Design Methodologies for Core-Based FPGA Designs

Jerry Case, Nupur Gupta, Jayant Mittal and David Ridgeway

Abstract

The adoption of design re-use has resulted in the availability of a variety of implementation options. Each option in turn offers a distinct design methodology that must be adhered to in order to successfully complete a custom single-chip design. This tutorial will focus on a system level design methodology for combining the predictability of a PCI interface core with the flexibility of a custom backend.

Specifically, this tutorial will explore the core-based design issues facing engineers and present a solution for addressing these issues during the design entry, design implementation and design verification stages of the product development cycle. This will be followed by an application example demonstrating the core-based design methodology used to integrate a PCI interface core with custom backend functions.

1 Application Specific Solution Options For Designers

A core is a pre-defined, pre-verified complex functional block that is integrated into the designer's logic. The rapid trend towards sub-micron technologies has brought forth the new concept of System-Level Integration (SLI). This new approach makes use of the core to save development time while focusing engineering time and energy on those parts of the design that add value and differentiation.

Core-based designs have several benefits:

- shorter design-cycle times
- reduced risk
- improved performance through higher levels of integration

The end result of SLI technology leads to shorter time-to-market, lower production costs, and improved system profit margins.

1.1 Types of Cores Available

One parameter for characterizing cores is the "hardness" of the block, or the degree to which the core has been optimized for a particular fabrication process. Cores can be classified in three categories: hard, firm and soft. Table 1 provides a summary of the tradeoffs between these cores.

Hard cores are optimized for a specific silicon technology and cannot be modified by the system designer. These cores have a pre-defined layout and floorplan that is included in the architecture of the design. They have the advantage of fixed timing and can be treated as library elements during the design cycle. The disadvantage of hard cores is that the designer can neither customize the functions nor adjust its timing to meet the requirements of the entire chip.

Firm cores are delivered as a mix of source code and technology-dependent netlist. In these cores, the source code is visible to the designer and specific parts of the core can be customized by the designer. However, a netlist is technology-specific and the user can not easily switch chip vendors or use a different technology with the same vendor.

HDL-based soft cores offer full technology independence and flexibility. The design can be readily modified or resynthesized to multiple technologies in order to switch vendors or target a new process. The disadvantage with soft cores is that critical timing is not guaranteed and the core must be synthesized, floorplanned, and placed-and-routed for each use.

PCI Local Bus interface cores are available in all three formats and provide a good example of the type of tradeoffs designers must make when selecting the core. The PCI core has a number of critical paths that must be controlled during synthesis to guarantee PCI compliance. At the same time, the system designer must be able to customize the core to specify memory and I/O space requirements, vendor and device specific information, as well as any configuration settings necessary to support Plug-N-Play operation.

TABLE 1. Tradeoffs of Hard, Firm, and Soft Cores

	Hard Cores	Firm Cores	Soft Cores
Hardness	pre-defined layout	mix of source code and technology-dependent netlist	delivered as behavioral source code
Modeling	modeled as a library element	mix of fixed and synthesizable blocks can share resources with other cores	synthesized with other logic
Flexibility	can not be modified by designer several hard cores on one chip can result in inefficient system implementation	technology dependent specific functions customizable	design can be modified technology independent
Predictability	timing guaranteed	critical path timing fixed	timing not guaranteed cannot be fully verified before released to user

1.2 System Level Integration Description and Requirements

Cores, complete with a set of common deliverables, are essential to design re-use and the SLI methodology. One key aspect is the ability to smoothly fit into a top-down design flow: concept, architecture, partitioning, logic design, and physical design/post layout verification.

One of the benefits of the SLI approach to high-volume design is the coupling of a core with user-defined circuitry on a single chip, as shown in Figure 1. This approach combines the benefits of a standard product (e.g. common feature set, fully-tested component, and well-defined interface) with the benefits of custom circuits (e.g. low production costs, small size, and low power consumption).

To guarantee that a core fits within the physical layout of a device, the core should allow for the use of floorplanning tools. The option to use floorplanning tools facilitates achieving an efficient die size while keeping the cores own logic grouped. For cores that have been previously laid out such as hard or firm cores, floorplanning and an efficient place-and-route tool will place logic in the most optimal manner possible around the core.

The Xilinx PCI LogiCORE module is an example of a firm core that has been optimized for the XC4000E FPGA architecture. To increase flexibility and reduce layout problems, the firm core is based on building blocks allowing the core to be broken down into functional blocks. In this case, a small portion of the core is parameterizable, such as the size of each base address register.

The I/O cells must be placed in predetermined locations on the die allowing efficient access to the PCI Local Bus. The performance requirements between the core and the I/O pads require the core to be placed directly next to the I/O cells. Because the primary bus routing resources run horizontally across the FPGA, the PCI Interface pins are placed primarily along the left side of the rectangular core structure.

A small number of pins, those that communicate with the user's application, appear on the right side of the core. Placing the core on the left edge of the device allows for maximum "connectability" to the user application and the remaining chip logic.

Floorplanning of chips with PCI cores is essential because of the wiring congestion created by these large functions, the interconnect compatibility issues imposed by the PCI specification, and the physical design constraints imposed by the target technology.

In order to realize the productivity gains and industry growth of sharing and integrating system-level cores, system designers are being asked to modify their traditional ASIC design methodologies to seamlessly integrate different types of cores.

2 Methodology for Core-Based Designs

The development of core technology and higher gate densities in FPGAs has increased the necessity for a design methodology that defines the integration of SLI cores and user application designs. This methodology must fulfill technical requirements and accommodate any design variations or formats. In addition, it should reduce the range of technical issues that are encountered during each stage of a design cycle.

A designer's primary focus must be on the development of a complete SLI solution that successfully complies with the objectives outlined for the project. However, much of a designer's time is spent addressing the technical issues that arise with the compatibility of the core technology and the user application design. The goal of this paper is to aid designers in this process and define a design methodology that inherently focuses on developing an SLI solution in an FPGA.

