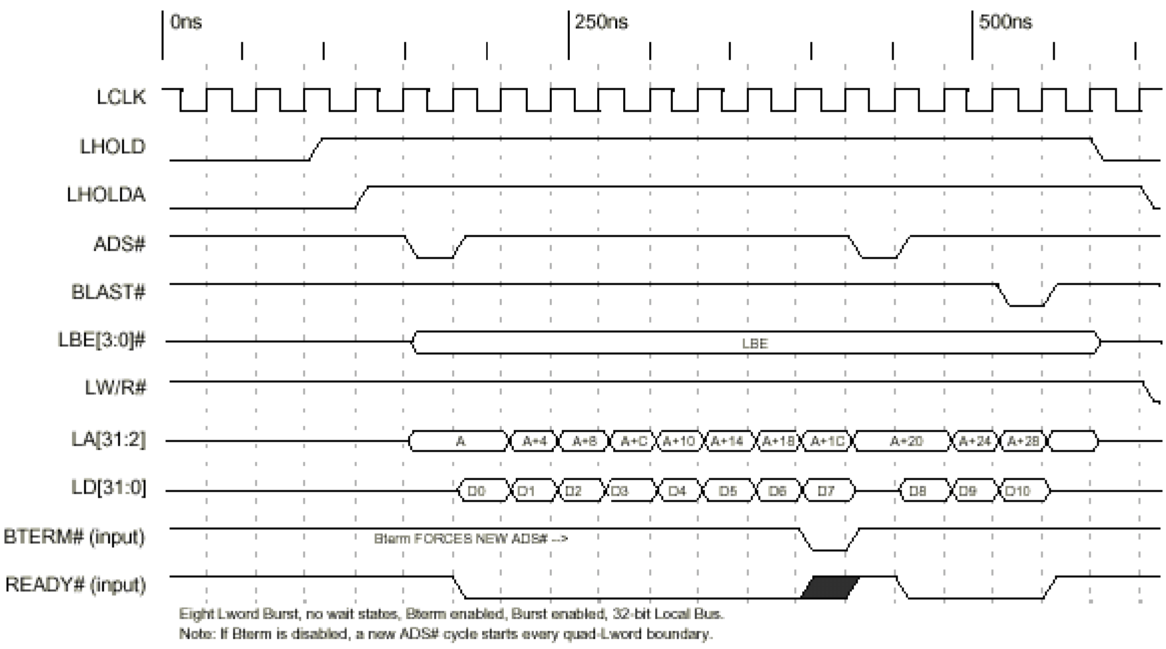


Figura 5.40: Ciclo de escritura modo ráfaga de 32bits del Bus Local con el registro Bterm activo.

5.6.14 Escritura modo ráfaga de 32bits del Bus Local con Bterm inactivo.

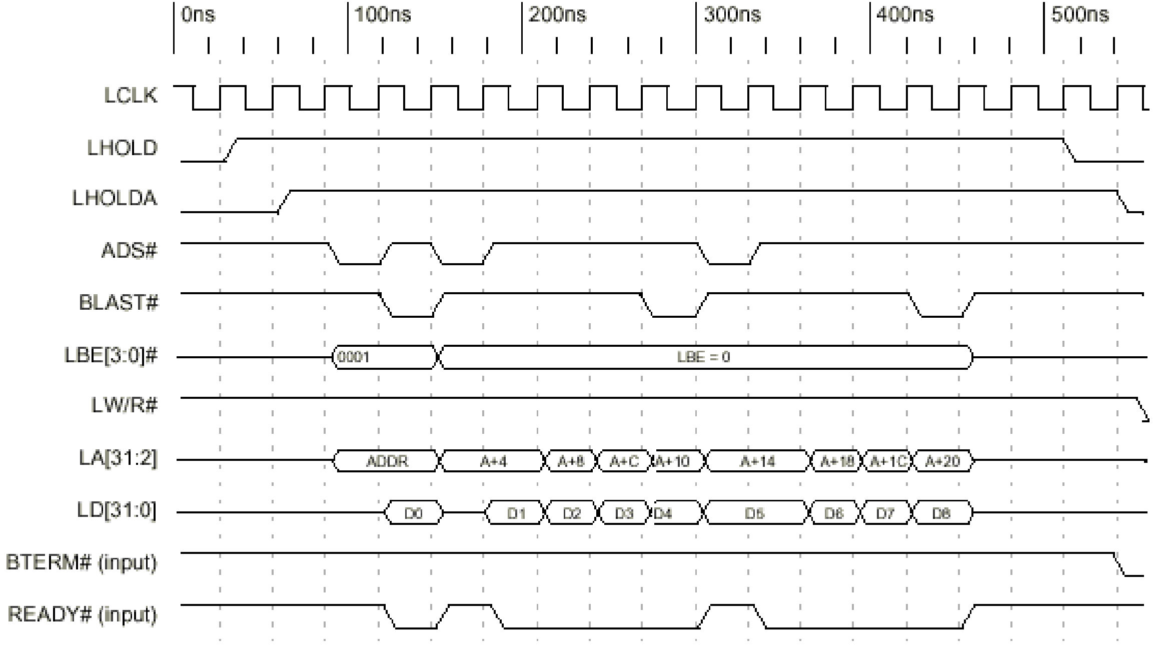


Figura 5.41: Ciclo de escritura modo ráfaga de 32bits del Bus Local con el registro Bterm inactivo.

5.6.15 Ciclo de lectura modo ráfaga de 32bits del Bus Local con el contador de pretransferencia puesto a 8.

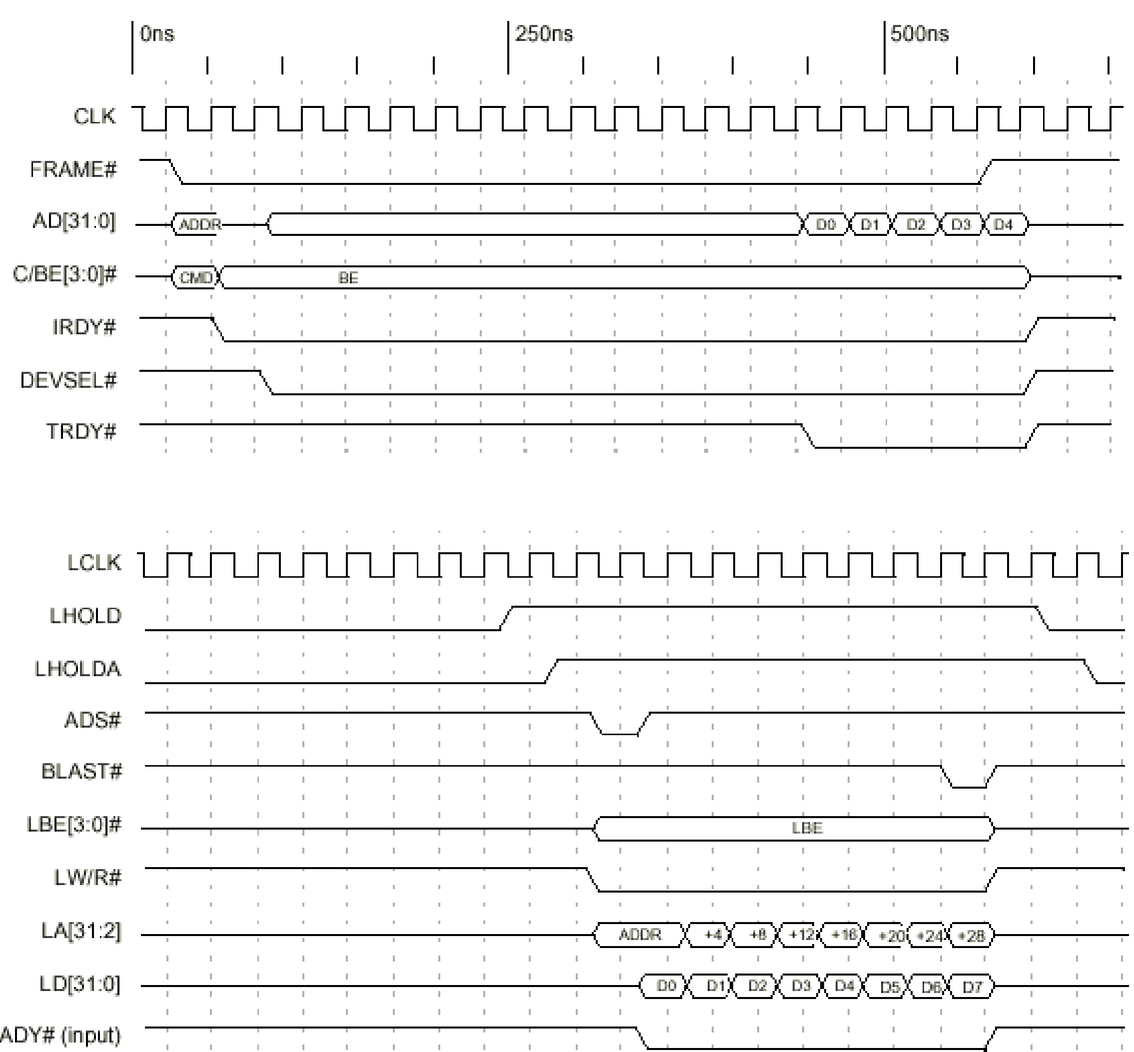


Figura 5.42: Ciclo de lectura modo ráfaga de 32bits del Bus Local con el contador de pretransferencia puesto a 8.

En el caso de transferencia en modo ráfaga, diferentes de 4 Lword. La señal BTERM# debe estar habilitada, indicando el inicio y fin de ráfaga con la activación y desactivación de la señal.
En esta se transfieren 8 Lword en modo ráfaga y se desactiva la señal. En el caso de estar desactivo BTERM#, una nueva señal ADS# se activará en cada transferencia de 4-Lword.
La dirección se irá incrementando de 4 en 4.
En este, el contador de transferencia puesto a 8 en el registro de configuración indica que el ciclo de transferencia terminará una vez llegado al límite de este.

5.6.16 Ciclo de escritura modo ráfaga de 32 bits del Bus Local.

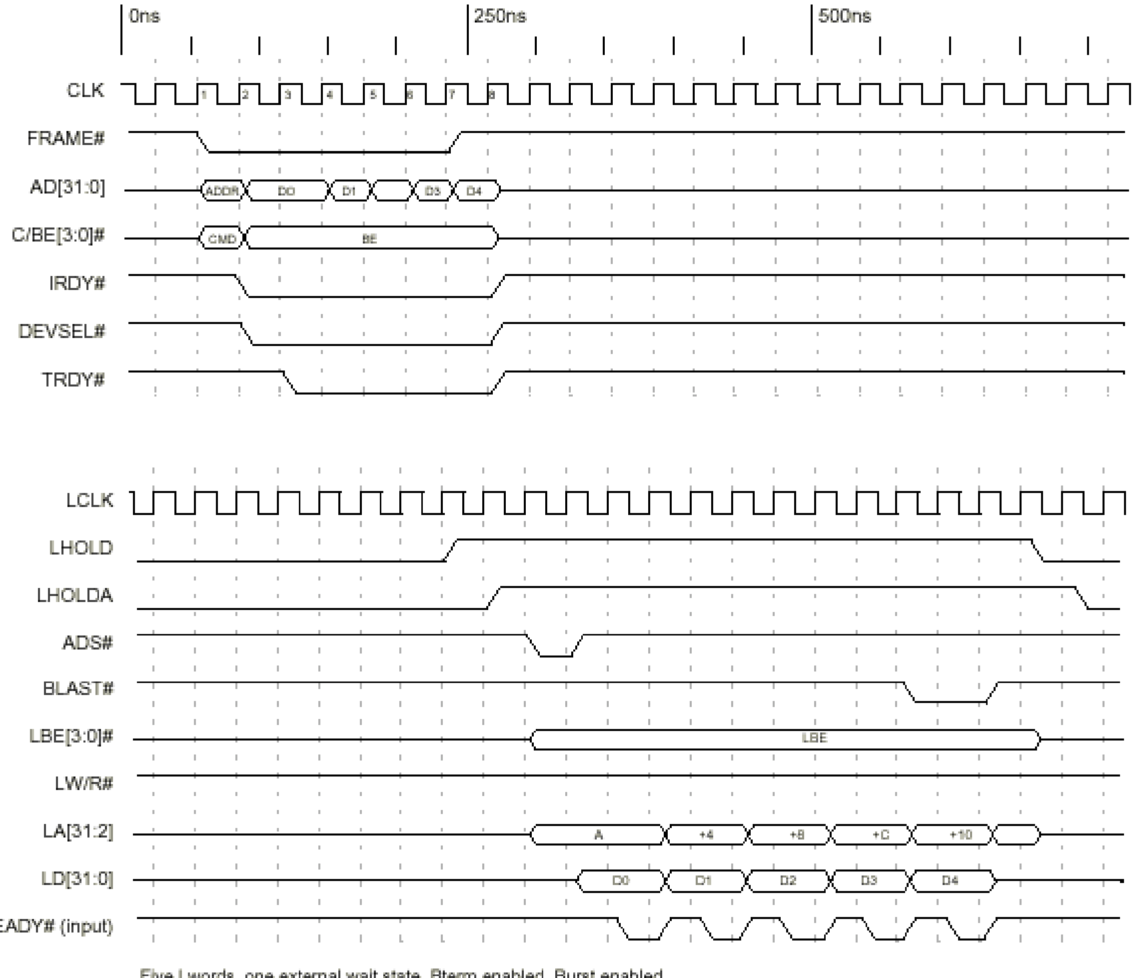


Figura 5.43: Ciclo de escritura modo ráfaga de 32 bits del Bus Local.

El maestro realiza una transferencia de 5 Lwords, estando deshabilitada la señal BTERM#

5.6.17 Ciclo de escritura modo ráfaga de 16 bits del Bus Local

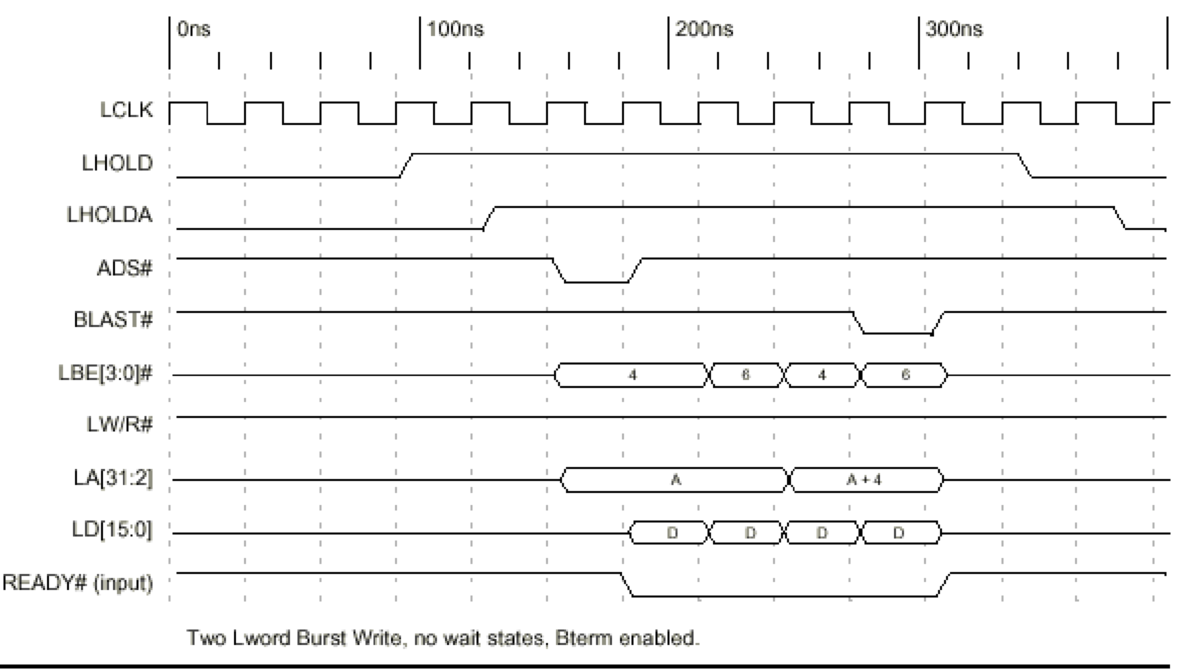


Figura 5.44: Ciclo de escritura modo ráfaga de 16 bits del Bus Local.

5.6.18 Ciclo de escritura modo ráfaga de 8 bits del Bus Local.

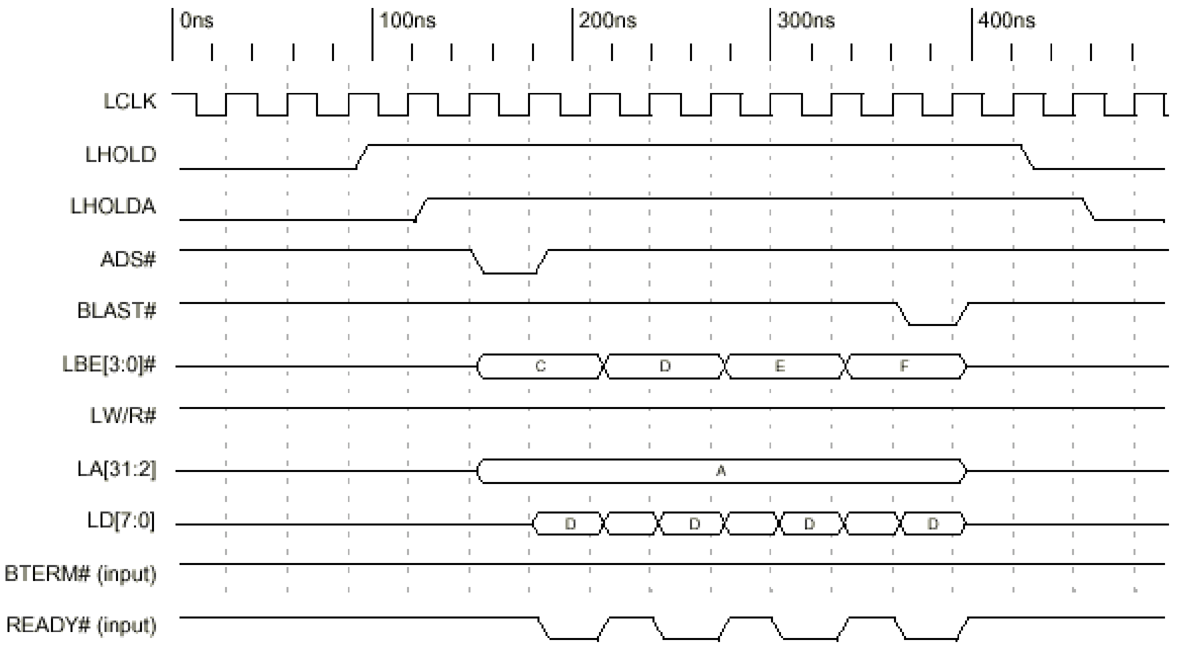
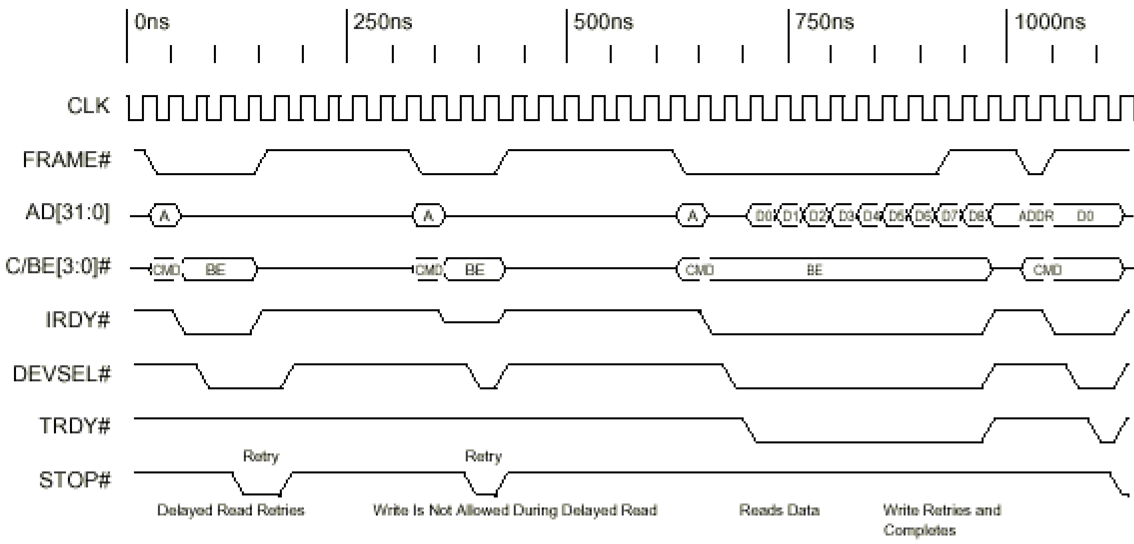


Figura 5.45: Ciclo de escritura modo ráfaga de 8 bits del Bus Local.

4.6.19 Retraso en la transferencia de lectura especificación PCI v2.1.



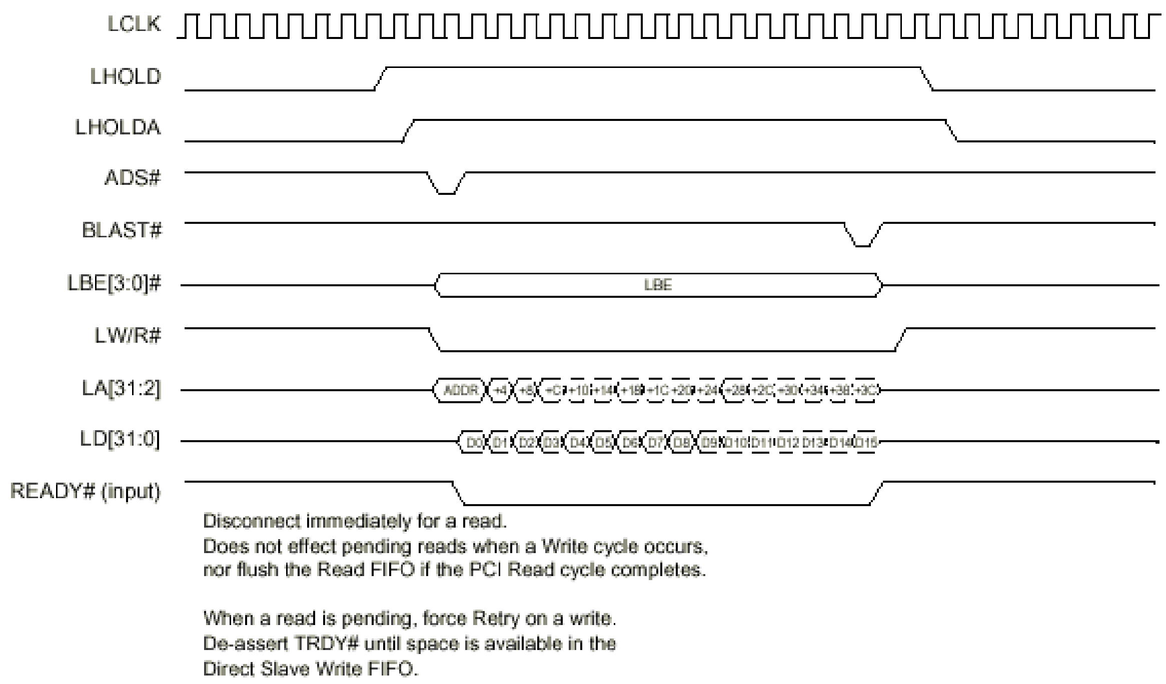


Figura 5.46: Retraso en la transferencia de lectura especificación PCI v2.1.

4.6.20 Ciclo de lectura sin modo flujo, (Read Ahead), pretransferencia activada y contador de la pretransferencia desactivada de 32 bits del Bus Local.

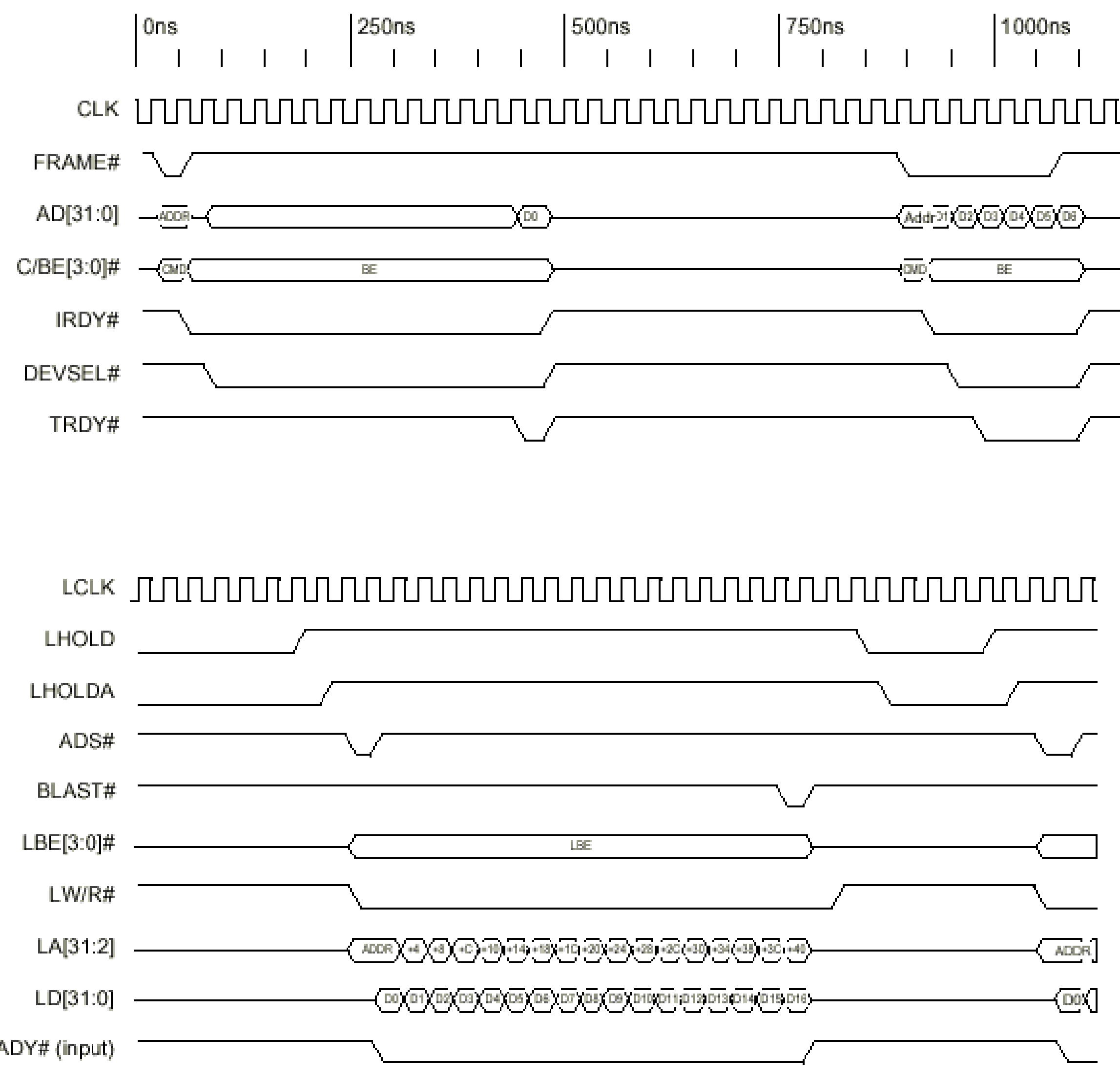


Figura 5.47: Ciclo de lectura sin modo flujo, (Read Ahead), pretransferencia activada y contador de la pretransferencia desactivada de 32 bits del Bus Local.

4.6.21 Ciclo de escritura en modo ráfaga, suspendida por la señal BREQi de 32 bits del Bus Local.

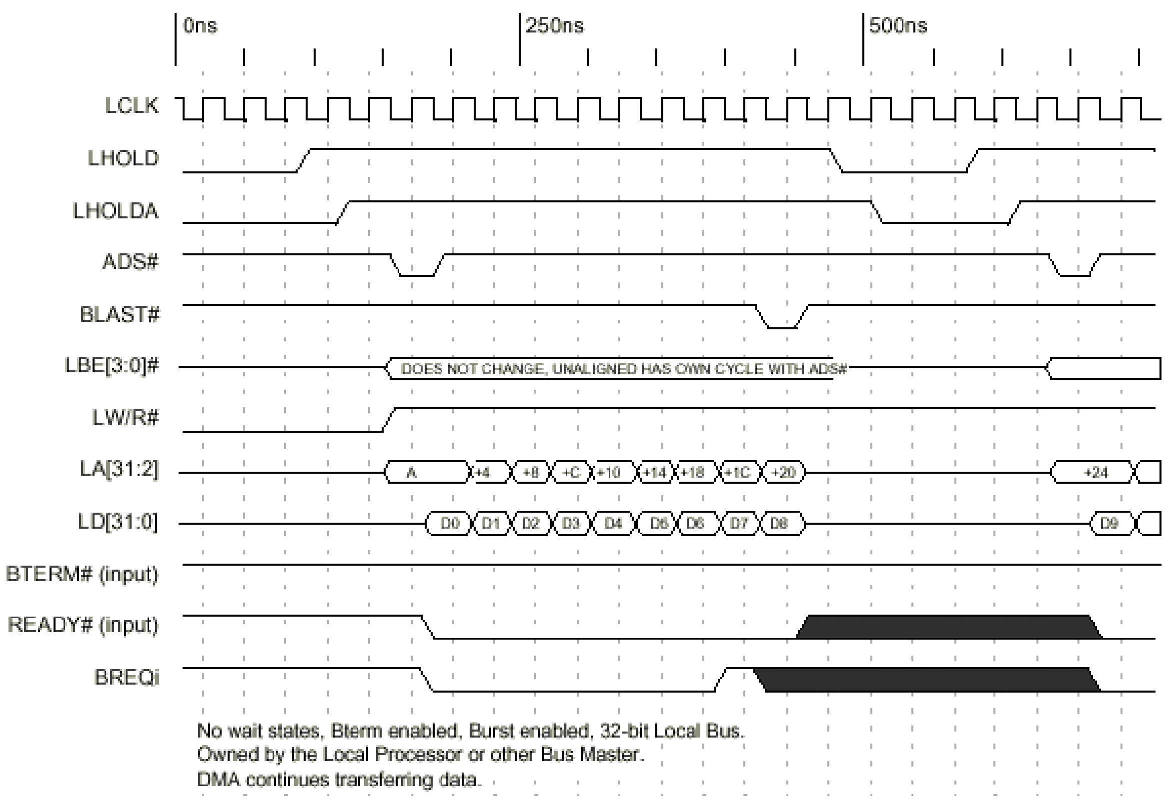
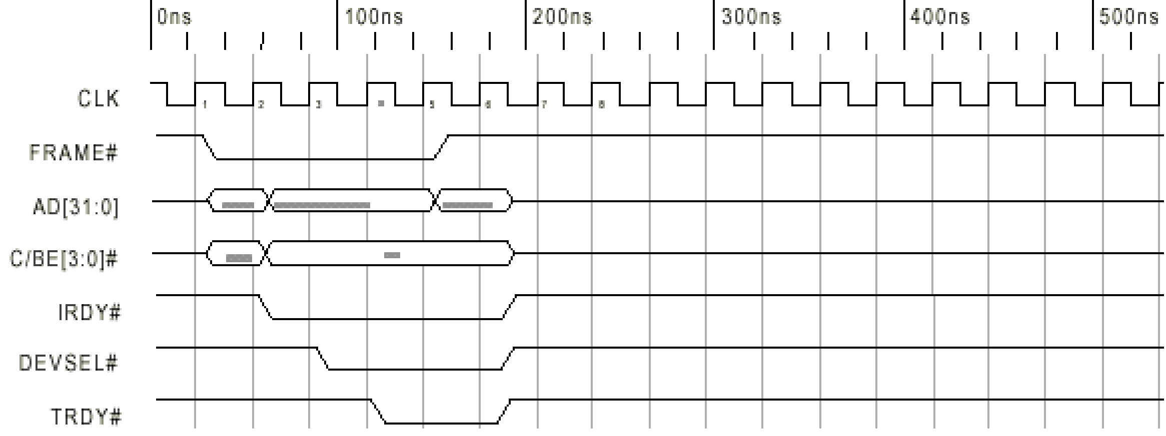


Figura 5.48: Ciclo de escritura en modo ráfaga, suspendida por la señal BREQi de 32 bits del Bus Local.

4.6.22 Ciclo de transferencia BIGEND en el Bus Local con la entrada BIGEND#.



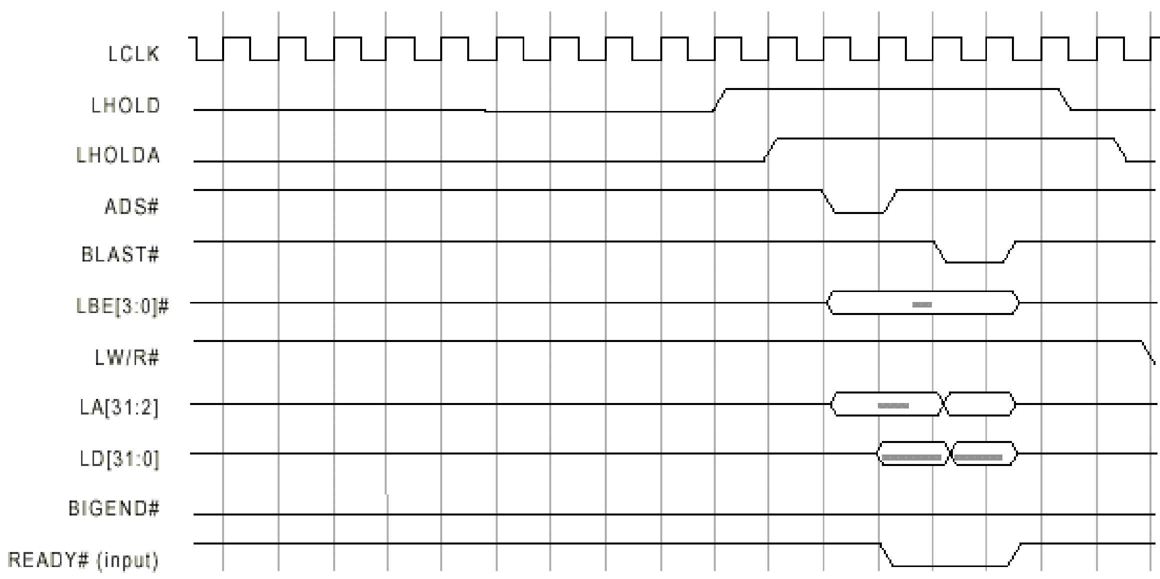


Figura 5.49: Ciclo de transferencia BIGEND en el Bus Local con la entrada BIGEND#.

5.6.23 Transferencia DMA desde PCI a dirección local BTERM y BURST activados

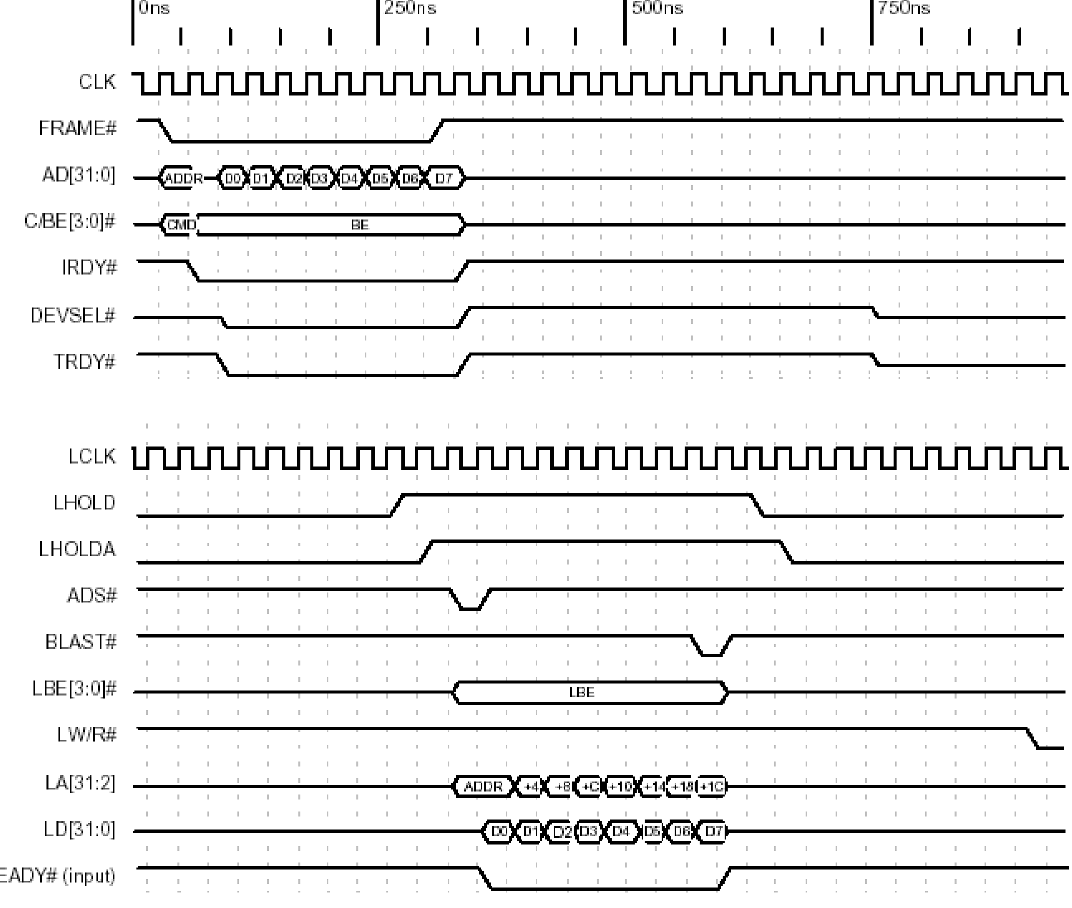


Figura 5.50: Transferencia DMA desde PCI a dirección local BTERM y BURST activados

4.6.24 Transferencia DMA desde PCI a dirección local BTERM desactivado y BURST activado

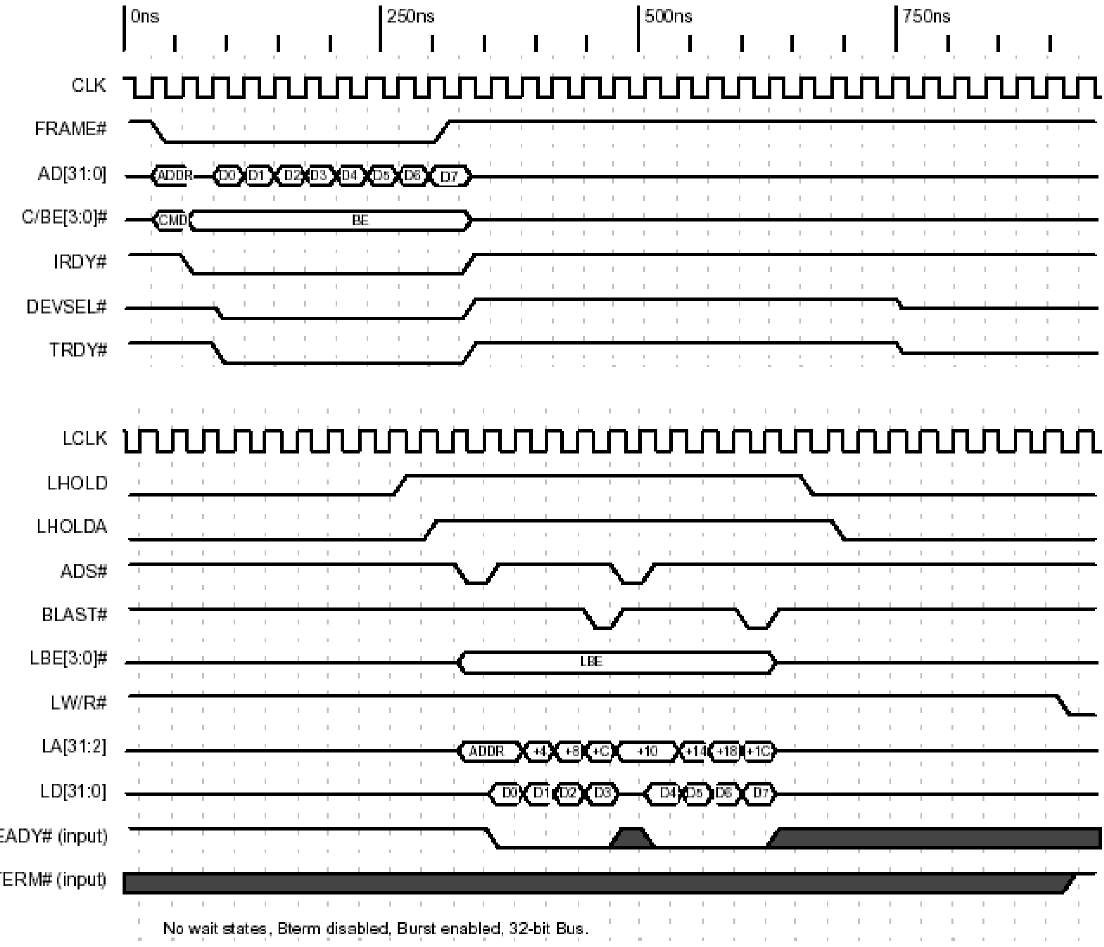


Figura 5.51: Transferencia DMA desde PCI a dirección local BTERM desactivado y BURST activado

4.6.25 DMA Scatter/gatter con descriptor en el bus local

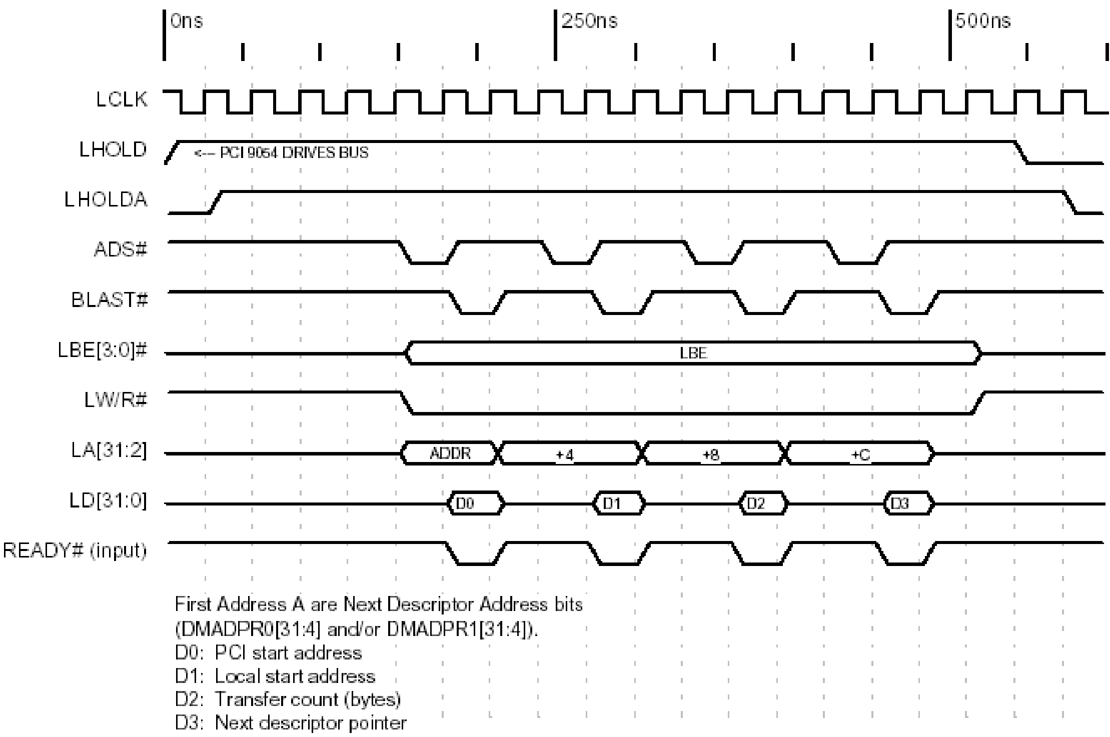


Figura 5.52: DMA Scatter/gatter con descriptor en el bus local

En este caso se puede ver como los datos del descriptor se van enviando a ráfagas en los momentos en los que la señal READY# indica el envío de los mismos, enviándose la dirección PCI de inicio de la transferencia, la dirección local, el tamaño de la transferencia y un puntero al siguiente descriptor.

5.6.26 DMA Scatter/gatter del bus local al PCI con descriptor en el bus PCI

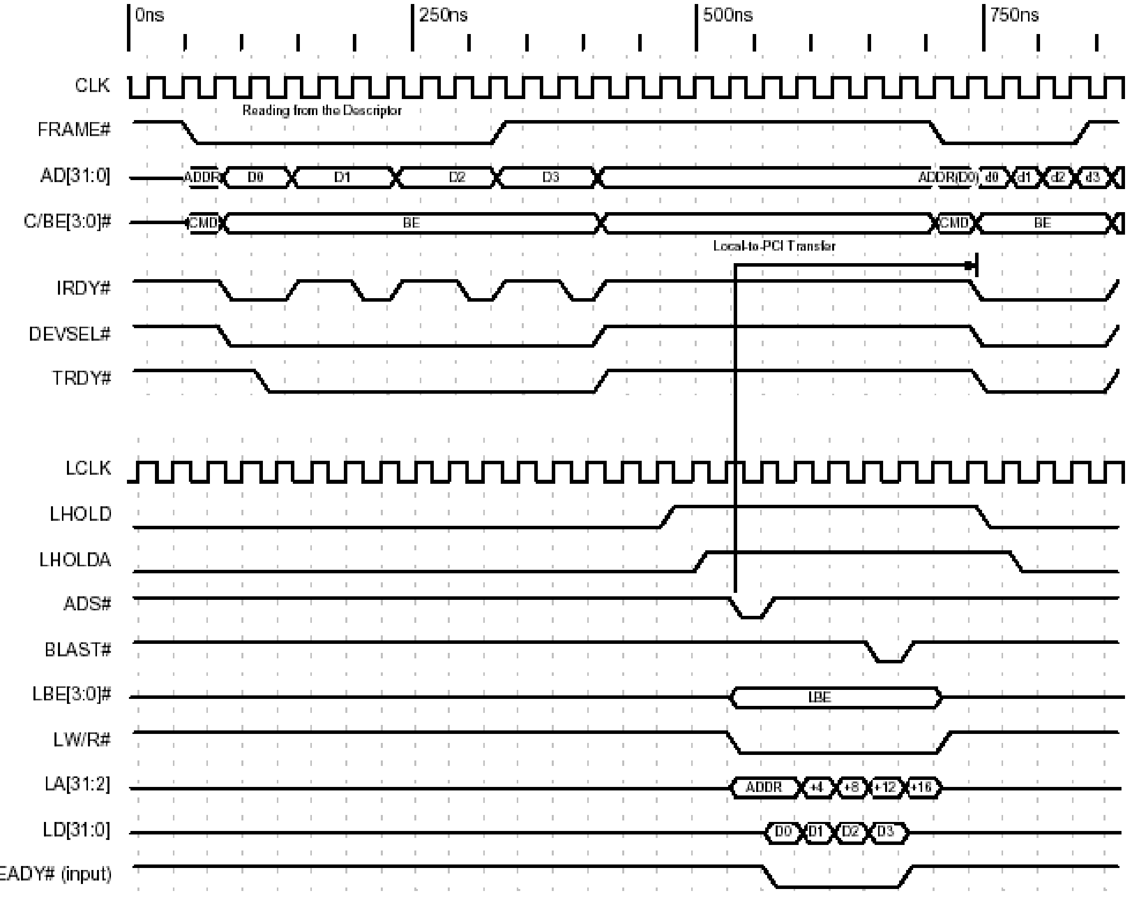


Figura 5.53: DMA Scatter/gatter con descriptor en el bus PCI

5.6.27 DMA modo demanda del bus local al PCI terminada con BLAST#

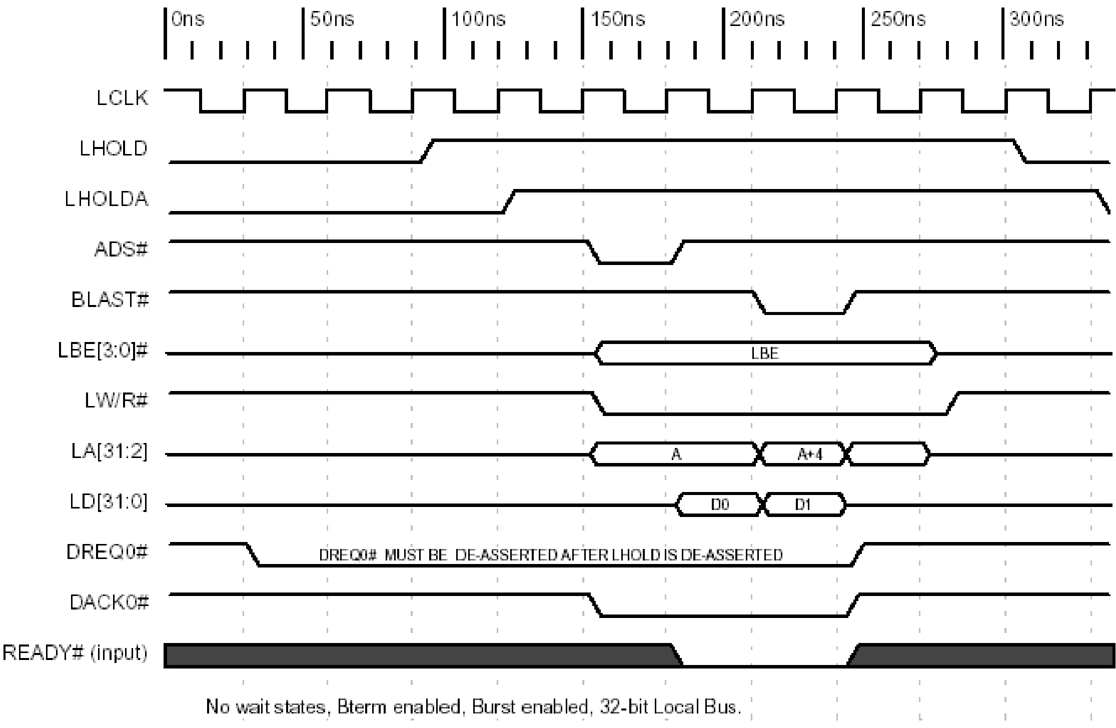


Figura 5.54: DMA modo demandadle bus local al PCI terminada con BLAST#

En este caso se realiza una transmisión de datos desde el bus local al PCI, con lo que en primer lugar la señal LW/R# debe estar en modo escritura, es decir, a cero, entonces se genera una petición del bus PCI generando una interrupción mediante la señal DREQ0#, entonces LHOLD# se activa, seguida de las demás señales que realizan la transferencia. En este caso para terminar la señal BLAST# (Burst last) indica que la transferencia ha concluido y se libera el bus.

5.6.28 DMA del bus local al PCI terminada con EOT#

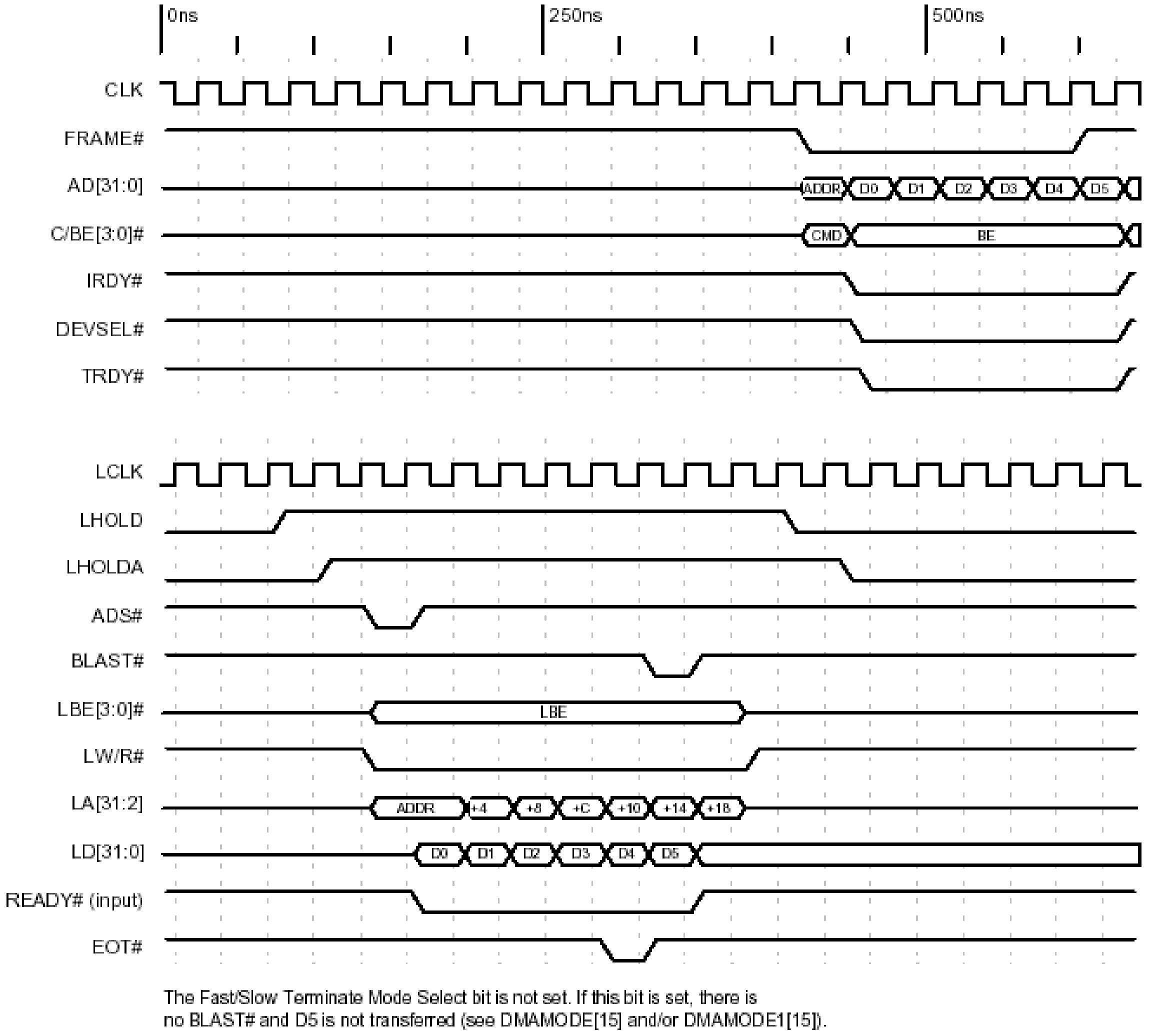


Figura 5.55: DMA modo demandadle bus local al PCI terminada con EOT#

Este caso es similar al anterior sólo que la señal BLAST# se genera al siguiente pulso de reloj que EOT# sin necesidad de que se haya terminado la transferencia.

