ML300 Reference Design

User Guide

UG057 (v1.1) March 18, 2004

XILINX®

The following table shows the revision history for this document..

	Version	Revision
01/12/04	1.0	Initial Xilinx release.
03/18/04	1.1	Updates for EDK 6.2i.

Table of Contents

Chapter 7: OPB to PCI Bridge Lite

Chapter 12: OPB PS/2 Controller (Dual)

Chapter 13: PLB TFT LCD Controller

About This Guide

This user guide documents the ML300 reference design.

Guide Contents

This manual contains the following chapters:

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records http://support.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm

Resource	Description/URL
Data Sheets	Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://support.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment http://www.support.xilinx.com/xlnx/xil_tt_home.jsp

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *`design_name`*
Helvetica bold	Commands that you select from a menu	**File** → **Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *`design_name`*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*`option_name`*] *`design_name`*
Braces { }	A list of items from which you must choose one or more	**`lowpwr ={on|off}`**

Convention	Meaning or Use	Example
Vertical bar |	Separates items in a list of choices	`lowpwr ={on|off}`

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Repetitive material that has been omitted	`allow block block_name loc1 loc2 ... locn;`

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Handbook*.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Introduction to ML300 Embedded PPC405 Reference System

Introduction

This chapter briefly describes the reference system provided for the ML300 Evaluation Platform. The ML300 Embedded PPC405 Reference System contains a combination of known working hardware and software elements that, together, create an entire system. It demonstrates a system utilizing the Processor Local Bus (PLB), On-Chip Peripheral Bus (OPB), Device Control Register (DCR) Bus, and PPC405 On-Chip Memory (OCM). The design operates under the Embedded Development Kit (EDK) suite of tools which provides a graphical tool framework for designing embedded hardware and software. The reference system is intended to familiarize users with the Virtex-II Pro product, its design tool flows, and its features. While it does not contain all elements a user system might require, it provides a foundation for those who are just learning about the embedded PowerPC processor in Virtex-II Pro FPGAs.

Requirements

The following hardware and software are required in order to use the ML300 Embedded PPC405 Reference System.

Operating System Requirements:

Windows 2000/XP Professional or Solaris 2.8/2.9

Note: A PC is required for FPGA download and debug via Xilinx download cables.

Hardware Requirements:

Xilinx ML300 Development Board

Software Requirements:

Embedded Development Kit (EDK) 6.2

ISE 6.2i

ModelSim SE 5.7b

Note: Later versions are expected to work, but were not tested.

V2PDK Users and New EDK Users

For users of the Virtex-II Pro™ Developer's Kit (V2PDK), the ML300 Embedded PPC405 Reference System provides an example design to help in the migration to the EDK tools. The EDK version of this design is very similar in functionality and capability as the V2PDK

version. The EDK designs attempts to preserve as much software and hardware compatibility as possible to the V2PDK design.

For new EDK users, the ML300 Embedded PPC405 Reference System provides an excellent example of how the EDK tools can be used to design a full featured embedded system consisting of hardware, software, and operating systems. The reference system also illustrates how to perform debug and simulation of designs under EDK.

References to additional information about learning to use EDK is available in "Further Reading," page 17.

CoreConnect

Download and installation of the IBM CoreConnect Toolkit is highly recommended for use with the ML300 Embedded PPC405 Reference System. The CoreConnect Toolkit is only available to CoreConnect licensees. Xilinx has simplified the process of becoming a CoreConnect licensee through a web based registration that is available at http://www.xilinx.com/coreconnect. CoreConnect licensees are entitled to full access to the CoreConnect Toolkit including powerful bus functional modeling, bus monitoring tools and periodic updates. To get the most out of the Embedded Development Kit, Xilinx recommends the use of the IBM CoreConnect Toolkit.

Reference System Information

This section is an overview of the features of the ML300 Embedded PPC405 Reference System. Although the information contained in the reference system chapter is not exhaustive, it covers the basic requirements to effectively use PPC405. Chapter 2, "ML300 Embedded PPC405 Reference System" and Chapter 3, "EDK Tutorial and Demonstration" have instructions on how to simulate, synthesize, and run the designs through the Xilinx Implementation Tools (ISE) for the Virtex-II Pro family.

The reference system chapters contain sections about:

- Hardware used in the system
- HDL file organization
- Simulation and verification
  - Using SWIFT
  - Using Bus Functional Model (BFM)
- Synthesis and implementation
- Software applications that interoperate with the system
- How to run the software applications
- Directory structure of each system

The ML300 Embedded PPC405 Reference System is an example of a completely embedded computer. It provides a wide variety of memory interfaces on three differing buses, as well as various peripherals such as IIC, General Purpose I/O (GPIO), UARTs, PCI interface, TFT LCD controller, and a memory-mapped DCR bus bridge. The ML300 Embedded PPC405 Reference System combines the elements of a typical embedded system by taking advantage of the architectural features of the PPC405, such as separated Instruction-Side and Data-Side OCM interfaces. It illustrates the use of a typical segmented bus design where the higher-speed elements (such as memory) are differentiated from the lower-speed elements (such as GPIO) through the use of bus arbiters and bridges. This system provides an excellent example of the various elements a user might use to run a Real-Time

Operating System (RTOS). The example software provided with this reference system is designed to demonstrate it as only a stand-alone application or under an operating system such as Linux or VxWorks.

The Embedded PPC405 Reference System provides additional study of the PLB, OCM, and DCR buses. In addition, it affords the opportunity to see how OPB-based devices are used in a system. Step-by-step instructions are provided to help the user through the design flow and to target a Virtex-II Pro device. Users can modify the ML300 Embedded PPC405 Reference System to add and subtract peripherals, as well as to change the software for their own custom-designed systems. These designs can be fully simulated, synthesized, and run through place-and-route to produce a bitstream for Virtex-II Pro devices.

Further Reading

Xilinx provides a wealth of valuable information to assist you in your design efforts. Some of the relevant documentation is listed below with more information available through the Xilinx Support website at http://support.xilinx.com. To obtain the most recent revision of documentation related to the ML300, see http://www.xilinx.com/ml300.

Resources for EDK Users (Including New Users)

EDK Main Web Page

http://www.xilinx.com/ise/embedded/edk.htm

Getting Started with the EDK

http://www.xilinx.com/ise/embedded/edk_getstarted.pdf

Embedded System Tools Guide

http://www.xilinx.com/ise/embedded/est_guide.pdf

EDK Tutorials and Design Examples

http://www.xilinx.com/ise/embedded/edk_examples.htm

Embedded Processor Discussion Forum

http://toolbox.xilinx.com/cgi-bin/forum?14@@/Embedded%20Processors

Documentation Provided by Xilinx

Virtex-II Pro Advance Product Specification (Data Sheet)

http://www.xilinx.com/bvdocs/publications/ds083.pdf

Virtex-II Pro Platform FPGA User Guide

http://www.xilinx.com/bvdocs/userguides/ug012.pdf

RocketIO Transceiver User Guide
http://www.xilinx.com/publications/products/v2pro/ug_pdf/ug024.pdf

IBM ® CoreConnect™ Documentation

The Embedded Development Kit integrates with the IBM CoreConnect Toolkit. This toolkit is not included with the EDK, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the EDK. To obtain the toolkit, you must be a licensee of the IBM

CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm. Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit. If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at http://www.ibm.com/chips/products/coreconnect.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

IBM CoreConnect Bus Architecture Specifications

IBM CoreConnect Processor Local Bus (PLB) Architecture Specification
see **$CORECONNECT/published/corecon/64bitPlbBus.pdf**

IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification
see **$CORECONNECT/published/corecon/OpbBus.pdf**

IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification
see **$CORECONNECT/published/corecon/DcrBus.pdf**

IBM CoreConnect Toolkit Documentation

PLB Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/PlbToolkit.pdf**

OPB Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/OpbToolkit.pdf**

DCR Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/DcrToolkit.pdf**

CoreConnect Test Generator - User's Manual
see **$CORECONNECT/published/corecon/ctg.pdf**

Note: $CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

XILINX®

Chapter 2

ML300 Embedded PPC405 Reference System

Introduction

The ML300 Embedded PPC405 Reference System is an example of a large Virtex-II Pro™ based system. An IBM Core Connect™ infrastructure connects the CPU to numerous peripherals using Processor Local Bus (PLB), On-Chip Peripheral Bus (OPB), and Device Control Register (DCR) buses to build a complete system. This document describes the contents of the reference system and provides information about how the system is organized and implemented. It also discusses verification methodologies including software-driven and bus-model-driven simulations. A complete design cycle incorporating simulation, synthesis, FPGA implementation, and download is described. The information presented introduces many aspects of the ML300 Embedded PPC405 Reference System, but the user should refer to additional specific documentation for more detailed information about the software, tools, peripherals, interface protocols, and capabilities of the FPGA.

Hardware

Overview

Figure 2-1, page 20 provides a high-level view of the hardware contents of the Embedded PPC405 System. This design demonstrates a system that uses PLB, OPB, and DCR devices. The PLB protocol generally supports higher bandwidths, so the memory devices are placed there. The OPB connects the peripheral devices to the CPU by way of a PLB-to-OPB Bridge and primarily is intended for lower performance devices. The OPB offers a less complex protocol relative to the PLB, making it easier to design peripherals that do not require high performance. The OPB also has the advantage that it can support a greater number of devices, and it acts to decouple the peripherals from the higher-speed memory devices. DCR is used with control and status registers for simplicity when performance is not important. Refer to the PLB, OPB, and DCR CoreConnect Architecture Specifications for more information. The hardware devices used in this design are described in more detail in the Processor IP User Guide (<*EDK Install Directory*>/doc/proc_ip_ref_guide.pdf) and in Chapter 4, "Introduction to Hardware Reference IP".

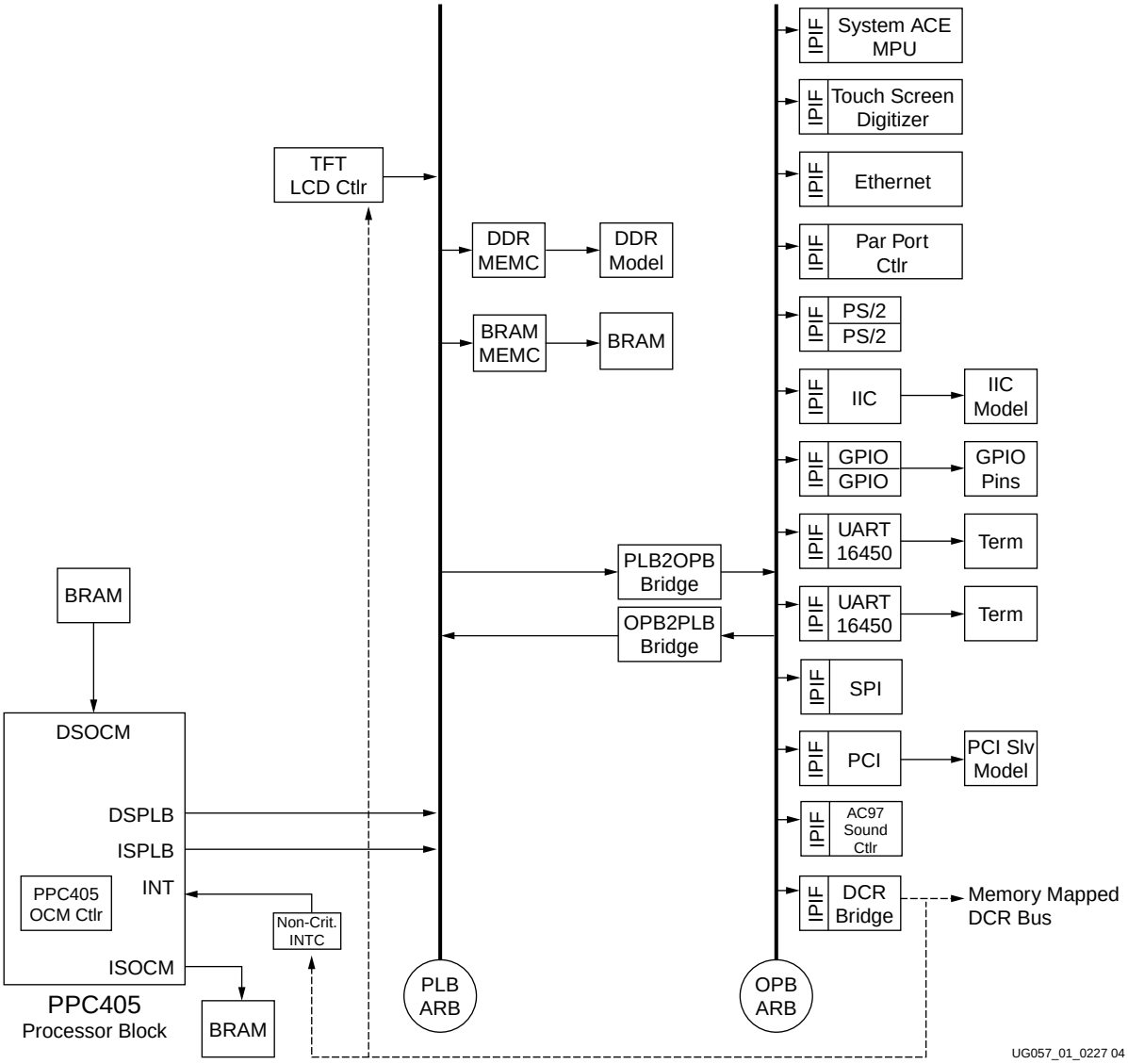


Figure 2-1: **High-Level Hardware View of ML300 Embedded PPC405 Reference System**

Processor Local Bus (PLB)

The PLB connects the CPU to high-performance devices, such as memory controllers. The PLB protocol supports higher bandwidth transactions and has a feature set that better supports memory operations from the CPU. Highlights of the PLB protocol include synchronous architecture, independent read/write data paths, and split transaction address/data buses. The reference design includes a 64-bit PLB infrastructure with 64-bit master and slave devices attached.

The PLB devices in the reference system include:

- PLB Masters
  - 640x480 VGA TFT LCD Controller
  - CPU Instruction-Side PLB Interface

- CPU Data-Side PLB Interface
- OPB-to-PLB Bridge

- PLB Slaves
  - BRAM Controller
  - Double Data Rate (DDR) SDRAM Controller
  - PLB-to-OPB Bridge-Out
- PLB Arbiter
  - 64-bit Xilinx PLB Arbiter

In general, all PLB devices are optimized around the Virtex-II Pro architecture and make use of pipelining to improve maximum clock frequencies and reduce logic utilization. Refer to the accompanying documentation for each device for more information about its design.

On-Chip Peripheral Bus (OPB)

The OPB connects lower-performance peripheral devices to the system. The OPB has a less complex architecture, which simplifies peripheral development. A PLB-to-OPB Bridge translates PLB transactions into OPB transactions, allowing the CPU to access the OPB devices. OPB devices can also access PLB devices by way of an OPB-to-PLB Bridge.

The OPB devices in the reference system include:

- OPB Masters
  - PLB-to-OPB Bridge-Out
  - Ethernet Controller (DMA Engine, if enabled)
  - OPB PCI Bridge
- OPB Slaves
  - IIC Controller
  - Dual 32-Bit General-Purpose Input/Output (GPIO)
  - 16450 UART #1
  - 16450 UART #2
  - PCI Bus Master
  - AC97 Sound Controller
  - OPB-to-DCR Bridge
  - Ethernet Controller
  - Dual PS/2 Controller
  - SPI Controller
  - Touch Screen Digitizer
  - Parallel Port
  - System ACE MPU Interface
  - OPB-to-PLB Bridge-In
- OPB Arbiter

In general, all OPB devices are optimized around the Virtex-II Pro architecture and make use of pipelining to improve maximum clock frequencies and reduce logic utilization.

Refer to the accompanying documentation for each device for more information about its design.

The OPB devices in the reference design make use of Intellectual Property InterFace (IPIF) modules to further simplify IP development. The IPIF converts the OPB protocol into common interfaces, such as an SRAM protocol or a control register interface. IPIF modules also provide support for DMA and interrupt functionality. IPIF modules simplify software development since the IPIF framework has many common features. Refer to Chapter 6, "IPIF Specification" for more information.

Note that the IPIF is designed mainly to support a wide variety of common interfaces, but may not be the optimal solution in all cases. Where additional performance or functionality is required, the user can develop a custom OPB interface. The IPIF protocols can also be extended to support other bus standards, such as PLB. This allows the backend interface to the IP to remain the same while the bus interface logic in the IPIF is changed. This provides an efficient means for supporting different bus standards with the same IP device.

The OPB specification supports masters and slaves of up to 64 bits with a *dynamic bus sizing* capability that allows OPB masters and slaves of different sizes to communicate with each other. The ML300 Embedded PPC405 Reference System uses a subset of the OPB specification which only supports 32-bit byte enable masters and slaves. Legacy devices utilizing 8- or 16-bit interfaces or those that require dynamic bus sizing functionality are not directly supported. It is recommended that all new OPB peripherals support byte enable operations for better performance and reduced logic utilization.

Device Control Registers (DCR)

The DCR offers a very simple interface protocol and is used for accessing control and status registers in various devices. It allows for register access to various devices without loading down the OPB and PLB interfaces. Since DCR devices are generally accessed infrequently and do not have high-performance requirements, they are used throughout the reference design for functions, such as error status registers, interrupt controllers, and device initialization logic.

The CPU contains a DCR master interface that is accessed through special *Move To DCR* and *Move From DCR* instructions. Some users may prefer to see DCR registers memory mapped so an alternative DCR Master is provided. An OPB-to-DCR Bridge can be instantiated to locate the 4-KB DCR space within the general system memory space. The reference design demonstrates both methods for accessing DCR.

The DCR slave devices connected to the CPU's DCR port include:

- Data-Side OCM (8 KB)
- Instruction-Side OCM (4 KB)

The DCR slave devices connected to the OPB-to-DCR Bridge include:

- Non-Critical Interrupt Controller
- PLB Arbiter *
- PLB-to-OPB Bridge *
- OPB-to-PLB Bridge*
- VGA TFT LCD Controller

  * The DCR connections to these devices are disabled to make the system more compact. The DCR ports can be enabled via the EDK GUI or by editing the system description file, **system.mhs**.

The DCR specification requires that the DCR master and slave clocks be synchronous to each other and related in frequency by an integer multiple. It is important to be aware of the clock domains of each of the DCR devices to ensure proper functionality.

Other Devices

In addition to the PLB, OPB, and DCR devices, the ML300 Embedded PPC405 Reference System contains Instruction-Side and Data-Side OCM modules. The OCM consists of BRAMs directly connected to the CPU. They allow the CPU fast access to memory and are useful for providing instructions or data directly to the CPU, bypassing the cache. This can prevent thrashing of caches to better process packet data or execute interrupt service routines. Refer to the OCM documentation for information about applications and design information.

Interrupts

The CPU also contains two interrupt pins, one for critical interrupt requests and the other for non-critical interrupts. An interrupt controller for non-critical interrupts is controlled through the DCR. It allows multiple edge or level sensitive interrupts from peripherals to be OR'ed together back to the CPU. The ability for bitwise masking of individual interrupts is also provided. Table 2-1 summarizes the connections from the IP to the interrupt controller.

Table 2-1: **List of IP Connections to the Interrupt Controller**

Interrupt Source
UART 16450 #1
UART 16450 #2
IIC Controller
Ethernet Controller
PS/2 Port #1
PS/2 Port #2
Touch Screen Digitizer
SPI
PCI (INTR A,B,C,D OR'd together)
SYSACE MPU
AC97 Sound Controller (Play Buffer)
AC97 Sound Controller (Record Buffer)
IIC (general wired-OR interrupt line)
IIC Temperature Sensor Alarm
External Ethernet PHY Chip
OPB to PLB Bridge Error
PLB Arbiter Error
PLB to OPB Bridge Error

Clock/Reset Distribution

Virtex-II Pro FPGAs have abundant clock management and global clock buffer resources. To demonstrate some of these capabilities, the ML300 Embedded PPC405 Reference System uses a variety of different clocks. Figure 2-2 illustrates use of the digital clock managers (DCMs) for generating the main clocks in the design. A 100 MHz input reference clock is used to generate the main 100 MHz PLB, OPB, and OCM clocks. The CLK90 output of the DCM produces a 100 MHz clock that is phase shifted by 90 degrees for use by the DDR SDRAM controller. The main 100 MHz clock is divided down by three to create a 33 MHz PCI clock. The CPU clock is multiplied up from the PLB clock to 300 MHz. A second DCM is used to recover and deskew the external clock from the DDR SDRAM. The second DCM also drives the 12.5 MHz clock to the VGA TFT controller. A third DCM (not shown) is used to deskew the externally driven PCI clock with the internal PCI clock.

Since each clock is referenced from the same 100 MHz clock, they are all phase aligned to each other. This synchronous phase alignment is required by the CPU and many other devices so they can pass signals from one clock domain to another. The CPU clock can run at any integer multiple of the PLB clock up to the maximum CPU clock frequency. During reset, internal clock synchronizers in the CPU detect the phase alignment of the PLB and CPU clocks and adjust for it automatically. The OCM clock must be divided down from the CPU clock by an integer multiple (up to eight), and the two clocks must be synchronous to each other.

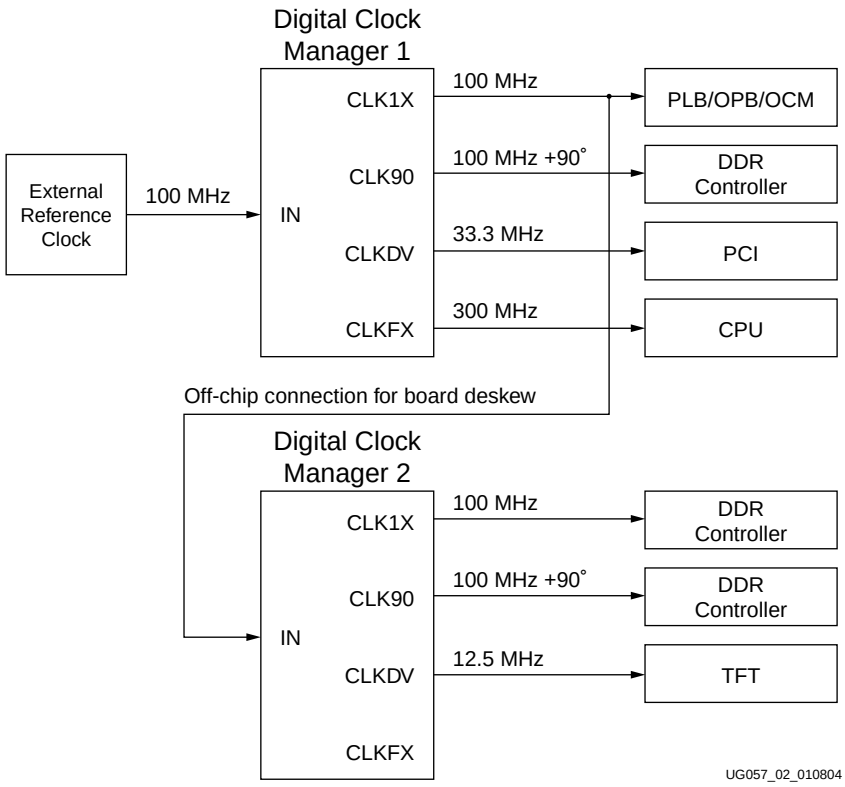


Figure 2-2: **Clock Generation**

After a system reset or at FPGA startup, a debounce circuit inside the Processor System Reset IP Module holds the FPGA in reset until the DCM has locked onto its reference clock. Once the DCM is locked and the clocks remain stable for several cycles (may take several microseconds in simulation), the reset condition is released to allow the system logic to

begin operating. (For example, the CPU will begin fetching instructions a few cycles after reset is released.) Since the reset net is a high-fanout signal, it may not be able to reach all the logic in the design within one clock cycle. User IP blocks should be designed to take into account the possible skew in the global reset and still start up properly. Alternatively, the global reset can be registered locally in each IP block to generate a synchronous reset signal.

The design implements the three levels of reset supported by the PPC405:

- Core reset
- Chip reset
- System reset

The core reset only affects the processor while the chip reset clears all the logic on the FPGA. The system reset is designed to reset the entire system including the FPGA and external devices connected to the FPGA. The CPU provides an internal special-purpose register that allows software to request that one of the three resets be performed.

The reset logic in the ML300 Embedded PPC405 Reference System is an example implementation of the PPC405 reset architecture. Designers should set the scope, boundaries, and effects of resets as appropriate to their designs. For more information, refer to the Processor System Reset Module documentation in the *EDK Processor IP User Guide*.

CPU Debug via JTAG

The CPU in the ML300 Embedded PPC405 Reference System can be debugged via JTAG with a variety of software development tools from VxWorks, GNU, IBM and others. The JTAG logic connected to the CPU in addition to the circuitry on the ML300 board offers two different types of JTAG chains for connecting to the CPU. This permits the widest compatibility among JTAG products that support the PPC405.

The preferred method of communicating with the CPU via JTAG is to combine the CPU JTAG chain with the FPGA's main JTAG chain which is also used to download bitstreams. This method requires the user to instantiate a JTAGPPC component from the library of Xilinx FPGA primitives and directly connect it to the CPU in the user's design. The primary advantage of sharing the same JTAG chain for CPU debug and FPGA programming is that this simplifies the number of cables needed since a single JTAG cable (like the Xilinx Parallel IV Cable) can be used for bitstream download as well as CPU software debugging.

An alternate method of using JTAG with the CPU is to directly connect the CPU's JTAG pins to the FPGA's user I/O. In this case the CPU is on a separate JTAG chain from the FPGA. This method requires two separate JTAG cables be used but is more compatible with third party JTAG tools which cannot perform the necessary JTAG commands to support a single combined JTAG chain with multiple devices on it.

The ML300 Embedded PPC405 Reference System contains a simple autosensing circuit to multiplex between the two types of JTAG chains. The JTAG circuit is normally in the state where it connects the CPU to the JTAGPPC component for a single combined JTAG chain. The design then senses the TCK pin on the CPU-only JTAG port. This pin is normally held high with a pull-up. If the TCK pin is ever pulled low (by an external JTAG programmer connected to this port) it will switch over the CPU JTAG pins to the other JTAG port. Any internal reset condition will return the JTAG multiplexer back to the default state. This circuit should be used for evaluation only. It should be replaced with a fixed circuit after the desired method of using JTAG has been determined. This autosensing circuit is not as

reliable as a fixed circuit since small glitches on TCK may cause a false detection. In addition, the JTAG switching circuit may prevent System ACE (described later) from functioning correctly because System ACE relies on using the combined JTAG chain to talk to the CPU. If System ACE is being used with the autosensing circuit present, any external JTAG programmer should not be connected to the CPU-only JTAG port until after System ACE download is complete.

IP Version and Source

summarizes the list of IP cores making up the ML300 Embedded PPC405 Reference System. The table shows the hardware version number of each IP core used in the design. The table also lists whether the source of the IP is from the EDK installation or whether it is reference IP in the local **pcores** directory.

Table 2-2: **IP Cores in the ML300 Embedded PPC405 Reference System**

Hardware IP	Version	Source
bram_block	1.00.a	EDK Installation
clocks	1.00.c	Local “pcores” Directory
dcr_intc	1.00.b	EDK Installation
misc_logic	1.00.a	Local “pcores” Directory
my_jtag_logic	1.00.a	Local “pcores” Directory
opb_gpio	2.00.a	EDK Installation
opb2dcr_bridge	1.00.a	EDK Installation
opb_iic	1.01.b	EDK Installation
opb_sysace	1.00.b	EDK Installation
opb_uart16550	1.00.c	EDK Installation
plb2opb_bridge	1.00.b	EDK Installation
opb2plb_bridge_ref	1.00.a	Local “pcores” Directory
plb_bram_if_cntlr	1.00.a	EDK Installation
plb_ddr	1.00.c	EDK Installation
ppc405	2.00.c	EDK Installation
dsocm_v10	1.00.b	EDK Installation
dsbram_if_cntlr	2.00.a	EDK Installation
bram_block	1.00.a	EDK Installation
isocm_v10	1.00.b	EDK Installation
isbram_if_cntlr	2.00.a	EDK Installation
ppc_trace	1.00.a	Local “pcores” Directory
proc_sys_reset	1.00.a	EDK Installation
opb_ethernet	1.00.m	EDK Installation

Table 2-2: **IP Cores in the ML300 Embedded PPC405 Reference System**

Hardware IP	Version	Source
opb_spi	1.00.b	EDK Installation
opb_ps2_dual_ref	1.00.a	Local “pcores” Directory
plb_tft_cntlr_ref	1.00.b	Local “pcores” Directory
opb_tsd_ref	1.00.a	Local “pcores” Directory
opb_ac97_controller_ref	1.00.a	Local “pcores” Directory
opb_par_port_ref	1.00.a	Local “pcores” Directory
opb_pci_ref	1.00.b	Local “pcores” Directory
pci_arbiter	1.00.a	Local “pcores” Directory
dcr_v29	1.00.a	EDK Installation
opb_v20	1.10.b	EDK Installation
plb_v34	1.01.a	EDK Installation

Simulation and Verification

Simulation Overview

Figure 2-3, page 28 diagrams the organization of the higher-level HDL files that comprise the system and testbench environment.

Note: The simulation testbench is available in Verilog only.

For simulation, the main testbench module (**testbench.v**) instantiates the FPGA (**system.v**) as the device under test and includes behavioral models for the FPGA to interact with. In addition to behavioral models for memory devices, clock oscillators, and external peripherals, the testbench also instantiates the CoreConnect bus monitors to observe the PLB, OPB, and DCR buses for protocol violations. The testbench can also preload some of the memories in the system for purposes such as loading software for the CPU to execute. The **sim_params.v** file is designed to be modified by the user to customize various simulation options. These options include message display options, maximum simulation time, and clock frequency. The user should edit this file to reflect personal simulation preferences.

Some of the testbench code is used to access signals internal to the design using hierarchy path names to reach into the design without changing any of the port interfaces. It is important that the design source files used for simulation match the source files for synthesis. Therefore, port interfaces should not be different or else inconsistencies can result.

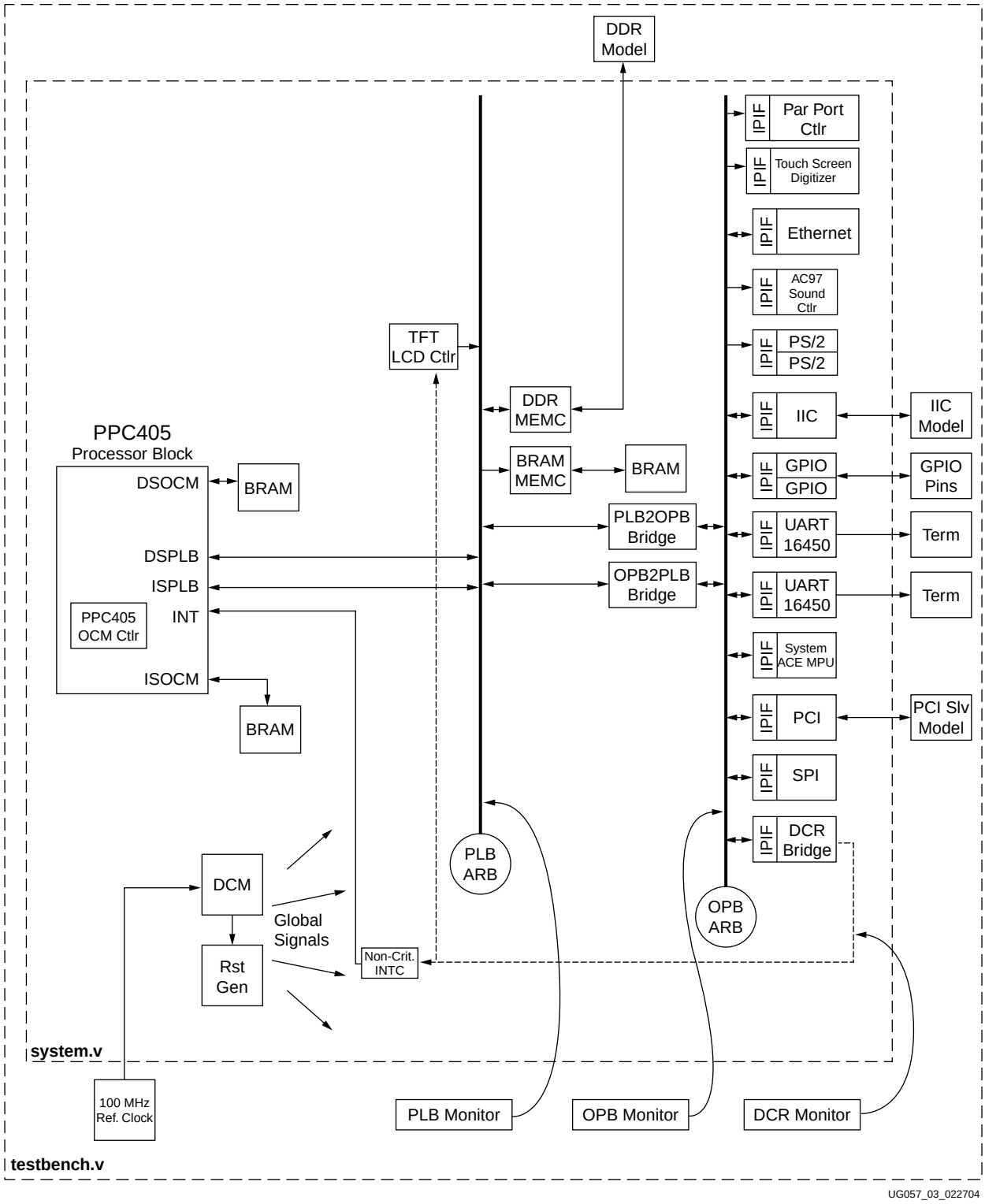


Figure 2-3: **Organization of Higher-Level HDL Files**

SWIFT and BFM CPU Models

The ML300 Embedded PPC405 Reference System demonstrates two different simulation methods to help verify designs using the PPC405 CPU. One method uses a full simulation model of the CPU based on the actual silicon. The second method employs bus functional models (BFMs) to generate processor bus cycles from a command scripting language. These two methods offer different trade-offs between behavior in real hardware, ease of generating bus cycles, and the amount of real time to simulate a given clock cycle.

A SWIFT model can be used to simulate the CPU executing software instructions. In this scenario, the executable binary images of the software are preloaded into memory from which the CPU can boot up and run the code. Though this is a relatively slow way to exercise the design, it more accurately reflects the actual behavior of the system.

The SWIFT model is most useful for helping to bring up software and for correlating behavior in real hardware with simulation results. The ML300 Embedded PPC405 Reference System demonstrates the SWIFT model simulation flow, by allowing the user to write a C program that is compiled into an executable binary file. This executable (in ELF format) is then converted into BRAM initialization commands using a tool called Data2MEM. (Note that Data2MEM can also generate memory files for the Verilog command **readmemh** to use to initialize external DDR memory.)

When a simulation begins and reset is released, the CPU SWIFT model fetches the instructions from BRAM (which is mapped to the boot vector) and begins running the program. The user can then observe the bus cycles generated by the CPU or any other signal in the design. For debugging purposes, the values of the CPU's internal program counter, general-purpose registers, and special-purpose registers are available for display during simulation.

Generating a desired sequence of bus operations from the CPU may require a lot of software setup or simulation time. For early hardware bring-up or IP development, a bus functional model can be used to speed up simulation cycles and avoid having to write software. A model of the CPU is available in which two PLB master BFMs and one DCR BFM are instantiated to drive the CPU's PLB/DCR ports. These BFMs are provided in the CoreConnect toolkits and allow the user to generate bus operations by writing a script written in the Bus Functional Language (BFL). The ML300 Embedded PPC405 Reference System provides a sample BFL script that exercises many of the peripherals in the system. Refer to the *CoreConnect Toolkit* documentation for more information.

Since the CPU SWIFT model and BFM model both have the same set of port interfaces, users can switch between the two simulation methods by compiling the appropriate set of files without having to modify the system's design source files. Users may, however, need to modify their testbenches to take into account which model is being used.

Behavioral Models

The ML300 Embedded PPC405 Reference System includes some behavioral models to help exercise the devices and peripherals in the FPGA. Many of these models are freely available from various manufacturers and include interface protocol-checking features. The behavioral models and features included in the reference design are:

- DDR memory models for testing the memory controllers:
  - These models can also be preloaded with data for simulations
- EEPROM model with IIC interface
- Pull-ups connected to the GPIO for reading and driving outputs without getting unknown values

- Terminal interface connected to the UARTs for sending and receiving serial data
  - The terminal allows a user to interact with the simulation in real time
  - Characters sent out by the UARTs are displayed on a terminal while characters typed into the terminal program are serialized and sent to the UARTs
  - A simple file I/O mechanism passes data between the hardware simulator and the terminal program
- Simple PCI Slave that responds to commands from the PCI Master in the reference design
  - The PCI Slave acts as a memory device that the PCI Master can write to and read back from
  - The PCI Slave responds to configuration, memory, and I/O PCI cycles

Synthesis and Implementation

The ML300 Embedded PPC405 Reference System can be synthesized and placed/routed into a Virtex-II Pro FPGA under the EDK tools. A basic set of timing constraints for the design is provided to allow the design to go through place and route.

Design Flow Environment

The EDK provides an environment to help manage the design flow for the ML300 Embedded PPC405 Reference System including simulation, synthesis, implementation, and software compilation. EDK offers a GUI or command line interface to run these tools as part of the design flow. Consult the EDK documentation for more information.

Memory Map

This section diagrams the system memory map for the ML300 Embedded PPC405 Reference System. It also documents the location of the DCR devices as mapped by the OPB to DCR Bridge. The memory map reflects the default location of the system devices as defined in the **system.mhs** file.

See Table 2-3 and Table 2-4, page 31.

Table 2-3: **CPU-Connected DCR Device Memory Map**

Device	Address Boundaries		Size
	Upper	Lower	
Data Side OCM Controller	203	200	32B
Instruction Side OCM Controller	103	100	32B

Table 2-4: **Memory Maps**

PLB Device Memory Map

Device	Address Max	Address Min	Size	Comment
DDR SDRAM	07FFFFFF	00000000	128 MB	
DDR SDRAM Shadow Memory	0FFFFFFF	08000000	128 MB	Shadow memory allows TFT video memory to be accessed as an uncached region. Video memory region starts at 0xFE000000.
Data Side OCM Space	40001FFF	40000000	8 KB	
Instruction Side OCM Space	50000FFF	50000000	4 KB	
PLB to OPB Bridge Space	DFFFFFFF	20000000	3.5 GB	
PLB BRAM Space	FFFF8000	FFFF0000	32 KB	Contains Boot Vector

OPB Device Memory Map

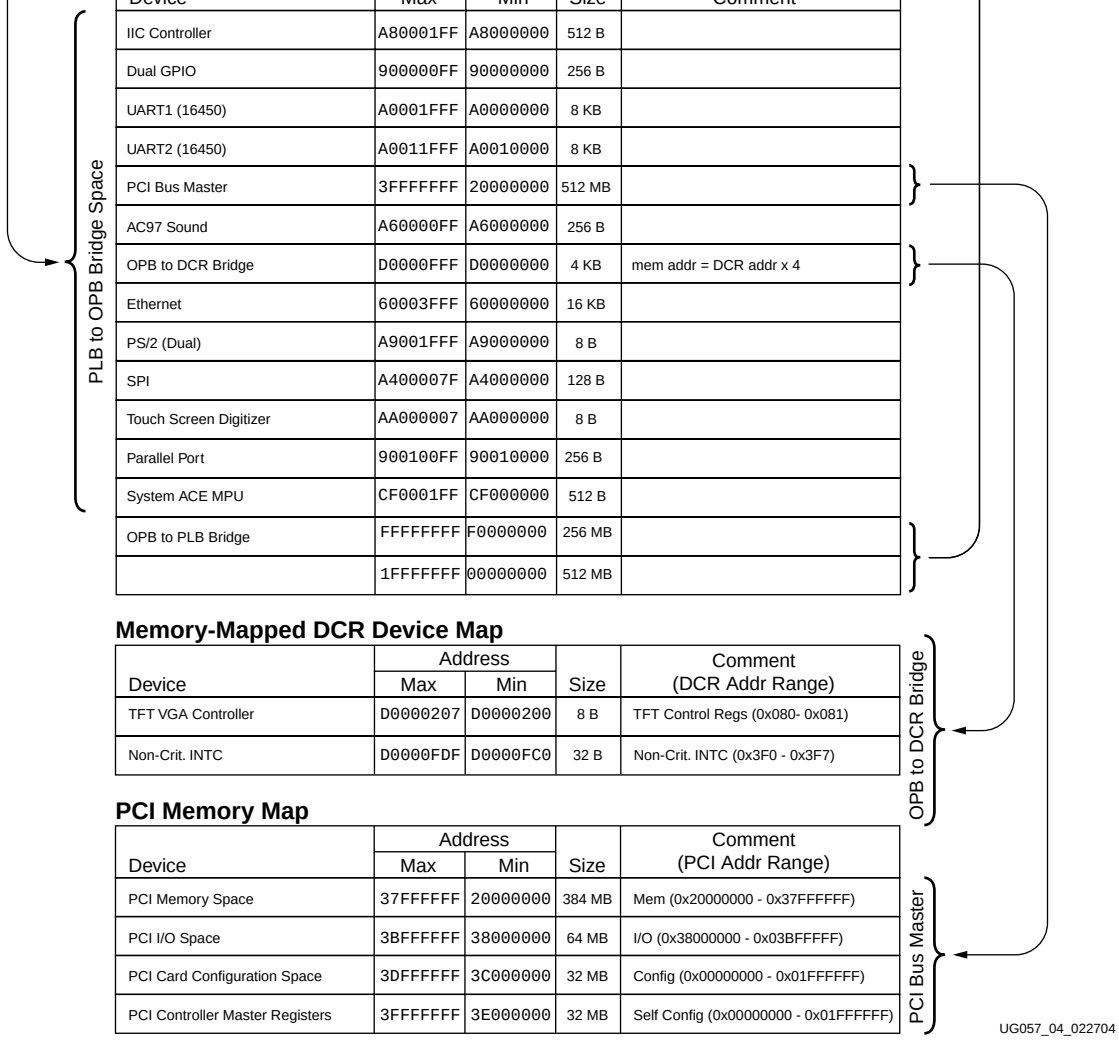
Device	Address Max	Address Min	Size	Comment
IIC Controller	A80001FF	A8000000	512 B	
Dual GPIO	900000FF	90000000	256 B	
UART1 (16450)	A0001FFF	A0000000	8 KB	
UART2 (16450)	A0011FFF	A0010000	8 KB	
PCI Bus Master	3FFFFFFF	20000000	512 MB	
AC97 Sound	A60000FF	A6000000	256 B	
OPB to DCR Bridge	D0000FFF	D0000000	4 KB	mem addr = DCR addr x 4
Ethernet	60003FFF	60000000	16 KB	
PS/2 (Dual)	A9001FFF	A9000000	8 B	
SPI	A400007F	A4000000	128 B	
Touch Screen Digitizer	AA000007	AA000000	8 B	
Parallel Port	900100FF	90010000	256 B	
System ACE MPU	CF0001FF	CF000000	512 B	
OPB to PLB Bridge	FFFFFFFF	F0000000	256 MB	
	1FFFFFFF	00000000	512 MB	

Memory-Mapped DCR Device Map

Device	Address Max	Address Min	Size	Comment (DCR Addr Range)
TFT VGA Controller	D0000207	D0000200	8 B	TFT Control Regs (0x080- 0x081)
Non-Crit. INTC	D0000FDF	D0000FC0	32 B	Non-Crit. INTC (0x3F0 - 0x3F7)

PCI Memory Map

Device	Address Max	Address Min	Size	Comment (PCI Addr Range)
PCI Memory Space	37FFFFFF	20000000	384 MB	Mem (0x20000000 - 0x37FFFFFF)
PCI I/O Space	3BFFFFFF	38000000	64 MB	I/O (0x38000000 - 0x03BFFFFF)
PCI Card Configuration Space	3DFFFFFF	3C000000	32 MB	Config (0x00000000 - 0x01FFFFFF)
PCI Controller Master Registers	3FFFFFFF	3E000000	32 MB	Self Config (0x00000000 - 0x01FFFFFF)



ML300 Specific Registers

The design also contains a number of register bits to control various items on the ML300 such as the buttons and LEDs. Note that the 32-bit GPIO pins on the ML300 are controlled with a standard set of GPIO registers at 0x90000008. See the EDK *Processor IP User Guide* documentation for more information about the GPIO. Table 2-5 and Table 2-6 contain information about control and status registers specific to the ML300 Embedded PPC405 Reference System.

