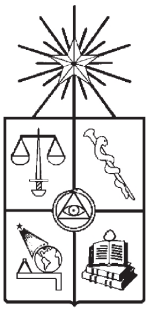


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

IMPLEMENTACIÓN DE INTERFAZ PCI SOBRE PLATAFORMA INDUSTRIAL BASADA EN DISPOSITIVO FPGA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRICISTA

ENRIQUE EFRAÍN ROMÁN ASENJO

PROFESOR GUÍA:
MAURICIO BAHAMONDE BARROS

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
FRANCISCO RIVERA SERRANO

SANTIAGO DE CHILE
ABRIL 2009

RESUMEN DE LA MEMORIA
PARA OPTAR AL TITULO DE
INGENIERO CIVIL ELECTRICISTA
POR: ENRIQUE ROMAN A.
FECHA: 21/04/2009
PROF. GUIA: Sr. MAURICIO BAHAMONDE B.

"IMPLEMENTACIÓN DE INTERFAZ PCI SOBRE PLATAFORMA INDUSTRIAL BASADA EN DISPOSITIVO FPGA"

ISIS es una placa madre industrial desarrollada en Chile por Continental Lensa S.A orientada al soporte de *SoPCs* (*Systems on a Programmable Chip*) sobre un dispositivo FPGA (*Field Programmable Gate Array*), integrado con una serie de periféricos on-board. La capacidad de soportar SoPCs basados en el procesador Nios II y el sistema operativo uClinux, en conjunto con diversos núcleos de hardware de propiedad intelectual o *IP cores*, abre un universo de aplicaciones que abarca desde el control de sistemas, procesamiento digital de señales, y sistemas de radio y televisión digital.

ISIS incorpora un conector PMC (PCI Mezzanine Card), que corresponde a una especificación mecánica para sistemas PCI de montaje paralelo y tamaño pequeño, contrario al estándar PCI convencional donde las tarjetas se montan en forma perpendicular. Sin embargo, no es posible controlar dispositivos PCI con la plataforma ISIS sin un adecuado soporte de hardware y software que provea una interfaz de bus acorde a los requerimientos del estándar PCI.

El presente trabajo otorga a la plataforma ISIS soporte para conectividad con dispositivos PCI 3.3V 32 bit @ 33 MHz. El trabajo aporta la implementación de un chipset PCI embebido en el dispositivo FPGA, el soporte de software para operación con el sistema operativo uClinux, y una aplicación para control y diagnóstico del hardware. Además, se aporta un nuevo hardware que brinda una solución a la incompatibilidad entre los complejos estándares mecánicos PCI Mezzanine Card y PCI convencional de PC.

Uno de los aportes es la implementación del IP core de libre distribución PCI Bridge de Opencores con interfaz de bus Wishbone, en un SoPC con arquitectura de comunicación nativa Avalon System Interconnect Fabric, lo que requiere implementar lógica de adaptación entre dos estándares de interconexión SoC incompatibles. Además, los requerimientos del sistema exigen que el IP core PCI Bridge sea implementado en modo *Host*, estando disponible solamente con pruebas de operación en modo *Guest*, lo que implica el desafío de implementar funcionalidades que no cuentan con un proceso de validación. También se desarrolla una capa de software que comunica el hardware PCI con el kernel de Linux, y un programa que permite el control y diagnóstico de los dispositivos presentes en el bus.

El presente trabajo se integra como parte fundamental del equipo de radiodifusión digital de tercera generación GSD-21 Exgine. El núcleo de hardware del equipo lo constituye la plataforma ISIS integrada con el dispositivo PCI DUC-II (*Next Generation Digital Up Converter*), por medio de los sistemas de hardware y software desarrollados. Se obtiene una tasa de transferencia promedio de 14,5 MByte/s para transferencias PCI usando DMA, y una tasa de error de bus igual a cero para 24 horas de operación sin interrupciones del equipo GSD-21.

Índice general

Índice de figuras

Índice de tablas

Glosario

FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP core	Núcleo de hardware de propiedad intelectual
IP wrapper	Encapsulador de hardware de propiedad intelectual
ISIS	Industrial Support for Integrated Systems
MMU	Memory Manegement Unit
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
SoC	System on a Chip
SoPC	System on a Programmable Chip
Test Bench	Marco de simulación

1 Introducción

1.1 Motivación

PCI es un estándar de bus local diseñado para sistemas basados en microprocesadores y periféricos de alta velocidad, incluyendo sistemas de audio y video, adaptadores de red, tarjetas aceleradoras gráficas, y controladores de almacenamiento de datos. La compatibilidad con el estándar PCI requiere una gran cantidad de pines de entrada/salida, satisfacción de requerimientos eléctricos y de potencia, alta densidad de señales, y adhesión a una serie de estrictos requerimientos de sincronización temporal o *timing*. El estándar PCI es uno de los más populares en la industria de computadores, y a la vez uno de los más complejos y bien documentados.

ISIS (figura 1) es una plataforma industrial desarrollada en Chile por Continental Lensa S.A, orientada al soporte de *SoPCs* (*Systems on a Programmable Chip*) sobre un dispositivo FPGA Cyclone II de Altera. Conectados al dispositivo FPGA se encuentran diversos periféricos tales como un chip de memoria SDRAM de 32 MB, memoria Flash, controlador USB, chip Ethernet 10/100 MAC/PHY, puerto RS-232, test points, botones, LEDs indicadores, conectores genéricos para acceso a más de 128 pines de entrada/salida, y un conector PCI Mezzanine (conectores X10 y X13 en la figura 1). La capacidad de soportar SoPCs basados en el procesador Nios II de Altera y el sistema operativo uClinux, en conjunto con diversos núcleos de hardware de propiedad intelectual o *IP cores*, abre un universo de aplicaciones que abarca desde el control de sistemas, procesamiento digital de señales, y sistemas de radio y televisión digital.

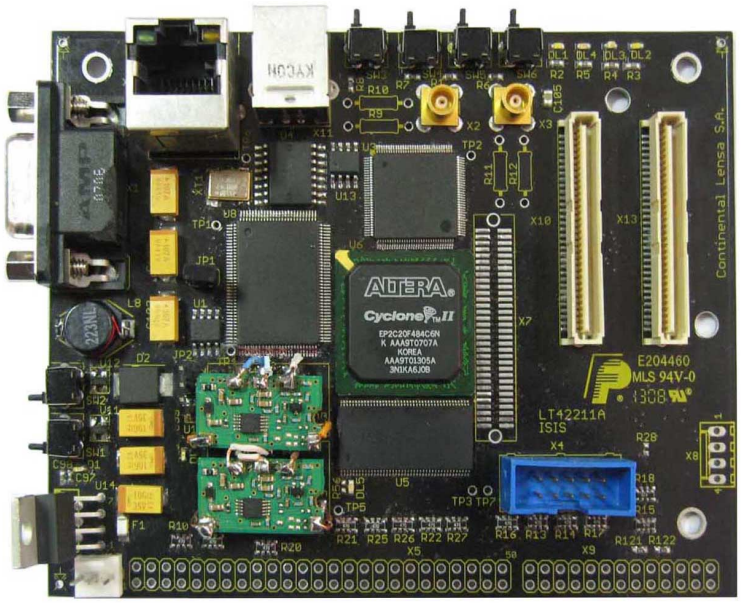


Figura 1: Plataforma ISIS

El estándar PMC (*PCI Mezzanine Card*) [1] corresponde a una especificación mecánica para sistemas y tarjetas PCI de montaje paralelo y tamaño pequeño como se muestra en la figura 2, contrario al estándar PCI convencional donde las tarjetas se montan en forma perpendicular. Luego, no es posible conectar directamente una tarjeta PCI convencional a la plataforma ISIS, y por lo tanto se requiere de un hardware de adaptación especial conforme a ambos estándares (PCI convencional y PCI Mezzanine).

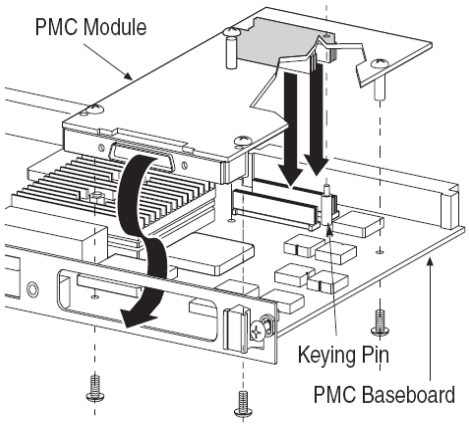


Figura 2: Montaje de sistemas PCI Mezzanine

Más aún, tampoco es posible conectar directamente tarjetas PMC a la plataforma ISIS debido a la falta de circuitería de potencia acorde a los requerimientos de la especificación PCI. Además, sin un IP core controlador de bus PCI embebido en el dispositivo FPGA, configurado apropiadamente y con un software que lo pueda controlar, no es posible conectar ningún tipo de dispositivo PCI a la plataforma ISIS.

En el mercado existe una variedad de IP cores controladores de bus PCI que tienen la ventaja de reducir el tiempo de desarrollo por medio de una solución portable, sometida a rigurosas pruebas que validan su funcionalidad, con soporte de software y completos ambientes de simulación y verificación. Sin embargo, el costo de tal solución es alto (decenas de miles de dólares), debido al costo de desarrollo que implica un hardware de tal complejidad. En efecto, el desarrollo de un IP core complejo desde cero puede tomar más de un año dependiendo de los recursos involucrados (horas-ingeniero).

Una solución alternativa a los IP cores presentes en el mercado es el uso de IP cores *open source*, como por ejemplo el IP core PCI Bridge de Opencores[1]. El uso de IP cores de libre distribución reduce drásticamente el costo de desarrollo, con la desventaja de no contar con soporte y requerir ingeniería para adaptar incompatibilidades de hardware, resolver diferencias entre arquitecturas de comunicación, dar solución a fallas en el diseño del IP core, incorporar nuevas funcionalidades para satisfacer los requerimientos, y desarrollar software para control y diagnóstico del hardware. Todo lo anterior con el apoyo de documentación que puede ser inconsistente, y el riesgo que involucra implementar una solución que no cuenta con un proceso de validación acorde a los requerimientos de la industria.

1 http://www.opencores.org

El presente trabajo aborda el desafío de otorgar a la plataforma ISIS conectividad con dispositivos PCI mediante el diseño de un hardware de adaptación entre los estándares PCI convencional y PCI Mezzanine, y la implementación del IP core de libre distribución PCI Bridge de Opencores sobre el dispositivo FPGA de la plataforma ISIS.

El desarrollo de un hardware de adaptación PCI a PMC no solo requiere de la satisfacción mecánica de dos estándares complejos, sino que también de un cuidadoso diseño electrónico que permita la operación del circuito impreso como un ambiente de transmisión de señal y no como un simple medio de interconexión entre puntos, para asegurar el correcto funcionamiento del bus y evitar problemas de ruido, integridad de señal, sobretensión, y ancho de banda. Además el diseño debe satisfacer los requerimientos de potencia de la especificación PCI, además de un adecuado manejo termal. Todo lo anterior bajo criterios de costo mínimo (área, componentes, número de capas de circuito, etc.).

Por otro lado, la implementación del IP core PCI Bridge sobre la plataforma ISIS involucra adaptación de protocolos de comunicación incompatibles, desarrollo de drivers, y el uso de funcionalidades y características no validadas, lo cual implica un alto riesgo pero al mismo tiempo un gran desafío de ingeniería.

1.2 Objetivos

El presente trabajo tiene como objetivo otorgar soporte para conectividad con tarjetas PCI 3.3V @ 33 MHz de PC, a la primera placa madre basada en computación reconfigurable desarrollada en Chile. Para esto se definen los siguientes objetivos generales:

1. Implementar sobre la plataforma ISIS un SoPC basado en el procesador Nios II y sistema operativo uClinux, que contenga el IP core PCI Bridge de Opencores configurado como Host, para otorgar control sobre dispositivos PCI a través del conector PMC. Proveer al sistema operativo uClinux un driver apropiado para controlar el hardware PCI.
2. Diseño e implementación de un hardware de adaptación para conectar tarjetas PCI 3.3V 32 bit @ 33 MHz de PC a la plataforma ISIS. El hardware debe proveer la potencia de alimentación a la tarjeta PCI.
3. Validar el sistema conectando una tarjeta PCI 3.3V 32 bit @ 33 MHz de PC a la plataforma ISIS por medio del hardware de adaptación PCI a PMC, y corriendo un software de aplicación que controle el dispositivo PCI. Obtener la tasa de transferencia, tasa de error, y caracterizar las formas de onda presentes en el bus.

1.3 Estructura del trabajo

En los capítulos 2 y 3 se exponen los fundamentos teóricos asociados a la interconexión de núcleos de hardware en SoCs. Se aborda el problema de encapsulación de hardware, metodologías de conversión de protocolos, fundamentos de buses de computadores, y el estándar PCI. En el capítulo 4 se aborda la descripción e implementación del sistema, y los protocolos de prueba para someter el sistema a un proceso de validación. En el capítulo 5 se describen los resultados obtenidos de las pruebas, y finalmente en los capítulos 6 y 7 se exponen las extensiones y líneas de investigación futuras, y las conclusiones finales del presente trabajo.

2 Introducción a las redes de interconexión en SoC

Un sistema digital se compone de tres bloques de construcción básicos: lógica, memoria, y comunicación. La lógica transforma y combina datos, por ejemplo realizando operaciones aritméticas o tomando decisiones. La memoria almacena datos en el tiempo para su posterior uso o modificación. La comunicación mueve los datos de un lugar a otro. Las *redes de interconexión* se usan para transportar datos entre subsistemas de un sistema digital.

Actualmente la limitación de los sistemas digitales no está en la lógica o memoria, sino en la interconexión de subsistemas. La mayor parte de la potencia se usa para cargar cables o *wires* (pistas o conductores metálicos en un chip), y la mayor parte del ciclo de una señal de reloj se ocupa en retardo de cables y no en retardo de compuertas lógicas. La densidad de pines y de cableado aumenta una tasa menor que la densidad de los componentes a interconectar. Por otro lado la frecuencia de comunicación entre componentes está quedando atrás en comparación a las frecuencias de clock de procesadores modernos. Los factores anteriores hacen de las redes de interconexión el cuello de botella y factor clave en el rendimiento de los sistemas digitales modernos.

2.1 Definición

Una red de interconexión es un sistema programable que transporta datos entre terminales, como se muestra en la figura 3. Un terminal *Ti* que se comunica con un terminal *Tj*, envía a la red un *mensaje* que contiene los datos, y ésta lo despacha al terminal *Tj*. La red es programable en el sentido de que hace diferentes conexiones en diferentes instantes. La red es un sistema porque se compone de un conjunto de elementos tales como buffers, canales, *switches*, y bloques lógicos de control que operan en conjunto para enviar datos.

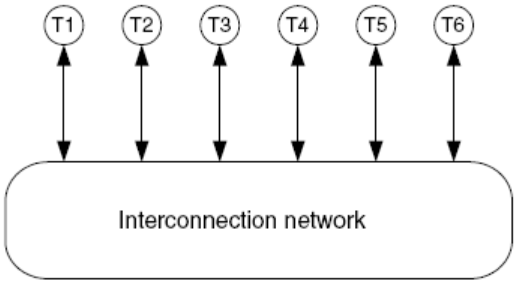


Figura 3: Red de interconexión genérica

La definición anterior es aplicable a distintas escalas. Por ejemplo, en un procesador existen redes de interconexión para transportar datos entre arreglos de memoria, registros, y unidades aritméticas. En *SoCs* (*Systems on Chip*) existen redes de interconexión o *SoC fabric* para integrar diversos núcleos lógicos de propiedad intelectual (procesadores, controladores de bus, memoria, audio, video, etc.) en un solo sistema de alto nivel. A nivel de placa de circuito impreso o *PCB* (*Printed Circuit Board*) se encuentran redes que conectan procesadores con chips de memoria, o puertos de salida con puertos de entrada. Las redes de interconexión a niveles de sistema e

inferiores se caracterizan por altas tasas de transferencia y canales de corta dimensión, y por lo tanto requieren soluciones distintas a las que se encuentran en redes de gran escala como Internet.

2.2 Clasificación de redes de interconexión

A continuación se describen los principales criterios con que se clasifican las redes de interconexión:

2.2.1 Modo de operación

Las redes de interconexión se pueden clasificar según el modo de operación en *sincrónicas* o *asíncronas*. En modo de operación sincrónico todas las operaciones se realizan solamente en transiciones de una señal de reloj o *clock* global. En modo de operación asíncrono las operaciones se realizan por medio de un protocolo de negociación o *handshaking*, sin necesidad de una señal de *clock*.

2.2.2 Estrategia de control

Según la estrategia de control se pueden clasificar en control *centralizado* o *descentralizado*. En esquemas de control centralizado, un único sistema central controla las operaciones del sistema. En sistemas de control descentralizado, las funciones de control se encuentran en forma distribuida entre distintos componentes del sistema.

2.2.3 Técnicas de switching

Según el mecanismo de *switching* se pueden clasificar en *conmutación de circuitos* o *packet switching* (conmutación de paquetes). En conmutación de circuitos debe establecerse un camino entre origen y destino previo al inicio de la comunicación. El camino establecido existe durante todo el periodo de comunicación. En conmutación de paquetes la comunicación se establece por medio de mensajes que son divididos en entidades más pequeñas llamadas paquetes. Existen diversos métodos de conmutación de paquetes, a continuación se describen los dos más conocidos:

- *Store and forward*: Todos los datos de un paquete entrante a un enlace se almacenan en un *buffer* intermedio para conmutación y reenvío.
- *Wormhole routing*: El paquete entrante se reenvía justo después de que su encabezado ha sido identificado, y el paquete completo sigue al encabezado sin ninguna discontinuidad.

2.2.4 Clasificación según topología

La topología de la red de interconexión es una función desde el conjunto de terminales (procesadores, memorias, etc.) al mismo conjunto de terminales, es decir, la topología describe la forma en que se conectan entre sí los componentes del sistema. En la figura 4 se muestra una

clasificación de redes de interconexión basada en topología.

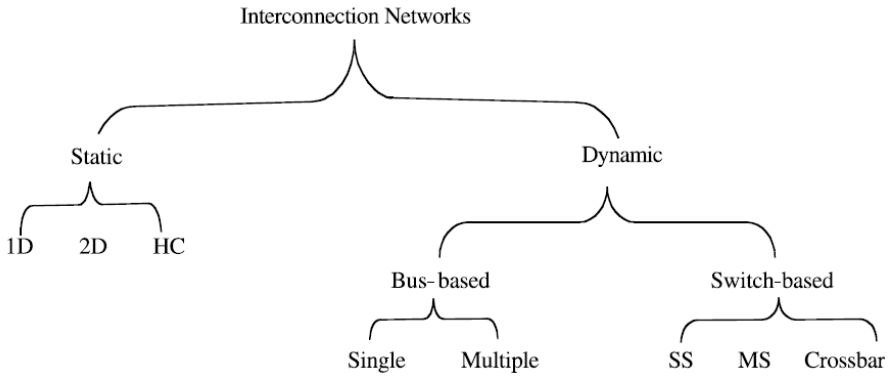


Figura 4: Clasificación de redes de interconexión basada en topología

En general, las redes de interconexión se pueden clasificar en redes *estáticas* o *dinámicas*. En redes estáticas existe un conjunto de enlaces fijos que conforman la red. En redes dinámicas, las conexiones se establecen en el tiempo a medida que se necesitan. Las redes estáticas pueden ser clasificadas según su patrón de interconexión como *1D* (una dimensión), *2D* (2 dimensiones), o *HC* (hipercubo). Por otro lado las redes de interconexión dinámicas se pueden clasificar según el esquema de interconexión en redes basadas en *bus* o redes basadas en *switch*. Las redes basadas en bus pueden ser de bus único o múltiples buses. Las redes basadas en switch se clasifican según su estructura en *etapa única* o *SS* (*single-stage*), *multietapa* o *MS* (*multistage*), o redes *crossbar*.

2.3 Redes de interconexión basadas en switch

La conmutación de circuitos es una forma de control de flujo *sin buffer*, que opera creando un canal o circuito para la transmisión de información entre origen y destino, es decir, los caminos se crean en la medida que se necesitan. Cuando no hay más información que transmitir o recibir, el canal se deshace. Según topología, las redes basadas en *switch* se pueden clasificar en *crossbar*, *single-stage*, y *multistage*.

2.3.1 Redes Crossbar

Una red *crossbar* de *n* x *m* conecta directamente *n* entradas con *m* salidas, sin etapas intermedias. Actualmente, la mayoría de las redes de interconexión *crossbar* tienen una estructura basada en multiplexores como se muestra en la figura 5. Cada una de las *n* entradas se conecta a una de las entradas de *m n:1* multiplexores. Las *m* salidas son las salidas de los *m* multiplexores.

La red *crossbar* se dice que es *no bloqueante* porque permite múltiples patrones de interconexión en forma simultánea. Más precisamente, se dice que una red es no bloqueante si permite cualquier patrón de interconexión que sea una permutación del conjunto de entradas con el conjunto de salidas.

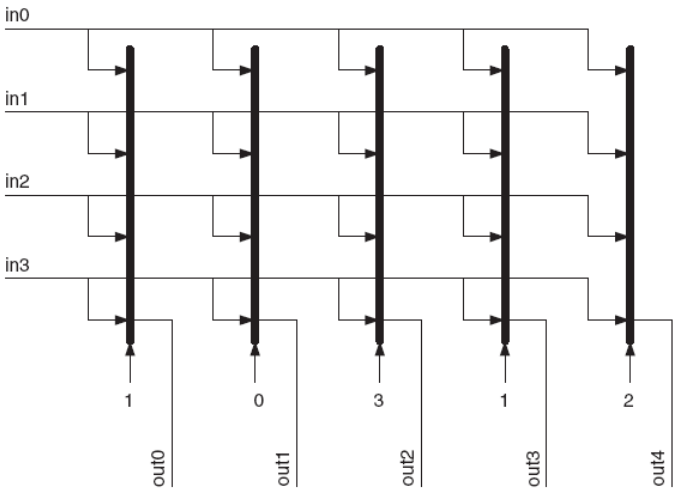


Figura 5: Ejemplo de red de interconexión crossbar

2.3.2 Redes Single-Stage

En redes *single-stage* existe una única etapa de conmutadores o *SEs* (*switching elements*) entre las entradas y salidas de la red como se muestra en la figura 6.

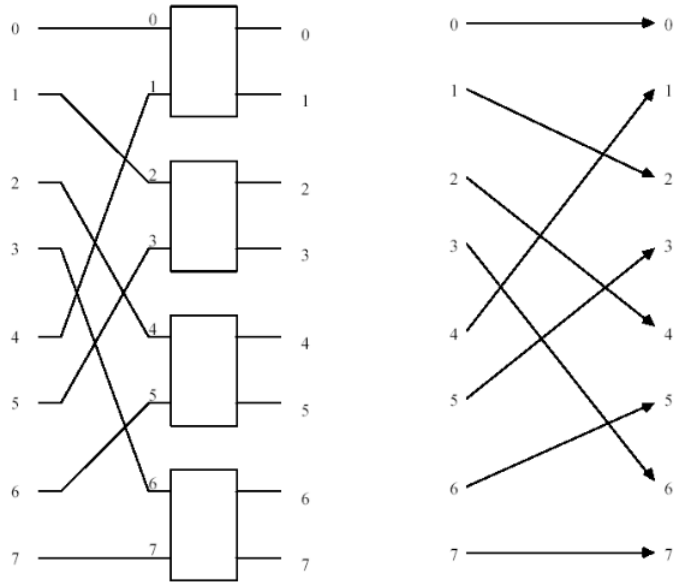


Figura 6: Ejemplo de red de interconexión single-stage

Para establecer una comunicación entre un cierto par origen-destino, puede ser necesario hacer recircular los datos en la red una cierta cantidad de veces. En el ejemplo de la figura se muestra un patrón de interconexión llamado *single-stage Shuffle-Exchange*. También destacan los patrones *Cúbico*, y *PM2I* (*Plus-Minus* 2^i).

2.3.3 Redes Multietapa

En general, una red *multistage* o multietapa es una red compuesta por un número de etapas, donde cada etapa consiste en un conjunto de *2 x 2* (2 entradas, 2 salidas) *SEs*. Las etapas se conectan entre sí mediante un cierto patrón de interconexión o ISC (*Interstage Connection*). Este patrón de interconexión entre etapa puede tener cualquier función, tal como Cúbica, *Shuffle-Exchange*, *Butterfly*, etc.

En la figura 7 se muestra una red multietapa de 8 nodos, con topología b*utterfly* o mariposa. Los datos fluyen desde los nodos de entrada (círculos a la izquierda) a través de 3 etapas de 2 x2 SEs (rectángulos) hacia los nodos de salida (círculos a la derecha). Los SEs están etiquetados con el par (etapa, dirección). Las flechas indican que todos los canales son unidireccionales.

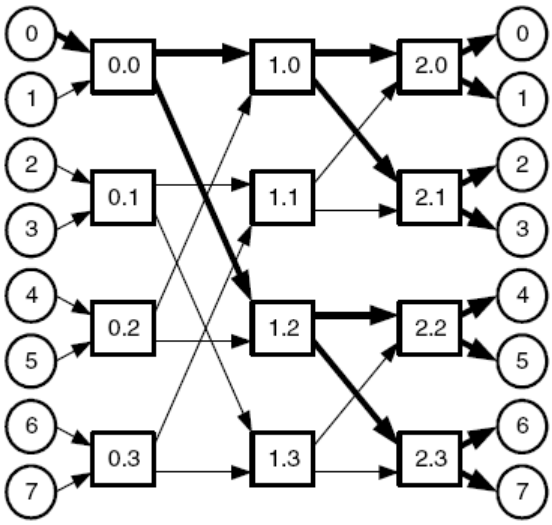


Figura 7: Ejemplo de red multietapa con topología butterfly

2.4 Redes de interconexión basadas en bus

Un bus se define como un medio de interconexión de componentes que consiste en un único canal físico compartido. Luego un bus es un medio de difusión o *broadcast*. En general, un mensaje difundido por un bus tiene como objetivo un componente en particular, y es ignorado por los demás componentes.

En un cierto instante solamente un dispositivo puede hacer uso del bus (es decir, cargar las líneas con voltaje), y por lo tanto el número de dispositivos que comparten el bus afecta el rendimiento global del sistema. Un protocolo de bus determina que componente tiene permiso para usar el bus en un determinado instante y define la forma en que se intercambian los mensajes.

Dado que solo un componente puede enviar mensajes en un determinado instante, los mensajes son *serializados*, es decir, ocurren en un orden fijo y determinístico. Por otro lado, dado

que los mensajes son *broadcast* (es decir, son difundidos por toda la red), se facilita la distribución global de información por un bus. Las dos propiedades anteriores son explotadas por los protocolos de *coherencia* de *memoria cache* o S*nooping Cache-Coherence*. *Snooping* es el proceso donde controladores de *cache* monitorean el bus en busca de mensajes *broadcast* que accedan a direcciones de memoria que estén almacenadas en *cache*. Luego, un controlador de *cache* puede invalidar o actualizar su copia local para mantener la consistencia entre memoria *cache* y memoria del sistema. Estos protocolos se hacen considerablemente más complejos en redes de interconexión genéricas donde existen restricciones para los mensajes *broadcast*, y la serialización requiere un protocolo de sincronización especial.

2.4.1 Definiciones y conceptos básicos

En la figura 8 se muestra un SoC con una arquitectura de comunicación basada en bus.

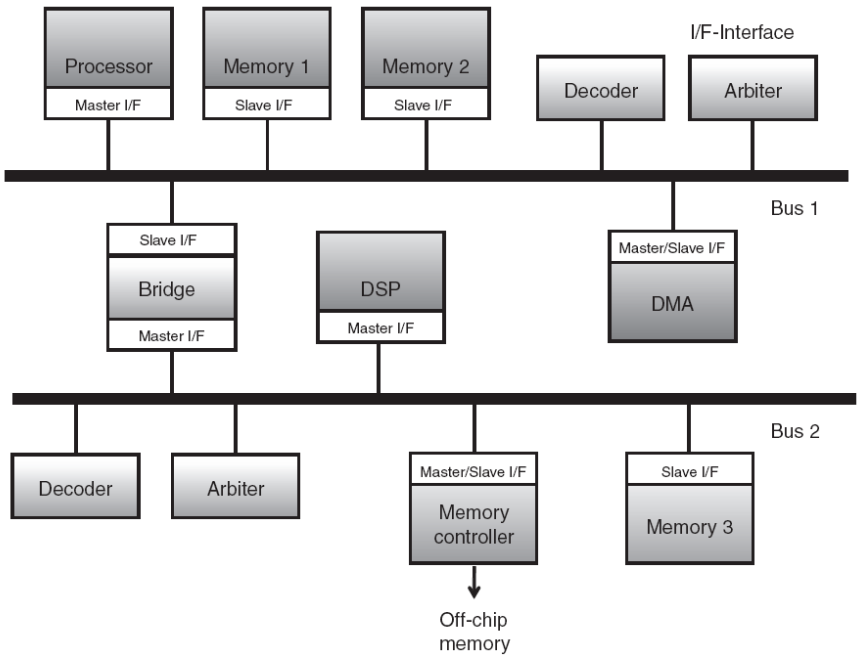


Figura 8: Ejemplo de SoC con arquitectura de comunicación basada en bus

A continuación se definen los elementos básicos que componen una arquitectura basada en bus:

- *Maestro*: Es un dispositivo capaz de iniciar una transacción de lectura o escritura en el bus. En la figura el Procesador y el *DSP* (*Digital Signal Processor*) son ejemplos de maestros que pueden leer o escribir datos a los demás componentes del sistema. El conjunto de señales que usa un maestro para conectarse al bus se llama interfaz o *puerto maestro*.
- *Esclavo*: Es un componente que solo puede responder a transacciones de lectura o

escritura iniciadas por un maestro. Se conecta al bus por medio de un *puerto esclavo*. Los bloques de memoria 1, 2 y 3 de la figura 8 son ejemplos de esclavos.

- *Interfaz de bus*: Una interfaz de bus o puerto es el conjunto de señales que usa un núcleo de hardware para conectarse al bus. Una interfaz de bus puede incorporar buffers triestado, lógica de adaptación de frecuencia, etc., para mejorar el rendimiento de la comunicación.
- *Bridge*: Es un componente usado para comunicar dos buses, que pueden tener protocolos y señales de clock diferentes. En el ejemplo de la figura, si el Procesador (residente en el bus 1) inicia una lectura al bloque de memoria 3 (residente en el bus 2), entonces el *bridge* es el esclavo inicial de la transacción. El *bridge* mediante su lógica interna traduce la transacción de lectura iniciada en el bus 1 a una transacción en el bus 2, y la inicia por medio de su interfaz maestro. Cuando la memoria responde a la transacción iniciada por la interfaz maestro del *bridge*, éste responde al procesador por medio de su interfaz esclavo con los datos solicitados. El *bridge* del ejemplo es unidireccional, es decir, sólo permite pasar transacciones iniciadas en el bus 1 al bus 2. Si por ejemplo el *DSP* deseara leer o escribir datos desde o hacia los esclavos residentes en el bus 1 entonces sería necesario un *bridge* adicional con interfaz esclavo en el bus 2 e interfaz maestro en el bus 1.
- *Árbitro*: Es un componente que resuelve conflictos cuando 2 o más maestros quieren usar el bus simultáneamente. Cuando un maestro quiere usar el bus levanta una solicitud al árbitro, y éste concede el control del bus en base a criterios de prioridad o equidad.
- *Decodificador*: Es la lógica que decodifica la dirección de destino de una transferencia de datos iniciada por un maestro, y selecciona el esclavo apropiado para recibir los datos. Puede ser un componente lógico separado o puede estar embebido en la interfaz de un componente.

Algunos *componentes híbridos* pueden tener una o varias interfaces maestro y una o varias interfaces esclavo. Por ejemplo, un componente *DMA* (*Direct Memory Access*) en general tiene un puerto esclavo destinado a configuración, y dos o más interfaces maestro para leer y escribir desde y hacia diferentes componentes del sistema.

En terminología de buses, cuando una señal lógica cambia de *estado inactivo* (0 en lógica positiva, o 1 en lógica negativa) a *estado activo* (1 en lógica positiva, o 0 en lógica negativa), se dice que la señal es activada o *asserted*. Si la transición es inversa se dice que la señal es desactivada o *deasserted*.

Un bus opera en unidades de *ciclos*, *mensajes*, y *transacciones*. En general, un mensaje en una red de interconexión se define como una unidad lógica de información transmitida desde un transmisor a un conjunto de receptores. Por ejemplo, un procesador que necesita leer una palabra de memoria envía a ésta un mensaje conteniendo la dirección e información de control. Una transacción se define como una secuencia de mensajes relacionados en forma causal. Por ejemplo, una transacción de lectura a memoria es la secuencia de mensajes que establece el protocolo de bus para transferir una palabra entre la memoria y el procesador.

Un bus puede ser *secuenciado internamente* o *externamente*. En un bus secuenciado

internamente cada componente genera sus propias señales de habilitación o *enable* para transmisión y recepción según el protocolo de bus. Por ejemplo, un procesador que genera su propia señal *output enable* cuando gana el control del bus. En el caso de buses secuenciados externamente las señales *enable* para transmisión y recepción son generadas por un secuenciador central externo.

Un bus puede ser *sincrónico* si las transferencias de datos están controladas por las transiciones (o flancos de subida) de una señal de "reloj" o *clock* de bus. La señal de *clock* actúa como referencia de sincronización para todas las señales del bus.

Un bus es *asíncrono* si las transferencias de datos en el bus se basan en la disponibilidad de los datos y no en una señal de *clock*. La transferencia de datos en un bus asíncrono se realiza por medio de un mecanismo llamado *handshaking* o protocolo de negociación.

La siguiente es una secuencia típica de eventos que ocurren cuando un dispositivo maestro tal como el Procesador o DSP desean transferir un dato hacia un dispositivo esclavo, como por ejemplo una memoria:

1. Maestro: Envía una solicitud para usar el bus
2. Maestro: La solicitud es aceptada y el control del bus se concede al maestro
3. Maestro: Coloca dirección y datos en el bus
4. Esclavo: Esclavo es seleccionado como objetivo de la transacción
5. Maestro: Señala transferencia de datos
6. Esclavo: Toma los datos
7. Maestro: Libera el bus

2.4.2 Tipos de señales de bus

Un bus es una conexión física para mover señales de un punto a otro. La señal transportada puede representar dirección, datos, control, o potencia. Típicamente, un bus se compone de conexiones operando en conjunto que permiten el movimiento de bits en paralelo. Cada conexión se llama "línea" y normalmente se identifican por un número. Grupos de líneas relacionados usualmente se identifican por un nombre. Dependiendo de la señal transportada, existen al menos cuatro tipos de buses: bus de dirección, datos, control, y potencia (voltajes de alimentación y tierra de referencia). El tamaño de cada bus (número de líneas) varía entre un sistema y otro.

Las señales o conductores de bus se pueden clasificar en tres grandes grupos: Dirección, Datos, y Control.

- *Dirección*: Es el conjunto de señales (llamado *bus de dirección*) que se usa para transmitir la dirección del origen o destino de los datos a transferir. El número de señales para transmitir la dirección o *ancho de dirección* es típicamente potencia de 2 (16, 32, o 64), aunque en algunos casos puede ser arbitrario, dependiendo del número de componentes en el sistema. El bus de dirección puede ser no compartido para lectura y escritura, lo que permite mayor cantidad de transacciones ocurriendo en paralelo, pero con un mayor costo en área de chip y consumo de potencia.
- *Datos*: Es el conjunto de señales (llamado *bus de datos*) usado para transmitir los datos a

la dirección destino. Típicamente el *ancho de datos* es potencia de 2 (16, 32, 64, 128, 256, 512 y 1024), aunque puede variar dependiendo de los requerimientos del sistema. Los anchos de datos de maestro y esclavo no siempre coinciden, y en estos casos se requiere de lógica para empaquetar y desempaquetar datos. Por ejemplo, si un procesador de 32-bit (ancho de datos) necesita leer de una memoria con tamaño de palabra de 64-bit, entonces se necesita una lógica que desempaquete o divida cada palabra de 64-bit en 2 de 32-bit antes de enviarlas al procesador. Por otro lado, en la interfaz del procesador se necesitan empaquetar las 2 palabras recibidas para formar la palabra original. El empaquetar y desempaquetar datos determina un *overhead* en términos de rendimiento y consumo de potencia, que se evita cuando los anchos de datos son iguales entre maestro y esclavo (lo que no siempre es posible). Los buses de datos también pueden ser implementados como buses separados para lectura y escritura, mejorando el rendimiento del sistema a un costo de mayor área y consumo de potencia. También es posible multiplexar dirección y datos sobre el mismo conjunto de cables, lo que determina menos cableado y cantidad de pines en las interfaces.