5.6.29 DMA del bus PCI al local terminada con EOT#

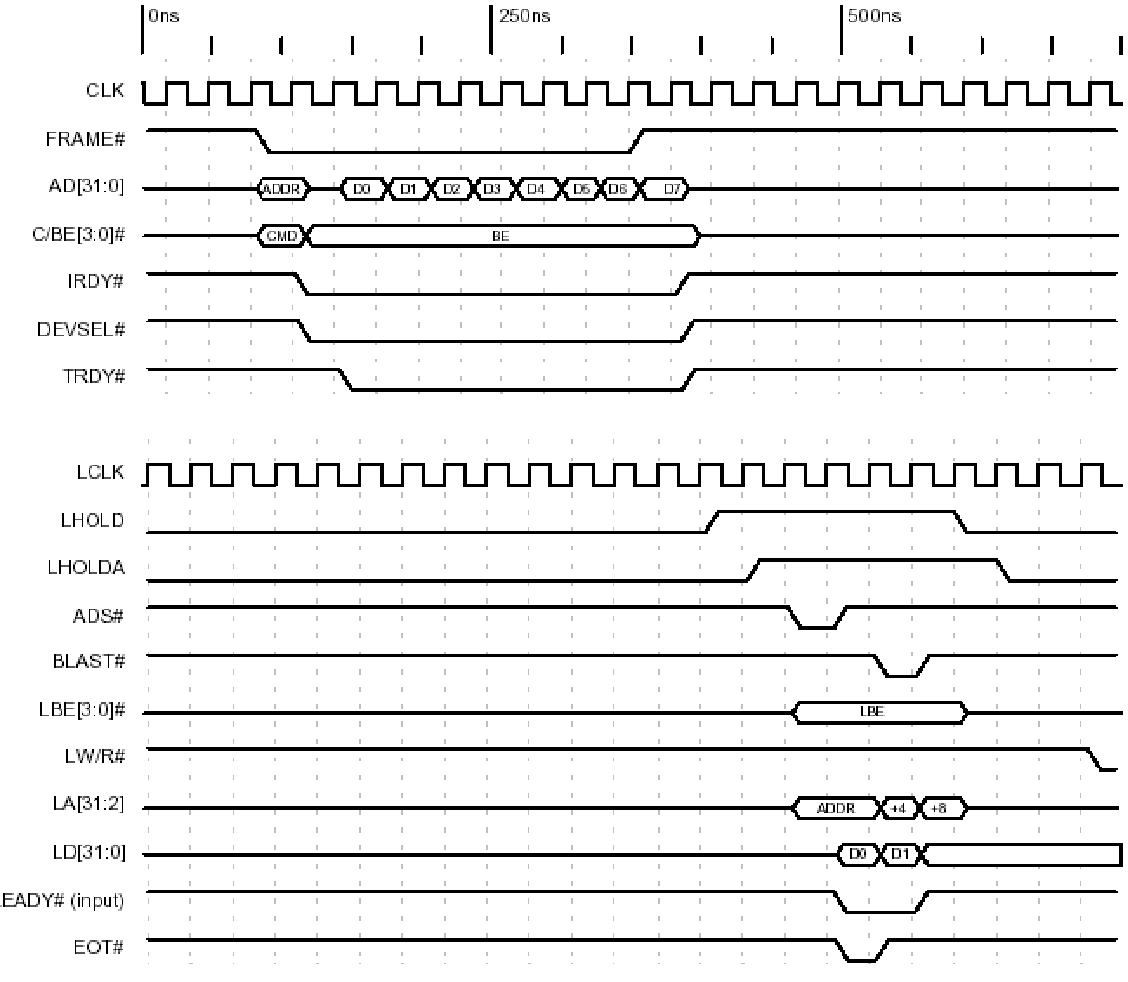


Figura 5.55: DMA modo demanda del PCI al localterminada con EOT#

En este caso la señal LW/R# a uno indica que el bus local realizará una lectura del PCI, por lo demás el comportamiento será similar al caso anterior.

5.6.30 DMA del puerto PCI al local con pausa y temporizadores de espera

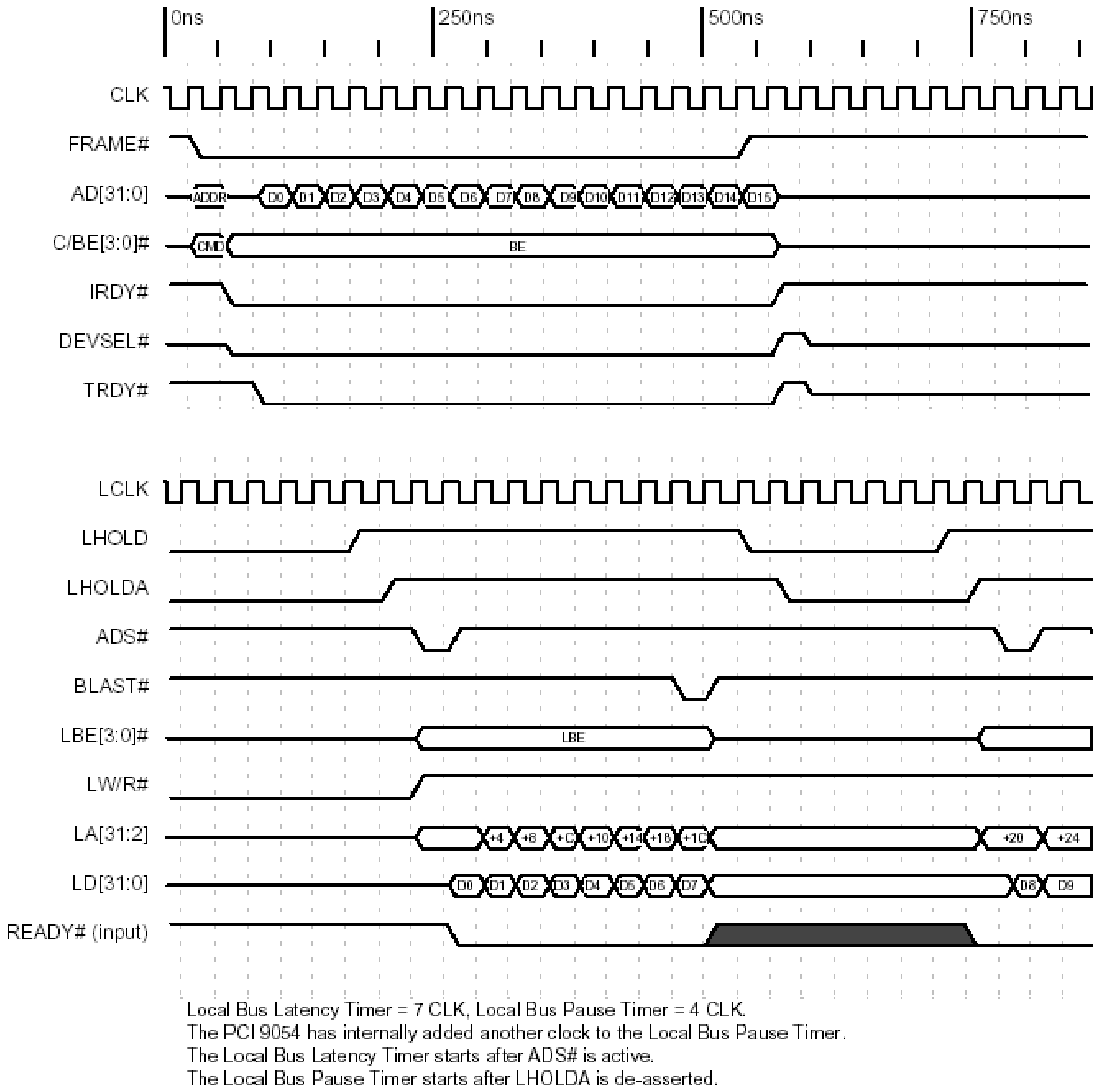


Figura 4.57: DMA del puerto PCI al local con temporizadores de espera

Aquí la señal BLAST# se genera debido a la que en los registros se ha programado una pausa, al final de la misma se volverá a generar ADS# para que continúe la transferencia

5.6.31 Ciclo singular DMA modo demanda PCI-local

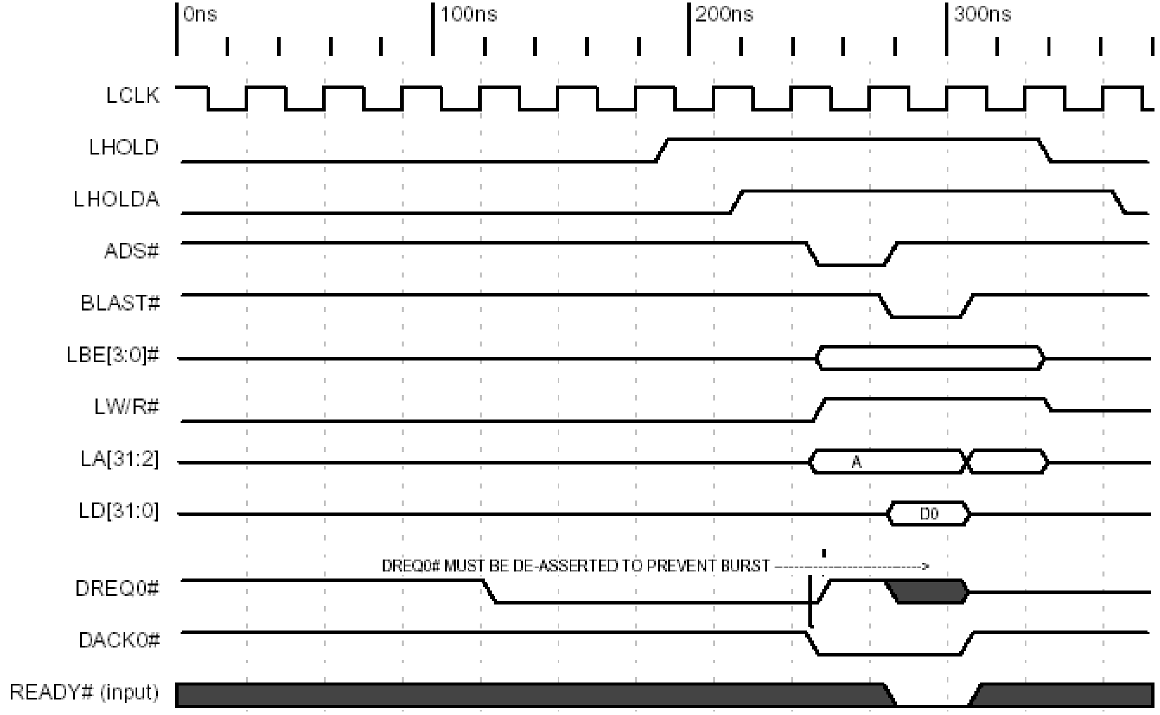


Figura 5.58: Ciclo singular DMA modo demanda del puerto PCI al local

Modo demanda DMA múltiples ciclos del bus PCI al local

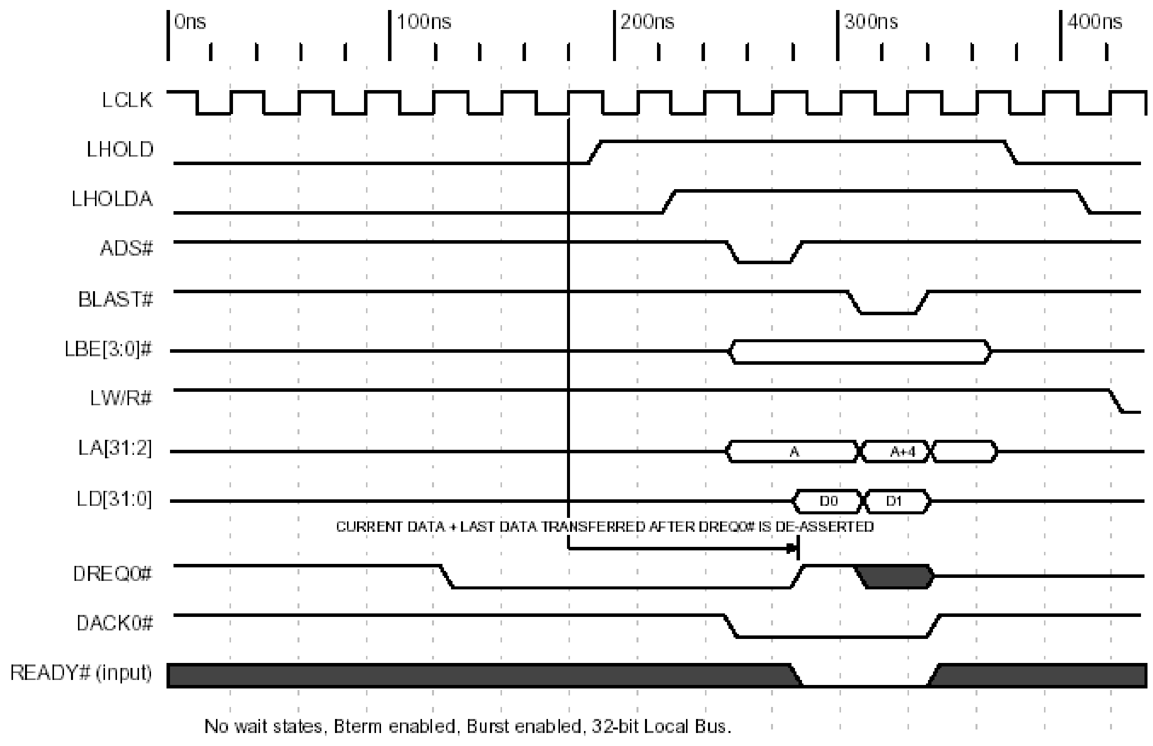


Figura 5.59: Modo demanda DMA de múltiples ciclos del bus PCI al local

4.7 Descripción de la transferencia PCI Target en el modo funcional C.

El PCI 9054 puede ser programada para trabajar en modo C mediante los bits de modo de funcionamiento del controlador, conectando el Bus PCI al Bus Local.
En este modo, el PCI 9054 esta diseñado con un interfaz realizado de forma que pueda conectarse con el Intel 960 o el IBM PPC 401 por el Bus Local, pudiendo conectar otro tipo de sistema como en nuestro caso, que conectaremos el sistema formado por la FPGA y la memoria SRAM.

En el modo PCI Target, el bus PCI como maestro, accede a la memoria local o al puerto I/O, actuando el bus PCI como maestro y la CPU local(1) como esclavo.

El PCI 9054 puede ser programada para trabajar en modo C mediante los bits de modo de funcionamiento del controlador, conectando el Bus PCI al Bus Local.
En nuestro caso este es el modo que esta configurado en la tarjeta es el modo C.

Pin 157	Pin 156	Modo	Descripción
1	1	M	32 bits no multiplexados
1	0	Reservado	-
0	1	J	32 bits multiplexados
0	0	C	32 bits no multiplexados

Tabla 4.9: Modos del Bus Local.

Las operaciones funcionales descritas, pueden ser modificadas a través de los registros internos del controlador.

4.7.1 Operación Reset.

4.7.1.1 Entrada RST# del Bus Local.

El pin de entrada RST# al bus PCI es un reset síncrono del Bus PCI, lo que causa que todas las salidas del Bus PCI se pongan en estado flotante, se reseté la entrada del controlador de bus y cause un reset local LRESET#. Activaremos esta señal cada vez que iniciemos nuestro PC.

4.7.1.2 Reset por software.

También se podrá realizar la operación de reset mediante el bit PCI Adapter Software Reset (CNTRL [30]=1) para resetear el PCI 9054 y activar la salida LRESET#. Todos los registros locales de configuración serán reseteados, aunque la configuración DMA del PCI 9054, los registros compartidos en tiempo de ejecución y el bit de estado de inicialización local (LMISC[2]) no se resetearán.
Cuando este bit se activa (CNTRL [30]=1), el controlador de bus responde a los accesos PCI, pero no a los accesos del bus Local.

(1) Sistema conectado al Bus Local.

4.7.2 Inicialización del PCI 9054.

Los registros de configuración del controlador de Bus PCI 9054, pueden ser programados por una EEPROM serie opcional y/o por un procesador local.

En nuestro caso una EEPROM serie está instalada en la tarjeta configurando el controlador cada vez que apliquemos tensión a la tarjeta, es decir, cada vez que encendamos el PC.

Procesador Local	EEPROM serie	Condición del sistema de arranque.
Ninguno	Ninguno	El PCI 9054 usa los valores por defecto. El pin EEDI/ EEDO debe ser puesto en bajo. Si el PCI 9054 detecta todos ‘0’ devuelve los valores por defecto.
Ninguno	Programada	Arranca con los valores de la EEPROM serie. El bit de estado local Init (LMISC[2]) debe ser puesto por la EEPROM.
Presente	Ninguno	El procesador local programa los registros internos del PCI 9054, y luego activa el bit de estado local Init (LMISC[2]=done)
Presente	Programada	Carga la serie EEPROM, pero el procesador local puede volver a reprogramar el PCI 9054. Uno de los dos debe poner el bit de estado local Init (LMISC[2])
Presente	Presente pero no programada	El PCI 9054 detecta la EEPROM serie en blanco y devuelve los valores por defecto. La EEPROM serie puede ser programada después de la inicialización.

Tabla 4.10: directiva de la EEPROM serie.

4.7.3 Operación PCI Target.

En este, el bus PCI actuando como maestro, accede a la memoria local o al puerto I/O.

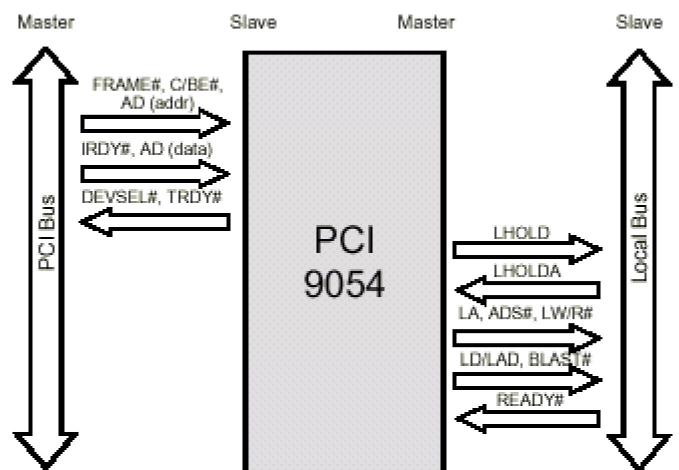


Figura 4.44: Ciclo de escritura PCI Target.

El Bus PCI, como maestro lee desde el Bus Local y escribe en el Bus Local.

Las características más importantes de este modo de transferencia son las siguientes:

- Múltiples espacios de dirección independientes.
- Control dinámico del ancho del Bus Local.
- Pretransferencia de lectura.
- Conversión Big/Little Endian.
- Control de prioridad del Bus Local.
- Temporizador de la latencia PCI.

El PCI 9054 soporta diferentes modos de transferencia, tales como transferencia de acceso modo ráfaga, a memoria mapeada y a I/O mapeado, transferencia de acceso singular al Bus Local desde el Bus PCI con lectura de 16 Lword (64 bytes) en la FIFO de lectura y con escritura de 32 Lword (128 bytes) en la FIFO de escritura.

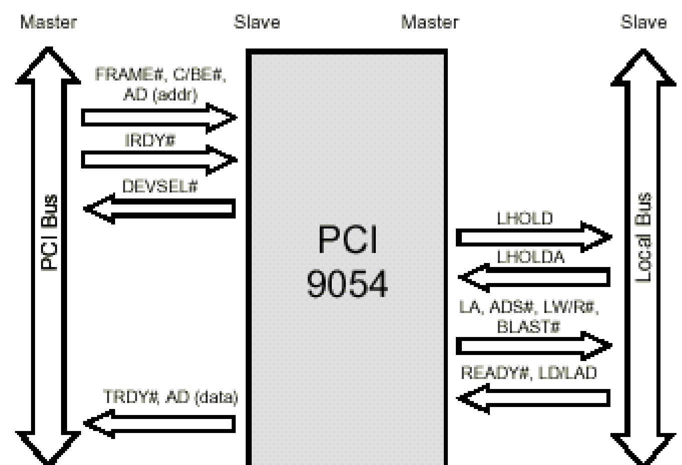


Figura 4.45: Ciclo de lectura PCI Target.

Los registros de dirección base PCI, están previstos para configurar la localización del dispositivo en la memoria PCI y el espacio I/O.

Se permiten 3 espacios; Espacio 0, Espacio 1 y extensión ROM.

Para un ciclo de lectura singular, el PCI 9054 lee un Lword singular o una parte del Lword en el Bus Local.

El PCI 9054 se desconecta tras una transferencia de todos los accesos de I/O.

En la más alta tasa de transferencia de datos, el PCI 9054 soporta post –escrituras, y puede ser programado para preescribir durante un ciclo de lectura tipo ráfaga. El tamaño de pre-transferencia, esta comprendido entre 0 y 16 Lwords, o hasta que el Bus PCI desactiva la solicitud.

En modo pretransferencia continuo, el controlador pretransfiere tanto espacio de la FIFO como sea permitido y detiene la pretransferencia cuando el bus PCI termina la solicitud del Bus.

Por otro lado si la pre-lectura esta desactivada, el controlador PCI 9054 se desconecta tras realizar un ciclo de lectura singular. Además, en el modo pretransferencia, el PCI 9054 soporta el modo Read Ahead.
Cada espacio Local puede ser programado para operar en un ancho de 8, 16 o 32 bits de Bus Local.
El PCI 9054 tiene un generador interno de estados de Wait y una entrada externa de estado de espera READY#, que puede estar habilitada o no, mediante los registros de configuración internos.
Con o sin estados de espera, el Bus Local independientemente del bus PCI puede:

- Rafagear tantos datos como precise (Modo Ráfaga continuo).
- Rafagear 4-Lwords por ciclo (recomendado).
- Realizar ciclos singulares continuos.

4.7.3.1 Retraso en Modo lectura PCI v2.1.

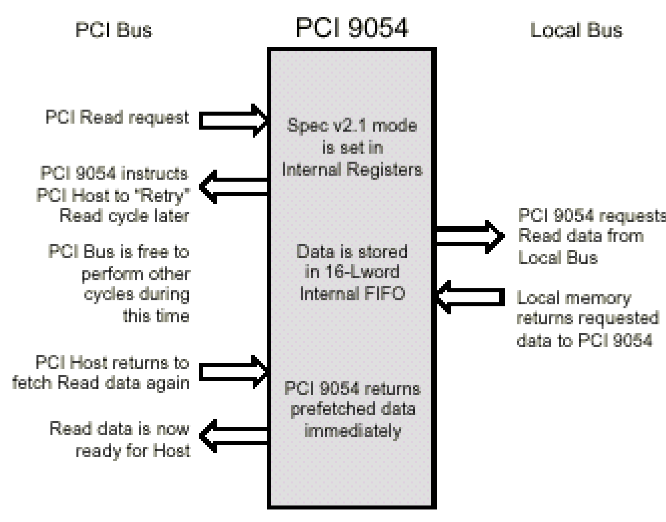


Figura 4.46: Retrasos de Lectura PCI v2.1

El controlador PCI 9054 puede ser programado mediante el bit MARBR[24]=1, para realizar retrasos de lectura o escritura tal y como se especifica en la PCI Specification v 2.1.

Además, en ciclos en lectura retrasadas, el PCI 9054 soporta las siguientes especificaciones:

- No escribe mientras un ciclo de lectura se este realizando.
- Escribir y realizar las lecturas pendientes.

4.7.3.2 Modo Read Ahead PCI.

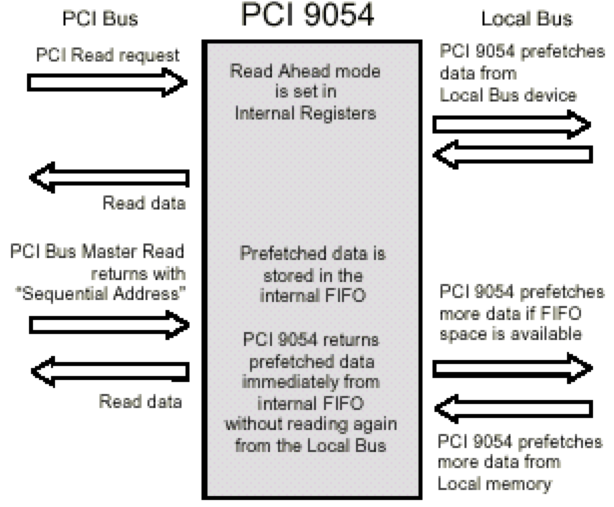


Figura 4.47: Transferencia Read Ahead.

El PCI 9054 soporta modo Read Ahead, donde los datos pretransferidos pueden ser leídos desde la FIFO interna del controlador en lugar de desde el Bus Local.
La dirección debe ser subsecuente a la dirección previa más 32 bits contiguos (siguiente dirección = dirección actual +4)
El modo Read Ahead funciona con o sin el modo de retraso de lectura PCI.

4.7.3.3 Transferencia.

Un Bus Maestro PCI direcciona el espacio seleccionado de memoria por el bus Local para iniciar las transferencias.
Hasta que se realice una transferencia de lectura o escritura, el controlador PCI 9054 actúa como maestro de bus Local y árbitro del mismo.
El controlador PCI 9054 lee datos en la FIFO de lectura del PCI Target o escribe datos en el Bus Local.
En un ciclo de escritura PCI Target, el Bus PCI escribe datos en el Bus Local. El PCI Target es el que manda en la transferencia, el cual tiene la más alta prioridad, de entre los tres modos de transferencia.
En un ciclo de lectura PCI Target, el Bus PCI Maestro lee datos desde el bus local esclavo.

4.7.3.4 Inicialización del Bus Local.

Rango: Especifica que bits de dirección PCI decodificar en un acceso PCI para direccionar el espacio del Bus Local. Cada bit corresponde al bit de dirección PCI, es decir, el bit 31 corresponde al bit de dirección 31.
Se escribe un ‘1’ en todos los bits que deban ser incluidos en la decodificación y ‘0’ en los demás.
Remapeo de direcciones PCI- Local, en un espacio de direcciones local: Los bits de este registro colocan los bits de dirección PCI usados para descodificarse como bits de dirección local.
Descriptor de región de Bus Local: Especifica las características de Bus Local.

4.7.3.5 Inicialización PCI.

Tras realizar un reset PCI, el software determina cuánto espacio de dirección se requiere, escribiendo 1´s en todos los registros de dirección base PCI, para luego volver a leer su valor.
El controlador, devuelve ceros si no son los bits de dirección requeridos, especificando el espacio de dirección necesario. Tras esto, el software PCI, mapea el espacio de dirección en el espacio de dirección PCI programando el registro de dirección base PCI. Esto puede ser visto en la figura 4.48.

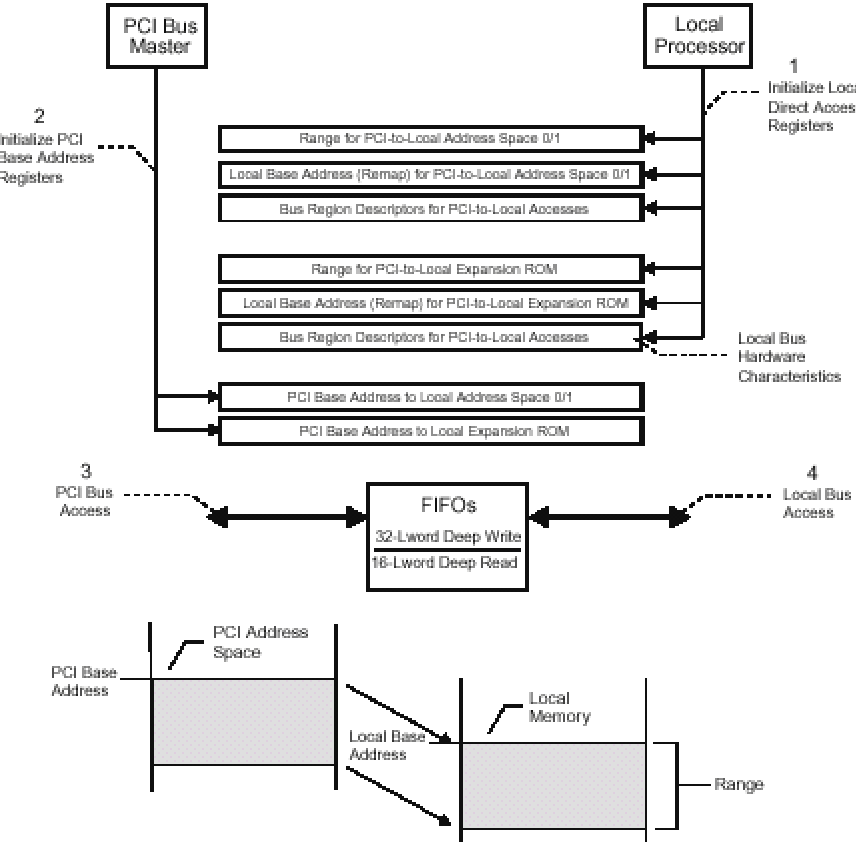


Figura 4.48: Acceso PCI Target al Bus Local.

4.7.3.6 Habilitación de los Bytes de datos (LBE[3:0]#)

Durante la transferencia, cada uno de los tres espacios (espacio 0, espacio1, y espacio de expansión ROM) puede ser programado para operar en un ancho de Bus Local de 8, 16 o 32 bits, codificando la señal LBE[3:0]#.
LBE[3:0]#, se codifica, según el ancho de Bus Local de la siguiente manera:

Bus de 32 bits: Los 4 bits activos indican cuál de los 4 bytes están activos durante un ciclo de datos.
BE3# (byte Enable 3)………………… LD[31:24]
BE2# (byte Enable 2)………………… LD[23:16]
BE1# (byte Enable 1)………………… LD[15:8]
BE0# (byte Enable 0)………………… LD[7:0]

Bus de 16 bits: LBE3#, LBE1# y LBE0# se codifican para proveer el byte alto, LA1 y el byte bajo como sigue:
BE3# (byte Enable 3)………………… Byte alto habilitado LD[15:8]
BE2# (byte Enable 2)………………… no usado
BE1# (byte Enable 1)………………… bit de dirección LA1
BE0# (byte Enable 0)………………… Byte bajo habilitado LD[7:0]

Bus de 8 bits: LBE1# y LBE0# se codifican para proveer LA1 y LA0:
BE3# (byte Enable 3)………………… no usado
BE2# (byte Enable 2)………………… no usado
BE1# (byte Enable 1)………………… bit de dirección LA1
BE0# (byte Enable 0)………………… bit de dirección LA0

4.7.3.7 Ejemplo.

Sea un espacio de 1MB de dirección local (1230 0000h hasta 123F FFFFh) accesible desde el Bus PCI en la dirección PCI desde (7890 0000h hasta 789F FFFFh).

a. El software de inicialización local pone Range y los registros de dirección local base como sigue:

a.1. Rango: FFF0 0000h (1 MB, codifica los 12 bits más altos de la dirección PCI).

a.2. Remapeo de la dirección base local: 123X XXXXh (dirección base local para acceso de bus PCI a bus Local) debe ser puesto para ser reconocido por el PCI Host (LASOBA[0]=1, LAS1BA[0]=1).

b. El software de inicialización PCI, escribe todo ‘1s’ a la dirección base PCI para luego volver a leer.

b.1. El controlador PCI 9054 devuelve el valor de FFF0 0000h El software PCI escribe al registro de dirección base PCI.

b.2. Dirección Base PCI: 789X XXXXh (dirección base PCI para acceso a los registros de espacio de dirección local (PCIBAR2 y PCIBAR3).

Para un acceso directo PCI al Bus Local, el PCI 9054 tiene una FIFO de escritura de 32 Lword (128 byte) y una FIFO de lectura de 16 Lword (64 bytes). Las FIFOs activan el Bus Local para operar

independientemente del Bus PCI. El controlador puede ser programado para devolver una respuesta Retry o para estrangular TRDY# para cualquier intento de transferencia del Bus PCI para escribir en el Bus Local del PCI 9054 cuando la FIFO este llena.
Para transferencias de lectura PCI desde el Bus Local el PCI 9054 mantiene desactivo TRDY# mientras captura datos desde el Bus Local.
Para accesos de lectura mapeados en el espacio de memoria PCI, el PCI 9054 pre transfiere hasta 16 Lwords (en modo ráfaga continua) desde el Bus Local. Los datos de lectura no usados son limpiados desde la FIFO.
Para accesos de lectura mapeados en el espacio de I/O PCI, el controlador PCI 9054 no pretransfiere datos de lectura. Tal vez, esto rompa cada lectura de un ciclo ráfaga en un ciclo de dirección /datos en el Bus Local.

4.7.3.8 Prioridad.

Los accesos del PCI Target tienen más alta prioridad que los accesos DMA, de este modo asegura la transferencia DMA.

Durante una transferencia DMA, si el controlador detecta un acceso PCI Target pendiente, el PCI 9054 libera el Bus Local en dos transferencias de datos Lword. El PCI 9054 resume la operación anterior, después de que se haya completado el acceso PCI Target.

Cuando el controlador PCI 9054 DMA posee el Bus Local, su salida LHOLD y entrada LHOLDA se activan, y cuando un acceso PCI Target ocurre, el PCI 9054 libera el Bus Local en 2 transferencias Lword desactivando LHOLD y dejando flotantes las salidas del bus Local.

Después del conocimiento de que LHOLDA está desactivo, pide el Bus Local para una transferencia PCI Target, activando LHOLD. Cuando el controlador de bus recibe LHOLDA, conduce el bus realizando una transferencia PCI Target. Una vez completada la transferencia PCI Target, el PCI 9054 libera el bus Local desactivando LHOLD y dejando flotantes las salidas del Bus Local.

Después de que el controlador detecte la desactivación de LHOLDA y el Temporizador de Pausa del Bus Local se ponga a cero, requiere una transferencia DMA desde el Bus Local, desactivando LHOLD. Cuando recibe LHOLDA, coge el bus y continua la transferencia DMA.

4.8 Señales.

4.8.1 Bus Local.

Símbolo	Señal	Función	Sentido
GND			
LCLK	Local Processor Clock	Entrada de reloj Local.	IN
+5V			
+3.3V			
GND			
/BTERM	Burst Terminate	Como entrada al PCI 9054: Para procesador que rafagean 4 Lword. Si el bit de modo Bterm esta deshabilitado, el PCI904 también transfiere en modo ráfaga a 4-Lword.Si esta habilitado, el PCI 9054 continua la transferencia hasta que la entrada BTERM# se active. BTERM# entrada que rompe transferencias en modo ráfaga y hace que otro ciclo de dirección ocurra. Como salida del PCI 9054: Activa, con la señal de READY#, para solicitar la ruptura de una ráfaga y empezar un nuevo ciclo de dirección.	IN/ OUT
/LBE[3:0]	Local Bus Enable	Codificado, para configurar el ancho del bus. **Bus de 32bits:** Los 4 bits activos indican cual de los cuatro bytes están activos durante un ciclo de datos: BE3# (byte Enable 3)--------------------- LD[31:24] BE2# (byte Enable 2)--------------------- LD[23:16] BE1# (byte Enable 1)--------------------- LD[15:8] BE0# (byte Enable 0)--------------------- LD[7:0] **Bus de 16 bits:** BE3# (byte Enable 3)--------------------- Byte alto habilitado LD[15:8] BE2# (byte Enable 2)--------------------- no usado BE1# (byte Enable 1)--------------------- bit de dirección LA1 BE0# (byte Enable 0)--------------------- Byte bajo habilitado LD[7:0] Bus de 8 bits: LBE1# y LBE0# se codifican para proveer LA1 y LA0: BE3# (byte Enable 3)--------------------- no usado BE2# (byte Enable 2)--------------------- no usado BE1# (byte Enable 1)--------------------- bit de dirección LA1 BE0# (byte Enable 0)--------------------- bit de dirección LA0	IN /OUT
DP[3:0]	Data Parity	Paridad de los 4 bytes en el Bus Local. La paridad se chequea en escrituras o lecturas del PCI 9054.	IN /OUT
/READY	Ready Input / Output	Cuando el PCI 9054 es un Bus Maestro, indica que la lectura del dato en el bus es valida, o que la transferencia de escritura se ha completado. Se usa en combinación del generador de estados internos de espera.	IN/OUT
LHOLD	Hold Request	Activado para pedir el uso del Bus Local. El árbitro del Bus Local activara la señal LHOLDA cuando el control se haya cedido.	OUT
GND			
LHOLDA	Hold Acknowledge	Activado por el árbitro del Bus Local cuando el control se ha cedido en respuesta a LHOLD. El bus no debe ser cedido a menos que sea requerido por LHOLD.	IN
/ADS	Address Strobe	Indica una dirección válida y comienzo de un nuevo acceso de bus. Se activa en el primer ciclo de reloj de un acceso de bus.	IN/OUT
/LW/R	Write/ Read	Alto para lectura. Bajo para escritura.	OUT

Tabla 4.11: Señales del Bus Local.

/LSERR	System Error Internal Output	Nivel síncrono de salida activo por el bit de PCI Target Abort o Received Master Abort. Si se requiere un flanco de nivel de interrupción, desactiva y luego activa LSERR#.	OUT
/BLAST	Burst Last	Conducido por el Bus Maestro(1) Local para indicar la ultima transferencia en un acceso de bus.	IN/OUT
BREQI	Burst Request	Activado para indicar que el Bus Maestro Local requiere el bus. Si esta habilitado a través de los registros de configuración, el PCI 9054 libera el bus durante la transferencia DMA si esta señal se activa.	IN
BREQO	Burst Request Out	Activado para indicar que el PCI 9054 requiere el bus para realizar una transferencia PCI Target Bus PCI al Bus Local, mientras un acceso PCI Initiator esta pendiente en el Bus Local.	OUT
/WAIT	Espera	Como entrada, se activa para causar que el PCI 9054 inserte estados de espera para accesos PCI Initiator al Bus PCI. Como salida, se activa por el PCI 9054 cuando el generador de estados interno causa un estado de espera.	IN/OUT
/LRESET	Local Bus Reset Out	Señal de reset del Bus Local, se activa cuando el chip PCI9054 se resetea. Puede ser usada para conducir la entrada RESET# de un procesador Local.	OUT
/ENUM	Enumeration	Interrupción de salida activada, cuando un adaptador que usa el controlador PCI9054 esta recién pinchado o listo para ser quitado, de la ranura PCI.	OUT
/LINT	Local Interrupt	Como entrada, cuando se activa, causa una interrupción PCI. Como salida, un nivel síncrono de salida que queda activo tantas interrupciones como existan. Si se requiere el flanco de interrupción, deshabilitando y luego habilitando la interrupción local, creando un flanco si una condición de interrupción todavía existe u ocurre una nueva condición de interrupción.	IN/OUT
/CCS	Configuration Register Select	Cuando se activa la señal CCS#, se seleccionan los registros internos PCI 9054.	IN
/U **/DRE** **/L**	USERo#: User output DREQ0#: Demand DMA Request LLOCKo#: Local lock Output	Pin multiplexado de entrada y salida. **USERo:** Salida de propósito general controlada desde los registros de configuración del PCI 9054.,user output **/DREQ0:** Cuando se programa un canal a través de los registros de configuración, para operar en modo Demanda, esta entrada sirve como una petición de DMA, DREQ0# corresponde al canal DMA Ch0 **/LLOCKo:** Indica una operación para PCI Target El acceso Bus PCI al Bus Local, puede requerir múltiples transferencias para ser completado.	IN/OUT
/DMP **/EOT**	PCI Initiator Programmable Almost Full End Of Transfer para el canal actual DMA	Pin multiplexado de entrada y salida. DMP : Estado de salida de Casi lleno del FIFO de escritura en una transferencia PCI Initiator. Programable a través de los registros internos de configuración. EOT# :Termina la transferencia actual DMA	IN/OUT
/U **/DAC** **/L**	USERi#: User input DACK0#: Demand Mode DMA Acknowledge. LLOCKi#: Local lock Input.	**/USERi:** Entrada de propósito general que puede ser leida por medio de los registros del PCI9054. **/DACK0:** Cuando un canal es programado a través de los registros de configuración para operar en Demand Mode. Esta salida indica que una transferencia DMA se esta ejecutando. /DAC0 corresponde a PCI9054 DMA ch 0. **/LLOCK I**ndica una operación que puede requerir múltiples transacciones para completarse. Usado por El PCI 9054 para acceso directo local al Bus PCI.	IN/OUT
/BIGEND	Big Endian Select	Selección del modo Big Endian.	IN
LEDON /IN	LEDon/ LEDin	Encendido/ apagado del Led.	IN/OUT

Tabla 4.11: Señales del Bus Local (continuación).

[1] Initiating agent: Bus Maestro. Target agent: dispositivo seleccionado.

4.8.2 Bus PCI.

Símbolo	Señal	Función	Tipo
C/BE[3:0]	**Bus Command and Bytes Enables**	Todos ellos multiplexados en el mismo pin. Durante una fase de dirección, define la orden del bus. Durante una fase de datos, se usa como bytes activos si cualquier dispositivo en el bus se selecciona.	t/s[1]
DEVSEL#	**Device Select**	La activa el esclavo indicando al Maestro, que el dispositivo activado, tiene decodificada su dirección del acceso. Como una entrada, indica si cualquier dispositivo en el bus esta seleccionado	s /t/s[2]
/ENUM	**Enumeration**	Interrupción de salida activada, cuando un adaptador que usa el controlador PCI9054 esta recién pinchado o listo para ser quitado, de la ranura PCI.	Out[3]
FRAME#	**Cycle Frame**	Conducido por el Maestro actual para indicar el comienzo y duración de un acceso. FRAME# se activa para indicar que la transferencia de bus esta comenzando. Mientras FRAME# este activa, la transferencia de Datos continua. Cuando se desactiva, la transferencia esta en el último ciclo de datos.	s /t/s
GNT#	**Grant**	Activada por el árbitro, indica que el acceso al bus se le ha concedido al Maestro. Cada Maestro tiene su propio REQ# y GNT#.	t/s
IDSEL	**Initialization Device Select**	Es usado como un Chip Select durante las transacciones de escritura y lectura de configuración.	In[4]
INTA#	**Interrupt A**	Solicitud de interrupción PCI.	Out
IRDY#	**Initiator Ready**	Activada por el maestro, en un ciclo de escritura indica que hay un dato valido en el bus AD, en un ciclo de lectura, indica que el maestro esta preparado para recibir un dato. Puede ser usado junto con la señal TRDY#.	s /t/s
LOCK#	**Lock**	Indica una operación que puede requerir múltiples transacciones para completarse.	In/Out
PAR	**Parity**	Paridad entre AD[31:0] y C/BE[3:0]. Todos los Bus Maestros requieren la generación de paridad. PAR es estable y valido un ciclo de reloj después de la fase de dirección. Para las fases de datos, Par es estable y valido un ciclo de reloj después de que cada IRDY# se active en una transferencia de escritura o se active TRDY# en una transferencia de lectura. Una vez Par sea valido, se queda valido hasta un ciclo de reloj después de la actual fase de datos.	t/s
PCLK	**Clock**	Provee tiempo a todas las transacciones en el PCI y es una entrada para cada dispositivo PCI. Todas las señales se muestran en flanco descendente, menos las señales RST#, INTA#, INTB, INTC, INTD#.	In
PERR#	**Parity Error**	Informa de un error de la paridad de datos durante todas las transacciones PCI, excepto durante los ciclos especiales.	s/ t/
PME#	**Power Management Event**	Evento de interrupción.	Out
REQ#	**Request**	Indica al arbitro que este agente debe usar el bus Cada Maestro tiene su propio REQ# y GNT#.	t/s
RST#	**Reset**	Inicializa todos los registros, secuencias y señales.	In
SERR#	**System Error**	Informa de error de paridad en la dirección, en la de datos en un ciclo especial[5]) o cualquier otro error de sistema donde el resultado sea catastrófico.	Out
STOP#	**Stop**	Lo activa el esclavo para indicarle al maestro que la transición actual se acaba.	s /t/s
TRDY#	**Target Ready**	Activada por el esclavo. En una lectura indica que hay un dato valido en el bus AD. En una escritura que esta preparado para recibir un dato.	s /t/s

Tabla 4.12: Señales del Bus PCI.

[1] entrada/ salida tri-estado bidireccional.
[2] tri-estado mantenido
[3] salida totem-pole.
[4] entrada estándar.
[5] Sirve para enviar un mensaje simple con el mecanismo de broadcast(envió a todos) al PCI.

CAPÍTULO 6

La tarjeta XS40 de XESS.

Para realizar nuestra aplicación, necesitamos de un sistema hardware que realice la función de un periférico conectado al Bus Local de la tarjeta de prototipo de Bus PCI. Para ello, utilizaremos la tarjeta XS40, compuesta, entre otros por un dispositivo lógico programable (FPGA) con el que realizaremos la función esclava de las transferencias, así como un sencillo procesado de las transferencias realizadas.

También posee el µcontrolador 8031, una memoria SRAM de 32 Kbytes, puertos paralelo, VGA y PS/2, un led de 7 segmentos y un oscilador programable hasta 100MHz.

En este capítulo se referirá a la composición, estructura y su funcionamiento de la tarjeta. También se mostrarán las herramientas y pasos necesarios para realizar el diseño y la síntesis del hardware a implementar en la FPGA y en el µcontrolador.

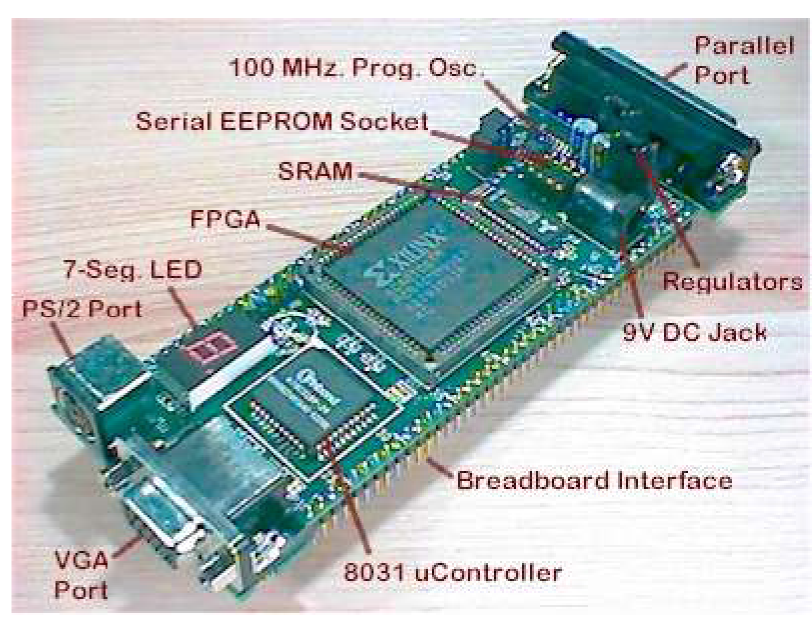


Figura 6.1: Tarjeta XESS.

6.1 ESTRUCTURA.

Los principales componentes de la tarjeta son los siguientes:

- FPGA de Xilinx Foundation
- Microcontrolador Intel 8031.
- Oscilador programable DS1075Z-100 de Dallas de 100 MHz.
- Memoria SRAM de 32 Kbytes Alliance AS7C256-15JC.
- Zócalo para EEPROM serie.
- Display de 7 segmentos.
- Puertos:
  - Paralelo.
  - PS/2.
  - VGA.
- Jumpers[(1)] de configuración.

Los principales componentes de esta tarjeta son la FPGA, el µcontrolador, la memoria SRAM y el oscilador programable. Para nuestra aplicación no usaremos el µcontrolador, el cuál será usado cuando queramos aumentar la complejidad de la aplicación. Todos estos componentes se encuentran conectados en la tarjeta, y podemos acceder a varios de ellos a través de los pines que se presentan al final.

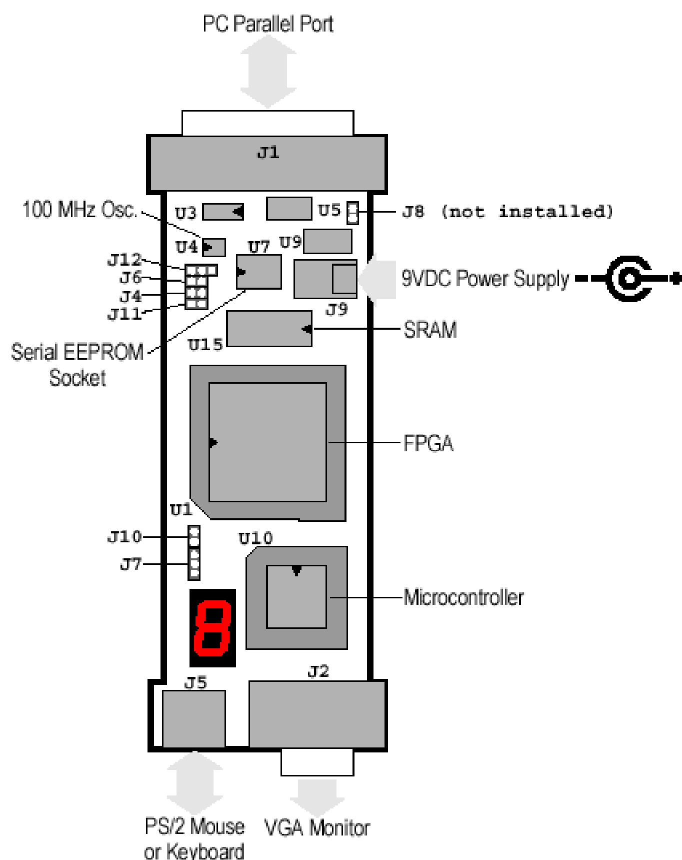


Figura 6.2: Componentes principales.

6.1.1 PUERTOS DE LA TARJETA XESS.

Puerto paralelo.

Para poder descargar un circuito en la FPGA deberemos conectar el PC con la tarjeta XS40, a través del puerto paralelo, pinchando el cable con el conector hembra DB-25 (J1) (Ver figura 6.1). En el caso de estar utilizando la memoria SRAM, tendremos que desconectar este cable, ya que ciertos pines utilizados para la aplicación, también están conectados al puerto paralelo.

Puerto VGA.

Este puerto puede ser usado para conectar un monitor VGA con la tarjeta XS40 pudiendo ver las imágenes descargadas en un monitor de VGA a través del conector de 15 contactos (J2) (véase la figura 6.2). Para ello, tendremos que descargar un circuito del programa piloto VGA en la tarjeta XS40 para visualizarla. Se puede encontrar un ejemplo para ello en http://www.xess.com.

(1) Puente

Puerto PS/2.

Mediante este puerto podemos pinchar un teclado o ratón para validar las entradas de información, conectándolo con el conector J5 PS/2 de la tarjeta XS40 (véase figura 6.2). Podemos encontrar un ejemplo para el teclado en http://www.xess.com.

6.1.2 JUMPERS DE CONFIGURACIÓN.

La tarjeta XS40 dispone de unos jumpers, que sirven para configurar el modo de operación de la misma. Esto se puede observar en la tabla 6.1. Las configuraciones que podemos cambiar mediante estos jumpers son:

1. Uso de la tarjeta XS40 en un modo independiente donde no este conectado al acceso paralelo del PC.
2. Reprogramación de la frecuencia de reloj en su tarjeta XS40.

3. Ejecución del código del microcontrolador de la ROM interna, en vez del SRAM externo en la tarjeta XS40. Para ello tendremos que sustituir el microcontrolador sin memoria ROM en la tarjeta XS40, por una versión con ROM.