Table 2-5: **Game Button/LED Register Map (Address 0x90000000)**

Bits	Description
0 (MSB)	Reserved
1	Left GAME Switch LFT
2	Left GAME Switch TOP
3	Left GAME Switch RT
4	Left GAME Switch BOT
5	Left TOP Switch
6	Left MID Switch
7	Left BOT Switch
8	Reserved
9	Right GAME Switch LFT
10	Right GAME Switch TOP
11	Right GAME Switch RT
12	Right GAME Switch BOT
13	Right TOP Switch
14	Right MID Switch
15	Right BOT Switch
[16:31] (LSB)	Game LEDs bits [15:0]. These LEDs are labeled on the ML300 board. Power on default value is 0.

Note: Status bits for buttons are read-only. A "1" value indicates the button was pushed. "0" indicates button was not pushed. LED control bits are read/write where a "1" value turns on the LED and a "0" turns off the LED.

Table 2-6: **Control Register Map (Address 0x90000004)**

Bits	Description
0 (MSB)	PLB error light clear. Writing a "1" to this bit will clear the PLB error light on the ML300 board. This bit must be written back to "0" to enable the PLB error light. Defaults to 1.
1	Unused

Table 2-6: **Control Register Map (Address 0x90000004) *(Continued)***

Bits	Description
2	IIC write protect bit. Setting this bit to "1" prevents the IIC devices with nonvolatile storage from being written to. A "0" allows writes to the EEPROM. This bit default to 0.
3	Caselight Enable. The illumination LEDs under the ML300 board are normally turned on when the system reset is inactive. Writing a "1" to this bit allows these LEDs to be on when reset is not present. A "0" turns off the LED. This bit defaults to 1.
4	OPB error light clear. Writing a "1" to this bit will clear the OPB error light on the ML300 board. This bit must be written back to "0" to enable the OPB error light. Defaults to 1.
[5:19]	Unused
[20:31] (LSB)	Software Powerdown. Writing the hex value 0x0FF as in "off" will cause the ML300 to power off. The 0x0FF value must be held for about 1-2 seconds before the board turns off. This register defaults to 0x00.

Extending or Modifying the Design

The ML300 Embedded PPC405 Reference System is a good starting point from which a user can add, remove, or modify components in the system. Since most of the IP in the design is attached to the CoreConnect infrastructure under EDK, adding or removing devices is a fairly straightforward process. Below is an overview for making various changes to the system.

Adding or Removing IP Cores

To remove an IP core:

- Delete the instantiation for that piece of IP from the **system.mhs** file (or use the **Add/Edit Cores** feature of the EDK GUI)
- Delete all corresponding external I/O ports from the **system.mhs** file
- Remove corresponding UCF file entries specifying timing or pinout locations for that IP
- Remove anything in the testbench connected to that IP and update the BFL scripts if necessary

To add an IP core:

- Instantiate the device by adding it to the **system.mhs** file (or use the **Add/Edit Cores** feature of the EDK GUI)
- Connect its external I/O to the top level
- Set its configuration parameters (i.e., base address) in the **system.mhs** file (or use the **Add/Edit Cores** feature of the EDK GUI)
- Add appropriate timing and pinout constraints to the UCF file
- Update the testbench to allow the new IP to be tested and update the BFL scripts if necessary

Other Modifications

Both the Instruction and Data Side OCM ports are connected to BRAMs. The user can change the memory base addresses, sizes, or clock frequencies. Refer to the OCM documentation for information about connecting BRAM memory to the OCM and configuring it.

Each Interrupt Controller supports up to 32 separate interrupts, but only a few of the interrupt inputs are used. New IP capable of generating interrupts should tie in their interrupt request lines to the interrupt controller so the CPU can see them.

Behavioral Models/Testbenches

Whenever new IP devices are added or new BFMs are added, the testbenches should be updated. This may include new device models or edits to the connections of some of the bus monitors. The sample BFL script supplied with the design can be used as a template for building custom test scripts.

Directory and File Listings

The files and directories specific to the ML300 Embedded PPC405 design are listed in the tables that follow. Note that the following tables only list files that are present after installation. After running simulation, synthesis, or place and route, additional files may be created. Directory path names are shown separated by the "/" character as is the UNIX convention. For Windows, the "\" character should be used to separate directory paths.

Table 2-7: **Directory and File Listings for ML300 Embedded PPC405 Design**

Directory/File	Description
projects/ml300_edk3/genace.tcl	Enhanced System ACE file generation script
projects/ml300_edk3/xmd.ini	XMD configuration files specifying OCM address ranges
projects/ml300_edk3/system.xmp	EDK system project file containing multiple software applications
projects/ml300_edk3/system_linux.xmp	EDK system project file utilizing Linux
projects/ml300_edk3/system_linux.mss	MSS file - Describes system software drivers under EDK (Linux specific)
projects/ml300_edk3/system.mhs	MHS file - Describes system hardware under EDK
projects/ml300_edk3/system.mss	MSS file - Describes system software drivers under EDK (For general software applications)
projects/ml300_edk3/bsp	Linux specific files
projects/ml300_edk3/data/AT24CXXX.v	IIC EEPROM model
projects/ml300_edk3/data/compile_corecon.f	Verilog –f file to compile CoreConnect modules
projects/ml300_edk3/data/compile_corecon_dummy.f	Verilog –f file to compile placeholder files if CoreConnect is not installed
projects/ml300_edk3/data/cygncurses6.dll	Used by terminal.exe
projects/ml300_edk3/data/cygwin1.dll	Used by terminal.exe
projects/ml300_edk3/data/edn_patch.do	ModelSim script to load pre-generated simulation netlists if needed
projects/ml300_edk3/data/gen_memfiles.sh	Script file to create MEM files for external memory initialization
projects/ml300_edk3/data/init_patch.sh	Script file to patch system_init.v simulation file to use simulation hierarchy

Table 2-7: **Directory and File Listings for ML300 Embedded PPC405 Design *(Continued)***

Directory/File	Description
projects/ml300_edk3/data/memory_init.bmm	BMM file for design, including all external memory
projects/ml300_edk3/data/mt46v32m8.v	DDR memory model
projects/ml300_edk3/data/opb_dcl.inc	OPB bus monitor configuration file
projects/ml300_edk3/data/pci_lc_i.v	Simulation model of PCI Core
projects/ml300_edk3/data/pci_targ32.v	Simulation model of PCI target device
projects/ml300_edk3/data/ppc405_0_wrapper.v	Simulation wrapper for PPC405
projects/ml300_edk3/data/sim_params.v	Simulation parameters
projects/ml300_edk3/data/start_terminal.bat	Batch file to create UART simulation terminal
projects/ml300_edk3/data/start_terminal.sh	Script file to create UART simulation terminal
projects/ml300_edk3/data/system.ucf	UCF of system
projects/ml300_edk3/data/terminal	Simulation Terminal executable (Solaris)
projects/ml300_edk3/data/terminal.exe	Simulation Terminal executable (PC)
projects/ml300_edk3/data/testbench.do	Main simulation .do file for Modelsim
projects/ml300_edk3/data/testbench.v	Main simulation testbench
projects/ml300_edk3/data/uart_rcvr.v	Simple UART receiver / decoder load
projects/ml300_edk3/data/use_ppc405_bfm.do	Used with BFM simulation to override SWIFT model
projects/ml300_edk3/data/view_reg.v	Generate PPC405 register messages
projects/ml300_edk3/data/wave.do	ModelSim waveform .do file
projects/ml300_edk3/data/bfl/X_proc_block_BFM.v	BFM of PPC405 for use with back-annotated simulation
projects/ml300_edk3/data/bfl/proc_block_BFM.v	BFM of PPC405 for use with functional simulation
projects/ml300_edk3/data/bfl/run_BFC.sh	Script to invoke CoreConnect Bus Functional Compiler
projects/ml300_edk3/data/bfl/*.bfl	BFM Simulation scripts for various hardware IP cores
projects/ml300_edk3/data/bfl/test_system_place_holder.v	Placeholder file for simulations where the CoreConnect toolkit is not installed
projects/ml300_edk3/data/corecon_dummy/*.*	Placeholder file for simulations where the CoreConnect toolkit is not installed
projects/ml300_edk3/drivers	Software drivers corresponding to IP in local “pcores” directory
projects/ml300_edk3/etc/bitgen.ut	Bitgen options
projects/ml300_edk3/etc/download.cmd	IMPACT command file for downloading system.bit
projects/ml300_edk3/etc/fast_runtime.opt	XFLOW option file for the FPGA implementation tool settings
projects/ml300_edk3/pcores/clocks_v1_00_c	Clock module
projects/ml300_edk3/pcores/misc_logic_v1_00_a	Miscellaneous glue logic
projects/ml300_edk3/pcores/my_jtag_logic_v1_00_a	Custom JTAG logic
projects/ml300_edk3/pcores/opb2plb_bridge_ref_v1_00_a	OPB-to-PLB Bridge (Lite) Reference IP
projects/ml300_edk3/pcores/opb_ac97_controller_ref_v1_00_a	AC97 Sound CODEC Controller Reference IP
projects/ml300_edk3/pcores/opb_par_port_ref_v1_00_a	OPB Parallel Port Controller Reference IP
projects/ml300_edk3/pcores/opb_pci_ref_v1_00_b	OPB PCI Bridge (Lite) Reference IP

Table 2-7: **Directory and File Listings for ML300 Embedded PPC405 Design *(Continued)***

Directory/File	Description
projects/ml300_edk3/pcores/opb_ps2_dual_ref_v1_00_a	Dual PS/2 Controller Reference IP
projects/ml300_edk3/pcores/opb_tsd_ref_v1_00_a	Touch Screen Digitizer Reference IP
projects/ml300_edk3/pcores/pci_arbiter_v1_00_a	PCI Arbiter Reference IP
projects/ml300_edk3/pcores/plb_tft_cntlr_ref_v1_00_b	PLB TFT LCD Controller Reference IP
projects/ml300_edk3/pcores/ppc_trace_v1_00_a	Trace Logic
sw/standalone/lib	Additional shared libraries (not shipped with EDK)
sw/standalone/mapfiles	Map files - Linker Scripts
sw/standalone/*<various other directories>*	Various software applications - See Software Section of EDK Tutorial Chapter for more information

Chapter 3

EDK Tutorial and Demonstration

Introduction

This chapter contains basic instructions for using the EDK tools with the ML300 Embedded PPC405 Reference System. It is designed to help illustrate the steps to build, download, and simulate the design. Information about demonstration software applications will also be provided. The instructions that follow provide only an overview of the capabilities of EDK. Much more detail about operating the EDK tools can be found in the EDK documentation. This chapter assumes that the reference design and all other necessary tools are properly installed according to the provided installation instructions.

Instructions for Invoking the EDK tools

This tutorial section and those that follow have directory path names that are shown separated by the "/" character as is the UNIX convention. For Windows, the "\" should be used to separate directory paths. The instructions that follow reference the <*EDK Project Directory*> located at **<*Reference Design Install Directory*>/projects/ml300_edk3/**. This is the area where the EDK Xilinx Microprocessor Project files (.xmp) reside after installing the ML300 Embedded PPC405 Reference System.

Launching Xilinx Platform Studio (XPS)

1. Open the XPS GUI.

   On PC, click:

   Start → **Programs** →**Xilinx Platform Studio**

   On Solaris, source the necessary environment scripts, and launch XPS:

   ```
   $ xps
   ```

2. Open XPS project file for the ML300 Embedded PPC405 Reference System:

   Click **File** →**Open Project**

   Browse to find the <*EDK Project Directory*>

   Select the file **system.xmp**, Click **Open**

   This opens the project file under EDK. It is now ready to build, download, or simulate the system using the user-selected software application program. You are now ready to proceed to follow the remaining instructions below.

Instructions for Selecting Software Application

The **system.xmp** EDK project file supports multiple user software applications. To select which software application to compile or simulate, follow the instructions below.

1. Click the Applications tab on the left-hand pane, then scroll down and look for **Project: ppc405_0_hello_uart**.
2. Right-click on **Project: ppc405_0_hello_uart** and select **Make Project Active**.

Note: This tutorial uses the **hello_uart** application as an example. To select a different software application, right-click on the active project and select **Make Project Inactive**. Then find the software application of interest and make that project active. See "Software," page 43 for more information.

Instructions for Running Functional Simulations

The ML300 Embedded PPC405 Reference System comes with SWIFT and BFM based simulation examples. This section describes the necessary steps for running these simulations.

1. Check that the simulation settings are correct and that the libraries are pointing to the correct directories on your system (the project defaults are not likely to match that of your system). Under XPS, go to **Options→Project Options**. Click the "HDL and Simulation" tab to show the current ModelSim library paths. Change these paths if incorrect.

   Note: If you followed the recommended default library names and locations from the installation instructions, the path settings are:

   ```
   EDK Library = <EDK Install Dir>/mti_se/edklib
   Xilinx Library = <EDK Install Dir>/mti_se
   ```

   In the HDL box, ensure that Verilog is selected for the simulation to work properly. In the Simulation Models box, select either Behavioral or Structural. The Behavioral setting allows for HDL source code level simulation, but requires a mixed-language Verilog/VHDL ModelSim license. The structural setting simulates the design at the netlist level, but only requires a Verilog ModelSim license.

   When complete, the **Project Options→HDL and Simulation** settings should look similar to Figure 3-1, page 39.

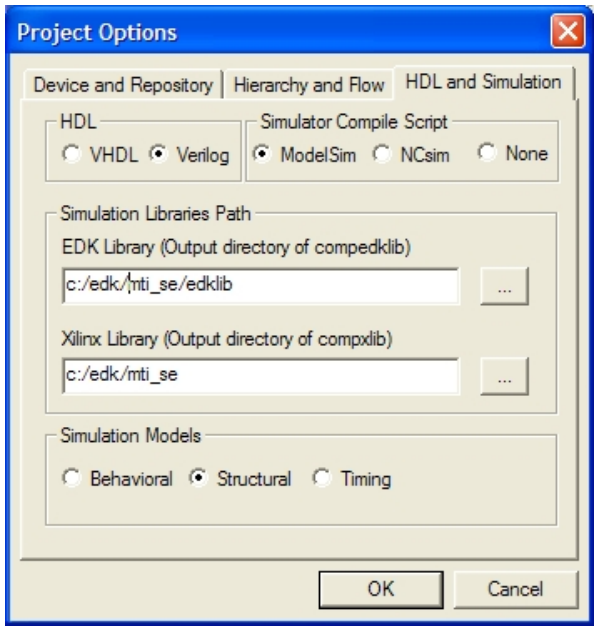


Figure 3-1: **HDL and Simulation Settings**

Click **OK** when finished.

2. Optional: Modify ***<EDK Project Directory>*/data/sim_params.v** to suit your simulation needs.
3. Speed up the UART baud rate for simulation.

   The UART is normally set to 9600 baud when running in real hardware. However, in simulation this is an excessively slow baud rate and would require prohibitive amounts of time to simulate the transmission of a single character. Therefore a user flag can be set so that software applications will compile in a baud rate of about 3 Mbaud to speed up the transfer rate of UART data. Setting this flag is recommended when simulating an application that uses a UART.

   Click the Applications tab on the left-hand pane, then double-click **Compiler Options** under the active project **ppc405_0_hello_uart**. Choose the Advanced tab in the "Set Compiler settings for..." window that pops up.

   In the top box for "Program Sources Compiler Options" change from the default of "-DSIM=0" to "-DSIM=1" for simulation.

   Note: Remember to set this back to "-DSIM=0" before generating bitstreams or running the software application in real hardware.

   Click **OK** when finished.

4. Run Simulation.

   Click **Tools→Hardware Simulation**

   XPS will compile the hardware and software files and invoke ModelSim. Note: EDK does not currently support behavioral simulation for Verilog designs. Therefore, structural simulation is used which requires the hardware IP simulation models to be generated from the post-synthesized netlist. It may take some time to complete this translation process.

5. Proceed to either Step 5a or 5b below to begin either a SWIFT or BFM simulation.

   5a. SWIFT simulation

   After ModelSim is up, load the design by entering the following command at the ModelSim console prompt:

   ```
   Modelsim> do ../../data/testbench.do
   ```

   This **testbench.do** file will perform a number of tasks:

   - Run data2mem to generate MEM files for external memory
   - Compile CoreConnect monitors (if installed)
   - Compile the testbench and peripheral simulation models
   - Invoke the UART terminal application

   Now start the simulation. At the ModelSim prompt, type:

   ```
   Modelsim> run -all
   ```

   For the hello_uart program, you can interact with the simulation using the terminal program. On the UART 1 terminal, the words "Hello world" followed by a carriage return will appear as the program executes in the simulation. You can also type into the terminal (after Hello world is displayed) and have the characters echoed back after some delay. Keep in mind that the program is running in simulation, so it may take many seconds of real time for the characters to get transmitted and received. Pressing the **Esc** (escape) key will exit the terminal and stop the simulation after a few seconds.

   Note: It is normal to see some warnings from the PLB monitor or behavioral memory models during reset, but the PLB/OPB/DCR monitors should not report any protocol errors during simulation (warnings and notes may occur depending on the circumstances). Some programs may run for relatively long periods of time or indefinitely. You can modify the **sim_params.v** file to stop the simulation after a desired amount of time or press **Ctrl c** (break) to stop the simulation.

   5b. BFM simulation

   After ModelSim is up, load the design by entering the following command at the ModelSim console prompt (Note: the CoreConnect toolkit must be installed to support BFM simulation):

   ```
   Modelsim> do ../../data/testbench.do bfm
   ```

   This testbench.do file will perform a number of tasks:

   - Run data2mem to generate MEM files for external memory
   - Compile CoreConnect monitors
   - Invoke Bus Functional Compiler (BFC) on Bus Functional Language (BFL) scripts
   - Compile the testbench and peripheral simulation models
   - Invoke the UART terminal application

   Now start the simulation. At the ModelSim prompt, type:

   ```
   Modelsim> run -all
   ```

   The BFL script performs a set of memory write/read tests to all memory devices in the system. It then sends out "Hello" over UART 1 and "world" over UART 2. The script finishes up by accessing the other peripherals in the system, including BRAM, PCI, IIC, and so on.

The simulation stops when the BFL script is finished executing or when an error occurs. If the simulation completes successfully, the simulator displays the following message:

```
Synch 31 received… Simulation Completed
```

If an error is detected, either in the form of a protocol violation reported by a bus monitor or a read comparison error, the simulation stops and an error message is displayed. It is a useful exercise to view the simulator's waveform display and correlate the commands in the **.bfl** script with the bus transaction waveforms over PLB, OPB, and DCR.

6. Close MTI.

   It may be necessary to close MTI after completing simulation in order to properly return control back to the XPS GUI.

Instructions for Building / Implementing Design

After successfully simulating the design, it can now be synthesized and place and routed to be run on real ML300 hardware.

1. Synthesize the design.

   In XPS, Click **Tools→Generate Netlist**.

   Note: This step may take some time to complete.

2. Implement the design.

   In XPS, Click **Generate Bitstream**.

   Note: This step may take some time to complete.

3. Restore software configuration to use normal baud rate.

   Software applications should be set back to 9600 baud in order to correctly send data via the UART.

   Go to **Options→Compiler Options→Others**.

   In the top box for "Program Sources Compiler Options" set it back to "-DSIM=0".

   Click **OK**.

Instructions for Downloading Design

The hardware bitstreams and software binary executable files can be downloaded to the ML300 board via the Xilinx Parallel Cable IV cable or System ACE.

The downloaded design will run the hello_uart program. To see this program running, connect a serial cable from a PC to the ML300 board. Use a terminal program like HyperTerminal which comes with Windows and set the COM port settings to 9600 baud, 8 Data Bits, No Parity, 1 Stop Bit, Hardware or No flow control. Once the program is downloaded using the instructions below, you should be able to see "Hello world" on your terminal. The board will then echo the characters you type until you hit escape.

Download Using Parallel Cable IV (iMPACT Program)

After the design is implemented, a bitstream can then be generated and downloaded into an FPGA using a program like Impact available with the Xilinx ISE tools. (A PC should be used for this step.)

1. Connect Parallel Cable IV from a PC to the ML300 board and power on the board.
2. Click **Tools→Download** within XPS.

   Note: This will load a bitstream that contains a bootloop program that effectively idles the PPC405 - NOT the software program that you have specified. You must continue on with the remaining steps in this section to load your program with GDB.
3. Click **Tools→XMD** to run the Xilinx Microprocessor Debug tool.
4. This opens an XMD command shell (which loads the configuration file ***<EDK Project Directory>/xmd.ini***).
5. Click **Tools→Software Debugger** to bring up GDB.
6. A menu window will open. Choose the software application project "ppc405_0_hello_uart" to download. Then click **OK**.

   Note: In this window, you must choose the software application that is marked as active under the Applications tab in the main left window pane. Do not choose an inactive software application.
7. Within GDB:

   Go to **Run** menu and select **Connect to Target**.

   For target, select XMD and make sure that Port is set to the same port number reported within the XMD shell (usually 1234) and click **OK**.

   Go to the Run menu and select **Download** - this will load your software and set the PC to the beginning.
8. Now you can set breakpoints and run as you wish. For example, the Control menu allows you to step through your code or run to completion (Finish).

Refer to the EDK documentation for further details.

Download Using System ACE

System ACE is a configuration management controller chip. It allows the user to store hardware and software information on a flash memory device and use it to program one or more devices via JTAG. The ML300 platform makes use of the System ACE chip in conjunction with standard compact flash cards to enable hardware and software programming of the FPGA. More information about System ACE is available from http://www.xilinx.com/systemace. EDK supports the generation of System ACE files to download bitstreams and software applications onto Virtex-II Pro FPGAs. This is accomplished by concatenating the JTAG commands to download the bitstream with the JTAG commands to download the software program. This combined set of JTAG commands is encoded into a **.ace** file that can be read from a compact flash card by the System ACE chip.

This download method creates an ACE file that contains the bitstream and software that can be saved to the Microdrive and inserted into ML300.

1. Within XPS, select **Tools→Generate System ACE File**.

   Note: This command uses the local script file *<EDK Project Directory>*/genace.tcl (overriding the EDK default script) to generate the ACE file *<EDK Project Directory>*/implementation/system.ace
2. Copy this file to your Microdrive or CompactFlash device.
   - If using a newly formatted Microdrive or CompactFlash device, copy it to the root directory.
   - If using the Microdrive that shipped with ML300, it is recommended that the ACE file be copied into the **XILINX\myace** directory of the Microdrive.

3. Insert the Microdrive or CompactFlash device into the ML300.
   - If the ACE file is in the root directory, it will be downloaded immediately
   - If the ACE file is in the **XILINX\myace** directory of the Microdrive, select "My OWN Ace File" on the bootloader touch screen menu.

Refer to the System ACE and EDK documentation for further details.

Software

Table 3-1 lists the software demo applications ported to the EDK design. The demo software is kept apart from the hardware design to make it reusable for other projects. The user selects the desired software project by selecting the Applications tab in the left-hand window pane and choosing which application to use.

Most software projects are linked with the built-in linker script. However, in certain cases specialized linker scripts are used. These linkers scripts are located in at ***<Reference Design Install Directory>*/sw/standalone/mapfiles**.

Mapfile5 is set up to run demo programs directly out of the processor caches.

Note: Software programs that utilize preloaded caches as a section of main memory cannot be simulated. These programs include hello_cache and xrom which can only be run in real hardware.

Table 3-1: **Software Applications**

Name	Description	Design Files
bootload	Displays graphical menu on TFT and loads appropriate ACE file based on user input on touch screen	sw/standalone/bootload, system.xmp
dispbmp	Displays a user-selectable bitmap image on the TFT	sw/standalone/dispbmp, system.xmp
hello	Using C's stdio library, prints "Hello world!" and echoes characters entered via standard input to standard output	sw/standalone/hello, system.xmp
hello_cache	Runs entirely out of caches, prints brief explanation of program and outputs characters entered using standard input to standard output	sw/standalone/hello_cache, system.xmp, sw/standalone/mapfiles/mapfile5
hello_gpio	Prints "Hello world!" to the GPIO data register, causing LEDs to blink	sw/standalone/hello_gpio, system.xmp
hello_pit	Prints "PIT" on TFT when PIT interrupt occurs and outputs characters entered via standard input to standard output until **ESC** key is pressed.	sw/standalone/hello_pit, system.xmp
hello_tft	Prints seven lines of ASCII characters (0x20 to 0x80) in seven color (foreground, background) combinations on the TFT	sw/standalone/hello_tft, system.xmp
hello_uart	Using the EDK UART driver, prints "Hello world!" on the UART and outputs characters entered via standard input to standard output	sw/standalone/hello_uart, system.xmp
iic_tft_brightness	Adjusts TFT screen brightness based on user input ('+' and '-' keys on standard input device). Value ranges from 0-255 (0 does not mean a completely blank TFT)	sw/standalone/iic_tft_brightness, system.xmp

Table 3-1: **Software Applications *(Continued)***

Name	Description	Design Files
linux	Generates BSP for Linux kernel	sw/standalone/linux, system_linux.xmp
ml300_set_eeprom	Stores user-specified MAC address into the IIC-EEPROM	sw/standalone/ml300_set_eeprom, system.xmp
ps2_scancodes_int	Interrupt-driven, reads keystrokes on a keyboard attached to PS/2 port 1 and displays corresponding PS/2 scancodes on standard output	sw/standalone/ps2_scancodes_int, system.xmp
ps2_scancodes_polled	Polled, reads keystrokes on a keyboard attached to PS/2 port 1 and displays corresponding PS/2 scancodes on standard output	sw/standalone/ps2_scancodes_polled, system.xmp
scalechar	Displays seven variations of "The quick brown fox jumps over the lazy dog 1234567890" in different sizes on the TFT	sw/standalone/scalechar, system.xmp
scan_pci	Initializes OPB PCI bridge and scans devices on PCI bus	sw/standalone/scan_pci, system.xmp
sysace_sector_ops_polled	Reads and writes data to a CompactFlash card or MicroDrive in the System ACE CF card slot	sw/standalone/sysace_sector_ops_polled, system.xmp
test_ac97	Records a buffer of sound from either the Line-In or Mic-In ports of the ML300 to the AC97 controller and plays it back through the Line-Out port using the AC97.	sw/standalone/test_ac97, system.xmp
test_spi	Sets up SPI, clears memory, writes and reads back sequence of bytes from 0x0 to 0xff eight times (addresses 0 to 2047)	sw/standalone/test_spi, system.xmp
touchcalibrate	Calculates display constants for the touch screen based on user input	sw/standalone/touchcalibrate, system.xmp
touchscreen_int	Interrupt-driven, detects touches to the touch screen and polls the raw coordinates and pressure variables before printing them to standard output	sw/standalone/touchscreen_int, system.xmp
touchscreen_polled	Polled, detects touches to the touch screen, prints raw coordinates and pressure variable to standard output	sw/standalone/touchscreen_polled, system.xmp
v2pdraw	Drawing program - polled (works better) and interrupt-driven versions	sw/standalone/v2pdraw, system.xmp
v2ptictactoe	Tic Tac Toe game	sw/standalone/v2ptictactoe, system.xmp
xrom	Tests memory on BRAM, DDR SRAM, LEDs, IIC	sw/standalone/xrom, system.xmp
xrom_ml300	Tests TFT, PS/2, touch screen + XROM tests	sw/standalone/xrom_ml300, system.xmp

Building the Software Demo Applications

1. Start XPS and load **system.xmp**.
2. Click the Applications tab on the left-hand pane, then scroll down and look for **Project: ppc405_0_hello_gpio**.
3. Right-click on **Project: ppc405_0_hello_gpio** and select **Make Project Active**. Make sure all other applications are inactive.
4. Implement the design using the steps outlined in "Instructions for Building / Implementing Design."
5. Select **Tools→Build All User Applications**.

The resulting ELF file is located in **ppc405_0/code/hello_gpio.elf**.

Building the Linux BSP

The EDK design comes with MLD/TCL technology to generate a Linux BSP for ML300. The MLD and TCL files are located in **sw_services**. To build a BSP for the Linux kernel proceed as follows:

1. Start XPS in command line mode and load the Linux XMP:

   ```
   $ xps -nw system_linux.xmp
   ```

2. Generate the Linux BSP from within XPS:

   ```
   XPS% run libs
   ```

The resulting Linux BSP is located in **ppc405_0/libsrc/linux_v1_00_a/linux**. Copy the whole sub-tree into the Linux kernel before configuration and compilation. The Linux kernel and the tools to build the Linux kernel are available from MontaVista (http://www.mvista.com).

Chapter 4

Introduction to Hardware Reference IP

Introduction

The ML300 Embedded PPC405 Reference System contains additional hardware IP beyond what may be shipped with the EDK tool suite. This hardware IP supports some of the features present on the ML300 board. The IP and its source code is provided as a reference example to illustrate how hardware can be designed to interface with the Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register (DCR) bus. Each of these is documented in greater detail in the chapters that follow. Generally, the interface and function of the IP is described, along with sufficient register information for customers to use the devices. The reference IP source code is located within the **pcores** directory of the ML300 Reference System's EDK project directory.

In addition to describing the individual hardware IPs, this volume also includes a specification of the IP InterFace, or IPIF modules. These modules are designed to greatly accelerate the process of hooking up pre-existent IP, or creating new IP in a system. The specification defines a CoreConnect compliant interface on one side, and a simple interface for hooking up existent IP on the other side. Since this is a building block piece of IP, it is available for use in any customer design.

The hardware IP uses the IBM CoreConnect bus standards as their means of communication between the PowerPC and other devices. These standards are documented in the IBM CoreConnect release. Please see the Further Reading section for more information on where to find the relevant documents.

Further information on each IP, and on the IPIF can be found in the following chapters of this volume. If you have suggestions for improvement, or have questions, please contact the relevant support contact listed in the letter that came with your ML300.

Hardware Reference IP Source Format and Size

The hardware reference IP available with the ML300 Embedded PPC405 Reference System originates in one language as either Verilog or VHDL source code. IP delivered in Verilog or VHDL source format is directly viewable and editable by the user as a text file. The EDK tools handle the process of building systems consisting of a mixture of IP written in different languages. For example, the PLB TFT LCD Controller is available only in Verilog source code so the EDK tools would need to convert into a blackbox netlist for use in a top level VHDL based design.

The table also includes information about the source code format and resource utilization for these cores. Many of the IP blocks are parameterizeable so their size may decrease depending on how they are configured. These area numbers represent a full implementation of each IP synthesized with the Xilinx tool XST. It is also important to note that as IP is connected together in a system, there are often cross boundary logic

optimizations and resource sharing that will further reduce the logic count of each IP. Slice utilization is only an estimate since the packing of lookup tables (LUTs) and flip-flops (FFs) into slices depends on the overall system implementation.

Table 4-1: Developer's Kit Hardware IP and Logic Utilization

Name	Source Code Format		Logic Utilization			
	Verilog	VHDL	Slice FFs	LUTs	Slices (Est)	BRAMs
OPB AC97 Sound Controller		X	177	219	129	0
OPB Parallel Port Controller	X		72	21	42	0
OPB PS/2 Controller	X		264	412	225	0
OPB to PCI Bridge Lite (Includes PCI Core)	X		1133	836	917	0
OPB to PLB Bridge-In Module Lite	X		221	84	132	0
OPB Touch Screen Controller	X		93	105	63	0
PLB TFT LCD Controller	X		260	226	163	1

Further Reading

Xilinx provides a wealth of valuable information to assist you in your design efforts. Some of the relevant documentation is listed below with more information available through the Xilinx Support website at http://support.xilinx.com. To obtain the most recent revision of documentation related to the ML300, see http://www.xilinx.com/ml300.

Resources for EDK Users (Including New Users)

EDK Main Web Page

http://www.xilinx.com/ise/embedded/edk.htm

Getting Started with the EDK

http://www.xilinx.com/ise/embedded/edk_getstarted.pdf

Embedded System Tools Guide

http://www.xilinx.com/ise/embedded/est_guide.pdf

EDK Tutorials and Design Examples

http://www.xilinx.com/ise/embedded/edk_examples.htm

Embedded Processor Discussion Forum

http://toolbox.xilinx.com/cgi-bin/forum?14@@/Embedded%20Processors

Documentation Provided by Xilinx

Virtex-II Pro Advance Product Specification (Data Sheet)

http://www.xilinx.com/bvdocs/publications/ds083.pdf

Virtex-II Pro Platform FPGA User Guide

http://www.xilinx.com/bvdocs/userguides/ug012.pdf

RocketIO Transceiver User Guide

http://www.xilinx.com/publications/products/v2pro/ug_pdf/ug024.pdf

IBM CoreConnect Documentation

The Embedded Development Kit integrates with the IBM CoreConnect Toolkit. This toolkit is not included with the EDK, but is required if bus functional simulation is desired. The toolkit provides a number of features which enhance design productivity and allow you to get the most from the EDK. To obtain the toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, Hardware IP, and the toolkit.

Xilinx provides a Web-based licensing mechanism that allows you to obtain the CoreConnect toolkit from our website. To license CoreConnect, use an Internet browser to access http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm. Once your request has been approved (typically within 24 hours), you will receive an e-mail granting access to a protected website. You may then download the toolkit. If you prefer, you can also license CoreConnect directly from IBM.

If you would like further information on CoreConnect Bus Architecture, please see IBM's CoreConnect website at http://www.ibm.com/chips/products/coreconnect.

Once you have licensed the CoreConnect toolkit, and installed it with the Developer's Kit, the following documents will be available to you in the following locations:

IBM CoreConnect Bus Architecture Specifications

IBM CoreConnect Processor Local Bus (PLB) Architecture Specification
see **$CORECONNECT/published/corecon/64bitPlbBus.pdf**

IBM CoreConnect On-chip Peripheral Bus (OPB) Architecture Specification
see **$CORECONNECT/published/corecon/OpbBus.pdf**

IBM CoreConnect Device Control Register (DCR) Bus Architecture Specification
see **$CORECONNECT/published/corecon/DcrBus.pdf**

IBM CoreConnect Toolkit Documentation

PLB Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/PlbToolkit.pdf**

OPB Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/OpbToolkit.pdf**

DCR Bus Functional Model Toolkit - User's Manual
see **$CORECONNECT/published/corecon/DcrToolkit.pdf**

CoreConnect Test Generator - User's Manual
see **$CORECONNECT/published/corecon/ctg.pdf**

Note: $CORECONNECT is an environment variable that is created when installing the Developer's Kit or CoreConnect Toolkit.

XILINX®

Chapter 5

Using IPIF to Build IP

Abstract

Virtex-II Pro™ devices combine PowerPC® CPUs and FPGA fabric into one integrated circuit. In the past, system development efforts relied on engineers building each component from scratch. Today, engineers have a wide variety of microprocessor peripherals in their IP libraries. The Intellectual Property InterFace (IPIF) is designed to ease the creation of new IP, as well as the integration of existent IP, within a Virtex-II Pro device. This chapter will illustrate the utility of the IPIF to integrate IP into a system.

Introduction

Intellectual Property InterFace (IPIF) modules simplify the development of CoreConnect™ devices. The IPIF converts complex system buses, such as the PLB or OPB, into common interfaces, such as an SRAM protocol or a control register interface. This makes IPIF modules ideal for quickly developing new bus peripherals, or converting existing IP to work in a CoreConnect bus-based system. The IPIF modules provide point-to-point interfaces using simple timing relationships and very light protocols.

The IPIF is designed to be bus-agnostic. This allows the back-end interface for the IP to remain the same while only the bus interface logic in the IPIF is changed. It therefore provides an efficient means for supporting different bus standards without change to the IP device.

IPIF modules also provide support for DMA and interrupt functionality. The IPIF is designed to support a wide variety of common interfaces (like SRAM, FIFO, and control register protocols), but may not be the optimal solution in all cases. Where additional performance or functionality is required, the user can develop a custom OPB or PLB bus interface.

IPIF modules simplify driver software development since the IPIF framework contains many common features. These include a consistent means of interrupt handling, DMA, and organizing control/status registers.

This document demonstrates how quickly and easily a new piece of IP can be developed using the IPIF. The process and steps for building a new CoreConnect device based on the SRAM protocol IPIF is described below. For this sample design, a 32-bit General Purpose I/O (GPIO) device will be created. The GPIO allows a CoreConnect master such as the CPU to be able to control a set of external pins using a simple memory-mapped interface.

SRAM Protocol Overview of IPIF

Figure 5-1 diagrams the connections between the IPIF and the user IP for SRAM protocol interface. The IPIF simplifies the design by providing a PLB or OPB interface and condensing it down to a small set of easily understood signals.

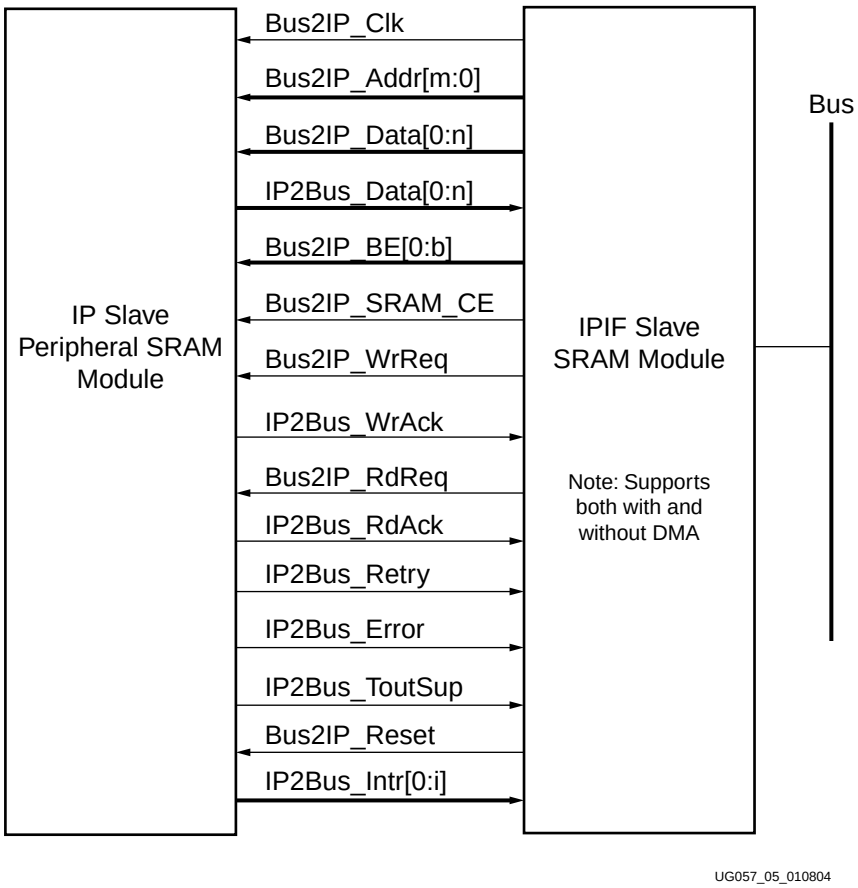


Figure 5-1: **IPIF SRAM Module Interface**

All interface signals with the IPIF are synchronous to rising clock edges. The IPIF takes the clock from the OPB or PLB bus interface and passes it to the IP, causing the IP to use the same global clock as the bus it is connected to. (Future IPIF designs may permit the IP clock and the bus clock to be independent.) The SRAM interface protocol used by the IPIF can be described by observing what a write and read transaction looks like.

Basic Write Transactions

Figure 5-2 shows the timing diagram for a write transaction. A write transaction begins when the IPIF drives the address (Bus2IP_Addr), byte enables (Bus2IP_BE), and write data (Bus2IP_Data) to the IP. Note that the signal direction is specified in the signal name: Bus2IP versus IP2Bus. The IPIF qualifies the write by asserting a single clock cycle High pulse (Bus2IP_WrReq) at the beginning of the transaction. It then waits for the IP device to acknowledge completion of the write by sending back a single clock cycle High pulse on IP2Bus_WrAck. During the entire transaction from Bus2IP_WrReq to IP2Bus_WrAck, the signal Bus2IP_SRAM_CE is held high as an enveloping signal around the transaction. After a completed transaction, the IPIF may issue a new transaction. Note that burst write transactions on the bus are converted into a series of single data transfers to the IP, which all look alike.

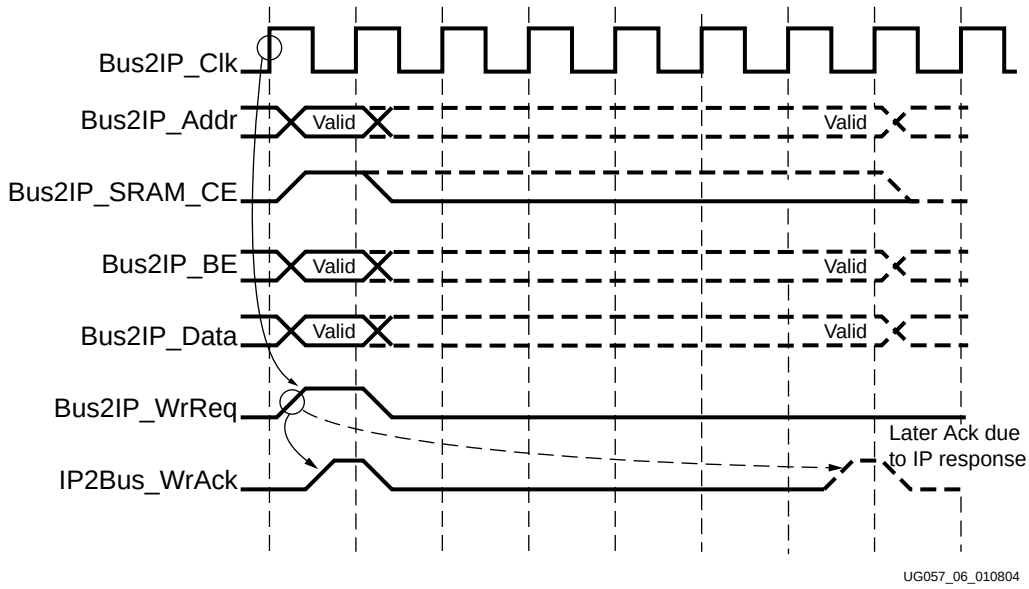


Figure 5-2: **IPIF Simple SRAM Write Cycle**

Basic Read Transactions

Figure 5-3 diagrams a read transaction, which looks very similar to a write transaction. A read transaction begins when the IPIF drives the address (Bus2IP_Addr) and byte enables (Bus2IP_BE) to the IP. It qualifies the read by asserting a single clock cycle high pulse (Bus2IP_RdReq) at the beginning of the transaction. It then waits for the IP device to acknowledge completion of the read by sending back a single clock cycle High acknowledge pulse on IP2Bus_RdAck. During the entire transaction from Bus2IP_RdReq to IP2Bus_RdAck, the signal Bus2IP_SRAM_CE is held high as an enveloping signal around the transaction. After a completed transaction, the IPIF may issue a new transaction. Note that burst read transactions on the bus are converted into a series of single data transfers to the IP, which all look alike.