FPGAs are primarily used because of their ability to significantly reduce time to market. Should any variations in a user application design arise, it should be a simple process to achieve an SLI solution that avoids the burden of redesigning the entire chip. A design flow for integrating cores within FPGA's offers the balance between predictability and flexibility.

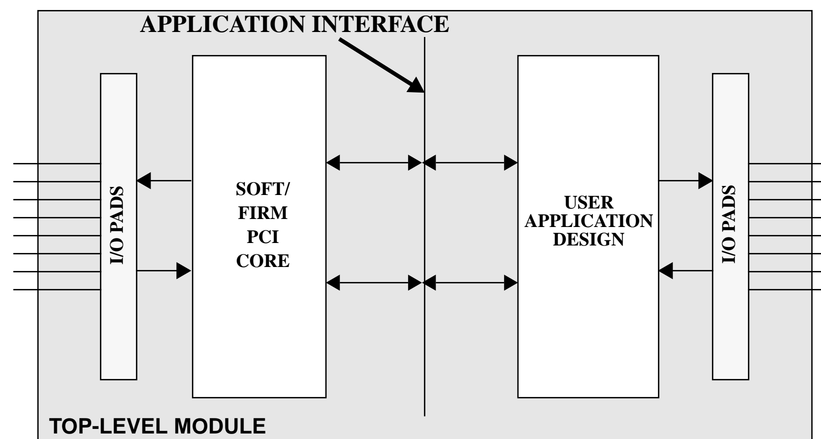

FIGURE 1. Physical Representation of Single Chip Environment

A design flow consists of three fundamental stages: design entry, design implementation, and design verification. This design flow is shown in Figure 2. The stages are composed of various design tasks. During design entry, the user application design is developed and synthesized. This produces a user design netlist. This netlist is translated along with the firm core netlist during the design implementation stage. This generates a design that contains the complete placement, routing and timing information. This information is then used during the design verification stage to perform static timing analysis and post route timing simulation.

Within these stages there are techniques employed to reduce the number of technical issues encountered in subsequent stages. The following sections will discuss each stage in greater detail. As a consequence, a design methodology that reduces time to market and increases flexibility and predictability will be presented.

2.1 Design Entry

Design entry consists of the design and synthesis of the top-level and user application design and integration with the firm core. In addition to this, functional simulation, which is part of design verification, is performed during this stage. It is covered in “Design Verification” on page 5. Specifically, this section focuses on:

- HDL representation - preliminary synthesis steps associated with the physical integration of the firm core and user application design
- facilitating design implementation - tasks to facilitate the design implementation
- facilitating design verification - a method for route budget analysis of the entire integrated design to derive an estimate of the timing constraints.

2.1.1 HDL Representation

Before synthesizing the user application design, it is necessary to create an HDL model of the top-level design that instantiates the firm core and the user application design. The firm core is treated as a black box and will not be a synthesizable element. For this reason, it is also necessary to create an HDL entity or module declaration of the firm core to serve as a black box representation when it is instantiated by the top-level HDL model. In addition, the designer must ensure that the EDA tools recognize this black box as a fixed element of the design. For example, this is commonly done with the “don’t touch” attribute in Synopsys synthesis tools.

Since firm cores are technology dependent, they may be delivered with guide files. Guide files ensure the core’s performance by locking the routing of critical paths. This requires the core’s logic to be consistent with the guide file. For this reason, the HDL representation of the top-level model should connect all module I/O or output only signals of the firm core to the user application design. All signals are physically mapped from the firm core to the user application design and are not trimmed by the FPGA translation tools. However, all signals do not have to be used internally within the user application design.

In addition, bus interface cores may impose further constraints on the design. For example, for the Xilinx PCI core, the user application design should derive clock and active high reset from the firm core. Otherwise the integrated FPGA netlist will have these coming directly from pads rather than IBUFs and BUFGs.

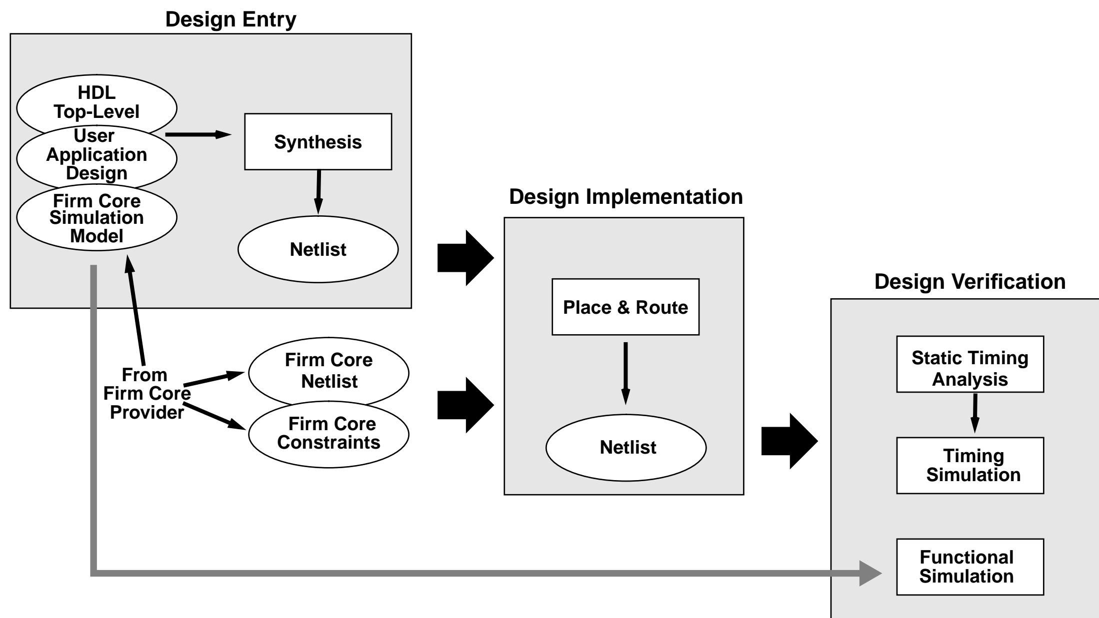

FIGURE 2. Core-Based Design Flow

2.1.2 Facilitating Design Implementation During Design Entry

The user application design must be synthesized separately with timing constraints to optimize on logic adjacent to the firm core. While fixing constraint delay values, only part of the cumulative path delay should be assigned. Specifically, these should be paths originating at the output of the firm core and terminating in the user application or FPGA output pins. The number of synthesis timing constraints, e.g. set_max_delay, set_output_delay, and set_input_delay in Synopsys, are strictly dependent on the complexity of the user application. For example, PCI setup time may not be an issue for a simple interface since all outputs from the firm core to the user application are latched. However, for a complicated bridge i.e. PCI-to-PCI, this will be a consideration. Once the top-level model and user application design has been synthesized, it is necessary to replace the synthesized generated netlist for the firm core with the actual netlist supplied by the firm core provider.