Jumper	Estado	Función
J4	**On** (Por defecto)	Programación de la tarjeta a través del puerto paralelo.
	Off	Si la tarjeta se configura a través de una EEPROM serie (U7) montada en la tarjeta.
J5	On	Mientras se programa la EEPROM serie (U7).
	Off (Por defecto)	Uso normal de la tarjeta.
J7	**1-2 (Ext)** (Por defecto)	Si el programa del uC 8031 se guarda en la SRAM externa de 32 Kbytes(U8).
	2-3 (int)	Si el programa del uC 8031 se guarda internamente en él.
J8	**On**	Uso de la XC4000XL a + 3.3V.
	Off	Uso de la XC4000XL a + 5V.
J10	On	Cuando la tarjeta este siendo configurada desde la EEPROM serie (U7)
	Off (Por defecto)	Cuando la tarjeta este siendo configurada a través del puerto paralelo.
J11	**On (Por defecto)**	Si la tarjeta esta siendo configurada desde el puerto paralelo.
	Off	Si la tarjeta esta siendo configurada a través de la EEPROM serie (U7).
J12	**1-2(osc)(Por defecto)**	En un funcionamiento normal, cuando el oscilador programable este generando señales de reloj.
	2-3 (set)	Cuando se esté configurando la frecuencia del oscilador programable.

Tabla 6.1: Jumpers de configuración de la tarjeta XS40.Se muestra en negrita la configuración de los jumpers en un uso normal de nuestra aplicación.

6.2 FUNCIONAMIENTO.

6.2.1 ALIMENTACIÓN.

La tarjeta XS40 no se puede alimentar a través del cable de descarga del puerto paralelo del PC, por lo que requiere una fuente de alimentación externa para funcionar, esta puede ser aplicada de dos maneras diferentes:

- A través del conector de alimentación 9VDC, como muestra la figura 6.2, para ello deberemos de asegurarnos que el terminal de conexión del enchufe sea positivo y la parte externa negativa.

- Mediante los pines de alimentación y masa de la tarjeta.

FPFGA	Pin GND	Pin +5V	Pin +3.3V
XS40-010XL	52	2	54

Tabla 6.2: Contactos de la fuente de alimentación para las tarjetas XS40.

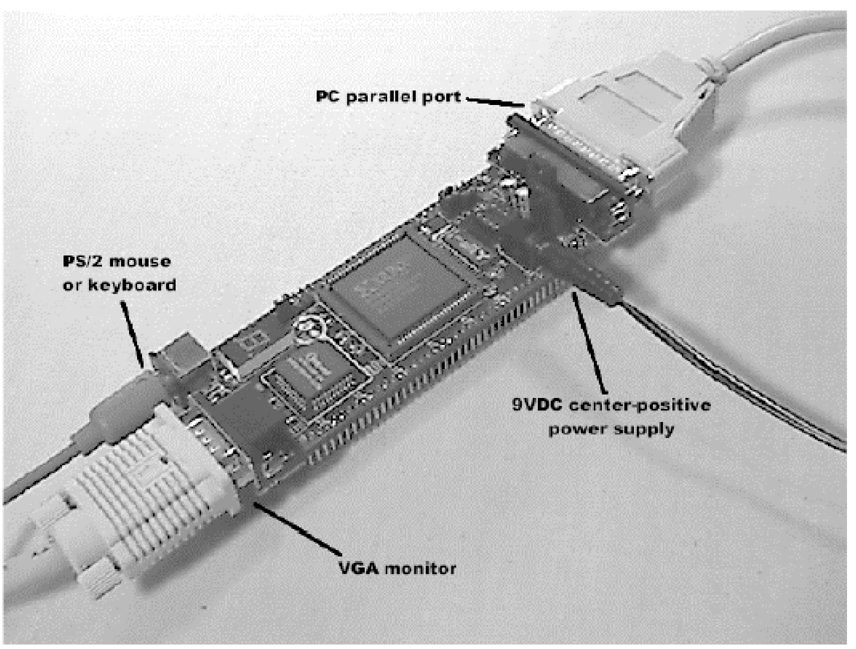


Figura 6.3: Alimentación de la tarjeta de aplicación mediante el conector de 9VDC

6.3 OSCILADOR PROGRAMABLE.

La versión V1.4 de la tarjeta XS40 utiliza un oscilador programable DS1075Z-100 de la compañía Dallas con frecuencia de valor por defecto de 50 MHz. Podremos reprogramarlo si deseamos utilizar otra frecuencia.

Su frecuencia principal de 100 megaciclos, se puede dividir por los factores desde 1, 2... hasta 2052 para conseguir frecuencias de reloj de 100 megaciclos, 50 megaciclos... hasta 48,7 KHz, respectivamente. La frecuencia dividida se envía a la FPGA como señal del reloj.

El divisor se salva en memoria no-volátil en el chip del oscilador, así que reasumirá su frecuencia programada, siempre que se aplique potencia a la tarjeta XS40, para ello deberemos seguir los siguientes pasos:

1. En una ventana de DOS, utilice el siguiente comando con la tarjeta XS40 y el divisor del reloj que usted desea (mencionado como argumentos).
   C: \ > XSSETCLK XS40-010XL 8
   El ejemplo mostrado fijará el oscilador programable de la tarjeta XS40-010XL a una frecuencia de 100 Megaciclos / 8 = 12,5 Megaciclos. Podemos utilizar cualquier divisor 1 y 2052 dependiendo de la frecuencia de reloj que deseemos utilizar.

2. El programa XSSETCLK nos llevará paso a paso a desconectar la alimentación y los cables de transferencia directa de la tarjeta XS40. Deberemos cambiar el jumper J12 y repinchar el cable de transferencia y el cable de alimentación. Cuando la potencia se restablece en la tarjeta XS40, el oscilador se accionará en su modo de programación, en vez de generar una señal del reloj.

3. Presionar RETURN y el divisor del reloj quedará programado en el chip del oscilador. Si se desea cambiar de nuevo el valor del divisor, podremos reeditar el comando de XSSETCLK con un nuevo divisor sin tener que apagar la tarjeta XS40.

4. Finalmente, desconectaremos la alimentación y el cable de *download* de la placa XS40, y volveremos a cambiar el puente J12. Vuelva a pinchar el cable de *download* y el cable de tensión. Cuando la potencia se restablece a la tarjeta XS40, el oscilador programable se accionará en su modo activo y hará salir una señal del reloj a la frecuencia programada.

6.4 PROGRAMACIÓN.

6.4.1 CHEQUEO DE LA TARJETA XS40.

Una vez que la tarjeta XS40 esté instalada y los jumpers estén en su configuración por defecto, podemos probar la tarjeta (véase apartado XSTOOLS).

Los programas de prueba de la FPGA, cargan en la SRAM un programa de test para el microcontrolador, para que este lo ejecute. El período total de prueba, incluyendo la programación de la tarjeta, es de 15 segundos para una tarjeta XS40. Si la prueba termina con éxito, entonces se visualizará un O en el display. Sin embargo, si el programa de prueba detecta un error, el display mostrará un E o un espacio en blanco. En este caso, controle los puntos siguientes:

1. Cerciórese de que la tarjeta XS40 esté recibiendo tensión de una fuente de potencia + 9V CC a través del conector J9 o a través de los contactos de VCC y de tierra.

2. Controle que la tarjeta XS40 esté sobre una superficie no conductora y que no haya conexiones a cualesquiera de los pines (a excepción de los contactos de VCC y de la tierra si éste es el modo con el que esta accionada la tarjeta).

3. Verifique que los jumpers estén en su configuración por defecto.

4. Cerciórese de que el cable de descarga esté bien pinchado en la tarjeta XS40 y en puerto paralelo del PC.

5. Verifique que el puerto paralelo esté en modo ECP. (el modo se fija generalmente en la BIOS como los SPP, EPP, ECP, o bidireccional[1]).

6. Si todos estos chequeos son positivos, chequear la tarjeta con otro PC. El 99,9% de todos los problemas se deben al acceso paralelo.

6.4.2 DESCARGA DE DISEÑOS EN XS40.

Durante las fases de desarrollo y prueba, conectaremos la tarjeta XS40 con el puerto paralelo de un PC y descargaremos un circuito en la FPGA cada vez que realicemos cambios en él. Para ello deberemos abrir el programa GXSLOAD que viene con el conjunto de herramientas GXSTOOLS de la tarjeta XS-40. Es un programa con una interfaz grafica muy sencilla, sólo hace falta arrastrar y soltar los diseños a programas (archivos .bit creados con el conjunto de herramientas de Xilinx) y pulsar el botón Load para descargar el diseño en la FPGA.

También se pueden arrastras diseños .hex para la SRAM de la tarjeta de la misma manera que si fuesen .bit.

6.4.3 SALVAR DISEÑOS PERMANENTES EN SU TARJETA XS40.

La FPGA XC4000 de XS40 salva su configuración en una chip SRAM que se borra siempre que se desconecte la alimentación, pero también podremos poner una EEPROM serie externa en el zócalo U7 para salvar la configuración de FPGA y que se cargue en un ciclo de inicio.

Podremos utilizar, por ejemplo, la serie de XILINX XC1700 de series EEPROMs, para esto necesitaremos un programador externo para descargar la cadena de bits en el chip XC1700. También el XC1700 es programable de una sola vez (OTP), así que necesitaremos un nuevo chip cada vez que cambiemos el diseño. El tamaño de la cadena de bits de esta EEPROM es de 283.424 bits.
Una vez cargado el diseño en la EEPROM, los siguientes pasos harán que la tarjeta XS40 se configure desde la EEPROM colocada en el zócalo U7 en vez de mediante el puerto paralelo del PC:

1. Quitar el cable de descarga del jumper J1 de la tarjeta XS40.[1]

2. Puentear el jumper J10. Esto fija la FPGA en modo serie activo proporcionando una señal del reloj a la EEPROM que ordena la carga de configuración de la EEPROM en la FPGA.

[1] El modo recomendado es ECP, que el modo bidireccional no resulta conveniente.
[1] Como alternativa, usted puede utilizar el comando **XSPORT 0** para cerciorarse de que los dos bits de datos superiores del puerto paralelo están en lógica 0. Estos dígitos binarios están conectados con los contactos de modo del FPGA y deben estar en la lógica 0 o el FPGA no iniciara el ciclo en el modo serie activo.

3. Quitar los jumpers J4 y J11. Para evitar que el trazado de circuito del interfaz del PC en la tarjeta XS40 interfiera con el reloj y con las señales de datos del FPGA.

4. Aplicar la alimentación de la tarjeta XS40. El FPGA será configurado desde la EEPROM serie. Se puede re-pinchar el cable que descarga si se necesita mandar señales de prueba en el diseño usando el programa de XSPORT.

6.4.4 LOS MODELOS DEL PROGRAMADOR.

En esta sección se discute la organización de componentes en la tarjeta XS40 se introducen los conceptos requeridos para crear las aplicaciones que utilizan el microcontrolador y el FPGA.

6.4.4.1 El microcontrolador + flujo del diseño de FPGA.

El flujo básico del diseño para las aplicaciones de uC + FPGA se muestra en la figura 6.4.

Deberemos configurar inicialmente el sistema que intentamos diseñar, determinando qué entradas están disponibles para el sistema y qué salidas generará. En este punto, tendremos que repartir las funciones del sistema entre el microcontrolador y el FPGA. Algunas de las señales de entrada irán al microcontrolador, algunas irán al FPGA, y otras a ambos.

Así mismo, algunas de las salidas serán computadas por el microcontrolador y algunas por el FPGA. También habrá algunas entradas y salidas del sistema creadas para la cooperación entre el microcontrolador y la FPGA. En general, la FPGA será utilizada principalmente para las funciones de bajo nivel, donde ocurren las transiciones de señal con más frecuencia y la lógica de control es más simple. Un transmisor - receptor serie, sería un buen ejemplo. Inversamente, el microcontrolador será utilizado para las funciones de alto nivel, donde ocurren las respuestas menos rápidas y la lógica de control es más compleja. Un procesado de las señales pasadas por el receptor sería un buen ejemplo, en nuestro caso al realizar un procesado poco complejo, lo realizaremos en la FPGA.

Una vez que el diseño se ha repartido y se han asignado las entradas, salidas, y funciones al microcontrolador y a la FPGA, comenzamos a hacer el diseño detallado del software lógico y hardware. Para el software, se puede utilizar cualquier editor para crear un fichero en ensamblador .ASM, ensamblarlo con ASM51 y crear un fichero HEX para el microcontrolador en la tarjeta XS40.

Para la porción del hardware de la FPGA, se incorporarán las tablas de verdad y las ecuaciones en un fichero ABL o VHDL y se compilará en un fichero de cadena de bits BIT usando el software de Xilinx Foundation. Como se describirá posteriormente podremos descargar el fichero HEX y BIT en la tarjeta XS40 usando el programa XSLOAD.

XSLOAD salva el contenido del fichero HEX en la memoria SRAM de la tarjeta XS40 y luego configura la FPGA cargándola con un fichero BIT. Una vez cargada la tarjeta con el hardware y el software, necesitaremos probarla para ver si trabaja correctamente.

La respuesta comienza generalmente como "no", así que necesitaremos un método para enviar señales de prueba y observar los resultados. XSPORT es un programa simple que le deja enviar señales de prueba al XS40 a través del puerto paralelo del PC. Podemos observar la reacción del sistema a las señales del puerto paralelo programando el microcontrolador y la FPGA para hacer

salir la información de estado sobre el dígito del LED, sería como poner declaraciones del " printf " en un programa en lenguaje C.

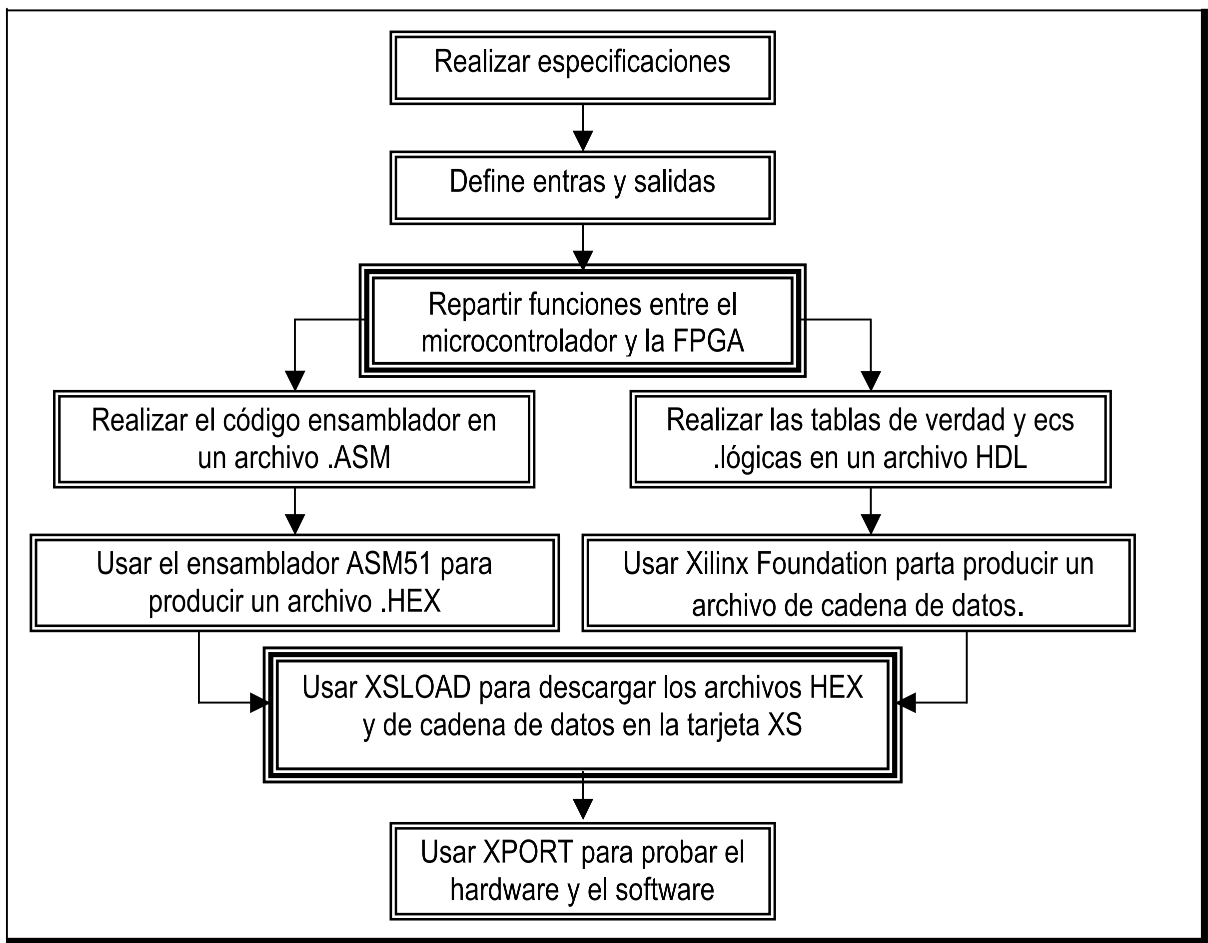


Figura 6.4: El microcontrolador + flujo del diseño de FPGA

6.4.4.2 Interconexiones en la tarjeta XS40.

El microcontrolador y la FPGA están conectadas juntos en la tarjeta XS40. Estas conexiones preexistentes quitan el esfuerzo de tener que unirlas, pero también imponen limitaciones, ante cómo obrarán recíprocamente el programa del microcontrolador y la dotación física de FPGA.

Un diagrama esquemático más detallado también se presenta en el final de este manual. La salida programable del oscilador va directamente a una entrada de reloj síncrona de la FPGA. La FPGA lo utiliza para generar una señal de reloj que envíe a la entrada de reloj XTAL1 del microcontrolador.

El microcontrolador multiplexa los ocho dígitos binarios más bajos de un direccionamiento de memoria con ocho dígitos binarios de datos y hace salir esto en su puerto P0.

Las líneas de datos de SRAM y FPGA están conectadas con P0, el SRAM utiliza esta conexión para enviar y recibir datos a y desde el microcontrolador. El FPGA se programa para latchear la dirección de salida en P0, bajo control de la señal ALE y enviar los bits de dirección latcheados a las ocho líneas de dirección más bajas del SRAM.

Mientras tanto, los ocho bits superiores de dirección se hacen salir por el puerto P2 del microcontrolador. Los 32 Kbytes SRAM en la tarjeta XS40 utilizan los siete bits más bajos de dirección. El FPGA también recibe los ocho bit superiores del dirección y los decodifica junto con el PSENB y la línea de lectura/ escritura de control (del pin P3.6 del puerto P3) del microcontrolador para generar las señales CEB y OEB que permiten el SRAM y sus drivers de salida, respectivamente. Las señales CEB o OEB se pueden poner altas para invalidar la SRAM y para evitar que tenga cualquier efecto en el resto del trazado de circuito de la tarjeta XS40.

Una de las salidas de la FPGA controla la línea de reset del microcontrolador. El microcontrolador se puede desactivar del circuito, forzando el pin RST a alto con la FPGA (cuando RST esta activo, los pins del microcontrolador se ponen arriba). Muchos de los contactos de entrada/ salida de los puertos P1 y P3 del microcontrolador, conectan con la FPGA y se pueden utilizar para la entrada/ salida de uso general entre el microcontrolador y la FPGA. Además de ser entrada-salida de uso general, el pin P3 también tiene funciones especiales tales como transmisores serie, receptores, entradas de interrupción, entradas del temporizador, y señales de control de lectura/ escritura externas de SRAM.

Si no se utiliza una función especial determinada, se puede utilizar el contacto asociado para la entrada-salida de uso general entre el microcontrolador y la FPGA. Aunque en muchos casos, se programará la FPGA para hacer uso de los pines de propósito especial del microcontrolador. (por ejemplo, el FPGA podría generar interrupciones del microcontrolador).

Si se desea utilizar el pin de propósito general desde un circuito externo, entonces el de entrada-salida de FPGA conectado con él debe ser tri-estado. Un dígito del siete-segmentos LED esta conectado directamente con l FPGA. (estos mismos contactos de FPGA pueden también ir a un monitor de VGA.)

El PC puede transmitir señales a la tarjeta XS40 a través de los ocho dígitos binarios de la salida de datos del puerto paralelo. El FPGA tiene acceso directa a estas señales. El microcontrolador puede también tener acceso a estas señales si se programa la FPGA para pasarlas sobre los pines I/O de la FPGA conectados con el microcontrolador.

La comunicación desde la tarjeta XS40 al PC también ocurre a través del puerto paralelo. Los pines de estado del puerto paralelo están conectados con los pines de los puertos P1 y P3 del microcontrolador, pudiendo el microcontrolador o la FPGA acceder a ellos. El PC puede leer los pines de estado para traer datos de la tarjeta XS40.

El FPGA también tiene acceso al reloj y a las líneas de datos de un teclado o de un ratón asociado al acceso PS/2 de la tarjeta.

6.4.4.3 Nuestra aplicación.

El uso que le daremos a la tarjeta XS40 para nuestra aplicación es el siguiente:
Durante la realización del sistema hardware prototipo, descargaremos el programa por el cable conectado al puerto paralelo del PC y al conector J1, para guardarlo en la FPGA. Este quedará almacenada mientras la tarjeta este alimentada

Una vez cargada la aplicación, tendremos que desconectar el cable para que no interfiera en los pines comunes utilizados del puesto paralelo.
Nuestra FPGA queda configurada para su uso, esta conectará con la memoria SRAM para almacenar la escritura de 4 Lword, que realizaremos desde el software realizado en Visual C++ en el PC, hasta su posterior lectura.

El reloj deberemos de configurarlo previamente con la frecuencia de funcionamiento deseado.
Podremos colocar una EEPROM en el zócalo U7 para dejar la tarjeta configurada cada vez que le apliquemos tensión, sin la necesidad de estar descargando el hardware en la FPGA, cada vez que queramos hacer uso de esta.

6.5 PINES.

Los pines de la tarjeta XESS corresponden con los pines del dispositivo FPGA, estos también están conectados a los demás componentes de la tarjeta de la forma que se muestra en la tabla 6.3.

Pin	Nombre	Descripción
1	-	-
2	+ 5 V	-
3	A0 (SRAM)	Dirección 0 de la SRAM.
4	A1 (SRAM)	Dirección 1 de la SRAM.
5	A2 (SRAM)	Dirección 2 de la SRAM.
6	P1.3 (mC)	Pin 3 del puerto 1 del uC.
7	P1.0 (mC)	Pin 0 del puerto 1 del uC.
8	P1.1(mC)	Pin 1 del puerto 1 del uC.
9	P1.2(mC)	Pin 2 del puerto 1 del uC.
10	P0.7 A7/D7(mC) **D7(SRAM)**	Pin 7 del puerto 0 del uC (acceso multiplexado de address/ data) **Dato 7 del SRAM.**
11	-	-
12	-	-
13	**CLK**	**Entrada de reloj programable (100MHz).**
14	/PSEN (mC)	Enable de almacenamiento del uC.
15	-	-
16	A16 (SRAM)**	No conectado.
17	-	-
18	**S5(LED)** RED1(VGA)	**Segmento 5 del Led** Color rojo1 del VGA.
19	**S6(LED)** HSYNC(VGA)	**Segmento 6 del Led** Señal sinc. horizontal del VGA.
20	**S3(LED)** GREEN1(VGA)	**Segmento 3 del Led** Color verde1 del VGA.
21	-	-
22	-	-
23	**S4(LED)** RED0(VGA)	**Segmento 4 del Led** Color rojo0 del VGA.
24	**S2(LED)** GREEN0(VGA)	**Segmento 2 del Led** Color verde0 del VGA.
25	**S0(LED)** BLUE0(VGA)	**Segmento 0 del Led** Color azul0 del VGA.
26	**S1(LED)** BLUE1(VGA)	**Segmento 1 del Led** Color azul1 del VGA.
27	P3.7(/RD) (mC)	Pin 7 del puerto 3 del uC(/RD)(lectura de datos).
28	P2.7(A15) (mC) A15 (SRAM)**	Pin 7 del puerto 2 del uC(address 15 del uC). Address 15 del SRAM(no conectada).

Tabla 6.3: Pines de la tarjeta XS40.

29	/ALE(mC)	Enable del latch de direccionamientodel uC.
30	-	-
31	-	-
32	PC_D6(PUERTO PARALELO DATA OUTPUT)*	Dato 6 salida puerto paralelo (señal de modo para el FPGA).
33	-	-
34	PC_D7(PUERTO PARALELO DATA OUTPUT)*	Dato 7 salida puerto paralelo (señal de modo para el FPGA).
35	P0.4(A4/D4) (mC) **D4(SRAM)**	Pin 4 del puerto 0 del uC (acceso multiplexado de address/ data). **Dato 4 del SRAM.**
36	RST	entrada de reset del uC.
37	XTAL1(mC)	Entrada de reloj del uC.
38	P0.3(A3/D3) (mC) **D3(SRAM)**	Pin 3 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 3 del SRAM.**
39	P0.2(A2/D2) (mC) **D2(SRAM)**	Pin 2 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 2 del SRAM.**
40	P0.1(A1/D1) (mC) **D1(SRAM)**	Pin 1 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 1 del SRAM.**
41	P0.0(AO/D0) (mC) **D0(SRAM)**	Pin 0 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 0 del SRAM.**
42	-	-
43	-	-
44	PC_D0(PUERTO PARALELO DATA OUTPUT)	Dato 0 salida puerto paralelo(señal de registro).
45	PC_D1(PUERTO PARALELO DATA OUTPUT)	Dato 1 salida puerto paralelo(señal de registro).
46	PC_D2(PUERTO PARALELO DATA OUTPUT)	Dato 2 salida puerto paralelo.
47	PC_D3(PUERTO PARALELO DATA OUTPUT)	Dato 3 salida puerto paralelo.
48	PC_D4(PUERTO PARALELO DATA OUTPUT)	Dato 4 salida puerto paralelo.
49	PC_D5(PUERTO PARALELO DATA OUTPUT)	Dato 5 salida puerto paralelo.
50	P2.4(A12) (mC) A12(SRAM)	Pin 4 del puerto 2 del uC (address12) Dirección 12 de la SRAM.
51	P2.2(A10) (mC) A10(SRAM)	Pin 2 del puerto 2 del uC (address10) Dirección 10 de la SRAM.
52	**GND(1.4V)**	-
53	-	-
54	**3.3V**	-
55	-	-
56	P2.3 (A11) (mC) A11(SRAM)	Pin 3 del puerto 2 del uC (address11) Dirección 11 de la SRAM.

Tabla 6.3: Pines de la tarjeta XS40.(Continuación).

57	P2.1 (A9) (mC) A9(SRAM)	Pin 1 del puerto 2 del uC (address9) Dirección 9 de la SRAM.
58	P2.5 (A13) (mC) A13(SRAM)	Pin 5 del puerto 2 del uC (address13) Dirección 13 de la SRAM.
59	P2.0 (A8) (mC) A8(SRAM)	Pin 0 del puerto 2 del uC (address8) Dirección 8 de la SRAM.
60	P2.6 (A14) (mC) A14(SRAM)	Pin 6 del puerto 2 del uC (address14) Dirección 14 de la SRAM.
61	**/OE (SRAM)**	**Enable de salida del SRAM.**
62	P3.6(/WR) **/WE (SRAM)**	Pin 6 del puerto 3 del uC(/WR)escritura de datos). **Pin de acceso a escritura del SRAM**.
63	-	-
64	-	-
65	**/CE (SRAM)**	**Pin de chip enable del SRAM.**
66	P1.6 (mC) PC_S5 (PUERTO PARALELO STATUS INPUT)	Pin 6 del puerto 1 del uC. Pin de estado 5 del puerto paralelo.
67	P1.7 (mC) VSYNC (VGA INPUTS)	Pin 7 del puerto 1 del uC. Señal de sinc. Vertical para monitor VGA.
68	P3.4 (TO) (mC) KB_CLK (PS/2)	Pin 4 del puerto 3 del uC(TO) Línea de reloj PS/2.
69	P3.1 (TXD) (mC) PC_S6 (PUERTO PARALELO STATUS INPUT) KB_DATA (PS/2)	Pin 1 del puerto 3 del uC(TXD) Pin de estado 6 del puerto paralelo Línea de datos de PS/2.
70	P1.5 (mC) PC_S3 (PUERTO PARALELO STATUS INPUT)	Pin 5 del puerto 1 del uC. Pin de estado 3 del puerto paralelo.
71	-	-
72	-	-
73	-	-
74	-	-
75	PC_S7 (PUERTO PARALELO STATUS INPUT)	Pin de estado del puerto paralelo.
76	-	-
77	P1.4 (mC) PC_S4(PUERTO PARALELO STATUS INPUT)	Pin 4 del puerto 1 del uC Pin de estado 4 del puerto paralelo.
78	A3 (SRAM)	Dirección 3 de la SRAM.
79	A4 (SRAM)	Dirección 4 de la SRAM.
80	P0.6 (A6/D6) (mC) **D6 (SRAM)**	Pin 6 del puerto 0 del uC(acceso multiplexado de address/ data). **Dato 6 del SRAM.**
81	P0.5 (A5/D5) (mC) **D5 (SRAM)**	Pin 5 del puerto 0 del uC(acceso multiplexado de address/ data). **Dato 5 del SRAM**
82	A5 (SRAM)	Dirección 5 de la SRAM
83	A6 (SRAM)	Dirección 6 de la SRAM
84	A7 (SRAM)	Dirección 7 de la SRAM

Tabla 6.3: Pines de la tarjeta XS40.(Continuación)

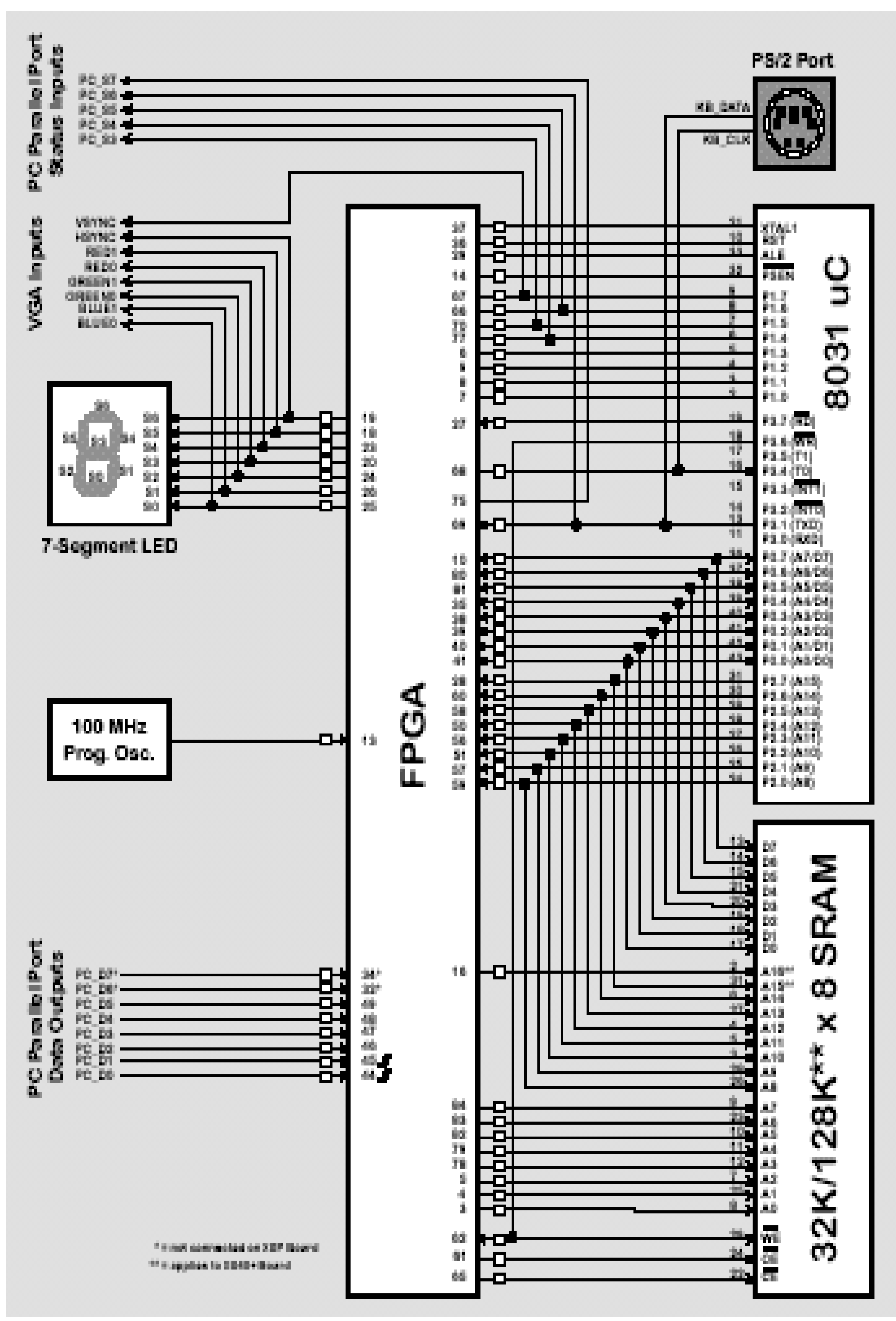


Figura 6.6: Arquitectura de la tarjeta XS40.

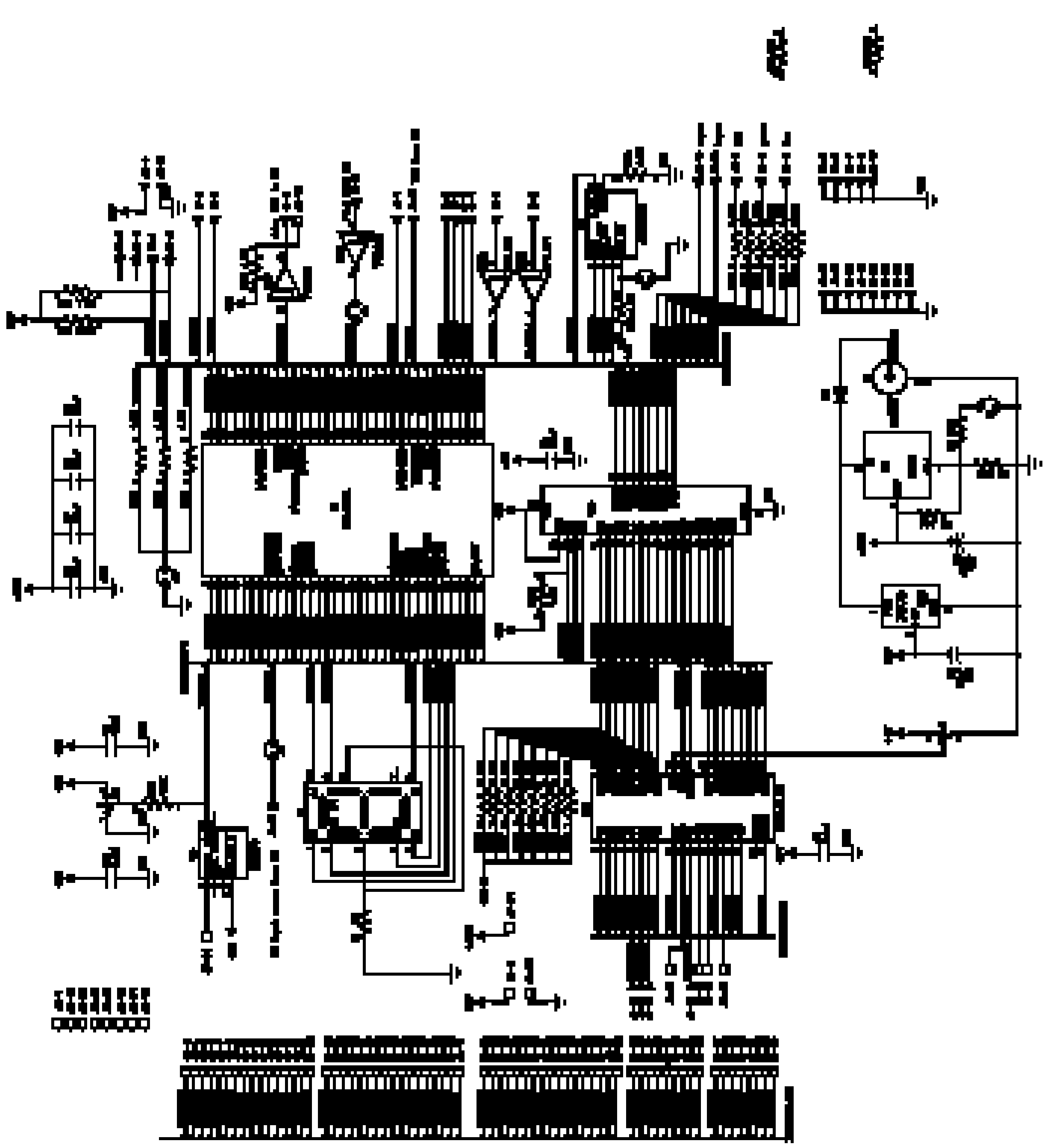


Figura 6.7: Esquema de la tarjeta XS40

6.6 XSTOOLs

El paquete XSTOOLs es una herramienta, que nos permite trabajar de forma cómoda en el entorno del sistema operativo Windows con la tarjeta XS40. Este software, se puede dividir en:

- GXSSETCLK. Permite configurar la frecuencia de reloj del oscilador programable.
- GXSTEST. Permite al usuario testear la tarjeta.
- GXSLOAD. Para cargar archivos de configuración de FPGA y CPLD y archivos HEX a XS40.
- GXSPORT. Permite enviar señales de entrada a la tarjeta XS mediante los pines de datos del puerto paralelo.

6.6.1 GXSSETCLK.

Una vez ejecutada la aplicación, debemos seleccionar el puerto paralelo al que esta conectada nuestra tarjeta en la lista desplegable con la etiqueta *Port* El siguiente paso será seleccionar en el menú desplegable identificado con la etiqueta *Board Type*, el tipo de FPGA que esta montado en la tarjeta, que en nuestro caso es XS4010-09, por último deberemos introducir un número entre 1 y 2052, que será el valor que dividirá la frecuencia de 100 Mhz maestra.

Una vez programado, el oscilador sacará una señal de reloj de valor de la frecuencia maestra de 100 MHz dividido por el valor que hayamos introducido. El divisor es guardado en el chip del oscilador en memoria no volátil, por lo que sólo volveremos a realizar esta operación cuando deseemos cambiar de nuevo la frecuencia de reloj.

También disponemos de la posibilidad de sustituir el reloj interno, por uno externo, seleccionando la casilla *External Clock*, para ello deberemos de disponer de una señal de reloj externa.
Pinchando en el botón *SET*, y siguiendo los pasos que nos indica el programa terminaremos de programar el reloj.

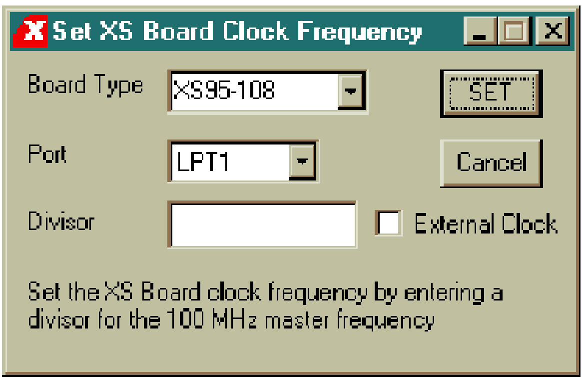


Figura 6.8: Cuadro de diálogo de GXSSETCLK.

6.6.2 GXSTEST.

Deberemos seleccionar, igual que en el programa anterior el puerto paralelo de conexión y el tipo de FPGA que tenemos pinchada en nuestra tarjeta.

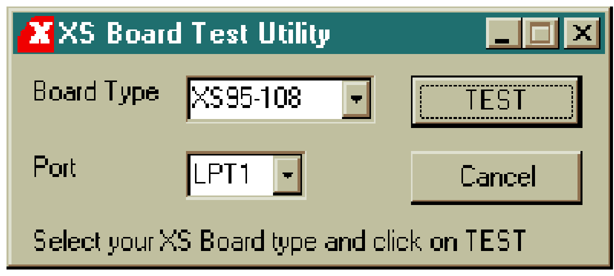


Figura 6.9: Cuadro de diálogo de GXSTEST.

Una vez seleccionados, haremos click en el botón de *TEST*. GXSTEST programará el microcontrolador y la FPGA para realizar la operación.
Al finalizar, mostrará un mensaje con la información del chequeo de la tarjeta informando si pasó o no el test.

6.6.3 GXSLOAD.

Una vez abierto el cuadro de diálogo de GXSLOAD, seleccionaremos el puerto al que esta conectada la tarjeta. En el caso que estemos programando una tarjeta con una EEPROM serie conectada, deberemos activar la casilla titulada *EEPROM* para activar la programación de ésta. En la mayoría de los casos, dejaremos la casilla sin seleccionar, por lo que programaremos la FPGA directamente.

Tras haber realizado estos pasos, podremos descargar los archivo a la tarjeta, arrastrando el archivo BIT o HEX a la ventana media del cuadro de diálogo de GXSLOAD como muestra la figura 6.10. Si seleccionamos un archivo no descargable[(1)], el programa lo ignorará.

Figura 6.10: Cuadro de diálogo GXSLOAD.

Durante el proceso, se mostrará el nombre del archivo que está siendo descargado, tal y como muestra la figura 6.11.

Una vez que la descarga ha finalizado, figura 6.12, los nombres de los archivos descargados serán mostrados en la ventana titulada *Recent Files*. A partir de este punto, podremos descargar los archivos que se muestran en la ventana pinchando en el botón *Reload*, por lo que podemos añadir o quitar archivos de configuración.

[1] Cualquier archivo diferente a .BIT, .SVF, o .HEX

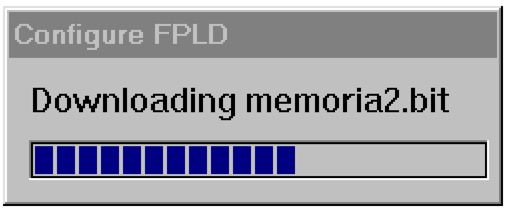


figura 6.11: Descarga realizada a la tarjeta XS40.

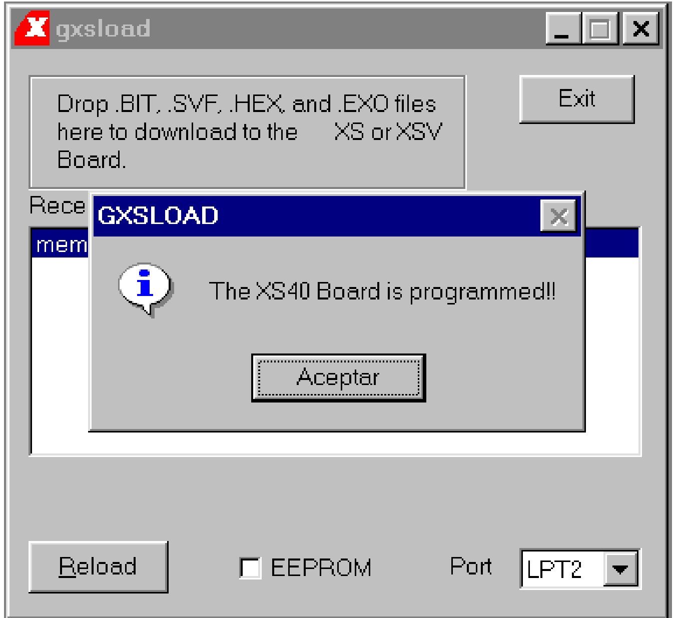


figura 6.12: Descarga realizada a la tarjeta XS40.

6.6.4 GXSPORT.

Esta ventana contiene diferentes controles las cuáles realizan las siguientes funciones:

De la misma manera que antes, mediante la lista desplegable titulada Port, permite seleccionar el puerto paralelo al que esta conectado al PC.

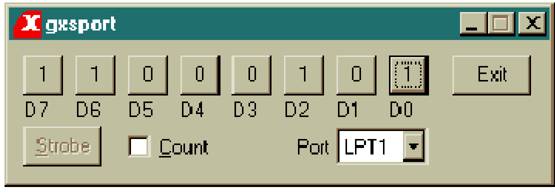


Figura 6.13: Cuadro de diálogo de GXSPORT.

Existen 8 botones, que se asocian con los 8 bits de datos del puerto paralelo.
Estos, se etiquetan con un valor binario, dependiendo de la salida de estos pines.
Cuando pinchas en uno de ellos, el valor del botón cambiará, pero no se modificará el pin correspondiente del puerto paralelo, hasta que presionemos el botón *Strobe*.

El botón de *Strobe* transfiere los valores de los botones a los pines de datos del puerto paralelo. Para que este botón esté habilitado, al menos debemos cambiar uno de los valores de los botones de datos.

Si pinchamos en el cuadro *Count*, el valor de salida de los pines de datos se incrementará.

CAPÍTULO 7

SISTEMA DE PROCESAMIENTO DE IMÁGENES BASADO EN BUS PCI.

Una vez descrita toda la plataforma hardware en la que se sustenta el sistema desarrollado, se va a proceder a proceder en este capitulo a explicar detenidamente el diseño realizado en este proyecto, los objetivos que se proponen alcanzar y todos los módulos de los que consta el sistema, tanto en la parte del PC como de la tarjeta XS-40.

El capitulo se estructura en tres partes: la primera con los objetivos del sistema, el punto inicial del que parte el proyecto y las justificaciones que han llevado ha desarrollar el sistema de la manera que se va a exponer (elección del bus PCI, transferencias DMA, etc..).

El segundo apartado trata de la descripción de sistema en si, módulos que lo componen de manera general y explicación de cada uno de los módulos, funcionamiento general y descripción del todo el software.

La última parte del capítulo pretende ser una guía de ayuda para todo aquel que quiera continuar el trabajo por donde alcanza este proyecto, con los problemas que pueden surgir a la hora de utilizar este sistema consideraciones de diseño importantes que se deben adoptar.

7.1 PLANTEAMIENTO DEL DISEÑO

Al final de este proyecto se pretende disponer de una plataforma en la cual se puedan programar todo tipo de aplicaciones de visión y tratamiento de imágenes en tiempo real de manera satisfactoria y con unas prestaciones aceptables para las demandas actuales de este tipo de aplicaciones.

Una parte fundamental del desarrollo ha sido el diseño de unas interfaces claras y reutilizables que permitan sustituir unos diseños por otros, implementar unas aplicaciones u otras sin tener que cambiar toda la plataforma. Además se ha pretendido que estas interfaces sean fácilmente ampliables con nuevas características si se desea, por lo que tienen que ser de fácil comprensión.

Como ya se sabe, se dispone de una tarjeta PCI PLX-9054 y una tarjeta XS-40 de Xilinxs con una FPGA de XC-4010-XL. Para el desarrollo de cualquier sistema para esta plataforma es necesario disponer en primer lugar de una interfaz hardware que comunique ambas tarjetas entre si, ya que la tarjeta PLX se conecta directamente a través del puerto PCI de cualquier PC. En nuestro caso, esta interfaz consiste en una sencilla placa de circuito impreso que conecta mediante pistas, los pines de bus plano conectados a la tarjeta PCI Proto-Lab con sus respectivos pines de la tarjeta XS-40. Esta placa de circuito impreso incluye un zócalo para la XS-40. En este proyecto se ha fabricado una placa de estas características mediante el uso del PCB que se encuentra para este caso en el proyecto fin de carrera [Rincón 01].

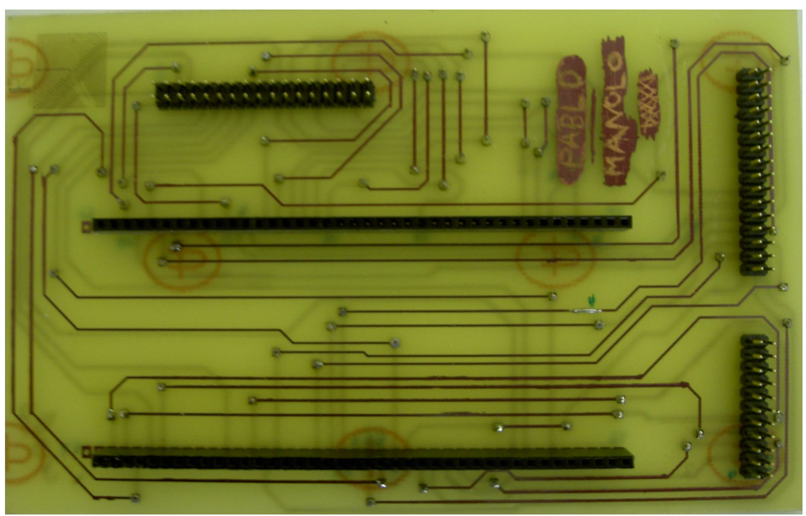


Figura 7.1: Placa de interconexión del hardware

Con esa base, en este proyecto se ha diseñado:

- Una nueva API para el desarrollo de aplicaciones que se comuniquen con la plataforma mediante transferencias DMA y maestro/esclavo, ya que la API que se distribuye con el kit de desarrollo es más compleja al no tener una programación orientada a objetos teniendo que almacenar diferentes datos tener en cuentas más consideraciones, por lo que su utilización exige un estudio más profundo de todas las posibles configuraciones, registros,

estructuras y funciones. En concreto se ha diseñado una clase interfaz denominada InterfazPlx que contiene todas la funciones.

- Una interfaz mejorada en la FPGA para transferencias DMA y maestro/esclavo que primitivamente se hizo mediante esquemáticos, pero que, para permitir que fuese más estándar y más fácil de comprender, se pasó a código de descripción hardware VHDL. Esto, además, facilita la ampliación de esta interfaz para futuras FPGAs de mayores prestaciones. Este modulo esta diseñado como un componente individual genérico, para que cualquiera que quiera disponer de comunicación a través de la tarjeta sólo tenga que instanciarlo y conectar las señales de control y datos al mismos.

- Se ha creado una aplicación de tratamiento de imágenes a partir del la interfaz grafica de usuario de la aplicación que se disponía en el proyecto anterior, pero en vez de usar la API que se distribuye con la tarjeta y transferencias maestro/esclavo, usa nuestra API y realiza transferencias DMA.

Las aplicaciones actuales de visión requieren una muy elevada velocidad de transmisión y una velocidad de procesado elevada, ya que la respuesta del sistema ha de ser inmediata y en muchos casos el procesado de la imagen puede ser complejo. Estas restricciones hacen que se elijan unas determinadas características en la plataforma como son, el bus de datos o el tipo de transferencia o el tipo de dispositivo lógico programable.

7.1.1 EL BUS PCI

En el capitulo 2 de este proyecto dedicado a este tipo de bus se ve claramente que éste es el que ofrece una mayor velocidad de transferencia de todos los disponibles. Pero esta no es la única característica que nos hace elegir este tipo de bus para nuestra plataforma.

Aparte de la diferencia de velocidades, el bus PCI es un estándar muy extendido, de fácil utilización, muy bajo coste y lo más importante, plug & play. Actualmente todos las placas del mercado para PC disponen de diferentes ranuras para tarjetas de este tipo, pudiendo conectar diferentes dispositivos a las mismas. En cambio, otros buses como los ISA o microcanal, han quedado anticuados y no es fácil encontrar actualmente PCs que dispongan de ranuras para este tipo de bus.

La tarjeta PCI Proto-Lab Plx incorpora un controlador PCI 9054 del fabricante Plx. Este controlador realiza todas las comunicaciones necesarias con el bus PCI ofreciendo de manera transparente para el usuario comunicación a través de este bus. El controlador PLX 9054 cumple con las especificaciones de la versión 2.2 del protocolo PCI, con una comunicación de 32 bits a 33 Mhz, lo que permite transferencias en modo ráfaga de hasta 132 MB/seg. Este es uno de los dispositivos más avanzados de propósito general de bus maestro.

Esta tarjeta es ampliable para futuras aplicaciones que requieran aun más velocidad de transferencia o un ancho de bus aun mayor de 32 bits. El fabricante Plx dispone de dos modelos para aceleración de I/O mediante bus PCI. Estos controladores son:

- El controlador Plx 9056 es el modelo siguiente al controlador Plx 9054. Dispone de un ancho de bus de 32 bits como el 9054, pero en este caso a 66Mhz, lo que aumenta considerablemente la velocidad de las transferencias. Al igual que el 9054 cumple con las

especificaciones de la versión 2.2 del protocolo PCI. Permite transferencias en modo ráfaga de hasta 264 MB/seg.

- El controlador Plx 9656 es el modelo de mayores prestaciones que ofrece este fabricante. Cumple con las especificaciones de la versión 2.2 del protocolo PCI. Dispone de un ancho de bus en PCI de 64 bits a 66 Mhz, que en el bus local es de 32 Mhz. Esto permite que se puedan realizar transferencias de hasta 528 MB/seg en el bus PCI y de 264 MB/s hacia el bus Local.

Se considera que las características que ofrece el controlador Plx 9054 superan, por el momento, las necesidades de nuestra plataforma, y al tener un precio más asequible se ha optado por este controlador y no por uno de los superiores. A pesar de esto, el protocolo del resto de dispositivos hacia el bus local es idéntico al del controlador 9054, por lo que no seria necesario cambiar la interfaz de comunicación con el controlador. Sólo sería necesario diseñar una nueva interfaz hardware para adaptar la tarjeta que contiene la FPGA con la nueva disposición y ancho de bus de otras tarjetas aceleradoras.

7.1.2. EL MODO DE TRANSFERENCIA DMA

A la hora de elegir el modo de transferencia hemos tenido dos opciones, el modo de transferencia maestro/esclavo, en el que el controlador pide el control del bus al procesador para realizar una transferencia y después lo devuelve, y las transferencias DMA en las cuales se abre una conexión directa entre la memoria y el controlador para transmitir ráfagas de datos.

