Embedded Particle Tracking Silicon Microscope: An Independent Data Acquisition System for Silicon Detector Characterization

by

© Kunal A. Arya

A thesis submitted to the

Baskin School of Engineering

in partial fulfilment of the

requirements for the degree of

Bachelor of Science

June 2007

Abstract

We present an upgraded read-out and data acquisition system for silicon detector characterization using off-the-shelf commodity hardware. This hardware centers around an embedded reconfigurable FPGA, which allows us to create custom digital logic for data conditioning. This data is sent over widely available and commoditized cable standards (such as ethernet and USB) via the on-board processor. This setup further obviates the need for data acquisition hardware (such as the current offerings of National Instruments).

Acknowledgements

This thesis and project would not have been possible without the consistent support, optimism, and allocation of funds of Dr. H. F.-W. Sadrozinski. I'm indebted to him for the opportunity and advice he's lent me over the past two years.

I am additionally indebted to Dr. Gavin Nesom, who volunteered his time in personally helping me get through the multitudes of obstacles that were inherent to the project. The problem-partitioning and -solving skills I've gained from him will undoubtably carry me through my future endeavors.

Dr. Tracy Larrabee has been incredibly generous with her time and patience, enabling me to go against the grain by sponsoring, advising, and revising this thesis.

Finally, I am and always will be grateful for my family. Their continual support, love, and concern have made my college experience possible.

Contents

List of Figures

Chapter 1

Introduction

This paper details the upgrade of an existing prototype data acquisition system from its current setup (described in Section 1.2) to a reconfigurable embedded system. My task was to design a custom peripheral to the embedded CPU that would perform the requisite data conditioning, to design the system as a whole such that the components (CPU, memory, ethernet, and custom peripheral) worked in harmony, to write the software that ran on the embedded CPU, and to write the software on the host PC that communicated with the embedded system. Data comes into the system in the form of a group of serialized wires, originating from an ASIC designed at the Santa Cruz Institute for Particle Physics (SCIPP), and leaves the embedded system over a standard crossover ethernet cable into a host PC.

The custom peripheral hardware was written in the Verilog Hardware Descriptive Language (HDL), and was based off of an existing design from Brian Keeney [1]. The systems upgrade is only on the digital back-end side; thus, the front-end ASIC designed by Edwin Spencer, and the front-end printed circuit board designed by

Forest Martinez-McKinney, remain the same. I had to create all of the cabling for the new connectors introduced by the embedded system.

The software on the embedded PowerPC CPU was written in C, while the software on the host PC was written in C++ using a cross-platform network library and GUI toolkit (SDLnet and wxWidgets, respectively).

1.1 Background and Overview

The Embedded Particle Tracking Silicon Microscope (EPTSM) project is a data acquisition system used for characterizing silicon detectors. The original system — Particle Tracking Silicon Microscope (PTSM) — was created for small-scale medical applications, and was the subject of B. Keeney's thesis [2].

The project was unique at the time in that it leveraged field-programmable grid arrays (FPGAs) along with existing National Instruments data acquisition (DAQ) hardware to make a relatively low-cost custom DAQ system. The original PTSM setup was to be used for radiobiological research, to determine the effects of induced mutations on simple organisms such as *C. Elegans* [4].

As the project evolved, Dr. H.F.-W. Sadrozinski proposed that the same system could be used to measure the electrical characteristics of silicon detectors for ATLAS RD50 research, whose main goal was to determine the effects of radiation on the efficiency of said detectors.

Silicon Detectors

Silicon detectors are one of the most attractive particle detection technologies currently used in high-energy particle physics experiments. They serve as the first and most precise detection layer in the concentric rings that comprise detector systems in particle colliders, allowing researchers to track the break-up, trajectories, and distribution of energy directly after two particles (such as a positron and electron) collide[3].

The detectors originate from both commercial vendors and universities from all over the world who have the resources to manufacture these detectors. Many of these detectors were designed for other experiments or for more general-purpose applications. They have not been tested for an application such as the Large Hadron Collider (LHC), which is the home of the ATLAS detector, where they will be exposed to an incredible amount of radiation. Thus, SCIPP researchers look at various candidates for the upgrade, and determine which ones are viable.

The primary goal in characterizing detectors is to learn as much as we can about the detectors, including interstrip capacitance, resistance, leakage currents, noise, and efficiency. More specifically, RD50 research is looking into the effects of radiation on the silicon material and its ability to read out particle hits. By constructing a deterministic framework for extracting these electrical details, researchers can make appropriate changes to purify the data in the final experiment.

1.2 Comparing Systems

In order to clarify the motivations behind this project, I will compare the current setup against the embedded system. There are a few common components between

the two systems, and as such, those parts are not discussed in this section. For now, it will suffice to understand that the front-end board and PMFE chip provide digital signals at a fixed data rate. Further details on the PMFE are in section 2.2.2, and details on the front-end board are in section 2.2.3.

The Current Setup

The current PTSM prototype uses an older Xilinx Virtex-II FPGA development board to read out the front-end chip (which is detailed in section 2.2.2). To get this information to the host PC, a National Instruments PCI DAQ solution collects the conditioned data from the FPGA at a maximum rate of 160 Mbps.

The FPGA generates signals conforming to the Low-Voltage Differential Signal (LVDS) standard (described in section 2.2.5) in order to facilitate longer cable lengths and less noise, but the commercial DAQ card only accepts standard CMOS voltages. This necessitates a signal translator board. This board was developed in-house, and has been one of a handful of points of failure as detailed below.

The cable interface between the front-end board and the FPGA board is another such point of failure. The FPGA board does not provide any structured input/output pin layout that works suitably with LVDS signal pairs, nor does it provide connection housings or headers. It simply provides direct vias to the FPGA chip.

The consequence of this design is that its original engineers had to create custom machined parts to form a make-shift plastic housing; this connector holds the female 100-mil connections that mate with the soldered-on male counterparts. While these worked initially, over time they became unreliable and were inconsistently making

contact.

Yet another weakness in the system was the cabling from the FPGA board to the translator board, and the subsequent translator board electronics. On the FPGA side, the original designers faced many issues with the housing and contact points as mentioned above. Additionally, the signal translator chips were not wired to handle every electrical scenario, and would short (and get damaged), often requiring a replacement. The researchers eventually tracked the problem down to an incorrect power-up sequence; for the sake of robustness, I have addressed this in the next generation embedded system.