- *Control*: Las señales de control son específicas al protocolo de bus, y se usan para enviar información sobre la transacción. Algunos ejemplos de señales típicas son *Request* y *Acknowledge*, que indican la solicitud de transferencia de datos por parte de un maestro, y la posterior respuesta del esclavo a esa solicitud respectivamente. En forma adicional a las señales de control, un bus de control puede contener señales de *timing* o "sincronización". Estas señales se usan para señalizar el instante exacto de una transferencia de datos desde y hacia el bus, es decir, señalan cuando un determinado dispositivo tal como el procesador, memoria, o dispositivos I/O, pueden poner datos en el bus y cuando pueden recibir datos desde el bus.

2.4.3 Estructura física

Físicamente un bus puede ser un único conductor, un *bus serial*, o un *bus paralelo*. En un bus serial el número de conductores paralelos es menor que el largo del mensaje, es decir, se requiere una cierta cantidad de ciclos para transmitir un mensaje por el bus. En un bus paralelo el número de conductores es tal que permite enviar el mensaje en un ciclo. Un esquema intermedio es aquel donde un mensaje es secuenciado sobre un conjunto más pequeño de conductores paralelos, tomando una cantidad de ciclos menor al caso serial para transmitir el mensaje.

La implementación más común buses ha sido mediante el uso de *buffers triestado* que cargan conductores compartidos bidireccionales como se muestra en la figura 9. Cada módulo se conecta al bus a través de una interfaz bidireccional que permite cargar una señal de *salida* denotada con el sufijo *_o* (*output*), cuando se activa una señal de *habilitación de salida* denotada con el sufijo *_oe* (*output enable*). Análogamente, el módulo toma una muestra de la señal en el bus con un registro interno cuando se activa la señal de *habilitación de entrada _ie* (*input enable*). Este esquema requiere menos conductores y por lo tanto ocupa menos área de PCB. Tiene la desventaja de presentar mayor consumo de potencia, y mayor retardo. Además, debido a la discontinuidad de impedancia que presenta cada interconexión de un módulo al bus (ramificación de conductores o *stubs,* y *vias*), se limita la operación a alta velocidad. El uso de buffers triestado es la elección preferida en buses *off-chip* o *backplane*.

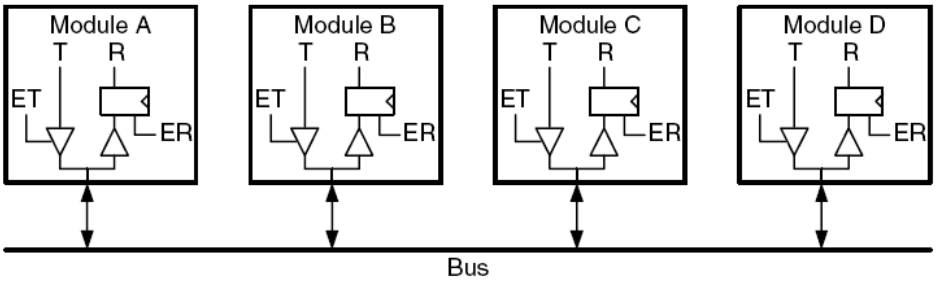


Figura 9: Implementación de bus usando buffers triestado

Alternativas al uso de buffers triestado son la implementación basada en multiplexores (figura 10) e implementación basada en estructuras AND-OR (figura 11). Estos dos esquemas son preferidos en *SoCs*.

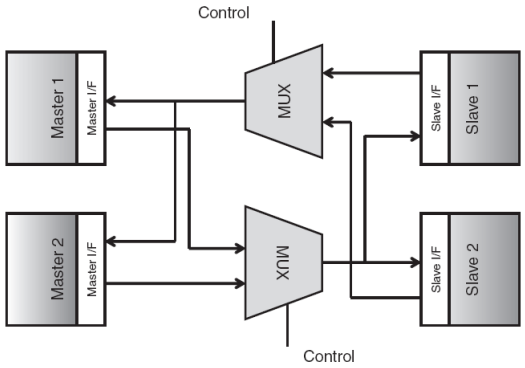


Figura 10: Ejemplo de implementación de bus basada en multiplexores

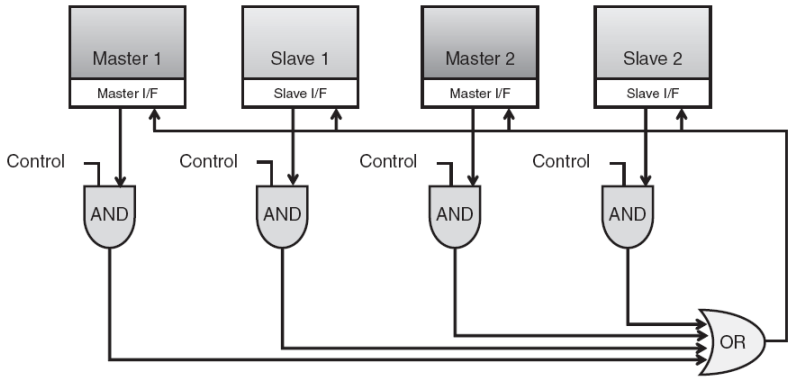


Figura 11: Ejemplo de implementación de bus basada en estructura AND-OR

2.4.4 Buses sincrónicos y asíncronos

En buses sincrónicos las transferencias de datos se ocurren sólo en transiciones de la señal de *clock*. Todos los eventos del bus están sincronizados al *clock*, que es una señal cuadrada y debe estar disponible tanto para dispositivos maestros como esclavos. Un *ciclo de bus* comienza con un flanco positivo del clock y termina con el siguiente flanco positivo, que es el inicio del siguiente ciclo.

En un ciclo de escritura típico, en el primer ciclo del *clock* el maestro carga el bus con dirección, datos y activa las señales de control según el protocolo de bus. El esclavo decodifica la dirección que presenta el maestro y si la acepta, toma el dato en el siguiente ciclo del *clock*.

Cuando se conectan dispositivos con distintas velocidades en un bus sincrónico, el dispositivo más lento determina la tasa de transferencia del bus. Por otro lado, dado que todas las transiciones están controladas por la misma señal de *clock*, cualquier *skew* o "deslizamiento" de la señal de *clock* vista por distintos dispositivos puede generar problemas. Luego, la longitud de un bus sincrónico está limitada por el máximo *clock skew* admisible[2].

Adicionalmente, el ciclo de bus (recíproco de la frecuencia del *clock*) no debe ser menor al tiempo de propagación de las señales por el bus. Por ejemplo, en un bus con señal de *clock* a una frecuencia $f = 100$ MHz, las señales tienen una ventana de tiempo máxima para viajar de origen a destino igual a $1/f = 10$ ns.

En un bus asíncrono o "sincrónico a fuente" (*source synchronous timing*), no se usa una señal de *clock* y la transferencia de información está a cargo de señales de control por medio de un protocolo de negociación o *handshaking*. En general, la señal que cumple la función de *clock* se llama *strobe* y es activada por la fuente para validar la información en el bus.

A continuación se muestra una secuencia de *handshaking* de tipo *fully-interlocked* para un ciclo de escritura asíncrono:

1. Maestro: pone dirección, datos y activa la señal de control "dato-listo"
2. Esclavo: cuando está listo, recibe el dato y activa la señal de control "dato-aceptado"
3. Maestro: cuando detecta el flanco positivo de la señal "dato-aceptado", desactiva la señal "dato-listo"
4. Esclavo: cuando detecta el flanco negativo de la señal "dato-listo", desactiva la señal "dato-aceptado"

El término *fully-interlocked* se refiere a que cada etapa del *handshaking* puede continuar solamente cuando la etapa previa ha sido respondida. Cada acción (activación o negación de una señal) toma lugar después de una estricta secuencia que termina cuando todas las señales son negadas. Esta clase de transferencia de datos también se llama en *lazo cerrado*, y como además se responden negaciones de señales, se dice que es *delay insensitive* (es decir, la correcta operación no depende de los retardos de las líneas del bus).

2 Por ejemplo, en el bus PCI el máximo *clock skew* admisible es de 2 ns, para un clock de 33 Mhz (periodo de 30 ns)

2.4.5 Análisis temporal de buses sincrónicos

A continuación se derivan las ecuaciones de *sincronización temporal* o *timing* necesarias para analizar arquitecturas basadas en buses sincrónicos de *clock* común. Estas ecuaciones permiten analizar los factores de *timing* que afectan el rendimiento, establecer objetivos de diseño, calcular velocidades máximas de un bus, y calcular márgenes para el *timing*.

Sea el sistema de la figura 12, compuesto por un maestro (A) y un esclavo (B) que se comunican por medio de un bus sincrónico único. Para el análisis se considera una transacción donde el maestro envía un bit desde un FF-D_A (*flip-flop* D_A), hacia un FF-D_B en el esclavo.

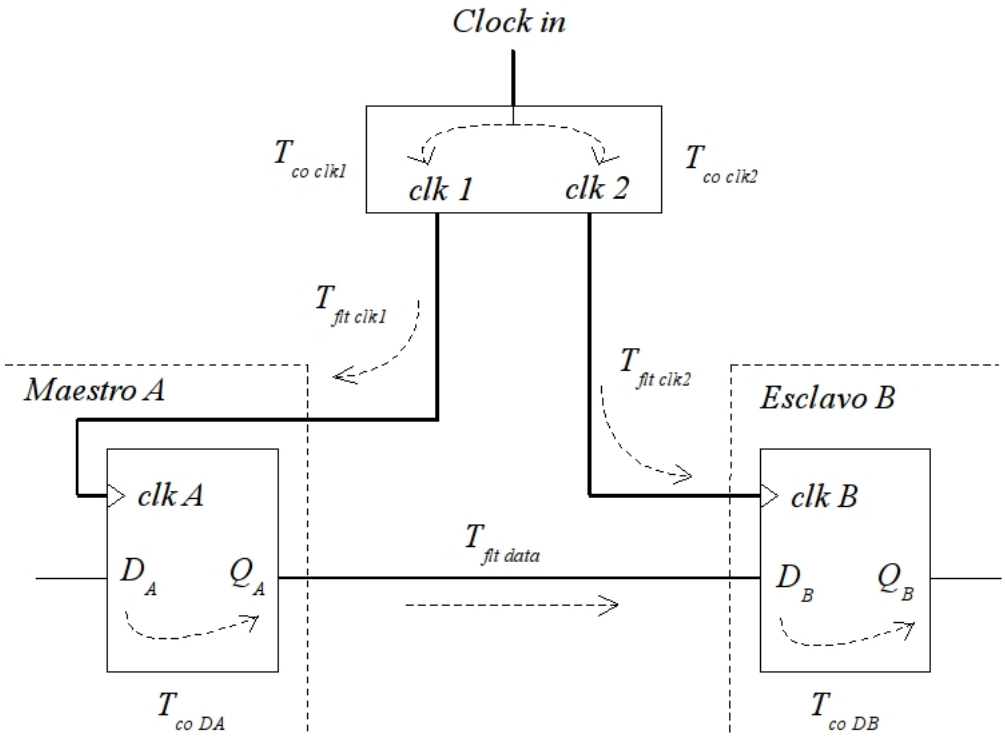


Figura 12: Esquema de bus sincrónico único con parámetros de timing

A continuación se definen los tres tipos de retardo principales asociados a este sistema:

- T_{co} (*clock to out*): El tiempo que tarda un elemento de lógica secuencial en cambiar de estado en respuesta al flanco positivo de una señal de *clock*.
- T_{flt} (*flight time*): Tiempo que tarda una señal en propagarse por una línea de transmisión.
- T_{jitter} (*clock jitter*): Variación aleatoria ciclo a ciclo del periodo del clock.

Inicialmente el maestro tiene un dato disponible a la entrada de FF-D_A. Cuando se produce una primera transición en el clock del sistema (*Clock in*), el flanco positivo se transmite a través del buffer de clock ($T_{co\ clk1}$), se propaga por la línea de transmisión hacia la entrada de clock de

FF-D_A ($T_{flt\ clk1}$), y el dato se registra desde D_A a Q_A ($T_{co\ DA}$), es decir, el dato se coloca en el bus. Esta señal en Q_A se propaga por la línea de transmisión hacia el esclavo ($T_{flt\ data}$), y finalmente el dato se registra desde D_B a Q_B cuando llega un segundo flanco positivo a la entrada de clock de FF-D_B ($T_{co\ clkB} + T_{flt\ clkB}$). Luego, para realizar la transferencia se requieren dos transiciones del clock: un primer flanco positivo para registrar el dato interno del maestro hacia el bus (a través de un buffer de salida), y un segundo para que el esclavo registre internamente el dato presente en el bus.

A partir de lo anterior se deduce que el retardo de la circuitería y las líneas de transmisión debe ser menor al ciclo de bus o periodo del clock. Luego, existe una cota superior teórica a la frecuencia de operación del bus dada por los retardos del sistema.

En la figura 13 se muestra el diagrama de *timing* asociado al sistema, que permite derivar las ecuaciones sincronización. Las flechas los indican retardos, y se pueden agrupar en dos grupos o *bucles de sincronización* (*timing loops*): *bucle de establecimiento* o *setup loop* (verde), y *bucle de mantenimiento* o *hold loop* (azul).

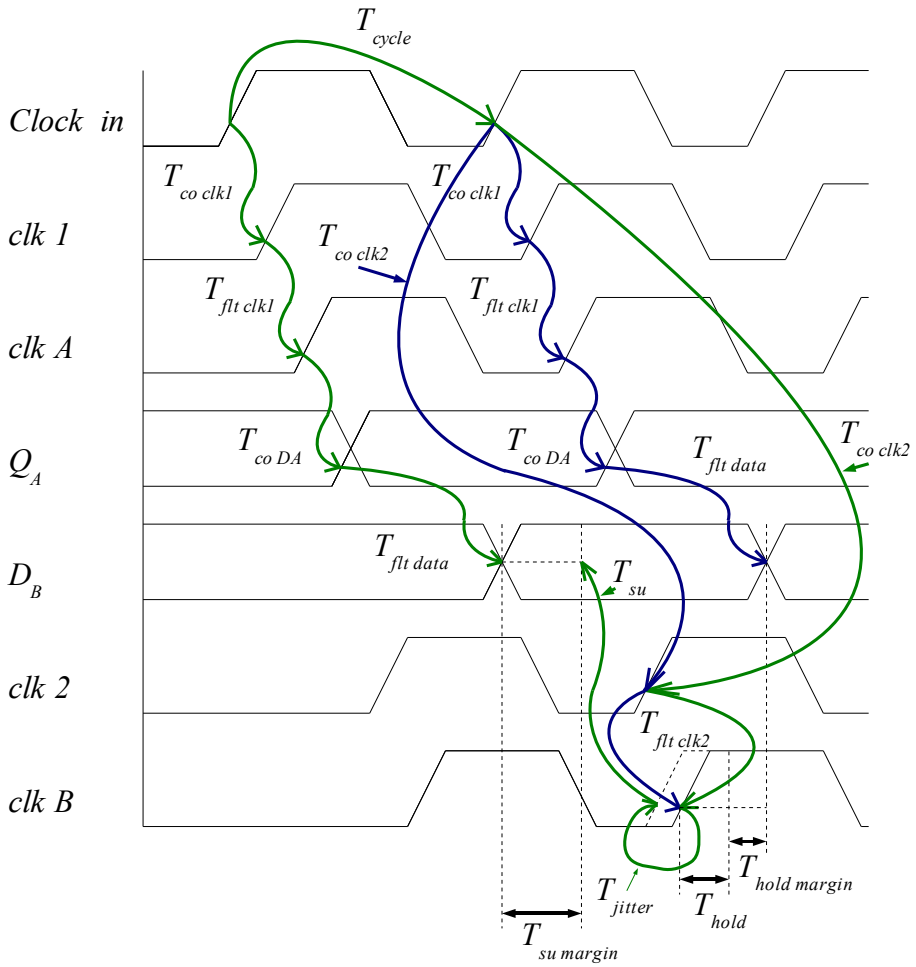


Figura 13: Diagrama de timing para bus sincrónico único

Se definen los requerimientos de *setup* (T_{su}) y *hold* (T_{hold}) de un receptor (en este caso FF-D_B) como los tiempos mínimos en que los datos deben mantenerse válidos antes y después de un flanco positivo del clock respectivamente, para asegurar el correcto registro o *latching* de los datos. Existen ciertos márgenes para T_{su} y T_{hold} que si son violados el *flip-flop* entra en un estado conocido como *estado metaestable* o *quasi-estable*, donde su salida es impredecible. A continuación se derivan las ecuaciones para los márgenes de T_{su} y T_{hold} que aseguran operación sin problemas de metaestabilidad.

2.4.5.1 Margen de establecimiento

Para registrar el dato correctamente en el flip-flop del receptor, éste debe llegar a la entrada y permanecer válido durante al menos T_{su} antes de la ocurrencia del clock. Dado que el receptor registra el dato en el segundo flanco positivo de *clk B*, entonces debe asegurarse que el retardo asociado a la llegada del dato a la entrada del receptor ($T_{dat\ total} = T_{co\ clk1} + T_{flt\ clk1} + T_{co\ DA} + T_{flt\ data}$) sea menor al retardo del clock del receptor ($T_{clk\ total} = T_{cycle} + T_{co\ \ clk2} + T_{flt\ clk2} - T_{jitter}$), ambos referidos al primer clock. Notar que $T_{dat\ total}$ y $T_{clk\ total}$ corresponden a los lados izquierdo y derecho respectivamente del *setup loop* (verde) en la figura 13. La diferencia entre ambos retardos comparada con el requerimiento de *setup* del receptor (T_{su}) corresponde al *margen de establecimiento* ($T_{su\ margin}$):

$$T_{su\ margin} = (T_{clk\ total} - T_{dat\ total}) - T_{su}$$
$$= T_{cycle} + T_{co\ \ clk2} + T_{flt\ clk2} - T_{jitter} - T_{co\ clk1} - T_{flt\ clk1} - T_{co\ DA} - T_{flt\ data} - T_{su}$$

Se define deslizamiento de clock o *clock skew* como la diferencia en tiempo entre dos transiciones de clock simultáneas en un sistema. Luego, el *clock skew* introducido por el buffer de clock está dado por:

$$T_{clk\ skew} = T_{co\ clk1} - T_{co\ clk2}$$

El *clock skew* introducido por el tiempo de vuelo en las líneas de transmisión está dado por:

$$T_{flt\ skew} = T_{flt\ clk1} - T_{flt\ clk2}$$

Reemplazando las expresiones de *clock skew* anteriores resulta la siguiente expresión para el *margen de setup*:

$$T_{su\ margin} = T_{cycle} - T_{flt\ skew} - T_{clk\ skew} - T_{jitter} - T_{co\ DA} - T_{flt\ data} - T_{su}$$

Si $T_{su\ \ margin}$ es mayor o igual a cero, no se violan los requerimientos de *setup* y el sistema

funciona correctamente. En caso contrario se obtiene comportamiento metaestable.

2.4.5.2 Margen de mantenimiento

Para que el receptor pueda registrar correctamente el dato en FF-D_B, éste debe permanecer válido a la entrada durante al menos T_{hold} a partir de la llegada del segundo flanco positivo a *clk B*. Además el segundo flanco positivo inicia la siguiente transferencia de datos. Por lo tanto, el receptor debe registrar el dato antes de que éste cambie por uno nuevo, es decir, el retardo del segundo flanco positivo del clock ($T_{clk\ delay} = T_{co\ \ clk2} + T_{flt\ clk2}$) sumado a T_{hold}, debe ser menor al tiempo que toma el siguiente dato en llegar al receptor ($T_{dat\ delay} = T_{co\ clk1} + T_{flt\ clk1} + T_{co\ DA} + T_{flt\ data}$).

$T_{dat\ delay}$ y $T_{clk\ delay}$ corresponden a los lados derecho e izquierdo respectivamente del *hold loop* (azul) en la figura 13. La diferencia entre ambos retardos comparada con el requerimiento de *hold* (T_{hold}) del receptor, corresponde al *margen de mantenimiento* ($T_{hold\ margin}$):

$$T_{hold\ margin} = (T_{dat\ delay} - T_{clk\ delay}) - T_{hold}$$
$$= T_{co\ clk1} + T_{flt\ clk1} + T_{co\ DA} + T_{flt\ data} - T_{co\ clk2} - T_{flt\ clk2} - T_{hold}$$

La ecuación anterior en función de $T_{clk\ skew}$ y $T_{flt\ skew}$ resulta:

$$T_{hold\ margin} = T_{co\ DA} + T_{flt\ data} + T_{clk\ skew} + T_{flt\ skew} + T_{cycle} - T_{hold}$$

Notar que la ecuación anterior no depende del periodo del clock (T_{cycle}), ni de la variación aleatoria ciclo a ciclo del periodo del clock o *jitter* (T_{jitter}), es decir, el margen de *hold* es independiente de la frecuencia del clock. Sin embargo, el requerimiento de *setup* impone un límite máximo teórico a la frecuencia del clock.

2.4.6 Decodificación

Para iniciar una transferencia de datos en un bus, la fuente requiere enviar la dirección del componente destino, que típicamente está asignada en un rango de direcciones predefinido o *mapa de direcciones* (también llamado *mapa de memoria*). El propósito de un mapa de memoria es definir claramente el rango que ocupa cada componente en el espacio de direcciones. Un procesador típicamente usa múltiples mapas de memoria, como por ejemplo puede tener un mapa definido de fábrica para datos internos, un mapa para memoria de programa, y un tercero para memoria de datos. Se requiere de lógica que decodifique las direcciones en base al mapa de memoria apropiado y seleccione el destino para la transferencia de datos. La lógica de decodificación puede ser implementada en forma centralizada (*decodificador* central) o en forma distribuida.

En la figura 14 se muestra un ejemplo de decodificación centralizada, donde el decodificador toma como entrada la dirección de una transacción iniciada por un maestro y luego activa una señal de selección (por ejemplo *chipselect*) para el esclavo apropiado, indicando que se

necesitan leer o escribir datos desde o hacia el esclavo en particular. Dado que la lógica asociada al mapa de direcciones se encuentra centralizada, la adición de componentes al sistema requiere mínimos cambios, y por lo tanto este esquema es fácilmente extensible.

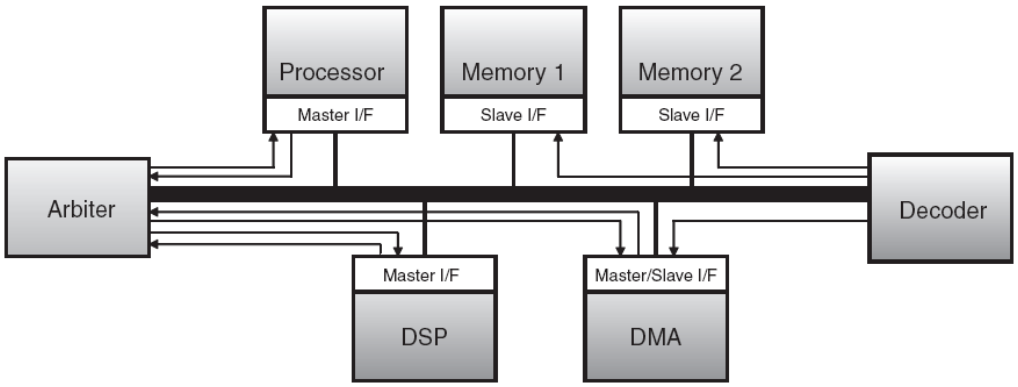


Figura 14: Ejemplo de esquema de arbitración y decodificación centralizada

En la figura 15 se muestra un ejemplo de decodificación distribuida, donde cada esclavo tiene su propia lógica de decodificación por separado. Cuando un maestro envía una dirección, cada esclavo decodifica la dirección para determinar si es el objetivo de la transacción en curso. Este esquema requiere menos conductores que el caso centralizado, sin embargo tiene la desventaja de presentar duplicación de lógica porque cada esclavo decodifica en forma independiente, y por lo tanto un cambio en el mapa de direcciones puede requerir cambiar la lógica de decodificación de todos los esclavos.

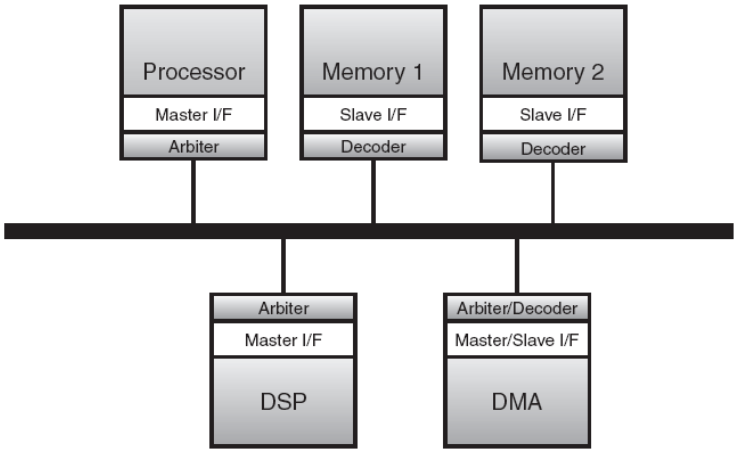


Figura 15: Ejemplo de esquema de decodificación y arbitración distribuida

2.4.7 Arbitración

Solamente un dispositivo maestro puede hacer uso del bus a la vez, y por lo tanto se necesita un sistema basado en permisos que resuelva conflictos cuando dos o más maestros quieren tomar el control del bus al mismo tiempo. La arbitración es el proceso de seleccionar el próximo maestro entre múltiples candidatos, utilizando criterios de equidad o prioridad, y en un esquema centralizado o distribuido.

2.4.7.1 Arbitración centralizada

En esquemas de arbitración centralizada, se usa un único árbitro para seleccionar al próximo maestro. Típicamente se usan las señales de arbitración *request* (solicitud), *grant* (conceder) y *busy* (ocupado). Estas señales pueden ser compartidas por los potenciales maestros (esquema *daisy-chain*) o pueden ser independientes para cada maestro.

En la figura 16 se muestra un ejemplo de secuencia arbitración entre 2 maestros en un bus sincrónico. El maestro 1 solicita el bus en el ciclo 1, obtiene el bus en el ciclo 2, realiza una transacción en los ciclos 3 y 4, y libera el bus en el ciclo 5 desactivando su señal *Req1*. El maestro 2 solicita el bus en el ciclo 2, pero debe esperar hasta el ciclo 6 para obtener el bus y poder realizar su transacción. Notar que el bus permanece en estado inactivo o *idle* por 2 ciclos entre el fin de la primera transacción y el comienzo de la segunda, a causa de la arbitración. En la sección 2.4.8.2 se muestra la técnica de *pipelining* para eliminar los ciclos en que el bus está inactivo.

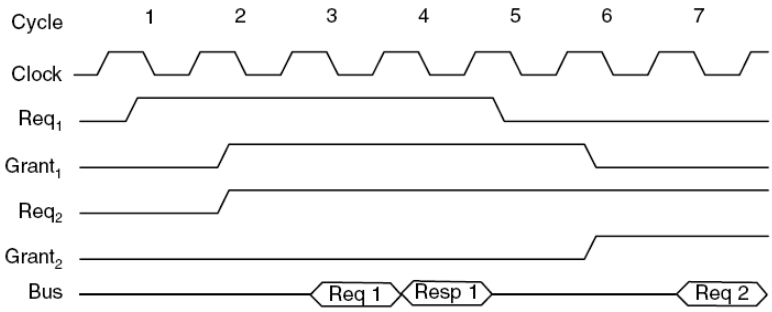


Figura 16: Ejemplo de secuencia de arbitración en un bus sincrónico

En la figura 17 se muestra un esquema de arbitración *daisy-chain* o "en cadena", donde una señal de arbitración *grant* se pasa de maestro en maestro, en orden de prioridad descendente (el maestro con mayor prioridad es el más cercano al árbitro).9 La equidad no se asegura, y es posible que el maestro con menor prioridad nunca obtenga permiso. Cuando una solicitud es recibida por el árbitro, éste concede un permiso activando la señal *grant*. Cuando el maestro potencial más cercano al árbitro ve la señal *grant*, chequea si él fue quién hizo la solicitud. Si él hizo la solicitud, toma el control del bus y detiene la propagación de la señal *grant* al resto de la cadena de maestros. En caso contrario, simplemente propaga la señal *grant* al maestro potencial adyacente en la cadena, y así sucesivamente. Cuando se completa la transacción se desactiva la señal *busy*.

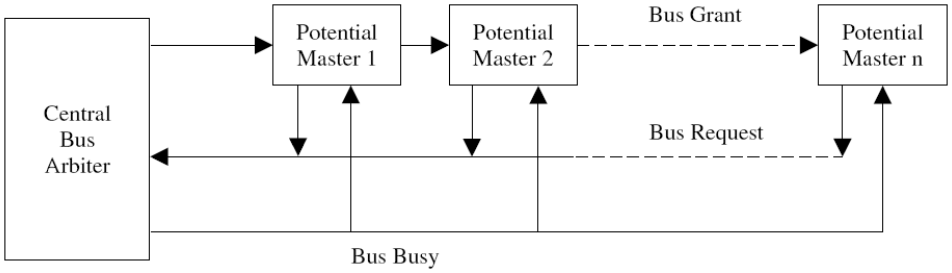


Figura 17: Esquema de arbitración centralizada en cadena

Otro esquema centralizado se muestra en la figura 18, donde cada maestro tiene líneas independientes conectada al árbitro central, que puede estar basado en criterio de equidad o prioridad. Este esquema es llamado *arbitración radial*.

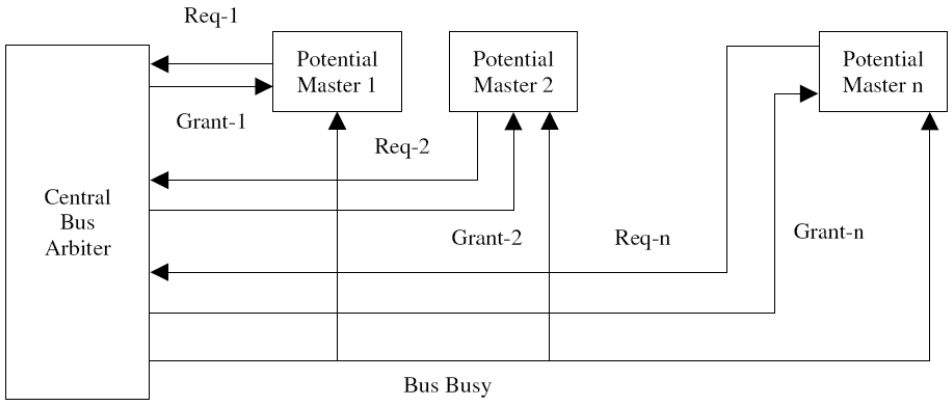


Figura 18: Esquema de arbitración centralizado con líneas grant y request independientes

También es posible un esquema donde los maestros tengan múltiples niveles de prioridad, donde por cada nivel se tiene una cadena de maestros controlada por un par *request/grant* como se muestra en la figura 19. Cuando el árbitro recibe múltiples solicitudes desde distintos niveles, otorga el bus a un nivel bajo criterio de prioridad.

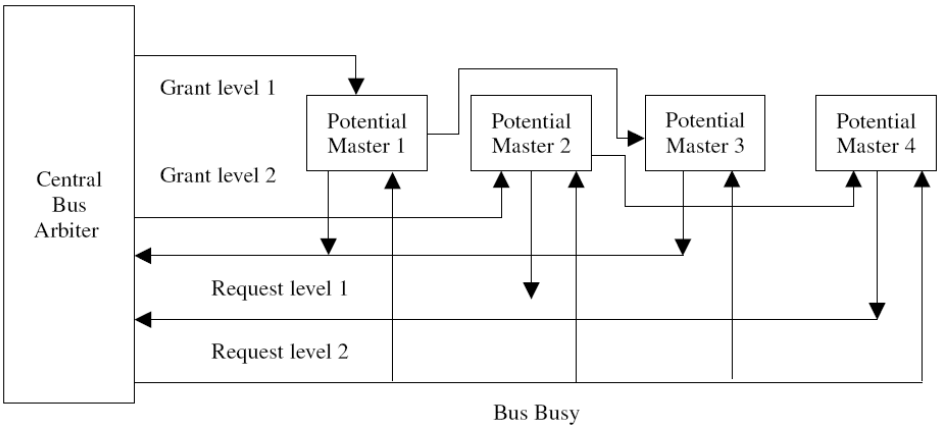


Figura 19: Esquema de arbitración centralizada con dos niveles de prioridad en cadena

2.4.7.2 Arbitración descentralizada

En esquemas de arbitración descentralizada se usa criterio de prioridad en forma distribuida. Cada maestro potencial tiene un único número de arbitración que es usado al resolver conflictos. Por ejemplo, sean tres maestros con prioridades 9, 10 y 7 (1001, 1010 y 0111) compitiendo por el control del bus. En el primer ciclo se operan las tres prioridades con un OR lógico para obtener un número de arbitración igual a 15 (1111). Comparando el bit más significativo, el maestro con prioridad 7 queda fuera de competición. Con los dos maestros restantes se obtiene un nuevo número de arbitración igual a 1011 aplicando un OR a sus prioridades. Ninguno de los maestros queda fuera de competición cuando son comparados contra el bit 2. El maestro con prioridad 9 queda fuera cuando la comparación se hace con el bit 1, y finalmente el maestro con prioridad 10 obtiene el control del bus.

Las prioridades se pueden asignar en forma arbitraria con la condición de que sean únicas. Este esquema tiene la ventaja que es posible implementarlo con las líneas de bus existentes y no necesita de un árbitro central. Tiene la desventaja de ser lento (completar el algoritmo puede tomar del orden de nanosegundos versus un árbitro central que toma del orden de cientos de picosegundos), y además usa las líneas del bus durante el proceso de arbitración, lo que impide hacer transacciones mientras se resuelve un conflicto.

2.4.8 Modos de transferencia de datos

En general, un protocolo de bus permite al menos un modo o manera de transferir datos entre componentes. A continuación se describen los principales modos de transferencia en arquitecturas basadas en bus.

2.4.8.1 Transferencia única sin pipeline

En la figura 20 se muestra un ejemplo donde un maestro realiza dos transacciones únicas (de escritura o lectura) sin pipeline, en forma secuencial. El maestro solicita el bus al árbitro en el

ciclo 1, obtiene el bus en el ciclo 2, envía la dirección en el ciclo 3, y envía los datos (caso de escritura) en el ciclo 4 o espera a que el esclavo envíe datos en los ciclos posteriores (caso de lectura). Los datos son registrados en el flanco de subida del ciclo 5, por el maestro en el caso de lectura o por el esclavo en el caso de escritura. Notar que en el caso de lectura el esclavo responde a las solicitudes del maestro sin *estados de espera*. Un estado de espera es un ciclo de bus inactivo o *idle* que introduce el esclavo activando una señal de solicitud de espera (por ejemplo *waitrequest*) cuando no es capaz de responder a la solicitud del maestro en forma inmediata. La secuencia se repite en los ciclos 5, 6, 7 y 8, y termina cuando se registran los datos en el flanco de subida del ciclo 9.

Al contrario del ejemplo, típicamente los esclavos introducen estados de espera tanto para transacciones de lectura como de escritura. Además, el árbitro puede tomar múltiples ciclos resolver un conflicto y otorgar el bus al maestro, especialmente en el caso donde el esquema de arbitración es complejo.

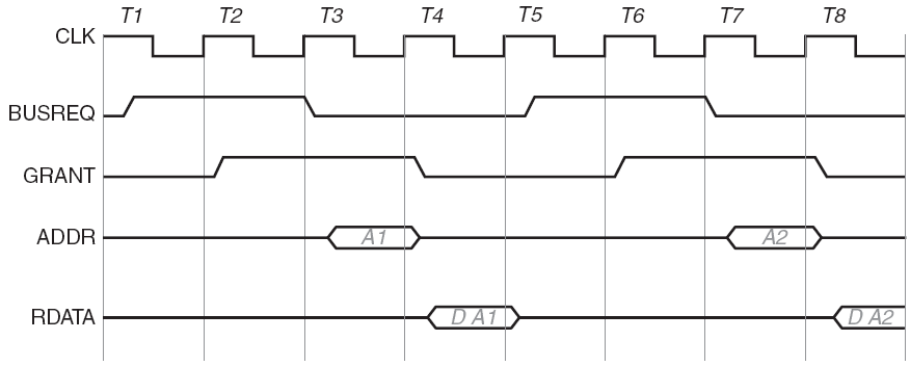


Figura 20: Ejemplo de transacciones únicas en forma secuencias

2.4.8.2 Transferencia pipeline

El tiempo que toma cada arbitración produce una significante pérdida de rendimiento en aplicaciones donde se realiza arbitración por casi toda transacción en el bus, como por ejemplo en memoria compartida por multiprocesadores. El rendimiento en estos casos se puede mejorar mediante *pipelining* de las fases de una transacción de bus. *Pipelining* se refiere a traslapar operaciones poniendo señales o datos en un conducto o *pipe* conceptual, donde las operaciones o eventos del proceso se realizan en forma simultánea o paralela. Para representar un *pipeline* se usan *tablas de reserva*, que son una forma de representar el patrón de flujo de eventos en un sistema con *pipelining*. Cada fila de la tabla de reserva representa un recurso del *pipeline* y cada columna representa una unidad temporal o ciclo. Cada elemento de la tabla puede tener solamente dos propiedades: estar usado o no.