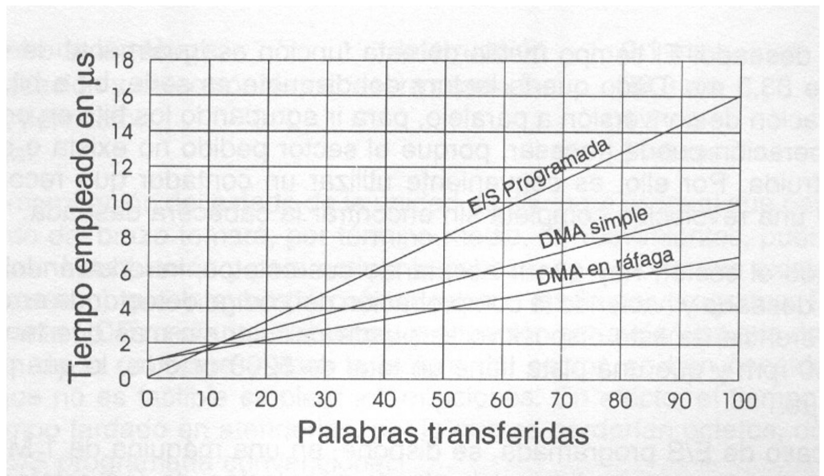


Figura 7.2: Velocidad de transferencia en función del tamaño

En el capitulo 3 dedicado al modo de transferencia DMA, se puede ver una profunda comparativa de estos modos de transferencias en sus diferentes modalidades. Es fácil llegar a la conclusión de que el modo de transferencia maestro/esclavo ofrece mejores prestaciones sólo cuando la cantidad de datos a transmitir es pequeña y se efectúan las transferencias de manera eventual, ya que no se necesita abrir canales para este tipo de comunicaciones. Es caso contrario, para transferencias mínimamente grandes, el modo de transferencia DMA supera con creces en velocidad al modo maestro/esclavo, además de permitir al procesador ejecutar otras tareas mientras se realizan las

transferencias (Figura 7.2). Por lo que el usar el modo de transferencia DMA no sólo afecta a la velocidad del sistema diseñado, sino que repercute en el funcionamiento global del sistema que lo contiene.
Como ya se sabe, precisamente las aplicaciones de visión, son consideradas como las que tienen una mayor tasa de transferencia de datos, ya que ésta transmite datos de gran tamaño (imágenes son o si procesamiento previo) de manera continuada. Un pequeño desfase en el envío y procesado de imágenes es fácilmente apreciable por el usuario, por lo que la tasa de transferencia debe se elevada y sin periodos esperas. Por ello, para nuestra plataforma no sólo es necesaria, sino que se hace obligatoria, que el modo de transferencia utilizado sea el DMA.

7.1.3. FPGA

Para el procesado de las imágenes es necesario un dispositivo lógico programable que tenga gran capacidad, ya que las operaciones que se realicen con las imágenes pueden ser muy complejas y contener un gran numero operaciones, con lo cual ocuparían mucha superficie. Además, el dispositivo que contenga este procesado debe ser de gran velocidad para que no produzca retardo en el sistema y se pueda devolver la imagen procesada rápidamente.

Lo más recomendable para este tipo de operaciones, es una FPGA, ya que ofrecen velocidades elevadas y puedes realizar operaciones muy complejas. Además es muy fácil realizar diseños para las mismas y programarlos gracias a que el software del que disponemos actualmente para ello es avanzado.

En nuestro caso usamos la tarjeta XS-40 de Xess (Figura 7.3). Esta tarjeta contiene la FPGA XC-4010XL del fabricante Xilinx. La tarjeta también dispone de un microcontrolador para operaciones de alto nivel con lógica de control más compleja, pero de menos frecuencia, y una memoria SRAM para almacenar datos intermedios. El ancho de datos del que dispone tanto la tarjeta como la memoria SRAM es de 8 bits.

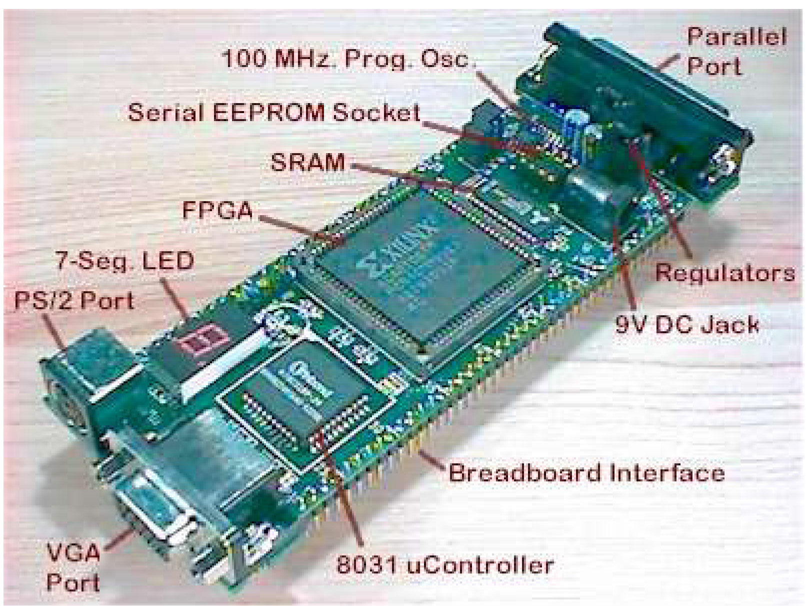


Figura 7.3: Tarjeta XS-40 de Xess

La tarjeta contiene un oscilador programable cuya velocidad máxima de operación es de 100Mhz. Esto es por el momento una velocidad suficientemente elevada para las operaciones que realizamos en nuestro caso.

El motivo de que en este proyecto se utilice esta tarjeta es simplemente por disponibilidad de la misma, ya que ésta es distribuida gratuitamente por Xilinx a los centro de enseñaza. Aunque no sea la que presenta mejores prestaciones, es suficiente para realizar los diseños de mucho más económica que el resto.

Se tiene previsto en futuras ampliaciones de este proyecto, disponer de una tarjeta que permita realizar operaciones de 32 bits y quizás de mayor velocidad. En muchas ocasiones, para realizar operaciones más complejas o configurar el controlador Plx 9054, es necesario comunicarse con palabras de 32 bits. Por lo que es necesario que la FPGA pueda procesar datos de este tipo para realizar éstas operaciones. Además aumentaría la velocidad del sistema en cuatro veces, ya que podríamos utilizar un mismo ciclo de escritura para transferir 32 bits en vez de los 8 actuales.

El diseño que se ha realizado para la interfaz que va a contener la FPGA prevé posibles ampliaciones de este tipo. El ancho del bus se datos se pude variar sin problemas, ya que la interfaz define este ancho de bus de tipo genérico, que por defecto será de 8 bits.

7.2 DESCRIPCIÓN DEL SISTEMA

El sistema global se puede dividir en 2 grupos principales como se puede observar en la figura 7.4. Por un lado, un modulo asociado al componente software, y por otro un modulo asociado al componente hardware o FPGA.

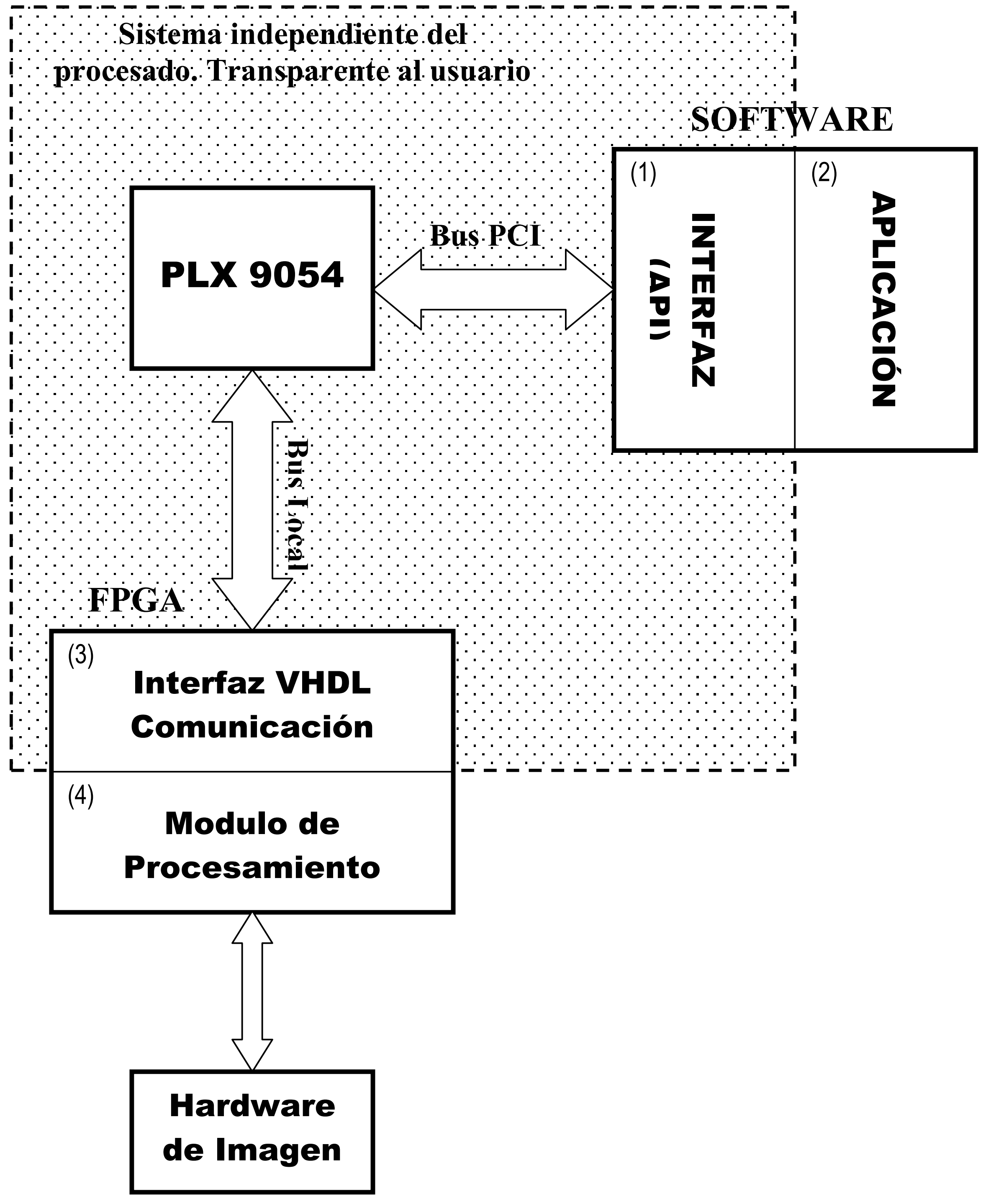


Figura 7.4: Diagrama de bloques del sistema desarrollado

En un diseño para una aplicación genérica, el usuario se comunicaría con el hardware de imagen, pudiendo enviar o recibir datos de manera transparente mediante una aplicación que hiciese uso de la API diseñada en este proyecto. Estos datos serían enviados del bus PCI al bus Local o viceversa gracias al modulo de comunicación programado a la FPGA y conectado a las señales de control del controlado Plx. Este modulo permite que diferentes algoritmos de imagen puedan ser programados en la FPGA o que se pueda comunicar el hardware de imagen con el usuario sin tener conocimiento de los protocolos que operan en el ambos buses.

7.2.1 MÓDULOS SOFTWARE

Se ha utilizado el entorno Visual C++ 6.0, como lenguaje de programación para el PC, por disponer de las librerías API y los drivers para Windows de nuestra tarjeta. Con éste se realizara la reserva de las cantidades de memoria necesarias para el manejo de las imágenes y las transferencias, el control y la configuración de la comunicación con la tarjeta de desarrollo conectada al Bus PCI y la configuración de las transferencias de datos en modo DMA a través del bus(Figura 7.4 (1)).

El modulo aplicación (Figura 7.4 (2)) se puede dividir en 2 submodulos. Uno de ellos sería el relativo a las operaciones que se realizan con las imágenes y la comunicación con el usuario (GUI). El otro es el relativo a la comunicación PCI. Las aplicaciones que se diseñen en el PC deben usar una interfaz para poder comunicarse con el controlador PLX de la tarjeta Proto-Lab. En el kit de desarrollo que distribuye el fabricante podemos encontrar una API para realizar esta comunicación y poder enviar y recibir datos, pero esta API es muy compleja y no tiene una estructura de programación orientada a objetos, por lo que además hay que tener unos conocimientos más avanzados de la tarjeta. Por ello se ha diseñado otra API más sencilla, orientada a objetos.

7.2.1.1 Diseño de la API

7.2.1.1.1 Descripción del soporte software suministrado por el fabricante

El soporte utilizado en el programa permite la utilización de diversas rutinas y funciones relacionadas con la transmisión de datos en modo PCI Target (Direct Slave) o DMA, la gestión de la memoria principal del PC y la comunicación y configuración con el controlador PLX9054. Todas estas funciones y rutinas están contenidas en los siguientes archivos ofrecidos con el kit de desarrollo:

PciTypes.h

Este archivo define los tipos básicos de datos disponibles para el código PCI. Entre ellos se encuentran:
unsigned long int (U32, *PU32); long int (S32, *SU32); unsigned short int (U16, *PU16); short int (S16, *PS16); volatile unsigned long int (VU32, *PVU32); volatile long int (V32, *PV32); unsigned char (U8, *PU8); char (S8, *PS8); LARGE_INTEGER (U64, *PU64; *PU64).

También define las estructuras básicas de localización del dispositivo, dirección virtual y memoria PCI.

PlxTypes.h

Este archivo de cabecera define los tipos básicos disponibles para el PCI SDK. Entre ellos cabe destacar; tipo de tamaño de acceso, definiciones para los canales DMA, direcciones de las transferencias DMA, definiciones de comandos y de los canales de prioridad DMA, definiciones para los tipos de memoria EEPROM y FLASH, definiciones de espacios PCI e I/O, estructuras de interrupciones, propiedades del Bus PCI, etc.

Plx.h

Este, contiene definiciones que son comunes a todos los códigos PCI SDK. Entre ellos, destacar tamaños de bloques, numero máximo de ranuras y buses PCI, Id validos del vendedor, dispositivo, etc.

PciApi.h

Este archivo contiene todas las funciones de prototipo del API PCI. Entre ellos destacamos los que se utilizan para esta aplicación:

PlxPciDeviceFind Busca los dispositivos PLX conectados al Bus PCI
PlxPciDeviceOpen: Abre el dispositivo PLX encontrado en el Bus PCI.
PlxPciDeviceClose Cierra el dispositivo PLX encontrado en el Bus PCI
PlxBusIopRead Lee un rango de valores del Bus Local de un dispositivo PCI que contenga un chip PLX(Lectura Direct Slave).
PlxBusIopWrite Escribe un rango de valores en el Bus Local de un dispositivo PCI que contenga un chip PLX(Escritura Direct Slave).

Del mismo modo, también se encuentran las funciones necesarias para transferencias DMA, acceso a la EEPROM serie, funciones para los pines de usuario, etc.

PlxError.h

Este archivo de cabecera define todos los códigos de error posibles del PLX PCI SDK.

7.2.1.1.2 Interfaz software desarrollada en el proyecto

Ésta interfaz ha sido desarrollada para facilitar la programación de aplicaciones que utilicen una comunicación PCI con la tarjeta Proto-Lab Plx (Figura 7.4 (1)). Al estar orientada a objetos es de muy fácil comprensión y utilización al contrario que la API se puede encontrar en el kit de desarrollo de la tarjeta. Esta interfaz proporciona funciones para realizar todo tipo de transferencias con la tarjeta y para configurar la misma. En la figura 7.5 podemos encontrar el esquema de la clase que muestra todas las variable que posee y todas las funciones programadas en la misma. Posteriormente se explica el funcionamiento de la clase y la utilización de todas y cada una de las funciones.

Class InterfazPlx
- HANDLE miDispositivoPlx; - PCI_MEMORY PciMemory; - RETURN_CODE rc; - DEVICE_LOCATION device;
+ InterfazPlx(); + InterfazPlx(DEVICE_LOCATION dev); + virtual ~InterfazPlx(); + static int NumeroDispositivos(); + static int NumeroDispositivosCriterio(DEVICE_LOCATION dev); + static int BuscaDispositivo(U32 num); - void CerrarDispositivo(); + int EscrituraMS(U32 address, PU32 datos); + int LecturaMS(U32 address, PU32 datos); + int CerrarCanalSglDma(int nC); + int AbrirCanalSglDma(int nC); + int EscrituraSglDma(U32 direccion,U32 tamano,PU32 datos,int nC); + int LecturaSglDma(U32 direccion,U32 tamano,PU32 datos, int nC); + int AbrirCanalBlockDma(int nC); + int CerrarCanalBlockDma(int nC); + PU32 EscrituraBlockDma(U32 direccion,U32 tamano,PU32 datos,int nC); + PU32 LecturaBlockDma(U32 direccion,U32 tamano,PU32 datos,int nC); + int AbrirCanalShuttleDma(int nC); + int CerrarCanalShuttleDma(int nC); + int EscrituraShuttleDma(U32 direccion,U32 tamano,PU32 datos,int nC); + int LecturaShuttleDma(U32 direccion,U32 tamano,PU32 datos,int nC); + void EscribirRegistro(U32 registro, PU32 dato); + U32 LeerRegistro(U32 registro); + U32 InterfazPlx::LeerRegistroConfiguracion(U32 registro); + void InterfazPlx::EscribirRegistroConfiguracion(U32 registro, U32 dato); + U32 InterfazPlx::DireccionFisicaBuffer(); + U32 InterfazPlx::BufferSize(); + U32 InterfazPlx::DireccionLogicaBuffer(); + bool InterfazPlx::setBuswidth(int w);

Figura 7.5: Clase InterfazPlx

La interfaz ha sido programa en C++ por lo que, gracias a la programación orientada a objetos, un usuario de la misma sólo tiene que instanciar e inicializar un objeto de la clase InterfazPlx para poder acceder a todas las funciones de transferencias y configuración del dispositivo. Esto supone una mejora considerable respecto a la anterior API que no ofrece las facilidades de uso de las clases de C++.
En esta interfaz se han implementado las funciones de transferencias basicas y de configuración. Posteriormente esta API puede ser ampliada fácilmente con más funciones para que, por ejemplo, realice otros modos de transferencia o realice un control avanzado de interrupciones.

Para realizar transferencias con esta API los pasos a seguir serían los siguientes:

1. Detectar las tarjetas que están conectadas al PC.

2. Obtener la localización de la tarjeta deseada e inicializarla.(En el caso de que sólo haya una tarjeta estos dos pasos se pueden realizar directamente en uno)

3. Si se desea, cambiar la configuración de la tarjeta a través de los registros.

4. Elegir el modo de transferencia que se desee. Pudiendo ser:

    a. Maestro/Esclavo

    b. DMA. En DMA podemos realizar 3 tipos de transferencias diferentes. Scatter/Gatter, DMA por bloques y Shuttle. Para realizar transferencias DMA los pases a seguir son:
        i. Apertura de un canal DMA.
        ii. Comunicación
        iii. Cierre del canal

5. Cerrar el dispositivo (destruyendo el objeto).

Para realizar todos estos pasos la API dispone de un gran numero de funciones que iremos explicando detenidamente.

PASO 1 - Búsqueda de dispositivos.

Para detectar las tarjetas conectadas al PC, la API incluye una serie de funciones de tipo static al efecto. Estas funciones buscan las tarjetas conectadas al PC sin necesidad de haber creado antes un objeto de la clase. De esta manera se puede elegir que tarjeta se desea utilizar para realizar transferencias. Esto permite trabajar con distintas tarjetas simultáneamente.

Estas funciones son:

- **static int NumeroDispositivos():** Esta función devuelve el numero total de dispositivos conectados al PC en forma de entero.

- **static int NumeroDispositivosCriterio(DEVICE_LOCATION dev):** Devuelve el numero de dispositivos conectados al PC según un criterio de búsqueda definido en dev. Este criterio

puede incluir el modelo del dispositivo, el bus o slot al que están conectados o según un numero de serie (en cuyo caso sólo devolvería el valor 1 si hay algún dispositivo conectado con ese numero de serie, ya que el numero de serie es único).

- **static DEVICE_LOCATION BuscaDispositivo(U32[1] num):** Esta función devuelve toda la configuración de localización del dispositivo especificado en num, que define el numero de dispositivo por orden de búsqueda. Posteriormente se puede usar esta configuración para abrir el dispositivo seleccionado mediante el constructor sobrecargado que incorpora dicha posibilidad.

Las funciones *NumeroDispositivosCriterio* y *BuscaDispositivo* trabajan con datos de tipo *DEVICE_LOCATION*. Esta es una estructura que contiene datos del modelo de la tarjeta, mediante los campos DeviceID (modelo de dispositivo) y VendorID (vendedor del dispositivo), de la localización de la tarjeta en el PC como bus y slot en la que se encuentra pinchada y el numero de serie de la misma. La estructura es la siguiente:

```
typedef struct _DEVICE_LOCATION
{
      U32 DeviceId;
      U32 VendorId;
      U32 BusNumber;
      U32 SlotNumber;
      U8 SerialNumber [16];
} DEVICE_LOCATION, *PDEVICE_LOCATION;
```

PASO 2 - Inicialización de la tarjeta

Una vez decidido qué tarjeta se va a utilizar, esta debe ser inicializada para poder trabajar con ella. Para ello hay que crear un objeto de la clase InterfazPlx y usar el constructor de la clase para inicializarlo. Disponemos de 2 constructores para esto. El primero de ellos, que es el que se ejecuta por defecto si no se pasan parámetros, evita tener que realizar el paso anterior de buscar el dispositivo.

- **InterfazPlx():** Constructor por defecto que abre directamente el primer dispositivo que encuentre conectado al PC. Esto es eficaz si sólo hay un dispositivo conectado, que será lo más normal, o si sabemos que el dispositivo que queremos abrir es el primero.

- **InterfazPlx(DEVICE_LOCATION dev):** Este constructor sobrecargado abre un dispositivo según una configuración de localización del mismo. Esta configuración es normal que no se conozca en principio, por lo que se han diseñado herramientas para obtenerla fácilmente que ya han sido explicadas detalladamente en el PASO 1. Al igual que en las funciones *NumeroDispositivoCritero* y *BuscaDispositivo* este constructor usa los datos de tipo *DEVICE_LOCATION*.

Como se verá más adelante en los ejemplos, el ahorro de código y la facilidad de uso con respecto a la API proporcionada en el kit de desarrollo es enorme, ya que, con una simple línea de código como "InterfazPlx miDispositivo( );" se ha buscado e inicializado comprobando posibles errores el

[1] Tipo de dato unsigned long. Este tipo de dato es de 32 bits.

dispositivo conectado en tu PC evitando tener que incluir en los programa engorrosos procesos de búsqueda e inicialización de más difícil comprensión.

PASO 3 - Configuración

La interfaz incluye la posibilidad de configurar la tarjeta leyendo y escribiendo en los diferentes registros de los que dispone la misma. Se diferencian dos grupos de registros a la hora de utilizar las funciones. Uno de estos grupos son los registros de configuración de transferencias, dedicados a definir el tipo de transferencia y el modo de la misma, dirección de la transferencia, configuración de canales, numero de octetos a transmitir y direcciones origen y destino. Generalmente las funciones de la API se encargan directamente de estos menesteres a la hora de realizar transferencias siendo el PC maestro, pero disponer de funciones que escriban estor registros, permite, por ejemplo, configurar la tarjeta en modo de demanda DMA para que la FPGA pueda iniciar transferencias DMA de manera maestra. El otro grupo contiene registros de configuración de la tarjeta en si (modelo y vendedor de la tarjeta, especificaciones eléctricas, LED, direcciones, etc...). En algunos registros de configuración de la tarjeta, por motivos de seguridad, sólo se permite la lectura, y otros sólo pueden ser escritos a través de la EEPROM serie que incorpora la tarjeta y que contiene, por ejemplo, el modelo y el vendedor de la tarjeta entre otras cosas.

La manera de acceder a los diferentes registros a través de las funciones es por el offset de memoria de registros en el que se encuentran. Al acceder a un registro, su offset siempre tiene que ser múltiplo de 4 octetos, de manera que no se pueda acceder a la mitad de un registro. Sólo se puede acceder a los registros con palabras de 32 bits. Por lo que se tienen que escribir siempre 32 bits. Si se desea escribir un registro de menor longitud, se escribirá también en el registro anterior/posterior (que en el caso que no se desee cambiar habrá que introducir los mismos datos). Igualmente para leer un registro, las funciones devuelven los 32 bits siguientes del offset perteneciente al registro seleccionado.

Las funciones de lectura y escritura de registros son:

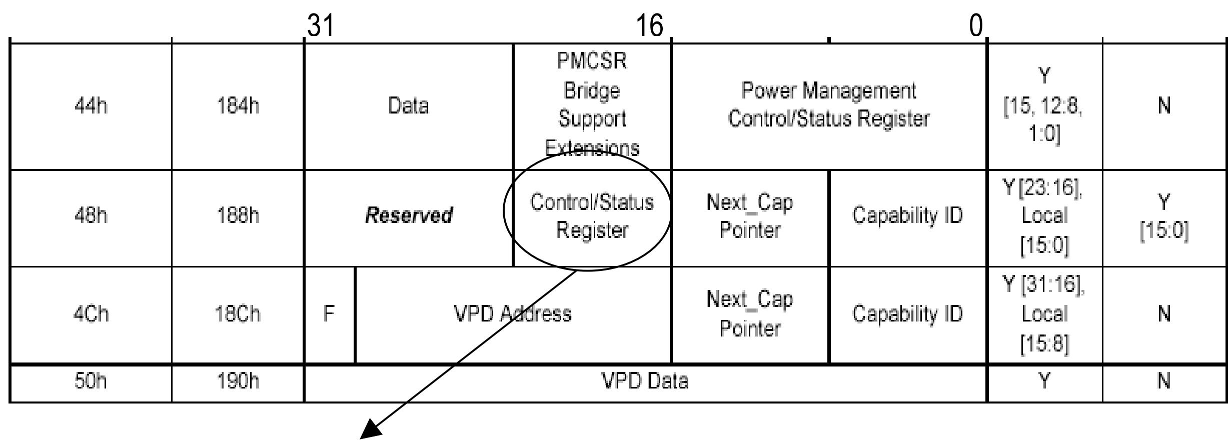
- **void EscribirRegistro(U32 registro, PU32 dato):** Escribe registros relativos a la tarjeta. Estos registros son desde el offset 00h hasta el FCh. La función requiere el offset del registro destino y la dirección donde se encuentran los 32 bits a escribir en dicho registro.

- **U32 LeerRegistro(U32 registro):** Esta función devuelve los 32 bits siguientes al offset de registro que se le pasa como argumento. Estos 32 bits componen el contenido de uno o varios registros. El offset debe encontrarse entre el 00h y el 7Ch. Igual que en caso anterior este espacio de registros pertenece a los de configuración de trabajo de la tarjeta y no a los de transferencias.

- **U32 LeerRegistroConfiguracion(U32 registro):** Esta es la función respectiva de lectura de registros de configuración DMA. Los registros a los que accede son del 80h al B8h.

- **void EscribirRegistroConfiguracion(U32 registro, U32 dato):** Función de escritura de registros de configuración de transferencias. Como se ha dicho antes, estos registros son del 80h al B8h. Igual que en todos los casos anteriores se tienen que escribir obligatoriamente 32 bits completos, por lo hay que tener especial cuidado en los registros de menos de 32 bits ya que se puede variar involuntariamente registros contiguos.

Por ejemplo, si deseamos encender o apagar el LED que se encuentra en la tarjeta Proto-Lab Plx deberíamos activar el bit 3 del registro Control Status (HS_CSR[3]) como se observa en la tabla 7.2. En esa misma tabla se observa que para encender el LED hay que escribir un '1' en esa posición y que por defecto este LED se encuentra apagado. El offset para acceder a este registro desde el PCI es el 4Ah en hexadecimal, pero el registro es de 16 bits y nosotros debemos escribir obligatoriamente datos de 32 bits para acceder a los registros. En la tabla 7.1 podemos ver los registros contiguos a este que forman la palabra de 32 bits a escribir, por lo que el offset que realmente deberemos pasar como parámetro a la función de escritura es el 48h.



		31		16	0		
44h	184h	Data	PMCSR Bridge Support Extensions	Power Management Control/Status Register		Y [15, 12:8, 1:0]	N
48h	188h	*Reserved*	Control/Status Register	Next_Cap Pointer	Capability ID	Y [23:16], Local [15:0]	Y [15:0]
4Ch	18Ch	F VPD Address		Next_Cap Pointer	Capability ID	Y [31:16], Local [15:8]	N
50h	190h	VPD Data				Y	N

Registro a escribir

Tabla 7.1: Segmento de tabla de registros de configuración PCI

Register 11-33. (HS_CSR; PCI:4Ah, LOC:18Ah) Hot Swap Control/Status

Bit	Description	Read	Write	Value after Reset
0	***Reserved.***	Yes	No	0
1	**ENUM# Interrupt Clear.** Writing a 0 enables the interrupt. Writing a 1 clears the interrupt.	Yes	Yes/Clr	0
2	***Reserved.***	Yes	No	0
3	**LED Software On/Off Switch.** Writing a 1 turns on the LED. Writing a 0 turns off the LED.	Yes	PCI	0
4	***Reserved.***	Yes	No	0
5	***Reserved.***	Yes	No	0
6	**Board Removal ENUM# Status Indicator.** Writing a 1 reports the ENUM# assertion for removal process.	Yes	Yes	0
7	**Board Insertion ENUM# Status Indicator.** Writing a 1 reports the ENUM# assertion for insertion process.	Yes	Yes	1
15:8	***Reserved.***	Yes	No	0h

Tabla 7.2: Registro Control Status de configuración PCI

Para por ejemplo encender el LED sin variar el resto de bits, se puede realizar un ciclo de lectura y con el resultado de esta operación realizar una operación OR junto con un 00080000h (sólo esta activo el bit numero 19 de los 32 que es el bit 3 del registro a cambiar). Esto activa (si no está ya activo) dicho bit si variar el resto. Posteriormente el resultado de la OR se escribe en el mismo registro. Para esto se utilizarían las funciones *LeerRegistro(0x48);* y *EscribirRegistro(0x48,dato);*.

La operación de lectura devuelve, en el caso de que no se haya variado antes, el dato 00804C06h, y después de la operación el registro quedaría como 00884C06h.

Vamos a ver otro ejemplo de escritura de registros en el cual se activa el modo de transferencia DMA Scatter/Gatter con modo demanda, en el cual el bus local puede iniciar transferencias DMA de manera maestra. En este caso, al ser registros pertenecientes a la parte de configuración DMA se usan las funciones *EscrituraRegistroConfiguración* y *LecturaRegistroConfiguración*.

Después de crear un objeto de la clase InterfazPlx, lo primero que hay que hacer es activar los bits *DMA Scatter/Gatter* y *DMA Demand Mode* del registro DMAMODE0 (offset 80h). Estos bits son el 9 y el 12 respectivamente de dicho registro, que tiene un ancho de 32 bits como se ve en la tabla 7.3. Para ello realizamos una operación de lectura, e igual que en el ejemplo anterior realizamos una operación OR con los datos para variar sólo los que nos interesan. Este trozo quedaría así:

```
registro=0x80; //Registro DMAMODE0

//Leemos el registro
printf("\n%X\n",ireg=miTarjeta.LeerRegistroConfiguracion(registro));

oreg=ireg|0x00001200;//Activamos sólo los bits deseados

//Escribimos el nuevo valor
miTarjeta.EscribirRegistroConfiguracion(registro,oreg);
```

Register 11-70. (DMAMODE0; PCI:80h, LOC:100h) DMA Channel 0 Mode

Bit	Description	Read	Write	Value after Reset
1:0	**Local Bus Width.** Writing a 00 indicates an 8-bit bus width. Writing a 01 indicates a 16-bit bus width. Writing a 10 or 11 indicates a 32-bit bus width.	Yes	Yes	M = 11 J = 11 C = 11
5:2	**Internal Wait States (**data-to-data**).**	Yes	Yes	0h
6	**TA#/READY# Input Enable.** Writing a 1 enables TA#/READY# input. Writing a 0 disables TA#/READY# input.	Yes	Yes	1
7	**BTERM# Input Enable.** Writing a 1 enables BTERM# input. Writing a 0 disables BTERM# input. For more information, refer to Section 2.2.5 for M mode or Section 4.2.5 for C and J modes.	Yes	Yes	0
8	**Local Burst Enable.** Writing a 1 enables Local bursting. Writing a 0 disables Local bursting.	Yes	Yes	0
9	**Scatter/Gather Mode.** Writing a 1 indicates Scatter/Gather mode is enabled. For Scatter/Gather mode, DMA source address, destination address, and byte count are loaded from memory in PCI or Local Address spaces. Writing a 0 indicates Block mode is enabled.	Yes	Yes	0
10	**Done Interrupt Enable.** Writing a 1 enables an interrupt when done. Writing a 0 disables an interrupt when done. If DMA Clear Count mode is enabled, the interrupt does not occur until the byte count is cleared.	Yes	Yes	0
11	**Local Addressing Mode.** Writing a 1 holds the Local Address bus constant. Writing a 0 indicates the Local Address is incremented.	Yes	Yes	0
12	**Demand Mode.** Writing a 1 causes the DMA controller to operate in Demand mode, as well as to make USERo/DREQ0#/LLOCKo# an input (DREQ0#) and USERi/DACK0#/LLOCKi# an output (DACK0#). In Demand mode, the DMA controller transfers data when its DREQ0# input is asserted. Asserts DACK0# to indicate the current Local Bus transfer is in response to DREQ0# input. DMA controller transfers Lwords (32 bits) of data. This may result in multiple transfers for an 8- or 16-bit bus.	Yes	Yes	0

Tabla 7.3: Fragmento del registro DMAMODE0

A continuación se debe escribir la dirección PCI en el registro DMAPADR0 (offset 84h). Esta es la dirección física (es importante recordar que este registro contiene la dirección física, no la dirección lógica) de la variable origen/destino en el PC. En nuestro ejemplo vamos a usar el buffer de la tarjeta para hacer intercambio de datos con el bus local, por lo que escribiremos la dirección física de este buffer que nos proporciona la función *DireccionFisicaBuffer()* que explicamos un poco más adelante. Para trabajar con este buffer y leer los datos que escriba el bus Local, o escribir los datos que se mandarán al mismo bus deberemos en el programa crear un puntero a la dirección lógica de este buffer. Esta dirección la obtenemos con la función *DireccionLogicaBuffer()*. El cambio de registro quedaría de la siguiente manera:

```
registro=0x84; //Registro DMAPADR0 que contiene la direccion PCI

oreg=miTarjeta.DireccionFisicaBuffer();   //El destino será el buffer de
                                          //la tarjeta

//Escribimos el nuevo valor
miTarjeta.EscribirRegistroConfiguracion(registro,oreg);
```

También hay que escribir la dirección del bus Local (si se va a trabajar con la memoria RAM de la tarjeta), el numero de octetos a transmitir y la dirección de la transferencia en los registros DMALADR0 (offset 88h), DMASIZ0 (offset 8Ch) y DMADPR0 (offset 90h). Esto se realiza de la misma manera que en la escritura del registro DMAPADR0, pero en este caso los datos a escribir serán la dirección en donde se encuentran los datos dentro de la RAM y el numero de octetos en los 2 primeros casos. Para definir la dirección de la transferencia debemos escribir el bit 3 del registro DMADPR0. Si este bit es igual a ‘0’ la dirección de la transferencia será en sentido PCI a Bus Local, si es un ‘1’ será Local a PCI.

Una vez cambiados todos estos registros, la tarjeta ya está configurada para trabajar en modo demanda DMA. A continuación, cada vez que la FPGA quiera realizar una transferencia, debe activar la señal DREQ0 para pedir el control del bus Local y una vez que lo posea controlar la transferencia con los bits DMA enable, DMA start, DMA done y DMA abort del registro DMACSR0. Estos bits sólo se pueden cambiar si se dispone de un bus Local de 32 bits. Esta es la causa de que nosotros no podamos usar este modo de transferencia, de enorme utilidad, ya que puede leer escribir en direcciones del PC sin necesidad de que un programa este realizando operaciones de lectura/escritura.

En el capitulo dedicado al controlador Plx se muestran todos los registros y su contenido por defecto, así como las diferentes configuraciones que podemos cargar mediante los mismos. También incluye una explicación del modo de transferencia DMA en modo de demanda y el protocolo de señales para obtener el control del bus a través de la utilización de la señal DREQ0.

La tarjeta provee un buffer para realizar transferencias con el bus local. Este buffer es de obligada utilización para operaciones de DMA por bloques, pero su utilización ya se incluye dentro de las funciones de DMA por bloques. A pesar de esto se han incluido funciones para obtener las direcciones física y lógica de este buffer así como el tamaño del mismo por si se desean realizar operaciones avanzadas con el o usarlo como caché de intercambio con la tarjeta como hacemos en el ejemplo anterior.

Las funciones encargadas de esta tarea son:

- **U32 DireccionFisicaBuffer():** Devuelve la dirección física del buffer.
- **U32 DireccionLogicaBuffer():** Devuelve la dirección lógica del buffer.
- **U32 BufferSize():** Devuelve el tamaño del buffer.

La última función en tareas de configuración es la de definir el ancho de bus con que se realizaran las transferencias DMA. Éste ancho esta definido por defecto en 8 bits. Para variar el ancho se usa la función *bool setBusWidth(int w)* a la cual se le pasa como parametro este ancho en forma de entero. El dato debe ser 8, 16 o 32, en caso contrario la función devuelve false.

PASO 4 - Transferencias

Una vez que se dispone de un manejador para la tarjeta y se ha configurado la misma, si es necesario, de la forma deseada, se puede empezar a realizar transferencias. Para realizar las transferencias podemos escoger entre el modo de transferencia maestro/esclavo, los distintos modos de DMA o una combinación de ellos. Para realizar las transferencias no es necesario cambiar ningún registro ni realizar ninguna configuración previa, ni tampoco se necesita realizar ninguna operación especial para cambiar el modo de transferencia. Simplemente de ejecutan las funciones de manera indiferente pudiendo trabajar simultáneamente con maestro/esclavo y DMA, con varios canales DMA o cambiar registro en tiempo de ejecución sin ningún problema. En cualquier caso, la operación es muy sencilla gracias a la programación orientada a objetos y vasta

con utilizar el objeto de la clase InterfazPlx creado para llamar a la función que se desee utilizar en cada momento.

A) MAESTRO/ESCLAVO

La interfaz permite escribir y leer en modo maestro/esclavo en el bus local, es decir en la FPGA conectada a la tarjeta Plx mediante la utilización de las siguentes funciones:

- **int EscrituraMS(U32 address, PU32 datos):** Esta función efectúa un ciclo simple de escritura en la FPGA en modo maestro/esclavo, configurando los registros necesarios y activando en el bus Local las señales necesaria para ello. Únicamente necesita la dirección de destino(address) y la dirección de origen de los datos (datos). Actualmente, debido a que la tarjeta XS-40 sólo puede trabajar con 8 bits, sólo serán útiles los ocho bits más bajos de la palabra que se escribe, por lo que se debe de tratar el dato como tal (tiene un valor comprendido entre 0 y 255) y trabajar con él de manera totalmente normal como si de un *char* se tratase. En caso de cambiar la tarjeta XS-40 por una que posea un ancho de bus de 32 bits, esta función trabaja de la misma manera y no es necesario realizar ningún cambio. La diferencia será que en vez de ser validos sólo los 8 bits de menos peso, será valido toda la palabra de 32 bits (el dato tendrá un valor comprendido entre 0 y 4294967295).
- **int LecturaMS(U32 address, PU32 datos):** Esta es la función contraria a la anterior. Efectúa un ciclo de lectura simple desde la FPGA. En este caso, los argumentos que necesita son la dirección origen de los datos, es decir, la dirección del bus local de la que proceden los datos; la dirección destino de los datos, que tiene que ser la dirección de una variable de tipo unsigned long. Al igual que en el caso anterior, con la tarjeta XS-40 sólo serán validos los 8 bits más bajos del dato, e igualmente, si se cambia de tarjeta, se podrá trabajar con la función de la misma manera sin ser necesario hacer ningún cambio en la interfaz.

En ambas funciones, la dirección que corresponde con el bus PCI (*address*) se refiere a la dirección que quedará escrita en el bus local y que esta conectada a los pines de la memoria RAM de la misma. Si no usamos la memoria, el valor que contenga este dato es indiferente.

B) DMA

Como se explico en el capitulo dedicado a DMA, la transferencia maestro/esclavo es muy ineficiente para grandes cantidades de datos. Por lo que la interfaz provee también funciones para realizar transferencias DMA.

Se permiten tres tipos de transferencia DMA distintos, transferencia de bloques, transferencia scatter/gatter (esparcir/recopilar), y transferencia shuttle. La dos primeras, las de bloque (direcciones contiguas de memoria) y las scatter/gatter (datos esparcidos por la memoria), se explicaron en el capitulo 3 dedicado al DMA de manera teórica y en el capitulo 5 dedicado al controlador Plx 9054 de manera especifica para esta tarjeta y su funcionamiento en el sistema. La transferencia shuttle o lanzadera son transferencias DMA del tipo scatter/gatter consecutivas. Estas transferencias repiten una transferencia DMA Scatter/Gatter con unas mismas direcciones origen y

destino, en las cuales el controlador lee el origen lo manda al destino y cuando este lo recibe vuelve ha realizar la misma operación hasta que se le indique lo contrario. Esto es útil por ejemplo para tener una variable que se vaya actualizando mediante un procesado y que este tipo de transferencia, siempre activo vaya enviando cada actualización de la variable al destino indicado. Igualmente podría darse el caso contrario y definir una variable que se vaya escribiendo con nuevos datos desde el bus PCI constantemente.

Abrir canales

En todos los casos, antes de realizar transferencias, es necesario abrir un canal DMA del tipo de transferencia que se desee utilizar. Esto permite que el procesador abra la conexión entre el controlador DMA del dispositivo y la memoria principal para que puedan intercambiar información. Al abrir los canales, se configuran directamente los registros necesarios para realizar transferencias.
La configuración con la que se abren los canales es idéntica para todos los tipos de canales. Los canales se configuran con una prioridad de canal rotacional *"DmaChannelPriority=Rotational;"*. Esto se puede variar para darle prioridad a cualquiera de los canales cambiando el valor *Rotational* por *Channel0Highest o Channel1Highest* en el código fuente de la interfaz.

El ancho del bus local esta definido por defecto en 8 bits, que es el ancho de bus máximo que permite nuestra tarjeta XS-40. Si se cambia la tarjeta por otra de mayor bus, se puede variar el ancho de bus en la configuración mediante la función que ofrece la API para ello denominada *setBusWidth()*. El resto de la configuración es fija y conviene no cambiarla ya que define el tipo de señales de control que utilizará el bus local para comunicarse con el controlador PLX y esto impediría el funcionamiento de la tarjeta con nuestro software.
Las funciones encargadas de abrir los canales son:

- **int AbrirCanalSglDma(int canal):** Esta función abre un canal de tipo scatter/gatter. Es necesario pasarle el numero de canal que se desea utilizar. El dispositivo permite usar dos canales DMA (el cero y el uno), con dos conexiones por cada uno (primaria y secundaria). En esta interfaz, por motivos de simplicidad, sólo se utilizan los canales primarios de cada canal. Si se desease ampliar la interfaz para tener disponibles el resto, bastaría con repetir en la función llamadas a la apertura del canal pero con los canales secundarios. La función abre el canal primario numero 0 si se le pasa como argumento un 0, en caso contrario abre el canal primario 1. Si el canal seleccionado ya esta abierto devuelve un código de error.

- **int AbrirCanalBlockDma(int canal):** Ídem de la anterior, pero en este caso el tipo de transferencias que permite hacer sobre el canal son transferencias de bloque. Al igual que en las scatter/gatter sólo abre los canales primarios 0 y 1.

- **int AbrirCanalShuttleDma(int canal):** Esta función abre un canal DMA para transferencias del tipo shuttle (lanzadera), que son muy similares a las transferencias scatter/gatter. De hecho la diferencia entre ellas es que este tipo de transferencias realiza una comunicación scatter/gatter entre un origen y un destino de forma permanente.

Transferir

Una vez abierto un canal se puede iniciar transferencias para ese canal con las funciones de transferencias que ofrece la interfaz. Las funciones utilizadas siempre deben ser del tipo para el cual esta dedicado el canal, en caso contrario, no se realizarán las transferencias y la función devolverá un código de error. Cabe la posibilidad de abrir los dos canales DMA que permite la interfaz y utilizar cada uno para realizar transferencias diferentes, tanto en el modo de transferencia como en la dirección y simultáneamente estar realizando transferencias maestro/esclavo para, por ejemplo, llevar el control de uno de los módulos conectados a la interfaz. Se ha dejado para futuros trabajos la posibilidad de habilitar los dos subcanales de que dispone cada canal DMA para hacer transferencias, pudiendo realizarse una multiplexación de las transferencias en los canales primario y secundario de cada canal DMA. Esto permitiría realizar hasta 4 transferencias DMA simultaneas multiplexadas dos a dos más las transferencias maestro/escavo para pocos datos, que no usan canales para la transmisión.

Las funciones de comunicación DMA son las siguientes:

- **int EscrituraSglDma(U32 direccion,U32 tamano,PU32 datos, int canal):** Esta función realiza una escritura de datos a la tarjeta en modo Scatter/Gatter. Al igual que las escrituras maestro/esclavo se necesita la dirección destino, el numero de octetos a transmitir y la dirección de la variable que contiene los datos. Además, al ser transferencia DMA, es necesario especificar el canal DMA por el cual se va a transmitir y que debe haber sido abierto con *AbrirCanalSglDma.*

- **int LecturaSglDma(U32 direccion,U32 tamano,PU32 datos,int canal):** Realiza una lectura desde la tarjeta en modo scatter/gatter. Los argumentos que se le pasan son: la dirección origen de la tarjeta, el numero de octetos que se van a leer, la dirección de la variable donde se guardaran los datos y el numero de canal, que al igual que en caso anterior, debe haber sido abierto con *AbrirCanalSglDma.*

- **int EscrituraBlockDma(U32 direccion,U32 tamano,PU32 datos, int canal):** Escritura de un bloque en DMA a través del canal seleccionado. Los argumentos son iguales que en el caso de la scatter/gatter.

- **int LecturaBlockDma(U32 direccion,U32 tamano,PU32 datos,int canal):** Lectura de un bloque de datos en DMA. Los argumentos son los mismos que en casos anteriores.

- **int EscrituraShuttleDma(U32 direccion,U32 tamano,PU32 datos, int canal):** Escritura DMA en modo shuttle. Este modo de escritura se encarga de escribir continuamente desde el origen al destino con transferencias similares a las scatter/gatter.

- **int LecturaShuttleDma(U32 direccion,U32 tamano,PU32 datos,int canal):** Realiza lectura de la tarjeta en modo shuttle. Permanece continuamente leyendo de la dirección origen y escribiendo los datos en el destino.

Como se puede ver la utilización de todas estas funciones es extremadamente sencilla. Más adelante se verán ejemplos de esto y de cómo se reduce mucho el código de los programas que usan esta interfaz en vez de la que se distribuye con el kit de desarrollo.

Cerrar canales

Siempre que se abre un canal, hay que tener especial cuidado en cerrarlo correctamente cuando se termine de utilizar. Dejar canales abiertos puede influir negativamente en el funcionamiento del programa, ya que no permitirá abrir ningún canal si este ya ha sido abierto previamente y no se ha cerrado. Además para cerrar el canal es necesario usar la función respectiva al tipo de transferencia para la cual se abrió el canal. De este modo, si nosotros abrimos el canal 0 para transferencia de bloques, tendremos que cerrar el canal con la función especifica para canales de bloques.

Se ha incluido por motivos de seguridad, un algoritmo que cierra los canales sean del tipo que sean en el destructor de la clase, para que nunca queden abiertos canales fuera del programa que los inició, ya que, de ser así, no se podrían abrir canales DMA en el dispositivo hasta que no se reiniciase el PC. Se recomienda al usuario de la interfaz que cierre los dispositivos por el mismo cuando termine de utilizar los canales para un optimo funcionamiento del dispositivo.

Las funciones encargadas de cerrar los canales DMA son las siguientes:

- **int CerrarCanalSglDma(int canal):** Cierra un canal que fuese abierto con *AbrirCanalSglDma*. Es necesario especificar el numero de canal. Si el canal especificado no esta abierto, esta ocupado o fue abierto para otro tipo de transferencias, la función devuelve un código de error.

- **int CerrarCanalBlockDma(int canal):** Cierra un canal abierto con *AbrirCanalBlockDma*. El funcionamiento es igual que la anterior. Si se el canal es 0, se dispondrá a cerrar el canal 0. En caso contrario cierra el 1.

- **int CerarCanalShuttleDma(int canal):** Esta función cierra un canal abierto con *AbrirCanalShuttleDma*.

A continuación se muestra un ejemplo de programa que realiza lecturas y escrituras tanto en DMA como en maestro/esclavo.

El primer paso es inicializar la tarjeta creando un objeto de tipo InterfazPlx, que será el que utilizaremos para llamar a las funciones de transferencias. Se supondrá que haymásde una tarjeta Plx conectada a nuestro PC para que se vea el caso más complicado. El siguiente trozo de código muestra un algoritmo que da a elegir que tarjeta de las que se encuentran conectadas al PC se desea abrir y la inicializa para poder empezar a trabajar con ella.

```
int num_disp;
int elec;
```

```
//INICIALIZACION DE LA TARJETA
InterfazPlx miTarjeta;

num_disp=InterfazPlx::NumeroDispositivos();

switch(num_disp){
case 0:
      printf("No hay ninguna tarjeta conectada en el PC");
      return;
      break;
case 1:
      printf("Sólo se ha encontrado un dispositivo");
      break;
default:
      DEVICE_LOCATION device;
      for (int i=1;i<=num_disp;i++){
            device = InterfazPlx::BuscaDispositivo(i-1);
            printf("%d: Bus= 0x%x; Ranura= 0x%x; Vendedor ID= 0x%x; ID
            Dispositivo= 0x%x\n",i, device.BusNumber, device.SlotNumber,
            device.VendorId,  device.DeviceId);
            }

      printf("Seleccione el dispositivo que desea abrir\n");
      cin>>elec;
      if(elec=0 || elec>num_disp){
            printf("ERROR: Debe introducir un numero entero entre 1 y
            %d.", num_disp);
            return;
      }
      else{
            InterfazPlx miTarjeta(InterfazPlx::BuscaDispositivo(elec));
      }
      break;
      }
```

Este trozo de código muestra todos los dispositivos que hay conectados por pantalla, da a elegir el que se quiere abrir y lo inicializa. En caso de que sólo haya una tarjeta, la abre directamente.

Si sabemos que sólo va a haber una tarjeta conectada a nuestro PC o bien queremos que se abra el primero de los dispositivos, basta con poner la línea *InterfazPlx miTarjeta;* en el programa para que este busque el dispositivo y lo inicialice.

Una vez se ha inicializado la variable miTarjeta ya podemos realizar transferencias. Por ejemplo para enviar una serie de datos por maestro/esclavo mediante un bucle y leerlos, se incluye el codigo.

```
U32 ibuff[1024];
U32 obuff[1024];
for (int j=0;j<1024; j++){
miTarjeta.EscrituraMS(0x100,&obuff[j]);
miTarjeta.LecturaMS(0x100,&ibuff[j]);

printf("Se ha transmitido %d y se ha recivido %d\n", obuff[j], ibuff[j]);
}
```

En este ejemplo se escribiría el dato en la posición 100 de la memoria RAM y a continuación se leería el dato ya procesado. Si se desea realizar esto mismo en DMA antes de iniciar la

transferencia hay que abrir el canal y posteriormente hacer el intercambio de datos. El siguiente código muestra la misma transferencia hecha en DMA.

```
miTarjeta.AbrirCanalSglDma(0);

for (j=0;j<1024; j++){
miTarjeta.EscrituraSglDma(0x200,1,&obuff[i],0);
miTarjeta.LecturaSglDma(0x200,1,&ibuff[i],0);

printf("Se ha transmitido %d y se ha recivido %d\n",obuff[j], ibuff[j]);
}
miTarjeta.CerrarCanal(0);
```

Cerrar el canal no es totalmente necesario, ya que al destruir el objeto cuando se acaba el programa, el destructor de la API se asegura que no quede ningún canal abierto, pero es muy recomendable cerrar los canales cuando se termina de trabajar con ellos para optimizar el funcionamiento y evitar posibles errores de programación.

7.2.1.2 Diseño del modulo aplicación.

El algoritmo realizado para la aplicación que correrá en el PC (Modulo Software de la Figura 7.4), se puede separar en dos partes. Por un lado para realizar el manejo del controlador conectado a nuestro Bus PCI (con el API explicada en el apartado anterior) (Figura 7.4 (1)), y la transferencia de datos, y por otro lado la manipulación de las imágenes que queremos enviar (Figura 7.4 (2)). Para trabajar con las imágenes utilizamos las librerías MIL, que nos permiten manipular las imágenes para el desarrollo de la aplicación.

7.2.1.2.1 Algoritmo para el manejo del controlador PLX conectado a nuestro Bus PCI.

1. Busca los dispositivos conectados al Bus PCI, obteniendo:

   a) el número de bus
   b) el número de ranura
   c) el ID del dispositivo
   d) el ID del Vendedor

2. Se abre el dispositivo seleccionado.
3. Reservamos espacio de memoria para el envío de nuestros datos.
4. Se abre un canal DMA para realizar la transferencia.
5. Realizamos la escritura de datos por el Bus PCI, en tamaños de 4- Lword.
6. Realizamos la lectura de los datos binarizados provenientes del Bus PCI, en tamaño de 4-Lword.
7. Cerramos el canal DMA y destruimos el objeto para cerrar el dispositivo.