1.3 Moving to an Embedded System

Researchers at SCIPP are currently using the original PTSM prototype to run experiments. The system, however, has proved to be unstable, unreliable, and riddled with technical problems. The issues are entirely in the implementation domain; that is, when all of the components are working as they are supposed to, the experimental data is sound (and has even led to some publications).

When the system isn't working, however, the energy and effort invested into debugging it has motivated the upgrade to a more robust and versatile solution. We made the decision to move towards a reconfigurable embedded system – one that would intrinsically and by design eliminate many of the points of failure.

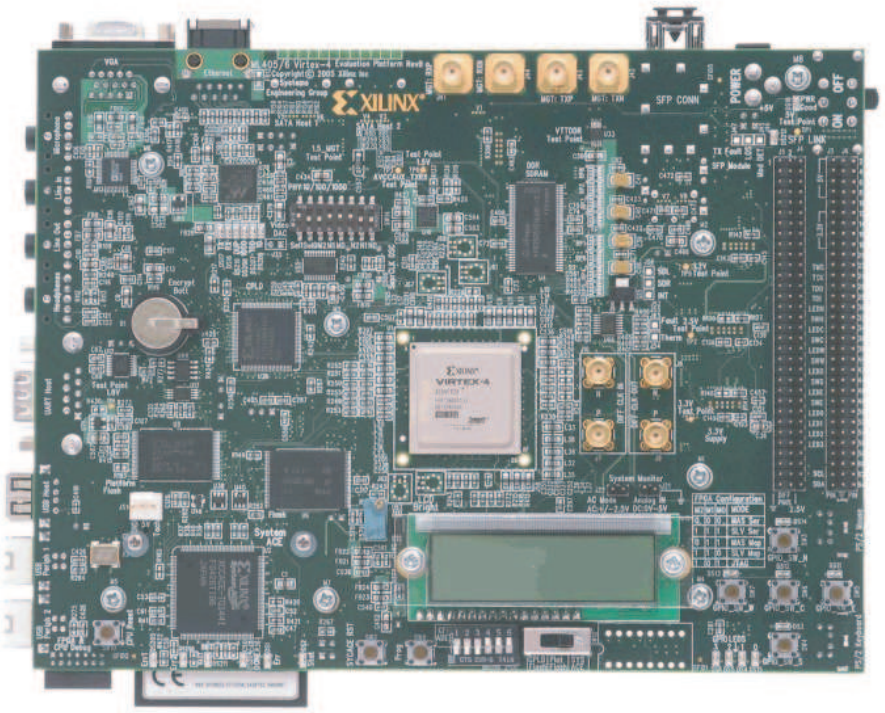

Figure 1.1: Xilinx ML405 Embedded System

Embedded System Advantages

The new system consists of the same front-end board as before that now feeds directly into the Xilinx ML405 reconfigurable embedded system. This development board has 16 differential inputs with mounted 100-mil headers, ideal for LVDS signals.

As far as cables go, there is precisely one custom data cable in the entire setup, and it goes from a standard D-sub connection on the front-end side to the headers on the ML405 just mentioned. This allows us to use off-the-shelf parts that are both structurally sound and transient tested – that is, they have been tested in the field for several decades now.

To get the data from the ML405 into the host PC, we use a standard RJ-45

ethernet jack and cable. The on-board ethernet MAC/PHY chip automatically detects a crossover connection, thereby obviating the need for a special crossover CAT5e/CAT6e cable.

Furthermore, both the Virtex-4 FPGA on the ML405 and its MAC/PHY chip have Gigabit ethernet capability. This functionality is not in the current requirements, and is therefore not implemented — it is, however, available as an option in the future. See section 1.4.2 for information regarding bandwidth.

1.4 Looking Ahead

1.4.1 Short-term Goals

As far as the embedded system goes, I hope to have it run experiments for the next two years without any engineering problems. The learning curve for debugging such a system is steep, and as a result, I have made every attempt to make troubleshooting a deterministic and simple task. Unfortunately, the nature of this project (and engineering in general) precludes me from designing a perfect system.

To accommodate further bug fixes after I leave, I have annotated all of the source code with comments. Furthermore, I've attempted to describe the technical details (such as detailing the timing requirements through timing diagrams, and listing the many non-obvious constraints in interfacing to the provided bus) as best as I can.

1.4.2 Long-term Goals

Given the Xilinx ML405's Gigabit ethernet capabilities (allowing practical data rates exceeding 200 Mbps), new higher-bandwidth applications in medical imaging applications could be realized. That is, if someone were to put in the energy, the basic foundation of such a system has already been implemented. As it stands, increasing the current bandwidth of the board (which is approximately 2 Mbps over a 100 Mbit crossover ethernet cable) would be a matter of: 1) rearranging the memory layout such that the internal caching schemes in the embedded CPU could be exploited; 2) redesigning the software readout system that manages the movement of processed data into memory, and then from the memory into the ethernet controller to be sent out; and 3) fine-tuning and exploiting the low-level features of the open-source lwIP TCP/IP stack used.

Chapter 2

EPTSM Overview

The Embedded PTSM system is comprised of four sub-systems: the detector/read-out chip, the digital back-end electronics, the scintillator and photomultiplier tube, and the host computer.

2.1 Flow of Data

The overall goal of the project, as mentioned earlier, is to characterize detectors. In order to do this, we use a controlled radiation source that precisely emits electrons conforming to a known Gaussian distribution. This source is placed directly underneath the detector such that the data that it reads out can be fitted against a function from the same class as the expected Gaussian distribution.

In order to separate actual particle hits from electrical noise, there is a scintillator mounted directly above the detector. Electrons from the radiation source pass through the detector (depositing a charge) and are absorbed in the scintillator. The scintillator's primary purpose is to turn the absorbed electron into a photon, which

is then detected by a photomultiplier tube (PMT) and read out as if it were one of the channels on the silicon detector.