En la figura 21 se muestran las tablas de reserva para una transacción de escritura a memoria (a) y para una transacción de lectura a memoria con un ciclo de latencia (b). Cada tabla muestra los ciclos que se necesitan para completar una transacción (columnas) y los recursos involucrados (filas). El ciclo de lectura empieza con tres ciclos de arbitración: un ciclo AR donde se activa una solicitud de bus, seguido por un ciclo ARB donde el árbitro toma una decisión, y un

ciclo AG donde la señal grant del árbitro se activa en retorno al solicitante. Luego de la arbitración, la transacción de lectura consiste en una solicitud RQ en donde se envía la dirección al dispositivo de memoria, un ciclo de procesamiento P donde se realiza el acceso a memoria, y un ciclo de respuesta RPLY donde los datos se retornan al solicitante.

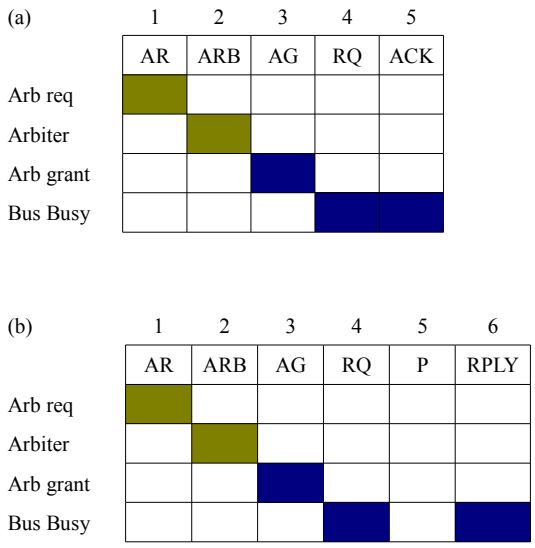


Figura 21: Tablas de reserva para transacciones de escritura (a) y lectura (b)

Las tablas de reserva muestran como se usan los recursos durante cada ciclo. Las casillas de color verde muestran las transacciones que pueden ser compartidas en un ciclo dado. Las casillas de color azul muestran los recursos que se usan exclusivamente durante la transacción durante un ciclo dado. Solo una transacción puede recibir un grant durante un ciclo dado y solo una transacción puede usar el bus durante un ciclo dado.

En un bus con estados de espera fijos se puede iniciar una transacción en cualquier ciclo de la tabla de reserva mientras no se solicite ningún recurso exclusivo que haya sido reservado por alguna transacción anterior.

En la tabla 1 se muestra la secuencia temporal para completar 6 transacciones en un *pipelined bus* (que toma 17 ciclos de bus). En un bus sin *pipeline* (donde la arbitración no puede comenzar hasta que la transacción anterior termina) la misma secuencia toma 34 ciclos para completar dos escrituras de 5 ciclos y cuatro lecturas de 6 ciclos sin traslape.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Read 1	AR	ARB	AG	RQ	P	RPLY											
Write 2		AR	ARB	AG	Stall	Stall	RQ	ACK									
Write 3			AR	ARB	Stall	Stall	AG	Stall	RQ	ACK							
Read 4				AR	Stall	Stall	ARB	Stall	AG	Stall	RQ	P	RPLY				
Read 5							AR	Stall	ARB	Stall	AG	RQ	P	RPLY			
Read 6									AR	Stall	ARB	AG	Stall	Stall	RQ	P	RPLY
Bus Busy																	

Tabla 1: Ejecución de cuatro lecturas y dos escrituras en un pipelined bus

Sin embargo, el bus aún está en estado de espera en el estado P de una lectura seguida por una escritura debido a las diferencias entre los *pipeline* de lectura y escritura. Por ejemplo, la transacción *Write 2* no puede tomar el bus hasta el ciclo 7 para evitar la colisión entre su ACK y el RPLY de la transacción *Read 1*. Esta situación empeora en el caso de transacciones con estados de espera variable. Por ejemplo, si una lectura toma un tiempo de espera entre 0 y 20 ciclos P, ninguna otra transacción puede iniciarse hasta que se complete la lectura, manteniendo el bus desocupado por todos los ciclos P que tome.

Las transferencias *pipeline* típicamente requieren la implementación de un esquema de arbitración más complejo capaz de traslapar la arbitración. Además solo es posible implementar *pipelining* en arquitecturas con buses de dirección y datos por separado (es decir, no multiplexados).

2.4.8.3 Transferencia burst

La cantidad de mensajes que involucra una transacción de bus es considerable. Para cada transacción debe realizarse arbitración, direccionamiento y *handshaking* dependiendo del protocolo. El *overhead* o exceso de uso de recursos (ciclos) que son necesarios para completar una tarea (transferir datos), es especialmente alto cuando el dato a transferir es una única palabra. El *overhead* del bus puede reducirse incrementando la cantidad de datos transferidos en cada mensaje. Cuando se transmiten múltiples mensajes, como por ejemplo un bloque de datos a un periférico, es más eficiente enviar un bloque por mensaje que enviar una sola palabra por vez. Además, el modo *burst* requiere ciclos de arbitración por cada bloque de datos y no por cada palabra a transferir.

En la figura 22 se muestra la reducción de overhead que se obtiene al enviar un bloque de 4 datos en modo burst: en (a) se muestra un bus con 2 ciclos de overhead por mensaje, donde se transfiere una palabra de datos durante un mensaje de 3 ciclos (eficiencia de 1/3); en (b) se envían 4 palabras por cada mensaje de 6 ciclos (eficiencia de 2/3). En general, con x ciclos de overhead y n palabras de datos transferidos la eficiencia es $n/(n+x)$.

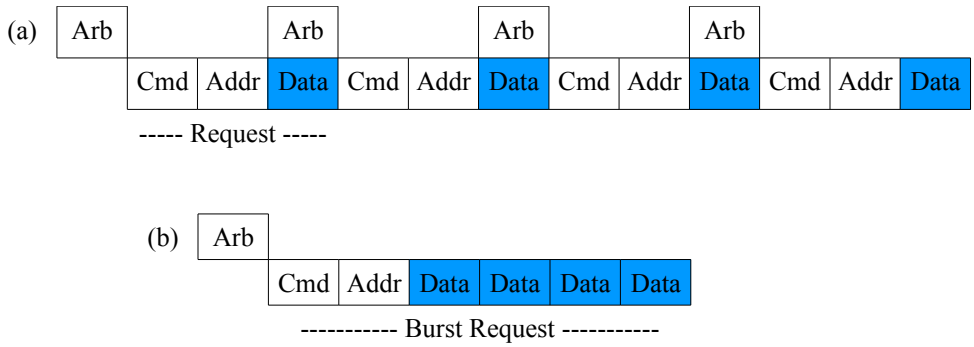


Figura 22: Reducción de overhead usando modo burst

Una desventaja del modo burst es que al incrementar el tamaño del bloque a transferir, se incrementa el tiempo máximo de espera que un maestro de alta prioridad debe esperar para tomar el control del bus. Luego, para permitir mensajes *burst* largos y al mismo tiempo proveer baja latencia de arbitración para maestros de alta prioridad, algunos buses permiten la interrupción de mensajes burst para posteriormente ser reanudados o reinicializados. Cuando se interrumpe un mensaje solo se incurre en *overhead* para la reanudación del mensaje. En esquemas donde se permite abortar mensajes para reinicializarlos afecta el rendimiento debido a la redundancia en el uso del bus.

2.4.8.4 Transferencia no atómica o split

El tiempo de espera entre los mensajes de una transacción puede aprovecharse mediante la partición de la transacción en 2, y liberando el bus para arbitración entre ambas transacciones. La técnica de *split-transaction* se usa para liberar el bus durante tiempos de espera largos y variables entre mensajes de solicitud y respuesta. Al tratar los mensajes de respuesta como transacciones separadas que necesitan arbitración, otras transacciones pueden hacer uso del bus durante los periodos de espera. En un bus sin *split-transaction* la respuesta se identifica por el momento en que ésta aparece en el bus. Con *split-transaction*, dependiendo de la arbitración, las respuestas pueden aparecer en un orden arbitrario y necesitan un medio de identificación.

El caso más simple es aquel donde un esclavo puede decidir si divide su transacción en 2 cuando requiere un gran número de ciclos antes de poder responder al maestro. En este caso el esclavo envía una señal de *split* al árbitro, para que éste enmascare la transacción del maestro (removiendo su permiso temporalmente) y conceda el bus a otro maestro en espera. Cuando el esclavo está listo para completar la transferencia, señala al árbitro que desenmascare la transacción retornando el permiso al maestro para completar la transacción. De este modo se mejora el rendimiento aprovechando los tiempos de espera para realizar otras transferencias de datos.

Un caso particular de transferencia *split* es el modo *Out-of-Order*, donde se permiten múltiples transferencias desde diferentes maestros, o incluso el mismo maestro, y éstas pueden ser divididas por un esclavo y estar en curso simultáneamente en un único bus. En este modo un maestro puede iniciar una transferencia sin esperar a que termine la transferencia anterior. Esto mejora el rendimiento porque permite el procesamiento de múltiples transferencias de datos en

paralelo. Las transferencias pueden completarse en cualquier orden o "fuera de orden" (*out-of-order*), y una forma de identificar las respuestas es mediante un *tag* o identificador, que se envía en el mensaje junto con cada solicitud. El esclavo recuerda el *tag* y lo envía en el mensaje de respuesta, para que el solicitante pueda discriminar cada respuesta con una solicitud en particular. Las transferencias de datos que tienen el mismo *tag* deben completarse en el mismo orden en que las inicio el maestro. Las transferencias de datos con *tags* distintos, iniciadas por diferentes maestros o desde un mismo maestro, no tienen restricciones de orden y pueden completarse en cualquier orden.

2.4.8.5 Transferencia broadcast

Una transferencia de difusión o *broadcast* es un tipo de transferencia donde un maestro emite un mensaje en el bus para que múltiples esclavos registren los datos. La aplicación más directa son los protocolos de *coherencia de cache* o *cache snooping*. Otra aplicación es aquella cuando un maestro necesita informar a otros dispositivos que el sistema se va a apagar o reiniciar.

2.4.9 Topologías de bus

Las arquitecturas basadas en bus pueden tener distintas topologías que afectan el costo, complejidad, potencia, y rendimiento de la arquitectura de comunicación. A continuación se muestran las topologías más usadas en *SoCs*.

En la figura 23 se muestra la topología de bus más simple, donde todos los componentes se conectan por medio de un único medio de difusión compartido, que es suficiente cuando el número de componentes es reducido pero que no es escalable a sistemas con mayor número de componentes y requerimientos de paralelismo. Este esquema se puede extender a una jerarquía de buses interconectados por medio de puentes o *bridges* como se muestra en la figura 24. Este esquema permite transferencias de datos concurrentes en múltiples buses, que pueden tener protocolos distintos y operar a distintas señales de *clock*. La ubicación de componentes es tal que minimiza la interacción entre compones ubicados en buses distintos. En la figura 25 se muestra una forma de jerarquía de buses (*split bus*) más simple donde la división entre buses se hace mediante buffers triestado. Este esquema es más eficiente en consumo de potencia y reduce los problemas que se presentan al usar líneas de bus largas. En general, si el retardo y consumo de potencia de la estructura de buffer triestado bidireccional es menor que la parte del bus que está siendo desconectada, entonces se prefiere la arquitectura *split bus*.

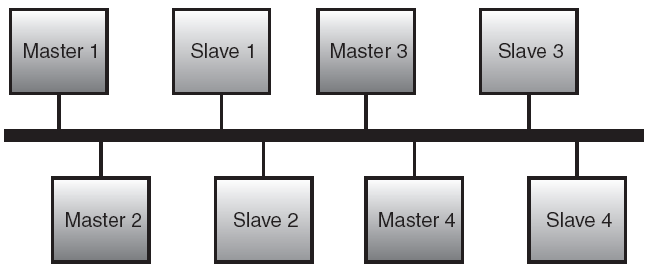


Figura 23: Ejemplo de bus único

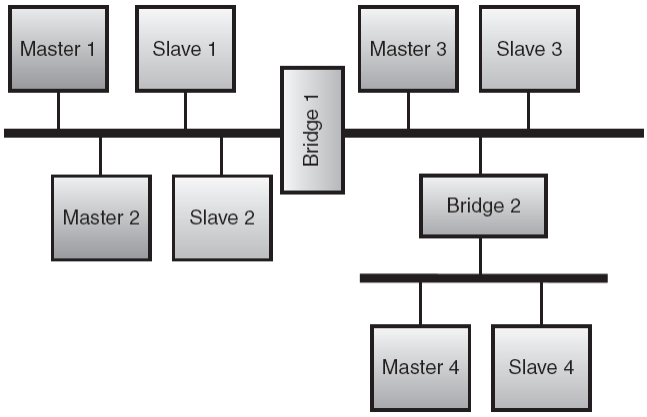


Figura 24: Ejemplo de jerarquía de buses

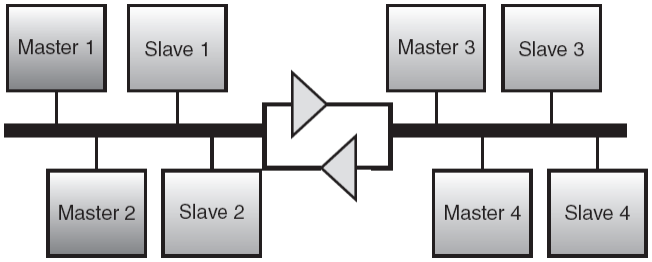


Figura 25: Ejemplo de bus dividido con buffers triestado o split bus

En la figura 26 se muestra una topología de bus usada en sistemas que requieren alto nivel de paralelismo, que combina buses compartidos y conexiones punto a punto, llamada *crossbar bus* o *matriz de buses*. En el ejemplo los componentes de procesamiento (procesadores y memorias) se ubican al lado izquierdo y los componentes periféricos a la derecha.

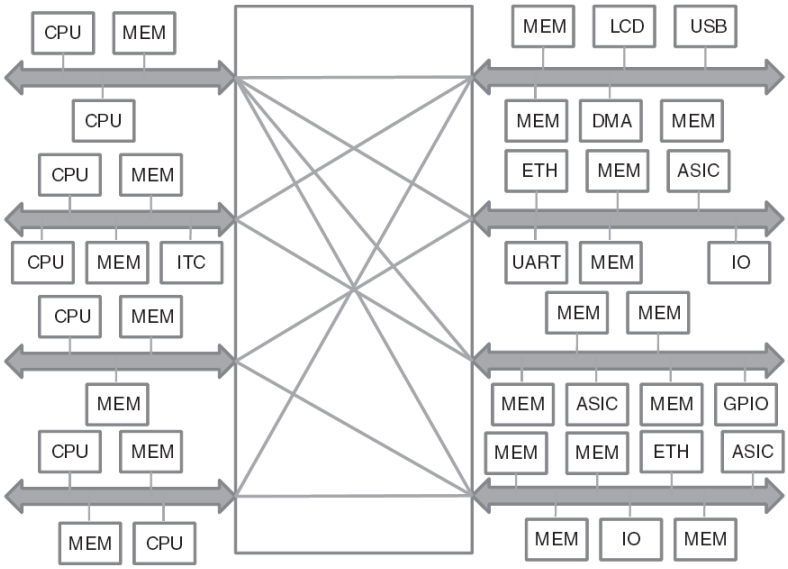


Figura 26: Ejemplo de arquitectura de bus crossbar

En la figura 27 se muestra la topología de *bus anillo*, que consiste en un conjunto de buses pipeline, unidireccionales, y concéntricos, que permiten alta frecuencia de operación y altas tasas de transferencia entre los componentes del bus. En este esquema los datos pueden ser transferidos desde fuente a destino tanto en sentido horario como anti-horario, dependiendo de la disponibilidad de un segmento de bus y la distancia más corta al destino.

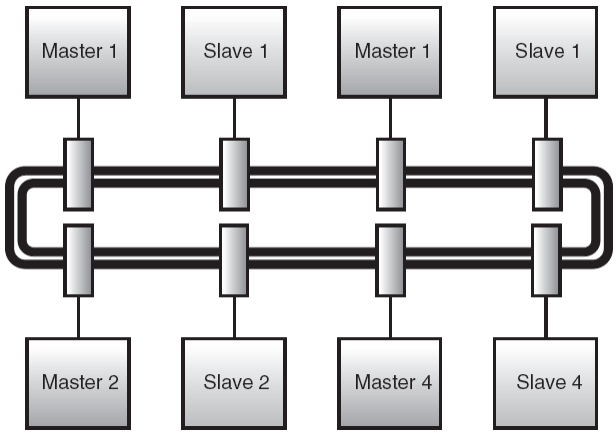


Figura 27: Ejemplo de bus anillo

2.5 Caso de estudio 1: El bus Avalon

El bus Avalon de Altera es una arquitectura de comunicación basada en bus sincrónico, orientada al diseño de *SoPCs* (*Systems on a Programmable Chip*). En particular, se orienta diseño de sistemas basados en el procesador Nios sobre plataforma FPGA de Altera, usando la herramienta de software *SOPC Builder* para generación automática de arquitectura de comunicación.

SOPC Builder toma la especificación del sistema (editado y parametrizado mediante interfaz gráfica), y genera la lógica de interconexión automáticamente. Se generan un conjunto de archivos HDL (ya sea Verilog o VHDL) que definen todos los componentes del sistema, y un módulo HDL de alto nivel que conecta todos los componentes entre sí. En forma adicional provee características típicas de herramientas *PBD*, como por ejemplo generación automática de código para acelerar el desarrollo de software y simulación.

El modelo de bus Avalon (llamado *Avalon System Interconnect Fabric*) está orientado al *PBD*, es decir, tiene un enfoque de diseño orientado a la integración y reuso de componentes por medio de un conjunto de herramientas de software que permiten la configuración y generación automática de un *System Interconnect Fabric*. Se define un *System Interconnect Fabric* como el conjunto de señales y lógica que interconecta los componentes de un sistema organizado en un mapa de memoria. En la figura 28 se muestra un ejemplo de *SoPC* basado en arquitectura de comunicación *System Interconnect Fabric*.

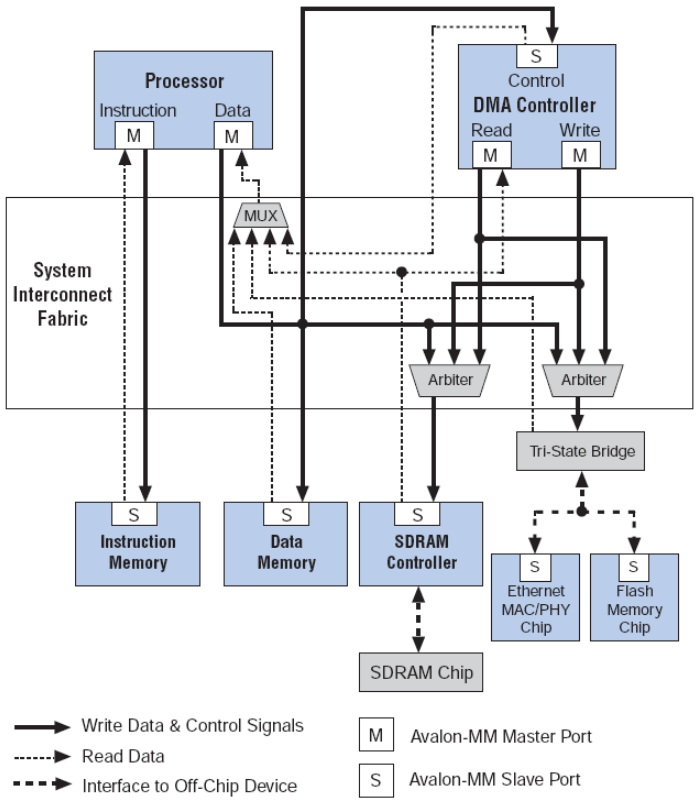


Figura 28: Ejemplo de sistema con arquitectura de comunicación Avalon System Interconnect Fabric

La especificación Avalon define los siguientes tipos de interfaz:

- *Avalon-MM (Memory Mapped) Interface*: Interfaz típica de lectura/escritura basada en dirección para interconexión de maestros y esclavos.
- *Avalon-ST (Streaming) Interface*: Interfaz que soporta flujo unidireccional de datos, incluyendo flujos multiplexados, paquetes, y datos de *DSP*.
- *Avalon-MM Tristate Interface*: Interfaz de lectura/escritura basada en dirección para soportar periféricos *off-chip*.
- *Avalon Clock*: Interfaz que transmite o recibe señales de *clock* y *reset* para sincronización del sistema.
- *Avalon Interrupt*: Interfaz que permite la señalización de eventos entre componentes.
- *Avalon Conduit*: Interfaz que permite la exportación de señales fuera del sistema generado por *SOPC Builder*, para libre conexión dentro o fuera del chip.

En la figura 29 se muestra un ejemplo de diseño basado en el procesador *Nios II* que incorpora todas las interfaces definidas por la especificación Avalon.

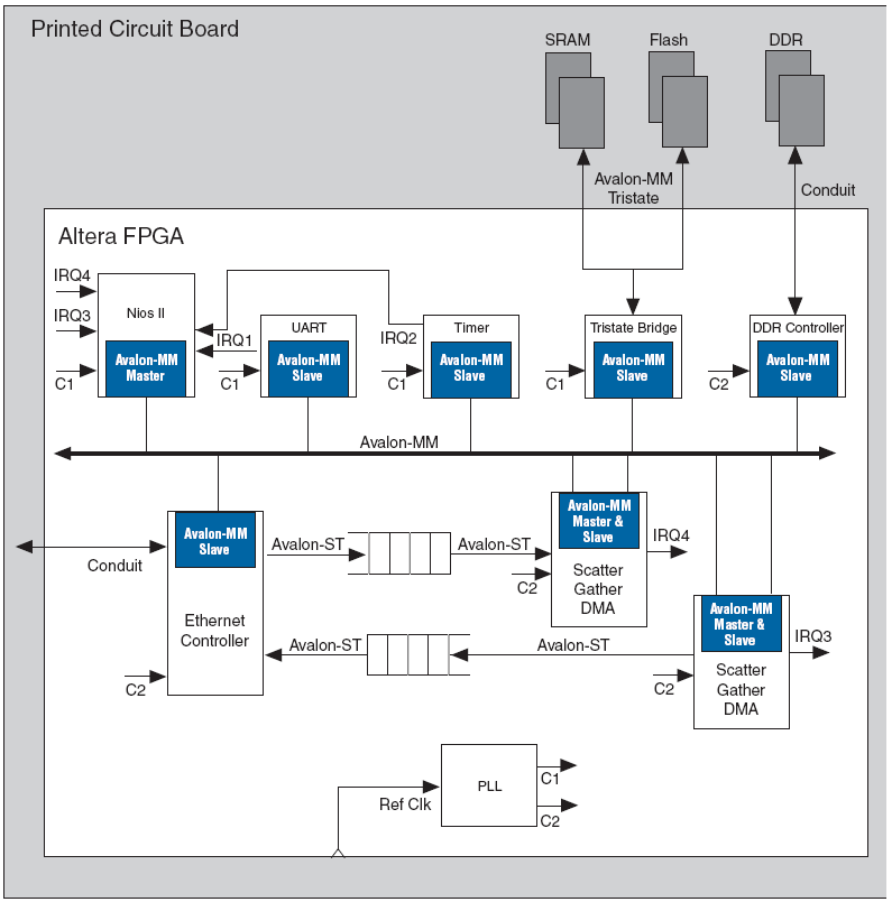


Figura 29: Interfaces Avalon en sistema basado en procesador Nios II

A continuación se describen las características más relevantes del estándar Avalon-MM:

- *Arquitectura maestro/esclavo basada en bus sincrónico*
- *Configurabilidad*: La interfaz de un componente tiene libertad para elegir el conjunto de señales que necesita, es decir, el conjunto de señales posee la interfaz varía en función de la complejidad de las transacciones que soporta.
- *Espacio de memoria de hasta 4GBytes*: Memoria y periféricos pueden ser organizados libremente en dentro de un espacio de dirección de 32-bits.
- *Decodificación centralizada en System Interconnect Fabric*: Los componentes del sistema no requieren lógica de decodificación en su interfaz de bus.
- *Arquitectura de bus multi-maestro*: La lógica de arbitración se genera automáticamente.
- *Soporte para transferencias de datos sobre múltiples dominios de clock*
- *Soporte para transferencias en modo burst y pipelined*
- *Buses de dirección, datos y control por separado*
- *Bus de datos de hasta 1024 bits*: Soporte para anchos de datos arbitrario, incluyendo anchos que no son potencia de dos.
- *Dynamic Bus Sizing*: Manejo automático de los detalles de transferencia de datos entre componentes con distintos anchos de datos.
- *Configuración vía interfaz gráfica*: El usuario especifica el sistema (agregar componentes, especificar relaciones maestro/esclavo, definición de mapa de memoria, parametrizar arbitración) vía interfaz gráfica. La generación automática del *System Interconnect Fabric* se realiza en base a la información especificada por el usuario.

2.5.1 Transferencia de lectura/escritura única

En la figura 30 se muestra el diagrama de *timing* asociado a un esclavo con interfaz Avalon-MM, que soporta transacciones de lectura/escritura con control de estados de espera por medio de la señal *waitrequest*. El esclavo puede pausar la transacción por la cantidad de ciclos que requiera activando la señal *waitrequest*, cuando no está listo para responder al maestro.

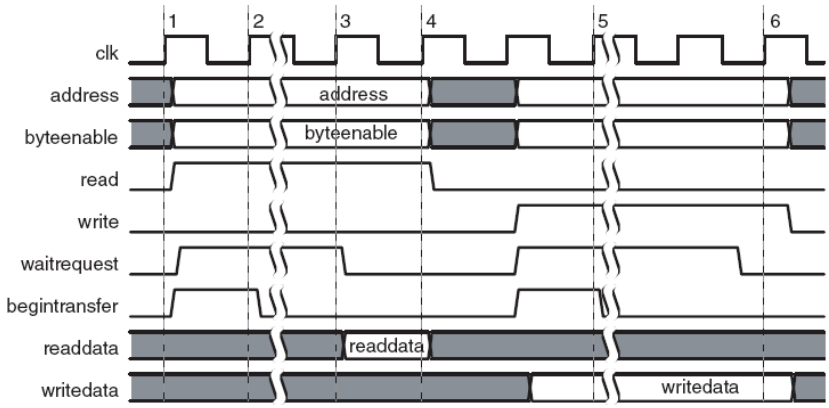


Figura 30: Diagrama de timing para transferencias Avalon de lectura/escritura con estados de espera

A continuación se detalla la secuencia de eventos asociados al ejemplo de la figura 30, ordenados por flanco positivo de los ciclos de bus (1 a 6):

1. El maestro activa las señales *address* (dirección de memoria), *begintransfer* (inicio de transacción) y *read* (transacción de lectura). El esclavo responde introduciendo un estado de espera activando la señal *waitrequest*.
2. El maestro registra la señal *waitrequest*, y en respuesta mantiene las señales activadas el ciclo anterior constantes, con la excepción de *begintransfer* (que se activa solamente durante el primer ciclo de una transacción).
3. El esclavo está listo para proseguir con la transacción y desactiva *waitrequest*. Presenta al maestro los datos solicitados en *readdata*.
4. El maestro detecta la desactivación de *waitrequest*, y registra los datos en *readdata* completando la transacción.
5. El maestro inicia una transacción de escritura activando *address*, *writedata* (datos a escribir), *byteenable* (indica los bytes que son válidos en el bus de datos), *begintransfer*, y *write* (transacción de escritura). El esclavo responde introduciendo un estado de espera activando la señal *waitrequest*.
6. El esclavo completa la transacción registrando los datos en *writedata* y desactivando *waitrequest*.

2.5.2 Transferencias en modo burst

El bus Avalon soporta transferencias en modo burst por medio de la señal *burscount*, que indica al inicio de la transacción el número de transferencias secuenciales que el maestro desea realizar. Para *burstcount* de N bits de ancho, el largo de la transferencia burst debe estar entre 1 y 2^{N-1}.

Un esclavo que soporta modo burst debe incluir como entrada la señal *burstcount*, y tener control de estados de espera por medio de la señal *waitrequest*. Para que un esclavo pueda soportar lecturas en modo burst, también debe soportar transferencias *pipeline* de latencia

variable con la señal *readdatavalid*.

En la figura 31 se muestra el diagrama de timing para una transferencia de escritura en modo burst de largo 4. En el ejemplo, el esclavo retarda la transacción dos veces activando la señal *waitrequest*.

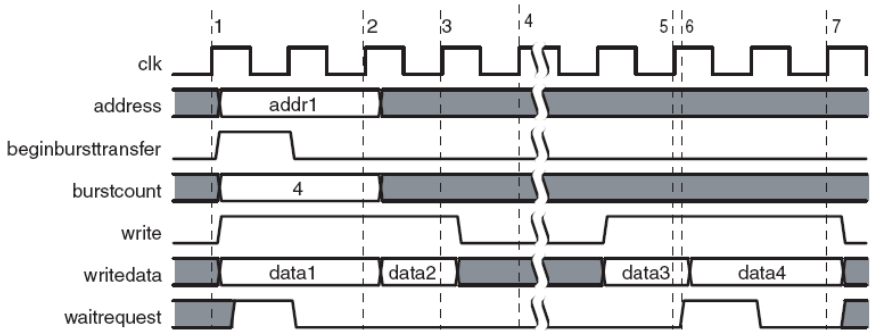


Figura 31: Diagrama de timing para transacción Avalon de escritura en modo burst

A continuación se detalla la secuencia de eventos asociados al ejemplo de la figura 31:

1. El maestro presenta las señales *address* (que contiene la dirección inicial *addr1*), *burstcount*, y la primera unidad de dato a escribir en *writedata*. En respuesta el esclavo activa inmediatamente la señal *waitrequest*, indicando que no está listo para proceder con la transferencia. La señal *beginbursttransfer* es opcional y se activa solamente durante el primer ciclo de la transferencia burst.
2. El esclavo captura la dirección inicial *addr1*, *burstcount*, y el primer dato. El maestro detecta la desactivación de *waitrequest*, y prosigue la transacción presentando el dato siguiente. En los ciclos siguientes las señales *address* y *burstcount* son ignoradas porque el esclavo tiene la información necesaria y suficiente para proseguir con la transacción.
3. El esclavo captura la siguiente unidad de dato.
4. El maestro pausa la transacción desactivando la señal *write*.
5. Con la señal *write* activada nuevamente, el esclavo prosigue la transacción capturando la siguiente unidad de dato.
6. El esclavo pausa nuevamente la transacción activando la señal *waitrequest*.
7. El esclavo termina la transacción capturando la última unidad de dato.

2.5.3 Alineamiento de dirección

En el estándar Avalon-MM la granularidad (unidad atómica de dato) del espacio de dirección de un maestro siempre es en bytes, con ordenamiento *little-endian* (los bytes se ordenan en direcciones que aumentan con su significancia).

Cuando un maestro se conecta a un esclavo de 32-bit, el bus de dirección del esclavo es el

bus de dirección del maestro desplazado a la izquierda en 2 bits, es decir, el esclavo recibe las direcciones provenientes del maestro multiplicadas por 4. El espacio de dirección de un maestro de 32-bit se ordena en bloques de 4 bytes (0x00, 0x04, 0x08, 0x0C, etc.). Para escribir un byte específico dentro de una palabra se indica por medio de las señales *byteenable*.

El *System Interconnect Fabric* provee un servicio llamado *Dynamic Bus Sizing*, que se encarga de manejar dinámicamente los datos en transferencias de pares maestro/esclavo que difieren en ancho de datos, de tal forma que los datos del esclavo se encuentren alineados en bytes contiguos en el espacio de dirección del maestro. Esta función requiere que el bus de datos del esclavo sea de ancho 8, 16, 32, 64, 128, 256, 512 o 1024 bits.

Si el maestro es más ancho que el esclavo, los bytes de datos en el espacio de dirección del maestro corresponden a múltiples locaciones en el espacio de direcciones del esclavo. Por ejemplo, si un maestro de 32-bit realiza una lectura a un esclavo de 16-bit, el *System Interconnect Fabric* ejecuta 2 transferencias de lectura en 2 direcciones consecutivas del esclavo, y presenta una sola palabra de 32-bit al maestro.

Si el maestro es menos ancho que el esclavo, durante transferencias de lectura el *System Interconnect Fabric* presenta al maestro solamente los bytes de datos apropiados del esclavo. En el caso de transferencias de escritura, se activan automáticamente las señales *byteenable* para escribir datos solo a los bytes especificados del bus de datos.

2.6 Caso de estudio 2: El bus Wishbone

Wishbone es una arquitectura de comunicación *on-chip* basada en bus sincrónico, mantenida como estándar abierto por la comunidad Opencores[3]. Su filosofía de diseño es proveer una especificación de bus sincrónico de alta velocidad para interconexión de *IP cores* en *SoC*. Debido a falta de soporte de características avanzadas tales como transacciones *split*, su espacio de aplicación se limita a sistemas embebidos de rango pequeño a mediano.

A continuación se describen las características más relevantes del estándar Wishbone:

- *Arquitectura maestro/esclavo basada en bus sincrónico*
- *Arquitectura de bus multi-maestro*
- *Esquema de arbitración flexible*
- *Bus de datos de hasta 64 bits*: configurable entre 8 y 64-bits.
- *Soporte para transacciones de lectura/escritura única y en modo burst*
- *Soporte para transacciones de tipo RMW (Read-Modify-Write) para operaciones de tipo semáforo*
- *Soporta topologías de tipo punto-a-punto, flujo de datos, bus compartido y crossbar*
- *Soporte para cancelación de transacción o modo Retry*
- *Provee identificadores o tags definidos por usuario para soportar requerimientos específicos de la aplicación*
- *Soporte para organización de datos little-endian y big-endian*

Wishbone soporta transferencia básica de lectura/escritura con handshaking, y transferencia en modo burst. Además provee un tipo de transferencia semáforo llamado *RMW* (*Read-Modify-Write*) que permite a múltiples componentes compartir recursos comunes. En este tipo de operación, una vez que el árbitro concede el bus al maestro, ningún otro maestro puede acceder al bus hasta que el maestro seleccionado haya leído, modificado, y escrito datos al esclavo.

En general, si una interfaz soporta más de un tipo de ciclo Wishbone sobre un conjunto de señales común, en general se incluyen *tags de ciclo* para discriminar entre los distintos tipos de ciclo. También se pueden usar *tags de dirección* o *tags de datos* para indicar eventos o información sobre la transacción, por ejemplo un maestro con lógica de chequeo de paridad puede incluir un tag llamado PAR_O que se activa cada vez que el bus de datos contiene un número par de unos.

En el estándar Wishbone se identifican las señales de entrada y salida con los sufijos _I (*input*) y _O (*output*) respectivamente, siempre referidas a una interfaz en particular. Por ejemplo, la señal DAT_O (*data output*) de un maestro, quiere decir que es entrada para el esclavo y salida para el maestro.

3 Opencores es una comunidad *open source* cuyo propósito es el desarrollo y libre distribución de *IP cores* en formato HDL (VHDL y/o Verilog).

2.6.1 Ciclo de lectura/escritura única

En la figura 32 se muestra el diagrama de *timing* asociado a un *ciclo de lectura única* para un par maestro/esclavo con interfaces Wishbone (las señales están referidas al maestro). El esclavo puede introducir estados de espera o *WSS* (*wait state slave*) retardando la activación de la señal de acuse de recibo ACK_I (*acknowledge input*). En un ciclo dado, el esclavo no posee información sobre lo que el maestro hará el próximo ciclo. A este tipo de ciclo se le llama C*iclo Wishbone Clásico*.

