
# DISEÑO DE UNA PLATAFORMA DE LECTURA, TEST Y CARACTERIZACIÓN PARA ROICS DE RAYOS X BASADA EN EL MICROPROCESADOR LEON

Memoria del Proyecto Final de Carrera
de Ingenieria Informática
realizado por
**Sergio Morlans Iglesias**
y dirigido por
**Lluís Terés Terés**
**Ricardo Martínez Martínez**
Bellaterra, 15 de Septiembre de 2009

Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria

El sotasignat,.....
Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA**:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per

I per tal que consti firma la present.

Signat:.....
Bellaterra,.....de.....de 2009

# Agradecimientos

A mi familia por todos estos años de apoyo y comprensión.

A Mireia, por estar siempre a mi lado. Por tener paciencia conmigo y soportarme.

A Lluís Terés por la confianza depositada hacia mi persona para el desarrollo de este proyecto. Por la oportunidad dada, de poder estar en el CNM-IMB (CSIC), durante estos meses.

A Ricardo Martínez porque sin su ayuda, sus consejos y supervisión este proyecto no habría sido posible.

A mis compañeros de despacho (Álvaro, Justo, Roger, Ricardo) y al resto del departamento del ICAS en el CNM-IMB (CSIC), por el apoyo dado, por el buen trato recibido y por haberme hecho sentir como uno más desde el primer día.

A todos los compañeros de universidad con los que ha pasado tantas horas de trabajo, sufrimiento y buen rollo, sin ellos no lo habría conseguido. Ellos han sido fundamentales para mi crecimiento tanto a nivel personal como educacional.

# Índice general

# Índice de figuras

# 1. Introducción

En este capítulo se hará una breve exposición sobre el proyecto final de carrera titulado "*Diseño de una plataforma de lectura, test y caracterización para Read Out Integrated Circuits (ROICs) de rayos X basada en el microprocesador LEON"*.Se situará dentro del marco de un proyecto mayor que se está realizando en el Centro Nacional de Microelectrónica (CNM)-Instituto de Microelectrónica de Barcelona (IMB) (Consejo Superior de Investigaciónes Científicas (CSIC)) y que consiste en la realización de un sistema de biopsia mamaria en tiempo real.
En la actualidad, en el mercado podemos encontrar diferentes productos que ofrecen el sistema de rayos X integrado dentro de la propia tabla de biopsia, permitiendo localizar el tejido a analizar mediante la adquisición de dos imágenes. Éstas se toman con un cierto ángulo generando una localización tridimensional. La forma de obtener las imágenes actualmente es manual ya que el médico ha de ir tomando las imágenes cada cierto periodo de tiempo para ver si la aguja de biopsia va por la trayectoria correcta. Lo que se pretende es conseguir un sistema que permita obtener vídeo en tiempo real del objeto a biopsiar y de la trayectoria de la aguja. Con este sistema se pretende reducir el tiempo que la operación durará y obtener mejores resultados ya que el médico verá las imágenes en tiempo real.
Este sistema se basa en la toma de imágenes mediante el ROIC Medipix2 MXR una evolución del Medixpix2 realizado por el Organización Europea para la Investigación Nuclear (CERN) como un proyecto de colaboración de ámbito europeo entre varios institutos de física de altas energías. Una vez el ROIC tome las imágenes, necesitaremos un sistema de lectura, test y caracterización del chip Medipix2 MXR, que lea esos datos y los envíe a la aplicación del ordenador donde se interpretarán los datos leídos del ROIC. Para poder realizar este sistema nos hemos de adentrar en el mundo de los sistemas encastados, los System-On-Chips (SOCs), las Field Programmable Gate Arrays (FPGAs) y los microcontroladores, tal como iremos viendo en los próximos apartados.

## 1.1 Objetivos y Motivaciones

En este capítulo vamos a explicar los objetivos y las motivaciones a cumplir durante el proyecto. El objetivo a cumplir es diseñar un microcontrolador basado en el microprocesador LEON3 sobre una FPGA de Altera (Stratix II) en una plataforma de desarrollo (Nios II Development Board), con los periféricos necesarios para poder establecer la comunicación entre el sistema y el ordenador. Para la comunicación entre el ordenador y el sistema generado utilizaremos el USB.
El hecho de usar el USB viene marcado porque se necesita poder transmitir datos a una velocidad elevada, también se ha de tener en cuenta que es un estándar que hoy en día se encuentra en cualquier ordenador. Por este motivo parte del desarrollo del proyecto está centrado en adaptar un controlador USB al sistema. Este microcontrolador formará parte de un sistema de lectura, test y caracterización para el ROIC Medipix2 MXR.

A continuación se hará un pequeño listado de los hitos del proyecto y una breve explicación sobre que consiste cada uno:

* **Setup del entorno de desarrollo del microprocesador LEON3:** Familiarización con el sistema, así como su configuración básica y las señales empleadas por el bus AMBA 2.0
* **Adaptación de un controlador USB:** Adaptación del controlador USB mediante un wrapper a las señales del bus AMBA 2.0 de un controlador que trabaja con señales del bus Avalon y sincronización del sistema a nivel de clock ya que el controlador trabaja a una velocidad menor que el sistema.
* **Desarrollar el firmware del microcontrolador:** Generación de un programa en C para la lectura, test y caracterización del Circuito Integrado (CI) Medipix2 MXR. Este programa será

el que contenga la funcionalidad para establecer la comunicación adecuada con la aplicación DMNewGUI (aplicación que contiene la funcionalidad necesaria para trabajar con el Medipix2).
* **Diseño de un microcontrolador basado en el LEON3 con la funcionalidad necesaria para la lectura, test y caracterización del CI Medipix2 MXR:** Adición del controlador al sistema y programación de la PROM con el firmware desarrollado.
* **Diseño del software necesario para el control del sistema:** Generación de una aplicación para el ordenador con la funcionalidad deseada.

La finalidad de este proyecto es generar un sistema que no dependa de la tecnología de la FPGA (en este caso usamos una Stratix II, pero podría ser otra), que no se tenga que pagar licencias por usar las diferentes IPs del sistema como sucede en el caso del NIOS II, sino que tengas un sistema montado mediante un conjunto de IPs (grlib library) con licencia General Public License (GPL) y basado en el LEON3.

## 1.2 Estructura de la memoria

En esta memoria iremos desgranando capítulo a capítulo todo el desarrollo del proyecto, veremos los conceptos más importantes que nos situarán en contexto el proyecto así como el entorno de desarrollo del sistema y los pasos seguidos hasta llegar al resultado final.

* **Capítulo 1 Introducción:** Capítulo en el que nos encontramos, sirve de explicación inicial del proyecto, de sus objetivos y de la estructura de la memoria.
* **Capítulo 2 Conceptos Teóricos:** Explicación de todos aquellos conceptos necesarios para situar el proyecto dentro de contexto. Haremos una breve explicación sobre Medipix, diseños basados en plataforma, IPs, SOC, Hardware Description Lenguage (HDL)
* **Capítulo 3 Entorno de desarrollo:** En este capítulo encontraremos todos aquellos elementos que tendremos en el sistema, desde la placa de desarrollo, el entorno de librerias grlib, las características del LEON3 y del bus AMBA...
* **Capítulo 4 Setup del LEON3:** Pasos seguidos hasta conseguir bajar un sistema a la FPGA y ejecutar un programa. Desde la configuración inical, la primera simulación hasta programar la memoria PROM y ejecutar el programa.
* **Capítulo 5 Diseño del sistema:** Pasos seguidos para agregar el wrapper y el controlador USB para generar el sistema final.
* **Capítulo 6 Problemas encontrados:** Contratiempos encontrados durante el modelado del sistema.
* **Capítulo 7 Resultados:** Objetivos conseguidos durante la realización del proyecto. Aquí se va enumerando los diferentes hitos conseguidos, hasta el resultado final.
* **Capítulo 8 Conclusiones:** Comentario sobre lo aprendido y futuras evoluciones que podría tener el proyecto realizado.
* **Acrónimos:** Lista de Acrónimos de la memoria.
* **Bibliografía y Referencias:** Documentación en papel y electrónica usada de apoyo para el desarrollo del proyecto.
* **Apéndices:** Información de apoyo sobre algunos aspectos del sistema.

# 2. Conceptos teóricos

## 2.1 Medipix

MediPix es un CI desarrollado en el CERN. La base científica radica en la detección de radiaciones ionizantes diseñadas para aplicaciones en medicina.

Es un CI de 256 x 256 (65536) píxeles (Figura 2.1) con un área sensible de 2 $cm^2$ y donde cada píxel es un detector individual de alta precisión capaz de detectar un fotón y que tiene un tamaño de 55 micras x 55 micras. Lo que implica que tenga una resolución espacial de micrómetros y una resolución energética de electronvoltios[1].
Su principal característica se encuentra en el sistema de detección, ya que utiliza el recuento de fotones en lugar de la tradicional integración de la carga. Éste es el hecho diferenciador que permite la completa eliminación de todo tipo de ruido reduciendo de forma ostensible la cantidad de dosis necesarias.
El sistema integración de carga consiste en que un píxel detecta los protones/electrones (dependiendo de la polarización) que le llega y los envia como pulsos mediante un preamplificador para que vaya cargando una capacidad. Cuando la capacidad llega a un valor determinado se incrementa en un contador existente para ello. El problema de la integración por carga es el ruido existente y la corriente de oscuridad. Este último parámetro contribuye significativamente a la carga final, haciendo que el contador se incremente antes de lo que debiera.

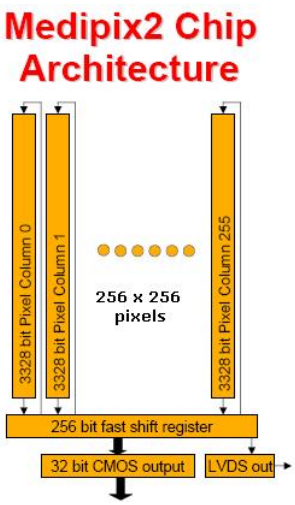



Figura 2.1: Arquitectura de Medipix2

El sistema de recuento de fotones (Figura 2.2) de un píxel detecta los protones/electrones (dependiendo de la polarización) que le llega y lo envía a dos discriminadores a través de un amplificador.
Cada discriminador marca un límite de energía, que si es atravesado, los pulsos entrantes son filtrados. Un discriminador marca el límite a baja y el otro a alta. Con esto se consigue eliminar el ruido existente y quedarnos con los pulsos que están dentro del rango deseado.
Una vez hemos filtrado el ruido, la salida de los discriminadores nos lleva a Basic Windows Descriminator (BDW) que su función es generar un pulso de un anchura fija y discriminar las salidas en función de la polaridad y del modo de operación. Este pulso, a través de un multiplexor, nos hará incrementar el contador. Para este proyecto, nos interesa del Medipix, la vertiente del diagnóstico médico. Dentro del diagnóstico médico nos dedicaremos a trabajar con el diseño de equipos de lectura de rayos X, mediante el chip Medipix2 MXR.

### 2.1.1 Sistemas de Lecturas

En la actualidad ya existen diversos sistemas de lectura, test y caracterización para el ROIC Medipix 2 MXR. Por ejemplo el Medipix2 re-Usable Readout System (MUROS) 2.0 desarrollado en el National Institute for Nuclear Physics and High Energy Physics (NIKHEF) que es un sistema de propósito general o el Dear-MaMa (Detection of Early Markers in Mammography) que es un proyecto desarrollado entre el Institut de Física d'Altes Energies (IFAE) y el CNM y que su funcionalidad va enfocada a las mamografías.

[1] Un ev es la energía adquirida por un electrón al atravesar una diferencia de potencial de 1 voltio

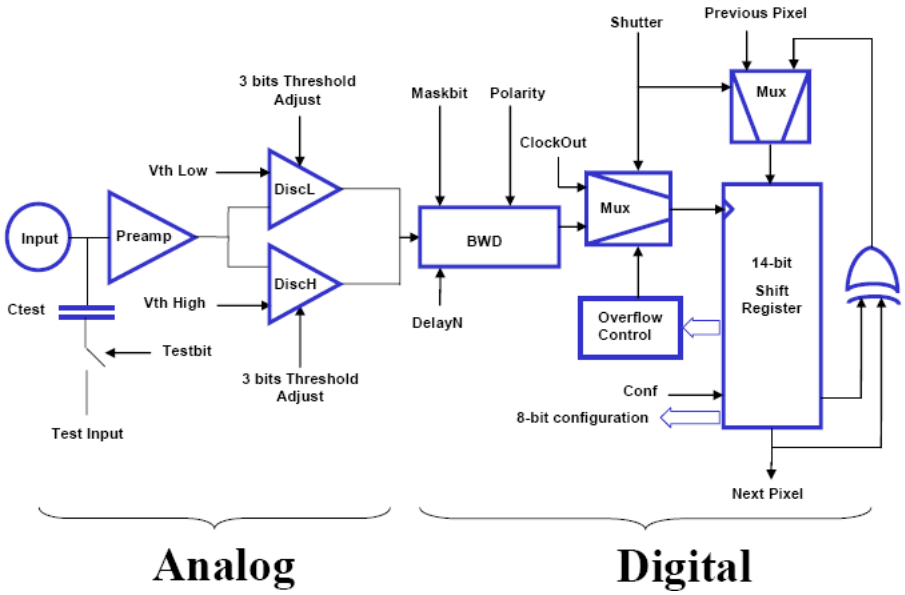



Figura 2.2: Diagrama de bloques del píxel

En estos momentos, entre el CNM y el IFAE se está desarrollando un sistema de lectura, test y caracterización basado en el microprocesador NIOSII. El hecho de usar este microprocesador obliga a pagar unas licencias para poder usar los diferentes componentes del sistema (IPs). El NIOS II al ser una IP de Altera, implica que únicamente se pueden utilizar las FPGAs de esta compañia. De ahí surge la idea de desarrollar un sistema con el LEON3 (que tiene como bus del sistema el AMBA). El sistema basado en LEON3 no se ha de pagar por la licencia de las IPs que componen el sistema, no dependen de la tecnología de la FPGA (se puede generar un sistema para FPGAs de Altera, Xilinx...), y el hecho de tener que adaptar nuestras IPs al bus AMBA deja abierto un gran abanico de posibilidades, ya que el bus AMBA es un bus muy utilizado en los SOC.
La desventaja de usar el LEON3, es que no es tan rápido de configurar el sistema, sintetizarlo y bajarlo a la FPGA, mientras que en NIOS II al utilizar el SOPC builder, se hace todo de más rápido.

## 2.2 Metodologías de diseño

Año tras año la capacidad de integración de FPGAs y Circuito Integrado para Aplicaciones Específicas (ASIC) aumenta, cumpliendo la ley de Moore, lo que provoca el aumento de complejidad y funcionalidad en los sistemas.
La Ley de Moore original nos decía que cada 18 meses se duplicaría el número de transistores en un CI. Más tarde el propio Moore (cofundador de Intel), modificaría esta ley indicando que el ritmo no seria tan elevado y que en lugar de cada 18 meses, sería cada 24 meses. En la Figura 2.3 podemos observar la evolución durante estos últimos 40 años, del número de transistores en un CI, comparado con la ley de Moore.

El hecho de que los sistemas sean cada vez más complejos, provoca la aparición de diferentes metodologías de reutilización. Permitiendo generar sistemas sin tener que empezar de cero, por este motivo, aparecieron diversas metodologías como las IPs tal como veremos a continuación.

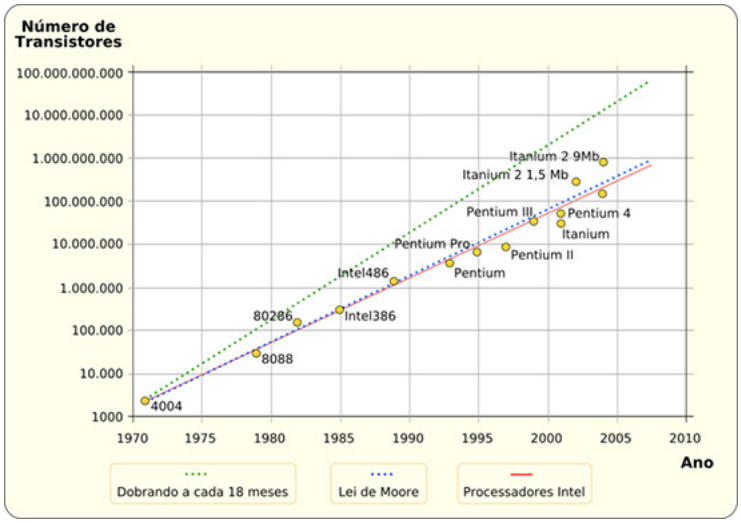



Figura 2.3: Ley de Moore

## 2.3 IPs

A continuación vamos a realizar una breve introducción sobre las IPs. Daremos su definición, que finalidad tienen, donde se utilizan y que influencia tienen en el diseño de sistemas para FPGAs o ASICs.
Una IP es un bloque prediseñado de lógica reusable con una funcionalidad específica que se utiliza en el diseño de sistemas a integrar en FPGAs o ASICs. Las IPs aparecieron por la necesidad de encontrar una metodología que nos permitiese gestionar sistemas complejos, debido a la alta integración. Al dividir un sistema en IPs, hace que la complejidad en el diseño del sistema se reduzca, ya que cada componente cumple con una funcionalidad determinada (bus, memorias, procesadores...).
El diseño por reutilización, se basa en el diseño de sistemas mediante IPs ya existentes y que, por lo tanto, han sido verificados anteriormente, lo que permite reducir el ciclo de diseño del sistema y liberar al diseñador, de generar y verificar algunas de sus partes. Generalmente las IPs suelen estar respaldadas por un proveedor de tecnología (Cypress, Advanced RISC Machines (ARM), Altera...) el cual, te suele facilitar una librería de funciones, para que desde el software, que se ejecutará en el sistema, se pueda interactuar con ella.
Otra característica de trabajar mediante IPs, es la portabilidad de los módulos de un sistema a otro, facilitando su inserción dentro de otro sistema. Para conseguirlo lo que se hace es generar un wrapper[2] para adaptar las diferentes señales de la IP a las señales del bus, esto se hace por culpa de que existe un gran número diferente de buses (Avalon, AMBA...), que complicaría mucho la portabilidad de las IPs de no ser de la existencia de los wrapper y de los dos estándares existentes (Open Core Protocol (OCP), Virtual Socket Interface Alliance (VSIA)). Existen 3 categorías diferentes dependiendo de la flexibidad para sus síntesis (Figura 2.4):

- **Soft cores:** Son IPs descritas en HDL y su código representa entidades hardware sintetizables. Este tipo de componentes son flexibles, ya que pueden ser personalizados e independientes de tecnologia pudiendo ser sintetizados en diferentes dispositivos.
- **Firm cores:** IP definidas como netlist a nivel de puertas lógicas y biestables, optimizadas para un dispositivo o familia de dispositivos. A falta del Place & Route, los recursos del dispositivo ya estan asignados.
- **Hard cores:** Son IPs dependientes de la tecnología, ya que incluyen tanto el layout como su caracterización temporal y por lo tanto no son modificables por el usuario. Son los que se encuentran más optimizados de los tres tipos.

[2] adaptador de dos IP que tienen diferentes señales.

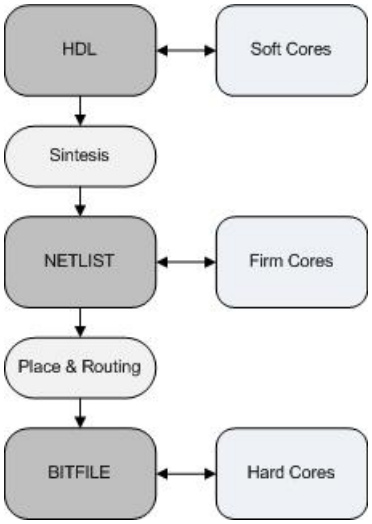



Figura 2.4: Tipos de IPs

## 2.4 Diseño basado en plataforma

Diseño basado en plataforma es un concepto para hacer frente a la creciente presión sobre el time-to-market, el rediseño y costes de fabricación.
Una plataforma es un conjunto de arquitecturas que satisfacen un conjunto de restricciones impuestas que permitan reutilizar componentes hardware y software.
Los principios básicos propuestos son los siguientes:

* El diseño basado en plataforma sigue un flujo de diseño meet-in-the-middle, donde los sucesivos refinamientos de las especificaciones (top-down), se encuentran con las diferentes abstracciones de posibles implementaciones (bottom-up).
* La identificación de las capas definidas con precisión se produce donde el refinamiento y el proceso de abstracción se encuentran. Estas capas dejan pasar bastante información acerca de los niveles más bajos de abstracción para permitir la exploración del espacio de diseño y conseguir una predicción bastante exacta de las propiedades de la aplicación. La información debe ser incorporada a las opciones de diseño de la capa de abstracción. Estas capas de abstracción se llaman plataformas para subrayar su papel en la el proceso de diseño y su solidez.

## 2.5 System-On-Chip

### Introducción

Tal como se ha ido viendo hasta ahora, en esta memoria, la necesidad de reducir los costes de producción, de usar metodologías de rediseño y de cumplir con el time-to-market, hace que vayan apareciendo diferentes elementos como las IPs, los diseños basados en plataforma... Si a estas necesidades le unimos la alta capacidad de integración en una oblea de silicio, ha permitido que se pueda integrar todo un sistema en el interior de un chip, provocando la aparición de los SOC.
Los SOC son sistemas empotrados donde la integración de todos sus componentes se encuentran dentro de un CI. Los SOC se diseñan utilizando los modelos pre-diseñados de funciones complejas conocidas (IPs), que ofrecen una gran variedad de aplicaciones.

### Estructura

Un SOC tiene un gran número de componentes, con funcionalidades diferentes. Tal como se explicó anteriormente, cada uno de esos componentes es una IP. A continuación haré una breve lista con algunos de los componentes (no todos) más comunes en un SOC:

* Un (o mas) procesador(es)
* Unidad de Punto Flotante (FPU): Dedicada a realizar cálculos aritméticos en coma flotante .
* Co-Procesador.
* Un bus: Elemento de comunicación entre los diferentes componentes de un sistema.
* direct memory access (DMA):Permite a ciertos periféricos acceder a la memoria principal (lectura o escritura) independientemente del procesador.
* Un Controlador de gestión de Memoria (MMU):Se encarga de los accesos a memoria por parte del procesador.
* Entrada/Salida de propósito general (GPIO).
* Bloques de memoria (PROM, Static Random Access Memory (SRAM), ...)
* Comunicaciones:
  - Serial (SCI): RS-232, RS-422, etc.
  - USB
  - Ethernet.
  - CAN.
* Tarjetas de memoria multimedia (SD, Compact Flash, etc.)
* Sistemas para el Debug (JTAG[3] , entre otros).
* Bloques analógicos
* Accelardores Hardware
* Micro-Electro-Mechanical Systems (MEMS)/Nano-Electro-Mechanical Systems (NEMS)

No es necesario que en un SOC estén todos los componentes enumerados anteriormente, únicamente se tendrán aquellos componentes que sean requeridos para el sistema que se esté desarrollando y en función de los recursos hardware disponibles.

### Estilos de materialización

Existen diferentes alternativas de materialización para los Soft-IP, desde los estilos clásicos (Full-Custom, Semi-Custom, Gate-Arrays), hasta las FPGAs, pasando por soluciones intermedias como los Cell-Based o Sea of Gates. Aunque existen más metodologías, nos centraremos en los estilos clásicos y en las FPGAs, éstas últimas son las que se usarán en la realización del proyecto pero haré un breve apunte sobre los estilos clásicos:

#### Full-Custom

Diseño de un circuito a nivel de transistor, para ello determinaremos las dimensiones de cada transistor uno a uno, posteriormente dibujaremos el layout. Éste método es muy costoso y con un elevado riesgo de equivocarse, por eso para realizar este tipo de diseño utilizaremos herramientas Computer Aided Design (CAD).
Estas herramientas suelen consistir en un editor eléctrico, uno de layout y de esquemático, un simulador y un conjunto de herramientas de verificación, que son las de Design Rule Checker (DRC) (cumple con las reglas de fabricación), Layout Versus Schematic (LVS) (comparación entre layout

[3] Nombre común utilizado para la norma IEEE 1149.1 titulada Standard Test Access Port and Boundary-Scan Architecture

y esquemático) y la herramienta de extracción de dispositivos y parásitos. Con estas herramientas podremos primeramente dibujar un layout con cierta flexibilidad, posteriormente verificar la funcionalidad del diseño mediante la simulación y por último poder dividir el circuito en partes, situarlo en el silicio, y conectarlas entre ellas.

Ventajas:

* Diseñar el circuito con las prestaciones electrónicas y temporales deseadas.
* Elevada integración en el silicio, con lo cual los costes menguan ya que se ocupa menos área de silicio.
* Adecuado para circuitos de elevadas prestaciones.

Desventajas:

* Tiempo de diseño elevado y altos costes de diseño.
* Riesgo de rediseño muy elevado.

**Standard-Cells**

La opción de Standard-Cells se basa en la existencia de una librería de celdas básicas para ser usada por los diseñadores. Esta librería contiene información del layout y los parámetros necesarios para ser usados por las herramientas CAD. Suele ser específica de la tecnología y por lo tanto suele ser suministrada por el fabricante del chip.
Las librerías aceleran considerablemente el diseño ya que no te tienes que preocupar de diseñar uno a uno todos los transistores, sino que usas las celdas existentes en la librería. El layout de las celdas de la librería suelen tener unas dimensiones comunes que faciliten el uso de herramientas de Placement & Routing.

Ventajas:

* Diseño mas sencillo que en el Full-Custom, facilitando y haciendo mas óptimas las herramienta de Placement & Routing.
* Disminuye el riesgo de rediseño (medio/bajo), en el diseño ya que los layouts de las celdas de la librería han sido comprobados y están siendo utilizado por otros diseñadores.
* Reducción de gastos en los costes de diseño.

Desventajas:

* En este tipo de diseño solo se pueden usar las celdas de las librerías, con lo que la flexibilidad de diseño disminuye y no se puede controlar las prestaciones del diseño como se hacia en el Full-Custom.
* Densidades de integración menor.

**Gate Arrays**

Los Gate-Arrays se basan en la idea de que sólo se han de personalizar la(s) máscara(s) de metal, permitiendo que el resto generen un conjunto de celdas básicas, que son comunes a todos los circuitos. Esta metodología comporta una reducción de costes en la fabricación a base de compartir máscaras. Estas máscaras comunes configuran el pre-fundido. El ASIC, es decir, el circuito personalizado, se construye sobre una oblea de silicio en la que previamente se ha pre-fundido las celdas básicas. La existencia de pre-fundido resulta transparente al diseñador que se limita a "verüna librería de celdas de puertas lógicas, como en el caso de los Standard-Cells.

Ventajas:

* Costes de fabricación reducidos (máscaras de personalización).
* El tiempo de fabricación se reduce.
* Costes de diseño similares a los Standard-Cells.
* Riesgo de rediseño similar al de los Standard-Cells.

Desventajas:

* Se obtienen prestaciones más bien bajas porque todos los transistores tienen el mismo tamaño y las conexiones suelen ser largas.
* Baja densidad de integración, por culpa de que las posiciones de los transistores estan fijadas a priori y dejan poco margen de trabajo a las herramientas de Placement & Routing.
* Dificultad en el uso de módulos programables dentro del ASIC, aunque a veces los Gate-Arrays tienen módulos predefinidos (Random Access Memory (RAM), módulos analógicos) que generalmente se suelen infrautilizar.

#### Field Programmable Gate Arrays

Las FPGAs son CIs que contienen un número elevado de puertas y dispositivos lógicos que llegan al usuario una vez han pasado todo el proceso tecnológico de fabricación y encapsulado. La característica más notable de una FPGA reside en que es programable. El proceso de programación de las FPGAs es transparente para el usuario, ya que no es necesario conocer el funcionamiento interno de una FPGA para poder programarla, aunque el saberlo podría implicar una programación más eficiente. Dentro del chip, no toda la lógica está dedicada al circuito funcional, sino que existe un área dedicada a que el chip pueda ser programado.
Las FPGAs se usan en aplicaciones semejantes a los ASIC, pero tienen un menor rendimiento, un mayor consumo y las aplicaciones suelen ser menos complejas. Poseen la ventaja de que el tiempo de desarrollo es menor, de que el coste en volúmenes de producciones pequeños, es mucho menor y el que al ser programable casi elimina los riesgos de rediseño, ya que si te equivocas, programas de nuevo la FPGA. Por eso se usan frecuentemente para el desarrollo de prototipos del sistema para posteriormente pasar al desarrollo del producto en un ASIC.
En algunos casos, donde existe prisa por lanzar un producto al mercado, se usan para desarrollar una primera remesa del producto, ya que el diseño es más rápido y sencillo que con un ASIC (Semi-Custom y Gate-Arrays), mientras en paralelo se desarrolla el mismo producto en un ASIC.
Los fabricantes mas conocidos son Altera y Xilinx aunque no són los únicos, ya que también existen otros fabricantes como Lattice, ACTEL, Quick logic.

#### Otras metodologias

La metodología **Cell-Based** es una metodología intermedia entre Full-Custom y Standard-Cell, que mantiene los beneficios de poder utilizar celdas suministradas por el fabricante a la vez que te permite definir tus propias celdas. Los **Sea-of-Gates** son Gate Arrays en las que se optmizan el aprovechamiento del silicio eliminando los canales de interconexión y llenándola de celdas. Las conexiones se realizan a través de celdas que son inutilizadas.

## 2.6 Hardware Description Language

Los HDLs son lenguajes utilizados en diversos estilos de materialización de los anteriormente mencionados, entre los cuales están las FPGAs. Estos lenguajes se usan para la descripción formal de circuitos electrónicos (generalmente lógica digital), pueden describir su funcionamiento, su diseño, su organización y diseñar pruebas de funcionalidad (test) mediante la simulación. Una de las grandes diferencias con los lenguajes de programación software, es que los HDLs introducen la noción de tiempo. Describen la conectividad entre los diferentes componentes del sistema mediante bloques jerárquicos (entities), lo que ofrecen modularidad al sistema ya que cada entity se puede llevar de un sistema a otro solamente adaptando las diferentes señales. Los dos mas conocidos son el Verilog y el VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL), éste segundo es el que se usó para diseñar el LEON3 y bus AMBA, aunque existen más, como el Altera HDL (AHDL) o Java HDL (JHLD).

# 3. Entorno de Desarrollo

Como se ha detallado en los capítulos anteriores el objetivo final del presente proyecto final de carrera es el diseño e implementación de un microcontrolador basado en el microprocesador LEON3. LEON3 es una IP de procesador para sistemas encastados basado en la arquitectura SPARC V8. En torno al citado microcontrolador se han desarrollado todo el conjunto de herramientas necesarias para desarrollar sistemas de manera rápida y eficiente basándose en el concepto de diseño basado en plataforma.
A lo largo del presente capítulo se detallaran las diferentes herramientas hardware y software que forman parte del entorno de desarrollo ofrecido por Gaisler para sistemas encastados basados en LEON.

## 3.1 Grlib IP library

La librería de IPs Grlib es un conjunto de IPs integradas que permiten reducir en tiempo y coste el desarrollo de un SOC. La forma de conseguirlo, es mediante la reutilización de IP cores ya existentes mediante su adaptación y configuración para una aplicación específica.
Este conjunto de IPs normalizadas son de proveedores independientes, entre las cuales se encuentra el procesador LEON3. El entorno de librerias está orientado en torno al bus AMBA y utiliza un método de plug & play para reconocer y configurar las IPs de la librería sin la necesidad de modificar ningún recurso.
Contiene diversos templates con un sistema diseñado para una placa determinada y preparado para ser configurado. Estos templates variaran de uno a otro, ya que las IPs disponibles dependerán de los recursos hardware (FPGA, tipo de memoria RAM, PROM, USB...) existentes en la placa de desarrollo. Esto hace que la decisión de utilizar una placa de desarrollo, sea la primera decisión a tomar, puesto que de ella dependerá los recursos disponibles a la hora de trabajar con nuestro sistema.

### 3.1.1 Estructura

La librería de IPs Grlib tiene una estructura en forma de árbol, donde cada directorio tiene una funcionalidad específica. A continuación mostraré la estructura y una breve descripción de los directorios más importantes.

* **Boards:** Ficheros para la programación de la FPGA, por ejemplo, contiene los ficheros con los pin assigment de los templates.
* **Designs:** Contiene los diferentes templates existentes.
* **Docs:** Documentación de la librería
* **Lib:** En ella encontramos un conjunto de subcarpetas donde se hallan los ficheros que definen las diferentes IPs.
* **Software:** Conjunto de programas para el testbench.

Dentro de cada template hemos de tener en cuenta tres ficheros que son los que marcan el contenido del sistema, su configuración y el testbench que te dan por defecto.

* **config.vhd:** Fichero que contiene los parámetros de configuración del sistema.
* **leon3mp.vhd:** Contiene el top level entity y que instancia todas las IP cores. Usa el config.vhd para configurar el sistema.
* **testbench.vhd**: Emula una placa con sus componentes hardware.

### 3.1.2 Componentes

Dentro de la carpeta lib (anteriormente mencionada) encontraremos las diferentes IPs con los diversos componentes disponibles para el sistema, se agrupan por funcionalidad o por proveedor al que pertenece la IP. No todos los componentes estarán disponibles en la licencia de evaluación pero sí en la de comercial. Algunos de los componentes que podemos encontrar son los siguientes:

* Procesador LEON3
* Bus AMBA 2.0
* Advanced High-performance Bus (AHB)/Advanced Peripheral Bus (APB) Bridge
* MMU
* Generic Universal Asinchronous Receiver/Transmitter (UART)
* Multiprocessor Interrupt Controller (irqmp)
* Modular Timer Unit (MTU)
* GPIO
* AHB Debug Uart
* USB
* Ethernet

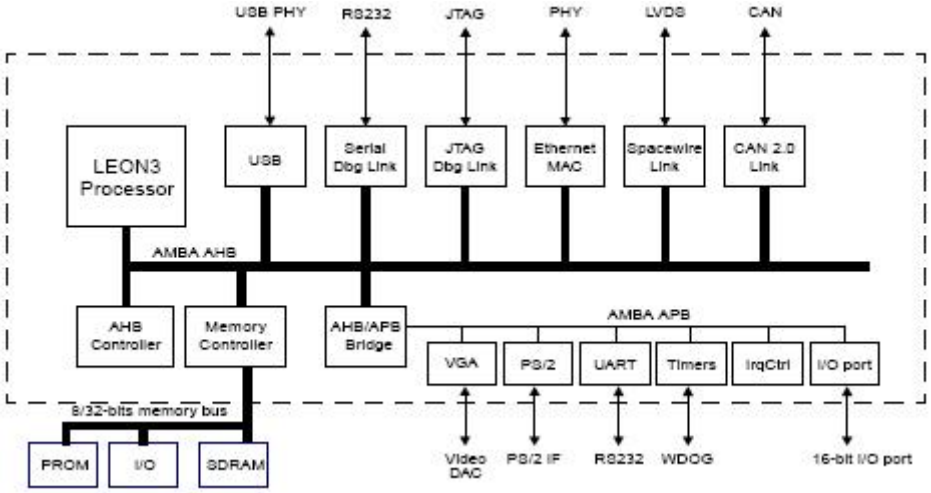



Figura 3.1: Ejemplo de Template del sistema

## 3.2 Leon3

### 3.2.1 Introducción

El LEON3 es un procesador Soft-IP, de 32 bits desarrollado inicialmente por la Agencia Espacial Europea (ESA) y posteriormente por la empresa Gaisler Research. Se basa en una arquitectura SPARC V8, que tiene como características principales instrucciones Reduced Instruction Set Computer (RISC) y big-endian. Todo su código está disponible bajo la licencia GPL, aunque también está bajo la comercial. Está diseñado en el lenguaje VHDL, es altamente configurable lo que lo hace idóneo para el diseño de un SOC.
Existen diversas versiones del LEON, la versión 3 es de la que estamos hablando en esta memoria, el LEON2 fue la versión anterior y existe además una versión tolerante a fallos llamada LEONFT.
El procesador esta distribuido bajo el paquete de librerías grlib, junto al resto de IPs que conforman el

sistema. Existe un amplio abanico de herramientas que nos permite desde configurar el sistema, debugarlo, como simular su ejecución, pero no para poder sintetizarlo, ya que si queremos sintetizarlo necesitamos herramientas de terceros como el ISE de Xilinx o el Quartus II de Altera.

### 3.2.2 Características

Sus características más destacables son:

* Pipeline avanzado de 7 etapas
* Multiplicaciones, divisiones Hardware y MAC[1]
* FPU, para mejorar el rendimiento.
* Bus AMBA 2.0 AHB y APB
* Arquitectura Hardvard[2].
* Memoria interna de alta velocidad utilizada para dejar momentáneamente los resultados de las operaciones de una tarea (Scratchpad memory).
* Técnicas de ahorro de energía (Power-down mode y clock gating).
* MMU, cumpliendo la arquitectura SPARC V8
* Multiprocesamiento Simétrico (SMP) y Multiprocesamiento Asimétrico (AMP).
* Implementación de un sistema de debug en un chip.

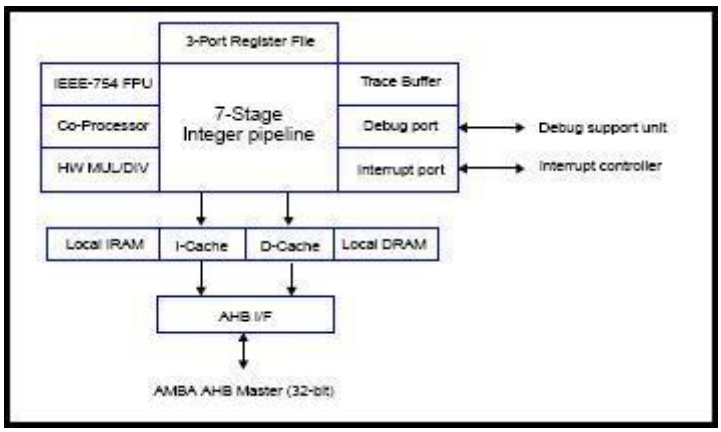



Figura 3.2: Diagrama de bloques Del Procesador LEON3

## 3.3 Bus AMBA 2.0

La primera especificación del AMBA fue introducida en 1995 por ARM. Es ampliamente utilizado como Bus-on-chip, inicialmente por los procesadores de ARM. Su segunda especificación es ampliamente utilizada en procesadores embebidos y no es necesario pagar para su utilización, sigue una estrategia de gestión e interconexión de los bloques funcionales que componen un SOC.
La especificación del bus AMBA tiene como objetivo satisfacer cuatro requerimientos:

[1] Multiply-accumulate que consiste en a = a + b
[2] cache de datos y de instrucciones por separado y configurables.

- Facilitar el desarrollo de microcontroladores embebidos con uno o más microprocesadores.
- Mejorar reusabilidad de las IPs.
- Mejorar la independencia del procesador mediante un diseño modular.
- Minimizar la infraestructura de silicio para soportar una comunicación on-chip y off-chip.

La especificación 2.0 define 3 tipos de buses:

- **AHB:** Este bus de alto rendimiento es la novedad de la especificación 2.0 y que tiene como características, pipeline, transferencias en ráfaga, múltiples masters en el bus (procesador, DMA...), y splits transfers (división de transferencias).
- **Advanced System Bus (ASB):** Fue el bus de alto rendimiento introducido en la primera especificación del AMBA. Acepta múltiples masters y está diseñado para tener un alto rendimiento en microntroladores de 16 o 32 bits. Como características destacables remarcar que no tiene splits transfers, es decir, solo acepta transferencias individuales, en ráfaga, o address-only (donde no hay una transacción con datos).
- **APB:** Es el bus que se usa para periféricos lentos, con un costo energético bajo (low power) y adecuado para un gran número de ellos. Se comunica con el bus AHB mediante un bridge (bridge AHB-APB), lo que hace que el que se comunica con los masters del AHB es el bridge y no el periférico. Esto otorga cierta independencia al bus apb ya que puede estar realizando transacciones mientras en el AHB se realizan otras tareas.

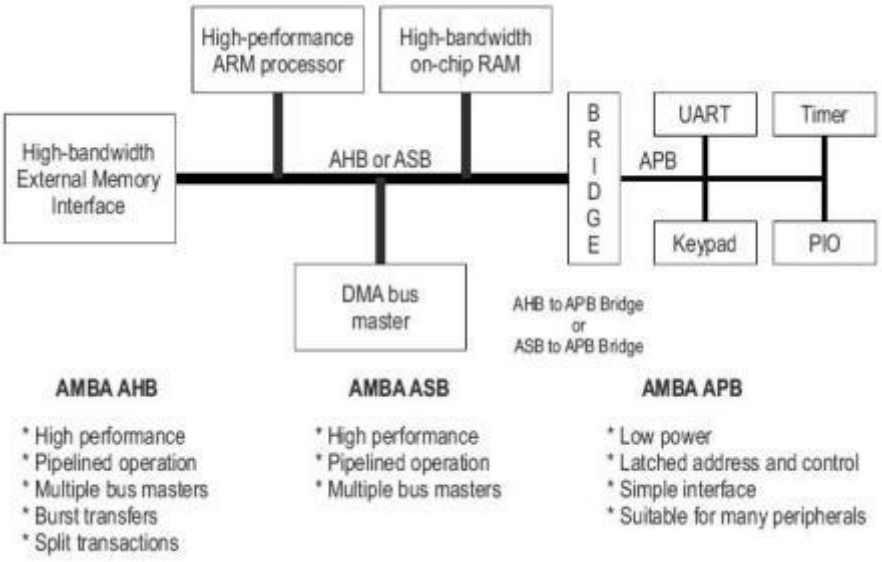



Figura 3.3: Típico Sistema Amba

Normalmente se tiene la concepción de que un bus es una línea que conecta todos los componentes, realmente no es así (Figura 3.4), ya que su estructura interna se compone de diferentes elementos. En el caso del bus AHB existen diversos elementos entre los cuales están el Decoder y el Arbiter.
El Arbiter lo que hace es asegurarse de que solamente un master está usando el bus al mismo tiempo, esto lo realiza mediante un multiplexor, que únicamente permite pasar las señales del master que tiene el control del bus, mientras que el Decoder, genera las señales de selección de cada esclavo en función de la dirección de transferencia.

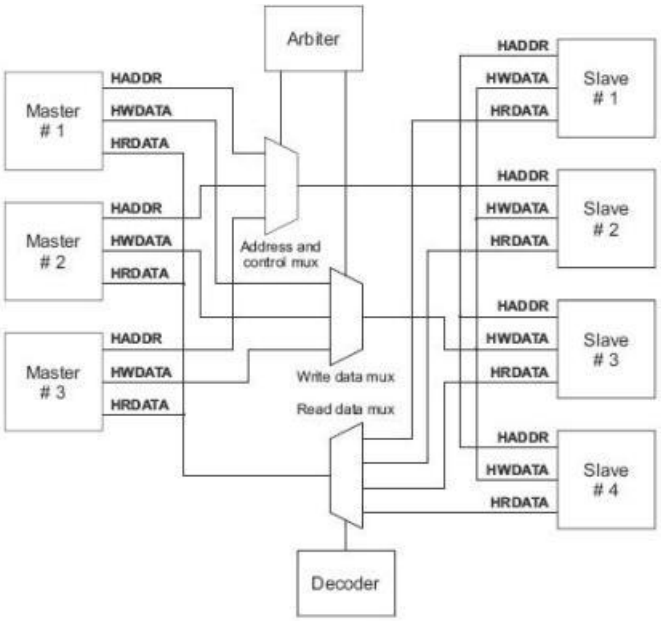



Figura 3.4: Arbitro/Decodificador del Bus Amba

## 3.4 Plataforma: NIOS II Development Board

El kit de Nios II Development board que se usa en este proyecto es la Nios II Development Board Stratix II Edition y consta de:

* La placa de desarrollo (Figura 3.5) que cumple con la directiva Restriction of Hazardous Substances (ROHS)[3].
* Software con un año de licencia (Quartus II en Windows, NIOS II embedded suite)
* Cables (USB, Ethernet cable, Adaptador a la corriente, pantalla LCD, entre otros).

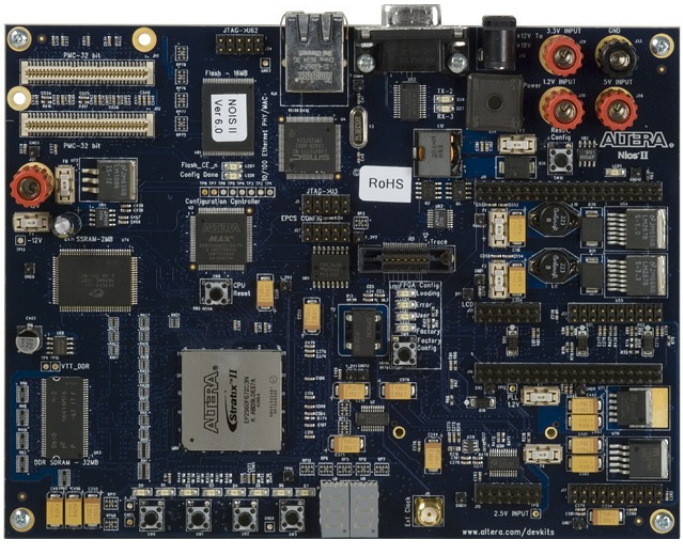



Figura 3.5: Nios II Development Board

[3] directiva que restringe el uso de 6 sustancias (Plomo, Mercurio, Cadmio, Cromo VI, PBB y PBDE). PBB y PBDE son sustancias usadas en plásticos.

Existen diversos motivos por los cuales, se han escogido, esta placa de desarrollo. El primero de todos es que era una placa que ya se tenía y por lo tanto el coste de adquirirlo ha sido cero. Esto de por sí solo no puede ser una razón suficiente para escoger entre las múltiples posibilidades existentes en el mercado, pero sí un factor a tener en cuenta.
La placa de desarrollo posee todo los componentes necesarios para el sistema, además la FPGA cumple con la necesidad de recursos (bloques lógicos, Phase-Locked Loopss (PLLs)...) del sistema. Tiene las memorias necesarias (RAM,Read Only Memory (ROM)), contiene herramientas de debug (JTAG) y aunque no hay un componente que sea un conector USB, sí que tiene el Conector de Expasión de Prototipos (Proto) que nos permitirá tener el conector USB del que carece.
Otra razón importante es la existencia de un template para esta placa de desarrollo facilitando el modelado y diseño del sistema.
Algunos componentes de la placa de desarrollo:

* FPGA Stratix II (EP2S60F672C3)
* 16 Mbytes de memoria flash
* Synchronous Static Random Access Memory (SSRAM) de 2 Mbytes
* Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) de 32 Mbytes
* Conector JTAG de Altera
* Oscilador a 50Mhz
* Conector Serial
* 4 Botones switches
* 8 Leds de configuración del estado de la FPGA.
* Entrada externa para un clock.
* 2 Conectores de expansión de prototipos (Proto1 & Proto2)
* Conector Compactflash
* Botón de reset de la Central Processing Unit (CPU)
* Botón de reset de la configuración de la FPGA
* Doble display siete segmentos

Cabe mencionar que esta placa de desarrollo no contiene USB, lo que provoca que necesitemos añadir en el proto2 una placa (Figura 3.6) con el chip de Cypress (CY7C6800) y el conector USB.