2.2 Breakdown of the Components

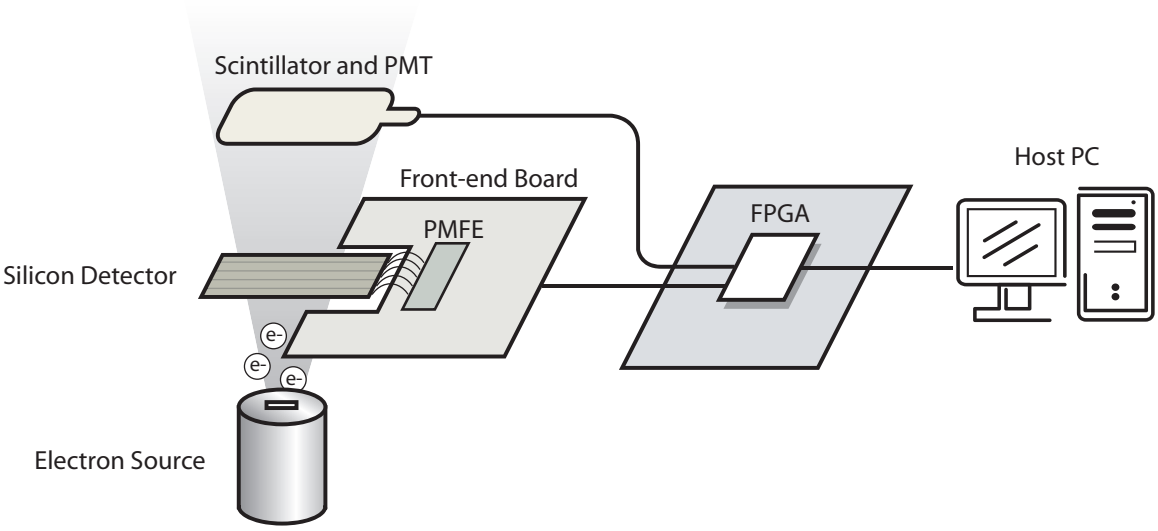

Figure 2.1: Overall system layout, including the electron source

The aforementioned subsystems' components will now be described. The detector/read-out chip subsystem consists of a target silicon strip detector for characterization, and an ASIC to convert the small charges deposited onto the detector into a usable electrical signal. This signal is then digitized and the digital back-end subsystem processes it; that is, the signals feed into a reconfigurable embedded system which subsequently tracks and timestamps any transitions. Additionally, an independent scintillator/photomultiplier tube pair feeds into the back-end electronics for reasons clarified below.

2.2.1 Silicon Detectors

The silicon strip detectors consist, as the name implies, of large strips of doped silicon. When a particle passes through one of the strips, it deposits a small charge by triggering energy band changes within the crystal lattice (in essence, ionizing that particular strip). The detectors must be biased during experiments in order for this desired effect to occur.

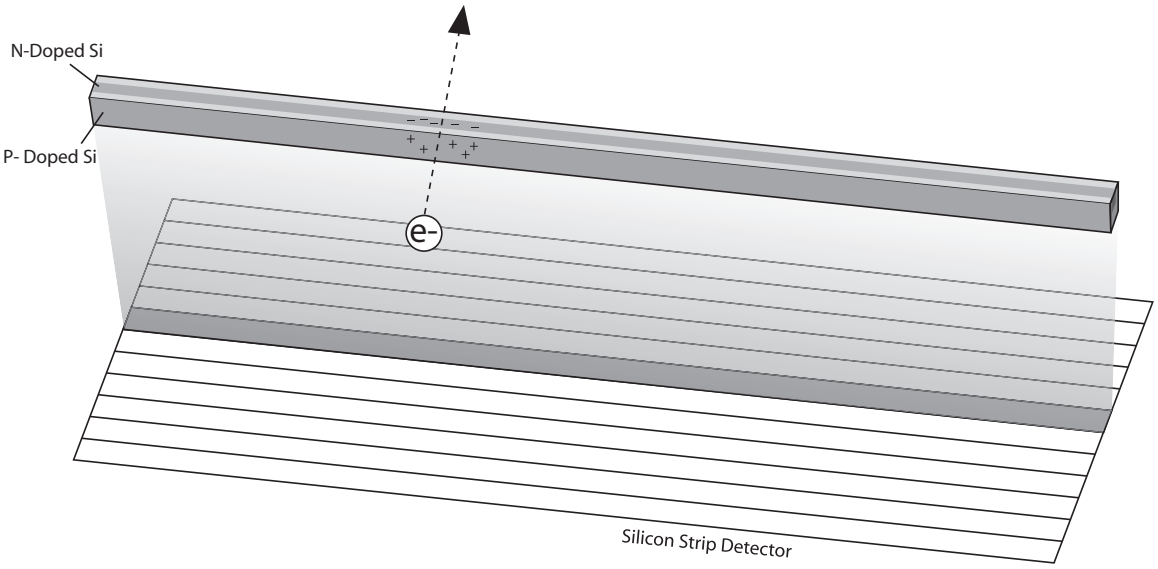

Figure 2.2: An electron depositing charge onto a single strip on an N-type silicon detector

Each one of these strips is referred to as a detector channel, and they are referred to by their channel ID (which is simply a number assigned sequentially). We are able to determine where a channel is located based on this number. See Figure 2.2.

2.2.2 The Particle Microscope Front-End (PMFE) Chip

In order to detect the deposited charge on the detector (which are typically on the order of a few femtocoulombs), we use an ASIC designed to amplify and digitize its input. This chip is called the Particle Microscope Front-End (PMFE) chip — internally, it has a charge amplifier, comparator, and a serializer. It was designed at SCIPP by Edwin Spencer.

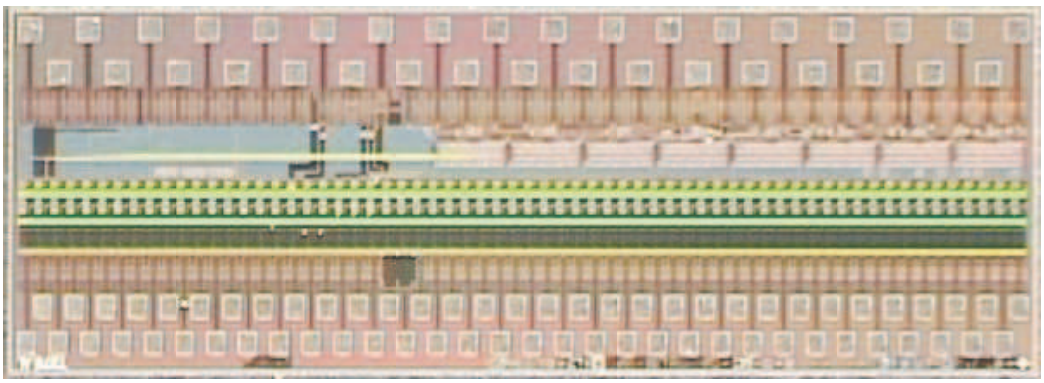

Figure 2.3: The Particle Microscope Front-End Chip (PMFE)

The PMFE must be sensitive to a very small charge, and as such, all electrical noise and interference must be minimized (if not eliminated altogether). To this end, the clock signals that drive the serializer logic must come from an external, off-board source. Crystal oscillators tend to generate too much noise, and there are few (if any) viably cheap alternatives. Thus, the FPGA generates the required clock signals and transmits it to the PMFE.