2.1.3 Facilitating Design Verification During Design Entry

As the complexity of the user application design increases, the amount of available resources on an FPGA will decrease. It is strongly recommended to do some preliminary route budget analysis on the application design. Often, firm cores are provided with constraint and guide files thereby indicating the critical timing of the design. Adding the additional complex logic may cause the overall system to fail to meet the required performance.

There are many techniques that can be used to ensure that the overall design meets timing. One commonly-used practice is to pipeline the user application design. Although this may increase the latency of the design, the overall throughput will increase and it is less likely that critical paths will fail. Floorplanning can also improve performance. Placing critical logic into the same or adjacent cells to optimize the data flow limits the amount of routing resources used and minimizes routing delays. If the design permits, these are effective methods to reduce the number of iterations to successfully route a design and meet timing.

Another technique is the use of constraints on the application design. Similar to the constraints provided for the firm core, the user application constraints can force a particular placement of logical blocks and specified interconnects to adhere to fixed timing specifications. Logic constraints can be specified in the two different stages of the design flow: synthesis and translation. Constraints specified during synthesis can give an estimation of the timing delay in the internal logic of the user application design and optimize logic depth. Constraints during translation can provide timing specifications and placement information for the user application design. Timing constraints that were used during the synthesis of the design can be forwarded in the firm core constraints file. This will ensure that internal user application logic adheres to these timing specifications during the translation stage.

The recommended method of determining the route budget for a design is to do a preliminary translation of the design without any route budget constraints but using core provided constraints. Static timing analysis will indicate the margin of delays allowed to the user application design. Constraints can then be placed during synthesis of the application design. This will give some indication as to the necessity to pipeline the design or further constrain it during the design implementation stage.

The syntax for placing constraints on a user application design during the design entry stage varies depending on the EDA tool vendor. For example, in Synopsys, timing constraints for pad-to pad can be set with the use of the set_max_delay and set_output_delay commands.

2.2 Design Implementation

The second fundamental stage is the design implementation stage. During this stage, translation tasks are executed. Many issues that would have arisen from the integration of the firm core and user application design are no longer visible. The translation of the design should be a simple process that is mainly administered by the translation tools. The following is a discussion on:

- facilitating design verification - further steps a designer can take to assist design verification
- translating the design

2.2.1 Facilitating Design Verification During Design Implementation

As discussed earlier, timing and placement constraints can be placed in the provided constraints file. A designer should time constrain critical paths that may have high fan out or directly effect the critical paths of the core. In addition, paths that have a very stringent time specification should also be constrained.

With the use of the constraints file, timing constraints can be placed on those paths that are critical to achieve performance. This facilitates design verification by allowing the designer to query a set of specific nets without specifying or searching for the complete path. This is beneficial with a design where only certain paths have stringent timing constraints, and the design has difficulty meeting timing.

2.2.2 Translating the Design

Once the firm core and application design netlists have been generated, it is a simple process to translate the complete design. For example, with the Xilinx tools, the flattened netlist, will be processed by the Xilinx Netlist Format Preparation (XNFPrep) tool. The designer must remember to specify any constraints file that may have accompanied the core. This will create a XTF netlist. XNFPrep is used to perform Design-Rule Check and remove any redundant and unused logic from the design. It is also used to prepare the delay information for PPR path analysis. Further information can be found in the Xilinx XACT software documentation.

After running XNFPrep, the complete design is processed through the Partition, Place-and-Route tools (PPR). During this step, any constraint and guide files for the design must be specified. PPR is used to map the design into logic blocks, find the optimal block placement and route the interconnect between blocks. Further information can be found in the Xilinx XACT software documentation. PPR will produce a physical netlist, an LCA file. Static timing analysis can now be performed.

2.3 Design Verification

The final stage in a design flow is design verification. This stage consists of two main tasks: verification and simulation. In verification, static timing analysis determines whether the design meets the required performance. Simulation verifies system timing and functionality. Design verification should be thoroughly done, and extensive research can show the designer where in the design flow to improve performance.

2.3.1 Static Timing Analysis

Static timing analysis is performed to survey timing critical paths that are constrained by design performance parameters. For example, in Xilinx tools, static timing analysis of an FPGA design can be done with the use of TIMESPEC timing specifications and the XACT XDelay tool. TIMESPECs are determined earlier on in the design cycle and are placed in the constraints file. Translation of the design will place the necessary path delays for the design in the LCA file. The XACT XDelay tool surveys this information from the LCA file to give an accurate static timing analysis of the path delays for any time specifications that were placed in the constraints file. Any additional interconnects that may not have been specified by time constraints can also be surveyed but may not necessarily meet the time specifications determined by the designer. It is important to assign all critical paths to time specifications in order to allow PPR to determine the most efficient and optimal placement of the design. Further information on the XACT XDelay tool can be found in the XACT software documentation.

2.3.2 Functional Simulation

Simulation is performed at two places in the core-based design methodology, during design entry to verify functionality with unit delays and after place-and-route to verify functionality and timing with back-annotated timing delays. During functional simulation, a VHDL or Verilog model of the core is used. This core has the same I/O interface as the core netlist used for place-and-route. This enables functional simulation and place-and-route without modifying the user application design.