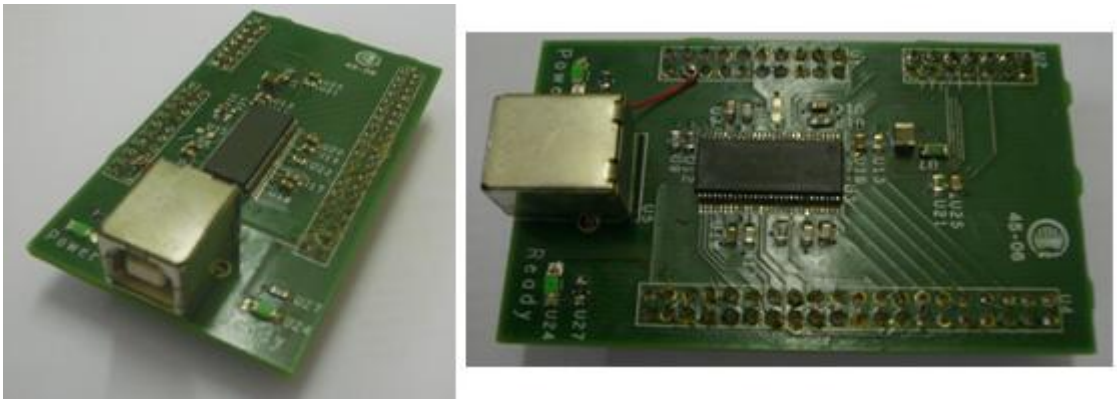


Figura 3.6: Placa de USB con el chip de Cypress

### Características de la FPGA EP2S60F672C3N

A continuación haremos una breve descripción de la FPGA utilizada, como de sus características más importantes. La familia Stratix de Altera, son las FPGA de alta gama de la cual su segunda generación es la Stratix II y sus características más destacables son las siguientes:

| *Table 2–2. Stratix II EP2S60 Device Features* | |
|---|---|
| LEs | 60,440 |
| M4K Memory Blocks | 255 |
| Total RAM Bits | 2,544,192 |
| Embedded 18x18 Multiplier Blocks | 144 |
| Enhanced PLLs | 4 |
| Fast PLLs | 8 |
| User I/O Pins | 718 |

Figura 3.7: Características Stratix II

Las FPGAs Stratix2 de Altera implementa dos tipos de PLL diferentes y tiene hasta un total de 14. También contiene 60,440 Logic Elements, 255 bloques de memoria RAM dual-port de 4Kbits mas paridad, con un total 2.544.192 bits de memoria RAM, bloques multiplicadores y 718 pins de E/S.

## 3.5 Licencias

Existen dos tipos de licencia disponibles, la primera es una licencia GPL[4] y otra comercial[5]. En la propuesta del proyecto ya venia dado que usaríamos la licencia GPL ya que no tiene coste económico, pero por el contrario comportará un conjunto de restricciones.
Estas restricciones vienen dadas en diferentes vertientes, la primera consiste en que el sistema que está bajo la licencia GPL no tiene todas las IPs disponibles y la otra viene dada por el software específico que utilizaremos, ya que te viene con licencias temporales (pasado un tiempo caduca) o con limitaciones funcionales, es decir, hay herramientas/comandos que sólo están disponibles en la versión de pago del programa.
Todo esto hace que el desarrollo del sistema se complique, con lo que tienes dos caminos a seguir, el decidirte a pagar la licencia comercial o adaptar una IP.

## 3.6 Software específico utilizado

### 3.6.1 Generación de Scripts

Para poder usar el sistema y configurarlo, primeramente se han de generar un conjunto de scripts que servirán para poder usar las diferentes aplicaciones que soporta. Esta generación de scripts se hace mediante el comando make[6] y los Makefiles.
Las opciones más interesantes para el desarrollo de este proyecto son las siguientes:

[4] licencia orientada principalmente a proteger la libre distribución, modificación y uso de software.
[5] se ha de pagar por ella
[6] make es una herramienta de generación o automatización de código, muy usada en los sistemas operativos tipo Unix/Linux. Lee las instrucciones para generar el programa u otra acción del fichero makefile

- **make scripts:** genera todos los scripts
- **make vsim:** genera los scripts para el Modelsim
- **make vsim-launch:** ejecuta el Modelsim y abre el proyecto del sistema.
- **make quartus:** genera los scripts para el Quartus II de Altera
- **make quartus-launch:** ejecuta el Quartus II y abre el proyecto del sistema.
- **make distclean:** borra los ficheros temporales y scripts, dejando los ficheros sistema.
- **make xconfig:** ejecuta la aplicación Xconfig
- **make xgrlib:** ejecuta la aplicación Xgrlib.

### 3.6.2 Xconfig

Herramienta gráfica que viene en la librería de IPs Grlib y que se utiliza para generar el fichero de configuración del sistema (config.vhd).
Contiene diferentes menús que nos permite encontrar las diversas opciones de configuración que hay para nuestro sistema. Cada template tiene una configuración específica del Xconfig, adaptada a los recursos disponibles. Los menús que vienen por defecto son los siguientes:

- Synthesis
- Clock Generation
- Processor
- Amba Configuration
- Debug Link
- Peripherals
- VHDL Debugging

Si te generas tu propio template del sistema, con IPs añadidas, el Xconfig permite la posibilidad de añadir menús a los ya existentes, para poder seguir configurando tu sistema en función de los recursos existentes en tu template(Figura 3.8).

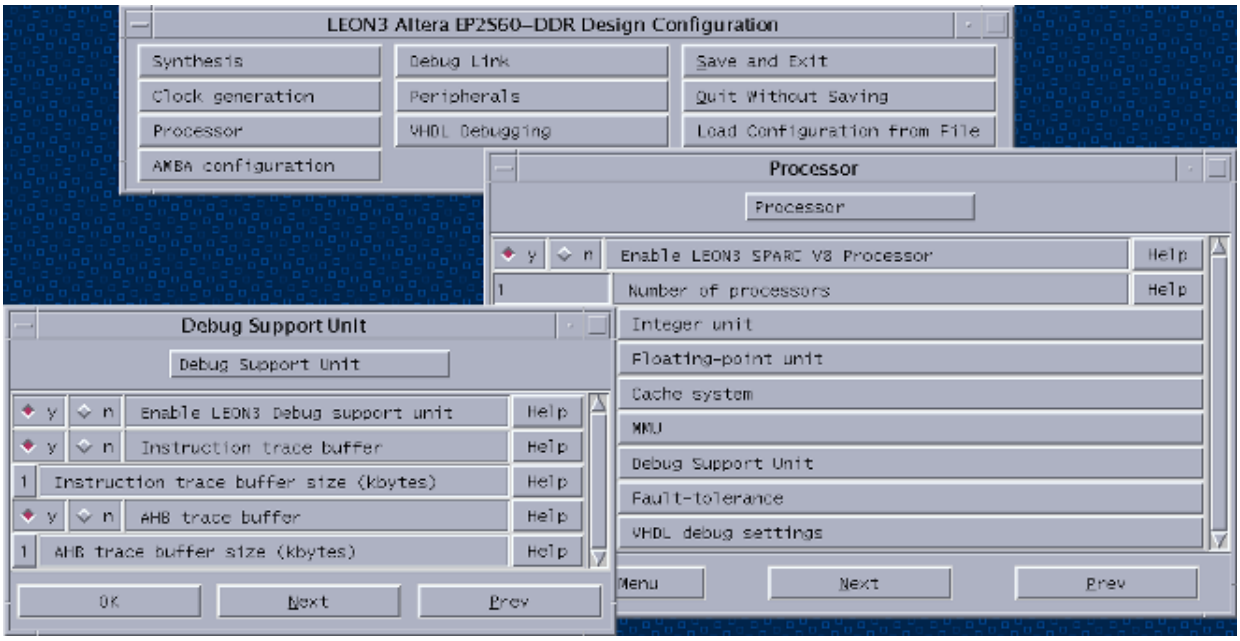



Figura 3.8: Aplicación Xconfig

### 3.6.3 BCC

Un Cross compiler (compilador cruzado) es un compilador capaz de crear código ejecutable para otra plataforma distinta a aquella en la que se ejecuta. BCC es un conjunto de aplicaciones que nos permitirá generar código para un sistema LEON3. En el caso que nos ocupa, usaremos la sparc-elf-gcc, ya que nuestro programa está escrito en C. Otra aplicación que utilizaremos será sparc-elf-objcopy para generar un fichero en formato SREC[7],que es un formato utilizado para programar las memorias flash.
BCC son un conjunto de aplicaciones que se ejecutan por línea de comandos, con lo cual puede ser usado bajo Linux o Windows mediante Cygwin.

### 3.6.4 Mkprom2

Este programa es una evolución del sparc-elf-mkprom (una de las múltiples aplicaciones existentes en el BCC), y que nos permite crear imágenes de arranque para diferentes sistemas entre ellos está el LEON3. Para poder generar esas imágenes anteriormente se han de haber compilado con el sparc-elf-gcc.
Éste genera la imagen de arranque con la que se programará la memoria PROM.
Dependiendo del sistema a generar y de los componentes a usar, existen diversas opciones aplicables al Mkprom2, para generar el archivo adecuado a la configuración del sistema en el que servirá de arranque. Aquí pongo el ejemplo de 3 opciones:

* **-freq :** Indica la frecuencia del clock del sistema (Mhz).
* **-uart :** Insertar la dirección de los registros de la uart.
* **-ddrram:** Especificar el tamaño del banco de memoria DDR.

Mkprom2 es una aplicación que se ejecuta por línea de comandos, con lo cual puede ser usado bajo Linux o Windows mediante Cygwin, esta aplicación se utiliza en combinación con el BBC para generar el software que se usará en el sistema.

### 3.6.5 Grmon RCP

Grmon es un monitor destinado a controlar el sistema debug de un SOC basado en el procesador LEON. Éste se comunica con el Debug System Unit (DSU) del LEON y permite la depuración no intrusiva del sistema de destino. Las características son las siguientes:

* Accesos de Lectura/Escritura a todos los registros y a la memoria.
* Utiliza un Disassembler que traduce de lenguaje máquina (código binario) a lenguaje ensamblador para poder ver la ejecución de instrucciones.
* Descargar y ejecutar las aplicaciones en el sistema.
* Manipulación de breakpoints and watchpoint.
* Conexión remota al gdb (GNU Compiler Collection (GNU) debugger).
* Programación de memoria Flash (Intel/AMD PROMs)
* Auto-detecta el hardware configurado del sistema
* Soporta el plug & play para las interfaces del LEON3/GRLIB.
* Interfaz de depuración mediante Serial, Ethernet, JTAG, Peripheral Component Interconnect (PCI) y USB.
    - En este caso usaremos el JTAG de Altera que se conecta al JTAG Debug Link, para comunicarnos en el sistema.

Se puede ejecutar en tres Sistemas Operativos Linux, Windows y Solaris. La versión gráfica del Grmon es el Grmon RCP que es la aplicación que he utilizado para programar la memoria PROM y ejecutarlo.

[7] Ver apendices

### 3.6.6 DMnewGUI

Esta aplicación está desarrollada en el IFAE y su función es la de interactuar con el ROIC Medipix2 MXR, estableciendo comunicación con el controlador USB del sistema LEON3-AMBA. Nos permite visualizar los datos leídos del ROIC en la pantalla, así como ejecutar las diversas opciones existentes (Figura 3.9).

Para establecer la comunicación este programa usa un pequeño protocolo que consiste en enviar dos

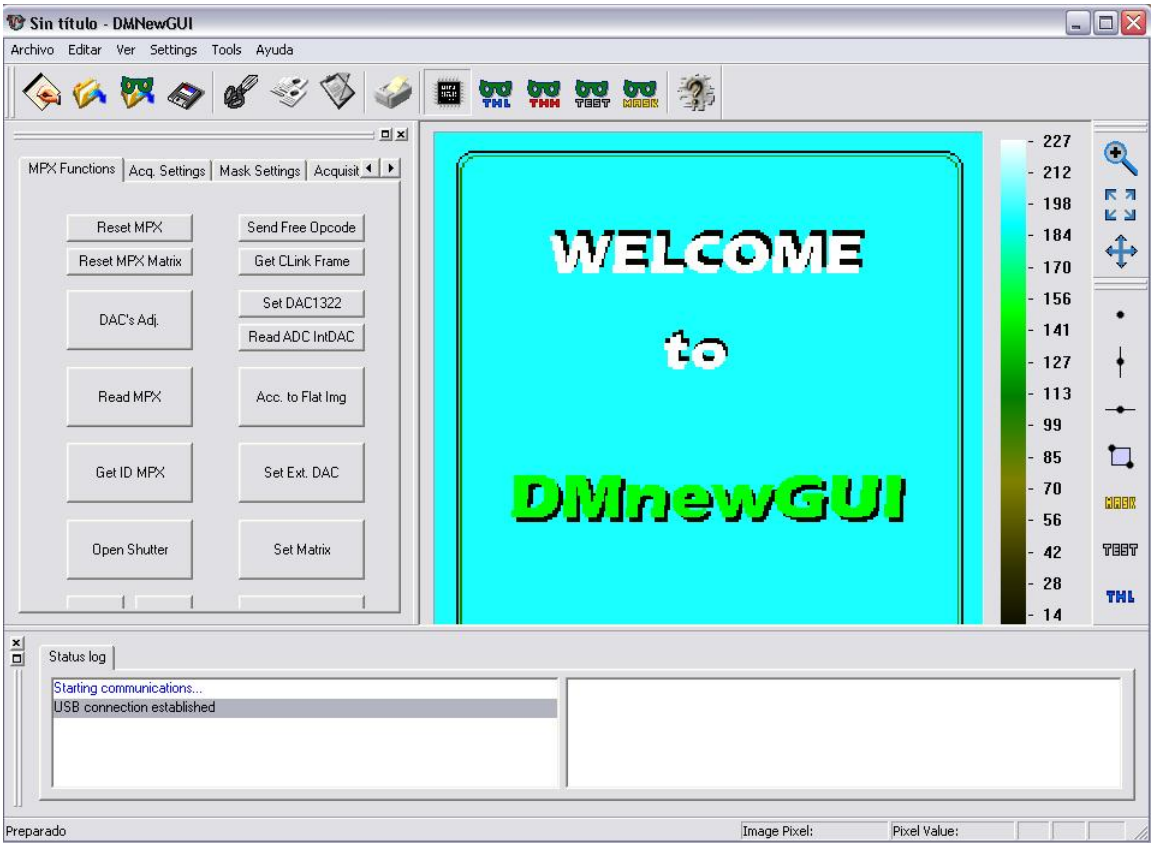



Figura 3.9: Aplicación DMNewGUI

paquetes de 16 bits indicándole al sistema la cantidad de datos que se transmitirán en la transferencia que se está iniciando. Una vez transmitidos estos dos paquetes, espera a que el sistema le responda con los mismos datos exactamente, si esto se confirma va enviando datos, hasta que finaliza el envío de datos. En cada envío de datos, espera un Ack (acknowledgement) de respuesta, hasta que se finaliza todas las transferencias.

## 3.7 Flujo de diseño

Para poder configurar, desarrollar y sintetizar el sistema en una FPGA se sigue un flujo de diseño con diversas etapas iterativas (Figura 3.10).
Las etapas son las siguientes:

1. Partimos de unos requerimientos iniciales para realizar el sistema
2. Se decide que partes serán software y que partes serán hardware.
3. Configuración del sistema: Decidir las características configurables del procesador, como los periféricos que tendrá nuestro sistema.
4. Generación del programa en C.
5. Compilación del sistema: Generar el fichero que irá contenido en la PROM
6. Simulación del sistema mediante Modelsim: Primer hito, que nos indicará si hemos de reconfigurar el sistema o recompilar el Software o podemos programar la FPGA con el sistema.
7. Programación de la FPGA mediante el programa Quartus II de Altera
8. Programación de la memoria ROM utilizando el programa GrmonRCP
9. Si el programa no funciona correctamente desde el paso 1, si funciona el programa el sistema ya está diseñado.

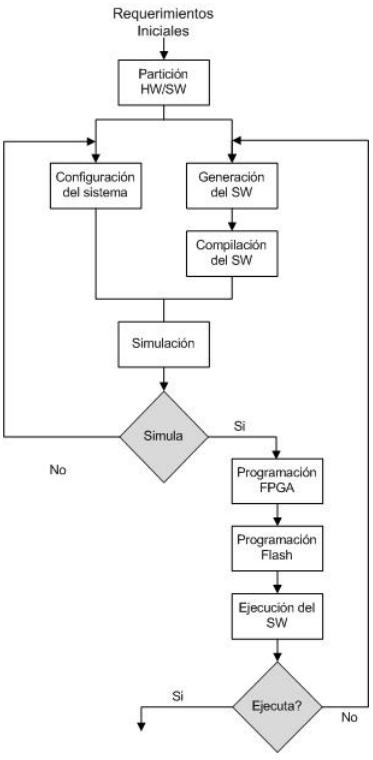



Figura 3.10: Diagrama de Flujo

# 4. Setup del LEON3

A continuación veremos todos los pasos necesarios para poder llegar a ejecutar un programa (hello.c) en un sistema LEON3-AMBA que habrá sido programado en una FPGA de Altera Stratix II.

## 4.1 Configuración Del Sistema

Primero antes de empezar a configurar el sistema hemos de mirar los diferentes templates existentes para ver si alguno se adapta a nuestras necesidades y nos encontramos con el template leon3-altera-ep2s60-ddr, que es justamente la placa de desarrollo seleccionada en su versión con memoria DDR SDRAM. Esto hace que se simplifique el trabajo en momentos puntuales ya que no tenemos que empezar a añadir componentes desde cero y los pins para el sistema ya están asignados, con lo cual solo nos tendremos que preocupar de añadir al sistema aquellas IPs que necesitemos.
Configurar el sistema mediante la aplicación Xconfig, genera un fichero (config.vhd) que contiene un conjunto de parámetros que nos definirá el sistema. Esto nos permite elegir que IP usaremos y que opciones de configuración tendrá cada IP, la FPGA sobra la que trabajaremos (en nuestro caso StratixII), el clock del sistema, la configuración del leon3...
Los parámetros utilizados para la configuración son los siguientes:

1. Tecnología de síntesis : Stratix II
2. Frecuencia del clock : 50 MHz
3. Número de CPU : 1
4. Controlador de memoria del LEON2 (smc_mctrl)
   - Usaremos la memoria SRAM de este controlador para la simulación pero no para la ejecución del sistema en la FPGA.
5. Caché de Instrucciones y Datos por separado.
   - Accelerar la ejecución del sistema.
6. UART
   - Nos sirve para poder ver los printf mediante el Hyperterminal.
7. Multiprocessor Interrupt Controller (irqmp)
   - Cada vez que se genera un cambio en los State_flags se produce una interrupción.
8. Bridge de AHB-APB
9. Controlador de Memoria DDR
   - Es la memoria RAM usada por el sistema, ya que no tenemos el controlador de la memoria SRAM disponible en la placa de desarrollo.

Este sistema es el que hemos usado para la simulación inicial.

## 4.2 Generación y Compilación de programas en C

Una vez generado el primer sistema, tenemos que generar el primer programa en C. Para ello usaremos el BCC y el mkprom2. Para realizar dicha tarea hemos generado el siguiente script que se basa en la ejecución de 3 instrucciones:

```
sparc-elf-gcc -c -g -O2 hello.c -o prog.o -msoft-float
mkprom2 -leon3 -baud 9600 -romsize 16384 -romws 12 -romwidth 8 prog.o
        -msoft-float -freq 50 -ramsize 2048 -ramws 1 -rmw -ddrram 32 -ddrfreq 100
        -ddrcol 512 -v -o prog.exe
sparc-elf-objcopy -O srec prom.exe prom.srec
```

A continuación, pasaremos a comentar las opciones de cada instrucción, para poder así comprender mejor el porqué de cada una.

**sparc-elf-gcc [options] file**

- -o filename: nombre del fichero de salida.
- -c: Esta opción hace que no se comprima el código y permita ejecutar desde la PROM el programa. Si no ponemos esta opción guardaría en la PROM el programa comprimido y lo tendría que descomprimir en la ram para poder ser ejecutado, tardando así un tiempo mayor para la ejecución del programa.
- -O2: optimización de código
- -msoft-float: Esta opción es necesaria cuando no se tiene una FPU, ya que lo que hace es emular las operaciones en punto flotante.

**mkprom2 [options] input_file**

- -o: filename nombre del fichero de salida.
- -leon3: Especifica que será un ejecutable para el procesador LEON3, ya que está aplicación también puede generar ficheros para LEON2 y ERC32.
- -baud: Está opción nos marca el baudrate[1] al que trabajará la UART. No hay un valor fijo que poner, la única condición es que los dos extremos tengan el mismo valor. Los valores estándar son 300, 1200, 2400, 9600 y 19200 baudios.
- -romsize: Le indicamos el tamaño de la rom que tenemos en la placa.
- -romws: Indica los waitstates de la ROM.
- -romwidth: La anchura en bits de la PROM.
- -freq X: El sistema funcionará a una frecuencia "X".
- -ramsize: Se le indica el tamaño de la RAM.
- -ramws: Se le indica los waitstate de la RAM.
- -rmw: Permite realizar operaciones de 16 y 8 bits.
- -ddrram: Se le indica el tamaño de la memoria DDR.
- -ddrfreq: Frecuencia de la memoria DDR.
- -ddrcol: Se le indica el tamaño de cada columna.
- -v: Modo Verbose. Informa de las estadísticas de comprensión y de los comandos de compilación.

La opción msoft-float como otras dos que no he usado (-mflat, -qsvt) se han de poner tanto en el sparc-elf-gcc como en el mkprom2.

**sparc-elf-objcopy [option(s)] input_file [output_file]**

- -O srec: opción que indica que el fichero de salida de este comando será del formato SREC.

## 4.3 Simulación Inicial

Para la simulación inicial tendremos que generar los correspondientes scripts (los necesarios para ejecutar el simulador modelsim).
Cuando ejecutemos el script make vsim-launch, éste nos lanzará el modelsim con el sistema configurado para ser simulado. El siguiente paso es realizar una modificación en el fichero testbench.vhd. Este fichero genera las señales que simulan los diferentes componentes hardware (externos a la FPGA) de la placa de desarrollo que estamos utilizando. Para poder ejecutar el programa que deseemos,

[1] Número de símbolos por segundo, transferido

tendremos que programar el módulo correspondiente a la memoria PROM con el fichero *.srec generado en el paso anterior.
Este programa (hello.c) que podemos ver su código en el apéndice Ejemplos de Código fuente, aprovecha el hecho de que el printf está configurado para que tenga su salida por la UART y así poder ver los resultados por la consola del modelsim. Una vez realizada satisfactoriamente la simulación pasaremos a la programación de la FPGA y de la memoria flash.

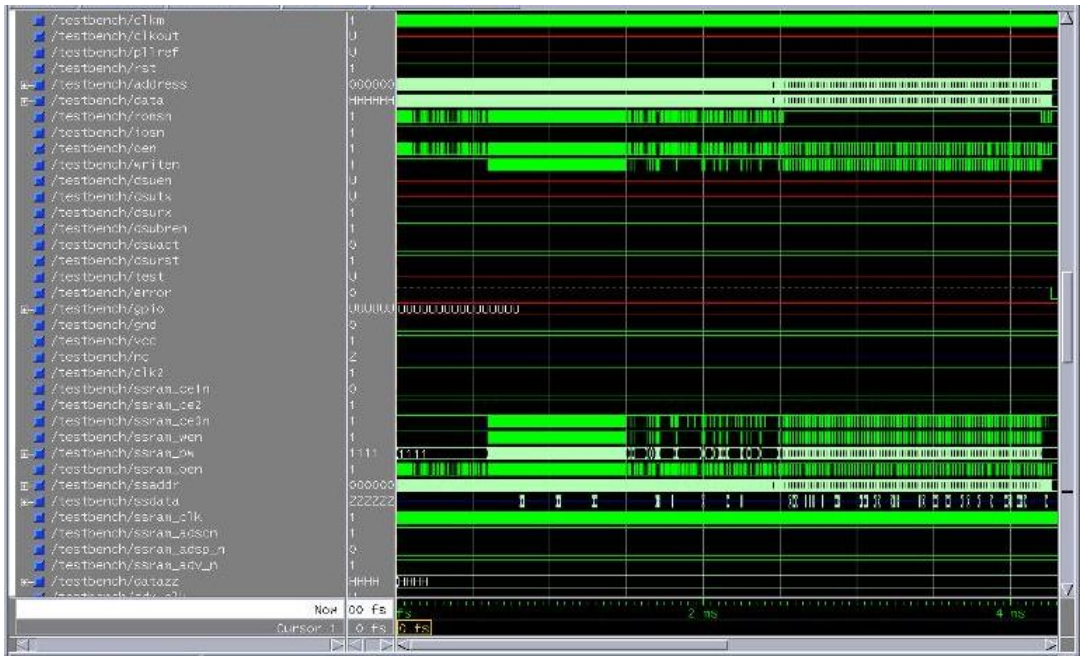



Figura 4.1: Simulación Inicial

## 4.4 Síntesis del sistema

El siguiente paso a realizar será la compilación del sistema con el Quartus II. Para ello usaremos el Compiler tool, este conjunto de herramientas permite pasar el sistema desde una descripción de alto nivel (vhdl) a la generación del bit stream para programar la FPGA, mientras que se asegura con un conjunto de verificaciones de que no tiene errores.
El Compiler tool consta de 4 pasos:

- **Analysis & Synthesis:** En este proceso se genera una netlist a nivel de puerta (gate level) y te informa de la cantidad de recursos que necesitará el sistema.
- **Fitter:** Comprueba que la FPGA tendrá suficientes recursos para el tamaño requerido por el sistema y en caso afirmativo, asigna a cada elemento de la netlist generada en el punto anterior uno de los recursos de la FPGA (placement) para posteriormente establecer un routing.
- **Assembler:** Genera el fichero que se utilizará para programar la FPGA.
- **Classic Timing Analyzer:** se ejecuta automáticamente como parte de una compilación completa para analizar e informar acerca de si se cumple o no las restricciones temporales del sistema.

Estos pasos te aseguran que se genera un sistema que se pueda programar en la FPGA, si no genera ningún error, significa que el código vhd es sintetizable, que los recursos necesarios del sistema son menores a los de la FPGA y que se cumple las restricciones temporales entre los diversos componentes que forman el sistema, por lo tanto generará un fichero *.sof con el sistema.

## 4.5 Programación de la FPGA

Para realizar la programación de la FPGA de Altera, utilizaremos el software subministrado por ellos mismos (Quartus II) y una vez se haya generado el fichero *.sof mediante el Compiler Tool, pasaremos a programar la FPGA con una herramienta existente en el Quartus II (Programmer), mediante el conector hardware (JTAG de altera).

## 4.6 Programación de la memoria PROM

Una vez se ha programado la FPGA, debemos programar la PROM con el programa que queremos que se ejecute en el procesador, para ello utilizaremos el ejecutable (bootloader para el sistema LEON3) generado anteriormente con la aplicación mkprom2.
Para poder programar la FPGA utilizaremos el programa GrmonRCP, mediante el JTAG de Altera. Una vez te has conectado al sistema, el siguiente paso es la programación de la memoria flash, primeramente desbloquearemos la memoria, para después borrarla y cargar el nuevo programa.
Los comandos son los 3 siguientes:

* Flash unlock
* Flash erase all
* Flash load path\prog.exe(bootloader generado por el mkprom2)

## 4.7 Ejecución del programa

Una vez realizado el paso anterior, el siguiente paso es ejecutar el programa, esto se puede hacer de dos formas diferentes y complementarias, una es desde el GrmonRCP y otra es desde la placa de desarrollo reseteando la CPU.
El programa GrmonRCP te permite controlar la ejecución del programa (permite ejecución paso a paso) o reiniciar el procesador, entre otras cosas. Si como es nuestro caso tenemos printf en el programa usaremos el HyperTerminal de Windows para ver la salida de la UART por pantalla y ver que hace correctamente aquello que deseemos.
El segundo método para ejecutar el programa, es mediante el reinicio de la CPU, con el botón de CPU reset que hay en la placa de desarrollo, ya que al resetear el sistema, vuelve a ejecutar el programa, una vez se ha inicializado correctamente.
A parte de poder ver los printf mediante el HyperTerminal existe la herramienta SignalTap dentro del QuartusII que nos permite ver las señales que se generan dentro de la FPGA. Se conecta mediante el JTAG de Altera, lo que implica que no podremos usar el SignalTap y el GrmonRCP a la vez. Este contratiempo hace que cobre importancia el reiniciar la CPU, ya que es la única forma de poder volver a ejecutar el programa mientras el SignalTap se está ejecutando. Y así, poder depurar el sistema, ya que podemos ver las señales deseadas en caso de que haya algo que no funciona correctamente.

# 5. Diseño del sistema

Una vez realizado el Setup del LEON3, tendremos que generar un sistema con la configuración que sea más adecuada para el desarrollo del proyecto.
Primeramente visualizaremos el sistema final tal como ha quedado, para posteriormente ir desarrollando todos los pasos realizados para llegar hasta él (en este capítulo no repetiremos pasos que son equivalentes a pasos dados en el capítulo anterior).

## 5.1 Estructura básica añadida

La estructura básica (Figura 5.1) de todo el sistema es la siguiente:

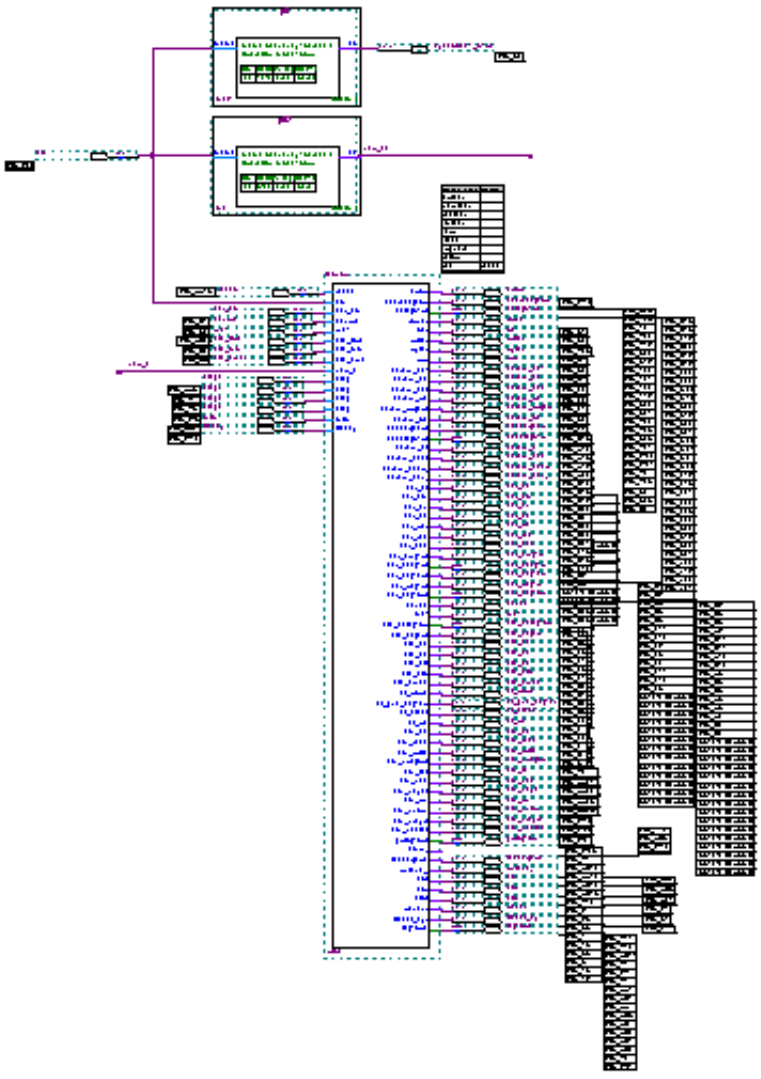


Figura 5.1: Esquema general del Sistema

En este sistema podemos observar tres componentes bien definidos por un lado hay dos altpll[1] (Figura 5.3) y por el otro lado está el sistema LEON3-AMBA. En la Figura 5.2 podemos observar la

[1] Altera pll

estructura clásica del sistema LEON3-AMBA con las IPs que hemos añadido.

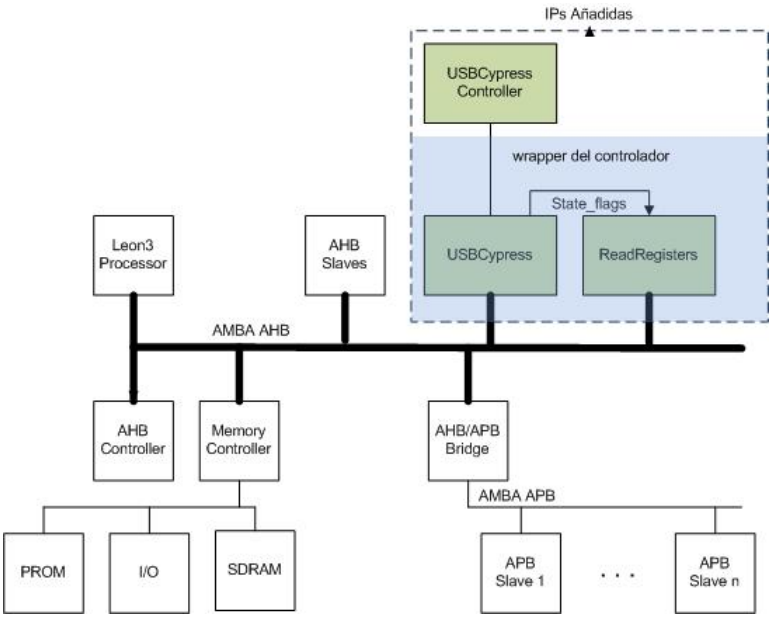



Figura 5.2: Esquema general del Sistema LEON3-AMBA

Els USBCypress Controller es el controlador de USB para el chip CY7C6800 de Cypress (este chip se encuentra juntamente con el conector USB, en la placa que se conecta en la proto2 de la NIOS Development Board). Este controlador, está diseñado para trabajar con señales del bus Avalon y por lo tanto necesitaremos de un wrapper, para poderlo adaptar a las señales del bus AMBA. Existen dos métodos diferentes a la hora de inicializar un USB, el primero es desde el software que se ejecuta en la FPGA (el programa en c, contiene los valores deseados para la configuración del USB) y la otra es mediante una inicialización hardware, es decir, el propio controlador es el que al iniciar la comunicación con el chip de Cypress, irá escribiendo los valores deseados, para que el USB se inicie correctamente. Como ya veremos más adelante este controlador funciona a 12Mhz y esto comportará unas complicaciones a la hora de sincronizar las señales.
El USBCypress y el ReadRegister son las IPs que componen el wrapper entre el controlador ya existente para bus Avalon y el sistema.
El USBCypress se comunica con el CY7C6800 de Cypress y por lo tanto se le tendrán que asignar pins a las señales que se comuniquen con el Chip.
La otra IP añadida es ReadRegisters que su funcionalidad radica en poder leer los State_flags desde el software cuando lo desee, y cuando estos generen un evento, activar una interrupción. Así que, no tendrá la necesidad de comunicarse con los diferentes recursos hardware de la placa de desarrollo y no tendrá pins asociados.
Los State_flags son una agrupación de señales que nos indican el estado del USB en todo momento. Estas señales son las siguientes:

- **Usb11Sentry:** Si esta señal está activa significa que el CY7C6800 identifica el host USB como 1.1
- **EnumOk:** Cuando la enumeración finaliza correctamente, el USB está configurado correctamente y esta preparado para hacer transferencias.
- **HSGrant**: Si esta señal está activa significa que el CY7C6800 identifica el USB como 2.0, si está a cero será USB 1.1.
- **Flags (a/b/c/d):** Nos indica el estado de la FIFO seleccionada por la FIFOADR[2:0].

- **ISS (Interrupt Status Byte):** Contiene los 7 bits que generan las diferentes interrupciones en el chip de Cypress.

Tal como mencionamos anteriormente, la placa de desarrollo no contiene USB y el hecho de añadir en el proto2 un módulo con USB, nos provoca que necesitemos un altpll en el sistema que vaya al chip de Cypress, ya que sin ese clock el módulo no funcionaria. Este clock ha de ser de 24Mhz.
El otro PLL de altera (pll2) es necesario, para el controlador de USB que tenemos en el sistema del LEON3-AMBA. El controlador ha de funcionar a 12Mhz y por lo tanto necesitamos un clock dedicado. Esto se hace porque el sistema ha de funcionar a una velocidad mayor (48Mhz) que el controlador de USB.

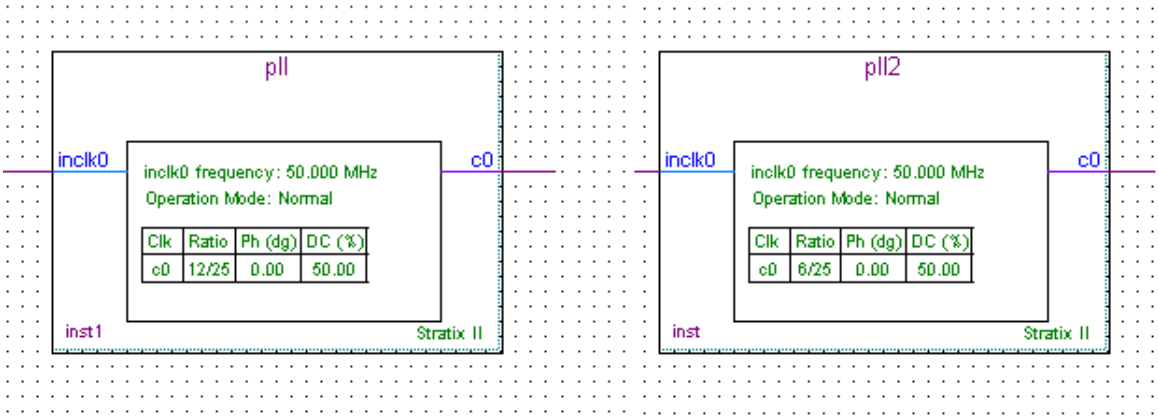



Figura 5.3: Plls de 24Mhz y de 12Mhz

Los dos plls funcionan con un clock de entrada de 50 Mhz y generan los clocks de salida deseados.

## 5.2 Inserción de un IP core en el sistema

En este apartado vamos a explicar como insertar un IP core como slave en el bus AHB. Para ello primeramente habremos de registrar la IP en el paquete de librerías Grlib. Este ejemplo se basa en la inserción de un USB.
Los pasos a dar son los siguientes:

- Crear la carpeta que contendrá la librería.
  Por ejemplo: *grlib-gpl-19-b3188\lib\cnm*
- Registrar la carpeta, añadiendo en el fichero *libs.txt* (grlib-gpl-19-b3188\lib) el nombre de la carpeta (en este ejemplo cnm).
- Crear la carpeta que contendrá los ficheros de la librería:
  *grlib-gpl-19-b3188\lib\cnm\ambausb.*
- Registrar la carpeta, creando un fichero dirs.txt en el path grlib-gpl-19-b3188\lib\cnm y añadiéndole en nombre de la carpeta ambausb.
- Ficheros que ha de contener la carpeta de la librería (ambausb):
  - Package de la librería : usb.vhd
  - Entities del package : USBCypressControlAmba.vhd, global_reset.vhd, USBCypressControl.vhd, cypressniosinterface.vhd
- Creación del fichero vhdlsim.txt (grlib-gpl-19-b3188\lib\cnm\ambausb) e inserción del nombre de los ficheros que se usaran para simular.
- Creación del fichero vhdlsyn.txt (grlib-gpl-19-b3188\lib\cnm\ambausb) e inserción del nombre de los ficheros que se usaran para generar el sistema que se utilizará para programar la FPGA.

* Añadir al Xconfig las opciones de menú del nuevo IP core. Para ello generaremos 4 archivos:
  - usb.in.vhd (definición de las variables para el config.vhd)

```
-- USB Cypress
constant CFG_USB0_EN : integer := CONFIG_USB0_ENABLE;
constant CFG_USB0_ADDR: integer := 16#CONFIG_USB0_ADDR#;
```

  - usb.in (definición de las opciones del menú)

```
mainmenu_option next_comment
comment 'USB Cypress '
bool 'USB Cypress Enable ' CONFIG_USB0_ENABLE
if["\$CONFIG_USB0_ENABLE" = "y"]; then
hex 'USB Cypress Address ' CONFIG_USB0_ADDR C00
fi
endmenu
```

  - usb.h (definición de las variables para el menú)

```
#ifndef CONFIG_USB0_ENABLE
#define CONFIG_USB0_ENABLE 0
#endif
#ifndef CONFIG_USB0_ADDR
#define CONFIG_USB0_ADDR C00
#endif
```

  - usb.in.help (help del menu)

```
USB Cypress
CONFIG_USB0_ENABLE
      Say Y here to enable USB Cypress
CONFIG_USB0_ADDR
      Set Addres for USB Cypress
```

  Editar el fichero config.in, añadiendo el siguiente código:

```
source lib/cnm/ambausb/usb.in
```

  Comprobación de los menús en el Xconfig (Figura 5.4, Página 30)

* Configuración del sistema, y generación del config.vhd al salir del Xconfig (Save and Exit). Al final del fichero encontraremos:

```
-- USB Cypress
constant CFG_USB0_EN: integer := 1;
constant CFG_USB0_ADDR: integer :=16#C00#
```

* Para que el Plug & Play que tiene el sistema nos detecte la IP deberemos de añadir la configuración de la siguiente forma:

```
constant REVISION: integer :=0;
constant hconfig : ahb_config_type := (
     0 => ahb_device_reg ( VENDOR_CNM, CNM_USB, 0, version, 0),
     4 => ahb_iobar(haddr, hmask),
others => zero32);
```

* Esta señal generada(hconfig), será una señal que irá al sistema de la siguiente forma:

```
ahbso.hconfig <= hconfig;
```

* Insertar la IP dentro del leon3mp.vhd (código en VHDL), si ha de comunicarse con algún recurso externo a la FPGA se la habrá de añadir los Pins.
  El código a añadir en el leon3mp.vhd será el siguiente:

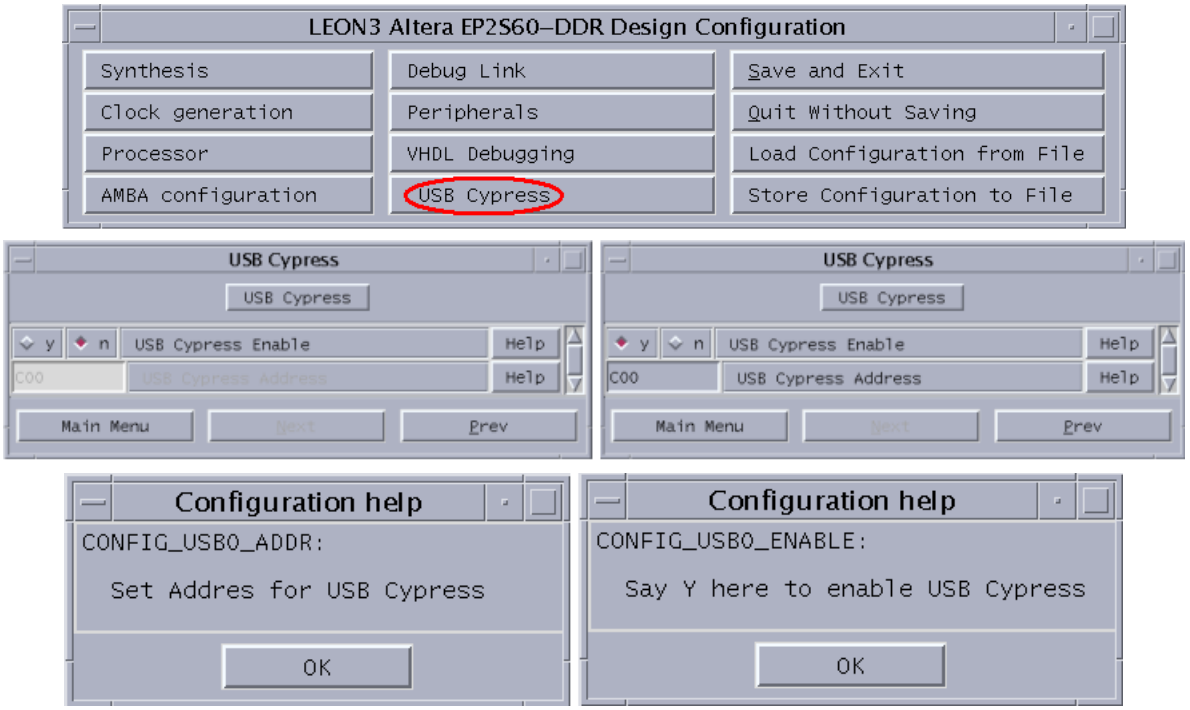



Figura 5.4: Xconfig

```
usbgen0 :if CFG_USB0_EN = 1 generate
   usb0 : USBCypressControlAmba generic map(hindex => 4,
          haddr => CFG_USB0_ADDR, hmask => 16#FFF#)
          port map (rstn, clkm, usb_clk, Ready, IntIn, Flaga, Flagb,
          Flagc, Flagd, pktend_n, Reset_cyp, ahbsi, ahbso(4), fd,
          fdaddr, slrd, slwr, sloe, wakeup, registerusb,
          usb_output.oen, cyp_running);
end generate;
```

* Inserción en el devices.vhd (grlib-gpl-19-b3188\lib\amba\devices.vhd) de los siguientes datos:
  - Vendor Code:

```
--vendor codes
constant VENDOR_CNM : amba_vendor_type := 16#03#;
```

  - Identificador de la Librería

```
-- cnm cores added
constant CNM_USB : amba_device_type := 16#0111#;
constant cnm_device_table : device_table_type := (
     CNM_USB => "USB                                ",
     others   => "Unknown Device                     ");

constant CNM_DESC : vendor_description :=  "USB CNM    ";

constant cnm_lib : vendor_library_type :=(
     vendorid           => VENDOR_CNM,
     vendordesc         => CNM_DESC,
     device_table       => cnm_device_table
);

   constant iptable : device_array := (
     VENDOR_GAISLER     => gaisler_lib,
     ...
     VENDOR_NASA        => nasa_lib,
     VENDOR_CNM         => cnm_lib,
```

```
        others              => unknown_lib);
```

* Comprobar que el sistema reconoce la IP
  - Pestaña sim del Modelsim una vez hemos compilado (Figura 5.5).

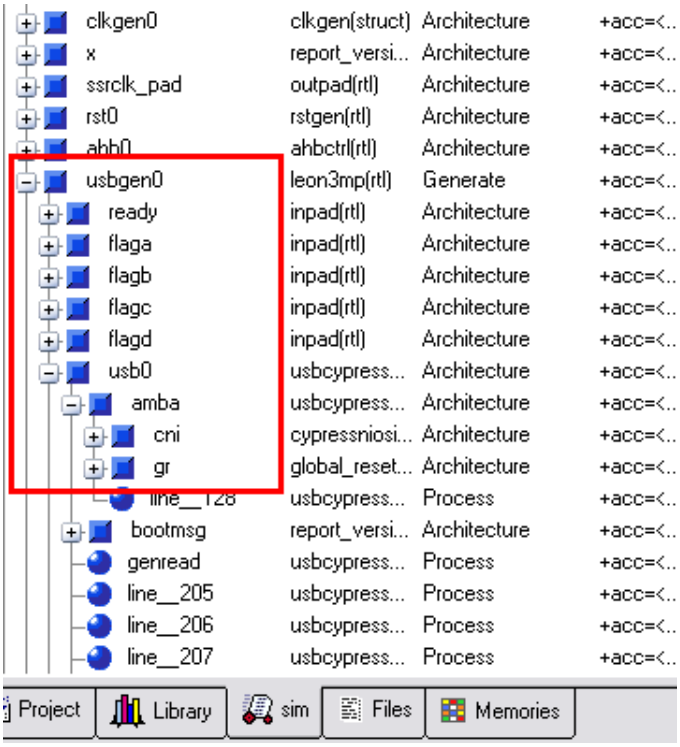



Figura 5.5: Simulación Modelsim

  - Simulación del sistema

```
# LEON3 Altera EP2C60 SSRAM/DDR CNM System Configuration
# Author of the system configuration: Sergio Morlans Iglesias
# GRLIB Version 1.0.19, build 3188
# Target technology: stratixii ,  memory library: stratixii
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area at 0xfff00000, 1 Mbyte
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        JTAG Debug Link
# ahbctrl: slv0: European Space Agency  Leon2 Memory Controller
# ahbctrl:      memory at 0x00000000,size 512 Mbyte,cacheable,prefetch
# ahbctrl:      memory at 0x20000000,size 512 Mbyte
# ahbctrl:      memory at 0x40000000,size 256 Mbyte,cacheable,prefetch
# ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
# ahbctrl:      memory at 0x80000000,size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        Leon3 Debug Support Unit
# ahbctrl:      memory at 0x90000000,size 256 Mbyte
# ahbctrl: slv4: USB CNM                 USB
# ahbctrl:       I/O port at 0xfffc0000,size 64kbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:      I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research Generic UART
# apbctrl:      I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research Multi-processor Interrupt Ctrl.
# apbctrl:      I/O ports at 0x80000200, size 256 byte
```

```
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:      I/O ports at 0x80000300, size 256 byte
# gptimer3:GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers,irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 8, irq 2
# Usb4: UsbCypress for amba rev 0
# ahbjtag AHB Debug JTAG rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 1 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
# clkgen_altera: altpll sdram/pci clock generator, version 1
# clkgen_altera: Frequency 50000 KHz, PLL scaler 2/2
# ** Note: Stratix PLL locked to incoming clock
#    Time: 110 ns  Iteration: 2  Instance: /testbench/d3/
clkgen0/alt/v/sdclk_pll/nosd/altpll0/stratix_altpll/m0
```

Estos son los pasos a seguir para añadir una IP como slave en el bus AHB. Este proyecto hemos añadido dos IPs al sistema, una de la cuales tiene que comunicarse con el chip de USB de Cypress y la otra solo se comunica con el sistema, tal como veremos a continuación.