7.2.1.2.2 Algoritmo para la manipulación de imágenes.

1. Reservamos espacio de memoria para la aplicación, el sistema, las imágenes y los displays donde se mostrarán.
2. Cargamos la imagen a enviar, y la mostramos en pantalla.
3. Convertimos nuestra imagen, en formato hexadecimal y la colocamos en una matriz de 1 columna para ser enviada.
4. Una vez devueltos los datos binarizados, los convertimos en imagen para poder ser mostrada por pantalla.
5. Eliminamos el espacio de memoria reservado para la aplicación, el sistema, las imágenes y los displays.

7.2.1.2.3 Soporte para las imágenes

Mil.h

Esta archivo de cabecera contiene todas las funciones y parámetros necesarios para la manipulación de las imágenes en nuestra aplicación, entre ellas cabe destacar:
MappAlloc, MsysAlloc, MdispAlloc MbufDiskInquire, MbufAllocColor Asigna espacio para una aplicación, sistema, dispositivo, y buffer definiendo sus parámetros. Estos deberán usarse por este orden a la hora de utilizarlos en la aplicación. Del mismo modo deberemos de eliminar los espacios reservados para estos en forma inversa, esto es; MbufFree, MdispFree, MsysFree, MappFree
MbufClear Limpia el buffer con un color.
MdispSelectWindow Selecciona un buffer de imagen para mostrarlo por una ventana definida por el usuario.
MbufLoad Carga un archivo de formato MIL desde un archivo a un buffer de datos.
MbufGet. Coge los datos de una imagen en formato MIL y los coloca en una matriz definida por el usuario.
Mbuf Put. Coge los datos desde una matriz definida por el usuario y los transforma en imagen en formato MIL.

Una vez mostradas las funciones MIL que utilizamos en nuestra aplicación, explicaremos de manera sencilla su funcionamiento:

Al iniciar la aplicación reservamos memoria para esta y el sistema para poder manejar las imágenes. Una vez seleccionada la imagen que queremos procesar, primero liberamos el espacio de memoria reservado para la imagen anterior en el caso de que hubiéramos cargado una previamente. Reservamos espacio para el display donde será mostrada la imagen, con los parámetros del sistema, el número de display, su formato, el modo de inicialización para ésta, y el lugar donde será almacenada. Del mismo reservamos espacio para el buffer donde será almacenada nuestra imagen. Con la función MbufDiskInquire, asignamos el alto y ancho de nuestra imagen y con MbufAllocColor le asignamos el color.
MdispSelectWindow se utiliza para asignar la imagen cargada a la ventana donde la queremos mostrar. Una vez terminados todos los preparativos para poder mostrar dicha imagen usamos MbufLoad y MbufGet para transformarla en una matriz de una columna, que será lo que escribiremos en el Bus PCI. El mismo proceso inversamente realizaremos cuando se lea del Bus PCI con MbufPut.
Al finalizar la aplicación debemos de eliminar el espacio reservado para la imagen en el orden inverso.

7.2.1.2.3 Flujograma de funcionamiento del software en el PC.

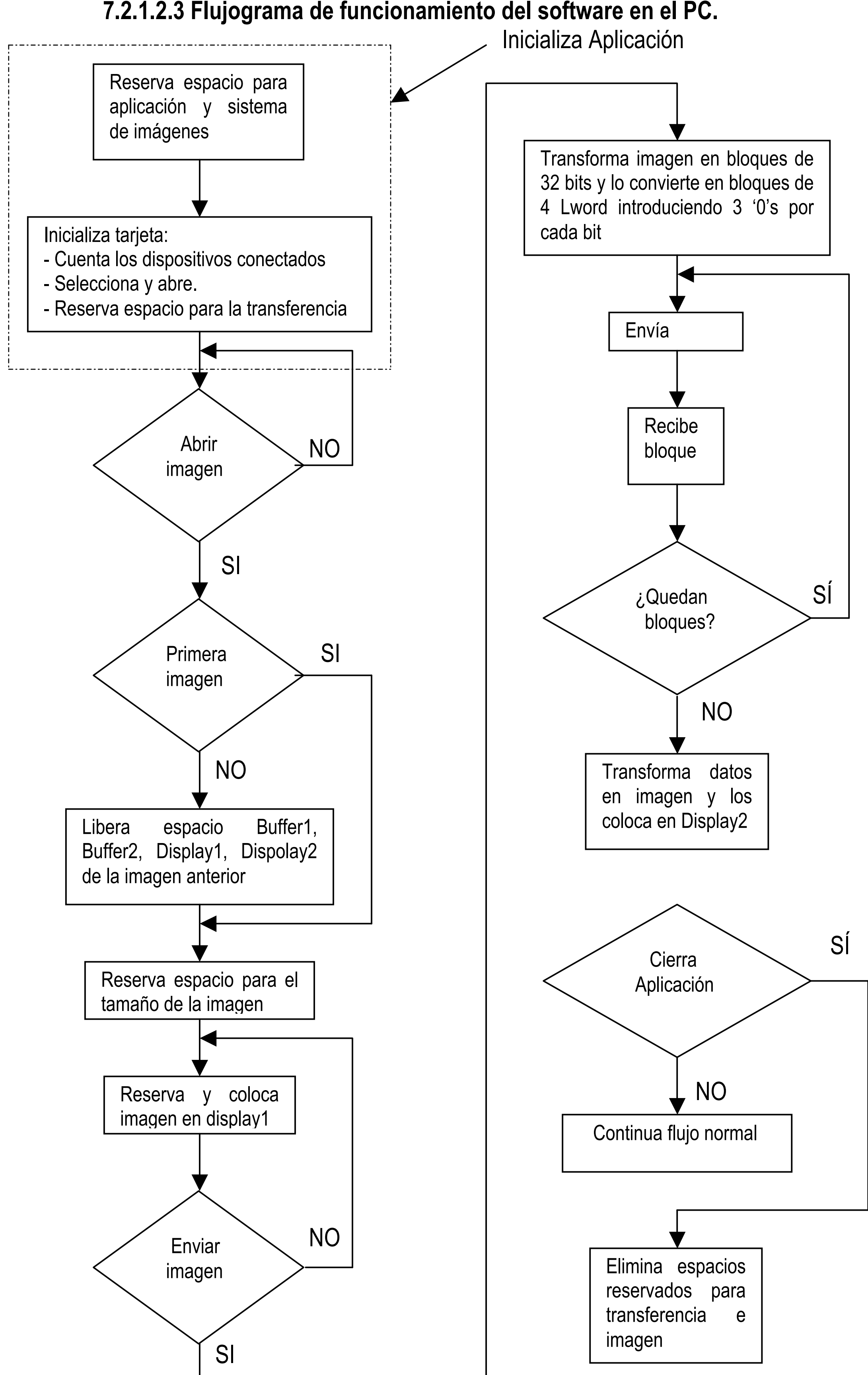


Figura 7.1 Flujograma del software realizado del PC (en Anexo C se encuentra el código completo)

7.2.2 MODULO ASOCIADO AL COMPONENTE HARDWARE

Los diseños lógicos realizados para la FPGA constituyen el elemento lógico de control de la tarjeta XS40 y su comunicación con el PC a través del Bus PCI mediante el controlador PLX9054. El diseñador podrá usar estos circuitos, junto con el software realizado en Visual C++ para implementar su aplicación particular.

Para ello, se han asignado todos los pines de control posibles a la FPGA, de manera que podamos cambiar el modo de transferencia y cambiar registros del controlador desde el bus Local si así lo deseamos o controlar las interrupciones.

Para el tipo de transferencia PCI Initiator, podemos usar las mismas líneas de control que para PCI Target, con la diferencia de que en ésta, tendremos que realizar el software necesario para que la FPGA actúe como maestro de la transferencia.

También utilizamos la FPGA, para configurar el reloj del Bus PCI a través del reloj de la tarjeta XS-40, mediante el software de XSTOOLS, GXSSETCLK. Para ello se han conectado ambas salidas de reloj, la de la FPGA y la del Bus Local de la tarjeta de prototipo, con lo que se modificaran ambos relojes ya que el reloj de la tarjeta PCI se ha eliminado para que ambas tarjetas trabajen con la misma frecuencia de reloj y estén perfectamente sincronizadas.

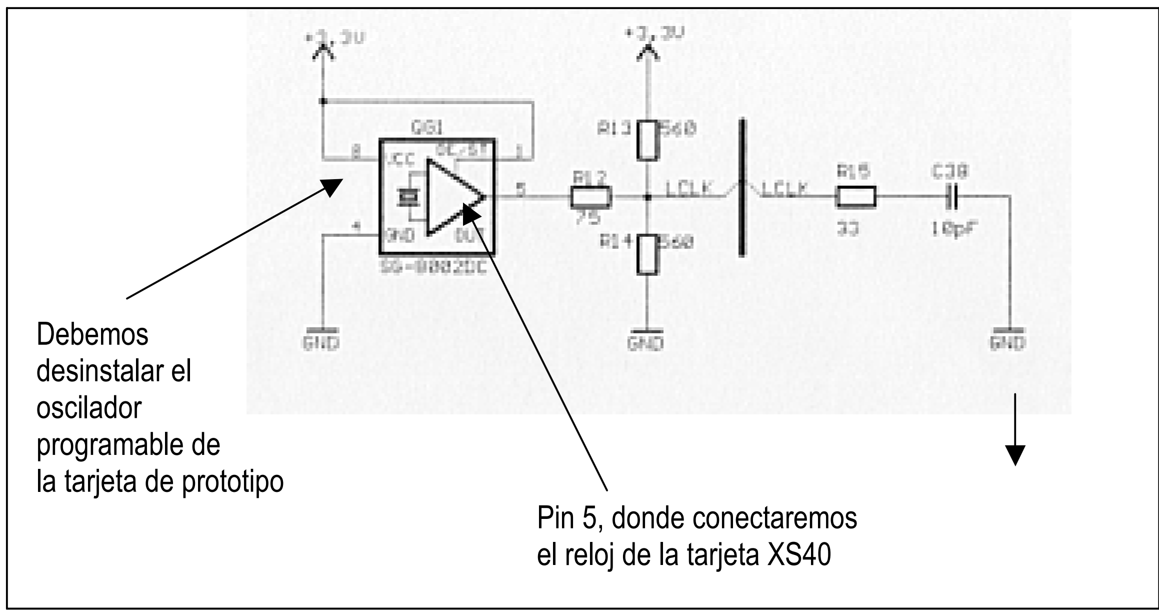


Figura 7.8 Oscilador programable de la tarjeta de prototipo PCI Proto Lab/ PLX

El diseño lógico de la FPGA se ha separado en dos bloques. En primer lugar se ha diseñado un modulo que haga de interfaz de comunicación con el controlador PLX y que posteriormente podrá ser utilizada en otros proyectos para el diseño de todo tipo de aplicaciones ya que acepta transferencias tanto DMA como esclavo directo (Figura 7.4 (3)). En segundo lugar se encuentra el modulo de binarización de imágenes que estará conectado al primero comunicándose con el PC a través de este (Figura 7.4 (4)).

El diseño de ambos componentes se ha realizado mediante el lenguaje de especificación hardware VHDL. Este lenguaje hace que sea mucho más sencillo y más rápido tener una visión clara del trabajo que realiza cada uno de los módulos. Además es un estándar muy utilizado en la actualidad. La conexión de los módulos entre sí y con los pines de las tarjetas se ha realizado con líneas de bus por esquemáticos para facilitar el entendimiento de todas las conexiones que tiene el diseño más fácilmente.

Se dispone también de una versión del modulo de interfaz de comunicación en esquemáticos perteneciente a un diseño anterior que después fue traducido a VHDL por los motivos explicados anteriormente. Este diseño funciona perfectamente del mismo modo que el diseño en VHDL. Tanto en el anexo C como en el CD se encuentra una copia de este diseño para el interesado.

7.2.2.1 Proceso de diseño

El proceso de diseño esta compuesto por las etapas de creación, verificación, simulación e implementación. Utilizando la herramienta del paquete de Xilinx Foundation es relativamente sencillo y no dista mucho del proceso seguido en la realización de cualquier programa en lenguaje de programación de alto nivel. Los pasos que debemos seguir para programar un circuito en la FPGA son los siguientes:

i. Realizamos un circuito usando las líneas necesarias para el control de la transferencia, así como los datos de la imagen que debemos procesar, estos serán introducidos en un modulo escrito en lenguaje VHDL, que incluye las librerías estándar IEEE, para binarizarlos

ii. También se conectaran directamente a la memoria SRAM de la tarjeta XS40 las direcciones donde guardaremos las transferencias 4 Lword.

iii. Configuramos el proceso de compilación del diseño; estilo de síntesis, opciones del dispositivo, etc., y simulamos el circuito para ver su correcto funcionamiento. En esta verificaremos la frecuencia máxima de operación del reloj de Bus Local.

iv. Una vez encontrado el correcto funcionamiento del sistema mediante las fases anteriores, pasamos a la compilación del circuito. La herramienta de Xilinx Foundation, posee un sencillo compilador que se divide en los siguientes pasos:

   - Translate: Traduce el archivo de diseño en un formato de base de datos interno de Xïlinx (NGD)
   - Implementation: Implementación del diseño compuesto por el mapeo del circuito su colocación en la FPGA y su ruteo
   - Timing: Verifica que los tiempos en el circuito ya implementado sean correctos
   - Configure: Genera un archivo de datos en cadena de bits que será el que se cargue en la FPGA (BIT).

v. Una vez obtenido el archivo BIT, lo cargamos en la FPGA de la tarjeta XS40, mediante el software GXSLOAD, seleccionando el puerto por donde se descargara, que en nuestra CPU, esta configurado como LPT2.

La parte más importante del proceso de diseño es la compilación. De esta parte depende la velocidad de operación que se obtenga y los recursos consumidos. En nuestra aplicación primamos la velocidad en contraposición con el espacio que ocupe en el dispositivo, debido al tamaño de nuestro circuito.

7.2.2.2 Descripción general

El diagrama de bloques del circuito de la FPGA se muestra en la figura 7.7. Este diagrama es general y representa el diagrama que se obtiene al combinar el diseño realizado para la aplicación con los elementos de la tarjeta XS40, sus entradas y sus salidas.

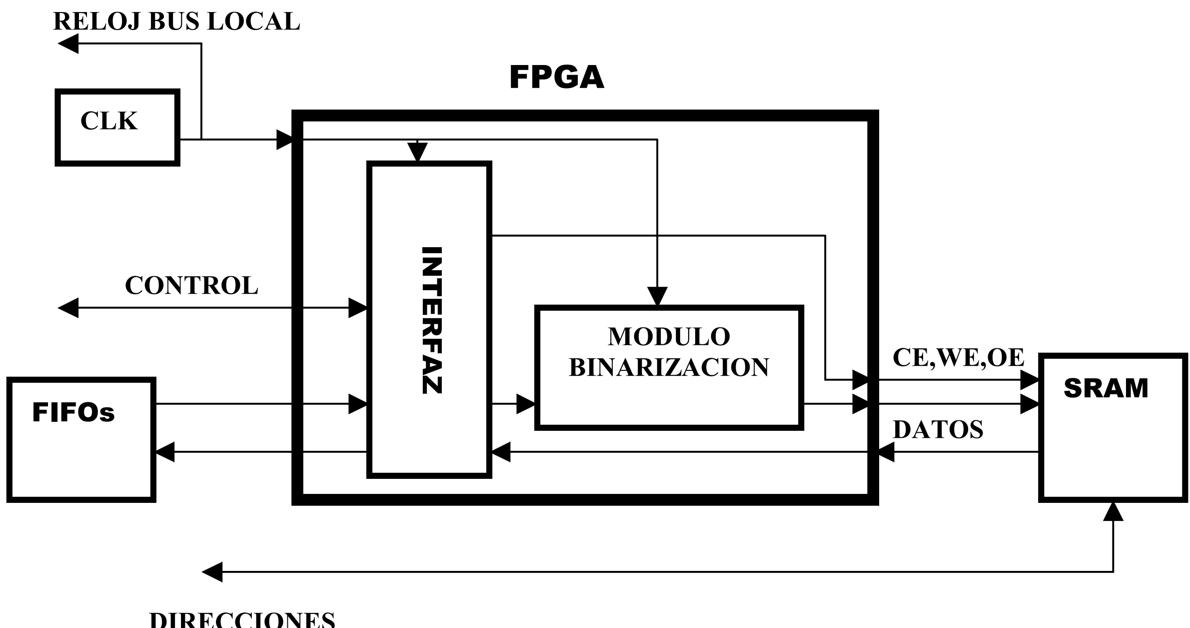


Figura 7.7 Diagrama de bloques general de la tarjeta XS40.

Cuando se produce un ciclo de escritura, la interfaz procesa las señales del protocolo del bus local y cuando se posee el control del bus y esta preparado, recibe los datos, que manda al modulo de binarización. Este procesa el dato y guarda el resultado de la operación en la memoria SRAM de la tarjeta XS-40. Cuando se produce un ciclo de lectura, la interfaz, que tiene conectada las señales de lectura/escritura a la SRAM, lee los datos anteriormente procesados directamente de la memoria y los envía a través del bus a la aplicación que los ha pedido.

Realmente esto no tiene porque ser necesariamente así, ya que también podría la interfaz mandar y pedir los datos al modulo de binarización con o sin ayuda de la SRAM y lo más aconsejable, que sería que el modulo binarización avisara a la interfaz cuando ha terminado de procesar los datos para enviárselos y que esta iniciase una transferencia DMA activando la señal DREQ0# con los datos procesado, pero esto sólo es posible con una tarjeta de 32 bits. A pesar de que en este

proyecto no se ha podido realizar esta última opción por falta de un ancho de bus de esas características, queda bien explicado para próximos proyecto como realizar estas transferencias en el capitulo dedicado al controlador.

Como se observa en el diagrama de la figura 7.7, las direcciones donde se guardaran los datos en la memoria SRAM vienen definidas por el bus de direcciones que viene directamente de la tarjeta Plx. Esto es un inconveniente, ya que es el usuario desde la aplicación del PC quien pasa este parámetro. El hecho de que se haya diseñado de esta manera no es otro que por falta de disponibilidad de pines en la tarjeta XS-40. En futuros diseños se debería conectar el bus de direcciones a la entrada de direcciones de la interfaz para leerlos a su salida, de esta manera se puede decidir que hacer con estas direcciones y si se desea conectar directamente a la memoria SRAM o bien realizar un procesado o una toma de decisiones a partir de la dirección dada por el usuario.

7.2.2.2.1 Modulo Interfaz

Este modulo interpreta todo el protocolo de funcionamiento del bus Local, haciendo transparente las transferencias a las aplicaciones que se conecten detrás de este. Todas las señales de control se conectan a este modulo para que interprete los comandos y señales y active las señales necesarias para iniciar transferencias. Una vez que la transferencia esta preparada, activa el bus de entrada o salida y la respectiva señal de lectura/escritura para que la aplicación envíe o lea los datos.

El funcionamiento del modulo es el siguiente:

- Cuando el controlador Plx quiere el control del bus Local activa la señal LHOLD para pedirlo.

- El modulo interfaz debe responder activando la señal LHOLDA para indicar al controlador que el bus Local esta disponible y cederle el control.

- Una vez hecho esto, el bus está asignado, y se deben empezar las transferencias. EL controlador indica que comienza un ciclo de dirección activando la señal ADS#.

- Mediante la señal LRW el controlador indica a la interfaz si el ciclo es de lectura o de escritura.

- Si el modulo interfaz esta preparado para la transmisión de los datos, activa la señal READY# e inmediatamente se comienzan a transmitir los datos. La señal READY# debe ir siempre retardada un pulso de reloj para indicar al controlador si se han recibido correctamente los últimos datos.

- En nuestro caso, el controlador esta configurado para que la señal que indica el fin de trama sea la señal BLAST#. También puede configurarse para que se active mediante la señal EOT#, pero es más aconsejable dejarlo con la configuración actual. Cada vez que se termine una transmisión a una dirección, el controlador activa BLAST#.

- La interfaz, cada vez que detecte la señal BLAST# debe desactivar READY# para indicar al controlador que los datos del siguiente pulso de reloj no serán recibidos.

- El bus sigue estando activo mientras las señales LHOLD y LHOLDA estén activas.

- Cuando la interfaz esta recibiendo datos válidos activa la señales de ENABLED para indicar al módulo al que esté conectado que se están recibiendo datos para que los procese.

- Junto con la señal de enabled también se activa la señal RW para indicar si el ciclo es de lectura o de escritura.

- Si el bus PCI esta mandando datos a mayor velocidad de lo que la FPGA es capaz de procesar, y se llenal los buffers intermedios, se puede activar la señal WAITL para que la interfaz informe de esto al bus PCI y pause la transferencia.

La pausa en sentido contrario no se ha habilitado porque actualmente es el bus PCI el que lee los datos del bus Local, por lo que no existe este problema. Sería interesante implementar esto en futuros trabajos, ya que cuando se disponga de un ancho de bus de 32 bits y se habilite el modo demanda de transferencia DMA, el bus Local podrá ser maestro de la transferencia y causar problemas de este tipo si se trabaja a diferentes frecuencias.

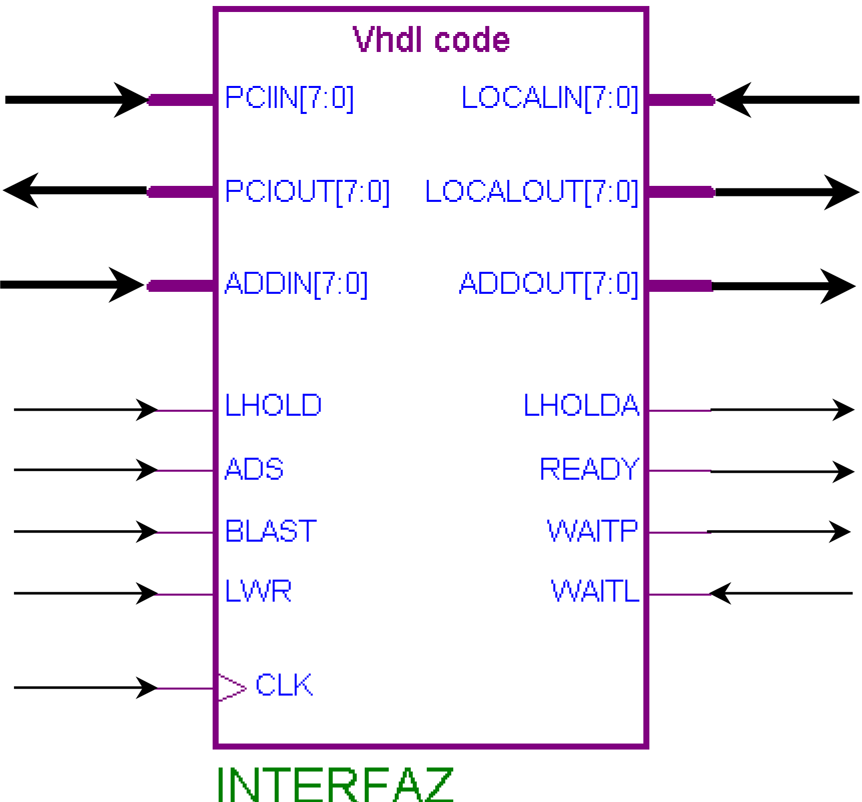


Figura 7.8: Modulo Interfaz

7.2.2.2.2 Modulo binarización

Este bloque tan sólo recibe los datos que le manda la interfaz, binarizandolos y escribiéndolos en la RAM. Los datos los lee la interfaz directamente de la RAM ya binarizados para mandarlos al usuario.

El componente se tiene una entrada de enable, un bus de datos de entrada de ocho bits y uno de salida también de ocho bits. También dispone de señal de reloj, conectada al mismo reloj de la interfaz y una entrada de lectura/escritura. Cuando se activa la señal enabled y la señal de lectura/escritura esta activa (ciclo de lectura), el componente lee los datos que se encuentran en el bus de entrada cuando se producen flancos de bajada del reloj y los compara con un valor umbral para decidir si el color que recibe (en la escala de grises entre el blanco y el negro) lo convierte en blanco o en negro. Los datos los lee en el flanco de bajada para asegurarse que los datos ya estan estables en el bus de entrada, ya que son puestos ahí por la interfaz en el flanco de subida. Cada vez que se lee un valor del bus de entrada, este valor representa a un píxel de la imagen a tratar, por lo que recibiremos tantos datos como píxeles contenga la imagen. Si el valor leído es menor que 200 en decimal el componente escribe en el bus de salida el valor “00000000” que corresponde con el color negro (ausencia de color). Si el valor que hay en el bus es mayor que ese mismo valor umbral entonces escribirá en la salida el valor “11111111” que corresponde con el color blanco puro (todos los colores). De esta manera, la imagen que se ha recibido en escala de grises (o 256 colores), se convierte en una imagen binaria en blanco y negro.

La señal de enabled del componente esta directamente conectada a la señal de ready del modulo interfaz, y la señal de lectura/escritura a la del mismo nombre de la interfaz. Esto hace que el modulo de binarización sólo trabaje cuando se encuentra en un ciclo de escritura. En caso contrario el componente estaría continuamente escribiendo datos a la salida y estos interferirían con los valores que se pretende leer de la RAM cuando se produce un ciclo de lectura. Para evitar este tipo de interferencias el componente pone a la salida el valor “11111111” cuando no se encuentra en un ciclo de escritura para que lo datos leídos de la RAM queden intactos. La señal de lectura/escritura está por defecto a ‘1’, pero esto no representa ningún problema, ya que aunque el componente esté procesando los datos de entrada, estos no son escritos en la memoria RAM porque sólo se activa la memoria en un ciclo de lectura/escritura (señal READY activada). Por ese motivo los datos no se escriben en la RAM, y a la hora de leerlo, la señal de lectura/escritura tiene el valor ‘0’ y el modulo no procesa datos.

Del mismo modo que se realiza un procesado de binarización de la imagen, se puede realizar cualquier otro tipo de procesado en este modulo. Varios ejemplos de procesados realizados en una FPGA de esta características pueden verse en el proyecto titulado “*DESARROLLO DE UNA ARQUITECTURA HARDWARE PARA LA IMPLEMENTACIÓN DE ALGORITMOS DE VISIÓN ARTIFICIAL*:” del proyecto final de carrera del autor Pedro Javier Navarro Lorente.

Esta plataforma está pensada para poder hacer conexiones de hardware de imagen a la tarjeta FPGA. Para ello se debería incluir un modulo de control del hardware de imagen a continuación de la interfaz y si se desea hacer un procesado de las misma o no. La interfaz puede leer perfectamente cualquier dato que se le pase por el bus de entrada local.

Figura 7.9: Modulo binarizador

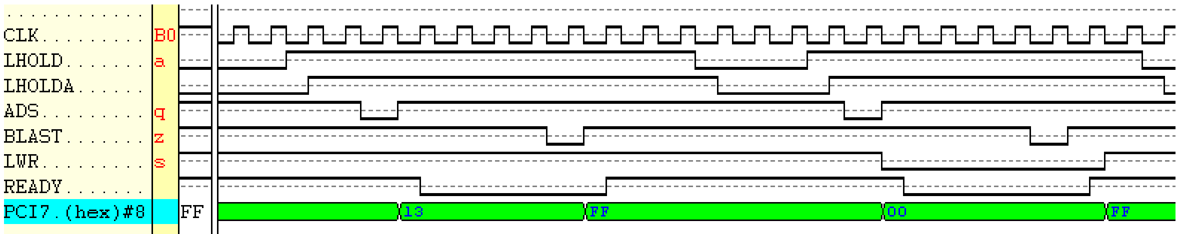


Figura 7.10: Ejemplo de transmisión y recepción

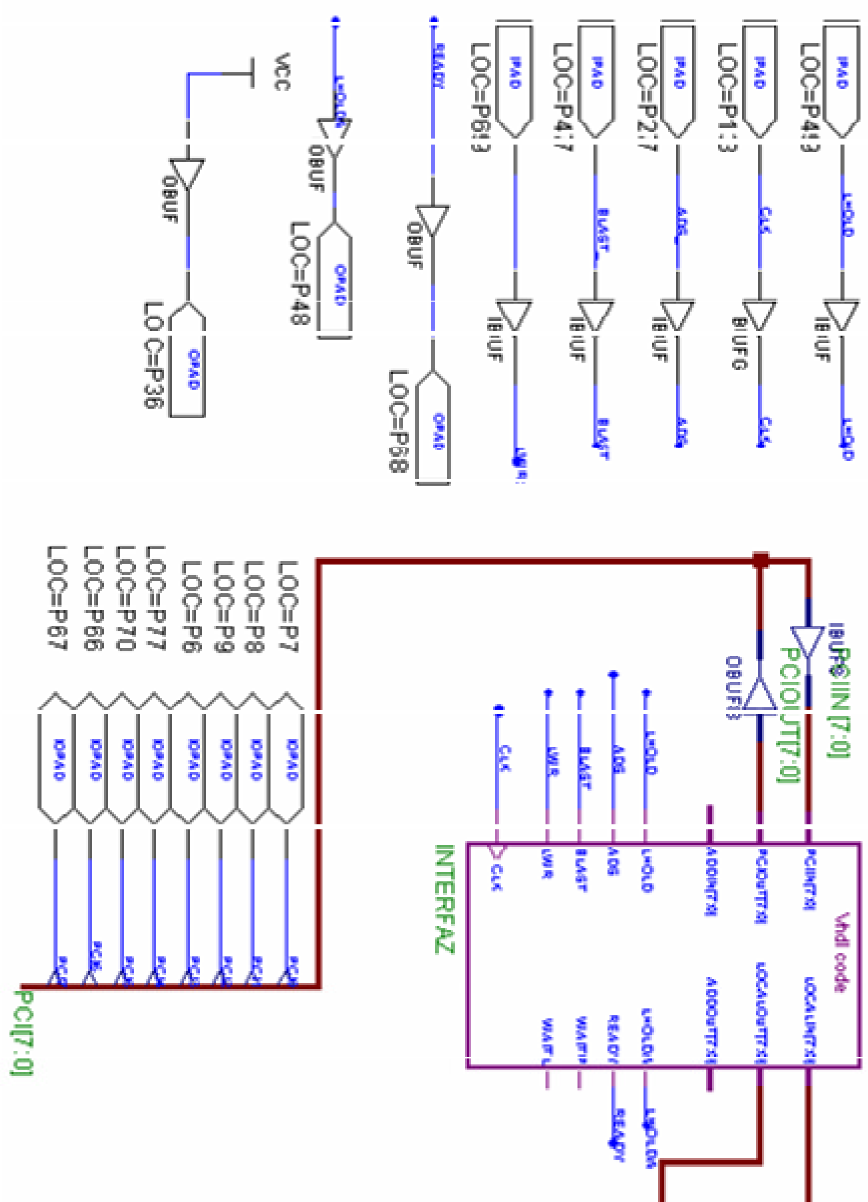


Figura 7.11: Esquemático general del diseño en la FPGA (parte 1)

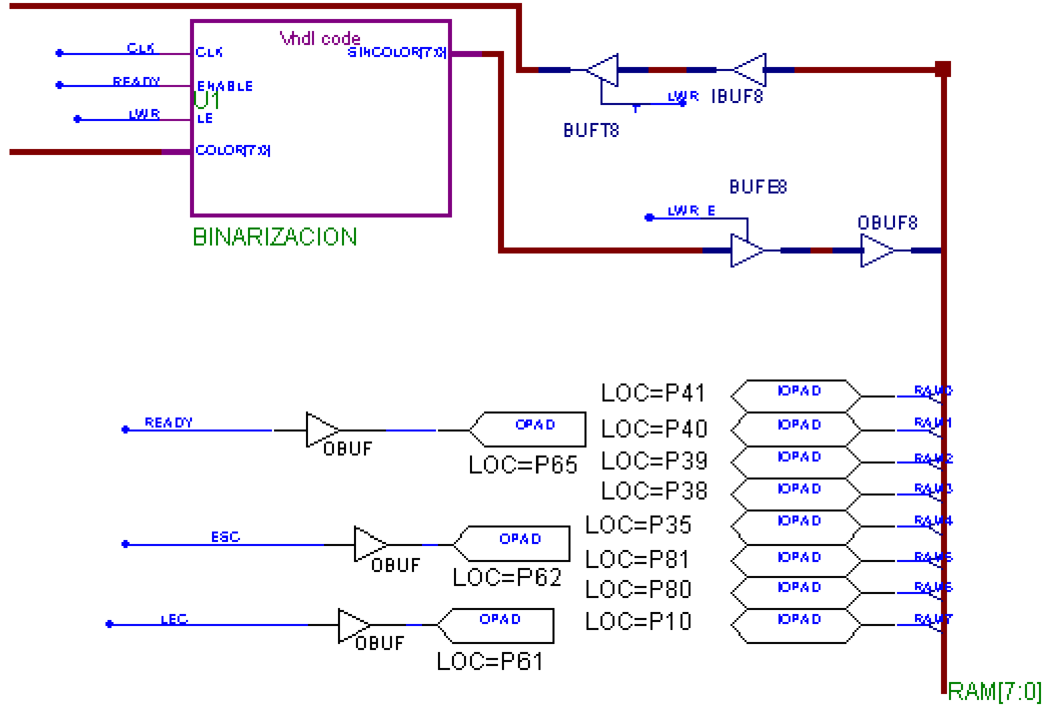


Figura 7.12: Esquemático general del diseño en la FPGA (Parte 2)

7.3 Precauciones de diseño

El diseño de tarjetas de alta frecuencia, es un factor importante a tener en cuenta y aunque en el presente proyecto no tengamos que crear ninguna tarjeta, debido a que nuestras tarjetas ya se encuentran diseñadas por el fabricante, cuidando este aspecto, encontramos de especial interés, comentar los problemas que presenta este tipo de transferencia, y exponer recomendaciones para evitar que estos ocurran, para una futura ampliación de este, en la que haya de realizar algún diseño.

La mayor parte de la literatura existente respecto al tema, considera que una tarjeta es susceptible a problemas debidos a su alta frecuencia de operación, cuando esta es superior a los 8-10 MHz. Por tanto, la tarjeta que conectamos al Bus Local se encuentra en esta categoría y deberemos diseñar su conexión cuidadosamente. Lo que se suele hacer en estos diseños de alta frecuencia, es recurrir a placas multiplaca que incorporan planos de masa intermedios, que apantallan las interferencias logrando una muy buena calidad, tal y como esta diseñada la tarjeta de prototipo.

De cualquier forma se tomarán todas las medidas posibles en la realización del PCB de conexión de ambas tarjetas, para reducir al máximo estos efectos.
A continuación se exponen los problemas más comunes que pueden surgir y algunas recomendaciones para el diseño del esquema eléctrico y del circuito impreso.

7.3.1 PRECAUCIONES AL CONECTAR HARDWARE DE ALTA FRECUENCIA.

En el diseño de alta frecuencia, toma relevancia el estudio de los componentes pasivos, es decir, el estudio de las inductancias presentes en las líneas de cobre de una placa y en los pines de los conectores. También es importante el estudio de las capacidades parásitas que aparecen entre dos líneas que llevan señales eléctricas y en los circuitos integrados, que poseen una pequeña capacidad parásita.

A baja frecuencia, ninguno de los componentes anteriores toma importancia, ya que su efecto depende de la velocidad de variación de las magnitudes de tensión y corriente (dV(t)/dt y dI(t)/dt).
A alta frecuencia si que comienzan a tomar valores significativos y a causar efectos importantes. Así pueden aparecer algunos problemas, destacando los siguientes:

7.3.1.1 Efectos de perturbación mutua entre líneas (cross-talk)

Dos líneas paralelas pueden perturbarse mutuamente en sistemas de alta frecuencia. Este efecto se divide en dos, atendiendo al tipo de perturbación.

- **Cross-talk capacitativo**.

Siempre que existan dos pistas de circuito que transportan señales, existirá una capacidad parásita entre ellas. La elevada velocidad de variación de la tensión en una de las líneas, puede provocar que circule una corriente a través de la capacidad parásita existente entre ambas pistas, perturbando a la segunda.

- **Cross-talk inductivo**

La elevada variación de la corriente en una de las pistas, genera un campo magnético que puede inducir una señal de tensión en la línea vecina.

En las placas de circuito impreso, las capacidades parásitas entre las pistas conductoras son pequeñas, por lo que el cross-talk capacitativo, no es relevante si se respeta siempre una pequeña distancia de separación entre las pistas de la placa. Por tanto, habrá que vigilar más las perturbaciones mutuas por efecto inductivo, generalmente de mayor impacto que las perturbaciones capacitativas

7.3.1.2 Reflexiones de las señales eléctricas.

Las señales que viajan por las líneas de circuito impreso, son generadas por una fuente y deben llegar a un destino (una carga) a través de las líneas de transmisión. Si las impedancias de la fuente de la línea de transmisión y de la carga son diferentes, se producen fenómenos de reflexión de las señales de tensión y corriente que viajan por la línea de transmisión, pudiendo aparecer señales reflejadas que se superponen a la señal que procede de la fuente y pueden llegar a cambiar su valor lógico, lo cual es inaceptable y puede provocar el funcionamiento inadecuado de toda la tarjeta.

7.3.1.3 Rebotes de masa.(ground bounces)

Los rebotes de masa son variaciones que se producen en el nivel de tensión de la masa, que debería ser constante para servir de referencia a los circuitos integrados. Se producen cuando en los sistemas digitales rápidos se produce una transición alto-bajo en muchas salidas de los circuitos integrados que estén en la placa. En este momento, los condensadores parásitos de los circuitos integrados situados entre las líneas de salida y masa, deben descargarse para que la salida pase a nivel bajo. Esta descarga se debe hacer a masa. Si conmutan muchas salidas a la vez y muy rápidamente, habrá una gran cantidad de corriente que circulará a la masa de los integrados. Como desde el pin de masa de los integrados hasta la pista de masa está presente una inductancia, cuando fluye gran cantidad de corriente en poco tiempo, aparecerá una tensión en la referencia de masa del integrado correspondiente, pudiendo afectar eso a su funcionamiento con la aparición momentánea de un valor de salida incorrecto.

7.3.1.4 Perturbaciones por radiaciones electromagnéticas.

Aparte de las radiaciones electromagnéticas procedentes de otros equipos o del entorno, las radiaciones electromagnéticas producidas por la propia tarjeta, son las que más afectan a su funcionamiento, ya que el foco de emisión es muy cercano. Cuanto más cerca esté el foco de emisión, más graves pueden ser las consecuencias.

7.3.1.5 Ruido de la fuente de alimentación

Las señales de alimentación que llegan a una tarjeta electrónica, suelen tener ruido de baja frecuencia (hasta unos poco KHz) que pueden introducirse a través de las líneas de alimentación en los circuitos integrados, alterando su funcionamiento.

7.3.2 RECOMENDACIONES PARA ALTA FRECUENCIA.

- Vistos los problemas, la principal recomendación seria usar placas multicapa, donde la alimentación llega a los integrados a través de planos enteros de VCC y GND. El que VCC y GND ocupen todo un plano provoca que exista muy poca inductancia y resistencia en el plano, de forma que los niveles de alimentación son constantes para todos los circuitos integrados de la placa y, además, se apantallan mejor las interferencias electromagnéticas. Pero estas tarjetas son costosas y requieren máquinas especiales para realizar un circuito con ellas.

En este caso, la placa a utilizar tiene dos caras de cobre. Obviamente, en un circuito de un tamaño medio/alto, es imposible efectuar todas las conexiones por una cara y dejar la otra cara para un eventual plano de masa, que sería lo ideal.

Lo que sí es necesario, es distribuir las señales de VCC Y GND por las dos caras de la placa, trazando estas líneas lo más anchas posibles. Ésto conseguirá reducir bastante las interferencias *cross-talk* de tipo inductivo y asegurará referencias de masa estables en todos los puntos de la placa. También consigue disminuir los efectos indeseados de los rebotes de masa. El ancho del resto de las pistas no ha de ser grande, ya que no se consigue mayor inmunidad al ruido. De esta forma, se pueden trazar las pistas estrechas para optimizar el tamaño del circuito impreso de la placa.

- Para evitar el ruido proveniente de la señal VCC, y para evitar que los circuitos integrados introduzcan ruido de alta frecuencia en esta línea, se deberá hacer lo siguiente:

  - Se situará lo más próximo posible al conector de alimentación de la placa, un condensador electrolítico de un valor en torno a los 100 μF. Si llegan varias líneas de alimentación, se debe de situar un condensador para cada una de ellas. Ésto no sólo filtrará el ruido de baja frecuencia que llegue, sino que además, servirá para proporcionar corriente cuando se requiera mucha cantidad, como cuando conmutan muchas salidas de circuitos integrados a la vez. De esa forma, el valor de la tensión de alimentación no sufrirá fluctuaciones importantes y permanecerá constante.

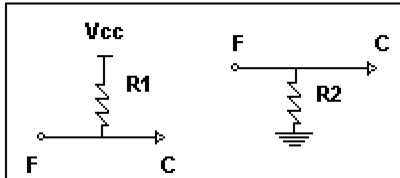
  - Se situarán condensadores de desacoplo de alimentación del valor recomendado por el fabricante, lo más cerca posible de los pines de VCC y GND de los circuitos integrados presentes en el circuito. De esa forma, se filtra el ruido de alta frecuencia que se puede introducir en las líneas de VCC y GND de la placa proveniente de los integrados.

Es importante trazar las pistas lo más cortas posibles para evitar los fenómenos de reflexión. Si se cree que una pista está sujeta a estos fenómenos, se puede considerar que la pista es una línea de transmisión y, por tanto, será preciso adaptarla para evitar problemas de reflexiones.

A continuación, se muestran los métodos de terminación de líneas que se usarán para adaptar las impedancias de la línea con la impedancia de la carga (del destino de la pista), ya que el problema de las reflexiones es que generalmente el valor de la impedancia de la fuente es bajo, menor que el de la línea de transmisión, siendo ésta menor todavía que la del destino de la línea. Por eso hay que realizar las adaptaciones, para que todas las impedancias tengan valores parecidos y no se produzcan reflexiones de la señal.

Se puede hacer la adaptación en la fuente, aumentando su impedancia de salida, o se puede hacer la adaptación en la carga, reduciendo su impedancia de entrada. Seguidamente, se describen los dos métodos de adaptación que se han utilizado en la tarjeta de evaluación.

- Terminación en paralelo: resistencias *de pulí-up y pull-down*



Es el tipo de adaptación más común en circuitos digitales TTL. Para circuitos CMOS no es recomendable usar este tipo de terminación debido a la elevada potencia que consume. Conviene que la resistencia en paralelo esté lo más próximo posible a la carga.

Figura 7.15.- Resistencia de pull-up *(der.)*
y pull-down (Izq.)

- Terminación Thevenin

La resistencia resultante del paralelo de la resistencia R1 y R2 coincide con el valor de la línea de transmisión, reduciendo la impedancia que presenta la carga. En la figura 7.16 se puede apreciar la configuración, donde 'F' es la fuente y 'C' es la carga. Conviene que el paralelo esté lo más próximo posible a la carga.

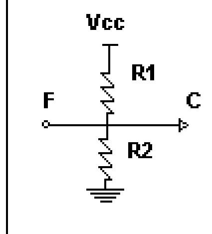
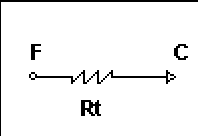


Figura 7.16.- Terminación Thevenin

- Terminación en serie

Aumenta la impedancia de la fuente con una resistencia en serie Rt. Como desventaja, aumenta la constante RC de la línea, reduciendo el nivel de las pendientes de subida y de bajada de la señal. Conviene que la resistencia Rt esté lo más próxima posible a la



fuente.

Figura 7.17.- Terminación Serie

- No hay que preocuparse demasiado por el efecto de los rebotes de masa cuando el circuito que hay en la placa opera de forma síncrona, ya que este tipo de perturbación es momentáneo y hay que tener en cuenta que en los circuitos síncronos las señales tienen que permanecer un tiempo estables antes del flanco de subida del reloj, tiempo de sobra para que ya haya pasado este efecto. Por tanto, no se suelen producir errores por los rebotes de masa en los circuitos síncronos, aunque sus efectos pueden ser perjudiciales en circuitos asíncronos que operen a alta frecuencia, ya que en los circuitos asíncronos, las salidas cambian su valor cuando cambian las señales de entrada, por lo que las señales espúreas causadas por los rebotes de masa podrían provocar que la señal de salida cambiase su valor cuando en realidad no debería.

- Muchas tarjetas tienen conectores de expansión para conectar señales del exterior a la tarjeta. Estos conectores son una verdadera fuente de perturbaciones a alta frecuencia. Por ejemplo, los conectores para cable paralelo plano de encapsulado tipo DIP, que son los más corrientes, llegan a operar sin problemas hasta las decenas de megahercios, pero a partir de ahí afectan al funcionamiento del circuito y a las señales que llegan a él. Para las tarjetas de alta frecuencia (desde 150 MHz hasta 30 GHz), se usan conectores que tienen un coste 100-1000 veces superior que el de los populares conectores de encapsulado tipo DIP. Los conectores introducen en la placa ruido c*ross-talk,* capacitivo e inductivo y, además, inducen perturbaciones electromagnéticas.

También hay que prestar atención a los cables que se usan para unir los conectores. Para conectar las señales de los conectores se suelen usar cables paralelos planos (como el que conectara ambos módulos), que son los que se usan en los ordenadores. Existen varias categorías de cables paralelos planos atendiendo a la frecuencia a la que deben trabajar.

Está claro que no se puede prescindir del uso de conectores y de cables. Para placas que no lleguen a 50 MHz no son tan graves las perturbaciones producidas por estos, aunque conviene tener presente algunas recomendaciones:

- Los conectores deben situarse en la placa lo más próximo posible a los circuitos integrados destinatarios de las señales que transportan.

- Conviene que los conectores estén cerca de alguna línea de masa. Además, generalmente, los conectores tienen entre sus líneas algunas de GND, que deben ser conectadas.

Por regla general cuanto más cortos sean los cables paralelos que se utilicen, mayor fiabilidad en la transmisión.

CAPÍTULO 8

Pruebas y resultados.

El punto de partida para la realización de éste proyecto final de carrera fue el proyecto [Rincón 01] en el que se que había realizado el estudio y la conexión de la tarjeta PCI Proto-Lab Plx y la tarjeta XS-40 de Xess. Por ello iniciamos las pruebas de nuestro proyecto instalando las tarjetas y haciendo las conexiones tal y como se explica en el mencionado proyecto, e inicialmente se optó por probar la aplicación realizada con la interfaz que se habían diseñado en [Rincón 01]. Esta prueba dió un resultado negativo y el PC se quedaba colgado cada vez que se intentaba iniciar una transferencia de cualquier tipo a la tarjeta XS-40.

Al encontrar este problema en la realización del proyecto, se optó por iniciar todas las pruebas desde el principio para localizar la fuente de dicho error. Para ello se reinstalo nuevamente la tarjeta como se explica en el Anexo B para posteriormente comprobar que la tarjeta había sido correctamente detectada por el PC mediante las opciones del panel de control de Windows.

Posteriormente, para comprobar que el problema no se encontrase en un fallo de la tarjeta Proto-Lab Plx se usó el programa Plx-Mon que es distribuido en el kit de desarrollo de la tarjeta para realizar transferencias a la misma y leer y escribir los registros de configuración de la tarjeta.
El manual de usuario de la aplicación Plx-Mon se encuentra en el Anexo A referido a manuales. Con esta aplicación se comprobó que la tarjeta funcionaba correctamente y que podía comunicarse perfectamente con el PC y realizar transferencia, así como leer los diferentes registros de la misma.

El siguiente pasó fue probar la tarjeta XS-40 para comprobar que funcionase correctamente, y para ello se realizaron diferentes aplicaciones muy simples en lenguaje VHDL . Una observación a tener muy en cuenta para futuros trabajos es que la configuración del puerto paralelo en la BIOS para la programación de la tarjeta XS-40 debe de ser en modo ECP para que todo funcione correctamente. Estas aplicaciones se valían del display de 7 segmentos del que dispone la tarjeta para comprobar que se programaba correctamente y se encendían. Posteriormente se realizó un decodificador que recibía los datos del puerto paralelo de la tarjeta y mostraba el correspondiente símbolo en el display. Esta prueba también fue satisfactoria y se comprobó con la misma que la tarjeta funcionaba correctamente.

Una vez verificado que ambas tarjetas funcionaban correctamente se comprobó el nivel físico de la plataforma. Para ello se testeó la continuidad de todas las pistas de la tarjeta de circuito impreso que sirve de interfaz entre ambas tarjetas y de los buses planos que las conectan, así como del zócalo que soporta y conecta la tarjeta XS-40, todo mediante el uso de un polímetro. Con ello se comprobó que algunos cables del bus plano no tenían una conexión optima, por lo que se fabricaron cables nuevos para todas las conexiones.

Al volver a probar la plataforma nuevamente, eliminando la CPLD y el reloj de la tarjeta prototipo y realizando todas las conexiones, el resultado obtenido fue idéntico a la primera vez, quedando el PC sin respuesta cada vez que se intentaba realizar una transferencia.

Llegados a este punto, se iniciaron las pruebas de la plataforma con todo el sistema montado en el interior del PC. Lo primero que se probó fue la conductividad de las pistas extremo a extremo, desde la tarjeta prototipo a los pines de la tarjeta XS-40 y se verificó que todas estaban en perfecto estado. Se comprobó que el reloj de la tarjeta XS-40 llegase correctamente a la tarjeta prototipo y que la primera estuviese alimentada correctamente a través de los pines GND, +5V y +3,3V de la XS-40. Todo esto, también estaba funcionando correctamente.

Al no encontrar fallos en ninguna de la conexiones ni en ninguna de las tarjetas por separado, se tomo la decisión de hacer las mismas pruebas que se realizaron con la XS-40 alimentada a través del conector de 9VDC, pero en este caso alimentada a través del PC. El resultado fue negativo. La tarjeta no se programaba correctamente cuando ésta era alimentada desde el PC. Por lo que la interfaz de comunicación con el PC y las respuestas de la misma al bus PCI no funcionaban correctamente impidiendo el correcto funcionamiento de la plataforma.

Para probar la programación de la XS-40 cuando ésta es alimentada a través de los pines de la misma, se usó una fuente de alimentación para conectarla a dichos pines y se programó la tarjeta con las mismas aplicaciones mencionadas anteriormente. Estas pruebas funcionaron correctamente y la tarjeta daba resultados favorables. También se hizo la prueba de programación con la tarjeta conectada al PC pero eliminando los cables de alimentación de los pines +3,3V y +5V y alimentando la tarjeta a través del conector 9VDC. La tarjeta no se programaba mediante el uso de este sistema.

Visto este resultado se coloco la tarjeta en el zócalo de la tarjeta de circuito impreso y se trato de programar sin estar conectada a la tarjeta prototipo, y se observo que las aplicaciones no funcionaban correctamente o simplemente no se programaban.

El nuevo paso fue fabricar una nueva tarjeta de circuito impreso para hacer de interfaz entre ambas tarjetas, ya que aunque ésta tenía continuidad perfecta en todas las pistas y parecía tener un estado optimo de todos los sentidos, la practica demostraba que fallaba en algún punto.

Con la nueva tarjeta de prototipo se volvieron a realizar todas las pruebas de programación con las aplicaciones de display y puerto paralelo y en este caso todas dieron resultados óptimos, funcionando todas las aplicaciones de manera satisfactoria. Así que nuevamente se programó con la interfaz para puerto Local en esquemáticos y la aplicación VHDL de binarización que se encuentran en el proyecto [Rincón 01], y mediante la aplicación en Visual C++ se intento enviar una imagen para ser procesada en la FPGA y el resultado fue idéntico que en anteriores ocasiones.

Para comprobar si el problema se encontraba en la parte de la aplicación del PC o de la interfaz de la tarjeta XS-40 se usaron los programas ejemplo para todo tipo de transferencias con el Bus Local que incorpora el programa Plx-Mon en el apartado Samples y el resultado fue el mismo que con la aplicación en Visual C++ de [Rincón 01].

Tras realizar un profundo estudio del protocolo de comunicación del controlador PLX 9054 y de la interfaz programada en esquemáticos se decidió realizar un nuevo diseño para esta interfaz que tuviese unas mejores prestaciones que la anterior. Este nuevo diseño, realizado mediante el uso de esquemáticos con el programa Xilinx Foundation, tuvo un funcionamiento óptimo. Este diseño ha sido sustituido posteriormente por uno nuevo en lenguaje de descripción hardware VHDL para mayor comodidad en la ampliación de futuros trabajos y para hacer de la plataforma un diseño más estándar, pero el diseño basado en esquemáticos se puede encontrar en el Anexo C dedicado a los códigos de las aplicaciones.

Para realizar el diseño de la nueva interfaz se crearon algunos programas de comunicación sencilla para probar el funcionamiento de las transferencias tanto en maestro/esclavo como en DMA tan sólo enviaban datos el PC y leían el resultado de la operación imprimiendo ambos por pantalla. El código de estos ejemplos se encuentra detallado en el Anexo C junto con los comentarios que acompañan al mismo para su mayos comprensión.

A continuación se muestran diferentes pruebas a realizar si se desea comprobar las diferentes prestaciones de la tarjeta prototipo que estaban incluidos en el proyecto [Rincón 01]. Algunas de estas pruebas hacen uso de la CPLD de la que dispone la tarjeta prototipo. En el capitulo 4 dedicado a esta tarjeta se explican los pasos para realización del cable de programación y los pasos para llevar a cabo la misma.