One feature of the PMFE is its ability to allow the user to calibrate the chip. The PMFE has an input signal that feeds directly into the inputs of the signal conditioning chain (that is, the aforementioned charge amplifier, comparator, and serializer). This

allows us to emulate a deposited charge in order to deterministically test both the chip's internal current biasing, any issues with the bonding of the silicon detector to the board, and the digital back-end electronics.

Currently, we can test the PMFE by wiring its calibration input to a pulse generator to emulate what a fixed number of pulses would look like on the back end. If we generate 100 pulses, we expect to see 100 detector hits read out. The host computer starts the data acquisition on the FPGA, which controls the pulse generator. The FPGA then reads out the data to the host computer, allowing the user to determine whether or not all of the synthetic hits where registered.

The internal comparators act as a data quantizer, taking the voltage input from the charge amplifier and discretizing it. Varying detectors, however, register different voltage levels at this amplification stage; as a result, we cannot have a fixed threshold. The PMFE provides a threshold voltage input that the experimenter changes based on what he or she knows about the detector.

2.2.3 Front-End Board

The Front-End Board acts as a host to the PMFE and its requisite electronics. It additionally provides the housing for the silicon detector, and a hole to allow the electrons to hit the detector directly. The electronics that support the PMFE are largely related to the threshold inputs and its internal current biasing (these are one-time potentiometer configurations that bias the internal transistors). The board additionally handles all of the local signal routing and power filtering.

2.2.4 Scintillator, PMT, and Coincidences

The scintillator absorbs particles of a certain size, then emits a photon. This photon is detected by the photomultiplier tube (PMT), which turns it into a signal that we can plug directly into the back-end electronics as if it were a regular detector channel.

The primary goal here is to eliminate any noise hits – that is, induced electrical fields from the outlying electronics or environment which register as false particle hits. The software package will only accept hits on the other detector channels if they coincide with the PMT hits – these are referred to as coincidences.

2.2.5 Signaling and Signal Standards

We use the Low-Voltage Differential Signaling standard wherever possible to ensure reliable and low-noise transmission of data. The PMFE has internal LVDS drivers and receivers for all of its signals – specifically, both input clocks and all eight data signals. On the FPGA side, the Xilinx Virtex-4 chip has internal LVDS receivers; moreover, the ML405 development board has 16 designated general-purpose LVDS inputs. They are specifically connected to the FPGA's LVDS receivers, and the traces are optimized for differential pairs.

2.2.6 Summary of the Digital Back-end Electronics

The primary duty of the digital back-end electronics is to record every transition in the channels that it receives and timestamp when these transitions occur. This job is relegated to the embedded system – the digital logic that handles this task is programmed into the FPGA fabric and communicates over the local bus to the hard

CPU within the FPGA.

This subsystem is detailed in the next chapter.

Chapter 3

Digital Back-End

3.1 Overview

The digital back-end electronics is defined as the embedded system and host PC coupled together to process the raw channel information (that is, whether or not there is charge on said channel) into formats that can then be used to deduce physical characteristics. Within this set, the digital back-end logic is defined as the custom design implemented in the FPGA that performs the primary task of monitoring the channel states and recording any changes.

3.1.1 FPGA Processing

The FPGA embedded system has the low-level processing and buffering required to sustain our desired bandwidth. It timestamps each individual channel's transitions, giving the host computer information about which channels changed state, what state they changed into, and when this happened (relative to the local clock on the

FPGA). The host PC runs this information through the ROOT data analysis package to generate graphs and fit functions.

In addition to timestamping the channel transitions, the back-end also contains glue logic to coordinate various components. This includes calibration pulse generation, inhibition of acquisition, and controlling the voltage/current sources via GPIB.

3.1.2 The Host Machine

Once the embedded system collects the data, it reads it out to the host computer over a 10/100 ethernet connection. The software on the embedded system runs a TCP/IP server that the client software on the host Windows or Linux machine connects to. The client sends control commands such as those to start and stop data acquisition and mask out certain channels.

3.1.3 Data Analysis

The client software that runs on the host PC runs the data through the open-source ROOT analysis package provided and maintained by CERN. This software allows us to generate various histograms of the hit distribution, then fit appropriate graphs to them.

The timestamp information that is recorded with each hit allows researchers to determine the time-over-threshold (TOT), which essentially gives them a relatively good approximation of the magnitude of the charge deposited.