3 Application Example

The following design example will demonstrate how to integrate an FPGA firm core with an example user back-end design. First, the individual components of the design are described. Second, the complete design flow is detailed. The components that comprise this design example are the LogiCORE PCI Master Interface core module, the user back-end example design, called Ping, and the top-level module which connects the firm core and the user design together into a single FPGA.

3.1 LogiCORE PCI

The LogiCORE PCI Master Interface from Xilinx is used as an application example to demonstrate the core-based design methodology. LogiCORE PCI is included in Xilinx complete PCI solutions for designing a customized single-chip PCI system. LogiCORE PCI is a firm core consisting of a fixed size and user configurable option.

Completing a single-chip PCI design is a significant challenge in any IC technology; FPGAs or ASICs. PCI is a significantly complex interface with many critical timing constraints, such as 7 ns setup time and 30 ns clock cycle times. To meet the rigorous PCI specification, an FPGA implementation must be carefully optimized for the targeted technology. Furthermore, the PCI design must be extensively verified in order to claim full PCI compliance of the end-system. Of course, this is especially important when designing a generic add-in board used in a wide variety of computer systems.

The LogiCORE PCI Master/Target core is a 32 bit, 33 MHz interface optimized for Xilinx XC4000E series FPGAs. The PCI interface can be integrated with an additional 5K to 30K gates of custom logic into one flexible FPGA and then automatically converted to a cost-effective HardWire for volume production. To ensure full PCI 2.1 compliance, Xilinx has chosen to pre-implement and lock down all the critical paths of the design. The pin-out and the partition and placement of the Configurable Logic Blocks (CLBs) in the FPGA are controlled by mapping constraints, pin-locking and Relatively-Placed Macros (RPMs). The routing of the most critical signals is controlled by guide files and timing specifications.

The result is that the functionality and performance of the core is predictable, and consequently, the core can be fully verified before it is released and used by a designer. Optimization for a specific device architecture has been completed by Xilinx, hence the users can focus on system-level design issues.

3.2 PCI Core Functional Description

The Core is partitioned into five major blocks and the user application, as shown in Figure 3.

3.2.1 PCI I/O Interface Block

The I/O interface block handles the physical connection to the PCI bus including all signaling, input and output synchronization, output three-state controls, and all request-grant handshaking for bus mastering.

3.2.2 Parity Generator/Checker

This block generates/checks even parity across the AD bus, the CBE lines, and the PAR signal. Data parity errors are reported on the PERR- signal and address parity errors on the SERR- signal.

3.2.3 Target State Machine

This block manages control over the PCI interface for Target functions. The states implemented are a subset of equations defined in “Appendix B” of the PCI Local Bus Specification. The controller is a high-performance state machine using state-per-bit (one-hot) encoding for maximum speed. State-per-bit encoding has narrower and shallower next-state logic functions that closely match the Xilinx FPGA architecture.

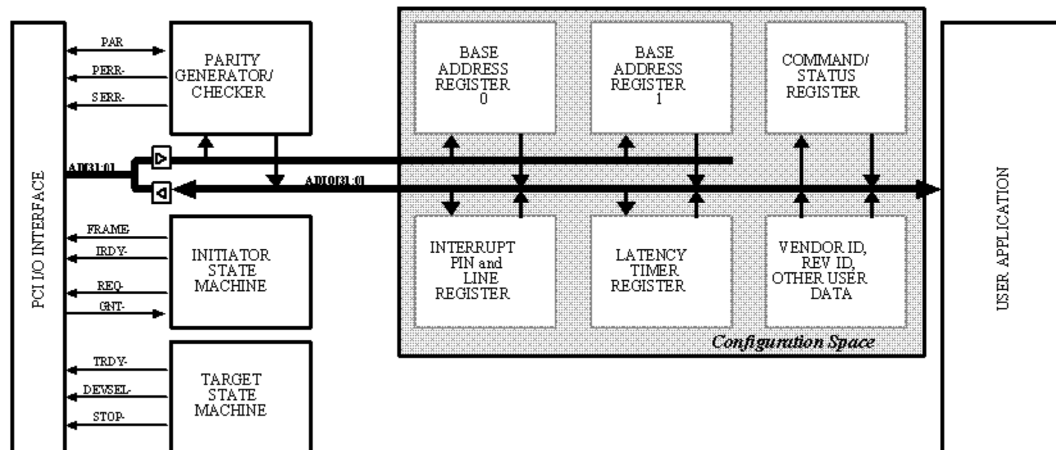

FIGURE 3. LogiCORE PCI Block Diagram

3.2.4 Initiator State Machine

This block manages control over the PCI interface for Initiator functions. The states implemented are a subset of equations defined in “Appendix B” of the PCI Local Bus Specification. The Initiator State Machine also uses state-per-bit encoding for maximum performance.

3.2.5 PCI Configuration Space

This block provides the first 64 bytes of Type 0, version 2.1, Configuration Space Header (CSH) to support software-driven “Plug-and-Play” initialization and configuration. Using a combination of Configurable Logic Block (CLB) flip-flops for the read/write registers and CLB look-up tables for the read-only registers results in optimized packing density and layout.

3.3 Available device and package options

Table 2 lists the available device and package options and remaining CLBs for XC4000E.

3.4 PCI Core Generator

For configuration of the LogiCORE PCI interface, a CORE Generator tool with a Graphical User Interface (GUI) is provided on the Xilinx home page, see Figure 4. To use the CORE Generator, a designer logs on to the web with a Java enabled browser such as Netscape Navigator 3.0 or greater, Sun Hot Java or Microsoft Internet Explorer 3.0 or greater. The PCI parameters are entered in the GUI by clicking on selections in a table. After parameter entry, a unique design file, including an XNF netlist with pre-defined placement constraint files to guide PPR, and a simulation model are downloaded.

The CORE generator is executable on the Xilinx home page so that a designer will always have instant access to the latest core enhancements and added features. Additionally, the tool is platform independent. A CD-ROM version will also be provided for users that prefer a local copy of the tool.