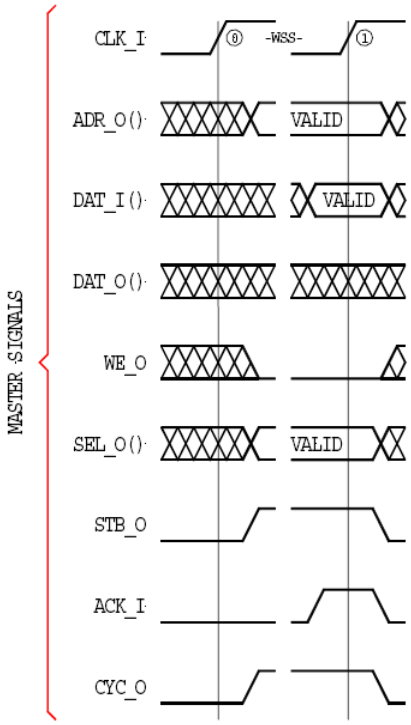


Figura 32: Diagrama de timing para ciclo Wishbone de lectura única con estado de espera

A continuación se detalla la secuencia de eventos asociados al diagrama de la figura 32:

1. Flanco positivo clock 0: El maestro presenta ADR_O (*address output*), desactiva WE_O (*write enable output*) para indicar ciclo de lectura (y no de escritura). Presenta SEL_O (*select output*) para indicar los bytes donde espera los datos (análogo a *byteenable* en Avalon). Para indicar el inicio de la transacción activa CYC_O (*cycle output*), y para indicar el inicio de la fase activa STB_O (*strobe output*).
2. Setup flanco positivo clock 1: El esclavo decodifica sus entradas y responde ACK_I cuando está listo. Para pausar el bus mientras no esté listo para responder al maestro, puede demorar la activación de ACK_I, durante la cantidad de ciclos que requiera. Una vez que está listo, activa ACK_I y presenta datos en DAT_I.
3. Flanco positivo clock 1: El maestro detecta la señal ACK_I, registra los datos en DAT_I, y

finaliza la transacción desactivando STB_O y CYC_O. El esclavo desactiva ACK_I en respuesta a la desactivación de STB_O.

En el caso de transferencia de escritura, el maestro activa la señal WE_O. En forma adicional, el estándar Wishbone permite la activación de señales indicadoras o *tags* definidas por el usuario para implementar características específicas a la aplicación.

2.6.2 Ciclo de transferencia por bloque

En la figura 33 se muestra un diagrama de timing para una transferencia en modo burst o *ciclo de transferencia por bloque*, donde la interfaz realiza una serie de transacciones individuales (llamadas *fases*) que conforman un bloque de datos. El ejemplo también muestra como el maestro y el esclavo pueden introducir estados de espera (*WSM* y *WSS* respectivamente). En rojo se muestra el timing asociado a los *tags* opcionales que puede incluir la interfaz, que pueden ser del tipo TGA_O (*address tag*), TGD_I (*data tag*) y TGC_O (*cycle tag*).

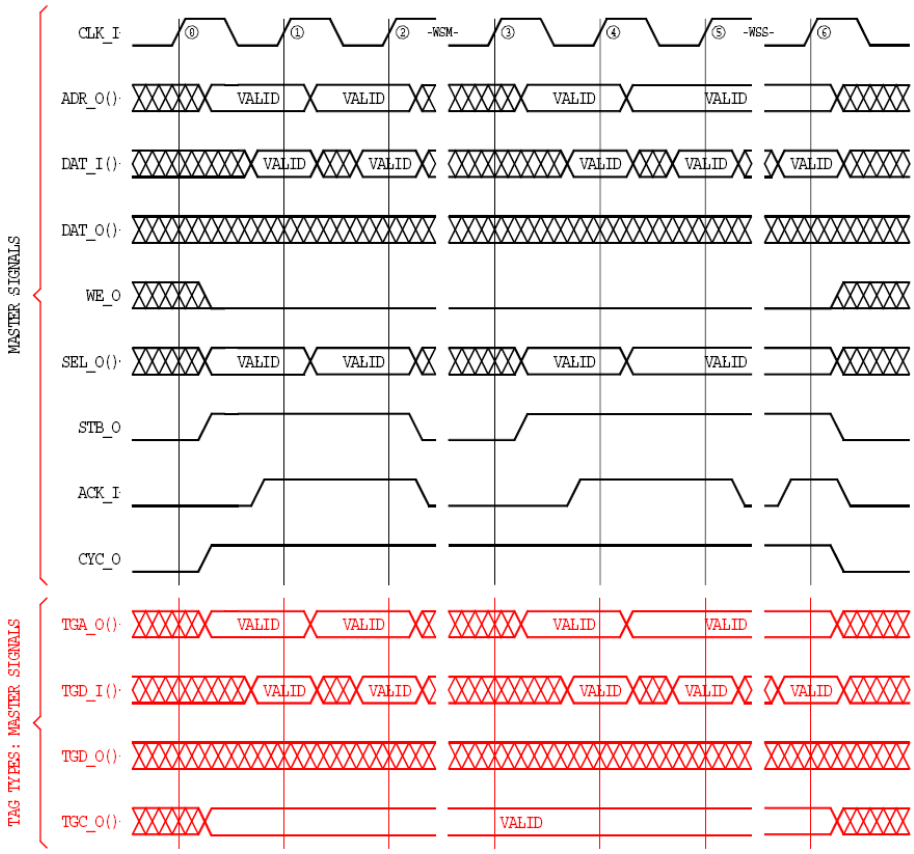


Figura 33: Diagrama de timing para ciclo Wishbone de transferencia por bloque

A continuación se detalla la secuencia de eventos asociados al diagrama de la figura 33:

1. Flanco positivo clock 0: El maestro presenta ADR_ y TGA_O, desactiva WE_O para

indicar ciclo de lectura, presenta SEL_O para indicar los bytes donde espera los datos, activa CYC_O y TAG_O para indicar el inicio del ciclo, y activa STB_O para indicar el inicio de la primera fase.

2. Setup flanco positivo 1: El esclavo decodifica sus entradas, presenta DAT_I y TGD_I, y responde activando ACK_I. El maestro detecta ACK_I y se prepara para capturar los datos.
3. Flanco positivo clock 1: El maestro captura DAT_I y TGD_I, y presenta nuevos datos en ADR_O, TGA_O, y SEL_O.
4. Setup flanco positivo 2: Similar a 2.
5. Flanco positivo clock 2: El maestro captura DAT_I y TGD_I, pero no está listo para responder. Luego, introduce un estado de espera desactivando STB_O.
6. Setup flanco positivo 3: El esclavo desactiva ACK_I en respuesta a la desactivación de STB_O.
7. Flanco positivo clock 3: El maestro está listo y presenta nuevos datos en ADR_O, TGA_O, y SEL_O. Reanuda la transacción activando STB_O.
8. Setup flanco positivo 4: Similar a 2.
9. Flanco positivo clock 4: Similar a 3.
10. Setup flanco positivo 5: Similar a 2.
11. Flanco positivo clock 5: El maestro captura DAT_I y TGD_I. El esclavo no está listo para responder. Luego, introduce un estado de espera desactivando ACK_I.
12. Setup flanco positivo 6: Similar a 2.
13. Flanco positivo clock 6: El maestro captura DAT_I y TGD_I, y termina el ciclo desactivando STB_O y CYC_O.

2.6.3 Ciclos burst con terminación sincrónica avanzada

El esquema de terminación de ciclos Wishbone en modo burst es un híbrido entre esquema sincrónico y asíncrono, llamado *terminación sincrónica avanzada*. En este esquema, una vez que se transfiere una unidad de dato (en forma sincrónica al flanco positivo del clock), el maestro puede desactivar en forma asíncrona CYC_O y STB_O, y en respuesta el esclavo termina el ciclo desactivando su señal de acuse de recibo ACK_I, al igual que en un ciclo Wishbone Clásico. Pero si el maestro mantiene activada la señal STB_O, el esclavo sabe con anticipación que será el objetivo de una transacción al siguiente ciclo, y decide mantener activada la señal ACK_I (ahorrando ciclos de bus), así el maestro inicia inmediatamente la siguiente transferencia como se muestra en la figura 34. Los ciclos de bus que implementan terminación sincrónica avanzada se llaman *Ciclos Wishbone Registered Feedback*, y usan el tag CTI (*Cycle Type Identifier*).

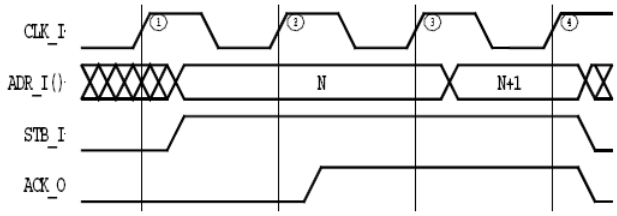


Figura 34: Modo burst con terminación sincrónica avanzada

El maestro envía el tag CTI (3 bits) al esclavo para indicar si el ciclo es de tipo Clásico o Registered Feedback (burst con terminación sincrónica avanzada). El esclavo captura esta información y se prepara para el siguiente ciclo. En la tabla 2 se muestran los posibles valores del tag CTI.

CTI	**Descripción**
000	Classic cycle
001	Constant address burst cycle
010	Incrementing burst cycle
011	Reserved
100	Reserved
101	Reserved
110	Reserved
111	End-of-burst

Tabla 2: Descripción de tag CTI para ciclos Registered Feeedback

Cuando el maestro realiza una transferencia en modo *incrementing burst cycle* (múltiples accesos a direcciones consecutivas) también debe enviar al esclavo el tag BTE (*Burst Type Extension*) indicando el modo de incremento en la dirección. En la tabla 3 se muestran los posibles valores del tag BTE.

BTE	**Descripción**
00	Linear burst
01	4-beat wrap burst
10	8-beat wrap burst
11	16-beat wrap burst

Tabla 3: Descripción de tag BTE para ciclos Registered Feedback

El incremento en la dirección puede ser lineal o *wrapped* (encapsulado). Incremento lineal significa que la siguiente dirección es un incremento más que la actual. El tamaño del incremento depende del ancho del bus de datos (para 8 bit es 1, para 16 bit es 2, para 32 bit es 4, etc.).

El incremento de dirección en el modo *wrapping burst* es similar al modo lineal, pero si la dirección inicial de la transferencia de datos no está alineada con el número de bytes por transferencia, entonces la dirección se va a ajustar cuando se alcance el límite. Por ejemplo, en el modo *wrapping burst* de tamaño 4, si se transfieren palabras de 32 bit (4 bytes) los bloques de datos se encapsularan cada 16 bytes. Entonces si la dirección de inicio es 0x04, las primeras 4 direcciones de la transferencia serán 0x04, 0x08, 0x0C y 0x00. Si la transacción fuera en modo burst lineal, la dirección excedería el límite de 16 bytes (0x04, 0x08, 0x0C y 0x10).

2.7 Interconexión de protocolos diferentes

Dada la creciente complejidad que exhiben los diseños de *SoCs* y las exigencias de *time to market* cada vez más cortos, la tendencia apunta hacia el reuso de componentes de propiedad intelectual o *IP cores*, versus el diseño de nuevos componentes desde cero. La gran mayoría de los componentes de hardware existe en la forma de *IP cores*, que son descripciones de hardware a nivel RTL (*register transfer level*) en un *HDL* (*hardware description language*) tal como Verilog, VHDL, o SystemC. El reuso de *IP cores* que fueron desarrollados por distintos proveedores (usando metodologías y tecnologías de implementación distintas), con distintos protocolos de comunicación y frecuencias de operación, es un problema de gran importancia en el diseño de *SoCs*. Cuando se requiere integrar un *IP core* de un proveedor distinto a un nuevo diseño, un problema importante a resolver es la adaptación de la interfaz del componente al modelo de comunicación específico del sistema sin afectar la funcionalidad del *IP core*.

La formulación del problema anterior puede expresarse de la siguiente manera [2]: *Dados dos agentes que se comunican intercambiando datos, y la descripción de los protocolos que cada uno usa para la transferencia de datos, determinar una interfaz tal que las transferencias de datos sean consistentes con ambos protocolos*. Se define *protocolo* como *la secuencia legal de valores que puede aparecer en los puertos de datos y control desde el inicio hasta el final de una transferencia de datos*.

La lógica de pegamento (o *glue logic*) necesaria para interconectar esquemas de comunicación distintos puede ser generada en forma manual o automática. La modificación manual de la interfaz de un *IP core* requiere de un conocimiento detallado tanto del protocolo nativo del componente, como del protocolo de comunicación del sistema. Luego, esta metodología tiene la desventaja de ser compleja y propensa a errores.

La generación automática de encapsuladores de propiedad intelectual o *IP wrappers* (cuya función es superponer un protocolo de comunicación distinto sobre el IP core sin afectar su funcionalidad), se enmarca dentro del problema general de *síntesis de interfaz*. La investigación en el área de generación automática *IP wrappers* ha producido una gran variedad de metodologías entre las cuales destacan las basadas en grafos, métodos formales, teoría de autómatas, redes de Petri, gramática de protocolos, y orientación a objeto. Sin embargo, pese a los grandes avances en automatización de síntesis de interfaz, debido al amplio rango de aplicaciones y distintas arquitecturas, con distintas restricciones de costo y rendimiento, el trabajo de automatización es aún específico a cada caso [3].

2.7.1 Metodologías de diseño de interconexión

Para abordar el problema de interconexión de *IP cores* se han propuesto metodologías basadas en *interfaz* [4], en *comunicación* [5], y en *transacción*. Estas metodologías enfatizan la necesidad de *ortogonalizar las piezas de interés* o *concerns* [6], es decir, separar la computación (el comportamiento funcional de los componentes) de la comunicación (interacción entre componentes).

El *diseño basado en interfaz* es una metodología de diseño que se enfoca en la explícita separación entre la funcionalidad de los IP cores y el modelo de comunicación del sistema. El diseño se lleva a cabo en dos etapas: (a) el diseño o selección de los IP cores, y (b) el diseño de la interacción entre ellos por medio de un protocolo de comunicación apropiado. Esta separación permite: explorar distintos modelos de comunicación, modelar el comportamiento del sistema en forma independiente de la comunicación, y el reuso de modelos de comunicación existentes.

La investigación en el área de *diseño basado en interfaz* se puede categorizar en:

1. *Optimización de interfaz*: Técnicas para mejorar el rendimiento del sistema optimizando la comunicación entre componentes a alto nivel, en términos de una utilización óptima de los recursos (buses, buffers, lógica de control, etc.).
2. *Lenguajes de especificación de interfaz*: Desarrollo de lenguajes de especificación de interfaz, o métodos formales para generar protocolos de comunicación a partir de especificaciones de alto nivel.
3. *Conversión de protocolos*: Diseño de conversores de protocolo que permiten la comunicación entre IP cores incompatibles.
4. *Síntesis de interfaz a nivel de sistema*: Expande el problema de generación de interfaz a comunicación entre componentes de hardware y software.
5. *Estandarización de interfaces*: Reconociendo la necesidad de simplificar la interconexión de componentes en el diseño de *SoCs*, el consorcio industrial VSIA (*Virtual Socket Interface Alliance*) ha propuesto estándares para el formato de datos, metodologías de test, e interfaces (estándar de interfaz VCI o *Virtual Component Interface*) [7]. Sin embargo, la intensa competencia en la industria de IP cores ha dificultado la adopción de un estándar de comunicación común, lo que preserva la necesidad de sintetizar interfaces entre componentes con interfaces incompatibles. Actualmente el consorcio VSIA ha cerrado sus operaciones y donado sus trabajos a la organización IEEE.

A continuación se presenta un marco teórico para la generación de IP wrappers en forma sistemática, usado en la metodología de diseño basado en interfaz [8].

2.7.1.1 Análisis de interfaz de IP cores

El espacio de diseño de IP cores es *multidimensional*, es decir, se aplican los principios de *separación multidimensional de piezas de interés* (o *multidimensional separation of concerns*) [9]. Esto permite considerar las diversas dimensiones ortogonales y con traslape en el espacio de diseño, así como también su separación, representación e integración cuando se describe e implementa un sistema complejo.

Una de las dimensiones de un IP core es su interfaz (figura 35), que representa su comportamiento externo y tiene varias capas de abstracción tales como capa física (niveles de voltaje lógicos, corriente, carga capacitiva), timing (espaciamiento temporal entre eventos), transacción de datos (transferencia de datos a nivel de bit), paquete (transferencia de datos a nivel de bloque), y mensaje (comunicación entre procesos). La interfaz de un IP core involucra solamente las capas de abstracción más altas (transacción de datos y superiores).

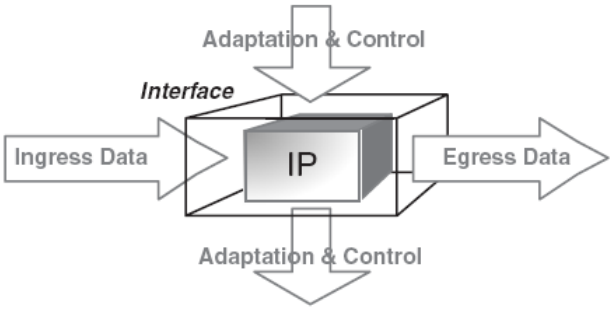


Figura 35: Vista general de una interfaz a nivel de sistema

El propósito principal de la interfaz es asegurar el movimiento de datos relevante (operandos, comandos, direcciones, valores, etc.) desde y hacia el IP core, en función de un conjunto de reglas o *protocolo*. En términos del diseño basado en interfaz, la interfaz del IP core cumple dos roles: (1) especificar *explícitamente* los puertos I/O del componente (por ejemplo en la declaración de puertos del módulo de alto nivel en Verilog), y (2) especificar *implícitamente* el modelo de comunicación del IP core (lo que requiere información adicional sobre los detalles de implementación del componente).

La interfaz de un IP core se descompone en las siguientes dos dimensiones:

1. *Dimensión vertical*: Corresponde a los diferentes tipos de señales, que pueden ser de datos, control, señales específicas del protocolo de comunicación, señales de acceso a memoria, configuración, estatus, etc.
2. *Dimensión horizontal*: Corresponde a los aspectos específicos de una señal en particular, tales como nombre de señal, dirección (entrada/salida), ancho de datos y tipo de datos.

2.7.1.2 Síntesis de interfaz

Para diseñar un sistema compuesto por distintos IP cores que se comunican entre sí, debe escogerse y usarse un modelo estándar de comunicación. Usualmente, cuando un diseñador escoge un nuevo IP core, inicialmente lo selecciona considerando solamente su comportamiento *estático* (funcionalidad). Sin embargo, para realmente implementar el IP core, eventualmente el diseñador debe diseñar o adaptar su comportamiento *dinámico* (protocolo) al modelo de comunicación del sistema. Este es el problema central de la *síntesis de interfaz*.

El proceso de síntesis de interfaz (figura 36) se compone de los siguientes pasos fundamentales:

1. Extraer los parámetros caracterizan la interfaz del IP core.
2. Generar una nueva interfaz compatible con el modelo de comunicación del sistema.
3. Implementar la lógica de pegamento necesaria para realizar la conversión de protocolo.

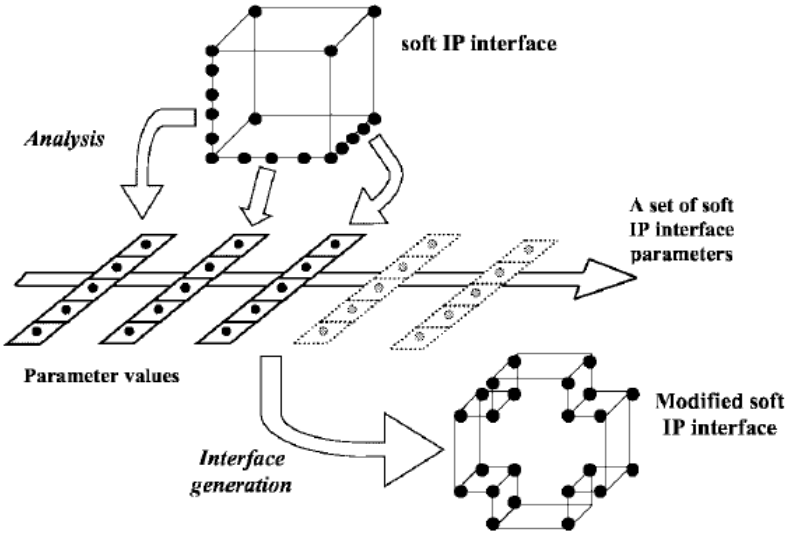


Figura 36: Esquema del proceso de síntesis de interfaz

Mediante el análisis de la descripción de alto nivel de la interfaz del IP core, se extraen los parámetros que la caracterizan únicamente. Estos parámetros permiten generar la interfaz modificada requerida por la lógica de pegamento.

La generación de la interfaz modificada requiere de (al menos) las siguientes *transformaciones de interfaz*:

1. *Composición inclusiva*: Se fusionan las interfaces de los IP cores distintos, y todas las señales aparecen en la nueva interfaz.
2. *Composición exclusiva*: Se fusionan las señales con un mismo significado (por ejemplo, una señal de clock compartida).
3. *Modificación de tipo*: Se realiza sobre un subconjunto de señales cuando se requiere conversión de tipo para la integración de IP cores.
4. *Modificación de ancho*: Se realiza sobre un subconjunto de señales cuando un IP core se usa en un contexto donde el ancho del camino de datos o *data path* es diferente.
5. *Inclusión de señal*: Se incluye una o varias señales nuevas en la interfaz, para control u otros propósitos (por ejemplo, *Built-in Self Test*, etc.).
6. *Exclusión de señal*: Se elimina una o varias señales de la interfaz.

2.7.1.3 Concepto de IP wrapper

Durante el proceso de síntesis de interfaz se aplican una serie de transformaciones por sobre

el IP core (visto como *caja negra*), es decir, el proceso genera un componente llamado *IP wrapper* (o encapsulador de propiedad intelectual) que encapsula el comportamiento del IP core sin modificar su implementación interna. De esta manera se superpone un protocolo de comunicación sobre el IP core.

Un IP wrapper es un componente que permite adaptar la interfaz y comportamiento de un componente al contexto y requerimientos de una aplicación. Esto facilita la reusabilidad de IP cores, y permite su integración en diferentes sistemas de hardware.

La encapsulación puede ser anidada, es decir, se pueden aplicar wrappers a un IP wrapper varias veces, incurriendo en un mayor costo de retardo y overhead. También pueden aplicarse distintos wrappers a un mismo IP core.

2.7.2 Estrategias de conversión de protocolos

La investigación en el área de síntesis de interfaz ha generado una variedad de enfoques, de los cuales la mayoría considera la generación de interfaz como un problema de síntesis de transductores.

K. Chung *et al.* [10] proponen un algoritmo que toma como entrada la caracterización de los diagramas de timing de dos sistemas en la forma de *STGs* (*signal transition graphs*), y genera lógica de pegamento entre ambos protocolos (circuito combinacional). La ventaja del algoritmo es la optimización de área y consumo de potencia, con la limitación de generar solamente lógica combinacional y no manejar diferencias en ancho de datos.

R. Passerone *et al.* [2] proponen un algoritmo que usa descripciones de protocolos basadas en *expresiones regulares*, para generar una máquina de estados finitos que implementa la conversión de protocolo. Primero, las expresiones regulares que representan los dos protocolos son traducidas en dos autómatas que reconocen el correspondiente lenguaje regular. Luego, se toma el producto de los autómatas tal que se mantiene la secuencia legal de operaciones, las señales se traducen en entradas y salidas, y por último se resuelve el no-determinismo que viola uno o ambos protocolos usando una o más de las siguientes reglas: (1) no poniendo nunca un dato en salida si aún no ha sido recibido, (2) transfiriendo datos, o (3) minimizando la latencia. Cualquier no-determinismo residual se elimina usando elecciones arbitrarias basadas en el orden que se generen los estados. El método no maneja información de alto nivel tal como el tipo de conversión y el ajuste de ancho de datos, y tiene la limitación de requerir una señal de clock común a ambos protocolos. Esta metodología ha sido extendida para manejar protocolos operando a distintas frecuencias por B. Park *et al.* [11], mediante la incorporación de estados adicionales y una cola en el conversor de protocolo. R. Passerone *et al.* [12] extendieron el trabajo anterior y proporcionaron un formalismo más riguroso para generar conversores de protocolo usando algoritmos basados en teoría de juegos.

D. Gajski y S. Narayan [13] proponen un método basado en descripciones HDL de los dos protocolos, donde se detallan los anchos de buses de datos y control, y la secuencia temporal de eventos sobre el bus. La descripción procedural (sentencias VHDL secuenciales) de un protocolo basa en un conjunto de 5 operaciones atómicas genéricas: (1) esperar un evento en una entrada de control, (2) asignar un valor a una salida de control, (3) lectura desde una entrada de datos, (4) asignar un valor a una salida de datos, y (5) esperar durante un intervalo fijo. La descripción del protocolo se descompone en bloques o grupos de relaciones, cuya ejecución está determinada por

una de las líneas de control, o un retardo. Luego, las relaciones extraídas de ambos protocolos se agrupan en conjuntos que transfieren la misma cantidad de datos. El algoritmo genera una descripción HDL del conversor de protocolo e información relacionada con la conexión de puertos de ambos protocolos. Este método permite el manejo de diferencias en los anchos de datos, y diferencias en las frecuencias de clock de ambos protocolos. Sin embargo, la descripción procedural de ambos protocolos dificulta la adaptación de casos donde el secuenciamiento de datos es distinto. J. Smith y G. Micheli [14] proponen un enfoque donde a partir de las descripciones en HDL de ambos protocolos, un algoritmo mapea ambos protocolos a un esquema de comunicación estándar, que luego es implementado como una máquina de estados con interfaces sincrónicas. Esta metodología puede manejar diferencias en las señales de clock, y tanto en el caso de buses unidireccionales como bidireccionales.

Trabajos más recientes en el área de síntesis de interfaz para adaptación de protocolos, han producido *bridges* y adaptadores de interfaz para estándares de comunicación emergentes en la industria tales como AMBA, CoreConnect, STBus, Avalon, y estándares de interfaz tales como OCP y VCI. Los conversores de protocolo o *bridges* permiten la transferencia de datos entre buses distintos. En vez de usar IP wrappers para cada componente (es decir, conversión de protocolo distribuida), un *bridge* es un componente que encapsula la lógica de conversión de forma centralizada.

2.7.3 Desarrollo de un nuevo Avalon/Wishbone wrapper para el Opencores PCI Bridge IP core

El presente trabajo aporta el desarrollo de un nuevo IP wrapper para el IP core PCI Bridge de Opencores en una aplicación industrial real[4]. Para esto fue necesario desarrollar una metodología que produce lógica combinacional que encapsula el IP core (con interfaz de bus Wishbone) superponiendo el protocolo de comunicación Avalon, para su integración en un sistema basado en arquitectura de comunicación System Interconnect Fabric. La metodología usada resulta en una interfaz que minimiza el uso de recursos lógicos y consumo de potencia, gracias al uso de lógica combinacional (es decir, sin uso de máquinas de estado).

4 Ver sección 4.1

3 Buses de computadores

Se define I/O (Entrada/Salida) como un subsistema que transporta datos codificados entre dispositivos externos y un sistema *host* (anfitrión), compuesto principalmente por una CPU y memoria del sistema. El subsistema I/O incluye al menos los siguientes elementos:

- Bloques de memoria dedicados para funciones de I/O.
- Buses de datos que proveen un medio para transportar datos desde y hacia el sistema.
- Módulos de control en el sistema host y en dispositivos periféricos.
- Interfaces a dispositivos externos tales como teclados y discos.
- Cableado o enlaces de comunicación entre el sistema host y sus periféricos.

En la figura 37 se ilustra como se pueden integrar los elementos anteriores para conformar un subsistema I/O.

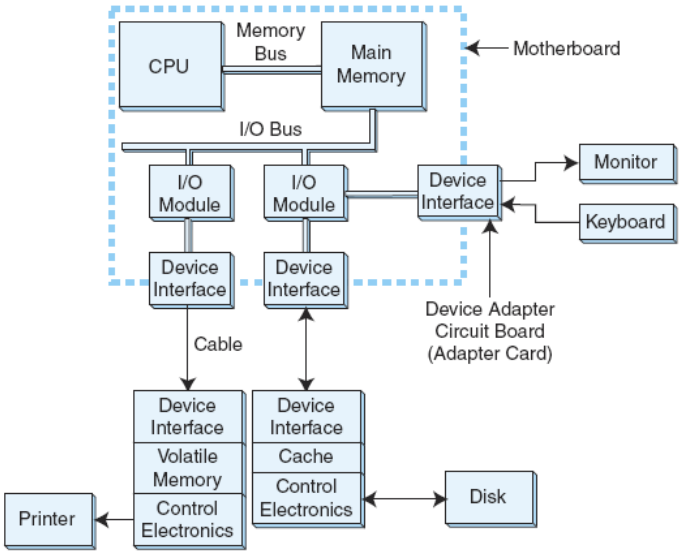


Figura 37: Ejemplo de subsistema I/O

3.1 Métodos de control I/O

A continuación se describen tres métodos para controlar el flujo de datos desde y hacia el sistema host.

3.1.1 I/O programada

Los sistemas basados en I/O programada contienen al menos un registro dedicado para cada

dispositivo I/O. La CPU monitorea continuamente el estado del registro (generalmente llamado "registro de estatus") para esperar la llegada de un dato al dispositivo. Este mecanismo recibe el nombre de *polling*. Luego, una vez que la CPU detecta la condición de "dato listo", actúa en función de las instrucciones programadas para ese evento en particular.

El beneficio de usar este enfoque es que se tiene un control programable sobre el comportamiento de cada dispositivo. El programa se puede ajustar al número y tipo de dispositivos en el sistema, así como también a sus prioridades de *polling* e intervalos. La desventaja del *polling* es que la CPU se mantiene en un continuo estado de "espera ocupada", donde no ejecuta ninguna instrucción útil hasta la llegada de un evento I/O.

3.1.2 I/O controlada por interrupciones

Al contrario del enfoque anterior donde la CPU monitorea continuamente un registro de estatus, los dispositivos controlados por interrupción notifican a la CPU la ocurrencia de un evento I/O. Luego la CPU puede ejecutar otras tareas hasta que un dispositivo lo interrumpa para el atendimiento de un evento, como por ejemplo la llegada de datos para enviar. En general, las interrupciones son señaladas mediante un bit en el *registro de flags* de la CPU, llamado *flag de interrupción*.

Una vez que el flag de interrupción es encendido, el sistema operativo interrumpe el actual programa en ejecución guardando su estado e información de variables. Luego la CPU busca el *vector de dirección* que apunta a la dirección de la *rutina de atendimiento de interrupción* I/O. Una vez que la CPU termina de atender la interrupción para el dispositivo I/O, entonces restaura la información guardada del programa que estaba corriendo cuando ocurrió la interrupción, y el programa se vuelve a ejecutar desde el punto donde fue interrumpido.

3.1.3 Acceso directo a memoria

El principal concepto de Acceso Directo a Memoria (*DMA*) es eliminar el rol de "intermediario" de la CPU para la transferencia de datos entre dispositivos. Permite a los dispositivos periféricos transferir datos directamente desde y hacia memoria sin la intervención de la CPU, liberándola para ejecutar otras tareas y así mejorar el rendimiento especialmente en los casos donde hay grandes transferencias de datos.

Un *controlador DMA* es un dispositivo que controla uno o más dispositivos periféricos. Generalmente se compone de al menos tres interfaces de bus: un maestro de lectura (que toma los datos desde el origen), un maestro de escritura (que escribe datos en destino), y un esclavo de control para acceder a un conjunto de registros de configuración internos (estatus, control, dirección de lectura, dirección de escritura, largo de transacción o cantidad de bytes a transferir, etc.).

La CPU inicializa el controlador DMA escribiendo a registros específicos a través de su esclavo de control. Algunos parámetros típicos incluyen la dirección de origen del bloque de datos, la dirección de destino, el largo del bloque (medido en cantidad de bytes) y un bit para habilitar generación de interrupción cuando la transferencia termina. En general es posible habilitar que el controlador DMA incremente automáticamente los registros de dirección para que después de transferir cada palabra, la próxima transferencia se realice desde y hacia la siguiente posición en memoria.

El controlador DMA y la CPU usan el mismo bus de memoria y por lo tanto debe existir un mecanismo de arbitración para gestionar los permisos cuando múltiples maestros desean tomar el control del bus al mismo tiempo. En la figura 38 se muestra un ejemplo donde el controlador DMA comparte el bus de memoria con una CPU que soporta DMA.

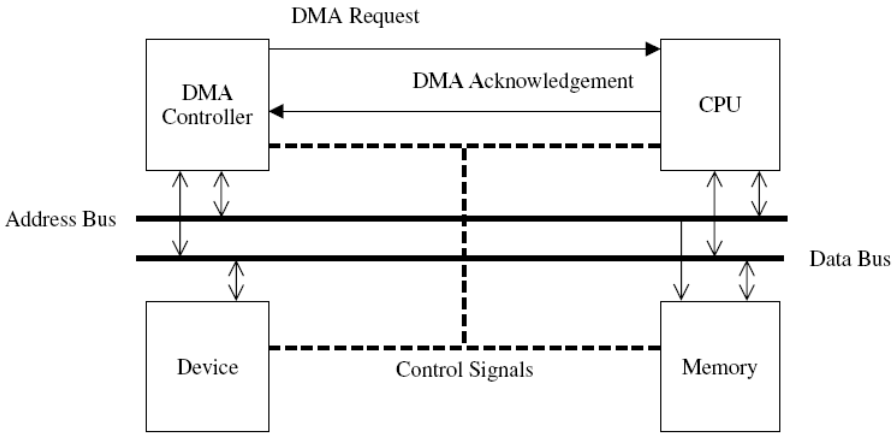


Figura 38: Controlador DMA compartiendo bus de CPU y memoria

Una transferencia DMA puede ser realizada en modo *burst* (donde el controlador DMA mantiene el control del bus hasta que todos los datos fueron transferidos) o modo de *ciclo único* (donde el controlador DMA libera el bus después de cada transferencia de una unidad de dato). El modo *burst* se usa en dispositivos de alta velocidad donde la transferencia de datos no puede ser detenida hasta que el bloque completo es transferido. El modo de ciclo único minimiza el tiempo que la DMA le quita el control del bus a la CPU, pero requiere que la secuencia de bus solicitud/respuesta se realice para cada uno de los datos del bloque a transferir, lo que puede resultar en la degradación del rendimiento del sistema. El modo de ciclo único se prefiere si el sistema no puede tolerar más de unos pocos ciclos de latencia adicional para interrupciones, o si los dispositivos periféricos tienen *buffers* para almacenar gran cantidad de datos causando que el controlador DMA tome el control del bus por una cantidad de tiempo excesiva.

3.2 Arquitecturas de bus off-chip

Los buses *off-chip* se usan para conectar SoCs con dispositivos externos, como por ejemplo chips de memoria SDRAM o DDR DRAM. A diferencia de los buses on-chip que hacen uso de implementaciones AND-OR o multiplexación unidireccional[5], los buses externos usan de preferencia implementaciones triestado bidireccionales para reducir el número de pines.

Existe una gran variedad de estándares de bus tales como S-100, PC-AT, Multibus, VME, PCI, PCI-X (PCI Extended), y PCIe (PCI Express). PCI ha sido el estándar más popular en su categoría, con casi toda la infraestructura de software de la industria de computadores sujeta al modelo de interconexión PCI. Sin embargo, el modelo de interconexión paralelo compartido tiene limitaciones inherentes tales como efecto *crosstalk*, alta carga capacitiva (y por lo tanto

5 Ver sección 2.4.3

retardos asociados), alta disipación de potencia, efectos de deslizamiento de señal o *signal skew* debido a las largas distancias cubiertas por el bus, lo que limita la máxima frecuencia de clock.

Para superar los problemas que incorporan las arquitecturas de bus paralelo, se han propuesto estándares de interconexión paralelos basados en *switch* tales como HyperTransport y RapidIO. Estos estándares usan conexiones punto a punto con un número de líneas reducido y frecuencias de clock más altas. Estas arquitecturas paralelas punto-a-punto basadas en switch resuelven los problemas de cargas capacitivas, velocidad, y confiabilidad de interconexiones paralelas. Sin embargo, no resuelven el problema de efecto crosstalk y signal skew.

Para operar a frecuencias de clock altas y superar los efectos de crosstalk han sido propuestos estándares de interconexión serial basados en switch, de los cuales destacan PCIe e Infiniband. Estos estándares usan una única señal para transmisión serial, capaz de alcanzar altas velocidades sin sufrir efectos de crosstalk. PCIe es uno de los estándares más dominantes gracias a su soporte del legado PCI.

3.3 El bus PCI

PCI (*Peripheral Component Interconnect*) es un bus sincrónico con buses de dirección/datos multiplexados y arbitración centralizada. Tiene versiones de 32 y 64 bits y frecuencias de clock de 33, 66 y 133 MHz. La versión más común es 32 bit @ 33 MHz (es decir, ancho de datos 32 bit operando a 33 Mhz).