8.3 PRUEBAS.

En la primera de ellas, se utiliza el Bus de usuario, para comprobar la utilidad de la CPLD instalada en la tarjeta

En la segunda prueba, se conectó mediante cables, ambas tarjetas, estando la tarjeta XS40 fuera del PC, lo que nos obligaba a tener éste abierto. En ella, realizamos diferentes implementaciones en la FPGA.

En la tercera prueba, se diseñó una placa donde pinchar la tarjeta XS40, para poder instalarla dentro del PC, y partimos de una implementación creada en la etapa anterior para la FPGA.

La cuarta prueba, que es la utilizada para la aplicación final, no difiere mucho de la etapa anterior. Sus diferencias con ella, son una mejor disposición donde colocar los conectores que se unirán al Bus Local y una mejor disposición de la tarjeta XS40, se realiza un procesado de los datos transmitidos y se optimizan las señales de control a utilizar, para futuras aplicaciones.

8.3.1 PRUEBA I.

Encontramos en este bloque, las pruebas que se realizaron mediante el Bus de Usuario, utilizando la CPLD, que van desde la conexión directa de los pines de entrada y salida de este Bus (recordamos que el Bus de usuario, presentaba pines de datos de entrada y de salida que eran divididos por la CPLD a través de los latches instalados en la tarjeta, véase capítulo acerca de la tarjeta de prototipo).

Para el uso de la primera y más simple prueba se conectaron cuatro pines de salida directamente a cuatro pines de entrada del Bus de Usuario, y utilizando el programa que acompaña al kit SDK, Dslave.exe, comprobamos buen funcionamiento de la tarjeta observando que los datos escritos son leídos correctamente como esperábamos.

La siguiente etapa que se realizó en esta prueba fue utilizar la FPGA de la tarjeta XS40 para probar algún procesamiento simple con estos datos. Entre ellos destacar una prueba en la cual se diseñaban cuatro flip flops que serían atravesados por un bit de datos (D0), para luego ser devuelto por el bit de datos D4.

El siguiente paso fue utilizar 4 bits de datos de salida que serían introducidos en la FPGA la cual disponía de un esquema lógico compuesto por varias puertas AND y OR para ser devueltos por los datos de entrada al Bus de Usuario, tal y como muestra la figura 8.5.

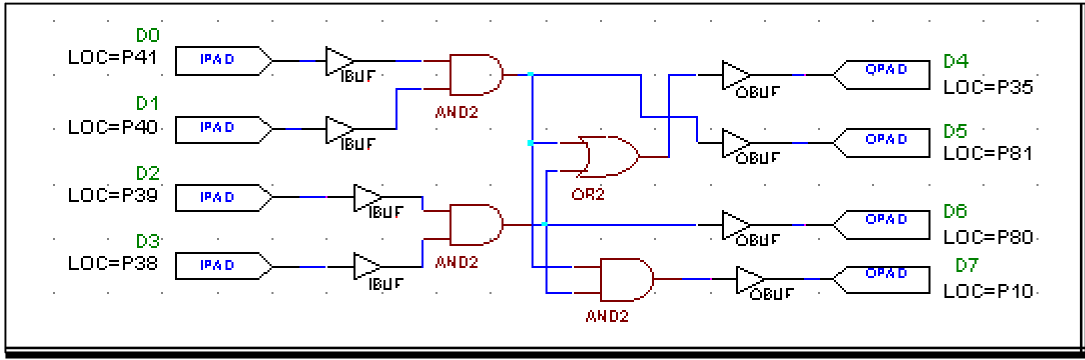


Figura 8.5 Esquema de la prueba 1.2

El último paso que se realizo utilizando el Bus de Usuario, fue mediante el uso de una memoria RAM implementada en la FPGA, la cual poseía 2 bits de datos y 4 bits de direcciones. Para ello utilizamos 2 bits del bus de datos de salida de usuario (Dout0-Dout1) para los datos de la memoria y 2 bits del mismo bus para las direcciones bajas (Dout2-Dout3). Así mismo, los dos bits de salida de la memoria, los conectamos a los bits de entrada del Bus de Usuario (Din0-Din1). Ver figura 8.6.

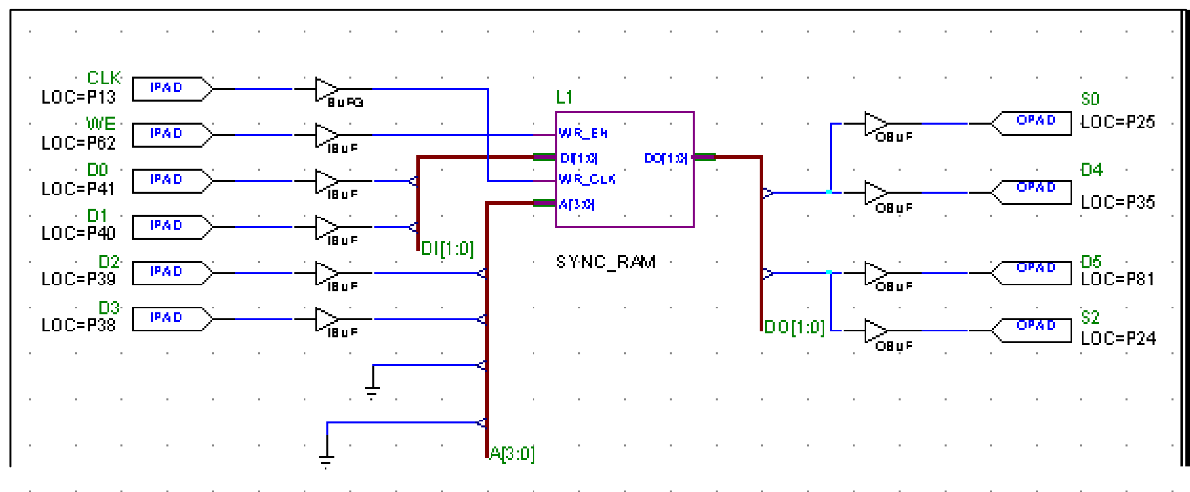


Figura 8.6 Esquema de la prueba 1.3.

Para la realización de estas dos últimas pruebas se hizo un software en el PC, en el entorno Visual C++, el cual manda un 1 o un 0 dependiendo si la casilla correspondiente al dato esta marcada. En la figura 8.7 se puede ver la pantalla principal del programa realizado para estas pruebas. En ésta, se esta utilizando el esquema de la prueba con las puertas lógicas. Éste programa se puede utilizar indistintamente con ambos diseños de la FPGA.

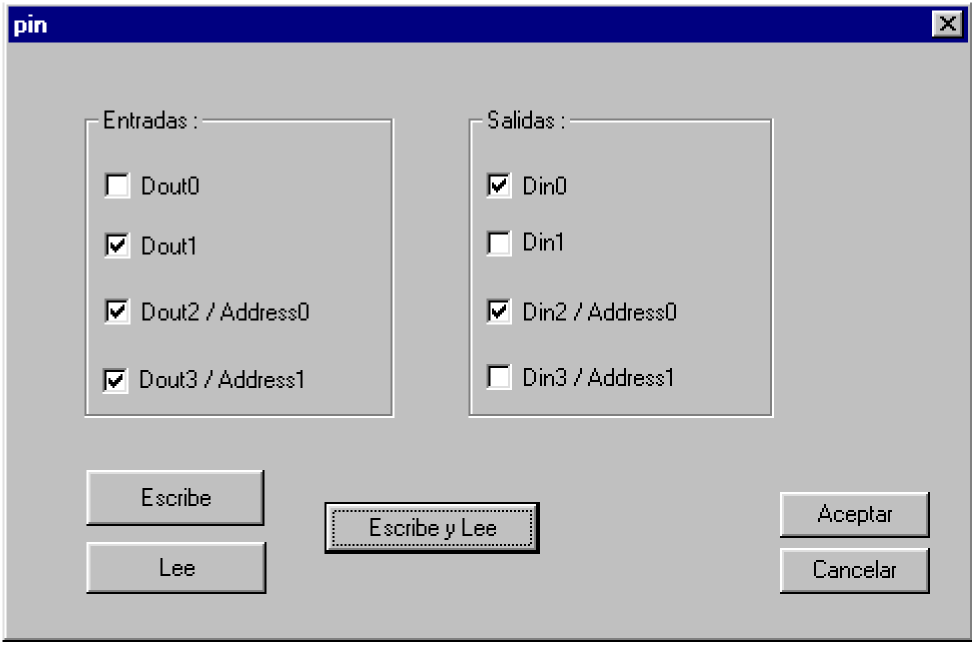


Figura 8.7 Pantalla principal de la aplicación realizada para las pruebas 1.2 y 1.3

```
void CPinDlg::EscribeSalidas()
{
RETURN_CODE rc;
U32 dato1;
UpdateData(true);
dato1=0x0000;

if (m_D0==TRUE)
		dato1= dato1  | 0x0001;
if (m_D1==TRUE)
		dato1= dato1  | 0x0002;
if (m_D2==TRUE)
		dato1= dato1  | 0x0004;
if (m_D3==TRUE)
		dato1= dato1  | 0x0008;

rc = PlxBusIopWrite(myPlxDevice,
				IopSpace0,
				0,
				TRUE,
				(PU32)(&dato1),
				1,
				BitSize8);

if (rc != ApiSuccess)
{
AfxMessageBox("Error: Unable to write data");
return;
}
UpdateData(false);
}
```

```
void CPinDlg::LeeEntradas()
{
RETURN_CODE rc;
U32 dato2;
UpdateData(true);
dato2=0x00000000;
rc = PlxBusIopRead(myPlxDevice,
				IopSpace0,
				0,
				TRUE,       /* remap */
				(PU32)(&dato2),
				1,
				BitSize8);
if (rc != ApiSuccess)
{
AfxMessageBox("Error: Unable to read data.\n");
return;
}
if ((dato2 & 0x0001) == 0x0001)
		m_D4 =true;
else
		m_D4 =false;
if ((dato2 & 0x0002) == 0x0002)
		m_D5 =true;
else
		m_D5 =false;
if ((dato2 & 0x0004) == 0x0004)
		m_D6 =true;
else
		m_D6 =false;
if ((dato2 & 0x0008) == 0x0008)
		m_D7 =true;
else
		m_D7 =false;
UpdateData(false);
```

Cuadro 8.1 Algoritmos de escritura y lectura para la aplicación de las pruebas 1.2 y 1.3

8.3.2 PRUEBA II.

En esta prueba se conectaron 8 bits del Bus de Datos Local, 8 bits del Bus de direcciones del Bus de Local de direcciones. En ésta usamos las mínimas líneas de control para poder llevar a cabo la transferencia PCI Target, en donde como comentamos anteriormente, es el PC el que actúa como maestro, siendo la tarjeta XS40 la que actúa como esclava de la transferencia.

En la figura 8.8, puede verse la disposición que presentaba este conexionado, bastante engorroso de manejar, y con el inconveniente de presentar ruido en la transmisión, debido a la longitud de las conexiones.

En esta prueba se realizaron diversos diseños para implementar en la FPGA, de los cuales pasaremos a comentar los más importantes.

En una de las implementaciones realizadas en la FPGA, se usaba una memoria RAM de 8 datos y 8 líneas de dirección. Las líneas de control que usamos en esta implementación son LHOLD, LHOLDA, ADS#, BLAST#, LW/R# y READY#.

Conectados los 8 bits bajos del Bus Local con los bits de datos de entrada de la FPGA y las 8 líneas de dirección bajas del Bus Local de direcciones con las 8 líneas bajas de dirección de la tarjeta XS40, sin estar habilitada la memoria de ésta, ya que se utilizaba la implementada en la FPGA, tal y como muestra la figura 8.8.

Para esta prueba se utiliza el programa realizado en Visual C++ en modo consola, cuyo principal algoritmo se presenta en el cuadro 8.2. En éste se realiza una escritura y lectura 4 Lword, hasta terminar la frase escrita.

```
printf("Enter the address of the local buffer (in hex):");
scanf( "%x", &localStartOffset );
strcpy(frase,"Hola Mundo, esta es una frase que será
enviada por el bus PCI, que tal lo pasaste anoche,
Marcos");
printf("La frase que voy a enviar es:%s\n",frase);
DoDSTest(
IN HANDLE drvHandle,
U32 localStartOffset,      /*The IOP local buffer*/
U32 totalSize,             /*Buffer length in 8 bits cells.*/
IN ACCESS_TYPE accessType, /*Type of memory
access 8,16 or 32 bits*/
PU32 bufFrom,              /*The starting host PCI buffer*/
PU32 bufTo                 /*The resulting host PCI buffer*/
){
        RETURN_CODE rc;
        U32 value=0;
        U32 length = totalSize;
        PU32 buf;
        S8 ch;
        char* temp;
        for (i=0;i<=l;i++)
                printf("%x,%x\n",frase[i],l);
        temp=malloc(l*4);
        for (i=0;i<=l;i++)
        {
                temp[i*4]=frase[i];
        for (j=1;j<4;j++)
                        temp[i*4+j]=0;
        }
        for (i=0;i<l*4;i++)
                printf("%x",temp[i] & 0xFF);
        printf("\n");
```

```
    rc = PlxBusIopWrite(drvHandle,
                        IopSpace0,
                        localStartOffset,
                        TRUE,
                        (PU32)(temp),
                        l*4,
                        BitSize32);
    if (rc != ApiSuccess)
    {
        printf("Error: Unable to write data.\n");
        free (temp);//delete temp;
        return -1;
    }
    rc = PlxBusIopRead(drvHandle,
                        IopSpace0,
                        localStartOffset,
                        TRUE,       // remap
                        (PU32)(bufTo),
                        l*4,
                        BitSize32);
    if (rc != ApiSuccess)
    {
        printf("Error: Unable to read data.\n");
        free (temp);//delete temp;
        return -1;
    }
    printf("Dato = %x, Valor = %x \n",i,bufTo[i]);
    for(i=0;i<=l;i++)
        printf("Dato = %x, Valor = %c ",i, bufTo[i]);
    printf("\n");
    for(i=0;i<l;i++)
        printf("%c",bufTo[i]);//bufTo[i]);
}
```

Cuadro 8.2 Algoritmo de escritura y lectura para la aplicación de las pruebas 2.1

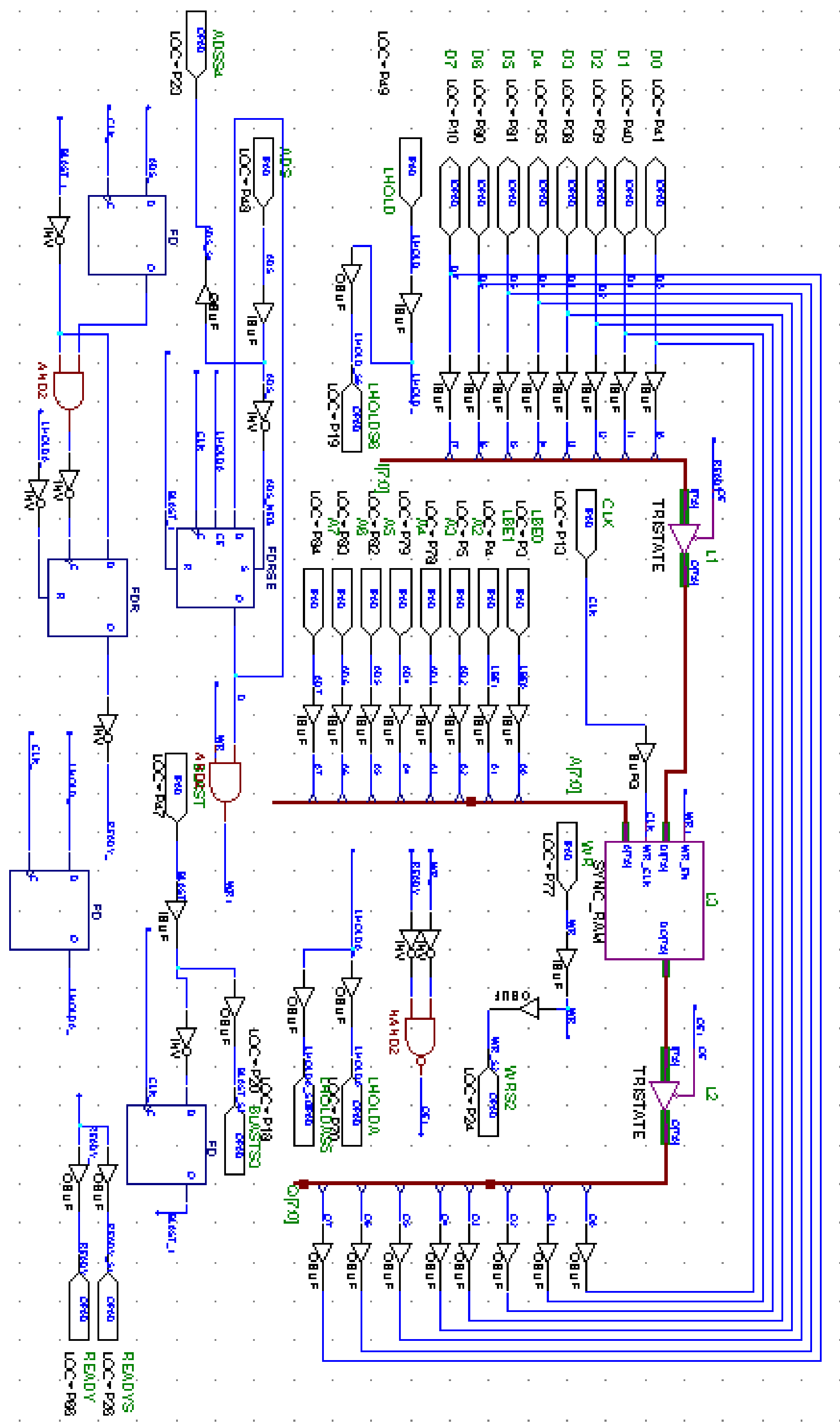


Figura 8.8 Esquema de la prueba 2.1

8.3.3 PRUEBA III.

El conjunto de pruebas que abarca este bloque, se realizaron con la primera tarjeta prototipo montada sobre la tarjeta de Bus PCI, en la cuál ya no aparecen ruidos al ser los cables de conexión mucho más cortos.

Una de las pruebas a resaltar, es la realización del mismo diseño anterior, con la diferencia que los datos y las direcciones están conectados directamente a la memoria SRAM de la tarjeta XS40, por lo que en la FPGA, sólo tendríamos que preocuparnos de la manipulación de las señales de control, tal y como muestra la figura 8.9. El software realizado para manejar las transferencias, es muy similar al utilizado en la aplicación final, por lo que no lo expondremos en este apartado, siendo mostrado en el punto siguiente.

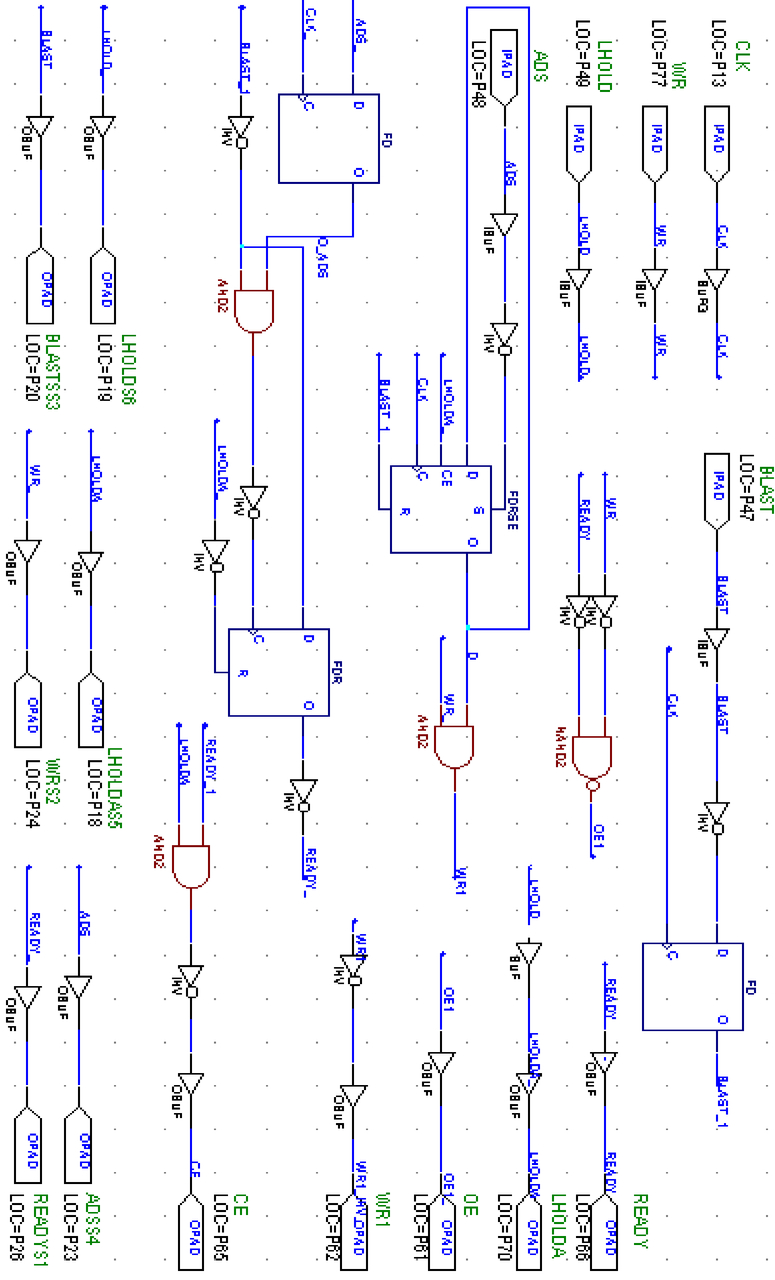


Figura 8.9 Esquema de la prueba 3.1

8.3.4 PRUEBA IV.

Se engloba el último bloque de pruebas, a las realizadas en la segunda tarjeta. En ésta las direcciones continúan conectadas directamente a la memoria SRAM de la tarjeta XS40. Sin embargo, los bits del Bus Local de datos atraviesan la FPGA para ser sometidos a un procesamiento de binarización antes de guardarlos en la memoria.

Las señales de control aquí usadas, son las mismas a las anteriores, añadiéndole otras señales que controlen nuestra aplicación y otras que flexibilice la tarjeta realizada para futuras aplicaciones en las que se quiera realizar otro tipo de transferencias.

Para estas pruebas se usaron en primer lugar la interfaz diseñada mediante esquemáticos y posteriormente con la interfaz VHDL. Conectado a esta interfaz se encuentra el modulo de binarización de imágenes que se detalla en el capitulo 7.

En la parte del PC se uso en primer lugar la aplicación diseñada en el proyecto [Rincón 01] y al tener un resultado favorable se pasó a usar la API diseñada en este proyecto para comprobar el funcionamiento correcto de ambas para estas transmisiones.

Por diferentes problemas de última hora con el hardware, no se pudo incluir en el texto los resultados de las pruebas de tiempo, que a pesar de eso es estima que al menos 100 veces menores que las pruebas realizadas en el anterior proyecto con transferencias maestro/esclavo. Estas pruebas, junto con los resultados de tiempos y los datos de ocupación y de timing de los diseños, se incluirán en el CD del proyecto.

CAPÍTULO 9

CONCLUSIONES Y FUTUROS TRABAJOS.

En este último capítulo, se presentan las conclusiones obtenidas tras el estudio de los diferentes elementos expuestos en los capítulos precedentes y la interacción entre ellos, atendiendo a las ventajas e inconvenientes que presenta.

Asi mismo efectuaremos algunas recomendaciones que podrán llevar al lector por futuras lineas de trabajo, útiles, para realizar con esta plataforma-

9.1 CONCLUSIONES

Una vez terminado la presentación de todo el proyecto, se dispone de una plataforma reconfigurable de prototipado rápido modular y reutilizable que permite realizar una amplia gama de aplicaciones de tratamiento de imágenes y visión artificial, y lo que es más importante, fija unos firmes cimientos para la utilización de las diferentes interfaces con nuevo hardware de mayor calidad que permitirá realizar cualquier procesado a nivel profesional, tasas de transferencias de gran velocidad a un precio muy bajo con una facilidad de uso y de implementación que permite que usuarios poco avanzados puedan realizar aplicaciones de manera fácil y rápida y usuarios avanzados puedan trabajar con altas prestaciones.

Los objetivos principales, de disponer de una interfaz de comunicaciones con el Bus Local en VHDL y una API para el desarrollo de aplicaciones para la plataforma, han sido satisfechos de manera eficiente.

La API diseñada, permite realizar transferencias tanto en modo maestro/esclavo como en DMA a través del bus PCI, y configurar la tarjeta para trabajar de modo deseado. Esta API, implementada en Visual C++, es de gran utilidad y facilita enormemente la programación de aplicaciones gracias a la programación orientada a objetos.

Tiene en cuenta posibles ampliaciones de la plataforma y permite trabajar con diferentes anchos de bus.

Se han simplificado las operaciones que había que realizar anteriormente para que la facilidad de uso sea una de sus características más importantes, sin menospreciar en absoluto la robustez, la fiabilidad y las prestaciones de la misma. Esta es una de sus características más, ya que al contrario de lo que ofrece el fabricante (en la que se basa nuestra interfaz) es muy fácil de utilizar y permite, a cualquier persona con conocimientos básicos de C++, hacer transferencias sin problemas con programas de lo más simple (menos de 10 líneas de código). Esto dará lugar a que quien la utilice pueda centrar su atención en la aplicación que esté desarrollando y no emplear un tiempo considerable en programar las transferencias.

También tiene la capacidad de ser fácilmente ampliable con nuevas características para usuarios más avanzados.

La interfaz diseñada para la interpretación del protocolo que el controlador Plx aplica al bus Local también ha sido conseguida de manera eficaz.

Esta interfaz, de la que se dispone tanto en VHDL como en esquemáticos, permite conectar cualquier aplicación que se programe en la FPGA o cualquier hardware de imagen con el bus PCI sin necesidad de incluir en la aplicación el protocolo del bus Local permitiendo comunicación de gran velocidad con el PC a través de este puerto de manera transparente, al usuario y a la aplicación. La interfaz es flexible permitiendo el uso de diferentes anchos de bus o diferentes FPGAs. También se ha preparado para que en futuros trabajos sea fácilmente ampliable.

Las ampliaciones realizadas en ambos extremos, han constituido el ultimo paso en el diseño del sistema, y gracias a la previa preparación de las interfaces, antes mencionadas, ha sido muy fácil este último paso. De esta manera se optó por un proceso de binarización de imágenes como

ejemplo de utilización de las interfaces y como demostración de sus posibles aplicaciones, funcionamiento y características principales. La transferencia de las imágenes se hace en DMA por ser el modo de transferencia que permite mayores velocidades en el bus PCI.

9.1.1 soporte físico empleado:

- Puerto PCI
- XS 40
- PLX 9054

9.1.1.1 Puerto PCI

Este tipo de puerto es el que utiliza nuestra tarjeta y ofrece prestaciones suficientes como para que se puedan implementar aplicaciones que requieran gran trasiego de datos, ya que ofrece unas velocidades de transferencia elevadas.

Otra ventaja que se puede deducir a simple vista sobre este puerto, es que es uno de los más extendidos, lo que impedirá que caiga en desuso, siendo presumiblemente de fácil acceso para cualquiera, incluso tras el paso del tiempo, lo que hace que nuestro sistema no pueda quedar fácilmente obsoleto.

9.1.1.2 XS 40

Esta tarjeta pertenece a la familia de tarjetas de lógica programable, característica que ha permitido utilizar potentes herramientas de diseño y también hacer pruebas y corregir los errores cometidos de forma rápida y eficaz.

En cuanto a este modelo en sí, queda un poco corto de prestaciones para este sistema, básicamente porque el bus de datos que utiliza es de 8 bits siendo el de la PLX 9054 de 32. Ésto hace que se desperdicie la mayor parte del potencial del sistema además de impedir que se puedan llevar a cabo cambios en los registros de configuración de la tarjeta PLX (necesita órdenes de 32 bits).

Esta tarjeta también podría quedar fácilmente escasa de pines de entrada/salida ya que muchos son utilizados para el manejo de señales y el bus de datos, contando con que ya hay muchos reservados para la RAM y para las direcciones de la misma, quedando pocos para la aplicación propiamente dicha.

9.1.1.3 PLX 9054

La tarjeta de prototipo PCI Proto-Lab/ PLX, nos ofrece una cómoda forma de comunicar un dispositivo hardware con el PC a través del Bus PCI, permitiendo unas elevadas velocidades de transferencia entre ambas partes.

Con el uso de esta tarjeta no debemos preocuparnos del protocolo de comunicación del controlador con el Bus PCI y así poder centrarnos en las líneas de control que el PLX9054 proporciona en el Bus Local.

8.2.3 Conclusión general

Se han conseguido muchas mejoras que han hecho posible realizar transferencias y controlar aplicaciones a través del puerto PCI. Los resultados obtenidos, que han sido satisfactorios, dan lugar a numerosas ampliaciones, tanto software (modo demanda) como hardware (sustitución de tarjetas por unas con mayores prestaciones), pudiéndose llegar a un sistema con mejores prestaciones si se emplean tarjetas de mayor velocidad ymásbits de dirección.

El tener la posibilidad de una comunicación sencilla y configurable por el puerto PCI, da lugar a incontables posibilidades de aplicación del sistema.

9.2 FUTUROS TRABAJOS

A continuación se comentarán algunas recomendaciones que pueden ser interesantes, y posibles líneas de trabajo, para un futuro diseñador que utilice la plataforma creada en este proyecto para implementar sus propias aplicaciones o que desee mejorar las prestaciones de la plataforma ampliando las capacidades de ambas interfaces.

En primer lugar, como mejora más importante, se recomienda sustituir la tarjeta XS-40 por una de mayores y mejores prestaciones. Este cambio no sólo es recomendable, si no que casi es necesario para realizar nuevas aplicaciones de mayor complejidad. La tarjeta XS-40 ha limitado el trabajo en el diseño de esta plataforma por varios motivos.

En primer lugar, por no disponer de un bus de datos lo suficientemente ancho para trabajar con los registros de los que dispone la tarjeta. Esto impide que se pueda trabajar en modo demanda DMA, que sería uno de los objetivos principales para incluir en un futuro diseño. A pesar de no estar implementado en este diseño por el problema comentado, la interfaz esta preparada para incluirlo y se ha explicado como debería realizarse la ampliación para que incluyese este tipo de transferencias, de gran utilidad.

El hecho de disponer de un bus de datos más grande, no sólo afecta a la posibilidad de trabajar con los registros, si no que podría cuadruplicar la velocidad de las transferencias si en cada pulso de reloj se enviasen 32 bits de datos en vez de los 8 bits actuales. La plataforma está preparada para un cambio de este tipo, por lo que no sería necesario ningún cambio para disfrutar de esta ventaja, si se realiza el cambio de FPGA.

Otra desventaja de la tarjeta XS-40 es la poca disponibilidad de pines para las conexiones externas. Se han tenido que desactivar varios componentes de la misma para poder realizar todas las conexiones necesarias para el diseño de la plataforma, entre ellos el micro controlador, que en futuros trabajos podría usarse conjuntamente con una FPGA para realizar simultáneamente procesados de imágenes muy complejos a gran velocidad.

Como se comenta en el capitulo 7, en el apartado dedicado a la interfaz VHDL, la falta de disponibilidad de pines también ha impedido que el bus de direcciones sea conectado a ésta interfaz, y tenga que estar conectada directamente a las entradas de direcciones de la memoria SRAM. Esto limita el trabajo con la misma e impide trabajar con las direcciones de manera deseada. Para futuras aplicaciones, que disponga demáspines en la tarjeta, el bus de direcciones debe ser conectado a la entrada de direcciones que se ha dispuesto en la interfaz para de esta manera poder procesar las direcciones o realizar una toma de decisiones a partir de este parámetro.

Cambiar la tarjeta XS-40 por una de mayor capacidad permitiría también realizar procesos más complejos con las imágenes. Por ejemplo, para realizar una convolución en la imagen, sería necesario almacenar en la FPGA al menos 3 líneas de imágenes simultáneamente para procesarlas. En el caso de trabajar con el formato estándar CCIR que tiene una resolución de imagen de 768x572 píxeles, por lo que sería necesario almacenar 3 líneas x 768 píxeles por línea x 8 bits por píxel, necesitaríamos una FPGA de más de 18432 CLBs y nuestra tarjeta no dispone de esto. Una memoria SRAM de mayor capacidad también ayudaría a aumentar la complejidad del procesado y la velocidad de las transmisiones.

Una recomendación para la posible nueva tarjeta que sustituya a la XS-40 en la plataforma es la tarjeta XSV-800 de XESS. Esta tarjeta esta especialmente diseñada por XESS para el tratamiento de imágenes, la visión artificial y el video en tiempo real.

Esta tarjeta dispone de una FPGA de Virtex con hasta 888 miles de puertas lógicas. Además tiene 2 memorias SRAM de 512K x 16 bits cada una, una memoria flash de 16 Mbits, un decodificador de video para PAL, SECAM y NTSC con entrada RCA o S-video, un codificador de audio que permite digitalizar y generar señales de audio hasta 50KHz y 20 bits de resolución.

La tarjeta permite conectar a la FPGA hasta 76 señales de propósito general a través de pines externos en dos buses de expansión. Tiene puertos PS/2, USB, VGA, serie y paralelo a parte de las conexiones RCA, S-Video y la entrada y salida de audio stereo. También dispone, igual que la XS-40, de un microcontrolador para acompañar a las tareas de la FPGA, dos displays de 7 segmentos y diversos botones de pulsación e interruptores. Esta tarjeta sería perfecta para realizar cualquier tipo de aplicación al más alto nivel, con unas prestaciones profesionales, un bajo coste y una flexibilidad total en los diseños.

Otro posible cambio para la plataforma, sería el de la tarjeta PCI Proto-Lab PLX por una que tenga una mayor tasa de transferencia. En este aspecto se han observado dos posibles ampliaciones a otras tarjetas del mismo fabricante. La tarjeta Plx 9056 dispone, al igual que la 9054, de un ancho de bus de 32 bits, pero en este caso, las transferencias se realizan a 66 MHz, por lo que doblamos la velocidad de la transmisión pudiendo transmitir 264Mbytes/seg en vez de los 132Mbytes/seg que permitía la 9054.

La siguiente tarjeta en la gama de aceleradoras PCI es la PCI 9656 Plx. En este caso el bus es de 64 bits para el bus PCI y 32 bits para el bus Local. Esto no es nada eficiente para la plataforma actual, ya que toda la comunicación se realiza con el bus Local y desaprovechamos 32 bits que van hacia el PCI. Por esto se recomienda que para cambiar de tarjeta, esta sea una 9056 como tarjeta de mayores prestaciones que la 9054 pero sin desaprovechar recursos, algo muy importante en el diseño de estas plataformas que pretenden ser de bajo coste.

Una posible línea de trabajo, sin necesidad de cambiar el hardware de la plataforma diseñada, es preparar la API para trabajar con los subcanales de los que dispone cada canal DMA. Esto permitiría multiplexar dos transmisiones independientes por cada canal, pudiendo disponer de hasta cuatro transmisiones independientes por DMA más las transmisiones maestro/esclavo para pequeñas cantidades de datos. Esta mejora es sencilla de implementar y tan sólo es necesario reproducir la apertura de canales y las funciones de transferencias con los canales secundarios de cada canal además de con los primarios que realiza actualmente.

Otro aspecto a mejorar en la plataforma es la velocidad de procesamiento en la FPGA, que actualmente se ve limitada por la velocidad de transmisión y de trabajo de la tarjeta PCI Proto-Lab Plx. Al estar conectado el reloj de la FPGA al reloj de la tarjera Plx, este sólo puede trabajar a una velocidad máxima de 33 Mhz, que es la velocidad de transmisión del bus PCI. Separando los relojes y efectuando un sincronismo entre las transmisiones con estados de espera y buffers intermedios, permitiría a la FPGA trabajar a una velocidad de hasta 100 Mhz si el diseño lo permite, realizando el procesado de las imágenes a mayor velocidad que las transmisiones, que se seguirían efectuando a 33 Mhz.

En el CD del proyecto se han incluido los PDFs con las especificaciones técnicas y los detalles de las tarjetas propuestas como cambio, junto con las de las tarjetas actuales para que pueda ser estudiado este cambio detenidamente, comparando ambas tarjetas y de esta manera decidir cual es la mejor opción.

ANEXO A

Manuales de usuario.

Este anexo contiene el manual de usuario para utilizar la aplicación de ejemplo desarrollada para este proyecto.

También contiene un manual que nos ayudará a utilizar el programa PLXMon2000, contenido en el kit de desarrollo de la tarjeta PCI Prtoto-Lab/PLX. El programa PLXMon 2000 es una aplicación en el entorno Windows-que permite modificar la configuración de varios registros en el dispositivo PLX de la tarjeta de prototipo conectada al Bus PCI. Esta aplicación, también permite al usuario realizar transferencias en bloque, scatter/gather, y transferencias en modo DMA. El programa PLXMon permite diseñar y descargar aplicaciones en la tarjeta, pudiéndose realizar desde el mismo, la programación de la memoria FLASH y EEPROM.

A.1 Manual de usuario de la aplicación de ejemplo.

El programa "Aplicación PCI" es un interfaz para realizar transferencias PCI Target en modo ráfaga 4-Lword. La transferencia se realizará desde el PC a la tarjeta XS40, y viceversa a través del Bus PCI, controlado por la tarjeta de prototipo de PCI Proto-Lab/PLX.

Al iniciar la aplicación, nos aparecerá una mensaje informándonos de la búsqueda que ha realizado la aplicación de los dispositivos PLX conectados a través del Bus PCI, a nuestro PC. En el caso de no encontrar ningún dispositivo visualizaremos el mensaje de la figura A.1, mientras que si se encontró dispositivo podremos ver el mensaje de la figura A.2.

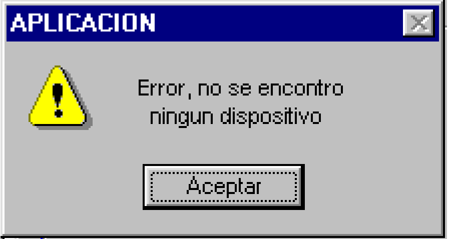


Figura A.1: No hay dispositivos conectados

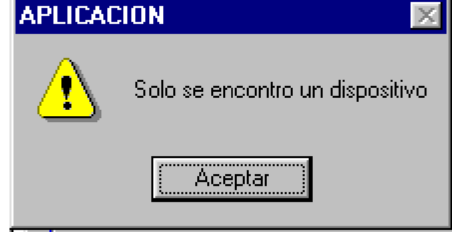


Figura A.2: Sólo se encontró un dispositivo

Una vez mostrado el mensaje anterior, se cargará la ventana principal, que presenta el aspecto de la figura A.3

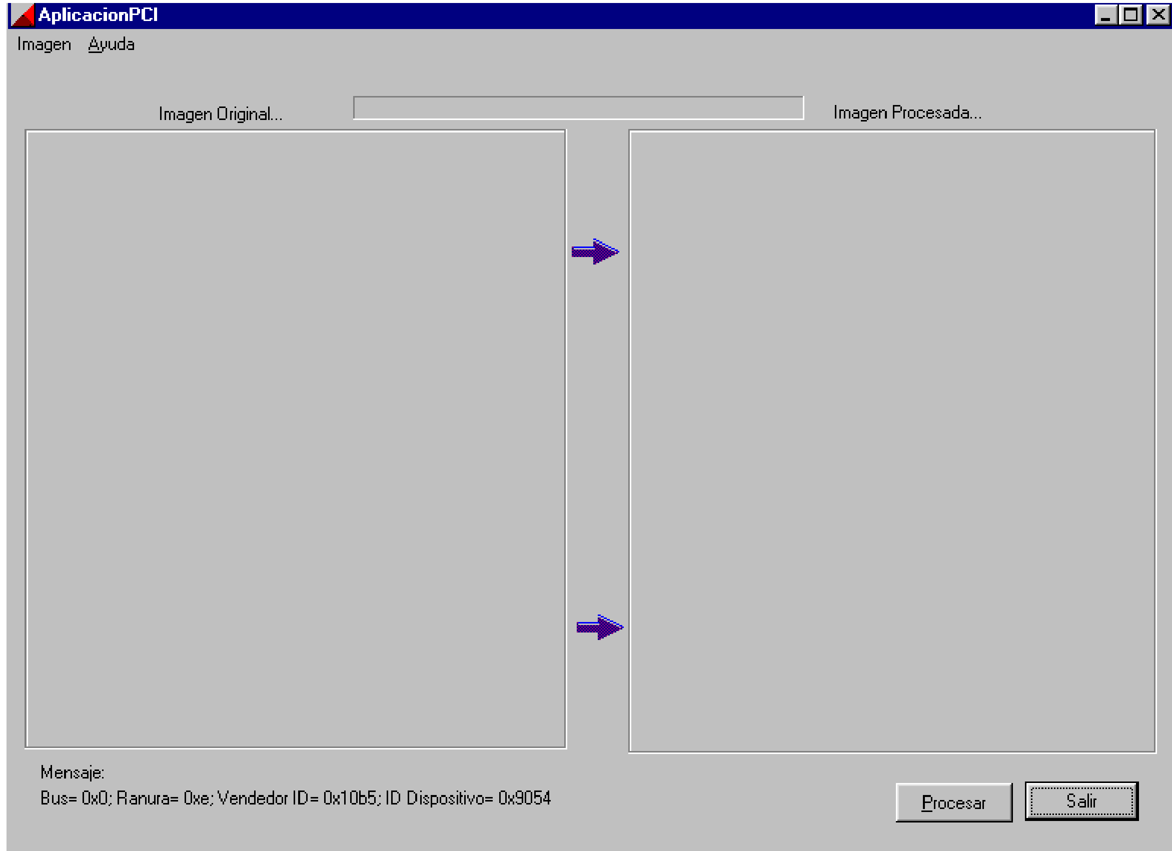


Figura A.3: Pantalla principal de la interfaz de usuario.

La figura A.3, muestra la pantalla principal de la interfaz de usuario, en ella encontramos las siguientes utilidades.

Pinchando en el menú desplegable <Imagen>, podemos cargar una imagen, procesar una imagen previamente cargada y salir de la aplicación.

Pinchando en la opción <Cargar Imagen>, aparecerá el diálogo de Windows, *Abrir archivo*, en este seleccionaremos que imagen deseamos cargar para ser procesada por nuestra aplicación.

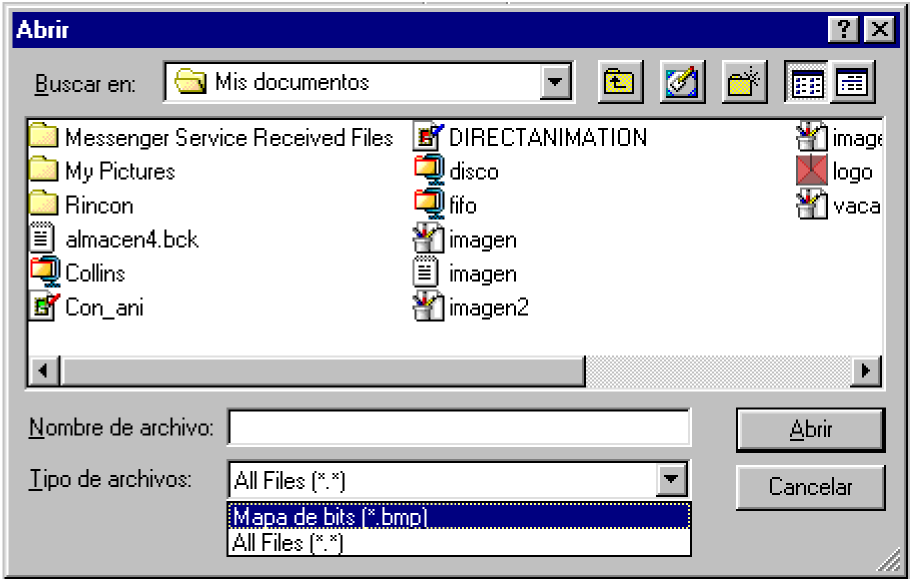


Figura A.4: Dialogo Abrir Imagen

Seleccionamos la imagen que deseamos utilizar y pinchamos en el botón <Abrir>. La imagen seleccionada, se mostrará en la pantalla principal en el cuadro titulado <Imagen Original...>. En la parte superior un mensaje nos indicará la ruta en la que se encuentra situada dicha imagen. También en el mismo menú <imagen>, se encuentra la opción <Procesar...>, con la que podremos realizar el envío de la imagen y la recepción de la misma procesada en la tarjeta XS40. En el caso de que no hayamos cargado ninguna imagen, la aplicación nos informará de ello, mediante el mensaje de la figura A.5

Por último, mencionar que en el menú <Imagen>, tenemos la opción con la que podremos salir de la aplicación.
Desde la ventana principal podemos acceder a estas dos ultimas opciones (<Procesar...> y <Salir>), mediante los botones situados en la parte inferior derecha.
La imagen procesada, será mostrada en el cuadro derecho de la ventana principal.

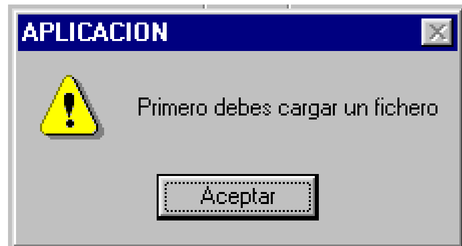


Figura A.5: Mensaje que nos informa que no hay ninguna imagen cargada para procesar

A.2 Manual de usuario del programa PLXMon2000.

El programa PLXMon 2000 es una aplicación para el entorno Windows-que permite realizar la configuración de varios registros del dispositivo PLX de la tarjeta de prototipo conectada al Bus PCI. Esta aplicación permite al usuario realizar transferencias en bloque, scatter/gather, y transferencias en modo DMA. El programa PLXMon permite diseñar y descargar aplicaciones a la tarjeta, pudiéndose también realizar desde el mismo, la programación de la memoria FLASH y EEPROM.

PLXMon puede funcionar en los sistemas operativos Windows 98, Windows 2000 y Windows NT, para operar con cualquiera de los dispositivos IOP 480, PCI 9030, PCI 9054, o PCI 9080.

Las ventanas de diálogo de la aplicación, contienen información detallada sobre los registros del controlador y sobre los bits individuales. Combinando esta información con los algoritmos de programación incluidos, podemos acceder a los componentes IOP por medio del Bus PCI y del puerto serie.

PLXMon 2000 se instala automáticamente en el sistema cuando instalamos el kit Host SDK v3.1, por lo que no deberemos preocuparnos en este punto.

Mediante este programa podemos comunicarnos de dos formas diferentes con la tarjeta de prototipo de Bus PCI, éstos son:

- Por el Bus PCI.
- Por el puerto serie.

El programa se configura automáticamente, cuando se detecte el dispositivo conectado al PC; si la tarjeta no esta conectada por el puerto serie, el programa se configurará en modo PCI. Ésta configuración se guardará permanentemente en un archivo con extensión plx, pudiéndolo encontrar en:
<Ruta de instalación>\PLX\PciSdk\bin\PlxMon2000.plx

Para la utilización del programa PlxMon200, El manual recomienda una resolución de 1024*768 con 256 colores como mínimo.

A.2.1 Instalación

El programa PlxMon2000 se instala automáticamente en el PC al instalar el Host SDK v3.1, como hemos dicho previamente. Este programa dispone de las siguientes características:

- Pantallas Graphical User Interface(GUI) basadas en los registros del dispositivo PLX
- Compatible con los chips IOP 480, PCI 9030, PCI 9054 y PCI 9080.(en nuestro caso es el PCI 9054)
- Utilidad para editar EEPROM o para programar una EEPROM en blanco.
- Split Screen Interface, Utilidad que permite entrada de línea de comando mientras se reciben datos serie.
- Comunicación serie con un puerto IOP. Esta característica es compatible con el protocolo PLX Back End Monitor.

- Posibilidad de descargar las siguientes imágenes estándar; Motorola S-Record, Archivo de imágenes IBM-401B, COFF, y binarias a memorias RAM y FLASH a través del bus PCI y del puerto serie.
- Pantallas de configuración para modificar los contenidos de las EEPROMs NM93CS46, NM93CS56, y NM93CS66. Estas pantallas de configuración permiten cargar y salvar valores desde y a un archivo.
- Permite visualizar el contenido de la memoria IOP o memoria PCI.
- Posee varios programas para realizar test y aplicaciones de ejemplo en el entorno Win32.

A.2.2 Configuración PLXMon2000

Dependiendo si el programa detecta la tarjeta, éste se iniciará en modo PCI o en modo IOP local (por el puerto serie).

En nuestro caso, al tener conectada la tarjeta al PC, se inicializará en modo PCI. De todas formas, se pude cambiar de modo una vez que nos encontremos en el programa. También podríamos hacer funcionar el programa en un segundo ordenador usándolo mediante el puerto serie.

La pantalla principal que aparece al iniciar el programa se muestra en la figura A.6. El programa PlxMon2000, tiene la característica Plug & Play, reconociendo automáticamente las tarjetas instaladas, chequeando los ID del Vendedor y del dispositivo. Pero si el programa no es capaz de detectar tarjeta PLX alguna, lo notificará, y deberemos añadir manualmente la existencia del dispositivo en el menú de propiedades o verificaremos que los datos del dispositivo son correctos en el menú de Properties...(File -> Properties...) La figura A.7 muestra la pantalla de propiedades.

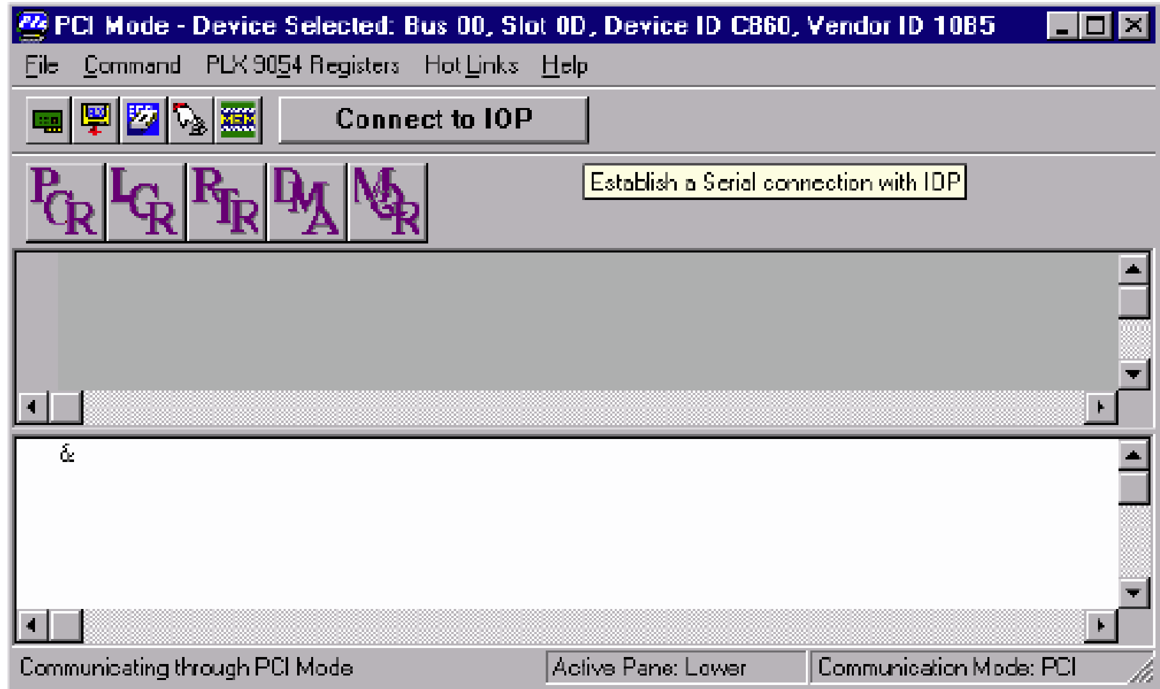


Figura A.6 Pantalla principal de la interfaz de usuario PLXMon2000.

Hasta que estas propiedades no sean designadas, el programa no sabrá los datos que necesita para acceder a la tarjeta. El menú de propiedades muestra la configuración del dispositivo.

Estos datos sólo son usados para accesos PCI. Para la configuración serie usamos los comandos Back End Monitor (BEM) directamente a través del puerto serie.