As an alternative to the CORE Generator, a designer can choose to configure the core using provided Viewlogic schematics. The Viewlogic schematics are designed to plug-in to the same HDL wrapper that the Core Generator uses, so no design modifications are necessary.

The overall flow used to configure the Xilinx PCI module is shown in Figure 5.

The Xilinx PCI module has two pre-defined areas which can be configured: PCI features to be enabled, and the PCI Configuration Space Header.

The top of the applet window shows three radio buttons, titled Master, Burst Mode, and Enable Interrupts.

The “Master” button enables the device to act as a PCI Bus Master. The “Master” button de-selected disables Bus Mastering. (The device would support Target transactions only.). The “Burst Mode” button enables the device to generate and receive burst mode transactions.

TABLE 2. Devices and Package Options

LogiCORE™ Facts

PCI Master & Slave Interfaces V1.1.0

Core Specifics	
	XC4000E
CLBs Used	152 - 268
IOBs Used (Master/ Slave)	53/ 51
System Clock f_{max}	≤ 33MHz
Device Features Used	Bi-directional data buses SelectRAM™ (optional user FIFO) Boundary scan (optional)

Supported Devices/Resources Remaining		
	I/O	**CLBs**
XC4013E PQ160 (Slave only)	76	308 - 424
XC4013E PQ208/ HQ208 (Master/ Slave)	107/ 109	308 - 424
XC4013E HQ240 (Slave only)	141	308 - 424
XC4020E HQ208 (Slave only)	107	516 - 632
XC4020E HQ240 (Slave only)	141	516 - 632

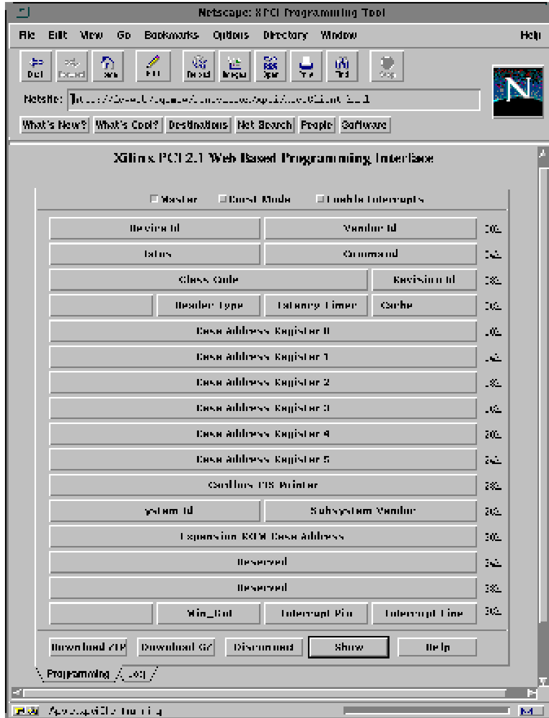

FIGURE 4. Web-based CORE Generator for PCI

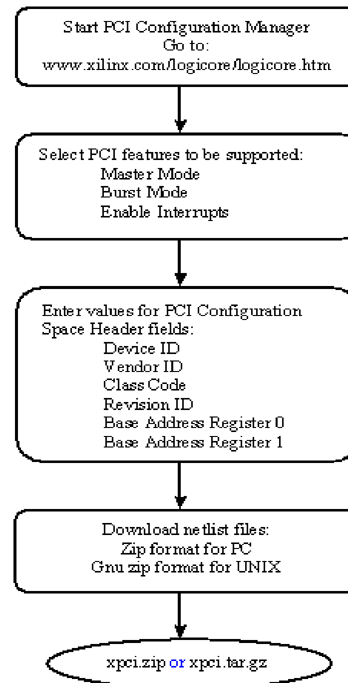

FIGURE 5. Core Generator Usage

The “Enable Interrupts” button enables the interrupt line and interrupt Ping registers in the Configuration Space Header.

The main window of the Xilinx PCI Core Generator is made to look identical to “Figure 6-1: Type 00h Configuration Space Header” in the PCI Local Bus Specification, Revision 2.1. Six of the fields in the Header must be configured with user information. These fields are listed below. To configure any of these fields, press the field with the left mouse button. (The field is actually a button.)

The Device ID field is used to identify a particular PCI device. The individual vendor determines the value.

The Vendor ID is used to identify the device manufacturer. Each vendor is assigned a unique Vendor ID by the PCI-SIG.

The Class Code is used to determine the device's general function. (See page 189 of the PCI Specification for more information.) The Xilinx PCI Core Generator displays the classes for this register, and allows the user to select from the list.

The Revision ID is used to identify a particular version of a PCI device. The individual vendor determines the value.

The Base Address Register is used to determine how to allocate memory or I/O space to the particular PCI device. The Xilinx PCI Core Generator displays the various Memory or I/O space options, and allows the user to select from the list of options.

The Base Address Register 0 is the first Base Address Register, located at offset 10h. The Base Address Register 1 is the second Base Address Register, located at offset 14h.The fields which are shadowed-out are not user-selectable. Base Address Registers 2 through 5 are not supported.

The Xilinx PCI Core Generator configures the XNF netlist according to user specifications, and shows two hyperlinks. One hyperlink downloads the files in zip format for PC, the other hyperlink downloads the files in Gnu zip format for UNIX. The contents of the zip and Gnu zip files are identical.

Table 3 lists the files are included in the xpci.zip and xpci.tar.gz files that the PCI Core Generator creates.