Como su nombre lo indica es un estándar que describe como conectar los periféricos de un sistema en forma estructurada y controlada. Describe la forma en que los componentes del sistema se conectan eléctricamente y la forma en que deben comunicarse. En la figura 39 se muestra un ejemplo de un sistema típico basado en el bus PCI.

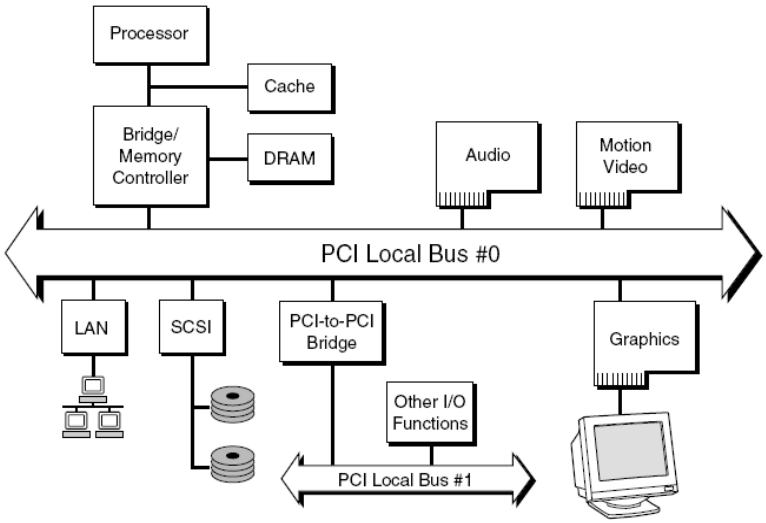


Figura 39: Ejemplo de sistema basado en el bus PCI

Dada la complejidad y tamaño del estándar PCI (compuesto por una colección de especificaciones), a continuación sólo se mencionan los conceptos más básicos y fundamentales para introducir al lector a la operación del bus PCI. Para más detalles se recomienda referirse a la especificación oficial de bus local PCI [15].

3.3.1 Fundamentos del protocolo PCI

En la tabla 4 se muestran las señales más importantes del bus para el caso 32 bit @ 33 MHz. La especificación PCI incluye otras señales para funciones de configuración, reporte de errores y manejo de potencia, entre otras. En la especificación PCI se denotan las señales *active-low* con el sufijo # (por ejemplo, la señal FRAME# es activada cuando ocurre una transición de nivel *high* a *low*).

Una transacción de bus PCI consiste en una fase de dirección seguida por una o más fases de datos. Es posible introducir estados de espera entre fases para sincronizar el iniciador (*initiator*) y el objetivo (*target*). El bus PCI soporta transacciones únicas y en modo burst.

En la figura 40 se muestra un ejemplo de transacción de lectura PCI. La transacción se inicia por el maestro (iniciador) activando su señal FRAME# en el ciclo 1. En este ciclo, el maestro también carga las líneas AD[31:0] con la dirección y las líneas C/BE# con el comando *memory read*. Luego el maestro activa su señal IRDY# en el ciclo 2 para indicar que está listo para recibir la primera palabra de datos. En el ciclo 2, mientras la señal TRDY# no sea activada por el esclavo ninguna transferencia puede ocurrir en el bus, es decir, el esclavo puede introducir estados de espera retardando la activación de TRDY#. Este ciclo inactivo es requerido en todas las transacciones de lectura para "dar vuelta el bus AD" o *AD bus turn around*, es decir, cambiar el agente que carga las líneas de bus AD (ahora el maestro ve el bus AD como entrada y no como salida). El esclavo provee los datos en el ciclo 3 y activa TRDY#. Como IRDY# y TRDY# están ambas activadas, ocurre la transferencia de la primera palabra de datos. En el ciclo 4, el esclavo provee la segunda palabra de datos manteniendo activada TRDY#. Sin embargo, el maestro no está listo (IRDY# desactivada), y la transferencia debe esperar hasta el ciclo 5. Durante el ciclo 5, el maestro activa IRDY# permitiendo completar la segunda trasferencia. Además desactiva FRAME# para indicar que ésta será la última transferencia de datos de la transacción. En el ciclo 6, tanto FRAME# como IRDY# están desactivadas indicando el fin de la transacción, y el árbitro puede conceder el bus al siguiente maestro para que inicie una transacción en el ciclo 7.

CLK	*Bus clock*. Todas las señales se muestrean en el flanco positivo de la señal CLK
AD[31:0]	*32 bit Address/Dada bus*. Estas líneas son cargadas con dirección o datos dependiendo si se trata de una fase de dirección o datos respectivamente.
C/BE[3:0]	*Command/Byte Enable*. Estas líneas son cargadas con el comando de bus (por ejemplo, lectura a memoria) durante una fase de dirección, y con señales de Byte Enable durante una fase de datos para indicar los bytes donde se encuentra la información.
FRAME#	*Transaction frame*. El maestro inicia una transacción activando su señal FRAME#. El primer ciclo de clock en donde la señal FRAME# está activada es la fase de dirección. La última fase de datos se inicia cuando se desactiva FRAME#.
IRDY#	*Initiator ready*. Indica cuando el iniciador está listo para transferir un dato. La transferencia se efectúa cuando tanto el iniciador como el objetivo están listos (es decir, cuando las señales IRDY# y TRDY# se encuentran ambas activadas).
TRDY#	*Target ready*. Indica cuando el objetivo de la transacción está listo para transferir datos.
REQ#	*Bus request*. Un maestro activa su señal REQ# para solicitar acceso al bus como iniciador.
GNT#	*Bus grant*. Un árbitro central muestrea la señales REQ# provenientes de cada dispositivo maestro presente en el bus y concede el bus a uno solo activando su señal GNT#. Esta señal indica que el dispositivo será el iniciador cuando la transacción en curso termine.

Tabla 4: Señales principales del bus PCI

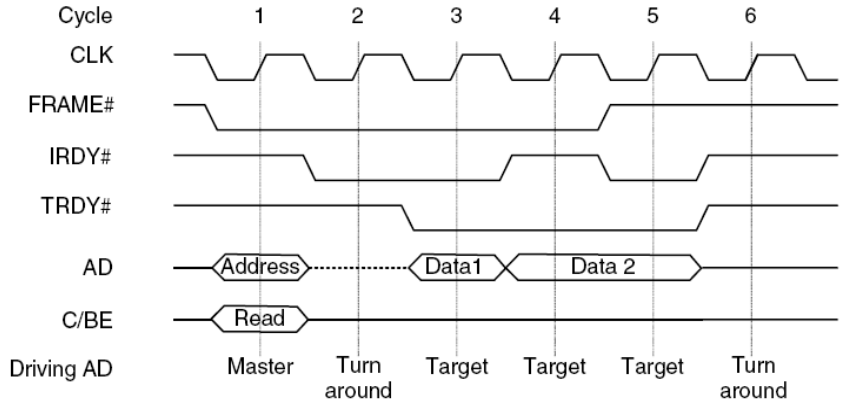


Figura 40: Diagrama de timing para transacción de lectura PCI

Una transacción de escritura es casi idéntica a la transacción de lectura anterior, con la excepción de que el iniciador es el que carga el bus AD durante las fases de datos, y el comando de bus es distinto (por ejemplo, *memory write*). Luego, para una transacción de escritura no se requiere un ciclo inactivo para *bus turn around* después de la fase de dirección. Esto se debe a que el iniciador carga tanto la dirección como los datos.

PCI no soporta modo de transferencia no atómico o *split*[6]. Cada transacción de lectura incluye tanto el mensaje de solicitud del iniciador al objetivo como el mensaje de respuesta desde el objetivo hacia el iniciador. Sin embargo, PCI incluye un mecanismo para abortar y reintentar una transacción para evitar el bloqueo de bus durante una operación de latencia muy larga. Un

6 Ver sección 2.4.8.4

esclavo puede capturar una solicitud y luego abortar la transacción activando la señal STOP# en vez de TRDY#. Luego el maestro reintenta la transacción en un instante posterior, con la posibilidad de que el esclavo haya completado su operación y pueda responder a los datos solicitados. La especificación PCI llama a este tipo de llamadas como transacción retardada (*delayed transaction*). Una similitud de este tipo de transferencia con transacción split es que permite otras transacciones de bus durante operaciones de retardo largo. Sin embargo, este mecanismo es considerablemente menos eficiente que una transacción split y está pensado para manejar casos excepcionales.

3.3.2 Arbitración

Para minimizar la latencia, el enfoque de arbitración PCI es basado en acceso en vez de basado en división de tiempo, es decir, un maestro requiere de un proceso de arbitración para ganar el acceso al bus. La arbitración es "oculta", lo que significa que ocurre durante el acceso previo de tal forma que no se consumen ciclos de bus en arbitración, excepto cuando el bus está en estado inactivo.

En la figura 41 se muestra un ejemplo típico de arbitración PCI *pipelined*. Durante el ciclo 1 los maestros 1 y 2 solicitan el bus activando sus señales REQ1# Y REQ2# respectivamente. El árbitro central decide otorgar el bus al maestro 1 y activa su señal GNT1# en el ciclo 2. El maestro 1 recibe el permiso e inicia la transacción activando FRAME# en el ciclo 3. El árbitro desactiva GNT1# y activa GNT2# en el ciclo 4 para indicar al maestro 2 que obtendrá el bus cuando la transacción en curso termine. En el ciclo 5 se señaliza el fin de transacción con la desactivación de FRAME# e IRDY#. El maestro 2 detecta el fin de transacción e inicia una nueva en el ciclo 6 activando FRAME#.

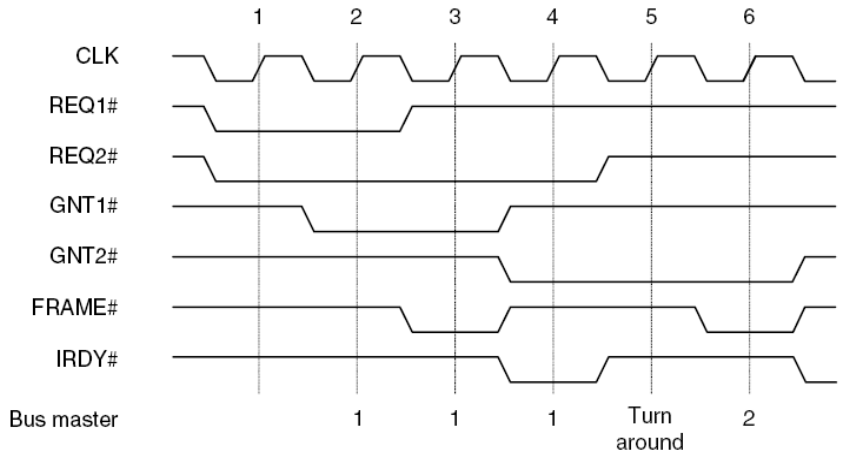


Figura 41: Diagrama de timing para arbitración PCI

3.3.3 Direccionamiento y configuración

PCI define tres espacios de dirección: espacio de memoria, espacio I/O y espacio de

configuración. Los dos primeros se usan para direccionamiento de periféricos organizados en un mapa de memoria. El espacio de configuración es usado por el sistema operativo para inicializar y configurar dispositivos PCI. La especificación PCI recomienda que todos los dispositivos sean mapeados en espacio de memoria si es posible, dada la inherente limitación del espacio I/O (altamente fragmentado en sistemas PC).

Todo dispositivo PCI (incluyendo PCI-to-PCI bridges) tiene un espacio de configuración de 256 bytes, donde el encabezado compuesto por los primeros 64 bytes (figura 42) contiene un conjunto predefinido de registros para identificación y configuración. El espacio restante es dependiente del dispositivo.

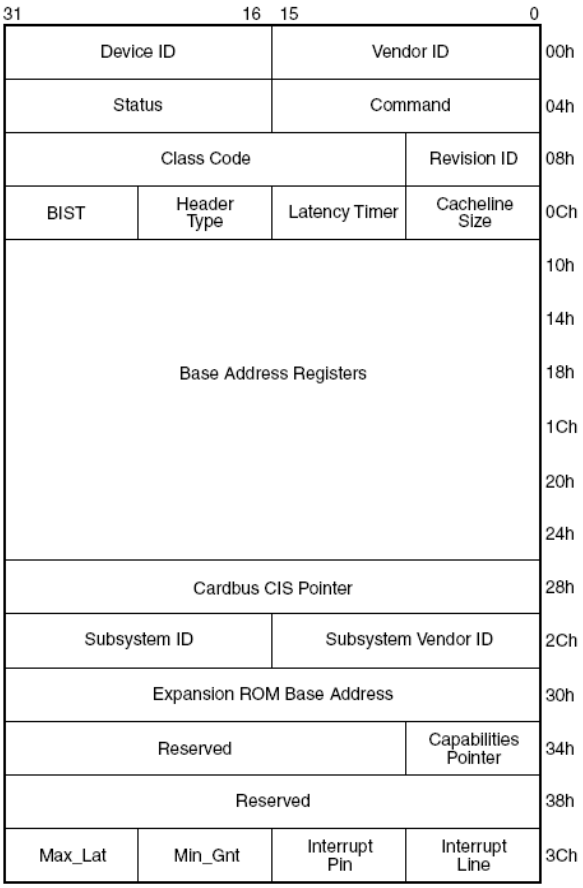


Figura 42: Encabezado del espacio de configuración PCI

Una de las funciones más importantes dentro del estándar PCI es la habilidad de reubicar los dispositivos PCI en el espacio de memoria. Cuando el sistema se inicia, un software independiente de los dispositivos examina la cantidad de agentes en el bus y los requerimientos de memoria de cada uno. Con esta información el software puede construir un mapa de memoria consistente tal que todos los dispositivos pueden coexistir sin conflictos. Este mapa se configura por medio de los BARs o *Base Address Registers*, que son programados con la dirección base asignada a cada dispositivo en el mapa de memoria. Una vez realizada esta operación el sistema

operativo puede cargar los drivers de los dispositivos presentes en el sistema.

Un dispositivo es seleccionado como objetivo de un ciclo de configuración activando su señal IDSEL. Por ejemplo, en la arquitectura x86 cada conector PCI o slot tiene su pin IDSEL conectado a una de las líneas del bus AD[31:11] como se muestra en la figura 43. Durante la primera fase de un ciclo de configuración, el tipo de ciclo se identifica con el comando adecuado (*Configuration Read* o *Configuration Write*) presente en C/BE[3:0], mientras que en las líneas AD[31:11] sólo se activa el bit conectado a la señal IDSEL del slot objetivo y las líneas restantes (AD[10:0]) contienen información sobre el registro destino de la operación dentro del espacio de configuración.

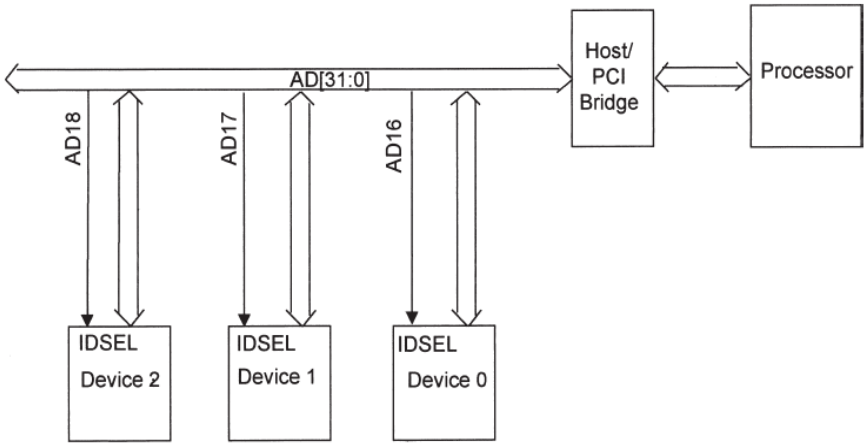


Figura 43: Ejemplo de conexión de señales PCI IDSEL a host bridge

3.3.4 Características eléctricas

La mayoría de las especificaciones eléctricas del estándar PCI están orientadas a reducir el consumo de potencia. PCI se basa en tecnología CMOS y por lo tanto las corrientes contínuas en estado estable son mínimas, siendo la mayoría de las corrientes dc aquellas que fluyen por resistencias pull-up. El protocolo está diseñado tal que los pines del bus no tengan libertad de flotar, para evitar el consumo innecesario de potencia por oscilación.

A continuación se introduce la técnica de propagación *reflected wave switching*, que es uno de los conceptos más importantes relacionados con la reducción del consumo de potencia usados por el estándar PCI.

Arquitecturas de bus tradicionales tales como Unibus y Qbus usan la técnica de *incident wave switching*, donde se enfatiza la necesidad de una apropiada terminación de todas las líneas del bus para prevenir indeseadas reflexiones de onda. Cada línea en un bus de placa madre es una línea de transmisión con una cierta impedancia característica (del orden de 120 Ohm). Si los extremos de una línea no tienen terminación, un pulso que viaja por la línea será reflejado de vuelta desde el fin causando interferencia indeseada. La solución en este caso es terminar ambos extremos de una línea con la impedancia característica. En este esquema cada terminación de

impedancia se implementa típicamente con un divisor de impedancia que consume entre 10 y 20 mA, y considerando un bus que usa del orden de 50 líneas el consumo máximo total es del orden de 1 A. Considerando un ambiente de señal que opera a 5V, la pérdida de potencia por las terminaciones de impedancia es del orden de 5 W. Además dada la gran cantidad de drivers presentes en el bus alternando entre encendido y apagado determina grandes peaks en las líneas de potencia, sin mencionar el efecto de crosstalk entre líneas de bus.

PCI usa la técnica de *reflected wave switching* que elimina las terminaciones de bus y se aprovecha del frente de onda reflejada. Un driver de bus PCI está diseñado para cargar la línea a la mitad del voltaje (por ejemplo 1.5 V). A medida que la onda se propaga por la línea, es insuficiente para producir niveles lógicos válidos en los receptores que encuentra a su paso. Cuando el frente de onda alcanza el fin de línea, ésta es reflejada de vuelta doblada en magnitud (3.3 V). Entonces los receptores pueden ver niveles válidos con el nuevo frente de onda en dirección contraria, que serán registrados con el siguiente flanco positivo del clock. Finalmente el frente de onda reflejado es absorbido por la pequeña impedancia del driver. La ventaja del método es que reduce el tamaño del driver y corta a la mitad el requerimiento de corriente.

En la figura 44 se muestran los parámetros de timing que permiten caracterizar el proceso de reflexión del frente de onda, para una transición de *low* a *high* en el inicio de una línea. En el punto A el driver carga la línea a medio nivel, y en el punto B el frente de onda es reflejado y dobla la magnitud del voltaje. Finalmente el nivel lógico se registra en el flanco positivo indicado por el punto C. Tval es el tiempo que toma la salida del driver en cargar el pin hasta su nivel lógico final. Tprop (*propagation delay*) es el tiempo que toma el frente de onda en llegar al final de la línea, reflejarse (y por lo tanto dobla el voltaje), y viajar de vuelta por la línea. Tsu (*setup time*) es el tiempo de establecimiento de señal antes del próximo flanco positivo del clock.

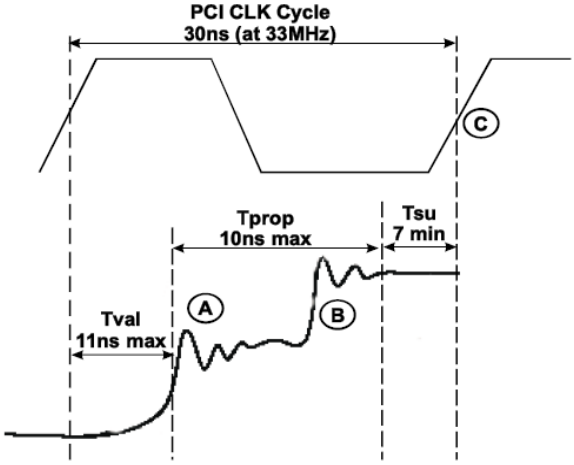


Figura 44: Diagrama de timing para transición de señal PCI usando reflected wave switching

La única señal PCI que no usa *reflected wave switching* es el clock. Para un bus PCI operando a 33 MHz, el ciclo de bus es de 30 ns. PCI requiere un deslizamiento de la señal CLK (medido entre 2 pines de clock de dos dispositivos PCI) de 2 ns como máximo.

Según el estándar PCI, el sistema host (donde se encuentra el bus PCI 0) define el ambiente de señal, que puede ser 5V o 3.3V. Por lo tanto un slot PCI está diseñado para trabajar en un solo ambiente de señal (5V o 3.3V), es decir, no existen en la especificación PCI los slots "universales" o duales. Sin embargo, una tarjeta PCI puede estar diseñada para funcionar en ambos ambientes de señal (tarjeta universal). Actualmente la especificación PCI ha migrado por completo al ambiente de señal 3.3V.

3.4 Implementación de dispositivos PCI en SoPCs

Típicamente los dispositivos PCI son implementados por medio de un ASIC que posee interfaz PCI, o usando dispositivos de lógica programable como CPLDs o FPGAs. Para el caso de SoPCs, la forma más común de implementar dispositivos PCI es por medio del uso de PCI bridges de propiedad intelectual (IP cores). Generalmente los PCI bridges están diseñados para operar en modo Host bridge (es decir, como bridge entre un bus de host específico y el bus PCI 0), y tienen la ventaja de ser reusados y portados a diferentes plataformas (CPLD, FPGA). Otra ventaja es que presenta el uso de PCI bridges de propiedad intelectual es que pueden ser adaptados a distintas arquitecturas de comunicación mediante un IP wrapper apropiado[7]. En general un PCI bridge IP core viene acompañado de un completo *test bench* o modelo no sintetizable para simulación que permite exploración y verificación del sistema. Además, para evitar el trabajo de diseñar e implementar un árbitro PCI, existen IP cores que incorporan lógica de arbitración PCI embebida. También existen IP cores de arbitración PCI disponibles para una gran variedad de arquitecturas.

A continuación se describen las principales características del PCI bridge IP core de Opencores.

3.5 Caso de estudio: Opencores PCI bridge IP core

El PCI bridge de Opencores [16] es un IP core construido a partir de la especificación PCI revisión 2.2 que implementa un bridge entre el bus PCI y el estándar de comunicación Wishbone[8] para SoC. Se compone de un conjunto de archivos Verilog sintetizables que proveen al usuario la capacidad de parametrizar una variedad características y funcionalidades, tales como operación como host o target bridge, tamaño y profundidad de fifos, valores prefinidos en espacio de configuración, etc.

A continuación se describen las principales características del PCI bridge de Opencores. Para más detalles se recomienda referirse a los documentos oficiales de Opencores [16], [17], [18] y [19].

3.5.1 Características principales

- Dominios de clock independientes para ambos lados del bridge (PCI y Wishbone).

7 Ver sección 2.7.1.3
8 Ver sección 2.6

- Provee dos implementaciones parametrizables posibles: HOST (usado para *host bridging* con Wishbone como host bus) y GUEST (para *expansion bus bridging* con Wishbone como bus de expansión).
- Compatible con Wishbone rev. B.1 y B.3 (señales CTI y BTE), con interfaces maestro y esclavo por separado.
- Compatible con especificación PCI rev. 2.2, 32-bit, 33 o 66 MHz.
- Soporta operación en modo *zero wait state burst*[9].
- Cuatro FIFOs sintetizables de puerto dual, con profundidad parametrizable.
- Número parametrizable de imágenes programables (1 imagen por defecto, máximo 6) con capacidad de traducción de dirección, y tamaño de imagen de 4KB a 1GB.
- Mapeo de espacio de dirección de imágenes programable (espacio I/O o de memoria).
- Realización de requerimientos de ordenamiento de transacciones PCI en forma interna (el bridge usa *posted writes* y *delayed reads* en cada dirección).
- Soporta *single delayed transaction* en cada dirección.
- Espacio de configuración expandible, implementado para funcionalidades de software programables adicionales del core.
- Uso transparente de comandos PCI, con apropiada configuración de registros via software.
- Funciones soportadas por unidad *PCI initiator*:
  - Comandos *Memory Read*, *Memory Read Line*, *Memory Read Multiple* y *Memory Write*
  - Comandos *IO Read* e *IO Write*
  - Comandos *Configuration Read* y *Configuraton Write*
  - Comando *Interrupt Acknowledge*
  - Soporta *Linear burst ordering*
  - Bus parking
  - Operación transparente de interfaz Wishbone, con apropiada configuración de registros por software
- Funciones soportadas por unidad *PCI target*:
  - Comandos *Memory Read*, *Memory Read Line*, *Memory Read Multiple*, *Memory Write* e *Invalidate*
  - Comandos *IO Read* e *IO Write*
  - Comandos *Configuration Read* y *Configuration Write*

9 Ver sección 2.4.8.3

- Soporta *Linear burst ordering*
- Soporta respuesta a transacciones *Fast Back-to-Back*

3.5.2 Arquitectura

El PCI bridge de Opencores se compone de dos grandes unidades funcionales (figura 45): la unidad *PCI target* y la unidad *esclavo Wishbone*. Cada una contiene las funciones necesarias para soportar la conversión de transacciones desde el lado PCI hacia el lado Wishbone, y del lado Wishbone hacia el lado PCI respectivamente. La unidad PCI target actúa como target en el lado PCI y como maestro en el lado Wishbone, mientras que la unidad esclavo Wishbone actúa como iniciador en el lado PCI y como esclavo en el lado Wishbone. Ambas unidades operan en forma independiente.

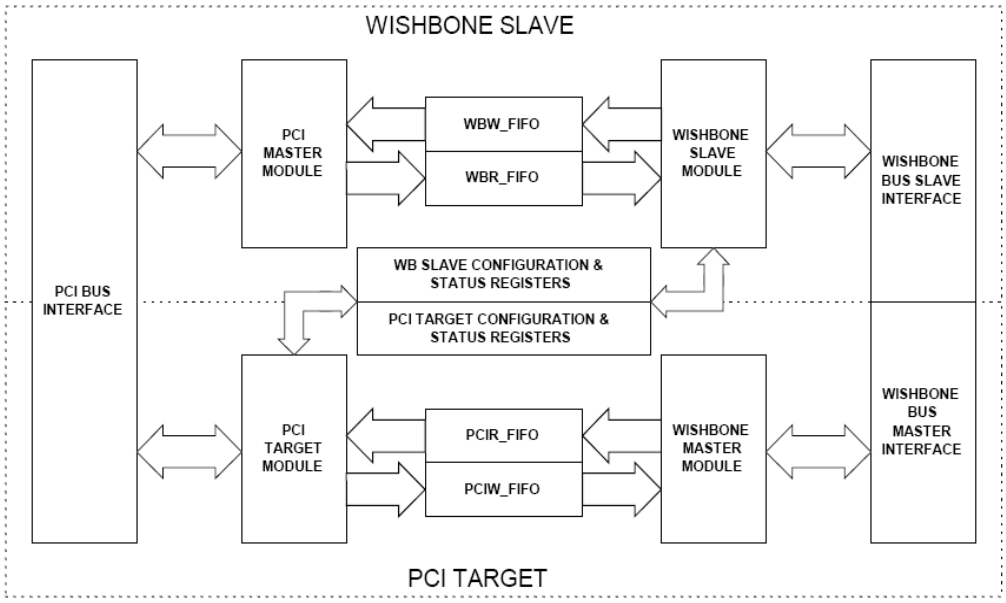


Figura 45: Arquitectura del Opencores PCI bridge IP core

A continuación se describen brevemente los principales bloques funcionales del PCI bridge:

- *Wishbone Slave module*:

La interfaz esclavo Wishbone es compatible con la especificación Wishbone Rev. B.1. Acepta accesos que caen en alguna región de las imágenes implementadas (Imagen 0 por defecto, Imágenes 1 a 5 opcionales), y responde según la configuración de la imagen respectiva. Las imágenes se pueden configurar vía software para realizar traducción de direcciones y/o usar comandos de accesos a memoria especiales cuando se solicita una transferencia burst. Las escrituras se manejan como *posted* y las lecturas como *delayed*. El espacio de configuración se puede acceder a través de la interfaz esclavo Wishbone con capacidad de lectura/escritura para implementación HOST bridge y sólo lectura para GUEST bridge. Los ciclos de configuración se realizan como *delayed transactions*, independientemente si se trata de una solicitud de lectura o

escritura. El bridge permite solo una solicitud de *delayed transaction* a la vez, en una misma dirección.

- *Wishbone Write FIFO*:

El módulo *Wishbone Write FIFO* almacena la información de dirección y datos cargadas por el maestro Wishbone externo que realiza una escritura a una dirección que cae dentro del espacio de una de las imágenes implementadas. En el lado PCI, la máquina de estado del maestro PCI toma los datos y completa la escritura en el bus PCI. La profundidad de la FIFO es parametrizable vía HDL.

- *Wishbone Read FIFO*:

El módulo Wishbone Read FIFO almacena datos de delayed reads que ya han sido completadas en el bus PCI. La máquina de estados del maestro PCI realiza una lectura cuando es solicitada, almacena los datos en la FIFO, y provee los datos para el maestro Wishbone externo cuando repite la solicitud a través de la interfaz esclavo Wishbone. Su profundidad también es parametrizable.

- *PCI Master module*:

El módulo *PCI master* es un iniciador PCI compatible con la especificación PCI Rev. 2.2. Es responsable de completar las *posted writes* almacenadas en la *Wishbone Write FIFO* (WBW_FIFO) o servir *delayed reads*, lecturas/escrituras de configuración o solicitudes de *interrupt acknowledge*.

- *PCI Target module*:

La máquina de estados del módulo *PCI target* acepta ciclos de configuración desde el bus PCI y ciclos que caen dentro del espacio de direcciones de una imagen. Los ciclos de configuración proveen acceso al *header Type 0* del espacio de configuración. El espacio de configuración extendido es accesible con accesos a memoria de lectura/escritura a través de una imagen especial de 4KB mapeada en memoria. Se permite acceso de sólo lectura al espacio de configuración PCI para HOST y lectura/escritura para GUEST. Todos los accesos excepto los de configuración se pasan a la interfaz maestro Wishbone. Cada imagen implementada puede ser configurada para realizar traducción de dirección y/o para configurar la máquina de estado Wishbone master para realizar transferencias burst aún cuando la transferencia PCI solicitada no use comandos optimizados de acceso a memoria.

- *PCI Write FIFO*:

El módulo *PCI Write FIFO* almacena información de dirección y datos provenientes de un iniciador PCI externo que realiza un comando de escritura a una dirección que cae dentro del espacio de una de las imágenes. En el otro lado, la máquina de estado del Wishbone master saca datos y completa las escrituras en el bus Wishbone. La profundidad es parametrizable.

- *PCI Read FIFO*:

El módulo *PCI Read FIFO* almacena datos de *delayed reads* que ya han sido completadas en el bus Wishbone. La máquina de estados del Wishbone master realiza una lectura cuando se solicita, almacena datos en la FIFO, y provee datos para el iniciador PCI externo cuando repite la solicitud a través del módulo PCI target. La profundidad es parametrizable.

- *Wishbone Master module*:

El módulo Wishbone master responde a solicitudes desde el bus PCI y las pasa a través del bridge. Cuando se usan comandos de acceso a memoria optimizados en el bus PCI, la interfaz Wishbone master puede mejorar el rendimiento guardando burst largas para lecturas almacenándolas en la FIFO de lectura. Las operaciones de escritura burst son aceptadas desde el bus PCI y almacenadas en la Write FIFO y también son realizadas como burst writes en el bus Wishbone.

3.5.3 Espacio de configuración

Se proveen 4KB de espacio de configuración (figura 46) para funcionalidades del bridge controlables por software (vía interfaz de esclavo Wishbone). Además del *PCI header Type 0* hay registros especiales para control de las imágenes, traducción de direcciones, control de interrupciones, reporte de errores, etc.

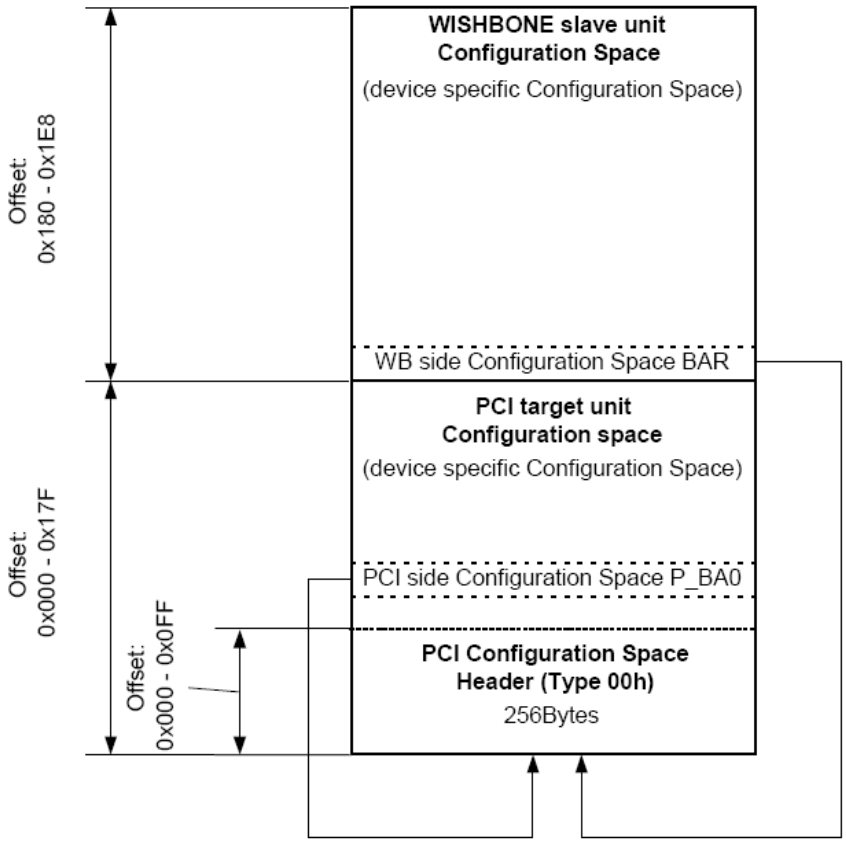


Figura 46: Espacio de configuración del PCI bridge

Si el bridge es implementado como HOST, la unidad Wishbone Slave tiene acceso exclusivo a este espacio, mientras que la unidad PCI Target solamente puede tener acceso de lectura. Si el bridge se implementa como GUEST, el acceso exclusivo lo tiene la unidad PCI

Target y la unidad Wishbone slave solamente puede tener acceso de lectura.

3.5.4 Dominios de clock

El PCI bridge tiene dos dominios de clock, uno para el bus PCI y el otro para el bus Wishbone. El cruce de dominios de clock se realiza mediante la lógica de control de las FIFOs, las cuales son simétricas (no hay diferencia entre las cuatro FIFOs asíncronas que incorpora el sistema), y por lo tanto no existen condiciones que establezcan que clock debe tener mayor frecuencia.

3.5.5 Interrupciones

Ciertas condiciones tales como error de paridad, *Target Abort* o señales similares durante *posted writes*, pueden disparar interrupciones. El espacio de configuración provee de un mecanismo para habilitar estas interrupciones y un registro de estatus para reportar interrupciones. El bridge además rutea y reporta interrupciones disparadas en los buses. La implementación de HOST bridge dispara interrupciones (si están habilitadas) en el bus Wishbone.

4 Implementación

4.1 Desarrollo de IP wrapper para el Opencores PCI Bridge IP core

4.1.1 Descripción del problema

La plataforma ISIS está basada en un dispositivo FPGA Cyclone II de Altera, y por lo tanto está orientada a la implementación de SoPCs bajo el paradigma de PBD (*Platform Based Design*), es decir, el dispositivo FPGA Cyclone II soporta en forma nativa sistemas en formato *sof* (s*ystem on a file*) diseñados con IP cores y herramientas propiedad intelectual de Altera, como por ejemplo el software de generación de interconexión SOPC Builder (parte de la suit de diseño Quartus II). En particular, en el marco de este trabajo se implementa un sistema basado en el procesador Nios II con arquitectura de comunicación Avalon System Interconnect Fabric.

Como hardware controlador de bus PCI se escoge al IP core PCI Bridge de Opencores, con interfaces de bus maestro y esclavo Wishbone en el lado host. Dado que el IP core de Opencores tiene interfaces de comunicación Wishbone incompatibles con la arquitectura de comunicación Avalon System Interconnect Fabric, se requiere el desarrollo de un *encapsulador de propiedad intelectual* o *IP wrapper* que encapsule el core PCI Bridge superponiendo el protocolo de bus Avalon sin afectar su funcionalidad[10].

A continuación se describe la implementación del IP wrapper, con detalle en la metodología desarrollada y la implementación de la solución.