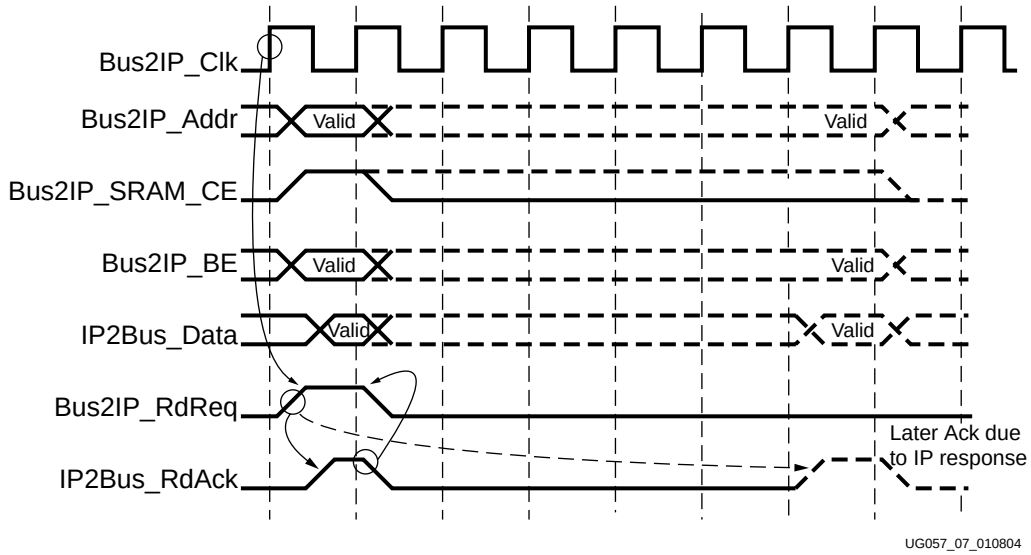


Figure 5-3: **IPIF Simple SRAM Read Cycle**

IPIF Status and Control Signals

Extra status and control signals are also present in the SRAM protocol. If the IP2Bus_Retry signal is asserted instead of IP2Bus_RdAck/IP2Bus_WrAck, the IPIF will assert retry on the bus side and terminate the transaction. IP2Bus_Error asserted with IP2Bus_RdAck/IP2Bus_WrAck will cause the IPIF to signal an error on the bus interface. For slow IP devices, an IP2Bus_ToutSup signal can be asserted to prevent timeouts on the bus interface. Finally the Bus2IP_Reset passes the bus-side reset to the IP.

Using IPIF to Create a GPIO Peripheral from Scratch

A General Purpose Input/Output (GPIO) peripheral can be used to show how the IPIF simplifies new peripheral creation. The GPIO module has three 32-bit registers: one register to control the TBUF for each I/O pin, one register to write the I/O pins, and one register to read the I/O pins. The GPIO peripheral uses a very small amount of additional "control logic" when used with a 32-Bit IPIF Slave SRAM module.

Figure 5-4 shows a conceptual view of the logic necessary to build the GPIO module using the IPIF Slave SRAM module. The IP2Bus_RdAck / IP2Bus_WrAck signals are directly connected to the corresponding Bus2IP_RdReq / Bus2IP_WrReq signals, since it only takes one clock cycle to read or write the GPIO registers. If more "access time" is required by the registers, a simple SRL16-based shift register between the Req/Ack signals could be used to set the number of cycles the register will respond in. An example use of this function is to gain timing margin by treating the register access as a multicycle path. This simple enhancement to the IPIF can have very positive effects in meeting the timing requirements typical of complex microprocessor-based systems. Note too that register response time can be tuned differently between the read and the write.

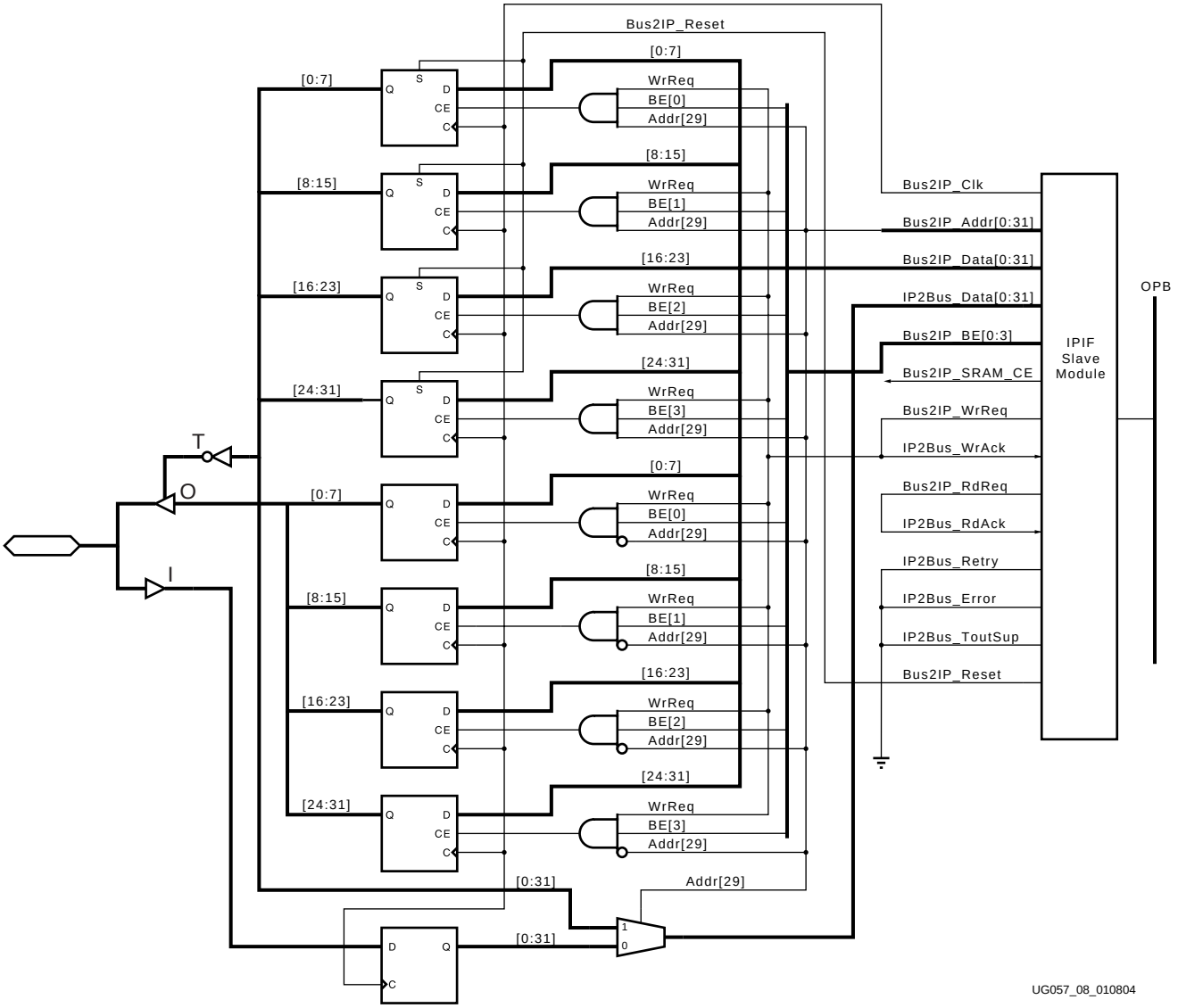


Figure 5-4: **IPIF SRAM Module to GPIO logic Interface**

To drive an external I/O pin, the output enable for that pin must be asserted, allowing the pin to be driven high or low based upon the contents of the write register. If the output enable for a given pin is deasserted, the pin's driver is put in a high impedance state allowing an external device to drive the pin. The CPU can sense the current value of any pin (regardless of its direction) by reading the read register. Driving the direction of the I/O pin is controlled by the contents of the three-state register.

The GPIO registers support byte enables during writes to the 32-bit registers. Note that a set of simple AND gates is all that is required to generate a Clock Enable to the registers. Four 3-input AND gates are used to drive the 4 bytes of the three-state control register, and four more 3-input AND gates are used to drive the 4 bytes of output-pin data.

The IPIF uses a set of user-specified parameters that allow common things such as the base address of the IP to be established. These parameters are specified before the system is implemented in order to minimize the logic area and maximize the performance of the

system. Note that in the GPIO example, additional decoding is used externally to specify two different memory locations. One location is used for reading or writing to the I/O pins (read register, and write register share the same address), and one location is used for reading and writing to the T of the I/O pin (three-state register).

Using IPIF to Connect a Pre-existent Peripheral to the Bus

Often, some legacy IP will need to be brought into a modern system. Many of these legacy IPs use some form of 8-bit microprocessor bus. Typically, this might consist of a few address lines, an 8 bit data bus, a read and write signal, a chip enable, a clock, a reset, and perhaps an interrupt pin. In most instances, this kind of IP can be almost directly connected to the IPIF SRAM module. This particular IPIF module was actually designed to serve this very purpose.

To connect a legacy IP, one would simply connect the address, data, chip enable, clock, reset, and interrupt pins to their corresponding versions in the IPIF SRAM module. Some small amount of logic might be needed to generate a properly timed read or write signal. The IPIF SRAM module provides separate Req/Ack pairs for read and write, because many older peripherals require different timing for reads and writes. For read or write, the logic between the IP and the IPIF must accomplish two things:

- Provide the proper response time to the IPIF so the peripheral's register can be read or written
- Provide the proper relationship of the read or write signal on the IP relative to the address and data

Consider the following example: The IP might expect its write signal to be valid one clock after address and data is valid, and be held for four clock cycles to properly write the data. Following write going invalid, the address and data must be held for one additional cycle. To accommodate this kind of pattern, a six-stage shift register (SR) can be implemented. The D input to the SR is tied to the Bus2IP_WrReq pin, and the Q output of the SR is tied to the IP2Bus_WrAck pin of the IPIF. This provides the proper timing for the length of time the cycle must be held on the bus. By using the first, second, third, and fourth taps of the SR, and feeding them into an OR gate, a write strobe can be generated for the IP. If this write strobe must be glitch-free, taps 0, 1, 2, and 3 could be used, OR'd and fed into a synchronizing register. While on the surface this may appear wasteful of logic, Xilinx FPGAs are abundantly equipped with flip-flops.

Conclusion

Using the IPIF with a small amount of logic makes it very easy to create CoreConnect devices with little knowledge of the buses used. For complex buses such as PLB, this saves the designer time and helps to ensure IP functions correctly, since the IPIF provides a pre-verified design to connect to. The GPIO design is just one example of how IPIF can be used. More examples of IPIF designs are provided within many of the other IP devices in the reference systems. It is recommended that the designer who wishes to learn about IPIF studies the sample source code for some of these IPIF-based designs in context with simulation to gain experience with IPIF.

The IPIF used in the ML300 Embedded PPC405 Reference System currently only supports the SRAM module. Additional IPIF modules are available through EDK that support many parameterizeable features. Refer to the IPIF chapter of the *Processor IP User Guide* located in <*EDK Install Directory*>/doc/proc_ip_ref_guide.pdf.

Chapter 6

IPIF Specification

Note: This document is provided as reference since some Hardware Reference IP is built using IPIF modules conforming to this version of the specification. Please refer to the IPIF chapter of the *Processor IP User Guide* (located in *<EDK Install Directory>*/doc/proc_ip_ref_guide.pdf) for the latest information and documentation on IPIF cores. New designs should use these IPIF modules available through EDK.

Overview

The Intellectual Property InterFace (IPIF) backplane is a framework that provides a common set of interfaces for connecting intellectual property to on-chip buses. These common interfaces are built in a modular fashion, allowing standardized connections between IP and the on-chip bus. Additionally, these modules provide an infrastructure to assist in developing new IP, as illustrated in the examples of on-chip bus interface techniques.

Each interface is described as a module that can be independently "plugged in" to the backplane. This modular approach allows the customer to pay only for the resources that are required to implement the desired functions.

Although this specification describes the overall use of interface modules in a system, but it does not address how the IPIF backplane integrates with the rest of the system. This information is addressed in the IPIF Architectural Specification, *not yet released*.

The following documents provide additional useful information:

- IPIF Architectural Specification
- Virtex-II Pro™ Data Sheet
- IBM CoreConnect™ PLB Architectural Specification

Xilinx IPIF modules, available to internal IP developers, third party IP developers, and customers, are designed around a common modular framework that permits mixing and matching of various modules. The interfaces provided in this framework are based on commonly used interfaces for existent IP, with a mind to future IP development as well. These interface modules represent an abstraction layer to assist in developing fully customized peripherals for an on-chip bus. Figure 6-1 illustrates the scope of the interfaces addressed in this specification.

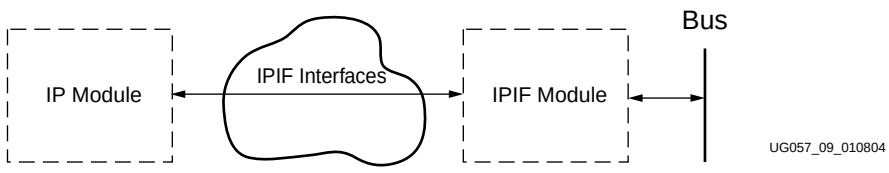


Figure 6-1: **Scope of Interfaces in This Specification**

The IPIF Backplane provides two basic interface classes: IPIF master and IPIF slave. This specification defines one IPIF master, and four IPIF slave modules. The various modules allow system integrators to minimize the FPGA resources required to implement a particular IP, easing difficulty in achieving performance targets and reducing cost.

IPIF Master Module Overview

The IPIF master module is a single interface that looks very much like a simple SRAM interface. The master IP must provide address, data, and a read or write command in order to initiate a transaction onto the bus. The IPIF master module provides 8, 16, or 32-bit interfaces to the master IP as required by the system implemented.

IPIF Slave Modules Overview

IPIF slave modules can be built with any combination of four different interfaces:

- SRAM
- Control Register
- FIFO
- DMA Handshake

Each interface is designed to address differing needs, but can be combined to meet multiple needs. For example, a customer with a typical microprocessor peripheral IP, such as a 16550 UART, can use the IPIF Slave SRAM module to easily connect a peripheral. Similarly, if a customer requires only a few control registers, the IPIF Slave Control Register module can be used to connect the control registers to the bus. The IPIF Slave FIFO module is designed to connect to communications devices with FIFO interfaces. The IPIF Slave DMA Handshake module provides the classic DMA_Req/DMA_Ack handshake to enable devices that require such handshakes for proper operation.

Note: The IPIF Slave DMA Handshake module differs from the optional DMA engine described in the "DMA Engine" section.

For certain existent microprocessor-style peripherals, mixing IPIF modules may be advantageous. The IPIF master module and IPIF slave modules can be instantiated as mix-and-match components. For example, a simple IP might only require an IPIF Slave SRAM module, whereas a more complicated IP might require IPIF Master, Slave SRAM, Slave Control Register, and Slave FIFO modules. All modules share logic on a common backplane, thereby minimizing FPGA resource utilization for the desired functions.

Master and slave IPIF modules have user-settable parameters that control their capabilities. Typical parameters include which IPIF modules to build, what addresses they reside at, how wide their IP data path is, etc. Additionally, the IPIF master and IPIF slave modules can be built with a Direct Memory Access (DMA) controller of varying capabilities, referred to in this specification as the DMA engine. The DMA engine can be instantiated into any of the IPIF modules to provide basic DMA service, Scatter Gather (SG) via linked list and/or linear list, as well as basic packet processing functions. The DMA engine and other capabilities of the IPIF are set at the time of instantiation, and are visible to software for device discovery and management. (For more details, see the DMA/SG section of the IPIF Architectural Specification, *not yet released*.)

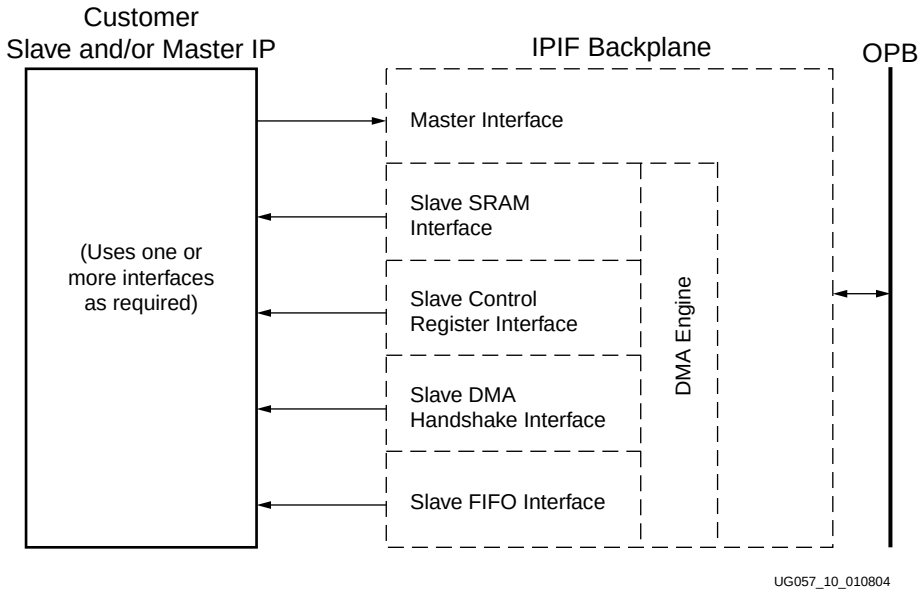


Figure 6-2: **General Model of the IPIF Backplane**

The IPIF modules are really subsets of a general model, demonstrated in Figure 6-2. The general IPIF model was developed such that each IPIF module could be instantiated as required by the IP system integrator. Note that the general model of the IPIF shows each of the module types: IPIF Slave DMA Handshake, IPIF Slave Control Register, IPIF Slave SRAM, IPIF Slave FIFO, and IPIF Master. Additionally, the general model of the IPIF shows the optional DMA engine. This general model offers the IP integrator broad design functionality.

Signal Conventions

The signal names used throughout this specification use a specific format to identify the direction and function of each signal. This signal name format is illustrated in Figure 6-3.

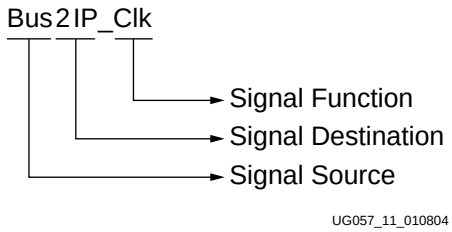


Figure 6-3: **Signal Name Format Example**

In addition, three source and destination signals have specific meaning:

- Bus - Signals that connect to the IPIF
- IP - Signals that connect to the IP
- FIFO - Signals that connect to a FIFO in the IPIF, when present

Bus Numbering and Bit Ordering

The IBM signal name convention is used for address, data, and control buses throughout this specification. IBM numbers the bits from left to right as 0 to n. For example, a 32-bit data bus is named IP2Bus_Data[0:31].

The bit order for buses always has the most significant bit (MSB) on the left, and the least significant bit (LSB) on the right. Thus, IP2Bus_Data[0:31] is identified as a 32-bit bus, whose MSB is bit 0, and whose LSB is bit 31. This is commonly known as "big endian" format.

The IPIF specification uses big endian bit ordering for all of its examples. However, it is possible to use parameters to alter the endianness of the IPIF for proper interaction with Motorola or Intel style IP. *(These parameters will be covered in a new section, TBD.)*

Parameter Indexing Versus Parameter Numbering

The function of the IPIF can be changed by setting various parameters. Please see "IPIF Parameterization" for more details. There are four main classifications of parameters that are used within this specification: number, index, boolean, and mask.

Consider a parameter that adjusts the size of the data bus. In this case, the parameter is best defined as a number. For example, the IPIF data bus width can be set to 8, 16, or 32, depending on how the parameter IP_DATA_BUS_WIDTH is set.

However, in the bits of the buses whose width is controlled by IP_DATA_BUS_WIDTH, the indices of the bus start at zero and end at IP_DATA_BUS_WIDTH - 1. For example, IP2Bus_Data[0:*n*], where *n* is the index parameter based on the IP_DATA_BUS_WIDTH number parameter minus one.

The result of this situation is that parameters in the IPIF are defined as either number parameters or index parameters, in order to clearly identify the function of the actual parameter value.

Boolean parameters are those that have binary states; 1 = on and 0 = off. For example, IP_HAS_OWN_INTC.

Parameters can also be mask values. A mask is a bit-wise operation that is used by the IPIF to apply a specific enable operation, typically against another parameter. For example, the IPIF Slave SRAM module defines an array of parameters called IP_SRAMb_BASE_ADDR. These parameters specify the start address of a particular region of memory which the IPIF will decode. In order to allow for a set of addresses to reside within the space began at IP_SRAMb_BASE_ADDR, a mask value parameter is used to set which specific bits of the address are decoded and which are ignored. This is set by the IP_SRAMb_BASE_ADDR_BIT_ENBL mask value.

IPIF Modules in an Example OPB System

Figure 6-4, page 63 is a diagram of an example system utilizing an IBM CoreConnect™ On-Chip Peripheral Bus (OPB). It illustrates a variety of IPIF modules connected to a variety of IP elements. This example system is intended only to illustrate general abilities, and should not be construed as limitations upon the IPIF.

The IPIF easily connects to new and existing IP. The example in Figure 6-4 illustrates possible scenarios.

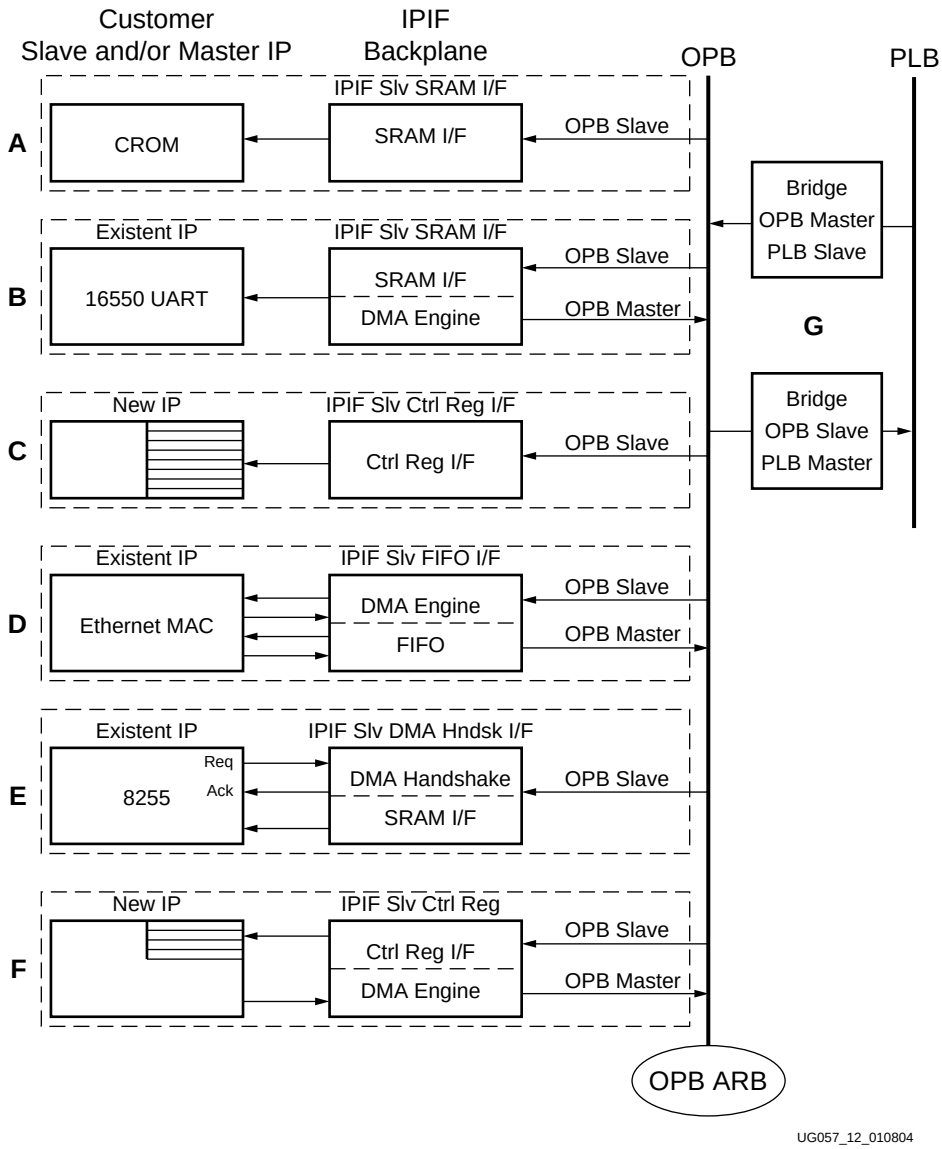


Figure 6-4: **Example System Using IPIF to Connect to OPB**

A: Slave SRAM to CROM

Section A in Figure 6-4 illustrates an IPIF Slave SRAM module used to connect to a configuration ROM (CROM) made from a Xilinx block RAM. The CROM can be used to store information about the entire system, such as the system version number, IP capabilities, and IP version numbers. (More information on CROM structures appears in the IPIF Architectural Specification, *not yet released*.)

B: Slave SRAM to UART

Section B in Figure 6-4 shows another IPIF Slave SRAM module that interconnects with an existent IP, such as the 16550 UART shown here. This example uses the optional DMA

engine that is instantiated to relieve the CPU burden in moving data. (More information on DMA engine capabilities appears in the IPIF Architectural Specification, *not yet released*.)

C: Slave Control Register to New IP

Section C in Figure 6-4 shows a customized piece of new IP that utilizes the IPIF Slave Control Register module in order to reduce complexity in the IP. By using this module, the new IP can directly instantiate its registers that are connected to the IPIF Slave Control Register module. The IPIF backplane operates independently of the registers in the new IP, and does not rely on whether the registers are read/write, read only, or write only. The IPIF Slave Control Register module allows other OPB masters to have memory mapped access to the registers in the IP.

D: Slave FIFO and DMA Engine to Ethernet MAC

Section D in Figure 6-4 illustrates the IPIF Slave FIFO module and optional DMA engine interfaced to an Ethernet MAC, directly providing the bulk of logic to allow quick design of an IP system. Some communications IP might also contain registers that are memory mapped, which may require additional IPIF Slave Control Register or IPIF Slave SRAM models, depending upon the type of interface provided by the IP.

E: Slave DMA Handshake to 8255

Section E in Figure 6-4 shows another example of an old legacy CPU peripheral, in this case an 8255 Parallel I/O IP. Legacy devices often contain fairly simple CPU interfaces and DMA handshaking provisions. These kinds of IP can easily connect to the IPIF backplane. The IPIF Slave SRAM module is used for the CPU interface to the IP and the IPIF Slave DMA Handshake module is used for the DMA handshake of the IP. The IPIF DMA Handshake module allows the IPIF backplane to store up a series of accesses before interrupting the processor that is requesting service.

While it is possible to implement the optional DMA engine in this example, it might be FPGA-area inefficient to use it for a slow device that does not have high bus utilization (e.g. small data rates). The DMA Handshake module allows support of the DMA function of the IP, but does not inherently require the use of the optional DMA Engine to effect the proper behavior. The point of this IPIF example is to illustrate the power to choose various options while considering other variables in the system, such as performance, area, and speed.

F: Master with Slave Control Register and DMA Engine to New IP

Section F in Figure 6-4 shows a typical instance of a master/slave IPIF. In general, most master IP devices also require slave interfaces to control the master interface. This example assumes that the a new piece of IP has been created that requires both master and slave access. Additionally, the optional DMA engine is used in this example to offload the CPU from the data transfers. The IPIF Slave Control Register module is used to communicate to the slave side of the IPIF backplane. This offers quick and easy memory mapped access to the internal registers of the IP, including the DMA engine registers.

G: Bus-to-Bus Bridges

Section G in Figure 6-4 illustrates a pair of bus-to-bus bridges. In this case, between Processor Local Bus (PLB) and the OPB. One bridge provides master access to the OPB, while the other bridge provides slave access.

Design Considerations

IPIF modules can be implemented with many different options. The available options and related issues are addressed in this section.

DMA Engine

The optional capabilities of the DMA engine are illustrated throughout Figure 6-4. For example, the DMA engine in the IPIF Slave SRAM module connected to the 16550 UART only requires very basic DMA movement operations. In contrast, the Ethernet MAC requires packet-aware DMA for highest bandwidth transmission and reception. While a single DMAC could be instantiated to handle both, it can complicate how embedded software handles the device. Xilinx's recommended system uses distributed DMA engines, one per device, to ensure a clean device driver environment. Additionally, distributed DMA will enhance performance of the overall system since many independent agents can be simultaneously active. Having optional DMA engine capabilities means that only the logic required by the IP is paid for in the FPGA fabric. (More information on DMA engine capabilities will appear in the IPIF Architectural Specification, *not yet released*.)

Interrupts

In addition to the DMA engine capabilities, the IPIF backplane can handle various aspects of interrupts for the system, as shown in Figure 6-5. Existent IPs typically have their own interrupt pins and sets of registers internal to the IP for status and control of interrupts. However, this infrastructure must be designed for new IPs. The IPIF model allows for all of this and more. The IPIF model generally allows from 0 to 8 independent interrupts. Two basic modes can be optionally enabled: existent IP mode, where a single INT pin is provided from the IP to the IPIF; and new IP mode where up to 8 INT pins are provided and the register infrastructure is built in the IPIF. The parameters to the IPIF modules permit the FPGA resources to be optimized to the minimum required by the IP. (More information on interrupts appears in the IPIF Architectural Specification, *not yet released*.)

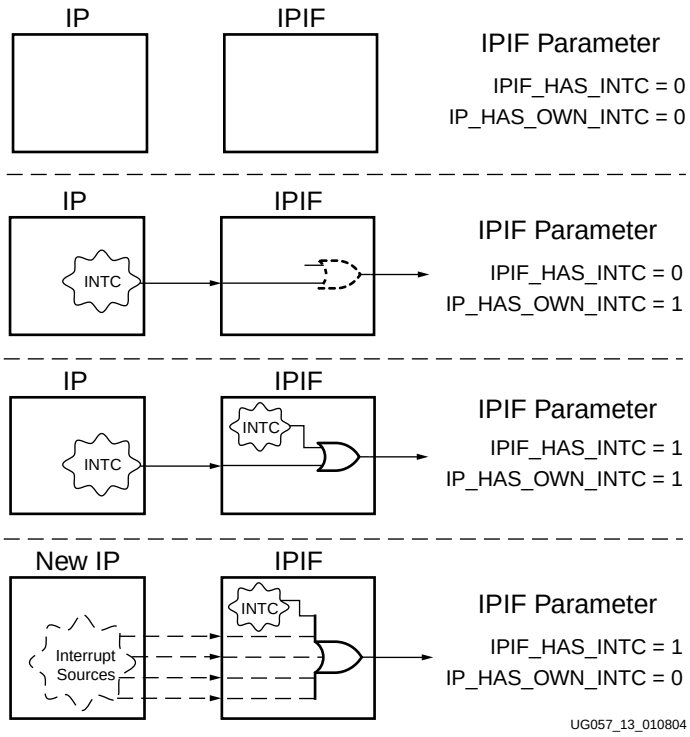


Figure 6-5: **Example of Three Interrupt Scenarios**

Bus Arbiter and Bridges

Every on-chip bus that contains multiple masters requires some form of arbitration control. Section G in Figure 6-4 illustrates an example of the OPB arbiter. Arbiters typically have a variety of parameters that can be statically or dynamically set to address system integration issues such as priority control and pipelining. (See the IBM CoreConnect™ PLB Architectural Specification for additional information on both PLB and OPB arbiters.)

Most embedded computing systems use layers of bus hierarchy to separate the high-speed devices from the low-speed devices. Typically, the CPU and memory sit on one high speed, lightly loaded bus, while the other elements reside on a heavily loaded, slower bus. To communicate between the two, a bridge-in/bridge-out function is used. The bridge out allows the CPU to act as a master to the slower bus. The bridge in function allows devices (such as DMA engines) to act as a master on the higher speed bus. Typically, to talk to system memory. The IPIF example in section G of Figure 6-4 illustrates both bridge-in and bridge-out functions of a more complex system.

Data Bus Width

The bus side of the IPIF is always 32 bits. Since an IPIF module can be 8, 16, or 32 bits wide, some unexpected behavior may occur. If a bus master issues a request for a word transfer (32 bits) across the bus to an IPIF module that implements only a byte-wide IP data width, the IPIF will sequence through the data as four separate requests to the IP. That is, four separate request/acknowledge cycles will occur between the IP and IPIF.

Retry, Error, and Timeout Suppress

All IPIF Slave modules permit the IP to tell the bus to retry, or that an error has occurred. The IP logic tells the IPIF this by asserting the **IP2Bus_Retry** or **IP2Bus_Error** signal, with or without the appropriate acknowledge signal for the intended cycle. Additionally, the IP may suppress the normal timeout (16 bus clocks) mechanism for transfers that take longer on the bus. The use of the **IP2Bus_ToutSup** signal is not recommended unless absolutely necessary since it has a detrimental effect on bus bandwidth.

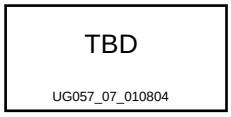


Figure 6-6: **Relationship of Retry, Error, and Timeout Suppress Signals**

Figure 6-6 shows the relationship of the **IP2Bus_ToutSup**, **IP2Bus_Error**, and **IP2Bus_Retry**. The **IP2Bus_Error** signal is used to indicate to the master (which initiated the transaction) that some unspecified error has occurred.

The **IP2Bus_Retry** is not a likely requirement in most applications. It is used to signal the bus master to retry the bus cycle, forcing the bus master off the bus. The master then rearbitrates for the bus and attempts the cycle again. In general, retry is only used for deadlock conditions, such as when a bus master attempts to access a bus master/slave while the bus master/slave is attempting a cycle of its own.

IP2Bus_Retry is sometimes used as a means to hold off further bus access to the IP until the bus is no longer busy. Typically, a bus cycle initiates an operation that forces the IP to a busy condition. Further bus cycles could result in inappropriate behavior, therefore the IP issues **IP2Bus_Retry**. This use would strain available bus bandwidth, but can be used to implement a primitive semaphoring mechanism. The **IP2Bus_Retry** signal would be issued by the IP while the device is not ready, and the cycle would be acknowledged once the IP was ready. Again, this is very expensive in bus bandwidth, but the use is not prohibited.

IPIF Module Specifications

Slave DMA Handshake Module

This section covers the following topics:

- "Example Slave DMA Handshake Application"
- "Generic Slave DMA Handshake Model"
- "Slave DMA Handshake Signal Protocol"
- "Slave DMA Handshake Signal List"
- "Slave DMA Handshake Parameters"

The simplest IPIF slave module is the DMA Handshake module. This module is typically used in concert with one of the other slave modules to connect existing IP to a bus. It uses two simple handshake signals, **IP2DMA_Req** and **DMA2IP_Ack**, along with two unidirectional data buses. This permits the IP to request service from the optional DMA engine, which can be built into an IPIF module. The IPIF Slave DMA Handshake module is not recommended for new IP designs since its functionality can be handled directly by the optional DMA engine in other modules.

Example Slave DMA Handshake Application

Figure 6-7 shows an example application of the IPIF Slave DMA Handshake module. This example shows how the **DMA_Req** and **DMA_Ack** pins of an existent IP peripheral can connect to the IPIF module. The IP block will likely use other IPIF modules too, even though they are not shown in this example.

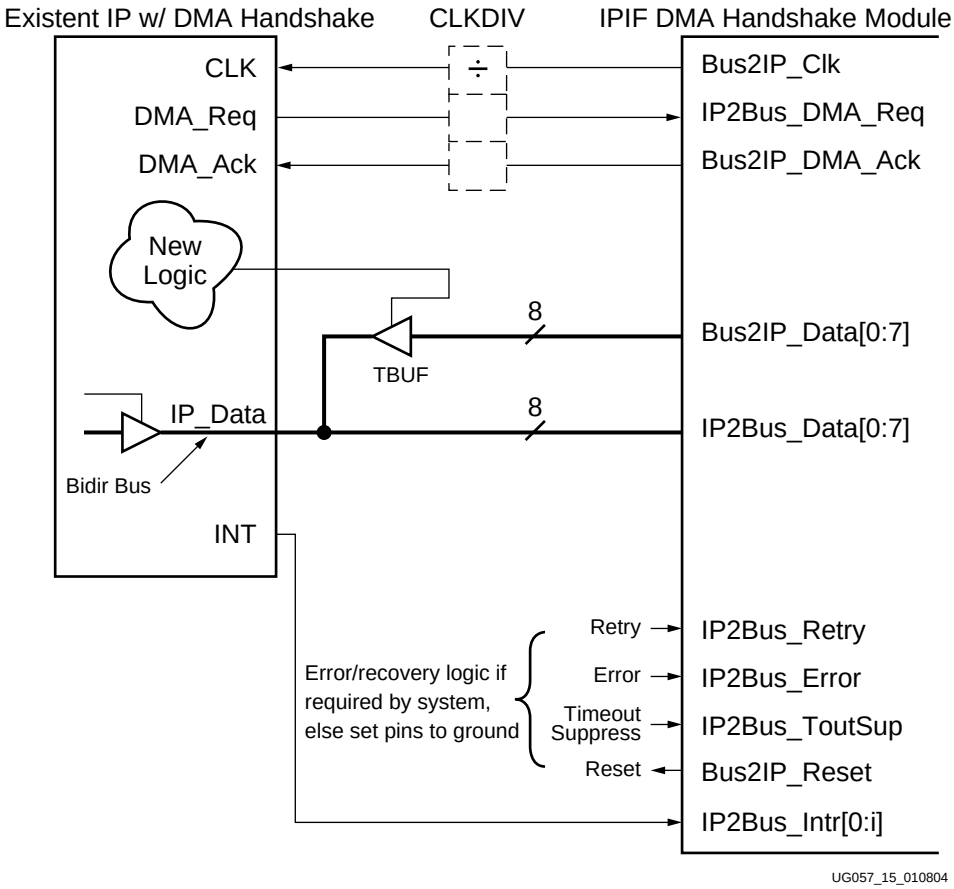


Figure 6-7: **Example Slave DMA Handshake Application**

Existent IP often utilizes a bidirectional bus instead of dual unidirectional buses. In these kinds of scenarios, the read and write sides must be separated into unidirectional buses. In some IP, the 3-state logic is implemented in the I/O where it is easy for it to cut out the I/Os. However, often the IP utilizes FPGA TBUFs to accomplish the same end. When TBUFs are used internal to the IP, removing them may be problematic.

The example application in Figure 6-7 assumes the use of TBUFs internal to the IP and, therefore, must be converted to single unidirectional data lines. The IP provides its data on the **IP_Data** bus. **IP_Data** is split into **Bus2IP_Data** by using a set of TBUFs to provide isolation from the **Bus2IP_Data** bus from the IP_Data bus. The T pins on the TBUFs must be driven from within the IP using some small, new logic. The IP must tell the **Bus2IP_Data** bus when it can write onto the **IP_Data** bus. This arrangement permits the **Bus2IP_Data** bus to be valid for long periods of time before a write cycle, if required. The **IP2Bus_Data** is simply connected to the **IP_Data** bus unless it is required to be otherwise qualified.

The IP's **DMA_Req** signal connects to the **IP2Bus_DMA_Req** signal and the IP's **DMA_Ack** signal connects to the **Bus2IP_DMA_Ack** signal on the IPIF Slave DMA Handshake module. Since this type of interface does not imply a direction of data flow, it is important that the direction is agreed to prior to any requests for service. Generally, this is done by the CPU setting specific registers inside the IP to indicate a read or write, and then doing the same in the IPIF module.

Since many existent IPs will not operate at the high clock frequency of a bus, it might be necessary to provide a clock divider and global clock buffer to provide a frequency that the IP can handle. If this occurs, the **IP2Bus_DMA_Req** and **Bus2IP_DMA_Ack** require additional circuitry to cross the clock domains. Simple synchronous set/reset flip-flops can be used to guarantee correct operation.

The 3-state control on the TBUF which drives **Bus2IP_Data** onto the **IP_Data** bus usually requires some small modifications to the existent IP logic. Typically, a write signal is available inside the IP, and can be used to build a read/write qualifier. Additionally, the **DMA_Req** and **DMA_Ack** signals can be fed into a set/reset synchronous flip-flop to build an enveloping signal. The enveloping signal, when AND'ed with the write signal of the IP, can then be used to control the 3-state control of the **Bus2IP_Data** TBUF.

Generic Slave DMA Handshake Model

Figure 6-8 illustrates a generic instance of the IPIF Slave DMA Handshake module. The data bus widths and number of interrupts for the IP are generically specified.

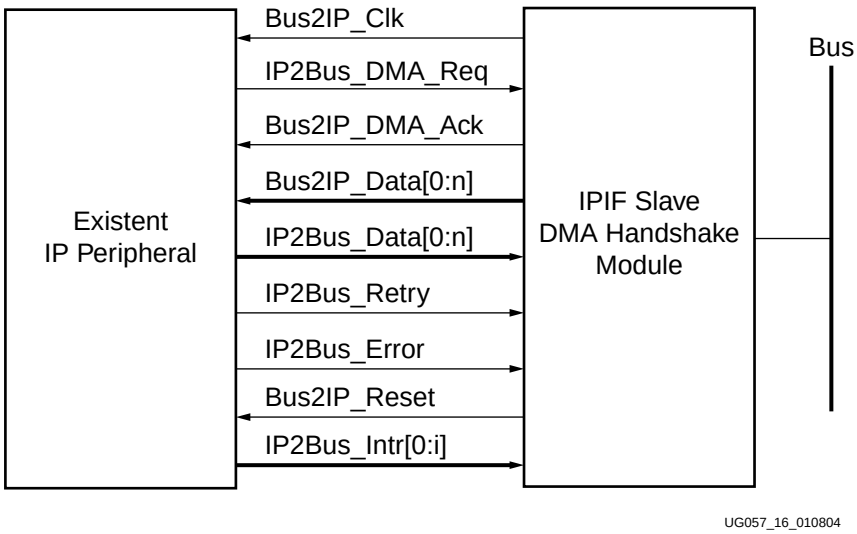


Figure 6-8: **Generic Instance of the IPIF Slave DMA Handshake Module**

The IPIF Slave DMA Handshake module outputs a clock, **Bus2IP_Clk**, that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP_Clk**. This clock can be asynchronous from the bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

Slave DMA Handshake Signal Protocol

Figure 6-9 illustrates a simplified signal protocol diagram of the IPIF Slave DMA Handshake module. Note that the **IP2Bus_DMA_Req** is required to be a single clock high at the **Bus2IP_Clk** rate. The data validity is based upon the direction agreed to between the IP and the IPIF. For IPIF read data, the data bus is **IP2Bus_Data**. For IPIF write data, the data is **Bus2IP_Data**.

The IPIF Slave DMA Handshake module may include internal parameters, such as response time for **Bus2IP_DMA_Ack**, and its width when valid (in **Bus2IP_Clk** cycles). These parameters are used to specify a minimum time that the IPIF module can answer a request for service, and a time that the IP requires the data to be valid. Since the IP is often legacy IP, and is generally slower than the IPIF Slave DMA Handshake module, the ability to determine the response speed is critical to ensure proper operation of the IP.