## 5.3 Wrapper del bus AMBA

Este paso es el más complejo en el desarrollo del proyecto, ya que se trata de generar un wrapper (Figura 5.6).

Para generar este wrapper, primeramente hemos de tener en cuenta que el controlador de USB, es un controlador diseñado para las señales del bus Avalon (Altera), que trabaja con un clock de 12 Mhz y que por lo tanto irá a una velocidad mucho menor que el sistema, ya que éste trabajará a 48Mhz.

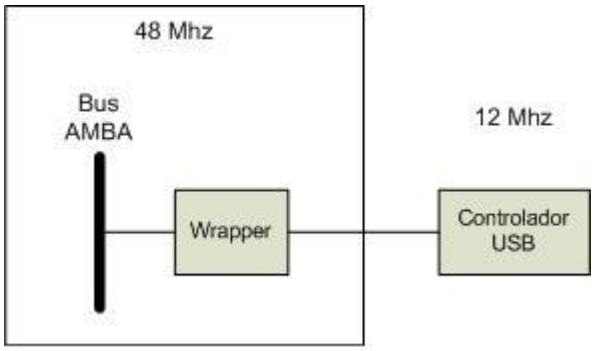



Figura 5.6: Esquema del Wrapper

Una vez hecho estos pequeños comentarios, lo primero de todo será registrar la IP en el paquete de librerías Grlib, posteriormente nos damos cuenta de que no solamente tendremos que adaptar las señales del bus Avalon al AMBA y viceversa, sino que también tendremos que realizar una adaptación de velocidad de las señales.

Este último punto será el que más conflictos genere a la hora de establecer una comunicación correcta entre el LEON3 y el controlador USB.

### 5.3.1 Adaptación del controlador al bus AMBA 2.0

Para adaptar la IP al sistema, hemos de tener en cuenta las señales que tiene el sistema originalmente (bus Avalon) y que señales tiene el bus AMBA, para poder averiguar cuales se pueden aprovechar (figura 5.7) y cuáles no.

Las señales no son asignaciones directas, ya que en el caso de Avalon, por ejemplo, trabaja con dos señales diferenciadas para la lectura/escritura (Avalon_Read/Avalon_Write), mientras que el AMBA trabaja con una sola (Hwrite). Como el AMBA solo tiene una señal para generar las dos de Avalon, una será la negación de la otra.

Para poder hacer un reset desde el software necesitamos acceder a la dirección de memoria XXXXXXX2h, donde las "X"dependen de la dirección dada a la IP en el sistema. Esto significa que se ha de coger el tercer bit menos significativo de Haddr para poder saber si lo que se quiere hacer es un reset, si vale 1 se hace el reset, sino no.

El caso de Chipselect, es una señal generada en el propio wrapper en función de dos señales del bus

| Avalon | Dirección | Amba |
|---|---|---|
| Avalon_Read | Entrada | Hwrite |
| Avalon_Write | Entrada | Hwrite |
| Avalon_Chipselect | Entrada | Chipselect |
| Avalon_Address | Entrada | Haddr |
| Avalon_WriteData | Entrada | Hwdata |
| Avalon_ReadData | Salida | Hrdata |

Figura 5.7: Señales Wrapper(AMBA-Avalon)

AMBA (Hsel, Hready[2] de entrada) y que se asignará a la señal de entrada Avalon_Chipselect. Esta señal solo se activará cuando las otras dos estén activas.
Hasta ahora hemos hablado de cómo adaptar las señales del bus AHB al bus Avalon, a continuación comentaremos las señales que se han generar para el bus AHB, desde el wrapper.
Podemos dividir en dos grupos las señales a generar para el bus AHB, en el primer grupo podríamos poner aquellas que sirven para que el sistema de plug & play del AMBA reconozca el periférico y en el otro aquellas que abarcan las transferencias que se realizan.
El wrapper generará el hconfig (ver Inserción de un IP core en el sistema), el hready en función de si se realizan las transferencias correctamente o no, el hindex indicando de que slave se trata, hresp indicando el tipo de transferencia realizada, hrdata que es de 32 bits y que por lo tanto se le duplican los datos de la siguiente manera:

```
ahbso.hrdata <= Avalon_ReadData & Avalon_ReadData
```

Una vez hemos generado este conjunto de señales el wrapper habrá sido adaptado correctamente al sistema y al controlador USB.

### 5.3.2 Sincronizador de las señales

Una vez ha sido adaptado correctamente el wrapper, a nivel de señales, al sistema hemos de realizar un sincronizador (adaptar las señales a nivel de clock, ya que se trabajan con dos velocidades diferentes).
El controlador USB realiza todas sus operaciones mediante una maquina de estados que opera en flanco de subida. Para poder sincronizar las señales que provienen del sistema con un clock cuatro veces más rápido, generaremos una señal clk_en.
Esta señal se activará cada cuatro ciclos, haciendo coincidir su flanco de bajada con el flanco de bajada del clock de 12Mhz. Con esto la máquina de estados solamente trabajará uno de cada cuatro ciclos de reloj del sistema, es decir, un ciclo de reloj del USB.
Una vez hemos realizado este paso nos encontramos con la máquina de estados sincronizada, pero esto genera un problema nuevo a solucionar y consiste en que todas las señales que lleguen con el ckl_en = ‘0’ serán ignoradas provocando que no se realicen las transferencias que se deberían realizar.
Para solucionar esto, lo que hemos hecho es generar unas señales nuevas que lo que hacen es coger las señales que iban a ser filtradas y alargarlas hasta que haya un clk_en = ‘1’ y poniendo el hready = ‘0’.
El hready es una señal del bus AMBA que si su valor es cero, se añade waitstates, hasta que la transferencia se realice y entonces tome el valor uno. Con esto se consigue que el procesador espere hasta que haya un clk_en = ‘1’, que es en el momento que se empezará a realizar realmente la transferencia.
En la Figura 5.8 podemos observar como el Avalon_chipselect que proviene del sistema no llega a la vez que el clk_en y por lo tanto la señal Chipselect_sync a realizar la función descrita en el párrafo anterior.

[2]Hay dos señales Hready una de entrada y otra de salida en cada IP. La primera te da la posibilidad que una IP empiece a actuar, mientras que la de salida sirve para indicar que la transferencia ha concluido.

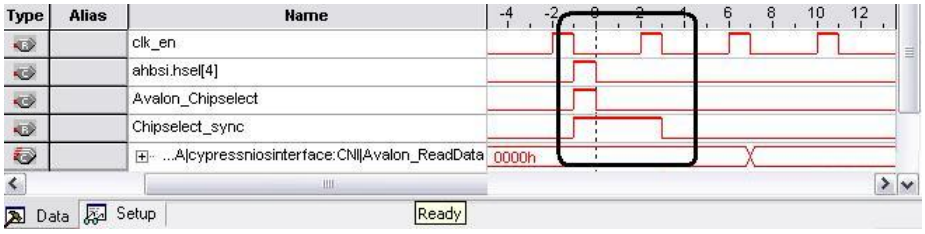



Figura 5.8: Ejemplo de Sincronización

### 5.3.3 IP ReadRegisters

Forma parte del wrapper del sistema y tiene dos funcionalidades claras. La primera permite que el sistema pueda leer los State_flags a través de esta IP. También cumple con el cometido de ser el generador de interrupciones para el USB. Cuando hay un evento en cualquiera de los State_flags activa la interrupción 11 del sistema.
A esta interrupción se le asociará una función en el software que irá actualizando los datos de los State_flags, para que el programa sepa en todo momento el estado del USB y así poder saber si el USB está esperando realizar alguna transacción o no está disponible.

## 5.4 Sistema de interrupciones del sistema LEON3/GRLIB

El controlador de interrupciones del LEON3 (irqmp) es un sistema multiprocesador que tiene dos niveles de prioridad y que dentro de cada nivel gestiona un total de 15 interrupciones. El nivel de mayor prioridad es el 1 y dentro de cada nivel la 15 es la que tiene una mayor prioridad mientras que la de menor es la 1. Para programar una interrupción con uno de los dos niveles tendremos que trabar con el registro Interrupt level register.
Para activar una interrupción tendremos que realizar tres tareas, cada una de estas tareas se realizara sobre el bit correspondiente a la interrupción. La primera consistirá en comprobar que no hay ninguna interrupción pendiente (Interrupt pending register), para posteriormente desenmascararla (interrupt mask register) y finalmente asignarle uno de los dos niveles de prioridad existentes.
Para desactivar una interrupción tendremos suficiente con enmascarar la interrupción (interrupt mask register).
Existe otro registro (Interrupt force register) que te permite forzar la ejecución de una interrupción desde el programa, si ésta ha sido activada, previamente.
A continuación pondré el ejemplo de un programa en código c, que viene con el Sparc-elf-3.4.4. El fichero se llama *c-irq.c*.

```
extern void *catch_interrupt(void func(), int irq);  //función que asocia
                             //una interrupción con una función del programa
int *lreg = (int *) 0x80000000;  //dirección donde se encuentra los registros

#ifdef LEON3  //definición de los desplazamientos
              //sobre la dirección base de los registros
#define ILEVEL 0x200
#define ICLEAR 0x20c
#define IMASK  0x240
#define IFORCE 0x208
#endif

enable_irq (int irq){
           lreg[ILEVEL/4] = 0;
lreg[ICLEAR/4] = (1 << irq); // quitamos la irq pendiente de la irq actual
lreg[IMASK/4] |= (1 << irq); // desenmascaramos la irq actual
}

disable_irq (int irq) { lreg[IMASK/4] &= ~(1 << irq); } // emmascaramos la irq
force_irq (int irq) { lreg[IFORCE/4] = (1 << irq); } // forzamos una irq
```

```
void irqhandler(int irq) {      //function asociada a la irq
       /** Code function **/
}

main(){
catch_interrupt(irqhandler, 10); //asociamos la irq 10 con la función irqhandler
catch_interrupt(irqhandler, 11); //asociamos la irq 11 con la función irqhandler
catch_interrupt(irqhandler, 12); //asociamos la irq 12 con la función irqhandler
enable_irq(10); //activamos la irq 10
enable_irq(11); //activamos la irq 11
enable_irq(12); //activamos la irq 12
  force_irq(10);  //forzar la irq 10 individualmente
force_irq(11);  //forzar la irq 11 individualmente
force_irq(12);  //forzar la irq 12 individualmente
force_irq(10);  //forzar la irq 10 individualmente
lreg[IFORCE/4] = (7 << 10); //force irq 10, 11 & 12
}                            //a la vez (prioridad máxima de la 12)
```

Este programa en c es el que utilizado como base, para la comprensión del controlador de interrupciones, para poder posteriormente generar el programa final que trabajará con el irqmp.

## 5.5 Pin Assigment

Una vez simulado el sistema antes de compilarlo con el Quartus II para programar la FPGA le hemos de añadir los pins necesarios para la nueva IP (controlador USB), esto se hará mediante una herramienta existente en el Quartus II (Assigment ->Pins).

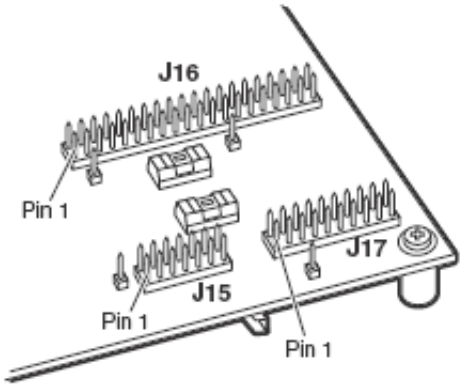



Figura 5.9: Esquema Proto2

La librería de IPs Grlib para el template altera-ep2s60-ddr te ofrece un fichero (*.qsf) con el pin assignment para la placa de prototipado correspondiente.
El trabajo a realizar se podría empezar desde cero o como en este caso añadir los pins, tomando como referencia un sistema ya existente para una FPGA Stratix (EP1S40F780C5). Esto es posible porque utilizan la misma placa de desarrollo (Nios Development Board) aunque sean modelos diferentes de FPGA y por lo tanto usaremos uno de los recursos hardware, el Expansion Prototype Connector (Proto2) Figura 5.9 para conectar la placa con el chip Cypress CY7C6800 y el conector USB. A continuación veremos la equivalencia entre los pins asociados de la FPGA y la Proto2 en el caso de las dos FPGAs y posteriormente veremos la tabla que nos servirá para pasar de una asignación de pins a otra.

Tal como podemos observar en Figura 5.11, Página 37, estos pins añadidos son los específicos de señales que van del sistema al chip y viceversa, pero todavía falta añadir un pin mas, que ira al Proto2 y que sin él, el sistema no funcionaria, ya que se trata de un clock que va al PLL que tiene el chip de Cypress, y sino ponemos ese pin, carece de importancia lo que se ponga ya que no funcionará. Necesitamos un clock de 24Mhz y lo generamos con un altpll (PLL de altera), que con una entrada de 50Mhz generada y tantos clocks de salida como deseemos. Este clock tendrá un pinout (Xtalin) que se conectará al pin K7 de la FPGA (proto2_pllclk). El pin ifclk únicamente se utiliza cuando el USB se configura de forma síncrona, en este caso, como trabaja de forma asíncrona no es necesario añadirlo.

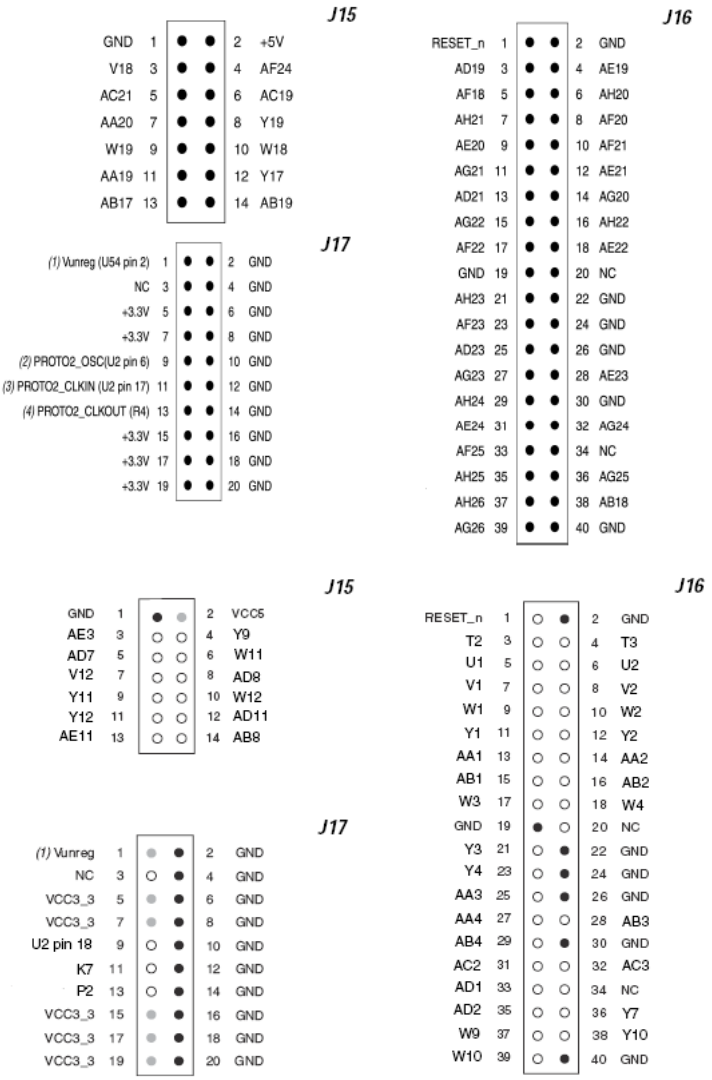



Figura 5.10: Esquema Proto2 para Stratix y Stratix II

## 5.6 Inicialización y Configuración USB

La inicialización y configuración del USB se realiza a nivel de hardware ya que es el controlador el que se comunica con el USB para configurarlo. La única señal que le afecta a la hora inicializar el sistema y configurar el USB es la señal de reset, el resto de señales no influyen hasta que el sistema ha sido configurado correctamente. Los otros dos requisitos para que el sistema se inicialice adecuadamente son que los pins (explicados en el paso anterior) estén correctamente asignados y que el clock del controlador sea de 12Mhz. Si estos requisitos se cumplen, al programar la FPGA y el sistema inicializarse, el USB también se inicializará correctamente.

## 5.7 Generación del Software para el IP core

El programa en c que se ha generar, primero de todo se ha de asegurar de que el USB se inicializa correctamente. Tal como ya se explicó durante el desarrollo de esta memoria existen tres señales que indicarán si el USB se ha inicializado como se desea o no. Como se necesita que el USB se inicialice en High-Speed, se necesita que la señal Usb11Sentry = '0' y la señal HSGrant = '1', si estas dos señales toman estos valores conjuntamente a EnumOk = '1', se tendrá el USB configurado como se desea. En estas señales se basa el programa a la hora de controlar la inicialización correcta o no

J15

| Pin Number | Signal | Pin Number | Signal |
|---|---|---|---|
| 1 | GND | 2 | +5V |
| 3 | | 4 | |
| 5 | pktend_n | 6 | reset_cyp |
| 7 | addr[0] | 8 | addr[1] |
| 9 | fd[15] | 10 | addr[2] |
| 11 | fd[13] | 12 | fd[14] |
| 13 | slwr | 14 | slrd |

J16

| Pin Number | Signal | Pin Number | Signal | Pin Number | Signal | Pin Number | Signal |
|---|---|---|---|---|---|---|---|
| 1 | RESET_N | 2 | GND | 21 | flagc | 22 | GND |
| 3 | | 4 | | 23 | flagb | 24 | GND |
| 5 | wakeup | 6 | | 25 | flaga | 26 | GND |
| 7 | fd[8] | 8 | | 27 | fd[7] | 28 | F[6] |
| 9 | fd[9] | 10 | | 29 | fd[5] | 30 | GND |
| 11 | fd[10] | 12 | | 31 | fd[4] | 32 | ifclk |
| 13 | fd[11] | 14 | | 33 | fd[0] | 34 | NC |
| 15 | fd[12] | 16 | IntIn | 35 | fd[1] | 36 | fd[2] |
| 17 | flagd | 18 | sloe | 37 | fd[3] | 38 | |
| 19 | GND | 20 | NC | 39 | ready | 40 | GND |

Figura 5.11: Correspondencia señales USB-Proto2

del USB, ya que lee los State_Flags y compara con los valores que debería tener. Posteriormente activamos las interrupciones, que estarán relacionadas con cualquier cambio en una de las señales de los State_flags.
Una vez se ha realizado estos dos pasos, el programa se pondrá a escuchar, esperando una interrupción que indique la inicialización de una transferencia de datos (Flaga = '1').
Para que llegue esa petición de datos primeramente hemos de conectar el dispositivo al programa DMNewgui (Archivo –>Reconnect device). Una vez se ha conectado correctamente el device, tal y como se ve el apartado de los resultados, el siguiente paso a dar es ejecutar la opción Send Free Opcode, que lo que envía es un opcode junto a la cantidad de datos que contendrá la transferencia. Estos datos desde el programa se recibirán mediante dos lecturas(cada lectura es de 16 bits), y se devolverán al DMNewgui con dos operaciones de escritura. Esta respuesta es el protocolo que establece el DMNewgui para saber que la transferencia está transcurriendo correctamente.
El siguiente paso es un bucle donde va leyendo un dato en cada iteración y le va respondiendo para que el DMNewgui sepa que todo transcurre correctamente, hasta que finaliza la transferencia y vuelve al estado inicial del programa.

# 6. Problemas Encontrados

El hecho de trabajar con la licencia GPL y no con la comercial, ha acarreado un conjunto de problemas. Para explicar esto vamos a recordar que si trabajas con una licencia GPL no tienes todas las IPs disponibles y que por lo tanto te puedes encontrar con problemas como el siguiente:

* No se ha podido utilizar la memoria SSRAM de la placa NIOS II development board ya que no estaba disponible el controlador adecuado para su utilización. Esto pasa porque en la licencia no comercial solamente tenemos disponible el controlador SRCTRL que trabaja con memoria SRAM y PROM, mientras que el controlador SSRCTRL solo esta disponible en la licencia comercial. La solución buscada para poder trabajar con una memoria RAM es aprovechar el controlador DDR que tiene para utilizar la memoria DDR SDRAM de la placa en lugar de la SSRAM, por eso en la compilación del programa le hemos tenido que especificar las características de la memoria DDR además de las otras opciones empleadas.
* No tenemos el controlador de USB, que aunque la placa de desarrollo no lo tenga, no hemos podido aprovecharlo, con lo cual, es otro problema relacionado con el tema de las licencias. Para poder solucionar este problema he adaptado el controlador de USB para el CY7C6800 de Cypress.

La inexperiencia al trabajar con memorias, hizo que me costara darme cuenta de que tenía que tener en cuenta los waitstates de la PROM y SRAM en el momento de configurar el programa a compilar, porque si no se tenía en cuenta, el sistema no simulaba correctamente. Es un problema que una vez te das cuenta, se soluciona rápidamente. Otro problema encontrado viene a raíz precisamente de usar la memoria DDR, no se puede simular el sistema con la memoria DDR (fichero README.TXT existente en el template utilizado).

```
Readme.txt
NOTE: the test bench cannot be simulated with DDR enabled
because the Altera pads do not have the correct delay models.
```

La solución aportada ha consistido en las simulaciones con la memoria SRAM del sistema y después en la síntesis del sistema, cambiar la configuración para poder trabajar con la memoria DDR, ya que es la única disponible.
Al intentar programar la FPGA con el sistema y querer conectarme a él mediante el GrmonRCP, me encontré con que no encontraba el sistema, el mensaje que me daba era *AMBA plug & play Not Found*. Esto era causado por una incorrecta implementación de la señal Hready, ya que siempre ha de estar la señal a alta, excepto cuando hemos de añadir waitstates, mientras que la implementación que había hecho de la señal, consistía en que siempre estaba a baja excepto cuando se terminaba una transferencia.
El sistema ha de trabajar a una velocidad de clock diferente al del USB, lo que implica que tendremos que realizar un sincronizador para adaptar las señales de un clock a otro. Dichas señales han de asegurar que el controlador del USB se entera de la peticiones del procesador y por lo tanto hemos de alargar las señales esto comportará que siempre que el procesador le pida algo al USB, la mayoría de veces le tendremos que añadir algún waitstate, ya que tendrá que esperar entre 0 y 3 estados (se ha de tener en cuenta que el sistema funciona a 48Mhz). Si no pusiéramos este sincronizador, el problema lo tendríamos con las restricciones temporales de las señales, ya que al ir más rápido el sistema que el controlador del USB, éste último no se enteraría de todas las peticiones que el sistema le realizase.

# 7. Resultados

Los resultados obtenidos son tantos como etapas ha tenido el proyecto. Veremos los diferentes pasos realizados hasta llegar a establecer una comunicación con la aplicación DMNewGUI.
El primer resultado obtenido se trata de la simulación del sistema en el template leon3-altera-ep2s60-ddr, el programa es fichero *hello.c* (apéndice de Ejemplos de Código fuente). Al simular nos muestra por pantalla la configuración del sistema y los printf mostrados. Esto lo puede realizar ya que el sistema cuando se inicializa, aprovecha el sistema de plug & play para detectar los diferentes componentes y cuando los detecta, saca su configuración por la consola del modelsim. Una vez ha realizado la inicialización ejecuta el programa existente en la memoria PROM.

```
# LEON3 Altera EP2C60 SSRAM/DDR CNM System Configuration
# Author of the system configuration: Sergio Morlans Iglesias
# GRLIB Version 1.0.19, build 3188
# Target technology: stratixii ,  memory library: stratixii
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area at 0xfff00000, 1 Mbyte
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        JTAG Debug Link
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 256 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        Leon3 Debug Support Unit
# ahbctrl:       I/O port at 0xfffd0000, size 256 byte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 8, irq 2
# ahbjtag AHB Debug JTAG rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 1 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
# clkgen_altera: altpll sdram/pci clock generator, version 1
# clkgen_altera: Frequency 50000 KHz, PLL scaler 6/2
# ** Note: Stratix PLL locked to incoming clock
#    Time: 110 ns  Iteration: 2  Instance: /testbench/d3/clkgen0/
                  alt/v/sdclk_pll/nosd/altpll0/stratix_altpll/m0
#
# Hello World
#
# Goodbye World
#
...
#
# Hello World
#
# Goodbye World
#
```

Nota: No he puesto todos los printfs que muestra el sistema.

Después de sintetizar el sistema y programar la FPGA y la memoria PROM, ejecutamos el sistema mediante el GrmonRCP y obtenemos los siguientes resultados:


Figura 7.1: Resultados Hello World

Una vez, hemos adaptado la IP del USB, mediante el wrapper, el primera paso es ver que la inicialización se efectúa correctamente, una vez has programado la FPGA y la memoria flash. Para ello inicializaremos el programa DMnewGUI y observaremos si detecta o no el USB, en caso afirmativo ya tenemos la comunicación establecida y podremos pasar a comprobar si interpreta bien los datos cuando se realizan transferencias.

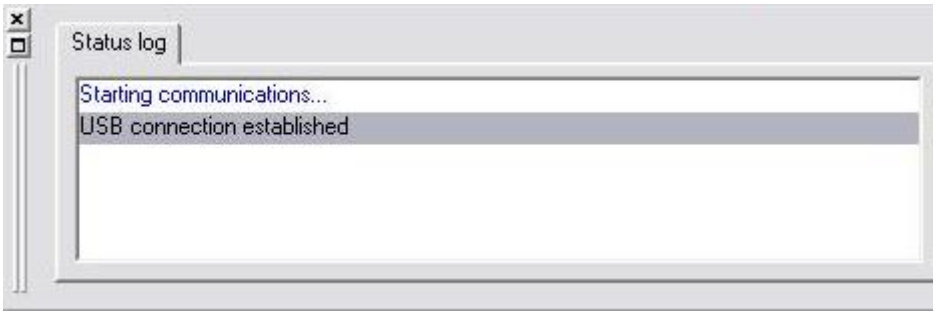



Figura 7.2: Dispositivo conectado

Resultado final del proyecto, intercambio de datos entre el PC (programa DMnewGUI y el LEON3). El programa final utilizado es el *usb.c* su código lo podemos ver en el apéndice de Ejemplos de Código fuente. El resultado obtenido lo podemos ver en el HyperTerminal, ya que hay los printf correspondientes para poder observar los pasos seguidos por el programa.

Estos resultados obtenidos que podemos ver mediante el HyperTerminal, nos vienen a decir, que

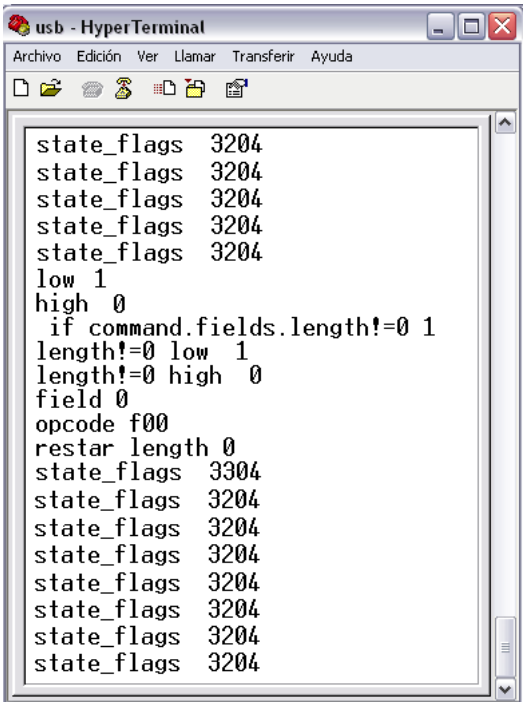



Figura 7.3: Resultado final

el primer paso, que es el de leer los datos enviados por el DMNewgui, se han leído correctamente:

```
Low 1
High 0
```

El siguiente paso es comprobar la cantidad de datos que se enviarán (en este caso 1):

```
if command.fields.length=!0 1
```

Posteriormente, realizamos las escrituras:

```
Length!=0 low 1
Length!=0 high 0
```

Finalmente realizamos la lectura de los datos y le respondemos:

```
Field 0
Opcode f00
```

Decrementamos el número de lecturas pendientes a realizar, si es cero no se vuelve a ejecutar el bucle de lectura, pero sí el bucle general del programa (while(1)).

# 8. Conclusiones

## 8.1 Objetivos realizados

Para poder explicar brevemente los objetivos que se han conseguido realizar, primero de todo se hará un breve recordatorio sobre los objetivos iniciales:

* Setup del entorno de desarrollo del microprocesador LEON3.
* Adaptación de un controlador USB.
* Desarrollar el firmware del microcontrolador.
* Diseño de un microcontrolador basado en el LEON3 con la funcionalidad necesaria para la lectura, test y caracterización del CI Medipix2 MXR.
* Diseño del software necesario para el control del sistema.

De los objetivos iniciales planteados se ha realizado el Setup del sistema. Se han seguido los pasos necesarios para generar un sistema para una placa de desarrollo NIOS II Development Board con una Stratix II, y compilar un software en c que se ejecutaría en el sistema. Posteriormente hemos generado un wrapper para adaptar el controlador USB al bus AMBA. Esta adaptación a tenido dos fases diferenciadas, la primera fue la de generar las señales que hagan de interface entre el bus AMBA y el controlador de USB. La segunda fase consistió en sincronizar el sistema LEON3-AMBA con el controlador a nivel de clock de reloj, porque el sistema tiene una frecuencia mayor que el controlador. Estos dos pasos son los que más costo han supuesto a nivel temporal, ya que eran los que entrañaban una dificultad más elevada. El siguiente paso fue generar el firmware para el sistema que establezca una comunicación con la aplicación DMNewGUI. Con estos objetivos conseguidos, podemos decir que se ha realizado un microcontrolador basado con el LEON3, que establece comunicación con la aplicación DMNewGUI que es la que tiene la funcionalidad necesaria para la lectura, test y caracterización del CI Medipix2 MXR.

## 8.2 Futuras ampliaciones

Dentro de esta apartado, creo que una de las ampliaciones, a realizar, debería ser que el wrapper generado para el controlador USB permitiera trabajar con DMA, ya que el controlador sí que lo permite. Con esto se conseguiría no depender de si el microprocesador esta disponible o no, para la realización de la transferencia de datos, únicamente dependeriamos de que el bus estuviese ocupado o no.
Otra posibilidad es la de que la memoria ROM esté dentro de la FPGA, mediante el AHBROM[1], para conseguir esto se ha realizado una aplicación ahbrom2mif.exe(Ver apéndice Software generador de AHBROM), pero se ha carecido de tiempo para realizarlo.
Desde mi punto de vista la ampliación más importante deberia ser la de generar una aplicación propia con toda la funcionalidad deseada para la lectura, test y caracterización del ROIC Medipix, esto nos ofrece una personalización adaptada a las necesidades del proyecto.

## 8.3 Valoración personal

A nivel personal, este trabajo me ha servido para adentrarme en la vertiente hardware de la informática. Durante la carrera se abarca más ampliamente el mundo Software que el Hardware, lo que ha hecho que cueste cambiar de forma de pensar, ya que conceptos esenciales cuando trabajas

[1] Core que implementa una memoria PROM de 32-bits de anchura que trabaja como un slave del bus AHB. Se puede realizar cualquier tipo de accesos de los que permite el bus AHB, así como accesos de 1 ó 2 bytes (media palabra).

a nivel hardware me han costado más de la cuenta asimilarlos. Ha sido un trabajo duro e intensivo, he programado por primera vez una FPGA (sin seguir un tutorial como se hacia en prácticas durante la carrera), he asimilado conceptos como los waitstates de una memoria o que no todo el VHDL que se simula es sintetizable (teóricamente lo sabes, pero hasta que no programas una FPGA, no te das cuenta de las limitaciones que eso supone), entre otros conceptos. Me ha servido para poder estar inmerso en el ambiente laboral de una empresa/centro de investigación como es el CNM-IMB (CSIC) y para poder ver la dinámica de trabajo y las exigencias existentes.
Mi valoración del proyecto es muy positiva porque me ha enriquecido muchísimo en muy poco tiempo, como también me ha preparado para la dinámica de trabajo que encontraré una vez finalice la carrera.

# Acrónimos

**AHB** Advanced High-performance Bus

**AHDL** Altera HDL

**AMBA** Advanced Microcontroller Bus Architecture

**AMP** Multiprocesamiento Asimétrico

**APB** Advanced Peripheral Bus

**ARM** Advanced RISC Machines

**ASB** Advanced System Bus

**ASIC** Circuito Integrado para Aplicaciones Específicas

**BCC** Bare-C Cross-Compiler System

**BDW** Basic Windows Descriminator

**CAD** Computer Aided Design

**CERN** Organización Europea para la Investigación Nuclear

**CI** Circuito Integrado

**CNM** Centro Nacional de Microelectrónica

**CPU** Central Processing Unit

**CSIC** Consejo Superior de Investigaciónes Científicas

**DDR** Double Data Rate

**DMA** direct memory access

**DRC** Design Rule Checker

**DSU** Debug System Unit

**ESA** Agencia Espacial Europea

**FPGA** Field Programmable Gate Array

**FPU** Unidad de Punto Flotante

**GNU** GNU Compiler Collection

**GPL** General Public License

**HDL** Hardware Description Lenguage

**IFAE** Institut de Física d'Altes Energies

**IMB** Instituto de Microelectrónica de Barcelona

**IP** Intellectual Property

**JHLD** Java HDL

**LVS** Layout Versus Schematic

**MEMS** Micro-Electro-Mechanical Systems

**MMU** Un Controlador de gestión de Memoria

**MTU** Modular Timer Unit

**MUROS** Medipix2 re-Usable Readout System

**NEMS** Nano-Electro-Mechanical Systems

**NIKHEF** National Institute for Nuclear Physics and High Energy Physics

**OCP** Open Core Protocol

**PCI** Peripheral Component Interconnect

**PLL** Phase-Locked Loops

**PROM** Programmable Read-Only Memory

**RAM** Random Access Memory

**ROM** Read Only Memory

**RISC** Reduced Instruction Set Computer

**ROHS** Restriction of Hazardous Substances

**ROIC** Read Out Integrated Circuit

**SDRAM** Synchronous Dynamic Random Access Memory

**SMP** Multiprocesamiento Simétrico

**SOC** System-On-Chip

**SRAM** Static Random Access Memory

**SSRAM** Synchronous Static Random Access Memory

**UART** Universal Asinchronous Receiver/Transmitter

**USB** Universal Serial Bus

**VHDL** VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

**VSIA** Virtual Socket Interface Alliance

# Bibliografía


[Vhdl] Lenguaje Estándard de Diseño Electrónico.
Autores: Lluís Terés, Yago Torroja, Serafin Olcoz, Eugenio Villar
Ed. Macgraw Hill

[Thesis] Characterization Of Solid-State Detectors For Imaging Applications Autor: Marino Maiorino Director: Dr.Manel Martinez Rodríguez Tutor: Prof. Enrique Fernandez Sanchez

[Vhdl Secuencial] http://www.dte.upct.es/personal/andres_iborra/docencia/elec_ind/pdfs/VHDLSecuencial.pdf

[VHDL Interactivo] http://www.cannic.uab.es/Docencia/VHDLI06/VHDLI06.htm

[Aeroflex Gaisler] http://www.gaisler.com/cms/

[Cypress] http://www.cypress.com/

[Altera] http://www.altera.com/

[CERN] http://medipix.web.cern.ch/MEDIPIX/

[AMBA] http://www.arm.com/

[LEON] http://tech.groups.yahoo.com/group/leon_sparc/

[DCISII] http://www.cannic.uab.es/Docencia/DCisII/DCIsIIcurs0809.htm

[Clock Management] http://www.altera.com/products/devices/stratix-fpgas/about/stx-clock-management.html

[SREC Format] http://linux.die.net/man/5/srec

[CHS] Información sobre SOC y Plataformas sacada de los apuntes de la asignatura Codisseny Hardware Software de Ingenieria Informática.

[DCIS] Información sobre estilos de materialización de Circuitos Integrados sacada de los apuntes de la asignatura de Diseño de Circuitos Integrados I de Ingenieria Informática.

# Apéndices

## Apéndice A: Formato de ficheros SREC

Un fichero Srec es una forma sencilla de codificación de datos binarios. Consiste en una secuencia de strings de caracteres en formato ASCII. El tamaño ha de ser igual o menor a 78 bytes de longitud. Un record es una línea que se acaba con un cambio de línea.
Tiene los siguientes campos:


Figura 8.1: Campos del formato

* Type:Campo de longitud 2 que nos marca el tipo de campo de datos

| Record | Description | Address Bytes | Data Sequence |
|---|---|---|---|
| **S0** | Block header | 2 | Yes |
| **S1** | Data sequence | 2 | Yes |
| **S2** | Data sequence | 3 | Yes |
| **S3** | Data sequence | 4 | Yes |
| **S5** | Record count | 2 | No |
| **S7** | End of block | 4 | No |
| **S8** | End of block | 3 | No |
| **S9** | End of block | 2 | No |

Figura 8.2: Tipo de Campo de Datos

* Count: Campo de longitud 2, en valor hexadecimal que nos indica el número de caracteres (n) que nos indica.
* Address: Campo que puede tener una longitud de 4, 6 u 8 caracteres. Muestra la dirección de datos en que los datos se carga en memoria.
* Data: La longitud de los campos es igual a 2n − (longitud de la address), es decir, 2n = longitud de address + longitud de Data.
* Checksum: Campo de longitud 2 caracteres, es el byte menos significativo de un complemento a uno de la suma de los valores representados por los dos pares de dígitos hexadecimales que componen la longitud de registro, la dirección y los campos código/datos.

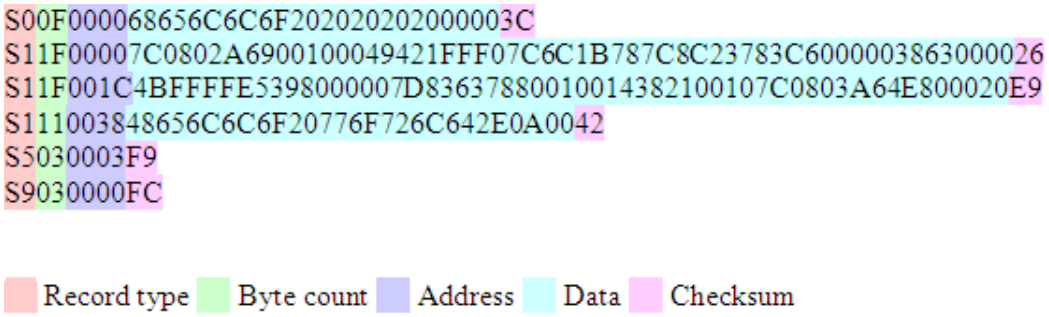



Figura 8.3: Ejemplo Formato SREC

# Apéndice B: Software generador de AHBROM

Para poder explicar el software generador del fichero en formato .mif, partiré del fichero ahbrom.vhd que te genera el sistema, y posteriormente explicaré en que consiste el formato .mif y el formato original del ahbrom.vhd. Es un programa en C, compilado mediante el gcc[2] y ejecutado desde línea de comandos (ya sea desde Linux o Solaris). El siguiente código del script compile.sh muestra la forma que genera el ejecutable:

```
rm -f ahbrom2mif.exe
gcc ahbrom2mif.c -o ahbrom2mif.exe
./ahbrom2mif.exe ahbrom.vhd rom.mif
```

Primeramente se borra el ejecutable ya existente con el comando rm. Posteriormente compilamos el programa (el ejecutable es el nombre que se pone después de la opción -o) y así creamos el ejecutable (ahbrom2mif.exe). Formato del ahbrom2mif.exe:

```
Format:
  ahbrom2mif.exe [path]file [path]file
Options:
1-[path]file: You have to choose a vhd file to convert
            (The path is optional)
2-[path]file: You have to choose a file name with mif extension
            (The path is optional)
```

Por último ejecutamos el programa con dos argumentos. El primero es el fichero VHDL que tiene la definición del ahbrom y el segundo es el nombre que se le dará al fichero generado en formato mif. Para hacer la explicación más sencilla y clara la haremos con un ejemplo paso a paso:

El fichero ahbrom.vhd en su architecture tiene el siguiente código:

```
constant abits : integer := 15;
constant bytes : integer := 19552;

constant hconfig : ahb_config_type := (
  0 => ahb_device_reg ( VENDOR_GAISLER, GAISLER_AHBROM, 0, 0, 0),
  4 => ahb_membar(haddr, '1', '1', hmask), others => zero32);

signal romdata : std_logic_vector(31 downto 0);
signal addr : std_logic_vector(abits-1 downto 2);
signal hsel, hready : std_ulogic;

begin

  ahbso.hresp   <= "00";
  ahbso.hsplit  <= (others => '0');
  ahbso.hirq    <= (others => '0');
  ahbso.hcache  <= '1';
  ahbso.hconfig <= hconfig;
  ahbso.hindex  <= hindex;

  reg : process (clk)
  begin
    if rising_edge(clk) then
      addr <= ahbsi.haddr(abits-1 downto 2);
    end if;
  end process;

  p0 : if pipe = 0 generate
    ahbso.hrdata  <= romdata;
    ahbso.hready  <= '1';
```

[2]Conjunto de compiladores considerados estándar para sistemas derivados de Unix y algunos propietarios como Mac OS. Abarcan diferentes lenguajes como el C, C++, Fortrand...

```
  end generate;

  p1 : if pipe = 1 generate
    reg2 : process (clk)
    begin
      if rising_edge(clk) then
hsel <= ahbsi.hsel(hindex) and ahbsi.htrans(1);
hready <= ahbsi.hready;
ahbso.hready <=  (not rst) or (hsel and hready) or
  (ahbsi.hsel(hindex) and not ahbsi.htrans(1) and ahbsi.hready);
ahbso.hrdata  <= romdata;
      end if;
    end process;
  end generate;

  comb : process (addr)
  begin
    case conv_integer(addr) is
    when 16#00000# => romdata <= X"88100000";
    when 16#00001# => romdata <= X"09000004";
    when 16#00002# => romdata <= X"81C122D8";
    ...
    when 16#01317# => romdata <= X"00000000";
    when 16#01318# => romdata <= X"00000000";
    when others => romdata <= (others => '-');
    end case;
end process;
```

El contenido de la memoria es lo que se encuentra dentro del *´case conv_integer(addr) is´* y por lo tanto son las líneas de código que nos interesa pasar a formato mif.