4.1.2 Requerimientos

La lógica de encapsulación del core PCI Bridge (módulo pci_bridge32.v) debe cumplir los siguientes requerimientos:

1. Encapsulación del core PCI Bridge en módulo de alto nivel con interfaces maestro y esclavo Avalon independientes, lógica de conversión para transacciones maestro Wishbone a esclavo Avalon y maestro Avalon a esclavo Wishbone, e instancias de buffers triestado para interfaz PCI.
2. Interfaz esclavo Avalon:
    - Soporte de ciclos de lectura y escritura Avalon con bus de datos de 32 bit
    - Bus de dirección para soportar al menos 4 MB de espacio de memoria (es decir, con ancho igual o superior a 20 bits)
    - Soporte de selección de bytes en bus de datos
    - Soporte de introducción de estados de espera
    - Soporte de interrupciones y señal de reset

10 Ver sección 2.7.1.3

3. Interfaz maestro Avalon:
    - Soporte de ciclos de lectura y escritura Avalon con buses de datos y dirección de 32 bit
    - Soporte de selección de bytes en bus de datos
    - Soporte de solicitud de estados de espera
4. Interfaz de bus PCI:
    - Interfaz PCI 3.3V, 32 bit @ 33 MHz
    - Implementación de señales externas bidireccionales con buffers triestado en arreglo bidireccional, con control de habilitación de salida (*output enable*)
    - Señales externas no bidireccionales implementadas con buffers triestado unidireccionales

4.1.3 Metodología de implementación

Se implementa la lógica de encapsulación en un módulo de alto nivel llamado pci_top.v (en HDL Verilog) como se muestra en diagrama de bloques de la figura 47. El módulo contiene una instancia del módulo pci_bridge32 (con interfaces maestro y esclavo Wishbone, e interfaz PCI), interfaces de bus maestro y esclavo Avalon, lógica de pegamento entre interfaces Avalon y Wishbone, e instancias de buffers triestado para señales externas bidireccionales PCI.

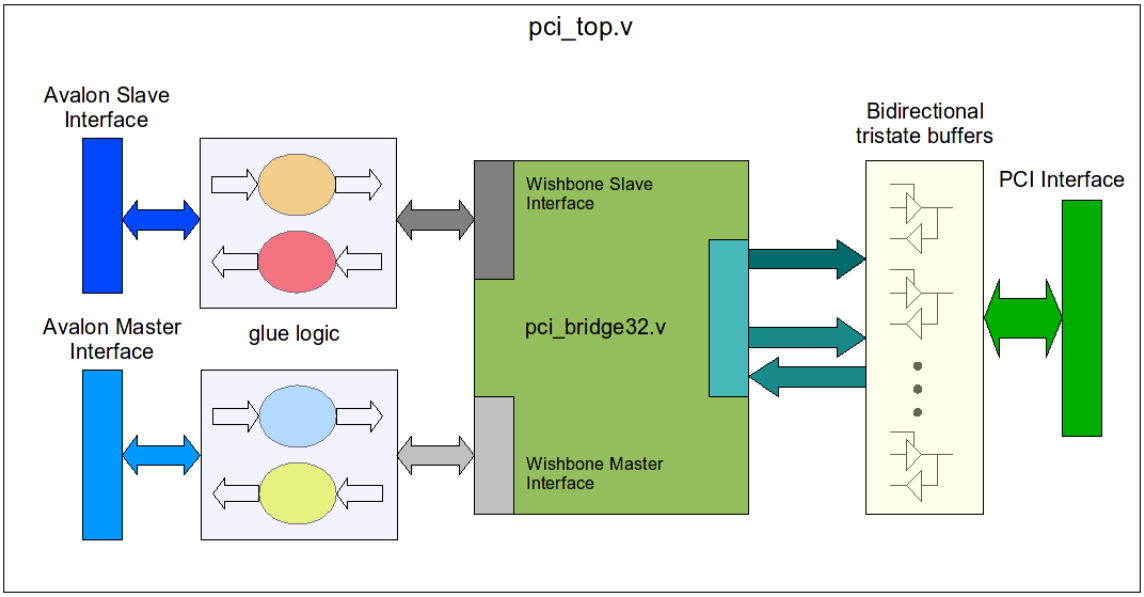


Figura 47: Diagrama de bloques módulo pci_top.v

La lógica de pegamento entre las interfaces de bus Avalon y Wishbone se estructura en dos bloques principales, uno para la conversión de transacciones iniciadas por maestros Avalon a un

esclavo Wishbone, y otro para la conversión de transacciones iniciadas por un maestro Wishbone hacia un esclavo Avalon. Dado que ambos estándares de comunicación no manejan señales bidireccionales, cada bloque de conversión de protocolo se subdivide en dos bloques de lógica combinacional. Un subbloque es para señales activadas por el System Interconnect Fabric que deben ser convertidas e interpretadas por una máquina de estado Wishbone, y otro subbloque para señales activadas por una máquina de estado Wishbone que deben ser convertidas e interpretadas por el System Interconnect Fabric. Las máquinas de estado Wishbone se encuentran embebidas en el módulo pci_bridge32, y son compatibles con la especificación Wishbone Rev. B.1.

En la tabla 5 se especifican las interfaces de bus del módulo pci_top.v, con el detalle de las señales de cada puerto y sus parámetros. El sufijo _n denota señales de lógica *active low*.

Interfaz PCI

bus PCI	Dirección	bits
pci_CLK_33_mhz	input	1
pci_AD	inout	32
pci_CBE_n	inout	4
pci_RST_n	output	1
pci_INTA_n	inout	1
pci_REQ_n	output	1
pci_GNT_n	input	1
pci_FRAME_n	inout	1
pci_IRDY_n	inout	1
pci_IDSEL	output	1
pci_DEVSEL_n	inout	1
pci_TRDY_n	inout	1
pci_STOP_n	inout	1
pci_PAR	inout	1
pci_PERR_n	inout	1
pci_SERR_n	output	1

Interfaz esclavo Avalon

Avalon slave	Dirección	bits
av_s_irq	output	1
av_s_address	input	9
av_s_readdata	output	32
av_s_writedata	input	32
av_s_byteenable	put	4
av_s_write	input	1
av_s_read	input	1
av_s_chipselect	input	1
av_s_waitrequest	output	1

Interfaz maestro Avalon

Avalon master	Dirección	bits
av_m_address	output	32
av_m_readdata	input	32
av_m_writedata	output	32
av_m_byteenable	output	4
av_m_write	output	1
av_m_read	output	1
av_m_waitrequest	out	1

Tabla 5: Especificación de puertos del módulo pci_top.v

El módulo pci_top.v se sintetiza usando el software Quartus II de Altera. La herramienta de análisis y síntesis genera una *netlist* para el módulo encapsulador pci_top.v que ocupa 11 celdas lógicas combinacionales o *LC Combinationals,* y 0 celdas lógicas de registros o *LC Registers*. En efecto, como se expone anteriormente la lógica de pegamento es puramente combinacional (no requiere de máquinas de estado y minimiza el retardo asociado a la conversión de protocolos), y por lo tanto el uso de registros es cero.

4.2 Implementación de lógica de arbitración PCI

Dado que el IP core PCI Bridge no incorpora lógica de arbitración, es responsabilidad del usuario implementar la lógica necesaria para administrar permisos cuando múltiples maestros quieren hacer uso del bus en forma simultánea. A continuación se describe la implementación de un módulo de arbitración PCI para administrar permisos de acceso para dos maestros: el maestro embebido en el core PCI Bridge y un maestro PCI externo presente en el bus a través del conector PMC de la plataforma ISIS.

4.2.1 Requerimientos

1. El árbitro debe administrar dos maestros acorde a la especificación de bus PCI: el maestro del core PCI Bridge (embebido en el dispositivo FPGA), y un maestro PCI externo (presente en el bus a través del conector PMC).
2. Después de la condición de *reset*, el árbitro debe estacionar el bus en el dispositivo PCI Bridge (soporte de *bus parking*).
3. Implementación de señales externas con buffer triestado.
4. Las 4 reglas de arbitración expuestas en la especificación PCI 2.2, sección 3.4.1 son:
   - El algoritmo de arbitración debe ser justo, para evitar *deadlocks.*
   - Si GNT# es desactivada y FRAME# es activada en el mismo clock, se inicia una referencia válida.
   - Una señal GNT# puede ser desactivada en el mismo clock que otra señal GNT# es activada, sólo cuando el bus no está en estado inactivo o *idle*. Si el bus está en estado idle debe insertarse un clock con todas las señales GNT# desactivadas para evitar contención en las líneas AD[31:0] y PAR.
   - Cuando se desactiva la señal FRAME#, la señal GNT# puede desactivarse en cualquier instante.

4.2.2 Metodología de implementación

El *test bench* del core PCI Bridge incorpora una serie de modelos y casos de test para estimular transacciones a través del PCI Bridge, entre los cuales se encuentran monitores de bus, dispositivos comportamentales PCI y Wishbone, y un árbitro PCI (módulo pci_blue_arbiter.v). Éste árbitro PCI tiene 4 pares REQ#/GNT# para dispositivos externos y un par REQ#/GNT# para el dispositivo interno (PCI host bridge).

El módulo pci_blue_arbiter.v está diseñado para la arbitración de modelos comportamentales de dispositivos PCI (es decir, no sintetizables), y por lo tanto no se puede implementar directamente en el SoPC objetivo. Se requiere lógica de encapsulación para proveer una interfaz apropiada y capturar las señales de bus que requiere el algoritmo de arbitración.

Se implementa el módulo pci_arbiter_top.v que encapsula al módulo pci_blue_arbiter.v, incorporando lógica de conversión de señales *active high* a *active low*, y un buffer triestado para

señal GNT# externa. Además se muestrean las señales REQ# interna, REQ# externa (presente y anterior), IRDY# (presente y anterior), y FRAME# (presente y anterior). En la figura 48 se muestra el diagrama RTL obtenido de la síntesis del módulo pci_arbiter_top.v. Los 3 registros que se observan en la figura 48 permiten capturar la información del estado de las señales en el ciclo de bus anterior (denotadas con el sufijo _prev), requerida por la máquina de estados del módulo pci_blue_arbiter.v.

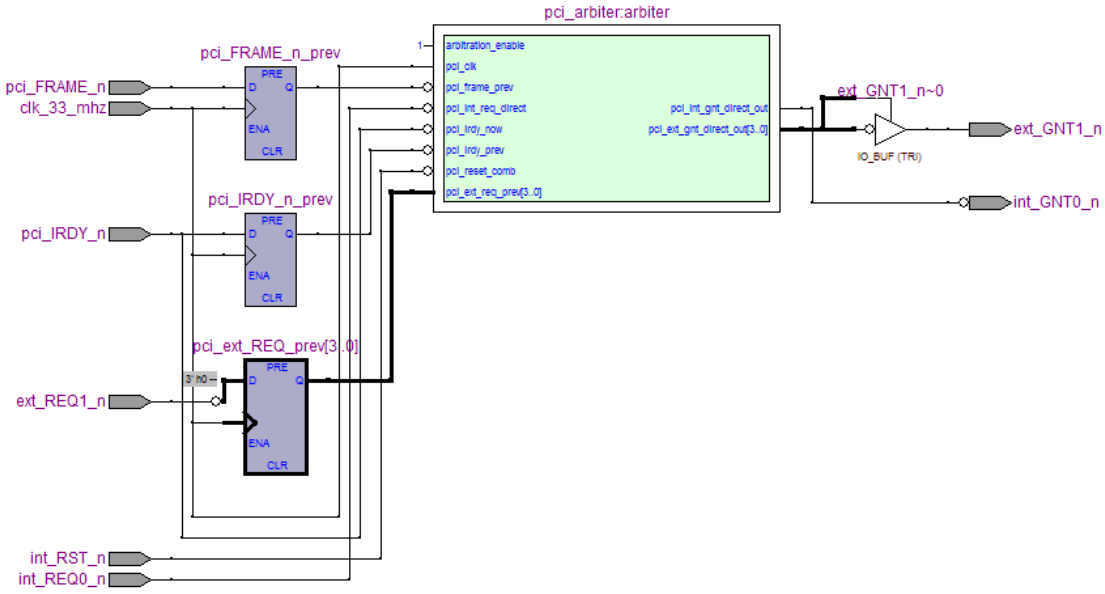


Figura 48: Diagrama RTL del módulo pci_arbiter_top.v

4.3 Implementación de core PCI Bridge en SoPC basado en procesador Nios II

4.3.1 Descripción del sistema

Se implementa el sistema de figura 49 sobre el dispositivo FPGA Altera Cyclone II EP2C20F484C6 de la plataforma ISIS. Mediante la herramienta SOPC Builder se genera un SoPC que contiene el procesador Nios II, el core PCI Bridge configurado como Host, controlador de memoria SDRAM, controlador de memoria FLASH, controlador DMA para transporte de datos entre memoria y PCI Bridge, controlador de chip Ethernet, JTAG, UART, entre otros periféricos. El clock del sistema es de 100 MHz, generado por un módulo PLL de la FPGA a partir del oscilador de 24 MHz presente en la plataforma ISIS.

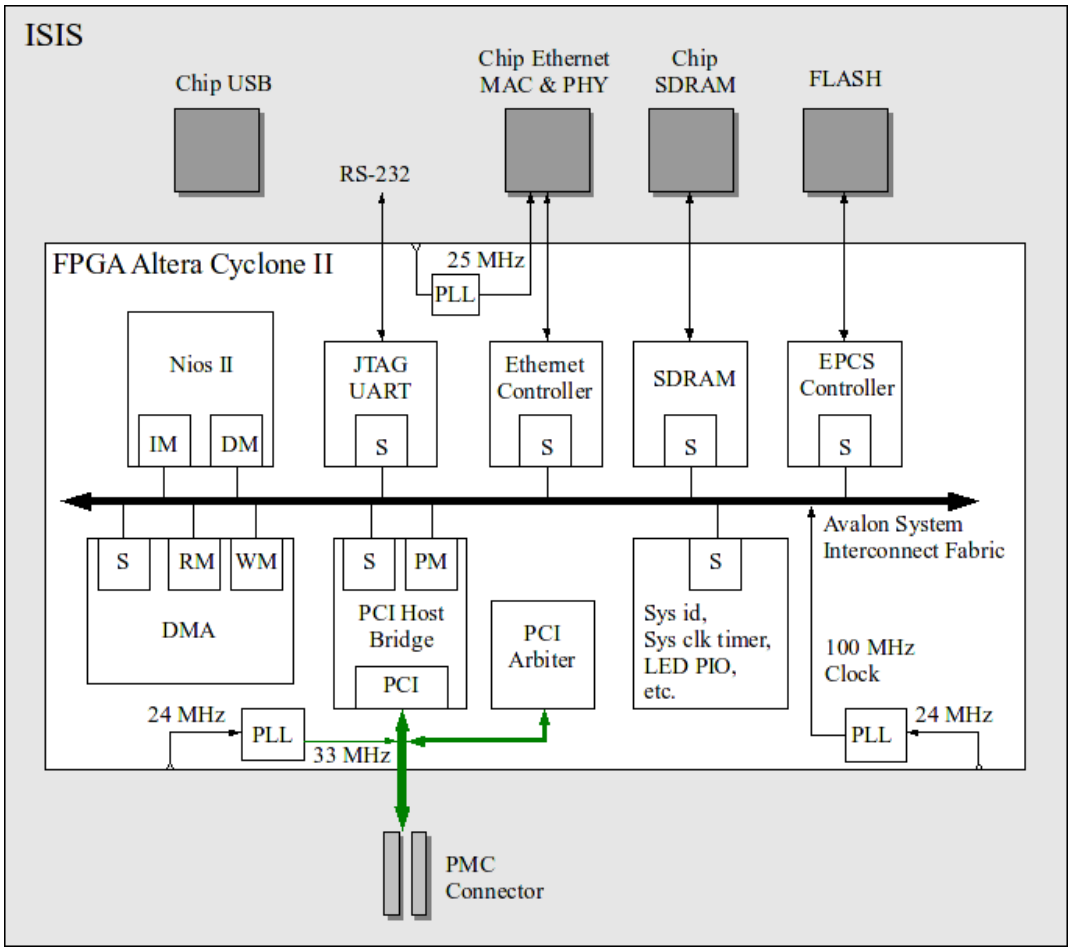


Figura 49: SoPC basado en procesador Nios II con interfaz PCI

A continuación se describen brevemente los elementos más relevantes del sistema embebido en la plataforma ISIS.

4.3.1.1 Dispositivo FPGA Altera Cyclone II EP2C20F484C6

ISIS contiene un dispositivo FPGA de Altera, familia Cyclone II, EP2C20F484C6. El grado de velocidad del dispositivo es C6, contiene 315 pines I/Os disponibles para usuario de un total de 484 (package F484). En la tabla 6 se muestran las principales características del dispositivo EP2C20F484C6 [20].

LEs	18.752
M4K RAM Blocks	52
Total RAM Bits	239.616
Embedded Multipliers	26
PLLs	4
Maximum user I/O pins	315

Tabla 6: Características del dispositivo FPGA Cyclone II EP2C20F484C6

El dispositivo FPGA puede ser configurado de las siguientes maneras:

- Automáticamente al inicio (encendido) por el dispositivo de configuración serial Altera EPCS64.
- Presionando el botón Init (SW8) el dispositivo se inicializa y reconfigura cargando la configuración almacenada en memoria flash.
- Vía JTAG usando la herramienta de programación de Altera USB-Blaster
- Presionando el botón Rz (SW7) se inicializa el sistema cargado en la FPGA.

El dispositivo EPCS64 es una memoria Flash serial con lógica adicional específica para programación de dispositivos FPGA de Altera Cyclone II. Después de un evento de encendido o reset, los dispositivos EP2C20F484 y EPCS64 se comunican entre sí para cargar un diseño desde memoria flash hacia las celdas SRAM del dispositivo FPGA. Una vez que se completa el proceso de configuración, empieza a correr el nuevo core sobre la FPGA.

4.3.1.2 Procesador Nios II

Nios II [21] es un procesador RISC embebido de 32 bit diseñado específicamente para dispositivos FPGA de Altera. Su naturaleza de IP core permite su parametrización (por ejemplo, definir tamaños de Cache de instrucciones y datos), y extensión de funcionalidades por medio de la adición de instrucciones y periféricos personalizados con el potencial de ejecutar algoritmos específicos en hardware dedicado, liberando al procesador para manejar otras tareas. Nios II usa la arquitectura de comunicación Avalon System Interconnect Fabric para interacción con otros IP cores.

Existen tres configuraciones posibles de Nios II: Nios II/f (*fast*), Nios II/s (*standard*), y Nios II/e (*economy*). En la tabla 7 se muestran las características del procesador Nios II/f implementado en el SoPC de la figura 49:

Nombre CPU	Core Nios II/f (fast)
Tipo	RISC
Data Bus	32 bit
Instruction Cache	4 Kbytes (burst disabled)
Data Cache	4 Kbytes (burst disabled)
Data cache Line Size	32 Bytes
Mecanismos de optimización	Branch prediction
Hardware para operaciones específicas	Hardware Multiply, Hardware Divide, Barrel Shifter
Performance @ 100 MHz	Hasta 101 DMIPS
Logic Usage	1400 – 1800 Les (Logic Elements)
Memory Usage	3 M4Ks + Cache
Reset Vector	Memory SDRAM, offset 0x0
Exception Vector	Memory SDRAM, offset 0x20
MMU, MPU	No
Exception Checking	No
Instruction Set	Default

Tabla 7: Características del procesador Nios II/f

Para la especificación y configuración del sistema se usa la herramienta SOPC Builder, que permite al usuario vía interfaz gráfica escoger y parametrizar el tipo de procesador Nios II, y añadir periféricos al sistema embebido. SOPC Builder genera el System Interconnect Fabric y el software Quartus II realiza la síntesis, ubicación y ruteo, ensamble, y análisis de sincronización temporal o *timing analysis*, para implementar el SoPC en el dispositivo FPGA objetivo.

Para generación de software Altera provee la herramienta Embedded Development Suite (EDS), que permite simular aplicaciones, o descargarlas y correrlas sobre el procesador embebido.

4.3.1.3 Interconexión de subsistema PCI

En la figura 50 se muestra el detalle de interconexión del subsistema PCI especificado con el software Quartus II, que incorpora un PLL (pll_33_mhz) dedicado para generación del clock PCI, el módulo pci_arbiter_top.v para arbitración de bus PCI, y el core PCI Bridge configurado como Host. Todas las señales PCI que van al conector PMC se implementan con buffers triestado. Además los pines del dispositivo FPGA que van al conector PMC pueden ser configurados como compatibles con el estándar PCI (esto asegura la satisfacción de los requerimientos eléctricos del estándar PCI para los bloques I/O del dispositivo FPGA).

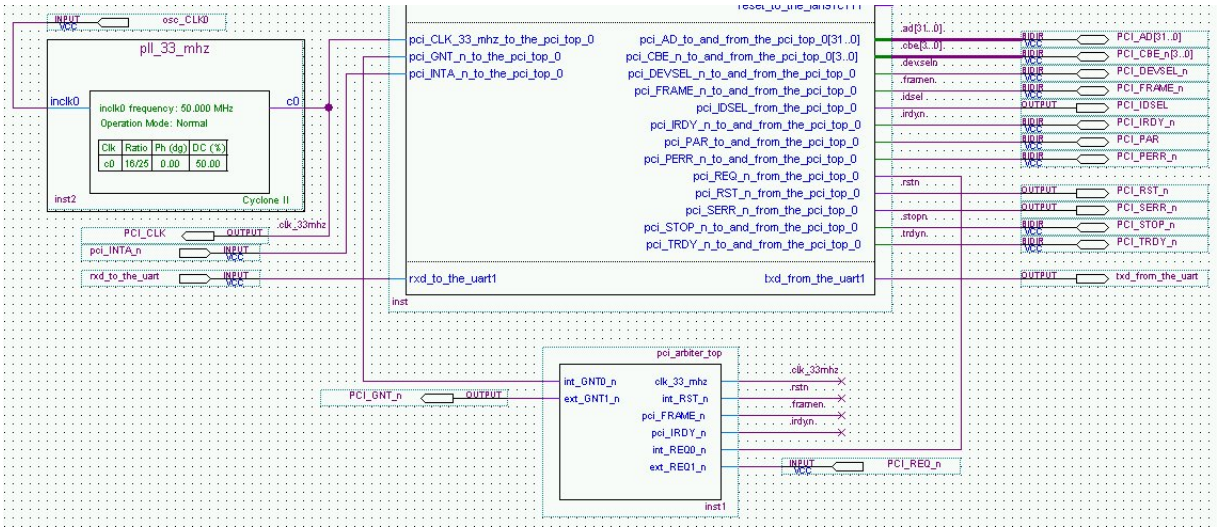


Figura 50: Detalle de interconexión del subsistema PCI

4.3.2 Asignación de recursos PCI en mapa de memoria

Se implementa un espacio de 16 MB de memoria para el core PCI Bridge, como se muestra en la figura 51.

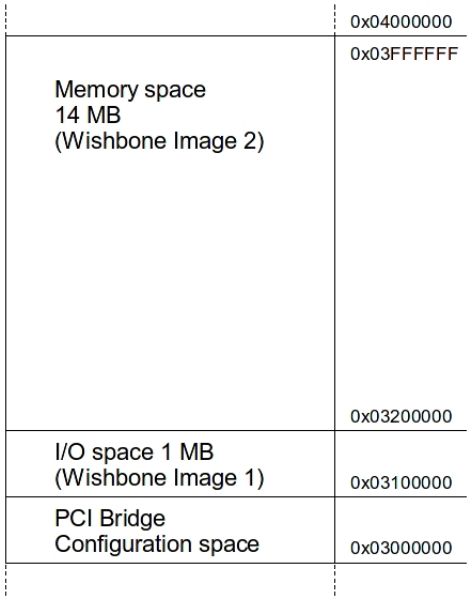


Figura 51: Asignación de recursos PCI en espacio de memoria

El primer segmento de 1 MB está destinado al espacio de configuración del core PCI Bridge, mientras que los siguientes segmentos de 1 y 14 MB corresponden a los espacios de I/O y de memoria PCI respectivamente. El espacio I/O (direcciones 0x03100000 a 0x031FFFFF) se accede mediante la Imagen Wishbone 1, y el espacio de memoria (direcciones 0x03200000 a

0x03FFFFFF) se accede mediante la Imagen Wishbone 2. Las imágenes Wishbone y PCI del core PCI Bridge se definen por medio del archivo pci_user_constants.v, donde se configuran especificando de sus direcciones base, máscaras de direcciones (para determinar tamaño de imágenes), y opciones de traducción de direcciones y decodificación, entre otras[11].

Cuando el PCI Host Bridge detecta un acceso a memoria desde el bus Avalon que cae dentro de una de las imágenes implementadas, éste traduce el ciclo al espacio PCI correspondiente (I/O o memoria) con el comando de bus apropiado. Por otro lado, cuando se detecta un acceso a memoria desde el bus PCI hacia el bus Avalon, el bridge mapea la transacción al espacio de memoria (física) del lado Avalon que corresponde a los 32 MB de memoria SDRAM.

La distribución de recursos PCI otorga 14 MB de espacio de memoria para mapeo de dispositivos PCI. El sistema operativo determina los requerimientos de memoria de los dispositivos presentes en el bus, y asigna ubicaciones dentro del espacio de memoria de modo que cada uno tenga un espacio de uso exclusivo, asegurando que puedan coexistir sin conflictos al cargar sus respectivos drivers.

4.3.3 Interrupciones

Se asignan identificadores numéricos a las interrupciones de dispositivos esclavo capaces de generar IRQs a la CPU. En particular, la interfaz esclavo del core PCI Bridge tiene una señal de interrupción que se activa cuando la señal de interrupción INTA# se activa en el bus. De este modo las interrupciones señaladas en el bus se transfieren en forma transparente al bus Avalon.

Para acusar recibo de interrupción al core PCI Bridge, la CPU debe realizar un ciclo de lectura al registro INT_ACK en el espacio de configuración. Este ciclo no se responde hasta que el módulo PCI Master arbitra por el control del bus, y obtiene el dato solicitado. La dirección durante un ciclo *interrupt acknowledge* no tiene significado, mientras que las líneas de *byte enable* indican el tamaño del vector de interrupción retornado. El registro ICR (*Interrupt Control Register*) permite habilitar/deshabilitar la generación de interrupciones desde distintas fuentes (detección de errores de paridad o del sistema), y el registro ISR (*Interrupt Status Register*) permite determinar la fuente de la interrupción y limpiar una solicitud de interrupción (adicionalmente permite activar la señal de reset PCI vía software).

4.4 Desarrollo de software de control y diagnóstico de hardware PCI

Una vez implementado el SoPC descrito en la sección 4.3.1 sobre la plataforma ISIS, se requiere un software que corra sobre el procesador Nios II capaz de estimular al core PCI Bridge para la generación de eventos y ciclos de bus específicos. Este programa tiene la importancia de ser la base para el desarrollo del driver del core PCI Bridge para operación con el sistema operativo uClinux descrito en la sección 4.5 . A continuación se describe el diseño e implementación de este software para facilitar las tareas de debug y verificación de hardware PCI.

11 Para más detalles referirse a los documentos oficiales de Opencores [18] y [19].

4.4.1 Descripción

Con el apoyo de la herramienta para desarrollo de software Nios II IDE, se diseña un programa en lenguaje C para correr sobre el procesador Nios II embebido en el dispositivo FPGA de la plataforma ISIS. Mediante una interfaz de usuario por consola, se escogen opciones de un menú simple que permite realizar las siguientes tareas:

1. Activar señal RESET en el bus PCI
2. Activar rutina de inicialización (software reset, y configuración de registros del host PCI bridge).
3. Realizar un *Configuration Read Cycle Type 0* en el bus PCI
4. Realizar un *Configuration Write Cycle Type 0* en el bus PCI
5. Leer los registros *Status* y *Command* del core PCI Bridge
6. Efectuar un ciclo de lectura al espacio de configuración del core PCI Bridge
7. Efectuar un ciclo de escritura al espacio de configuración del core PCI Bridge
8. Chequear registros de reporte de errores del core PCI Bridge
9. Leer los registros *Status* y *Command* del dispositivo PCI presente en el bus
10. Realizar un test de memoria a la tarjeta PCI DUC-II
11. Realizar un test de memoria a la tarjeta PCI DUC-I

El conjunto de operaciones anterior permite al usuario estimular al core PCI Bridge y a los dispositivos presentes en el bus a través del conector PMC, para diagnosticar y validar su comportamiento. Por ejemplo, permite leer y escribir registros específicos del espacio de configuración, tanto del core PCI Bridge como del dispositivo presente en el bus, generar ciclos de bus de lectura/escritura con direcciones y datos ingresados por usuario, y el chequeo de la correcta realización de las operaciones comandas. Además se beneficia del mecanismo de reporte de errores del core PCI Bridge para informar al usuario sobre errores tales como condiciones de *Master Abort* (recibido y señalado), *Target Abort* (recibido y señalado), error de paridad, error de sistema, indicando además las direcciones y datos fuente del error.

Adicionalmente se incorporan rutinas para realizar test de memoria a dos tarjetas PCI de Digital Up Conversion (DUC), versiones I y II. En particular, la tarjeta PCI DUC-II forma parte del sistema final donde se realizan las pruebas finales del presente trabajo, como se describe en la sección 4.8 .

Una gran ventaja del programa desarrollado cuando es incorporado como archivo ejecutable sobre uClinux, es el monitoreo del bus PCI durante la operación de un software de aplicación distinto. Por ejemplo, mientras corre un software de aplicación sobre uClinux que hace uso de la tarjeta PCI DUC-II, el usuario puede acceder vía Telnet desde un PC remoto a la plataforma ISIS y ejecutar el programa de control y diagnóstico PCI, para observar los registros del core PCI Bridge, registros de la tarjeta DUC-II, leer o escribir datos a direcciones de memoria específicos, etc., todo mientras opera el software de aplicación. También permite efectuar un reset remoto sobre el hardware PCI. Luego, el programa de control y diagnóstico PCI es una

herramienta esencial para el desarrollo de sistemas de hardware y software que hacen uso del bus PCI.

4.4.2 Funciones

Se desarrollan las siguientes funciones que encapsulan las operaciones básicas de control del core PCI Host Bridge, y constituyen los bloques de construcción del programa. Además, estas funciones facilitan el desarrollo de la capa de software que comunica el core PCI Bridge con el kernel de Linux.

- `void oc_pci_init(void)`

  Rutinas de inicialización del PCI Host Bridge. Activación señal de reset PCI, habilitación de interrupción, limpieza del registro Status, habilitación de operación como Master y respuesta a espacios de memoria e I/O, y habilitación de mecanismo de reporte de errores.

- `void wb_error_reporting(void)`

  Examina los registros de reporte de errores Wishbone del PCI Host Bridge, y en caso de error señaliza una advertencia o *warning*. Extrae información sobre el tipo de error, dirección de origen, y dato asociado cuando corresponde.

- `void read_host_status(void)`

  Lee el registro Status del PCI Host Bridge, y extrae información sobre su estado. En caso de detectar condición de *Received Master Abort*, *Received Target Abort*, *Signaled Target Abort*, *Parity Error*, o *System Error*, señaliza el o los *warnings* correspondientes.

- `void read_host_command(void)`

  Lee el registro Command del PCI Host Bridge, y extrae información sobre sus funcionalidades habilitadas.

- `int oc_pci_read_conf_cycle (int bus, int devfn, int where, int size)`

  Generación de un *Configuration Read Cycle Type 0*. Para generar el ciclo de configuración primero escribe la dirección objetivo en el registro CNF_ADDR (*Configuration Address*), donde se indica el bus, dispositivo, función, número de registro, y tamaño de la lectura (1 para byte, 2 para half-word, y 4 para word). Luego se lee el registro CNF_DATA (*Configuration Data*) para generar el ciclo, lo que impone estado de espera en el bus Avalon hasta que se complete el ciclo en el bus PCI, retornando el valor leído.

- `void oc_pci_write_conf_cycle (int bus, int devfn, int where, int size, int val)`

  Generación de un *Configuration Write Cycle Type 0*. Similar a la función anterior, con la diferencia de que se efectúa un ciclo de escritura en el registro CNF_DATA con el valor int val. Este ciclo de escritura se acusa como recibido inmediatamente en el bus Avalon, sin importar si el ciclo termina con éxito en el bus PCI. Por esta razón el core PCI Bridge incorpora un mecanismo de reporte de errores en el bus Wishbone.

- `void read_device_status(void)`

  Lee el registro Status del dispositivo PCI presente en el conector PMC (bus 0, slot 0, función 0), y extrae información sobre su estado. En caso de detectar condición de *Received Master Abort*, *Received Target Abort*, *Signaled Target Abort*, *Parity Error*, o *System Error*, señaliza el o los *warnings* correspondientes.

- `void read_device_command(void)`

  Lee el registro Command del dispositivo PCI presente en el conector PMC (bus 0, slot 0, función 0), y extrae información sobre sus funcionalidades habilitadas.

4.4.3 Diagrama de flujo

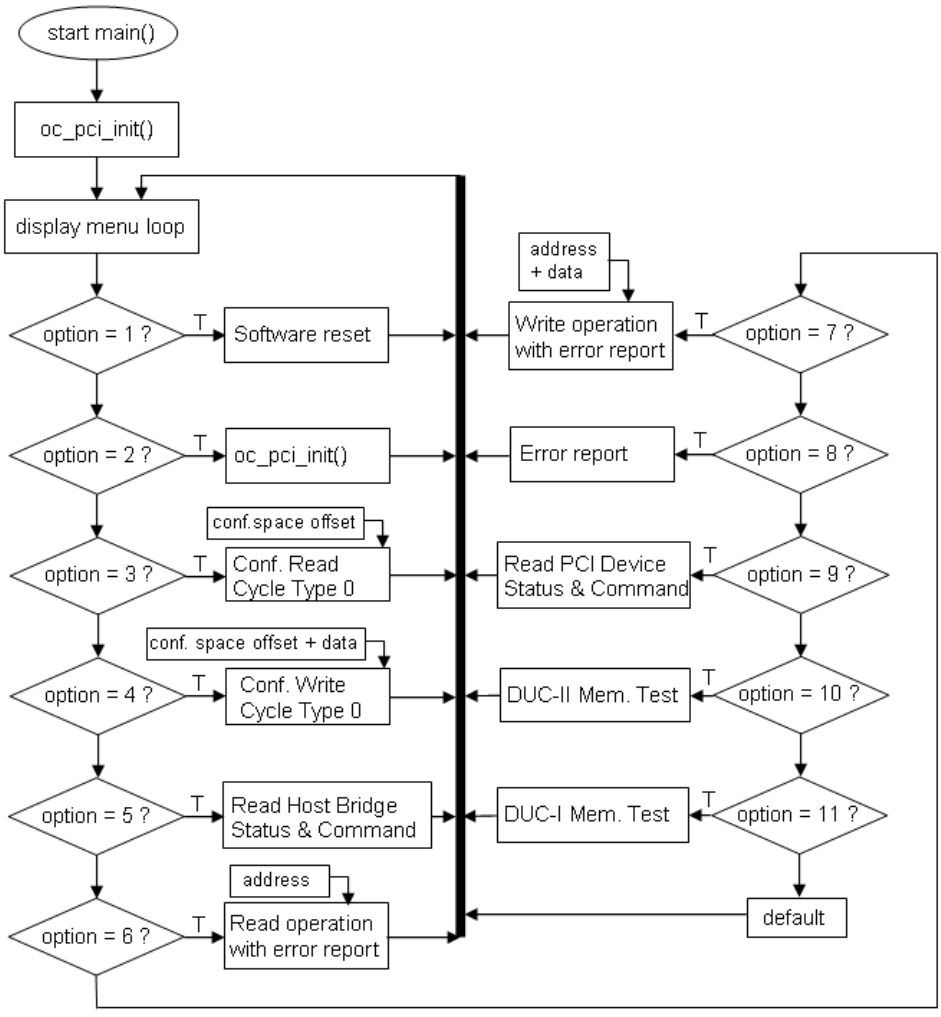


Figura 52: Diagrama de flujo general de programa de control y diagnóstico PCI

4.5 Desarrollo de interfaz entre core PCI Host Bridge y kernel de Linux

El subsistema PCI es uno de los códigos más complejos que conforman el kernel del sistema operativo Linux. Los dispositivos PCI Host Bridge de una placa madre requieren de una capa de software (específica a cada dispositivo) que comunica el hardware con el subsistema PCI del kernel. Este software permite al sistema operativo construir un mapa de memoria consistente en base a los dispositivos PCI presentes en el bus, de modo que al cargar los drivers de los dispositivos estos puedan coexistir sin conflictos.

A continuación se describe la implementación de la capa de software de permite al kernel del sistema operativo uClinux comunicarse con el core PCI Bridge configurado como Host.