The minimum time before **Bus2IP_DMA_Ack** will go high is controlled by the DMA_HNDSHK_RESPONSE_TIME_MIN parameter. This parameter guarantees that the IPIF module will not respond any sooner than the specified number of **Bus2IP_Clk** cycles. The IPIF module may respond in more cycles, however, depending on bus activity on the bus. In the example shown in Figure 6-9, the DMA_HNDSHK_RESPONSE_TIME_MIN parameter is set to two cycles, but the IPIF does not acknowledge the cycle until the third cycle after the request.

The DMA_HNDSHK_DATA_VALID_WIDTH parameter sets the number of **Bus2IP_Clk** cycles that the **Bus2IP_DMA_Ack** signal will be held high. During IPIF write cycles, the IPIF will drive the **Bus2IP_Data** bits with write data for the entire duration of **Bus2IP_DMA_Ack**. IPIF read cycles require the IP to drive the IP2Bus_Data buses with read data prior to the last positive edge of Bus2IP_Clk that samples **Bus2IP_DMA_Ack** high. The **IP2Bus_Data** line must continue to be held until the **Bus2IP_DMA_Ack** had drawn low. Read data is sampled by the IPIF on the rising edge of the **Bus2IP_Clk** that caused **Bus2IP_DMA_Ack** to fall.

Note:

1. Data buses are shared between the IPIF Slave SRAM and IPIF Slave Control Register modules. Data qualifiers are used to indicate which module has access to the bus.
2. The specification allows driving data into the next cycle, as shown in Figure 6-9.

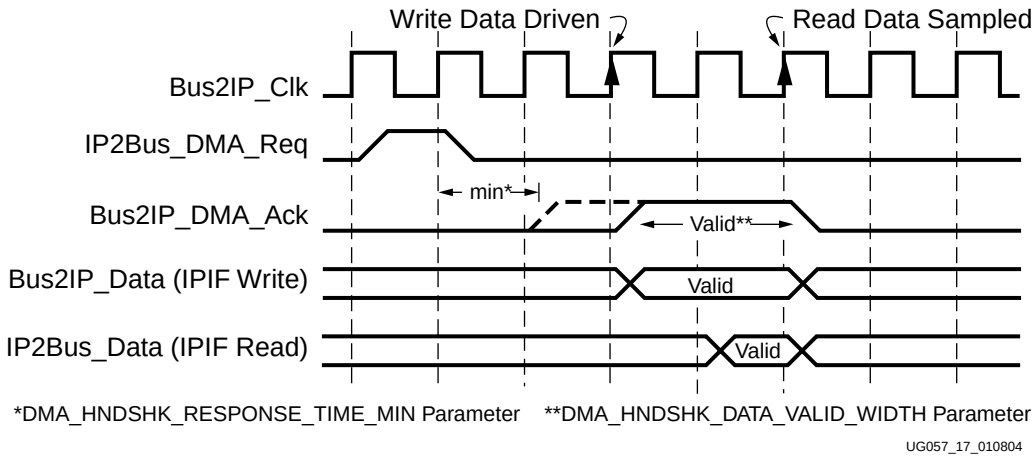


Figure 6-9: **IPIF Slave DMA Handshake Signal Protocol**

Slave DMA Handshake Signal List

Table 6-1 shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave DMA Handshake module.

Table 6-1: **IPIF Slave DMA Handshake Signals Connecting to IP**

Name	Direction	Description
Bus2IP_Clk	From IPIF	Clock source (from global buffer)
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF
IP2Bus_Error	To IPIF	Error signal from IP to IPIF (valid only during a data acknowledge cycle)
IP2Bus_Retry	To IPIF	Indicates IP wants master to retry the cycle
IP2Bus_ToutSup	To IPIF	Forces the suppression of watch dog timeout on the bus
Bus2IP_Data[0:n]	From IPIF	IPIF Write data (where n = IPIF_DATA_BUS_WIDTH –1)
IP2Bus_Data[0:n]	To IPIF	IPIF Read data (where n = IPIF_DATA_BUS_WIDTH –1)
Bus2IP_BE[0:b]	From IPIF	Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)
IP2Bus_DMA_Req	To IPIF	DMA handshake transfer request from IP
Bus2IP_DMA_Ack	From IPIF	DMA handshake transfer acknowledge from IPIF

Slave DMA Handshake Parameters

Table 6-2 shows the parameters that can be selected for the IPIF Slave DMA Handshake module..

Table 6-2: **IPIF Slave DMA Handshake Module Parameters**

Affects	Parameter	Value	Type
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
General IPIF Slave DMA Handshake Module	DMA_HNDSHK_RESPONSE_TIME_MIN Sets the minimum number of **Bus2IP_Clk** cycles in which the IPIF will respond with a DMA_ACK	0 to 255	number
	DMA_HNDSHK_DATA_VALID_WIDTH Sets the number of **Bus2IP_Clk** cycles that the data will remain valid during DMA handshaking	0 to 255	number

Slave Control Register Module

This section covers the following topics:

- "Example Slave Control Register Application"
- "Generic Slave Control Register Model"
- "Slave Control Register Signal Protocol"
- "Slave Control Register Signal List"
- "Slave Control Register Parameters"

The IPIF Slave Control Register module provides a set of signals which permit basic control registers to directly attach to a bus. The overall number of registers, number of bits in each specified register, and direction for each bit of each register are parameters that may be specified when instantiating this module.

Example Slave Control Register Application

Figure 6-10 contains an example application of the IPIF Slave Control Register module. It illustrates how read/write, read only, and write only registers can be implemented. Typical systems contain many kinds of registers, each of which can be easily connected to the IPIF Slave Control Register module.

The read/write (R/W) register shown in Figure 6-10 is 24 bits wide, and is centered about the 32-bit data bus as D[8:31] logically. Since it is a read/write register, it requires data into the flip-flops from both user logic (if present) and the IPIF module. Accordingly, a multiplexer, controlled by the user logic, allows the write update to the register to be selected between the IP and IPIF. Also note that the CE pin of the R/W register is OR'ed together with the CE (**Bus2IP_RegCE1**) provided by the IPIF and the CE from the user logic. Also note that the Q outputs of the R/W register are available to user logic as needed, and are multiplexed into the **IP2Bus_Data** path for read access of the register.

The read-only (R/O) register is read back across the bus when the address on the bus is set for the decodes of **IP2Bus_RegCE2**. Address decoding for all the registers is contained within the IPIF Slave Control Register module and is specified by the user during instantiation. In this example, the R/O register is 16 bits and connects to **D[16:31]** logically. This R/O register is truly read only. The user logic is responsible for updating the contents of the register. While the IP logic might utilize the Q outputs of the R/O register, the IPIF Slave Control Register module can not write to the register. The R/O register's data path is multiplexed with the R/W register back to the **Bus2IP_Data** bus of the IPIF Slave Control Register.

The write-only (W/O) register in this example is instantiated as a 24-bit register and connects to **D[0:23]** logically. Like the R/W register, the W/O register may have user logic which can write to the it. Accordingly, a mux is needed on the D input and at least an OR gate on the CE input to the register. The Q outputs of the W/O register are not available to the IPIF Slave Control Register module, making this register truly write-only. The W/O register in this example also demonstrates the use of an additional qualifying signal to update the content during IPIF write operation. A signal called **WE** causes the write to occur at a specific time in the bus cycle. Similar logic can be used to force the write to occur at any time during the bus cycle.

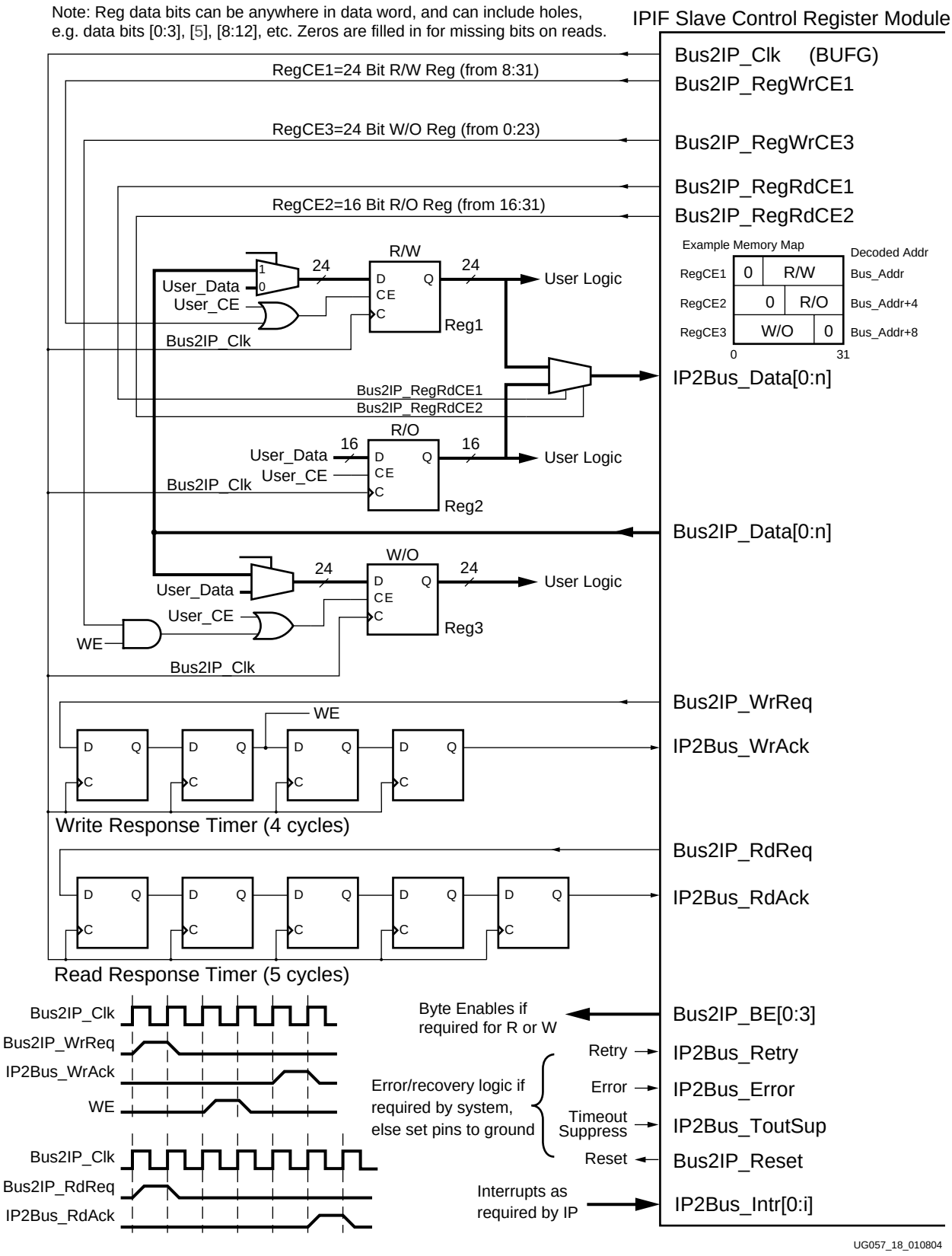


Figure 6-10: **Example Application of an IPIF Slave Control Register Module**

It should be noted that the IPIF Slave Control Register module does not inherently know about the read/write ability of any register that is connected to it. It is how the various registers are connected that allows for read/write, read only, or write only behavior. In addition, the width of the registers, size of the registers, etc. are defined during the instantiation of the IPIF Slave Control Register module. See Table 6-4 for further information.

The IPIF Slave Control Register module can never generate both read and write requests at the same time because the bus, unlike the Processor Local Bus, does not permit simultaneous read and write transactions. Accordingly, the IP logic can be developed to look at the read and write sides independently without and fear of collision between the two sides.

Figure 6-10 illustrates one simple means of answering the request for service by the IPIF Slave Control Register module. When a bus address hit (in the ranges defined by parameters) is detected in this module, the module will generate either **Bus2IP_WrReq** or **Bus2IP_RdReq** requesting write or read service. The connecting IP is required to issue an acknowledge when it has completed the transaction. In the case of writes, the **IP2Bus_WrAck** signal indicates that the control register has been properly written. Reads utilize the **IP2Bus_RdAck** signal and present data on the **IP2Bus_Data** bus during the same cycle that the acknowledge is valid. Also shown in Figure 6-10 is a simple shift register chain that allows a four-cycle write time and a five-cycle read time. For very fast systems, the requests can be tied directly to the acknowledge signals, assuming that the data can be dealt with in the given time. By using FPGA SRL16 elements, the shift registers can be built in a single look-up table (LUT) instead of using a multitude of LUT flip-flops. For more information on how to do this, see Chapter 2 of the Virtex-II Platform FPGA User Guide (UG002).

The IPIF Slave Control Register module also provides the ability to write only certain bytes of the registers. The **Bus2IP_BE** byte enable signals can be used to qualify which bytes are valid during the transfer. Figure 6-10 does not illustrate this usage, however, the byte enables can simply be used as additional qualifier terms in the clock enable term of the register.

All IPIF slave modules, including the Control Register module, permit the IP to tell the bus to retry, or that an error has occurred. The IP logic tells the IPIF this by asserting the **IP2Bus_Retry** or **IP2Bus_Error** signal, with or without the appropriate acknowledge signal for the intended cycle. Additionally, the IP may suppress the normal timeout (16 bus clocks) mechanism for transfers that will take longer on the bus. The use of the **IP2Bus_ToutSup** signal is not recommended unless absolutely necessary, since it reduces bus bandwidth.

Generic Slave Control Register Model

Figure 6-11 illustrates a generic instance of the IPIF Slave Control Register module. The data bus widths, number of registers, number of byte enables, and number of interrupts for the IP are generically specified.

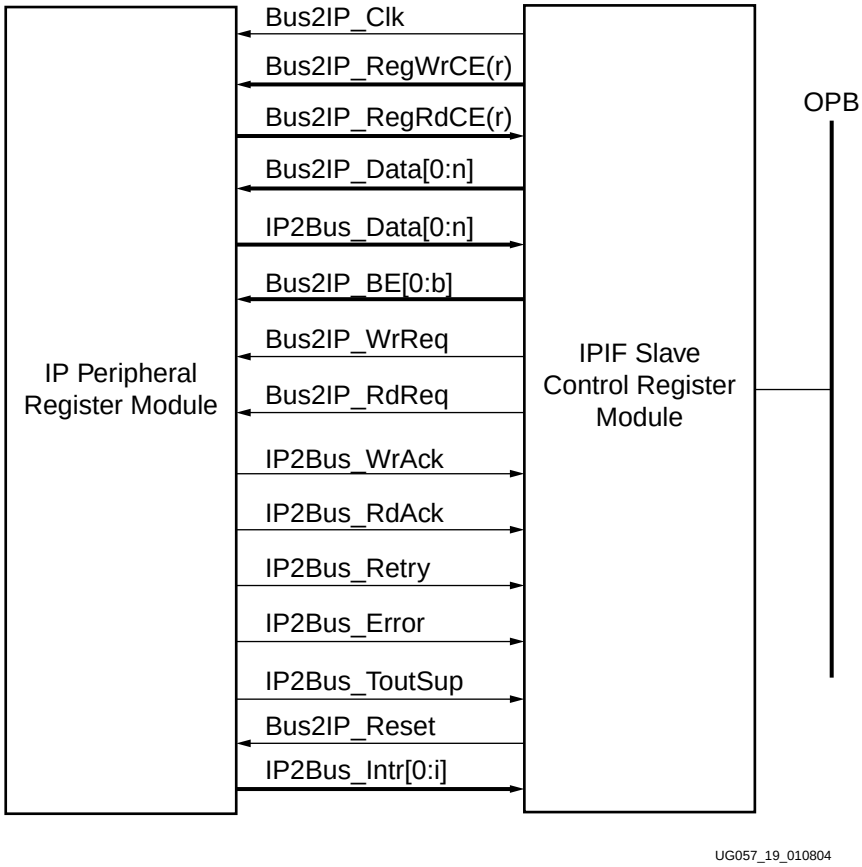


Figure 6-11: **Generic Instance of the IPIF Slave Control Register**

The IPIF Slave Control Register module outputs a clock, **Bus2IP_Clk**, that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP_Clk**. This clock can be asynchronous from the bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

The registers generate a unique decode off the internal address decoder of the IPIF. The number of bits available for each register depends on the data width of the IPIF module. For example, if an 8-bit IPIF Slave Control Register module is instantiated, all registers are limited to a maximum of 8 bits. Each register can be from 0 to 8 bits in this case. The 0 case seems odd at first; however, it may be advantageous for software to “skip” an address location, therefore, 0-bit registers are allowed. Based upon the size of the data bus, the IPIF Slave Control Register module assigns address decodes for the **Bus2IP_RegWrRdCE** signals. If the data bus is 8 bits, then each **Bus2IP_RegWrRdCE** will be up to a byte wide, and will start at the base address of the IPIF Slave Control Register module, and increment by one. If the data bus is 16 bits, then each **Bus2IP_RegWrRdCE** will be up to 16 bits wide, and will increment by two from the base address. Similarly, for 32-bit data bus, each register decode is offset by four from the base address.

The indexing of the **Bus2IP_RegWrCE** or **Bus2IP_RegRdCE** is based upon the bus size. Regardless of whether the bus is 8 bits or 32 bits, **Bus2IP_RegWrRdCE** will increment by one for each instantiated register. There will be holes for zero-bit registers, however. (For

example, if register 3 is zero bits and register 4 is 12 bits, no register 3 clock enable decodes are provided.)

The IPIF Slave Control Register module allows the IP implementor to emplace only the register bits required for the function. This minimizes the logic required for the implementation of registers. Since undriven read bits left are forced to a zero inside the IPIF module, even data multiplexing requirements are eased.

Control over this flexibility is represented as an array of four basic parameters. Each register has control over the total number of bits (0 to 32 data bus width) that can be contained within the register. Additionally, each register can identify the bit-wide position of every bit in the register. Each bit position can also be marked as readable and/or writeable by the IPIF.

Slave Control Register Signal Protocol

Figure 6-12 illustrates the IPIF Slave Control Register module protocol, highlighting the simplicity of both read and write transactions. Note that all signals are active-high true, and are referenced to the positive edge of the **Bus2IP_Clk** signal. These states are generally more favorable to FPGA logic implementation.

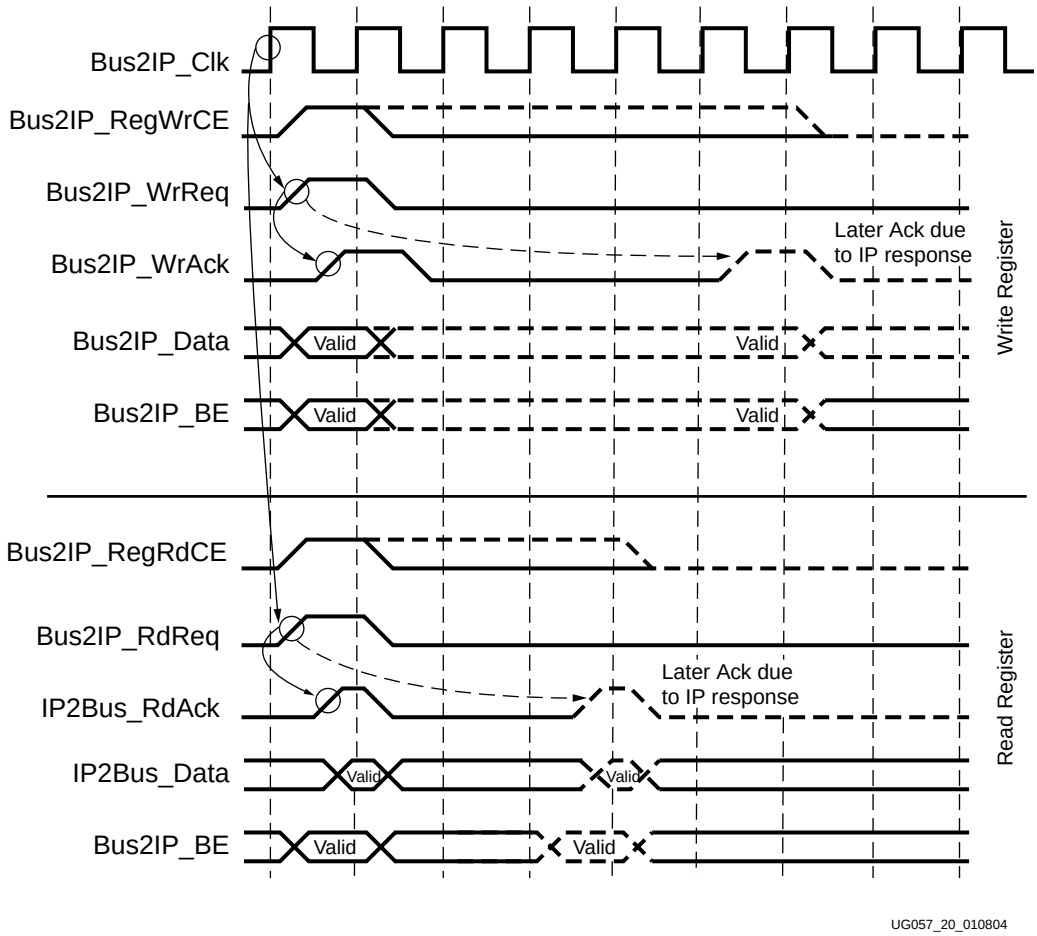


Figure 6-12: **Timing Diagrams for Slave IP Module Register Protocol Read and Write**

Write Transactions

Write transactions begin with the IPIF Slave Control Register issuing a **IP2Bus_WrReq** on a positive edge of **Bus2IP_Clk**. The data to be written is presented on **Bus2IP_Data**, qualified by **Bus2IP_RegRdWrCE(r)** to indicate a register IPIF cycle, and is byte-qualified by the **Bus2IP_BE** signals. (If high, the byte lane is valid; if low, ignore the data.) When the IP has correctly stored the data, the **IP2Bus_WrAck** signal is asserted. Two cases are illustrated in Figure 6-12: immediate acknowledgement, and extended **IP2Bus_WrAck** by five clock cycles. The limit to issuing **IP2Bus_WrAck** is approximately 10 cycles(1) after **Bus2IP_WrReq**, unless the **IP2Bus_ToutSup** signal is held high. If **IP2Bus_ToutSup** is used, it must go low in the same cycle that **IP2Bus_WrAck** transitions from high to low.

Note: Actual number of cycles to be determined.

Read Transactions

Read transactions are very similar to write transactions in the IPIF Slave Control Register module. Reads utilize the **Bus2IP_RdReq** and **IP2Bus_RdAck** counterparts to the write signals and **Bus2IP_RegRdCE(r)**. The data must be presented to the **IP2Bus_Data** bus during the cycle that **Bus2IP_RdAck** is valid. The **Bus2IP_BE** signals can be used to mask read data values, but this is not required. The master that generated the transaction on the bus is always required to pull only the data it wants from the bus. The **Bus2IP_BE** signals are generally only used for writes.

Slave Control Register Signal List

Table 6-3 shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave Control Register module.

Table 6-3: **Signals for the IPIF Slave Control Register Module**

Name	Dir	Description
Bus2IP_Clk	From IPIF	Clock source (from global buffer)
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF
IP2Bus_Error	To IPIF	Error signal from IP to IPIF (valid only during a data acknowledge cycle)
IP2Bus_Retry	To IPIF	Indicates IP wants master to retry the cycle
IP2Bus_ToutSup	To IPIF	Forces the suppression of watch dog timeout on the bus
Bus2IP_Data[0:n]	From IPIF	IPIF Write data (where n = IPIF_DATA_BUS_WIDTH –1)
IP2Bus_Data[0:n]	To IPIF	IPIF Read data (where n = IPIF_DATA_BUS_WIDTH –1)
Bus2IP_BE[0:b]	From IPIF	Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)
Bus2IP_WrReq	From IPIF	Write request from IPIF to IP, single clock high

Table 6-3: **Signals for the IPIF Slave Control Register Module *(Continued)***

Name	Dir	Description
IP2Bus_WrAck	To IPIF	Acknowledge that write data has been taken from **Bus2IP_Data[0:n]**, single **Bus2IP_Clk** high
Bus2IP_RdReq	From IPIF	Read request from IPIF to IP, single clock high
IP2Bus_RdAck	To IPIF	Acknowledge that read data has been placed on **IP2Bus_Data[0:n]**, single **Bus2IP_Clk** high
Bus2IP_RegCE(r)	From IPIF	Clock enable of decoded "r" register (where r = 0 to IPIF_NUMBER_OF_REGS –1)

Slave Control Register Parameters

Table 6-4 shows the parameters that can be selected for the IPIF Slave DMA Handshake module.

Table 6-4: **IPIF Slave Control Register Parameters**

Affects	Parameter	Value	Type
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean

Table 6-4: **IPIF Slave Control Register Parameters *(Continued)***

Affects	Parameter	Value	Type
General IPIF Slave Control Register Module	IPIF_NUMBER_OF_REGS Sets the total number of registers at the bus width set by IPIF_DATA_BUS_WIDTH	1 to 255	number
	IPIF_REG_BASE_ADDR Sets the base address where IPIF registers will start in memory	32-bit decode	number
	IPIF_REG_BASE_ADDR_BIT_ENBL[0:31] Allows specification of which address bits to decode in IPIF_REG_BASE_ADDR	1 = decode respective address bit	mask
IPIF Slave Control Register (Arrays of Parameters)	IPIF_REGx_NUMBER_OF_BITS Defines the number of bits for each register (where x = 0 to IP_NUMBER_OF_REGS –1)	ordinal between 0 and 32	number
	IPIF_REGx_DATA_BIT_VALID_MASK[0:n] Defines which bits in each register are physically present (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH -1)	1 = bit position is used in this register	mask
	IPIF_REGx_READABLE_BITS[0:n] Defines which bits in each register are readable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH -1)	1 = bit position will be readable by IPIF	mask
	IPIF_REGx_WRITEABLE_BITS[0:n] Defines which bits in each register are writable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH -1)	1 = bit position will be writable by IPIF	mask

Slave SRAM Module

This section covers the following topics:

- "Example Slave SRAM Application"
- "Generic Slave SRAM Model"
- "Slave SRAM Signal Protocol"
- "Slave SRAM Signal List"
- "Slave SRAM Parameters"

The IPIF Slave SRAM module is virtually identical to the IPIF Slave Control Register module. The primary difference is that the IPIF Slave SRAM module also includes an address, and only provides a single clock enable for the range of valid addresses. The similarity between the two interfaces is intentional. It is recognized that many forms of IP require both registers and random accessible storage space. Accordingly, the IPIF Slave SRAM module and IPIF Slave Control Register module can be easily combined. In high-performance systems, a single IPIF Slave SRAM module per IP is necessary; whereas in lower performance systems, several IPs can share one IPIF Slave SRAM module.

Example Slave SRAM Application

Figure 6-13 illustrates an example application of the IPIF Slave SRAM module. This example assumes use of a 16-bit wide asynchronous SRAM with some additional logic surrounding it to handle byte data, and setting of the read and write cycle times. While this example shows connection to a real SRAM, the IPIF Slave SRAM module can be connected to any IP that requires a multitude of addresses for its operation. For example, an existent disk drive controller whose interface is a microprocessor memory mapped device can take advantage of this IPIF module with little modification.

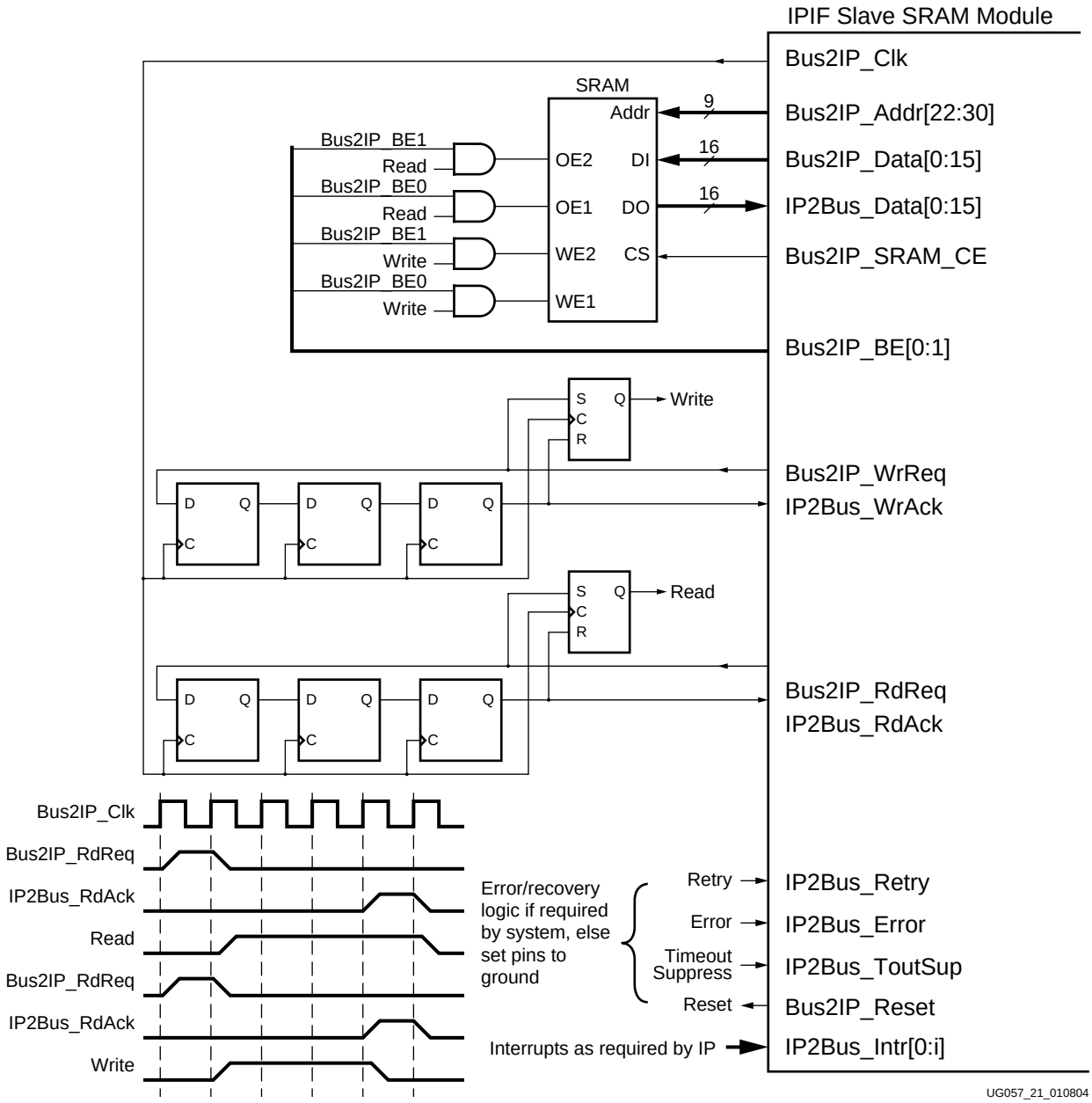


Figure 6-13: **Example Application of IPIF Slave SRAM Module**

The **Bus2IP_Addr** bus is used to broadcast the address of the bus Slave transaction to the SRAM. The SRAM in this example is organized as 512 x 16 or 1 KB total. In this case, the

byte address is the lower 10 bits of the address, **Bus2IP_Addr[22:31]**. Since the example uses half words (16 bit quantities) and the IPIF Slave SRAM module always provides byte addresses, the lower order address line is left unconnected, leaving the next nine address lines to provide the 512 half words.

Since most systems require the ability to read and write individual bytes of data from an SRAM, the IPIF Slave SRAM module provides byte enables by way of the **Bus2IP_BE** signals. In the example in Figure 6-13, these byte enables are used to qualify both writes and reads by way of the active-high WE and OE pins. This permits the SRAM to be read and written a single byte at a time, if required. Byte enable usage on read cycles is not required, since the IPIF module will automatically handle the read data alignment. (More information on data alignment will appear in the IPIF Architectural Specification, *not yet released*.)

This example also shows that the **Bus2IP_SRAM_CE** pin is connected directly to the active-high chip select (CS) pin on the SRAM. This example assumes that the SRAM CS pin qualifies the **OE** and **WE** signals. If the SRAM does not use CS to qualify the **OE** and **WE** signals, the **Bus2IP_SRAM_CE** signal must qualify both reads and writes. This is a particular issue that warrants attention when the IPIF Slave SRAM module is used to talk with several ranges of memory which may be discrete SRAMs or IPs.

Figure 6-13 also demonstrates a simple technique to generate both the acknowledgement responses and a set of enveloping signals. The acknowledgement of read and write cycles is handled by delaying the respective request signal by a known number of clocks (three in this example). This is very efficient to implement in a single FPGA look-up table (LUT) by utilizing an SRL16 model. The SRL16s allow the customer to instantiate a 1- to 16-bit shift register at the price of a single LUT.

In addition to providing the **IP2Bus_WrAck** and **IP2Bus_RdAck**, the outputs of the shift register can be used to build a simple enveloping signal. The write and read signals illustrated in the figure are created by flip-flops that are set by **Bus2IP_WrReq** or **Bus2IP_RdReq**, respectively. They are reset by the **IP2Bus_WrReq** or **IP2Bus_RdAck** signals, respectively. This provides a single clock delayed envelope that is valid for the transaction, and can be used to enable specific read or write operations. The read and write signals are used to qualify the **OE** and **WE** inputs to the SRAM. The write signal can also be derived as a single cycle delayed clock enable, if necessary to meet data hold time requirements (e.g., another tap can be used).

Generic Slave SRAM Model

Figure 6-14 illustrates a generic instance of the IPIF Slave SRAM module. The data bus widths, number of address bits, number of byte enables, and number of interrupts for the IP are generically specified.

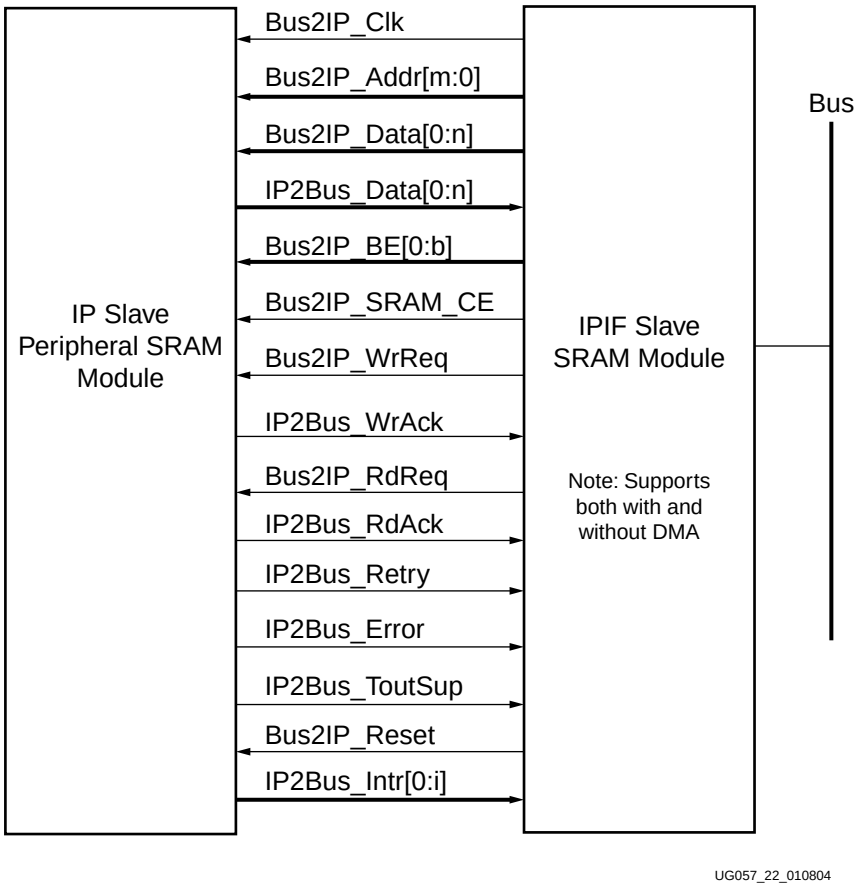


Figure 6-14: **Slave IP Module SRAM Protocol Block Diagram**

The IPIF Slave SRAM module outputs a clock, **Bus2IP_Clk**,that is sourced by a BUFG elsewhere in the design. The required BUFG reduces clock skew on all the synchronous elements clocked by **Bus2IP_Clk**. This clock can be asynchronous from the bus, but requires extra synchronization in the IPIF module. This synchronization is provided for in the IPIF module.

The number of address locations covered by the IPIF Slave SRAM interface, and the number of address bits required to support those address locations is specified during instantiation of the module.

The data width for the IPIF Slave SRAM interface, illustrated generically in Figure 6-14, can be configured as 8, 16 or 32 bits. The *n* variable in the figure is always subtracted by one from the number of bits specified by the IP_DATA_BUS_WIDTH parameter.

Figure 6-14 also shows the active-high signal, **Bus2IP_SRAM_CE**. This signal indicates that the address presented on **Bus2IP_Addr** is valid. Since the regions of memory decoded by the IPIF Slave SRAM module are parameterizeable, it is possible that the **Bus2IP_SRAM_CE** may be valid in discontinuous regions. The bus addresses that will

initiate the transaction on the IPIF Slave SRAM module are set during the instantiation of the module by the parameters defined in Table 6-6.

The remaining signals illustrated in Figure 6-14 are illustrated by example in Figure 6-13.

Slave SRAM Signal Protocol

Figure 6-15 and Figure 6-16 illustrate the IPIF Slave SRAM module protocol for write and read cycles, respectively. All signals shown in both figures are active-high true, and are referenced to the positive edge of the Bus2IP_Clk signal. These states are often most favorable to FPGA logic implementation.

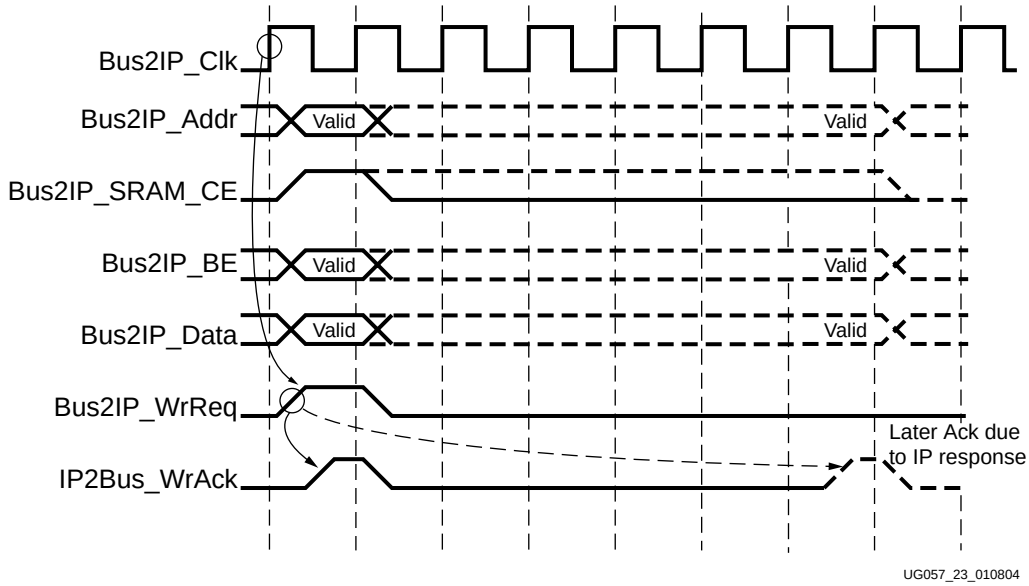


Figure 6-15: **IPIF Slave SRAM Module Single Write Transaction**

Write Transactions

Write transactions begin with the IPIF Slave SRAM module issuing a **IP2Bus_WrReq** on a positive edge of **Bus2IP_Clk**. The data to be written is presented on **Bus2IP_Data**, qualified by **Bus2IP_SRAM_CE** to indicate an SRAM cycle, and is byte qualified by the **Bus2IP_BE** signals. (High = data in byte lane is valid; low = ignore data.) The address of the write transfer is presented on the **Bus2IP_Addr** bus on the same rising edge of the clock that generated **Bus2IP_WrReq**.

When the IP has correctly stored the data, it asserts the **IP2Bus_WrAck** signal. Figure 6-15 illustrates two cases. The first case is immediate acknowledgement, and the second case extends the **IP2Bus_WrAck** out six clock cycles. The limit to issuing **IP2Bus_WrAck** is 10 cycles after the **Bus2IP_WrReq**, unless the **IP2Bus_ToutSup** signal is held high. If **IP2Bus_ToutSup** is used, it must go low at the same time that **IP2Bus_WrAck** transitions from high to low.

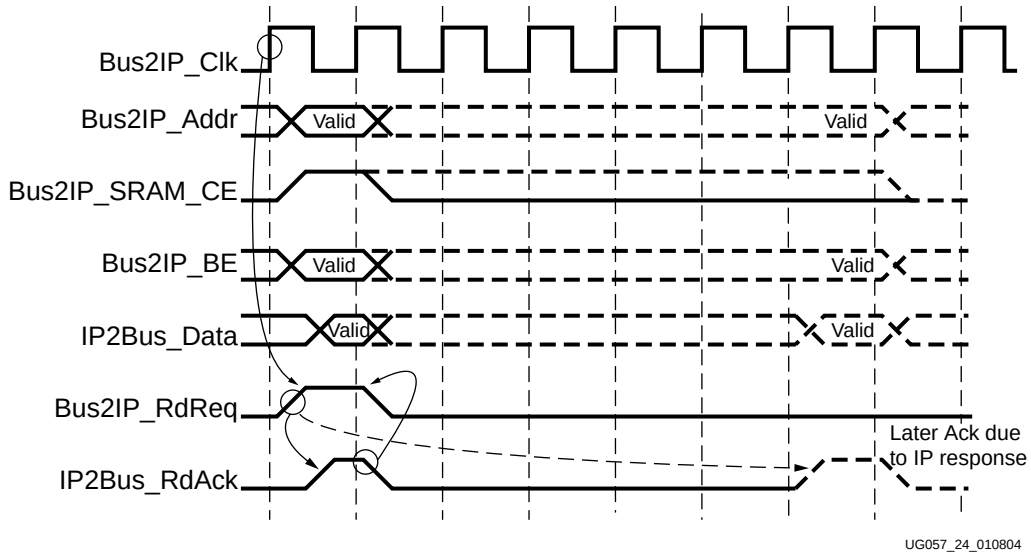


Figure 6-16: **IPIF Slave SRAM Module Single Read Transaction**

Read Transactions

Read transactions are very similar to write transactions in the IPIF Slave SRAM module. Reads utilize the **Bus2IP_RdReq** and **IP2Bus_RdAck** counterparts to the write signals. Data must be presented to the **IP2Bus_Data** bus during the cycle that **Bus2IP_RdAck** is valid. The **Bus2IP_BE** signals can be used to mask read data values, but this is not required. The master that generated the transaction on the bus is always required to pull only the data it wants from the bus. The **Bus2IP_BE** signals are generally used only for writes.