3.5 User Application Design

The user back-end design example is called Ping. Ping provides an easy-to-understand user application example interface that demonstrates many of the principles and techniques required to successfully use the PCI LogiCORE macro in a real-world application. Ping takes its name from the TCP/IP utility named ping which allows network users to test that a particular machine is ‘alive’ and communicating on the network. As such Ping is designed to provide the same type of functionality for ‘bringing up’ new PCI designs for the first time on a new PCB, as well as providing a design tutorial based on a simple DMA engine which illustrates the following key control elements in a user’s application interface to the Macro:

- Initiator read and write operations - burst transfers
- Initiator response to various Target termination conditions (Retry, Disconnect, Target Abort)

TABLE 3. PCI Core Directory Structure

Directory	File	Comments
xpci/		Top directory containing all subdirectories and files described below
	config.txt	Contains the user-selected configuration settings used to generate the PCI module
cst_file/	i13p208h.cst	Constraint file for 4013EPQ208 Master design
	t13p208h.cst	Constraint file for 4013EPQ208 Slave design
guide/	i13p208h.lca	Guide file for 4013EPQ208 Master design
	t13p208h.lca	Guide file for 4013EPQ208 Slave design
verilog/	pci_top.v	Example top-level HDL design module
	pcim_lc.v	Black-box PCI Interface module (Master)
	pcis_lc.v	Black-box PCI Interface module (Slave)
	userapp.v	Template for back-end module
	synopsys.dc	Synopsys synthesis script
vhdl/	pci_top.vhd	Example top-level HDL design entity
	top_cfg.vhd	Example top-level HDL design configuration (for simulation)
	pcim_lc.vhd	Black-box PCI Interface (Master design)
	pcis_lc.vhd	Black-box PCI Interface (Slave design)
	userapp.vhd	Template for back-end entity
	synopsys.dc	Synopsys synthesis script
wrapper/	pcim_lc.sxnf	HDL “wrapper” which is used instead of the pci_lc_i schematic symbol (Master Design)
	pcis_lc.sxnf	HDL “wrapper” which is used instead of the pci_lc_t schematic symbol (Slave Design)
xnf/	*.xnf	Hierarchical XNF netlists for the PCI module; the pcim_lc.sxnf file calls these files for lower level modules

- Initiator responds to mid-burst Target Termination with another request
- Target read and write operations (single transfer and burst)
- Target generates various Target termination conditions (Retry, Disconnect, Target Abort)
- How to interface to memory space
- How to interface multiple devices to a single BAR, as opposed to a BAR per device
- How to interface to I/O space
- Basic PCI data flow control
- Basic user data flow control

The Ping design accepts data as a Target PCI device, then uses the data to perform Initiator transactions over the PCI bus. Ping is implemented in VHDL and intended to function as a self-contained PCI Master and Slave. No additional logic is needed on the PCI add-in card beyond the XC4013E device that contains the PCI core module and the Ping design.

3.6 User Application Functional Description

The user application design, PING, is partitioned into six major blocks. The block diagram for PING is shown in Figure 6.

3.6.1 Ping Registers

Ping consists of two 32-bit read/write (R/W) registers DATA and ADDRESS multiplexed to a common user ADIO bus and interfaced to the user application part of the PCI core. These provide the data during address/data phases of initiator cycles. They are both memory-mapped to base address register 0 with offsets 0 and 4 respectively. A 16-bit WRITE ONLY register namely CONTROL is I/O mapped on BAR1. This is shown in Figure 7. Table 4 has also been provided as a summary of the register description.

3.6.2 Command Decoder

This block decodes the PCI_CMD bus coming from the PCI core to perform memory reads/writes into the DATA and ADDRESS registers and I/O writes into the CONTROL register. CMD_DEC bus from this block controls the register logic.

3.6.3 Register Select Logic

This block generates the chip select (CS) for Ping registers upon receiving BASE_HIT on either of the PCI core base registers. The registers are disabled upon END of address phase in either of target/initiator cycles. This is indicated by signals S_ADDR_VLD and M_ADDR_VLD coming from the PCI core.

3.6.4 Ping Address Generator

This block has a 4-bit counter that increments with each strobe of DATA_VLD and SRC_EN coming from the PCI core during BURST WRITE and READ cycles respectively. The counter output is then added to the address during ADDRESS PHASE (LAT_ADDR from PCI core) of the PCI bus transaction to generate subsequent addresses for burst operation.

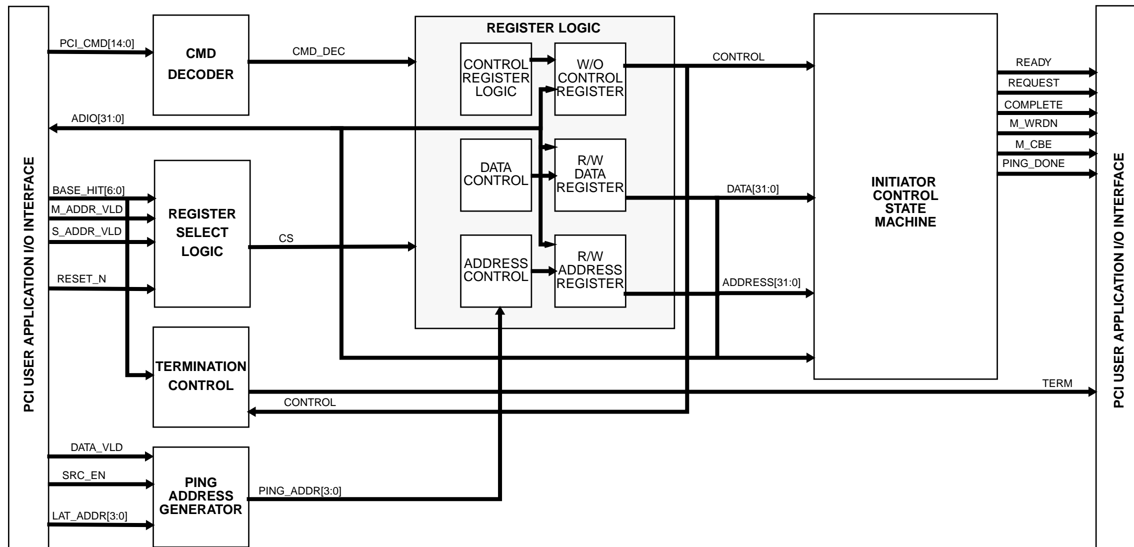

FIGURE 6. PING Block Diagram

3.6.5 Termination Control

This decodes the relevant bits of the Ping Control register to set the target termination conditions for RETRY, ABORT, DISCONNECT and NORMAL operations. It also outputs READY and TERM inputs to the PCI core to assert PCI STOP. This has been summarized in Table 5.