El Formato mif, tiene los siguientes parámetros:

```
DEPTH = 4988;       -- The size of memory in words
WIDTH = 32;         -- The size of data in bits
ADDRESS_RADIX = HEX; -- The radix for address values
DATA_RADIX = HEX;     -- start of (address : data pairs)
CONTENT             -- start of (address : data pairs)
BEGIN

00000: 88100000;
00001: 09000004;
00002: 81C122D8;
...
01317: 00000000;
01318: 00000000;

END;
```

Como podemos observar, tiene 5 líneas iniciales, donde se encuentra el tamaño de la memoria en palabras, la profundidad de datos (número de bits), y la base que utiliza el formato de las direcciones y la de datos (en este caso hexadecimal). El contenido existente entre el BEGIN y el END es el contenido de la memoria y por lo tanto es lo que el programa cogerá del ahbrom.vhd y lo meterá en el fichero .mif Una vez tenemos el fichero .mif generado, deberemos modificar el ahbrom.vhd para que llame a un romgen2 (memoria dual port), y que éste cargue el fichero .mif.
El fichero romgen2.vhd es una memoria RAM dual-port, tiene un puerto de direcciones tanto para las operaciones de lectura como la de escritura. La memoria RAM dual-port se puede configurar mediante el MegaWizard Plug-In.
El fichero ahbrom.vhd, una vez modificado quedará de la siguiente manera:

```
library ieee;
```

```
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;

entity ahbrom is
  generic (
    hindex  : integer := 0;
    haddr   : integer := 0;
    hmask   : integer := 16#fff#;
    pipe    : integer := 0;
    tech    : integer := 0;
    kbytes  : integer := 1);
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbsi   : in  ahb_slv_in_type;
    ahbso   : out ahb_slv_out_type
  );
end;

architecture rtl of ahbrom is
constant abits : integer := 15;
constant bytes : integer := 19552;

constant hconfig : ahb_config_type := (
  0 => ahb_device_reg ( VENDOR_GAISLER, GAISLER_AHBROM, 0, 0, 0),
  4 => ahb_membar(haddr, '1', '1', hmask), others => zero32);

signal romdata : std_logic_vector(31 downto 0);
signal addr : std_logic_vector(abits-1 downto 2);
signal hsel, hready : std_ulogic;

component ROMGEN2 IS
PORT(
address : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
clock : IN STD_LOGIC ;
q : OUT STD_LOGIC_VECTOR (31 DOWNTO 0));
END component;

begin

  ahbso.hresp   <= "00";
  ahbso.hsplit  <= (others => '0');
  ahbso.hirq    <= (others => '0');
  ahbso.hcache  <= '1';
  ahbso.hconfig <= hconfig;
  ahbso.hindex  <= hindex;

  reg : process (clk)
  begin
    if rising_edge(clk) then
      addr <= ahbsi.haddr(abits-1 downto 2);
    end if;
  end process;

  p0 : if pipe = 0 generate
    ahbso.hrdata  <= romdata;
    ahbso.hready  <= '1';
  end generate;

  p1 : if pipe = 1 generate
    ahbso.hrdata  <= romdata;
```

```
reg2 : process (clk)
    begin
      if rising_edge(clk) then
hsel <= ahbsi.hsel(hindex) and ahbsi.htrans(1);
hready <= ahbsi.hready;
ahbso.hready <=  (not rst) or (hsel and hready) or
  (ahbsi.hsel(hindex) and not ahbsi.htrans(1) and ahbsi.hready);

      end if;
    end process;
  end generate;

romgen_inst : romgen2 PORT MAP (
address  => addr,
clock  => clk,
q  => romdata
);

  -- pragma translate_off
  bootmsg : report_version
  generic map ("ahbrom" & tost(hindex) &
  ": 32-bit AHB ROM Module,  " & tost(bytes/4) & " words, " & tost(abits-2)
   & " address bits" );
  -- pragma translate_on
  end;
```

## Romgen2.vhd:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY altera_mf;
USE altera_mf.all;

ENTITY ROMGEN2 IS
PORT(
address : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
clock : IN STD_LOGIC ;
q : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END ROMGEN2;

ARCHITECTURE SYN OF romgen2 IS

SIGNAL sub_wire0 : STD_LOGIC_VECTOR (7 DOWNTO 0);

COMPONENT altsyncram
GENERIC (
clock_enable_input_a : STRING;
clock_enable_output_a : STRING;
init_file               : STRING;
intended_device_family : STRING;
lpm_hint : STRING;
lpm_type : STRING;
numwords_a : NATURAL;
operation_mode : STRING;
outdata_aclr_a : STRING;
outdata_reg_a : STRING;
widthad_a : NATURAL;
width_a               : NATURAL;
width_byteena_a : NATURAL
);
PORT (
clock0        : IN STD_LOGIC ;
address_a     : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
```

```
q_a        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

BEGIN
q    <= sub_wire0(7 DOWNTO 0);
altsyncram_component : altsyncram
GENERIC MAP (
clock_enable_input_a => "BYPASS",
clock_enable_output_a => "BYPASS",
init_file => "rom.mif",
intended_device_family => "Stratix II",
lpm_hint => "ENABLE_RUNTIME_MOD=NO",
lpm_type => "altsyncram",
numwords_a => 32,
operation_mode => "ROM",
outdata_aclr_a => "NONE",
outdata_reg_a => "CLOCK0",
widthad_a => 5,
width_a => 8,
width_byteena_a => 1
)
PORT MAP(
clock0 => clock,
address_a => address,
q_a => sub_wire0
);
END SYN;
```

# Apéndice C: Ejemplos de Código Fuente

## Hello.c

```
main(){
int i = 0;
while(i<40){
printf("\nHello World\n");
i++;
printf("\nGoodbye World\n");
}
}
```

## usb.c

```
#include<stdio.h>
#include "alt_types.h"
#include "typedefs.h"

/**** program definitions ***/

volatile short state_flags=0;
volatile short int pe=0;
xray_usb_command command;

volatile short int *usb = (short int *) USB_ADDR;
volatile short int *reg = (short int *) READREG_ADDR;

/*** Interrupt definitions ***/
extern void *catch_interrupt(void func(), int irq);

int *lreg = (int *) IRQCTRL_ADDR;

#ifdef LEON3
#define ILEVEL 0x200
#define ICLEAR 0x20c
#define IMASK  0x240
#define IFORCE 0x208
#endif

enable_irq (int irq){
  lreg[ILEVEL/4] = (1 << irq);
lreg[ICLEAR/4] = (1 << irq); // clear any pending irq
lreg[IMASK/4] |= (1 << irq); // unmaks irq
}

disable_irq (int irq) { lreg[IMASK/4] &= ~(1 << irq); } // mask irq
force_irq (int irq) { lreg[IFORCE/4] = (1 << irq); } // force irq

void  SendUSBACK(alt_u32 opcode, alt_u32 field){
   usb[0]=field;
   usb[0]=opcode;
   usb[1]=1;
}

void init_usb(void){
   int i=0;
   while(state_flags!=0x3204){
      do{
      state_flags=reg[1];
      }while((state_flags & 0x1000)!=0x1000);
      while((state_flags & 0x2000)!=0x2000);
   }
}
```

```
void irqhandler(int irq){
    lreg[IMASK/4] &= ~(1 << irq);  //disable irq
    state_flags=reg[1];
    lreg[ICLEAR/4] = (0 << irq);
    lreg[IMASK/4] |= (1 << irq); //enable irq
}

//xray_usb_command
void usb_polldata32(void){
    if((state_flags & FLAGA_MASK) == FLAGA_MASK){
        command.parts.low = usb[0];
        printf("low low %x\n",command.parts.low);
        command.parts.high = usb[0];
        printf("high high  %x\n", command.parts.high);
    }
}

main(){
  int i;
  alt_u32 opcode;
  alt_u32 field;
  int TimerCount;
  /* inicializacion */
  init_usb();
  /* inicializacion ok */

  catch_interrupt(irqhandler, 11);
  enable_irq(11);
  /* interrupt enable */

  while(1){
    printf("state_flags  %x\n",state_flags);
    usb_polldata32();

    if(command.fields.length==0xEEEE) printf("COM ERROR\n");
    if(command.fields.length!=0){
      printf(" if command.fields.length!=0 x\n",command.fields.length);
      printf("length!=0 low  %x\n",command.parts.low);
      printf("length!=0 high  %x\n", command.parts.high);
      usb[0]=command.parts.low;
      usb[0]=command.parts.high;
      usb[1]=1; //send pktend
    }