3.2 Implementation

3.2.1 Serialized Data and Clock Signals

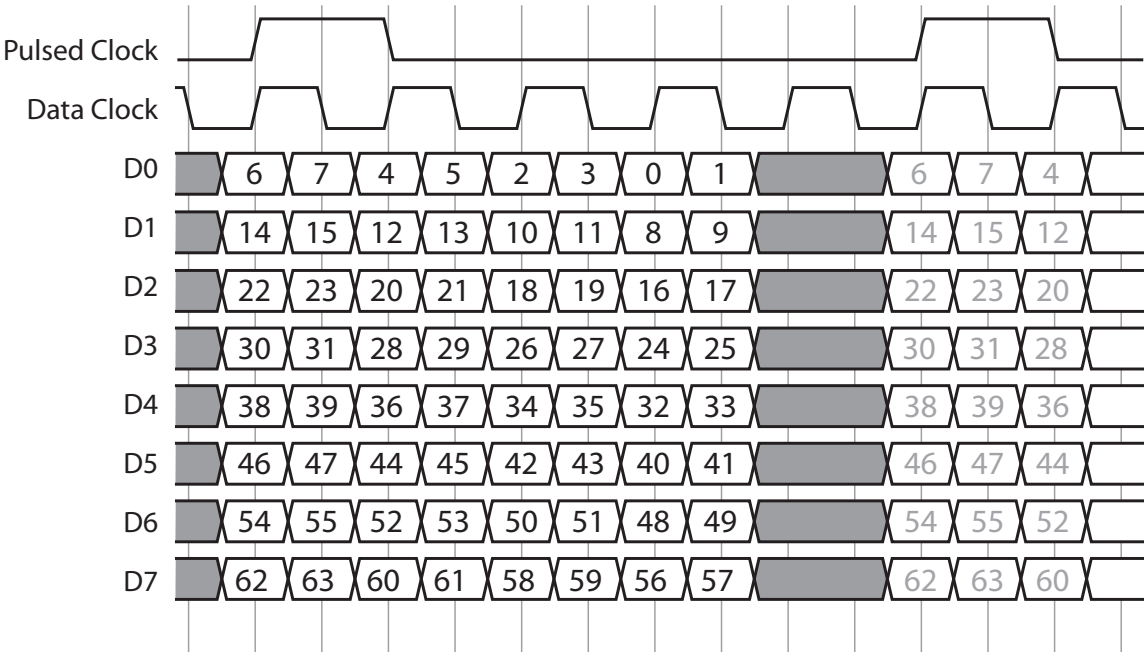

Figure 3.1: Main clock signals and data

The PMFE sends channel state information through eight wires, each containing eight serialized bits, for a total of 64 bits. Each bit corresponds to that specific channel's state – that is, it is high so long as charge is detected on the strip.

Due to noise constraints, the PMFE cannot generate its own clock signals to do the serialization. It instead relies on the FPGA to generate these signals using its internal PLLs. To serialize data, the PMFE needs two pieces of information: the clock edge to send bits out on, and the clock edge to synchronize each packet of eight bits. These are known as the data clock and pulsed clock, respectively. The former

runs at 50 MHz on both edges (yielding an effective bit rate of 100 MHz), while the latter is high every fifth data clock for the duration of its period (that is, it is high for one of the 50 MHz clocks, and low for four of them).

One added benefit of generating these clocks on the FPGA is that these same PLLs can be used to drive the internal deserializer on the receiving end. The only change that needs to happen is the phase must be adjusted to account for the round-trip delays intrinsic in such a design.

3.2.2 The Channel Servers

Brian Keeney's original design implemented the timestamping feature by allocating what he called a channel server for groups of channels – this design was ported over to the embedded system. The goal is twofold: first, the problem space is subdivided in an elegant manner by assigning each channel server with a unique group of channels; second, the size of the groups are chosen to exploit the existing clock structure. For the former goal, such a subdivision negates the need for individual logic and buffers for each channel, which would inadequately allocate resources. In the latter case, the two clocks (the data clock and the pulsed clock) are used to clock and control the state machines that check for transitions.

The pulsed clock goes high every five data clocks. This allowed Brian to have each channel server monitor four channels each. On every data clock, the state machine progresses to the next channel in its group, and synchronizes on every pulsed clock. Groups of four channels per channel server make a total of 16 channel servers – a managable and reasonable balance of FPGA resources.

The channel servers constantly check for changes in channel state, and record those changes to a small, local first-in first-out (FIFO) data structure. These independent buffers need a state machine to consolidate all of the transition information into one single buffer to be exported into the software realm on the embedded system. The FIFO server handles this task.

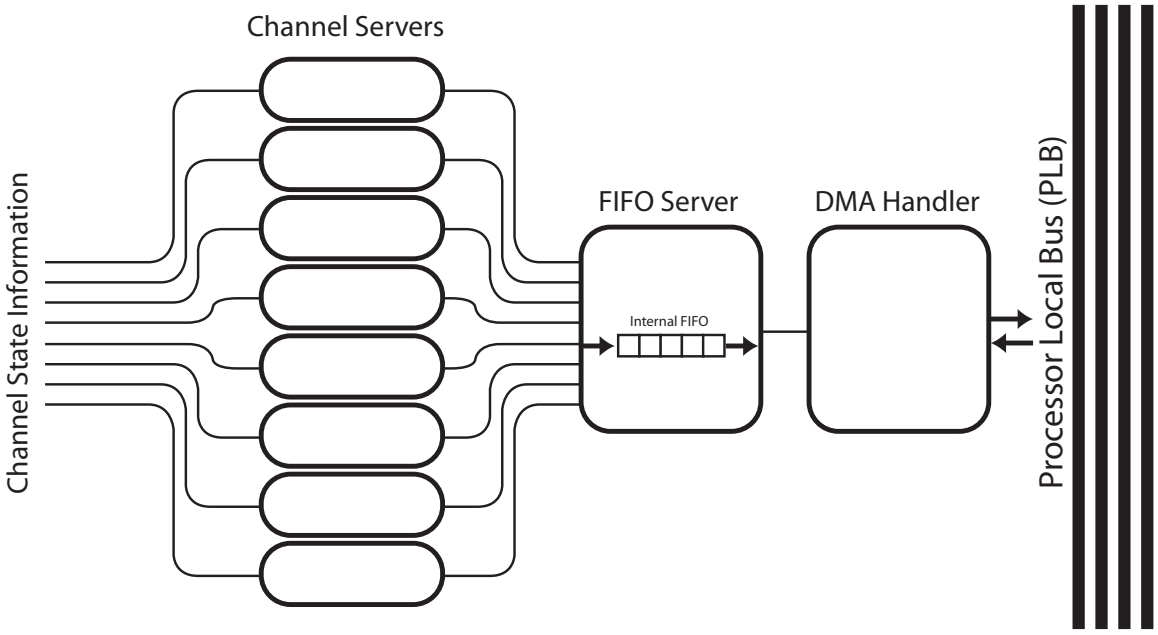

Figure 3.2: The PTSM peripheral flow of data

3.2.3 The FIFO Server

The FIFO server's job is to collect data from each of the channel servers and feed them into a single larger FIFO. Additionally, it contains logic to prioritize those channel servers that need to be emptied first. If none of the channel servers' FIFOs are beyond half of their capacity, the FIFO server's state machine simply cycles through them

incrementally, removing one item at a time. If there is at least one channel server whose FIFO exceeds half of its depth, the FIFO server will empty qualifying channel servers out entirely. The reasoning behind this decision is that the channel servers which are more occupied are more likely to produce future channel hits.