3.6.6 Initiator Control State Machine

This is the heart of user application design. It generates all the inputs to the PCI core for requesting initiator cycle (REQUEST), ending transaction (COMPLETE), indicating read/write (M_WRDN) and generating command/byte enable (M_CBE). These assist the PCI core to achieve master functionality. Once the CONTROL register is set for one of the two operational modes, the state machine triggers off by external stimulus either immediately upon writing into ADDRESS register (auto_ping mode) or waits until PING_REQUEST (ping_wait_request mode) is asserted.

3.7 Design Flow

The PCI module is designed to be instantiated in the user's Verilog/VHDL design. A black-box description of the PCI module is used during synthesis. An XNF netlist of the PCI module is used during place-and-route to incorporate the actual PCI design into the user's design. The XNF netlist is used to maintain the performance and predictability of the design. The overall HDL design flow is shown in Figure 8.

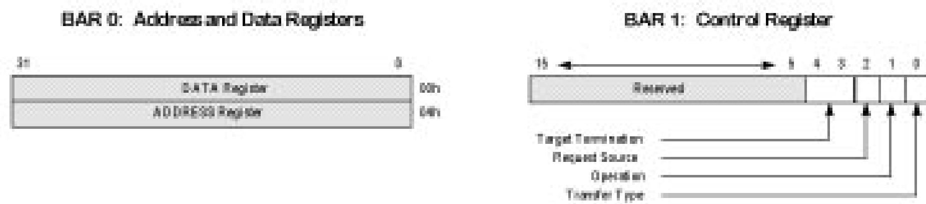

FIGURE 7. Address, Data and Control Registers

TABLE 4. Register Description

Register	Purpose	Size	Operation	Mapping
DATA	Holds the data for a single Initiator Write transaction or receives the data for a single Initiator Read transaction.	32-bits	R/W	Memory (BAR0) Offset = 0
ADDRESS	Holds the source or destination address for the Initiator transaction.	32-bits	R/W	Memory (BAR0) Offset = 4
CONTROL	Controls the type of operation.	16-bits	WO	I/O (BAR1) Offset = 0

TABLE 5. Control Register Commands

Command	Options	Position
Transfer Type	0 = Initiator transfers (default) 1 = Target terminations	0
Operation	0 = memory read 1 = memory write	1
Request Source	0 = immediately after Write to ADDRESS register 1 = wait until user REQUEST pin asserted High	2
Target Termination	00 = normal (default). Target access to BAR0 terminates normally. 01 = disconnect. Target access to BAR0 causes Target Disconnect. 10 = retry. Target access to BAR0 causes Target Retry. 11 = target abort. Target access to BAR0 causes Target Abort.	4:3
Reserved	000	15:5

.

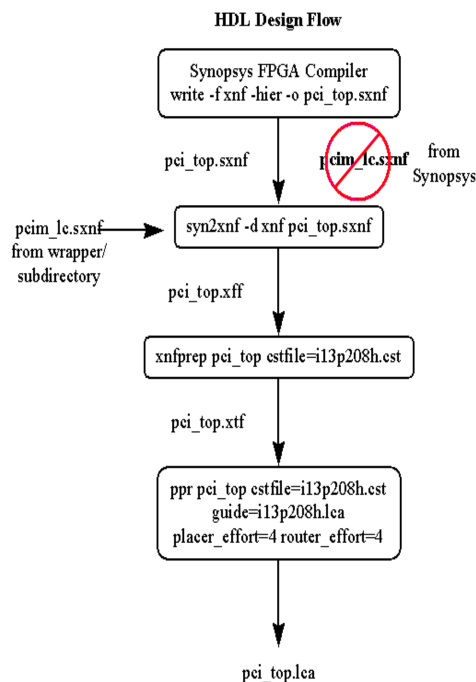

FIGURE 8. Implementation Flow Diagram

3.7.1 Design Entry Example

The top-level module that connects the PCI Core to Ping is included in the files generated by the Core Generator. The Core Generator includes VHDL and Verilog templates for the top-level module, including instantiation of the PCI Core and a back-end “userapp” design. The top-level templates include features needed for the place-and-route guide file to correctly guide the placement and routing of the PCI Core. These features are:

1. The PCI Core instance name must be “pci_core”
2. All PCI I/O pins on the PCI Core must be connected to top-level I/O ports with the exact names used in the template files.
3. All Core inputs and outputs which are not PCI I/O must be connected to the back-end “userapp” symbol, with the names used in the template files. This guarantees that net names are identical with those in the guide file.

3.7.2 Design Implementation Example

Verilog and VHDL templates are included in the verilog/ and vhdl/ subdirectories. These templates include a top-level design, called “pci_top”, which has two instantiated modules inside, called “pcim_lc” and “userapp.” The PCI module is contained in the module called “pcim_lc.” The user's design can be substituted for the “userapp” module. Verilog and VHDL templates for the “userapp” module have also been included. A sample Synopsys synthesis script is included in both the verilog/ and vhdl/ subdirectories, and is called synopsys.dc. In the specific case of Ping, the ping.vhd file is substituted for the “userapp” module in the top-level VHDL file, and specific constraints are added to the synopsys.dc file.

A black-box description of the pcim_lc PCI module is included for both VHDL and Verilog. The VHDL file is called pcim_lc.vhd. The Verilog file is called pcim_lc.v. This black-box model is read into the synthesis tool, and a “don’t_touch” attribute is applied to it, prior to synthesizing the top-level design. In this way, the synthesis tool has a complete model for the design. With the black-box model, all components are resolved, even though the synthesis tool does not know what is “inside” of the PCI Core. Without the black-box model, the synthesis tool reports errors indicating unresolved components in the design.