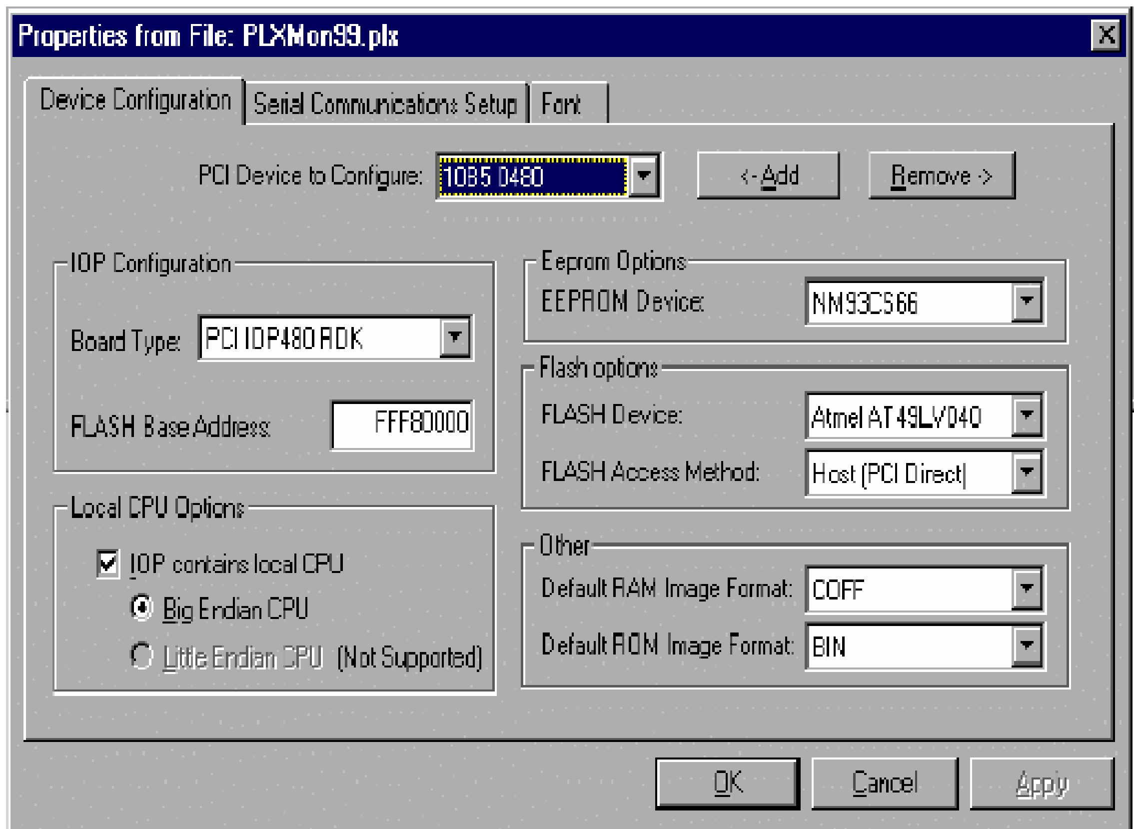


Figura A.7 Pantalla de propiedades PLXMon2000.

Los pasos a seguir para asegurar una operación correcta en PlxMon2000 son:

1. En el ComboBox *PCI Device to Configure,* deben encontrarse los valores *Vendor* y el *Device*, si no es así, pinchamos el botón *Add* para crearlos. En nuestro caso será: 10B5 9054.

2. Introducir toda la información necesaria del dispositivo. Pinchando primero el botón *RDK Default*. Los valores por defecto de la tarjeta RDK se mostrarán en pantalla. Deberemos verificar que las configuraciones de EEPROM y FLASH EEPROM sean correctas.

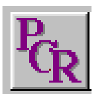
3. También desde esta ventana, podremos cambiar el mapa de memoria o seleccionar la extensión del tipo de archivo de imagen por defecto, para la RAM y la ROM.

4. Pinchando en la pestaña *Serial Communication Setup*, entraremos en otra pantalla donde podremos indicar el puerto serie al que esta conectado la tarjeta y configurarlo. Este programa esta diseñado para alcanzar hasta 38400 baudios de velocidad.

5. También podemos cambiar la fuente de la letra pinchando en la pestaña *Fonts.*

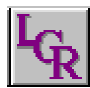
A.2.3 Edición de los registros de la tarjeta.

Tanto si nos encontramos en modo PCI como en modo serie, podemos acceder a los registros de la tarjeta PLX. Para ello, pinchamos en el botón de la pantalla principal PCR(1).

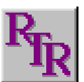


Aparecerá una pantalla con los registros PCI por defecto, en donde podremos modificar alguno de ellos.

Del mismo modo pinchando en el botón LCR(2) entraremos la pantalla con los registros de configuración Local por defecto, en donde del mismo modo que antes podremos cambiar alguno de ellos.



También podemos acceder a los registros que se utilizan en tiempo de ejecución, pinchando en el botón RTR(3). Estos registros son los registros Mailbox, registros Doorbell del PCI a Local y viceversa, IDs del dispositivo y del vendedor, etc.



El botón DMA, nos permite acceder a los registros de configuración DMA, donde podremos cambiarlos.

Por último, entre estos botones encontramos el MQR(4), donde aparecen los registros relacionados con las FIFO del controlador PLX.

A.2.4 Edición de la EEPROM.

El programa PlxMon presenta la posibilidad de editar la EEPROM o programar una en blanco. Esta utilidad sólo estará disponible si no tenemos conectada ninguna tarjeta en el PC.
Para ello, seleccionamos el chip PLX que estemos utilizando, y pincharemos en el botón *Edit Values*. Esto nos llevará a una pantalla especifica para la EEPROM usada. Todos los valores que aparecen en esta pantalla son cero, pudiendo introducir nosotros los valores manualmente o desde una archivo. El kit SDK posee un archivo *.eep para cada dispositivo PLX.

A.2.5 Descargas IOP.

Si deseamos realizar la descarga de una aplicación a través del puerto serie, deberemos hacerlo con este comando. Alguna de las características que dispone este tipo de descargas, son:

- Traduce desde diferentes formatos de archivo, incluyendo: COFF, IBM, ELF, Motorola S-Record, y binario puro.
- Habilidad de descarga del archivo a cada RAM o Flash ROM.
- Lectura binaria desde la memoria FLASH ROM a un archivo binario de datos.
- Soporta descargas serie a la RAM.

(1) PCI Configuration Register
(2) Local Configuration Register
(3) Run Time Register
(4) Message Queue Register

A.2.5.1 Descargas a RAM:

Para ello, seleccionamos primero el canal por donde realizaremos la descarga, serie o PCI antes de abrir la ventana *download.*
Abrimos la ventana de *DownLoad*, pinchando en el botón Aparece una ventana como la mostrada en la figura A.8. Los datos de descarga son guardados en formato COFF para el dispositivo seleccionado, llegaremos a ellos por la ruta:
<Ruta de instalación>\PLX\PciSdk300\IOP\Samples\Hello\9054RDK-860
y seleccionamos el archivo RamHello.cof

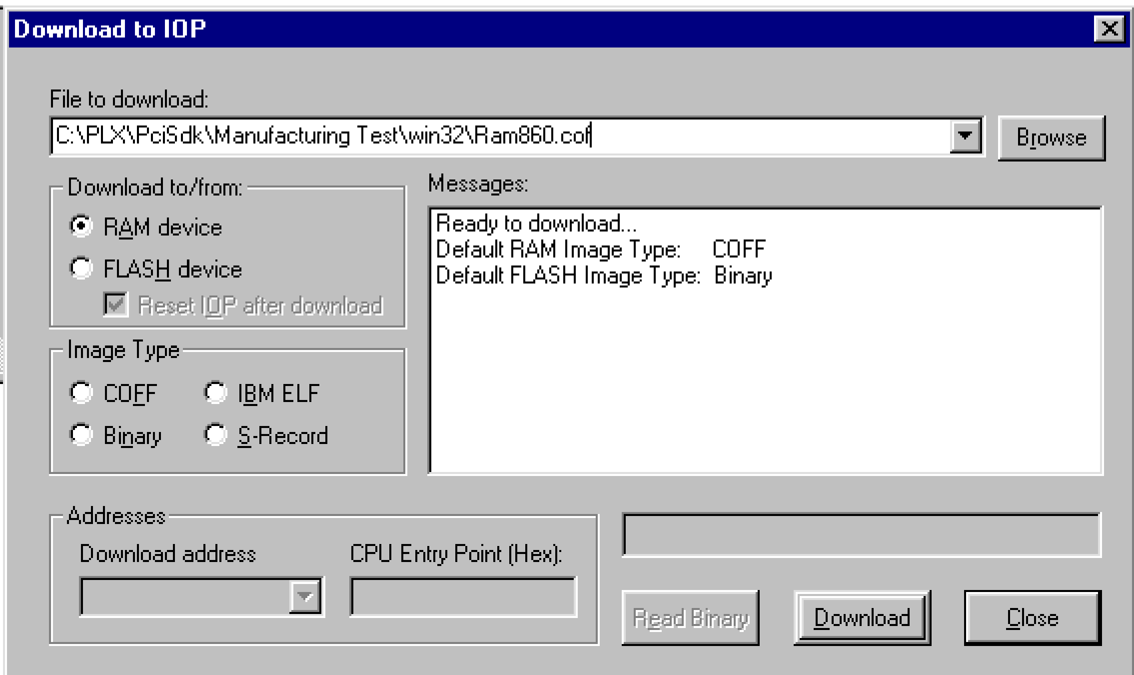


Figura A.8: Ventana de la ventana de descarga en la RAM.

Después de haber seleccionado el archivo, pinchamos en el botón *download*. En la pantalla de estado podemos verificar la correcta descarga del programa.

A.2.5.2 Descarga a FLASH ROM

Si lo que queremos es descargar a una memoria FLASH ROM, deberemos seleccionar la opción *FLASH device.*
Si en el transcurso de la escritura de un programa en el FLASH ocurre un error, no deberemos de apagar el ordenador hasta que hayamos terminado de descargar un programa ROM. Un fallo al hacer esto requerirá reprogramar el chip de FLASH ROM.
Las descargas a FLASH pueden realizarse vía Bus PCI o vía el puerto serie. En este ejemplo, nosotros lo haremos mediante el Bus PCI. Así que, seleccionamos el dispositivo FLASH.

Después de seleccionar la imagen deseada, debemos seleccionar la dirección offset del FLASH, -La dirección offset de FLASH para nuestro controlador es la 0.

Una vez realizados todos estos pasos debemos ir al directorio:
<directorio de instalación>\PLX\PciSdk300\IOP\Samples\Dmaster\9054RDK-860
y seleccionar el archivo RomDM.bin.

Ahora pinchamos en el botón *download*. Podemos verificar la correcta descarga del programa leyendo desde el puerto serie en otro PLXMon 2000, y también en la ventana de estado. Una buena manera de testear el código ROM, es reseteando la tarjeta y verificando que el mismo programa funciona tras el reset.

A.2.6. Transferencias DMA.

La opción *DMA Transfers* puede ser usada para realizar transferencias rápidas entre el IOP y el Bus PCI. Este ejemplo muestra como realizar una simple transferencia DMA de IOP a PCI.

Para realizar una transferencia DMA, algunos registros deben ser inicializados apropiadamente. Primero pinchamos en el botón *DMA* para modificar los registros correspondientes. Ponemos los datos de transferencia como muestra la siguiente figura:

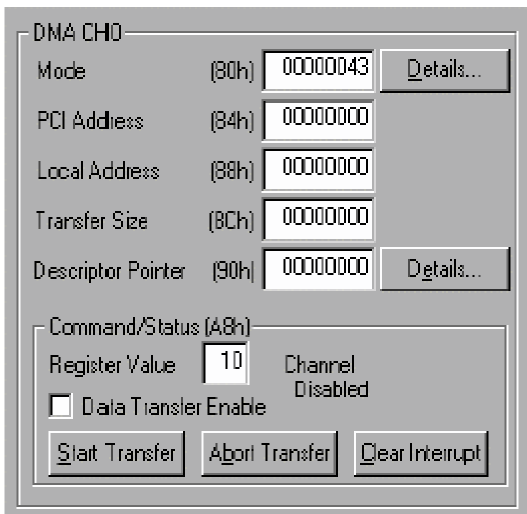


Figura A.9: Pantalla de configuración del canal 0 DMA.

En *prompt &*, el cual esta localizado en el panel de debajo de la ventana principal, podemos ejecutar varios parámetros para recoger información de la tarjeta RDK que hayamos seleccionado. El comando VARS (escribiendo vars en el prompt &) da dos valores para el Hbuf, una dirección virtual y una física. La dirección física se usa para la dirección de PCI(menor de 32 bits).

Introducimos un valor para el contador de la transferencia, 500 es la que usamos en este ejemplo. Introduciendo dl s0 +100000 el cuadro mostrará los contenidos actuales de la memoria local en la tarjeta PLX RDK en la localización 1MB.

Podemos usar el comando *EL,* para editar y escribir en la memoria local. Los requerimientos típicos para configurar el registro *Mode* se realizan habilitando la opción *Ready Input Enable* y la anchura del bus a 32 bits. Esta operación le da al registro *Mode* un valor de 43h. La dirección Local dada aquí es válida para todas las tarjetas y es la 100000h.
Para asignar la dirección de la transferencia de datos lo hacemos en el cuadro *Descriptor Pointer, para* ello, pinchamos en el botón *Details..*, y la escribimos con la opción *Local to PCI* seleccionada. Por último indicar que el registro *Threshold* tiene un valor por defecto de 0.

Para comenzar la transferencia, primero activamos el modo DMA de canal 0. Una vez realizado esto, ya podemos pinchar en el botón *Start Transfer* para comenzara la operación. El bit *DMA Done/ Ready* permanecerá chequeado porque el contador de transferencia de 80 es muy pequeño, y la transferencia se completará rápidamente.

Para verificar la correcta transferencia de los datos, podemos leer el buffer PCI, *Hbuf*. Para ello, escribimos *dl hbuf* en el prompt & del panel de debajo de la pantalla principal.

A.2.7 Tarjetas sin CPU local.

Si estamos usando una tarjeta PCI XXXXRDK-LITE o una tarjeta sin CPU local, la pantalla de propiedades tendrá el contenido que muestra la figura A.10.

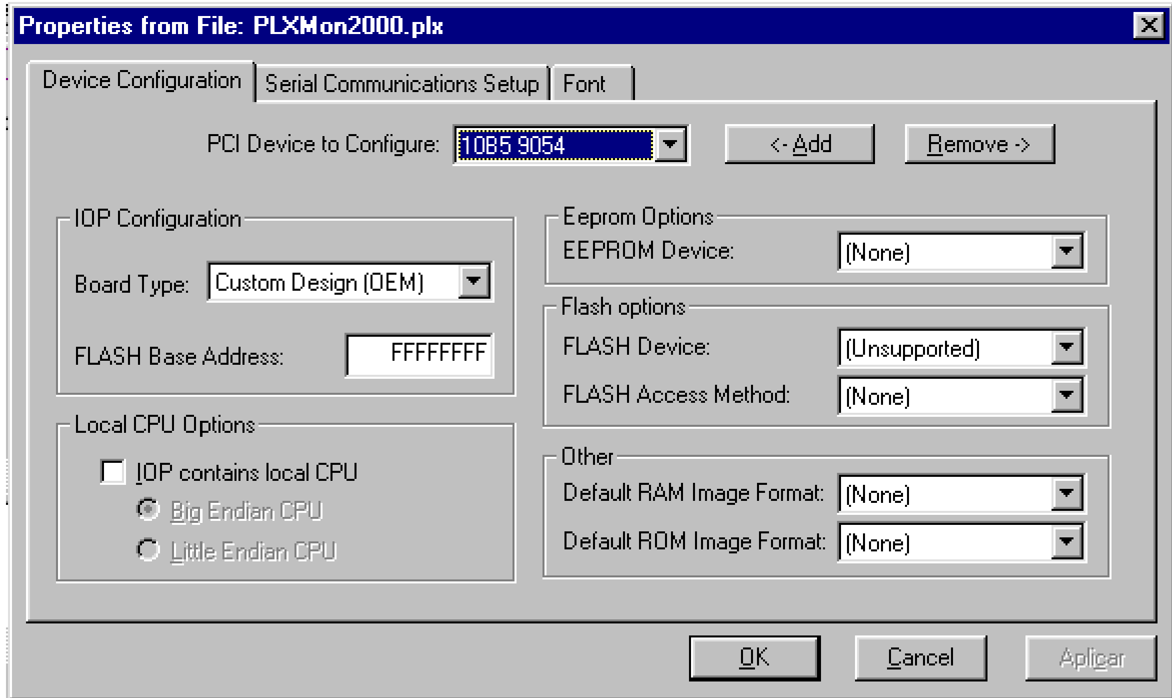


Figura A.10: Ventana de propiedades de una tarjeta sin CPU Local.

Si la tarjeta no tiene un CPU local, deberíamos quitar la opción *IOP contains local CPU* en el cuadro *Local CPU Options*. En la figura se muestra la pantalla de configuración para una tarjeta con un Device ID : ABCD y un Vendor ID: 10B5. Estos IDs fueron añadidos pinchando en el botón *Add*

Después de esto, podremos configurar alguna o todas las demás opciones dependiendo del diseño de la tarjeta.

A.2.8 Utilidades de PLXMon2000.

A.2.8.1 Modos de acceso.

A.2.8.1.1 Modo PCI.

Este es el modo con el que tenemos conectada nuestra tarjeta al PC.

Cuando seleccionamos el modo PCI, todas las comunicaciones entre PLXMon2000 y el chip PLX se hacen a través del Bus PCI, mediante la librería Host SDK API y el driver del dispositivo para Windows. Este método le será familiar a los usuarios de las versiones PLXMon anteriores.

El modo se selecciona pulsando el botón de la barra de herramientas de la pantalla principal *Connect to IOP/Disconnect*. Todas las propiedades RDK usadas en la comunicación PCI podemos encontrarlas en el menú de configuración del dispositivo. La figura A.6 muestra el interface PlxMon2000 en modo PCI

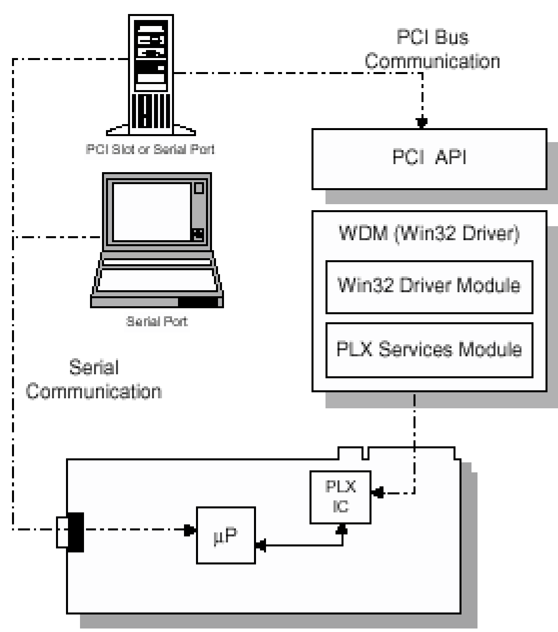


Figura A.11 Conexión PCI vs. Serie.

A.2.8.1.2 Modo Serie.

Nota: esta característica esta permitida con el PCI Pro SDK
Pinchando en el botón de la barra de herramientas de la pantalla principal *Connect to IOP,* entraremos en el modo de transferencia serie. El programa PLXMon 2000 entra en este modo automáticamente si ningún dispositivo PLX esta conectado al Bus PCI.

Cuando tenemos seleccionado el modo serie, todas las comunicaciones entre el programa y el dispositivo PLX, se realizarán a través de la conexión serie, mediante la librería Host SDK API y el driver del dispositivo para Windows.

A.2.9. La interface.

La pantalla principal de la interface PLXMon2000 se mostró en la primera figura del manual. De ella destacamos:

Línea de comandos

Esta, la podemos encontrar en la parte inferior de la pantalla principal de la aplicación. En el prompt &, se pueden introducir varios comandos para enviar o recoger información a la tarjeta que este seleccionada. Esta línea de comando puede ser usada en ambos modos (serie y PCI)

Los principales comandos que podemos introducir desde esta línea son los siguientes:

Displaying Memory Via Memory Cicles, (dl, dw, db)
Estos comandos representan diferentes tamaños de datos que son accedidos desde los ciclos de memoria.
dl: devolverá un valor de 32 bits.

dw: devolverá un valor de 16 bits.
db: devolverá un valor de 8 bits.
Formato:
<command> <address> [[1] bytelength]
Por defecto la longitud del byte es 80h. Escribiendo de nuevo el comando sin argumentos harán que PLXMon2000 continúe presentando el rango con la misma longitud de byte de antes.

Displaying Memory Via I/O Cicles, (il, iw, ib)
Los comandos iX son similares en sintaxis a los dX, salvo que estos acceden a memoria usando ciclos I/O en vez de ciclos de memoria.
il: devolverá 32 bits de datos
iw: devolverá 16 bits de datos
il: devolverá 1 bits de datos
Por defecto, la longitud del byte es dependiente del comando. Todas estas longitudes pueden ser cambiadas. Escribiendo el comando sin argumentos, PLXMon2000 continuará escribiendo las localizaciones de memoria usando el tamaño de los datos escritos con el tamaño del incremento
Formato:
<command> <address> [[1] bytelength]

Writing Memory Via Memory Cicles, (el, ew, eb)
De nuevo la sintaxis de escritura usando los ciclos de memoria son similares a los comandos dX.
el: Escribirá valores con anchura de 32 bits
ew: Escribirá valores con anchura de 16 bits
el: Escribirá valores con anchura de 1 bits
Formato:
<command> <address> [INC increment] [value]
Mientras se requiera entrada de dirección, los valores de incrementos son opcionales. Si no se mete el parámetro de valor, el programa ira requiriendo datos en modo interactivo.
Writing Memory Via I/O Cicles, (ol, ow, ob)
Los comandos oX son sintacticamente más simples que la escritura a los ciclos de memoria. Cada comando escribe un diferente dato de tamaño a la dirección del puerto.
Formato:
<command> <address> <value>

The Pci Command
Este comando permite acceso de lectura / escritura a los registros de configuración PCI.
Formato:
pci <pci offset> [value]

Para escribir el offset requerido, añade el valor a escribir; si no el valor será puesto como un valor de registro de 32 bits.

NOTA: El valor de offset será diferente dependiendo del modo de acceso: PCI o Serie.

The Quit Command
Finaliza la aplicación PLXMon2000.

The Reg Command
El comando reg permite al usuario acceder a los registros de configuración local del chip PLX. Los datos pueden ser leídos o escritos en tamaño de 32 bits.
Formato:
reg < register offset> [value]
Si se da un valor, el comando escribirá los datos en la dirección especificada.

The Repeat Command(r)
El comando repeat puede ser usado para hacer que PLXMon2000 repita los comandos antes del un número determinado de veces.
Formato:
[command] r [iterations]
Si no se da el número de iteraciones, entonces PLXMon2000 ejecutará el comando indefinidamente hasta que el usuario presione una tecla.

El comando que será repetido, deberá estar en la misma expresión que el comando r

User Variables(var)
PLXMon2000 crea algunas etiquetas de uso como una ayuda mnemónica para las localizaciones de la memoria común.

Estas cadenas pueden ser usadas intercambiablemente con los valores que representan.

The ver Command.
Representa los datos de la versión contenida en el host SDK software.Esta versión de PLXMon2000 es compatible sólo con Host SDK v3.1. Este comando sólo será usado en modo PCI

A.2.10. Acceso a los registros

Los contenidos de los registros pueden representarse de una o dos formas en PLXMon 2000.

Normalmente cuando un registro lleno de 32 bits esta siendo mostrado, se muestra en un edit box en formato hexadecimal. También podemos meter nuevos valores, que serán actualizados cuando el usuario cierra la ventana o cuando se mueve el cursor a otro edit box.

Los check box se usan también para representar y cambiar los bits individuales en un registro. Si el check box esta en una ventana de dialogo principal entonces un cambio de estado será inmediato, si el check box esta en un cuadro de diálogo con otros check boxes, entonces los cambios se llevaran a efecto cerrando el diálogo.

A.2.11 Selección de dispositivos.

El menú *Select A Devive*, que se muestra en la figura, se usa para seleccionar un dispositivo PCI al que queremos acceder. El puntero apunta al dispositivo actualmente seleccionado.

La figura lista la tarjeta utilizada en nuestro proyecto (Device ID=0x9054, Device ID=x10B5). Para seleccionar un nuevo dispositivo, si lo hubiese, simplemente deberemos pinchas sobre él, y pulsar el botón *Select*.

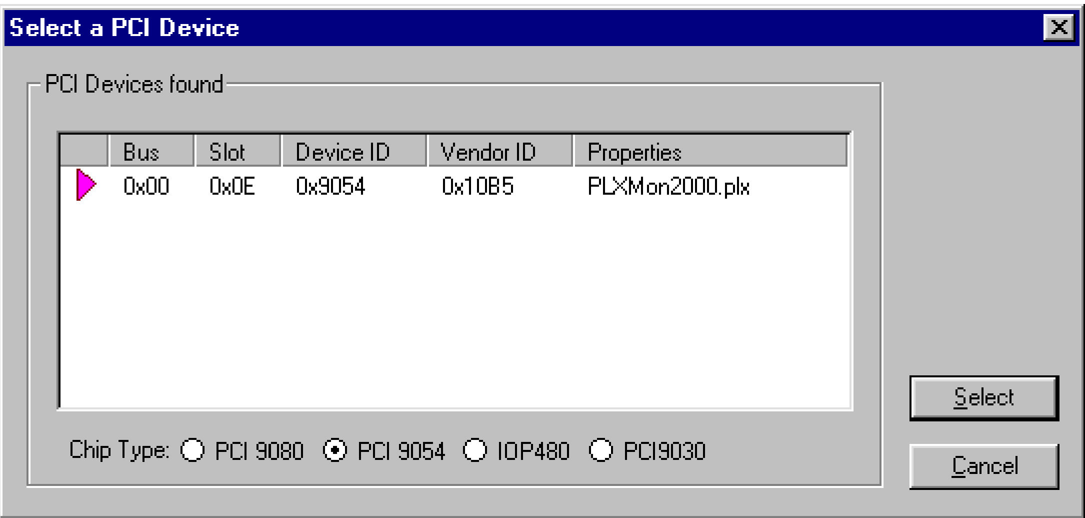


Figura A.12 Cuadro de diálogo de selección de dispositivos.

A.2.12 Accesos a memoria.

Presionando el botón de memoria en la barra de herramientas, PLXMon2000 abrirá el cuadro de diálogo de *Memory Display*. La pantalla de diálogo de nuestro dispositivo se muestra en la siguiente figura.

El display de memoria soporta los modos de lectura y escritura PCI e IOP.
En modo PCI, pinchando en *Read Block* leeremos 0x100 bytes de la dirección de offset del espacio de memoria especificado, con el ancho del bus y representando los datos con la dirección virtual correspondiente de Windows. Pinchando el botón *Write Block* escribiremos los datos mostrados en memoria en la dirección offset especificada.

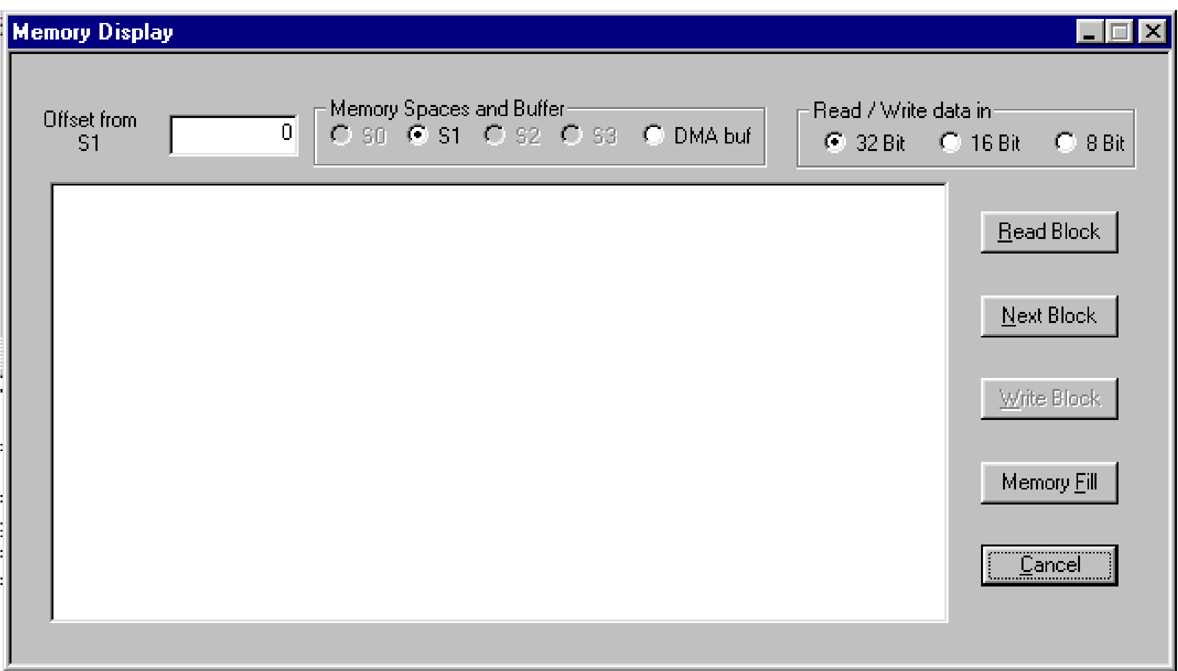
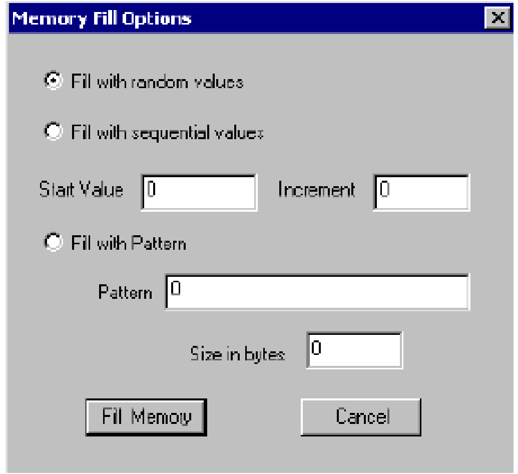


Figura A.13 Pantalla de accesos a memoria.

En modo IOP, pinchando el botón *Read Block* leeremos 0x100 bytes de la dirección local especificada por el usuario, el ancho de bus y los datos con la dirección correspondiente (dirección de memoria local absoluta). Pinchando en el botón *Write Block* escribiremos los datos mostrados en el puerto IOP en la dirección offset especificada.

NOTA: Si damos una dirección de offset o dirección local errónea, podemos causar que el PLXMon2000 Lea/ Escriba en áreas de memoria inválidas. El acceso a memoria en modo IOP esta disponible con PCI Pro SDK.
La función *Memory Fill* permite al usuario escribir posiciones de memoria con un patrón de datos especificado. Pinchando en el botón *Memory Fill*, abrimos la caja de dialogo *Memory Fill Options* como se muestra en la figura A.14:



Seleccionamos el tipo de valores y rellenamos las cajas de texto si fuera necesario. Luego damos el tamaño de las posiciones de memoria, en numero de bytes. Finalmente pinchamos en *Fill Memory* para escribir las posiciones de memoria especificadas que deseemos.

Figura A.14 Pantalla Memory Fill Options.

A.2.13 Información necesaria para contactar con el fabricante de la tarjeta.

El fabricante de la tarjeta recomienda tener los siguientes datos, y mantener el PC encendido, antes de consultar cualquier duda

Datos de nuestro controlador.

1. Número de modelo del PLX PCI RDK.
2. Versión del PLX Host SDK.
3. Sistema operativo y versión instalada en el PC.
4. Versión PLX2000
5. Descripción del diseño o aplicación que queremos realizar.
- Chip PLX usado
- Microprocesador.
- Sistema operativo local y versión
- Dispositivos de I/O

6 Descripción del problema
7. Pasos para recrear el problema:

Para contactar con él, tenemos varias alternativas:

Dirección:

PLX Technology, Inc.
Attn. Technical Support
390 Potrero Avenue
Sunnyvale, CA 94086

Teléfono

408-774-9060

Fax

408-774-2169

Web

http://www.plxtech.com

e-mail

euro-apps(a)plxtech.com

ANEXO B

Instalación de la tarjeta PCI Proto-Lab/ PLX.

Este apéndice contiene los pasos que se deberán seguir para realizar la correcta instalación de la tarjeta de prototipo de Bus PCI en el ordenador.

B.1 Desinstalación de versiones anteriores de PCI SDK

Aviso: Si se ha modificado cualquier archivo del original directorio de instalación PCI SDK, tales como archivo C, el desinstalado puede eliminarlos. Comprueba que no existe ningún archivo necesario en la ruta de instalación.

La instalación de la última versión SDK actualizará los registros y drivers del sistema operativo Windows y del directorio del driver.
Antes de instalar una nueva versión de PCI SDK, se recomienda desinstalar las versiones anteriores que se puedan encontrar en la unidad.
Para eliminar el software antiguo de PCI SDK, incluyendo los driver del dispositivo, completa los siguientes pasos.

1. Cierra todas las aplicaciones PLX.
2. Abre el panel de control de Windows.
3. Pincha en Añadir / eliminar programas.
4. Elige el paquete PCI SDK de la lista, y
5. Pincha en el botón Añadir /Eliminar programas

Nota Esto sólo elimina los archivos que fueron instalados por el programa de instalación PCI SDK.

B.2 Instalación en el entorno Windows.

Para instalar el paquete software PCI SDK, deberemos de completar los siguientes pasos.

1 Una vez insertado el CDROM, sigue las instrucciones que aparecerán en la página HTLM, la cual se cargará automáticamente. Si esto no ocurre, en el explorador de Windows pinchamos en el archivo Setup.exe, colocado en el CDROM.

2 Reinicia el ordenador tras para concluir el proceso de instalación.

El driver del dispositivo es necesario para que el software PCI SDK, se comunique con la tarjeta pinchada al Bus PCI. Las aplicaciones de Windows, no pueden comunicarse con la tarjeta si el driver del mismo no esta instalado.

El paquete PCI SDK, incluye los drivers para todos los dispositivos PLX, aunque en Windows98/2000, el paquete de instalación no puede asignar automáticamente a los dispositivos conectados al PC.

El administrados Plug & Play de Windows, es el responsable de detectar los dispositivos conectados al ordenador e indicar al usuario del correcto driver. Para asignar un driver a un dispositivo, Windows busca en un archivo INF. Este archivo provee de la información necesaria tal como los archivos del driver y los registros de configuración de cada uno. Para instalar un driver para una tarjeta que contenga un dispositivo PLX, completa los siguientes pasos:

1. Tras instalar con éxito el PCI SDK, reinicia el PC.
2. Inserta la tarjeta PLX RDK con el dispositivo PLX en una ranura PCI libre.
3. Reinicia de nuevo el PC. Windows debería detectar el nuevo hardware en el sistema, mostrándonoslo mediante un mensaje Nuevo hardware encontrado

4. Windows ejecutara la aplicación semiautomática "Añadir nuevo Hardware", la cual buscará nuevos drivers para la tarjeta.

Una vez que Windows ha concluido la búsqueda, mostrara el cuadro de la figura B.1, indicando la búsqueda para el mejor driver para el dispositivo.

Figura B.1:Ventana Instalar los driver del dispositivo.

Selecciona la localización del archivo INF. Por defecto PLX, incluye el archivo INF en la ruta: <*Sdk_Install_Dir*>*\Win32\Driver\Wd*m. y pincha *Next.*

Figura B.2: Ventana de los resultados de la búsqueda de los dispositivos.

Cuando aparezca la siguiente ventana, indicará que el proceso de instalación ha concluido

Figura B.3: Ventana que indica que se ha completado la instalación.

ANEXO C

Códigos en Visual C++ y VHDL.

Este anexo contiene los ficheros software necesarios para la realización de nuestra aplicación, incluyendo los archivos de cabecera de la aplicación realizada en Visual C++.

También se ha incluido el código implementado por el fabricante en el dispositivo CPLD, situado en la tarjeta de prototipo PCI Proto-Lab/PLX.

C.1 Códigos de Visual C++ de la API diseñada

INTERFAZPLX.H

```
// InterfazPlx.h: interface for the InterfazPlx class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_INTERFAZPLX_H__44A51AA0_C995_11D7_978E_0050BABF7DD6__INCLUDED_
)
#define AFX_INTERFAZPLX_H__44A51AA0_C995_11D7_978E_0050BABF7DD6__INCLUDED_
#include "pcitypes.h"
#include "plxtypes.h"
#include "plxerror.h"
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
//#define EXPORT extern "C" __declspec(dllexport)


class InterfazPlx
{
public:
	__declspec(dllexport) InterfazPlx();
	__declspec(dllexport) InterfazPlx(DEVICE_LOCATION dev);
	__declspec(dllexport) virtual ~InterfazPlx();
	static int __declspec(dllexport) NumeroDispositivos();
	static int __declspec(dllexport) NumeroDispositivosCriterio(DEVICE_LOCATION dev);
	static DEVICE_LOCATION __declspec(dllexport) BuscaDispositivo(U32 num);
	int __declspec(dllexport) EscrituraMS(U32 address, PU32 datos);
	int __declspec(dllexport) LecturaMS(U32 address, PU32 datos);
	int __declspec(dllexport) CerrarCanalSglDma(int nC);
	int __declspec(dllexport) AbrirCanalSglDma(int nC);
	int __declspec(dllexport) EscrituraSglDma(U32 direccion,U32 tamano,PU32 datos,int nC);
	int __declspec(dllexport) LecturaSglDma(U32 direccion,U32 tamano,PU32 datos, int nC);
	int __declspec(dllexport) AbrirCanalBlockDma(int nC);
	int __declspec(dllexport) CerrarCanalBlockDma(int nC);
	PU32 __declspec(dllexport) EscrituraBlockDma(U32 direccion,U32 tamano,PU32 datos,int
nC);
	PU32 __declspec(dllexport) LecturaBlockDma(U32 direccion,U32 tamano,PU32 datos,int
nC);
	int __declspec(dllexport) AbrirCanalShuttleDma(int nC);
	int __declspec(dllexport) CerrarCanalShuttleDma(int nC);
	int __declspec(dllexport) EscrituraShuttleDma(U32 direccion,U32 tamano,PU32 datos,int
nC);
	int __declspec(dllexport) LecturaShuttleDma(U32 direccion,U32 tamano,PU32 datos,int
nC);
	void __declspec(dllexport) EscribirRegistro(U32 registro, PU32 dato);
	U32 __declspec(dllexport) LeerRegistro(U32 registro);
	U32 __declspec(dllexport) InterfazPlx::LeerRegistroConfiguracion(U32 registro);
	void __declspec(dllexport) InterfazPlx::EscribirRegistroConfiguracion(U32 registro, U32
dato);
	U32 __declspec(dllexport) InterfazPlx::DireccionFisicaBuffer();
	U32 __declspec(dllexport) InterfazPlx::BufferSize();
	U32 __declspec(dllexport) InterfazPlx::DireccionLogicaBuffer();
	bool __declspec(dllexport) InterfazPlx::setBusWidth(int w);
```

```
private:
        HANDLE miDispositivoPlx;
        PCI_MEMORY PciMemory;
        RETURN_CODE rc;
        DEVICE_LOCATION device;
        U32 width;

        void CerrarDispositivo();

};

#endif //
!defined(AFX_INTERFAZPLX_H__44A51AA0_C995_11D7_978E_0050BABF7DD6__INCLUDED_
)
```

INTERFAZPLX.CPP

```
// InterfazPlx.cpp: implementation of the InterfazPlx class.
//
//////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "InterfazPlx.h"

#include "stdafx.h"
#include <stdio.h>
#include <iostream.h>
#include <conio.h>


#include "pcitypes.h"
#include "plxtypes.h"
#include "plx.h"
#include "pciapi.h"
#include "plxError.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

DEVICE_LOCATION device;
HANDLE miDispositivoPlx;
PCI_MEMORY PciMemory;
RETURN_CODE rc;


//////////////////////////////////////////////////////////////////////
```

```
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

//Constructor por defecto
InterfazPlx::InterfazPlx()
{

	//Inicializamos todos los valores de localizacion del dispositivo por
	//defecto para q abra el primer dispositivo q encuentre.
	device.BusNumber = MINUS_ONE_LONG;
	device.SlotNumber = MINUS_ONE_LONG;
	device.DeviceId = MINUS_ONE_LONG;
	device.VendorId = MINUS_ONE_LONG;
	device.SerialNumber[0] = '\0';
	width=0x0;

	//Abre el primer dispositivo y guarda su controlador en miDispositivoPlx
	rc = PlxPciDeviceOpen(&device,&miDispositivoPlx);


	//Si se produce un error al abrir el dispositivo muestra el mensaje de error.
	if (rc != ApiSuccess)
	{
		printf ("\n\nError al abrir el dispositivo.\n");
		printf ("Returned code is %d\n",rc);
		return;
	}

	printf("Bus= 0x%x; Ranura= 0x%x; Vendedor ID= 0x%x; ID Dispositivo= 0x%x\n",
				device.BusNumber,
				device.SlotNumber,
				device.VendorId,
				device.DeviceId);

	//Obtenemos la direccion fisica, logica y el tamaño del buffer de intercambio
	//y los guardamos en la variable global PciMemory para utilizarlosmástarde
	rc = PlxPciCommonBufferGet(miDispositivoPlx, &PciMemory);
	if (rc != ApiSuccess)
	{
		printf("\a\nPlxPciCommonBufferGet tested: FAILED.");
	}

}

//Contructor sobrecargado: Abre el dispositivo especificado por dev
InterfazPlx::InterfazPlx(DEVICE_LOCATION dev)
{

	rc = PlxPciDeviceOpen(&dev,&miDispositivoPlx);
```

```
        if (rc != ApiSuccess)
        {
                printf ("\n\nError al abrir el dispositivo.\n");
                printf ("Returned code is %d\n",rc);
                return;
        }

}

//El destructor se asegura de q no quede ningun canal DMA abierto
//y cierra el dispositivo PLX.
InterfazPlx::~InterfazPlx()
{
        try{
        rc=PlxDmaSglChannelClose(miDispositivoPlx,PrimaryPciChannel0);
        if(rc==ApiSuccess){
                        printf("Se ha cerrado correctamente el canal 0");
                }
        rc=PlxDmaSglChannelClose(miDispositivoPlx,PrimaryPciChannel1);
        if(rc==ApiSuccess){
                        printf("Se ha cerrado correctamente el canal 1");
                }
        }catch(CException e){}

        rc = PlxPciDeviceClose(miDispositivoPlx);
        if (rc != ApiSuccess)
        {
                printf ("\nError al cerrar el dispositivo Plx.\n");
                printf ("Returned code is %d\n",rc);
        }
}

//Devuelve el numero de dispositivos del sistema
static int NumeroDispositivos()
{

        U32 plxDeviceCount;

   device.BusNumber  = MINUS_ONE_LONG;
   device.SlotNumber = MINUS_ONE_LONG;
   device.DeviceId   = MINUS_ONE_LONG;
   device.VendorId   = MINUS_ONE_LONG;
   device.SerialNumber[0] = '\0';


        //Ordena a PlxPciDeviceFind que busque todos los dispositivos del sistema.
   plxDeviceCount = FIND_AMOUNT_MATCHED;
```

```
    rc = PlxPciDeviceFind(&device, &plxDeviceCount);

    if (rc != ApiSuccess)
    {
                      printf("Error, no se ha encontrado dispositivo");
        return (-1);
    }

   return plxDeviceCount; //Devuelve en numero de dispositivos del sistema.

}

//Devuelve el numero de dispositivos correspondientes
//a un cierto criterio de busqueda.
static int NumeroDispositivosCriterio(DEVICE_LOCATION dev)
{
            U32 plxDeviceCount;

    plxDeviceCount = FIND_AMOUNT_MATCHED;          //Parametro que indica que se debe realizar
la busqueda
                                                                                    //de todos
los dispositivos conectado

    rc = PlxPciDeviceFind(&dev, &plxDeviceCount);

    if (rc != ApiSuccess)
    {
                      printf("Error, no se ha encontrado dispositivo");
        return (-1);
    }

   return plxDeviceCount;

}

//Devuelve los datos de localizacion del numero
//de dispositivo deseado
static DEVICE_LOCATION BuscaDispositivo(U32 num){

            U32 plxDeviceCount;

    device.BusNumber  = MINUS_ONE_LONG;
    device.SlotNumber = MINUS_ONE_LONG;
    device.DeviceId   = MINUS_ONE_LONG;
    device.VendorId   = MINUS_ONE_LONG;
    device.SerialNumber[0] = '\0';

    plxDeviceCount = num;
```

```
    rc = PlxPciDeviceFind(&device, &plxDeviceCount);

    if (rc != ApiSuccess)
    {
                    printf("Error, no se ha encontrado dispositivo");
    }

   return device;
}

/*Cierra el dispositivo (Esta funcion es privada para q sólo
se cierre el disposito al destruir el objeto)*/
void InterfazPlx::CerrarDispositivo(){

        rc = PlxPciDeviceClose(miDispositivoPlx);
        if (rc != ApiSuccess)
        {
                printf ("\n\nErrors in closing device.\n");
                printf ("Returned code is %d\n",rc);
                return;
        }
}



/*      FUNCIONES DE TRANSFERENCIAS        */


/* ESCRITURA MAESTRO/ESCLAVO */
int InterfazPlx::EscrituraMS(U32 address, PU32 datos){

        try{
        rc = PlxBusIopWrite(miDispositivoPlx,IopSpace0,address,TRUE,datos,0x4,BitSize32);


        if (rc != ApiSuccess)
        {
                printf("Return Code %d.\n", rc);
                printf("Error: Unable to write data.\n");
                return -1;
        }
        } catch (int ex){
                        printf("Se produjo una excepcion: %d",ex);
}

        return 0;
}
```

```
/* LECTURA MAESTRO/ESCLAVO */
int InterfazPlx::LecturaMS(U32 address, PU32 destino){


	//Se comprueba q el destino sea valido.
	if (destino==NULL){
		printf("El destino de los datos es incorrecto");
		return -1;}


	try{
	rc = PlxBusIopRead(miDispositivoPlx,IopSpace0,address,TRUE,destino,0x4,BitSize32);
	if (rc != ApiSuccess)
	{
		printf("Error: No se ha podido leer.\n");
		printf("Return Code %d.\n", rc);

		return -1;
	}
	}
	catch(int ex){
		printf("Se produjo una excepcion: %d",ex);
	}
	return 0;
}


/* TRANSFERENCIAS SCATTER/GATTER DMA */

//Abre el canal SCATTER/GATTER DMA
int InterfazPlx::AbrirCanalSglDma(int canal){

	DMA_CHANNEL_DESC desc;

	memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));

	desc.DmaChannelPriority=Rotational; //La prioridad de los canales es rotacional
	//Permite que las direcciones origen y destino puedan incrementarse
	desc.HoldIopAddrConst=0;
	desc.HoldIopDestAddrConst=0;
	desc.HoldIopSourceAddrConst=0;
	desc.EnableReadyInput=1; //Activa la entrada READY del controlador PLX
	desc.IopBusWidth=width;			//Determina el ancho del bus local
	desc.DmaStopTransferMode=AssertBLAST; //La transferencia terminara con la señan
								//BLAST (en vez
de con EOT)

	//Abre el canal 0 si nC es 0 o el 1 en cualquier otro caso
	if(canal)
```

```
                rc = PlxDmaSglChannelOpen(miDispositivoPlx,PrimaryPciChannel1,&desc);
        else
                rc = PlxDmaSglChannelOpen(miDispositivoPlx,PrimaryPciChannel0,&desc);

        if(rc==ApiDmaChannelUnavailable){
                printf("El canal DMA%d esta ocupado",canal);
                return -1;
        }
        if(rc!=ApiSuccess){
                printf("Error al abrir el canal.\n Return Code: %d\n",rc);
                return -1;
        }

        return 0;

}

//Cierra el canal DMA Scatter/Gatter
int __declspec(dllexport) InterfazPlx::CerrarCanalSglDma(int canal){


        try{
        if(canal)
                rc=PlxDmaSglChannelClose(miDispositivoPlx,PrimaryPciChannel1);
        else
                rc=PlxDmaSglChannelClose(miDispositivoPlx,PrimaryPciChannel0);

        if(rc!=ApiSuccess){
                printf("Error al cerrar el canal.\n Return Code: %d\n",rc);
                return -1;
        }
        }
        catch(CException e){}

        return 0;
}


/* ESCRITURA DMA SCATTER/GATTER */
int __declspec(dllexport) InterfazPlx::EscrituraSglDma(U32 direccion,U32 tamano,PU32 datos, int
canal){

DMA_TRANSFER_ELEMENT dmaStruct;

memset(&dmaStruct, 0, sizeof(DMA_TRANSFER_ELEMENT)); //Inicializa a 0 todos los valores de
dmaStruct


dmaStruct.Pci9054Dma.IopAddr = direccion;      //Direccion destino de la transferencia
```

```
dmaStruct.Pci9054Dma.TransferCount = tamano;//Tamaño de los datos a transferir
dmaStruct.Pci9054Dma.IopToPciDma = false;    //Escritura
dmaStruct.Pci9054Dma.UserAddr= (U32)datos; //Direccion de origen

if(canal)
        rc = PlxDmaSglTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct,FALSE);
else
        rc = PlxDmaSglTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct,FALSE);

if (rc != ApiSuccess){
        printf("ERROR: Unable to perform SGL transfer, code = %d\n", rc);
        return -1;
}
return 0;


}


/* LECTURA SCATTER/GATTER DMA */
int InterfazPlx::LecturaSglDma(U32 direccion,U32 tamano,PU32 datos,int canal){

DMA_TRANSFER_ELEMENT dmaStruct;

memset(&dmaStruct, 0, sizeof(DMA_TRANSFER_ELEMENT));

dmaStruct.Pci9054Dma.IopAddr = direccion;
dmaStruct.Pci9054Dma.TransferCount = tamano;
dmaStruct.Pci9054Dma.IopToPciDma = true;     //Lectura
dmaStruct.Pci9054Dma.UserAddr= (U32)datos;

if(canal)
        rc = PlxDmaSglTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct,FALSE);
else
        rc = PlxDmaSglTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct,FALSE);

if (rc != ApiSuccess){
        printf("ERROR: Unable to perform SGL transfer, code = %d\n", rc);
        return -1;
}

return 0;
}


/*                TRANSFERENCIAS DMA DE BLOQUES                */

//Abre el canal DMA para transferencia de bloques
int __declspec(dllexport) InterfazPlx::AbrirCanalBlockDma(int canal){
```

```
	DMA_CHANNEL_DESC desc;

	memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));

	desc.DmaChannelPriority=Rotational;
	desc.HoldIopAddrConst=0;
	desc.HoldIopDestAddrConst=0;
	desc.HoldIopSourceAddrConst=0;
	desc.EnableReadyInput=1;
	desc.IopBusWidth=width;
	desc.DmaStopTransferMode=AssertBLAST;

	if(canal)
	rc = PlxDmaBlockChannelOpen(miDispositivoPlx,PrimaryPciChannel1,&desc);
	else
	rc = PlxDmaBlockChannelOpen(miDispositivoPlx,PrimaryPciChannel0,&desc);
	if(rc!=ApiSuccess){
		printf("Error al abrir el canal %d.\n Return Code: %d\n",canal,rc);
		return -1;
		}
	return 0;
}


//Cierra el canal DMA para transferencias de bloque
int __declspec(dllexport) InterfazPlx::CerrarCanalBlockDma(int canal){

	try{
	if(canal)
		rc = PlxDmaBlockChannelClose(miDispositivoPlx,PrimaryPciChannel1);
	else
		rc = PlxDmaBlockChannelClose(miDispositivoPlx,PrimaryPciChannel0);
		if(rc!=ApiSuccess){
			printf("Error al cerrar el canal %d.\n Return Code: %d\n",canal,rc);
			return -1;
			}
	}catch(CException e){}
		return 0;

}


//Reliza un ciclo de escritura DMA en modo de transferencia de bloques
//Es obligado el uso del buffer de la tarjeta para estas transferencias.
PU32 InterfazPlx::EscrituraBlockDma(U32 direccion,U32 tamano,PU32 datos,int canal){

	DMA_TRANSFER_ELEMENT dmaStruct;
	PU32 buffer;
```

```
	memset(&dmaStruct, 0, sizeof(DMA_TRANSFER_ELEMENT));


	if(tamano>PciMemory.Size){ //Comprueba que el tamaño a transferir no sea mayor q el
buffer
		printf("Tamaño demasiado grande. Debe ser menor de %d",PciMemory.Size);
		return (PU32)-1;
	}

	buffer=(PU32)PciMemory.UserAddr; //Direccion lógica del buffer

	//Mueve los datos al buffer de la tarjeta para ser transmitidos
	for(unsigned int i =0; i<tamano/4; i++)
		*(buffer+i) = *(datos+i);

	dmaStruct.Pci9054Dma.IopAddr = direccion;
	dmaStruct.Pci9054Dma.TransferCount = tamano;
	dmaStruct.Pci9054Dma.IopToPciDma = false;
	dmaStruct.Pci9054Dma.LowPciAddr= (U32)PciMemory.PhysicalAddr; //Direccion Fisica del
buffer
	dmaStruct.Pci9054Dma.TerminalCountIntr=0;

	if(canal)
	rc = PlxDmaBlockTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct,FALSE);
	else
	rc = PlxDmaBlockTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct,FALSE);
		if (rc != ApiSuccess){
	printf("ERROR: Unable to perform Block transfer, code = %d\n channel %d", rc,canal);
	return (PU32)-1;
}

return buffer;
}


//Realiza la lectura de un bloque por DMA
//Es obligado el uso del buffer de la tarjeta para estas transferencias.
PU32 InterfazPlx::LecturaBlockDma(U32 direccion,U32 tamano,PU32 datos,int canal){

	DMA_TRANSFER_ELEMENT dmaStruct;
	PU32 buffer;


	memset(&dmaStruct, 0, sizeof(DMA_TRANSFER_ELEMENT));


	//Comprueba que los datos a leer no sean mayores que el tamaño del buffer
	if(tamano>PciMemory.Size){
```

```
			printf("Tamaño demasiado grande. Debe ser menor de %d",PciMemory.Size);
			return (PU32)-1;
		}

		buffer=(PU32)PciMemory.UserAddr;

		dmaStruct.Pci9054Dma.IopAddr = direccion;
		dmaStruct.Pci9054Dma.TransferCount = tamano;
		dmaStruct.Pci9054Dma.IopToPciDma = true;
		dmaStruct.Pci9054Dma.LowPciAddr= (U32)PciMemory.PhysicalAddr;
		dmaStruct.Pci9054Dma.TerminalCountIntr=0;


if(canal)
		rc = PlxDmaBlockTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct,FALSE);
else
		rc = PlxDmaBlockTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct,FALSE);
if (rc != ApiSuccess){
		printf("ERROR: Unable to perform Block transfer, code = %d\n channel %d\n", rc,canal);
		return (PU32)-1;
}

//Mueve los datos leido del buffer de la tarjeta a la variable para devolverlos
for(unsigned int i =0; i<tamano/4; i++)
		*(datos+i)=*(buffer+i);

return buffer;
}



/*		TRANSFERENCIAS SHUTTLE DMA		*/

//Abre un canal Shuttle DMA
int __declspec(dllexport) InterfazPlx::AbrirCanalShuttleDma(int nC){

		DMA_CHANNEL_DESC desc;
		memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));

		desc.DmaChannelPriority=Rotational;
		desc.HoldIopAddrConst=0;
		desc.HoldIopDestAddrConst=0;
		desc.HoldIopSourceAddrConst=0;
		desc.EnableReadyInput=1;
		desc.IopBusWidth=width;
		desc.DmaStopTransferMode=AssertBLAST;

if(nC)
		rc = PlxDmaShuttleChannelOpen(miDispositivoPlx,PrimaryPciChannel1,&desc);
```

```
else
        rc = PlxDmaShuttleChannelOpen(miDispositivoPlx,PrimaryPciChannel0,&desc);
if(rc!=ApiSuccess)
        {
                printf("Error al abrir el canal %d.\n Return Code: %d\n",nC,rc);
                return -1;
        }

        return 0;
}


//Cierra un canal shuttle DMA
int __declspec(dllexport) InterfazPlx::CerrarCanalShuttleDma(int nC){

        try{
if(nC)
        rc=PlxDmaShuttleChannelClose(miDispositivoPlx,PrimaryPciChannel1);
else
        rc=PlxDmaShuttleChannelClose(miDispositivoPlx,PrimaryPciChannel0);
if(rc!=ApiSuccess){
                printf("Error al cerrar el canal %d.\n Return Code: %X\n",nC,rc);
                return -1;
        }
        }catch(CException e){}
        return 0;
}


//Inicia la transferencia de escritura shuttle DMA
int __declspec(dllexport) InterfazPlx::EscrituraShuttleDma(U32 direccion,U32 tamano,PU32 datos,int
nC){

DMA_TRANSFER_ELEMENT dmaStruct;

dmaStruct.Pci9054Dma.IopAddr = direccion;
dmaStruct.Pci9054Dma.TransferCount = tamano;
dmaStruct.Pci9054Dma.IopToPciDma = 0;
dmaStruct.Pci9054Dma.UserAddr= (U32)datos;

if(nC)
rc = PlxDmaShuttleTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct);
else
rc = PlxDmaShuttleTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct);
if (rc != ApiSuccess){
        printf("ERROR: Unable to perform Shuttle transfer, code = %X\n channel %d", rc, nC);
        return 0;
}
return 1;
```

```
}

//Inicia la lectura shuttle DMA
int __declspec(dllexport) InterfazPlx::LecturaShuttleDma(U32 direccion,U32 tamano,PU32 datos,int
nC){

DMA_TRANSFER_ELEMENT dmaStruct;

dmaStruct.Pci9054Dma.IopAddr = direccion;
dmaStruct.Pci9054Dma.TransferCount = tamano;
dmaStruct.Pci9054Dma.IopToPciDma = 1;
dmaStruct.Pci9054Dma.UserAddr= (U32)datos;

if(nC)
rc = PlxDmaShuttleTransfer(miDispositivoPlx,PrimaryPciChannel1,&dmaStruct);
else
rc = PlxDmaShuttleTransfer(miDispositivoPlx,PrimaryPciChannel0,&dmaStruct);
if (rc != ApiSuccess){
        printf("ERROR: Unable to perform Shuttle transfer, code = %X\n, channel %d",rc,nC);
        return 0;
}
return 1;
}

//Escribe un dato de 32 bits en los registros iniciales de la tarjeta (del 00h al FCh)
void __declspec(dllexport) InterfazPlx::EscribirRegistro(U32 registro, PU32 dato){
        PlxPciConfigRegisterWrite(device.BusNumber, //Bus donde se encuentra la tarjeta
                                                device.SlotNumber, //Slot de la tarjeta
                                                registro, dato); //Offset del registro y dato
a escribir
        if (rc != ApiSuccess)
        {
                printf("\n Error Writing a valid Register");
        }
}


//Devuelve el contenido del registro especificado por si offset (registros del 00h al FCh)
U32 __declspec(dllexport) InterfazPlx::LeerRegistro(U32 registro){
        U32 dato;
        dato=PlxPciConfigRegisterRead(device.BusNumber, device.SlotNumber,registro, &rc);
        if (rc != ApiSuccess)
        {
                printf("\n Error Writing a valid Register");
                return (U32)-1;
        }
        return dato;
}
```

```
//Escritura de registros para configuración de DMA
void __declspec(dllexport) InterfazPlx::EscribirRegistroConfiguracion(U32 registro, U32 dato){
	PlxRegisterWrite(miDispositivoPlx,registro, dato);
	if (rc != ApiSuccess)
	{
		printf("\n Error Writing a valid Register");
	}
}


//Lectura de registros de configuración DMA
U32 __declspec(dllexport) InterfazPlx::LeerRegistroConfiguracion(U32 registro){
	U32 dato;
	dato=PlxRegisterRead(miDispositivoPlx,registro, &rc);
	if (rc != ApiSuccess)
	{
		printf("\n Error Writing a valid Register");
		return (U32)-1;
	}
	return dato;
}


//Devuelve la dirección fisica del buffer de la tarjeta
U32 __declspec(dllexport) InterfazPlx::DireccionFisicaBuffer(){

	return PciMemory.PhysicalAddr;
}

//Devuelve la dirección lógica del buffer de la tarjeta
U32 __declspec(dllexport) InterfazPlx::DireccionLogicaBuffer(){

	return PciMemory.UserAddr;

}


//Devuelve el tamaño del buffer de la tarjeta
U32 __declspec(dllexport) InterfazPlx::BufferSize(){

	return PciMemory.Size;

}

bool __declspec(dllexport) InterfazPlx::setBusWidth(int w){

	switch (w){
```

```
	case 8:
		width=0x0;
		return true;
		break;

	case 16:
		width=0x2;
		return true;
		break;

	case 32:
		width=0x3;
		return true;
		break;
	default:
		return false;
		break;
	}


}
```