    while(command.fields.length!=0){
    if((reg[1] & FLAGA_MASK) == FLAGA_MASK){
      field=usb[0];
        printf("field %x\n",field);
        opcode=usb[0];
        printf("opcode %x\n",opcode);
        command.fields.length=command.fields.length-1;
        printf("length %x\n",command.fields.length);
        if(command.fields.length==0){
           SendUSBACK(opcode, field);
             TimerCount=0;
             opcode= opcode | 0x0080 | field>>16;
             SendUSBACK(opcode, field);
        }
      }
    }
  }
}
```

# Apéndice D: Especificación del Bus AHB



## 2.2 AMBA AHB signal list

This section contains an overview of the AMBA AHB signals (see Table 2-1). A full description of each of the signals can be found in later sections of this document.

All signals are prefixed with the letter **H**, ensuring that the AHB signals are differentiated from other similarly named signals in a system design.

**Table 2-1 AMBA AHB signals**

| Name | Source | Description |
|---|---|---|
| **HCLK**<br>Bus clock | Clock source | This clock times all bus transfers. All signal timings are related to the rising edge of **HCLK.** |
| **HRESETn**<br>Reset | Reset controller | The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW signal. |
| **HADDR[31:0]**<br>Address bus | Master | The 32-bit system address bus. |
| **HTRANS[1:0]**<br>Transfer type | Master | Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY. |
| **HWRITE**<br>Transfer direction | Master | When HIGH this signal indicates a write transfer and when LOW a read transfer. |
| **HSIZE[2:0]**<br>Transfer size | Master | Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits. |
| **HBURST[2:0]**<br>Burst type | Master | Indicates if the transfer forms part of a burst. Four, eight and sixteen beat bursts are supported and the burst may be either incrementing or wrapping. |
| **HPROT[3:0]**<br>Protection control | Master | The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection.<br>The signals indicate if the transfer is an opcode fetch or data access, as well as if the transfer is a privileged mode access or user mode access. For bus masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable. |

**Table 2-1 AMBA AHB signals (continued)**

| Name | Source | Description |
|---|---|---|
| **HWDATA[31:0]**<br>Write data bus | Master | The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation. |
| **HSELx**<br>Slave select | Decoder | Each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is simply a combinatorial decode of the address bus. |
| **HRDATA[31:0]**<br>Read data bus | Slave | The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation. |
| **HREADY**<br>Transfer done | Slave | When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.<br>Note: Slaves on the bus require **HREADY** as both an input and an output signal. |
| **HRESP[1:0]**<br>Transfer response | Slave | The transfer response provides additional information on the status of a transfer.<br>Four different responses are provided, OKAY, ERROR, RETRY and SPLIT. |

AMBA AHB also has a number of signals required to support multiple bus master operation (see Table 2-2). Many of these arbitration signals are dedicated point to point links and in Table 2-2 the suffix **x** indicates the signal is from module X. For example there will be a number of **HBUSREQx** signals in a system, such as **HBUSREQarm**, **HBUSREQdma** and **HBUSREQtic**.

**Table 2-2 Arbitration signals**

| Name | Source | Description |
|---|---|---|
| **HBUSREQx**<br>Bus request | Master | A signal from bus master x to the bus arbiter which indicates that the bus master requires the bus. There is an **HBUSREQx** signal for each bus master in the system, up to a maximum of 16 bus masters. |
| **HLOCKx**<br>Locked transfers | Master | When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW. |
| **HGRANTx**<br>Bus grant | Arbiter | This signal indicates that bus master x is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when **HREADY** is HIGH, so a master gets access to the bus when both **HREADY** and **HGRANTx** are HIGH. |
| **HMASTER[3:0]**<br>Master number | Arbiter | These signals from the arbiter indicate which bus master is currently performing a transfer and is used by the slaves which support SPLIT transfers to determine which master is attempting an access.<br>The timing of **HMASTER** is aligned with the timing of the address and control signals. |
| **HMASTLOCK**<br>Locked sequence | Arbiter | Indicates that the current master is performing a locked sequence of transfers. This signal has the same timing as the **HMASTER** signal. |
| **HSPLITx[15:0]**<br>Split completion request | Slave (SPLIT-capable) | This 16-bit split bus is used by a slave to indicate to the arbiter which bus masters should be allowed to re-attempt a split transaction.<br>Each bit of this split bus corresponds to a single bus master. |

## 3.1 About the AMBA AHB

AHB is a new generation of AMBA bus which is intended to address the requirements of high-performance synthesizable designs. AMBA AHB is a new level of bus which sits above the APB and implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single cycle bus master handover
- single clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits).

### 3.1.1 A typical AMBA AHB-based microcontroller

An AMBA-based microcontroller typically consists of a high-performance system *backbone* bus, able to sustain the external memory bandwidth, on which the CPU and other *Direct Memory Access* (DMA) devices reside, plus a bridge to a narrower APB bus on which the lower bandwidth peripheral devices are located. Figure 3-1 shows both AHB and APB in a typical AMBA system.

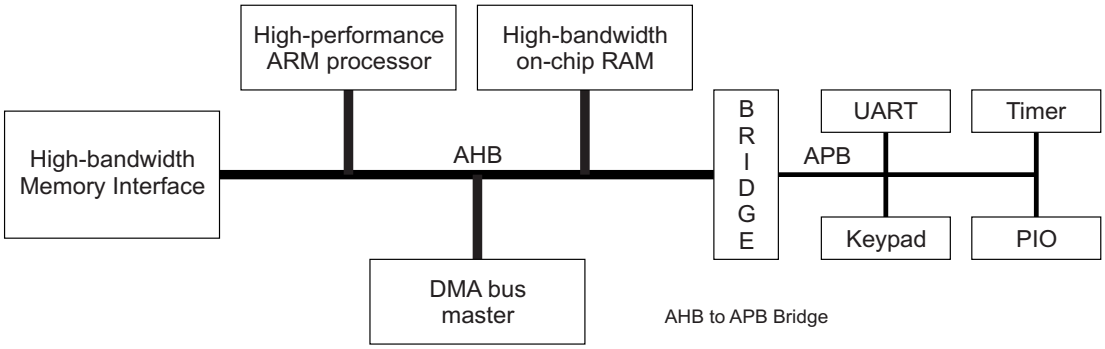



**Figure 3-1 A typical AMBA AHB-based system**

## 3.2 Bus interconnection

The AMBA AHB bus protocol is designed to be used with a central multiplexor interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer.

Figure 3-2 illustrates the structure required to implement an AMBA AHB design with three masters and four slaves.

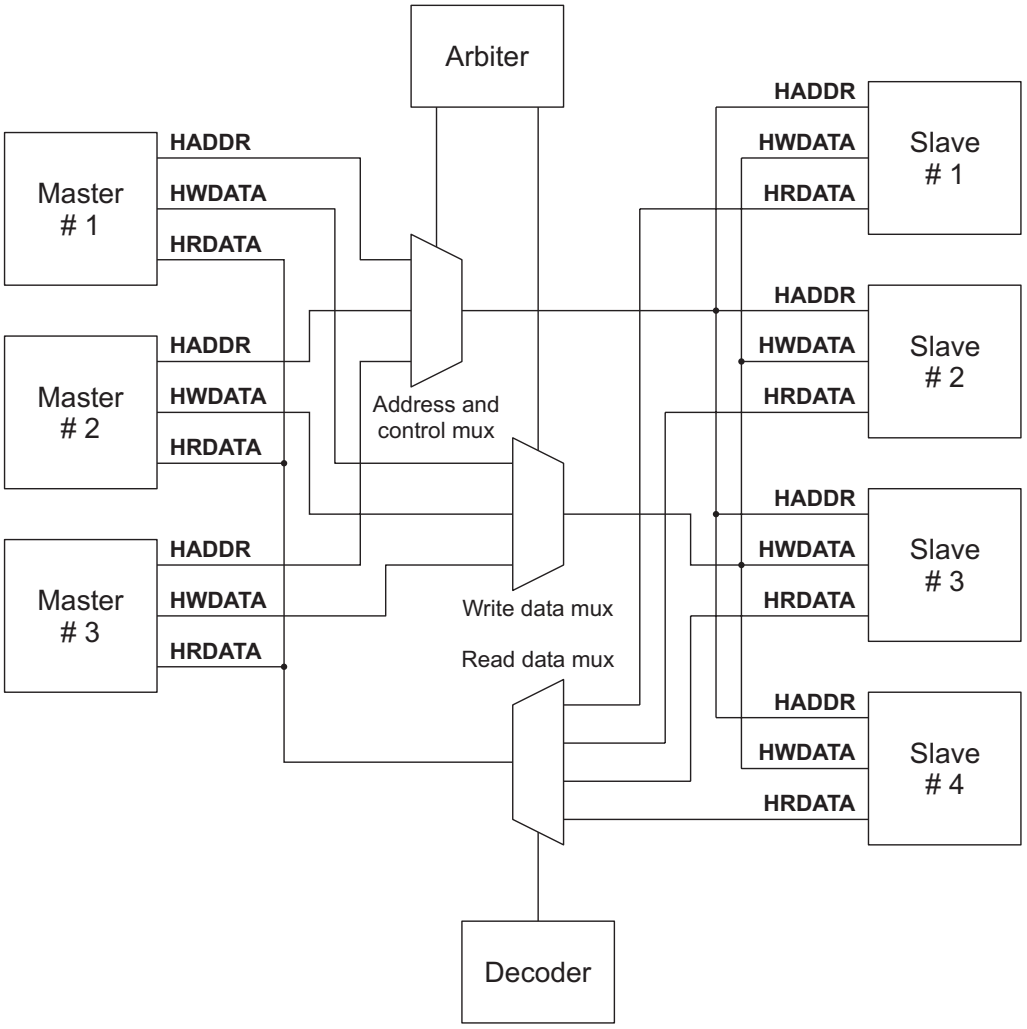



**Figure 3-2 Multiplexor interconnection**

## 3.3 Overview of AMBA AHB operation

Before an AMBA AHB transfer can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus.

A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst. Two different forms of burst transfers are allowed:

- incrementing bursts, which do not wrap at address boundaries
- wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master.

Every transfer consists of:

- an address and control cycle
- one or more cycles for the data.

The address cannot be extended and therefore all slaves must sample the address during this time. The data, however, can be extended using the **HREADY** signal. When LOW this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.

During a transfer the slave shows the status using the response signals, **HRESP[1:0]**:

OKAY — The OKAY response is used to indicate that the transfer is progressing normally and when **HREADY** goes HIGH this shows the transfer has completed successfully.

ERROR — The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.

RETRY and SPLIT — Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies it is possible for the arbiter to break up a burst and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

## 3.4 Basic transfer

An AHB transfer consists of two distinct sections:

- The address phase, which lasts only a single cycle.
- The data phase, which may require several cycles. This is achieved using the **HREADY** signal.

Figure 3-3 shows the simplest transfer, one with no wait states.

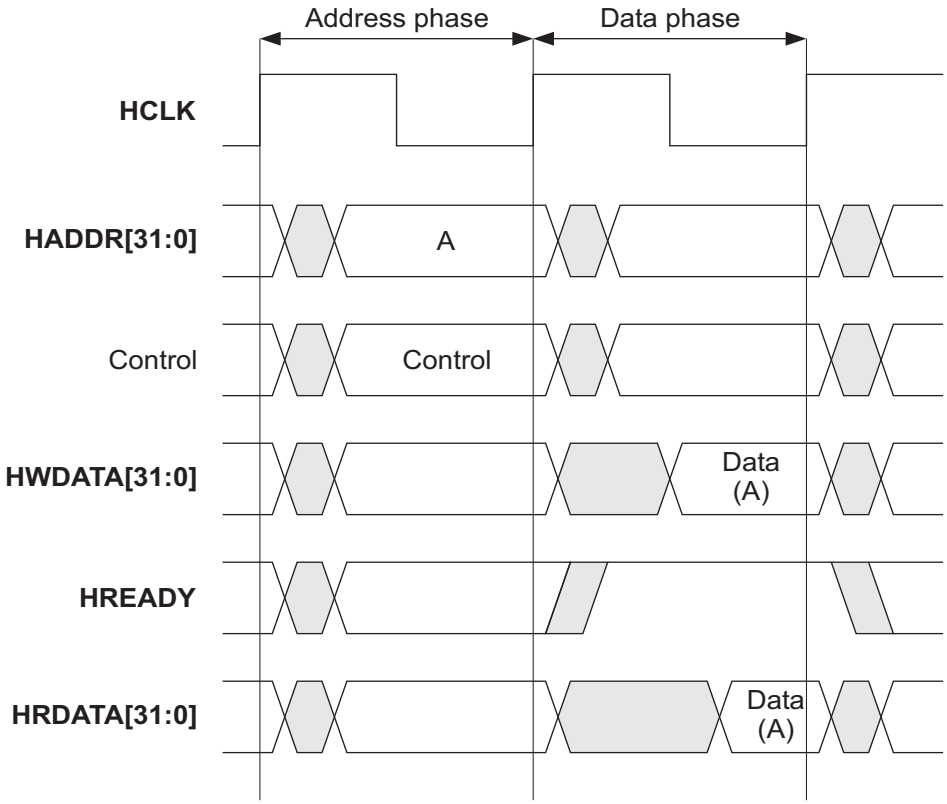



**Figure 3-3 Simple transfer**

In a simple transfer with no wait states:

- The master drives the address and control signals onto the bus after the rising edge of **HCLK**.
- The slave then samples the address and control information on the next rising edge of the clock.

- After the slave has sampled the address and control it can start to drive the appropriate response and this is sampled by the bus master on the third rising edge of the clock.

This simple example demonstrates how the address and data phases of the transfer occur during different clock periods. In fact, the address phase of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus and allows for high performance operation, while still providing adequate time for a slave to provide the response to a transfer.

A slave may insert wait states into any transfer, as shown in Figure 3-4, which extends the transfer allowing additional time for completion.

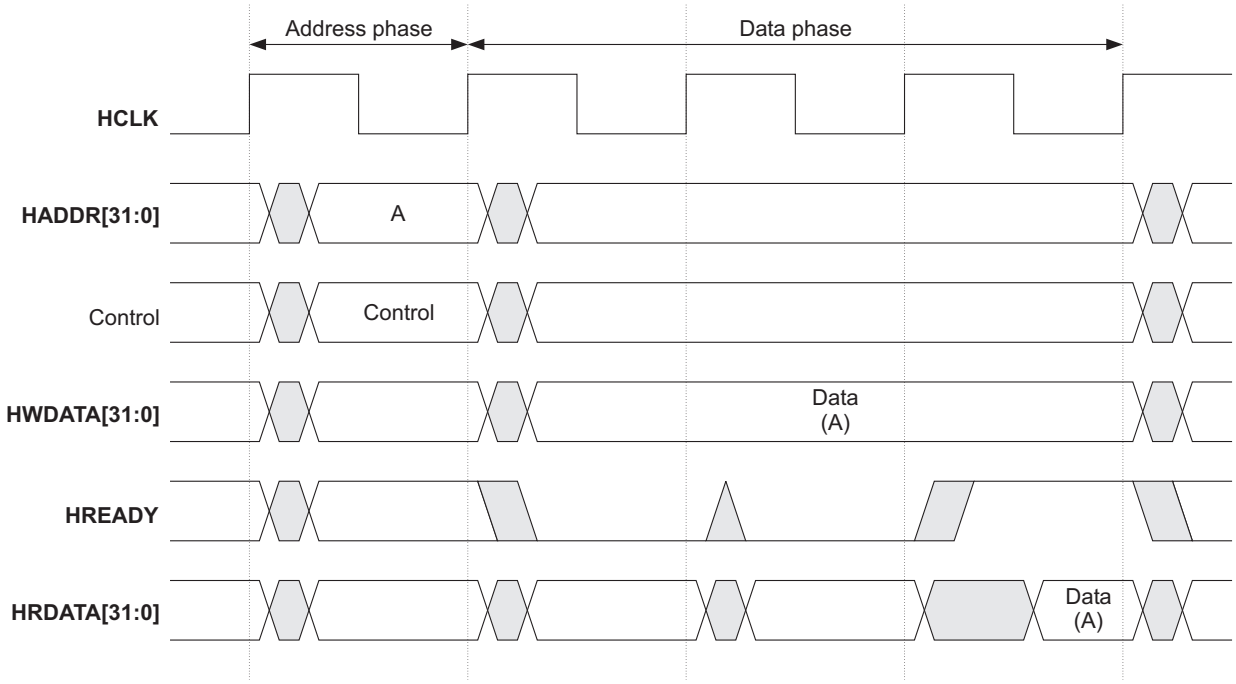



**Figure 3-4 Transfer with wait states**

**Note**

For write operations the bus master will hold the data stable throughout the extended cycles.

For read transfers the slave does not have to provide valid data until the transfer is about to complete.

When a transfer is extended in this way it will have the side-effect of extending the address phase of the following transfer. This is illustrated in Figure 3-5 which shows three transfers to unrelated addresses, A, B & C.

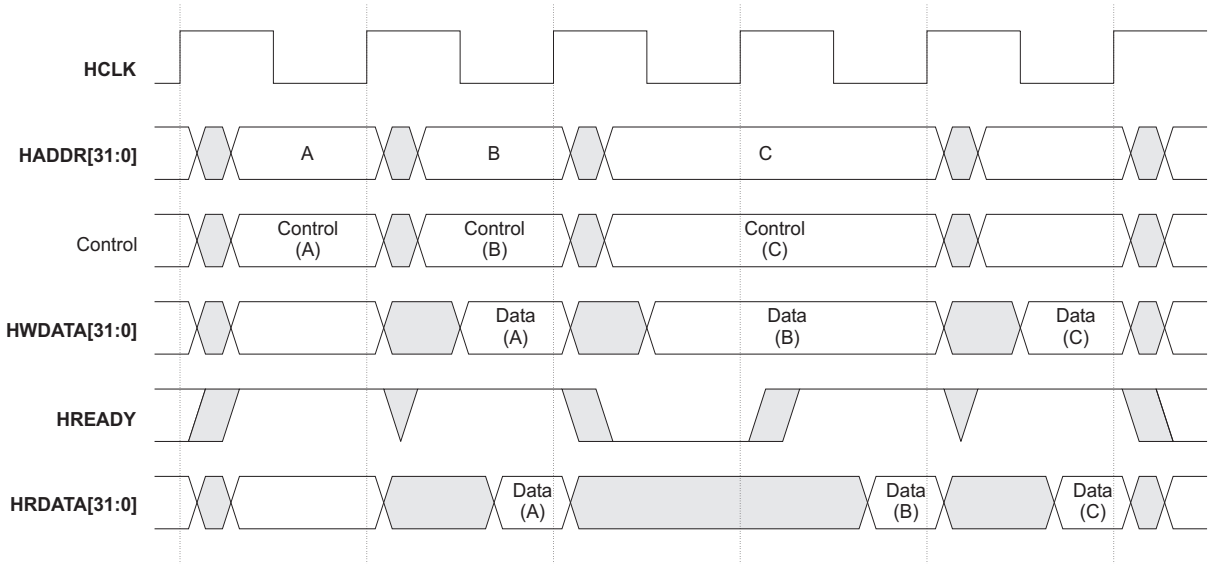



**Figure 3-5 Multiple transfers**

In Figure 3-5:

- the transfers to addresses A and C are both zero wait state
- the transfer to address B is one wait state
- extending the data phase of the transfer to address B has the effect of extending the address phase of the transfer to address C.

## 3.5 Transfer type

Every transfer can be classified into one of four different types, as indicated by the **HTRANS[1:0]** signals as shown in Table 3-1.

**Table 3-1 Transfer type encoding**

| HTRANS[1:0] | Type | Description |
|---|---|---|
| 00 | IDLE | Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer.<br>Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave. |
| 01 | BUSY | The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst.<br>The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers. |
| 10 | NONSEQ | Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer.<br>Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL. |
| 11 | SEQ | The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16). |

Figure 3-6 shows a number of different transfer types being used.

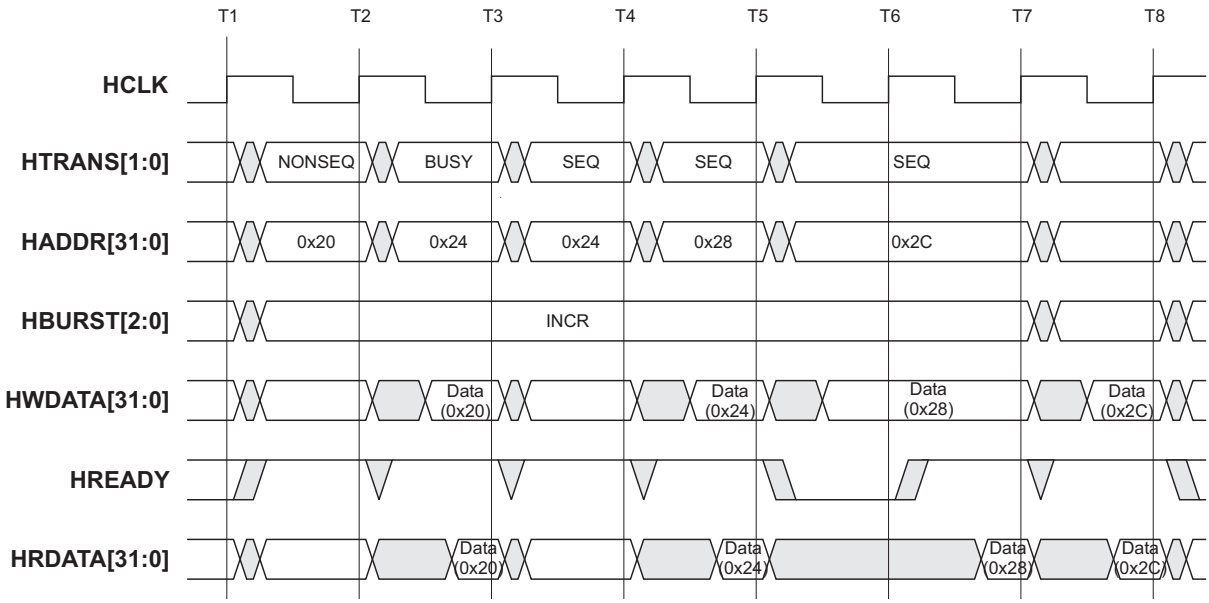



**Figure 3-6 Transfer type examples**

In Figure 3-6:

- The first transfer is the start of a burst and therefore is NONSEQUENTIAL.
- The master is unable to perform the second transfer of the burst immediately and therefore the master uses a BUSY transfer to delay the start of the next transfer. In this example the master only requires one cycle before it is ready to start the next transfer in the burst, which completes with no wait states.
- The master performs the third transfer of the burst immediately, but this time the slave is unable to complete and uses **HREADY** to insert a single wait state.
- The final transfer of the burst completes with zero wait states.

## 3.6 Burst operation

Four, eight and sixteen-beat bursts are defined in the AMBA AHB protocol, as well as undefined-length bursts and single transfers. Both incrementing and wrapping bursts are supported in the protocol:

- Incrementing bursts access sequential locations and the address of each transfer in the burst is just an increment of the previous address.
- For wrapping bursts, if the start address of the transfer is not aligned to the total number of bytes in the burst (size x beats) then the address of the transfers in the burst will wrap when the boundary is reached. For example, a four-beat wrapping burst of word (4-byte) accesses will wrap at 16-byte boundaries. Therefore, if the start address of the transfer is 0x34, then it consists of four transfers to addresses 0x34, 0x38, 0x3C and 0x30.

Burst information is provided using **HBURST[2:0]** and the eight possible types are defined in Table 3-2.

**Table 3-2 Burst signal encoding**

| HBURST[2:0] | Type | Description |
|---|---|---|
| 000 | SINGLE | Single transfer |
| 001 | INCR | Incrementing burst of unspecified length |
| 010 | WRAP4 | 4-beat wrapping burst |
| 011 | INCR4 | 4-beat incrementing burst |
| 100 | WRAP8 | 8-beat wrapping burst |
| 101 | INCR8 | 8-beat incrementing burst |
| 110 | WRAP16 | 16-beat wrapping burst |
| 111 | INCR16 | 16-beat incrementing burst |

Bursts must not cross a 1kB address boundary. Therefore it is important that masters do not attempt to start a fixed-length incrementing burst which would cause this boundary to be crossed.

It is acceptable to perform single transfers using an unspecified-length incrementing burst which only has a burst of length one.

An incrementing burst can be of any length, but the upper limit is set by the fact that the address must not cross a 1kB boundary

**Note**

The burst size indicates the number of beats in the burst, not the number of bytes transferred. The total amount of data transferred in a burst is calculated by multiplying the number of beats by the amount of data in each beat, as indicated by **HSIZE[2:0]**.

All transfers within a burst must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is A[1:0] = 00), halfword transfers must be aligned to halfword address boundaries (that is A[0] = 0).

### 3.6.1 Early burst termination

There are certain circumstances when a burst will not be allowed to complete and therefore it is important that any slave design which makes use of the burst information can take the correct course of action if the burst is terminated early. The slave can determine when a burst has terminated early by monitoring the **HTRANS** signals and ensuring that after the start of the burst every transfer is labelled as SEQUENTIAL or BUSY. If a NONSEQUENTIAL or IDLE transfer occurs then this indicates that a new burst has started and therefore the previous one must have been terminated.

If a bus master cannot complete a burst because it loses ownership of the bus then it must rebuild the burst appropriately when it next gains access to the bus. For example, if a master has only completed one beat of a four-beat burst then it must use an undefined-length burst to perform the remaining three transfers.

Examples are shown on the following pages:

- Figure 3-7 shows a *Four-beat wrapping burst* on page 3-13
- Figure 3-8 shows a *Four-beat incrementing burst* on page 3-14
- Figure 3-9 shows an *Eight-beat wrapping burst* on page 3-15
- Figure 3-10 shows an *Eight-beat incrementing burst* on page 3-15
- Figure 3-11 shows *Undefined-length bursts* on page 3-16.

The example in Figure 3-7 shows a four-beat wrapping burst with a wait state added for the first transfer.

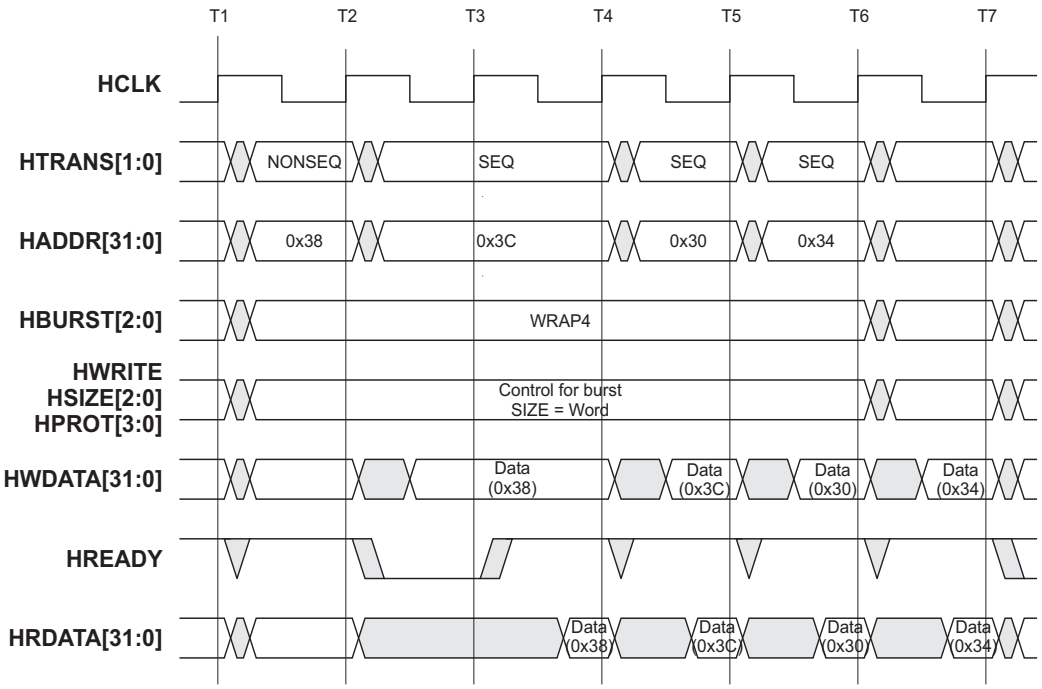



**Figure 3-7 Four-beat wrapping burst**

As the burst is a four-beat burst of word transfers the address will wrap at 16-byte boundaries, hence the transfer to address 0x3C is followed by a transfer to address 0x30. The only difference with the incrementing burst, shown in Figure 3-8 on page 3-14, is that the addresses continue past the 16-byte boundary.

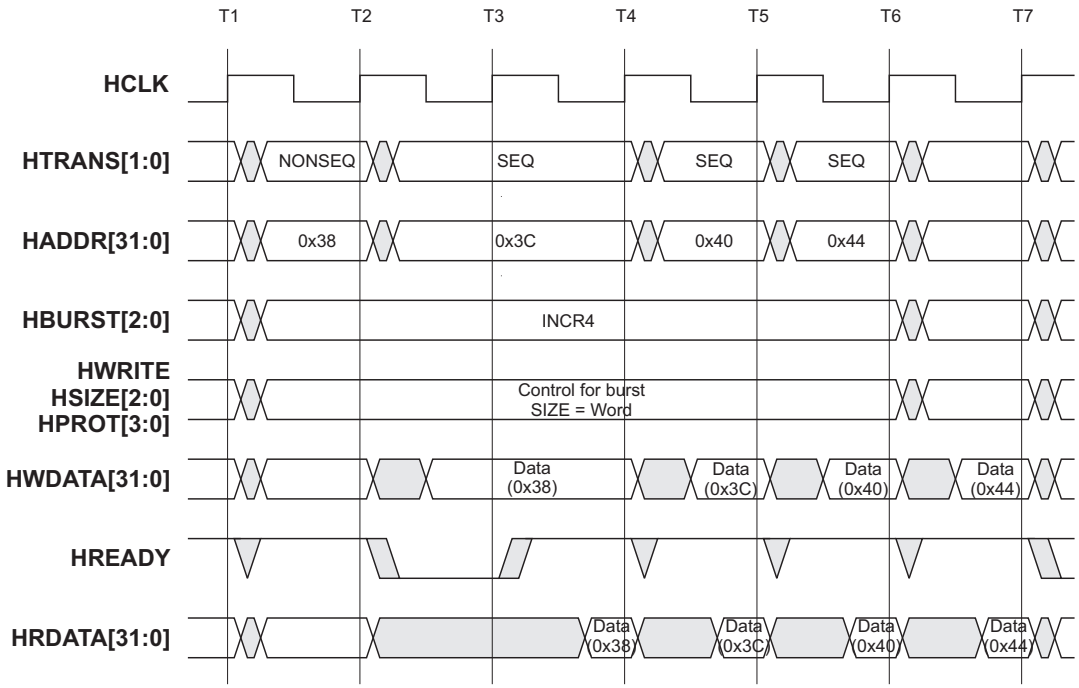



**Figure 3-8 Four-beat incrementing burst**

The example in Figure 3-9 is an eight-beat burst of word transfers.

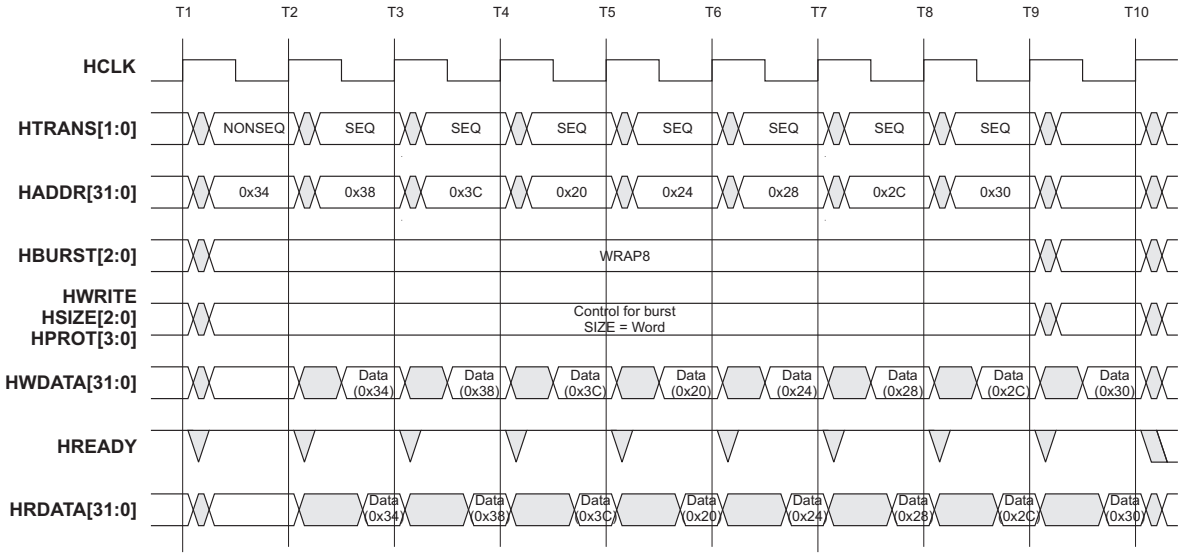



**Figure 3-9 Eight-beat wrapping burst**

The address will wrap at 32-byte boundaries and therefore address 0x3C is followed by 0x20.

The burst in Figure 3-10 uses halfword transfers, so the addresses increase by 2 and the burst is incrementing so the addresses continue to increment past the 16-byte boundary.

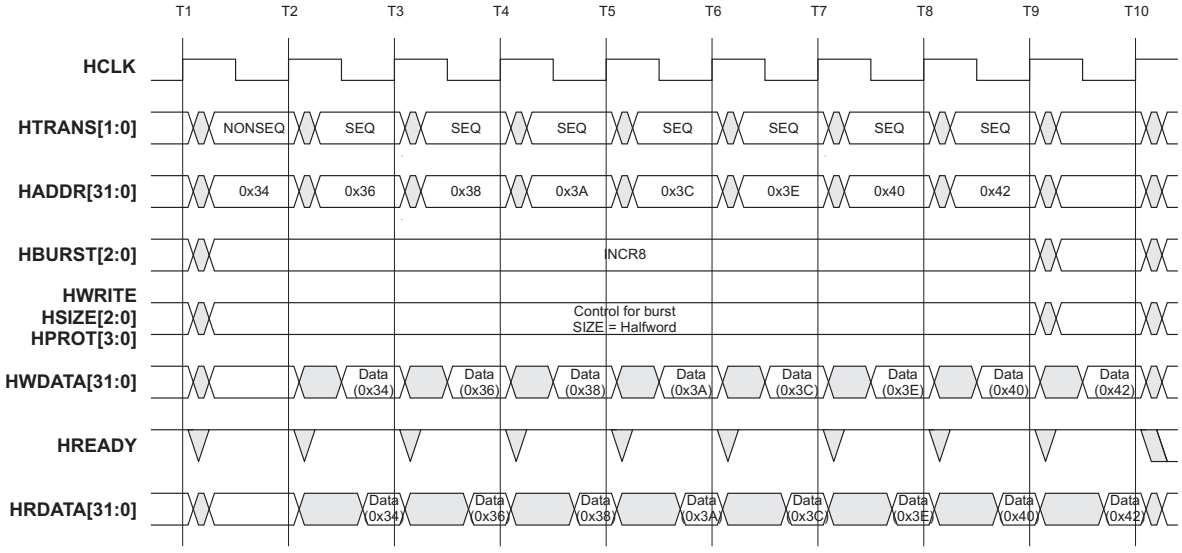



**Figure 3-10 Eight-beat incrementing burst**

The final example in Figure 3-11 shows incrementing bursts of undefined length.

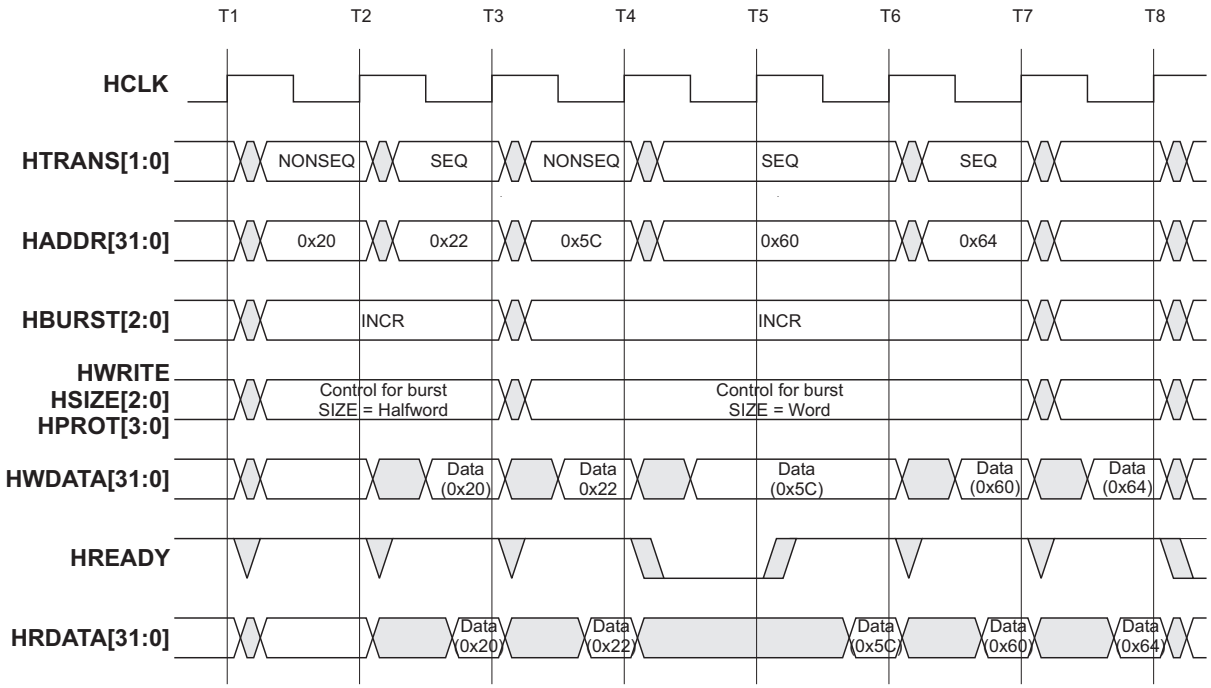



**Figure 3-11 Undefined-length bursts**

Figure 3-11 shows two bursts:

- Two halfword transfers starting at address 0x20. The halfword transfer addresses increment by 2.
- Three word transfers starting at address 0x5C. The word transfer addresses increment by 4.

## 3.7 Control signals

As well as the transfer type and burst type each transfer will have a number of control signals that provide additional information about the transfer. These control signals have exactly the same timing as the address bus. However, they must remain constant throughout a burst of transfers.

### 3.7.1 Transfer direction

When **HWRITE** is HIGH, this signal indicates a write transfer and the master will broadcast data on the write data bus, **HWDATA[31:0]**. When LOW a read transfer will be performed and the slave must generate the data on the read data bus **HRDATA[31:0]**.

### 3.7.2 Transfer size

**HSIZE[2:0]** indicates the size of the transfer, as shown in Table 3-3.

**Table 3-3 Size encoding**

| HSIZE[2] | HSIZE[1] | HSIZE[0] | Size | Description |
|---|---|---|---|---|
| 0 | 0 | 0 | 8 bits | Byte |
| 0 | 0 | 1 | 16 bits | Halfword |
| 0 | 1 | 0 | 32 bits | Word |
| 0 | 1 | 1 | 64 bits | - |
| 1 | 0 | 0 | 128 bits | 4-word line |
| 1 | 0 | 1 | 256 bits | 8-word line |
| 1 | 1 | 0 | 512 bits | - |
| 1 | 1 | 1 | 1024 bits | - |

The size is used in conjunction with the **HBURST[2:0]** signals to determine the address boundary for wrapping bursts.

### 3.7.3 Protection control

The protection control signals, **HPROT[3:0]**, provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection (see Table 3-4).

The signals indicate if the transfer is:

- an opcode fetch or data access
- a privileged mode access or user mode access.

For bus masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable.

**Table 3-4 Protection signal encodings**

| HPROT[3] cacheable | HPROT[2] bufferable | HPROT[1] privileged | HPROT[0] data/opcode | Description |
|---|---|---|---|---|
| - | - | - | 0 | Opcode fetch |
| - | - | - | 1 | Data access |
| - | - | 0 | - | User access |
| - | - | 1 | - | Privileged access |
| - | 0 | - | - | Not bufferable |
| - | 1 | - | - | Bufferable |
| 0 | - | - | - | Not cacheable |
| 1 | - | - | - | Cacheable |

Not all bus masters will be capable of generating accurate protection information, therefore it is recommended that slaves do not use the **HPROT** signals unless strictly necessary.

## 3.8 Address decoding

A central address decoder is used to provide a select signal, **HSELx**, for each slave on the bus. The select signal is a combinatorial decode of the high-order address signals, and simple address decoding schemes are encouraged to avoid complex decode logic and to ensure high-speed operation.

A slave must only sample the address and control signals and **HSELx** when **HREADY** is HIGH, indicating that the current transfer is completing. Under certain circumstances it is possible that **HSELx** will be asserted when **HREADY** is LOW, but the selected slave will have changed by the time the current transfer completes.

The minimum address space that can be allocated to a single slave is 1kB. All bus masters are designed such that they will not perform incrementing transfers over a 1kB boundary, thus ensuring that a burst never crosses an address decode boundary.

In the case where a system design does not contain a completely filled memory map an additional default slave should be implemented to provide a response when any of the nonexistent address locations are accessed. If a NONSEQUENTIAL or SEQUENTIAL transfer is attempted to a nonexistent address location then the default slave should provide an ERROR response. IDLE or BUSY transfers to nonexistent locations should result in a zero wait state OKAY response. Typically the default slave functionality will be implemented as part of the central address decoder.

Figure 3-12 shows a typical address decoding system and the slave select signals.

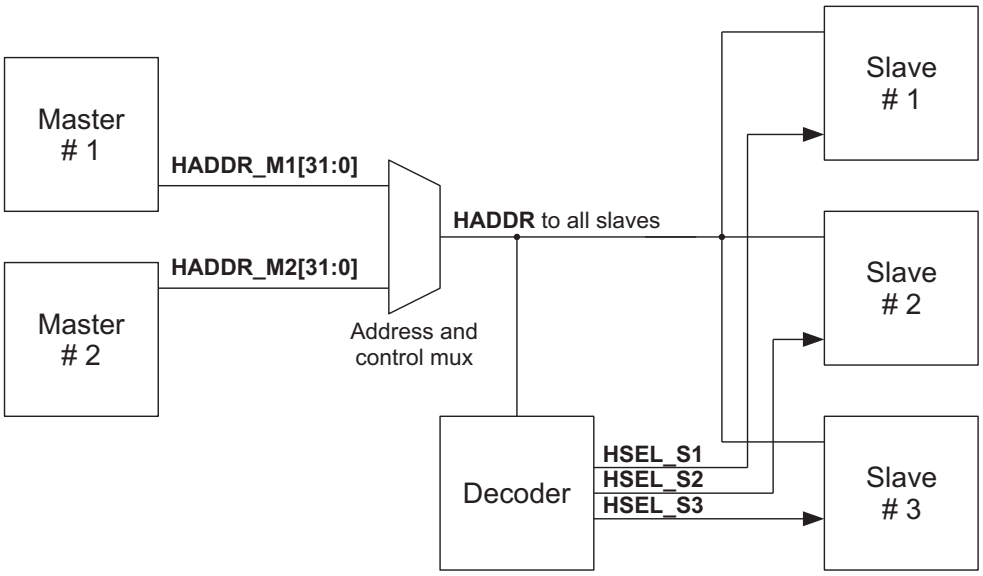



**Figure 3-12 Slave select signals**

## 3.9 Slave transfer responses

After a master has started a transfer, the slave then determines how the transfer should progress. No provision is made within the AHB specification for a bus master to cancel a transfer once it has commenced.

Whenever a slave is accessed it must provide a response which indicates the status of the transfer. The **HREADY** signal is used to extend the transfer and this works in combination with the response signals, **HRESP[1:0]**, which provide the status of the transfer.

The slave can complete the transfer in a number of ways. It can:

- complete the transfer immediately
- insert one or more wait states to allow time to complete the transfer
- signal an error to indicate that the transfer has failed
- delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers.

### 3.9.1 Transfer done

The **HREADY** signal is used to extend the data portion of an AHB transfer. When LOW the **HREADY** signal indicates the transfer is to be extended and when HIGH indicates that the transfer can complete.

**Note**

Every slave must have a predetermined maximum number of wait states that it will insert before it backs off the bus, in order to allow the calculation of the latency of accessing the bus. It is recommended, but not mandatory, that slaves do not insert more than 16 wait states to prevent any single access locking the bus for a large number of clock cycles.

### 3.9.2 Transfer response

A typical slave will use the **HREADY** signal to insert the appropriate number of wait states into the transfer and then the transfer will complete with **HREADY** HIGH and an OKAY response, which indicates the successful completion of the transfer.

The ERROR response is used by a slave to indicate some form of error condition with the associated transfer. Typically this is used for a protection error, such as an attempt to write to a read-only memory location.

The SPLIT and RETRY response combinations allow slaves to delay the completion of a transfer, but free up the bus for use by other masters. These response combinations are usually only required by slaves that have a high access latency and can make use of these response codes to ensure that other masters are not prevented from accessing the bus for long periods of time.

A full description of the SPLIT and RETRY operations can be found in *Split and retry* on page 3-24.

The encoding of **HRESP[1:0]**, the transfer response signals, and a description of each response are shown in Table 3-5.

**Table 3-5 Response encoding**

| HRESP[1] | HRESP[0] | Response | Description |
|---|---|---|---|
| 0 | 0 | OKAY | When **HREADY** is HIGH this shows the transfer has completed successfully.<br>The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses. |
| 0 | 1 | ERROR | This response shows an error has occurred. The error condition should be signalled to the bus master so that it is aware the transfer has been unsuccessful.<br>A two-cycle response is required for an error condition. |
| 1 | 0 | RETRY | The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes.<br>A two-cycle RETRY response is required. |
| 1 | 1 | SPLIT | The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete.<br>A two-cycle SPLIT response is required. |

When it is necessary for a slave to insert a number of wait states prior to deciding what response will be given then it must drive the response to OKAY.

### 3.9.3 Two-cycle response

Only an OKAY response can be given in a single cycle. The ERROR, SPLIT and RETRY responses require at least two cycles. To complete with any of these responses then in the penultimate (one before last) cycle the slave drives **HRESP[1:0]** to indicate ERROR, RETRY or SPLIT while driving **HREADY** LOW to extend the transfer for an extra cycle. In the final cycle **HREADY** is driven HIGH to end the transfer, while **HRESP[1:0]** remains driven to indicate ERROR, RETRY or SPLIT.

If the slave needs more than two cycles to provide the ERROR, SPLIT or RETRY response then additional wait states may be inserted at the start of the transfer. During this time the **HREADY** signal will be LOW and the response must be set to OKAY.

The two-cycle response is required because of the pipelined nature of the bus. By the time a slave starts to issue either an ERROR, SPLIT or RETRY response then the address for the following transfer has already been broadcast onto the bus. The two-cycle response allows sufficient time for the master to cancel this address and drive **HTRANS[1:0]** to IDLE before the start of the next transfer.

For the SPLIT and RETRY response the following transfer must be cancelled because it must not take place before the current transfer has completed. However, for the ERROR response, where the current transfer is not repeated, completion of the following transfer is optional.

Figure 3-13 shows an example of a RETRY operation.

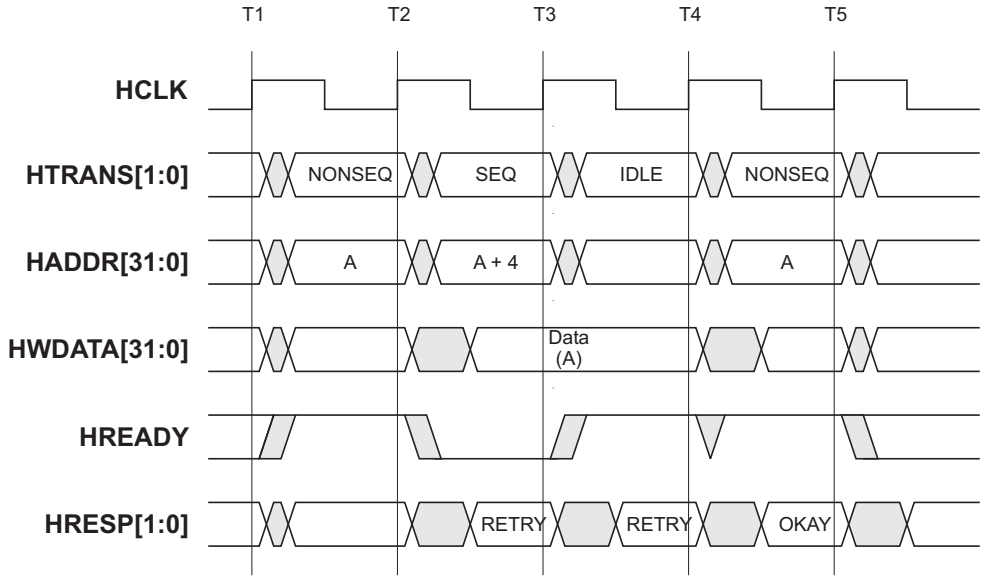



**Figure 3-13 Transfer with retry response**

The following events are illustrated:

- The master starts with a transfer to address A.
- Before the response is received for this transfer the master moves the address on to A + 4.
- The slave at address A is unable to complete the transfer immediately and therefore it issues a RETRY response. This response indicates to the master that the transfer at address A is unable to complete and so the transfer at address A + 4 is cancelled and replaced by an IDLE transfer.

Figure 3-14 shows a transfer where the slave requires one cycle to decide on the response it is going to give (during which time **HRESP** indicates OKAY) and then the slave ends the transfer with a two-cycle ERROR response.

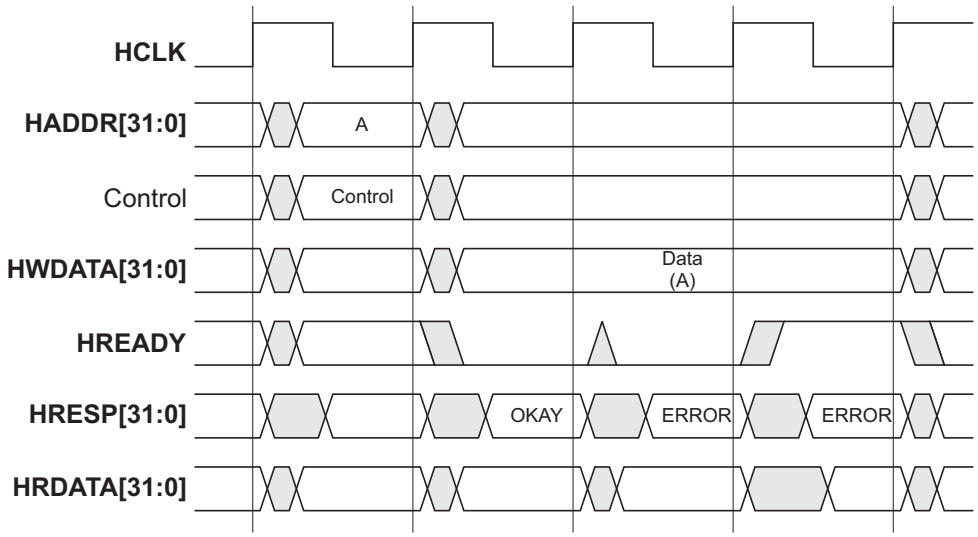



**Figure 3-14 Error response**

### 3.9.4 Error response

If a slave provides an ERROR response then the master may choose to cancel the remaining transfers in the burst. However, this is not a strict requirement and it is also acceptable for the master to continue the remaining transfers in the burst.

### 3.9.5 Split and retry

The SPLIT and RETRY responses provide a mechanism for slaves to release the bus when they are unable to supply data for a transfer immediately. Both mechanisms allow the transfer to finish on the bus and therefore allow a higher-priority master to get access to the bus.

The difference between SPLIT and RETRY is the way the arbiter allocates the bus after a SPLIT or a RETRY has occurred:

- For RETRY the arbiter will continue to use the normal priority scheme and therefore only masters having a higher priority will gain access to the bus.
- For a SPLIT transfer the arbiter will adjust the priority scheme so that any other master requesting the bus will get access, even if it is a lower priority. In order for a SPLIT transfer to complete the arbiter must be informed when the slave has the data available.

The SPLIT transfer requires extra complexity in both the slave and the arbiter, but has the advantage that it completely frees the bus for use by other masters, whereas the RETRY case will only allow higher priority masters onto the bus.

A bus master should treat SPLIT and RETRY in the same manner. It should continue to request the bus and attempt the transfer until it has either completed successfully or been terminated with an ERROR response.

## 3.10 Data buses

In order to allow implementation of an AHB system without the use of tristate drivers separate read and write data buses are required. The minimum data bus width is specified as 32 bits, but the bus width can be increased as described in *About the AHB data bus width* on page 3-41.

### 3.10.1 HWDATA[31:0]

The write data bus is driven by the bus master during write transfers. If the transfer is extended then the bus master must hold the data valid until the transfer completes, as indicated by **HREADY** HIGH.

All transfers must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is A[1:0] = 00), halfword transfers must be aligned to halfword address boundaries (that is A[0] = 0).

For transfers that are narrower than the width of the bus, for example a 16-bit transfer on a 32-bit bus, then the bus master only has to drive the appropriate byte lanes. The slave is responsible for selecting the write data from the correct byte lanes. Table 3-6 on page 3-26 and Table 3-7 on page 3-26 show which byte lanes are active for a little-endian and big-endian system respectively. If required, this information can be extended for wider data bus implementations. Burst transfers which have a transfer size less than the width of the data bus will have different active byte lanes for each beat of the burst.

The active byte lane is dependent on the endianness of the system, but AHB does not specify the required endianness. Therefore, it is important that all masters and slaves on the bus are of the same endianness.

### 3.10.2 HRDATA[31:0]

The read data bus is driven by the appropriate slave during read transfers. If the slave extends the read transfer by holding **HREADY** LOW then the slave only needs to provide valid data at the end of the final cycle of the transfer, as indicated by **HREADY** HIGH.

For transfers that are narrower than the width of the bus the slave only needs to provide valid data on the active byte lanes, as indicated in Table 3-6 and Table 3-7. The bus master is responsible for selecting the data from the correct byte lanes.

A slave only has to provide valid data when a transfer completes with an OKAY response. SPLIT, RETRY and ERROR responses do not require valid read data.

**Table 3-6 Active byte lanes for a 32-bit little-endian data bus**

| Transfer size | Address offset | DATA [31:24] | DATA [23:16] | DATA [15:8] | DATA [7:0] |
|---|---|---|---|---|---|
| Word | 0 | ✓ | ✓ | ✓ | ✓ |
| Halfword | 0 | - | - | ✓ | ✓ |
| Halfword | 2 | ✓ | ✓ | - | - |
| Byte | 0 | - | - | - | ✓ |
| Byte | 1 | - | - | ✓ | - |
| Byte | 2 | - | ✓ | - | - |
| Byte | 3 | ✓ | - | - | - |

**Table 3-7 Active byte lanes for a 32-bit big-endian data bus**

| Transfer size | Address offset | DATA [31:24] | DATA [23:16] | DATA [15:8] | DATA [7:0] |
|---|---|---|---|---|---|
| Word | 0 | ✓ | ✓ | ✓ | ✓ |
| Halfword | 0 | ✓ | ✓ | - | - |
| Halfword | 2 | - | - | ✓ | ✓ |
| Byte | 0 | ✓ | - | - | - |
| Byte | 1 | - | ✓ | - | - |
| Byte | 2 | - | - | ✓ | - |
| Byte | 3 | - | - | - | ✓ |

### 3.10.3 Endianness

In order for the system to function correctly it is essential that all modules are of the same endianness and also that any data routing or bridges are of the same endianness.

Dynamic endianness is not supported, because in the majority of embedded systems, this would lead to a significant silicon overhead that is redundant.

For module designers it is recommended that only modules which will be used in a wide variety of applications should be made bi-endian, with either a configuration pin or internal control bit to select the endianness. For more application-specific blocks, fixing the endianness to either little-endian or big-endian will result in a smaller, lower power, higher performance interface.

## 3.11 Arbitration

The arbitration mechanism is used to ensure that only one master has access to the bus at any one time. The arbiter performs this function by observing a number of different requests to use the bus and deciding which is currently the highest priority master requesting the bus. The arbiter also receives requests from slaves that wish to complete SPLIT transfers.

Any slaves which are not capable of performing SPLIT transfers do not need to be aware of the arbitration process, except that they need to observe the fact that a burst of transfers may not complete if the ownership of the bus is changed.

### 3.11.1 Signal description

A brief description of each of the arbitration signals is given below:

**HBUSREQx** The bus request signal is used by a bus master to request access to the bus. Each bus master has its own **HBUSREQx** signal to the arbiter and there can be up to 16 separate bus masters in any system.

**HLOCKx** The lock signal is asserted by a master at the same time as the bus request signal. This indicates to the arbiter that the master is performing a number of indivisible transfers and the arbiter must not grant any other bus master access to the bus once the first transfer of the locked transfers has commenced. **HLOCKx** must be asserted at least a cycle before the address to which it refers, in order to prevent the arbiter from changing the grant signals.

**HGRANTx** The grant signal is generated by the arbiter and indicates that the appropriate master is currently the highest priority master requesting the bus, taking into account locked transfers and SPLIT transfers.

A master gains ownership of the address bus when **HGRANTx** is HIGH and **HREADY** is HIGH at the rising edge of **HCLK**.

**HMASTER[3:0]** The arbiter indicates which master is currently granted the bus using the **HMASTER[3:0]** signals and this can be used to control the central address and control multiplexor. The master number is also required by SPLIT-capable slaves so that they can indicate to the arbiter which master is able to complete a SPLIT transaction.

**HMASTLOCK** The arbiter indicates that the current transfer is part of a locked sequence by asserting the **HMASTLOCK** signal, which has the same timing as the address and control signals.

**HSPLIT[15:0]** The 16-bit *Split Complete* bus is used by a SPLIT-capable slave to indicate which bus master can complete a SPLIT transaction. This information is needed by the arbiter so that it can grant the master access to the bus to complete the transfer.

Further information is provided in:

- *Requesting bus access*
- *Granting bus access* on page 3-30
- *Early burst termination* on page 3-33
- *Locked transfers* on page 3-34.

### 3.11.2 Requesting bus access

A bus master uses the **HBUSREQx** signal to request access to the bus and may request the bus during any cycle. The arbiter will sample the request on the rising of the clock and then use an internal priority algorithm to decide which master will be the next to gain access to the bus.

Normally the arbiter will only grant a different bus master when a burst is completing. However, if required, the arbiter can terminate a burst early to allow a higher priority master access to the bus.

If the master requires locked accesses then it must also assert the **HLOCKx** signal to indicate to the arbiter that no other masters should be granted the bus.

When a master is granted the bus and is performing a fixed length burst it is not necessary to continue to request the bus in order to complete the burst. The arbiter observes the progress of the burst and uses the **HBURST[2:0]** signals to determine how many transfers are required by the master. If the master wishes to perform a second burst after the one that is currently in progress then it should re-assert the request signal during the burst.

If a master loses access to the bus in the middle of a burst then it must re-assert the **HBUSREQx** request line to regain access to the bus.

For undefined length bursts the master should continue to assert the request until it has started the last transfer. The arbiter cannot predict when to change the arbitration at the end of an undefined length burst.

It is possible that a master can be granted the bus when it is not requesting it. This may occur when no masters are requesting the bus and the arbiter grants access to a default master. Therefore, it is important that if a master does not require access to the bus it drives the transfer type **HTRANS** to indicate an IDLE transfer.

### 3.11.3 Granting bus access

The arbiter indicates which bus master is currently the highest priority requesting the bus by asserting the appropriate **HGRANTx** signal. When the current transfer completes, as indicated by **HREADY** HIGH, then the master will become granted and the arbiter will change the **HMASTER[3:0]** signals to indicate the bus master number.

Figure 3-15 shows the process when all transfers are zero wait state and the **HREADY** signal is HIGH.

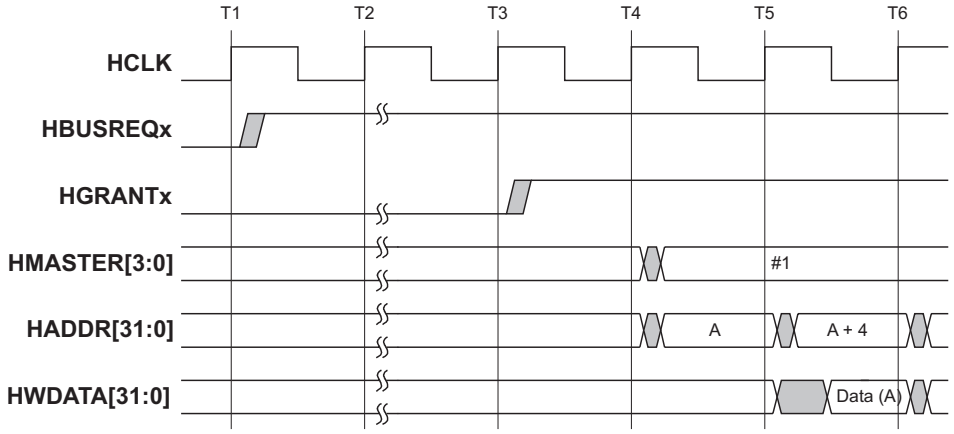



**Figure 3-15 Granting access with no wait states**

Figure 3-16 shows the effect of wait states on the bus handover.

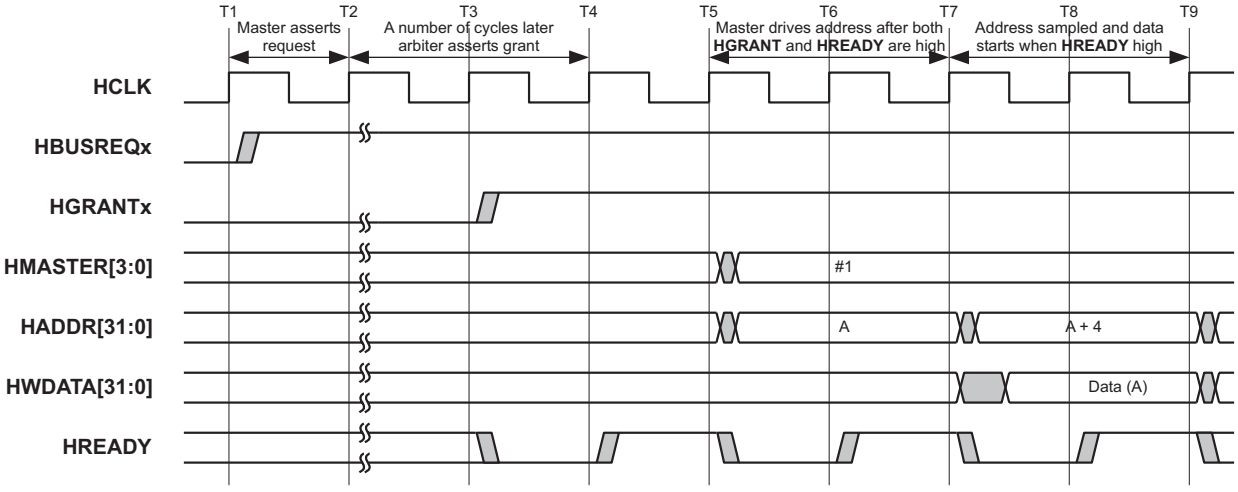



**Figure 3-16 Granting access with wait states**

The ownership of the data bus is delayed from the ownership of the address bus. Whenever a transfer completes, as indicated by **HREADY** HIGH, then the master that owns the address bus will be able to use the data bus and will continue to own the data bus until the transfer completes. Figure 3-17 shows how the ownership of the data bus is transferred when handover occurs between two bus masters.

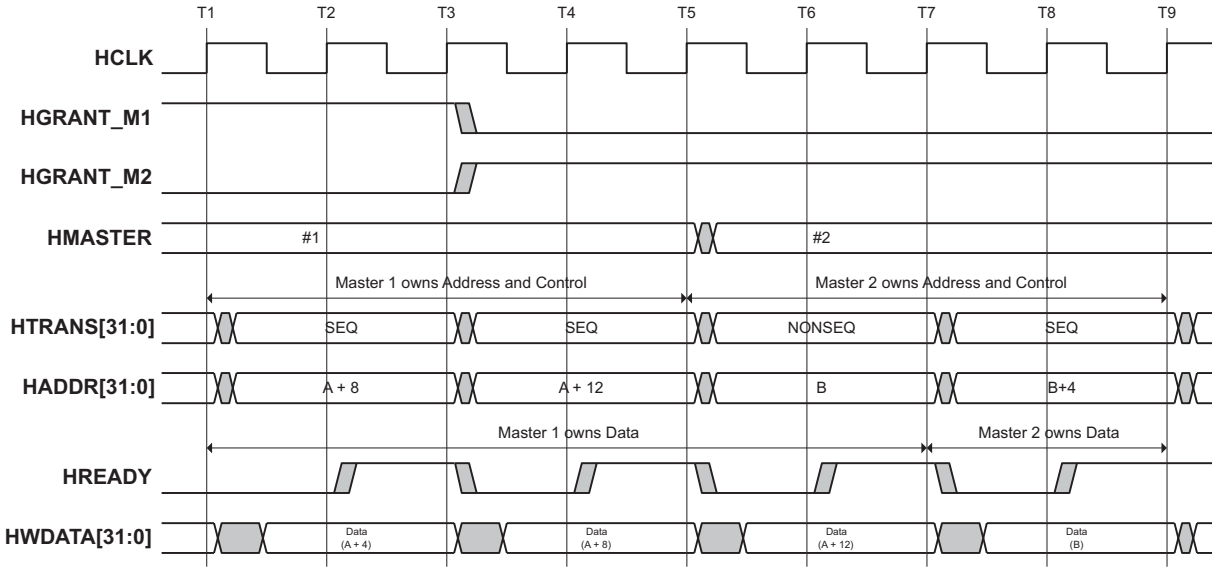



**Figure 3-17 Data bus ownership**

Figure 3-18 shows an example of how the arbiter can hand over the bus at the end of a burst of transfers.

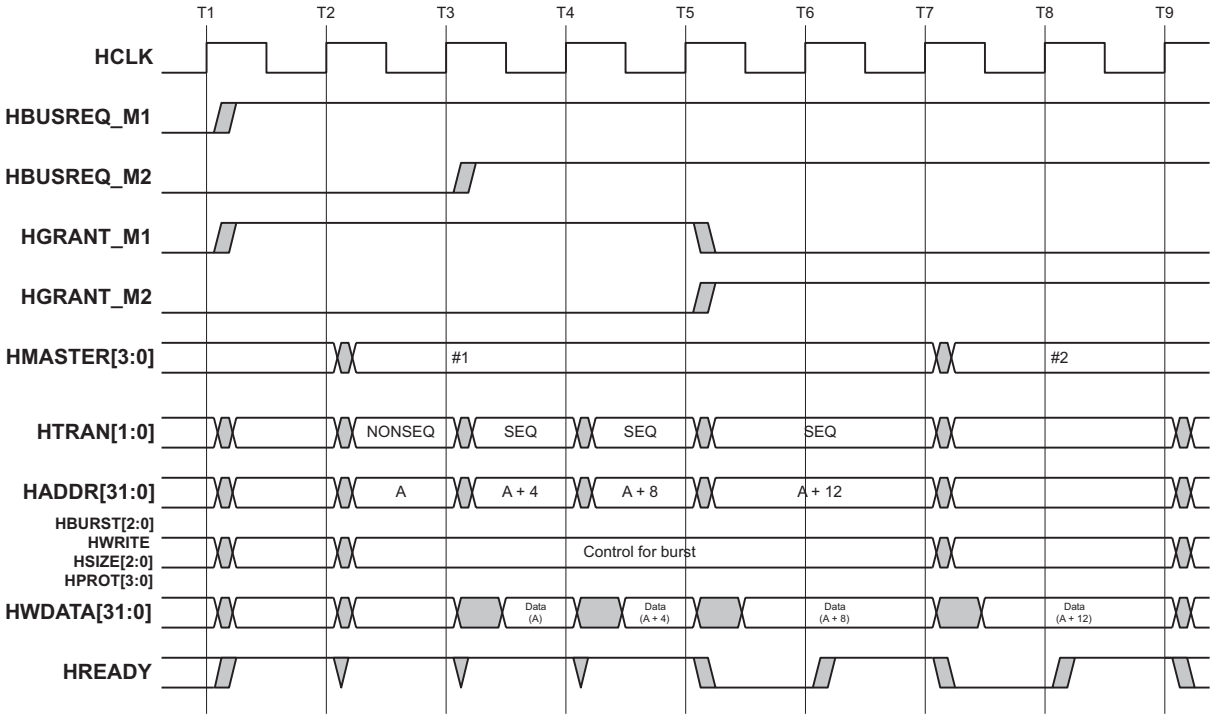



**Figure 3-18 Handover after burst**

The arbiter changes the **HGRANTx** signals when the penultimate (one before last) address has been sampled. The new **HGRANTx** information will then be sampled at the same point as the last address of the burst is sampled.

Figure 3-19 shows how **HGRANTx** and **HMASTER** signals are used in a system.

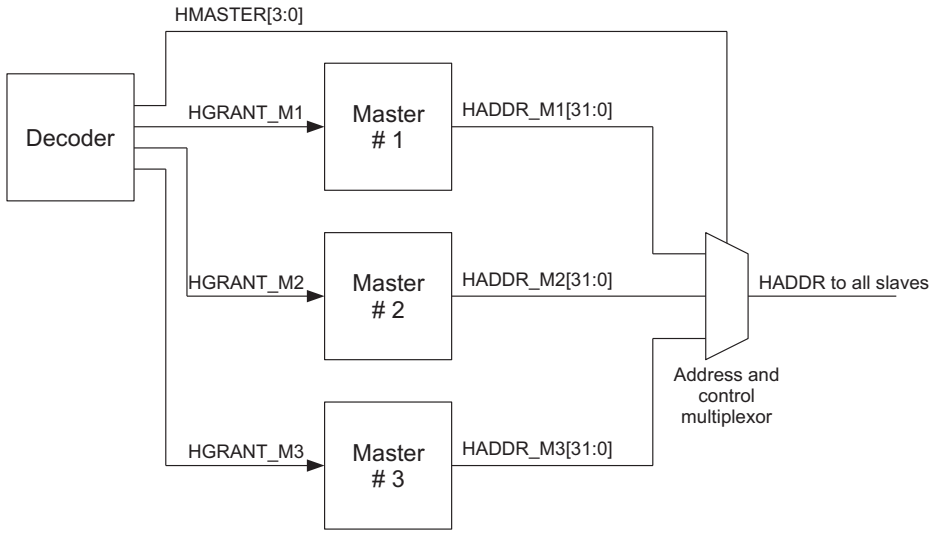



**Figure 3-19 Bus master grant signals**

**Note**

Because a central multiplexor is used, each master can drive out the address of the transfer it wishes to perform immediately and it does not need to wait until it is granted the bus. The **HGRANTx** signal is only used by the master to determine when it owns the bus and hence when it should consider that the address has been sampled by the appropriate slave.

A delayed version of the **HMASTER** bus is used to control the write data multiplexor.

### 3.11.4 Early burst termination

Normally the arbiter will not hand over the bus to a new master until the end of a burst of transfers. However, if the arbiter determines that the burst must be terminated early in order to prevent excessive access time to the bus then it may transfer the grant to another bus master before a burst has completed.

If a master loses ownership of the bus in the middle of a burst it must re-arbitrate for the bus in order to complete the burst. The master must ensure that the **HBURST** and **HTRANS** signals are adapted to reflect the fact that it no longer has to perform a complete 4, 8 or 16-beat burst.

For example, if a master is only able to complete 3 transfers of an 8-beat burst, then when it regains the bus it must use a legal burst encoding to complete the remaining 5 transfers. Any legal combination can be used, so either a 5-beat undefined length burst or a 4-beat fixed length burst followed by a single-beat undefined length burst would be acceptable.

### 3.11.5 Locked transfers

The arbiter must observe the **HLOCKx** signal from each master to determine when the master wishes to perform a locked sequence of transfers. The arbiter is then responsible for ensuring that no other bus masters are granted the bus until the locked sequence has completed.

After a sequence of locked transfers the arbiter will always keep the bus master granted for an additional transfer to ensure that the last transfer in the locked sequence has completed successfully and has not received either a SPLIT or RETRY response. Therefore it is recommended, but not mandatory, that the master inserts an IDLE transfer after any locked sequence to provide an opportunity for the arbitration to change before commencing another burst of transfers.

The arbiter is also responsible for asserting the **HMASTLOCK** signal, which has the same timing as the address and control signals. This signal indicates to any slave that the current transfer is locked and therefore must be processed before any other masters are granted the bus.

### 3.11.6 Default bus master

Every system must include a default bus master which is granted the bus if all other masters are unable to use the bus. When granted, the default bus master must only perform IDLE transfers.

If no masters are requesting the bus then the arbiter may either grant the default master or alternatively it may grant the master that would benefit the most from having low access latency to the bus.

Granting the default master access to the bus also provides a useful mechanism for ensuring that no new transfers are started on the bus and is a useful step to perform prior to entering a low-power mode of operation.

The default master must be granted if all other masters are waiting for SPLIT transfers to complete.

## 3.12 Split transfers

SPLIT transfers improve the overall utilization of the bus by separating (or splitting) the operation of the master providing the address to a slave from the operation of the slave responding with the appropriate data.

When a transfer occurs the slave can decide to issue a SPLIT response if it believes the transfer will take a large number of cycles to perform. This signals to the arbiter that the master which is attempting the transfer should not be granted access to the bus until the slave indicates it is ready to complete the transfer. Therefore the arbiter is responsible for observing the response signals and internally masking any requests from masters which have been SPLIT.

During the address phase of a transfer the arbiter generates a tag, or bus master number, on **HMASTER[3:0]** which identifies the master that is performing the transfer. Any slave issuing a SPLIT response must be capable of indicating that it can complete the transfer, and it does this by making a note of the master number on the **HMASTER[3:0]** signals.

Later, when the slave can complete the transfer, it asserts the appropriate bit, according to the master number, on the **HSPLITx[15:0]** signals from the slave to the arbiter. The arbiter then uses this information to unmask the request signal from the master and in due course the master will be granted access to the bus to retry the transfer. The arbiter samples the **HSPLITx** bus every cycle and therefore the slave only needs to assert the appropriate bit for a single cycle in order for the arbiter to recognize it.

In a system with multiple SPLIT-capable slaves the **HSPLITx** buses from each slave can be ORed together to provide a single resultant **HSPLIT** bus to the arbiter.

In the majority of systems the maximum capacity of 16 bus masters will not be used and therefore the arbiter only requires an **HSPLIT** bus which has the same number of bits as there are bus masters. However, it is recommended that all SPLIT-capable slaves are designed to support up to 16 masters.

### 3.12.1 Split transfer sequence

The basic stages of a SPLIT transaction are:

1. The master starts the transfer in an identical way to any other transfer and issues address and control information
2. If the slave is able to provide data immediately it may do so. If the slave decides that it may take a number of cycles to obtain the data it gives a SPLIT transfer response.

   During every transfer the arbiter broadcasts a number, or tag, showing which master is using the bus. The slave must record this number, to use it to restart the transfer at a later time.
3. The arbiter grants other masters use of the bus and the action of the SPLIT response allows bus master handover to occur. If all other masters have also received a SPLIT response then the default master is granted.
4. When the slave is ready to complete the transfer it asserts the appropriate bit of the **HSPLITx** bus to the arbiter to indicate which master should be regranted access to the bus.
5. The arbiter observes the **HSPLITx** signals on every cycle, and when any bit of **HSPLITx** is asserted the arbiter restores the priority of the appropriate master.
6. Eventually the arbiter will grant the master so it can re-attempt the transfer. This may not occur immediately if a higher priority master is using the bus.
7. When the transfer eventually takes place the slave finishes with an OKAY transfer response.

### 3.12.2 Multiple split transfers

The bus protocol only allows a single outstanding transaction per bus master. If any master module is able to deal with more than one outstanding transaction it requires an additional set of request and grant signals for each outstanding transaction that it can handle. At the protocol level a single module may appear as a number of different bus masters, each of which can only have one outstanding transaction.

It is, however, possible that a SPLIT-capable slave could receive more transfer requests than it is able to process concurrently. If this happens then it is acceptable for the slave to issue a SPLIT response without recording the appropriate address and control information for the transfer and it is only necessary for the slave to record the bus master number. The slave can then indicate that it can process another transfer by asserting the appropriate bits on the **HSPLITx** bus for all masters that the slave has previously SPLIT, but that the slave has not recorded the address and control information.

The arbiter is then able to regrant the masters access to the bus and they will retry the transfer, giving the address and control information required by the slave. This means that a master may be granted the bus a number of times before it is finally allowed to complete the transfer it requires.

### 3.12.3 Preventing deadlock

Both the SPLIT and RETRY transfer responses must be used with care to prevent bus deadlock. A single transfer can never lock the AHB as every slave must be designed to finish a transfer within a predetermined number of cycles. However, it is possible for deadlock to occur if a number of different masters attempt to access a slave which issues SPLIT or RETRY responses in a manner which the slave is unable to deal with.

#### Split transfers

For slaves that can issue a SPLIT transfer response, bus deadlock is prevented by ensuring that the slave can withstand a request from every master in the system, up to a maximum of 16. The slave does not need to store the address and control information for every transfer, it simply needs to record the fact that a transfer request has been made and a SPLIT response issued. Eventually all masters will be at a low priority and the slave can then work through the requests in an orderly manner, indicating to the arbiter which request it is servicing, thus ensuring that all requests are eventually serviced.

When a slave has a number of outstanding requests it may choose to process them in any order, although the slave must be aware that a locked transfer will have to be completed before any other transfers can continue.

It is perfectly legal for the slave to use a SPLIT response without latching the address and control information. The slave only needs to record that a transfer attempt has been made by that particular master and then at a later point the slave can obtain the address and control information by indicating that it is ready to complete the transfer. The master will be granted the bus and will rebroadcast the transfer, allowing the slave to latch the address and control information and either respond with the data immediately, or issue another SPLIT response if a number of additional cycles are required.

Ideally the slave should never have more outstanding transfers than it can support, but the mechanism to support this is required to prevent bus deadlock.

### Retry transfers

A slave which issues RETRY responses must only be accessed by one master at a time. This is not enforced by the protocol of the bus and should be ensured by the system architecture. In most cases slaves that issue RETRY responses will be peripherals which need to be accessed by just one master at a time, so this will be ensured by some higher level protocol.

Hardware protection against multiple masters accessing RETRY slaves is not a requirement of the protocol, but may be implemented as described in the following paragraph. The only bus-level requirement is that the slave must drive **HREADY** HIGH within a predetermined number of clock cycles.

If hardware protection is required then this may be implemented within the RETRY slave itself. When a slave issues a RETRY it can sample the master number. Between that point and the time when the transfer is finally completed the RETRY slave can check every transfer attempt that is made to ensure the master number is the same. If it ever detects that the master number is different then it can take an alternative course of action, such as:

- an ERROR response
- a signal to the arbiter
- a system level interrupt
- a complete system reset.

### 3.12.4 Bus handover with split transfers

The protocol requires that a master performs an IDLE transfer immediately after receiving a SPLIT or RETRY response allowing the bus to be transferred to another master. Figure 3-20 shows the sequence of events that occur for a split transfer.

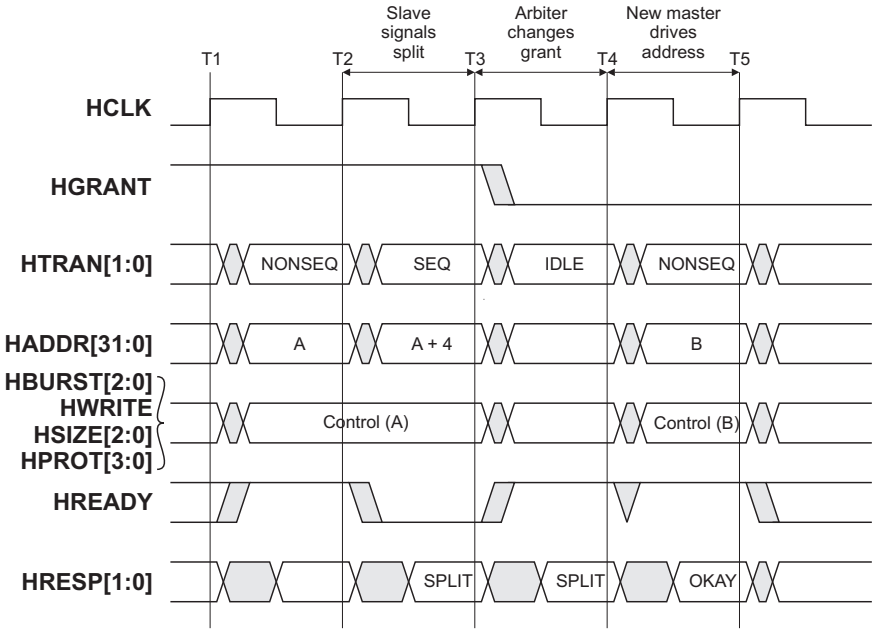



**Figure 3-20 Handover after split transfer**

The following points should be noted:

- The address for the transfer is on the bus after time T1. The slave returns the two-cycle SPLIT response after the clock edges at T2 and T3.
- At the end of the first response cycle, T3, the master can detect that the transfer will be SPLIT and so it changes the control signals for the following transfer to show an IDLE transfer.
- Also at time T3 the arbiter samples the response signals and determines that the transfer has been SPLIT. The arbiter can then adjust the arbitration priorities and the grant signals change during the following cycle, such that the new master can be granted the address bus after time T4.
- The new master is guaranteed immediate access because the IDLE transfer always completes in a single cycle.

## 3.13 Reset

The reset, **HRESETn**, is the only active LOW signal in the AMBA AHB specification and is the primary reset for all bus elements. The reset may be asserted asynchronously, but is deasserted synchronously after the rising edge of **HCLK**.

During reset all masters must ensure the address and control signals are at valid levels and that **HTRANS[1:0]** indicates IDLE.

# Apéndice E: Datos sobre CY7C68001

CY7C68001

## EZ-USB SX2™ High Speed USB Interface Device

### Features

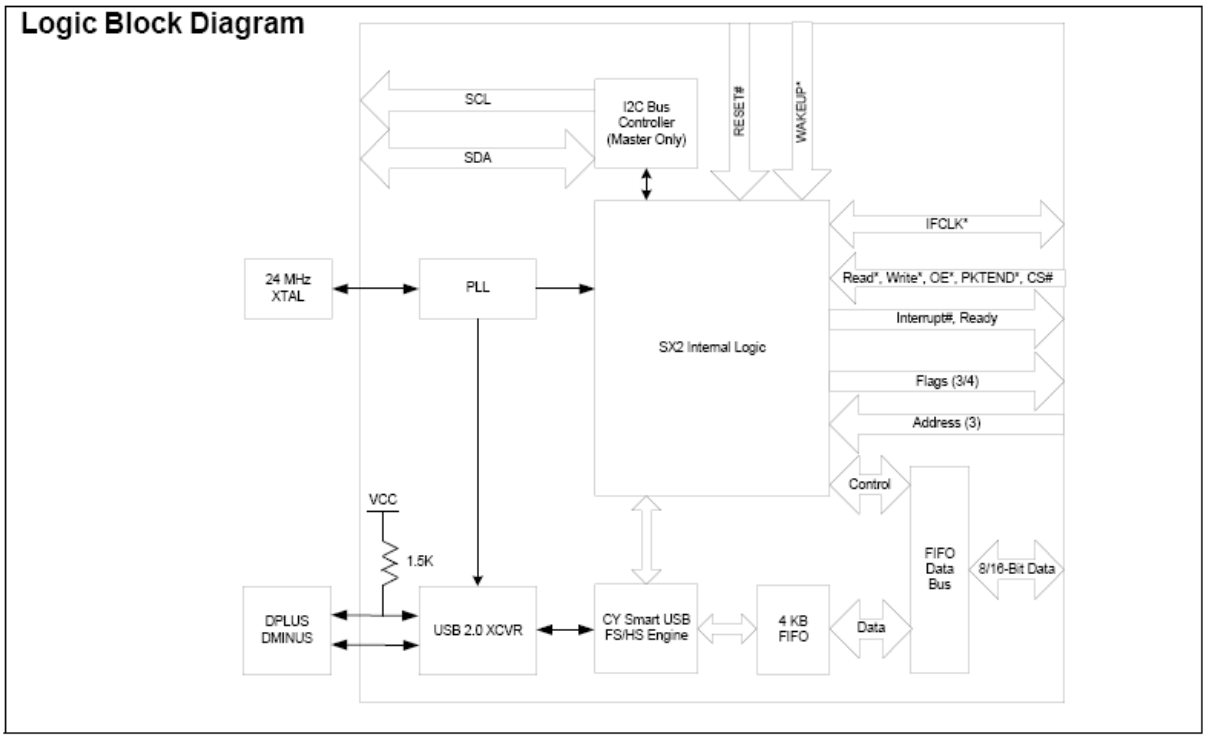
- USB 2.0-certified compliant