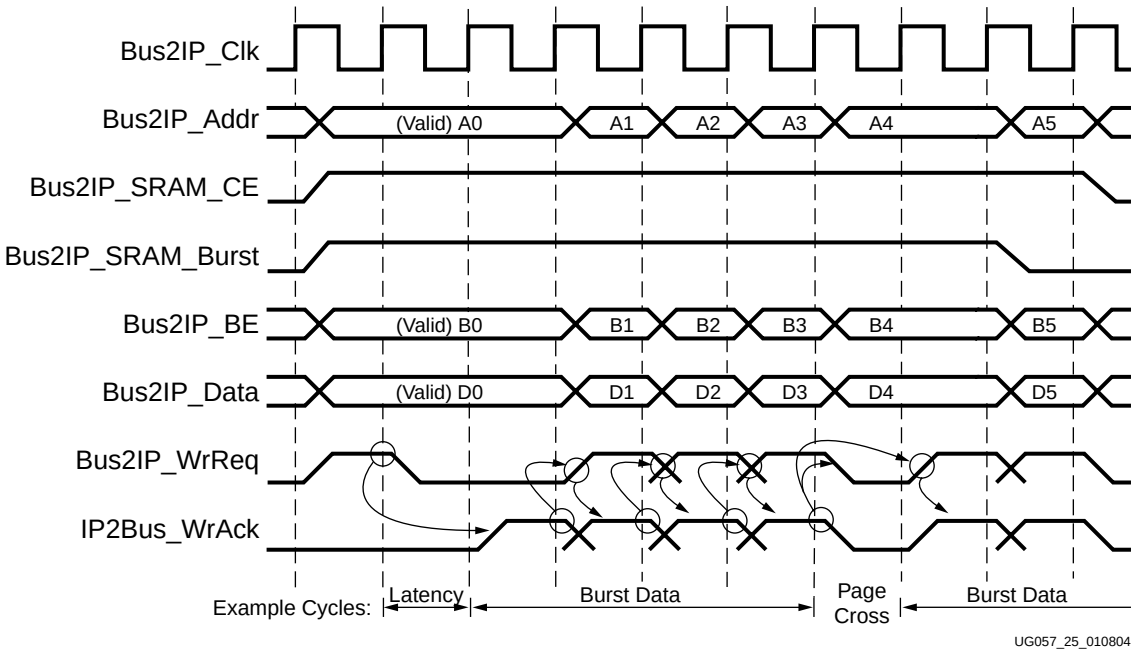


Figure 6-17: **IPIF Slave SRAM Module Write Burst Transaction**

Write Burst Transactions

Figure 6-17 illustrates the IPIF Slave SRAM module protocol for write burst cycles. Bursts from a bus master are treated as individual requests for service per datum. That is, the IPIF Slave SRAM module issues a set of requests acknowledges per datum. If the IP data bus is smaller, then the IPIF will sequence the data in the order it was received. Thus, a 32-bit bus transfer to an 8-bit IP is turned into four req/ack pairs.

Write bursts are designed to utilize maximum bandwidth across the IPIF. It is wisest to burst only to 32-bit IP devices, or else poor bus bandwidth will result. In the case of Figure 6-17, it is assumed that the IP is a 32-bit device. Note that the IPIF indicates an SRAM burst cycle by generating Bus2IP_Addr, asserting Bus2IP_SRAM_CE, and issuing **Bus2IP_SRAM_Burst** and **Bus2IP_WrReq** all in the same cycle. The IP then acknowledges the cycle by way of the **IP2Bus_WrAck** signal after a 1-cycle latency. The IP continues to acknowledge the next three cycles. In the next cycle, the IP chooses to throttle the transaction by deasserting the **IP2Bus_WrAck** signal. If the **IP2Bus_WrAck** signal is deasserted for more than 16 clocks, the bus will time out, unless the IP also asserts the **IP2Bus_ToutSup** signal, shown elsewhere. The IP eventually completes the data transfers, and the IPIF deasserts the **Bus2IP_WrReq**. The IP will then deassert **IP2Bus_WrAck**. Note that data is always transferred on the rising edge of the **Bus2IP_Clock**, which sampled the **IP2Bus_WrAck** as high.

Also note that Bus2IP_SRAM_Burst is deasserted after the second-to-last data is transferred. This allows the IP to know which datum is last in the burst. In effect, there is always only one more **IP2Bus_WrAck** after **Bus2IP_SRAM_Burst** is deasserted.

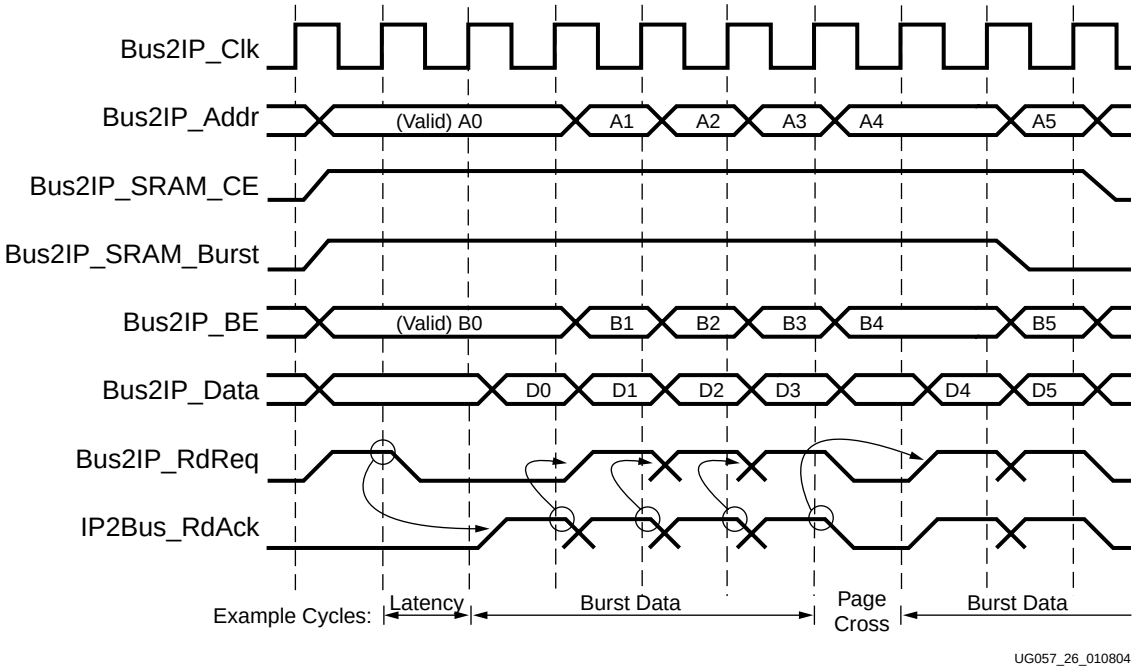


Figure 6-18: **IPIF Slave SRAM Module Read Burst Transaction**

Read Burst Transactions

Figure 6-18 illustrates the IPIF Slave SRAM module protocol for read burst cycles. Bursts to a bus master are treated as individual requests for service per datum. That is, the IPIF Slave SRAM module issues a set of request acknowledges per datum. If the IP data bus is

smaller, then the IPIF sequences the data in the order it is received. Thus, a 32-bit bus transfer to an 8-bit IP gets turned into four req/ack pairs.

Read bursts are designed to utilize maximum bandwidth across the IPIF. It is wisest to only burst to 32-bit IP devices, or else poorer bus bandwidth will result. In the case of Figure 6-18, it is assumed that the IP is a 32-bit device. Note that the IPIF indicates an SRAM burst cycle by generating **Bus2IP_Addr**, asserting **Bus2IP_SRAM_CE** and issuing **Bus2IP_SRAM_Burst** and **Bus2IP_RdReq** all in the same cycle. The IP then acknowledges the cycle by way of the **IP2Bus_RdAck** signal after a 1-cycle latency. The IP continues to acknowledge the next three cycles. The IP throttles the transaction, for example, by deasserting the **IP2Bus_RdAck** signal. If this signal is deasserted for more than 16 clocks, the bus will time out, unless the IP also asserts the **IP2Bus_ToutSup** signal. The IP eventually completes the data transfers, the IPIF deasserts the **Bus2IP_RdReq**, and the IP deasserts **IP2Bus_RdAck**. Note that data is always transferred on the rising edge of the **Bus2IP_Clk**, which sampled the **IP2Bus_RdAck** as high.

Also note that **Bus2IP_SRAM_Burst** is deasserted after the second-to-last data is transferred. This allows the IP to know which datum is last in the burst. In effect, there is always only one more **IP2Bus_WrAck** after **Bus2IP_SRAM_Burst** is deasserted.

Other SRAM Module Uses

The IPIF Slave SRAM module can be used in many more forms than are illustrated in this specification. For example, it can be used as a simple external bus controller to talk to SRAM, FLASH, or external IP peripherals. Additionally, if a system is not performance intensive, it can be used to connect to all IP CPU peripherals. Since the address ranges of the IPIF module are adjustable, and multiple ranges are possible, this is easily achieved by adding an address decoder between the IPIF **Bus2IP_Addr** lines (qualified by **Bus2IP_SRAM_CE**) and the IP peripherals.

Slave SRAM Signal List

Table 6-5 shows the name, direction, and a brief description of the signals that connect to the IP from the IPIF Slave Control Register module.

Table 6-5: **Signals for the IPIF Slave SRAM Module**

Name	Direction	Description
Bus2IP_Clk	From IPIF	Clock source (from global buffer)
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF
IP2Bus_Error	To IPIF	Error signal from IP to IPIF (Valid only during a data acknowledge cycle)
IP2Bus_Retry	To IPIF	Indicates IP wants master to retry the cycle
IP2Bus_ToutSup	To IPIF	Forces the suppression of watch dog timeout on the bus
Bus2IP_Data[0:n]	From IPIF	IPIF Write data (where n = IPIF_DATA_BUS_WIDTH –1)
IP2Bus_Data[0:n]	To IPIF	IPIF Read data (where n = IPIF_DATA_BUS_WIDTH –1)

Table 6-5: **Signals for the IPIF Slave SRAM Module *(Continued)***

Name	Direction	Description
Bus2IP_BE[0:b]	From IPIF	Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)
Bus2IP_WrReq	From IPIF	Write request from IPIF to IP, single clock high
IP2Bus_WrAck	To IPIF	Acknowledge that write data has been taken from **Bus2IP_Data[0:n]**, single **Bus2IP_Clk** high
Bus2IP_RdReq	From IPIF	Read request from IPIF to IP, single clock high
IP2Bus_RdAck	To IPIF	Acknowledge that read data has been placed on **IP2Bus_Data[0:n]**, single **Bus2IP_Clk** high
Bus2IP_Addr[al:ah]	From IPIF	IPIF Slave SRAM address, where: al = IPIF_SLV_SRAM_ADDR_BUS_MSB ah = IPIF_SLV_SRAM_ADDR_BUS_LSB (*al* is a lower number than *ah* due to big-endian numbering)
Bus2IP_SRAM_CE	From IPIF	Clock enable of decoded SRAM address space, high for entire bus cycle

Slave SRAM Parameters

Table 6-6 shows the parameters that can be selected for the IPIF Slave SRAM module..

Table 6-6: **Parameters for the IPIF Slave SRAM Module**

Affects	Parameter	Value	Type
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean

Table 6-6: **Parameters for the IPIF Slave SRAM Module *(Continued)***

Affects	Parameter	Value	Type
General IPIF Slave SRAM Module	IPIF_NUMBER_OF_SRAM_DECODER_REGIONS Defines the number of decoded regions of memory address space for the IPIF Slave SRAM module	1 to 4 decoded regions allowed	number
	IPIF_SLV_SRAM_ADDR_BUS_LSB Sets the LSB index number for the address provided by the IPIF to the IP	ordinal between 0 to 31	number
	IPIF_SLV_SRAM_ADDR_BUS_MSB Sets the MSB index number for the address provided by the IPIF to the IP	ordinal between 0 to 31	number
IPIF Slave SRAM Module (Arrays of Parameters)	IPIF_SRAMd_BASE_ADDR Sets the base address of the "d" region for the Slave SRAM module (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1)	32-bit decode of one region	number
	IPIF_SRAMd_BASE_ADDR_BIT_ENBL[0:31] Allows specification of which bits address bits to decode in the IPIF_SRAMd_BASE_ADDR (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1)	1 = decode respective address bit	mask

Slave FIFO Module

This section covers the following topics:

- "Example Slave FIFO Application"
- "Generic Slave FIFO Model"
- "Slave FIFO Signal Protocol"
- "Slave FIFO Signal List"
- "Slave FIFO Parameters"

The IPIF Slave FIFO module is designed to interface between basic communications devices and a bus. The module looks substantially like a bidirectional FIFO. The IPIF Slave FIFO module provides a set of signals to the IP that permit the IP to access the module's internal FIFOs. These FIFOs can be configured from one datum deep up to 2 KB deep.

Example Slave FIFO Application

Figure 6-19 illustrates a simplified example application of the IPIF Slave FIFO module, showing a vastly simplified communication IP with both transmit and receive functions. Each of the functions are connected to the IPIF by way of the IPIF Slave FIFO module.

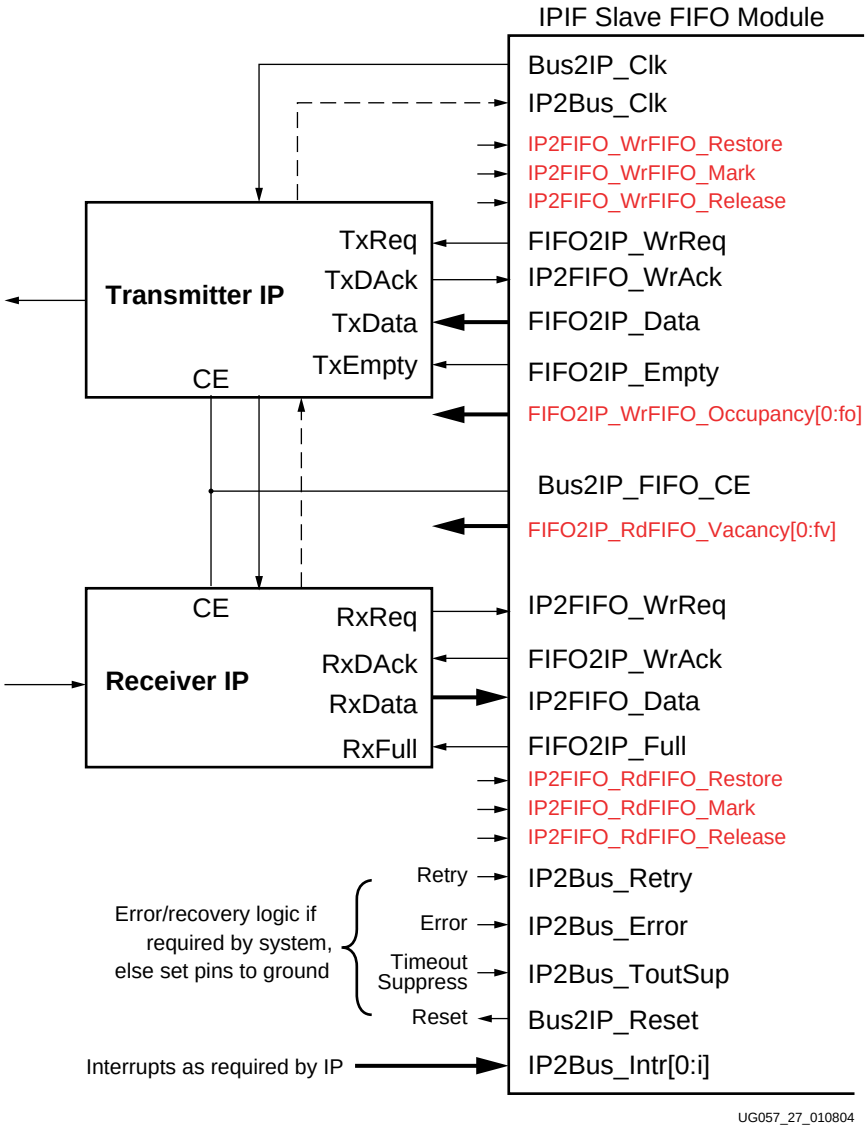


Figure 6-19: **Example Application of IPIF Slave FIFO Module**

The generic IP communication transmitter and receiver pair shown in Figure 6-19 are connected to the IPIF module's write FIFO and read FIFO respectively. The transmitter IP waits for data requests from the IPIF module. The receiver IP initiates requests for transfers when data is pending inside its receive buffer.

The the transmitter IP in this example has signals such as:

- **TxReq**, a transmit request input
- **TxDAck**, a transmit data acknowledge output
- **TxData**, a transmit data input
- **TxEmpty**, a transmit empty flag input (indicating the IPIF has no more data to transmit)

The receiver IP in this example has signals such as:

- **RxReq**, a receive request output
- **RxDAck**, a receive data acknowledge input
- **RxData**, a receive data output
- **RxFull**, a receiver full flag input (indicating the IPIF can no longer accept data from the receiver IP)

Both transmitter IP and receiver IP may also require a clock enable (CE) to indicate when cycles are for them rather than another IP block which might be connected.

The transmitter and/or receiver can be clocked from the **Bus2IP_Clk** output of the IPIF Slave FIFO module. While this is recommended, the transmitter and/or receiver IPs can also generate a clock, **IP2FIFO_Clk**, which is used to control the FIFO interfaces between the IPIF and the IP.

In Figure 6-19, when a bus master wishes to transmit data to the IP, it issues a write cycle to the IPIF Slave FIFO module. The address of this operation can be set by way of user parameters, and can be either a range of addresses spanning the depth of the FIFO, or can be a single "keyhole" address. If the IPIF Slave FIFO module detects an address hit on a write cycle, it can issue a request for service to the transmitter. The number of bus write cycles to the IPIF Slave FIFO module, before **FIFO2IP_WrReq** goes valid, is parametrically settable at the time of instantiation of the IPIF module. (See Table 6-8.) Once the IPIF Slave FIFO contains a single datum, the **FIFO2IP_Empty** flag is lowered to indicate that the IPIF has data in the write FIFO. When the IPIF Slave FIFO module has hit the write data threshold in its write FIFO, it will issue write requests until the transmitter IP causes the IPIF FIFO to go empty. Should the IPIF Slave FIFO module attempt to overflow the transmitter IP, the transmitter simply stops giving back **IP2FIFO_WrAck** until the condition is cleared. If the transmitter is stalled so long that the internal FIFO in the IPIF fills, then bus retry is issued until the condition clears.

Additional signals are also illustrated in Figure 6-19 which are available if the optional Packet mode of the Slave FIFO module is used. These signals are designed to add packet level awareness to the FIFO. For each FIFO, three basic signals are provided: mark, restore, and release. Vacancy or Occupancy level signals are also provided so that the Tx and Rx IP can know how much data is present in the FIFO of the IPIF.

The function of the mark, restore, and release pins is illustrated using the Write FIFO as context. The Read FIFO works identically. The purpose behind the **IP2FIFO_WrFIFO_Mark** signal is to mark the beginning of a packet of data. This mark causes the Slave FIFO module to memorize where in the FIFO the datum that was marked resides. This permits the IP to request a retransmit of the packet from the FIFO via the **IP2FIFO_WrFIFO_Restore** signal, should it be required. Once a packet has been known to be properly transmitted, or past a point where the data for the packet is committed, then the IP can issue the **IP2FIFO_WrFIFO_Release** signal to release the contents of the FIFO to accept new data. The overall effect of these signals is to permit smarter transmission of data between the bus and the IP. In addition to the mark, restore, and release signals, the Write FIFO also provides a set of signals which indicate how much data is present in the FIFO. This is done via the **FIFO2IP_WrFIFO_Occupancy[0:fo]** signals, where **fo** represents the index of bits required to contain the count of data. Similarly, the read FIFO has a set of vacancy signals, **FIFO2IP_RdFIFO_Vacancy[0:fv]**, where **fv** represents the index of bits required to contain the count of data.

These signals are not available at present for user designs, but may be made available in a future release of the IPIF specification. Accordingly, no timing diagrams are currently presented to illustrate this functionality.

Generic Slave FIFO Model

Figure 6-20 illustrates the generic instance of the IPIF Slave FIFO module. The data bus widths, and number of interrupts for the IP have been generically specified.

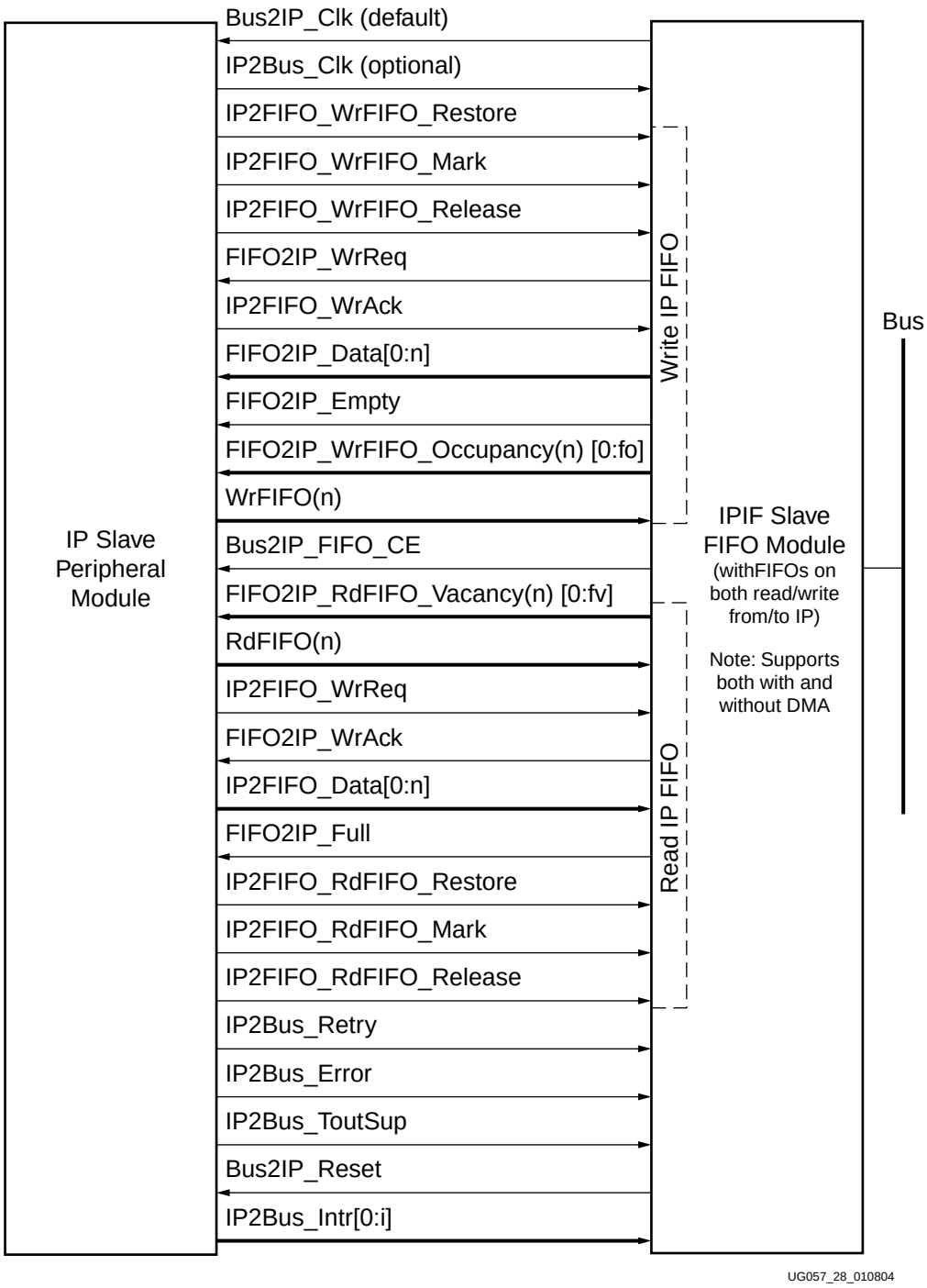


Figure 6-20: **Generic Slave FIFO Module Block Diagram**

The IPIF Slave FIFO module also provides a clock enable signal, **Bus2IP_FIFO_CE**, to indicate to other elements in the system that the FIFO is currently busy. **Bus2IP_FIFO_CE**

is valid anytime that the IPIF module is busy transferring data on the bus, and may not reflect the current values of the write or read FIFOs. Typically, the clock enable is necessary only when using additional IPIF modules.

Slave FIFO Signal Protocol

Figure 6-21 shows a protocol diagram of the IPIF Slave FIFO module, illustrating both reads and writes.

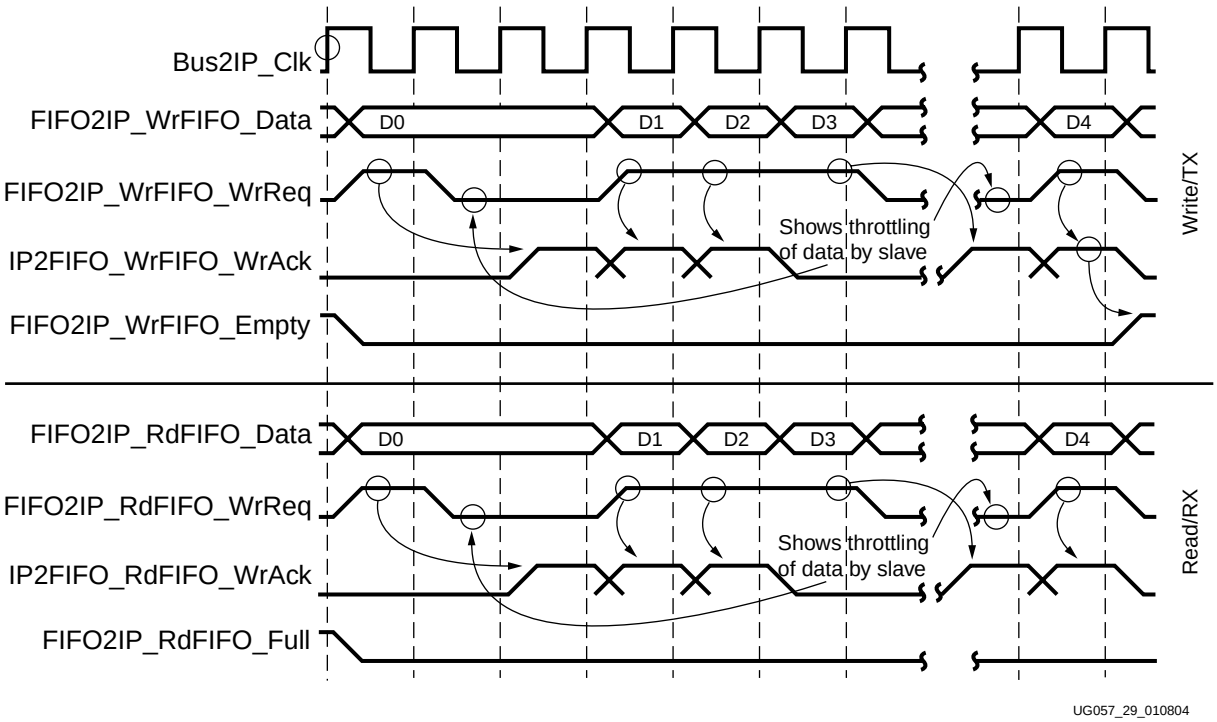


Figure 6-21: **Timing Diagram for Slave IP FIFO Module Read and Write Transactions**

Write Transactions

The write FIFO in the IPIF Slave FIFO module works as the transmit FIFO.

In addition to the write FIFO data, the write FIFO example in Figure 6-21 has the following signals:

- **FIFO2IP_WrFIFO_WrReq**, a write request output (requesting the IPIF to write data to the IP)
- **IP2FIFO_WrFIFO_WrAck**, a write acknowledge input (from the IP to the IPIF, indicating it has taken the FIFO2IP_Data)
- **FIFO2IP_WrFIFO_Empty**, a write FIFO empty output (indicates to the IP that the IPIF has no more data)

Figure 6-21 (top half) illustrates the IPIF Slave FIFO module write protocol. In this diagram, the IP is first notified that there is data in the IPIF's write FIFO by the deassertion of the **FIFO2IP_WrFIFO_Empty** signal. When it transitions from high to low, it indicates that the write FIFO has data in it. Note that the **FIFO2IP_WrFIFO_Data** bus is also now valid, and remains so until the cycle that acknowledges the data. The IPIF Slave FIFO module generates a request to the IP to take data from the IPIF by asserting the

FIFO2IP_WrFIFO_WrReq signal. The IP can acknowledge immediately, or delay as required. The signal for acknowledging the cycle is the **IP2FIFO_WrFIFO_WrAck**. This permits the IP to throttle the data from the IPIF as required. The end of the diagram illustrates emptying the FIFO again.

Read Transactions

The read FIFO of the IPIF Slave FIFO module works as the receive FIFO. In order to allow communications between the IP and the IPIF, four basic signals are provided.

In addition to the read FIFO data, the read FIFO example in Figure 6-21 has the following signals:

- **IP2FIFO_RdFIFO_WrReq**, a write request input (indicating a request by the IP to write data to the IPIF)
- **FIFO2IP_RdFIFO_WrAck**, a write acknowledge output (from the IPIF to the IP, indicating it has taken the **IP2FIFO_Data**)
- **FIFO2IP_RdFIFO_Full**, a read FIFO full output (indicating to the IP that the IPIF is currently full and no other requests will be acknowledged until the FIFO empties)

Figure 6-21 (bottom half) illustrates the IPIF Slave FIFO module read protocol. In this diagram, the IP has issued a request to write data to the IPIF FIFO via the **IP2FIFO_RdFIFO_WrReq** signal. Note that in this case, the IPIF read FIFO was full, indicated by **FIFO2IP_RdFIFO_Full** signal. The IP must not request another data transfer until the full condition has been removed. Since the IPIF Slave FIFO module is busy transferring data on the bus, it will eventually reach a condition where it is no longer full. Once this occurs, the IPIF permits more data transfers. It holds the **FIFO2IP_RdFIFO_WrAck** signal low until the same cycle that the **FIFO2IP_RdFIFO_Full** signal will go low. Then it permits the **FIFO2IP_RdFIFO_Full** to transition to low, and the **FIFO2IP_RdFIFO_WrAck** to transition high to acknowledge the requested data transfer. The data transfers can continue for as long as the IPIF read FIFO is not full. The IP can throttle the transfers by not issuing **IP2FIFO_RdFIFO_WrReq**.

Slave FIFO Signal List

Table 6-7 shows the name, direction, and a brief description of the signals which connect to the IP from the IPIF Slave FIFO module.

Table 6-7: **Signals for the IPIF Slave FIFO Module**

Name	Direction	Description
Bus2IP_Clk	From IPIF	Clock source (from global buffer)
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF
IP2Bus_Error	To IPIF	Error signal from IP to IPIF (Valid only during a data acknowledge cycle)
IP2Bus_Retry	To IPIF	Indicates IP wants master to retry the cycle
IP2Bus_ToutSup	To IPIF	Forces the suppression of watch dog timeout on the bus
IP2Bus_Clk (optional)	To IPIF	Optional clock source to control IPIF Slave FIFO module signals (not yet available
Bus2IP_FIFO_CE	From IPIF	Clock enable of decoded IPIF Slave FIFO address space, high for entire bus cycle

Table 6-7: **Signals for the IPIF Slave FIFO Module *(Continued)***

Name	Direction	Description
FIFO2IP_WrFIFO_Data[0:n]	From IPIF	IPIF write FIFO data (where n = IPIF_WR_FIFO_DATA_WIDTH –1)
FIFO2IP_WrFIFO_Empty	From IPIF	IPIF write FIFO is empty when high
FIFO2IP_WrFIFO_WrReq	From IPIF	IPIF write FIFO request, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum
IP2FIFO_WrFIFO_WrAck	To IPIF	IPIF write FIFO acknowledge, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum
IP2FIFO_RdFIFO_Data[0:n]	To IPIF	IPIF read FIFO data (where n = IPIF_RD_FIFO_DATA_WIDTH –1)
FIFO2IP_RdFIFO_Full	From IPIF	IPIF write FIFO is empty when this signal is high
FIFO2IP_RdFIFO_WrReq	To IPIF	IPIF read FIFO request, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum
IP2FIFO_RdFIFO_WrAck	From IPIF	IPIF read FIFO acknowledge, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum
IP2FIFO_WrFIFO_Mark	To IPIF	Marks datum as first datum of packet, high during **IP2FIFO_WrFIFO_WrAck** only
IP2FIFO_WrFIFO_Restore	To IPIF	Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an **IP2FIFO_WrFIFO_WrAck**)
IP2FIFO_WrFIFO_Release	To IPIF	Releases the marked position, write FIFO can now accumulate data from mark forward
FIFO2IP_WrFIFO_Occupancy[0:fo]	From IPIF	Indicates how much data is in the write FIFO (where *fo*=IPIF_WR_FIFO_NUMBER_OF_OCC_BITS –1)
IP2FIFO_RdFIFO_Mark	To IPIF	Marks datum as first datum of packet, high during **IP2FIFO_RdFIFO_WrAck** only
IP2FIFO_RdFIFO_Restore	To IPIF	Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an **IP2FIFO_RdFIFO_WrAck**)
IP2FIFO_RdFIFO_Release	To IPIF	Releases the marked position, read FIFO can now accumulate data from mark forward
FIFO2IP_WrFIFO_Vacancy[0:fv]	From IPIF	Indicates how much data is in the read FIFO (where *fv*=IPIF_WR_FIFO_NUMBER_OF_VAC_BITS –1)

Slave FIFO Parameters

Table 6-8 shows the parameters that can be selected for the IPIF Slave FIFO module.

Table 6-8: **Parameters for the IPIF Slave FIFO Module**

Affects	Parameter	Value	Type
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean

Table 6-8: **Parameters for the IPIF Slave FIFO Module *(Continued)***

Affects	Parameter	Value	Type
General IPIF Slave FIFO Module	IPIF_RD_FIFO_DEPTH Sets the depth of the Read FIFO in IPIF_RD_FIFO_DATA_WIDTH units	1 to 2048, in units of data width	number
	IPIF_RD_FIFO_DATA_WIDTH Sets the width of the Read FIFO data bus	1 to 32 bits	number
	IPIF_RD_FIFO_TRIG_THRESHOLD Sets the trigger point of the Read FIFO if packet processing	1 to 2048	number
	IPIF_WR_FIFO_DEPTH Sets the depth of the Write FIFO in IPIF_WR_FIFO_DATA_WIDTH units	1 to 2048, in units of data width	number
	IPIF_WR_FIFO_DATA_WIDTH Sets the width of the Write FIFO data bus	1 to 32 bits	number
	IPIF_WR_FIFO_TRIG_THRESHOLD Sets the trigger point of the Write FIFO if packet processing	1 to 2048	number
	IPIF_PACKET_FIFO_MODE_ENBL Enables the IPIF Slave FIFO module with packet FIFO functionality	0 or 1, 1 = true	boolean
	IPIF_WR_FIFO_NUMBER_OF_OCC_BITS Sets the total number of bits in the **FIFO2IP_WrFIFO_Occupancy[0:fo]** signals	0 TO 12	number
	IPIF_WR_FIFO_NUMBER_OF_VAC_BITS Sets the total number of bits in the **FIFO2IP_WrFIFO_Vacancy[0:fv]** signals	0 TO 12	number

Master Module

Example Master Application

Figure 6-22, page 99 is a simplified example of an IPIF Master module application and is intended for illustrative and informative purposes only. This example illustrates a basic state machine that monitors a block RAM (BRAM) for a semaphore, and based upon receiving the semaphore initiates a dump from the BRAM across the bus to a slave device.

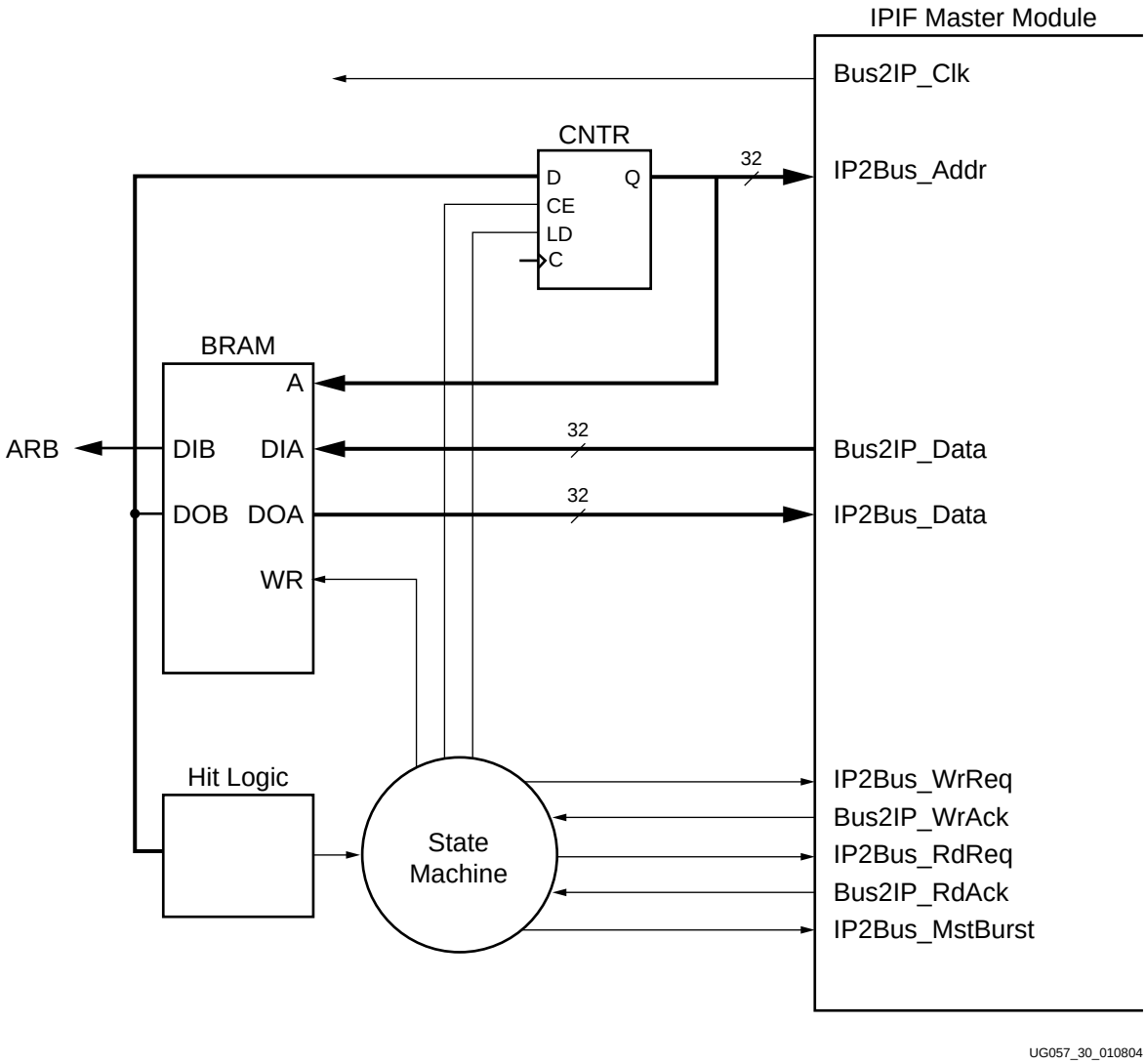


Figure 6-22: **Example Application of IPIF Master Module**

In Figure 6-22, a state machine watches for a semaphore, and then initiates a set of burst cycles across the bus as required to dump the data frame. The **IP2Bus_Addr** signal is generated by a simple 32-bit counter controlled by the state machine. The **IP2Bus_MstWrReq** and **IP2Bus_MstRdReq** signals are also controlled by the state machine. When the state machine is told that the appropriate semaphore is found, it initiates a read or write cycle. The **Bus2IP_MstWrAck** and **Bus2IP_MstRdAck** signals are also brought into the state machine in order to properly sequence the address from the counter and the data to and from the BRAM. The dual port feature of the BRAM allows independence between the IPIF and the rest of the IP.

The address counter generates addresses for the BRAM using the lower-ordered bits. This allows the IPIF to have a range of memory on the bus that coincides with the BRAM. The address counter also provides the address for the IPIF Master module.

The hit logic is connected to the B port of the BRAM and the IPIF Master module is connected to the A port. It is possible to use the BRAM to cross clock boundaries with other

elements of the IP. This is a useful technique for dealing with the often asynchronous nature of communications systems.

Generic Master Model

Figure 6-23 illustrates a general model of the IPIF Master module. This module is used to initiate transactions across the IPIF and onto the bus and is only required for those IPs that need to initiate bus cycles.

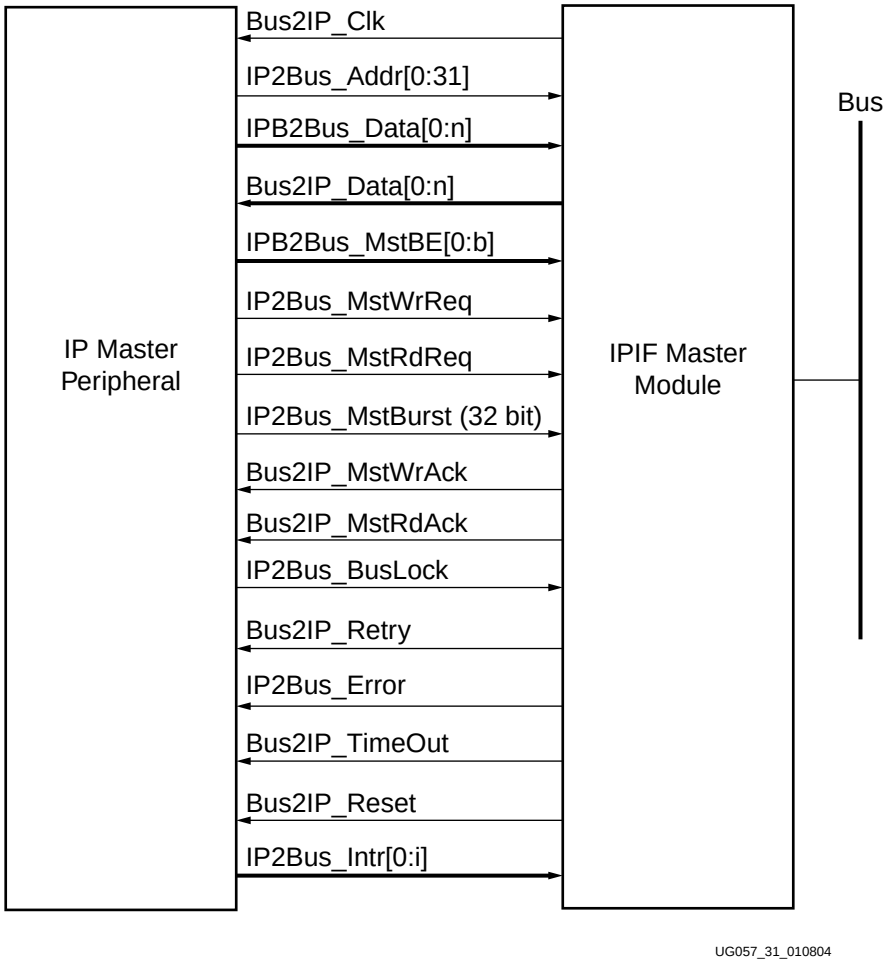


Figure 6-23: **Master IP Protocol Block Diagram**

The IPIF Master module provides a clock output, **Bus2IP_Clk**, which allows the IP to be clocked synchronously by the IPIF. This clock is provided via a BUFG in the FPGA, and thus no extra clock buffering is required in the IP.

Initiating a Cycle

In order to initiate a cycle on the bus, the IP must provide a full 32-bit address to the IPIF module by way of **IP2Bus_Addr[0:31]**. The addresses that are valid during IP to IPIF cycles are placed on the bus at the appropriate time by the IPIF. The lower two address bits, **IP2Bus_Addr[30:31]**, can be tied to logic 0 if the IP always performs 32-bit transfers across the IPIF.