C.2 Códigos de Visual C++ de la aplicación del PC

APLICACIONDLG.H

```
// AplicacionDlg.h : header file
//

#if
!defined(AFX_APLICACIONDLG_H__C3D25226_E7E7_11D5_8B2C_004F4E039FBF__INCLUDED
_)
#define
AFX_APLICACIONDLG_H__C3D25226_E7E7_11D5_8B2C_004F4E039FBF__INCLUDED_
#include <mil.h>
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CAplicacionDlg dialog

class CAplicacionDlg : public CDialog
{
```

```
// Construction
public:
	int ImagenSize;
	int ImagenY;
	int ImagenX;
	bool PrimeraCarga;
	MIL_ID milBuffer2;
	MIL_ID milBuffer1;
	MIL_ID milDisplay2;
	MIL_ID milDisplay1;
	MIL_ID milSistema;
	MIL_ID milAplicacion;
	int EnviarImagen();
	CAplicacionDlg(CWnd* pParent = NULL);	// standard constructor

// Dialog Data
	//{{AFX_DATA(CAplicacionDlg)
	enum { IDD = IDD_APLICACION_DIALOG };
	CStatic	m_Imagen2;
	CStatic	m_Imagen1;
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAplicacionDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	//{{AFX_MSG(CAplicacionDlg)
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	afx_msg void OnBEnviar();
	afx_msg void OnImagenCargarImagen();
	afx_msg void OnImagenEnviarImagen();
	afx_msg void OnClose();
	afx_msg void OnMAcercaDe();
	virtual void OnOK();
	afx_msg void OnImagenSalir();
	afx_msg void OnDestroy();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif //
!defined(AFX_APLICACIONDLG_H__C3D25226_E7E7_11D5_8B2C_004F4E039FBF__INCLUDED
_)
```

APLICACIONDLG.CPP

```
// AplicacionDlg.cpp : implementation file
//

#include "stdafx.h"
#include <windows.h>//posibles de borrar??
#include <stdio.h>
#include <conio.h>
#include <time.h>
//#include <winbase.h>

#include "pcitypes.h"
#include "plxtypes.h"
#include "plx.h"
#include "pciapi.h"
#include "plxError.h"

#include <mil.h>

#include "Aplicacion.h"
#include "AplicacionDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//#define Xsize 512
//#define Ysize 512
//#define offsetX 0
//#define offsetY 0

static unsigned char* abuffer1;

U32 *bufTo;
int i,j;
int k;
```

```
CString texto;
U32 totalSize=128;
CString ImagenPath;
CString archivo;
InterfazPlx miTarjeta;

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CAplicacionDlg dialog

CAplicacionDlg::CAplicacionDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CAplicacionDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CAplicacionDlg)
		// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CAplicacionDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAplicacionDlg)
	DDX_Control(pDX, IDC_PImage2, m_Imagen2);
	DDX_Control(pDX, IDC_PImage1, m_Imagen1);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAplicacionDlg, CDialog)
	//{{AFX_MSG_MAP(CAplicacionDlg)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_BN_CLICKED(IDC_BEnviar, OnBEnviar)
	ON_COMMAND(IDC_Imagen_CargarImagen, OnImagenCargarImagen)
	ON_COMMAND(IDC_Imagen_EnviarImagen, OnImagenEnviarImagen)
	ON_WM_CLOSE()
	ON_COMMAND(IDC_MAcercaDe, OnMAcercaDe)
	ON_COMMAND(IDC_Imagen_Salir, OnImagenSalir)
	ON_WM_DESTROY()
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CAplicacionDlg message handlers
void CAplicacionDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID & 0xFFF0) == IDM_ABOUTBOX)
	{
		CAboutDlg dlgAbout;
		dlgAbout.DoModal();
	}
	else
```

```
	{
		CDialog::OnSysCommand(nID, lParam);
	}
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CAplicacionDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CAplicacionDlg::OnQueryDragIcon()
{
	return (HCURSOR) m_hIcon;
}

/* ///////////////////////////////Comienzo de aplicacion//////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
*/


BOOL CAplicacionDlg::OnInitDialog()
{
```

```
	CDialog::OnInitDialog();
	// Add "About..." menu item to system menu.
	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
		}
	}
	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon
	// TODO: Add extra initialization here
	PrimeraCarga=0;

	MappAlloc(M_DEFAULT,&milAplicacion);
	MsysAlloc(M_DEF_SYSTEM_TYPE,M_DEV0,M_DEFAULT,&milSistema);

	// Defino tres displays Imagen Original/ Procesado MIL/ Procesado Aristotle
	//printf("Sistema Mil inicializado\n");
	InterfazPlx miTarjeta;
	return TRUE;  // return TRUE  unless you set the focus to a control
}



void CAplicacionDlg::OnImagenCargarImagen()
{
	//void*  ErrorCarga;


	// TODO: Add your command handler code here
	static char BASED_CODE szFilter[] = "Mapa de bits (*.bmp)|*.bmp|All Files (*.*)|*.*||";
	CFileDialog DialogoAbrir (true,   "bmp",NULL,NULL,szFilter);
	DialogoAbrir.DoModal();
	//archivo.ReleaseBuffer();
	archivo = DialogoAbrir.GetFileName();
	texto.Format("archivo= %s",archivo);
	AfxMessageBox( texto );
	ImagenPath = DialogoAbrir.GetPathName();
```

```
	if (archivo!="")
	{

		if (PrimeraCarga==1)
		{
			MbufFree(milBuffer1);
			MbufFree(milBuffer2);
			MdispFree(milDisplay1);
			MdispFree(milDisplay2);
		}


	MdispAlloc(milSistema,M_DEV0,M_DEF_DISPLAY_FORMAT,M_DEFAULT,&milDisplay1);

	MdispAlloc(milSistema,M_DEV1,M_DEF_DISPLAY_FORMAT,M_DEFAULT,&milDisplay2);//
/

		MbufDiskInquire(archivo.GetBuffer(0), M_SIZE_X		, &ImagenX) ;
		MbufDiskInquire(archivo.GetBuffer(0), M_SIZE_Y		, &ImagenY) ;


	MbufAllocColor(milSistema,1,ImagenX,ImagenY,8+M_UNSIGNED,M_IMAGE+M_DISP,&mil
Buffer1);

	MbufAllocColor(milSistema,1,ImagenX,ImagenY,8+M_UNSIGNED,M_IMAGE+M_DISP,&mil
Buffer2);

		MbufClear(milBuffer1,193);//borra con el color 193(gris)
		MbufClear(milBuffer2,193);

		MdispSelectWindow(milDisplay1,milBuffer1,m_Imagen1.m_hWnd);

		MdispSelectWindow(milDisplay2,milBuffer2,m_Imagen2.m_hWnd);

//		texto.Format("tras esto error");
//		AfxMessageBox( texto );

		//MappGetError(M_CURRENT,(void*)ErrorCarga);
		//texto.Format("error: %s",ErrorCarga);
		//AfxMessageBox( texto );

		MbufLoad( archivo.GetBuffer(0), milBuffer1);
    archivo.ReleaseBuffer();
    //MappGetError(M_CURRENT,(void*)ErrorCarga);

		//texto.Format("error: %s",ErrorCarga);
		//AfxMessageBox( texto );
//		texto.Format("paso1");
//		AfxMessageBox( texto );
```

```
//			MbufInquire(milBuffer1, M_PITCH_BYTE, &ImagenSize);
		//texto.Format("Tamaño Imagen= %d",ImagenSize);
			//AfxMessageBox( texto );

			//texto.Format("Tamaño X= %d",ImagenX);
			//AfxMessageBox( texto );
			abuffer1=(unsigned char*)malloc(ImagenX*ImagenY);
		   //texto.Format("paso2");
			//AfxMessageBox( texto );

			MbufGet(milBuffer1,(void*)abuffer1);

			//texto.Format("paso3");
			//AfxMessageBox( texto );

			texto.Format("%s",ImagenPath);
			SetDlgItemText(IDC_MenImagen1,texto);

			PrimeraCarga=1;
		}
		else
		{
			texto.Format("No piensas cargar ninguna imagen??");
			AfxMessageBox( texto );
		}
}

void CAplicacionDlg::OnBEnviar()
{
		// TODO: Add your control notification handler code here
		if (archivo!="")
		{
			EnviarImagen();
		}
		else
		{
			texto.Format("Debes cargar un fichero, primero");
			AfxMessageBox( texto );
		}
}

int CAplicacionDlg::EnviarImagen()
{
		int bloques, resto;

		bloques=(ImagenX*ImagenY)/256;
		resto=(ImagenX*ImagenY)%256;
```

```
	//texto.Format("bloques: %d \n resto: %d \n tam: %d",bloques, resto,bloques*32+resto);
	//AfxMessageBox( texto )
	Starttime=GetTickCount();
	miTarjeta.AbrirCanalSglDma(0);
	for(k=0; k<bloques;k++)
	{
		miTarjeta.EscrituraSglDma(0x0,0x256,(PU32)(abuffer1+(k*256)),0);

		miTarjeta.LecturaSglDma(0x0,0x256,(PU32)bufTo+(k*256),0);
	}

		miTarjeta.EscrituraSglDma(0x0,resto,(PU32)(abuffer1+(bloques*256)),0);
		miTarjeta.LecturaSglDma(0x0,resto,(PU32)(bufTo+(bloques*256)),0);
}
void CAplicacionDlg::OnImagenEnviarImagen()
{
	// TODO: Add your command handler code here
	if (archivo!="")
	{
		EnviarImagen();
	}
	else
	{
		texto.Format("Primero debes cargar un fichero");
		AfxMessageBox( texto );
	}
}

void CAplicacionDlg::OnClose()
{
	// TODO: Add your message handler code here and/or call default
	CAplicacionDlg::OnOK();
	/*MbufFree(milBuffer1);
	MbufFree(milBuffer2);
	MdispFree(milDisplay1);
	MdispFree(milDisplay2);
	MsysFree(milSistema);
	MappFree(milAplicacion);*/
	CDialog::OnClose();
}

void CAplicacionDlg::OnMAcercaDe()
{
	// TODO: Add your command handler code here
	CAboutDlg dlgAbout;
	dlgAbout.DoModal();
}

void CAplicacionDlg::OnOK()
```

```
{
	CDialog::OnOK();
	// TODO: Add extra validation here
	free(bufTo);
	free(abuffer1);
	if (PrimeraCarga==1)
		{
			MbufFree(milBuffer1);
			MbufFree(milBuffer2);
			MdispFree(milDisplay1);
			MdispFree(milDisplay2);
		}
	MsysFree(milSistema);
	MappFree(milAplicacion);
	PlxPciDeviceClose(myPlxDevice);

}

void CAplicacionDlg::OnImagenSalir()
{

	// TODO: Add your command handler code here
	CAplicacionDlg::OnOK();
	/*free(bufTo);
	free(abuffer1);
	if (PrimeraCarga==1)
		{
			MbufFree(milBuffer1);
			MbufFree(milBuffer2);
			MdispFree(milDisplay1);
			MdispFree(milDisplay2);
		}
	MsysFree(milSistema);
	MappFree(milAplicacion);
	CDialog::OnOK();*/
}



void CAplicacionDlg::OnDestroy()
{
	CDialog::OnDestroy();
	// TODO: Add your message handler code here

}
```

C.3 Códigos de VHDL

INTERFAZPLX

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity Interfaz is
        generic(width: natural:=8;
                addw: natural:=8);
   port (
      LHOLD: in STD_LOGIC;
      ADS: in STD_LOGIC;
      BLAST: in STD_LOGIC;
      CLK: in STD_LOGIC;
      LWR: in STD_LOGIC;
      PCIin: in STD_LOGIC_VECTOR (width-1 downto 0);
      PCIout: out STD_LOGIC_VECTOR (width-1 downto 0);
      Localout: out STD_LOGIC_VECTOR (width-1 downto 0);
      Localin: in STD_LOGIC_VECTOR (width-1 downto 0);
      addin: in STD_LOGIC_VECTOR (addw-1 downto 0);
      addout: out STD_LOGIC_VECTOR (addw-1 downto 0);
      LHOLDA: out STD_LOGIC:='0';
      DREQ0: out STD_LOGIC:='1';
      WAITL: in STD_LOGIC;
      WAITP: out STD_LOGIC;
      READY: out STD_LOGIC:='1'
   );
end Interfaz;

architecture Interfaz_arch of Interfaz is
SIGNAL readyD: std_logic;
begin

        process(LHOLD,clk)
        begin
        if clk'event and clk='1' then
                if LHOLD='1' then
                        LHOLDA<='1';
                        if ADS='0' then
                                readyD<='0';
                                end if;

                        if BLAST='0' then
```

```
                                readyD<='1';
                        end if;
                else
                        LHOLDA<='0';
                        --readyD<='1';
                end if;

        end if;
        end process;

process (clk)
begin
        if clk'event and clk='1' then
                READY<=readyD;
        end if;
end process;



process(clk)
begin
if clk'event and clk='1' then
        if (readyD='0') and (lwr='1') then
                Localout<=PClin;
--      else
--              Localout<="11111111";
        end if;

        if (readyD='0') and (lwr='0') then

                PClout<=Localin;
        else
                PClout<="11111111";
        end if;
end if;

end process;

WAITP<=WAITL;

end Interfaz_arch;
```

MÓDULO BINARIZACION

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity Binarizacion is
   port (
      color: in STD_LOGIC_VECTOR (7 downto 0);
      sincolor: out STD_LOGIC_VECTOR (7 downto 0);
      enable: in STD_LOGIC;
      clk: in STD_LOGIC;
      le: in STD_LOGIC
   );
end Binarizacion;

architecture Binarizacion_arch of Binarizacion is
signal aux:STD_LOGIC_VECTOR(7 downto 0);
begin

  process(color,clk)
  begin
          if clk'event and clk='0' then
          if enable='0' then
                   if le='1' then
                   aux<=color;
                   end if;

          end if;
          end if;
  end process;
  sincolor<=aux;
end Binarizacion_arch;
```

ANEXO D

Tablas, esquemas y figuras más importantes.

Para facilitar el trabajo con la plataforma se muestran a continuación las tablas, esquemas y figuras de mayor utilidad y uso.

D.1 CONEXIONES ENTRE LA FPGA Y LA PLX.

A continuación se muestran las conexiones realizadas entre la tarjeta de prototipo y la tarjeta XS40 que contiene la FPGA. Para ello hemos diseñado una placa donde instalaremos la tarjeta XS40, y unos conectores donde por medio de cable plano se conectan a los conectores de la tarjeta de Bus PCI.
De la tarjeta de prototipo se utilizará el Bus Local de datos(datos 0-15), el Bus Local de direcciones (direcciones 0-32) y el Bus Local de control. De estos utilizaremos para nuestra aplicación y aplicaciones posteriores, 8 bits de datos, 13 líneas de dirección y 19 líneas de control., estas son:

PConect	Señal	FPGA	PConect	Señal	FPGA
1	**GND**	P52	2	**LCLK**	P13
3	+ 5 V	X	4	**+3.3 V**	P54
5	GND	X	6	BTERM	P75
7	**LBE0**	P3	8	DPO0	X
9	**LBE1**	P4	10	DPO1	X
11	**LBE2**	P16	12	DPO2	X
13	**LBE3**	P14	14	DPO3	X
15	**READY#**	P68	16	**LHOLD**	P49
17	GND	X	18	**LHOLDA**	P48
19	**ADS#**	P27	20	**LW/R#**	P69
21	LSERR#	P28	22	**BLAST#**	P47X
23	BREQ1#	X	24	**BREQ0#**	X
25	**WAIT#**	P29	26	LRESET#	P46
27	ENUM#	X	28	LINT#	P454
29	CCS#	X	30	U/DRE/L#	P44
31	DMP/EOT#	P25	32	U/DAC/L#	P37
33	BIGEND#	X	34	LEDON/IN	X

Tabla E.1: Señales de control conectadas a la FPGA

PConect	Señal	FPGA	PConect	Señal	FPGA
1	+ 5 V	X	2	+ 3.3 V	
3	**LD 00**	P7	4	**LD 01**	P8
5	**LD 02**	P9	6	**LD 03**	P6
7	**LD 04**	P77	8	**LD 05**	P70
9	**LD 06**	P70	10	**LD 07**	P67
11	GND	X	12	GND	X
13	LD 08	X	14	LD 09	X
15	LD 10	X	16	LD 11	X
17	LD 12	X	18	LD 13	X
19	LD 14	X	20	LD 15	X
21	+ 5 V	X	22	+ 3.3 V	X

Tabla E.2: Señales de datos conectadas a la FPGA

PConect	Señal	FPGA	PConect	Señal	FPGA
1	**+ 5 V**	P2	2	**LA 02**	P5
3	**LA 03**	P78	4	**LA 04**	P79
5	**LA 05**	P82	6	**LA 06**	P83
7	**LA 07**	P84	8	LA 08	P59
9	LA 09	P57	10	LA 10	P51
11	LA 11	P56	12	LA 12	P50
13	LA 13	P58	14	LA 14	P60
15	LA 15	X	16	LA 16	X
17	LA 17	X	18	LA 18	X
19	LA 19	X	20	LA 20	X
21	LA 21	X	22	LA 22	X
23	LA 23	X	24	LA 24	X
25	LA 25	X	26	LA 26	X
27	LA 27	X	28	LA 28	X
29	LA 29	X	30	LA 30	X
31	LA 31	X	32	GND	X

Tabla E.3: Señales de dirección conectadas a la FPGA

Para la interfaz se han habilitado las señales marcadas en negrita.

D.2 PINES DE LA TARJETA XS-40.

Los pines de la tarjeta XESS corresponden con los pines del dispositivo FPGA, estos también están conectados a los demás componentes de la tarjeta de la forma que se muestra en la tabla 6.3.

Pin	Nombre	Descripción
1	-	-
2	+ 5 V	-
3	A0 (SRAM)	Dirección 0 de la SRAM.
4	A1 (SRAM)	Dirección 1 de la SRAM.
5	A2 (SRAM)	Dirección 2 de la SRAM.
6	P1.3 (mC)	Pin 3 del puerto 1 del uC.
7	P1.0 (mC)	Pin 0 del puerto 1 del uC.
8	P1.1(mC)	Pin 1 del puerto 1 del uC.
9	P1.2(mC)	Pin 2 del puerto 1 del uC.
10	P0.7 A7/D7(mC) **D7(SRAM)**	Pin 7 del puerto 0 del uC (acceso multiplexado de address/ data) **Dato 7 del SRAM.**
11	-	-
12	-	-
13	**CLK**	**Entrada de reloj programable (100MHz).**
14	/PSEN (mC)	Enable de almacenamiento del uC.
15	-	-
16	A16 (SRAM)**	No conectado.
17	-	-
18	**S5(LED)** RED1(VGA)	**Segmento 5 del Led** Color rojo1 del VGA.
19	**S6(LED)** HSYNC(VGA)	**Segmento 6 del Led** Señal sinc. horizontal del VGA.
20	**S3(LED)** GREEN1(VGA)	**Segmento 3 del Led** Color verde1 del VGA.
21	-	-
22	-	-
23	**S4(LED)** RED0(VGA)	**Segmento 4 del Led** Color rojo0 del VGA.
24	**S2(LED)** GREEN0(VGA)	**Segmento 2 del Led** Color verde0 del VGA.
25	**S0(LED)** BLUE0(VGA)	**Segmento 0 del Led** Color azul0 del VGA.
26	**S1(LED)** BLUE1(VGA)	**Segmento 1 del Led** Color azul1 del VGA.
27	P3.7(/RD) (mC)	Pin 7 del puerto 3 del uC(/RD)(lectura de datos).
28	P2.7(A15) (mC) A15 (SRAM)**	Pin 7 del puerto 2 del uC(address 15 del uC). Address 15 del SRAM(no conectada).

Tabla E.4: Pines de la tarjeta XS40.

29	/ALE(mC)	Enable del latch de direccionamientodel uC.
30	-	-
31	-	-
32	PC_D6(PUERTO PARALELO DATA OUTPUT)*	Dato 6 salida puerto paralelo (señal de modo para el FPGA).
33	-	-
34	PC_D7(PUERTO PARALELO DATA OUTPUT)*	Dato 7 salida puerto paralelo (señal de modo para el FPGA).
35	P0.4(A4/D4) (mC) **D4(SRAM)**	Pin 4 del puerto 0 del uC (acceso multiplexado de address/ data). **Dato 4 del SRAM.**
36	RST	entrada de reset del uC.
37	XTAL1(mC)	Entrada de reloj del uC.
38	P0.3(A3/D3) (mC) **D3(SRAM)**	Pin 3 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 3 del SRAM.**
39	P0.2(A2/D2) (mC) **D2(SRAM)**	Pin 2 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 2 del SRAM.**
40	P0.1(A1/D1) (mC) **D1(SRAM)**	Pin 1 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 1 del SRAM.**
41	P0.0(AO/D0) (mC) **D0(SRAM)**	Pin 0 del puerto 0 del uC(acceso multiplexado de address/data). **Dato 0 del SRAM.**
42	-	-
43	-	-
44	PC_D0(PUERTO PARALELO DATA OUTPUT)	Dato 0 salida puerto paralelo(señal de registro).
45	PC_D1(PUERTO PARALELO DATA OUTPUT)	Dato 1 salida puerto paralelo(señal de registro).
46	PC_D2(PUERTO PARALELO DATA OUTPUT)	Dato 2 salida puerto paralelo.
47	PC_D3(PUERTO PARALELO DATA OUTPUT)	Dato 3 salida puerto paralelo.
48	PC_D4(PUERTO PARALELO DATA OUTPUT)	Dato 4 salida puerto paralelo.
49	PC_D5(PUERTO PARALELO DATA OUTPUT)	Dato 5 salida puerto paralelo.
50	P2.4(A12) (mC) A12(SRAM)	Pin 4 del puerto 2 del uC (address12) Dirección 12 de la SRAM.
51	P2.2(A10) (mC) A10(SRAM)	Pin 2 del puerto 2 del uC (address10) Dirección 10 de la SRAM.
52	**GND(1.4V)**	-
53	-	-
54	**3.3V**	-
55	-	-
56	P2.3 (A11) (mC) A11(SRAM)	Pin 3 del puerto 2 del uC (address11) Dirección 11 de la SRAM.

Tabla E.5: Pines de la tarjeta XS40.(Continuación).

57	P2.1 (A9) (mC) A9(SRAM)	Pin 1 del puerto 2 del uC (address9) Dirección 9 de la SRAM.
58	P2.5 (A13) (mC) A13(SRAM)	Pin 5 del puerto 2 del uC (address13) Dirección 13 de la SRAM.
59	P2.0 (A8) (mC) A8(SRAM)	Pin 0 del puerto 2 del uC (address8) Dirección 8 de la SRAM.
60	P2.6 (A14) (mC) A14(SRAM)	Pin 6 del puerto 2 del uC (address14) Dirección 14 de la SRAM.
61	**/OE (SRAM)**	**Enable de salida del SRAM.**
62	P3.6(/WR) **/WE (SRAM)**	Pin 6 del puerto 3 del uC(/WR)escritura de datos). **Pin de acceso a escritura del SRAM**.
63	-	-
64	-	-
65	**/CE (SRAM)**	**Pin de chip enable del SRAM.**
66	P1.6 (mC) PC_S5 (PUERTO PARALELO STATUS INPUT)	Pin 6 del puerto 1 del uC. Pin de estado 5 del puerto paralelo.
67	P1.7 (mC) VSYNC (VGA INPUTS)	Pin 7 del puerto 1 del uC. Señal de sinc. Vertical para monitor VGA.
68	P3.4 (TO) (mC) KB_CLK (PS/2)	Pin 4 del puerto 3 del uC(TO) Línea de reloj PS/2.
69	P3.1 (TXD) (mC) PC_S6 (PUERTO PARALELO STATUS INPUT) KB_DATA (PS/2)	Pin 1 del puerto 3 del uC(TXD) Pin de estado 6 del puerto paralelo Línea de datos de PS/2.
70	P1.5 (mC) PC_S3 (PUERTO PARALELO STATUS INPUT)	Pin 5 del puerto 1 del uC. Pin de estado 3 del puerto paralelo.
71	-	-
72	-	-
73	-	-
74	-	-
75	PC_S7 (PUERTO PARALELO STATUS INPUT)	Pin de estado del puerto paralelo.
76	-	-
77	P1.4 (mC) PC_S4(PUERTO PARALELO STATUS INPUT)	Pin 4 del puerto 1 del uC Pin de estado 4 del puerto paralelo.
78	A3 (SRAM)	Dirección 3 de la SRAM.
79	A4 (SRAM)	Dirección 4 de la SRAM.
80	P0.6 (A6/D6) (mC) **D6 (SRAM)**	Pin 6 del puerto 0 del uC(acceso multiplexado de address/ data). **Dato 6 del SRAM.**
81	P0.5 (A5/D5) (mC) **D5 (SRAM)**	Pin 5 del puerto 0 del uC(acceso multiplexado de address/ data). **Dato 5 del SRAM**
82	A5 (SRAM)	Dirección 5 de la SRAM
83	A6 (SRAM)	Dirección 6 de la SRAM
84	A7 (SRAM)	Dirección 7 de la SRAM

Tabla E.6: Pines de la tarjeta XS40.(Continuación)

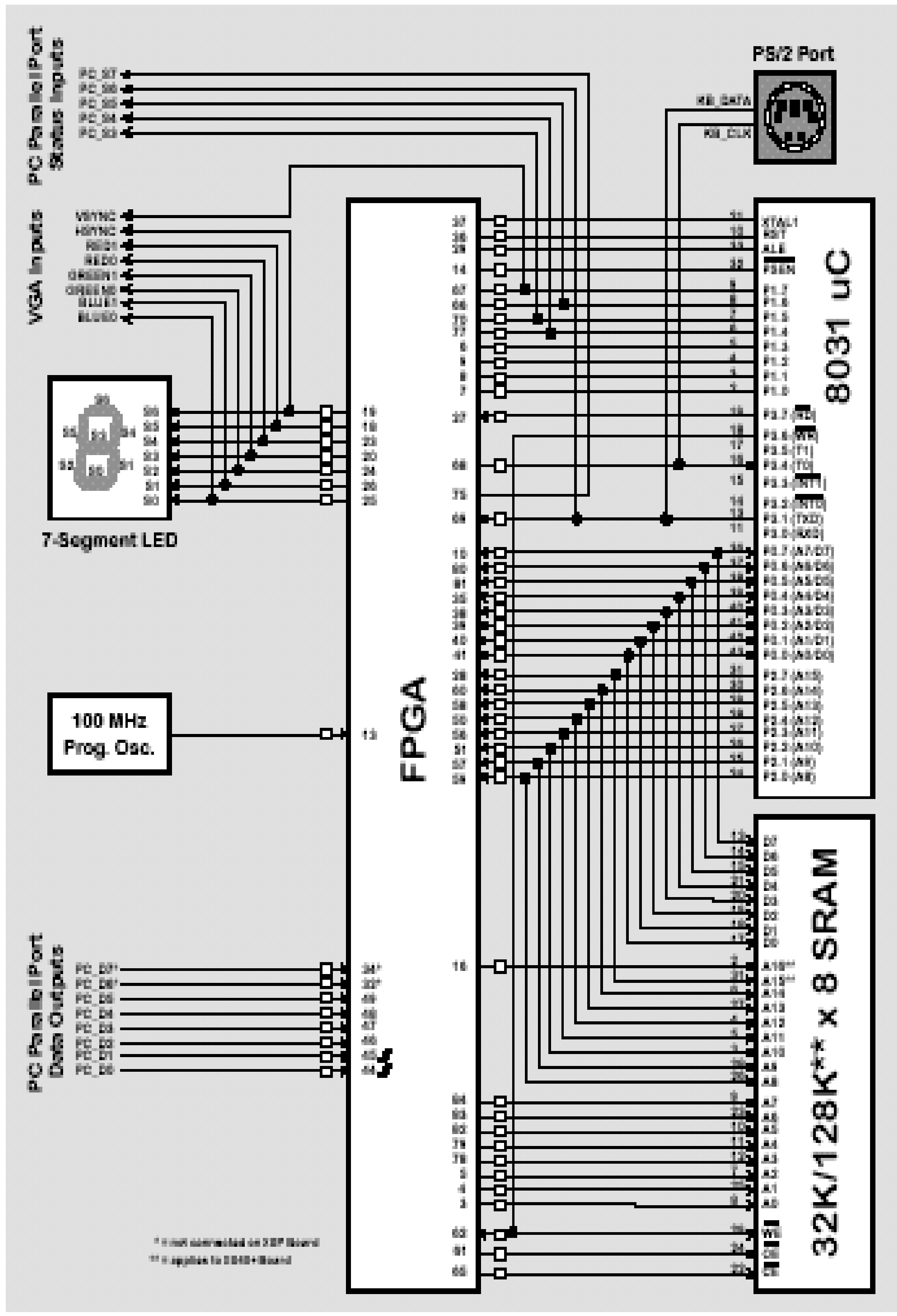
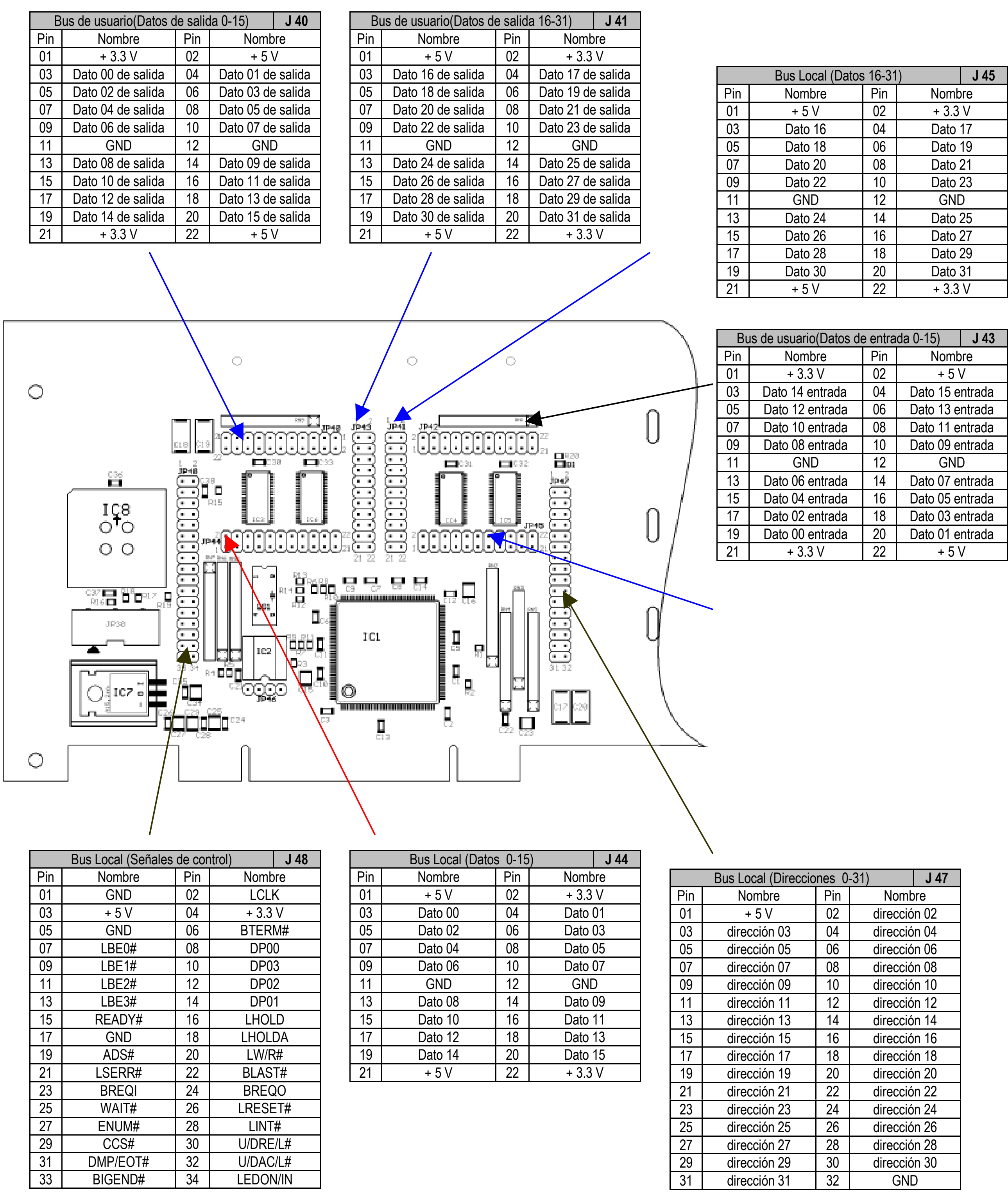


Figura E.1: Arquitectura de la tarjeta XS40.

D.4 PINES DE LA TARJETA PCI PROTO- LAB/PLX



Bus de usuario(Datos de salida 0-15)			J 40
Pin	Nombre	Pin	Nombre
01	+ 3.3 V	02	+ 5 V
03	Dato 00 de salida	04	Dato 01 de salida
05	Dato 02 de salida	06	Dato 03 de salida
07	Dato 04 de salida	08	Dato 05 de salida
09	Dato 06 de salida	10	Dato 07 de salida
11	GND	12	GND
13	Dato 08 de salida	14	Dato 09 de salida
15	Dato 10 de salida	16	Dato 11 de salida
17	Dato 12 de salida	18	Dato 13 de salida
19	Dato 14 de salida	20	Dato 15 de salida
21	+ 3.3 V	22	+ 5 V

Bus de usuario(Datos de salida 16-31)			J 41
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 16 de salida	04	Dato 17 de salida
05	Dato 18 de salida	06	Dato 19 de salida
07	Dato 20 de salida	08	Dato 21 de salida
09	Dato 22 de salida	10	Dato 23 de salida
11	GND	12	GND
13	Dato 24 de salida	14	Dato 25 de salida
15	Dato 26 de salida	16	Dato 27 de salida
17	Dato 28 de salida	18	Dato 29 de salida
19	Dato 30 de salida	20	Dato 31 de salida
21	+ 5 V	22	+ 3.3 V

Bus Local (Datos 16-31)			J 45
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 16	04	Dato 17
05	Dato 18	06	Dato 19
07	Dato 20	08	Dato 21
09	Dato 22	10	Dato 23
11	GND	12	GND
13	Dato 24	14	Dato 25
15	Dato 26	16	Dato 27
17	Dato 28	18	Dato 29
19	Dato 30	20	Dato 31
21	+ 5 V	22	+ 3.3 V

Bus de usuario(Datos de entrada 0-15)			J 43
Pin	Nombre	Pin	Nombre
01	+ 3.3 V	02	+ 5 V
03	Dato 14 entrada	04	Dato 15 entrada
05	Dato 12 entrada	06	Dato 13 entrada
07	Dato 10 entrada	08	Dato 11 entrada
09	Dato 08 entrada	10	Dato 09 entrada
11	GND	12	GND
13	Dato 06 entrada	14	Dato 07 entrada
15	Dato 04 entrada	16	Dato 05 entrada
17	Dato 02 entrada	18	Dato 03 entrada
19	Dato 00 entrada	20	Dato 01 entrada
21	+ 3.3 V	22	+ 5 V

Bus Local (Señales de control)			J 48
Pin	Nombre	Pin	Nombre
01	GND	02	LCLK
03	+ 5 V	04	+ 3.3 V
05	GND	06	BTERM#
07	LBE0#	08	DP00
09	LBE1#	10	DP03
11	LBE2#	12	DP02
13	LBE3#	14	DP01
15	READY#	16	LHOLD
17	GND	18	LHOLDA
19	ADS#	20	LW/R#
21	LSERR#	22	BLAST#
23	BREQI	24	BREQO
25	WAIT#	26	LRESET#
27	ENUM#	28	LINT#
29	CCS#	30	U/DRE/L#
31	DMP/EOT#	32	U/DAC/L#
33	BIGEND#	34	LEDON/IN

Bus Local (Datos 0-15)			J 44
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	+ 3.3 V
03	Dato 00	04	Dato 01
05	Dato 02	06	Dato 03
07	Dato 04	08	Dato 05
09	Dato 06	10	Dato 07
11	GND	12	GND
13	Dato 08	14	Dato 09
15	Dato 10	16	Dato 11
17	Dato 12	18	Dato 13
19	Dato 14	20	Dato 15
21	+ 5 V	22	+ 3.3 V

Bus Local (Direcciones 0-31)			J 47
Pin	Nombre	Pin	Nombre
01	+ 5 V	02	dirección 02
03	dirección 03	04	dirección 04
05	dirección 05	06	dirección 06
07	dirección 07	08	dirección 08
09	dirección 09	10	dirección 10
11	dirección 11	12	dirección 12
13	dirección 13	14	dirección 14
15	dirección 15	16	dirección 16
17	dirección 17	18	dirección 18
19	dirección 19	20	dirección 20
21	dirección 21	22	dirección 22
23	dirección 23	24	dirección 24
25	dirección 25	26	dirección 26
27	dirección 27	28	dirección 28
29	dirección 29	30	dirección 30
31	dirección 31	32	GND

Figura E.2 Pines de la tarjeta de prototipo.

D.5 DIAGRAMA DE CONEXIONES Y JUMPERS.

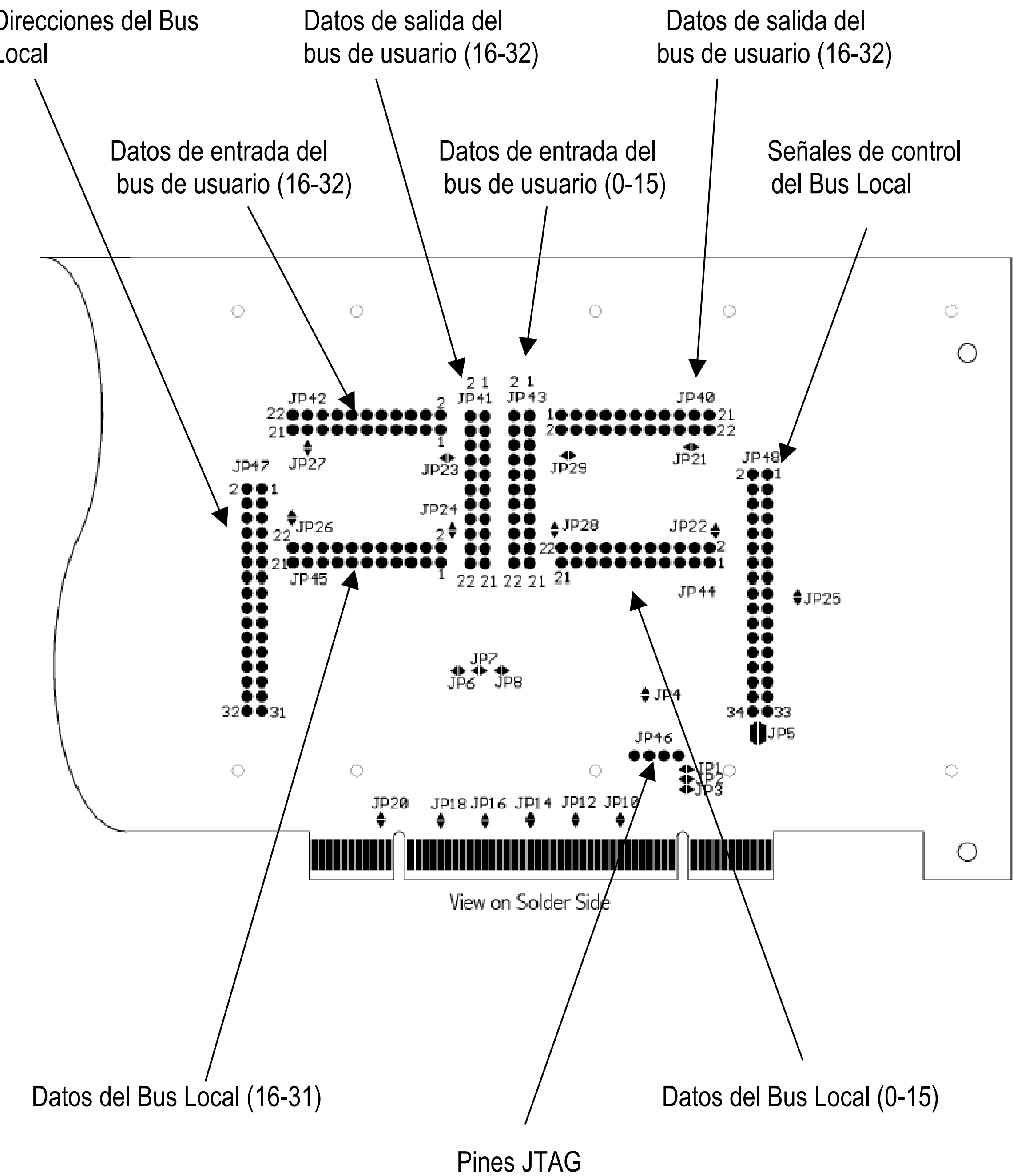


Figura E.3: Diagrama de conexiones y jumpers.

D.3 DEFINICION DE LA CLASE INTERFAZPLX

```
Class InterfazPlx

- HANDLE miDispositivoPlx;
- PCI_MEMORY PciMemory;
- RETURN_CODE rc;
- DEVICE_LOCATION device;

+ InterfazPlx();
+ InterfazPlx(DEVICE_LOCATION dev);
+ virtual ~InterfazPlx();
+ static int NumeroDispositivos();
+ static int NumeroDispositivosCriterio(DEVICE_LOCATION dev);
+ static int BuscaDispositivo(U32 num);
- void CerrarDispositivo();
+ int EscrituraMS(U32 address, PU32 datos);
+ int LecturaMS(U32 address, PU32 datos);
+ int CerrarCanalSglDma(int nC);
+ int AbrirCanalSglDma(int nC);
+ int EscrituraSglDma(U32 direccion,U32 tamano,PU32 datos,int nC);
+ int LecturaSglDma(U32 direccion,U32 tamano,PU32 datos, int nC);
+ int AbrirCanalBlockDma(int nC);
+ int CerrarCanalBlockDma(int nC);
+ PU32 EscrituraBlockDma(U32 direccion,U32 tamano,PU32 datos,int nC);
+ PU32 LecturaBlockDma(U32 direccion,U32 tamano,PU32 datos,int nC);
+ int AbrirCanalShuttleDma(int nC);
+ int CerrarCanalShuttleDma(int nC);
+ int EscrituraShuttleDma(U32 direccion,U32 tamano,PU32 datos,int nC);
+ int LecturaShuttleDma(U32 direccion,U32 tamano,PU32 datos,int nC);
+ void EscribirRegistro(U32 registro, PU32 dato);
+ U32 LeerRegistro(U32 registro);
+ U32 InterfazPlx::LeerRegistroConfiguracion(U32 registro);
+ void InterfazPlx::EscribirRegistroConfiguracion(U32 registro, U32 dato);
+ U32 InterfazPlx::DireccionFisicaBuffer();
+ U32 InterfazPlx::BufferSize();
+ U32 InterfazPlx::DireccionLogicaBuffer();
```

Tabla E.7: Clase InterfazPlx

Bibliografía.

[PLXMon, 2000]	"PLXMon User´s Manual v3.1" Zhen Chen, PLX Technology, Inc. 2000
[SDK User, 2000]	"PCI Host SDK User´s Manual v3.1" Zhen Chen, PLX Technology, Inc. 2000
[Manufact., 2000]	"Manufacturing Test Specification v3.1" PLX Software, PLX Technology, Inc. 2000
[SDK Progammer, 2000]	"PCI Host SDK Programmer´s Manual v3.1" PLX Software, PLX Technology, Inc. 2000
[9054 Hardware, 1999]	"PCI 9054RDK-860 Hardware Reference Manual v1.1" Joe Hurst, PLX Technology, Inc. 1999
[9054 D. Book, 2000]	"PCI 9054, Data Book v2.1" Trish, PLX Technology, Inc. 2000
[PCI Syst. Architec., 1999]	"PCI System Architecture Fourth Edition" Tom Shanley, Don Anderson, MindShare, Inc.1999
[PCI 32, 1998]	"PCI32 4000 Master & Slave Interfaces v2.0" Linda Patrick, Xilinx, Inc. 1998
[PCI Proto, 2001]	"PCI Proto-Lab/PLX. Technical Manual v1.1" Hinz, HK Meβsysteme GmbH, 2001
[Visión, 2000]	"Arquitectura Hardware para Visión Artificial" Pedro Javier Navarro Lorente, Universidad Politécnica de Cartagena. 2000
[Mach4A, 1998]	"ispm4A Mach 4" Coughlin, Lattice Semiconductor Coprporation. 1998
[Latt, Pro, 2000]	"Lattice Pro Programming Software v8.0" Nancy Knowlton, Lattice Semiconductor Coprporation. 1998

- [isp Cable, 2000] “ispDownload Cable v8.0”
  Lattice Semiconductor Coprporation. 2000

- [Isp Expert, 2000] “ispDesign Expert Tutorial & User´s Manual”
  F. Couch,Hinz,
  Lattice Semiconductor Coprporation. 2000

- [GXSTOOLS, 2001] “GXSTOOLS v3.3. User´s Manual. XS Board utilities with graphical user interfaces”
  Dave Venden Boult,
  XESS Corporation. 2001

- [VHDL Ref, 2001] “VHDL Reference Guide”
  Xilinx Inc. 2001

- [XS40, 2000] “XS40, XSP Board v1.4 User´s Manual”
  Dave Venden Boult,
  XESS Corporation. 2001

- [Pagmatic, 2001] “Pragmatic Logic Design”
  David E. Venden Boult
  XESS Corporation. 2001

- [Foundation,2001] “Xilinx Foundation 2.1i”
  Xilinx Inc. 2001

- [XC4000, 1999] “XC4000E & XC4000X Series Field Programmable Gate Arrays v1.6l”
  Xilinx Inc. 1999

- [VisualC++(1), 1999] “Visual C++. Aplicaciones para Win32”
  Fco. Javier Ceballos
  Ed.ra-ma. 1999.

- [VisualC++(2), 1999] “Aprenda Visual C++ 6.0 Ya”
  Chuck Sphar
  Microsoft Press. 1999.

- [Pract Xilinx, 1998] “The Practical Xilinx Designer Lab Book”
  Dave Van den Bout.
  Ed. Prentice Hall, 1998

- [Doc1.MIL99] MIL Online Documentation.
  Buffer management commands

- [Doc2.MIL99] On Line reference manual

[Rincón 01] “Diseño de una plataforma para prototipos de tarjeta PCI” Javier Rincón.

[UNED 00] “Estructura y tecnología de computadores” S. Dormido, Mª Antonia Canto, José Mira, Ana E. Delgado

[Paraninfo 96] “Fundamentos de computadores” Pedro de Miguel Anasagasti

Recursos on-line:

Página web de PLX Technology, Inc

http://www.plxtech.com

Grupo de usuarios de PCI España

http://www.pcisig.com/faq.txt
http://www.plcisig.com/forum.htlm

Página web de HK Meβsysteme GmbH

http://www.pci-tools.com

Página web de Lattice Semiconductor Coprporation
http://www.latticesemi.com

Página web de Xilinx, Inc.

http://www.xilinx.com