  - On the USB-IF Integrators List: Test ID Number 40000713
- Operates at high (480 Mbps) or full (12 Mbps) speed
- Supports Control Endpoint 0:
  - Used for handling USB device requests
- Supports four configurable endpoints that share a 4-KB FIFO space
  - Endpoints 2, 4, 6, 8 for application-specific control and data
- Standard 8- or 16-bit external master interface
  - Glueless interface to most standard microprocessors DSPs, ASICs, and FPGAs
  - Synchronous or Asynchronous interface
- Integrated phase-locked loop (PLL)
- 3.3V operation, 5V tolerant I/Os
- 56-pin SSOP and QFN package
- Complies with most device class specifications

### Applications

- DSL modems
- ATA interface
- Memory card readers
- Legacy conversion devices
- Cameras
- Scanners
- Home PNA
- Wireless LAN
- MP3 players
- Networking
- Printers

The "Reference Designs" section of the Cypress web site, www.cypress.com, provides additional tools for typical USB applications. Each reference design comes complete with firmware source code and object code, schematics, and documentation.

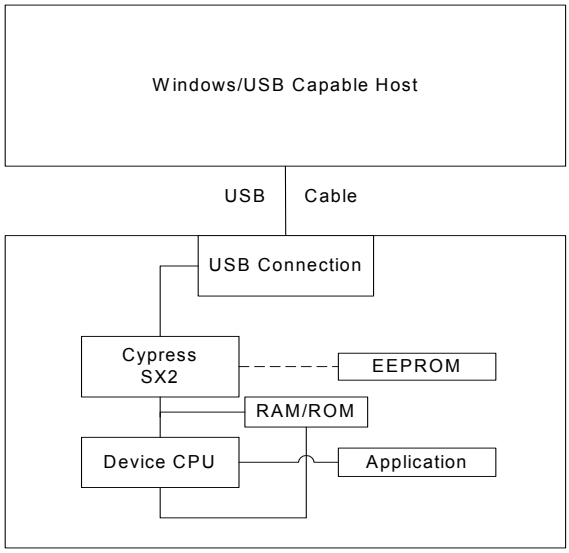
### Logic Block Diagram



## Introduction

The EZ-USB *SX2*™ USB interface device is designed to work with any external master, such as standard microprocessors, DSPs, ASICs, and FPGAs to enable USB 2.0 support for any peripheral design. SX2 has a built-in USB transceiver and Serial Interface Engine (SIE), along with a command decoder for sending and receiving USB data. The controller has four endpoints that share a 4-KB FIFO space for maximum flexibility and throughput, as well as Control Endpoint 0. SX2 has three address pins and a selectable 8- or 16- bit data bus for command and data input or output.

**Figure 1. Example USB System Diagram**




## Functional Overview

### USB Signaling Speed

*SX2* operates at two of the three rates defined in the *Universal Serial Bus Specification Revision 2.0*, dated April 27, 2000:

- Full speed, with a signaling bit rate of 12 Mbits/s
- High speed, with a signaling bit rate of 480 Mbits/s.

*SX2* does not support the low speed signaling rate of 1.5 Mbits/s.

### Buses

*SX2* features:

- A selectable 8- or 16-bit bidirectional data bus
- An address bus for selecting the FIFO or Command Interface.

### Boot Methods

During the power-up sequence, internal logic of the *SX2* checks for the presence of an $I^2C$ EEPROM.[1,2] If it finds an EEPROM, it will boot off the EEPROM. When the presence of an EEPROM is detected, the *SX2* checks the value of first byte. If the first byte is found to be a 0xC4, the *SX2* loads the next two bytes into the IFCONFIG and POLAR registers, respectively. If the fourth byte is also 0xC4, the *SX2* enumerates using the descriptor in the EEPROM, then signals to the external master when enumeration is complete via an ENUMOK interrupt (Section ). If no EEPROM is detected, the *SX2* relies on the external master for the descriptors. Once this descriptor information is received from the external master, the *SX2* will connect to the USB and enumerate.

*EEPROM Organization*

The valid sequence of bytes in the EEPROM are displayed in the following tables.

**Table 1. Descriptor Length Set to 0x06: Default Enumeration**

| Byte Index | Description |
|---|---|
| 0 | 0xC4 |
| 1 | IFCONFIG |
| 2 | POLAR |
| 3 | 0xC4 |
| 4 | Descriptor Length (LSB):0x06 |
| 5 | Descriptor Length (MSB): 0x00 |
| 6 | VID (LSB) |
| 7 | VID (MSB) |
| 8 | PID (LSB) |
| 9 | PID (MSB) |
| 10 | DID (LSB) |
| 11 | DID (MSB) |

**Table 2. Descriptor Length Not Set to 0X06**

| Byte Index | Description |
|---|---|
| 0 | 0xC4 |
| 1 | IFCONFIG |
| 2 | POLAR |
| 3 | 0xC4 |
| 4 | Descriptor Length (LSB) |
| 5 | Descriptor Length (MSB |
| 6 | Descriptor[0] |
| 7 | Descriptor[1] |
| 8 | Descriptor[2] |

**Notes**
1. Because there is no direct way to detect which EEPROM type (single or double address) is connected, *SX2* uses the EEPROM address pins A2, A1, and A0 to determine whether to send out one or two bytes of address. Single-byte address EEPROMs (24LC01, etc.) should be strapped to address 000 and double-byte EEPROMs (24LC64, etc.) should be strapped to address 001.
2. The SCL and SDA pins must be pulled up for this detection method to work properly, even if an EEPROM is not connected. Typical pull-up values are 2.2K–10K Ohms.

- **IFCONFIG**: The IFCONFIG byte contains the settings for the IFCONFIG register. The IFCONFIG register bits are defined in Section . If the external master requires an interface configuration different from the default, that interface can be specified by this byte.
- **POLAR**: The Polar byte contains the polarity of the FIFO flag pin signals. The POLAR register bits are defined in Section . If the external master requires signal polarity different from the default, the polarity can be specified by this byte.
- **Descriptor**: The Descriptor byte determines if the *SX2* loads the descriptor from the EEPROM. If this byte = 0xC4, the *SX2* will load the descriptor starting with the next byte. If this byte does not equal 0xC4, the *SX2* will wait for descriptor information from the external master.
- **Descriptor Length**: The Descriptor length is within the next two bytes and indicate the length of the descriptor contained within the EEPROM. The length is loaded least significant byte (LSB) first, then most significant byte (MSB).
- **Byte Index 6 Starts Descriptor Information**: The descriptor can be a maximum of 500 bytes.

*Default Enumeration*

An optional default descriptor can be used to simplify enumeration. Only the Vendor ID (VID), Product ID (PID), and Device ID (DID) need to be loaded by the *SX2* for it to enumerate with this default set-up. This information is either loaded from an EEPROM in the case when the presence of an EEPROM (Table 1) is detected, or the external master may simply load a VID, PID, and DID when no EEPROM is present. In this default enumeration, the *SX2* uses the in-built default descriptor (refer to Section ).

If the descriptor length loaded from the EEPROM is 6, *SX2* will load a VID, PID, and DID from the EEPROM and enumerate. The VID, PID, and DID are loaded LSB, then MSB. For example, if the VID, PID, and DID are 0x0547, 0x1002, and 0x0001, respectively, then the bytes should be stored as:

- 0x47, 0x05, 0x02, 0x10, 0x01, 0x00.

If there is no EEPROM, *SX2* will wait for the external master to provide the descriptor information. To use the default descriptor, the external master must write to the appropriate register (0x30) with descriptor length equal to 6 followed by the VID, PID, and DID. Refer to Section  for further information on how the external master may load the values.

The default descriptor enumerates four endpoints as listed in the following page:

- Endpoint 2: Bulk out, 512 bytes in high speed mode, 64 bytes in full speed mode
- Endpoint 4: Bulk out, 512 bytes in high speed mode, 64 bytes in full speed mode
- Endpoint 6: Bulk in, 512 bytes in high speed mode, 64 bytes in full speed mode
- Endpoint 8: Bulk in, 512 bytes in high speed mode, 64 bytes in full speed mode.

The entire default descriptor is listed in Section  of this data sheet.

## Interrupt System

*Architecture*

The *SX2* provides an output signal that indicates to the external master that the *SX2* has an interrupt condition, or that the data from a register read request is available. The *SX2* has six interrupt sources: SETUP, EP0BUF, FLAGS, ENUMOK, BUSACTIVITY, and READY. Each interrupt can be enabled or disabled by setting or clearing the corresponding bit in the INTENABLE register.

When an interrupt occurs, the INT# pin will be asserted, and the corresponding bit will be set in the Interrupt Status Byte. The external master reads the Interrupt Status Byte by strobing SLRD/SLOE. This presents the Interrupt Status Byte on the lower portion of the data bus (FD[7:0]). Reading the Interrupt Status Byte automatically clears the interrupt. Only one interrupt request will occur at a time; the *SX2* buffers multiple pending interrupts.

If the external master has initiated a register read request, the *SX2* will buffer interrupts until the external master has read the data. This insures that after a read sequence has begun, the next interrupt that is received from the *SX2* will indicate that the corresponding data is available. Following is a description of this INTENABLE register.

*INTENABLE Register Bit Definition*

Bit 7: SETUP

If this interrupt is enabled, and the *SX2* receives a set-up packet from the USB host, the *SX2* asserts the INT# pin and sets bit 7 in the Interrupt Status Byte. This interrupt only occurs if the set-up request is not one that the *SX2* automatically handles. For complete details on how to handle the SETUP interrupt, refer to Section  of this data sheet.

Bit 6: EP0BUF

If this interrupt is enabled, and the Endpoint 0 buffer becomes available to the external master for read or write operations, the *SX2* asserts the INT# pin and sets bit 6 in the Interrupt Status Byte. This interrupt is used for handling the data phase of a set-up request. For complete details on how to handle the EP0BUF interrupt, refer to Section  of this data sheet.

Bit 5: FLAGS

If this interrupt is enabled, and any OUT endpoint FIFO's state changes from empty to not-empty and from not-empty to empty, the *SX2* asserts the INT# pin and sets bit 5 in the Interrupt Status Byte. This is an alternate way to monitor the status of OUT endpoint FIFOs instead of using the FLAGA-FLAGD pins, and can be used to indicate when an OUT packet has been received from the host.

Bit 2: ENUMOK

If this interrupt is enabled and the *SX2* receives a SET_CONFIGURATION request from the USB host, the *SX2* asserts the INT# pin and sets bit 2 in the Interrupt Status Byte. This event signals the completion of the *SX2* enumeration process.

Bit 1: BUSACTIVITY

If this interrupt is enabled, and the *SX2* detects either an absence or resumption of activity on the USB bus, the *SX2* asserts the INT# pin and sets bit 1 in the Interrupt Status Byte. This usually

indicates that the USB host is either suspending or resuming or that a self powered device has been plugged in or unplugged. If the *SX2* is bus-powered, the external master must put the *SX2* into a low-power mode after detecting a USB suspend condition to be USB-compliant.

Bit 0: READY

If this interrupt is enabled, bit 0 in the Interrupt Status Byte is set when the *SX2* has powered up and performed a self-test. The external master should always wait for this interrupt before trying to read or write to the *SX2*, unless an external EEPROM with a valid descriptor is present. If an external EEPROM with a valid descriptor is present, the ENUMOK interrupt will occur instead of the READY interrupt after power up. A READY interrupt will also occur if the *SX2* is awakened from a low-power mode via the WAKEUP pin. This READY interrupt indicates that the *SX2* is ready for commands or data.

*Qualify with READY Pin on Register Reads*

Although it is true that all interrupts will be buffered once a command read request has been initiated, in very rare conditions, there might be a situation when there is a pending interrupt already, when a read request is initiated by the external master. In this case it is the interrupt status byte that will be output when the external master asserts the SLRD. So, a condition exists where the Interrupt Status Data Byte can be mistaken for the result of a command register read request. In order to get around this possible race condition, the first thing that the external master must do on getting an interrupt from the *SX2* is check the status of the READY pin. If the READY is low at the time the INT# was asserted, the data that will be output when the external master strobes the SLRD is the interrupt status byte (not the actual data requested). If the READY pin is high at the time when the interrupt is asserted, the data output on strobing the SLRD is the actual data byte requested by the external master. So it is important that the state of the READY pin be checked at the time the INT# is asserted to ascertain the cause of the interrupt.

## Resets and Wakeup

*Reset*

An input pin (RESET#) resets the chip. The internal PLL stabilizes after $V_{CC}$ has reached 3.3V. Typically, an external RC network (R = 100 KOhms, C = 0.1 μF) is used to provide the RESET# signal. The Clock must be in a stable state for at least 200 μs before the RESET is released.

*USB Reset*

When the *SX2* detects a USB Reset condition on the USB bus, *SX2* handles it like any other enumeration sequence. This means that *SX2* will enumerate again and assert the ENUMOK interrupt to let the external master know that it has enumerated. The external master will then be responsible for configuring the *SX2* for the application. The external master should also check whether *SX2* enumerated at High or Full speed in order to adjust the EPxPKTLENH/L register values accordingly. The last initialization task is for the external master to flush all of the *SX2* FIFOs.

*Wakeup*

The *SX2* exits its low-power state when one of the following events occur:

- USB bus signals a resume. The *SX2* will assert a BUSACTIVITY interrupt.
- The external master asserts the WAKEUP pin. The *SX2* will assert a READY interrupt[3].

## Endpoint RAM

*Size*

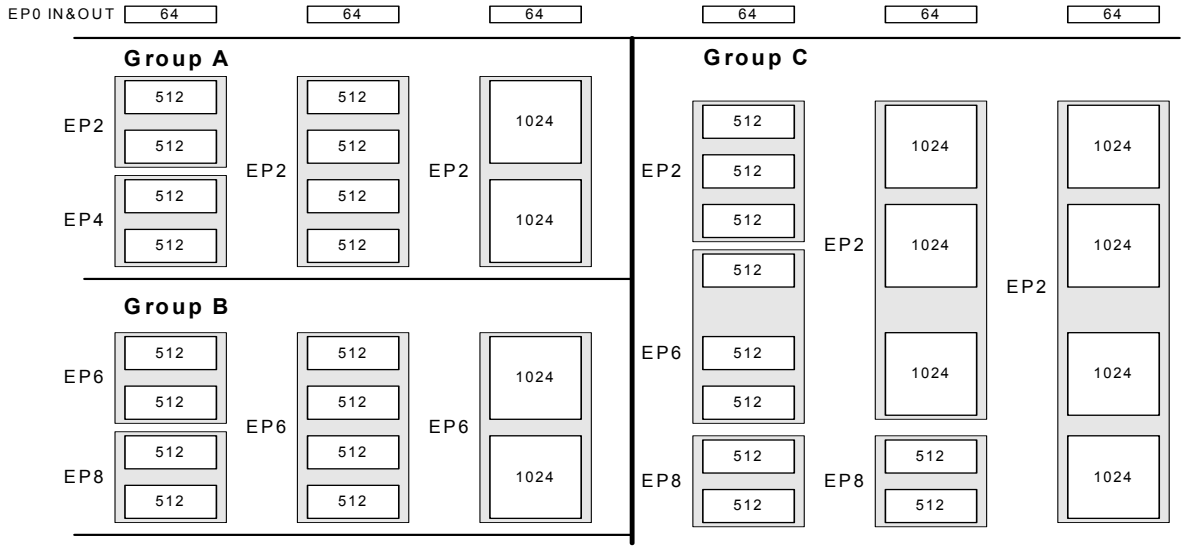
- Control endpoint: 64 Bytes: 1 × 64 bytes (Endpoint 0).
- FIFO Endpoints: 4096 Bytes: 8 × 512 bytes (Endpoint 2, 4, 6, 8).

*Organization*

- EP0–Bidirectional Endpoint 0, 64-byte buffer.
- EP2, 4, 6, 8–Eight 512-byte buffers, bulk, interrupt, or isochronous. EP2 and EP6 can be either double-, triple-, or quad-buffered. EP4 and EP8 can only be double-buffered. For high speed endpoint configuration options, see Figure 3 on page 11.

**Note**
3. If the descriptor loaded is set for remote wakeup enabled and the host does a set feature remote wakeup enabled, then the SX2 logic will perform RESUME signalling after a WAKEUP interrupt.

**Figure 2. Endpoint Configurations (High Speed Mode)**




Endpoint 0 is the same for every configuration as it serves as the CONTROL endpoint. For Endpoints 2, 4, 6, and 8, refer to Figure 3 on page 11. Endpoints 2, 4, 6, and 8 may be configured by choosing either:

- One configuration from Group A and one from Group B
- One configuration from Group C.

Some example endpoint configurations are as follows.

- EP2: 1024 bytes double-buffered, EP6: 512 bytes quad-buffered.
- EP2: 512 bytes double-buffered, EP4: 512 bytes double-buffered, EP6: 512 bytes double-buffered, EP8: 512 bytes double buffered.
- EP2: 1024 bytes quad-buffered.

*Default Endpoint Memory Configuration*

At power-on-reset, the endpoint memories are configured as follows:

- EP2: Bulk OUT, 512 bytes/packet, 2x buffered.
- EP4: Bulk OUT, 512 bytes/packet, 2x buffered.
- EP6: Bulk IN, 512 bytes/packet, 2x buffered.
- EP8: Bulk IN, 512 bytes/packet, 2x buffered.

## External Interface

The *SX2* presents two interfaces to the external master.

1. A FIFO interface through which EP2, 4, 6, and 8 data flows.
2. A command interface, which is used to set up the *SX2*, read status, load descriptors, and access Endpoint 0.

*Architecture*

The *SX2* slave FIFO architecture has eight 512-byte blocks in the endpoint RAM that directly serve as FIFO memories and are controlled by FIFO control signals (IFCLK, CS#, SLRD, SLWR, SLOE, PKTEND, and FIFOADR[2:0]).

The *SX2* command interface is used to set up the *SX2*, read status, load descriptors, and access Endpoint 0. The command interface has its own READY signal for gating writes, and an INT# signal to indicate that the *SX2* has data to be read, or that an interrupt event has occurred. The command interface uses the same control signals (IFCLK, CS#, SLRD, SLWR, SLOE, and FIFOADR[2:0]) as the FIFO interface, except for PKTEND.

*Control Signals*

#### 0.0.0.1 FIFOADDR Lines

The *SX2* has three address pins that are used to select either the FIFOs or the command interface. The addresses correspond to the following table.

**Table 3. FIFO Address Lines Setting**

| Address/Selection | FIFOADR2 | FIFOADR1 | FIFOADR0 |
|---|---|---|---|
| FIFO2 | 0 | 0 | 0 |
| FIFO4 | 0 | 0 | 1 |
| FIFO6 | 0 | 1 | 0 |
| FIFO8 | 0 | 1 | 1 |
| COMMAND | 1 | 0 | 0 |
| RESERVED | 1 | 0 | 1 |
| RESERVED | 1 | 1 | 0 |
| RESERVED | 1 | 1 | 1 |

The *SX2* accepts either an internally derived clock (30 or 48 MHz) or externally supplied clock (IFCLK, 5–50 MHz), and SLRD, SLWR, SLOE, PKTEND, CS#, FIFOADR[2:0] signals from an external master. The interface can be selected for 8- or 16- bit operation by an internal configuration bit, and an Output Enable signal SLOE enables the data bus driver of the selected width. The external master must ensure that the output enable signal is inactive when writing data to the *SX2*. The interface can operate either asynchronously where the SLRD and SLWR signals act directly as strobes, or synchronously where the SLRD and SLWR act as clock qualifiers. The optional CS# signal will tristate the data bus and ignore SLRD, SLWR, PKTEND.

The external master reads from OUT endpoints and writes to IN endpoints, and reads from or writes to the command interface.

#### 0.0.0.2 Read: SLOE and SLRD

In synchronous mode, the FIFO pointer is incremented on each rising edge of IFCLK while SLRD is asserted. In asynchronous mode, the FIFO pointer is incremented on each asserted-to-deasserted transition of SLRD.

SLOE is a data bus driver enable. When SLOE is asserted, the data bus is driven by the *SX2*.

#### 0.0.0.3 Write: SLWR

In synchronous mode, data on the FD bus is written to the FIFO (and the FIFO pointer is incremented) on each rising edge of IFCLK while SLWR is asserted. In asynchronous mode, data on the FD bus is written to the FIFO (and the FIFO pointer is incremented) on each asserted-to-deasserted transition of SLWR.

#### 0.0.0.4 PKTEND

PKTEND commits the current buffer to USB. To send a short IN packet (one which has not been filled to max packet size determined by the value of PL[X:0] in EPxPKTLENH/L), the external master strobes the PKTEND pin.

All these interface signals have a default polarity of low. In order to change the polarity of PKTEND pin, the master may write to the POLAR register anytime. In order to switch the polarity of the SLWR/SLRD/SLOE, the master must set the appropriate bits 2, 3 and 4 respectively in the FIFOPINPOLAR register located at XDATA space 0xE609. Please note that the *SX2* powers up with the polarities set to low. POLAR Register 0x04 on page 18 provides further information on how to access this register located at XDATA space.

*IFCLK*

The IFCLK pin can be configured to be either an input (default) or an output interface clock. Bits IFCONFIG[7:4] define the behavior of the interface clock. To use the *SX2*'s internally-derived 30- or 48-MHz clock, set IFCONFIG.7 to 1 and set IFCONFIG.6 to 0 (30 MHz) or to 1 (48 MHz). To use an externally supplied clock, set IFCONFIG.7=0 and drive the IFCLK pin (5 MHz – 50 MHz). The input or output IFCLK signal can be inverted by setting IFCONFIG.4=1.

*FIFO Access*

An external master can access the slave FIFOs either asynchronously or synchronously:

- Asynchronous–SLRD, SLWR, and PKTEND pins are strobes.
- Synchronous–SLRD, SLWR, and PKTEND pins are enables for the IFCLK clock pin.

An external master accesses the FIFOs through the data bus, FD [15:0]. This bus can be either 8- or 16-bits wide; the width is selected via the WORDWIDE bit in the EPxPKTLENH/L registers. The data bus is bidirectional, with its output drivers controlled by the SLOE pin. The FIFOADR[2:0] pins select which of the four FIFOs is connected to the FD [15:0] bus, or if the command interface is selected.

*FIFO Flag Pins Configuration*

The FIFO flags are FLAGA, FLAGB, FLAGC, and FLAGD. These FLAGx pins report the status of the FIFO selected by the FIFOADR[2:0] pins. At reset, these pins are configured to report the status of the following:

- FLAGA reports the status of the programmable flag.
- FLAGB reports the status of the full flag.
- FLAGC reports the status of the empty flag.
- FLAGD defaults to the CS# function.

The FIFO flags can either be indexed or fixed. Fixed flags report the status of a particular FIFO regardless of the value on the FIFOADR [2:0] pins. Indexed flags report the status of the FIFO selected by the FIFOADR [2:0]pins.[4]

*Default FIFO Programmable Flag Set-up*

By default, FLAGA is the Programmable Flag (PF) for the endpoint being pointed to by the FIFOADR[2:0] pins. For EP2 and EP4, the default endpoint configuration is BULK, OUT, 512, 2x, and the PF pin asserts when the entire FIFO has greater than/equal to 512 bytes. For EP6 and EP8, the default endpoint configuration is BULK, IN, 512, 2x, and the PF pin asserts when the entire FIFO has less than/equal to 512 bytes. In other words, EP6/8 report a half-empty state, and EP2/4 report a half-full state.

*FIFO Programmable Flag (PF) Set-up*

Each FIFO's programmable-level flag (PF) asserts when the FIFO reaches a user-defined fullness threshold. That threshold is configured as follows:

1. For OUT packets: The threshold is stored in PFC12:0. The PF is asserted when the number of bytes *in the entire FIFO* is less than/equal to (DECIS = 0) or greater than/equal to (DECIS = 1) the threshold.
2. For IN packets, with PKTSTAT = 1: The threshold is stored in PFC9:0. The PF is asserted when the number of bytes written into *the current packet in the FIFO* is less than/equal to (DECIS = 0) or greater than/equal to (DECIS = 1) the threshold.
3. For IN packets, with PKTSTAT = 0: The threshold is stored in two parts: PKTS2:0 holds the number of committed packets, and PFC9:0 holds the number of bytes in the current packet. The PF is asserted when the FIFO is at or less full than (DECIS = 0), or at or more full than (DECIS = 1), the threshold.

**Note**
4. In indexed mode, the value of the FLAGx pins is indeterminate except when addressing a FIFO (FIFOADR[2:0]={000,001,010,011}).

*Command Protocol*

An address of [1 0 0] on FIFOADR [2:0] will select the command interface. The command interface is used to write to and read from the *SX2* registers and the Endpoint 0 buffer, as well as the descriptor RAM. Command read and write transactions occur over FD[7:0] only. Each byte written to the *SX2* is either an address or a data byte, as determined by bit7. If bit7 = 1, then the byte is considered an address byte. If bit7 = 0, then the byte is considered a data byte. If bit7 = 1, then bit6 determines whether the address byte is a read request or a write request. If bit6 = 1, then the byte is considered a read request. If bit6 = 0 then the byte is considered a write request. Bits [5:0] hold the register address of the request. The format of the command address byte is shown in Table 4.

**Table 4. Command Address Byte**

| Address/ Data# | Read/ Write# | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |

Each Write request is followed by two or more data bytes. If another address byte is received before both data bytes are received, the *SX2* ignores the first address and any incomplete data transfers. The format for the data bytes is shown in Table 5 and Table 6. Some registers take a series of bytes. Each byte is transferred using the same protocol.

**Table 5. Command Data Byte One**

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | D7 | D6 | D5 | D4 |

**Table 6. Command Data Byte Two**

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | D3 | D2 | D1 | D0 |

The first command data byte contains the upper nibble of data, and the second command byte contains the lower nibble of data.

#### 0.0.0.5 Write Request Example

Prior to writing to a register, two conditions must be met: FIFOADR[2:0] must hold [1 0 0], and the Ready line must be HIGH. The external master should not initiate a command if the READY pin is not in a HIgh state.

**Example**: To write the byte <10110000> into the IFCONFIG register (0x01), first send a command address byte as follows.

**Table 7. Command Address Write Byte**

| Address/ Data# | Read/ Write# | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

- The first bit signifies an address transfer.
- The second bit signifies that this is a write command.
- The next six bits represent the register address (000001 binary = 0x01 hex).

Once the byte has been received the *SX2* pulls the READY pin low to inform the external master not to send any more information. When the *SX2* is ready to receive the next byte, the *SX2* pulls the READY pin high again. This next byte, the upper nibble of the data byte, is written to the *SX2* as follows.

**Table 8. Command Data Write Byte One**

| Address/ Data# | Don't Care | Don't Care | Don't Care | D7 | D6 | D5 | D4 |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | 1 | 0 | 1 | 1 |

- The first bit signifies that this is a data transfer.
- The next three are don't care bits.
- The next four bits hold the upper nibble of the transferred byte.

Once the byte has been received the *SX2* pulls the READY pin low to inform the external master not to send any more information. When the *SX2* is ready to receive the next byte, the *SX2* pulls the READY pin high again. This next byte, the lower nibble of the data byte is written to the *SX2*.

**Table 9. Command Data Write Byte Two**

| Address/ Data# | Don't Care | Don't Care | Don't Care | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | 0 | 0 | 0 | 0 |

At this point the entire byte <10110000> has been transferred to register 0x01 and the write sequence is complete.

#### 0.0.0.6 Read Request Example

The Read cycle is simpler than the write cycle. The Read cycle consists of a read request from the external master to the *SX2*. For example, to read the contents of register 0x01, a command address byte is written to the *SX2* as follows.

**Table 10. Command Address Read Byte**

| Address/ Data# | Read/ Write# | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

When the data is ready to be read, the *SX2* asserts the INT# pin to tell the external master that the data it requested is waiting on FD[7:0].[5]

**Note**

5. An important note: Once the *SX2* receives a Read request, the *SX2* allocates the interrupt line solely for the read request. If one of the six interrupt sources described in Section  is asserted, the *SX2* will buffer that interrupt until the read request completes.

## Enumeration

The *SX2* has two modes of enumeration. The first mode is automatic through EEPROM boot load, as described in Boot Methods on page 2. The second method is a manual load of the descriptor or VID, PID, and DID as described below.

### Standard Enumeration

The *SX2* has 500 bytes of descriptor RAM into which the external master may write its descriptor. The descriptor RAM is accessed through register 0x30. To load a descriptor, the external master does the following:

- Initiate a Write Request to register 0x30.
- Write two bytes (four command data transfers) that define the length of the entire descriptor about to be transferred. The LSB is written first, followed by the MSB.[6]
- Write the descriptor, one byte at a time until complete.[6] Note: the register address is only written once.

After the entire descriptor has been transferred, the *SX2* will float the pull-up resistor connected to D+, and parse through the descriptor to locate the individual descriptors. After the *SX2* has parsed the entire descriptor, the *SX2* will connect the pull-up resistor and enumerate automatically. When enumeration is complete, the *SX2* will notify the external master with an ENUMOK interrupt.

The format and order of the descriptor should be as follows (see Default Descriptor on page 35 for an example):

- Device.
- Device qualifier.
- High speed configuration, high speed interface, high speed endpoints.
- Full speed configuration, full speed interface, full speed endpoints.
- String.

The SX2 can be set to run in full speed only mode. To force full speed only enumeration write a 0x02 to the unindexed register CT1 at address 0xE6FB before downloading the descriptors. This disables the chirp mechanism forcing the SX2 to come up in full speed only mode after the descriptors are loaded. The CT1 register can be accessed using the unindexed register mechanism. Examples of writing to unindexed registers are shown in Resetting Data Toggle on page 9. Each write consists of a command write with the target register followed by the write of the upper nibble of the value followed by the write of the lower nibble of the value.

### Default Enumeration

The external master may simply load a VID, PID, and DID and use the default descriptor built into the *SX2*. To use the default descriptor, the descriptor length described above must equal 6. After the external master has written the length, the VID, PID, and DID must be written LSB, then MSB. For example, if the VID, PID, and DID are 0x04B4, 0x1002, and 0x0001 respectively, then the external master does the following:

- Initiates a Write Request to register 0x30.
- Writes two bytes (four command data transfers) that define the length of the entire descriptor about to be transferred. In this case, the length is always six.
- Writes the VID, PID, and DID bytes: 0xB4, 0x04, 0x02, 0x10, 0x01, 0x00 (in nibble format per the command protocol).

The default descriptor is listed in Default Descriptor on page 35. The default descriptor can be used as a starting point for a custom descriptor.

## Endpoint 0

The *SX2* will automatically respond to USB chapter 9 requests without any external master intervention. If the *SX2* receives a request to which it cannot respond automatically, the *SX2* will notify the external master. The external master then has the choice of responding to the request or stalling.

After the *SX2* receives a set-up packet to which it cannot respond automatically, the *SX2* will assert a SETUP interrupt. After the external master reads the Interrupt Status Byte to determine that the interrupt source was the SETUP interrupt, it can initiate a read request to the SETUP register, 0x32. When the *SX2* sees a read request for the SETUP register, it will present the first byte of set-up data to the external master. Each additional read request will present the next byte of set-up data, until all eight bytes have been read.

The external master can stall this request at this or any other time. To stall a request, the external master initiates a write request for the SETUP register, 0x32, and writes any non-zero value to the register.

If this set-up request has a data phase, the *SX2* will then interrupt the external master with an EP0BUF interrupt when the buffer becomes available. The *SX2* determines the direction of the set-up request and interrupts when either:

- IN: the Endpoint 0 buffer becomes available to write to, or
- OUT: the Endpoint 0 buffer receives a packet from the USB host.

For an IN set-up transaction, the external master can write up to 64 bytes at a time for the data phase. The steps to write a packet are as follows:

1. Wait for an EP0BUF interrupt, indicating that the buffer is available.
2. Initiate a write request for register 0x31.
3. Write one data byte.
4. Repeat steps 2 and 3 until either all the data or 64 bytes have been written, whichever is less.
5. Write the number of bytes in this packet to the byte count register, 0x33.

**Note**
6. These and all other data bytes must conform to the command protocol.

To send more than 64 bytes, the process is repeated. The *SX2* internally stores the length of the data phase that was specified in the wLength field (bytes 6,7) of the set-up packet. To send less than the requested amount of data, the external master writes a packet that is less than 64 bytes, or if a multiple of 64, the external master follows the data with a zero-length packet. When the *SX2* sees a short or zero-length packet, it will complete the set-up transfer by automatically completing the handshake phase. The *SX2* will not allow more data than the wLength field specified in the set-up packet. Note: the PKTEND pin does not apply to Endpoint 0. The only way to send a short or zero length packet is by writing to the byte count register with the appropriate value.

For an OUT set-up transaction, the external master can read each packet received from the USB host during the data phase. The steps to read a packet are as follows:

1. Wait for an EP0BUF interrupt, indicating that a packet was received from the USB host into the buffer.
2. Initiate a read request for the byte count register, 0x33. This indicates the amount of data received from the host.
3. Initiate a read request for register 0x31.
4. Read one byte.
5. Repeat steps 3 and 4 until the number of bytes specified in the byte count register has been read.

To receive more than 64 bytes, the process is repeated. The *SX2* internally stores the length of the data phase that was specified in the wLength field of the set-up packet (bytes 6,7). When the *SX2* sees that the specified number of bytes have been received, it will complete the set-up transfer by automatically completing the handshake phase. If the external master does not wish to receive the entire transfer, it can stall the transfer.

If the *SX2* receives another set-up packet before the current transfer has completed, it will interrupt the external master with another SETUP interrupt. If the *SX2* receives a set-up packet with no data phase, the external master can accept the packet and complete the handshake phase by writing zero to the byte count register.

The *SX2* automatically responds to all USB standard requests covered in chapter 9 of the USB 2.0 specification except the Set/Clear Feature Endpoint requests. When the host issues a Set Feature or a Clear feature request, the *SX2* will trigger a SETUP interrupt to the external master. The USB spec requires that the device respond to the Set endpoint feature request by doing the following:

- Set the STALL condition on that endpoint.

The USB spec requires that the device respond to the Clear endpoint feature request by doing the following:

- Reset the Data Toggle for that endpoint
- Clear the STALL condition of that endpoint.

The register that is used to reset the data toggle TOGCTL (located at XDATA location 0xE683) is not an index register that can be addressed by the command protocol presented in “Command Protocol” on page 7. The following section provides further information on this register bits and how to reset the data toggle accordingly using a different set of command protocol sequence.

### Resetting Data Toggle

**Table 11. Bit definition of the TOGCTL register**

| TOGCTL | | | | | | | | 0xE683 |
|---|---|---|---|---|---|---|---|---|
| **Bit #** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Bit Name** | Q | S | R | I/O | EP3 | EP2 | EP1 | EP0 |
| **Read/Write** | R | W | W | R/W | R/W | R/W | R/W | R/W |
| **Default** | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

**Bit 7: Q,** *Data Toggle Value*

Q=0 indicates DATA0 and Q=1 indicates DATA1, for the endpoint selected by the I/O and EP3:0 bits. Write the endpoint select bits (IO and EP3:0), before reading this value.

**Bit 6: S,** *Set Data Toggle to DATA1*

After selecting the desired endpoint by writing the endpoint select bits (IO and EP3:0), set S=1 to set the data toggle to DATA1. The endpoint selection bits should not be changed while this bit is written.

**Bit 5: R,** *Set Data Toggle to DATA0*

Set R=1 to set the data toggle to DATA0. The endpoint selection bits should not be changed while this bit is written.

**Bit 4: IO,** *Select IN or OUT Endpoint*

Set this bit to select an endpoint direction prior to setting its R or S bit. IO=0 selects an OUT endpoint, IO = 1 selects an IN endpoint.

**Bit 3-0: EP3:0,** *Select Endpoint*

Set these bits to select an endpoint prior to setting its R or S bit. Valid values are 0, 1, 2, 6, and 8.

A two-step process is employed to clear an endpoint data toggle bit to 0. First, write to the TOGCTL register with an endpoint address (EP3:EP0) plus a direction bit (IO). Keeping the endpoint and direction bits the same, write a “1” to the R (reset) bit. For example, to clear the data toggle for EP6 configured as an “IN” endpoint, write the following values sequentially to TOGCTL:

00010110b

00110110b

Following is the sequence of events that the master should perform to set this register to 0x16:

- Send Low Byte of the Register (0x83)
  - Command **address** write of address 0x3A
  - Command **data** write of upper nibble of the Low Byte of Register Address (0x08)
  - Command **data** write of lower nibble of the Low Byte of Register Address (0x03)
- Send High Byte of the Register (0xE6)
  - Command **address** write of address 0x3B
  - Command **data** write of upper nibble of the High Byte of Register Address (0x0E)
  - Command **data** write of lower nibble of the High Byte of Register Address (0x06)
- Send the actual value to write to the register Register (in this case 0x16)
  - Command **address** write of address0x3C
  - Command **data** write of upper nibble of the register value (0x01)
  - Command **data** write of lower nibble of the register value (0x06)

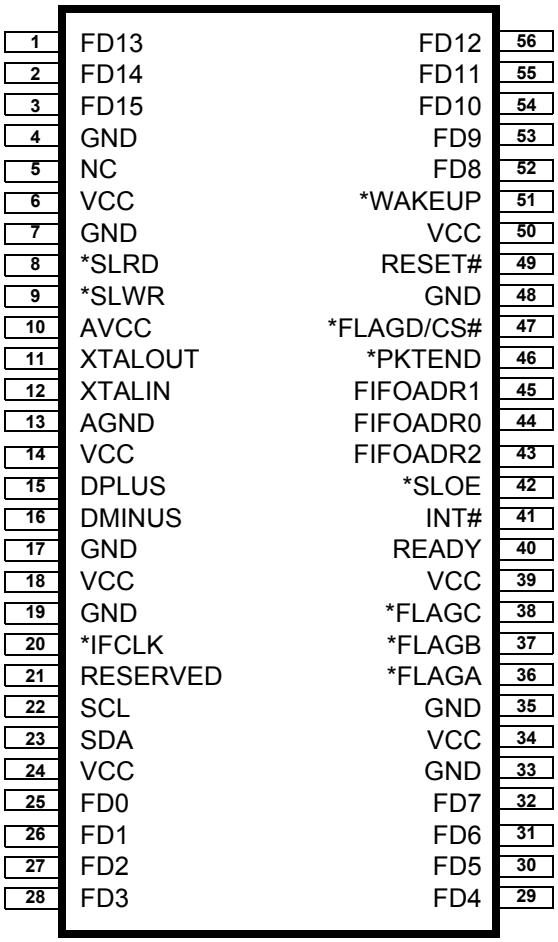
The same command sequence needs to be followed to set TOGCTL register to 0x36. The same command protocol sequence can be used to reset the data toggle for the other endpoints.

In order to read the status of this register, the external master must do the following sequence of events:

- Send Low Byte of the Register (0x83)
  - Command **address** write of 0x3A
  - Command **data** write of upper nibble of the Low Byte of Register Address (0x08)
  - Command **data** write of lower nibble of the Low Byte of Register Address (0x03)
- Send High Byte of the Register (0xE6)
  - Command **address** write of address 0x3B
  - Command **data** write of upper nibble of the High Byte of Register Address (0x0E)
  - Command **data** write of lower nibble of the High Byte of Register Address (0x06)
- Get the actual value from the TOGCTL register (0x16)
  - Command **address READ** of 0x3C

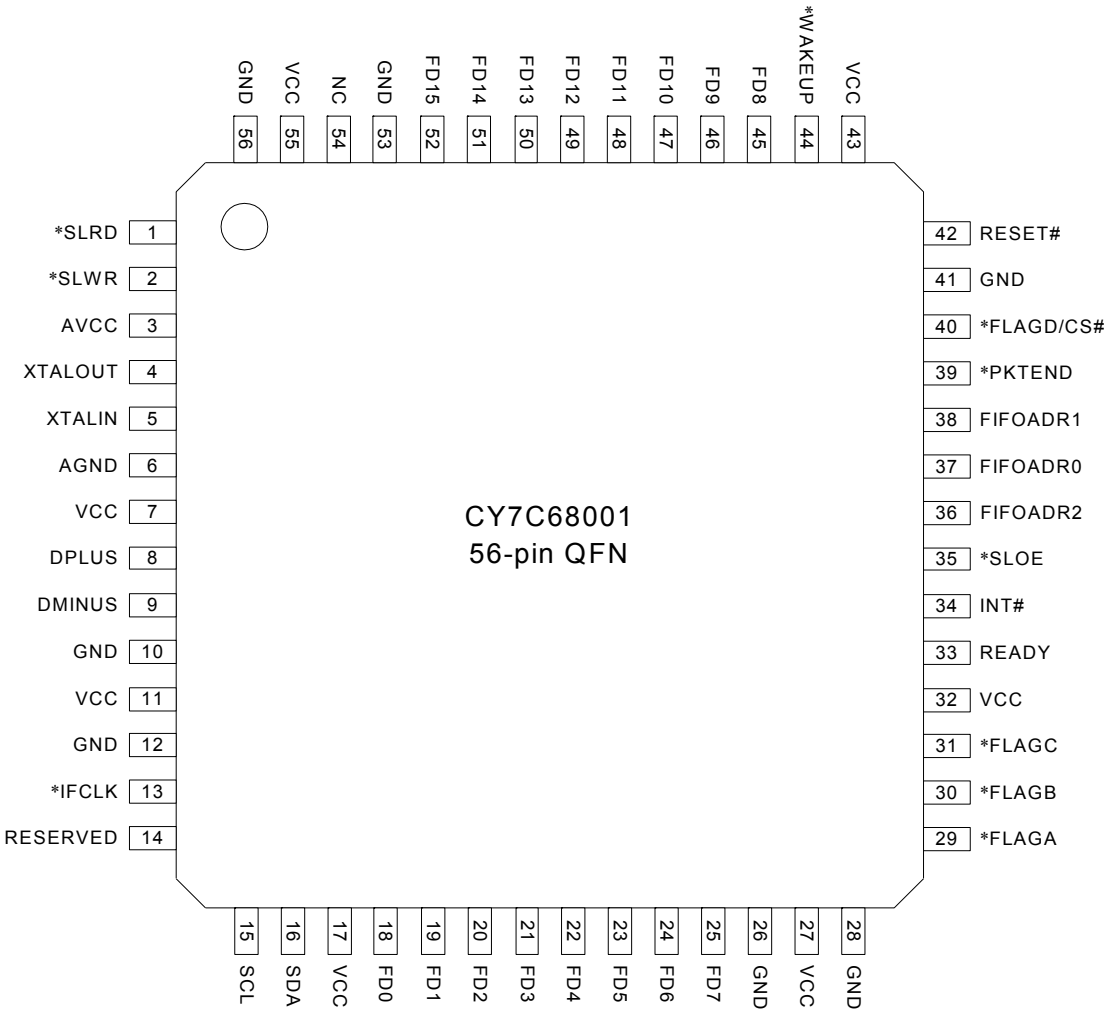
## Pin Configurations

**Figure 3. CY7C68001 56-Pin SSOP Pin Assignment**[7]




**Note**
7. A * denotes programmable polarity.

**Figure 4. CY7C68001 56-Pin QFN Assignment**[7]



## CY7C68001 Pin Definitions

**Table 12. *SX2* Pin Definitions**

| QFN Pin | SSOP Pin | Name | Type | Default | Description |
|---|---|---|---|---|---|
| 3 | 10 | AVCC | Power | N/A | **Analog $V_{CC}$**. This signal provides power to the analog section of the chip. |
| 6 | 13 | AGND | Power | N/A | **Analog Ground**. Connect to ground with as short a path as possible. |
| 9 | 16 | DMINUS | IOZ | Z | **USB D– Signal**. Connect to the USB D– signal. |
| 8 | 15 | DPLUS | IOZ | Z | **USB D+ Signal**. Connect to the USB D+ signal. |
| 42 | 49 | RESET# | Input | N/A | **Active LOW Reset**. Resets the entire chip. This pin is normally tied to $V_{CC}$ through a 100K resistor, and to GND through a 0.1-µF capacitor. |
| 5 | 12 | XTALIN | Input | N/A | **Crystal Input**. Connect this signal to a 24-MHz parallel-resonant, fundamental mode crystal and 20-pF capacitor to GND. It is also correct to drive XTALIN with an external 24-MHz square wave derived from another clock source. |
| 4 | 11 | XTALOUT | Output | N/A | **Crystal Output**. Connect this signal to a 24-MHz parallel-resonant, fundamental mode crystal and 20-pF capacitor to GND. If an external clock is used to drive XTALIN, leave this pin open. |
| 54 | 5 | NC | Output | O | **No Connect**. This pin must be left unconnected. |
| | | | | | |
| 33 | 40 | READY | Output | L | **READY** is an output-only ready that gates external command reads and writes. Active High. |
| 34 | 41 | INT# | Output | H | **INT#** is an output-only external interrupt signal. Active Low. |
| 35 | 42 | SLOE | Input | I | **SLOE** is an input-only output enable with programmable polarity (POLAR.4) for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 36 | 43 | FIFOADR2 | Input | I | **FIFOADR2** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 37 | 44 | FIFOADR0 | Input | I | **FIFOADR0** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 38 | 45 | FIFOADR1 | Input | I | **FIFOADR1** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 39 | 46 | PKTEND | Input | I | **PKTEND** is an input-only packet end with programmable polarity (POLAR.5) for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 40 | 47 | FLAGD/CS# | CS#:I FLAGD:O | I | **FLAGD** is a programmable slave-FIFO output status flag signal. CS# is a master chip select (default). |
| 18 | 25 | FD[0] | IOZ | I | **FD[0]** is the bidirectional FIFO/Command data bus. |
| 19 | 26 | FD[1] | IOZ | I | **FD[1]** is the bidirectional FIFO/Command data bus. |
| 20 | 27 | FD[2] | IOZ | I | **FD[2]** is the bidirectional FIFO/Command data bus. |
| 21 | 28 | FD[3] | IOZ | I | **FD[3]** is the bidirectional FIFO/Command data bus. |
| 22 | 29 | FD[4] | IOZ | I | **FD[4]** is the bidirectional FIFO/Command data bus. |
| 23 | 30 | FD[5] | IOZ | I | **FD[5]** is the bidirectional FIFO/Command data bus. |
| 24 | 31 | FD[6] | IOZ | I | **FD[6]** is the bidirectional FIFO/Command data bus. |
| 25 | 32 | FD[7] | IOZ | I | **FD[7]** is the bidirectional FIFO/Command data bus. |
| 45 | 52 | FD[8] | IOZ | I | **FD[8]** is the bidirectional FIFO data bus. |
| 46 | 53 | FD[9] | IOZ | I | **FD[9]** is the bidirectional FIFO data bus. |
| 47 | 54 | FD[10] | IOZ | I | **FD[10]** is the bidirectional FIFO data bus. |
| 48 | 55 | FD[11] | IOZ | I | **FD[11]** is the bidirectional FIFO data bus. |
| 49 | 56 | FD[12] | IOZ | I | **FD[12]** is the bidirectional FIFO data bus. |
| 50 | 1 | FD[13] | IOZ | I | **FD[13]** is the bidirectional FIFO data bus. |
| 51 | 2 | FD[14] | IOZ | I | **FD[14]** is the bidirectional FIFO data bus. |

**Table 12. *SX2* Pin Definitions** (continued)

| QFN Pin | SSOP Pin | Name | Type | Default | Description |
|---|---|---|---|---|---|
| 52 | 3 | FD[15] | IOZ | I | **FD[15]** is the bidirectional FIFO data bus. |
| | | | | | |
| 1 | 8 | SLRD | Input | N/A | **SLRD** is the input-only read strobe with programmable polarity (POLAR.3) for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 2 | 9 | SLWR | Input | N/A | **SLWR** is the input-only write strobe with programmable polarity (POLAR.2) for the slave FIFOs connected to FD[7:0] or FD[15:0]. |
| 29 | 36 | FLAGA | Output | H | **FLAGA** is a programmable slave-FIFO output status flag signal. Defaults to PF for the FIFO selected by the FIFOADR[2:0] pins. |
| 30 | 37 | FLAGB | Output | H | **FLAGB** is a programmable slave-FIFO output status flag signal. Defaults to FULL for the FIFO selected by the FIFOADR[2:0] pins. |
| 31 | 38 | FLAGC | Output | H | **FLAGC** is a programmable slave-FIFO output status flag signal. Defaults to EMPTY for the FIFO selected by the FIFOADR[2:0] pins. |
| 13 | 20 | IFCLK | IOZ | Z | **Interface Clock**, used for synchronously clocking data into or out of the slave FIFOs. IFCLK also serves as a timing reference for all slave FIFO control signals. When using the internal clock reference (IFCONFIG.7=1) the IFCLK pin can be configured to output 30/48 MHz by setting bits IFCONFIG.5 and IFCONFIG.6. IFCLK may be inverted by setting the bit IFCONFIG.4=1. Programmable polarity. |
| 14 | 21 | Reserved | Input | N/A | **Reserved**. Must be connected to ground. |
| 44 | 51 | WAKEUP | Input | N/A | **USB Wakeup**. If the *SX2* is in suspend, asserting this pin starts up the oscillator and interrupts the *SX2* to allow it to exit the suspend mode. During normal operation, holding WAKEUP asserted inhibits the *SX2* chip from suspending. This pin has programmable polarity (POLAR.7). |
| 15 | 22 | SCL | OD | Z | **$I^2C$ Clock**. Connect to $V_{CC}$ with a 2.2K-10 KOhms resistor, even if no $I^2C$ EEPROM is attached. |
| 16 | 23 | SDA | OD | Z | **$I^2C$ Data**. Connect to $V_{CC}$ with a 2.2K-10 KOhms resistor, even if no $I^2C$ EEPROM is attached. |
| | | | | | |
| 55 | 6 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 7 | 14 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 11 | 18 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 17 | 24 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 27 | 34 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 32 | 39 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 43 | 50 | $V_{CC}$ | Power | N/A | **$V_{CC}$**. Connect to 3.3V power source. |
| 53 | 4 | GND | Ground | N/A | **Connect to ground**. |
| 56 | 7 | GND | Ground | N/A | **Connect to ground**. |
| 10 | 17 | GND | Ground | N/A | **Connect to ground**. |
| 12 | 19 | GND | Ground | N/A | **Connect to ground**. |
| 26 | 33 | GND | Ground | N/A | **Connect to ground**. |
| 28 | 35 | GND | Ground | N/A | **Connect to ground**. |
| 41 | 48 | GND | Ground | N/A | **Connect to ground**. |

## Register Summary

**Table 13. *SX2* Register Summary**

| Hex | Size | Name | Description | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Default | Access |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | General Configuration | | | | | | | | | | | |
| 01 | 1 | IFCONFIG | Interface Configuration | IFCLKSRC | 3048MHZ | IFCLKOE | IFCLKPOL | ASYNC | STANDBY | FLAGD/CS# | DISCON | 11001001 | bbbbbbbb |
| 02 | 1 | FLAGSAB | FIFO FLAGA and FLAGB Assignments | FLAGB3 | FLAGB2 | FLAGB1 | FLAGB0 | FLAGA3 | FLAGA2 | FLAGA1 | FLAGA0 | 00000000 | bbbbbbbb |
| 03 | 1 | FLAGSCD | FIFO FLAGC and FLAGD Assignments | FLAGD3 | FLAGD2 | FLAGD1 | FLAGD0 | FLAGC3 | FLAGC2 | FLAGC1 | FLAGC0 | 00000000 | bbbbbbbb |
| 04 | 1 | POLAR | FIFO polarities | WUPOL | 0 | PKTEND | SLOE | SLRD | SLWR | EF | FF | 00000000 | bbbrrrbb |
| 05 | 1 | REVID | Chip Revision | Major | Major | Major | Major | minor | minor | minor | minor | xxxxxxxx | rrrrrrrr |
| | | Endpoint Configuration[9] | | | | | | | | | | | |
| 06 | 1 | EP2CFG | Endpoint 2 Configuration | VALID | dir | TYPE1 | TYPE0 | SIZE | STALL | BUF1 | BUF0 | 10100010 | bbbbbbbb |
| 07 | 1 | EP4CFG | Endpoint 4 Configuration | VALID | dir | TYPE1 | TYPE0 | 0 | STALL | 0 | 0 | 10100000 | bbbbrbrr |
| 08 | 1 | EP6CFG | Endpoint 6 Configuration | VALID | dir | TYPE1 | TYPE0 | SIZE | STALL | BUF1 | BUF0 | 11100010 | bbbbbbbb |
| 09 | 1 | EP8CFG | Endpoint 8 Configuration | VALID | dir | TYPE1 | TYPE0 | 0 | STALL | 0 | 0 | 11100000 | bbbbrbrr |
| 0A | 1 | EP2PKTLENH | Endpoint 2 Packet Length H | INFM1 | OEP1 | ZEROLEN | WORDWIDE | 0 | PL10 | PL9 | PL8 | 00110010 | bbbbbbbb |
| 0B | 1 | EP2PKTLENL | Endpoint 2 Packet Length L (IN only) | PL7 | PL6 | PL5 | PL4 | PL3 | PL2 | PL1 | PL0 | 00000000 | bbbbbbbb |
| 0C | 1 | EP4PKTLENH | Endpoint 4 Packet Length H | INFM1 | OEP1 | ZEROLEN | WORDWIDE | 0 | 0 | PL9 | PL8 | 00110010 | bbbbbbbb |
| 0D | 1 | EP4PKTLENL | Endpoint 4 Packet Length L (IN only) | PL7 | PL6 | PL5 | PL4 | PL3 | PL2 | PL1 | PL0 | 00000000 | bbbbbbbb |
| 0E | 1 | EP6PKTLENH | Endpoint 6 Packet Length H | INFM1 | OEP1 | ZEROLEN | WORDWIDE | 0 | PL10 | PL9 | PL8 | 00110010 | bbbbbbbb |
| 0F | 1 | EP6PKTLENL | Endpoint 6 Packet Length L (IN only) | PL7 | PL6 | PL5 | PL4 | PL3 | PL2 | PL1 | PL0 | 00000000 | bbbbbbbb |
| 10 | 1 | EP8PKTLENH | Endpoint 8 Packet Length H | INFM1 | OEP1 | ZEROLEN | WORDWIDE | 0 | 0 | PL9 | PL8 | 00110010 | bbbbbbbb |
| 11 | 1 | EP8PKTLENL | Endpoint 8 Packet Length L (IN only) | PL7 | PL6 | PL5 | PL4 | PL3 | PL2 | PL1 | PL0 | 00000000 | bbbbbbbb |
| 12 | 1 | EP2PFH | EP2 Programmable Flag H | DECIS | PKTSTAT | IN: PKTS[2] OUT:PFC12 | IN: PKTS[1] OUT:PFC11 | IN: PKTS[0] OUT:PFC10 | 0 | PFC9 | PFC8 | 10001000 | bbbbbbbb |
| 13 | 1 | EP2PFL | EP2 Programmable Flag L | PFC7 | PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 | 00000000 | bbbbbbbb |
| 14 | 1 | EP4PFH | EP4 Programmable Flag H | DECIS | PKTSTAT | 0 | IN: PKTS[1] OUT:PFC10 | IN: PKTS[0] OUT:PFC9 | 0 | 0 | PFC8 | 10001000 | bbbbbbbb |
| 15 | 1 | EP4PFL | EP4 Programmable Flag L | PFC7 | PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 | 00000000 | bbbbbbbb |
| 16 | 1 | EP6PFH | EP6 Programmable Flag H | DECIS | PKTSTAT | IN: PKTS[2] OUT:PFC12 | IN: PKTS[1] OUT:PFC11 | IN: PKTS[0] OUT:PFC10 | 0 | PFC9 | PFC8 | 00001000 | bbbbbbbb |
| 17 | 1 | EP6PFL | EP6 Programmable Flag L | PFC7 | PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 | 00000000 | bbbbbbbb |
| 18 | 1 | EP8PFH | EP8 Programmable Flag H | DECIS | PKTSTAT | 0 | IN: PKTS[1] OUT:PFC10 | IN: PKTS[0] OUT:PFC9 | 0 | 0 | PFC8 | 00001000 | bbbbbbbb |
| 19 | 1 | EP8PFL | EP8 Programmable Flag L | PFC7 | PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 | 00000000 | bbbbbbbb |
| 1A | 1 | EP2ISOINPKTS | EP2 (if ISO) IN Packets per frame (1-3) | 0 | 0 | 0 | 0 | 0 | 0 | INPPF1 | INPPF0 | 00000001 | bbbbbbbb |
| 1B | 1 | EP4ISOINPKTS | EP4 (if ISO) IN Packets per frame (1-3) | 0 | 0 | 0 | 0 | 0 | 0 | INPPF1 | INPPF0 | 00000001 | bbbbbbbb |
| 1C | 1 | EP6ISOINPKTS | EP6 (if ISO) IN Packets per frame (1-3) | 0 | 0 | 0 | 0 | 0 | 0 | INPPF1 | INPPF0 | 00000001 | bbbbbbbb |
| 1D | 1 | EP8ISOINPKTS | EP8 (if ISO) IN Packets per frame (1-3) | 0 | 0 | 0 | 0 | 0 | 0 | INPPF1 | INPPF0 | 00000001 | bbbbbbbb |
| | | FLAGS | | | | | | | | | | | |
| 1E | 1 | EP24FLAGS | Endpoints 2,4 FIFO Flags | 0 | EP4PF | EP4EF | EP4FF | 0 | EP2PF | EP2EF | EP2FF | 00100010 | rrrrrrrr |
| 1F | 1 | EP68FLAGS | Endpoints 6,8 FIFO Flags | 0 | EP8PF | EP8EF | EP8FF | 0 | EP6PF | EP6EF | EP6FF | 01100110 | rrrrrrrr |
| | | INPKTEND/FLUSH[10] | | | | | | | | | | | |
| 20 | 1 | INPKTEND/FLUSH | Force Packet End / Flush FIFOs | FIFO8 | FIFO6 | FIFO4 | FIFO2 | EP3 | EP2 | EP1 | EP0 | 00000000 | wwwwwwww |
| | | USB Configuration | | | | | | | | | | | |

**Table 13. *SX2* Register Summary**

| Hex | Size | Name | Description | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Default | Access |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2A | 1 | USBFRAMEH | USB Frame count H | 0 | 0 | 0 | 0 | 0 | FC10 | FC9 | FC8 | xxxxxxxx | rrrrrrrr |
| 2B | 1 | USBFRAMEL | USB Frame count L | FC7 | FC6 | FC5 | FC4 | FC3 | FC2 | FC1 | FC0 | xxxxxxxx | rrrrrrrr |
| 2C | 1 | MICROFRAME | Microframe count, 0-7 | 0 | 0 | 0 | 0 | 0 | MF2 | MF1 | MF0 | xxxxxxxx | rrrrrrrr |
| 2D | 1 | FNADDR | USB Function address | HSGRANT | FA6 | FA5 | FA4 | FA3 | FA2 | FA1 | FA0 | 00000000 | rrrrrrrr |