The order of priority is only to one degree – that is, the decision to empty a channel server is based solely on whether or not it is at least half full. If all of the channel servers are in this state, then all channels are given equal priority, and are thus emptied out sequentially. Since they are all considered peers of equal weight, the metric beyond FIFO capacity is an arbitrary one.

3.2.4 Continuity of Data

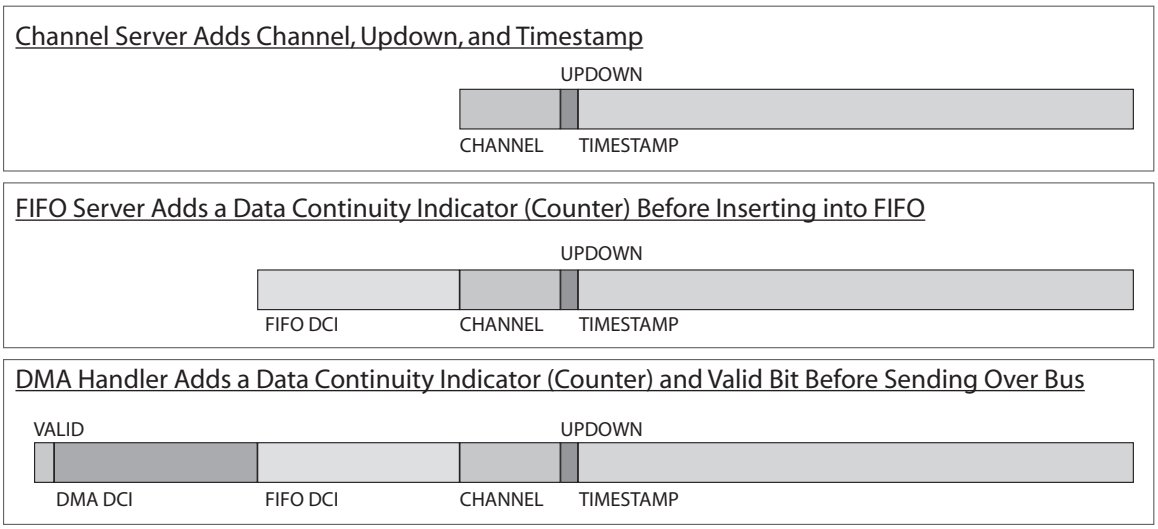

Figure 3.3: Construction of a 64-bit Data Packet

From the channel servers to the host machine, there are many potential points of data loss due to timing and design errors. Often, throughout the project, I discovered

that a register latching on the wrong clock cycle could result in data loss. To ensure that I was correctly transmitting the data, I included counters that were embedded in the 64 bits – these counters are referred to as data continuity indicators (DCI).

The counters serve two primary purposes. First, they have allowed me to debug the aforementioned dropped data issue. Second, if there is no data to read out, but the FPGA reads out the data anyway, it becomes easy to eliminate duplicates. Thus, they serve to both eliminate extra data, and notify the user of missing data.

Two counters are part of each packet: the FIFO DCI counter, which gets incremented before data enters the primary FIFO (in the FIFO server), and the DMA DCI counter, which is incremented on each clock of a DMA transaction. These are often off by a constant value, since we sometimes read more data than is in the buffer (although it should be noted that no errors will occur in this situation. Too much data is not a problem so long as it can be identified as such).

Chapter 4

Embedded System

4.1 Hardware

4.1.1 DMA Transactions

The custom PTSM peripheral sits on the Processor Local Bus (PLB), and copies data into the software domain via direct memory access (DMA) transactions. DMA transactions require a master peripheral and a slave peripheral, and allow large amounts of data to be copied between two devices without any CPU intervention. Xilinx's Intellectual Property InterFace (IPIF), detailed in appendix A, provides DMA capabilities by making the peripheral a bus master. This allows the DMA portion of the code to take control of the bus and copy data from one slave to another. DMA master capability is integrated into the peripheral to allow for fast transactions to the target destination. It's faster to read from the local peripheral (without any bus transactions), and then write directly to the bus.

The data to be transferred over the bus is stored in the FIFO located in the FIFO