This is a summary of the design flow (the current working directory is xpci):

1. Instantiate pcim_lc in top-level HDL design.
2. Read pcim_lc black-box Verilog or VHDL file into Synopsys. For Ping, the pcim_lc.vhd VHDL file is read.
3. Read top-level and back-end Verilog or VHDL files into Synopsys. For Ping, the pci_top.vhd VHDL file is read, along with ping.vhd.
4. Add “don’t_touch” attribute to pcim_lc design.
5. Use “set_port_is_pad” on user I/O pads. Do not use “set_port_is_pad” on any of the PCI Interface I/O ports. (IBUFs and OBUFs are already included in the PCI design for the PCI I/O pads.) Insert pads with the “insert_pads” command. For Ping, the I/O ports

PING_REQUEST and PING_DONE are given the set_port_is_pad command. Pads are only inserted on these two back-end I/O ports.

6. Compile top-level design with constraints.
7. For FPGA Compiler, run “replace_fpga”, to replace CLBs with gate primitives.
8. Save design as an XNF file. Make sure the “Save Hierarchy” option is used.
9. Synopsys will have written out a file called “pcim_lc.sxnf”. Copy the correct file “pcim_lc.sxnf” from the wrapper/ sub-directory to the current working directory. This file replaces the file that Synopsys created.
10. Run the command:

    syn2xnf -d xnf pci_top.sxnf

11. Run xnfprep, using i13p208h.cst as the constraint file.

    xnfprep pci_top cstfile=i13p208h.cst

12. Run PPR, using i13p208h.cst as the constraint file and i13p208h.lca as the guide file.

    ppr pci_top cstfile=i13p208h.cst guide=i13p208h.lca placer_effort=4 router_effort=4

13. The PCI design is now placed-and-routed.

3.7.3 Design Verification Example

Static timing analysis is used to determine the performance of the overall PCI design. The constraint file downloaded from the Core Generator includes timing specifications that control the static timing analysis program, XDelay. XDelay evaluates the design against these timing constraints. If XDelay verifies that the design meets all of these timing constraints, then the design meets the 33 MHz PCI performance requirements.

The Ping design is tested with a VHDL testbench that includes a simple PCI arbiter and a simple PCI Target device (see Figure 9). The testbench, as initiator, writes base address registers BAR0 and BAR1 and also sets the core as master during configuration cycles. It writes into the data and address registers of Ping in addition to setting control register bits for read/write and operation mode in different target cycles. This in turn, triggers Ping state machine to provide these values to PCI bus during address and data phases of initiator cycles. The testbench also sets target termination modes in the PCI target, upon which the core responds. The core-based approach eases the system simulation of the overall design because the PCI section is already extensively tested and verified. To facilitate simulation, a placed and routed vendor specific netlist is converted into a structural VHDL timing model. This is compiled and analyzed together with other system behavioral VHDL models and the testbench. The waveforms are traced and debugged in the VHDL simulator. This is summarized below:

1. Add the timing data and overwrite PCI design LCA file.

    xdelay -d -w pci_top.lca

2. Process the LCA file to create post routed XNF.

    lca2xnf -g pci_top.lca pci_top.xnf

3. Run the model_io Perl script to correct known conservative modeling values for setup/hold and clock-to-output timing of I/O flip flops.

    model_io pci_top.xnf pci_top.xnf

4. Generate the required design structural VHDL, SDF files using xnf2vss. However, just create the architecture but no entity to retain busses.

    xnf2vss pci_top.xnf

5. Compile and analyze the support VHDL models, testbench and design.

    vhdlan pci_top.vhd

    vhdlan dumb_target.vhd

    vhdlan ping_tb.vhd

6. Simulate and include the design traces.

    vhdlsim -t ns -i ping.include cfg_ping_tb

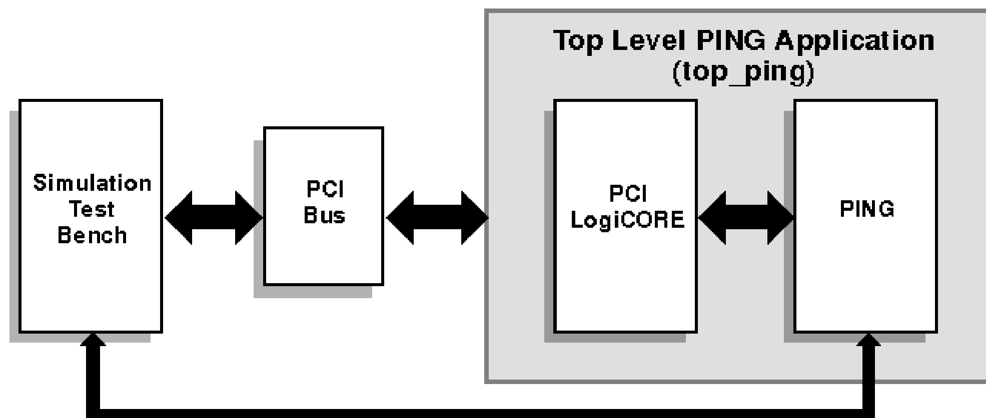

FIGURE 9. System Block Diagram

4 Conclusion

4.1 Strengths of Core-Based Methodologies

The described core-based design methodology enhances the key benefits of an FPGA; low design risk and time to market. By using an EDA tool independent CORE Generator for generating a customized, pre-defined and fully-verified core, design time can be cut by 6 months or more. The user's engineering time and effort can be focused on system-level considerations in favor of hand-tuning, implementing and verifying the PCI core. Because the CORE Generator is executable on the web, it is platform independent and a designer will always have access to the latest core enhancements and new features.

The FPGA-based PCI solution allows a designer to integrate a PCI interface with added-value custom logic into a single chip. The solution can automatically be converted to a HardWire devices for lower cost in volume production.

Acknowledgments

Steve Knapp, Gary Lawman & P. Rissmann, for their original work in developing PING. Gary Lawman, Joe Linoff and Jerry Case for their work in developing the PCI Core Generator tool.

Patents

The PCI Core Generator^{TM} tool is a Xilinx., product and is subject of multiple worldwide patent applications.