4.5.1 Descripción

El sistema operativo uClinux es una versión compacta de Linux orientada a arquitecturas sin MMU (*Memory Management Unit*), como es el caso del procesador Nios II usado en el presente trabajo. Luego, uClinux no maneja memoria virtual (sólo memoria física), necesita cargar todo el código de programa en memoria RAM (en vez de cargar solamente páginas en la medida que se necesitan), no posee protección de memoria, y no posee swapping. Sin embargo uClinux es de tamaño reducido, rápido, y otorga todas las capacidades del kernel de Linux.

El subsistema PCI del kernel de uClinux requiere estructuras y funciones de bajo nivel para comunicación con el PCI Host Bridge. Estas se pueden categorizar de la siguiente forma:

- Funciones de acceso al espacio de configuración PCI
- Recursos PCI (espacio I/O y espacio de memoria)
- Funciones PCI BIOS (rutinas de inicialización)
- Funciones para manejo de interrupciones
- Código específico del dispositivo PCI Host Bridge

Las funciones anteriores de bajo nivel permiten comunicación directa con el hardware PCI y constituyen la interfaz que requieren las rutinas de más alto nivel del subsistema PCI. Estas rutinas de más alto nivel (dependientes de la versión del kernel) son independientes del dispositivo y se pueden categorizar de la siguiente forma:

- PCI BIOS: rutinas de inicialización, escaneo de bus, etc.
- Autoconfiguración PCI: rutinas de asignación automática de recursos de memoria
- Manejo de interrupciones PCI

A continuación se describen las funciones de bajo nivel implementadas en un archivo oc_pci.c para comunicar el core PCI Bridge con el subsistema PCI del kernel 2.4 de Linux.

4.5.2 Funciones de acceso al espacio de configuración PCI

Se implementan las siguientes funciones de bajo nivel para generación de ciclos de configuración de lectura y escritura PCI Tipo 0 (es decir, no para dispositivos PCI-to-PCI Bridges):

- `static int oc_pci_read(struct pci_bus *bus, unsigned int devfn, int where, int size, u32 *val)`

  Generación de un *Configuration Read Cycle Type 0*. Similar en estructura a la función `oc_pci_read_conf_cycle`. Llama a la función `pci_range_ck` para chequear si el dispositivo PCI que trata de acceder existe.

- `static int oc_pci_write(struct pci_bus *bus, unsigned int devfn, int where, int size, u32 val)`

  Generación de un *Configuration Write Cycle Type 0*. Similar en estructura a la función `oc_pci_write_conf_cycle`. Al igual que la función anterior también llama a `pci_range_ck`.

- `static inline int pci_range_ck(struct pci_bus *bus, unsigned int devfn)`

  Chequea que el bus y slot PCI se encuentren en el rango correcto. En el caso de la plataforma ISIS se define solamente el bus 0 y el slot 0. Si el bus es 0 y el slot es 0, entonces retorna 1, en caso contrario retorna -1.

4.5.3 Recursos PCI

La información sobre el tamaño de los espacios I/O y de memoria que implementa el PCI Host Bridge se proveen al kernel por medio de las siguientes estructuras:

```
static struct resource oc_io_resource = {
      .name   = "OPENCORES PCI IO",
      .start  = OC_PCI_IO_BASE,
      .end    = OC_PCI_IO_BASE + OC_PCI_IO_SIZE - 1,
      .flags  = IORESOURCE_IO
}
static struct resource oc_mem_resource = {
      .name   = "OPENCORES PCI MEM",
      .start  = OC_PCI_MEM_BASE,
      .end    = OC_PCI_MEM_BASE + OC_PCI_MEM_SIZE - 1,
      .flags  = IORESOURCE_MEM
}
```

Cada una de estas estructuras definen los recursos de espacio I/O y memoria PCI como se muestra en la sección 4.3.2 . Por ejemplo, el recurso de memoria PCI queda definido con los parámetros OC_PCI_MEM_BASE = 0x03200000 y OC_PCI_MEM_SIZE = 0xE00000 = 14 MB.

4.5.4 Funciones PCI BIOS

Se implementan las siguientes funciones de bajo nivel para inicialización del hardware PCI:

- `static __init int oc_pci_init(void)`

  Rutinas de inicialización del PCI Host Bridge. Activación señal de reset PCI, habilitación de interrupción, limpieza del registro Status, habilitación de operación como Master y respuesta a espacios de memoria e I/O, y habilitación de mecanismo de reporte de errores. Una vez finalizadas las tareas anteriores llama a la función `pcibios_init`, que se encarga de la auto asignación de recursos, crea la estructura de árbol PCI, y ejecuta código específico de la placa madre si es necesario. El prefijo `__init` indica al kernel que es una función de inicio.

4.5.5 Manejo de interrupciones

Se implementan las siguientes funciones para proveer al kernel información sobre interrupciones que implementa el hardware PCI:

- `int __init pcibios_map_platform_irq(u8 slot, u8 pin)`

  Esta función retorna el número asociado a la interrupción INTA# que el PCI Host Bridge mapea al bus Avalon.

- `static int oc_pci_pci_lookup_irq(struct pci_dev *dev, u8 slot, u8 pin)`

  Esta función llama a `pcibios_map_platform_irq` para mapear la interrupción PCI al kernel.

4.6 Diseño de placa de adaptación PMC a PCI

4.6.1 Requerimientos

Se diseña el hardware de adaptación PMC (PCI Mezzanine Card) a PCI para permitir la conexión de tarjetas PCI de PC a la plataforma ISIS. Este hardware debe proveer los voltajes de alimentación a la tarjeta PCI sin interferir con las fuentes de la placa madre ISIS. Los requerimientos del hardware son los siguientes:

1. Alimentación externa con voltaje dc entre 12 y 18V.
2. Conector PCI Mezzanine Plug (macho) en configuración *host*, según especificación IEEE P1386.1-2001 [1].
3. Conector PCI 3.3V, 32 bit @ 33 MHz (hembra), en configuración para *add-in card*.
4. Fuentes de voltaje independientes de +3.3V, +5V y +/-12V para alimentar tarjetas PCI 3.3V, 32 bit @ 33 MHz a través del conector PCI.
5. Requerimientos de potencia y capa física según especificación PCI 3.0 [15].

En las figuras 53 y 54 se muestran las vistas superior e inferior de la placa de adaptación PMC a PCI.

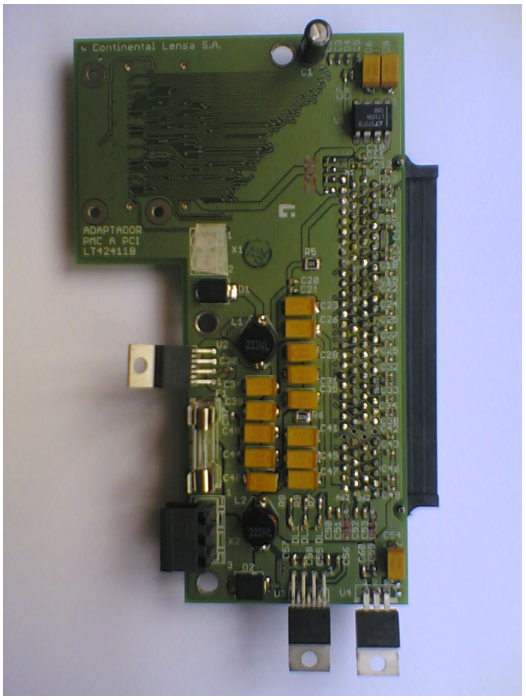


Figura 53: Vista superior Adaptador PMC a PCI

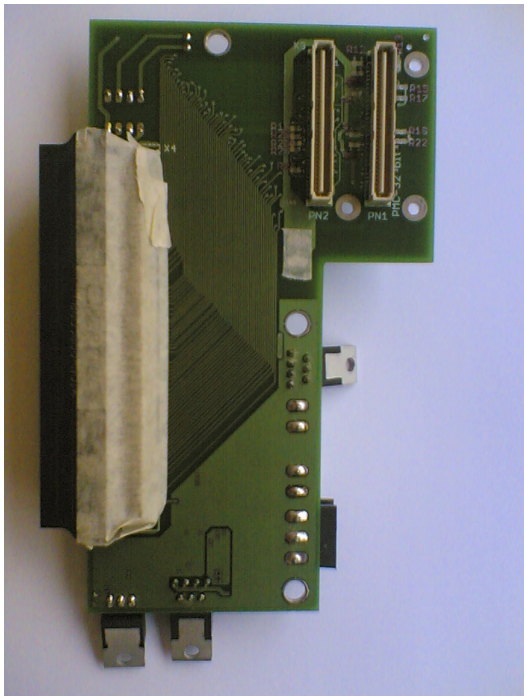


Figura 54: Vista inferior Adaptador PMC a PCI

4.6.2 Especificaciones

4.6.2.1 Características

El adaptador PMC a PCI incorpora un conector PCI Mezzanine Plug (macho) en configuración *host*, compatible con la especificación IEEE P1386.1-2001, conectado por medio de un PCB de dos capas a un conector PCI 3.3V 32-bit @ 33 MHz de PC ángulo recto, en configuración para *add-in card*. El diseño de hardware posee las siguientes características:

- Adaptador PCI *bridgeless* activo (es decir, no existe ningún tipo de lógica *on-board*, sólo fuentes de alimentación y resistencias pull up/down).
- Compatible con sistemas PCI 3.3V, 32-bit @ 33 MHz (señal M66EN por defecto a tierra con resistencia pull down).
- Permite la interconexión de placas madres que incorporen chipset PCI y conector PMC, con tarjetas PCI 3.3V 32-bit @ 33 MHz, o tarjetas PCI Universales 32-bit @ 33 MHz.
- Se alimenta con una fuente externa entre 12 y 18Vdc por medio de un conector de poder.
- Incorpora fuentes de potencia independientes (+3,3V, +5V y +/-12V) y aisladas del conector PMC (para no interferir con las fuentes de poder internas de la placa madre).
- La fuente de +3.3V permite una potencia máxima de +3.3V @ 5A max = 49.5 [Watts] (menor al requerimiento de potencia PCI). Las fuentes de +5V @ 5A y +/-12V @ 5A satisfacen los requerimientos de potencia PCI.
- Incorpora resistencias pull up/down acorde a la especificación PCI para estabilidad de señales en power up.

4.6.2.2 Componentes principales

- Conector PCI Mezzanine IEEE 1386 SMD Plug.
- Conector PCI 32 Bit @ 3.3V en ángulo recto.
- Fuentes independientes de +3.3V/5A, +5V/5A con reguladores switching step down, y fuente de +/-12V/5A con regulador LDO (Low Drop Out).
- Fusible de protección en conector de poder.
- Conector auxiliar para fuente externa de +3.3V.
- Disipadores de potencia de 10 [ºC/W] en cada regulador.
- 3 LEDs de *power on* independientes para cada voltaje de alimentación PCI.

En el Anexo E se detalla el dimensionamiento de las pistas del PCB, y en el Anexo F se detalla el cálculo de los disipadores de calor.

4.6.2.3 Características y funcionalidades no soportadas

- No soporta Boundary Scan (JTAG IEEE 1149.1)
- No soporta funcionalidades PCI de Power Management
- No soporta Plug & Play
- La fuente de +3.3V @ 5A no satisface los requerimientos de potencia de la especificación PCI (+3.3V @ 7.6A max). Para interconexión de PCI add-in cards que requieran potencias mayores a +3.3V @ 5A se provee un conector auxiliar para conexión de fuente externa de +3.3V.

4.6.2.4 Consideraciones de uso

- Antes de conectar una tarjeta PCI a la plataforma ISIS por medio de la placa de adaptación, el dispositivo FPGA debe estar configurado con un SoPC que conecte los pines de la interfaz PCI con el conector PMC correctamente a través de buffers triestado. Si la FPGA no se encuentra configurada correctamente al momento de alimentar el sistema, se puede causar daño a la FPGA y/o a la tarjeta PCI. Por lo tanto se recomienda que la plataforma ISIS contenga un SoPC con interfaz PCI almacenado en memoria Flash, para configuración automática de pines al encendido.
- Antes de conectar una tarjeta PCI a una placa madre con ambiente de señal +3.3V, es responsabilidad del usuario asegurarse que la tarjeta es compatible con el respectivo ambiente de señal (es decir, no puede conectarse una tarjeta de +5V a un conector PCI de +3.3V). Existen ciertos fabricantes que venden tarjetas de +5V con marcas de "Universal", que al ser conectadas a un sistema de +3.3V pueden causar serios daños. Para comprobar el verdadero estatus de "Universal" de una tarjeta PCI, el usuario debe realizar un test de continuidad para comprobar que los pines V(I/O) estén aislados de los pines de +5V y +3.3V.

4.6.3 Protocolos de prueba

Para verificar la correcta operación del hardware se diseña un protocolo de pruebas que evalúa características eléctricas fundamentales del estándar PCI. El protocolo de pruebas para la placa de adaptación se divide en las siguientes procesos de validación:

1. Validación de voltajes de alimentación PCI.
2. Validación de recursos centrales PCI (resistencias pull up/dow en señales PCI y pinout de conectores PMC y PCI).
3. Verificación de parámetros de timing PCI.

El protocolo de prueba anterior se describe con detalle en el Anexo D.

4.6.4 Esquemáticos

Los esquemáticos de la placa de adaptación PMC a PCI se encuentran en el Anexo C.

4.7 Integración de plataforma de desarrollo de Altera con tarjeta PCI DUC-II

Para el desarrollo de hardware y software PCI se utiliza la plataforma Altera Nios II Development Kit, Cyclone II Edition (figura 55), que se basa en un dispositivo FPGA Cyclone II EP2C35F672 e incorpora 16 MB de memoria flash, 2 MB de memoria SRAM, 32 MB de memoria DDR SDRAM, chip Ethernet 10/100 MAC & PHY, puerto RS-232, botones, LEDs indicadores, conector para acceso a 40 pines I/O, entre otros periféricos [22]. Además, la plataforma posee un conector PMC 3.3V de 32 bit hembra similar al de la plataforma ISIS.

Figura 55: Altera Nios II Development Kit, Cyclone II Edition

El SoPC diseñado en el presente trabajo (sección 4.3) se puede portar fácilmente entre las plataforma de desarrollo de Altera y la plataforma ISIS. En efecto, dado que los dispositivos FPGA de ambas plataformas pertenecen a la misma familia Cyclone II, la especificación del sistema es la misma a excepción del controlador de memoria (DDR SDRAM), configuración de PLLs para generación de clocks, y asignaciones de pines y parámetros de compilación en el software Quartus II.

La plataforma de desarrollo de Altera presenta las siguientes diferencias en la implementación del conector PMC, respecto de la plataforma ISIS:

- Incorpora resistencias pull up/down on-board según especificación PCI.
- Provee buffers (resistencias serie de 5 kOhm) para adaptación de señales de 5V a 3.3V. Esto permite la conexión de tarjetas PCI 5V, 32 bit @ 33 MHz.

Luego, para hacer uso del adaptador PMC con la plataforma de Altera se requiere que la placa no incorpore las resistencias pull up/down, es decir, se usa una placa especial donde no se soldan estas resistencias. Esto evita la replicación de recursos centrales PCI y violación de la especificación de una carga por señal, para evitar reflexiones indeseadas y minimizar consumo de potencia.

En la figura 56 se muestra la plataforma de desarrollo de Altera controlando a la tarjeta PCI DUC-II por medio del adaptador PMC a PCI. La plataforma de desarrollo contiene el SoPC basado en procesador Nios II con interfaz PCI, donde corre el software de control y diagnóstico PCI sobre uClinux. El adaptador se alimenta con una fuente externa de PC que provee 12Vdc.

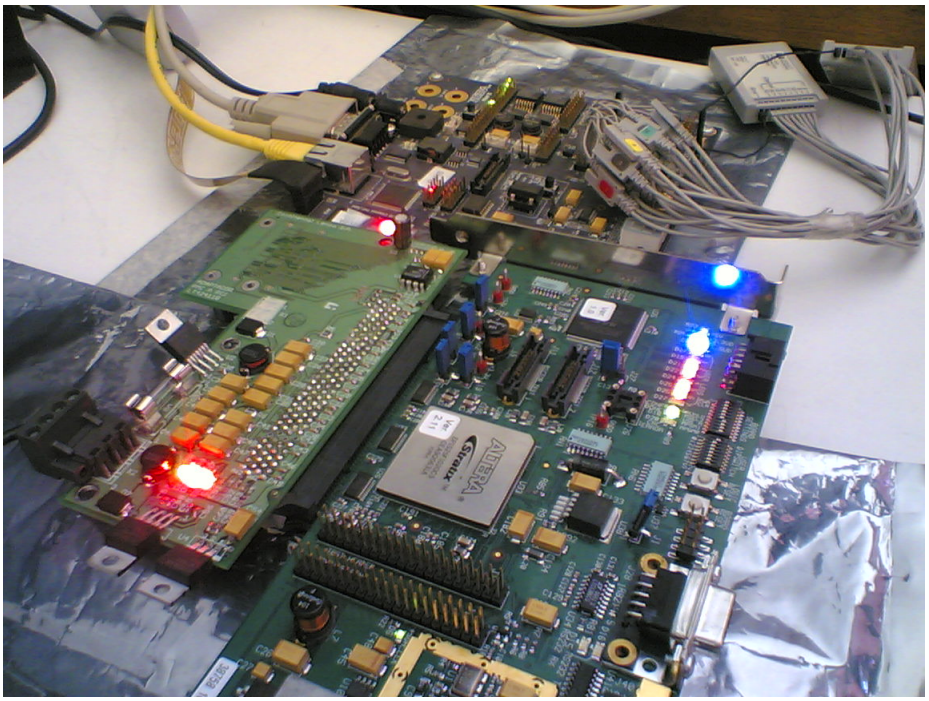


Figura 56: Integración de Altera Nios II Dev. Kit y tarjeta PCI DUC-II

El control del sistema se hace desde un PC remoto por dos vías posibles:

1. Terminal uClinux vía cable JTAG UART: permite a través de línea de comandos ejecutar programas, iniciar y terminar procesos, etc.
2. Consola Telnet vía cable Ethernet (conectado a la LAN): Se accede a la plataforma vía Telnet desde un PC remoto conectado a Internet, para ejecutar programas, iniciar y terminar procesos desde línea de comandos. Requiere configurar dirección IP, iniciar demonio de Internet, y habilitar NFS (network file system) en uClinux para compartir el sistema de archivos con el PC remoto (facilita el desarrollo de software al habilitar la actualización automática remota de los ejecutables que se desarrollan en el PC).

El sistema anterior facilita la detección y corrección de errores de hardware y software, al otorgar la posibilidad de monitorear tanto los registros del core PCI Host Bridge como del PCI Target Bridge (en la tarjeta PCI). Por otro lado, mediante el uso de las utilidades PCI de Linux (por ejemplo *lspci*) se puede observar si el sistema operativo fue capaz de reconocer el hardware PCI y construir la estructura de árbol del bus durante el proceso de inicialización.

A través de los pines del conector PCI se pueden capturar las formas de onda PCI con osciloscopio, y por medio de los pines de I/O de la plataforma se puede analizar la sincronización temporal o timing del bus con un analizador lógico.

4.8 Integración de plataforma ISIS con tarjeta PCI DUC-II

4.8.1 Descripción del sistema

En la figura 57 se muestra la interconexión entre la plataforma ISIS y la tarjeta PCI DUC-II por medio del adaptador PMC a PCI.

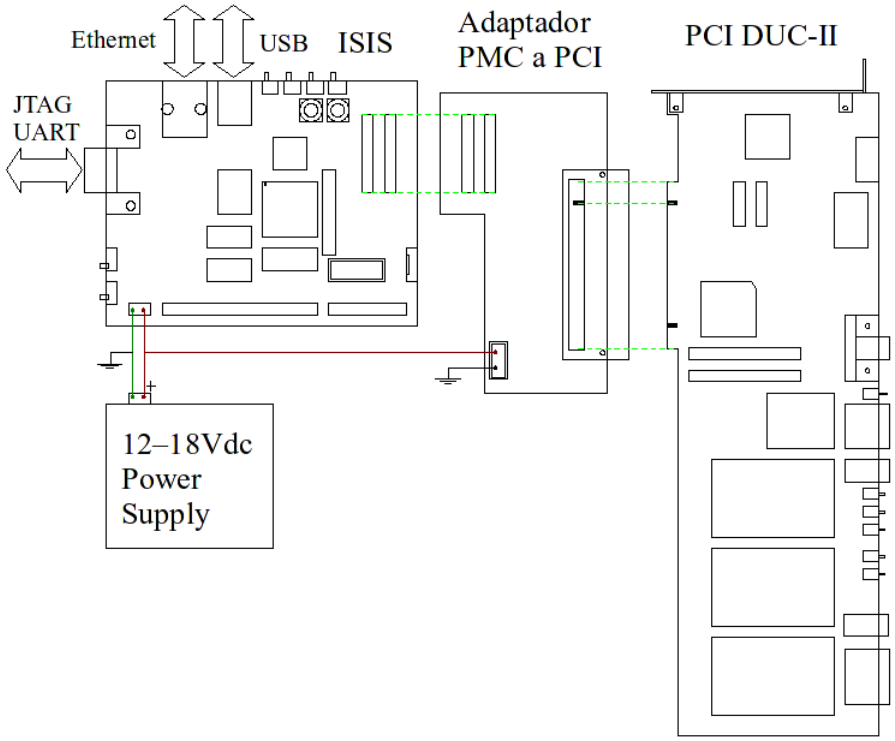


Figura 57: Diagrama de interconexión ISIS-DUC-II

La alimentación de la plataforma ISIS puede ser ya sea por medio de una fuente DC externa entre 12 y 18V como se muestra en la figura 57, o a través del puerto USB desde un PC. La alimentación de la tarjeta PCI está a cargo del adaptador PMC a PCI, que genera todos los voltajes de alimentación necesarios por medio de reguladores *on-board*, y los suministra junto con el voltaje de referencia (tierra) a través del conector PCI. El adaptador se alimenta con una fuente externa entre 12 y 18Vdc.

La plataforma ISIS contiene el SoPC descrito en la sección 4.3 , donde el procesador Nios II corre sobre el sistema operativo uClinux. El usuario puede controlar el sistema a través de un PC remoto por las siguientes dos vías: terminal uClinux vía cable JTAG UART, y consola remota vía Telnet, a través del puerto Ethernet conectado a la LAN. Por medio de un terminal remoto el usuario puede iniciar o terminar programas, en particular puede iniciar el software de control y diagnóstico de hardware PCI descrito en la sección 4.4 para evaluar el funcionamiento del sistema.

4.8.2 Protocolos de prueba y mediciones

4.8.2.1 Protocolo de prueba #1: Validación de subsistema PCI con maestro único

Se define como *subsistema PCI* al sistema compuesto por el core PCI Host Bridge y el árbitro PCI embebidos en el dispositivo FPGA, la capa de software entre el PCI Host Bridge y el kernel de uClinux, el adaptador PMC a PCI, y el dispositivo PCI DUC-II.

Para el sistema de la figura 57 se definen las siguientes pruebas (ejecutadas con el programa de control y diagnóstico PCI) que permiten validar la integración del subsistema PCI, en el caso que el dispositivo DUC-II actúa solamente como esclavo:

(1) Test de funcionalidad básica del core PCI Host Bridge:

- Validación de acceso de lectura/escritura a espacio de configuración del core PCI y activación por software de señal PCI RESET.

(2) Validación de comunicación con dispositivo PCI DUC-II como esclavo:

- Validación de generación de ciclos de configuración lectura/escritura PCI tipo 0. Lectura de registro Status, configuración de registro Command, y programación de BARs (*Base Address Registers*) para mapear bloques de memoria PCI a memoria del sistema (SRAM).
- Validación de generación de ciclos de lectura/escritura a bloques de espacio I/O y de memoria implementados por el dispositivo PCI DUC-II. Ejecutar test de memoria a los bloques existentes.
- Medición de tasa de transferencia promedio y tasa de error al transportar bloques de datos usando DMA, entre la plataforma ISIS y la tarjeta PCI DUC-II.

(3) Test de funcionalidad de subsistema PCI del kernel de Linux:

- Verificación de ejecución de rutinas de inicialización, reconocimiento de dispositivos y autoconfiguración PCI que forman parte del kernel de Linux.

La realización de las pruebas anteriores requiere la validación previa del adaptador PMC a PCI por medio del protocolo de pruebas definido en la sección 4.6.3 , para garantizar la operación segura de los dispositivos y descartar fallos de operación introducidos por el hardware de adaptación.

Además, la funcionalidad básica del árbitro PCI se valida previamente por medio de un *test bench* simple que pone a prueba las funcionalidades de *bus parking* y activación/desactivación de las señales REQ# y GNT# en un escenario típico con 2 maestros. La simulación permite descartar fallos de operación por arbitración de bus para las pruebas (1), (2) y (3). El árbitro se valida completamente con el protocolo de prueba 2 descrito a continuación.

4.8.2.2 Protocolo de prueba #2: Validación de sistema final multimaestro

Una vez validado el subsistema PCI con las pruebas anteriores, se requiere validar la operación del sistema final llamado GSD-21 Exgine[12], cuyo núcleo de hardware es el sistema de la figura 57. El equipo soporta un complejo software de aplicación corriendo sobre uClinux, que controla el dispositivo DUC-II operando como maestro y esclavo. La aplicación de alto nivel es para generación de señal de radiodifusión digital AM IBOC (*In-Band On-Channel*), a partir de información digital de bandabase compleja (I-Q) y de magnitud, enviada por Internet desde un transmisor digital remoto como se muestra en la figura 58. El dispositivo DUC-II recibe los datos de bandabase I-Q y de magnitud, realiza el proceso de *Digital Up Conversion* a los datos para generación de frecuencia portadora AM, realiza conversión digital a análoga, y luego filtra y genera una señal basada en fase con cruce por cero. El dispositivo DUC-II también interpola y filtra los datos de magnitud. Las señal de fase con cruce por cero y la señal de magnitud son usadas para generar la señal de radiodifusión digital AM IBOC.

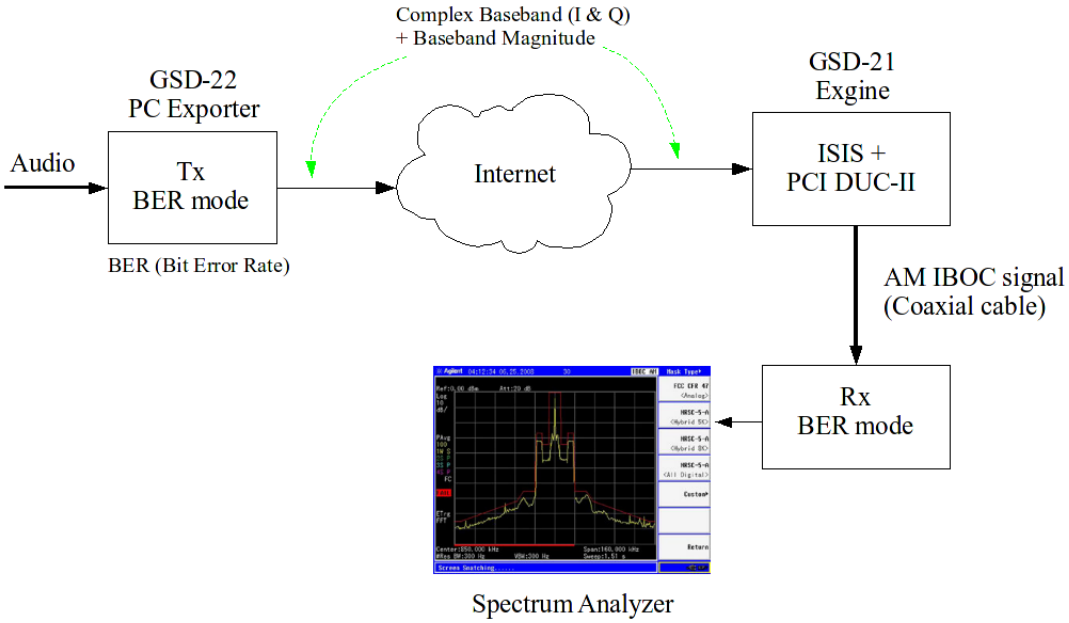


Figura 58: Arquitectura del sistema final para test en modo BER

Para la arquitectura de radiodifusión digital anterior se define el siguiente protocolo de prueba:

(1) Se ejecuta la aplicación de radio digital en modo BER (Bit Rate Error) por un periodo de 24 horas cronológicas. Este modo se usa para enviar un patrón de detección de error, y por lo tanto no se envía radio digital durante la prueba.

(2) Caracterización de las formas de onda PCI capturadas con osciloscopio en el adaptador PMC a PCI durante operación del sistema. En particular, se capturan las formas de onda de la señal CLK, y una muestra del bus AD[31:0].

12 El equipo GSD-21 Exgine es propiedad intelectual de Continental Lensa S.A

5 Análisis de resultados

5.1 Hardware embebido en dispositivo FPGA

El IP wrapper desarrollado permite la integración del IP core PCI Bridge de Opencores configurado como Host en un SoPC basado en el procesador Nios II, otorgando capacidad de conectividad con dispositivos PCI. La lógica de adaptación entre las interfaces de bus Avalon y Wishbone convierte de manera correcta los ciclos de bus iniciados por maestros Avalon (CPU Nios II y controlador DMA) a ciclos de bus Wishbone interpretados por la máquina de estados Wishbone Slave del core PCI Bridge. Además los ciclos de bus iniciados por la máquina de estados Wishbone Master del core PCI Bridge, son convertidos a ciclos de bus Avalon e interpretados correctamente por el System Interconnect Fabric.

La lógica de pegamento que resulta de la síntesis del IP wrapper es puramente combinacional (minimiza retardo y consumo de recursos lógicos), y soporta la conversión de ciclos de bus Avalon y Wishbone. Sin embargo, no se implementa soporte para conversión de ciclos de bus en modo burst.

En la tabla 8 se muestra la tasa de transferencia promedio obtenida para el sistema de la figura 57, con el detalle de los parámetros del sistema bajo prueba.

CPU	Nios II/f 4kBytes Cache de Instrucciones (sin burst) 4kBytes Cache de Datos (sin burst)
Software	Programa de control y diagnóstico PCI Opción de Menú: (11)
Sistema Operativo	uClinux
Clock Avalon	100 MHz
Clock PCI	33 MHz
Origen de datos	Memoria SRAM plataforma ISIS
Destino de datos	Address Space 1 (BAR 1) dispositivo DUC-II
Mecanismo de transporte desde memoria a PCI Host Bridge	DMA sin modo burst
Tasa de transferencia promedio	14,5 [MBytes/s]
Tasa de error promedio	0 Bytes/bloque
Retardo por transferencia promedio	1,122 [ms]
Tamaño de bloque de datos	17.720 Bytes
Número de experimentos	100
Función Linux usada para obtención de marcas de tiempo	`#include <sys/time.h>` `int gettimeofday(struct timeval *restrict tp, void *restrict tzp)`

Tabla 8: Parámetros de experimento para medir tasa de transferencia PCI

La tasa de transferencia teórica para un sistema PCI 32 bit @ 33 MHz es de 4 Bytes/data * 33M data phases/s = 132 [Mbytes/s]. Luego, la tasa de transferencia obtenida corresponde al 11% del máximo teórico. Los principales factores que inciden en la tasa de transferencia son el acceso de múltiples maestros a memoria, y el retardo de resolución de arbitración. En efecto, el maestro de datos y el maestro de instrucciones de la CPU Nios II necesitan acceder a la misma memoria SDRAM para obtener datos o tomar instrucciones respectivamente, en función de los procesos que ejecuta el sistema operativo, y los datos e instrucciones almacenados en Cache. Además el maestro de lectura del controlador DMA accede a memoria para leer (en unidades de 4 Bytes) el bloque de datos (que puede ser del orden de kBytes) a transmitir por el PCI Host Bridge. Otros maestros que también requieren acceder a memoria para leer o escribir datos son el controlador DMA para transporte de datos desde y hacia la interfaz Ethernet. Por lo tanto, considerando además el maestro Avalon del PCI Host Bridge, existen al menos 6 maestros en el sistema que necesitan acceder al mismo chip de memoria. Dado que el System Interconnect Fabric implementa un solo bloque de arbitración (que consiste en un multiplexor con lógica de control en base a parámetros de prioridad), se produce el fenómeno de contención de bus, es decir, se degrada el rendimiento del sistema a causa de los múltiples maestros que solicitan en forma intensiva al mismo esclavo.

El rendimiento puede mejorarse incorporando soporte para transacciones burst, donde la arbitración se bloquea para el par maestro/esclavo hasta que termine la transferencia, y la dirección es aumentada en forma automática, logrando un uso más eficiente del bus.

En el Anexo A se muestra el resumen del proceso de síntesis del SoPC diseñado en el presente trabajo, donde se detalla el uso de recursos lógicos de los principales componentes del sistema.

5.2 Subsistema PCI del kernel de Linux

La capa de software desarrollada para comunicación entre el core PCI Host Bridge y el subsistema PCI del kernel de Linux, permite la correcta ejecución de las rutinas de inicialización, autoconfiguración, y mapeo de interrupciones. Estas rutinas facilitan la implementación de drivers (compatibles con kernel 2.4), y permiten el uso de herramientas como el paquete *pciutils* (*PCI Utilities*), que es una colección de programas para inspeccionar y manipular la configuración de dispositivos PCI, todo basado en una librería portable llamada *libpci* que ofrece acceso al espacio de configuración PCI. En particular, como parte del proceso de verificación del funcionamiento de las rutinas PCI del kernel se utiliza el comando *lscpi*, que despliega información detallada sobre todos los buses y dispositivos presentes en el sistema.

En el Anexo B se muestra el log del proceso de arranque del sistema operativo uClinux que incorpora la interfaz de software entre el core PCI Bridge y el kernel, corriendo sobre la plataforma ISIS conectada al dispositivo DUC-II por medio del adaptador PMC a PCI.

El desarrollo de software se hace tomando como referencia el subsistema PCI del kernel 2.4 de Linux, es decir, hace uso de estructuras y funciones obsoletas en versiones más modernas del kernel. Luego, presenta la desventaja de dificultar el proceso de implementación de drivers de dispositivos compatibles con el kernel 2.6.

5.3 Placa de adaptación PMC a PCI

La placa de adaptación PMC a PCI permite la correcta comunicación entre placas madres con chipset PCI que poseen un conector PMC 3.3V, 32 bit @ 33 MHz, y tarjetas PCI 3.3V o Universales, 32 bit @ 33 MHz de PC. En particular, permite conectar a la plataforma ISIS (con chipset PCI embebido en dispositivo FPGA) el dispositivo PCI DUC-II.

La versión inicial del adaptador PMC a PCI presentó problemas de ancho de banda, integridad de señal y ruido, que fueron solucionados por medio de la implementación de planos de tierra en ambas caras del PCB, usando láminas de cobre cubriendo las zonas con mayor densidad de señales. También se incorporó una resistencia de pull up en el pin de clock del conector PCI, que determinó una mejora en el tiempo de subida del flanco positivo del clock. Estas modificaciones permitieron al diseño original operar el dispositivo DUC-II de forma estable.

En la figuras 59 y 60 se muestran las forma de onda del clock PCI capturadas con osciloscopio en el pin B16 del conector PCI del adaptador, conectado a la plataforma ISIS. La figura 59 corresponde al caso donde el adaptador opera con el conector PCI en vacío. La figura 60 corresponde al caso donde el dispositivo DUC-II se encuentra conectado.

En el segundo caso la señal de clock aumenta su frecuencia en un 0,33% y su voltaje Pk-Pk en 0,11%. El overshoot aumenta de 550 a 800 [mV] (+45%). Como se aprecia en la figura 60, al conectar el dispositivo PCI se degrada el ancho de banda del adaptador, es decir, se reduce el espectro de la señal original a un contenido casi sinusoidal. Esto se explica por el carácter de carga altamente capacitivo del dispositivo PCI, y que por lo tanto presenta baja impedancia a alta frecuencia.

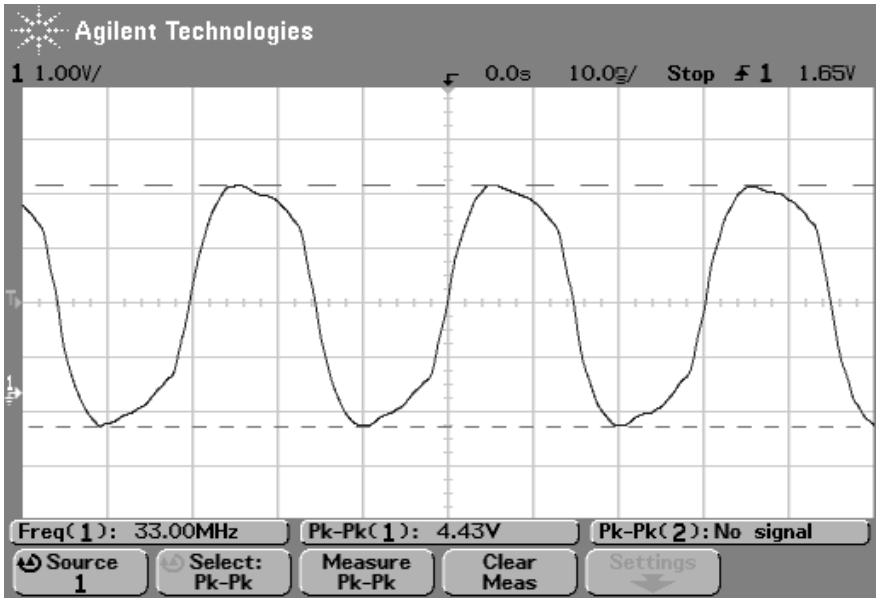


Figura 59: Señal de clock PCI con adaptador PMC a PCI operando en vacío.