Figure 6-23 shows the data buses for the IP. The **IP2Bus_Data** bus is the write data bus from the IP; that is, when the IP wishes to write data to the IPIF Master module, it asserts its write data on the **IP2Bus_Data** lines. Similarly, when the IP wishes to read data from the IPIF Master module, it looks for read data on the **Bus2IP_Data** lines.

The IPIF Master module uses byte enables to select the size of the data transferred during the initiated bus cycle. For 32-bit IP masters, the IPIF requires four byte enables, **IP2Bus_MstBE[0:3]**. Each byte enable corresponds to the byte lane in which to enable transfers. Accordingly, byte, half word, and word transfers can be accommodated by asserting the correct **IP2Bus_MstBE[0:3]**. The **IP2Bus_MstBE[0]** corresponds to the byte lane contained in **IP2Bus_Data[0:7]** or **Bus2IP_Data[0:7]** when in IBM-endianess mode.

Requesting Service

In order to initiate the request for service, the IP must assert the **IP2Bus_Addr**, the **IP2Bus_MstBE**, **IP2Bus_Data** (if required), and issue an **IP2Bus_MstWrReq** or **IP2Bus_MstRdReq**. Asserting the **IP2Bus_MstWrReq**, along with the proper qualifiers, initiates a master write transaction across the bus to the address pointed at by the **IP2Bus_Addr**. Asserting the **IP2Bus_MstRdReq**, along with the proper qualifiers, initiates a master read transaction across the bus to the address pointed at by the **IP2Bus_Addr**.

IP masters that are 32 bits can also issue an **IP2Bus_MstBurst** signal along with the **IP2Bus_MstWrReq** or **IP2Bus_MstRdReq** signals, indicating that the IP wishes to place a set of contiguous 32-bit transfers on the bus. The address of the transfer must always be word aligned (e.g. address bits 30, 31 are both low). For best bus performance across the IPIF, the master must only perform cacheline-aligned transfers using the **IP2Bus_MstBurst** signal.

When the IPIF receives an acknowledge from the bus slave it is talking to, it issues either the **Bus2IP_MstWrAck** or **Bus2IP_MstRdAck** according to the cycle type initiated by the IP. In the case of read cycles, the assertion of **Bus2IP_MstRdAck** indicates that valid data will be removed from the **Bus2IP_Data** bus.

Asserting Bus Locking

The IP may also choose to lock multiple transactions together via the **IP2Bus_MstBusLock** signal. This asserts the Bus Locking mechanism on the bus and prevents any other Masters from gaining access to the bus until the **IP2Bus_MstBusLock** signal is deasserted. A substantial loss in performance may result if the bus locking mechanism is used too frequently. In general, only use bus lock if a set of transactions must remain atomic across the bus.

Completing a Cycle

The IPIF provides error feedback to the IP master about how the IP-initiated cycle completed. If the cycle is completed normally, then none of the error feedback signals will be true. However, the IPIF provides the **Bus2IP_MstRetry** signal to tell the IPIF Master module it must back off the bus and restart the transaction. Additionally, the IPIF also provides **Bus2IP_MstError** to indicate that the bus cycle(s) ended in an error condition. Finally, the IPIF provides the **Bus2IP_MstTimeout** signal to indicate that the bus cycle timed out. Timeouts are usually caused by an invalid address.

Master Signal Protocol

Figure 6-24 and Figure 6-25 illustrate single write and read cycles of an IP bus cycle initiation. Each figure shows two possible access times in response to the IP, short and long.

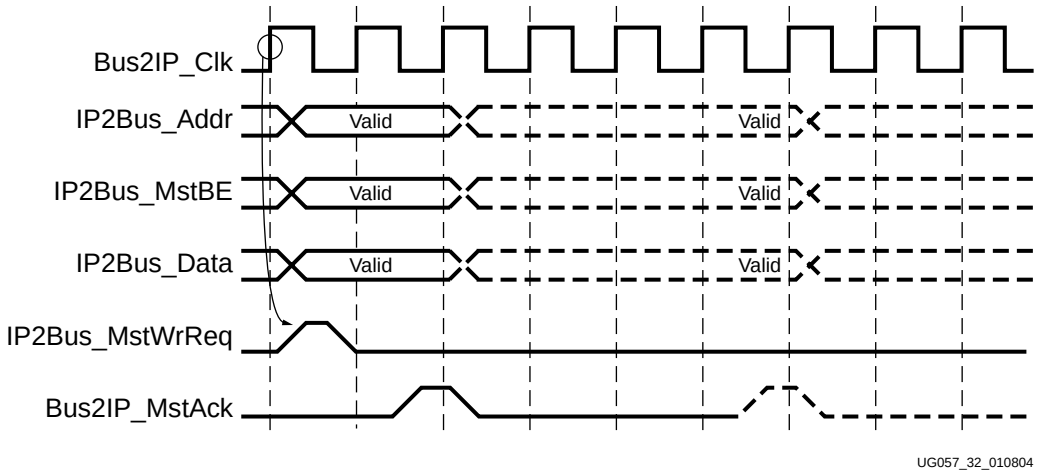


Figure 6-24: **Timing Diagram for the Master IP Module Protocol Single Write Transaction**

Write Transactions

The IP initiates a single write cycle by presenting the **IP2Bus_MstWrReq**, along with all the transaction qualifiers (including **IP2Bus_MstAddr**, **IP2Bus_MstBE**, and **IP2Bus_Data**) to the IPIF master. When the IPIF master receives a write request, it initiates a write transaction. The transaction qualifiers are also used to correctly construct the master bus cycle. In general, if the bus is not busy, then the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

Figure 6-24 illustrates both *not busy* and *busy* cases. In the first case (left of diagram), the IP has requested the bus write data to the bus slave, specified by the address provided by the IP. In this case, the bus answers quickly, allowing the IPIF to quickly acknowledge the cycle. In contrast, the right side of the diagram indicates a case where the IPIF Master module has to wait for another master to get off the bus, and then arbitrate for the bus. In this case, the bus takes much longer to acknowledge the cycle back to the IPIF, therefore delaying the IPIF's acknowledge to the IP.

Note: In this example, the bus transactions all completed normally. In the case of an error, retry, or timeout, the IPIF Master module generates the Bus2IP_Error, Bus2IP_Retry, and/or Bus2IP_Timeout during the Bus2IP_MstWrAck signal. It is the IP's responsibility to correctly address the problem.

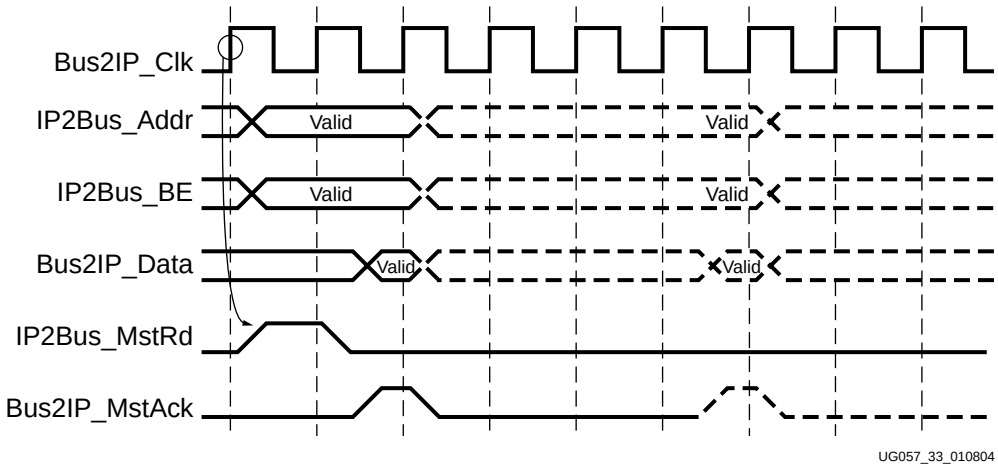


Figure 6-25: **Timing Diagram for the Master IP Module Protocol Single Read Transaction**

Read Transactions

The IP initiates a single read cycle by presenting the **IP2Bus_MstRdReq**, along with all the transaction qualifiers including **IP2Bus_MstAddr** and **IP2Bus_MstBE**. When the IPIF master receives a read request, it initiates a bus read transaction. The transaction qualifiers are also used to correctly construct the master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

Figure 6-25 illustrates both *not busy* and *busy* cases. In the first case (left of diagram), the IP has requested the bus read data to the bus slave specified by the address provided by the IP. In this case, the bus answers quickly, allowing the IPIF to quickly acknowledge the cycle. In contrast, the right side of the diagram indicates a case where the IPIF Master module has to wait for another master to get off the bus, and then arbitrate for the bus. In this case, the bus takes much longer to acknowledge the cycle back to the IPIF, therefore delaying the IPIF's acknowledge to the IP.

Note: In this example, the bus transactions all completed normally. In the case of an error, retry, or timeout, the IPIF Master module generates the Bus2IP_Error, Bus2IP_Retry, and/or Bus2IP_Timeout during the Bus2IP_MstWrAck signal. It is the IP's responsibility to correctly address the problem.

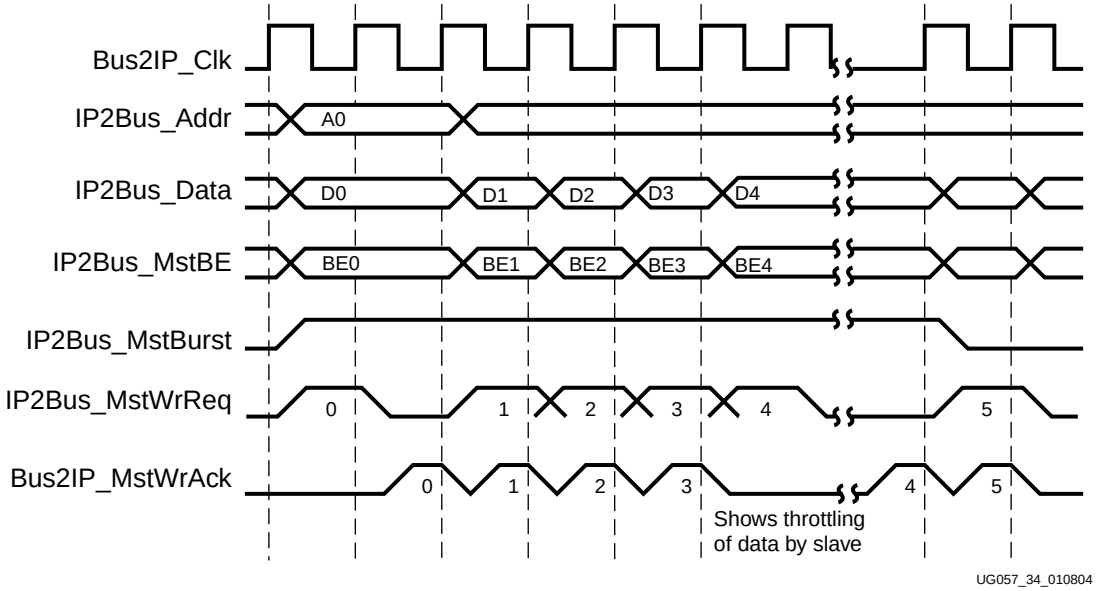


Figure 6-26: **Timing Diagram for the Master IP Module Protocol Burst Write Transaction**

Burst Write Transactions

Figure 6-26 illustrates the burst write cycle case of an IP bus cycle initiation. This diagram indicates a burst write transfer. There is no real limit on the number of datums that can be transferred, other than the attendant needs of other masters on the bus or the requirements of the memory subsystem.

The IP initiates a a burst write cycle by presenting the **IP2Bus_MstWrReq**, along with all the transaction qualifiers including **IP2Bus_MstAddr**, **IP2Bus_MstBE**, **IP2Bus_MstBurst**, and **IP2Bus_Data**. When the IPIF master receives a burst write request, it initiates a bus burst write transaction. The transaction qualifiers are also used to correctly construct the Master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

Note:

The first write datum is held until acknowledged by **Bus2IP_MstWrAck**. Datums are then sent per every active **Bus2IP_MstWrAck**. Since the bus slave on the other end of the transaction can throttle the data rate, it is possible that gaps may exist such as that shown between cycle 3 and cycle K-1.

There is no limit to the number of burst cycles that can be accomplished other than the practical utilization of the bus. It is recommended that only cache-aligned bursts be done in order to maximize system performance.

The IP deasserts the **IP2Bus_MstBurst** signal when the second to last datum is transferred. This is required to inform the IPIF logic that the next datum is the last datum of the transfer, and to get off the bus efficiently. Figure 6-26 indicates no gap in the **Bus2IP_MstWrAcks** for the last two cycles. However, in some cases the datums may be separated by several clocks. In this case, the **IP2Bus_MstBurst** signal must be terminated in the same cycle that a the second to last datum is acknowledged.

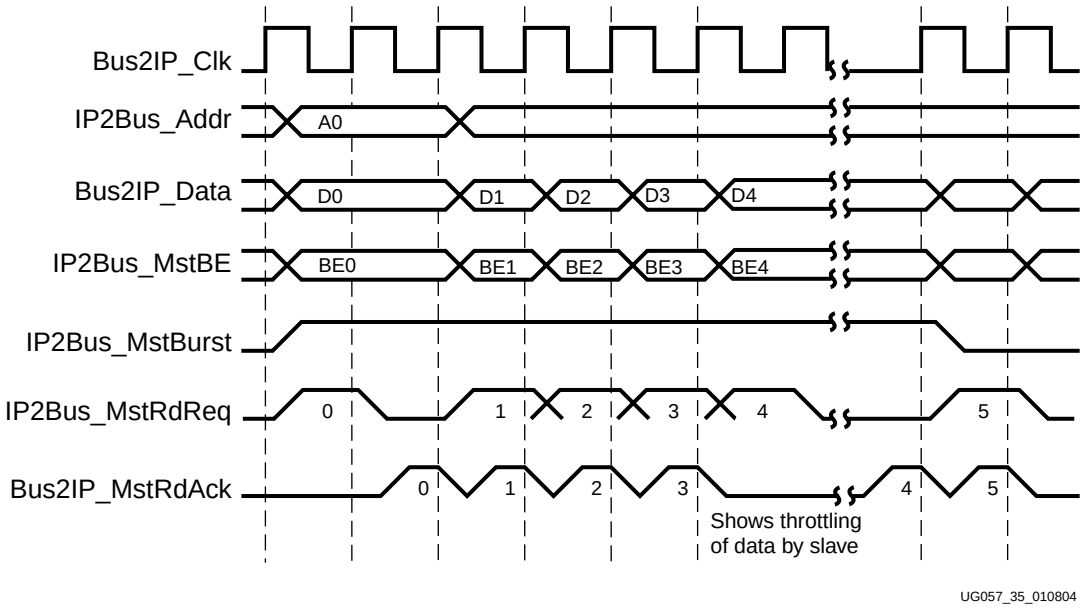


Figure 6-27: **Timing Diagram for the Master IP Module Protocol Burst Read Transaction**

Burst Read Transactions

Figure 6-27 illustrates the burst read cycle case of an IP bus cycle initiation. This diagram indicates a burst read transfer. There is no real limit on the number of datums that can be transferred, other than the attendant needs of other masters on the bus or the requirements of the memory subsystem.

The IP initiates a burst read cycle by presenting the **IP2Bus_MstRdReq**, along with all the transaction qualifiers including **IP2Bus_MstAddr**, **IP2Bus_MstBE**, and **IP2Bus_MstBurst**. When the IPIF master receives a burst read request, it initiates a bus burst read transaction. The transaction qualifiers are also used to correctly construct the bus Master bus cycle. In general, if the bus is not busy, the IPIF will quickly acknowledge the IP's initiated bus cycle. If the bus is busy, and the IPIF Master module has to wait for another master before it gets the bus, then the IPIF may take longer to acknowledge the cycle.

Note:

The first read datum is not available until acknowledged by **Bus2IP_MstRdAck**. Datums are then sent per every active **Bus2IP_MstRdAck**. Since the bus slave on the other end of the transaction can throttle the data rate, it is possible that gaps may exist such as that shown between cycle 3 and cycle K-1.

There is no limit to the number of burst cycles that can be accomplished other than the practical utilization of the bus. It is recommended that only cache-aligned bursts be done in order to maximize system performance.

The IP deasserts the **IP2Bus_MstBurst** signal when the second to last datum is transferred. This is required to inform the IPIF logic that the next datum is the last datum of the transfer, and to get off the bus efficiently. Figure 6-27 indicates no gap in the **Bus2IP_MstWrAcks** for the last two cycles. However, in some cases the datums may be separated by several clocks. In this case, the **IP2Bus_MstBurst** signal must be terminated in the same cycle that the second to last datum is acknowledged.

General Burst Cycle Issues

When issuing burst cycle, the IPIF Master module must be capable of providing or receiving a datum per **Bus2IP_Clk**. The bus permits slaves to respond each cycle of a burst operation, so there is no means to tell the slave to hold off. Only slaves can throttle transactions.

Master Signal List

Table 6-9 shows the names, direction, and a brief description of the signals that connect to the IP from the IPIF Master module.

Table 6-9: **Signals for the IPIF Master Module**

Name	Direction	Description
Bus2IP_Clk	From IPIF	Clock source (from global buffer)
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF
IP2Bus_Addr[0:31]	To IPIF	32-bit address of location Master wants to read or write
IP2Bus_MstBE[0:b]	To IPIF	Master byte enable, 1 = byte late valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)
IP2Bus_MstWrReq	To IPIF	Master requests write access to the bus, single **Bus2IP_Clk** high
IP2Bus_MstRdReq	To IPIF	Master requests read access to the bus, single **Bus2IP_Clk** high
IP2Bus_MstBurst	To IPIF	Master requests burst access to bus, goes low at data acknowledge for second to last datum
IP2Bus_MstBusLock	To IPIF	Master requires bus to lock, not allowing release of this master until transfer is complete
Bus2IP_MstWrAck	From IPIF	Acknowledge from bus that Master write datum has been accepted, single **Bus2IP_Clk** high
Bus2IP_MstRdAck	From IPIF	Acknowledge from bus that Master read datum has been presented, single **Bus2IP_Clk** high
Bus2IP_MstError	From IPIF	Bus tells IP that last transfer was in error, valid during **MstRdAck** or **MstWrAck** only
Bus2IP_MstTimeout	From IPIF	Bus tells IP that transfer timed out at bus slave. (Data may be unknown.) Valid in place of **MstRdAck** or **MstWrAck**. If valid at same time as ack, then data transferred OK.
Bus2IP_MstRetry	From IPIF	Bus tells IP to get off bus now. Valid in place of **MstRdAck** or **MstWrAck** **NOTE:** Acks are OK, but data must be assumed not have been transferred in any case.

Master Parameters

Table 6-10 shows the parameters that are associated with the IPIF Master module..

Table 6-10: **Parameters for the IPIF Master Module**

Affects	Parameter	Value	Type
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean
General IPIF Master Module	TBD		

IPIF Parameterization

Table 6-11 includes parameters and their accompanying descriptions for all modules.

Table 6-11: **Complete Parameters for All IPIF Modules**

Affects	Parameter	Value	Type	Module
General IPIF	IPIF_DATA_BUS_WIDTH Sets the size of the data bus for IPIF (where "n" in **Bus2IP_Data[0:n]** or **IP2Bus_Data[0:n]** is equal to IPIF_DATA_BUS_WIDTH)	8, 16, or 32	number	All
	IPIF_NUMBER_OF_BYTE_ENABLES Sets the number of byte enables	1, 2, or 4	number	
	IPIF_NUMBER_OF_INTR Sets the number of interrupts the IP provides to the IPIF (where "i" in **IP2Bus_Intr[0:i]** is equal to IPIF_NUMBER_OF_INTR)	0 to 8	number	
	IPIF_INTR_ID Sets the unique interrupt ID for this IPIF	16 bits	number	
	IPIF_HAS_INTC Sets whether the IPIF has a built-in interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean	
	IP_HAS_OWN_INTC Sets whether the IP has its own interrupt controller (See Figure 6-5 for more information)	0 or 1, 1 = true	boolean	
General IPIF Slave DMA Handshake Module	DMA_HNDSHK_RESPONSE_TIME_MIN Sets the minimum number of **Bus2IP_Clk** cycles in which the IPIF will respond with a DMA_ACK	0 to 255	number	Slave DMA Handshake
	DMA_HNDSHK_DATA_VALID_WIDTH Sets the number of **Bus2IP_Clk** cycles that the data will remain valid during DMA handshaking	0 to 255	number	

Table 6-11: **Complete Parameters for All IPIF Modules *(Continued)***

Affects	Parameter	Value	Type	Module
General IPIF Slave Control Register Module	IPIF_NUMBER_OF_REGS Sets the total number of registers at the bus width set by IPIF_DATA_BUS_WIDTH	1 to 255	number	Slave Control Register
	IPIF_REG_BASE_ADDR Sets the base address where IPIF registers will start in memory	32-bit decode	number	
	IPIF_REG_BASE_ADDR_BIT_ENBL[0:31] Allows specification of which address bits to decode in IPIF_REG_BASE_ADDR	1 = decode respective address bit	mask	
IPIF Slave Control Register (Arrays of Parameters)	IPIF_REGx_NUMBER_OF_BITS Defines the number of bits for each register (where x = 0 to IP_NUMBER_OF_REGS –1)	ordinal 0 to 32	number	
	IPIF_REGx_DATA_BIT_VALID_MASK[0:n] Defines which bits in each register are physically present (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH -1)	1 = bit position is used in this register	mask	
	IPIF_REGx_READABLE_BITS[0:n] Defines which bits in each register are readable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH -1)	1 = bit position will be readable by IPIF	mask	
	IPIF_REGx_WRITEABLE_BITS[0:n] Defines which bits in each register are writable by the IPIF (where x = 0 to IP_NUMBER_OF_REGS –1 and n = IPIF_DATA_BUS_WIDTH –1)	1 = bit position will be writable by IPIF	mask	
General IPIF Slave SRAM Module	IPIF_NUMBER_OF_SRAM_DECODER_REGIONS Defines the number of decoded regions of memory address space for the IPIF Slave SRAM module	1 to 4 decoded regions allowed	number	Slave SRAM
	IPIF_SLV_SRAM_ADDR_BUS_LSB Sets the LSB index number for the address provided by the IPIF to the IP	ordinal 0 to 31	number	
	IPIF_SLV_SRAM_ADDR_BUS_MSB Sets the MSB index number for the address provided by the IPIF to the IP	ordinal 0 to 31	number	
IPIF Slave SRAM Module (Arrays of Parameters)	IPIF_SRAMd_BASE_ADDR Sets the base address of the “d” region for the Slave SRAM module (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1)	32-bit decode of one region	number	
	IPIF_SRAMd_BASE_ADDR_BIT_ENBL[0:31] Allows specification of which bits address bits to decode in the IPIF_SRAMd_BASE_ADDR (where d = 0 to IPIF_NUMBER_SRAM_DECODER_REGIONS –1)	1 = decode respective address bit	mask	

Table 6-11: **Complete Parameters for All IPIF Modules *(Continued)***

Affects	Parameter	Value	Type	Module
General IPIF Slave FIFO Module	IPIF_RD_FIFO_DEPTH Sets the depth of the Read FIFO in IPIF_RD_FIFO_DATA_WIDTH units	1 to 2048, in units of data width	number	Slave FIFO
	IPIF_RD_FIFO_DATA_WIDTH Sets the width of the Read FIFO data bus	1 to 32 bits	number	
	IPIF_RD_FIFO_TRIG_THRESHOLD Sets the trigger point of the Read FIFO if packet processing	1 to 2048	number	
	IPIF_WR_FIFO_DEPTH Sets the depth of the Write FIFO in IPIF_WR_FIFO_DATA_WIDTH units	1 to 2048, in units of data width	number	
	IPIF_WR_FIFO_DATA_WIDTH Sets the width of the Write FIFO data bus	1 to 32 bits	number	
	IPIF_WR_FIFO_TRIG_THRESHOLD Sets the trigger point of the Write FIFO if packet processing	1 to 2048	number	
	IPIF_PACKET_FIFO_MODE_ENBL Enables the IPIF Slave FIFO module with packet FIFO functionality	0 or 1, 1 = true	boolean	
	IPIF_WR_FIFO_NUMBER_OF_OCC_BITS Sets the total number of bits in the **FIFO2IP_WrFIFO_Occupancy[0:fo]** signals	0 TO 12	number	
	IPIF_WR_FIFO_NUMBER_OF_VAC_BITS Sets the total number of bits in the **FIFO2IP_WrFIFO_Vacancy[0:fv]** signals	0 TO 12	number	
IPIF Master Module	TBD			Master

IPIF Signals

Table 6-12 includes signals and their accompanying descriptions for all modules.

Table 6-12: **IPIF Signals Connecting to IP for All Modules**

Name	Dir	Description	Slave DMA Hand-shake	Slave Ctrl Reg	Slave SRAM	Slave FIFO	Mstr
Bus2IP_Clk	From IPIF	Clock source (from global buffer)	x	x	x	x	x
Bus2IP_Reset	From IPIF	Active-high synchronous reset source (from global buffer)	x	x	x	x	x
IP2Bus_Intr[0:i]	To IPIF	Interrupt input from IP to IPIF	x	x	x	x	x
IP2Bus_Error	To IPIF	Error signal from IP to IPIF (Valid only during a data acknowledge cycle)	x	x	x	x	
IP2Bus_Retry	To IPIF	Indicates IP wants master to retry the cycle	x	x	x	x	

Table 6-12: **IPIF Signals Connecting to IP for All Modules *(Continued)***

Name	Dir	Description	Slave DMA Hand-shake	Slave Ctrl Reg	Slave SRAM	Slave FIFO	Mstr
IP2Bus_ToutSup	To IPIF	Forces the suppression of watch dog timeout on the bus	x	x	x	x	
Bus2IP_Data[0:n]	From IPIF	IPIF Write data (where n = IPIF_DATA_BUS_WIDTH –1)	x	x	x		
IP2Bus_Data[0:n]	To IPIF	IPIF Read data (where n = IPIF_DATA_BUS_WIDTH –1)	x	x	x		
Bus2IP_BE[0:b]	From IPIF	Byte enable, 1 = byte lane valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)	x	x	x		
Bus2IP_WrReq	From IPIF	Write request from IPIF to IP, single clock high		x	x		
IP2Bus_WrAck	To IPIF	Acknowledge that write data has been taken from **Bus2IP_Data[0:n]**, single **Bus2IP_Clk** high		x	x		
Bus2IP_RdReq	From IPIF	Read request from IPIF to IP, single clock high		x	x		
IP2Bus_RdAck	To IPIF	Acknowledge that read data has been placed on **IP2Bus_Data[0:n]**, single **Bus2IP_Clk** high		x	x		
IP2Bus_DMA_Req	To IPIF	DMA handshake transfer request from IP	x				
Bus2IP_DMA_Ack	From IPIF	DMA handshake transfer acknowledge from IPIF	x				
Bus2IP_RegCE(r)	From IPIF	Clock enable of decoded "r" register (where r = 0 to IPIF_NUMBER_OF_REGS –1)		x			
Bus2IP_Addr[al:ah]	From IPIF	IPIF Slave SRAM address, where: al = IPIF_SLV_SRAM_ADDR_BUS_MSB ah = IPIF_SLV_SRAM_ADDR_BUS_LSB (*al* is a lower number than *ah* due to big-endian numbering)			x		
Bus2IP_SRAM_CE	From IPIF	Clock enable of decoded SRAM address space, high for entire bus cycle			x		
IP2Bus_Clk (optional)	To IPIF	Optional clock source to control IPIF Slave FIFO module signals (not yet available				x	
Bus2IP_FIFO_CE	From IPIF	Clock enable of decoded IPIF Slave FIFO address space, high for entire bus cycle				x	
FIFO2IP_WrFIFO_Data[0:n]	From IPIF	IPIF write FIFO data (where n = IPIF_WR_FIFO_DATA_WIDTH –1)				x	
FIFO2IP_WrFIFO_Empty	From IPIF	IPIF write FIFO is empty when high				x	
FIFO2IP_WrFIFO_WrReq	From IPIF	IPIF write FIFO request, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum				x	
IP2FIFO_WrFIFO_WrAck	To IPIF	IPIF write FIFO acknowledge, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum				x	
IP2FIFO_RdFIFO_Data[0:n]	To IPIF	IPIF read FIFO data (where n = IPIF_RD_FIFO_DATA_WIDTH –1)				x	
FIFO2IP_RdFIFO_Full	From IPIF	IPIF write FIFO is empty when this signal is high				x	
FIFO2IP_RdFIFO_WrReq	To IPIF	IPIF read FIFO request, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum				x	
IP2FIFO_RdFIFO_WrAck	From IPIF	IPIF read FIFO acknowledge, single **Bus2IP_Clk** (or **IP2Bus_Clk**) high per datum				x	

Table 6-12: **IPIF Signals Connecting to IP for All Modules *(Continued)***

Name	Dir	Description	Slave DMA Hand-shake	Slave Ctrl Reg	Slave SRAM	Slave FIFO	Mstr
IP2FIFO_WrFIFO_Mark	To IPIF	Marks datum as first datum of packet, high during **IP2FIFO_WrFIFO_WrAck** only				x	
IP2FIFO_WrFIFO_Restore	To IPIF	Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an **IP2FIFO_WrFIFO_WrAck**)				x	
IP2FIFO_WrFIFO_Release	To IPIF	Releases the marked position, write FIFO can now accumulate data from mark forward				x	
FIFO2IP_WrFIFO_Occupancy[0:fo]	From IPIF	Indicates how much data is in the write FIFO (where *fo* = IPIF_WR_FIFO_NUMBER_OF_OCC_BITS –1)				x	
IP2FIFO_RdFIFO_Mark	To IPIF	Marks datum as first datum of packet, high during **IP2FIFO_RdFIFO_WrAck** only				x	
IP2FIFO_RdFIFO_Restore	To IPIF	Restores write FIFO to marked location, next request will be at marked data (this signal is valid high only during an **IP2FIFO_RdFIFO_WrAck**)				x	
IP2FIFO_RdFIFO_Release	To IPIF	Releases the marked position, read FIFO can now accumulate data from mark forward				x	
FIFO2IP_WrFIFO_Vacancy[0:fv]	From IPIF	Indicates how much data is in the read FIFO (where *fv* = IPIF_WR_FIFO_NUMBER_OF_VAC_BITS –1)				x	
IP2Bus_Addr[0:31]	To IPIF	32-bit address of location Master wants to read or write					x
IP2Bus_MstBE[0:b]	To IPIF	Master byte enable, 1 = byte late valid (where b = IPIF_NUMBER_OF_BYTE_ENABLES –1)					x
IP2Bus_MstWrReq	To IPIF	Master requests write access to the bus, single **Bus2IP_Clk** high					x
IP2Bus_MstRdReq	To IPIF	Master requests read access to the bus, single **Bus2IP_Clk** high					x
IP2Bus_MstBurst	To IPIF	Master requests burst access to bus, goes low at data acknowledge for second to last datum					x
IP2Bus_MstBusLock	To IPIF	Master requires bus to lock, not allowing release of this master until transfer is complete					x
Bus2IP_MstWrAck	From IPIF	Acknowledge from bus that Master write datum has been accepted, single **Bus2IP_Clk** high					x
Bus2IP_MstRdAck	From IPIF	Acknowledge from bus that Master read datum has been presented, single **Bus2IP_Clk** high					x
Bus2IP_MstError	From IPIF	Bus tells IP that last transfer was in error, valid during **MstRdAck** or **MstWrAck** only					x
Bus2IP_MstTimeout	From IPIF	Bus tells IP that transfer timed out at bus slave. (Data may be unknown.) Valid in place of **MstRdAck** or **MstWrAck**. If valid at same time as ack, then data transferred OK.					x
Bus2IP_MstRetry	From IPIF	Bus tells IP to get off bus now. Valid in place of **MstRdAck** or **MstWrAck** **NOTE:** B Acks are OK, but data must be assumed not have been transferred in any case.					x

Chapter 7

OPB to PCI Bridge Lite

Overview

The OPB to PCI Bridge translates transactions between OPB and the PCI Bus. It has a master/slave OPB interface and can be an initiator or target on PCI. Its design utilizes an Intellectual Property InterFace (IPIF) module to abstract OPB transactions into a simple protocol that is easier to design with.

This document describes the "Lite" or simplified version of the OPB PCI Bridge. It is fully capable of translating data transfers between OPB to PCI, but does not have large data buffering capabilities. This makes the overall design much smaller in terms of FPGA resource utilization, but also reduces the potential bandwidth for data transfers.

The PCI interface logic utilizes the **Xilinx LogiCORE PCI 32/33** product which helps designers to develop high performance, fully compliant PCI devices. The OPB to PCI Bridge uses the PCI Core to implement a simple 32-bit, 33 MHz PCI initiator or target. Configuration, I/O, and Memory transfer types are supported over PCI to provide compatibility with a large number of common PCI devices.

Related Documents

- IPIF Specification
- IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architecture Specifications
- Virtex-II Pro Platform FPGAs (Advance Product Specification)

Features

- 32-bit OPB Master/Slave interface with IPIF-based design
- 32-bit/33 MHz PCI interface that is V2.2 compliant to the PCI specification
- Capable of generating Configuration, I/O, and Memory transactions
- PCI clock can be divided down from the OPB clock by any integer divisor. PCI to OPB clock ratios of 1:1, 1:2, 1:3, etc. are possible

Module Port Interface

Table 7-1: **Global Signals**

Name	Direction	Description
OPB_Clk	Input	OPB system clock
OPB_Rst	Input	OPB system reset

Table 7-2: **OPB Slave Signals**

Name	Direction	Description
OPB_ABus[0:31]	Input	OPB address bus
OPB_BE[0:3]	Input	OPB byte enables
OPB_DBus[0:31]	Input	OPB data bus
OPB_RNW	Input	OPB read not write
OPB_select	Input	OPB select
OPB_seqAddr	Input	OPB sequential address
S1_DBus[0:31]	Output	Slave data bus
S1_errAck	Output	Slave error acknowledge
S1_retry	Output	Slave bus cycle retry
S1_toutSup	Output	Slave timeout suppress
S1_xferAck	Output	Slave transfer acknowledge

Table 7-3: **PCI Master Signals**

Signal	Direction	Description
GNT_N	Input	PCI Grant
PCLK_IN	Input	PCI Reference Clock In
IDSEL	Output	PCI Identification Select
REQ_N	Output	PCI Request Bus Access
RST_N	Output	PCI Reset External PCI Devices
ACK64_N	Input-Output	PCI 64-Bit Transfer Acknowledge (Should be connected to external pull-up for a 32-bit PCI bus)
AD[31:0]	Input-Output	PCI Address/Data Bus
CBE[3:0]	Input-Output	PCI Command/Byte Enables
DEVSEL_N	Input-Output	PCI Device Select

Table 7-3: **PCI Master Signals *(Continued)***

Signal	Direction	Description
FRAME_N	Input-Output	PCI Framing Signal
IRDY_N	Input-Output	PCI Initiator Ready
PAR	Input-Output	PCI Parity
PERR_N	Input-Output	PCI Parity Error
REQ64_N	Input-Output	PCI 64-Bit Transfer Request (Should be connected to external pull-up for a 32-bit PCI bus)
SERR_N	Input-Output	PCI Error
STOP_N	Input-Output	PCI Stop
TRDY_N	Input-Output	PCI Target Ready

Table 7-4: **OPB Master Signals**

Signal	Direction	Description
M_Abus[0:31]	Output	Master address bus
M_BE[0:3]	Output	Master Byte Enables
M_busLock	Output	Master bus arbitration lock
M_request	Output	Master bus request
M_RNW	Output	Master read not write
M_select	Output	Master select
M_seqAddr	Output	Master sequential address
OPB_errAck	Input	OPB error acknowledge
OPB_MnGrant	Input	OPB master bus grant
OPB_retry	Input	OPB bus cycle retry
OPB_timeout	Input	OPB timeout error
OPB_xferAck	Input	OPB transfer acknowledge

Table 7-5: **Misc. Signals**

Name	Direction	Description
FRAMEQ_N	Output	FRAME_N signal delayed by one PCI clock (for PCI Arbiter)
IRDYQ_N	Output	IRDY_N signal delayed by one PCI clock (for PCI Arbiter)
INTA	Input	PCI Interrupt A

Table 7-5: **Misc. Signals *(Continued)***

Name	Direction	Description
INTB	Input	PCI Interrupt B
INTC	Input	PCI Interrupt C
INTD	Input	PCI Interrupt D
INTR_OUT	Output	Logical OR of all PCI interrupts (active high)
GRST_N	Output	Secondary PCI global reset

Table 7-6: **Parameters**

Name	Description
C_BASEADDR	32-bit base address of OPB to PCI Bridge (must be aligned to a 512 MB boundary)
C_HIGHADDR	Must be set to C_BASEADDR + 0x1FFFFFFF

Implementation

Figure 7-1 shows a conceptual high-level view of the design OPB PCI Lite Bridge. The Bridge uses master and slave IPIF modules to help abstract the OPB interface into a simpler protocol. It also includes a Xilinx PCI Core to simplify the task of building a fully compliant PCI interface. The interface logic between the IPIFs and PCI Core is roughly based on the sample target and initiator designs described in detail by the *LogiCORE PCI Design Guide*. The IPIF signals are decoded into a simple set of signals that control the initiator and target state machine logic described by the above document.

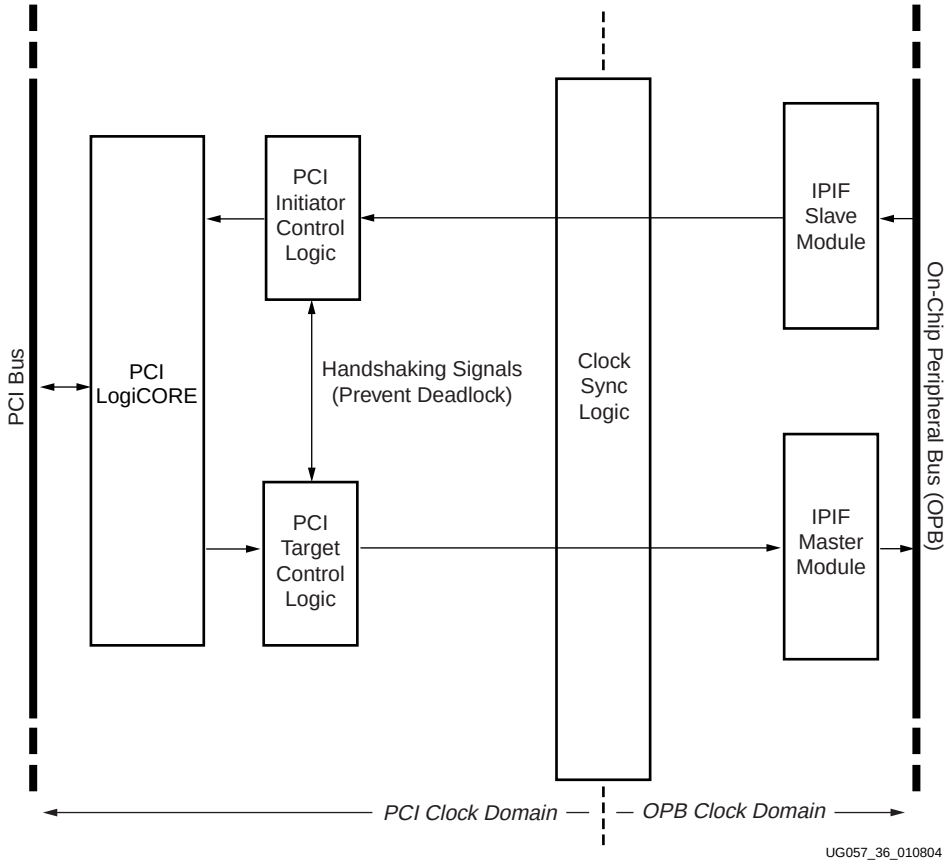


Figure 7-1: **OPB to PCI Bridge**

The IPIF signals are passed through synchronization logic to allow the OPB and PCI clock domains to be different. The synchronization logic requires that the OPB clock be an integer multiple of the PCI clock and that the rising edges of the two clocks be phase-aligned to each other. In typical systems the PCI clock will operate at 25-33 MHz while the OPB clock would be 50-100 MHz. Supporting different clock frequencies between OPB and PCI prevents the PCI clock from limiting the OPB clock.

The bridge is designed for 33 MHz PCI clock frequencies. The designer is responsible for ensuring that the external PCI clock is phase aligned with respect to the internal PCI clock. This can be accomplished by utilizing the Digital Clock Managers available in the Virtex II-Pro FPGA to deskew the external PCI clock from the internal reference clock.

OPB uses big-endian byte addressing, while PCI uses little-endian byte addressing. To translate data between the two busses and preserve byte addressing, the data bytes and byte enable bits for each 32-bit OPB word are swapped when going to or from PCI. Accesses to the Configuration Address/Data Registers (described below) are also affected by the endian swapping logic.

OPB Slave to PCI Initiator Transactions

The OPB PCI Bridge takes read/write requests at its OPB interface and completes the transaction on PCI. Since this bridge is “Lite” it does not have large data buffers. Therefore each single 32-bit OPB data transfer must be completed on PCI before the next data

transfer can begin. This is accomplished by holding back Sl_xferAck in between each word of OPB data until the corresponding PCI data transfer has completed. This allows the design to be very small but also reduces performance.

PCI errors due to target abort conditions are passed back to the OPB as an errAck response. To allow for PCI device discovery without causing CPU machine-check exceptions, errors that occur during configuration cycles are not reported back to OPB. Reading from a non-existent PCI device during a configuration cycle will return data of 0xFFFFFFFF. PCI retry is fed back to cause an OPB retry.

PCI Target to OPB Master Transactions

The OPB PCI Bridge takes read/write requests at its PCI target interface and completes the transaction on OPB. Since this bridge is "Lite" it does not have large data buffers. Therefore each single 32-bit PCI data transfer must be completed on OPB before the next data transfer can begin. This is accomplished by performing a target disconnect with data. The interface disconnects the PCI bus after the OPB data transfer of the first word has completed.

OPB retries are reflected back to PCI as a target disconnect without data. OPB timeouts or OPB transactions terminated with OPB_errAck result in a target abort on the PCI interface.

Arbiter

A simple six-master companion PCI arbiter is provided with the OPB PCI Bridge. The arbiter implements a simple round robin arbitration scheme.

Memory Map

The Bridge maps a 512 MB OPB address space into four PCI memory regions. Each memory region corresponds to a different transaction type as shown in Table 7-7. The base address is user-specified as a parameter during module instantiation. Note that the mapping from OPB to PCI address space is transparent (1:1) for the Memory transaction type. For transactions going from PCI to OPB, the bridge responds to PCI addresses of 0x00000000 - 0x0FFFFFFF (PCI BAR 0) and generates an OPB transaction to the same address. Edit the **pci_cfg.v** to change this memory window.