| | | Interrupts | | | | | | | | | | | |
| 2E | 1 | INTENABLE | Interrupt Enable | SETUP | EP0BUF | FLAGS | 1 | 1 | ENUMOK | BUSAC-TIVITY | READY | 11111111 | bbbbbbbb |
| | | Descriptor | | | | | | | | | | | |
| 30 | 500 | DESC | Descriptor RAM | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 | xxxxxxxx | wwwwwww |
| | | Endpoint 0 | | | | | | | | | | | |
| 31 | 64 | EP0BUF | Endpoint 0 Buffer | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 | xxxxxxxx | bbbbbbbb |
| 32 | 8/1 | SETUP | Endpoint 0 Set-up Data / Stall | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 | xxxxxxxx | bbbbbbbb |
| 33 | 1 | EP0BC | Endpoint 0 Byte Count | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 | xxxxxxxx | bbbbbbbb |
| | | Un-Indexed Register control | | | | | | | | | | | |
| 3A | 1 | | Un-Indexed Register Low Byte pointer | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 | | |
| 3B | 1 | | Un-Indexed Register High Byte pointer | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 | | |
| 3C | 1 | | Un-Indexed Register Data | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 | | |
| Address | | Un-Indexed Registers in XDATA Space | | | | | | | | | | | |
| 0xE609 | | FIFOPIN-POLAR | FIFO Interface Pins Polarity | 0 | 0 | PKTEND | SLOE | SLRD | SLWR | EF | FF | 00000000 | rrbbbbbb |
| 0xE683 | | TOGCTL | Data Toggle Control | Q | S | R | IO | EP3 | EP2 | EP1 | EP0 | xxxxxxxx | rbbbbbbb |

8.

**Notes**

9. Please note that the *SX2* was not designed to support dynamic modification of these endpoint configuration registers. If your applications need the ability to change endpoint configurations after the device has already enumerated with a specific configuration, please expect some delay in being able to access the FIFOs after changing the configuration. For example, after writing to EP2PKTLENH, you must wait for at least 35 µs measured from the time the READY signal is asserted before writing to the FIFO. This delay time varies for different registers and is not characterized, because the *SX2* was not designed for this dynamic change of endpoint configuration registers.
10. Please note that the *SX2* was not designed to support dynamic modification of the INPKTEND/FLUSH register. If your applications need the ability to change endpoint configurations or access the INPKTEND register after the device has already enumerated with a specific configuration, please expect some delay in being able to access the FIFOs after changing this register. After writing to INPKTEND/FLUSH, you must wait for at least 85 µs measured from the time the READY signal is asserted before writing to the FIFO. This delay time varies for different registers and is not characterized, because the *SX2* was not designed for this dynamic change of endpoint configuration registers.

### IFCONFIG Register 0x01

| IFCONFIG | | | | | | | | 0x01 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | IFCLKSRC | 3048MHZ | IFCLKOE | IFCLKPOL | ASYNC | STANDBY | FLAGD/CS# | DISCON |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

*Bit 7: IFCLKSRC*

This bit selects the clock source for the FIFOs. If IFCLKSRC = 0, the external clock on the IFCLK pin is selected. If IFCLKSRC = 1 (default), an internal 30 or 48 MHz clock is used.

*Bit 6: 3048MHZ*

This bit selects the internal FIFO clock frequency. If 3048MHZ = 0, the internal clock frequency is 30 MHz. If 3048MHZ = 1 (default), the internal clock frequency is 48 MHz.

*Bit 5: IFCLKOE*

This bit selects if the IFCLK pin is driven. If IFCLKOE = 0 (default), the IFCLK pin is floated. If IFCLKOE = 1, the IFCLK pin is driven.

*Bit 4: IFCLKPOL*

This bit controls the polarity of the IFCLK signal.

- When IFCLKPOL=0, the clock has the polarity shown in all the timing diagrams in this data sheet (rising edge is the activating edge).
- When IFCLKPOL=1, the clock is inverted (in some cases may help with satisfying data set-up times).

*Bit 3: ASYNC*

This bit controls whether the FIFO interface is synchronous or asynchronous. When ASYNC = 0, the FIFOs operate synchronously. In synchronous mode, a clock is supplied either internally or externally on the IFCLK pin, and the FIFO control signals function as read and write enable signals for the clock signal.

When ASYNC = 1 (default), the FIFOs operate asynchronously. No clock signal input to IFCLK is required, and the FIFO control signals function directly as read and write strobes.

*Bit 2: STANDBY*

This bit instructs the *SX2* to enter a low-power mode. When STANDBY=1, the *SX2* will enter a low-power mode by turning off its oscillator. The external master should write this bit after it receives a bus activity interrupt (indicating that the host has signaled a USB suspend condition). If *SX2* is disconnected from the USB bus, the external master can write this bit at any time to save power. Once suspended, the *SX2* is awakened either by resumption of USB bus activity or by assertion of its WAKEUP pin.

*Bit 1: FLAGD/CS#*

This bit controls the function of the FLAGD/CS# pin. When FLAGD/CS# = 0 (default), the pin operates as a slave chip select. If FLAGD/CS# = 1, the pin operates as FLAGD.

*Bit 0: DISCON*

This bit controls whether the internal pull-up resistor connected to D+ is pulled high or floating. When DISCON = 1 (default), the pull-up resistor is floating simulating a USB unplug. When DISCON=0, the pull-up resistor is pulled high signaling a USB connection.

### FLAGSAB/FLAGSCD Registers 0x02/0x03

The *SX2* has four FIFO flags output pins: FLAGA, FLAGB, FLAGC, FLAGD.

| FLAGSAB | | | | | | | | 0x02 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | FLAGB3 | FLAGB2 | FLAGB1 | FLAGB0 | FLAGA3 | FLAGA2 | FLAGA1 | FLAGA0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| FLAGSCD | | | | | | | | 0x03 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | FLAGD3 | FLAGD2 | FLAGD1 | FLAGD0 | FLAGC3 | FLAGC2 | FLAGC1 | FLAGC0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

These flags can be programmed to represent various FIFO flags using four select bits for each FIFO. The 4-bit coding for all four flags is the same, as shown in Table 14.

**Table 14. FIFO Flag 4-bit Coding**

| FLAGx3 | FLAGx2 | FLAGx1 | FLAGx0 | Pin Function |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | FLAGA = PF, FLAGB = FF, FLAGC = EF, FLAGD = CS# (actual FIFO is selected by FIFOADR[2:0] pins) |
| 0 | 0 | 0 | 1 | Reserved |
| 0 | 0 | 1 | 0 | Reserved |
| 0 | 0 | 1 | 1 | Reserved |
| 0 | 1 | 0 | 0 | EP2 PF |
| 0 | 1 | 0 | 1 | EP4 PF |
| 0 | 1 | 1 | 0 | EP6 PF |
| 0 | 1 | 1 | 1 | EP8 PF |
| 1 | 0 | 0 | 0 | EP2 EF |
| 1 | 0 | 0 | 1 | EP4 EF |
| 1 | 0 | 1 | 0 | EP6 EF |
| 1 | 0 | 1 | 1 | EP8 EF |
| 1 | 1 | 0 | 0 | EP2 FF |
| 1 | 1 | 0 | 1 | EP4 FF |
| 1 | 1 | 1 | 0 | EP6 FF |
| 1 | 1 | 1 | 1 | EP8 FF |

For the default (0000) selection, the four FIFO flags are fixed-function as shown in the first table entry; the input pins FIFOADR[2:0] select to which of the four FIFOs the flags correspond. These pins are decoded as shown in Table 3 on page 5.

The other (non-zero) values of FLAGx[3:0] allow the designer to independently configure the four flag outputs FLAGA-FLAGD to correspond to any flag-Programmable, Full, or Empty-from any of the four endpoint FIFOs. This allows each flag to be assigned to any of the four FIFOs, including those not currently selected by the FIFOADR [2:0] pins. For example, the external master could be filling the EP2IN FIFO with data while also checking the empty flag for the EP4OUT FIFO.

### POLAR Register 0x04

This register controls the polarities of FIFO pin signals and the WAKEUP pin.

| POLAR | | | | | | | | 0x04 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | WUPOL | 0 | PKTEND | SLOE | SLRD | SLWR | EF | FF |
| Read/ Write | R/W | R/W | R/W | R | R | R | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Bit 7: WUPOL*

This flag sets the polarity of the WAKEUP pin. If WUPOL = 0 (default), the polarity is active LOW. If WUPOL=1, the polarity is active HIGH.

*Bit 5: PKTEND*

This flag selects the polarity of the PKTEND pin. If PKTEND = 0 (default), the polarity is active LOW. If PKTEND = 1, the polarity is active HIGH.

*Bit 4: SLOE*

This flag selects the polarity of the SLOE pin. If SLOE = 0 (default), the polarity is active LOW. If SLOE = 1, the polarity is active HIGH. This bit can only be changed by using the EEPROM configuration load.

*Bit 3: SLRD*

This flag selects the polarity of the SLRD pin. If SLRD = 0 (default), the polarity is active LOW. If SLRD = 1, the polarity is active HIGH. This bit can only be changed by using the EEPROM configuration load.

*SLWR Bit 2*

This flag selects the polarity of the SLWR pin. If SLWR = 0 (default), the polarity is active LOW. If SLWR = 1, the polarity is active HIGH. This bit can only be changed by using the EEPROM configuration load.

*EF Bit 1*

This flag selects the polarity of the EF pin (FLAGA/B/C/D). If EF = 0 (default), the EF pin is pulled low when the FIFO is empty. If EF = 1, the EF pin is pulled HIGH when the FIFO is empty.

*FF Bit 0*

This flag selects the polarity of the FF pin (FLAGA/B/C/D). If FF = 0 (default), the FF pin is pulled low when the FIFO is full. If FF = 1, the FF pin is pulled HIGH when the FIFO is full.

Note that bits 2(SLWR), 3(SLRD) and 4 (SLOE) are READ only bits and cannot be set by the external master or the EEPROM. On power-up, these bits are set to active low polarity. In order to change the polarity after the device is powered-up, the external master must access the previously undocumented (un-indexed) *SX2* register located at XDATA space at 0xE609. This register has exact same bit definition as the POLAR register except that bits 2, 3 and 4 defined as SLWR, SLRD and SLOE respectively are Read/Write bits. Following is the sequence of events that the master should perform for setting this register to 0x1C (setting bits 4, 3, and 2):

1. Send Low Byte of the Register (0x09)
   a. Command address write of address 0x3A
   b. Command data write of upper nibble of the Low Byte of Register Address (0x00)
   c. Command data write of lower nibble of the Low Byte of Register Address (0x09)
2. Send High Byte of the Register (0xE6)
   d. Command address write of address 0x3B
   e. Command data write of upper nibble of the High Byte of Register Address (0x0E)

f. Command data write of lower nibble of the High Byte of Register Address (0x06)

3. Send the actual value to write to the register Register (in this case 0x1C)

g. Command address write of address 0x3C

h. Command data write of upper nibble of the register value (0x01)

i. Command data write of lower nibble of the register value (0x0C)

In order to avoid altering any other bits of the FIFOPINPOLAR register (0xE609) inadvertently, the external master must do a read (from POLAR register), modify the value to set/clear appropriate bits and write the modified value to FIFOPINPOLAR register. The external master may read from the POLAR register using the command read protocol as stated in "Command Protocol" on page 7. Modify the value with the appropriate bit set to change the polarity as needed and write this modified value to the FIFOPINPOLAR register.

### REVID Register 0x05

These register bits define the silicon revision.

| REVID | | | | | | | 0x05 | |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | Major | Major | Major | Major | Minor | Minor | Minor | Minor |
| Read/ Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| De-fault | X | X | X | X | X | X | X | X |

The upper nibble is the major revision. The lower nibble is the minor revision. For example: if REVID = 0x11, then the silicon revision is 1.1.

### EPxCFG Register 0x06–0x09

These registers configure the large, data-handling *SX2* endpoints, EP2, 4, 6, and 8. Figure 3 on page 11 shows the configuration choices for these endpoints. Shaded blocks group endpoint buffers for double-, triple-, or quad-buffering. The endpoint direction is set independently—any shaded block can have any direction.

| EPx-CFG | | | | | | | 0x06, 0x08 | |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | VALID | DIR | TYPE1 | TYPE0 | SIZE | STALL | BUF1 | BUF0 |
| Read/ Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| De-fault | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

*Bit 7: VALID*

The external master sets VALID = 1 to activate an endpoint, and VALID = 0 to deactivate it. All *SX2* endpoints default to valid. An endpoint whose VALID bit is 0 does not respond to any USB traffic. (Note: when setting VALID=0, use default values for all other bits.)

*Bit 6: DIR*

0 = OUT, 1 = IN. Defaults for EP2/4 are DIR = 0, OUT, and for EP6/8 are DIR = 1, IN.

*Bit [5,4]: TYPE1, TYPE0*

These bits define the endpoint type, as shown in Table 15. The TYPE bits apply to all of the endpoint configuration registers. All *SX2* endpoints except EP0 default to BULK.

**Table 15. Endpoint Type**

| TYPE1 | TYPE0 | Endpoint Type |
|---|---|---|
| 0 | 0 | Invalid |
| 0 | 1 | Isochronous |
| 1 | 0 | Bulk (Default) |
| 1 | 1 | Interrupt |

*Bit 3: SIZE*

0 = 512 bytes (default), 1 = 1024 bytes.

Endpoints 4 and 8 can only be 512 bytes and is a read only bit. The size of endpoints 2 and 6 is selectable.

*Bit 2: STALL*

Each bulk endpoint (IN or OUT) has a STALL bit (bit 2). If the external master sets this bit, any requests to the endpoint return a STALL handshake rather than ACK or NAK. The Get Status-Endpoint Request returns the STALL state for the endpoint indicated in byte 4 of the request. Note that bit 7 of the endpoint number EP (byte 4) specifies direction.

*Bit [1,0]: BUF1, BUF0*

For EP2 and EP6 the depth of endpoint buffering is selected via BUF1:0, as shown in Table 16. For EP4 and EP8 the buffer is internally set to double buffered and are read only bits.

**Table 16. Endpoint Buffering**

| BUF1 | BUF0 | Buffering |
|---|---|---|
| 0 | 0 | Quad |
| 0 | 1 | Invalid[11] |
| 1 | 0 | Double |
| 1 | 1 | Triple |

**Note**
11. Setting the endpoint buffering to invalid causes improper buffer allocation

## EPxPKTLENH/L Registers 0x0A–0x11

The external master can use these registers to set smaller packet sizes than the physical buffer size (refer to the previously described EPxCFG registers). The default packet size is 512 bytes for all endpoints. Note that EP2 and EP6 can have maximum sizes of 1024 bytes, and EP4 and EP8 can have maximum sizes of 512 bytes, to be consistent with the endpoint structure.

In addition, the EPxPKTLENH register has four other endpoint configuration bits.

| EPxPK-TLENL | | | | | | | | 0x0B, 0x0D, 0x0F, 0x11 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | PL7 | PL6 | PL5 | PL4 | PL3 | PL2 | PL1 | PL0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| EP2PKTLENH, EP6PKTLENH | | | | | | | | 0x0A, 0x0E |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | INFM 1 | OEP1 | ZERO LEN | WORD WIDE | 0 | PL10 | PL9 | PL8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

| EP4PKTLENH, EP8PKTLENH | | | | | | | | 0x0C, 0x10 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | INFM1 | OEP1 | ZERO LEN | WORD WIDE | 0 | 0 | PL9 | PL8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

*Bit 7: INFM1 EPxPKTLENH.7*

When the external master sets INFM = 1 in an endpoint configuration register, the FIFO flags for that endpoint become valid one sample earlier than when the full condition occurs. These bits take effect only when the FIFOs are operating synchronously according to an internally or externally supplied clock. Having the FIFO flag indications one sample early simplifies some synchronous interfaces. This applies only to IN endpoints. Default is INFM1 = 0.

*Bit 6: OEP1 EPxPKTLENH.6*

When the external master sets an OEP = 1 in an endpoint configuration register, the FIFO flags for that endpoint become valid one sample earlier than when the empty condition occurs. These bits take effect only when the FIFOs are operating synchronously according to an internally or externally supplied clock. Having the FIFO flag indications one sample early simplifies some synchronous interfaces. This applies only to OUT endpoints. Default is OEP1 = 0.

*Bit 5: ZEROLEN EPxPKTLENH.5*

When ZEROLEN = 1 (default), a zero length packet will be sent when the PKTEND pin is asserted and there are no bytes in the current packet. If ZEROLEN = 0, then a zero length packet will not be sent under these conditions.

*Bit 4: WORDWIDE EPxPKTLENH.4*

This bit controls whether the data interface is 8 or 16 bits wide. If WORDWIDE = 0, the data interface is eight bits wide, and FD[15:8] have no function. If WORDWIDE = 1 (default), the data interface is 16 bits wide.

*Bit [2..0]: PL[X:0] Packet Length Bits*

The default packet size is 512 bytes for all endpoints.

## EPxPFH/L Registers 0x12–0x19

The Programmable Flag registers control when the PF goes active for each of the four endpoint FIFOs: EP2, EP4, EP6, and EP8. The EPxPFH/L fields are interpreted differently for the high speed operation and full speed operation and for OUT and IN endpoints.

Following is the register bit definition for high speed operation and for full speed operation (when endpoint is configured as an isochronous endpoint).

| Full Speed ISO and High Speed Mode: EP2PFL, EP4PFL, EP6PFL, EP8PFL | | | | | | | | 0x13, 0x15, 0x17, 0x19 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | PFC7 | PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Full Speed ISO and High Speed Mode: EP4PFH, EP8PFH | | | | | | | | 0x14, 0x18 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | DECIS | PKTSTAT | 0 | IN: PKTS[1] OUT: PFC10 | IN: PKTS[0] OUT: PFC9 | 0 | 0 | PFC8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

| Full Speed ISO and High Speed Mode: EP2PFH, EP6PFH | | | | | | | | 0x12, 0x16 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | DECIS | PKTSTAT | IN: PKTS[2] OUT: PFC12 | IN: PKTS[1] OUT: PFC11 | IN: PKTS[0] OUT: PFC10 | 0 | PFC9 | PFC8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Following is the bit definition for the same register when the device is operating at full speed and the endpoint is not configured as isochronous endpoint.

| Full Speed Non-ISO Mode: EP2PFL, EP4PFL, EP6PFL, EP8PFL | | | | | | | | 0x13, 0x15, 0x17, 0x19 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | IN: PKTS[1] OUT: PFC7 | IN: PKTS[0] OUT: PFC6 | PFC5 | PFC4 | PFC3 | PFC2 | PFC1 | PFC0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Full Speed Non-ISO Mode: EP2PFH, EP6PFH | | | | | | | | 0x12, 0x16 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | DECIS | PKTSTAT | OUT: PFC12 | OUT: PFC11 | OUT: PFC10 | 0 | PFC9 | IN: PKTS[2] OUT: PFC8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

| Full Speed Non-ISO Mode: EP4PFH, EP8PFH | | | | | | | | 0x14, 0x18 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | DECIS | PKT-STAT | 0 | OUT: PFC10 | OUT: PFC9 | 0 | 0 | PFC8 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

*DECIS: EPxPFH.7*

If DECIS = 0, then PF goes high when the byte count i is equal to or less than what is defined in the PF registers. If DECIS = 1 (default), then PF goes high when the byte count equal to or greater than what is set in the PF register. For OUT endpoints, the byte count is the total number of bytes in the FIFO that are available to the external master. For IN endpoints, the byte count is determined by the PKSTAT bit.

*PKSTAT: EPxPFH.6*

For IN endpoints, the PF can apply to either the entire FIFO, comprising multiple packets, or only to the current packet being filled. If PKTSTAT = 0 (default), the PF refers to the entire IN endpoint FIFO. If PKTSTAT = 1, the PF refers to the number of bytes in the current packet.

| PKTSTAT | PF applies to | EPnPFH:L format |
|---|---|---|
| 0 | Number of committed packets + current packet bytes | PKTS[] and PFC[] |
| 1 | Current packet bytes only | PFC[ ] |

*IN: PKTS(2:0)/OUT: PFC[12:10]: EPxPFH[5:3]*

These three bits have a different meaning, depending on whether this is an IN or OUT endpoint.

#### 0.0.0.7 IN Endpoints

If IN endpoint, the meaning of this *EPxPFH[5:3]* bits depend on the PKTSTAT bit setting. When PKTSTAT = 0 (default), the PF considers when there are PKTS packets plus PFC bytes in the FIFO. PKTS[2:0] determines how many packets are considered, according to Table 17.

**Table 17. PKTS Bits**

| PKTS2 | PKTS1 | PKTS0 | Number of Packets |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |

When PKTSTAT = 1, the PF considers when there are PFC bytes in the FIFO, no matter how many packets are in the FIFO. The PKTS[2:0] bits are ignored.

#### 0.0.0.8 OUT Endpoints

The PF considers when there are PFC bytes in the FIFO regardless of the PKTSTAT bit setting.

### EPxISOINPKTS Registers 0x1A–0x1D

| EP2ISOINOKTS, EP4ISOINPKTS, EP6ISOINPKTS, EP8ISOINPKTS | | | | | | | 0x1A, 0x1B, 0x1C, 0x1D | |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | 0 | 0 | 0 | 0 | 0 | INPPF2 | INPPF1 | INPPF0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

For ISOCHRONOUS IN endpoints only, these registers determine the number of packets per frame (only one per frame for full speed mode) or microframe (up to three per microframe for high speed mode), according to the following table.

**Table 18. EPxISOINPKTS**

| INPPF1 | INPPF0 | Packets |
|---|---|---|
| 0 | 0 | Invalid |
| 0 | 1 | 1 (default) |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

### EPxxFLAGS Registers 0x1E–0x1F

The EPxxFLAGS provide an alternate way of checking the status of the endpoint FIFO flags. If enabled, the *SX2* can interrupt the external master when a flag is asserted, and the external master can read these two registers to determine the state of the FIFO flags. If the INFM1 and/or OEP1 bits are set, then the EPxEF and EPxFF bits are actually empty +1 and full –1.

| EP24FLAGS | | | | | | | | 0x1E |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | 0 | EP4PF | EP4EF | EP4FF | 0 | EP2PF | EP2EF | EP2FF |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

| EP68FLAGS | | | | | | | | 0x1F |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | 0 | EP8PF | EP8EF | EP8FF | 0 | EP6PF | EP6EF | EP6FF |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

*EPxPF Bit 6, Bit 2*

This bit is the current state of endpoint x's programmable flag.

*EPxEF Bit 5, Bit 1*

This bit is the current state of endpoint x's empty flag. EPxEF = 1 if the endpoint is empty.

*EPxFF Bit 4, Bit 0*

This bit is the current state of endpoint x's full flag. EPxFF = 1 if the endpoint is full.

### INPKTEND/FLUSH Register 0x20

This register allows the external master to duplicate the function of the PKTEND pin. The register also allows the external master to selectively flush endpoint FIFO buffers.

| INPKTEND/FLUSH | | | | | | | | 0x20 |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | FIFO8 | FIFO6 | FIFO4 | FIFO2 | EP3 | EP2 | EP1 | EP0 |
| Read/Write | W | W | W | W | W | W | W | W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Bit [4..7]: FIFOx

These bits allows the external master to selectively flush any or all of the endpoint FIFOs. By writing the desired endpoint FIFO bit, *SX2* logic flushes the selected FIFO. For example setting bit 7 flushes endpoint 8 FIFO.

Bit [3..0]: EPx

These bits are is used only for IN transfers. By writing the desired endpoint number (2,4,6 or 8), *SX2* logic automatically commits an IN buffer to the USB host. For example, for committing a packet through endpoint 6, set the lower nibble to 6: set bits 1 and 2 high.

### USBFRAMEH/L Registers 0x2A, 0x2B

Every millisecond, the USB host sends an SOF token indicating "Start Of Frame," along with an 11-bit incrementing frame count. The *SX2* copies the frame count into these registers at every SOF.

| USBFRAMEH | | | | | | | | 0x2A |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | 0 | 0 | 0 | 0 | 0 | FC10 | FC9 | FC8 |
| Read/Write | R | R | R | R | R | R | R | R |
| Default | X | X | X | X | X | X | X | x |

| USBFRAMEL | | | | | | | | 0x2B |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | FC7 | FC6 | FC5 | FC4 | FC3 | FC2 | FC1 | FC0 |
| Read/Write | R | R | R | R | R | R | R | R |
| Default | X | X | X | X | X | X | X | X |

One use of the frame count is to respond to the USB SYNC_FRAME Request. If the *SX2* detects a missing or garbled SOF, the *SX2* generates an internal SOF and increments USBFRAMEL–USBRAMEH.

### MICROFRAME Registers 0x2C

| MICROFRAME | | | | | | | | 0x2C |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | 0 | 0 | 0 | 0 | 0 | MF2 | MF1 | MF0 |
| Read/Write | R | R | R | R | R | R | R | R |
| Default | X | X | X | X | X | X | X | x |

MICROFRAME contains a count 0–7 that indicates which of the 125 microsecond microframes last occurred.

This register is active only when *SX2* is operating in high speed mode (480 Mbits/sec).

### FNADDR Register 0x2D

During the USB enumeration process, the host sends a device a unique 7-bit address that the *SX2* copies into this register. There is normally no reason for the external master to know its USB device address because the *SX2* automatically responds only to its assigned address.

| FNADDR | | | | | | | | 0x2D |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | HSGRANT | FA6 | FA5 | FA4 | FA3 | FA2 | FA1 | FA0 |
| Read/Write | R | R | R | R | R | R | R | R |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Bit 7: HSGRANT, Set to 1 if the *SX2* enumerated at high speed. Set to 0 if the *SX2* enumerated at full speed.

Bit[6..0]: Address set by the host.

### INTENABLE Register 0x2E

This register is used to enable/disable the various interrupt sources, and by default all interrupts are enabled.

| INTENABLE | | | | | | | | 0x2E |
|---|---|---|---|---|---|---|---|---|
| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Name | SETUP | EP0 BUF | FLAGS | 1 | 1 | ENUM OK | BUS ACTIVITY | READY |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

*SETUP Bit 7*

Setting this bit to a 1 enables an interrupt when a set-up packet is received from the USB host.

*EP0BUF Bit 6*

Setting this bit to a 1 enables an interrupt when the Endpoint 0 buffer becomes available.

*FLAGS Bit 5*

Setting this bit to a 1 enables an interrupt when an OUT endpoint FIFO's state transitions from empty to not-empty.

*ENUMOK Bit 2*

Setting this bit to a 1 enables an interrupt when *SX2* enumeration is complete.

*BUSACTIVITY Bit 1*

Setting this bit to a 1 enables an interrupt when the *SX2* detects an absence or presence of bus activity.

*READY Bit 0*

Setting this bit to a 1 enables an interrupt when the *SX2* has powered on and performed an internal self-test.

### DESC Register 0x30

This register address is used to write the 500-byte descriptor RAM. The external master writes two bytes (four command data transfers) to this address corresponding to the length of the descriptor or VID/PID/DID data to be written. The external master then consecutively writes that number of bytes into the descriptor RAM in nibble format. For complete details, refer to "Enumeration" on page 8.

### EP0BUF Register 0x31

This register address is used to access the 64-byte Endpoint 0 buffer. The external master can read or write to this register to complete Endpoint 0 data transfers. For complete details, refer to "Endpoint 0" on page 8.

### SETUP Register 0x32

This register address is used to access the 8-byte set-up packet received from the USB host. If the external master writes to this register, it can stall Endpoint 0. For complete details, refer to "Endpoint 0" on page 8.

### EP0BC Register 0x33

This register address is used to access the byte count of Endpoint 0. For Endpoint 0 OUT transfers, the external master can read this register to get the number of bytes transferred from the USB host. For Endpoint 0 IN transfers, the external master writes the number of bytes in the Endpoint 0 buffer to transfer the bytes to the USB host. For complete details, refer to "Endpoint 0" on page 8.

## Absolute Maximum Ratings

Storage Temperature ..... –65°C to +150°C

Ambient Temperature with Power Supplied..... 0°C to +70°C

Supply Voltage to Ground Potential.....–0.5V to +4.0V

DC Input Voltage to Any Pin ..... 5.25V

DC Voltage Applied to Outputs in High-Z State ..... –0.5V to $V_{CC}$ + 0.5V

Power Dissipation..... 936 mW

Static Discharge Voltage..... > 2000V

## Operating Conditions

$T_A$ (Ambient Temperature Under Bias) ..... 0°C to +70°C

Supply Voltage.....+3.0V to +3.6V

Ground Voltage..... 0V

$F_{OSC}$ (Oscillator or Crystal Frequency) ..... 24 MHz ± 100-ppm Parallel Resonant

## DC Electrical Characteristics

**Table 19. DC Characteristics**

| Parameter | Description | Conditions[12] | Min | Typ | Max | Unit |
|---|---|---|---|---|---|---|
| $V_{CC}$ | Supply Voltage | | 3.0 | 3.3 | 3.6 | V |
| $V_{IH}$ | Input High Voltage | | 2 | | 5.25 | V |
| $V_{IL}$ | Input Low Voltage | | –0.5 | | 0.8 | V |
| $I_I$ | Input Leakage Current | 0< $V_{IN}$ < $V_{CC}$ | | | ±10 | µA |
| $V_{OH}$ | Output Voltage High | $I_{OUT}$ = 4 mA | 2.4 | | | V |
| $V_{OL}$ | Output Voltage Low | $I_{OUT}$ = –4 mA | | | 0.4 | V |
| $I_{OH}$ | Output Current High | | | | 4 | mA |
| $I_{OL}$ | Output Current Low | | | | 4 | mA |
| $C_{IN}$ | Input Pin Capacitance | Except D+/D– | | | 10 | pF |
| | | D+/D– | | | 15 | pF |
| $I_{SUSP}$ | Suspend Current | Includes 1.5k integrated pull-up | | 250 | 400 | µA |
| $I_{SUSP}$ | Suspend Current | Excluding 1.5k integrated pull-up | | 30 | 180 | µA |
| $I_{CC}$ | Supply Current | Connected to USB at high speed | | 200 | 260 | mA |
| | | Connected to USB at full speed | | 90 | 150 | mA |
| $T_{RESET}$ | RESET Time after valid power | $V_{CC}$ min = 3.0V | 1.91 | | | mS |

**Note**
12. Specific conditions for $I_{CC}$ measurements: HS typical 3.3V, 25°C, 48 MHz; FS typical 3.3V, 25°C, 48 MHz.

## AC Electrical Characteristics

### USB Transceiver

USB 2.0-certified compliant in full and high speed.

### Command Interface

**Figure 5. Command Synchronous Read Timing Diagram**[13]

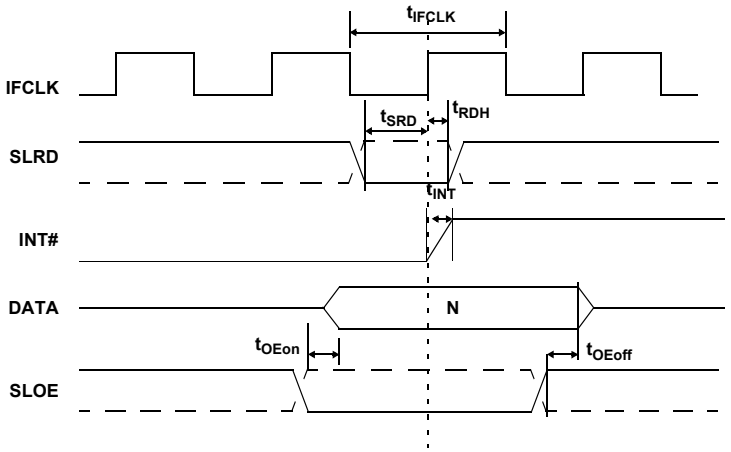



**Table 20. Command Synchronous Read Parameters with Internally Sourced IFCLK**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK period | 20.83 | | ns |
| $t_{SRD}$ | SLRD to Clock Set-up Time | 18.7 | | ns |
| $t_{RDH}$ | Clock to SLRD Hold Time | 0 | | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |
| $t_{INT}$ | Clock to INT# Output Propagation Delay | | 9.5 | ns |

**Table 21. Command Synchronous Read with Externally Sourced IFCLK**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20 | 200 | ns |
| $t_{SRD}$ | SLRD to Clock Set-up Time | 12.7 | | ns |
| $t_{RDH}$ | Clock to SLRD Hold Time | 3.7 | | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |
| $t_{INT}$ | Clock to INT# Output Propagation Delay | | 13.5 | ns |

**Notes**
13. Dashed lines denote signals with programmable polarity.
14. Externally sourced IFCLK must not exceed 50 MHz.

**Figure 6. Command Synchronous Write Timing Diagram**[13]

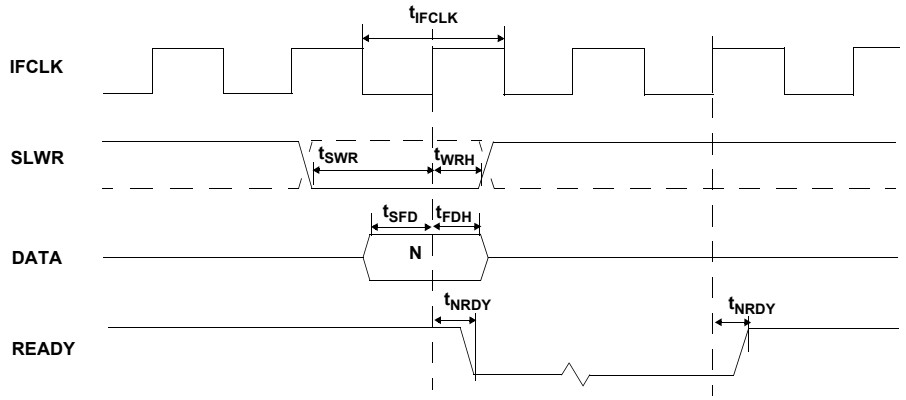



**Table 22. Command Synchronous Write Parameters with Internally Sourced IFCLK**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20.83 | | ns |
| $t_{SWR}$ | SLWR to Clock Set-up Time | 18.1 | | ns |
| $t_{WRH}$ | Clock to SLWR Hold Time | 0 | | ns |
| $t_{SFD}$ | Command Data to Clock Set-up Time | 9.2 | | ns |
| $t_{FDH}$ | Clock to Command Data Hold Time | 0 | | ns |
| $t_{NRDY}$ | Clock to READY Output Propagation Time | | 9.5 | ns |

**Table 23. Command Synchronous Write Parameters with Externally Sourced IFCLK**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20 | 200 | ns |
| $t_{SWR}$ | SLWR to Clock Set-up Time | 12.1 | | ns |
| $t_{WRH}$ | Clock to SLWR Hold Time | 3.6 | | ns |
| $t_{SFD}$ | Command Data to Clock Set-up Time | 3.2 | | ns |
| $t_{FDH}$ | Clock to Command Data Hold Time | 4.5 | | ns |
| $t_{NRDY}$ | Clock to READY Output Propagation Time | | 13.5 | ns |

**Figure 7. Command Asynchronous Read Timing Diagram**[13]

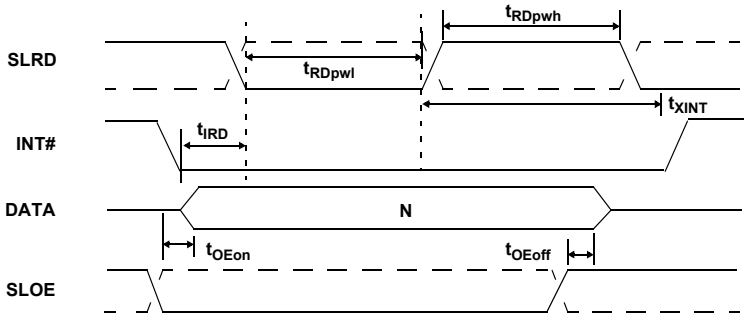


**Table 24. Command Read Parameters**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{RDpwl}$ | SLRD Pulse Width LOW | 50 | | ns |
| $t_{RDpwh}$ | SLRD Pulse Width HIGH | 50 | | ns |
| $t_{IRD}$ | INTERRUPT to SLRD | 0 | | ns |
| $t_{XINT}$ | SLRD to INTERRUPT | | 70 | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |

**Figure 8. Command Asynchronous Write Timing Diagram**[13]

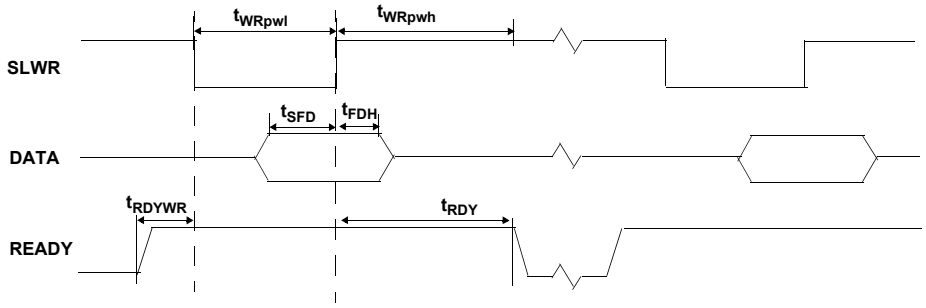



**Figure 9.**

**Table 25. Command Write Parameters**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{WRpwl}$ | SLWR Pulse LOW | 50 | | ns |
| $t_{WRpwh}$ | SLWR Pulse HIGH | 70 | | ns |
| $t_{SFD}$ | SLWR to Command DATA Set-up Time | 10 | | ns |
| $t_{FDH}$ | Command DATA to SLWR Hold Time | 10 | | ns |
| $t_{RDYWR}$ | READY to SLWR Time | 0 | | ns |
| $t_{RDY}$ | SLWR to READY | | 70 | ns |

## FIFO Interface

**Figure 10. Slave FIFO Synchronous Read Timing Diagram**[13]

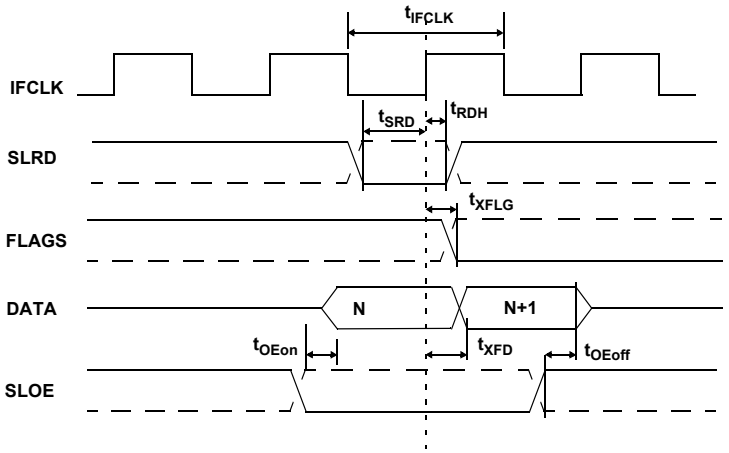


**Table 26. Slave FIFO Synchronous Read with Internally Sourced IFCLK**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20.83 | | ns |
| $t_{SRD}$ | SLRD to Clock Set-up Time | 18.7 | | ns |
| $t_{RDH}$ | Clock to SLRD Hold Time | 0 | | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Delay | | 9.5 | ns |
| $t_{XFD}$ | Clock to FIFO Data Output Propagation Delay | | 11 | ns |

**Table 27. Slave FIFO Synchronous Read with Externally Sourced IFCLK**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20 | 200 | ns |
| $t_{SRD}$ | SLRD to Clock Set-up Time | 12.7 | | ns |
| $t_{RDH}$ | Clock to SLRD Hold Time | 3.7 | | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Delay | | 13.5 | ns |
| $t_{XFD}$ | Clock to FIFO Data Output Propagation Delay | | 15 | ns |

**Figure 11. Slave FIFO Synchronous Write Timing Diagram**[13]

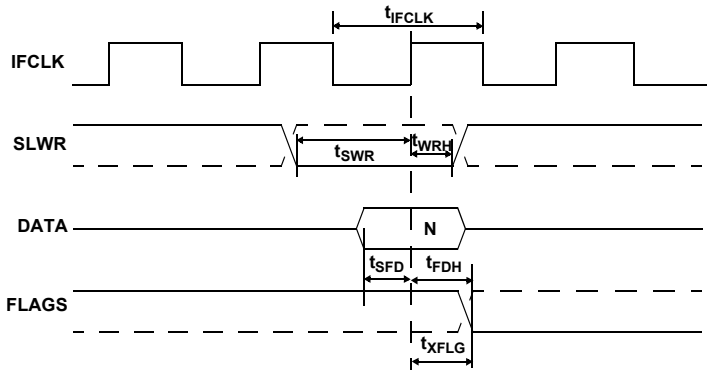



**Figure 12.**

**Table 28. Slave FIFO Synchronous Write Parameters with Internally Sourced IFCLK**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20.83 | | ns |
| $t_{SWR}$ | SLWR to Clock Set-up Time | 18.1 | | ns |
| $t_{WRH}$ | Clock to SLWR Hold Time | 0 | | ns |
| $t_{SFD}$ | FIFO Data to Clock Set-up Time | 9.2 | | ns |
| $t_{FDH}$ | Clock to FIFO Data Hold Time | 0 | | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Time | | 9.5 | ns |

**Table 29. Slave FIFO Synchronous Write Parameters with Externally Sourced IFCLK[14]**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20 | | ns |
| $t_{SWR}$ | SLWR to Clock Set-up Time | 12.1 | | ns |
| $t_{WRH}$ | Clock to SLWR Hold Time | 3.6 | | ns |
| $t_{SFD}$ | FIFO Data to Clock Set-up Time | 3.2 | | ns |
| $t_{FDH}$ | Clock to FIFO Data Hold Time | 4.5 | | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Time | | 13.5 | ns |

**Figure 13. Slave FIFO Synchronous Packet End Strobe Timing Diagram[13]**

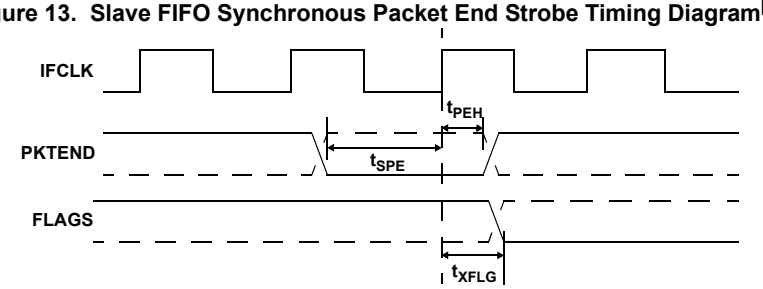



**Table 30. Slave FIFO Synchronous Packet End Strobe Parameters, Internally Sourced IFCLK[14]**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20.83 | | ns |
| $t_{SPE}$ | PKTEND to Clock Set-up Time | 14.6 | | ns |
| $t_{PEH}$ | Clock to PKTEND Hold Time | 0 | | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Delay | | 9.5 | ns |

**Table 31. Slave FIFO Synchronous Packet End Strobe Parameters, Externally Sourced IFCLK[14]**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | IFCLK Period | 20 | 200 | ns |
| $t_{SPE}$ | PKTEND to Clock Set-up Time | 8.6 | | ns |
| $t_{PEH}$ | Clock to PKTEND Hold Time | 2.5 | | ns |
| $t_{XFLG}$ | Clock to FLAGS Output Propagation Delay | | 13.5 | ns |

There is no specific timing requirement that needs to be met for asserting PKTEND pin with regards to asserting SLWR. PKTEND can be asserted with the last data value clocked into the FIFOs or thereafter. The only consideration is the set-up time $t_{SPE}$ and the hold time $t_{PEH}$ must be met.

Although there are no specific timing requirement for the PKTEND assertion, there is a specific corner case condition that needs attention while using the PKTEND to commit a one byte/word packet. There is an additional timing requirement that need to be met when the FIFO is configured to operate in auto mode and it is desired to send two packets back to back: a full packet (full defined as the number of bytes in the FIFO meeting the level set in AUTOINLEN register) committed automatically followed by a short one byte/word packet committed manually using the PKTEND pin. In this particular scenario, user must make sure to assert PKTEND at least one clock cycle after the rising edge that caused the last byte/word to be clocked into the previous auto committed packet. Figure 14 on page 29 shows this scenario. X is the value the AUTOINLEN register is set to when the IN endpoint is configured to be in auto mode.

**Figure 14. Slave FIFO Synchronous Write Sequence and Timing Diagram**

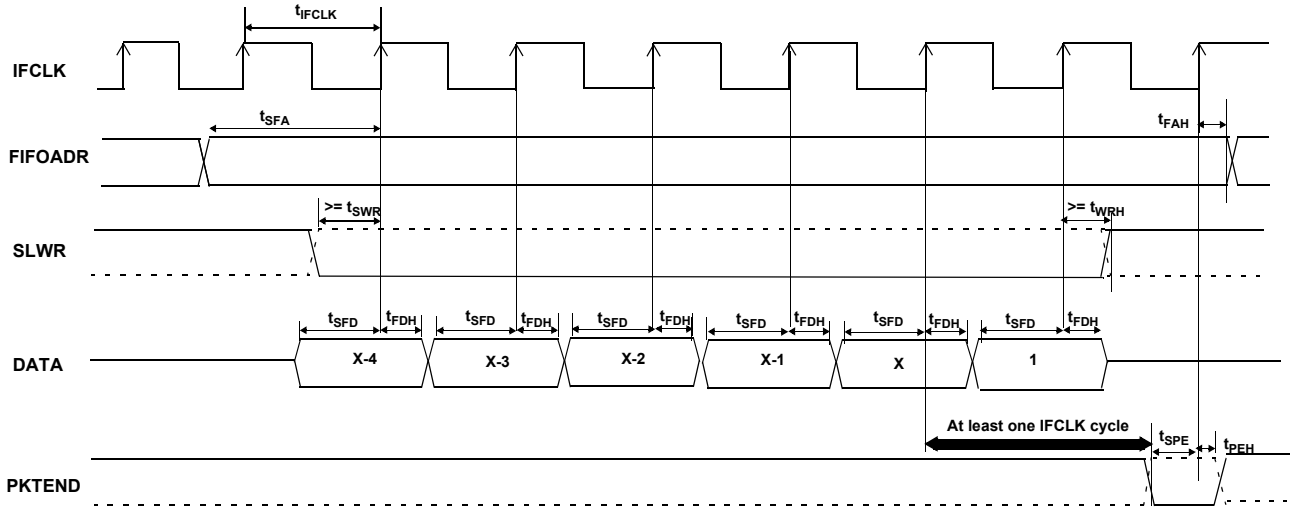



Figure 14 shows a scenario where two packets are being committed. The first packet gets committed automatically when the number of bytes in the FIFO reaches X (value set in AUTOINLEN register) and the second one byte/word short packet being committed manually using PKTEND. Note that there is at least one IFCLK cycle timing between the assertion of PKTEND and clocking of the last byte of the previous packet (causing the packet to be committed automatically). Failing to adhere to this timing, will result in the FX2 failing to send the one byte/word short packet.

**Figure 15. Slave FIFO Synchronous Address Timing Diagram**

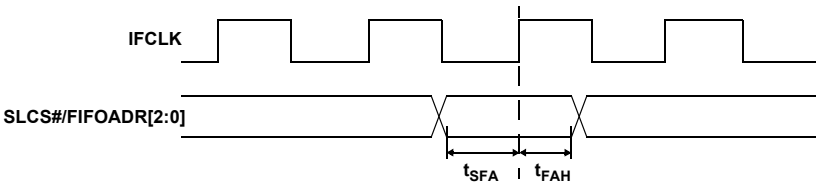



**Table 32. Slave FIFO Synchronous Address Parameters**[14]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{IFCLK}$ | Interface Clock Period | 20 | 200 | ns |
| $t_{SFA}$ | FIFOADR[2:0] to Clock Set-up Time | 25 | | ns |
| $t_{FAH}$ | Clock to FIFOADR[2:0] Hold Time | 10 | | ns |

**Figure 16. Slave FIFO Asynchronous Read Timing Diagram**[13]

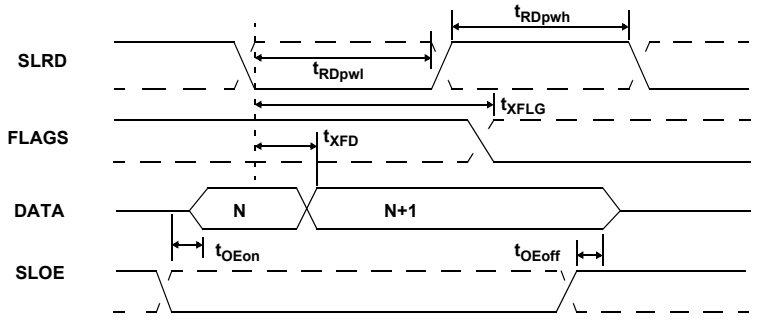



**Table 33. Slave FIFO Asynchronous Read Parameters**[15]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{RDpwl}$ | SLRD Pulse Width Low | 50 | | ns |
| $t_{RDpwh}$ | SLRD Pulse Width HIGH | 50 | | ns |
| $t_{XFLG}$ | SLRD to FLAGS Output Propagation Delay | | 70 | ns |
| $t_{XFD}$ | SLRD to FIFO Data Output Propagation Delay | | 15 | ns |
| $t_{OEon}$ | SLOE Turn-on to FIFO Data Valid | | 10.5 | ns |
| $t_{OEoff}$ | SLOE Turn-off to FIFO Data Hold | | 10.5 | ns |

**Figure 17. Slave FIFO Asynchronous Write Timing Diagram**[13]

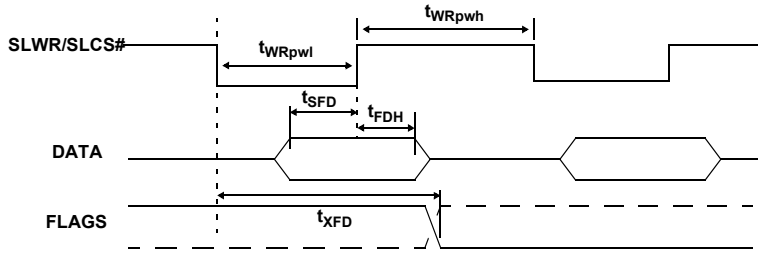



**Table 34. Slave FIFO Asynchronous Write Parameters with Internally Sourced IFCLK**[15]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{WRpwl}$ | SLWR Pulse LOW | 50 | | ns |
| $t_{WRpwh}$ | SLWR Pulse HIGH | 70 | | ns |
| $t_{SFD}$ | SLWR to FIFO DATA Set-up Time | 10 | | ns |
| $t_{FDH}$ | FIFO DATA to SLWR Hold Time | 10 | | ns |
| $t_{XFD}$ | SLWR to FLAGS Output Propagation Delay | | 70 | ns |

**Figure 18. Slave FIFO Asynchronous Packet End Strobe Timing Diagram**

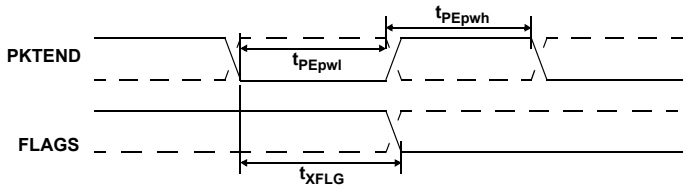



**Table 35. Slave FIFO Asynchronous Packet End Strobe Parameters**[15]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{PEpwl}$ | PKTEND Pulse Width LOW | 50 | | ns |
| $t_{PWpwh}$ | PKTEND Pulse Width HIGH | 50 | | ns |
| $t_{XFLG}$ | PKTEND to FLAGS Output Propagation Delay | | 110 | ns |

**Figure 19. Slave FIFO Asynchronous Address Timing Diagram**[13]

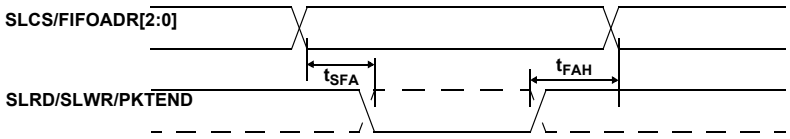



**Table 36. Slave FIFO Asynchronous Address Parameters**[15]

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{SFA}$ | FIFOADR[2:0] to RD/WR/PKTEND Set-up Time | 10 | | ns |
| $t_{FAH}$ | SLRD/PKTEND to FIFOADR[2:0] Hold Time | 20 | | ns |
| $t_{FAH}$ | SLWR to FIFOADR[2:0] Hold Time | 70 | | ns |

## Slave FIFO Address to Flags/Data

Following timing is applicable to synchronous and asynchronous interfaces.

**Figure 20. Slave FIFO Address to Flags/Data Timing Diagram**[12]

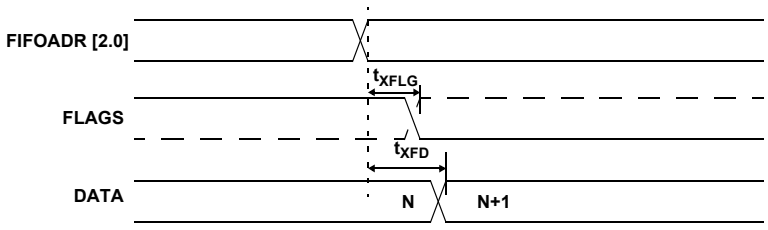



**Table 37. Slave FIFO Address to Flags/Data Parameters**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{XFLG}$ | FIFOADR[2:0] to FLAGS Output Propagation Delay | | 10.7 | ns |
| $t_{XFD}$ | FIFOADR[2:0] to FIFODATA Output Propagation Delay | | 14.3 | ns |

**Note**
15. Slave FIFO asynchronous parameter values are using internal IFCLK setting at 48 MHz.

## Slave FIFO Output Enable

Following timings are applicable to synchronous and asynchronous interfaces.

**Figure 21. Slave FIFO Output Enable Timing Diagram[12]**

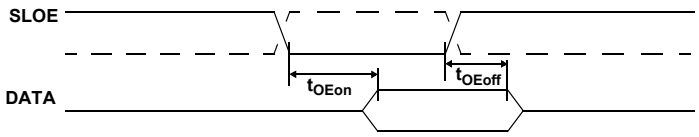



**Table 38. Slave FIFO Output Enable Parameters**

| Parameter | Description | Min | Max | Unit |
|---|---|---|---|---|
| $t_{OEon}$ | SLOE assert to FIFO DATA Output | | 10.5 | ns |
| $t_{OEoff}$ | SLOE deassert to FIFO DATA Hold | | 10.5 | ns |

## Sequence Diagrams

**Figure 22. Slave FIFO Synchronous Read Sequence and Timing Diagram**

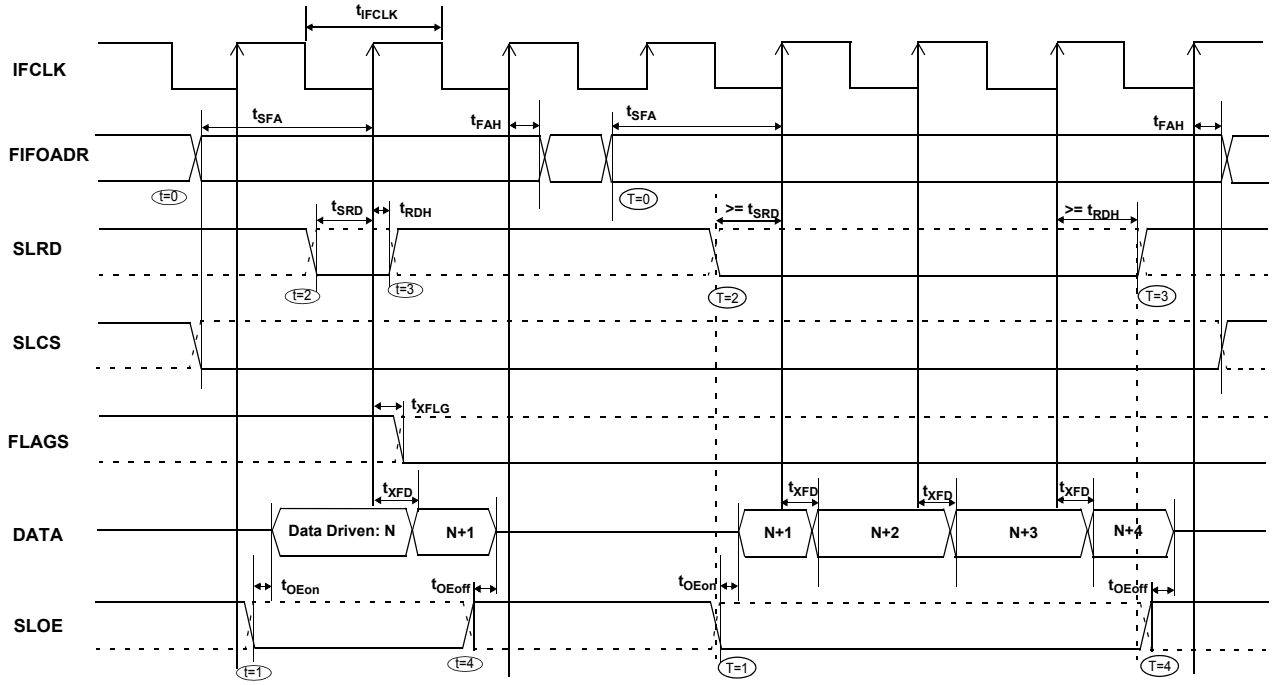



**Figure 23. Slave FIFO Synchronous Sequence of Events Diagram**

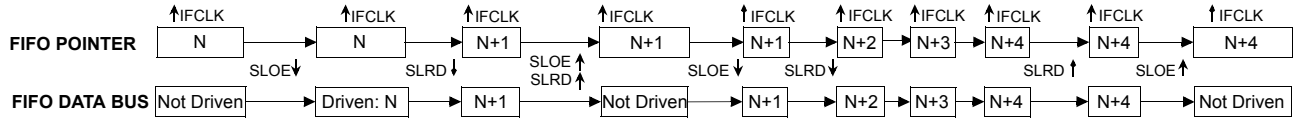



Figure 22 shows the timing relationship of the SLAVE FIFO signals during a synchronous FIFO read using IFCLK as the synchronizing clock. The diagram illustrates a single read followed by a burst read.

- At t = 0 the FIFO address is stable and the signal SLCS is asserted (SLCS may be tied low in some applications).

**Note**: $t_{SFA}$ has a minimum of 25 ns. This means when IFCLK is running at 48 MHz, the FIFO address set-up time is more than one IFCLK cycle.

- At = 1, SLOE is asserted. SLOE is an output enable only, whose sole function is to drive the data bus. The data that is driven on the bus is the data that the internal FIFO pointer is currently pointing to. In this example it is the first data value in the FIFO. Note: the data is pre-fetched and is driven on the bus when SLOE is asserted.
- At t = 2, SLRD is asserted. SLRD must meet the set-up time of $t_{SRD}$ (time from asserting the SLRD signal to the rising edge of the IFCLK) and maintain a minimum hold time of $t_{RDH}$ (time from the IFCLK edge to the deassertion of the SLRD signal). If the SLCS signal is used, it must be asserted with SLRD, or before SLRD is asserted (that is, the SLCS and SLRD signals must both be asserted to start a valid read condition).
- The FIFO pointer is updated on the rising edge of the IFCLK, while SLRD is asserted. This starts the propagation of data from the newly addressed location to the data bus. After a propagation delay of $t_{XFD}$ (measured from the rising edge of IFCLK) the new data value is present. N is the first data value read from the FIFO. In order to have data on the FIFO data bus, SLOE MUST also be asserted.

The same sequence of events are shown for a burst read and are marked with the time indicators of T = 0 through 5.

**Note**: For the burst mode, the SLRD and SLOE are left asserted during the entire duration of the read. In the burst read mode, when SLOE is asserted, data indexed by the FIFO pointer is on the data bus. During the first read cycle, on the rising edge of the clock the FIFO pointer is updated and increments to point to address N+1. For each subsequent rising edge of IFCLK, while the SLRD is asserted, the FIFO pointer is incremented and the next data value is placed on the data bus.

**Figure 24. Slave FIFO Synchronous Write Sequence and Timing Diagram**[13]

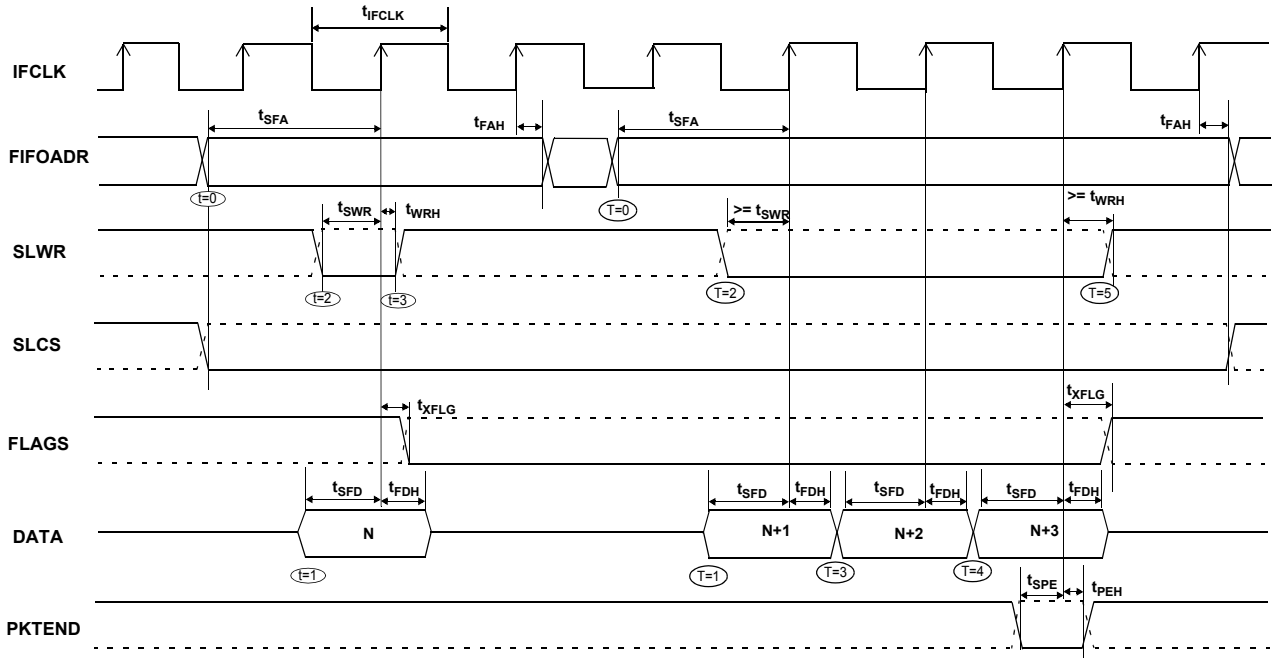



Figure 24 shows the timing relationship of the SLAVE FIFO signals during a synchronous write using IFCLK as the synchronizing clock. The diagram illustrates a single write followed by burst write of 3 bytes and committing all 4 bytes as a short packet using the PKTEND pin.

- At t = 0 the FIFO address is stable and the signal SLCS is asserted. (SLCS may be tied low in some applications) **Note**: $t_{SFA}$ has a minimum of 25 ns. This means when IFCLK is running at 48 MHz, the FIFO address set-up time is more than one IFCLK cycle.
- At t = 1, the external master/peripheral must output the data value onto the data bus with a minimum set up time of $t_{SFD}$ before the rising edge of IFCLK.
- At t = 2, SLWR is asserted. The SLWR must meet the setup time of $t_{SWR}$ (time from asserting the SLWR signal to the rising edge of IFCLK) and maintain a minimum hold time of $t_{WRH}$ (time from the IFCLK edge to the de-assertion of the SLWR signal). If SLCS signal is used, it must be asserted with SLWR or before SLWR is asserted. (i.e.,the SLCS and SLWR signals must both be asserted to start a valid write condition).
  - While the SLWR is asserted, data is written to the FIFO and on the rising edge of the IFCLK, the FIFO pointer is incremented. The FIFO flag will also be updated after a delay of $t_{XFLG}$ from the rising edge of the clock.

The same sequence of events are also shown for a burst write and are marked with the time indicators of T = 0 through 5. Note: For the burst mode, SLWR and SLCS are left asserted for the entire duration of writing all the required data values. In this burst

write mode, once the SLWR is asserted, the data on the FIFO data bus is written to the FIFO on every rising edge of IFCLK. The FIFO pointer is updated on each rising edge of IFCLK. In Figure 24 on page 33, once the four bytes are written to the FIFO, SLWR is deasserted. The short 4-byte packet can be committed to the host by asserting the PKTEND signal.

There is no specific timing requirement that needs to be met for asserting PKTEND signal with regards to asserting the SLWR signal. PKTEND can be asserted with the last data value or thereafter. The only consideration is the set-up time $t_{SPE}$ and the hold time $t_{PEH}$ must be met. In the scenario of Figure 24 on page 33, the number of data values committed includes the last value written to the FIFO. In this example, both the data value and the PKTEND signal are clocked on the same rising edge of IFCLK. PKTEND can be asserted in subsequent clock cycles. The FIFOADDR lines should be held constant during the PKTEND assertion.

Although there are no specific timing requirement for the PKTEND assertion, there is a specific corner case condition that needs attention while using the PKTEND to commit a one byte/word packet. Additional timing requirements exists when the FIFO is configured to operate in auto mode and it is desired to send two packets: a full packet (full defined as the number of bytes in the FIFO meeting the level set in AUTOINLEN register) committed automatically followed by a short one byte/word packet committed manually using the PKTEND pin. In this case, the external master must make sure to assert the PKTEND pin at least one clock cycle after the rising edge that caused the last byte/word to be clocked into the previous auto committed packet (the packet with the number of bytes equal to what is set in the AUTOINLEN register).

**Figure 25. Slave FIFO Asynchronous Read Sequence and Timing Diagram**

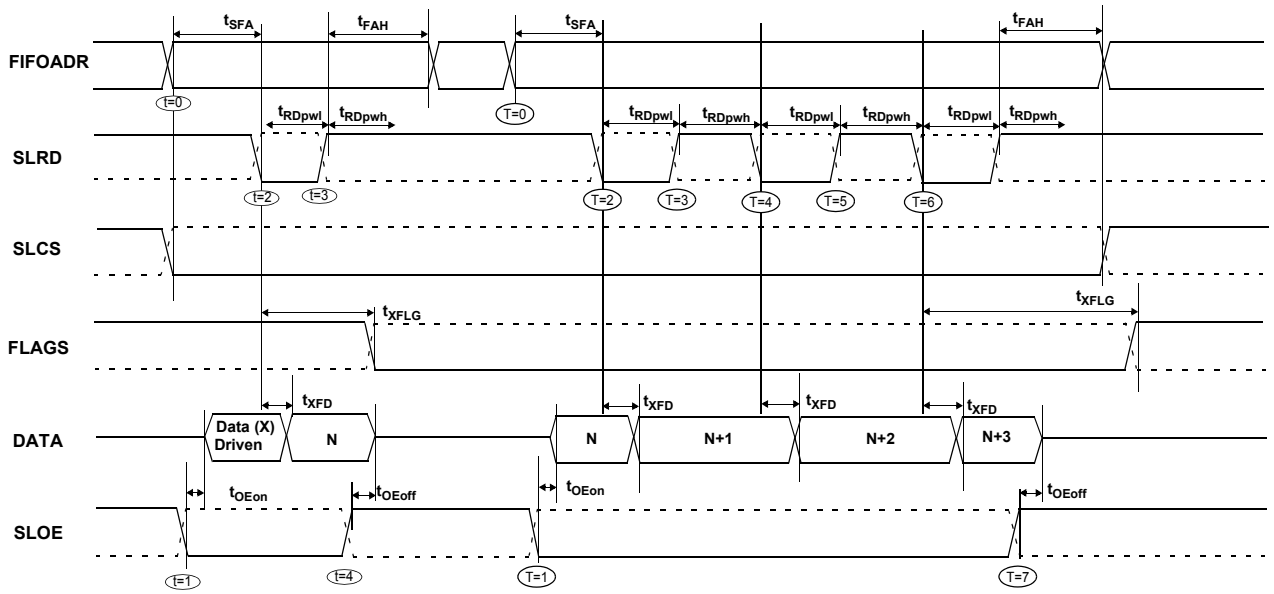



**Figure 26. Slave FIFO Asynchronous Read Sequence of Events Diagram**

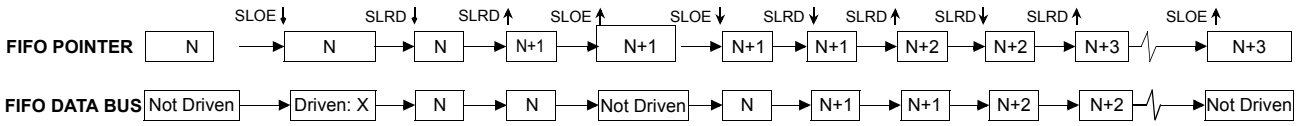



Figure 25 shows the timing relationship of the SLAVE FIFO signals during an asynchronous FIFO read. It shows a single read followed by a burst read.

- At t = 0 the FIFO address is stable and the SLCS signal is asserted.
- At t = 1, SLOE is asserted. This results in the data bus being driven. The data that is driven on to the bus is previous data, it data that was in the FIFO from a prior read cycle.
- At t = 2, SLRD is asserted. The SLRD must meet the minimum active pulse of $t_{RDpwl}$ and minimum de-active pulse width of $t_{RDpwh}$. If SLCS is used then, SLCS must be asserted with SLRD or before SLRD is asserted (i.e., the SLCS and SLRD signals must both be asserted to start a valid read condition).
- The data that will be driven, after asserting SLRD, is the updated data from the FIFO. This data is valid after a propagation delay of $t_{XFD}$ from the activating edge of SLRD. In Figure 26, data N is the first valid data read from the FIFO. For data to appear on the data bus during the read cycle (i.e. SLRD is asserted), SLOE MUST be in an asserted state. SLRD and SLOE can also be tied together.

The same sequence of events is also shown for a burst read marked with T = 0 through 5. Note that in burst read mode, during SLOE is assertion, the data bus is in a driven state and outputs the previous data. Once SLRD is asserted, the data from the FIFO is driven on the data bus (SLOE must also be asserted) and then the FIFO pointer is incremented.

**Figure 27. Slave FIFO Asynchronous Write Sequence and Timing Diagram**[13]

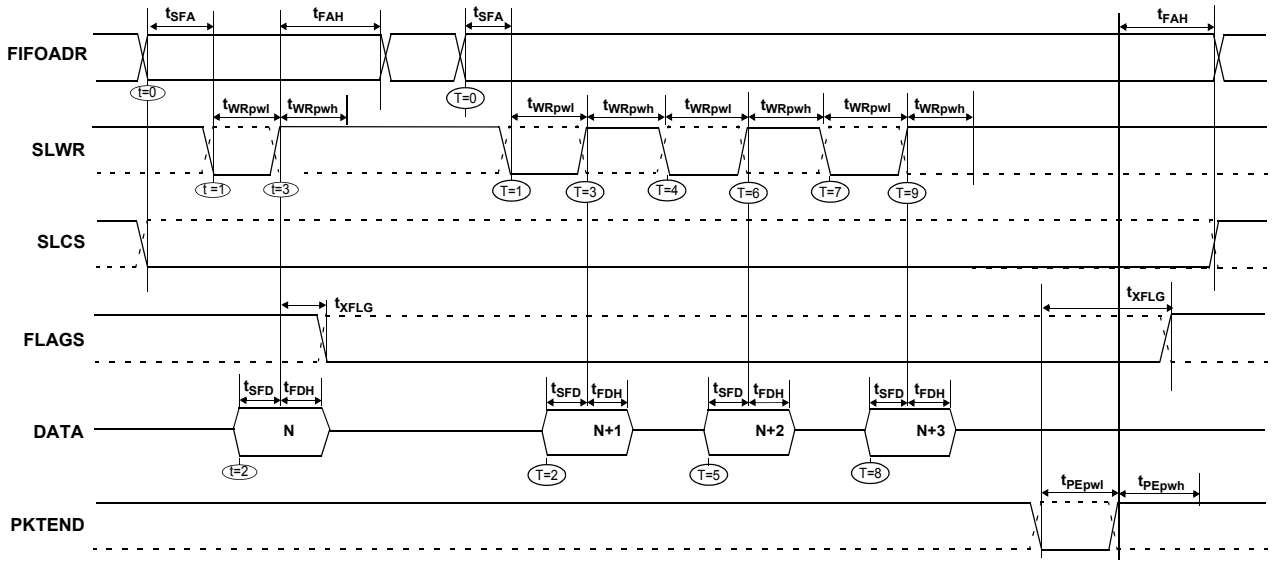



Figure 27 diagrams the timing relationship of the SLAVE FIFO write in an asynchronous mode. The diagram shows a single write followed by a burst write of 3 bytes and committing the 4-byte-short packet using PKTEND.

- At t = 0 the FIFO address is applied, insuring that it meets the set-up time of $t_{SFA}$. If SLCS is used, it must also be asserted (SLCS may be tied low in some applications).
- At t = 1 SLWR is asserted. SLWR must meet the minimum active pulse of $t_{WRpwl}$ and minimum de-active pulse width of $t_{WRpwh}$. If the SLCS is used, it must be asserted with SLWR or before SLWR is asserted.
  - At t = 2, data must be present on the bus $t_{SFD}$ before the deasserting edge of SLWR.
  - At t = 3, deasserting SLWR will cause the data to be written from the data bus to the FIFO and then increments the FIFO pointer. The FIFO flag is also updated after $t_{XFLG}$ from the deasserting edge of SLWR.

The same sequence of events are shown for a burst write and is indicated by the timing marks of T = 0 through 5.

**Note**: In the burst write mode, once SLWR is deasserted, the data is written to the FIFO and then the FIFO pointer is incremented to the next byte in the FIFO. The FIFO pointer is post incremented.

In Figure 27 once the four bytes are written to the FIFO and SLWR is deasserted, the short 4-byte packet can be committed to the host using the PKTEND. The external device should be designed to not assert SLWR and the PKTEND signal at the same time. It should be designed to assert the PKTEND after SLWR is deasserted and met the minimum de-asserted pulse width. The FIFOADDR lines are to be held constant during the PKTEND assertion.

## Default Descriptor