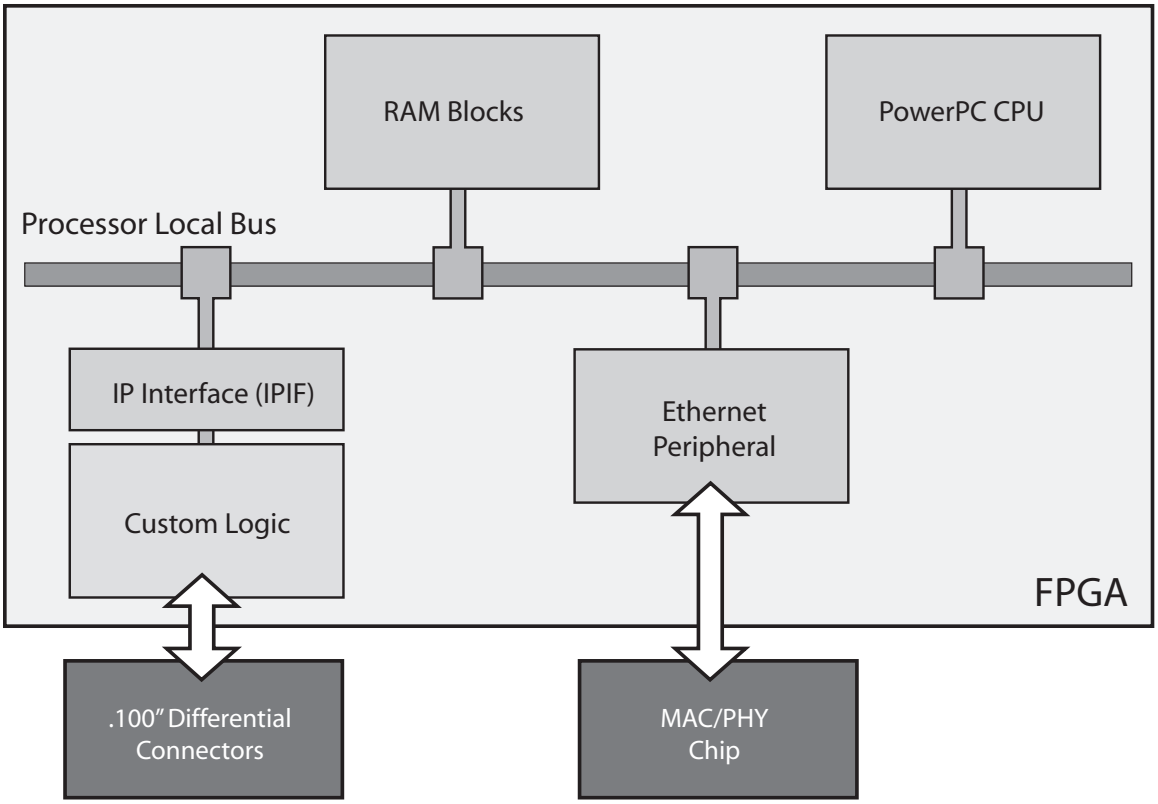

Figure 4.1: FPGA Embedded System Block Diagram

server module of the digital back-end logic. To get this data out of the peripheral, software must instruct the local DMA master to perform a burst-read transaction (that is, multiple words in one continuous transaction) to a memory device. On the peripheral side, this appears as a read instruction with a specific burst input signal set high for as many clock cycles as it is attempting to read. On this system, reads are done in 16-word bursts, so the user IP can expect this signal to remain high for 16 bus clock cycles. According to the specifications of the IPIF, however, we cannot anticipate this to be consistent with future versions. Therefore, no assumptions about the size of each burst transfer are made. The data must appear on the bus so long as this burst and acknowledge signals remain asserted, where each individual 64-bit

word is read out on the rising edge of the clock.

4.1.2 Software Registers

The easiest and simplest way to have the embedded software control and communicate with the EPTSM peripheral is through the IPIF's software register feature. This feature provides the software side with registers that it can read or write to. The hardware can then determine how to allocate and react to these I/O operations.

For this project, I've used the software registers to: start and stop data acquisition; set the calibration mode and type; allow the software to write the channel mask data (to mask out specific channels); read parallelized raw channel data for debugging/troubleshooting; and read how many channel transitions were lost due to bandwidth and latency issues.

Implementing this feature was a matter of modifying the existing sample Verilog code and abstracting the bits into logical bus names. This was done in the Register Handler module, whose I/O consisted of two sets: the logically named buses that connected to other modules, and the IPIF bus interaction signals.

4.2 Software

4.2.1 Coordinating the DMA

From the software side, DMA transactions are coordinated by the CPU by writing to special registers in the peripheral's I/O memory. The three registers of interest are the source address, destination address, and transaction length (in words). The

source address is the peripheral's base address, while the destination is the raw base address of the on-chip RAM blocks. These are not used by any other part of the design, and are ideal because they are both directly connected to the PLB bus, and tolerate 64-bit-wide transactions.

The DMA master notifies the software when the transaction is complete via an interrupt. Therefore, we register an interrupt handler with the on-system interrupt controller and its driver, and enable the specific DMA 'done' bit. As will be detailed below, the software is multithreaded. This affords us the use of IPC functions – more specifically, we can use semaphores to allow the interrupt handler to wake up a process that is sleeping and waiting for the DMA transaction to complete.

4.2.2 The Network Stack

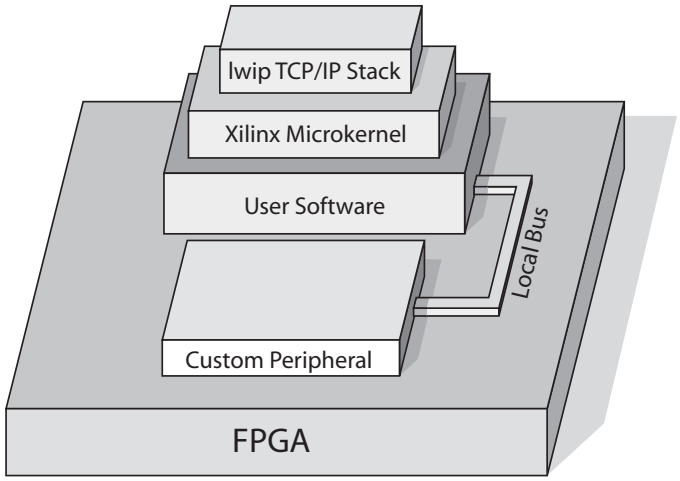

Figure 4.2: Software Layers

The network stack is implemented using Xilinx's port of the open-source Light-Weight IP (lwIP) library. The library comes pre-configured to work with Xilinx's

OPB Ethernet core, eliminating any initial overhead. It features a fully implemented TCP/IP stack that can be accessed using the standard BSD socket interface for server functionality.

The embedded system thus acts as a TCP/IP server, while the host PC's software is the TCP/IP client. I've kept the system simple and easy to debug by using a single cross-over ethernet cable – putting only two nodes on the network removes the chance of collisions and chatter from nodes we are not interested in.

4.2.3 Multithreading

Xilinx provides a microkernel to use with their chips that provides several POSIX services such as pthreads, processes, IPC, and dynamic memory allocation. Support for pthread allows us to set up a multithreaded network application where the control logic lies in one thread, and each client connection (and its subsequent code to send data out) lies in another. Additionally, the lwIP library requires several of these features to run – it relies on the internal mailbox feature to handle buffering packets.

Another benefit of using the Xilinx Microkernel is its IPC functions for situations where a thread is waiting for data (such as the case presented above where a specific thread may sleep until a particular DMA transaction is completed).

4.2.4 Integrating the Modules

To ship data out of the board, the software must first wait for a client's connection. Once this is received, and the client instructs the embedded system to send it data, it attempts to do so as fast as possible. The software must start a DMA transaction,

then wait for it to complete. However, while it is waiting, it uses the CPU to transmit data over the network in order to maximize the CPU's time.

This is accomplished through two buffers. It first triggers a DMA transaction, then immediately sends out a pending data buffer. Once this data buffer has been transmitted, the thread sleeps until the DMA transaction is completed. In practice, the thread typically does not sleep more than a handful of clock cycles, since a large portion of the DMA transaction time overlaps with the TCP/IP stack sending data out.