Table 7-7: **OPB to PCI Bridge Memory Map**

PCI Transfer Type / Register Name	OPB Address Boundaries (Hex)		PCI Address Boundaries (Hex)	
	Lower	Upper	Lower	Upper
Memory	Base Addr + 00000000	Base Addr + 17FFFFFF	Base Addr + 00000000	Base Addr + 17FFFFF
I/O	Base Addr + 18000000	Base Addr + 1BFFFFFF	Base Addr + 00000000	Base Addr + 03FFFFF
Configuration Address	Base Addr + 1C000000	Base Addr + 1C000003	N/A	N/A
Configuration Data	Base Addr + 1C000004	Base Addr + 1C000007	N/A	N/A
Configuration (External Device, Memory Mapped, Type 0 only)	Base Addr + 1C000008	Base Addr + 1DFFFFFF	00000000	01FFFFF

Table 7-7: **OPB to PCI Bridge Memory Map *(Continued)***

PCI Transfer Type / Register Name	OPB Address Boundaries (Hex)		PCI Address Boundaries (Hex)	
	Lower	Upper	Lower	Upper
Configuration (Self)(1)	Base Addr + 1E000000	Base Addr + 1FFFFFFF	00000000	01FFFFFF

Note: Software must first use self-configuration accesses to enable the master interface on the bridge before it can perform any Memory or I/O transactions.

Configuration

Before the PCI Bridge can generate any PCI transactions, its master interface must first be enabled. This can be performed by another external PCI bus master if one is available. Alternatively, the PCI Bridge supports a self configuration process whereby its master interface can be enabled via OPB commands. Reading/writing to the self configuration space will access the PCI configuration registers in the PCI Core (via a loopback over the PCI Bus). Therefore it is recommended you write the value 0xFFFF0147 to address (Base Addr + 1E000004) in order to enable the initiator functions of the PCI Core).

For configuration of external PCI devices, the PCI Bridge also supports the configuration address and data registers that are commonly used on PCs and other embedded PCI controllers. This configuration mechanism is described in detail in the PCI Specification. In summary, the user first writes to the Configuration Address Register to specify the bus number, device number, function number, register number, and so on, of the PCI device they wish to access. Subsequently a read from Configuration Data Register will return the contents of the specified register. Similarly, a write to the Configuration Data Register will write data to the specified register. Note that the enable bit (MSB of the Configuration Address Register) must be set before configuration reads or writes can be performed over the PCI bus. A configuration read that results in a PCI transaction abort will return 0xFFFFFFFF. See Table 7-8 for a summary of the Configuration Address/Data Registers. Note that the bit definitions for the fields of the configuration registers reflect the register value in the PCI domain. Therefore, data being read from or written to these registers will pass through the byte swapping logic.

Table 7-8: **Configuration Address Register (CAR) and Configuration Data Register (CDR)**

Register Name	Address	Bits	Description
CAR_EN	Base Addr + 1C000000	[31]	Enable Configuration Data Register Transaction. 1 = enable 0 = disable
	Base Addr + 1C000000	[30:24]	Unused
CAR_BN	Base Addr + 1C000000	[23:16]	Bus Number **Note:** For configuration access to PCI bus number 0, a Type 0 PCI configuration access is generated. For all other bus numbers, a Type 1 PCI configuration access is generated.

Table 7-8: **Configuration Address Register (CAR) and Configuration Data Register (CDR) *(Continued)***

Register Name	Address	Bits	Description
CAR_DN	Base Addr + 1C000000	[15:11]	Device Number
CAR_FN	Base Addr + 1C000000	[10:8]	Function Number
CAR_RN	Base Addr + 1C000000	[7:2]	Register Number
	Base Addr + 1C000000	[1:0]	Unused
CDR	Base Addr + 1C000004	[31:0]	Read/write to the register specified by the CAR. The CAR_EN bit must be high to allow a PCI configuration transaction to be generated.

Xilinx LogiCORE PCI

The OPB PCI Bridge includes the V3 Xilinx PCI LogicCore. The PCI Core helps to abstract the PCI bus into a simpler interface that makes it easier to implement full compliant PCI devices. An evaluation version of the PCI Core is provided that will stop responding after some hours of use. Contact your local sales office or go to http://www.xilinx.com/pci for more information about purchasing the full (non-evaluation) version of the PCI core or for other PCI offerings from Xilinx.

Chapter 8

OPB to PCI Bridge Lite

Overview

The OPB to PCI Bridge translates transactions between OPB and the PCI Bus. It has a master/slave OPB interface and can be an initiator or target on PCI. Its design utilizes an Intellectual Property InterFace (IPIF) module to abstract OPB transactions into a simple protocol that is easier to design with.

This document describes the "Lite" or simplified version of the OPB PCI Bridge. It is fully capable of translating data transfers between OPB to PCI, but does not have large data buffering capabilities. This makes the overall design much smaller in terms of FPGA resource utilization, but also reduces the potential bandwidth for data transfers.

The PCI interface logic utilizes the **Xilinx LogiCORE PCI 32/33** product which helps designers to develop high performance, fully compliant PCI devices. The OPB to PCI Bridge uses the PCI Core to implement a simple 32-bit, 33 MHz PCI initiator or target. Configuration, I/O, and Memory transfer types are supported over PCI to provide compatibility with a large number of common PCI devices.

Related Documents

- IPIF Specification
- IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architecture Specifications
- Virtex-II Pro Platform FPGAs (Advance Product Specification)

Features

- 32-bit OPB Master/Slave interface with IPIF-based design
- 32-bit/33 MHz PCI interface that is V2.2 compliant to the PCI specification
- Capable of generating Configuration, I/O, and Memory transactions
- PCI clock can be divided down from the OPB clock by any integer divisor. PCI to OPB clock ratios of 1:1, 1:2, 1:3, etc. are possible

Module Port Interface

Table 8-1: **Global Signals**

Name	Direction	Description
OPB_Clk	Input	OPB system clock
OPB_Rst	Input	OPB system reset

Table 8-2: **OPB Slave Signals**

Name	Direction	Description
OPB_ABus[0:31]	Input	OPB address bus
OPB_BE[0:3]	Input	OPB byte enables
OPB_DBus[0:31]	Input	OPB data bus
OPB_RNW	Input	OPB read not write
OPB_select	Input	OPB select
OPB_seqAddr	Input	OPB sequential address
S1_DBus[0:31]	Output	Slave data bus
S1_errAck	Output	Slave error acknowledge
S1_retry	Output	Slave bus cycle retry
S1_toutSup	Output	Slave timeout suppress
S1_xferAck	Output	Slave transfer acknowledge

Table 8-3: **PCI Master Signals**

Signal	Direction	Description
GNT_N	Input	PCI Grant
PCLK_IN	Input	PCI Reference Clock In
IDSEL	Output	PCI Identification Select
REQ_N	Output	PCI Request Bus Access
RST_N	Output	PCI Reset External PCI Devices
ACK64_N	Input-Output	PCI 64-Bit Transfer Acknowledge (Should be connected to external pull-up for a 32-bit PCI bus)
AD[31:0]	Input-Output	PCI Address/Data Bus
CBE[3:0]	Input-Output	PCI Command/Byte Enables
DEVSEL_N	Input-Output	PCI Device Select

Table 8-3: **PCI Master Signals *(Continued)***

Signal	Direction	Description
FRAME_N	Input-Output	PCI Framing Signal
IRDY_N	Input-Output	PCI Initiator Ready
PAR	Input-Output	PCI Parity
PERR_N	Input-Output	PCI Parity Error
REQ64_N	Input-Output	PCI 64-Bit Transfer Request (Should be connected to external pull-up for a 32-bit PCI bus)
SERR_N	Input-Output	PCI Error
STOP_N	Input-Output	PCI Stop
TRDY_N	Input-Output	PCI Target Ready

Table 8-4: **OPB Master Signals**

Signal	Direction	Description
M_Abus[0:31]	Output	Master address bus
M_BE[0:3]	Output	Master Byte Enables
M_busLock	Output	Master bus arbitration lock
M_request	Output	Master bus request
M_RNW	Output	Master read not write
M_select	Output	Master select
M_seqAddr	Output	Master sequential address
OPB_errAck	Input	OPB error acknowledge
OPB_MnGrant	Input	OPB master bus grant
OPB_retry	Input	OPB bus cycle retry
OPB_timeout	Input	OPB timeout error
OPB_xferAck	Input	OPB transfer acknowledge

Table 8-5: **Misc. Signals**

Name	Direction	Description
FRAMEQ_N	Output	FRAME_N signal delayed by one PCI clock (for PCI Arbiter)
IRDYQ_N	Output	IRDY_N signal delayed by one PCI clock (for PCI Arbiter)
INTA	Input	PCI Interrupt A

Table 8-5: **Misc. Signals *(Continued)***

Name	Direction	Description
INTB	Input	PCI Interrupt B
INTC	Input	PCI Interrupt C
INTD	Input	PCI Interrupt D
INTR_OUT	Output	Logical OR of all PCI interrupts (active high)
GRST_N	Output	Secondary PCI global reset

Table 8-6: **Parameters**

Name	Description
C_BASEADDR	32-bit base address of OPB to PCI Bridge (must be aligned to a 512 MB boundary)
C_HIGHADDR	Must be set to C_BASEADDR + 0x1FFFFFFF

Implementation

Figure 8-1 shows a conceptual high-level view of the design OPB PCI Lite Bridge. The Bridge uses master and slave IPIF modules to help abstract the OPB interface into a simpler protocol. It also includes a Xilinx PCI Core to simplify the task of building a fully compliant PCI interface. The interface logic between the IPIFs and PCI Core is roughly based on the sample target and initiator designs described in detail by the *LogiCORE PCI Design Guide*. The IPIF signals are decoded into a simple set of signals that control the initiator and target state machine logic described by the above document.

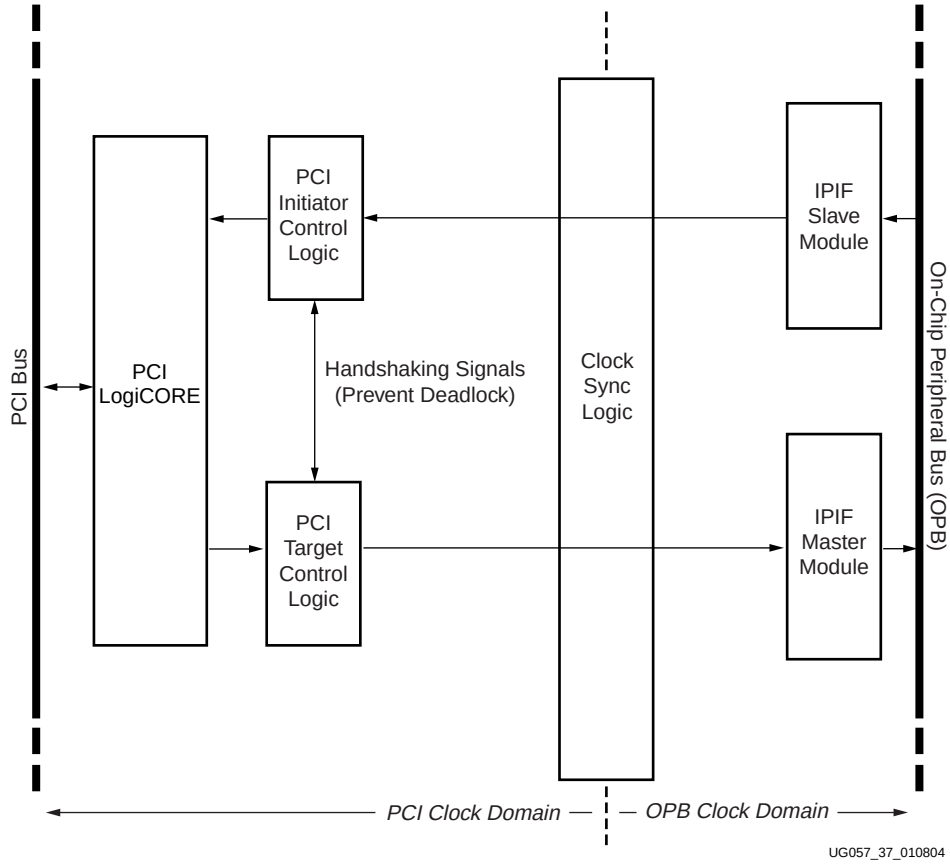


Figure 8-1: **OPB to PCI Bridge**

The IPIF signals are passed through synchronization logic to allow the OPB and PCI clock domains to be different. The synchronization logic requires that the OPB clock be an integer multiple of the PCI clock and that the rising edges of the two clocks be phase-aligned to each other. In typical systems the PCI clock will operate at 25-33 MHz while the OPB clock would be 50-100 MHz. Supporting different clock frequencies between OPB and PCI prevents the PCI clock from limiting the OPB clock.

The bridge is designed for 33 MHz PCI clock frequencies. The designer is responsible for ensuring that the external PCI clock is phase aligned with respect to the internal PCI clock. This can be accomplished by utilizing the Digital Clock Managers available in the Virtex II-Pro FPGA to deskew the external PCI clock from the internal reference clock.

OPB uses big-endian byte addressing, while PCI uses little-endian byte addressing. To translate data between the two busses and preserve byte addressing, the data bytes and byte enable bits for each 32-bit OPB word are swapped when going to or from PCI. Accesses to the Configuration Address/Data Registers (described below) are also affected by the endian swapping logic.

OPB Slave to PCI Initiator Transactions

The OPB PCI Bridge takes read/write requests at its OPB interface and completes the transaction on PCI. Since this bridge is “Lite” it does not have large data buffers. Therefore each single 32-bit OPB data transfer must be completed on PCI before the next data

transfer can begin. This is accomplished by holding back Sl_xferAck in between each word of OPB data until the corresponding PCI data transfer has completed. This allows the design to be very small but also reduces performance.

PCI errors due to target abort conditions are passed back to the OPB as an errAck response. To allow for PCI device discovery without causing CPU machine-check exceptions, errors that occur during configuration cycles are not reported back to OPB. Reading from a non-existent PCI device during a configuration cycle will return data of 0xFFFFFFFF. PCI retry is fed back to cause an OPB retry.

PCI Target to OPB Master Transactions

The OPB PCI Bridge takes read/write requests at its PCI target interface and completes the transaction on OPB. Since this bridge is "Lite" it does not have large data buffers. Therefore each single 32-bit PCI data transfer must be completed on OPB before the next data transfer can begin. This is accomplished by performing a target disconnect with data. The interface disconnects the PCI bus after the OPB data transfer of the first word has completed.

OPB retries are reflected back to PCI as a target disconnect without data. OPB timeouts or OPB transactions terminated with OPB_errAck result in a target abort on the PCI interface.

Arbiter

A simple six-master companion PCI arbiter is provided with the OPB PCI Bridge. The arbiter implements a simple round robin arbitration scheme.

Memory Map

The Bridge maps a 512 MB OPB address space into four PCI memory regions. Each memory region corresponds to a different transaction type as shown in Table 8-7. The base address is user-specified as a parameter during module instantiation. Note that the mapping from OPB to PCI address space is transparent (1:1) for the Memory transaction type. For transactions going from PCI to OPB, the bridge responds to PCI addresses of 0x00000000 - 0x0FFFFFFF (PCI BAR 0) and generates an OPB transaction to the same address. Edit the **pci_cfg.v** to change this memory window.

Table 8-7: **OPB to PCI Bridge Memory Map**

PCI Transfer Type / Register Name	OPB Address Boundaries (Hex)		PCI Address Boundaries (Hex)	
	Lower	Upper	Lower	Upper
Memory	Base Addr + 00000000	Base Addr + 17FFFFFF	Base Addr + 00000000	Base Addr + 17FFFFF
I/O	Base Addr + 18000000	Base Addr + 1BFFFFFF	Base Addr + 00000000	Base Addr + 03FFFFF
Configuration Address	Base Addr + 1C000000	Base Addr + 1C000003	N/A	N/A
Configuration Data	Base Addr + 1C000004	Base Addr + 1C000007	N/A	N/A
Configuration (External Device, Memory Mapped, Type 0 only)	Base Addr + 1C000008	Base Addr + 1DFFFFFF	00000000	01FFFFFF

Table 8-7: **OPB to PCI Bridge Memory Map *(Continued)***

PCI Transfer Type / Register Name	OPB Address Boundaries (Hex)		PCI Address Boundaries (Hex)	
	Lower	Upper	Lower	Upper
Configuration (Self)(1)	Base Addr + 1E000000	Base Addr + 1FFFFFFF	00000000	01FFFFFF

Note: Software must first use self-configuration accesses to enable the master interface on the bridge before it can perform any Memory or I/O transactions.

Configuration

Before the PCI Bridge can generate any PCI transactions, its master interface must first be enabled. This can be performed by another external PCI bus master if one is available. Alternatively, the PCI Bridge supports a self configuration process whereby its master interface can be enabled via OPB commands. Reading/writing to the self configuration space will access the PCI configuration registers in the PCI Core (via a loopback over the PCI Bus). Therefore it is recommended you write the value 0xFFFF0147 to address (Base Addr + 1E000004) in order to enable the initiator functions of the PCI Core).

For configuration of external PCI devices, the PCI Bridge also supports the configuration address and data registers that are commonly used on PCs and other embedded PCI controllers. This configuration mechanism is described in detail in the PCI Specification. In summary, the user first writes to the Configuration Address Register to specify the bus number, device number, function number, register number, and so on, of the PCI device they wish to access. Subsequently a read from Configuration Data Register will return the contents of the specified register. Similarly, a write to the Configuration Data Register will write data to the specified register. Note that the enable bit (MSB of the Configuration Address Register) must be set before configuration reads or writes can be performed over the PCI bus. A configuration read that results in a PCI transaction abort will return 0xFFFFFFFF. See Table 8-8 for a summary of the Configuration Address/Data Registers. Note that the bit definitions for the fields of the configuration registers reflect the register value in the PCI domain. Therefore, data being read from or written to these registers will pass through the byte swapping logic.

Table 8-8: **Configuration Address Register (CAR) and Configuration Data Register (CDR)**

Register Name	Address	Bits	Description
CAR_EN	Base Addr + 1C000000	[31]	Enable Configuration Data Register Transaction. 1 = enable 0 = disable
	Base Addr + 1C000000	[30:24]	Unused
CAR_BN	Base Addr + 1C000000	[23:16]	Bus Number **Note:** For configuration access to PCI bus number 0, a Type 0 PCI configuration access is generated. For all other bus numbers, a Type 1 PCI configuration access is generated.

Table 8-8: **Configuration Address Register (CAR) and Configuration Data Register (CDR) *(Continued)***

Register Name	Address	Bits	Description
CAR_DN	Base Addr + 1C000000	[15:11]	Device Number
CAR_FN	Base Addr + 1C000000	[10:8]	Function Number
CAR_RN	Base Addr + 1C000000	[7:2]	Register Number
	Base Addr + 1C000000	[1:0]	Unused
CDR	Base Addr + 1C000004	[31:0]	Read/write to the register specified by the CAR. The CAR_EN bit must be high to allow a PCI configuration transaction to be generated.

Xilinx LogiCORE PCI

The OPB PCI Bridge includes the V3 Xilinx PCI LogicCore. The PCI Core helps to abstract the PCI bus into a simpler interface that makes it easier to implement full compliant PCI devices. An evaluation version of the PCI Core is provided that will stop responding after some hours of use. Contact your local sales office or go to http://www.xilinx.com/pci for more information about purchasing the full (non-evaluation) version of the PCI core or for other PCI offerings from Xilinx.

Chapter 9

OPB Touch Screen Controller

Overview

This module is an On-Chip Peripheral Bus (OPB) slave device that is designed to control a touch screen digitizer chip. It is designed to interface to the Texas Instruments (Burr-Brown) ADS7846 touch screen controller chip present on the ML300 board but will likely work with other compatible digitizer chips. The OPB Touch Screen Controller module utilizes the Xilinx Intellectual Property InterFace (IPIF) to simplify its design. Interrupts for "pen-down" and "pen-up" events are also supported.

Related Documents

The following documents provide additional information:

- IPIF Specification
- IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1
- Virtex-II Pro™ Platform FPGAs (Data Sheets)
- Texas Instruments (Burr-Brown) ADS7846 Touch Screen Controller Data Sheet (http://www-s.ti.com/sc/ds/ads7846.pdf)

Features

- 32-bit OPB slave utilizing a 32-bit IPIF Slave SRAM interface
- Handles serial-to-parallel and parallel-to-serial data conversions
- Generates interrupts for "pen-down" and "pen-up" events

Module Port Interface

Information about the signals, pins, and parameters for the module are listed in the following tables: Table 9-1, Table 9-2, Table 9-3, and Table 9-4 .

Table 9-1: **Global Signals**

Name	Direction	Description
OPB_Clk	Input	OPB system clock
OPB_Rst	Input	OPB system reset

Table 9-2: **OPB Slave Signals**

Name	Direction	Description
OPB_ABus[0:31]	Input	OPB address bus
OPB_BE[0:3]	Input	OPB byte enables
OPB_DBus[0:31]	Input	OPB data bus
OPB_RNW	Input	OPB read not write
OPB_select	Input	OPB select
OPB_seqAddr	Input	OPB sequential address
S1_DBus[0:31]	Output	Slave data bus
S1_DBusEn	Output	Slave data bus enable
S1_errAck	Output	Slave error acknowledge
S1_retry	Output	Slave bus cycle retry
S1_toutSup	Output	Slave time-out suppress
S1_xferAck	Output	Slave transfer acknowledge

Table 9-3: **External I/O Pins**

Name	Direction	Description
BUSY	Input	Busy Status flag from Touch Screen Digitizer Chip
CS	Output	Chip Select
DCLK	Output	Serial Data Clock
DIN	Output	Data input to Touch Screen Digitizer Chip
DOUT	Input	Data Output from Touch Screen Digitizer Chip
Intr	Output	Interrupt to CPU
PENIRQ	Input	Pen down interrupt from Touch Screen Digitizer Chip

Table 9-4: **Parameters**

Name	Description
C_BASEADDR	32-bit base address of Touch Screen Controller (must be aligned to 8 byte boundary)
C_HIGHADDR	Upper address boundary, must be set to value of C_BASEADDR + 0x7 (8 byte boundary)

Implementation

The OPB Touch Screen Controller module uses an IPIF slave (with SRAM interface) attached to a state machine to provide a simple interface to the touch screen digitizer (TSD) chip. This state machine begins running when a byte is written to the control register. This control byte is serialized and sent on to the TSD chip. The TSD chip asserts busy while it performs the necessary analog-to-digital conversion. When complete, the TSD deasserts busy and shifts out the result data. The serial data is converted back to parallel form and stored in a data buffer. This buffer can then be read to return the X (horizontal), Y (vertical), or Z (pressure) information.

Another group of logic provides simple debouncing (filtering) of the **PENIRQ** input to look for clean "pen-up"/"pen-down" transitions. The Touch Screen Controller can be operated in a polled or interrupt driven mode. In the polled mode, a status register can be continuously read to determine when the screen is being touched. To reduce the amount of wasted CPU time due to polling, an interrupt driven mode is also supported. In the interrupt driven mode, a level sensitive interrupt is generated for pen-up or pen-down transitions. Once triggered, these interrupts stay actively asserted until the corresponding bits of the interrupt status register are cleared. To reduce the effect of noise causing spurious interrupts, the **PENIRQ** signal from the TSD chip is filtered by only sampling the **PENIRQ** signal at the **DCLK** rate (typically < 1 MHz) instead of the OPB clock rate (typically > 50 MHz). Additionally, **PENIRQ** sampling is disabled during Analog-to-Digital conversion operations in the TSD chip to further prevent unnecessary interrupts.

An internal clock divider divides down the OPB clock by a factor of 200 to generate the serial clock to the TSD chip. It may be necessary to adjust the counter logic in the design to support a different ratio between the OPB clock and TSD serial clock.

Memory Map

Information about the memory mapped registers is shown in Table 9-5.

Table 9-5: **Memory Map**

Register Address	Bits	Read/ Write	Description
Base Address + 0	[0:7]	W	Send Command Byte: A Write to this register will send the corresponding command byte to the TSD Chip. Consult TSD Chip data sheet for a description of the possible command bytes. These command bytes control many different functions in the TSD chip (i.e. initiating X, Y, and Z digitize operations, setting various power mode, etc.).
	[0:3]	R	Returns “0000“
	[4:15]	R	Return Data: Returns 12 bits of data returned by the TSD chip from the previous command sent to it. This data remains until the next time a command byte is sent.
	[16:30]	-	Undefined.
	[31]	R	PENIRQ status: 0=Screen is not being touched 1=BusyScreen is being touched This status bit is typically used when operating in a polled mode. It reflects the current state of the **PENIRQ** input from the TSD chip. Note that the value read in this register is inverted from the actual I/O pin.

Table 9-5: **Memory Map *(Continued)***

Register Address	Bits	Read/ Write	Description
Base Address + 4	[0:29]	-	Undefined
	[30]	R	Pen-Down Interrupt Status Bit: * 0 = Pen-Down condition was not detected 1 = Pen-Down condition was detected
	[30]	W	Pen-Down Interrupt Acknowledge Bit: 0 = Do not Clear/Acknowledge a Pen-Down interrupt 1 = Clear/Acknowledge a Pen-Down interrupt
	[31]	R	Pen-Up Interrupt Status Bit: * 0 = Pen-Up condition was not detected 1 = Pen-Up condition was detected
	[31]	W	Pen-Up Interrupt Acknowledge Bit: 0 = Do not Clear/Acknowledge a Pen-Up interrupt 1 = Clear/Acknowledge a Pen-Up interrupt

* **Note**: If either the Pen-Down or Pen-Up Interrupt Status bit is active, then an interrupt is generated to the CPU on the **Intr** pin.

Chapter 10

OPB AC97 Sound Controller

Overview

This module is an On-Chip Peripheral Bus (OPB) slave device that is designed to control an AC97 Audio Codec chip. It provides a simple memory mapped interface to communicate with the high-speed serial ports of the AC97 Codec. The OPB AC97 Sound Controller module allows full access to all control and status registers in the AC97 chip and provides data buffering for stereo playback and recording.

Related Documents

The following documents provide additional information:

- IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1
- Virtex-II Pro™ Platform FPGAs (Data Sheets)
- Intel AC'97 Specification (http://www.intel.com/labs/media/audio/)

Features

- 16-deep FIFO buffer for record and playback data
- Capable of generating interrupts when play/record FIFOs reach given fullness thresholds

Module Port Interface

Information about the signals, pins, and parameters for the module are listed in tables Table 10-1, Table 10-2, Table 10-3, Table 10-4.

Table 10-1: **Global Signals**

Name	Direction	Description
OPB_Clk	Input	OPB system clock
OPB_Rst	Input	OPB system reset

Table 10-2: **OPB Slave Signals**

Name	Direction	Description
OPB_ABus[0:31]	Input	OPB address bus
OPB_BE[0:3]	Input	OPB byte enables
OPB_DBus[0:31]	Input	OPB data bus
OPB_RNW	Input	OPB read not write
OPB_select	Input	OPB select
OPB_seqAddr	Input	OPB sequential address
OPB_AC97_CONTROLLER_DBus[0:31]	Output	Slave data bus
OPB_AC97_CONTROLLER_errAck	Output	Slave error acknowledge
OPB_AC97_CONTROLLER_retry	Output	Slave bus cycle retry
OPB_AC97_CONTROLLER_toutSup	Output	Slave time-out suppress
OPB_AC97_CONTROLLER_xferAck	Output	Slave transfer acknowledge

Table 10-3: **External I/O Pins**

Name	Direction	Description
Playback_Interrupt	Output	Interrupt generated when play buffer fullness is at or below programmed threshold
Record_Interrupt	Output	Interrupt generated when record buffer fullness is at or above programmed threshold
Bit_Clk	Input	Serial Bit Clock from AC97 Codec
Sync	Output	Frame synchronization signal to AC97 Codec
SData_Out	Output	Serial Data output to AC97 Codec
SData_In	Input	Serial Data input from AC97 Codec

Table 10-4: **Generics (Parameters)**

Name	Default	Description
C_OPB_AWIDTH	32	Address bus width of OPB. Should be set to 32.
C_OPB_DWIDTH	32	Data bus width of OPB. Should be set to 32.
C_BASEADDR	N/A	Base Address of AC97 Sound Controller. Should be set to 256 byte (or higher power of 2) boundary.
C_HIGHADDR	N/A	End Address of AC97 Sound Controller. Should be set to (Base Address + 0xFF) or higher. Total memory space from C_BASEADDR to C_HIGHADDR must be power of 2.

Table 10-4: **Generics (Parameters) *(Continued)***

Name	Default	Description
C_PLAYBACK	1	Playback Enable. Set to "1" to allow playback. Set to "0" to remove playback logic.
C_RECORD	1	Record Enable. Set to "1" to allow record. Set to "0" to remove record logic.
C_PLAY_INTR_LEVEL	2	Sets playback FIFO fullness threshold at which interrupt is generated: 0 = No Interrupt 1 = empty NumWords = 0 2 = halfempty Num Words <= 7 3 = halffull Num Words >= 8 4 = full Num Words = 16
C_REC_INTR_LEVEL	3	Sets record FIFO fullness threshold at which interrupt is generated: 0 = No Interrupt 1 = empty Num Words = 0 2 = halfempty Num Words <= 7 3 = halffull Num Words >= 8 4 = full Num Words = 16

Implementation

Figure 10-1, page 138 shows a block diagram of the OPB AC97 Sound Controller. The OPB AC97 Sound Controller module manages three primary functions to control the AC97 Codec chip. It handles the playback FIFO, record FIFO, and the Codec's control/status registers.

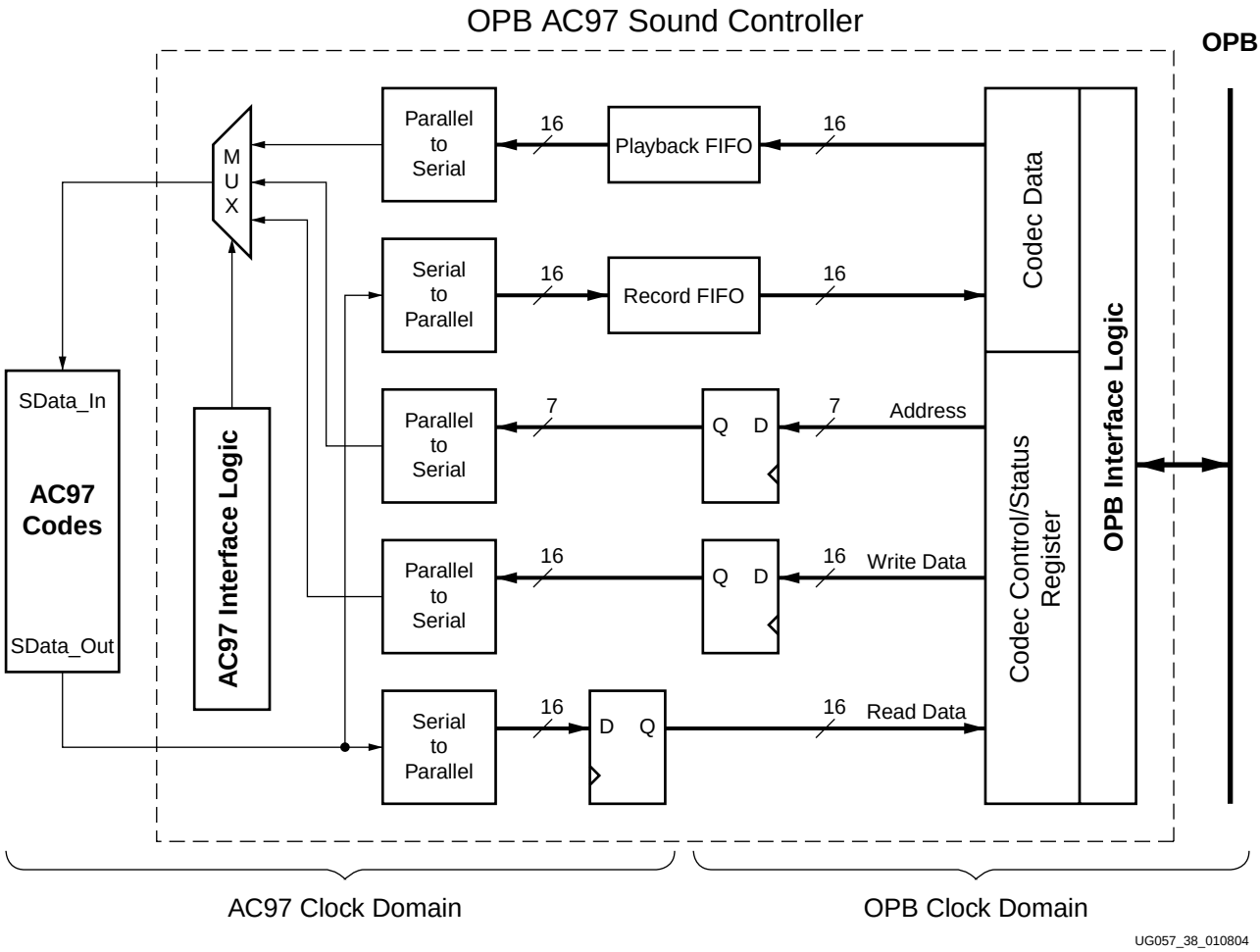


Figure 10-1: **OPB AC97 Sound Controller Block Diagram**

The playback FIFO is a 16 entry deep x 16 bit wide FIFO. The playback data is stored by alternating between Left and Right channel data (beginning with the Left channel). This allows the 16 entry FIFO to store a total of eight stereo data samples. Software should be interrupt driven and programmed to refill the playback FIFO after an interrupt is received stating that the FIFO is nearly empty. If operating in polled mode, the software should poll the playback FIFO full status bit and refill the FIFO when it is not full. If the playback FIFO goes into an underrun condition (FIFO is empty and Codec requests more data), an error flag bit is set. If the playback FIFO is underrun, the FIFO must be reset to clear the error flag and to ensure proper operation. The FIFO threshold at which an interrupt is generated can be set to one of four possible fullness levels. The OPB AC97 controller logic automatically handles the process of serializing the left/right playback data and sending it out to the Codec chip when requested.

The record FIFO is a 16 entry deep x 16 bit wide FIFO. The record data is stored in an alternating fashion between Left and Right channel data (beginning with the Left channel). This allows the 16 entry FIFO to store a total of eight stereo data samples. Software should be interrupt driven and programmed to empty the record FIFO after an interrupt is received stating that the FIFO is nearly full. If operating in polled mode, the software should poll the playback FIFO empty status bit and get data from the FIFO when it is not

empty. If the record FIFO goes into an overrun condition (FIFO is full and Codec sends more data), an error flag bit is set. If the record FIFO is overrun, the FIFO must be reset to clear the error flag and to ensure proper operation. The FIFO threshold at which an interrupt is generated can be set to one of four possible emptiness levels. The OPB AC97 controller logic automatically handles the process of parallelizing the left/right serial record data that is received from the Codec chip.

The playback and record FIFOs must be operated with the same sampling frequency between the left and right channels. The FIFO logic does not support the left and right channels operating at different frequencies.

Access to the control/status registers in the Codec chip is performed through a set of keyhole registers. To write to the control registers in the Codec chip, the write data and then the address to be accessed are written to two registers in the OPB AC97 controller. This causes the write data to be serialized and sent to the Codec chip. A status bit signals when the write is complete. Reading a status register in the Codec chip is performed in a similar manner. The read address is written to the OPB AC97 controller. This causes a read command to be serialized and sent to the Codec chip. When the Codec responds with the read data, a status bit is set indicating that the return data is available. See the "Memory Map" section below for more information about using these registers.

The **Bit_Clk** from the AC97 Codec typically runs at a frequency of 12.288 MHz while the OPB clock runs with a typical frequency of 50-100 MHz. Because of the asynchronous relationship between these two clock domains, the OPB AC97 controller contains special logic to pass data between these two clock domains. In order for this synchronizing logic to function properly, it is important that the OPB clock frequency by at least two times higher than the AC97 **Bit_Clk** frequency.

Memory Map

Information about the memory mapped registers is shown in Table 10-5.

Table 10-5: **Memory Map**

Register Address	Bits	Read/ Write	Description
Base Address + 0	[16:31]	W	Write 16 bit data sample to playback FIFO. Data should be written two at a time to write data to the left channel followed by the right channel.
Base Address + 4	[16:31]	R	Read 16 bit data sample from record FIFO. Data should be read two at a time to get data from the left channel followed by the right channel.

Table 10-5: **Memory Map *(Continued)***

Register Address	Bits	Read/ Write	Description
Base Address + 8	[24]	R	Record FIFO Overrun: 0 = FIFO has not overrun 1 = FIFO has overrun **Note:** Record FIFO must be reset to clear this bit. Once an overrun has occurred, the Record FIFO will not operate properly until it is reset.
	[25]	R	Play FIFO Underrun: 0 = FIFO has not underrun 1 = FIFO has underrun **Note:** Play FIFO must be reset to clear this bit. Once an underrun has occurred, the Play FIFO will not operate properly until it is reset.
	[26]	R	Codec Ready: 0 = Codec is not ready to receive commands or data. (This may occur during initial power-on of immediately after reset.) 1 = Codec ready to run
	[27]	R	Register Access Finish: 0 = AC97 Controller waiting for access to control/status register in Codec to complete. 1 = AC97 Controller is finished accessing the control/status register in Codec. **Note:** This bit is cleared when there is a write to the "AC97 Control Address Register" (described below).
	[28]	R	Record FIFO Empty: 0 = Record FIFO not Empty 1 = Record FIFO Empty
	[29]	R	Record FIFO Full: 0 = Record FIFO not Full 1 = Record FIFO Full
	[30]	R	Playback FIFO Half Full: 0 = Playback FIFO not Half Full 1 = Playback FIFO Half Full
	[31] (LSB)	R	Playback FIFO Full: 0 = Playback FIFO not Full 1 = Playback FIFO Full
Base Address + 12	[30]	W	Clear/Reset Record FIFO: 0 = Do not Reset Record FIFO 1 = Reset Record FIFO. Resetting the record FIFO also clears the "Record FIFO Overrun" status bit.
	[31]	W	Clear/Reset Play FIFO: 0 = Do not Reset Play FIFO 1 = Reset Play FIFO. Resetting the Play FIFO also clears the "Play FIFO Underrun" status bit.

Table 10-5: **Memory Map *(Continued)***

Register Address	Bits	Read/ Write	Description
Base Address + 16	[24:30]	W	AC97 Control Address Register: Sets the 7 bit address of control or status register in the Codec chip to be accessed. Writing to this register clears the "Register Access Finish" status bit.
	[31]	W	AC97 Control Address Register: 0 = Perform a write to the address specified above. The write data comes from the "AC97 Control Data Write Register" which should be set beforehand. 1 = Performs a read to the address above. Writing to this register clears the "Register Access Finish" status bit. This bit will be asserted high when the operation is complete.
Base Address + 20	[24:31]	R	AC97 Status Data Read Register. Returns data from the status register in the Codec that was read by the command above. Data is valid when the "Register Access Finish" flag is set.
Base Address + 24	[24:31]	W	AC97 Control Data Write Register. Contains the data to be written to the control register in the Codec. This register is used in conjunction with the "AC97 Control Address Register" described above.

XILINX®

Chapter 11

OPB to PLB Bridge-In Module (Lite)

Overview

The OPB to PLB Bridge-In module translates OPB transactions into PLB transactions. It functions as a slave on the OPB side and a master on the PLB side. The Bridge-In is necessary in systems where an OPB master device requires access to PLB slave devices. This document describes the "Lite" or simplified version of the OPB to PLB Bridge. It is fully capable of translating data transfers from OPB to PLB, but does not have large data buffering capabilities. This makes the overall design much smaller in terms of FPGA resource utilization, but also reduces the potential bandwidth for data transfers. For systems requiring maximum bandwidth across the OPB to PLB Bridge, the full version (available through EDK) should be used. The implementation of the OPB to PLB Bridge uses a slightly modified version of the Xilinx Intellectual Property InterFace (IPIF) module to simplify its design.

Related Documents

The following documents provide additional information

- IPIF Specification
- IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1
- IBM CoreConnect™ 64-Bit Processor Local Bus: Architecture Specification
- Virtex-II Pro™ Platform FPGAs (Data Sheets)

Features

- 32-bit OPB slave interface utilizing a 32-bit IPIF Slave SRAM interface
- 64-bit PLB master interface
- Configurable address decoders in bridge memory window
- Small size: low FPGA resource utilization
- Logic that prevents deadlock conditions when used in systems containing the PLB to OPB Bridge

Module Port Interface

Table 11-1: **Global Signals**

Name	Direction	Description
OPB_Clk	Input	OPB system clock
OPB_Rst	Input	OPB system reset
PLB_Clk	Input	PLB system clock
PLB_Rst	Input	PLB system reset

Table 11-2: **OPB Slave Signals**

Name	Direction	Description
OPB_ABus[0:31]	Input	OPB address bus
OPB_BE[0:3]	Input	OPB byte enables
OPB_DBus[0:31]	Input	OPB data bus
OPB_RNW	Input	OPB read not write
OPB_select	Input	OPB select
OPB_seqAddr	Input	OPB sequential address
Sl_DBus[0:31]	Output	Slave data bus
S1_errAck	Output	Slave error acknowledge
S1_retry	Output	Slave bus cycle retry
S1_toutSup	Output	Slave time-out suppress
S1_xferAck	Output	Slave transfer acknowledge

Table 11-3: **PLB Master Signals**

Name	Direction	Description
PLB_MnAddrAck	Input	PLB master address acknowledge
PLB_MnBusy	Input	PLB master slave busy indicator
PLB_MnErr	Input	PLB master slave error indicator
PLB_MnRdBTerm	Input	PLB master terminate read burst indicator
PLB_MnRdDAck	Input	PLB master read data acknowledge
PLB_MnRdDBus[0:63]	Input	PLB master read data bus
PLB_MnRdWdAddr[0:3]	Input	PLB master read word address
PLB_MnRearbitrate	Input	PLB master bus rearbitrate indicator
PLB_Mnssize[0:1]	Input	PLB slave data bus size

Table 11-3: **PLB Master Signals *(Continued)***

Name	Direction	Description
PLB_MnWrBTerm	Input	PLB master terminate write burst indicator
PLB_MnWrDAck	Input	PLB master write data acknowledge
PLB_pendPri[0:1]	Input	PLB pending request priority
PLB_pendReq	Input	PLB pending bus request indicator
PLB_reqPri[0:1]	Input	PLB current request priority
Mn_abort	Output	Master abort bus request indicator
Mn_ABus[0:31]	Output	Master address bus
Mn_BE[0:7]	Output	Master byte enables
Mn_busLock	Output	Master bus lock
Mn_compress	Output	Master compressed data transfer indicator
Mn_guarded	Output	Master guarded transfer indicator
Mn_lockErr	Output	Master lock error indicator
Mn_msize[0:1]	Output	Master data bus size
Mn_ordered	Output	Master synchronize transfer indicator
Mn_priority[0:1]	Output	Master bus request priority
Mn_rdBurst	Output	Master burst read transfer indicator
Mn_request	Output	Master bus request
Mn_RNW	Output	Master read/not write
Mn_size[0:3]	Output	Master transfer size
Mn_type[0:2]	Output	Master transfer type
Mn_wrBurst	Output	Master burst write transfer indicator
Mn_wrDBus[0:63]	Output	Master write data bus

Table 11-4: **DCR Slave Signals**

Name	Direction	Description
DCR_ABus[0:9]	Input	DCR Address Bus
DCR_DBusIn[0:31]	Input	DCR Data Bus In
DCR_Read	Input	DCR Read Strobe
DCR_Write	Input	DCR Write Strobe
DCR_Ack	Output	DCR Acknowledge
DCR_DBusOut[0:31]	Output	DCR Data Bus Out

Table 11-5: **Miscellaneous I/O Pins**

Name	Direction	Description
Bus_Error_Det	Output	Bus Error Detected - Interrupt Request
BGI_Trans_Abort	Input	Retry Request - Prevents deadlock between bridges

Table 11-6: **Parameters**

Name	Default	Description
C_CLK_ASYNC	1	Specifies if OPB and PLB clocks are synchronous (= 0) or asynchronous (= 1) to each other. The synchronous setting can only be used if the PLB clock is an integer frequency multiple of the OPB clock and if the two clocks are rising edge phase aligned. Setting this parameter to 1 will increase the latency through the bridge.
C_CLK_SAME	0	Specifies that OPB and PLB clocks are driven by the same global clock buffer (= 1). Set to 0 if clocks are not driven from the same global buffer. Setting this parameter to 1 is only allowed if C_CLK_ASYNC = 0.
C_PRECISE_ABORTS	0	Specifies that an aborted OPB write must never cause a PLB write transaction to be requested (=1). Otherwise set to 0. This parameter should normally be set to 0 and is mainly provided to support special OPB master devices requiring precise aborts. Setting this parameter to 1 will increase the latency through the bridge.
C_NUM_ADDR_RNG	2	Specifies number of address range comparators are used to decode OPB addresses destined for the PLB. Valid values are 1 or 2 only.
C_RNG0_BASEADDR	N/A	32-bit lower address boundary of range comparator #0.
C_RNG0_HIGHADDR	N/A	32-bit upper address boundary of range comparator #0. Note: this parameter has no effect on hardware, it is used only for EDK address checking.