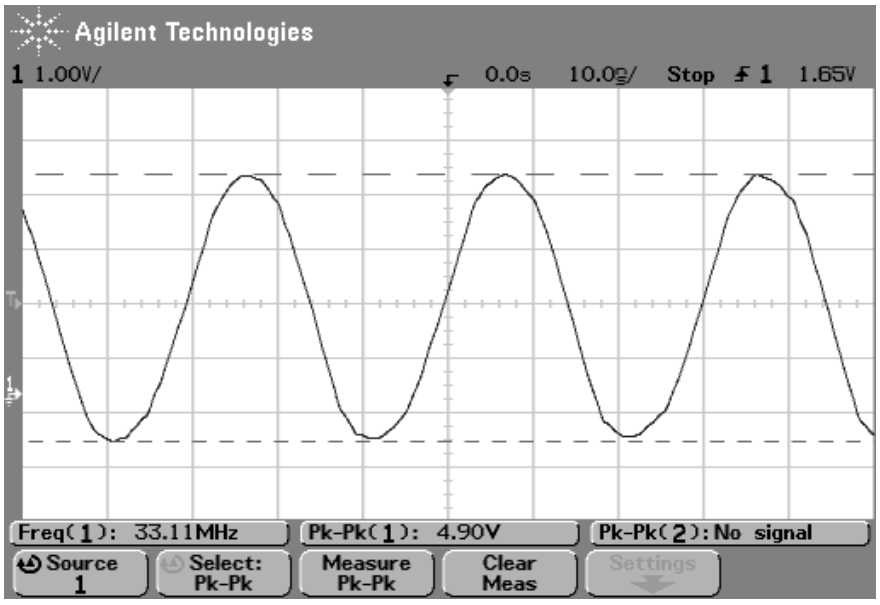


Figura 60: Señal de clock PCI con dispositivo DUC-II conectado

5.4 Operación del equipo GSD-21 Exgine

Como ejemplo se señal de datos, en la figura 61 se muestra la forma de onda capturada en el pin B24 AD[25] del conector PCI del adaptador, en el equipo GSD-21 Exgine durante operación normal (transmisión de radio digital). La señal de la figura presenta las características típicas de una señal PCI de placa madre.

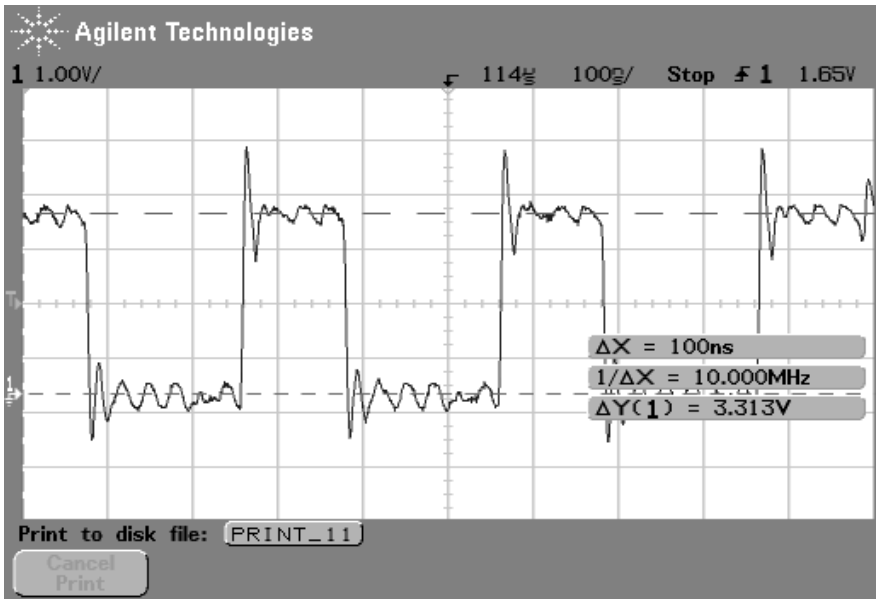


Figura 61: Señal AD[25] en conector PCI durante operación del equipo GSD-21

La superposición de oscilaciones amortiguadas sobre la señal digital se conoce como *ringing*. Cuando la impedancia de línea es mayor que la impedancia de la fuente, el coeficiente de reflexión visto hacia la fuente es negativo, lo que produce el fenómeno de *ringing*. Esta condición también se conoce como *línea de transmisión sobrecargada*. Un factor que incide en esta condición es la carga principalmente capacitiva que impone el dispositivo PCI, y las capacidades parásitas introducidas por la punta del osciloscopio sobre el pin del conector (lo que determina cierto grado de corrupción de la medición).

El protocolo de prueba descrito en la sección 4.8.2.2 arroja una tasa de error nula después de 24 horas de operación, donde el bus transporta bloques de datos a una tasa promedio de 14,5 [Mbytes/s]. Por lo tanto, luego de que el bus PCI ha transportado aproximadamente 24 [horas]*3600 [s/hora]*14,5 [Mbytes/s] = 1252 [Gbytes], la tasa de error que arroja la aplicación en modo ver (Bit Error Rate) es cero. Este resultado valida el funcionamiento de todos los sistemas de hardware y software desarrollados en el presente trabajo, y tiene el valor de otorgar soporte PCI a la primera placa madre basada en computación reconfigurable desarrollada en Chile.

6 Desarrollos futuros

Se plantean las siguientes extensiones y líneas de investigación futura al trabajo realizado:

1. *Modificación de IP wrapper del core PCI Bridge para dar soporte a conversión de ciclos de bus en modo burst:*

   Esta modificación tiene la ventaja de hacer un uso más eficiente del bus para transferencias de datos por bloque, lo que determina una mejora sensible en el rendimiento del bus.

2. *Soporte de programación de prioridades para el árbitro PCI:*

   Requiere añadir una interfaz esclavo Avalon al módulo de arbitración, para programación dinámica de la máquina de estados del árbitro con criterio de prioridad, considerando la información contenida en el registro Max_Lat del espacio de configuración de los dispositivos PCI presentes en el bus.

3. *Rediseño de la interfaz entre el core PCI Bridge y el kernel de Linux para compatibilidad con kernel 2.6:*

   Facilita la implementación de drivers compatibles con versiones más recientes del kernel de Linux.

4. *Rediseño de placa de adaptación PMC a PCI con PCB de cuatro capas:*

   La incorporación de planos de tierra mejora los problemas de integridad de señal, ruido, *crosstalk*, y ancho de banda. También evaluar la incorporación de buffers para adaptación de voltajes de 5V a 3.3V, para compatibilidad con dispositivos PCI 5V, 32 bit @ 33 MHz.

7 Conclusiones y contribuciones

El presente trabajo otorga soporte para conectividad con dispositivos PCI 3.3V 32 bit @ 33 MHz, a la primera placa madre basada en computación reconfigurable desarrollada en Chile. El trabajo aporta la implantación de un chipset PCI embebido en un dispositivo FPGA, el soporte necesario para operación con el sistema operativo embebido uClinux, y una aplicación para control y diagnóstico del hardware. Además, se aporta un nuevo hardware que brinda una solución de bajo costo a la incompatibilidad entre los estándares mecánicos PCI Mezzanine Card y PCI convencional de PC.

Uno de los aportes fundamentales es la implantación de un IP core con interfaz de bus Wishbone en un SoPC basado en arquitectura de comunicación Avalon System Interconnect Fabric (solución que prácticamente no existe en el mercado). Además, los requerimientos del sistema exigen que el IP core PCI Bridge de Opencores sea implementado en modo Host, estando disponible solamente con pruebas de operación en modo Guest. Luego, el presente trabajo aborda en forma exitosa la tarea de poner un funcionamiento un núcleo de hardware que no cuenta con procesos de validación acorde a los requerimientos de la aplicación.

El presente trabajo se integra como parte fundamental del equipo de radiodifusión digital de tercera generación GSD-21 Exgine, desarrollado en Chile por Continental Lensa S.A. El núcleo de hardware del equipo lo constituye la plataforma ISIS integrada con el dispositivo PCI DUC-II (*Next Generation Digital Up Converter*), por medio de los sistemas de hardware y software desarrollados.

Una de las ventajas de las plataformas basadas en dispositivos reconfigurables es la flexibilidad y reducción de costo que otorga el uso de dispositivos y chipsets controladores de bus en la forma IP cores, que pueden ser parametrizados, modificados o rediseñados en función de los requerimientos de la aplicación. En efecto, una misma plataforma puede soportar una variedad de sistemas con distintas interfaces de bus, lo que permite adaptarse fácilmente a los cambios tecnológicos y necesidades de la industria.

El presente trabajo disminuye la brecha de costo y conocimiento para el desarrollo de modernos sistemas basados en computación reconfigurable, que hacen uso de estándares industriales basados en PCI tales como PCI-X, PXI, PCIe, CompactPCI, PISA, PMC, Small PCI, entre otros. En particular, la capacidad de integrar SoPCs de alta complejidad con dispositivos y equipos compatibles con estándares industriales basados en PCI, abre un vasto universo de aplicaciones que abarca desde el control de sistemas, sistemas de radio y televisión digital, procesamiento de señales en tiempo real, predicción de fallas, telemetría, etc.

Referencias

[1] IEEE P1386.1-2001. *Standard Physical and Environmental Layers for PCI Mezzanine Cards*, 2001.

[2] R. Passerone, J. Rowson and A. Sangiovanni-Vicentelli. *Automatic synthesis of interface between incompatible protocols*. In Proc. of the 35th Design Automation Conference (DAC98), San Francisco, CA, USA, pp. 8-13, 1998.

[3] A. Rajawat, M. Balakrishnan, A. Kumar. *Interface sythesis: Issues and Approaches*. In Proc. of the 13th International Conference on VLSI Design, Calcutta, India, pp. 92-97, 2000.

[4] J. Rowson and A. Sangiovanni-Vicentelli. *Interface-based design*. In Proc. of the 34th Design Automation Conference (DAC 97), Anaheim, CA, USA, pp. 178-83, 1997.

[5] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey and A. Sangiovanni-Vicentelli. *Addressing the SoC interconnect woes through communication-based design*. In Proc. of the 38th Design Automation Conference (DAC 2001), Las Vegas, Nevada, USA, pp. 667-672, 2001.

[6] K. Keutzer, S. Malik, R. Newton, J. Rabaey and A. Sangiovanni-Vicentelli. *System level design: Orthogonalization of concerns and platform-based design*. IEEE Trans. on CAD, 2000

[7] G. Cyr, G. Bois and M. Aboulhamid. *Sythesis of communication interface for SoC using VSIA recommendations*. In Proc. of DATE Design Forum 2001, Munich, Germany, pp. 155-159, 2001.

[8] R. Damasevicius, V. Stuikys. *Wrapping of soft IPs for interface-based design using heterogeneous metaprogramming*. INFORMATICA, Vol. 14, No 1, pp. 3-18, Lithuanian Academy of Sciences, Vilnius, Lithuania, 2003.

[9] H. Ossher and P. Tarr. *Multi-dimensional separation of concerns and the hyperspace approach*. In M. Aksid (Ed.), Software Architectures and Component Technology: the state of the art in software development. Kluwer Academic Publishers, 2000.

[10] K. S. Chung, R. K. Gupta, C. L. Liu. *An algorithm for synthesis of system-level interface circuits*. In Proc. of IEEE International Conference on CAD, San Jose, California, USA, pp. 442-447, 1996.

[11] B. Park, H. Choi, I. Park and C. Kyung. *Synthesis and optimization of interface between ip's operating at different clock frequencies*. In Proc. of the International Conference on Computer Design 2000, Austin, Texas, USA, pp. 519-524, 2000.

[12] R. Passerone, L. de Alfaro, T. Henzinger and A. Sangiovanni-Vicentelli. *Convertibility verification and converter synthesis: two faces of the same coin*. In Proc. of the International Conference on CAD 2002, San Jose, California, USA, pp. 132-139, 2002.

[13] D. Gajski and S. Narayan. *Interfacing incompatible protocols using interface process generation*. In the 32nd ACM/IEEE Design Automation Conference, San Franciso, California, USA, pp. 468-473, 1995.

[14] J. Smith and G. Micheli. *Automated composition of hardware componens*. In Proc. of the 35th Design Automation Conference (DAC) 1998, San Franciso, California, USA, pp. 14-19, 1998.

[15] PCI Special Interest Group. *PCI Local Bus Specification, Revision 3.0*, 2002.

[16] Opencores. *PCI bridge IP core datasheet*, 2002.

[17] Opencores. *PCI IP core data book*, 2002.

[18] Opencores. *PCI IP core design document*, 2002.

[19] Opencores. *PCI IP core specification, Rev. 1.2*, 2004.

[20] Altera. *Cyclone II Device Handbook*, 2008.

[21] Altera. *Nios II Processor Reference Handbook*, 2008.

[22] Altera. *Nios II Development Board Cyclone II Edition Rerefence Manual*, 2007.

Anexos

A. Resumen de flujo de análisis y síntesis

A.1 Resumen de análisis y síntesis

Quartus II version	8.1 Build 163 10/28/2008 SJ Web Edition
Family	Cyclone II
Device	EP2C20F484C6
Met timing requirements	Yes
Total Logic Elements	10.374 / 18.752 (52%)
- Total Combinational Functions	8.411 / 18.752 (45%)
- Dedicated Logic Registers	5.999 / 18.752 (32%)
Total registers	6.22
Total pins	186 /315 (59%)
Total virtual pins	0
Total memory bits	87.512 / 239.616 (37%)
Embedded Multiplier 9-bit elements	4 / 52 (8%)
Total PLLs	3 / 4 (75%)

A.2 Resumen de uso de recursos por entidad

Entity	LC Comb.	LC Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18
cpu	2449 (2197)	1751 (1564)	81.54	4	0	2
pci_top	2359 (11)	2552 (0)	0	0	0	0
pci_arbiter_top	43 (1)	21 (3)	0	0	0	0

B. Registro de arranque de uClinux

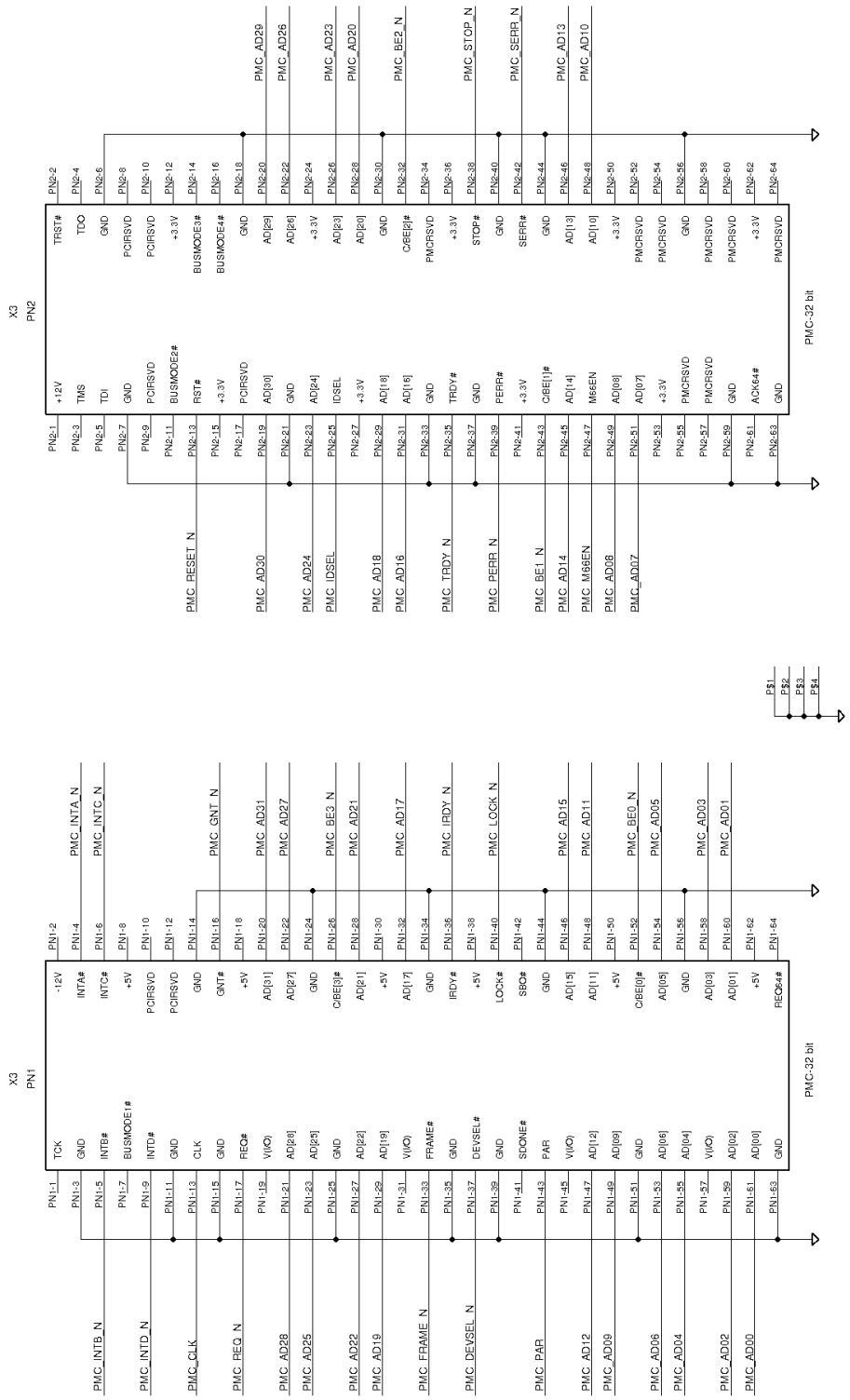
```
Uncompressing Linux... Ok, booting the kernel.
Linux version 2.6.27-rc6 (enroman@tech) (gcc version 3.4.6) #116 PREEMPT Tue Nov 18 18:25:13 CLST
2008
\0x0f
uClinux/Nios II
Built 1 zonelists in Zone order, mobility grouping off.  Total pages: 8128
Kernel command line:
PID hash table entries: 128 (order: 7, 512 bytes)
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory available: 29844k/2627k RAM, 0k/0k ROM (1645k kernel code, 982k data)
Calibrating delay loop... 42.18 BogoMIPS (lpj=210944)
Mount-cache hash table entries: 512
net_namespace: 256 bytes
NET: Registered protocol family 16
Autoconfig PCI channel 0x001d7c84
Scanning bus 00, I/O 0x83100000:0x83200000, Mem 0x83200000:0x84000000
00:00.0 Class 0402: a5a5:f0f1
Mempci range check OK: bus num = 0 , slot = 0 at 0x83200000 [size=0x100]
I/Opci range check OK: bus num = 0 , slot = 0 at 0x83100000 [size=0x100]
Mempci range check OK: bus num = 0 , slot = 0 at 0x83200400 [size=0x400]
Mempci range check OK: bus num = 0 , slot = 0 at 0x83220000 [size=0x20000]
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP reno registered
NET: Registered protocol family 1
io scheduler noop registered
io scheduler deadline registered (default)
ttyJ0 at MMIO 0x2011888 (irq = 5) is a Altera JTAG UART
ttyS0 at MMIO 0x2011820 (irq = 3) is a Altera UART
console [ttyS0] enabled
smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre <nico@cam.org>
eth0: SMC91C11xFD (rev 2) at 82000300 IRQ 2 [nowait]
eth0: Ethernet addr: 08:00:27:a6:c0:53
TCP cubic registered
NET: Registered protocol family 17
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
Freeing unused kernel memory: 696k freed (0x1e2000 - 0x28f000)

Shell invoked to run file: /etc/rc
Command: hostname uClinux
Command: mount -t proc proc /proc
Command: mount -t sysfs sysfs /sys
Command: mount -t usbfs none /proc/bus/usb
mount: mounting none on /proc/bus/usb failed: No such file or directory
Command: mkdir /var/tmp
Command: mkdir /var/log
Command: mkdir /var/run
Command: mkdir /var/lock
Command: mkdir /var/empty
Command: ifconfig lo 127.0.0.1
Command: route add -net 127.0.0.0 netmask 255.0.0.0 lo
Command:
Command: ifconfig eth0 hw ether 080027A6C053
Command: ifconfig eth0 192.168.0.201
eth0: link down
Command: route add default gw 192.168.0.1
Command: inetd &
[23]

Command:
Command:
```

```
Command: cat /etc/motd
Welcome to
```

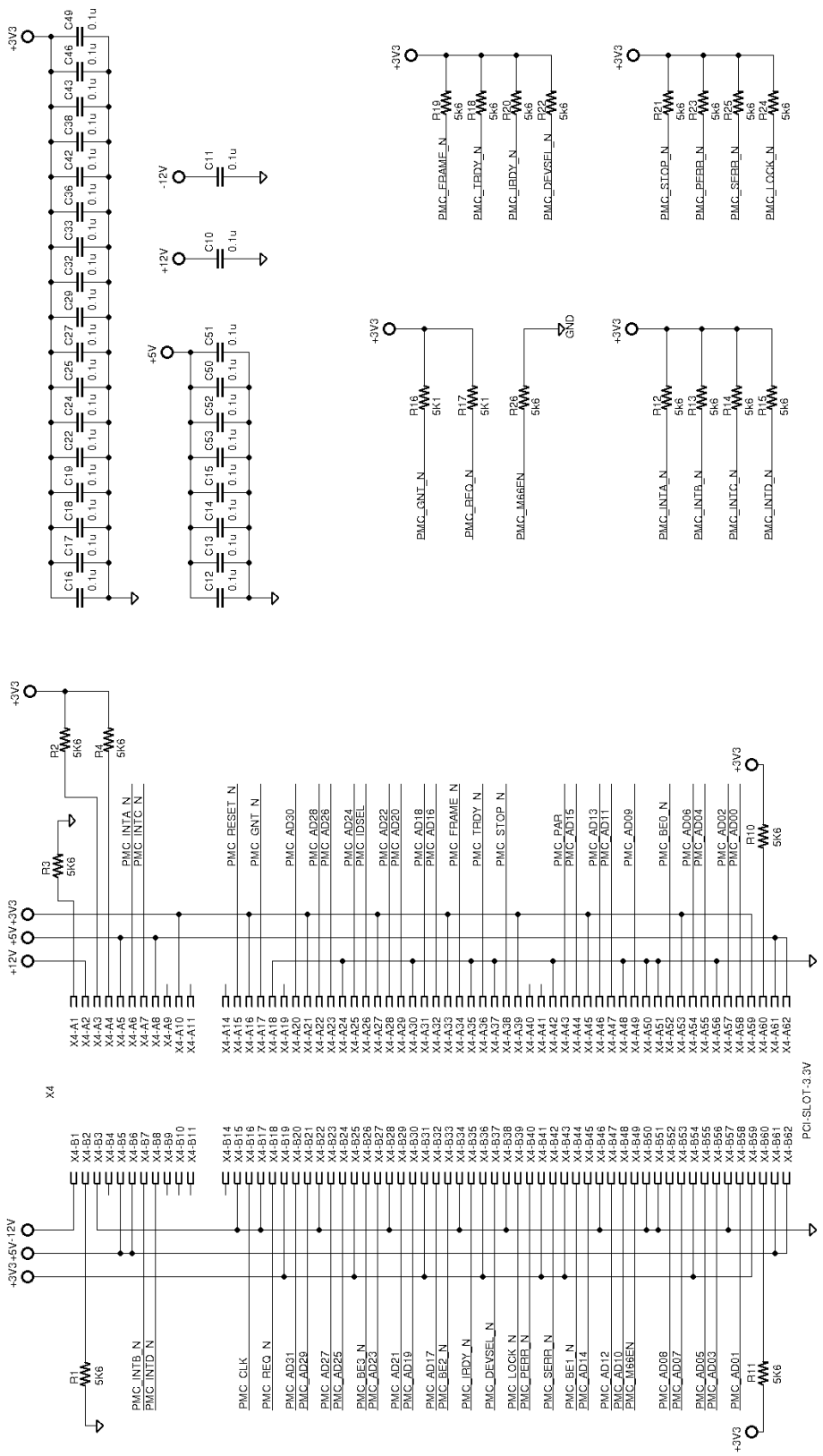
```
For further information check:
http://www.uclinux.org/

Command:
Command:
Execution Finished, Exiting

Sash command shell (version 1.1.1)
/> /> eth0: link up, 100Mbps, full-duplex, lpa 0x45E1
cd /proc/bus/pci
/proc/bus/pci> ls
00
devices
/proc/bus/pci> cat devices
0000\0x09a5a5f0f1 83200000    83100000    83200400    83220000    0     0     0
                        100         100         400       20000
```

C. Esquemáticos de Adaptador PMC a PCI

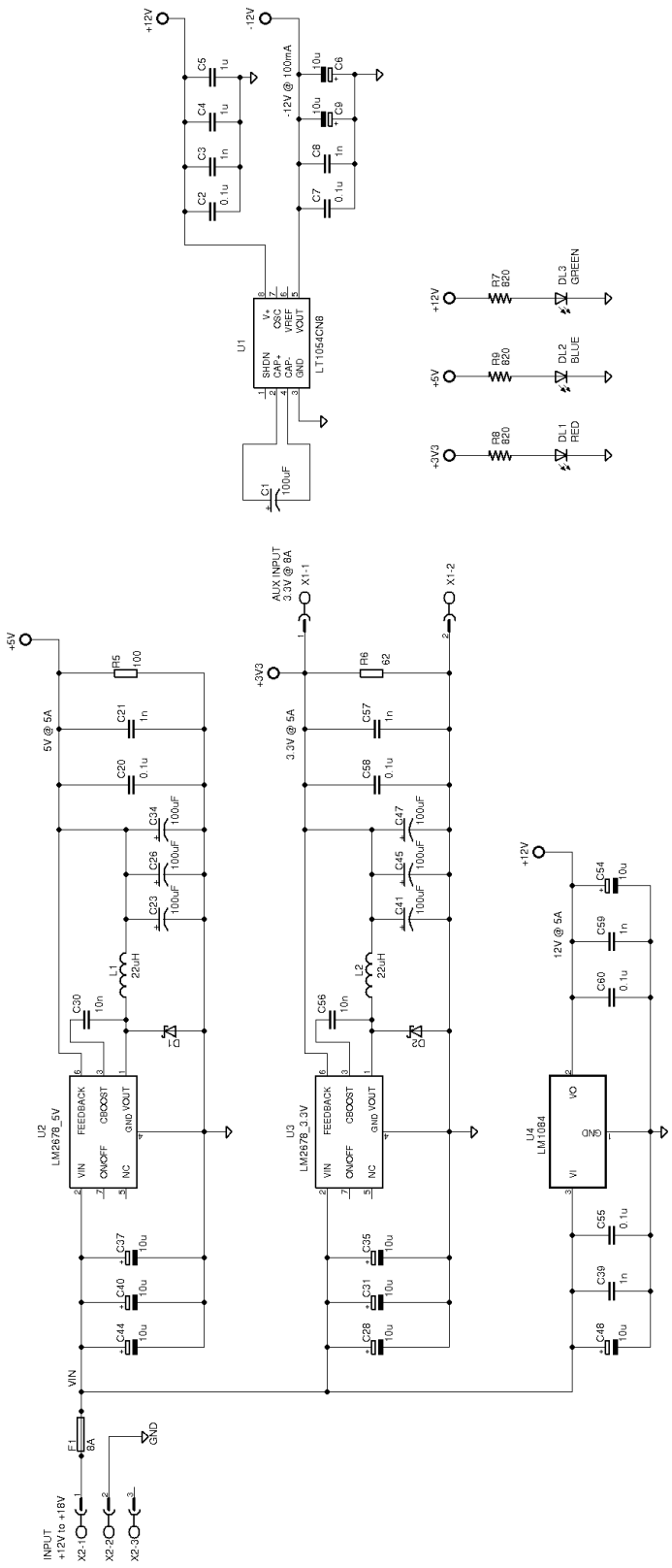
C.1 Conector PMC 32 bit



C.2 Conector PCI 3.3V 32 bit @ 33 MHz



C.3 Fuente de poder PCI



D. Protocolos de prueba Adaptador PMC a PCI

D.1 Validación de PCI Power Rails

Condición	Si/No
Al energizar la placa de adaptación en forma aislada (en vacío), los pines de voltaje del conector PCI son correctos? (+3.3V, +5V, +/-12V y V(I/O) = +3.3V)	
Todos los pines GND correctamente a tierra en conector PCI?	
Todos los pines GND correctamente a tierra en conectores PN1 & PN2?	
Todos los pines de voltaje del conector PN1 & PN2 (+3.3V, +5V y +/-12V) están abiertos (no conectados)?	
Pines en dispositivos reguladores de voltaje presentan voltajes correctos? (U1, U2, U3)	

D.2 Chequeo de recursos centrales PCI

Condición	Si/No
Las señales FRAME#, TRDY#, IRDY#, DEVSEL#, STOP#, SERR#, PERR#, LOCK# tienen resistencias pull-up de 5.6k?	
Las señales INT{A,B,C,D}_n tienen resistencias pull-up de 5.6k?	
Las señales TMS y TDI tienen resistencias pull-up de 5.6k?	
Las señales GNT_n y REQ_n tienen resistencias pull-up de 5.1k?	
Las señales M66EN, BUSMODE1_n, TRST_n y TCK tienen resistencias pull-down de 5.6k?	
El pinout del conector PCI y su interconexión con el conector PMC se ajusta a la especificación PCI?	

D.3 Integridad de señales PCI

Una vez validadas las fuentes de poder y *PCI Central Resources* del adaptador PMC a PCI, pueden verificarse por medio de un Analizador Lógico las siguientes condiciones de integridad de señal para el IUT (*Implementation Under Test*) definido (sistema ISIS-DUC-II).

Condición	**Si/No**
Todas las señales PCI están controladas por *PCI compliant components*? (por ejemplo, pines FPGA configurados como *standard PCI pins*)	
Periodo mínimo de señal CLK es de 30 ns? (PCI 33 MHz)	
CLK skew entre los dos dispositivos PCI es menor que 2 ns? (PCI 33 MHz)	
CLK llega al componente con al menos 11 ns de high/low time? (PCI 33 MHz)	
Peak-to-Peak CLK swing es de al menos 2V (0.4V a 2.4V)?	
Ambos flancos de señal RST# son monotónicos en el rango de switching?	
Alimentación es estable por al menos 1 ms antes de la desactivación de RST#?	
Alimentación es valida por al menos 100 ms antes de la desactivación de RST#?	
CLK es estable por al menos 100 us antes de la desactivación de RST#?	
El bus permanece en estado *idle* por al menos 5 clocks después de la desactivación de RST#?	
Tprop desde cualquier driver (*PCI compliant*) a cualquier receptor PCI debe ser menor o igual a Tcyc - (Tval + Tskew + Tsu) = 10 ns para Tcyc = 30 ns (PCI 33 MHz)	

E. Dimensionamiento de pistas PCB

El siguiente cálculo se basa en la aplicación del estándar general para el diseño de circuitos impresos ANSI-IPC 2221 desarrollado por la IPC (Interconnecting and Packaging Electronic Circuits).

Datos de entrada:

Para calcular el ancho de una determinada pista se necesitan 3 datos:

1. Imax [A]: Corriente máxima que puede circular por la pista.
2. temp_rise [ºC]: Incremento máximo permitido de temperatura que puede soportar esa pista, respecto a la temperatura ambiente. Por ejemplo, si el circuito se diseña para que funcione a una temperatura ambiente de 50ºC y el diseño contempla una temperatura máxima de pista de 60ºC, entonces el incremento máximo de temperatura permitido debe ser de 60ºC - 50ºC = 10ºC.
3. thickness [mils] o [oz/ft²] (1 [oz] = 1.378 [mils] = 35 [um]): Altura o grosor de la pista.

   Las medidas estándar típicas de grosor del cobre (capas externas) son 1/2, 1 y 2 [oz] (17.5, 35 y 70 [um]) y 1/4, 1/2, 1 y 2 [oz] (capas internas).

Cálculo de ancho de pista:

Las ecuaciones que relacionan el ancho de pista con el área efectiva Area_ef (que varia inversamente con la temperatura, es decir, a mayor temperatura menor área efectiva) y el grosor de la pista (width) son las siguientes:

(1) width = Area_ef/[(thickness [oz/ft²])*1.378] [mils]

(2) Area_ef = [Imax/(k1*temp_rise^k2)]^(1/k3) [mils²]

Las constantes k1, k2 y k3 están definidas por el estándar y tienen los siguientes valores:

- k1 = 0.0150 para *capas internas* y 0.0647 para *capas externas*.
- k2 = 0.5453 para *capas internas* y 0.4281 para *capas externas*.
- k3 = 0.7349 para *capas internas* y 0.6732 para *capas externas*.

Por ejemplo, se requiere dimensionar el ancho de pista externa para soportar una corriente máxima de 2 [A] con un incremento de temperatura de 10ºC (sobre la temperatura ambiente). El grosor del cobre es de 1 [oz/ft²].

Aplicando la fórmula se obtiene un área efectiva de 37.8 [mils²] para pista externa y 141 [mils²] para pista interna. Luego, se obtiene un ancho de pista de 27.4 [mils] para pista externa y de 102 [mils] para pista interna.

F. Cálculo de disipadores de calor

Las tres formas en que un dispositivo puede disipar calor son *radiación*, *conducción* y *convección*. Los PCBs usan disipadores de calor para mejorar la disipación de calor. La transferencia de energía termal de los disipadores de calor se debe a la baja *resistencia termal* entre el disipador y el aire.

La resistencia termal es una medida de la habilidad de una substancia para disipar calor, o la eficiencia de la transferencia de calor a través de la interfaz entre dos medios (por ejemplo, interfaz juntura-package, interfaz package-aire). Un disipador con área grande y buena circulación de aire provee una "baja resistencia termal" o inversamente una "conductividad termal grande". La resistencia termal se mide en ºC/W y como notación se usa la letra griega Theta con subíndice indicando la interfaz (por ejemplo, subíndice JC se refiere a interfaz juntura-case). Es importante notar que la resistencia termal es un parámetro que depende de muchas variables, tales como la disposición física del dispositivo, tamaño del PCB, grosor del cobre, entre otras. Un disipador de calor ayuda a un dispositivo a operar en el rango de temperatura recomendado. El calor del dispositivo fluye desde la juntura (J) hacia el package o case (C), y desde el case hacia el aire (A).

Parámetros

Parámetro	Resistencia termal	Unidades	Descripción
Theta_JA	Junction-to-Ambient	ºC/W	Especificada en datasheet
Theta_JC	Junction-to-Case	ºC/W	Especificada en datasheet
Theta_CS	Case-to-Heat Sink	ºC/W	Correspondiente a material de interface (e.g., pasta disipadora)
Theta_CA	Case-to-Ambient	ºC/W	
Theta_SA	Heat Sink-to-Ambient	ºC/W	Especificada por fabricante del disipador
TJ	Temperatura de juntura	ºC	Especificada bajo las condiciones de operación recomendadas por el fabricante del dispositivo
TJmax	Temperatura max de juntura	ºC	Especificada bajo las condiciones de operación recomendadas por el fabricante del dispositivo
TA	Temperatura ambiente	ºC	Temperatura del aire en el ambiente local donde se ubica el dispositivo
TS	Temperatura disipador	ºC	
TC	Temperatura del case	ºC	
P	Potencia	W	Potencia total del dispositivo. Usar el valor estimado para dimensionar el disipador

Ecuaciones

(1) *Dispositivo sin disipador*: theta_JA = theta_JC + theta_CA = (TJ – TA)/P

(2) *Dispositivo con disipador*: theta_JA = theta_JC + theta_CS + theta_SA = (TJ - TA)/P

Procedimiento para calcular disipador

1. Para determinar la necesidad de un disipador de calor, calcular la temperatura de juntura y compararla con la máxima temperatura aceptable especificada por el fabricante:

   TJ = TA + P*theta_JA > TJmax => se requiere un disipador

2. Seleccionar el disipador usando las siguientes ecuaciones:

   theta_JA = theta_JC + thetaCS + theta_SA = (TJ - TA)/P
   theta_SA = (TJmax - TA)/P - theta_JC – theta_CS