When the system is running as it is designed to, it's able to perform these tasks at a rate of approximately 2 Mbps. Most of the CPU time is spent copying buffers from the block RAMs, into lwIP's buffer pool, and then back over the bus to the ethernet peripheral. In the current implementation, only the PTSM peripheral uses DMA. Future versions will both reduce the amount of copying done, and rely more on DMA (i.e. directly between memory buffers and the ethernet peripheral).

Chapter 5

Challenges

The challenges in this project have been bountiful. For the most part, they stemmed from the steep learning curve in these embedded systems, coupled with the fact that I was working primarily on my own with very little knowledge. I spent quite a bit of time attempting to understand the technology based on technical documentation — this was a challenge due to the documentation's assumptions of knowledge that I didn't have.

To add to the complexity, the technology was relatively new when I started. This meant that there were often issues related to inadequate documentation, or yet-to-be-implemented features in software (namely those that we required). In particular, there was no Verilog support in the embedded system development suite. All of our prior designs were in Verilog, and porting it to VHDL was not worth the effort at the time.

The project first began as an attempt to implement the embedded system on the Xilinx ML310 board. The board had just been released at the time (it was a mere 2

months out of its first board run), so comprehensive documentation wasn't initially available. Eventually, we moved away from this board because getting data off of it required going over the local PCI bus. We deemed this a waste of energy, since we could easily accomplish this requirement via a MAC/PHY directly connected to the FPGA (on the ML310, it went over the PCI bus to a PCI MAC/PHY slave).

I believe this project would have been completed much earlier had I been privy to the debugging tools that Xilinx and other vendors offer. I eventually learned to use ChipScope to analyze real-time signals in the FPGA – this would have saved me time attempting to debug bus transaction timing issues. Later, we acquired a license for the IBM Bus Functional Model (BFM) toolkit, which allows developers to simulate various bus transactions without having to simulate the entire logic behind it. The advantage here was speed and interface simplicity. Finally, while there were several obstacles in getting it working, I learned to simulate the design in Modelsim. This was especially beneficial in monitoring my peripheral's interaction with its peers on the bus.

Bibliography

[1] IBM. *IBM CoreConnect Architecture 2.0*, 2005.

[2] B. Keeney. The design, implementation, and characterization of a prototype readout system for the particle tracking silicon microscope. Master's thesis, University of California, Santa Cruz, 2004.

[3] G. Lindstrom and et al. Radiation hard silicon detectors–developments by the rd48 (rose) collaboration. *Nuclear Instruments and Methods in Physics Research*, Volume 466(2):308–326, 2001.

[4] H. Sadrozinski, B. Keeney, and et al. The particle tracking silicon microscope. *IEEE Transactions on Nuclear Science*, 51(5):2032–2036, 2004.

Appendix A

Interfacing to the OPB/PLB System Bus

A.1 Introduction

The PowerPC processor embedded in the FPGA fabric communicates with peripherals via one of two main buses (both of which are part of IBM's CoreConnect technology): the On-Chip Peripheral Bus (OPB) and the Processor Local Bus (PLB).

The differences between the two are documented in detail in the CoreConnect documentation [1]. To summarize, the PLB allows for high-speed transfers directly to the CPU. Devices attached to this bus are typically memory devices and any data-intensive peripheral. The OPB, on the other hand, is slightly more flexible, and better suited for general-purpose peripherals. Examples include interrupt controllers, ethernet devices, UART interfaces, and bridges to other buses (such as USB, PCI, or IIC).

Additionally, for the Xilinx implementation, the PLB is natively 64 bits wide, while the OPB is natively 32 bits wide.

A.2 Connecting to a Xilinx Chip's Bus

Xilinx has made interfacing to either of the two buses incredibly simple via the IPIF (Intellectual Property InterFace). The IP core that ships with the Embedded Development Kit (EDK) comes in two flavors, one for each of the aforementioned buses.

The IPIF provides various services to the user, including (but not limited to): slave or master operation, and an abstract layer thereof; DMA and DMA/Scatter-Gather control; read and write FIFOs; user interrupt service; software-accessible registers; and multiple memory addresses. These features are simplified and summarized below. Details can be found in Xilinx's documentation.

Slave or master operation simply refers to the IPIF's ability to act as a bus slave or bus master, and providing the user (the developer) simplified signals. The user does not need to worry about tri-stating or any specific details beyond the basic request-acknowledge signals that are provided. The user should, however, know what all of the signals are for. For example, the Byte Enable (BE) bus is critical for all transactions. To summarize, the PLB transactions will often be smaller than 64 bits wide, and as such, the standard provides the BE bus where each bit represents one byte. Since the bus is 64 bits wide, and there are 8 bits in a bytes, this means that the BE bus is 8 bits wide. When writing, this is important; when reading, it doesn't particularly matter.

DMA and DMA/Scatter Gather are provided via the Bus2IP_Burst signal. The

signal will go high, and data is expected on the next clock cycle. Once it goes low, the next clock cycle will be the last word registered and posted to bus. Scatter Gather functionality is abstracted on the IPIF side.

Read and write FIFOs are ideal for projects that need to share large amounts of data with the software in a stream fashion (such as ethernet devices). They can be coupled with DMA to allow direct writing and reading to memory devices.

The user interrupt service provides the IPIF user logic with a signal input which the logic can assert to trigger an interrupt. The IPIF provides various sensitivities (level high, level low, rising edge, and falling edge). The user must connect this interrupt signal either directly into the CPU's interrupt input port, or (more likely) to an interrupt controller peripheral. Xilinx provides the OPB and PLB INTC device to perform this function (within which priorities are assigned to interrupts).

Software-accessible registers are typically used for software configuration of a hardware device. On the user-logic side, the input data bus will assert an input on a write transaction, and the output data bus can be asserted with a register's contents on a read transaction. Note that read and write are set from the software's point of view.

A.3 Debugging The IP

Debugging a custom peripheral on a bus can get tricky. The most efficient way to troubleshoot is via the ChipScope software suite. There are two primary means of using ChipScope: the first would be to use the PLB Integrated Bus Analyzer (IBA), which will automatically provide ChipScope with data, address, and control buses.

For debugging the internals of the user logic, however, the only sensible approach

is by exporting the desired signals as outputs of your peripheral. This means that deeply nested signals must traverse the heirarchy (output to output) until they get to the IPIF interface. From there, the signals need to be designated as device outputs, and on the embedded system layer, they can then be connected to a generic ChipScope peripheral.