Table 11-6: **Parameters *(Continued)***

Name	Default	Description
C_RNG0_ADDR_LSB	3	Least significant bit for address comparator #0 to look at when checking addresses. For example: 0 = 2 GB address comparator 1 = 1 GB address comparator ... 11 = 1 MB address comparator
C_RNG1_BASEADDR	N/A	32-bit lower address boundary of range comparator #1. Valid only if C_NUM_ADDR_RNG = 2
C_RNG1_HIGHADDR	N/A	32-bit upper address boundary of range comparator #0. Note: this parameter has no effect on hardware, it is used only for EDK address checking. Valid only if C_NUM_ADDR_RNG = 2
C_RNG1_ADDR_LSB	3	Least significant bit for address comparator #1 to look at when checking addresses. For example: 0 = 2 GB address comparator 1 = 1 GB address comparator ... 11 = 1 MB address comparator

Implementation

High Level Description

Figure 11-1, page 148 provides a high-level overview of the design of the OPB to PLB Bridge. OPB transactions destined for the PLB are first received and decoded in the OPB IPIF slave logic. The IPIF simplifies the design of the bridge since it converts the OPB transactions into a simpler SRAM-like protocol.

On write transactions, up to 32 bits of write data is buffered. The OPB interface logic then signals the PLB interface logic with the necessary information to begin the PLB transaction at the address specified. Once the PLB write transaction is complete, the next data transfer can begin.

On read transactions, the OPB interface logic signals the PLB interface logic with the necessary information to begin the PLB transaction at the address specified. Up to 32 bits of read data is then latched from the PLB to be sent back to the OPB. When the PLB transaction is complete, the PLB interface logic signals the OPB interface so that another transaction can be performed.

The PLB and OPB logic is decoupled so that the OPB and PLB clocks can be different. In order to reduce the FPGA resource utilization, this “Lite” version of the bridge supports only minimal data buffering. The Lite Bridge design can only transfer 32 bits of data per OPB-to-PLB transaction. Therefore, on the OPB interface it will only accept a single 32-bit data transfer at a time and must complete the transfer over the PLB before the next piece of

data can be transferred. This will result in a lower maximum throughput compared to the full bridge.

The Lite version of the OPB to PLB Bridge does not implement DCR registers. The module contains port declarations for a DCR interface, but this is only provided for compatibility with the full bridge. The DCR interface holds **DCR_Ack** at 0 and passes data through from **DCR_DBusIn** to **DCR_DBusOut**. Also, the **Bus_Error_Det** interrupt bit is not implemented, so it is driven to 0.

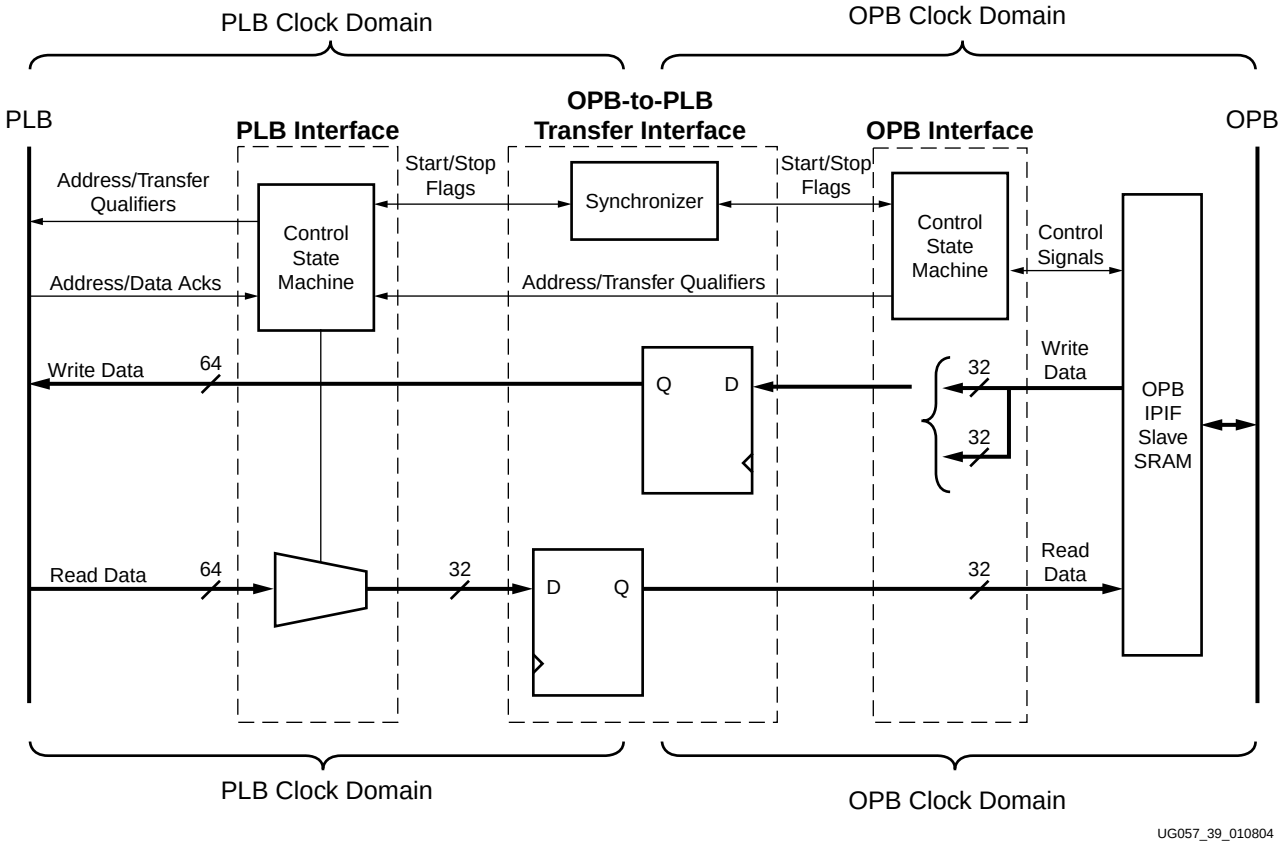


Figure 11-1: **High-Level Overview of OPB to PLB Bridge**

OPB Interface

The OPB slave interface is only designed to respond as a 32 bit byte-enable device. It does not support dynamic bus sizing transactions.

Address Decode Cycle

OPB transactions in the IPIF begin with an address decode cycle. An extra clock cycle is used to decode the address and transfer qualifiers of an OPB request before the request is presented on the IP side of the IPIF.

The OPB interface is designed to handle a single data transfer at a time. It must complete a transaction before it will accept another transaction. While it is busy performing a transaction it will assert **Sl_Retry** on any subsequent transactions. Since data is transferred one word (32 bits) at a time and there is no FIFO buffering capability, the **OPB_seqAddr** signal is ignored.

In order to prevent deadlock between the PLB to OPB Bridge and the OPB to PLB Bridge, a **BGI_Trans_Abort** signal goes from the PLB to OPB Bridge to the OPB to PLB Bridge. The signal causes the OPB to PLB Bridge to issue a retry over OPB. This forces the OPB to PLB Bridge to relinquish the OPB when the PLB to OPB Bridge is waiting for the bus. Note that **BGI_Trans_Abort** can only interrupt a read transaction. Since writes are buffered by the OPB to PLB Bridge, they do not cause deadlock. Also note that the PLB to OPB Bridge also buffers write data so it only needs to assert **BGI_Trans_Abort** when it has an OPB read pending. If an OPB read transaction is interrupted by **BGI_Trans_Abort**, but the read transaction has already been requested over PLB, the PLB transaction is allowed to complete, but the result will be discarded.

The OPB interface is designed to handle OPB master abort conditions. An OPB master abort occurs when the master deasserts **Mn_Select** before a transaction has completed. By default, the handling of aborts is "imprecise." This means that an aborted OPB write transaction may still cause a PLB write transaction to be requested. If this is not acceptable, the parameter **C_PRECISE_ABORTS** can be set to "1" to guarantee that aborted writes never result in a PLB write being generated. It is uncommon for OPB masters to utilize abort functionality, so it is recommended that this parameter be set to "0" to reduce the overall latency of the bridge.

Write Transactions

On a write transaction from the OPB, the write data is registered allowing the OPB side of the transaction to be acknowledged and completed. Next, the OPB interface logic signals to the PLB interface logic to carry out the transaction over the PLB. It then provides relevant information such as the byte enable values, destination address, and write data. The OPB interface logic then waits for confirmation that the data transfer is complete before it can accept another OPB transaction.

Read Transactions

On a read transaction, data must be requested from the PLB before the OPB transaction can be acknowledged. A single word of data is requested from PLB to complete the transaction. When the read data is available, the PLB interface signals that the data is present.

Transfer Interface

The transfer interface facilitates the movement of data between the OPB and PLB interface logic, which may be operating in different clock domains. Control signals are passed through synchronizing logic to handle the transition between OPB and PLB clock domains. Before any control signals are asserted, the data being transferred between the OPB and PLB interfaces is latched and held valid. The user has the option to specify either a synchronous or asynchronous timing relationship between the PLB and OPB clocks. If the clocks are known to be synchronous, then much of the transfer interface logic can be simplified to reduce both logic utilization and latency. The PLB clock frequency must be greater than or equal to the frequency of the OPB clock.

PLB Interface

The PLB interface logic initiates PLB read/write transactions as a PLB master to transfer data as requested by the OPB interface logic. Though the PLB interface is 64 bits wide, it will not request more than 32 bits of data be transferred at a time.

Chapter 12

OPB PS/2 Controller (Dual)

Overview

This module is an On-Chip Peripheral Bus (OPB) slave device that is designed to control two PS/2 devices such as a mouse and keyboard. It utilizes the Xilinx Intellectual Property InterFace (IPIF) to simplify its design. The OPB PS/2 Controller module generates interrupts upon various transmit or receive conditions. This document assumes the user is already familiar with the PS/2 interface protocol. Additional information about PS/2 ports and peripherals is widely available on the Internet or through a computer hardware reference manual.

Related Documents

The following documents provide additional information:

- IPIF Specification
- IBM CoreConnect™ 64-Bit On-Chip Peripheral Bus: Architecture Specifications, Version 2.1
- Virtex-II Pro™ Platform FPGAs (Data Sheets)

Features

- 32-bit OPB slave utilizing a 32-bit IPIF Slave SRAM interface
- Implements 8-bit read/write interface found in many PCs to control each PS/2 port

Module Port Interface

Information about the signals, pins, and parameters for the module are listed in the following tables.

Table 12-1: **OPB Slave Signals**

Name	Direction	Description
IPIF_Rst	Input	OPB system reset
OPB_BE[0:3]	Input	OPB byte enables
OPB_Select	Input	OPB select
OPB_Dbus[0:31]	Input	OPB data bus
OPB_Clk	Input	OPB system clock

Table 12-1: **OPB Slave Signals**

Name	Direction	Description
OPB_Abus[0:31]	Input	OPB address bus
OPB_RNW	Input	OPB read not write
OPB_seqAddr	Input	OPB sequential address
Sln_XferAck	Output	Slave transfer acknowledge
Sln_Dbus[0:31]	Output	Slave data bus
Sln_DBusEn	Output	Slave data bus enable
Sln_errAck	Output	Slave error acknowledge
Sln_retry	Output	Slave bus cycle retry
Sln_toutSup	Output	Slave timeout suppress

Table 12-2: **External I/O Pins**

Name	Direction	Description
Sys_Intr1	Output	Interrupt, Port #1
Clkin1	Input	PS/2 Clock In, Port #1
Clkpd1	Output	PS/2 Clock Pulldown, Port #1
Rx1	Input	PS/2 Serial Data In, Port #1
Txpd1	Output	PS/2 Serial Data Out Pulldown, Port #1
Sys_Intr2	Output	Interrupt, Port #2
Clkin2	Input	PS/2 Clock In, Port #2
Clkpd2	Output	PS/2 Clock Pulldown, Port #2
Rx2	Input	PS/2 Serial Data In, Port #2
Txpd2	Output	PS/2 Serial Data Out Pulldown, Port #2

Table 12-3: **Parameters**

Name	Description
C_BASEADDR	32-bit base address of PS/2 controller (must be aligned to 8K Byte boundary)
C_HIGHADDR	Upper address boundary, must be set to value of C_BASEADDR + 0x1FFF (8K Byte boundary)

Implementation

Figure 12-1 shows a block diagram of the OPB PS/2 Controller module. It uses an IPIF slave with an SRAM interface in addition to simple state machines and shift registers to implement its functionality. Each PS/2 port is controlled by a separate set of eight byte-wide registers.

For transmitting data, a byte write to the transmit register will cause that data to be serialized and sent to the PS/2 device. Status registers and interrupts then signal when the transmission is complete and if there are any errors reported. Similarly, receiver status registers and interrupts signal when data has been received from the PS/2 device. Any errors with received data are also reported.

The PS/2 controller can be operated in a *polled* mode or an *interrupt driven* mode. In the interrupt driven mode, separate register bits for setting, clearing, and masking of individual interrupts are provided.

Since the PS/2 interface uses a open collector circuit for transmitting data, the output signals **Clkpd** and **Txpd** should be tied to a transistor or logic gate capable of pulling the 5V PS/2 clock and data signals low. Note that the PS/2 protocol specifies 5V signalling. Therefore, it is necessary to have the proper interface circuitry to prevent over-voltage conditions on the FPGA I/O. Consult the schematics and documentation for the Xilinx ML300 board for an example implementation of a PS/2 port interface circuit.

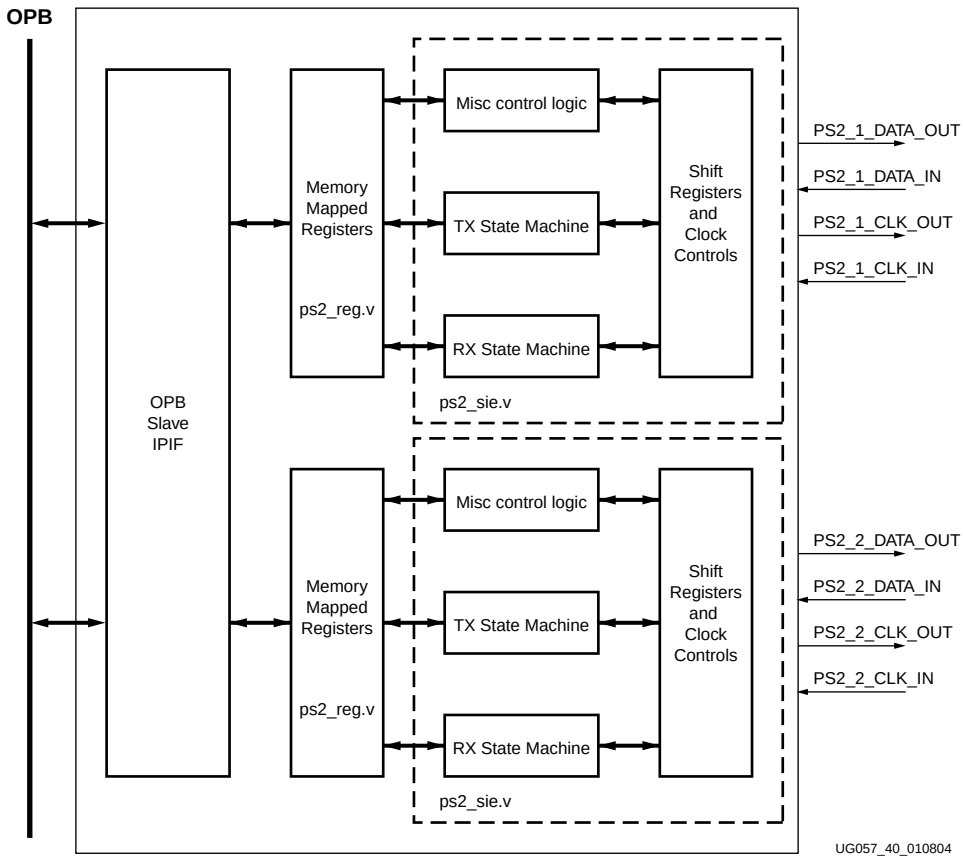


Figure 12-1: **OPB PS/2 Controller Block Diagram**

Memory Map

Information about the memory mapped registers is shown in Table 12-4.

Note:

1. Control/status registers for PS/2 Port #1 start at the base address (value of parameter C_BASEADDR).
2. Control/status registers for PS/2 Port #2 start at the base address + 0x1000 (value of parameter C_BASEADDR + 0x1000).

Table 12-4: **Memory Map Table**

Offset	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8-31
x00	Reserved							SRST	R*
x04	Reserved						STR.6 tx_full_sta	STR.7 rx_full_sta	R*
x08	RXR								R*
x0c	TXR								R*
x10	Reserved		INSTA.2 rx_full	INSTA.3 rx_err	INSTA.4 rx_ovf	INSTA.5 tx_ackf	INSTA.6 tx_noack	INSTA.7 wdt_tout	R*
x14	Reserved		INTCLR.2 rx_full	INTCLR.3 rx_err	INTCLR.4 rx_ovf	INTCLR.5 tx_ackf	INTCLR.6 tx_noack	INTCLR.7 wdt_tout	R*
x18	Reserved		INTMSET.2 rx_full	INTMSET.3 rx_err	INTMSET.4 rx_ovf	INTMSET.5 tx_ackf	INTMSET.6 tx_noack	INTMSET.7 wdt_tout	R*
x1c	Reserved		INTMCLR.2 rx_full	INTMCLR.3 rx_err	INTMCLR.4 rx_ovf	INTMCLR.5 tx_ackf	INTMCLR.6 tx_noack	INTMCLR.7 wdt_tout	R*

* R = Reserved

All fields marked “Reserved” will return zero. The field in INTSTA(x10) will be AND'ed with the fields in INTM(x18), then the bits get OR'ed to form a single Interrupt signal.

The register pairs INTSTA/INTCLR and INTMSET/INTMCLR are implemented to allow single bit register updates which reduce the latency of interrupt handling. INTCLR and INTMCLR are helper functions that make setting and clearing the Interrupt Status Register and Interrupt Mask Register faster.

The register definitions are shown in Table 12-5, page 155 (this table spans several pages).

Note:

The second PS/2 Port has an identical set of control/status registers at an additional offset of 0x1000.

Table 12-5: **OPB PS/2 Slave Device Pin Description**

Name	Field Name	Bit	Direction	Description
Base Address + 0 (Offset x00)	SRST	7	W	Software Reset. Writing '1' into this register results in the PS/2 controller being reset to idle state. Also, registers at offset x04, x10, x14 will be reset by this bit as well.
Base Address + 4 (Offset x04)	STR.6 tx_full_sta	6	R	TX Register Full. PS/2 Serial Interface Engine is busy. This register can only be modified by PS/2 SIE hardware. Software does not have direct write permission to change this field since this field is set by the state machine in the SIE. Software can clear this field indirectly is by using the SRST register.
	STR.7 rx_full_sta	7	R	RX Register Full. PS/2 Serial Interface Engine received a byte package. The associated interrupt “rx_full” (INTSTA.3) will also be set. Software does not have direct write permission to change this field since this field is set by the state machine in the SIE. Software can clear this field indirectly is by using the SRST register.
Base Address + 8 (Offset x08)	RXR	[0:7]	R	RX received data.
Base Address + 12 (Offset x0c)	TXR	[0:7]	W	TX transmission data.

Table 12-5: **OPB PS/2 Slave Device Pin Description *(Continued)***

Name	Field Name	Bit	Direction	Description
Base Address + 16 (Offset x10)	INSTA.2 rx_full	2	R	Interrupt Status Register - RX data register full. This field will be updated by the PS/2 Serial Interface when the SIE has received a data packet. Software can clear this field by writing a '1' into the corresponding interrupt clear register INTCLR.2 (offset x14.2)
	INSTA.3 rx_err	3	R	Interrupt Status Register - RX data error. This field will be updated by the PS/2 Serial Interface when the SIE has found that RX data is a bad packet. Software can clear this field by writing a '1' into the corresponding interrupt clear register INTCLR.3 (offset x14.3)
	INSTA.4 rx_ovf	4	R	Interrupt Status Register - RX data register overflow. This field will be updated by the PS/2 Serial Interface when the SIE overwrites a data packet before the previous data was read. Software can clear this field by writing an '1' into the corresponding interrupt clear register INTCLR.4 (offset x14.4)
	INSTA.5 tx_ackf	5	R	Interrupt Status Register - TX acknowledge received. This field will be updated by the PS/2 Serial Interface when the SIE completes transmission of a data byte and has received acknowledgement from the PS/2 device.Software can clear this field by writing an '1' into the corresponding interrupt clear register INTCLR.5 (offset x14.5)
	INSTA.6 tx_noack	6	R	Interrupt Status Register - TX acknowledge not received. This field will be updated by the PS/2 Serial Interface when the SIE completes transmission of a data byte but has not yet received acknowledgement from the PS/2 device.Software can clear this field by writing an '1' into the corresponding interrupt clear register INTCLR.6 (offset x14.6)
	INSTA.7 wdt_tout	7	R	Interrupt Status Register - Watch dog timer timeout. This field will be updated by the PS/2 Serial Interface when the SIE does not receive a PS/2 Clock while a packet is still being transmitted. Software can clear this field by writing an '1' into the corresponding interrupt clear register INTCLR.7 (offset x14.7)

Table 12-5: **OPB PS/2 Slave Device Pin Description *(Continued)***

Name	Field Name	Bit	Direction	Description
Base Address + 20 (Offset x14)	INTCLR.2 rx_full	2	R*/W	Interrupt Clear Register - RX data register full. Writing a '1' to this field will clear INTSTA.2. Writing a '0' has no effect.
	INTCLR.3 rx_err	3	R*/W	Interrupt Clear Register - RX data error. Writing a '1' to this field will clear INTSTA.3. Writing a '0' has no effect.
	INTCLR.4 rx_ovfl	4	R*/W	Interrupt Clear Register - RX data register overflow. Writing a '1' to this field will clear INTSTA.4. Writing a '0' has no effect.
	INTCLR.5 tx_ack	5	R*/W	Interrupt Clear Register - TX acknowledge received. Writing a '1' to this field will clear INTSTA.5. Writing a '0' has no effect.
	INTCLR.6 tx_noak	6	R*/W	Interrupt Clear Register - TX acknowledge not received. Writing a '1' to this field will clear INTSTA.6. Writing a '0' has no effect.
	INTCLR.7 wdt_toutl	7	R*/W	Interrupt Clear Register - Watch dog timer timeout. Writing a '1' to this field will clear INTSTA.7. Writing a '0' has no effect.
* If software tries to read from INTCLR (offset x14), the value of INTSTA (offset x10) will be returned.				
Base Address + 24 (Offset x18)	INTMSET.2 rx_full	2	R*/W	Interrupt Mask Set Register - RX data register full. Writing a '1' to this field will set INTM.2. Writing a '0' has no effect.
	INTMSET.3 rx_err	3	R*/W	Interrupt Mask Set Register - RX data error. Writing a '1' to this field will set INTM.3. Writing a '0' has no effect.
	INTMSET.4 rx_ovf	4	R*/W	Interrupt Mask Set Register - RX data register overflow. Writing a '1' to this field will set INTM.4. Writing a '0' has no effect.
	INTMSET.5 tx_ack	5	R*/W	Interrupt Mask Set Register - TX acknowledge received. Writing a '1' to this field will set INTM.5. Writing a '0' has no effect.
	INTMSET.6 tx_noack	6	R*/W	Interrupt Mask Set Register - TX acknowledge not received. Writing a '1' to this field will set INTM.6. Writing a '0' has no effect.
	INTMSET.7 wdt_tout	7	R*/W	Interrupt Mask Set Register - Watch dog timer timeout. Writing a '1' to this field will set INTM.7. Writing a '0' has no effect.
* If software tries to read from INTMSET (offset x18), the value of INTM register will be returned.				

Table 12-5: **OPB PS/2 Slave Device Pin Description *(Continued)***

Name	Field Name	Bit	Direction	Description
Base Address + 28 (Offset x1C)	INTMCLR.2 rx_full	2	R*/W	Interrupt Mask Clear Register - RX data register full. Writing a '1' to this field will clear INTM.2. Writing a '0' has no effect.
	INTMCLR.3 rx_err	3	R*/W	Interrupt Mask Clear Register - RX data error. Writing a '1' to this field will clear INTM.3. Writing a '0' has no effect.
	INTMCLR.4 rx_ovf	4	R*/W	Interrupt Mask Clear Register - RX data register overflow. Writing a '1' to this field will clear INTM.4. Writing a '0' has no effect.
	INTMCLR.5 rx_ack	5	R*/W	Interrupt Mask Clear Register - TX acknowledge received. Writing a '1' to this field will clear INTM.5. Writing a '0' has no effect
	INTMCLR.6 rx_noack	6	R*/W	Interrupt Mask Clear Register - TX acknowledge not received. Writing a '1' to this field will clear INTM.6. Writing a '0' has no effect.
	INTMCLR.7 wdt_tout	7	R*/W	Interrupt Mask Clear Register - Watch dog timer timeout. Writing a '1' to this field will clear INTM.7. Writing a '0' has no effect.

* If software tries to read from IINTMCLR (offset x1C), the value of INTM (offset x18) will be returned.

Chapter 13

PLB TFT LCD Controller

Overview

The PLB TFT LCD Controller is a hardware display controller for a 640x480 resolution VGA screen. It is capable of showing up to 256K colors and is designed for the ***NEC TFT Color LCD Module NL6448BC20-08*** that is mounted on the Xilinx ML300 board. The design contains a PLB master interface that reads video data from a PLB attached memory device (not part of this design) and displays the data onto the TFT screen. The design also contains a Device Control Register (DCR) interface used for configuring the controller.

Related Documents

The following documents provide additional information

- IBM CoreConnect™ 32-Bit Device Control Register Bus: Architecture Specifications
- IBM CoreConnect™ 64-Bit Processor Local Bus: Architecture Specification
- Virtex-II Pro™ Platform FPGAs (Data Sheets)
- NEC TFT Color LCD Module: NL6448BC20-08 (http://www.nec-lcd.com/english/pdf/en0442ej.pdf)

Features

- 32-bit DCR slave interface for control registers
- 64-bit PLB master interface for fetching pixel data
- Support for asynchronous PLB and TFT clocks

Module Port Interface

Table 13-1: **Global Signals**

Name	Direction	Description
SYS_dcrClk	Input	DCR System Clock
SYS_plbClk	Input	PLB System Clock
SYS_plbReset	Input	PLB System Reset
SYS_tftClk	Input	TFT Video Clock

Table 13-2: **PLB Master Signals**

Name	Direction	Description
PLB_MnAddrAck	Input	PLB master address acknowledge
PLB_MnBusy	Input	PLB master slave busy indicator
PLB_MnErr	Input	PLB master slave error indicator
PLB_MnRdBTerm	Input	PLB master terminate read burst indicator
PLB_MnRdDAck	Input	PLB master read data acknowledge
PLB_MnRdDBus[0:63]	Input	PLB master read data bus
PLB_MnRdWdAddr[0:3]	Input	PLB master read word address
PLB_MnRearbitrate	Input	PLB master bus rearbitrate indicator
PLB_Mnssize[0:1]	Input	PLB slave data bus size
PLB_MnWrBTerm	Input	PLB master terminate write burst indicator
PLB_MnWrDAck	Input	PLB master write data acknowledge
PLB_pendPri[0:1]	Input	PLB pending request priority
PLB_pendReq	Input	PLB pending bus request indicator
PLB_reqPri[0:1]	Input	PLB current request priority
Mn_abort	Output	Master abort bus request indicator
Mn_ABus[0:31]	Output	Master address bus
Mn_BE[0:7]	Output	Master byte enables
Mn_busLock	Output	Master bus lock
Mn_compress	Output	Master compressed data transfer indicator
Mn_guarded	Output	Master guarded transfer indicator
Mn_lockErr	Output	Master lock error indicator
Mn_msize[0:1]	Output	Master data bus size
Mn_ordered	Output	Master synchronize transfer indicator
Mn_priority[0:1]	Output	Master bus request priority
Mn_rdBurst	Output	Master burst read transfer indicator
Mn_request	Output	Master bus request
Mn_RNW	Output	Master read/not write
Mn_size[0:3]	Output	Master transfer size
Mn_type[0:2]	Output	Master transfer type
Mn_wrBurst	Output	Master burst write transfer indicator
Mn_wrDBus[0:63]	Output	Master write data bus

Table 13-3: **DCR Slave Signals**

Name	Direction	Description
DCR_ABus[0:9]	Input	DCR Address Bus
DCR_DBusIn[0:31]	Input	DCR Data Bus In
DCR_Read	Input	DCR Read Strobe
DCR_Write	Input	DCR Write Strobe
DCR_Ack	Output	DCR Acknowledge
DCR_DBusOut[0:31]	Output	DCR Data Bus Out

Table 13-4: **External Output Pins**

Name	Direction	Description
TFT_LCD_HSYNC	Output	Horizontal Sync (Negative Polarity)
TFT_LCD_VSYNC	Output	Vertical Sync (Negative Polarity)
TFT_LCD_DE	Output	Data Enable
TFT_LCD_CLK	Output	Video Clock
TFT_LCD_DPS	Output	Selection of Scan Direction
TFT_LCD_R[5:0]	Output	Red Pixel Data
TFT_LCD_G[5:0]	Output	Green Pixel Data
TFT_LCD_B[5:0]	Output	Blue Pixel Data

Table 13-5: **Parameters**

Name	Default	Description
C_DCR_BASEADDR	N/A	Base address of DCR control registers. Must be aligned on an even DCR address boundary (least significant bit = 0)
C_DCR_HIGHADDR	N/A	Upper address boundary, must be set to value of C_DCR_BASEADDR + 1
C_DEAFULT_TFT_BASE_ADDR[0:10]	N/A	Most significant bits of base address for video memory. The 11 most significant bits of this address define the 2 MB region of memory used for the video frame storage.

Table 13-5: **Parameters *(Continued)***

Name	Default	Description
C_DPS_INIT	1	Initial Reset State of DPS control bit: 0 = DPS output bit resets to 0. This initializes the display to use a normal scan direction. 1 = DPS output bit resets to 1. This initializes the display to use a reverse scan direction (rotates screen 180 degrees).
C_ON_INIT	1	Initial Reset State of TFT enable/disable bit: 0 = Disable TFT display on reset. The causes a black screen to be displayed on reset. 1 = Enable TFT display on reset. The causes the PLB TFT LCD Controller to operate normally on reset.

Hardware

Implementation

Figure 13-1, page 163 shows a high-level block diagram of the design. The PLB TFT LCD Controller has a PLB master interface that reads pixel data from an external PLB memory device. It reads the pixel data for each display line using a series of 8-word cacheline transactions. The pixel data is stored in an internal line buffer and then sent out to the TFT display with the necessary timing to correctly display the image. The video memory is arranged so that each RGB pixel is represented by a 32-bit word in memory (See "Memory Map," page 166). As each line interval begins, data is fetched from memory, buffered, and then displayed. This process repeats continuously over every line and frame to be displayed on the 640x480 VGA TFT screen.

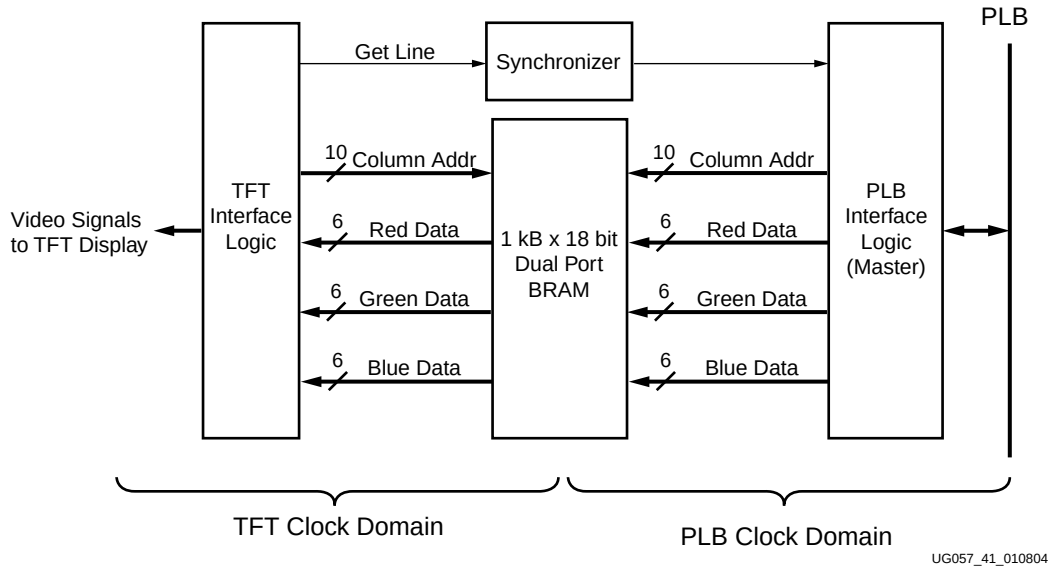


Figure 13-1: **High-Level Block Diagram**

The back-end logic driving the TFT display operates in the same clock domain as the video clock. It reads out data from the dual port line buffer and transmits the pixel data to the TFT. The back-end logic automatically handles the timing of all the video synchronization signals including back porch and front porch blanking. See "Video Timing," page 164 for more information.

The PLB TFT LCD Controller allows for the PLB clock and TFT video clocks to be asynchronous to each other. Special logic allows control signals to be passed between asynchronous PLB and TFT clock domains. A dual port BRAM is used as the line buffer to pass video data between the two clock domains.

It is important to design the system so that there is sufficient bandwidth between the PLB TFT LCD Controller and the PLB memory device to meet the video bandwidth requirements of the TFT. Furthermore, there must be enough available bandwidth left over for the rest of the system. If more bandwidth is needed for the rest of the system, the TFT clock frequency can be reduced. However, reducing the TFT clock frequency also lowers the refresh rate of the screen. This may lead to a noticeable flicker on the screen if the TFT clock is too slow.

The PLB interface logic has the ability to skip reading a line of data if it fails to finish reading data from a previous line. This prevents temporary shortages of available PLB bandwidth from causing the PLB TFT controller from losing synchronization between the PLB and TFT interface logic. Note that extreme shortages of available bandwidth for the PLB TFT controller may cause the screen to appear "unstable" as stale lines of video data are displayed on the screen.

A DCR interface allows software to change the base address of video memory to be read from. This allows frames of video to be drawn in other memory locations without being seen on the display. The software can then change the video memory base address to display a different frame when it is ready. The DCR interface also allows the display to be rotated by 180 degrees or turned off. When the display is turned off a black screen is output while the PLB interface stops requesting data.

Video Timing

The diagrams in Figure 13-2 through Figure 13-5 describe the timing of video signals from the PLB TFT LCD Controller.

Figure 13-2: **Hsync and TFT Clock**

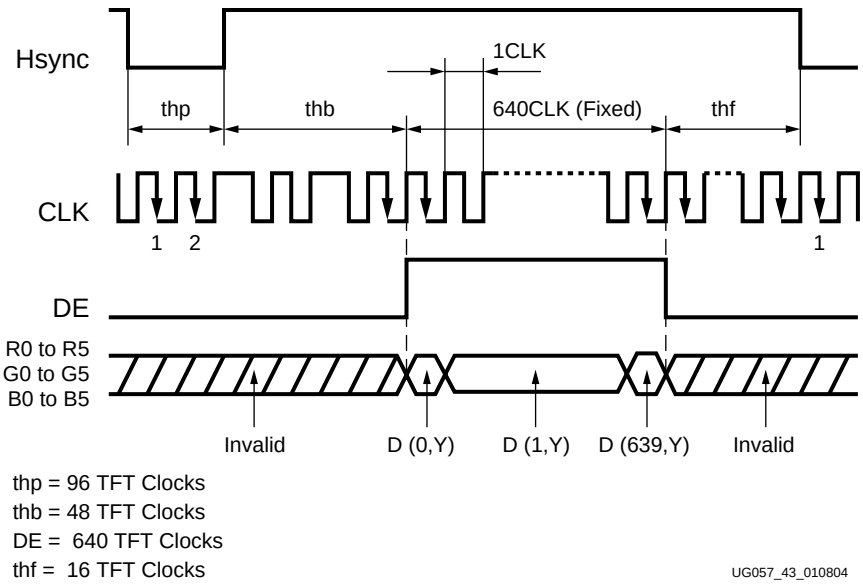
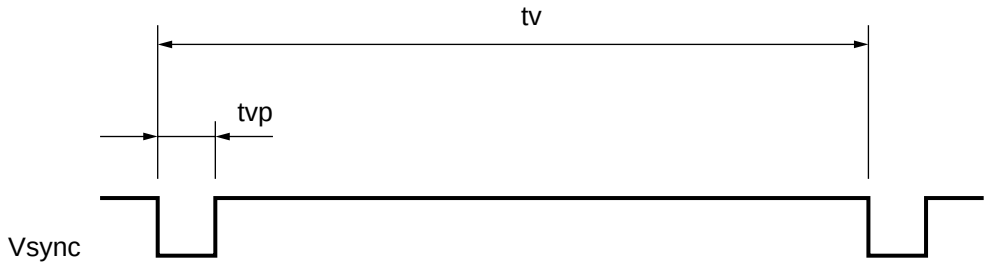


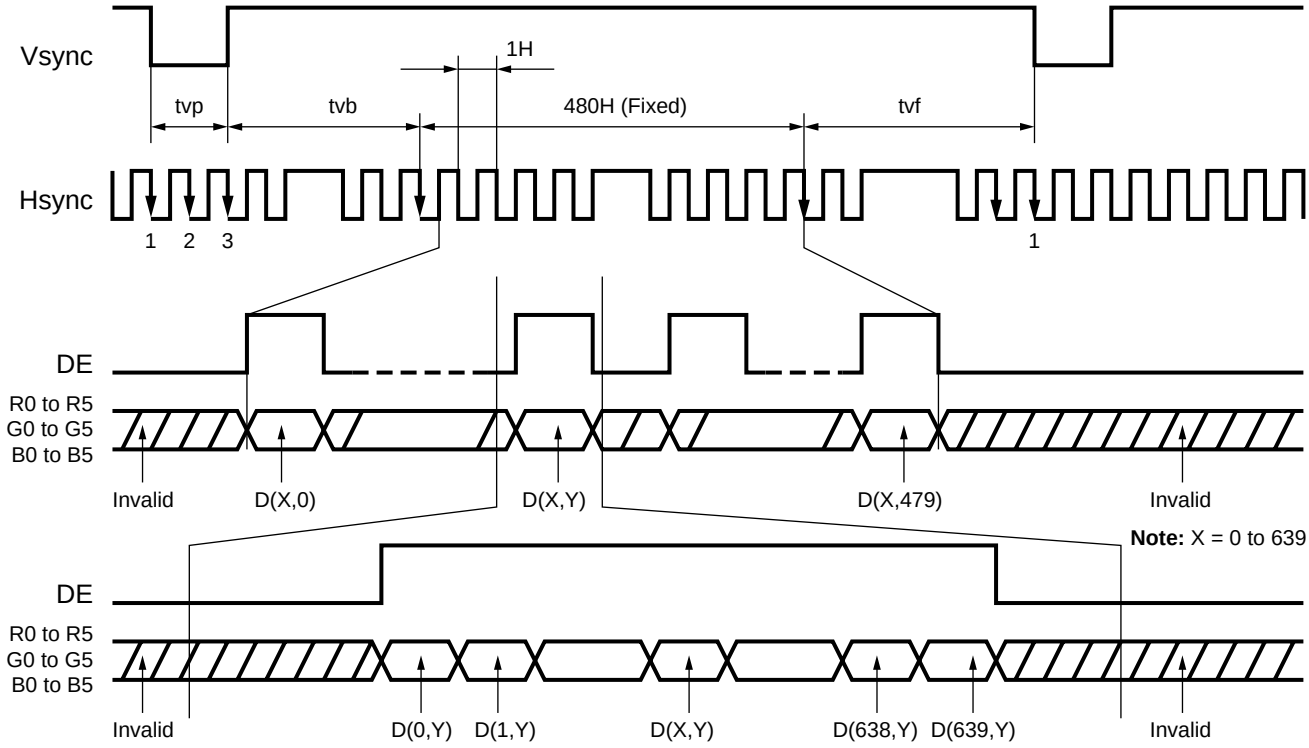
Figure 13-3: **Horizontal Data**



tv = 525 h_syncs (Vertical)
tvp = 2 h_syncs
Display period is 480 h_syncs

UG057_44_010804

Figure 13-4: **Vsync and h_syncs**



tvp = 2 h_syncs
tvb = 31 h_syncs
DE = 640 TFT Clocks
tvf = 12 h_syncs
Display period is 480 h_syncs

UG057_45_010804

Figure 13-5: **Vertical Data**

Memory Map

Video Memory

The video memory is stored in a 2 MB region of memory consisting of 1024 data words (1 word = 32 bits) per line by 512 lines per frame. Of this 1024 x 512 memory space only the first 640 columns and 480 rows are displayed on the screen.

For a given row (0 to 479) and column (0 to 639), the pixel color information is encoded as shown in Table 13-6.

Table 13-6: **Pixel Color Encoding**

Pixel Address	Bits	Description
TFT Base Address + (4096 * row) + (4 * column)	[31:24]	Undefined
	[23:18]	Red Pixel Data: 000000 = darkest → 111111 = brightest
	[17:16]	Undefined
	[15:10]	Green Pixel Data: 000000 = darkest→111111 = brightest
	[9:8]	Undefined
	[7:2]	Blue Pixel Data: 000000 = darkest→111111 = brightest
	[1:0]	Undefined

Control Registers (DCR Interface)

Table 13-7: **Control Registers (DCR Interface)**

Register Address	Bits	Read/Write	Description
DCR Base Address + 0	[31:0]	RW	Base Address of video memory. This is the address of a PLB accessible memory device that acts as the video memory. This address must be aligned on a 2 MB boundary (i.e. only the upper 11 bits are writable, the remaining address bits are always 0)
DCR Base Address + 1	[31:2]	-	Undefined
	[1]	RW	DPS control bit: 0 = Set DPS output bit to 0. The sets the display to use a normal scan direction. 1 = Set DPS output bit to 1. The sets the display to use a reverse scan direction (rotates screen 180 degrees).
	[0]	RW	TFT enable/disable bit: 0 = Disable TFT display. The causes a black screen to be displayed and it disables the generation of PLB read transactions. 1 = Enable TFT display. The causes the PLB TFT LCD Controller to operate normally.