```
//Device Descriptor
18,               //Descriptor length
1,                //Descriptor type
00,02,            //Specification Version (BCD)
00,               //Device class
00,               //Device sub-class
00,               //Device sub-sub-class
64,               //Maximum packet size
LSB(VID),MSB(VID),//Vendor ID
LSB(PID),MSB(PID),//Product ID
LSB(DID),MSB(DID),//Device ID
1,                //Manufacturer string index
2,                //Product string index
```

```
0,                  //Serial number string index
1,                  //Number of configurations

//DeviceQualDscr
10,                 //Descriptor length
6,                  //Descriptor type
0x00,0x02,          //Specification Version (BCD)
00,                 //Device class
00,                 //Device sub-class
00,                 //Device sub-sub-class
64,                 //Maximum packet size
1,                  //Number of configurations
0,                  //Reserved

//HighSpeedConfigDscr
9,                  //Descriptor length
2,                  //Descriptor type
46,                 //Total Length (LSB)
0,                  //Total Length (MSB)
1,                  //Number of interfaces
1,                  //Configuration number
0,                  //Configuration string
0xA0,               //Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
50,                 //Power requirement (div 2 ma)

//Interface Descriptor
9,                  //Descriptor length
4,                  //Descriptor type
0,                  //Zero-based index of this interface
0,                  //Alternate setting
4,                  //Number of end points
0xFF,               //Interface class
0x00,               //Interface sub class
0x00,               //Interface sub sub class
0,                  //Interface descriptor string index

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x02,               //Endpoint number, and direction
2,                  //Endpoint type
0x00,               //Maximum packet size (LSB)
0x02,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x04,               //Endpoint number, and direction
2,                  //Endpoint type
0x00,               //Maximum packet size (LSB)
0x02,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
```

```
0x86,               //Endpoint number, and direction
2,                  //Endpoint type
0x00,               //Maximum packet size (LSB)
0x02,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x88,               //Endpoint number, and direction
2,                  //Endpoint type
0x00,               //Maximum packet size (LSB)
0x02,               //Max packet size (MSB)
0x00,               //Polling interval

//FullSpeedConfigDscr
9,                  //Descriptor length
2,                  //Descriptor type
46,                 //Total Length (LSB)
0,                  //Total Length (MSB)
1,                  //Number of interfaces
1,                  //Configuration number
0,                  //Configuration string
0xA0,               //Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
50,                 //Power requirement (div 2 ma)

//Interface Descriptor
9,                  //Descriptor length
4,                  //Descriptor type
0,                  //Zero-based index of this interface
0,                  //Alternate setting
4,                  //Number of end points
0xFF,               //Interface class
0x00,               //Interface sub class
0x00,               //Interface sub sub class
0,                  //Interface descriptor string index

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x02,               //Endpoint number, and direction
2,                  //Endpoint type
0x40,               //Maximum packet size (LSB)
0x00,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x04,               //Endpoint number, and direction
2,                  //Endpoint type
0x40,               //Maximum packet size (LSB)
0x00,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
```

```
5,                  //Descriptor type
0x86,               //Endpoint number, and direction
2,                  //Endpoint type
0x40,               //Maximum packet size (LSB)
0x00,               //Max packet size (MSB)
0x00,               //Polling interval

//Endpoint Descriptor
7,                  //Descriptor length
5,                  //Descriptor type
0x88,               //Endpoint number, and direction
2,                  //Endpoint type
0x40,               //Maximum packet size (LSB)
0x00,               //Max packet size (MSB)
0x00,               //Polling interval

//StringDscr

//StringDscr0
4,                  //String descriptor length
3,                  //String Descriptor
0x09,0x04,                          //US LANGID Code

//StringDscr1
16,                 //String descriptor length
3,                  //String Descriptor
'C',00,
'y',00,
'p',00,
'r',00,
'e',00,
's',00,
's',00,

//StringDscr2
20,                 //String descriptor length
3,                  //String Descriptor
'C',00,
'Y',00,
'7',00,
'C',00,
'6',00,
'8',00,
'0',00,
'0',00,
'1',00,
```

## General PCB Layout Guidelines[16]

The following recommendations should be followed to ensure reliable high-performance operation.

- At least a four-layer impedance controlled boards are required to maintain signal quality.
- Specify impedance targets (ask your board vendor what they can achieve).
- To control impedance, maintain trace widths and trace spacing.
- Minimize stubs to minimize reflected signals.
- Connections between the USB connector shell and signal ground must be done near the USB connector.
- Bypass/flyback caps on VBus, near connector, are recommended.
- DPLUS and DMINUS trace lengths should be kept to within 2 mm of each other in length, with preferred length of 20–30 mm.
- Maintain a solid ground plane under the DPLUS and DMINUS traces. Do not allow the plane to be split under these traces.

- It is preferred to have no vias placed on the DPLUS or DMINUS trace routing.
- Isolate the DPLUS and DMINUS traces from all other signal traces by no less than 10 mm.

## Quad Flat Package No Leads (QFN) Package Design Notes

Electrical contact of the part to the Printed Circuit Board (PCB) is made by soldering the leads on the bottom surface of the package to the PCB. Hence, special attention is required to the heat transfer area below the package to provide a good thermal bond to the circuit board. A Copper (Cu) fill is to be designed into the PCB as a thermal pad under the package. Heat is transferred from the *SX2* through the device's metal paddle on the bottom side of the package. Heat from here, is conducted to the PCB at the thermal pad. It is then conducted from the thermal pad to the PCB inner ground plane by a 5 x 5 array of via. A via is a plated through hole in the PCB with a finished diameter of 13 mil. The QFN's metal die paddle must be soldered to the PCB's thermal pad. Solder mask is placed on the board top side over each via to resist solder flow into the via. The mask on the top side also minimizes outgassing during the solder reflow process.

For further information on this package design please refer to "Application Notes for Surface Mount Assembly of Amkor's *Micro*LeadFrame® (MLF®) Packages." This application note can be downloaded from Amkor's web site from the following URL: http://www.amkor.com/products/notes_papers/MLFAppNote.pdf. The application note provides detailed information on board mounting guidelines, soldering flow, rework process, etc.

Figure 28 displays a cross-sectional area underneath the package. The cross section is of only one via. The solder paste template needs to be designed to allow at least 50% solder coverage. The thickness of the solder paste template should be 5 mil. It is recommended that "No Clean" type 3 solder paste is used for mounting the part. Nitrogen purge is recommended during reflow.

Figure 29 is a plot of the solder mask pattern and Figure 30 displays an X-Ray image of the assembly (darker areas indicate solder).

**Figure 28. Cross section of the Area Underneath the QFN Package**

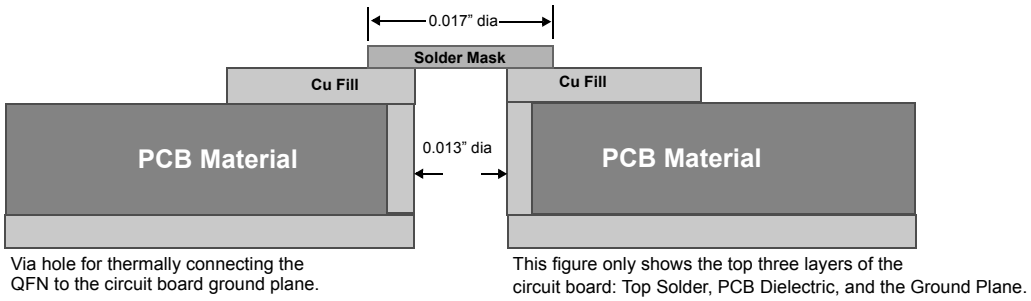



**Figure 29. Plot of the Solder Mask (White Area)**

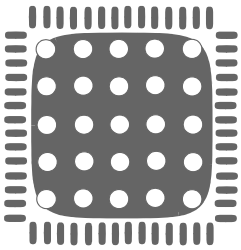
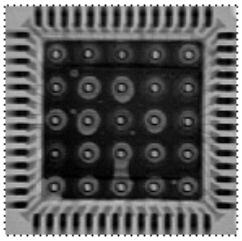


**Figure 30. X-Ray Image of the Assembly**



**Note**
16. Source for recommendations: *High Speed USB Platform Design Guidelines*, http://www.usb.org/developers/data/hs_usb_pdg_r1_0.pdf.

## Ordering Information

| Ordering Code | Package Type |
|---|---|
| CY7C68001-56PVC | 56 SSOP |
| CY7C68001-56LFC | 56 QFN (Punch) |
| CY7C68001-56PVXC | 56 SSOP, Pb-free |
| CY7C68001-56LFXC | 56 QFN, Pb-free (Punch) |
| CY3682 | *EZ-USB SX2* Development Kit |
| CY7C68001-56LTXC | 56 QFN (Sawn), Pb-free |

## Package Diagrams

**Figure 31. 56-Pin Shrunk Small Outline Package 056**

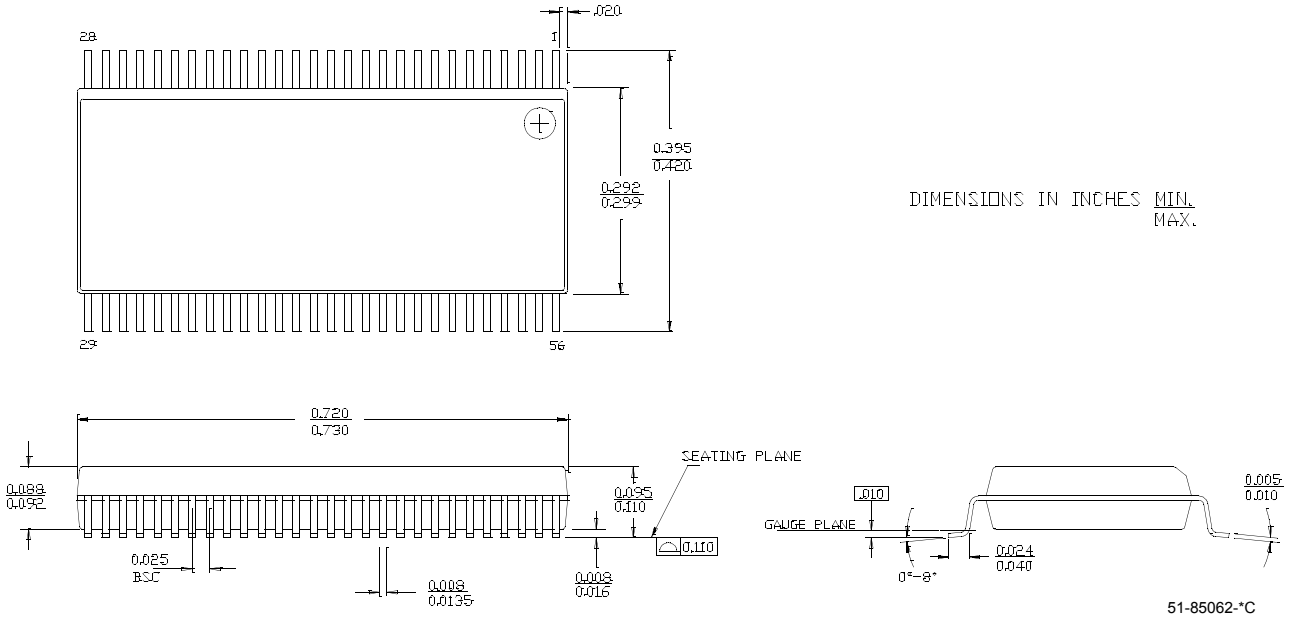
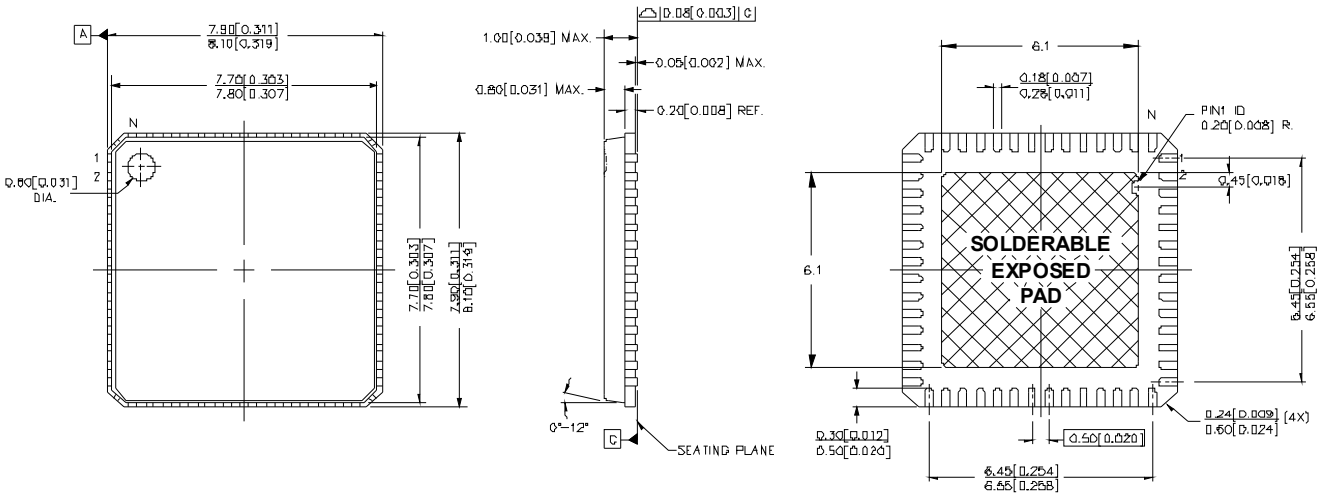


**Figure 32. LF56A 56-Pin (Punch) QFN Package**

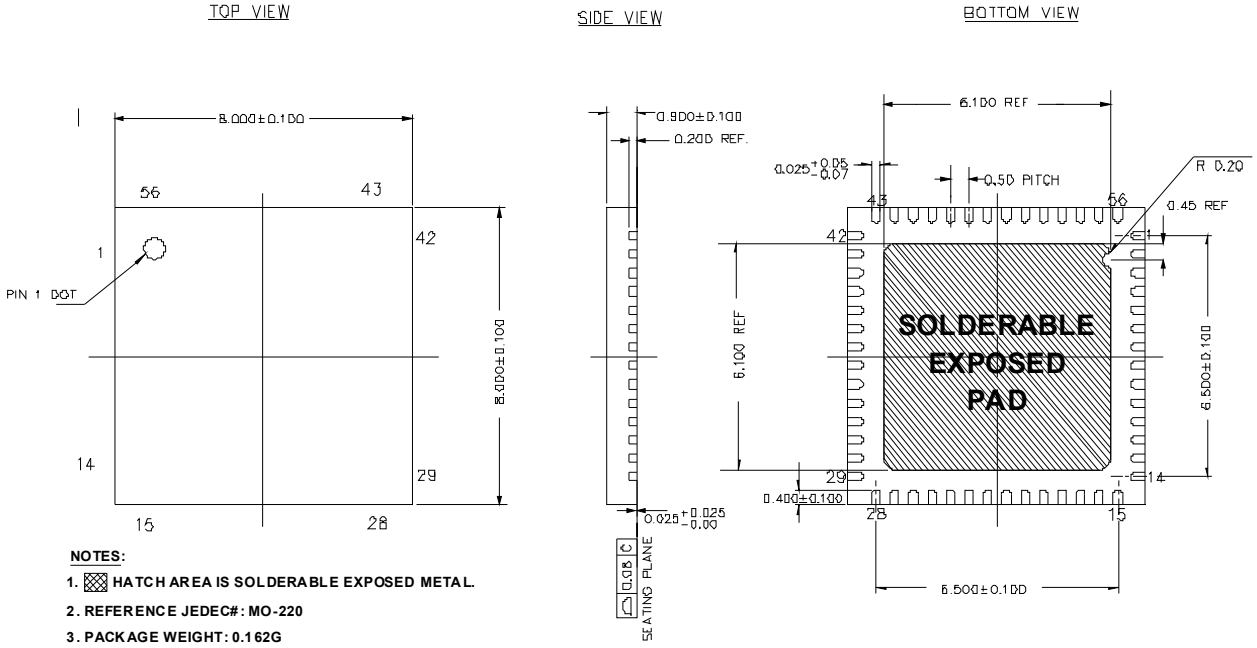



**NOTES:**

1. HATCH AREA IS SOLDERABLE EXPOSED METAL.
2. REFERENCE JEDEC#: MO-220
3. PACKAGE WEIGHT: 0.162g
4. ALL DIMENSIONS ARE IN MM [MIN/MAX]
5. PACKAGE CODE

51-85144 *G

**Figure 33. 56-Pin Sawn QFN (8X8X0.90 MM)**




**NOTES:**

1. HATCH AREA IS SOLDERABLE EXPOSED METAL.
2. REFERENCE JEDEC#: MO-220
3. PACKAGE WEIGHT: 0.162G

51-85187 *C

Firmat: Sergio Morlans Iglesias
Bellaterra, 15 de Setembre 2009

**Resum**

Aquest project es situa dins del marc del CNM-IMB (CSIC). Consisteix en el disseny de un sistema de biòpsia mamaria en temps real. Per realitzar aquest sistema s'ha dissenyat una plataforma de lectura, test y caracterització pel ROIC Medipix2 que es basa en el microprocessador LEON3 y es programat sobre una FPGA.

**Resumen**

Este proyecto se sitúa dentro de marco del CNM-IMB (CSIC). Consiste en el diseño de un sistema de biopsia mamaria en tiempo real. Para poder realizar este sistema se ha diseñado una plataforma de lectura, test y caracterización para el ROICs Medipix2 basada en el microprocesador LEON3 y programado sobre una FPGA.

**Abstract**

This project is within CNM-IMB (CSIC)'s scope. It consists of the design of a breast biopsy system in real time. To make this system has been designed a reader, test and characterization platform for ROIC Medipix2 based on LEON3 microprocessor and programmed on FPGA.